FPGA Partial Reconfiguration in Software Defined Radio Devices

Sara Grassi SARA.GRASSI@HEIG-VD.CH
Anthony Convers ANTHONY.CONVERS@HEIG-VD.CH
Alberto Dassatti ALBERTO.DASSATTI@HEIG-VD.CH
REDS, School of Management and Engineering Vaud, HES-SO // University of Applied Sciences Western Switzerland, CH-1401 Yverdon-les-Bains, Switzerland

Abstract

Many SDR systems make effective use of FPGAs for data acquisition and heavy lifting DSP processing. This has resulted in several dedicated frameworks being developed, RFNoC being the most renowned. Even though FPGAs fabrics are, by their nature, reconfigurable, SDR systems often fail in exploiting this interesting opportunity at run-time. In this paper, we show how it is possible to make effective use of the Partial Reconfiguration capabilities of modern FPGA devices, extending the range of applications RFNoC can be applied to. In particular, it allows the live reconfiguration of signal processing chains, for instance to switch between wireless standards. This results in a better use of the limited FPGA resources by time-sharing them between processing blocks. Unfortunately, support for Partial Reconfiguration is not yet available in the software stack of commercially-available SDR devices. Our work thus aims at encouraging its integration.

1. Introduction

Most of the Software Defined Radio (SDR) devices currently available on the market have an FPGA between the General-Purpose Processor (GPP) and the RF module. For instance, this is the case for all the SDR devices from Ettus Research (N. Pandeya, N. Temple, 2016b) generally known as USRPs (Universal Software Radio Peripherals). The FPGA of these SDR platforms is, out of the box, only used to configure the RF module and to handle data transfers and rate conversion, while all the base-band signal processing is done on the GPP.

SDR frameworks, such as GNU Radio (gnu), simplify the development of signal processing applications on the GPP in a highly modular, flowgraph-oriented fashion. Some of these signal processing operations, such as FIR filters and FFT, require high computational power and would be perfectly suited for deployment in the FPGA. FPGA development is, unfortunately, difficult and time-consuming, so frameworks like the RFNoC (Radio Frequency Network on Chip) (M. Braun, J. Pendlum, M. Ettus, 2016) are used to foster FPGA adoption in USRP devices. The RFNoC framework takes care of all the tasks that are not directly related to the DSP processing, such as clocking and data transfers. Thus significantly eases the integration of custom signal processing components in a modular fashion, and can be used alone or seamlessly integrated within GNU Radio flow graphs. From GNU Radio it is then possible to decide which blocks of the graph will run on the GPP and which blocks will be executed on the FPGA (although with some limitations).

Modern FPGAs are configured by loading a configuration file, the bitstream, into their configuration memory. Partial Reconfiguration (PR) is the modification of one or more portions of the FPGA logic while keeping the remaining portions unaltered (Xilinx, 2017c). If Partial Reconfiguration occurs while the FPGA is in running state, is it called dynamic PR. Dynamic PR saves area and power by time-multiplexing hardware resources that are mutually exclusive, and reduces the reconfiguration time compared with a full configuration. However, PR is not without shortcomings: PR works on die regions of fixed size, hence PR tools reserve an area big enough to contain the largest block that has to be mapped. When blocks whose size is smaller than the partition are instantiated, FPGA resources are wasted: a compromise should thus be found. Furthermore, FPGA tools are notoriously complex and PR adds more steps to the bitstream generation process.

Partial Reconfiguration is particularly useful in an SDR application requiring, for instance, the reconfiguration of one signal processing chain while maintaining the communication link (T. Kazaz, C. Van Praet, M. Kulin, P. Willemen, I. Moerman, 2017). However, PR is not yet exploited in SDR devices, even though most of them contain a PR-capable FPGA (M. Braun, J. Pendlum, 2017). The objective of our

Proceedings of the 10^{th} GNU Radio Conference, Copyright 2020 by the author(s).

work is to demonstrate the feasibility of Partial Reconfiguration in SDR devices, thus encouraging the developers to integrate support for this feature. To demonstrate our ideas, we have used the RFNoC framework on a USRP E310 device (Ettus Research, a) together with the UHD driver and its user-space API (Ettus Research, c) to control the device. In Section 2 we explain the structure of RFNoC blocks and how we could apply Partial Reconfiguration to them. In Section 3 we show how to do a Partial Reconfiguration design on the E310 platform, using a simple RFNoC custom block, and how to test it using UHD utilities and a C++ application. In Section 4 we apply Partial Reconfiguration design to RFNoC library blocks, and we test this PR design using UHD utilities and GNU Radio. Section 5 draws conclusions and future work ideas.

2. RFNoC blocks and Partial Reconfiguration

RFNoC (M. Braun, J. Pendlum, M. Ettus, 2016) is an open source framework used to simplify the deployment of signal processing algorithms in the FPGA of Ettus Research SDR devices. An RFNoC design consists of a network of blocks connected via a crossbar that routes streams of data packed in accordance with the compressed header (CHDR) format. Figure 1 shows the internal structure of an RFNoC block, which consists of two main parts: the NoC-Shell, which is the same for all the blocks, and the actual user IP, which implements the signal processing algorithm, and is connected to the Noc-Shell via an AXI-Stream interface (Xilinx, 2017a). Each RFNoC block exposes the same interface, also AXI-stream based, to the crossbar.

In an RFNoC design blocks can be either available in the UHD library (Ettus Research, b) or custom. Custom blocks are created using the *rfnocmodtool* (M. Braun, N. Cuervo, 2016) command which generates an empty block in which the user can instantiate her own IP. Several RFNoC blocks can be combined in an FPGA configuration bitstream using the *uhd_image_builder* command. Generating the FPGA bitstream fixes the number and type of available blocks (M. Braun, J. Pendlum, 2017). Though by using the UHD API, the crossbar connections between these blocks can be modified arbitrarily at run-time. However, there is no provision to add new blocks or swap them out at run-time. Furthermore, the number of RFNoC blocks that we can instantiate in a bitstream is quickly limited by the FPGA available resources (see Section 4). Even though having multiple FPGA bitstreams ready for loading and re-program the entire bitstream at run-time is possible, this is slow and the FPGA must be stopped, interfering with the current operation of the SDR device. So adding this dynamic dimension to RFNoC is the main contribution of our work.

Indeed, Partial Reconfiguration (Xilinx, 2017c) can be used to circumvent these shortcomings, allowing the modification of an operating FPGA without stopping it completely. After configuring the FPGA with a full (static) bitstream, a partial bitstream (.bit file), is used to modify one or more PR partitions in the FPGA, as shown in Figure 2. This can be achieved without modifying the rest of the FPGA configuration. One of the main limitations of Partial Reconfiguration usage in traditional HDL designs is that the reconfigurable block should have exactly the same interface signals connected to the static design when we replace one reconfigurable module by another. However in our case, as all the RFNoC blocks expose the same interface to the crossbar (Figure 1) they can be well combined with Partial Reconfiguration. From a practical stance, we instantiate an RFNoC block in the full design, defining it as a reconfigurable module associated with a PR partition and we generate the full (static) bitstream. We then add the other RFNoC blocks to the design, as reconfigurable modules associated with the same PR partition, and generate a different partial bitstream for each of the alternative RFNoC blocks. These partial bitstreams can be loaded at run-time, allowing the switching among RFNoC blocks.

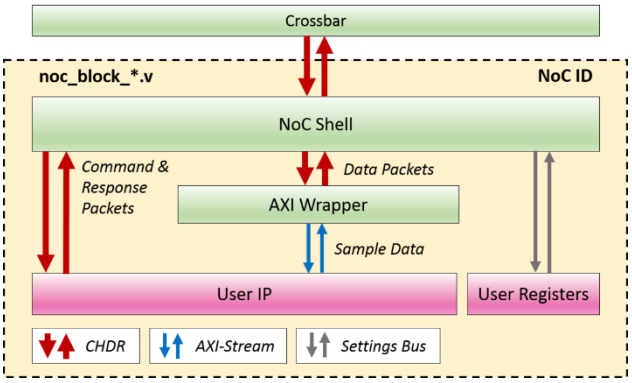

Figure 1. Internal structure of an RFNoC block, from (M. Braun, N. Cuervo, 2016).

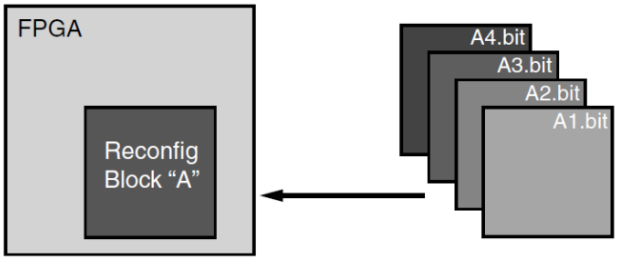

Figure 2. Partial Reconfiguration procedure, from (Xilinx, 2017c).

3. Partial Reconfiguration on the USRP E310 SDR Platform

As demonstration platform, we chose to use a USRP E310 (Ettus Research, a). This SDR platform is based on a Xilinx Zynq 7020 chip, which contains a dual-core ARM9 processor running an embedded Linux distribution and an Artix-7 FPGA, connected using a standard AXI interface. The Zynq 7020 device is particularly suitable for Partial Reconfiguration in an SDR application, because the complex signal processing tasks can be done in the FPGA while the ARM processor is controlling the overall system, deciding when to perform the Partial Reconfiguration, and performing it by delivering the partial bitstream to the FPGA.

We have proved the feasibility of Partial Reconfiguration on the USRP E310 SDR platform using a simple RFNoC custom block, and tested it using UHD utilities and a C++ application. This work was carried out under the SDR Makerspace initiative (Libre-Space Foundation), funded by the European Space Agency (ESA). The project Wiki (A. Convers, S. Grassi, 2020) contains a step-by-step procedure that can be used for replicating this work, and that we summarize below.

We installed the necessary tools to do RFNoC development, on both the host PC, running Ubuntu 18.04, and the target E310 platform, as explained in (Ettus Research, 2017). These tools are: (1) UHD 3.14, driver and API for application development on USRP SDR platforms. (2) GNU Radio: a framework to implement signal processing in software radios. It contains *gr-uhd*, enabling control and data transfer with the E310. (3) *gr-ettus*: out-of-tree module that extends *gr-uhd* adding support for RFNoC within GNU Radio.

Using *rfnocmodtool* we created two custom RFNoC blocks *Gain1* and *Gain2*, which multiply the input signal by a different gain factor, 1 and 2, respectively. Then, using *uhd_image_builder*, we created a bitstream that contains the custom block *Gain1*, and we used the option $-g$ to open the Vivado GUI (Xilinx, a) during the FPGA building process. We saved this design as a Vivado project. Then, we added to the project the custom block *Gain2*, and we applied Partial Reconfiguration [1] generating static and partial bitstreams for the *Gain1* and *Gain2* modules. We tested these bitstreams using UHD utilities and a C++ application (see Section 3.3). The first tests failed. Only after finding a means to freeze the outputs of the RFNoC block while we are reconfiguring it (see Section 3.1) we tested our design successfully. Finally, we added support for PR from the ARM processor to the FPGA of the E310 (see Section 3.2).

[1] A good reference for this process is the Xilinx tutorial for Partial Reconfiguration (Xilinx, 2017b)

3.1. Freezing the output of the RFNoC block while writing the partial bitstream

To avoid glitches at the output during the Partial Reconfiguration of an RFNoC block, we added a *freeze* input signal to the design that disables all the outputs of an RFNoC block. In our example this signal is connected to a pin (GPIO54) that is controlled by the ARM processor of the Zynq 7020 device. From the embedded Linux, we can thus first export and initialize the pin, using the commands:

```
# echo 54 > /sys/class/gpio/export

# echo out > /sys/class/gpio/gpio54/
    direction
```

Then, we can freeze and unfreeze the output of the RFNoC block by using, respectively, the commands:

```
# echo 1 > /sys/class/gpio/gpio54/value

# echo 0 > /sys/class/gpio/gpio54/value
```

Other approaches may be possible, but this approach suits our demonstration purposes.

Figure 3. Writing the bitstreams to the FPGA of the E310 using a JTAG connector.

3.2. Writing the bitstream

At first we used a JTAG probe to configure the FPGA with static and partial bitstreams (see Figure 3). This approach requires that the user opens the SDR case and buys, or builds, a non-standard cable. We thus found a means to write the bitstreams using the FPGA manager drivers in Linux (version *3.14.2-xilinx*). We first freeze the output of the RFNoC block (see Section 3.1). Then, we write the PR bitstream using the UHD utility *uhd_image_loader*. Finally,

we unfreeze the output of the RFNoC block. As we are using a partial bitstream, we should inform the FPGA driver of the embedded Linux distribution before writing the bitstream with *uhd_image_loader*. We can enable this option by using the command:

```
# echo 1 > /sys/devices/amba.5/f8007000.ps7
    -dev-cfg/is_partial_bitstream
```

We found out that, depending on the Linux kernel version, there are some limitations in the driver used to manage the FPGA. With UHD 3.14 and Linux *3.14.2-xilinx*, both the *xilinx_devcfg* driver and the device file */sys/devices/amba.5/f8007000.ps7-dev-cfg/* are available. We could send the full and partial bitstream through the processor as explained above. Unfortunately, with UHD 3.15 there is a newer Linux kernel, *4.18.33.yocto.standard*, that has a new FPGA manager driver that does not support the writing of partial bitstreams yet (Xilinx, b). We hope that this support will be added in a near future.

3.3. Testing the Partial Reconfiguration design

We tested the Partial Reconfiguration design using the UHD utility program *uhd_usrp_probe* to list the RFNoC blocks that were available in the FPGA, with the option *--args=no_reload_fpga* to avoid overwriting the FPGA configuration with the default bitstream. For example, we used *uhd_image_loader* to load the static bitstream containing the *Gain1* block, we then used *uhd_usrp_probe* to list the blocks and verified that the *Gain1* block was present in the FPGA. We loaded finally the partial bitstream of the *Gain2* block and listed again the blocks, observing that the *Gain2* block was present, and the *Gain1* block was not present anymore.

We wrote a C++ program using the UHD API (N. Pandeya, N. Temple, 2016a), derived from the example code *rfnoc_rx_to_file.cpp*, which receives radio data, passes it through an RFNoC block, and streams the results to a file. We modified this program to add PR. The program loads a static bitstream (given with the input option *--fpga-path*) and starts streaming data through the RFNoC block (*Gain1* in our test). Data received are stored in a file. Then the first partial bitstream (input parameter *--fpga-path-pr1*, containing *Gain2*) is loaded and output data recorded in a second file. Finally, the second partial bitstream (input parameter *--fpga path-pr2*, again *Gain1* here) is loaded and data streamed to a third file. The content of the three output files is plotted in Figure 4. We observe that after loading the partial bitstream for *Gain2*, the magnitude of the complex I-Q data doubles. After loading the partial bitstream for *Gain1*, the magnitude of the data halves. These results validate the right functioning of the PR RFNoC blocks.

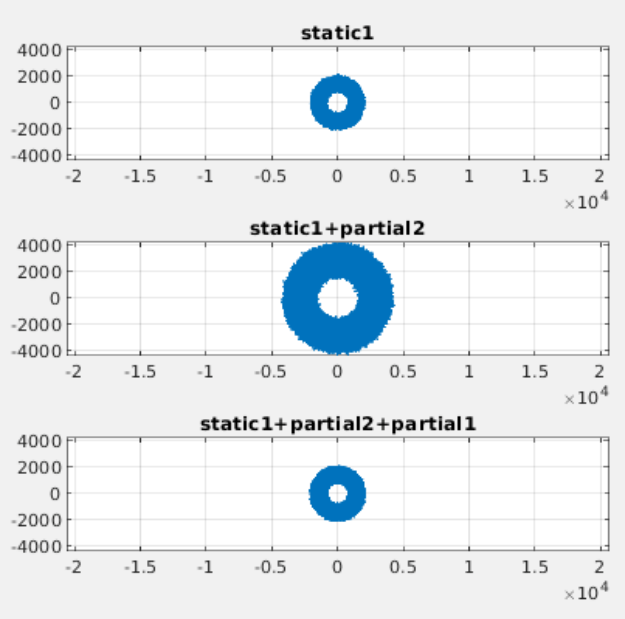

Figure 4. Testing the Partial Reconfiguration design with RFNoC Gain blocks in run-time. Gain1 block in the upper figure, Gain2 in the middle figure, Gain1 in the lower figure.

4. Partial Reconfiguration of RFNoC blocks from the UHD library

In the previous section, we have shown that FPGA PR is possible on the E310 platform, using very simple RFNoC custom blocks. Here we demonstrate PR on more complex signal processing blocks, using five RFNoC blocks found in the library delivered with UHD: DDC, FFT, FIFO, FIR, and SigGen.

Using *uhd_image_loader* we created four different FPGA bitstreams, each with different combinations of the five RFNoC blocks, as shown in Table 1. We also report the resource utilization for each bitstream. We observe that the number of RFNoC blocks that we can instantiate in a bitstream is quickly limited by the available resources in the FPGA. For instance, we cannot generate the static bitstream containing the four blocks (DDC + FFT + FIR + FIFO) because it exceeds the resources of the Zynq 7020 device. This could be circumvented using PR if the blocks can be time-multiplexed.

We did a PR design, as explained in Section 3, generating static and partial bitstreams for three blocks: FFT, FIR and SigGen. We tested the PR design using the UHD utilities *uhd_image_loader* and *uhd_usrp_probe* (see Section 3.3). Loading first the static bitstream and then loading in turn PR bitstream for the FIR, the FFT, and the SigGen blocks, and observing that they are mutually exclusive (see Fig-

	DDC + FFT + FIFO	DDC + FIR + FIFO	DDC + SigGen + FIFO	DDC + FFT + FIR + FIFO	Available in Zynq 7020
LUT	41732 (78.4%)	38914 (73.2%)	39711 (74.6%)	55883 (105%)	53200
LUTRAM	10583 (60.8%)	9605 (55.2%)	9440 (54.3%)	Implementation failed to complete	17400
FF	60574 (56.9%)	55941 (52.6%)	56192 (52.8%)		106400
BRAM	69 (48.9%)	59 (42.1%)	59 (42.1%)		140
DSP	126 (57.3%)	156 (70.9%)	76 (34.6%)		220

Table 1. Resource utilization for the four different FPGA bitstreams, containing different combinations of five RFNoC blocks from the UHD library: DDC, FIFO, FFT, FIR, and SigGen.

ure 5 and Figure 6).

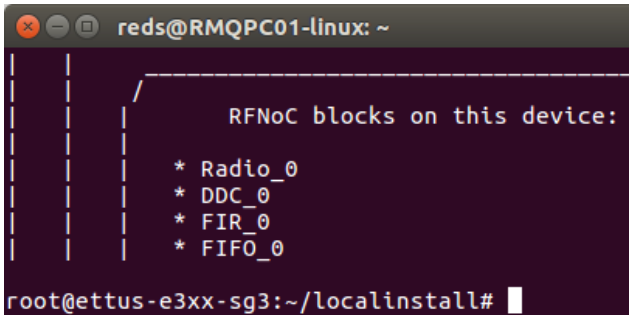

Figure 5. List of RFNoC blocks on E310 board with PR FIR block.

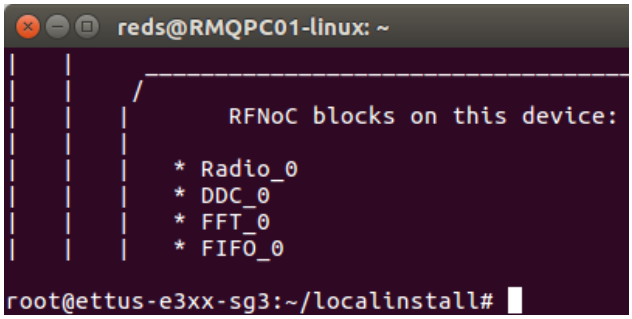

Figure 6. List of RFNoC blocks on E310 board with PR FFT block.

To demonstrate the fact that other logic in the FPGA continues to work without disruption during the dynamic PR swap, we ran another test. With UHD API we configured and enabled the RFNoC radio block to produce a signal on the output, and we visualized it on an oscilloscope. While the design was running, we loaded a PR bitstream: no interruption has been observed on the generated signal.

During our tests on the E310 platform, we measured the time elapsed to load a static bitstream and a PR bitstream. A full configuration, from the ARM processor, last around 143 ms; only 33 ms are required for a PR bitstream. This means that in our case, PR is four time faster than a full configuration.

To test the PR design with GNU radio we built, using GNU radio companion, a flow graph containing the FFT block (Figure 7) and a flow graph containing the SigGen block (Figure 8). Before running the first flow graph, on the target E310, we loaded the static bitstream containing the FIR, followed by the partial bitstream with the FFT. Similarly, before running the flow graph for the SigGen, we loaded the static bitstream, followed by the SigGen partial bitstream. In all cases, we ran the GNU radio flow graphs and verified that the output corresponded to the expected processing. This step validates the use of PR with real-world RFNoC blocks.

5. Conclusion and Future Work

We have shown that FPGA Partial Reconfiguration (PR) is feasible in SDR devices, and we have set up a procedure to do PR on the USRP E310 SDR device. Using UHD to control the device, and the RFNoC framework to deploy the signal processing components on the FPGA, PR integrates nicely with existing design methodologies. We have used simple custom RFNoC blocks to produce a first PR design, and we have then applied PR to more complex RFNoC blocks from the UHD library, confirming the generality of the approach. Finally, we have successfully tested these PR designs using: (1) UHD utilities, (2) a C++ program directly using UHD, and (3) GNU Radio flow graphs.

Partial Reconfiguration is particularly useful in an SDR application requiring, for instance, the reconfiguration of the signal processing chains or swapping wireless standards at run-time. It also allows a better use of the limited FPGA resources by time-sharing hardware resources between processing blocks that are mutually exclusive, avoiding the in-

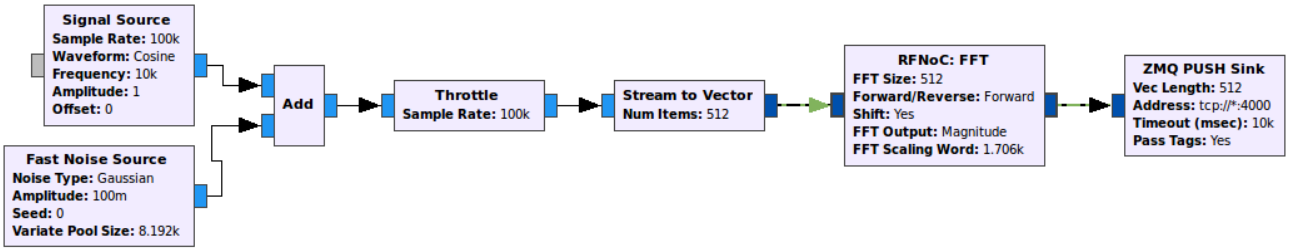

Figure 7. GNU Radio graph for the FFT block, target side.

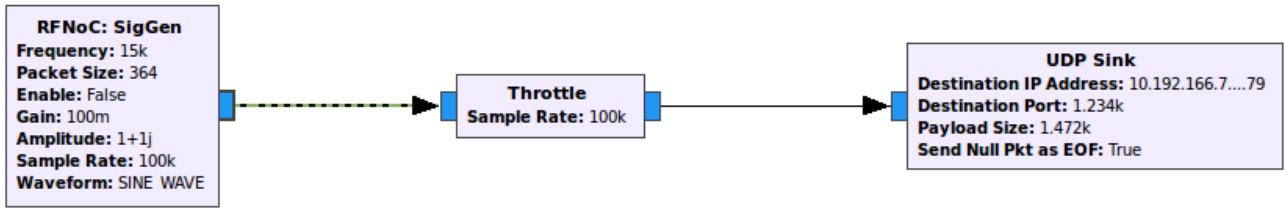

Figure 8. GNU Radio graph for the SigGen block, target side.

crease in costs and power consumption that a bigger FPGA would entail.

However, support for Partial Configuration is not mainstream yet. For instance, it is unavailable in the software of the currently available commercial SDR platforms, such as the USRP devices from Ettus Research, even though they contain an FPGA that support this feature. The objective of our work was to encourage the integration of PR support into the software of these platforms. For instance, the support for writing partial bitstreams could be integrated in UHD, and the freezing of the output of an RFNoC block could be integrated in the controls of the NoC-Shell.

Acknowledgements

This work was partly funded by the European Space Agency (ESA) through the activity *SDR's FPGA Partial reconfiguration of the SDR-Makerspace initiative*, https://sdrmaker.space.

References

GNU Radio Project. URL https://www.gnuradio.org.

A. Convers, S. Grassi. *SDR's FPGA Partial reconfiguration - wiki*, 2020. URL https://gitlab.com/librespacefoundation/sdrmakerspace/sdr-pr/-/wikis/home.

Ettus Research. USRP E310, a. URL https://www.ettus.com/all-products/e310/.

Ettus Research. USRP Hardware Driver Repository, b. URL https://github.com/EttusResearch/fpga/tree/UHD-3.14/usrp3/lib/rfnoc.

Ettus Research. UHD (USRP Hardware Driver), c. URL https://www.ettus.com/sdr-software/uhd-usrp-hardware-driver/.

Ettus Research. *Software Development on the E3xx USRP - Building RFNoC UHD / GNU Radio / gr-ettus from Source (AN-315)*, 2017. URL https://kb.ettus.com/Software_Development_on_the_E3xx_USRP_-_Building_RFNoC_UHD_/_GNU_Radio_/_gr-ettus_from_Source.

Libre-Space Foundation. SDR Makerspace initiative. URL https://sdrmaker.space/.

M. Braun, J. Pendlum. Flexible Data Processing Framework for Heterogeneous Processing Environments: RF Network-on-Chip. In *FPGA4GPC Conference*, 2017.

M. Braun, J. Pendlum, M. Ettus. RFNoC: RF Network-on-Chip. In *GNU Radio Conference*, 2016.

M. Braun, N. Cuervo. *Getting Started with RFNoC Development AN-823*, 2016. URL https://kb.ettus.com/Getting_Started_with_RFNoC_Development.

N. Pandeya, N. Temple. *Getting Started with UHD and C++*, 2016a. URL https://kb.ettus.com/Getting_Started_with_UHD_and_C++.

N. Pandeya, N. Temple. *Selecting a USRP Device*, 2016b. URL https://kb.ettus.com/Selecting_a_USRP_Device.

T. Kazaz, C. Van Praet, M. Kulin, P. Willemen, I. Moerman. Hardware Accelerated SDR Platform for Adaptive Air Interfaces. In *ETSI Workshop on Future Radio Technologies*, 2017.

Xilinx. Vivado Design Suite Tools, a. URL https://www.xilinx.com/products/design-tools/vivado.html.

Xilinx. *Solution Zynq PL Programming With FPGA Manager*, b. URL https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841645/Solution+Zynq+PL+Programming+With+FPGA+Manager.

Xilinx. *Vivado Design Suite, Vivado AXI Reference Guide UG1037 (v4.0)*, 2017a. URL https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

Xilinx. *Vivado Design Suite Tuorial, Partial Reconfiguration UG947 (v2017.4)*, 2017b. URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug947-vivado-partial-reconfiguration-tutorial.pdf.

Xilinx. *Vivado Design Suite User Guide, Partial Reconfiguration UG909 (v2017.4)*, 2017c. URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug909-vivado-partial-reconfiguration.pdf.