Optimized Rate Matching Architecture for a LTE-Advanced FPGA-based PHY

Karlo G. Lenzi, José A. Bianco F., Felipe A. de Figueiredo, Fabrício L. Figueiredo
DRC – Convergent Networks Department
CPqD – Research and Development Center
Campinas, SP - Brazil
{klenzi, jbianco, felipep, fabricio}@cpqd.com.br

***Abstract*—In this paper we present an optimized rate matching architecture for a LTE-Advanced FPGA-based physical layer. Since LTE-Advanced can reach up to rates of 1 Gbps in downlink, and since rate matching is in that critical path, it is very important that the design of the hardware architecture be efficient enough to allow this high data rate with little resources as possible. If not well planned, implementations on FPGAs can be quite challenging, limiting the choices of speed grades and FPGAs sizes capable of supporting such requirements. We propose efficient hardware architecture for the LTE-Advanced rate matching generic procedure; occupying only 218 slices and 9 block RAMs and performing in frequencies greater than 400 MHz in a FPGA-based solution.**

Keywords- LTE; rate matching; FPGA design; hardware architecture; wireless communication.

I. INTRODUCTION

The rapid growth of mobile users and the ever increasing demand for broadband wireless access are driving the development of new and innovative mobile standards. The Long Term Evolution (LTE) standard, devised by the 3rd Generation Partnership Project (3GPP) [12], is one of the candidates to become the 4G de facto standard around the world. LTE-Advanced [11] is an enhancement over the original LTE standard which is set to provide higher bitrates in a cost efficient way. One of the approaches used by LTE-Advanced to attain higher rates is through improving the channel efficiency, i.e., channel coding [1].

In order to improve channel efficiency, it was decided that the LTE-Advanced standard would employ Adaptive Modulation and Coding (AMC) as one of the means to adapt the data rate to the channel conditions. AMC is based on a fixed mother code rate of 1/3 of a turbo or convolutional encoder. The coded words are then forwarded to the rate matching procedure, where different code rates are generated from the same fixed mother code rate [4].

With the purpose of generating any arbitrary code rate, the rate matching procedure repeats, for code rates less than 1/3, or punctures, for code rates greater than 1/3, the bits of the mother code word. Thus, an arbitrary code rate may be achieved. Additionally, Hybrid Automatic repeat Request (HARQ) combining is also allowed by the use of rate matching, once all encoded bits are obtained from the same 1/3-encoded code word [5-7, 13-15].

To achieve this, the rate matching procedure uses a concept of a virtual circular buffer, where the code words are stored. Since both encoders produce 3 outputs per input bit, the rate matching must handle parallel bits streams for later serialization, where different code rates will be produced. This parallel to serial behavior together with the need to perform all PHY processing in 1 ms [18], which include channel coding, modulation and transmission, places the rate matching procedure in the LTE critical performance path, thus demanding careful consideration in the design.

This work assesses the efficiency and effectiveness of an optimized rate matching architecture used in a FPGA LTE-Advanced eNodeB (Evolved Node B - similar to a base station in GSM networks) in order to achieve the strict timing constraints of the standard. The remainder of this work is organized as follows. Section II describes the LTE-Advanced Rate Matching procedure. Section III presents the hardware architecture proposed. Section IV discusses the FPGA design considerations of this architecture. Results are presented and discussed in Section V. Finally, conclusion remarks are shown in Section VI.

II. LTE RATE MATCHING

The rate matching procedure is one of the five generic procedures defined in the LTE-Advanced physical layer channel coding standard [1]. It is also the last one performed in the coding processing chain, being preceded by a CRC calculation, turbo coding (with code block segmentation) or convolutional coding [1, 19].

The purpose of rate matching is to create different code rates according to some physical layer (PHY) parameters, such as modulation, channel quality, etc. It works in a per code block basis with sizes ranging from a few bits and up to thousands of bits [1,4,5], demanding a very flexible architecture to handle such variations.

There are two types of rate matching: one specified for turbo encoded data streams and one for convolutional encoded one. For both encoders a fixed code rate of 1/3 is employed. From this fixed code rate input, the rate matching procedure is capable of deriving other common code rates based-on a circular buffer concept. The structure of the rate matching procedure is depicted in Fig. 1.

978-1-4799-1337-4/13/$31.00 ©2013 IEEE

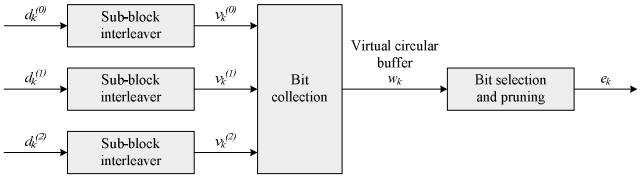

Figure 1. Rate Matching generic procedure.

By examining Fig. 1 we can see that rate matching behaves as a serializer for three parallel bit streams with very high bit rates. For this reason, rate matching becomes a bottleneck for the system's performance if not well designed.

The encoder generates three encoded bit streams for each code block (CB) input, where K is the CB size and may assume one of the values presented in [1], section 5.1.3.2.3. These streams represents one systematic bit $d_k^{(0)}$ and two parity bits $d_k^{(1)}$ and $d_k^{(2)}$, where the bits $k = 0, .., K-1$, are fed to one of the three sub-blocks interleavers. The interleaved bits from each sub-block $v_k^{(0)}$, $v_k^{(1)}$ and $v_k^{(2)}$, now of size K_π, are stored in a circular buffer [5-7] of size $K_w = 3K_\pi$. The performance of Circular Buffer Rate-Matching (CB-RM) is discussed in [8, 9].

If the rate matching procedure is preceded by a turbo encoder, then the interleaved systematic code is stored first in memory, followed by an interlaced $v_k^{(1)}$ and $v_k^{(2)}$ parity bits. On the other hand, if a convolutional encoder is used, the interleaved codes are stored sequentially, following the order $v_k^{(0)}$, $v_k^{(1)}$ and $v_k^{(2)}$.

The procedure of storing these three bit streams on memory is called bit collection. Selecting which bits to read in order to create different code rates is called bit selection. The bit selection reads out the systematic/parity bits (w_k) and through the repetition or puncturing of the code stored in the circular buffer, it implements different code rates (e_k).

Fig. 2 illustrates each stage from encoding to collecting and selecting the code words for a turbo encoder processing chain. In case of a convolutional encoder, the same data flow may be applied, except for the interlacing of parities $v_k^{(1)}$ and $v_k^{(2)}$.

A. Sub-block Interleaver

The sub-block interleaver is defined according to the encoder type used. For both cases, the interleaver is based on matrix interleaving, in which K data bits are written in rows and read out column by column. With C columns, where C is constant and equal to 32 for both encoder types, the number of rows R can be computed through equation (1).

$$R = \text{ceil}(K / C) \quad (1)$$

This matrix size $K_\pi = R \times C$, has a varying row size. For turbo coded streams, the block size ranges from 44 bits to 6148 bits (turbo encoder trellis termination bits included), which limits R between 2 and 193 [5]. If the matrix size is greater than the actual CB size, N_D dummy bits are padded at the beginning of the bit stream, where N_D ranges from 0 to $C - 1$ dummy bits and it is computed by (2).

$$N_D = K_\pi - K \quad (2)$$

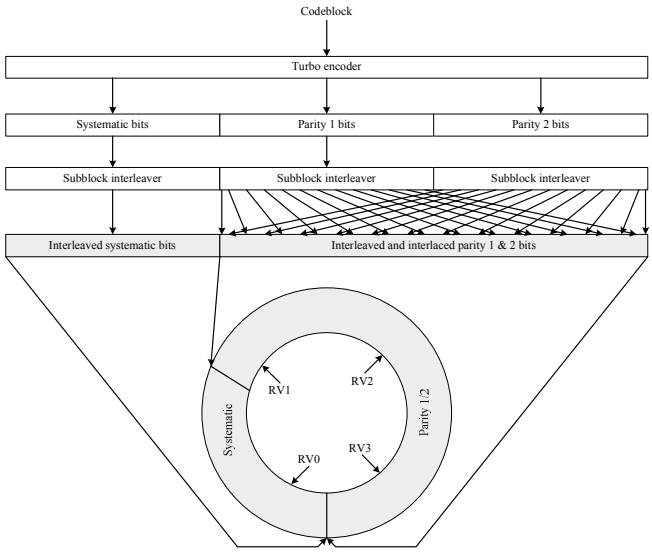

Figure 2. Rate matching dataflow for turbo encoder.

The order in which each column is read follows a predefined permutation pattern [1], which depends on the type of encoder used. Table I presents the permutation patterns for each encoder.

TABLE I. PERMUTATION PATTERN.

Number of columns $C_{sub-block} = 32$	**Permutation pattern** $< P(0), P(1), ..., P(C_{sub-block} - 1) >$
Turbo Encoder	< 0, 16, 8, 24, 4, 29, 12, 28, 2, 18, 10, 26, 6, 22, 14, 30, 1, 17, 9, 25, 5, 21, 13, 29, 3, 19, 11, 27, 7, 23, 15, 31>
Convolutional Encoder	< 1, 17, 9, 25, 5, 21, 13, 29, 3, 19, 11, 27, 7, 23, 15, 31, 0, 16, 8, 24, 4, 20, 12, 28, 2, 18, 10, 26, 6, 22, 14, 30 >

The second difference caused by the encoder type is that for the turbo codes, the output of parity bits $d_k^{(2)}$ follows a slightly different interleaving rule. Equation (3) and (4) map the behavior of both interleavers. Equation (4) is used for the $v_k^{(2)}$ parity bits of the turbo codes only.

The order in which the matrix will be filled and emptied is a function of the matrix size K_π, the number of rows R, the permutation pattern P and the current bit index k, where $k = 0, \ldots, K_\pi - 1$.

$$\pi(k) = \{P[\text{floor}(k / R)] + C * (k \bmod R)\} \bmod K_\pi \quad (3)$$

$$\pi'(k) = \{P[\text{floor}(k / R)] + C * (k \bmod R) + 1\} \bmod K_\pi \quad (4)$$

B. Bit Collection, Selection and Transmission

The circular buffer [5, 8-9] should be at least of size $K_w = 3K_\pi$ to accommodate the three parallel input bit streams. For turbo codes, the bit stream should be stored in memory according to (5):

$$\begin{aligned} w_k &= v_k^{(0)}, \text{ for } k = 0, \ldots, K_\pi - 1 \\ w_{K+2k} &= v_k^{(1)}, \text{ for } k = 0, \ldots, K_\pi - 1 \\ w_{K+2k+1} &= v_k^{(2)}, \text{ for } k = 0, \ldots, K_\pi - 1 \end{aligned} \quad (5)$$

Which means that the systematic bits are stored first, followed by the interlaced $v_k^{(1)}$ and $v_k^{(2)}$ parity bits. For convolutional codes, the procedure is straight forward, as presented by (6). Here, there is no need to interlace the parity bits.

$$\begin{aligned} w_k &= v_k^{(0)}, \quad \text{for } k = 0, \ldots, K-1 \\ w_{K+k} &= v_k^{(1)}, \quad \text{for } k = 0, \ldots, K-1 \\ w_{2K+k} &= v_k^{(2)}, \quad \text{for } k = 0, \ldots, K-1 \end{aligned} \tag{6}$$

After storing the bit streams, bits are read out of the circular buffer to achieve some specified overall code rate by puncturing or repeating these stored bits. The rate matching output sequence length is denoted by E, while the rate matching output bits are represented by e_k, where k =0, .., E-1.

The value of E is dependent of several PHY parameters and can be viewed as an input to the rate matching generic procedure. Again, there is a small difference in the way bits are read based on the type of encoder used. The listing (a) demonstrates how bit selection is done:

Listing (a). Rate matching bit selection.

```
k =0 and j = 0
while k < E
   If w(k0+j) mod Ncb ≠ <NULL>
      ek = w(k0+j) mod Ncb
      k= k + 1
   end if
   j = j + 1
end while
```

In the pseudo-code listed in (a), k_0 is the initial offset of the circular buffer, N_{cb} is the CB buffer size, k is the bit index and j is an auxiliary counter. For turbo codes, k_0 and N_{cb} are defined by the 3GPP standard as a function of the redundancy version used, modulation type, transport block soft buffer size, among others [1,4,5]. To compute these parameters, along with the rate matching output sequence length E it is necessary to follow the procedure show in [1], section 5.1.4.1.2. For convolutional codes, $k_0 = 0$ and $N_{cb} = K_w$.

The pseudo-code listed in (a) reads E valid bits from a circular buffer of size N_{cb}, starting from index k_0, discarding any dummy bits that might have been inserted by the channel coding procedures.

III. Proposed Hardware Architecture

The proposed hardware architecture for the generic rate matching procedure is depicted in Fig 3. This architecture is optimized for FPGAs, allowing a low resource count and a high frequency rate. This architecture is part of a complete LTE-Advanced PHY solution for an eNodeB base-station on an FPGA device, since cost, size and power consumption are less critical compared with user equipments (UE) constraints.

There are two main types of components in the rate matching architecture: memories (RAMs) and data address generators (DAGs).

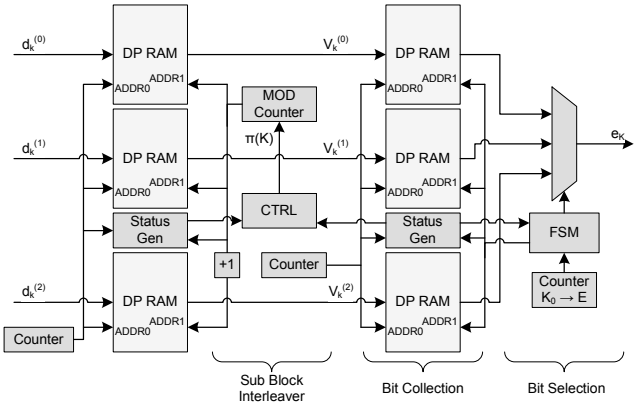

Figure 3. Rate matching proposed architecture.

Before interleaving $d_k^{(0)}$, $d_k^{(1)}$ and $d_k^{(2)}$, it is necessary to store them into memories. Three input memories of a size greater than K takes care of that task. After storing and interleaving each of the bit streams, bit collection takes place and another memory is used for gathering these streams and arranging them into a predefined order, according to the encoder type.

To conclude the rate matching procedure, bit selection chooses based on PHY parameters the starting point k_0 and the amount of data E to be read out, and thus resuming the basic memory structure needed for the rate matching procedure. Memory requirements can be specified based on the maximum input code block size K = 6144 allowed in the LTE channel coding standard.

All rate matching memories require random access. If a simple dual-port RAM (one read/write port and one read port) is used, a total memory size of two times the maximum code block size K would allow the rate matching sub-block interleaver to run without blocking the encoder.

A ping-pong memory structure would avoid any buffer overflow. That way, while the rate matching procedure is processing one code block, another code block could be written into memory at the same time without any memory conflicts.

If these memory requirements are too demanding for a certain FPGA device, it is possible to reduce the buffer size, but additional feedback control logic (ready to send, clear to receive, busy signals, etc.) would be necessary to manage the data flow and to prevent previous stages from overwriting the memory contents in case of memory overflow.

Since CB sizes dynamically chance in each transmission, it is necessary to keep track of some CB parameters so that the rate matching procedure can operate accordingly. Basically, the input block size, its entry point in memory (base address) and the k_0 and E values are enough to guarantee the correct operation of the rate matching procedure.

FIFOs are well suited for storing theses parameters. Control logic will be needed to synchronize such parameters with the current code block being processed and to implement some control flow in the hardware architecture. In

Fig. 3, this block is called 'CTRL'. It receives status information of the memories (full, empty, etc.) and decides whether to start a new block processing or not.

A. Sub-block Interleaver

To design the sub-block interleaver only one input parameter is needed: the code block size. From K, it is possible to compute R and N_D, according to equations (1) and (2).

Regarding equation (1), since the number of columns C is 32, the division becomes a bit routing circuit (no resources required). The ceiling operation can be designed through a 5-bit comparator and an 8-bit adder. Based on the value of the 5 LSB bits, the number of rows R is incremented or not.

To determine N_D, we need to find the sub-block interleaver block size K_π. To compute K_π, just shift R to the left by 5 bit positions. Again, no hardware resource will be needed to implement this operation.

Additionally, the permutation patterns shown in Table I need to be stored based on the selected encoder to support the interleaving operation. A look-up table (LUT) of 32 x 5 bits is enough to store all of these parameters.

In order to design the interleaver DAG, depicted in Fig. 3 by the name 'MOD counter', equations (3) and (4) can be split into two expressions. The first one is the indexing of the permutation pattern, which is based on an integer division of the bit index k by R, where k ranges from 0 to K_π - 1. The second part is the reminder of that k integer division, expressed by k modulus R.

These two sub-expressions can be described as modulus counters, one for a column count and one for a row count. Rows are incremented step by step in a per clock basis, while columns are incremented every time the row counter reaches its maximum value.

In expression (3) there also is the addition of a column pattern with the row counter multiplied by C. This operation can be implemented by the concatenation of these two bit vectors, row count value with the pattern value indexed by the column count. Listing (b) shows the resulting address, where the operator '&' represents concatenation.

Listing (b). Sub-block interleaver DAG.

```
addr = row_counter & pattern(col_counter)
```

B. Bit Collection

The bit collection needs to gather the bit streams of three different parallel bit streams coming from the sub-block interleaver and store them for later serialization.

For each type of encoder, there is a different rule to store data in memory; therefore, the bit collection core processing, like the sub-block interleaver, is based on generating addresses efficiently.

In this case, the challenge is to work with a non-power of two numbers of parallel input streams, which increases control logic, due to the limited options of block RAM styles in a FPGA architecture. Another difficulty is to have a non-constant and also non-power of two code block sizes, which increases the complexity of the architecture.

FPGAs do not have a memory type capable of accepting three input write ports so that each bit stream can be stored in a different memory region [17], according to (5) and (6). From a HDL design point of view, it is possible to describe such memory, but at the implementation level, this memory will be mapped as a composition of dual-port RAMs, which will increase control logic to handle the three parallel input bit streams in FPGA.

The first alternative is to use a dual-port RAM to store systematic bits, and a second memory, which is also a dual-port RAM but with asymmetric aspect ratio 2:1, for storing the parity bits.

This approach can be very efficient for turbo coded streams, since the interlacing of the parity bits would be done automatically by the asymmetry of the memory. A second approach is to use three simple dual-port RAMs, each one storing one interleaved bit stream, which will transfer the hardware complexity to the bit selection DAG.

While the first approach works better with turbo codes, because it automatically interlaces the parity bits, the second one is recommended for convolutional coded streams, once they are stored sequentially in a per block basis. For the architecture proposed in this paper, we chose the second one for both encoders, since using the same structure will allow the same bit selection hardware to be used for both. The requirements for the bit collection memory is 18Kb ($K_w = 3K_{\pi\_max}$).

C. Bit Selection

The bit selection hardware generates read addresses for the bit collection circular buffer. Its behavior depends on the encoder being used and follows the pseudo-code presented in listing (a).

This block has two input parameters, E and k_0. For convolutional encoded streams, k_0 is always equal to zero, while for turbo encoded streams, the values of k_0 and E are defined by the PHY configuration. These two parameters will define the overall system's code rate.

Due to the fact that bit selection will have to handle three different memories, from an implementation perspective, as well as to ignore dummy bits from the stream, while maintaining a circular buffer behavior, in the proposed architecture, this module was implemented with a finite state machine (FSM). Fig. 3 illustrates this block by the name 'FSM'.

IV. FPGA Design Considerations

In order to achieve the performance required by the LTE-Advanced standard [1, 7] in a FPGA environment, some considerations should be made.

First, a new TB may arrive every 1ms for PHY processing. Assuming a SISO (single-input, single-output) system, where the maximum TB size is 75Kb [20], we need to perform all LTE-Advanced PHY procedures in that period of time, which include channel coding, modulation and transmission. This requirement imposes a latency constraint that will limit the total number of cycles available for each processing stage. The faster each stage processes its data, the easier will be to the next stage to complete its computation

on time, and the lower will be the total latency of the PHY engine.

For that reason, a common frequency requirement for the channel coding procedures is to operate at some multiple frequency of the base-band processing frequency, which is 30.72MHz.

Since most modern FPGAs devices may achieve a maximum frequency of 550MHz, and since this theoretical value is impossible for any complex systems, a frequency constraint between 300 and 450 MHz would be a feasible design choice. One common constraint value for LTE-Advanced PHY engines is to operate at least at 307.2 MHz in the channel coding stage, 10 times the base-band sampling frequency.

As mentioned in previous sections, the rate matching architecture is composed basically of two elements: RAMs and DAGs, in a total of theoretically four RAMs and four DAGs. Three of these RAMs sizes have at least K bits (K_{max} = 6 Kb) and one memory of at least K_w (K_{w_max} = 18Kb). These memories gather the data from the three different bit streams store them into one single memory (conceptually) and releases the stored bits according to the system's overall rate for a particular PHY configuration.

The DAGs, on the other hand, are used as follows: one for generating the sub-block interleaver read addresses, if turbo encoder is applied, a second DAG is needed for the $d_k^{(2)}$ sub-block bit stream; one for generating the bit collection write addresses and one for generating the bit selection read addresses.

There are two ways of implementing memories in FPGAs: 1) through distributed memory, which are implemented with the LUTs available in slices, or 2) through dedicated memories blocks (RAMB) in FPGA, which are large RAM cells placed on the silicon chip.

The first approach is recommended when small quantities of memory are needed or when there are lots of available resources in FPGA. This will reduce path delays from memory to logic, but will reduce the total number of available logic. The second approach is best suited for applications that demands large quantities of memory.

In the case of designing the entire LTE PHY engine in a single FPGA and due to the fact that code block sizes can reach up to 18Kb at the end of the coding chain, it is highly desired to keep the number of occupied slices to a minimum. Therefore, the use of block RAMs is recommended in this case and will be used in the architecture proposed.

RAMBs in most FPGAs are placed in columns distributed evenly across the FPGA die [3]. Since these resources are not abundant and are kept separate (and sometimes far) from where the combinational logic is implemented (slices), moving data to/from them can become a problem for the final system performance.

Pipelining and careful place and route (PAR), such as placing DAGs near to the RAMs, need to be considered in order to achieve the performance constraints of the LTE-Advanced PHY, especially for FPGAs, where the maximum frequency is in the order of 500MHz [2]. This theoretical value is hardly achieved in complex applications and is common to have a decrease of 50 to 100MHz in system performance even when using performance design techniques.

V. Results And Discussion

The results shown in this section are based on the implementation of the architecture proposed in a Xilinx Virtex-6 XC6VLX75T-1FF484 FPGA [16]. To synthesize and implement the proposed architecture, we used the ISE 13.3 design suite from Xilinx. All synthesis and implementation options were set to its default values. I/O pins were free to float, since in a full channel coding processing chain, the rate matching will not have any external connections. Some user constraints were designed to assure best performance.

Table II and III presents a summary of the implementation results of the sub-block interleaver DAG architecture. Values of slices registers, LUTs, total number of occupied slices and memory resources are presented. The performance achieved for these modules individually was 520 and 425MHz, respectively.

Depending on the choices made in hardware description and on some synthesis and implementation options of the synthesis tool, as well as any user constraints, it is possible to have slightly different results from the ones presented here.

TABLE II. SUB-BLOCK INTERLEAVER DEVICE UTILIZATION.

Slice Logic	***Used***	***Available***	***Utilization***
Number of Slice Registers	*21*	*93120*	*1%*
Number of Slices LUTs	*36*	*46560*	*1%*
Number of Occupied Slices	*15*	*11640*	*1%*
Number of RAMB18	*0*	*312*	*0*

Table III shows the implementation results for the bit collection and selection modules. The results are shown together because it makes more sense to see this two elements as a single component, since they read and write to/from the same memory.

TABLE III. BIT COLLECTION AND SELECTION DEVICE UTILIZATION.

Slice Logic	***Used***	***Available***	***Utilization***
Number of Slice Registers	*147*	*93120*	*1%*
Number of Slices LUTs	*189*	*46560*	*1%*
Number of Occupied Slices	*68*	*11640*	*1%*
Number of RAMB18	*3*	*312*	*1%*

Although the utilization of the device is only 1%, it is important to remember that, first, this FPGA device is a large one, and second, the choice of this device was made to implement the whole PHY engine. We limited the discussion only to the rate matching procedure in this paper.

For a complete rate matching procedure, including data signaling and control flow, Table IV shows the final

implementation results. This architecture, with careful placement and pipelining, was able to achieve a maximum frequency of 418 MHz and occupied a total of 218 slices and 9 RAMB18 resources (6 RAMB to store data and 3 RAMB to handle the parameters of each CB, such as size *K*, initial offset index, total bits, base entry address, number of rows and dummy bits).

TABLE IV. RATE MATCHING DEVICE UTILIZATION.

Slice Logic	***Used***	***Available***	***Utilization***
Number of Slice Registers	*454*	*93120*	*1%*
Number of Slices LUTs	*562*	*46560*	*1%*
Number of Occupied Slices	*218*	*11640*	*1%*
Number of RAMB18	*9*	*312*	*2%*

Of the 218 occupied slices, only 83 slices (38%) are directly employed in implementing the basic components of the rate matching procedure. The remaining slices are used to implement the control path of the procedure, such as generating frame signaling (start, stop, valid) for each bit stream, controlling against buffer overflow, managing the parameters across the different hardware components, synchronizing streams across block boundaries, and so on. This extra logic can be reduced or increased based on the level of control and status information desired for the procedure.

Even with very broad control logic, still a very low occupancy rate was reached, allowing other FPGAs with lower cost to be employed.

There are two points to pay careful attention when choosing a low-cost FPGA: 1) the maximum frequency, since low-cost FPGAs tend to have the worst timing characteristics and 2) slice count could increase for technologies where LUT sizes are lower than 6-bit inputs, such as the Virtex-6 used in this demonstration.

VI. CONCLUSIONS

This work presented an optimized rate matching architecture for a LTE-Advanced FPGA-based PHY. The purpose of the paper was to show how to use efficiently the resources available on FPGA architecture to implement this specific 3GPP LTE generic procedure, allowing the application to run on a high frequency rate with very low resource count. The results presented here enabled this rate matching architecture to run at 418MHz with 218 occupied slices and 9 block RAMs in a Virtex-6 XC6VLX75T-1FF484 FPGA.

ACKNOWLEDGMENT

The authors thanks the support given to this work, developed as part of the RASFA project, financed by the Fundo de Desenvolvimento das Telecomunicações FUNTTEL, from the Brazilian Department of Communication, through a partnership no. 01.09.0631.00 with FINEP – Finaciadora de Estudos e Projetos.

REFERENCES

[1] 3rd Generation Partnership Project, "3GPP TS 36.212 version 10.6.0 Release 10: Multiplexing and Channel Coding", July, 2012.

[2] Xilinx, "Virtex-6 Family Overview Datasheet", Jan, 2012.

[3] Xilinx, "Virtex-6 FPGA Memory Resources User Guide", April, 2011.

[4] Long Yu, et. Al, "An Improved Rate Matching Algorithm for 3GPP LTE Turbo Code", Third International Conference on Communications and Mobile Computing, IEEE Computer Society, 2011.

[5] Jung-Fu (Thomas) Cheng, et. Al, "Analysis of Circular Buffer Rate Matching for LTE Turbo Code", Vehicular Technology Conference, 2008. VTC 2008-Fall. IEEE 68th.

[6] Chixiang Ma, Ping Lin, "Efficient Implementation of Rate Matching for LTE Turbo Codes", IEEE, 2010.

[7] Josep Colom Ikuno, Stefan Schwarz, Michal Simko, "LTE Rate Matching Performance with Code Block Balancing", 17th European Wireless Conference: EW2011 27–29 April 2011, Vienna, Austria.

[8] R1-072137, Motorola, "Turbo rate-matching in LTE", 3GPP RAN1#49bis, Kobe, Japan, May 07-11, 2007.

[9] R1-072452, Ericsson, "Performance Evaluation of Rate Matching Algorithms", 3GPP RAN1#49bis, Kobe, Japan, May 07-11, 2007.

[10] R1-070054, Motorola, "Contention-free Interleaver designs for LTE Turbo Codes", 3GPP RAN1#47bis, Sorrento, Italy, January 15-19, 2007.

[11] 3GPP LTE Advanced homepage, http://www.3gpp.org/lte-advanced, accessed on March 06, 2013.

[12] 3GPP LTE homepage, http://www.3gpp.org/Technologies/Keywords-Acronyms/LTE, accessed on March 05, 2013.

[13] J. Hagenauer, "Rate-compatible punctured convolutional codes (RCPC codes) and their applications", IEEE Transactions on Communications, Apr. 1988.

[14] I. Sohn and S. C. Bang, "Performance studies of rate matching for wcdma mobile receiver", in Vehicular Technology Conference, 2000. IEEE VTS-Fall VTC 2000. 52nd, 2000.

[15] J. C. Ikuno, C. Mehlf`urer, and M.Rupp, "A novel LEP model for OFDM systems with HARQ", in Proc. IEEE International Conference on Communications (ICC) 2011, June 2011.

[16] Virtex 6 family homepage, http://www.xilinx.com/products/silicon-devices/fpga/virtex-6/index.htm, accessed on March 05, 2013.

[17] Ian Kuon and Jonathan Rose, "Measuring the Gap between FPGAs and ASICs", FPGA'06, February 22–24, 2006, Monterey, California, USA

[18] Small Cell Forum, "LTE eNB L1 API Definition", Oct. 2010.

[19] Figueiredo, F. A. P; Lenzi, K. G.; Filho, J. A. B.; Figueiredo, F. L. "LTE-Advanced Channel Coding Procedures: A High-level Model to Guide Low-level Implementations", IEEE WTS 2013.

[20] 3rd Generation Partnership Project, "3GPP TS 36.213 version 10.6.0 Release 10: Physical Layer Procedures", July, 2012.