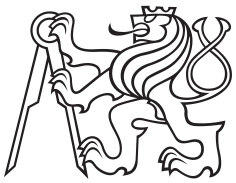
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of radio engineering

Physical Layer TestBed for Communication V2X Systems in 5.9GHz Band

Bc. Michael Kimmer

Supervisor: prof. Ing. Jan Sýkora, CSc.
May 2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kimmer** Jméno: **Michael** Osobní číslo: **491909**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra radioelektroniky**

Studijní program: **Elektronika a komunikace**

Specializace: **Komunikace a zpracování informace**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Laboratorní TestBed fyzické vrstvy V2X komunikačního systému v pásmu 5.9GHz

Název diplomové práce anglicky:

Physical Layer TestBed for Communication V2X Systems in 5.9GHz Band

Pokyny pro vypracování:

Cílem práce je návrh a realizace laboratorního, testovacího a verifikačního systému pro implementace rádiového rozhraní V2X komunikace - modulace, kódování, algoritmy M-MIMO, zpracování signálu, kooperativní algoritmy potlačení interference a diversity a další pokročilé algoritmy, např. Joint Communication and Sensing, kooperativní kódování, atd. V základní části student implementuje PHY vrstvu komunikací třídy IEEE 802.11p, s přípravou pro budoucí rozšíření na 5G NR C-V2V. TestBed umožní (a) verifikaci celkové funkce a srovnávací výkonnostní a provozní testy, (b) evaluaci vlastností dílčích subsystémů a ověřování jejich možných výzkumných/vývojových vylepšení a poslouží jako základ pro vývoj prototypu a rovněž pro výzkumně-vývojové práce. Výchozí HW platforma je ADRV9002 a Xilinx ZedBoard doplněná o další RF, baseband a řídicí komponenty.

Seznam doporučené literatury:

[1] ADRV9002 eval board (3-6GHz version) [online] https://www.analog.com/en/products/adrv9002.html#product-reference
[2] ZedBoard tech resources [online]
https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/zedboard/
[3] Specifikace IEEE 801.11p

Jméno a pracoviště vedoucí(ho) diplomové práce:

prof. Ing. Jan Sýkora, CSc. katedra radioelektroniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **05.02.2024** Termín odevzdání diplomové práce: **24.05.2024**

Platnost zadání diplomové práce: **21.09.2025**

prof. Ing. Jan Sýkora, CSc. podpis vedoucí(ho) práce	doc. Ing. Stanislav Vítek, Ph.D. podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. | Podpis studenta

Datum převzetí zadání

Acknowledgements

I would like to thank my supervisor, prof. Ing. Jan Sýkora, CSc. for his guidance and valuable advice. Next, I would like to thank my family, and my friends from CB Dejvice for their constant prayers and encouragements.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, May 24, 2024

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 24. 5. 2024

. . . . . .

Abstract

The goal of this thesis is to get acquainted with the IEEE 802.11p standard and subsequently to create a custom receiver which is able to process the signal in real-time. This receiver should be then ready for further improvements. The chosen method is an FPGA block design. The used hardware is ZedBoard and EVAL-ADRV9002.

Keywords: digital communication, V2X, IEEE 802.11p, FPGA, SoC, VHDL, AXI4, ZedBoard, SDR, ADRV9001, Libiio, Vivado and Vitis, scrambling, convolutional code, interleaving, OFDM, synchronization, channel response estimation, Viterbi algorithm, Python GUI

Supervisor: prof. Ing. Jan Sýkora, CSc.

Abstrakt

Cílem této práce je seznámit se se standardem IEEE 802.11p a následně implementovat vlastní přijímač, schopný real-time příjmu. Tento přijímač má být možné v budoucnu rozšiřovat. Vybraná metoda je vytvořit FPGA blokový design. Použitý hardware je ZedBoard a EVAL-ADRV9002.

Klíčová slova: digitální komunikace, V2X, IEEE 802.11p, FPGA, SoC, VHDL, AXI4, ZedBoard, SDR, ADRV9001, Libiio, Vivado and Vitis, scrambling, konvoluční kód, interleaving, OFDM, synchronizace, odhad odezvy kanálu, Viterbiho algoritmus, Python GUI

Překlad názvu: Laboratorní TestBed fyzické vrstvy V2X komunikačního systému v pásmu 5.9GHz

Contents

Figures

Tables

Assignment translation

The goal of the work is to design and implement a laboratory, testing and verification system for radio interface implementations of V2X communication - modulation, coding, M-MIMO algorithms, signal processing, cooperative interference suppression algorithms and diversity and other advanced algorithms, e.g. Joint Communication and Sensing, cooperative coding, etc.

In the primary part, the student will implement the PHY layer of IEEE 802.11p communication class, with preparation for future expansion to 5G NR C-V2V. The TestBed will allow (a) verification of the overall function and comparative performance and operational tests, (b) evaluation of the properties of the sub-systems and verification of their possible research and development improvements and it will serve as a basis for prototype development as well as research and development work.

The default HW platform is ADRV9002 and Xilinx ZedBoard supplemented with additional RF, baseband and control components.

Chapter 1

Introduction

There is a trend in automotive industry to develop smarter vehicles every year, in recent past many companies were also experimenting with self-driving technologies. For such vehicles it is essential to have as much complete picture of what's happening around them as it is possible. The V2X communication offers one way of reliable, fast and cheap communication among such vehicles and surrounding infrastructure.

For a fast and reliable communication advanced algorithms have to be often used. Our work should bring us a workflow with a working prototype of a receiver on a real hardware. This prototype should be able to get improved in the future to test different approaches without the need of building everything from the ground.

The biggest challenge is to meet all timing requirements of the assigned standard IEEE 802.11p, we will get to know this standard in Chapter 2. The options are either, first, to build a receiver with a CPU, or second, more challenging but also more powerful, is to build a receiver with an FPGA. This latter method will be content of whole Chapter 4, while Chapter 5 will make a way how to access the transceiver via custom GUI.

Chapter 2

IEEE 802.11p standard

2.1 V2X Communication theory

The term V2X (vehicle to everything) communication can involve many communication situations. Let us denote the most common situations which are V2V (vehicle to vehicle), V2I (vehicle to infrastructure) and V2P (vehicle to pedestrian).

The most important contributions of this type of communication emerge from its direct character enabling ad-hoc and mesh network architectures. It can work in areas with no mobile network coverage and offers lower latency. V2X communication protocols are furthermore designed in a way that improve reliability and low latency in quickly changing channels present around fast-moving with Doppler spread. [10, p. 1166]

A sample example of V2X communication could be a pair of vehicles exchanging information about speed, emergency braking or approaching ambulance. Another example could be a tram or a police car requesting a traffic light to prioritize their direction. [1]

These principles have a potential to increase road safety, prevent accidents, reduce emissions or optimize traffic flow. [2]

To illustrate, some existing standards are: Cellular V2X (LTE-V2X, 5G-V2X), IEEE 802.11p, or recent IEEE 802.11bd.

2.2 IEEE 802.11p standard introduction

IEEE 802.11p is an amendment for IEEE 802.11 standards (also called standard) from 2010. It is focused on Dedicated short-range communications (DSCR) especially on vehicular environment. Most of changes are defined for the MAC layer to enable efficient communication setup without much of the overhead typically needed. [11]

2.3 IEEE 802.11p MAC layer

The IEEE 802.11p Medium access control (MAC) layer serves for management of the wireless medium. It is responsible for packet transmission timing, collision avoidance (CSMA/CA), power selection, data rate (that is code rate and modulation) selection and packet integrity check (CRC).

Used frequency range is 5855 - 5925 MHz in Europe and 5850 - 5925 MHz in USA. This range is divided into channels of 10 MHz. Maximal transmitted powers in Europe and USA are 23 - 33 dBm, depending on the region and specific IEEE 802.11p channel (for Europe see Figure 2.1).

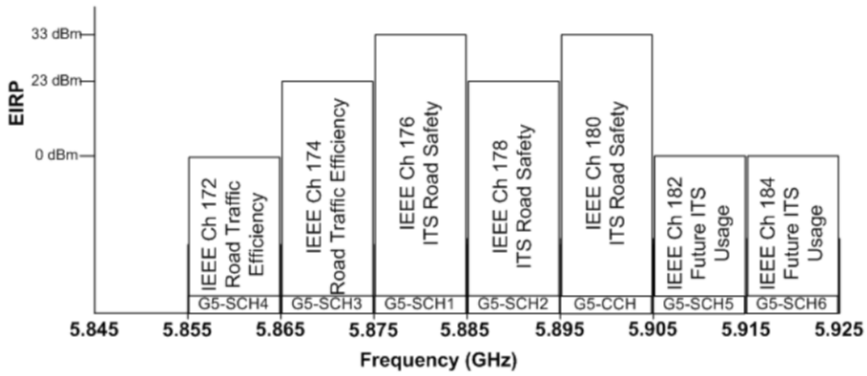


Figure 2.1: Europe ITS Channel Plan [8]

Collisions in IEEE 802.11p are solved by using *Carrier Sense Multiple Access with Collision Avoidance* (CSMA/CA) with variuos Interframe Spaces. The *Short Interframe Space* (SIFS) defines periods between contol packets as Acknowledgements (ACK) while the longer *Arbitration interframe space* (AIFS; or DIFS in other standards) defines periods when a device cannot start transmitting after preceding frame (see Figure 2.2). The SIFS period for the IEEE 802.11p seems not to be defined, however, most of sources use 16 μs (same as IEEE 802.11a) or 32 μs (doubled time as in PHY layer). [5, p. 7] [6, p. 3]

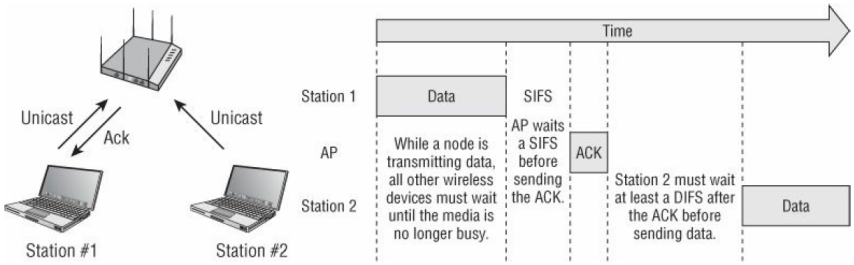


Figure 2.2: SIFS and AIFS (DIFS) [7]

The MAC layer frame used by the PHY layer is called *PHY sublayer service*

data unit (PSDU). It is composed of a variable-length MAC Header, frame body, and a Frame Check Sequence (FCS) CRC-32 (see Figure 2.3).

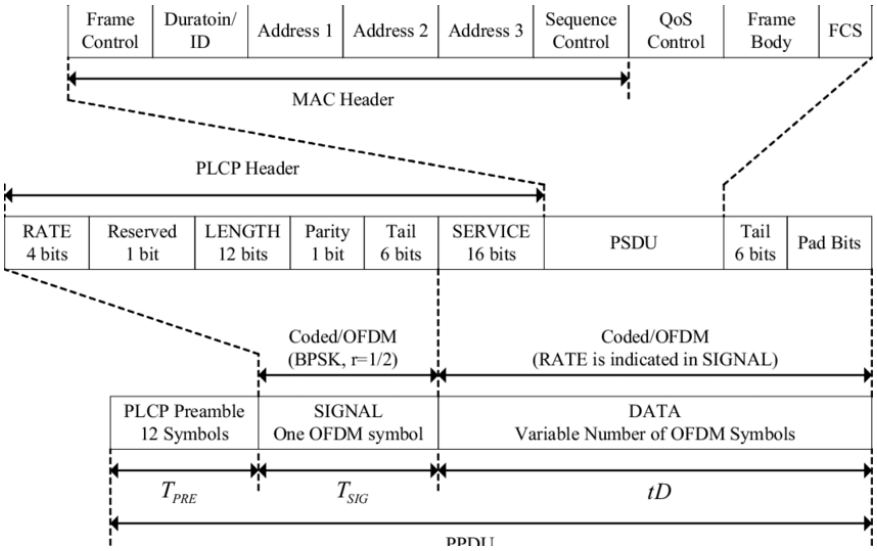


Figure 2.3: MAC and PHY packets [4]

2.4 IEEE 802.11p PHY layer

This section mainly follows IEEE 802.11a description in [9].

The physical (PHY) layer of IEEE 802.11p standard is very similar to 802.11a standard. The differences are in bandwidth (defined for 5, 10 and 20 Mhz with default value of 10 MHz instead of 20 MHz in 802.11a), operating frequencies (5.9 GHz band) and allowed transmit powers. [3] The lower bandwidths are achieved by reducing the sample rate of 802.11a to half or quarter while not changing the algorithms PHY algorithms. Lowering the bandwidth is beneficial for a receiver in terms of lower noise and lower inter-symbol interference due to a delay spread that exceeds the extended cyclic prefix length. [10]

2.4.1 PLCP and PHY frame

The *Physical Layer Convergence Procedure* (PLCP) defines a method of mapping the IEEE 802.11 MAC layer frame PSDU into a frame format suitable for sending and receiving user data and management information between two or more stations. [9] This means, the PLCP defines a synchronization preamble and maps PSDU together with management information fields (*RATE*, *LENGTH* and *SERVICE*) into PHY fields *SIGNAL* and *DATA* (see Figure 2.3 and Figure 2.4).

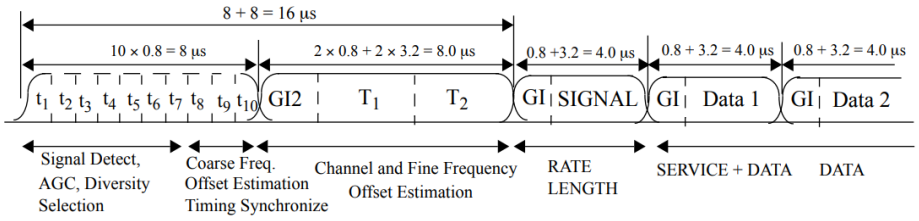


Figure 2.4: PHY PLCP preamble 801.11a (halved periods at 20 MHz) [9]

Management information fields

The *SIGNAL* field contains *RATE* and *LENGTH* fields (see Figure 2.5). The *RATE* field defines used modulation and code rate (see Table 2.1) while *LENGTH* is length of PSDU in bytes, this information is then used for processing the *DATA* field. The *SIGNAL* field is then independently mapped (without scrambling) onto one OFDM symbol with lowest data rate of 3 Mb/s.

The PHY layer must support transmission and reception of the 3, 6, and 12 Mb/s data rates. [10, p. 1165]

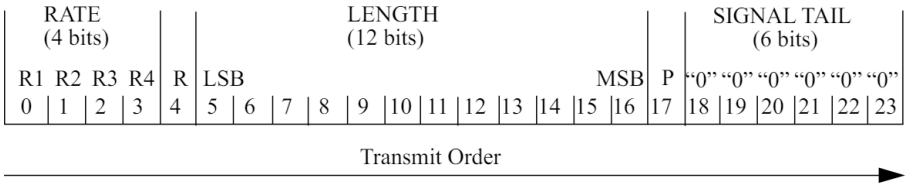


Figure 2.5: The *SIGNAL* field structure [9, p. 14]

Data rate (Mb/s)	*RATE* field	Modulation	Coding rate (R)	N_{BPSC}	N_{DPSC}
3	1101	BPSK	1/2	48	24
4.5	1111	BPSK	3/4	48	36
6	0101	QPSK	1/2	96	48
9	0111	QPSK	3/4	96	72
12	1001	16-QAM	1/2	192	96
18	1011	16-QAM	3/4	192	144
24	0001	64-QAM	2/3	288	192
27	0011	64-QAM	3/4	288	216

Table 2.1: Data rate [9, p. 9, 14]

The *DATA* field is composed of *SERVICE*, PSDU, *TAIL* and *PAD* fields. The 16-bit *SERVICE* field is used for the descrambler initialization (7 zero bits, see subsection 2.4.2) and the rest is reserved for a "future use" (9 zero bits). The 6-bit *TAIL* field of zeros is used for decoder state flushing and the *PAD* field fills remaining empty space in the last OFDM symbol with zeros. [9, p. 7]

PLCP preamble

Every IEEE 802.11p packet PLCP preamble consist of two synchronization sequences, these sequences does not change and are used for packet detection and receiver synchronization.

The first sequence, called Short training sequence (STS), let us denote it $s^{\mathrm{STS}}[k], k = 0..159$, consists of ten repetitions of a 16 samples long signal. Four repetitions of this sequence can be generated by computing 64-IFFT of following sequence $S^{\mathrm{STS}}_{-26,26}$. (See subcarrier to IFFT mapping in Table 2.3.) The normalization of $\sqrt{\frac{13}{6}}$ is used to get the same average power as following LTS sequence and OFDM modulated data. In a receiver this sequence should be used for packet detection, timing acquisition and coarse frequency acquisition. [9] [12]

$$\begin{aligned} S^{\mathrm{STS}}_{-26,26} = \sqrt{\frac{13}{6}} \cdot \{&0,0,1+j,0,0,0,-1-j,0,0,0,1+j,0,0,\\ &0,-1-j,0,0,0,-1-j,0,0,0,1+j,0,0,0,0,\\ &0,0,0,-1-j,0,0,0,-1-j,0,0,0,1+j,0,\\ &0,0,1+j,0,0,0,1+j,0,0,0,1+j,0,0\} \end{aligned} \tag{2.1}$$

The second sequence, Long training sequence (LTS), let us denote it $s^{\mathrm{LTS}}[k], k = 0..159$, consists of two repetitions of a 64 samples long signal with one 32 samples long cyclic prefix (GI2). One repetition of this sequence can be generated by computing 64-IFFT of following sequence $S^{\mathrm{LTS}}_{-26,26}$. (See subcarrier to IFFT mapping in Table 2.3.) This sequence can be used for fine frequency acquisition and channel estimation. [9] [12]

$$\begin{aligned} S^{\mathrm{LTS}}_{-26,26} = \{&1,1,-1,-1,1,1,-1,1,-1,1,1,1,1,1,1,-1,-1,1,\\ &1,-1,1,-1,1,1,1,1,0,1,-1,-1,1,1,-1,1,-1,\\ &1,-1,-1,-1,-1,-1,1,1,-1,-1,1,-1,1,-1,1,1,1,1\} \end{aligned} \tag{2.2}$$

2.4.2 DATA scrambler

The DATA field (SERVICE field, PSDU field and Pad field; the Tail field is not scrambled) is randomized by a 127 bits long pseudo-random sequence produced by the synchronous scrambler with generator polynomial $S(x)$, visualized in Figure 2.6. The scrambler in transmitter and the descrambler in receiver are of the same structure. The DATA field is fed into the scrambler byte by byte with order of least significant bits first.

$$S(x) = x^7 + x^4 + 1 \tag{2.3}$$

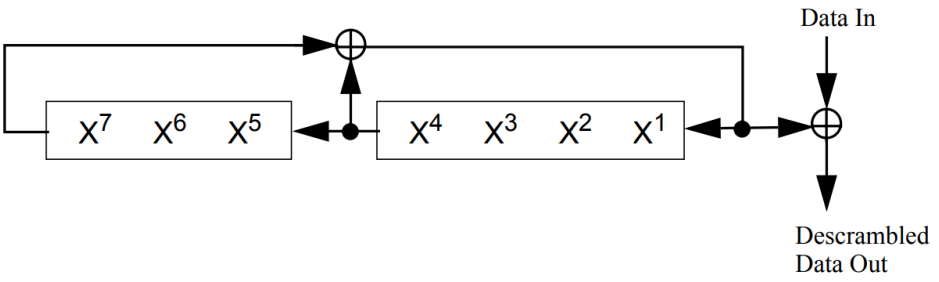


Figure 2.6: Data scrambler [9, p. 16]

The initial state of the scrambler is selected randomly with each packet. It can be then synchronized in a receiver's descrambler from decoded *SERVICE* field, scrambling the initial sequence of seven zeros will directly expose scrambler's following state.

2.4.3 Convolutional encoder

The convolutional encoder with code rate $R = 1/2$ is used, its generator polynomials are $g_0 = 133_8$ and $g_1 = 171_8$, this is visualized in Figure 2.7. The data outputs A and B are then alternated into the coded output $(A_0B_0A_1B_1...)$.

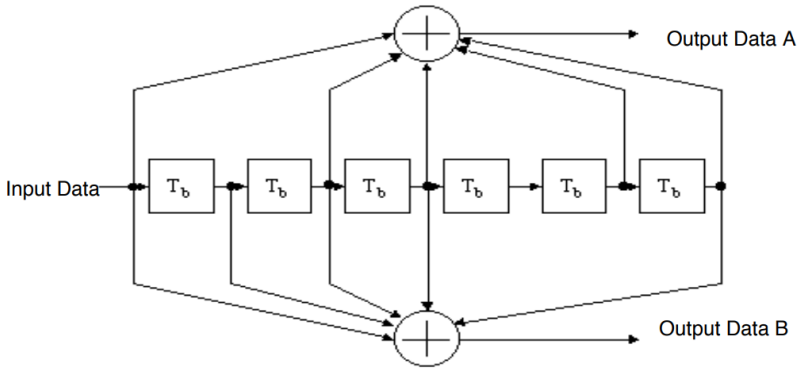


Figure 2.7: Convolutional encoder [9, p. 17]

Higher coding rates $R = 2/3$ and $R = 3/4$ can be used by applying a process called puncturing. This is done by so-called bit-stealing and bit-insertion. In coding rate $R = 2/3$ puncturing is done by deleting all $\{B_i\}_{i \text{ odd}}$ from the coded output, while $R = 3/4$ puncturing is done by deleting (stealing) all $\{B_i\}_{(i \bmod)=1}$ and $\{A_i\}_{(i \bmod 3)=2}$. (In other words, repeat: 1. Keep both A_iB_i, 2. Keep only A_i, 3. Keep only B_i.) The stolen bits are then reinserted as zero confidence bits in a receiver.

These code rates are used according to selected *RATE* field (see Table 2.1).

2.4.4 Data interleaving

SIGNAL field bits and encoded *DATA* field bits are then interleaved by a block interleaver with block size corresponding to the number of bits in a single OFDM symbol N_{CBPS} (see Table 2.1). *SIGNAL* field bits are interleaved separately into one OFDM symbol. This is done by two consecutive permutations. [9, p. 17]

The first permutation maps adjacent bits onto nonadjacent subcarriers, it is defined by following relation.

$$i = (\frac{N_{\mathrm{CBPS}}}{16}) \cdot (k \bmod 16) + \mathrm{floor}(\frac{k}{16}), \quad k = 0, 1, \ldots, N_{\mathrm{CBPS}} - 1 \tag{2.4}$$

The second permutation alternately maps consecutive bits onto less and more significant bits of the constellation, it is defined by following relation,

$$j = s \cdot \mathrm{floor}(\frac{i}{s}) + (i + N_{\mathrm{CBPS}} - \mathrm{floor}(\frac{16 \cdot i}{N_{\mathrm{CBPS}}})) \bmod s, \quad i = 0, 1, \ldots, N_{\mathrm{CBPS}} - 1 \tag{2.5}$$

where

$$s = \max(1, \frac{N_{\mathrm{CBPS}}}{2}) \tag{2.6}$$

2.4.5 Subcarrier modulation mapping

Interleaved data can be then modulated onto OFDM subcarriers by BPSK, QPSK, 16-QAM or 64-QAM modulation using Gray-coded constellations according to Figure 2.8. To achieve the same average power for all modulations the result is multiplied by normalization factor K_{MOD} as stated in Table 2.2. The resulting value in each subcarrier can be thus expressed as

$$d = K_{\mathrm{MOD}} \cdot (I + \mathrm{j}Q) \tag{2.7}$$

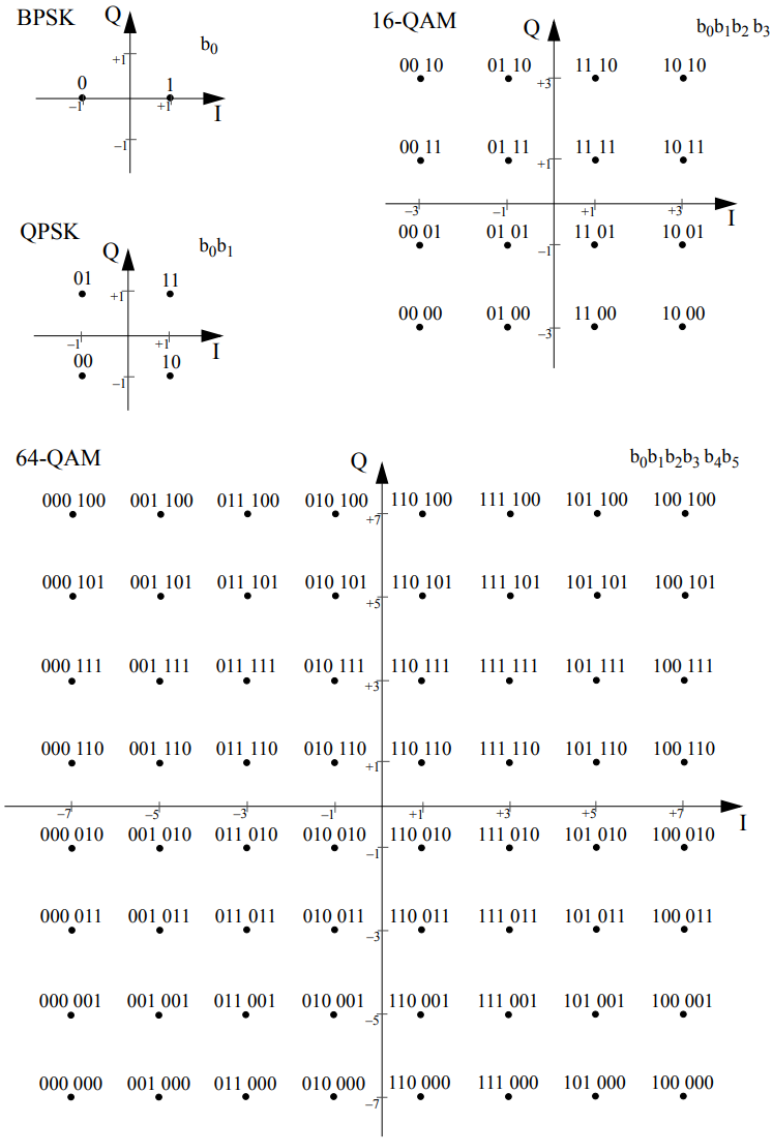


Figure 2.8: Constellation mapping [9, p. 20]

Modulation	K_{MOD}
BPSK	1
QPSK	$1/\sqrt{2}$
16-QAM	$1/\sqrt{10}$
64-QAM	$1/\sqrt{42}$

Table 2.2: Modulation normalization factor [9, p. 19]

2.4.6 Pilot subcarriers

Pilot subcarriers are four known BPSK modulated data (+1, +1, +1, -1) placed at subcarriers (-21, -7, +7, +21), respectively. These pilots are scram-

bled by multiplying all of them by ±1, this scrambling value is generated for each OFDM symbol by the scrambler described in subsection 2.4.2 (initialized with "all ones") with output mapping: $0 \rightarrow 1$ and $1 \rightarrow -1$. These pilot subcarriers should be used in a receiver to track (correct) frequency offset value during reception. [12, p. 4925]

OFDM modulation

The result of modualtion mapping (48 subcarriers) and computed pilots (4 subcarriers) are passed to 64-IFFT input, remaining 12 subcarrier inputs are zeroed. This mapping is captured into details in Table 2.3.

Subcarrier	FFT input	Data index
-32..-27	32..37	Null
-26..-22	38..42	0..5
-21	43	Pilot 1
-20..-8	44..56	6..18
-7	57	Pilot 2
-6..-1	58..63	19..23
0	0	Null
1..6	1..6	24..29
7	7	Pilot 3
8..20	8..20	30..42
21	21	Pilot 4
22..26	22..26	43..47
27..31	27..31	Null

Table 2.3: OFDM subcarriers to IFFT and data mapping

The 64-samples IFFT outputs are prepended with 16 samples of cyclic prefix called Guard interval (GI; see Figure 2.4) taken from each IFFT output end.

2.4.7 Signal joining

The prepared 80-samples long OFDM symbols are connected by windowing. This process is defined by multiplying each OFDM symbol by window $w_T(t)$ defined except for parameter T_{TR} [9]:

$$w_T(t) = \begin{cases} \sin^2(\frac{\pi}{2}(0.5 + \frac{T}{T_{\mathrm{TR}}})), & (-T_{\mathrm{TR}}/2 < t < T_{\mathrm{TR}}/2) \\ 1 \quad (T_{\mathrm{TR}}/2 \leq t < T - T_{\mathrm{TR}}/2) \\ \sin^2(\frac{\pi}{2}(0.5 - (t-T)/T_{\mathrm{TR}})), & (T - T_{\mathrm{TR}}/2 \leq t < T + T_{\mathrm{TR}}/2) \end{cases} \tag{2.8}$$

Chapter 3

Used Hardware and Software

3.1 Used Hardware

In the range of the assignment we are limited in possible hardware to ZedBoard platform and ADRV9002 transceiver. Our laboratory owns two ZedBoard platforms and two ADRV9002 transceiver evaluation boards EVAL-ADRV9002 that we can use (see in Figure 3.1). In this section we would like to describe them.

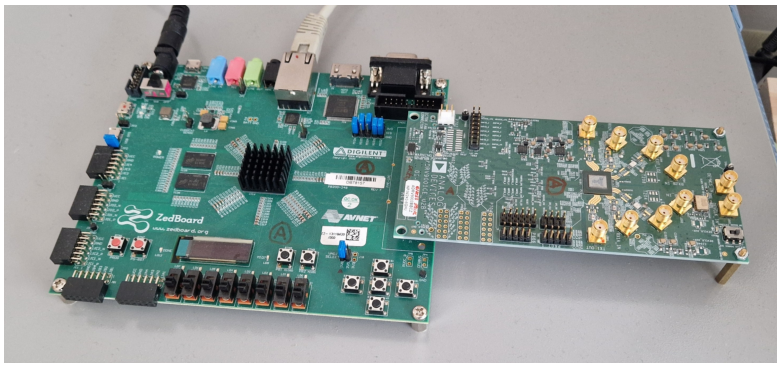


Figure 3.1: ZedBoard and ADRV9002 photo

3.1.1 ZedBoard

ZedBoard™ [13] is a development board built around a System on Chip (SoC) that combines an FPGA Programmable logic (PL) with a Processing system (PS). The used SoC is Xilinx *Zynq®-7000 All Programmable SoC XC7Z020-CLG484-1*. [14]

The PL part (equivalent to AMD *Artix-7* FPGA) contains 85K Programmable Logic Cells (53,200 LUTs and 106,400 flip-flops), 140 block RAMs (BRAM, each 36 Kb) and 220 DSP Slices (18x25 MACs) with maximal frequency of 741 MHz.

The PS part is composed of a dual-core ARM Cortex-A9 based CPU with maximal frequency of 667 MHz and common peripherals connected to MIO port (see Figure 3.2).

Both PS and PL parts are connected by 9 various AMBA AXI4 interfaces, extended multiplexed I/O interface (EMIO), 3 DMA channels, clock and reset outputs and other miscellaneous interfaces. That above mentioned can be seen in Figure 3.2.

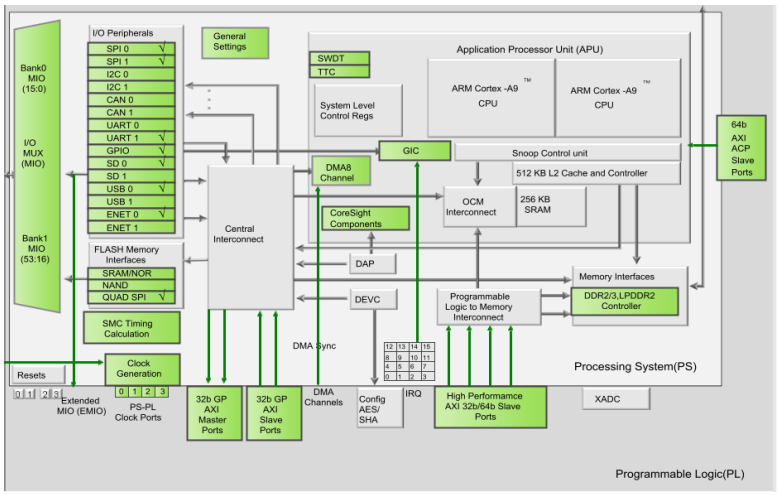


Figure 3.2: Zynq-7000 SoC XC7Z020 PL-PS connections (Vivado)

Except for that mentioned above, ZedBoard also features:

- 512 MB DDR3 RAM
- SD card storage
- 10/100/1000 Ethernet
- USB-JTAG programmer
- USB OTG 2.0 and USB-UART
- Multiple displays (1080p HDMI, 8-bit VGA, 128 x 32 OLED)
- Multiple I/O (including FMC)

For more details on ZedBoard see [13] , on Zynq-7000 SoC XC7Z020 see [14].

3.1.2 ADRV9002 transceiver

ADRV9002 [15] is a Dual Narrow-Band and Wide-band RF Transceiver manufactured by company Analog Devices Inc. (ADI). ADRV9002 belongs to the family ADRV9001 where other models differ only in absence of either the second channel or digital predistortion, therefore, most of designs and documents are referenced for the entire ADRV9001 family.

Key specifications:

- 2 × 2 transceiver (2 channels for both RX and TX)

- Frequency range of 30 MHz to 6 GHz
- Transmitter and receiver bandwidth from 12 kHz to 40 MHz
- Two fully integrated, fractional-N, RF synthesizers
- LVDS and CMOS synchronous serial data interface options (see below)
- Dynamic profile switching for dynamic data rates and sample rates
- Fully programmable via a 4-wire SPI

For our project we will use ADRV9002 evaluation board EVAL-ADRV9002 [16], version *ADRV9002NP/W2/PCBZ* for frequency band 3GHz to 6GHz. It is equipped with a 38.4 MHz oscillator and a single FMC connector compatible with ZedBoard. It is also worth mentioning that ZedBoard does not support the same voltage on LVDS (Low-voltage differential signaling) high-speed IQ data synchronous-serial interface (SSI) interface (LSSI), this implies that the IQ data transfer has to use the second option, complementary metal oxide semiconductors (CMOS) interface (CSSI), which offers maximal transfer rate only 20 MSPS. [17, p. 59] Nevertheless, this value is sufficient for the further development.

The ADRV9002 ADC fullscale translates to approximately 8.6 dBm (assuming 0 dB attenuation) with maximum safe (peak) input power on Rx1 and Rx2 of 18 dBm. [17, p. 187] Maximum transmitter output power is approximatelly 7.2 dBm at 5800 MHz (assuming 0 dB attenuation). [18, p. 5]

3.2 Used Software

3.2.1 AMD Vivado and Vitis

Company Advanced Micro Devices, Inc. (AMD) (formerly Xilinx, Inc.) offers entire environment of products for FPGA, embedded and (MP)SoC design. In this project we will use AMD Vivado with AMD Vitis, these environments can be used in combination, since Vivado aims for FPGA and (MP)SoC HDL design while Vitis aims for software development for each programmable core in a design.

We will be using Vivado and Vitis version 2023.2.2 (an update of version 2023.2) for the entire project and version 2022.2 for the initial build. When installing Vivado we will need only device: Devices –> SoC –> Zynq-7000.

3.2.2 ADI HDL Reference Designs

Analog Devices Inc. (ADI) offers a collection of HDL reference designs. These designs are AMD and Intel reference projects for various combinations of (MP)SoC boards with ADI products, enabling given (MP)SoC to use basic functionalities of compatible ADI products at hardware level.

We will be using last stable reference design for the ADRV9001 and ZedBoard available at ADI GitHub repository *hdl* [19], branch *hdl_2022_r2*, subdirectory *./projects/adrv9001/zed/*. The project is not built and it references libraries (also not built) and other files higher in the repository tree.

3.2.3 ADI Kuiper Linux

The Analog Devices Kuiper Linux [21] is an edited distribution of *Raspberry Pi OS image* (previously known as *Raspbian*). The Linux features preinstalled ADI and other related libraries and applications (including Libiio, py-adi and IIO Oscilloscope).

We will use this prepared Linux distribution as an operating system running on the PS part of the ZedBoard SoC.

3.2.4 ADI TES and Libiio

Analog Devices Inc. offers two ways of configuring the ADRV9001.

- The first possibility is to use Transceiver Evaluation Software (TES) [25], TES offers detailed ADRV9001 configuration, being able to set every possible combination parameters, however, its full functionalities (including connection to the board) are not offered for the ZedBoard. [26]
- The second possibility is Libiio [22], Libiio is an open source library [23] that has been developed by Analog Devices to ease the development of software interfacing Linux Industrial I/O (IIO) devices. It is easy to use, supports more devices (including the ZedBoard) and can be connected remotely (via IIO Daemon (IIOD) server). On the other hand, Libiio cannot control all ADRV9001 parameters and it makes an additional interface between the user and the underlying hardware, making it harder to track all actual hardware parameters. We can also use library called *py-adi*, this library is built on-top of Libiio, it offers object oriented approach with many methods prepared so regular tasks are very easy to perform on just a few lines. When speaking about Libiio we will be mostly using it via *py-adi*.

Nevertheless, these approaches can be combined as the Libiio offers a possibility to load a profile (set of ADRV9001 parameters) generated by TES.

3.3 Workflow preparation

In this section we prepare the overall workflow for the further progress. Our project development will be based on Windows 11. This will mean a little complication in the workflow preparation, however, the overall work should be the same as done on a Linux distribution.

3.3.1 ADI HDL Reference Designs preparation

We will use the last stable release *2022_r2* of the ADI HDL Reference Designs [19], this release can be found as branch *hdl_2022_r2*.

Build *hdl_2022_r2* in Vivado 2022.2

A downside of using ADI HDL Reference Designs is that each HDL version has to be built by an exact Vivado version. Specifically, release *hdl_2022_r2* has to be built by Vivado 2022.2.

To build our Reference Design (on Windows) following steps need to be made (see ADI Building HDL page [20]):

1. Install Vivado 2022.2
2. Install Cygwin program with *make* command
3. Include Vivado and 2022.2 path into Cygwin *~/.bashrc* file

Listing 3.1: Include Vivado and 2022.2 path into Cygwin

```
1 export PATH=$PATH:/cygdrive/path_to /Xilinx/Vivado/2022.2/bin
2 export PATH=$PATH:/cygdrive/path_to /Xilinx/Vivado_HLS/2022.2/bin
3 export PATH=$PATH:/cygdrive/path_to /Xilinx/Vitis/2022.2/bin
4 export PATH=$PATH:/cygdrive/path_to ...
      /Xilinx/Vitis/2022.2/gnu/microblaze/nt/bin
5 export PATH=$PATH:/cygdrive/path_to /Xilinx/Vitis/2022.2/gnu/arm/nt/bin
6 export PATH=$PATH:/cygdrive/path_to ...
      /Xilinx/Vitis/2022.2/gnu/microblaze/linux_toolchain/nt64_be/bin
7 export PATH=$PATH:/cygdrive/path_to ...
      /Xilinx/Vitis/2022.2/gnu/microblaze/linux_toolchain/nt64_le/bin
8 export PATH=$PATH:/cygdrive/path_to ...
      /Xilinx/Vitis/2022.2/gnu/aarch32/nt/gcc-arm-none-eabi/bin
```

4. Create the project folder with a short path (Vivado makes long paths that can grow over Windows 11 limit). In my case: *C:/zedboard_adrv9002_project/*. This path will be our default path for the rest of the project, referenced as *./*
5. Place the ADI HDL Reference Designs repository *adi_hdl_2022_r2/* into the project folder: *./*
6. In Cygwin run *make* command in the ADRV9001 + ZedBoad project folder (*./adi_hdl_2022_r2/projects/adrv9001/zed/*). Now the project and all needed libraries should be built by Vivado 2022.2
7. Open the project built Vivado 2022.2 in Vivado 2023.2.2 and select *Automatically upgrade to the current version*
8. Export the block design script by: IP INTEGRATOR –> Open Block Design –> File –> Export –> Export Block Design... –> Tcl file: *./adi_hdl_2022_r2/projects/adrv9001/zed/system.tcl*

Rebuild the project in Vivado 2023.2.2

The project migrated to Vivado 2023.2.2 includes some files outside of its folder, from which some of them have been copied and unused while some of them were not copied. In this state it is not easy to make amendments in the project, so the easiest solution to this problem is creating a new clean Vivado 2023.2.2 project. This can be done following these steps:

1. Create new Vivado project folder *./src_HDL/*
2. Create Vivado 2023.2.2 project named *adrv9001_zed* in *./src_HDL/* for *ZedBoard Zynq Evaluation and Development kit* from vendor *avnet.com*, leave other configurations in the default state
3. Include previously built ADI libraries to the current Vivado project: PROJECT MANAGER –> IP Catalog –> right click –> Add Repository... –> select *./adi_hdl_2022_r2/library/*
4. Generate block design by running previously exported script: Tools –> Run Tcl script –> *./adi_hdl_2022_r2/projects/adrv9001/zed/system.tcl*
5. Create block design Verilog wrapper: Sources –> Hierarchy –> Design Sources –> right click on system.bd –> Create HDL Wrapper... -> Let Vivado manage wrapper and auto-update
6. Copy source files from the old project to *./src_HDL/adrv9001_zed.srcs/sources_1/new/*:
   - *./adi_hdl_2022_r2/projects/adrv9001/zed/system_top.v*
   - *./adi_hdl_2022_r2/library/common/ad_iobuf.v*

   Add these source files to the project: Sources –> Add Sources –> Add or create design sources –> Add Files –> select files from above
7. Copy constraints files from the old project to *./src_HDL/adrv9001_zed.srcs/constrs_1/new/*:
   - *./adi_hdl_2022_r2/projects/adrv9001/zed/system_constr.xdc*
   - *./adi_hdl_2022_r2/projects/adrv9001/zed/cmos_constr.xdc*
   - *./adi_hdl_2022_r2/projects/common/zed/zed_system_constr.xdc*

   Add these source files to the project: Sources –> Add Sources –> Add or create constraints –> Add Files –> select files from above
8. Set higher priority of *system_constr.xdc* by: right click on *system_constr.xdc* –> Source File Properties... –> Properties –> set *PROCESSING_ORDER* to *LATE*
9. Add gitignore file to the *./src_HDL/* ignoring Vivado syntheses, implementations and simulations folders (This method is not optimal since Vivado stores a lot of files)

- *adrv9001_zed.sim/*
- *adrv9001_zed.runs/*

Remove DDS and build

Now, the project should be prepared to be built. However, we make one amendment. We found out that in ADI file *./adi_hdl_2022_r2/library/axi_adrv9001/axi_adrv9001_tx_channel.v* usage of entity *ad_dds* wastes a lot of resources (see section 4.3). This DDS entity is responsible for generating example harmonic signal IQ data for both transmitters at arbitrary frequency, however, we will not use it.

1. We can comment the whole *ad_dds* usage out in *./adi_hdl_2022_r2/library/axi_adrv9001/axi_adrv9001_tx_channel.v* and assign its output *dac_dds_data_s* with zeros.
2. This change needs to be updated in the project: IP INTEGRATOR –> Open Block Design –> Show IP Status –> Upgrade Selected

After this change we are ready to build and export the Vivado project output, it can be done by following:

1. PROGRAM AND DEBUG –> Generate Bitstream (this can take approximately 20 minutes)
2. File –> Export –> Export Hardware... –> Include bitstream –> select *./src_HDL/system_top.xsa* (preselected)

3.3.2 Vitis workflow

The Vivado output *./src_HDL/system_top.xsa* generated in the previous section contains all information about the PL and PS initialization. However, we need a few more features available in Vitis that cannot be done in Vivado alone. Namely:

- FSBL: Loads the PL and PS configuration into the SoC from an SD card
- U-Boot: Boots Linux kernel in the PS part (see subsection 3.2.3)

These functionalities have to be implemented in the PS part (the SD card controller is connected to PS part only). Booting the SoC from an SD card is a common and convenient practise and Vitis has an option to generate this option (FSBL) automatically.

The ADI Kuiper Linux U-Boot file has also been prepared by ADI, it can be found in ADI Kuiper Linux imaged SD card (see subsection 3.3.3) at location */zynq-zed-adv7511-adrv9002/bootgen_sysfiles.tgz/u-boot_zynq_zed.elf*. We will extract the compressed file and copy whole content of */zynq-zed-adv7511-adrv9002/* to *./src_SDK/analog_copy/*.

The overall process of creating an SD card boot image *BOOT.bin* in Vitis is following:

1. Create new folder *./src_SDK/* and open it as a Vitis 2023.2.2 workspace

2. Represent the Vivado project by a Vitis *Platform Component*: Create Platform Component –> Fill component name: *platform_system_top* –> Hardware Design –> Browse –> select: *./src_HDL/system_top.xsa* –> select *Operating system*: *linux*, check *Generate Boot artifacts*

3. Build *platform_system_top* (under *Flow* tab)

4. Create a *System Project* (wrapper for application on a given platform): File –> New Component –> System Project –> select *System project name*: *system_linux* –> Select platform: *platform_system_top*

5. Create a blank application so that *system_linux* would not be empty: File –> New Component –> Application –> Component name: *app_blank_app* –> Select platform: *platform_system_top* –> continue with defaults

6. Add the blank application to *system_linux*: VITIS COMPONENTS –> *system_linux* –> Settings –> vitis-sys.json –> Add Existing Component –> Application –> *app_blank_app*

7. Build *system_linux* (under *Flow* tab)

8. Generate an SD card boot image: VITIS COMPONENTS –> *system_linux* –> Flow: Create Boot Image –> Add partition –> File path: *./src_SDK/analog_copy/bootgen_sysfiles/u-boot_zynq_zed.elf* –> select default *Output BIF File Path* and *Output Image* (*./src_SDK/system_linux/BOOT.bin*) –> Create Image

Generated file *./src_SDK/system_linux/BOOT.bin* is an SD card boot image for the ZedBoard. It contains all First and Second Stage Bootloaders and both PL and PS parts configurations. This configuration is done automatically after boot. The Linux kernel for the PS part (booted by the Second Stage Bootloader *./src_SDK/analog_copy/bootgen_sysfiles/u-boot_zynq_zed.elf*) is described in following subsection 3.3.3.

When we need to update an existing *Platform Component* by new Vivado output (*./src_HDL/system_top.xsa* file), we can do it by :

1. VITIS COMPONENTS –> *platform_system_top* –> Settings –> vitis-comp.json –> Switch XSA –> *./src_HDL/system_top.xsa*

2. Build *platform_system_top* (see above)

3. Build *system_linux* (see above)

4. Generate an SD card boot image (see above)

3.3.3 ADI Kuiper Linux preparation

The ADI Kuiper Linux compiled image can be downloaded at its webpage [21].

We can image the SD card (min. 16 GB) by following steps:

1. Download the ADI Kuiper Linux compiled image [21]
2. Image an SD card by the downloaded image file by an imager application (Etcher, Raspberry Pi Imager, WinDisk32)
3. Copy following prepared files on the SD card into its root directory of the *BOOT* partition:
   - zynq-zed-adv7511-adrv9002/BOOT.BIN
   - zynq-zed-adv7511-adrv9002/zynq-zed-adv7511-adrv9002/devicetree.dtb
   - zynq-common/uImage
4. Insert the SD card into the ZedBoard and power it on.
5. Connect by UART to PC (Baud: 115200)
6. Change IP address and enable display output by running included: *enable_static_ip.sh* and *enable_dummy_display.sh*

We can notice that the ADI Kuiper Linux already contains its PL and PS default boot image file *BOOT.bin*. However, in subsection 3.3.1 and subsection 3.3.2 we showed a way how to build this file, this enables us to make PL part modifications while keeping the Kuiper Linux intact in the PS part.

3.3.4 ADI TES profile configuration

After we have prepared our HW and ADI Kuiper Linux, we would like to configure the ADRV9002. As mentioned before (see subsection 3.2.4), we can combine TES and Libiio by generating a profile (set of ADRV9002 parameters) in TES that can be then easily loaded from the Libiio.

All TES files (including the used session) can be found in folder *./others/TES_802_11p/*. A brief description of the used ADRV9002 TES configuration is following:

- **Device Configuration**: Channel 1 only, FDD duplex, CMOS 4-Lane DDR SSI, I/Q 16-bit Signal Type, Dataport and Interface Rate of 20 MSPS, RF Channel Bandwidth of 10 MHz
- **Clock**: Frequency 38.4 MHz (according to the EVAL-ADRV9002 oscillator)
- **Carriers**: Both Carrier frequencies of 5900 MHz

- **Radio**: SPI Channel 1 Enablement Mode, Second Order Analog Low-Pass Filter with -1 dB Frequency of 7 MHz, Transmit Data Source - Set data through FPGA
- **RX (TX) Filters**: Source: *low_pass_10MHz_20MHz.txt* (TX: Interpolation Factor: 2)

The *FDD Duplex* is used because it allows us to use both TX and RX ports at the same time. *Dataport and Interface Rate* of 20 MSPS is used because we did not find a combination of parameters to set the bandwidth and the interface rate both to wanted 10 MHz, however, while using the following filter, the downsampling in the receiver and upsampling in the transmitter will be both effortless to make.

File *low_pass_10MHz_20MHz.txt* of 128-tap half-band low-pass filter coefficients was generated by the MATLAB script *filter.m*.

Selected parameters can be validated by using *Demo Mode* in *Connection* tab and clicking *Program* tab. After configuring the ADRV9002 the *Profile File* and its corresponding *Stream Image* can be generated in the *File* tab, these two files contain all needed ADRV9002 configuration parameters.

Chapter 4

Hardware and Signal processing Implementation of IEEE 802.11p

This chapter is devoted to implementation of a custom Vivado IP block for IEEE 802.11p reception in VHDL. This reusable block in the PL part of *Zynq-7000 SoC XC7Z020*, connected to the PS part, shall process incoming IQ samples and implement each stage of a IEEE 802.11p receiver, then it must be capable of providing the processed data to the ADI Kuiper Linux running in the PS part.

We chose the implementation in the PL part because of the strict realtime requirements on response time, especially the 16 or 32 μs SIFS interval for frame Acknowledgements (see section 2.3).

The IP project location is *./src_HDL/IP_802_11p/*.

4.1 AXI IP block preparation

4.1.1 Create IP block

We start by creating a new IP block for our built Vivado design. If not specified otherwise in this chapter, we will automatically consider the project in this folder.

1. Open the built *adrv9001_zed* Vivado project
2. Change Target language to VHDL: PROJECT MANAGER –> Settings –> Project Settings –> General –> Target language –> VHDL
3. Create a new AXI4 IP: Tools –> Create and Package New IP... –> Create a new AXI4 peripheral –> fill Name: *IP_802_11p*, IP location: *./src_HDL/IP_802_11p/* –> Interface type: *Lite*, Data Width: 32 Bits, (*Number of Registers* will be changed manually later) –> Next Steps: *Edit IP*
4. Change Target language back to Verilog in *adrv9001_zed*: PROJECT MANAGER –> Settings –> Project Settings –> General –> Target language –> Verilog

5. In the new IP project disable the project deletion: PROJECT MANAGER –> Settings –> Project Settings –> IP –> Packager –> uncheck *Delete project after packaging*
6. Add a new blank block design: IP INTEGRATOR –> Create Block Design –> Design name: *block_design_0*
7. Create *block_design_0_wrapper.vhd* VHDL wrapper for *block_design_0*: Sources –> Hierarchy –> Design Sources –> right click *block_design_0* –> Create HDL Wrapper... –> Let Vivado manage wrapper and auto-update
8. Instantiate *block_design_0_wrapper.vhd* in *./src_HDL/IP_802_11p/IP_802_11p_1_0/hdl/IP_802_11p_v1_0_S00_AXI.vhd*
9. Run Synthesis (When making changes, it is often needed to *Open Block Design* and click on *Refresh Changed Modules* first)
10. Package the IP: PROJECT MANAGER –> Edit Packaged IP –> (When displayed: File Groups –> click on Merge changes from File groups Wizard) –> Review and Package –> Re-package IP

In this state the new IP block contains a few registers implemented by LUTs and Flip-Flops, these registers are accessible from the AXI4 Lite interface and can be modified in file *./src_HDL/IP_802_11p/IP_802_11p_1_0/hdl/IP_802_11p_v1_0_S00_AXI.vhd*.

4.1.2 Implement registers in BRAM

The following needed step is replacing registers implemented by LUTs and Flip-Flops since this combination wastes a lot of resources when having bigger number of registers (see section 4.3). We will use a block RAM (BRAM) available in the *Zynq-7000 SoC XC7Z020* instead of this AXI registers implementation.

The process of creating a BRAM follows:

1. Add new Vivado IP *Block Memory Generator* to *block_design_0*, use following *Simple Dual-Port RAM* configuration with depth of 4096 and width of 32 bits:
   - Basic –> Mode –> *Stand Alone*
   - Basic –> Memory Type –> *Simple Dual Port RAM*, check *Common Clock*
   - Port A Options –> Port A Width –> 32
   - Port A Options –> Port A Depth –> 4096
   - Port A Options –> Operating Mode –> *Read First*
   - Port A Options –> Enable Port Type –> *Use ENA Pin*
   - Port B Options –> Port B Width –> 32

- Port B Options –> Operating Mode –> *Read First*
- Port B Options –> Enable Port Type –> *Use ENB Pin*
- Other Options –> Fill Remaining Memory Location –> check and enter 0

2. Connect BRAM ports to AXI4 Lite:
   a. Connect the BRAM to the *block_design_0*: right click on the new IP –> Make External –> remove trailing *_0* in the new ports names
   b. (Re-)Create HDL Wrapper for *block_design_0*
   c. In *./src_HDL/IP_802_11p/IP_802_11p_1_0/hdl/IP_802_11p_v1_0_S00_AXI.vhd*
      - Add the new ports for *block_design_0_wrapper*
      - Remove all *slv_reg* AXI4 Lite registers, process for writing into them and process for reading from them
      - Connect BRAM ports to AXI4 Lite control and data signals from the previous point (in future, these AXI signals will be multiplexed with another source from the PL part, see subsection 4.2.9)
      - Amend *C_S_AXI_ADDR_WIDTH* to 14 and *OPT_MEM_ADDR_BITS* to 11 (AXI4 addressing is in bytes)
   d. Amend *C_S_AXI_ADDR_WIDTH* to 14 in *./src_HDL/IP_802_11p/IP_802_11p_1_0/hdl/IP_802_11p_v1_0.vhd*
   e. Amend PROJECT MANAGER –> Edit Packaged IP –> Customization Parameters –> *C_S_AXI_ADDR_WIDTH* to 14
   f. Amend PROJECT MANAGER –> Edit Packaged IP –> Addressing and Memory –> *Range* to 16384
   g. Re-run the synthesis and Re-package the IP (see in subsection 4.1.1)

4.1.3 Connection to ADI HDL design and PS part

The first version of our packaged IP block *IP_802_11p* is in this state ready to be used in its parent project *adrv9001_zed*. It can be added to its block design by: Open project *adrv9001_zed* –> Open Block Design –> right click –> Add IP... –> select *IP_802_11p_v1.0*

AXI Connection

The connection to the PS part can be done by using previously created AXI port in our IP block. Connecting the IP block via AXI4 Lite and assigning it a valid address range will use a part of unused PS address space. We found out that address space 0x5000_0000..0x5000_3FFF (range: 16384) is in the AXI range of (*PL AXI slave port #0* [14]) and it is unused, this can be found out by looking into decompiled device tree available at *./src_SDK/analog_copy/zynq-zed-adv7511-adrv9002/devicetree.dtb* (Linux

command *dts* can be used for the decompilation) and checking *Address Editor* in the block design of the project *adrv9001_zed*.

For connecting the new IP block to the AXI4 bus */sys_ps7/M_AXI_GP0* follow:

1. In *axi_adrv9001* open the block design with the IP block already added
2. Click *Run Connection Automation* –> select *S00_AXI* –> select */sys_ps7/M_AXI_GP0*
3. Open Block Design –> Window –> Address Editor –> Network 1 –> /sys_ps7 –> /sys_ps7/Data –> /IP_802_11p_0/S00_AXI –> change *Master Base Address* to 0x5000_0000 and *Range* to 16K

These steps will connect the AXI4 Lite slave interface *S00_AXI* to the PS part AXI4 master interface *M_AXI_GP0* via automatic AXI Interconnect at required address space. These addresses can be then easily accessed in the PS part by standard C function *mmap* or by opening */dev/mem* in the ADI Kuiper Linux (for example from Python).

ADVR9002 SSI connection

The continuous IQ data flow from the ADRV9002 is provided by a 4-lane CMOS SSI. We could be able to connect to this SSI interface and parallelize the serialized IQ samples, on the other hand, the ADI design IP block *axi_adrv9001* (in *adrv9001_zed* project) already implements this so we can take advantage of it.

We found out that we can use following *axi_adrv9001* outputs with continuous IQ data stream:

1. *adc_1_clk*
2. *adc_1_rst*
3. *adc_1_valid_i0*
4. *adc_1_enable_i0*
5. *adc_1_data_i0[15:0]*
6. *adc_1_data_q0[15:0].*

We can extend inputs of our IP block *IP_802_11p* and connect them to the signals above. To do this, following input signals have to be added to *./src_HDL/IP_802_11p/IP_802_11p_1_0/hdl/IP_802_11p_v1_0.vhd*, *./src_HDL/IP_802_11p/IP_802_11p_1_0/hdl/IP_802_11p_v1_0_S00_AXI.vhd* and to *IP_802_11p block_design_0* (including its wrapper).

1. *RX_CLOCK*
2. *RX_RESET*

3. *RX_VALID*
4. *RX_ENABLE*
5. *RX_IDATA[15:0]*
6. *RX_QDATA[15:0]*

After *IP_802_11p* Re-packaging (see subsection 4.1.1) these new inputs can be connected to the ADI IP block *axi_adrv9001* outputs mentioned above (in the same order).

CLOCK, Switches and LEDs Connection

Now, as we have wired the AXI4 and the SSI interfaces into the *block_design_0* we need a fast clock signal for intended signal and data processing blocks in the *block_design_0*. We are able to use already existing PS clock source *FCLK_CLK0* set to 100 MHz, this clock source is also used for our AXI4 Lite interface. This selection will simplify the communication between the BRAM and other blocks in the *block_design_0*.

In this point we also wanted to leverage available ZedBoard eight LEDs and eight switches (we will not use the switches in the end).

To wire all: clock, switches and LEDs into *block_design_0*, add following signals in the same way as for the SSI connection above:

1. *CLOCK* (input)
2. *SW[7:0]* (input)
3. *LEDS[7:0]* (output)

After *IP_802_11p* Re-packaging (see subsection 4.1.1) the *CLOCK* port can be connected to *FCLK_CLK0* output in *ZynQ7 Processing System* block.

The process of connecting already used LEDs and switches (connected to PS part) to *IP_802_11p* can follow:

1. In *system* block design right click on the IP ports *LEDS[7:0]* and *SW[7:0]* –> Make External –> remove trailing *_0* in their names –> save the block design
2. (Re-)Create HDL Wrapper for *system* block design
3. Add *SW* and *LEDS* wires to *./src_HDL/adrv9001_zed.srcs/sources_1/new/system_top.v*, connect *SW* from the wrapper to hardware switches (*gpio_bd*[18:11], keep the existing connection from PS), connect *LEDS* from the wrapper to hardware LEDs (*gpio_bd*[26:19], remove the existing connection from PS). Other *gpio_bd* wires can stay untouched.

BRAM control registers

We use the AXI4 Lite interface from the PS part to generate control signals for our design. We create control registers synchronized with particular BRAM addresses by the AXI4 Lite from PS part. This can be achieved by intercepting the AXI4 Lite write signals in *./src_HDL/IP_802_11p/IP_802_11p_1_0/hdl/IP_802_11p_v1_0_S00_AXI.vhd* and wiring themm to *block_design_0* (including its wrapper). We implemented following control signals:

- *RESET* (synchronized with: NOT *registers*(0)(0))
- *DETECTION_THRESHOLD* (synchronized with: *registers*(1), synchronized only even bits while odd are forced to zero)
- *SELECT_AXI_REGS_MODE* (synchronized with: *registers*(2)(7 downto 0), in the design often referenced as *MODE*)

4.2 IEEE 802.11p signal and data processing

This section describes theory and implementation of custom VHDL blocks and Vivado IP blocks used in the IP Vivado project *IP_802_11p*.

Custom IEEE 802.11p signal and data processing VHDL files location: *./src_HDL/IP_802_11p/edit_IP_802_11p_v1_0.srcs/sources_802_11p/*. (For better clarity we will call entities or blocks in these VHDL files with their *.vhd* filename extension.)

We also made a simualtion fora whole frame so that we could easily simulate the entire design at the same time. This simulation can load IQ data from a text file. The simulation source is at location *./src_HDL/IP_802_11p/edit_IP_802_11p_v1_0.srcs/sim_1/new/atan_simulation.vhd*. In Figure 4.1 we can see control signals of the most important blocks of the design propagating to the output.

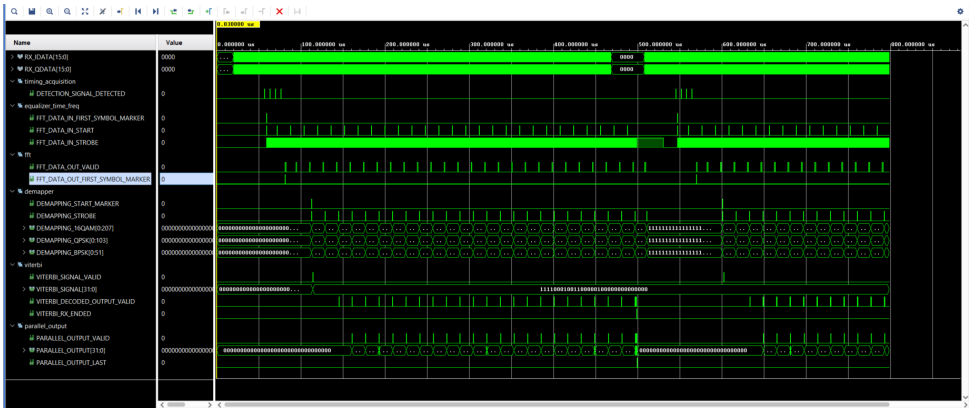


Figure 4.1: Simulation of the whole signal processing

4.2.1 IQ data preparation

In Figure 4.2 we can see a simulation of interface block between the *axi_adrv9001* block and our design. These simple blocks are described below.

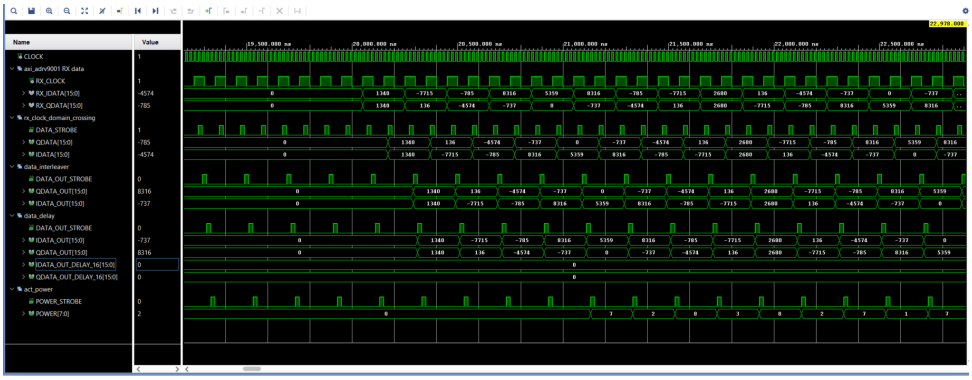


Figure 4.2: Simulation of the IQ data preparation

Clock domain crossing

The IQ data samples provided from ADI IP block *axi_adrv9001* (2x16-bit, at sample rate of 20 MHz, see subsection 3.3.4) are wired into the custom block *rx_clock_domain_crossing.vhd* which buffers all inputs into shift registers (with default depth of 3) by *CLOCK* at 100 MHz. The clock domain crossing is usually done with bigger depth to overcome possible metastability issues.

After reading the data from last buffer registers, a new IQ data sample is detected when following hold for this data:

- *RX_CLOCK* rising edge comes
- *RX_RESET* is LOW
- *RX_VALID* is HIGH
- *RX_ENABLE* is HIGH

For a better timing stability we choose to detect new IQ data with falling edges of buffered *RX_CLOCK*. When a new IQ data is detected, it is passed with a one-cycle *DATA_STROBE* to the block output as *IDATA* and *QDATA*.

IQ Data interleaving

As mentioned in subsection 3.3.4 the IQ data sample rate can be set to 10 MHz or 20 MHz. The data sampled at 20 MHz filtered by a half-band low-pass filter (our case, done in ADRV9002) can be decimated to 10 MHz by interleaving every second sample. This is done in *data_interleaver.vhd* block, the block can be set to pass every i-th sample only, by default this block is set to pass every second sample.

IQ data delay

Due to the planned need of delayed IQ data samples (see subsection 4.2.2) we decided to implement IQ data delay within the separate block *data_delay.vhd*. This block delays the IQ samples in a manner of a shift register triggered by every new IQ sample. The block outputs these data: current data, data delayed by 16, 32, 48 and 64 samples and *DATA_OUT_STROBE* which synchronize all these data outputs.

For the further processing, let us denote the current IQ data sample stream by $x[n]$ (the 16 samples delayed IQ data samples stream then $x[n-16]$, etc.).

Instantaneous power

We created an extra block *act_power.vhd* for measuring instantaneous power of IQ samples. However, we did not use it for signal processing. Therefore, Its *POWER* output has been truncated to 8 MSBs and connected to LEDs output.

4.2.2 Signal detection and Time synchronization

Theory

The first algorithmic task in a receiver is usually signal detection that tests a hypothesis that a signal is being received. An IEEE 802.11p detection block computes (absolute square of) crosscorrelation of the incoming IQ samples and saved 160-samples long STS sequence, denoted $|\mathcal{R}^{\mathrm{STS}}_{160}[n]|^2$, the n-th sample is the newest crosscorrelation sample we can get with newest sample $x[n]$. Then it tests the hypothesis by comparing this value to a threshold dependent on measured noise power.

$$|\mathcal{R}^{\mathrm{STS}}_{160}[n]|^2 = |\sum_{k=0}^{159} x[n-159+k]\bar{s}^{\mathrm{STS}}[k]|^2 \tag{4.1}$$

In the crosscorrelation absolute square computation we can take advantage of the STS periodic structure (see Figure 2.4) and compute the sum of absolute squares of each 16-samples long STS period, we will denote it as $|\tilde{\mathcal{R}}^{\mathrm{STS}}_{160}[n]|^2$.

$$|\tilde{\mathcal{R}}^{\mathrm{STS}}_{160}[n]|^2 = \sum_{i=1}^{10} |\mathcal{R}^{\mathrm{STS}}_{16}[n-16i]|^2 \tag{4.2}$$

where

$$|\mathcal{R}^{\mathrm{STS}}_{16}[n]|^2 = |\sum_{k=0}^{15} x[n-15+k]\bar{s}^{\mathrm{STS}}[k]|^2 \tag{4.3}$$

This computation amendment can worsen the frequency of misdetections by raising this 'crosscorrelation' value for non-STS signals (see Figure 4.3), on the other hand, it increases resistance of the detector to frequency offset up to ten times (see Figure 4.4). We can compute this frequency offset resistance (or

maximal allowed frequency offset $f_{\mathrm{o,max}}$) by setting the phase rotation of our crosscorrelated sequence (from its start to end) equal to $\Delta_\phi = \pi/2$, when a part of the crosscorrelated signal passes this phase shift value, its contribution to the resulting crosscorrelation absolute square becomes negative. The maximal allowed frequency offset is given by following equation

$$f_{\mathrm{o,max}} = \frac{\Delta_\phi}{2\pi}\frac{1}{T_{\mathrm{seq}}} = \frac{\Delta_\phi}{2\pi}\frac{f_s}{N_{\mathrm{seq}}} \tag{4.4}$$

When we set $f_s = 10$ MHz, $\Delta_\phi = \pi/2$ and sequence lengths of $N_{\mathrm{seq}} = 160$ and $N_{\mathrm{seq}} = 16$ we get $f_{\mathrm{o,max}} = 15.6$ kHz and $f_{\mathrm{o,max}} = 156$ kHz, respectively. The difference of detection in frequency offset of 15.6 kHz or 156 kHz can play a significant role, since at 5.9 GHz these values can be translated to oscillator accuracies of 2.6 ppm or 26 ppm.

The Doppler effect on the frequency offset can be neglected since it contributes in units of kHz. For example we compute the Doppler effect for 300 km/h at $f_c = 5.9$ GHz:

$$f_d = f_c\left(\frac{c + v_{\mathrm{RX}}}{c} - 1\right) \approx f_c\left(\frac{c}{c - v_{\mathrm{TX}}} - 1\right) \approx 1.6 \text{ kHz} \tag{4.5}$$

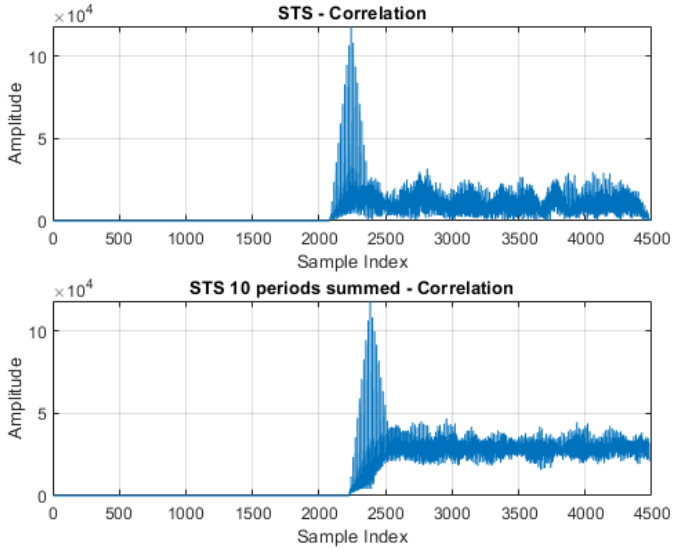


Figure 4.3: STS crosscorrelation methods (no offset)

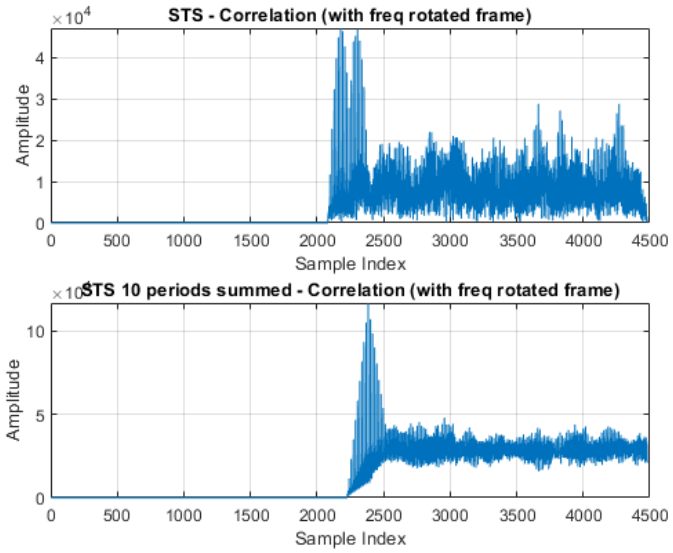


Figure 4.4: STS crosscorrelation methods (50kHz offset

STS Crosscorrelation filter

For computation of STS single period crosscorrelation absolute square $|\mathcal{R}_{16}^{\mathrm{STS}}[n]|^2$, we designed block *Parallel_STS_FIR_Filter.vhd*. This block takes 2x16-bit signed IQ data and filters it with a complex 16-tap FIR filter whose coefficients are selected to compute aforesaid crosscorrelation. These coefficients can be computed by scaling, flipping and conjugating one period of the STS sequence. Subsequently, the filter IQ outputs are both squared and summed to the block output *POWDATA_OUT_XCORR* synchronized with *DATA_OUT_STROBE*. The *POWDATA_OUT_XCORR* output is equal to $|\mathcal{R}_{16}^{\mathrm{STS}}[n]|^2$ in Equation 4.3 delayed by additional 6.1 samples due to buffering and pipelining. Precision of this computation is crucial, therefore, despite including two fixed-point multiplications, the rounding was nearly omitted, resulting in *POWDATA_OUT_XCORR* being of 61 bits (next 3 bits will be added in the following block).

Besides the crosscorrelation computation, block *Parallel_STS_FIR_Filter.vhd* computes input's mutual energy with its 16-taps delayed version $\mathcal{E}_{16}^{\mathrm{STS},3}[n-16]$ (or here called autocorrelation).

$$\begin{aligned}\mathcal{E}_{16}^{\mathrm{STS},3}[n-16] &= x[n-16]\bar{x}[n-32] + x[n-32]\bar{x}[n-48] + x[n-48]\bar{x}[n-64] \\ &= \sum_{i=1}^{3} x[n-16i]\bar{x}[n-16(i+1)]\end{aligned} \tag{4.6}$$

This signal is computed on only 3 sample pairs all 16 taps apart (using delayed IQ samples from *data_delay.vhd*), this mutual energy is delayed exactly 16 taps behind its corresponding *POWDATA_OUT_XCORR*, it is outputted as *STS_AUTOCORR_I_16_DELAYED* and *STS_AUTOCORR_Q_16_DELAYED*. This will enable us to add 48 sample pairs in just 16 taps in the following block

timing_acquisition_802_11p.vhd and use it to coarsely estimate frequency offset from STS sequence (see Equation 4.7).

Detection and timing acquisition

Block *timing_acquisition_802_11p.vhd* contains an instance of *Parallel_STS_FIR_Filter.vhd*, it accumulates its crosscorrelation absolute square output *POWDATA_OUT_XCORR* according to Equation 4.2, this is done in a shift register manner inside the process *shift_correlation_process* (the accumulator is 64-bits wide).

Having this amended crosscorrelation absolute square $|\tilde{\mathcal{R}}_{160}^{\mathrm{STS}}[n]|^2$ (delayed by additional 6.2 samples, the 0.1 delay originated due to buffering into the last *SHIFT_REGISTER*) we can use it for the 802.11p frame detection in process *detection_process*. This process implements a simple state machine, that compares mentioned crosscorrelation absolute square with threshold *DETECTION_THRESHOLD* (block input, see subsection 4.1.3), when the crosscorrelation absolute square exceeds the threshold, its value is saved into signal *MAX_XCORR* the and following 16 crosscorrelation absolute square samples are checked in following way (and order):

1. When any sample 1..16 $\geq$ *MAX_XCORR* $\Rightarrow$ Update *MAX_XCORR* and check following 16 crosscorrelation absolute square samples
2. When any sample 1..15 $\geq$ *MAX_XCORR*/2 $\Rightarrow$ Cancel detection
3. When sample 16 $\leq$ *MAX_XCORR*/2 $\Rightarrow$ Cancel detection

When the threshold was exceeded and all above mentioned conditions were passed, block output *DETECTION_SIGNAL_DETECTED* is raised to signalize 802.11p frame detection, output *DETECTION_XCORR* outputs the *MAX_XCORR*.

This state machine adds another 18 taps delay so the signal detection is delayed by 24.2 taps from the last sample $x[n]$ (or 183.2 samples after the beginning of the frame or 8.8 samples ahead of the beginning of the first LTS sequence).

Signals *DETECTION_STS_AUTOCORR_I* and *DETECTION_STS_AUTOCORR_Q* representing $\mathcal{E}_{16}^{\mathrm{STS},48}[n]$ are synchronized with their corresponding *DETECTION_SIGNAL_DETECTED*. It is computed from previously computed $\mathcal{E}_{16}^{\mathrm{STS},3}[n-16]$ in a way stated in the following equation (the computation takes place when the state machine above tests a detection, thus only for some n)

$$\mathcal{E}_{16}^{\mathrm{STS},48}[n] = \sum_{i=1}^{16} \mathcal{E}_{16}^{\mathrm{STS},3}[n-16+i] = \sum_{i=0}^{47} x[n-i]\bar{x}[n-i-16] \tag{4.7}$$

Above mentioned signals will be needed to provide its phase for the coarse frequency offset estimation (see subsection 4.2.3), even in a case of a weak signal, therefore they were implemented with a higher precision of 36 bits.

4.2.3 Frequency offset estimation and correction

Block *equalizer_time_frequency.vhd* is a first more elaborate block, it uses detection signal to start reception, with help of *atan_block.vhd* and *rotation_block.vhd* it computes and equalizes the frequency offset (coarsely from STS first, then finely from LTS), meanwhile, it feeds ofdm symbols to the FFT block *fft_ofdm.vhd* and watches input signal *VITERBI_RX_ENDED* for the end of the reception from the decoder (see subsection 4.2.7).

The core of *equalizer_time_frequency.vhd* block is a state machine implemented in processes *RX_state_machine* and *RX_outputs*. This state machine can be described by following states (following each other):

1. *IDLE*: Wait for 802.11p frame detection signalized by *DETECTION_SIGNAL_DETECTED* and save current STS mutual energy
$$\mathcal{E}_{16}^{\mathrm{STS},48} = \mathcal{E}_{16}^{\mathrm{STS},48}[n] \tag{4.8}$$
2. *STS_ATAN_INIT*: Feed *atan_block.vhd* with the STS mutual energy from the detection $\mathcal{E}_{16}^{\mathrm{STS},48}$
3. *STS_ATAN_WAIT*: Wait for the output $\arg(\mathcal{E}_{16}^{\mathrm{STS},48})$ of *atan_block.vhd*
4. *SET_ROTATION_BLOCK*: compute coarse frequency offset estimation $\hat{\alpha}^{\mathrm{STS}}$ from *atan_block.vhd* output (in phase change per one sample) and set *rotation_block.vhd* to equalize it [12]
$$\hat{\alpha}^{\mathrm{STS}} = \frac{1}{16}\arg(\mathcal{E}_{16}^{\mathrm{STS},48}) \tag{4.9}$$
5. *WAIT_FOR_LTS_MARKER*: Wait for the 'first' (because of OFDM symbols windowing, we decided to start each OFDM symbol with the last IQ sample of its cyclic prefix) LTS sequence IQ sample marked in the process *IQ_counter_process* and passed through *rotation_block.vhd* (LTS has already coarsely synchronized frequency offset)
6. *RECEIVE_LTS*: Send both LTS sequences to *fft_ofdm.vhd* (mark the first LTS OFDM symbol with *FFT_DATA_IN_FIRST_SYMBOL_MARKER*) and accumulate their mutual energy $\mathcal{E}_{64}^{\mathrm{STS},64}$ (similar to STS 16-tap mutual energy, but 64-tap on LTS)
$$\mathcal{E}_{64}^{\mathrm{LTS},64} = \sum_{i=0}^{64} x[n-i]\bar{x}[n-i-64] \tag{4.10}$$
7. *RECEIVE_DATA*: Ignore cyclic prefixes and send each data OFDM symbol to *fft_ofdm.vhd* (again start at the last prefix sample). When LTS mutual energy $\mathcal{E}_{64}^{\mathrm{LTS},64}$ is computed feed it into the *atan_block.vhd*, when *atan_block.vhd* output is valid compute the fine frequency offset estimation $\hat{\alpha}^{\mathrm{LTS}}$ and set the *rotation_block.vhd* to equalize this updated offset $\hat{\alpha}^{\mathrm{STS}} + \hat{\alpha}^{\mathrm{LTS}}$. This state can be interrupted by *STOP_RX_DONE*

from the decoder or by completing the last possible (1366th) OFDM symbol.

$$\hat{\alpha}^{\mathrm{LTS}} = \frac{1}{64} \arg(\mathcal{E}_{64}^{\mathrm{LTS},64}) \tag{4.11}$$

atan_block.vhd

Block *atan_block.vhd* provides buffering and custom interface for Vivado IP block *CORDIC* set to Arc Tan mode. In theory, this block could be omitted without difficulty.

Used Vivado IP block *CORDIC v6.0* (component name: *hier_atan/cordic_0*) inputs complex signed data and outputs its phase in scaled radians format. To reach this we used following configuration (important fields only):

- Configuration Options –> Functional Selection –> *Arc Tan*
- Configuration Options –> Architectural Configuration –> *Word Serial*
- Configuration Options –> Phase Format –> *Scaled Radians*
- Configuration Options –> Input Width –> 36
- Configuration Options –> Output Width –> 20
- Configuration Options –> Coarse rotation –> tick

Architectural Configuration *Word Serial* creates a serial CORDIC implementation saving resources, however, input data cannot be fed each clock cycle. Phase Format *Scaled Radians* is a fixed point format 2QN (twos complement number with an integer width of 3 bits) where value 1 corresponds to phase π. Ticking *Coarse rotation* allows the input to be in any quadrant.

rotation_block.vhd

Similarly as *atan_block.vhd* the *rotation_block.vhd* provides buffering and custom interface for Vivado IP block *CORDIC*, however, set to Rotate mode. On top of that, it features the buffered input *ROTATION_PHASE_NEW_DIFF* that is used for phase incrementing with each IQ sample.

Used Vivado IP block *CORDIC v6.0* (component name: *hier_rotation/cordic_0*) inputs complex signed data with a phase in scaled radians format and outputs those input complex data rotated by the given phase. To reach this we used following configuration (important fields only):

- Configuration Options –> Functional Selection –> *Rotate*
- Configuration Options –> Architectural Configuration –> *Parallel*
- Configuration Options –> Phase Format –> *Scaled Radians*
- Configuration Options –> Input Width –> 16

- Configuration –> Output Width –> 16
- Configuration –> Coarse rotation –> tick
- AXI4 Stream Options –> Cartesian Channel Options –> Has TLAST –> tick

Architectural Configuration *Parallel* creates a parallel pipelined CORDIC implementation capable of processing new input data each clock cycle. Phase Format *Scaled Radians* is a fixed point format 2QN (twos complement number with an integer width of 3 bits) where value 1 corresponds to phase π. Ticking *Coarse rotation* allows the input to be in any quadrant. We will use AXI4 *TLAST* signal as data marker *ROTATION_DATA_IN_MARKER* and *ROTATION_DATA_OUT_MARKER* (the *TLAST* signal is passed alongside the computed data and has no effect on the CORDIC block data exchange).

4.2.4 OFDM FFT Demodulator

Block *fft_ofdm.vhd* uses equalized (in time and frequency) IQ samples from the *equalizer_time_frequency.vhd* block and performs OFDM demodulation by computing FFT from these IQ samples.

The functionality is divided into three processes. The first process, *move_input_buffers_process*, buffers 64 IQ samples at 10 MHz (from the *equalizer_time_frequency.vhd* block). When the last of 64 IQ samples is buffered, the second process, *move_buffers_to_fft_process*, moves this entire buffer into the Vivado FFT IP block (see below). The third process, *output_process* buffers and outputs the Vivado FFT IP block output and adds *FFT_DATA_OUT_FIRST_SYMBOL_MARKER* marker to the first LTS OFDM symbol.

Let us denote this OFDM demodulated by $X_k[m]$ where $k = -32, ..., 31$ represents the subcarrier index and m=$-2, ..., 1363$ represents the OFDM symbol index (indexes $m = -2$ and $m = -1$ represent two LTS symbols).

The *fft_ofdm.vhd* block output can be displayed in the GUI (see Figure 5.4 and Figure 5.5).

Vivado FFT IP block

Vivado IP block *Fast Fourier Transform (9.1)* (component name: *hier_fft_ofdm/xfft_0*) inputs 64 (2x16-bit) complex signed data samples and outputs its Discrete Fourier Transform image of 64 (2x23-bit) complex signed data samples. To reach this we used following configuration (important fields only):

- Configuration –> Number of Channels –> 1
- Configuration –> Transform Length –> 64
- Configuration –> Target Clock Frequency –> 100 MHz

- Configuration –> Target Data Throughput –> 10 MSPS
- Implementation –> Data Format –> AUTO: *Fixed Point*
- Implementation –> Scaling Options –> *Unscaled*
- Implementation –> Input Data Width –> AUTO: 16
- Implementation –> Control Signals –> tick *ARESETn*
- Implementation –> Output Ordering –> *Natural Order*
- Implementation –> Throttle Scheme –> *Real Time*

Target Data Throughput of 10 MSPS optimizes the architecture for 10 MHz sampling, the output data are available before a new input data in each computational cycle. The *Fixed Point* and Data Format with *Unscaled* option will enlarge the data output width by 7 to 23 bits (however, we will use 24-bit signals).

4.2.5 Channel response estimation and tracking

Next more elaborate block is *constellation_tracker.vhd* block. This block uses two (OFDM demodulated) initial LTS sequences to create a frequency selective channel estimate (phase and gain for each subcarrier), then it updates (tracks) this estimate using available pilot subcarriers. The tracked channel phase estimate is used for (OFDM demodulated) data subcarriers equalization, the corresponding channel gain estimate (not tracked) is outputted alongside the phase-equalized data subcarrier. For channel phase and gain estimate computation we use *atan_constellation_block.vhd* block, for phase equalization we use *rotation_constellation_block.vhd* block.

The core of the *constellation_tracker.vhd* block is a state machine that can be described by following states (following each other):

1. *IDLE*: Wait for the first LTS from the *fft_ofdm.vhd* block (input: *FFT_DATA_IN_FIRST_SYMBOL_MARKER*)
2. *RX_LTS_FIRST*: Negate inverted LTS subcarriers (see $S^{\mathrm{LTS}}_{-26,26}$ in Equation 2.2) and buffer the first LTS (all 52 used subcarriers)
3. *RX_LTS_SECOND*: Average the second LTS with the first LTS and send all these 52 subcarriers sums (or channel estimates $\hat{H}_k[0]$) to the *atan_constellation_block.vhd* block.

$$\hat{H}_k[0] = \frac{X_k[-2] + X_k[-1]}{2}, \quad k = -26, .., -1, 1, ..., 26 \tag{4.12}$$

Then receive and save channel estimate (phase $\arg(\hat{H}_k[0])$ and gain $|\hat{H}_k[0]|$) for each subcarrier $k = -26, ... - 1, 1, ..., 26$ into *CHANNEL_RESPONSE_PHASE* and *CHANNEL_RESPONSE_AMPLITUDE*.

4. *RX_DATA*: Use *rotation_constellation_block.vhd* to equalize (rotate back) the incoming data by *CHANNEL_RESPONSE_PHASE* according to their subcarrier. Then output the phase-equalized subcarriers (let us denote them $Y_k[m]$) with their respective channel gain $|\hat{H}_k[0]|$ (can be called BPSK amplitude reference, it is not tracked) *CHANNEL_RESPONSE_AMPLITUDE* in subcarrier order $k = -26$ first, $k = +26$ last.

$$Y_k[m] = X_k[m]e^{-\mathrm{j}\arg(\hat{H}_k[m])}, \quad k = -26, .., -1, 1, ..., 26, \quad m = 0, ...1363, \tag{4.13}$$

Meanwhile, accumulate all four (equalized) Pilot subcarriers according to their scrambled polarities (127-bit long scrambling sequence is saved in *PILOT_POLARITIES*, see subsection 2.4.6) and send the result $P[m]$ to *atan_constellation_block.vhd* block.

$$P[m] = \pm(Y_{-21}[m] + Y_{-7}[m] + Y_7[m] - Y_{21}[m]) \tag{4.14}$$

Then receive the pilot phase $\arg(P[m])$ and rotate (track) all elements of *CHANNEL_RESPONSE_PHASE* to cancel this pilot phase error.

$$\arg(\hat{H}_k[m+1]) = \arg(\hat{H}_k[m])e^{\mathrm{j}\arg(P[m])} \tag{4.15}$$

The *constellation_tracker.vhd* block output can be displayed in the GUI (see Figure 5.6 and Figure 5.7).

atan_constellation_block.vhd

Block *atan_constellation_block.vhd* provides buffering and custom interface for Vivado IP block *CORDIC* set to Translate mode. In theory, this block could be omitted without difficulty.

Used Vivado IP block *CORDIC v6.0* (component name: *hier_atan_constellation/cordic_0*) inputs complex signed data and outputs its phase in scaled radians format and amplitude (the difference from *Arc Tan* mode). To reach this we used following configuration (important fields only):

- Configuration Options –> Functional Selection –> *Translate*
- Configuration Options –> Architectural Configuration –> *Parallel*
- Configuration Options –> Phase Format –> *Scaled Radians*
- Configuration Options –> Input Width –> 24
- Configuration Options –> Output Width –> 24
- Configuration Options –> Coarse rotation –> tick
- AXI4 Stream Options –> Cartesian Channel Options –> Has TUSER –> tick

- AXI4 Stream Options –> Cartesian Channel Options –> TUSER Width –> 6

Architectural Configuration *Parallel* creates a parallel pipelined CORDIC implementation capable of processing new input data each clock cycle. Phase Format *Scaled Radians* is a fixed point format 2QN (twos complement number with an integer width of 3 bits) where value 1 corresponds to phase π. Ticking *Coarse rotation* allows the input to be in any quadrant. The TUSER data is passed alongside the cartesian input data and outputted nonchanged with corresponding translated data, we will use this data as a subcarrier counter.

rotation_constellation_block.vhd

Block *rotation_constellation_block.vhd* provides buffering and custom interface for Vivado IP block *CORDIC* set to Rotate mode. In theory, this block could be omitted without difficulty.

Used Vivado IP block *CORDIC v6.0* (component name: *hier_rotation_constellation/cordic_0*) inputs complex signed data with a phase in scaled radians format and outputs those input complex data rotated by the given phase. To reach this we used following configuration (important fields only):

- Configuration Options –> Functional Selection –> *Rotate*
- Configuration Options –> Architectural Configuration –> *Parallel*
- Configuration Options –> Phase Format –> *Scaled Radians*
- Configuration Options –> Input Width –> 24
- Configuration Options –> Output Width –> 24
- Configuration Options –> Coarse rotation –> tick
- AXI4 Stream Options –> Cartesian Channel Options –> Has TUSER –> tick
- AXI4 Stream Options –> Cartesian Channel Options –> TUSER Width –> 6

Architectural Configuration *Parallel* creates a parallel pipelined CORDIC implementation capable of processing new input data each clock cycle. Phase Format *Scaled Radians* is a fixed point format 2QN (twos complement number with an integer width of 3 bits) where value 1 corresponds to phase π. Ticking *Coarse rotation* allows the input to be in any quadrant. The TUSER data is passed alongside the cartesian input data and outputted nonchanged with corresponding rotated data, we will use this data as a subcarrier counter.

4.2.6 Modulation demapping and data deinterleaving

In this point the phase equalized data from *constellation_tracker.vhd* can be perceived as constellation points, thus they are prepared for the modulation demapping, this is done in *demapper_soft.vhd*. This block receives equalized subcarriers $Y_k[m]$ with their respective BPSK amplitude reference $\hat{H}\_k[m]|$ (computed from LTS) one by one. These subcarriers are demapped to bits according to Figure 2.8 and a 2-bit heuristic distance $\rho \in \{0, 1, 2\}$ (we reserve $\rho = 3$ for uncertainty) is computed for each demapped bit, both done by comparing the subcarriers to a set of thresholds scaled by their respective BPSK amplitude reference. This process is done for all BPSK, QPSK and 16-QAM modulations (modulation 64-QAM is not supported in this receiver, this should not be an essential problem because it does not have to be supported according to the IEEE 802.11p, see section 2.4). The resulting data bits and their respective 2-bit distances are parallelized for each OFDM symbol.

These demapped bit data with their corresponding distances are then deinterleaved by *deinterleaver_soft.vhd* block. The deinterleaving tables were precomputed in MATLAB script *deinterleaver_vhd.m* for all three modulations (by numerically inverting permutations Equation 2.4 and Equation 2.5). This small block deinterleaves all data subcarriers in a single clock.

4.2.7 Soft Viterbi decoder

The last more elaborate block is *viterbi_soft.vhd*. This block takes demapped and deinterleaved data with their demapped (heuristic) distances and performs soft Viterbi decoding. This includes decoding the SIGNAL field first with code rate and modulation selection for successive data.

The core of the *viterbi_soft.vhd* block is process *viterbi_process*. This process is updated with each new data indicated *VITERBI_INPUT_VALID* and performs following operations:

1. Compute (heuristic) soft distance for all 64 incoming states for both incoming paths: $\Delta\rho_{i,0}$, $\Delta\rho_{i,1}$, $i = 0, ..., 63$. These distances are computed between received coded (demapped) bits and coded bits generated by given state and its incoming path. Each bit contributes to the distance according to the match or mismatch.

$$\Delta\rho_{i,j} = \begin{cases} \rho, & \text{if match} \\ 6 - \rho, & \text{otherwise} \end{cases} \tag{4.16}$$

   The results are saved to *PATH_0_SOFT_DISTANCE*(*state*) and *PATH_0_SOFT_DISTANCE*(*state*).

2. Accumulate current state distance for all 64 states into ρ_i or *STATE_DISTANCE* registers by selecting the incoming path with lower accumulated distance. According the incoming path selection, fill zeroth bit into each *STATE_TRACEBACK_REGISTERS*(*state*) and shift other (older) bits to the right.

3. In three steps, find minimal value of *STATE_DISTANCE* and save its corresponding last traceback bit. The minimal value is found in a tree structure where each step narrows the possibilities by four.

4. When the *viterbi_process* is filled with enough input coded bits, output the decoded (traceback) bit into *VITERBI_OUTPUT* (marked by *VITERBI_OUTPUT_VALID*) with each new input

With control signal *VITERBI_RESET* all Viterbi *STATE_DISTANCE* accumulators are reset to zero and the *viterbi_process* is ready to start new decoding.

The Viterbi core is controlled by a state machine located in processes *state_update_process* and *input_output_process*. This state machine can be described by its following states (following each other):

1. *IDLE*: Wait for first deinterleaved data block (marked by input signal *DEINTERLEAVER_START_MARKER*)

2. *RX_SIGNAL*: Feed the *viterbi_process* by BPSK demapped data from this first *SIGNAL* OFDM symbol (without depuncturing, code rate R=1/2, see subsection 2.4.1), then feed the *viterbi_process* by zeros with zero distance until all 24 decoded *SIGNAL* bits are outputted from this process, buffer this *SIGNAL* field to *VITERBI_SIGNAL_OUTPUT_BUFFER* register. Meanwhile, compute parity for first 16 *SIGNAL* bits, if odd raise *VITERBI_RX_ENDED* output signal (used by *equalizer_time_frequency.vhd* and *descrambler.vhd* blocks) and go to *IDLE*.

3. *PROCESS_RATE*: In this one-clock state, use the *RATE* field *VITERBI_SIGNAL_OUTPUT_BUFFER*(31 downto 28) to decode parameters needed for further reception, these are: *MODULATION*, *CODE_RATE*, number of bits per symbol *N_CBPS* (coded bits) and *N_DBPS* (data bits). If *RATE* field is unknown or unsupported, raise *VITERBI_RX_ENDED* output signal and go to *IDLE*. Also decode length of the transmitted message *LENGTH_BYTES* from *VITERBI_SIGNAL_OUTPUT_BUFFER*(26 downto 15). Also reset the *viterbi_process* by raising *VITERBI_RESET* signal.

4. *WAIT_FOR_DATA*: Wait for data OFDM symblol. Enable *viterbi_process* but do not feed it.

5. *RX_DATA*: Feed *viterbi_process* with appropriate data according to *MODULATION* and *CODE_RATE* (depuncuring is done by inserting zero bits with neutral heuristic distance of 3 that translates to uncertainty). There is hidden a bug in the depuncturing since it works in Behavioral simulation only, see section 4.4). Forward the decoded data to output *VITERBI_DECODED_OUTPUT*. When not entire message received in the current OFDM block, go to *WAIT_FOR_DATA*.

6. *END_DECODING*: Feed the *viterbi_process* by zeros with zero distance until all decoded *SIGNAL* bits are outputted from this process. then raise *VITERBI_RX_ENDED* signal and go to *IDLE*.

4.2.8 Data descrambler and output parallelization

The decoded data descrambling is implemented in *descrambler.vhd* block. This small block implements descrambler identical to the scrambler in Figure 2.6.

As stated in subsection 2.4.2, the descrambler is initialized by first seven bits of the decoded data (the first bits location are found by observing *VITERBI_SIGNAL_VALID* output control signal from the *viterbi_soft.vhd* block). After initialization, the descrambler generates its descrambling bit sequence that is XORed with the received data. When the input signal *VITERBI_RX_ENDED* is detected, stop descrambling and generate pulse at *DESCRAMBLED_OUTPUT_LAST* output signal with the last descrambled bit.

The descrambled stream of bits is then parallelized into 32-bit registers in the *output_ser2par.vhd* block. The start of this stream is also synchronized by the *VITERBI_SIGNAL_VALID* output control signal from the *viterbi_soft.vhd* block and the end of the stream by the *descrambler.vhd* block control signal *DESCRAMBLED_OUTPUT_LAST*, the remaining empty register part is appended with zeros when the last data bit comes.

4.2.9 PL writing to BRAM

The last block in our block design is *axi_regs_mux.vhd*. Its function is to multiplex different data from the PL design, assign them an address (output *FPGA_REG_WRITE_ADDRESS*), and forwarding them into the BRAM as *FPGA_REG_WRITE_DATA* output (synchronized by *FPGA_REG_WRITE_STROBE*).

The data sources should not involve any serious conflicts (timing simulated for all combinations except IQ samples output set by *SELECT_AXI_REGS_MODE* = 1), however, they they are outputted with following priority (descending):

1. *registers*[3]: STS Coarse frequency offset from *equalizer_time_frequency.vhd* block (scaled radians per one IQ sample in signed(19 downto 0) format)

2. *registers*[4]: Additional LTS Fine frequency offset from *equalizer_time_frequency.vhd* block (scaled radians per one IQ sample in signed(19 downto 0) format)

3. *registers*[5]: Started receptions counter (counts frames by counting LTS offset computations) stored in *START_PROCESSING_CNTR* (signed(31 downto 0) format)

4. *registers*[14]: Done decoding counter (counts frames by counting last parallel decoded outputs from *output_ser2par.vhd* block) stored in *DECODED_OUTPUTS_CNTR* (signed(31 downto 0) format)

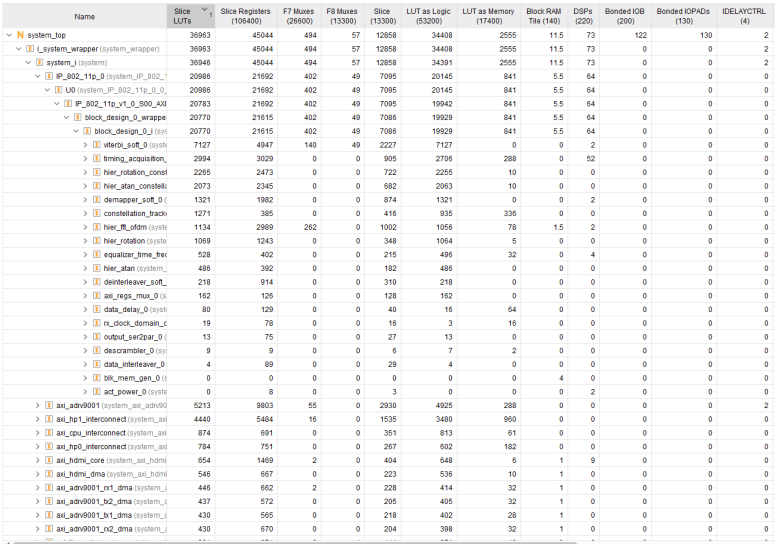
5. *registers*[15]: *SIGNAL* field from *viterbi_soft.vhd* (orientation: *SIGNAL*(0) bit at *FPGA_REG_WRITE_DATA*(31) bit)

6. *registers*[16:4094]: Data selected by the *MODE* input, address is reset with each STS Coarse frequency offset write and incremented until 4095 reached (this last address is not used for writing). Modes from 1 to 5 follow format of two signed(15 downto 0) where *FPGA_REG_WRITE_DATA*(31 downto 16) represents the imaginary part. The data selection by the *SELECT_AXI_REGS_MODE* (referenced as *MODE]*) input follows:
   a. *MODE* = 0: No data written
   b. *MODE* = 1: Original IQ samples, synchronized after STS Coarse frequency offset computation (starting at 1-7 last IQ samples of the LTS prefix, depends on delay from *atan_block.vhd*)
   c. *MODE* = 2: OFDM demodulated subcarriers (all 64 subcarriers; including both LTS sequences, all 64 subcarriers) from *fft_ofdm.vhd* block (16/24 MSB)
   d. *MODE* = 3: OFDM demodulated subcarriers (all 64 subcarriers; including both LTS sequences) from *fft_ofdm.vhd* block (16/24 LSB with sign and out of limit value limitation)
   e. *MODE* = 4: Phase equalized (OFDM demodulated) subcarriers (all used 52 subcarriers; excluding LTS sequences) from *constellation_tracker.vhd* block (16/24 MSB)
   f. *MODE* = 5: Phase equalized (OFDM demodulated) subcarriers (all used 52 subcarriers; excluding LTS sequences) from *constellation_tracker.vhd* block (16/24 LSB with sign and out of limit value limitation)
   g. *MODE* = 6: BPSK demapped subcarriers (including pilots) from the *demapper_soft.vhd* block (every two registers represent one OFDM symbol, each odd register is filled with 16 LSB zeros)
   h. *MODE* = 7: QPSK demapped subcarriers (including pilots) from the *demapper_soft.vhd* block (every three registers represent one OFDM symbol)
   i. *MODE* = 8: 16-QAM demapped subcarriers (including pilots) from the *demapper_soft.vhd* block (every six registers represent one OFDM symbol)
   j. *MODE* = 9: Decoded and descrambled data from *output_ser2par.vhd* block

axi_regs_mux.vhd connection to BRAM

As mentioned in the previous paragraph, the *axi_regs_mux.vhd* block outputs a simple one-directional bus of strobe, address and data. These signals are driven into the *./src_HDL/IP_802_11p/IP_802_11p_1_0/hdl/IP_802_11p_v1_0_S00_AXI.vhd* AXI4 Lite block where they are connected to the BRAM write port A. The connection to the BRAM port A is done by multiplexing the AXI4 Lite writing from the PS part (see section 4.1) and this *FPGA_REG_WRITE_DATA* writing, the AXI4 writing is prioritized.

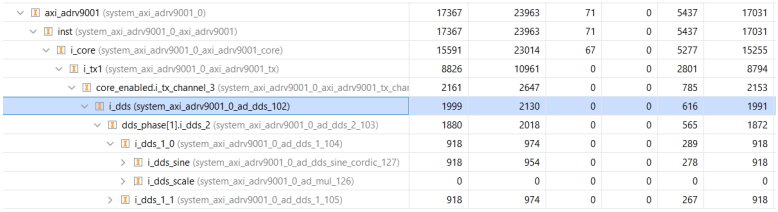
4.3 Hardware utilization



Name	Slice LUTs	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Block RAM Tile (140)	DSPs (220)	Bonded IOB (200)	Bonded IOPADs (130)	IDELAYCTRL (4)
system_top	36963	45044	494	57	12858	34408	2555	11.5	73	122	130	2
i_system_wrapper (system_wrapper)	36963	45044	494	57	12858	34408	2555	11.5	73	0	0	2
system_i (system)	36946	45044	494	57	12858	34391	2555	11.5	73	0	0	2
IP_802_11p_0 (system_IP_802_1	20986	21692	402	49	7095	20145	841	5.5	64	0	0	0
U0 (system_IP_802_11p_0_0_	20986	21692	402	49	7095	20145	841	5.5	64	0	0	0
IP_802_11p_v1_0_S00_AXI	20783	21692	402	49	7095	19942	841	5.5	64	0	0	0
block_design_0_wrappe	20770	21615	402	49	7086	19929	841	5.5	64	0	0	0
block_design_0_i (sy	20770	21615	402	49	7086	19929	841	5.5	64	0	0	0
viterbi_soft_0 (syst	7127	4947	140	49	2227	7127	0	0	2	0	0	0
timing_acquisition_	2994	3029	0	0	905	2706	288	0	52	0	0	0
hier_rotation_const	2265	2473	0	0	722	2255	10	0	0	0	0	0
hier_atan_constell	2073	2345	0	0	682	2063	10	0	0	0	0	0
demapper_soft_0 (	1321	1982	0	0	874	1321	0	0	2	0	0	0
constellation_track	1271	385	0	0	416	935	336	0	0	0	0	0
hier_fft_ofdm (syst	1134	2989	262	0	1002	1056	78	1.5	2	0	0	0
hier_rotation (syste	1069	1243	0	0	348	1064	5	0	0	0	0	0
equalizer_time_frec	528	402	0	0	215	496	32	0	4	0	0	0
hier_atan (system_	486	392	0	0	182	486	0	0	0	0	0	0
deinterleaver_soft_	218	914	0	0	310	218	0	0	0	0	0	0
axi_regs_mux_0 (s	162	126	0	0	128	162	0	0	0	0	0	0
data_delay_0 (syst	80	129	0	0	40	16	64	0	0	0	0	0
rx_clock_domain_c	19	78	0	0	16	3	16	0	0	0	0	0
output_ser2par_0 (	13	75	0	0	27	13	0	0	0	0	0	0
descrambler_0 (sy	9	9	0	0	6	7	2	0	0	0	0	0
data_interleaver_0	4	89	0	0	29	4	0	0	0	0	0	0
blk_mem_gen_0 (s	0	0	0	0	0	0	0	4	0	0	0	0
act_power_0 (syste	0	8	0	0	3	0	0	0	2	0	0	0
axi_adrv9001 (system_axi_adrv90	5213	9803	55	0	2930	4925	288	0	0	0	0	2
axi_hp1_interconnect (system_axi	4440	5484	16	0	1535	3480	960	0	0	0	0	0
axi_cpu_interconnect (system_axi	874	691	0	0	351	813	61	0	0	0	0	0
axi_hp0_interconnect (system_axi	784	751	0	0	267	602	182	0	0	0	0	0
axi_hdmi_core (system_axi_hdmi	654	1469	2	0	404	648	6	1	9	0	0	0
axi_hdmi_dma (system_axi_hdmi	546	667	0	0	223	536	10	1	0	0	0	0
axi_adrv9001_rx1_dma (system_	446	662	2	0	228	414	32	1	0	0	0	0
axi_adrv9001_tx2_dma (system_	437	572	0	0	205	405	32	1	0	0	0	0
axi_adrv9001_tx1_dma (system_	430	565	0	0	218	402	28	1	0	0	0	0
axi_adrv9001_rx2_dma (system_	430	670	0	0	204	398	32	1	0	0	0	0

Figure 4.5: FPGA PL part utilization (Implemented design)

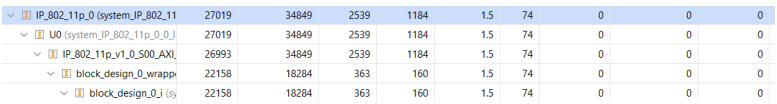
As mentioned in subsection 3.3.1 we removed the DDS. This *ad_dds* block occupied a lot of resources and was present in each of six transmission channels. For details see Figure 4.6.



axi_adrv9001 (system_axi_adrv9001_0)	17367	23963	71	0	5437	17031
inst (system_axi_adrv9001_0_axi_adrv9001)	17367	23963	71	0	5437	17031
i_core (system_axi_adrv9001_0_axi_adrv9001_core)	15591	23014	67	0	5277	15255
i_tx1 (system_axi_adrv9001_0_axi_adrv9001_tx)	8826	10961	0	0	2801	8794
core_enabled.i_tx_channel_3 (system_axi_adrv9001_0_axi_adrv9001_tx_cha	2161	2647	0	0	785	2153
i_dds (system_axi_adrv9001_0_ad_dds_102)	1999	2130	0	0	616	1991
dds_phase[1].i_dds_2 (system_axi_adrv9001_0_ad_dds_2_103)	1880	2018	0	0	565	1872
i_dds_1_0 (system_axi_adrv9001_0_ad_dds_1_104)	918	974	0	0	289	918
i_dds_sine (system_axi_adrv9001_0_ad_dds_sine_cordic_127)	918	954	0	0	278	918
i_dds_scale (system_axi_adrv9001_0_ad_mul_126)	0	0	0	0	0	0
i_dds_1_1 (system_axi_adrv9001_0_ad_dds_1_105)	918	974	0	0	267	918

Figure 4.6: DDS utilization (6x)

At of creating the IP we replaced default AXI Lite registers by BRAM (see subsection 4.1.2), the reason was again that a lot of registers occupied a lot of resources. For details (with 512 AXI registers see Figure 4.7).



IP_802_11p_0 (system_IP_802_11	27019	34849	2539	1184	1.5	74	0	0	0
U0 (system_IP_802_11p_0_0_	27019	34849	2539	1184	1.5	74	0	0	0
IP_802_11p_v1_0_S00_AXI_	26993	34849	2539	1184	1.5	74	0	0	0
block_design_0_wrapp	22158	18284	363	160	1.5	74	0	0	0
block_design_0_i (sy	22158	18284	363	160	1.5	74	0	0	0

Figure 4.7: DDS AXI4 registers (512)

4.4 Problems and possible improvements

Depuncturing not working

The depuncturing (in *viterbi_soft.vhd*) does not work anywhere after the synthesis (no matter how the synthesis is set), however, the Behavioral simulation returns correct decoded outputs (simulated for BPSK, R=3/4). There were not found any errors nor warning concerning the synthesis of *viterbi_soft.vhd*. This could be caused by an ambiguous description in the design, that was grasped right by the simulator, however, we were not able to find the error in the Post synthesis functional simulation. It can be seen in Figure 5.9.

With an error in depuncturing we can use only rate R=1/2, meaning data rates of the 3, 6, and 12 Mb/s (those required ones). These three data rates were all successfully tested.

Naming conventions shifted during the design

Some strobe signals should be renamed, some inputs to outputs and vice versa to valid signals to comply with the axi naming conventions.

Multiple Block designs

In the beginning of the design we got into a trouble with Vivado. Behavioral simulations of the *atan_block.vhd* were alright, however, the implementation and sometimes even the synthesis were failing (error of type: No formal port,). Then we found out that the Vivado 2023.2.2 does not support multiple Block designs in one project, IP cores does not count. You can explore git branch *multiple_bd_errors*.

FFT Vivado block TLAST input

The Vivado FFT block *hier_fft_ofdm/xfft_0* regularly raises error signal of missing and unexpected AXI4 Stream TLAST input. The design was checked and nothing was found, however the values are computed right. (There is a possibility that one sample is not inputted right.)

IEEE 802.11a expanding

The IEEE 802.11a differs in its bouble bandwidth achieved double sampling rate. The processing would work for IEEE 802.11a if the *CLOCK* and *S_AXI_ACLK* could double their frequency. If this would not be possible or beneficial, the FFT Vivado FFT block *hier_fft_ofdm/xfft_0* would have to be reconfigured to higher *Target Data Throughput*. It is not excluded that any different block in the design would not stop meeting its deadlines, however, the majority should be transferable to double IQ data rate.

Realtime receptions

An interrupt and DMA could be implemented to transfer data from the *IP_802_11p* PL block to the PS part. This would shorten intervals of frame reception while freeing the PS.

BRAM multiplexing seems not to be fully stable

BRAM port A PL/PS writing multiplex is probably not the best option since we had to make it without buffering. Sometimes BRAM addresses 0 and 1 are randomly overwritten. (It does not affect the *RESET* and *DETECTION_THRESHOLD* registers). One solution could be to use a True-dual port RAM with bigger AXI4 Lite signals modification, another option would be to use Vivado IP block *AXI BRAM Controller*.

There is a non-optimality in time or frequency acquisition

The threshold is implemented from BRAM *registers*[1] (*DETECTION_THRESHOLD*) and not automatic from power and noise power by hypothesis testing.

The synchronization starts failing at sufficiently strong signal. The error is most likely in frequency synchronization or equalization since no matter of the *registers*[1] (*DETECTION_THRESHOLD*) the frequency synchronization fails for signal in Figure 4.8 and Figure 4.9.

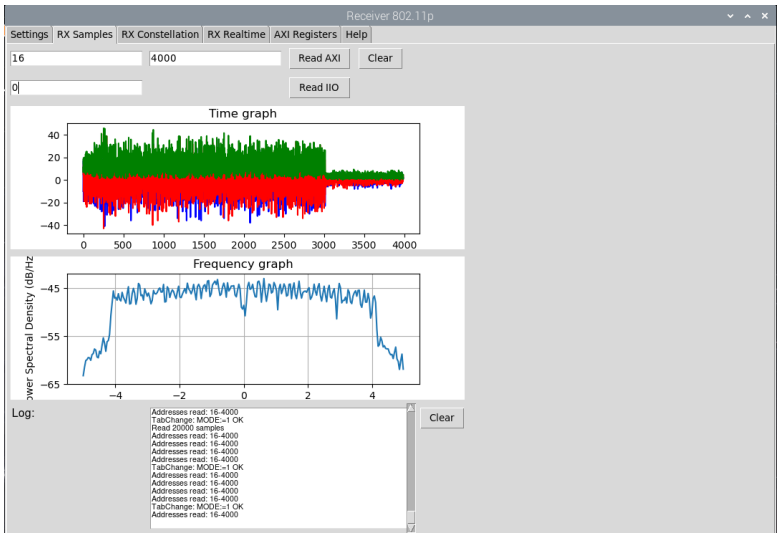


Figure 4.8: Failing level of synchronization (IQ)

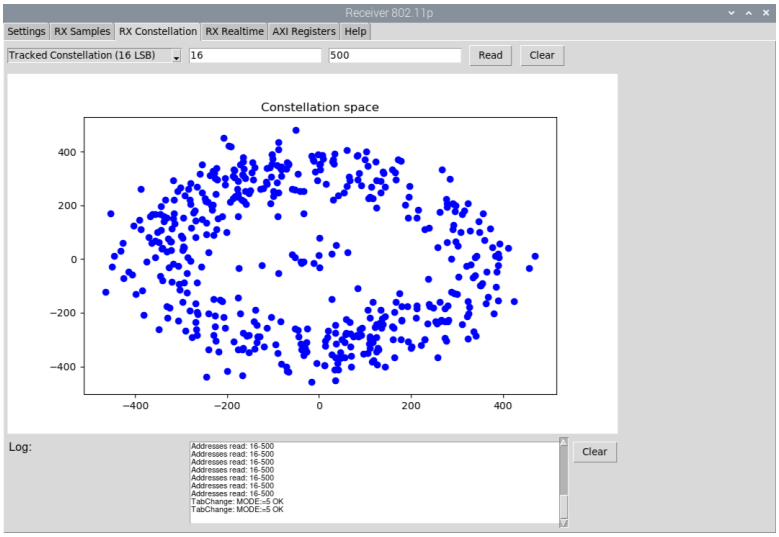


Figure 4.9: Failing level of synchronization (Constellation)

Chapter 5

GUI for IEEE 802.11p hardware block

This GUI (Graphical User Interface) is a custom application for controlling the ADRV9002 via Libiio and our custom *IP_802_11p* PL block (see chapter 4) via AXI4 Lite interface.

The GUI is written completely in Python using mostly *tkinter* standard Python interface library. The GUI can be found in folder *./others/gui_axi/* with main file *GUI_axi.py*.

For enabling all of its features the GUI must be run directly in the ADI Kuiper Linux on ZedBoard with our *IP_802_11p* PL block. However, it can be running on any PC (having standard libraries and ADI *py-adi* library), this will enable Libiio functionalities by remotely connecting to a ZedBoard. In both cases the ZedBoard IP address (*IIO_uri*) has to be set in the *GUI_axi.py* file.

5.1 ZedBoard GUI interfaces

5.1.1 AXI4 Lite interface

As mentioned in subsection 4.1.2, the AXI4 Lite interface at address range 0x5000_0000..0x5000_3FFF is connected to the 32-bit wide BRAM memory of length 4096. This memory (in the design sometimes wrongly called registers) can be accessed either via AXI4 Lite from the PS (the purpose of this GUI) or from the *IP_802_11p* PL block (see subsection 4.2.9), therefore, we will use it as a connection to the *IP_802_11p* block. For the (BRAM) register map see subsection 4.2.9.

Low level BRAM read and write python functions are implemented in *access_axi_regs.py*, this file is imported into the GUI.

Function *axi_read_regs* takes address range (from 0 to 4095) to read this range from the BRAM memory. The output is a *numpy.array* object of *numpy.uint32* datatype. For each reading the */dev/mem* device memory is opened by *os.open* function, mapped by *mmap.mmap* function at range 0x5000_0000..0x5000_3FFF and read with offset according to the read address. This approach requires root privileges.

Function *axi_write_regs* takes write start address and a *numpy.array* object of *numpy.uint32* datatype, this entire array is then written to the BRAM

starting with its zeroth element at the start address. The mapping approach is identical to the *axi_read_regs* function in the preceding paragraph.

5.1.2 Libiio inteface

The Libiio library (see subsection 3.2.4) shares no connection to our *IP_802_11p* PL block. On the other hand, it can use many features of the original ADI HDL design. Therefore, we can take advantage of it in the GUI (called directly from *GUI_axi.py*) alongside the AXI4 Lite communication. Its major advantages are a straightforward ADRV9002 control as well as possibility of remote connection. As already said, no matter of local or remote connection, the ZedBoard IP address (*IIO_uri*) has to be set in *GUI_axi.py* code before starting.

We will use the Libiio (specifically *py-adi* library) for ADRV9002:

- Profile loading
- Setting carrier frequencies and gains
- Large asynchronous IQ samples reads
- Transmitting 802.11p test packets

5.2 Tabs description

The GUI is composed of 5 functional tabs and a Help tab. Functional tabs control or read a specific hardware function. In this section we would like to describe each of these tabs, a brief version of this description is included in the Help tab in the GUI.

The tab change is implemented to switch mode of *IP_802_11p* PL block by writing into *SELECT_AXI_REGS_MODE* (also called: *MODE* or BRAM *registers*[2]).

5.2.1 Settings tab

The Settings tab is implemented for the ADRV9002 control, all features (except buttons *Check AXI* and *802.11p Reset/Disable*) are implemented using the Libiio (specifically *py-adi*). Let us describe these features:

- *(Re-) Connect IIO* button: Connect to the Libiio.
- *Check AXI* button: On Linux, try reading address 0x5000_0000 (BRAM *registers*[0]).
- *Load Stream & Profile* button: Load TES Stream and Profile (from *data/*; see subsection 3.3.4), then set both RX and TX to *calibrated* state, set carrier frequencies to 5.9 GHz, turn ON RX AGC and set TX attenuator to -10 dB.

- *802.11p Reset* and *802.11p Disable*: Both buttons clear the BRAM (including *registers*[0] synchronized with NOT *RESET*), the reset button then enables the *IP_802_11p* PL block (writes 1 to registers[0]) and set default threshold (*registers*[1] or even bits of *DETECTION_THRESHOLD*, see subsection 4.1.3) to 1e5.

- Other control buttons (with their corresponding entries) serve for manual ADRV9002 channel gains, carrier frequencies and state control.

- *Control (IIO) transmitter*: Use the transmitter for transmitting a pre-computed IEEE 802.11p packet in more modulations and code rates. The packets (saved at *data/signals/*) were generated by MATLAB script *./others/matlab 802_11a_p/generate_signal_802_11p.m* (changing *DATARATE* variable) according to the example in [3, p. 55].

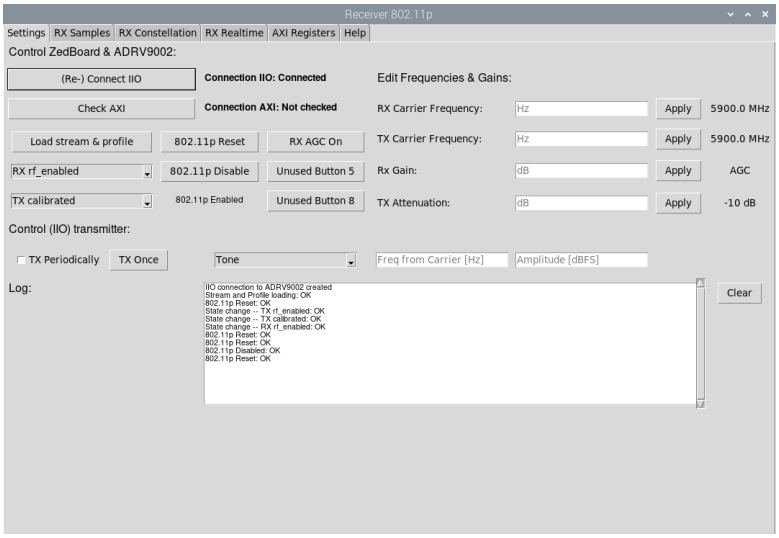


Figure 5.1: GUI Settings tab

5.2.2 RX Samples tab

The RX Samples tab is created for visualizing the received IQ data, specifically I part (blue), Q part (red), amplitude (green) in the upper graph and power spectral density in the lower graph. It supports two types of IQ data.

The first IQ data type sets *IP_802_11p* PL block BRAM *registers*[2] $MODE = 1$ (when entering the tab) and reads the selected register address range. The IQ samples address range can be between 16 and 4094, this is read by *Read AXI* button. The retrieved values are time-synchronized to the moment of the STS Coarse frequency offset computation (that is 1-7 last IQ samples of the LTS prefix, depends on delay from *atan_block.vhd*) (see subsection 4.2.9), thus the data is only valid when a reception happened between the setting of $MODE = 1$ and the AXI reading.

The second IQ data type is implemented by calling the Libiio (via *py-adi*), thus it does not need the *IP_802_11p* block. The data is read asynchronously after pressing *Read AXI* button with maximal buffer length of 2^{22} samples.

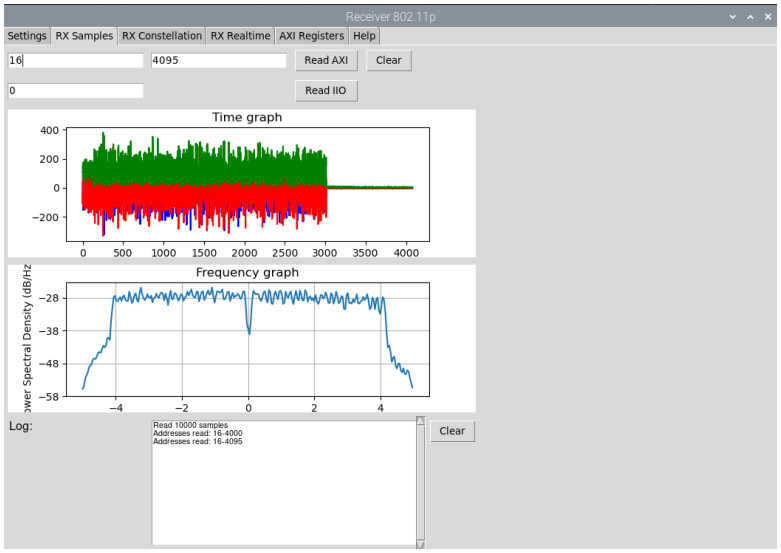


Figure 5.2: RX Samples tab (AXI BRAM)

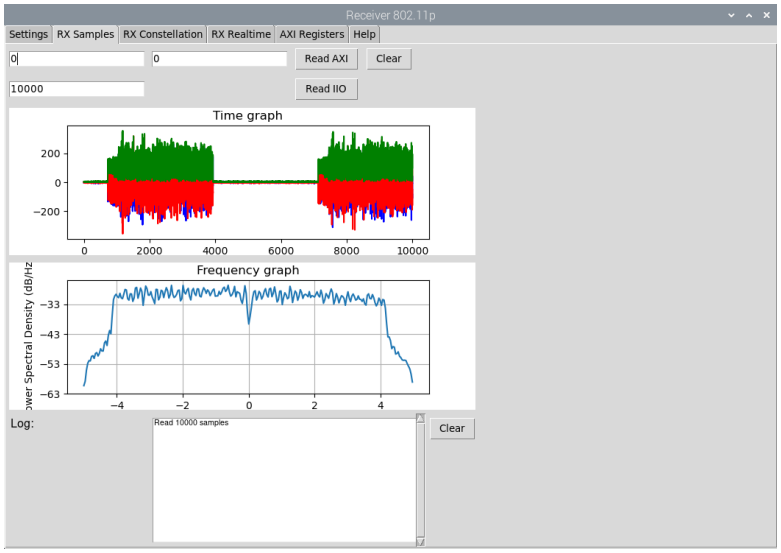


Figure 5.3: GUI Settings tab (Libiio)

5.2.3 RX Constellation tab

The RX Constellation tab is made for visualizing the OFDM demodulated subcarrier data from *IP_802_11p* PL block in constellation space.

There are four modes that can visualize different data, all differ in setting *IP_802_11p* PL block BRAM *registers*[2] *MODE*, therefore, as in the previous tab, the data is only valid when a reception happened between the setting

of *MODE* and the AXI reading. The address range for reading remains between 16 and 4094. There are following modes for visualization (see also subsection 4.2.9):

- *FFT output (16 MSB)* (*MODE* = 2): OFDM demodulated subcarriers (all 64 subcarriers; including both LTS sequences) from *fft_ofdm.vhd* block (16/24 MSB, then 8 LSB zeros added)

- *FFT output (16 LSB)* (*MODE* = 3): OFDM demodulated subcarriers (all 64 subcarriers; including both LTS sequences) from *fft_ofdm.vhd* block (16/24 LSB with sign and out of limit value limitation)

- *Tracked constellation (16 MSB)* (*MODE* = 4): Phase equalized (OFDM demodulated) subcarriers (all used 52 subcarriers including pilots; excluding LTS sequences) from *constellation_tracker.vhd* block (16/24 MSB, then 8 LSB zeros added)

- *Tracked constellation* (*MODE* = 5): Phase equalized (OFDM demodulated) subcarriers (all used 52 subcarriers including pilots; excluding LTS sequences) from *constellation_tracker.vhd* block (16/24 LSB with sign and out of limit value limitation)

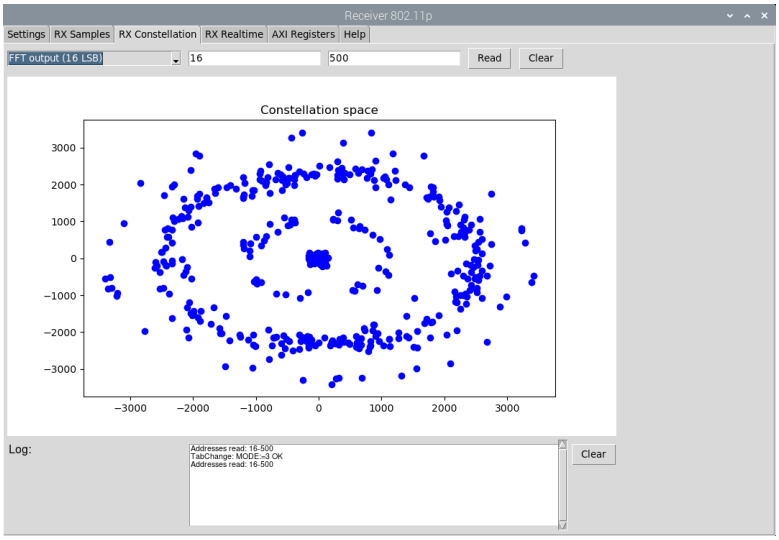


Figure 5.4: GUI Constellation tab (OFDM demodulated subcarriers)

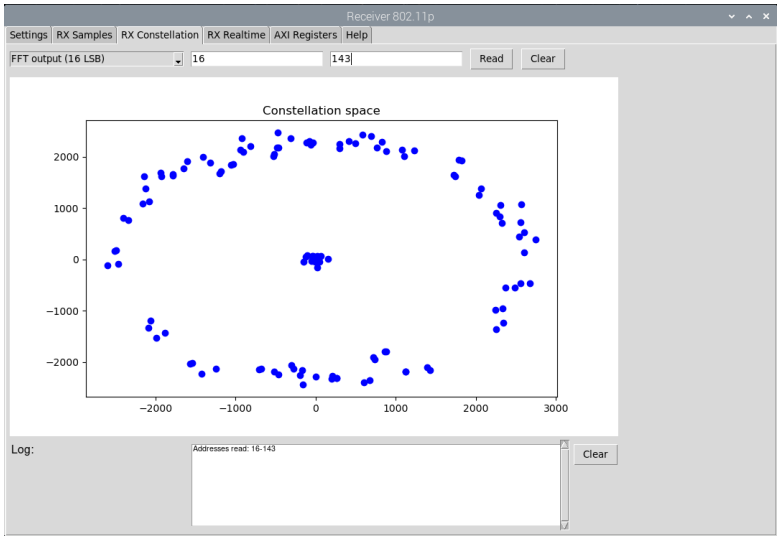


Figure 5.5: GUI Constellation tab (OFDM demodulated subcarriers, LTS only)

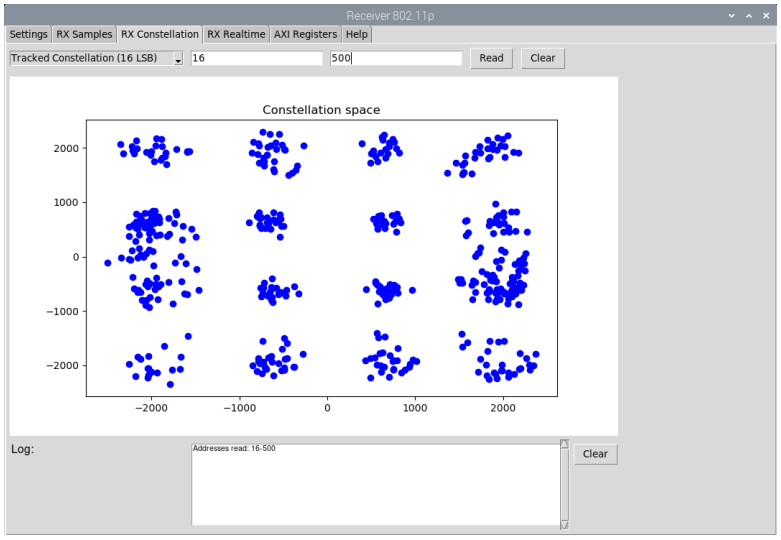


Figure 5.6: GUI Constellation tab (OFDM demodulated and equalized subcarriers)

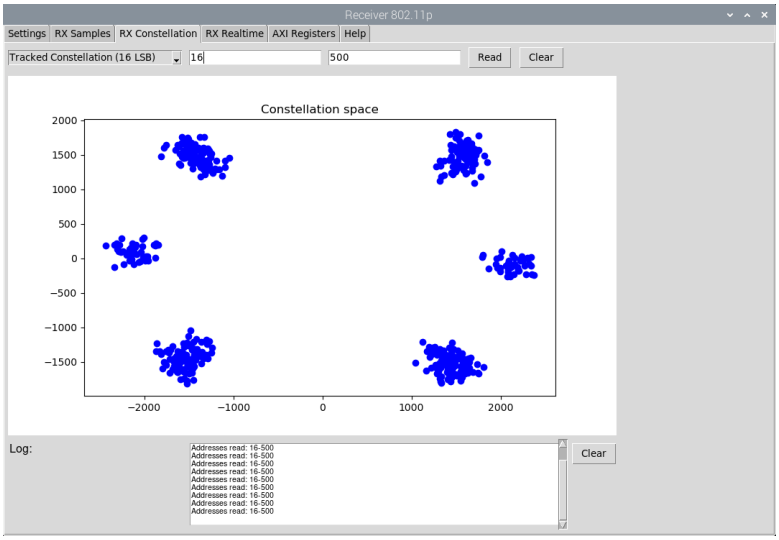


Figure 5.7: GUI Constellation tab (OFDM demodulated and equalized subcarriers)

5.2.4 RX Realtime tab

The RX Realtime tab is made for reception of the final decoded and descrambled data as well as providing parameters of the reception, it uses register $MODE = 9$ (when entering the tab, see subsection 4.2.9). When clicking the *Start* buttton the AXI BRAM is read every 500 ms and a new data is written to the log. It includes:

1. Counters of started receptions and done decodings (see subsection 4.2.9)
2. STS and LTS frequency offset synchronization (the fine LTS offset is the difference to the coarse STS offset)
3. *SIGNAL* field in transmission order (LSB first)
4. From the *SIGNAL* field: Modulation, Code rate and Length (in bytes)
5. *SERVICE* field in transmission order (LSB first)
6. DATA field in hex format (space divided)
7. DATA field in ASCII format (only displayable characters (32-126), CR and LF, others displayed as '?')

When clicking the *Read Once...* button the reading is performed only once but the data (PSDU field in binary format) is also written to the file *received_data/received_data.txt*.

An example of a RX Realtime tab output can be seen in Figure 5.8.

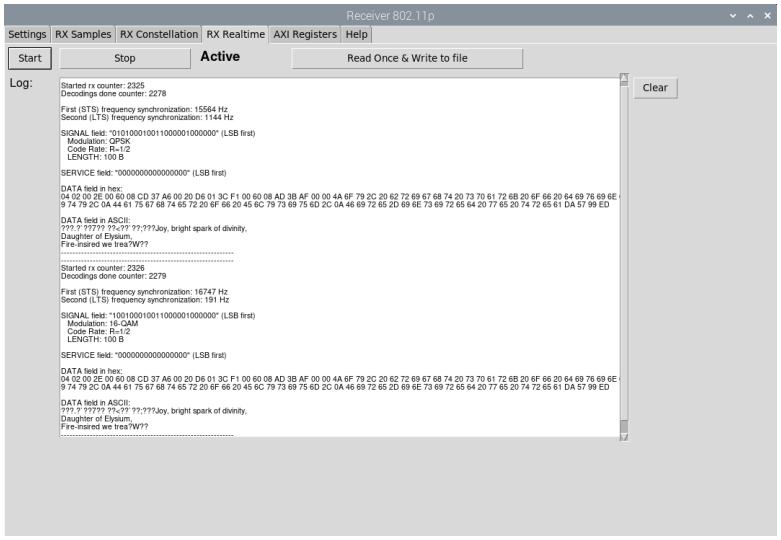


Figure 5.8: GUI Realtime tab

The Viterbi depuncturing does not work, this can be seen in Figure 5.9. See section 4.4 for more details

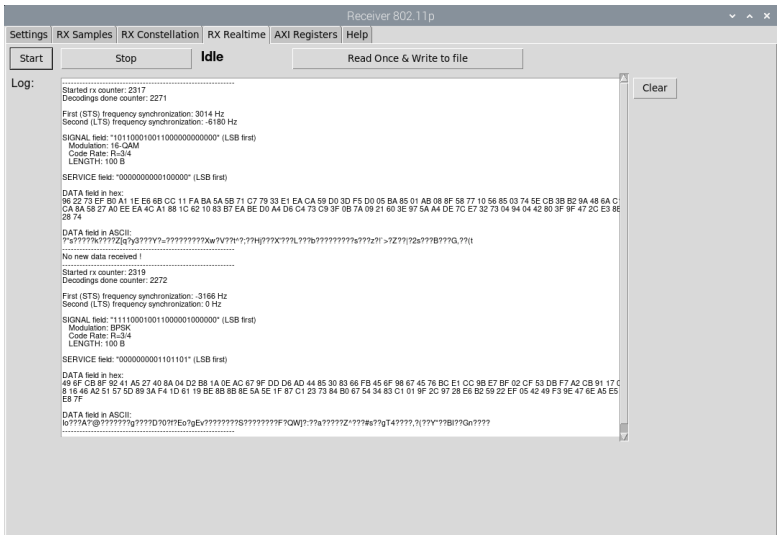


Figure 5.9: GUI Realtime tab (Viterbi error)

5.2.5 AXI Registers tab

The AXI Registers tab is used for manual writing and reading to/from the BRAM memory. This includes manual selection of the detection threshold (*registers*[1] or even bits of *DETECTION_THRESHOLD*, see subsection 4.1.3) or *MODE* (*registers*[2]). Also all 4096 BRAM addresses can be read and displayed in more predefined formats, that is: one hex format, one uint32, two int16 and 4 ASCII characters with reversed bit order (see subsection 2.4.2). The checkbutton *TabChange* can be used to disable *MODE* (*registers*[2])

writing with tab changes. For the BRAM (register) usage see subsection 4.1.3 and subsection 4.2.9.

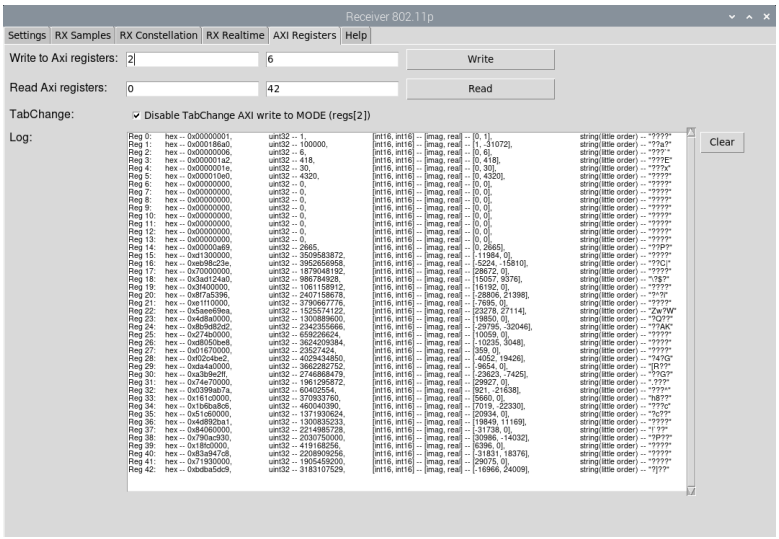


Figure 5.10: GUI Registers tab (MODE = 6, demapped BPSK bits)

Chapter 6

Conclusion

In Chapter 2 we got acquainted with the IEEE 802.11p standard. This included data scrambling, coding, interleaving, modulating by mapping, modulating by OFDM and creating synchronization sequences.

In Chapter 3 we used the available hardware and software to prepare a custom workflow using mainly AMD Vivado and Vitis.

In Chapter 4 we used the gained information about the standard and our workflow to make a custom FPGA receiver as a Vivado IP block (not fully dependent on the current FPGA). Chapter 5 described a custom GUI running on a PS part (CPU) of the given board to access the block in the FPGA.

In this project many things could be improved and developed further, this relates mainly to the demapper and Viterbi algorithm block which were made using a few heuristics, the frequency offset synchronization should be reviewed, as well as used BRAM interfacing.

On the other hand, the resulting project is working and is able to receive given test packet. The project is also quite easy to edit. To conclude the goal of basic reception in the test bed was accomplished

Bibliography

[1] Kaja H, Stoehr JM, Beard C. *V2X-assisted emergency vehicle transit in VANETs. SIMULATION.* 2024;100(3):229-244. doi:10.1177/00375497231209774. https://doi.org/10.1177/00375497231209774

[2] *Vehicle-to-Everything (V2X) Communication: Enhancing Road Safety and Traffic Management*, AUTOMOTIVE Technology, accessed May 2024. https://www.automotive-technology.com/articles/vehicle-to-everything-v2x-communication-enhancing-road-safety-and-traffic-management

[3] S. Gräfling, P. Mähönen and J. Riihijärvi, "Performance evaluation of IEEE 1609 WAVE and IEEE 802.11p for vehicular communications," 2010 Second International Conference on Ubiquitous and Future Networks (ICUFN), Jeju, Korea (South), 2010, pp. 344-348, doi: 10.1109/ICUFN.2010.5547184. https://doi.org/10.1109%2FICUFN.2010.5547184

[4] *Admission control based on rate-variance envelop for VBR traffic over IEEE 802.11e HCCA WLANs - Scientific Figure on ResearchGate*. accessed May 2024. https://www.researchgate.net/figure/The-data-frame-format-of-80211a_fig4_3156831,

[5] Bilstrup, Katrin & Uhlemann, Elisabeth & Ström, Erik & Bilstrup, Urban. (2009). On the Ability of the 802.11p MAC Method and STDMA to Support Real-Time Vehicle-to-Vehicle Communication. EURASIP J. Wireless Comm. and Networking. 2009. 10.1155/2009/902414. https://www.researchgate.net/publication/220537383_On_the_Ability_of_the_80211p_MAC_Method_and_STDMA_to_Support_Real-Time_Vehicle-to-Vehicle_Communication

[6] Bilgin, Bilal & Gungor, V.C.. (2013). Performance Comparison of IEEE 802.11p and IEEE 802.11b for Vehicle-to-Vehicle Communications in Highway, Rural, and Urban Areas. International Journal of Vehicular Technology. 2013. 10.1155/2013/971684. https://www.researchgate.net/publication/289662891_Performance_

Comparison_of_IEEE_80211p_and_IEEE_80211b_for_Vehicle-to-Vehicle_Communications_in_Highway_Rural_and_Urban_Areas

[7] *Interframe spaces (RIFS, SIFS, PIFS, DIFS, AIFS, EIFS) - Figure 8.2 SIFS and DIFS*, Dot11AP @AmitPindoria. accessed May 2024. https://dot11ap.wordpress.com/interframe-spaces-rifs-sifs-pifs-difs-aifs-eifs/

[8] L. Ward, Dr. M. Simon, Rohde & Schwarz GmbH & Co KG. *Intelligent Transportation Systems Using IEEE 802.11p*. 2019. https://scdn.rohde-schwarz.com/ur/pws/dl_downloads/dl_application/application_notes/1ma152/1MA152_5e_ITS_using_802_11p.pdf

[9] "IEEE Standard for Telecommunications and Information Exchange Between Systems - LAN/MAN Specific Requirements - Part 11: Wireless Medium Access Control (MAC) and physical layer (PHY) specifications: High Speed Physical Layer in the 5 GHz band," in IEEE Std 802.11a-1999 , vol., no., pp.1-102, 30 Dec. 1999, doi: 10.1109/IEEESTD.1999.90606. https://pdos.csail.mit.edu/archive/decouto/papers/802.11a.pdf

[10] J. B. Kenney, "Dedicated Short-Range Communications (DSRC) Standards in the United States," in Proceedings of the IEEE, vol. 99, no. 7, pp. 1162-1182, July 2011, doi: 10.1109/JPROC.2011.2132790. https://ieeexplore.ieee.org/document/5888501

[11] Jiang, Daniel & Delgrossi, Luca. (2008). IEEE 802.11p: Towards an International Standard for Wireless Access in Vehicular Environments. IEEE Vehicular Technology Conference. 2036 - 2040. 10.1109/VETECS.2008.458. https://standards.ieee.org/ieee/802.11p/3953/

[12] Sourour, Essam & El-Ghoroury, H. & McNeill, D.. (2004). Frequency offset estimation and correction in the IEEE 802.11a WLAN. 4923 - 4927 Vol. 7. 10.1109/VETECF.2004.1405033. https://www.researchgate.net/publication/4128357_Frequency_offset_estimation_and_correction_in_the_IEEE_80211a_WLAN

[13] *ZedBoard product details*, Avnet, Inc., accessed May 2024.https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/zedboard/

[14] *Zynq 7000 SoC product details*, Advanced Micro Devices, Inc., accessed May 2024.https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html#documentation

[15] *ADRV9002 product details*, Analog Devices, Inc., accessed May 2024. https://www.analog.com/en/products/adrv9002.html#product-overview

[16] *EVAL-ADRV9002 Overview*, Analog Devices, Inc., accessed May 2024. https://www.analog.com/en/resources/evaluation-hardware-and-software/evaluation-boards-kits/eval-adrv9002.html#eb-overview

[17] *ADRV9001 User Guide*, Analog Devices, Inc., accessed May 2024. https://www.analog.com/media/en/technical-documentation/user-guides/adrv9001-system-development-user-guide-ug-1828.pdf

[18] *ADRV9001 datasheet*, Analog Devices, Inc., accessed May 2024. https://www.analog.com/media/en/technical-documentation/data-sheets/adrv9002.pdf

[19] *HDL GitHub*, Analog Devices, Inc., accessed May 2024.https://github.com/analogdevicesinc/hdl

[20] *build ADI HDL*, Analog Devices, Inc., accessed May 2024. https://wiki.analog.com/resources/fpga/docs/build

[21] *Kuiper Linux*, Analog Devices, Inc., accessed May 2024. https://wiki.analog.com/resources/tools-software/linux-software/kuiper-linux

[22] *About libiio*, Analog Devices, Inc., accessed May 2024. https://wiki.analog.com/resources/tools-software/linux-software/libiio_internals

[23] *Libiio GitHub*, Analog Devices, Inc., accessed May 2024. https://github.com/analogdevicesinc/libiio/tree/2021_R2

[24] *IIO Oscilloscope*, Analog Devices, Inc., accessed May 2024. https://wiki.analog.com/resources/tools-software/linux-software/iio_oscilloscope

[25] *Transceiver Evaluation Software (TES)*, Analog Devices, Inc., accessed May 2024. https://www.analog.com/en/license/licensing-agreement/transceiver-evaluation-software.html

[26] *ADRV9001 SOFTWARE AND HARDWARE SELECTION GUIDE*, Analog Devices, Inc., accessed May 2024. https://www.analog.com/media/en/evaluation-boards-kits/evaluation-software/adrv9001_software_and_hardware_selection_guide.pdf

Appendix A

Attached files

Full repository can be accessed by my school GitLab or my personal GitHub:
`https://gitlab.fel.cvut.cz/kimmemic/zedboard_adrv9002_project.git`
`https://github.com/michaelkimmer/zedboard_adrv9002_project.git`

Necessary files are attached.