Bipartite edge coloring approach for designing parallel hardware interleaver architecture

Sani Awais Hussein

N° d'ordre : 261

THESE

pour obtenir le grade de :

DOCTEUR DE L'UNIVERSITE DE BRETAGNE SUD

Spécialité : Sciences de l'ingénieur
Mention : Electronique et Informatique Industrielle

AWAIS HUSSAIN SANI

Bipartite edge coloring approach for designing parallel hardware interleaver architecture

Soutenue publiquement le 11 mai 2012

COMPOSITION DU JURY

E. Casseau	Rennes 1 / ENSSAT	Rapporteur
C. Jego	ENSEIRB	Rapporteur
J-L. Danger	Telecom ParisTech	Examinateur
D. Gnaedig	Turboconcept	Examinateur
F. Kienle	Technical University of Kaiserslautern	Examinateur
Cyrille Chavet	Université de Bretagne Sud	Encadrant
Philippe Coussy	Université de Bretagne Sud	Co-directeur de Thèse
Eric Martin	Recteur Académie de Besançon	Direction de Thèse

Laboratoire en sciences et techniques de l'information,
de la communication et de la connaissance (Lab-STICC)
Université de Bretagne Sud

Acknowledgment

First of all, i want to thank my Ph.d. supervisors Philippe COUSSY and Cyrille CHAVET for providing me a change to work on this thesis and for guiding me to complete it. Looking back the three years in my PhD program, now I can actually sense the beauty behind the research of new approaches. I gratefully acknowledge Philippe and Cyrille for their advice, supervision, and crucial contributions, which made them a backbone of this thesis work. I have learnt a lot both professionally and personally from their involvement in my PhD, their originality has triggered and nourished my intellectual maturity which I will benefit from for a long time to come. I express all my gratitude to prof. Christophe JEGO and prof. Emmanuel CASSEAU for honoring us by accepting the difficult task of being reviewer of this thesis.

I would also like to thank my family especially my mother who inspired and supported me all these years of my life, my wife for being a constant source of support throughput my stay in France. I would also like to acknowledge all my colleagues at Lab-STICC in France for making my PhD a enjoyable experience. In particular, I'd like to thank Saeed ur REHMAN at Lab-STICC for helping me during my research on LDPC codes. Lot of thanks to my pakistani friends who worked in Lorient for making my stay a wonderful experience with their love. Finally i would like to thank Life for providing me with this opportunity to explore myself more and grow as a person.
.

For my family and Friends

Table of Contents

Chapter 1 PROBLEMATIC 3

1. Forward Error Correction (FEC) Coding 5

2. Convolutional Codes 5

2.1. Convolutional Encoder ---- 6
2.2. Convolutional Code state and Trellis Diagram ---- 7
2.3. Decoding Convolutional Codes ---- 9
2.4. Turbo Codes ---- 10
2.4.1. Turbo Encoder ---- 10
2.4.2. Interleaver ---- 11
2.4.3. Turbo Decoder ---- 11
2.4.4. Parallelism in Turbo Codes Decoding ---- 12
2.4.4.1 Recursive Unit Parallelism ---- 13
2.4.4.2 Trellis Level Parallelism ---- 13
2.4.4.3 SISO Decoder Level Parallelism ---- 14
2.4.4.4 Conclusion ---- 15

3. Block Codes 15

3.1. Encoding of Linear Block Codes ---- 16
3.2. Low Density Parity Check (LDPC) Codes ---- 17
3.3. Tanner Graph Representation ---- 18
3.4. Decoding ---- 19
3.5. Implementation of LDPC Decoder ---- 21
3.5.1. Fully Parallel Implementation ---- 21
3.5.2. Fully Serial Implementation ---- 22
3.5.3. Partially Parallel Implementation ---- 22

4. Memory conflict problem 22

4.1. Memory conflict problem for Turbo Codes ---- 23
4.2. Memory conflict problem for LDPC Codes ---- 24

5. Conclusion 26

Chapter 2 STATE OF THE ART 27

1. Introduction ---- 29

2. Approaches to tackles memory conflict problem for turbo and LDPC codes ---- 29

2.1. Conflict Free Interleaving laws ---- 30
2.1.1. Architecture aware Turbo Codes ---- 30
2.1.2. Structured LDPC Codes ---- 32
2.2. Run Time Conflict Resolution ---- 33
2.2.1. Run Time Conflict Resolution for Turbo Codes ---- 33
2.2.2. Run Time Conflict Resolution for LDPC Codes ---- 37
2.3. Design Time Conflict Resolution ---- 38

2.3.1. Design Time Conflict Resolution for Turbo Codes----------38
2.3.2. Design Time Conflict Resolution for LDPC Codes ----------40
2.4. Conclusion ----------41
3. Node and Edge Coloring of Graph ----------42
3.1. Graph Theory ----------42
3.2. Conflict Graph----------44
3.2.1. Conflict Graph for Turbo Codes----------44
3.2.2. Conflict Graph for LDPC Codes ----------45
3.3. Bipartite Graph ----------46
3.3.1. Bipartite Edge Coloring----------47
3.3.1.1 Vizing Method to color the edges of Bipartite Graph ----------47
3.3.1.2 Gabow Method to color the edges of Bipartite Graph----------48
4. Transportation Problem ----------51
5. Conclusion ----------53

Chapter 3 METHODS BASED ON BIPARTITE GRAPH FOR SOLVING MEMORY MAPPING PROBLEMS 55

1. Introduction 57
2. A Methodology based on Transportation Problem Modeling for Designing Parallel Interleaver Architecture 57
2.1. Problem Formulation for Memory Mapping Problem of Turbo codes----------58
2.2. Modeling ----------59
2.2.1. Construction of Bipartite Graph ----------59
2.2.2. Transformation of bipartite graph into Transportation Matrix ----------60
2.3. Algorithm to find semi 2-factors in Turbo Bipartite Graph----------61
2.4. Pedagogical Example to explain algorithm ----------63
3. Methodologies Based on Bipartite Graph to find conflict Free Memory Mapping for LDPC ----------66
3.1. Problem Formulation for Memory Mapping Problem of LDPC codes ----------66
3.2. Modeling ----------67
3.3. Algorithm ----------68
3.3.1. Partition the Bipartite Graph ----------69
3.3.2. Coloring edges of Bipartite Graph ----------70
3.4. Pedagogical Example to explain algorithm ----------71
4. Conclusion ----------75

Chapter 4 METHODS BASED ON TRIPARTITE GRAPH FOR SOLVING MEMORY MAPPING PROBLEMS 77

1. Introduction 79
2. Methodology Based on Tripartite Graph to find conflict Free Memory Mapping for LDPC 79
2.1. Modeling ----------80

2.2. 2-Step Coloring Approach ---------- 82
2.3. Pedagogical Example to explain algorithm ---------- 84
3. Constructing Bipartite Graph for Turbo and LDPC Codes 87
3.1. Construction of Bipartite Graph for Mapping Problem of Turbo Codes ---------- 88
3.2. Construction of Bipartite Graph for Mapping Problem of LDPC Codes ---------- 89
3.3. Bipartite Edge Coloring Algorithm ---------- 90
3.4. Pedagogical Example to explain algorithm ---------- 93
4. Complexity comparison of Different algorithms 96
5. Conclusion 97

Chapter 5 EXPERIMENTS 99

1. Introduction 101
2. Design Flow for Memory Mapping Tool 102
3. Ultra Wide Band Communication System 103
3.1. Bit Interleaver used in WPAN IEEE 802.15.3a Physical Layer ---------- 103
3.2. Experiments and Results ---------- 106
4. Designing Parallel Interleaver Architecture for Turbo Decoder 107
4.1. Interleaver used in HSPA Evolution ---------- 107
5. Designing Partially Parallel Architecture for LDPC Decoder 112
5.1. Partially Parallel Architecture for structured LDPC codes ---------- 112
5.2. Decoder Architecture for Non-Binary LDPC codes ---------- 116
6. Case Study: Designing Parallel architecture for Quadratic Permutation Polynomial Interleaver 119
6.1. Configurations used in this study ---------- 120
6.2. Experiments and Results ---------- 121
7. Conclusion 124

Conclusion and Future Perspectives 125

Bibliography 129

Annexure 135

List of Figures

Figure 1. 1. Communication System 5
Figure 1. 2. Convolutional Encoder 7
Figure 1. 3. State Diagram 8
Figure 1. 4. Trellis Diagram 9
Figure 1. 5. Turbo Encoder 11
Figure 1. 6. Turbo Decoder 12
Figure 1. 7. Serial Implementation of Turbo Decoder 12
Figure 1. 8. Recursive Unit Parallelism 13
Figure 1. 9. Trellis Level Parallelism 14
Figure 1. 10. SISO Decoder Level Parallelism 14
Figure 1. 11. Systematic format of a codeword 16
Figure 1. 12. Encoder circuit for the (7, 4) Systematic code 17
Figure 1. 13. Tanner Graph representation of H 19
Figure 1. 14. Partially Parallel Architecture 23
Figure 1. 15. Parallel Processing of Turbo Codes 24
Figure 1. 16. Memory Conflict Problem in Parallel Turbo Decoder 24
Figure 1. 17. Memory Conflict Problem in Partially Parallel LDPC Decoder 25
Figure 2. 1. Temporal and Spatial Permutation for interleaver construction 30
Figure 2. 2. H matrix Representation 32
Figure 2. 3. H_{Base} Matrix for WiMax Standard of code word size = 576, Z = 24 and r = 1/2 33
Figure 2. 4. LLR Distributor Architecture 34
Figure 2. 5. Heterogeneous Multistage Network 35
Figure 2. 6. Binary de Bruijn graph with 16 nodes 36
Figure 2. 7. 8-Processor de Bruijn network with processors, routers and network interfaces 38
Figure 2. 8. Matrices used in [TAR04] 39
Figure 2. 9. Matrices used in *SAGE* 40
Figure 2. 10. MAP matrix for Multiple Read Multiple write (MRMW) approach 41
Figure 2. 11. Graph Representation 43
Figure 2. 12. Conflict Graph for Turbo Decoder 45
Figure 2. 13. Resultant architecture 45
Figure 2. 14. Conflict Graph for LDPC Decoder 46
Figure 2. 15. Resultant Architecture 46
Figure 2. 16. Graph Representation 47
Figure 2. 17. Vizing Algorithm 48
Figure 2. 18. Constructing Δ-regular graph 51
Figure 2. 19. Network Model of Transportation Problem for 3 producers and 3 consumers 52
Figure 2. 20. Matrix Model of Transportation Problem for 3 producers and 3 consumers 52
Figure 3. 1: Data Access Matrices for Turbo Codes 58
Figure 3. 2: Bipartite Graph representation 59
Figure 3. 3: Matrix Model for Transportation Problem of Figure 3. 2.a 61
Figure 3. 4: partitioning algorithm 62
Figure 3. 5: Assignment of memory banks in Transportation matrix 63
Figure 3. 6: Assignment of memory banks in Transportation matrix 64

Figure 3. 7: Assignment of memory banks in Transportation matrix 64
Figure 3. 8: Resultant Memory Mapping 65
Figure 3. 9: Data Access Matrices for LDPC Codes 67
Figure 3. 10: Bipartite Representation of Data Access Matrix of Figure 3. 9.a 68
Figure 3. 11: Representation of edges on data node *d* 68
Figure 3. 12: Partitioning Algorithm 70
Figure 3. 13: Coloring Algorithm 71
Figure 3. 14: Path construction through Partitioning Algorithm 72
Figure 3. 15: Path construction through Partitioning Algorithm 73
Figure 3. 16: Conflict free edge coloring of sg_1 74
Figure 3. 17: Conflict free edge coloring of sg_1 74
Figure 3. 18: Conflict free edge coloring of *G* and corresponding mapping 74
Figure 4. 1: Tripartite graph for mapping matrix of Figure 3.9 80
Figure 4. 2: Single Node Representation on Tripartite graph 81
Figure 4. 3: Partitioning Algorithm 83
Figure 4. 4: Coloring Algorithm 84
Figure 4. 5: Path construction through Partitioning Algorithm 85
Figure 4. 6: Path construction through Partitioning Algorithm 86
Figure 4. 7: Conflict Free Edge Coloring of Par_1 86
Figure 4. 8: Conflict Free Edge Coloring of Par_1 and Par_2 87
Figure 4. 9: Conflict free edge coloring of *G* and corresponding mapping matrix 87
Figure 4. 10: Bipartite Graph representation 88
Figure 4. 11: Tripartite graph for mapping matrix of Figure 3.9 89
Figure 4. 12: Bipartite graph for mapping matrix of Figure 3.2.b 90
Figure 4. 13: Matching Algorithm 93
Figure 4. 14: Matching Algorithm 93
Figure 4. 15: Matching Algorithm 94
Figure 4. 16: Euler Partitioning 95
Figure 4. 17: Edge Coloring of Bipartite Graph 95
Figure 5. 1. Design Flow for performing experiments 102
Figure 5. 2. Resultant generated architecture 103
Figure 5. 3. Multiband OFDM System [BAT04] (Copyright @ 2004 IEEE) 104
Figure 5. 4. Input and Interleaved order for 30 bits 105
Figure 5. 5. Implementation of Bit Interleaver 105
Figure 5. 6. Comparison of two different architectures 107
Figure 5. 7. Arrangement of $K = 44$ data into 5*10 matrix 109
Figure 5. 8. Matrix after Intra-row Permutation 110
Figure 5. 9. Matrix after Inter-row Permutation 110
Figure 5. 10. Comparison of CPU time for various Mapping Approaches 111
Figure 5. 11. H_{Base} Matrix for WiMAX Standard of code word size = 576, Z = 24 and r = 1/2 113
Figure 5. 12. Partially parallel architecture for ½ Rate WiMAX standtard 113
Figure 5. 13. Data Access Matrix for ½ Rate WiMAX standtard 114
Figure 5. 14. Resultant Mapping for data access matrix of Figure 5. 12 114
Figure 5. 15. Comparison of CPU time for various Mapping Approaches 115
Figure 5. 16. Architecture for NB-LDPC 117
Figure 5. 17. Data Access Matrix for D = 192 and d_c = 6 using Serial Architecture 117

Figure 5. 18.Data Access Matrix for D = 192 and d_c = 6 using Partially Parallel Architecture..... 118
Figure 5. 19. Comparison of CPU time for various Mapping Approaches 119
Figure 5. 20. Decoding Architecture for Turbo Decoders..... 120
Figure 5. 21. Scheduling for Turbo Decoding..... 121
Figure 5. 22. Latency and cost Analysis for all configurations..... 123

List of Tables

Table 1. 1: State Transition Table 8
Table 1. 2: Linear block code with $x = 4$ and $c = 7$. 16
Table 4. 1: Edge Coloring Algorithm 91
Table 4. 2: Perfect Matching Algorithm from [SCH98] 91
Table 4. 3: Euler Partitioning Algorithm from [GAB76] 92
Table 4. 4: Complexity Comparison of approaches used in this thesis 96
Table 5. 1. CPU time (second) for various Memory Mapping Approaches 106
Table 5. 2. Resultant area of different components for parallel implementation of bit interleaver 106
Table 5. 3. List of prime number p and associated primitive root v 108
Table 5. 4. CPU time (seconds) for various Memory Mapping Approaches 111
Table 5. 5. Resultant area (in number of Nand-gates) of different components for different types 112
Table 5. 6. CPU time (seconds) for various Memory Mapping Approaches 114
Table 5. 7. Resultant area of different components for structured LDPC Decoder Architecture 115
Table 5. 8. CPU time for various Memory Mapping Approaches 118
Table 5. 9. Resultant area of different components for NB-LDPC Decoder Architecture 119
Table 5. 10. Different configuration to explore the design space for turbo decoding 121
Table 5. 11. Comparison of Configuration 1 and 4 for latency 122
Table 5. 12. Comparison of all the configurations for hardware cost and latency 122
Table 5. 13. Resultant area for different configurations used in case study 123

INTRODUCTION

Nowadays, Turbo and LDPC codes are two families of codes that are extensively used in current communication standards due to their excellent error correction capabilities. However, hardware design of coders and decoders for high data rate applications is not a straightforward process. For high data rates, decoders are implemented on parallel architectures in which more than one processing elements decode the received data. To achieve high memory bandwidth, the main memory is divided into smaller memory banks so that multiple data values can be fetched from or stored to memory concurrently. However, due to scrambling caused by interleaving law, this parallelization results in communication or memory access conflicts which occur when multiple data values are fetched from or stored in the same memory bank at the same time. This problem is called *Memory conflict Problem.* It increases latency of memory accesses due to the presence of conflict management mechanisms in communication network and unfortunately decreases system throughput while augmenting system cost.

To tackle the memory conflict problems, three different types of approaches are used in literature. In first type of approaches, different algorithms to construct conflict free interleaving law are proposed. The main reason to develop these techniques is to construct "architecture friendly" codes with good error correction capabilities in order to reduce hardware cost. However, architectural constraints applied during code design may impede error correction performance of the codes. In a second type of approaches, different design innovations are introduced to tackle memory conflict problem. Flexible and scalable interconnection network with sufficient path diversity and additional storing elements are introduced to handle memory conflicts. However, flexible networks require large silicon area and cost. In addition, delay introduced due to conflict management mechanisms degrades the maximum throughput and makes these approaches inefficient for high data rate and low power applications. In third type of approaches deals with algorithms that assign data in memory in such a manner that all the processing elements can access memory banks concurrently without any conflict. The benefit of this technique is that decoder implementation does not need any specific network and extra storage elements to support particular interleaving law. However, till now no algorithm exists that can solve memory mapping problem for both turbo and LDPC codes in polynomial time.

The work presented in this thesis belongs to the last type of approaches. We propose several methods based on graph theory to solve memory mapping problem for both turbo and LDPC codes. Different formal models based on bipartite and tripartite graphs along with different algorithms to color the edges of these graphs are detailed. The complete path we followed before it is possible to solve mapping problem in polynomial time is hence presented. For the first two approaches, mapping problem is modeled as bipartite graph and then each graph is divided into different sub-graphs in order to facilitate the coloring of the edges. First approach deals with Turbo codes and uses transportation problem algorithms to divide and color the bipartite graph. It can find memory mapping that supports particular interconnection network if the interleaving rule of the application allows it. Second approach solves memory mapping problem for LDPC codes using two different complex algorithms to partition and color each partition. In the third algorithm, each time instance and edge is divided into two parts to model our problem as tripartite graph. Tripartite

graph is partitioned into different sub-graphs by using an algorithm based on divide and conquer strategy. Then each subgraph is colored individually by using a simple algorithm to find a conflict free memory mapping for both Turbo and LDPC codes. Finally, in the last approach tripartite graph is transformed into bipartite graph on which coloring algorithm based on Euler partitioning principle is applied to find memory mapping in polynomial time.

Several experiments have been performed using interleaving laws coming from different communication standards to show the interest of the proposed mapping methods. All the experiments have been done by using a software tool we developed. This tool first finds conflict free memory mapping and then generates VHDL files that can be synthesized to design complete architecture i.e. network, memory banks and associated controllers. In first experiment, bit interleaver used in Ultra Wide Band (UWB) interleaver is considered and a barrel shifter is used as constraint to design the interconnection network. Results are compared regarding area and runtime with state of the art solutions. In second experiments, a turbo interleaving law defined in High Speed Packet Access (HSPA) standard is used as test case. Memory mapping problems have been solved and associated architectures have been generated for this interleaving law which is not conflict free for any type of parallelism used in turbo decoding. Results are compared with techniques used in state of the art in terms of runtime and area. Third experiment focuses on LDPC. First, last algorithm we proposed is used to find conflict free memory mapping for non-binary LDPC codes defined in the DaVinci Codes FP7 ICT European project. Then, conflict free memory mapping have also been found for partially parallel architecture of LDPC codes used in WiMAX and WiFi for different level of parallelism. It is shown that the proposed algorithm can be used to map data in memory banks for any structured codes used in current and future standards for partially parallel architecture. In last experiment, thanks to the proposed approach we explored the design space of Quadratic Permutation Polynomial (QPP) interleaver that is used in 3GPP-LTE standard. The QPP interleaver is maximum contention-free i.e., for every window size W which is a factor of the interleaver length N, the interleaver is contention free. However, when trellis and recursive units parallelism are also included in each SISO, QPP interleaver is no more contention-free. Results highlight tradeoffs between area and performances based on for different radixes, parallelisms, scheduling (replica versus butterfly)…

Chapter 1

PROBLEMATIC

Table of Contents

1. Forward Error Correction (FEC) Coding 5
2. Convolutional Codes 5
2.1. Convolutional Encoder ---- 6
2.2. Convolutional Code state and Trellis Diagram ---- 7
2.3. Decoding Convolutional Codes ---- 9
2.4. Turbo Codes ---- 10
2.4.1. Turbo Encoder ---- 10
2.4.2. Interleaver ---- 11
2.4.3. Turbo Decoder ---- 11
2.4.4. Parallelism in Turbo Codes Decoding ---- 12
2.4.4.1 Recursive Unit Parallelism ---- 13
2.4.4.2 Trellis Level Parallelism ---- 13
2.4.4.3 SISO Decoder Level Parallelism ---- 14
2.4.4.4 Conclusion ---- 15
3. Block Codes 15
3.1. Encoding of Linear Block Codes ---- 16
3.2. Low Density Parity Check (LDPC) Codes ---- 17
3.3. Tanner Graph Representation ---- 18
3.4. Decoding ---- 19
3.5. Implementation of LDPC Decoder ---- 21
3.5.1. Fully Parallel Implementation ---- 21
3.5.2. Fully Serial Implementation ---- 22
3.5.3. Partially Parallel Implementation ---- 22
4. Memory conflict problem 22
4.1. Memory conflict problem for Turbo Codes ---- 23
4.2. Memory conflict problem for LDPC Codes ---- 24
5. Conclusion 26

In this chapter, error correction coding techniques with particular emphasis on Turbo and LDPC codes are discussed. Error correction coding can be classified into two broad categories: convolutional codes and block codes. Chapter starts by presenting encoding and decoding related to convolution codes. Afterwards, brief description of turbo codes that are a subclass of convolutional codes are presented. In the later part of this chapter, block codes with particular emphasis on LDPC codes are explained. Finally, problems in implementing these algorithms on parallel architecture are introduced to highlight the importance of the work presented in this thesis.

1. Forward Error Correction (FEC) Coding

Early developers of digital communication systems assumed that information could be transmitted through noisy channel with high reliability by increasing the signal to noise ratio. This can only be achieved at that time by increasing transmitted signal power enough to ensure that signal can reliably be transmitted to source. The revolutionary work of Shannon [SHA48] changed this view by proofing that it is possible to send digital data to receiver through noisy channel with high reliability by first encoding digital message with error correction code at transmitter and then subsequently decode it at receiver to generate original message. The function of the encoder is to map X digits message into C digits codeword where $C > X$. The code rate $r = X/C$ defines the redundancy introduced by corresponding error correction code. Encoded message passes through channel which corrupts the message by adding some noise into it. At receiver, error correction decoder uses this added redundancy to determine the original message despite the noise introduced by channel. Typical communication system is shown in Figure 1. 1.

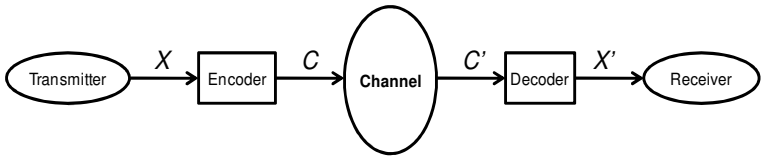


Figure 1. 1. Communication System

Different error correction codes are introduced in literature. They can be classified into two broad categories: *Block codes* and *Convolutional codes*. In block codes, original information sequence is divided into different message blocks and each message is independently encoded to generate codeword bits whereas in convolutional codes, encoder takes information sequence as a continuous stream and generates a continuous stream of codeword bits. Therefore in block codes, encoder must wait for the entire message block before it starts encoding whereas convolutional encoder can start encoding and transmitting codeword before it obtains the entire message.

In current telecommunication standards, two error correcting codes, one from block codes (called Low Density Parity Check (LDPC) codes) and the other from convolutional codes (called Turbo codes) are extensively used due to their excellent error correcting capabilities. However, implementation of decoder for these two codes for high data rate applications is not straightforward. In this thesis, we restrict our attention to the implementation of both of these codes on parallel architecture.

2. Convolutional Codes

Convolutional codes perform like a finite state machine which converts continuous stream of X message bits into continuous stream of C coded bits (where $X > C$). Due to their simple structure and efficiently implementable iterative decoding algorithm, convolutional codes are increasingly used in different telecommunication standards. Currently, convolutional codes are part of standards for mobile communication (HSPA [HSP04], LTE [LTE08]) and digital broadcasting (DVB-SH [DVBS08]).

2.1. Convolutional Encoder

Convolutional encoder consists on modulo-2 adders and shift registers which acts as a memory for past inputs and acts as encoder state. Each shift register contains one or more register element and each register element introduces a delay of one time unit. Due to the presence of these shift registers, output coded bits depend not only on the present message bits but also on the states of encoder i.e., on the already coded message bits.

Definition ***Systematic Encoder***

A convolutional encoder is called *systematic* if it maps all its input bits directly to the output bits, otherwise it is called *non-systematic*.

In Figure 1. 2.b & d, input bit x is directly map to output bit $c^{(1)}$. So, these encoders are example of a systematic encoder.

Definition ***Recursive Encoder***

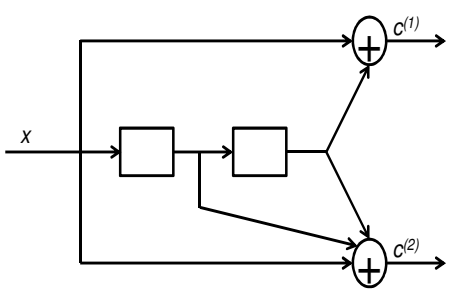
A convolutional encoder is called *recursive* if it's state depends on its output, otherwise it is called *non-recursive*.

Recursive Encoder are shown in Figure 1. 2.c & d.

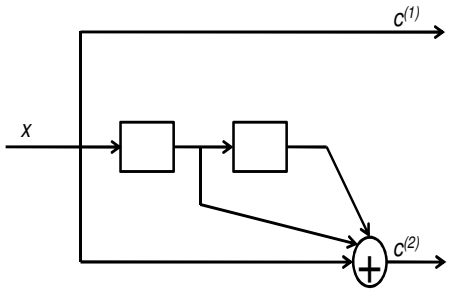
Depending on these definitions, there are four types of convolutional encoders exit:

1. Non-systematic non-recursive convolutional encoder (Figure 1. 2.a)
2. Systematic non-recursive convolutional encoder (Figure 1. 2.b)
3. Non-systematic recursive convolutional encoder (Figure 1. 2.c)
4. Systematic recursive convolutional encoder (Figure 1. 2.d)

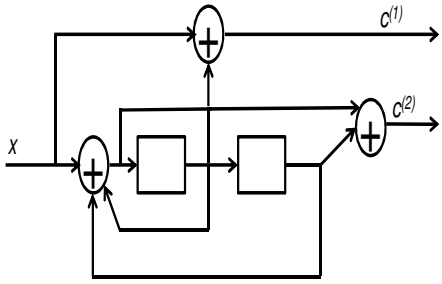
All these encoders are shown in Figure 1. 2.



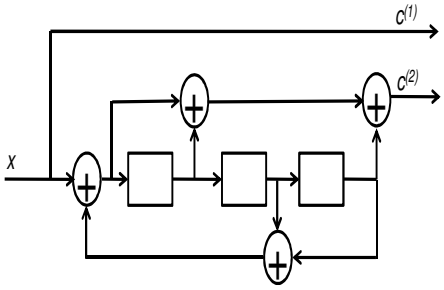
(a) Non-Systematic Non-Recursive Convolutional Encoder



(b) Systematic Non-Recursive Convolutional Encoder



(c) Non-Systematic Recursive Convolutional Encoder



(d) Systematic Recursive Convolutional Encoder

Figure 1. 2. Convolutional Encoder

In all of these figures, shift register elements and modulo-2 adders are represented by square blocks and circles respectively. To explain the terminologies used in convolutional codes, we consider the encoder shown in Figure 1. 2.d. This encoder is a part of current 3GPP LTE standard to encode message bits. At time instance t, encoder takes one message bit x_t and generates two coded bits $c_t^{(1)}$ & $c_t^{(2)}$ results into the code rate of ½. Interleaving of coded bits results into the codeword $\mathbf{c} = [c_1^{(1)}\, c_1^{(2)}, c_2^{(1)}\, c_2^{(2)}, c_3^{(1)}\, c_3^{(2)}, \ldots\ldots\ldots\ldots, c_t^{(1)}\, c_t^{(2)}]$.

For systematic codes, coded bits are further differentiated into *systematic* and *parity bits*. Input message bits are called systematic bits whereas extra output bits which are not systematic are called parity bits. So for the encoder of Figure 1. 2.d, coded bits $\boldsymbol{c}_t^{(1)}$ are replaced by message bits $\boldsymbol{x}^{(1)}$ and coded bits $\boldsymbol{c}_t^{(2)}$ by parity bits $\boldsymbol{p}^{(1)}$ to generate the codeword **c**.

$$\mathbf{c} = [x_1^{(1)}\, p_1^{(1)}, x_2^{(1)}\, p_2^{(1)}, x_3^{(1)}\, p_3^{(1)}, \ldots\ldots\ldots\ldots\ldots\ldots, x_t^{(1)}\, p_t^{(1)}]$$

2.2. Convolutional Code state and Trellis Diagram

Convolutional encoder can be represented as finite-state machine in which the relationship between input, state and output can be explained through state transition table and state diagram. Encoder discussed in previous section consists of three register elements. So the state of encoder, represented by $S = (s^{(1)}, s^{(2)}, s^{(3)})$ where $s^{(1)}, s^{(2)}, s^{(3)} \in \{0,1\}$, is the contents of three register elements from left to right. If v is the number of elements then there are 2^v possible states in which encoder can be at any time instance. So, the eight possible states for encoder with three register elements are, $S_0 = (000), S_1 = (001), S_2 = (010), S_3 = (011), S_4 = (100), S_5 = (101), S_6 = (110), S_7 = (111)$.

For the encoder in discussion, state transition table is shown in Table 1. 1. In this table, S_c and S_n are the current and next states respectively and x represents the input message bit which causes this transition. Also the output coded bits generated as a result of transition from S_c and S_n are labeled as $c_{c,n}^{(1)}$ *(systematic bit)* and $c_{c,n}^{(2)}$ *(parity bit)*.

Table 1. 1: State Transition Table

Current state S_c		Input $x_{c,n}$	Next State S_n		Output $c_{c,n}$	
S_c	$(s_c^{(1)}s_c^{(2)}s_c^{(3)})$	x	S_n	$(s_n^{(1)}s_n^{(2)}s_n^{(3)})$	$c_{c,n}^{(1)}$	$c_{c,n}^{(2)}$
S_0	000	0	S_0	000	0	0
S_0	000	1	S_4	100	1	1
S_1	001	0	S_4	100	0	0
S_1	001	1	S_0	000	1	1
S_2	010	0	S_5	101	0	1
S_2	010	1	S_1	001	1	0
S_3	011	0	S_1	001	0	1
S_3	011	1	S_5	101	1	0
S_4	100	0	S_2	010	0	1
S_4	100	1	S_6	110	1	0
S_5	101	0	S_6	110	0	1
S_5	101	1	S_2	010	1	0
S_6	110	0	S_7	111	0	0
S_6	110	1	S_3	011	1	1
S_7	111	0	S_3	011	0	0
S_7	111	1	S_7	111	1	1

State transition table is represented graphically as a *state diagram* in which node represents one of the eight states and directed edge represent the state transition between nodes as shown in Figure 1. 3. Label on each edge mentions the input bit that generates the state transition and output bits as input/output.

The state diagram completely explains the relationship between state transitions and input/output bits but it does not provide any information about how this relationship evolved with time. *Trellis diagram* gives the required information by expanding state diagram at each time instance as shown in

Figure 1. 4. Two copies of states are represented at time t and $t+1$ and directed edges are drawn from states at t to states at $t+1$ to show the state transitions. Due to trellis diagram, convolutional codes are decoded very efficiently by algorithms which operate on code trellis to find most likely codewor

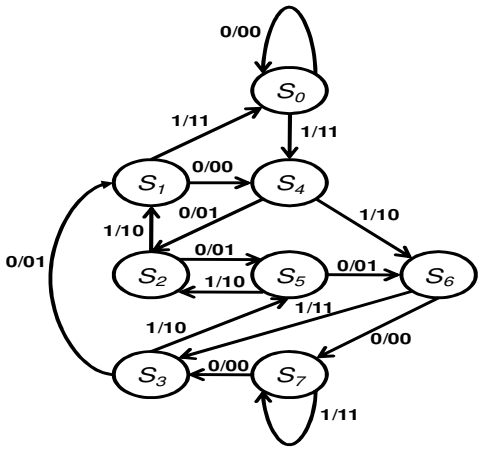


Figure 1. 3. State Diagram

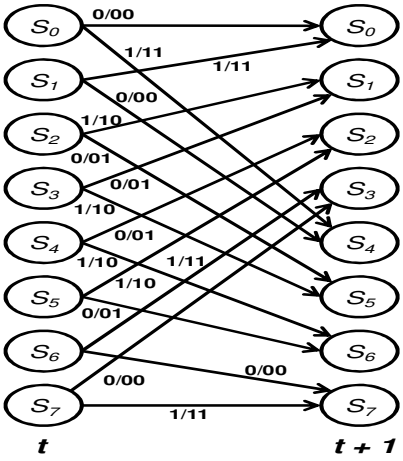


Figure 1. 4. Trellis Diagram

2.3. Decoding Convolutional Codes

As explained in previous section, each state transition from S_c to S_n or trellis edge corresponds to a particular input. So, decoder computes the probability of each state transition to find the maximum probability input bit. An efficient decoding algorithm based on trellis was first presented by Bahl, Cocke, Jelenik and Raviv [BAH74] and is called the *BCJR algorithm*.

Algorithm is based on the concept that codeword bits c_t sent at time t are influenced by the codeword bit c_t^+ sent before it. Thus, they may also affects the codeword bits c_t^- sent after it. So in order to estimate the message bits, the algorithm makes two passes through the trellis: *forward pass* and *backward pass*. *Forward Pass* estimates the message bit x_t at t based on c_t^+ whereas *backward Pass* uses c_t^- to estimate x_t. Suppose

y_t represents the symbols received for the codeword bits c_t sent at time t.

y_t^+ represents the symbols received for the codeword bits c_t^+ sent after time t and

y_t^- represents the symbols received for the codeword bits c_t^- sent before t.

Then, the probability of the state transition from state S_c at time *t-1* to state S_n at time t is given by the following equation,

$$\Gamma_t(S_c, S_n) = A_{t-1}(S_c)\, \Lambda_t(S_c, S_n)\, B_t(S_n) \qquad 1.1$$

The probability is a function of following three terms:

(I) A_{t-1} represents the probability that the encoder is in state S_c at *t-1* based on the information about symbols received before t i.e., y_t^-;

(II) $B_t(S_n)$ represents the probability that the encoder is in state S_n at t based on the information about symbols received before after t i.e., y_t^+;

(III) $\Lambda_t(S_c, S_n)$ represents the probability of the state transition from S_c to S_n based on the information about symbols received at t i.e., y_t;

The calculation of A and B values are respectively called the *forward* and *backward recursion* of the BCJR decoder and are calculated through following equations:

$$A_{t-1}(S_c) = \sum_{i=0}^{2^v-1} A_{t-2}(S_i)\ \Lambda_{t-1}(S_i, S_n) \quad 1.2$$

$$B_t(S_n) = \sum_{i=0}^{2^v-1} B_{t+1}\ (S_i)\ \Lambda_{t+1}(S_c, S_i) \quad 1.3$$

From this equations it is clear that values of both A and B or *path metrics* can be calculated recursively and each step requires multiplication operations over real numbers which increases the decoder complexity in terms of area and cost when implementing in hardware. This problem is overcome by re-formation of original algorithm in the logarithmic domain. The benefit of representing path metrics as log metrics is that now we can replace multiplication with addition. If α,β and γ represents the log metrics of A, B and Λ then previous equations transform into the log domain as follows:

$$\Gamma_t(S_c, S_n) = \alpha_{t-1}(S_c) + \gamma_t(S_c, S_n) + \beta_t(S_n) \quad 1.4$$

$$\alpha_{t-1}(S_c) = \log \sum_i e^{\alpha_{t-2}(Si) + \gamma_{t-1}(Si,\ Sn)} \quad 1.5$$

$$\beta_t(S_n) = \log \sum_i e^{\beta_{t+1}(Si) + \gamma_{t-1}(Sn,\ Si)} \quad 1.6$$

The derivation of these equations is out of the scope of this thesis. Interested reader can consult [JOH10] to understand the derivation and respected terms.

2.4. Turbo Codes

Excellent error correction capabilities of Turbo codes [BER93] make it integral part of current telecommunication standards such as [LTE08] [HSP04] [DVBS08]. Turbo code completely changes the way we perform error correction to reach near Shannon limit of channels capacity. Turbo codes are constructed through the parallel concatenation of two convolutional codes which during decoding share their information to achieve good error correction performance. This outstanding performance is also possible due to the presence of pseudo-random interleaver that scrambles data to break up neighborhood relations. Moreover, low-complexity iterative decoding algorithm makes its implementation feasible at the hardware level to be included in the current standards.

2.4.1. Turbo Encoder

Turbo encoder consists of two convolution encoder in which first component encodes the message x in natural (original) order to produce parity bits $p^{(1)}$ whereas second one encodes the message in interleaved order (after passing the original message through interleaver) to generate parity bits $p^{(2)}$ as shown in Figure 1. 5. Since Turbo codes are systematic codes, so at the output, message and parallel concatenation of parity bits are all transmitted to construct turbo codeword. Normally, two component encoders used in Turbo codes are identical but it is also possible to use different components encoders. However, due to presence of interleaver, the parity bits output by two encoders are always different even if identical encoders are used in turbo codes.

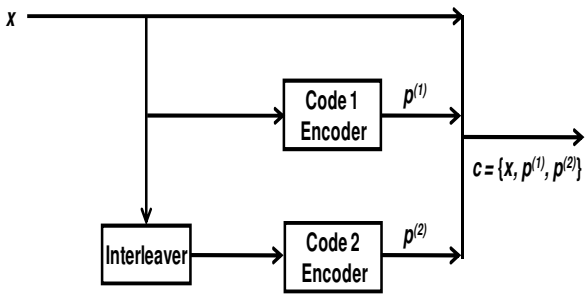


Figure 1. 5. Turbo Encoder

2.4.2. Interleaver

An interleaver is expressed through a permutation sequence $\Pi = \{\pi_1, \pi_2, \pi_3, \ldots\ldots, \pi_n\}$, where the sequence $\{\pi_1, \pi_2, \pi_3, \ldots\ldots, \pi_n\}$ represents the permutation of the integers from 1 to n.

The function of the interleaver is to generate two completely different set of parity bits after passing through two constituent encoders in order to obtain capacity-approaching performance of turbo codes. This performance is normally achieved by using interleaver with a length of several thousand bits and that performs random permutation on the required length.

2.4.3. Turbo Decoder

In turbo decoding, the decoding of each individual convolutional code is carried out using BCJR algorithm with the following two modifications [BER93]: First of all, in turbo decoding, the two component codes share information about the message bits. This extra information is called *extrinsic information*. Each decoder fed this extrinsic information to other decoder in order to estimate the message bits. Secondly, in turbo decoding, extrinsic information is updated and shared between the decoders over many iterations. As a result, BCJR decoding algorithm is used by each decoder more than once to decode the message bits. The block diagram for turbo decoder is shown in Figure 1. 6.

The decoder receives input values $Y^{(u)}$, $Y^{(1)}$, $Y^{(2)}$ from the channel for x, $p^{(1)}$, $p^{(2)}$ respectively. One complete iteration of turbo decoder is carried out through two half iterations. In the first half iteration, Decoder 1 receives channel values for message bit $Y^{(u)}$, first parity bit $Y^{(1)}$ and deinterleaved extrinsic value from Decoder 2 to generate extrinsic value. However, for initial iteration, Decoder 1 has no extrinsic value from Decoder 2, so it uses only $Y^{(u)}$, $Y^{(1)}$ to produce extrinsic value. During second half iteration, Decoder 2 creates extrinsic value from interleaved message bits, second parity bit $Y^{(2)}$ and interleaved extrinsic value from Decoder 1. After the fixed number of iterations, a final decision about the message bits is made based on the extrinsic values from two decoders and channel values for message bits. It is important to note that only extrinsic values are updated at each iteration, channel values for message bits always remain unchanged.

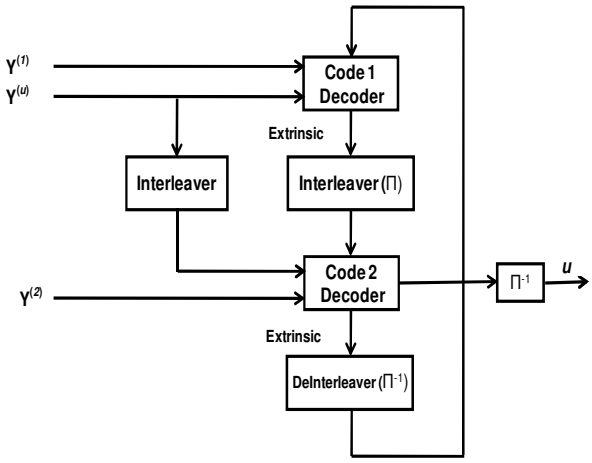


Figure 1. 6. Turbo Decoder

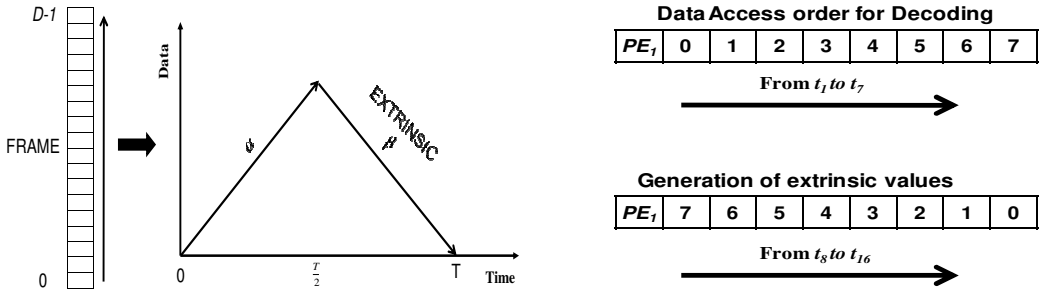
2.4.4. Parallelism in Turbo Codes Decoding

As explained in previous sections 2.3, to decode each component code, turbo decoder first calculates α-values (using forward recursion) and β-values (through backward recursion) in order to generate extrinsic values. This approach is called serial implementation of turbo decoder. Serial implementation can be represented through *Forward Backward Scheme* as shown in Figure 1. 7.a. Implementation of this scheme can best be explained through data access order shown in Figure 1. 7.b. for *D = number of data elements used in a code = 8*, *T = time to decode a code = 2D = 16* and *PE = number of processing elements = 1*. For first *T/2 = 8* time instances, decoder accesses data elements from extrinsic memory from D = 0 to D = 7, calculate α-values and store them into decoder inside decoder memory. For next *T/2* time instances from *9 to 16*, decoder calculates β and extrinsic values and writes updated extrinsic values into extrinsic memory to complete the decoding of component code. It is important to note that decoder can process only one data elements at each time instance in serial implementation.

Serial implementation has following two drawbacks.

1. The calculation of *path metrics* (α and β-values) requires that the whole transmitted data has to be received and stored which increases the decoder memory.
2. The approach significantly increases the latency (i.e., time to compute extrinsic values) and make it unsuitable for high data rate application.

To tackle these drawbacks, different parallelism approaches are proposed which are explained below.



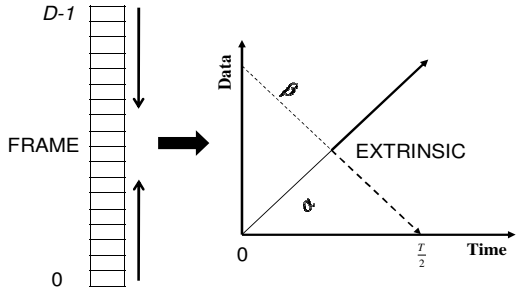
(a) Forward Backward Scheme (b) Data access order in Serial Implementation

Figure 1. 7. Serial Implementation of Turbo Decoder

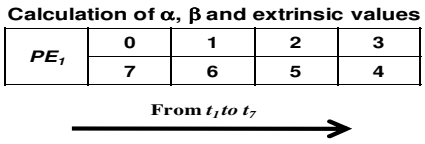
2.4.4.1 Recursive Unit Parallelism

The first type of parallelism [ZHA04] can be directly extracted from decoding algorithm by calculating in parallel α, β and extrinsic values used in BCJR algorithms. This approach can be best explained through *Butterfly Scheme* as shown in Figure 1. 8.a. In this scheme, decoder calculates *path metrics* (α and β-values) at the same time. As a result, butterfly scheme increases the parallelism level by treating two data elements at the same time and requires half the time to decode the component code. Data access order for *D = 8*, *PE = 1* is shown in Figure 1. 8.b.

This scheme parallelizes the forward and backward recursive units to double the parallelism degree without any increase in decoder memory. Also the increase in decoder throughput motivates the use of butterfly scheme at the cost of duplication of recursive units.



(a) Butterfly Scheme



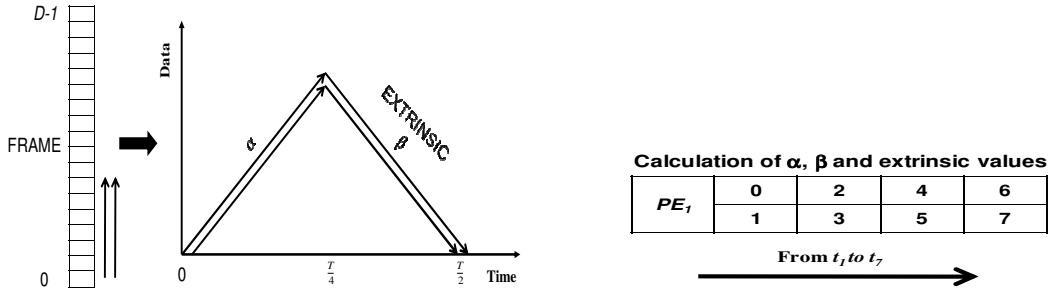
(b) Data access order for Recursive Unit Parallelism

Figure 1. 8. Recursive Unit Parallelism

2.4.4.2 Trellis Level Parallelism

In this type of parallelism [WOO00], rather than calculating path matrices (α and β values) and extrinsic values for each trellis transition, these values are calculated for more than one trellis transition at the same time. Degree of parallelism for calculating these values are bounded by the total number of transitions in a trellis. Trellis parallelism is usually represented as radix-2^S where *S* is the number of trellis transitions parallelized in decoder computation. Forward Backward scheme (explained in section 2.4.4) is modified for Trellis level parallelism in Figure 1. 9.a for *S = 2*. From data access order of this scheme, it is important to note that number of data elements processed by processing element at a given time instant is equal to *S*. Data access order for *D = 8, S = 2, & PE = 1* is shown in Figure 1. 9.b.

This approach has low area overhead [ASG10] since only computational units need to be duplicated, however, decoder memory is reduced which compensates the increase in computation units area cost. The increase in decoder throughput with slight increase in decoder area motivates to implement higher radix implementation for current standards [ASG10].



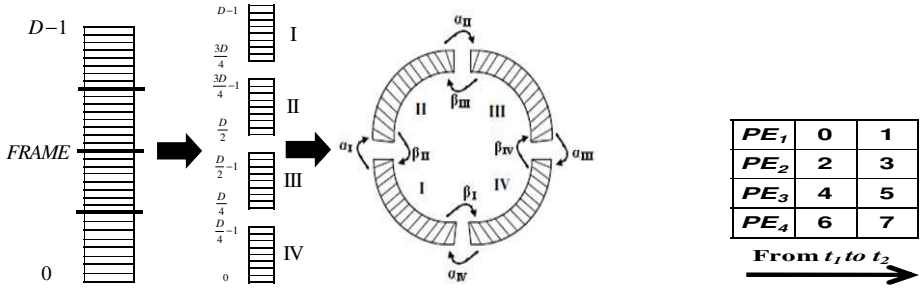
(a) Modified Forward Backward Scheme for Trellis Level Parallelism (b) Data access order for Trellis Level Parallelism

Figure 1. 9. Trellis Level Parallelism

2.4.4.3 SISO Decoder Level Parallelism

As explained at the start of the section 2.4.4, serial implementation of turbo decoder increases the decoder memory and latency for large amount of data. To tackle this problem, *sliding window BCJR algorithm* [BLA05] is proposed in which data block to be decoded is partitioned into number of segments or windows where each window is a subset of original block and has length D_w. Each window is then treated as separate data block and allocated to separate BCJR-SISO decoder. These SISO decoders then work in parallel so that α, β, γ and extrinsic values for one window are calculated at the same time as these values are calculated for other windows. Since the window can start or end in the middle of overall data block, the BCJR decoding algorithm approximated the initial values needed to start the calculation of α and β values. To make sufficiently accurate estimates, on either side of the window, forward and backward recursion have been run on some extra bits of the adjacent windows. This is called *acquisition*. Alternately, *message passing method* is used to initialize path metrics by the values generated in the previous iteration by adjacent windows. Sliding window method with message passing technique is shown in Figure 1. 10.a. Number of data processed at a given time instance depends on the number of partitions and window. If one processing element process one window then data access order for $D = 8$, $PE = 4$, $D_w = D/P = 2$ is shown in Figure 1. 10.b.

Approximated initial values for path metrics, calculated in sliding window technique, causes negligible loss in error correction performance of turbo decoder. In contrast, sliding window technique significantly increases throughput and reduces memory of turbo decoder.



(a) Sliding Window Method with Message Passing Technique (b) data access order for SISO Decoder Level Parallelism

Figure 1. 10. SISO Decoder Level Parallelism

2.4.4.4 Conclusion

To increase the throughput, any combinations of above described parallelisms can be used to implement turbo decoder. For example, we can use both trellis level parallelism and SISO level parallelism to implement turbo decoder. This technique will increase the throughput at the cost of increase in hardware cost. However, implementation of all these parallelisms causes memory access conflict problem (discussed in Section 4.1). With the increase of parallelism, this conflict problem also increases. In this thesis, we present algorithms that can resolve memory conflict problem for any combination of parallelism in polynomial time.

3. Block Codes

Block codes are second class of error correction codes that are used to transmit digital data reliably through unreliable communication channels in the presence of noise. Many types of block codes are used in different applications but among the classical block codes, Reed-Solomon [AHA] is the most popular due it widespread use in CD, DVD and hard disk drives. Other examples of classical block codes are Golay codes [GOL61] and Hamming codes [HAM50].

In block coding, long data stream is segmented into pieces of fixed length called a message block. Each message block, denoted by X, consist of x information bits that results in 2^x possible distinct messages. The function of the encoder is to transform each input message X into a binary c-tuple C where c > x. This binary c-tuple C is called codeword of the message X. In block coding, there are 2^x codewords corresponds to the 2^x possible messages and this set of 2^x codewords is called *block code*. In order to use this block code for practical purposes, it is necessary that these 2^x codewords must be distinct and have one-to-one correspondence with 2^x messages.

Encoding of block code with 2^x codewords with each codeword has length x is prohibitively complex since encoder has to store these 2^x codewords into memory. To reduce encoding complexity, linear block codes are used in practical application and can be defined as follows:

Definition ***Linear Block Codes***

A *linear block code* is a class of block codes in which modulo-2 sum of two codewords is also a codeword.

A generator matrix G is used to generate codewords in linear block codes. Generator matrix contains x linearly independent codewords with each codeword length c. Each message is multiplied by G to generate codeword corresponding to this message. Construction of (7, 4) linear code in which c = 7 and x = 4 is explained through an example. This example and the other information related to linear block codes are taken from [LIN04]. G for this example is shown below.

$$G = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

If X = (1 1 0 1) is the message to be encoded then its corresponding codeword is obtained as follows:

C = X.G

C = 1(1 1 0 1 0 0 0) + 1(0 1 1 0 1 0 0) + 0(1 1 1 0 0 1 0) + 1(1 0 1 0 0 0 1)

C = (1 1 0 1 0 0 0) + (0 1 1 0 1 0 0) + (1 0 1 0 0 0 1)

C = (0 0 0 1 1 0 1)

From this example, it is clear that (*c, x*) linear block code is completely specified by *G* and encoder has to store *x* rows of *G* to generate codeword of length *c* for any input message. All the codeword for this (7, 4) code is shown in Table 1. 2. From this table, it is clear that modulo-2 sum of any two codewords is also a codeword.

Table 1. 2: Linear block code with $x = 4$ and $c = 7$.

Message	Codewords
(0000)	(0000000)
(1000)	(1101000)
(0100)	(0110100)
(1100)	(1011100)
(0010)	(1110010)
(1010)	(0011010)
(0110)	(1000110)
(1110)	(0101110)
(0001)	(1010001)
(1001)	(0111001)
(0101	(1100101)
(1101)	(0001101)
(0011)	(0100011)
(1011)	(1001011)
(0111)	(0010111)
(1111)	(1111111)

3.1. Encoding of Linear Block Codes

As explained previously, encoder needs to store *x* rows of length *c* to encode any message in (*c, x*) linear block code. Another simplification in the implementation of encoder is carried out by introducing systematic structure during the construction of linear block codes. In this structure, codeword is divided into two parts namely message part and parity part as shown in Figure 1. 11. The message part contains *x* information (message) bits and the parity part contains $c - k$ parity bits that are linear sum of message bits. A linear block with systematic structure is called *linear systematic block code*.

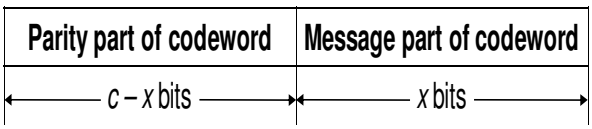


Figure 1. 11. Systematic format of a codeword

The (7, 4) code, shown in

Table 1. 2, is a linear systematic block code in which rightmost *x* bits of codeword are identical to the message. A liner systematic code is completely specified by its generator matrix *G* that can be divided in two matrices of order $x * x$ and $x * p$ where $p = c - x$ and $x * x$ is an identity matrix. Generator matrix for (7, 4) linear systematic block code is expressed as follows.

$$G = \begin{pmatrix} \overbrace{1\ 1\ 0}^{p\text{ matrix}} & \vdots & \overbrace{1\ 0\ 0\ 0}^{x * x\text{ identity matrix}} \\ 0\ 1\ 1 & \vdots & 0\ 1\ 0\ 0 \\ 1\ 1\ 1 & \vdots & 0\ 0\ 1\ 0 \\ 1\ 0\ 1 & \vdots & 0\ 0\ 0\ 1 \end{pmatrix}$$

To explain the simplicity introduced by linear systematic codes, let X = (x_0, x_1, x_2, x_3) is the message to be encoded and C = (c_0, c_1, c_2, c_3, c_4, c_5, c_6) is the resultant codeword then,

$$C = X.\begin{pmatrix} 1\ 1\ 0\ 1\ 0\ 0\ 0 \\ 0\ 1\ 1\ 0\ 1\ 0\ 0 \\ 1\ 1\ 1\ 0\ 0\ 1\ 0 \\ 1\ 0\ 1\ 0\ 0\ 0\ 1 \end{pmatrix}$$

Using matrix multiplication,

$$c_6 = x_3$$
$$c_5 = x_2$$
$$c_4 = x_1$$
$$c_2 = x_1 + x_2 + x_3$$
$$c_1 = x_0 + x_1 + x_2$$
$$c_0 = x_0 + x_2 + x_3$$

From these equations, it is possible to generate encoding circuit rather that storing rows of *G* [LIN04]. Encoding circuit for (7, 4) systematic linear code discussed in this section is shown in Figure 1. 12.

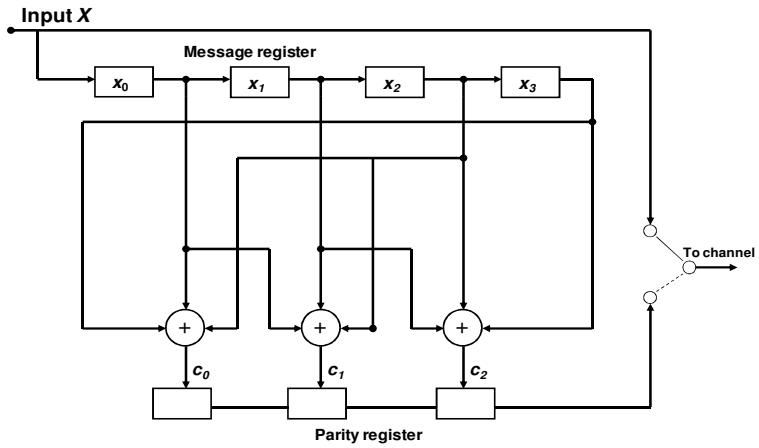


Figure 1. 12. Encoder circuit for the (7, 4) Systematic code

3.2. Low Density Parity Check (LDPC) Codes

Low density parity check Codes (LDPC) are a class of linear block codes with error correction capabilities very close to the channel capacity. Due to their excellent error correction performance, it has already included in several wireless communication standards such as DVB-S2 and DVB-T2 [DVB08], WiFi (IEEE 802.11n) [WIF08] or WiMAX (IEEE 802.16e) [WIM06].

But first we explain how we represent code with parity check equations. Consider a codeword **C = [c_1 c_2 c_3 c_4 c_5 c_6]** which satisfies the following three parity check equations.

$$c_2 \oplus c_3 \oplus c_4 = 0,$$
$$c_1 \oplus c_2 \oplus c_4 = 0,$$
$$c_1 \oplus c_3 \oplus c_4 = 0$$

Codeword constraints or parity check equations are often expressed in matrix form as follows:

$$\underbrace{\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}}_{H} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

The above H matrix is an $M * N$ binary matrix where each row M_i of H corresponds to a parity check equation whereas each column N_j associated with codeword bit. A nonzero entry at (i, j)th location means that the jth codeword bit is included in the ith parity check equation.
For a codeword $x \in C$ to be valid, it must satisfy the equation:

$$xH^t = 0, \forall\, x \in C$$

As the name explains, LDPC codes are block codes that contains only a small number of 1's in comparison to the number of 0's in parity check matrix H. This sparseness in H keeps the complexity of iterative decoding at reason limit and increases linearly with the code length.

3.3. Tanner Graph Representation

Due to the sparseness of *H-matrix*, LDPC codes can be graphically represented as a bipartite graph called *Tanner Graph* which depicts the association between code bit and parity check equation. The Tanner graph consists of two sets of vertices: variable node set (VNs) and check node set (CNs). A data $v_i \in$ VN represents one bit in the codeword (i.e. data to be processed) whereas $c_j \in$ CN represents a check equation used in generating parity check bits (i.e. operation to be done on the data). An edge e_{ij} connects the ith check node with jth variable node if jth variable node (VN) is checked by or included in ith check node (CN) which means that number of edges in the Tanner graph is the same as the number of 1s in the H matrix. Tanner graph is helpful in understanding the decoding process which functions by exchanging messages between CN and VN along the edges of these graphs. Tanner graph for the H-matrix in the previous section is shown in Figure 1. 13. In this figure, a message passed from CN to VN is called $Mes_{c \to v}$ whereas a message going from VN to CN is represented by $Mes_{v \to c}$.

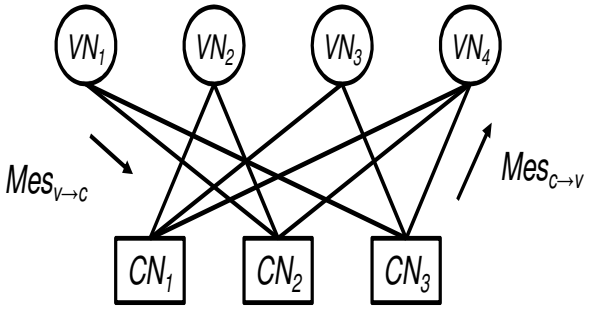


Figure 1. 13. Tanner Graph representation of H

3.4. Decoding

As explained in the previous sections, the algorithms to decode LDPC codes functions by exchanging messages along the edges of a Tanner graph. These algorithms are collectively called message-passing algorithms. Message passing algorithms are a type of iterative decoding algorithm in which CN and VN iteratively exchange massages forward and backward until decoding is completed (or stopping criteria is reached). These algorithms are named for the type of operations executed at the nodes such as belief-propagation or sum-product decoding [PEA88], min-sum decoding [FOS99] or normalized Min-Sum decoding [CHE02].

Sum-Product algorithm is a message passing algorithm which accepts the probability for each received bit as input. These probabilities represent a level of belief regarding the value of codeword bits. Due to simplicity in computation, the probabilities are converted in logarithmic domains that are called Log likelihood Ratio (LLR). The LLRs are defined by the following equation:

$$LLR = \log\left(\frac{P(v=0)}{P(v=1)}\right)$$

where $P(v = i)$ is the probability that bit v is equal to i.
The input LLRs are also called *a-priori* values because they are know in advance even before the decoding of LDPC codes is started.
The decoding is carried out in four steps which are presented below.

I- Initialization

In this step, a-priori LLR values are assigned to all the outgoing edges of every VNs. The sign of LLR provides a hard decision on the transmitted bit whereas it magnitude $|LLR|$ gives an indication on the reliability of this decision.

II- Check node Update

In this step, *jth* CN estimates the value of *ith* VN based on the values received from other VNs connected to this CN which means that the decision about the *ith* VN value is made completely

independently from the value just received from it. The CN supposed to create extra, *extrinsic*, information about *ith* VN value. In check node Update, each CN updates all connected VNs through the extrinsic information calculated for each VN. Bayes law in the logarithmic domain is used to calculate the sign (Equation 1.7) and the absolute value (Equation 1.8) of the extrinsic message for each VN.

$$sign(Mes_{c\to v}) = \prod_{v'\in v_c/v} sign(Mes_{v'\to c}) \tag{1.7}$$

$$\left|Mes_{c\to v}\right| = g\left(\sum_{v'\in v_c/v} g\left(\left|Mes_{v'\to c}\right|\right)\right) \tag{1.8}$$

where v_c represents the set of all the VNs connected with current CN and v_c / v means all the VNs in v_c except v. The function $g(x)$ is represented through Equation 1.9 as below.

$$g\left(x\right) = -\ln\tanh\left(\frac{x}{2}\right) = \ln\frac{\exp x+1}{\exp x-1} \tag{1.9}$$

III- Variable node Update

In this step, each VN updates its value based on the extrinsic information received from all the connected CNs. The soft output (SO) value also called A-Posteriori Probability (APP) is calculated using Equation 1.10, where *LLR* is the initial soft input or a-priori value. The *SO* value is used to calculate new VN to CN message $Mes_{v\to c}$ and is give by Equation 1.11.

$$SO_v = LLR + \sum_{c\in c_v} Mes_{c\to v} \tag{1.10}$$

$$Mes_{v\to c} = SO_v - Mes_{c\to v} \tag{1.11}$$

IV- Iterative Process

The forth step is to repeat the check and variable node update processes for new APP values until all the parity-check equations are satisfied or until a maximum number of iterations has reached and the decoder halts. At the end of iterative process or decoding, a hard decision is made based on the signs of values of VNs to output the codeword.

3.5. Implementation of LDPC Decoder

Analyzing the equations presented in LDPC decoding, it is clear that implementation of variable node update is quite simple and straight forward whereas the implementation of check node update is complex. The *g(x)* function used in check node update is highly non-linear and Look-Up Tables (LUTs) are required to directly map *g(x)* into hardware. However, to cope with finite precision issues, significantly large number of bits is required which results in large LUTs and significant increase in hardware cost. To reduce the computational complexity and hardware cost, different suboptimal algorithms are proposed [FOS99] [CHE02] to avoid the evaluation of the *g(x)* function.
The Min-Sum algorithm approximates and hence simplifies the sum-product algorithm by replacing g(x) function with the most minimum incoming message. The approach eliminates the complexity of check node update and can be expressed as:

$$Mes_{c\to v}^{new} \approx \min_{v'\in v_c/v} \left| Mes_{v'\to c} \right| \qquad 1.12$$

Dramatic reduction in computational complexity through Min-Sum algorithm [FOS99] results in the degradation of decoding performance because the approximated magnitude is always overestimated. To avoid this overestimation, a normalization factor μ is multiplied with the output obtained from equation 1.12. The resultant algorithm is called Normalized Min-Sum (NMS) [CHE02] and is given by following equation.

$$Mes_{c\to v}^{new} \approx \mu \min_{v'\in v_c/v} \left| Mes_{v'\to c} \right| \qquad 1.13$$

where μ is the normalization factor, $0 < \mu < 1$.

Despite simplifying the computations at the nodes of Tanner graph, the routing of the edges of Tanner graph is not a straight forward process and requires trade-off between hardware cost and throughput. To cope with routing of edges of the Tanner graph, following three implementations are presented in literature.

Each implementation has its own merits and demerits with respect to routing complexity and latency that is presented in the next section.

3.5.1. Fully Parallel Implementation

In this architecture, dedicated computational circuitry and network architecture is constructed to directly map every Tanner graph node and edge in hardware. In this way all the check node and variable node update is processed in two steps [BLA02] [FAN06] [NAG04] [ZHO07]. This architecture results in an excellent decoding speed but at the cost of huge circuit size. For example, if we have E_{Tan} edges in Tanner graph and each message requires N_b bits to represent each message, then routing circuit of this tanner graph requires $E_{Tan} * N_b$ wires to implement fully parallel architecture. This circuit size increases proportionally with the increase of code length since new check and variable

nodes along with wires are required for parallel implementation. Furthermore, hard-wiring of Tanner graph precludes the Flexible implementation of LDPC decoder.

3.5.2. Fully Serial Implementation

In this architecture, check and variable nodes updates are processed individually one at a time. This architecture results in lowering the hardware cost of decoder implementation but introduces huge latency in LDPC decoding. For example, if check node update requires T_{ch} clock cycles per input edge and variable node update takes T_{var} clock cycles per input edge then serial implementation requires $E_{Tan} * N_b(T_{ch} + T_{var})$ clock cycles to complete one iteration in LDPC decoding. Also, this delay increases proportionally with the increase of code length. Furthermore, significant amount of decoder memory is required to store all check to variable messages $Mes_{c \to v}$ and variable to check messages $Mes_{v \to c}$. For example tanner graph in discussion requires $2E_{Tan} * N_b$ bits decoder memory to implement serial architecture.

3.5.3. Partially Parallel Implementation

Partially Parallel implementation [MAS07] [LEE08] is a trade-off between prohibitive hardware cost (resulted through parallel implementation) and low throughput (caused by serial implementation). In this architecture, several processing elements (PEs) are realized in hardware and are shared between groups of variable and check nodes. Proper numbers of PEs are used in this implementation to obtain required throughput. Memory is used to store different message generated during check and variable node update. These messages are written into and read out of the memory according to particular edge permutation of Tanner graph. This implementation requires decoder memory just like serial implementation. However, this architecture suffers from memory access collision problem where more than two processing elements want to access the same memory bank. Collision problem becomes a significant issue with the increase of code word and discussed in the next section.

4. Memory conflict problem

As explained in previous section of this chapter, parallel architecture is the only feasible tradeoff between hardware cost and throughput to implement decoder for turbo and LDPC codes. Typical architecture of parallel implementation is shown in Figure 1. 14. In this figure, P processing elements (PEs), used to process data elements, are connected to B memory banks, where $P = B$, through interconnection network.

Memory conflict is a major source of concern in designing parallel architecture. The problem arises when more than two processing elements concurrently want to access two data elements that are stored in a same memory bank. Memory conflict problem is almost the same for both turbo and LDPC codes. However, due to difference in accessing the data in parallel, this section presents this problem separately for turbo and LDPC codes.

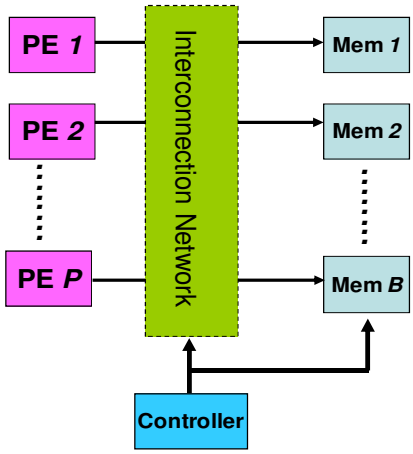


Figure 1. 14. Partially Parallel Architecture

4.1. Memory conflict problem for Turbo Codes

As explained in section 3.4.2, interleavers has been used to improve error correction performance of turbo codes by scrambling data so that parity bits generated by two constituent encoders are completely different. To implement parallelism presented in previous sections, interleaver needs to be parallelized in order to increase the communication bandwidth proportionally. To manage this bandwidth, memory is divided into smaller memory banks so that multiple data values can be fetched from memory concurrently in both natural and interleaved order. However, due to scrambling caused by interleaver, this parallelism results in communication or memory access conflicts which occur when multiple data values are fetched from or stored in the same memory bank at the same time.

The problem can best be explained through simple interleaver. In this interleaver, we first choose a matrix of particular order and then write data in this matrix row by row so that data is filled in this matrix. For *D = 16* and matrix of order *4x4*, first four data are placed in first row, next four data are placed in second row until matrix is filled with all the data as shown in Figure 1. 15.a. Afterwards, for natural order, we read this matrix row by row and for interleaved order we read this matrix column by column. We can represent both of these orders as:

Natural order = Content of first row , Contents of 2nd row , ……., Contents of last row

Interleaved order = Content of first column , Contents of 2nd column , ……..., Contents of last column

For our example,

Natural order = (0,1,2,3) , (4,5,6,7) , (8,9,10,11) , (12,13,14,15)

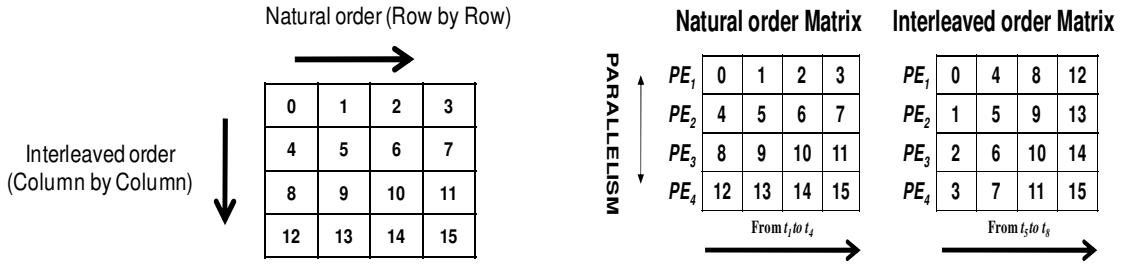
Natural order = 0,1,2,3, 4,5,6,7, 8, 9, 10, 11, 12,13,14,15

Where as

Interleaved order = (0,4,8,12) , (1,5,9,13) , (2,6,10,14) , (3,7,11,15)

Interleaved order = 0,4,8,12, 1,5,9,13, 2,6,10,14 ,3,7,11,15

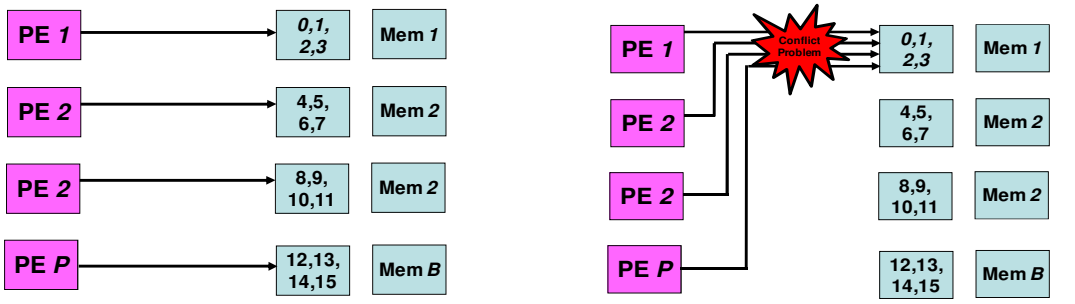
For parallel processing using sliding window technique, this codeword is divided into four windows in both natural and interleaved order where each window is processed by one processing elements as shown in Figure 1. 15.b.



(a) Interleaving (b) Parallel Processing

Figure 1. 15. Parallel Processing of Turbo Codes

To increase memory bandwidths, four memory banks are used so that each processing element can concurrently get data elements in parallel. If data elements are stored in banks in such a manner that at each time instant in natural order, all the processing elements always access different memory banks as shown in Figure 1. 16.a. than in interleaved order all processing elements always access the same memory bank at each time instance as shown in Figure 1. 16.b. This results in *Memory conflict Problem* and increases latency in data fetching from memory due to the presence of conflict management mechanism in communication network. Furthermore, this problem reduces system throughput and increases system cost.



(a) Conflict free natural order Access (b) Conflict full Interleaved order Access

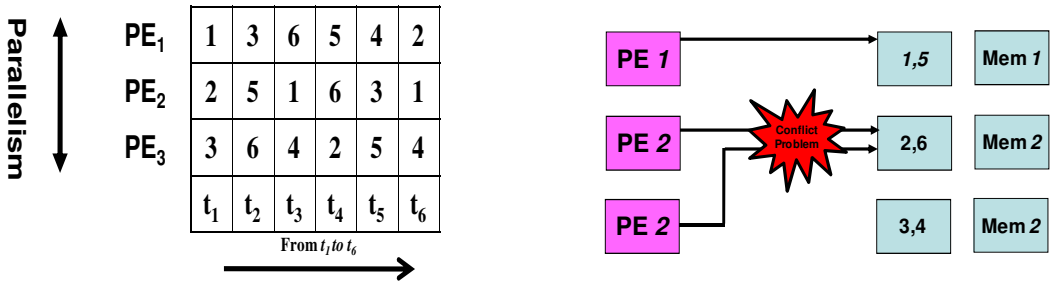
Figure 1. 16. Memory Conflict Problem in Parallel Turbo Decoder

4.2. Memory conflict problem for LDPC Codes

As explained in previous section, computation at both check node and variable node is quite simple and the implementation issues mainly arise due to routing complexity between VNs and CNs. Partially parallel architecture proves to be only feasible solution to implement LDPC decoder but this architecture suffers from *Memory Conflict Problem*.

To explain the problem, we introduce a mapping matrix in which we have P rows, related to the processing elements, and N columns, related to the time instances t_i. Each column represents the data which need to be accessed in parallel by P processing elements at t_i. Also, data in each row are processed by the processing element connected with this row. Figure 1. 17.a represents the mapping matrix in which we have D = 6, P = B = 3, M = 2 and N = 6 where D is the number of data elements, *B*

is the number of memory banks and M = D/B, is the size of each memory bank. If data elements stored in bank 1, bank 2 and bank 3 are (1,5), (2,6) and (3,4) respectively then at both time instances t_4 and t_5, more than one PEs want to access the same memory bank. Figure 1. 17.b shows the conflict at t_4. This arises in memory conflict problem and results in increased system latency & hardware cost and reduced system throughput due to the presence of control structure to handle conflicts.



(a) Mapping Matrix

(b) Conflict at t_4

Figure 1. 17. Memory Conflict Problem in Partially Parallel LDPC Decoder

5. Conclusion

The basic purpose of this chapter is to give a brief introduction to error correction coding. Error correction coding can be divided into two broad categories namely convolutional codes and block codes. Both of these codes are integral part of current telecommunication standards. Basic concepts related to encoding and decoding of convolutional codes have been first presented. These concepts have been used to explain turbo codes that are a subclass of convolutional codes with error correction capabilities near to channel capacity. Different techniques to speed up the decoding of turbo codes for high data rate applications have been explained. Then, brief description related to block codes with particular emphasis on LDPC codes has been given. Decoding approach related to LDPC codes and implementation of its decoder on different architectures has been explained. Finally, memory conflict problem related to the implementation of turbo and LDPC decoders on parallel architectures have been highlighted through simple examples.

The main purpose of this thesis is to simplify the design of parallel interleavers. The target architecture is composed of several processing units connected to memory banks through interconnection network. State of the art approaches to design such hardware architectures are presented in next chapter. Existing approaches to find conflict free memory mappings are based on complex heuristics. Prime objective of this work is to propose new approaches based on bipartite graph and edge coloring algorithm that allow to find a solution in a polynomial time.

Chapter 2

STATE OF THE ART

Table of Contents

1. Introduction ----------29
2. Approaches to tackles memory conflict problem for turbo and LDPC codes -----29
2.1. Conflict Free Interleaving laws ----------30
2.1.1. Architecture aware Turbo Codes ----------30
2.1.2. Structured LDPC Codes ----------32
2.2. Run Time Conflict Resolution ----------33
2.2.1. Run Time Conflict Resolution for Turbo Codes ----------33
2.2.2. Run Time Conflict Resolution for LDPC Codes----------37
2.3. Design Time Conflict Resolution----------38
2.3.1. Design Time Conflict Resolution for Turbo Codes----------38
2.3.2. Design Time Conflict Resolution for LDPC Codes ----------40
2.4. Conclusion ----------41
3. Node and Edge Coloring of Graph ----------42
3.1. Graph Theory----------42
3.2. Conflict Graph----------44
3.2.1. Conflict Graph for Turbo Codes----------44
3.2.2. Conflict Graph for LDPC Codes ----------45
3.3. Bipartite Graph ----------46
3.3.1. Bipartite Edge Coloring----------47
3.3.1.1 Vizing Method to color the edges of Bipartite Graph ----------47
3.3.1.2 Gabow Method to color the edges of Bipartite Graph----------48
4. Transportation Problem----------51
5. Conclusion ----------53

This chapter is divided into two parts. In the first part, different techniques to tackle the memory conflict problem on parallel architecture for turbo and LDPC codes are presented. Advantages and disadvantages of each technique and motivation to present our work are also explained in this part. In the second part, algorithms to color the edges of bipartite graph and methods to solve the transportation problem are explained through example so that algorithms proposed in this work can easily be comprehended. These algorithms are used in later chapter to solve memory mapping problem in polynomial time.

1. Introduction

In the previous chapter of this thesis, we introduced different error correction techniques for reliable data transfer between transmitter and receiver. Different architectures for implementing decoders for these techniques are also presented in this chapter. It is clear from this discussion that partially parallel architectures provide the reasonable tradeoff between cost and throughput to be implemented in practical applications. However, this kind of architecture suffers from the memory conflict problem. In the first part of this chapter, different techniques to implement parallel architectures taking into account the conflict problem are discussed. The second part presents different techniques to solve bipartite edge coloring and transportation problem that are used in the approaches explained in next chapters to solve memory mapping problems.

First part discuses different techniques to tackle memory conflict problem for turbo and LDPC codes. In first approach, different algorithms to construct conflict free interleaving law are discussed. For turbo codes, techniques to design conflict free interleaving law by taking into account different architecture constraints are presented whereas, for LDPC, structured codes are discussed in this section. The main reason to develop these techniques is to construct architecture friendly codes with good error correction capabilities in order to reduce hardware cost during implementation of these codes for practical applications. In second approach, different design innovations are introduced to tackle conflict problem. Flexible and scalable interconnection with sufficient path diversity and memory elements are introduced in this section to handle memory conflict. Third approach deals with algorithms that assign data in memory in such a manner that all the processing elements can access memory banks concurrently without any conflict.

In the second part of this chapter, we first explain different graph definitions that are helpful in understanding algorithms based on graph theory presented in next chapters. Then, modeling based on conflict graph is presented to explain the complexity of mapping problem and why we need new algorithms to solve this problem. Edge coloring is the main technique used in our thesis, so we present different approaches to find edge coloring of bipartite graph from literature. In the final section, transportation problem is presented that is used to solve mapping problem related to turbo codes in next chapter. This section first explains how we model our bipartite graph as transportation matrix and then present an algorithm to find optimal solution for transportation problem.

2. Approaches to tackle the memory conflict problem for turbo and LDPC codes

As explained in the previous chapter, turbo and LDPC codes are part of current telecommunication standards due to their excellent error correction capabilities. For high data rate applications, turbo decoders are implemented with parallel architecture whereas LDPC decoders are realized with partially parallel architectures in practical cases. Both of these architectures result in memory conflict problems and different approaches have been used to solve this problem in literature. These approaches can be classified in three broad categories. In this section, we try to present the state

of the art related to each category along with their advantages and disadvantages and explains which factors motivate us to present current polynomial time algorithm to solve memory mapping problem.

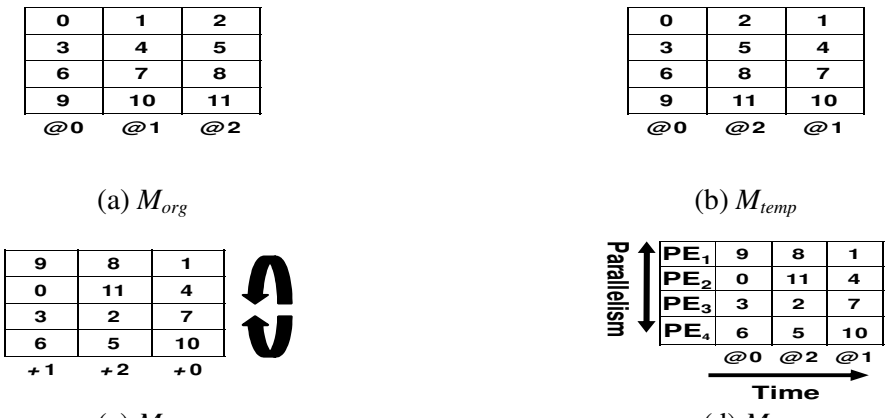
2.1. Conflict Free Interleaving laws

In the first category, code developers take into account memory conflict problem during code construction for both turbo and LDPC codes. Due to difference in the code construction procedure, approaches to construct architecture-aware codes for both turbo and LDPC codes with good performance are presented separately in this section.

2.1.1. Architecture aware Turbo Codes

For turbo codes, the main reason for conflicts in parallel architecture is the presence of interleaver. So, the code developers put all their attention in the development of conflict free interleaving law with good error correction performance. Conflict free interleaving law provides parallel concurrent accesses to each memory bank without any conflict.

In [GNA03], spatial and temporal permutations are introduced to construct conflict free interleaver with random interleaver like properties. The approach can best be explained through example. Consider a block length of 12 data arranged row by row into a matrix M_{org} as shown in Figure 2. 1.a. Interleaver function is the sum of both temporal and spatial permutations. In the second step, temporal permutation is obtained by changing the positions of the column in M_{org} to obtain M_{temp} as shown in Figure 2. 1.b. After wards, spatial permutation is performed by applying different circular permutations to different columns to obtain the interleaved matrix M_{spa} as shown in Figure 2. 1.c. Each row is processed by each processing elements and the memory size is represented by the number of columns as shown in Figure 2. 1.d. The benefit of this approach is that we can use barrel shifter interconnection network to realize turbo decoder for this interleaving law in practical applications. However, the approach is not standard compliant and cannot be used for other interleaving laws.



(a) M_{org} (b) M_{temp}

(c) M_{spa} (d) M_{int}

Figure 2. 1. Temporal and Spatial Permutation for interleaver construction

Similar techniques are used in [GIU02] and [KWA02]for designing conflict free interleaver, however, the principal purpose of designing interleaver is to construct good performance error correcting turbo codes that supports different block lengths. It is necessary that each telecommunication standard supports different block lengths in order to fulfill different user requirements and channel conditions. Prunable and deterministic interleaver is one of such interleaver that can easily realize different block lengths in practical implementation.

Definition ***Prunable Interleaver***

Prunable interleaver is the one which can be modified to obtain the interleaver of shorter length that keeps the error correction capabilities of the original larger interleaver.

Prunable interleaver provides flexibility in codeword or interleaver length to meet the changing user requirements and channel conditions. In order to obtain the interleavers of different lengths, prunable interleaver is stored and interleavers of shorter lengths are obtained by modifying it.

Definition ***Deterministic Interleaver***

Deterministic Interleaver uses some algorithm to generate address of interleaved data on the fly.

Deterministic interleaver is easy to implement as compared to *random interleaver* which randomly generates interleaved addresses and requires interleaver table or memory to store interleaver values. In applications or standards (such as DVB or LTE) where the frame size is large or different interleavers are used, storing each interleaver for both encoding and decoding is not a feasible solution from implementation perspectives

One of the deterministic interleaver is Quadratic permutation polynomial (QPP) interleaver [SUN05]. For long frame size, decoding performance of QPP interleaver is near to random interleaver whereas for short frame size, QPP interleaver performs better than random interleaver. QPP interleaver is a part of current 3GPP LTE standard [LTE08] and for block size N, it is represented by following equation.

$$\Pi(x) = (f_1 x^2 + f_2 x) \bmod N$$

where x and $\Pi(x)$ represents the original and interleaved address respectively and integers f_1, f_2 are different for different block lengths and can be found in the standard.

Also it is shown in [TAK06] that QPP interleaver is maximum contention-free i.e., for every window size W which is a factor of the interleaver length N, the interleaver is contention free. This is true for SISO decoder level parallelism. *However, for higher data rate applications when trellis and recursive units parallelism are also included in each SISO, QPP interleaver is not contention-free and requires a router and buffer mechanism to solve memory conflicts.*

The above mentioned conflict free interleavers are deterministic but they have the following two problems: first of all, architectural constraints apply during interleaver design impede the error correction performance of the turbo codes. Secondly, conflict free interleavers are regenerated interleaving pattern for particular parallel level M which makes it difficult to optimize code performance of interleaving pattern for different parallel levels M. The approach results in performance difference for different parallel levels M when applied to same turbo decoder which is not a desirable trade off in order to design conflict free parallel interleaver.

2.1.2. Structured LDPC Codes

From the introduction of the LDPC codes presented in previous chapter, it is clear that this code is completely specified by its H matrix and proper construction of this matrix is necessary to obtain excellent error correction capabilities of LDPC. Different constraints can be added during the construction of H matrix either to achieve significant coding gains or to simplify the decoder architecture.

So, in the first king of approach, H matrix is constructed is such a way that data transfer between check nodes (CNs) and variable nodes (VNs) can be made without any conflict for partially parallel architecture. To tackle memory conflict problem, H matrix is divided into different blocks of sub-matrices where each sub-matrix is obtained by permuting rows of the identity matrix [ZHA01] [MAN03]. The codes obtained by adding this regularity during construction of codes is called *structured codes*. Structured codes removes memory conflict problem because transfer of messages between (CNs) and (VNs) are carried out through simple rules (like indices permutation). Also, structured codes simplified the decoder architecture since interconnection network can be implemented through simple network (like barrel shifter) by exploiting the regularity introduced during code construction.

Due to simplicity in construction, structured codes are part of current telecommunication standards such as IEEE 802.11n (WiFi) [WIF08] and IEEE 802.16e (WiMAX) [WIM06]. In each of these standards, structured codes are constructed by dividing original matrix into different sub-matrices as shown in Figure 2. 2.a. This matrix has X row and Y columns. To construct structured codes, each entry $\Pi_{x,y}$ is replaced by Z*Z permutation matrix where each matrix is either a all-zero matrix or rotation of the identity matrix. Rotation of 3*3 identity matrix by 2 is shown in Figure 2. 2.b. Representing H matrix using identity matrix is a difficult and cumbersome task. Since each sub-matrix is either a null matrix or a permutation of the identity matrix, H matrix can be represented in a compact form by placing value of permutation rotation for non zero sub-matrix and by placing -1 for null sub-matrix as shown in Figure 2. 3. This compact form of H matrix is called a H_{Base} matrix. Figure 2. 3 represents H_{Base} matrix for WiMAX standard with

X = number of rows = 12, Y = number of columns = 24,

*Z = 24, codeword size = Y * Z = 576 and code rate = Y - X / Y = ½.*

$$H = \begin{bmatrix} \Pi_{0,0} & \Pi_{0,1} & \cdots\cdots & \Pi_{0,Y} \\ \Pi_{1,0} & \Pi_{1,1} & \cdots\cdots & \Pi_{1,Y} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \Pi_{X,0} & \Pi_{X,1} & \cdots\cdots & \Pi_{X,Y} \end{bmatrix} \qquad I = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

(a) Division of H matrix into different sub-matrices (b) Rotation of identity Matrix

Figure 2. 2. H matrix Representation

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	-1	94	73	-1	-1	-1	-1	-1	55	83	-1	-1	7	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	27	-1	-1	-1	22	79	9	-1	-1	-1	12	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	24	22	81	-1	33	-1	-1	-1	0	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1
4	61	-1	47	-1	-1	-1	-1	-1	65	25	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1
5	-1	-1	39	-1	-1	-1	84	-1	-1	41	72	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	46	40	-1	82	-1	-1	-1	79	0	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1
7	-1	-1	95	53	-1	-1	-1	-1	-1	14	18	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1
8	-1	11	73	-1	-1	-1	2	-1	-1	47	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1
9	12	-1	-1	-1	83	24	-1	43	-1	-1	-1	51	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1
10	-1	-1	-1	-1	-1	94	-1	59	-1	-1	70	72	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1
11	-1	-1	7	65	-1	-1	-1	-1	39	49	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0
12	43	-1	-1	-1	-1	66	-1	41	-1	-1	-1	26	7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0

Figure 2. 3. H_{Base} Matrix for WiMax Standard of code word size = 576, Z = 24 and r = 1/2

Although it is proved in [MAN03] that performance of structured codes is very close to random codes, adding constraints to construct structured codes may degrade the code's decoding performance. Therefore, special attention should be taken while selecting constraints to develop structured codes to keep remarkable error correction capabilities of LDPC. Also, structured codes only support one class of LDPC codes and to handle diverse existing and future classes of LDPC codes (such as non-binary LDPC codes), a general approach to handle the memory mapping problem is required which is discussed in this thesis.

2.2. Run Time Conflict Resolution

In a second family of approaches, data are mapped into memory in such a manner that there is no conflict in accessing data in natural order whereas for accessing data in interleaved order optimized interconnection network and extra memory elements are used to tackle memory conflict problems. This approach supports any type of turbo and LDPC codes because interconnections networks are used to manage the conflict at run time.

2.2.1. Run Time Conflict Resolution for Turbo Codes

The approach for turbo codes works in two steps. In the first step, interconnection network is built which is optimized to general or particular interleaver to reduce memory conflicts, then in the second step, extra memory elements are introduced which stores the remaining conflicts.

In [THU02], single LLR distributor (interconnection network) is connected with all the P processing elements on one side and all memory banks on the other side. This LLR distributor received all the P incoming data and their addresses information to determine target RAM. A tree like structure called *Tree Interleaver Bottleneck Breaker (TIBB)* (see Figure 2. 4.a) is proposed in which LLR distributor is served as root and the buffers associated with target RAMs are acted as leaves. Every buffer must have access to all incoming data in order to determine which data needs to be stored in its local RAM. Furthermore, each buffer has *P* connection with LLR distributor in order to store multiple inputs in one cycle. Therefore, complexity of interconnection network increases exponentially with increase in number of *P*. The optimization based on two stages of buffer is proposed which reduces the number of inputs per buffer but still LLR distributor spans the whole chip which makes

this solution inefficient in terms of area and cost. In [THU02a], another structure called *Ring Interleaver Bottleneck Breaker (RIBB)* (see Figure 2. 4.b) is proposed in which each processing element has its own local LLR distributor which results in less complex architecture as compared to single LLR distributor in TIBB structure. Each LLR distributor has to decide whether to store the incoming data into the local memory or to send the data to left or right distributor. The direction of non-local data is determined based on shortest distance to the target RAM. As several data can have the same target RAM, buffer that can store more than one data per cycle is needed. The size of the buffer can be determined by performing automated profiling for targeted interleaver. *RIBB* reduces the control and network complexity as compared to *TIBB* but introduces latency in order to transfer data to targeted RAM when comparing with *TIBB*.

(a) TIBB Architecture (b) RIBB Architecture

Figure 2. 4. LLR Distributor Architecture

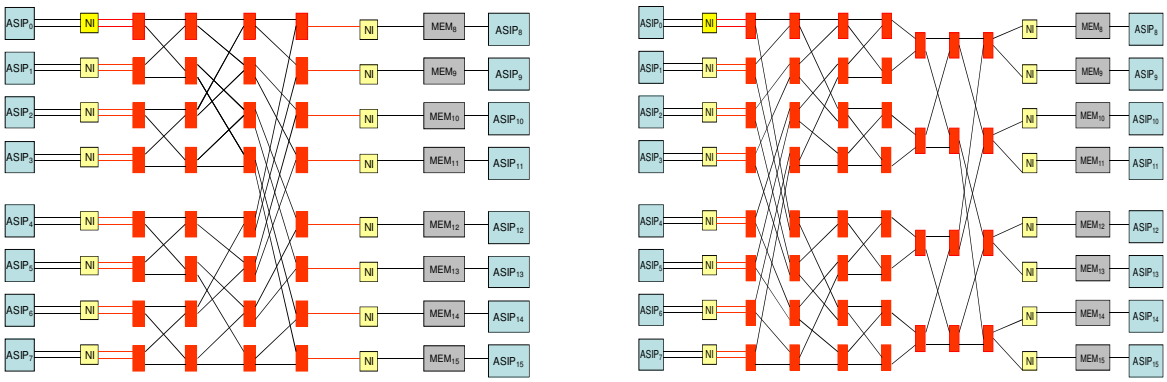
To improve the latency, new structure named *General Interleaver Bottleneck Breaker (GIBB)* is introduced in [THU03]. In *RIBB*, each local LLR distributor is connected to two neighboring LLR distributors but, in *GIBB*, it can be connected to any number of distributors. The approach increases the network capacity and hence throughput of the decoder. The topology of *GIBB* is represented as directed graph with associated routing information which uses shortest path routing to transfer data between different nodes or LLR distributor. The approach suffers from the problem that determining shortest path in general graphs is NP-complete and no optimum algorithm exists to find shorted path routing. Moreover, as the parallelism degree increases, hardware complexity of LLR distributor and increased buffer size makes above mentioned approaches prohibitive in terms of area and latency.

To reduce the high wiring and bad buffer sizing scalability, packet switched *Network-on-Chip* network topologies have been proposed in order to resolve the conflicts on run time. In [NEE05], mesh, torus and cube networks have been proposed in which packets contains the information about target processing unit in header and target memory unit & decoder data in payload. Sufficient numbers of nodes per dimension have been used to guarantee required network bandwidth suitable to cope with high volume of interleaving traffic during turbo decoding. Different dead-lock free routing algorithms based on an input-queued (IQ) and an output queued (OQ) packet switch router architecture have been investigated. For low to moderate throughput applications, input queuing scheme is used with reasonable implementation cost whereas high throughput can be achieved through output queuing strategy at the cost of high silicon area. However, all these topologies suffer from reduced scalability and available bandwidth necessary to construct high throughput flexible on-chip communication

network. Also, router complexity and cost increases significantly with the increase of parallelism due to complex buffer management architecture to store conflicting data.

To increase the scalability and to meet higher throughput requirement on flexible communication network, two heterogeneous multistage networks are investigated in [MOU07]. The Butterfly is a multistage on-chip communication network with unidirectional links and 2 input, 2 output routers (see Figure 2. 5.a). Butterfly network has following two advantages over the previously presented network:First of all, the network exhibits huge scalability because a network of diameter of *D* can be constructed from two networks of diameter *D-1*. Secondly, the packet routing algorithm is very simple that uses destination address bits for selecting output port of router at each stage of the network. This network consists of routers and Network Interfaces. Routers stores conflicting packets (through FIFO queue) and the network Interface includes interleaving (or deinterleaving) information into the header of the packet coming from processing element. However, Butterfly network provides unique path between each source and each destination and lacks in path diversity. This requires complex buffering architecture to manage conflicting packets and increased the silicon area and cost of the network.

The Benes network is the second multistage network studied in this article [MOU07]. It is constructed by putting two Butterflies back-to-back. The Benes network provides path diversity and all possible permutation between its inputs and outputs which is necessary to construct flexible network capable of supporting any type of turbo interleaver. However, this network avoids conflicts between packets if and only if all the packets have different destinations that are not the case in turbo decoding. To optimize Benes network for turbo decoding, a modified topology (see Figure 2. 5.b) and routing algorithm is presented. In this topology, number of routers in first stage is *2N* where as number of routers in second stage is reduced to *N* where *N* is the number of inputs and outputs of the network. Routing algorithm is based on the rule that packets which are intended for different router at each cycle are transmitted at the same time. To achieve this rule, some preprocessing is required to schedule all the packets by allocating each one a certain time slot. Packet format is the same as in Butterfly network and FIFO is replaced by registers in router. Also, two down counters are introduced in Network Interface in order to schedule packet transmission.



(a) Butterfly Network (b) 2N-N Benes Network

Figure 2. 5. Heterogeneous Multistage Network

Butterfly network suffers from buffer management architecture that significantly increases its area whereas Benes network requires pre-computation of routing paths and packet scheduling that is not a feasible solution for implementing different standards on the flexible decoder architecture. Also, path diversity provided by these networks is not sufficient to manage packet conflicts for communication intensive architecture such as turbo decoder. To tackle these limitations, Binary de Bruijn Interconnection Network is presented in [MOU08] that is scalable and allows any permutation to be routed efficiently. The network is best expressed through Binary de Brujin graph [BRU46]. A de Brujin graph of 16 nodes is shown in Figure 2. 6. to describe the path diversity between different nodes. Due to this path diversity, communication conflicts are managed by deflecting the conflicting packets appropriately until they reach the target processor rather than blocking or buffering them. To manage packet deflection efficiently, a modified shortest path routing algorithm is presented and based on the following rule: "If the two packets are intended for the same output port of the router, then the "youngest" of the two conflicting packets is deflected".

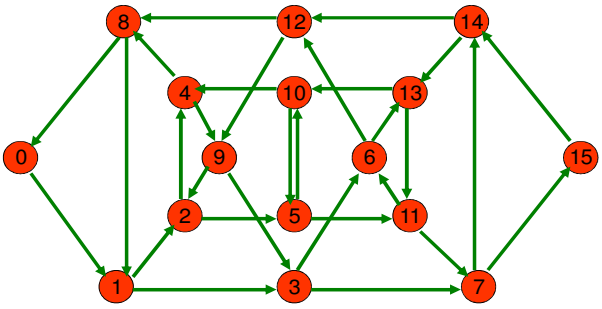


Figure 2. 6. Binary de Bruijn graph with 16 nodes

The flexible networks presented above suffer from large silicon area and cost due to increased buffer control architecture necessary to manage conflicting packets. Also, delay introduced due to conflict management mechanisms degrades the maximum throughput and makes these approaches inefficient for high data rate and low power applications.

Recently, some low cost optimized interconnection networks and buffer management schemes are proposed for interleavers used in current telecommunication standards. In [WON10], multistage network based on the barrel shifter is proposed for 3GPP LTE System for parallel decoder architecture. Due to the permutation characteristics of QPP interleaver used in LTE, the connection between each memory module and its corresponding SISO is established by shifting each sub block by a certain offset. The resultant interconnection network has short path delay and simplified routing control mechanism which results immediate transfer of data between SISO and memory. Most importantly, for high data rate and low power applications, the proposed network can significantly reduce the decoder hardware cost. However, the proposed network, can only apply to QPP interleaver where the number of SISOs are power of 2 and the approach does not work for any other interleaving law.

2.2.2. Run Time Conflict Resolution for LDPC Codes

As stated in Chapter 1, computational complexity at both VN and CN is reduced using suboptimal algorithm. Consequently, a real source of concern for designing LDPC decoder is a transfer of massages between CN and VN based on the routing of edges of Tanner graph. To solve this message routing problem, different scalable and flexible interconnection networks have been proposed in literature to solve the memory conflict at run time. This increases the flexibility and scalability in implementing different LDPC codes at the cost of increased latency and hardware cost.

In [THE05], an approach based on Network on Chip (NoC) is presented in which VN and CN act as processing elements (PEs) and uses on chip network based on 2-D mesh topology to communicate with each other. Based on the H matrix of a given LDPC code, the approach generates the configuration data which contains the information regarding the number of nodes (either VN or CN) allocated to particular PE and the connectivity information between different PE based on the communication pattern of different nodes. Each PE has a dedicated memory to store this configuration data. Packet sent by each PE contains information about data and the address of sender and receiver for routing data on the network. To minimize routing overhead, intelligent mapping algorithm is used to map Tanner graph on the physical network to reduce the distance packet must travel across the network to reach the destination. Similarly, in [KIE03] another heterogeneous network called message distribution network (MDN), whose topology is based on randomly generated graph, is used to resolve memory conflict at run-time. Exchange of data between VN and CN is carried out by adding destination header onto produced message. Different memories are use to store data, addresses, produced messages and received channel values. Due to conflict management mechanism, achievable throughput is low which increases the decoder latency. Also, implementation of LDPC decoder using MDN requires many resources which increase the area and power consumption of the overall system.

However, on-chip networks presented above offer little support for scalability and flexibility in designing decoder architecture. The resultant overhead such as low latency and large area and power consumption makes their implementation difficult for practical application. To provide scalability at reasonable latency and area overhead in [MOU08], binary de-bruin network (see Figure 2. 7) based on Binary de Bruijn graph [BRU46] is presented to handle communication between VNs and CNs. Due to path diversity depicted through Figure 2. 6, de bruijn network provides flexible interconnection network to map routing of edges of Tanner graph. Allocation of VNs and CNs to each processor depends on the code rate, node degrees and size of extrinsic memories. Furthermore, since the sum of all the VNs degrees is equal to the sum to all the CNs degrees, so half the processors can be use for CN computation and half for VN computation. To map the edges of Tanner graph, header of each packet contains information about targeted processor along with destination memory write address. For packets routing, modified shortest path algorithm is used in which one of the two packets that are destined for the same output is deflected rather than blocking or buffering it in order to reduce the size of router. The criterion for deflecting packet is based on “round-robin” arbitration that deflects “youngest” of the two conflicting packets.

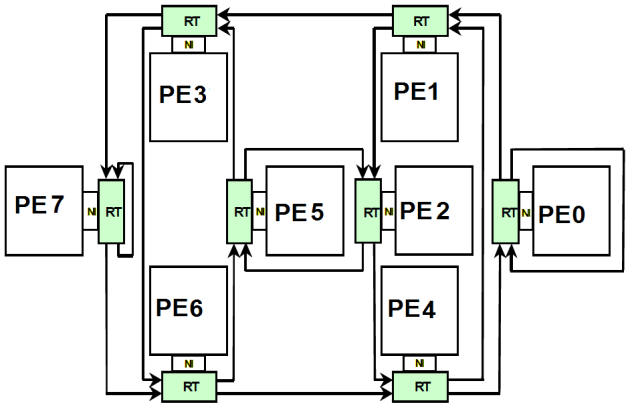


Figure 2. 7. 8-Processor de Bruijn network with processors, routers and network interfaces

2.3. Design Time Conflict Resolution

In a third kind of approaches, different algorithms are proposed to provide conflict free parallel concurrent access to all Processing Elements (PEs) through some pre-processing to determine the memory locations for each data element used in the computation. The benefit of these approaches is that decoder implementation does not need any specific network and extra memory elements to support particular interleaving law. Rather any network which supports all the permutation patterns between inputs and outputs can be used to implement any interleaving law. However, the approach requires some preprocessing to map data in different memory banks for different block lengths and parallelism degree.

2.3.1. Design Time Conflict Resolution for Turbo Codes

First algorithm to resolve conflict problem for turbo codes at design time is presented in [TAR04] using simulated annealing metaheuristic. Mathematical model and mapping constraints related to the problem can be found in [TAR04]. The main emphasis is to present the method in this section so that one can better understand the algorithm. As explained in the previous chapter (section 4.1), turbo codes can be represented through two matrices: one related to the natural and the other related to the interleaved order of access. Data elements in each column of both natural and interleaved matrices are needed to be accessed in parallel. For example in Figure 2. 8.a, data 1, 6, 11, 16, 21 are need to be accessed in parallel. For this algorithm, each column in interleaved matrix is called *tile* and is given the alphabetical names as shown in the last row of interleaved matrix of Figure 2. 8.b. This tiling information is placed in natural matrix to prepare *tiling matrix* as shown in Figure 2. 8.c. For example, data elements 1 and 2 is in tile *E*, so in tiling matrix we place *E* at the positions where both 1 and 2 exists in natural matrix. Now for conflict free memory mapping, all the banks assigned to the data elements in each tile of tiling matrix and each column of natural matrix should be different.

Natural order Matrix

P1	1	2	3	4	5
P2	6	7	8	9	10
P3	11	12	13	14	15
P4	16	17	18	19	20
P5	21	22	23	24	25

(a)

Interleaved order Matrix

P1	25	10	22	3	1
P2	14	11	5	6	2
P3	15	13	7	16	20
P4	24	18	8	19	17
P5	12	21	4	23	9
Tile	A	B	C	D	E

(b)

Tiling Matrix

E	E	D	C	C
D	C	C	E	B
B	A	B	A	A
D	E	B	D	E
B	C	D	A	A

(c)

Figure 2. 8. Matrices used in [TAR04]

Algorithm presented in [TAR04] is divided in two steps which are described below,

First step : "*Any step that produces a preliminary mapping matrix with the following properties: every column and every tile contains at most one element equal to every symbol in {1,…,P}"*. The construction of this preliminary matrix is further improved through greedy initialization of mapping matrix which is presented in [TAR05] as, "*For i = 1,…, P, read the i-th row of the mapping matrix from left to right and set the value of the current element to i, provided that it does not collide with the other elements on the same row. Otherwise, keep it blank"*.

Second step: In this step, the algorithm apply simulated annealing algorithm on the preliminary mapping matrix to fill all blanks present in it. For this, the algorithm injects a collision, i.e, either in one column or in one tile, more than one data contains the same symbol, and solve this collision at each step by possible introducing another one.

The problem with this algorithm starts from second step because one does not know how long the algorithm takes to determine conflict free memory mapping by iteratively injecting and solving the collisions. Though the proof is given in [TAR04] that algorithm is always able to find memory mapping but computational complexity of the problem inhibits the addition of other constraints such as architecture oriented memory mapping into the algorithm.

In [LIN10], optimized memory address remapping (OPMM) is presented in which certain Collision-Free Exchange rules are defined to complete the simulated annealing procedure much faster than that achieved in traditional method presented in [TAR04]. OPMM accelerates the annealing procedure in two ways: First it selects pairs of data elements which need to exchange their bank in order to perform number of OPMM steps in one iteration. Secondly, during future iterations, selected data exchange pairs interchange their bank information so that exchanged data elements can only be varied between two banks instead of *P* memory banks. As a result, number of iterations to complete the annealing procedure is much smaller than [TAR04] when annealing randomly select exchange data element pairs.

Experiments show that OPMM procedure finds conflict free memory mapping in much smaller CPU time but still the method is based on metaheuristic and complexity or time of completion of the algorithm is unknown. This results in lower CPU time for particular interleaver pattern and longer CPU time for others depending on the choice of data exchange pairs. Also the complexity of the algorithm makes it difficult to add additional constraints for finding architecture or network oriented memory mapping.

In [CHA10a], new simplified approach called Static Address Generation Easing *SAGE* is presented that takes additional constraints to determine architecture oriented memory mapping. In this approach, two empty matrices called *SAGE Mapping Matrices* are used to store banks information during algorithm execution. These matrices (MAP_{Nat}, MAP_{Int}) have the same order as the natural or interleaved order matrices as shown in Figure 2. 9. To find architecture oriented memory mapping, two constraints are defined to be respected during algorithm execution. First, each column of the mapping matrices should contain different memory banks and second, if interleaving law allows, each column should respect the rules of the steering network component. Algorithm initializes by assigning memory banks to the first column of MAP_{Nat}. Next, algorithm updates the entries corresponding to the data in MAP_{Int} with this mapping information. After that, at each iteration, the algorithms select the most constraint column (column which has minimum number of filled entries), fills that column with mapping information respecting the constrains and update that mapping information into other matrix until all the columns of the mapping matrices are filled with mapping information.

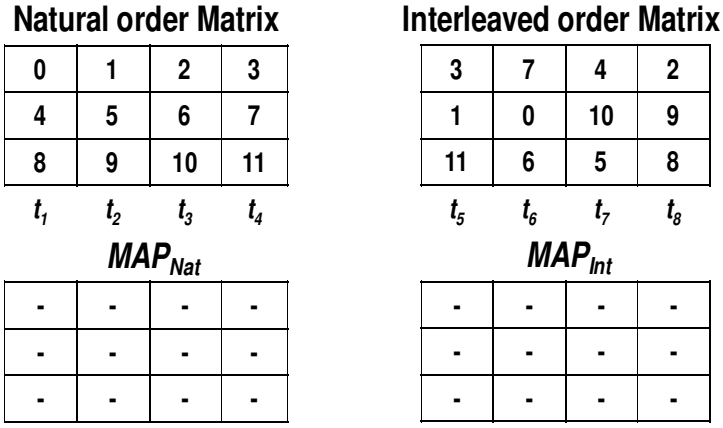


Figure 2. 9. Matrices used in *SAGE*

In this thesis, we present different approaches based on bipartite graph to tackle mapping problem and explore the design space exploration for interleavers used in current telecommunication standards.

2.3.2. Design Time Conflict Resolution for LDPC Codes

To provide flexibility at reasonable hardware cost, algorithm innovation is introduced at design time to map edges of the tanner graph in such a manner that all VNs and CNs communicate their messages without any conflict in partially parallel architecture. There are two benefits in using this approach: First, algorithms proposed for conflict resolution can be applied to any current and future classes of LDPC codes and second the approach does not require any specific network or extra memory elements to manage message communication. Rather any network which supports all the permutation patterns between inputs and outputs can be used to route any Tanner Graph.

Although it is claimed in [TAR04] and [LIN10] that approaches presented in these articles can be used for LDPC but as it is shown in next section through conflict graph that single memory mapping is not always possible for every class of LDPC codes and double memory mapping is the

only solution to map any LDPC Tanner graph. Furthermore, algorithms presented in [TAR04] and [LIN10] are based on a metaheuristic called simulated annealing and the time of completion of algorithm is not known. Memory mapping algorithm in [TAR04] is implemented in [QUA06] that requires two crossbar or Benes networks to manage scrambling of data between different processors. One crossbar is used to transfer messages to VN processors and other to CN processors. Traditional Belief Propagation algorithm is modified into Low-Traffic Belief Propagation Algorithm (LTBPA) to use only one interconnection network but the LTBPA is not the standard algorithm used in current decoder implementations.

In [CHA10], a technique based on multiple read, multiple write approach is presented. In this technique, each data d_i has two mapping, one from where approach reads the data and the second in which the approach writes the data. To accommodate this mapping approach, two additional cells called mapping cells are associated with each d_i. MAP matrix used in this technique is shown in Figure 2. 10.

For functional correctness, if data is accessed several times, then i^{th} read access of d_i is the same as the $(i-1)^{th}$ write access of d_i. Two additional mapping constraints are added in this approach to find conflict free memory mapping: First is that in any column all the memory banks are different and second is that first read and last write of d_i should be performed in same bank.

1	3	6	5	4	2
2	5	1	6	3	1
3	6	4	2	5	4

Figure 2. 10. MAP matrix for Multiple Read Multiple write (MRMW) approach

Algorithm initializes by assigning read and write memory banks to the first column of the *MAP* matrix. Algorithm then continues assigning read and write mapping to the data in the next columns respecting mapping constraints until some conflicts occur in read or write mapping at some column. In that case, the algorithm performs recursion to go to the nearest occurrence of the conflicted data in order to change the mapping and remove the conflict. This process continues until *MAP* matrix is filled with mapping respecting the mapping constraints. The algorithm presents novel idea to solve computationally hard problem through multiple read and multiple write technique but it is based on recursive approach and the time of completion is unknown.

2.4. Conclusion

Three techniques presented in this section have their own merits and demerits. The code developed using first technique results in low area overhead and easy implementation. But this technique is not always standard compliant and codes developed using this technique do not always gives good error correction performance. Flexible and scalable interconnections networks developed during second technique can handle memory conflict for any type of interleaver. However, they suffer

from large silicon area and increased latency due to conflict management mechanism developed in these networks. As a result, these networks are not a suitable solution for high data rate and low power applications. A modification in above mentioned approaches can be made by finding conflict free memory mapping that respects routing structure of particular interconnection network. This removes the need for designing complex router and buffer control architecture and reduces the silicon area and cost to design flexible interconnection network. Third approach deals with this idea to allocate data in memory banks in such a manner to avoid memory conflict problem either using particular network or the network that supports all the permutations at the cost of some preprocessing.

In this thesis, we present polynomial time algorithm to solve mapping problem for any type of interleaver using novel modeling approaches based on graph theory in Chapter 3 and 4.

3. Node and Edge Coloring of Graph

Graphs are among the most widely used models of both human and natural-made structures. They can be used to model many types of relations and process dynamics in physics, chemistry, biology, computer science, sociology and network analysis. Many problems of practical interest can be modeled as graphs.

In this section, the definitions and algorithms related to graph theory are presented which are helpful to understand the approaches presented in the next chapters for designing conflict free parallel interleaver. Also the previous techniques to model memory mapping problem as node and edge coloring are also presented in this portion and explained why these techniques do not give optimal solution in designing parallel interleaver architecture. Finally, bipartite edge coloring algorithms are presented to understand the memory mapping algorithms proposed in this thesis.

3.1. Graph Theory

Definition ***Graph***

A *graph G = (V, E)* is a set of nodes *V,* and a set of edges *E*. If $v,w \in V$ then an edge $(v,w) \in E$ is *incident* to *v* and to *w*, and vertices *v* and *w* are said *adjacent*. A *subgraph* of *G* is a graph whose vertices and edges are in *G*. (see Figure 2. 11.a)

Definition ***Deletion of edge from Graph***

Deletion of edge (v,w) from *G* means to form the subgraph *G – (v,w)*, consisting of all vertices of *G* and all edges of *G* except *(v,w)*.

Definition ***Edge Coloring of Graph***

An edge coloring of *G* is an assignment of a color to each edge in *G*. An *edge chromatic number*, $\chi'(G)$, is the fewest number of colors necessary to color each edge of a graph so that no two edges incident to the same vertex have the same color.

Definition ***Degree of Graph***

The *degree* of vertex *v* is the number of edges incident to *v*. A graph is *deg-regular* if all vertices have the same degree *deg*. A graph is *semi regular,* if all the vertices in any of its vertex set have the same degree.

Definition ***Path in a graph***

A *path P* is a sequence of edges (v_1, v_2), (v_2, v_3),..., (v_{n-1}, v_n). The ends of *P* are vertices v_1 and v_n. If $v_1 \neq v_n$, *P* is *open*; otherwise *P* is *closed*. A *cycle* or *circuit* is a closed path, with no repeated vertices other than the starting and ending vertices. If the number of edges in a circuit is even then it is called *even circuit* otherwise it is called *odd circuit*.

Definition ***Eulerian Circuit and Connected Graph***

An *Eulerian circuit* is a closed path which uses all edges precisely once. A graph *G* is *connected* if there is a path between any two distinct vertices otherwise *G* is disconnected.

Definition ***Matching and Perfect Matching***

A *matching M* in a graph is a set of edges without common vertices. A *perfect matching* M_p is a matching which matches all vertices of the graph i.e., every vertex of the graph is incident to exactly one edge of the matching. (see Figure 2. 11.b)

(a) Simple Graph (b) Perfect Matching with bold lines

Figure 2. 11. Graph Representation

Definition ***2-matching and 2-Factor of Graph***

A *2-matching H* of a graph *G = (V, E)* is a subset of *E* such that every node of *G* is incident with at most two edges of *H*. The 2-matching *H* of *G* is called *2-factor* if every node of *G* is incident with *exactly two edges* of *H* [HAR06].

The following two theorems [GRO03] define the necessary and sufficient condition for the graph to contain 2-factor.

Theorem 2.1

Every *2k-regular* graph contains a 2-factor, where *k* is integer.

Theorem 2.2

Every 2-edge-connected *(2k + 1)-regular* graph contains a 2-factor.

These theorems result into the following two corollaries [GRO03]:

Corollary 2.1

Every *2k-regular* graph contains *k* 2-factors.

Corollary 2.2

Every *(2k + 1)-regular* graph contains *k* 2-factors and one 1-factor.

3.2. Conflict Graph

In [KEY01], memory mapping problem is modeled as conflict graph. In conflict graph all the data elements are modeled as nodes and edge is incident between two data nodes if the corresponding data elements need to be accessed in parallel.

Definition ***Conflict Graph***

The Conflict Graph $G(N,E)$ is a graph with node set $N = \{n_1,n_2, \ldots , n_n\}$ and is defined as follows:

(a) $n_i \epsilon N$, node n_i corresponds to the data element used in the computation, where $i = \{1,2,\ldots,n\}$

(b) $e_{ij} \epsilon E$, edge e_{ij} is incident between node n_i and node n_j if data elements i and j are accessed concurrently by system.

After the construction of conflict graph, the mapping problem is converted into famous node coloring problem which is defined as:

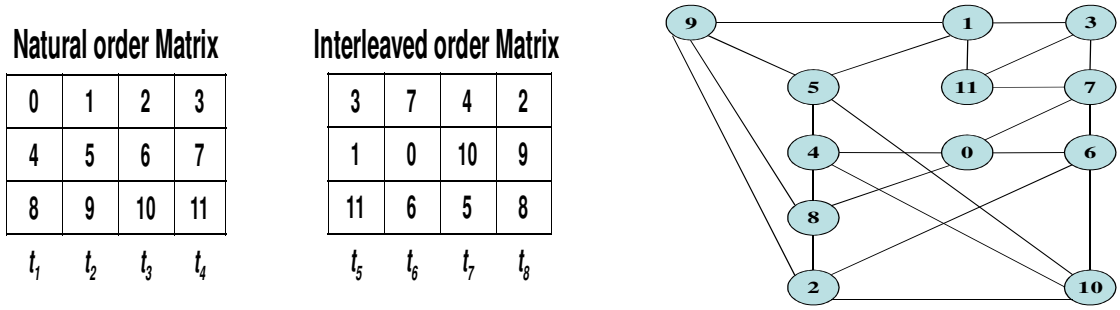
Problem ***Node Coloring Problem***

Given a graph G, can the nodes of G be colored with n colors, provided that the nodes connected with same edge must be given a different color? where n is the minimum number of colors required to color the nodes of G.

If we consider n colors as n memory banks which can be accessed in parallel then the problem of assigning data elements in memory banks in such a way that no conflict occurs during execution of parallel architecture is essentially the problem of node coloring. Since [KOZ92] proves node coloring as NP-complete, modeling memory mapping problem as conflict graph does not give any approach that solves mapping problem in polynomial time. In this thesis, mapping problem is modeled as bipartite graph and different algorithms are presented to solve the problem in polynomial time.

3.2.1. Conflict Graph for Turbo Codes

To show the complexity of memory mapping problem for turbo codes, a conflict graph based on natural and interleaved order of turbo codes (see Figure 2. 12.a) is presented in Figure 2. 12.b. In conflict graph, the number of nodes is equal to the number of data elements used in the computation. Data in each column of natural and interleaved order matrices are needed to be accessed in parallel and connected through edges in conflict graph. For example, data 0, 4, 8 , required to be accessed in parallel at t_1 are connected through edges in conflict graph. Although in [TAR04] and [LIN10], authors proves that it is always possible to find conflict free memory mapping for any type of interleaver used in turbo codes, the proposed heuristics are unable to provide polynomial time algorithms to solve the problem. The proposed architecture for turbo decoder after finding conflict free memory mapping is shown in Figure 2. 13. In this thesis, we not only prove that conflict free memory mapping always exist for every type of interleaver but also present polynomial time algorithm to solve this problem by modeling it as bipartite graph.



(a) Data Access Matrices (b) Conflict Graph

Figure 2. 12. Conflict Graph for Turbo Decoder

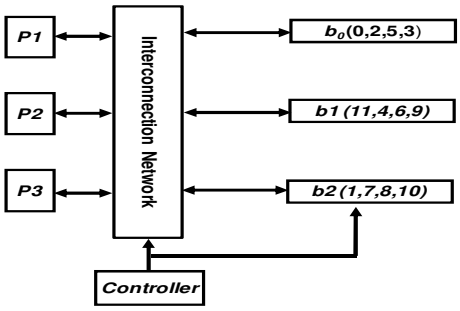


Figure 2. 13. Resultant architecture

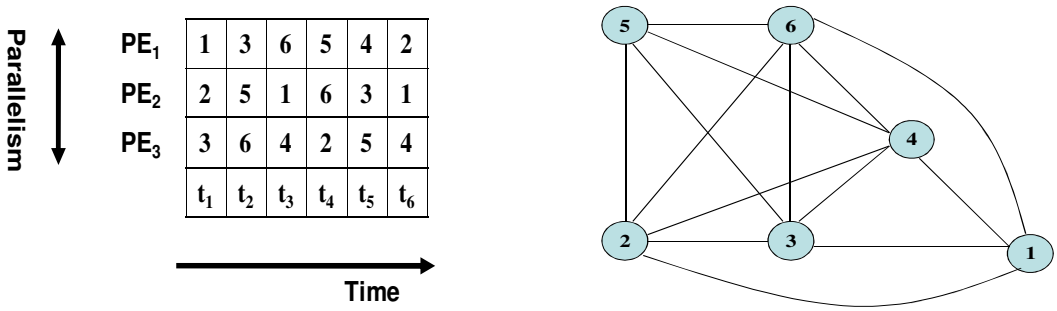
3.2.2. Conflict Graph for LDPC Codes

A conflict graph for LDPC based on access order of Figure 2. 14.a is depicted in Figure 2. 14.b. It is impossible to determine optimal node coloring for the conflict graph of LDPC because the resultant graph *G* itself depicts that we need more than *P* colors or memory banks, where *P = Number of processing elements in the system*, in order to color the nodes of the *G*. This results in an architecture in which more than *P* memory banks are used to provide conflict free concurrent parallel access to *P* processing elements. After finding node coloring, the resultant architecture for conflict graph of Figure 2. 14.b. is shown in Figure 2. 15. in which 5 memory banks are needed to store 6 data elements whereas three processors are used to process the data. This result in following two disadvantages for the modeling based on conflict graph.

1. There is no known algorithm which can find minimum node coloring for the conflict graph of LDPC because the problem in NP. Also the algorithms of [TAR04] and [LIN10] are not applicable in LDPC.
2. If one can be able to find some heuristic like [TAR04] and [LIN10], then still the resultant architecture increases the cost and complexity of the system since we require more memory banks than needed to store the data.

To find memory mapping with optimal number of memory banks, we introduce in this thesis a concept called "*Double Memory Mapping*" in which a memory mapping of each data element is divided into two memory mapping: First mapping called *read mapping* represents read access to that data element whereas second mapping called *write mapping* expresses write access of that data element. In this thesis, we not only model *Double Memory Mapping* as bipartite or tripartite graph but

also propose polynomial time algorithm to find this mapping in which resultant number of memory banks are equal to the number of processing elements to reduce the complexity of the system.



(a) Data Access Matrix (b) Conflict Graph

Figure 2. 14. Conflict Graph for LDPC Decoder

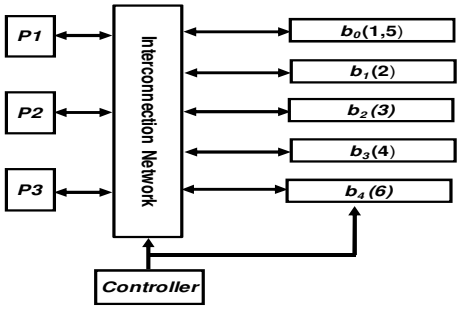


Figure 2. 15. Resultant Architecture

3.3. Bipartite Graph

Computational complexity of node coloring motivates us to model our problem in a class of graphs in which polynomial time algorithms exists. Bipartite graph belongs to this category in which polynomial time algorithms exit for different graph problems. In this section, we present some definitions and algorithms related to the bipartite graph which are used in the next chapters to model and solve memory mapping problems.

Definition ***Bipartite Graph***

A graph $G = (V_1 \cup V_2, E)$ is *bipartite*, if V_1 and V_2 divide the vertices set so that each edge is incident to a vertex in V_1 and a vertex in V_2 i.e. $V_1 \cap V_2 = \varnothing$ (as shown in Figure 2. 16.a). A bipartite graph is called multi-graph if any two of its vertices may be connected through more than one edge.

As explained previously, a *circuit* is even if there are even number of edges in its respective closed path. It is important to note that a *circuit* is always even in bipartite graph.

Definition ***Semi Regular Bipartite Graph***

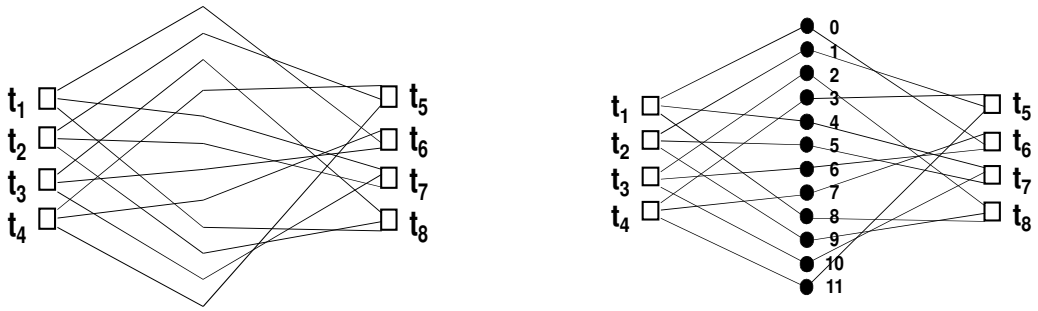
A bipartite graph is *semi regular,* if all the vertices in any of its vertex set V_1 or V_2 have the same degree.

Definition ***semi 2-factor in Bipartite Graph***

A *semi 2-factor* in bipartite graph *G* is defined as a *2-regular* subgraph in *G* with *2Y* vertices where every node is incident with exactly two edges and where Y = Min ($|V_1|$, $|V_2|$).

Definition ***Tripartite Graph***

A graph $G = (V_1 \cup V_2 \cup V_3, E)$ is *tripartite*, if *a set of graph vertices decomposed into three disjoint sets such that no two graph vertices within the same set are adjacent* i.e. $V_1 \cap V_2 \cap V_3 = \varnothing$ (as shown in Figure 2. 16.b).



(a) Bipartite Graph (b) Tripartite Graph

Figure 2. 16. Graph Representation

3.3.1. Bipartite Edge Coloring

Bipartite edge coloring is the procedure to color the edges of bipartite graph and can be presented as follows:

Lemma 2.1 ***Bipartite Edge Coloring***

If the maximum vertex degree of a bipartite graph *G* is Δ then, $\chi'(G) = \Delta$ or the edges of *G* can be colored with exactly Δ colors. Proof of this algorithm can be found in [KON16].

After the presentation of Lemma 2.1, different algorithms have been proposed to color the edges of bipartite graph with Δ colors. In this regard, Vizing [ORE67] first proposed an algorithm to color the edges in $O(VE)$ by repeatedly creating altering paths until all the edges are colored. This algorithm is improved by Gabow [GAB76] which introduces the concept of Euler partition to equally divide each graph of Δ degree into two subgraphs of $\Delta/2$ degree. By recursively applying Euler partition, the algorithm can find edge coloring in $O(\sqrt{V}\ E \log \Delta)$ time. Both of these algorithms are presented next.

3.3.1.1 Vizing Method to color the edges of Bipartite Graph

The algorithm to find bipartite edge coloring is first presented by Vizing [ORE67] which uses alternating path to color the edges of the graph and has a complexity of $O(VE)$ where *V* is total number of vertices and *E* is the total number of edges in the graph.

The algorithm starts by assigning one of Δ possible colors to all the edges of the graph *G*. After the initial assignment, uncolored edge (a,b) is colored as follows:

Suppose a color *x* is missing on vertex *a* and a color *y* is missing at vertex *b*. Construct an "alternating *(x,y)* path *p*" starting at *b*. The path begins with an edge incident to *b* and have a color *x*. Path *p* is constructed by alternately choosing *x* and *y* colored edges until path reaches at the vertex *c* where the next color is missing. It is clear that if the graph is bipartite then $c \neq a$ & $c \neq b$.
After the construction of *p*, colors are interchanged along *p* by switching *x* to *y* and *y* to *x*. This makes color *x* missing at both *a* and *b* and edge *(a,b)* can now be colored with color x.
To explain the algorithm, we present a simple example. Initial coloring of Graph *G* with $\Delta = 3$ colors namely bold, dotted and gray as shown in Figure 2. 17.a. Gray color is missing at vertex *4* whereas dotted color is missing at vertex *F*. So, algorithm constructs (*gray, dotted*) alternating path starting from *F* until the path reaches at vertex *A* where gray color is missing as shown in Figure 2. 17.b. Alternating path is shown through long dotted lines in this figure. After the construction of alternating path, the algorithm interchanges the color on the path (as shown in Figure 2. 17.c) so that gray color becomes missing on both vertices *F* and *4*. Edge (*F,4*) is now assigned with gray color to complete the edge coloring as shown in Figure 2. 17.d.

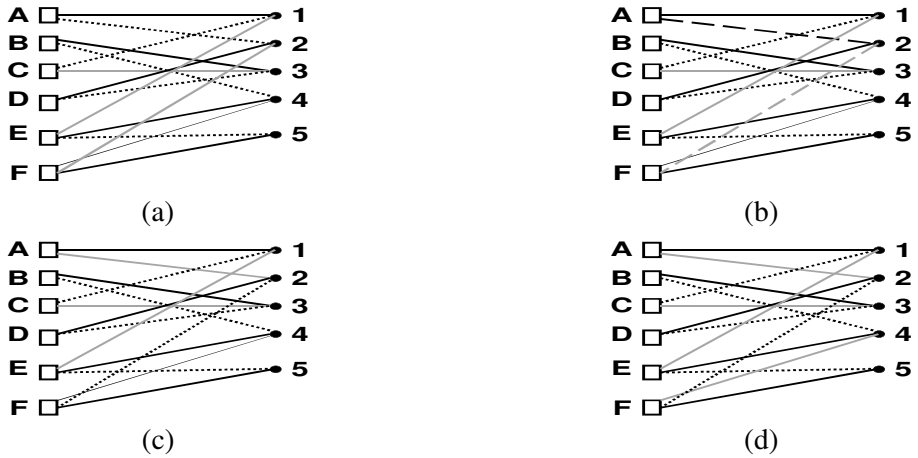


Figure 2. 17. Vizing Algorithm

3.3.1.2 Gabow Method to color the edges of Bipartite Graph

First breakthrough algorithm to reduce the complexity of bipartite edge coloring algorithm is presented by Gabow in [GAB76]. The algorithm uses Euler partition to divide original graph *G* of maximum vertex degree Δ into subgraphs G_1 and G_2 where maximum vertex degree of each subgraph is $\Delta/2$. By recursively applying Euler partition, the algorithm colors edges of *G* in $O(\sqrt{V}\ E \log \Delta)$ time. But first we explain algorithm to find Euler partition because all the presented algorithms use this algorithm to reduce the complexity of coloring the edges of bipartite graph.

Euler Partition

An Euler partition is a technique that partitions the edges of graph into open and closed paths. Each vertex of odd degree is the end vertex of one open path where as each vertex of even degree is not the end vertex of any open path. An Euler partition is constructed as follows:

Choose any vertex of odd degree, if no vertex of odd degree exists then choose any vertex of even, nonzero degree. Start traversing a graph from one vertex to another by including a traversed edge into the path *p* and removing that edge from the graph until a vertex with zero degree is reached. This completes *p* in the partition. Then start constructing another path *p'* by choosing another start vertex. Repeat this process until no vertex of nonzero degree remains. The algorithm finds euler partition in $G = (V_1 \cup V_2, E)$ in $O(E)$ time. Procedure to find Euler Partition is presented in [GAB76].

Edge Coloring Algorithm used by Gabow in [GAB76]

After the construction of Euler partition, a bipartite graph *G* splits into two bipartite graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ by traversing each path of the partition and alternately placing one edge into G_1 and one edge into G_2. Every even degree vertex of *G* will have the same degree in both G_1 and G_2 but every odd degree vertex of *G* will have degree in G_1 and G_2 differing by one. This implies that if Δ is even in *G* then maximum vertex degree in both G_1 and G_2 is $\Delta/2$ but if the Δ is odd then maximum vertex degree in both G_1 and G_2 is $\lceil \Delta/2 \rceil$ or $\Delta+1$ which results in a coloring of *G* with $\Delta+1$ colors which is not minimal.

To avoid this, if Δ is odd, then the algorithm first finds matching *M* that covers all the vertices of degree Δ, give one color to the edges of *M* and delete these edges from *G*. Now Δ is even, so the Euler partitioning method can be applied to *G-M*. The complete edge Coloring algorithm is presented below.

```
1   edgeColor (G, Δ )
2       if (Δ is odd) then
3           if (Δ = 1) then
4               M = G
5               assignOneColour(M )
6           else
7               M = Matching(G)
8               assignOneColour(M )
9               edgeColour(G- M, Δ-1)
10          end if
11      end if
12      if (Δ is even) then
13          C_euler = eulerCircuit(G)
14          G1, G2 = split(G, C_euler)
15          edgeColour(G1, Δ/2)
16          edgeColour(G2, Δ/2)
17      end if
```

Edge Coloring algorithm used in this thesis with modified euler partitioning method and pedagogical example is presented in Chapter 4.

The algorithm to find matching (see section 3.1) that covers all the vertices of degree Δ takes $O(\sqrt{V}\ E)$ time and the complete algorithm to find bipartite edge coloring takes $O(\sqrt{V}\ E \log \Delta)$ time. This implies that in the algorithm most of the time is consumed in constructing the matching because Euler partitioning is completed in $O(E)$ time. Therefore, after the work of Gabow, all the future algorithms tried to reduce the complexity of finding matching in bipartite graph. It has been found that the algorithm to find perfect matching in regular bipartite graph is more efficient than the algorithm to

find matching in irregular bipartite graph which covers all the vertices of degree Δ. So, the current algorithms first convert irregular bipartite graph of maximum vertex degree Δ into Δ-regular bipartite graph through a simple method which is presented below.

Converting irregular graph into Δ-regular graph

Initially the bipartite graph G consists of two sets of vertices called $V_1 \cup V_2$. The procedure [COL82] to construct Δ-regular graph is completed in two steps.

Merge Step

In the first step, vertices of degree < Δ are merged, if possible. To do this, both sets of vertices are sorted out by their degree. Then, for each vertex set, the procedure starts merging the vertices until at most one vertex of degree < ½ Δ remains in each vertex set.
If the resultant graph is Δ-regular, then we apply our bipartite edge coloring algorithm, otherwise second step is executed to construct Δ-regular bipartite graph.

Copy Step

In the second step, procedure makes a copy G' of G and put the two graphs together by placing vertex set V_1' of G' into vertex set V_2 of G and vertex set V_2' of G' into vertex set V_1 of G. The new graph G'' has the same number of vertices in both V_1'' and V_2'' and for each vertex v in G, its copy v' is in the opposite vertex set in G''. Both vertices v and v' have the same degree *deg*, so procedure joins them with $\Delta - deg$ edges to construct Δ-regular graph G'''. Edge coloring algorithm is then applied on G''' and G is extracted from G''' after coloring. It is important to note that edge coloring is still applicable on G after we retrieve it from G'''.

The procedure is explained in Figure 2. 18. Vertices D and E are merged in merge step (see Figure 2. 18.b) whereas in copy step we make a copy of merged graph and place the two graphs together in such a way that the vertices of one set is placed along with the vertices of the other set (see Figure 2. 18.c). Vertices B & B' and C & C' have equal degree i.e., 3, so we join B & B' and C & C' by two edges to construct Δ-regular graph as shown with dotted lines in Figure 2. 18.d.

Fortunately, in our modeling of mapping problem as bipartite graph, the degree of vertices depends on the degree of parallelism that is always constant in hardware architecture. So bipartite graph for mapping problem is always Δ-regular graph where Δ is the number of processing elements.

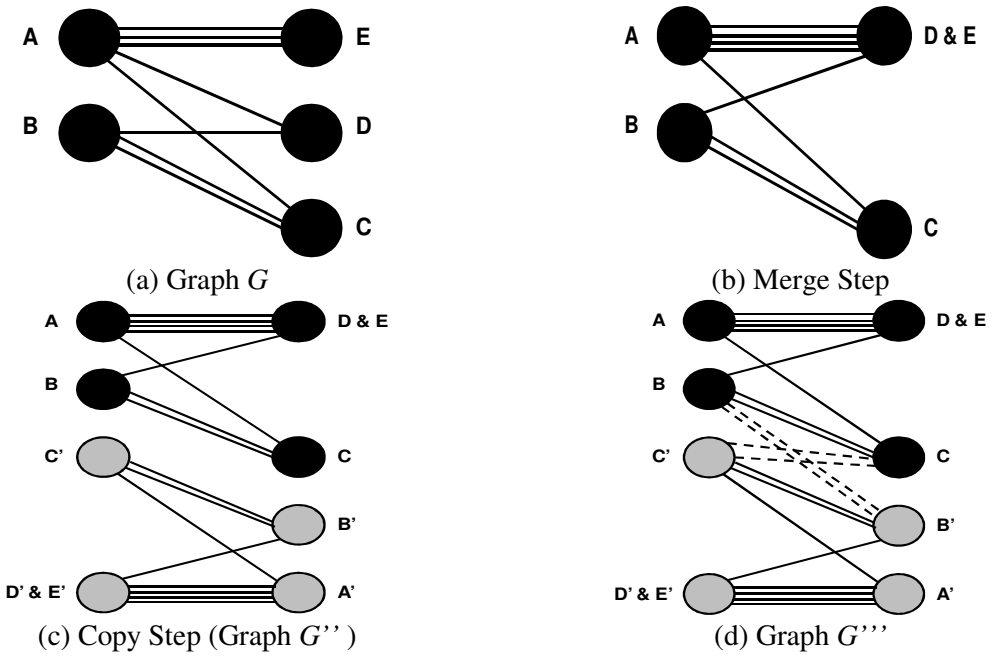


Figure 2. 18. Constructing Δ-regular graph

4. Transportation Problem

One of the key problem in the current manufacturing and in service organization is how to allocate scarce resources between various projects. In terms of resource allocations, Linear Programming (LP) is a method for allocating limited resources in order to reduce the cost of the project. In LP, limited resources are called *decision variables*, the criterion for selecting the best values of the decision variables are called *objective function* whereas limitations on resource availability are known as *constraint set*.

Transportation Problem [BAZ97] is a class of problem of LP which deals with the physical distribution of resources between producers and consumers. *The transportation problem is to minimize the cost of shipping item from producers to consumers so that each consumer fulfills its demand and every producer operates within its capacity.*

To understand the problem, consider a set of I producers where producer i has a supply of a_i units of a particular item. In addition, there are J consumers where consumer j requires z_j units of item. We assume that $a_i, z_j > 0$. For each link from producer i to consumer j, l_{ij}, let o_{ij} be the cost to transport a single item on l_{ij}. Transportation problem can be modeled in two ways: First in network model and second in matrix model. In network model, transportation problem is represented as bipartite graph, as shown in Figure 2. 19, where first node set contains all the producers and second node set contains all the consumers. An edge is connected between a producer i and consumer j if a route l_{ij} exits between them. Also the cost to transport one item on each route is mentioned on the corresponding edge. Supply of all the producers and demand of all the consumers are also mentioned on the corresponding node of the bipartite graph. In matrix model, transportation problem is represented as matrix, as shown in Figure 2. 20, in which rows contain all the producers and columns represent all the consumers. Each cell M_{ij}, corresponding to the producer i and consumer j, contains o_{ij} of the route l_{ij}.

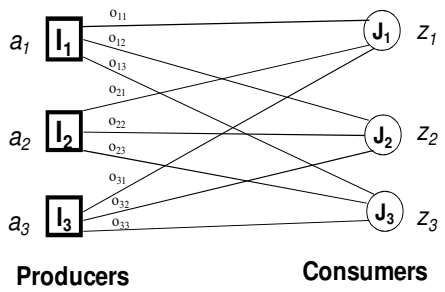


Figure 2. 19. Network Model of Transportation Problem for 3 producers and 3 consumers

Consumer / Producer	J_1	J_2	J_3	Supply
I_1	o_{11}	o_{12}	o_{13}	a_1
I_2	o_{21}	o_{22}	o_{23}	a_2
I_3	o_{31}	o_{32}	o_{33}	a_3
Demand	z_1	z_2	z_3	

Figure 2. 20. Matrix Model of Transportation Problem for 3 producers and 3 consumers

Transportation problem can be expressed in linear programming as follows:

Objective Function is to reduce the cost of transporting items from each producer to each consumer.

If x_{ij} is the number of items transported on l_{ij} then this objective function can be expressed mathematically as:

MIN ($o_{11}x_{11} + o_{12}x_{12} + \ldots\ldots\ldots\ldots\ldots\ldots + o_{i\,(j-1)}x_{i\,(j-1)} + o_{ij}x_{ij}$)

whereas the constraints for transportation problem are represented as:

$$x_{11} + x_{12} \ldots\ldots\ldots\ldots\ldots\ldots + x_{1j} \geq a_1 \qquad \text{Supply of Producer } I_1$$
$$x_{21} + x_{22} \ldots\ldots\ldots\ldots\ldots\ldots + x_{2j} \geq a_2 \qquad \text{Supply of producer } I_2$$
$$\vdots$$
$$x_{i1} + x_{i2} \ldots\ldots\ldots\ldots\ldots\ldots + x_{ij} \geq a_i \qquad \text{Supply of Producer } I_i$$
$$x_{11} + x_{21} \ldots\ldots\ldots\ldots\ldots\ldots + x_{i1} \geq z_1 \qquad \text{Demand of Consumer } J_1$$
$$x_{12} + x_{22} \ldots\ldots\ldots\ldots\ldots\ldots + x_{i2} \geq z_2 \qquad \text{Demand of Consumer } J_2$$
$$\vdots$$
$$x_{1j} + x_{2j} \ldots\ldots\ldots\ldots\ldots\ldots + x_{ij} \geq z_j \qquad \text{Demand of Consumer } J_j$$

An example to explain transportation problem is presented in Annexure.

5. Conclusion

In this chapter, different techniques that were already presented in literature to handle memory mapping problem for turbo and LDPC codes have been explained. The easiest solution is to develop an interleaving law taking into account architectural constraints. This results in a code that supports the implementation of decoder on parallel architecture. However, codes developed using conflict free interleaving law do not always present good error correction capabilities and this is not a desirable tradeoff to simplify decoder implementation. To add flexibility in decoder implementation, different architectures based on System on Chip (SoC) that supported any type of interleaving law have been presented in literature as a second technique to tackle memory mapping problem. However, large hardware cost and latency in implementing this architecture limit its use in practical system. Third technique is to develop algorithms that can map data in memory banks in such a manner that all processing elements can access their required data concurrently from memory without any conflict. This technique solves memory mapping problem for any type of interleaving law and provided medium complexity in decoder implementation.

Afterwards, two algorithms have been explained through examples: first algorithm is related to the coloring of the edges of bipartite graph and second one is related to the solving of transportation problem. Bipartite edge coloring can be solved in polynomial time. Different techniques and approaches to solve this problem efficiently have been presented with simple examples in this chapter. Transportation Problem is a class of linear programming problem that deals with optimal distribution of resources between producer and consumers. This is another problem that could be solved efficiently in linear time. In this chapter, Modeling of bipartite graph as transportation problem has been discussed and an efficient algorithm has been presented to find optimal allocation of resources between producer and consumer.

In this thesis work, different algorithms are presented to tackle memory mapping problem for turbo and LDPC codes. Two last algorithms presented in this chapter have been used to propose several approaches that allow designing parallel hardware architecture. This work allowed to define finally a method that find in polynomial time a solution to the memory mapping problem. Next two chapters present related algorithms and highlight their behaviors on pedagogical examples. Proposed methods are compared with state of the art solutions in the last chapter and interests are discussed through experiments.

Chapter 3
METHODS BASED ON BIPARTITE GRAPH FOR SOLVING MEMORY MAPPING PROBLEMS

Table of Contents

1. Introduction 57

2. A Methodology based on Transportation Problem Modeling for Designing Parallel Interleaver Architecture 57

2.1. Problem Formulation for Memory Mapping Problem of Turbo codes----------58
2.2. Modeling ----------59
2.2.1. Construction of Bipartite Graph ----------59
2.2.2. Transformation of bipartite graph into Transportation Matrix ----------60
2.3. Algorithm to find semi 2-factors in Turbo Bipartite Graph----------61
2.4. Pedagogical Example to explain algorithm ----------63

3. Methodologies Based on Bipartite Graph to find conflict Free Memory Mapping for LDPC ----------65

3.1. Problem Formulation for Memory Mapping Problem of LDPC codes ----------66
3.2. Modeling ----------67
3.3. Algorithm ----------68
3.3.1. Partition the Bipartite Graph ----------69
3.3.2. Coloring edges of Bipartite Graph ----------70
3.4. Pedagogical Example to explain algorithm ----------71

4. Conclusion ----------75

In this chapter, first two approaches that are developed during this thesis work to tackle memory mapping problem, are presented. In both of these approaches, the mapping problem is modeled as bipartite graph and then each graph is divided into different subgraphs in order to facilitate the coloring of the edges. First approach deals with turbo-like codes and uses transportation problem algorithms to divide the bipartite graph. The approach can also find memory mapping that supports particular interconnection network if interleaver of the application allows doing it. Second approach solves memory mapping problem for LDPC codes and uses double memory mapping technique to store data in optimal memory banks.

1. Introduction

As explained in Chapter 2, all the approaches to model memory mapping problem as a conflict graph proves to be inefficient because of the unavailability of polynomial time algorithm to find node coloring. This seems to be consistent with the already developed NP-Complete Theory which states that node coloring of the conflict Graph is NP-complete Problem. Heuristics proposed for node coloring are efficient for some particular applications but fails to give optimum solution in other domains. Also it is difficult to add other constraints in these heuristics to find architecture-oriented memory mapping that is necessary to reduce cost and area of the system.

The basic purpose of this thesis is to transform our memory mapping problem into the class of problems in which polynomial time algorithms already exists. Bipartite Edge Coloring is one of such problems in which polynomial time algorithms exits to find color of the edges of the bipartite graph. In this chapter, we propose two approaches which are based on bipartite graph model to solve memory mapping problem. These methods are the first two approaches used to solve memory mapping problem in this thesis work.

The first approach is applied on slightly simple mapping problem for turbo-like codes. In this approach, mapping problem is transformed into transportation problem and then linear programming approach is used to find conflict free memory mapping. In the second approach, memory mapping problem for double memory mapping is tackled. Time instances and data elements represent two sets of nodes of bipartite graph. Edge is connected between time node and data node to represent the access order at that time instance. To simplify edge coloring, each graph is partitioned into different subgraphs and each subgraph is colored independently to find conflict free memory mapping.

2. A Methodology based on Transportation Problem Modeling for Designing Parallel Interleaver Architecture

In this section, we present an algorithm for special cases in which data is accessed only two times such as in turbo codes and non-binary LDPC. Simple data access pattern of this problem allows us to find "*Single Memory Mapping*" or "*in place Memory Mapping*" in which data is read and write from the same memory bank. This simplicity motivates us to model the problem as "Transportation Problem" and then uses linear programming approach to solve it. This work has been presented in 36th International Conference on Acoustics, Speech and Signal Processing, 2011 [SAN11a].

However, before presenting algorithm to solve mapping problem, it is necessary to formulate our problem and constraints so that one can easily understand the algorithms presented next. Memory mapping problem is same for both turbo and LDPC that we want to access data elements concurrently without any conflict in both read and write accesses. However, depending on the pattern of access of data elements, LDPC (excluding non-binary LDPC) memory mapping problem is more complex than turbo mapping problem as explained in Chapter 2. Therefore, we formulate these problems separately.

In this section, memory mapping problem is formulated for turbo codes whereas in the next section the problem is formulated for LDPC.

The section starts by formulating memory mapping problem for turbo codes. Afterward, we model our problem as bipartite graph along with some proves to show that it is possible to partition this graph into different subgraphs of equal sizes. Next we transform this graph into transportation Problem and apply linear programming approach to find each subgraph separately along with memory mapping. The approach also tries to simplify the interconnection network if interleaver of the application allows doing it.

2.1. Problem Formulation for Memory Mapping Problem of Turbo codes

To explain the problem, consider a set of D data elements $\{d_1, d_2,..., d_D\}$ and a set of P processing elements $\{PE_1, PE_2,..., PE_P\}$ which process these D data elements first in natural order and then in interleaved order in T time instances $\{t_1, t_2,..., t_T\}$, where $T = 2D/P$.
In order to store these D data elements and to achieve parallel processing of data for high throughput, a set of B memory banks $\{b_1, b_2,..., b_B\}$, where $B = P$, is used. All the memory banks have the same size R which is equal to $R = D/P$.

Mapping problem

Store D data elements in B memory banks in such a manner that P processing elements can access B memory banks in parallel in both natural and interleaved order time instance without any conflict.

Mapping problem can easily be explained using two matrices: one is related to natural order access and the other is to interleaved order access. These matrices together are called data access matrices. Each matrix has P rows, related to the processing elements, and $T/2$ columns, related to the time instances. Data elements in each row are processed by the processing element connected with this row. Similarly, the P data elements in each column need to be accessed in parallel by P processing elements for parallel decoding architecture. Figure 3. 1. depicts two access matrices in which we have $D = 12$, $P = B = 3$, $R = 4$ and $T = 8$.

(a) Natural Order Matrix (b) Interleaved Order Matrix

Figure 3. 1: Data Access Matrices for Turbo Codes

Memory Mapping Constraints and architecture objectives

To successfully map the data (i.e. to allow conflict free parallel memory accesses) in a given number of memory banks, the two following mapping constraints must be fulfilled:

- At each time instance, all the memory banks have to be used one and only one time,
- Each data must be mapped in one and only one memory bank.

To optimize interconnection network, the following architectural objective must be fulfilled if the interleaving allows doing it:

- Resultant memory mapping must follow targeted steering rule (e.g., a barrel shifter).

2.2. Modeling

In this section, we model our mapping problem as transportation problem. This transformation is carried out in two steps. In the first step, mapping problem is modeled as bipartite graph and different proves have been provided in order to explain that it is always possible to divide this bipartite graph into different subgraphs of equal sizes. Afterwards, this bipartite graph is transformed into transformation matrix on which any algorithm to solve transformation problem can be applied to find memory mapping.

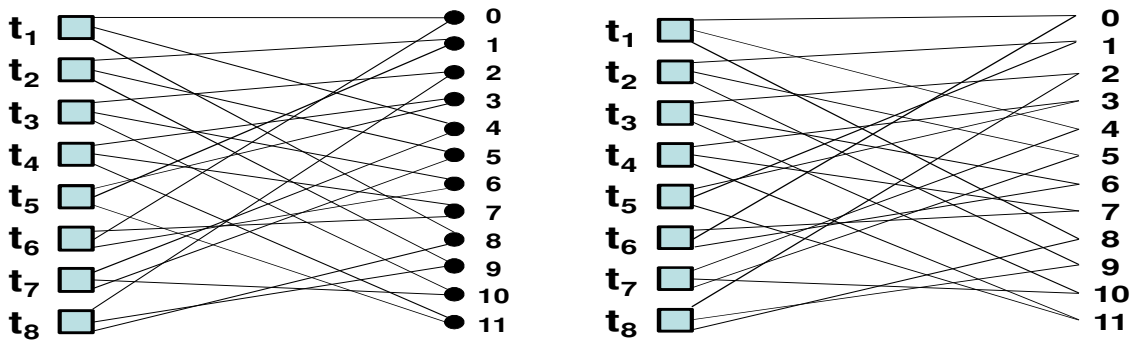
2.2.1. Construction of Bipartite Graph

The first step is to construct a bipartite graph $G = (T \cup D, E)$ in which vertex set T represents all the time instances and vertex set D represents all the data elements used in the computation. An edge (t, d) is incident to the data element vertex d and to the time instance vertex t if d needs to be processed at t (i.e. data d will be read and next written at time t). This bipartite graph is called turbo bipartite graph (*TBG*) due to following distinct properties.

1- *The number of accesses to data or processing elements at any time instance is always identical (the number of PEs is equal to the number of memory banks) which implies that corresponding bipartite graph is always semi regular and each time node has same degree* $f_t = P$.

2- *Each data element is accessed only two times: one in natural order access and the other in interleaved order access. This implies that all the data nodes in the bipartite graph have the same degree,* $f_d = 2$.

TBG for data access matrices of Figure 3. 1 is shown in Figure 3. 2.a.



(a) Turbo bipartite graph for Figure 3. 1 (b) Graph without data nodes

Figure 3. 2: Bipartite Graph representation

As explained in previous chapter, every *2k* or *(2k + 1)-regular* graph contains *k* 2-factors that results in following corollary.

Corollary 3.1

Every Turbo Bipartite Graph with $f_t = 2k$ or $f_t = 2k + 1$, where *k* is an integer, contains *k* disjoint *semi 2-factors*.

Proof: we first join the two edges connected with each data node and then remove all the data nodes to form regular graph $G_1 = (T, E_1)$ as shown in Figure 3. 2.b. In this graph, $|E_1| = |D|$ i.e., each edge in G_1 corresponds to two edges or a data node in *G*. Since G_1 is regular, theorem 2.1 and 2.2 give us the proof that *2-factor* always exists in G_1 which implies that semi 2-factor of 2Y vertices where $|T| = Y$ always exists in *G*. ▪

Every 2-factor is a collection of cycles that spans all vertices of the graph going from 1 cycle with 2Y vertices up to Y cycles of 2 vertices.

Additionally, each cycle c_i in G_1 is even which means c_i contains even number of edges or time nodes because vertex set *T* is divided into two partitions i.e. natural or interleaved orders. Edges of every even cycle can be assigned with two colors which implies that edges in c_i and every 2-factor in G_1 can be colored with two colors. This results in the following lemma.

Lemma 3.3

All the data nodes in semi 2-factors of a turbo bipartite graph can be assigned with two memory banks.

2.2.2. Transformation of bipartite graph into Transportation Matrix

The second step is to divide *TBG* into *k semi 2-factors* and to give two colors to the edges of each semi 2-factor. To find *semi 2-factor*, we transform our mapping problem for turbo codes as transportation problem by considering all the data nodes as producers and all the time nodes as consumers. The route l_{ij} exists between data node d_i and time node t_j if data d_i is accessed at t_j. One additional constraint must be considered while modeling our problem as transportation problem: the capacity of each route is fixed in our mapping problem. The reason is that each route represents a connection between processor and memory banks whose size is always fixed. In our case, the capacity x_{ij} of l_{ij} is kept 1 since only one data can be accessed at a given time instant t_j.
In order to find *semi 2-factor*: (1) the demand of each consumer is kept to 2 and (2) each producer either provides 2 items (i.e. one item for the natural access and one item for the interleaved access) or is not included in the current *semi 2-factor* (i.e. each producer must work at its full capacity). The cost o_{ij} of l_{ij} is kept 1 since the cost is not taken into account in the current version. It is only used when we will consider the complexity of the network architecture. The matrix model for the bipartite graph of Figure 3. 2.a is shown in Figure 3. 3.a. In this matrix, if the route l_{ij} does not exist between producer *i* and consumer *j*, then the corresponding cell M_{ij} is kept empty

To facilitate the construction *of semi 2-factor* respecting a particular interconnection network, the processor P_c which accesses the data d_i at time t_j is placed in M_{ij}. Since all the routes l_{ij} have the same capacity and cost, we remove all the x_{ij} and o_{ij} values from matrix representation and only keep the processors which access data. The concise representation of the Transportation matrix is shown in Figure 3. 3.b.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	
0	1(1) P_1					1(1) P_2			2
1		1(1) P_1			1(1) P_2				2
2			1(1) P_1					1(1) P_1	2
3				1(1) P_1	1(1) P_1				2
4	1(1) P_2						1(1) P_1		2
5		1(1) P_2					1(1) P_3		2
6			1(1) P_2			1(1) P_3			2
7				1(1) P_2		1(1) P_1			2
8	1(1) P_3							1(1) P_3	2
9		1(1) P_3						1(1) P_2	2
10			1(1) P_3				1(1) P_2		2
11				1(1) P_3	1(1) P_3				2
	2	2	2	2	2	2	2	2	

(a) Matrix model of Figure 3. 2.a

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	
0	P_1					P_2			2
1		P_1			P_2				2
2			P_1					P_1	2
3				P_1	P_1				2
4	P_2						P_1		2
5		P_2					P_3		2
6			P_2			P_3			2
7				P_2		P_1			2
8	P_3							P_3	2
9		P_3						P_2	2
10			P_3				P_2		2
11				P_3	P_3				2
	2	2	2	2	2	2	2	2	

(b) Concise matrix model of Figure 3. 2.a

Figure 3. 3: Matrix Model for Transportation Problem of Figure 3. 2.a

After construction of transportation matrix, any algorithm to solve transportation problem can be used to find semi-2 factor. The complete algorithm to solve mapping using transportation problem is explained in the next section.

2.3. Algorithm to find semi 2-factors in Turbo Bipartite Graph

In this section, an algorithm to solve the memory mapping problem is presented that traverse within the transportation matrix to construct semi 2-factors. Our algorithm starts by first calculating the number of *semi 2-factors k* by using the degree of each time node f_t of *TBG* as explained in corollary 3.1. After that, it starts constructing the cycle c_1 of current *semi 2-factor* sf_{cur} by choosing a first route l_{i1} connected with consumer t_1 (see Figure 3. 3). Flow diagram of our partitioning algorithm is shown in Figure 3. 4. The selection of the route l_{i1} decreases the demand of t_1 and the supply of d_i to 1. Bank b_a is also assigned to the route to facilitate the construction of interconnection network respecting target steering rule. Algorithm then selects the route connected with t_1 whose processor identification P_c respects the particular steering rule, if it is possible, otherwise algorithm chooses any route l_{k1} and assigns it with bank b_{a+1}. The selection of l_{k1} completes the demand of t_1, so all the producers connected with t_1 are completely removed from sf_{cur} because now they are unable to provide 2 items in sf_{cur} or they cannot work at their full capacity. The other route l_{km} connected with d_k is assigned the same bank b_{a+1} to reach to the consumer t_m. This completes the supply of d_k. Algorithm repeats the same process and selects the route whose processor identification respects target steering rule, decreases the supply of producer and demand of consumer and alternately assigns banks to the route until c_1 is completed i.e., no producer with supply of 1 and no consumer with demand of 1 remains in the transportation matrix.

At this point, the algorithm tests whether all the consumers fulfill their demands. If not, the algorithm starts constructing another cycle c_2. For this, our algorithm selects consumer whose demand is still unfulfilled and which has at least one deleted route. Using this deleted route, the algorithms selects the route whose selection respects the targeted steering rule (if it is possible) and assigns a bank b_a to this route (selection procedure for steering rule based on barrel shifter is presented in section 2.4). After the assignment of b_a, the algorithm repeats the same process used for the construction of c_1 to complete c_2. When the algorithm finds that demands of all the consumers are fulfilled then it declares that sf_{cur} is constructed. In that case, the algorithm tests whether *k semi 2-factors* are constructed. If not, the algorithm removes sf_{cur} from transportation matrix, initializes all consumers with demand of 2 and starts constructing sf_{next} from remaining matrix using the process described above until *k semi 2-factors* are constructed. Partitioning algorithm is explained through a pedagogical example in next section.

Algorithm needs to traverse f_t edges at each time instance to select two accesses that can be included in sf_{cur}. To construct a partition sf_{cur}, algorithm needs to select a couple of accesses for each time nodes (number of time nodes: |T|). So, the number edges to be traversed for one partition is in the worst case: $f_t * |T|$. Since there are $k = f_t/2$ semi-2 factors (see definition), so in order to construct all the partitions, overall complexity of the partitioning algorithm is $O(f_t/2 * (f_t |T|)) = O(f_t^2 * |T|/2)$. This is the complexity of finding a mapping that respects single barrel shifter that can be used for both natural and interleaved order.

However, in order to test whether two barrel shifters (one for natural order and other for interleaved order) are possible or not, algorithm has to test all possible permutation combinations of two barrel shifters. It traverses $2(f_t!)$ times the graph to confirm whether a "two barrel shifters" based architecture is possible or not, and the overall complexity of the algorithm is $O(f_t^2 * |T|/2)(2(f_t!)) = O((f_t^2 * |T|)(f_t!))$.

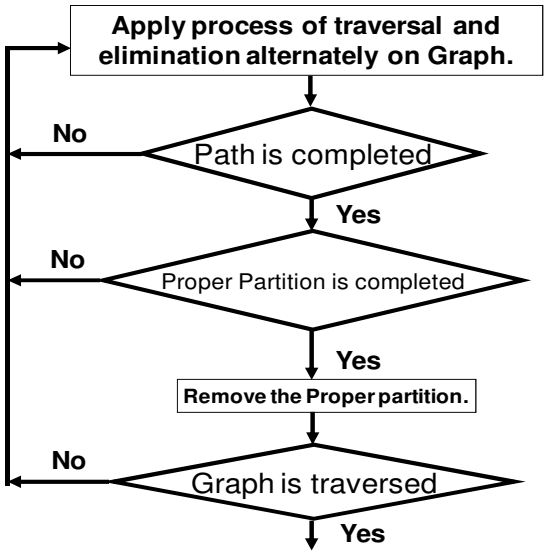


Figure 3. 4: partitioning algorithm

2.4. Pedagogical Example to explain algorithm

Let us present an example based on the natural and interleaved order access matrices presented in Figure 3. 1. The first step is to constructs the bipartite graph which is depicted in Figure 3. 2.a. This semi regular bipartite graph has time vertex with degree f_t is 3. Following corollary 3, we will have one *semi 2-factor* after applying partitioning algorithm. The target architecture is a barrel-shifter based interconnection network. The second step first transforms the bipartite graph into matrix model of the transportation problem which is depicted in Figure 3. 3.b. As already explained all the routes l_{ij}, have the same cost and capacity i.e. 1. Our algorithm starts constructing the cycle c_1 from the first route l_{01} (i.e. from producer d_0 to consumer t_1) and assigns the memory bank b_0 to l_{01}. Since one route of capacity 1 is occupied, the algorithm reduces the supply of d_0 and demand of t_1 to 1 in the matrix as shown in Figure 3. 5.a. Our algorithm then fulfills the demand of t_1 by choosing a processor which follows the targeted steering rule which is barrel shifter in our case. The algorithm selects a route l_{41}, which connects a processor next to P_1 i.e. P_2 and assigns l_{41} another memory bank b_1. Now the demand of t_1 is completely fulfilled and the remaining producers connected with t_1 (d_8 in this case) is completely removed because these producers are unable to work at their full capacity. The second route connected with producer d_4, l_{47}, is also assigned the same bank b_1. All this process is shown in Figure 3. 5.b.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	
0	$P_1(b_0)$					P_2			1
1		P_1			P_2				2
2			P_1					P_1	2
3				P_1	P_1				2
4	P_2						P_1		2
5		P_2					P_3		2
6			P_2			P_3			2
7				P_2		P_1			2
8	P_3							P_3	2
9		P_3						P_2	2
10			P_3				P_2		2
11				P_3	P_3				2
	1	2	2	2	2	2	2	2	

(a) Construction of c_1

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	
0	$P_1(b_0)$					P_2			1
1		P_1			P_2				2
2			P_1					P_1	2
3				P_1	P_1				2
4	$P_2(b_1)$						$P_1(b_1)$		X
5		P_2					P_3		2
6			P_2			P_3			2
7				P_2		P_1			2
8	~~P_3~~							~~P_3~~	~~2~~
9		P_3						P_2	2
10			P_3				P_2		2
11				P_3	P_3				2
	X	2	2	2	2	2	1	2	

(b) Construction of c_1

Figure 3. 5: Assignment of memory banks in Transportation matrix

At this point, the algorithm selects the route based on the bank and processor suitable for supporting barrel shifter rule. Since the current bank and processor is b_1 and P_1 respectively, the algorithm searches the route (i.e. the cell) which contains the processor just before the P_1 i.e. P_3 to assign the bank b_0. Cell M_{57} contains the processor P_3, so the route l_{57} is assigned the bank b_0. The assignment completes the demand of t_7 and the other producer connected with t_7 (d_{10} in this case) is removed from the matrix. The other route connected with d_5, l_{52}, is also assigned the same bank b_0 as shown in Figure 3. 6.a. Now the current bank and processor is b_0 and P_2 respectively and the algorithm searches the cell containing the processor just after the P_2 i.e., P_3. The algorithm finds P_3 in M_{92} and assigns the route l_{92} bank b_1. The same bank b_1 is assigned to other route connected with d_9 i.e., l_{98}. All this process is shown in Figure 3. 6.b.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	
0	$P_1(b_0)$					P_2			1
1		P_1			P_2				2
2			P_1					P_1	2
3				P_1	P_1				2
4	$P_2(b_1)$						$P_1(b_1)$		X
5		$P_2(b_0)$					$P_3(b_0)$		X
6			P_2			P_3			2
7				P_2		P_1			2
8	~~P_3~~							~~P_3~~	~~2~~
9		P_3						P_2	2
10			~~P_3~~				~~P_2~~		~~2~~
11				P_3	P_3				2
	X	1	2	2	2	2	X	2	

(a) Construction of c_1

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	
0	$P_1(b_0)$					P_2			1
1		~~P_1~~			~~P_2~~				~~2~~
2			P_1					P_1	2
3				P_1	P_1				2
4	$P_2(b_1)$						$P_1(b_1)$		X
5		$P_2(b_0)$					$P_3(b_0)$		X
6			P_2			P_3			2
7				P_2		P_1			2
8	~~P_3~~							~~P_3~~	~~2~~
9		$P_3(b_1)$						$P_2(b_1)$	X
10			~~P_3~~				~~P_2~~		~~2~~
11				P_3	P_3				2
	X	X	2	2	2	2	X	1	

(b) Construction of c_1

Figure 3. 6: Assignment of memory banks in Transportation matrix

The algorithm fulfils the supply and demand of producers and consumers respectively by taking into account the barrel shifter rule (if it is possible) until c_1 is completed i.e., the algorithm do not contain any producer with supply of 1 and any consumer with demand of 1 in the transportation matrix as shown in Figure 3. 7.a. At this point, the algorithm searches for unassigned consumer which is connected with at least one deleted producer in order to starts assigning banks to the remaining routes respecting the targeted steering rule. The algorithm selects consumer t_4 which is connected with deleted producer d_7. The other producer connected with d_7 is t_6. So the algorithm searches the route which has bank assignment b_0 in t_6. The route l_{06} has bank assignment of b_0 and processor assignment of P_2 whereas the processor assignment of l_{76} (deleted route) is P_1 (one processor before the P_2). The algorithm finds that deleted route l_{74} has processor assignment P_2. Respecting the barrel shifter rule, the route $l_{11,4}$ which has processor assignment P_3 (one processor after P_2) should assign the bank b_0. After the assignment of first bank, the algorithms continues to find cycle c_2 using the same approach used in construction of c_1 until the cycle c_2 is completed as shown with gray highlighted cells in Figure 3. 7.b.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	
0	$P_1(b_0)$					$P_2(b_0)$			X
1		~~P_1~~			~~P_2~~				~~2~~
2			$P_1(b_0)$					$P_1(b_0)$	X
3				P_1	P_1				2
4	$P_2(b_1)$						$P_1(b_1)$		X
5		$P_2(b_0)$					$P_3(b_0)$		X
6			$P_2(b_1)$			$P_3(b_1)$			X
7				~~P_2~~		~~P_1~~			~~2~~
8	~~P_3~~							~~P_3~~	~~2~~
9		$P_3(b_1)$						$P_2(b_1)$	X
10			~~P_3~~				~~P_2~~		~~2~~
11				P_3	P_3				2
	X	X	X	2	2	X	X	X	

(a) Cycle c_1

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	
0	$P_1(b_0)$					$P_2(b_0)$			X
1		~~P_1~~			~~P_2~~				~~2~~
2			$P_1(b_0)$					$P_1(b_0)$	X
3				$P_1(b_1)$	$P_1(b_1)$				X
4	$P_2(b_1)$						$P_1(b_1)$		X
5		$P_2(b_0)$					$P_3(b_0)$		X
6			$P_2(b_1)$			$P_3(b_1)$			X
7				~~P_2~~		~~P_1~~			~~2~~
8	~~P_3~~							~~P_3~~	~~2~~
9		$P_3(b_1)$						$P_2(b_1)$	X
10			~~P_3~~				~~P_2~~		~~2~~
11				$P_3(b_0)$	$P_3(b_0)$				X
	X	X	X	X	X	X	X	X	

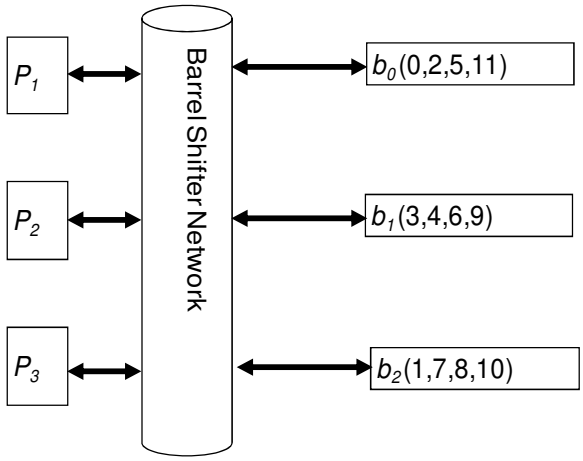
(b) Cycle c_2 in gray cells

Figure 3. 7: Assignment of memory banks in Transportation matrix

After the traversal, the algorithm finds that all the consumers fulfil this needs and the current selected producers and consumers construct a semi-2 factor. Since at the start, the algorithm determines that only one semi-2 factor can be constructed, the algorithm assigns all the deleted producers with bank assignment b_2 to complete the assignment as shown in Figure 3. 8.a. The resultant mapping which respects the barrel-shifter network is shown in Figure 3. 8.b.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	
0	$P_1(b_0)$					$P_2(b_0)$			X
1		$P_1(b_3)$			$P_2(b_3)$				X
2			$P_1(b_0)$					$P_1(b_0)$	X
3				$P_1(b_1)$	$P_1(b_1)$				X
4	$P_2(b_1)$						$P_1(b_1)$		X
5		$P_2(b_0)$					$P_3(b_0)$		X
6			$P_2(b_1)$			$P_3(b_1)$			X
7				$P_2(b_3)$		$P_1(b_3)$			X
8	$P_3(b_3)$							$P_3(b_3)$	X
9		$P_3(b_1)$						$P_2(b_1)$	X
10			$P_3(b_3)$				$P_2(b_3)$		X
11				$P_3(b_0)$	$P_3(b_0)$				X
	X	X	X	X	X	X	X	X	

(a) Complete Mapping



(b) Mapping respecting BS network

Figure 3. 8: Resultant Memory Mapping

3. Methodologies Based on Bipartite Graph to find conflict Free Memory Mapping for LDPC

In this section, we present an algorithm to solve memory mapping problem for LDPC. Modeling of problem as bipartite graph is similar to the previous section but in this case each edge represents current read and previous write access in order to accommodate "*Double Memory Mapping*" whereas in previous approach each edge represent current read and write access where mapping problem can be solved with "*Single Memory Mapping*" approach. This work has been presented in 17th IEEE International conference on Electronics, Circuits and Systems, 2010 [SAN10].

The section starts by formulating memory mapping problem for LDPC codes. The algorithm used in this section is divided into two parts. In the first part, approach models our data access matrix as bipartite graph and then, in the second part, we apply our mapping algorithms to color the edges of this bipartite graph. To facilitate the coloring of the edges of graph, mapping algorithm is further divided into two portions: Bipartite graph is first partitioned into different subgraphs and then each subgraph is colored individually to complete the mapping of data access matrix.

3.1. Problem Formulation for Memory Mapping Problem of LDPC codes

Memory mapping problem for LDPC differs from turbo due to its pattern of access of data elements in following way:

In turbo codes, each data element is accessed only two times i.e., first in natural order and the second in interleaved order by processing elements whereas, in LDPC, it can be accessed more than two times depending on the degree of variable node. Also, it is possible that, in LDPC, some data elements can be accessed by more than one time whereas some others still remains unprocessed.

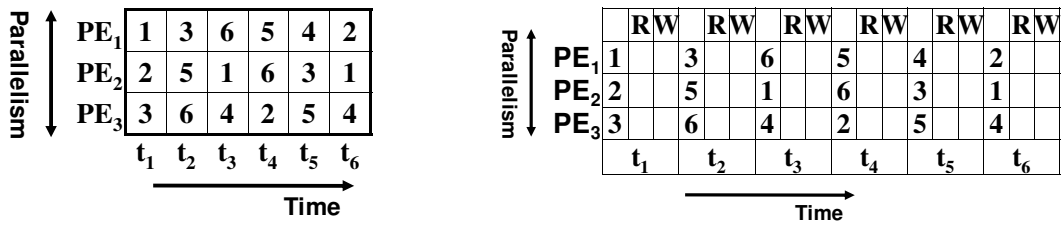
Taking into account the above pattern of access which increases the complexity of mapping problem, in this thesis, a new concept of "*Double Memory mapping*" is introduced in chapter 2. This slightly modifies the mapping problem for LDPC and can be explained as follows:

Consider a set of D data elements $\{d_1, d_2,\ldots, d_D\}$ and a set of P processing elements $\{PE_1, PE_2,\ldots, PE_P\}$ which process these D data elements in T time instances $\{t_1, t_2,\ldots, t_T\}$. In order to store these D data elements and to achieve parallel iterative processing of data for high throughput, a set of B memory banks $\{b_1, b_2,\ldots, b_B\}$, where $B = P$, is used.

Mapping problem

Store D data in B memory banks in such a manner that P processing elements can, at each time instance, access B memory banks in parallel for first reading P data and then writing back these P data without any conflict.

Mapping problem can easily be explained using a matrix called data access matrix as shown in Figure 3. 9.a. This matrix has P rows, related to the processing elements, and N columns, related to the time instances t_i. To accommodate the "Double Memory Mapping" concept, each column is further divided into three sub-columns. First sub-column shows the data which need to be accessed in parallel by P processing elements at t_i whereas second sub-column contains the memory banks from where data are read and third sub-column represents the memory banks in which these data are written at t_i. Also, data in each row are processed by the processing element connected with this row. Figure 3. 9.b. represents the mapping matrix in which we have $D = 6$, $P = B = 3$, $R = 2$ and $T = 6$. Each data is processed by 3 times which shows the iterative nature of the data access. However, data accesses are interleaved in time and there is no regularity in processing the data; e.g., data 3 is successively processed in time instances t_1 and t_2 whereas the first access to the data 4 occurs at time instance t_3.



(a) Data Access Matrix (b) Mapping Matrix

Figure 3. 9: Data Access Matrices for LDPC Codes

Memory Mapping Constraints

To successfully map the data (i.e. to allow conflict free parallel memory access) (1) in a given number of memory banks and (2) to tackle the iterative nature of data access in error correction coding, the mapping matrix must fulfill the following two constraints:

1- At each time instance, all the memory banks have to be used one and only one time.

2- The bank of the last write access to a data must be the same as the bank of its first read access.

Architectural objectives for LDPC memory mapping problem is not discussed in this thesis but could be included in current algorithms for future works.

3.2. Modeling

In first part, a bipartite graph $G = (T \cup D, E)$ based on data access matrix of Figure 3. 9.a. is prepared and is shown in Figure 3. 10. In G, vertex set T represents all the time instances and vertex set D represents all the data elements used in the computation. An edge (t, d) is incident to the data element vertex d and to the time instance vertex t if d needs to be processed at t (i.e. data d will be first read and next written at time t). Moreover, different data accesses are represented based on the relative position i of edges at the data vertex i.e. first edge at d represents the first read and write access and so on (see Figure 3. 11.a).

Property ***Placement Property***

For finding memory mapping which is valid for both read and write accesses, the following placement property is considered while searching the edges during execution of partitioning and coloring algorithms. (see Figure 3. 11.b.)

$$i_{th}\ write\ access \quad = \quad modulo_{degree\ d}(i + 1)\ read\ access$$

Definition ***Direct Edge and Induced Edge***

An edge that represents the j_{th} read access will be next referred in this algorithm as a *direct edge* and the edge corresponding to the associated write access as the *induced edge*.

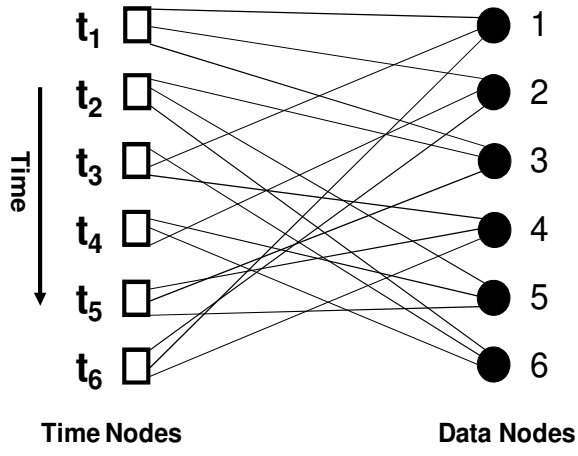
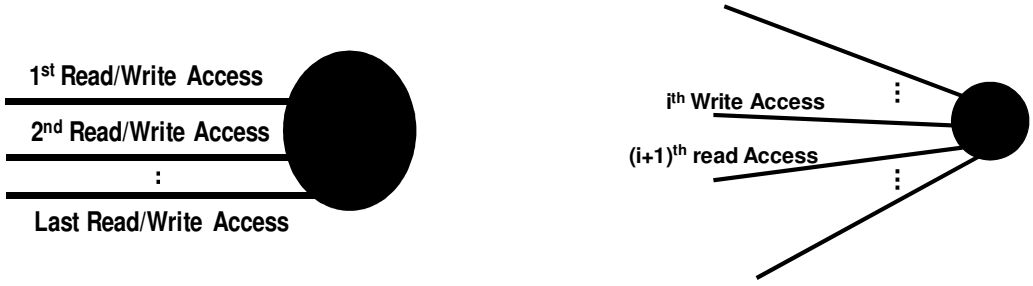


Figure 3. 10: Bipartite Representation of Data Access Matrix of Figure 3. 9.a



(a) Representation of edges on data *d* (b) Placement Property

Figure 3. 11: Representation of edges on data node *d*

Property ***Mapping Graph***

One interesting property of LDPC decoding is that the number of accesses to data at any time instance is always the same which implies that corresponding bipartite graph is always semi regular. This implies that all the time nodes in the bipartite graph have the same degree f_t. The graph with this property is called *mapping graph* in our algorithm.

Based on this property of mapping graph, the following definitions and lemma resulted:

Definition ***Partition***

Partition in mapping graph is defined as a sub graph containing all time vertices.

Lemma ***3.1***

When the degree f_t of the time vertex in a *mapping graph* is even then we have $f_t/2$ partitions in which each time vertex's degree f_t' is 2.

Lemma ***3.2***

When the degree f_t of the time vertex in a *mapping graph* is odd then we have $\lfloor f_t/2 \rfloor$ partitions in which each time vertex's degree f_t' is 2 and one partition in which f_t' is 1.

Definition ***Proper Partition***

Partition which respects either Lemma 1 or Lemma 2 in mapping graph is called *proper partition*.

3.3. Algorithm

After modeling the mapping problem as bipartite graph, the next step is to color the edges of the graph respecting the "*memory mapping constraints*" (presented in section 3.1). To make coloring easy, the algorithm first divides the graph into different *proper partitions* and then color the edges of each *proper partition* independently depending on its degree.

3.3.1. Partition the Bipartite Graph

In this step, bipartite graph *G* is divided into *proper partitions*. In order to simplify the coloring algorithm presented next, one constraint named *partitioning constraint* must be respected during constructions of *proper partitions*.

Constraint ***Partitioning Constraint***

No more than 2 read or write accesses have to be done at each time instance in a proper partition. Following this constraint always allows constructing proper partitions.

In this algorithm, two processes called *Process of traversal* and *Process of elimination* are worked side by side and applied at each time and data vertex in order to construct proper partitions.

Definition ***Process of traversal***

This process randomly selects one edge available at current data or time node and records its induced edge.

Definition ***Process of elimination***

This process removes all the edges (either direct or indirect) from the current partition which contradict the partitioning constraint.

Hence if f_t' selected direct edges (i.e. read accesses) appear in a time node then the remaining (i.e. non-selected) available edges at that time instance are eliminated. Also, if f_t' recorded induced edges (i.e. write accesses) appear in a time node then the direct edges associated to the remaining (i.e. non-recorded) induced edges of that time node are eliminated.
Each proper partition is constructed using the partitioning algorithm in Figure 3. 12.

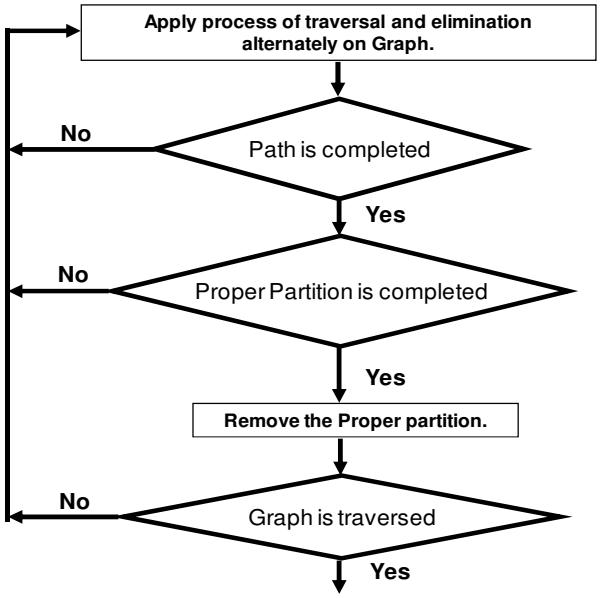


Figure 3. 12: Partitioning Algorithm

The algorithm starts constructing a path p_{cur} by choosing any data vertex d_{cur} and then by applying process of traversal which selects randomly an edge (d_{cur}, t_{cur}) to reach at the time vertex t_{cur}. Process of elimination is then applied to remove all the edges which contradict the partitioning constraint. At t_{cur}, the process of traversal is again applied to choose another edge (t_{cur}, d_{next}) to reach at the data vertex d_{next}. Again the process of elimination is applied to remove all the edges which contradict the partitioning constraint. At that time $p_{cur} = \{(d_{cur}, t_{cur}), (t_{cur}, d_{next})\}$. The algorithm continues until p_{cur} is completed, i.e. the process of traversal does not find any valid edge to be included in p_{cur}. The path is added in the current subgraph sg_{cur}. The algorithm tests if the sg_{cur} is a partition (i.e. all the time node has been traversed). Once a partition has been extracted the algorithm stops. Otherwise, the algorithm starts constructing another path p_{next} by using the remaining edges of G (that have not been removed by the process of elimination). Once sg_{cur} becomes a partition, the algorithm starts constructing another partition on the remaining graph $G = G\text{-}sg_{cur}$. Partitioning is explained through a pedagogical example at the end of this section.

Algorithm needs to traverse in the worst case $2*f_t$ edges at each time instance to find two accesses (read and write) that can be included in sg_{cur}. To construct a partition sg_{cur}, algorithm needs to traverse all the time nodes, i.e. in the worst case: $2*f_t*|T|$ edges. The complexity of constructing one partition, in terms of total number of edges to be traversed, is $O(2*f_t*|T|)$. In order to construct all the partitions, overall complexity of the partitioning algorithm is $O(f_t/2 * (2*f_t*|T|)) = O(f_t^2*|T|)$.

3.3.2. Coloring edges of Bipartite Graph

Thanks to the construction of proper partitions respecting the partitioning constraint, our coloring algorithm whose flow chart is shown in Figure 3. 13, colors each partition with at most two colors. For this it uses a strategy to color each edge in each partition independently so that there is no conflict in the read and write access at each time node.

For each uncolored partition sg_{cur}, the algorithm starts by removing the read conflict accesses by assigning different color to each edge (d_i, t_{cur}) of t_{cur}. After that, following the *placement property* (see section 3.2) the algorithm searches in G for each edge (d_i, t_{cur}) of t_{cur} for the induced edge (t_{pred}, d_i). Since only two write accesses are possible at each time node (by respecting partitioning constraint), the algorithm searches in G for the direct edge (d_m, t_k) of the induced edge (t_{pred}, d_m) that belongs to sg_{cur}. The algorithm then colors (d_m, t_k) differently from (d_i, t_{cur}) and continues until it reaches the starting node whose both direct edges are already colored. While the partition is not completely colored the algorithm selects another time node t_{cur} and repeats. It should be noticed that simply giving different colors to both the direct edges at each time node in each partition without taking into account the write access memory conflicts makes the algorithm recursive.

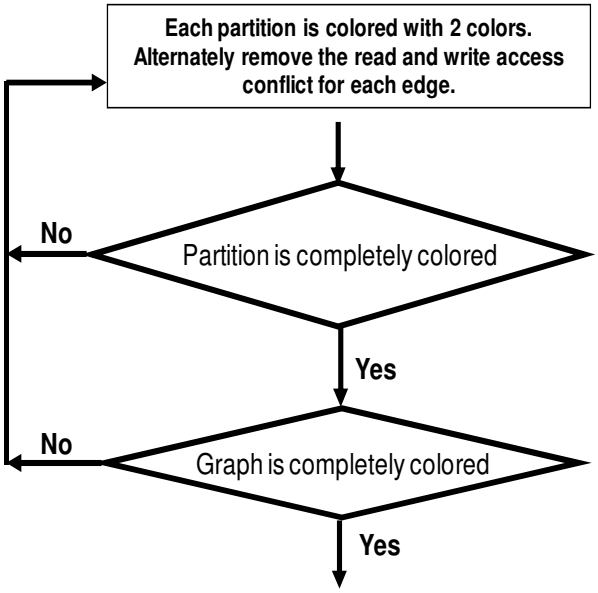


Figure 3. 13: Coloring Algorithm

Algorithm needs to traverse $2*|T|$ edges (read and write) to color both read and write accesses of each time nodes in partition sg_{cur}. So in order to color all the edges of the graph, algorithm has to traverse all the edges two times, so the complexity of coloring algorithm is O ($2*f_t*|T|$).

After calculating complexities of both partitioning and coloring algorithm, the resulting complexity to find memory mapping using this approach is O ($f_t^2*|T| + 2*f_t*|T|$).

3.4. Pedagogical Example to explain algorithm

In this section we present an example based on the data access matrix of Figure 3. 9.a. The first step is the construction of bipartite graph which is already depicted in Figure 3. 10. The degree of every time vertex of this semi regular graph is $f_t = 3$. Following Lemma 2, after applying partitioning algorithm, we will have one partition in which each time vertex's degree f_t' is 2 and one partition in which f_t' is 1.

The algorithm starts constructing the path p_1 by using the first available edge of data *1* which is $(1, t_1)$, leading to $p_1 = \{(1, t_1)\}$. The selected edge $(1, t_1)$ and its corresponding recorded induced edge $(1, t_6)$ appears respectively as bold and dotted line in Figure 3. 14.a. Using *placement property*, the write access of the edge $(1, t_1)$ indeed appears on the edge $(1, t_6)$. The process of elimination is applied and no edge is removed since number of direct edges at t_1 and induced edges at t_6 are less than f_t' = 2. The process of traversal continues and adds the edge $(t_1, 3)$ into path $p_1 = \{(1, t_1), (t_1, 3)\}$ as shown in Figure 3. 14.b. According to the partitioning constraint only two read accesses are possible at each time node. Since two read accesses are completed at t_1 therefore process of elimination deletes all the remaining edges at t_1: $(t_1, 2)$ in that case. Deleted edges are simply removed from the graph in Figure 3. 14.c. Edge $(3, t_5)$ is then selected and added in the path. Since this edge is both a recorded induced edge and a direct selected edge, it thus appears in bold and dotted line in this figure.

The process continues until we traverse the path $p_1 = \{(1, t_1), (t_1, 3), (3, t_5), (t_5, 5), (5, t_4)\}$ and reach at the time node t_4. At this point, recorded induced edges at t_2 increase to two and the process of elimination deletes all the direct edges associated to the remaining (i.e. non-recorded) induced edges at t_2. All this process is shown in Figure 3. 14.d.

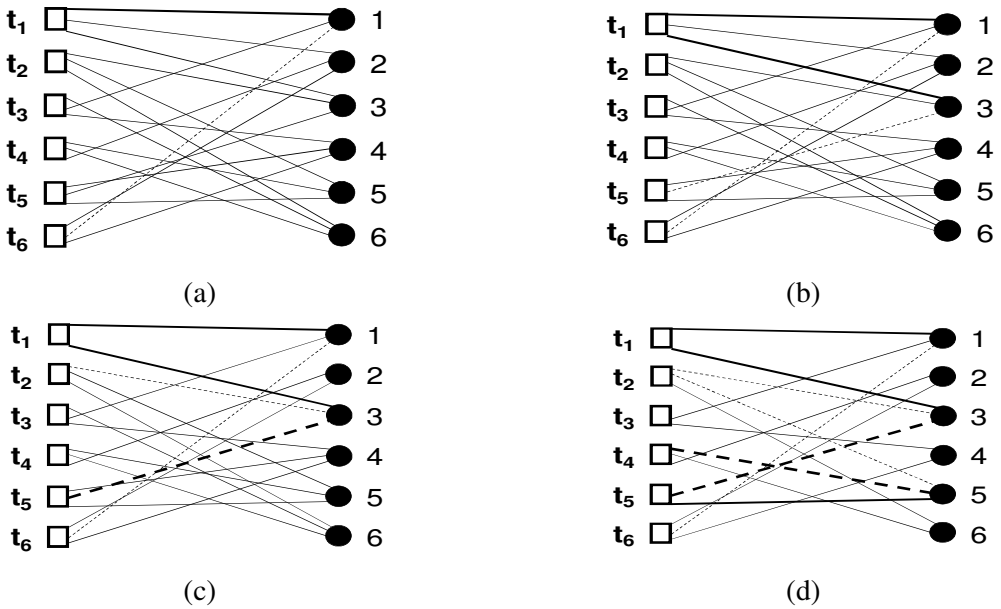


Figure 3. 14: Path construction through Partitioning Algorithm

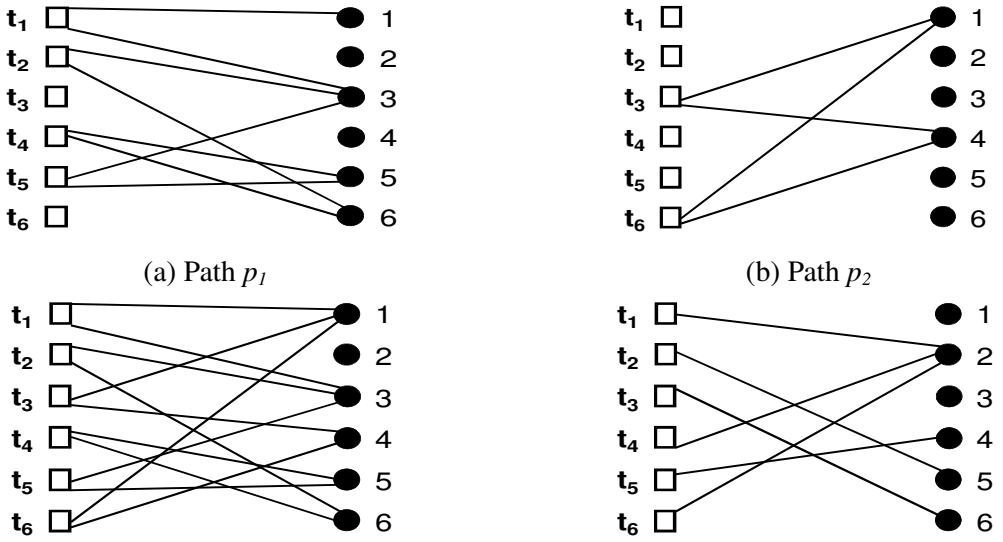
The traversal continues until the path extends to $p_1 = \{(1, t_1), (t_1, 3), (3, t_5), (t_5, 5), (5, t_4), (t_4, 6), (6, t_2), (t_2, 3)\}$ as shown in Figure 3. 15.a. No more edge can be added in the current path. We thus obtain a subgraph $sg_1 = p_1$. However, the current subgraph sg_1 is not a partition because the time nodes t_3 and t_6 are not included in p_1. Using the process of traversal, the path p_2 is obtained: $p_2 = \{(1, t_3), (t_3, 4), (4, t_6), (t_6, 1)\}$ (see Figure 3. 15.b). The partition sg_1 is the union of all the traversed paths, $sg_1 = p_1 + p_2$ (see Figure 3. 15.c)

Unfortunately, the graph is not completely traversed so the algorithm removes sg_1 to obtain the graph $G' = G - sg_1$ and applies again the processes on the remaining graph to obtain the following paths,

$p'_1 = \{(2 , t_1)\}$, $p'_2 = \{(2 , t_4)\}$, $p'_3 = \{(2 , t_6)\}$, $p'_4 = \{(4 , t_5)\}$, $p'_5 = \{(5 , t_2)\}$, $p'_6 = \{(6 , t_3)\}$.

Similarly partition sg_2 is the sum of all the traversed paths as given below, $sg_2 = p'_1 + p'_2 + p'_3 + p'_4 + p'_5 + p'_6$ (see Figure 3. 15.d).

After the construction of sg_2, the algorithm finds that the graph is completely traversed and stops.



(a) Path p_1 (b) Path p_2

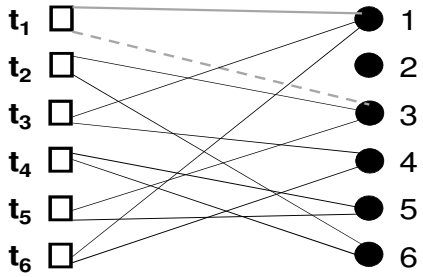
(c) partition sg_1 (d) partition sg_2

Figure 3. 15: Path construction through Partitioning Algorithm

After the generation of the partitions, each partition is colored depending on the degree f_t' of its time node. For example, the sg_1 is colored with, f_t' = 2, colors and the sg_2 is colored with, f_t' = 1, color. To better understand and explain the coloring algorithm, entries in the mapping matrix of Figure 3. 9.b is also filled along with coloring the edges of the graph.

To color the partition sg_1, we apply already presented coloring algorithm. We start by coloring the edges connected with t_1 with different colors b_0 and b_1 to avoid a conflict access. Edge (t_1, *1*) = b_0 and edge (t_1, *3*) = b_1 as shown in Figure 3. 16.a. In this figure, bold grey straight line represents color b_0 and bold grey dotted line represents color b_1. The corresponding mapping matrix is also shown in Figure 3. 16.b.

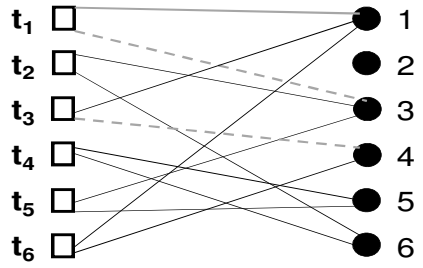
After that we search in *G* for the induced edges of these previously colored edges. Induced edge of (t_1, *1*) is (*1*, t_6) so we search for the other direct edges that belong to sg_1 and which have an induced edge at t_6 in *G*. Edge (t_3, *4*) must be colored with different color of (t_1, *1*) in order to remove the write access conflict at t_6. So we color (t_3, *4*) = b_1 (see Figure 3. 16.c). The write access of (t_1, *2*) also occurs at t_6. However (t_1, *2*) does not belong to sg_1, it is not colored at that time. The corresponding mapping matrix is shown in Figure 3. 16.d.



(a) First step of sg_1 coloring

	R	W		R	W		R	W		R	W		R	W		R	W
1	b_0		3			6			5			4			2		
2			5			1			6			3		b_1	1		b_0
3	b_1		6			4			2			5			4		
	t_1			t_2			t_3			t_4			t_5			t_6	

(b) Mapping matrix: first step of sg_1 coloring



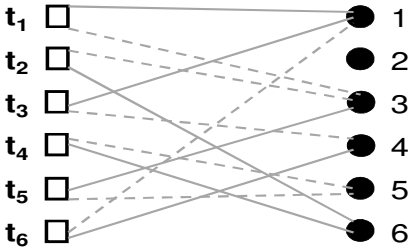
(c) Second step of sg_1 coloring

	R	W		R	W		R	W		R	W		R	W		R	W
1	b_0		3			6			5			4			2		
2			5			1			6			3		b_1	1		b_0
3	b_1		6			4	b_1		2			5			4		b_1
	t_1			t_2			t_3			t_4			t_5			t_6	

(d) Mapping matrix: second step of sg_1 coloring

Figure 3. 16: Conflict free edge coloring of sg_1

This process continues until the partition is completely colored . The complete coloring of sg1 is shown in Figure 3. 17.a. The corresponding mapping matrix is presented in Figure 3. 17.b.



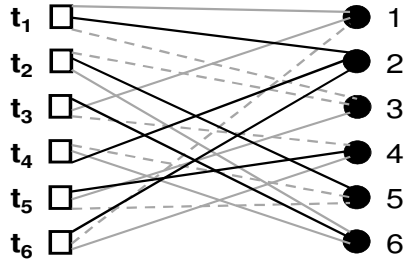
(a) Coloring of sg_1

	R	W		R	W		R	W		R	W		R	W		R	W
1	b_0	b_0	3	b_1	b_0	6		b_0	5	b_1	b_1	4		b_0	2		
2			5		b_1	1	b_0	b_1	6	b_0	b_0	3	b_0	b_1	1	b_1	b_0
3	b_1	b_1	6	b_0		4	b_1		2			5	b_1		4	b_0	b_1
	t_1			t_2			t_3			t_4			t_5			t_6	

(b) Complete coloring of sg_1

Figure 3. 17: Conflict free edge coloring of sg_1

The coloring of sg_2 is easier: all edges are colored with one single color b_2 since the degree of every time vertex in this partition is $f_t' = 1$. Complete coloring of G is shown in Figure 3. 18.a. The corresponding mapping matrix is presented in Figure 3. 18.b.



(a) Data Access Matrix

	R	W		R	W		R	W		R	W		R	W		R	W
1	b_0	b_0	3	b_1	b_0	6	b_2	b_0	5	b_1	b_1	4	b_2	b_0	2	b_2	b_2
2	b_2	b_2	5	b_2	b_1	1	b_0	b_1	6	b_0	b_0	3	b_0	b_1	1	b_1	b_0
3	b_1	b_1	6	b_0	b_2	4	b_1	b_2	2	b_2	b_2	5	b_1	b_2	4	b_0	b_1
	t_1			t_2			t_3			t_4			t_5			t_6	

(b) Bipartite Graph of a

Figure 3. 18: Conflict free edge coloring of G and corresponding mapping

4. Conclusion

In this chapter, two approaches based on bipartite graph to tackle memory mapping problem for turbo and LDPC codes have been presented. Memory mapping problems for turbo and LDPC codes have first been formulated as data access and mapping matrices next used to construct graphs. Based on these formal models, a first algorithm, based on method to solve transportation problem, has been presented to tackle mapping problem. A second approach to solve mapping problem using Double Memory mapping technique has then been introduced.

In next chapter, two more approaches to tackle memory mapping problem are presented. The novelty in these approaches is that they are modeled as tripartite graph whereas in this chapter each approach is modeled as bipartite graph. In first section of next chapter, LDPC memory mapping problem is tackled and different terms are defined to understand the modeling and 2-step coloring algorithm. This approach uses the same mapping matrix formulated in a current chapter for LDPC codes and followed the mapping constraints developed during mapping problem formulation. The approach is explained through pedagogical example at the end of this section.

The approach that is presented in the second part of next chapter uses the same algorithm to solve both memory mapping problems. In this approach, both mapping problems are modeled as tripartite graph and then this tripartite is converted into bipartite graph. The conversion is carried out by using distinct data access properties for both turbo and LDPC processing elements. After the conversion, bipartite edge coloring algorithm that is already presented and proved in literature is applied on the graph to color the its edges. One efficient algorithm to color the edges of the graph is presented and explained through pedagogical example in this section to describe the approach. Using bipartite edge coloring algorithm validates the algorithms presented in this thesis work by proving that it is always possible to find memory mapping using Single and Double Memory Mapping techniques and the algorithm finishes in polynomial time.

Chapter 4
METHODS BASED ON TRIPARTITE GRAPH FOR SOLVING MEMORY MAPPING PROBLEMS

Table of Contents

1. Introduction **79**

2. Methodology Based on Tripartite Graph to find conflict Free Memory Mapping for LDPC **79**

2.1. Modeling ----------80
2.2. 2-Step Coloring Approach----------82
2.3. Pedagogical Example to explain algorithm ----------84

3. Constructing Bipartite Graph for Turbo and LDPC Codes **87**

3.1. Construction of Bipartite Graph for Mapping Problem of Turbo Codes----------88
3.2. Construction of Bipartite Graph for Mapping Problem of LDPC Codes ---------89
3.3. Bipartite Edge Coloring Algorithm ----------90
3.4. Pedagogical Example to explain algorithm ----------93

4. Complexity comparison of Different algorithms **96**

5. Conclusion **97**

In this chapter, two more approaches that model memory mapping problem as tripartite graph are presented. Due to "Double Memory mapping" technique introduced during this thesis work, each time instance is divided into read and write time instances and mapping matrix is modeled as tripartite graph in this chapter. In the first approach introduced in this chapter, each tripartite graph constructed from mapping matrix is partitioned into different subgraphs and then each subgraph is colored individually to find "Double Memory mapping". In the second approach, both Single and Double Memory Mapping techniques are modeled as tripartite graph and then this graph transforms into bipartite graph on which any bipartite edge coloring algorithm is applied to color the edges of the graph and to find memory mapping.

1. Introduction

The second approach presented in previous chapter models our mapping problem as bipartite graph and then finds conflict free memory mapping using edge coloring. Since each edge represents current read and previous write access, applying existing edge coloring algorithms which are already presented in literature is difficult to color the edges of our bipartite graph. This drawback motivated us to present our own algorithm based on "divide on conquer" approach to color the edges respecting the mapping constraints. However, due to double meaning of edge, it seemed pertinent to represent our mapping matrix as tripartite graph in which each time vertex is divided into two time vertices: one vertex is used to represent read access and the other to write access at that time instant. Afterwards, we correlated two edges (one that is connected with read access vertex and other with write access vertex) that follow our mapping constraints to develop a new algorithm based on tripartite edge coloring to find memory mapping. The advantage of this modeling is that we can transform this tripartite graph into bipartite graph on which any bipartite edge coloring algorithm can be applied. This technique provides a modeling on which current and future bipartite edge coloring algorithms can be applied. This approach also validates our double memory mapping approach by proving that it is always possible to provide conflict free concurrent accesses to all the processing elements for any type of memory mapping problem using double memory mapping approach in polynomial time.

The chapter starts by modeling our mapping matrix as tripartite graph that contains three sets of vertices: read access time vertex, data nodes and write access time vertex. Edges related to current read and previous write access are correlated using mapping constraints and then this tripartite graph is partitioned into different subgraphs of equal sizes. Each subgraph is then colored separately to generate memory mapping. In the second part of this chapter, the correlated edges of tripartite graph are joined and data nodes are removed to generate bipartite graph that respects mapping constraints and on which any bipartite edge coloring algorithm can be applied to find conflict free mapping. One efficient bipartite edge coloring algorithm is also presented in this section. Both algorithms are explained through pedagogical examples in this thesis.

2. Methodology Based on Tripartite Graph to find conflict Free Memory Mapping for LDPC

In the first approach, we transform our mapping matrix into tripartite graph and then successively partition our graph into several subgraphs of equal size. The concept of related edges are introduced in this approach in which current read and previous write access are associated during partitioning process. Also, different constraints are imposed during partitioning process so that algorithm can color the edges of each subgraph without any recursion. Finally, edges of each subgraph are colored separately, with the help of related edges, to find conflict free memory mapping using *Double memory mapping* approach. This work was presented in IEEE International Symposium on Circuits and Systems, 2011 [SAN11].

2.1. Modeling

A Tripartite graph $G = (T_R \cup T_W \cup D, E)$ is constructed based on mapping matrix of Figure 3.9.b. as shown in Figure 4. 1. Vertex sets T_R and T_W represent all the time instances at which data are read and written respectively. Vertex set D represents all the data used in the computation. An edge (d, t_{aR}) is incident to the data vertex d and to the read access time instance vertex t_{aR} if d needs to be read at t_{aR} . Similarly, an edge (t_{cW}, d) is incident to d and to the write access time instance vertex t_{cW} if d needs to be written at t_{cW}. Moreover, at each data vertex d, edges (d, t_{aR}) and (t_{cW}, d) are placed on two different sides of d as shown in Figure 4. 2.a.

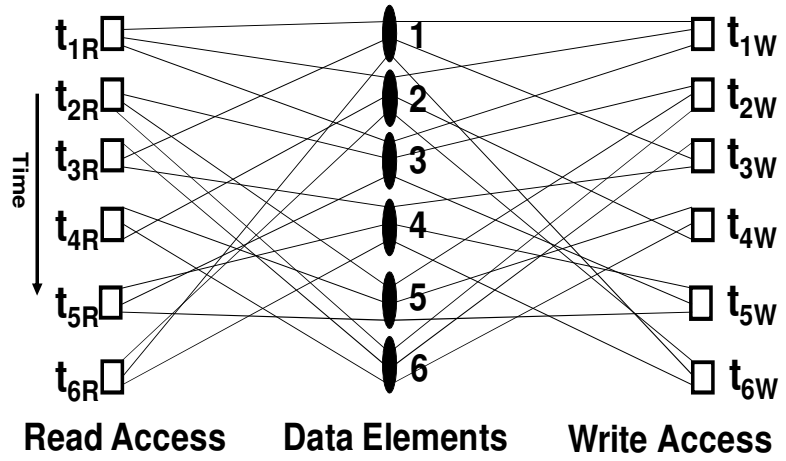


Figure 4. 1**:** Tripartite graph for mapping matrix of Figure 3.9

In order to follow the mapping constraint and for functional correctness of data accesses, the memory bank from which data is read from its current access must be the same as the memory bank in which the data has been previously written. If i is the access order of data d and n is the total number of times the data d is accessed, then $i = \{1, 2, \ldots\ldots, n\}$.

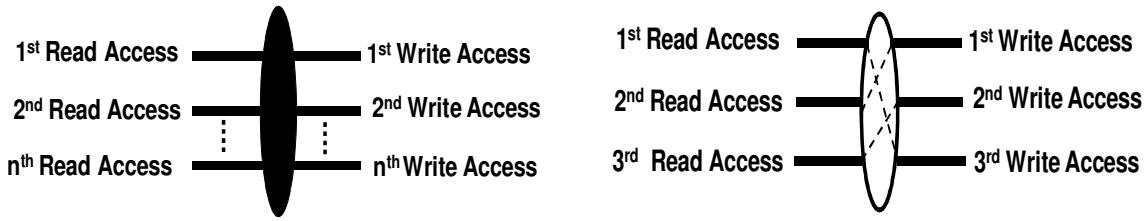
Definition ***Related Edges***

Two edges (d, t_{aR}) and (t_{cW}, d) are called *related edges* if

$$i = j - 1 \text{ for } i > 1$$
$$n \quad \text{for } i = 1$$

where $i = Order(d, t_{aR})$, $j = Order(t_{cW}, d)$ and where $Order(d, t_{aR})$ *and* $Order(t_{cw}, d)$ are respectively the read and the write access order of data d.

If colors of edges represent memory banks (as shown in *section 2.2*), then at each data vertex *d, related edges* must have the same color. *Related edges* representation of data node d for $i = 3$ is shown in Figure 4. 2.b. Related edges are connected with dotted line.



(a) Data Node *d* Representation (b) Related edges representation for $i = 3$ of *d*

Figure 4. 2: Single Node Representation on Tripartite graph

Definition ***Semi Regular Tripartite Graph***

A tripartite graph is *semi regular,* if all the vertices in any of its vertex set have the same degree.

Definition ***Partition***

If S_i is the vertex set whose all the vertices have the same degree in a semi regular tripartite graph $G = (S_1 \cup S_2 \cup S_3, E)$ then *partition* in G is defined as a subgraph containing all the elements of S_i (i.e. S_1, S_2 or S_3).

Lemma 4.1

When the degree f_t of a vertex of S_i in a semi *regular* graph is even then we have $f_t/2$ partitions in which each vertex's degree f_t' is 2.

Lemma 4.2

When the degree f_t of a vertex of S_i in a *semi regular* graph is odd then we have $\lfloor f_t/2 \rfloor$ partitions in which each vertex's degree f_t' is 2 and one subgraph in which f_t' is 1.

Definition ***Regular Partition***

A *regular partition* in semi regular tripartite graph is a partition that respects either Lemma 4.1 or Lemma 4.2.

Property

One interesting property of parallel LDPC decoding architecture is that the number of accesses to data or processing elements at any time instance is always equal which implies that corresponding tripartite graph is always semi regular at vertex set T_R and T_W. This implies that all the time nodes (either for read or write accesses) in the tripartite graph have the same degree f_t=P.

Since vertex set T_R and T_W are always semi regular, the regular partitions contain all the vertices of both T_R and T_W with the degree requirement mentioned in *Lemma 1* and *2*.

2.2. 2-Step Coloring Approach

As the name suggest, algorithm colors the edges of tripartite graph in two steps. In the first step, the algorithm divides tripartite graph into different regular partitions and then in the second step, each partition is colored independently to generate conflict free memory mapping. This 2-step approach is necessary because constraints imposed during construction of regular partition in first step avoid the recursion during coloring the edges of graph in second step.

2.2.1. Step 1: Partitioning Algorithm

The flow chart for the partitioning algorithm is presented in Figure 4. 3. Some definitions that need to be explained to understand the flow of algorithm are presented below.

Definition ***Valid and Invalid Edges***

An edge is *invalid* if its selection decreases the number of edges at any read or write access vertex to less than f_t' (where f_t' = 2) . Otherwise, it is a *valid edge*.

Definition ***Process of addition & Process of deletion***

Process of addition includes valid and its related edge in the current regular partition whereas *process of deletion* removes invalid and its related edge from the current regular partition.

The algorithm begins constructing a path p_{curr} by selecting a data vertex $d_{curr.}$ *Process of addition* is then invoked to add a valid edge (d_{curr}, t_{aR}) into p_{curr} and its *related edge* (t_{cW}, d_{curr}) into the current set of related edges $Relp_{curr}$. If number of selected edges at any read or write time instance reaches to f_t' then *Process of deletion* removes all other unselected and their related edges from that time node. The purpose of these processes is to converge into the construction of regular partition.

Process of addition is again invoked at $t_{curr,R}$ to add another *valid edge* (t_{aR}, d_{next}) into p_{curr} and to reach at d_{next}. The related edge of (t_{aR}, d_{next}) is included in $Relp_{curr}$. Process of deletion is then applied. At that point $p_{curr} = \{(d_{curr}, t_{aR}), (t_{aR}, d_{next})\}$. The algorithm continues by alternating applying process of addition and process of deletion until p_{curr} is completed, i.e. the process of addition does not find any valid edge to be included in p_{curr}. At that point, p_{curr} and $Relp_{curr}$ are included in the current partition $Par_{curr.}$

While the partition is not regular (i.e. degree of valid edges at each read and write access time nodes is exactly d_t'), the algorithm starts constructing another path by using the remaining edges of G_{tmp}. The algorithm starts constructing another partition on the remaining graph $G_{tmp} = G - Par_{cur}$ until G_{tmp} is not empty.

The proposed algorithm needs to traverse f_t edges at each time instance to find two read and writes accesses that can be included in Par_{cur}. In order to construct a partition Par_{cur}, algorithm needs to traverse in the worst case $2*f_t*|T|$ edges. The resulting exploration complexity to construct one partition is $O(2*f_t*|T|)$. In order to construct all the partitions, overall complexity of the partitioning algorithm is $O((f_t/2)*(2*f_t*|T|)) = O(f_t^2*|T|)$.

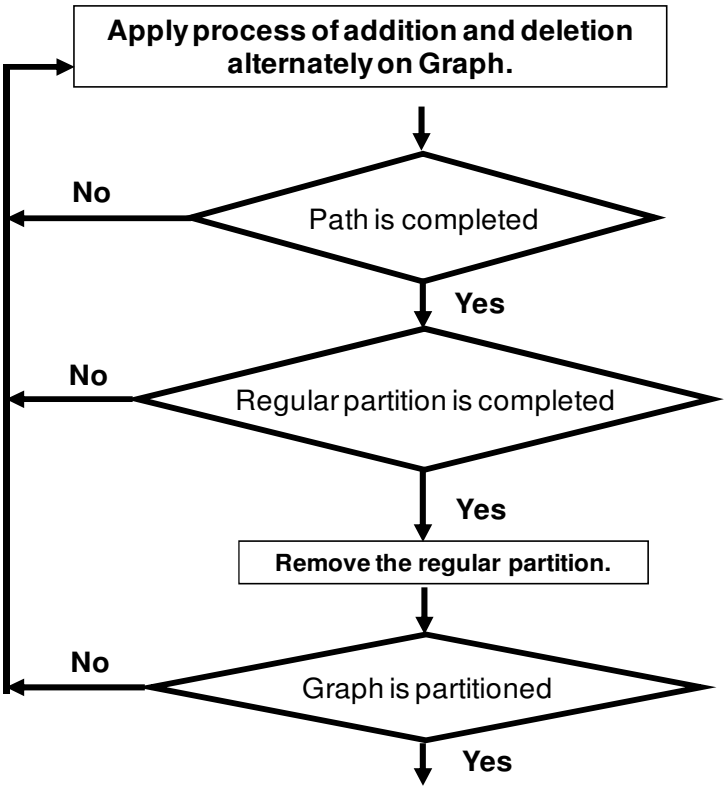


Figure 4. 3: Partitioning Algorithm

2.2.2. Step II: Coloring Algorithm

As explained in previous section, in each regular partition, degree f_t of any vertex is either 1 or 2. For f_t = 1, we do not need any algorithm because we can give one color to all the edges in the partition and its related edges. For f_t = 2, algorithm gives two color to the edges of regular partition using the algorithm whose flow chart is shown in Figure 4. 4. As described previously, the algorithm is recursion free due to constraints imposed during construction of regular partition in step I.

For each partition Par_{curr} of f_t = 2, the algorithm starts by choosing any read access time vertex $t_{init,R}$ whose edges are still not colored. First color is given to edge $(t_{init,R}, d)$ and its related edge $(d, t_{curr,W})$. Since at most two edges exist at each time vertex thanks to the construction of regular partitions, the algorithm gives a second color to $(t_{curr,W}, d_{next})$ and its related edge $(d_{next}, t_{curr,R})$. After reaching at $t_{curr,R}$, the algorithm tests whether $t_{curr,R} = t_{init,R}$. If not, then algorithm gives first color to the 2nd edge at $t_{curr,R}$ and its related edge until algorithm reaches at $t_{init,R}$. In case algorithm reaches $t_{init,R}$, it tests whether partition is completely colored. If not, then algorithm chooses another node as $t_{init,R}$ and repeats the same process until Par_{curr} is completely colored.

The coloring algorithm needs to traverse $2*|T|$ edges to color both read and write accesses in partition Par_{cur}. So in order to color all the edges of the graph, algorithm traverses $2*f_t*|T|$ edges in the worst case, then the coloring algorithm (in terms of number of edges to be explored) is $O(2*f_t*|T|)$.

After calculating complexities of both partitioning and coloring algorithm, the resulting algorithm complexity to find memory mapping using bipartite edge coloring approach is $O(f_t^2|T| + 2f_t|T|)$.

Partitioning and coloring algorithms are explained through a pedagogical example in the next section.

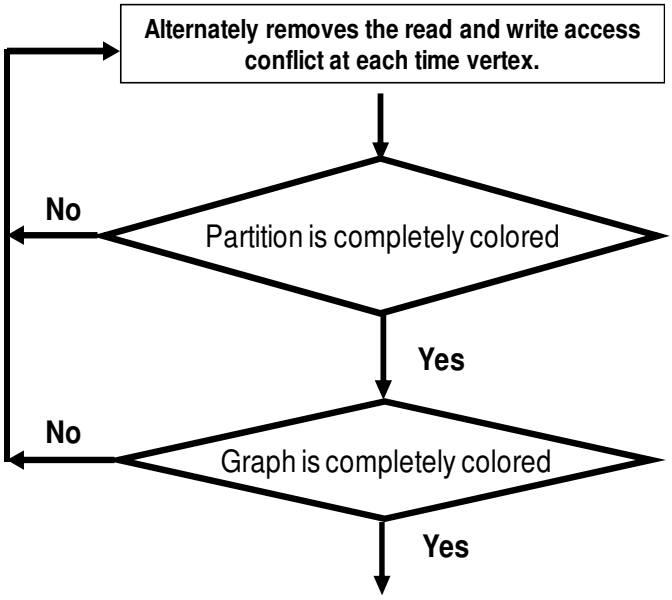


Figure 4. 4: Coloring Algorithm

2.3. Pedagogical Example to explain algorithm

Let us present an example based on the mapping matrix in Figure 3.9.b. The first step is the construction of tripartite graph which is already depicted in Figure 4. 1. This semi regular tripartite graph has each time vertex with degree $f_t = 3$. Following Lemma 2, we will have after applying the partitioning algorithm two partitions: one partition in which each time vertex's degree f_t' is 2 and one partition in which f_t' is 1.

The algorithm starts by selecting *data 1* and then invokes the process of addition which adds the first available edge $(1, t_{1R})$ into the path p_1, leading to $p_1 = \{(1, t_{1R})\}$. The related edge of $(1, t_{1R})$ that is $(1, t_{6w})$ is also included in the current set $Relp_1$. The selected read access edges and their related edges are represented by bold lines in Figure 4. 5.a. The process of deletion is invoked but no invalid edge is found to be deleted. The process of addition continues by adding the edge $(t_{1R}, 2)$ into the path p_1 and its related edge $(2, t_{6W})$ into $Relp_1$. We have now $p_1 = \{(1, t_{1R}), (t_{1R}, 2)\}$ and $Relp_1 = \{(1, t_{6w}), (2, t_{6W})\}$. At this point, the number of access at t_{1R} and t_{6W} increases to 2 and the other unselected edges at t_{1R} and t_{6W} becomes invalid edges. So the process of deletion removes the invalid edge $(4, t_{6W})$ and its related edge $(4, t_{3R})$ as shown in Figure 4. 5.b. The algorithm continues by alternately invoking the two processes until the path p_1 reaches at t_{6R}. We have at that point $p_1 = \{(1, t_{1R}), (t_{1R}, 2), (2, t_{4R}), (t_{4R}, 6), (6, t_{3R}), (t_{3R}, 1), (1, t_{6R})\}$ and $Relp_1 = \{(1, t_{6w}), (2, t_{6W}), (2, t_{1W}), (6, t_{3W}), (6, t_{2W}), (1, t_{1w}), (1, t_{3w})\}$. The edges of p_1 and $Relp_1$ are shown in Figure 4. 5.c. At this point, the process of addition can choose either $(t_{6R}, 2)$ or $(t_{6R}, 4)$ to augment p_1. But choosing $(t_{6R}, 2)$ makes $(t_{6R}, 4)$ and its related edge $(4, t_{5W})$ invalid because the number of edges at t_{5W} becomes less than 2. So the process adds $(t_{6R}, 4)$ (the only available valid edge) into p_1, $(4, t_{5W})$ *into* $Relp_1$ and declares $(t_{6R}, 2)$ and its related edge $(2, t_{4W})$ as invalid as shown in Figure 4. 5.d.

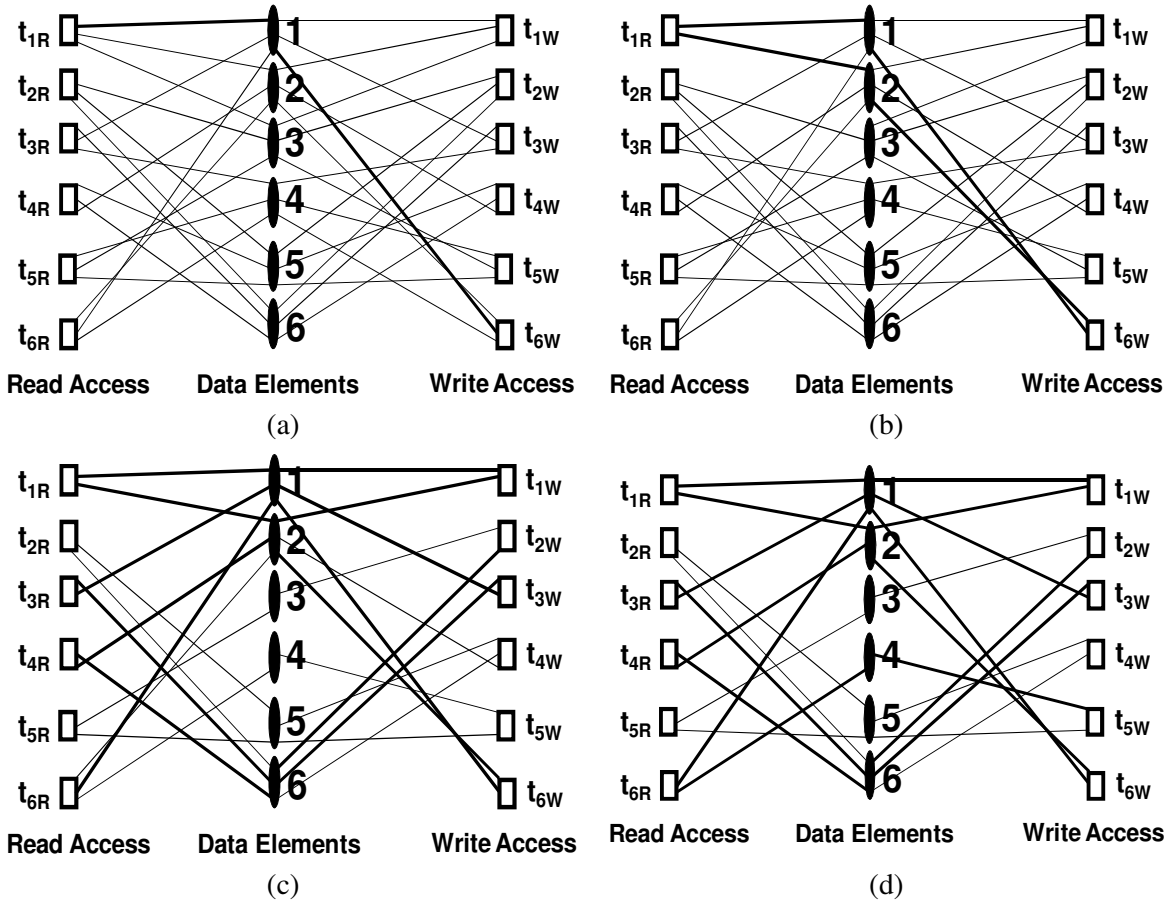


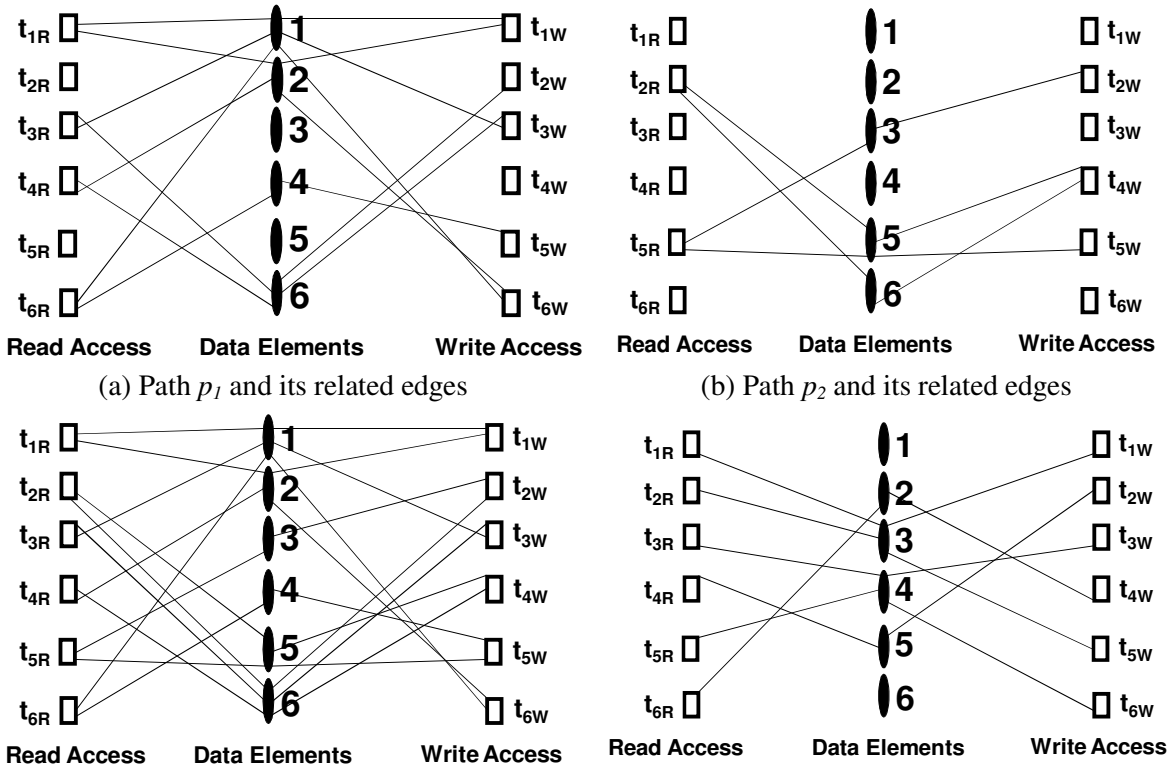
Figure 4. 5: Path construction through Partitioning Algorithm

At this stage, the algorithm finds that no more valid edges are available at data vertex 4 to be added in $p_1 = \{(1, t_{1R}), (t_{1R}, 2), (2, t_{4R}), (t_{4R}, 6),(6, t_{3R}), (t_{3R}, 1), (1, t_{6R}), (t_{6R}, 4)\}$ and $Relp_1 = \{(1, t_{6w}), (2, t_{6W}), (2, t_{1W}), (6, t_{3W}), (6, t_{2W}), (1, t_{1w}), (1, t_{3w}), (4, t_{5W})\}$ as shown in Figure 4. 6.a. The algorithm adds p_1 and $Relp_1$ into Par_1 but Par_1 does not form a regular partition because t_{2R} and t_{5R} are not included in p_1. So the algorithm starts to construct a new path p_2 by again invoking the process of addition and deletion. The resultant path $p_2 = \{(3, t_{5R}), (t_{5R}, 5), (5, t_{2R}), (t_{2R}, 6)\}$ is shown in Figure 4. 6.b. Now the partition Par_1 is the union of all the paths and their related edge sets, $Par_1 = p_1 + p_2 + Relp_1 + Relp_2$. Again the algorithm tests whether Par_1 constitutes a regular partition. This time the test is successful and the Par_1 is declared as regular partition (see Figure 4. 6.c).

After the construction of Par_1, the algorithm finds that the graph is not completely traversed. So the algorithm deletes Par_1 to obtain the graph $G_{tmp} = G - Par_1$ and applies again the processes on G_{tmp} to obtain the paths, $p'_1 = \{(2, t_{6R})\}$, $p'_2 = \{(3, t_{1R})\}$, $p'_3 = \{(3, t_{2R})\}$, $p'_4 = \{(4, t_{3R})\}$, $p'_5 = \{(4, t_{5R})\}$, $p'_6 = \{(5, t_{4R})\}$.

Similarly partition Par_2 is the sum of all the traversed paths and their related edges as given below, $Par_2 = p'_1 + p'_2 + p'_3 + p'_4 + p'_5 + p'_6 + Relp'_1 + Relp'_2 + Relp'_3 + Relp'_4 + Relp'_5 + Relp'_6$ (see Figure 4. 6.d).

After the construction of Par_2, the partitioning algorithm finds that the graph is completely traversed and is terminated.



(a) Path p_1 and its related edges

(b) Path p_2 and its related edges

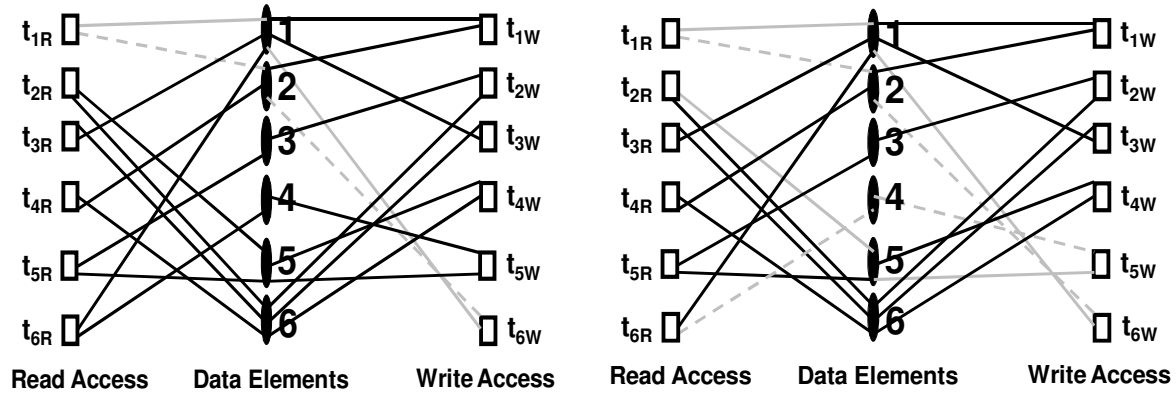
(c) Regular partition Par_1

(d) Regular partition Par_2

Figure 4. 6: Path construction through Partitioning Algorithm

After the generation of the partitions, each partition is colored depending on the degree d_t' of its time node. For example, the Par_1 is colored with, d_t' = 2, colors and the Par_2 is colored with, d_t' = 1, color. To color the partition Par_1, the algorithm starts from any read access time vertex whose edges are still not colored. In this example, the algorithm begins from t_{1R} and gives one color b_0 to $(t_{1R}, 1)$ and its related edge $(1, t_{6W})$ to reach at t_{6W}. At t_{6W}, the algorithm then gives different color b_1 to the other edge $(t_{6W}, 2)$ and its related edge $(t_{1R}, 2)$ to reach at t_{1R} as shown in Figure 4. 7.a. In this figure, grey straight line represents color b_0 and grey dotted line represents color b_1.

The algorithm finds that t_{1R} is completely colored so it chooses another uncolored read access time vertex t_{2R} and gives color b_0 to $(t_{2R}, 5)$ and its related edge $(5, t_{5W})$ to reach at t_{5W}. At t_{5W}, the algorithm gives different color b_1 to the other edge $(t_{5W}, 4)$ and its related edge $(t_{6R}, 4)$ to reach at t_{6R} as shown in Figure 4. 7.b.



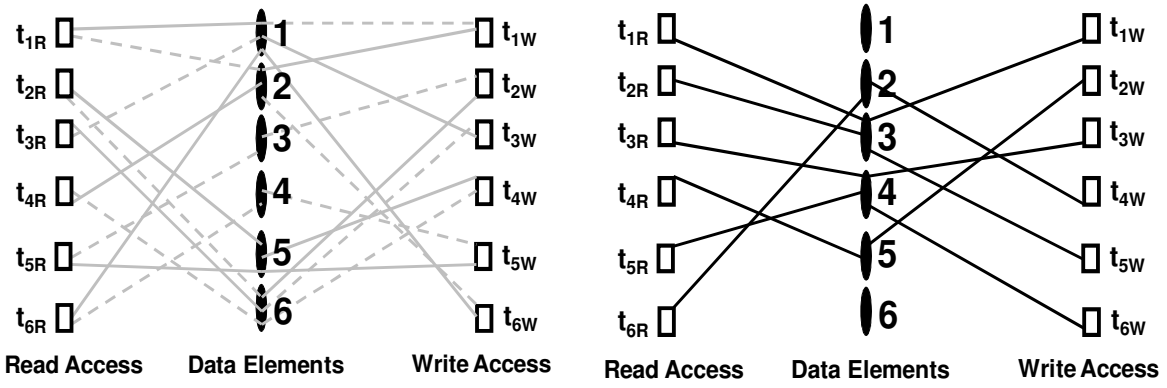
(a)

(b)

Figure 4. 7: Conflict Free Edge Coloring of Par_1

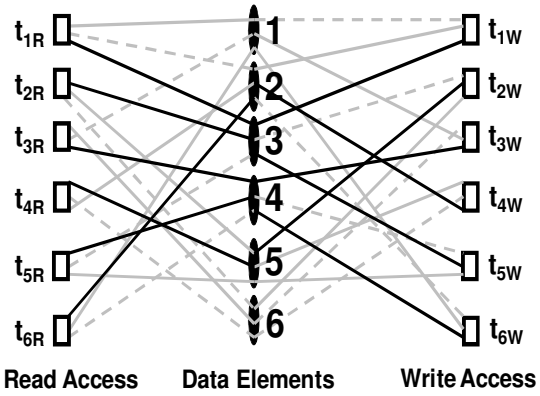
The algorithm continues until the partition is completely colored. The complete coloring of Par_1 is shown in Figure 4. 8.a. The coloring of Par_2 is easier: all the edges are colored with one single color b_2 represented by black lines in Figure 4. 8.b because d_t' = 1 as already mentioned.

The complete coloring of *G* is shown in Figure 4. 9.a. The corresponding mapping matrix is presented in Figure 4. 9.b.



(a) (b)

Figure 4. 8: Conflict Free Edge Coloring of Par_1 and Par_2



	R	W		R	W		R	W		R	W		R	W		R	W
1	b_0	b_1	3	b_2	b_1	6	b_0	b_1	5	b_2	b_0	4	b_2	b_1	2	b_2	b_1
2	b_1	b_0	5	b_0	b_2	1	b_1	b_0	6	b_1	b_1	3	b_1	b_2	1	b_0	b_0
3	b_2	b_2	6	b_1	b_0	4	b_2	b_2	2	b_0	b_2	5	b_0	b_0	4	b_1	b_2
t_1			t_2			t_3			t_4			t_5			t_6		

(a) (b)

Figure 4. 9: Conflict free edge coloring of *G* and corresponding mapping matrix

3. Constructing Bipartite Graph for Turbo and LDPC Codes

As explained in introduction, the basic idea is to transform our mapping problem for both turbo and LDPC codes into a problem for which polynomial time algorithm can be applied. The approach also validates our *Double memory mapping* method by proving that it is always possible to provide conflict free concurrent access to all the processing elements for any type of memory mapping problem using Double memory mapping approach in polynomial time.

Bipartite Edge coloring is one of such problem. So in this section, we model data access matrices for turbo codes and mapping matrices for LDPC as bipartite graphs on which any bipartite edge coloring which is already proved in literature is implemented. Since mapping problem for both Turbo and LDPC is slightly different, we model both of these problems separately in this section so that one can easily understand the transformation from matrix to graph.

3.1. Construction of Bipartite Graph for Mapping Problem of Turbo Codes

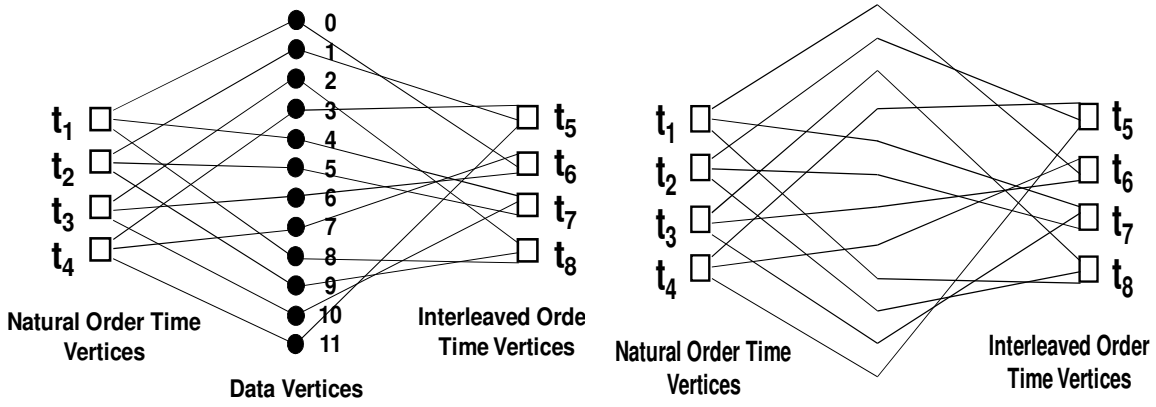
To construct bipartite graph, we first construct a tripartite graph $G' = (T_{NAT} \cup T_{INT} \cup D, E)$ and then convert this tripartite graph into bipartite graph due to specific pattern of access of turbo codes. The tripartite graph *G'*, that is constructed using natural and interleaved order data access matrices of Figure 3.1, is shown in Figure 4. 10.a. In *G'*, vertex sets T_{NAT} and T_{INT} represent all the time instances used in natural order access and interleaved order access respectively whereas vertex set *D* represents all the data elements used in the computation. An edge (t_i, d) is incident to the data vertex *d* and to the natural order time vertex t_i if *d* needs to be processed at t_i (i.e. data *d* will be read and next written at time t_i) where $t_i \in T_{NAT}$. Similarly, an edge (t_j, d) is incident to the data vertex *d* and to the interleaved order time vertex t_j if *d* needs to be processed at t_j where $t_j \in T_{INT}$.

Turbo codes have the following two distinct properties:

1- *The number of accesses to data P (i.e., number of data required to access concurrently) at any time instance is always the same (the number of accesses is equal to the number of memory banks). This property imply that in G', each time node has same degree,* $f_t = P$.

2- *Each data element is accessed only two times: one time in natural order and the other time in interleaved order. This property imply that in G', all the data nodes have the same degree,* $f_d = 2$.

Thanks to the property 2, the tripartite graph *G'* is converted into bipartite graph *G* by first joining two edges at each data vertex and then removing all the data vertices from *G*' as shown in Figure 4. 10.b. Thanks to property 1, *G* is regular with the degree of each time node, $f_t = P$.



(a) Tripartite graph for Figure 3.1. (b) Bipartite Graph for a.

Figure 4. 10: Bipartite Graph representation

One point to remember that each edge in *G* now corresponds to a data node in *G'*, so the coloring of edges in *G* actually means coloring of data nodes in *G'*. Also, according to minimum edge coloring algorithm for bipartite graph [KON16], edges of *k*-regular bipartite graph are colored with $k = P$ colors (i.e. the number of concurrent data accesses) and the colors of edges at each time node are different.

If a color corresponds to a memory bank then the above observations results in the following Lemma,

Lemma 4.3

All the data elements used in turbo codes can be stored in B memory banks where B = P so that P processors concurrently access B memory bank in parallel without any conflict in both natural and interleaved order time instance.

3.2. Construction of Bipartite Graph for Mapping Problem of LDPC Codes

The same approach is used to construct bipartite graph for LDPC codes as presented for turbo codes. First we construct tripartite graph based on mapping matrix and then convert this tripartite graph into bipartite graph. A Tripartite graph $G' = (T_R \cup T_W \cup D, E)$ is constructed based on mapping matrix of Figure 3.9.b. as shown in Figure 4. 11. Vertex sets T_R and T_W represent all the time instances at which data are read and written respectively. Vertex set D represents all the data used in the computation. An edge (d, t_{aR}) is incident to the data vertex d and to the read access time instance vertex t_{aR} if d needs to be read at t_{aR} . Similarly, an edge (t_{cW}, d) is incident to d and to the write access time instance vertex t_{cW} if d needs to be written at t_{cW}.

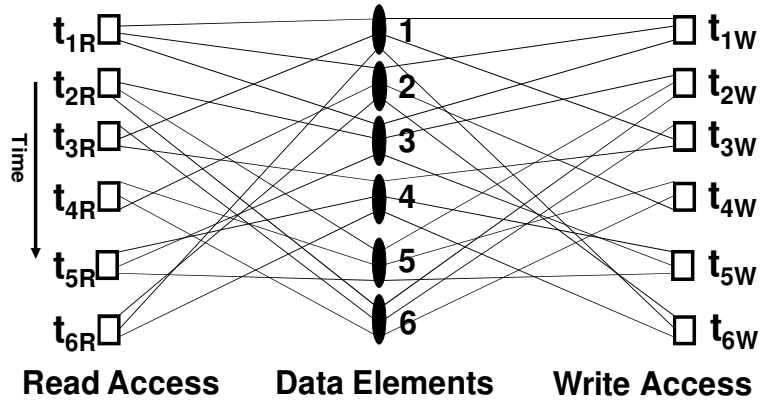


Figure 4. 11**:** Tripartite graph for mapping matrix of Figure 3.9

In order to follow the mapping constraint and for functional correctness of data accesses, the memory bank from which data is read from its current access must be the same as the memory bank in which the data has been previously written. If i is the access order of data d and n is the total number of times the data d is accessed, then $i = \{1, 2, \ldots\ldots, n\}$.

Definition ***Related Edges***

Two edges (d, t_{aR}) and (t_{cW}, d) are called *related edges* if

$$i = j - 1 \text{ for } i > 1$$
$$n \quad \text{for } i = 1$$

where $i = Order(d, t_{aR})$, $j = Order(t_{cW}, d)$ and where $Order(d, t_{aR})$ *and* $Order(t_{cw}, d)$ are respectively the read and the write access order of data d.

To convert this tripartite graph into bipartite graph *G*, we first join related edges on each data node and then remove data node to construct bipartite graph as shown in Figure 4. 12.

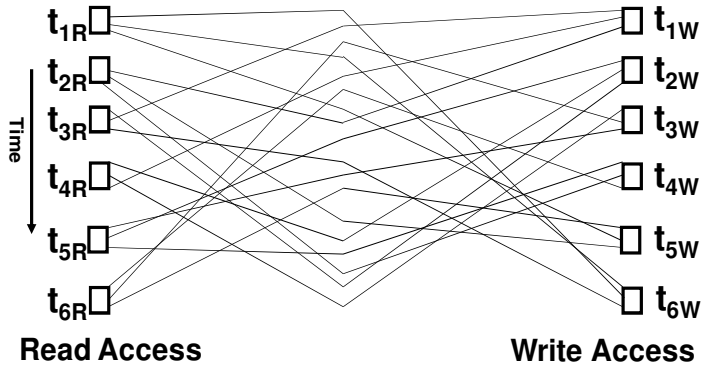


Figure 4. 12**:** Bipartite graph for mapping matrix of Figure 3.2.b

LDPC codes have following distinct property:

The number of accesses to data P at any time instance (i.e., number of data required to access concurrently) is always the same (the number of accesses is equal to the number of memory banks).
This property implies that in G:

Each time node (either read or write) has same degree $f_t = P$ which means G is always regular.

It is important to note that implementation of bipartite edge coloring fulfills the two constraints which is necessary to follow to find conflict free memory mapping. First of all, at each time instance during both read and write operation, all the memory banks have been used one and only one time. Secondly, current read and previous write accesses are connected in this bipartite graph, so these two accesses should always be taken from the same memory bank and results in fulfilling the constraint that bank of the last write access to a data must be the same as the bank of its first read access in this bipartite graph. The above observations give us the proof that "*Double Memory Mapping*" is always possible for every type of LDPC codes and results in the following Lemma,

Lemma 4.4

All the data elements used in LDPC codes can be stored in B memory banks where B = P so that P processors concurrently access B memory bank in parallel for first reading P data and then writing back these P data without any conflict.

3.3. Bipartite Edge Coloring Algorithm

After constructing bipartite graph, the next step is to apply bipartite edge coloring algorithm to color the edges of that graph in polynomial time. As explained in chapter 2, current edge coloring algorithms first convert irregular bipartite graph into regular graph. Fortunately in our case, our bipartite graph is always regular so we can directly apply edge coloring algorithm on our graphs. To find minimum edge coloring efficiently, following divide and conquer approach is used on regular bipartite graph:

Find an Euler partitioning C_{euler} (see section 3.3.1.2 of chapter 2 for explanation). Then, take every other edge along C_{euler} to obtain two ($f/2$)-regular subgraphs and reduce the problem to two ($f/2$)-regular graphs. However, to divide a regular graph into two regular subgaphs of equal degree, it is necessary that f is even. If f is odd, then algorithm first finds perfect matching M_p in G, assign one color to the edges of M_p, remove M_p from G and reduce the problem to $(f-1)$-regular graph where $(f-1)$ is even.

Complete bipartite edge coloring algorithm is shown in Table 4. 1.

Table 4. 1: Edge Coloring Algorithm

```
1   edgeColor (G, f)
2       if (f is odd) then
3           if (f = 1) then
4               M_p = G
5               assignOneColour(M_p )
6           else
7               M_p = perfectMatching(G)
8               assignOneColour(M_p )
9               edgeColour(G- M_p, f-1)
10          end if
11      else
12          C_euler = eulerPartition(G)
13          {G_1, G_2} = split(G, C_euler)
14          edgeColour(G_1, f/2)
15          edgeColour(G_2, f/2)
16      end if
```

Perfect Matching Algorithm

In this algorithm, it is important to find perfect matching M_p in G efficiently. The algorithm [SCH98] finds a perfect matching in a f_t-regular bipartite graph in O(fD) time where $f = P$ as explained in Table 4. 2.

At each point in the algorithm, the sum of the weights of the edges adjacent to a given vertex is always f_t. When the algorithm terminates, edges have either weight f or weight 0. The edges that have weight f_t form a perfect matching [SCH98].

Table 4. 2: Perfect Matching Algorithm from [SCH98]

I.	Initially, assign a weight $w(e) = 1$ to each edge e.
II.	Find a circuit C in E^* using depth first search (DFS) tree by starting at the vertex of T_{NAT} where E^* is the set of edges with $w(e) > 0$ and $w(C) := \sum_{e \in C} w(e)$.
III.	Since the number of edges in C in bipartite graph is always even (this is the intrinsic property of bipartite graph), divide C into two matching M and N by assigning alternate edges to M and N such that $w(M) > w(N)$.
IV.	Update the weights of the edges in C as follows: If edge $e \in M$, then $w(e) = w(e) + 1$

If edge $e \in N$, then $w(e) = w(e) - 1$

V. Go to II and repeat until no more circuits can be found. Note that if the graph is disconnected, algorithm will need to start a *DFS* tree at each vertex in T_{NAT} to find all circuits.

Euler Partitioning

Procedure to find Euler partitioning is taken from [GAB76] . This algorithm is modified form the original version because in our case, the graph on which this procedure is applied is always regular and each vertex has even degree. The procedure can be stated as below:

Choose any vertex of even nonzero degree. Start traversing a graph from one vertex to another by including a traversed edge into the path *p* and removing that edge from the graph until a vertex with zero degree is reached. This completes *p* in the partition. Then start constructing another path *p'* by choosing another start vertex. Repeat this process until no vertex of nonzero degree remains.

Complete Euler partitioning algorithm is shown in Table 4. 3.

Table 4. 3: Euler Partitioning Algorithm from [GAB76]

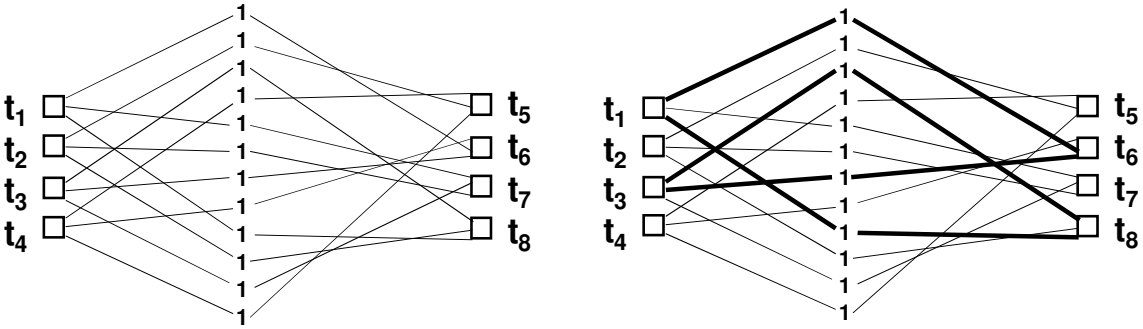
```
1   eulerPartition (G)
2       P is an empty list;
3       Q_euler is an empty queue:
4       store all vertecies of nonzero even degree in Q_euler.
5       while Q_euler is not empty loop
6           let q be the first vertex in Q_euler
7           delete q from Q_euler
8           if vertex q has nonzero degree then
9               construct a new empty path p;
10              v_init = q;
11              while vertex v_init has nonzero degree loop
12                  select an edge (v_init, v_end) in G;
13                  delete (v_init, v_end) from G;
14                  put (v_init, v_end) in p;
15                  v_end = v_init ;
16              end while
17              put path p in P
18          end if
19      end while
20      return P;
```

If the graph is bipartite and $2q$-regular where q is a positive integer, then it is possible to split the edges into 2 groups of equal sizes. After the construction of Euler circuit, the edges are oriented according to the path construction. The $2q$-regular graph can be split into two q-regular graphs by putting the edges oriented from T_{NAT} to T_{INT} in one graph and the edges oriented from T_{INT} to T_{NAT} in the other.

Bipartite edge coloring presented in [SCH98] takes O (Δm) time where Δ = degree of each node and m = number of edges in a graph. In our bipartite graph of turbo codes, degree of each time node is f_t and number of edges in a graph is $f_t * |T|$. This means that overall complexity of finding memory mapping using bipartite edge coloring for turbo codes is O ($f_t^2 * |T|$). In bipartite graph of LDPC codes, each time node is partitioned into read and write access nodes. This means that overall complexity of finding memory mapping using bipartite edge coloring for LDPC codes is O ($2*f_t^2 * |T|$).

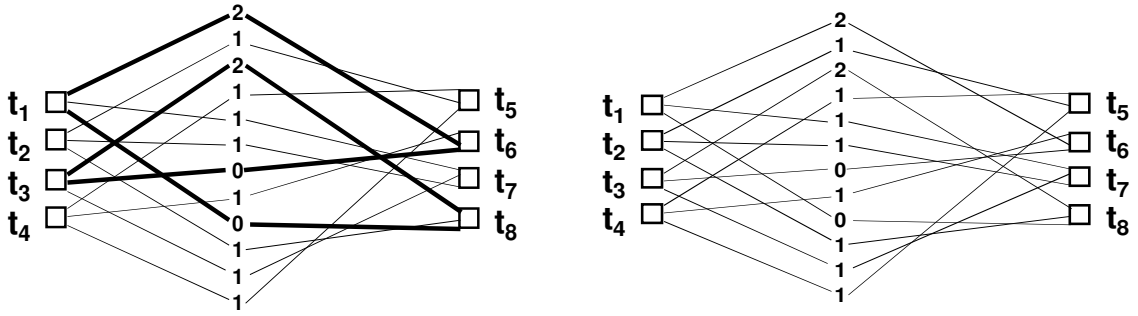
3.4. Pedagogical Example to explain algorithm

Let us present an example based on the natural and interleaved order access matrices presented in Figure 3.1. Bipartite graph for these data access matrices is depicted in Figure 4. 10.b. This bipartite graph is f-regular where $f = 3$. As explained previously, edge coloring algorithm first finds the perfect matching because f is odd. To find the perfect matching, the algorithm assigns each edge a weight 1 as shown in Figure 4. 13.a. Using DFS tree, the algorithm finds a circuit, C={(t_1, t_6), (t_6, t_3), (t_3, t_8), (t_8, t_1)} as shown in Figure 4. 13.b. Since all the edges have the weight 1, the algorithm arbitrary assigns alternate edges to the matching M and N. In this case, algorithm assigns M ={(t_1, t_6), (t_3, t_8)} and N={(t_6, t_3), (t_8, t_1)}. Also, the algorithm increases the weight of the edges in M by 1 and decreases the weight of the edges in N by 1 as shown in Figure 4. 14.a. The edges in circuit C with weight modification are represented with bold lines in this figure. After weight modification, edges with degree 0 or 3 are not used in determining the circuit in next iteration. Graph used in next iteration is shown in Figure 4. 14.b.



(a) Initialization (b) Construction of Circuit C

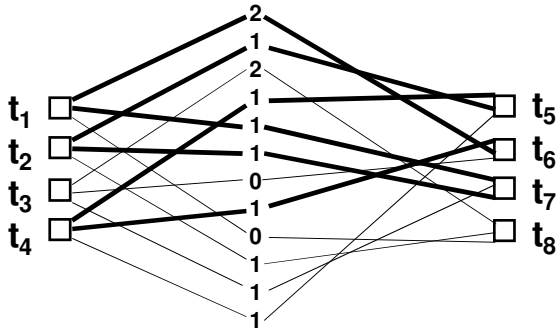
Figure 4. 13: Matching Algorithm



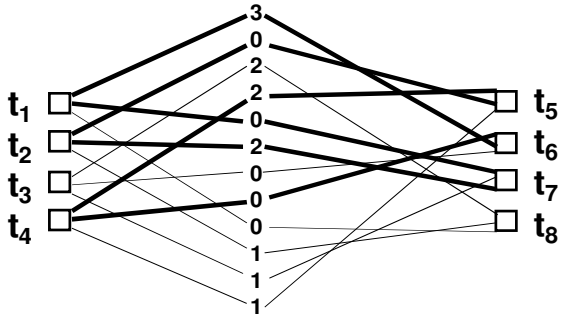
(a) Modification of weights of Edges on C (b) Graph used in next iteration

Figure 4. 14: Matching Algorithm

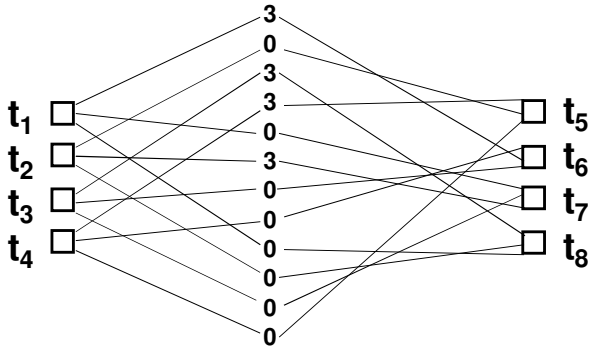
However, the algorithm is only terminated when all the edges have either weight 3 or 0. Since at the termination of previous iteration, some edges have weight 1 or 2, so the algorithm continues by constructing another circuit $C = \{(t_1, t_6), (t_6, t_4), (t_4, t_5), (t_5, t_2), (t_2, t_7), (t_7, t_1)\}$ as shown in Figure 4. 15.a. Taking alternate edges, the algorithm finds that first set of edges $s_1 = \{(t_1, t_6), (t_4, t_5), (t_2, t_7)\}$ have total weight 4 whereas the second set of edges $s_2 = \{(t_6, t_4), (t_5, t_2), (t_7, t_1)\}$ have total weight 3. So the algorithm assigns edges in s_1 to matching M and edges in s_2 to matching N. Also, the algorithm increases the weights of edges in M by 1 and decreases the weights of edges in N by 1 as shown in Figure 4. 15.b. The algorithm continues finding circuits and increases and decreases the weights of alternate edges until all the edges have either weight 3 or 0 as shown in Figure 4. 15.c. In this figure, the edges with weight 3 give us a perfect matching.



(a) Construction of Circuit C



(b) Modification of weights of Edges on C

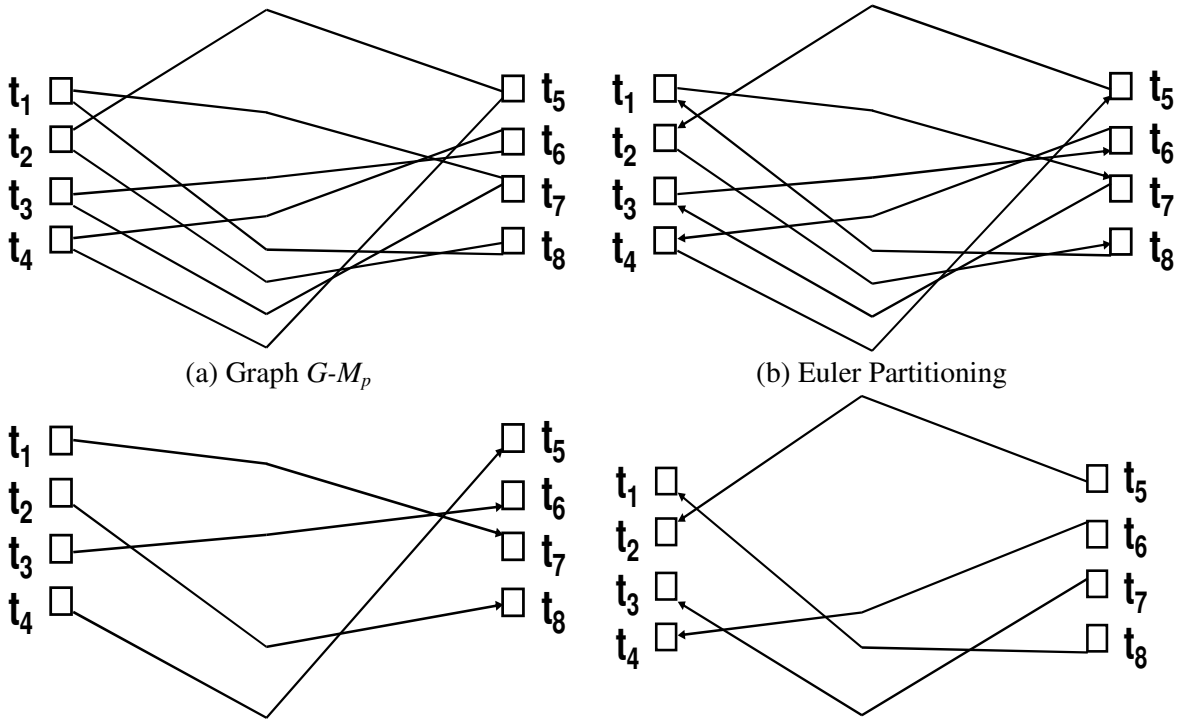


(c) Perfect Matching

Figure 4. 15: Matching Algorithm

After finding perfect matching M_p, the algorithm assigns a color to the edges of M_p, remove M_p from the graph G to construct graph $G\text{-}M_p$ as shown in Figure 4. 16.a. Now the graph transforms into 2-regular graph, so edge coloring algorithm applied eulerPartition procedure to find Euler partition $C_{euler} = \{(t_1, t_7), (t_7, t_3), (t_3, t_6), (t_6, t_4), (t_4, t_5), (t_5, t_2), (t_2, t_8), (t_8, t_1)\}$ on graph $G\text{-}M_p$. The edges in C_{euler} are also given orientation with edges from vertex in T_{NAT} to vertex in T_{INT} are given clockwise orientation whereas the edges from vertex in T_{INT} to vertex in T_{NAT} are given anti-clockwise orientation as shown in Figure 4. 16.b. After giving orientations, 2-regular graph is split into two 1-regular graphs by putting the edges oriented clockwise in $G_1 = \{(t_1, t_7), (t_3, t_6), (t_4, t_5), (t_2, t_8)\}$ and the edges oriented anticlockwise in $G_2 = \{(t_7, t_3), (t_6, t_4), (t_5, t_2), (t_8, t_1)\}$ as shown in Figure 4. 16.c. and

Figure 4. 16.d. Since both G_1 and G_2 are now 1-regular, algorithms terminates by assigning one color each to the edges of G_1 and G_2.



(a) Graph G-M_p (b) Euler Partitioning

(c) Edges with clockwise orientation(G_1) (d) Edges with anti clockwise orientation(G_2)

Figure 4. 16: Euler Partitioning

The complete edge coloring of G' after attaching data vertices in G is shown in Figure 4. 17. In this figure, the three colors of the edges which correspond to three memory banks are represented with gray bold, gray narrow and gray dotted lines. The resultant memory mapping is, *Bank A* = {*0,2,3,5*}, *Bank B* = {*1,7,8,10*} and *Bank C* = {*4,6,9,11*}.

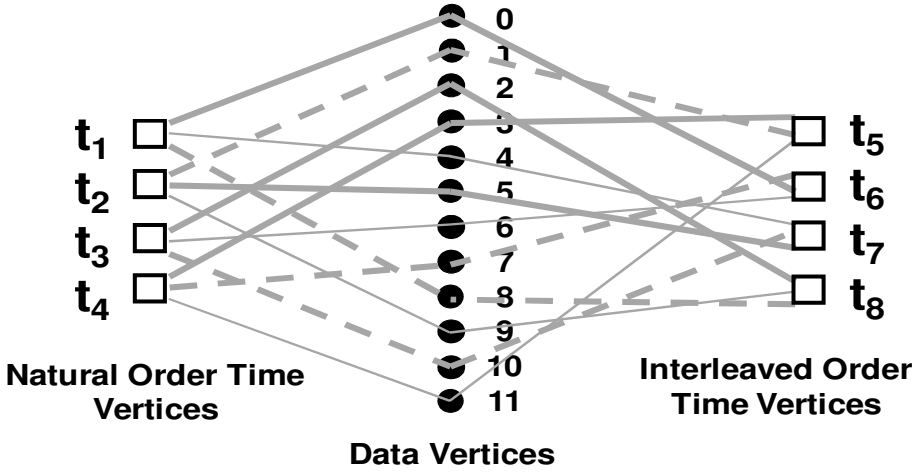


Figure 4. 17: Edge Coloring of Bipartite Graph

4. Complexity comparison of Different algorithms

In this section, the proposed memory mapping algorithms are compared in terms of internal representation model complexity and algorithm complexity. The complexities of different algorithms are compared in terms of different parameters used to develop and execute these algorithms. This comparison is shown in Table 4. 4. Internal model complexity of approaches using tripartite graph and bipartite edge coloring is low whereas algorithm implementation is easier for Transportation problem and Bipartite edge coloring approach. Runtime complexity of all these approaches is compared in terms of number edges to be explored in order to find a conflict free memory mapping. To do so, we uses the degree of each time node f_t and number of time nodes in the graph |T|. In this regard, runtime complexity of bipartite edge coloring is the lowest one for both Turbo and LDPC codes as compared to other approaches presented in this thesis.

Table 4. 4: Complexity Comparison of approaches used in this thesis

Algorithm Name	Modeling Used	Algorithm Used	Modeling complexity	Algorithm complexity	Complexity	Application Used
Algorithm 1	Transportation Problem	Northwest Corner	High	High	$O((f_t^2 \|T\|)(f_t!))$	Turbo
Algorithm 2	Bipartite Graph	Sub graph Partitioning	High	High	$O(f_t^2 \|T\| + 2f_t \|T\|)$	LDPC
Algorithm 3	Tripartite Graph	Sub graph Partitioning	Low	High	$O(f_t^2 \|T\| + 2f_t \|T\|)$	LDPC
Algorithm 4	Bipartite Graph	Euler Partitioning	Low	Low	***$O(k f_t^2 \|T\|)$	Turbo + LDPC

*** $k = 1$ for turbo and $k = 2$ for LDPC

5. Conclusion

In this chapter, two more approaches to tackle memory mapping problem are explained. The novelty in these approaches is that they are modeled as tripartite graph whereas in the previous chapter each approach is modeled as bipartite graph. The advantage of this modeling is that this tripartite graph can now be converted into bipartite graph on which any bipartite edge coloring algorithm already presented and proved in the literature is applied to find conflict free memory mapping. This validates our approaches by proving that it is always possible to find memory mapping using *Single and Double Memory Mapping* techniques and the algorithm finishes in polynomial time.

This chapter completes the algorithmic work performed during this thesis. At this point, one question rises in the mind that why we need lot of algorithms to solve the same problem and why algorithm based on bipartite edge coloring algorithm is presented directly? Also it can be asked that why algorithm based on bipartite edge coloring is better than other algorithms? There are lots of reasons to present all these algorithms in this work. The first reason is that we want to show the complete path through which we reach last algorithm. It is not possible that we can develop last algorithm without developing initial algorithms. First algorithm models problem as bipartite graph in which time instances and data forms two sets of nodes. This approach works well for turbo codes in which each data node is accessed only two times. However, both the edges connected with each data node should have same color so that we can apply Single memory mapping approach on turbo codes which is not possible using bipartite edge coloring algorithm. Similarly for LDPC codes, each edge in bipartite graph contains information about both read and write accesses in order to accommodate Double memory mapping approach. So again bipartite edge coloring algorithm cannot be applies on this graph. This gave us the idea that since each edge represents two accesses then why not we divide each time instance and edge so that each edge represents one access. This approach results in modeling our problem as tripartite graph but still no algorithm exists in literature to color the edges of tripartite graph. To use the algorithm proved in literature, it is necessary to transform this tripartite graph into bipartite graph. So we remove data nodes from this tripartite graph and construct a bipartite graph on which bipartite edge coloring algorithm can be applied. The thesis presents the quest to model mapping problem into class of problems in which polynomial time algorithm exists.

The second reason is that we want to model the mapping problem for both turbo and LDPC in such a manner that same algorithm can be used to find memory mapping. In this thesis, we present different algorithms for solving both turbo and LDPC mapping problems. However, at the end, both mapping problems can be solved using single bipartite edge coloring algorithm.

Third reason is that algorithm to color the edges of bipartite graph is an active domain of research in graph theory. Bipartite edge coloring is efficiently used in other scientific domains to solve problems. So either we can use the results of the other domain or use our work to solve problems in other domain. Also, in future, any improvement in computation time for coloring the edges of bipartite graph can directly be applied to improve the computation time for solving memory mapping problem.

Forth reason is that simplification in coloring the edges of bipartite graph motivates future works to add more constraints into the current mapping problem. Solving the mapping problem using polynomial time algorithm opens challenge to develop new algorithms that generate memory mapping in order to simplify network and addressing logic.

In next chapter, different experiments have been performed for generating control and network architecture for different applications and different types of parallelisms. For this purpose, a tool has been developed during this thesis work that takes access order of data for turbo and LDPC codes. Afterwards, it finds conflict free memory mapping for required number of parallelism and generates VHDL files for memory banks, network and control logic that can be synthesized on FPGA or ASIC.

Chapter 5

Experiments

Table of Contents

1. Introduction **101**
2. Design Flow for Memory Mapping Tool **102**
3. Ultra Wide Band Communication System **103**
3.1. Bit Interleaver used in WPAN IEEE 802.15.3a Physical Layer ---------- 103
3.2. Experiments and Results ---------- 106
4. Designing Parallel Interleaver Architecture for Turbo Decoder **107**
4.1. Interleaver used in HSPA Evolution ---------- 107
5. Designing Partially Parallel Architecture for LDPC Decoder **112**
5.1. Partially Parallel Architecture for structured LDPC codes ---------- 112
5.2. Decoder Architecture for Non-Binary LDPC codes ---------- 116
6. Case Study: Designing Parallel architecture for Quadratic Permutation Polynomial Interleaver **119**
6.1. Configurations used in this study ---------- 120
6.2. Experiments and Results ---------- 121
7. Conclusion **124**

In this chapter, different experiments have been performed to validate the theoretical work presented in this thesis. Chapter starts with brief presentation of the design flow utilized to perform experiments. Afterwards, parallel architecture for bit interleaver used in UWB communication system is designed to show the importance of memory mapping that supports particular interconnection network. In second experiment, first conflict free memory mapping is found for Turbo interleaver used in HSPA for different types of parallelism and then parallel hardware architecture is designed for each of these parallelism. In next two experiments, Double memory mapping approach is applied to find conflict free memory mapping for partially parallel implementation of structured LDPC codes and serial and partially parallel implementation of non-binary LDPC codes. Parallel hardware architecture is also designed for both of these approaches in this thesis. In last experiment, a study to design conflict free parallel architecture for QPP interleaver used in LTE is presented along with comparison of different configurations to design high speed LTE decoder.

1. Introduction

This chapter explores the design space for different types of interleavers used in current telecommunication standards. The biggest hurdle in designing parallel architecturew for interleaving law is that either this law is not conflict free or it is not conflict free for particular parallelism degree. Algorithms presented in this thesis can be used to find conflict free memory mapping for every type of interleaver and for every parallelism. This is demonstrated through experimental results in this chapter. In the next section, brief description of the design flow that has been to perform experiments is given.

First experiment is used to design parallel architecture for bit interleaver. Bit interleaver is a part of different telecommunication standards to tackle burst errors. Bit interleaver that is part of UWB communication system is taken as test case in this experiment. Algorithm used in this experiment is also able to find memory mapping that supports particular interconnection network. In second experiment, turbo interleavers used in different telecommunication standards for enhancing forward error correction capabilities of turbo codes are tackled. Different experiments have been performed to design parallel interleaver architectures used in HSPA Evolution. This interleaver is not conflict free for every type of turbo decoder parallelism. Conflict free memory mapping is found and hardware architecture is proposed for different decoder parallelisms in this experiment. Single memory mapping approach is utilized to find conflict free memory mapping in these experiment and both runtime of the algorithm and area of the resultant architecture are compared with the state of the art approaches.

In third experiment, partially parallel architecture is designed for structured LDPC codes. Structured LDPC codes are increasingly used in different telecommunication standards. However, allocating block of data in different memory banks results in a memory conflict problem. In this experiment, double memory mapping approach is utilized to solve memory conflict problem and different experiments have been performed to design parallel interleaver architecture for different parallelism and block sizes. In forth experiment, a parallel interleaver architecture is designed for non-binary LDPC (NB-LDPC). NB-LDPC is developed to enhance the performance of binary LDPC. However, interleaving law used in NB-LDPC is not conflict free even for serial implementation of NB-LDPC decoder. In this experiment, parallel interleaver architecture is designed for both serial and partially parallel implementation of NB-LDPC decoder. In both of these experiments, double memory mapping approach is utilized to find conflict free memory mapping. Both runtime and resultant architecture are compared with state of the art solutions.

In final experiment, both single and double memory mapping approaches are used to design parallel architecture for QPP interleaver of LTE. The goal of the experiment is to compare different configurations in order to design high speed LTE decoder. These configurations differ in their modes (shuffled or non-shuffled), schemes (butterfly or replica), radices and whether internal memory inside SISO decoder is used or not. Hardware cost and latency are calculated for each of these configurations and results are detailed to show the tradeoff between area and throughput to design LTE decoder.

2. Design Flow for performing experiments

In this section, the design flow for performing the experiments is presented. This design flow consists of three parts as shown in Figure 5. 1. In the first part, input constraint file is prepared that can be used in the next step to generate conflict free memory mapping. The different interleavers used in different standards are implemented in software, in order to automatically generate the corresponding constraint files. In second step, bipartite edge coloring algorithm for Single and Double memory mapping approaches are implemented in software. The software takes these input constraint files and generates conflict free memory mapping. The third step generates VHDL files based on the memory mapping found in the previous step. These files that can be synthesized to design complete architecture i.e. network, memory banks and associated controllers.

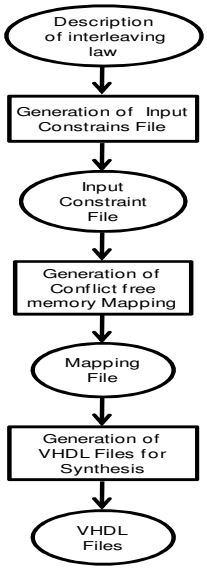


Figure 5. 1. Design Flow for performing experiments

Target Architecture

In all these experiments, Target architecture using single port memory bank is shown in Figure 5. 2. This architecture consists of processing elements, memory banks (RAM) and controller. Two extra ROMs are used to control network and to generate addresses for memory banks. Controller is designed to address these ROM and to generate R/W signal for memory banks.

In all these experiments, *B = Number of memory banks*, *P = Total number of processing elements, T = Total number of accesses to the memory and R = Number of data in each bank*. So, the size of each addressing *ROM* = $T * \lceil \log_2(R) \rceil$ such that each *ROM* contains *T* words and each word has a size of $\lceil \log_2(R) \rceil$ bits. Similarly, if the network is crossbar then the size of network *ROM* = $T * (P * \lceil \log_2(P) \rceil)$. Also the size of bus from network ROM to network is $P * \lceil \log_2(P) \rceil$ bits and the size of each bus from addressing ROM to bank is $\lceil \log_2(R) \rceil$ bits. In all these experiments, it is considered that each data has a size of 8-bits.

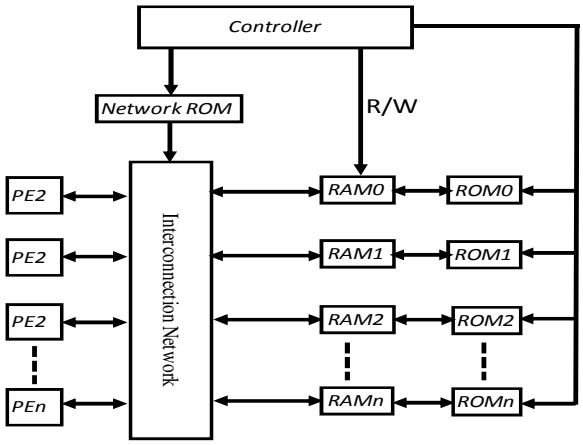


Figure 5. 2. Resultant generated architecture

Execution Platform and Targeted Technology

Results obtained in these experiments are presented in term of CPU time and area. In all these experiments, CPU time of the proposed approaches are compared with existing approaches using DELL Core i7 M 620 (2,67 Ghz) machine with 4G RAM. Also, 90 nm technology from STMicroelectronics is used in these experiments to calculate area of different components of resultant architecture. These area are given in NAND-gate equivalent to respect non-disclosure agreement with STMicroelectronics.

3. Ultra Wide Band Communication System

Ultra Wide Band (UWB) provides a promising solution for indoor and home networking due to its large bandwidth (>500 MHz). It can accommodate multiple high data rate terminals and becomes an attractive candidate for future indoor networks. The principal advantage of UWB [SIR08] [KAI05] is its potential to satisfy high data rate requirements at very low hardware cost and power consumption. Also, it provides accurate location tracking capabilities at low-rate transmission. Due to these advantages, UWB can significantly be used for high data-rate Wireless Personal Area Network (WPAN) and sensor identification networks.

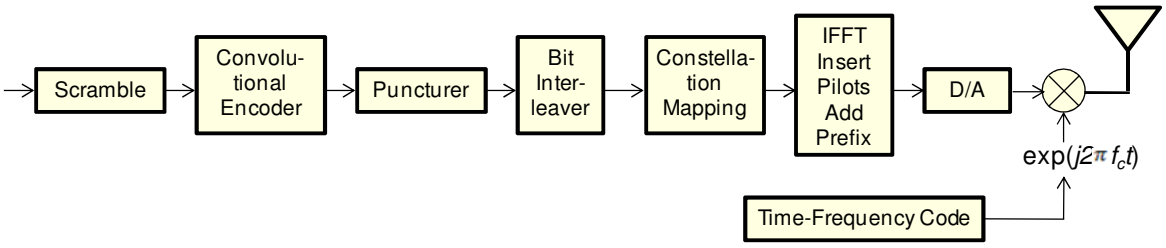
3.1. Bit Interleaver used in WPAN IEEE 802.15.3a Physical Layer

WPAN Network

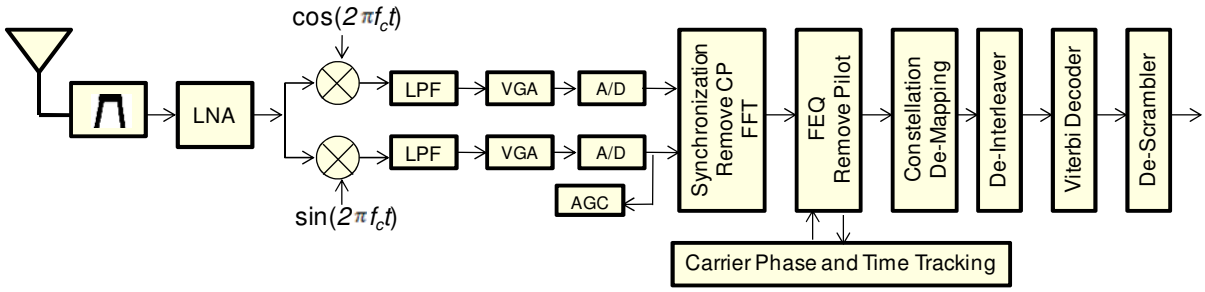
A network is called high data-rate WPAN network if it contains medium density of active devices (5 to 10) that transmit data at a rate ranging from 100 to 500Mbps within a distance of 20m.

The IEEE 802.15.3 standard task group has constituted the 805.15.3a study group to standardized UWB wireless communication for WPAN transmission. The goal of this group is to develop high-speed physical layer for applications that require imaging and multimedia.

The physical layer proposed by this group for multiband OFDM (Orthogonal Frequency Division Multiplexing) system is shown in Figure 5. 3, Figure 5. 3.a and Figure 5. 3.b shows the block diagrams for transmitter and receiver respectively used in this system. The details of all of these components can be found in [SIR08].



(a) Transmitter



(b) Receiver

Figure 5. 3. Multiband OFDM System [BAT04] (Copyright @ 2004 IEEE)

In this section, we only discuss bit interleaver that is used to provide robustness against burst errors. Bit interleaving is performed in two stages namely symbol interleaving and tone interleaving. In symbol interleaving, the bits across OFDM symbols are interleaved whereas in tone interleaving bits within an OFDM symbol are interleaved to take advantage of frequency diversity. The input-output relationships for both of these interleaving laws are as follows [BAT04]:

$$Y(i) = X\left\{ Floor(\frac{i}{N_{CBPS}}) + 6Mod(i, N_{CBPC}) \right\}$$

$$Z(i) = Y\left\{ Floor(\frac{i}{N_{T\,\mathrm{int}}}) + 10Mod(i, N_{T\,\mathrm{int}}) \right\}$$

Where *X(i)* is the input to the symbol interleaver, *Y(i)* is the output from symbol interleaver and input to the tone interleaver, *Z(i)* is the output from tone interleaver, i = 0, 1, ……, $3N_{CBPC} - 1$, *Floor(.)* and *Mod(.)* represents the floor and modulo functions respectively. N_{CBPC} is the number of bits per OFDM symbol and $N_{Tint} = N_{CBPC}/10$.

The function of interleaver can best be explained through small example. Consider an example in which i = 0, 1, ……., 29, N_{CBPC} = 10, N_{Tint} = 1. Thirty bits, equivalent to 3 OFDM symbols, are input to the bit interleaver that produces output after performing both symbol and tone interleaving. This interleaved output is:

Π(i) = 0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 10, 13, 16, 19, 22, 28, 1, 4, 7, 20, 23, 26, 29, 2, 5, 8, 11, 14, 17

To interleave all 3 OFDM symbols in parallel, 3 bits arrived at each time instance from each OFDM symbol. To shown the parallelism, the input and interleaved bits are arranged in matrix with number of columns equal to N_{CBPC} and number of rows equal to the number of OFDM symbol i.e., 3 as shown in Figure 5. 4 a and b.

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29

(a) Input order

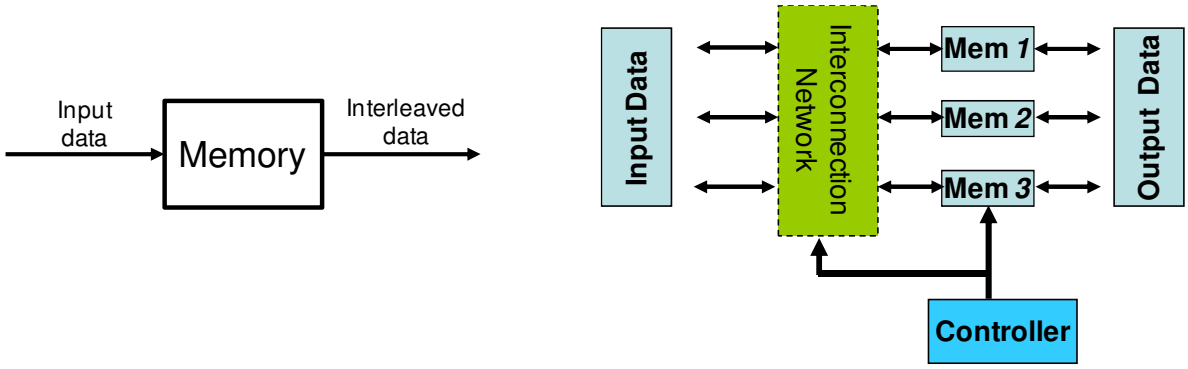
0	3	6	9	12	15	18	21	24	27
10	13	16	19	22	25	28	1	4	7
20	23	26	29	2	5	8	11	14	17

(b) Interleaved order

Figure 5. 4. Input and Interleaved order for 30 bits

Two implementations are possible to perform bit interleaving: Serial implementation or parallel implementation. In serial implementation, single bit is arrived at each time instance whereas in parallel implementation 3 bits are needed to be interleaved at each time instance. Serial implementation is quite simple and straightforward because we just need one buffer or memory to store these bits. Afterwards, these bits are read out from the memory in interleaved order to be transmitted to the next module in the system. Serial implementation is shown in Figure 5. 5.a. In parallel implementation (as shown in Figure 5. 5.b), main memory is divided into three memory banks to achieve high memory bandwidth and to accommodate bits that arrive in parallel. Interconnection network is required to permute bits that arrive in parallel.

The problem is to store data in memory banks in such a manner that data required in interleaved order in parallel can be accessed from each memory bank without any conflict. Another objective is to design interleaving architecture using minimal hardware.



(a) Serial Implementation (b) Parallel Implementation

Figure 5. 5. Implementation of Bit Interleaver

3.2. Experiments and Results

In this section, experiments have been performed for different codeword length and different parallelisms. The goal of this experiment is to show the interest and limits of the algorithm originally developed to tackle transportation problem. We also wanted to determine whether targeted codeword and parallelism supports single barrel shifter based architecture, for both natural and interleaved orders. The experiments have been performed for codeword lengths of 300, 600, 1200 and for parallelism of 2, 3, 4 as shown in Table 5. 1.

Table 5. 1. CPU time (second) for various Memory Mapping Approaches

Total Data	Parallelism	Single BS possible	Bipartite Edge Coloring Algorithm	Algorithms based on Transportation Problem	[CHA10a]
300	2	Yes	0,016	0,031	0,031
300	3	Yes	0,016	0,016	0,03
300	4	No	0,015	NA	0,031
600	2	Yes	0,015	0,062	0,062
600	3	No	0,016	NA	0,078
600	4	No	0,015	NA	0,109
1200	2	Yes	0,031	0,202	0,202
1200	3	No	0,109	NA	0,405
1200	4	No	0,031	NA	0,265

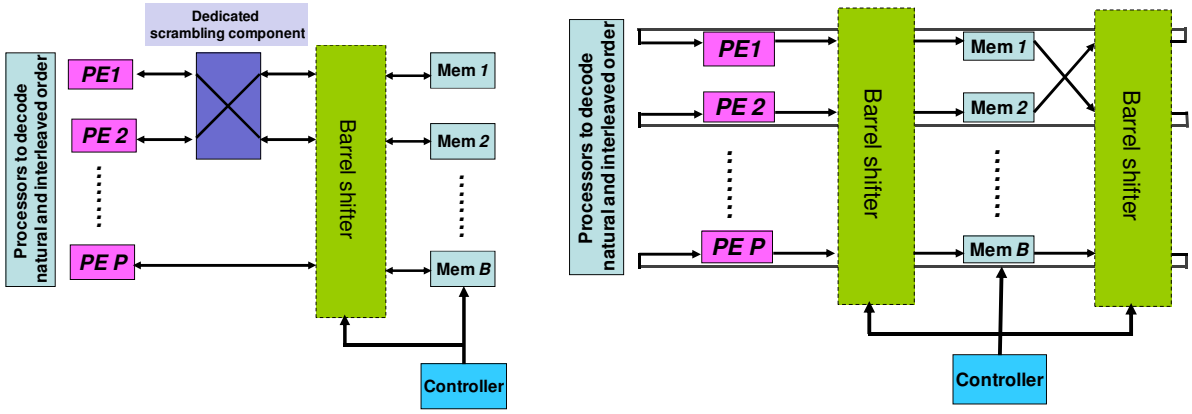
The “transportation algorithm” is compared with [CHA10a] and bipartite edge coloring approach. When barrel shifter is possible, then the runtime of algorithm inspired from transportation problem domain is same as [CHA10a] since the length of codeword is not quite large. However, our approach stops as soon as it detects barrel shifter cannot be used while [CHA10a] continues searching and exits when a conflict free memory mapping has been found. Afterwards, bipartite edged coloring takes less time than the other approaches ([CHA10a] is 354 % slower on average) to determine memory mapping for bit interleaver. However, this approach is not able to generate a memory mapping with respect to a user defined interconnection network, because it does not support any design constraint. This requires to use a complete network which supports any permutations (e.g. crossbar, Benes…).

Resultant architecture is generated after finding mapping for bit interleaver. VHDL files are generated for memory banks, barrel shifter or crossbar, addressing ROMs (address controller) and network ROM (NW controller) for parallel architecture. The resultant area for K = 300 and B = 3 with barrel shifter and cross bar network using targeted technology is shown in Table 5. 2

Table 5. 2. Resultant area of different components for parallel implementation of bit interleaver

K = 300	NW Cost	NW (BS) Controller	Extrinsic Memory	Address Controller	Total Area
Barrel Shifter	256	14350	172200	150675	337481
Crossbar	384	43050	172200	150675	366309

From this table, it is clear that total area of the generated architecture using a crossbar network is greater than barrel shifter based architecture. This mainly comes from the interconnection network and its associated controller. This shows the interest of the approach that finds a conflict free memory mapping that respects a targeted interconnection network.



a. Single barrel shifter based architecture b. Double barrel shifter based architecture

Figure 5. 6. Comparison of two different architectures

Moreover, the "transportation algorithm", i.e. our proposed method inspired from the transportation problem domain, only supports single barrel shifter based memory mapping. This is due to its internal representation which formalizes both natural and interleaved access orders in a single matrix (see Figure 5.6.a). However, [CHA10a] has been supported memory mapping for double barrel shifter architecture (natural and interleaved access, see Figure 5.6.b) because it is based on a double matrices internal model. In the future, additional constraints and improvements could be added in the current "transportation algorithm" to support such architecture model.

4. Designing Parallel Interleaver Architecture for Turbo Decoder

Currently turbo codes are used in different standards. However, all of these interleavers are not conflict free for every type of parallelism used in turbo decoding. The proposed approach can be able to find conflict free memory mapping for any type of interleaver and for any type of parallelism. However, for experimental purpose, we implemented interleavers used in two of the most widely used standards: HSPA Evolution [HSP08] and UMTS LTE [LTE08]. Both of these standards are used in wireless communication for handheld devices. In this section, parallel implementation of interleaver used in HSPA Evolution is discussed

4.1. Interleaver used in HSPA Evolution

After the release of initial draft by 3GPP-WCDMA, series of specification have been released from time to time for *high speed packet access* (HSPA). With the use of more channels and addition of modulation scheme, Release 5 [HSP04] has upgraded the HSPA to *high speed downlink packet access* (HSDPA) to support high data rate applications. It increased the data throughput up to 14Mbps. With

the addition of MIMO and 64 QAM, HSPA is evolved into HSPA+ in Release 8 [HSP08] with the throughput requirement of up to 43.2Mbps. To obtain high throughput, it is necessary to perform turbo decoding on parallel architecture. However, the interleaver used in HSPA+ is not conflict free to support parallel implementation of turbo decoder. Also it is necessary to design interleaver architecture that support wide range of block sizes used in HSPA+ i.e. from 40 to 5114.

4.1.1. Algorithm for HSPA Interleaver

The interleaving algorithm for HSPA+ defined in [HSP04] is mentioned below:

K Number of bits input to Turbo code internal interleaver

R Number of rows of rectangular matrix described in standard

C Number of columns of rectangular matrix described in standard

p Prime number described in standard

v Primitive root describe in standard

- Determine *R* of the rectangular matrix, such that

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

- Determine value of *p* and *C*, such that

if (481 ≤ K ≤ 530) then

p = 53 and C = p

else

Find *p* from Table 5. 3 such that

$K \le R*(p + 1)$,

and determine *C* of matrix such that,

$C = p - 1;$ *if* $(K \le R * (p-1))$

$C = p;$ *if* $(R*(p-1) < N \le R * p))$

$C = p + 1;$ *if* $(R*p < N)$

Table 5. 3. List of prime number *p* and associated primitive root *v*

p	*v*	*p*	*v*	*p*	*v*	*p*	*v*	*p*	*v*
7	9	47	5	101	2	157	5	223	3
11	2	53	2	103	5	163	2	227	2
13	2	59	2	107	2	167	5	229	6
17	3	61	2	109	6	173	2	233	3
19	2	67	2	113	3	179	2	239	7
23	5	71	7	127	3	181	2	241	7
29	2	73	5	131	2	191	19	251	6
31	3	79	3	137	3	193	5	257	3
37	2	83	2	139	2	197	2		
41	6	89	3	149	2	199	3		
43	3	97	5	151	6	211	2		

- Write the input bit sequence into the rectangular matrix row by row and if R*C > K, the dummy bits are padded to fill the matrix.
- Construct the base sequence *S(j)* for intra-row permutation as:
  $S(j) = [v * S(j-1)] \% p;$ where $j = 1,2, \ldots\ldots, p-2$
- Determine the least prime integer sequence $q(i)$ for $i = 1, 2, \ldots\ldots, R-1$, by assigning
  $q(0) = 1,$ such that $gcd(q(i), p-1) = 1$ and $q(i) > 6$ and $q(i) > q(i-1).$
- Permute the sequence q(i) to construct the sequence $r(i)$ such that
  $r_{T(i)} = q(i)$ where $i = 0, 1, \ldots., R-1$ and $T(i)$ is the inter-row permutation defined in the standard.
- Perform the intra row permutation $U_i(j)$, such that
  for $i = 0, 1,\ldots., R-1$ and $j = 0,1, \ldots\ldots, p-2$;
  If $(C = p)$ then
  $U_i(j) = S[(j*r(i)) mod (p-1)]$ and $U_i(p-1) = 0;$
  If $(C = p+1)$ then
  $U_i(j) = S[(j*r(i)) mod (p-1)]$ and $U_i(p-1) = 0$ and $U_i(p) = p$
  and $if (K = R*C)$ then exchange $U_{R-1}(p)$ with $U_{R-1}(0)$
  if $(C = p-1)$ then
  $U_i(j) = S[(j*r(i)) mod (p-1)] – 1$
- Perform the inter row permutation of the matrix based on the pattern $T(i)$ where $T(i)$ is the original row position of the *i-th* permuted row and defined in the standard.
- Read the bits column by column from the rectangular matrix by deleting the dummy bits padded to the input bits sequence.

The algorithm can be explained best through a small example of K = 44. Different parameters obtained from the specifications explained previously are:

$R = 5, C = 10, p = 11, v = 2$

Next step is to put 44 data into matrix of order 5*10 (R*W) starting from row 0. Since there are 50 cells in the matrix, so the last 6 cells are filled with dummy bits represented by -1 in the last row as shown in Figure 5. 7.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
20	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**
40	**41**	**42**	**43**	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**

Figure 5. 7. Arrangement of $K = 44$ data into 5*10 matrix

Afterward, values for sequences *s, q, r, u* are calculated based on the rules defined in the standard. These values are:

S = 1 2 4 8 5 10 9 7 3 6 where number of values in S is $(p-1) = 10$

q = 1 7 11 13 17 where number of values in q is $R = 5$

r = 17 13 11 7 1 where number of values in q is $R = 5$

Value of U

0 6 4 1 2 9 3 5 8 7

0 7 8 5 3 9 2 1 4 6

0 1 3 7 4 9 8 6 2 5
0 6 4 1 2 9 3 5 8 7
0 1 3 7 4 9 8 6 2 5

where number of values in U is $R*C = 5*10 = 50$

The values in U are used to perform intra-row permutation. First row of U values are used to permute values in first row of matrix. For the values calculated for this example, first value remain at the first place, second value is permuted to sixth value, third value is permuted to forth value and so on. The matrix after intra-row permutation is shown in Figure 5. 8.

0	**3**	**4**	**6**	**2**	**7**	**1**	**9**	**8**	**5**
10	**17**	**16**	**14**	**18**	**13**	**19**	**11**	**12**	**15**
20	**21**	**28**	**22**	**24**	**29**	**27**	**23**	**26**	**25**
30	**33**	**34**	**36**	**32**	**37**	**31**	**39**	**38**	**35**
40	**41**	**-1**	**42**	**-1**	**-1**	**-1**	**43**	**-1**	**-1**

Figure 5. 8. Matrix after Intra-row Permutation

In the last step, inter-row permutation is performed on rectangular matrix using the permutation pattern defined in the standard. For this example, inter-row permutation pattern is:

$T = 4, 3, 2, 1, 0$ where number of values in T is $R = 5$

The matrix after inter-row permutation is shown in Figure 5. 9.

40	**41**	**-1**	**42**	**-1**	**-1**	**-1**	**43**	**-1**	**-1**
30	**33**	**34**	**36**	**32**	**37**	**31**	**39**	**38**	**35**
20	**21**	**28**	**22**	**24**	**29**	**27**	**23**	**26**	**25**
10	**17**	**16**	**14**	**18**	**13**	**19**	**11**	**12**	**15**
0	**3**	**4**	**6**	**2**	**7**	**1**	**9**	**8**	**5**

Figure 5. 9. Matrix after Inter-row Permutation

Afterwards, the values in the matrix are read out column by column after pruning dummy bits to construct interleaved order of data values. An interleaved order for $K = 44$ is:

Interleaved order = Π = 40 30 20 10 0 41 33 21 17 3 34 28 16 4 42 36 22 14 6 32 24 18 2 37 29 13 7 31 27 19 1 43 39 23 11 9 38 26 12 8 35 25 15 5

4.1.2. Experiments and Results

The HSPA interleaving law has been used in this second set of experiments to show the performance of bipartite edge coloring algorithm for finding Single Memory Mapping. Table 5. 4. shows the CPU time in seconds for proposed approach and the existing approaches [CHA10a] and [TAR04] for K = 5120 and P = 4, and for different types of parallelism (see chapter 1) used in turbo decoding.

Table 5. 4. CPU time (seconds) for various Memory Mapping Approaches

K = 5120, P = 4	Edge Coloring	[CHA10a]	[TAR04]
Forward Backward Mapping	0,42	10,6	1,3
Butterfly Mapping	0,483	32,9	2,1
Butterfly with Radix 4 Mapping	0,374	Failed	2,3
Forward Backward with Radix 4 Mapping	0,483	Failed	2,4

The table indicates that the computational time of edge coloring approach is up to 70 times less than the existing approaches. Also, as the length of the frame size increases, the time to calculate a memory mapping using edge coloring approach remains nearly constant compared to [CHA10a] and [TAR04]. Moreover, heuristic presented in [CHA10a] fails to find memory mapping for Butterfly with Radix 4 and Forward backward with Radix 4 parallelisms. These experiments clearly indicate a great advantage of polynomial time algorithm (designed for every type of parallelism) in terms of computational time compared to [CHA10a] and [TAR04]. The comparison is depicted graphically in Figure 5. 10.

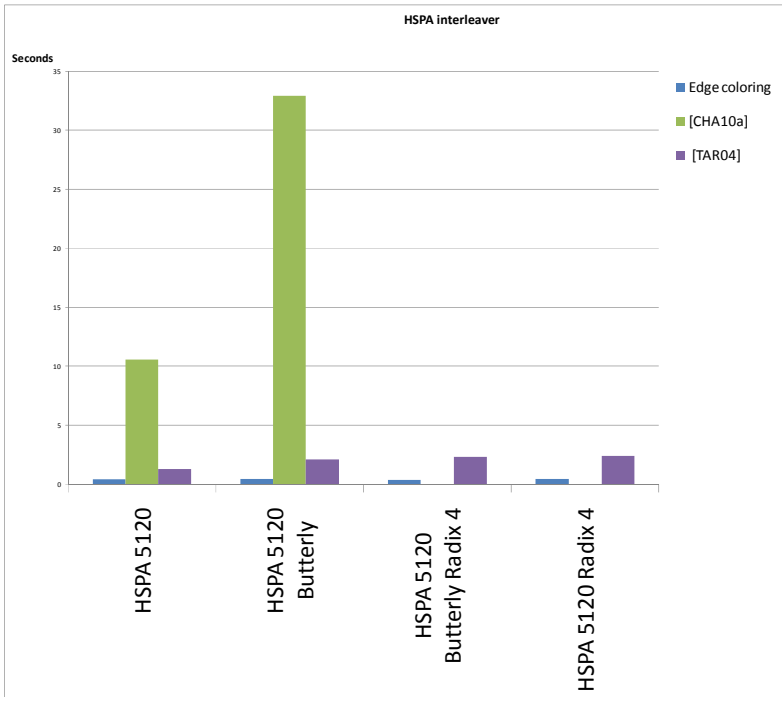


Figure 5. 10. Comparison of CPU time for various Mapping Approaches

Targeted architecture (see Figure 5. 2) is generated after finding the conflict free memory mapping. It contains 4 memory banks for forward backward parallelism, 8 memory banks for Butterfly and Forward Backward with Radix 4 parallelism and 16 memory banks for Butterfly with Radix 4 parallelism. VHDL files have been generated for memory banks, crossbar network, addressing ROMs (address controllers) and network ROM (NW controller) for each type of parallelism. The resultant area using targeted technology is shown in Table 5. 5.

Table 5. 5. Resultant area (in number of NAND-gates) of different components for different types of Turbo decoder parallelism

K = 5120, P = 4	NW Cost	NW Controller	Extrinsic Memory	Address Controller	Total Area
Forward Backward Parallelism	96	10240	40960	56320	107616
Butterfly Parallelism	448	15360	40960	51200	107968
Forward Backward with Radix 4 Parallelism	448	15360	40960	51200	107968
Butterfly with Radix 4 Parallelism	1920	20480	40960	46080	109440

From this table, it is clear that NW controller and Address Controller are the two components that occupy the most area in designing resultant architecture for Turbo decoder. To reduce total area of the architecture, optimization is required to reduce the size of the ROMs that store addressing and network control information. This could also be interesting in implementing different codewords or parallelism on same chip. This point will be developed in the perspectives.

5. Designing Partially Parallel Architecture for LDPC Decoder

Low density parity check Codes (LDPC) has error correction capabilities very close to the channel capacity and it has already included in several wireless communication standards to reliably transfer data between source and destination. However, for high data rate application, implementation of LDPC Decoder on partially parallel architecture suffers from memory conflict problem (explained in Chapter 1 section 4.2). In this section, bipartite edge coloring algorithm is applied to solve memory mapping problem for structured LDPC codes and non-binary LDPC.

5.1. Partially Parallel Architecture for structured LDPC codes

Structured LDPC codes are part of current telecommunication standards [WIF08] [WIM06] for performing forward error correction. To achieve high data rate, these codes are implemented using partially parallel architecture. However, partially parallel architecture suffers from memory conflict problem as discussed in chapter 1 section 4. To tackle this memory mapping problem, in this thesis, we use bipartite edge coloring algorithm to solve this problem using *Double Memory Mapping* in polynomial time.

5.1.1. Description of Partially Parallel Architecture

In partially parallel architecture, there are P processing elements that are equal to the number of check nodes (CN) to be processed in parallel. To achieve high memory bandwidth, main memory is divided into B number of memory banks. Each P has B number of connections in order to access all the memory banks in one cycle. The problem is to allocate each Z matrix in a memory bank in such a

manner that there is no conflict in accessing the data concurrently in parallel at each time instance. The architecture can be explained best through a small example taking from WiMAX standard. Figure 5. 11 represents H_{Base} matrix for WiMAX standard with following characteristics:
W = number of rows = 12, Y= number of columns = 24, $d_{c,max}$ *= maximum check node degree = 7*
*Z = 32, codeword size = Y * Z = 768 and code rate = Y - W / Y = ½.*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	-1	94	73	-1	-1	-1	-1	-1	55	83	-1	-1	7	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	27	-1	-1	-1	22	79	9	-1	-1	-1	12	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	24	22	81	-1	33	-1	-1	-1	0	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1
4	61	-1	47	-1	-1	-1	-1	-1	65	25	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1
5	-1	-1	39	-1	-1	-1	84	-1	-1	41	72	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	46	40	-1	82	-1	-1	-1	79	0	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1
7	-1	-1	95	53	-1	-1	-1	-1	-1	14	18	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1
8	-1	11	73	-1	-1	-1	2	-1	-1	47	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1
9	12	-1	-1	-1	83	24	-1	43	-1	-1	-1	51	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1
10	-1	-1	-1	-1	-1	94	-1	59	-1	-1	70	72	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1
11	-1	-1	7	65	-1	-1	-1	-1	39	49	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0
12	43	-1	-1	-1	-1	66	-1	41	-1	-1	-1	26	7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0

Figure 5. 11. H_{Base} Matrix for WiMAX Standard of code word size = 576, Z = 24 and r = 1/2

For this example, proposed partially parallel architecture is shown in Figure 5. 12. In this architecture, $B = d_{c,max} = 7$, so that at all time instances, we can access all check nodes in one cycle. Also for this example, $P = Z = 32$ so that one Z matrix can be processed in one cycle. However, any combination of B and P can be used to find conflict free memory mapping in order to fulfill the throughput requirement of targeted application.

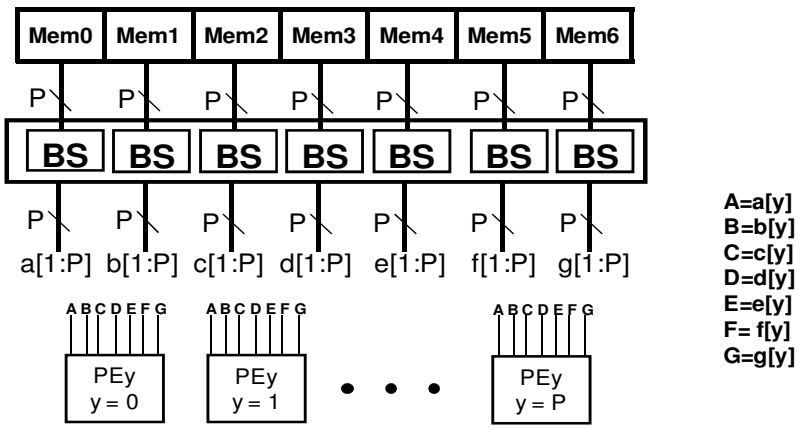


Figure 5. 12. Partially parallel architecture for ½ Rate WiMAX standtard

The first step is to prepare data access matrix (explained in chapter 3 section 3.1). This data access matrix contains all the Z-matrix that need to be accessed in parallel. Data access matrix for this example is shown in Figure 5. 13. From this figure, it is clear that at cycle 1, Z matrices 2, 3, 9, 10, 13, 14 are needed to be accessed in parallel. It means that these Z matrices should be stored in different memory bank in order to access them concurrently in parallel without any conflict.

2	2	4	1	3	5	3	2	1	6	3	1
3	6	5	3	7	6	4	3	5	8	4	6
9	7	6	9	10	8	10	7	6	11	9	8
10	8	8	10	11	12	11	10	8	12	10	12
13	12	12	16	17	13	19	20	12	22	23	13
14	14	15	17	18	18	20	21	21	23	24	24
	15	16			19			22			

Parallelism

t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12}

Time

Figure 5. 13. Data Access Matrix for ½ Rate WiMAX standtard

In order to support *Double Memory Mapping*, this access matrix is converted into mapping matrix (explained in chapter 3 section 3.1) with the addition of two more columns at each time instance. First column stores all the banks from where processing elements fetched data for processing and the second column stores all the banks from where these processed data need to be written back. Afterwards, this mapping matrix is converted into bipartite graph (explained in chapter 4 section 3.2) and bipartite edge coloring algorithm is used to find memory mapping. Resultant memory mapping along with mapping matrix is shown in Figure 5. 14.

	R	W		R	W		R	W		R	W		R	W		R	W		R	W		R	W		R	W		R	W		R	W
2	b_0	b_0	**2**	b_0	b_0	**4**	b_0	b_2	**1**	b_0	b_4	**3**	b_0	b_0	**5**	b_0	b_0	**3**	b_0	b_4	**2**	b_0	b_0	**1**	b_4	b_5	**6**	b_0	b_0	**3**	b_4	b_1	**1**	b_5	b_0
3	b_1	b_4	**6**	b_1	b_4	**5**	b_1	b_0	**3**	b_4	b_0	**7**	b_1	b_1	**6**	b_4	b_2	**4**	b_2	b_0	**3**	b_4	b_4	**5**	b_0	b_1	**8**	b_2	b_1	**4**	b_0	b_0	**6**	b_0	b_1
9	b_4	b_1	**7**	b_2	b_1	**6**	b_4	b_4	**9**	b_1	b_1	**10**	b_2	b_4	**8**	b_1	b_6	**10**	b_4	b_2	**7**	b_1	b_2	**6**	b_2	b_0	**11**	b_1	b_4	**9**	b_1	b_4	**8**	b_1	b_4
10	b_2	b_5	**8**	b_4	b_2	**8**	b_2	b_1	**10**	b_5	b_2	**11**	b_4	b_5	**12**	b_5	b_3	**11**	b_5	b_1	**10**	b_2	b_5	**8**	b_6	b_2	**12**	b_4	b_2	**10**	b_5	b_2	**12**	b_2	b_5
13	b_3	b_2	**12**	b_5	b_6	**12**	b_6	b_5	**16**	b_3	b_5	**17**	b_3	b_2	**13**	b_2	b_4	**19**	b_1	b_6	**20**	b_3	b_3	**12**	b_3	b_4	**22**	b_3	b_5	**23**	b_3	b_5	**13**	b_4	b_3
14	b_5	b_3	**14**	b_3	b_5	**15**	b_3	b_6	**17**	b_2	b_3	**18**	b_5	b_3	**18**	b_3	b_5	**20**	b_3	b_3	**21**	b_6	b_1	**21**	b_1	b_6	**23**	b_5	b_3	**24**	b_2	b_3	**24**	b_3	b_2
			15	b_6	b_3	**16**	b_5	b_3							**19**	b_6	b_1							**22**	b_5	b_3									

Parallelism

t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12}

Time

Figure 5. 14. Resultant Mapping for data access matrix of Figure 5. 13

5.1.2. Experimental Results

In this section, different experiments have been performed to expose the interest of bipartite edge coloring in order to find conflict free memory mapping for structured LDPC codes on partially parallel architectures. Table 5. 6. shows the CPU time (in seconds) for proposed approach and the existing approaches [CHA10] [CHA10a]. Test case: WiMAX ½ using Z = P = 32, B =7 and for WiMAX ¾ using Z = P = 32, B =10 for partially parallel architecture of structured LDPC code.

The table shows that the runtime of the proposed approach and [CHA10] the algorithms are almost same. However, this is mainly due to the reason that we are mapping Z matrices, not the data in

the standard. Due to limited number of Z matrices in current standards, runtime is almost is the same for both the algorithms. As explained in previous experiment, the difference in polynomial time algorithm and heuristic becomes clear as the size of data to be mapped increased. The method proposed in [CHA10a] does not work in this experiment because it has been designed to find Single Memory Mapping. This gives another advantage to our method that works for both Single and Double Memory Mapping approaches whereas other works present different heuristics for both mapping approaches. The comparison is depicted graphically in Figure 5. 15.

Table 5. 6. CPU time (seconds) for various Memory Mapping Approaches

	Edge Coloring	[CHA10a]	[CHA10]
WiMAX ½	0,015	Not worked	0.015
WiMAX ¾	0.015	Not worked	0.016

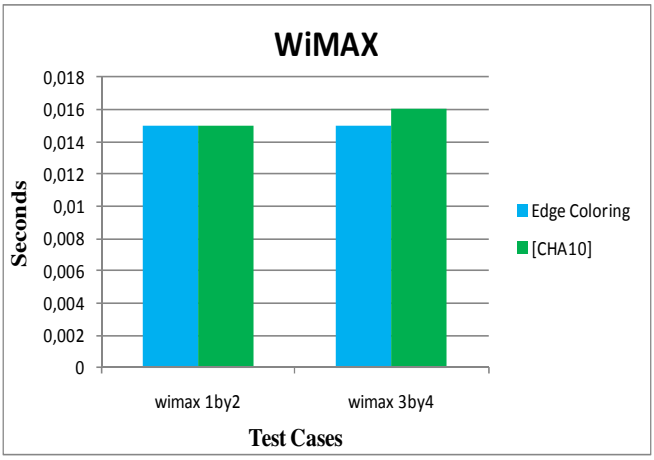


Figure 5. 15. Comparison of CPU time for various Mapping Approaches

Resultant architecture is generated after finding mapping for Z matrices. VHDL files are generated for memory banks, crossbar network, addressing ROMs (address controller) and network ROM (NW controller) for WiMAX ½ and WiMAX ¾. The resultant area for each component of this architecture is shown in Table 5. 7.

Table 5. 7. Resultant area of different components for structured LDPC Decoder Architecture

	NW(BS) Cost	NW Controller	Extrinsic Memory	Address Controller	Total Area
WiMAX ½	71680	30135	440832	18081	560728
WiMAX ¾	102400	28700	440832	22960	594892

Once again, NW controller and Address controller occupies most of the area in resultant architecture of structured LDPC codes. To reduce the chip area and also to support implementation of

multiple code rates, it is necessary to reduce the size of control and addressing ROM. This needs to add some additional constraints in current algorithms to support optimization of addressing and network control logic. These optimizations are explored in the PhD thesis of Aroua Briki, under the direction of E. Martin.

5.2. Decoder Architecture for Non-Binary LDPC codes

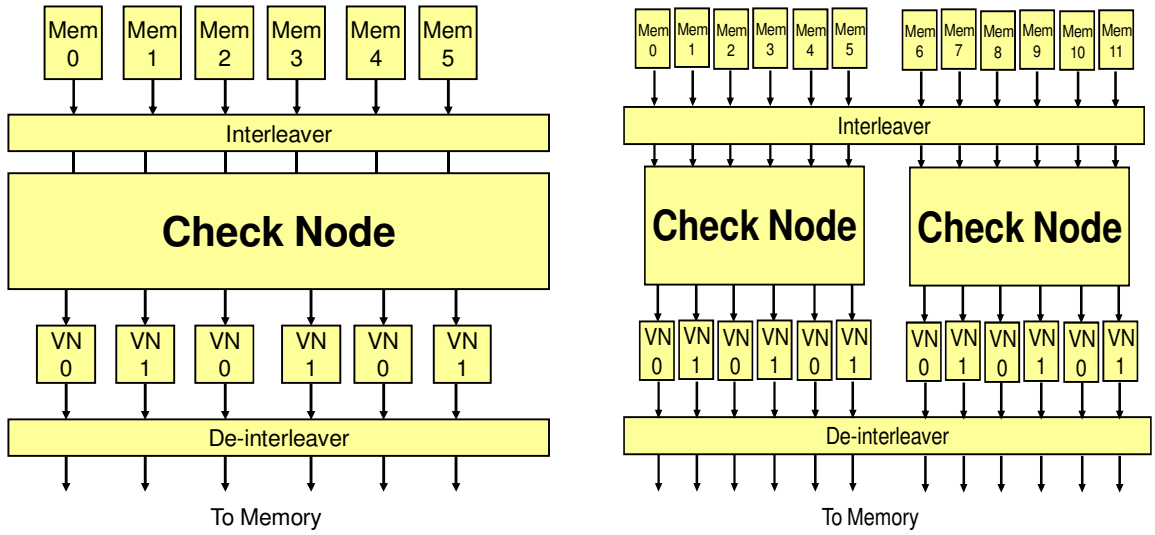
An extension of binary LDPC codes has been developed to further reduce the gap of performance with Shannon limit. This new class of codes is known as non-binary LDPC (NB-LDPC) codes [DAV]. These codes improve the performance of binary LDPC codes for small and moderate codeword lengths. However, increase in decoder complexity for NB-LDPC motivates to develop decoding algorithms that are easily implementable. Also, unlike structured codes, routing of the edges of tanner graph is not regular and even implementing NB-LDPC on serial architecture suffers from memory conflict problem. In this experiment, conflict free memory mapping for both serial and partially parallel architecture is found using *Double Memory Mapping* in polynomial time

5.2.1. Decoder Architecture for Non-Binary LDPC codes

The DAVINCI project [DAV], funded by the European Commission under the seventh framework (FP7) of collaborative research, designs the novel NB-LDPC codes and related link level technologies. The purpose of this project is to construct codes that are suitable for implementation and outperform the state of the art techniques to design NB-LDPC codes.

Typical serial decoder architecture [DAV] for NB-LDPC codes developed in DAVINCI project is shown in Figure 5. 16.a. This decoder is used to decode NB-LDPC codes with check node degree = d_c = 6 and variable node degree = d_v = 2. The decoder consists on one CN processor and six VN processors. The decoder is designed based on serial implementation to process one check node at each cycle. To achieve high memory bandwidth, main memory is divided into d_c number of memory banks to simultaneously receive d_c messages from memory. The interleaver and deinterleaver are designed to transfer data between CN processor, VN processors and memory banks.

For partially parallel architecture, two check nodes are processed in parallel at the same time as shown in Figure 5. 16.b. The main memory is divided into $2*d_c$ number of memory banks to concurrently fetch $2*d_c$ messages from memory.



(a) Serial Architecture (b) Partially Parallel Architecture

Figure 5. 16. Architecture for NB-LDPC

The problem is to allocate messages into memory banks in such a manner that at each cycle CN processor can fetch d_c number of messages from d_c number of memory banks concurrently without any conflict.

The first step is to prepare data access matrix (explained in chapter 3 section 3.1) for both the architectures. Data access Matrices for serial architecture and partially parallel architecture is shown in Figure 5. 17 and Figure 5. 18. For serial architecture, at each cycle, six data elements are needed to be accessed in parallel whereas for partially parallel architecture 12 data elements are needed to be accessed in parallel. To find conflict free memory mapping, all data should be stored in each memory bank in such a manner that there is no conflict in accessing them in each cycle.

2	48	10	19	62	53	23	42	59	51	29	47	33	46	26	7	25	58	16	28	17	5	63	6	36	1	21	50	15	44	11	35
3	49	11	20	63	54	24	43	60	52	30	48	34	47	27	8	26	59	17	29	18	6	64	7	37	2	22	51	16	45	12	36
149	146	139	104	159	112	67	79	156	113	97	85	142	81	130	84	115	184	83	70	155	167	65	101	78	136	132	108	181	87	116	102
122	100	123	94	84	90	114	154	133	183	110	72	70	164	119	146	181	142	127	159	109	130	106	155	184	139	98	71	68	112	96	174
90	88	86	160	126	137	178	129	73	148	191	177	172	166	80	103	153	89	128	111	152	118	176	93	157	92	171	141	107	131	140	77
157	153	179	177	182	172	158	151	124	89	141	147	88	107	148	168	66	95	143	170	75	86	131	173	103	111	126	175	144	186	178	150
t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	t_{19}	t_{20}	t_{21}	t_{22}	t_{23}	t_{24}	t_{25}	t_{26}	t_{27}	t_{28}	t_{29}	t_{30}	t_{31}	t_{32}

18	61	9	20	38	56	34	37	40	54	31	14	3	45	12	55	41	8	32	57	39	27	13	49	30	52	22	64	43	4	60	24
19	62	10	21	39	57	35	38	41	55	32	15	4	46	13	56	42	9	33	58	40	28	14	50	31	53	23	1	44	5	61	25
96	119	174	68	109	164	98	154	106	121	125	133	183	71	91	100	117	114	169	120	123	127	74	122	190	72	189	135	162	185	110	94
156	149	83	97	85	116	190	121	169	132	78	102	91	101	115	162	81	74	67	171	134	87	108	79	117	125	136	104	167	65	189	185
175	180	99	134	170	69	82	165	161	179	188	186	138	192	173	143	168	187	144	151	95	163	145	158	66	182	75	76	105	124	147	150
105	69	129	187	145	191	138	80	77	73	135	92	128	165	161	76	163	137	118	93	180	160	188	176	152	140	113	192	82	166	99	120
t_{33}	t_{34}	t_{35}	t_{36}	t_{37}	t_{38}	t_{39}	t_{40}	t_{41}	t_{42}	t_{43}	t_{44}	t_{45}	t_{46}	t_{47}	t_{48}	t_{49}	t_{50}	t_{51}	t_{52}	t_{53}	t_{54}	t_{55}	t_{56}	t_{57}	t_{58}	t_{59}	t_{60}	t_{61}	t_{62}	t_{63}	t_{64}

Figure 5. 17. Data Access Matrix for D = 192 and d_c = 6 using Serial Architecture

2	10	62	23	59	29	33	26	25	16	17	63	36	21	15	11	18	9	38	34	40	31	3	12	41	32	39	13	30	22	43	60
3	11	63	24	60	30	34	27	26	17	18	64	37	22	16	12	19	10	39	35	41	32	4	13	42	33	40	14	31	23	44	61
149	139	159	67	156	97	142	130	115	83	155	65	78	132	181	116	96	174	109	98	106	125	183	91	117	169	123	74	190	189	162	110
122	123	84	114	133	110	70	119	181	127	109	10 6	184	98	68	96	156	83	85	190	169	78	91	115	81	67	134	108	117	136	167	189
90	86	126	178	73	191	172	80	153	128	152	176	157	171	107	140	175	99	170	82	161	188	138	173	168	144	95	145	66	75	105	147
157	179	182	158	124	141	88	148	66	143	75	131	103	126	144	178	105	129	145	138	77	135	128	161	163	118	180	188	152	113	82	99
48	19	53	42	51	47	46	7	58	28	5	6	1	50	44	35	61	20	56	37	54	14	45	55	8	57	27	49	52	64	4	24
49	20	54	43	52	48	47	8	59	29	6	7	2	51	45	36	62	21	57	38	55	15	46	56	9	58	28	50	53	1	5	25
146	104	112	79	113	85	81	84	184	70	167	101	136	108	87	102	119	68	164	154	121	133	71	100	114	120	127	122	72	135	185	94
100	94	90	154	183	72	164	146	142	159	130	155	139	71	112	174	149	97	116	121	132	102	101	162	74	171	87	79	125	104	65	185
88	160	137	129	148	177	166	103	89	111	118	93	92	141	131	77	180	134	69	165	179	186	192	143	187	151	163	158	182	76	124	150
153	177	172	151	89	147	107	168	95	170	86	173	111	175	186	15	69	187	191	80	73	92	165	76	137	93	160	176	140	192	166	120
t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	t_{19}	t_{20}	t_{21}	t_{22}	t_{23}	t_{24}	t_{25}	t_{26}	t_{27}	t_{28}	t_{29}	t_{30}	t_{31}	t_{32}

Figure 5. 18.Data Access Matrix for D = 192 and d_c = 6 using Partially Parallel Architecture

To find memory mapping using *Double Memory Mapping* approach, data access matrix is converted to mapping matrix (explained in chapter 3 section 3.1) to store read and write accessed to the data at each time instance. Afterwards, this mapping matrix is converted into bipartite graph (explained in chapter 4 section 3.2) and bipartite edge coloring algorithm is used to find memory mapping. Resultant memory mapping along with mapping matrix explained in Annexure.

5.2.2. Experimental Results

Some experiments have been performed with NB-LDPC codes on serial and partially parallel architecture. Since [CHA10a] is not able to deal with Double Memory Mapping, as shown in previous experiment, we do not use this approach for the current experience. Table 5. 8 shows the CPU time (in seconds) for proposed approach and the existed approaches [CHA10] for NB-LDPC using P = 6, B = 6, K = 192 and using P = 12, B = 12, K = 192

Table 5. 8. CPU time for various Memory Mapping Approaches

	Edge Coloring	[CHA10a]	[CHA10]
NB-LDPC (paral = 6)	0,047	Not worked	0,0473
NB-LDPC (paral = 12)	0,031	Not worked	Failed

The table shows the runtime of both the algorithms are almost the same in serial architecture. However, this is mainly due to the small size of codeword in the standard. Also for partially parallel architecture, heuristic fails to find memory mapping. The comparison is depicted graphically in Figure 5. 19.

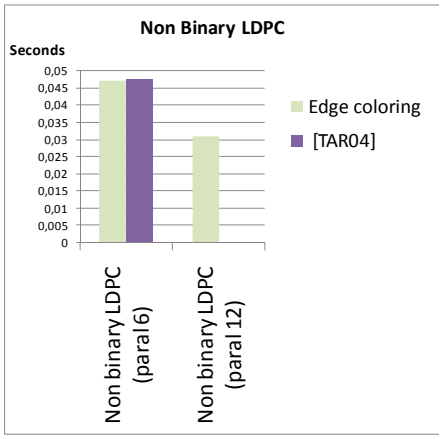


Figure 5. 19. Comparison of CPU time for various Mapping Approaches

The VHDL architecture has been generated after finding mapping for NB-LDPC. The resultant area for each component of this architecture is shown in Table 5. 9

Table 5. 9. Resultant area of different components for NB-LDPC Decoder Architecture

	NW Cost	NW Controller	Extrinsic Memory	Address Controller	Total Area
Serial Architecture	**1920**	**144**	**110208**	**137760**	**250032**
Partially Parallel Architecture	**8448**	**384**	**110208**	**110208**	**229248**

NW controller and Address controller still take most of the area in the resultant architecture of NB-LDPC codes, about 50%, as depicted in Table 5. 9. As it has been previously shown, the optimization of these parts of the design requires to improve our current approach by adding some additional constraints in our models.

6. Case Study: Designing Parallel architecture for Quadratic Permutation Polynomial Interleaver

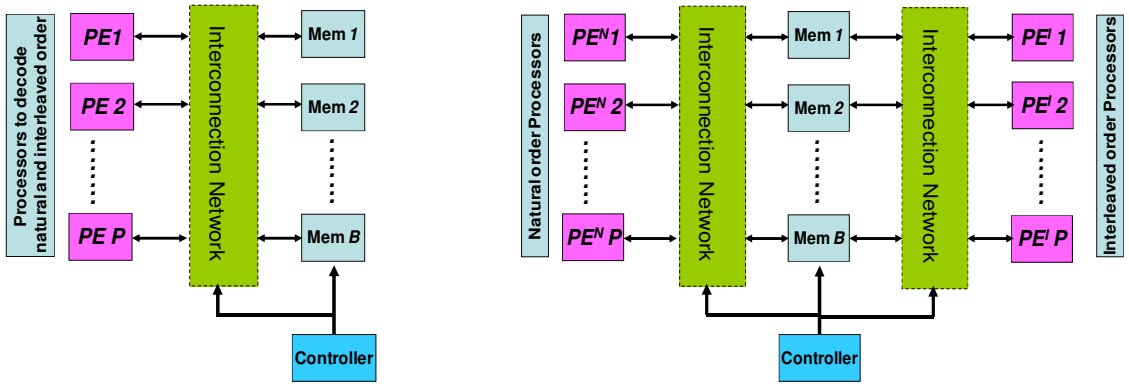
Quadratic Permutation Polynomial (QPP) interleaver used in LTE [LTE08] (discussed in chapter 2, section 2.1.1) is maximum contention-free i.e., for every window size *W* which is a factor of the interleaver length *N*, the interleaver is contention free. This means that for SISO decoder level, this interleaver is mostly conflict free. However, for higher data rate applications when trellis and recursive units parallelism are also included in each SISO, QPP interleaver is not contention-free and requires a router and buffer mechanism to solve memory conflicts. For block size *N*, QPP interleaver is represented by following equation.

$$\Pi(x) = (f_1 x^2 + f_2 x) \bmod N$$

where x and Π(x) represents the original and interleaved address respectively and integers f_1, f_2 are different for different block lengths and can be found in the standard.

6.1. Configurations used in this study

In this section, a study has been presented for implementing turbo decoder for different types of parallelism for QPP interleaver used in LTE. This study is performed in collaboration with ENST Bretagne for developing high speed turbo decoder. This work has been done in collaboration with O. Sanchez-Gonzalez, who is a PhD student under the direction of M. Jezequel. In this study, conflict free memory mapping has not only been found for the parallelisms that we discussed in chapter 1 but also for a scheduling called Replica shuffled decoding. In shuffled decoding, both natural and interleaved orders are decoded at the same time. However, in non-shuffled decoding (as discussed in Chapter 1 section 2.4.4) first natural order and then interleaved order are processed to decode the codeword. The implementation of non-shuffled and shuffled decoding architectures is shown in Figure 5. 20.a & b respectively. In shuffled decoding two sets of processing elements and interconnection networks are used whereas in non-shuffled decoding only one set of processing elements and interconnection network is used to decode the codeword.

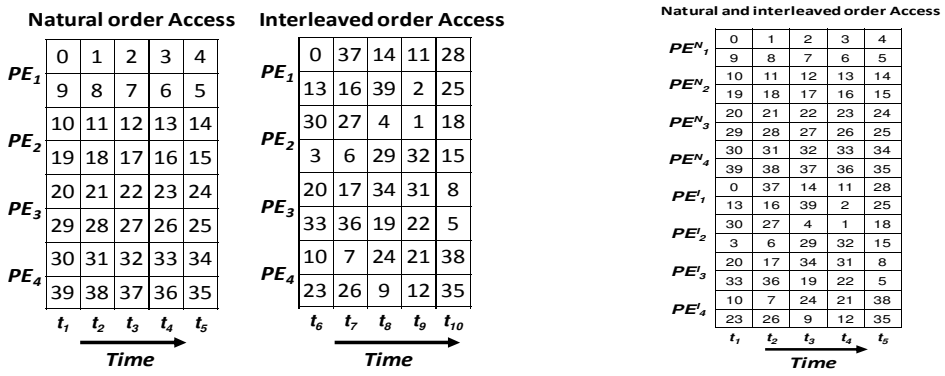


(a) non-shuffled decoding Architecture (b) shuffled decoding Architecture

Figure 5. 20. Decoding Architecture for Turbo Decoders

From this diagram, it is clear that decoding time is reduced to half for shuffled decoding at the cost of doubling the interconnection network and processing elements. However, QPP interleaver is no more conflict free for shuffled decoding and bipartite edge coloring algorithm is used to map the data in different memory banks in order to reduce the cost of interconnection network and increase the throughput of the system. Scheduling of $K = 40$, $P = 4$ for non-shuffled decoding using butterfly scheme and shuffled decoding using replica scheme is shown in Figure 5. 21.a & b respectively.

For non-shuffled decoding, *Single memory mapping* approach is used whereas for shuffled decoding, *Double memory Mapping* approach is used to map the data and design interconnection network.



(a) Scheduling for non-shuffled decoding (b) Scheduling for shuffled decoding

Figure 5. 21. Scheduling for Turbo Decoding

In this study, nine different configurations are studied to explore the design space for turbo decoding for both non-shuffled and shuffled decoding using QPP interleaver. All these configurations are shown in Table 5. 10. In this table, second column shows whether decoding is non-shuffled or shuffled whereas the third column contains the scheme that is used in this configuration. Two addition constraints are added in this study to widen the design space exploration for designing turbo decoders. First constraint is the radix and the second is whether we use internal memory in the decoder or not. or non-shuffled mode, design is explored for radix-2 to radix-16 whereas for shuffled mode only radix-2 and radix-4 is used to design decoder. Use of internal memory reduces the number of accesses to the memory and hence increases the throughput of the system at the cost of increased hardware cost. Number of accessed to the memory is also studied to determine the latency of the system.

Table 5. 10. Different configuration to explore the design space for turbo decoding

	Mode	Scheduling	Radix	Internal Memory
Config. 1	Non-Shuffled	Butterfly	2	YES
Config. 2	Non-Shuffled	Butterfly	4	YES
Config. 3	Non-Shuffled	Butterfly	16	YES
Config. 4	Non-Shuffled	Butterfly	2	No
Config. 5	Non-Shuffled	Butterfly	4	No
Config. 6	Shuffled	Replica	2	No
Config. 7	Shuffled	Replica	2	YES
Config. 8	Shuffled	Replica	4	No
Config. 9	Shuffled	Replica	4	YES

6.2. Experiments and Results

This study is performed using $K = 2048$ (with size of each data is 10 bits) for QPP interleaver used in LTE. However, for non-shuffled mode $P = 32$ and for shuffled mode $P=64$. We suppose that the latency of processing element and network is one cycle in all these configurations. Number of banks in each configuration can be found using the below formula:

$$B = P * 2 * \log_2(R) \qquad \text{where R = radix used in decoding}$$

To show the impact of internal memory, a comparison between configuration 1 and 4 is presented in Table 5. 11. Configuration 1 uses internal memory whereas configuration 4 works without internal memory. Hence, configuration 4 needs $P = 32$ more memory accesses to decode the codeword.

Table 5. 11. Comparison of Configuration 1 and 4 for latency

Configuration	Memory Accesses for Natural order		Memory Accesses for Interleaved order		Total Memory Accesses
	Read	Write	Read	Write	
1	32	32	32	32	128
4	64	32	64	32	192

The complete comparison for hardware cost and latency for all the configurations is given in Table 5. 12. In this table, *NS* stands for non-shuffled and *S* for shuffled, *Rs* means radix-s decoding is performed in this configuration, *mem* means decoder has internal memory and *No.mem* represents that decoder functions without internal memory. Values in *NW* column stands for *input x output* Benes network and *NW cost* represents number of MUX for 10 bit data used in this network. Values in NW controller and Memory controller represents number of control bits required to control network and generating addresses for memory for decoding codeword in one complete iteration.

Table 5. 12. Comparison of all the configurations for hardware cost and latency

	Total Memory Accesses	NW	NW Stages	NW Cost	NW Controller	Memory Controller	Extrinsic Memory
Config. 1 NS_R2_mem	128	64x64	11	704	45056	40960	20480
Config. 2 NS_R4_mem	64	128x128	13	1664	53248	32768	20480
Config. 3 NS_R16_mem	32	256x256	15	3840	61440	24576	20480
Config. 4 NS_R2_No.mem	192	64x64	11	704	67584	61440	20480
Config. 5 NS_R4_No.mem	96	128x128	13	1664	79872	49152	20480
Config. 6 S_R2_No.mem	128	128x128	13	1664	106496	114688	33280
Config. 7 S_R2_mem	128	128x128	13	1664	106496	114688	33280
Config. 8 S_R4_No.mem	64	256x256	15	3840	122880	98304	35840
Config. 9 S_R4_mem	64	256x256	15	3840	122880	98304	35840

VHDL files are generated for memory banks, crossbar network, addressing ROMs (address controller) and network ROM (NW controller) for each configuration. The resultant area for each component of this architecture using targeted technology for each configuration is shown in Table 5. 13.

Table 5. 13. Resultant area for different configurations used in case study

	Total Memory Accesses	NW	NW Stages	NW Cost	NW Control	Memory Controller	Extrinsic Memory	Total
Config. 1 NS_R2_mem	128	64x64	11	61,248	3232,768	2938,880	1469,440	7702,336
Config. 2 NS_R4_mem	64	128x128	13	144,768	3820,544	2351,104	1469,440	7785,856
Config. 3 NS_R16_mem	32	256x256	15	334,080	4408,320	1763,328	1469,440	7975,168
Config. 4 NS_R2_No.mem	192	64x64	11	61,248	4849,152	4408,320	1469,440	10788,160
Config. 5 NS_R4_No.mem	96	128x128	13	144,768	5730,816	3526,656	1469,440	1087,1680
Config. 6 S_R2_No.mem	128	128x128	13	144,768	7641,088	8228,864	2387,840	18402,560
Config. 7 S_R2_mem	128	128x128	13	144,768	7641,088	8228,864	2387,840	18402,560
Config. 8 S_R4_No.mem	64	256x256	15	334,080	8816,640	7053,312	2571,520	18775,552
Config. 9 S_R4_mem	64	256x256	15	334,080	8816,640	7053,312	2571,520	18775,552

In order to ease the analysis, information about latency and architectural cost for all the configurations is shown graphically in Figure 5. 22. From this comparison, it appears that configuration 3, 2, 1 are pareto optimal (i.e. lowest architectural cost, and lower latency, which means higher throughput).

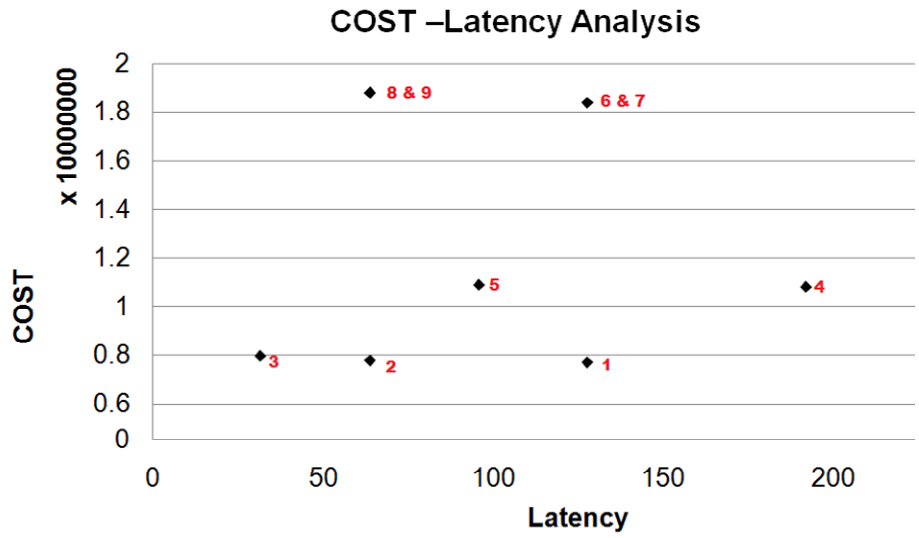


Figure 5. 22. Latency and cost Analysis for all configurations

7. Conclusion

The experiments performed in this chapter can be divided into three categories based on the type of memory mapping approach applied. First two experiments have been used single memory mapping approach, next two experiments have been used double memory mapping approach and the last experiment has been used both single and double memory mapping approaches to find the conflict free memory mapping.

First experiment has been found memory mapping using targeted interconnection network. The generated architecture based on barrel shifter has been occupied less area than the architecture which has been generated using memory mapping without considering targeted interconnection network. This experiment shows the interest of the approach that finds architecture oriented mapping. Second experiment finds memory mapping for every type of turbo decoder parallelism in lesser time than existing approaches. Actually, the gain is up to 70 time faster for longer codewords used in turbo decoding. Moreover, heuristics based approaches fails to find mapping for some parallelism types whereas our algorithm is always able to find one.

Third experiment has successfully able to map Z matrices for structured LDPC codes. Due to small number of Z matrices, gain in computation time from the existing approaches in not very much. However, as the size of data increased, gain in computation time also increases. The advantage of polynomial time algorithm has clearly depicted in forth experiment. Polynomial time algorithm is always able to find mapping for every type of parallelism whereas heuristic fails to find mapping for certain parallelism degree.

Last experiment uses the same algorithm to find single and double memory mapping to design high speed turbo decoding. This experiment compares different configurations to generate cost effective high speed turbo decoder.

CONCLUSION AND FUTURE PERSPECTIVES

Turbo and LDPC due to their excellent error correction capabilities are part of current telecommunication standards. However, implementation of these codes introduces new challenges mainly due to support the high data rate applications. For high data rate applications, more than one processing elements are used to decode the received data. However, single memory proves to be the bottleneck in accessing multiple data concurrently. For achieving high memory bandwidth, the main memory is partitioned into smaller memory banks and multiple data values are accessed in parallel through memory to acquire required throughput. However, implementation issues never ends with this technique since scrambling of data caused by interleaving law results in *Memory Conflict Problem*. The conflict management mechanism used to tackle this problem increases latency of memory accesses and decreases system throughput and is not a desirable solution for high data rate applications.

Different approaches have been proposed in literature to tackle this problem. These approaches can be divided into three broad categories. In first category of approaches, different algorithms have been proposed that designed interleaving law taking into account memory conflict problem. The interleaving law has been designed in such a manner that conflict problem never occurs for parallel implementation of decoder. Other purpose of designing interleaving law is to simplify interconnection and addressing logic of the system. However, these interleaving laws have never always been able to construct codes with good error correction capabilities. In second category of approaches, different flexible interconnections networks with sufficient path diversity and additional storing elements have been proposed to handle memory conflict problem. These networks have been designed to handle any interleaving law. However, these networks have been suffered from large silicon area and latency which make them inefficient for high data rate applications. Third approach has dealt with the idea to allocate data in memory banks in such a manner to avoid memory conflict problem either using particular network or the network that supports all the permutations at the cost of some preprocessing. However, till now no algorithm has been existed that can solve memory mapping problem for both turbo and LDPC codes in polynomial time.

In this thesis, different methods have been proposed to allocate data in different memory banks so that different processing elements can access them concurrently without any conflict. All these methods are based on graph theory and can be divided into two parts. In the first part, mapping problem is modeled as bipartite or tripartite graph based on the access order of data. In the second part, different algorithms are proposed to color the edges of these graphs and to map the data into different memory banks. In this thesis, we present complete path that we followed before it becomes possible to solve mapping problem in polynomial time. In this regard, first two approaches model mapping problem as bipartite graph and then in order to facilitate the coloring of the edges each graph is divided into different sub-graphs. First approach tackle memory conflict problem for turbo codes and uses transportation problem algorithms to partition and color the edges of bipartite graph. The

approach can also be able to find memory mapping that supports particular interconnection network if interleaving law of the application allows it. Second approach solves memory mapping problem for LDPC codes using two different complex algorithms based on path traversal. First algorithm partitions the graph into different sub-graphs using some constrains where as second algorithm color the edges of each sub-graph. In the third approach, bipartite graph modeled in first approach is converted into tripartite graph by dividing each time instance and edge into two. Using an algorithm based on divide and conquer strategy, each tripartite graph is partitioned into different sub-graphs by using a complex algorithm. Afterwards, each sub-graph is colored individually by using simple algorithm to find conflict free memory mapping for both turbo and LDPC codes. In the last approach, further optimization in modeling is performed by converting tripartite graph into bipartite graph on which coloring algorithm based on Euler partition principle is applied to find memory mapping in polynomial time.

To show the interest of the proposed mapping methods, several experiments have been performed using interleaving laws coming from different communication standards. All the experiments have been done by using a software tool we developed. This tool first finds conflict free memory mapping and then generates VHDL files that can be synthesized to design complete architecture i.e. network, memory banks and associated controllers. First experiment has been used to design parallel architecture for bit interleaver used in different standards to tackle burst errors. In this experiment parallel architecture for bit interleaver used in UWB communications system is designed. Algorithm used in this experiment has also been able to find memory mapping that supports targeted interconnection network. In second experiment, turbo interleavers used in different telecommunication standards for enhancing forward error correction capabilities of turbo codes are tackled. Different experiments have been performed to design parallel interleaver architectures used in HSPA Evolution. This interleaver is not conflict free for every type of turbo decoder parallelism. Conflict free memory mapping is found and hardware architecture is proposed for different decoder parallelisms in this experiment. Single memory mapping approach has been used to find conflict free memory mapping in these experiments and both runtime of the algorithm and area of the resultant architecture are compared with the state of the art approaches. In third experiment, partially parallel architecture is designed for structured LDPC codes. Structured LDPC codes are increasingly used in different telecommunication standards. In this experiment, double memory mapping approach is used to solve memory conflict problem and different experiments have been performed to design parallel interleaver architecture for different parallelism and block sizes. In forth experiment, parallel interleaver architecture is designed for non-binary LDPC (NB-LDPC). NB-LDPC is developed to enhance the performance of binary LDPC. However, interleaving law used in NB-LDPC is not conflict free even for serial implementation of NB-LDPC decoder. In this experiment, parallel interleaver architecture is designed for both serial and partially parallel implementation of NB-LDPC decoder. In both of these experiments, double memory mapping approach is used to find conflict free memory mapping. Both runtime and resultant architecture are compared with state of the art solutions. In final experiment, both single and double memory mapping approach is used to design parallel architecture for QPP interleaver used in LTE. The goal of the experiment is to compare different configurations to design high speed LTE decoder. These configurations differ in their modes (shuffled or non-shuffled), schemes (butterfly or replica), radices and whether internal memory inside SISO decoder is used or

not. Hardware cost and latency are calculated for each of these configurations and results are detailed to shown the tradeoff between area and throughput to design LTE decoder in this experiment.

Perspectives

In all resultant architectures we generate, ROM is used to control interconnection network and to generate addresses for different memory banks. This approach may be sufficient to design parallel architecture that supports single codeword or applications. However, to design optimized hardware architecture that supports complete standard or different applications, several enhancements are required.

First, ROM based approach results in important hardware cost and area. To reduce hardware cost, optimizations are required to use as less ROMs as possible to support different applications. This problem is explored in the PhD thesis of Aroua Briki under the direction of E. Martin. In this work, the goal is to find a conflict free memory mapping which both memory addressing and network control sequences are as constant/regular as possible.

Second, in order to reduce the cost of the network, additional constraints can be added in current algorithms. This will allow the designer to define the network he wants to target. Algorithm could also be modified in order to generate an optimized interconnection network, i.e. composed of several basic components such as barrel-shifters, butterflies...

Finally, in order to support several codewords or applications, the current approach requires the architecture to include several ROMs i.e. one set of ROMs per codeword. Further information on that topic cannot currently be disclosed because key concepts are patent pending.

BIBLIOGRAPHY

[AHA] "Primer: Reed-Solomon Error Correction Codes", *AHA Application Note*.

[ASG10] Rizwan Asghar and Dake Liu "Towards Radix-4, Parallel Interleaver Design to Support High-Throughput Turbo Decoding for Re-Configurability" *33rd IEEE SARNOFF Symposium 2010*, Princeton, NJ, USA.

[BAH74] L. R. Bahl, J. Cocke, F. Jelinek and J. L. R. Bahl, J. Cocke, F. Jelinek and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. IT-20, no. 2, pp. 284–287, March 1974.

[BAT04] A. Batra et al., "Design of a multiband OFDM system for realistic UWB channel environments," *IEEE Trans. Microwave Theory Tech*., vol. 52, no. 9, pp. 2123–2138, Sept. 2004.

[BAZ97] M. S. Bazaraa, J. J. Jarvis, "Linear Programming and Network Flows". Chapter 8. *John Wiley and Sons*. 1997.

[BER93] C. Berrou, A. Glavieux and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: turbo-codes" in *Proc. IEEE Int. Conf. on Communications (ICC 93)*, Geneva, Switzerland, pp. 1064–1070, 1993.

[BLA02] A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density paritycheck code decoder," *IEEE J. Solid-State Circuits*, vol. 37, no. 3, pp. 404–412, March 2002.

[BLA05] Blankenship, B. Classon, and V. Deai. "High-throughput turbo decoding techniques for 4G" *In Proc. Int. Conf. 3G Wireless and Beyond*. San Francisco, CA, page 137142, June 2005.

[BRU46] De Bruijn, N.G., "A Combinatorial Problem" *Koninklijke Nederlandse Akademie* v. Wetenschappen 49,758–764, 1946.

[CHA10] C. Chavet, P. Coussy, "A memory Mapping Approach for Parallel Interleaver design with multiples read and write accesses". *In Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)* 2010.

[CHA10a] C. Chavet, P. Coussy, P. Urard and E. Martin, "Static Address Generation Easing: a Design Methodology for Parallel Interleaver Architecture". In *proc. ICASSP 2010*.

[CHE02] J. Chen and M. Fossorier. "Density evolution of two improved bp-based algorithms for LDPC decoding". *IEEE Communication letters*, March 2002.

[COL82] R. Cole, J. Hopcroft, " On edge coloring bipartite graphs", *SIAM Journal on Computing* 11 (1982) 540-546.

[DAV] http://www.ict-davinci-codes.eu/

[DVBS08] *ETSI EN 302-583 V1.1.1, (2008).* "Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for satellite services to handheld devices (SH) below 3 GHz". March. 2008.

[DVB08] *DVB DocumentA122.* "Frame structure channel coding and modulation for the second generation digital terrestrial television broadcasting system (DVB-T2)," 2008.

[FAN06] L. Fanucci, P. Ciao and G. Colavolpe, "VLSI design of a fully-parallel high-throughput decoder for turbo Gallager codes," *IEICE Trans. Fund. Electronics, Communications and Computer Sci*., vol. E89-A, no. 7, pp. 1976–1986, July 2006.

[FOS99] M.P.C Fossorier, M. Mihaljevic, and H. Imai. "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation". *IEEE Transactions on communications*, 47:673–680, May 1999.

[GAB76] H.N. Gabow, "Using Euler partitions to edge color bipartite multigraphs", *International Journal of Computer and Information Sciences 5* (1976) 345-355.

[GIU02] A. Giulietti, L. v. d. Perre, and A. Strum, "Parallel turbo coding interleavers: Avoiding collisions in accesses to storage elements," *Electron.Lett*., vol. 38, no. 5, pp. 232–234, Feb. 2002.

[GNA03] D.Gnaedig, E.Boutillon, M.Jezequel, V.C.Gaudet, and P.G.Gulak, "On multiple slice turbo codes," in *Proc. 3rd Int. Symp. Turbo Codes, Related Topics*, Brest, 2003, pp. 343–346.

[GOL61] M. J. E. Golay, "Complementary series", *IRE Trans. on Info. Theory*, vol. IT-7, pp. 82-87, April 1961.

[GRO03] Jonathan L. Gross, Jay Yellen, "Handbook of Graph Theory". *CRC Press*. 2003.

[HAM50] R.R. W. Hamming, "The Bell System Technical Journal," *Bell Syst. Tech. J*., vol. XXVI, no. 2, pp. 147–160, Apr. 1950.

[HAR06] D. Hartvigsen, "Finding maximum square-free 2-matcing in bipartite graphs". *J. combin. Theory*, Ser. B 96 (2006) 693-705.

[HSP04] 3GPP, "Technical specification group radio access network;multiplexing and channel coding (FDD)" *(25.212 V5.9.0).*June 2004.

[HSP08] 3GPP, "Technical Specification Group Radio Access Network; Multiplexing and Channel Coding (FDD)" *Tech. Spec. 25.212 V8.4.0* Dec. 2008.

[JOH10] Sarah J. Johnson, "Iterative Error Correction Turbo, Low-Density Parity-Check and Repeat–Accumulate Codes" *Cambridge University Press* 2010

[KAI05] Kaiser, Ed. "UWB Communications Systems: A Comprehensive Overview" *EURASIP Series on Signal Processing and Communications, Hindawi Publishing*, New York, 2005.

[KEY01] P. Keyngnaert, B. Demoen, B. De Sutter, B. De Sus, and K. De Bosschere. "Conflict Graph Based Allocation of Static Objects to Memory Banks" *Informal proceedings of the First workshop on Semantic, Program Analysis, and Computing Environments*, pages 131–142, September 2001.

[KIE03] Kienle, F., Thul, M. J., and When, N., 2003. "Implementation Issues of Scalable LDPC-Decoders". *in Proceeding of 3rd International Symposium on Turbo Codes and Related Topics, Brest, France,* 291-294.

[KON16] D. König, "Graphok és alkalmazásuk a determinánsok és a halmazok elméletére". *Mathematikai és Természettudományi Értzsitö* 34 (1916) 101-119.

[KOZ92] D.C.Kozen, "The Design and Analysis of Algorithms", *Springer Verlag, USA*, 1992.

[KWA02] J. Kwak and K. Lee, "Design of dividable interleaver for parallel decoding in turbo codes," *Electron. Lett*., vol. 38, no. 22, pp. 1362–1364, 2002.

[LEE08] J.-Y. Lee and H.-J. Ryu, "A 1-Gb/s flexible LDPC decoder supporting multiple code rates and block lengths," *IEEE Trans. Consumer Electronics*, vol. 54, no. 2, pp. 417–424, May 2008.

[LIN04] Shu Lin and Daniel J. Costello, Jr., "Error control Coding" *Pearson Education*, Inc 2004

[LIN10] Jing-ling, "Parallel Interleavers Through Optimized Memory Address Remapping" *IEEE Trans. VLSI Systems* vol. 18, no.6, pp.978-987, June. 2010.

[LTE08] "Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access; Multiplexing and Channel Coding (Release 8)", *3GPP Std. TS 36.212*, Dec. 2008.

[MAN03] M. Mansour and N. Shanbhag, "High-throughput LDPC decoders," *IEEE Trans. VLSI Syst*., vol. 11, no. 6, pp. 976–996, Dec 2003.

[MAS07] G. Masera, F. Quaglio and F. Vacca, "Implementation of a flexible LDPC decoder," *IEEE Trans. Circuits and Systems – II: Express Briefs*, vol. 54, no. 6, pp. 542–546, June 2007

[MOU07] H. Moussa, O. Muller, A. Baghdadi, and M. Jezequel, "Butterfly and Benes-based on-chip communication networks for multiprocessor turbo decoding," in *Proc. of the conference on Design, Automation and Test in Europe*, pp. 654-659, April 2007.

[MOU08] H. Moussa, A. Baghdadi, and M. Jezequel, "Binary de Bruijn interconnection network for a flexible LDPC/turbo decoder," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2008, pp. 97–100.

[MOU08] H.Moussa, A.Baghdadi, M.Jezequel."Binary de Bruijn on-chip netwok for a flexible multiprocessor LDPC decoder".*45th ACM/IEEE DAC, p.429-434*, 2008.

[NAG04] V. Nagarajan, N. Jayakumar, S. Khatri and G. Milenkovic, "High-throughput VLSI implementations of iterative decoders and related code construction problems," in Proc. *IEEE Global Telecommunications Conf. (GLOBECOM 2004)*, vol. 1, pp. 361–365, 2004.

[NEE05] C. Neeb, M. Thul, and N. Wehn, "Network-on-chip-centric approach to interleaving in high throughput channel decoders," in *Proc. IEEE International Symposium on Circuits and Systems (ISCAS),* May 2005, pp. 1766 – 1769 Vol. 2.

[ORE67] Ore, "The Four Color Problem" *Academic Press*, New York, 1967.

[PEA88] J.Pearl, "Probabilistic Reasoning in Intelligent Systems: Networks of Plausible reference", *Morgan Kaufmann*, 1988.

[QUA06] F.Quaglio, F.Vacca, C.Castellano, A.Tarable, M.G.Asera. "Interconnection Framework for High-Throughput, Flexible LDPC Decoders". *In proceeding Design Automation and Test in Europe Conference and Exhibition*, 2006.

[SAN10] Awais Sani, Philippe Coussy, Cyrille Chavet, Eric Martin: Design of parallel LDPC interleaver architecture: A bipartite edge coloring approach, *ICECS 2010*, 466-469.

[SAN11] Awais Sani, Philippe Coussy, Cyrille Chavet, Eric Martin "An approach based on edge coloring of tripartite graph for designing parallel LDPC interleaver architecture" *ISCAS 2011*, 1720-1723.

[SAN11a] Awais Sani, Philippe Coussy, Cyrille Chavet, Eric Martin "A methodology based on Transportation problem modeling for designing parallel interleaver architectures" *ICASSP 2011*, 1613-1616.

[SCH98] A.Schrijver, “Bipartite edge coloring in O(Δ*m*) time”, *SIAM Journal on Computing*, vol.28, pp.841–846, 1998.

[SHA48] C.E. Shannon, A mathematical theoryof communication, *Bell System Tech*. J. 27 (July and October 1948) 379–423, 623–656.

[SIR08] W. P. Siriwongpairat, K. J. Ray Liu “Ultra-Wideband Communications Systems” *Joh Wiley & Sons, Ins., Publication,* 2008.

[SUN05] J. Sun and O. Y. Takeshita, “Interleavers for turbo codes using permutation polynomials over integer rings ,” *IEEE Trans. Inf. Theory*, vol. 51, no. 1, pp. 101–119, Jan. 2005.

[TAK06] O. Y. Takeshita, “On maximum contention-free interleavers and permutation polynomials over integer rings,” *IEEE Trans. Inf. Theory*, vol. 52, no. 3, pp. 1249–1253, Mar. 2006.

[TAR04] A. Tarable, S. Benedetto, and G. Montorsi, “Mapping interleaving laws to parallel turbo and LDPC decoder architectures”, *IEEE Trans.Inf.Theory*, vol. 50, no.9, pp.2002-2009, Sep. 2004.

[TAR05] A. Tarable and S. Benedetto, “Further results on mapping functions,” *in Proc. Inform. Th. Workshop 2005 (ITW2005)*, pp. 221-225.

[THE05] Theocharides, T., Link, G., Vijaykrishnan, N., and Irwin, M. J. “Implementing LDPC Decoding on a Network-on-Chip”. I*n Proc. of the Int. Conf.* VLSI Design, 2005, 134-137.

[THU02] M. Thul, N. Wehn, and L. Rao, “Enabling high-speed turbo decoding through concurrent interleaving,” in *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 1, 2002, pp. 897–900.

[THU02a] M. I. Thul, F. Gilbert. and N. Wehn. “Optimized Concurrent Interleaving for High-speed Turbo-Decoding”. In *Proc. 9rh IEEE International Conference on Electronics, Circuits and Systems - ICECS* 2002, Dubrovnik, Croatia, Sept. 2002.

[THU03] M. Thul, F. Gilbert, and N. Wehn, “Concurrent interleaving architectures for high-throughput channel coding,” in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP),* 2003, pp. 613–616 vol.2.

[WIF08] *IEEE P802.11n/D5.02, Part 11*. “ Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Enhancements for Higher Throughput” , July 2008.

[WIM06] *IEEE P802.16e, Part 16*. “ Air Interface for Fixed and Mobile Broadband Wireless Access Systems,” Amendment 2: Physical and Medium Access Control Layers for Combined Fixed and Mobile Operation in Licensed Bands, and Corrigendum 1, Feb. 2006.

[WON10] C.-C.Wong and H.-C. Chang, “Reconfigurable turbo decoder with parallel architecture for 3GPP LTE system,” *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 57, no. 7, pp. 566–570, Jul. 2010.

[WOO00] J. P. Woodard and L. Hanzo, “Comparative study of turbo decoding techniques: An overview,” *IEEE Trans. Veh. Tech*., vol. 49, no. 6, pp. 2208–2233, Nov. 2000.

[ZHA01] T. Zhang and K. K. Parhi. “Joint code and decoder design for implementation-oriented (3;k)-regular LDPC codes”. *in Proc. Asilomar Conference on Signals, Systems and Computers*, 2:1232_1236, Nov.2001.

[ZHA04] Y. Zhang and K. Parhi, “Parallel turbo decoding,” *in Proceedings of the ISCAS ’04.*, vol. 2, 23-26 May 2004, pp. II–509–12Vol.2.

[ZHO07] L. Zhou, C.Wakayama and C.-J. R. Shi, “CASCADE: a standard super-cell design methodology with congestion-driven placement for three-dimensional interconnect-heavy very large scale integrated circuits,” *IEEE Trans. Computer-Aided Design*, no. 7, July 2007.

PERSONNAL BIBLIOGRAPHY

International conference

[SAN10] Awais Sani, Philippe Coussy, Cyrille Chavet, Eric Martin: “Design of parallel LDPC interleaver architecture: A bipartite edge coloring approach”, *ICECS 2010*, 466-469.

[SAN11] Awais Sani, Philippe Coussy, Cyrille Chavet, Eric Martin “An approach based on edge coloring of tripartite graph for designing parallel LDPC interleaver architecture” *ISCAS 2011*, 1720-1723.

[SAN11a] Awais Sani, Philippe Coussy, Cyrille Chavet, Eric Martin ”A methodology based on Transportation problem modeling for designing parallel interleaver architectures” *ICASSP 2011*, 1613-1616.

National conference

[SAN10a] Awais Sani, Philippe Coussy, Cyrille Chavet, Eric Martin, ”A Bipartite Edge Coloring Approach for designing Parallel Interleaver architecture", In Colloque National du GDR SoC-SiP, Paris, France, june 2010.

[SAN11b] Awais Sani, Philippe Coussy, Cyrille Chavet, Eric Martin “Designing Parallel Interleaver architecture through Tripartite Edge Coloring Approach ” In Colloque National du GDR SoC-SiP, Lyon, France, june 2011.

ANNEXURE

1. Pedagogical Example to solve Transportation Problem

We present a simple example in this section to solve transportation problem. Consider a matrix model of transportation problem with 3 sources and 3 destinations with the cost to transport a unit item from source to destination is placed in the corresponding cell as shown in Figure A. 1.a.

Objective Function

MIN ($90x_{11} + 100x_{12} + 150x_{13} + 120x_{21} + 140x_{22} + 100x_{23} + 120x_{31} + 80x_{32} + 80x_{33}$)

Constraints

$x_{11} + x_{12} + x_{13} \geq 20$	Supply of Producer I_1
$x_{21} + x_{22} + x_{23} \geq 15$	Supply of Producer I_2
$x_{31} + x_{32} + x_{33} \geq 10$	Supply of Producer I_3
$x_{11} + x_{21} + x_{31} \geq 5$	Demand of Consumer J_1
$x_{12} + x_{22} + x_{23} \geq 20$	Demand of Consumer J_2
$x_{13} + x_{23} + x_{33} \geq 20$	Demand of Consumer J_3

Two more constraints are needed to be fulfilled in order to solve the transportation problem:

First constraint: Total Supply must be greater than equal to total Demand i.e.,

$$a_1 + a_2 + a_3 \geq z_1 + z_2 + z_3$$

For our example, $20 + 15 + 10 = 5 + 20 + 20 = 45$

Second Constrain:

Number of constraints must be equal to the number of rows + number of columns

For our example: 3 (rows) + 3 (columns) = 6 (number of constraints)

To solve the transportation problem, first we find initial basic feasible solution using simple methods and then check using duality to determine whether the basic solution is optimal or not. If the solution is not optimal then further iterations are performed to find the optimal solution.

To find basic feasible solution (i.e., non optimal solution), there are number of methods but in our example we use *northwest corner* method that is easy to use and requires simple calculation to get results. The method starts by making an allocation to the northwest corner or upper left cell of the matrix i.e., M_{11}. The allocated items are either all the supply for the row or all the demand for the column, connected with that cell, whichever is *smaller*. From an example, it is clear that demand of J_1 is smaller than supply of I_1, so the method allocated 5 to M_{11}. This completes the demand of J_1 and eliminates the column J_1 from further consideration in this method as shown in Figure A. 1.b.

Producer \ Consumer	J_1	J_2	J_3	Supply
I_1	90	100	150	20
I_2	120	140	100	15
I_3	120	80	80	10
Demand	5	20	20	45

(a) Transportation Problem Matrix

Producer \ Consumer	J_1	J_2	J_3	Supply
I_1	90 5	100	150	20
I_2	120	140	100	15
I_3	120	80	80	10
Demand	5	20	20	45

(b) Items Allocation

Figure A. 1. Northwest Corner Method to solve Transportation Problem

The next step is to move horizontally since, vertically demand of J_1 is completed. Since the remaining supply of I_1 (15) is less than demand of J_2 (20), so algorithm assigns 15 to M_{12} using criteria explained previously. This completes the supply of I_1 as shown in Figure A. 2.a. The method continues until supply of all the producers and demand of all the consumers are fulfilled. The complete allocation of items is shown in Figure A. 2.b. Initial basic feasible solution for this problem is:

$$90(5) + 100(15) + 140(5) + 100(10) + 80(10) = 4450.$$

Producer \ Consumer	J_1	J_2	J_3	Supply
I_1	90 5	100 15	150	20
I_2	120	140	100	15
I_3	120	80	80	10
Demand	5	20	20	45

(a) Items Allocation

Producer \ Consumer	J_1	J_2	J_3	Supply
I_1	90 5	100 15	150	20
I_2	120	140 5	100 10	15
I_3	120	80	80 10	10
Demand	5	20	20	45

(b) Complete Allocation

Figure A. 2. Northwest Corner Method to solve Transportation Problem

The next step is to apply optimally test to determine whether the basic solution is optimal or not. To perform test, duality theory is used and two variables u_i and v_j are introduced where

u_i = dual variable corresponding to row i

v_j = dual variable corresponding to row j

From duality theory:

$$o_{ij} = u_i + v_j \qquad (1)$$

Values of u_i and v_j can be calculated from initial matrix using equation 1 as follows:

$o_{11} = u_1 + v_1 = 90$

$o_{12} = u_1 + v_2 = 100$

$o_{22} = u_2 + v_2 = 140$

$o_{23} = u_2 + v_3 = 100$

$o_{33} = u_3 + v_3 = 80$

Since there are (U + V) unknowns for (U + V – 1) equations, where U and V are number of dual variable for row and columns respectively, arbitrary value can be assigned to one of the unknown. A common approach is to choose the row with largest number of allocations. I_1 and I_2 both have two allocations each so we arbitrary choose row 1 and assign $u_1 = 0$. Now other values can be calculate during substitution as follows:

$o_{11} = u_1 + v_1 = 90$	$0 + v_1 = 90$	$v_1 = 90$
$o_{12} = u_1 + v_2 = 100$	$0 + v_2 = 100$	$v_2 = 100$
$o_{22} = u_2 + v_2 = 140$	$u_2 + 100 = 140$	$u_2 = 40$
$o_{23} = u_2 + v_3 = 100$	$40 + v_3 = 100$	$v_3 = 60$
$o_{33} = u_3 + v_3 = 80$	$u_3 + 60 = 80$	$u_3 = 20$

Two columns corresponding to u_i and v_j are added in the allocated matrix to facilitate the future calculations as shown in Figure A. 3.a.

The cells which have allocations in the matrix are called *basic cells* otherwise they are called *nonbasic cells*. To recognize whether the solution is optimal or not, the equation 2 must be fulfilled for all the nonbasic cells.

$$o_{ij} - u_i - v_j \geq 0 \qquad (2)$$

If the equation is false for any of the nonbasic cell then the solution is not optimal.

For cell M_{13}: $130 - 0 - 60 \geq 0$ is true
For cell M_{21}: $100 - 40 - 90 \geq 0$ is false
For cell M_{31}: $100 - 20 - 90 \geq 0$ is false
For cell M_{32}: $80 - 20 - 100 \geq 0$ is false

Since equation 2 is not fulfilled so the initial solution is not optimal. The cells with false condition or where the resultant value is negative are the locations where we need to assign values in order to reduce our shipping costs. These negative values are put in the matrix as shown in Figure A. 3.b..

For further iterations, we prepare *closed loop path* in which certain cells exchange their status. A current basic cell becomes nonbasic and empty nonbasic cell turns into basic. The new basic or nonempty cell is called *entering cell* and cell with which it exchanges is called the *exiting cell*. To construct closed loop path, some rules need to be followed:

First rule is there cannot be more than one increasing or decreasing cell in any row or column.

Second rule is except for the entering cell all changes occur in basic cells.

Producer \ Consumer	J_1	J_2	J_3	Supply	u_i
I_1	90 5	10 15	150	20	0
I_2	120	140 5	100 10	15	40
I_3	120	80	80 10	10	20
Demand	5	20	20	45	
v_j	90	100	60		

(a) duality

Producer \ Consumer	J_1	J_2	J_3	Supply	u_i
I_1	90 5	10 15	150	20	0
I_2	120 -30	140 5	100 10	15	40
I_3	120 -10	80 -40	80 10	10	20
Demand	5	20	20	45	
v_j	90	100	60		

(b) items Allocation

Figure A. 3. Procedure to check the optimal solution for Transportation Problem

For minimization problem, the method chooses the cell with largest negative number as the entering cell to construct closed loop. In our example, M_{32} has the largest negative value so the method selects this cell as entering cell and place "+" in this cell to indicate that the method want to increase the value in this cell as much as possible. Since all the other cells in the loop should be basic so the method selects M_{22} as the next cell to be included in the loop and place "-" in this cell to keep everything in equilibrium and indicate that the method reduces the assigned value from this cell. Complete closed loop is shown in Figure A. 4.a.

Few things need to be considered while constructing a loop. First of all, the procedure must start and end at the new entering cell in order to construct a complete circuit. A *junction* is made at each cell where "+" or "-" is entered to indicate that at these cells we can either move from vertical to horizontal or from horizontal to vertical. Sometime this results in stepping over basic or nonbasic cells to construct closed loop. Secondly, If the procedure reaches at a cell from where it cannot turn (because all other cells in its column or row are nonbasic) then the procedure must backtrack to the last *junction* cell to find a basic cell.

After the construction of closed loop, the next step is the shifting of values in the junction cells. The smallest quantity in the losing (-) cell is the amount that is shifted in the current loop. In our example the smallest quantity is 5 in M_{22}, so the procedure assigns 5 to M_{32}, increases the assignment by 5 in M_{23} and reduces the assignment by 5 in cells M_{22} and M_{33} as shown in Figure A. 4.b.

Producer \ Consumer	J_1	J_2	J_3	Supply	u_i
I_1	90 5	10 15	150	20	0
I_2	120 -30	140 5 (-)	100 10 (+)	15	40
I_3	120 -10	80 -40 (+)	80 10 (-)	10	20
Demand	5	20	20	45	
v_j	90	100	60		

(a) closed Loop Construction

Producer \ Consumer	J_1	J_2	J_3	Supply	u_i
I_1	90 5	10 15	150	20	
I_2	120 -30	140 (-)	100 15 (+)	15	
I_3	120 -10	80 5 -40 (+)	80 5 (-)	10	
Demand	5	20	20	45	
v_j					

(b) Exchange of Values

Figure A. 4. Procedure to obtain optimal Solution

The new solution for this problem is:

$$90(5) + 100(15) + 100(15) + 80(5) + 80(5) = 4250$$

This means that the procedure improves the solution by 200.

Again the procedure determines whether the new solution is optimal as done previously using equations 1 & 2.

From Equation 1:

$u_1 = 0$

$o_{11} = u_1 + v_1 = 90$	$0 + v_1 = 90$	$v_1 = 90$
$o_{12} = u_1 + v_2 = 100$	$0 + v_2 = 100$	$v_2 = 100$
$o_{32} = u_3 + v_2 = 80$	$u_3 + 100 = 80$	$u_3 = -20$
$o_{33} = u_3 + v_3 = 80$	$-20 + v_3 = 80$	$v_3 = 100$
$o_{23} = u_2 + v_3 = 100$	$u_2 + 100 = 100$	$u_2 = 0$

The new values for u_i and v_j are shown in Figure A. 5.a.

From Equation 2:

For cell M_{13}: 130 - 0 - 100 ≥ 0 is true

For cell M_{21}: 100 - 0 - 90 ≥ 0 is true

For cell M_{22}: 140 - 0 - 100 ≥ 0 is true

For cell M_{31}: 80 – (-20) - 90 ≥ 0 is true

Since all statements are true, the procedure concludes that 4250 is the lowest cost and optimal solution for our example and is shown in .Figure A. 5.b.

Producer \ Consumer	J_1	J_2	J_3	Supply	u_i
I_1	90 5	10 15	150	20	0
I_2	120	140	100 15	15	0
I_3	120	80 5	80 5	10	-20
Demand	5	20	20	45	
v_j	90	100	100		

(a) duality

Supply \ Demand	J_1	J_2	J_3	Supply
I_1	90 5	10 15	150	20
I_2	120	140	100 15	15
I_3	120	80 5	80 5	10
Demand	5	20	20	45

(b) Optimal Solution

Figure A. 5. Procedure to check the optimal solution for Transportation Problem

Algorithm tests after each solution whether it is optimal or not. If the solution is optimal then algorithm terminates otherwise algorithm continues to search the optimal solution.

2. Memory Mapping for Non-Binary LDPC

Data in each row need to be accessed in parallel.

	R	W		R	W		R	W		R	W		R	W		R	W		R	W		R	W		R	W		R	W		R	W
2	bo	b_6	3	b_7	b_3	149	b_1	b_{11}	122	b_4	b_{10}	9	b_2	b_9	157	b_8	bo	48	b_5	b_8	49	b_6	b_5	146	b_3	b_7	1	b_9	b_2	88	b_{10}	b_4	153	b_{11}	b_1
1	b_3	b_8	11	b_6	b_1	139	b_2	b_{10}	123	b_{11}	b_3	86	b_4	b_7	179	bo	b_{11}	19	b_9	bo	2	b_7	b_4	14	b_1	b_9	94	b_{10}	b_5	16	b_5	b_6	177	b_8	b_2
62	b_5	b_9	63	b_{10}	b_3	159	b_1	b_8	84	b_o	b_{10}	126	b_7	b_1	182	b_2	b_{11}	53	b_3	b_7	54	b_{11}	b_4	112	b_6	b_5	9	b_9	b_2	137	b_4	b_6	172	b_8	b_o
23	b_2	b_{10}	24	b_1	b_8	67	b_5	b_{11}	114	b_3	b_7	178	b_7	b_3	158	b_o	b_9	42	b_{10}	b_o	43	b_9	b_2	79	b_8	b_4	154	b_4	b_6	129	b_6	b_5	151	b_{11}	b_1
59	b_o	b_9	6	b_9	b_o	156	b_2	b_7	133	b_4	b_6	73	b_1	b_{10}	124	b_3	b_{11}	51	b_5	b_8	52	b_6	b_4	113	b_7	b_2	183	b_{11}	b_5	148	b_8	b_1	89	b_{10}	b_3
29	b_o	b_{10}	3	b_{10}	b_o	97	b_4	b_7	11	b_7	b_4	191	b_1	b_9	141	b_3	b_{11}	47	b_5	b_6	48	b_8	b_5	85	b_6	b_2	72	b_9	b_3	177	b_2	b_8	147	b_{11}	b_1
33	b_1	b_6	34	b_{10}	b_o	142	b_5	b_{11}	7	b_3	b_9	172	b_o	b_8	88	b_4	b_{10}	46	b_2	b_7	47	b_6	b_5	81	b_7	b_3	164	b_9	b_1	166	b_8	b_4	17	b_{11}	b_2
26	b_6	b_5	27	b_3	b_7	13	b_2	b_9	119	b_o	b_{10}	8	b_5	b_{11}	148	b_1	b_8	7	b_4	b_6	8	b_8	b_4	84	b_{10}	b_o	146	b_7	b_3	13	b_{11}	b_2	168	b_9	b_1
25	b_2	b_9	26	b_5	b_6	115	b_4	b_7	181	b_7	b_3	153	b_1	b_{11}	66	b_8	b_2	58	b_o	b_8	59	b_9	b_o	184	b_6	b_4	142	b_{11}	b_5	89	b_3	b_{10}	95	b_{10}	b_1
16	b_o	b_{11}	17	b_6	b_2	83	b_5	b_6	127	b_1	b_9	128	b_4	b_{10}	143	b_2	b_8	28	b_7	b_5	29	b_{10}	b_o	7	b_9	b_3	159	b_8	b_1	111	b_3	b_7	17	b_{11}	b_4
17	b_2	b_6	18	b_{11}	b_3	155	b_1	b_{11}	19	b_8	b_5	152	b_5	b_{10}	75	b_3	b_8	5	b_4	b_7	6	b_o	b_9	167	b_6	b_1	13	b_9	b_2	118	b_{10}	b_o	86	b_7	b_4
63	b_3	b_{10}	64	b_o	b_{11}	65	b_1	b_6	16	b_7	b_3	176	b_2	b_7	131	b_5	b_8	6	b_9	b_o	7	b_6	b_4	11	b_8	b_2	155	b_{11}	b_1	93	b_4	b_9	173	b_{10}	b_5
36	b_1	b_{10}	37	b_3	b_9	78	b_{11}	b_4	184	b_4	b_6	157	b_o	b_8	13	b_2	b_{11}	1	b_8	b_5	2	b_6	b_o	136	b_9	b_1	139	b_{10}	b_2	92	b_5	b_7	111	b_7	b_3
21	b_o	b_{11}	22	b_{10}	b_o	132	b_4	b_8	98	b_7	b_4	171	b_2	b_{10}	126	b_1	b_7	5	b_5	b_6	51	b_8	b_5	18	b_9	b_2	71	b_3	b_9	141	b_{11}	b_3	175	b_6	b_1
15	b_o	b_9	16	b_{11}	b_o	181	b_3	b_7	68	b_1	b_{10}	17	b_2	b_{11}	144	b_6	b_4	44	b_7	b_3	45	b_9	b_1	87	b_4	b_8	112	b_5	b_6	131	b_8	b_5	186	b_{10}	b_2
11	b_1	b_8	12	b_o	b_9	116	b_5	b_{11}	96	b_7	b_4	14	b_4	b_8	178	b_3	b_7	35	b_6	b_3	36	b_{10}	b_1	12	b_2	b_{10}	174	b_{11}	b_o	77	b_9	b_5	15	b_8	b_2
18	b_3	b_{11}	19	b_o	b_9	96	b_4	b_7	156	b_7	b_2	175	b_1	b_6	15	b_5	b_8	61	b_6	b_3	62	b_9	b_5	119	b_{10}	b_o	149	b_{11}	b_1	18	b_8	b_4	69	b_2	b_{10}
9	b_1	b_8	1	b_8	b_3	174	b_o	b_{11}	83	b_6	b_5	99	b_3	b_{10}	129	b_5	b_6	2	b_4	b_7	21	b_{11}	b_o	68	b_{10}	b_1	97	b_7	b_4	134	b_9	b_2	187	b_2	b_9
38	b_o	b_{10}	39	b_6	b_o	1 9	b_5	b_8	85	b_2	b_6	17	b_4	b_{11}	145	b_7	b_3	56	b_8	b_4	57	b_3	b_7	164	b_1	b_9	116	b_{11}	b_5	69	b_{10}	b_2	191	b_9	b_1
34	b_o	b_{10}	35	b_3	b_6	98	b_4	b_7	19	b_7	b_1	82	b_2	b_9	138	b_5	b_8	37	b_9	b_3	38	b_{10}	b_o	154	b_6	b_4	121	b_8	b_2	165	b_1	b_{11}	8	b_{11}	b_5
4	b_o	b_{10}	41	b_6	b_5	16	b_3	b_7	169	b_9	b_2	161	b_1	b_6	77	b_5	b_9	54	b_4	b_{11}	55	b_7	b_3	121	b_2	b_8	132	b_8	b_4	179	b_{11}	b_o	73	b_{10}	b_1
31	b_o	b_6	32	b_8	b_3	125	b_1	b_9	78	b_4	b_{11}	188	b_3	b_8	135	b_5	b_7	14	b_{11}	b_1	15	b_9	b_o	133	b_6	b_4	12	b_{10}	b_2	186	b_2	b_{10}	92	b_7	b_5
3	b_3	b_7	4	b_o	b_{10}	183	b_5	b_{11}	91	b_6	b_o	138	b_8	b_5	128	b_{10}	b_4	45	b_1	b_9	46	b_7	b_2	71	b_9	b_3	11	b_2	b_8	192	b_4	b_6	165	b_{11}	b_1
12	b_9	b_o	13	b_1	b_{11}	91	b_o	b_6	115	b_7	b_4	173	b_5	b_{10}	161	b_6	b_1	55	b_3	b_7	56	b_4	b_8	1	b_2	b_9	162	b_{10}	b_5	143	b_8	b_2	76	b_{11}	b_3
41	b_5	b_6	42	b_o	b_{10}	117	b_{11}	b_5	81	b_3	b_7	168	b_1	b_9	163	b_2	b_{11}	8	b_4	b_8	9	b_8	b_1	114	b_7	b_3	74	b_{10}	b_o	187	b_9	b_2	137	b_6	b_4
32	b_3	b_8	33	b_6	b_1	169	b_2	b_9	67	b_{11}	b_5	144	b_4	b_6	118	b_o	b_{10}	57	b_7	b_3	58	b_8	b_o	12	b_5	b_7	171	b_{10}	b_2	151	b_1	b_{11}	93	b_9	b_4
39	b_o	b_6	4	b_{10}	b_o	123	b_3	b_{11}	134	b_2	b_9	95	b_1	b_{10}	18	b_4	b_8	27	b_7	b_3	28	b_5	b_7	127	b_9	b_1	87	b_8	b_4	163	b_{11}	b_2	16	b_6	b_5
13	b_{11}	b_1	14	b_1	b_{11}	74	b_o	b_{10}	18	b_2	b_9	145	b_3	b_7	188	b_8	b_3	49	b_5	b_6	5	b_6	b_5	122	b_{10}	b_4	79	b_4	b_8	158	b_9	b_o	176	b_7	b_2
3	b_o	b_{10}	31	b_6	b_o	19	b_1	b_7	117	b_5	b_{11}	66	b_2	b_8	152	b_{10}	b_5	52	b_4	b_6	53	b_7	b_3	72	b_3	b_9	125	b_9	b_1	182	b_{11}	b_2	14	b_8	b_4
22	b_o	b_{10}	23	b_{10}	b_2	189	b_4	b_6	136	b_1	b_9	75	b_8	b_3	113	b_2	b_7	64	b_{11}	b_o	1	b_5	b_8	135	b_7	b_5	14	b_9	b_1	76	b_3	b_{11}	192	b_6	b_4
43	b_2	b_9	44	b_3	b_7	162	b_5	b_{10}	167	b_1	b_6	15	b_8	b_5	82	b_9	b_2	4	b_{10}	b_o	5	b_7	b_4	185	b_o	b_{11}	65	b_6	b_1	124	b_{11}	b_3	166	b_4	b_8
6	b_o	b_9	61	b_3	b_6	11	b_4	b_7	189	b_6	b_4	147	b_1	b_{11}	99	b_{10}	b_3	24	b_8	b_1	25	b_9	b_2	94	b_5	b_{10}	185	b_{11}	b_o	15	b_2	b_8	12	b_7	b_5

Memory Mapping for K = 192 & P = 12

	R	W		R	W		R	W		R	W		R	W		R	W
2	bo	b_3	3	b_3	bo	149	b_1	b_4	122	b_4	b_1	90	b_2	b_5	157	b_5	b_2
48	bo	b_3	49	b_3	bo	146	b_2	b_5	100	b_5	b_2	88	b_1	b_4	153	b_4	b_1
10	bo	b_3	11	b_3	bo	139	b_2	b_5	123	b_5	b_2	86	b_1	b_4	179	b_4	b_1
19	bo	b_3	20	b_3	bo	104	b_2	b_5	94	b_5	b_2	160	b_1	b_4	177	b_4	b_1
62	bo	b_3	63	b_3	bo	159	b_2	b_5	84	b_4	b_1	126	b_1	b_4	182	b_5	b_2
53	bo	b_3	54	b_3	bo	112	b_2	b_5	90	b_5	b_2	137	b_1	b_4	172	b_4	b_1
23	bo	b_3	24	b_3	bo	67	b_2	b_5	114	b_4	b_1	178	b_1	b_4	158	b_5	b_2
42	bo	b_3	43	b_3	bo	79	b_2	b_5	154	b_4	b_1	129	b_1	b_4	151	b_5	b_2
59	bo	b_3	60	b_3	bo	156	b_2	b_5	133	b_5	b_2	73	b_1	b_4	124	b_4	b_1
51	bo	b_3	52	b_3	bo	113	b_1	b_4	183	b_4	b_1	148	b_2	b_5	89	b_5	b_2
29	bo	b_3	30	b_3	bo	97	b_1	b_4	110	b_4	b_1	191	b_2	b_5	141	b_5	b_2
47	bo	b_3	48	b_3	bo	85	b_2	b_5	72	b_4	b_1	177	b_1	b_4	147	b_5	b_2
33	bo	b_3	34	b_3	bo	142	b_2	b_5	70	b_5	b_2	172	b_1	b_4	88	b_4	b_1
46	bo	b_3	47	b_3	bo	81	b_2	b_5	164	b_4	b_1	166	b_1	b_4	107	b_5	b_2
26	bo	b_3	27	b_3	bo	130	b_2	b_5	119	b_4	b_1	80	b_1	b_4	148	b_5	b_2
7	bo	b_3	8	b_3	bo	84	b_1	b_4	146	b_5	b_2	103	b_2	b_5	168	b_4	b_1
25	bo	b_3	26	b_3	bo	115	b_2	b_5	181	b_4	b_1	153	b_1	b_4	66	b_5	b_2
58	bo	b_3	59	b_3	bo	184	b_1	b_4	142	b_5	b_2	89	b_2	b_5	95	b_4	b_1
16	bo	b_3	17	b_3	bo	83	b_2	b_5	127	b_5	b_2	128	b_1	b_4	143	b_4	b_1
28	bo	b_3	29	b_3	bo	70	b_2	b_5	159	b_5	b_2	111	b_1	b_4	170	b_4	b_1
17	bo	b_3	18	b_3	bo	155	b_2	b_5	109	b_5	b_2	152	b_1	b_4	75	b_4	b_1
5	bo	b_3	6	b_3	bo	167	b_2	b_5	130	b_5	b_2	118	b_1	b_4	86	b_4	b_1
63	bo	b_3	64	b_3	bo	65	b_2	b_5	106	b_5	b_2	176	b_1	b_4	131	b_4	b_1
6	bo	b_3	7	b_3	bo	101	b_1	b_4	155	b_5	b_2	93	b_2	b_5	173	b_4	b_1
36	bo	b_3	37	b_3	bo	78	b_1	b_4	184	b_4	b_1	157	b_2	b_5	103	b_5	b_2
1	bo	b_3	2	b_3	bo	136	b_2	b_5	139	b_5	b_2	92	b_1	b_4	111	b_4	b_1
21	bo	b_3	22	b_3	bo	132	b_2	b_5	98	b_5	b_2	171	b_1	b_4	126	b_4	b_1
50	bo	b_3	51	b_3	bo	108	b_1	b_4	71	b_5	b_2	141	b_2	b_5	175	b_4	b_1
15	bo	b_3	16	b_3	bo	181	b_1	b_4	68	b_5	b_2	107	b_2	b_5	144	b_4	b_1
44	bo	b_3	45	b_3	bo	87	b_2	b_5	112	b_5	b_2	131	b_1	b_4	186	b_4	b_1
11	bo	b_3	12	b_3	bo	116	b_1	b_4	96	b_5	b_2	140	b_2	b_5	178	b_4	b_1
35	bo	b_3	36	b_3	bo	102	b_2	b_5	174	b_5	b_2	77	b_1	b_4	150	b_4	b_1
18	bo	b_3	19	b_3	bo	96	b_2	b_5	156	b_5	b_2	175	b_1	b_4	105	b_4	b_1
61	bo	b_3	62	b_3	bo	119	b_1	b_4	149	b_4	b_1	180	b_2	b_5	69	b_5	b_2
9	bo	b_3	10	b_3	bo	174	b_2	b_5	83	b_5	b_2	99	b_1	b_4	129	b_4	b_1
20	bo	b_3	21	b_3	bo	68	b_2	b_5	97	b_4	b_1	134	b_1	b_4	187	b_5	b_2
38	bo	b_3	39	b_3	bo	109	b_2	b_5	85	b_5	b_2	170	b_1	b_4	145	b_4	b_1
56	bo	b_3	57	b_3	bo	164	b_1	b_4	116	b_4	b_1	69	b_2	b_5	191	b_5	b_2
34	bo	b_3	35	b_3	bo	98	b_2	b_5	190	b_4	b_1	82	b_1	b_4	138	b_5	b_2
37	bo	b_3	38	b_3	bo	154	b_1	b_4	121	b_5	b_2	165	b_2	b_5	80	b_4	b_1
40	bo	b_3	41	b_3	bo	106	b_2	b_5	169	b_5	b_2	161	b_1	b_4	77	b_4	b_1
54	bo	b_3	55	b_3	bo	121	b_2	b_5	132	b_5	b_2	179	b_1	b_4	73	b_4	b_1
31	bo	b_3	32	b_3	bo	125	b_1	b_4	78	b_4	b_1	188	b_2	b_5	135	b_5	b_2
14	bo	b_3	15	b_3	bo	133	b_2	b_5	102	b_5	b_2	186	b_1	b_4	92	b_4	b_1
3	bo	b_3	4	b_3	bo	183	b_1	b_4	91	b_5	b_2	138	b_2	b_5	128	b_4	b_1
45	bo	b_3	46	b_3	bo	71	b_2	b_5	101	b_4	b_1	192	b_1	b_4	165	b_5	b_2
12	bo	b_3	13	b_3	bo	91	b_2	b_5	115	b_5	b_2	173	b_1	b_4	161	b_4	b_1
55	bo	b_3	56	b_3	bo	100	b_2	b_5	162	b_5	b_2	143	b_1	b_4	76	b_4	b_1
41	bo	b_3	42	b_3	bo	117	b_2	b_5	81	b_5	b_2	168	b_1	b_4	163	b_4	b_1
8	bo	b_3	9	b_3	bo	114	b_1	b_4	74	b_5	b_2	187	b_2	b_5	137	b_4	b_1
32	bo	b_3	33	b_3	bo	169	b_2	b_5	67	b_5	b_2	144	b_1	b_4	118	b_4	b_1
57	bo	b_3	58	b_3	bo	120	b_1	b_4	171	b_4	b_1	151	b_2	b_5	93	b_5	b_2
39	bo	b_3	40	b_3	bo	123	b_2	b_5	134	b_4	b_1	95	b_1	b_4	180	b_5	b_2
27	bo	b_3	28	b_3	bo	127	b_2	b_5	87	b_5	b_2	163	b_1	b_4	160	b_4	b_1
13	bo	b_3	14	b_3	bo	74	b_2	b_5	108	b_4	b_1	145	b_1	b_4	188	b_5	b_2
49	bo	b_3	50	b_3	bo	122	b_1	b_4	79	b_5	b_2	158	b_2	b_5	176	b_4	b_1
30	bo	b_3	31	b_3	bo	190	b_1	b_4	117	b_5	b_2	66	b_2	b_5	152	b_4	b_1
52	bo	b_3	53	b_3	bo	72	b_1	b_4	125	b_4	b_1	182	b_2	b_5	140	b_5	b_2
22	bo	b_3	23	b_3	bo	189	b_2	b_5	136	b_5	b_2	75	b_1	b_4	113	b_4	b_1
64	bo	b_3	1	b_3	bo	135	b_2	b_5	104	b_5	b_2	76	b_1	b_4	192	b_4	b_1
43	bo	b_3	44	b_3	bo	162	b_2	b_5	167	b_5	b_2	105	b_1	b_4	82	b_4	b_1
4	bo	b_3	5	b_3	bo	185	b_2	b_5	65	b_5	b_2	124	b_1	b_4	166	b_4	b_1
60	bo	b_3	61	b_3	bo	110	b_1	b_4	189	b_5	b_2	147	b_2	b_5	99	b_4	b_1
24	bo	b_3	25	b_3	bo	94	b_2	b_5	185	b_5	b_2	150	b_1	b_4	120	b_4	b_1

Memory Mapping for K = 192 & P = 6