RFNoC™ — RF Network-on-Chip

Martin Braun, Jonathon Pendlum, and Matt Ettus
{MARTIN.BRAUN, JONATHON.PENDLUM, MATT}@ETTUS.COM
Ettus Research / National Instruments
4600 Patrick Henry Dr, Santa Clara, CA 95054 USA

Abstract

RFNoC™ (RF Network-on-chip) is an open source framework for developing data processing applications which can run on the FPGA as well as a host computer. Similar to GNU Radio, which facilitates the development of signal processing applications, RFNoC boosts developers' productivity by handling the majority of non-processing related tasks which are necessary for such a framework, such as data movement, exposing settings APIs, routing, flow control etc. RFNoC works standalone, but is particularly useful in combination with GNU Radio, in which it is well integrated. It has been developed for USRP devices, and is part of the USRP Hardware Driver (UHD).

1. Introduction

What if your software-defined radio application could run components on FPGAs just as easily as it can run them on a GPP? With RFNoC (RF Network-on-Chip), that's what we set out to do. Using RFNoC, it's simple to distribute processing components on various hardware platforms (GPP or FPGA), depending on what is most suitable for your application. This turns an SDR device such as a USRP X310 into more than just an adapter to RF frequencies for software, but makes it a platform for arbitrary processing chains.

As an example, consider a modem written in an SDR framework such as GNU Radio. Parts of the modem's PHY might include signal processing components such as FIR filter or equalizers, which are great algorithms to deploy on FPGAs, but can tie up a lot of processing power when run on a GPP. Many SDR devices will allow modifying the FPGA to do this processing, but typically, that means those components leave the modular configuration of the GNU Radio application. With RFNoC, it does not matter where signal processing components are deployed, applications stay modular and composable.

In Section 2, we lay out what motivated us to develop RFNoC. Section 3 lays out the architecture of RFNoC and explains which components interact to make RFNoC possible. The development process for using and extending the framework is described in Section 4. We conclude with Section 6.

2. Motivation

Consider the USRP N210: For a long time, this was the most popular Ettus Research USRP device, and it has been deployed in all kinds of scenarios and use cases. UHD made it very easy to use it in any kind of software, since it abstracted away all the low-level details going on inside the device. If the USRP was instructed to receive on a certain center frequency, UHD would internally figure out how to apply those settings, automatically and transparently converting the frequency into a series of hardware commands, which would in turn configure synthesizers or other components until the device was actually receiving on the desired center frequency.

While this made software development for UHD-controlled devices very simple, it also hid away the signal processing chains internal to the device and made it difficult to add functionality to the FPGA. UHD did support inserting custom logic into the FPGA, but it could only be placed into a certain spot in the signal processing chain, and there was no way to dynamically enable or disable custom modifications.

With the release of the third generation of USRPs (specifically, the E310 and the X-Series), it became obvious that were was a dire need to make FPGA development for USRPs easier. The USRP X310 and X300 devices ship with a fairly large FPGA (a Kintex 7 by Xilinx), and it would be wasteful to not use that FPGA. The USRP E310 on the other hand, as an embedded device, simply does not have the computational power to run complex signal processing software on its CPU (a dual-core ARM Cortex A9), and thus must be able to simply off-load processing onto the

Proceedings of the 6th GNU Radio Conference, Copyright 2016 by the author(s).

FPGA (the E310 ships with a Zynq processor, combining FPGA and CPU on the same die).

To address these issues, RFNoC was created as a novel way to control and program the FPGAs on these devices. Unlike before, it not only gives more fine-grained control over components on the FPGA, but it also allows to insert user-defined, custom-built modules into the FPGA and control them through generic interfaces.

Developing for FPGAs can be a daunting task for beginners, but even experienced FPGA developers will testify that a lot of time spent developing doesn't necessarily go into the target algorithms, but also into housekeeping tasks such as setting up clocking, configuring transports, etc. A primary goal of RFNoC was to take away all the development work unrelated to implementing algorithms, and maximize the time spent on the actual problem at hand. This includes both the FPGA side (i.e., as little code as possible should be required to connect up custom IP) as well as the software side (no heavy host-side software development should be necessary to enable or configure custom IP).

Even without loading custom FPGA IP, RFNoC can be a useful tool, by using the IP that ships with RFNoC. An example is the implementation of Welch's spectral analysis algorithm (Stoica, Petre G. and Moses, Randolph, 2005) using GNU Radio shown in Fig. 1: While correct in theory, it won't be particularly useful, for multiple reasons: First, data needs to be streamed from the device at the full rate, which means the maximum bandwidth of the spectral analysis is limited by the transport. The actual signal processing is simple: A window function, an FFT, a magnitude-square operation and an averaging function. The averaging is followed by a decimation, which reduces the data rate.

A more efficient implementation would be to move the signal processing blocks onto the *FPGA* instead of running them on the GPP. Since the final data rate is heavily reduced, even a slow link between the FPGA and host computer would allow to display a high-bandwidth spectrum, without being a heavy burden on the CPU.

With RFNoC, this is not only possible to do easily, but it also maintains the modularity of the original spectral estimation application. In fact, the final GNU Radio application looks mostly like the original, host-based flow graph — and yet, signal processing components are seamlessly moved from the CPU to the FPGA (cf. Section 5.1).

3. Architectural Overview

At its core, RFNoC is a method to modularize signal or data processing components on an FPGA and efficiently access them. Processing is split into *blocks* (or *computation engines*), and data is passed between blocks. These blocks

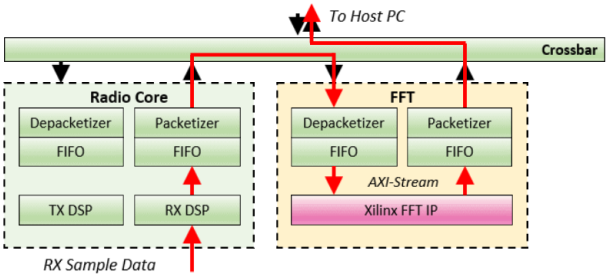

Figure 2. Data flow for a simple FFT processing application

are connected to a crossbar, which allows arbitrary routing of packets between any block. RFNoC takes care of passing data between blocks, routing, and configuring the blocks from software. Data can also be passed back and forth between RFNoC blocks and software.

A block can be anything from a simple DSP algorithm such as FFTs or FIR filters to more complex algorithms or even full packet demodulators. The question of how much functionality to put into a single block depends on many factors; having more fine-grained blocks allows for more flexibility, but having more integrated, monolithic blocks may lead to more efficient implementations overall. Typically, a block will perform one kind of DSP algorithm, e.g., an FIR filter would be a single block instead of multiple adders and multipliers, but a full demodulator might split up tasks such as forward error correction, equalization, synchronization etc. into multiple blocks.

Any block in an RFNoC configuration can be configured to communicate with any other blocks. There are two basic types of communication: Command/control, and data. Fig. 2 shows how data from the radio can be routed to an FFT block before going back to the host PC. The same figure also shows another aspect: All data between blocks is *packetized*. Since we're using packet data, it's an easy matter to route those packets not only to other blocks on the same crossbar, but by using Ethernet interfaces, we can send those packets to any device that's on the same network.

Internally, all blocks consist of a common framework interface called the *Noc-Shell*. This allows connecting any kind of AXI-Stream compliant IP into an RFNoC network. Noc-Shell takes care of packetization and depacketization, routing, flow control, and any kind of settings that are common between blocks.

Common settings are mostly for housekeeping tasks, and the user does not need to care about those in most cases (although all settings are made available through software APIs). The other responsibilities make sure that data is correctly passed between blocks. By setting appropriate rout-

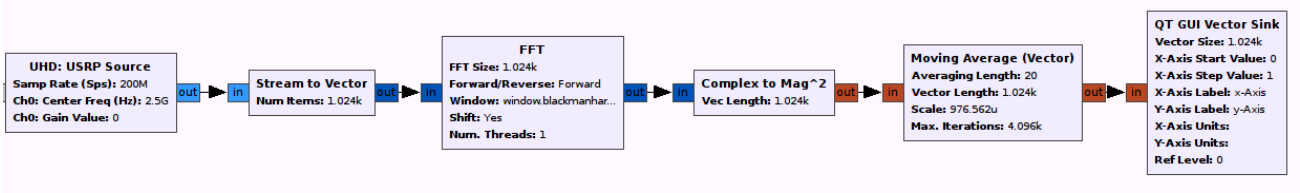

Figure 1. A simple spectrum visualization application in GNU Radio using Welch's algorithm

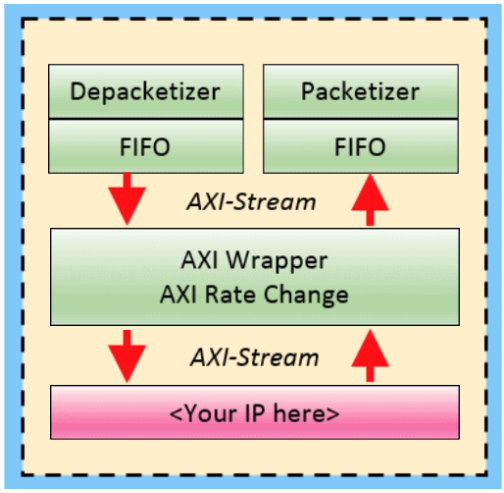

Figure 3. Components of Noc-Shell

ing information, Noc-Shell will correctly address all outgoing packets, so they can reach their intended destination. Flow control is a feature which guarantees that blocks can only send data to another block if it's in a state that allows receiving data. By putting this mechanism into Noc-Shell, custom IP doesn't need to manage when and where to send data.

3.1. Limitations

Using RFNoC makes using FPGAs almost as flexible and viable as using GNU Radio, but there are some limitations:

- The number of available blocks per application is limited by the blocks available on the FPGA, unlike a pure software implementation, where any number of blocks can be added at any time. This affects the total number of blocks per FPGA (which are limited by FPGA resources) and the currently available blocks (which must be chosen a-priori at build time).
- Synthesizing new FPGA images is time-consuming[1]. It is currently not feasible to modify the FPGA contents at runtime.

[1]There are multiple investigations and efforts on their way to improve the build time for FPGA images, and this is likely to improve significantly in future versions of RFNOC.

3.2. Comparison and relation to GNU Radio

GNU Radio and RFNoC have a lot of similarities: Both are frameworks to create signal processing applications, and both take care of a bulk of housekeeping tasks. Both frameworks allow the developer to focus entirely on developing processing algorithms, and provide means to connect with other blocks in the same framework. A typical GNU Radio development cycle starts off by connecting existing blocks provided by the GNU Radio framework (or third-party extension modules for GNU Radio, known as out-of-tree modules in the GNU Radio community). In many cases, existing blocks will get developers quite far before they need to extend GNU Radio with their own blocks. At this point, GNU Radio provides tools and tutorials to make the addition of blocks as simple and painless as possible. In GNU Radio, blocks are software components, typically written in either C++ or Python. Tools such `gr_modtool` help developers by making sure as little non-relevant code as possible needs to be written manually. Block authors have a variety of tools at their disposal for a smooth integration into the framework, and GNU Radio provides a unit testing framework to quickly develop tests for new blocks.

RFNoC is very similar, the biggest difference being that blocks are developed for FPGAs instead of in software. Like GNU Radio, there are tools available to aid in the development of blocks, such as a `gr_modtool`-based tool to create boilerplate code, a testbenching infrastructure to verify blocks' functionality (automatically or GUI-based during development). The framework provides all the code requirement to connect blocks, initiate data streaming and pass data from one algorithm to the next.

RFNoC and GNU Radio are separate projects, and RFNoC does not require GNU Radio work. However, the two frameworks dovetail well, and the large amount of blocks available in GNU Radio can be a real boon for developing applications in RFNoC. The integration of RFNoC into GNU Radio is very complete, and from within the GNU Radio Companion, it is very simple to seamlessly pass data from GNU Radio blocks to RFNoC blocks and vice versa. For RFNoC developers using GNU Radio, this makes things a lot easier, since now tools from both frame-

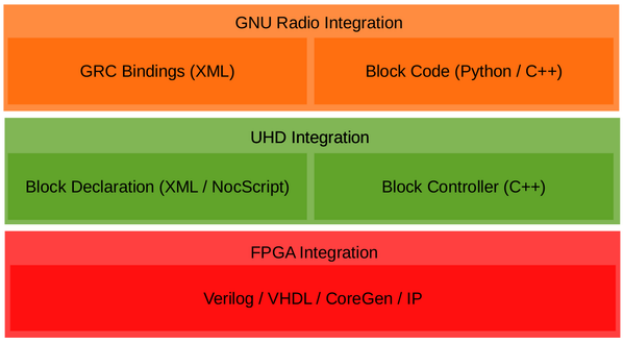

Figure 4. The RFNoC development stack

works are immediately available in the same GUI. For example, a developer might choose to develop a signal processing algorithm in software first (e.g., in Python) where development cycles are very fast. GNU Radio has a wide range of signal generation and visualization components which are excellent at testing new blocks. Then, when the developer chooses to move to FPGA development, the same test applications can be used from GNU Radio to test blocks on the FPGA. This way, data is passed through real hardware, but with no additional development overhead. Furthermore, it avoids having to test algorithms in combination with analog hardware, which can be an additional source of errors.

A tabular comparison of features is given in Table 1.

4. Development Process

Fig. 4 shows the components of an RFNoC block including its integration into GNU Radio. In a nutshell, there are three layers that require some development work:

1. The actual FPGA IP. This is where the vast majority of the development happens.
2. Block control. In order to control the block from software, some host code needs to be written.
3. GNU Radio integration (or integration into other frameworks). This may not be required, depending on the application.

RFNoC is designed such that development of anything non-FPGA-related is kept to a minimum. This means that even if RFNoC is only used as a development tool and not for deployment, the amount of overhead that needs to be invested exclusively for RFNoC is negligible.

These three layers of development are elaborated on in the rest of this section.

4.1. FPGA Development

There is no single required way to develop the actual IP that runs inside an RFNoC block. Developers can write straight up Verilog or VHDL, integrate IP from Xilinx or other vendors, or use other FPGA development tools. The only requirement is that the logic uses AXI-stream for its inputs and outputs.

See Fig. 3, which shows the internals of an RFNoC block. Several converters connect the AXI stream-compliant IP to the rest of the network. To configure the connection, several things need to be defined:

- Input and output data streams
- Settings and readback registers

As an example, consider the FFT block. It has one input stream, which is data pre-FFT, and one output stream, which is data post-FFT. There are at least three settings that need to be exposed: The FFT size, the FFT direction (inverse vs. forward FFT), and the output format (complex, magnitude, or magnitude-squared).

4.2. Block control

The *block controller* is the software component that represents the digital logic on the software side. It handles the following tasks:

- Declare input and output data streams
- Declare configurable properties
- Provide custom code to be run when properties change

Many of these responsibilities are purely declarative in nature, and don't require writing C++ code. These declarations are done through an XML file, which we call the *block descriptor file*.

In many cases, the block descriptor file is all that is required. If no custom block control code exists, a default block controller is instantiated which will parse the contents of the XML file and configure the system accordingly. If more specific actions on the host side are required, code needs to be written to handle this. Example: In the FFT block mentioned above, the FPGA fabric expects a control word which depends on the base-2 logarithm of the FFT size, the FFT direction and other miscellaneous settings. To know how to configure this control word, studying the Xilinx documentation for the FFT IP is required. This is not a desirable situation, so rather than having the user write the control word, we ask for the actual FFT size (not its logarithm) and the direction, and compute the control word in software before committing that setting.

Feature	GNU Radio	RFNoC
Handles data movement between blocks:	*Yes.* Data is passed via circular buffers.	*Yes.* Data is framed and passed via AXI-Stream interfaces.
Handles data routing:	*Yes.* Connections imply data paths.	*Yes.* Packets are transparently forwarded to blocks locally or over the network.
Provides Test Infrastructure:	*Yes.* Via cppunit and Python unittest.	*Yes.* Provides hooks for ModelSim and Xilinx xsim.
Provides development tools:	*Yes.* `gr_modtool`, and others.	*Yes.* Provides `rfnocmodtool`, a `gr_modtool` variant.

Table 1. Feature comparison between GNU Radio and RFNoC

If block control code is required, it can be written in C++, but we also provide a domain-specific language (DSL) called *Noc-Script* to enable some custom logic on the software side without having to write any C++. Noc-Script is also parsed and interpreted from the block description file at runtime, so no recompilation (or setting up of a C++ toolchain) is required to use Noc-Script, resulting in even less development overhead.

4.3. GNU Radio Integration

As mentioned before, GNU Radio is not required to run RFNoC. However, in many cases, users might want to pull RFNoC designs into GNU Radio, or other frameworks. In this case, some additional amount of work is required.

To make RFNoC available in GNU Radio, Ettus Research provides the `gr-ettus` out-of-tree module[2]. For most use cases, this provides all the code required to run RFNoC blocks, be it standard blocks provided by Ettus or custom blocks. Should authors of custom blocks want to run those blocks inside the GNU Radio Companion (GRC), they would simply have to provide a GRC bindings file.

More complex designs might require writing custom GNU Radio blocks (typically in C++). This is typically necessary when changing a setting on the block requires actions inside GNU Radio, that can't be handled by UHD. There are tutorials for both writing GRC bindings and GNU Radio blocks (GNU Radio).

4.4. RFNoC Modtool

In order to write an RFNoC block, including testbenches and the initialization of Noc-Shell, there is a certain amount of boilerplate code that needs to be instantiated. To minimize the time spent on tasks that are not related to the actual digital logic, Ettus Research provides a tool called `rfnocmodtool`, which is derived from GNU Radio's `gr_modtool`[3]. A full introduction to this tool is provided as a separate tutorial (Ettus Research).

This tool will instantiate, upon request, the following files:

- Skeleton code for the FPGA implementation (including an instantiation of Noc-Shell)
- A template for testbenches
- Templates for the block definition and block controller classes
- Templates for GNU Radio files (C++ block and GRC bindings)

The templates and skeleton files are marked up to make it easy to modify and to add own functionality.

5. Examples and Use Cases

5.1. Example applications provided by `gr-ettus`

By downloading and installing the `gr-ettus` out-of-tree module, a selection of examples are made available. One of these examples is shown in Fig. 5, which shows an implementation of Welch's algorithm (as discussed in Section 2). Data coming from the radio is passed through a series of blocks which perform the windowing, FFT, and averaging before sending a low-data-rate signal back to the host for plotting (in this example, the logarithm and modulus are calculated on the host computer, but could also be moved to the FPGA).

Connections between blocks are colour-coded to indicate the domain in which they are processed. Green arrows indicated on-chip connections, whereas black arrows indicate

[2] In the long term, the functionality provided by `gr-ettus` will be moved into `gr-uhd` and be provided by default GNU Radio.

[3] In future versions of GNU Radio, `gr_modtool` might be modified to work with 3rd-party plugins, in which case `rfnocmodtool` would be superseded by an extension to `gr_modtool`.

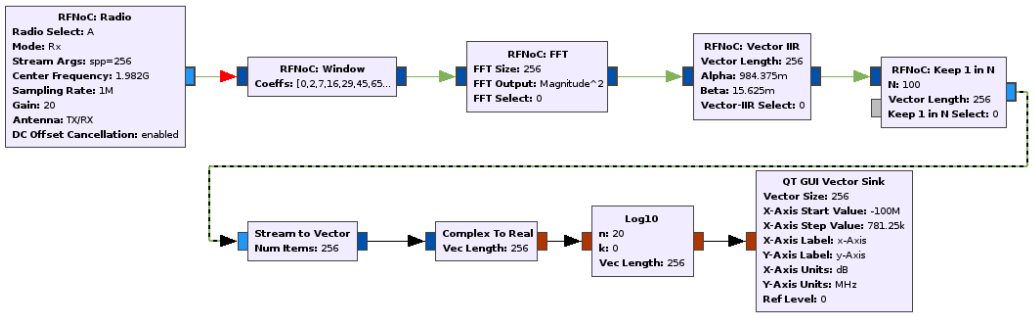

Figure 5. RFNoC example: Welch's algorithm, implemented on the FPGA

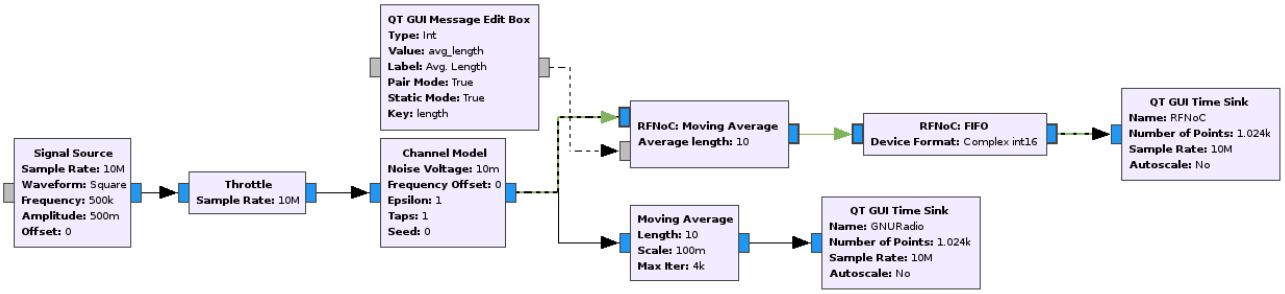

Figure 6. RFNoC example: Moving average in software and FPGA

connections handled by GNU Radio. Dashed lines indicate domain crossings, i.e., where data is passed from the RFNoC domain into the GNU Radio domain.

Fig.6 shows an example on how to test and develop using RFNoC. A signal is generated in GNU Radio, then sent through the same processing paths twice, once on the FPGA, and once within GNU Radio. Assuming that the GNU Radio implementation is functional, this is an easy way to verify if the FPGA implementation is producing the same output data. It is also a more comprehensive test than relying on testbenches, since this will actually require a full instantiation of the block in hardware, and will utilize all the components of a USRP. At the same time, it allows providing user-defined signals, instead of relying on analog signals coming from the radio, enabling reproducible tests.

5.2. Default UHD

Starting with UHD version 3.10, even default images for the X-Series devices actually use RFNoC. This is hidden away by some abstraction layers, but Fig. 7 shows a mock-up of the flow graph that gets instantiated under the hood when a regular session is instantiated (an X310 or X300 default FPGA image has a total of four receive and two transmit channels, which are not shown in this picture). On the transmit side, a large FIFO (using the DRAM on the X310 motherboard) is the first block to receive all the data from the host. From there, samples go to a digital upconverter (DUC), which performs sample rate conversion, and then to the radio block. The receive side is similar, using a digital downconverter, and no FIFO is required on the device for this data path.

5.3. Using RFNoC to test or validate existing IP

RFNoC is a viable framework for deployment, but it's also a powerful development tool and can be used purely for development purposes. As an example, consider a modem implemented fully in an FPGA, which needs to be verified. For verification, it would help to inject custom signals, as well as plot output signals, or even implement automated tests running on the hardware.

Since the overhead to pull existing designs into RFNoC is so small compared to the actual development, using RFNoC is even an option if there are no plans to use RFNoC in the final deployment. In the same manner, GNU Radio can be a useful development/debugging tool, even if it is not intended for use in final designs.

6. Conclusion and Preview

RFNoC is a very powerful extension of SDR tools such as GNU Radio, and can also be used stand-alone to build mod-

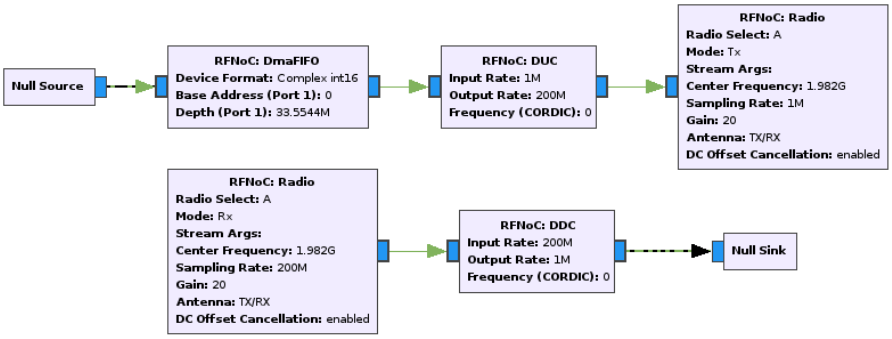

Figure 7. Mock-up of a default X310 configuration. Null sources and sinks are used as placeholders for the host side.

ular SDR applications running on FPGAs. In conjunction with other SDR frameworks such as GNU Radio we can get the best of both worlds (GPP-based and FPGA-based platforms) for better and faster development.

In the coming months, RFNoC will become more and more complete. Already, regular Ettus Research products are using RFNoC as drivers under the hood, and RFNoC will be the one and only architecture for future USRPs. For now, RFNoC is available as a separate feature branch available through the Ettus Research repositories, but will become part of regular UHD in the future.

References

Ettus Research. Getting Started with RFNoC Development, 2016. URL kb.ettus.com/Getting_Started_with_RFNoC_Development.

GNU Radio. GNU Radio Guided Tutorials, 2016. URL gnuradio.org/redmine/projects/gnuradio/wiki/Guided_Tutorials.

Stoica, Petre G. and Moses, Randolph. *Spectral analysis of signals*. Pearson Prentice Hall, Upper Saddle River, NJ, 2005. ISBN 0-13-113956-8.