Designing a RFNoC Block implementing a SISO Processor using High-Level Synthesis

Andrea Guerrieri ANDREA.GUERRIERI.IT@IEEE.ORG
Processor Architecture Laboratory, EPFL, Lausanne, Switzerland

Abstract

RFNoC (RF Network on Chip) is an open-source processing tool developed by Ettus Research™. SISO processor (Soft In Soft Out) is one of the basic component used in modern FEC (Forward Error Correcting) techniques such as Turbo Codes and LDPC Codes. This paper presents the development of an RFNoC block (in-progress) which implements a SISO Processor as an answer to *"The RFNoC & Vivado HLS Challenge"* sponsored by Ettus Research and Xilinx.

1. Introduction

Nowadays the computation abilities of FPGAs are becoming the most effective solution for complex DSP algorithms which are time expensive and power hungry if executed in software using traditional CPUs (Nurvitadhi, 2016). Neverthless, the design effort and complexity of FPGAs create a huge gap between the two design methods. In the years, several approach at different levels have been appointed to bridge this gap.
High-Level Synthesis is one answer at the EDA(Electronic Design Automation)-level: it is able to produce a hardware description starting from a software model, in general with minimum effort (Camposano, 1990).
At device level, a working example of this trend can be represented by RFNoC (Ettus, c). The goal of this framework is to allow the user developers to create FPGA applications using the same design flow for create standard application using GNU Radio Companion tool (GNURadioCompanion), moving the development effort to higher level of abstraction.
One example of high-complexity and CPU-cycle-expensive DSP processing is the Forward Error Correction (FEC) technique, widely used in telecommunication systems. The performance of modern digital communication systems is often restricted due to power limitation and the presence of Additive White Gaussian Noise (AWGN). The addition of FEC techniques to transmitted digital waveforms overcomes these limitations. Turbo codes (Berrou, 1993) is one of the FEC techniques used in the standards 3GPP mobile telecommunication system, such as UMTS, HSDPA, and LTE, but also in WiMax and DVB. The SISO processor is one of the building block of the Turbo Codes.

1.1. Motivation

This activity was started as answer to *"The RFNoC & Vivado HLS Challenge"* sponsored by Ettus Research and Xilinx. The original call was (Ettus, a) :

> *"This challenge rewards engineers for creating innovative and useful open-source RF Network on Chip (RFNoC) blocks that highlight the productivity and development advantage of Xilinx Vivado High-Level Synthesis (HLS) for FPGA programming using C, C++, or System C."*

This design has been submitted to the challenge in early December'16 and was one of the seven accepted proposal (Xilinx, 2017)

1.2. Contribution

Previous work has been done related to the integration of Turbo Codes in GNU Radio (Karra, 2012) and into USRP devices (Talasila, 2010). However, this design is the first attempt to create an RFNoC block implementing this type of DSP processing. This paper shows the RFNoC Block structure and its development environment, underling the advantages coming from the use of the High-Level Synthesis process to create digital hardware IPs. In detail, will be show the early development stage of an RFNoC block represented by the High-Level Synthesis process of the DSP algorithm. The mathematical model of the SISO processor, as well as the algorithm model are not the main focus of this paper. Neverthless, to provide a general overview of the block's complexity, we will report the mathematical equations that govern the information processing as well a practical hardware implementation structure. Furthermore, we will be briefly illustrate how High-Level Synthesis works with its internal operations and phases.

Proceedings of the 7^{th} GNU Radio Conference, Copyright 2017 by the author(s).

1.3. Paper Organization

The paper is organized as follows. Section 2 presents the RFNoC purposes and its internal structure; Section 3 provides an overview of High-Level Synthesis; Section 4 shows the mathematical model as well the practical structure of the SISO Processor. Section 5 shows the High-Level Synthesis of the original SISO Processor model, giving a panoramic on the possible solutions and further optimizations; Finally, Section 6 concludes the paper with final considerations and the future work.

2. RFNoC

RFNoC (Radio Frequency Network on Chip) is an open-source processing tool focused on the development of heterogeneous applications on SDR(Software Defined Radio) devices provided from Ettus Research^{TM}, generally knows as USRP (Universal Software Radio Peripheral).
The concept of NoC refers to a communication system integrated into a single chip used for exchanging data and control between the internal PEs (Processing Engines). The main difference with respect to a normal SoC (System on Chip) is the flexibility in terms of heterogeneous capabilities: in fact, the internal composition of PEs is abstracted from the communication framework.

2.1. RFNoC Blocks

RFNoC is internally composed of RFNoC blocks, the PEs implementing the DSP algorithms. To reduce the effort of integrating digital signal processing IPs as PEs into an RFNoC block, the design framework provides a pre-cooked interface wrapper. The internals of a RFNoC block are independent from any other block and can be designed with any language and tool that supports AXI stream interfaces, including VHDL, Verilog, and Xilinx Vivado HLS.

2.1.1. NoC Shell

We briefly report the structure of the RFNoC's interface, the NoC Shell. As explained previously, this wrapper represents the common part present in all blocks constituting the RFNoC, independently of the respective internal DSP algorithm. The role of the NoC Shell is to interface the internal PE with the rest of RFNoC, implementing a standard registers and command interface to allow it to be integrated within the RFNoC. It presents a user interface and an the interface to the RFNoC AXI stream crossbar. The latest expects a Compressed Header packets as defined in CHDR (Ettus, b). Figure 1 shows the structure of the RFNoC Block (Pendlum, 2014). To create this NoC Shell and all the file system structure, Ettus has deployed a tool called RFNoC Modtool. This tool creates a custom GNU Radio OOT (Out of Tree) module as well the necessary files for the RFNoC block development and simulation.

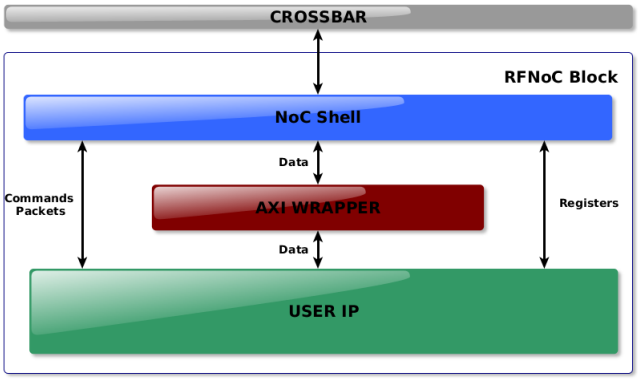

Figure 1. RFNoC Block Internal Structure (Ettus, c)

2.1.2. Simulation

As anticipated in the previous paragraph, RFNoC Modtool creates the NoC Shell as well the testbench environment(M. Braun, 2015). Figure 2 shows a testbench architecture created by the RFNoC Modtool.
Several advantages are coming from this verification architecture: first of all, it permits the verification the PE operation in the same environment it will be placed in when it is built into the RFNoC architecture, and it allows to test multiple blocks with multiple streams.

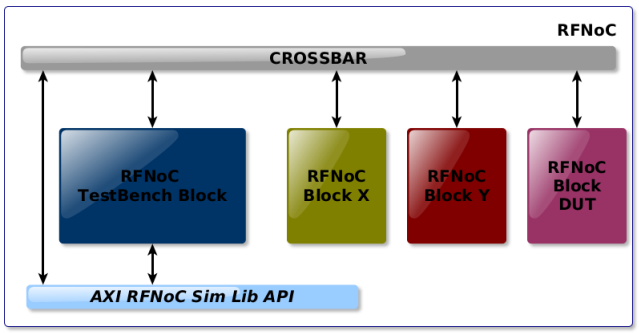

Figure 2. RFNoC Testbench Structure(Ettus, c)

2.2. GNU Radio Integration

The main application of USRP devices are inside the GNU Radio. The most efficient way to create a GNU Radio application is using the GNU Radio Companion (GRC), the GUI interface of the tool. The GNU Radio environment has been made under the concept of blocks, in line on what is done for RFNoC. With RFNoC the user can decide what will be executed in the host machine (PC) and what will be passed to the USRP (FPGA) simply by mapping the application as a GNU Radio or as a RFNoC block.

2.2.1. RFNoC Existing Blocks

Usually, the RFNoC blocks are very similar to the respective GNU Radio version, however the new blocks generated for the challenge will be added to the library of available open-source blocks to easily integrate into an RFNoC application. A brief list of available existing blocks is reported in (Ettus, c). The ultimate goal of this design is to add the SISO processor to this list. Once completed, the SISO-RFNoC block will be available at *https://github.com/Andrea-Guerrieri/SISO-RFNoC* under the MIT license.

3. High-Level Synthesis

3.1. Overview

In DSP algorithm design flows, the system architect models the algorithm in a high-level language such as C or C++, without worrying about implementation details. Historically, the RTL designer takes this algorithmic model and tries to hand-code it while respecting the expected performance/area tradeoffs. However, this manual coding is error sensitive and overall time consuming (Fingeroff, January 2010).
High-Level Synthesis (HLS), is a new step in the design evolution of a digital electronic circuit, which moves the design effort to higher abstraction levels, improving design productivity by automating the refinement from the algorithmic level to RTL (Takach, 2016).

3.1.1. Advantages

HLS generates Verilog/VHDL code from C/C++ and SystemC, providing several advantages in terms of:

- implementation degrees of freedom for low-power, high-performance and small-area;
- reuse of existing code reducing the development effort;
- reduction of the verification effort;

3.2. How HLS works

The HLS process is essentially composed by three phases: resource allocation, scheduling and binding (Coussy, 2009)
In the ***Resource Allocation*** phase, the tool tries to understand how the algorithmic operation can be represented using hardware operators, such as adders, multiplier and connectivity structures (multiplexer, ...).
The ***Scheduling*** is the most complex part of the process. The tool insert the concept of clock cycle, in the design, absent in the design's entry model.
In the ***Binding*** phase, the resources allocated in the previous phase will be bound to algorithmic operators respecting the cycles assigned during the scheduling phase.
The advantage of this approach consists in having different possible outputs from the same algorithm. And these differences are tuned using design directives.

3.2.1. Design Directives

Design directives are a set of configurations used by the HLS tool to manipulate the micro-architectural implementation of the algorithmic code. Examples of these directives are: PIPELINING, UNROLLING, MERGE, FLATTEN, INLINE, etc... (Xilinx, 2015). The simplest example can be represented by a *for loop* that execute an array sum function (Listing 1).

```
for ( i=0; i<4; i++ )
{
  y = x[i] + y;
}
```

Listing 1. Array Sum Function

The same loop in hardware can be executed using a variable numbers of clock cycles, depending on the parallelism of the hardware resources. For example, setting the UNROLL factor to 4 (full unroll), the loop will requires only one clock cycle to complete. The drawback is the resources required (4 adders) and the critical path, therefore limiting then the maximum frequency achievable. Instead, by setting the PIPELINE directive, the same loop can be executed in two clock cycles using less resources (for example just 2 adders) and without increasing the critical path. Both solutions are technically acceptable, with different implementation trade-offs. Several works on this topic have been done in the past (Watanabe, 2012) and (Andrade, 2015) and in practice, there is no golden rule to be applied always to all designs. The right directives are related to the single use cases and a single implementation goal. In some cases, for the same loop or the same functions, different directives could be applied depending on the overall context such as the requirements specifications and design constraints.

3.3. Commercial Tools

Over the years the HLS tools are evolved from simple software-to-hardware converters to a real industrial trend. Historically, the evolution of high-level synthesis can be divided into three generations (Martin, 2009). Examples of state of art HLS tools include: Synphony(Synopsys), CyberWorkBench(NEC), Stratus(Cadence), Catapult (Mentor Graphics), Vivado HLS (Xilinx) and so on. Recently Intel has announced the release of its HLS tool (Intel, 2017) for FPGAs, now part of Intel's portfolio after Altera's acquisition.

3.3.1. VIVADO HLS

Vivado is the official High-Level Synthesis tool from Xilinx. It was originally developed by AutoESL, under the name Autopilot. After the acquisition by Xilinx they renamed it as Vivado HLS, in coherence with the new full-featured EDA tool. Its output can be targeted into the standard Xilinx FPGAs as a traditional design flow.

4. SISO Processor

4.1. General Introduction

The encoding procedure of Turbo Codes is relatively simple to be implemented, while the decoding side presents several complex mathematical statements. In the first paper about turbo codes (Berrou, 1993), the MAP (maximum a posteriori probability) algorithm is applied to the SISO decoder to evaluate the soft values of each component's convolutional code.

4.2. Mathematical Model

4.2.1. MAP ALGORITHM

In Bayesian statistics, the MAP estimation consists of a mode evaluation of the posterior distribution. Mathematically, it can be expressed as (1)

$$\hat{\theta}_{MAP} = arg_{MAX} \frac{f(x|\theta)g(\theta)}{\int\limits_{\vartheta} f(x|\theta)g(\theta)d\vartheta} \tag{1}$$

Where θ is an unobserved parameter on the basis of observations x, g is density function of θ and $f(x)$ a sampling distribution of x. The MAP algorithm referred in the first publication was a modified version of the BCJR algorithm (Bahl, 1974), the optimal symbol decoding algorithm that minimizes the probability of a symbol error. It computes the a posteriori probabilities (APPs) of the information bits given the received sequence. The MAP algorithm, also known as forward-backward algorithm, can be summarized as the ratio between the conditional probability (2)

$$L(\hat{u}_k) = \log \frac{P(u_k = +1|y)}{P(u_k = -1|y)} \tag{2}$$

To evaluate the probability over the sequence y, we need to introduce the joint probability $P(s', s, y)$ defined as (3)

$$P(s', s, y) = \alpha_{K-1}(s') \cdot \gamma_K(s') \cdot \beta_K(s) \tag{3}$$

Where α_K and β_K are the forward and backward metrics, respectively computed as (4) and (5).

$$\alpha_K(s) = \sum_{s'} \gamma_k(s', s) \cdot \alpha_{K-1}(s') \tag{4}$$

$$\beta_K(s') = \sum_{s} \gamma_k(s', s) \cdot \beta_{K+1}(s) \tag{5}$$

In the above equations, γ is the state transition probability and is computed as (6). In practice, is it the probability that the received symbol is y_k at time k and the current state is $S_k = s$, knowing that the state from which the connecting branch came was $S_{k-1} = s'$.

$$\gamma_K(s', s) = P(s|s')P(y_k|s', s) = P(u_k)P(y_K|u_k) \tag{6}$$

Then the a posteriori probability (APP) log-likelihood ratio (LLR) of the information bits can be expressed as 7:

$$L(\hat{u}_k) = \log \frac{\sum_{u_k=+1} \alpha_{K-1}(s') \cdot \gamma_K(s') \cdot \beta_K(s)}{\sum_{u_k=-1} \alpha_{K-1}(s') \cdot \gamma_K(s') \cdot \beta_K(s)} \tag{7}$$

The probability α is being computed as the sequence y is received, while β can only be computed after we have received the whole sequence. This is the reason why this algorithm is also know as forward-backward algorithm (Abrantes, April 2004). However, due to the high implementation complexity of the MAP algorithm in hardware in terms of memory requirements and related to the computation of transcendental functions, several simplified implementations like Log-MAP and Max-Log-MAP were developed.

4.2.2. LOG-MAP ALGORITHM

To avoid the complicated multiplications and solve the numerical instability issues, it is possible to compute the MAP algorithm in the log domain, but this requires the introduction the max^* operator (8)

$$max^*(a, b) = \log(e^a + e^b) = max(a, b) + \log(1 + e^{-|a-b|}) \tag{8}$$

For the Log-MAP algorithm, a lookup table with high accuracy can lead to the same performance as the MAP algorithm, but it also results in the use of larger memory.

4.2.3. MAX-LOG-MAP ALGORITHM

A further simplification can be achieved by completely discarding the lookup table. The correction term will be omitted from the max^* function (9)

$$max^*(a, b) = max(a, b) \tag{9}$$

This approximation leads to the Max-Log-MAP algorithm that involves only addition and the max function. However, the lack of correction term would make the LLR calculation too optimistic and degrade performance. This degradation can be compensated by a technique known as extrinsic scaling, which consists in multiplying $L_e(u_K)$ by a scaling factor ζ (10).

$$L'_e(u_K) = \zeta \cdot L_e(u_K) \tag{10}$$

Usually the value of ζ is between 0.5 and 0.75, depending on the code's structure and channel conditions.

Replacing the max^* operator in (4) and (5) we obtain (11) and (12):

$$\alpha_K(s) = max_{sk+1}\{\alpha_{K-1}(s_{K-1}) + \gamma_k(s_{K-1}, s_K)\} \tag{11}$$

$$\beta_K(s) = max_{sk+1}\{\beta_{K+1}(s_{K+1}) + \gamma_k(s_{K+1}, s_K)\} \tag{12}$$

To obtain the extrinsic information, $\Lambda(\hat{u}_K)$ can be split in three terms: extrinsic LLR $L_e(u_K)$, a priori LLR $L_a(u_K)$ and systematic LLR $L_c(y^s_K)$ as (13)

$$\Lambda(\hat{u}_K) = L_e(u_K) + L_a(u_K) + L_c(y^s_K) \tag{13}$$

4.3. Practical Architecture

Over the years several implementation techniques have been developed in order to increase the computation efficiency and reduce the complexity of the SISO processor. These include such as the *Sliding Windows*, *Early Stopping*, *Parallel Architectures*, ... In this section we present a simple hardware implementation of the SISO processor (Wong, 2014). Practically, the main processing elements

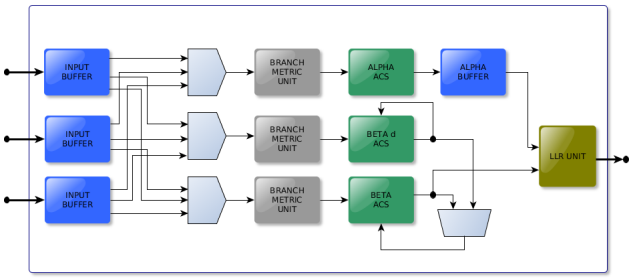

Figure 3. SISO Processor-Internal blocks

inside SISO processor are units which execute the BCJR equations, usually expressed in logarithmic form (Ahmed, 2013). Basically, the processor is composed of the following units:

- *BMU Branch Metric Unit;*
- *α ACS and β ACS;*
- *LLR Unit;*
- *Buffers;*

4.3.1. Branch Metric Unit

Is the unit that calculates the branch metrics $\gamma s'$, combining the LLRs and the apriori information. From the hardware point of view, it is composed of adder modules.

4.3.2. ACS

The ACS (Add Compare Select) unit evaluates the forward and backward metrics α and β. It compute the max*() function executing sum, compare and selecting the output(Fig.4). ACS operations, in general are the most time-consuming and resource expensive of the SISO processor.

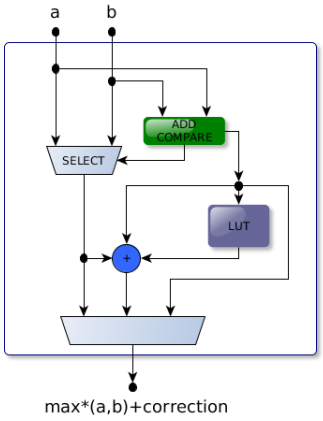

Figure 4. ACS Unit-Internal blocks

4.3.3. LLR Unit

This unit executes the elaboration of the extrinsic information; It is composed only by adders.

4.4. Standard Applications

As mentioned early in the paper, the SISO processor is the main functional component in Turbo Codes and LPDC FEC techniques used in standard communication systems such as 3GPP and DVB. Each standard has its own code length and block size and in general, the hardware architecture is optimized for a specific standard. Since this design starts from the algorithmic model, the block size is

simply represented as a preprocessor *#define* in the source code. In particular, the following synthesis tests have been performed using LTE standard (ETSI, 2014).

5. High-Level Synthesis of SISO Processor

The first part in designing an RFNoC block consists in designing the hardware module to be later encapsulated in the NoC Shell. In this case, HLS will be used to obtain the hardware model of the DSP processing engine.

5.1. Development Flow

5.1.1. Tool Setup

Since the goal of this design is to create an RFNoC block to be used in USRP devices, the solution settings are tailored following their hardware specifications (Table 1).

USRP Device	**X310**	**E313**
FPGA [part number]	XC7K410T	XC7Z020
LEs [K Units]	406	85
Memories [Kb]	28620	5040
DSPs [Units]	1540	220
Frequency [MHz]	200	100

Table 1. USRP Specifications(Ettus, d)

5.1.2. Design Entry

The algorithmic model of the SISO processor is described in SystemC, a language supported by Vivado HLS. The top_level function takes care to load data into the array memory passed as reference to the SISO function to be processed. In order to be integrated into the RFNoC, the top_level module expose an AXI4Stream interface.

5.2. Preliminary Experimental Results

As described in Section 3, different RTL implementations could be generated from the same source code. For this design, four different implementation approaches have been tested: using default directives, maximizing performance, minimizing resource utilization and mixing both performance/resources directives.

5.2.1. First Solution

By default Vivado synthesize C functions into functional blocks in the RTL hierarchy. Then, all instances of the same function will be assigned to the same RTL hardware resource. All the arrays will be synthesized as BRAMs into the FPGA and all the loops are left rolled. This means that synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence.

	X310		**E313**	
Resources	**Tot**	**Use%**	**Tot**	**Use%**
BRAMs	24	1	24	8
FFs	7520	1	6247	5
LUTs	42121	16	41616	78
	Timing			
***Latency**[clocks]*	25029		18981	

Table 2. Resource Utilization - *Default* directives

Table (2) reports the synthesis results for the two targets without applying any specific synthesis directive.

5.2.2. Performance optimizations

Optimizing the performance of the SISO processor algorithm, means improving the overall concurrent execution of its internal operations. This can be achieved by introducing a level of parallelism for the forward and backward metric computation and trying to provide minimum latency and maximum throughput for the ACS unit.

	X310		**E313**	
Resources	**Tot**	**Use%**	**Tot**	**Use%**
BRAMs	24	1	24	8
FFs	38057	7	28955	27
LUTs	138071	54	128568	240
	Timing			
***Latency**[clocks]*	23094		17974	

Table 3. Resource Utilization - *Performance optimizations* directives

Table (3) shows a reduction of latency in exchange of an increase in resource utilization with respect to the first solution.

5.2.3. Resources optimizations

Optimizing for resource utilization is not simple and requires a deep inspection of the original algorithmic model. As a first analysis, one can investigate on the precision needed (data width) for the variables used. When predefined software data types of 8/16/32 bits are mapped to hardware, any unused bit will result in wasted area resources. Another option to improve resource utilization is to apply the INLINE directive to functions that result in very simple hardware module. In this way, some extra-control logic will be avoided and some area could be saved. Table (4) reports the synthesis results with the resource optimization directives.

	X310		E313	
Resources	**Tot**	**Use%**	**Tot**	**Use%**
BRAMs	21	1	21	7
FFs	4807	1	3872	3
LUTs	9256	3	8778	16
	Timing			
***Latency**[clocks]*	25029		18981	

Table 4. Resources Utilization - *Resource optimization* directives

5.2.4. MIXED OPTIMIZATIONS

As anticipated in *Section 3.2.1*, the HLS directives are powerful methods for optimizing the solution towards one specific implementation metric and goal. However, the best usage of these directives consists in mixing them to get most of the advantages coming from the high-level synthesis process. In fact, the best results were achieved by applying the right tradeoff between performance and resource optimization directives.(Table 5).

	X310		E313	
Resources	**Tot**	**Use%**	**Tot**	**Use%**
BRAMs	21	1	21	7
FFs	33419	6	25578	24
LUTs	100600	39	85547	160
	Timing			
***Latency**[clocks]*	23741		17981	

Table 5. Resource Utilization - *Mixed optimizations* directives

5.3. Results Analysis

Figure (5) shows the results for E313 and X310 USRP devices. The percentage of occupation area is expected results since the large difference between the two FPGAs. The surprising results is the difference in latency (expressed in clock cycles) obtained just synthesizing for the two targets.

5.3.1. CONSIDERATIONS

The different results in terms of latency are due to two main reasons: The first is the different performance level of the device and the second is the operating frequency used. Furthermore, the solutions outgoing from the HLS process are heuristic and not optimal. The resource utilization results obtained show how the X310 can host multiple implementation versions. Neverthless, the E313 can be also used for a limited subset. Particular focus have to be highlight in the solutions exploration effort. In fact, all the solution proposed have been performed in a few days instead of months as in traditional RTL design exploration. Another relevant

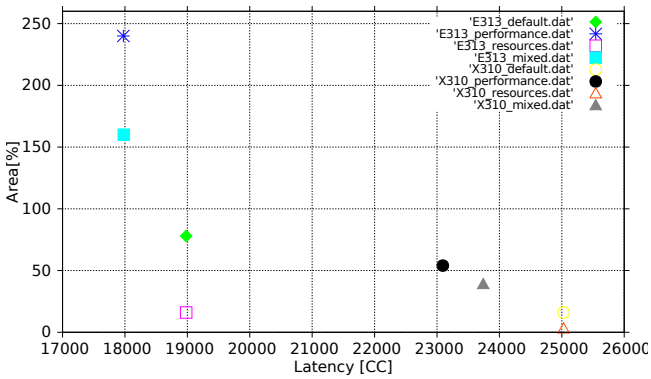

Figure 5. E313/X310 Solution-Area/Latency

element is constituted by the Vivado version. All the results showed in this paper has been obtained using version 2015.4 as required from the challenge. However, different results have been obtained with recent versions of the same tool.

6. Conclusion

In this paper, we introduced the design of an RFNoC block implementing a SISO processor, using the High-Level Synthesis process to create the hardware description starting from the algorithmic model. The analysis of the latency/area tradeoffs show how the same model can be synthesized in several different ways, sometimes causing infeasibility issues for specific devices. More specifically, we have performed some benchmarks targeting the SISO processor into two different USRP devices provided from Ettus, the X310 and E313. We highlight the advantages coming from the use of HLS tools for DSPs algorithms design exploration.

6.0.2. FUTURE WORK

At this stage the RFNoC block cannot be considered ready to be used by the GNU Radio Community. A number of extensions to this preliminary activity are being investigated. The first consists in the selection of a solution among the ones we explored and continuing through the design flow to obtain a complete RFNoC block. After that, it will be possible to test the block's functionality in the GNU Radio environment. However, since the preliminary experimental results are not particular promising in terms of area/latency, an algorithmic code-restructuring cannot be ruled out.

References

Abrantes. From bcjr to turbo decoding: Map algorithms made easier, April 2004.

Ahmed, Maurizio Martina, Guido Masera. Turbo decoder

vlsi architecture with non-recursive max operator for 3gpp lte. In IEEE (ed.), *SPACOMM 2013 : The Fifth International Conference on Advances in Satellite and Space Communications*, pp. 40–45, 2013.

Andrade, Nithin George, Kimon Karras David Novo Vitor Silva Paolo Ienne Gabriel Falcao. From low-architectural expertise up to high-throughput non-binary ldpc decoders: Optimization guidelines using high-level synthesis. In IEEE (ed.), *25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, 2015.

Bahl, J. Cocke, F. Jelinek J. Raviv. Optimal decoding of linear codes for minimizing symbol error rate. pp. 284–287, 1974.

Berrou, Alain Glavieux, Punya Thitimajshima. Near shannon error-correcting coding and decoding - turbo codes. In IEEE (ed.), *Communications Technical Program*, pp. 1064–1070, Geneva, Switzerland, 1993.

Camposano. From behavior to structure: high-level synthesis. 1990.

Coussy, Daniel D. Gajski, Michael Meredith Daniel D. Gajski Andres Takach. An introduction to high-level synthesis. 2009.

ETSI. Lte: Evolved universal terrestrial radio access (e-utra) multiplexing and channel coding. Technical report, ETSI, 2014.

Ettus. https://www.ettus.com/sdr-software/detail/rfnoc-vivado-challenge, a.

Ettus. https://files.ettus.com/manual/page_rtp.html, b.

Ettus. https://kb.ettus.com/getting_started_with_rfnoc_development, c.

Ettus. https://www.ettus.com/product, d.

Fingeroff. *High-Level Synthesis Blue Book*. Mentor Graphics, 1nd edition, January 2010.

GNURadioCompanion. https://wiki.gnuradio.org/index.php/gnuradiocompanion.

Intel. https://networkbuilders.intel.com/blog/hardware-programmers-achieve-design-goals-faster-with-intel-high-level-synthesis-compiler-for-fpgas, 2017.

Karra. Implementation of turbo product codes in the fec-api, 2012.

M. Braun, Jonathon Pendlum. Rfnoc: Rf network on chip. *GNU Radio Conference*, 2015.

Martin, Gary Smith. High-level synthesis: Past, present, and future. 2009.

Nurvitadhi, Jaewoong Sim, David Sheffield Asit Mishra Srivatsan Krishnan Debbie Marr. Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic. In IEEE (ed.), *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, Lausanne, Switzerland, 2016.

Pendlum. Rfnoc deep dive: Fpga side, 2014.

Takach. High-level synthesis: Status, trends, and future directions, 2016.

Talasila. Implementation of turbo codes on gnu radio, 2010.

Watanabe, Kondratyev Luciano Lavagno Mike Meyer Yosinori. Exploiting area/delay tradeoffs in high-level synthesis. pp. 1024 – 1029, 2012.

Wong, Hsie-Chia Chang. *Turbo Decoder Architecture for Beyond-4G Applications*. Springer, 1nd edition, 2014.

Xilinx. Vivado design suite user guide - high-level synthesis, 2015.

Xilinx. https://forums.xilinx.com/t5/xcell-daily-blog/ettus-research-accepts-7-teams-to-compete-in-the-10k-rfnoc-amp/ba-p/751761, 2017.