High-Level and Compact Design of Cross-Channel LTE DownLink Channel Encoder

Jieming Xu and Miriam Leeser(✉)

Northeastern University, Boston, MA 02115, USA
xu.jiem@husky.neu.edu, mel@coe.neu.edu
https://www.northeastern.edu/rcl/

Abstract. Field Programmable Gate Arrays (FPGAs) provide great flexibility and speed in Software Defined Radio (SDR). However, as a mobile wireless protocol, the LTE system needs to maintain coding procedures for different channels, and the hardware's implementation is more complex than other wireless local area network (WLAN) specifications. Thus a compact and resource reusable LTE channel coder is needed as hardware resources and speed are the main pain points in SDR implementation. Traditional FPGA design and synthesis only focus on low levels of resource reuse, and IPs are independently designed without considering the whole system, which causes resource waste. In this paper, we describe a LTE downlink channel encoder processing chain implemented in FPGA hardware. Reuse in the whole system is done at a channel level and above, and scarce resources like BRAM are shared between processing units to maximize reuse. The system can efficiently process data and control channel signals at the same time using the same hardware. For the data channel, we use cross-component optimization to reduce the usage of BRAMs up to 25% for high volume data buffering. A novel rate matching design reduces the latency which improves the performance. By applying high-level reuse, the cross-component design can reduce resource usage while maintaining a good processing speed.

Keywords: LTE · DL-SCH · Reconfigurable hardware · SDR

1 Introduction

To keep up with the pace of updating standards, an SDR should have reconfigurability and software programmable hardware; FPGAs provide a good implementation platform that achieve these goals. The gap between wireless communications and hardware design requires developers to be proficient in both these areas. Libraries of Intellectual Property (IP) for wireless communications simplify hardware design for those proficient in SDR. However, without considering

I. Moerman et al. (Eds.): CROWNCOM 2018, LNICST 261, pp. 15–24, 2019.
https://doi.org/10.1007/978-3-030-05490-8_2

the whole processing chain and dependencies between different IPs, systems may suffer from resource waste and lower than optimal speed.

To implement the LTE system, many features can be efficiently implemented on a general purpose processor. Some blocks, such as Orthogonal Frequency-Division Multiplexing (OFDM), are best placed in hardware. Further, as massive MIMO and wideband OFDM will be applied in next generation systems, resources for baseband channel encoding will be quite limited. For the Downlink Shared Channel (DL-SCH) using QAM256 modulation, the maximum size of the transport block is 97896 bits and 105528 bits in releases 12 and 14 respectively. This requires the encoding system to be compact and efficient. In addition we design our system to accommodate processing for several different channels.

The contributions of this paper are: (1) We process both data and control channels using the same hardware, (2) we optimize rate matching to reduce latency, and (3) we optimize across the whole system to improve on-chip memory usage. In our implementation, reuse is considered across different channels and between IPs, which saves resources.

Previous research has focused on optimizing individual components for channel encoding independently. Researchers have investigated paralellizing the CRC and turbo encoder [3], and optimizing code block segmentation [5]. Santnanam et al. [8] examine choosing the FPGA Block RAM to achieve optimal power consumption and resource usage. Others have proposed a high speed architecture and a solution to reduce latency and resource usage in rate matching [2,4]. Fahmy et al. [7] introduces a method to improve resource sharing for DSP blocks, which can reduce the DSP consumption in OFDM systems. Hassan et al. [1] have designed a LTE downlink transceiver with synchronization and equalization; however, implementation details are missing. While excellent work has been done on each processing unit independently, to make the system work as a whole requires consideration of components' compatibility and resource usage.

2 Background

In LTE systems, information in the logical channel from the MAC layer are assigned to the transport channel in the physical layer for encoding. At the same time, control information is added for encoding which is irrelevant to the higher layer. The standard defines turbo encoding, tail biting convolutional coding, block coding etc. as coding schemes. For turbo and tail biting convolutional coding, each has its own rate matching scheme.

In the downlink, turbo coding is applied to DL-SCH, Paging Channel (PCH) and Multicast Channel (MCH). Tail biting convolutional coding is applied to Broadcast Channel (BCH) and Downlink Control Information (DCI). We refer to the DL-SCH, PCH and MCH as data channels and BCH and DCI as control channels. In each Transmission Time Interval (TTI), the CRC encoder receives data from the MAC layer and attaches parity bits to the transport block bit stream. The padded transport block is segmented into code blocks for turbo coding in predefined sizes. To align the size of padded transport block with different segmented code blocks, a number of F filler bits may be added to the

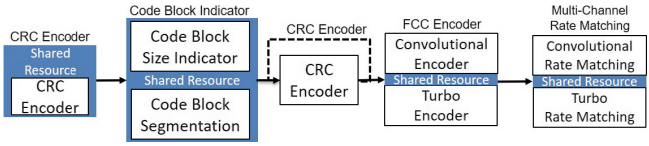

Fig. 1. Hardware architecture

head of the first code block as null bits. For data channel encoding, if the total number of segmented code blocks is larger than one, an additional CRC encoding process will be added to each code block before the turbo encoding process. For the control channel, the data will be sent directly to the convolutional encoder for processing. Next, each code block is processed by the turbo encoder, a Parallel Concatenated Convolutional Code (PCCC) with two 8-state constituent encoders with one (Quadratic Polynomial Permutation) QPP interleaver, and code rate 1/3. For control, The convolutional encoder first initializes its registers in the Shift Registers (SR) to the last 7 bits of each code block. Then the encoder encodes the data with these initialized SR with constraint length 7 at a coding rate of 1/3. Rate matching, which contains bit selection and pruning, is the last step of channel coding. It merges three bit streams (one information bit stream and two parity bit streams) generated by the turbo (or convolutional) encoder into one. After rate matching, the data will be send to the rest of the LTE physical layer processing.

3 Hardware Architecture

The processing chain for our hardware architecture, shown in Fig. 1, contains CRC encoder, code block indicator, Forward Correction Coder (FCC) encoder and multi-channel rate matching. To support encoding different channels' data using the same hardware, we merge different processing units by sharing hardware resources. The first CRC encoder is the same for different channels. To make the control channel's processing chain compatible with the data channel, we redefine the use of the code segmentation processing unit as code block size indicator without changing the hardware design. This redefinition scheme allows CRC encoded control information to be passed to the FCC encoder while using the same hardware architecture. The convolutional and turbo encoder are merged into one FCC encoder, and their rate matching is also merged into multi-channel rate matching with some resource sharing. In this design, the data and control information can be encoded using the same hardware. Optimized rate matching is also applied to improve the performance for both the data and control channels.

3.1 Rate Matching

The sub-block interleaver is based on matrix interleaving. Bit streams $d_k^{(0)}$, $d_k^{(1)}$ and $d_k^{(2)}$ of each code block are first reshaped into a $R \times 32$ matrix, where

$R = \lceil K/32 \rceil$, and K is the size of each code block after turbo encoding. The bit stream is stored in the matrix in row-wise order and N_D dummy bits padded to the head of each bit stream as null bits if K is less than K_Π (defined as $R \times 32$). Then a column-wise permutation is performed for the resized bit streams. For the convolutional encoder, all bit streams use the same permutation pattern. For turbo coding, the permutation is applied to bit streams $d_k^{(0)}$ and $d_k^{(1)}$ and bit stream $d_k^{(2)}$ uses the permutation pattern shown in Eq. 1.

$$\pi(k) = \left(P\left(\left\lfloor \frac{k}{R} \right\rfloor \right) + 32 \times (k \bmod R) + 1 \right) \bmod K_\Pi \tag{1}$$

where $\pi(k)$ is the index of the kth bit in $d_k^{(2)}$ after permutation.

In bit collection, the permuted bit streams v_k^0, v_k^1, v_k^2 are read out from the matrix in column-wise order and written into a circular buffer in interleaved order, where w_n is the nth data bit in the circular buffer:

$$w_k = v_k^0, \text{ for } k = 0, 1, 2..., K_\Pi \tag{2}$$

$$w_{K_\Pi + 2k} = v_k^1, \text{ for } k = 0, 1, 2..., K_\Pi \tag{3}$$

$$w_{K_\Pi + 2k + 1} = v_k^2, \text{ for } k = 0, 1, 2..., K_\Pi \tag{4}$$

In bit selection and pruning, data is read out from the circular buffer skipping the null bits until the data size reaches the capacity of a channel. The read starting point for turbo coding is chosen through the calculation of redundancy version (rv) and bits capacity for each code block (N_{cb}). For convolutional coding, the data is read from the first bit of v_k^0.

In DL-SCH processing, the rate matching process slows down the whole system's speed which needs to merge three bit streams into one. Researchers [4] have optimized this performance by implementing two RAM sets for sub-block interleaving and bit collection, directly following the process described in the LTE standard. In hardware, the interleaving process can be done by calculating the address while writing data bits to RAM; thus, the sub-block interleaving and bit collection can be done using only one RAM set. In this way, the RAM cost will be cut in an half. This work does not clearly describe how to design the FSM for bit selection, which needs to select information bits and skip dummy bits. Without proper optimization, at least one clock cycle is needed for skipping the dummy bits which results in extra latency. This latency is obvious when the code block is small. A patent [6] describes an approach to reduce this latency by calculating and storing the number of dummy bits before the information bits. However this approach cannot be efficiently implemented in hardware because storing the number of dummy bits before the information bits requires large amounts of RAM, and eliminates the ability for parallelization, which results in extra latency.

The proposed rate matching hardware architecture is shown in Fig. 2. Three RAMs are used for sub-block interleaving to mimic a circular buffer. Compared to [4], which uses two RAMs for each bit stream to do matrix resize and inter-column permutation in sub-block interleaving, we use only one RAM for each

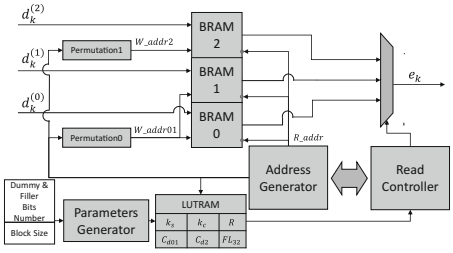

Fig. 2. Rate matching

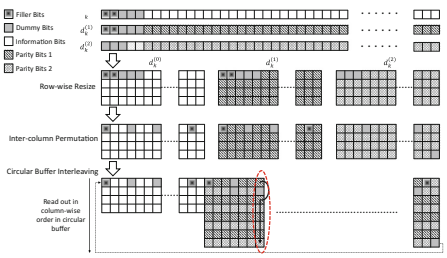

Fig. 3. Rate matching circular buffer (Color figure online)

bit stream to finish interleaving. After address permutation, the data streams are written into BRAMs in interleaved order. By reading out the parameters from LUTRAM, the read controller controls the address read from the RAMs and selects the correct output from the three RAMs.

In bit selection and pruning, the processing unit should distinguish null (filler and dummy bits) bits that are scattered in valid bits (the information, parity bits0 and parity bit1, shown in Fig. 3). Solutions include using 2 bits to store valid and null bits or locating the address of null bits using Eq. 1 and the permutation table. However, the first solution doubles the cost of RAM and both solutions consume at least one clock cycle to read out or calculate whether a null bit is there. This reduces the performance of rate matching, especially when the code block is very small. The difficulty in skipping the null bits is how to locate these bits before the address generator reaches their addresses. To do this, we must have knowledge of where these null bits are. Because the null bits are always padded at the head of each bit stream, after inter-column permutation, these null bits will be located in the first several rows of each matrix (see Fig. 3). If we know how many null bits exist in each column, the address generator can preload this number and increase the address from that number in column-wise order, so the null bits will be skipped. After the address number reaches the last row, it moves to the next column and repeat this process.

As each bit stream is reshaped into a $R \times 32$ matrix, the number of null bits d_c for each column c is represented as:

$$d_c = \left\lfloor \frac{N}{32} \right\rfloor + J(c) \quad \left(J(c) = \begin{cases} 1, & \text{if } P^{-1}(c) < C_d \\ 0, & \text{if } P^{-1}(c) \geq C_d \end{cases} \right) \tag{5}$$

where $P^{-1}()$ is the inverse of the inter-column permutation function for different coding schemes, and C_d is the pattern of null bits calculated using:

$$C_d = \begin{cases} C_{d01} = (N_D + F) \bmod 32, \\ (\text{for } d_k^{(0)}, d_k^{(1)} \text{in turbo}) \\ \\ C_{d2} = N_D \bmod 32, \\ (\text{for } d_k^{(2)} \text{in turbo and } d_k^{(0)}, d_k^{(1)}, d_k^{(2)} \text{in convolutional}) \end{cases} \tag{6}$$

In hardware implementation, term $\lfloor \frac{N}{32} \rfloor$ can easily be calculated by bit shifting. Permutations are parity permutation without overlap, which share the same pattern as their inverses. Therefore, for $d_k^{(0)}$ and $d_k^{(1)}$ streams in turbo code and all the bit streams in convolutional code, the function $P^{-1}(c)$ is the same as the permutation $P(c)$. Thus a 32×5 (LUT) can implement function $P^{-1}(c)$. However, the permutation of $d_k^{(2)}$ in turbo code involves the calculation of Eq. 1 whose inverse cannot be directly implemented as a small LUT. Looking at Eq. 1, we find it cyclic shifts the $d_k^{(1)}$ stream and then applies the sub-block interleaving using the same turbo permutation. Therefore, to implement function $P^{-1}(c)$ for that bit stream, we only need to add one to the LUT for permutation to represent the cyclic shift. In a real implementation, all the permutation LUTs should be left-cycle shifted by one; this helps the row address generator load the null bit number before the next column's read begins. To reduce latency, parameters calculated in Eqs. 5 and 6 are stored in LUTRAM as k_s and k_c for read starting point, R, C_{d01}, C_{d2} and FL_{32} for filler bits information (Fig. 2).

We use three BRAMs with control logic to mimic a circular buffer structure. However, the interleaving order shown in Eqs. 3 and 4 introduces another challenge to the control logic's implementation for the circular buffer. The problem is that the $d_k^{(2)}$ stream uses a different permutation and the null bits it contains are also different from the other two bit streams. This may result in the data not being read from BRAM1 and BRAM0 in sequential order. Sometimes, a number of data bits may need to be read from a single BRAM continuously, for example when the data is read from the column (red circle in Fig. 3). To solve this BRAM iteration problem, we use Algorithm 1 to decide which BRAM to select while comparing the row address of different BRAMs dynamically. Here, d_{c01} and d_{c2} are calculated as in Eq. 5. In this way, BRAM selecting can be easily done using a comparator.

3.2 High Level Resource Reuse

Xilinx Vivado provides IP including 3GGP Turbo Encoder and LTE DL Channel Encoder which provide ease of use but remove freedom for the designer. The LTE DL Channel Encoder includes the downlink channels in LTE. It is powerful however it consumes a lot of resource which restricts its usage in limited resource situations such as MIMO and OFDM transceiver. The independent 3GGP Turbo Encoder IP cannot be optimized with other packaged IPs when directly connect to them. We use a high level of resource reuse between blocks for more compact systems. We fully merge the CRC encoder for control and

Algorithm 1: Circular Buffer Interleaved Data Read Out with null Bits

if $row\_address1 == R$ & $row\_address2 == R$ **then**
  $row\_address1 \leftarrow d_{c01}$;
  $row\_address2 \leftarrow d_{c2}$;
else
  if $row\_address1 < row\_address2$ **then**
    $row\_address1 \leftarrow row\_address1 + 1$;
    select BRAM1;
  else
    $row\_address2 \leftarrow row\_address2 + 1$;
    select BRAM2;
  end
end

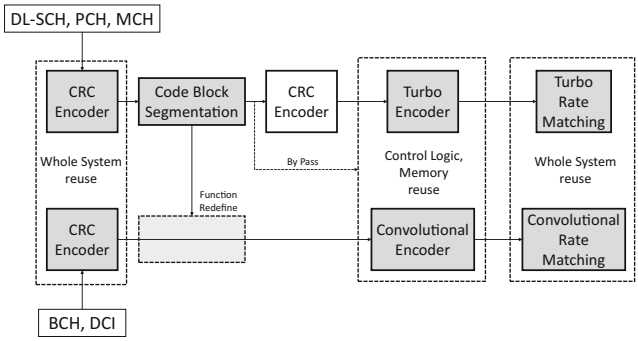

Fig. 4. High level resource reuse

data channels by simply using the same hardware, as shown in Fig. 4, because of their same options for the coding polynomial. Compared to the control channel, the data channel has additional code block segmentation and a CRC encoder. Code block segmentation divides a long code into small code blocks and outputs their sizes. The control channel also needs that code block size information for encoding; however no segmentation is needed. In the data channel's encoding, if there is only one code block in the TTI, this code block will be bypassed to the turbo encoder without a second CRC encoding. The control channel code blocks can also bypass the second CRC encoding. To apply this, the code block segmentation in the control channel's processing is redefined as a code block size indicator that segments the control information into a single code block to avoid the next CRC encoding. In this way, the encoding before the FCC encoder can share the same resources without changing the hardware architecture. The FCC is designed to use the same control logic and memory to finish these two coding schemes. The convolutional encoder shares one RAM with the turbo encoder for data buffering and register initialization. Independent design of the convolutional encoder alongside the turbo encoder costs resources. In multi-channel

rate matching, we use the same hardware for the data channel by making some slight changes. Another permutation table for convolutional coding is added. The algorithm for the circular buffer data read works for both channels. As the code blocks in the control channel have no filler bits, the null bits pattern will be the same as C_{d2} in turbo rate matching. In this way, the bit collection and bit selection and pruning elements can also be applied to control data's rate matching using the same hardware.

3.3 Buffer Optimization

For the FPGA implementation, the largest code block size requires a 13-bit address. As a result, the 18 Kb BRAM is divided into two areas for code block storage and the 36 Kb BRAM is divided into four areas for data buffering. Therefore, a simple ping-pong buffer can be realized using one 18 Kb BRAM for the CRC while two 18 Kb BRAMs are need for turbo encoding as it uses two independent data streams. Maximum delay is achieved when the lowest code rate is chosen at 1/3 which requires the largest amount of data buffering. To reach this rate, the total number of code blocks is 6, and the output size is three times the input size. If the data ports timing strictly follows the timing diagram shown in Fig. 5, the rate matching can only process five continuous code blocks at a time with a 36 Kb BRAM for each bit stream. However, the output code block size may exceed the size shown in Fig. 5 because the output data size of each code block needs to match the modulation order, the number of bits coded into one modulated symbol. Consider that the system has some timing slack, more than four code blocks should be stored in RAM. Therefore, two 36 Kb BRAMs are needed per data stream. Because the rate matching process uses three data streams, a total of six 36 Kb BRAMs are needed. These are the BRAM resources needed if we independently implement the processing units. Turbo encoding triples the size of RAM needed for data buffering and rate matching because three bit streams are generated. However, if the system is optimized together, the buffer usage in rate matching can be greatly reduced. The buffers in CRC and turbo encoder can help rate matching to buffer data. A simple approach is to assign one code block to the turbo encoder processing unit and four code blocks to rate matching. When the rate matching processing finishes any code block's encoding, which frees the space for one code block's buffering, the turbo encoder can then transfer its buffered data to the rate matching processing unit. If the system is optimized as a whole, the rate matching processing unit needs only three 36 Kb BRAMs to buffer four code blocks. As a result, the whole system can buffers more code blocks while using fewer RAMs than if the processing units are implemented independently.

4 Results

We use Mathworks and Xilinx tools to support our design flow. Simulink is used with HDL Workflow Advisor to generate HDL code from Simulink blocks and

Matlab code. Xilinx Vivado 2016.4 is used for synthesis and implementation. Except for the LTE CRC and turbo encoder blocks provided by Mathworks, we formulate and implement the whole system in our design and run the results on a Zedboard. Resource usage on a Zedboard is shown in Table 1 along with the percentage reduction achieved compared to the design without resource sharing. BRAM usage is shown in Table 2. Note that the turbo encoder uses 5 18 Kb RAMs, two for data buffering and three for the QPP interleaver. Thus, the non-resource sharing design costs more resources, with 33% more BRAMs consumed. Although a single LTE downlink encoding system consumes very few resources, the LTE OFDM modulator requires a large amount of resources, which won't leave abundant resource for the encoder. As a result, the resource consumption should be kept as small as possible. In addition, MIMO requires multiple encoders to work together to achieve high throughput. When multiple encoders are implemented, the resources they save together can be considerable. Because the size of filler bits may vary from 0 to 31 randomly, we pick the mean of 16 bits for testing. Results show that optimized rate matching can reduce delay by 28.6% for small blocks when compared to non-optimized running at the same clock frequency. For very large blocks the optimized design has lower delay, but only 1% delay improvement.

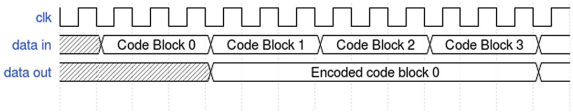

Fig. 5. Data ports timing of rate matching

Table 1. Resource usage of downlink encoder (with resource sharing)

Resource	Utilization	Available	Utilization%	Utilization reduced%
LUT	1073	53200	2.02	27.34
LUTRAM	163	17400	0.94	18.97
Flip-Flop	1243	106400	1.17	22.00
BRAM	6	140	4.29	39.91

Table 2. BRAM usage

Items	Independent non-optimized		Cross-component optimized	
	18 Kb RAM	36 Kb RAM	18 Kb RAM	36 Kb RAM
CRC encoder	1	0	1	0
FCC encoder	5	0	5	0
Rate matching	0	6	0	3
Total usage	6	6	6	3

5 Conclusions and Future Work

We have shown that optimizing the data channel processing chain as a whole and high level resource reuse saves scarce resources in FPGA implementations such as BRAMs. In addition, we presented a novel architecture for a rate matching system with low latency. If multiple LTE downlink encoders are implemented for parallel processing in a MIMO system, the saved resources are considerable. The designs presented in this paper run at frequencies of 130 MHz on Zedboard. In the future, we plan to implement multi-channel resource reuse as a tool to facilitate research and realization of SDR designs. We also plan to investigate tools that optimize designs across blocks, and not just within individual blocks. If these tools can work with vendor tools such as Vivado HLS, it will simplify the design of SDR system and result in improved resource efficiency. Furthermore, we plan to investigate targeting the RFSoC chip from Xilinx for similar designs. The RFSoC included cores for FEC that can be used for turbo encoding.

Acknowlegdements. This research is funded in part with support from Mathworks.

References

1. Hassan, S.M., Zekry, A.: FPGA implementation of LTE downlink transceiver with synchronization and equalization. Commun. Appl. Electron. **2**(2) (2015)
2. He, S., Hu, Q., Zhang, H.: Implementation of rate matching with low latency and little memory for LTE turbo code. J. Inf. Comput. Sci. **10**(13), 4117–4125 (2013)
3. Hwang, S.Y., Kim, D.H., Jhang, K.S.: Implementation of an encoder based on parallel structure for LTE systems. In: IEEE Wireless Communication and Networking, April 2010
4. Lenzi, K.G., de Figueiredo, F.A., Figueiredo, F.L.: Optimized rate matching architecture for a LTE-advanced FPGA-based PHY. In: IEEE CAS (2013)
5. Lenzi, K.G., Figueiredo, F.A., Bianco Filho, J.A., Figueiredo, F.L.: Fully optimized code block segmentation algorithm for LTE-Advanced. Int. J. Parallel Prog. **43**(6), 988–1003 (2015)
6. Reinhardt, S.: Technique for rate matching in a data transmission system, US Patent 8,446,300, 21 May 2013
7. Ronak, B., Fahmy, S.A.: Improved resource sharing for FPGA DSP blocks. In: Field Programmable Logic (FPL), pp. 1–4. IEEE (2016)
8. Santhanam, V., Kabra, L.: Optimal low power and scalable memory architecture for turbo encoder. In: DASIP, October 2012