HIGH THROUGHPUT VLSI ARCHITECTURES FOR ITERATIVE DECODERS

by

Engling Yeo

Memorandum No. UCB/ERL M03/39

16 October 2003

HIGH THROUGHPUT VLSI ARCHITECTURES FOR ITERATIVE DECODERS

by

Engling Yeo

Memorandum No. UCB/ERL M03/39

16 October 2003

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

High Throughput VLSI Architectures for Iterative Decoders

By

Engling Yeo

B.S. (University of California, Berkeley) 1994
M.S. (University of California, Berkeley) 1995

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy
in

Engineering – Electrical Engineering
and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA, BERKELEY

Committee in charge:

Professor Borivoje Nikolić, Chair
Professor Venkatcha Anantharam
Professor Jasmina L. Vujić

Fall 2003

The dissertation of Engling Yeo is approved:

09/15/2003

Borivoje Nikolić (Chair)   (Date)

Anant   09/18/2003

Venkatachalam Anantharam   (Date)

Jasmina Vujić   10/07/2003

Jasmina L. Vujić   (Date)

ABSTRACT

High Throughput VLSI Architectures for Iterative Decoders

by

Engling Yeo

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Borivoje Nikolić, Chair

This project addresses the algorithms for and implementations of iterative decoders for error control in communication applications. The iterative codes are based on various concatenated schemes of convolutional codes, and low-density parity check (LDPC) codes. The decoding algorithms are instances of message passing or belief propagation algorithms, which rely on the iterative cooperation between soft-decoding modules known as soft-input-soft-output (SISO) decoders.

Iterative decoding is a recent advance in communication theory that is applicable to wireless, wireline, and optical communications systems. It promises significant advantage in bit error rate performance at signal to noise ratios very close to the theoretical capacity bound. However, a direct mapping of the decoding algorithms leads to a multifold increase in the implementation complexity. As deep submicron technology matures, there is a possibility of implementing these applications that were once thought to be too complex to fit onto a single silicon die. We investigate the architectural and implementation issues related to building iterative decoders in VLSI.

In this research, the computational hardware and memory requirements of magnetic storage applications provide a platform for evaluation of iterative decoders. The accomplishments include modifications of algorithms, their simulation, efficient mapping into architectures, and VLSI implementations provide the final measure of complexity in terms of power, size, and speed. The VLSI implementations of iterative decoders based on concatenated convolutional codes or LDPC codes will demonstrate the effectiveness of various methods for reduced-complexity decoding and reduced control logic. Besides storage applications, these research results are applicable to wireless, wireline and optical communications systems.

Professor Borivoje Nikolić
Dissertation Committee Chair

1. Introduction 1

1.1. Motivation 1

1.2. Objective 2

1.3. Scope of work 4

1.4. Development work in error correction codes 5

1.5. Related Work 12

2. Transmission Channels and Coding 14

2.1. Channel capacity 15

2.2. Partial response channel 17

2.3. Partial response maximum likelihood (PRML) systems 20

2.4. Concatenated codes in partial response channels 22

3. Message Passing Algorithms 24

3.1. Constrained coding and SISO decoders 24

3.2. SISO algorithms 27

3.2.1. Maximum a-posteriori decoder 27

3.2.2. Soft-output Viterbi algorithm (SOVA) 33

3.2.3. Message passing algorithm used in LDPC decoder 35

3.3. Requirements of iterative decoders 38

3.3.1. Computational requirements 38

3.3.2. Message-passing requirements for iterative codes 41

3.4. Summary 44

4. Architectures of SISO decoders for turbo codes 46

4.1. System level considerations 46

4.2. MAP decoder 47

4.2.1. Forward and backward recursion 48

4.2.2. Combining forward / backward state metrics 54

4.2.3. Memory requirements 56

4.3. Soft output Viterbi decoder 61

4.3.1. SOVA decoder architecture 61

4.3.2. Add-Compare-Select structures 62
4.3.3. Survivor path decoding 67
4.4. Summary 71
5. LDPC decoder Architectures 73
5.1. Parallel architectures 73
5.2. Serial architectures 76
5.3. Shared memory architectures and partitioned matrix 79
5.4. Computation blocks 82
5.4.1. Fixed-point implementation of lookup table 85
5.4.2. Fixed-point LDPC decoder building blocks 91
5.5. Effects of code construction on implementation 92
5.5.1. Finite field constructions 92
5.5.2. Latin rectangles and improved Ramanujan graphs 93
5.5.3. Turbo product codes 95
5.6. Summary 95
6. Processing schedules of LDPC decoders 96
6.1. Original concurrent schedule 96
6.2. New staggered schedule 97
6.3. Simulation analysis 100
6.4. Pipelined processing elements 102
6.5. Summary 104
7. Physical implementations 105
7.1. FPGA implementation of a MAP decoder 106
7.2. ASIC implementation of a SOVA decoder 107
7.2.1. Design flow 107
7.2.2. Design parameters 109
7.2.3. Physical Tests 110
7.3. Implementation of an LDPC decoder 115
7.3.1. LDPC decoder based on finite field construction 115
7.3.2. Design input 118

7.3.3. FPGA implementation 122

7.3.4. ASIC implementation 122

7.4. Other state of the art implementations 124

7.5. Summary 125

8. Conclusions 126

8.1. Future 127

9. Appendix A: Pin list for SOVA decoder 130

9.1. Pad Frame 130

9.2. Pin List 130

10. Appendix B: PIN LIST FOR LDPC DECODER 133

10.1. Pad Frame 133

10.2. Pin List 133

11. Appendix C: Modifications to ASIC DESIGN FLOW 138

11.1. Design Synthesis 138

11.2. Placement 139

11.3. I/O pad placement 139

11.4. Power routing 141

11.5. Clock tree generation 145

11.6. Detailed routing 145

11.7. Cadence Opus dfII 147

11.8. Pullup and Pulldown Cells 148

Figure 1-1. Turbo (a) encoder and (b) decoder consisting of serial concatenation of a convolution code with EPR4 channel. 7

Figure 1-2. Complexity comparisons between various coding schemes. 9

Figure 2-1. Generic representation of communication systems........15

Figure 2-2. Plot of relationship between minimum rate of code and available SNR for a band-limited binary transmission........17

Figure 2-3. Various types of partial response channels. 18

Figure 2-4. Impulse response for a PR4 system........19

Figure 2-5. Impulse response for an EPR4 system. 19

Figure 2-6. Rate ½ convolutional encoder; code polynomial is $(1, \frac{1+D}{1+D^2})$. 20

Figure 2-7. Finite state machine (a) and trellis (b) representation of convolutional encoder. 21

Figure 2-8. Serially concatenated codes that make use of a partial-response channel as an inner encoder........23

Figure 2-9. Iterative decoder system for serially concatenated codes........23

Figure 3-1 Bi-partite graph representation of a 4-state trellis code. 25

Figure 3-2. Bi-partite graph representation of LDPC codes. 26

Figure 3-3. Convolutional encoder for $1+D-D^2-D^3$. 28

Figure 3-4. Trellis of an 8-state convolutional code........28

Figure 3-5. Convolutional encoder with MAP decoder. 28

Figure 3-6. Two-stage traceback in a SOVA decoder to determine the two ML paths, α and β. 34

Figure 3-7. Plot of $\Phi(x)$ against x. 37

Figure 3-8. Arithmetic computation associated with nodes in factor graphs of (a) a convolutional code, and (b) an LDPC code. 40

Figure 3-9. Pseudo random interleaver realized with three single-bit registers........43

Figure 3-10. Interleavers and deinterleavers implemented using alternating read/write buffers. 43

Figure 4-1. Pipelined blocks in a generic communications receiver. 47

Figure 4-2. MAP algorithm defines operations on a vertical slice of trellis. 48

Figure 4-3. Add-Compare-Select-Add unit for either forward or backward recursions using the $\Psi(.)$ operator as indicated within the box....................50

Figure 4-4. Retimed ACSA that hides penalty of final addition....................50

Figure 4-5. Radix-2 trellis and AACS structure. 51

Figure 4-6. Radix-4 trellis and AACS structure. 51

Figure 4-7. Radix-2 concurrent *AACS* structure. 52

Figure 4-8. Radix-2 compare-select-add-add (*CSAA*) structure 53

Figure 4-9. Transformed add-compare-select-add (*ACSA*) structure....................53

Figure 4-10. ξ block makes use of a binary tree of $\Psi(.)$ operators. 55

Figure 4-11. Direct implementation of BCJR requires memory storage of path metrics corresponding to each node in the trellis for the entire length of the block code. 57

Figure 4-12. Backward iteration using 2 overlapping windows, W_0 and W_1 for BCJR algorithm. The shaded outputs are not used in the ensuing ξ block. 59

Figure 4-13. State-slice of a MAP decoder structure....................59

Figure 4-14. Memory read and write access of branch metrics γ_k. 60

Figure 4-15. System architecture of 8-state SOVA decoder....................63

Figure 4-16. Area comparisons of (a) various ACS structures and (b) a detailed magnification of the ACS, concurrent ACS, CSA comparison....................65

Figure 4-17. Power comparisons of (a) various ACS structures and (b) a detailed magnification of the ACS, concurrent ACS, CSA comparison....................66

Figure 4-18. Example 8-state register-exchange survivor memory unit used in VA-SMU. 69

Figure 4-19. State-slice of register-exchange used in the path-equivalence detector (PED). 71

Figure 4-20. Pipelined section of reliability measure unit (RMU). 71

Figure 5-1. Parallel architecture....................74

Figure 5-2. Parallel LDPC decoder with serial input stream from channel decoder. 75

Figure 5-3. Serial architecture....................77

Figure 5-4. Serial decoding by alternating between two memory buffers containing consecutive blocks of data 77

Figure 5-5. Tracing dependencies through a bipartite graph. 79

Figure 5-6. $M\times N$ parity check matrix partitioned into $j\times k$ subblocks. 80

Figure 5-7. Shared memory architectures with shared computational logic, and interconnect........82

Figure 5-8. Binary adder tree to compute Q_{nm} messages in a multi-stage pipeline. 84

Figure 5-9. Recursive pipelined implementation to compute R_{mn} messages. 84

Figure 5-10. the $\Phi^{-1} \cdot \Phi$ mapping........86

Figure 5-11. The original $\Phi(\cdot)$ function, and its scaled derivatives. 86

Figure 5-12. Tree structure of 2-input Θ operators evaluate check-to-variable messages. 89

Figure 5-13. Correction terms $f(x,y)$ plotted against 3-bit integer inputs x and y.........89

Figure 5-14. Evaluation of correction terms rounded to 0.25 levels. 90

Figure 5-15. Simulation results with a rate 3/4 4095-bit LDPC code based on finite-field construction, with the Θ approximation. 91

Figure 5-16. Rendition of 1023×4092 parity check matrix used in codes based on finite fields. Black dots represent non-zero entries. 93

Figure 5-17. Shared memory decoder architecture for LDPC codes based on Latin rectangles. 94

Figure 5-18. Architecture of TPC decoder partitions the code into 16×16 blocks. 94

Figure 6-1. Concurrent decoding schedule of the message passing algorithm; circles represent variable nodes; boxes represent check nodes. 97

Figure 6-2. Staggered decoding schedule of the modified message passing algorithm with $|S(t)|=1$. 99

Figure 6-3. Architecture for random LDPC decoder with K iterations of staggered schedule. 99

Figure 6-4. Simulation results from random codes (a), and GF codes (b) with concurrent vs. staggered decoding schedule. 101

Figure 6-5. Staggered decoding schedule using one processing element with a 3-stage pipeline. 103

Figure 7-1. Dataflow graph of the ACS of a SOVA decoder design in Matlab Simulink^{TM}. 108

Figure 7-2. High-level dependency graph for the automated design flow [74]. 109

Figure 7-3. Printed circuit board used for testing of SOVA decoders. 110

Figure 7-4. Setup for logical verification of SOVA decoders 111

Figure 7-5. Test structure around CSA for critical path analysis 112

Figure 7-6. Separate clock trees reduce parasitic effects of additional speed test requirements 112

Figure 7-7. Variable delay controlled by *Vdd2* 113

Figure 7-8. Plot of correct output vectors (blue) and actual vectors output by chip (red) at skew delay of 1.6ns 113

Figure 7-9. Die micrograph 114

Figure 7-10. Performance of EPR4 SOVA decoder. 114

Figure 7-11. Example of 1:4 column splitting 117

Figure 7-12. Shift register-based implementation of LDPC code generated from 2D GF(2M) with 1:4 column splitting and a message computation latency of 2. 117

Figure 7-13. Horizontal partitioning of the 1023×4092 parity check matrix 117

Figure 7-14. Top level design entry view in Mathworks Simulink 119

Figure 7-15. Screen capture of top-level LDPC decoder design in Simulink 121

Figure 7-16. Layout of LDPC decoder in 0.13μm CMOS, occupying 3.2mm × 2.7mm. 123

Figure 7-17. Design methodology. Shaded symbols indicate steps modified from the standard Insecta flow. 124

Figure 11-1. I/O Pad arrangement 142

Figure 11-2. Power and ground supply grid. 143

Figure 11-3. Connection between supply I/O pads and metal4 supply rings 144

Figure 11-4. Filler cells with decoupling capacitors 146

Figure 11-5. Pullup and pulldown cells inserted during synthesis. 148

ACKNOWLEDGEMENTS

My lovely wife, Ailee, has supported me in much more ways than meets the eye. She put down everything she had in Singapore, to start a new life with me in United States. She took care of all administrative details, including the final spell check of every journal I ever published. She volunteered to take care of my mother when she was hospitalized. Her happy disposition and composed demeanor calmed my nerves in difficult times. She had made the last four years the best of my life. In short, she has given me more than what I could ever hope to give in return.

I would like to thank my parents, Ng Yoke Foon and Yeo Hoon Chor, my sister, Lou Miang, and brother Eng Khiew, for their constant support and encouragement through every major decision in life, including the choice to return to college after a three-year stint in Singapore. I like to thank my parents-in-law, Ho Choon Meng and Ong Ngoh for their love and financial help. Despite the distance, support from family has remained strong. This has kept Ailee and I motivated throughout this experience.

My advisor, Professor Borivoje Nikolic, played an enormous role in helping me learn more than what I have set out to accomplish. He has been mentor and friend to me. Although working with him can hardly be described as a smooth ride, the results have been more than rewarding. I would also like to thank Professor Venkat Anantharam for providing technical advice over the length of this research, as well as Professor Bob Brodersen and Professor Phil Spector for contributing valuable suggestions to my research.

I survived the pressures and rigors of graduate school, thanks to support from the group of Kangngee Chia, Kinkok Chan, Youyenn Teo, Boonkiat Law, Yeechia Yeo, Ruolei Ng, and alas, four-year-old Sheyuan. These folks were family to me. Incidentally, they were the first people on earth to celebrate my passing of the qualifying examination; the party conveniently took place on the weekend *before* the examination, and proceeded without me. Thanks also to Joshua Garrett, Benny Warlick, Fujio Ishihara, Mike Chen, and Liangteck Pang for riding with me almost every other weekend. For a number of years, I have preached the benefits of mountain biking to everybody I met at Cory Hall. Little did this group realize that my intentions were to get these weekend warriors to share the burden of carrying emergency bike equipment.

My research has had no shortage of help coming from the students and staff at Cory Hall and the Berkeley Wireless Research Center. Thanks to Payam Pakzad, in particular, for providing most of the theoretical background at the early stages of this work. Thanks to Rhett Davis, Stephanie Augsburger, Tina Smilkstein, and Brian Richards for the help in the SSHAFT design flow, which led to the first tapeout. Thanks to Chen Chang, Kimmo Kuusilinna, Nathan Chan, Joshua Garrett and Fujio Ishihara for help in the BEE design flow, which led to the second tapeout. Radu Zlatanovici, Socrates Vamvakos, Stephanie Augsburger, Isaac Sever, Ben Wild, Dejan Markovic, Roy Sutton, and Kostas Sarrigeoridis, thanks for the constructive criticisms for many of my practice talks. In addition, Tom Boot and Brenda Vanoni have been extremely helpful with purchase reimbursements and last-minute courier packages.

I also had the opportunity to work with a number of distinguished undergraduates from UC Berkeley and University of Wisconsin. Melinda Ler, Jane Nguyen, Steve Fang, Henry Lam, and Allen Chen volunteered their time to work on my project, and contributed in many aspects.

Finally, this research has benefited from the generosity extended from beyond the Berkeley campus. Thanks to Kiyoshi Fukahori, Tom Souvignier, and David Gruetter of Texas Instruments for sponsoring the UC Micro project, and supervising my summer internship in 2000. Thanks to Brian Marcus, Bruce Wilson, and Yuan Xing Lee of IBM for technical advice. Thanks to Pantas Sutardja, Nersi Nazari, Toai Doan, and Zining Wu of Marvell for sponsoring the UC Micro project, and feedback on the progress of the work. Thanks to Bhusan Gupta, Ben Coates and Srikanth Muroor of STMicroelectronics for advice with physical IC design and fabrication of test chips.

1. INTRODUCTION

1.1. Motivation

The development of the communications industry is characterized by exponential growth in volume of data and throughput rates. These growths are accompanied by reduced signal-to-noise ratios at which data is detected. In order to maintain the signal integrity, the level of sophistication in error correction methods is required to keep pace with the communication applications. Modern communication systems employ various forms of redundancies to achieve resilience against interference and noise arising out of a multitude of sources.

The complexity of integrated circuits for signal processing has historically tracked the progress in silicon process technology. Each new silicon process generation has allowed integration of increasingly more complex signal processing schemes into a chip, constrained by cost and power requirements. For example, detectors used in disk-drive read channel integrated circuits have moved from 8-state conventional Viterbi decoders, common in 0.35µm technology, to current, state of the art 0.13µm detectors that incorporate 32-state noise-predictive decoders. Despite an exponential growth in implementation complexity, there is diminishing marginal improvement in bit error rate (BER) performance. The situation is thus ripe for revolutionary changes to the coding and signal processing techniques to challenge currently prevalent classes of error-correction algorithms.

Recently, a new class of error correcting codes has demonstrated performance within 0.5dB of the theoretical limits. These codes comprise two or more concatenated block codes with corresponding decoders that iteratively exchange messages reflecting the confidence of each decoded bit. The messages are based on a probability measure rather than the decisions of the decoded bits. Known as the 'soft' information', the value of this measure is repeatedly accessed and refined over the several iterations of decoding. This approach to decoding represents a departure from traditional error-correction algorithms. These methods are collectively known as *iterative decoding.*

Although various forms of iterative decoding have existed for four decades, the discovery of turbo codes [11] and methods for their decoding in 1993 were largely acknowledged to be the *raison d'être* for the current surge in iterative decoding research and development, both in academia and industry. A large number of publications have appeared in the areas of code design and ultimate code performance, but somewhat less attention has been paid to decoding architectures, implementation and system issues. As the communication industry begins to explore the deployment of iterative codes that operate at the capacity of a given channel, a detailed understanding of the physical requirements is necessary to provide an unbiased evaluation. Comparison between BER performance and implementation complexities are indispensable, but will require detailed analysis of the intrinsic requirements for implementation of iterative decoders.

1.2. Objective

Effective error correction can reduce the signal-to-noise ratio (SNR) requirement for an end-to-end reliable communication. Lower SNR requirements in a communication system result in a variety of implementation advantages. Each 3dB of coding gain is capable of doubling the system throughput or transmission range, or reducing the required bandwidth by ½. To an end-user, these benefits can translate into extended battery life in portable wireless devices by lowering transmit power, improved range in high throughput wireline systems such as very high speed digital subscriber lines (VDSL), or increased storage densities on magnetic media.

With this in mind, there is a necessity to explore implementation issues of iterative decoders based on turbo codes or low-density parity check codes [56] for future generation of communication systems. The realization of an iterative decoder will weigh the tradeoffs between coding gain performance and factors affecting the implementation: namely, power, throughput and area. A number of platforms are evaluated for their suitability towards realization of the decoder requirements. In particular, the focus will be on viable high-performance ASIC architectures.

The introduction of any new error-correction scheme on silicon must preserve the manufacturability and testability of today's digital systems. Although initial iterative decoders [111], [120] were based on analog signal processing, these early implementations are sensitive to process and temperature variations, and are difficult to test in production. On the other hand, successful digital implementations, being less susceptible to these adverse effects, will quickly displace the analog predecessors. Hence, the analysis of iterative decoder architectures will be centered on digital implementations.

Iterative decoders are based on block codes, and both encoding and decoding are processed in the context of a block of data. Depending on the application, the number of bits in each block ranges between a few hundred (wireless) to a few thousand (magnetic storage). The soft information exchanged between decoders is typically stored as a three to five-bit fixed-point number. In general, large block sizes and multiple-bit messages combine to form a memory requirement that is an order of magnitude larger than a comparable Viterbi decoder.

The necessity to perform multiple decoding iterations implies that the complexity of the overall decoder is several times larger than traditional decoders. In order to keep the area of implementation and power consumption within practical limits, reduced-complexity methods for the implementation of these decoders will be proposed. In addition, complexity reduction methods are often advantageous towards improving the throughput of the decoders. The analysis explores the tradeoffs between throughput, area and power of implementation, as well as the effects on BER performance of the decoders.

This work demonstrates the effectiveness of the proposed iterative decoder architectures on field-programmable gate arrays (FPGA) and application-specific integrated circuits (ASIC). An FPGA implementation will offer flexibility in code design and effective emulation of iterative decoding algorithms with fixed-point representations. The simulation or run time of an FPGA is expected to be at least an order of magnitude faster than the use of microprocessor-based programs. ASIC implementations offer the best balance between performance, power, and area of implementation. Using the latest process technology, iterative decoding at throughput rates between 500Mb/s and 1Gb/s will be shown.

1.3. Scope of work

This research is aimed at combining the knowledge of iterative decoders at both algorithmic and architectural levels. The results will be presented in three different facets: architectural analysis of decoder structures, code construction exploration with emphasis on hardware implications, and physical demonstration of decoder implementations.

Architectural analysis will permit the realization of iterative decoding hardware with reduced complexities. Effective architectural modifications provide the most impact on the operating performance of the final decoder implementations. The types of structures studied include high throughput decoders applying the Maximum A-Posteriori (MAP) [45] algorithm or the soft-output Viterbi algorithm (SOVA) [38]. These decoders form the building blocks of a turbo decoder. The throughput bottleneck is identified to be the one-step recursion known as the Add-Compare-Select (ACS). This is followed by an evaluation of several competing micro architectures, which demonstrates the optimal range of decoding frequencies that are associated with each option. The architectural evaluations will also include comparisons between serial and parallel structures. The issues and difficulties related to implementation of a parallel or serial architecture are particularly significant to the realization of an LDPC decoder. The decoding algorithm of the LDPC decoder has inherent parallelism, which promotes the implementation of a fully-parallel decoder capable of high-throughput with a low clock rate and low power consumption. In practice, the interconnect properties of LDPC codes exacerbates the exploitation of this parallelism. Alternate architectures, including fully and partially serial LDPC decoders are therefore examined.

The exploration of new code construction techniques will produce LDPC codes that can be efficiently implemented with reasonable amounts of parallel hardware. These new codes combine properties that enhance the feasibility of the decoder implementation, as well as good error-correction performance. Reduced complexity algorithms are also proposed to replace the standard message-passing algorithm that is commonly used with LDPC codes.

The results will be demonstrated on FPGAs and ASICs. The proposed architectures are mapped onto Xilinx Virtex-E FPGAs, and successful implementations of SOVA and LDPC decoders are achieved in 0.18μm and 0.13μm CMOS technologies, respectively. The physical ASIC design includes logical synthesis and back-end placement, routing, and final verification steps. The silicon chips are fabricated and tested on custom-designed digital test-boards.

The magnetic recording channel is used as a demonstration platform. Current trends in magnetic recording demand throughput rates near 1Gb/s. Magnetic read channels typically employ partial response signaling, which influences the choice of iterative decoders. As previously alluded, developments in the magnetic storage industry are centered about improving the storage density. Increased linear densities on magnetic media lead to degraded SNR. However reliability with very low BER must be maintained by the error correction mechanisms.

1.4. Development work in error correction codes

During the period from the 1950's to 1993, the development of code construction techniques has largely obtained incremental results in terms of coding gain performance. The historical milestones achieved by various rate-1/2 codes can illustrate this. In 1960's, a Bose-Chaudhuri-Hocquenghem (BCH) code, [113-114], demonstrated BER less than 10^{-5} with a SNR that is 5.4dB away from the theoretical limit. With the rise in popularity of the Viterbi decoder [3], these codes were replaced with convolutional codes in the 70's, which achieved similar performance at 4.5dB. By the late 80's, the combination of convolutional codes with Reed-Solomon decoding in magnetic recording applications provided less than 10^{-15} BER at 3.9dB from the theoretical limit. Table 1-1 shows that the BER performance has progressed at a rate of less than 1dB per decade between 1960 and 1990.

In 1993, turbo codes demonstrated the ability to achieve 10^{-5} BER at only 0.7dB away from the theoretical limit. The impact was revolutionary. Not only has BER performance jumped by 3dB, it has also proven that the theoretical limit of a transmission channel, based on the communication principles pioneered by Shannon in 1948 [97], was within reach of less than 1dB.

The birth of turbo codes renewed interests in the class of low-density parity check (LDPC) codes. These codes were first proposed in 1962 [56], but were largely ignored because the size of the code, in tens of thousands bits, made it intractable for practical applications. The advent of faster microprocessors in the late 90s paved the way to more effective design and evaluation of both turbo codes and LDPC codes.

In the decade that followed the introduction of turbo codes, iterative decoding has been a subject of continual interest in the communications community. This is evident in the number of new communication standards that have been specified of late. Table 1-2 lists some recent communication standards that have adopted iterative decoding for forward error correction. In addition, iterative decoding is also currently being considered for high performance storage applications that require above 1Gb/s throughput, as well as 10Gb/s optical communications.

TABLE 1-1.
HISTORY OF RATE ½ CODES

Year	Rate ½ code	SNR required for 10^{-5} BER
1948	Shannon Limit	0 dB
1967	(255,123) BCH	5.4dB
1977	Convolutional Code	4.5dB
1990	Convolutional + Reed-Solomon Codes	3.9dB
1993	Iterative Turbo Code	0.7dB
2001	Iterative LDPC Code	0.00245dB

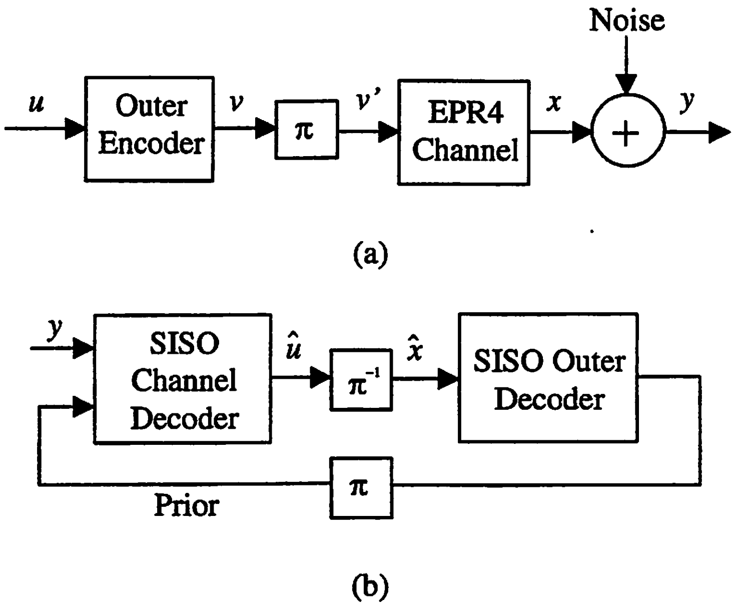


Figure 1-1. Turbo (a) encoder and (b) decoder consisting of serial concatenation of a convolution code with EPR4 channel.

TABLE 1-2
STANDARD SPECIFICATIONS FOR TURBO DECODING

Standard	Application	Iterative Code	Throughput
DVB-RCS	Digital Video Broadcast	Parallel conc. of 8-state conv. codes	68Mb/s (rate 7/8)
DVB-S2	Digital Video Broadcast	LDPC (block size = 64800 bits)	165Mb/s (rates 1/2, 2/3, 3/4, 4/5, 5/6, 7/8, 8/9, 9/10)
IEEE 802.16	Wireless Networking (MAN)	Turbo product codes	25Mb/s (rate 5/6)
3GPP UMTS	Wireless Cellular	Parallel conc. of 8-state conv. codes	2Mb/s (rate 1/3)
CCSDS	Space Telemetry	Parallel conc. of 16-state conv. codes	384kb/s (rate 1/2)

Turbo codes are formed using two or more component convolutional encoders, arranged either in a parallel or serial concatenation, and separated by interleavers. The interleavers construct a long code from short memory convolutional codes. Decoding relies on the iterative passing of posterior-probabilities between two or more soft-input-soft-output (SISO) decoders separated by interleavers and de-interleavers. An example of a serially concatenated turbo code is shown in Figure 1-1. The interleaver and deinterleaver are shown as π and π^{-1} respectively.

One other class of codes that will be analyzed in detail are LDPC codes. These codes are constructed from bipartite graphs consisting of variable nodes and constraint nodes. Each variable node represents a bit, while each constraint node represents a parity checksum of the subset of variable nodes adjacent to it. A sequence of bits that satisfy all the parity constraints is a valid codeword. The iterative decoding of LDPC codes computes messages corresponding alternatively to the variable nodes and the constraint nodes, and passes these messages along edges defined by the underlying graph.

The bipartite graph of an LDPC decoder defines the network for messages to be passed between a large number of nodes. Similar to the interleaver, a direct mapping of the network using hard-wired routes leads to congestion in the interconnect. The congestion can be circumvented through the use of memory. However, unlike the interleavers used in turbo codes, which have a one-to-one connectivity, LDPC graphs have at least a few edges emanating from each variable node. The number of edges is several times larger than that in an interleaver network, and results in higher cost of memory requirement and placing the memory access in the critical path of the decoder.

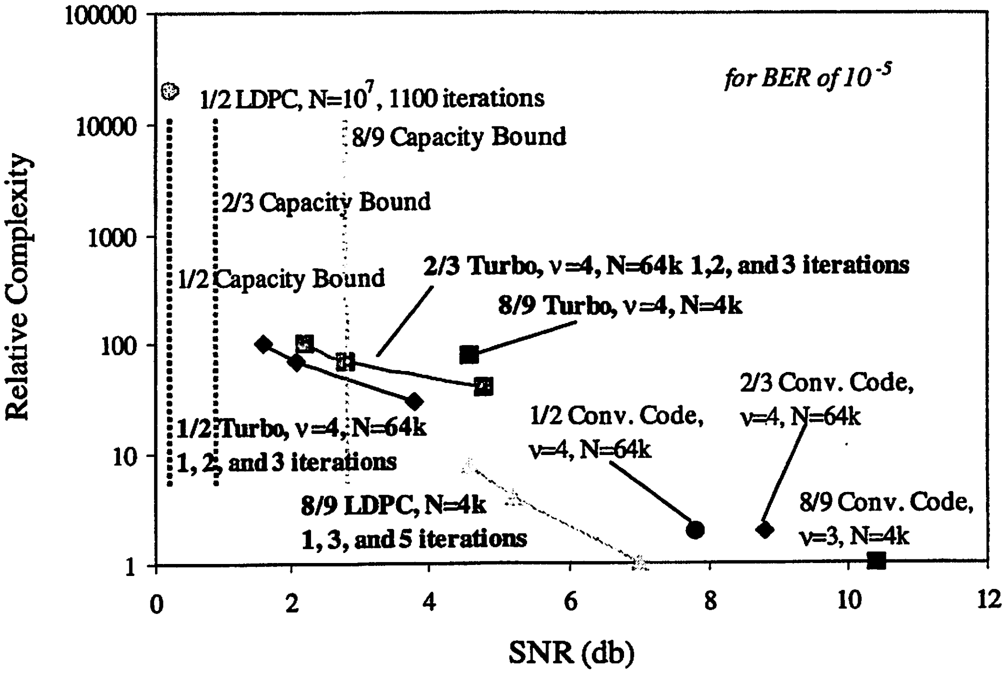


Figure 1-2. Complexity comparisons between various coding schemes.

Although interests have flourished in the development of iterative decoding techniques based on turbo or low-density parity-check codes, difficulties persists at incorporating iterative decoding systems into commercial products. Figure 1-2 shows the relative computational complexity and memory requirement comparisons between conventional convolutional codes and the iterative codes, based on the number of additions required. The plotted values are obtained through 64 iterations of decoding. Both turbo and low-density parity check decoders require soft-input-soft-output decoders, which are about 3 to 5 times the complexity of a convolutional decoder. Additionally, the iterative nature of the decoding leads to overall complexities that are at least an order greater than that of existing convolutional decoders.

The choice of platforms for the implementation of iterative decoding is dictated primarily by the performance constraints such as throughput, power, area, and latency, as well as two often understated and intangible considerations: flexibility and scalability. Flexibility of a platform represents the ease with which an implementation can be updated for changes in the target specification. Scalability captures the ease of using the same platform for extensions of the application that may require higher throughputs, increased code block sizes, higher edge degrees for low density parity check codes, or increased number of states in the constituent convolutional code of the turbo system.

General-purpose microprocessors and digital signal processors (DSPs) have a limited number of single-instruction-per-cycle execution units but provide the most flexibility. These platforms naturally implement the serial architecture for iterative decoding. Microprocessors and DSPs are used as tools by the majority of researchers in this field to design, simulate, and perform comparative analysis of iterative codes. Performing simulations with BER below 10^{-6}, however, is a lengthy process on such platforms. Recently, there has been increased momentum in the use of DSPs in wireless devices built to standards specified by the third generation partnership program (3GPP). These specifications require turbo decoding at throughputs up to 2Mb/s, which is an order of magnitude faster than rates that are typically achievable by a handful of execution units. The advanced DSPs include a "turbo coprocessor" [118], which is essentially an ASIC accelerator with limited programmability.

FPGAs offer more opportunities for parallelism with reduced flexibility. However, fully parallel decoders face mismatch between the routing requirements of the programmable interconnect fabric and edges in a factor graph. FPGAs are intended for datapath intensive designs, and thus have an interconnect grid optimized for local routing. The disorganized nature of an LDPC or interleaver graph, for instance, requires global and significantly longer routing. Existing implementations of iterative decoders on FPGA continue to circumvent this problem by using time-shared hardware and memories in place of interconnect.

Custom ASIC is well suited for direct mapped architectures, offering even higher performance with further reduction in flexibility. An LDPC decoder [1] implemented in 0.16μm CMOS technology achieves a 1Gb/s throughput by fully exploiting the parallelism in the LDPC decoding algorithm. The logic density of this implementation is limited to only 50% to accommodate a large on-chip interconnect. In addition, the parallel architecture is not easily scalable to codes with larger block sizes. For decoding within 0.1dB of the capacity bound, block sizes with tens of thousands of bits are required [93]. With at least 10 times more interconnect wires, a parallel implementation will face imminent routing congestion, and may exceed viable chip areas.

Current ASIC implementations of turbo decoders [98] are serial, targeting wireless applications. Decoding throughput is 2Mb/s with 10 iterations of the two constituent convolutional decoders. A high throughput ASIC turbo decoder, limited by the interleaver memory access, should be able to decode at throughputs over 500Mb/s. Table 1-3 provides a summary of related implementations on different computational platforms.

Table 1-3

SUMMARY OF PLATFORMS FOR ITERATIVE DECODERS

Platform	Architecture	Example implementations	Implementation difficulty
Microprocessor/ DSP	Serial	133kb/s rate-½ LDPC decoder on DSP [110]	Limited number of processing units (ALU)
FPGA	Parallel	None	Mismatch of interconnect requirements and capabilities
FPGA	Serial	56Mb/s rate-½ LDPC decoder [109] 6.5Mb/s 8-state MAP decoder [108] (3 windows)	Control for memory access
Custom ASIC	Parallel	1Gb/s rate-½ LDPC decoder [1] 1024-bit code block	Routing congestion; Not scalable
Custom ASIC	Serial	2Mb/s 8-state MAP decoder [33]	Interleaver addresses computed on the fly. Implementation was optimized for low power. 500Mb/s high throughput MAP decoder is theoretically feasible.
Custom ASIC (Analog)	Parallel	Analog MAP decoder in BiCMOS technology [111]	Interleavers not included. Sensitive to process and temperature variations. Difficult to test in production. Not scalable with improvements in process technology

1.5. Related Work

Currently, a handful of research implementations of the Maximum A-Posteriori decoder (used in turbo decoders) [33], [89], [98], soft-output Viterbi decoders [14], [16], and LDPC decoders [31], [96] are available. The industry [1], [113], [114] has also been active in the development of iterative decoders. In general, these efforts are targeted towards wireless applications. Low rate codes such as rate-1/2, or 2/3 codes, are considered, and the decoders have throughput requirements in the order of a few Mb/s and stringent power constraints. In addition, higher throughputs in the neighborhood of Gb/s have been achieved using analog methods [111] [120]. Although these efforts differ from the objective of this work (high-throughput digital implementations with high code rates), they provide valuable data points for comparison of the various architectures.

To date, most efforts in implementations of iterative decoders pay little attention to the construction of the code. Conversely, notable results from [87], [88], and [93] have demonstrated very successful methods for code construction techniques, but with little considerations towards the implementation issues of the decoders. The discontinuity between code construction and decoder implementations often leads to conflicts between the requirements of high performance error-correction codes and the practical constraints that limit the realizations of the decoders. This research is a deliberate departure from the above methodology by considering both code construction and its implications on the decoder architectures. We combine properties that lead to good performing codes with structural designs that permit the practical implementation of the decoders. In this light, the approach has similar objectives with ongoing efforts in LDPC implementations by Mansour and Shanbhag [96], though the detailed approaches are tangential with respect to one another.

Iterative decoding has been considered for use in magnetic recording [37], [66], [68] [69] applications. The codes considered have high code rates (e.g. 8/9, 16/17), and decoder throughput requirements are above 500Mb/s. The prevalent use of a partial response signaling has driven the use of the transmission channel as a rate-1 convolutional encoder. The next chapter will provide details of such a channel model. The partial response channel is described and evaluated for its suitability towards different types of iterative codes. A number of current channel detection methods are also introduced.

2. TRANSMISSION CHANNELS AND CODING

This chapter defines characteristics of a binary communication channel that will be used as a demonstration platform for the proposed iterative decoder architectures. The properties of the channel model are motivated by requirements from magnetic storage application. Conventional magnetic hard disk channels employ partial response maximum likelihood detection methods. As areal recording densities rise towards 100Gb/inch^2 and beyond, the detection will be required to operate at lower SNRs. Perfect channel equalization, which used to be the basis of forward error correction, becomes increasingly difficult to achieve. Hence there is a requirement for advanced signal processing that will be able to maintain the integrity of the system.

Iterative decoding techniques based on turbo or low-density parity-check codes promise substantial gains in SNR performance. The main difficulties in incorporating iterative decoding systems in existing commercial products are the complexity of the decoder, its size, and implementation of timing recovery, as well as possible byte-error propagation. The successful implementation of these systems will allow the use of iterative decoders in magnetic disk drive read channels with data throughputs that significantly outperform that of current commercial systems, while maintaining manufacturability and testability.

As a vehicle for the performance analysis, a common channel model is described in the following sections. The sector size comprises 4096 user bits, which is a representative block size for most iterative decoding applications such as high-speed wireline and optical communications.

2.1. Channel capacity

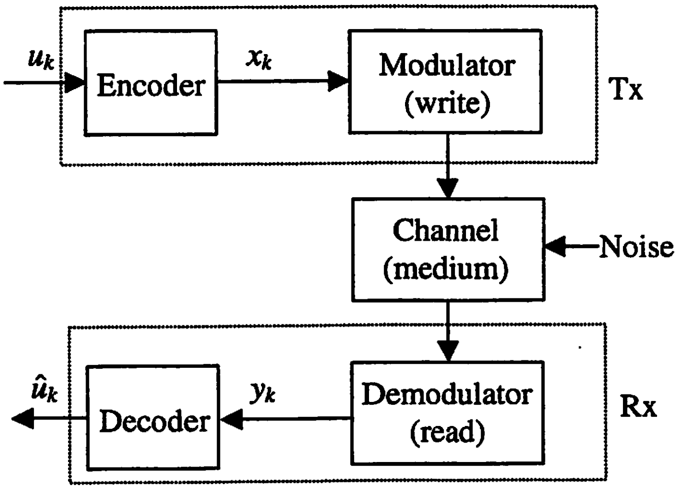


Figure 2-1. Generic representation of communication systems

The classical communication system can be categorized into a number of broad areas depicted in Figure 2-1. The *user* bits, u_k, are encoded through a pre-determined coding scheme. The output, x_k, is passed through the modulator and demodulator, which perform physical transmission of the encoded information through a non-ideal channel, which introduces noise, attenuation, phase delay and other detrimental effects. Forward error correction commonly assumes additive white Gaussian noise. This affects the design of the demodulator, which usually includes a channel detector and equalizer.

In 1948, Shannon introduced the general theory of *coding*. The objectives are two-fold. First, the number of bits required for representation of a given sequence of user bits is minimized (source coding). Secondly, the transfer rate achievable with error-free transmission is maximized (channel coding). The following discussion is restricted to discrete-time systems with discrete-inputs and continuous outputs. This model is relevant for the majority of digital communication systems in which the encoders operate on binary inputs, while the decoders are subject to thermal noise from the channel.

Source coding is based on a statistical model of a generator for the user bits, u_k. Assuming the simplified special case that these input samples are drawn from a random process U_k with independent and identically distributed (i.i.d) samples, the average number of bits required to represent each user bit without distortion is given by the entropy of U, defined in 2.1, with Ω_U as the alphabet space of U. Sampling the source is at r repetitions per second, the rate of the source, R, is given by 2.2. The theorem then states the source can be encoded into a bit stream with minimum bit rate of R.

$$H(U) = -\sum_{u \in \Omega_U} p(U_k = u)\log_2 p(U_k = u) \tag{2.1}$$

$$R = rH(U) \tag{2.2}$$

Likewise, the output of the communication system shown in Figure 2-1 can be considered as a random process $\hat{U}_k$. The concept of channel coding defines the capacity per symbol, C_S, of a channel to be the maximum mutual information, I, between the input random variable U and the output random variable $\hat{U}$.

$$I(U;\hat{U}) = H(U) - H(U \mid \hat{U}) \tag{2.3}$$

$$C_s = \underset{p(U_k=u)}{Max}\, I(U;\hat{U}) \tag{2.4}$$

Transmitting at a rate of s symbols per second, the capacity of the channel, C, is given by

$$C = sC_s \tag{2.5}$$

The source coding and channel coding theorems can be combined to form a general channel capacity theorem. Given a source with rate R, and a channel with capacity C, there exist encoders that will ensure asymptotically error-free transmission if $R < C$.

From an implementation perspective, it is often more useful to define the capacity of a channel in terms of the available SNR. For example, the capacity of a band-unlimited channel for binary transmission is defined in 2.6 as a function of the signal power, P, and noise variance, N. A plot of the relationship is shown in Figure 2-2.

$$R \leq C = ½ \log_2(1+P/N) \tag{2.6}$$

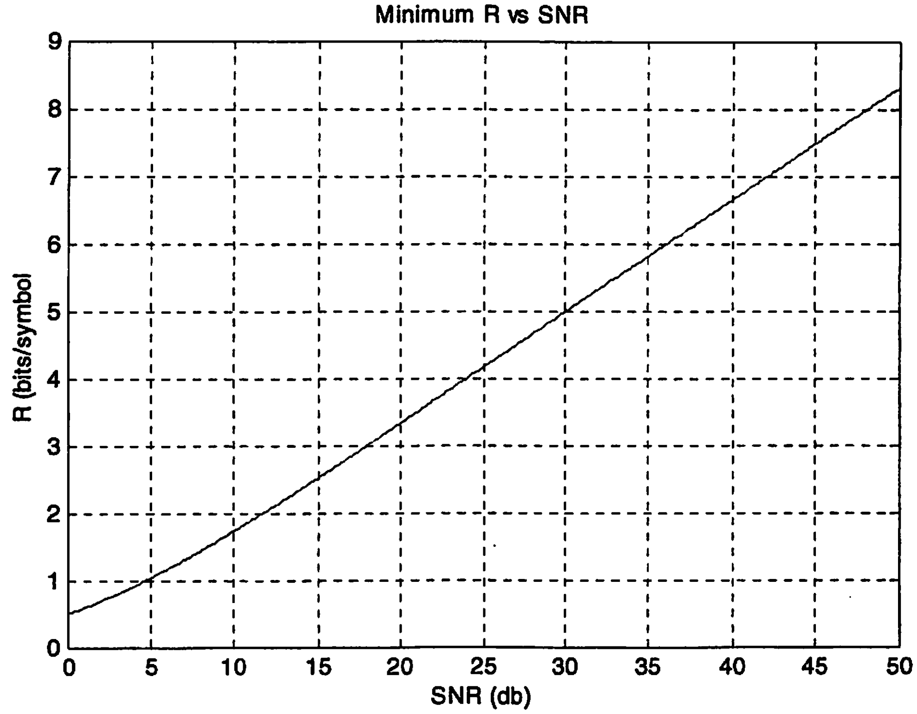


Figure 2-2. Plot of relationship between minimum rate of code and available SNR for a band-limited binary transmission.

2.2. Partial response channel

Magnetic read channel systems make use of non-return-to-zero invert (NRZI) modulation. Each binary bit is stored by magnetizing the medium with one of two possible magnetic field directions. The read head contains a ferromagnetic material that hovers less than 20μm above the rotating medium. Changes in magnetic fields induce in a pulse. A positive pulse corresponds to a 0→1 transition and vice-versa. The isolated pulse shown in Figure 2-3 is modeled as a Lorentzian, with the 50% pulse-width, PW_{50}, defined as the interval over which the height of the pulse is greater or equal than 50% of its maximum value. As data rates increase, consecutive analog pulses may be overlapped due to the smoothening effect of the bandwidth limitations in the receiver. This phenomenon causes simple peak detection techniques to fail.

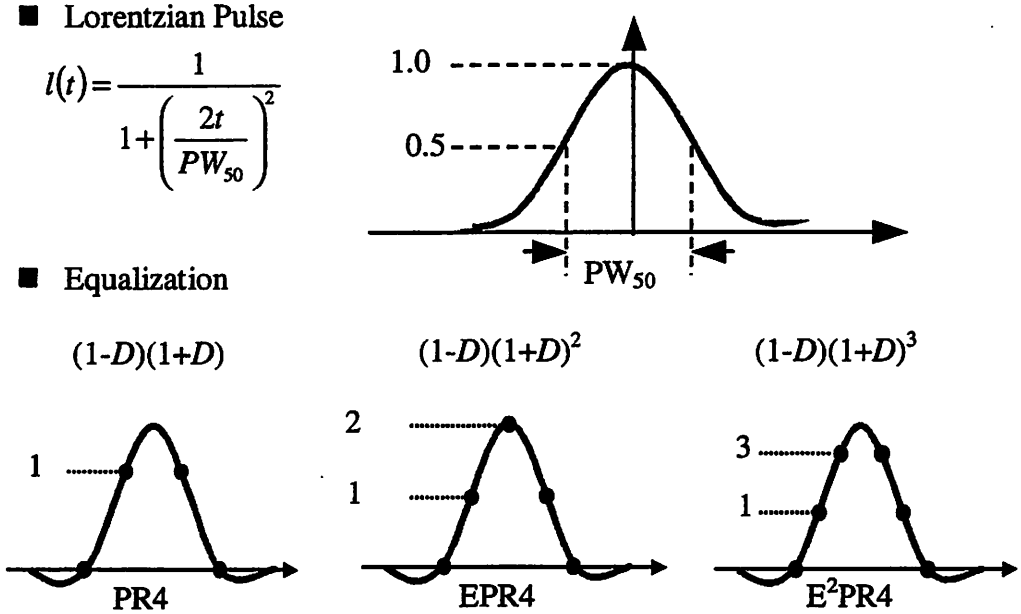


Figure 2-3. Various types of partial response channels.

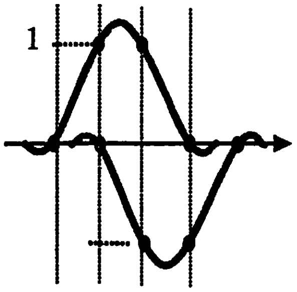
Partial response systems equalize the isolated pulse to preset response targets shown in Figure 2-3. The sampled sequence of an isolated pulse in a partial response class 4 (PR4) system is [0 1 1 0]. Likewise, an enhanced partial response class 4 (EPR4) system is sampled as [0 1 2 1 0], and a double-enhanced partial response class 4 (E^{2}PR4) system, as [0 1 3 1 0].

These partial response systems correspond to convolutional codes with different equivalent polynomials. The impulse response of the channel is given by the superimposition of a pair of positive and negative pulses, separated by a sampling period. Figure 2-4 and Figure 2-5 illustrate such behavior and provide the equivalent code polynomials for the two example partial response channels.

Stored bits: 0 0 0 1 0 0 0

Magnetic field: → → → ← → → →

Generated pulses:



Sampled sequence: 0 0 1 0 -1 0 0 0 0

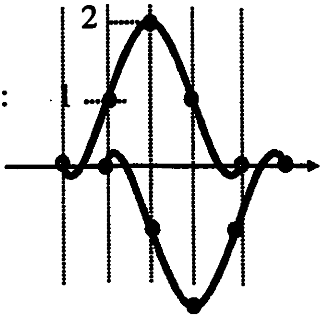
Polynomial: $1 - D^2 = (1+D)(1-D)$

Figure 2-4. Impulse response for a PR4 system.

Stored bits: 0 0 0 1 0 0 0

Magnetic field: → → → ← → → →

Generated pulses:



Sampled sequence: 0 1 1 -1 -1 0 0…

Polynomial: $1 + D - D^2 - D^3 = (1-D)(1+D)^2$

Figure 2-5. Impulse response for an EPR4 system.

2.3. Partial response maximum likelihood (PRML) systems

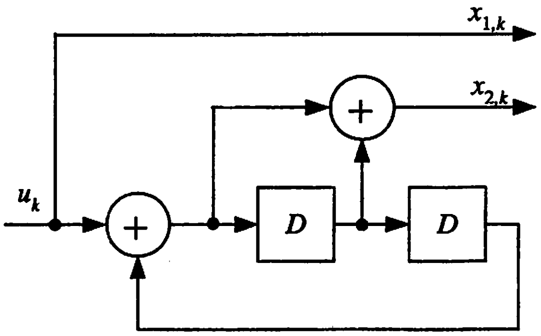


Figure 2-6. Rate ½ convolutional encoder; code polynomial is (1, $\frac{1+D}{1+D^2}$).

Since the 1990's 'Partial Response Maximum Likelihood' (PRML) detectors have been the choice detectors for uncoded, linear, intersymbol-interference (ISI) magnetic channels. PRML systems convert the continuous-time signal into discrete-time samples that are sufficient statistics for decoding. Under this conversion, the ISI channel can be treated as a convolutional encoder, with coefficients that are extracted from the equalized pulse shape. This permits the channels to be decoded with the Viterbi algorithm, which is used with convolutional codes to detect the sequence of input bits. The operation of a convolutional encoder and the Viterbi decoder is briefly explained.

Convolutional encoders make use of v shift-registers to store a short history of the input bits. The specific wiring between the shift registers is defined by the code polynomial. A rate ½ convolutional encoder with a polynomial of (1, $\frac{1+D}{1+D^2}$) is shown in Figure 2-6. The code polynomial contains a recursive feedback loop. For applications that require other code rates, the sequence of x_k and x'_k can be punctured accordingly.

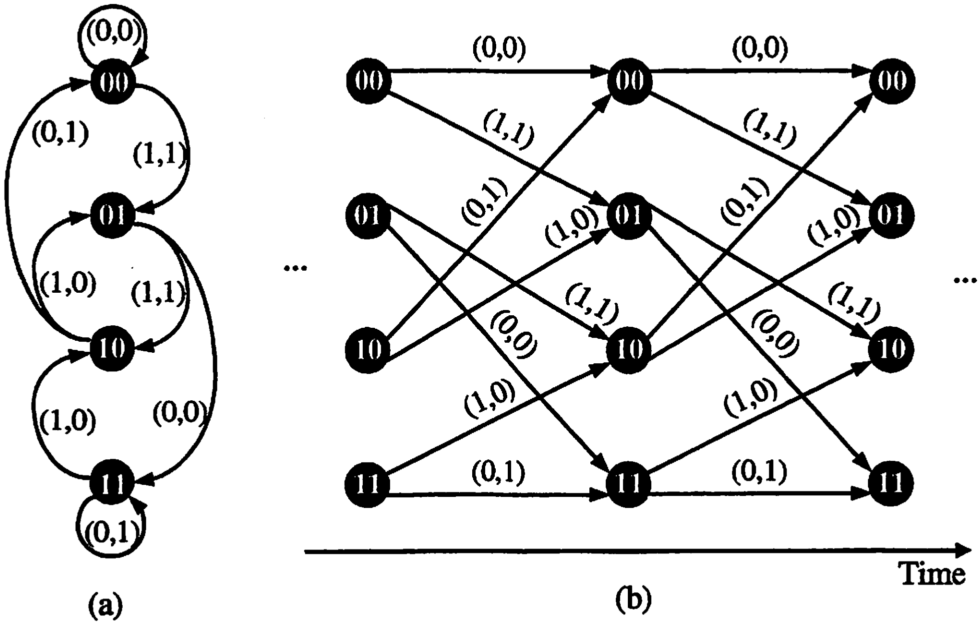


Figure 2-7. Finite state machine (a) and trellis (b) representation of convolutional encoder.

The configuration of the registers of a convolutional encoder define a finite state machine. For a binary convolutional encoder with ν registers, there are up to 2^{ν} states. Figure 2-7(a) shows this finite-state-machine with each transition edge labeled with an input/output pair. The demodulator decodes by recreating the sequence of states traversed in the encoder. This introduces an additional time dimension, and the encoder is represented as a time-expanded state-machine, also known as a trellis (Figure 2-7b).

The trellis is a convenient way to represent the evolution of the finite-state machine. In addition, the function of a Viterbi decoder is very often explained as a search for the most likely path through a trellis. The nodes found along this path reflect the sequence of input bits. The binary decisions produced by this decoder are known as 'hard' outputs. These differ from 'soft' output decoders that provide an additional numerical value measuring the confidence of each decoded bit. Soft output decoders are introduced briefly in the next section, and will be elaborated in the context of building blocks for iterative decoding in the following chapters.

In PRML systems, the equivalent code polynomials indicate the number of states required in the Viterbi decoder. For example, the PR4 and EPR4 systems shown in Figure 2-4 and Figure 2-5 require 4-state and 8-state decoders respectively. As the amount of ISI increases with higher recording densities, state-of-the-art detectors evolve to 16-state and 32-state systems. This causes an exponential growth in the complexity of the detectors despite improvements in error correction performance that are measured in less than 0.5dB increments.

2.4. Concatenated codes in partial response channels

Partial response systems are particularly well suited for implementation of serially concatenated forward error correction codes [68]. Figure 2-8 shows an application of turbo code with an outer encoder serially concatenated with an inner channel encoder.

An interleaver, π, separates and decorrelates the sequence of bits between the two encoders. This allows an iterative suboptimum-decoding algorithm based on uncorrelated information exchange between the two component decoders to be applied. The interleaver improves the coding gain through constructing a long code from short memory convolutional codes. Long codes have larger minimum distance and lower number of low-weight codewords. Interleavers also make the overall code more resilient against burst errors. The minimum sizes of the interleavers that will result in substantial improvement in error performance is in the order of a few hundred bits.

Iterative decoders rely on the repetitive exchange of information and cooperation between two or more soft-input soft-output (SISO) decoders that are matched to the inner or outer codes. Soft-output decoders for convolutional codes implement the Bahl-Cocke-Jelinek-Raviv (BCJR) [4] or Soft-Output Viterbi algorithms (SOVA) [5]). Both algorithms are instances of a larger class of message-passing algorithms, which exploit the linear structure of convolutional codes. An example configuration is shown in Figure 2-9.

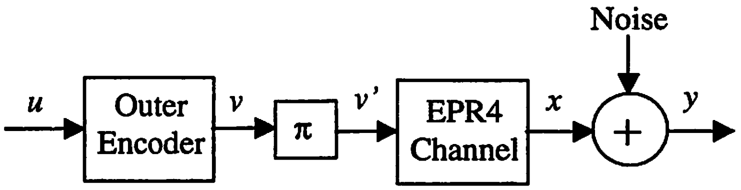


Figure 2-8. Serially concatenated codes that make use of a partial-response channel as an inner encoder.

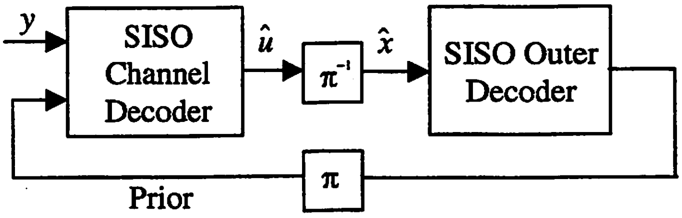


Figure 2-9. Iterative decoder system for serially concatenated codes.

The outer code can be replaced with a LDPC code. Systems employing LDPC encoding do not require explicit interleaver/deinterleavers. By construction, each LDPC code introduces random ordering of at least a few thousand bits. These codes generally have less ordered structures, which leads to increased routing-complexity in LDPC decoders.

In the following chapter, the message-passing algorithm will be discussed in detail. The decoding algorithms for both turbo codes and LDPC codes will be elaborated. These will reveal the implementation requirements of the corresponding decoders.

3. MESSAGE PASSING ALGORITHMS

This chapter introduces the concept of constrained coding, which is the basis of all iterative decoding. Iterative decoders rely on the cooperation between two or more SISO decoders. Each SISO decoder implements a message-passing algorithm, which is defined according to the type of coding constraint(s). These concepts are presented in the context of turbo codes and LDPC codes.

The computational complexities and message-passing network requirements associated with realization of message passing algorithms are introduced. These algorithms are analyzed in terms of their VLSI requirements and limitations, and the impact of complexity-reduction techniques on BER performance. For example, the messages are represented in log-likelihood form for the benefit of reduced hardware complexity. These issues will be examined within a unified graphical framework that consists of an interconnected network of variable nodes and constraint nodes [7].

3.1. Constrained coding and SISO decoders

Constrained coding is loosely defined as the conversion of a block of user bits into a codeword comprising intersecting subsets of coded bits. The elements in each subset are bounded by a constraint, such as an even parity or a valid codeword under BCH encoding. For instance, consider the two convolutional encoders used in the serially concatenated turbo scheme, which was presented in the last chapter. Each output bit of a convolutional encoder is constrained by an even parity with respect to a small number of bits in the input sequence. As the parity results from the outer encoder feed through the interleaver and into the inner encoder implemented by the EPR4 channel, the sequence of neighboring bits is further constrained.

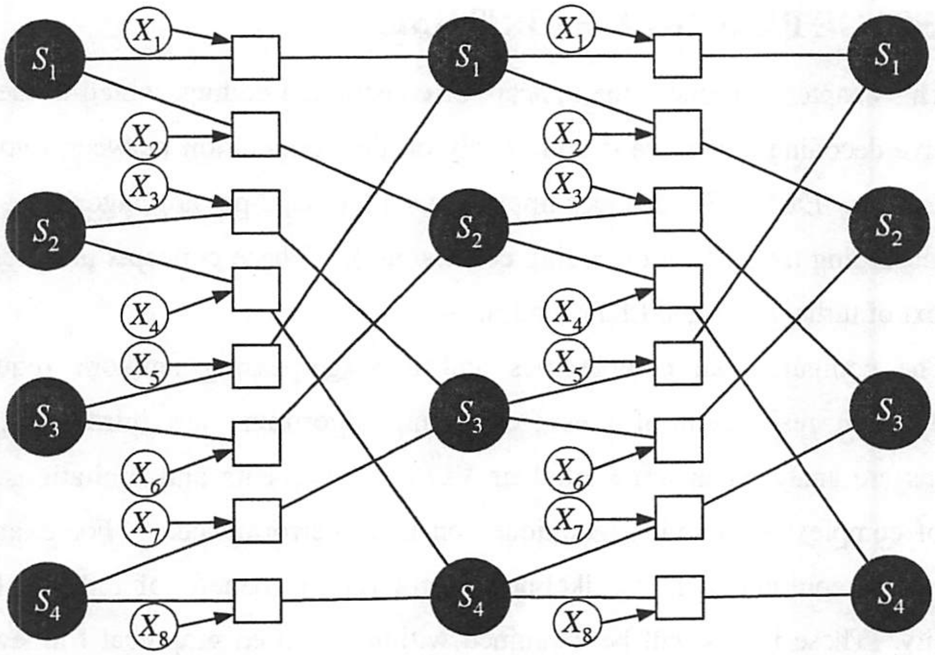


Figure 3-1 Bi-partite graph representation of a 4-state trellis code.

SISO decoders operate with respect to one or more of these constraints specified by the construction of the code. This explains the intuitive use of the two SISO decoders matched to the convolutional codes. In general, the relationship between the variables in the code and the constraints that bind them together can be represented by a bi-partite graph. This graph consists of two classes of nodes: variable nodes and constraint nodes.

Each variable node is connected to a group of check nodes, and vice-versa. Variable nodes symbolize the outcome of either individual bits, or groups of bits. Each constraint node represents a particular rule that is applied on the adjacent variable nodes. An example bi-partite graph representation of a turbo convolutional code is shown in Figure 3-1. It has the form of a trellis, which characterizes the time-indexed finite state machine within a convolutional encoder. The circles correspond to variable nodes. Dark circles ($S_{1\text{-}4}$) represent the states in the trellis, which are formed by stored input values in the encoder, while the light circles ($X_{1\text{-}8}$) represent the output values of the encoder. Each constraint node (white squares) binds two adjacent states with the output of the encoder. In other words, by observing the starting and ending states, the corresponding transmitted output of the encoder can also be determined.

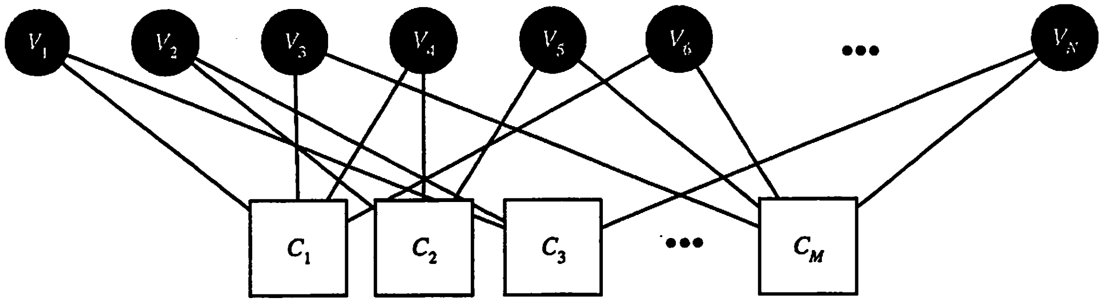


Figure 3-2. Bi-partite graph representation of LDPC codes.

The linear structure of the trellis provides an implementation advantage, which is exploited by SISO decoders implementing either the MAP algorithm or the SOVA. The trellis is partitioned into cascading sections of identical slices. Each slice is representative of a sample period. The decoders employ recursive algorithms to implement the decoding operations in a serial fashion. Over time, a number of possible paths through the trellis of the code are reconstructed, and the most-likely outcome will provide a decoded decision. The soft values, or confidence measures relating to each decoded bit, are based on the differences between the aggregate weights associated with these paths.

Another bipartite graph representation, which is used for representing LDPC codes, is shown in Figure 3-2. The variable nodes (dark circles) represent the coded bits (including both user and parity bits) while the constraint nodes (squares) represent even-parity checksum constraints. In general, LDPC codes are described by sparse graphs with far less structure than the previous trellis example.

SISO decoders used with LDPC codes have a finer granularity, and are organized as check node or variable node processing elements (PE) according to the bipartite graph. The edges in the graph correspond to interconnect between the PEs. In the above example, a check node PE, $C_{1,}$ inputs a list of messages from the adjacent variable node PEs, V_1, V_3, V_4, and V_6. The computation evaluates a list of posterior probabilities, which are returned to the same set of adjacent variable nodes. Likewise, an example variable node PE, V_2 will be evaluating messages that are passed between V_2 and the adjacent check node PEs, C_2 and C_3. These PEs implement the message-passing algorithm, which will be described in Section 3.2.3.

In both turbo codes and LDPC codes, the decoding process begins by initializing each variable node with a prior probability. The prior probabilities are based on some initial assessment provided by the demodulator, such as the sampled signal obtained at the receiver, or any a-priori information of the probability distribution of the set of information bits. The iterative nature is reflected by the repeated evaluation and relay of messages between the two classes of nodes. The process of decoding with respect to distinct constraints and information exchange effectively propagates information throughout the entire graph. It eventually results in a solution based on the weighted inter-dependencies between all variables in the block code.

3.2. SISO algorithms

Three particular SISO algorithms are discussed: The MAP decoder and a SOVA decoder, which are used in turbo codes, and the message-passing algorithm used with decoding LDPC codes and block product codes.

3.2.1. Maximum a-posteriori decoder

A MAP decoder implements the BCJR algorithm [45]. It is used to obtain the *a-posteriori* information for partial response channel decoding, as well as outer decoding when a convolutional code is employed as the outer code. A MAP decoder provides the log-likelihood of each bit received from a convolution encoder. Convolutional encoders are typically described by a discrete, causal, linear, time-invariant transfer function, such as the $1+ D - D^2$-D^3 encoder shown in Figure 3-3. D^n represents an input delayed by n sample periods.

The encoder implements each delay with a single-bit register. A finite state machine, whose states are given by the contents in the registers, can model the entire encoder. An L-delay encoder with binary inputs will therefore have 2^L states. An example of a radix-2 trellis for a binary input system is shown in Figure 3-4.

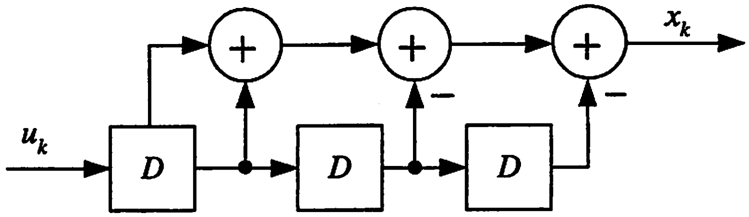


Figure 3-3. Convolutional encoder for $1+D-D^2-D^3$.

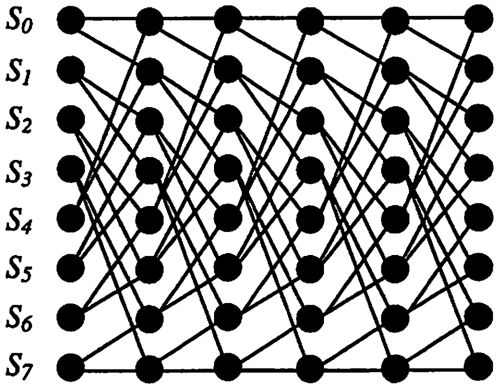


Figure 3-4. Trellis of an 8-state convolutional code

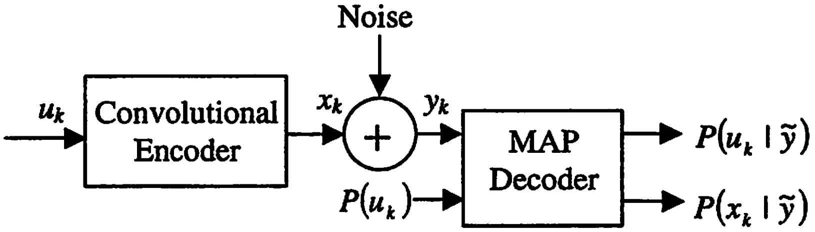


Figure 3-5. Convolutional encoder with MAP decoder.

The operation of the MAP decoder is explained in the context of a single convolutional encoder/decoder system, as shown in Figure 3-5. In general, the MAP decoder is a two-input, two-output discrete-time system that is applied to a block of N bits. The inputs are the received symbols y_k, and prior probabilities, $p(u_k)$, of the encoder inputs, $u_k \in \{0,1\}$. The time indices are represented by $k \in \{1,\cdots,N\}$.

Define the set of received symbols as $\tilde{y}=\{y_1, y_2, y_3, \ldots, y_N\}$. Based on the observations, $\tilde{y}$, the decoder outputs the conditional probabilities of x_k and u_k, expressed as $p(x_k \mid \tilde{y})$ and $p(u_k \mid \tilde{y})$ respectively. Recall the trellis representation of the encoder finite state machine. The sequence of x_k can be deterministically derived from a given sequence of u_k. Therefore, this discussion will focus on the evaluation of probability measures for u_k.

In particular, for binary inputs, $p(u_k \mid \tilde{y})$ comprises two components: $p(u_k = 0 \mid \tilde{y})$ and $p(u_k = 1 \mid \tilde{y})$. The evaluation of these values is simplified using a combination of Bayes' theorem and a different representation of the probabilities, known as the *likelihood ratio*. Bayes' theorem defines that

$$\begin{aligned} p(u_k = 1 \mid \tilde{y}) &= \frac{p(u_k = 1, \tilde{y})}{p(\tilde{y})} \\ p(u_k = 0 \mid \tilde{y}) &= \frac{p(u_k = 0, \tilde{y})}{p(\tilde{y})} \end{aligned} \tag{3.1}$$

In many cases, the only inputs available to the decoder are the observations, $\tilde{y}$. Without any a-priori knowledge of the source of user bits, u_k, it is impossible to determine the denominator, $p(\tilde{y})$, in 3.1. The likelihood ratio removes the necessity to compute this value by defining

$$\begin{aligned} \text{Likelihood ratio} &= \frac{p(u_k = 1 \mid \tilde{y})}{p(u_k = 0 \mid \tilde{y})} \\ &= \frac{p(u_k = 1, \tilde{y})}{p(u_k = 0, \tilde{y})} \end{aligned} \tag{3.2}$$

Hence the decoder is required to evaluate the joint probabilities, $p(u_k = 1, \tilde{y})$ and $p(u_k = 0, \tilde{y})$. These values are expressed as the sum of transition probabilities in 3.3. A transition between each pair of states, $(s_{k-1}=S_i,\ s_k=S_j)$, is possible only when there is a corresponding connection in the trellis. Each valid transition uniquely defines u_k and x_k.

$$
\begin{aligned}
p(u_k = 1, \tilde{y}) &= \sum_{S_i, S_j} p(u_k = 1, s_{k-1} = S_i, s_k = S_j, \tilde{y}) \\
p(u_k = 0, \tilde{y}) &= \sum_{S_i, S_j} p(u_k = 0, s_{k-1} = S_i, s_k = S_j, \tilde{y}) \\
&\qquad S_i, S_j \in \{S_1, S_2, S_3 \ldots S_{2^L}\}
\end{aligned}
\tag{3.3}
$$

Given a particular state, s_k, subsequent outputs of the encoder, $x_{k+1}, x_{k+2}, \ldots$, are independent of internal state of the encoder prior to step k. This observation provides the following expansion of the individual terms in (3.3). The symbol $y_k^{\,l}$ is defined as the set of received symbols between step k and step l, $\{y_k, y_{k+1}, y_{k+2}, \ldots, y_l\}$.

$$
\begin{aligned}
&p(u_k, s_{k-1} = S_i, s_k = S_j, \tilde{y}) \\
&= p(s_{k-1} = S_i, y_1^{\,k-1}) \cdot p(u_k, s_k = S_j, y_k^{\,N} \mid s_{k-1} = S_i) \\
&= p(s_{k-1} = S_i, y_1^{\,k-1}) \cdot p(u_k, s_k = S_j, y_k \mid s_{k-1} = S_i) \cdot p(y_{k+1}^{\,N} \mid s_k = S_j,) \\
&= \alpha_{k-1}(S_i) \cdot \gamma(u_k, S_i, y_k, S_j) \cdot \beta_k(S_j)
\end{aligned}
\tag{3.4}
$$

where α, β, and γ are defined as:

$$
\begin{aligned}
\alpha_k(S_i) &= p(s_k = S_i, y_1^{\,k}) \\
&= \sum_j \alpha_{k-1}(S_j) \cdot \gamma(u_{k-1}, S_j, y_{k-1}, S_i) \\
\\
\beta_k(S_j) &= p(y_{k+1}^{\,N} \mid s_k = S_j) \\
&= \sum_i \beta_{k+1}(S_i) \cdot \gamma(u_{k+1}, S_j, y_{k+1}, S_i) \\
\\
\gamma(u_k, S_j, y_k, S_i) &= p(u_k, s_k = S_i, y_k \mid s_{k-1} = S_j) \\
&= p[u_k = f(S_i, S_j)] \cdot p(y_k \mid x_k = g(S_i, S_j))
\end{aligned}
\tag{3.5}
$$

Each term is expanded as the product of three terms, $\alpha_{k-1}(S_i)$, $\beta_k(S_j)$, and $\gamma(u_k, S_j, y_k, S_i)$, which are explicitly defined in (3.5).The messages $\alpha_{k-1}(S_i)$, and $\beta_k(S_j)$ are computed through forward and backward iteration, and are commonly known as the forward and backward state metrics respectively. $\gamma(u_k, S_j, y_k, S_i)$ is a probability measure associated with the transition from state S_j to state S_i and the received symbol as a result of the transition, y_k. It is commonly known as the branch metric in relation to a branch in the trellis representation.

The above equations are computationally intensive due to the requirement to evaluate multiplicative terms. The algorithm is rearranged in the log-domain in order to ease the computational load. Details of these operations can be found in [102], but in general, the computations defined by BCJR algorithm fall into three categories: branch metric computation, forward/backward iteration, and combination of state metrics.

3.2.1.1.Branch metric computation

For each edge in the trellis connecting s_{k-1} with s_k, define a branch metric based on the a-priori information of u_k and the observed symbol, y_k. The values of u_k and x_k can be inferred from the starting and ending states, and are therefore expressed as functions, f and g respectively. The value of y_k is a sample of x_k in the presence of additive noise. As such, $P[y_k \mid x_k)]$ depends on the probability distribution of the noise source.

$$\begin{aligned}\gamma_k(s_{k-1},s_k) = \ln\{P(u_k &= f(s_{k-1},s_k)\} \\ &+ \ln\{P[y_k \mid x_k = g(s_{k-1},s_k)]\}\end{aligned} \tag{3.6}$$

3.2.1.2.Forward/backward iteration

For each node in the trellis, define a pair of forward/backward iterating state metrics as shown in (3.7) and (3.8). Each state metric is dependent on the values of the preceding/succeeding state metrics that are adjacent to the node.

$$\alpha_k(s_k) = \ln\left\{\begin{matrix}\exp[\alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1},s_k)] + \\ \exp[\alpha_{k-1}(s'_{k-1}) + \gamma_k(s'_{k-1},s_k)]\end{matrix}\right\} \tag{3.7}$$

$$\beta_k(s_k) = \ln\left\{\begin{matrix}\exp[\beta_{k+1}(s_{k+1}) + \gamma_{k+1}(s_k,s_{k+1})] + \\ \exp[\beta_{k+1}(s'_{k+1}) + \gamma_{k+1}(s_k,s'_{k+1},)]\end{matrix}\right\} \tag{3.8}$$

3.2.1.3. Combination of state metrics

Working in log-likelihood domain, this last step involves the combination of appropriate forward and backward state metrics to evaluate the log-likelihood of the encoder inputs $\ln\left[\frac{P(u_k=1\mid y)}{P(u_k=0\mid y)}\right]$, and outputs, $\ln\left[\frac{P(x_k=1\mid y)}{P(x_k=0\mid y)}\right]$.

$$\begin{aligned}
&\ln\left[\frac{P(u_k=1\mid \tilde{y})}{P(u_k=0\mid \tilde{y})}\right] \\
&=\ln\left\{\sum_{S_i,S_j \ni u_k=f(S_i,S_j)=1} \exp\left[\alpha_{k-1}(S_i)+\gamma\left(u_k=f(S_i,S_j),S_i,y_k,S_j\right)+\beta_k(S_j)\right]\right\} \qquad (3.9)\\
&\quad -\ln\left\{\sum_{S_i,S_j \ni u_k=f(S_i,S_j)=0} \exp\left[\alpha_{k-1}(S_i)+\gamma\left(u_k=f(S_i,S_j),S_i,y_k,S_j\right)+\beta_k(S_j)\right]\right\}
\end{aligned}$$

$$\begin{aligned}
&\ln\left[\frac{P(x_k=1\mid \tilde{y})}{P(x_k=0\mid \tilde{y})}\right] \\
&=\ln\left\{\sum_{S_i,S_j \ni x_k=g(S_i,S_j)=1} \exp\left[\alpha_{k-1}(S_i)+\gamma\left(x_k=g(S_i,S_j),S_i,y_k,S_j\right)+\beta_k(S_j)\right]\right\} \qquad (3.10)\\
&\quad -\ln\left\{\sum_{S_i,S_j \ni x_k=g(S_i,S_j)=0} \exp\left[\alpha_{k-1}(S_i)+\gamma\left(x_k=g(S_i,S_j),S_i,y_k,S_j\right)+\beta_k(S_j)\right]\right\}
\end{aligned}$$

In summary, the MAP decoder calculates the conditional probabilities of the transmitted bits by observing a long sequence of received symbols. The memory effect of convolutional codes is reproduced by propagating the state metrics in both forward and backward directions through the trellis. The algorithm is simplified by the use of log-probability and defining a log-likelihood representation. Despite this, the MAP decoder remains significantly more complex than the Viterbi algorithm [3], which has been the choice decoder for non-concatenated convolutional codes in the past. However, the significance of the MAP decoder is brought to light in the context of turbo decoders, which require soft outputs. In the next section, the Viterbi algorithm is adapted to provide such soft outputs for use with turbo decoders.

3.2.2. Soft-output Viterbi algorithm (SOVA)

The Viterbi algorithm searches for the maximum-likelihood decision, $\hat{x}$, by finding the most-likely (*ML*) path through a trellis representation. In a binary channel, an error in the decoding implies that the correct output is the complementary decision $\bar{\hat{x}}$.

Based on this observation, the soft-output Viterbi algorithm (SOVA) searches for the two most-likely (ML) paths that trace back to complementary bit decisions, $\hat{x}$ and $\bar{\hat{x}}$. The difference in path metric between these two paths is representative of the confidence of the decoded bit.

The path metrics are determined by an iterative process that is used in all Viterbi decoders. The derivation of the procedure is very similar to that described for MAP decoders in Section 3.2.1. Branch metrics are computed in the same manner as in (3.6). The state metric corresponding to state s_k at instance k is represented by $sm(s_k)$. There is no backward iteration, and the forward iteration is based on the following recursion.

$$sm(s_k) = Max\{\exp[sm(s_{k-1}) + \gamma_k(s_{k-1}, s_k)], \exp[sm(s'_{k-1}) + \gamma_k(s'_{k-1}, s_k)]\} \quad (3.11)$$

Similarities between (3.7) and (3.11) can be observed. The log-of-sum function in (3.7) is approximated with a maximization function, $Max(\cdot)$. The maximum selection also determines a particular branch at the corresponding node in the trellis. After some latency, these decisions are read back recursively in a process known as a traceback to determine the most-likely path, α.

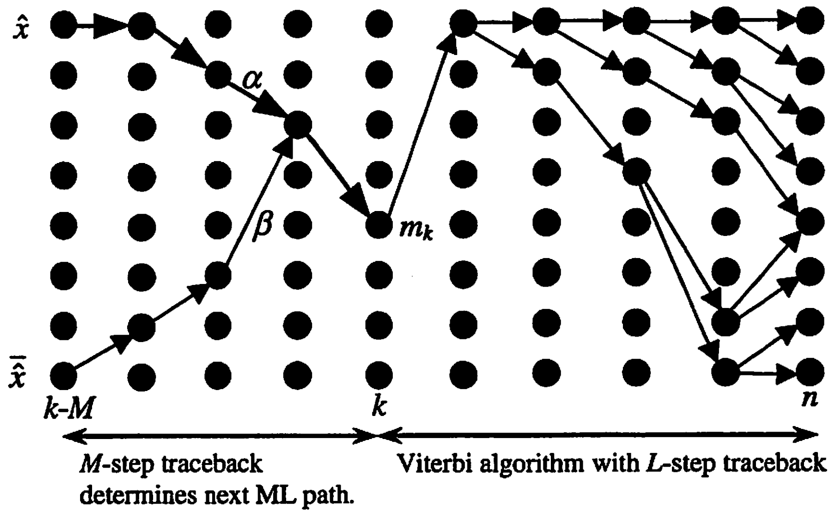

$\hat{x}$
$\bar{\hat{x}}$
α
β
m_k
k-M
k
n
M-step traceback determines next ML path.
Viterbi algorithm with L-step traceback

Figure 3-6. Two-stage traceback in a SOVA decoder to determine the two ML paths, α and β.

With high probability, the next most-likely traceback path, β, will coincide with α for some number of steps before a branch occurs. In order to determine β, each node along α is evaluated for possible branching. Only branches that lead to a complementary bit decision are considered valid candidates for β. The difference in path metric between each valid branch and α, observed at the node where branching occurs, is compared. The next most-likely path is determined by the branch which corresponds to the minimum difference.

It is further assumed that the absolute values of the path metrics, SM_α and SM_β, dominate over that of other paths. The probability of selecting β over α (i.e. the wrong decision) is given by (3.12). The log-likelihood of a correct output by the SOVA decoder is then given by (3.13).

$$\begin{aligned} P_{err} &= \frac{\exp(-M_\beta)}{\exp(-M_\alpha)+\exp(-M_\beta)} \\ &= \frac{1}{1+\exp(\Delta)} \quad ; \quad \Delta = M_\beta - M_\alpha \end{aligned} \tag{3.12}$$

$$\log\left[P\left(\frac{CorrectDecision}{WrongDecision}\right)\right]=\log\left(\frac{1-P_{err}}{P_{err}}\right) \\ =\Delta=M_{\beta}-M_{\alpha} \tag{3.13}$$

Figure 3-6 shows that the *ML* path, α, is determined using the Viterbi algorithm with an *L*-step traceback. This is followed by another *M*-step traceback that resolves the next ML path, β, based on maximal probability of a deviation from α.

3.2.3. Message passing algorithm used in LDPC decoder

As briefly introduced, LDPC decoders implement the message-passing algorithm by associating each node and edge in the underlying bipartite graph (Figure 3-2) with a PE and interconnect, respectively. The outputs from each PE are sent as messages to other adjacent PEs. An iteration of LDPC decoding consists of a round of message computation at all variable node PEs, followed another round of message computation at all check node PEs.

An alternative mathematical definition of an LDPC code uses a parity check matrix, ***H***. Each valid codeword $\tilde{x}=[x_1,x_2,...,x_N]^T$ satisfies the definition in (3.14) under GF(2) computations.

$$H\cdot\tilde{x}=\tilde{0} \tag{3.14}$$

An *N*×*M* parity check matrix defines a code with *N* bits (variable nodes) and *M* parity checks (check nodes). Hence, each LDPC code comprises *M* different parity check constraints. Each column in the matrix represents a particular bit, while each row represents a parity checksum. The entries in the matrix are binary; a non-zero (i.e. '1') entry at row *i* and column *j* of the matrix indicates that the j^{th} bit is a member of the i^{th} parity checksum. In most LDPC codes, each bit is a member of four or five parity checksums, thus forming the intersecting constraints discussed in Chapter 1.

For the same reasons as discussed in earlier sections, the LDPC decoder of interest operates in the log-probability domain. The inputs to an N-bit LDPC decoder are the log-likelihood ratios of x_n, as defined in (3.15).

$$\alpha_n = \ln\left[\frac{\Pr(x_n = 1)}{\Pr(x_n = 0)}\right], n{=}1, 2, \ldots, N \tag{3.15}$$

Let $\nu(m) = \{n\text{: } \boldsymbol{H}_{m,n} = 1\}$ be the set of variable nodes that are connected to check node m, and $\mu(n) = \{m\text{: } \boldsymbol{H}_{m,n} = 1\}$ be the set of check nodes that are connected to variable node n. Q_{nm} and R_{mn} refer to the messages that are passed between variable n and check m as defined in (3.16) and (3.17) respectively.

Message from variable n to check m:

$$Q_{nm} = \alpha_n + \sum_{m \in \mu(n) \backslash m} R_{m'n} \tag{3.16}$$

Message from check m to variable n:

$$R_{mn} = \Phi^{-1}\left(\sum_{n \in N(m) \backslash n} \Phi(Q_{n'm})\right) \times \left(\prod_{n \in \nu(m) \backslash n} \operatorname{sgn}(Q_{n'm})\right) \cdot (-1)^{|\nu(m)|} \tag{3.17}$$

$$\Phi(x) = -\log\left(\tanh\left(\frac{1}{2}x\right)\right) = \Phi^{-1}(x); \qquad x \geq 0 \tag{3.18}$$

$$\begin{aligned} \Phi(x) &= -\log\left(\tanh\left(\tfrac{1}{2}x\right)\right) \\ \Phi^{-1}(x) &= 2\tanh^{-1}[\exp(-x)] \end{aligned} \tag{3.19}$$

The decoder begins by initializing the first round of variable-to-check messages with the input log-likelihood ratios of the corresponding variable nodes. The operation at each bit node is conceptually straightforward. The output messages are obtained by summing all input messages together with the prior information.

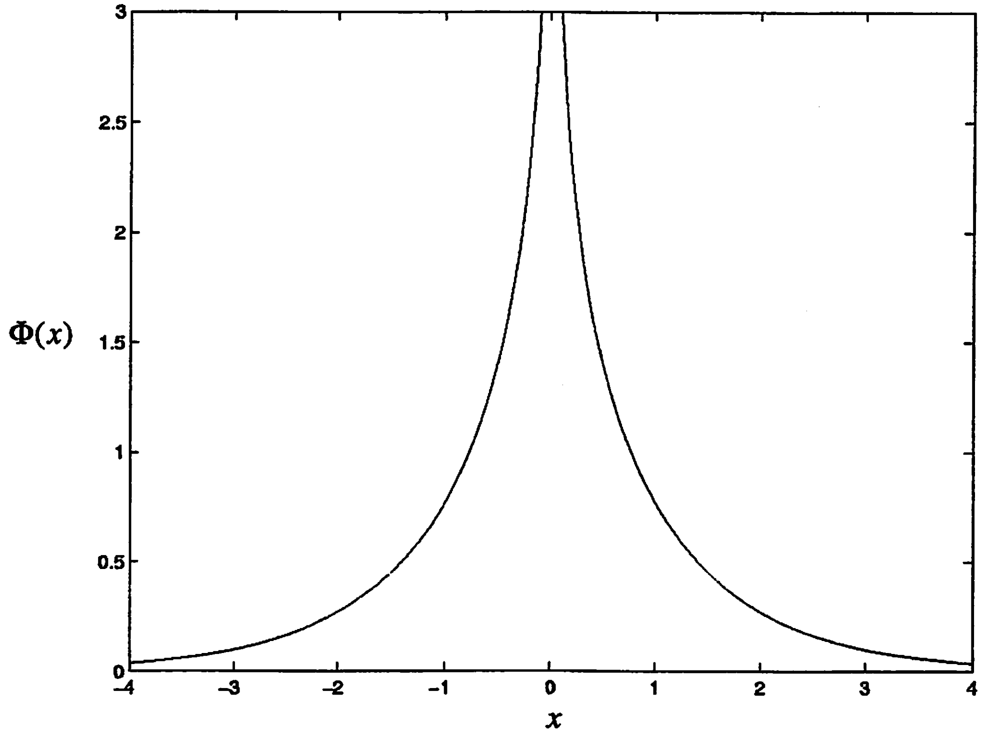


Figure 3-7. Plot of $\Phi(x)$ against x.

Each check node, m, receives a number of messages, R_{nm}, from a pre-determined set of variable nodes. The calculations shown in (3.17) can be partitioned into three segments. Most codes are constructed on the basis of even parity, which implies that $(-1)^{|\nu(m)|} = 1$. The checksum is obtained by evaluating the product of the signum of input messages, $\prod_{n \in \nu(m) \backslash n} \operatorname{sgn}(Q_{n'm})$. Finally, the absolute value of the output is determined by summing the results of applying the $\Phi(\cdot)$ function on the input messages, followed by the inverse function, $\Phi^{-1}(\cdot)$.

Figure 3-7 shows a plot of $\Phi(x)$ against. x. The inverse function, $\Phi^{-1}(\cdot)$, can be easily verified to be equal to $\Phi(\cdot)$. The function has a reciprocal effect on its operands. A detailed derivation of the function is beyond the scope of this text. However, an intuitive explanation is offered as follows.

In evaluating the parity checksum of a group of bits, the confidence of the output is strongly affected by the input bit with the least confidence. This is because flipping the outcome of a single bit changes the checksum value. Consider a log-likelihood ratio with a minimum absolute value amongst a group of messages, Q_{nm}. Applying $\Phi(\cdot)$ on this input results in a large output value, which dominates the sum, $\sum_{n \in N(m)\backslash n} \Phi(Q_{n'm})$. Finally, the inverse function is applied to this sum, and the message from the check node PE shows a weak confidence, which reflects the small value of the input message.

In the rest of this chapter, an overall view of the operational requirements of the SISO decoders is presented.

3.3. Requirements of iterative decoders

The requirements of SISO decoders can be mostly classified into two categories, computational and message passing requirements. This final section distinguishes several SISO decoder algorithms in terms of these requirements. This will permit the next two chapters, which describe specific architectures for implementations of the SISO decoders, to describe the work in the context of targeted requirements.

3.3.1. Computational requirements

The key arithmetic computation in any iterative decoder is the evaluation of marginal posterior functions. Each node in the graph is associated with a processing element that evaluates the marginal function of its input messages. Let X and Y be two random variables jointly defined such that

$$\sum_x \sum_y p(X = x, Y = y) = 1 \tag{3.20}$$

The marginalized functions are described by

$$P(X = x) = \sum_y p(X = x, Y = y) \tag{3.21}$$

$$P(Y = y) = \sum_x p(X = x, y = y) \tag{3.22}$$

The joint probabilities summed on the right-hand side of (3.21) and (3.22) are evaluated by the product of probabilities of independent events. Such computations are sometimes referred to as the sum-product algorithm. Although the decoding of iterative codes is derived from the sum-product algorithm, expressing the equations as sum of log-probabilities or log-likelihood ratios is preferred in implementation because it replaces the required multipliers with adders. However, the sum of probabilities is transformed into a complex combination of exponentials and logarithmic functions. To simplify the hardware, the computation is approximated with the maximum value of the input operands, followed by an additive correction factor, which is determined through a table lookup.

An example of the sum-product algorithm processed in log-probability domain is the Add-Compare-Select-Add (ACSA) recursion in a maximum a-posteriori (MAP) decoder (Figure 3-8a). The "add" operation simplifies what would otherwise be two product terms if the decoder was implemented in the probability domain. The "compare" and "select" operations approximate the logarithm of a sum of exponentials. This approximation leads to an implementation loss of about 0.5dB in a turbo system. However, adding a correction factor to the output of the ACS can bring the performance back within 0.1dB of the performance with a MAP decoder [55]. This correction factor is based on the weights of the difference of the two sums. The throughputs of MAP decoders are limited by the implementation of the add-compare-select (ACS) structure due to the single-step recursion that prevents pipelining. This is similar to the more commonly used Viterbi decoders.

Another example of the sum-product computation in log-probability domain can be found in LDPC decoders (Figure 3-8b). The check-to-variable message computation needs to evaluates $\prod_{n}(1-2p_n)$, where p_n represents the probability that a bit x_n equals to 1. Performing the same computations in the log-probability domain simplifies the evaluation of the product, but also requires the implementation of $\Phi(x) = -\log\left(\tanh\left(\frac{x}{2}\right)\right)$. This non-linear, monotonic complex function is not easily implemented with full precision, requiring the use of CORDIC functions [113] that can be significantly more costly than the adders used in summation. Fortunately, iterative decoders require low fixed-point precisions, frequently just three to five bits. This implies that $\Phi(x)$ can be evaluated using lookup tables that are efficiently implemented with simple combinatorial logic functions or small ROM-based lookups.

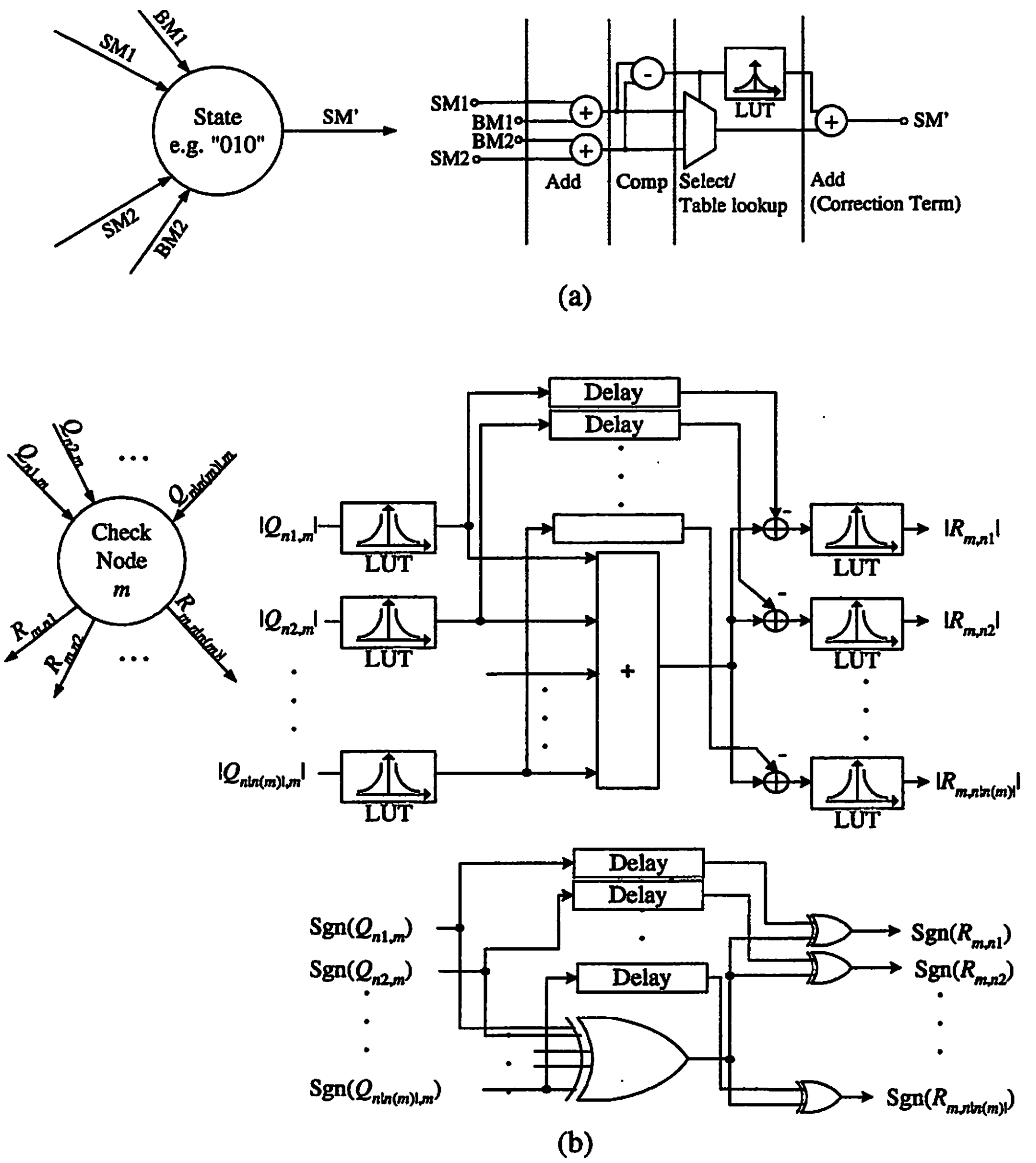


Figure 3-8. Arithmetic computation associated with nodes in factor graphs of (a) a convolutional code, and (b) an LDPC code.

3.3.2. Message-passing requirements for iterative codes

The edges shown in the bipartite graph representations (Figure 3-1 or Figure 3-2) of iterative decoders correspond to a network of interconnects. These wires implement a message-passing network by facilitating the exchange of messages between nodes in a factor graph.

The properties of these connections affect both the performance of the code as well as the practical implementation of the message-passing network. In general, good codes exhibit codewords with large minimum distance and a small number of low-weight codewords. These characteristics are found in graphs with large expansibility, girth, and absence of short cycles. These graphs tend to have a disorganized structure, which complicates the implementation of the message-passing network by requiring long global interconnects or memories accessed through an unstructured pattern.

3.3.2.1. Interleavers

Although the SISO decoders used with turbo codes take advantage of the well-structured linear construction of convolutional codes, the required interleavers, which separate the SISO decoders, will destroy the organized structure. Good interleavers are known to break low weight input sequences and increase free Hamming distance of the code or reduce number of codewords with small distances in the code distance spectrum. Much of this depends on the ability of the interleaver output to appear random with respect to the input sequence.

Although a high throughput interleaver can be realized through a direct-mapped network of interconnects, this will potentially result in routing congestion due to the sparseness of the interleaving network. In practice, the interleaving function is executed by writing the relayed messages sequentially into random access memory, and output by reading through a permuted sequence. The order of addresses used in the read-access can be stored in a separate read-only memory (ROM). However, storing the output sequence in a ROM results in overhead that can be greater than the actual amount of memory actually required for the soft outputs only. For example, a block code of 4096 bits will require 12-bit addresses. This contrasts with the 5 to 7 bits that are typical of the soft outputs permuted by the interleaver. The choice of interleaver is essentially a search space of $N!$. In the past, a number of different algorithms performing this search have been proposed. These algorithms lead to interleavers that are either deterministic or non-deterministic.

Non-deterministic interleavers require the output sequence to be stored in a ROM. Common examples of such interleavers are the random interleaver and the S-random interleaver [105]. The latter is often used as a benchmark for interleaver designs. S-random interleavers employ an iterative process that randomly selects numbers and compares them to ensure a minimum spreading effect. In general, indices that are less than s_1 apart in an input sequence must be mapped to indices that are further than s_2 apart in the output sequence.

Deterministic interleavers are of more interest to the implementation of turbo decoder hardware. This class of interleavers include pseudo-random interleavers realized with shift registers, congruential interleavers [98] [107] and golden interleavers [106]. An example of a pseudo-random interleaver based on three single-bit registers or flip-flops is shown in Figure 3-9. Given a non-zero initial value, the binary contents in the registers will form a 3-bit address, whose value will cycle between 1 and 7. Such interleavers are always limited to a period of (2^k-1) cycles, and are often not applicable for turbo codes with arbitrary block sizes.

Congruential interleavers, by far, are the most commonly implemented. They can be realized using simple shifting or modulo division [98]. The output sequence $\pi(i)$ is defined by

$$\pi(i) = (ip + s) \bmod N \tag{3.23}$$

where the value of p is relatively prime with respect to N, and s represents an integer chosen by heuristics. The basic example of a congruential block interleaver writes the inputs row-wise into a memory array, and reads the outputs column-wise. The sequential write/read pattern along rows/columns allows the memory access operations of this interleaver to make use of cycle counters to activate both word (row) lines and bit (column) lines, thereby eliminating the necessity to perform memory-address decoding.

More complex interleavers are often derived from congruential interleavers. The PIL interleaver that is currently adopted for the UMTS-3GPP channel-coding standard is defined with a two-dimensional interleaving table containing the output addresses. The entries in the table are accessed through a pre-determined row permutation sequence, and a column permutation based on a modulo function.

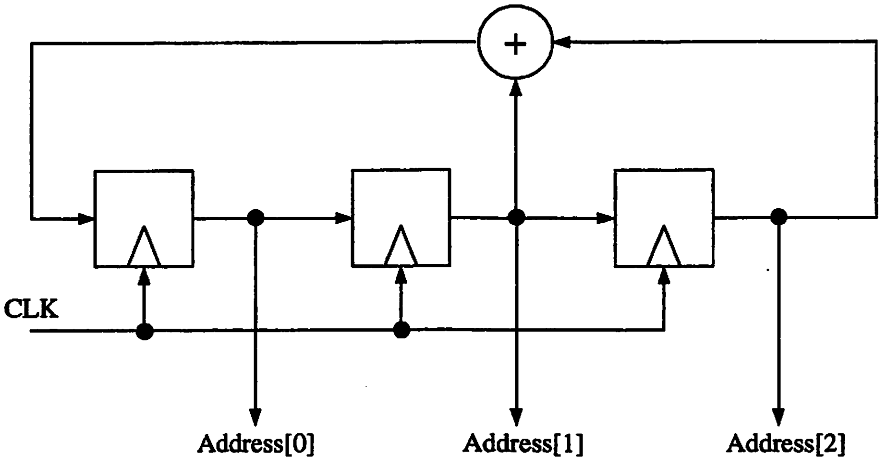


Figure 3-9. Pseudo random interleaver realized with three single-bit registers.

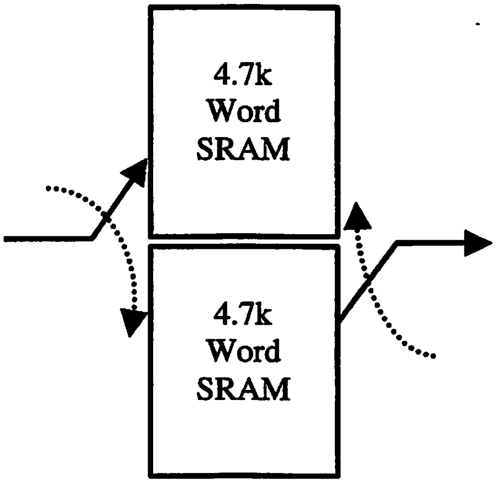


Figure 3-10. Interleavers and deinterleavers implemented using alternating read/write buffers.

Finally, golden interleavers are based on the golden section value g=0.618 that satisfies

$$\frac{1-g}{g}=g \tag{3.24}$$

This algorithm has good distance properties, but requires multiplicative and sorting operations, which are computationally much more intensive compared to the congruential interleavers. Hence they have not been very widely used.

The practice of writing the messages into an array of SRAM, and reading them out through a permuted sequence usually requires two banks of buffers alternating between read/write for consecutive blocks of data (Figure 3-10). This is due to the randomness of the interleaver output sequence, which makes it difficult to realize in-place storage. Although a multitude of interleaver designs exists, practical considerations are likely to determine the type of interleaver selected for implementation with a turbo decoder.

3.3.2.2. LDPC interconnect

Likewise, an LDPC decoder is required to provide a network for connectivity between a large number of variable nodes and check nodes. A direct mapping of the network using hard-wired routes can lead to congestion in the interconnect fabric because the LDPC graphs typically have unstructured factor graphs. As in the turbo decoder, the congestion can be circumvented through the use of memory at the cost of large memory requirement and placing the memory access in the critical path of the decoder.

More recently however, LDPC codes employing graphs with structured patterns have emerged, and they impose a reduced set of constraints on the implementation of the decoders. A few examples of these codes and the implementation advantages that they provide will also be described in the description of architectures for LDPC decoders.

3.4. Summary

The chapter has described the message passing algorithms implemented by BCJR, SOVA and LDPC decoders. It has also highlighted some of the difficulties associated with implementation of the computational and message passing requirements. In the next two chapters, suitable architectures will be presented for each of these decoders.

4. ARCHITECTURES OF SISO DECODERS FOR TURBO CODES

The efficient mapping of algorithms into architectures is presented. The system level implications of a turbo decoder are discussed in this chapter. This is followed by detailed analysis of the SISO decoders and interleavers that realize the building blocks. The successful implementation of a decoder is dependent on application constraints that determine the target throughputs, power, and area of implementation. The choice of direct-mapped architectures for iterative decoding algorithms affects both computational and message-passing requirements as described in the previous chapter.

4.1. System level considerations

In general, communication systems break their processing into a sequence of operations. An example is a generic receiver, shown in Figure 4-1. The signal received at the antenna is passed through a low-pass filter (LPF) and immediately digitized by an analog-digital converter (ADC). Beyond this, equalization, error correction and detection are performed in the digital domain.

Traditional receivers implement different operations at line rates to ensure a constant flow of data. However, the implementation of block-based iterative codes will depart from this practice. At the output of the channel equalizer, sufficient memory will have to be allocated to store several frames of received symbols. This allows the block-based symbol interleaving/deinterleaving in turbo codes, or message passing in LDPC codes to be carried out.

Increased memory requirement, iterative decoders translates into large latencies. The minimum latency is the delay required to transmit one block of data. With block sizes that start from a few hundred bits and multiple decoding iterations, the resulting latencies are a few orders of magnitude longer than that of traditional decoders.

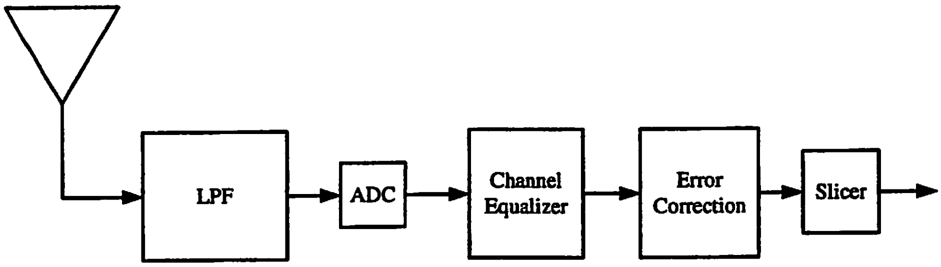


Figure 4-1. Pipelined blocks in a generic communications receiver.

Current technology permits systems with throughputs at up to a few Mb/s to be clocked at a multiple of the symbol frequency. For example, a 2Mb/s MAP decoder is clocked at 88MHz such that more than ten decoding iterations can be completed within an additional latency equivalent to one block period [98]. On the other hand, applications such as magnetic storage require throughputs close to 1Gb/s. This high throughput, coupled with the bottlenecks imposed by the ACS recursions in MAP and SOVA decoders implies that the turbo decoders will need to operate at line rates. SISO decoders operated at line rates result in additional latencies of one block delay per decoding iteration.

The extended latencies are also intolerable for traditional decision-directed timing recovery techniques. These methods employ adaptive channel equalization filter, which relies on the decisions from the detector. In order be effective, the latency through the receiver chain must be kept to a minimum. This issue has not been widely researched, and remains a potential barrier to the successful shift towards use of iterative codes. Nonetheless, timing recovery is beyond the scope of this work, and perfect timing recovery has been assumed where necessary.

4.2. MAP decoder

The MAP algorithm defines the operations corresponding to a vertical slice of the trellis, as shown in Figure 4-2. The architectures for implementation of these recursive structures are discussed. This will provide estimates of the sustainable throughputs that will eventually affect the decision towards realization of other non-timing-critical elements, such as the memory and the branch metric generator blocks.

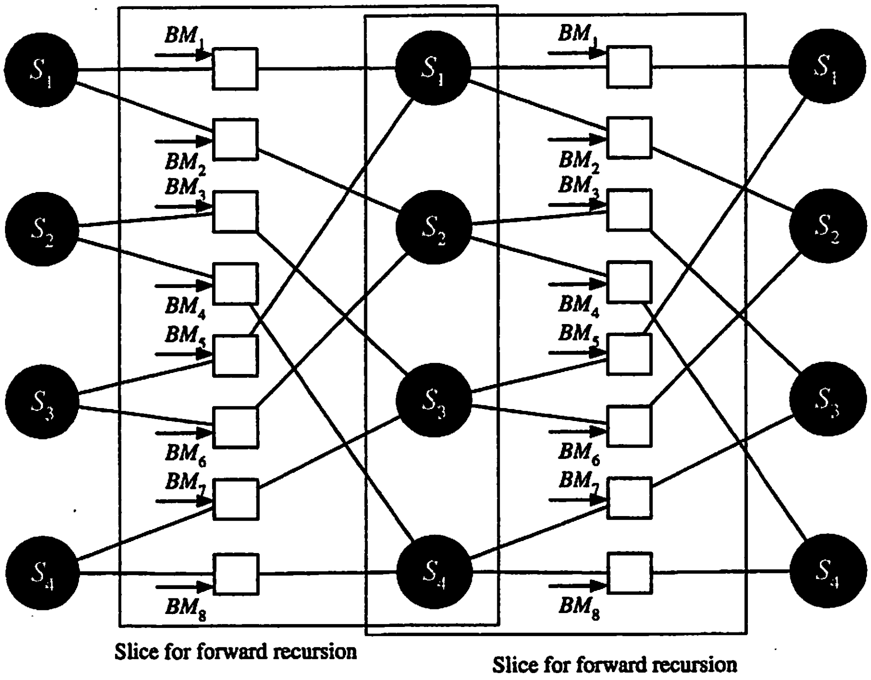


Figure 4-2. MAP algorithm defines operations on a vertical slice of trellis.

4.2.1. Forward and backward recursion

Both forward (3.7) and backward (3.8) recursions make use of similar structures. At time-instance k, the current branch metrics (γ_k) are added to the corresponding state metrics (α_k) from the previous iteration at (k-1):

$$\begin{aligned} A_0 &= \alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k) \\ A_1 &= \alpha_{k-1}(s'_{k-1}) + \gamma_k(s'_{k-1}, s_k) \end{aligned} \tag{4.1}$$

The logarithm of the sum of exponentials is evaluated with a Ψ-operator:

$$\begin{aligned} \Psi(A_0, A_1) &= \ln\left(e^{A_0} + e^{A_1}\right) \\ &= \max\{A_0, A_1\} + \ln\left\{1 + e^{-|A_0 - A_1|}\right\} \end{aligned} \tag{4.2}$$

The second term in (4.2) can be approximated with a lookup table [55]. The lookup table function is strictly monotonic. A 32-entry table can be efficiently implemented as combinatorial logic in less than 0.05mm^2 in $0.13\mu\text{m}$ CMOS process. Figure 4-3 depicts a structure that evaluates this sequence of operations. The comparison is implemented through subtraction and the most-significant bit (MSB) of the difference, which is the sign bit, selects the maximum value. The recursion elements used in MAP decoders are named as *add-compare-select-add* (*ACSA*) units after the sequence of operations required in each iteration or symbol period. This term is intentionally similar to the use of *add-compare-select* (*ACS*) in Viterbi decoders and reflects the similarity in path recursion as applied towards trellis-based convolutional codes, as well as the throughput bottleneck it poses towards decoder implementations.

Most of the delay penalty of the final add in the *ACSA* can be removed by retiming. Figure 4-4 shows an *add-add-compare-select* (*AACS*), which has shifted the final addition in Figure 4-3 to the head of the recursion. The *AACS* structure naturally favors ripple-carry adders because the carry profile of each adder follows that of the preceding adder. Assuming B-bit fixed-point representations of the recurring state metrics,

$$\text{Total delay of all additions} = (B+1+1) \times \text{Delay}_{\text{Full Adder}} \tag{4.3}$$

The delay penalty of the additive LUT term is reduced to the delay of a single-bit full-adder. In 0.13 μm CMOS technology, this delay corresponds to approximately 100ps. With 9-bit representations, the delay overhead is 10%.

Further arithmetic optimization of the *AACS* structure yields small gains. Accelerating the additions by using, for instance, a carry-select makes it difficult for each adder to follow the carry profile, resulting in little performance improvement at the expense of large area penalties.

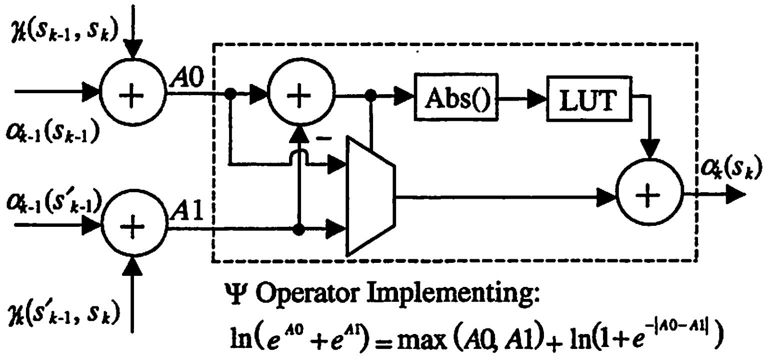


Figure 4-3. Add-Compare-Select-Add unit for either forward or backward recursions using the Ψ(.) operator as indicated within the box.

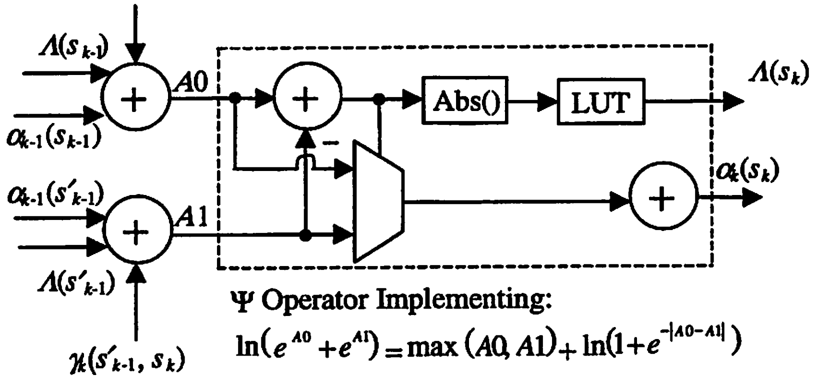


Figure 4-4. Retimed ACSA that hides penalty of final addition.

The implementation of the *AACS* can take advantage of previously published Viterbi decoder add-compare-select structures. The state metrics (either α_k or β_k) are represented with *sm*, while the branch metrics (γ_k) are represented with *bm*. The change of notation helps to highlight the parallels between an ACS used in a Viterbi decoder, and the AACS used in a MAP decoder.

Previous high throughput implementations of the Viterbi decoder, [6] [51] [53], unrolled the ACS loop in order to perform two-step iterations of the trellis recursions within a single clock period. These lookahead methods replace the original radix-2 trellis (Figure 4-5) with a radix-4 trellis (Figure 4-6) at the cost of increased interconnect and computational complexity.

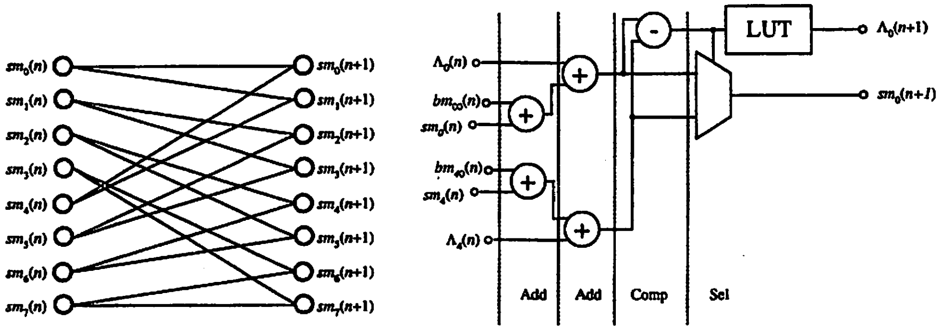


Figure 4-5. Radix-2 trellis and AACS structure.

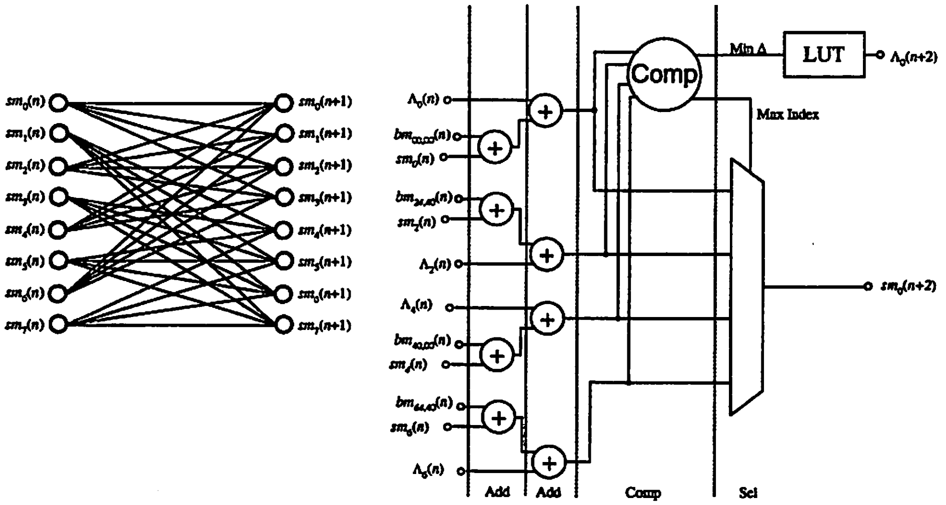


Figure 4-6. Radix-4 trellis and AACS structure.

A radix-4 AACS computes four sums in parallel followed by a four-way comparison. In order to minimize the critical-path delay, the comparison is realized using six parallel pair-wise subtractions of the four output sums. In addition, the comparator has to output the minimum difference that separates any pair of inputs. This value is used as input to the lookup table, which provides the correction term. Furthermore, the two-level addition is replaced with faster 3-input adders. In general, the overall critical-path delay increases. However, due to the doubled symbol rate, the effective throughput is improved if this increase is less than two-fold.

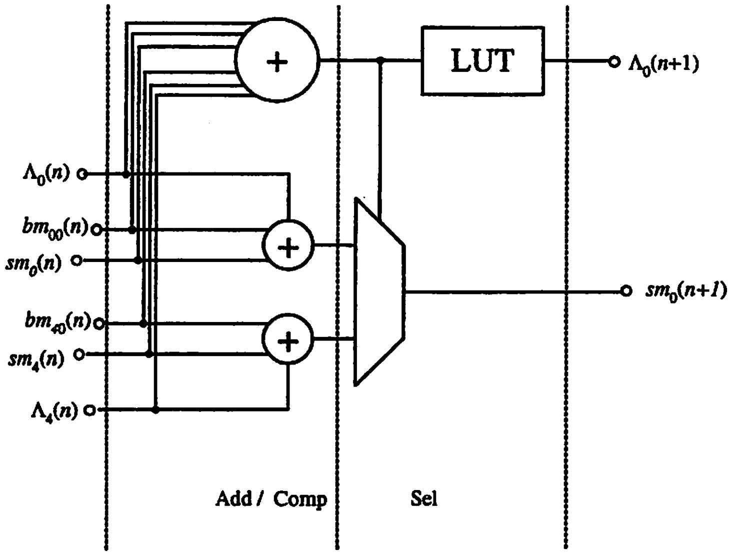


Figure 4-7. Radix-2 concurrent *AACS* structure.

An alternative approach with a lower area overhead is the concurrent ACS [65] that was proposed for a Viterbi decoder. The concept maintains the use of a radix-2 trellis, but performs the addition and comparison operations in parallel. It requires the comparison to be realized with a four-input adder using carry-lookahead structures [103]. A sub-8ns four-input adder was implemented in 0.6μm CMOS using two layers of three-to-two carry-save adders, followed by a final carry-lookahead adder. However, when applied to MAP decoding, a concurrent AACS (Figure 4-7) will require the parallel execution one 6-input adder and two 3-input adders. The critical path through the six-input adder and a multiplexer determines the throughput of the concurrent AACS.

Finally, an architecture, obtained through further retiming and transformation of the *AACS* unit, [34] [80], has a critical path comprising a 3-input adder and a multiplexer. The sequence of operations are reordered as a comparison between the two sums, followed by selection of the appropriate maximum value, and finally, addition of the two pairs of corresponding branch metrics and lookup table outputs. The resulting structure has been labeled as a Compare-Select-Add-Add (*CSAA*) unit. The reordering yields no performance gain: the subtraction no longer follows the addition and the carry profile is flattened by the multiplexer. The complete delays of the additions and subtraction appear in the critical path.

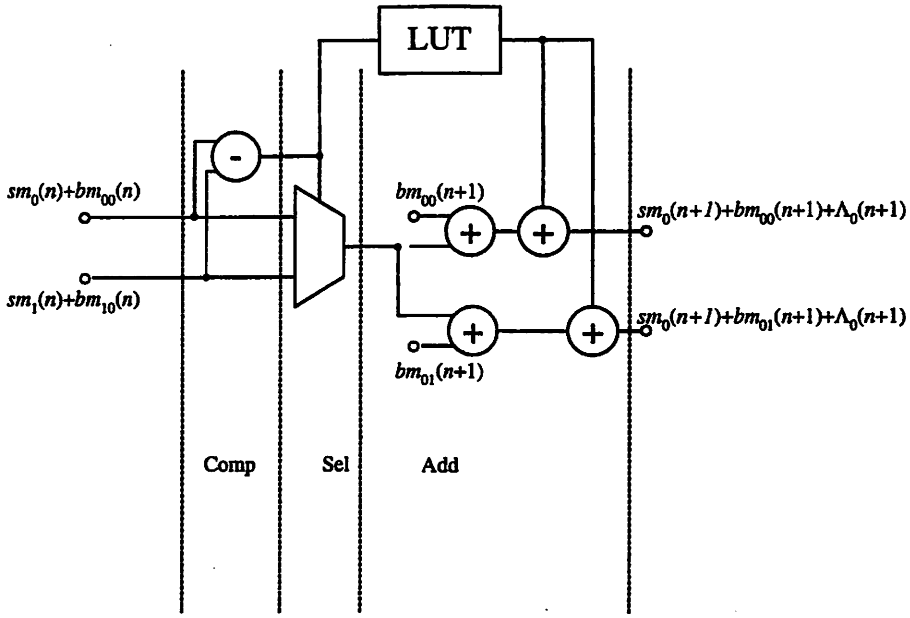


Figure 4-8. Radix-2 compare-select-add-add (*CSAA*) structure

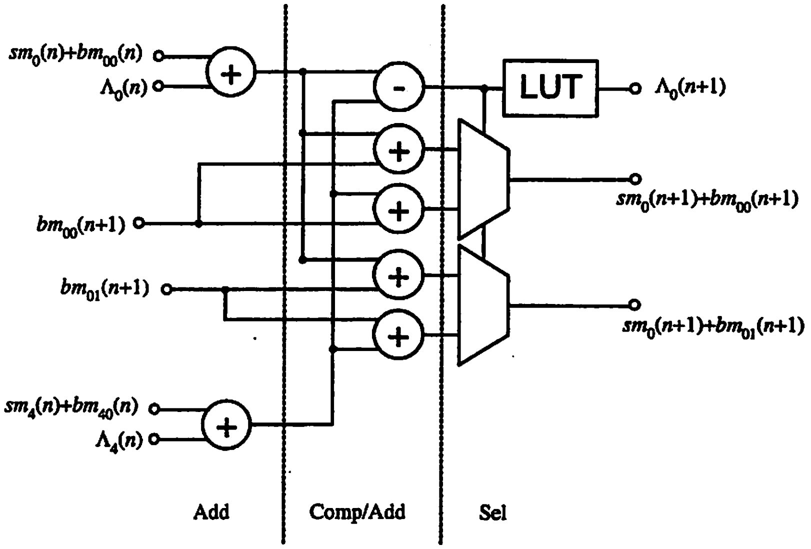


Figure 4-9. Transformed add-compare-select-add (*ACSA*) structure.

This delay can be hidden by moving the add operations before the select operation, as shown in Figure 4-9. The resulting structure executes both the compare and adds in parallel. This modification decreases the critical path delay at the cost of doubled number of adders and multiplexers. This structure is labeled as a transformed add-compare-select-add (*ACSA*). Compared with the concurrent *AACS*, the transformed *ACSA* enjoys lower overall complexity and a shorter critical path, which consists of a 3-input adder and a multiplexer.

The use of redundant numbering system with MSB-first computations has also been previously explored as an option to enhance the throughput of Viterbi decoders [6]. These methods achieve improvement in performance at the expense of large area due to the cost of carry-save representation. In section 4.3.2, further comparative analysis of the various *ACS* structures under varying delay constraints will be described.

4.2.2. Combining forward / backward state metrics

In the previous chapter, the final step in obtaining the confidence measures from a MAP decoder is evaluated in a ξ block. The path metrics obtained from both forward and backward recursions are combined. The definition in (3.9) is rearranged to make use of the Ψ-operators described by (4.2).

$$\begin{aligned}
&\ln\left\{\sum_{S_i,S_j \ni\, x_k=g(S_i,S_j)=1} \exp\left[\alpha_{k-1}(S_i)+\gamma\left(u_k=f\left(S_i,S_j\right),S_i,y_k,S_j\right)+\beta_k\left(S_j\right)\right]\right\}- \\
&\qquad \ln\left\{\sum_{S_i,S_j \ni\, x_k=g(S_i,S_j)=0} \exp\left[\alpha_{k-1}(S_i)+\gamma\left(u_k=f\left(S_i,S_j\right),S_i,y_k,S_j\right)+\beta_k\left(S_j\right)\right]\right\} \qquad (4.4) \\
&=\ln\left\{\sum_{S_j \ni\, x_k=g(S_j)=1} \exp\left[\alpha_k\left(S_j\right)+\beta_k\left(S_j\right)\right]\right\}-\ln\left\{\sum_{S_i \ni\, x_k=g(S_i)=1} \exp\left[\alpha_k(S_i)+\beta_k(S_i)\right]\right\}
\end{aligned}$$

Let

$$j=[1..8]\,\text{s.t.}\,S_j \ni x_k=g(S_j)=1$$

$$i=[9..16]\,\text{s.t.}\,S_i \ni x_k=g(S_i)=1$$

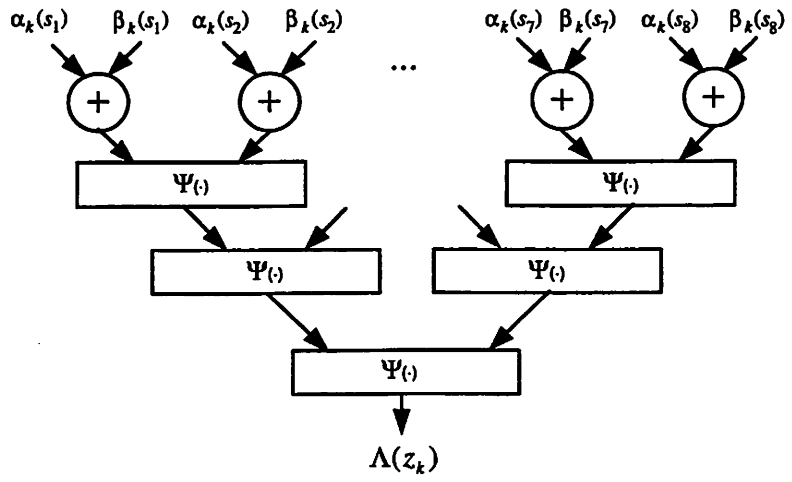


Figure 4-10. ξ block makes use of a binary tree of $\Psi(.)$ operators.

$$\begin{aligned}
&\ln\left\{\sum_{j=[1..8]}\exp\left[\alpha_k(s_j)+\beta_k(s_j)\right]\right\}\\
&=\ln\left\{\exp\left[\ln\left\{\sum_{j=[1..4]}\exp\left[\alpha_k(s_j)+\beta_k(s_j)\right]\right\}\right]+\exp\left[\ln\left\{\sum_{j=[5..8]}\exp\left[\alpha_k(s_j)+\beta_k(s_j)\right]\right\}\right]\right\}\\
&=\Psi\left\{\ln\left\{\sum_{j=[1..4]}\exp\left[\alpha_k(s_j)+\beta_k(s_j)\right]\right\},\ln\left\{\sum_{j=[5..8]}\exp\left[\alpha_k(s_j)+\beta_k(s_j)\right]\right\}\right\}\\
&=\Psi\left\{\begin{array}{l}\Psi\left\{\ln\left\{\sum_{j=[1..2]}\exp\left[\alpha_k(s_j)+\beta_k(s_j)\right]\right\},\ln\left\{\sum_{j=[3..4]}\exp\left[\alpha_k(s_j)+\beta_k(s_j)\right]\right\}\right\},\\
\qquad\Psi\left\{\ln\left\{\sum_{j=[5..6]}\exp\left[\alpha_k(s_j)+\beta_k(s_j)\right]\right\},\ln\left\{\sum_{j=[7..8]}\exp\left[\alpha_k(s_j)+\beta_k(s_j)\right]\right\}\right\}\end{array}\right\}
\end{aligned}\tag{4.5}$$

The nested Ψ-operations derived in (4.4) can be realized with a tree of Ψ blocks. Figure 4-10 shows this structure. There absence of feedback loops makes this structure suitable for pipelined processing. Inserting registers or flip-flops between each level of the tree will ensure that the critical-path delay of the ξ block is less than that of the *AACS* block or any one of its derivative structures.

4.2.3. Memory requirements

A direct implementation of the BCJR algorithm initiates the two recursions from opposite ends of the underlying trellis representation. This will require the memory storage of at least one full set of path metrics corresponding to all nodes in the entire length of the trellis of a block code, which is illustrated in Figure 4-11 for a block length of N bits. The forward recursion tracks the rate and direction of the arriving symbols, and completes immediately after the decoder has received the entire block of symbols at time N. The backward recursion follows next. Intermediate α values are stored in the memory until the corresponding β values are available to be combined in the ξ block.

For an example code with constraint length 3, block size of 4096, and 5-bit representation for the state metrics, the memory requirement will be $2^3 \times 4096 \times 5 = 164$ kb. Although this requirement can be implemented in SRAM, the critical path of the MAP decoder will invariantly be affected. Memory access is approximately 2ns (general-purpose single-ported 32kb memories in 0.13 μm CMOS technology), significantly more than the 1ns required to add two pairs of short-wordlength numbers and select the maximum result in the ACS decoding logic (0.13μm CMOS ASIC design). Decreasing average memory access time by increasing the number of I/O ports is unsuitable because it leads to geometric growth in memory area. In addition to difficulties associated with memory requirements, a direct implementation of the BCJR algorithm also leads to extended latencies through each round of decoding. As previously noted, the decoder only initiates the backward recursions after receiving all symbols in a block. In principle, the prolonged latency is insignificant compared to the total latency required to coalesce the outputs of the SISO decoders for interleaving and deinterleaving through a few rounds of iterative decoding. However, a shortened latency through the first round of decoding may be advantageous towards decision-directed equalization and timing recovery efforts.

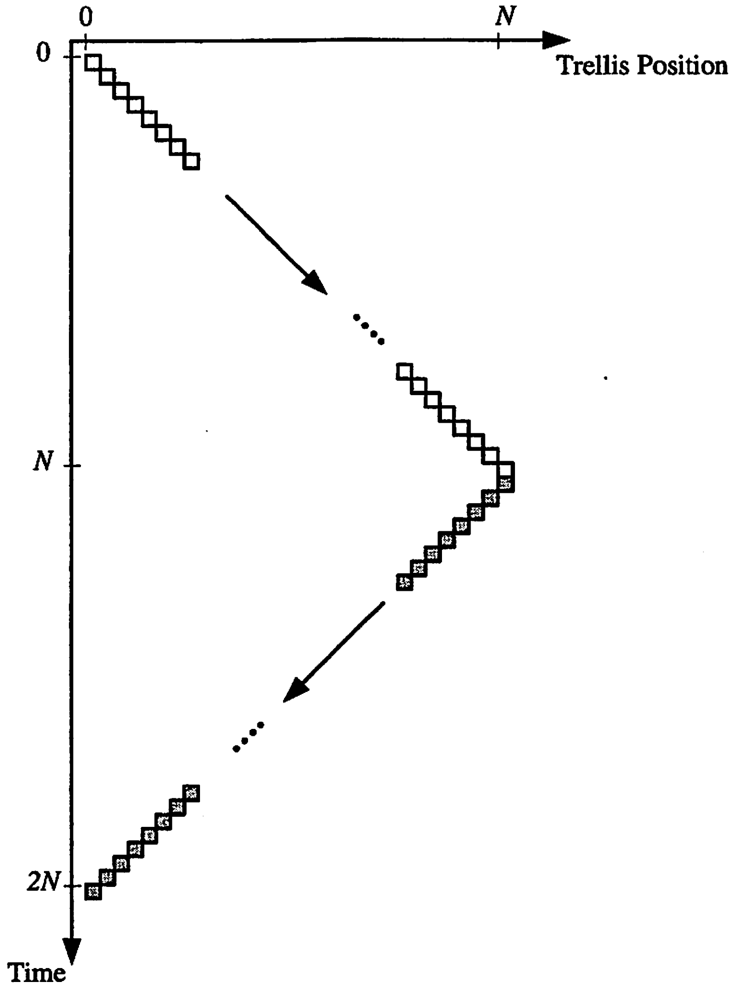


Figure 4-11. Direct implementation of BCJR requires memory storage of path metrics corresponding to each node in the trellis for the entire length of the block code.

The MAP decoder can avoid the pitfalls of large memory requirements and lengthy latencies by adopting windowing methods. Windowed BCJR algorithms address the difficulties of implementing backward recursions. Instead of initiating the backward recursions from the end of the trellis, these approximate algorithms take advantage of the observation that the backward recursion can begin from any arbitrary position along the trellis with uniform initial values. After a number of startup steps, defined as L, the path metrics converge with high certainty towards their asymptotic values, hence defined as the values obtained through a full backward recursion. This property is commonly exploited with the use of a finite traceback window length in Viterbi decoders. The value of L is historically set at five times the constraint length of the underlying convolutional code [3].

An implementation of windowed BCJR with asymptotically equivalent performance can be achieved using two overlapping windows for the backward recursion, shown in Figure 4-12 [4]. Each window spans a length of $2L$ and processes in two distinct modes. In the first mode, a startup sequence consisting of the initial L cycles performs the backward recursions. The values of β_ks obtained in this mode are unreliable and are not processed in the ξ block. The second mode comprises the next L cycles, which produces β_k outputs that are considered to have insignificant difference from the asymptotic values. Path metrics obtained from the second mode are processed in the ensuing ξ block together with the appropriate α_ks. The use of two overlapping windows ensures that an overall recursion rate of 1 symbol per period is maintained. One of the windows always produces reliable state metrics, while the other is processing the startup sequence. This method hides the overhead delay incurred in computing the startup sequence. It results in lower memory requirement and decoding latency at the expense of additional computational hardware.

Figure 4-13 shows a state-slice of the MAP decoder that is able to maintain a throughput equal to the symbol arrival rate. The γ-memory stores the branch metrics. An *α-ACSA* performs the forward recursion and stores its outputs in the α-Memory. Two *β-ACSAs* perform the backward recursion in accordance with the overlapping window method.

As previously noted, high throughput implementation of a MAP decoder requires a high-speed memory access that is faster than the typical SRAM-based memories. Memories using flip-flop based registers are fast (100ps delay in 0.13μm CMOS ASIC design) but each register costs about 5 times the area of a comparable SRAM cell. As such, it is necessary to schedule the memory access pattern in order to minimize both the memory requirement and the amount of overhead in control logic.

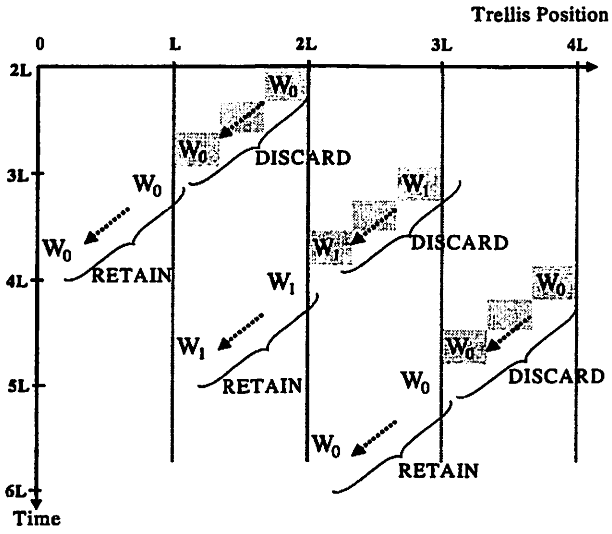


Figure 4-12. Backward iteration using 2 overlapping windows, W_0 and W_1 for BCJR algorithm. The shaded outputs are not used in the ensuing ξ block.

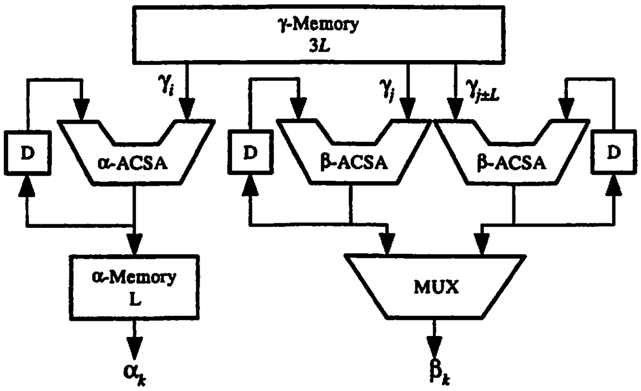
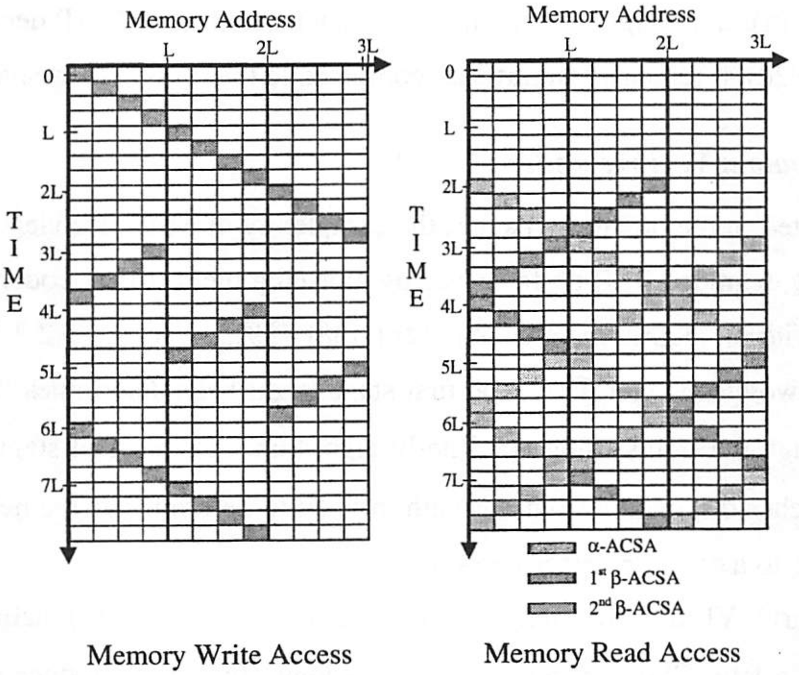


Figure 4-13. State-slice of a MAP decoder structure.



Memory Write Access Memory Read Access

Figure 4-14. Memory read and write access of branch metrics γ_k.

The timing diagram of a scheme that would limit the interval between the production and three consumption cycles to $3L$ is shown in Figure 4-14. The implementation partitions each γ-memory block into 3 sections of length L (3 sections of L columns in Figure 4-14) and deliberately delays the first forward iteration by $2L$. New data is cyclically written into each of the partitions while the write/read access pattern within each partition is continuously alternated between left-to-right and right-to-left directions every L periods. Each branch metric entry, γ_k, in memory is read once by each of the three *ACSA*'s. After the third and final read access, the memory location is immediately replaced with new data. The repetitive nature of the memory access within each partition promotes reduction in control logic, compared to random access memory, and is implemented as a bi-directional shift register.

Similarly, observations on the production and consumption patterns of the α values will indicate that each α-Memory block can be implemented with a bi-directional L-word shift register. The α_k and β_k values are summed in a tree structure (Figure 4-10) that evaluates (4). Although the maximum latency through each MAP decoder is $4L$ (80 cycles for $L = 20$), it remains insignificant compared to that of the interleaver.

4.3. Soft output Viterbi decoder

As noted in the previous chapter, the complexity of MAP decoders can be traded for marginally degraded BER performance by replacing them with decoders applying the soft-output Viterbi algorithm (SOVA), [26] [38] [39]. Section 3.2.2 described the process as a two-stage algorithm. The first stage (right half) determines the most-likely path, α, through the trellis using the Viterbi algorithm. The second stage evaluates all possible branches originating from this path in order to determine β, the next-most-likely path that leads to a complimentary decision.

An early VLSI implementation of a SOVA decoder [51] achieved 40Mb/s throughput in a 1μm CMOS standard cell technology. In order to reduce the power and area of the implementation, RAM macros were used. The path selections were done with the register-exchange technique to reduce the overall latency. A low power implementation of the SOVA decoder [16] uses DRAM blocks for path selection. The DRAMs need to be clocked at a multiple frequency of the decoding symbol rate. This places the memory as the limiting factor in the decoding throughput of the design.

The architecture of a high throughput SOVA decoder will be discussed in the next section, followed by the micro-architectural analysis of various add-compare-select structures in power, area, and delay space. A detailed description of the deeply pipelined mechanisms for the traceback, equivalence detection, and comparison of competing path metrics will also be provided.

4.3.1. SOVA decoder architecture

The architecture of an eight-state high throughput SOVA decoder is shown in Figure 4-15. The choice of eight states is synonymous with the application of EPR4 channels in the magnetic recording industry. The branch metric generator, eight ACS units, and the L-step survivor memory unit (SMU) form the building blocks of a conventional Viterbi decoder. Eight parallel ACSs compute the pairs of cumulative path metrics and select the winning paths in the underlying trellis representation of the convolutional code.

Each ACS also outputs the difference in path metrics between the two competing paths. The path decisions are stored into an array of L-step flip-flop-based FIFO buffers. The choice of flip-flops, in contrast to the use of RAM blocks in [16], permits a high throughput implementation that is independent of the delay of SRAM modules. The delayed signals are used in the M-step path-equivalence detector (PED) to determine the equivalence between each pair of competing decisions obtained through a j-step traceback, $j \in \{1,2,\ldots,M\}$.

The path metric differences from the eight ACSs are stored in FIFOs registers of depth L. Using the output decision from the SMU as a multiplexer select signal, the delayed metric difference at the most-likely state is input to a reliability measure unit (RMU). The SMU output is also used to select the results of the equivalence tests performed on competing traceback paths that start to deviate from the most-likely state. The selected equivalence results are evaluated in the RMU in order to output the minimum path metric difference reflecting competing traceback paths that result in complementary bit decisions, $\hat{x}$ and $\bar{\hat{x}}$.

Architectural and implementation details of the blocks in a SOVA decoder will be discussed in the following sections. The ACS structures are analyzed in Section 4.3.2, and several options for survivor path decoding are presented in Section 4.3.3.

4.3.2. Add-Compare-Select structures

The implementation of a high throughput SOVA decoder is dependent upon the realization of the *ACS* blocks under practical power and area constraints. Prior to this work, the only form of comparison amongst the various competing structures was the assessment of area penalty for ACS structures optimized strictly for minimum delay [82] [54]. The surveyed structures were derived from prior publications spanning a 20-year history in research of high throughput Viterbi decoders.

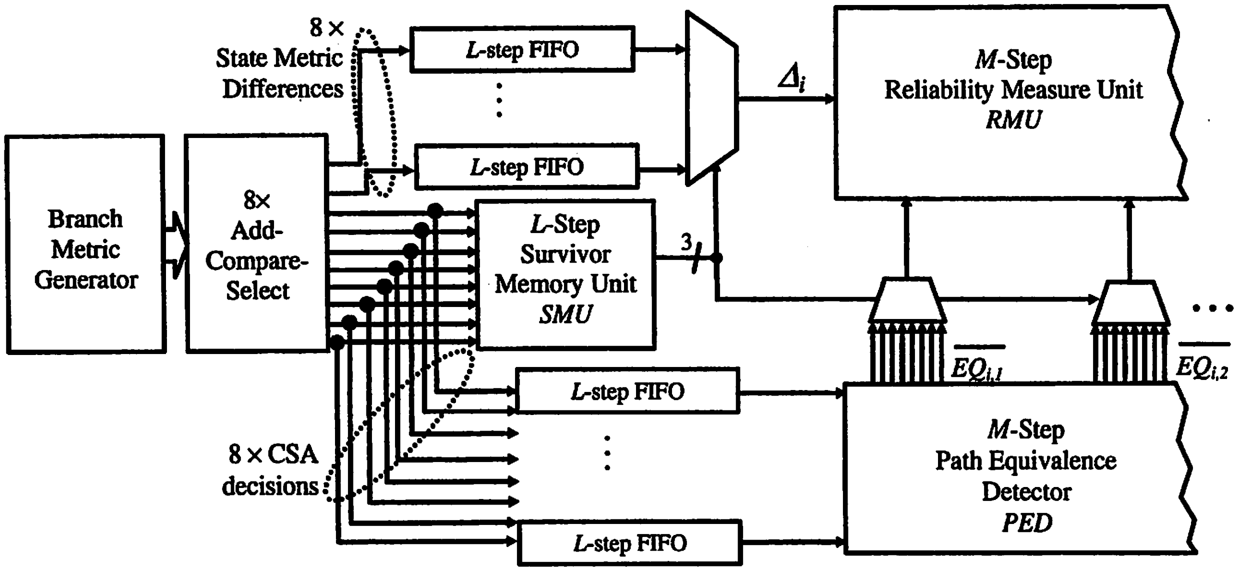


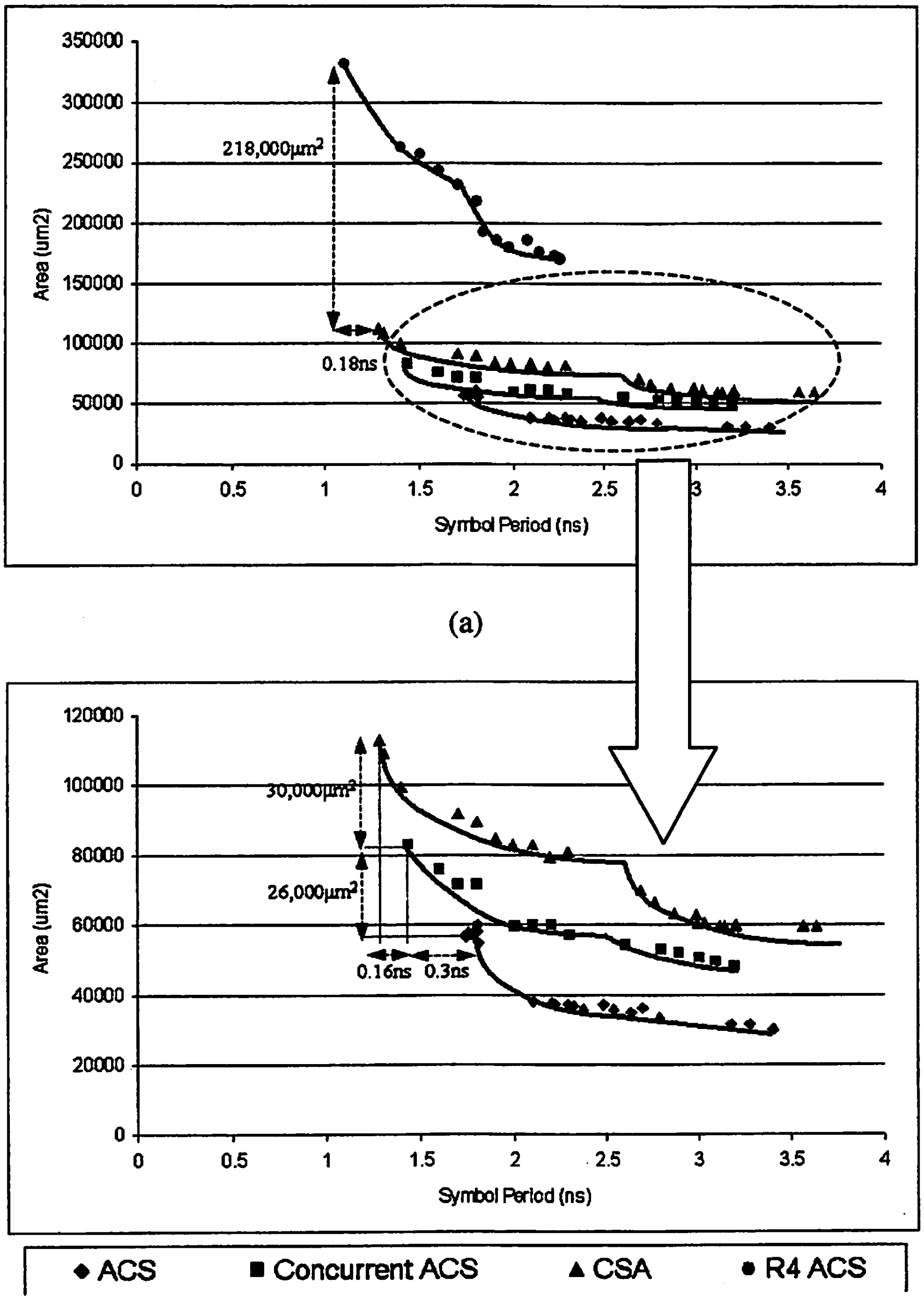
Figure 4-15. System architecture of 8-state SOVA decoder.

In contrast, this work performed an exploration to examine the area-throughput and power-throughput tradeoffs across a range of permissible critical path delay constraints. Using the radix-2 baseline, the different *ACS* structures were synthesized using low-threshold cells with high supply at best-case conditions. The test was conducted through architectural synthesis of a block of eight *ACSs* using general-purpose standard-cell 0.18μm CMOS technology. A network resembling the underlying trellis structure interconnected the ACSs. The decision outputs of the ACS structures were loaded with 200fF to simulate the large capacitive load in the register-exchange and FIFO memories.

Besides the baseline, structures tested include the concurrent *ACS*, the transformed and retimed *CSA*, as well as the radix-4 *ACS*. These structures are respectively analogous to the concurrent *AACS*, transformed *ACSA*, and radix-4 *AACS*. Since the SOVA does not require an additive correction term, the associated set of adders can be removed.

Results of the synthesis experiment are plotted in Figure 4-16 and Figure 4-17. The synthesis algorithm [104] trades a higher area for delay reduction through sizing and logic transformations. Each curve tracks the same behavior. As the decreasing critical path constraint approaches a minimum value, the area and power consumption of the synthesized structure increases asymptotically due to the use of increased gate sizes. The kinks in the curves correspond to optimization boundary conditions where logic transformations are preferred over increased sizing.

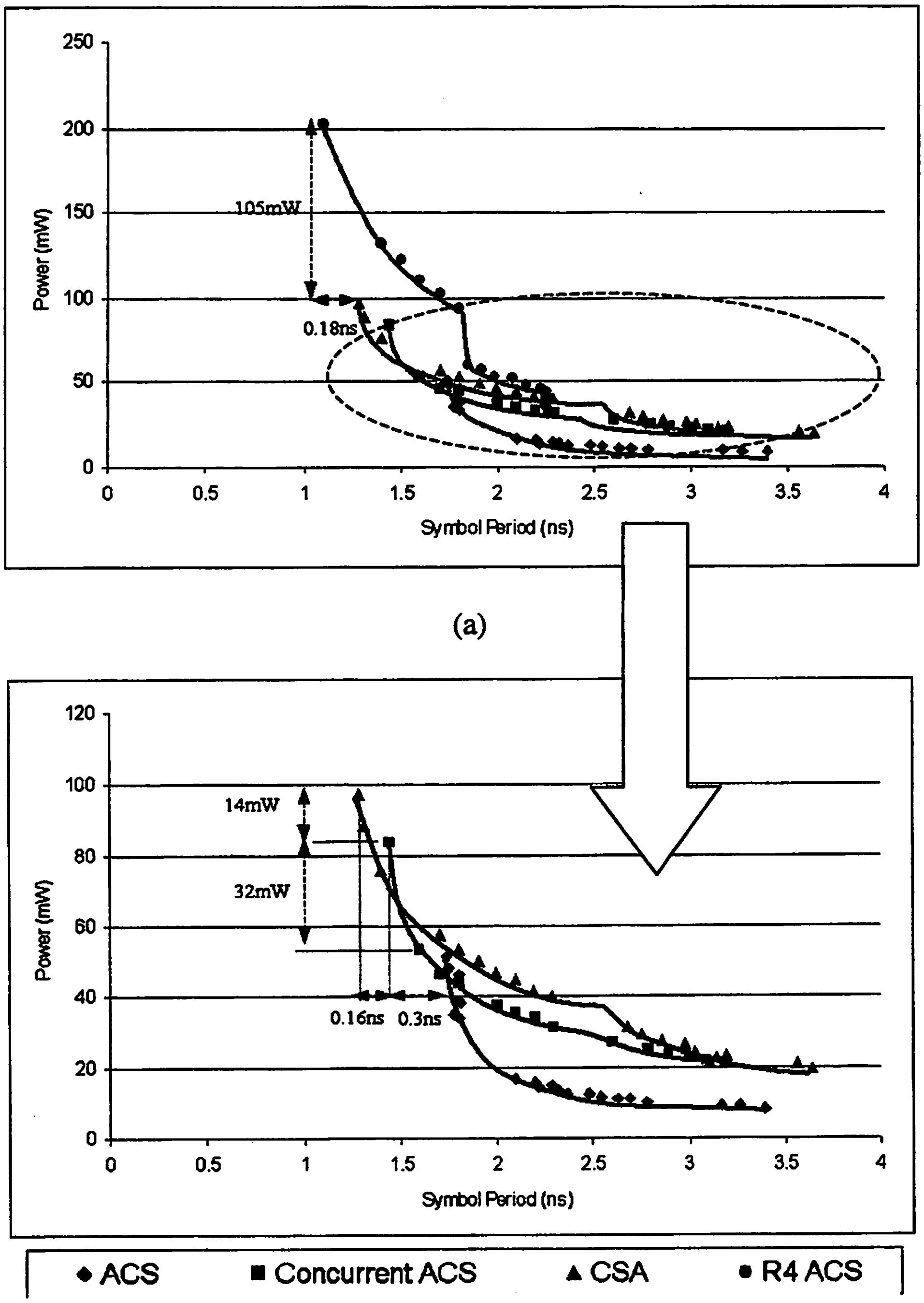
Table 4-1 shows a comparison of the power, area and delay of the test structures. The absolute numbers are dependent on the setup of the experiment such as the exact drive strengths of the inputs and capacitive output loads. However, the relative numbers are applicable for a wide range of operating conditions. As expected, the radix-4 ACS, which has been accounted for the doubled symbol rate, has the least critical path delay. The throughput is faster than the next-fastest structure by a margin of 17%, but requires almost three times the area and two times the power. The radix-4 ACS is consistently larger, and consumes more power than any of the other structures.



(a)

(b)

Figure 4-16. Area comparisons of (a) various ACS structures and (b) a detailed magnification of the ACS, concurrent ACS, CSA comparison.



(a)

(b)

Figure 4-17. Power comparisons of (a) various ACS structures and (b) a detailed magnification of the ACS, concurrent ACS, CSA comparison.

Both transformed CSA and concurrent ACS are able to achieve improvement in throughput with significantly less area and power penalty. The choice of ACS structure is dependent on the required critical path delay, and can be inferred from Figure 4-16 and Figure 4-17. The FO4 delay in this implementation technology is 50ps. At this particular set of operating conditions, the transformed CSA structure is suitable for applications with critical path delays specified between 26 to 29 FO4 delays. The concurrent ACS becomes the choice structure for delays between 29 to 35 FO4 delays. For low throughput rates with critical path delays above 35 FO4 delays, the ACS structure is the best choice in terms of both area and power consumption.

4.3.3. Survivor path decoding

The two ML paths are determined by a two-stage traceback. A survivor memory unit (SMU) is cascaded with a combination of path-equivalence detector (PED) and reliability measure unit (RMU). The SMU and PED have similar functions. Both essentially examine a list of competing paths by retracing a history of decisions and path metric differences. Previous implementations of the SOVA used either the register-exchange method [51] or memory traceback [83] methods.

TABLE 4-1.
MAXIMUM THROUGHPUT EFFICIENCY OF VARIOUS ACS ARCHITECTURES

	Relative Symbol Throughput	Relative Area	Relative Power	Critical Path
Radix-2 ACS	1	1.00	1.00	(2 x 2-input Adders) + (1x Multiplexer)
Radix-2 Concurrent ACS	1.2	1.46	1.63	(1 x 4-input Adder) + (1 x Multiplexer)
Radix-2 CSA	1.4	1.99	1.89	(1 x 2-input Adder) + (1 x Multiplexer)
Radix-4 ACS	1.6	5.86	3.94	(1 x 2-input Adder) + (1 x 4-way comparator) + (1 x Multiplexer)

A. Register-exchange and memory-traceback methods

A **register-exchange** consists of a two-dimensional array of one-bit registers and multiplexers as shown in Figure 4-18. The registers in successive stages are interconnected to resemble the trellis structure of the convolutional code. A global clock signal controls the registers. The frequency of the clock determines the throughput of the Viterbi decoder. The path decision from each of the eight ACSs is input to the register-exchange pipeline, and also selects the outputs of a corresponding row of multiplexers. At each clock cycle, a multiplexer located at row i and column k $\{i \in [1,2,\ldots,8]$, $k \in [1,2,\ldots,L]\}$ outputs a bit decision corresponding to a traceback of length k, originating from state i. This bit is stored in a register, and will be input to a multiplexer at column k+1 in the following clock cycle.

The **memory traceback** method has commonly been used in low-throughput, low-power applications. It simply writes a vector of path decisions from the ACS recursions into RAM in each iteration of the Viterbi algorithm. After an initial startup delay, the decisions are retraced by reading the stored decisions in the reverse direction. Previous solutions have generally employed some variation of the k-pointer traceback architecture [83]. They used k-1 parallel read pointers that accessed as many independent banks of memory, while a write pointer simultaneously stored the decisions from the ACS recursions into a k^{th} memory bank. An alternative [16] is to use a single bank of multi-ported DRAM.

The memory traceback method permits the design of very compact RAM that provides significant area advantages. In 0.18μm CMOS technology, the area of a typical SRAM cell is about 2.4μm^2, in contrast with the 50μm^2 area required for a flip-flop used in the register-exchange method.

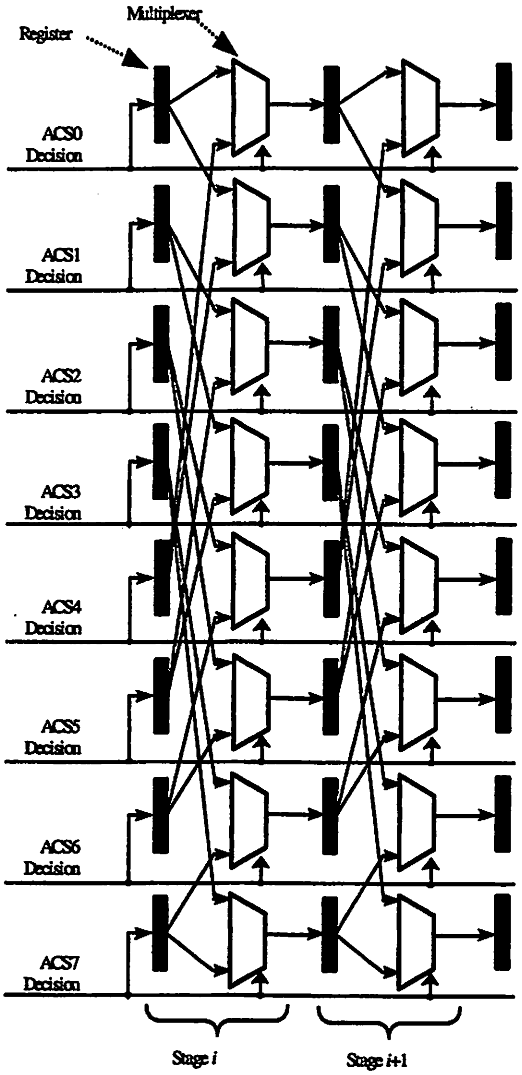


Figure 4-18. Example 8-state register-exchange survivor memory unit used in VA-SMU.

The memory traceback method stores the intermediate bit decisions in static locations in memory. Since SRAM blocks typically operate by reading or writing multiple bits per cycle, a vector of decisions output by the parallel ACSs can be written into memory simultaneously. The traceback operation only needs to recall those bit decisions that constitute part of a traceback path. This contrasts with the register-exchange method, which constantly moves an array of bit decisions through a pipeline of flip-flops. In principle, this gives the memory traceback method inherent power benefits. As the number of states rises, the register exchange is required to shift an increasing number of bits through its pipeline. However, for decoders with small number of states, the use of standard SRAM modules offers little power or area advantage over register-exchange because of the overhead of peripheral circuitry and standard word addressing [52]. Register-exchange achieves high throughputs easily because its critical path consists only of a multiplexer and a register. On the other hand, standard SRAM macros in 0.18μm technology have much longer cycle times than the synthesized CSA recursion. Therefore, with the small number of states in our decoder, the register-exchange is the appropriate structure for high throughput implementations.

B. Path equivalence detector (PED) and reliability measure unit (RMU)

With the emphasis on high throughput implementation, this section examines the use of register-exchange and the modifications necessary to implement the path equivalence detector (PED) and reliability measure unit (RMU). The register-exchange method used in the SMU provides a convenient way to determine if competing traceback paths lead to equivalent bit decisions. The two inputs to each multiplexer reflect the competing bit decisions, and a test for their equivalence can be realized by the addition of an XOR gate at each multiplexer location (Figure 4-19). The ensuing Boolean outputs $\overline{EQ}_{ij}(n)$ indicate the equivalence between the two competing decisions obtained through a j-step traceback from state i.

From ACS$_i$, the difference between the two path metrics, $\Delta_i(n)$, arriving at time n, state i, is retained in FIFO buffers; $i \in \{1,2,...,8\}$. The output from the SMU selects $\Delta_i(n)$ and $EQ_{i,j}(n)$, which correspond to the values along the ML path, as inputs to the RMU (Figure 4-20).

The RMU consists of comparators and multiplexers in a pipeline that selects the minimum $\Delta(n)$ along the ML path. It is initialized with the maximum binary representation of the reliability measure, "111111". Based on the $\overline{EQ}$ inputs, each pipelined section outputs one of the following:

$\overline{EQ} = 0$: Reliability measure from the previous step

$\overline{EQ} = 1$: Min{Δ_i, Reliability measure from the previous step }

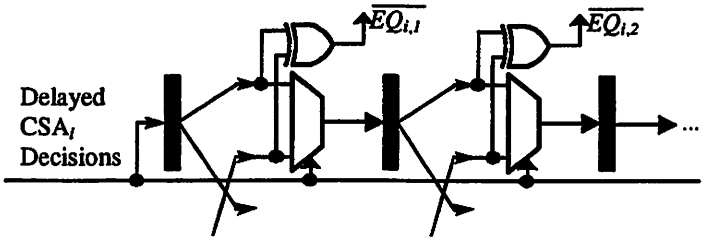


Figure 4-19. State-slice of register-exchange used in the path-equivalence detector (PED).

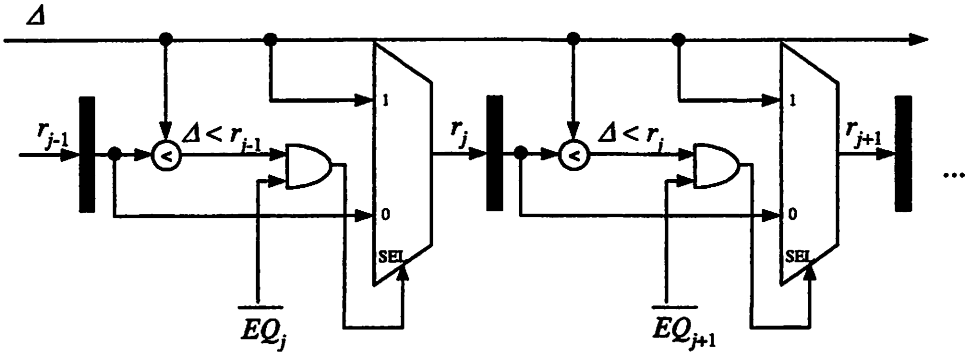


Figure 4-20. Pipelined section of reliability measure unit (RMU).

Compared with a Viterbi decoder implementation, the total size of the SMU and PED is approximately doubled ($L = M$). The RMU overhead includes M pipeline stages, each of which consists of a 2-input comparator with its Boolean output logically AND'd with the $\overline{EQ}_j$ input, a multiplexer and a 6-bit register. The overall latency through the SOVA decoder is $L + M$. The additional latency remains insignificant compared to the overall latency in the Turbo-SOVA system, which is dominated by the latency through the interleavers.

Based on the discussion provided in this section, a 500Mb/s 8-state SOVA decoder has been implemented in 0.18μm CMOS technology. The decoder occupies a core area of 0.5mm^2, and dissipates 400mW power with random input data.

4.4. Summary

The chapter has described architectures suitable for high throughput implementations of the MAP and SOVA decoders. The memory requirements were realized using fast shift registers. The throughput bottleneck is found at the AACS or ACS recursions, and a number of micro-architectures have been discussed and evaluated.

In particular, the SOVA architecture has been implemented on an ASIC platform with the CSA structure. The details will be presented in Chapter 7. The next two chapters will look at architectural issues that are relavant to the design of an LDPC decoder.

5. LDPC DECODER ARCHITECTURES

Two types of LDPC decoder implementations, the parallel and serial architectures, are introduced. These belong to opposite ends in the spectrum of possible architectures, and affects the manner in which PEs in an LDPC decoder communicate with one another, either through an interconnect fabric or memory elements. The benefits and difficulties of each of these structures are investigated, and implementation issues such as area, power, and throughput are discussed.

The building blocks of an LDPC decoder are presented first. The fixed-point implementation of PEs associated with each class of nodes, and their implications on the overall decoder architecture are elaborated.

In addition, the final section in this chapter introduces some aspects of specific classes of LDPC codes based on finite field geometries [77] and rectangular lattices [114]. These codes have demonstrated properties that are highly advantageous towards the implementation perspective.

5.1. Parallel architectures

The message-passing algorithm used in LDPC decoders is inherently parallel. A hardware implementation of an LDPC decoder can exploit maximum amount of parallelism by associating each node in the underlying bipartite graph (Figure 5-1) with a processing element, and each edge with wire interconnect. A fully parallel architecture provides potentially the fastest decoding throughput and lowest power dissipation.

In order to demonstrate the effects of scalability, a decoder for an example rate-8/9 LDPC code, with a block size of 4608 is examined. The variable nodes and check nodes have edge degrees of 3 and 27 respectively. The fully parallel decoder requires 512 check node processing elements and 4608 bit node processing elements. Based on the computational requirements of the processing elements, the approximate complexity of the decoder implementation is obtained through synthesis in 0.13μm CMOS technology. The estimates are listed in Table 5-1

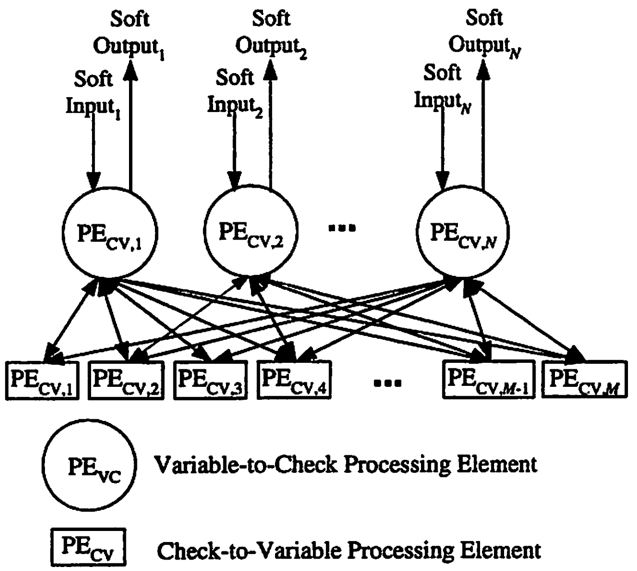


Figure 5-1. Parallel architecture.

Assuming perfect timing recovery and ignoring the issue of decoding latency, a fully parallel LDPC decoder can maintain a high throughput even when operated at low clock frequency and power supply. This results in a low power implementation. Figure 5-2 shows an example structure that takes advantage of the serial nature of a channel decoder. The extrinsic information is input through a shift-register chain. The soft inputs at the variable-to-check PEs are only valid once every *N* cycles, for a block size of *N*. Therefore, the parallel LDPC decoder is required to complete the decoding iterations in *N* cycles.

TABLE 5-1
COMPLEXITY ESTIMATES FOR OUTER DECODER WITH LDPC APPLICATION

	512x4608 decoder (Speed opt.)	512x4608 decoder (Power opt.)
Number Gates*	3,152,896	1,560,576
Delay (ns) (per iteration)	6.7	10.1
Area (um^2)	55,698,944	36,611,072

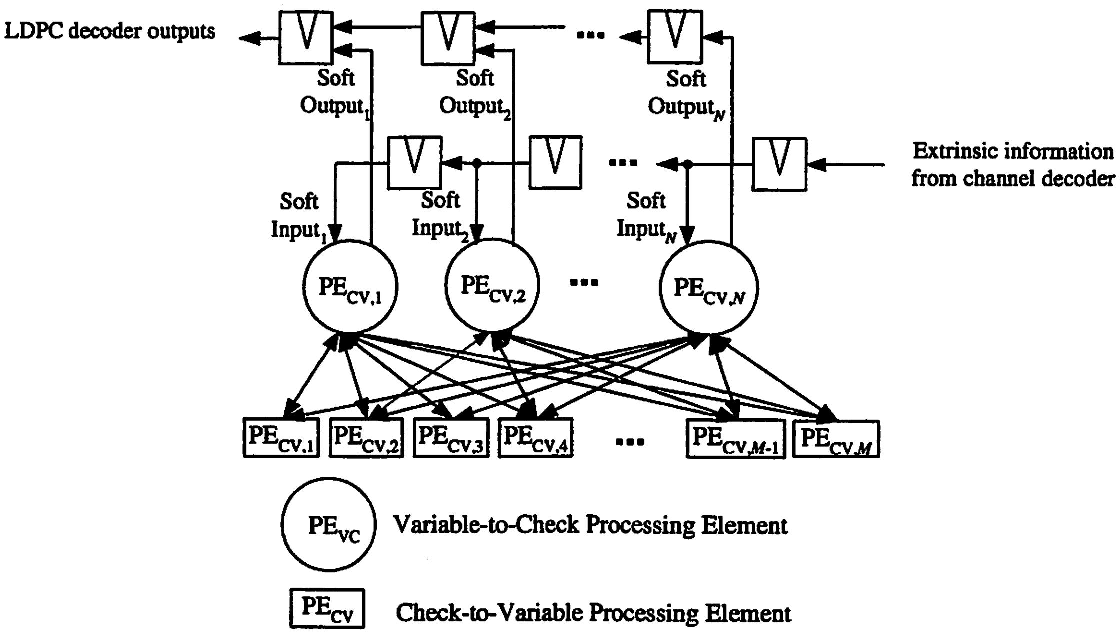


Figure 5-2. Parallel LDPC decoder with serial input stream from channel decoder.

An example implementation of a rate-½, 1024-bit LDPC decoder [1] in 0.16μm technology with 5 metal layers succeeded in demonstrating such a parallel architecture.. The log-likelihood inputs enter the decoder via 16 shift-register chains, which also operate as serial to parallel converters. The outputs feed directly into a fabric of interconnect that links 1024 variable node PEs with 512 check node PEs. This effectively maps the entire bipartite graph onto silicon. The massive parallelism permits a high throughput implementation of 1Gb/s with a relatively low-frequency clock at 64MHz. The decoder completes 64 iterations of decoding within one block-delay (1024ns) and dissipates less than 700mW.

Such throughput and power efficiency comes at the cost of area, which is not revealed by Table 5-1 either. Due to a high level of routing congestion, the decoder occupies an area of 7mm×7mm, where logic density is only 50% in order to accommodate the complexity of the interconnect fabric. Compared with the implementation in [1], a silicon realization of a 4608-bit decoder will have at least 4 times more interconnect wires that are also prohibitively long. Increased number of routing layers will help, but the logic density is unlikely to fall below 50%. In general, the random connectivity defined in LDPC codes leads to long interconnect lengths that are of the order of the core dimensions. As the size of the code grows, the floorplan utilization is also expected to deteriorate.

A usual solution to address congestion in a design is partitioning. The number of long global interconnects are reduced at the expense of increased number of short local interconnects; the target is to achieve a net reduction in average length of interconnects. However, due to the interlaced data dependencies in LDPC codes, partitioning is difficult, although a notable effort is based on simulated annealing to minimize the total length of interconnect [112].

5.2. Serial architectures

A serial architecture removes the problem of routing congestion by replacing the physical interconnect with SRAM. Using a limited number of processing elements [31] as shown in Figure 5-3. Serial architectures require significantly less gates and area of implementation. An LDPC decoder with a targeted clock of more than 500MHz will require direct-mapped hardware of the processing elements that compute both variable-to-check and check-to-variable messages, Q_{nm} and R_{mn} respectively. These messages are stored temporarily in memory between their generation and consumption cycles. This structure results in less area and much less routing, but dramatically increases memory requirements.

A direct implementation of the architecture shown in Figure 5-4 will face imminent stalls as each stage of decoding waits for the previous stage to fill the data dependencies. The solution to this alternates the two sets of processing elements between consecutive blocks of codewords. Each memory block in the LDPC decoder is implemented as a pair of buffers that are alternatively accessed (Figure 5-4). This ensures that the input data is always available in memory. The outputs of each processing elements overwrite the memory locations of its inputs. Under such a scheme, full utilization of hardware is assured if both sets of hardware evaluate and compute messages at the same throughput rate.

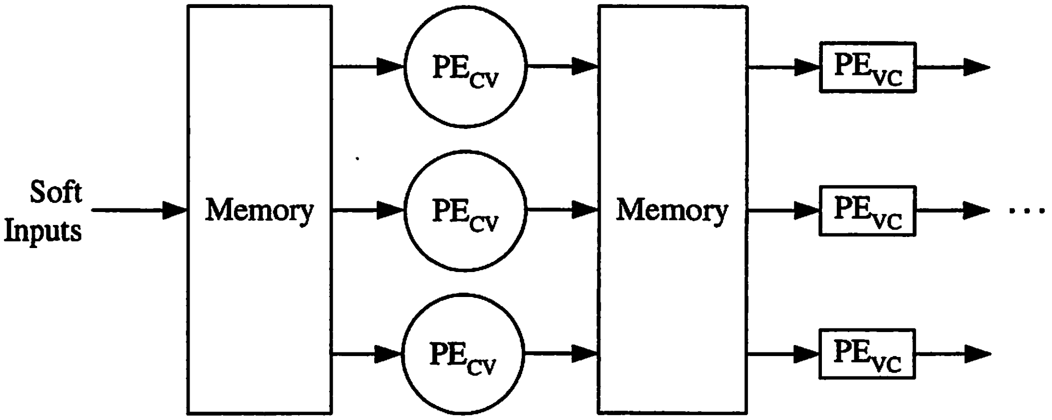


Figure 5-3. Serial architecture.

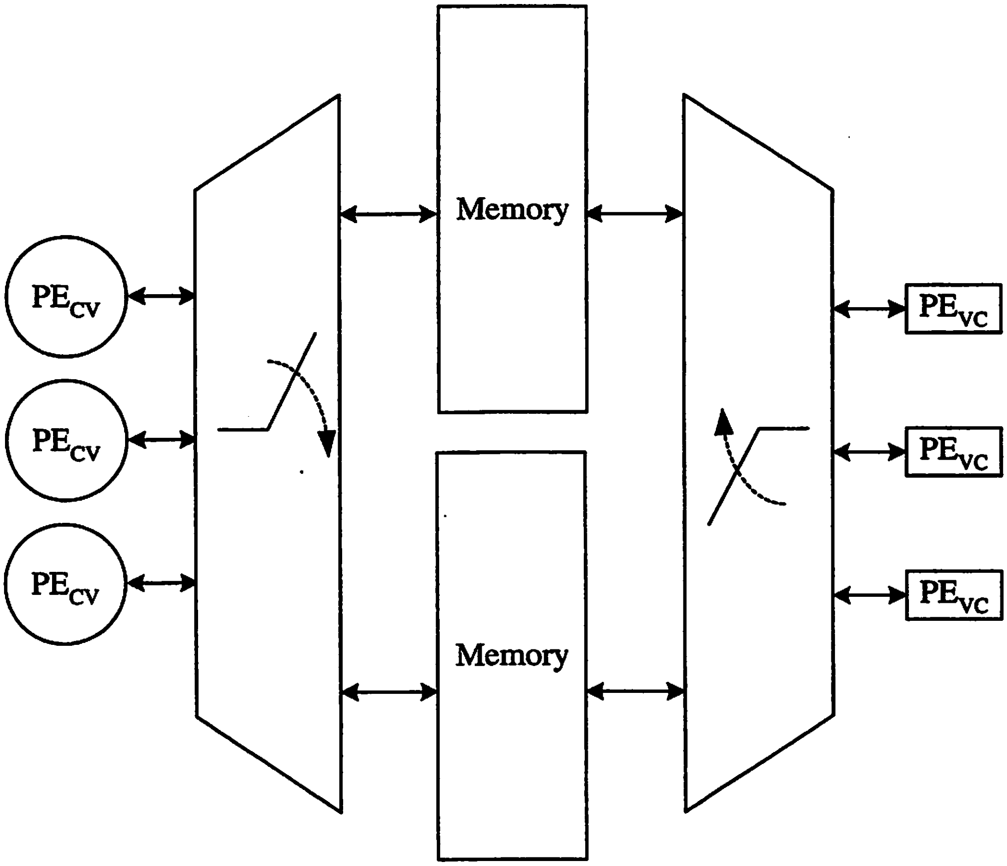


Figure 5-4. Serial decoding by alternating between two memory buffers containing consecutive blocks of data

The memory implementation of serial architectures has to address the random network of edges in the bipartite graph and interlaced data dependencies in the message computations. The example in Figure 5-5 shows that dependencies from the variable node *B* fan out to four other variable nodes *A, D, E,* and *F* in the one iteration. The dependencies from nodes *A* and *F*, in turn, fan out to the remaining node, *C*. In LDPC codes with good asymptotic performance, the trace of dependencies fans out rapidly through each iteration. This behavior is related to the girth of the graph [84] and implies that the two classes of computations over a single block of inputs, variable-to-check and check-to-variable processing, cannot be overlapped.

Hence, the size of the memory required is dependent on the total number of edges in the particular code design. The example 4608-bit LDPC code used in previous sections comprises a total of 13824 (4608 × 3) edges in the graph. Each edge corresponds to a message that has to be stored in memory. The lack of spatial locality between any subset of bit nodes connected to the same check node (and vice versa) makes it impossible to update the messages for the next round of decoding until a majority of the messages in a particular direction are received. Therefore, the size of the memory requirement is dependent on the total number of edges in the particular code design. A serialized rate 8/9 4608-bit LDPC In 0.13μm CMOS technology, the density of an SRAM is approximately 15μm^2/bit. Using 4-bit fixed-point messaging, the area of memory required is 0.8mm^2. This is equivalent to the size of one hundred bit-node processing elements.

Serial architectures that use more than one processing element require memory devices with operating frequencies that are faster than the datapath, or multiple I/O ports. With a memory access time of approximately 2ns in 0.13 μm for 1024 32kB, the increased frequency approach is not suitable for high throughput decoder implementation. Memories with multiple I/O ports have significantly larger areas due to the requirement for additional address decoders, word and bit lines. The access time for multiported SRAM is also slower to account for the increased loading effect of the bit lines. The next section discusses architectures comprising several processing element that communicate with a bank of memories via a common bus interface.

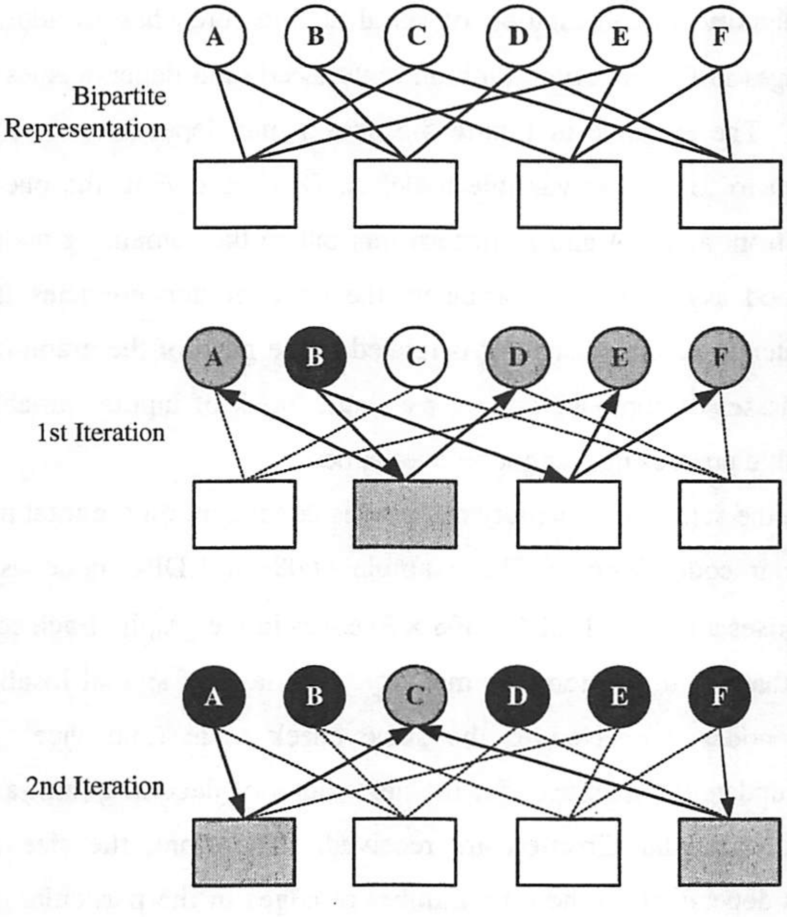


Figure 5-5. Tracing dependencies through a bipartite graph.

5.3. Shared memory architectures and partitioned matrix

In order to improve the speed of multiple-memory access, a solution [31] divides the memory into a bank of *P* independent SRAMs, and uses a crossbar switch to enable communication between *P* processing elements. Each processing element selects an input from one of the SRAMs in each period of the memory cycle. However, inadvertent memory access collisions will cause the processing elements to stall. An ad-hoc scheduling method is used to minimize the number of such conflicts. Once determined, this schedule has to be stored as ROM data. This requirement is similar with the case of interleavers that cannot reproduce the interleaving sequence on the fly.

$$\left(\begin{array}{|c|c|c|c|c|}
\hline
0001 & 0000 & 0000 & 0000 & 1000 \\
0100 & 0000 & 0000 & 0000 & 0010 \\
0010 & 0000 & 0000 & 0000 & 0010 \\
1000 & 0000 & 0000 & 0000 & 0001 \\
\hline
0000 & 0000 & 1000 & 0000 & 0000 \\
0000 & 0000 & 0010 & 0000 & 0000 \\
0000 & 0000 & 0010 & 0000 & 0000 \\
0000 & 0000 & 0001 & 0000 & 0000 \\
\hline
0000 & 0001 & 0000 & 0000 & 0000 \\
0000 & 0100 & 0000 & 0000 & 0000 \\
0000 & 0010 & 0000 & 0000 & 0000 \\
0000 & 1000 & 0000 & 0000 & 0000 \\
\hline
\end{array}\right)$$

Figure 5-6. $M \times N$ parity check matrix partitioned into $j \times k$ subblocks.

More generally, the scheduling problem is analogous to shuffling the rows and columns of the parity check matrix in order to obtain a well-partitioned design. A well-partitioned design permits the parity check matrix to be divided into groups of $j \times k$ sub-blocks, with each of the $j \times k$ partitions being either a zero matrix or some permutation of the identity matrix. The latter has a maximum of one non-zero entry in each column and each row. Recently, a number of deterministic algebraic method, [96] and [114], have produced LDPC codes that exhibit this nature. The corresponding LDPC parity check matrix can be arranged in the manner shown Figure 5-6. Such partitioning permits messages, variable nodes, and check nodes to be grouped according to the dividing perforations. Grouped messages can be stored in a common bank of memory. The partitioning divides the matrix into groups of rows and columns. Each of these grouped columns represents a collection of variable nodes that share a single processing element for variable node processing. Likewise, the grouped rows represent a collection of parity checks and share a single PE designed for check node processing. Each non-zero sub-block in the partitioned matrix represents a custom interconnect between a pair of shared PEs. Although these codes have properties that benefit the decoder hardware implementations, they tend to display higher error floors [101], which is a result of low weight codewords.

Contrary to the common practice of defining implementation architecture after a particular code has been defined; it is possible to design a pseudo-random code such that the implementation benefits of the shared memory architecture are intrinsic. One could start with the definition of the size, column weight, and row weight of a parity check matrix. The column weight determines the number of parallel ports that the shared memories need to support. Both column weight and row weight define the fan-in and fan-out of the corresponding PEs, which affects the area, power consumption and throughput of the implementation. Depending on practical constraints, the maximum numbers of realizable variable-node and check-node PEs are determined as ${}^{N}/_{j}$ and ${}^{M}/_{k}$ respectively. Finally, the parity check matrix is assembled by concatenating blocks obtained from a list that comprises the $j{\times}k$ zero matrix and all permutations of binary matrixes containing a non-zero entry in each column and each row. The column weight and row weight specifications are observed by stacking as many $j{\times}k$ sub-blocks in each column or row, respectively. During the process of block assembly, one should be aware that short cycles in the graph should be avoided. For example, cycles of length four can be prevented by specifying that the bit-wise XOR between any two columns in the parity check matrix will not have more than one non-zero entry in the resulting vector.

Partitioning of an LDPC decoder does not decrease the overall size of memory requirement, but can result in smaller and faster memory implementations. All messages corresponding to variable nodes in the same group are stored on a single bank of memory, local to the shared computational logic implementing the variable node processing. Since each sub-block has at most one non-zero entry in each row, each parity check will read/write from at most one location within the shared bank of memory. Without further constraint on the property of the parity check matrix, the edge degree on a variable node determines the number of ports required in the bank of memory. LDPC codes with small edge degrees will therefore benefit most from this architecture. Figure 5-7 shows an example of shared memory architecture of an LDPC decoder for a code described with a column weight of 3 (variable nodes have edge degree of 3).

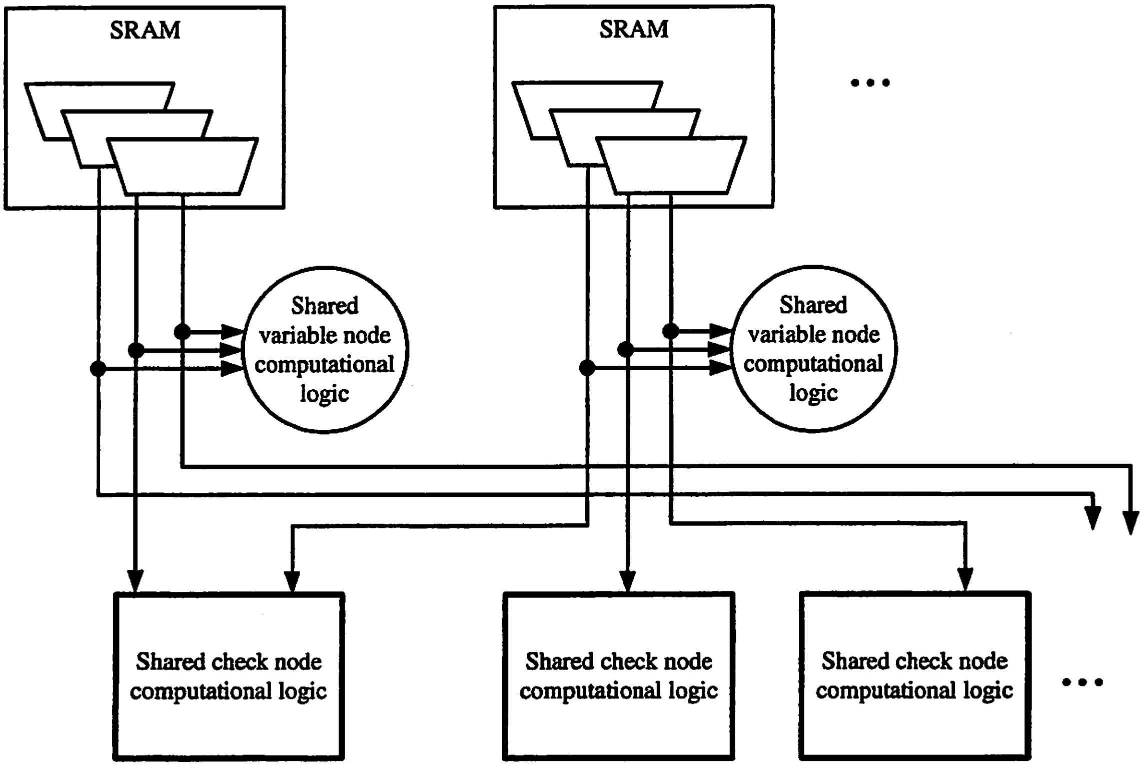

SRAM
SRAM
• • •
Shared variable node computational logic
Shared variable node computational logic
Shared check node computational logic
Shared check node computational logic
Shared check node computational logic
• • •

Figure 5-7. Shared memory architectures with shared computational logic, and interconnect.

5.4. Computation blocks

The arithmetic requirements of an LDPC decoder were presented in the Chapter 3. Q_{nm} was defined as the message computed at variable node *n*, and passed to check node *m*. Conversely, R_{mn} represents the message computed at check node *m*, and passed to variable node *n*. The description of the variable node processing in (3.16) shows a large number of common terms in the summation. The equation can be reformulated as (5.1) to reduce the overall number of required additions. Likewise, (5.2) shows a reformulation of (3.17) that exploits common summation and sign-product terms.

$$\begin{aligned} Q_{nm} &= \alpha_n + \sum_{m' \in \mu(n) \backslash m} R_{m'n} \\ &= \alpha_n + \left(\sum_{m' \in \mu(n)} R_{m'n} \right) - R_{mn} \end{aligned} \tag{5.1}$$

$$
\begin{aligned}
R_{mn} &= \Phi^{-1}\left(\sum_{n' \in N(m) \backslash n} \Phi(Q_{n'm}) \right) \times \left(\prod_{n' \in v(m) \backslash n} \operatorname{sgn}(Q_{n'm}) \right) \cdot (-1)^{|v(m)|} \\
&= \Phi^{-1}\left\{ \left(\sum_{n' \in N(m)} \Phi(Q_{n'm}) \right) - \Phi(Q_{nm}) \right\} \\
&\qquad \times \left(\operatorname{sgn}(Q_{nm}) \cdot \prod_{n' \in v(m)} \operatorname{sgn}(Q_{n'm}) \right) \cdot (-1)^{|v(m)|}
\end{aligned} \tag{5.2}
$$

An example structure for computation of Q_{nm} is shown in Figure 5-8. Introducing pipeline registers within the tree structure can increase the throughput of the computation. If the edge degree of the variable node is high, the tree structure will require a large number of messages to be simultaneously available at the inputs of the processing elements. This is feasible if each input message is routed through a custom wire interconnect. However, architectures that make use of SRAM memory will face difficulties in providing a large set of parallel messages due to limited I/O bandwidth associated with SRAMs.

Conversely, a processing element for R_{mn} messages may make use of a recursive structure, as shown in Figure 5-9. This is preferred over a tree structure because of the typically larger edge degrees of check nodes. Using signed-magnitude representation, the most-significant bits (MSB) of the input messages, which are the signed bits, are recursively XOR'd in the lower half of the structure to evaluate $\prod_{n' \in v(m)} \operatorname{sgn}(Q_{n'm})$. The magnitude bits are fed into a lookup table (LUT) that implements the $\Phi(\cdot)$ function. Outputs of the lookup table are also recursively summed to evaluate $\sum_{n' \in v(m)} \Phi(Q_{n'm})$.

The outputs of both recursions are sampled at $\frac{1}{|u(n)|}$ of the overall clock period. The divided clock is also used to reset the feedback registers. Two $|\mu(n)|$-cycle FIFO are used to delay the values of $\operatorname{sgn}(Q_{n'm})$ and $\Phi(Q_{nm})$. This allows their individual contributions to $\prod_{n' \in v(m)} \operatorname{sgn}(Q_{n'm})$ and $\sum_{n' \in v(m)} \Phi(Q_{n'm})$ to be removed respectively after the recursions such that marginalization is implemented. Finally, an output lookup table implements $\Phi^{-1}\{\cdot\} \cdot (-1)^{|v(m)|}$.

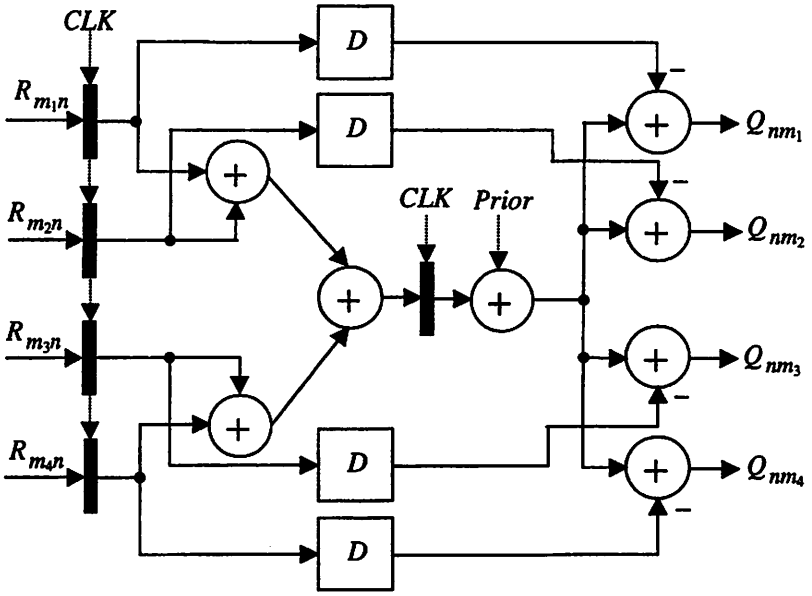


Figure 5-8. Binary adder tree to compute Q_{nm} messages in a multi-stage pipeline.

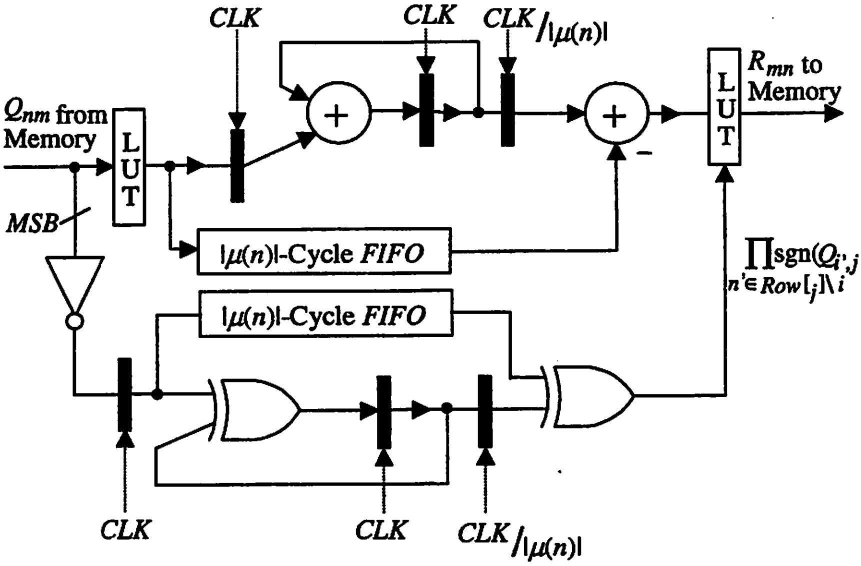


Figure 5-9. Recursive pipelined implementation to compute R_{mn} messages.

With the exception of the lookup tables implementing $\Phi(\cdot)$ or $\Phi^{-1}(\cdot)$, the computational elements in a LDPC decoder are fairly straightforward. Unlike the ACS or AACS used in SOVA and MAP decoders, the recursions described in the LDPC decoder can be easily pipelined for higher throughputs without a quadratic growth in complexity.

In the following sections, the fixed-point implementations of the decoder building blocks are presented.

5.4.1. Fixed-point implementation of lookup table

The output of the lookup table implementing $\Phi(\cdot)$ function rapidly increases as the value of the input approaches zero. In order to mitigate the quantization effects compounded by the nonlinear behavior, the usual approach would be to increase the wordlength of the outputs of the $\Phi(\cdot)$ lookup tables. This results in increased arithmetic complexity since the adders, $\Phi^{-1}(\cdot)$ lookup tables, and shift registers will have to operate with larger input wordlengths.

An empirical approach is used to scale the $\Phi(\cdot)$ and $\Phi^{-1}(\cdot)$ functions such that both inputs and outputs continue to span the dynamic range provided by the four-bit fixed-point representation. The choice set of scaled functions shown in (5.3) and (5.4) maximizes the effectiveness of a limited dynamic range while ensuring that the transformation is nearly constant under its own inverse. The scaling factors employ powers of 2 such that the multiplications and divisions are realized with simple binary shifts.

$$\Phi(x) = -\log\left(abs\left(\tanh\left(\frac{x}{16}\right)\right)\right) \tag{5.3}$$

$$\Phi^{-1}(x) = 4\left[-\log\left(abs\left(\tanh\left(\frac{x}{4}\right)\right)\right)\right]. \tag{5.4}$$

Since the arithmetic datapath comprises both $\Phi(\cdot)$ and $\Phi^{-1}(\cdot)$ functions, the quantized output of the nested function, $\Phi^{-1} \cdot \Phi$, is used to reflect the amount of nonlinear error introduced by the fixed-point implementations. The rounding, truncation, and non-linear effects are apparent in Figure 5-10, which compares the theoretical floating-point results with simulated fixed-point results with four-bit integer representations. Both (5.3) and (5.4) are plotted against the original function (3.18) in Figure 5-11. Both scaled functions show a higher utilization of the output range between 0 and 16, which corresponds to a 4-bit representation.

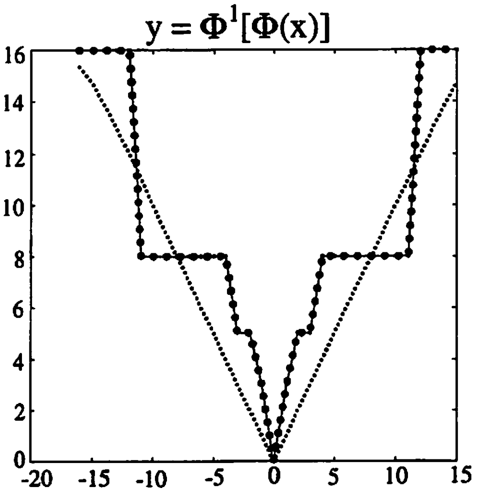


Figure 5-10. Comparison of floating point results (solid) and fixed point results (dotted) under the $\Phi^{-1} \cdot \Phi$ mapping.

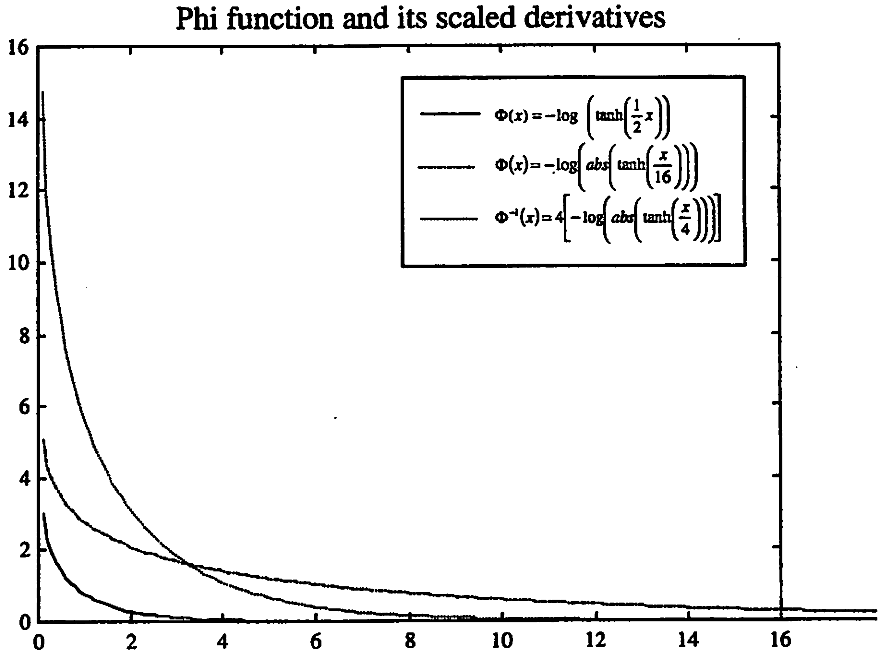


Figure 5-11. The original $\Phi(\cdot)$ function, and its scaled derivatives.

The scaled $\Phi(\cdot)$ and $\Phi^{-1}(\cdot)$ functions perform adequately when the average edge degree of the check nodes in an LDPC code is small (less than five). As the average edge degree of check nodes increases, the check-to-variable processing is required to evaluate the sum of a large number of $\Phi(\cdot)$ outputs. This leads to potential saturation of the adder outputs. The non-linear effect introduced by the saturation makes the subsequent marginalization operation (subtraction with the delayed inputs) irrelevant.

In previous reduced-complexity solutions, [85] [86], the computational requirements represented by (3.13) were replaced with simpler minimization operations. This removed the requirement for a lookup table implementation and the system is not subjected to truncation errors. The tradeoff in error performance, however, could be as high as 1dB [85].

The use of the minimization approximation is a good estimate for (5.2) when the difference between the two smallest input values is large. The value of $\sum_n \Phi(q_{nm})$ is dominated by $Max_n\{\Phi(q_{nm})\}$, which is the output $\Phi(Min_n\{q_{nm}\})$. This derivation is similar to the use of the maximum approximation, (5.3), commonly used in Viterbi decoders [4]. The approximation provides a good estimate when the difference between the input values is large.

$$\begin{aligned}R_{mn} &= \Phi^{-1}\left\{\left(\sum_{n'\in N(m)\backslash n}\Phi(Q_{n'm})\right)\right\} \approx \Phi^{-1}\left\{\left(\underset{n'\in N(m)\backslash n}{Max}\left[\Phi(Q_{n'm})\right]\right)\right\}\\ &= \Phi^{-1}\left\{\Phi\left(\underset{n'\in N(m)\backslash n}{Min} Q_{n'm}\right)\right\}\\ &= \underset{n'\in N(m)\backslash n}{Min} Q_{n'm}\end{aligned} \qquad (5.5)$$

$$\ln\sum_i \exp x_i \approx \max(x_i) \qquad (5.6)$$

Drawing from the above observation, the error introduced by the minimization function can be reduced by adding a correction term based on the difference between pairs of input values. This has the same connotation as the correction term used in the MAP decoders discussed previously. Let a particular check node *m* be adjacent to variable nodes *n*, *n*=1,2,...,*N*+1. The sum can be decomposed into a number of recurring pair-wise representations:

$$
\begin{aligned}
&\sum_{n'=1,2,\ldots,\eta(m)} \Phi(Q_{n'm}) \\
&= [\Phi(Q_{1m})+\Phi(Q_{2m})]+[\Phi(Q_{3m})+\Phi(Q_{4m})]+\ldots\_[\Phi(Q_{[\eta(m)-1]m})+\Phi(Q_{[\eta(m)]m})] \\
&= \Phi\{\Phi^{-1}[\Phi(Q_{1m})+\Phi(Q_{2m})]\}+\ldots+\Phi\{\Phi^{-1}[\Phi(Q_{[\eta(m)-1]m})+\Phi(Q_{[\eta(m)]m})]\} \\
&= \Phi\{\Phi^{-1}[\Phi(q_{1,1})+\Phi(q_{1,2})]\}+\ldots+\Phi\left\{\Phi^{-1}\left[\Phi\left(q_{1,\frac{\eta(m)}{2}-1}\right)+\Phi\left(q_{1,\frac{\eta(m)}{2}}\right)\right]\right\} \\
&= \Phi\{\Phi^{-1}[\Phi(q_{2,1})+\Phi(q_{2,2})]\}+\ldots+\Phi\left\{\Phi^{-1}\left[\Phi\left(q_{2,\frac{\eta(m)}{4}-1}\right)+\Phi\left(q_{2,\frac{\eta(m)}{4}}\right)\right]\right\}
\end{aligned}
\tag{5.7}
$$

where the intermediate variables $Q_{j,k}$ are defined as

$$
\begin{aligned}
q_{0,k} &= Q_{km} \\
q_{1,k} &= \Phi\{\Phi^{-1}[\Phi(q_{0,2k-1})+\Phi(q_{0,2k})]\} \\
q_{2,k} &= \Phi\{\Phi^{-1}[\Phi(q_{1,2k-1})+\Phi(q_{1,2k})]\} \\
&\vdots \\
q_{j,k} &= \Phi\{\Phi^{-1}[\Phi(q_{j-1,2k-1})+\Phi(q_{j-1,2k})]\} \\
&\vdots
\end{aligned}
\tag{5.8}
$$

The above equations are evaluated in a recursive structure of 2-input Θ operators (5.5) defined and approximated as follows.

$$
\Theta(x,y)=\Phi^{-1}[\Phi(x)+\Phi(y)]\approx Min\{x,y\}+f(x,y) \tag{5.9}
$$

The final right-hand term in (5.9), $f(x,y)$, is a correction term that alleviates the effects of fixed-point implementations. The implementation of $f(x,y)$ uses a look-up table with values derived using exhaustive search of the finite number of distinct pairs of fixed-point inputs. Figure 5-13 shows the correction terms plotted against the minimum of the two input integer values. Each plot corresponds to a set of (x, y) values with constant absolute difference, $|x\text{-}y|$.

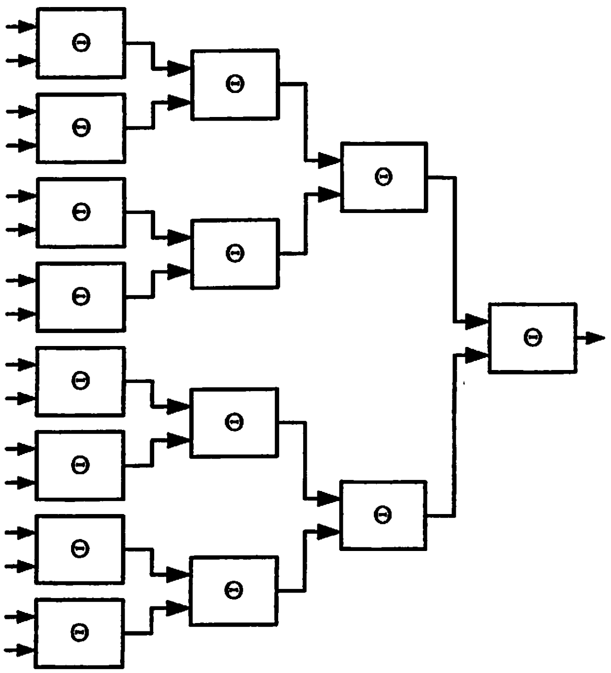


Figure 5-12. Tree structure of 2-input Θ operators evaluate check-to-variable messages.

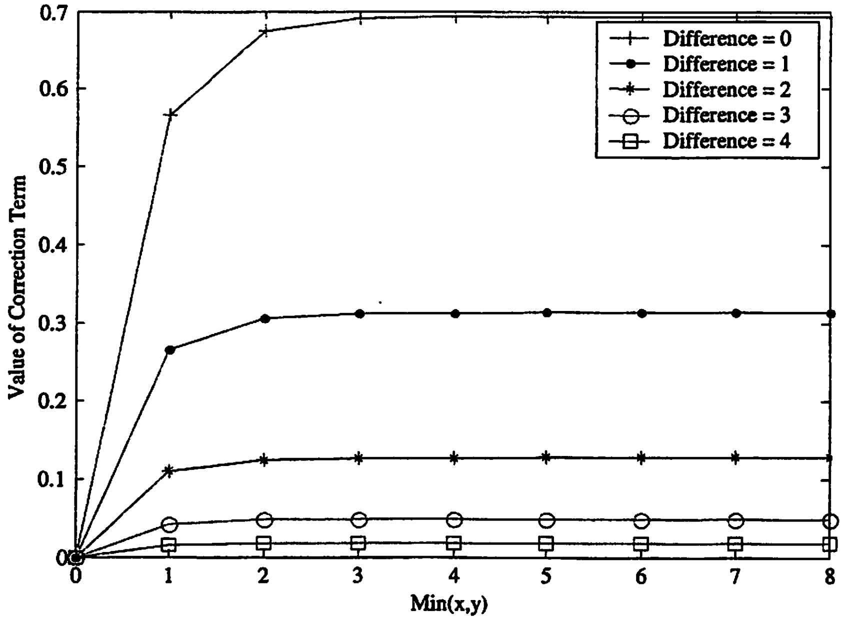


Figure 5-13. Correction terms $f(x, y)$ plotted against 3-bit integer inputs x and y.

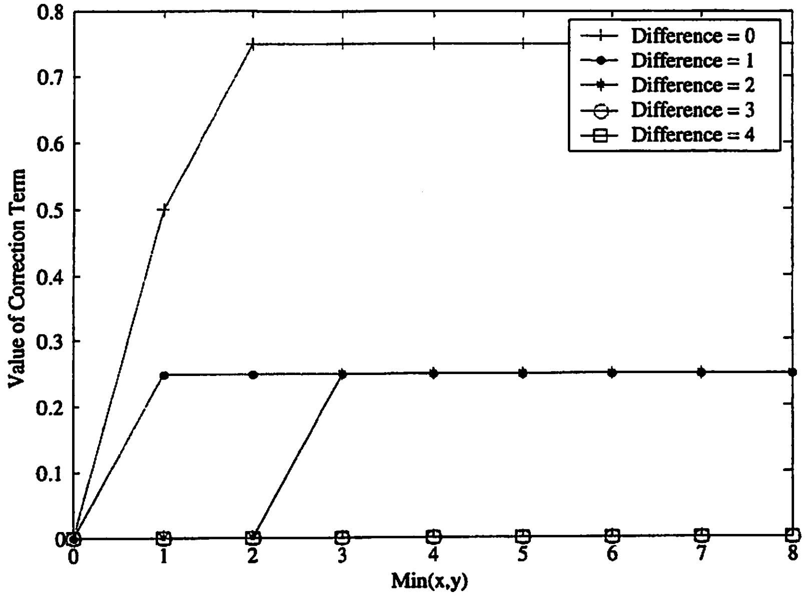


Figure 5-14. Evaluation of correction terms rounded to 0.25 levels.

The correction term is equal to zero when either of the inputs is zero. As the minimum of the two inputs is increased, the value of the correction terms increase rapidly, and then flattens out for the entire range of inputs. This behavior echoes the intentions of using a correction term based on the difference between pairs of inputs.

The correction terms are quantized to two fractional bits as shown in Figure 5-14. The exact quantization effect of the Θ approximation will vary between different LDPC codes. The discrepancy, though, is likely to be small since the estimates are derived from a precise evaluation of the Φ function. The behavior of the Θ approximation is simulated on a rate ¾, 4095-bit LDPC code based on finite field geometry [77]. Details of this particular code will be discussed in Chapter 6. The simulation results, shown in Figure 5-15, were obtained using the indicated number of bits for representation of messages, and with 2 additional binary bits that are only visible to the internal operation during the computation of messages, and truncated as the messages exit the computational block. It shows that the performance with 5-bit representations is comparable to the floating-point performance.

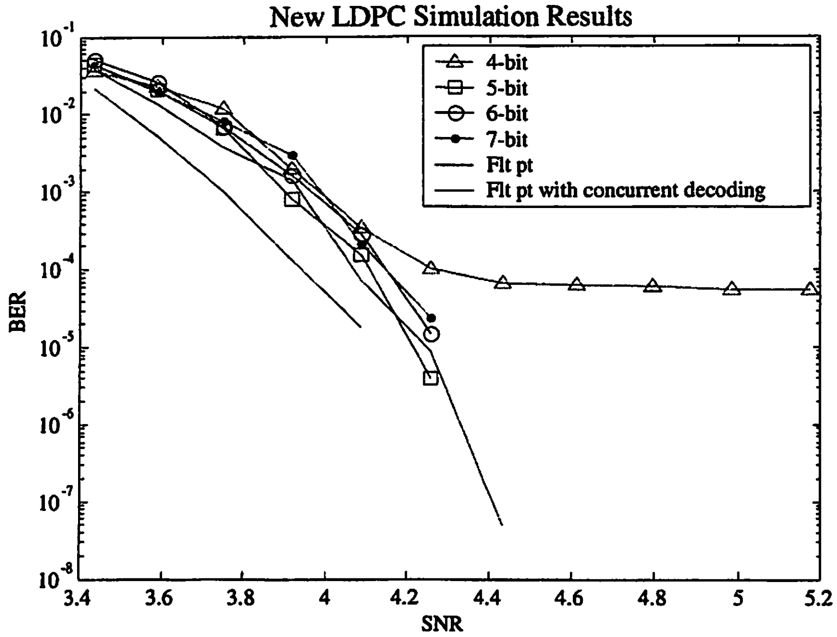


Figure 5-15. Simulation results with a rate 3/4 4095-bit LDPC code based on finite-field construction, with the Θ approximation.

5.4.2. Fixed-point LDPC decoder building blocks

The check node and variable node processing elements are the primary building blocks of an LDPC decoder. The delay, area, and power consumption of a processing element is dependent on the edge degree of the associated node. Assume an LDPC code with degree-3 variable nodes and degree-27 check nodes. Table 5-2 shows the implementation estimates of the processing elements implemented with 4-bit fixed-point messages. These values are obtained from synthesis of the described architectures using 0.13um standard cell low-leakage general-purpose CMOS library, and are optimized for either high throughput or low power.

TABLE 5-2
COMPLEXITY ESTIMATES OF PEs FOR LDPC APPLICATION

	Bitnode3 (Speed opt.)	Checknode27 (Speed opt.)	Bitnode3 (Power opt.)	Checknode27 (Power opt.)
Gate Count	466	1964	240	888
Delay (ns)	3	3.7	4.4	5.7
Area (um^2)	8517	32134	5972	17758
Number of FFs	0	108	0	108

5.5. Effects of code construction on implementation

In terms of implementation-related issues, recent development in LDPC code construction techniques can be differentiated along the lines of whether the code has a structured graph, whether the code has a uniform edge degree (regular codes), and the maximum edge degrees of both check and variable nodes

The method of LDPC construction based on density evolution [88] [93] has one of the best performances at only 0.0045dB away from the theoretical bound. An example of this construction method yields a rate-$^1/_2$ irregular code with a maximum variable degree of 100 and block size of 10^7 bits. It also requires an average of more than 1000 iterations to achieve the above decoding results. Practical implementations of decoders, however, desire a regular code with a small maximum edge degree in order to avoid detrimental arithmetic precision effects, and the complexity of collating a large number of inputs and outputs at the processing elements. A parallel decoder implementation with 10^7 processing elements will exceed realistic area constraints. A serial implementation is possible, but will suffer from extended decoding delays.

More recently, some structured and regular codes have been proposed. These include construction techniques based on properties of finite fields [77], mutually orthogonal Latin rectangles [114], improved Ramanujan graphs [96], and turbo product codes (TPC) [116]. The properties of these codes that lead to the implementation benefits are presented.

5.5.1. Finite field constructions

LDPC codes based on finite geometries avoid short cycles of length four in the underlying graph [77]. Short cycles are known to degrade the performance of the code [95]. An example 1023×1023 LDPC matrix was constructed from a 2-dimensional (Euclidean geometry) Galois Field, GF(2^5). Consecutive rows in this parity check matrix are cyclic shifts. In order to reduce its density, each column in the matrix was further expanded into four consecutive columns. This process is known as column splitting. The non-zero entries in each column in the initial matrix are rotated amongst four columns in the new matrix. This resulting 1023×4092 parity-check matrix is illustrated in Figure 5-16 with dots representing ones.

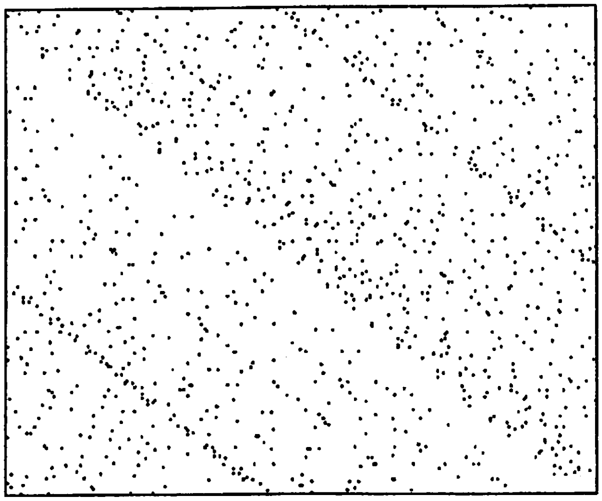
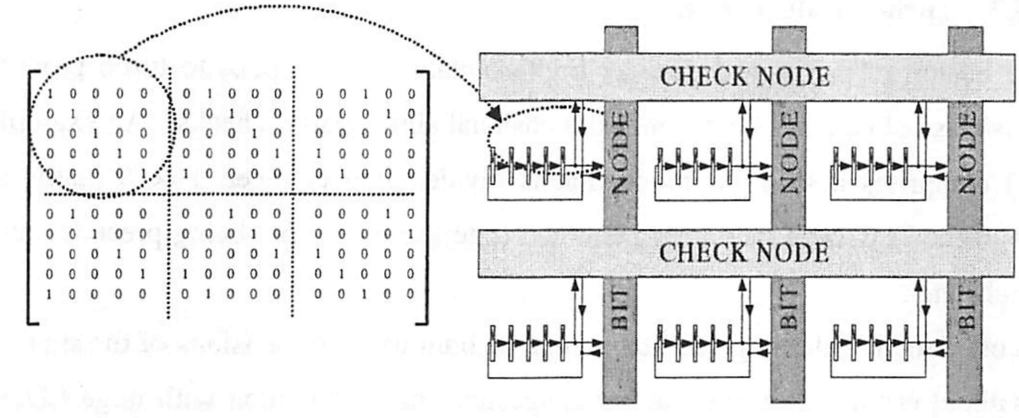


Figure 5-16. Rendition of 1023×4092 parity check matrix used in codes based on finite fields. Black dots represent non-zero entries.

The code comprises 3070 user bits and 1022 parity bits; a code rate of 3/4. The cyclic nature of the parity check matrix permits a serial decoder implementation where the messages are stored in shift registers, instead of memory. The use of shift registers has speed advantages, at the cost of higher area and power.

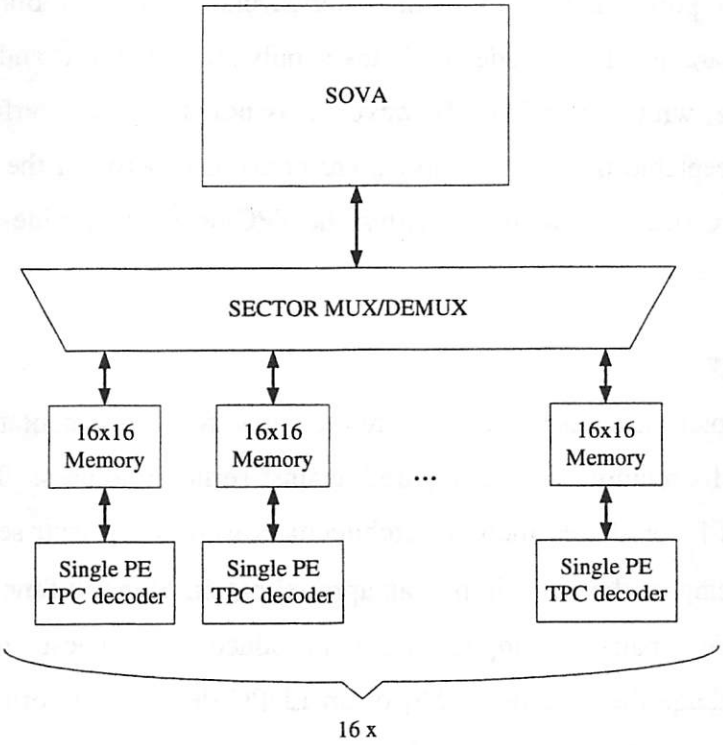
5.5.2. Latin rectangles and improved Ramanujan graphs

Two independent groups of researchers have introduced structured LDPC codes based on mutually orthogonal Latin rectangles [116], and modified Ramanujan graphs [96]. Both classes of constructions lead to parity check matrices that exhibit the natural partition lines that lead to smaller sub-blocks that enhance the ability to implement shared memory architectures. The example parity check matrix shown in Figure 5-17 shows these partition lines. The sub-blocks have square dimensions, and are either the zero-matrix, or some permutation of the identity matrix. The partitioning allows messages corresponding to a non-zero sub-block to share a memory element. Codes based on Latin rectangles also have additional benefit derived from the fact that consecutive rows within the sub-blocks are cyclic. This permits the memory to be implemented with high-speed shift registers.



$$\left[\begin{array}{ccccc|ccccc|ccccc}
1&0&0&0&0&0&1&0&0&0&0&0&1&0&0\\
0&1&0&0&0&0&0&1&0&0&0&0&0&1&0\\
0&0&1&0&0&0&0&0&1&0&0&0&0&0&1\\
0&0&0&1&0&0&0&0&0&1&1&0&0&0&0\\
0&0&0&0&1&1&0&0&0&0&0&1&0&0&0\\
\hline
0&1&0&0&0&0&0&1&0&0&0&0&0&1&0\\
0&0&1&0&0&0&0&0&1&0&0&0&0&0&1\\
0&0&0&1&0&0&0&0&0&1&1&0&0&0&0\\
0&0&0&0&1&1&0&0&0&0&0&1&0&0&0\\
1&0&0&0&0&0&1&0&0&0&0&0&1&0&0
\end{array}\right]$$

Figure 5-17. Shared memory decoder architecture for LDPC codes based on Latin rectangles.



528 cycles to complete each iteration of TPC decoding.

For single block-latency,
TPC decoder cycle period = 8 x SOVA cycle period

Figure 5-18. Architecture of TPC decoder partitions the code into 16×16 blocks.

5.5.3. Turbo product codes

The benefit of partitioned decoder implementation also applies to turbo product codes. This class of codes is based on 2-dimensional single parity checks. An example TPC [117] comprises a 4096-bit block that is divided into eighteen 16×16 matrices. Coded bits of the TPC were randomly permuted (interleaved) before being precoded and sent to the channel.

An efficient decoder implementation can be built to the dimensions of the smaller matrices without encountering the routing congestion that is common with large LDPC codes. The message-passing algorithm described in Chapter 3 can be implemented for each of these blocks using a single processing element (Figure 5-18). This allows the use of several double-ported memories in the interleavers, with a combined size of about 0.48mm^2. The parallel TPC decoder logic takes only about 0.2mm^2 and dissipates about 150mW at 1GHz, with 4 iterations. However, it is noted that the performance of TPC codes is only acceptable if global iterations are performed between the channel decoder and the outer TPC decoder; iterations within the TPC decoder provide insufficient BER performance.

5.6. Summary

This chapter has described the architectures for implementation of a LDPC decoder. Parallel structures were compared against serial structures. The properties of LDPC codes that favor shared memory architectures were also discussed. A number of complexity-reducing methods, including an approximation of the Φ function based on the difference between pairs of inputs were introduced. These methods do not fundamentally change the definition [56] of an LDPC decoding algorithm. In contrast, the next chapter will discuss the VLSI and performance implications of altering the decoding schedule, and propose further complexity-reducing techniques, which will change the underlying message-passing algorithm.

6. PROCESSING SCHEDULES OF LDPC DECODERS

Considerations for the processing schedules of an LDPC decoder are only applicable to serial structures. These architectures comprise one or more processing elements in a uni-memory or shared-memory environment. In these implementations, a large number of computations need to be scheduled onto a limited number of processing elements. To address the difficulty with the large memory requirement of serial decoders, this chapter presents a new decoding method, which uses a different schedule and modified decoding computations. These ideas are applied to LDPC codes based on random construction as well as finite-field constructions [77].

This chapter frequently uses the notations, $\mu(n)$ and $S(k)$, to denote the set of check nodes adjacent to variable node n, and the subset of check nodes that are processed at time step k, respectively. Indeed, a decoding schedule can be defined as a sequence of $S(k)$, and this sequence should at least cover the entire set of check nodes within a finite number of cycles.

6.1. Original concurrent schedule

The original concurrent decoding schedule is illustrated in Figure 6-1. The underlying graph of the example code comprises 10 variable nodes (labeled as circles) and 3 check nodes (labeled as squares). In each iteration, the decoding process evaluates messages corresponding to all edges in the graph. These messages are passed concurrently between the two classes of nodes.

Serial decoders have a limited number of processing elements that can only process a subset of the check nodes or variable nodes at a time. This results in decoding latencies ranging from several hundred cycles to thousands of cycles. In the meantime, the intermediate values are stored in memory. Due to the unstructured nature of most LDPC codes, the decoding must complete the computations in the current iteration before proceeding to the next iteration. This causes a large memory requirement in the decoder.

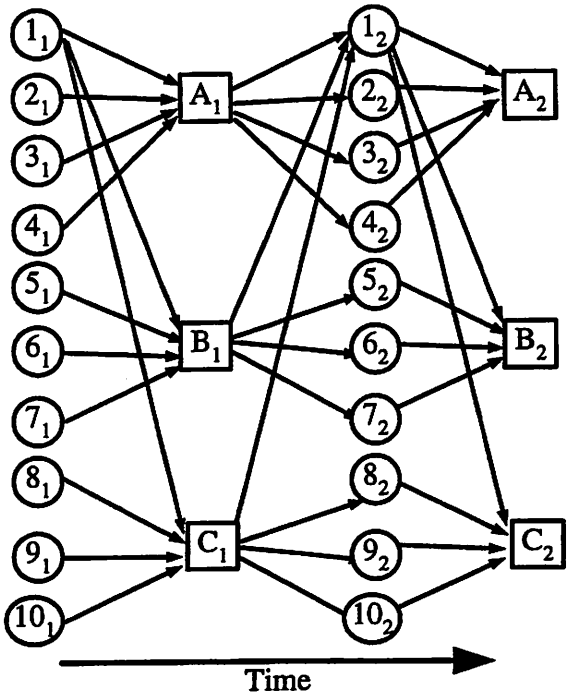


Figure 6-1. Concurrent decoding schedule of the message passing algorithm; circles represent variable nodes; boxes represent check nodes.

6.2. New staggered schedule

A new, staggered, schedule processes only a limited subset of the check nodes. Similar to the original algorithm, check node m computes messages R_{mn}, $n \in V(m)$, according to (3.17). However, variable node n computes messages Q_{nm} with (6.1). This approximates the update equation defined previously in (5.1) by omitting the rightmost term, $R_{mn'}$. The effect of this simplification is minimal when the ratio $R_{mn} / \sum_{m \in \mu(n)} R_{m'n}$ is small. This occurs with high probability when the average edge degree of the variable nodes is high.

$$Q_{nm} \approx \ln\left[\frac{p_i(1)}{p_i(0)}\right] + \left(\sum_{m \in \mu(n)} R_{m'n}\right) \tag{6.1}$$

Each variable node, n, is therefore, associated only with a single message $Q'_n(k)$, which is broadcasted to the subset of active check nodes, $S(k)$, at step k. Each message, $Q'_n(0)$ for n=1, 2, ..., N is initialized with the input LLR of the corresponding variable n, and updated according to 5.7 at the end of each step.

$$Q'_n(k) = Q'_n(k-1) + \sum_{m' \in \{S(t) \cap \mu(n)\}} R_{m'n} \tag{6.2}$$

An implementation of the staggered schedule only needs to store one message, $Q_n(k)$, for each variable n. This compares favorably with the concurrent schedule where a list of messages Q_{mn}, $m \in \mu(n)$, need to be stored for each variable node n. For typical LDPC codes where the number of edges in the graph ranges between 4 to 8 times the number of variables in the code-block, this schedule provides more than 75% savings in memory requirement.

Each iteration of the staggered LDPC decoding is defined as one complete cycle through all the sequential parity checks in the graph. This ensures that the number of messages processed in an iteration of decoding is the same in both types of schedules. As such, the decoding arithmetic complexity is similar.

Figure 6-2 shows the staggered decoding schedule with only one check-to-variable processing element active per step, while Figure 6-3 illustrates the use of one sector of memory and one constraint check arithmetic unit, per code block processed. A full throughput, K-iteration LDPC decoder, demultiplexes each input block of data into one of the K copies of memory and computation unit pair. This removes the need to move large amounts of data through memory, at a slight cost of control mechanisms to perform the multiplex operations.

The empirical results of comparisons performed between the concurrent and staggered decoding schedules with $|S(t)| = 1$ are presented. Two LDPC codes with parity check matrices based on random construction and finite-field construction are evaluated. The random code comprises a 512×4608 parity check matrix, which is obtained by appending 4 random 128x4608 matrices with column weight of 1 and average row weight of 36. Each block of data comprises 4096 user bits and 512 parity bits; a code rate of $^8/_9$.

For a fair comparison, an LDPC code was generated from a 2-dimensional finite-field of GF(2^5). This code underwent a process of column-splitting to obtain a block size of approximately 4000 bits. The various column splitting factor that were considered are listed in Table 6-1. It shows that the 2D×GF(2^5) construction results in the only feasible parity check matrix, which does not have a column weight of one.

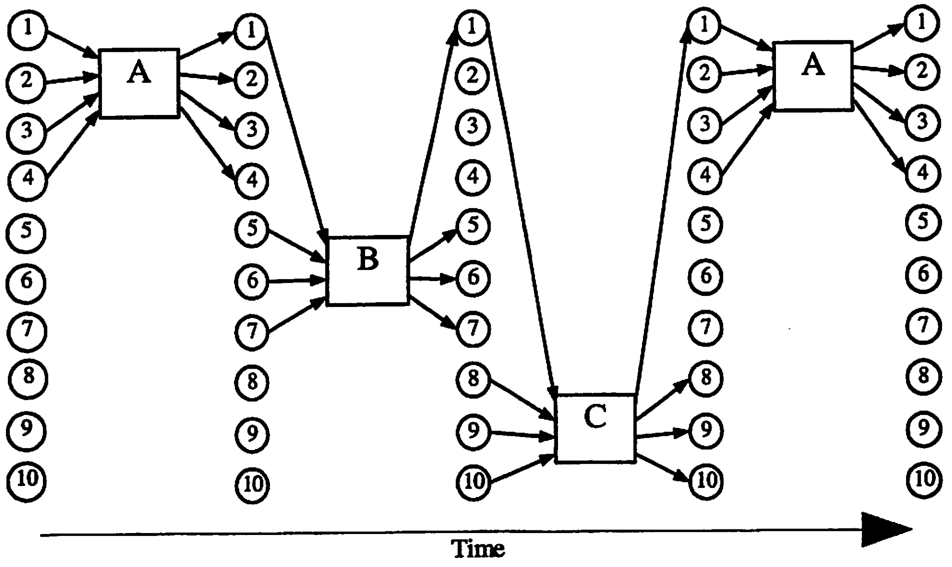


Figure 6-2. Staggered decoding schedule of the modified message passing algorithm with $|S(t)|=1$.

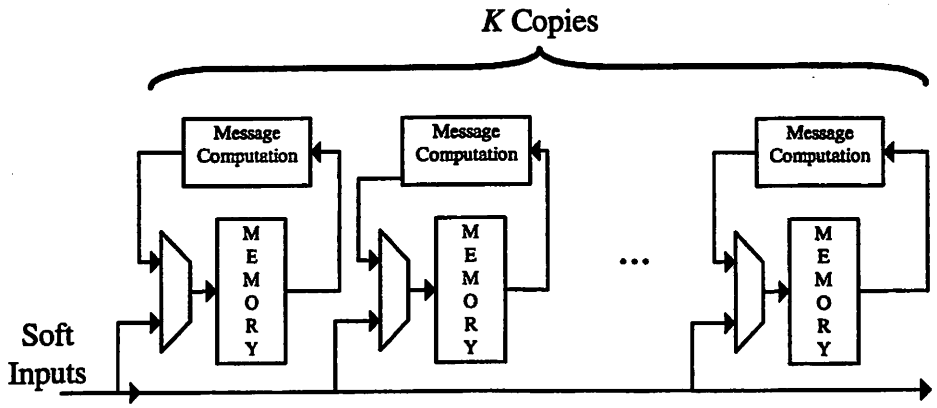


Figure 6-3. Architecture for random LDPC decoder with K iterations of staggered schedule.

TABLE 6-1.
LDPC CODES CONSIDERED FOR MAGNETIC RECORDING APPLICATIONS

Finite Field Construction	Matrix Dimension	Column Split Factor	PC Matrix Dimension.	Row Wt. × Col Wt.
2D× GF(2^5)	1023 × 1023	4	1023 × 4092	32×8
2D× GF(2^4)	255 × 255	16	255 × 4080	16×1
3D× GF(2^3)	511 × 511	8	511 × 4088	8×1

6.3. Simulation analysis

BER experiments are performed with a binary antipodal Gaussian channel and LLR inputs. The performance is evaluated through 1, 3, and 5 iterations of message passing decoding. These iteration counts represent the expected number of iterations that are realizable in serial decoder implementations, with considerations given to logic density and overall memory requirement. The empirical results of comparisons performed between the concurrent and staggered decoding schedules with $|S(t)| = 1$ are presented.

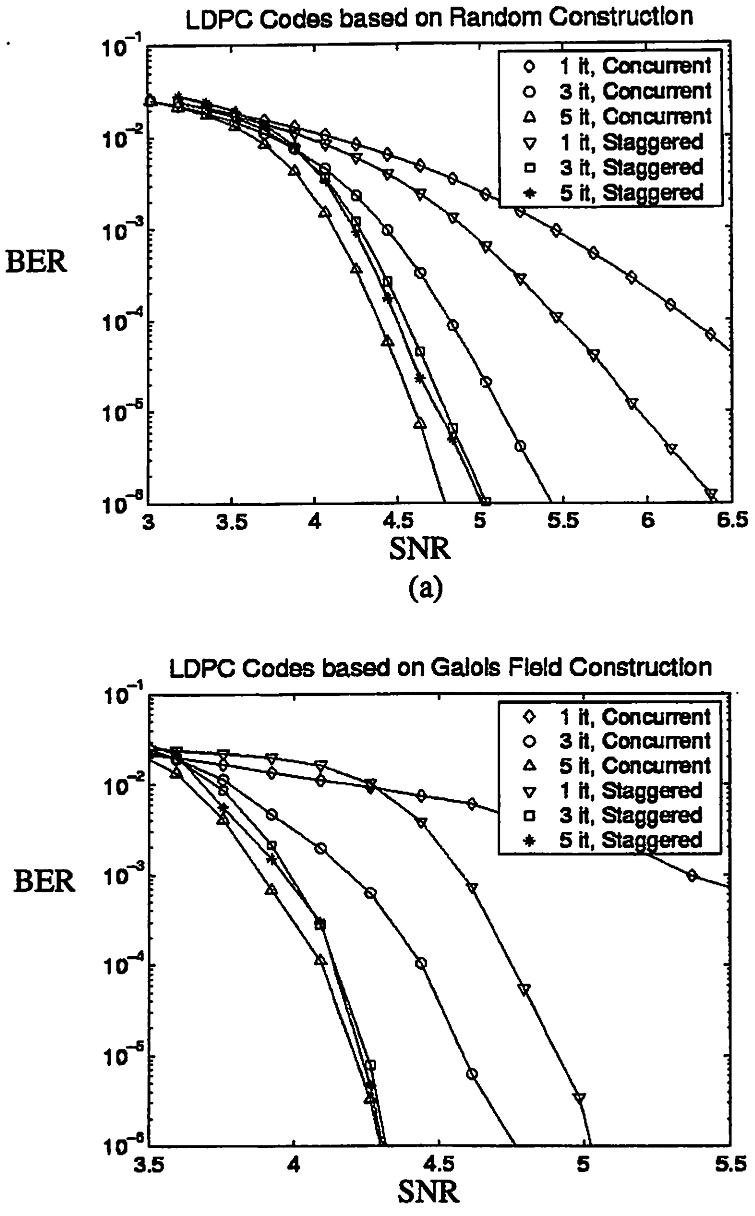
In order to determine the BER at a given SNR, the simulation model adds Gaussian noise to a maximum of 30,000 blocks of data. The BER figure is continuously evaluated over the total number of decoded message bits until it has converged within 1% of the eventual value. The SNR definition in (5.9) represents the user and code bit energies with E_B and E_C respectively, and the noise variance with σ^2.

$$R = \frac{Num.ofUserBits}{Num.ofCodeBits};\ E_c = R \cdot E_b \tag{6.3}$$

$$SNR = 10 \cdot \log\left(\frac{E_b}{N_0}\right) = 10 \cdot \log\left(\frac{E_c}{2R\sigma^2}\right) \tag{6.4}$$

The results of performing 1, 3, and 5 iterations with the concurrent decoding schedule and the staggered decoding schedule, for both the random and finite-field-based codes are plotted in Figure 6-4. With either code, the concurrent decoding schedule yields a steady improvement in error performance with each increase in number of iterations.

Figure 6-4a shows that using the random code with 3 iterations, the staggered decoding schedule achieves 0.4dB improvement over the concurrent decoding schedule, measured at BER=10^{-5}. With 5 iterations, it results in less than 0.2dB degradation. Likewise, Figure 6-4b shows that using the code based on finite-fields, the staggered decoding schedule achieves 0.3dB improvement with 3 iterations, and has less than 0.1dB degradation with 5 iterations, both measured at BER=10^{-5}.



(a)

(b)

Figure 6-4. Simulation results from random codes (a), and GF codes (b) with concurrent vs. staggered decoding schedule.

In the concurrent decoding schedule, messages from each bit node are relayed to the closest neighbors in a single iteration. Figure 6-1 shows that the path for a message from bit node 4 to bit node 10 would require 2 iterations $(4_1 \rightarrow A_1 \rightarrow 1_1 \rightarrow C_2 \rightarrow 10_2)$. On the other hand, the staggered decoding schedule has the capacity for messages to be relayed to more distant nodes; in the example given in Figure 6-2, a path exists from bit node 4 to bit node 10 within a single iteration $(4 \rightarrow A \rightarrow 1 \rightarrow B \rightarrow 1 \rightarrow C \rightarrow 10)$. The further reaching schedule is expected to converge faster, thus the better performance with low number of iterations.

The staggered decoding schedule is an approximation of the belief propagation algorithm [40], as noted earlier. The use of a running sum $Q'_n(k)$ creates cycles in the underlying graph. The effects are noticeable with increased number of decoding iterations. Figure 6-2 shows a cycle, which starts and ends at check node A ($A \rightarrow 1 \rightarrow B \rightarrow 1 \rightarrow C \rightarrow 1 \rightarrow A$). These cycles limit the performance of the code, as is evident in Figure 6-4. The staggered schedule has no BER performance advantage over the concurrent decoding schedule if the decoding is performed with more than 5 iterations. However, in applications such as high-throughput magnetic recording, practical constraints will limit the number of iterations, and staggered decoding provides a key to significant reduction in implementation complexity for the serial decoder architecture.

6.4. Pipelined processing elements

Using the original concurrent decoding schedule with both parallel and serial architectures, pipelining increases the effective throughput without causing any difference in the decoding algorithm. A parallel LDPC decoder flip-flops constantly between two sets of processing elements. At a macroscopic level, this implies that half the processors are idle at any instance. This waste of resources can be eliminated by introducing an N-stage pipeline that fills the collective delay through the two sets of processing elements. This will also allow a decoder to operate on N independent blocks of bits simultaneously. In a serial LDPC decoder, each processing element typically calculates hundreds of messages through each iteration of decoding. This implies a delay of a few hundred cycles. The cycle time is determined by the delay of the processing element. Using pipelined processing, the delay is reduced to a fraction of its original value. Although the added latency increases the total number of cycles, the total delay is significantly reduced.

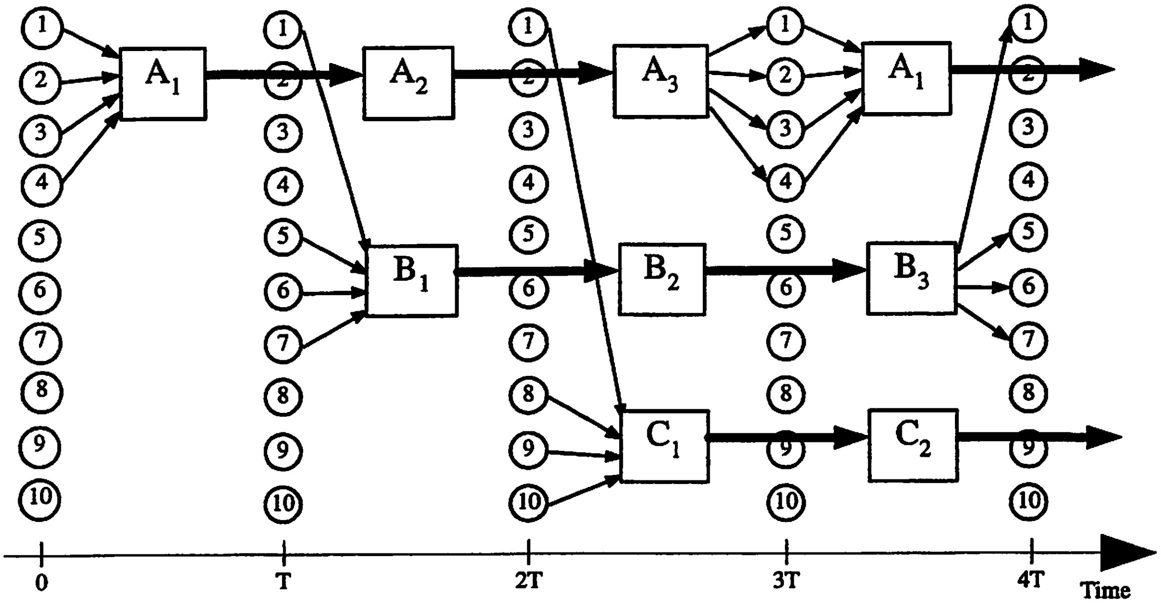


Figure 6-5. Staggered decoding schedule using one processing element with a 3-stage pipeline.

As a numerical example, let each processing element calculate 200 messages through each iteration of decoding. Furthermore, assume that the critical-path delay, which determines the cycle time, is 5ns. Thus, the total delay is 1μs. Suppose that a 6-stage pipeline is introduced such that the critical path delay is reduced to 1ns after accounting for overhead in additional register-setup time. The entire processing will now complete in 206ns.

Using the previously proposed staggered decoding schedule, which is only applicable for serial architectures, the pipelining of the processing elements will result in deviation of the algorithm. Figure 6-5 shows the decoding schedule using a single processing element with a pipeline of 3 cycles. *A*, *B*, and *C* represent three check nodes. A_i refers to the i^{th} stage of the pipeline in the check node processing corresponding to check node *A*. The ten numerical nodes represent the variable nodes. Each node, *k*, $1 \leq k \leq 10$, is associated with a message Q'_k. Note that the value of Q'_1 is not updated until time 3*T*. Hence, the same message is sent from node 1 to A_1, B_1, *and* C_1 in the first three cycles.

In practice, such occurrences are rare with LDPC codes with large girth and random connectivity. The effects of a pipelined processing with the staggered schedule is evaluated on both LDPC codes previously mentioned, with parity check matrices based on random and 2-dimensional GF(2^5) constructions. The depth of the pipeline is varied between 1 and 50. In all cases, there was no indication that pipelined processors cause any difference in the BER performance.

6.5. Summary

The arithmetic computational complexity of the LDPC decoder is misleadingly low, when compared against existing Turbo decoders. Both serial and parallel implementations of the LDPC decoder have to address the issue of large memory requirement and interconnect congestion respectively. This arises from the random and often sparse structure of LDPC codes, in general. These two factors are mutually exclusive, and choices between serial or parallel implementation is tied with tradeoff between memory or interconnect concerns.

The proposed staggered decoding schedule decouples the memory requirement from its dependency on the number of edges in the graph. It is recognized that the staggered decoding schedule will not achieve the same results as LDPC decoding under belief propagation. However, it results in significant reduction in implementation complexity. In consideration that practical power and area constraints are likely to limit the number of iterations to three, the staggered decoding schedule performs systematically better than the concurrent decoding schedule.

7. PHYSICAL IMPLEMENTATIONS

This work has implemented the high throughput decoder architectures described in the previous chapters on a number of platforms. Details of the design flow for the successful implementations on FPGAs and ASIC hardware are presented, as well as the tests conducted to verify the designs. The physical design methodologies are derived from automated design flow projects conducted at the Berkeley Wireless Research Center. These are the Simulink to Silicon Hierarchical Automated Flow Tool (SSHAFT) [74] and its successor, the Berkeley Emulation Engine (BEE) FPGA-ASIC design flow [99].

The results shown here demonstrate that the performance achieved on the various platforms can differ by orders of magnitude. ASIC implementations of the SOVA and LDPC decoders were able to achieve line rates between 500Mb/s and 1Gb/s with 0.18μm and 0.13μm standard CMOS processes respectively. A MAP decoder mapped onto an FPGA demonstrated a throughput of 10Mb/s; while the same LDPC decoder implemented on FPGA has a maximum operating throughput of 50Mb/s. Nevertheless, these rates remain significantly higher than the tens of kb/s simulation throughputs that were obtained on general-purpose microprocessors platforms.

Each physical implementation was realized with the help of design automation that provided rapid capability for comparing several architectures along the lines of hardware specific considerations. A large amount of scripting automata ensures the synergy of a suite of commercial tools that supply a preliminary layout, from which comparison statistics in speed, area, and throughputs of implementation can be obtained. The initial design parameters were centered on a pool of heuristically well-accepted values, which ensure that the tools have a high probability of success. For example, the default core utilization density, defined as the ratio of area of active transistors to the area of the ASIC core, is set to 50% to avoid routing congestion. In general, these values lead to implementations that are sub-optimal. However, the designer maintains the ability to modify the scripts at all levels. This permits experimentation with key design parameters. As the design matures, the focus is shifted towards the optimization of each descending hierarchy in this mostly top-down methodology.

Sections 7.1 and 7.2 will discuss the turbo decoder implementations based on the MAP algorithm and SOVA, respectively. The MAP decoder was realized on an FPGA, while the SOVA decoder was realized on an ASIC. Likewise, the FPGA and ASIC implementations of a 4092-bit LDPC decoder will be elaborated in Section 7.3.

7.1. FPGA implementation of a MAP decoder

A pair of MAP decoders applying the BCJR algorithm was implemented on the Berkeley Emulation Engine (BEE). The BEE is a real-time hardware comprising 20 high-density Xilinx Virtex-E FPGAs operating on a backbone of interconnect fabric. The collective processing power is capable of executing over 600 Giga-operations per second and emulating a 10 Million ASIC gate equivalent system. The system design tool is a combination of Simulink from Mathworks and Xilinx System Generator toolbox [100]. The latter is a library of hardware-like components that model the cycle-accurate and bit-true behavior of the hardware. It comprises low-level components, such as adders and multipliers, that also accept hardware specific parameters such wordlengths, binary-point positions and latencies.

The objective of the BEE design flow is to enable algorithm designers to use system-level design tools to create hardware designs that can be both implemented on a BEE system and as an ASIC. The system design tool is a combination of Simulink from Mathworks and the Xilinx System Generator toolbox. The designer is provided with a set of hardware-like components that model the cycle-accurate and bit-true behavior of the hardware. The block library comprises low-level components, such as adders and multipliers. Hardware specific parameters, such as word-lengths, rounding/truncation methods, and latencies, are provided as design parameters to these building blocks. The Xilinx System Generator compiles the Simulink description into a VHDL structural netlist. The same netlist is used for both FPGA and ASIC implementations.

A 16-state windowed MAP decoder with the architecture described in Section 4.3 was implemented together with a E^2PR4 channel. The decoder was matched to the E^2PR4 target. The memories were realized using shift-registers, and control signals were obtained from a number of linear counters. The design occupied 6000 slices, or 31% of a single FPGA chip. The minimum period was 95ns, corresponding to a decoding throughput of 10.5Mb/s.

7.2. ASIC implementation of a SOVA decoder

Two 8-state, 7-bit soft output Viterbi decoders matched to an enhanced partial response class-4 (EPR4) channel and a rate-8/9 convolutional code has been implemented in a 0.18μm CMOS technology. The throughput of the decoders is increased through architectural transformation of the add-compare-select recursion, with a small area overhead. The survivor-path decoding logic of a conventional Viterbi decoder register-exchange is adapted to detect the two most-likely paths. The 4mm^2 chip has been verified to decode at 500Mb/s with 1.8V supply. These decoders can be used as constituent decoders for Turbo codes in high performance applications requiring information rates that are very close to the Shannon limit.

Both decoders have the same architecture, but are matched to different generator polynomials; the *SOVA_EPR4* decoder is matched to an EPR4 channel with a $(1 \oplus D)^{-1}$ precoder while the *SOVA_13* decoder is matched to an Octal(13) generator. The equivalent generator polynomials are $\frac{1}{1 \oplus D}(1+D-D^2-D^3)$ and $(1 \oplus D^2 \oplus D^3)$ respectively.

7.2.1. Design flow

The implementation of the SOVA decoders employed the SSHAFT design flow that performs direct mapping of signal processing algorithms into integrated circuits. The design entry uses a dataflow graph editor, Simulink from Mathworks because of its familiarity to algorithm designers. Figure 7-1 shows a screen capture of part of the dataflow graph. Floating-point primitives are swapped with fixed-point primitives to explore finite word-length effects. These fixed-point types express the signal decisions for the automated flow and are needed in order for the simulation to be bit-true. The discrete-time model was used because it can be made cycle-accurate with respect to the hardware, which is necessary to verify that hardware generated by the flow faithfully represents the functional description.

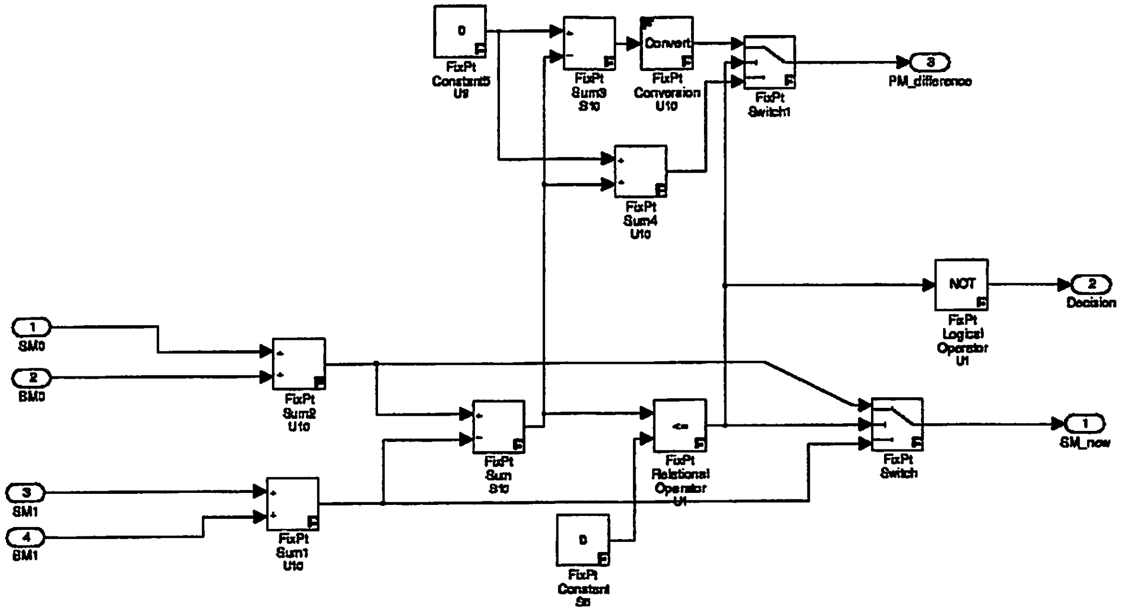


Figure 7-1. Dataflow graph of the ACS of a SOVA decoder design in Matlab Simulink^{TM}.

The data-path logic is constructed with the help of a commercial data-path generator, Synopsys Module Compiler^{TM}. The description language for this generator has a highly-restrictive set of operators that is very similar to the set of primitives used by the dataflow simulator, which minimizes the effort involved in creating new library elements. This approach does result in a second functional specification for a block that must be made equivalent to its corresponding dataflow graph model. This is accomplished with cross-check simulations between the dataflow graph and RTL code created by the data-path generator.

A floorplan is required in addition to the above functional description in order to capture the use directives such as pin placement and relative placement of hierarchical macros. The initial skeleton floorplan is generated by the automated flow, and designers, using commercial physical design tools, are allowed to edit the floorplan, placing instances and boundary pins using simplified commands. Once the floorplan is completed, the design can be routed and characterized by the automated flow.

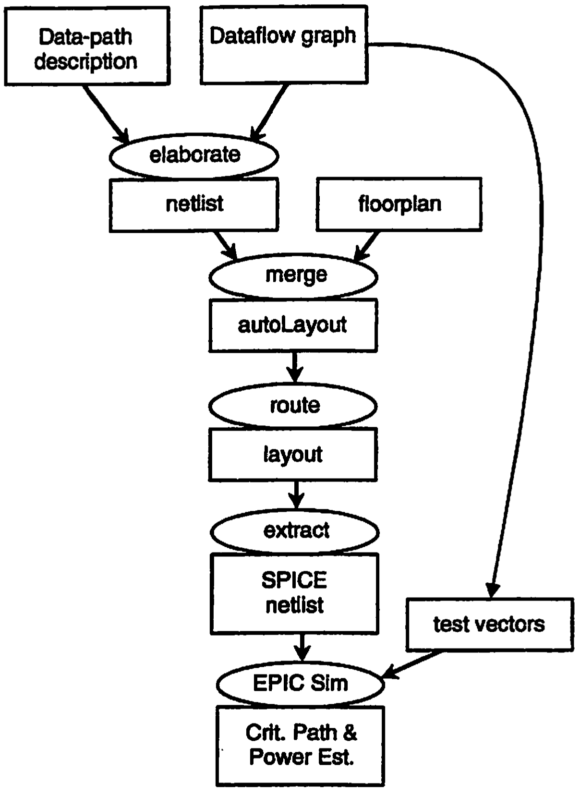


Figure 7-2. High-level dependency graph for the automated design flow [74].

Estimates are obtained after the flow has synthesized, routed, performed parasitic extraction and run EPIC PathMill and PowerMill simulations on each macro using test vectors generated from the dataflow graph. This information is stored in order to provide fast estimates of performance and required resources and ultimately should be available from within the dataflow graph editor to provide information for reuse. Figure 7-2 shows a summary of the automated design flow.

7.2.2. Design parameters

The required wordlength of each SOVA decoder was determined by comparing the performance difference between floating-point computation and several fixed-point types. 7-bit sign-magnitude signals were necessary to provide less than 0.1dB degradation of required signal-to-noise ratio at a BER of 10^{-5} after 5 iterations.

7.2.3. Physical Tests

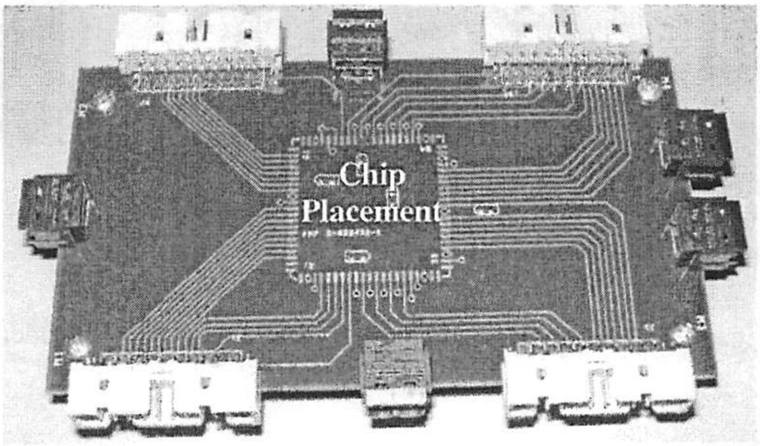


Figure 7-3. Printed circuit board used for testing of SOVA decoders.

The silicon die was mounted and wire-bonded to an 84-pin windowed ceramic J-lead chip carrier (JLCC) package. The package fitted into a socket, which was surface-mounted onto a custom-built test board. Figure 7-3 is a photograph of the four-layer printed circuit board with 75 discrete components. The board design made use of surface-mounted decoupling capacitors on the bottom plane such that operating frequencies up to 100MHz are sustainable. The 20-pin connectors on the four outer corners permit communication with a networked logic analyzer. This section describes the physical tests that were performed with these apparatus.

7.2.3.1.Logical verification

The logical verification of the SOVA chip used the setup depicted in Figure 7-4. Using a workstation, a range of test vectors was generated by running random numbers through the Simulink description of the SOVA design. These vectors were then downloaded onto the logic analyzer, which presented the input pattern to the test board. A pattern generator supplied the global clock. Outputs from the test board were sampled and compared with the test vectors. This test was performed at 50MHz and achieved logical verification of the test chip.

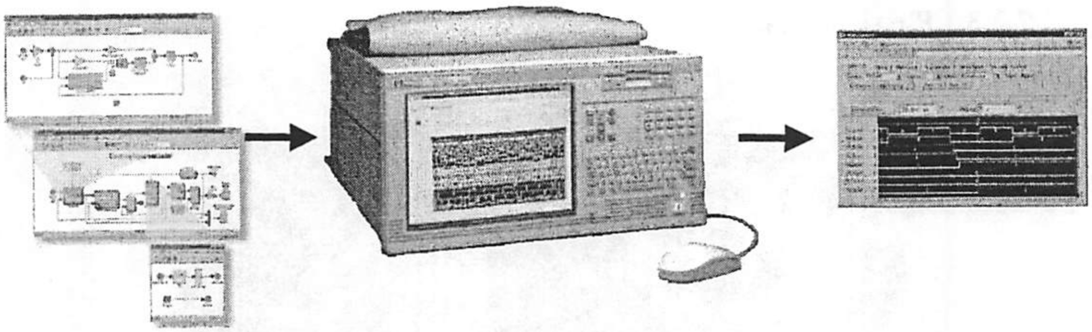


Figure 7-4. Setup for logical verification of SOVA decoders

7.2.3.2.Speed test

The decoder is fully synchronous with respect to a single global clock. The architectural evaluation of the SOVA decoder identifies the ACS recursion as the throughput bottleneck of the decoder design. Therefore the critical path delay through the ACS recursion determines the minimum cock period.

In order to prevent false deterioration of the timing results, the on-chip critical-path delay was evaluated with minimal intrusion provided by the triple clocking scheme shown in Figure 7-5. A global clock signal, *CLK_GLOBAL*, is delivered throughout the chip, and drives all sequential elements during normal operation. Two additional signals, *CLK2* and *CLK3*, can be selected to sample the inputs and outputs of the ACS, respectively. These secondary signals are only delivered to the ACS block as the rest of the decoder continues to be synchronized to *CLK_GLOBAL*. The three clocks are distributed with separate networks as shown in Figure 7-6. This minimizes any parasitic effects on the primary clock tree for distribution of *CLK_GLOBAL*.

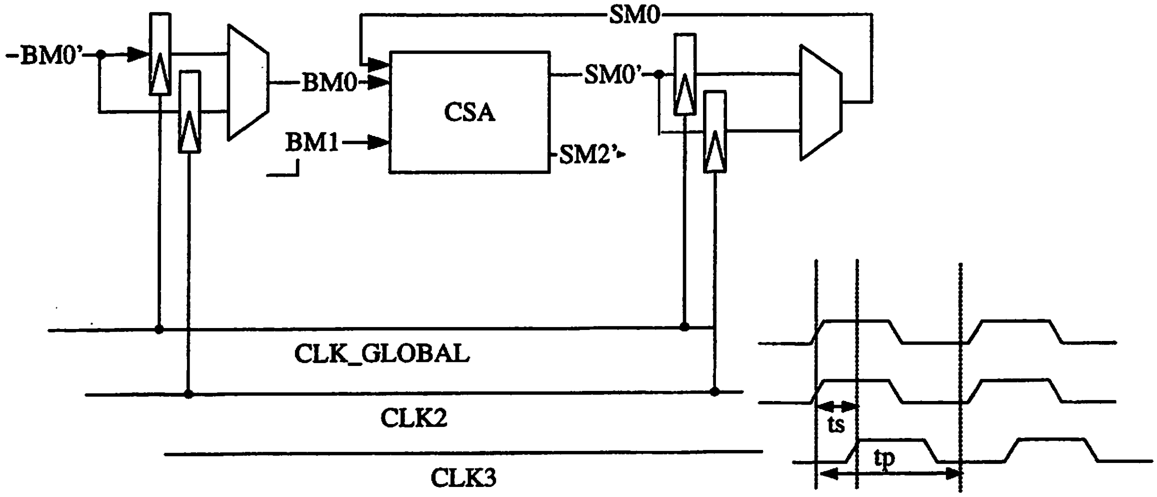


Figure 7-5. Test structure around CSA for critical path analysis.

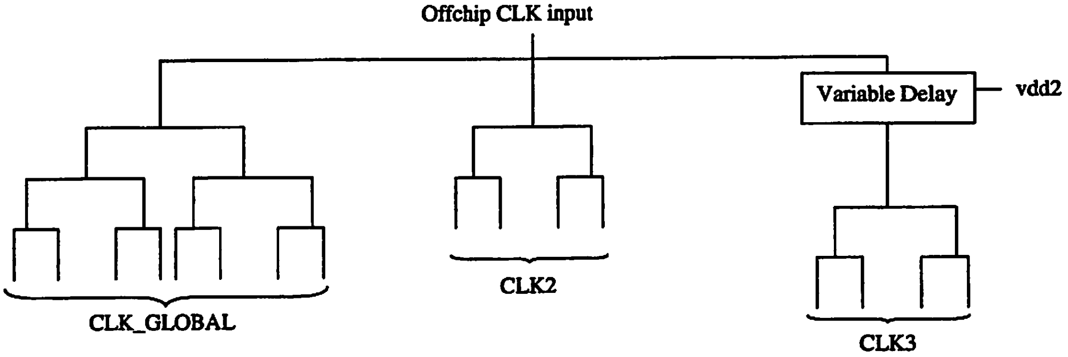


Figure 7-6. Separate clock trees reduce parasitic effects of additional speed test requirements

The minimum skew between *CLK2* and *CLK3* must be greater than the critical-path delay in order to maintain correct operation. The error margin for this design is specified at 10%. At the targeted 500Mb/s operation, the speed test requires clock edges that can be reproduced with 200ps resolution. A variable-delay line introduced deliberate skew between CLK2 and CLK3. The delay was controlled by the voltage of the power supply, shown as *Vdd2* in Figure 7-7. In order to account for the parasitic effects of wiring delays and clock switches, an RC-layout extraction of the three clock trees (*CLK_GLOBAL, CLK2* and *CLK3*) followed by transistor-level Spice simulations provided the expected delay values across a range of *Vdd2* values. This is shown in Table 7-1.

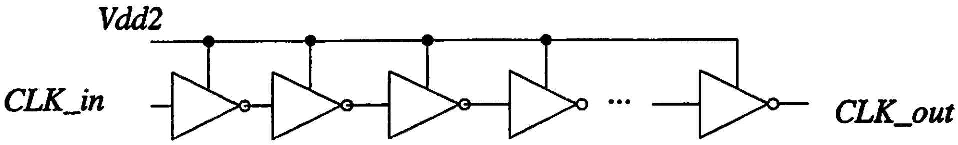


Figure 7-7. Variable delay controlled by *Vdd2*

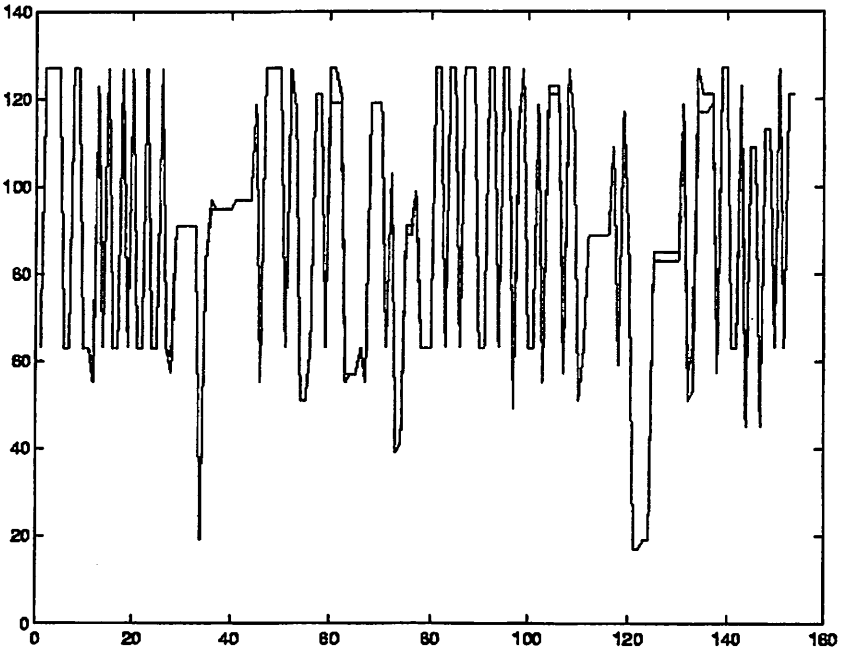


Figure 7-8. Plot of correct output vectors (blue) and actual vectors output by chip (red) at skew delay of 1.6ns.

TABLE 7-1.
VARIABLE DELAY INTRODUCED BY *VDD2*

Vdd2	Skew (ns)
1.2	2.0
1.4	1.8
1.6	1.6
1.8	1.5
2.0	1.5

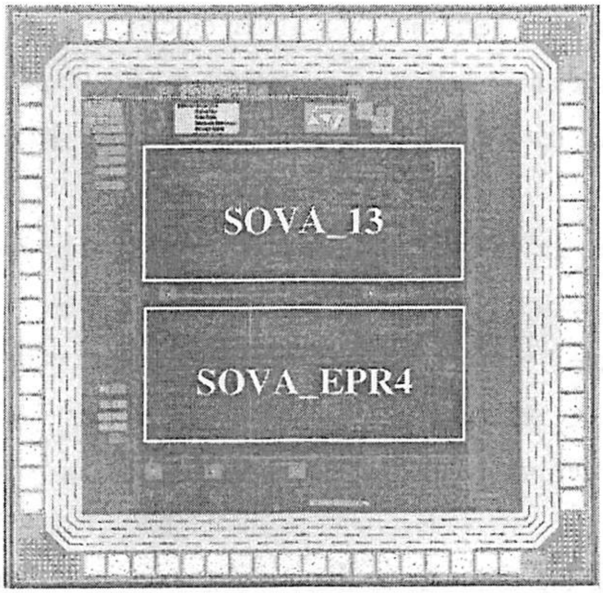


Figure 7-9. Die micrograph.

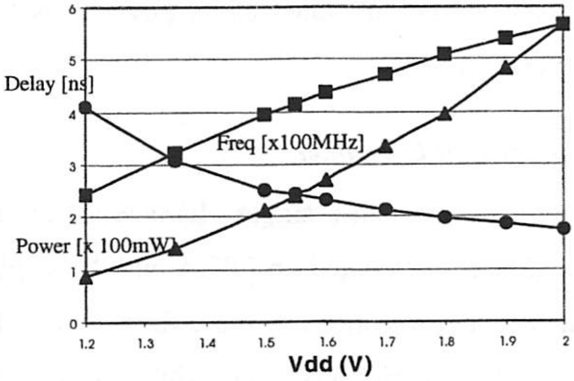


Figure 7-10. Performance of EPR4 SOVA decoder.

TABLE 7-2
SUMMARY OF SOVA CHIP IMPLEMENTATION

Decoder Type	SOVA_EPR4	SOVA_13
Number of States	8	8
Transistor Count	164K	174K
Core Area	1mm × 0.5mm	1mm × 0.5mm
Speed	500Mb/s	500Mb/s
Avg. Power @ 500MHz	395mW	400mW

The speed tests were performed at a global clock frequency of 20MHz, using the same setup as the verification test previously described. The output vectors from the chip were compared against the simulated vectors. The value of *Vdd2* was increased from 1.4V to 2.0V. As the delay constraint tightened, the number of errors started to increase. Figure 7-8 depicts the actual output vector superimposed on a simulated vector at *Vdd2* = 1.6V. It shows errors occurring at a rate of approximately 9%.

The chip (Figure 7-9) has been verified to decode data with 1.8V supply at 25°C. Throughput rates above 500 Mb/s were achieved and power dissipation was 400mW. The speed and power performance of the EPR4 SOVA decoder is plotted in Figure 7-10. The power measurements were performed at the highest frequencies permitted by the supply voltage. Table 6-2 summarizes the characteristics of the decoders. The above-average power dissipation can be attributed to the increased parallel CSA activities and the continuous movement of data through rows of shift registers and FIFO buffers. The latter is especially significant as the simulated clock power was 50% of the measured power consumed by the overall decoder.

7.3. Implementation of an LDPC decoder

The 4092-bit LDPC decoder with a single Message Computation Block (MCB), as described in Section 6.2.2 is implemented on FPGA and ASIC platforms. The LDPC code is based on two-dimensional projections in finite field geometries. The rate ¾ code comprises 3070 user bits and 1022 parity bits. The exhibited cyclic property is attractive for shift-register-based high throughput implementation. Soft messages in the decoder are represented as 5-bit fixed-point types.

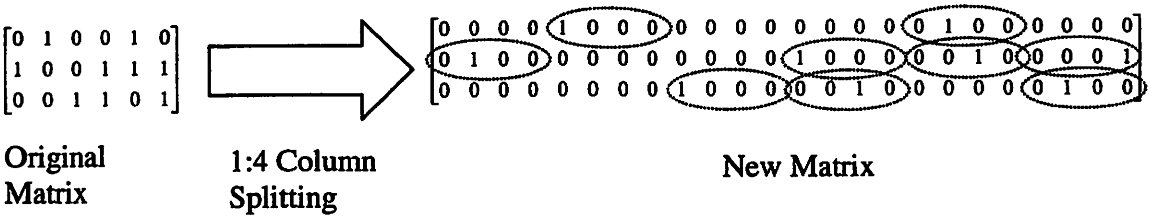
7.3.1. LDPC decoder based on finite field construction

A decoder for an LDPC code based on finite field geometries [77] has been implemented on FPGA as well as in 0.13μm CMOS technology. Due to the high (32) edge degree of the variable nodes, the staggered decoding schedule becomes a suitable option. Aside from the use of the Θ-operators presented in Section 5.4.1, this effort has also exploited a number of interesting structural properties that are specific to codes constructed using this technique.

The cyclic property of the rows in the 1023×1023 matrix (before column splitting) is exploited by replacing random access memory with fast shift registers. Using staggered decoding, the running $Q_n(t)$ messages can be moved through the shift registers. The replacement with shift registers lifts the bottleneck associated with memory throughputs. Shift register delays are typically less than 100ps in 0.13μm CMOS technology.

The 1:4 column-splitting (Figure 7-11) performed on the 1023×1023 matrix also provided advantages that are similar to the effects of memory partitioning. Since each non-zero value in the original matrix is transformed into some permutation of the $\begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix}$ vector, a check node will not be connected to more than one out of four consecutive variable nodes. Hence, the variable-to-check messages from four consecutive variable nodes, Q'_n for n=4k, 4k+1, 4k+2, 4k+3, can be distributed amongst the four parallel shift register chains. At 32 different positions along the shift register chains, the appropriate register contents are multiplexed into the MCB, which evaluates the corresponding check-to-variable messages. The 32 locations can be determined from the non-zero locations of the first row in the original 1023×1023 matrix. The outputs from the MCB are, likewise, updated to the register contents via another 32 input de-multiplexers.

Signed-magnitude representation is a natural choice for implementation of the Check-to-Bit computation block because the lookup table functions are sign-invariant, while the XOR operators are magnitude invariant. Figure 7-12 shows the overall architecture. For simplicity, the example shows the check-to-variable message computation with a pipeline latency of two cycles. In the actual implementation, the latency is 22 cycles.



$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \Longrightarrow \left[\begin{array}{cccccccccccccccccccccccc} 0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&1 \\ 0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&1&0&0 \end{array}\right]$$

Figure 7-11. Example of 1:4 column splitting.

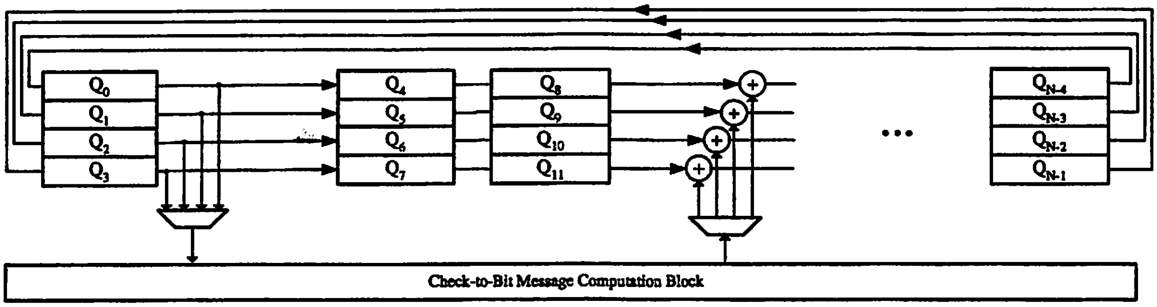


Figure 7-12. Shift register-based implementation of LDPC code generated from 2D GF(2M) with 1:4 column splitting and a message computation latency of 2.

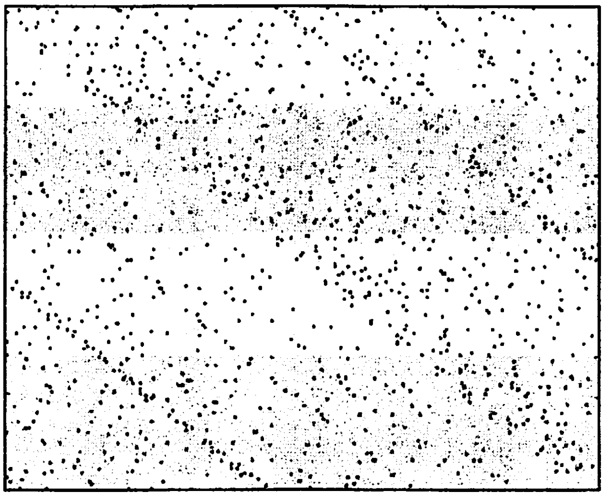


Figure 7-13. Horizontal partitioning of the 1023×4092 parity check matrix.

It is noted that the throughput of the above serial LDPC decoder can be further increased at the expense of linear increase in implementation complexity of the MSBs. For example, the 1023×4092 parity check matrix can be partitioned into four horizontal regions, shown as striped bands in Figure 7-13. The LDPC decoder is implemented with four MCBs. These blocks obtain their inputs from four sets of thirty-two 4:1 multiplexers. The locations of the first set of multiplexers along the register chain are determined from the top row of the matrix. The next three sets of multiplexer locations are cyclically displaced by approximately *M*/4. The displacement is approximate because placing more than one multiplexer at a separation in the register chain should be avoided. This prevents the necessity to add more than two inputs as the check-to-variable messages are updated into the register chain. This design can be viewed as four parallel MCBs that begin the check node processing in the four separate horizontal partitions. Over time, the data in the registers are shifted out, and likewise, the processing in the computation blocks move into neighboring partitions. The added parallelism allows the circuit to be operated at a quarter of the frequency while maintaining the same throughput.

7.3.2. Design input

The design entry for both FPGA and ASIC implementations follows a common path through the BEE design flow described in Section 7.1. Hardware directives such as word-lengths, rounding/truncation methods, and latencies are captured at the granularity of an adder, a multiplier, or memories. The Simulink description is parsed and translated into a VHDL structural netlist. From here the design flows deviate.

The design description of the LDPC decoder used a hierarchical approach. The top-level design, shown in Figure 7-14, captures the test vectors ("From Workspace" and "To Workspace" blocks) used during the design phase and contains the decoder beneath the level of abstraction described by the block *S*.

The decoder description is shown as a screen capture in Figure 7-15. The shift register chains, each of length 1023, were described as sub-blocks found at the top of the figure. Each sub-block has 32 input and 32 output ports that communicate with the MCB through multiplexers and demultiplexers. The 1:4 and 4:1 multiplexers and demultiplexers, which are visible in the middle section, obtain their control signals from a ROM block located in the left-middle section.

The values stored in the ROM were determined by examining the parity check matrix. The matrix was divided into non-overlapping vertical sections each consisting of four consecutive columns (corresponding to the 1:4 column splitting process). The intersection of each row with a vertical section produced a selection vector of four elements. The selection vector was either the zero-vector, or contained at most one non-zero entry. This "one-hot" signal is used to control the multiplexers. However, a binary representation provides more efficient use of ROM area. Finally, the index numbers of the non-zero sections within the top row of the matrix were identified. These corresponded to locations in the shift-register chains where messages were read. The locations where messages are updated to the shift-register chains were calculated by adding the 22-cycle latency of the MCB.

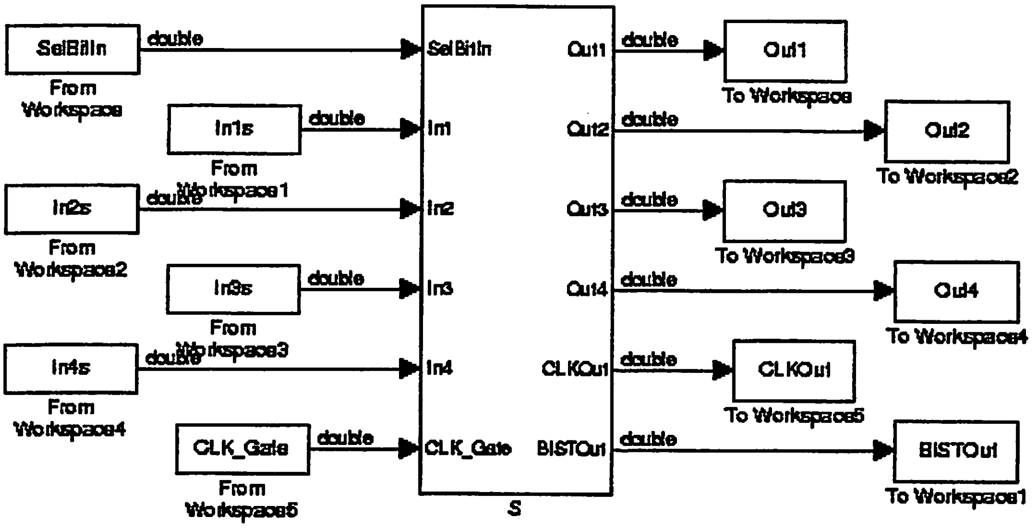


Figure 7-14. Top level design entry view in Mathworks Simulink

Even though Simulink is widely considered as a tool intended for graphical design entry, the large number of sub-blocks and the connections between the ports in this design makes this non-feasible. Instead, a script was used to place and route the sub-blocks together.

The Xilinx System Generator (*XSG*) was applied to create a behavioral VHDL description of the hierarchical macro blocks. Core blocks such as multiplexer, adder, or logical functions were mapped into Module Compiler Language (*MCL*) descriptions. The *MCL* syntax permits entry of design directives such as latencies and type of fixed-point representations. More significantly, it permitted experimentation with different micro-architectures that take advantage of, for instance, the bit profile of the computations or retiming of pipelined elements. Synopsys Module Compiler (*MC*) was used to generate the gate-level netlist for each module, as well as initial estimates of the critical delay, power, and area of implementation.

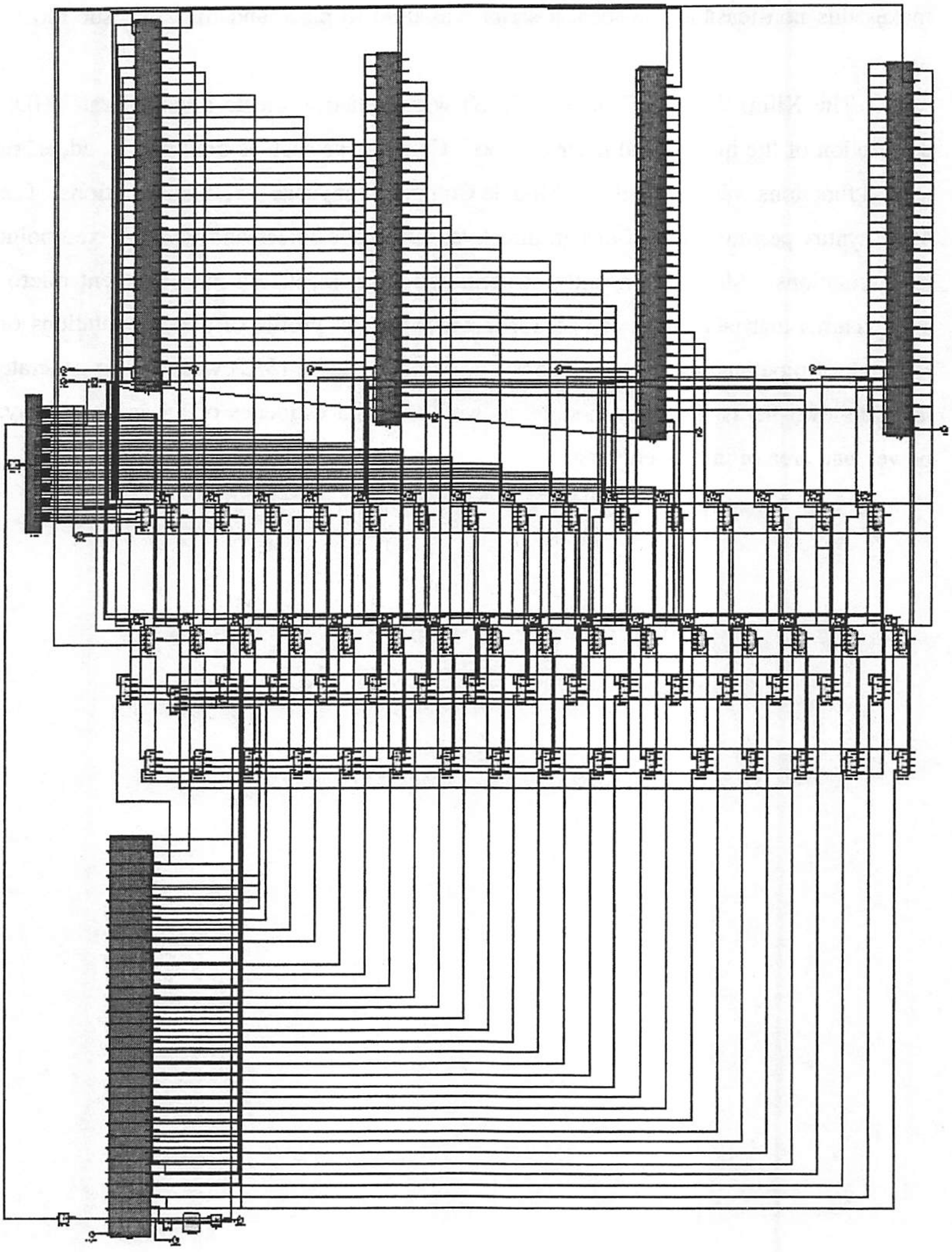


Figure 7-15. Screen capture of top-level LDPC decoder design in Simulink

7.3.3. FPGA implementation

From the VHDL netlist, compiling a bit file to upload into the FPGA was straightforward. The design occupied 16,000 slices, or 82% of a single FPGA chip. The minimum period was 38ns, imposed by the critical path through the MCB. This corresponds to a decoding throughput of 26Mb/s.

7.3.4. ASIC implementation

The physical design of the LDPC decoder was loosely based on the automated *Insecta* design flow [115]. The details of the custom modifications to the original flow are discussed. The final ASIC occupies an area 3.1mm × 2.6mm and contains 1.8 million transistors or approximately 256,000 gates. The chip is estimated to dissipate 3W at a clock frequency of 1GHz. The area of implementation was pad limited by the 23 inputs, 22 output pins, and 29 pairs of supply pads delivering dedicated power/ground to the core. Logical verification is provided through VHDL simulation at the gate level. The layout of the design is shown in Figure 7-16.

The *Insecta* flow provided a basic skeletal structure, complete with default scripts for most of the tools employed in the physical design. These scripts are generated using a top-down approach and in general, required very little design directives from the system architect. However, in order to achieve a successful chip tapeout, it was necessary to perform a number of modifications to the flow. For example, as the functionality of a design grows, a top-down approach will lead to longer run-times and larger server-memory requirement. A methodology for manual partitioning was thus developed. In the process, the structural properties of the LDPC code were studied and provided a natural way to set up boundaries between the partitioned blocks. The details of this and other modifications are documented in Appendix C. Much of this required customized solutions. Some of the key modifications include:

1. Synthesis scripts require a mixture of bottom-up and top-down compilation strategies in order to keep the memory usage of the synthesis software below a threshold.
2. Manual placement of I/O pads to found a contiguous ring around the decoder core.

3. Custom power routing to ensure minimum I-R drop across supply lines and avoid slot metal DRC errors associated with connection between supply I/O pads and supply rings.

4. Custom clock tree generation for < 50ps clock skew and < 60ps transition times.

5. Modification of LEF definitions in order to allow detailed router to handle antenna violation rules.

6.

The final design methodology is shown in Figure 7-17. The shaded boxes indicate the steps modified from the original automated flow.

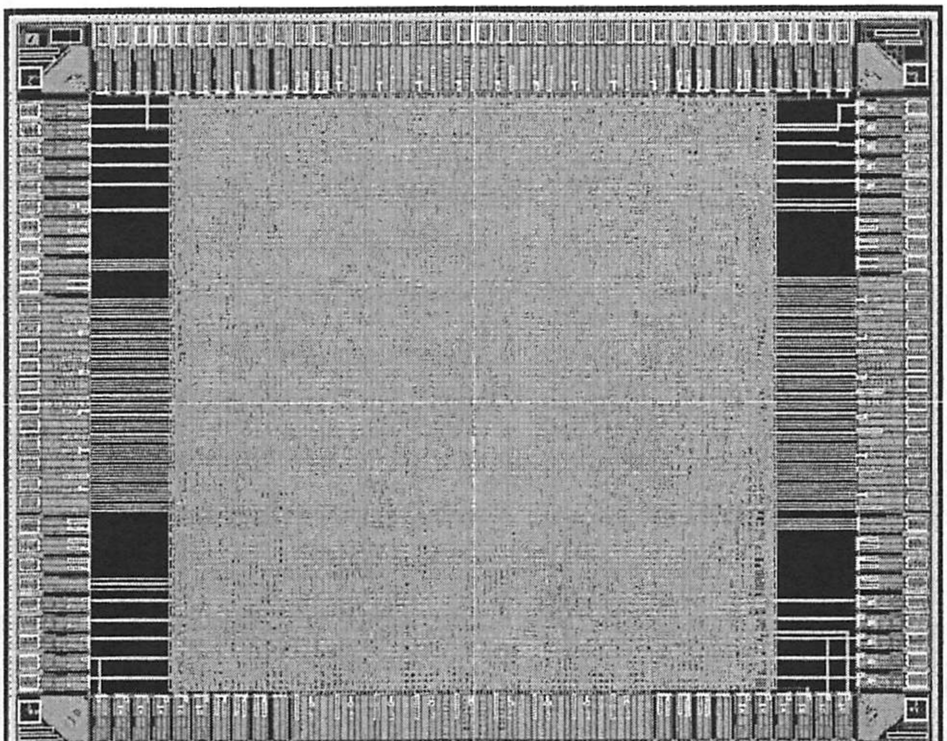


Figure 7-16. Layout of LDPC decoder in 0.13μm CMOS, occupying 3.2mm × 2.7mm.

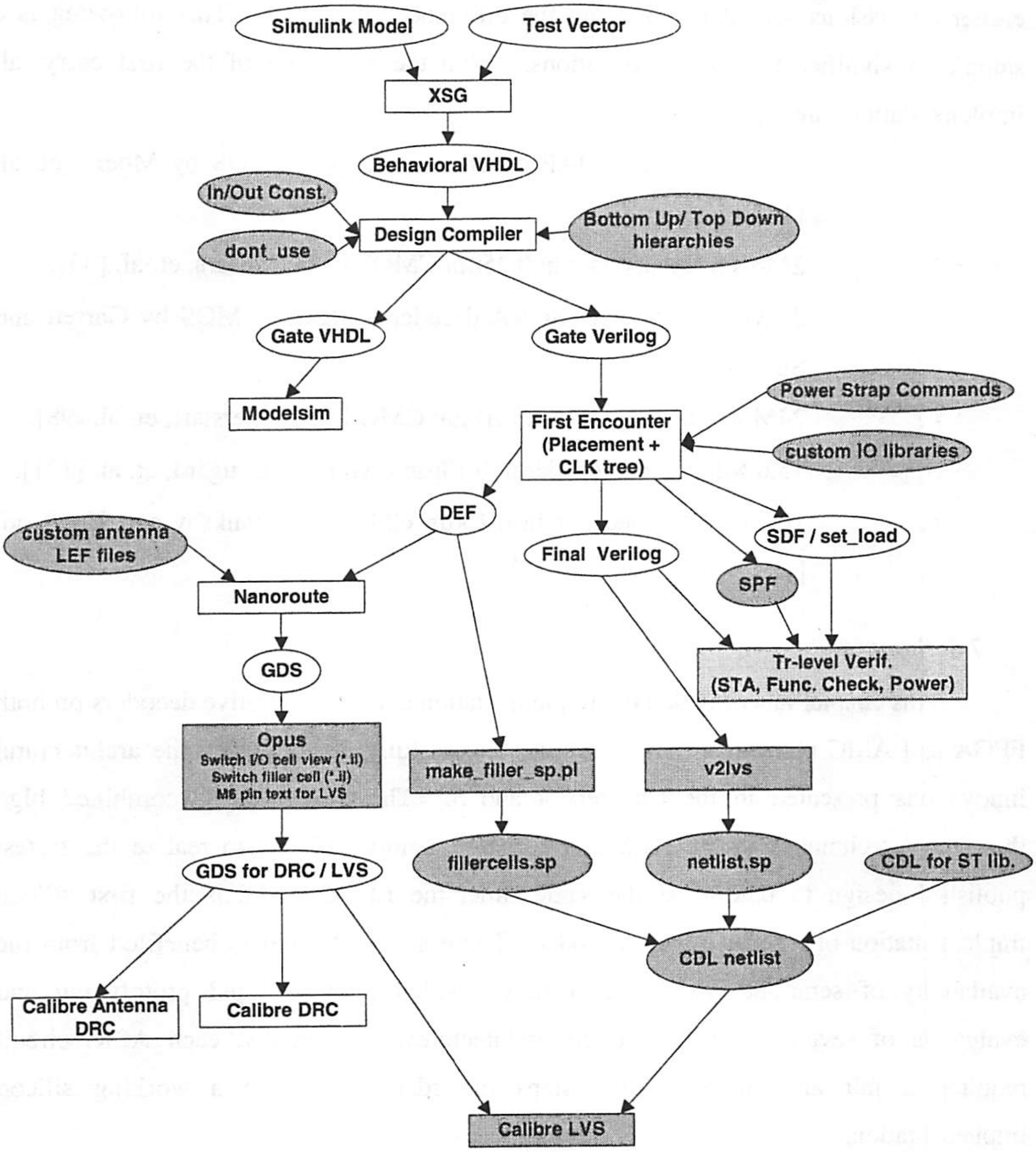


Figure 7-17. Design methodology. Shaded symbols indicate steps modified from the standard Insecta flow.

7.4. Other state of the art implementations

At the time of this publication, a number of implementations of iterative decoder hardware have surfaced. These works have been mentioned in previous chapters as the earlier discussions sought to point out the differences in details. This following is a sample of significant ASIC contributions. With the exception of the first entry, all implementations are digital.

1. 320Mb/s analog MAP decoder in 0.25μm CMOS by Moerz, et. al. [120]
2. 2Mb/s turbo decoder in 0.25μm CMOS by G. Masera, et. al. [33].
3. 2.5Mb/s low power SOVA decoder in 0.3μm CMOS by Garrett and Stan [16].
4. 24Mb/s turbo decoder in 0.18μm CMOS by Bickerstaff, et. al. [98].
5. 75.6 Mb/s turbo decoder in 0.18μm CMOS by Bougard, et. al. [121].
6. 1Gb/s LDPC decoder in 0.15μm CMOS by Blanksby and Howland, [1].

7.5. Summary

This chapter has provided the implementation details of iterative decoders on both FPGA and ASIC platforms. These designs successfully demonstrated the architectural innovations presented in the Chapters 4 and 5. The SOVA ASIC combined high throughput techniques in the ACS and survivor memory blocks to realize the fastest published design to date. At the same time, the LDPC ASIC is the first silicon implementation of a serial LDPC decoder. These accomplishments benefited from the availability of semi-automated design flows, which allowed rapid prototyping and evaluation of several competing micro-architectures. Eventually, each ASIC circuit required a fair amount of custom steps in order to achieve a working silicon implementation.

8. CONCLUSIONS

This research has addressed the algorithms and implementations of iterative decoders for forward error control in high throughput communication applications. The architectures of iterative codes based on various concatenated schemes of convolutional codes, and low-density parity check (LDPC) codes were presented and analyzed.

The message passing algorithms implemented by BCJR, SOVA and LDPC decoders were discussed. The difficulties associated with implementation of the computational and message-passing requirements were highlighted in Chapter 3.

Architectures that are suitable for high throughput implementations of the MAP and SOVA decoders were presented in Chapter 4. The memory requirements were realized using fast shift registers. The throughput bottleneck is found at the AACS or ACS recursions. These structures were categorically discussed and evaluated.

Architectures for implementation of LDPC decoders were introduced in Chapter 5. Parallel structures were compared against serial structures. The properties of LDPC codes that favor shared memory architectures were also discussed. A number of complexity-reducing methods were introduced. The arithmetic computational complexity of the LDPC decoder is low, when compared against existing Turbo decoders. However, both serial and parallel implementations of the LDPC decoder have to address the issue of large memory requirement and interconnect congestion respectively. This arises from the random and often sparse structure of general LDPC codes. These two factors are mutually exclusive, and choices between serial or parallel implementation is tied with tradeoff between memory or interconnect concerns.

Finally, the implementation details of iterative decoders on both FPGA and ASIC platforms were discussed in Chapter 7. These designs successfully demonstrated the architectural innovations in this work. These accomplishments benefited from the availability of semi-automated design flows, which allowed rapid prototyping and evaluation of several competing micro-architectures, but also required a significant of manual tooling in order to achieve a successful and working silicon implementation.

In particular, the key contributions of this work are summarized below:

1. In evaluating the throughput bottleneck of turbo decoders, a number of micro-architectures for implementation of the ACS or ACSA were categorically discussed, synthesized and evaluated.
2. Efficient scheduling of the MAP decoder permits the memory to be implemented with fast shift registers, with minimal control logic.
3. Discovery that column splitting in LDPC codes based on finite field geometries permits a natural partitioning of the decoder design.
4. Approximation of the Φ function based on the difference between pairs of check-node input messages in the LDPC decoder.
5. Proposed staggered decoding schedule for the LDPC decoder successfully achieved significant reduction of memory requirement and improved performance (with low number of iterations) at the same time.
6. Successful implementation of a SOVA ASIC, which combined high throughput techniques in the ACS and survivor memory blocks to realize the fastest published design to date.
7. Successful implementation of a LDPC ASIC. This is the first silicon implementation of a serial LDPC decoder.

8.1. Future

As communications systems continue to demand higher throughputs in environments with decreasing SNR, iterative decoding can provide the necessary coding gain to ensure sustained progress. Already, this class of codes has been specified as part of standards in wireless MAN (IEEE 802.16), wireless cellular communications (CDMA2000 and UMTS 3GPP), wireline communications (HiperLAN2), and digital video broadcast from satellite (DVBS-2). Without doubt, the deployment of these codes will become more widespread as the requirements of end-users grow and the technology advances to make the implementation of the decoders feasible. Although the examples presented in this work have shown that iterative decoders are at least four to five times the complexity of current non-iterative methods, each silicon process technology generation will allow the integration of more complex signal processing schemes. The same implementations in 90nm CMOS technology would lower the area and power requirements by a factor of two, while improving the throughput by 40% [119].

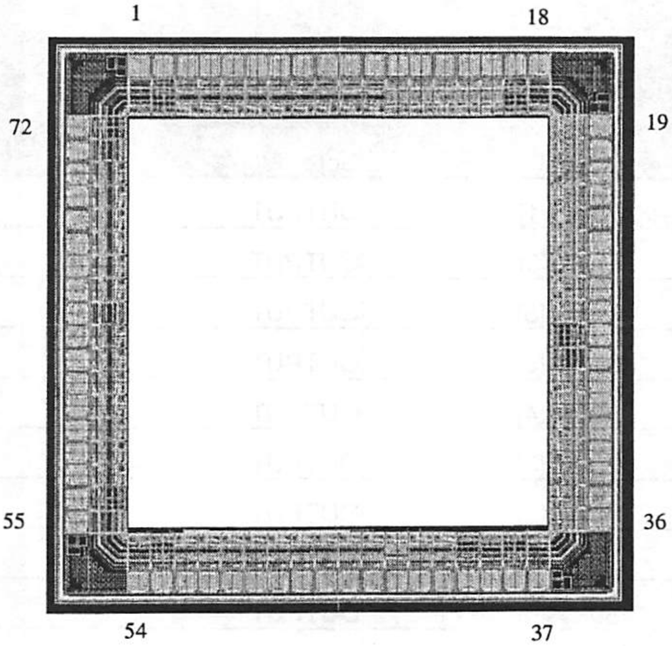
As researchers in coding techniques continue the search for better bit-error performance, there will be increased awareness of the implications on the VLSI aspects of the decoders. For example, parallel decoder architectures for the general class of LDPC codes will have to address the potential problem of routing congestion. By limiting the search space to structured LDPC codes, a large message-passing graph can be suitably partitioned into several smaller graphs, thereby increasing the possibility of fully parallel implementations that are also scalable to larger code sizes. To a certain extent, architectural manipulations and transformations alone can affect the throughput bottleneck, required number of processing elements, logic density, etc. However, the successful implementation of a decoder is also dependent on simple considerations such as the number of bits required to represent the messages, or resolution of the required lookup tables. These details directly affect the performance of the code, but their effects are not easily determined. Currently, the most common approach is to evaluate the bit-error performance via empirical methods involving long hours of simulation in Matlab or C programs. It will be interesting to see an emergence of studies providing theoretical bounds on the performance improvement or degradation as a function of these implementation parameters.

Speaking of simulations, it should be pointed out that one of the primary difficulties with the evaluation of BER performance was the long simulation times that are required by iterative decoders. Since iterative codes routinely decode down to BER of 10^{-8}, the performance simulations are often conducted with up to 10^{11} samples or more in order to obtain results that are statistically significant. The situation is exacerbated by the need to evaluate fixed-point effects, which can lower the simulation throughputs on general-microprocessor platforms to hundreds of kb/s. The FPGA design flow described in Chapter 7 offers a viable option for the performance analysis of iterative codes. With minimal user intervention, the designs implemented on the FPGAs operate at throughputs between 10Mb/s and 25Mb/s. However, this requires a major paradigm shift from general microprocessor-based evaluation techniques to FPGA-based simulations.

In short, the need for improved coding gains in all forms of communications applications will drive the adoption of iterative decoding. The exploration for practical solutions will combine studies between code design and VLSI considerations. Existing evaluation methods based on simulations are too slow. As the research gathers pace, there is a requirement for formal assessment of the relationship between implementation parameters and the bit-error performance. The alternative is to carry out the simulations on reconfigurable hardware such as FPGA, which can provide a speed improvement of at least an order of magnitude.

9. APPENDIX A: PIN LIST FOR SOVA DECODER

9.1. Pad Frame



9.2. Pin List

See above for pin number description

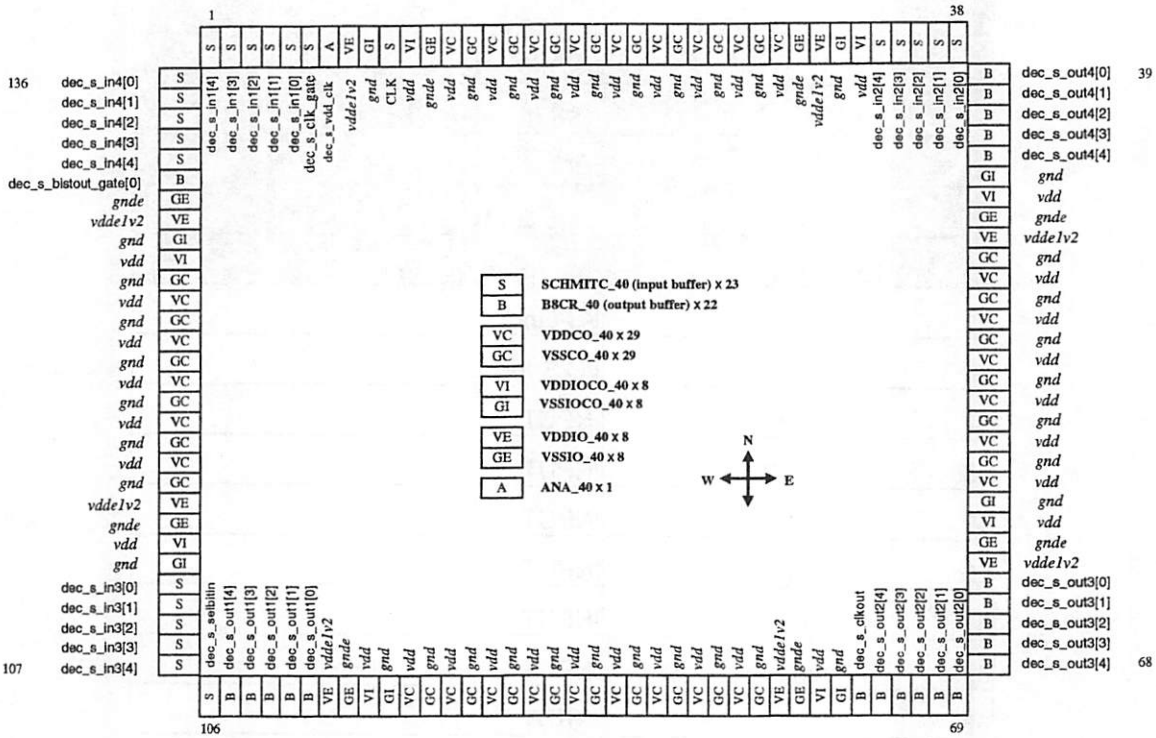
Pin No.	SIGNAL	COMMENTS
1	vdd	Core Supply
2	gnd	Core Gnd
3	SysIn[6]	INPUT
4	ParIn[0]	INPUT
5	ParIn[1]	INPUT
6	ParIn[2]	INPUT
7	ParIn[3]	INPUT
8	ParIn[4]	INPUT
9	ParIn[5]	INPUT
10	ParIn[6]	INPUT
11	SI	SCAN CHAIN INPUT

12	SoPar[6]	OUTPUT
13	SoPar[5]	OUTPUT
14	SoPar[4]	OUTPUT
15	SoPar[3]	OUTPUT
16	SoPar[2]	OUTPUT
17	gnd	Core Gnd
18	vdd	Core Supply
19	SoPar[1]	OUTPUT
20	SoPar[0]	OUTPUT
21	SoSys[6]	OUTPUT
22	SoSys[5]	OUTPUT
23	SoSys[4]	OUTPUT
24	SoSys[3]	OUTPUT
25	SoSys[2]	OUTPUT
26	SoSys[1]	OUTPUT
27	SoSys[0]	OUTPUT
28	gnd	I/O Gnd
29	vdd	I/O Supply
30	SoftOut[6]	OUTPUT
31	SoftOut[5]	OUTPUT
32	SoftOut[4]	OUTPUT
33	SoftOut[3]	OUTPUT
34	SoftOut[2]	OUTPUT
35	SoftOut[1]	OUTPUT
36	SoftOut[0]	OUTPUT
37	vdd	Core Supply
38	gnd	Core Gnd
39	vdd	I/O Supply
40	gnd	I/O Gnd
41	Reset[0]	INPUT CONTROL
42	SEN	INPUT SCAN ENABLE
43	SO	OUTPUT SCAN CHAIN DATA
44	test_en	INPUT CONTROL
45	I1[6]	INPUT

46	I1[5]	INPUT
47	I1[4]	INPUT
48	I1[3]	INPUT
49	I1[2]	INPUT
50	I1[1]	INPUT
51	I1[0]	INPUT
52	SoftIn[6]	INPUT
53	gnd	Core Gnd
54	vdd	Core Supply
55	vdd	I/O Gnd
56	gnd	I/O Supply
57	SoftIn[5]	INPUT
58	SoftIn[4]	INPUT
59	SoftIn[3]	INPUT
60	SoftIn[2]	INPUT
61	SoftIn[1]	INPUT
62	SoftIn[0]	INPUT
63	CLk	INPUT
64	vdd2	VARIABLE CLOCK SUPPLY
65	SysIn[0]	INPUT
66	SysIn[1]	INPUT
67	SysIn[2]	INPUT
68	SysIn[3]	INPUT
69	SysIn[4]	INPUT
70	SysIn[5]	INPUT
71	gnd	I/O Gnd
72	vdd	I/O Supply

10.APPENDIX B: PIN LIST FOR LDPC DECODER

10.1. Pad Frame



10.2. Pin List

See above for pin number description

Pin No.	SIGNAL	COMMENTS
1	dec_s_in1[4]	INPUT
2	dec_s_in1[3]	INPUT
3	dec_s_in1[2]	INPUT
4	dec_s_in1[1]	INPUT
5	dec_s_in1[0]	INPUT
6	dec_s_clk_gate	INPUT
7	dec_s_vdd_clk	VARIABLE CLOCK SUPPLY
8	vdde1v2	I/O Supply2
9	gnd	I/O Gnd
10	CLK	INPUT

11	vdd	I/O Power
12	gnde	I/O Gnd2
13	vdd	Core Supply
14	gnd	Core Gnd
15	vdd	Core Supply
16	gnd	Core Gnd
17	vdd	Core Supply
18	gnd	Core Gnd
19	vdd	Core Supply
20	gnd	Core Gnd
21	vdd	Core Supply
22	gnd	Core Gnd
23	vdd	Core Supply
24	gnd	Core Gnd
25	vdd	Core Supply
26	gnd	Core Gnd
27	vdd	Core Supply
28	gnd	Core Gnd
29	vdd	Core Supply
30	gnde	I/O Gnd2
31	vdde1v2	I/O Supply2
32	gnd	I/O Gnd
33	vdd	I/O Power
34	dec_s_in2[4]	INPUT
35	dec_s_in2[3]	INPUT
36	dec_s_in2[2]	INPUT
37	dec_s_in2[1]	INPUT
38	dec_s_in2[0]	INPUT
39	dec_s_out4[0]	OUTPUT
40	dec_s_out4[1]	OUTPUT
41	dec_s_out4[2]	OUTPUT
42	dec_s_out4[3]	OUTPUT
43	dec_s_out4[4]	OUTPUT
44	gnd	I/O Gnd

45	vdd	I/O Power
46	gnde	I/O Gnd2
47	vdde1v2	I/O Supply2
48	gnd	Core Gnd
49	vdd	Core Supply
50	gnd	Core Gnd
51	vdd	Core Supply
52	gnd	Core Gnd
53	vdd	Core Supply
54	gnd	Core Gnd
55	vdd	Core Supply
56	gnd	Core Gnd
57	vdd	Core Supply
58	gnd	Core Gnd
59	vdd	Core Supply
60	gnd	I/O Gnd
61	vdd	I/O Power
62	gnde	I/O Gnd2
63	vdde1v2	I/O Supply2
64	dec_s_out3[0]	OUTPUT
65	dec_s_out3[1]	OUTPUT
66	dec_s_out3[2]	OUTPUT
67	dec_s_out3[3]	OUTPUT
68	dec_s_out3[4]	OUTPUT
69	dec_s_out2[0]	OUTPUT
70	dec_s_out2[1]	OUTPUT
71	dec_s_out2[2]	OUTPUT
72	dec_s_out2[3]	OUTPUT
73	dec_s_out2[4]	OUTPUT
74	dec_s_clkout	OUTPUT
75	gnd	I/O Gnd
76	vdd	I/O Power
77	gnde	I/O Gnd2
78	vdde1v2	I/O Supply2

79	gnd	Core Gnd
80	vdd	Core Supply
81	gnd	Core Gnd
82	vdd	Core Supply
83	gnd	Core Gnd
84	vdd	Core Supply
85	gnd	Core Gnd
86	vdd	Core Supply
87	gnd	Core Gnd
88	vdd	Core Supply
89	gnd	Core Gnd
90	vdd	Core Supply
91	gnd	Core Gnd
92	vdd	Core Supply
93	gnd	Core Gnd
94	vdd	Core Supply
95	gnd	Core Gnd
96	vdd	Core Supply
97	gnd	I/O Gnd
98	vdd	I/O Power
99	gnde	I/O Gnd2
100	vdde1v2	I/O Supply2
101	dec_s_out1[0]	OUTPUT
102	dec_s_out1[1]	OUTPUT
103	dec_s_out1[2]	OUTPUT
104	dec_s_out1[3]	OUTPUT
105	dec_s_out1[4]	OUTPUT
106	dec_s_selbitin	INPUT
107	dec_s_in3[4]	INPUT
108	dec_s_in3[3]	INPUT
109	dec_s_in3[2]	INPUT
110	dec_s_in3[1]	INPUT
111	dec_s_in3[0]	INPUT
112	gnd	I/O Gnd

113	vdd	I/O Power
114	gnde	I/O Gnd2
115	vdde1v2	I/O Supply2
116	gnd	Core Gnd
117	vdd	Core Supply
118	gnd	Core Gnd
119	vdd	Core Supply
120	gnd	Core Gnd
121	vdd	Core Supply
122	gnd	Core Gnd
123	vdd	Core Supply
124	gnd	Core Gnd
125	vdd	Core Supply
126	gnd	Core Gnd
127	vdd	I/O Power
128	gnd	I/O Gnd
129	vdde1v2	I/O Supply2
130	gnde	I/O Gnd2
131	dec_s_bistout_gate[0]	OUTPUT
132	dec_s_in4[4]	INPUT
133	dec_s_in4[3]	INPUT
134	dec_s_in4[2]	INPUT
135	dec_s_in4[1]	INPUT
136	dec_s_in4[0]	INPUT

11.APPENDIX C: MODIFICATIONS TO ASIC DESIGN FLOW

This appendix documents the modifications performed on the different stages in the physical design process of the LDPC decoder. The *Insecta* flow provided a default design methodology, in which these modifications were applied.

11.1. Design Synthesis

Using Synopsys Design Compiler (*DC*), the process of synthesis reads the behavioral VHDL and the *MC*-created gate-level modules, and produces a netlist of gates obtained from a high-speed standard-cell library in 0.13μm 6-metal layer CMOS technology. The design was inserted with Schmitt-triggered input pads to capture the effect of input drive strengths. Likewise, buffered output pads are inserted to emulate the effects of output loading. The global clock is constrained to a 1ns period. Clock buffer cells and some datapath library cells that have pass gate inputs are excluded from the usable cell list by applying "set_dont_use" commands to these cells.

In a hierarchical design, multiple references of a sub-hierarchy may occur. The default *Insecta* flow resolves these multiple instances by creating a uniquely named copy of the sub-design for each instance. The synthesis process then performs a flat compilation, which optimizes each design copy based on the unique environment of its cell instance. However, as the size of the design exceeds a hundred thousand gates, this process can easily take up to a few days on a gigahertz-Xeon processor. In addition, the Linux-based binaries operating on a 32-bit architecture are unable to access memory addresses above a 4GB limit. These factors made the flat synthesis approach unsuitable for a design of the size of the LDPC decoder. In response, the synthesis is performed with a mix of top-down and bottom-up methods. Taking advantage of the knowledge of system partitioning that is already evident in the design input process (e.g. Figure 7-15), the key blocks such as the shift-register chains (instantiated four times), are compiled once. The *set_dont_touch* command is then used to preserve the sub-design during the remaining optimization. This permitted a successful synthesis in about 2 hours of processor time.

DC outputs both VHDL and Verilog gate-level netlists. The VHDL output is used for logical verification in *Mentor Modelsim.* This uses the test vectors created by *XSG* when it parsed the initial design in the Simulink description. The Verilog output is the primary design interface with Cadence First Encounter (*FE*), which performs power/ground routing and standard cell placement.

11.2. Placement

The scripts generated by the *Insecta* flow provides global power and ground routing, location-constrained cell placement, and clock tree generation. However, there were issues with the automated flow that caused unacceptable results. The default input/output pad placement distributed the I/O pads evenly around the chip area, but failed to maintain gaps between I/O pads that were divisible by the width of the smallest I/O filler pad. Consequently, the pads failed to form a continuous ring around the chip. Moreover, the global power and ground routing relied on long rails in metal1 that span the width of the core. It also used long and wide metal2 for the power and ground rings, which led to slot metal DRC violations.

11.3. I/O pad placement

By default, *FE* distributes the I/O pads evenly around the perimeter of the chip area. This leads to gaps with widths that are not divisible by that of the smallest I/O filler pad. The I/O filler pads available in the I/O library have widths that are multiples of 0.41μm. The naming convention of the I/O filler pads, IOFILLERXX_40, reflects this property; XX represents the multiplicative factor. It is necessary to provide a file specifying customized I/O placements that also controls the widths of any remaining gaps. This file should include definitions of corner I/O cells preceded by the "Orient: R0" statement. With the exception of the dedicated core power (VDDCO_40) and ground (VSSCO_40) supply pads, all pad cells have widths of 39.77 μm (97×0.41μm). Since this chip is intended to be fabricated with an I/O pitch of 65.19μm (159×0.41μm), an IOFILLER62_40 (25.42μm) cell is placed next to every I/O pad. An example excerpt from the I/O placement file is also provided below.

```
Orient: R0
Pad:    NE_CORNER     NE    RTCORNER_40ISV
Skip:   319.8
Pad:    IoFill62_N_4  N     IOFILLER62_40
```

```
Skip:   0
Pad:    dec_s_in1_gatex4x_PAD  N
Skip:   0
Pad:    IoFill62_N_6    N       IOFILLER62_40
Skip:   0
Pad:    dec_s_in1_gatex3x_PAD  N
Skip:   0
Pad:    vddx_padx3x     N
Skip:   0
Pad:    gndx_padx5x     N
Skip:   0
Pad:    vddx_padx2x     N
Skip:   0
```

In addition, the customized I/O placement plan made it possible to determine any gaps along the perimeter that could accommodate additional power and ground pads. These additional pads have to be annotated in the *FE* input Verilog file in order to generate an accurate netlist that is used for LVS comparison. The following lines, written in Verilog syntax, represent an example of a pair of power and ground pads.

```
VDDCO_40 vddx_padx102x ( .VDDCORE(\vdd! ) );
VSSCO_40 gndx_padx103x ( .VSSCORE(\gnd! ) );
```

Each pair of power and ground pads is current-limited to 50mA. The final design contains 29 pairs of pads, and is capable of delivering 1.74W to the core at nominal voltage of 1.2V.

The I/O library documentation also specified that additional I/O supply pads are necessary to provide the power and ground supplies to the Schmitt-triggered input pads, buffered output pads, and ESD protection circuits. Eight sets of pads are added to the design; each set comprised the four I/O pads "VDDIOCO_40", "VSSIOCO_40", "VDDIO_40", and "VSSIO_40". Each pair of I/O supply pads ("VDDIO_40" and "VSSIO_40") can drive a maximum of 12 output buffers ("B2CR_40"); there are only 22 output pins in the design. As is true previously, the input Verilog file is annotated with these additional cells. An example is provided below.

```
VDDIOCO_40 vddioco_pad4 ();
VSSIOCO_40 vssioco_pad4 ();
VDDIO_40 vddio_pad4 ();
VSSIO_40 vssio_pad4 ();
```

The final configuration includes two sets of I/O supply pads on each side of the chip. It satisfies all maximum distance rules for ESD protection. (1.6mm separation between "VDDIO_40" or "VSSIO_40" pads, 4.5mm between "VDDIOCO_40", and 2.5mm between "VSSIOCO_40" pads.

The *Insecta* flow reads the Verilog design input and a custom file describing the I/O placement, before building an initial floorplan. Using information on both standard cells and I/O pads in the design, *FE* automatically determines a guides/regions/fences drawing density. An initial floorplan based on this density value is created. However, the peculiarities of the placement constraints on the I/O pads causes the drawing density to be significantly lower than the core density, defined as the ratio of area of standard cells to the core area. As a result, core densities in the neighborhood of 30% are not uncommon. Changing the sequence of operations in FE increases the drawing density. The input of the I/O placement plan is delayed until an initial floorplan is created. Such a floorplan is based strictly on the standard cell information. A core density of 56% prior to clock tree insertion is observed. The final arrangement of the I/O pad is shown in Figure 11-1.

11.4. Power routing

The custom power routing of the chip is the most significant deviation from the automated design flow. It is based on the design application notes provided by STMicroelectronics. The power structure, shown in Figure 11-2, is made up of a grid of 7.14μm metal6 horizontal stripes, 6.28μm metal5 vertical stripes, and metal1 horizontal rails. The metal1 rails correspond to standard cell rails. The stripes alternate between power and ground supplies with a pitch of 19.68μm. A 6.28μm × 7.14μm array of via5 connects between the horizontal and vertical stripes. The vertical stripes are connected to the horizontal rails using a 3×5 array of stacked vias that connect between metal1 and metal5.

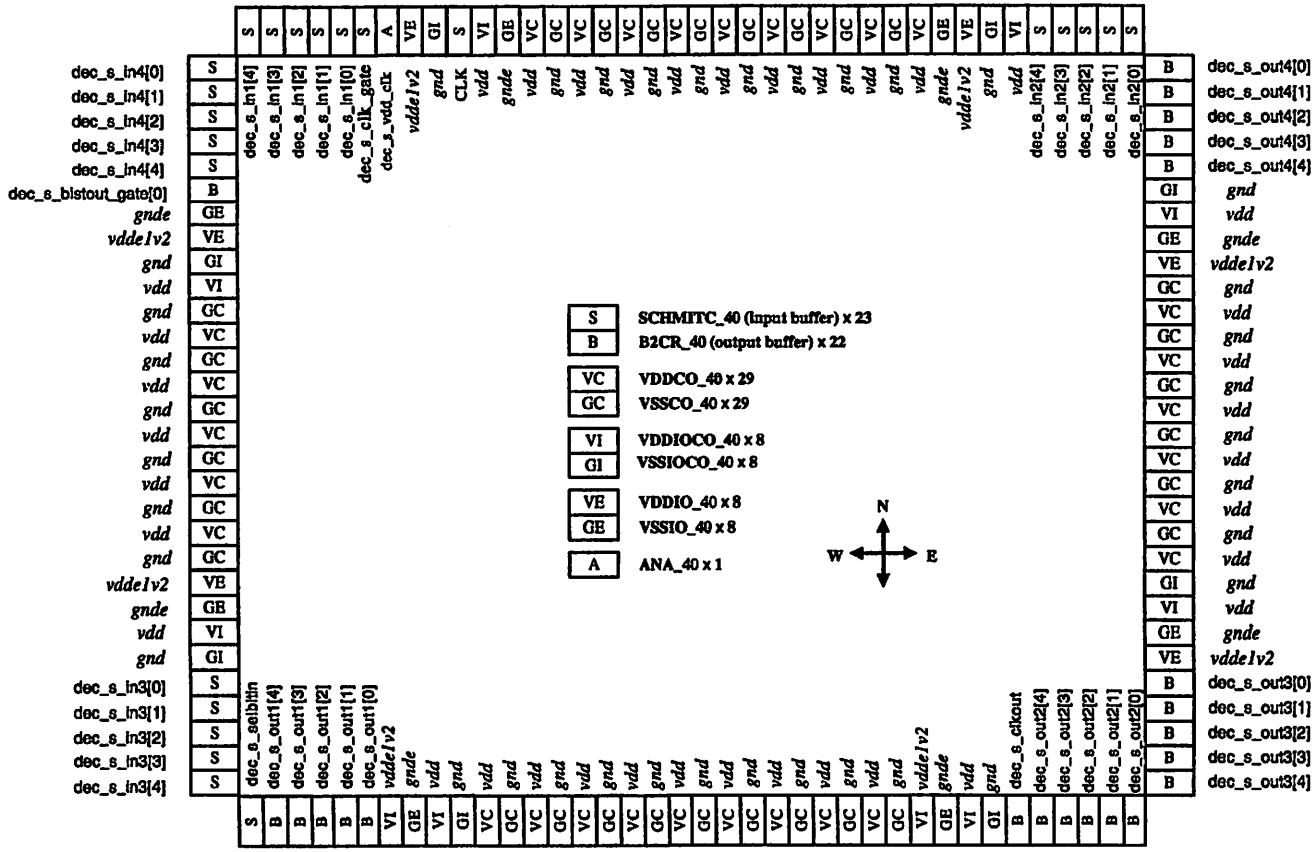


Figure 11-1. I/O Pad arrangement

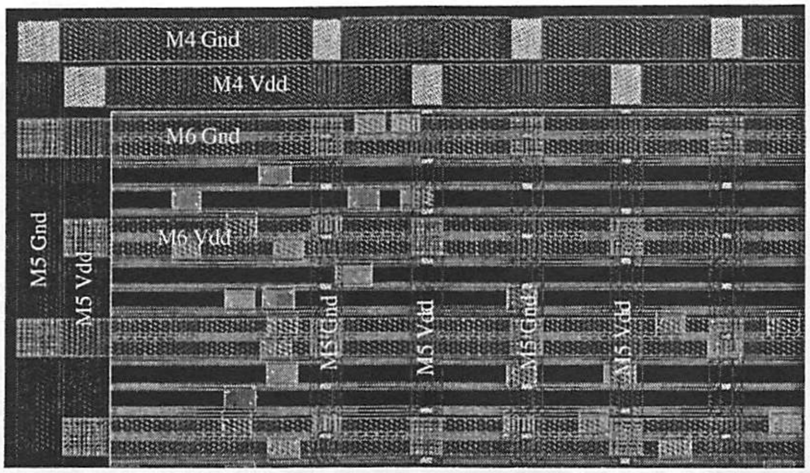


Figure 11-2. Power and ground supply grid.

A dedicated power ring is placed around the core, using metal5 vertical sides, and metal4 horizontal sides. *FE* connects the supply pads to the supply ring by routing a continuous vertical stripe of metal2 from each pin of an I/O pad to the ring location, and placing arrays of stacked vias that connect metal2 up to metal6. The widths of the metal2 interconnect and the stacks of via arrays matches the width of the pin on the I/O pad, which is 59.26μm wide.

Slot metal DRC rules mandated the insertion of slots in metal connectors that are wider than 12μm and longer than 30μm. Slot metal rules prevent the occurrence of long wide metal interconnects that increase the risks of oxide erosion or copper dishing during the fabrication process of chemical-mechanical polishing (CMP). In the power routing scheme described above, a metal interconnect from a power pad pin, which measures 59.26μm-wide, to an inner core ring is likely to lead to slot-metal violations. The solution is to modify the *cdump* library definitions of the power and ground supplies. A single 59.26μm pin is split into three separate pins, each measuring 11.9μm in width. This causes *FE* to route three separate stripes instead of a single, wide interconnect between each power or ground supply pad and the corresponding supply ring. This is shown in Figure 11-3.

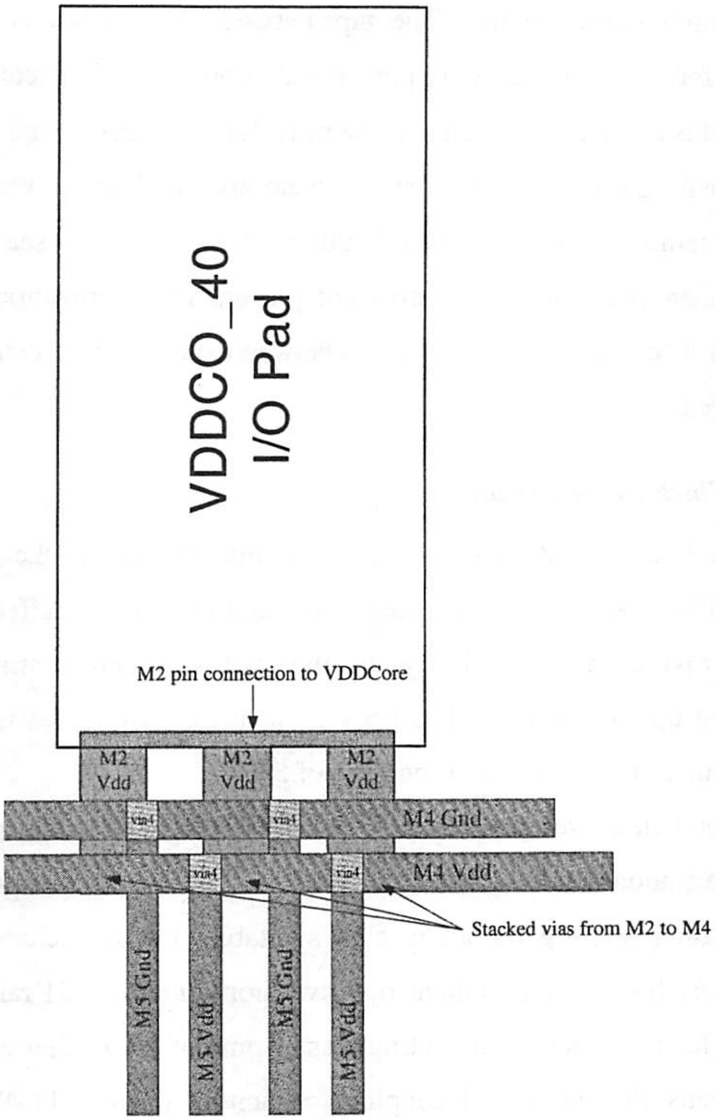


Figure 11-3. Connection between supply I/O pads and metal4 supply rings

Inside the core area, the large number of highly stacked vias between metal1 horizontal rails and metal5 vertical stripes pose potential routing congestion. The standard cells should therefore avoid being placed near the vertical stripe locations. This can be simply implemented if the cell placement is performed after power-grid routing, since the placement optimization attempts to minimize the amount of congestion. On the other hand, the power-grid routing requires knowledge of the size of the core, and locations of standard cell rails. This implies performing cell placement prior to power routing. Taking into account of these opposing requirements, the following sequence of operations are performed in *FE*. An initial raw placement of the standard cell is performed with limited optimization. The gaps between the standard cells are filled with filler cells, thus forming a continuous power and ground rail in metal1. The metal5-metal6 power grid is created, and routed to the rails through stacked vias at every junction between corresponding rail and vertical-stripe locations. With the power grid connected, all filler cells are removed, and all standard cells are "unplaced". A second placement of standard cell is then performed with stringent placement optimization constraints. A clock tree is then inserted. Finally, the gaps between the standard cells are filled once again with filler cells.

11.5. Clock tree generation

The clock tree generation is specified for minimal clock skews and transition delays at the 34,000 sink nodes. The phase delay and maximum buffer transition times are relaxed to provide more flexibility for the tool in meeting stringent skew and transition delays at the sink nodes. The final skew at the sink nodes is estimated to be less than 42ps, with a maximum transition time of 59ps.

FE outputs a final *Design Exchange Format* (DEF) file that captures placement information of the standard cells, I/O pads, clock tree, filler cells, and power and ground supply grid. A final Verilog output is also available, but it excludes the filler cell information since each filler cell contains only two horizontal metal1 rails for power and ground supplies. In subsequent post-routing steps, some of these filler cells are swapped with new filler cells that include decoupling capacitors (Figure 11-4) formed by the polysilicon gates of transistors with non-minimal feature size.

11.6. Detailed routing

Nanoroute (*NR*) reads the DEF output from *FE* and completes the detailed routing. The standard cells libraries and routing rules, which include metal and via layer descriptions, are defined in *Library Exchange* Format (LEF). The LEF file used by *NR* is created manually by referring to a design manual provided by the foundry. Every attempt has been made to ensure that the definitions in the LEF file correspond with the manufacturing rules, although updates remain a continual process. Designs with fewer than 200,000 transistors, or 33,000 gates and less than 50% core densities can generally complete the detailed route without incurring any DRC violations.

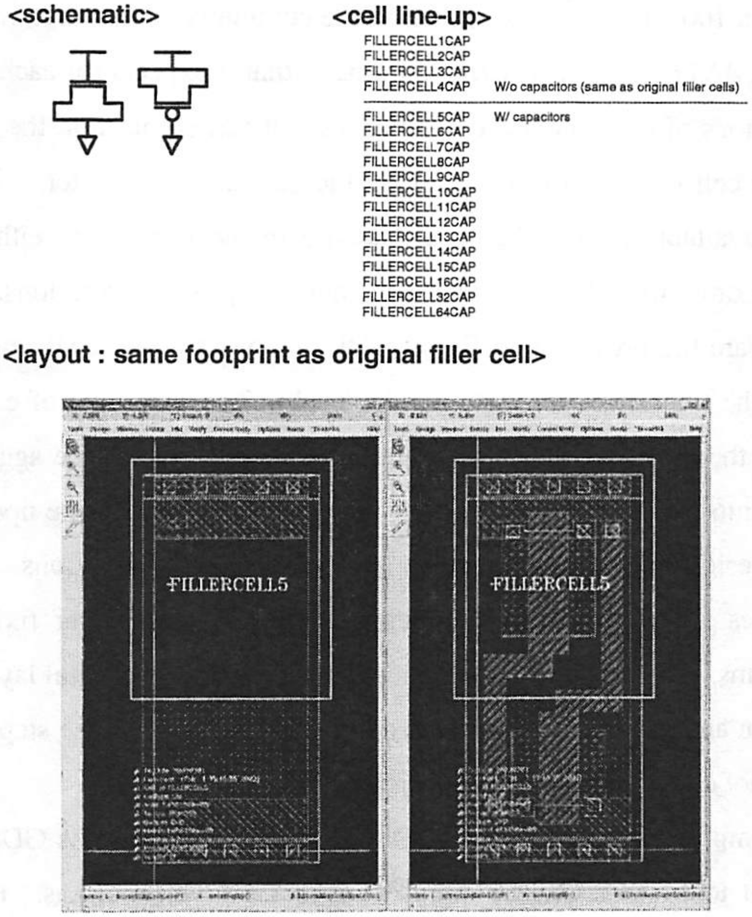


Figure 11-4. Filler cells with decoupling capacitors.

One of the pitfalls of building a larger design is that lengths of interconnects tend to increase with the dimensions of the overall design. Long metal routing can cause an electrical charge accumulation during the fabrication etching process. The build-up charge can potentially damage a gate oxide that is connected directly to the metal route. Inserting reverse-biased diodes between long interconnects and a ground node can avoid this process antenna effect. If the charge build-up during etching process is greater than a few volts, the diodes will become reverse-biased junctions and provide a path for the charges to the substrate. During normal operation at 1.8V supply, the diodes do not cause any effect. Alternatively, the routing is divided into shorter sections, which are connected through higher metal layers.

The DRC rules require that the cumulative area of any single metal layer of interconnect over field oxide divided by the area of the transistor gate (thin oxide area) must be less than 100. This ratio is known as the cumulative antenna ratio. *NR* calculates this based on an *ANTENNAGATEAREA* parameter that it expects for each input pin in the LEF file definitions of the standard cell libraries. It turned out that the LEF definitions for the standard cell libraries did not include the gate area parameter. Nonetheless, this information is available in the Cadence database in the formats of either CDL or dfII descriptions. In order to make *NR* address the antenna process violations, a perl script is written to compare the provided LEF and CDL databases. The CDL database included dimensions of the transistors in each standard cell. The gate area of each transistor is calculated from the product of the width and length of the gate. The aggregate gate area is then written into the LEF file as the missing parameter. Using the updated LEF files, all subsequent designs are routed without any process antenna violations.

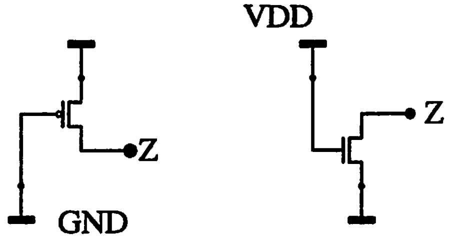
NR places a higher priority on correcting DRC errors over fixing the antenna violation problems. The tool will automatically shuffle between metal layers, or replace a filler cell with an antenna protection diode cell (*ANTPROT8*). These steps are performed only if they do not cause additional DRC errors.

Upon completion, *NR* outputs a GDS file. Conceptually, this GDS output should be directly input to Mentor Calibre for DRC and LVS verifications. However, a few problems persist with this output, and require the conversion of the design into Opus dfII format, as described in the next section.

11.7. Cadence Opus dfII

The GDS output from *NR* is converted to Opus dfII format. This permits custom edits (using *Virtuoso*) on the layout for DRC and LVS error corrections, which may not be completely removed by *NR*.

Taking advantage of the Opus dfII format, a Skill script has been written to convert the I/O cell views from "layout" to "layout_65u" in accordance to the 65.19µm pitch specifications. The "layout_65u" view has a coarser pitch, and is typically used in pre-production, research, or experimental circuits.



Pullup Cell Pulldown Cell

Figure 11-5. Pullup and pulldown cells inserted during synthesis.

11.8. Pullup and Pulldown Cells

The synthesis this design flow introduces a large number of pullup and pulldown cells. These cells are necessary to represent tied-high or tied-low signals (e.g. control signals, synthesized ROMs and constants) in the synthesis environment, which does not have explicit references to the power and ground supplies. The pullup and pulldown cells are simply minimum-sized PMOS and NMOS transistors with gates permanently tied to ground and the power supply respectively, as shown in Figure 11-5.

When left in the circuit description, the physical implementation tools will handle these pullup/pulldown cells as standard cells. The router connects the outputs of these cells as local signals, which is a waste of resources. Both power and ground supplies are traditionally designed to be easily assessed through the top and bottom rails of each standard cell. Moreover, as the design gets larger, a single pullup/pulldown cell may be required to drive a large number of gates, spaced far apart from each other. This increases the potential of antenna process violations due to the substantial length of interconnects. Overloading of the minimal sized transistors, though, is not an issue. The pullup/pulldown cells drive the gates of other devices with a constant voltage, and are not subjected to high currents.

The obvious solution to the above problem is to remove the pullup/pulldown cells, and connect the driven gates directly to the nearest available power or ground supply. The power and ground grid is resilient against antenna process violations due to the wide presence of substrate contacts, which act as antenna protection diodes. The Verilog output from the synthesis stage is parsed to delete most pullup/pulldown cells and have their previous outputs declared with *supply1* and *supply0* constructs. These declarations define a signal either as a tied-high or tied-low respectively. In a few rare cases, the original outputs of the pullup/pulldown cells are defined as references to a signal array (e.g. dout[4]). Since the Verilog format does not permit use of *supply1* and *supply0* declarations with an array reference, these occurrences are left unaltered.

Nevertheless, the removal of pullup/pulldown cells creates a resolution problem during LVS verification. This is also an artifact of the *Insecta* front end, which needs to be addressed eventually. The generation of the VHDL files using *XSG* creates a large number of redundant control signals. These are usually in the form of input or output valid bits, reset and enable signals. In a datapath-oriented design with limited amount of data/clock gating, these control signals are almost always uniformly set to constant values. The constants are provided from the outputs of pullup/pulldown cells. In many cases, a global enable bit may be logically AND'd with a control reset signal to provide a local enable bit. In many cases, the output is left unconnected by the synthesis step.

With the presence of pullup/pulldown cells, *NR* routes these constant signals as local interconnects, and attaches a name property that reflects the hierarchical connection. The removal of pullup/pulldown cells causes these constants to be simply renamed as either "Vdd" or "Gnd". The subsequent *gds* format describes a substantial number of cells sharing identical logical functions, with ports connected to the same "Vdd" or "Gnd" signals. As a consequence, a flat LVS check is likely to face ambiguities, where parts of the circuit can be interchanged without affecting the connectivity. It certain designs, Calibre LVS will examine the device properties in order to resolve these ambiguities. This is only applied to groups of ambiguous elements up to a maximum number, specified in the rule file with the "LVS Property Resolution Maximum" parameter. In this design, the clearest sign of failure to resolve ambiguities was when Calibre LVS reported a 100% connectivity correspondence, but found mismatch of properties (primarily gate widths) in a list of devices. These devices are found to be contained within standard cells that have inputs connected to either "Vdd" or "Gnd", and unconnected outputs. Setting the "LVS Property Resolution Maximum" to 1000 rectifies this problem.

References

1. A. Blanksby and C. J. Howland, "A 690-mW 1-Gbit/s 1024-bit rate-1/2 low density parity check code decoder," *IEEE J. Solid-Stage Circuits*, vol. 37, no. 3, Mar 2002, pp. 404-412.
2. A. Dholakia, E. Eleftheriou, and T. Mittelholzer, "On iterative decoding for magnetic recording channels," in *Proc. of the 2nd International Symposium on Turbo Codes and Related Topics*, 2000, pp. 219–226.
3. A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Trans. Inform. Theory*, vol. IT-13, pp. 260–269, Apr. 1967.
4. A. Viterbi, "An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes," *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 2, pp. 260–264, Feb. 1998.
5. A. Worm, H. Lamm, N. Wehn, "A high-speed MAP architecture with optimized memory size and power consumption," in *Proc. IEEE SiPS*, 2000, pp. 265–274.
6. A. Yeung, and J. M. Rabaey, "A 210 Mb/s radix-4 bit-level pipelined viterbi decoder," in *Proc. IEEE* ISSCC, 1995, pp. 88–89, 344, 440.
7. B. Frey and F. Kschischang, "Early detection and trellis splicing: Reduced-complexity iterative decoding," *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 2, pp. 153–159, Feb. 1998.
8. B. Lee, S. Bae, S. Kang, and E. Joo, "Design of swap interleaver for turbo codes," *Electronics Letters*, vol. 35, no. 22, IEE, pp. 1939–1940, Oct. 1999.
9. B. Sklar, "A primer on turbo code concepts," *IEEE Communications Magazine*, Dec. 1997.
10. C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: Turbo-codes," *IEEE Trans Comms.*, vol. 44, no. 10, pp.1261–1271, Oct. 1996.
11. C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo codes," in *Proc. IEEE ICC,* 1993, vol. 2, May 1993, pp.1064–1070.
12. C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo codes," in *Proc. IEEE Int. Conf. Communications*, 1993.
13. C. Berrou, A. Glavieux, and P. Thitmajshima, "Near Shanon limit error-correcting coding and decoding: Turbo codes," in *Proc. IEEE Int. Conf. on Communication*, pp. 947–951, Jun. 1998.
14. C. Berrou, P. Adde, E. Angui, and S. Faudeil, "A low complexity soft-output viterbi decoder architecture," *IEEE International Conference on Communications*, 1993, pp. 737–740.
15. C. Heegard and S. Wicker, "Turbo coding," *Kluwer Academic Publishers*, pp. 71–73, 1999.
16. D. Garrett and M. Stan, "Low power architecture of the soft-output viterbi algorithm," *ACM ISLPED98*, pp. 262–267, 1998.

17. D. Garrett and M. Stan, "A 2.5 Mb/s, 23 mW SOVA traceback chip for turbo decoding applications," in *Proc. IEEE ISCAS*, 2001, pp. 61–64.
18. D. Heidel, S Dhong, P. Hofstee, M. Immediato, K. Nowka, J. Silberman, and K. Stawiasz, "High speed serializing/de-serializing design-for-test method for evaluating a 1 GHz microprocessor," in *Proc. IEEE VLSI Test Symposium*, 1998, pp. 234–238.
19. D. J. C. Mackay and M. C. Davey, "Evaluation of Gallager codes for short block length and high rate applications," in *Proc. of IMA workshop on Codes, Systems and Graphical Models*, 1999.
20. D. J. C. Mackay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *IEE Electronics Letters*, vol. 33, no. 6, pp. 457–458, Mar. 1997.
21. D. Mackay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. on Information Theory*, vol. 45, no. 2, Mar. 1999.
22. E. Casseau and E. Luthi, "Architecture of a high-rate VLSI viterbi decoder," in *Proc. IEEE ICECS*, 1996, pp. 21–24.
23. E. Yeo, B. Nikolic, and V. Anantharam, "Architectures and implementations of low-density parity check decoding algorithms," *IEEE MWSCAS*, Aug. 2002.
24. E. Yeo, P. Pakzad, B. Nikolic, and V. Anantharam, "High throughput low-density parity-check architectures," in *Proc. IEEE Globecom*, 2001, pp.3019–3024.
25. E. Yeo, P. Pakzad, B. Nikolić, and V. Anantharam, "High throughput low-density parity check decoder architectures," in *Proc. IEEE Globecom*, 2001, pp.
26. E. Yeo, P. Pakzad, B. Nikolic, and V. Anantharam, "VLSI architectures for iterative decoders in magnetic recording channels," *IEEE Trans. Magnetics*, vol. 37, no. 2, pp. 748–755, Mar. 2001.
27. E. Yeo, P. Pakzad, B. Nikolić, and V. Anantharam, "VLSI architectures for iterative decoders in magnetic recording channels," *IEEE TMRC*, 2000.
28. E. Yeo, P. Pakzad, B. Nikolic, V. Anantharam, "VLSI architectures for iterative decoders in magnetic recording channels," in *Digests of The Magnetic Recording Conference, TMRC 2000, on Magnetic Recording Systems*, 2000, pp. E6.
29. E. Yeo, S. Augsburger, W. R. Davis, and B. Nikolic, "500 Mb/s soft output viterbi decoder," *IEEE ESSCIRC*, Sep. 2002.
30. E. Yeo, S. Augsburger, W. R. Davis, and B. Nikolic, "Implementation of high throughput soft output viterbi decoders," *IEEE SIPS*, Oct. 2002.
31. G. Al-Rawi; J. Cioffi, and M. Horowitz, "Optimizing the mapping of low-density parity check codes on parallel decoding architectures," in *Proc. IEEE ITCC*, 2001, pp. 578–586.
32. G. Fettwies, "Algebraic survivor memory management design for viterbi detectors," *IEEE Trans. on Communications*, vol. 43, no. 9, pp. 2458–2463, Sep. 1995.
33. G. Masera, G. Piccinini, M. Roch, and M. Zamboni, "VLSI architectures for turbo codes," *IEEE Trans. on VLSI Systems*, vol. 7, no. 3, pp. 369–379, Sep. 1999.
34. I. Lee and J. L. Sonntag, "A new architecture for the fast viterbi algorithm," in *Proc. IEEE Globecom*, 2000, pp. 1664–1668.
35. J. Boutros, O. Pothier, and G. Zemor, "Generalized low density (Tanner) codes," in *Proc. IEEE ICC*, 1999, pp. 441–445.

36. J. Fan and J. Cioffi, "Constrained coding techniques for soft iterative decoders," *GLOBECOM*, 1999, vol. 16, pp. 723–727.
37. J. Fan, "Constrained coding and soft iterative decoding for storage," *Dissertation for PhD, Stanford University*, Dec. 1999.
38. J. Hagenauer and P. Hoeher, "A viterbi algorithm with soft-decision outputs and its applications," in *Proc. IEEE GLOBECOM*, 1989, pp. 47.11–47.17.
39. J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Trans. on Information Theory*, pp. IT 42:429–42:445, 1996.
40. J. Pearl, "Probabilistic reasoning in intelligent systems: Networks of plausible inference," San Mateo, CA, Morgan Kaufmann, 1988.
41. K. Abend and B. D. Fritchman, "Statistical detection for communication channels with intersymbol interference," in *Proc. IEEE*, 1970, vol. 58, no. 5.
42. K. Andrews, C. Heegard, and D. Kozen, "Interleaver design methods for turbo codes," *IEEE International Symposium on Information Theory*, 1998, pp. 420.
43. K. Tzou and J. Dunham, "Sliding block decoding of convolutional codes," *IEEE Trans. on Communications*, vol. COM-29, no. 9, pp.1401–1403, Sep. 1981.
44. K. Tsukano, T. Nishiya, T. Hirai, and T. Nara, "Simplified EEPR viterbi detector based on a transformed radix-4 trellis for a disk drive," *IEEE Trans Magnetics*, vol. 35, no. 5, pt. 3, pp. 4387–4401, Sep. 1999.
45. L. Bahl, J. Cocke, F. Jelinek, and R. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. on Inform. Theory*, pp. 284–287, Mar. 1974.
46. L. Ping, W.K. Leung, and N. Phamdo, "Low density parity check codes with semi-random parity check matrix," *Electronic Letters*, vol. 35, no. 1, Jan. 1999.
47. M. Davey and D. Mackay, "Low-density parity check codes over GF(q)," *IEEE Communications Letters*, vol. 2, no. 6, pp. 165–167, Jun. 1998.
48. M. Fossorier, F. Burkert, S. Lin, and J. Hagenauer, "On the equivalence between SOVA and max-log MAP decodings," *IEEE Communications Letters*, vol. 2, no. 5, pp. 137–139, May 1998.
49. M. Lentmaier and Z. S. Ziganfirov, "Iterative decoding of generalized low-density parity-check codes," in *Proc IEEE ISIT*, 1998, pp. 149.
50. M. Oberg and P. Siegel, "Parity check codes for partial response channels," in *Proc. IEEE Global Telecommunications Conference*, pp. 717–722, 1999.
51. O. J. Joeressen and H. Meyr, "A 40 Mb/s soft-output viterbi decoder," *IEEE JSSC*, vol. 30, no.7, pp. 812–818, Jul. 1995.
52. P. Black and T. Meng, "A 1 Gb/s, 4-state, sliding block viterbi decoder," *IEEE Journal of Solid-State Circuits*, vol. 32, no. 6, pp.797–805, Jun. 1997.
53. P. Black and T. Meng, "A 140 Mb/s 32-state radix-4 viterbi decoder," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 12, pp.1877–1885, Dec. 1992.
54. P. Black, "Algorithms and architectures for high speed viterbi decoding," *Dissertation for PhD, Stanford University*, Mar. 1993.
55. P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain," in *Proc. IEEE ICC*, 1995, pp. 1009–1013.

56. R. G. Gallager, "Low density parity check codes," *IRE Trans. Inform. Theory*, vol. IT-8, pp. 21–28, Jan. 1962.

57. R. Lucas, "Iterative decoding of one-step majority logic decodable codes based on belief propagation," *IEEE Trans. On Comm.*, vol. 48, no. 6, Jun. 2000.

58. R. V. Joshi and W. Hwang, "Design considerations and implementations of a high performance dynamic register file," *IEEE 12th Int. Conf. on VLSI Design*, 1999, pp. 526–531.

59. R.M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. on Information Theory*, vol. IT-27, no. 5, pp. 533–547, Sep. 1981.

60. S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Soft-output decoding algorithms in iterative decoding of turbo codes," *TDA Progress Report*, pp. 42–124, Feb. 1996.

61. S. Dolinar and D. Divsalar, "Weight distributions for turbo codes using random and non-random interleaving," *TDA Progress Report*, vol. 42–122, pp. 56–65, Aug. 1995.

62. S. Lin and D. J. Costello Jr., "Error control coding: Fundamentals and applications," Prentice-Hall, Inc., 1983, pp. 388–426.

63. S. Lin, T. Kasami, T. Fujiwara, and M. Fossorier, "Trellises and trellis-based decoding algorithms for linear block codes," *Kluwer Academic Publishers*.

64. S. M. Aji and R. J. McEliece, "The generalized distributive law," *IEEE Trans. Inform. Theory*, vol. 46, no. 2, pp. 325–343, Mar. 2000.

65. S. Sridharan and L. R. Carley, "A 110 MHz 350mW 0.6mm CMOS 16-state generalized-target Viterbi detector for disk drive read channels," *IEEE Journal Solid-State Circuits*, vol.35, no.3, pp.362-370, Mar 2000.

66. T. Mittelholzer, A. Dholakia, E. Eleftheriou, "Reduced-complexity decoding of low density parity check codes for generalized partial response channels," *IEEE Trans. on Magnetics*, vol. 37, no. 2, Mar. 2001, pp. 721–728.

67. T. Ngo and I. Verbauwhede, "Turbo codes on the fixed point DSP TMS320C55x," in *Proc. IEEE SiPS*, 2000, pp. 255–264.

68. T. Souvignier, A. Friedmann, M. Oberg, P. Siegel, R. Swanson, and J. Wolf, "Turbo decoding for PR4: Parallel vs. serial concatenation," in *Proc. IEEE Int. Conf. Communication*, pp. 1638–1642, Jun. 1999.

69. T. Souvignier, M. Oberg, P. Siegel and R. Swanson, and J. Wolf, "Turbo decoding for partial response channels", *IEEE Trans. Of Comm.*, vol. 48, no. 8, Aug. 2000.

70. V. Franz and J. Anderson, "Concatenated decoding with a reduced-search BCJR algorithm," *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 2, pp. 186–195, Feb. 1998.

71. Viterbi, "An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes," *IEEE Journ. Sel. Areas in Communications*, vol. 16, no. 2, pp. 260–264, Feb. 1998.

72. W. Hwang and W. Henkels, "A 500MHz, 32-word x 64-bit, eight-port self-resetting CMOS register file," *IEEE Journal of Solid-State Circuits*, vol. 34, no. 1, pp. 56–67, Jan. 1999.

73. W. K. Taek, D. W. Kim, W. T. Kim, E. K. Joo, J. R. Choi, P. Choi, J. J. Kong, S. H. Choi, W. H. Chung, and K. W. Lee, "A modified two-step SOVA-based turbo

decoder for low power and high performance," in *Proc IEEE TENCON*, 1999, pp. 297–300.

74. W. R. Davis, N. Zhang, K. Camera, D. Markovic, T. Smilkstein, M. J. Ammer, E. Yeo, S. Augsburger, B. Nikolic, and R. W. Brodersen, "An automated design flow for high-throughput low-power dedicated signal processing systems," *IEEE Journ. Solid-State Circuits*, vol. 37, no. 3, pp. 420–431, Mar. 2002.
75. W. R. Davis, N. Zhang, K. Camera, D. Markovic, T. Smilkstein, M. J. Ammer, E. Yeo, S. Augsburger, B. Nikolic, and R. W. Brodersen, "An automated design flow for low-power, high-throughput dedicated signal processing systems," in *Proc. of the Asilomar Conf. on Signals, Systems and Computers*, 2001.
76. W. R. Davis, N. Zhang, K. Camera, F. Chen, D. Marković, N. Chan, B. Nikolić, and R. W. Brodersen, "A design environment for high throughput, low power dedicated signal processing systems," in *Proc IEEE CICC*, 2001, pp. 545–548.
77. Y. Kou, S. Lin and M. Fossorier, "Low density parity check codes based on finite geometries: A rediscovery and more," *IEEE Trans. on Information Theory*, Oct. 1999.
78. Y. Kou, S. Lin, and M. P.C. Fossorier, "Low density parity check codes based on finite geometries: A rediscovery," *IEEE International Symposium on Information Theory*, 2000, pp. 200.
79. Y. Li, B. Vuvetic, and Y. Sato, "Optimum soft-output detection for channels with intersymbol interference," *IEEE Trans. Inform. Theory*, vol. 41, no. 3, May 1995.
80. G Fettweis, R. Karabed, P.H. Siegel, and H.K. Thapar, "Reduced-complexity Viterbi detector architectures for partial response signaling," in *Proc. IEEE Global Telecommunications Conference, Singapore*, Nov 13-17, 1995, pp.559-563.
81. T. Gemmeke, M. Gansen, and T. Noll, "Implementation of scalable power and area efficient high-throughput Viterbi decoders," *IEEE Journal Solid-State Circuits*, vol.37, no.7, pp 941-948, Jul 2002.
82. T. Conway, "Implementation of high speed Viterbi detectors," *IEE Electronics Letters*, vol.35, no.24, Nov 25 1999, pp.2089-2090.
83. G. Feygin and P. Gulak, "Architectural tradeoffs for survivor sequence memory management in Viterbi decoders," *IEEE Trans. Communications*, vol.41, no.3, pp.425-429, Mar 1993.
84. M. Sipser and D. A. Spielman, "Expander codes," *IEEE Trans. Information Theory*, vol.42, pp.1710-1722, Nov 1996.
85. J. Chen, M. P. C. Fossorier, "Near optimum universal belief propagation based decoding of low-density parity check codes," *IEEE Trans. Communications,* vol.50, no.3, March 2002, pp.406-414.
86. T. Mittelholzer, A. Dholakia E. Eleftheriou, "Reduced-complexity decoding of low density parity check codes for generalized partial response channels," *IEEE Transactions on Magnetics,* vol.37, no.2, pt.1, March 2001, pp.721-8.
87. J. Rosenthal and P. O. Vontobel, "Constructions of regular and irregular LDPC codes using Ramanujan graphs and ideas from Margulis," *Proc. IEEE ISIT*, Washington, DC, USA, Jun. 24-29, 2001, p.5.
88. T. J. Richardson, M. A. Shokrollahi, and R. L. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Trans. Information Theory*, vol.47, pp.619-637, Feb. 2001.

89. F. Berens, A. Worm, H. Michel, and N. Wehn, "Implementation Aspects of Turbo-Decoders for Future Radio Applications," in Proc. VTC '99 Fall, Amsterdam, The Netherlands, Sept. 1999, pp. 2601--2605.

90. D. Raphaeli and Y. Zarai, "Combined turbo equalization and turbo decoding," *IEEE Communication Letters*, vol. 2, pp. 107-109, Apr. 1998.

91. S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Parallel concatenated trellis coded modulation," *Proc. IEEE Int. Conf. Communications*, Dallas, Tx, pp. 974-978, Jun 1996.

92. "Universal mobile telecommunication system (UMTS). Multiplexing and channel coding (TDD)", *ETSI 3GTS 25.222 document*, ver. 3.1.1, Release 1999.

93. S. Chung; G.D. Forney, T.J. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," IEEE Comm. Letters, vol.5, pp.58-60, Feb. 2001.

94. R. J. McEliece, D. J. Mackay, and J. Cheng, "Turbo decoding as an instance of Pearl's "belief propagation" algorithm," *IEEE Journ. on Selected Areas in Communications*, vol. 16, no. 2, Feb. 1998, pp. 140-152.

95. D. J. C. Mackay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *IEE Electronics Letters*, vol.33, no.6, pp.457-8, March 1997.

96. M. M. Mansour and N. R. Shanbhag, "Memory-efficient turbo decoder architectures for LDPC codes," *Proc. IEEE SIPS 2002*, San Diego, CA, Oct 2002.

97. C. Shannon, CE, "A Mathematical Theory of Communication," *The bell system technical journal*, vol. 27, pp. 379-423, 623-656, 1948.

98. M. Bickerstaff, et. al., "A unified turbo/Viterbi channel decoder for 3GPP mobile wireless in 0.18 μm CMOS," *IEEE Journ. Solid-State Circuits,* vol.37 no.11, pp1555-1564, Nov 2002.

99. C. Chang, K. Kuusilinna, B. Richards, and R.W. Brodersen, "Implementation of BEE: a Real-time Large-scale Hardware Emulation Engine," *Proc. FPGA 2003*, pp. 91-99, Feb 2003.

100. Xilinx System Generator version 3.1 for Simulink user guide, *Xilinx Inc.*, http://www.xilinx.com/ipcenter/dsp/ref_guide.pdf

101. D. J. C. Mackay and M. S. Postol, "Weakness of Margulis and Ramanujan-Margulis low-density parity-check codes," *Electronic Notes in Theoretical Computer Science*, vol. 74, 2003.

102. B. Vucetic and J. Yuan, "Turbo Codes: Principles & Applications," Kluwer Academic Publishers, May 2000.

103. J.M. Rabaey, A. Chandrakasan, B. Nikolic, "Digital Integrated Circuits: A Design Perspective," 2nd edition, Prentice-Hall 2002.

104. "Design Compiler User Guide," *Synopsys Inc.*, Mountain View, CA, 2003.

105. D. Divsalar and F. Pollara, "Multiple turbo codes for deep-space communications," *JPL TDA Progress Report*, pp. 42-121, May 1995.

106. S. Crozier, J. Lodge, P. Guinand and A. Hunt, "Performance of Turbo Codes with Relative Prime and Golden Interleaving Strategies", Proceedings of the 6th International Mobile Satellite Conference (IMSC '99), Ottawa, Ontario, Canada, pp. 268-275, June 16-18, 1999.

107. O. Y. Takeshita and D. J. Costello, "New deterministic interleaver designs for turbo codes," *IEEE Trans. Inform. Theory,* vol. 46, no. 6, Sep 2000, pp. 1988-2006.

108. J. Steensma, and C. Dick, "FPGA implementation of a 3GPP turbo codec," *Proc IEEE Thirty-Fifth Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, USA, 4-7 Nov 2001, pp.61-65.
109. Zhang and K. Parhi, "A 56Mbps (3,6)-Regular FPGA LDPC Decoder," *Proc. IEEE SIPS 2002*, San Diego, CA, USA, Oct 16-18, 2002, pp.127-132.
110. T. Bhatt, K. Narayanan, and N. Kehtarnavaz, "Fixed Point DSP Implementation of Low-Density Parity Check Codes," *Proc IEEE DSP2000*, Hunt, TX, USA, Oct 15-18, 2000.
111. H. A. Loeliger, F. Lustenberger, M. Helfenstein, and F. Tarkoy, "Probability propagation and decoding in analog VLSI," *IEEE Trans Inform. Theory*, vol.47, pp.837-843, Feb 2001.
112. J. Thorpe, "Design of LDPC graphs for hardware implementation," *Proc. IEEE ISIT,* Lausanne, Switzerland, Jun 30 – Jul 5, 2002, p.483.
113. J. E. Volder, "The CORDIC Trigonometric computing technique," *IRE Transactions EC-8*, pp 330-334. 1959.
114. B. Vasic, E. M. Kurtas, and A. V. Kuznetsov, "LDPC codes based on mutually orthogonal Latin rectangles and their application in perpendicular magnetic recording, " *IEEE Trans Magnetics*, vol. 38, No. 5, Sep 2002, pp. 2346-2348.
115. K. Kuusilinna, C. Chang, etc, " Winning the SoC Revolution: Real-time System-on-Chip Emulation," *Kluwer Academic Publishers*, Chap10, p. 229-253, 2003.
116. J. Li, K. R. Narayanan, E. Kurtas, and AC. N. Georghiades, "On the performance of the high-rate TPC/SPC and LDPC codes over partial response channels," *IEEE Trans. Comms*, vol 50, no. 5, pp. 723-735, May 2002.
117. R. Lynch, E. Kurtas, A. Kuznetsov, E. Yeo, and B. Nikolic, "The search for a practical iterative detector for magnetic recording," *Proc. IEEE The Magnetic Recording Conference*, Aug 2003.
118. Texas Instruments, "TMS320C6000 ™ Platform : Application Notes Abstract Turbo Decoder Coprocessor User's Guide," Aug 2001.
119. S. Borkar, "Design challenges of technology scaling," *IEEE Micro*, vol.19, no.4, pp.23-29, July-Aug. 1999.
120. M. Moerz, T. Gabara, R. Yan, J. Hagenauer, "An analog 0.25μm BiCMOS Tailbiting MAP Decoder," 2000. IEEE International Solid-State Circuits Conference, ISSCC 2000, Digest of Technical Papers, pp. 356-357, San Francisco, CA, February 7-9, 2000.
121. B. Bougard, A. Giulietti, V. Derudder, J-W. Weijers, S. Dupont, L. Hoolevoet, F. Catthoor, L. Van der Perre, H. De Man, R. Lauwereins, "A scalable 8.7nJ/bit 75.6Mb/s parallel concatenated convolutional (turbo) CODEC," *Proc. IEEE* ISSCC, 2003, pp. 88–89, 344, 440.