High level synthesis of L-bubble check algorithm for check node processing

Himanshu Sharma, Manju Choudhary, Vikas Pathak, Ila Roy Saxena
Department of Electronics and Communication, Swami Keshvanand Institute of Technology, Management and Gramothan, Jaipur-302017 (INDIA)
Email- sharmahimanshu290493@gmail.com
Received 23.08.2019 received in revised form 16.12.2019, accepted 21.12.2019

Abstract: This paper deals with the low complexity algorithm for the check node processing i.e. L-bubble check algorithm in non-binary LDPC decoders. After a review of the state-of-the-art, there is a focus on a reduction of hardware requirement for check node processing using high level synthesis. High level synthesis helps in optimizing the hardware design to a great extent. This motivated to use high level synthesis for implementing the L-bubble check algorithm. Finally, high level synthesis results are presented which shows that the number of slices required to implement the L-bubble check algorithm using HLS is 204.

Keywords: Non-Binary Low Density Parity Check decoders, check node processing, L-bubble check algorithm, high level synthesis.

1. INTRODUCTION

Low-density parity-check codes that are popularly called as LDPC codes are a subset of linear block codes having characteristic that they can approach the Shannon limit. They were first presented in [1] by R. Gallager in 1962 but did not get much attention for use. Later in 1999, they were rediscovered by D.J.C. Mackay in [2]. Non-Binary Low Density Parity Check (NB-LDPC) codes have various advantages like they have good performance, high speed and high throughput codes with reference to the currently used codes like turbo codes, hamming codes, etc [3]. They work well when the code rate is high. There are mainly two important issues behind not using them. The first issue is the decoders that are used to decode these codes have high complexity [4] in terms of the hardware requirement. The second issue was that the IC technology was not that much in use as it is in the present decade. So, no attention was given to them for their practical use. For the last few years, due to the development of IC technology, the focus is given to these codes for practical use in current applications where its benefits can be exploited. Currently, the Binary variant of these codes gains much popularity and is used in various applications like WiMAX wireless communications, digital video broadcasting, 10GBase-T Ethernet and magnetic and solid-state drives. They are also in use in many Orthogonal Frequency Division Multiplexing (OFDM) systems.

Non-binary LDPC codes have better performance and error correction capabilities. But the major problem is that they require high amount of hardware when implemented using VHDL code. So, in this work, the L-bubble check algorithm used in its decoder is implemented using High level synthesis to reduce hardware requirement which is the novelty of this work.

The paper is organized as follows: Section II explains the basic overview of check node processing, L-bubble check algorithm, and high level synthesis. Section III explains the methodology used. Section IV shows the synthesis results for the L-bubble check algorithm. Section V concludes the discussion.

2. CHECK NODE PROCESSING

2.1 Decoding of NB-LDPC codes

Those decoding algorithms which are used for binary LDPC codes can be easily modified and used for decoding the non-binary LDPC codes. Currently, belief propagation decoding algorithm, Min-Max decoding algorithm, Extended Min-Sum (EMS) [4] decoding algorithm are used for decoding these codes. Some decoders involve multiplication operation for decoding but implementing multiplication operation on the hardware results in high resource utilization and floor area. So, for reducing the hardware requirement, Extended Min-Sum (EMS) [4] decoding algorithm is used where instead of multiplication operation, addition operation is utilized with a sorting algorithm.

In the decoding process of non binary LDPC codes, there are mainly four steps involved. They are initialization, check node processing, variable node processing, and decision making. These steps are repeated again until a valid codeword is found in decoding [4]. The whole decoding process is

based upon the tanner graph which is constructed for the codes on the basis of the parity check equations which defines the code. A tanner graph has two types of node i.e. check node and variable node. There are also connections present between the two types of nodes. The check node represents the rows of the parity check matrix used to define the codes while a variable node represents the columns.

So, there are two important phases of NB-LDPC code decoders. They are Check node processing and Variable node processing. While decoding the codes, the processing of rows and columns of the parity check matrix is carried out. If all the parity check equations are found to be satisfied then only a received bit is called as correct. So, the processing of rows of parity check matrix with the received bits is called check node processing and the processing of columns of parity check matrix is called variable node processing. Check node processing is very much complex than variable node processing when compared in terms of hardware required to implement them on the hardware. This restricts there use in hardware limited applications. So, there is a need to reduce this hardware need for efficiently implementing the decoder for these codes.

2.2 Check Node Processing

As previously mentioned, the processing of rows of parity check matrix with the received bits is termed as check node processing. The belief propagation algorithm as used for decoding binary LDPC codes can be used to decode NB-LDPC codes. But here, the check node processing part will also calculate the sum of products of probabilities. In the calculations of the non-binary variant of these codes, check node processing can be considered as the convolutions of the received messages. The belief propagation decoding algorithm used for NB-LDPC codes has many disadvantages like it needs multiplication operation for performing the decoding which is expensive to be implemented on the hardware or it needs to calculate Fourier transforms in log domain which is harder to implement, etc. So, extended min-sum algorithms and the Min-max algorithm were developed which are in the log domain. The hardware complexity was highly reduced because only the addition operation is required in there check node processing [5]. In EMS and min-max algorithms, Log-Likelihood Ratios are defined that are associated with each symbol. These LLRs are defined with respect to the most likely finite field element. So, according to this, all the LLRs are always positive, and the LLRs for the most likely field element is always zero in each and every vector. Those LLRs having smaller values have more chances for the received symbol to be equal to the respective field element.

The LLRs in the array is then sorted and transferred to the resultant output array. It can be noted that due to the sum operation performed, the next smallest entry may not be always the neighbor of the current smallest entry [6],[7],[8]. That's why some sorting needs to be implemented. When a new smallest value is found then its respective field elements are compared with those of previous entries. If a field element is found to be different then only the next smallest entry is swapped to the output array. For sorting, the L-Bubble check [8] algorithm is used.

2.3 L-Bubble Check Algorithm

The L-bubble check algorithm [8] can be described as follows: -

1. Read U(i') and V(j') from memories U and V.
2. Compute $T_{\Sigma}(i',j')$ = U(i')+V(j'), this value becomes the **ind**th bubble (position of the bubble extracted in the preceding cycle) and the corresponding register is thus bypassed.
3. Determine the minimum value in the sorter and its associated index @**ind** (min operator).
4. From @**ind**, update the address of the ith bubble and store it for the next cycle. The replacing rule is:
    a) if @**ind** = 0 or 1, then j' = j + 1
    b) elsif (@**ind** = 2 & j = 1) then j' = 2
    c) else i' = i + 1

Here, U(i') and V(j') are the two input vectors or arrays which will contain the input pre sorted LLR values. $T_{\Sigma}(i',j')$ is a two dimensional array which will contain the bitwise sum of U(i') and V(j'). @**ind** is used as a flag for replacing new values in the bubbles.

2.4 High Level Synthesis

High level synthesis or HLS which is also called as C synthesis or behavioral synthesis or algorithmic synthesis. It is a process where it converts a description of an algorithm into an RTL design which is a digital hardware The description of the algorithm is given as input in the form of code in any of the high level languages like ANCI C/ C++/ System C/ MATLAB, etc [12].

The code is first analyzed, architecturally constrained, and then scheduled to transcompiled into a register-transfer level (RTL) design in a hardware description language (HDL), which is generally synthesized into the gate level by the use of a logic synthesis tool. Here the user needs not to

worry about hardware description language [12]. Instead of it, the user just needs to know any of the above-mentioned high-level languages. This makes the task easier as now the designer only needs to concentrate on just the description part.

The tool will automatically construct an RTL design that has the same functionality as the input description of the algorithm has. The main focus of HLS allows the user to focus only on the description part and the optimization part and rest are handled automatically by the tool [12]. The tool also tries to optimize the design in an optimum way to exactly meet the timing constraints. If the user wants to optimize specific parameters according to the demand then this can also be done by applying the directives. The user has to apply certainly required directive which forces the tool to optimize the hardware according to the user's requirement. High level synthesis works as a bridge between the hardware and software [9].

When an algorithm is high level synthesized than the HLS tool follows a course of action. As the algorithm's description is written in the high level language, it has to be converted into hardware language or hardware operation and that is done by the HLS tool. The decisions like when an operation will be executed, when an operation will be ended, what will be the sequence of operation to achieve the desired functionality, etc. these all things are taken care of by the tool itself [10].

The overall process of High level synthesis is divided into three phases [11],[12]:

1. Scheduling
2. Binding
3. Control logic extraction

The whole description of the algorithm should be present in the top-level function as this file is the one which is converted into the hardware design by the HLS tool. Only a single function can be the top-level function. The test bench is also a very important part of the whole High level synthesis process as verification is always an important part of any design [11]. The test bench calls the top-level function with some test inputs and verifies the generated output with the test output i.e. the original output which represents the original functionality of the algorithm. The tool itself verifies the functionality of the top-level function and the created hardware with the help of a test bench. By default, the tool is designed such that if the test bench returns non-zero value then the hardware has failed in the verification phase and if it returns a zero value then only the hardware has passed the verification [12].

3. IMPLEMENTATION OF L-BUBBLE CHECK ALGORITHM USING HIGH LEVEL SYNTHESIS

3.1 Methodology used for implementation

The methodology adopted for the high level synthesis of the L-bubble check algorithm is shown in Fig. 1 as follows:

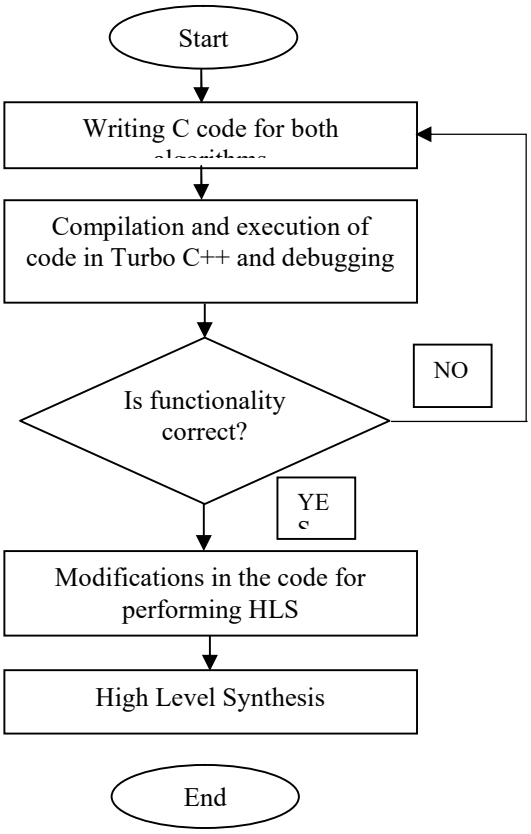

Figure 1 : Methodology for the high level synthesis of L-bubble check algorithm

The tool used for the high level synthesis of the L-bubble check algorithm is Vivado HLS 2017.4. The code for the L-bubble check algorithm is written in C language and then simulated with the same tool. High level synthesis is performed and an RTL is generated after verification using a test bench. The RTL is exported from Vivado HLS 2017.4 tool.

4. SYNTHESIS RESULTS

After performing the high level synthesis, the C code for the L-bubble check algorithm is converted into an RTL design [12]. The C simulation result for the simulation of the C code for the L-bubble check algorithm is shown in Fig. 2.

Then the high level synthesis is performed for the C code. The timing result generated is shown in table 1.

```
   Generating csim.exe
The first array input for L-Bubble check algorithm is: -
0 5 9 14 19 22 27 29 34 38 41 49

The second array input for L-Bubble check algorithm is: -
0 1 8 12 18 21 25 30 33 39 45 51

The bitwise sum array, T is: -
0 5 9 14 19 22 27 29 34 38 41 49
1 6 10 15 20 23 28 30 35 39 42 50
8 13 17 22 27 30 35 37 42 46 49 57
12 17 21 26 31 34 39 41 46 50 53 61
18 23 27 32 37 40 45 47 52 56 59 67
21 26 30 35 40 43 48 50 55 59 62 70
25 30 34 39 44 47 52 54 59 63 66 74
30 35 39 44 49 52 57 59 64 68 71 79
33 38 42 47 52 55 60 62 67 71 74 82
39 44 48 53 58 61 66 68 73 77 80 88
45 50 54 59 64 67 72 74 79 83 86 94
51 56 60 65 70 73 78 80 85 89 92 100

The sorted output array is: -
0 1 5 6 8 9 10 12 13 14 15 17 18 19 20 21 22 23 23 25 26 27 28 29
Test Passed!!!
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Figure 2 : C simulation result for L-bubble check algorithm

Table 1: Timing result after performing High level synthesis

Clock	Target	Estimate	Uncertainty
ap_clk	10.00 ns	8.13 ns	1.25 ns

The utilization estimate result is shown in table 2.

Table 2 : Utilization estimate for L-bubble check algorithm

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	997
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	1	-	34	10
Multiplexer	-	-	-	438
Register	-	-	604	-
Total	1	0	638	1445
Available	270	240	126800	63400
Utilization (%)	~0	0	~0	2

The resource usage results and timing results generated after high level synthesis is shown in table 3 and table 4, respectively.

5. CONCLUSION

In this paper, the L-bubble check algorithm which is used in check node processing for sorting out optimum LLR values is synthesized using high level synthesis. As this algorithm has never been implemented using high level synthesis so it the novelty of this work. For this the C code for the algorithm is written and then high level synthesis is carried out. The resource usage result shows that the number of slices used here is 204. The results in paper [8] shows the number of slices required to implement L-bubble check algorithm is 498, when implemented using writing VHDL code. So, there is a reduction in the number of slices of around 59%. So, it is concluded that when the algorithm is implemented using high level synthesis, there a significant reduction in the hardware requirement.

Table 3 : Resource usage for L-bubble check algorithm synthesized using high level synthesis

Resource usage	VHDL
SLICE	204
LUT	559
FF	550
DSP	0
BRAM	1
SRL	0

Table 4: Timing summary for L-bubble check algorithm synthesized using high level synthesis

Timing summary	VHDL
CP required	10.000
CP achieved post-synthesis	8.476
CP achieved post-implementation	9.213

*(CP is Clock Pulse)

REFERENCES

[1] R. Gallager, “Low-density parity-check codes”, IRE Transactions on Information Theory, vol. 8, pp.21-28, 1962.

[2] D.J.C. MacKay,” Good error-correcting codes based on very sparse matrices”, IEEE Transactions on Information Theory, vol. 45, pp.399-431, 1999.

[3] P. Venkateshwari and M. Anbuselvi, “Decoding performance of binary and non-binary LDPC codes for IEEE 802.11n standard”, International Conference on Recent Trends in Information Technology, pp.292-296, 2012.

[4] Kai He, Jin Sha, and Zhongfeng Wang, “Nonbinary LDPC code decoder architecture with efficient check node processing”, IEEE transactions on circuits and systems-II: Express briefs, pp.381-385, 2012.

[5] Leixin Zhou, Jin Sha, and Zhongfeng Wang, “Efficient EMS decoding for non-binary LDPC codes”, 2012 international Soc design conference (ISOCC), pp.339-342, 2012.

[6] Xiao Ma, Kai Zhang, Haiqiang Chen and Baoming Bai, “Low complexity X-EMS algorithms for nonbinary LDPC codes”, IEEE transactions on communication, vol.60, pp.9-13, 2012.

[7] Oussama Abassi, Laura Conde-Canencia, Ali Al Ghouwayel, and Emmanuel Boutillon, “A novel architecture for elementary check node processing in

non-binary LDPC decoders", IEEE transactions on circuits and systems II: Express briefs, pp.136-140, 2017.

[8] E. Boutillon and L. Conde-Canencia, "Simplified check node processing in nonbinary LDPC decoders", International Symposium on Turbo Codes & Iterative Information Processing, pp.201-205, 2010.

[9] Michael C. McFarland, Alice C. Parker and Raul Camposano, "Tutorial on high-level synthesis", IEEE design automation conference, pp.331-336, 1988.

[10] Dan Gajaski, Todd Austin and Steve Svoboda, "What input-language is the best choice for high level synthesis (HLS)?", Design automation conference, pp. 857-858, 2010.

[11] Philippe Coussy and Adam Morawiec, "High-Level Synthesis: from algorithm to digital circuit", Published by Springer Science & Business Media, ISBN-978-1-4020-8588-8, year 2008.

[12] "Vivado design suite user guide: High-Level Synthesis (UG902)", published by Xilinx Inc., year 2017.