Code Block Segmentation Hardware Architecture for LTE-Advanced

Karlo G. Lenzi, José A. B. Filho and Felipe A. P. Figueiredo
DRC – Convergent Networks Department
CPqD – Research and Development Center on Telecommunication
Campinas, SP - Brazil
{klenzi, jbianco, felipep}@cpqd.com.br

Abstract — **A very efficient algorithm and hardware architecture for code block segmentation used on LTE-Advanced channel coding physical layer (PHY) is presented in this paper. Code block segmentation is a generic procedure which is commonly applied before turbo encoding. Its main function is to fragment a large transport block into smaller code blocks. This approach reduces memory requirements of the turbo code interleaver, without compromising its coding gain, since turbo encoder improves its performance as the size of the code block increases. The current work presents not only an optimized procedure with reduced computational complexity, but also an architecture with very low resource count, regarding ASIC or FPGA implementations, performing at a maximum frequency of 351 MHz on a FPGA architecture.**

Keywords: LTE, code block segmentation, hardware design, FPGA.

I. Introduction

The computational complexity of today's algorithms is pushing hardware architectures to its limits. Algorithms specially developed for LTE [1], Software-Defined Radio (SDR) [5] and High Performance Computing (HPC) [6] architectures for instance are forcing hardware designers to make the most from current technologies. In this context, General Purpose Processors (GPP), Graphic Processing Units (GPU), Digital Signal Processors (DSP), Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs) are striving to support current requirements. Of those, due mainly to its flexibility and parallelism, FPGAs and ASICs are a feasible choice for such strict performance requirements, compared to other solutions [7].

Due to this gap left by modern CPUs, flexible hardware architectures are making their way into today's complex solutions, whether as an accelerator for common GPP architectures or even as a complete embedded solution.

Let us consider the LTE-Advanced as an example, which is a 4G wireless standard for broadband access. This technology can reach data rate peaks of one gigabits per second [8] in downlink. The high throughput demands intensive computational operations and therefore, without a careful design of its hardware architecture, it will be difficult to achieve the requirements of such standard. Currently, there are some solutions for LTE-Advanced PHY available on the market [9-12].

From channel coding to modulation, LTE-Advanced PHY presents several design challenges to engineers and researchers, since it aggregates the most recent and efficient techniques available in wireless communication. Most of these processing blocks are stream-based, which are very suited for FPGAs architectures, however, the code block segmentation does not have such characteristic. In fact, code block segmentation is described as a pseudo-code, having conditional branches and loops in it.

Opposed to GPPs and DSPs, which take a set of instructions as an input to describe its behavior, with good support of high level languages to describe branches and loops, modeling complex algorithmic behaviors to hardware is not a simple task, specially if these algorithms are dominated by sequential and control logic. In this case, if the design of the architecture is not well planned, performance and silicon area will be greatly impacted. A well-defined architecture can improve performance and reduce both power consumption and area, which in turn will cut down costs of ASIC fabrication or to allow a lower end device in the case of FPGAs.

In this paper we introduce a very efficient hardware architecture for a code block segmentation scheme, defined by the 3GPP standard [2] and applied in the channel coding of LTE-Advanced. This procedure involves some tasks that are very well suited for GPPs or DSPs, since it is mainly composed of sequential and control flow logic. After code block segmentation, most of the procedures listed in [2] are stream based processing, which is straightforward to FPGAs and ASIC designs. If not well designed, code block segmentation can become a bottleneck to the performance of the system, which in such high bitrates architecture can compromise the entire application.

This paper is organized as follows. First, details of the code block segmentation procedure are given in Section II. Section III presents the hardware architecture proposed, its components, control signals and data flows. In Section IV we present some optimizations made to the original procedure so that hardware design can be simplified, keeping low resource count and high-frequency rate. Section V discusses and presents the results of such architecture when implemented in a Virtex-6 FPGA [3].

II. LTE Block Segmentation

The LTE code block segmentation is a generic procedure defined by 3GPP TS 36.212 "Multiplexing and Channel Coding" Standard [2] which breaks an input transport block (TB) plus CRC (Cyclic Redundancy Check) of arbitrary size greater than 6144 bits, into a series of smaller code blocks (CB), whose sizes (K) are selected from a predefined table, before turbo encoding [28, 29] can be performed.

Although turbo codes improve their performance as the code block size K increases, after a certain value, these improvements become insignificant (< 0.05 dB coding gain between 8192-bit and 12800-bit code block sizes) [13]. Hence, to reduce memory requirements without compromising its coding gain, code block segmentation is applied. Table I presents the code block sizes selected for the LTE 3GPP Std.

Table I. Code block sizes (K) For Transport Block Segmentation.

40	48	56	64	72	80	88	96	104	112
120	128	136	144	152	160	168	176	184	192
200	208	216	224	232	240	248	256	264	272
280	288	296	304	312	320	328	336	344	352
360	368	376	384	392	400	408	416	424	432
440	448	456	464	472	480	488	496	504	512
528	544	560	576	592	608	624	640	656	672
688	704	720	736	752	768	784	800	816	832
848	864	880	896	912	928	944	960	976	992
1008	1024	1056	1088	1120	1152	1184	1216	1248	1280
1312	1344	1376	1408	1440	1472	1504	1536	1568	1600
1632	1664	1696	1728	1760	1792	1824	1856	1888	1920
1952	1984	2016	2048	2112	2176	2240	2304	2368	2432
2496	2560	2624	2688	2752	2816	2880	2944	3008	3072
3136	3200	3264	3328	3392	3456	3520	3584	3648	3712
3776	3840	3904	3968	4032	4096	4160	4224	4288	4352
4416	4480	4544	4608	4672	4736	4800	4864	4928	4992
5056	5120	5184	5248	5312	5376	5440	5504	5568	5632
5696	5760	5824	5888	5952	6016	6080	6144		

Two important aspects were taken into consideration by the 3GPP workforce due to the diminishing returns of turbo encoders for very large block sizes: 1) to limit the maximum block size K, since performance do not improve significantly, but storage requirements do as the block size grows (due to the QPP interleaver of the turbo encoder) [2], and 2) to keep the table entries small, so that the access time to search and retrieve code block sizes is as fast as possible.

When segmentation is applied, an additional CRC sequence of length L = 24 bits is attached to each new CB. The CRC used in code block segmentation differs from the CRC used with the TB in order to improve error detection capability. As opposed to the CRC-24A used and defined by [2] on the input TB [15, 16], the CRC-24B [14] is used in each segment CB. A comparison of some error detection CRC code standards is presented in [17]. If the TB size is smaller than the total amount of CB lengths selected during the segmentation procedure, filler bits (nulls) are padded at the beginning of the first CB.

Code block segmentation can be divided into two steps, computing some segmentation parameters and filling CBs with the TB information. For computing these parameters, code block segmentation follows the procedures presented by the pseudo-code listed in (a) and (b). The filling of CBs will not be presented in this paper. For more information, see [2].

Before discussing these pseudo-codes, it is important to define such parameters. The maximum block size defined by the 3GPP 36.212 Std. [2] is Z = 6144 bits. C is one of the outputs of this procedure and represents the number of code block segments. B is equal to the size of the TB plus L, where L is the length of the CRC sequence and is equal to 24 in this case. The input block size B is arbitrary and may range from a few bits to thousands of bits, serving as the only input parameter to the block segmentation procedure. B' is the new block size considering all additional CRCs after segmentation. At last, F represents the number of filler bits padded at the beginning of the first code block (if necessary).

Listing (a). Determining the number of code blocks.

```
if $B \leq Z$
   $L = 0$
   $C = 1$
   $B' = B$
else
   $L = 24$
   $C = \text{ceil}\,[B / (Z - L)]$
   $B' = B + C * L$
end if
```

From pseudo-code (a), block sizes lower than 6144 bits do not need segmentation, since a single CB from Table I is able to store the entire TB plus CRC data. On the other hand, if B is greater than 6144 bits, segmentation is applied and a new CRC sequence is appended to each CB segment.

The number of segments (or blocks) C is determined from the current block size B, the maximum block size Z and the CRC length L. A ceiling operation is applied to guarantee that the number of CBs is sufficient to store all the original information.

The pseudo-code presented in (b) shows how to select from Table I, based on C and B', the segmented code block size K.

Listing (b). Selection of K_+ and K_-.

```
$K_+$ = minimum K in Table I such that $C * K \geq B'$
if $C = 1$
   $C_+ = 1, K_- = 0, C_- = 0$
else if $C > 1$
   $K_-$ = maximum K in Table I such that $K < K_+$
   $\Delta_K = K_+ - K_-$
   $C_- = \text{floor}\,[(C * K_+ - B') / \Delta_K]$
   $C_+ = C - C_-$
end if
$F = K_+ * C_+ + K_- * C_- - B'$
```

After computing its output parameters,. i.e., the size of the two segments (K_+ and K_-, where $K_- < K_+$), the number of segments of each size (C_+ and C_-) and the number of filler bits F, the next step is to fill these segments, starting with K_- and moving up to the K_+ blocks, with the original data bits from the TB (appending filler bits at the beginning of the first CB if needed) and to concatenate a new CRC sequence at the end of each CB) [2]. Once each of the CBs is filled with information, they are moved forward to the turbo encoder for processing.

Fig. 1 and 2 illustrates both listings, (a) and (b), in a control and dataflow graph (CDFG) for clarification of the code block segmentation procedure.

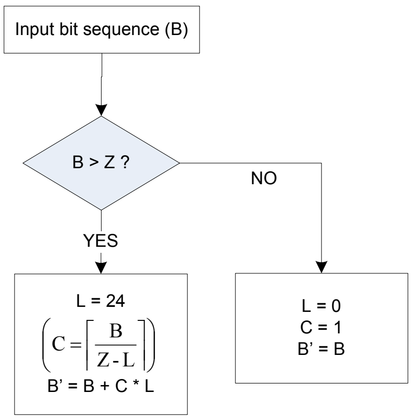
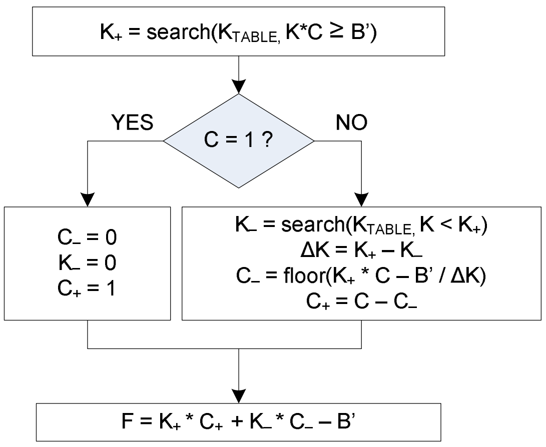


Figure 1. Determining the number of code blocks.



K_+ = search(K_{TABLE}, $K*C \geq B'$)
YES
C = 1 ?
NO
$C_- = 0$
$K_- = 0$
$C_+ = 1$
K_- = search(K_{TABLE}, $K < K_+$)
$\Delta K = K_+ - K_-$
C_- = floor($K_+ * C - B' / \Delta K$)
$C_+ = C - C_-$
$F = K_+ * C_+ + K_- * C_- - B'$

Figure 2. Selection of K_+ and K_-.

There are two important aspects that should be kept in mind regarding the procedure listed in (b). First, there are apparently two search procedures: one for the minimum K_+ and one for the maximum K_-. However, taking a closer look, it is possible to conclude that K_+ and K_- are adjacent code block sizes and therefore, a single search is enough to retrieve both block sizes. Second, Δ_K is always a power of two, since K_+ and K_- are adjacent and all blocks sizes are spaced by multiples of 8 bits, making all blocks sizes byte-aligned.

From Table I, notice that blocks ranging from 40 to 512 bits are multiple of 8. Blocks from 512 up to 1024 bits long are multiple of 16. From sizes of 1024 up to 2048 bits, blocks are multiple of 32 and blocks ranging from 2048 to 6144 are multiple of 64.

The use of adjacent code block sizes is supported by two main reasons: 1) by doing so, the number of filler bits is minimized, which in turn reduces the burden of the decoder due to padding; and 2) it limits the worst-performing segment, i.e., the smallest code block segment. These observations will lead to significant hardware improvements which will be presented in section IV.

III. Proposed Hardware Architecture

The proposed hardware architecture for the LTE block segmentation procedure is depicted in Fig. 3.

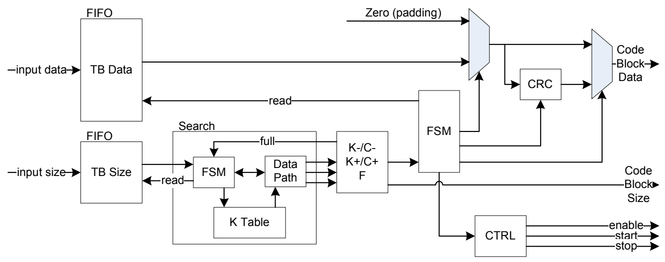


Figure 3. Code Block Segmentation Proposed Hardware Architecture.

The hardware structure proposed is composed of 2 input FIFOs, one to store the input bit stream and the other to store its size; a search block, that seeks values from Table I (which are stored in a ROM memory) for a possible code block size candidate (K_+ and K_-); another FIFO to store the output parameters of the block segmentation procedure, namely the number of code blocks (C_+ and C_-), their sizes (K_+ and K_-) and if any, the number of filler bits (F); a finite-state machine (FSM) that will use these output parameters to control the filling of each CB, the insertion of filler bits (if necessary), the attaching of a new CRC-24B to each code block and the generation of auxiliary control signals, such as start of frame end of frame and data valid, for the turbo encoder.

The most complex logic block of this architecture is the search block, which requires procedures listed in (a) and (b) to be resolved. The remaining hardware components are straightforward in the way they should be employed in the design. The search procedure has two steps: first, determine if segmentation is needed, and if it is, determine the number of code blocks C. The second step is to compute the number of blocks C_+ and C_-, the size of segments K_+ and K_- and the filler bits F. To implement this procedure, a FSM+D (finite-state machine plus data path) is employed.

The segmentation procedure starts with an input data stream representing an entire TB plus CRC-24A. Both the data stream and the stream size are stored for later processing. The buffering allows for multiple data streams of different TBs to be sent for processing without needing to wait for the segmentation of the previous TB to be completed. It is necessary to allocate an adequate buffer size to support multiples TBs and to avoid memory overflow. The block segmentation controller (FSM) keeps track of all stored TBs and processes them as soon as possible.

If no block segmentation is needed, i.e., TB plus CRC-24A size is lower than 6144 bits, all input data is stored in a single CB, adding filler bits if necessary, and then sent to the turbo encoder. Proper control signals are generated to interface the code block segmentation procedure with the turbo encoder.

If the TB plus CRC-24A size is greater than 6144 bits, segmentation is applied. The search block starts looking for an adequate number of code blocks C for segmentation, as well as their sizes (K_+ and $K_–$). To implement the search procedure, several algorithms are available. In this architecture it was chosen the binary search [18]. Since we have 188 entries in Table I, we need ceil($\log_2$(188)) = 8 bits to address this ROM memory with a databus width of ceil($\log_2(K_{MAX})$) = 13 bits. A latency of no longer than 8 cycles is enough to find the correct block size based on the size of B' and the number of code blocks C.

Once we found the values of C_+ and $C_–$; K_+ and $K_–$; F, the FSM (block controller) makes sure each code block is filled with the necessary information bits from the original bit stream, inserting filler bits if needed and attaching the CRC-24B at the end of each code block.

IV. Design Optimization

The first optimization proposed is related to the search block. This block computes the number of code blocks C and the new block size B'. To do so, it is necessary according to listing (a) to perform a division operation of the block size B by (Z – L). Since Z and L are constants, a strength reduction technique may be applied, and the expression can be rewritten as a multiplication by a constant P as presented by (1).

$$P = 1 / (Z – L) = 1 / 6120 \qquad (1)$$

Further improvements of (1) may be accomplished by scaling the constant P by an integer factor of 2^{27} (or a shift by 27 to the left), which would transform this multiplication in an integer one. In a full precision fixed-point representation it would be necessary 65 bits to store this constant without quantization error. There are 12 leading zeros before the first significant bit in a fixed-point fractional representation. Thus, shifting eliminates these leading zeros and a final 15-bit integer P constant is obtained.

Limiting the precision of P to 15 significant bits would give an error of 9.73×10^{-12} to the full precision representation, which is lower than a single precision floating-point quantization error [19]. This strategy would eliminate the need for fixed-point arithmetic (integer and fractional), reducing the word-length of operations, and consequently reducing hardware complexity.

To obtain the final (Z-L) division result, it is necessary to scale back this value with a simple shift to the right of 27 bits after multiplying B by P. It is worth mention that shift operations do not consume any hardware resources, since at gate level it only represents a hardwire re-routing. The ceiling operation is done by comparing if the least-significant 27 bits are different from zero. If true, add one to the final result.

It is also possible to apply strength reduction to the multiplication by L operation listed in (a), which can be re-write as (2).

$$C * L = (C * 16 + C * 8) = (C * 2 + C) * 8 \qquad (2)$$

Since L = 24, both multiplications are powers of two, and therefore can be implemented as shift operations. This way, the multiplication can be replaced by a shift and add operation.

Regarding listing (b), two optimizations are possible. The first is a direct consequence of the two code block sizes, K_+ and $K_–$, being adjacent blocks. After searching for the K_+ address in the ROM memory that stores Table I, $K_–$ address can be obtained immediately by decrementing the K_+ address by one.

The second optimization relates to the Δ_K parameter. Δ_K is computed by the subtraction of K_+ by $K_–$. Since both block sizes are adjacent, and since all code block sizes are byte-aligned, the results of that operation will always be 8, 16, 32 or 64 (as long as Table I do not change in future releases).

All of these values are powers of two, which transform the Δ_K division operation, once again, into a shift to the right by $\log_2(\Delta_K)$. No hardware resources are required to implement this division. A multiplexer decides among the 4 possible outputs of that division, according to the value of Δ_K (observe that Δ_K is one-hot encoded). The output of the multiplexer will be $\log_2(\Delta_K)$, assuming one of the four possible values.

The last optimization of this architecture is that it is possible to apply copy propagation to the expression C * K used on the search for K_+. At the end of the search, K = K_+, thus producing C * K_+, which is used once again (it is a copy) during the computation of $C_–$. This saves a multiplication operation from the original procedure.

This copy propagation can also be used if we rewrite the filler bits expression as follows:

Listing (c). Optimized filler bits computing

```
$F = K_+ * C_+ + K_– * C_– – B'$
$F = K_+ * (C – C_–) + K_– * C_– – B'$
$F = K_+ * C – K_+ * C_– + K_– * C_– – B'$
$F = [K_+ * C – B'] + C_– * (K_– – K_+)$
$F = [K_+ * C – B'] – C_– * (K_+ – K_–)$
$F = [K_+ * C – B'] – [C_– << \log_2(\Delta_K)]$
```

Listing (c) presents an efficient way to compute the filler bits. Rewriting the expression, it is possible to see that both values of K_+ * C – B' and $\log_2(\Delta_K)$ were already computed in previous iterations, therefore copy propagation may be applied and, in the end only a subtraction operation needs to be performed.

With these optimizations, the code block segmentation can be implemented only with integer operations without any need to support fixed or floating-point types. Another improvement is that no division circuit is needed to design the hardware architecture, which saves logic resources and reduces latency.

To summarize all optimizations suggested in this section, Fig. 4 and 5 presents the CDFG for an optimized code block segmentation. Fig. 4 illustrates the first part of code block segmentation, related to listing (a). Fig. 5 depicts the second part of the procedure, related to listing (b).

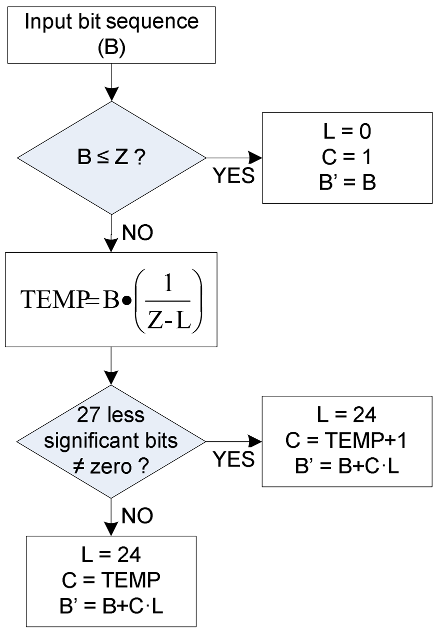


Figure 4. Optimized computing of number of code blocks.

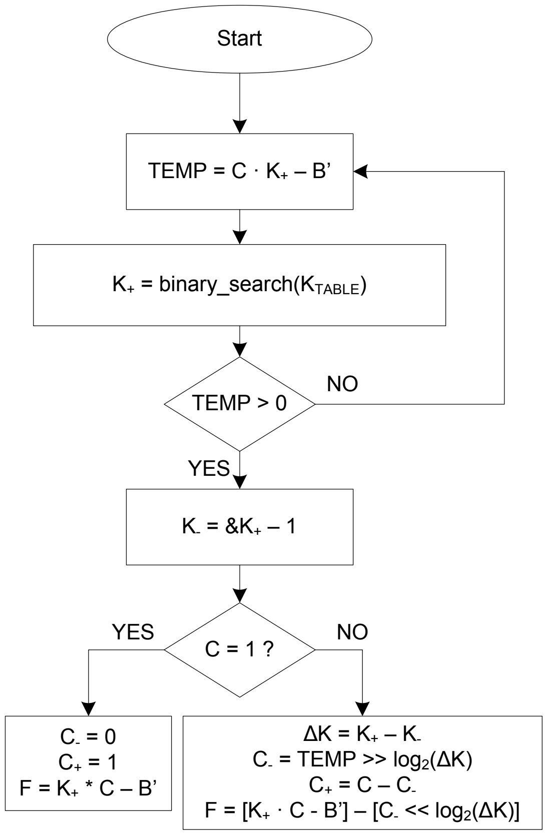


Figure 5. Optimized computing of K_+ and K_-.

V. Results & Discussions

The results shown in this section are based on the implementation of the architecture proposed for the code block segmentation on a Xilinx Virtex-6 XC6VLX75T-2FF484 FPGA [3, 4]. For synthesis and implementation, ISE 13.1tool from Xilinx was used [20] with default settings.

The architecture proposed may assume a serial TB input bit stream or a parallel (byte or word) stream. For the core processing of the code block segmentation, it makes no difference how data is received, since it only needs the TB plus CRC size to perform its function.

The choice between serial or parallel data input will impact on the depth of the input FIFO buffers and how the CRC computation is implemented. There are several papers showing how to design parallel or serial CRC circuits [21-26]. Depending on the scheme chosen for the architecture, CRC should follow the same approach (serial or parallel).

The complexity of the FSM could increase if a parallel architecture is chosen. This increase in complexity arises from handling the number of filler bits. If the number of filler bits of a code block is not a multiple of a byte, the FSM would need to byte-align the data flow to guarantee the correct operation of the circuit. Since the LTE-Advanced FemtoForum API interface between L2/L1 layers [27] will deliver byte aligned words, this will not be an actual problem.

Table II presents the resource utilization after implementation of a serial TB input streaming for the code block segmentation. It summarizes the main implementation results obtained from this architecture. The number of registers, LUTs, occupied slices, memory resources and DSP resource blocks are presented.

TABLE II. CODE BLOCK SEGMENTATION DEVICE UTILIZATION.

Slice Logic	*Used*	*Available*	*Utilization*
Number of Slice Registers	*636*	*93120*	*1%*
Number of Slices LUTs	*590*	*46560*	*1%*
Number of Occupied Slices	*190*	*11640*	*1%*
Number of RAMB18	*3*	*312*	*1%*
Number of RAMB36	*2*	*156*	*1%*
DSP48E	*1*	*288*	*1%*

From the results of Table II, it can be pointed out that 5 memories of the type RAM Block (RAMB) are used. One to store the input data stream, one for the TB size, one for the code block C_+ and K_+, one for the code block C_-, K_- and F, and the last one for storing the values from Table I. The DSP48 resource, a multiply-and-accumulate (MAC) unit of 48 bits wide, is used to implement the C * K_+ multiplication, since it demands a large bit width to maintain full precision. With this architecture only 190 slices were used to implement all the block segmentation logic.

The maximum frequency achieved with this implementation, assuming default settings, is 351 MHz. Performance results may be improved if a more thorough back-end design, specially on place and route (PAR), is carried out. The optimizations made in this architecture focused exclusively on front-end design (RTL - register-transfer level).

VI. Conclusion

In this paper we present an optimized code block segmentation architecture for LTE-Advanced channel coding PHY. The purpose of the paper is to show how to efficiently implement this architecture in hardware, ASIC or FPGA, making good use of resources and allowing high-frequency operations. Block segmentation in heavily based on sequential and control, making the hardware design more challenging for parallel architectures, such as FPGAs. The proposed architecture was implemented on a Virtex-6 FPGA for analysis. The results achieved here enabled the code block segmentation architecture to run at a maximum frequency of 351 MHz with 190 occupied slices and 5 block RAMs.

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 the partnership no. 01.09.0631.00 with FINEP – Finaciadora de Estudos e Projetos.

References

[1] 3rd Generation Partnership Project, Technical Specifications Series 36 for E-UTRA (Release 10), available online at http://www.3gpp.org.

[2] 3rd Generation Partnership Project, “3GPP TS 36.212 version 10.6.0 Release 10: Multiplexing and Channel Coding”, July, 2012.

[3] Xilinx, “Virtex-6 Family Overview Datasheet”, Jan, 2012.

[4] Xilinx, “Virtex-6 FPGA Memory Resources User Guide”, April, 2011.

[5] Wireless Innovation Forum, http://www.wirelessinnovation.org, accessed on September 13, 2012.

[6] Charles Severance and Kevin Dowd, “High Performance Computing”, O'Reilly Media, Second Edition (July 9, 1998).

[7] Stratix IV FPGA High-Performance DSP Features, http://www.altera.com/devices/fpga/stratix-fpgas/stratixiv/overview/architecture/stxiv-dsp-block.html, accessed on September 13, 2012.

[8] 3GPP LTE Advanced homepage, http://www.3gpp.org/lte-advanced, accessed on September 06, 2012.

[9] Texas Instruments, “Enabling LTE development with TI’s new multicore SoC architecture”, Feb. 2010.

[10] Aricent, “LTE eNodeB PHY Framework”, http://www.aricent.com/software/lte-enodeb-phy-framework.html, accessed on September 14, 2012.

[11] Picochip, “Developing LTE Small Cell with Picochip (MWC2011)”, http://www.picochip.com, 2011.

[12] Ubiquisys, http://www.ubiquisys.com, accessed on September 14, 2012.

[13] Motorola, "Code Block Segmentation for LTE Channel Coding", R1-071059, 3GPP TSG RAN WG1 #48, Feb. 2007; XP-050105053.

[14] R1-074848, Ericsson, “CRC Computation Method”, 3GPP RAN1#51bis, Jeju, Korea, November 05-09, 2007.

[15] R1-074473, Ericsson, ETRI, ITRI, LGE, Motorola, Nokia, Nokia Siemens Networks, Nortel, Qualcomm, Samsung, ZTE, “TB CRC Generator Polynomial,” 3GPP TSG RAN WG1#50b, Shanghai, China, Oct. 8 - 12, 2007.

[16] R1-074448, Qualcomm Europe, “Generator polynomial for transport block CRC,” 3GPP TSG RAN WG1#50b, Shanghai, China, Oct. 8 — 12, 2007.

[17] K. A. Witzke and C. Leung, “A comparison of some error detection CRC code standards,” IEEE Trans. on Comm., vol. 33, no. 9, pp. 996--998, Sep. 1985.

[18] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, “Introduction to Algorithms”, (1st ed.). MIT Press and McGraw-Hill, 1990. ISBN 0-262-03141-8.

[19] B. Widrow and I. Kollár, "Quantization Noise: Roundoff Error in Digital Computation, Signal Processing, Control, and Communications", Cambridge University Press, Cambridge, 2008.

[20] Xilinx, “ISE In-Depth Tutorial”, March, 2011.

[21] Giuseppe Campobello, Giuseppe Patane and Marco Russo, “Parallel CRC realization”, IEEE Transactions on Computers, Oct. 2003.

[22] Ming-Der Shieh, Ming-Hwa Sheu, Chung-Ho Chen and Hsin-Fu Lo, “A Systematic Approach for Parallel CRC Computations”, Journal of Information Science and Engineering 14, 445-461 (2001).

[23] Richard Black, www.cl.cam.ac.uk/Research/SRG/bluebook/21/crc/crc.html, University of Cambridge Computer Laboratory Systems Research Group, February 1994.

[24] Peterson, W. W. and Brown, D.T., “Cyclic Codes for Error Detection”, In Proceedings of the IRE, January 1961, 228–235.

[25] T. V. Ramabadran and S. S. Gaitonde, “A tutorial on CRC computations”, IEEE Micro, vol. 8, no. 4, pp. 62–75, 1988.

[26] M. Walma, “Pipelined cyclic redundancy check (CRC) calculation”, in ICCCN’07: Proceedings of 16th International Conference on Computer Communications and Networks, 2007, pp. 365–370.

[27] FemtoForum API, "LTE eNB L1 API Definition v1.1", FemtoForum Technical Document, December, 2010.

[28] P. Adde and R. Pyndiah, "Recent simplifications and improvements of Block Turbo Codes," Proc. 2nd Int. Symp. on Turbo Codes & Related Topics ISTC’00, Brest, France, Sept. 2000, pp. 133-136.

[29] C. Leroux, C. Jego, P. Adde and M. Jezequel, "Toward Gb/s turbo decoding of product codes onto an FPGA device," Proc. IEEE Int. Symp. on Circuits And Systems ISCAS'07, pp. 909-912, May 2007.