The CBSR receiver is one of the main components of the ground segment of the communication system (i.e. ground station). It is implemented using Software Defined Radio technique, however, both “hardware oriented” and “software oriented” versions will be developed for system flexibility. In this document the electrical and mechanical design of all versions are discussed along with implementation aspects.

Contents

List of acronyms

BB	Base-band
CBSR	C-band Satellite Radio
DAC	Digital to Analog Converter
DMA	Direct Memory Access
FSM	Finite State Machine
NCBR	Narodowe Centrum Badań I Rozwoju
PA	Power Amplifier
PN	Pseudorandom Noise
PUT	Poznań University of Technology
SDR	Software Defined Radio
SR	SatRevolution S.A.
SoM	System-on-Module
SoC	System-on-Chip
t.b.d	to be determined

1. Module Characteristics

1.1. General Parameters

The CBSR receiver has been designed as the main component of the Cubesat ground station which was developed and implemented at PUT.

Based on the system specification presented in [1], the electrical parameters of the receiver are the following:

- carrier frequency – user selectable between 5500 MHz and 6000 MHz, default 5840 MHz
- channel bandwidth – user selectable: 1 MHz, 1.25 MHz, 5 MHz, 10 MHz, 20 MHz
- modulation type - digital (quadrature) - OQPSK
- channel coding – Turbo, user selectable code rate: 0.19 – 0.91
- data/control interface – RS-232, Ethernet

1.2. Implementation

The receiver has been implemented using SDR technique, however, both “hardware oriented” and “software oriented” versions will be developed for system flexibility.

1.2.1. “Hardware oriented” version

In “hardware oriented” version most of the receiver functionality is implemented using Xilinx Zynq7000 SoC based hardware platforms. Using the receiver Simulink model, developed in the project, the proprietary IP-cores are generated for Zynq7000 Programmable Logic (PL). Zynq7000 Processing System (PS) runs Linux applications which exchange data with the PL and perform non-time-critical operations.

The receiver implementation has been developed for two hardware platforms:

- ADRV9361 (a.k.a PicoZed) SoM featuring AD9361 transceiver IC and Zynq7035 device
- Mini-ITX board featuring Zynq7100 device with AD-FMCOMMS3 AD9361 evaluation module attached via FMC-LPC connector

The “hardware oriented” version performs real-time processing of the received signal, delivering the transmitted data “on-line” for the user in the ground station.

1.2.2. “Software oriented” version

In “software oriented” version most of the receiver functionality is implemented using a high-performance PC running the software receiver based on GnuRadio platform. The RF part uses the off-the-shelf USRP B210 SDR hardware platform for signal reception and analog-to-digital conversion, attached to the PC via USB interface.

The “software oriented” version performs non-real-time processing of the received signal, delivering the transmitted data “off-line” for the user in the ground station.

2. Module Electrical Design - “hardware oriented” version

An efficient SDR implementation of the receiver requires application of an FPGA device. For this purpose a Xilinx ZYNQ System-on-Chip device, which combines dual-core ARM Cortex-A9 processor with the FPGA and a choice of interfaces was selected [2]. Most of the BB processing blocks are implemented in the FPGA, while the ARM processor is used for system control and management functions.

2.1. ADRV9361 based receiver

ADRV9361 is a Software Defined Radio (SDR) that combines the Analog Devices AD9361 integrated RF Agile Transceiver™ with the Xilinx Z7035 Zynq®-7000 All Programmable SoC in a small system-on-module (SoM) footprint suitable for end-product integration.

The key features of the module [3] are listed below:

- **Low-power -** Designed with a -2LI version of the Zynq SoC (low power, mid speed, industrial temp), DDR3L, and high-efficiency voltage regulators with margining capability to scale power with performance. Built-in sequencing and monitoring make it easy to power to the module.
- **High bandwidth data connectivity -** Move data quickly with dual Gigabit Ethernet, USB2.0, four 6.6 Gb/s serial links (PCIe x4, SFP+, others), and high-speed LVDS I/O for custom interfaces.
- **Wideband, frequency agile RF -** Uses the AD9361 to provide a highly integrated radio that enables wideband 2x2 MIMO receive and transmit paths from 70 MHz to 6.0 GHz with tunable channel bandwidth <200kHz to 56MHz.
- **Programmable SoC -** Embedded processing with the Zynq Z-7035 SoC provides a Dual ARM® Cortex™-A9 MPCore™ running at 800MHz, with built in peripherals like USB, Gigabit Ethernet, and memory interfaces.
- **Small form factor -** 100mm x 62mm footprint.
- **Production-ready module -** System-on-Module designed for immediate prototype and quick integration in your end application. Industrial temperature rated and tested against MIL-STD 202G methods for Thermal, Vibration, and Shock.
- **Operating systems -** Comes with Analog Devices Linux reference design for Zynq, bootable from an SD card. Also supports Linux, Android, FreeRTOS, eCos, VxWorks, and others not listed here.
- **Development tools -** A broad range of SDR prototype and development environments are supported, including Analog Devices Linux Applications, and MATLAB® and Simulink® for data streaming and Zynq targeting.
- **Open-source code -** Analog Devices provides precompiled reference designs on their PicoZed SDR wiki page and a source code support package hosted on Github, including the HDL and software code (except non-ADI).

ADRV9361 module block diagram is depicted in Fig. 2.1 and Fig. 2.2 shows its layout.

ADRV9361 can not work as a stand-alone module, it requires a corresponding carrier card. For this purpose the ADRV1CRR-FMC (a.k.a AES-PZSDRCC-FMC-G) carrier is used. The card gives designers access to a wide variety of peripherals and user I/O required to evaluate and develop with ADRV9361 SoM. The carrier card provides all necessary SoM power, reset control, and Zynq SoC I/O pin accessibility through the board-to-board (B2B) micro headers (see Fig. 2.3, 2.4).

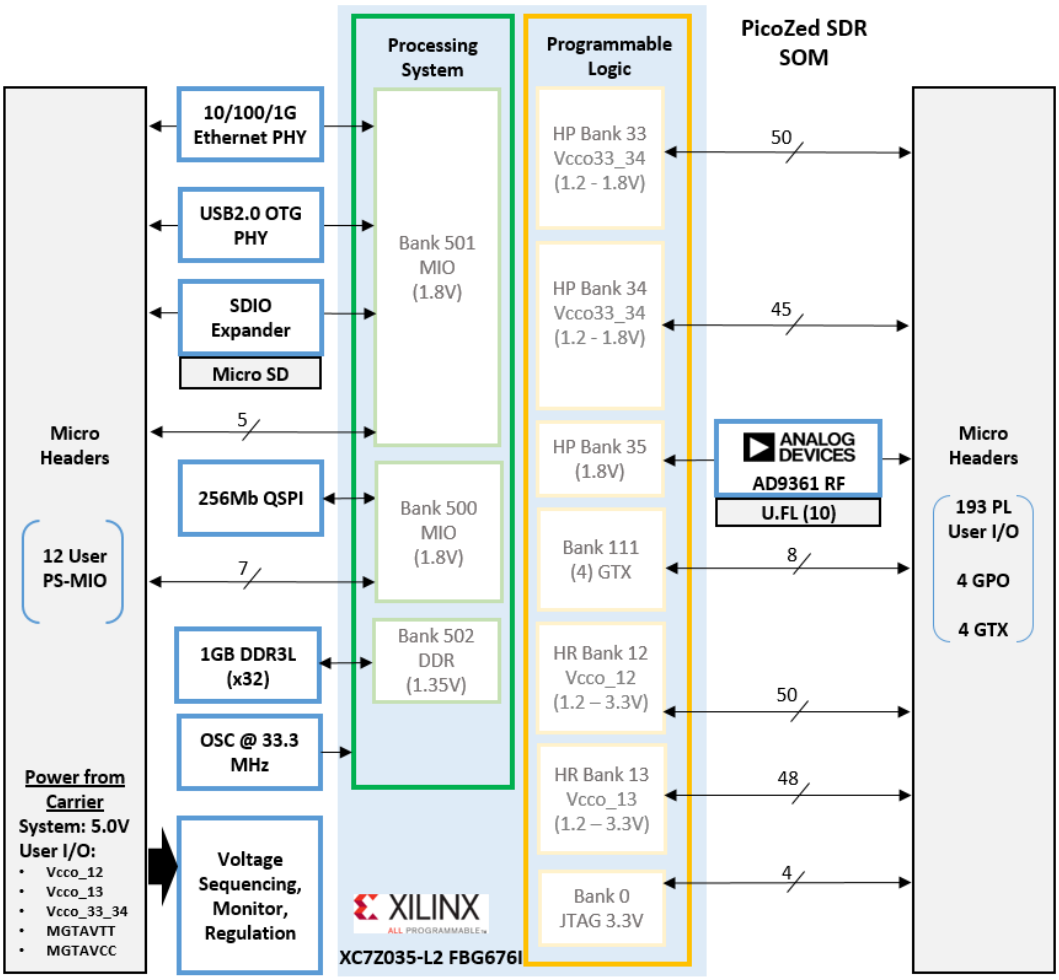

Fig. 2.1 ADRV9361 SoM block diagram

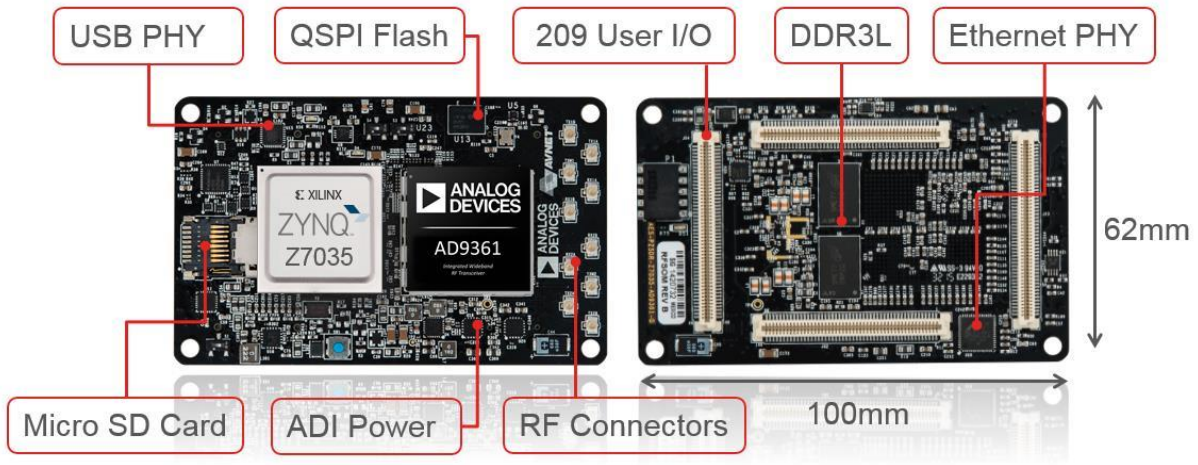

Fig. 2.2 ADRV9361 SoM device layout

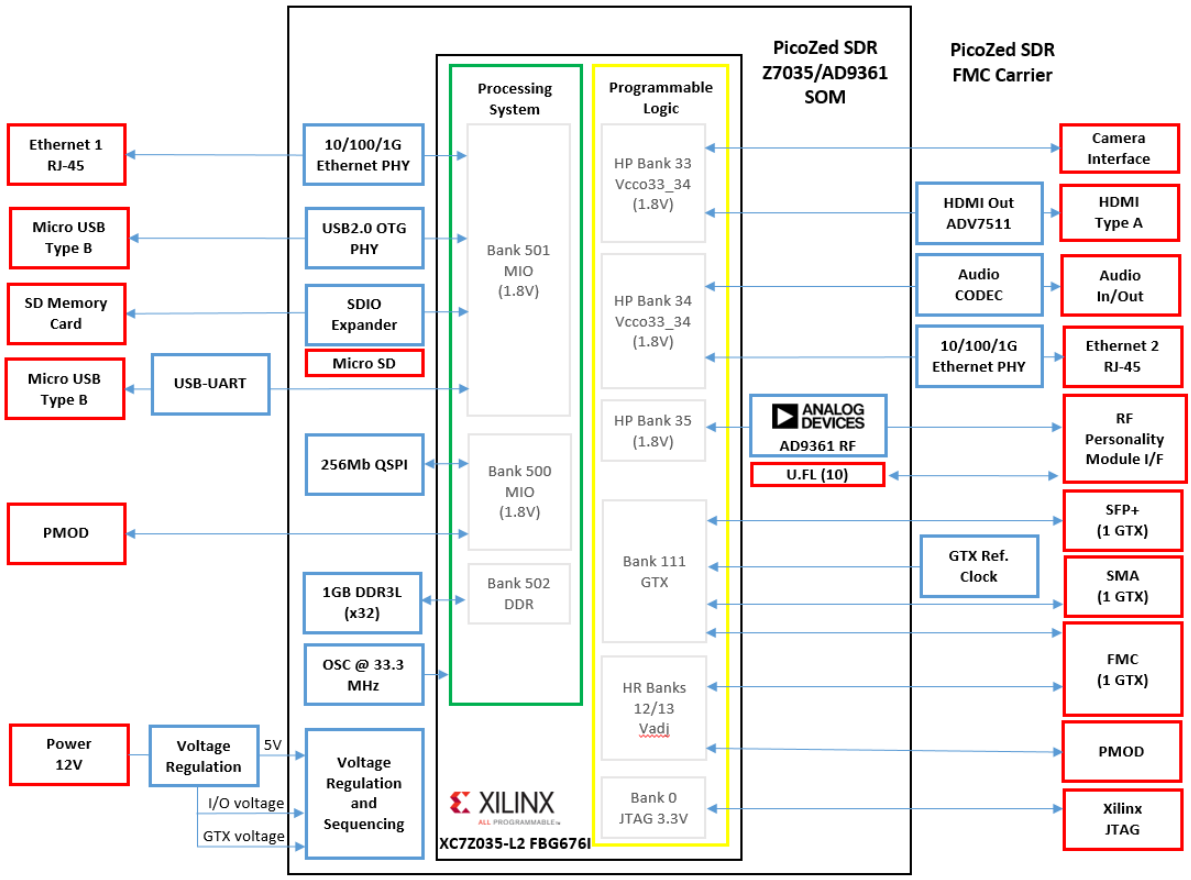

Fig. 2.3 ADRV1CRR-FMC carrier card block diagram

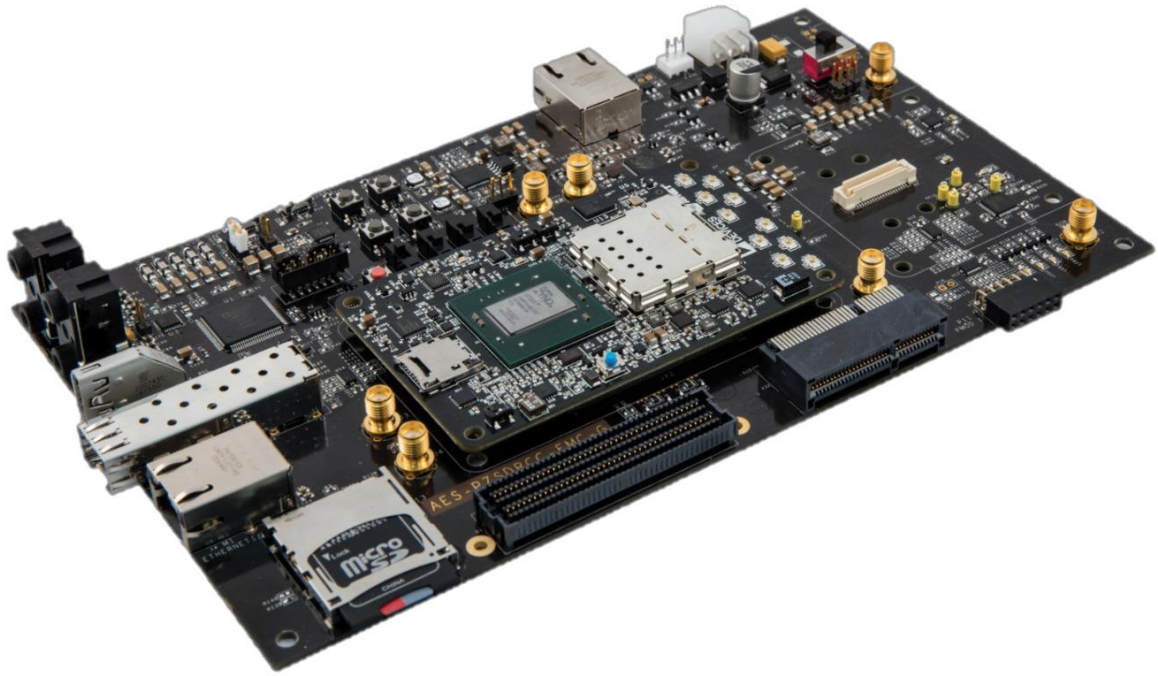

Fig. 2.4 ADRV1CRR-FMC carrier card layout

2.2. Xilinx Zynq Mini-ITX based receiver

The Xilinx Zynq Mini-ITX Development Kit provides a complete development platform for designing and verifying applications based on the Xilinx Zynq-7000 All Programmable SoC family. Available with the Zynq XC7Z045-2FFG900 or the XC7Z100-2FFG900 device in a small Mini-ITX form factor, the kit enables designers to prototype high-performance designs with ease, while providing expandability and customization through the FMC HPC expansion slot. The Zynq Mini-ITX development board features consist of:

- Xilinx Zynq Z7100 SoC
- 1GB PS DDR3 SDRAM
- 1GB PL DDR3 SDRAM
- 32MB of QSPI Flash
- 8KB of I2C EEPROM
- Real-Time Clock
- 10/100/1000 Ethernet Interface
- USB-UART Interface
- microSD Card Interface
- USB 2.0 4-Port Hub
- PCIe x4 Root-Port (x16 physical Slot)
- SATA-III Interface
- FMC HPC Slot (VADJ of 1.8V, 2.5V, or 3.3V)
- SFP Socket
- LVDS Touch Panel Interface
- HDMI Interface
- Audio Codec
- User LEDs and Switches
- Programmable LVDS Clock Source (GTX reference clock)
- 200 MHz LVDS Oscillator (system clock)
- JTAG Header

The Mini-ITX Development Kit block diagram is depicted in Fig. 2.5.

The on-board FMC slot is used to connect to the AD-FMComms3-EBZ transceiver board. It provides AD9361 based RF platform, a highly integrated radio that enables wideband 2x2 MIMO receive and transmit paths from 70 MHz to 6.0 GHz with tunable channel bandwidth <200kHz to 56MHz.

The Mini-ITX platform, including the RF part, is shown in Fig. 2.6. The boards are mounted in a Mini-ITX format enclosure, together with a power supply.

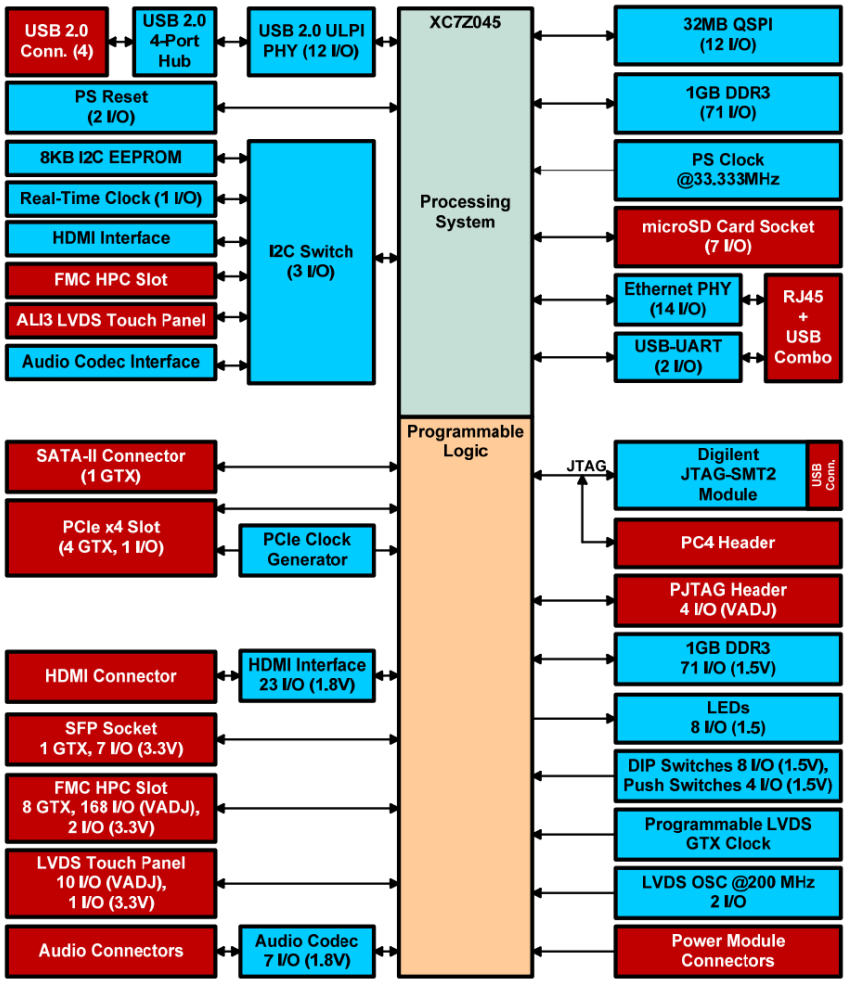

Fig. 2.5 Mini-ITX Development Kit block diagram

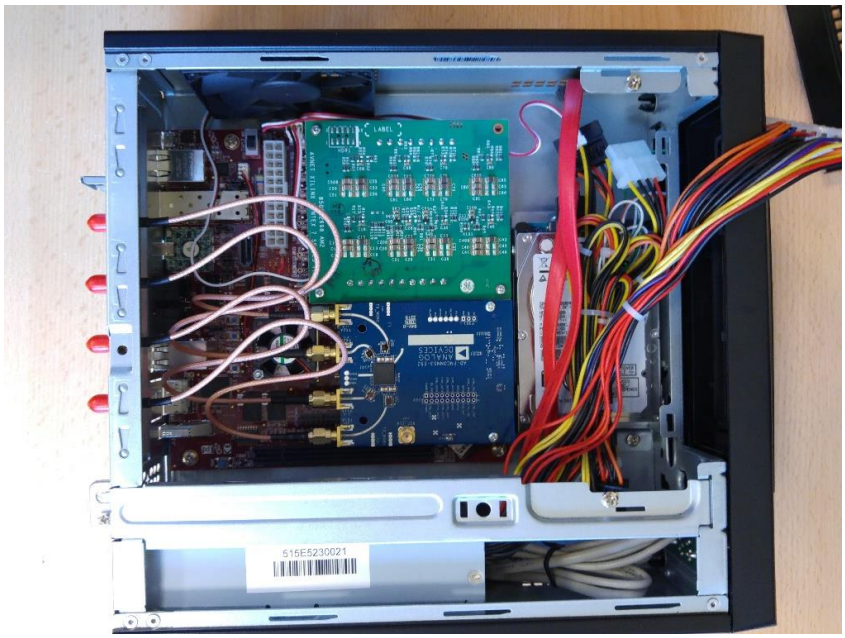

Fig. 2.6 Mini-ITX SDR platform

3. Module Electrical Design - “software oriented” version

The “software oriented” receiver is designed for low-cost applications, e.g. at supporting ground stations which do not require on-line decoding of the transmitted data.

3.1. USRP B210 hardware platform []

The USRP B210 provides a fully integrated, single-board, Universal Software Radio Peripheral (USRP™) platform with continuous frequency coverage from 70 MHz – 6 GHz. Designed for low-cost experimentation, it combines the AD9361 RFIC direct-conversion transceiver providing up to 56MHz of real-time bandwidth, an open and reprogrammable Spartan6 FPGA, and fast SuperSpeed USB 3.0 connectivity with convenient bus-power. Full support for the USRP Hardware Driver™ (UHD) software allows you to immediately begin developing with SDR software environment, e.g. GNU Radio.

The integrated RF frontend on the USRP B210 is designed with the new Analog Devices AD9361, a single-chip direct-conversion transceiver, capable of streaming up to 56 MHz of real-time RF bandwidth. The B210 uses both signal chains of the AD9361, providing coherent MIMO capability. Onboard signal processing and control of the AD9361 is performed by a Spartan6 XC6SLX150 FPGA connected to a host PC using SuperSpeed USB 3.0. The USRP B210 real time throughput is benchmarked at 61.44MS/s quadrature, providing the full 56 MHz of instantaneous RF bandwidth to the host PC for additional processing.

Since the receiver uses a single receive channel, a simpler (and cheaper) USRP B200 platform can be used without any degradation.

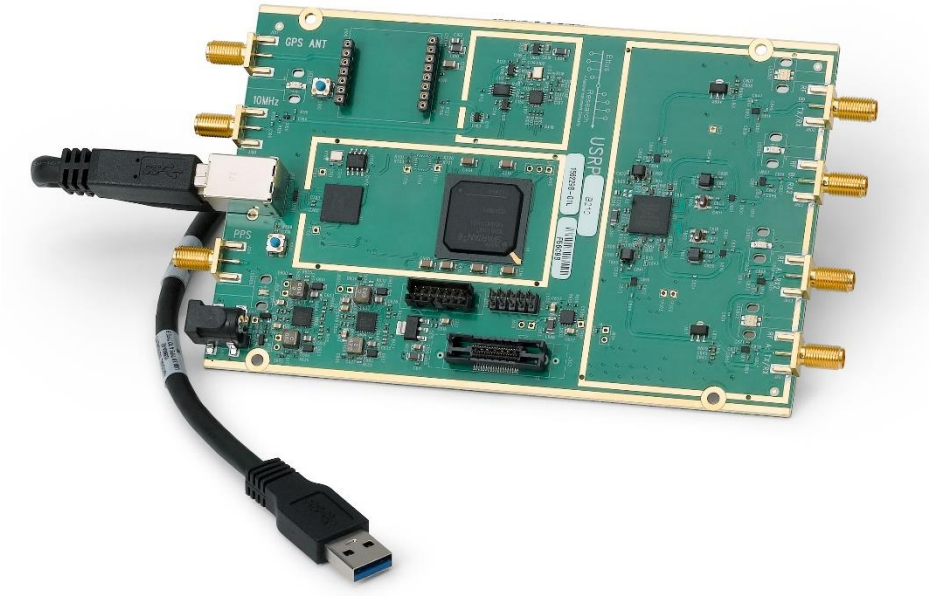

Fig. 2.7 USRP B210 hardware platform

4. Receiver implementation – PL part (FPGA) for "hardware oriented" version

4.1. Simulink model

The receiver implementation in FPGA is prepared similarly to the transmitter with the help of Matlab/Simulink and its HDL Coder toolbox. In Fig. 4.1 the receiver block inputs and outputs are presented, the relevant ones are described in Table 4.1.

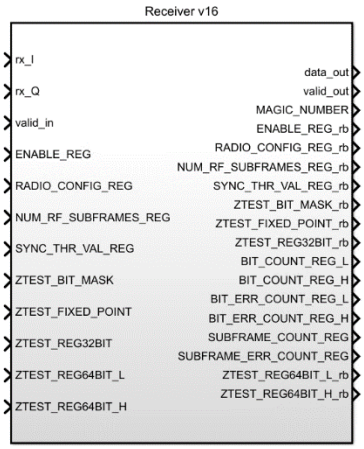

Fig. 4.1 Receiver block

Table 4.1 Inputs and outputs of receiver block

Name	Type	Description
rx_I/rx_Q	Input	Receivers' in-phase and quadrature component samples
valid_in	Input	Valid signal for ADC samples
ENABLE_REG	Input/ Output	Register that stores i.a. a value that enables/disables the receiver. When its state is high, the signal can be received, when it's low the receiver will stop collecting samples on its input
RADIO_CONFIG_REG	Input/ Output	A register used to set: • the transmission mode (described in [3]) • the roll-off factor of the shaping RRC filter. There are 2 settings available i.e. 0 and 1. • the length of the frequency offset estimation part of the preamble (described in [3])
NUM_RF_SUBFRAMES_REG	Input/ Output	Sets the number of subframes in each radio frame.
data_out	Output	32-bit data words
valid_out	Output	Indication whether output data is valid
SUBFRAME_COUNT_REG	Output	The number of received subframes
SUBFRAME_ERR_COUNT_REG	Output	The number of erroneous subframes
BIT_COUNT_REG	Output	The number of received bits in transmission modes 1 and 2

BIT_ERR_COUNT_REG	Output	The number of erroneous bits in transmission modes 1 and 2

In Fig. 4.2 and Fig. 4.3. the general structure of the receiver is presented. The main components of the receiver are shown and described in detail in the subsequent sections.

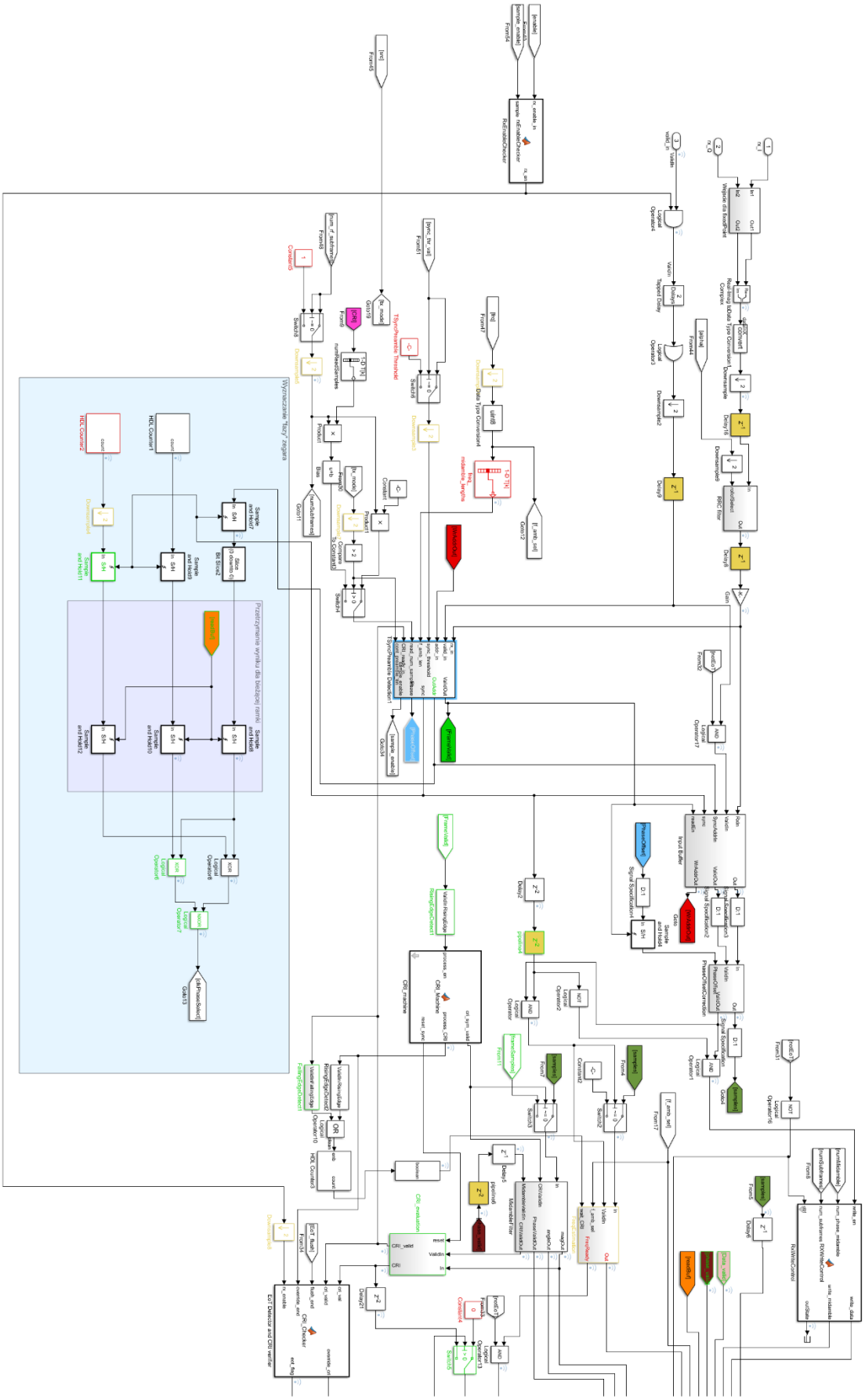

Fig. 4.2 Overview of the receiver Simulink schematics - part 1

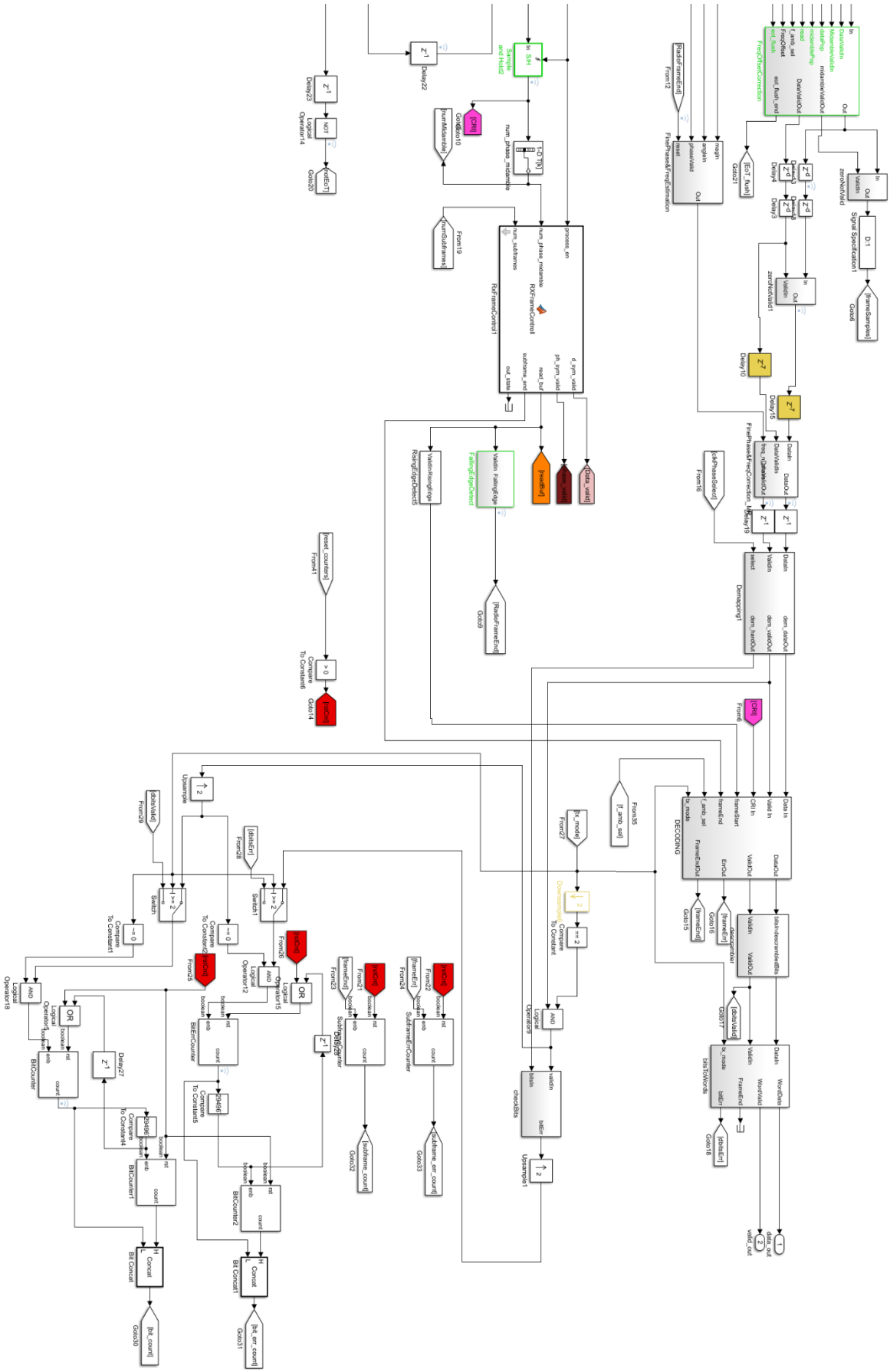

Fig. 4.3 Overview of the receiver Simulink schematics - part 2

4.1.1. Matched Filtering

The operations performed in the receiver are the inversion of those performed in the transmitter. The first part of the receiver (depicted in Fig. 4.4.) is responsible for filtering received signal samples with a matched RRC filter. The output samples from the filter are stored in an input buffer (*Input Buffer* block in Fig. 4.2) but also fed into a signal detection and time synchronization block named *TSyncPreambleDetection*

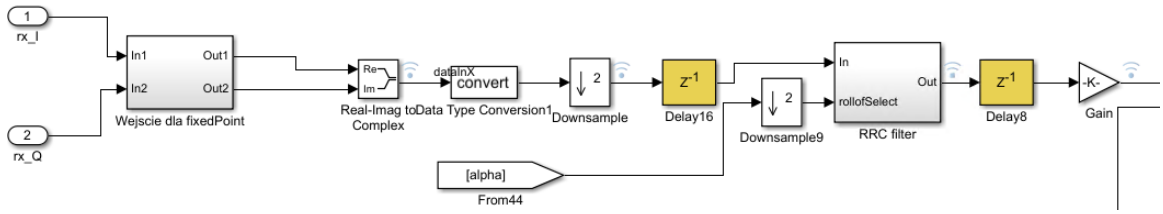

Fig. 4.4 Matched filtering

4.1.2. Time synchronization

The internal structure of the detection and time synchronization block is shown in Fig. 4.5. The input samples are first filtered with a FIR filter with coefficients matching the Zadoff-Chu sequence used in the *T_AMB* part of the preamble. The output of the filter is fed into a magnitude and phase calculation block. The magnitude is used to construct a correlation metric used for signal detection and time synchronization, whereas the phase is used for initial coarse phase offset estimation. The main part of the synchronization subsystem is the *Sync Machine* FSM (Fig. 4.6). The inputs and outputs of this block are shown and described in Table 4.2.

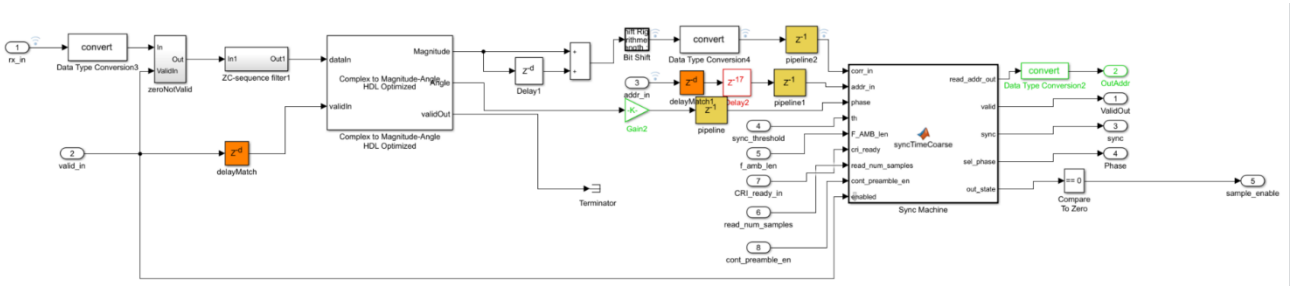

Fig. 4.5 Signal detection and time synchronization

The main task of the *Sync Machine* block is to find the address in the input buffer for which the correlation metric is the highest (assuming that it is higher than the threshold). When the peak in correlation value has been found the *Sync Machine* starts the process of reading samples from the input buffer. The output from the buffer is fed into several subsystems that perform operations such as coarse frequency offset estimation and coding rate estimation, simultaneously/

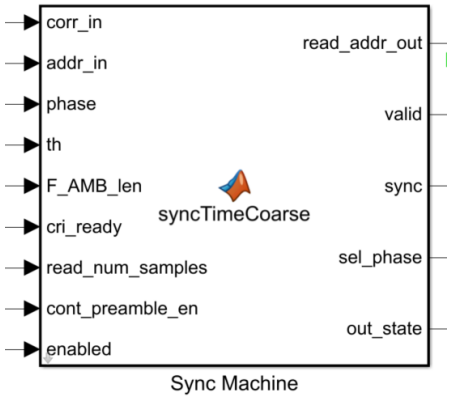

Fig. 4.6 Sync Machine FSM

Table 4.2 Selected Inputs and outputs of *Sync Machine*

Name	Type	Description
corr_in	Input	Received signal correlation metric used for signal detection and time synchronization
addr_in	Input	Current write address in the input buffer
phase	Input	Received signal phase metric used for initial phase offset estimation
th	Input	A threshold value for signal detection
F_AMB_LEN	Input	The length of the preamble used for frequency offset estimation
cri_ready	Input	Indication whether the coding rate has been evaluated
read_num_sampels	Input	Number of samples to read from the input buffer before the next signal detection window
cont_preamble_en	Input	Used to enable continuous preamble mode in the synchronization block
enabled	Input	A line used to enable/disable synchronization is related to the enable signal of the receiver
read_addr_out	Output	Current read address from the input buffer
valid	Output	A signal indicating that the output samples of the input buffer are valid
sync	Output	A signal indicating that radio frame has been detected. The samples output by the input buffer when this signal is high are fed into a coarse frequency offset estimation block. These samples belong to the F_AMB part of the preamble.
sel_phase	Output	Initial coarse phase offset estimate

4.1.3. Coarse Frequency offset estimation

The first *F_AMB_len* samples from the input buffer are used for coarse frequency offset estimation. The goal of this subsystem is to mitigate the effect of a frequency shift. The main source of this shift is the Doppler effect, which impact is also mitigated outside of the receiver implementation, and this subsystem was designed to reduce the impact of any leftover shift that wasn't already compensated.

The internal structure of the coarse frequency offset estimation is shown in Fig. 4.7. Its operation is based on performing FFT on *F_AMB* preamble samples. The index of the maximum magnitude value of the FFT outcome and the *F_AMB* preamble samples are used for calculating metrics used for frequency offset estimation. The detailed algorithm was described in the report [4]**.**

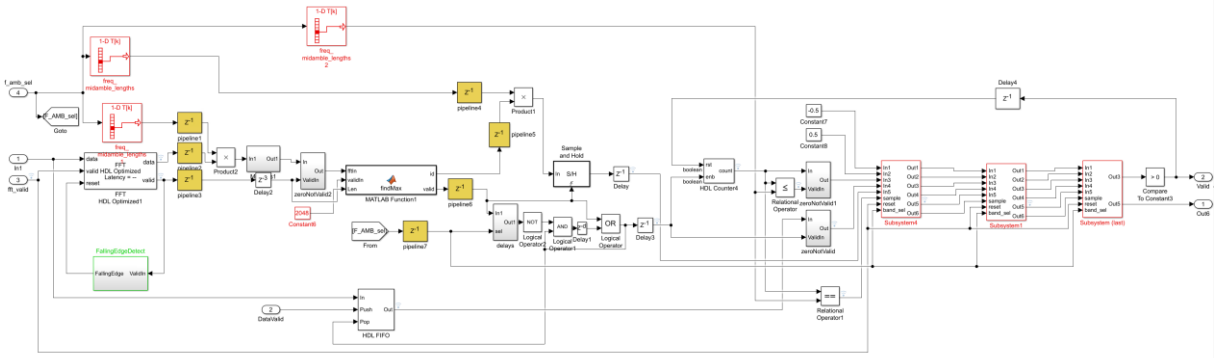

Fig. 4.7 Frequency offset estimation

4.1.4. Coding rate evaluation

The information about the coding rate used in the transmitter is carried by a phase midamble. Each coding rate has its own form of midamble created by performing a cyclical shift on a base Zadoff-Chu sequence. In order to correctly disassemble the received radio frame the correct value of the coding rate used in the transmitter is essential. Hence, the coding rate has to be known before we can process the radio frame. The block responsible for evaluating the coding rate is shown in Fig. .4.8. It operates on the samples coming from a midamble filter subsystem shown in Fig. 4.9. The filter subsystem performs filtering with base Zadoff-Chu coefficients and the magnitude of the filtered samples is used in the *CRI evaluation* subsystem. In the evaluation, we search for the sample index for which the magnitude is maximal. This index is fed into a post-processing system responsible for converting it to the coding rate indicator in the range from 0 to 7 (where values 0-6 denotes valid coding rates and value 7 denotes the EoT frame).

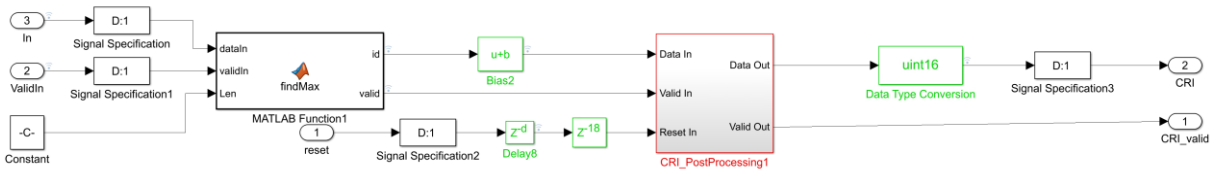

Fig. 4.8 Coding rate evaluation

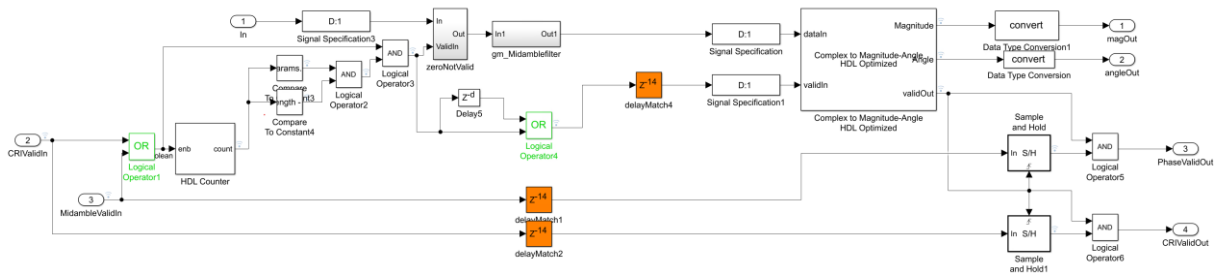

Fig. 4.9 Midamble filter

4.1.5. Radio frame disassembly

The samples coming from the input buffer are stored in dedicated queues (in the frequency offset correction block), where they wait for the coding rate evaluation and coarse frequency offset estimation to finish their processing. There is one queue for the data symbols and one for midamble symbols. The write operation to each of the queues is managed by a controller depicted in Fig. 4.10. The inputs and outputs of this controller are described in Table 4.3.

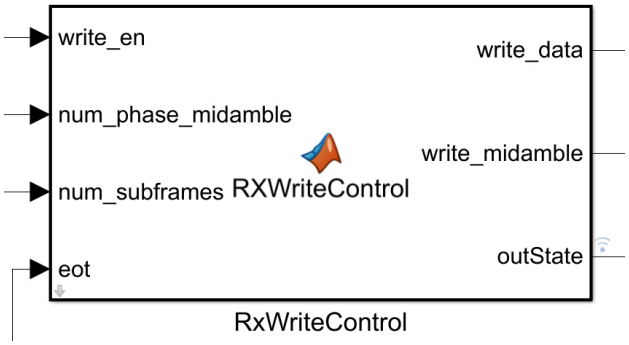

Fig. 4.10 Write controller

Table 4.3 Inputs and outputs *RxWriteControl* block

Name	Type	Description
write_en	Input	A line indicating that samples are valid and can be written to processing queues
num_phase_midamble	Input	A line that specifies the number of phase midambles in subframe (evaluated based on coding rate)
num_subframes	Input	Number of subframes in radio frame
eot	Input	A line indicating that EoT frame has been received and no further write operation is allowed until the next reception session.
write_data	Output	A line indicating that the currently processed sampes should be written to data samples queue
write_midamble	Output	A line indicating that the currently processed sampes should be written to midamble samples queue

The evaluated coding rate value is fed to a EoT Detector and CRI verifier controller (Fig. 4.11), which task is to analyze the coding rate value and recognize the EoT frame and also check whether the coding rate value is in the allowed range of 0 to 6. The inputs and outputs of the controller are described in Table 4.4.

In the case that the EoT frame is recognized the controller indicates the fact by setting the *eot_flag* which will disable some parts of the receiver until the next reception session is started. If the controller detects that the coding rate value is outside the allowed range an *override_cri* signal is set which will cause the receiver to use the default coding rate value which is 0. The frame where the coding rate evaluation is incorrect will be lost.

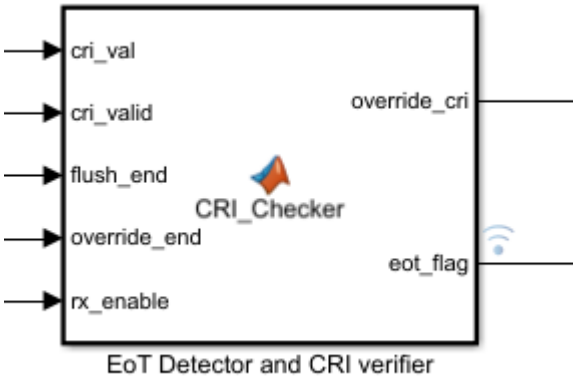

Fig. 4.11 EoT detector and coding rate verifier

Table 4.4 Inputs and outputs of EoT detector and coding rate verifier block

Name	Type	Description
cri_val	Input	Currently estimated coding rate value
cri_valid	Input	A line indicating that the cri_val is valid
flush_end	Input	A line indicating that the EoT procedure has been completed
override_end	Input	A line used to reset coding rate override in the case that the coding rate evaluation resulted in invalid value
rx_enable	Input	An indication whether the receiver is enabled.
override_flag	Output	A line indicating that the evaluated coding rate value is invalid and should be replaced with default value of 0
eot_flag	Output	A line indicating that the EoT frame has been detected used to start the EoT procedure

When the frequency offset estimation and coding rate evaluation are done, the samples stored in the data and midamble queues can be read and coarse frequency correction can be performed on these samples. The read process is controlled by the *RxFrameControl* block depicted in Fig. 4.12. The inputs and outputs of the controller are described in Table 4.5.

The samples coming from the midamble queue are fed into a midamble filter subsystem described in section 4.1.4, where the magnitude and phase of the filtered samples are calculated. The results are fed into a fine frequency and phase offset estimation block.

The samples coming from the data symbols queue are fed into fine frequency and phase offset correction block where the offsets are compensated and the result is fed into a demodulator.

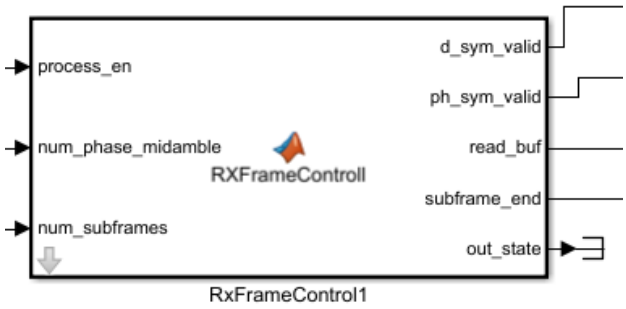

Fig. 4.12 RxFrameControl block

Table 4.5 Inputs and outputs of *RxFrameControl* block

Name	Type	Description
process_en	Input	A line indicating that coarse frequency offset estimation has been completed and the ramining part of the received samples can be processed
num_phase_midamble	Input	A line that specifies the number of phase midambles in subframe (evaluated based on coding rate)
num_subframes	Input	Number of subframes in radio frame
d_sym_valid	Output	A line indicating that the data samples should be read from processing queues
ph_sym_valid	Output	A line indicating that the midamble samples should be read from processing queues
read_buf	Output	A line indicating that samples should be read from processing queues
subframe_end	Output	A line indicating the end of a subframe

4.1.6. Fine phase and frequency offset estimation

The fine phase and frequency offset estimation subsystem is depicted in Fig. 4.13. It accepts magnitude and phase values calculated by the midamble filter described in section 4.1.4. The offsets are calculated for the phase value for which the magnitude had a maximal value. The exact algorithm is described in the report [4].

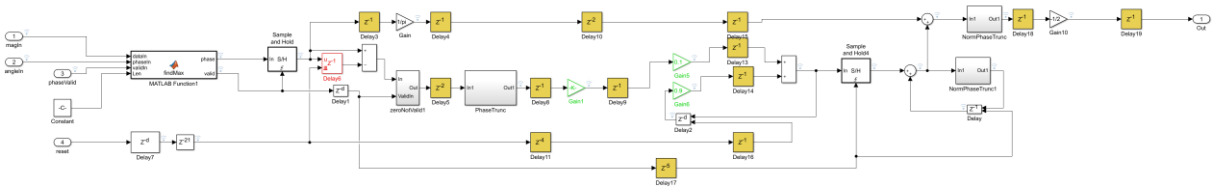

Fig. 4.13 Fine phase and frequency offset estimation

4.1.7. Demodulation

The demodulation of the OQPSK symbols is performed in the subsystem shown in Fig. 4.14. The OQPSK symbol is first converted to a QPSK equivalent by delaying the real component of the symbol by half the base sampling rate of the modulated signal. The QPSK symbols are then mapped to a likelihood metrics that can be used in the decoder.

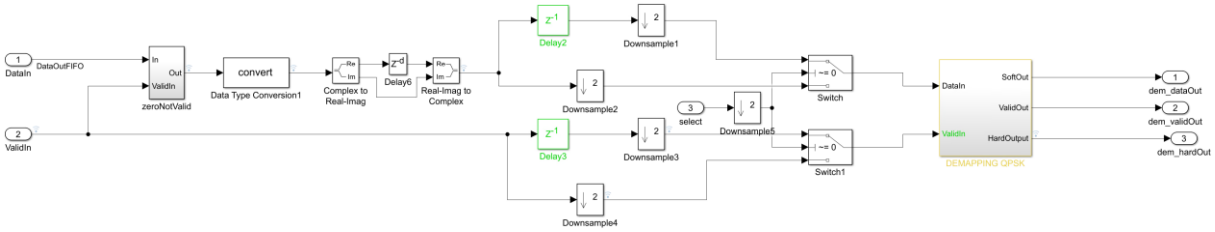

Fig. 4.14 demodulation subsystem

4.1.8. Decoding

The decoding subsystem is depicted in Fig. 4.15. It consists of a set of parallel turbo decoders. The need for more than one decoder is due to the fact that the processing delay of a single instance is too high and a single decoder would not be able to decode all the received subframes in time. Based on a thorough analysis the number of required decoders was set to 6. The decoders are run in a sequential manner and are configured in such a way that the output from each does not overlap with others, hence there is no need to add more than one CRC verification block. The CRC verification block is responsible for checking the CRC of the received data and indicating whether it is correct or not. The data with the error flag is discarded in transmission mode 0. However, for transmission mode 1 the erroneous data is analyzed and error statistics are gathered.

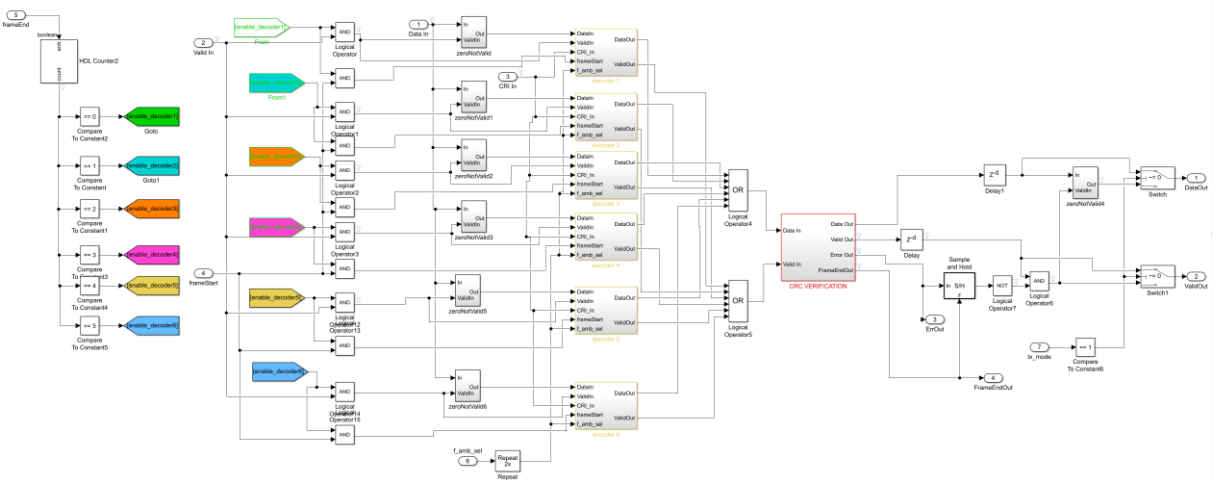

Fig. 4.15 Decoding subsystem

4.1.9. Data packing

The final step in the receiver processing chain is the data packing operation. The block responsible for this task is depicted in Fig. 4.16. Its main goal is to convert bits into 32-bit words that are accepted by the software. In addition, this block is also responsible for detecting fake subframes which are either discarded in transmission mode 0 or analyzed for bit errors in transmission mode 1.

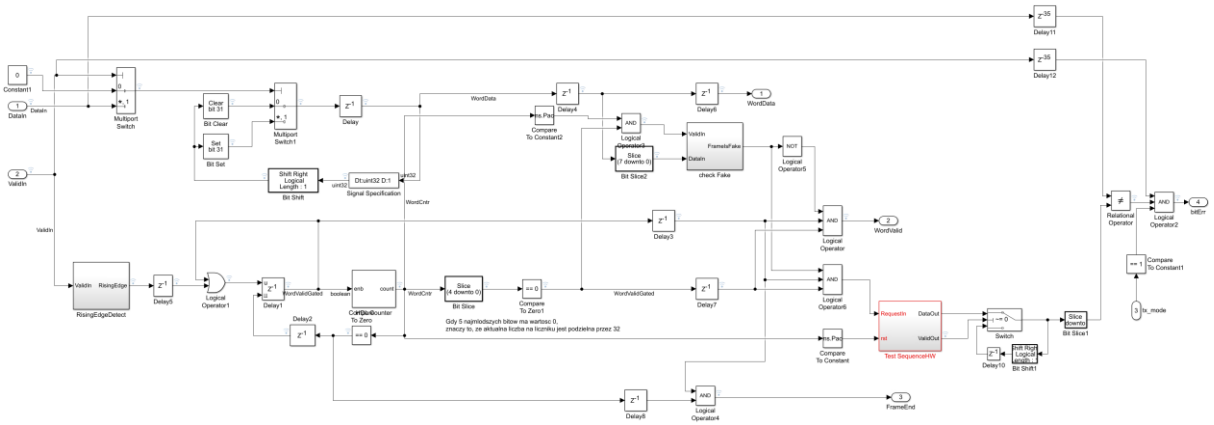

Fig. 4.16 Data packing subsystem

4.2. Custom-made IP cores

4.2.1. Introduction

The FPGA-based signal processing routine, created with the aid of MATLAB HDL Coder, is encapsulated into a user-developed Vivado IP core. To incorporate a receiver IP core into a Vivado block diagram, a Matlab HDL workflow coder is used. The workflow operates according to a given so-called reference design, which specifies the Vivado block diagram, the way in which the custom-made IP core is merged with it, and the board pin assignment (constraint file). The reference designs have a form of TCL scripts. In the current project, a reference design by Analog Devices, dedicated to Mini-ITX device, equipped with Xilinx Zynq®-7000 All Programmable SoC XC7Z045 and an FMC2/3/4 transceiver card with AD9361, is used as a baseline. The device used in the project is Mini-ITX with XC7Z100 plus the FMC2/3/4 card, so the reference design requires some customization. Another reason to do that is the fact that the original reference design seems to be most accurate in the cases where the IQ samples (for 2 receiver channels) are conveyed through PL directly to PS since there are four interleaved 16-bit data lines on the PL to PS interface. Signal processing in PL is an option (the user-specified IP core can be by-passed in some cases).
In the current project, the data transferred from PL to PS via DMA have the meaning of binary vectors instead of complex IQ samples; the data rate on the PL<>PS interface is significantly smaller than the symbol rate on the SoC<>AD9361 interface as there is nothing else but PL responsible for the physical-layer signal processing (frame detection, demodulation, channel estimation, channel decoding, descrambling, etc.). As a consequence, it is more accurate to consider only one wide data line on the PL<>PS interface. The data vectors to be transferred to PS via DMA are generated asynchronously once a new frame has been acquired and decoded with no errors. For that reason it is necessary to wire a “data valid” line along with the data line.
Another problem to be solved when using MiniITX-XC7Z100 part is that Vivado has been lacking its definition for a few years. To overcome that issue, some tweaks in Vivado installation are necessary. Making them results with appearance of MiniITX-XC7Z100 part on a board selection pane (see Fig. 4.17) and xc7z100ffg900-2 SoC in project settings (Fig. 4.18)

4.2.2. Reference design customization

With the aim to overcome the disadvantages of the original reference design, the datapath is significantly modified. Four 16-bit data lines have been replaced with one 32-bit data line. Consequently, data streams are not interleaved anymore (interleaving required troublesome synchronization between the streams), so the blocks responsible for stream interleaving and deinterleaving are removed. The *utility_buffer* IP, originally placed between FIFO at the clock domains’ border and the DMA interface, has been removed. It was devoted to alleviate the problem of asynchronous type of data feed through DMA interface, but it has appeared to cause highly undesired random delays.

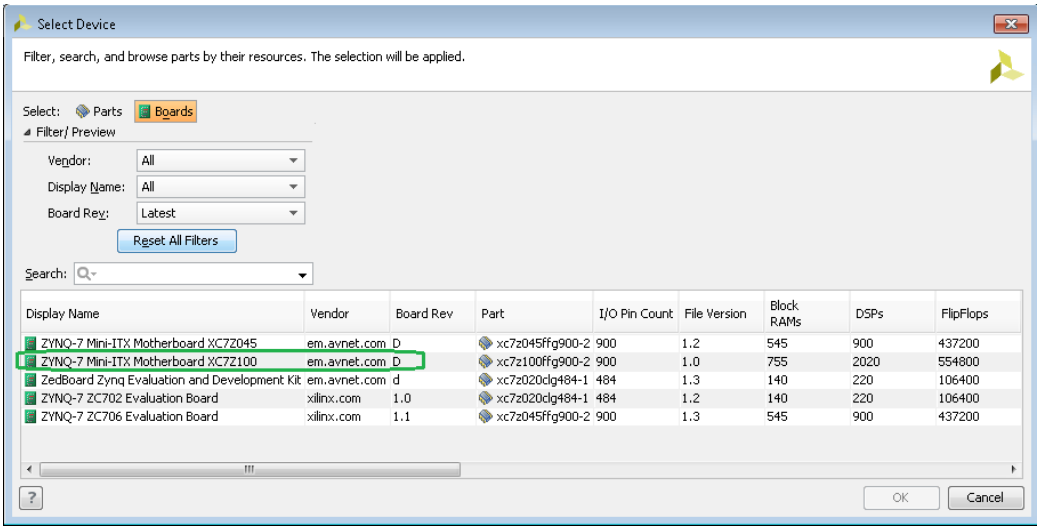

Fig. 4.17 Device selection window in Vivado with the desired board included

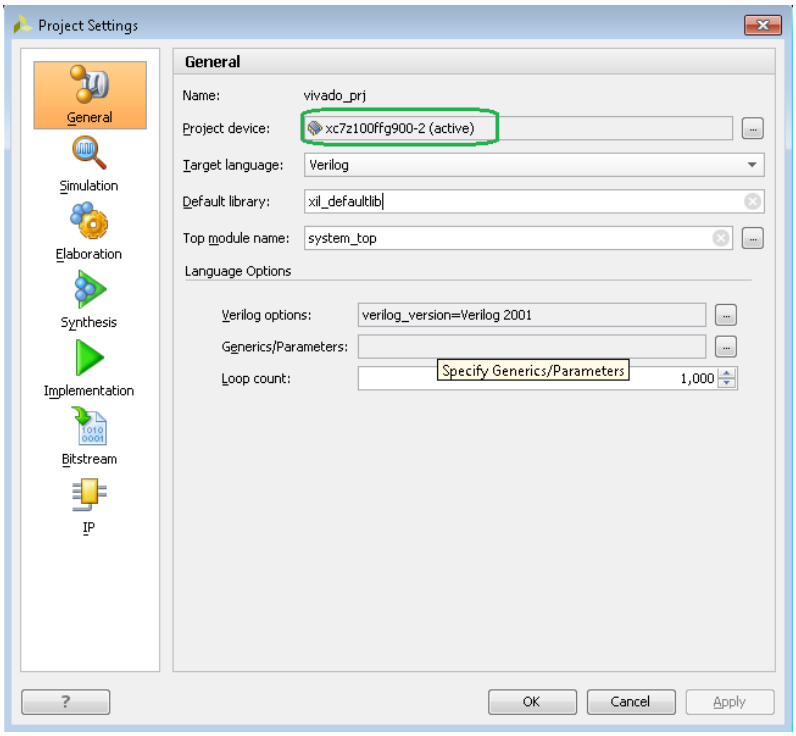

Fig. 4.18 Desired SoC version shown in the project settings

The genuine reference design by Analog Devices features more IP cores useless from the perspective of the current project. In particular, it refers to the IP cores playing the role of HDMI, SPDIF, and I2S interfaces; their removal brings reasonable FPGA resources savings. Note that removing unnecessary IP cores involves modification of both wrapper- and system top HDL files.

Some minor changes, shown in Fig. 4.19, have been made in the settings of *axi_ad9361* IP core, responsible for transferring IQ passband signal samples, received by AD9361, to FPGA.

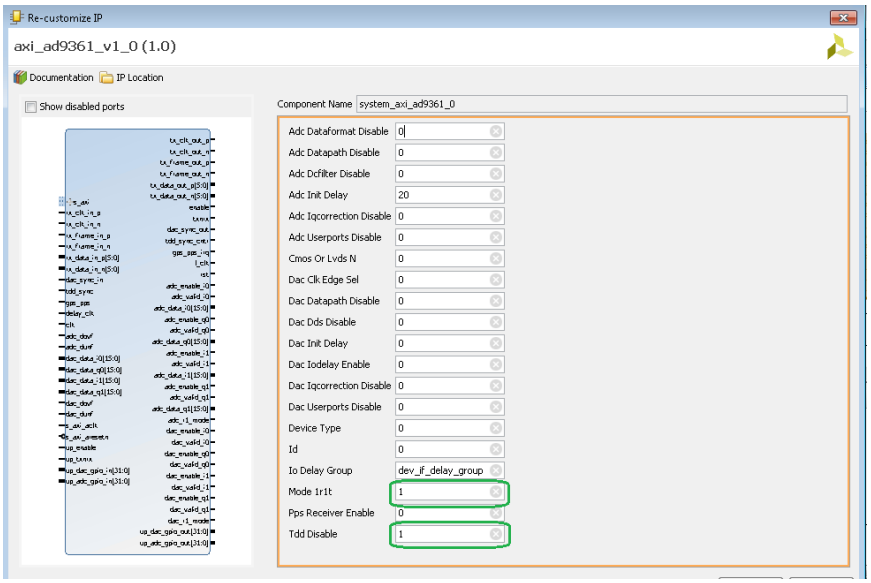

Fig. 4.19 Configuration window of axi_ad9361 IP core

In detail, *1R1T mode* is chosen to eliminate redundant support for two receive channels (according to the project assumptions, only one receive channel is utilized). Additionally, *TDD disable* option is checked, since the AD9361 is forced to operate permanently in Rx mode by an SPI write to AD9361 registers instead of periodic Tx/Rx toggling, controlled via FPGA pins. Thanks to that, neither 24-bit TDD counter nor a few reference registers of the same size are implemented in FPGA. The rest of configuration fields of axi_ad9361 IP core take the default values. DDS feature is enabled for testing purposes.

Another improvement has been made in the domain of custom-made IP core clocking. In the original reference design, the user's IP core is clocked by the AD9361 clock divided by 2 (or by 4 in the case of 2 transmit streams – not applicable to the current design). It limits the system capability of serial data processing, since half of the clock cycles are not usable. Instead, the custom-made IP core is now clocked with the original AD9361 clock (*rx_clk*), distributed throughout the FPGA device directly from a respective BUFG element.

The decision to eliminate a separate clock domain for custom-made IP core results with a simpler clock cross-domain management: there is only one clock-domain crossing in the data path, handled safely by means of a FIFO in *axi_ad9361_adc_dma* IP core. To transfer commands and status messages data between the time domains (*clk_fpga_0* and *rx_clk*) through AXI4-Lite, a 3-stage synchronizer is placed in the *axi_cpu_interconnect* IP core. Together with AXI protocol handshaking, it guarantees safe transfers.

Taking into account asynchronous data transfer between the clock domains (for both data path and the control/status AXI channel), it is desired to constrain intra-clock paths on FPGA as false paths, thereby instructing the Vivado placer to ignore them; it helps overcome timing-related issues when routing. Fig. 4.20 proves that the paths between clock domains: *clk_fpga_0* and *rx_clk* are successfully set as false paths in Vivado.

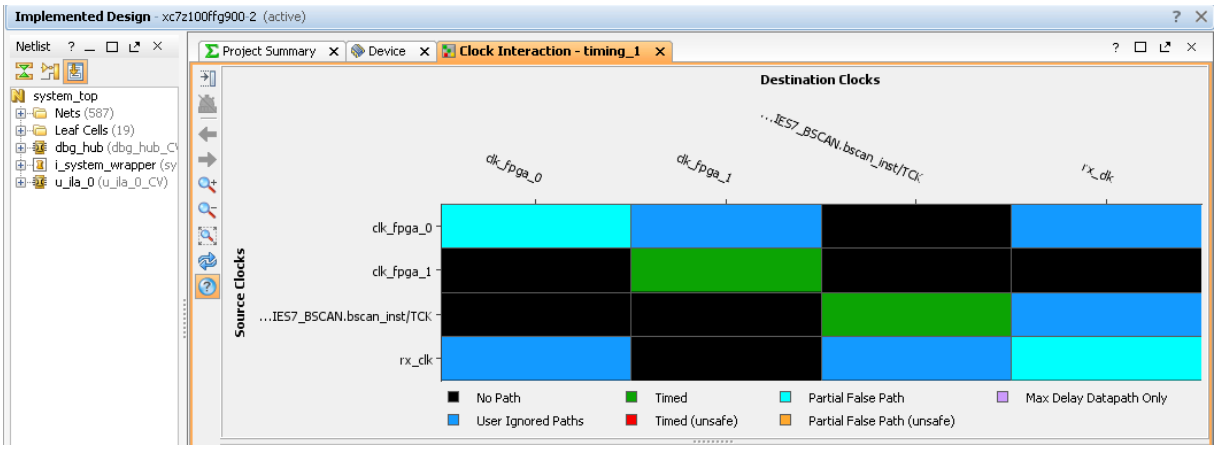

Fig. 4.20 Clock interaction report for the implemented design in Vivado

Not only is the MATLAB HDL workflow responsible for generating appropriate interfaces of the custom-made IP core and incorporating it into the reference design, but also for attaching extraordinary constraint files to the project. The constraint files contain the settings related to the hardware pinout, clock frequency, false paths, etc. The pinout for MiniITX-XC7Z100 is identical as for MiniITX-XC7Z045, so the constraint files related to the pins' voltage and their assignment imported from the original reference design are kept untouched.

4.2.3. The use of Simulink HDL Workflow Advisor

To avoid compatibility issues with a quite outdated reference design for MiniITX-XC7Z045, the authors have decided to target their own MiniITX-XC7Z100 reference design for Vivado 2016.4. It has been integrated with the HDL Workflow of Matlab 2017b under the name of *MiniITX* – it should be chosen as the Target platform in Step 1.1 of the HDL Workflow Advisor, as shown in Fig. 4.21. Note that the accurate MiniITX device (xc7z100) is associated with it, automatically. As the result of researchers' efforts, Step 1.2 (shown in Fig. 4.22) enables setting specific board peripherals to be used or not. Thanks to that one can decide to activate: LEDs, buttons, or DIP switches to control or observe selected lines. According to his/her choice, one of alternative reference designs is loaded.
Step 1.3, shown in Fig. 4.23, brings the possibility to connect the inputs and outputs of the developed Simulink block diagram to appropriate reference design wires (aka target platform interfaces). Note the possibility to choose LEDs in the middle column. The use of board peripherals, like LEDs, buttons, DIP switches must match the choices made in Step 1.2. If not, an error message will appear. The meaning of specific target platform interfaces is explained in Table 4.6. It does not include the Simulink ports attached to the AXI4-Lite interface, used to send control commands from PS to PL and read diagnostic messages in the opposite direction.
After passing checks in Steps 2.1-2.4, the HDL code for the custom-made IP core is generated in Step 3.2 of HDL Workflow Advisor. The generated IP core is deposited in a folder specified by the user and can be manually placed into any Vivado block diagram. However, the customized reference design *MiniITX* features the possibility to automatically integrate the IP core with the block diagram. It can be done in Step 4.1 of HDL Workflow Advisor. If the process ends successfully, a link to a new-created Vivado project appears in a log window of HDL Workflow Advisor, as shown in Fig. 4.24). Clicking the link launches Vivado and the project opens. It is not suggested to run remaining steps of HDL Workflow Advisor, as they are accurate only for the case when FPGA processing is controlled by Simulink (a kind of hardware-software co-simulation).

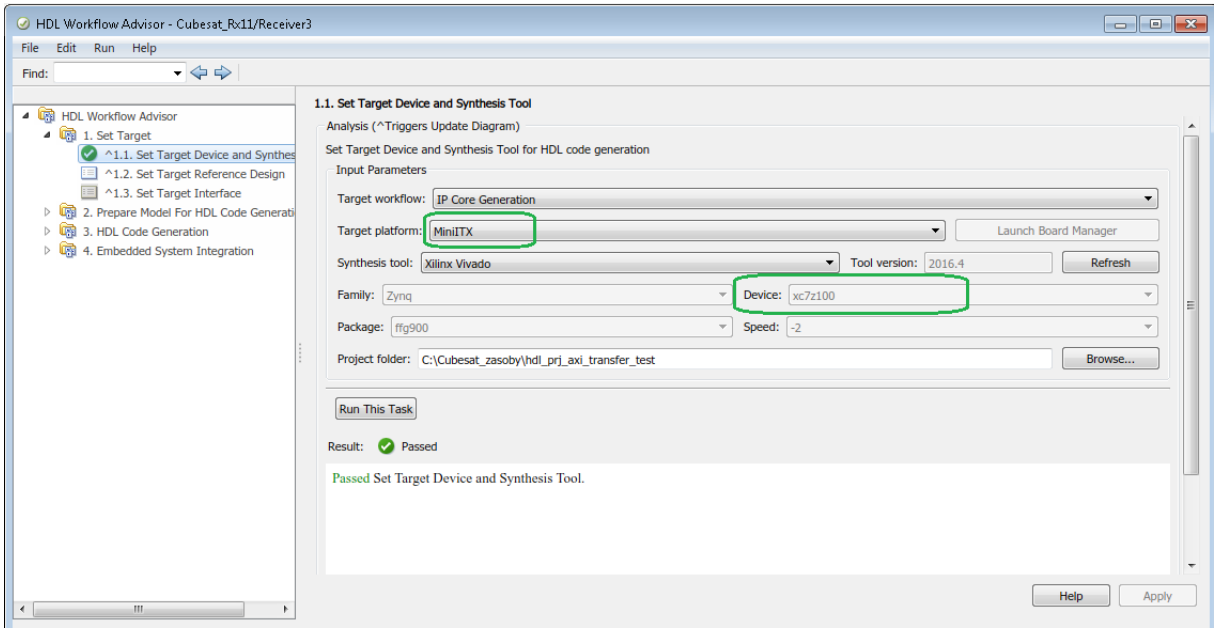

Fig. 4.21 Step 1.1 of HDL Workflow Advisor

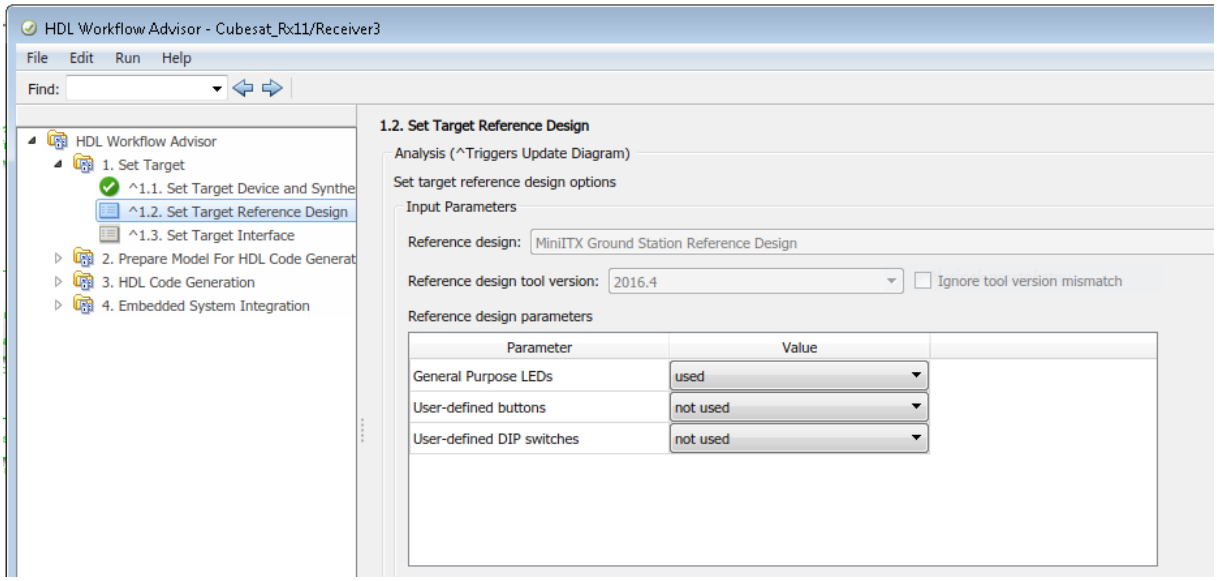

Fig. 4.22 Step 1.2 of HDL Workflow Advisor

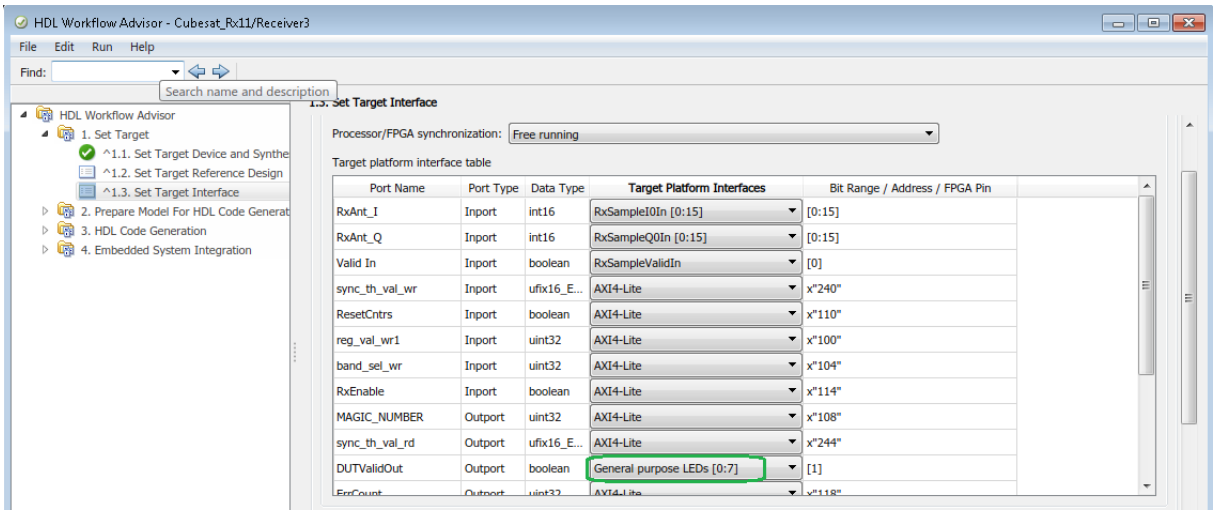

Fig. 4.23 Step 1.3 of HDL Workflow Advisor (assignment of target platform interfaces to ports is random)

Table 4.6 Target platform interfaces in HDL Workflow Advisor

Name	Type	Mating pin on Simulink diagram	Description
RxSampleI0In *RxSampleQ0In*	Input	*RxAnt_I* *RxAnt_Q*	16-bit inputs for IQ samples of the passband signal, represented in 2's complement format; actually, 4 leading bits are redundant
RxSampleValidIn	Input	*Valid In*	This line is periodically strobed by *axi_ad9361* IP core to indicate useful samples; for 1R1T mode, a pulse appears every 2nd AD9361 clock cycle
RxDataOut	Output	*DUTDataOut*	32-bit vector conveying received decoded data to PS via DMA
RxDataValid	Output	*DutValidOut*	Strobe for *RxDataOut* – it forces FIFO at the clock domain border to accept another data load

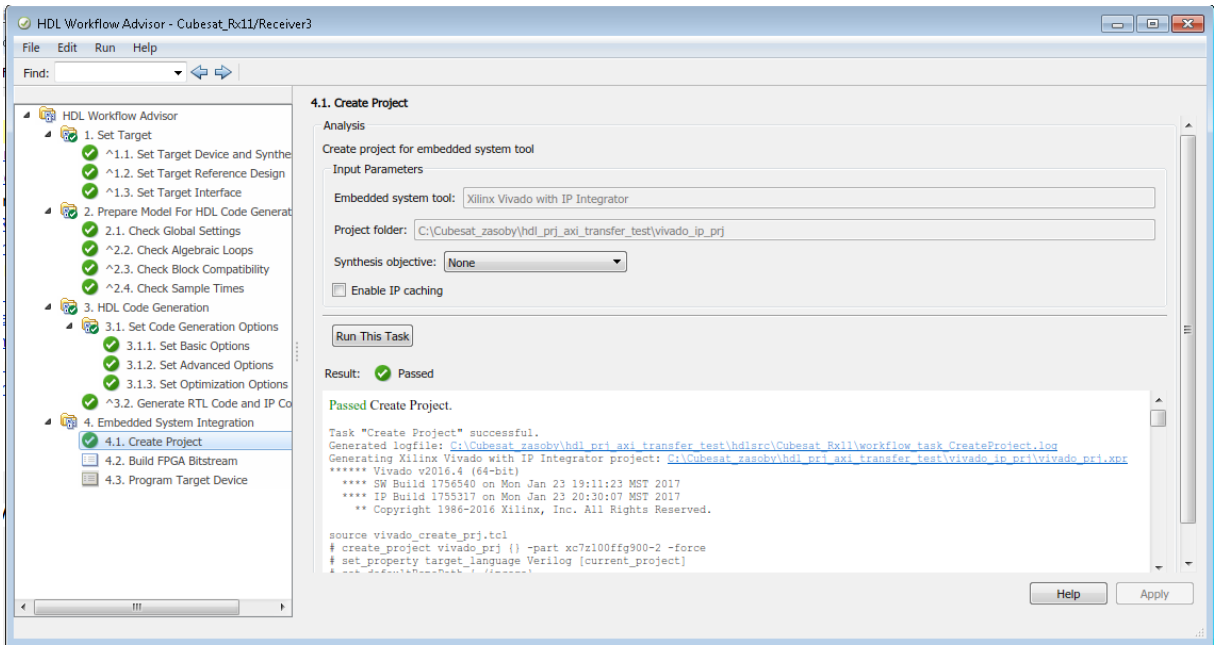

Fig. 4.24 Result of successful execution of the last step of HDL Workflow Advisor

4.2.4. Vivado project details

The complete block design of Vivado project is shown in Fig. 4.25, while a closeup on the custom-made IP core is presented in Fig. 4.26. The target platform interfaces from Matlab HDL Workflow Advisor are mapped to legacy IP core interfaces in Vivado block diagram according to Table 4.7.

Table 4.7 Target platform interfaces mapping

Name in HDL workflow advisor	**IP core interface name in Vivado block design**
RxSampleI0In *RxSampleQ0In*	*sys_wfifo_0_dma_wdata* *sys_wfifo_1_dma_data*
RxSampleValidIn	*sys_wfifo_valid_in*
RxDataOut	*dut_data_0*
RxDataValid	*dut_data_valid*

There are some additional lines: AXI4-Lite bus (to receive control commands from PS and send status messages), as well as reset and clock lines (separate for AXI bus sub-module and the rest of the IP core). Since the clock-domain crossing is located in *axi_cpu_interconnent*, the whole custom-made IP core clocking belongs to a single clock domain of *rx_clk*, originated from *l_clk* pin of *axi_ad9361* block. The IP core reset line is conjugated with PS reset by *util_ad9361_divclk_reset block*, responsible for transferring the PS-generated reset to *rx_clk* clock domain. Note that the IP core must be additionally resetted by an AXI write after AD9361 has finished all callibrations. Failure to do so might lead to unpredictable IP core operation and metastability.
The design is synthesized with the clock constraints specified according to the most demanding 20 MHz bandwidth transmission mode. The mode choice is managed by appropriate frequency setting of *rx_clk* on AD9361 and impacts the speed of data passing through the whole data path in the *rx_clk* domain.

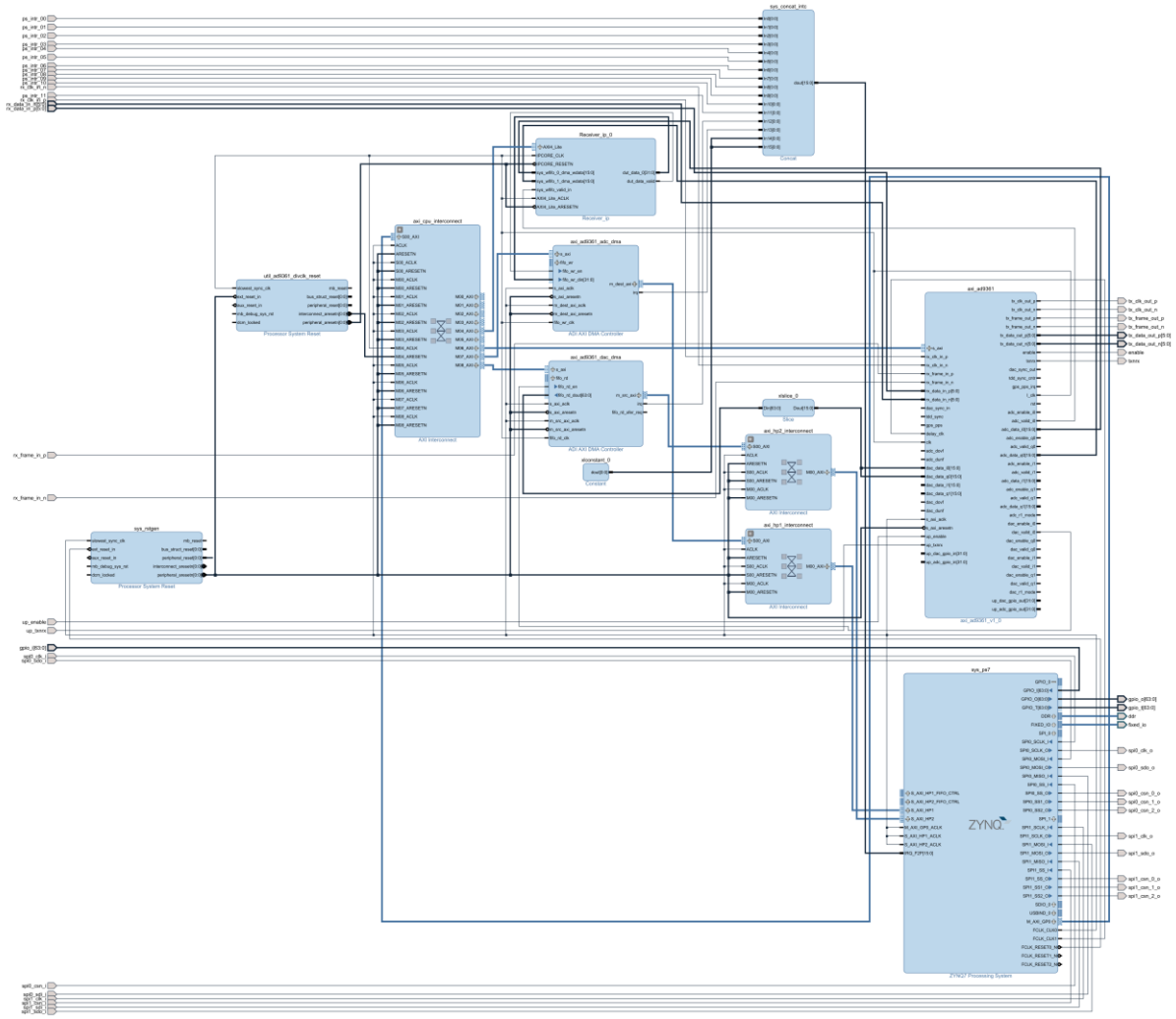

Fig. 4.25 Vivado block design

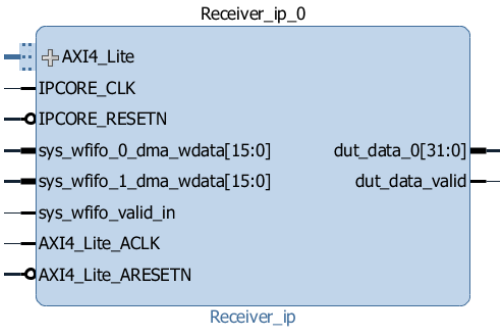

Fig. 4.26 Custom-made IP core

5. Receiver implementation – PS part (software) for "hardware oriented" version

5.1. TX vs RX – disambiguation

Processing System part is run on two boards, mainly:

- Trenz board (board version: TE0715-04-30-1I3),
- MITX (aka MiniITX board with Z100 FPGA, board version: Mini-ITX-7Z-ASY-G).

The most recent HDL version Analog Devices' HDL that supports AVNET's MiniITX box is hdl_2017_r1, while HDL version that is compatible with Trenz board is hdl_2018_r2. A chain of dependencies caused by such a seemingly unimportant version difference results in two different Linux O/S versions that run on those boards. It is of an uttermost importance to note that such a choice of O/S version was dictated by technical arguments, not by dogmatic or opinionated ones. Details are shown in the table below.

Side	Board	HDL	O/S	Vivado	Matlab
TX	Trenz TE0715-04-30-1I3	hdl_2018_r2	2018_R2	2018.2	2019a
RX	Mini-ITX-7Z-ASY-G	hdl_2017_r1	2017_R1	2016.4	2017b

5.2. Operating System

Transmitter module is based on Trenz board TE0715-04-30-1I3, a consequence of which is using Analog Devices' Linux version `2018_R2` (kernel 4.14.0). [Note, however, that due to the unavailability of the final board at the time of preparation of this document, the chosen version of O/S was tested only on Trenz's motherboard TE0705-04. As a result, this O/S version is not yet decided to be final.]

5.2.1. Cross-compiling Tools

Linux kernel together with all supporting libraries and tools were built with GCC 11.2.0. [As of the preparation time of this document, the final version of GCC is practically frozen, although it still might be changed to a different one if necessary.]

5.2.2. Shell

O/S is interfaced via `ash` (Almquist shell).

5.2.3. Kernel Configuration

Kernel was configured using a customized Xilinx configuration provided by Analog Devices Inc. in the source tree of Linux kernel under the name [`xilinx_zynq_defconfig`]. The customization involved additional configuration of:

- DMA Engines (with Xilinx DMA Engines),
- AXI DMAC,

- AD9361,
- AD9517, and
- AXI DDS (Digital Direct Synthesizer).

5.2.4. FPGA Driver and Kernel Modules

FPGA Driver is loaded as a kernel module during the system boot-up. Detailed documentation of the driver attributes can be found in the auto generated documentation file [10]. Delivered as an attachment to this document. [Note, however, that eventual further changes to the implementation of the IP Core may require corresponding updates to the driver documentation].

5.2.4.1. Driver Attributes Description

As mentioned previously, the documentation describing driver attributes is automatically generated. A similar approach is used to create the driver code (obviously, only the most redundant parts). This is achieved by describing driver's attributed in a YAML file, which in turn is used to generate driver's and Latex's code for the driver itself and its documentation, respectively. Such an approach was used in order to easily keep in sync changes made to the driver and its documentation. An example entrance that describes the `hardware_version` attribute is shown in the listing below.

```
# Snip...
- name: hardware_version
  generated_driver_code:
[offsets,driver_entrances,driver_attributes_short,help_messages]
  type: __u32
  rd_offset: 0x04■■
  wr_offset: null
  rd_buffer_size: 16
  wr_buffer_size: 10
  rd_function: scnprintf
  wr_function: kstrtou32
  mask_spec: null
  trx_side: [tx,rx]
  help_msg: |-
      None
  description: |-
      Version of bit-stream hardcoded in the hardware.
      It is not possible to write into this register!
  hardware_name: DEV_ID
  available_values_hardware: null
# Snip...
```

5.2.4.2. Kernel Modules Loading

Despite most of the kernel modules being compiled directly into the kernel itself, the FPGA driver is not. Such an approach allows eventual changes to the driver without the need of recompiling the whole kernel. The script used to load / unload kernel modules (which is located in [`/etc/rc.d/init.d/modules_conf`]) is presented below. Note that contrary to most arguments that such scripts accept, this particular one also accept arguments: `load`, `unload` and `reload` (which corresponds to standard: `start`, `stop` and `restart`, respectively). Such an approach allows using semantic that is closer to module loading / unloading. The configuration file that drives the modules loading is located in [`/etc/modules.d/modules.conf`] and it is discussed in the next section.

```
#!/bin/ash
#
# modules auto loading/unloading
#

. /etc/rc.d/init.d/functions

modules_config_file="/etc/modules.d/modules.conf"

modules_load_unload() {
    local modprobe_param="${1}"
    local msg="loading"
    [ "${modprobe_param}" == "-r" ] && msg="unloading";


    echo -n "${0}: Checking whether ${modules_config_file} exists and is valid: "
    [ -r ${modules_config_file} ] && grep -qv "^($|#)" ${modules_config_file}
    local ERR=$?
    [ 0 == ${ERR} ] && true || false
    check_status
    [ 0 == ${ERR} ] || { echo "${0} Not ${msg} modules! Problems with
${modules_config_file} file!"; exit 0; }

    sleep 0.1

    while read module args; do
        # Skipping blank or commented-out lines"
        case "${module}" in
            ""|"#"*) continue;;
        esac

        modprobe ${modprobe_param} ${module} ${args}
        ERR=$?
        echo -n "${0}: ${msg} ${module} with params: `[ "" != "${args}" ] && echo
${args} || echo -*NONE*-`: "
        [ 0 == ${ERR} ] && true || false
        check_status
        sleep 0.1

    done < ${modules_config_file}

}

case "$1" in
    start|load)
        modules_load_unload ""
    ;;

    unload|stop)
        modules_load_unload "-r"
    ;;
    reload|restart)
        $0 stop

        sleep 1

        $0 start
    ;;
    *)
        echo "Usage: ${0} {start|stop|restart|load|unload|reload}"
        exit 1
    ;;
esac

exit 0
```

5.2.4.2.1. Modules configuration file

The aforementioned modules configuration file (which is located in [`/etc/modules.d/modules.conf`]) allows loading arbitrary modules (not only the FPGA driver). The example script is shown in the listing below. Besides specifying modules to load, it also allows specifying module's parameter, e.g., in the listing below module `fpgatrx` is loaded with parameter `DEBUG` set to `1`. Kernel object files that contain the modules' code are located in [`/lib/modules/$(uname -r)/kernel/drivers/`], where [`$(uname -r)`] is release of the running kernel, in our case it is: `4.14.0-xilinx-ge77ffb40e9a0-dirty`. [But as already mentioned this release might still be subject to an eventual change.]

```
# File: /etc/modules.d/modules.conf
#
# In order to load module at the system boot-up, add:
#
# module_name module_param_1 module_param_2
#

fpgatrx DEBUG=1
```

5.2.5. Libraries and Tools

O/S is delivered with tools and libraries described in the table below. [Please note that the final versions of some of these artifacts may be changed if deemed necessary.]

Tools / Library	Version TX	Version RX	Description
Binutils	2.27		Set of tools and libraries for building binary executable(s), e.g., liker, assembler, etc. All build for ARM Cortex A9 processor, but without dubious optimisation flags.
Busybox	1.24.2		Swiss-army-knife toolbox with standard set of tools for working in a Linux environment. (All tools are delivered as via symbolic links to one executable.)
IANA-ETC	2.30 patched		Data / information package for network protocols and services.
MPC	1.0.3		Arbitrary precision floating-point complex arithmetic library. (GCC dependency.)
MPFR	3.1.4		Arbitrary precision floating-point library. (GCC dependency.)
musl-libc	1.1.19		Standard C library for embedded systems.
zlib	1.2.11		Data compression library.
netplug	1.2.9.2		GNU/Linux daemon for network services.
Dropbear	2018.76		Lightweight implementation of SSH library.
LibXML2	2.9.8		XML parsing library, implemented in C. libiio dependency.
Boost	1.67		Boost – an umbrella of C++ utility libraries. Most of them are header-only libraries. Only three are installed on the final system:

			`libboost_atomic`, `libboost_chrono`, `libboost_system`.
tree	1.7.0	1.7.0	Command line recursive directory viewer / explorer.
libiio	0.14		Hardware abstraction layer library via IIO module (Industrial Input / Output) for GNU/Linux. Mainly used to
gtest	1.11.0	1.11.0	Unit test library.
gflags	2.2.1		Command line parsing library.
googlebenchmark	1.6.1	1.6.1	Benchmarking library
{fmt}	8.1.1	8.1.1	Text formatting library
iproute2	ss190197		Network support tools.

5.2.6. Device Tree and Node Configuration

In order to easily distinguish between various systems configurations we add to the device tree file parameter describing specific configuration of the board. An excerpt from a device tree is shown below.

```
/{
    wzldevicemode {
        mode = "trenz";
    };
};
```

On the running system, current [`wzldevicemode`] (WZL here stands for **W**ireless **Z**YNQ **L**ab) can be read form [`/sys/firmware/devicetree/base/wzldevicemode/mode`] file. In the case the node describing the current configuration changes, the file with the fixed name that contains the actual location of the current configuration is located in [`/etc/radio/wzl-dev-mode-file-location`]. Such an approach ensures a single reference point to the actual location of the the file describing the device mode.

Definition of the FPGA implementation of the custom made IP Core is also provided in the device tree (in the `FPGA/amba_pl` section). The entrance in the device tree for `SimpleQPSK` IP Core is shown below.

```
/{
    amba_pl: amba_pl {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "simple-bus";
        ranges ;
        SimpleQPSK_ip_0: SimpleQPSK_ip@43c00000 {
            compatible = "xlnx,SimpleQPSK-ip-1.1";
            reg = <0x43c00000 0x10000>;
        };
        // ...
    };
};
```

5.2.7. RF Configuration

RF configuration files reside in the [/etc/radio] directory on the primary/root partition. An example listing of subset of its directories is shown below.

```
/etc/radio/filters
├── cubesat-filter-v0001.ftr
├── cubesat-filter-v0002-1R1T-mode.ftr
├── cubesat-filter-v0003-61dot44.ftr
├── cubesat-filter-v0004-30dot72.ftr
├── cubesat-filter-v0005-7dot68-pll-ad9364.ftr
├── cubesat-filter-v0006-15dot36-pll-ad9364.ftr
└── lte_5MHz.ftr

/etc/radio/current/
├── ad9361-config.gflags
├── config-dispatcher.gflags
├── session-plan.yaml
└── session-scheduler-config.gflags
```

As can be deduced from the listing above, definition of FIR filters is in [/etc/radio/filters] directory. An example FIR filter definition file is show below. Its format is self-explanatory.

```
$ head -12 /etc/radio/filters/lte_5MHz.ftr
# Generated with AD9361 Filter Design Wizard 16.1.3
# MATLAB 9.2.0.538062 (R2017a), 25-May-2018 16:55:22
# Inputs:
# Data Sample Frequency = 7680000 Hz
TX 3 GAIN 0 INT 2
RX 3 GAIN -6 DEC 2
RTX 983040000 122880000 61440000 30720000 15360000 7680000
RRX 983040000 122880000 61440000 30720000 15360000 7680000
BWTX 4372840
BWRX 4694670
-5,-10
0,-21
...
```

File [lte_5MHz.ftr] is used only for demonstration without disclosing actual details of the FIR filters used in the real system (mainly number and values of consecutive filter taps).

5.2.8. Application Configuration Files

Besides FIR filters configuration files [/etc/radio] directory also contains

```
/etc/radio/current/
├── ad9361-config.gflags
├── config-dispatcher.gflags
├── session-plan.yaml
└── session-scheduler-config.gflags
```

These files contain configurations of the custom applications and are described more thoroughly further in the document. The [`/etc/radio/current`] directory is in fact a soft link to the actual directory that contains configuration for a particular transmission side (TX or RX).

5.2.9. Pre-O/S Components and Boot Sequence / Order

Boot sequence on ARM-based hardware is divided into separate stages. Initially the FSBL (First Stage Boot Loader) prepares hardware, initializes CPUs and starts SSBL (Second Stage Boot Loader), which in our case is U-boot. Then SSBL/U-Boot decompresses the Linux kernel image and loads it together with a device tree describing the hardware and peripherals into memory. Next, the control is passed to the kernel, which boots itself, launches [`/sbin/init`] program that finalizes the Linux booting-up and starts services and applications required for ensuring the whole system is in an operational state.

5.2.9.1. FSBL

Beside standard initialization, Xilinx's FSBL allows configuring additional hardware via FSBL hooks. For example, patch provided by Trenz allows configuring SI5338 module.

In the next three sections we show logs from FSBL, U-Boot and loading Linux kernel. These logs can be used as a reference for adjusting and / or fine-tuning different versions of the mentioned software components. They should be treated more as a guidance that a gold-standard when preparing custom solutions.

5.2.9.1.1. FSBL Boot Logs

FSBL loading logs are presented below. The manifest info section describes internals used to create a final [`BOOT.BIN`] file, it is not a necessary part and it is used solely for simplifying identification of the loaded bitstream.

```
MANIFEST INFO:
-----------------------
HDF FILE:                 sr-cubesat-trenz-tx-v0011.hdf
HDF GIT SHA:              8b041aee83724fadcd64867d266cabf8cdbfb005
IP CORE REPORT PATH:
d:/TrenzPrebuild/system/ip_lib/SimpleQPSK_ip_v1_1/doc/doc_arch_axi4_lite.jpg
ZYNQ XTOOLCHAIN GIT SHA:  bad57fe9919f23f83c19aeeb71f0a3bb37e2e70a
FSBL build date:          Tue, 14 Dec 2021 17:12:32 +0100
-----------------------
Xilinx Zynq First Stage Boot Loader (TE + PUT/TGM modified)
Release 2018.2  Dec 14 2021-17:12:50
```

5.2.9.2. SSBL / U-boot

U-boot logs are only for the reference.

```
U-Boot 2018.01 (Oct 11 2021 - 16:38:10 +0200) Xilinx Zynq ZC702

Board: Xilinx Zynq
Silicon: v3.1
I2C:   ready
DRAM:  ECC disabled 1 GiB
MMC:   sdhci@e0100000: 0 (SD)
** No device specified **
Using default environment
```

```
In:    serial@e0000000
Out:   serial@e0000000
Err:   serial@e0000000
Board: Xilinx Zynq
Silicon: v3.1
Net:   ZYNQ GEM: e000b000, phyaddr ffffffff, interface rgmii-id
eth0: ethernet@e000b000

U-BOOT for petalinux

ethernet@e000b000 Waiting for PHY auto negotiation to complete..... TIMEOUT !
Hit any key to stop autoboot:  0
reading uEnv.txt
486 bytes read in 12 ms (39.1 KiB/s)
Loaded environment from uEnv.txt
Importing environment from SD ...
Running uenvcmd ...
Copying Linux from SD to RAM...
reading uImage
4076304 bytes read in 238 ms (16.3 MiB/s)
reading devicetree.dtb
10683 bytes read in 17 ms (613.3 KiB/s)
** No boot file defined **
```

5.2.9.3. Linux Kernel

Linux kernel logs are only the reference.

```
## Booting kernel from Legacy Image at 03000000 ...
   Image Name:   Linux-4.14.0-xilinx-ge77ffb40e9a
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    4076240 Bytes = 3.9 MiB
   Load Address: 00008000
   Entry Point:  00008000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 02a00000
   Booting using the fdt blob at 0x2a00000
   Loading Kernel Image ... OK
   Loading Device Tree to 07ffa000, end 07fff9ba ... OK

Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 4.14.0-xilinx-ge77ffb40e9a0-dirty (tgm@asus) (gcc version 8.3.0
(GCC)) #1 SMP PREEMPT Mon Oct 25 18:38:15 CEST 2021
CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=18c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
OF: fdt: Machine model: xlnx,zynq-7000
Memory policy: Data cache writealloc
cma: Reserved 16 MiB at 0x3f000000
random: fast init done
percpu: Embedded 16 pages/cpu @ef7cf000 s35084 r8192 d22260 u65536
Built 1 zonelists, mobility grouping on.  Total pages: 260608
Kernel command line: console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk
rootfstype=ext4 rootwait
PID hash table entries: 4096 (order: 2, 16384 bytes)
Dentry cache hash table entries: 131072 (order: 7, 524288 bytes)
Inode-cache hash table entries: 65536 (order: 6, 262144 bytes)
Memory: 1012868K/1048576K available (6144K kernel code, 266K rwdata, 1672K
rodata, 1024K init, 152K bss, 19324K reserved, 16384K cma-reserved, 24576)
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
    fixmap  : 0xffc00000 - 0xfff00000   (3072 kB)
    vmalloc : 0xf0800000 - 0xff800000   ( 240 MB)
    lowmem  : 0xc0000000 - 0xf0000000   ( 768 MB)
    pkmap   : 0xbfe00000 - 0xc0000000   (   2 MB)
```

```
    modules : 0xbf000000 - 0xbfe00000   (  14 MB)
      .text : 0xc0008000 - 0xc0700000   (7136 kB)
      .init : 0xc0900000 - 0xc0a00000   (1024 kB)
      .data : 0xc0a00000 - 0xc0a42a80   ( 267 kB)
       .bss : 0xc0a42a80 - 0xc0a68e44   ( 153 kB)
Preemptible hierarchical RCU implementation.
        RCU restricting CPUs from NR_CPUS=4 to nr_cpu_ids=2.
        Tasks RCU enabled.
RCU: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=2
NR_IRQS: 16, nr_irqs: 16, preallocated irqs: 16
efuse mapped to f0800000
slcr mapped to f0802000
L2C: platform modifies aux control register: 0x72360000 -> 0x72760000
L2C: DT/platform modifies aux control register: 0x72360000 -> 0x72760000
L2C-310 erratum 769419 enabled
L2C-310 enabling early BRESP for Cortex-A9
L2C-310 full line of zeros enabled for Cortex-A9
L2C-310 ID prefetch enabled, offset 1 lines
L2C-310 dynamic clock gating enabled, standby mode enabled
L2C-310 cache controller enabled, 8 ways, 512 kB
L2C-310: CACHE_ID 0x410000c8, AUX_CTRL 0x76760001
zynq_clock_init: clkc starts at f0802100
Zynq clock init
clocksource: ttc_clocksource: mask: 0xffff max_cycles: 0xffff, max_idle_ns:
537538477 ns
sched_clock: 16 bits at 54kHz, resolution 18432ns, wraps every 603975816ns
timer #0 at f080a000, irq=16
sched_clock: 64 bits at 333MHz, resolution 3ns, wraps every 4398046511103ns
clocksource: arm_global_timer: mask: 0xffffffffffffffff max_cycles: 0x4ce07af025,
max_idle_ns: 440795209040 ns
Switching to timer-based delay loop, resolution 3ns
Console: colour dummy device 80x30
Calibrating delay loop (skipped), value calculated using timer frequency.. 666.66
BogoMIPS (lpj=3333333)
pid_max: default: 32768 minimum: 301
Mount-cache hash table entries: 2048 (order: 1, 8192 bytes)
Mountpoint-cache hash table entries: 2048 (order: 1, 8192 bytes)
CPU: Testing write buffer coherency: ok
CPU0: thread -1, cpu 0, socket 0, mpidr 80000000
Setting up static identity map for 0x100000 - 0x100060
Hierarchical SRCU implementation.
smp: Bringing up secondary CPUs ...
CPU1: thread -1, cpu 1, socket 0, mpidr 80000001
smp: Brought up 1 node, 2 CPUs
SMP: Total of 2 processors activated (1333.33 BogoMIPS).
CPU: All CPU(s) started in SVC mode.
devtmpfs: initialized
VFP support v0.3: implementor 41 architecture 3 part 30 variant 9 rev 4
clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns:
19112604462750000 ns
futex hash table entries: 512 (order: 3, 32768 bytes)
pinctrl core: initialized pinctrl subsystem
NET: Registered protocol family 16
DMA: preallocated 256 KiB pool for atomic coherent allocations
cpuidle: using governor menu
hw-breakpoint: found 5 (+1 reserved) breakpoint and 1 watchpoint registers.
hw-breakpoint: maximum watchpoint size is 4 bytes.
zynq-ocm f800c000.ocmc: ZYNQ OCM pool: 256 KiB @ 0xf0880000
zynq-pinctrl 700.pinctrl: zynq pinctrl initialized
e0000000.serial: ttyPS0 at MMIO 0xe0000000 (irq = 36, base_baud = 6249999) is a
xuartps
console [ttyPS0] enabled
vgaarb: loaded
SCSI subsystem initialized
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb
media: Linux media interface: v0.10
```

```
Linux video capture interface: v2.00
pps_core: LinuxPPS API ver. 1 registered
pps_core: Software ver. 5.3.6 - Copyright 2005-2007 Rodolfo Giometti
<giometti@linux.it>
PTP clock support registered
EDAC MC: Ver: 3.0.0
FPGA manager framework
fpga-region fpga-full: FPGA Region probed
Advanced Linux Sound Architecture Driver Initialized.
clocksource: Switched to clocksource arm_global_timer
NET: Registered protocol family 2
TCP established hash table entries: 8192 (order: 3, 32768 bytes)
TCP bind hash table entries: 8192 (order: 4, 65536 bytes)
TCP: Hash tables configured (established 8192 bind 8192)
UDP hash table entries: 512 (order: 2, 16384 bytes)
UDP-Lite hash table entries: 512 (order: 2, 16384 bytes)
NET: Registered protocol family 1
RPC: Registered named UNIX socket transport module.
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
RPC: Registered tcp NFSv4.1 backchannel transport module.
hw perfevents: no interrupt-affinity property for /pmu@f8891000, guessing.
hw perfevents: enabled with armv7_cortex_a9 PMU driver, 7 counters available
workingset: timestamp_bits=30 max_order=18 bucket_order=0
jffs2: version 2.2. (NAND) (SUMMARY)  ©2001-2006 Red Hat, Inc.
bounce: pool size: 64 pages
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
io scheduler mq-deadline registered
io scheduler kyber registered
dma-pl330 f8003000.dmac: Loaded driver for PL330 DMAC-241330
dma-pl330 f8003000.dmac:        DBUFF-128x8bytes Num_Chans-8 Num_Peri-4
Num_Events-16
```

[Trenz board version: TE0715-04-30-1I3.]

[Final O/S version is not yet decide due to the lack of final PUT/Trenz boards.]

5.3. Custom Made Applications

5.3.1. Application: ad9361-config.run [TX + RX]

[`ad9361-config.run`] application is used to configure AD9361 and the internal IP Core. The program is launched automatically at the system startup. To an extent it might be re-launched during the normal operational state of the system, although such an on-the-fly-re-configuration **is strongly discouraged**, as it may result in a non-optimal system state (e.g. not every AD9361 and IP Core internals could be properly configured).

The configuration of [`ad9361-config.run`] application is kept in [`/etc/radio/current/ad9361-config.gflags`] file. Note, however, that on the TX side, the location of the configuration file might be changed due to the availability of the pre-boot/post-boot configuration update mechanism. This mechanism is realized by an appropriate software and it is transparent from the point of view of [`ad9361-config.run`] application.

The example of the configuration file mentioned in this section is shown below.

```
# Default configuration of AD9361

# Note that config lines cannot have comments!

### Cubesat specific configuration [BEGIN]

# Without this option nothing in this section
# is taken into account during configuration
--conf_cubesat

# Quadrature tracking
--quad_track=ON

# ENSM mode
# Only TX and RX are support.
# Anything else will lead to problems
# Values: TX, RX
--ensm_mode=TX

### Cubesat specific configuration [END]


### FPGA TRX config [BEGIN]

# Turn ON/OFF FPGA TRX (TDD)
--fpgatrx_enable=ON

# Select code rate
# Values: 0, 1, 2, 3, 4, 5, 6
--code_rate=0

# TX Data source
# Values: 0, 1, 2
--fpgatrx_tx_data_src=0

# Length of frequency offset estimation preamble
# Values: 0, 1, 2
--fpgatrx_frequency_offset_estimation_preamble_length=2

# Waiting time (in ms) before configuring / enabling FPGA TRX module
--fpgatrx_enable_wait_time_ms=950

# Wait time (in ms) after resetting the outer FPGA TRX IP CORE
# We wait only if --fpgatrx_outer_ipcore_reset is present, i.e.,
# it is not uncommented.
--fpgatrx_wait_after_outer_ipcore_reset_ms=45

# Whether we do or do not reset the outer FPGA TRX IP CORE.
# Comment if you want to disable resetting.
--fpgatrx_outer_ipcore_reset


### FPGA TRX config [END]

# Direction
--direction=TRX

# FIR filter configuration
--fir_filter_file=/etc/radio/filters/cubesat-filter-v0004-30dot72.ftr
--fir_filter=ON


# Extra register content
# Format used is: reg1 << val1; reg2 << val2
# NO QUOTES AROUND REGISTERS!
# --extra_registers_content=0x035 << 0x0B


# Carrier frequency in GHz
# TX/RX filters on the small PUT radio boards have range 2120 -- 2170 [MHz]
--c_frq=2.145
```

```
# Bandwidth in MHz
#--bandwidth=15
--bandwidth=28

# Sampling rate in MSPS (mega samples per second)
#
#--sampling_rate=7.68
--sampling_rate=40.816326

# TX power gain in dB
--tx_power_gain=-25


# Logging capability
--log

# Disable ADI digital interface FIR tune
# (tuning must be disabled on picozed/ADRV1CRR-FMC)
--disable_digital_interface_tune_fir

# To simulate the bahaviour without setting any AD9361 config
# Uncomment the following line:
--dry-run
```

6. Receiver implementation – “software oriented” version

6.1. GnuRadio platform

As explained in chapter 1.2.2, the baseband processing part of the receiver is implemented using GNURadio SDR software platform. It is a free and open-source software development toolkit that provides signal processing blocks to implement software radios. It can be used with readily-available low-cost external RF hardware to create software-defined radios, or without hardware in a simulation-like environment. It is widely used in research, industry, academia, government, and hobbyist environments to support both wireless communications research and real-world radio systems.
Figure 6.1 shows an example flowgraph within the GNURadio Companion visual editor:

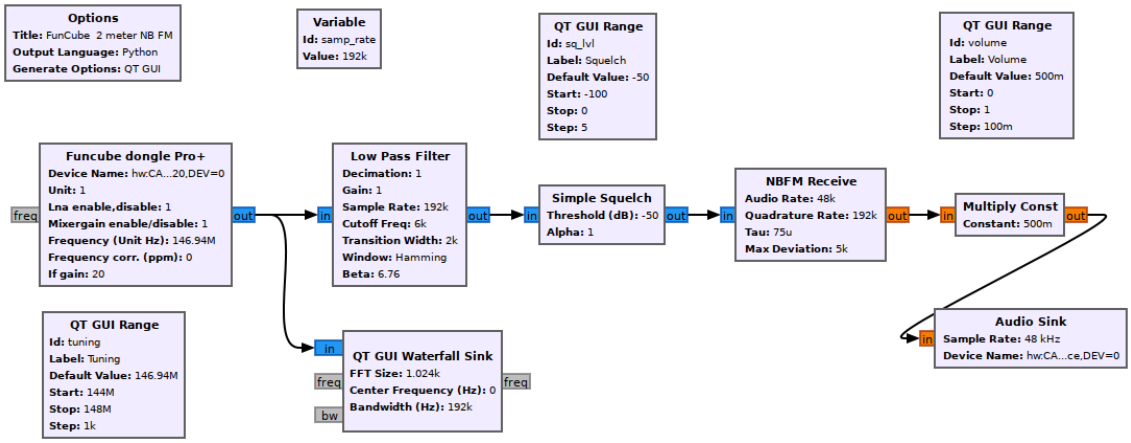

Fig. 6.1 Example flowgraph within the GNURadio Companion visual editor

GNURadio is a framework that enables users to design, simulate, and deploy highly capable real-world radio systems. It is a highly modular, "flowgraph"-oriented framework that comes with a comprehensive library of processing blocks that can be readily combined to make complex signal processing applications. GNURadio has been used for a huge array of real-world radio applications, including audio processing, mobile communications, tracking satellites, radar systems, GSM networks, Digital Radio Mondiale, and much more - all in computer software. It is, by itself, not a solution to talk to any specific hardware. Nor does it provide out-of-the-box applications for specific radio communications standards (e.g., 802.11, ZigBee, LTE, etc.,), but it can be (and has been) used to develop implementations of basically any band-limited communication standard.
If specific processing blocks are not available in the libraries or out-of-tree modules, they can be easily implemented by the interested user, which is an important feature of GNURadio.

To optimize performance of the software version Volk extension may be used (Vector-Optimized Library of Kernels). It speeds up the execution of signal processing blocks in GNURadio. It is a collection of low-level C++ routines that are optimized for vectorized execution on modern CPUs.

6.2. Receiver architecture

The “software oriented” version of the receiver includes two main components: the hardware RF front-end and the software running on a PC.
The developed receiver uses the USRP B210 from Ettus Research as the RF front-end responsible for receiving and digitizing the signals from the antenna system. It is connected via USB 3.0 interface to the PC. The base-band signal samples are processed using the GNURadio platform and the DSP blocks developed specifically for the CBSR system, as described in the following sections.
The “software oriented” receiver processes the signals “off-line”, i.e. the base-band signal samples are first saved in the files and next processed in a non-real-time manner on the PC. The decoded data stream is available with some delay (again, stored in the files), depending on the PC performance.
The USRP B210 may be replaced with another SDR hardware, however the following requirements must be met:

1. sampling rate – min. 30.72 Msps
2. frequency range – min. 6 GHz
3. GNURadio compatibility

6.3. Custom-made processing blocks

6.3.1. Receiver front-end

The receiver front-end is designed to be simple to avoid any processing on live data, which might introduce delays or errors. The front-end consists of six blocks: options block, UHD USRP source block, file sink block, and parameter blocks for center frequency, channel gain value, and sampling rate (see Fig. 6.2).
The parameter blocks are used to set the correct values for the variables used. Sampling rate can be one of the following values (MHz): 1.536, 1.92, 7.68, 15.36, 30.72. It is important to ensure that the sample rate is set correctly to avoid any issues with the collected data. Using an incorrect sample rate can result in aliasing or loss of information. Channel gain can be adjusted depending on the conditions. There should be no reason to modify center frequency, but it is implemented for completeness.
The UHD USRP source block is used to connect to the physical device and collect the data. It is important to ensure that the device is configured correctly, and that the connection is stable to avoid any data loss. The center frequency for the receiver is set to 5.84GHz, and the default gain value is set to 60dB. These values can be adjusted if needed.
The file sink block is used to store the collected samples. The program runs without a GUI to avoid any overflows while collecting data. The collected samples can be used for further processing or analysis offline, by reading the file and operating upon it.
To use the receiver, set the sample rate variable to one of the available options and run the program. The UHD USRP source block will connect to the physical device, and the file sink block will store the collected samples. It is important to ensure that the sample rate is appropriate for the intended use case. Using a higher sample rate will result in larger files and longer processing times.
All processing and analysis are performed offline, using the collected samples, to avoid any interference with collecting of the data.

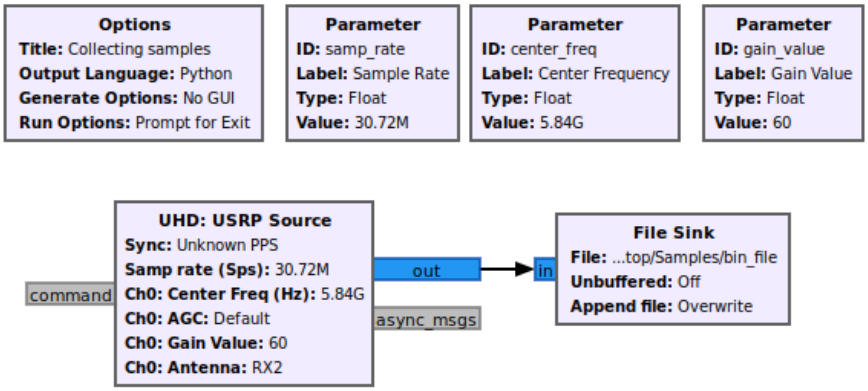

Fig. 6.2 Receiver front-end

To run the schematic we can either open GNURadio, modify values and run. It is also possible to automate the whole workflow by using a script that calls for the Python file with parameter values to be used.

6.3.2. Matched Filtering

In GNURadio, we can implement matched filtering using a series of signal processing blocks to filter the input signal and detect the presence of the known signal. Since all processing is done offline, the data is fetched from a file using File Source Block. The Throttle Block is used to control the rate at which the input signal is processed by the subsequent blocks, ensuring that the processing is done at a consistent and manageable rate. To filter the input signal and amplify the desired signal while removing unwanted noise, we use a series of Decimating FIR Filters. In this case, we use two filters with different numbers of taps: the first filter has a decimation of 1 and 17 taps, while the second filter has a decimation of 1 and 214 taps. After filtering, the Complex to Mag Phase Block is used to convert the complex signal to its magnitude and phase components. The resulting magnitude is then summed up with itself delayed by 256 samples for correlation. The frame_sync_ff block is responsible for matched filtering process. By using correlation magnitude and a set threshold we are able to get signal itself, frequency preamble and SOF (see Fig. 6.3).

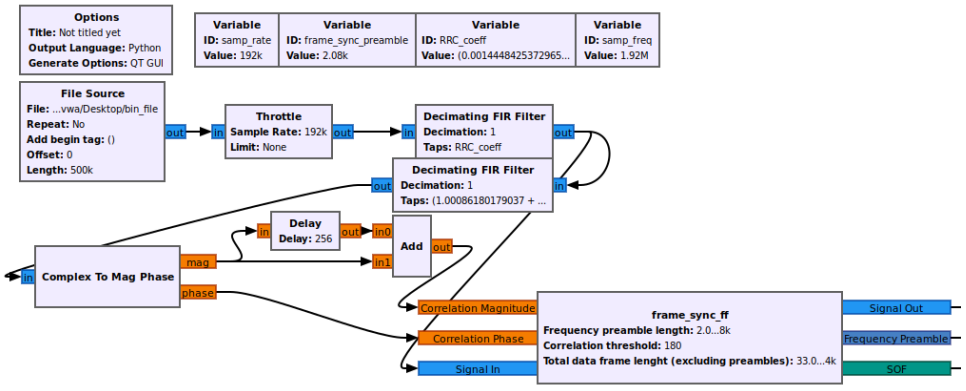

Fig. 6.3 Matched filtering

6.3.3. Time synchronization

The time synchronization part is responsible for detection and marking of the start of data part in a radio frame, as well as for initial correction of the timing mismatch. The overall procedure follows the same procedure as for the hardware receiver, as described in Section **Błąd! Nie można odnaleźć źródła odwołania.**. Fig. 6.4 presents the time synchronization part of the software receiver. The input samples, coming from the matched filtering RRC block, are first filtered with a FIR filter with coefficients matching the Zadoff-Chu sequence used in the *T_AMB* part of the preamble, with the output of the filter fed into a magnitude and phase calculation block. The magnitude is used to construct a correlation metric used for frame detection, whereas the phase is used for initial coarse phase offset estimation. Furthermore, the magnitude squared of the correlation metric is used to perform timing offset estimation according to the Center of Gravity (CoG) method described in [5].

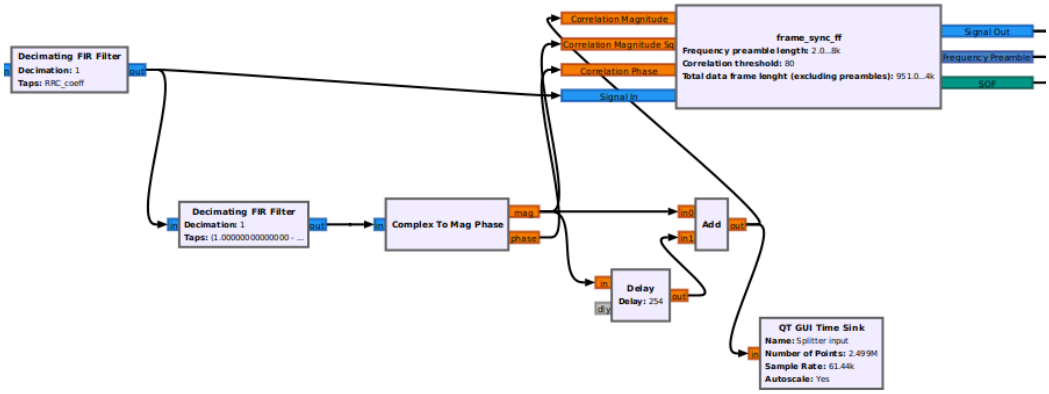

Fig. 6.4 Signal detection and time synchronization

The core of the time synchronization subsystem is the *frame_sync_ff* function implementing a FSM responsible for marking of the start of the data part in a radio frame using a dedicated *SOF* tag and performing initial timing offset correction using a Farrow interpolating filter. Furthermore, the FDM also extracts and correlates the *F_AMB* preamble with a reference sequence to provide the input signal to the coarse frequency offset estimation part. The inputs, outputs and parameters of this block are described in Table 6.1, while the example of tagged output signal (*Signal Out*)is shown in Fig. 6.5.

Table 6.1 Selected Inputs and outputs of *frame_sync_ff*

Name	Type	Description
Correlation Magnitude	Input	Received signal correlation metric used for signal detection and time synchronization
Correlation Magnitude Sq	Input	Received signal correlation metric used (after being squared) for initial timing offset estimation using the CoG method
Correlation Phase	Input	Received signal phase metric used for initial phase offset estimation
Signal In	Input	Input complex signal, to be tagged with the found start of radio frame

Frequency preamble length	Parameter	The length of *F_AMB* preamble used for frequency offset estimation (including the extension samples)
Correlation threshold	Parameter	Minimum correlation magnitude value to consider a start of a radio frame to be found.
Total data frame length	Parameter	Expected minimum number of samples of the data part of the radio frame to determine the point in which a search for a new frame will start.
Signal Out	Output	Samples of the radio frame, including the indication of the start of data part of the radio frame in form of *SOF* tag.
Frequency Preamble	Output	Samples of correlation of *F_AMB* preamble with reference sequence used for coarse frequency synchronization
SOF	Output	Signal indicating the detected start of data part of the radio frame – nonzero value for a sample where the start is identified.

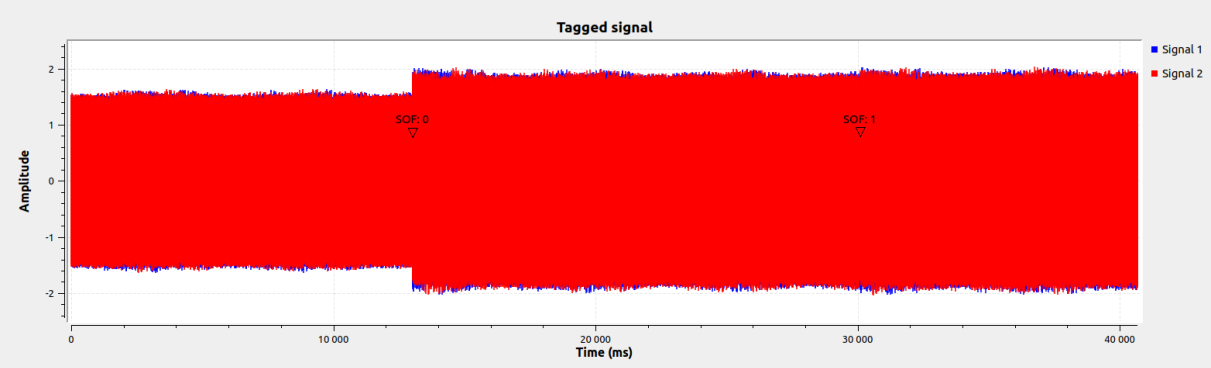

Fig. 6.5 Example of tagged output of *frame_sync_ff* block
(*SOF* indicates start of data part of a radio frame)

6.3.4. Coarse Frequency offset estimation

The result of correlation of *F_AMB* preamble with the reference sequence is fed to the coarse frequency offset estimation part to find the initial CFO caused by Doppler effect and clock mismatch between the transmitter and the receiver, that still remains after the initial correction performed based on the satellite trajectory estimation. The structure of the coarse frequency offset estimation part is shown in Fig. 6.6 and comprises blocks performing calculation of the FFT magnitude and a *CFO_estimator_ff* custom block responsible for finding the maximum bin of Magnitude FFT and calculation of the CFO with the use of Fractional Fourier Coefficients, as described in [4]. The output of the *CFO_estimator_ff* block is then fed to the custom midamble extraction and processing block to perform CFO correction.

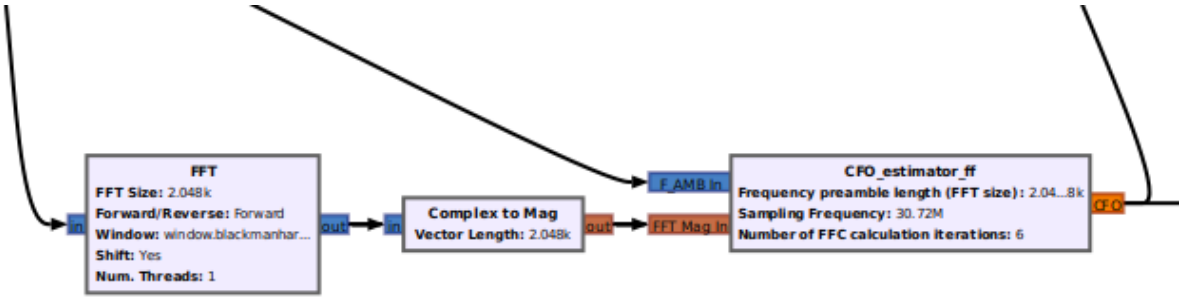

Fig. 6.6 Frequency offset estimation part of the “software receiver”

The inputs, outputs and parameters of the *CFO_estimator_ff* block are described in Table 6.2.

Table 6.2 Selected Inputs and outputs of *CFO_estimator_ff*

Name	Type	Description
F_AMB in	Input	Signal resulting from correlation of *F_AMB* with reference sequence (input to the coarse frequency synchronization part)
FFT Mag in	Input	Magnitude FFT of the correlation of *F_AMB* with reference sequence
Frequency preamble length	Parameter	The length of *F_AMB* preamble used for frequency offset estimation (excluding extension samples – FFT size)
Sampling Frequency	Parameter	Sampling frequency of the received signal
Number of FFC calculcation iterations	Parameter	Number of iterations used in calculation of CFO using the FFC method
CFO	Output	Found CFO estimate (relative to the sampling frequency)

6.3.5. Midamble processing and frame disassembly part

The main part of the synchronization functions is performed in the midamble processing and frame disassembly part, show in Fig. 6.6. It comprises three custom functional blocks: *midamble_processing_ff*, *convertToQPSK_ff* and *subframes_to_files_ff*. The role of *midamble_processing_ff* is to perform coding rate (CRI) detection, frame disassembly with extraction of midambles and fine timing, phase and frequency offset estimation and correction. This block processes the input signal tagged with SOF block, (coming from the *frame_sync_ff* output) using also the CFO estimated in the coarse frequency estimation part. The inputs, outputs and parameters of the *midamble_processing_ff* block are described in Table 6.3, whereas the detailed functionality of this block is given in the following subsections.

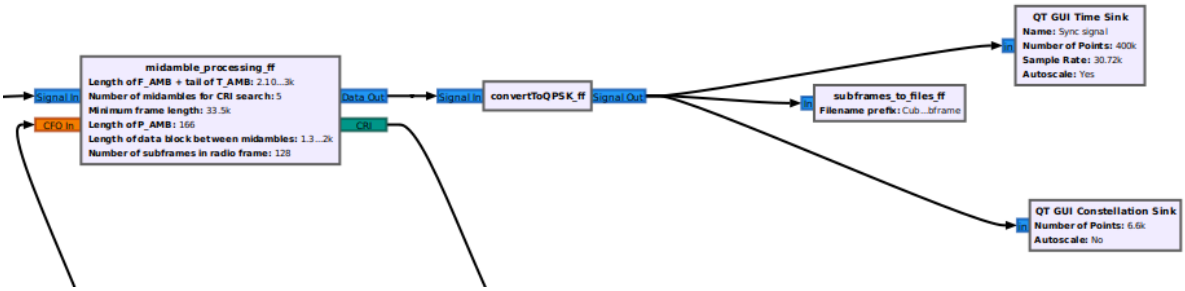

Fig. 6.6 Midamble processing and frame disassembly part of the "software receiver"

Table 6.3 Selected Inputs and outputs of *midamble_processing_ff*

Name	Type	Description
Signal in	Input	Samples of the radio frame, including the indication of the start of data part of the radio frame in form of *SOF* tag (output of *frame_sync_ff* block)
CFO in	Input	Initial (coarse) estimate of CFO from *CFO_estimator_ff* block
Length of F_AMB + tail of T_AMB	Parameter	The expected number of samples used as a tail of T_AMB and *F_AMB* sequence, to be accounted for in CFO correction process.
Number of midambles for CRI search	Parameter	Number of midambles (*P_AMBs*) used for averaging in CRI search procedure
Minimum frame length	Parameter	Minimum expected number of samples of a radio frame
Length of P_AMB	Parameter	Number of samples of the *P_AMB* midamble sequence.
Length of data blocks between midambles	Parameter	Size of the data block between two subsequent midambles (*P_AMBs*)
Number of subframes in radio frame	Parameter	Expected number of subframes (codewords) contained in a single radio frame
Data Out	Output	Output signal (data only) with tags indicating the start of each subframe (codeword) and the identified CRI
CRI	Output	Output indicating the estimated CRI values

The output of *midamble_processing_ff* block is a complex signal comprising only the data samples of the received radio frame, with tags indicating start of each separate subframe (codeword) and the identified code rate (CRI). While these signal consists of OQPSK symbols, it is fed to the *convertToQPSK_ff* block, where it is downsampled twice, with the symbols converted from OQPSK form to standard QPSK constellation. Finally, the output tagged QPSK signal, with example presented in Fig. 6.7 and the constellation shown in Fig. 6.8, respectively, is fed to the *subframes_to_files_ff* block, where each subframe (codeword) complex samples are stored in individual data files used as an interface to the demodulation and decoding part of the receiver. This last block makes use of a single parameter which is a specification of the common prefix used for files to store the data subframes (files are saved with format *prefix* + *subframe_number*).

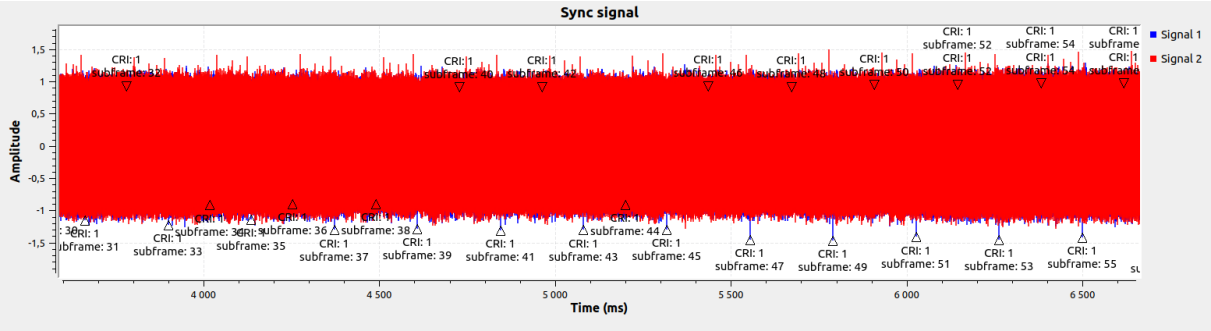

Fig. 6.7 Example of the tagged QPSK data sequence being input to the *subframes_to_files_ff* block.

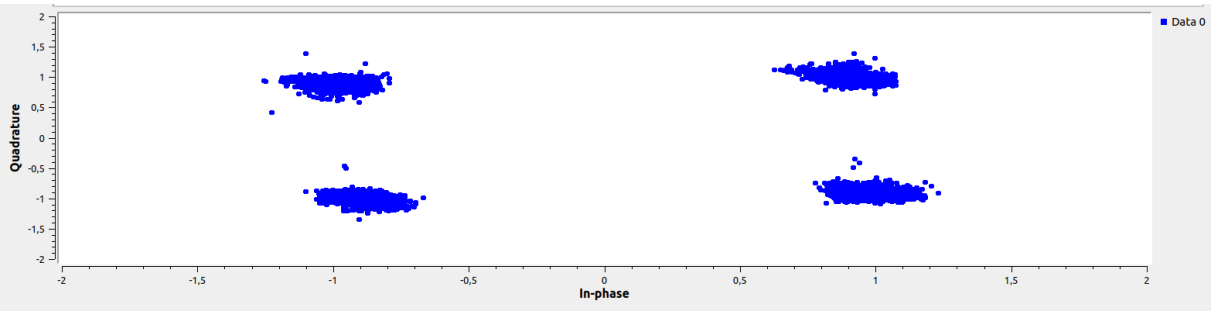

Fig. 6.8 Example of the received QPSK data constellation.

6.3.5.1. Coding rate evaluation

There are 8 possible sequences to be used as P_AMB midamble, with 7 representing the differend coding rate used, and the additional one used to indicated end of transmission (EoT). All these sequences result from a cyclic shift of the same base Zadoff-Chu sequence, with different shift values applied. Therefore, in the *midamble_processing_ff* block the received midambles are correlated with all 8 possible versions of the reference midamble, and the version (shift) resulting in the highest correlation metric is assumed to be the used one – representing the applied coding rate. In order to mitigate the impact of noise on this estimation, the results of correlation of N consecutive midambles are averaged, where N is the value specified using *Number of midambles for CRI search* parameter of the *midamble_processing_ff* block. The identified index of coding rate (CRI) is then signaled using the dedicated output of this block, as well as added to the output symbols sequence in form of tags (CRI tag in Fig. 6.7).

6.3.5.2. Radio frame disassembly

Radio frame disassembly is performed by the *midamble_processing_ff* block based on the detection of the *SOF* tag, inserted to the input sequence in *frame_sync_ff* block and indicating the start of the first midamble (P_AMB) in the radio frame. The sample corresponding to the *SOF* tag is considered to be the first midamble sample. Then a counter is started that counts the consecutive samples and categorizes them to one of the following sets:

- Midamble (P_AMB) samples – these are the samples where the value of counter is less than the parameter *Length of P_AMB*.
- Data samples – those where the counter value is greater than or equal *Length of P_AMB*.

The counter is reset to 0 every (Length of P_AMB + Length of data blocks between midambles) samples. This procedure continues for the duration of the whole radio frame, that depends on the expected length of a data subframe (determined based on CRI value – found as described in 6.3.5.1) and number of data subframes (codewords) contained in a radio frame (Number of subframes in radio frame parameter).
Only the data samples are later output from the frame_sync_ff block, with the start of consecutive subframes (codewords) marked using subframe tag.

6.3.5.3. Fine timing, phase and frequency offset estimation

The final correction of the received data is performed in the midamble_processing_ff block. It consists of:

- Fine timing error tracking – performed using the CoG method [5], with the results of midamble correlation used to find the remaining timing error. The aim of this procedure is to correct the eventual shift in timing resulting from mismatch of the oscillators at the transmitter and the receiver. The timing error is then corrected using Farrow interpolating filter.
- Fine phase and frequency offset estimation – found as the gradient of the change of phase offset of subsequent midambles (P_AMBs). For the purpose of its estimation the results of correlation of subsequent midambles are used, with the residual remaining CFO estimated based on averaging of results obtained for consecutive midambles pairs. This estimated phase and frequency offset is then used along with the coarse CFO estimate from CFO_estimator_ff block to correct the phase of the data symbols forming the output of the midamble_processing_ff block. The exact procedure is described in [4].

6.3.6. Demodulation

After successful phase and frequency offset estimation, the data are passed to the demodulation block. The role of this block is to calculate the LLR (log-likelihood ratio) values that subsequently fed the decoding block. The demodulation is performed according to the formula:

$$log\left(\frac{P(b=0|r)}{P(b=1|r)}\right) = \frac{d_1^2 - d_0^2}{N_0}$$

where $d_i = |r_k - s_i|$, r_k represents the received signal and s_i represents the constellation points.

6.3.7. Decoding

The decoding process consists of three separate stages. The first stage is a Rate De-Matching, the second stage is iterative decoding and the third step is CRC verification. Each of the stages is realized with a separate block.
The decoding process starts with rate de-matching. Its role is to match the number of transmitted bits to the size of the unpunctured codeword. It is caused by the fact that in the system a set of different CRI can be used, and each CRI carries a different number of coded bits.
The information generated by the Rate De-Matching is fed to the iterative decoder. Within the system, the LTE-compliant turbo code is used with the maximal number of 8 iterations. For a single SISO decoder, a Max-Log-MAP decoding algorithm was assumed.

After the decoding process frame is subjected to CRC verification. Its role is to decide whether the decoding process was successful. Moreover, CRC verification is performed after each iteration.

6.3.8. Data packing

Data packing is the last block in the software receiver chain. Its role is to prepare a file (or set of files) that contains all the information transmitted via the transmission link. However, decoded data is not the only information stored in the files, the additional information is the following:

- CRI index – contains information on what CRI was used to transmit the data,
- The number of iterations – this information indicates how many decoding iterations were used in the decoding process to decode the obtained codeword. It needs to be mentioned that in the case of a correctly decoded codeword, the number of iterations used can be lower or equal to the maximal number of iterations allowed (8 is assumed as the maximal number of iterations). In the case of erroneous decoding, this field must contain the maximal number of decoding iterations.
- Information if codeword was decoded correctly – this field informs if the data stored in the file are valid (information obtained via CRC check); if not data retransmission will be requested by higher protocol layers.

Bibliography

[1] CS.S1.Gen System specification
[2] Zynq-7000 All Programmable SoC Overview (DS190)
[3] CS.S2.Gen Transmitter module
[4] CS.S5.Gen Transmission and reception
[5] A. Gesell, J. B. Huber, B. Lankl, G. Sebald, “Data-Aided Symbol Timing Estimation for Linear Modulation”, *AEU - International Journal of Electronics and Communications*, Volume 56, Issue 5, 2002, Pages 303-311, ISSN 1434-8411