Semester Project

Sensing and Networking Laboratory

Enabling Joint Communication and Sensing Through RFSoC-Controlled Phased Array Waveforms

Student:
Giovanni Ranieri (326870)

Supervisor:
Samah Hussein
Professor:
Haitham Al Hassanieh

EPFL, Spring 2026

Ecole Polytechnique Fédérale de Lausanne

Contents

1 Introduction

A phased array is a collection of antennas that can be used on the transmit or receive side in both communication and radar systems. We typically refer to the antennas that make up an array as elements. These array elements are most often omnidirectional antennas, equally spaced in either a line or across two dimensions. Phased arrays do not necessarily have the computational power to process digitally samples. Instead, they are connected to more advanced radio frequency (RF) boards for sampling, down converting, etc. A typical signal processing technique used alongside phased arrays is **Beamforming** (BF). It uses the antenna array to create a spatial filter. BF can be used to increase Signal-to-Noise Ratio (SNR) of desired signals, null out interferer or even shape beam patterns. As part of BF, we use weights applied to each element, either digitally or in analog circuitry. The most important concept in BF is the *steering vector*. Consider a uniform linear array (ULA) of M elements spaced by d. The phase difference between adjacent elements for a signal arriving at angle θ from broadside is

$$\Delta\phi = \frac{2\pi d}{\lambda}\sin\theta, \tag{1}$$

where λ is the wavelength. Starting from the edge, we can write the signal received on all elements with

$$\boldsymbol{x}(t) = s(t)\boldsymbol{a}(t) + \boldsymbol{w}(t), \tag{2}$$

where $\boldsymbol{w}(t)$ is the noise signal, $s(t)$ the data signal, and

$$\boldsymbol{a}(t) = \begin{bmatrix} 1 \\ e^{j\Delta\phi} \\ \vdots \\ e^{(M-1)j\Delta\phi} \end{bmatrix} \tag{3}$$

is the steering vector. The output after BF is the inner product

$$y_k(t) = \boldsymbol{w}_k^H \boldsymbol{x}(t) = s(t)w_k^H \boldsymbol{a}(t) + w_k^H \boldsymbol{w}(t), \tag{4}$$

where $\boldsymbol{w}_k \in \mathbb{C}^M$ is a weight vector. When $\boldsymbol{w}_k$ is designed to steer toward angle θ_k, the weights are essentially a conjugated steering vector for that direction, so $\boldsymbol{w}_k^H \boldsymbol{a}(\theta_k)$ is maximized. Switching between indexes is critical because finding the best index k is actually the optimization problem. We do not want to waste time switching configurations of beam patterns. The Sivers EVK06005 provides a BF interface for fast-switching between $\boldsymbol{w}_k$. By combining it with the ZCU208 Evaluation Board, we have a very-optimized setup for playing with BF techniques. The key feature we want is to be able to switch between beam patterns between OFDM symbols.

In fact, the optimal $\boldsymbol{w}_k$ can be different per symbol. The optimal beam is the one that maximizes the channel gain. But we have to find that beam first, which requires testing multiple configurations. If you can only switch beams between frames (many symbols), we waste a lot of time on non-optimal beams. If you can switch per symbol, we can interleave a few probing symbols (to test other beams) with many data symbols (on the best known beam), minimizing throughput loss.

One key advantage that we can use is in Joint Communication and Sensing (JCAS). It is a key enabler of future 6G wireless systems: the same RF hardware and waveform are used both to transmit data and to sense the environment. In this project we take advantage of a 6G platform called HELIX [Ruiz et al., 2025] to experiment dual-functional radar-communications over the ZCU208 and the EVK06005. More concretely, we want to experiment the approach described in a paper called "Orthogonal Coexistence of Overlapped Radar and Communication Waveforms" [Memisoglu, Sahin, and Arslan, 2022]. They split OFDM bins between data and

sensing bins. Using basic DFT properties of periodic signals, we know that a signal that repeats itself P times over L samples will have its non-zero DFT coefficients at bins $k = 0, P/L, 2P/L$, etc. Because OFDM modulation starts in frequency domain, if we manage to place data bins only at bins where the DFT of the sensing waveform is 0, then after the IFFT, adding this sensing waveform that repeat itself in the time-domain will result in a waveform carrying sensing and data bins that do not overlap in the frequency domain (we can say the sensing waveform is not leaking onto the data waveform).

2 Hardware/Software Setup

We introduce in this section the hardware used and important software specifications. All manuals and documents are summarized in Table [6], giving next users a good start on documents to read.

2.1 ZCU208 Evaluation Board

The ZCU208 is an evaluation board from AMD/Xilinx featuring the Zynq UltraScale+ MPSoC (XCZU48DR-2FSVG1517). The board comes with two different daughter boards for (i) loopback evaluations with ADCs/DACs (HW-XM655), and (ii) a clocking generation system. The board features a 10G Ethernet Quad zSFP/zSFP+ connector, 8 TX (10 Gsps) and 8 RX (5 Gsps) antennas, one quad core Cortex A53 processor, one dual core real-time processor, and a FMC+ expansion port connector.

(a) The ZCU208 Evaluation Board without the XM655 Balun daughter card mounted for ADCs/DACs.

(b) The Sivers EVK06005 Phased Array, mounted with a Beamforming RF Module.

Figure 1: Hardware modules used in this project. One of the first goal is to physically connect them such that (i) I/Q samples can be sent and received, and (ii) BF/SPI signals can be interfaced.

We summarize some important information for the project (from the user manual UG1410):

- The FMC+ connector (J28) has the High Pin Count (HSPC) connectivity and follows the VITA 57.4 standard. **It is the only exit point for extracting signals from the board**.
- The block diagram shows the connectivity of the Zynq, which Banks are connected to which systems. For example, Bank 64 and 67 drive the LA bus of the FMC+ (general purpose I/O differential pairs).
- Different boot modes are available and explained at page 19.
- I/O voltage rails are shown in Table 6. These levels should be respected when driven out of the Zynq, such as loads and current.
- The Zynq features several built-in master and slave interfaces for SPI, I2C and UART with the help of EMIO (Extended Multiplexed I/O).
- Multiple clocks are running through out the board. Some of them are fixed, others are programmable. Table 17 summarizes that, while Table 18 shows which one are driven out/in of the fabric. The most important ASIC for clock generation is the I2C-programmable SI570.
- I2C lines have switches (MUX) to handle multiple signals through out the board. The full connectivity is shown in Figure 6.

2.2 Sivers EVK06005

The EVK06005 is a complete radio front-end designed with 16 receive and 16 transmit beamforming elements, working between 57 and 71 GHz. It performs zero-IF down-conversion and outputs analog I/Q samples over MCX connectors. Micro-USB and classical GPIOs are provided to interact with it. The first one is a user-friendly connection, but for deployment and real experimentation, the EVK06005 has to be connected to a motherboard that control the data streams. A SPI interface configures the phased array at register level, while a Beamforming (BF) interface gives a very-fast way of updating the phase shifters (so-called weights).

The EVK06005 takes advantage of **beambooks**. The phased array does not let you change phase shifters one by one separately on the fly. For TX and RX paths we have an AWV (Antenna Weight Vector) table of size 64x16 Bytes. Each table has 64 different configurations of phase shifters. In addition, an AWV pointer table of size 64 is present for RX and TX respectively. These tables are used to create sequences of beam configurations, where we jump from one to the next. A bypass pointer lets you select the configuration you want directly without this table. Everything is summarized in Figure [2].

2.3 FPGA and Firmware Pipeline

This project is a good introduction to the world of FPGA/Firmware development using the Vivado and Vitis suite of AMD/Xilinx. The path from block design to a running FPGA goes through a fixed sequence of stages.

1. **Synthesis:** the tool reads the HDL and translates it into logic primitives (LUTs, flip-flops, BRAMs, DSP slices) while respecting the timing and I/O constraints in the XDC (Xilinx Design Constraints) files.
2. **Implementation:** it takes these blocks and physically realizes it on the target part. It runs opt_design, place_design, and route_design, which together decide which specific LUT, slice, and routing track each piece of logic occupies, and verify that all paths still meet timings.

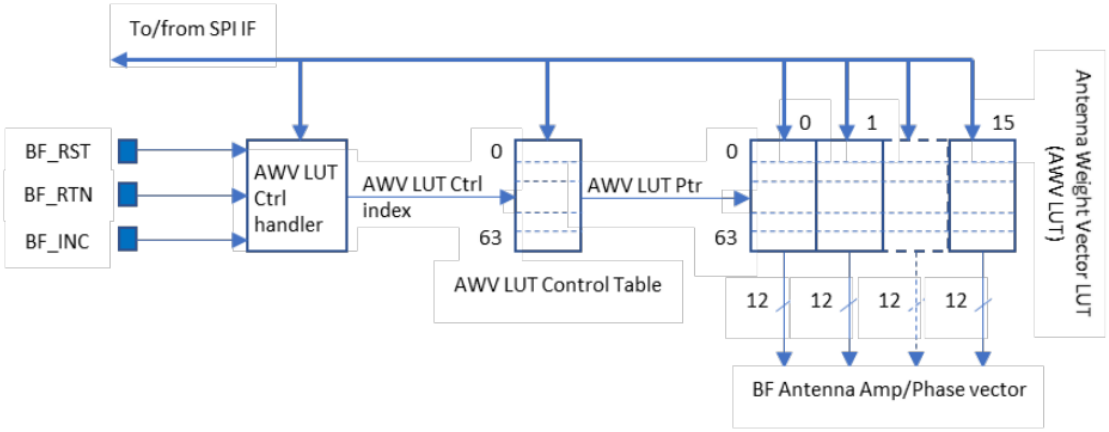


Figure 2: External BF interface for Sivers EVK06005, showing the different tables in memory.

3. **Bitstream generation:** it serializes that design into a .bit file (a binary configuration whose bits directly drive the FPGA's configuration memory cells). Loading this bitstream is what physically defines the hardware.

In our case, it does not stop here. As mentioned before, Zynq UltraScale+ features multi-purpose processors that we call the PS (Processing System), compared to the PL (Programmable Logic) which defines the FPGA fabric. The code that runs on the processors we call it the firmware (sometimes the word baremetal application can be used for non-RTOS applications). In the AMD/Xilinx development, Vitis is the application used. The first important element is the Xilinx Support Archive (XSA). The hardware specifications are stored in the XSA (which is a zip-folder containing the bitstream), containing for example the register addresses (with naming) from different IPs that can be configured via AXI4-Lite interfaces, and more.

In Vitis, that XSA is consumed to build a platform. It takes the hardware definition from the XSA, generates the drivers and headers for every IP block, picks an OS target (standalone/bare-metal, FreeRTOS, or Linux) per processor and produces the libraries and linker scripts that any software running on this hardware will link against. On Zynq UltraScale+ the platform also generates firmware components like the FSBL (First Stage Boot Loader) and PMUFW (Platform Management Unit firmware), which run before user code and bring the System-on-Chip up. On top of the platform we build an application: the actual C/C++ source, compiled and linked against the platform's BSP (Board Support Package) into an ELF binary that runs on one or more of the processor cores. At deployment, the bitstream and the ELF are typically packaged together into a BOOT.BIN file so that a single boot image programs the PL and starts the PS in one shot.

2.4 Understanding FPGA and FMC+ pinout using Vivado constraints

Physically connecting signals between the FPGA and external hardware requires information from three sources: the FPGA pinout, the board datasheet, and the Vivado constraints file.

FPGA pinout. The FPGA exposes a fixed set of pins, organized into groups called *Banks*. Each pin has specific properties such as its supported voltage level, whether it is clock-capable, and whether it belongs to a differential pair.

Board-level routing. Only a subset of FPGA pins is routed to external components on a given board. For the ZCU208, the FMC+ connector is wired to specific FPGA banks, which can be identified from UG1410. For instance, Banks 64 and 67 are routed to the FMC+ connector.

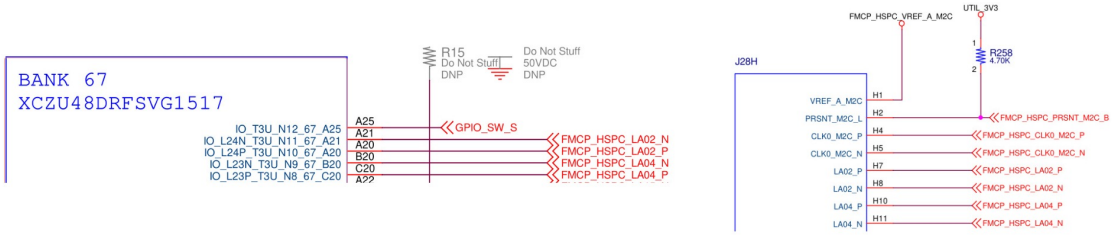


Figure 3: Schematic screens from the ZCU208 datasheet showing the routing of Bank 67 signals to the FMC+ connector (J28H). The differential pair on pins A20/A21 is routed to H7/H8 of the connector as FMCP_HSPC_LA02_P/N.

Vivado constraints (XDC). When a signal in the design must be exposed as an external I/O, the XDC file assigns it to a physical FPGA pin. The correct pin number is determined by cross-referencing the FMC+ signal of interest with the ZCU208 datasheet to find which FPGA pin it is connected to.

In summary, defining an external signal requires tracing it through all three layers: from the design, through the FMC+ connector, down to the specific FPGA pin declared in the XDC file. As a concrete example, consider exposing a general-purpose differential I/O pair through the FMC+ connector so that it can be used on another board.

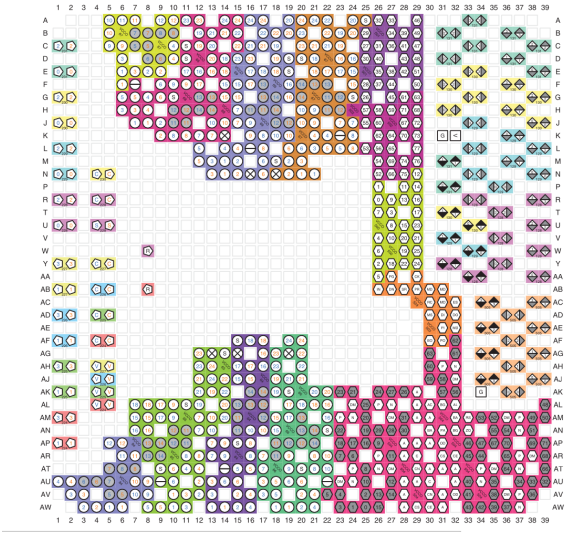
Select a signal pair from the datasheet In Bank 67, we can choose FMCP_HSPC_LA02_P (pin A20) and FMCP_HSPC_LA02_N (pin A21). These pin numbers refer to the FPGA pinout.

Trace the signals to the FMC+ connector. The ZCU208 datasheet (see Figure [3]) shows that A20 and A21 are routed to pins H7 and H8 of the FMC+ connector (J28H), respectively.

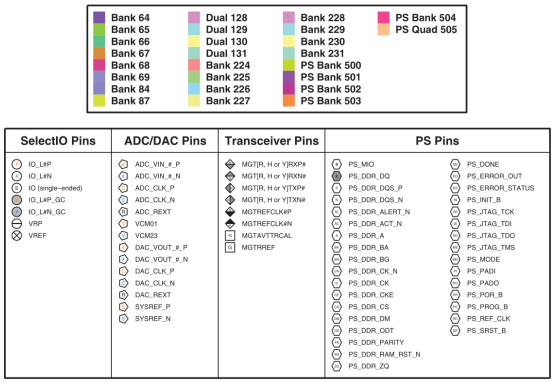
Verify the bank and pin properties. The FPGA pinout confirms that A20 and A21 belong to Bank 67 and form a differential pair, labelled `IO_L24P` and `IO_L24N`. The FPGA pinout is referenced in UG1075 at page 224, and placed here for completeness in Figure [4]. AMD/Xilinx provides a master XDC constraints file in the XTP653 documentation. Searching for pins A20 and A21 in this file reveals an IOSTANDARD of LVCMOS18, where the suffix `18` indicates a 1.8 V voltage level.

2.4.1 Mismatches in Datasheet

A mismatch has been found between the ZCU208 schematic and the UG1410, where Bank 66 was shown to be connected to the FMC+ connector, whereas the ZCU208 schematic clearly showed signals going out from Bank 64. A confirmation of the AMD forum confirmed the mismatch. In summary, Bank 64 and 67 are connected to the FMC+ connector, and not Bank 66. The second mismatch was in the XDC Master File: the PL_I2C1_SDA_LS signal (see later why I2C was important) coming out of Bank 68 was described as a POD_DCI I/O standard, which is wrong because its a DDR4 standard. We should use LVCMOS33 similarly to PL_I2C1_CLK_LS.



(a) XCZU48DR Square Pinout



(b) XCZU48DR Square Pinout description

Figure 4: FFVG1517, FSVG1517 Packages–XCZU48DR and FSRG1517 Package–XQZU48DR Pinout description for ZCU208 Evaluation Board.

3 HELIX

HELIX [Ruiz et al., 2025] is an open-source 6G real-time experimentation platform built around the ZCU208. It implements a complete 5G low and high physical layer in hardware, supports configurable functional splits (e.g., 6, 7.2) on both transmitter and receiver, and integrates seamlessly with O-RAN systems. The RF chain is wired as follows, per I/Q path:

TX: DAC → DC blocker → 1 GHz LPF → 3 dB pad → front-ends.

RX: front-ends → 1 GHz LPF → DC blocker → ADC.

3.1 Project Structure

3.1.1 Key Elements

Because this project is close to hardware, its important to understand what is actually done at that level. At RF level, two DAC tiles and two ADC tiles operate in IQ-mode, meaning the in-phase and quadrature components are processed independently. RX-side parameters are summarized in Table [1].

The reasoning behind these choices deserves a closer look. Although the ADCs can sample up to 5 Gsps, decimation by $M = 8$ is needed to discard the unnecessary high-frequency content. Decimation keeps every M-th sample of $x[n]$, producing $y[n] = x[nM]$ at a new sample rate f_s/M, where f_s is the original sampling frequency. In the frequency domain, this creates M shifted copies of the spectrum that sum together:

$$Y(f) = \frac{1}{M} \sum_{k=0}^{M-1} X\left(f - k \cdot \frac{f_s}{M}\right). \tag{5}$$

Any content of $X(f)$ outside the new Nyquist zone $\pm f_s/(2M)$ would fold back on top of the in-band signal, causing aliasing. To avoid this, a LPF with cutoff $f_s/(2M)$ is applied *before* down-sampling, forcing $X(f) \approx 0$ outside the new band so that only the $k = 0$ term survives. In

Table 1: Key RX parameters in HELIX

Step	**Explanation**
Sampling Rate	XCZU48DR has 14-bit ADCs running up to 5 Gsps, capturing frequency components up to 2.5 GHz.
Digital Mixer (NCO)	Translates real RF samples to complex baseband by multiplying the signal with a complex exponential.
Decimation	Low-Pass Filter (LPF) and down-conversion with decimation factor $M = 8$.
General clock frequency	$f_{\text{clk}} = 245.76$ MHz, chosen to relax the PL datapath.

HELIX, this stage takes the signal from 3.93216 Gsps at the ADC down to 491.52 Msps. The NCO is configured at $f_{\text{NCO}} = 400$ MHz so that the DC subcarrier, which carries information in 5G, is preserved. Finally, HELIX processes two I/Q samples per beat, which relaxes the PL datapath and yields $f_{\text{clk}} = 491.52/2 = 245.76$ MHz.

The received I/Q samples are carried over different types of buses. Most of the time **two 32-bit AXI4-Stream buses** are used, one for the real part and one for the imaginary part (see Figure [5]). A central block called the *stream manager* orchestrates this datapath. It ensures precise control of data streaming, including the start and stop of streams to maintain sample alignment. It also handles critical tasks such as clock domain crossing between the 10Gb Ethernet clock and the clock used by the processing blocks, ensuring seamless data transfer between different domains. The 10Gb Ethernet link serves as the interface between the server and the PL, with packets transmitted as UDP over Ethernet.

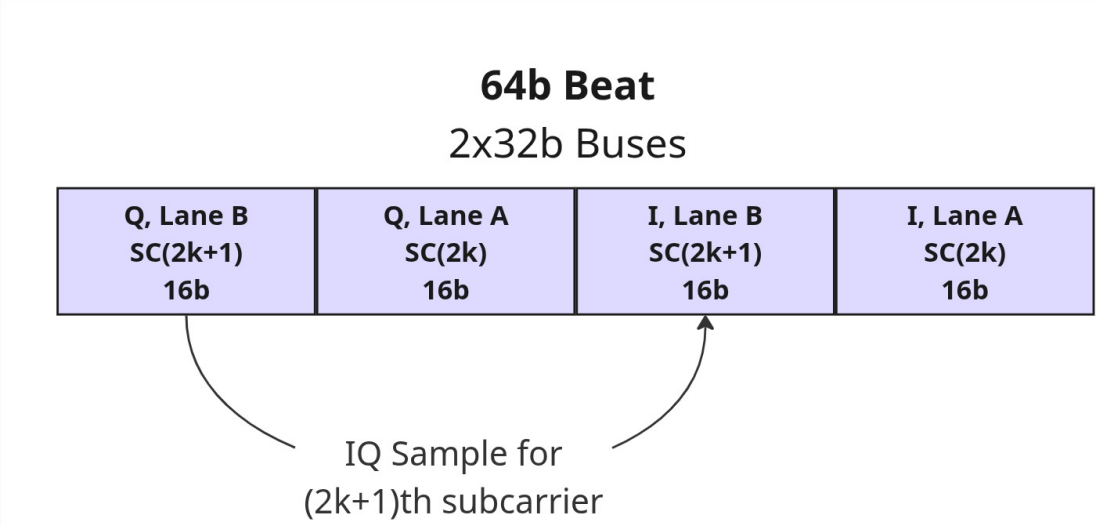


Figure 5: Structure of I/Q samples internally in HELIX. Per beat, 2 I/Q samples are processed, each one on a specific 32b wide bus.

The key parameters for OFDM are listed in Table [2]. One important consideration is that the size of the cyclic prefix (CP) is not constant, *something not taken into account in [Memisoglu, Sahin, and Arslan, 2022].*

Table 2: OFDM parameters in HELIX

Name	Description
Number of subcarriers	FFT size $N = 2048$ is fixed (see `FFT_mod_top.vhd:124`).
Number of symbols	One slot contains $N_{\text{SYM}} = 14$ symbols.
Cyclic Prefix	The CP length is not constant: the first OFDM symbol in a slot has a longer CP than the others. HELIX sets `CP1` and `CP2` to 400 and 144 samples respectively (see `FFT_mod_top.vhd:195-196`). Note the division by 2 to account for the 2 I/Q samples per beat.
Pilots positioning	Pilot positions are configurable at runtime, following position = dmrsOffset + $k \times$ dmrsSpacing. The pilot values themselves are fixed, stored in the ROM files `txBuildGrid_dmrs_V_{0..3}_ROM_AUTO_1R.vhd` and the corresponding `_ptrs_` versions.

3.1.2 Sensing

The DMRS symbol is a special symbol among the 14 symbols of a slot which carries sensing pilots. The idea is similar to the dual-functional radar-communications approach we use elsewhere, but at a smaller scale. HELIX provides registers to define the location of these bins within the DMRS symbol. It must be stated that sensing cannot be disabled at runtime, even with `DMRS_SYM_LENGTH` set to 0. Hardware changes have to be made. The slot length s_l is therefore

$$\begin{aligned} s_l &= \text{CP1} + (N_{\text{SYM}} - 1)\,\text{CP2} + N_{\text{SYM}} \times N \\ &= 400 + 13 \times 144 + 14 \times 2048 = 30{,}944, \end{aligned} \tag{6}$$

which matches software side (see radio_control.cpp).

3.1.3 Firmware and Server

On the embedded side, an application running on FreeRTOS on the Cortex-A53 handles clocking, RFDC bring-up, NCO setup, GPIO, the SD card, DMA, the UDP server and, as added by this project, the SPI, BF and JSAC drivers. On the server side, a user-space C++ API communicates with the RFSoC over three UDP sockets: a data plane, a control plane, and a metadata side-info plane. The API exposes register access, sample streaming, and the radio control class used to (re)configure the physical layer at runtime, including split selection, numerology, DMRS positions, and CP length.

3.2 Dual-Stream in HELIX

Because we had only one ZCU208, we couldn't replicate completely the HELIX setup for sensing and communication benchmarks. We decided to modify the hardware design to include a second phased array on the ZCU208. During one month, we tried to include everything needed, but (i) the lack of documentation in the original design, (ii) and the difficulty to change a verilog custom UDP stack, left us no choice but to abandon this idea.

4 SENS of Doom

4.1 Printed Circuit Board

The ZCU208 was never designed to be wired to the EVK06005. The phased array exposes its SPI and BF interfaces on two small 8-pins Molex (51021-0800) connectors. The ZCU208 exposes its FPGA-side through the FMC+ HSPC connector (following the VITA 57.4 standard), a 560-pin Samtec connector with no breakout to any pin headers needed for the phased array. Three options were available:

1. Solder thin wires directly on the FMC+ pins. We rejected this idea because the pins are too dense to be reliably accessed and the board costs 16k CHF.
2. Buy an off-the-shelf FMC+ to header breakout (so-called Mezzanine). We found no commercial board on the market matched the phased-array's 8-pin Molex pinout that match the FPGA pinout of the ZCU208.
3. Design our own breakout Printed Circuit Board (PCB), called SENS of Doom. *We decided to go with this third option.*

The features of SENS of Doom, shown in Figure [6] are:

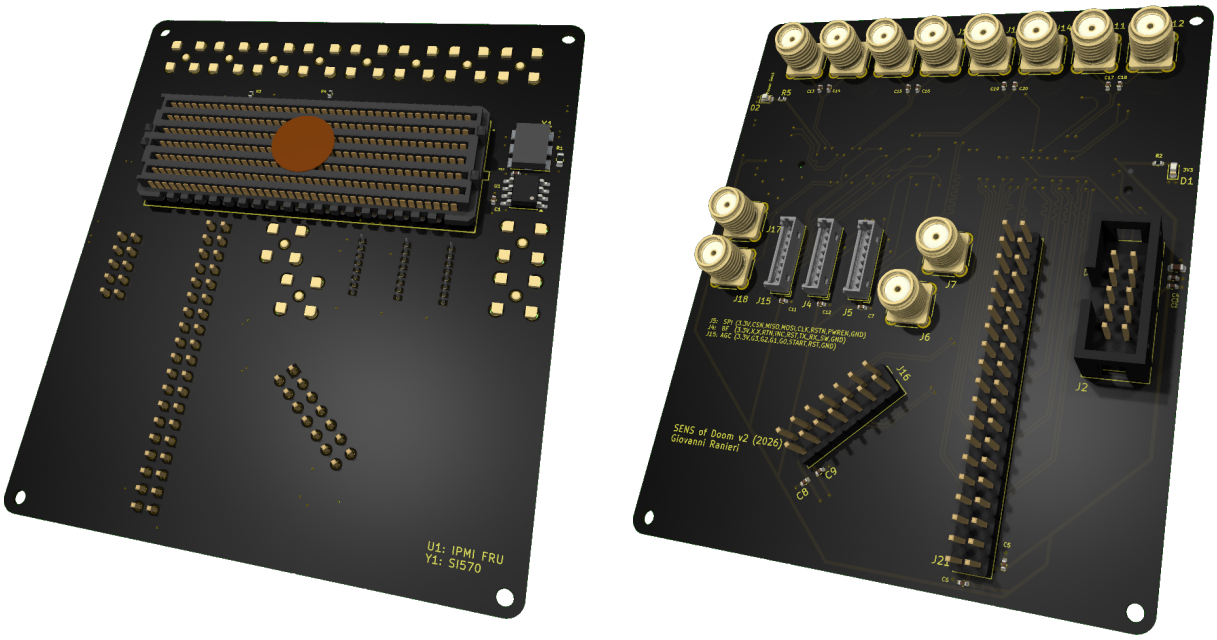
- FMC+ female connector mating with the ZCU208's HPC.
- Three 8-pin Molex headers mirroring the EVK pinout for SPI, BF and AGC.
- 2 kB I2C EEPROM (M24C02-RMN6TP)
- Si570 I2C-programmable XO (10–810 MHz, 3.3V) for low-jitter reference clocking.
- GTY high-speed differential transceivers (5 Gbps) routed to 8 SMA connectors at 50 Ω impedance.
- Pin headers breaking out 12 Low Bank A (LA) differential pairs (with paired GND), including one clock-capable pair, and 3.3V/VADJ power pins.

The important constraints where (i) the *signal integrity* for high-speed transceivers and clocks, and (ii) the simple *accessibility* for the BF and SPI interfaces. For further details on the PCB itself, please refer to its documentation.

4.2 VADJ_FMC and Mezzanine's EEPROM

Following VITA 57.4, any Mezzanine board has to provide a small EEPROM to store a valid descriptor: voltage level of configurable power rails such as VADJ_FMC, number of LA pairs used, etc. The ZCU208 System Controller reads this descriptor and powers VADJ_FMC at the desired voltage level. Without a valid mezzanine's EEPROM, the VADJ_FMC power rail is set to 0V. PL Banks 64 and 67 are powered by this rail, and are the one driving most FMC+ signals. Two workarounds exist:

- Use the System Controller GUI (SCUI) to override VADJ_FMC to a pre-defined voltage. The software did not override correctly even with help of AMD forum, and only the Windows version of it could be launched.
- Embed an EEPROM on the mezzanine that follows VITA 57.4 standard (this is what SENS of Doom does).



(a) SENS of Doom's front layer, highlighting the FMC+ female connector, the low-jitter clock generator SI750 (Y1) and the 2kB I2C EEPROM (U1).

(b) SENS of Doom's back layer. Critical high-speed SMA connectors are isolated from the rest of the board to respect signal integrity and 100 Ω differential impedance.

Figure 6: Render of front and back layers of SENS of Doom. The breakout board is designed primarily to interface the ZCU208 Evaluation Board for communicating with Sivers EVK06005.

The process of generating the EEPROM's data is not documented because the standard is **closed source**. Companies such as `IAMELECTRONICS` buy this standard to develop hardware/-firmware based on this standard. The FRU (Field Replaceable Unit) data is written by the XCZU48DR, with its PS built-in I2C controller. It can be routed via EMIO (emulated MIO pins), which lets us reach the FMC+ I2C bus *without instantiating any fabric logic*. More documentation on EMIO and controller setup can be found in Appendix D. It is worth noting two facts for next users of the ZCU208:

Voltage-Level Translators. FPGA PL Banks are typically powered below 3.3 V (the typical communication protocols voltage level). Even if Bank 68 is driven at 1.2 V for the I2C lines, a PCA9306 Dual Bidirectional I2C Bus and SMBus Voltage-Level Translator is present on the line. More explanations can be found in the datasheet of SENS of Doom.

I2C channel switch (MUX). In large-sized boards we typically find MUXs for communication protocol lines to enable more devices. Following UG1410, the I2C0 and I2C1 lines go through a TCA9548A Low-Voltage 8-Channel I2C Switch. Table 7-2 of its datasheet shows how to activate the different channels. Using Figure 6 of UG1410 we see its the I2C1 line that connects to FMC+ HSPC, through channel 0. We can deduce that activation of this channel is by writing `0x1` at address `0x75`, while deactivation is through the value `0x0`.

4.3 Concerns on VADJ

We couldn't validate the structure of the FRU data written in the EEPROM. The System Controller could reject the data if it does not respect the standard at 100%. We tried to look at the UART output of the System Controller (just like the output of the PS) but without success.

Sometimes we experienced a deactivation of VADJ_FMC, almost certainly indicating incorrect FRU data.

5 Implementations

In this section we show the different implementations for interfacing the EVK06005 and the ZCU208. We implemented (i) a full register configuration using SPI, (ii) a top-level and BF HDL module that can implement BF algorithms, and (iii) a two HDL module to implement [Memisoglu, Sahin, and Arslan, 2022]. All HDL was first **simulated using `GTKWave`** and some of them have been **validated using Integrated Logic Analyzers (ILA) blocks on real hardware**.

5.1 SPI driver

All registers of the EVK06005 (Slave) are programmed via its SPI interface (J15). Four wires are used: SPI_CS_N (active low), SPI_CLK, SPI_MOSI, and SPI_MISO. Every SPI transaction starts with a 16-bit header: 13-bit address field followed by a 3-bit command field, with MSB first. The address auto-increments internally for each subsequent data byte within the same transaction, allowing efficient multi-byte transfers. For reads, the total transaction length is $16 + 8N$ SPI clock cycles, where N is the number of data bytes. No termination clocks are needed. A 1-byte read is 24 clock cycles. For writes, the transaction length is $16 + 8N + T$ clock cycles, where T is between 3 and 8 termination clock cycles required after the last data byte. A 1-byte write is therefore 27 to 32 clock cycles.

We developed a **SPI communication** with the EVK06005 from the ZCU208 using the `XSpiPs` bare-metal driver from Xilinx. Figure [7] shows the memory structure for accessing RX phase shifters. We use a built-in SPI controller from the PL Bank 67 through EMIO. The PS handles it because (i) configuring registers is not time-sensitive and (ii) no SPI is needed directly in PL.

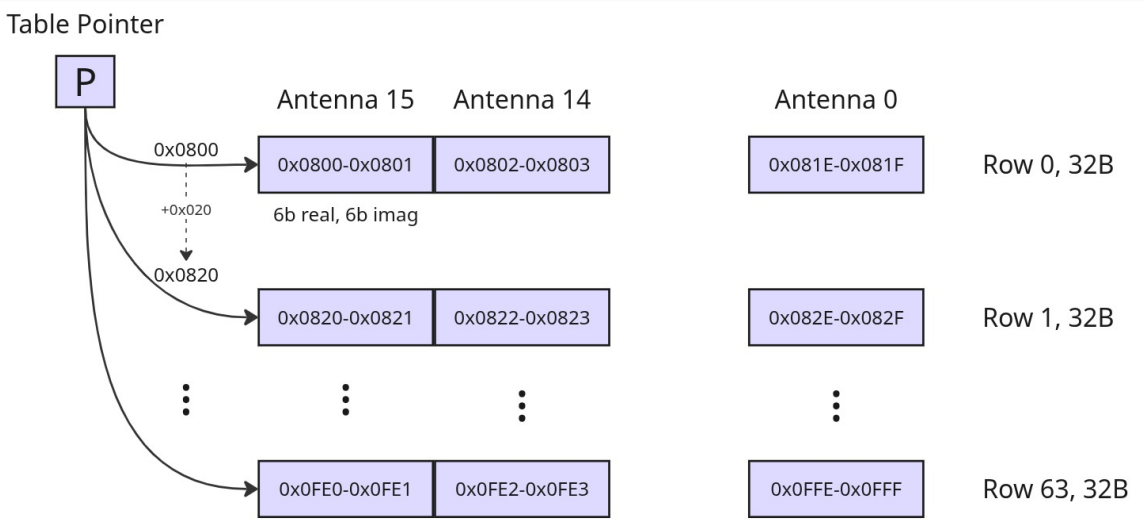


Figure 7: Memory structure of RX phase shifters. Each element is stored on 12b (6b real and 6b imag).

Validation on real hardware. We had large issues validating the driver. Using the ZCU208 SPI_MISO is reacting weirdly and does not respond with the correct register values. We decided to switch to a RFSOC 4x2 with its GPIOs to understand if it was the driver or the hardware

connectivity. Interestingly, on this board SPI_MISO is stuck in high state, even before and after the SPI transaction, and we didn't find any answer to this. Still, we validated the SPI_MOSI using ILAs, as shown in Figure [8]. Find in Appendix E the discussion.

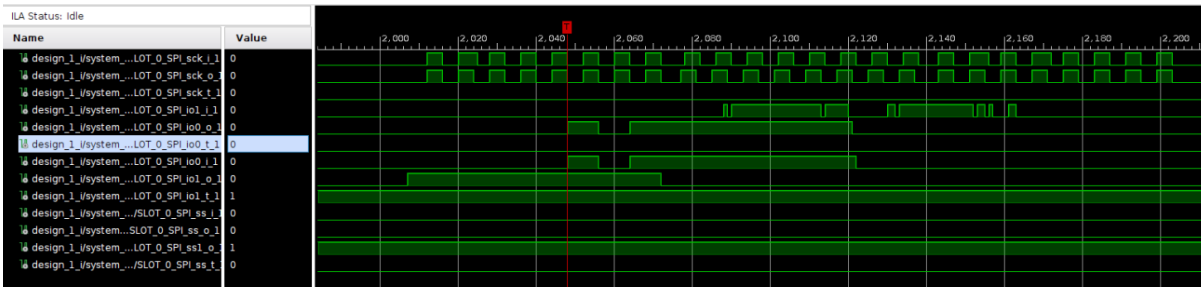


Figure 8: 1 Byte SPI read transaction on ZCU208, triggered by our driver going through the PL using EMIO. MOSI is referenced by `io0_o` and MISO by `io1_i`. We can count 24 SPI cycles, coherent with the EVK06005 specifications. The header access `0x00BF` giving with read command `0b100` a command

$$0\text{x}00\text{BF} \ll 3 \mid 0\text{b}100$$

$$0\text{b}\underbrace{00000110}_{MSB}\underbrace{11111100}_{LSB}$$

We can see on `io0_o` that MSB comes first, followed by LSB, by counting the SPI cycles starting from the left.

5.2 Beamforming IP

The BF interface is composed of 3 signals: (i) BF_INC increases the index in the AWV pointer table by one, changing the phase shifters configuration. For example, element 2 could point to the configuration number 10 in the AWV LUT, (ii) BF_RST resets the index in the AWV pointer table to a pre-programmed default value, and (iii) BF_RTN temporarily sets the index in the AWV pointer table to the pre-programmed default value. We also have access to a particular register `bf_rx_awv_ptr` which bypass this control table and its value directly apply the configuration wanted of the phase shifters.

The top-level IP module, shown in Figure [10], serves primary to instantiate HDL BF algorithms, or any algorithm that requires to process raw I/Q samples. We structured it in the following way (also shown in Figure [9]):

- Parameters of each algorithm are stored in multiple 32b registers through AXI4-Lite. Activate of algorithms is done through PS.
- Busy and Done signals indicate the status of the algorithm.
- Results are stored also in 32b registers and can be read anytime by PS.

Limits in BF. The constraint imposed by the phased array is a minimum of 5 ns high, low and separation time for each pulses (Table 3-2 of EVK06005 datasheet). Because HELIX's PL runs at 245.76 MHz, pulses cannot be on a single clock cycle (ie., this is ≈ 4.08 ns). We widened the pulses to 2 cycles via a counter (in our case its 2). In addition, 35 ns are required to load the weights when switching configurations, giving a theoretical maximum frequency of 25 MHz.

Reading results. The PL cannot save results for only one cycle because the PS has to read results through the AXI4-Lite interface. The result must remain long enough valid to be able to

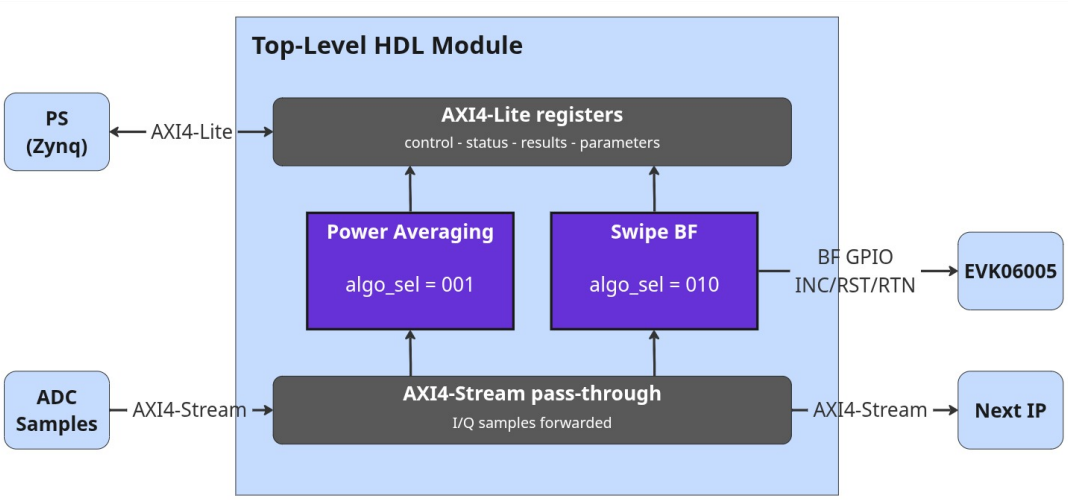


Figure 9: High-Level overview of HDL BF module. I/Q samples pass through the module. Algorithms are selected and results are sent over the AXI4-Lite interface. Any algorithm can be instantiated inside, giving the developer full control.

read it. We decided to leave the PS clear the "done" signal. In addition, we add one cycle of latency to the combinational tvalid/tready paths between AGC (upstream) and CFO_correction (downstream). This is required because a purely combinational pass-through adds a tready loop through this IP that can fail timing at 245.76 MHz (4 ns budget per path).

5.2.1 Power Algorithm

To initially validate the top-level HDL module, we decided to compute the power of I/Q samples over time with a simple average module.

5.2.2 Swipe BF Algorithm

This HDL module is a command-driven module that drives the three BF signals. Rather than toggling these pins directly from software, the higher-level IP issues single-cycle *triggers* and the module autonomously generates the correct pulses. Four commands are available: a reset pulse (`trig_rst`), a single return-to-home pulse (`trig_rtn`), an N-step increment (`trig_inc` together with `num_inc`), and a fully autonomous sweep (`trig_sweep`). While a command is in flight, the module raises `busy`, and it emits a single-cycle `done` pulse on completion, so the caller never needs to know the internal timing.

The core of the algorithm is a simple swipe through the beambook: starting from the current index, the module issues a sequence of `BF_INC` pulses to step through consecutive beams. In sweep mode the module first emits a `BF_RST` to define the starting beam and then walks the requested number of steps on its own.

Return-to-home is handled so that it never corrupts an ongoing sweep. If `trig_rtn` is pulsed while an increment sequence or a sweep is already running, the request is *latched* rather than acted upon immediately. The module finishes the pulse it is currently shaping and, at the next safe point (the low phase of the current pulse, or the end of a dwell), inserts a single `BF_RTN` pulse before resuming the remaining increments. The complete state machine is given in Appendix B, with a simulated waveform to illustrate the algorithm.

We could have developed more advanced HDL for incorporating other algorithms to swipe

differently the beambook, for example we could precompute a set of beam patterns covering an angular space. But the lack of time (due to the initial hardware limitations) excluded this for the project.

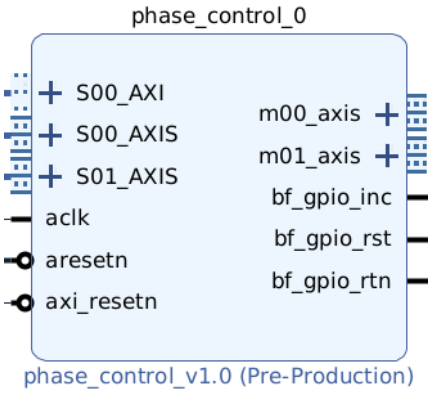


Figure 10: Top-level HDL module for BF algorithms. The module is placed right after AGC in the RX path.

Name	Description
`S00_AXIS`	Input AXI4-Stream, I/Q Lane A. 32 bits wide, carrying the $2k$-th I/Q sample.
`S01_AXIS`	Input AXI4-Stream, I/Q Lane B. 32 bits wide, carrying the $(2k+1)$-th I/Q sample.
`m00_axis`	Output AXI4-Stream, I/Q Lane A. 32 bits wide, carrying the $2k$-th I/Q sample.
`m01_axis`	Output AXI4-Stream, I/Q Lane B. 32 bits wide, carrying the $(2k+1)$-th I/Q sample.
`S00_AXI`	AXI4-Lite control interface.
`bf_gpio_*`	Beamformer interface to the Sivers EVK06005.
Others	Clock and reset signals, driven by the AGC control logic of HELIX.

Table 3: Interface of BF IP with its specifications.

5.3 JSAC IP

The sensing waveform in Memisoglu, Sahin, and Arslan, 2022 proposes is a simple chirp over a finite time T_c. Because this chirp does not change over time, we do not need to use a Direct Digital Synthesizer (DDS) compiler IP from Vivado. We can pre-compute the I/Q samples (real and imaginary parts on 16b), pack them into 32b, and synthesize them in the PL. If we wanted to have control over parameters of the chirp at runtime, we would have needed a DDS compiler (see Appendix [A]). An important remark: the bandwidth it covers in this form is from 0 to B MHz. This means that when the chirp frequency goes above $\frac{f_s}{2}$, it will ramp back up to $-B$ MHz ; which is simply caused by aliasing. Complex baseband can only represent frequencies in the range $[-\frac{f_s}{2}, \frac{f_s}{2}]$. The solution is to center the chirp around DC by changing the waveform to

$$e^{j\pi\frac{B}{T_c}t^2} \rightarrow e^{j\pi\frac{B}{T_c}t^2} \cdot e^{-j\pi Bt}. \tag{7}$$

The instantaneous frequency goes now from $-\frac{B}{2}$ to $\frac{B}{2}$. We created two custom IPs to add and remove chirp samples in the time domain. The IPs possess a counter that synchronize the chirp and the OFDM symbol, and use a trigger signal (`triggerOut`) that goes high when a new slot begins (first OFDM symbol in a slot) to synchronize everything (see this in the FFT_mod_top.vhd:610-611 with the signal `triggerOut`). Visually Figure [11] shows where these blocks have been added, and Tables [4] and [5] their interfaces.

Concerns on T_c. The strict requirement imposed by [Memisoglu, Sahin, and Arslan, 2022] is T_c to be a common divisor of both the OFDM symbol time T_o and the CP time T_{cp}. As mentioned previously, the CP length in HELIX vary (the first OFDM symbol in a slot has a longer prefix). To have no spectrum leakage over time, T_c must be a common divisor of both T_o and the two T_{cp} values. We need the chirp to have a number of samples N_c that respects

$$N_c \mid \gcd(N, \mathrm{CP1}, \mathrm{CP2}). \tag{8}$$

In this case $N_c \in \{1, 2, 4, 8, 16\}$. It is very unfortunate because even with $N_c = 16$ the chirp has to sweep $\frac{2048}{16} = 128$ times over the symbol period, and giving a very few sensing bins in each symbol.

Name	Description
`S00_AXIS`	64-bit Input AXI4-Stream for I/Q samples.
`M00_AXIS`	64-bit Output AXI4-Stream for I/Q samples.
`S00_AXI`	AXI4-Lite interface.
`triggerIn`	Beginning of slot detection trigger.
Others	Clock and reset signals, following the OFDM logic of HELIX.

Table 4: Interface of chirp adder IP.

Name	Description
`S00_AXIS`	Input AXI4-Stream Lane A. 32 bits wide for 2k'th I/Q sample.
`S01_AXIS`	Input AXI4-Stream Lane B. 32 bits wide for (2k+1)'th I/Q sample
`M00_axis`	Output AXI4-Stream I/Q Lane A. 32 bits wide for (2k+1)'th I/Q sample
`M01_axis`	Output AXI4-Stream I/Q Lane B. 32 bits wide for (2k+1)'th I/Q sample
`S00_AXI`	AXI4-Lite interface
`triggerIn`	Beginning slot detection trigger.
others	Clock and resets mechanisms, following OFDM of HELIX.

Table 5: Interface of chirp remover IP.

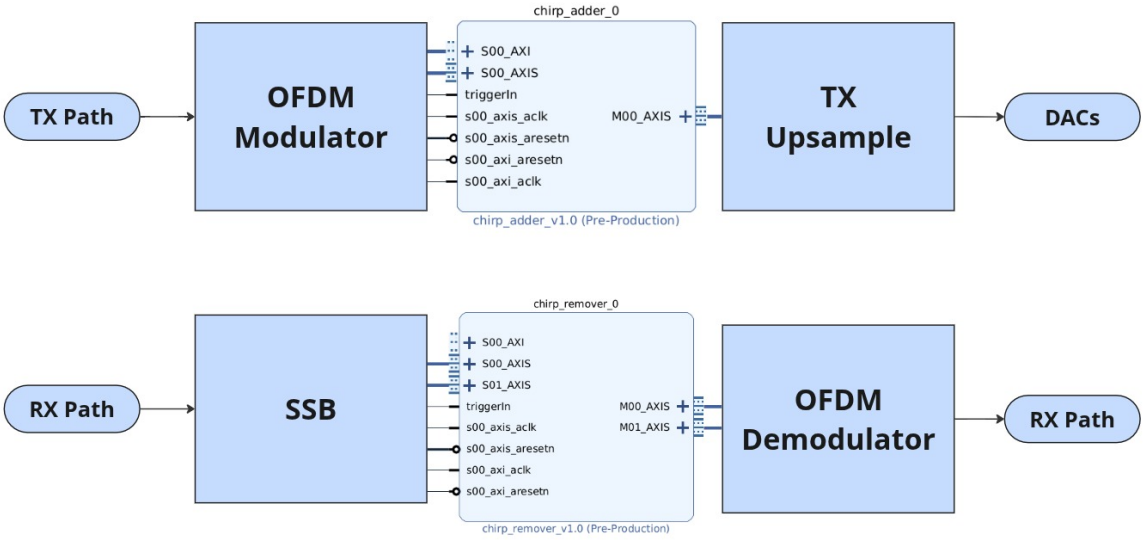


Figure 11: Chirp adder and chirp remover IPs shown in the block design of Vivado.

6 Conclusion

This project investigated the integration of the Sivers EVK06005 phased array with the ZCU208 RFSoC within HELIX, and targeting fast beam-switching and the overlapped radar-communication waveform of [Memisoglu, Sahin, and Arslan, 2022]. Three contributions were delivered. First, *SENS of Doom*, a custom FMC+ breakout board featuring a VITA 57.4 EEPROM and dedicated headers for the EVK06005 SPI/BF interfaces, was designed. Second, a bare-metal SPI driver and a configurable BF IP were implemented; the latter exposes a command-driven swipe algorithm whose FSM respects the 5 ns pulse-width requirement of the phased array at the 245.76 MHz HELIX clock. Third, two custom IPs integrate the JSAC waveform directly into the HELIX OFDM pipeline, with the synchronization logic aligned on the existing `triggerOut` slot marker. Several aspects did not reach completion and are worth stating explicitly.

- The SPI `MISO` path could not be validated: the ZCU208 returned inconsistent register reads and a fallback attempt on a RFSoC 4x2 showed `MISO` stuck high, so only the `MOSI` side is confirmed via ILA.
- Because the VITA 57.4 standard is closed-source, the FRU descriptor written to the mezzanine EEPROM could not be checked against the specification, and intermittent deactivations of `VADJ_FMC` indicate that the descriptor is not yet fully compliant.
- The planned dual-stream extension of HELIX was abandoned after roughly one month due to the lack of documentation of the original design and the difficulty of modifying its custom Verilog UDP stack.
- Finally, the constraint $N_c \mid \gcd(N, \mathrm{CP}_1, \mathrm{CP}_2) = 16$ restricts the chirp to repeat at least 128 times per OFDM symbol, leaving substantially fewer sensing bins available than [Memisoglu, Sahin, and Arslan, 2022] assumes.

Taken together, the project leaves behind a working hardware/software base and a documented codebase on which follow-up work can build. The most natural next steps are to probe the I2C bus to validate the FRU data, debug the SPI return path on the EVK06005, and extend the JSAC IPs with runtime-configurable chirp parameters through a DDS compiler. A second ZCU208 would then enable the full bistatic measurements that were originally envisioned.

7 Acknowledgements

The goal with this project was to gain knowledge with the Vivado and Vitis software suite from AMD/Xilinx, and experience in hardware/software co-design for communication systems. I really want to thank Samah Hussein for letting me dig into this world with this ambitious project, even if we ended up struggling with hardware issues. I also want to thank Dr. Philipp Födisch from IAMELECTRONICS for helping me so fast on the VITA 57.4 standard, but also on reviewing the PCB and debugging via e-mail the System Controller output.

References

AMD/Xilinx (2026). *DDS Compiler LogiCORE IP Product Guide (PG141)*. URL: https://docs.amd.com/r/en-US/pg141-dds-compiler/Introduction.

Memisoglu, Ebubekir, Mehmet Mert Sahin, and Huseyin Arslan (2022). "Orthogonal coexistence of overlapped radar and communication waveforms". In: *2022 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, pp. 2190–2195.

Table 6: AMD/Xilinx documentation for this project report.

Codename	**Description**
UG1410	User manual for the ZCU208 RFSoC. Provides a high-level overview of the board and the elements needed to work with it (boot mode, maximum ratings, etc.).
UG1075	Product specification of the Zynq UltraScale+ device. Includes the pinouts of all chips in the family; page 224 covers the Zynq used in HELIX.
UG643	Describes software packages, including device drivers, libraries, and board support packages used to build a software platform in bare-metal and RTOS-based environments on Vitis.
XTP653	Full schematic of the ZCU208 RFSoC, including the master XDC file with the pinout constraints of the Zynq.
XTP607	Board setup document providing tutorials to correctly configure the ZCU208 with its daughterboards (clocking system and loopback evaluations).
XTP600	Documentation related to the System Controller GUI for the ZCU208.

Ruiz, Rafael et al. (2025). "HELIX: High-speed real-time experimentation platform for 6G wireless networks". In: *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, pp. 305–318.

A DDS Compiler

Native exponential functions do not exist in Verilog. Instead Vivado provides a Digital Synthesizer (DDS) compiler IP, which is essentially a phase accumulator and a Look Up Table (LUT), to generate a specific clock frequency. We would need the DDS compiler if we would need to have runtime modification over the chirp sequence. The output frequency f_o of a DDS Compiler is

$$f_o = \frac{f_c \Delta\theta}{2^{B_\Theta}}, \tag{9}$$

where f_c (Hz) is the input clock, $\Delta\theta$ is the phase increment value (decimal), and B_Θ is the phase width (decimal). In addition

$$B_\Theta = \lceil \log_2(\frac{f_c}{\Delta f}) \rceil, \tag{10}$$

where Δf is the frequency resolution. This second equation is important to deal with the quantization errors that occur when generating phase signals. We recommend to read AMD/Xilinx, 2026 for more advanced users.

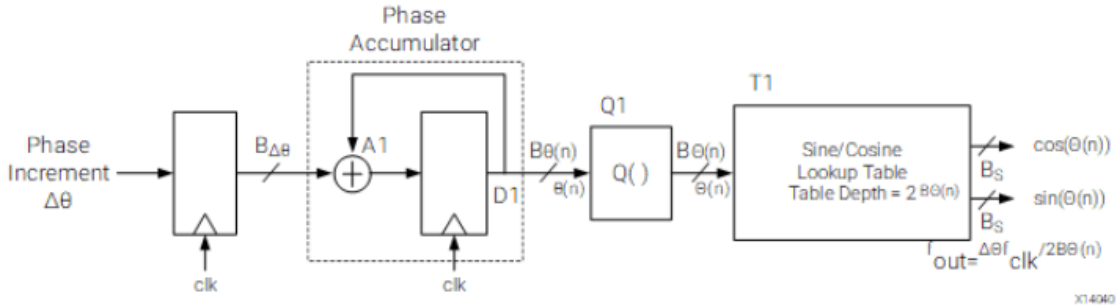


Figure 12: DDS Compiler block diagram, showing the phase accumulator and the Lookup Table storing the cosine and sine values for $\Theta(n)$.

We can't create a LUT with the same resolution because it would be too big. What we do typically is to truncate (quantize) the angle produces before indexing the LUT by discarding some bits. The cost of this truncation is spectral purity. The discarded phase bits introduce small periodic errors into the output values. The level of these spurs is governed by the rule of thumb SFDR: each additional bit of LUT address gains roughly 6 dB of spurious-free dynamic range. The DDS Compiler exposes this trade-off through the SFDR parameter in the GUI rather than asking the user to set it directly. SFDR is the ratio (in dB) between the desired signal power and the worst single spurious component anywhere in the spectrum. A 60 dB SFDR means the worst spur is one millionth the power of the signal, or 0.1% of its amplitude. In a DDS compiler this corresponds to approximately 10 bits.

B Beamforming Verilog Pulses

B.1 FSM of the Verilog Module

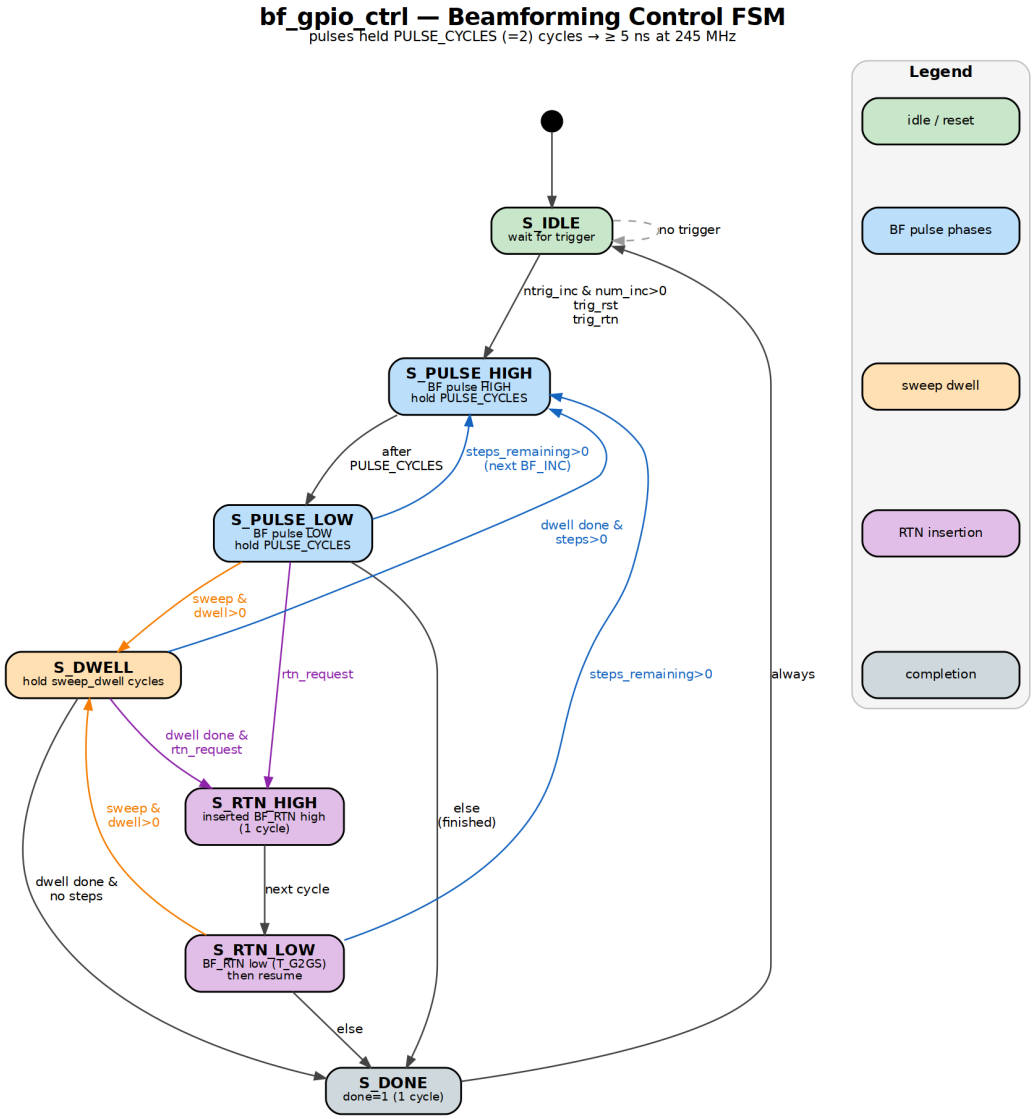


Figure 13: Complete FSM for the BF Verilog module capable of sending BF signals to the phased array while respecting the timing requirements. This module must be instantiated in a top-level Verilog BF algorithm.

B.2 Waveform Examples

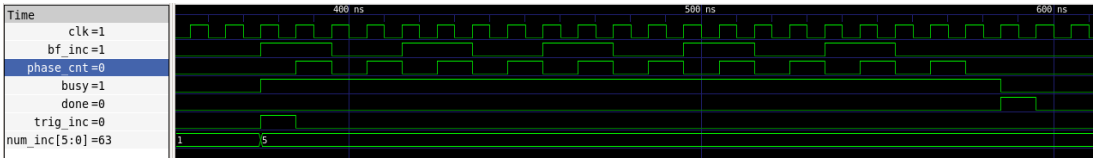


Figure 14: GTKWave simulation of 5 bf_inc pulses triggered by trig_inc. A busy signal is raised from the beginning and a done signal asserts the completion. Note the counter phase_cnt that stretch each pulses on 2 clock cycles such that, as 245.76 MHz, we met the requirements of 5 ns for the pulse width.

C SENS of Doom

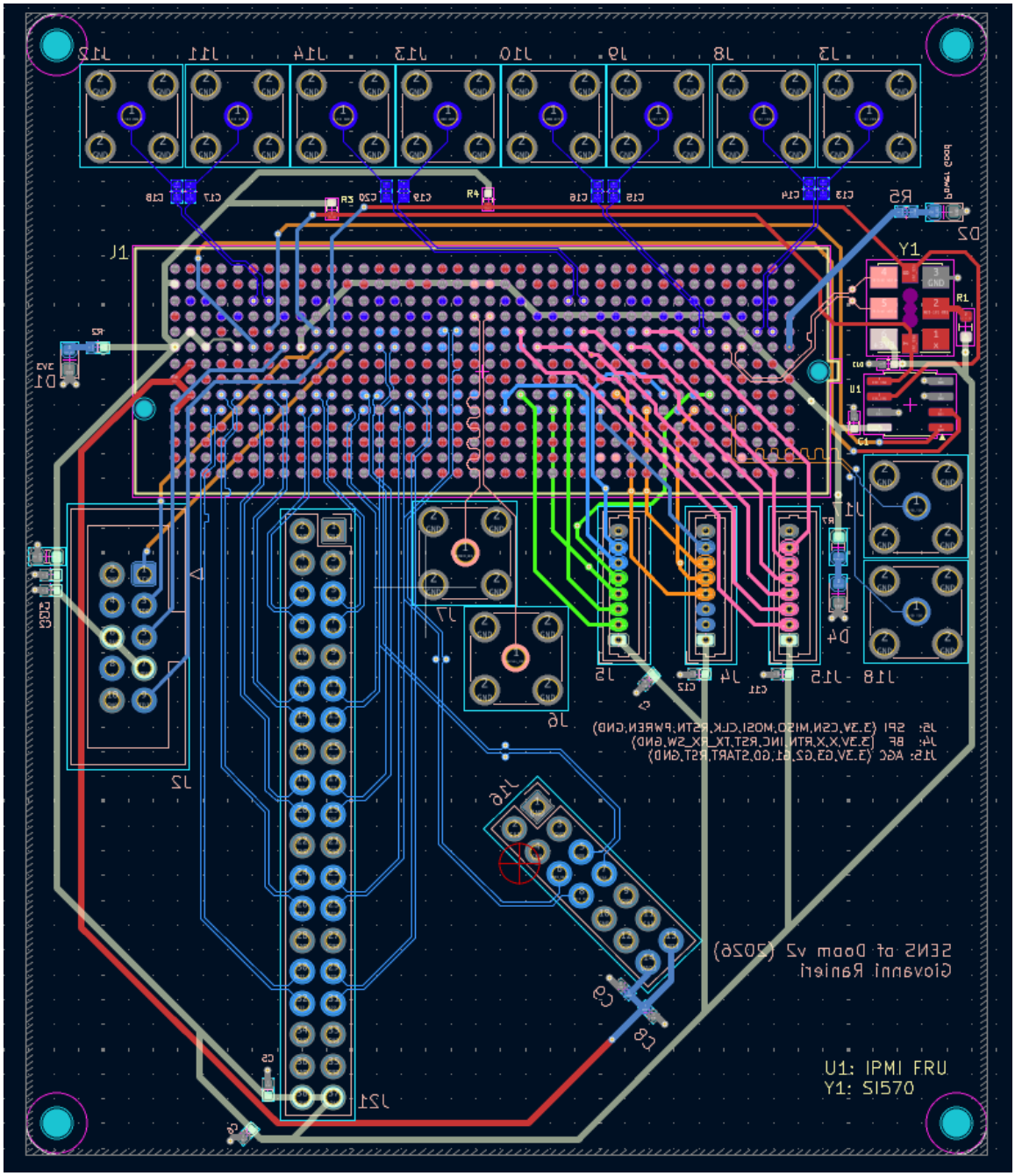
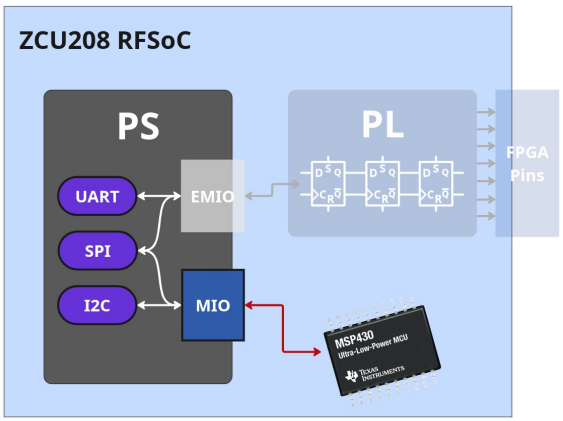


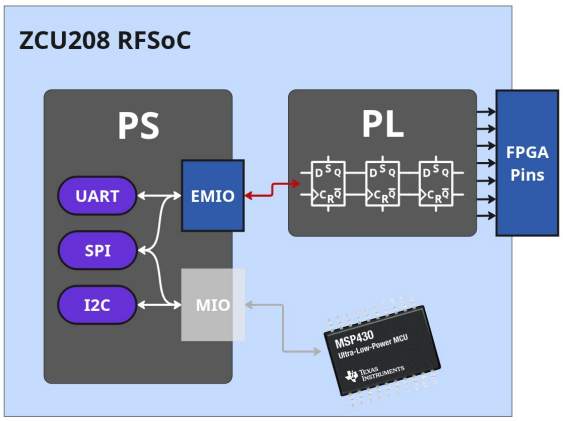
Figure 15: SENS of Doom's Layout

D PS/PL Peripheral Controllers on Zynq UltraScale+

Zynq UltraScale+ devices that combine a Processing System (PS) with Programmable Logic (PL) in the same package. Many standard peripherals, such as I2C, SPI, UART, GPIO, etc can be implemented in either domain. Picking the right one matters for pin assignment, resource cost, software stack, and boot-time availability. Figure [16] shows the differences.



(a) In the case of MIO routing, the controllers are directly multiplexed to different ASICs without going through the PL.



(b) In the case of EMIO from PS to PL, controllers are multiplexed to the PL and are driven by the FPGA pins fabric.

Figure 16: Schema of MIO vs EMIO routing in PS/PL.

D.1 Processing System

It contains a fixed set of peripherals implemented in dedicated silicon (for example two I2C controllers I2C0 and I2C1). These peripherals can reach the outside world through two routing paths:

- **MIO (Multiplexed I/O)**: PS pins on banks 500–503. Pin assignments are constrained to a small set of options defined by the silicon.
- **EMIO (Extended MIO)**: The same peripheral, but its signals are routed through the PL fabric and exit on PL bank pins of your choice.

D.2 Programmable Logic

The PL is the FPGA fabric. To get an I2C bus here for example, you instantiate a Xilinx's AXI IIC IP in the block design. The controller is built from LUTs/FFs/BRAM and connects to PL bank pins through standard FPGA I/O. In a board schematic, you may see PL pins labeled with names like PL_I2C0_SCL. This is just a label indicating what the board designer wired that pin to. It does not imply a hardened controller is attached.

D.3 When to pick which

Use PS and MIO when the bus is already wired to dedicated MIO pins on the board (typical for onboard peripherals like EEPROMs). This is the simplest path and is available before the bitstream loads.

Use PS and EMIO when you want the controller and Linux driver stack, but the physical

pins you need are in the PL banks (e.g. on a custom daughtercard connected via FMC+). This is the cleanest option for "I want firmware on the A53s to talk I2C to something on my add-on board." No fabric IP, no HDL, just a few external ports and an XDC.

D.4 Example: I2C out of the PL bank pins via EMIO

This example is for this project. The reason we want to use I2C is because we need to configure a 2kB-EEPROM on the Mezzanine board via the FMC+ connector. The I2C bus of the FMC+ is routed to the PL Bank 68 (see datasheet of ZCU208). Since we just want the firmware to access this bus, the option of EMIO is the best: expose I2C1 on two pins of a PL bank 68 so it reaches the FMC+ connector on the board.

D.4.1 Configure the PS block

In the Zynq UltraScale+ MPSoC IP GUI go to I/O Configuration, enable I2C1, and set its routing to EMIO. The PS block in the block design will now expose an IIC_1 interface port.

D.4.2 Make the interface external

In the block design canvas: Right-click the IIC_1 interface port, and select Make External. A new external port IIC_1_0 appears on the canvas. Why the interface, not the individual signals? I2C is open-drain bidirectional. The PS exposes three signals per line (_o, _t, _i). That need to drive an IOBUF primitive at the pad. Making the interface external causes Vivado to infer the IOBUFs automatically during synthesis.

D.4.3 Regenerate the HDL wrapper

In the Sources panel, right-click the block design and select Create HDL Wrapper and then Let Vivado manage wrapper and auto-update. In Figure [17], you see a design_1_wrapper.v or .vhd. If you open it, you will see input/output with the protocol name, almost surely with _io suffix. You may see also a VHDL folder with some folder called "Unreferenced (2)". If its the case, Vivado regenerate a second wrapper because the old one had another extension. You can simply click-right on the file in it and do Remove file from project.

D.4.4 Write the XDC

At this point we told Vivado that we wanted to route the I2C1 bus through EMIO pins, and we dealt with tristate and bidirectionality. We need to finish with the actual XDC constraint file to tell the fabric which pins we want to use. Following the master_xdc file provided by Xilinx (XTP653), we see the lines with the names of the schematic PL_I2C1_SCL_LS and PL_I2C1_SDA_LS giving respectively pins G10 and K12 with IOSTANDARD LVCMOS12. Create a new XDC file with the previous specifications.

D.4.5 Software

Because the controller is the hardened PS I2C1, the Linux driver is already in place. You can use the XIicPs of xiicps.h to use the driver.

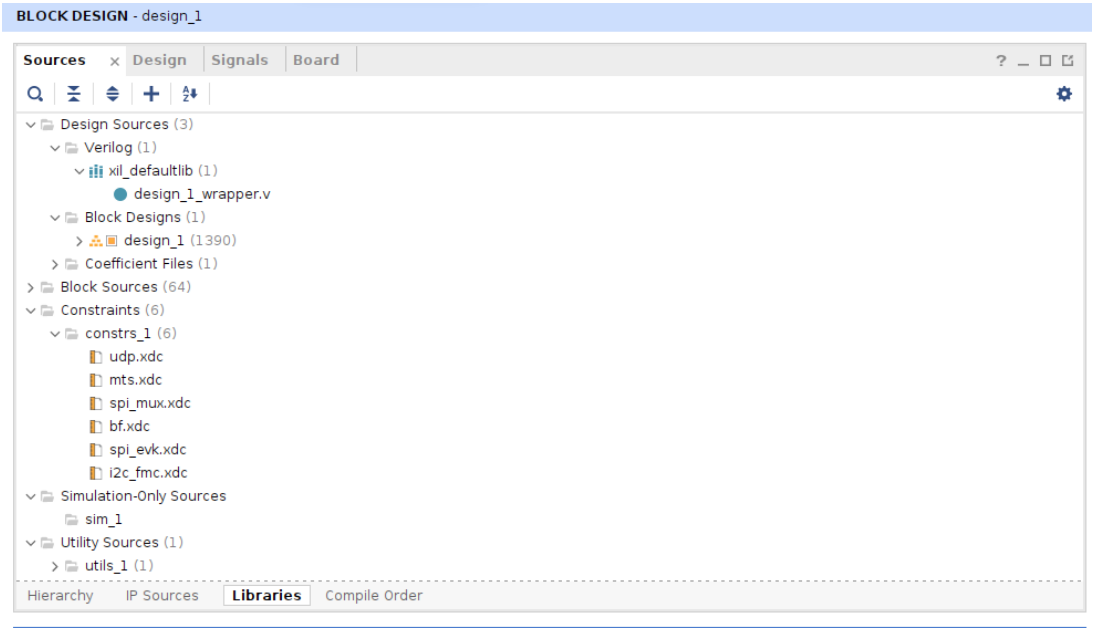


Figure 17: Design Wrapper Example

E Research-Oriented Issues Encountered

Please find in this Appendix section the list of issues (mostly hardware) encountered. **These issues shows the limitations we were not expecting and slowed down the entire project**.

E.1 VITA 57.4 Standard

The first step of the project was to physically connect the ZCU208 and the Sivers EVK06005 together in a way of using the external fast-beam switching mechanism of the phased array. No one would have thought that we needed to develop a PCB, which is a skill in itself, especially for IC students who do not know anything of electrical engineering. Hopefully, we had knowledge of PCB design and managed to create two version of it. They have to be assembled by the manufacturer because of layout of the FMC+ connector.

The PCB in itself is not difficult. The bottleneck was the standard of the FMC+ connector. The ZCU208 must follow the standard since it incorporates this connector. UG1410 describes in its chapter "VADJ_FMC Power Rail" and "ZCU208 MSP430 System Controller (SC)" in a high-level overview what happens with VITA 57.4: the voltage rail VADJ is a programmable power rail used as voltage level for LA and HA/HB signal banks going in and out the FMC+ connector. When a breakout board is connected, a sequence of operation happens with the SC to select a valid voltage level. The V1 of the PCB did not have an EEPROM to store this voltage level, setting to 0V this rail by default. The V2 included this small 2-kB I2C-EEPROM. Now the next question was how and what to write in it.

The I2C communication is done using Appendix D. We didn't think that generating the data was difficult. We started to look for tutorials or documents explaining this procedure, but we have been surprised that nothing was online. We initially, with the help of AI, created the FRU data using *frugy* which is a Python-based tool that generates EEPROM images according to the IPMI FRU (Platform Management FRU Information Storage Definition) standard from YAML configuration files. Without any result we decided to contact people in this domain. We

contacted Dr. Philipp Födisch because we searched for open-source FMC+ PCBs, and we ended discussing for one week. He reviewed the PCB and our methodology to correctly setup the EEPROM. He also shared that normally **people buy the standard (licensed from Samtec)** to have all required information.

The last step would have been to probe the I2C line or debug the output the the MSP430 but the lack of time imposed us to continue without the SPI communication and the EEPROM.