Design And Implementation of a Configurable Interleaver/Deinterleaver for Turbo Codes in 3GPP Standard

Héctor Borrayo-Sandoval and R. Parra-Michel
Department of Elec. Eng.
CINVESTAV-Gdl, México.
hborrayo@gdl.cinvestav.mx

Luis F. González-Pérez and Fernando Landeros Printzen
Electronic Design Center
ITESM-Gdl, México
gonzalez.luis@itesm.mx

Claudia Feregrino-Uribe
Department of Computer Science
INAOE, Puebla, México
cferegrino@inaoep.mx

***Abstract*— During the last decade, Turbo codes have been taking an increasing importance in channel coding due to its good performance in error correction. One key component in Turbo codes is the interleaver/deinterleaver pair, often designed as reconfigurable coprocessors able to deal with requirements of large data length variability found in the newest communication standards. In this work we introduce a configurable interleaver architecture for the turbo decoder in 3GPP standard. It is implemented under the idea of "iterative modulo computation" presented in [1] and exploited in [2], but capable to handle all the data-length configurations. Additionally, the presented solution not only generates the interleaved addresses, but also deals with the flow of data streams through the interleaver. The architecture and FPGA implementation results are also presented.**

Keywords- 3GPP Turbo code; Configurable Interleaver.

I. INTRODUCTION

Nowadays, communications systems have been growing and developing very quickly. Applications in satellite communications, wireless networks, mobile telephony, internet, etc. continuously demand an increase in bit rates, while lowering bit error rates and power consumption. In this context, channel coding schemes have been developed and consist basically in adding redundancy bits to the original information that are utilized later on in the detection process by the receiver. Among those schemes, Turbo codes, that were introduced in 1993 by C. Berrou in [3], have become a popular technique in communications research and a key component in recent communications standards (such as 3GPP mobile phones and recently, LTE standard), mainly because its performance is near to Shannon coding limit.

Turbo codes' high performance in error correction is heavily related to both iterative decoding and the use of interleavers. Interleavers help to increase the minimum distance in the code distance spectrum, which is intimately related to the correction capability of the code [4]; therefore, designing Turbo codes requires a meticulous design of its constituent interleaver.

An interleaver is a device that rearranges the order of data or bit sequences in a one-to-one pseudo random format. The inverse operation to interleaving is called deinterleaving, devoted to restore the received sequence into its original order. The mapping rules for the interleaver/deinterleaver (I/D) processes are usually given in form of equations, tables and special architectures [5].

In Turbo codes, the interleaver is located, at the encoder side, between the two recursive systematic convolutional (RSC) encoders as shown in Fig. 1. At the decoder side, interleaver and deinterleavers are required, as shown in Fig. 2, for the Log-MAP-based iterative decoding turbo decoder, as the one presented in [6].

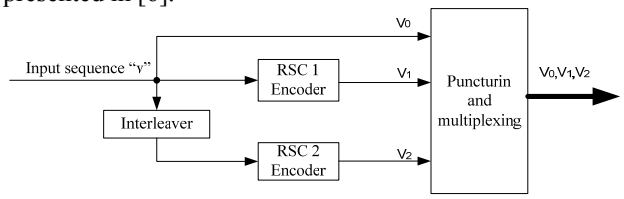

Figure 1. Turbo coder diagram

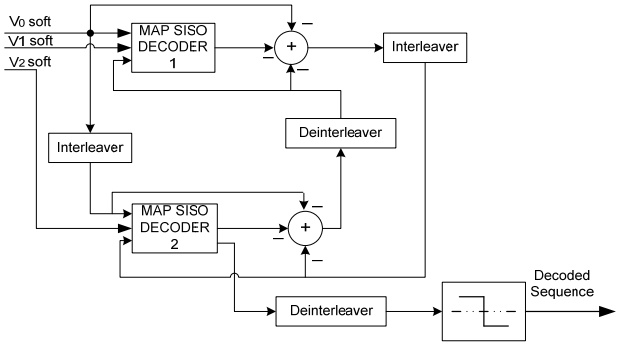

Figure 2. General structure of a turbo decoder

In this paper we focus on the design of the 3GPP standard Turbo code I/D. The main objective is to devise an architecture capable of managing every one of the 5074 different block sizes of data defined in the 3GPP standard, while maintaining low hardware complexity and small use of resources.

This paper is organized as follows: In section II, the equations defined in 3GPP standard, over which our interleaver is based on, are introduced. In section III the architecture presented in [2] and how it should be completed to manage all sizes of data specified by the standard, as well as how to deal with data and not only with addresses is discussed. Section IV presents the proposed configurable interleaver architecture. A comparison of the proposed architecture with related approaches, as well as performance results of an FPGA implementation is given in section V. Finally, section VI shows the gathered conclusions.

II 3GPP INTERLEAVER ALGORITHM

From the 3GPP standard [7], we can get the interleaver algorithm defined for turbo coding/decoding. In this algorithm input bits feed a rectangular matrix, and then some permutations are performed, where i, j are the indexing variables for row and column, starting from 0, top to bottom and left to right, respectively. The algorithm works as follows:

- Block size: *K is the integer number of input bits and takes a value from 40 to 5114.*
- Determine the number of rows, R, of the rectangular matrix, such that:

  $R=5$, if $(40 \le K \le 159)$
  $R=10$, if $((160 \le K \le 200) \text{ or } (481 \le K \le 530))$
  $R=20$, if $(K=\text{any other value})$

  rows are numbered from 0 to R-1.

- Determine the prime number p to be used in the intra permutation, and the number of columns, C, of the rectangular matrix such that:

  If(481≤K≤530)then
  p=53 and C=p
  else
  find the minimum prime number p from table 2 in the standard such that K≤R×(p+1)
  and determine C such that:
  C=p-1, if K≤R×(p-1)
  C=p, if R×(p-1)<K<R×p
  C=p+1, if R×p<K
  end if.

  Note: For every p exists a value v in the same table, so when we find p we also obtain its corresponding v.

- Construct the base sequence $S(j)$ for intra row permutation such that:

  $S(j)=(v \times S(j-1)) \bmod p, \quad j=1,2\ldots,(p-2)$
  $S(0)=1.$

  Where *mod* stands for modulo operator.

- Determine the prime integers in the sequence q_i such that:

  g.c.d$(q_i, p-1)=1$, $q_i>6$, and $q_i>q_{i-1}$ *for* $i=1,2,\ldots,R-1$.
  Where g.c.d means greatest common divisor.

- Permute the sequence q_i to make the sequence r_i such that:

  $r_i = T(q_i)$, $i=0,1,\ldots,R-1$

  where $T(i)$ is a simple indexing transformation and is defined by the standard.

- Perform the intra-row permutation as:

  If(C=p), $U_{i,j}=S[(j \times r_i) \bmod (p-1)]$
  Where $U_{i,(p-1)}=0;$
  If(C=p+1), $U_{i,j}=S[(j \times r_i) \bmod (p-1)]$
  Where $U_{i,(p-1)}=0;\ U_{i,p}=p;$
  And If(K=R×C) then
  Exchange $U_{R-1,0}$ *with* $U_{R-1,p}$
  If(C=p-1), $U_{i,j}=S[(j \times r_i) \bmod (p-1)]-1$

  Where $i=0,1,\ldots,R-1$; and $j=0,1,\ldots,p-2$.

- Perform the inter-row permutation for the rectangular matrix based on the pattern $T(i)$.

  $U_i=U_{(T(i))}$, *where* $i=0,1,\ldots,R-1$.

- Read out the addresses columnwise.

As we can see from the algorithm, it is necessary to perform different kinds of operations to achieve the interleaving pattern, some of those operations are:

- Comparison
- Addition
- Multiplication
- Permutation
- Modulo
- Calculation of g.c.d.

In order to develop our interleaver architecture, we based our design on previously reported architecture [2]. Nevertheless, this architecture was improved in some relevant aspects that will be highlighted in the following sections.

III. 3GPP INTERLEAVER ARCHITECTURE

The interleaving process can be separated in two main modes. The first one, the *pre-computation mode*, has to be performed each time a change in the block size occurs; that means, for a fixed block size K, these operations have to be performed only once. This mode computes the number of rows (R), number of columns (C), the integer prime (p), the primitive root (v), the base sequence for intra-row permutation $S(j)$, the sequence of minimum prime integers (q_i) and the permuted prime integers (r_i).

The second mode, called *run time mode*, calculates the permutation sequence $U_{i,j}$ and the interleaved address *i_addr* where we can write or read the data bits (in/from a data RAM) depending on whether interleaving or de-interleaving is performed. This *run time mode* has to be performed for each data block. Note that the set of addresses could also be computed and saved in memory only once, however, the approach presented here, based in two modes and computing addresses at *run time mode*, is what permits to reduce the implementation size of the I/D algorithm as compared to a memory or LUT (Look-Up Table) based approach, where interleaving addresses are saved in memory. Next, the two modes of the algorithm as well as their associated architectures are presented.

A. Pre-Computation Mode

First, we calculate R with logical functions. Parameters p and v are stored in pairs in a Look Up Table (LUT), just like in the standard, so we read this LUT addressed via a counter generated by the controller based on equation (1).

$$K \le R \times (p+1) \quad \rightarrow \quad (K-R) \le (R \times p) \qquad (1)$$

To perform the operations in equation (1), we use the architecture proposed in [2] and shown in Fig. 3.

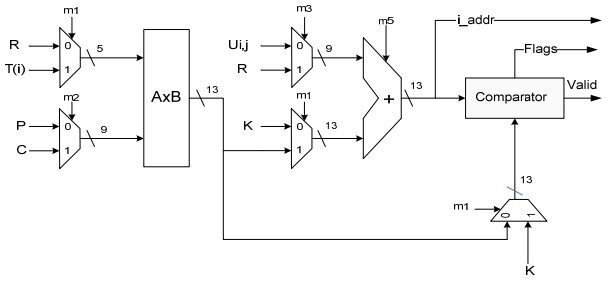

Figure 3. Hardware architecture for multiplication, addition and comparison operations

This architecture is capable of performing multiplication, addition and comparison, and it is controlled by m1-m5 flags generated by the controller.

Once p and v are obtained, we can proceed to get the number of columns (C). We know that C can take one of the three values $p-1$, p or $p+1$, and should be the minimum value that gets $(R \times C) \geq K$. The controller generates C and the architecture shown in Fig. 3 performs the comparison.

According to the algorithm, we have to calculate the prime sequences q_i. However, it can be noticed that the q_i sequences (as mentioned in [2]) are almost the same in most cases and they only differ by one or two elements from other sequences. Based on this observation, we decided to place sub groups of q_i sequence into a ROM and then choose one according to the p value.

To calculate the $S(j)=(v \times S(j-1)) mod\ p$ sequence it is mandatory to perform the modulo operation. Fortunately, this operation is presented in simplified form in [1], where an iterative numerical algorithm based basically on additions, shifts, comparisons and bit retrieval is presented. A flow diagram depicting this algorithm is presented in Fig. 4.

From Fig. 4 we can design its corresponding hardware architecture, shown in Fig. 5. With this architecture we can calculate and write in a RAM every $S(j)$ in at most four clock cycles. The number of cycles needed to calculate every $S(j)$ depends on v, which can take only six different values: 2,3,5,6,7 and 19. For v =2 and v =3 we need only one iteration, for v =5, v =6 and v =7 we need two iterations and for v =19 we need four iterations.

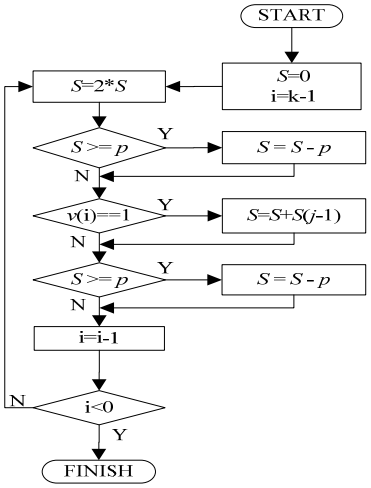

Figure 4. Flow diagram for the operation $(v \times S(j-1)) \bmod p$

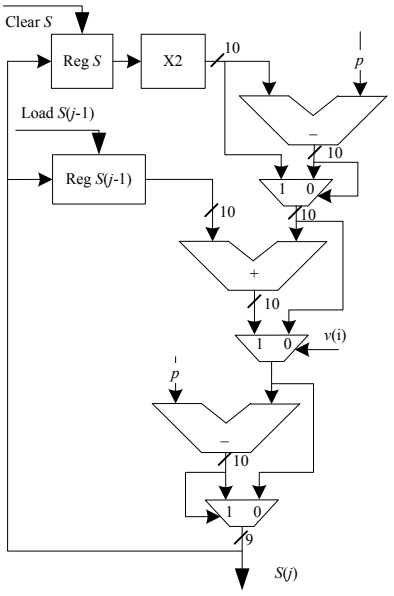

Figure 5. Hardware architecture for performing $(v \times S(j-1)) mod\ p$

B. Run Time Mode

In *run time mode* we obtain the interleaved addresses by performing the inter row $T(i)$ and intra row $U_{i,j}$ permutations, where the interrow order T(i) is stored in a LUT while the intra row order $U_{i,j}$ is stored in a RAM. With these parameters and the use of equation (2), the interleaving addresses *i_addr* are finally generated.

$$i\_addr((i \times C)+j)=(C \times T(i))+ U_{i,j} \qquad (2)$$

with i=0,1,…,R-1; j=0,1,…,C-1.

Notice that in this mode, the architecture shown in Fig. 3 is used to compute $C \times T(i)$. The result of this computation points to the first element of each of the R rows in the rectangular matrix, and then by adding $U_{i,j}$ with the same architecture we obtain a displacement along every row. An example of this is shown in Fig 6.

To obtain $U_{i,j}$ in *run time mode* we have to calculate the modulo operation $(j \times r_i) \bmod (p-1)$ which has the same form as $(v \times S(j-1)) mod\ p$ in the *pre-computation mode*. As a result, and since these operations are performed at different times, we can reuse the architecture of Fig. 5 to perform both operations with little modifications. As we can see in Fig. 5. a "Circular Buffer" needed for iterative operations and a multiplexer used to enable this buffer in *run time mode* were added. In that architecture another multiplexer was added trough which signal q_i is feed when functioning in the *run time mode.*

There already exists an architecture to compute these operations and it works fine in the *pre-computation mode*, but in *run time mode* it fails almost in 12 percent of the block sizes [2].

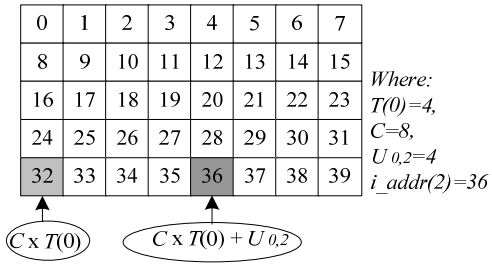

Figure 6. Example of calculating i_addr

The problem occurs when $q(i)-2p+1>0$, in this case the modulo operation cannot be performed correctly (in *run time mode*); this architecture is shown in Fig. 7.

In our interleaver we solved this problem by modifying this architecture in the piece of hardware shown in Fig. 8. With this modification the obtained hardware architecture works properly in both the *pre-computing* and *run time* modes.

From architecture of Fig. 7 and using b) from Fig. 8 instead of a), we obtain the RAM address $U_{i,j}$ to read, from equation (3).

$$Ram\_Adr(i,j) = [Ram\_Adr(i, j-1) + Q\_mod(i)] \bmod (p-1) \qquad (3)$$

Furthemore, analyzing the architecture presented in [2], it can be seen that, as most of the existing designs, it does not handle data streams but only addresses. Normally this should not represent a problem, but for 3GPP, exceptions exists in the interleaved addresses which we have to deal with. In order to solve this problem we added some blocks, the most important being a data RAM and exceptions ROM.

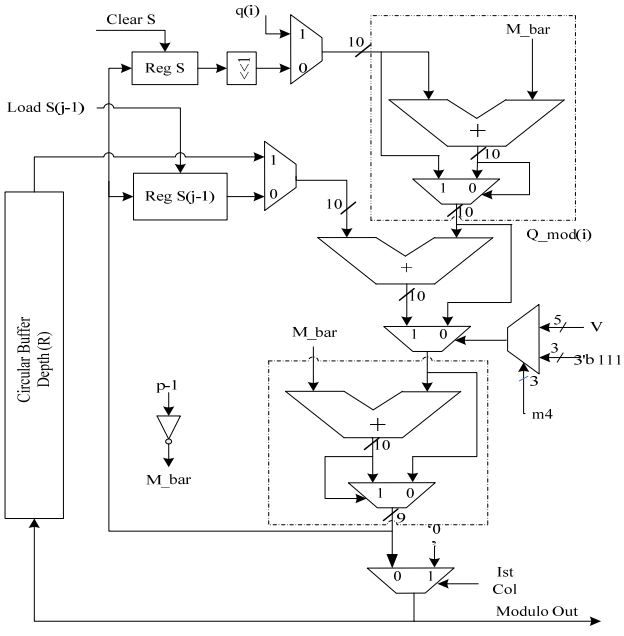

Figure 7. Architecture used in [2] for computing modulo operation in pre-computing and run time modes

IV. Proposed Architecture and Comparison With Other Interleavers

In the 3GPP standard's interleaving algorithm, there are some exceptions that are reflected in the architecture shown in Fig. 10 with an "Exceptions Handling" block. There are three exceptions, the first occurs when we generate addresses bigger than K, these addresses are tagged as invalid. For example, for K=41 the rectangular matrix size is 5x10, then there would be 9 invalid addresses.

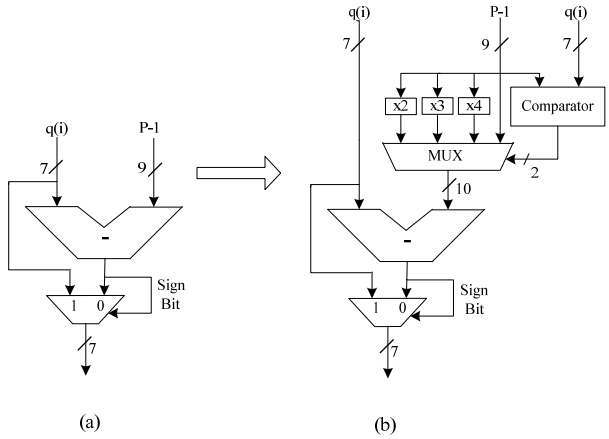

Figure 8. a) Hardware architecture used in [2] to obtain Q_mod(i) =q(i)mod(p-1), b) Modified architecture.

Because of that, we cannot generate a valid address per clock cycle (see Table 1). The second exception occurs when $C=p+1$, in this case we assign 0 to $U_{i,(p-1)}$, and p to $U_{i,p}$. The third exception occurs when both conditions $C=p+1$, and $K=RxC$, occur. In this case, besides assigning 0 to $U_{i,(p-1)}$, and p to $U_{i,p}$, we exchange $U_{(R-1,0)}$ with $U_{(R-1,p)}$.

Another important block in the architecture is the "Modulo Computation" block that besides performing the modulo operations in *pre-computation* and *run time modes*, it also writes the $S(j)$ sequence in the $S(j)$ RAM. Connected to this block is the "Circular Buffer" block used in *run time mode* for recursive operations. In order to synchronize every block in the architecture, a "Controller" block generates all control signals, its state machine diagram is shown in Fig. 9.

Most interleavers like the ones presented in [2],[5],[8],[9] and [10] provide the interleaved addresses and a label indicating whether or not those addresses are valid. Our interleaver architecture besides providing the addresses, it deals with the input data stream. For example, if we receive an input data stream of any size between 40 and 5114 our interleaver takes it and rearranges it according to its corresponding 3GPP interleaved path. The architecture outputs a data stream already processed, even when exceptions are present. In order to manage data streams, we use a data RAM which increases the size of our design, but as it is mandatory for Turbo codes to store this data stream (in a RAM), a Turbo code architecture that includes our interleaver may put aside the RAM.

TABLE 1. AVERAGE INTERLEAVED ADDRESS RATE FOR DIFFERENT BLOCK SIZES

Block Size (*K*)	Run time mode Cycles / frame	Average valid address /cycle
40	40	1.00
41	50	0.82
2041	2060	0.99
4241	4440	0.95
4840	4840	1.00
5114	5120	0.99

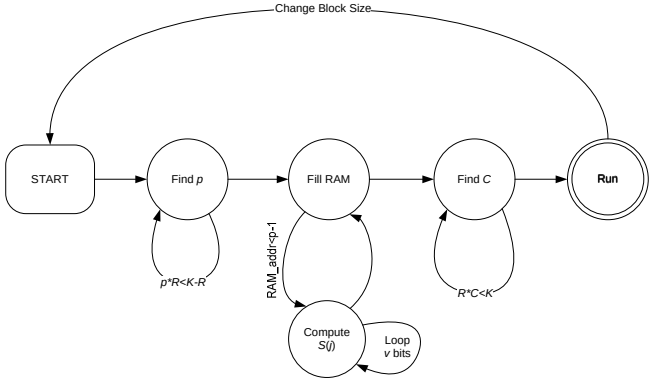

Figure 9. State machine of the controller for calculating interleaved addresses

Usually Turbo codes' interleavers work with the same block size *K* several times before changing it. When an interleaving operation is required the *pre-computation mode* is performed, where S(j) is calculated and placed in RAM. Then the *Run time mode* starts where interleaved addresses are calculated continuously meanwhile parameter *K* does not change. Only when *K* is changed, *pre-computation mode* is performed again. This is shown in Fig. 9.

We also incorporate an extra RAM, called I/O RAM for data movement when in exception handling. In interleaver mode this RAM is used to hold data input when invalid addresses are generated, waiting for a valid address in order to write this data to DATA RAM. In Deinterleaver mode, when garbage is read from DATA RAM, because a invalid address, the I/O RAM ignores this garbage data waiting for a valid data and then output this data. In this way, we can receive or deliver data in a consecutive way even when invalid addresses caused by exceptions are generated. This is another main difference with respect to [2]. In fact, in [2] it is not mentioned how this data swapping is implemented.

A maximum delay of 199 clock cycles is generated due to invalid addresses; this is the number of locations of the extra I/O RAM.

When we use our design as interleaver, we use de i_addr signal for writing the data stream in the DATA RAM and we read in a increasing order, and when we use it as deinterleaver, we write the data stream in a increasing order and we use i_addr signal for reading. To achieve this we use some multiplexers as shown in Fig. 10, as an input of these multiplexers we have an exceptions ROM that works like a LUT, and it is addressed by the controller.

V. PERFORMANCE RESULTS

Finally, after designing our architecture, we compiled and downloaded it to the FPGA Cyclone II from ALTERA. Table 2 summarizes the results and compares them against other interleaver designs. As it can be seen from the table, our design occupies about 16.5% (5000 out of 30216 Logic Elements -L.E.-) of the available resources.

It is worth mentioning that although our design is bigger than others, it includes exceptions ROM, an I/O RAM and a DATA RAM as well as extra hardware to control them. In this way, our architecture is capable of working both, as an interleaver and as a deinterleaver, which other designs are not capable of.

TABLE 2. HARDWARE USAGE COMPARISON FOR DIFERENT INTERLEAVER IMPLEMENTATIONS

Implementation	Size	Comments
ROM (with all possible patterns)	> 100 Mbit	Can easily manage data. Impractical size
Design in [9]	~30 K gates	No data streams
Processor based interleaver [8]	~32 K gates	Lower average bit per cycle than our design and no data streams
Design [2] excluding 2Kb RAM	~2.2 K gates	Obtain around 88% of the interleaved paths, and no data streams
Our design including RAM and ROM modules	~5000 L.E. ~100 K gates	Generates 100% of the interleaved paths and manages data streams. Works as Interleaver / Deinterleaver

Another advantage of using this extra hardware is that our design can manage data streams, and really performs the interleaving/ deinterleaving operation. This means that, if we receive a data block of any size, our architecture not only generates the interleaved/deinterleaved addresses but it also can perform data interleaving/deinterleaving as it is, i.e. without any modification.

V. CONCLUSIONS

In this paper we presented a fully functional 3GPP Turbo code interleaver/ deinterleaver architecture that receives an input data stream of any size established by the 3GPP standard and delivers this stream interleaved or de-interleaved depending on the user requirements. In this design we used a computer algorithm to calculate modulo operation, and we took advantage of using the same hardware in *pre-computation* and *run time* modes by multiplexing it. Finally by adding RAMs for data handling we achieved a complete architecture that can perform interleaving/deinterleaving operations as required by the 3GPP standard for Turbo codes.

REFERENCES

[1] G. R. Blakley, "A computer algorithm for calculating the product A*B mod M",IEEE Trans. On Computer, vol. C-32 , No 5, pp. 497-500, may 1983.

[2] Rizwan Asghar and Dake Liu "Very low cost configurable Hardware Interleaver for 3G turbo decoding", 3rd International Conference on Information and Communications Technologies: From Theory to Applications, ICTTA 2008, pp. 1-5.

[3] C. Berrou, A. Glavieus, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-Codes," in IEEE ICC'93, May 1993, vol. 2, pp. 1064-1070.

[4] Branka Vucetic and Jinhong Yuan, "Turbo Codes Principles and Applications," Kluwer Academic Publishers, 2000.

[5] Carlos R. Sánchez, R. Parra-Michel and M. E. Guzmán-Renteria, "Design and implementation of a multi-standard interleaver for 802.11a, 802.11n, 802.16e & DVB standards", International Conference on Reconfigurable Computing and FPGAs, ReConFig 2008, pp. 379 - 384.

[6] Anabel Morales-Cortés, R. Parra-Michel, Luis F. González-Pérez and Gabriela Cervantez T, "Finite precision analysis of the 3GPP standard turbo decoder for fixed-point inplementation in FPGA devices", International Conference on Reconfigurable Computing and FPGAs, ReConFig 2008, pp. 43-48.

[7] "3rd Generation Partnership Project, Technical Specification Group Radio Access Network; Multiplexingand Channel Coding (FDD)," Release 6, 3GPP TS 25.212 v6.0.0(2003-12).

[8] M. Shin and I.-C. Park, "Processor-based turbo interleaver for multiple third-generation wireless standard," IEEE Commun. Lett., vol. 7, no. 5, pp. 210-212, May 2003.

[9] P. Ampadu and K. Kornegay, "An efficient hardware interleaver for 3G turbo decoding," Proc. RAWCON'03, pp. 199-201, August 2003.

[10] Z. Wang and Q. Li, "Very low-complexity hardware interleaver for turbo decoding," IEEE Trans. On Circuits and Sys.- II: vol. 54, no. 7, pp. 636-640, July 2007.

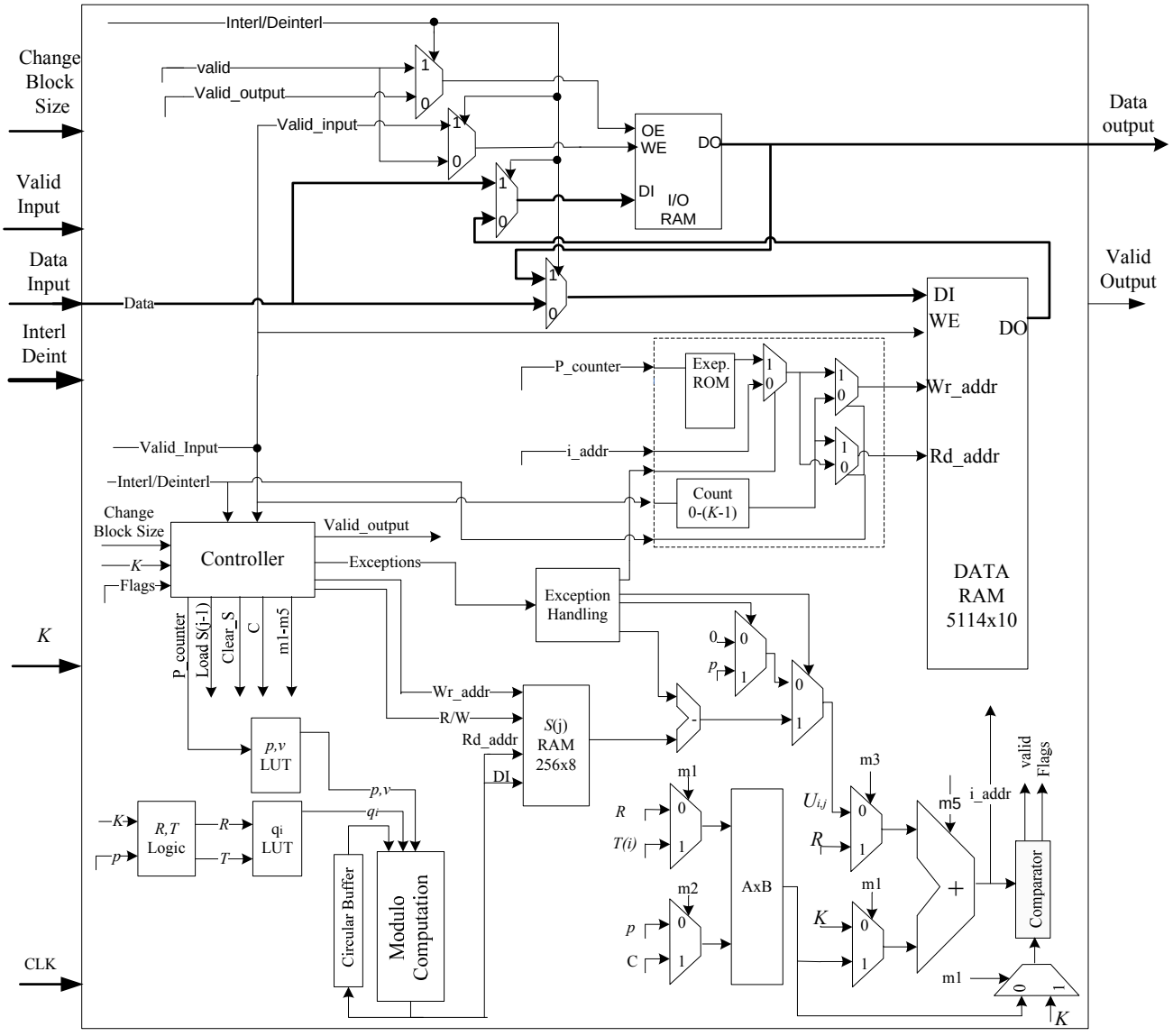

Figure 10. Complete architecture for 3GPP Interleaver/De-interleaver