POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Efficient LDPC implementation on FPGAs for 5G networks

Supervisors

Prof. LUCIANO LAVAGNO

Prof. MIHAI LAZARESCU

Candidate

ZE CHEN

July 2021

Summary

Low-Density Parity-Check code, known as LDPC code, or Gallager code, has been introduced by Robert Gallager in 1963. After 18 years, In 1981, Michael Tanner extended Gallager's LDPC code paper to graph theory. In 1996, Mackay, Spielman, and Wiberg found out the advantage of LDPC code regarding its low linear decoding complexity and its excellent performance, close to Shannon's Limit. Since then, the LDPC code was massively studied in the search field. Nowadays, it became a coding standard of 5G technology.

This thesis work builds on top of an existing research focusing on hardware acceleration on FPGA based on a software implementation of an LDPC decoder provided by OpenairInterface Software Alliance Consortium (OAI). It included two versions of the code, Advanced Vector eXtension2 (AVX2) solution and Compute Unified Device Architecture (CUDA) solution. An attempt to synthesize the AVX2 version for acceleration on FPGA was made using the High-Level Synthesis (HLS) tool provided by Xilinx. The synthesis did not succeed because the original C code had unsynthesizable constructs. In consequence, the CUDA solution was adopted, imported in OpenCL language, then optimized inside the SDAceel development environment by Xilinx. For this version, the bitstream was generated and tested on the FPGA board. The final performance of CUDA solution on FPGA is 41.152 ms, whilst the software emulation result of the AVX2 solution running at 3.2 GHz is 0.257 ms and the GPU solution is 0.107 ms, which are 160 × and 400 × better than the CUDA solution on FPGA, respectively.

This work adopts the AVX2 code focusing mainly on converting the source code for synthesizability, accepted by the HLS synthesis tool. The conversion proved highly time-consuming and error-prone due to the multiple constraints on coding style. Firstly, the AVX2 instructions supported by Intel processors have to be explicitly implemented utilizing intrinsic functions, which has been already started in the previous work. During the work, multiple pointer casting problems have to be solved, because the HLS tool does not accept pointer casting among non-native C data types. The data type used in the whole project has been modified to solve this type of problem. Instead of having an alignment of 32 bytes composed of 4×64 bit long integers in the processing buffer, the basic composition of the processing array

is modified to 32×8 bit, maintaining the alignment of 256 bits and performing the calculations on 8 bits instead of 64 bits. Another issue is related to the constraint of pointers to unique addresses, as well as the system calls and the usage of dynamic memory. In computation functions and memory transfer functions, the directives that contain pointers as operands are replaced by explicit expressions on arrays in favor of eliminating synthesis errors caused by pointers. And then the address offsets of each involved processing buffer are re-calculated. In top-level function, the functions that measure the execution time of software emulation are removed since they are system calls.

Once synthesizable, the code requires a very long synthesis time. Therefore, the configuration of the LDPC decoder was limited to the most commonly used one, which is $Z = 384$, BG1, CodeRate = 1/3, BlockLength = 8,448, and in this way the synthesis time was reduced from 3 d to approximately 15 min.

The implementation of the chosen configuration represents the most critical case, since it requires the longest clock cycle to finish. Therefore, the worst-case scenario is analyzed for performance evaluation. Different LDPC decoder configurations modify the loop bounds of the grouped processing, because the number of check nodes or bit nodes varies in each group. The maximum loop bounds obtained from the selected configuration contribute to not only determining the worst case in performance, but also to the generalization of the decoder, because the unrolling method can be applied on a loop with fixed bound and conditional execution inside, with which can be generated the other LDPC decoder configurations.

For the High-Level Synthesis of the C source onto the FPGA accelerator, Xilinx Vivado HLS 2018.2 was chosen initially. However, this version did not support well array partitioning. Thus we ported the code to Vitis HLS 2020.2, the latest HLS tool version provided from Xilinx.

Several HLS techniques were adopted to improve the implementation performance. The code functions are divided into two broad groups, namely computation functions and memory transfer functions. Before performing optimization on the function level, data is copied from the off-chip memory to the on-chip memory in order to take advantage of the parallel access provided by the on-chip BRAM. Then, the internal processing buffers are cyclically partitioned by a factor of 32 to fully support the operations in parallel, which is adopted from the mechanism of AVX2. Optimization methods are performed group by group, starting from memory transfer functions: Firstly, observing that the loop bounds of all innermost loops are variable because of the characteristic of the quasi-cyclic algorithm, I wrote a sub-function in which the loop bounds are set to a fixed number and the loop body is conditionally executed. This way, the innermost loops could be unrolled by a factor of 32 since we have 32 ports on each partitioned buffer. However, copying between unaligned ports inhibits some synthesizer optimizations, hence three cases of memory transfer blocks are proposed: 32 bytes aligned starting ports of source

buffer with unaligned destination buffer, aligned destination buffer with unaligned source buffer, and both source and destination buffer aligned. Compared to the ideal case, in which both buffers are port aligned and results in a $32\times$ increase of performance, the functions that meet the first case could obtain $16\times$ better performance, whilst functions that meet the second case could obtain only $4\times$ performance improvements, along with higher resources occupation. Secondly, loop unrolling and pipelining were also exploited to accelerate the computation functions. The pseudo AVX2 functions are fully unrolled and the outermost loops in each function are pipelined. Moreover, the dependency pragma is essential to improve the performance of computation functions and memory transfer functions, because the synthesizer may not be able to determine the data dependency by itself.

The work stopped at the RTL exporting phase due to the extremely long running time. Thus, the final report is an estimation of actual performance on execution on the FPGA. In summary, the non-optimized performance is around 10,120,000 clock cycles. After performing all optimization directives, the computation functions are approximately $40\times$ better than the initial performance, whilst the memory transfer functions are approximately $20\times$ better. The final performance is 424,379 clock cycles, which is $24\times$ better. Comparing to the result of previous work and references, while counting the configuration of $Z = 384$, BG1, CodeRate = $1/3$, blocklength = 8,448, the execution time of the current AVX2 solution running on FPGA at 250 MHz is 1.701 ms, $25\times$ better than the CUDA solution running at 250 MHz. However, the software emulation result of AVX2 solution running at 3.2 GHz is still $8\times$ better than the current FPGA implementation, and the GPU implementation is $16\times$ better.

Acknowledgements

Firstly, I'm grateful to Prof. Luciano Lavagno and Prof. Mihai Lazarescu for having provided me the chance to work with them as a thesis student, also for their professional and patient instructions during the period of thesis work.

Secondly, I would thank Osama, Junnan, Giorgia, Walid, Nasir, and Wenlu for their helpfulness and care; they have always been generous in helping me whenever I encountered difficulties in my work. Besides, I'm thankful for the coffee breaks we had together and those chats from which I always got plenty of inspiration.

A warm thank and embrace to Alfonso, who was, is, and will be essential in my life. Without these relaxation moments we had together, I would not process my work as smoothly as expected. Thanks for the encouragement and those debates that helped a lot in releasing pressure.

My particular thanks to Pedro and Wenlu, who accompanied me for lunch in these months, those moments we had together are priceless.

Lastly, my parents' support is irreplaceable, I'm grateful that I could have their unconditional support and love forever.

Table of Contents

List of Tables

List of Figures

Listings

Abbreviations and Acronyms

3GPP
Third Generation Partnership Project

5G
Fifth Generation

AVX2
Advanced Vector eXtension2

AWGN
Additive White Gaussian Noise

BER
Bit Error Rate

BG
Base Graph

BN
Bit Node

BRAM
Block of Random Access Memory

CN
Check Node

CPU

Central Processing Unit

CUDA

Compute Unified Device Architecture

DRAM

Dynamic Random Access Memory

FPGA

Field Programmable Gate Array

HDL

Hardware Description Language

HLS

High-Level Synthesis

II

Initiation Interval

IoT

Internet of Things

LDPC

Low-Density Parity-Check Code

LLR

Logarithm Likelihood Ratio

LUT

Look Up Table

NR

New Radio

OAI

OpenairInterface Software Alliance

OpenCL

Open Computing Language

QC

Quasi-Cyclic

RAT

Radio Access Technology

RTL

Register Transfer Level

SISO

Soft-In Soft-Out

SNR

Signal to Noise Ratio

Chapter 1

Introduction

The purpose of communication is to transmit information known from part of the transmitter and unknown from part of the receiver in a reliable manner. With the rapid growth in demand for high-efficiency and high-reliability digital communication systems, the development of large-scale high-speed broadband networks has made it possible to transmit voice, images, and other multimedia information. Communication system designers are concerned chiefly about achieving as accurate information transmission as possible with limited data source power, channel bandwidth, system complexity, and equipment cost, minimizing the bit error rate of information transmission. This contributes to the development of the Internet of Things (IoT).

The Internet of Things (IoT), which is also called the Internet of Everything or the Industrial Internet, is a new technology paradigm envisioned as a global network of machines and devices that are capable of interacting with each other [1]. It allows the real-time capture of data from sensors [2]. The existing 4G networks have been widely used in the IoT and are continuously evolving to match the needs of future IoT applications [3]. With the research of 5G technology in recent years, fast and reliable wireless communication is no longer a bottleneck [4].

Channel coding is an effective method to eliminate or reduce the probability of information error during transmission. With the arrival of the information era and the fast development of information science, error correction codes are no more an issue. In theory, they have become an indispensable standard of the modern communication field [5]. Because of the continuous and rapid development of wireless and mobile communication applications, the requirements for error correction coding technology used in high data rate digital mobile communications

and other fields are getting higher and higher. The signal will inevitably be interfered with in transmitting through the channel, resulting in signal distortion. Therefore, error control codes are used to detect and correct information transmission errors caused by channel distortion.

In the noisy channel coding theorem, C.E. Shannon gives a method for achieving reliable communication in digital communication systems and the upper limit of the information transmission rate for reliable communication on a specific channel [6]. Nowadays, the theorem still has a profound impact on communication science. Recent research [7] proves its applicability in a different aspect.

Shannon put forward three primary conditions in his theorem:

1. Random encoding and decoding method
2. Construct an extended code with good progressive characteristics
3. Use the maximum likelihood decoding algorithm

Since when in the 1940s R. Hamming and M. Golay presented the first practical error control coding scheme, the development of coding theory in applied mathematics has been extensively promoted. The method used by Hamming is to combine every 4 bits of the input data into a group and then linearly combine these information bits of one group to obtain three parity bits. The obtained 7 bits are sent to a computer that reads these codewords following certain principles. Using specific algorithms, one can detect whether an error has occurred and find the position of the bit where a single bit error has occurred. This code can effectively correct single-bit error occurred in 7 bits. The coding scheme proposed by Hamming was later named Hamming code. Although the idea of Hamming code is relatively advanced, it also has many unacceptable disadvantages. First, the requirement of 3 redundant check bits for every four information bits decreases the coding efficiency. In addition, only one single bit error can be corrected in each code group. Based on the principle idea of coding, many coding methods are presented in the following decades, such as Golay code [8], which accepts 12 bits as a group and generates 11 parity bits by encoding and detects three errors in each group. Reed-Muller code [9] is more advanced than Hamming code and Golay code as it has more robust adaptability in terms of code length and error correction capabilities.

RM code still has a high research value [10] [11], its fast decoding algorithm is very suitable for optical fiber communication systems. One of the most widely researched codes is Bose Chaudhuri Hocquenghem code (BCH code) [12], the BCH code can be used to correct multiple random error patterns in multi-level,

cyclic, error correction, and variable-length digital coding with error-correction solid ability, especially for short and medium code lengths, its performance is close to the theoretical value thanks to the strict algebraic structure. Reed-Solomon code (RS code) also has good error correction performance for short and medium code lengths. Both BCH codes and RS codes belong to the category of linear block codes [13].

Before the Turbo code was proposed, there was always a gap of 2 dB to 3 dB between the gain and the Shannon theoretical limit. The channel cut-off frequency R0 has always been considered as the practical limit of error control code performance. The Shannon limit is only a theoretical limit, and it is impossible to reach it. Turbo code was first proposed in 1993. Because of its good application of the random coding and decoding conditions in Shannon's channel coding theorem, it has obtained decoding performance close to the Shannon theoretical limit [14]. However, Turbo code defects result from unsatisfied randomization in encoding and decoding: large delay in decoding, high computation, and error-floor effectively prevent it from applications in Communication systems with low latency requirements (e.g., Digital phone). In 1996, Mackay, Spielman, and Wiberg rediscovered the LDPC code [15], which Gallager proposed in 1963, a code with a higher potentiality of reaching Shannon limit than Turbo code. In the next chapter, more theoretical background of LDPC code will be given.

In this thesis, one software implementation of LDPC decoder published by OpenairInterface Software Alliance Consortium (OAI) is selected for acceleration in an FPGA implementation. After being converted to synthesizable code, it is transformed into HDL and accelerated in RTL by the HLS tool of Xilinx. These parts will be illustrated in chapters third and fourth, respectively.

Xilinx HLS tools provided an interface from software design to hardware design. It reduces the complexity of HDL coding by means of transforming C, C++, or system C into VHDL or Verilog. It essentially increases the effectiveness and flexibility of Hardware design by accepting software languages that facilitate the logic design and description of complex computation as a starting point [16]. Vivado HLS provides the following design flow: Compile, simulate, and debug the C/C++ algorithm, view reports to analyze and optimize the design, synthesize the C algorithm into an RTL design, verify the RTL implementation using the RTL co-simulation and finally package the RTL implementation into a compiled object file (.xo) extension, or export to an RTL IP which is further used by other tools in the Xilinx design flow. Several pragmas are supported to optimize the RTL design and explore the design space to find an optimal solution according to specific space and throughput requirements.

Chapter 2

Low-Density Parity-Check code

2.1 Theoretical background

All digital communication systems such as radar, remote control and measurement, internal calculations of digital computer storage systems, and data transmission between digital computers can be depicted as the model shown in Figure 2.1. The information source generates the information that needs to be transmitted. It can be either an analog signal or a digital signal. If the information source was an analog signal, it needs to be sampled and digitized before being sent to the digital system for transmission. Instead, if it was a digital signal, it can be symbols such as words, codewords, etc. The output of the information source is converted into a symbol series according to a given code table. Generally, the binary symbol series are commonly used, and the code elements are taken from the binary set $\{0, 1\}$. If the output signal of the source encoder is R_b bit/s, then R_b is called the Data Transmission Rate, or Data Rate for short.

The task of the source encoder is to convert the message sent by the source, such as language, image, text, etc., into a specific form that can resist channel noise and distortion and is in favor of transmission. The number sequence is called the information sequence after encoded and sent into the information channel.

Channel coding is one of the necessary means to achieve reliable signal transmission from the transmitter to the receiver. The transmission channel has inevitable noise and attenuation, introducing distortion and signal-decision errors to the transmitted

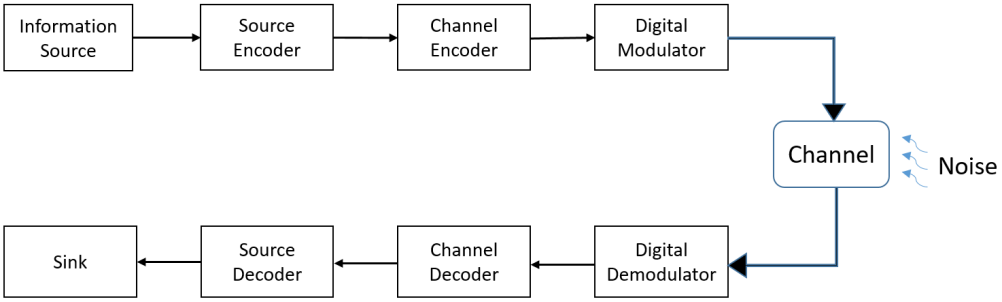

Figure 2.1: Basic architecture of digital communication system

data. Therefore, it needs to adopt error control codes to detect and correct bit errors by inserting redundant symbols in the information sequence to improve its error correction capability and system reliability.

The digital modulator transforms the information into signals transmitted via the channel because the information in the form of digital bits is not suitable for transmission on the physical channel, so it is necessary to convert these coded bits into a continuous waveform signal which is suitable for transmission. In our case, we only care about channel encoding and decoding in which both input and output are binary sequences. A simplified model is depicted as the model shown in Figure 2.2.

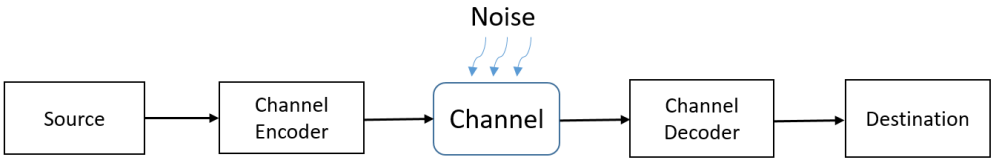

Figure 2.2: Simplified architecture of digital communication system

2.1.1 Tanner graph

Codeword of LDPC code consists of message bits and parity bits formed by firstly calculating modulo 2 of specific pair of message bits and then being concatenated to message bits. In this way, parity bits check the parity of message pair at receiver favoring correct possible bit flips. The codeword is obtained by multiplication of message bits and the Generator matrix. From (2.1), we can see that the first three columns of the matrix keep the message bits unchanged, and the rest columns outputs modulo 2 of m_1,m_1; m_1,m_2; m_0,m_2, respectively. p_0 protects m_0, and m_1.

p_1, and p_2 provide corresponding protection in the same manner

$$\begin{bmatrix} m_0 & m_1 & m_2 & p_0 & p_1 & p_2 \end{bmatrix} = \begin{bmatrix} m_0 & m_1 & m_2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{2.1}$$

We notate the first part of the Generator matrix as Identity Matrix $[I]$ and the second part as Parity Matrix $[p]$. In order to evaluate the correctness of the codeword, a checking matrix is required. In theory, modulo 2 of message bits in pair and their protection bit is supposed to be 0. In (2.2) shows the construction of the Parity-check matrix $[H]$. In case (2.3) is satisfied, all message bits alone with their parity bits are successfully recovered.

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \\ m_2 \\ p_0 \\ p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.2}$$

$$\begin{bmatrix} HC^T \end{bmatrix} = 0 \tag{2.3}$$

LDPC code is a linear block code with a sparse parity-check matrix [15] in which the number of 1s is dramatically less than the number of 0. Tanner graph provides a graphical representation of parity-check matrix that contributes effectively to later research on LDPC code. In Figure 2.3, an example of H matrix with row length 3, which stands for the number of message bits, and column length 7, which stands for the number of codeword bits. Conventionally notation of (3,7) linear code [17] is used to describe the parity-check matrix that consists of 3 message bits and four parity bits. On the left side of the Tanner graph, each column is listed vertically and denoted as round node. The first node stands for the first column in the H matrix and the following node as so. Instead, on the right side of the Tanner Graph, each row of the H matrix is denoted as a square node and listed from top to bottom in the same manner. Edges in between represent the relation among different bit nodes and check nodes, e.x. H_{14} is 1, and then in Tanner Graph, the 4th-bit node is connected with the 1st check node, H_{23} is 0, then there is no connection between the 3rd bit node and the 2nd check node.

Tanner graph is helpful for its direct topology that helps to denote the parity-check matrix in terms of recursive passing algorithm, which will be explained in detail later.

$$H = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

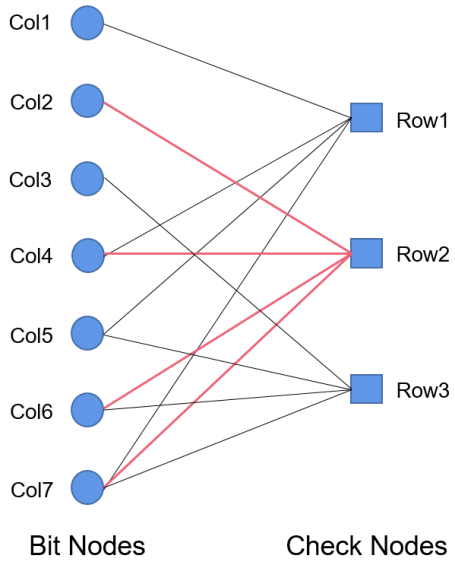

Figure 2.3: Graphical representation of a parity-check matrix

2.1.2 LDPC in New Radio

New Radio (NR), a new radio access technology (RAT) developed by the Third Generation Partnership Project (3GPP) for the 5G (Fifth Generation) mobile network. It was designed to be the global standard for the air interface of 5G networks. In NR LDPC code, two base matrices of fixed size are used:

$$\text{BG1} : 46 \times 68, \qquad \text{BG2} : 42 \times 52 \tag{2.4}$$

Several expansions are needed to obtain the accurate scale of the H matrix; indices and expansion factors are reported in Appendix A, Table A.1. Given a certain Base Graph and expansion factor Z_c, values in Base Graph are constrained within a range of $-1 \sim \mathrm{Z_c} - 1$. The expansion factor does a specific transformation to base graphs to obtain the H matrix's full scale. If an entry of the base graph equal to -1, then it needs to be expanded into an all-zero matrix of size $(\mathrm{Z_c}, \mathrm{Z_c})$. For 0 or any value less than $\mathrm{Z_c} - 1$, assumed as I, the entry is expanded into an identity matrix of size $(\mathrm{Z_c}, \mathrm{Z_c})$ then shifted right I times. So that the full H matrix is recovered employing an indicated specification. The full Z_c table of 3GPP standard is reported in Appendix A, Table A.2.

2.1.3 Performance considerations

The code rate is defined as the ratio of the length of message bits k over the length of codewords n

$$R = \frac{k}{n} \tag{2.5}$$

The new code rate after channel encoding is

$$R_{\mathrm{c}} = \frac{R_{\mathrm{b}}}{R} = \frac{R_{\mathrm{b}} \times n}{k} \text{ (bit/s)} \tag{2.6}$$

where R_b is the data rate of the source being sent into the channel.

Signal to Noise Ratio (SNR) and Bit Error Rate (BER) are two critical parameters that evaluate the performance of an LDPC decoder, and the BER is the number of bit errors divided by the total number of transferred bits during a specific time interval. Channel with higher SNR means information bits are less contaminated by noise. An exemplary implementation of LDPC encoder/decoder should achieve BER under the condition of SNR as low as possible. In Appendix A, Figure A.1 shows the comparison of OAI LDPC software implementation performance ran at five iterations and 50 iterations, respectively, with the reference performance provided by HUAWEI.

2.2 Encoding

As discussed in Chapter 2.1.1, the parity-check matrix is obtained by interchanging the identity matrix $[I]$ and the transposed parity matrix $[P]$ of the generator matrix. In Figure 2.4 given an example of a base matrix with 10 rows, 20 columns of $BG2$, expansion factor is 48, according to 5G NR the message part has 10×48bits, and the parity-check part has an equal size, starting from the left edge of the square till the last column. It usually does not enough to protect a set of message bits by a single parity bit because the erased bit may not be one. If several bits are erased during transmission, one parity bit cannot recover correctly original information since it might be erased as well. A more feasible way is to protect the parity bit in the same way as message bits. Standard of 5G NR adopts the multi-protection encoding method, which effectively handles the multi-erasure issue.

An interesting code structure is presented in the square part shown in Figure 2.4, and it is given another simplified example which emphasizes the top-middle part shown in Figure 2.5, H matrix [4,8] stands for 4 message bits on the left side and 4 parity bits in the right side, overall 8 codewords to be encoded. Considering the

24	14	23	37	−1	−1	47	−1	−1	8	1	0	−1	−1	−1	−1	−1	−1	−1	−1
5	−1	−1	12	19	12	19	8	29	31	−1	0	0	−1	−1	−1	−1	−1	−1	−1
8	35	−1	46	47	−1	−1	−1	43	−1	0	−1	0	0	−1	−1	−1	−1	−1	−1
−1	41	6	−1	36	28	28	14	12	37	1	−1	−1	0	−1	−1	−1	−1	−1	−1
8	16	−1	−1	−1	−1	−1	−1	−1	−1	−1	5	−1	−1	0	−1	−1	−1	−1	−1
41	42	−1	−1	−1	6	−1	27	−1	−1	−1	−1	−1	−1	−1	0	−1	−1	−1	−1
27	−1	−1	−1	−1	7	−1	31	−1	30	−1	17	−1	−1	−1	−1	0	−1	−1	−1
−1	7	−1	−1	−1	13	−1	9	−1	−1	−1	6	−1	37	−1	−1	−1	0	−1	−1
3	43	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	8	−1	−1	−1	−1	−1	0	−1
−1	2	−1	−1	−1	−1	−1	−1	30	−1	40	35	−1	−1	−1	−1	−1	−1	−1	0

Figure 2.4: Example of parity-check matrix

decoding phase, which is the reversed case of encoding, we modify the notation by replacing the entries number by In, which means an identity matrix right-shifted by n times. As discussed previously, the -1 value in the H matrix means all-zero matrix after expanded, 0 and other values are expanded to identity matrix with a proper right shift. 0 stands for all-zero matrix instead. Parity bits always have a double-diagonal structure of identity matrix, as shown in Figure 2.5.

$$H = \left[\begin{array}{cccc|cccc} I_1 & 0 & I_3 & I_1 & I_2 & I & 0 & 0 \\ I_2 & I & 0 & I_3 & 0 & I & I & 0 \\ 0 & I_4 & I_2 & I & I_1 & 0 & I & I \\ I_4 & I_1 & I & 0 & I_2 & 0 & 0 & I \end{array}\right]$$

Figure 2.5: Double-diagonal structure of parity-check matrix

The first equation of (2.7) is the condition to be satisfied for successful decoding. Apply it to our illustrated case, and we could obtain the following four equations with corresponding alignment depending on the parity check bit p_i .

$$\begin{aligned} & H\begin{bmatrix} m_1 & m_2 & m_3 & m_4 & p_1 & p_2 & p_3 & p_4 \end{bmatrix}^T = 0 \\ & I_1 m_1 + I_3 m_3 + I_1 m_4 + I_2 p_1 + I\ p_2 = 0 \\ & I_2 m_1 + I\ m_2 + I_3 m_3 \qquad\quad + I\ p_2 + I\ p_3 = 0 \\ & I_4 m_2 + I_2 m_3 + I\ m_4 + I_1 p_1 \qquad\quad + I\ p_3 + I\ p_4 = 0 \\ & I_4 m_1 + I_2 m_2 + I\ m_3 + I_2 p_1 \qquad\quad\quad + I\ p_4 = 0 \end{aligned} \tag{2.7}$$

If we add all four equations together, we can see the advantage of the double-diagonal structure. Entries with the same codeword are cancelled so we eventually

can obtain (2.8)

$$\begin{aligned} I_1 p_1 =& I_1 m_1 + I_3 m_3 + I_1 m_4 + I_2 m_1 + I\ m_2 + I_3 m_3 + \\ & I_4 m_2 + I_2 m_3 + I\ m_4 + I_4 m_1 + I_1 m_2 + I\ m_3 \end{aligned} \tag{2.8}$$

From the equation, we can see that the value of 4 message bits expresses the p1 and it is able to find p2 using the value of p1, then find the p3 from the value of p2, etc. After all four codewords are retrieved, the fifth one is easier to be obtained as it has only one unknown value at the fifth row of the H matrix, shown in Figure 2.4. To be noticed that the double-diagonal structure always has the same size that assures the reliability and applicability in all configurations of the LDPC encoder.

2.3 Decoding

Before Gallager [15] published his method, which takes a posteriori probability as a decision mechanism. Hard-output was proved to be less accurate alone with low applicability. Two kinds of decisions are widely studied in the decoding field:

1. Hard decision: At output takes the strictly closest value as confirmation value
2. Soft decision: At output takes confidence value to determine the stronger belief

The hard decision method has been dropped in consequence of its low reliability in results. In the actual case of the channel, the noise usually is less deterministic, could not be within a range of quantity. If the noise is high enough to flip the bit, the hard decision probably gives back the wrong value. Thus the information will not be retrieved correctly. Gallager [15] proposed the soft decision method, namely soft-in soft-out (SISO) that takes a sequence of beliefs on received values, where soft-in stands, then for output a sequence of the beliefs of the decoded messages are decided using hard decision, noticed that soft-out stands for the believe to be calculated and taken into account, finally a hard decision is necessary since the decoded messages are much more reliable compared to pure hard decision and believes cannot be considered as information, they are the probability of being 0 or 1.

In NR LDPC provided by OAI, the channel is simulated as if it went through an Additive White Gaussian Noise (AWGN) with variance δ^2 and modulated employing Binary Phase Shift Keying (BPSK), given a specific bit, Log-Likelihood Ratio (LLR) is used to determine the log value of the ratio between the probability of being 0 and the probability of being 1.

In (2.9), c is one bit of codeword at the output of the encoder and r is the corresponding bit received by the decoder. In other words, r is the contaminated value of c. $P_r(\mathrm{c}=0)$ and $P_r(\mathrm{c}=1)$ are the prior probabilities so both values are equal to 0.5.

$$\begin{aligned} P_r(c_1 = 0 \mid r_1) &= \frac{f(r_1 \mid c_1 = 0) \cdot P_r(c_1 = 0)}{f(r_1)} \\ P_r(c_1 = 1 \mid r_1) &= \frac{f(r_1 \mid c_1 = 1) \cdot P_r(c_1 = 1)}{f(r_1)} \end{aligned} \tag{2.9}$$

Take the ratio of the two probabilities. We obtain (2.10)

$$\frac{P_r(c_1 = 0 \mid r_1)}{P_r(c_1 = 1 \mid r_1)} = \frac{f(r_1 \mid c_1 = 0)}{f(r_1 \mid c_1 = 1)} \tag{2.10}$$

Where $f(r \mid c = 0)$ and $f(r \mid c = 1)$ are the Probability Distributed Functions, in the AWGN channel, the normal density is $\frac{1}{\sqrt{2\pi}\delta}$, so the equation can be rewritten as (2.11)

$$\frac{P_r(c_1 = 0 \mid r_1)}{P_r(c_1 = 1 \mid r_1)} = \frac{\frac{1}{\sqrt{2\pi}\delta} e^{\frac{-(r_1-1)^2}{2\delta^2}}}{\frac{1}{\sqrt{2\pi}\delta} e^{\frac{-(r_1+1)^2}{2\delta^2}}} = e^{\frac{2r_1}{\delta^2}} \tag{2.11}$$

Considering the channel modulation is BPSK, which transmits the value 0 as $+1$ and transmit value 1 as -1, another expression of (2.11) could be as following:

$$\begin{aligned} c_1 = 0 &\implies \text{symbol} = +1 \implies r_1 = 1 + N(0, \delta^2) \\ c_1 = 1 &\implies \text{symbol} = -1 \implies r_1 = -1 + N(0, \delta^2) \end{aligned} \tag{2.12}$$

The conventional expression of likelihood ratio can be concluded as:

$$\frac{P_r(c_1 = 0 \mid r_1)}{P_r(c_1 = 1 \mid r_1)} = e^{\frac{2r_1}{\delta^2}} \tag{2.13}$$

Calculate the log value on both sides of the formula. Eventually, we obtain the

expression of Log-Likelihood Ratio:

$$l_i = \log \frac{P_r(c_i = 0 \mid r_i)}{P_r(c_i = 1 \mid r_i)} = \frac{2}{\delta^2} \cdot r_i \tag{2.14}$$

Message passing algorithm, which exploits Tanner Graph that explained in Chapter 2.1.1 is adopted by OAI LDPC implementation. The hardware could take advantage of its recursive and iterative feature; thus, the parallelism of calculation helps to optimise the total computation latency. Two types of LLR are presented:

1. Intrinsic LLR: Passed belief on the same edge
2. Extrinsic LLR: Passed belief came from other edges

Deriving from the topology of Tanner Graph, each check node can be considered as a Single Parity Check (SPC) code. It eases the process for each check node and bit node because beliefs passing on each edge are analysed separately. The left side of the graph shown in Figure 2.6 contains Bit nodes and on the right side are all Check nodes.

Several steps to complete the first iteration in Tanner Graph:

1. The first estimate l_i comes from the channel
2. All l_i pass from Bit node i to all neighbouring Check nodes
3. Estimates for Bit 1, l_{11}, l_{13}, l_{17}, l_{19} are calculated at Check nodes c_1, c_3, c_7, c_9
4. Estimates pass from Check nodes back to neighbouring Bit nodes
5. The steps above are repeated in parallel for all nodes

After the first iteration, bit nodes start to pass the corresponding LLR alone to the neighbouring edges by taking into account together with the intrinsic LLR and extrinsic LLR. Channel LLR remains l_1 then $\vec{m_{11}} = l_1 + l_{13} + l_{17} + l_{19}$ in which l_{13}, l_{17}, l_{19} are extrinsic LLR and l_1 are considered as intrinsic LLR in replacement of l_{11}. The same rule applied on other bit nodes, e.x. $\vec{m_{17}} = l_{11} + l_{13} + l_1 + l_{19}$. Figure 2.7 shows the BN to CN operation.

As reported in Figure 2.8. The next iteration performs the same LLR passing operation from Check nodes to Bit nodes. Each Check node receives from its

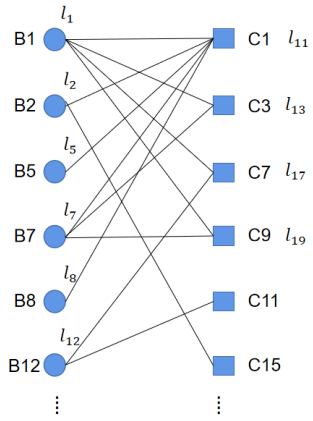

Figure 2.6: Interconnection between bit nodes and check nodes

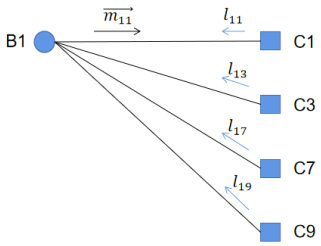

Figure 2.7: Bit nodes to Check nodes

neighbouring Bit nodes, calculates the corresponding LLR and transmit them back in parallel. Intrinsic LLR is l_{11} that replaces the received estimate of the right edge for each Check node.

2.3.1 Min-sum algorithm

Previous introduction of message passing algorithm reveals the fact that computation of LLR is sophisticated so that in hardware implementation it could be deduced that plenty resources will be wasted to support the enormous amount of calculation in case of strick demand on high throughput. Min-sum algorithm provides an approximation method on passing effortlessly the LLR back and fourth.

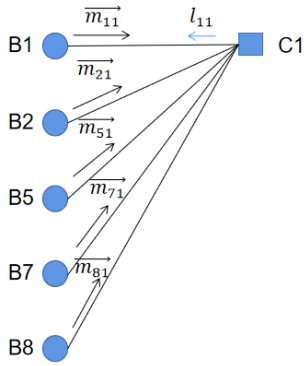

Figure 2.8: Check nodes to Bit nodes

Given an example of (3,2) code:

$$\begin{aligned} l_1 &= \log \frac{P_r(c_1 = 0 \mid r_1)}{P_r(c_1 = 1 \mid r_1)} \\ l_2 &= \log \frac{P_r(c_2 = 0 \mid r_2)}{P_r(c_2 = 1 \mid r_2)} \\ l_3 &= \log \frac{P_r(c_3 = 0 \mid r_3)}{P_r(c_3 = 1 \mid r_3)} \end{aligned} \tag{2.15}$$

Replace $P_r(c_1 = 0 \mid r_1)$ by p_1, then $P_r(c_1 = 1 \mid r_1)$ is replaced by $1 - p_1$, same rules applied on l_2 and l_3, then the equations above become:

$$l_1 = \frac{p_1}{1 - p_1}; l_2 = \frac{p_2}{1 - p_2}; l_3 = \frac{p_3}{1 - p_3} \tag{2.16}$$

In (3,2) code, one parity bit protects the other two information bits. In SPC code, one Check code is responsible for all connected message bits. In this respect, the rules that will be listed below are valid also for general SPC code in the NR LDPC decoder.

According to the relation among three codewords: $c_1 = c_2 \oplus c_3$, it can be deduced the (2.17)

$$\frac{P_1 - (1 - P_1)}{P_1 + (1 - P_1)} = \frac{P_2 - (1 - P_2)}{P_2 + (1 - P_2)} \frac{P_3 - (1 - P_3)}{P_3 + (1 - P_3)} \tag{2.17}$$

Divide the numerator and denominator by P_1, P_2 and P_3 respectively.

$$\frac{1-\frac{(1-P_1)}{P_1}}{1+\frac{(1-P_1)}{P_1}} = \frac{1-\frac{(1-P_2)}{P_2}}{1+\frac{(1-P_2)}{P_2}} \frac{1-\frac{(1-P_3)}{P_3}}{1+\frac{(1-P_3)}{P_3}} \tag{2.18}$$

Then the equation becomes:

$$\frac{1-e^{-l_{ext,1}}}{1+e^{-l_{ext,1}}} = \frac{1-e^{-l_2}}{1+e^{-l_2}} \cdot \frac{1-e^{-l_3}}{1+e^{-l_3}} \tag{2.19}$$

In order to express the conformity of each term, tanh function is adopted thanks to its monotonous characteristics that for $x < 0$, $\tanh(x) < 0$; for $x > 0$, $\tanh(x) > 0$.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1-e^{-2x}}{1+e^{-2x}} \tag{2.20}$$

The (2.20)can be re-expressed as:

$$\tanh\left(\frac{l_{\text{ext},1}}{2}\right) = \tanh\left(\frac{l_2}{2}\right) \cdot \tanh\left(\frac{l_3}{2}\right) \tag{2.21}$$

Function $sgn()$ is able to replace the $tanh()$ in our case of study because of its strong monotonicity.

$$sgn(l_{ext,1}) = sgn(l_2) \cdot sgn(l_3) \tag{2.22}$$

The following (2.23) has interesting characteristics :

$$f(x) = \left|\log\tanh(\frac{|x|}{2})\right|; \quad f^{-1}(x) = f(x) \tag{2.23}$$

Extrinsic log-likelihood ratio of one codeword can be expressed as the minimum among the absolute value of other LLR. The algorithm digs the minimum absolute

value of receiving LLR to be applied for the rest estimates as extrinsic. The second minimum absolute value is served for the minimum absolute value itself.

$$\begin{aligned} |l_{ext,1}| = f(l_2) + f(l_3) &\approx f(min(|l_2| + |l_3|)) \\ &\approx f(f(min(|l_2| + |l_3|))) \\ &\approx min(|l_2| + |l_3|) \end{aligned} \tag{2.24}$$

A significant advantage of the min-sum algorithm can be expected since it prevents the message passing from being previously calculated. The structure of the LDPC decoder can be implemented in a straightforward way which provides the possibility of achieving a good decoding speed in favour of 5G technology.

Chapter 3

Implementation of OAI LDPC Decoder

OpenAirInterface(OAI) is a french non-profit consortium that provides advanced software solutions for 5G wireless networks. They released a software emulation solution for LDPC decoder using message passing and min-sum algorithm, of which instructions and vectors exploit Advanced Vector Extension2 (AVX2) which Inter processor supports. These instructions are Single Input Multiple Data instructions (SIMD) that take vectors data type consisting of 4×64 bits integer and 2×64 bits. The primary data type is integer of 8 bits, so there are 8 integers concatenated in one vector. AVX2 instructions are able to perform operations on vectors in parallel, thus accelerate the execution time effectively.

In the following subsections, main configurations and novelties in implementation with respect to code structures and vector alignments are described in detail, and then the code structure is illustrated block by block.

3.1 Supported configurations

Two base graphs are supported by the OAI LDPC decoder. According to the 3GPP standard, BG1 is of size 46×68, BG2 is of size 42×52. All expansion factors listed in Appendix A.2 are supported. Block size is the size of the message, and total size codewords can be obtained by multiplying the column number of the base graph by the expansion factor, e.g., BG1 has 68 codewords, considering $Z = 384$, then actual codewords should be $68 \times 384 = 26112$. Code rate $1/3$ can be achieved if the

block size is 8448 since $^{8448}/_{26112} = {}^{1}/_{3}$.

Base Graph	BG1 46x68	BG2 42x52
Block Size[bits]	3841~8448	192~3840
Code Rate R	1/3 2/3 8/9	1/5 1/3 2/3

Table 3.1: Configurations of OAI LDPC Decoder

Generally, BG1 is used for a high payload because of the code rate of $^{8}/_{9}$, which means the number of codewords is close to the number of message bits, fewer parity bits are dedicated to protecting massages. On the other hand, BG2 supports a smaller code block and a code rate down to $^{1}/_{5}$, which means most parts of codewords consist of parity bits that protect strongly message bits. The expansion factor Z is determined by the equation shown is:

$$Z_c = \min_{z \in Z} \left[z > \frac{m}{N_b} \right] \tag{3.1}$$

To be noticed that in order to achieve different code rates and improve the performance, rate matching is applied by means of puncturing and shortening.

In NR LDPC, before sending the encoded message into a channel, the first $2Z_c$ bits for BG1 are always punctured and will be considered as erasures by the decoder. Thus code rate is increased. These $2Z_c$ values are not transmitted but still received by the decoder. Since they are "erasures", their corresponding LLR is equal to zero, and the decoder will manage to recover these punctured bits as if there were sent into the channel. Moreover, the rightmost part of parity bits is punctured in case a specific code rate is required. The puncturing technique has a negative impact on performance because more erased bits are supposed to be decoded apart from the erasures caused by the noise channel. However, more code rates can be supported. Therefore the trade-off between performance and code rates to be supported stands.

Shortening happens only on message bits before encoding. A certain number of zero are concatenated to the message bits to adapt the length. Since they are determined to be zero, then their LLR will be high. Before being sent to the channel, additional bits are removed from codewords. Shortening has a positive impact on performance, and those removed bits are zeros. Therefore their high LLR contributes to decoding. Two techniques are complementary in the aspect of performance so that they are usually adopted together in favour of improve performance and enrich the supported code rate.

3.2 Top-level structure

Implementation of OAI LDPC decoder is based on message passing algorithm, thus static buffers that store the input messages, output decoded messages, check node's data, bit node's data, results of bit node operation to be passed to check node buffer and results of check node operation to be passed to bit node buffer. Massive data transferring among different buffers takes place within an indicative number of iteration. Arithmetic and logic computations are done block by block. Results of each computational block are stored in specific global buffers and passed to corresponding buffers for computation of the next step by various buffer transferring functions. Stop criteria can be either achieve the maximum iteration or pass the parity check of check nodes. Two parity check functions take place soon after the LLR updates at check nodes and bit nodes.

In the following, the list of all functions used by top-level function is shown:

1. llr2llrProcBuf
2. llr2CnProcBuf
3. cnProc
4. cnProcPc
5. cn2bnProcBuf
6. bn2cnProcBuf
7. bnProcPc
8. bnProc
9. llrRes2llrOut
10. llr2bitPacked

Functions are classified in Arithmetic and logic computation models or Buffer transferring models. A summary of the LDPC decoder functions is reported in Appendix A. In top-level function, the call sequence of these functions is shown in the following chart:

Firstly, codewords are sent into the decoder after contaminated by noise; llrprofbuf is the buffer that stores the input LLR. In order to start the first iteration, data is copied into cnProcBuf. Then the function cnProc aims at calculating the new LLR

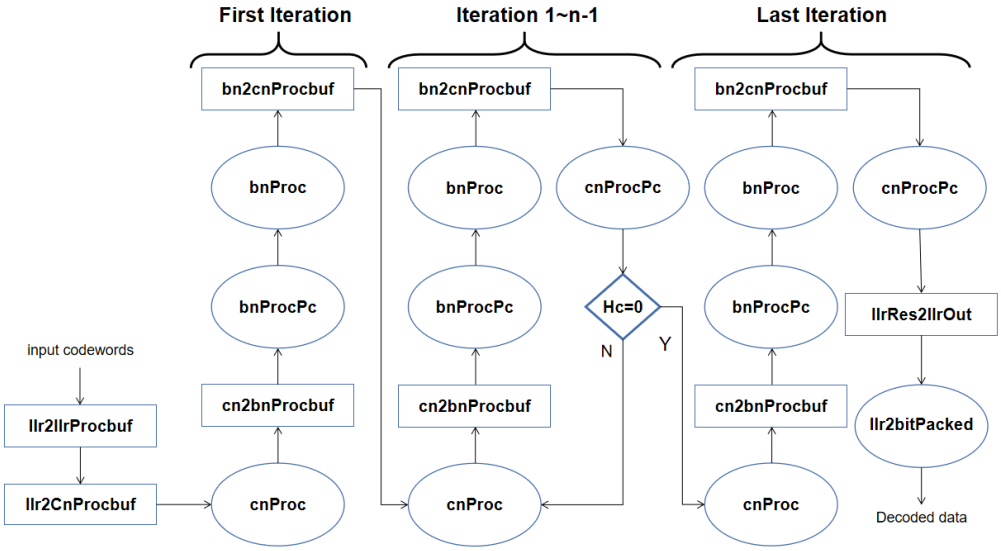

Figure 3.1: Function flow of LDPC decoder

at check nodes, and this operation takes the input LLR as neighbouring believes since that in the first iteration, there isn't any previous belief transmitted back from bit nodes. The results are stored in cnProfBufRes, namely the result of the check node process. The same nomenclature is also adopted for bit node process result buffer bnProcBufRes. Function cn2bnProcbuf copies the new believes from buffer cnProcBufRes to buffer bnProBuf, where new LLRs are calculated from part of bit nodes and stored in bnProBufRes. In parallel, a parity check of bit nodes beliefs takes place. Even though in the code structure function bnProc anticipates bnProcPc, there are no stop criteria for bit node parity check so they can be be considered independent. In function bnProcPc, results are stored in buffer llrRes. Later function bn2cnProcBuf transfers all beliefs on bit nodes back to check nodes.

After the first iteration, believes in check node buffers are recalculated again and copied back to bit node buffer, repetitive operations valid for cnProc, cn2bnProcbuf, bnProcPc, bnProc and bn2cnProcbuf, the following function cnProcPc evaluates the result of check nodes' believes and return the parity check result, in case the parity check passed, it returns 0. It will jump out of loops, namely stop the intermediate iterations and enter into the last iteration. Otherwise, the data is passed back to cnProc buffer and repeat operations in function flow until the maximum iteration reached or parity check passed.

The last iteration follows the same function flow, and the iteration counter is added by one in the end if the parity check still didn't pass. No more recursive LLR computation is performed; from the return value of testbench, we can be informed

that the current iteration number is less than the requirement. bnProcPc functions store the LLR output of the current iteration once the stopping condition meets, namely, the decoder has retrieved the original messages. Function llrRes2llrOut transfers the currently decoded codewords to llrout buffer. Then they are converted back to binary values using hard decisions.

To be noticed that the parameters of the decoder are packed in structure data type using pointers. Before performing the first step shown in Figure 3.1, the decoder is filled with proper parameters by the nrLDPC_init function, which initialize and conform to the configuration of each function block, nrLDPC_init returns the number of iteration which is passed to nrLDPC_decoder_core that carries out the function flow. The snippet of topmost function is shown in the following:

Listing 3.1: The top-level function of the OAI LDPC Decoder

```
int32_t nrLDPC_decod(t_nrLDPC_dec_params* p_decParams, int8_t* p_llr,
                     int8_t* p_out, t_nrLDPC_procBuf* p_procBuf,
                     t_nrLDPC_time_stats* p_profiler)
{
    uint32_t numLLR;
    uint32_t numIter = 0;
    t_nrLDPC_lut lut;
    t_nrLDPC_lut* p_lut = &lut;
    // Initialize decoder core(s) with correct LUTs
    numLLR = nrLDPC_init(p_decParams, p_lut);
    // Launch LDPC decoder core for one segment
    numIter = nrLDPC_decoder_core(p_llr, p_out, p_procBuf, numLLR,
                                  p_lut, p_decParams, p_profiler);
    return numIter;
}
```

3.3 Arithmetic and logic computation models

In this chapter, each function block that belongs to the arithmetic and logic computation class will be illustrated and explained in detail.

3.3.1 Check Node processing

Operation on check nodes is performed separately as illustrated in figure 2.7. Current implementation maps CNs with the same BNs connected to it into groups, namely, each CNs in one group is connected to only a small number of BNs. More

details are reported in Appendix B, Table B.1. According to the min-sum algorithm, the returned value from CN to neighbouring BNs is the overall sign value of all received BN believes multiply by the minimum among absolute values of received believes. Let q_{ij} be the value from BN j to CN i and Bi be the set of connected BNs to the ith CN, and then the algorithm can be expressed:

$$r_{ji} = \prod_{j' \in \mathcal{B}_{i \setminus j}} sgn\, q_{ij'} \min_{j' \in \mathcal{B}_{i \setminus j}} |q_{ij'}| \tag{3.2}$$

In the function cnProc, the CN process is performed group by group. In the snippet below, the CN processing of BG1 for group 5 is reported.

Listing 3.2: BG1 processing for CNs with 5 connected BNs

```
    // Process group with 5 BNs
    const uint16_t lut_idxCnProcG5[5][4] =
    {{108,216,324,432}, {0,216,324,432},
    {0,108,324,432}, {0,108,216,432},
    {0,108,216,324}};
    if (lut_numCnInCnGroups[2] > 0)
    {
        // Number of groups of 32 CNs for parallel processing
        // Ceil for values not divisible by 32
        M = (lut_numCnInCnGroups[2]*Z + 31)>>5;
        bitOffsetInGroup = (lut_numCnInCnGroups_BG2_R15[2]
                            *NR_LDPC_ZMAX)>>5;
        // Set pointers to start of group 5
        p_cnProcBuf    = (__m256i*) &cnProcBuf
                         [lut_startAddrCnGroups[2]];
        p_cnProcBufRes = (__m256i*) &cnProcBufRes
                         [lut_startAddrCnGroups[2]];
        // Loop over every BN
        for (j=0; j<5; j++)
        {
            // Set of results pointer to correct BN address
            p_cnProcBufResBit = p_cnProcBufRes + (j*bitOffsetInGroup)
    ;
            // Loop over CNs
            for (i=0; i<M; i++)
            {
                // Abs and sign of 32 CNs (first BN)
                ymm0 = p_cnProcBuf[lut_idxCnProcG5[j][0] + i];
                sgn  = _mm256_sign_epi8(*p_ones, ymm0);
                min  = _mm256_abs_epi8(ymm0);
                // Loop over BNs
                for (k=1; k<4; k++)
                {
```

```
33                          ymm0 = p_cnProcBuf[lut_idxCnProcG5[j][k] + i];
34                          min  = _mm256_min_epu8(min,
35                                 _mm256_abs_epi8(ymm0));
36                          sgn  = _mm256_sign_epi8(sgn, ymm0);
37                      }
38                      // Store result
39                      min = _mm256_min_epu8(min, *p_maxLLR);
40                      // 128 in epi8 is -127
41                      *p_cnProcBufResBit = _mm256_sign_epi8(min, sgn);
42                      p_cnProcBufResBit++;
43                  }
44              }
45          }
```

AVX2 instructions are fully exploited in the condition of casting pointers of processing buffers from *int8_t to *___m256i and of setting the group to offset adequately, to be mentioned that offset is based on the alignment of 32 bytes so that in each innermost iteration 32 check nodes proceed in parallel.

Overall there are three loops in the code section, and the first loop looks over five BNs connected to all CNs of group 5. Each BN address is given respectively, and the pointer updates for the new iteration of the outermost loop. The second loop looks over CNs bonded in 32 bytes; the total number of iterations in this loop can be calculated by multiplying the number of CNs in the corresponding group by the expansion factor Z, then add 31 ceil for value not divisible by 32. Innermost loop iterates for calculating the minimum value of BNs' LLR and their sign value. The 32byte-aligned result is updated to the corresponding address outside the innermost loop. After the completion of one set of CN, the variables ymm0 store the value of the new CN set, in data type ___mm256i, looks over again the rest of BNs connected to it and performs updates until the extrinsic LLR has been calculated. Results are stored in buffer cnProcBufRes.

3.3.2 Check Node parity check

Parity check is enabled in top-level function an by the define directive #define NR_LDPC_ENABLE_PARITY_CHECK, Same as cnProc function, the 32-byte alignment alone with computation parallelism is also exploited for parity check since parity-check of CN happens only after BN process whose following function transfers BN results to CN processing buffer, the p_cnProcBuf is the pointer to where stores the latest LLR updated in BN process. A snippet of CN group 5 for BG1 is shown in the following, M32 stands for the number of aligned CN, and Mrem is the rest nodes that do not fit in alignment. ymm0 and ymm1 are LLRs

after the BN process and CN process, respectively. The parity check operation is straightforward. Firstly it iterates over all CN nodes and set the variable pcRes to zero for each aligned CN vector, compute the parity check of neighbouring BNs and concatenate their sign bits into data type int. Intermediate value pcRes updates for each CN, outside the innermost loop, variable pcResSum computes the bit-wise OR of each parity check result and keeps updating all CN that fit in 32byte-alignment.

For the rest CNs that are considered as reminders of modulo 32, an exceptional loop is dedicated to performing parity check, but in the end, valid CN is kept, and the redundancy is shifted out. pcResSum updates outside the loop as well. In line 1712 of the snippet, an if statement enables the parity check to quit once pcResSum is higher than 0. This implementation saves the execution time significantly in software emulation because early parity check is meant to fail, and it would cost more time to finish a full check without the stopping criteria.

Listing 3.3: BG1 check node parity check for CNs of group 5

```
1    // Process group with 5 BNs
2    if (lut_numCnInCnGroups[2] > 0)
3    {
4        pcResSum = 0;
5        M = lut_numCnInCnGroups[2]*Z;
6        Mrem = M&31;
7        M32 = (M + 31)>>5;
8        // Set pointers to start of group 5
9        p_cnProcBuf    = (__m256i*) &cnProcBuf
10                        [lut_startAddrCnGroups[2]];
11       p_cnProcBufRes = (__m256i*) &cnProcBufRes
12                        [lut_startAddrCnGroups[2]];
13       // Loop over CNs
14       for (i=0; i<(M32-1); i++)
15       {
16           pcRes = 0;
17           // Loop over every BN
18           for (j=0; j<5; j++)
19           {
20               ymm0 = p_cnProcBuf    [j*216 + i];
21               ymm1 = p_cnProcBufRes[j*216 + i];
22               pcRes ^= _mm256_movemask_epi8
23                   (_mm256_adds_epi8(ymm0,ymm1));
24           }
25           // If no error pcRes should be 0
26           pcResSum |= pcRes;
27       }
28       pcRes = 0;
29       // Loop over every BN
30       for (j=0; j<5; j++)
31       {
```

```
32                ymm0 = p_cnProcBuf    [j*216 + i];
33                ymm1 = p_cnProcBufRes[j*216 + i];
34                pcRes ^= _mm256_movemask_epi8(_mm256_adds_epi8(ymm0,ymm1)
    );
35            }
36            // If no error pcRes should be 0
37            pcResSum |= (pcRes&(0xFFFFFFFF>>(32-Mrem)));
38            if (pcResSum > 0)
39            {
40                return pcResSum;
41            }
42        }
```

3.3.3 Bit Node processing

BN processing is similar to CN processing in aspects of alignment and computation parallelism. BNs are grouped in terms of the number of CNs connected to it. More details are reported in Appendix B, table B.2.2. The operation in BN processing can be written as (3.3). To be noticed that the BNs that are connected to a single CN does not need to be considered in the BN processing since the $q_{ij} = \wedge j$.

$$q_{ij} = \wedge j + \sum_{i' \in \mathcal{C}_{j \setminus i}} r_{ji'} \tag{3.3}$$

In other words, each BN accumulates all neighbouring CNs' LLR following by addition, what stores in the result buffer to be copied back to CN process buffer is the subtraction of summed value and the initial LLR came from the channel, namely extrinsic value. The BN process is performed with less complexity than the CN process. As shown in the snippet reported below. The first loop determines the address offset of the BN result buffer and old entry where contains the addition of LLRs outputted by the bnProcPc function. Looking over each BN, new entries are calculated and updated to the result buffer of BN, repetitive operation is also applied for other BN groups.

Listing 3.4: BN processing for BNs with 5 connected CNs

```
1     // Process group with 5 CNs
2     if (lut_numBnInBnGroups[4] > 0)
3     {
4         idxBnGroup++;
5         // Number of groups of 32 BNs for parallel processing
6         M = (lut_numBnInBnGroups[4]*Z + 31)>>5;
7         cnOffsetInGroup = (lut_numBnInBnGroups[4]*NR_LDPC_ZMAX)>>5;
```

```
8           p_bnProcBuf     = (__m256i*) &bnProcBuf
9           [lut_startAddrBnGroups[idxBnGroup]];
10          p_bnProcBufRes = (__m256i*)
11          &bnProcBufRes[lut_startAddrBnGroups[idxBnGroup]];
12          // Loop over CNs
13          for (k=0; k<5; k++)
14          {
15              p_res = &p_bnProcBufRes[k*cnOffsetInGroup];
16              p_llrRes = (__m256i*) &llrRes
17              [lut_startAddrBnGroupsLlr[idxBnGroup]];
18              // Loop over BNs
19              for (i=0; i<M; i++)
20              {
21                  *p_res = _mm256_subs_epi8(*p_llrRes,
22                  p_bnProcBuf[k*cnOffsetInGroup + i]);
23                  p_res++;
24                  p_llrRes++;
25              }
26          }
27      }
```

3.3.4 Bit Node parity check

Pointers of bnProcBuf is casting from *int8_t to *__m128i, instead of *__m256i, namely only 16 bytes are packed and calculated while the rest 16 bytes are stored in another intermediate variable. Both two parts are enlarged to 32bytes aligned by adding additional 16 zeros at MSB. The reason for enlargement is to increase the accuracy of total LLR. Saturation back to 8 bits takes place in the end for CN processing, but a significant loss of sensitivity is not negligible. In the following snippet, the BN process of group 5 is shown, in which there are two loops, the first one iterates over all BN and stores separately the first 16 LLR and the second 16 LLR received from the CN process, then enlarge them to 32 bytes aligned. The second loop looks over the other 4 CNs connected to the current BN, extracting and enlarging the input LLR and adding them together group by group; ymmRes0 and ymmRes1 recursively save the latest result and the intrinsic input LLR is added on result buffer outside the second loop. Results are packed back to 8 bits long and permuted in a specific way, eventually stored in buffer llrRes. bnProcPc function is followed by bnProc, which is explained in the last sub-chapter. It is observed that in the source code of current implementation the shown segment of code is repeated 30 times, with minor modifications on loop bounds that are increased depends on the number of groups. From the perspective of all supported configurations, half of the group numbers are not valid, although the functionality remains good. The

counter idxBnGroup showed in line 4 self-increases only in case the if statement in line 2 stands. In fact, there are 11 groups never been triggered. Group number is increased to 15 if we enable BG1 only. Therefore, code cancellation has been performed at both functions bnProc and bnProcPc, whilst the code functionality is verified to be unchanged.

Listing 3.5: BN Parity Check for BNs with 5 connected CNs

```
1   // Process group with 5 CNs
2   if (lut_numBnInBnGroups[4] > 0)
3   {
4       idxBnGroup++;
5       M = (lut_numBnInBnGroups[4]*Z + 31)>>5;
6       cnOffsetInGroup = (lut_numBnInBnGroups[4]*NR_LDPC_ZMAX)>>4;
7       p_bnProcBuf     = (__m128i*) &bnProcBuf[lut_startAddrBnGroups
8                         [idxBnGroup]];
9       p_llrProcBuf = (__m128i*) &llrProcBuf
10      [lut_startAddrBnGroupsLlr[idxBnGroup]];
11      p_llrRes     = (__m256i*) &llrRes
12      [lut_startAddrBnGroupsLlr[idxBnGroup]];
13      // Loop over BNs
14      for (i=0,j=0; i<M; i++,j+=2)
15      {
16          ymmRes0 = _mm256_cvtepi8_epi16(p_bnProcBuf[j]);
17          ymmRes1 = _mm256_cvtepi8_epi16(p_bnProcBuf[j+1]);
18          // Loop over CNs
19          for (k=1; k<5; k++)
20          {
21              ymm0 = _mm256_cvtepi8_epi16(p_bnProcBuf[
22                                  k*cnOffsetInGroup + j]);
23              ymmRes0 = _mm256_adds_epi16(ymmRes0, ymm0);
24              ymm1 = _mm256_cvtepi8_epi16(p_bnProcBuf[
25                                  k*cnOffsetInGroup +j+1]);
26              ymmRes1 = _mm256_adds_epi16(ymmRes1, ymm1);
27          }
28          ymm0    = _mm256_cvtepi8_epi16(p_llrProcBuf[j]);
29          ymmRes0 = _mm256_adds_epi16(ymmRes0, ymm0);
30          ymm1    = _mm256_cvtepi8_epi16(p_llrProcBuf[j+1]);
31          ymmRes1 = _mm256_adds_epi16(ymmRes1, ymm1);
32          *p_llrRes = _mm256_permute4x64_epi64(ymm0, 0xD8);
33          p_llrRes++;
34      }
35  }
```

3.3.5 Hard-decision on output LLR

After data flows through decoding and transferring functions mentioned in Figure 3.1, one of the two stopping criteria meets. LLRs are considered reliable; namely, the initial message bits are recovered. As explained in Chapter 2.3, the hard decision takes place soon after the soft-out phase. There are three outmodes supported by OAI LDPC decoder implementation:

1. nrLDPC_outMode_BIT: 32 bits per unit32_t output
2. nrLDPC_outMode_BITINT8: 1 bit per int8_t output
3. nrLDPC_outMode_LLRINT8: Single LLR value per int8_t output

Three outmodes are defined in enum data type and included by the structure of parameters. On the top-level function, an if-else statement is used for selecting one of these outmodes. The first outmode is selected by default which is shown in the following snippet, the function takes the output LLR as input and the decoded values as output. Operations are performed in ___m256i whilst the data type of decoded message is uint32_t. Constant vector constShuffle_256_epi8 adopts the 3GPP technical specification [18], every 32 bytes of output LLR is shuffled so as for reordering. A sign bit of each 8 bits number of the intermediate variable inPerm is extracted and packed together to form a 32 bits value for output. Since the module uses BPSK modulation, for each 8 bits LLR, if it is positive, the decoded bit is zero. Otherwise, the decoding result should be 1.

In the end, the remaining bits that do not fit in 32 bytes alignment will be decided one by one.

Listing 3.6: Hard-decision function

```
1  static inline void nrLDPC_llr2bitPacked(int8_t* out, int8_t* llrOut,
       uint16_t numLLR)
2  {
3      const uint8_t constShuffle_256_epi8[32] __attribute__ ((aligned
       (32))) =
       {7,6,5,4,3,2,1,0,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0,15,14,13,12
4       ,11,10,9,8};
5      __m256i*  p_llrOut = (__m256i*)  llrOut;
6      uint32_t* p_bits   = (uint32_t*) out;
7      __m256i inPerm;
8      int8_t* p_llrOut8;
9      uint32_t bitsTmp = 0;
10     uint32_t i;
```

```
11      uint32_t M  = numLLR>>5;
12      uint32_t Mr = numLLR&31;
13      const __m256i* p_shuffle = (__m256i*) constShuffle_256_epi8;
14      for (i=0; i<M; i++)
15      {
16          inPerm = _mm256_shuffle_epi8(*p_llrOut,*p_shuffle);
17          // Hard decision
18          *p_bits++ = _mm256_movemask_epi8(inPerm);
19          p_llrOut++;
20      }
21      if (Mr > 0)
22      {
23          p_llrOut8 = (int8_t*) p_llrOut;
24          for (i=0; i<Mr; i++)
25          {
26              if (p_llrOut8[i] < 0)
27              {
28                  bitsTmp |= (1<<((7-i) + (16*(i/8))));
29              }
30              else
31              {
32                  bitsTmp |= (0<<((7-i) + (16*(i/8))));
33              }
34          }
35      }
36      *p_bits = bitsTmp;
37  }
```

3.4 Buffer transferring models

In order to process with AVX2 instructions, the data needs to be aligned in a particular manner. Starting address and offset for each processing is well organized. In all cases of storing results into their corresponding result buffer, values cannot be used by the following function whilst the condition of exploit AVX2 instructions is satisfied. Considering the different grouping indices of BN and CN, buffer transferring functions have to be introduced.

The speed-up in computation with AVX2 instructions is much more than the exceptional wasted time on buffer transferring. However, this statement stands only for software implementation. In hardware implementation, the drawback of a sophisticated passing algorithm is mainly in the degeneration of latency. Graphical representation of two types of memory copy is shown in Figure 3.2. The left one is an inverted-circular copy, and the right one is a circular copy. Every edge in

the BG is a circular shift of a $Z \times Z$ identity matrix. Therefore, depending on the circular shift in the BG definition and value of expansion factor Z, the shift parameters are defined diversely. All shift factors are stored in nrLDPC_lut.h in array circShift_BGx_Zx_CNGx, where CNG is the CN group. A small segment of LUT is reported in Appendix B.3.

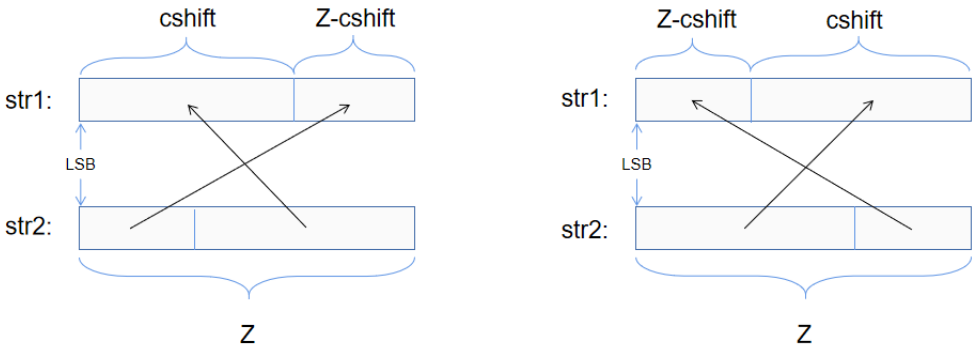

Figure 3.2: Inverted-circular copy(left) vs. Circular copy(right)

Data transferring among buffers shares almost the same code structure since they copy data in 8 bits. Moreover, the arrays are arranged depending only on the CN group for convenience. In the first loop, bit offset and starting address for each iteration are defined, function nrLDPC_circ_memcpy performs Z times single copying in 8bits according to the shift factor. Loop bound of the innermost loop is determined by the number of CN in one CN group. Every time one transferring function is called, data in one buffer is copied in another one entirely without any remaining bit.

Listing 3.7: Memory transferring function

```
1     // CN group with 5 BNs
2     bitOffsetInGroup = lut_numCnInCnGroups_BG1_R13[2]*NR_LDPC_ZMAX;
3     for (j=0; j<4; j++)
4     {
5         p_cnProcBuf = &cnProcBuf[lut_startAddrCnGroups[2]
6                       + j*bitOffsetInGroup];
7         for (i=0; i<lut_numCnInCnGroups[2]; i++)
8         {
9             idxBn = lut_startAddrBnProcBuf_CNG5[j][i] +
10                    lut_bnPosBnProcBuf_CNG5[j][i]*Z;
11            nrLDPC_circ_memcpy(p_cnProcBuf, &bnProcBufRes[idxBn],
12                               Z, lut_circShift_CNG5[j][i]);
13            p_cnProcBuf += Z;
14        }
15    }
```

Chapter 4

Stand-alone AVX2-based synthesizable decoder model

In the last chapter, the code structure and main functions are illustrated and explained in detail. OAI LDPC decoder aims at providing software solutions for simulation. In the top-level function, several measurements are enabled through conditional compilation. Macro #ifdef NR_LDPC_PROFILER_DETAIL enables the function start_meas(), which obtains the current system time. It has been inserted before and after each function in order to calculate the meantime of execution. Moreover, macro #ifdef NR_LDPC_DEBUG_MODE occurs after the call of each function, inside functions nrLDPC_debug_initBuffer2File and nrLDPC_debug_writeBuffer2File are called in sequence to initialize the intermediate storage buffer and store the intermediate value into the buffer in favour of validating the correctness of data output after each step of the decoding process. The internal structure and implementation of these auxiliary functions will not be discussed since it is out of the scope of this thesis. An example of how they are called in top-level function is reported in Appendix B.4.

Vivado HLS 2018.2 was chosen to perform the high-level synthesis, namely, transform the source code in C directly to HDL, however, the tool does not accept all syntax of C-like code. The constraints are:

1. System calls are not supported
2. Dynamic memory usage is not supported
3. Recursive functions are not synthesizable

4. Pointer casting, function pointers and arrays of pointers that point to additional pointers are not supported

A stand-alone model of the decoder is extracted from the whole project, which also contains the implementation of the encoder. Measurement functions are removed accordingly since the code will be transformed in the format of HDL. Software-based measurement is no longer required in the stand-alone model. Additionally, the implementation of measurement functions is based on system calls, e.g. fprintf() for writing the validation data into a buffer, sleep() for hanging up the current process for a specified time. These functions are not synthesizable for the HLS tool, so they are removed from the stand-alone model.

Figure 4.1 reports the errors relevant to the unsynthesizability of the stand-alone decoder model. These errors can be divided into double or triple pointers, pointer casting, array initialization, and memory copying.

```
ERROR: [SYNCHK 200-41] ../../LDPC/AVX2/HLS_test/nrLDPC_bnProc.c:2800: unsupported pointer
reinterpretation from type 'i8*' to type 'i32*' on variable 'out'.
ERROR: [SYNCHK 200-61] ../../LDPC/AVX2/HLS_test/nrLDPC_bnProc.c:2825: unsupported memory access on
variable 'llrOut' which is (or contains) an array with unknown size at compile time.
ERROR: [SYNCHK 200-11] ../../LDPC/AVX2/HLS_test/nrLDPC_cnProc.c:46: Argument 'p_procBuf.cnProcBuf' of
function 'nrLDPC_decoder' (../../LDPC/AVX2/HLS_test/nrLDPC_decoder.c:43) has an unsynthesizable type
(possible cause(s): pointer to pointer or global pointer).
ERROR: [SYNCHK 200-22] ../../LDPC/AVX2/HLS_test/nrLDPC_decoder.c:90: memory copy is not supported
unless used on bus interface possible cause(s): non-static/non-constant local array with
initialization).
ERROR: [SYNCHK 200-43] ../../LDPC/AVX2/HLS_test/nrLDPC_decoder.c:386: use or assignment of a non-
static pointer 'llrOut' (this pointer may refer to different memory locations).
ERROR: [SYNCHK 200-11] ../../LDPC/AVX2/HLS_test/nrLDPC_cnProc.c:438: Variable 'ymm0' has an
unsynthesizable type 'm256i' (possible cause(s): structure variable cannot be decomposed due to (1)
unsupported type conversion; (2) memory copy operation; (3) function pointer used in struct; (4)
unsupported pointer comparison).
```

Figure 4.1: Error report of stand-alone decoder model

4.1 Destruction of struct

Vivado HLS does not support an array of pointers or a structure of pointers that point to another pointer. Namely, the synthesizer cannot interpret such code structure in C-like languages to HDL language. In order to eliminate the constraint, structures that contain pointers or double pointers must be broken up into single pointers that are called and passed into functions separately. In the source code, there are two structures of pointers defined in header file nrLDPC_types.h, and the first structure includes all necessary pointers that point to global vectors that store the corresponding address and offset depending on the LDPC decoder configuration. This structure is firstly passed into function nrLDPC_init described in chapter 3.2 to specify the read-only vectors used by the configured decoder, and then it is

passed into the decoder_core function, which contains the specific functionality of decoding.

Listing 4.1: Pointer structure of LUT

```
typedef struct nrLDPC_lut {
    const uint32_t* startAddrCnGroups;
    /**< Start addresses for CN groups in CN processing buffer */
    const uint8_t*  numCnInCnGroups;
    /**< Number of CNs in every CN group */
    const uint8_t*  numBnInBnGroups;
    /**< Number of CNs in every BN group */
    const uint32_t* startAddrBnGroups;
    /**< Start addresses for BN groups in BN processing buffer  */
    const uint16_t* startAddrBnGroupsLlr;
    /**< Start addresses for BN groups in LLR processing buffer  */
    const uint16_t** circShift[NR_LDPC_NUM_CN_GROUPS_BG1];
    /**< LUT for circular shift values for all CN groups and Zs */
    const uint32_t** startAddrBnProcBuf[NR_LDPC_NUM_CN_GROUPS_BG1];
    /**< LUT of start addresses of CN groups in BN proc buffer */
    const uint8_t**  bnPosBnProcBuf[NR_LDPC_NUM_CN_GROUPS_BG1];
    /**< LUT of BN positions in BG for CN groups */
    const uint16_t* llr2llrProcBufAddr;
    /**< LUT for transferring input LLRs to LLR processing buffer */
    const uint8_t*  llr2llrProcBufBnPos;
    /**< LUT BN position in BG */
    const uint8_t** posBnInCnProcBuf[NR_LDPC_NUM_CN_GROUPS_BG1];
    /**< LUT for llr2cnProcBuf */
} t_nrLDPC_lut;
```

Pointers in the structure are global and independent one from another. Their keyword 'const' reveals the invariability of the selected vectors for one execution of the decoder so that they can be extracted from the structure by means of destruction then redeclared both 'const' and 'static'. Therefore vectors and parameters are protected from access by other source files except for the decoder. Moreover, the declaration and definition of both nrLDPC_init and decoder_core functions are shifted to header files in favour of the authority of accessing static data. Pointers are no longer passed into function bodies but called directly by sub-functions since they are global from the perspective of the top-level decoder function. Reformed pointers are reported in the following.

Listing 4.2: Destruction of pointers

```
static const uint32_t* startAddrCnGroups;
static const uint8_t*  numCnInCnGroups;
static const uint8_t*  numBnInBnGroups;
static const uint32_t* startAddrBnGroups;
static const uint16_t* startAddrBnGroupsLlr;
```

```
6  static const uint16_t* circShift[9];
7  static const uint32_t* startAddrBnProcBuf[9];
8  static const uint8_t*  bnPosBnProcBuf[9];
9  static const uint16_t* llr2llrProcBufAddr;
10 static const uint8_t*  llr2llrProcBufBnPos;
11 static const uint8_t* posBnInCnProcBuf[9];
```

Additionally, pointers to processing buffers are contained by a structure that is declared in the same file where pointers of LUT are defined. The main difference between pointers to processing buffers and pointers to LUT is that buffers are repeatedly read and written. Instead, configuration vectors are read-only and addressed only once. LUTs are declared and initialized in a specific file so that their corresponding pointer can address the needed vector and read the value. The source code edited by the author does not initialize the buffers in the source code of the decoder. Strangely they are initialized in the testbench file. Namely, the pointers are wild pointers without calling the testbench. It would be understandable, considering the original aim of the OAI implementation is to provide a reliable test platform for the LDPC decoder. However, the C code design for HLS must contain the entire functionality. Destruction of the following code leads to array uninitialized error, which is further explained in chapter 4.3.

Listing 4.3: Pointer structure of processing buffers

```
1 typedef struct nrLDPC_procBuf {
2     int8_t* cnProcBuf; /**< CN processing buffer */
3     int8_t* cnProcBufRes; /**< Buffer for CN processing results */
4     int8_t* bnProcBuf; /**< BN processing buffer */
5     int8_t* bnProcBufRes; /**< Buffer for BN processing results */
6     int8_t* llrRes; /**< Buffer for LLR results */
7     int8_t* llrProcBuf; /**< LLR processing buffer */
8 } t_nrLDPC_procBuf;
```

The only solution is to initialize the buffers with a fixed size and destruct the pointer structure, and then all buffer pointers are eliminated due to the pointer constraints on HLS.

From the decoder's point of view, processing buffers which store the processing and result data are local. In other words, data stored in buffers are non-volatile during the execution. All sub-functions access these local buffers for the aims of performing partial operations of decoding. Temporary data is kept at the end of each step and passed to the next step. Thus they can be considered as global buffers from the top-level's point of view. Therefore, buffers are initialized in top-level function with keyword 'static' because we do not want other programs to control them. Additionally, an FPGA possesses fix-sized local memories on board, namely BRAM

or UltraRAM, where data of significant size is stored. In the current implementation of the LDPC decoder, the size of buffer cnProcBuf, cnProcBufRes, bnProcBuf and bnProcBufRes is 121344 bits, and the buffer llrRes and llrProcBuf are of size 27000 bits. Each of these buffers is distributed in static RAMs on board favouring other HLS techniques on memory, which help achieve high throughput.

Listing 4.4: Initialization of processing buffers

```
1     static int8_t buf_llrProcBuf[27000];
2     static int8_t buf_cnProcBuf[121344];
3     static int8_t buf_cnProcBufRes[121344];
4     static int8_t buf_llrRes[27000];
5     static int8_t buf_bnProcBuf[121344];
6     static int8_t buf_bnProcBufRes[121344];
```

4.2 Pointer Casting

AVX2 solution is based on the parallelism of computation and data alignment of 32 or 26 bytes. Two main difficulties of transforming the source code to the synthesizable version: First, as an instruction set used for Intel processor, AVX2 is not supported by the HLS tool, so all AVX2 instructions must be rewritten in nature C by keeping the same functionality. Second, pointer casting from/to int8_t to/from ___mm256i must be removed and replaced by passing arguments in the same data type. This work is error-prone and time-consuming because addresses and offsets of buffers need to be recalculated to satisfy data alignment.

4.2.1 AVX2 instructions in C language

Single Instruction Multiple Data(SIMD) mode is the characteristic of the AVX2 instruction set. Instead of performing an arithmetical or logical calculation in scalar, it provides a parallel approach of performing multiple operations on aligned data. Data type ___mm256i consists of 256 bits, 'i' stands for the int data type. In order to construct one data of ___mm256i, there could have 4×long(64 bits), 8×int(32 bits), 16×short(16 bits) or 32×char(8 bits). AVX2 instruction can perform an operation on one or more data of AVX2 data type with the same latency of the corresponding scalar mode operation, namely accelerator of X depends on the original data type listed above.

Synopsis and descriptions of instructions used in LDPC decoder source code are listed in Appendix C.1.

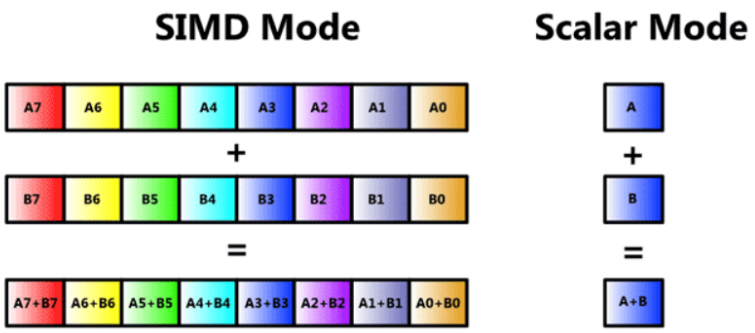

Figure 4.2: SIMD Mode vs. Scalar Mode

The conversion of instruction starts from the reconstruction of data type ___mm256i, which could be composed basing on various C nature data types. In this project, the original message is of data type int8_t, which is supposed to be the base data type for construct new ___mm256i. The following code shows how the new data type is defined.

Listing 4.5: New data type m256i for replacing ___mm256i

```
1 typedef struct mm256i {int8_t data[32] __attribute__ ((aligned(32)))
    ;} m256i;
2 typedef struct mm128i {int8_t data[16] __attribute__ ((aligned(16)))
    ;} m128i;
```

By definition, m256i consists of one array composed of 32×int8_t with an alignment of 32 bytes. One byte has the same size as int8_t. AVX2 instructions are rewritten in functions adopting the new data type. In Appendix C.2 comparison between the new synopsis of converted functions and their original version are reported. The method of conversion is straightforward since each function performs a simple functionality. As an example, the internal implementation of function mm256_min_epu8 is given. Two data of m256i are passed by value into the function body, and the function returns one m256i data. Inside the function body, temporary variable dest is initialized by filling in all '0'. In the loop with 32 times iteration, each aligned 8 bits value is forced transformed to unsigned in order to perform the comparison, the minimum value between a and b will be stored in the corresponding position of dest. After the iteration of 32 times. All 256 bits packed in new data type m256i are returned for further operations that receive the same data type. In summary, for performing the conversion on the instruction group, the coherence of returned value's and input parameter's data types has to be taken into account.

Listing 4.6: Example of new AVX2-like function

```
1 m256i mm256_min_epu8(m256i a, m256i b){
```

```
    m256i dest =
    {{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0}};
    int i;
    for(i=0; i<32; i++){
        if((uint8_t)a.data[i] >(uint8_t)b.data[i])
        dest.data[i] = b.data[i];
        else
        dest.data[i] = a.data[i];
    }
    return dest;
}
```

By adopting the new functions and new data type passed through the chain of executive functions, the remaining problem in CN and BN process is about packing 256 bits of aligned data into each iteration. The compiler is too stupid to interpret pointer casting directives which aims at obtaining data of converted length by directly manipulating the memory. Thus such operation needs to be performed manually, namely create a new function to store the consecutive 32 bytes data into an intermediate variable of data type m256i. Besides, in the case of the BN process, data is fetched in the alignment of 16 bytes. Synopsis and description of functions mentioned above are reported in Table 4.1.

Num.	Synopsis	Description
1	m256i mm256_conv_int8(int8_t* a)	convert pointers int8_t* into returned m256i
2	m128i mm128_conv_int8(int8_t* a)	convert pointers int8_t* into returned m128i
3	int8_t* mm32_store_int8(uint32_t a, int8_t* b)	Store a 32 bits value into sequential pointers of int8_t*

Table 4.1: Additional functions in favour of synthesizability

The last function mm32_store_int8 replaces storing function which includes pointer casting from *uint32_t to *int8_t. More details will be discussed in the next chapter.

Listing 4.7: Pseudo-code of partial CN processing

```
for (j=0; j<4; j++)
{
    // Loop over CNs
    for (i=0; i<M; i++)
    {
        ymm0 = mm256_conv_int8(&buf_cnProcBuf[offset_1]);
        sgn  = mm256_sign_epi8(ones256_epi8, ymm0);
        min  = mm256_abs_epi8(ymm0);
        // Loop over BNs
        for (k=1; k<3; k++)
```

```
11          {
12              ymm0 = mm256_conv_int8(&buf_cnProcBuf[offset_2]);
13              min  = mm256_min_epu8(min, mm256_abs_epi8(ymm0));
14              sgn  = mm256_sign_epi8(sgn, ymm0);
15          }
16          // Store result
17          min = mm256_min_epu8(min, maxLLR256_epi8);
18          // 128 in epi8 is -127
19          tmp = mm256_sign_epi8_ori(min, sgn);
20
21      }
22  }
```

Taking the above pseudo-code of one group of CN processing as a reference, the main idea of enabling synthesizability is illustrated in Figure 4.3. Firstly, 32×int8_t data in CN processing buffer is extracted through new address and offsets, mm256_conv_int8 packs the data into 32 bytes aligned vector. The rest of nested loops 1-2 is performed without casting problem because all computations are carried out in m256i. The result is passed to the first port of the starting function of the second loop, the input value of the second port can be obtained by adopting the same method. In conclusion, instead of performing pointer casting on the processing buffer, data is fetched in an intermediate variable with the same way of alignment.

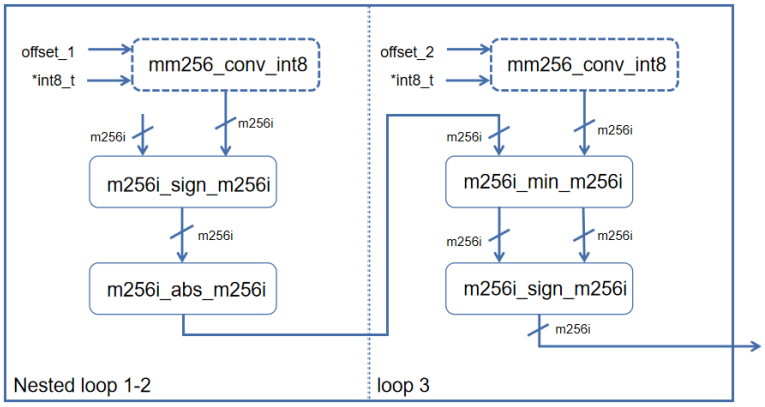

Figure 4.3: Partial pseudo-flow of CN processing

4.2.2 Re-addressing in computation models

By analysing the repeated code segment in CN and BN processing, only one fragment of data stored in the buffer is sent into the computational function for

each iteration, 32 bytes vector in m256i in case of CN processing and two 16 bytes vector in case of BN parity check processing because of sign extension in favour of high data accuracy. The LDPC decoder is quasi-cyclic. Therefore LLRs of the same CN or BN group are placed in order from the minimum group number to the maximum number, i.e. starting address of CN group 3 is 0, starting address of CN group 4 is 1152, and for CN group 5 the starting address is 8832. Take the listing 3.2 as an example. The variable M is the number of groups of 32 CNs, the intermediate pointers p_cnProcBuf and p_cnProcBufRes are cast from int8_t to m256i, which means the offset of new m256i pointers is 32 times the offset of original pointers. In the previous chapter, we have discussed two types of additional functions whose input is a pointer of int8_t and output is m256i. Hence intermediate pointers can be dropped, and the packed data used by iterative computation is expressed directly by the processing buffer itself with proper address. The following snippet reports the synthesisable version of case group 5 in CN processing.

Initially, the constant two-dimensional vector lut_idxCnProG5 remains unchanged even though these indices are served for pointer-cast buffers. Outside the first loop, variable M and bitoffsetInGroup contain AVX2-based offset as well. These indices are supposed to be weighted by 32 in order to fetch the correct data segment.

In the first loop, intermediate pointer p_cnProcBufResBit is eliminated, but the offset is reserved for re-addressing the cnProcBufRes buffer in which the result is stored.

In the second loop, ymm0 is obtained by adding the starting address of the CN processing buffer and the offset with proper weight. To be noticed that the loop bound is based on 256 bits aligned data. Thus the iteration counter i needs to be weighted by a factor of 32. The same rule is valid also for lut_idxCnProG5. No more modification is needed for the following functions in the current loop because they all take m256i as inputs and returned values.

The third loop is similar to the second loop. Firstly the re-addressing of the CN processing buffer is performed in order to extract the 32×8bits data to be stored in ymm0. The rest of the functions remains the same. Outside the third loop, the LLRs are calculated and stored in the result buffer. In the previous implementation, the buffer cnProcBufRes is cast to 256 bits aligned and stores 256 bits in every iteration. The carried out solution is storing the iterative result in one transitory buffer of data type m256i, then the fourth loop is added in parallel with the third loop, offsets are weighted by 32 except the starting address and the iterating counter of the current fourth loop. Inside the loop, data is passed from the tmp buffer to the CN result buffer one by one. HLS tool is able to flatten the loop and perform the iterative operations in parallel. Additionally, Vivado HLS is capable of nesting

the loops between which there is no directive in favour of pipelining.

Listing 4.8: Synthesizable CN processing function

```
// Process group with 5 BNs
const uint16_t lut_idxCnProcG5[5][4] =
{{216,432,648,864},{0,432,648,864},
   {0,216,648,864},{0,216,432,864},
                   {0,216,432,648}};
if (lut_numCnInCnGroups[2] > 0)
{
    M = (lut_numCnInCnGroups[2]*Z + 31)>>5;
    bitOffsetInGroup = (lut_numCnInCnGroups_BG1_R13[2]
                        *NR_LDPC_ZMAX)>>5;
    for (j=0; j<5; j++)
    {
        for (i=0; i<M; i++)
        {
          ymm0 = mm256_conv_int8(&buf_cnProcBuf
                 [lut_startAddrCnGroups_BG1[2] +
                  lut_idxCnProcG5[j][0]*32 + i*32]);
          sgn  = mm256_sign_epi8(ones256_epi8, ymm0);
          min  = mm256_abs_epi8(ymm0);
            for (k=1; k<4; k++)
            {
              ymm0 = mm256_conv_int8(&buf_cnProcBuf
                     [lut_startAddrCnGroups_BG1[2] +
                      lut_idxCnProcG5[j][k]*32 + i*32]);
                min  = mm256_min_epu8(min, mm256_abs_epi8(ymm0));
                sgn  = mm256_sign_epi8(sgn, ymm0);
            }
            min = mm256_min_epu8(min, maxLLR256_epi8);
            // 128 in epi8 is -127
            tmp = mm256_sign_epi8_ori(min, sgn);
            for(l=0; l<32; l++)
            {
              buf_cnProcBufRes[lut_startAddrCnGroups_BG1[2] +
              (j*bitOffsetInGroup*32) + i*32+l] = tmp.data[l];
            }
        }
    }
}
```

The same method is applied to the rest of CN processing, CN parity check, BN processing, and BN parity check since they share the same basic implementation principle. Another type of pointer casting appears in the hard decision function nrLDPC_llr2bitPacked, where the 32 bits data is divided into 4×8bits and passed sequentially into llrRes buffer. The original implementation is casting the result

buffer of LLR to ∗uint32 then passing the result into it. The third function reported in Table 4.1 takes the 32 bits value and the pointer which points to result buffer as input, then perform bit shifting four times to obtain all independent 8 bits values which are copied into destination in sequence. The function returns the updated pointer for iterative operation.

4.3 Pointer constraints and memory copying

From experience gathered from previous solutions, passing by value is more reliable than passing by reference due to massive constraints on pointers. Strange problems may occur on pointers, and it is hard to find out why. Therefore, data buffers should be expressed and passed in a straightforward way, avoiding as much as possible the usage of pointers. Moreover, a pointer can ONLY point to one buffer. Synthesis compiler is not able to determine which buffer it refers to in case the pointer points to different buffers conditionally. On top-level function the pointer p_llrout intended to either point to the output buffer if outmode is set to LLRINT8, or point to llrProcBuf in order to use LLR processing buffer as a temporary output buffer. The solution is to abandon the pointer and express the two cases by corresponding buffers explicitly. The relevant code snippet is reported in Appendix C.4. The next class of unsupported C type code is related to memory copying. The function memcpy() in C library <string.h> is not supported by High Level Synthesis. One straightforward solution is proposed:

Listing 4.9: New memory copy function

```
static void memcpy_syn(int8_t str1[], int8_t str2[], uint16_t n)
{
    uint16_t i;
    for(i=0; i<n; i++)
      {
       str1[i] = str2[i];
      }
     return;
}
```

The new function replaces the unsupported version maintaining the same functionality. The code snippet reported in listing 3.7 is converted as follows:

Listing 4.10: New memory transferring function

```
    bitOffsetInGroup = lut_numCnInCnGroups_BG1_R13[4]*NR_LDPC_ZMAX;
    for (j=0; j<6; j++)
    {
```

```
4           for (i=0; i<lut_numCnInCnGroups[4]; i++)
5           {
6               idxBn = lut_startAddrBnProcBuf_CNG7[j][i] +
7                       lut_bnPosBnProcBuf_CNG7[j][i]*Z;
8                       nrLDPC_circ_memcpy(&buf_cnProcBuf[
9                       lut_startAddrCnGroups_BG1[4] +
10                      j*bitOffsetInGroup + Z*i],
11                      &buf_bnProcBufRes[idxBn], Z,
12                      lut_circShift_CNG7[j][i]);
13          }
14      }
```

Since memory copying is performed only on int8_t, the reason for which intermediate pointer is eliminated is to keep the same code style as processing functions. Updates of pointer for next iteration is replaced by the additional offset Z×i, which keeps the functionality unchanged.

So far, all aspects of unsynthesizability have been discussed, starting from the pointer structure, which is considered as double or triple pointers, to the pointing casting issue which prevents the synthesis compiler from interpreting the C source code. Moreover, pointers are removed as much as possible due to the massive constraints and the conflict on a different object to which one pointer points. Source code has been modified in order to support synthesizability, especially the new address and offset of buffers without casting and the unsupported functions from the C library. The code is completely synthesizable and is sent to Vivado HLS for further acceleration from the perspective of hardware implementation.

Chapter 5

Hardware acceleration via High-Level Synthesis tool

HLS tool provides a powerful mechanism for automatically synthesizing software implementation (C, C++ or SystemC) to FPGA hardware implementation. In order to obtain an efficient implementation, namely a good trade-off between area and throughput. Further optimization needs to be applied after the software design's synthesizability and functionality is verified. It has to be performed more code modification according to a certain coding style, which guarantees high efficiency. Moreover, the HLS tool provides a set of pragmas in favour of optimizing the design without changing a lot in C source code.

The synthesis report provides an estimation of performance in terms of clock frequency and latency of the whole design and each non-inline function block. The hardware utilization from the perspective of different components of the whole design is also reported in detail, i.e. Instance, memory, expressions etc. The available hardware units are fixed on the specific FPGA board, which differs from the model of the board. For a practical and efficient design, the selection of the board model has to be taken into account because the different board contains different available hardware basic units, which may bring limitation in the process of optimization. For the current design, initially, no specific board is selected. The synthesis is running on the default board of Vivado HLS 2018.2 because of the undetermined loop bound, which prevents the synthesizer from generating the latency of the loops. In this step, the priority is to specify statically a loop bound by inserting the trip count directive or using the maximum loop bound alone with asserting stopping criteria internally. More details will be introduced in the next chapter.

Description of the reported basic units of a general Xilinx FPGA board is listed:

1	**BRAM**	Basic on-chip storage unit, single port or dual port, with two sets of reading and writing data, address and control buses
2	**DSP**	DSP chip refers to the chip that can realize digital signal processing technology, which can complete one multiplication and one addition in one instruction cycle
3	**FF**	Flip-Flop(Register) stores the result of the LUT
4	**LUT**	LUT is the basic building block of FPGA, which can realize any function of N Boolean variables by combining N outputs into a certain function and generating output values
5	**URAM**	It is also a storage unit, but only the UltraScale Plus chip(Xilinx) has UltraRAM, which has a larger capacity and deeper width than BRAM.

Table 5.1: Basic units on Xilinx FPGA

The first step of optimization is burst reading from off-chip memory and writing to on-chip one since the input data read from the interface undergoes iterative operations. On-chip memory(BRAM or URAM) can be partitioned in a way that computation or copying can be performed in parallel. Thus the latency is reduced, and performance is increased. Function memcpy is supported only when data is copied from or to a top-level function argument specified with an $AXI-4$ interface. In the top-level function of the current implementation, input LLRs of total size 26112 bits are read from off-chip memory through the m_axi port. The output of the decoder has a totally of 8448 bits, and the local buffer is initialized for temporarily keeping the output. Eventually, the decoder output is copied to the output port on the interface. Input LLRs are not copied back because they will not be used anymore after successful decoding.

Listing 5.1: Reading from Interface and writing on BRAM

```
1  int32_t nrLDPC_decoder(t_nrLDPC_dec_params* p_decParams, int8_t p_llr
       [], int8_t p_out[])//, t_nrLDPC_time_stats* p_profiler)
2  {
3  #pragma HLS INTERFACE s_axilite port=return
4  #pragma HLS INLINE off
5  #pragma HLS INTERFACE m_axi depth=26112 port=p_llr
6  #pragma HLS INTERFACE m_axi depth=8448 port=p_out
7      uint32_t numLLR;
8      uint32_t numIter = 0;
9      int8_t buff0[8448];
10     int8_t buff1[26112];
11     int8_t *p_buff0 = buff0;
12     int8_t *p_buff1 = buff1;
```

```
13      memcpy(p_buff0, p_out, 8448);
14      memcpy(p_buff1, p_llr, 26112);
15      // Initialize decoder core(s) with correct LUTs
16      numLLR = nrLDPC_init(p_decParams);
17      // Launch LDPC decoder core for one segment
18      numIter = nrLDPC_decoder_core(buff1, buff0, numLLR, p_decParams);
19      memcpy(p_out ,p_buff0, 8448);
20      return numIter;
21  }
```

In the process of optimization, an unexpected issue that regards unsuccessful array partitioning performed by Vivado HLS 2018.2. After discussion with the supervisor, it was probably due to the unsupported feature of the HLS tool of the old version. Hence the Vivado HLS 2018.2 was dropped, and instead, the newest version of the HLS tool provided by Xilinx was adopted, namely Vitis 2020.2 on which the array partitioning can be completed successfully.

As introduced in Chapter 3.4, in favour of performing grouped LLR calculation in parallel, intermediate LLRs stored in buffers are mapped in a specific way. From the perspective of memory transferring functions, since their scope is re-mapping data in a certain aligned way which is predefined by the configuration of the decoder. The loop bound is determined to be variable, and this feature is considered as a disadvantage in adopting parallelism in execution. Besides, the long synthesis time caused by a huge amount of LUT where stores corresponding parameters to be fed into function blocks has to be considered into consideration. The first synthesis took around 4 days to finish due to the huge amount of supported configurations. The bottleneck in terms of time consumption in waiting for every synthesis has to be broken through. In this respect, a set of experiments of discovering the relationship between a number of supported parameters, whilst the other two parameters remains fixed, and the corresponding synthesis elapsed time and hardware utilization. Table 5.2 reports the elapsed synthesis time for BG2, CodeRate = 1/3, different numbers of supported expansion factor Z, neglecting the performance of various configurations.

Elapsed synthesis time and usage of FF and LUT are increased with the increasing number of supported Z. The same feature is obtained from experiments of fixed Z, fixed BG, various R and fixed Z, fixed R, various BG. In conclusion, for performing effective experiments of hardware acceleration as many as possible within a certain period of time, preferably the configuration is limited to a specific set of BG, R, Z that helps reducing the synthesis time and determining the maximum latency. In the following chapters, this approach is adopted and further discussed in favour of the possible generalization of the LDPC decoder.

# OF Z	1	2	5	10	15	20	30	40	51
BRAM_18K	379	379	379	379	379	379	379	379	379
DSP45E	58	58	58	58	58	58	58	58	58
FF	44830	45990	46804	47942	49540	50782	55516	58902	61534
LUT	118573	130525	163276	214261	267278	320099	423881	526609	651015
Elapsed time(s)	660	818	1102	1690	1977	2210	3605	4995	6460

Table 5.2: Synthesis time for different number of supported expansion factor Z

5.1 Optimization on memory transferring

5.1.1 Array partitioning

There are two kinds of RAM resources in FPGA, namely Block RAM (BRAM) and Distributed RAM (DRAM). DRAM is synthesized by synthesis tools and realized through multi-level cascade LUT that are far apart. Hence it is given the name Distributed RAM. DRAM makes use of abundant and flexible LUT resources and can be flexibly configured according to usage conditions. It is suitable for occasions where RAM latency is not high. After all, it is not 'real' RAM. In fact, it is also real RAM physically, but not dedicatedly.

Then Block RAM is more 'professional' compared to DRAM. BRAM is a dedicated RAM block resource added to FPGA by manufacturers in addition to logic resources. Compared with DRAM, the RAM block and logic resources have been specially placed and routed so that BRAM has a high operating speed, a certain low latency period, but a limited number of resources.

The limitation of BRAM is the number of ports. Generally speaking, there are two types: single port and dual port. In a design that requires high throughput, the limit on the number of ports may become a bottleneck because data can only be read or write once or twice in one clock cycle. The solution is to divide the single BRAM resource into multiple BRAM, which increases the total number of ports effectively. Vivado HLS provides three types of array partitioning:

1. Block: The same number of consecutive elements of the original array are

stored in a specified number of BRAM

2. Cyclic: The same number of interleaving elements of the original array are stored in a specified number of BRAM

3. Complete: Elements of the original array is split individually, then stored in registers

As shown in Figure 5.1, block array partitioning is suitable for operations on an array that regularly read or write data with a specific interval in between. Cyclic array partitioning is suitable for consecutive reading or writing from/to an array since the interleaving structure allows accessing a serial of data stored in BRAMs in parallel. Complete array partition breaks the BRAM storage and stores all elements in the register. It suits arrays of small size due to the limited resources of FF on FPGA.

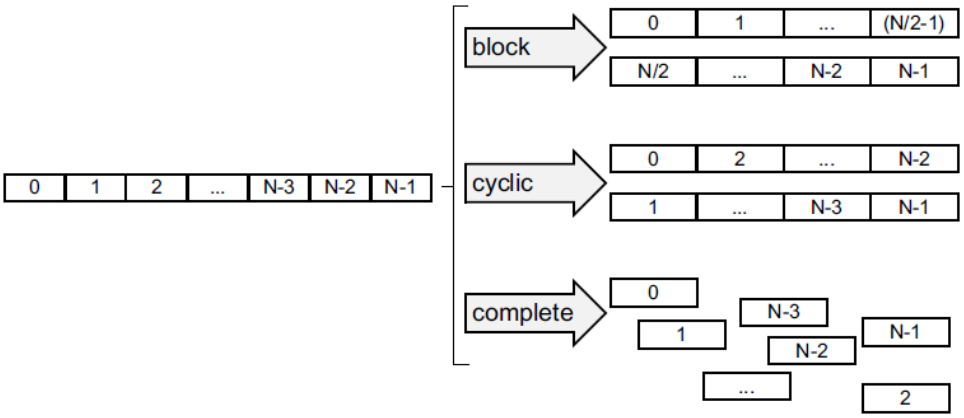

Figure 5.1: Three types of array partitioning

For the current implementation, cyclic array partitioning with factor 32 is chosen since the pseudo-AVX2 instruction performs byte by byte arithmetical and logical computation on a total number of 32 bytes of data. The interleaving structure allows access to each element within one clock cycle which theoretically increases the performance by a factor of 32. As for memory transferring functions, without array partitioning, only one LLR of 8 bits is copied from one buffer to another in one clock cycle, which is less efficient due to the limitation on ports. In theory, 32 cyclic partitioned arrays support 32×8 bits coping in one clock cycle. However, in actual implementation, the performance is not as expected because of the constraints on buffer alignment. More details will be discussed in chapter 5.1.3.

It has been proved that Vivado HLS 2018.2 might encounter some unknown issues that regard array partitioning. It could not successfully partition an array of large sizes, so the project has been suspended for several weeks in order to find out a solution. After discussed with supervisors, Vivado HLS 2018.2 was dropped and the project was continued by using Vitis HLS 2020.2, which is the latest version of the HLS tool also provided by Xilinx. The snippet reported in the following shows the corresponding pragmas of cyclic array partitioning with a factor of 32.

Listing 5.2: Pragma of Array Partition

```
1  static int8_t buf_llrProcBuf[27000];
2  #pragma HLS ARRAY_PARTITION variable=buf_llrProcBuf
3                                          dim=1 factor=3 cyclic
4  static int8_t buf_cnProcBuf[121344];
5  #pragma HLS ARRAY_PARTITION variable=buf_cnProcBuf
6                                          dim=1 factor=32 cyclic
7  static int8_t buf_cnProcBufRes[121344];
8  #pragma HLS ARRAY_PARTITION variable=buf_cnProcBufRes
9                                          dim=1 factor=32 cyclic
10 static int8_t buf_llrRes[27000];
11 #pragma HLS ARRAY_PARTITION variable=buf_llrRes
12                                         dim=1 factor=32 cyclic
13 static int8_t buf_bnProcBuf[121344];
14 #pragma HLS ARRAY_PARTITION variable=buf_bnProcBuf
15                                         dim=1 factor=32 cyclic
16 static int8_t buf_bnProcBufRes[121344];
17 #pragma HLS ARRAY_PARTITION variable=buf_bnProcBufRes
18                                         dim=1 factor=32 cyclic
```

5.1.2 Fixing loop bounds

From the synthesis report, we can check the performance of each function and the whole project. If the loop bound is variable, the synthesizer is not able to determine how many clock cycles are required to finish. Thus, Vivado HLS reports the latency as a question mark(?) instead of using exact values. The primary goal is to determine the exact number of iteration by means of adding tripcount directives or modifying the loop structure.

Firstly the tripcount directives are added. Taken the code section reported in Listing 4.10 as an example, for simplicity of reading, the function nrLDPC_circ_memcpy is expressed explicitly in the code section where it is called. A minimum and maximum loop bound is specified on the loop so that the synthesizer reports both minimum and maximum latency bound depending on the specified loop bound

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
?	?	?	?	none

Figure 5.2: Incomplete performance report due to variable loop bounds.

range. Moreover, tripcount has an impact only on reporting, without any effect on the hardware implementation.

Listing 5.3: Tripcount directives for determine the bounds of loop

```
    bitOffsetInGroup = lut_numCnInCnGroups_BG1_R13[4]*NR_LDPC_ZMAX;
loop_bn2cnProcBuf_BG2_8:for (j=0; j<6; j++)
    {
  loop_bn2cnProcBuf_BG2_9:for (i=0; i<lut_numCnInCnGroups[4]; i++)
        {
#pragma HLS LOOP_TRIPCOUNT min=5 max=5
            idxBn = lut_startAddrBnProcBuf_CNG7[j][i]+
                    lut_bnPosBnProcBuf_CNG7[j][i]*Z;
loop_circ_memcpy_27:for(k=0; k<Z-circShift_BG1_Z384_CNG7[j][i]; k++)
            {
                #pragma HLS LOOP_TRIPCOUNT min=1 max=384
                buf_cnProcBuf[lut_startAddrCnGroups_BG1[4]+
                              j*bitOffsetInGroup+Z*i+k] =
                              buf_bnProcBufRes[idxBn+k+
                              circShift_BG1_Z384_CNG7[j][i]];
            }
loop_circ_memcpy_28:for(k=0; k<circShift_BG1_Z384_CNG7[j][i]; k++)
            {
                #pragma HLS LOOP_TRIPCOUNT min=0 max=383
                buf_cnProcBuf[lut_startAddrCnGroups_BG1[4]+
                              j*bitOffsetInGroup+Z*i+k+Z
                              -circShift_BG1_Z384_CNG7] =
                              buf_bnProcBufRes[idxBn+k];
            }
        }
    }
```

Tripcount directives are specified on all loops of memory transferring functions. Then the synthesis is run without neither array partitioning nor pipelining. The performance report determines the minimum and maximum latency, but it has a significant difference that is not supposed to exist. Through an analysis of the code section reported in Listing 5.3, two loops are executed inside the nested outmost loop, the loop bound of the first loop is Z - circShift_BG1_Z384_CNG7[j][i] whilst the loop bound of the second loop is circShift_BG1_Z384_CNG7[j][i]. The total

number of iteration is supposed to be Z no matter what the value of the circle shift index is because these two loops are paralleled instead of nested. The loop bounds determined by tripcount directive misleads the synthesizer from analyzing the correlation between the paralleled loops. Besides, the reported latency of functions nrLDPC_llrRes2llrOut and nrLDPC_llr2llrProcBuf are correct since the method of memory copying is different from the other three functions where nrLDPC_circ_memcpy and nrLDPC_inv_circ_memcpy are called. In conclusion, tripcount directive is not well suitable for the determination of actual latency.

Funtions	Minimum Latency	Maximum Latency
nrLDPC_cn2bnProcBuf_BG1	6659	240193
nrLDPC_llr2cnProcBuf_BG1	8642	238216
nrLDPC_bn2cnProcBuf_BG1	6075	207947
nrLDPC_llrRes2llrOut	26117	26117
nrLDPC_llr2llrProcBuf	26117	26117

Table 5.3: Incorrect performance report of memory transferring functions

The remaining solution is to modify the loop structure in order that the loop bound became fixed. The paralleled loops can be rewritten to be a single loop in which an if-else statement controls conditional execution. The loop bound is explicitly set to the maximum number of iteration, which is the fixed number of total iteration depending on the expansion factor Z. Inside the if statement, one part of memory copying is executed and inside the else statement, in order to keep unchanged the functionality, the address offset is shifted back by circShift_BG1_Z384_CNG7[j][i] since the iteration number is constantly increasing.

Listing 5.4: Modified function body that has fixed loop bound

```
for(k=0;k<384;k++)
{
    if(k<circShift_BG1_Z384_CNG7[j][i])
    {
        buf_cnProcBuf[lut_startAddrCnGroups_BG1[4]
                      +j*bitOffsetInGroup+Z*i+k+
                      Z-circShift_BG1_Z384_CNG7[j][i]]=
                      buf_bnProcBufRes[idxBn+k];
    }
    else
    {
```

```
12            buf_cnProcBuf[lut_startAddrCnGroups_BG1[4]
13                          +j*bitOffsetInGroup+Z*i+k-
14                          circShift_BG1_Z384_CNG7[j][i]]=
15                          buf_bnProcBufRes[idxBn+k];
16        }
17    }
```

Data of size 121344×8 bits are copied back and forth between CN buffers and BN buffers, the same size data is written from LLR buffer to CN buffer, it could be deduced that at least 242688 clock cycles are required for each function call in case no optimization method was adopted.

5.1.3 Buffer alignments

By adopting the pipeline, the reading and writing operation can be overlappingly performed from the second clock cycle so that an acceleration factor of 2 is achieved. Array partitioning provides the possibility of copying 32 bytes of data in parallel. In theory, it should have an additional acceleration factor of 32. However, in the actual case, it is not able to perform the expected synthesis. Even though the array has 32 ports thanks to cyclic partitioning, it does 32 times read of 8 bits in parallel, and one multiplexer is used to select one source data to be written to the destination port instead of writing each read data in their corresponding destination. This issue occurs due to the unsatisfied alignment in either the source buffer or destination buffer. In order to perform reading/writing in parallel, it has to be assured that the buffers have been aligned by the same factor of how memories are cyclically partitioned. In some cases, the tool is not able to determine if the reading ports or writing ports are aligned by the required factor. Thus a little trick of coding is applied to clarify the demand to HLS.

To perform the alignment test, initially, the cycle memory copy and inverted cycle memory copy functions are rewritten, as shown in listing 5.5.

Listing 5.5: Implementation of aligned memory copy functions

```
1  static void nrLDPC_inv_circ_memcpy(int8_t a[]. int8_t b[], uint16_t Z
       , uint16_t shift)
2  {
3      uint16_t k;
4      int8_t assert;
5      loop_func_1:for(k=0;k<384;k++)
6      {
7          assert=(k<shift)?1:0;
8          a[k]=b[Z*assert+k];
```

```
9       }
10  }
11  static void nrLDPC_circ_memcpy(int8_t a[], int8_t b[], uint16_t Z,
        uint16_t shift)
12  {
13      uint16_t k;
14      int8_t assert;
15      loop_func_2:for(k=0;k<384;k++)
16      {
17          assert=(k<shift)?1:0;
18          a[Z*assert+k]=b[k];
19      }
20  }
```

If two arrays are both cyclically partitioned by a factor of 32, there are at least 32 BRAM in which several sets of 32 bytes data are stored separately. The constraint of unrolling is the alignment of sequential data to be accessed, namely inside the loop body of either nrLDPC_inv_circ_memcpy or nrLDPC_inv_circ_memcpy reported in listing 5.5, the starting addresses of source buffer and destination buffer are supposed to be aligned to 32, i.e. 0, 32, 64, etc. The expansion factor is 384, which is a multiple of 32. Therefore no matter what value is assigned to the variable 'assert' port alignment is always satisfied. Another advantage of the coding style is the fixed loop bound which enables the loop unrolling to be performed. Considering the 256 bits alignment structure of AVX2 implementation, the loop unrolling directive with factor 32 is applied on loop_func_1 and loop_func_2.

An additional address offset is passed to the memory copy functions in upper-layer functions, which breaks the alignment on either destination buffer or source buffer. According to the pseudo-code reported in listing 5.6, three cases are presented:

Listing 5.6: Generic case of how memcpy fuctions are called

```
1  for(i=0; i<6; i++)
2      {
3        for(j=0; j<8;j++)
4          {
5              Memcpy(buff_a[idx0],buff_b[idx1],parameter_0,parameter_1);
6          }
7      }
```

1. Ideal case: Both idx0 (destination) and idx1 (source) are aligned by a factor of 32
2. Real case: idx1 (source) is aligned by a factor of 32, whilst idx0 (destination) is NOT aligned by a factor of 32

3. Worst case: idx1(source) is NOT aligned by a factor of 32, whilst idx0 (destination) is aligned by a factor of 32

Unfortunately, the ideal case never meets in the current implementation since the cycle shift factor sets a breakpoint at an arbitrary position in the memory section of length Z×8 bits, the generic cyclic memory copy functions provides the universal structure but offset passed to it gives a negative impact on the alignment and naturally harms the performance.

Module nrLDPC_llr2CnProcBuf meets the "real case", and it is called only once during the execution of the LDPC decoder since it passes the input LLR into the CN processing buffer without further operation. The module nrLDPC_bn2cnProcBuf also meets the "real case", but it is called for the times equal to the number of iteration, namely at the end of each iteration, the LLR results computed in BN processing are copied back in CN processing buffer for parity check. A modification is necessary because of the inability to determine alignment from the perspective of the synthesizer. Hence, the offset is shifted right for 5 bits and shifted back left for 5 bits. This trick guarantees the alignment in the mode that could be interpreted by the HLS tool. The method introduced above is reported in the following:

Listing 5.7: Case of calling circle memory copy function

```
1  bitOffsetInGroup = lut_numCnInCnGroups_BG1_R13[3]*NR_LDPC_ZMAX;
2  loop_cn2bnProcBuf_BG1_13:for (j=0; j<6; j++)
3      {
4  loop_cn2bnProcBuf_BG1_14:for (i=0; i<lut_numCnInCnGroups[3]; i++)
5          {
6  #pragma HLS LOOP_TRIPCOUNT min=8 max=8
7              idxBn = startAddrBnProcBuf_CNG6[j][i] +
8                      bnPosBnProcBuf_CNG6[j][i]*Z;
9              shift = lut_startAddrCnGroups_BG1[3] +
10                     j*bitOffsetInGroup + Z*i -
11                     circShift_BG1_Z384_CNG6[j][i];
12             nrLDPC_circ_memcpy(&buf_bnProcBuf[shift],
13             &buf_cnProcBufRes[idxBn],Z,lut_circShift_CNG6[j][i]);
14         }
15     }
```

Circle shift index impacts the alignment of source buffer then the performance meets the worst case, namely loop unrolling is not applied as expected. The worst case fits the module nrLDPC_cn2bnProcBuf where the inverted circle memory copy occurs. In this case, the source code is supposed to be modified, conforming to the source buffer alignment requirement as much as possible. A sub-function for storing transitory aligned data is proposed:

Listing 5.8: Sub-function for buffer ports alignment

```
static void mem_pass(int8_t a[], int8_t b[])
{
#pragma HLS INLINE
    uint16_t k;
    trans_1: for(k=0;k<768;k++)
    {
#pragma HLS PIPELINE II=2
#pragma HLS UNROLL factor=32
        a[k] = b[k];
    }
}
```

The basic idea is passing the value to a transitory buffer whose starting address is 0, then inverted cyclically copying data from the transitory buffer to the destination buffer whose starting address is aligned. Finally, the ideal case meets at the second phase. As for the first phase, redundant bits of size Z is passed to the transitory buffer since the algorithm of the inverted circle memory passing function takes the aligned starting address of the source buffer as input but copies the same size of data with a certain shift index cshift as the actual starting address. In the example shown in Figure 5.3, at the second phase of memory transferring, data of size cshift×8 bits are copied consecutively from address Z of the transitory buffer to the initial address of the destination buffer, then data of size (Z-cshift)×8 bits are transferred sequentially from address cshift of transitory buffer to the same address of the destination buffer, where is completely aligned.

In order to efficiently copy data from the source buffer to the transitory buffer, constraints on HLS optimization have to be taken into consideration. The extra copying is not able to be accelerated in case the starting address of the source buffer is not aligned. Hence the copying operation is divided into two phases. During the first one, non-aligned data are copied to the starting address of the transitory buffer, and no optimization method could be applied except pipeline. The size of the data is the subtraction of 32 and the remainder of starting address of the source buffer divided by 32. Then the consecutive 2Z×8 bits data is aligned since the starting address is already modified to the nearest 32-aligned value. For the second phase, the proposed sub-function reported in listing 5.8 is called. In addition, the HLS optimization directive is well applied because it perfectly meets the "real case" where the source buffer's address is aligned whilst the destination buffer's address does not. An unroll factor of 32 and pipeline is applied. The sub-function is also inlined to achieve the best performance.

Listing 5.9: The case of calling inverted circle memory copy function

```
bitOffsetInGroup = lut_numCnInCnGroups_BG1_R13[3]*NR_LDPC_ZMAX;
```

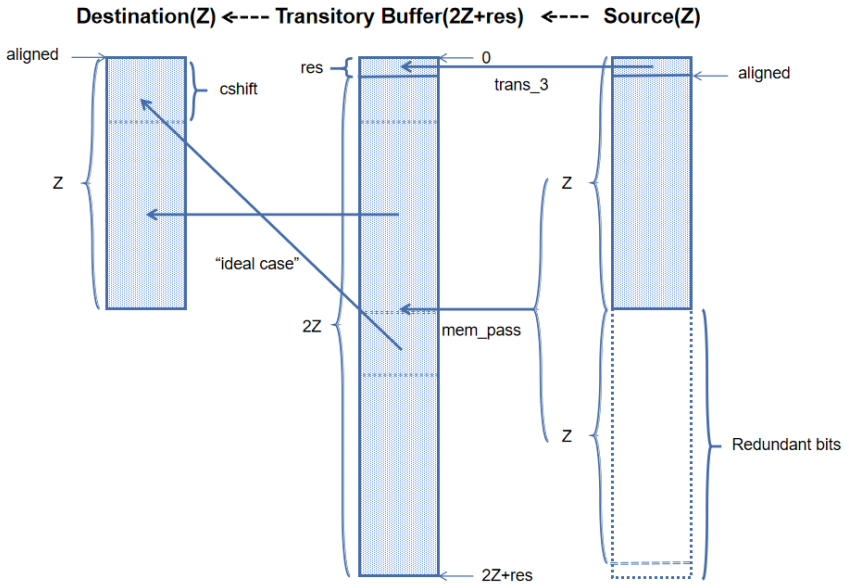

Figure 5.3: Transitory buffer ensures port alignment

```
loop_cn2bnProcBuf_BG1_9:for (j=0; j<6; j++)
    {
loop_cn2bnProcBuf_BG1_10:for (i=0; i<lut_numCnInCnGroups[3]; i++)
        {
#pragma HLS LOOP_TRIPCOUNT min=0 max=8
            idxBn = startAddrBnProcBuf_CNG6[j][i] +
            bnPosBnProcBuf_CNG6[j][i]*Z;
            shift = lut_startAddrCnGroups_BG1[3] +j*bitOffsetInGroup+
            Z*i - circShift_BG1_Z384_CNG6[j][i];
            shift_res = 32 - shift % 32;
            shift_new = shift >> 5;
            shift_new = shift_new << 5;
            shift_new = shift_new + 32;
            idxBn = idxBn >> 5;
            idxBn = idxBn << 5;
trans_3 for(s=0; s<32; s++)
            {
#pragma HLS PIPELINE
              if(s < shift_res) {
                    buff_tran[s] = buf_bnProcBufRes[shift + s];
                                }
            }
              mem_pass(&buff_tran[shift_res],
              &buf_bnProcBufRes[shift_new]);
              nrLDPC_inv_circ_memcpy(&buf_bnProcBuf[idxBn],
              &buf_tran[0],Z,lut_circShift_CNG6[j][i]);
```

```
28            }
29        }
```

5.1.4 Results from High-Level Synthesis report

In the last chapter, three cases of memory transferring function and their corresponding code modification are reported. Even though the ideal case never meets in stand-alone circle memory copy modules, in the proposed solution of worst case, the ideal case is achieved alone with certain drops in performance due to the complicated implementation. Taking group 6 of the CN group as an example, the comparison among three cases is listed:

Resources/Cases	Original	Ideal case	Real case	Worst case
BRAM_18K	0	0	0	0
DSP45E	1	1	3	1
FF	101	481	965	1,019
LUT	564	3,648	11,722	16,082
Iteration latency(II)	1	1	2	80
Trip count	18432	576	576	48
Overall latency	18434	579	1,155	3,840

Table 5.4: Performance and hardware utilization reports of three cases(CN Group 6)

The latency of the real case is 16 times less than the original implementation, whilst the utilization of FF is approximately increased by a factor of 10, and the utilization of LUT is roughly increased by a factor of 20. As for the worst case, after performing the modification of implementation, the performance increases by a rough factor of 5. However, the utilization of FF and LUT is increased approximately by a factor of 10 and 28, respectively. The initiation latency(II) of the real case is optimally dropped to 2 since the calculation of the new address takes one additional clock cycle. The II value of the worst case is higher than expected under the condition that the outermost loops cannot be pipelined. The final result can be considered an unroll factor of 5 applied on the inverted circle memory copy module without pipelining, which is the bottleneck of both performance and resource among all memory transferring functions.

Functions	Original Latency	Optimized Latency	Acc. Factor	BRAM	DSP45E	FF	LUT
Cn2BnProcBuf	240,193	32,388	741%	0	1	1995 -> 9299	8167 -> 105636
llr2CnProcBuf	238,216	7,991	2981%	0	1 -> 0	1752 -> 5203	6476 -> 94072
Bn2CnProcBuf	207,947	6,968	2984%	0	1 -> 0	1871 -> 5450	6768 -> 89766
llr2llrProcBuf	26,117	1,446	1806%	0	0	95 -> 757	494 -> 13229
llrRes2llrOut	26,117	1,446	1806%	0	0	123 -> 457	508 -> 13467

Table 5.5: Original functions vs. Optimized functions

Table 5.5 reports the initial and optimized performance and resource utilization of memory transferring functions. We can see that the performance is dramatically optimized, with an acceleration factor of approximately $30\times$ for module llr2CnProcBuf and Bn2CnProcBuf, the acceleration factor of $18\times$ for module llr2llrProcBuf and llrRes2llrOut since their implementation is independent of the other three modules. However, the module Cn2BnProcBuf remains the bottleneck even after acceleration because of the unalignment on the source buffer. It eventually obtains $7\times$ better performance and a dramatic increase in resources.

5.2 Parallelism of Computation functions

The optimization of computation functions is straightforward since the buffers are already cyclically partitioned with a factor of 32, which has the exact alignment with pseudo AVX2 instructions. Keeping the current implementation of LLR computation, an efficient hardware implementation can be obtained through loop unrolling and pipelining.

The reason for which the current configuration(BG1, R = 1/3, $Z = 384$) is selected is that the quantity of computation is the maximum among all supported configurations, the upper bound of performance is obtained by adopting such configuration. Then the LDPC decoder module can be generalized to support more configurations without affecting the hardware implementation of the core computational sub-module.

5.2.1 Loop unrolling

The Loop unrolling technique is widely adopted in the memory transferring function, where a fixed number Z or 2Z of loop bound is defined. The unrolling factor has to be conformed with the number of available ports on the operated buffer. Otherwise, the loop cannot be flattened for generating indicated number of copies. The same partitioning factor 32 suits perfectly the optimization of computation functions since their core computational unit performs parallel operations on aligned data of 256 bits. The maximum loop bound among all pseudo AVX2 functions is 32. Hence the full unrolling on the loops inside these sub-functions can be performed. In the following snippet of code, an example of unrolling directive's usage is shown.

Listing 5.10: Loop unrolling applied on pseudo AVX2 functions

```
1  m256i mm256_min_epu8(m256i a, m256i b){
2  #pragma HLS INLINE
3       m256i dest =
       {{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
4       0,0,0}};
5       int i;
6  loop_intrin_min:  for(i=0; i<32; i++)
7      {
8  #pragma HLS UNROLL
9           if((uint8_t)a.data[i] >(uint8_t)b.data[i])
10          dest.data[i] = b.data[i];
11          else
12          dest.data[i] = a.data[i];
13      }
14      return dest;
15 }
```

The structured array a, b and the intermediate variable dest are automatically partitioned by the HLS tool. Their elements are respectively divided into 32 individual data stored in registers. Therefore there is no port limitation. The loop can be completely unrolled, generating 32 copies and finishing the execution within one clock cycle, whilst the same factor of unrolling expands the resources utilization.

5.2.2 Loop pipelining

The implementation of computation functions is through several nested loops that iterate through either CN nodes or BN nodes and successively their neighbourhood nodes. The HLS tool provides loop pipelining, which allows the next iteration to be started before the previous iteration finished. In order to perform

pipeline on the outermost loop, its' inner loops are flattened automatically, namely they have to be completely unrolled. Considering the code example of the cnProc module shown in listing 5.10, in Chapter 3, the variable which stores the offset of the result buffer is already substituted into the innermost loop. Therefore loop_cnProcBG1_8 and loop_cnProcBG1_9 are perfectly nested. HLS tool takes advantage of this code structure because the additional clock cycle to enter the loop_cnProcBG1_9 is skipped so that the two outermost loops are bind together in case loop_cnProcBG1_9 is pipelined, the loop bound of the new outermost loop becomes 5×M. For an imperfect nested loop, which is loop_cnProcBG1_9 and loop_cnProcBG1_10 due to the existed three directives in between, when pipelining the loop_cnProcBG1_10, additional cycles are taken to enter and exit the loop, which increases the latency and decline the overall throughput. Since the pseudo AVX2 instructions are completely unrolled, generating more copies of these instructions increases the utilization of resources but improves the throughput effectively. Pipelining is supposed to be applied on loop_cnProcBG1_9 for the best performance because pipelines loop_cnProcBG1_10 results in an unsuccessful pipeline of the outermost loop. There are generated four copies of function mm256_conv_int8 and mm256_min_epu8, five copies of mm256_sign_epi8 in the flattened loop loop_cnProcBG1_9. The dependence directive helps determine the data dependencies among operations from different iterations and generate the correct logic control unit in hardware to avoid dependency violation.

Additionally, the loop_cnProcBG1_11 is automatically unrolled even without the HLS directive because the loop is parallel to loop_cnProcBG1_10, which is the inner loop of the pipelined loop. According to the alignment rule, the memory copying operation is considered an ideal case since the starting address of both source buffer and destination buffer is aligned to 32.

Listing 5.11: Optimization example of cnProc module

```
1  loop_cnProcBG1_8:for (j=0; j<5; j++)
2            {
3  loop_cnProcBG1_9:for (i=0; i<M; i++)
4                 {
5  #pragma HLS PIPELINE II=4
6  #pragma HLS LOOP_TRIPCOUNT min=0 max=216
7                       ymm0 = mm256_conv_int8(&buf_cnProcBuf
8                       [lut_startAddrCnGroups_BG1[2] +
9                       lut_idxCnProcG5[j][0]*32 + i*32]);
10                      sgn  = mm256_sign_epi8(ones256_epi8, ymm0);
11                      min  = mm256_abs_epi8(ymm0);
12 loop_cnProcBG1_10:for (k=1; k<4; k++)
13 #pragma HLS DEPENDENCE true inter
14                      {
15                           ymm0 = mm256_conv_int8(&buf_cnProcBuf
```

```
16                          [lut_startAddrCnGroups_BG1[2] +
17                          lut_idxCnProcG5[j][k]*32 + i*32]);
18                          min  = mm256_min_epu8(min, mm256_abs_epi8(ymm0));
19                          sgn  = mm256_sign_epi8(sgn, ymm0);
20                      }
21                      min = mm256_min_epu8(min, maxLLR256_epi8);
22                      tmp = mm256_sign_epi8(min, sgn);
23  loop_cnProcBG1_11: for(l=0; l<32; l++)
24  #pragma HLS UNROLL
25                      {
26                          buf_cnProcBufRes[lut_startAddrCnGroups_BG1[2] +
27                          (j*bitOffsetInGroup*32) + i*32+l] = tmp.data[l];
28                      }
29                  }
30              }
31          }
```

Functions can be inlined in favour of merge the logic with the logic of the surrounding functions, generally inlining small functions may optimize the performance and the area. Table 5.6 reports the improvement of performance while inlining the pseudo AVX2 instructions.

Model: cnProc	Inline-OFF	Inline
Loop: 1-3	II = 1, Latency: 38	II = 1, Latency: 37
Loop: 4-5	II = 3, Latency: 721	II = 3, Latency: 720
Loop: 8-9	II = 4, Latency: 4321	II = 4, Latency: 4320
Loop: 12-13	II = 4, Latency: 2881	II = 5, Latency: 2880
Loop: 16-17	II = 6, Latency: 2521	II = 5, Latency: 2100
Loop: 20-21	II = 7, Latency: 1345	II = 6, Latency: 1152
Loop: 24-25	II = 8, Latency: 1729	II = 6, Latency: 1296
Loop: 28-29	II = 9, Latency: 1081	II = 7, Latency: 840
Loop: 32-33	II = 18, Latency: 16417	II = 11, Latency: 10032

Table 5.6: Impacts of inlining functions

In summary, computation functions share a similar code structure which is nested

loops with pseudo AVX2 instructions inside. The critical work is to change the outermost loop into a perfect loop. Modification on source code is relatively more minor than on memory copying functions. Directives provided by HLS assures the efficient optimization of hardware implementation without massive modification on software-based code. While considering the generalized computational module, the loop bound of the second outermost loop becomes variable since the M is determined by the code rate and expansion factor of the decoder. The HLS performance report provides the maximum latency among all available configurations. Accordingly, the loop bound can be set to the maximum value and use the if statement to execute the loop body for indicated times conditionally. This proposed solution results in additional hardware resources for storing configuration parameters and performing the selection. However, the performance of pipeline will not be affected.

5.3 Results summary

So far the optimization method has been presented, starting from the array partitioning and alignment of starting addresses which assure in maximum the parallelization of memory copying to the loop unrolling and pipelining for computational modules. For the current implementation, the HLS tool cannot optimize more through directives, and further optimization is possible to be obtained by modifying the algorithm or generalizing the sub-functions. The result of final performance and resources utilization is reported in the following table.

The optimized module obtains an acceleration factor of 24 $\times$ in terms of latency, precisely the most significant value of acceleration factor belongs to module cnProc whose performance is increase by around 55 times, even though it remains to be the critical module among all computation functions. To be noticed that the initial report was not based on a module without any HLS optimization directives. The Vitis HLS 2020.2 performs automatically specific optimizations by default, i.e. pipelining or array partitioning for buffers with small sizes. As for the resource usage, the BRAM doubles because of array partitioning; buffers with relatively small sizes are partitioned by the same factor of 32 in order to achieve the best performance. DSP is rarely used because most of the computation is performed on integer, which is done through LUT instead of DSP that dedicates more to floating point computation. Usage of LUT is almost quadruple the initial one due to the loop unrolling. Copies of basic sub-functions occupy additional space on FPGA and result in better performance. The most critical function is bn2cnProcBuf which copies data from buffer bnProcBufRes to buffer cnProcBuf. The usage of FF and LUT is increased roughly by a factor of 5 and 12, respectively, whilst the latency

obtains an acceleration factor of 7 × only. The current optimization method for solving buffer transferring with an unaligned starting address of the source buffer may not be optimum. Extra resources are wasted in plenty of redundant copies of inlined sub-functions that can be allocated only one copy in the Register Transfer Level (RTL) instance since the ports of arrays are limited, and the functions are grouped by the number of BN connected to CN nodes.

Fun. block/Area	BRAM	DSP	FF	LUT	Initial Latency	Final Latency	Acc. factor
CnProc	1 → 0	0→ 1	8063→ 6410	39044→ 72639	1,315,909	23,395	5600%
CnProcPc	0	1	2021→ 1332	15560→ 8448	533 ~ 59,575	37 ~ 3,751	1588%
BnProc	0	0	1631→ 404	15970→ 31867	52,824	3,312	1594%
BnProcPc	0	0	9492→ 9299	24868→ 64731	90,821	4,362	2082%
Cn2BnProcBuf	0	1	1995→ 9299	816→ 105636	240,193	32,388	741%
llr2CnProcBuf	0	1→ 0	1752→ 5203	6476→ 94072	238,216	7,991	2981%
Bn2CnProcBuf	0	1→ 0	1871→ 5450	6768→ 89766	207,947	6,968	2984%
llr2llrProcBuf	0	0	95→ 757	494 → 13229	26,117	1,446	1806%
llrRes2llrOut	0	0	123 → 457	508→ 13467	26,117	1,446	1806%
llr2bit	0	0	405→ 303	2021→ 1681	13,076	821	1592%
llr2bitPacked	0	0	500→ 454	1772→ 3818	13,064	3268	399%
Total	157→ 388	4→ 3	29791→ 38711	128369→ 537476	10,123,394	424,379	2385%

Table 5.7: Comparison of performance and resources utilization between initial module and optimized module

Hardware	Execution time[ms]
CUDA Solution 1	38550
CUDA Solution 2	98.849
CUDA Solution 3	41.152
AVX2 Solution	1.701

Table 5.8: Comparison of Execution time between proposed solutions

In addition, the comparison between the current solution and the previous solution, which focused on another implementation of LDPC decoder written in CUDA code,

the performance is $24\times$ times better than the final solution of CUDA code.

Chapter 6

Future work

Despite the fact that the current result is better than the previous ones, the synthesis report of Vitis HLS is only an estimation of actual implementation on FPGA. Due to the extremely long time of synthesis after all optimization directives are applied on the source code, around one day of time is elapsed to finish the synthesis. The further optimization is terminated because of the time-consuming issue. However, the current result reveals that the AVX2 implementation of the OAI LDPC decoder is a suitable starting module for FPGA. The next step will be looking through the whole module and functionalize the repetitive operations of either computational functions or memory transferring functions. The HLS optimization pragma ALLOCATION restricts the number of RTL models of the called function, containing the same structure for all CN/BN groups. These operations MUST be performed in sequence since the port number is limited. A more prominent factor of array partitioning will not be considered because resource utilization becomes the bottleneck. A reasonable trade-off is obtained by unifying the function and reducing the resources until it fits in one Super Logic Region(SLR) of the current selected board ALVEO u280 on which three SLRs are available.

From the report shown above, the utilization of LUT exceeds the capacity on one SLR, which means that the RTL model occupies only two available SLR while wasting the third one. If the usage of LUT could be reduced to less than 434,568, the board will be capable of executing three decoder modules in parallel, which obviously will increase the throughput by a factor of 3. Moreover, the current implementation has the code structure on the top-level function, as shown in Figure 6.2.

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+---------------------+---------+------+---------+---------+-----+
|         Name        | BRAM_18K|  DSP |    FF   |   LUT   | URAM|
+---------------------+---------+------+---------+---------+-----+
|DSP                  |        -|     -|        -|        -|    -|
|Expression           |        -|     -|        0|      120|    -|
|FIFO                 |        -|     -|        -|        -|    -|
|Instance             |      324|     3|    52741|   542895|    -|
|Memory               |       64|     -|        0|        0|    -|
|Multiplexer          |        -|     -|        -|     3147|    -|
|Register             |        -|     -|      268|        -|    -|
+---------------------+---------+------+---------+---------+-----+
|Total                |      388|     3|    53009|   546162|    0|
+---------------------+---------+------+---------+---------+-----+
|Available SLR        |     1344|  3008|   869120|   434560|  320|
+---------------------+---------+------+---------+---------+-----+
|Utilization SLR (%)  |       28|    ~0|        6|      125|    0|
+---------------------+---------+------+---------+---------+-----+
|Available            |     4032|  9024|  2607360|  1303680|  960|
+---------------------+---------+------+---------+---------+-----+
|Utilization (%)      |        9|    ~0|        2|       41|    0|
+---------------------+---------+------+---------+---------+-----+
```

Figure 6.1: HLS report of the final implementation

Potential DATAFLOW could be applied to the top-level function. However, several requirements should be satisfied in order to perform a successful dataflow optimization.

1. Dataflow should be inside a loop body, and the loop condition is a positive numerical constant or constant function argument
2. Latency of each sub-function should be approximate without a huge difference
3. Loops without neither multiple exit conditions nor feedback between tasks

The stopping criteria adopted by the current implementation prevents the dataflow from performing since the parity check result is checked after each iteration for deciding if the following iteration is necessary. The removal of stopping criteria sets the fixed iteration bound for the decoder, but in the case of a channel with high SNR, the throughput will be decreased since the needed iteration number is less than the critical case. However, it is possible to perform further analysis of the current algorithm to see the possibility of adopting dataflow optimization.

In recent years, an AVX-512 based software decoding for LDPC codes has been proposed [19], which utilizes 512 bits aligned structure for iterative LLR computation

```
function_0(input A[26112], output B[121344]); // memory copy

function_1 (input B[121344], output C[121344]);  //calculation

function_2(input C[121344], output D[121344]); // memory copy

function_3(input D[121344], output E[27000]);  //calculation

function_4(input D[121344], input E[27000], output F[121344]);  //calculation

function_5(input F[121344], output B[121344]); // memory copy

   while ( (i < (numMaxIter-1)) && (pcRes != 0) )  // iterate for 3 times
       {
            i++;
        function_1(input B[121344], output C[121344]);  //calculation

        function_2(input C[121344], output D[121344]); // memory copy

        function_3(input D[121344], output E[27000]);  //calculation

        function_4(input D[121344], input E[27000], output F[121344]);  //calculation

        function_5(input F[121344], output B[121344]); // memory copy

pcRes = function_6(input B[121344], input C[121344]);  //parity check

       }

    if ( (i < numMaxIter) && (pcRes != 0) )  // last iteration
       {
            i++;
        function_1(input B[121344], output C[121344]);  //calculation

        function_2(input C[121344], output D[121344]); // memory copy

        function_3(input D[121344], output E[27000]);  //calculation

        function_4(input D[121344], input E[27000], output F[121344]);  //calculation

        function_5(input F[121344], output B[121344]); // memory copy

pcRes = function_6(input B[121344], input C[121344]);  //parity check
```

Figure 6.2: Structure of top-level function

and data transferring between buffers. Considering the AVX2 performance on FPGA, it can be deduced that AVX-512 is the potential to be a better solution since the parallelization is further applied.

Chapter 7

Conclusion

This work adopts the AVX2 solution of quasi-cyclic LDPC decoder for exploring the adaptability on FPGA by utilizing the High-Level Synthesis tool provided by Xilinx, and the main goal is solving the unsynthesizability issue of the source code, which is based on AVX2 instructions for Intel processor, this module does not fit in HLS tool because the dedicated instruction set is not supported. Thus several modifications are applied to the source code to guarantee synthesizability. Specifically speaking, AVX2-based instructions are explicitly rewritten in C while keeping the same functionality. Accordingly, the pointer casting issue is resolved by unifying the single data type of which all processing buffers perform computational operations. Besides, the usage of pointers is effectively reduced since the pointer may result in unknown errors while running synthesis. Source code of sub-functions is massively modified while maintaining the top-level structure.

After the code becomes synthesizable, Vivado HLS 2018.2 is initially chosen to perform high-level synthesis. Due to the inability of partitioning arrays, the applied tool is shifted to Vitis HLS 2020.2 that provides a more powerful synthesis ability. Optimization methods are applied in two groups, namely computation functions' group and memory transferring functions' group, starting from the array partitioning, which is fundamental for exploiting parallelism in either computation modules or memory transferring modules. Then the optimization of the two groups is performed separately. The most critical function in the memory transferring group is the cn2bnProcBuf which utilizes the inverted circle memory copy whose starting address of destination buffer is port-aligned whilst the starting address of source buffer is not port-aligned. HLS tool cannot effectively unroll the loop and exploit the parallelism. The proposed modification decreases the latency by a factor of 5, but it also increases the resources utilization by a factor of approximately 15. Other

memory transferring functions are effectively optimized in terms of performance.

The computation functions share a similar code structure. Therefore, the optimization method applied to them is straightforward. Completely unroll the loops in converted AVX2 instruction enables the second outermost loop in each computational module to be pipelined, and in consequence, the best performance is obtained based on the characteristic of AVX2 instructions that put constraints on the maximum bits (256 bits) of parallelism.

The final performance and resources utilization are obtained from the HLS report. The usage of BRAM is $2\times$ more than the un-optimized module and the usage of LUT is $4\times$ more, while the DSP remains almost the same percentage of utilization and the number of FF is only $1.5\times$ more. The HLS performance report shows an acceleration factor of $24\times$ compared to the original model, and the same $24\times$ could be obtained while comparing to the previous work based on CUDA code. However, due to the long synthesis execution time, the generated RTL model is not exported and place on the real board. The analyzed data on the HLS report is just an estimation of the final performance. In addition, Two references are taken for comparison: the software emulation result of AVX2 solution running at 3.2 GHz, which is 0.257 ms, and the GPU solution is 0.107 ms. The AVX2 solution is still $8\times$ better than the current FPGA implementation since the clock frequency is higher, but in terms of an trade-off between energy consumption and throughput, the FPGA implementation is more energy-efficient compared to the AVX2 solution. This characteristic will be proved in the future work. In addition, the GPU model is the best because of the massive repetitive simple computations on the LLR, which is done better on GPU and relatively worse on CPU. However, the previous work demonstrates that the GPU model does not fit well in the FPGA implementation. In any case, the software implementation of the OAI LDPC decoder is considered to be a good model for FPGA.

Appendix A

Appendix A reports several configurations and software emulation results of NR LDPC code.

A.1 Expansion factors in NR LDPC

According to technical specifications released by 3GPP, the expansion factor Z_c can be obtained by the equation (2.1), the detailed factors are listed in Table A.1

$$Z_c = a \times 2^j \tag{A.1}$$

Indices	Parameters
iLS	0, 1, 2, 3, 4, 5, 6, 7
a	2, 3, 5, 7, 9, 11, 13, 15
J_a	7, 7, 6, 5, 5, 5, 4, 4

Table A.1: Indices of expansion factor

A.2 Zc table of 3GPP standard

The full list of supported configurations of expansion factors are reported in Table A.2, provided by 3GPP [20]

Lift index i_{LS}	Lifting factor Z_C
0	2, 4, 8, 16, 32, 64, 128, 256
1	3, 6, 12, 24, 48, 96, 192, 384
2	5, 10, 20, 40, 80, 160, 320
3	7, 14, 28, 56, 112, 224
4	9, 18, 36, 72, 144, 288
5	11, 22, 44, 88, 176, 352
6	13, 26, 52, 104, 208
7	15, 30, 60, 120, 240

Table A.2: Lifting factor Zc table in 5G NR

A.3 BER vs SNR

The performance graph is obtained from the OAI LDPC technical report that shows the performance of BG1 with the largest block size of B = 8448 and the highest code rate R = 8/9. Since the current implementation transmits only one block into testbench in each iteration, the Block Error Rate(BLER) is equal to BER. It could be observed that the performance gap is only about 0.3 dB if 50 iterations are used compared to the reference performance provided by HUAWEI. However, for 5 iterations, there is still a significant performance loss of about 2.3dB at BLER 10^{-2}.

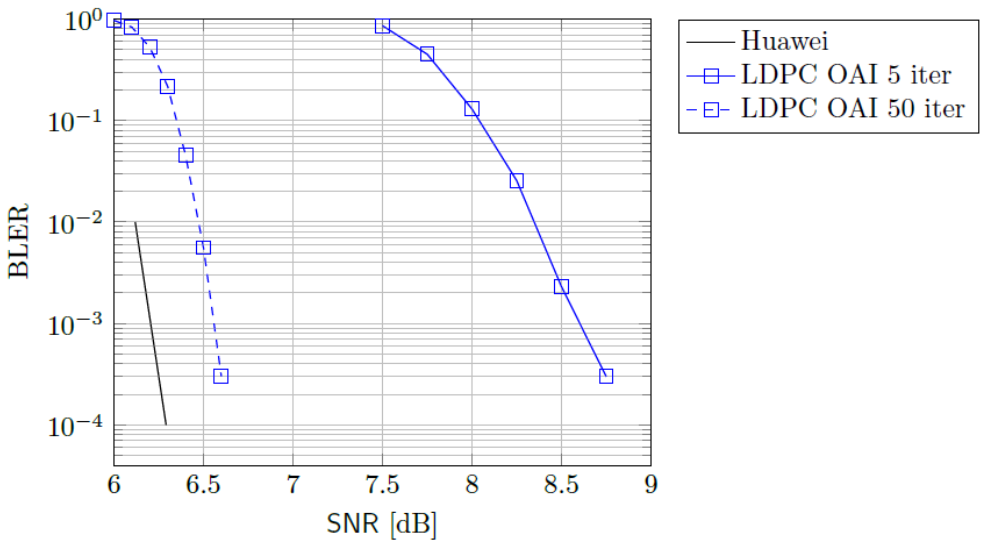

Figure A.1: BLER vs. SNR, BG2, Rate = 1/5, max iteration = 50, B= 1280

Appendix B

Appendix B reports the detailed implementation of the OAI AVX2-based LDPC decoder and its corresponding synthesizable version.

B.1 Description of LDPC decoder functions

A summary of the LDPC decoder functions provided by the author of the source code is reported in Table B.1

Function	Description
`llr2llrProcBuf`	Copies input LLRs to LLR processing buffer
`llr2CnProcBuf`	Copies input LLRs to CN processing buffer
`cnProc`	Performs CN signal processing
`cnProcPc`	Performs parity check
`cn2bnProcBuf`	Copies the CN results to the BN processing buffer
`bnProcPc`	Performs BN processing for parity check and/or hard-decision
`bnProc`	Utilizes the results of `bnProcPc` to compute LLRs for CN processing
`bn2cnProcBuf`	Copies the BN results to the CN processing buffer
`llrRes2llrOut`	Copies the results of `bnProcPc` to output LLRs
`llr2bit`	Performs hard-decision on the output LLRs

Table B.1: Summary of the LDPC decoder functions

B.2 Check node groups and bit node groups

Interconnection between BNs and CNs is determined by Base Graph. In NR LDPC, two base graphs have fixed ways of nodes organization. Denote |Bi| the g=number of connected BNs to CN i and let M|Bi| be the number of CNs that are connected to the same number of BNs.

$\lvert\mathcal{B}_i\rvert$	3	4	5	6	7	8	9	10	19
$M_{\lvert\mathcal{B}_i\rvert}^{\mathrm{BG1}}$	1	5	18	8	5	2	2	1	4
$M_{\lvert\mathcal{B}_i\rvert}^{\mathrm{BG2}}$	6	20	9	3	0	2	0	2	0

Table B.2: Check node groups for BG1 and BG2

The same mapping method is applied on BN groups. The number of connected CNs to BN j is |Cj|; besides, K|Cj| is the number of BNs that are connected to the same number of CNs.

$\lvert\mathcal{B}_i\rvert$	3	4	5	6	7	8	9	10	19
$M_{\lvert\mathcal{B}_i\rvert}^{\mathrm{BG1}}$	1	5	18	8	5	2	2	1	4
$M_{\lvert\mathcal{B}_i\rvert}^{\mathrm{BG2}}$	6	20	9	3	0	2	0	2	0

Table B.3: Bit node groups for BG1 and BG2 for base rates 1/3 and 1/5, respectively

B.3 Shift factors for memory copy

According to current implementation, shift factor varies depending on BG, Z and group number. A snippet of constant factors is shown:

Listing B.1: Example of shift factors

```
1 static const uint16_t circShift_BG2_Z2_CNG6[6][3] = {{1, 1, 1},{1, 0,
      1},{0, 1, 0},{1, 0, 1},{1, 0, 0},{0, 0, 0}};
2 static const uint16_t circShift_BG2_Z3_CNG6[6][3] = {{1, 0, 2},{2, 2,
      0},{1, 1, 1},{2, 1, 0},{0, 1, 2},{0, 0, 0}};
3 static const uint16_t circShift_BG2_Z4_CNG6[6][3] = {{3, 3, 1},{1, 0,
      3},{2, 1, 0},{3, 0, 3},{3, 2, 0},{0, 0, 0}};
4 static const uint16_t circShift_BG2_Z5_CNG6[6][3] = {{0, 0, 0},{1, 4,
      0},{2, 4, 3},{1, 0, 1},{4, 3, 3},{0, 0, 0}};
5 static const uint16_t circShift_BG2_Z6_CNG6[6][3] = {{4, 3, 2},{2, 2,
      0},{1, 4, 4},{2, 1, 0},{0, 4, 5},{0, 0, 0}};
6 static const uint16_t circShift_BG2_Z7_CNG6[6][3] = {{3, 4, 5},{5, 6,
      6},{2, 3, 0},{2, 5, 3},{4, 6, 6},{0, 0, 0}};
7 static const uint16_t circShift_BG2_Z8_CNG6[6][3] = {{7, 3, 1},{1, 4,
      3},{2, 5, 4},{7, 4, 3},{7, 6, 4},{0, 0, 0}};
```

```
8  static const uint16_t circShift_BG2_Z9_CNG6[6][3] = {{4, 1, 7},{5, 6,
       0},{3, 8, 8},{2, 7, 0},{8, 4, 2},{0, 0, 0}};
9  static const uint16_t circShift_BG2_Z10_CNG6[6][3] = {{0, 0, 0},{1,
       4, 5},{2, 9, 8},{1, 5, 6},{4, 8, 8},{0, 0, 0}};
10 static const uint16_t circShift_BG2_Z11_CNG6[6][3] = {{2, 3, 4},{7,
       0, 4},{2, 10, 2},{4, 1, 3},{5, 7, 0},{0, 0, 0}};
11 static const uint16_t circShift_BG2_Z12_CNG6[6][3] = {{10, 9, 8},{8,
       8, 6},{1, 4, 4},{8, 1, 6},{0, 4, 11},{0, 0, 0}};
12 static const uint16_t circShift_BG2_Z13_CNG6[6][3] = {{6, 7, 7},{10,
       8, 0},{10, 3, 0},{2, 12, 3},{10, 12, 6},{0, 0, 0}};
13 static const uint16_t circShift_BG2_Z14_CNG6[6][3] = {{3, 11, 5},{12,
       13, 13},{2, 3, 7},{9, 12, 10},{11, 13, 13},{0, 0, 0}};
14 static const uint16_t circShift_BG2_Z15_CNG6[6][3] = {{1, 2, 11},{11,
       12, 6},{4, 7, 12},{13, 4, 7},{12, 11, 4},{0, 0, 0}};
15 static const uint16_t circShift_BG2_Z16_CNG6[6][3] = {{7, 11, 1},{9,
       4, 3},{2, 13, 12},{15, 12, 3},{7, 14, 4},{0, 0, 0}};
16 static const uint16_t circShift_BG2_Z18_CNG6[6][3] = {{4, 1, 7},{14,
       15, 9},{12, 17, 17},{11, 7, 0},{17, 13, 11},{0, 0, 0}};
17 static const uint16_t circShift_BG2_Z20_CNG6[6][3] = {{0, 0, 0},{11,
       4, 5},{2, 19, 8},{1, 5, 16},{4, 8, 18},{0, 0, 0}};
18 static const uint16_t circShift_BG2_Z22_CNG6[6][3] = {{13, 3, 4},{18,
       0, 15},{13, 10, 2},{15, 1, 3},{16, 7, 11},{0, 0, 0}};
19 static const uint16_t circShift_BG2_Z24_CNG6[6][3] = {{10, 9, 8},{20,
       20, 18},{1, 4, 16},{8, 1, 6},{0, 16, 23},{0, 0, 0}};
20 static const uint16_t circShift_BG2_Z26_CNG6[6][3] = {{6, 7, 7},{23,
       8, 0},{10, 16, 13},{2, 12, 3},{10, 12, 6},{0, 0, 0}};
21 static const uint16_t circShift_BG2_Z28_CNG6[6][3] = {{17, 11,
       19},{26, 27, 13},{2, 3, 21},{23, 26, 10},{25, 13, 13},{0, 0, 0}};
22 static const uint16_t circShift_BG2_Z30_CNG6[6][3] = {{1, 17,
       26},{11, 12, 6},{19, 22, 27},{28, 4, 22},{27, 26, 4},{0, 0, 0}};
23 static const uint16_t circShift_BG2_Z32_CNG6[6][3] = {{7, 27, 1},{9,
       4, 19},{2, 13, 12},{31, 28, 3},{7, 30, 20},{0, 0, 0}};
24 static const uint16_t circShift_BG2_Z36_CNG6[6][3] = {{4, 1, 25},{32,
       15, 27},{12, 35, 35},{29, 25, 0},{35, 31, 29},{0, 0, 0}};
25 static const uint16_t circShift_BG2_Z40_CNG6[6][3] = {{0, 0, 0},{11,
       4, 5},{22, 19, 28},{21, 5, 16},{24, 28, 38},{0, 0, 0}};
```

B.4 Auxiliary functions

Enable macros for calling these auxiliary functions so that execution time is printed and staged output is stored.

Listing B.2: Conditional compilation of measurement functions

```
1      // BN processing
2  #ifdef NR_LDPC_PROFILER_DETAIL
```

```
3       start_meas(&p_profiler->bnProcPc);
4   #endif
5       nrLDPC_bnProcPc(p_lut, p_procBuf, Z);
6   #ifdef NR_LDPC_PROFILER_DETAIL
7       stop_meas(&p_profiler->bnProcPc);
8   #endif
9
10  #ifdef NR_LDPC_DEBUG_MODE
11      nrLDPC_debug_initBuffer2File(nrLDPC_buffers_LLR_RES);
12      nrLDPC_debug_writeBuffer2File(nrLDPC_buffers_LLR_RES, p_procBuf);
13  #endif
14
15  #ifdef NR_LDPC_PROFILER_DETAIL
16      start_meas(&p_profiler->bnProc);
17  #endif
18      nrLDPC_bnProc(p_lut, p_procBuf, Z);
19  #ifdef NR_LDPC_PROFILER_DETAIL
20      stop_meas(&p_profiler->bnProc);
21  #endif
22
23  #ifdef NR_LDPC_DEBUG_MODE
24      nrLDPC_debug_initBuffer2File(nrLDPC_buffers_BN_PROC_RES);
25      nrLDPC_debug_writeBuffer2File(nrLDPC_buffers_BN_PROC_RES,
        p_procBuf);
26  #endif
```

Appendix C

Appendix C reports several source codes of synthesizable functions converted from the software solution of LDPC decoder provided by OAI.

C.1 AVX2 instructions

Synopsis and description of original AVX2 instructions used by decoder are listed below.

Num.	Synopsis	Description
1	__m256i _mm256_abs_epi8 (__m256i a)	Compute the absolute value of packed signed 8-bit integers in a, and store the unsigned results in dst.
2	__m256i _mm256_adds_epi8 (__m256i a, __m256i b)	Add packed 8-bit integers in a and b using saturation, and store the results in dst.
3	__m256i _mm256_and_si256 (__m256i a, __m256i b)	Compute the bitwise AND of 256 bits (representing integer data) in a and b, and store the result in dst.
4	__m256i _mm256_cmpgt_epi8 (__m256i a, __m256i b)	Compare packed signed 8-bit integers in a and b for greater-than, and store the results in dst.
5	__m256i _mm256_min_epu8 (__m256i a, __m256i b)	Compare packed unsigned 8-bit integers in a and b, and store packed minimum values in dst.
6	int _mm256_movemask_epi8 (__m256i a)	Create mask from the most significant bit of each 8-bit element in a, and store the result in dst.
7	__m256i _mm256_subs_epi8 (__m256i a, __m256i b)	Subtract packed signed 8-bit integers in b from packed 8-bit integers in a using saturation, and store the results in dst.
8	__m256i _mm256_sign_epi8 (__m256i a, __m256i b)	Negate packed signed 8-bit integers in a when the corresponding signed 8-bit integer in b is negative, and store the results in dst. Element in dst are zeroed out when the corresponding element in b is zero.
9	__m256i _mm256_shuffle_epi8 (__m256i a, __m256i b)	Shuffle 8-bit integers in a within 128-bit lanes according to shuffle control mask in the corresponding 8-bit element of b, and store the results in dst.
10	__m256i _mm256_permute4x64_epi64 (__m256i a, const int imm8)	Shuffle 64-bit integers in a across lanes using the control in imm8, and store the results in dst.
11	__m256i _mm256_packs_epi16 (__m256i a, __m256i b)	Convert packed signed 16-bit integers from a and b to packed 8-bit integers using signed saturation, and store the results in dst.
12	__m256i _mm256_adds_epi16 (__m256i a, __m256i b)	Add packed 16-bit integers in a and b using saturation, and store the results in dst.
13	__m256i _mm256_cvtepi8_epi16 (__m128i a)	Sign extend packed 8-bit integers in a to packed 16-bit integers, and store the results in dst.

Table C.1: Description of AVX2 instructions

C.2 Converted AVX2 functions

Function names are slightly modified for simplicity in recognizing. Several functions are passing by reference instead of passing by value.

Num.	From	To
1	__m256i _mm256_abs_epi8 (__m256i a)	m256i mm256_abs_epi8 (int8_t *a)
2	__m256i _mm256_adds_epi8 (__m256i a, __m256i b)	m256i mm256_adds_epi8(int8_t *a, int8_t *b)
3	__m256i _mm256_and_si256 (__m256i a, __m256i b)	m256i mm256_and_si256(m256i a, m256i b)
4	__m256i _mm256_cmpgt_epi8 (__m256i a, __m256i b)	m256i mm256_cmpgt_epi8(m256i a, int8_t *b)
5	__m256i _mm256_min_epu8 (__m256i a, __m256i b)	m256i mm256_min_epu8(m256i a, m256i b)
6	int _mm256_movemask_epi8 (__m256i a)	int mm256_movemask_epi8(m256i a)
7	__m256i _mm256_subs_epi8 (__m256i a, __m256i b)	m256i mm256_subs_epi8(int8_t *a, int8_t *b)
8	__m256i _mm256_sign_epi8 (__m256i a, __m256i b)	m256i mm256_sign_epi8(m256i a, int8_t *b)
9	__m256i _mm256_shuffle_epi8 (__m256i a, __m256i b)	m256i mm256_shuffle_epi8(int8_t *a, m256i b)
10	__m256i _mm256_permute4x64_epi64 (__m256i a, const int imm8)	m256i mm256_permute4x64_epi64(m256i a, int8_t b)
11	__m256i _mm256_packs_epi16 (__m256i a, __m256i b)	m256i mm256_packs_epi16(m256i a, m256i b)
12	__m256i _mm256_adds_epi16 (__m256i a, __m256i b)	m256i mm256_adds_epi16(m256i a, m256i b)
13	__m256i _mm256_cvtepi8_epi16 (__m128i a)	m256i mm256_cvtepi8_epi16(int8_t *a)

Table C.2: Synopsis of new AVX2 instructions and the original version

C.3 Intrinsic AVX2 functions

Listing C.1: Intrinsic AVX2 functions

```
m256i mm256_abs_epi8(int8_t *a){
    m256i dest =
    {{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
     ,0,0,0}};
    int i;
loop_intrin_abs: for(i=0; i<32; i++){
        if(*a < 0)
            dest.data[i] = (*a ^ 0xFF)+1;
        else
            dest.data[i] = *a;
           a++;
    }
    return dest;
}

```

```
15 m256i mm256_adds_epi8(int8_t *a, int8_t *b){
16     int i;
17     m256i dest =
       {{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
18     ,0,0,0}};
19     int16_t adds; // to handle overflow
20 loop_intrin_adds_epi8:for(i=0; i<32; i++){
21       adds = *a + *b;
22       if(adds < -128)
23      dest.data[i] = -128;
24       else if(adds > 127)
25      dest.data[i] = 127;
26       else
27      dest.data[i] = (int8_t) adds;
28        a++;
29        b++;
30      }
31       return dest;
32 }
33
34 m256i mm256_and_si256(m256i a, m256i b){
35      m256i dest
       ={{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
36      ,0,0,0}};
37      int i;
38 loop_intrin_and: for(i=0; i<32; i++){
39          dest.data[i] = a.data[i] & b.data[i];
40      }
41      return dest;
42 }
43
44 m256i mm256_cmpgt_epi8(m256i a, int8_t *b){
45      m256i dest
       ={{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
46      ,0,0,0}};
47      int i;
48 loop_intrin_cmp: for(i=0; i<32; i++){
49          dest.data[i] = ( a.data[i] > *b ) ? 0xFF : 0;
50                  b++;
51      }
52      return dest;
53 }
54
55 m256i mm256_min_epu8(m256i a, m256i b){
56      m256i dest =
       {{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
57      ,0,0,0}};
58      int i;
59 loop_intrin_min:  for(i=0; i<32; i++)
```

```
60      {
61          if((uint8_t)a.data[i] >(uint8_t)b.data[i])
62          dest.data[i] = b.data[i];
63          else
64          dest.data[i] = a.data[i];
65      }
66      return dest;
67  }
68
69  int mm256_movemask_epi8(m256i a){
70      int i;
71      int MSB;
72      int mask = 0;
73  loop_intrin_movemask:    for(i=0; i<32; i++){
74          // to expand 8 bits to 32, 24 zeroes are padded @ msb side
75          MSB = (a.data[i] >> 7) & 0x00000001;
76          mask = mask | (MSB << (i));
77      }
78      return mask;
79  }
80
81  m256i mm256_subs_epi8(int8_t *a, int8_t *b){
82      m256i dest
        ={{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
83      ,0,0,0}};
84      int16_t subs; // to handle overflow
85      int i;
86  loop_intrin_subs: for(i=0; i<32; i++){
87          subs = *a - *b;
88          if(subs < -128)
89              dest.data[i] = -128;
90          else if(subs > 127)
91              dest.data[i] = 127;
92          else
93              dest.data[i] = (int8_t) subs;
94            a++;
95            b++;
96      }
97      return dest;
98  }
99
100 m256i mm256_sign_epi8_ori(m256i a, m256i b){
101 m256i dest =
        {{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
102 ,0,0,0}};
103     int i;
104 loop_intrin_sign: for(i=0; i<32; i++){
105         if(b.data[i] < 0)
106             dest.data[i] = a.data[i] *(-1);
```

```
107         else if(b.data[i] == 0)
108             dest.data[i] = 0;
109         else
110             dest.data[i] = a.data[i];
111     }
112     return dest;
113 }
114
115 m256i mm256_sign_epi8(m256i a, int8_t *b){
116     m256i dest =
        {{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
117     ,0,0,0,0}};
118     int i;
119 loop_intrin_sign_new: for(i=0; i<32; i++){
120         if(*b < 0)
121             dest.data[i] = a.data[i] *(-1);
122         else if(*b == 0)
123             dest.data[i] = 0;
124         else
125             dest.data[i] = a.data[i];
126                 b ++;
127 }
128     return dest;
129 }
130
131 m256i mm256_shuffle_epi8(int8_t *a, m256i b){
132     m256i dest
        ={{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
133     ,0,0,0,0}};
134     int i;
135     // msb part
136 loop_intrin_shuffle_1:  for(i=0; i<16; i++){
137         if(b.data[i] & 0x80)
138             dest.data[i] = 0;
139         else
140             dest.data[i] = a[b.data[i] & 0x0F];
141     }
142     // lsb part
143 loop_intrin_shuffle_2:  for(i=0; i<16; i++){
144         if(b.data[i+16] & 0x80)
145             dest.data[i+16] = 0;
146         else
147             dest.data[i+16] = a[16+(b.data[i+16] & 0x0F)];
148     }
149     return dest;
150 }
151
152 // brief explanation
```

```
153 // b is 8 bit wide, according to b the value of the i-64 pack is
        decided
154 // the decision is taken on two bits of b. To evaluate them, a right
        shift is done
155 // then a mask of 00000011 is ANDED to evaluate only those two bits
156 // first couple  . no shift
157 // second couple . shift by 2 (must trash couple 1)
158 // third couple  . shift by 4 (must trash couples 1-2)
159 // fourth couple . shift by 6 (must trash couples 1-2-3)
160 m256i mm256_permute4x64_epi64(m256i a, int8_t b){
161         m256i dest
        ={{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
162         ,0,0,0}};
163         int i;
164      switch(b & 0x03){  // checking bits 0-1
165         case 0:
166 loop_intrin_permute_1:  for(i=0; i<8; i++) {
167             dest.data[i] = a.data[i]; }
168             break;
169         case 1:
170 loop_intrin_permute_2:  for(i=0; i<8; i++) {
171             dest.data[i] = a.data[i+8];  }
172             break;
173         case 2:
174 loop_intrin_permute_3:  for(i=0; i<8; i++) {
175             dest.data[i] = a.data[i+16];  }
176             break;
177         case 3:
178 loop_intrin_permute_4:  for(i=0; i<8; i++) {
179             dest.data[i] = a.data[i+24];  }
180             break;
181         default: ;
182             break;
183     }
184     switch((b>>2 & 0x03)){  // checking bits 2-3
185         case 0:
186 loop_intrin_permute_5:  for(i=0; i<8; i++) {
187             dest.data[i+8] = a.data[i];  }
188             break;
189         case 1:
190 loop_intrin_permute_6:  for(i=0; i<8; i++) {
191             dest.data[i+8] = a.data[i+8];  }
192             break;
193         case 2:
194 loop_intrin_permute_7:  for(i=0; i<8; i++) {
195             dest.data[i+8] = a.data[i+16];  }
196             break;
197         case 3:
198 loop_intrin_permute_8:  for(i=0; i<8; i++) {
```

```
199             dest.data[i+8] = a.data[i+24];  }
200             break;
201         default: break;
202     }
203     switch((b>>4 &0x03)){  // checking bits 4-5
204         case 0:
205 loop_intrin_permute_9:  for(i=0; i<8; i++) {
206             dest.data[i+16] = a.data[i];  }
207             break;
208         case 1:
209 loop_intrin_permute_10:  for(i=0; i<8; i++) {
210             dest.data[i+16] = a.data[i+8];  }
211             break;
212         case 2:
213 loop_intrin_permute_11:  for(i=0; i<8; i++) {
214             dest.data[i+16] = a.data[i+16];  }
215             break;
216         case 3:
217 loop_intrin_permute_12:  for(i=0; i<8; i++) {
218             dest.data[i+16] = a.data[i+24];  }
219             break;
220          default: ;
221             break;
222     }
223     switch((b>>6 &0x03)){  // checking bits 6-7
224         case 0:
225 loop_intrin_permute_13:  for(i=0; i<8; i++) {
226             dest.data[i+24] = a.data[i];  }
227             break;
228         case 1:
229 loop_intrin_permute_14:  for(i=0; i<8; i++) {
230             dest.data[i+24] = a.data[i+8];  }
231             break;
232         case 2:
233 loop_intrin_permute_15:  for(i=0; i<8; i++) {
234             dest.data[i+24] = a.data[i+16];  }
235             break;
236         case 3:
237 loop_intrin_permute_16:  for(i=0; i<8; i++) {
238             dest.data[i+24] = a.data[i+24];  }
239             break;
240         default: ;
241             break;
242     }
243     return dest;
244 }
245
246 m256i mm256_packs_epi16(m256i a, m256i b){
247         int16_t p_a[16], p_b[16];
```

```
248         int16_t tmp_a1,tmp_a2,tmp_b1,tmp_b2, tmp_a, tmp_b ;
249         int j;
250 loop_intrin_packs_1:for (j=0; j<16; j++)  {
251         tmp_a1 = a.data[2*j+1];  // a : 1, 3, 5..
252         tmp_a2 = a.data[2*j]; // a : 0, 2, 4 ..
253         tmp_b1 = b.data[2*j+1];  // b : 1, 3, 5 ..
254         tmp_b2 = b.data[2*j];  // b : 0, 2, 4..
255         tmp_a1 = tmp_a1 << 8;
256         tmp_b1 = tmp_b1 << 8;
257                 tmp_a2 = tmp_a2 & 0xFF;
258                 tmp_b2 = tmp_b2 & 0xFF;
259                 p_a[j] = tmp_a1 + tmp_a2;
260                 p_b[j] = tmp_b1 + tmp_b2;
261 }
262     m256i dest
    ={{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
263     ,0,0,0,0}};
264     int i;
265     int16_t tmp; // for saturation
266 loop_intrin_packs_2:     for(i=0; i<8; i++){
267         tmp = p_a[i];
268         if(tmp < -128)
269             dest.data[i] = -128;
270         else if(tmp > 127)
271             dest.data[i] = 127;
272         else dest.data[i] = tmp;     // p_a, dest  updated,
273     }
274 loop_intrin_packs_3:     for(i=0; i<8; i++){
275         tmp = p_b[i];
276         if(tmp < -128)
277             dest.data[i+8] = -128;
278         else if(tmp > 127)
279             dest.data[i+8] = 127;
280         else dest.data[i+8] = tmp;     // p_b, dest updated
281     }
282 loop_intrin_packs_4:     for(i=0; i<8; i++){
283         tmp = p_a[i+8];
284         if(tmp < -128)
285             dest.data[i+16] = -128;
286         else if(tmp > 127)
287             dest.data[i+16] = 127;
288         else dest.data[i+16] = tmp;    // p_a, dest updated
289     }
290 loop_intrin_packs_5:     for(i=0; i<8; i++){
291         tmp = p_b[i+8];
292         if(tmp < -128)
293             dest.data[i+24] = -128;
294         else if(tmp > 127)
295             dest.data[i+24] = 127;
```

```
296         else dest.data[i+24] = tmp;
297     }
298     return dest;
299 }
300
301 // position i of a given data has the MSB part, i+1 has the LSB part
    of the result
302 m256i mm256_adds_epi16(m256i a, m256i b){
303         int16_t p_a[16], p_b[16];
304         int16_t tmp_a1,tmp_a2,tmp_b1,tmp_b2, tmp_a, tmp_b ;
305         int j;
306 loop_intrin_adds_epi16_1:for (j=0; j<16; j++)  {
307         tmp_a1 = a.data[2*j+1];  // a : 1, 3, 5 ..
308         tmp_a2 = a.data[2*j]; // a : 0, 2, 4..
309         tmp_b1 = b.data[2*j+1];  // b : 1, 3, 5 ..
310         tmp_b2 = b.data[2*j];  // b : 0, 2, 4..
311         tmp_a1 = tmp_a1 << 8;
312         tmp_b1 = tmp_b1 << 8;
313                 tmp_a2 = tmp_a2 & 0xFF;
314                 tmp_b2 = tmp_b2 & 0xFF;
315                 p_a[j] = tmp_a1 + tmp_a2;
316                 p_b[j] = tmp_b1 + tmp_b2;
317 }
318     m256i dest
    ={{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
319     ,0,0,0}};
320     int adds;
321     int i;
322 loop_intrin_adds_epi16_2:for(i=0; i<16; i++){
323         adds = p_a[i] + p_b[i];
324         if(adds < -32768) {
325             dest.data[2*i+1] = 0x80;
326             dest.data[2*i] = 0; }
327         else if(adds > 32767) {
328             dest.data[2*i+1] = 0xEF;
329             dest.data[2*i] = 0xFF; }
330         else {
331             dest.data[2*i+1] = (int8_t)((adds & 0xFF00)>>8)  ;// MSB
332             dest.data[2*i] =  (int8_t)(adds & 0x00FF)      ;// LSB
333 }   }
334     return dest;
335 }
336 m256i mm256_cvtepi8_epi16(int8_t *a){
337     m256i dest
    ={{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
338     ,0,0,0}};
339     int i;
340 loop_intrin_cvt:for(i=0; i<16; i++){
341         dest.data[2*i] = *a;// LSB // the original value
```

```
342                 if(*a < 0) {
343         dest.data[2*i+1] = -1; }// MSB // if >0, then 0, if <0, then
    -1
344                 else {
345                 dest.data[2*i+1] = 0;  }
346                 a++;
347     }
348     return dest;
349 }
350 m256i  mm256_conv_int8(int8_t* a) {
351     m256i dest =
    {{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
352     ,0,0,0}} ; //32 values
353     int8_t*  p_a;
354     p_a = a;
355     int i;
356 loop_intrin_conv256:     for(i=0; i<32; i++) {
357         dest.data[i] = *p_a;
358         p_a ++;
359            }
360     return dest;
361 }
362 m128i  mm128_conv_int8(int8_t* a) {
363     m128i dest = {{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}} ; // 16 values
364     int8_t*  p_a;
365     p_a = a;
366     int i;
367     for(i=0; i<16; i++) {
368         dest.data[i] = *p_a;
369         p_a ++;
370            }
371     return dest;
372 }
373 // 8 bits of a, from LSB, every 8 bit copy to the pointer(array) b,
    in sequence.
374 int8_t*  mm32_store_int8(uint32_t a, int8_t* b) {
375      int8_t* p_b;
376      p_b = b;
377      int8_t tmp1,tmp2,tmp3,tmp4;
378      tmp1 = (int8_t)(a & 0x000000FF);
379      tmp2 = (int8_t)((a >> 8) & 0x000000FF);
380      tmp3 = (int8_t)((a >> 16) & 0x000000FF);
381      tmp4 = (int8_t)((a >> 24)& 0x000000FF);
382      *p_b     = tmp1;
383      *(p_b+1) = tmp2;
384      *(p_b+2) = tmp3;
385      *(p_b+3) = tmp4;
386      return b;
387     }
```

C.4 Constraints on usage of pointers

In listing 4.2 and 4.3 the original unsynthesizable implementation is reported. In listing 4.4 the synthesizable version is reported.

Listing C.2: Multiple destination of a pointer(1)

```
    if (outMode == nrLDPC_outMode_LLRINT8)
    {
        p_llrOut = p_out;
    }
    else
    {
        // Use LLR processing buffer as temporary output buffer
        p_llrOut = p_procBuf->llrProcBuf;
        // Clear llrProcBuf
        memset(p_llrOut,0, NR_LDPC_MAX_NUM_LLR*sizeof(int8_t));
    }
```

Listing C.3: Multiple destination of a pointer(2)

```
    // Assign results from processing buffer to output
#ifdef NR_LDPC_PROFILER_DETAIL
    start_meas(&p_profiler->llrRes2llrOut);
#endif
    nrLDPC_llrRes2llrOut(p_lut, p_llrOut, p_procBuf, Z, BG);
#ifdef NR_LDPC_PROFILER_DETAIL
    stop_meas(&p_profiler->llrRes2llrOut);
#endif

    // Hard-decision
#ifdef NR_LDPC_PROFILER_DETAIL
    start_meas(&p_profiler->llr2bit);
#endif
    if (outMode == nrLDPC_outMode_BIT)
    {
        nrLDPC_llr2bitPacked(p_out, p_llrOut, numLLR);
    }
    else if (outMode == nrLDPC_outMode_BITINT8)
    {
        nrLDPC_llr2bit(p_out, p_llrOut, numLLR);
    }

#ifdef NR_LDPC_PROFILER_DETAIL
    stop_meas(&p_profiler->llr2bit);
#endif

    return i;
}
```

Listing C.4: Synthesizable version

```
    // Assign results from processing buffer to output
    if (outMode == nrLDPC_outMode_LLRINT8)
        {
        nrLDPC_llrRes2llrOut(p_out, buf_llrRes, Z, BG);
        }
        else
        {
        nrLDPC_llrRes2llrOut(buf_llrProcBuf, buf_llrRes, Z, BG);
        }

    // Hard-decision
    if (outMode == nrLDPC_outMode_BIT)
    {
        nrLDPC_llr2bitPacked(p_out, buf_llrProcBuf, numLLR);
    }
    else if (outMode == nrLDPC_outMode_BITINT8)
    {
        nrLDPC_llr2bit(p_out, buf_llrProcBuf, numLLR);
    }

    return i;
}
```

Bibliography

[1] In Lee and Kyoochun Lee. «The Internet of Things (IoT): Applications, investments, and challenges for enterprises». In: *Business Horizons* 58.4 (2015), pp. 431–440 (cit. on p. 1).

[2] Dennis Miller. «Blockchain and the internet of things in the industrial sector». In: *IT professional* 20.3 (2018), pp. 15–18 (cit. on p. 1).

[3] Shancang Li, Li Da Xu, and Shanshan Zhao. «5G Internet of Things: A survey». In: *Journal of Industrial Information Integration* 10 (2018), pp. 1–9 (cit. on p. 1).

[4] Erik Dahlman, Gunnar Mildh, Stefan Parkvall, Janne Peisa, Joachim Sachs, Yngve Selén, and Johan Sköld. «5G wireless access: requirements and realization». In: *IEEE Communications Magazine* 52.12 (2014), pp. 42–47 (cit. on p. 1).

[5] Robert C Daniels and Robert W Heath. «60 GHz wireless communications: Emerging requirements and design recommendations». In: *IEEE Vehicular technology magazine* 2.3 (2007), pp. 41–50 (cit. on p. 1).

[6] Claude Elwood Shannon. «Communication in the presence of noise». In: *Proceedings of the IRE* 37.1 (1949), pp. 10–21 (cit. on p. 2).

[7] Michaël Antonie van Wyk, Li Ping, and Guanrong Chen. «Multivaluedness in Networks: Shannon's Noisy-Channel Coding Theorem». In: *IEEE Transactions on Circuits and Systems II: Express Briefs* (2021) (cit. on p. 2).

[8] MJEE Golay. «Binary coding». In: *Transactions of the IRE Professional Group on Information Theory* 4.4 (1954), pp. 23–28 (cit. on p. 2).

[9] David E Muller. «Application of Boolean algebra to switching circuit design and to error detection». In: *Transactions of the IRE professional group on electronic computers* 3 (1954), pp. 6–12 (cit. on p. 2).

[10] Yongwoo Lee, Wijik Lee, Young Sik Kim, and Jong-Seon No. «Modified pqsigRM: RM Code-Based Signature Scheme». In: *IEEE Access* 8 (2020), pp. 177506–177518 (cit. on p. 2).

[11] Palash Sarkar, Sudhan Majhi, and Zilong Liu. «Optimal Z-complementary code set from generalized Reed-Muller codes». In: *IEEE Transactions on Communications* 67.3 (2018), pp. 1783–1796 (cit. on p. 2).

[12] R Muthammal and S Srinivasa Rao Madhane. «Design, analysis and FPGA implementation LDPC codes with BCH codes». In: *2013 International Conference on Current Trends in Engineering and Technology (ICCTET)*. IEEE. 2013, pp. 242–244 (cit. on p. 2).

[13] Ahmad Baheej Al-Khalil and Alyaa Al-Barrak. «Performance of BCH and RS Codes in MIMO System Using MPFEC Diversity Technique». In: *2018 International Conference on Advanced Science and Engineering (ICOASE)*. IEEE. 2018, pp. 122–127 (cit. on p. 3).

[14] Cagri Tanriover, Bahram Honary, Jun Xu, and Shu Lin. «Improving turbo code error performance by multifold coding». In: *IEEE communications letters* 6.5 (2002), pp. 193–195 (cit. on p. 3).

[15] Robert Gallager. «Low-density parity-check codes». In: *IRE Transactions on information theory* 8.1 (1962), pp. 21–28 (cit. on pp. 3, 6, 10).

[16] Declan O'Loughlin, Aedan Coffey, Frank Callaly, Darren Lyons, and Fearghal Morgan. «Xilinx vivado high level synthesis: Case studies». In: (2014) (cit. on p. 3).

[17] Kavitha Sunil, Poorna Jayaraj, and KP Soman. «Message passing algorithm: A tutorial review». In: *International Organisation of Scientific Research* 2 (2012), pp. 12–24 (cit. on p. 6).

[18] *Multiplexing and channel coding*. 3GPP, 2018 (cit. on p. 28).

[19] Yi Xu, Wenjin Wang, Zhen Xu, and Xiqi Gao. «AVX-512 based software decoding for 5G LDPC codes». In: *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE. 2019, pp. 54–59 (cit. on p. 65).

[20] *Medium Access Control (MAC) protocol specification*. 3GPP, 2019 (cit. on p. 69).