Maia SDR: an Open-Source FPGA-Based SDR Project Focusing on the ADALM Pluto

Daniel Estévez
Independent Researcher
Madrid, Spain
daniel@destevez.net

***Abstract*—Maia SDR is a new open-source project with the main goal of promoting FPGA development for SDR and increasing the collaboration between the open-source SDR and FPGA communities. In the first stage of the project, the focus is on the development of a firmware image for the ADALM Pluto that performs most of the signal processing on the FPGA of its Zynq system-on-chip.**

A first version already provides a real-time waterfall display at up to 61.44 Msps in a WebSDR-like interface, and supports recording IQ samples at up to 61.44 Msps in the Pluto DDR (400 MiB maximum recording size).

The FPGA design is written in Amaranth, a Python-based HDL. There is an application running on the Zynq ARM CPU that provides a web server with a REST API for user interface and control. It is written in asynchronous Rust. Finally, the user interface is a web application written in Rust and compiled to WebAssembly. It uses WebGL2 to render the waterfall using a GPU.

***Index Terms*—SDR, FPGA, Zynq, DSP, Rust, WebAssembly, WebGL2, ADALM Pluto**

I. Introduction

Maia SDR [1] is a new open-source project with the goal of developing radio applications whose signal processing runs mainly on an FPGA. The longer term goals of the project are to foster open-source development of SDR applications on FPGA, and to promote the collaboration between the open-source SDR and FPGA communities. At the start of this project, instead of developing custom hardware, focusing on developing a firmware image for the Analog Devices ADALM Pluto that uses the FPGA for most of the signal processing gives realistic goals and provides a product based on readily available hardware that people can already use during early stages of development.

The first version of Maia SDR was released in February 2023, though its development started in September 2022. This version consists of a firmware image for the ADALM Pluto with the following features:

- Web-based interface that can be accessed from a smartphone, PC or other device.
- Real-time waterfall display supporting up to 61.44 Msps (limit given by the AD936x RFIC of the Pluto).
- IQ recording in SigMF format [2], at up to 61.44 Msps and with a 400 MiB maximum data size (limit given by the Pluto DDR size). Recordings can be downloaded to a smartphone or other device.

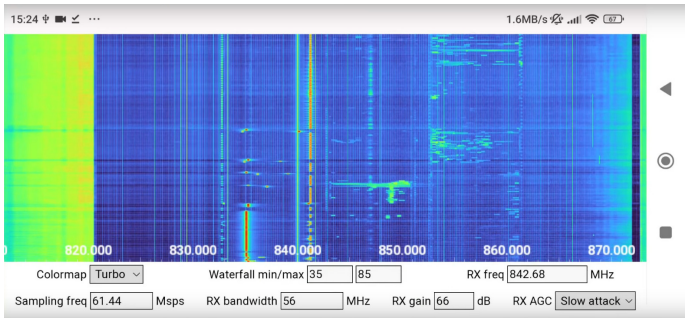

Fig. 1. Screenshot of the Maia SDR web user interface running on an Android phone.

Exploring the RF world in the field with a portable device is one of the goals of Maia SDR, so its web user interface is developed having in mind the usage from a smartphone (see Figure 1). It fully supports touch gestures to zoom and scroll the waterfall. A Pluto connected by USB Ethernet to a smartphone gives a capable and portable tool to discover and record signals.

Maia SDR is formed by three components:

- `maia-hdl`, which is the FPGA design. It is written in Amaranth [3], a hardware description language based on Python. It includes a custom pipelined FFT core focusing on low resource usage and flexible configuration, and DMAs with low resource usage and high throughput.
- `maia-httpd`, which is an application that runs in the ARM CPU in the Pluto Zynq. It is written in asynchronous Rust [4] and provides an HTTP server that serves the web user interface and allows control using a REST API. The waterfall data is sent using WebSockets. This allows to control and access Maia SDR from scripts and other applications easily.
- `maia-wasm`, which is a web application that serves as the user interface. It is written in Rust, which gets compiled to WebAssembly. The waterfall is rendered on the client's GPU using WebGL2.

Additionally, there is a Linux kernel module called `maia-kmod` that is used to handle the DMA buffers required by the `maia-hdl` IP core.

This paper describes the main technical features in Maia SDR. Section II shows the main components, clocking and

FPGA utilization of `maia-hdl`. Section III mentions the main features of the `maia-httpd` application that runs on the ARM CPU of the Zynq. Section IV describes the web user interface `maia-wasm`. Section V explains some tricky aspects regarding the CPU caches, and how they are solved by the Linux kernel module `maia-kmod`. The paper finishes with some conclusions in Section VI.

II. FPGA DESIGN: MAIA-HDL

The FPGA design of Maia SDR is called `maia-hdl`. It is bundled as a Python package, to make it easier to reuse the modules in third-party designs. The main design drivers of `maia-hdl` are low resource usage and flexibility. The Zynq 7010 FPGA in the ADALM Pluto is not particularly large (see Section II-F), so low resource usage is important to fit as many features as possible in the FPGA.

The `maia-hdl` implementation is written in Amaranth, which is a Python-based HDL and toolchain. Amaranth is an important project in the open-source FPGA community, and there are some previous projects using Amaranth for radio applications [5] [6]. Amaranth gives many advantages over a traditional HDL such as Verilog or VHDL, since Python has much higher expressiveness than these HDLs. This is very useful to decide design parameters at synthesis time, following fairly complex rules described in code. See [7] for concrete examples.

A. FFT core

One of the main modules of `maia-hdl` is a custom pipelined FFT core. It supports radix-2, radix-4 and radix-2^2 single-delay-feedback decimate-in-frequency architectures (see [8], for instance), and can optionally apply a window before the FFT computation. To reduce the number of DSPs used, it is possible to perform the complex products required to apply the twiddle factors with a single DSP that runs with a clock frequency three times that of the data, instead of using three DSPs (an efficient implementation of the product of two complex numbers requires only three real multiplications). Likewise, it is possible to apply the window to the real and imaginary parts of the input using a single DSP that runs at twice the clock frequency of the data instead of two DSPs. Calculations are done in fixed-point arithmetic, and truncation (instead of rounding or convergent rounding) is used for scaling between the stages, in order to save logic resources. The truncation schedule is configurable.

The FFT core can store the delay lines required by the single-delay-feedback butterflies either in SRL primitives (which use LUTM resources) or in BRAM. Likewise, the twiddle factors can be stored in LUTs or BRAM. By default, there are automatic rules that decide which resource to use depending on the number of elements to be stored (which depends on the pipeline stage and the FFT size), but this choice can be overridden.

In the Maia SDR Pluto firmware, the FFT core is used to implement a spectrometer that computes the waterfall data. It is configured with a size of 4096 points, a Blackman-harris window, and a radix-2^2 single-delay-feedback architecture (since among the supported ones, this architecture gives the lowest resource usage). The input is 12 bits wide (matching the AD936x ADC), and the output is 18 bits wide. The FFT core runs using a clock of 62.5 MHz used as data clock (this value is slightly larger than the maximum sample rate of 61.44 Msps), and 2x and 3x clocks at 125 MHz and 187.5 MHz to reuse DSPs as described above.

B. Spectrometer

The FFT core is used in a spectrometer, which generates the waterfall data. First, the windowed FFT of each 4096-point vector of IQ samples is computed. No IQ samples are discarded in this process, so the FFT runs at the full sample rate of the AD936x. Next, the modulus squared of the FFT output is calculated, and several consecutive FFT vectors are integrated (non-coherently) to reduce the data rate and increase the sensitivity. This produces measurements representing average power spectral density. A max-hold option that uses a maximum function instead of a summation will be implemented in the future.

The measurement rate, defined by the number of FFT vectors that are integrated together, is configurable at runtime. The maximum number of integrations per measurement that is supported is 1023. This gives a measurement rate of 14.7 Hz at 61.44 Msps. The minimum number of integrations per measurement is one, but this produces a very high output rate and measurements are lost in the RAM circular buffer used to pass the data from the FPGA to the CPU using a DMA.

C. DMAs

There are two custom DMA engines used in Maia SDR:

- `DmaBRAMWrite`. It copies the contents of a BRAM into one of the buffers of a circular buffer in DDR. It is used in the spectrometer. Each spectrometer measurement (integrated FFTs) corresponds to one of the copies of this DMA.
- `DmaStreamWrite`. It reads data from an interface similar to AXI4-Stream and writes to a buffer in DDR until the end of the buffer is reached or a stop command is received. It is used in the IQ recorder.

The main design driver of these DMAs is simplicity, low resource usage and high performance. They have a minimal set of control inputs, and the addresses and sizes of the buffers are statically defined at synthesis time. Additionally, they implement the AXI3 protocol instead of AXI4, because the HP (high performance) ports that connect the FPGA to the Zynq PS (processing system, meaning CPU) use AXI3. In contrast, Xilinx's ecosystem, which is commonly used in FPGA designs for the Zynq, is based on AXI4, so a protocol converter IP core is required to interface to the PS.

These two DMAs are a good example of how an AXI3 Manager can be implemented in a minimalist way.

D. AD936x interface

The interface to the AD936x RFIC in the Maia SDR FPGA design is done using the Analog Devices `axi_ad9361` IP core. This is the same IP core that is used in the default firmware for the Pluto and in other reference designs from Analog Devices. In Maia SDR, this IP core has been configured to reduce its resource usage: IQ corrections, ADC DC filtering and DAC DDS are disabled, and the number of RX/TX data path channels is set to one (the AD9363 and AD9361 support two RX and two TX channels, but in Pluto hardware revisions B and earlier the second channel is not even routed in the PCB).

Even with this configuration, the `axi_ad9361` IP core has a relatively high resource usage compared to other modules in the Maia SDR FPGA design. Probably a custom AD936x interface could be implemented with much lower resource usage. Perhaps this will be done in the future, when resource usage becomes important to add more functionality to the design.

E. Clocking

These are the clocks used in the Maia SDR FPGA design:

- AD936x interface clock. It is a clock generated by the AD936x and received through the CMOS interface between the AD936x and the FPGA. Its frequency corresponds to the sample rate, so it ranges between slightly above 2 MHz and 61.44 MHz. This clock is used by the `axi_ad9361` core and by the part of the Maia SDR IP core that receives the ADC IQ samples from the `axi_ad9361` core. The IQ samples go through clock domain crossings at the inputs of the spectrometer and the IQ recorder as described below. For these, the FIFO18_36 primitive is used, since it gives an effective and low resource usage way of doing clock domain crossing for a stream of data.
- 100 MHz clock FCLK0. In Zynq designs it is very common to use one of the four PL (programmable logic) clocks available in the PS to generate a 100 MHz clock, which is used for the AXI4-Lite interfaces that allow controlling FPGA IP cores from the CPU, and also for other general clocking needs in the FPGA. The Maia SDR design also does this. The 100 MHz clock is used in the AXI4-Lite interfaces of Maia SDR and the `axi_ad9361`, and also in the Maia SDR IQ recorder DMA and the Zynq HP port it is connected to. This 100 MHz clock is also used to generate the following three clocks with an MMCM tile. These clocks are used in the Maia SDR IP core.
  - 62.5 MHz clock. This clock is used to run the Maia SDR FFT core, spectrometer, and its DMA and corresponding Zynq HP port. Additional signal processing functions to be added in the future will also use this clock. The reason for using this clock instead of the AD936x interface clock directly is that this clock has a fixed frequency, while the interface clock frequency is variable. This simplifies generating clocks at twice and three times the frequency, which are required by the FFT core (in fact, for low sampling rates the AD936x interface clock frequency is too low to be used as the input clock for an MMCM tile). The frequency of this clock is chosen to be slightly higher than 61.44 MHz, while supporting the required frequency multiplication ratios in the MMCM.
  - 125 MHz clock. This clock is synchronous to the 62.5 MHz clock and has twice its frequency. It is used in the FFT core and other future signal processing functions that need to run DSPs at twice the data clock frequency.
  - 187.5 MHz clock. Similarly to the previous clock, this is used to run DSPs at three times the data clock frequency.
- 200 MHz clock FCLK1. This clock is used by the `axi_ad9361` IP core to control the delay of the AD936x interface, using IDELAY and ODELAY primitives.

TABLE I
MAIA SDR FPGA RESOURCE UTILIZATION

	LUTs	**Registers**	**BRAM**	**DSPs**
Zynq 7010	17600	35200	60	80
Full design	4421	4767	20.5	8
- `axi_ad9361`	1410	2190	0	0
- AXI interconnect	520	675	0	0
- Maia SDR IP	2461	1871	20.5	8
- - IQ recorder	68	112	0.5	0
- - Spectrometer	2333	1495	19.5	8
- - - DMA	26	46	0	0
- - - FFT	2196	1368	9.5	6
- - - Integrator	108	80	10	2

F. Resource utilization

Table I shows the FPGA resource utilization of the Maia SDR ADALM Pluto design. The first row shows the total amount of resources available in the Zynq 7010, and the next line shows the resources used by the complete Maia SDR design. The full design is divided into its main components: the `axi_ad9361` IP core, the Maia SDR IP core, and a Xilinx AXI interconnect that is used to connect the AXI4-Lite interfaces of the `axi_ad9361` and Maia SDR IP cores to the Zynq GP0 port. Inside the Maia SDR IP, the usage of its two main modules, the IQ recorder and the spectrometer, is shown. The spectrometer is broken down into another level, which lists how the resources are divided between the DMA, the FFT and the integrator.

From this table, it is clear that there is more than enough room to add new functionality in Maia SDR, even though the Zynq 7010 FPGA in the Pluto is relatively small. Another conclusion is that a custom AD936x interface and AXI interconnect would greatly reduce the utilization of these parts of the design. Note that the spectrometer DMA, and also the complete IQ recorder (which includes another DMA) use only a few dozens of LUTs and registers.

III. ARM CPU APPLICATION: MAIA-HTTPD

The Zynq ARM runs an application called `maia-httpd` that controls the FPGA IP core and the AD936x, and provides an HTTP server that serves the web user interface and gives a REST API for control. The REST API is used by the web user interface and can also by used by third-party applications and scripts. This application is implemented in Rust using `axum`, which is a web server framework written in `async` Rust.

Access to the Maia SDR IP core is done through a UIO device. This is a mechanism in the Linux kernel that allows user space applications to `mmap()` the registers of a hardware device and to wait for interrupts sent by the device. The `svd2rust` tool is used to generate an idiomatic and safe Rust API to access the registers. This tool generates code for such an API from an SVD file, which is an XML description of the registers in an SoC (system-on-chip). The SVD file is produced directly by the Python Amaranth code that implements the Maia SDR IP core registers. This automatic code generation process ensures the consistency between software and hardware regarding register addresses, sizes, etc.

The control of the AD936x is done through the sysfs interface of the IIO Linux kernel module for this device. This allows to control parameters such as the LO frequency, sampling rate, RF bandwidth and AGC through reads and writes to files in `/sys/bus/iio/devices/`.

The REST API uses JSON to format the requests and responses. The "schema" for these JSON messages is defined in Rust code, in the `maia-json` crate, which is shared by `maia-httpd` and `maia-wasm`. This crate uses `serde` (a framework for serialization and deserialization in Rust) to implement translation between JSON and Rust `struct`'s according to this schema.

The waterfall data is sent by a WebSocket server. Each waterfall line (corresponding to one measurement of the spectrometer) is sent as a WebSocket binary message that contains 4096 `float32`'s in little-endian format. This makes it easy to process the waterfall data in other applications. An example Python script that displays a real-time spectrum using Matplotlib is provided with `maia-httpd`.

Another interesting aspect of `maia-httpd` is the IQ recording download function. The IQ data is written to the DDR by the FPGA, either as packed 12-bit integers (so each IQ sample occupies 3 bytes), or as 8-bit integers (so each IQ sample occupies 2 bytes), depending on the recording mode that was selected. From the user's point of view, the recording is downloaded as a SigMF archive, which is a `tar` file that contains a JSON metadata file and a binary data file arranged in a specific way. Since SigMF does not support 12-bit sampling, 16-bit sampling is used to store recordings done in 12-bit mode. The raw recording data occupies up to 400 MiB, out of the total 512 MiB of the Pluto DDR, so there is not enough extra space to format the SigMF archive in RAM. Therefore, the SigMF archive is formatted on the fly when the download is done (this includes the `tar` format and the conversion from 12-bit to 16-bit if needed).

IV. WEB USER INTERFACE: MAIA-WASM

The user interface of Maia SDR is a web application called `maia-wasm`. It is written in Rust and compiled to WebAssembly. The application connects to the `maia-httpd` WebSocket server to fetch waterfall data, renders the waterfall in an HTML canvas using WebGL2, and provides HTML form controls to read and write settings (receive frequency, sampling rate, recording controls, etc.) using the REST API.

A custom render engine is used to render the waterfall. The waterfall data (in dB units) is written to a texture as it is received line by line. To do this efficiently, the WebGL2 method `texSubImage2D()` is used to update only the parts of the texture that have changed. The texture is mapped on a rectangle whose height is 3 times larger than the height that is visible on the screen. The waterfall texture is mapped 1.5 times onto this rectangle, giving the illusion that the top and the bottom of the texture seamlessly glue together. This makes it appear that the waterfall is continuously scrolling up as time advances. What actually happens is that waterfall data is written to the texture in such a way that when the end is reached, writing goes back to the beginning of the texture. Only one third of this rectangle is shown on the screen, with the freshly written data always at the bottom. This arrangement of geometry causes the last half of the written waterfall data to be seamlessly visible on the screen even if the write wraps around the end of the texture.

The fragment shader for the waterfall maps the waterfall texture data to a colormap stored in a 1D texture, by using the `texture()` shading language function to perform a lookup. Currently, the Turbo, Viridis and Inferno colormaps are supported, and it is easy to add new colormaps represented by an array of values.

The render engine also manages the ticks of the frequency axis of the waterfall. These are drawn using the `LINES` WebGL2 draw mode and displayed or hidden dynamically according to the zoom level. The labels in the frequency axis are small pieces of a single texture to which all the required labels have been previously rendered. The render engine has a text render function that uses an HTML canvas (not visible on the screen) and the `fillText()` method of the `CanvasRenderingContext2D` to render all the required text in the canvas. The canvas bitmap is then copied to a WebGL2 texture.

The waterfall can be used with a mouse or with a touchscreen. With a mouse, the waterfall can be zoomed using the wheel, and scrolled in frequency by clicking and dragging horizontally. With a touchscreen, zoom is done using two-finger pinch gestures, and frequency scrolling is done by dragging horizontally. In either case, if the user attempts to drag far enough past the left or right edge of the waterfall, the receive frequency of the AD936x will change. This feature can be used to scroll quickly through large portions of RF spectrum.

The `maia-wasm` waterfall is prepared to be re-usable in other projects. There have already been some experiments

about using it in FutureSDR [9], and a demo version that uses waterfall data pre-stored in a JPEG file is available in Maia SDR's website[1].

Another important part of `maia-wasm` handles the HTML user interface. This is composed by HTML `input`, `select` and `button` elements. The user can read settings such as the receive frequency and the sample rate in `input` elements, write new values in these elements to change the settings, use `select` elements to change settings such as the waterfall colormap, and press `button` elements to start or stop the IQ recording.

Most of these HTML user interface elements are tied to variables in the `maia-httpd` REST API by `maia-wasm` (a few others correspond to settings that are implemented in the client side, such as changing the waterfall colormap). A periodic GET request fetches all the values of the REST API and updates the values displayed by these elements. When the user changes the value of one of the elements, a PATCH request is sent to the REST API to modify the value. The code that implements this has a substantial amount of boilerplate, especially because REST API requests are done using the JavaScript fetch API, which is asynchronous. Therefore, the code often involves converting Rust futures into JavaScript promises. Extensive use of Rust macros is done to generate most of the boilerplate code, taking advantage of the fact that many user interface elements are handled in the same way.

There are some additional interesting features of `maia-wasm` which account for the fact that the ADALM Pluto does not have persistent storage (unless a flash partition is formatted using JFFS2, which Maia SDR does not do) nor a real-time clock. The current settings (frequency, sample rate, etc.) are saved in the web browser local storage. This means that if the Pluto is rebooted and the same web browser is connected to it again, then `maia-wasm` will read the last settings from the local storage and apply them. One of the values included in the REST API of `maia-httpd` is the server time, which corresponds to the time according to the Linux clock on the Pluto. If `maia-wasm` detects that the server time is off by more than one second compared to the clock of the device in which `maia-wasm` runs, then it will send an API request to update the server time. This is used to keep the Pluto Linux clock synchronized, which is important because IQ recordings are timestamped using this clock.

V. Linux kernel module: maia-kmod

Maia SDR uses the Linux kernel module `maia-kmod` to access the DMA buffers in DDR from the user space application `maia-httpd`. This kernel module implements the `mmap()` system call for these buffers and performs cache invalidation as required.

The Zynq has two different high-throughput ports from which AXI Managers in the FPGA can access resources of the SoC such as the DDR. There are four HP ports, which give non-coherent access to the DDR controller. This means that accesses through these ports do not go through the L1 and L2 caches used by the ARM CPUs. If the FPGA writes to the DDR through these ports, the corresponding cache lines need to be invalidated by the CPU to prevent reading old data. There is also one ACP port (accelerator coherency port), which gives coherent access to the DDR. Accesses from this port go through the SCU (snoop control unit), which gives access through the L2 cache and invalidates data in the L1 caches when needed.

Using the ACP port is much simpler from the point of view of the software, because no cache management is needed. However, in many use cases it is undesirable to use this port, because writes through the ACP port can evict cache lines used by the application that is currently running (especially because the cache eviction policy in the Zynq is pseudo-random). It is preferable that the data ends up in the DDR without being written to the caches. The IQ recorder is a good example. A large amount of data is written to the DDR, but it will only be used later on, so there is no point in caching it during the write.

Another possibility to avoid cache management is to map the DDR buffers from the CPU as uncached memory. However, this causes a large performance penalty.

Maia SDR uses the HP ports and, for best performance, maps the DDR buffers as regular cached memory. Therefore, `maia-kmod` must handle the cached invalidation for the buffers. This is done in the following ways:

- For the IQ recorder, which is a single buffer, cache invalidation is done as part of the `mmap()` implementation. This matches the usage that `maia-hdl` does of this buffer. After a recording has finished, when the data download is requested by the user, the buffer is `mmap()`-ed. It is `munmap()`-ed when the download finishes.
- For the spectrometer there is a ring of buffers, each holding a single measurement. The ring is mapped all the time in `maia-hdl` through a single `mmap()` call. The `ioctl()` system call is used by `maia-hdl` to command the cache invalidation of a single buffer in the ring, immediately before the buffer is read.

The details of the implementation of cache invalidation are quite tricky. The Linux kernel has a DMA API [10] that allows mapping DMA buffers with `dma_map_single()` and to perform cache invalidation with `dma_sync_single_for_cpu()`. However, this has an important limitation, which is that the buffers need to be mapped in the kernel address space. This makes sense in most applications, because the data will be accessed by the kernel. However, in Maia SDR the data is accessed by the user space application `maia-httpd`. In a 32-bit architecture such as the Zynq ARM, the kernel typically uses the 3G/1G split, where only 1 GiB of the available virtual address space is mapped by the kernel. Because the kernel uses virtual memory mappings for many purposes, this means that in practice it is not possible to map the 400 MiB buffer used by the IQ recorder in the kernel virtual address space.

[1] https://maia-sdr.org/waterfall-demo/

The `maia-kmod` implementation gets around this limitation by not using the DMA API, and calling the cache invalidation functions directly. Bypassing the DMA API means that the kernel infrastructure that abstracts different cache controllers cannot be used, so low-level functions need to be called directly. These functions are `v7_dma_inv_range()` for the ARMv7 L1 cache (this is an assembler function in `arch/arm/mm/cache-v7.S`), and `outer_inv_range()` for the L2 cache (this is a C function in `arch/arm/include/asm/outercache.h` that calls the `inv_range()` function of the L2 cache handler, which in the case of the Zynq PL310 is the `l2c210_inv_range()` function in `arch/arm/mm/cache-l2x0.c`). These low-level functions are not exported by the kernel, because in principle they should not be called from kernel modules. Maia SDR uses a modified Linux kernel that exports these functions.

It should be mentioned that performing cache invalidation in this way is not exempt of problems. The L1 ARM cache is physically indexed and physically tagged, but cache invalidation is performed line by line using virtual addresses. Similarly, the L2 cache is physically addressed and physically tagged, and cache invalidation is done line by line using physical addresses. This is quite inefficient for invalidating large buffers, of which only a few or no cache lines will actually be present in the caches. The alternative would be to invalidate the complete caches. However, it would be quite difficult to do this without disturbing Linux's cache management system.

VI. Conclusions

In its present state, Maia SDR provides a firmware image for the ADALM Pluto with interesting features beyond the capabilities offered by the default firmware image by Analog Devices. It also gives a solid foundation for building FPGA signal processing applications connected to a web user interface, since the plumbing that moves the data from the FPGA to the web user interface is already taken care of.

In the future, it is planned to implement FPGA modules to downconvert and decimate signals and perform analog demodulation in the usual SSB, AM and FM modes. This will give a user experience similar to a WebSDR. The roadmap is not limited to this. Decoding of some digital signals and transmission of analog and digital modulations are candidate features. The longer term goal would be to have a complete transceiver that can be used in the field with only the ADALM Pluto, a smartphone, and perhaps a small power amplifier and battery.

References

[1] Maia SDR, `https://maia-sdr.org`

[2] B. Hilburn et al., "SigMF: the signal metadata format," Proceedings of the GNU Radio Conference, [S.l.], v. 3, n. 1, sep. 2018.

[3] Amaranth, `https://github.com/amaranth-lang/amaranth`

[4] S. Klabnik and C. Nichols, The Rust programming language. No Starch Press, 2023.

[5] N. Gallone, G. Goavec-Merou, J.-M Friedt, "Free, opensource Field Programmable Gate Array (FPGA) development frameworks for radio frequency communication – Digital communication using GNU Radio", European GNU Radio days / Software Defined Radio Academy 2022 tutorials.

[6] K. Shila, "An Amaranth-based Packetizer for the CASPER Toolflow", `https://blog.kiranshila.com/blog/casper_amaranth.md`

[7] D. Estévez, "Amaranth in Practice: a case study with Maia SDR", Open Research Institute Inner Circle newsletter, April 2023, `https://mailchi.mp/db8a4ece023c/31zpq7fkuj-15084741`

[8] S. He, M. Torkelson, "A new approach to pipeline FFT processor", Proceedings of IPPS '96, 1996.

[9] B. Blössl, "FutureSDR: An async SDR runtime for heterogeneous architectures", Software Defined Radio Academy 2022.

[10] D. S. Miller, R. Henderson, J. Jelinek, "Dynamic DMA mapping guide", `https://docs.kernel.org/core-api/dma-api-howto.html`