A 1Mbps 0.18µm CMOS Soft-Output Decoder for Product Turbo Codes

by

Peter Edward Bade

A thesis submitted in conformity with the requirements for the degree of Masters of Applied Science
Department of Electrical and Computer Engineering
University of Toronto

© Copyright by Peter Edward Bade 2009
A Product Turbo Code Decoder Application Specific Integrated Circuit

Peter Edward Bade
Masters of Applied Science
Department of Electrical and Computer Engineering
University of Toronto
2009

Abstract

A product turbo code (PTC) decoder application specific integrated circuit (ASIC) is designed in 0.18µm 1P6M CMOS with embedded SRAM. From simulation, an operating frequency of 73.1 MHz at typical conditions is obtained, yielding a throughput of 3.8 Mbps with 4 decoding iterations, while consuming 103.4 mW. The total area is 5.13 mm$^2$. Assuming the ASIC would be used as a hard macro, the area could be reduced to 1.7 mm$^2$. The ASIC was tested at 20 MHz under typical conditions, which resulted in a throughput of 1.0 Mbps at 1.8V supply while consuming 36.6 mW.

By making a slight modification, this design can be easily scaled to support IEEE 802.16d WiMAX. Allow for this, and moving to a 45nm process an estimated throughput of 9.44 Mbps with 4 iterations can be obtained. Total macro area would be approximately 0.11 mm$^2$. 
Acknowledgments

I would like to thank my advisor, Professor Glenn Gulak, for his encouragement and guidance during the preparation of this thesis.

I would also like to acknowledge the assistance of several current and former graduate students. Warren Gross provided advice with the information theory aspects of this thesis. His publications were also an invaluable source of information. Kostas Pagiamtzis provided advice regarding the digital design flow and the integration of SRAM into the flow. Vincent Gaudet provided assistance with the information theory and simulation aspects of this work. Dimpesh Patel provided test software [1] and advice that was of great assistance in laboratory testing. Nasim Nikkhoo provided much assistance with obtaining the die photomicrograph presented in this thesis.

Furthermore, I would like to thank Jeetendar Narsinghani and Marius Jarosz of the Canadian Microelectronics Corporation (CMC) for their help with testing and debugging this design. I would also like to thank CMC for providing fabrication and packaging resources.

I would also like to acknowledge financial support from the University of Toronto.
Table of Contents

Acknowledgments .................................................................................................................. iii
Table of Contents .................................................................................................................. iv
List of Figures ......................................................................................................................... x
Table of Acronyms .................................................................................................................. xii
Table of Symbols .................................................................................................................. xiv

Chapter 1: Introduction ......................................................................................................... 1
   1.1 Thesis Objective ................................................................................................. 1
   1.2 Literature Review ............................................................................................. 2
   1.3 Thesis Outline ................................................................................................. 4

Chapter 2: Turbo Decoding of Product Codes ..................................................................... 5
   2.1 Block Codes and the Forward-Backward Algorithm ........................................... 5
      2.1.1 A (15,11) Block Code ............................................................................ 5
      2.1.2 The Forward-Backward Algorithm for Block Codes ............................. 7
      2.1.3 The Forward-Backward Algorithm and the Log Likelihood Ratio ........ 7
      2.1.4 Branch Metrics ...................................................................................... 8
      2.1.5 Forward Recursion State Metrics .......................................................... 8
      2.1.6 Backward Recursion State Metrics ....................................................... 9
      2.1.7 Log Likelihood Ratio Calculation ......................................................... 9
      2.1.8 The Forward-Backward Algorithm in the Log Domain ....................... 10
      2.1.9 Branch Metrics in the Log Domain ....................................................... 10
      2.1.10 Forward Recursion State Metrics in the Log Domain ....................... 11
2.1.11 Backward Recursion State Metrics in the Log Domain ................................. 11
2.1.12 Log Likelihood Ratio Calculation in the Log Domain ................................. 12
2.1.13 A Comparison of Forward-Backward Algorithms ........................................ 12
2.1.14 A Forward-Backward Decoder ................................................................. 12
2.2 Turbo Decoding of Product Codes using the Forward-Backward Algorithm ............ 13
2.2.1 Product Codes ................................................................................................. 13
2.2.2 Turbo Decoding .............................................................................................. 15
2.3 Software Simulation ............................................................................................ 17
2.3.1 Baseline Results ............................................................................................. 19
2.3.2 Coding Performance and Constituent Codes .................................................. 20
2.4 Comparison with Convolutional Turbo Codes .................................................... 20
2.5 Industry Standard Product Turbo Codes ........................................................... 21
2.5.1 IEEE 802.16 WiMAX ................................................................................. 21
2.6 Summary ........................................................................................................... 22

Chapter 3: Architecture ............................................................................................. 23

3.1 Fixed-Point and Quantization Effects .................................................................... 23
3.1.1 Fixed-Point Arithmetic ................................................................................... 23
3.1.2 Truncation Effects .......................................................................................... 24
3.1.3 Overflow and Saturation Effects ..................................................................... 24
3.1.4 Normalization ................................................................................................ 24
3.1.5 Analysis of the Floating-Point Simulation .................................................... 26
3.1.6 MAX* In Fixed-Point .................................................................................... 27
3.2 A Software Fixed-Point Turbo Decoder ............................................................... 28
3.3 Fixed-Point Simulation Results ........................................................................... 31
3.4 Forward-Backward Decoder Architecture ......................................................... 32
Chapter 4: ASIC Design ................................................................. 46

4.1 Implementation Technology .................................................. 46

4.2 EDA Methodology ................................................................. 46

4.3 HDL Design and Verification ................................................ 49

4.4 ASIC Design Objectives ........................................................ 49

4.4.1 Maximizing Operating Frequency ....................................... 49

4.4.2 Minimizing Area .............................................................. 49

4.4.3 Minimizing Power Consumption ....................................... 50

4.5 Synthesis ............................................................................. 50

4.6 Design for Test Features (DFT) .............................................. 53

4.6.1 Scan Chain Registers ........................................................ 53

4.6.2 Black Box Cells .............................................................. 54

4.6.3 Shadow Logic ................................................................. 55

4.6.4 Registering the Inputs and Outputs ................................... 56

4.6.5 Multiplexing the Inputs and Outputs .................................. 56

4.6.6 Synopsys Shadow Logic DFT ............................................ 57

4.6.7 Automatic Test Pattern Generation (ATPG) for Logic Cells .... 57

4.6.8 Memory Testing .............................................................. 57

4.7 Physical Design ................................................................... 58

4.7.1 Power and Ground Distribution ....................................... 58

4.7.2 Signal Integrity ............................................................... 58

4.8 Floorplanning ..................................................................... 59

4.9 Placement .......................................................................... 60

4.9.1 Clock Distribution via a Clock Tree ................................... 62

4.10 Routing ............................................................................. 63

4.11 Layout - LVS and DRC ........................................................ 64
4.12 Post Layout Results ........................................................................................................ 64
   4.12.1 Area and Cell Statistics ....................................................................................... 64
   4.12.2 Critical Path ........................................................................................................ 67
   4.12.3 Throughput .......................................................................................................... 68
   4.12.4 Power Estimates and IR Drop Analysis ............................................................. 69
   4.12.5 Adjusted Area for Custom SRAM Blocks and IO Ring .................................... 69

4.13 Fabrication and Packaging ........................................................................................ 70

4.14 Testing and Test Results ............................................................................................ 70

4.15 Technology Scaling ................................................................................................... 73
   4.15.1 Advanced Techniques For Minimizing Power Consumption ............................ 74

4.16 Hardware Scaling to Support Other Product Codes Including WiMAX .................. 75
   4.16.1 Area Estimates .................................................................................................... 75
   4.16.2 Throughput Estimates ....................................................................................... 76

4.17 Summary ..................................................................................................................... 77

Chapter 5: Conclusions ....................................................................................................... 78
   5.1 Summary and Conclusions ....................................................................................... 78
   5.2 Contributions ............................................................................................................ 78
   5.3 Future Work .............................................................................................................. 79
      5.3.1 Alternatives to the Forward-Backward Algorithm ............................................. 79
      5.3.2 Normalization of State Metrics ........................................................................ 79
      5.3.3 Parallel Decoding of Forward and Backward Recursions ............................... 80
      5.3.4 Radix-4 Trellis Processing ................................................................................ 80
      5.3.5 Interleaver Design ............................................................................................ 80
      5.3.6 Semicustom Design ......................................................................................... 80
      5.3.7 EDA Software .................................................................................................. 81
5.3.8 Summary Impact .......................................................... 81

Appendix 1: Analysis of Floating-Point Simulation ........................................ 82
Appendix 2: Fixed-Point Simulation Results .............................................. 83
Appendix 3: IR Drop Analysis .............................................................. 84

References ........................................................................ 86
List of Figures

Figure 1.1: Model of A Digital Communication System ................................................................. 1

Figure 2.1: The Encoder for a (15,11) Block Code ................................................................. 5

Figure 2.2: Path in the Trellis for Codeword 101001011101101 .............................................. 7

Figure 2.3: Simplified Block Diagram of a Forward-Backward Decoder .................................. 13

Figure 2.4: Row-Column Interleaver ......................................................................................... 14

Figure 2.5: Block diagram of a Turbo Decoder .......................................................................... 15

Figure 2.6: Flowchart of Product Code Simulation .................................................................... 18

Figure 2.7: Bit Error Rate for Floating-Point Simulation ......................................................... 19

Figure 2.8: WiMAX Product Turbo Code .................................................................................. 21

Figure 3.1: Block Diagram of a Forward-Backward Decoder .................................................. 25

Figure 3.2: Fixed-Point Simulation Flow .................................................................................... 29

Figure 3.3: Block Diagram of a Fixed-Point Turbo Decoder .................................................. 31

Figure 3.4: Bit Error Rate for Fixed-Point Simulations (Using MAX* Approximation) .......... 32

Figure 3.5: ACS Circuit Diagram ............................................................................................. 33

Figure 3.6: Trellis Processing Block Diagram ............................................................................ 34

Figure 3.7: Conventional Schedule ........................................................................................... 34

Figure 3.8: Modified Schedule .................................................................................................. 35

Figure 3.9: Forward-Backward Decoder Architecture ............................................................ 36

Figure 3.10: Modified ACS Circuit Diagram .............................................................................. 37
# Table of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>3GPP</td>
<td>Third Generation Partnership Program</td>
</tr>
<tr>
<td>ACS</td>
<td>Add Compare Select</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>ATPG</td>
<td>Automated Test Pattern Generation</td>
</tr>
<tr>
<td>AWGN</td>
<td>Additive White Gaussian Noise</td>
</tr>
<tr>
<td>BCH</td>
<td>Bose-Chaudhuri-Hocquenghem</td>
</tr>
<tr>
<td>BCJR</td>
<td>Bahl, Cocke, Jelinek and Raviv algorithm</td>
</tr>
<tr>
<td>BER</td>
<td>Bit Error Rate</td>
</tr>
<tr>
<td>BPS</td>
<td>Bits Per Second</td>
</tr>
<tr>
<td>BPSK</td>
<td>Binary Phase Shift Keying</td>
</tr>
<tr>
<td>CMC</td>
<td>Canadian Microelectronics Corporation</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal Oxide Semiconductor</td>
</tr>
<tr>
<td>CTC</td>
<td>Convolutional Turbo Code</td>
</tr>
<tr>
<td>DEF</td>
<td>Design Exchange Format</td>
</tr>
<tr>
<td>DFT</td>
<td>Design For Test</td>
</tr>
<tr>
<td>DRC</td>
<td>Design Rule Check</td>
</tr>
<tr>
<td>EDA</td>
<td>Electronic Design Automation</td>
</tr>
<tr>
<td>FO4</td>
<td>Fan-Out 4 Delay</td>
</tr>
<tr>
<td>GDS</td>
<td>Graphical Display System</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>IO</td>
<td>Input Output</td>
</tr>
<tr>
<td>LEF</td>
<td>Library Exchange Format</td>
</tr>
<tr>
<td>LLR</td>
<td>Log Likelihood Ratio</td>
</tr>
<tr>
<td>LTE</td>
<td>Long Term Evolution</td>
</tr>
<tr>
<td>LUT</td>
<td>Look Up Table</td>
</tr>
<tr>
<td>LVS</td>
<td>Layout Versus Schematic</td>
</tr>
<tr>
<td>MAP</td>
<td>Maximum A Posteriori</td>
</tr>
<tr>
<td>MEMBIST</td>
<td>Memory Built In Self Test</td>
</tr>
<tr>
<td>PGA</td>
<td>Pin Grid Array</td>
</tr>
<tr>
<td>PTC</td>
<td>Product Turbo Code</td>
</tr>
<tr>
<td>PVT</td>
<td>Process / Voltage / Temperature</td>
</tr>
<tr>
<td>RM</td>
<td>Reed-Muller</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal to Noise Ratio</td>
</tr>
<tr>
<td>SoC</td>
<td>System on a Chip</td>
</tr>
<tr>
<td>SOVA</td>
<td>Soft Output Viterbi Algorithm</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static Random Access Memory</td>
</tr>
<tr>
<td>TLF</td>
<td>Timing Library Format</td>
</tr>
<tr>
<td>UMTS</td>
<td>Universal Mobile Telecommunications System</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integration</td>
</tr>
</tbody>
</table>
### Table of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_k(s)$</td>
<td>Forward state metric in the log domain</td>
</tr>
<tr>
<td>$\text{Apr}(u_k)$</td>
<td>A priori information about $u_k$</td>
</tr>
<tr>
<td>$B_k(s')$</td>
<td>Backward state metric in the log domain</td>
</tr>
<tr>
<td>$C$</td>
<td>Capacitance</td>
</tr>
<tr>
<td>$C_k$</td>
<td>Coded bits for the product code</td>
</tr>
<tr>
<td>$c_k$</td>
<td>Coded bits for the block code</td>
</tr>
<tr>
<td>$E_b/N_0$</td>
<td>Signal to noise ratio (usually expressed in dB)</td>
</tr>
<tr>
<td>$F$</td>
<td>Frequency</td>
</tr>
<tr>
<td>$G$</td>
<td>Branch metric in the log domain</td>
</tr>
<tr>
<td>$g(x)$</td>
<td>The generator polynomial for a cyclic code</td>
</tr>
<tr>
<td>$K$</td>
<td>The number of check bits in a block code</td>
</tr>
<tr>
<td>$L(U_k)$</td>
<td>Extrinsic information for the product code</td>
</tr>
<tr>
<td>$L(u_k)$</td>
<td>Extrinsic information for the block code</td>
</tr>
<tr>
<td>$L(\hat{U}_k)$</td>
<td>Best estimate of $U_k$ for the product code</td>
</tr>
<tr>
<td>$L(\hat{u}_k)$</td>
<td>Best estimate of $u_k$ for the block code</td>
</tr>
<tr>
<td>$L_0(U_k)$</td>
<td>Extrinsic information for the product code after horizontal decoding</td>
</tr>
<tr>
<td>$L_0(u_k)$</td>
<td>Extrinsic information for the block code after horizontal decoding</td>
</tr>
<tr>
<td>$L_1(U_k)$</td>
<td>Extrinsic information for the product code after vertical decoding</td>
</tr>
<tr>
<td>$L_1(u_k)$</td>
<td>Extrinsic information for the block code after vertical decoding</td>
</tr>
<tr>
<td>$N$</td>
<td>The total number of bits in a block code</td>
</tr>
<tr>
<td>$n_k$</td>
<td>Noise vector</td>
</tr>
<tr>
<td>$P$</td>
<td>Power</td>
</tr>
<tr>
<td>$P_0$</td>
<td>Parity bits for the product code, horizontal encoding</td>
</tr>
<tr>
<td>$P_1$</td>
<td>Parity bits for the product code, vertical encoding</td>
</tr>
<tr>
<td>$p_k$</td>
<td>Parity bits for the block code</td>
</tr>
<tr>
<td>$R_D$</td>
<td>Seed for random data generation</td>
</tr>
<tr>
<td>$R_N$</td>
<td>Seed for random noise generation</td>
</tr>
<tr>
<td>$\hat{U}_k$</td>
<td>Hard decision of bit estimate $L(\hat{U}_k)$</td>
</tr>
<tr>
<td>$U_k$</td>
<td>Data bits for the product code</td>
</tr>
<tr>
<td>$u_k$</td>
<td>Data bits for the block code</td>
</tr>
<tr>
<td>$\hat{u}_k$</td>
<td>Hard decision of bit estimate $L(\hat{u}_k)$</td>
</tr>
<tr>
<td>$V$</td>
<td>Voltage</td>
</tr>
<tr>
<td>$x_k(s',s)$</td>
<td>Expected bit, based on the trellis and state transition table</td>
</tr>
<tr>
<td>$Y_{D1}$</td>
<td>Received data bits for the product code</td>
</tr>
<tr>
<td>$Y_k$</td>
<td>Received bits for the product code</td>
</tr>
</tbody>
</table>
\( y_k \)  
Received bits for the block code

\( Y_{P0} \)  
Received parity bits of the horizontal encoding for the product code

\( Y_{P1} \)  
Received parity bits of the vertical encoding for the product code

\( \alpha_k(s) \)  
Forward state metric

\( \beta_k(s') \)  
Backward state metric

\( \gamma \)  
Branch metric

\( \sigma \)  
\( \sigma \) is the standard deviation of a Gaussian distribution, \( \sigma^2 \) is the variance
Chapter 1: Introduction

Since the emergence of digital communication systems, there has been a need for error correction. This is due to the non-ideal nature of practical communication channels, which are often corrupted by noise. Error correction attempts to compensate for the errors introduced by this noise. An example digital communication system is shown in Figure 1.1.

![Figure 1.1: Model of A Digital Communication System](image)

At the transmitter, the data $u_k$ (in the form of bits) to be transmitted are fed into the encoder, which adds redundant information to the bit stream. This series of bits $c_k$, is then mapped to “+1” and “-1” for a logical 1 or 0 respectively and transmitted over the channel. At the receiver, the received bit $y_k$ is processed by the decoder to calculate hard decision bit estimates $\hat{u}_k$. The purpose of the decoder is to use the redundant information in the bit stream to correct any errors introduced during transmission due to noise $n_k$.

In 1993, turbo codes were invented by Berrou, Glavieux, and Thitimajshima [2]. These codes represented a breakthrough in coding theory as they performed closer to the Shannon channel capacity [3] than any other practical code. They are based on three key ideas: recursive systematic component codes, interleaving and iterative processing. In this thesis, we apply the interleaving and iterative principles of turbo codes to product codes.

1.1 Thesis Objective

The objective of this thesis is to design a product turbo code decoder application specific integrated circuit (ASIC). We wish to achieve the best possible bit error rate (BER) while maximizing throughput, and minimizing the area and power consumption of the final design.
1.2 Literature Review

With the wide deployment of wireless networks, there has been tremendous interest in designing turbo decoders. Convolutional Turbo Codes (CTC) and Product Turbo Codes (PTC) are widely used in many communications standards.

CTCs are used in both the Universal Mobile Telecommunication System / 3rd Generation Partnership Project (UMTS/3GPP) [4] and in the IEEE802.16d [5] WiMAX wireless networking standard. The CTC used in UMTS/3GPP is a binary convolutional code, while the one used in WiMAX is a dual binary convolutional code. Much work has been focused on these two standards.

In the case of PTCs, several are based on Hamming codes, and are used in WiMAX, in the Homeplug Powerline Alliance broadband home networking over power line specification [6] and in several proprietary satellite communication systems [7]. The PTC for WiMAX supports either a (16,11), (32,26) or a (64,57) extended Hamming code which are very similar to the (15,11) Hamming codes used in this thesis. Other forms of PTCs based on either Bose-Chaudhuri-Hocquenghem (BCH) codes or Reed-Muller (RM) codes, are used for high bit rate applications, such as terrestrial and submarine optical data transmission [8] [9].

We will briefly survey the current state of the art and other noteworthy publications. Some of these turbo decoders are from academic sources and others are from the data sheets of commercially available parts.

Most of the PTC decoders described in the recent literature are based on the Chase algorithm. This was developed by Chase [10] in 1972 for linear block codes and was refined and applied to PTCs by Pyndiah [11]. This algorithm is suboptimal, but it is much less area and power intensive than the Forward-Backward algorithm which is used in this thesis. It has been incrementally improved by various authors ([12] [13] [14] [9]) and is widely used in a variety of commercial available decoders.

More recently, after the ASIC described in this thesis had been fabricated, another method of maximum a posteriori decoding of Hamming codes was found. This is based on the fast Hadamard transform and was described in [15]. It has a decoding complexity of $n \log n$ as opposed to the Forward-Backward algorithm that scales as $n^2$ where $n$ is the code length.
However, there are several challenges involved with a hardware implementation of this algorithm since it makes extensive use of exponentiations and natural logarithms. As of this writing, there have been no reported results of hardware implementation using this approach.

A final method of decoding this code is to use the Green algorithm or Green Machine as described in [16].

Table 1.1 presents recent PTC and CTC decoders and their performance.

<table>
<thead>
<tr>
<th>Author</th>
<th>Year</th>
<th>Type</th>
<th>Algorithm</th>
<th>Clock Frequency (MHz)</th>
<th>Throughput (Mbps)</th>
<th>Technology</th>
<th>Area</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turbo-concepts [17]</td>
<td>2008</td>
<td>Hamming and BCH PTC</td>
<td>Chase</td>
<td>N/A</td>
<td>40 to 420</td>
<td>IP Block</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>AHA Comtech [18]</td>
<td>2008</td>
<td>Hamming and BCH PTC</td>
<td>Chase</td>
<td>N/A</td>
<td>36 to 311</td>
<td>IP Block</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>[8] Tagami et al.</td>
<td>2004</td>
<td>BCH PTC</td>
<td>Chase</td>
<td>N/A</td>
<td>10 Gbps</td>
<td>200nm SiGe BiCMOS</td>
<td>81 mm²</td>
<td>6.7 W</td>
</tr>
<tr>
<td>Kim et al [19]</td>
<td>2008</td>
<td>WiMAX CTC</td>
<td>Forward-Backward</td>
<td>200</td>
<td>24</td>
<td>130nm CMOS</td>
<td>2.24 mm²</td>
<td>274 mW</td>
</tr>
<tr>
<td>Lin et al [20]</td>
<td>2008</td>
<td>WiMAX CTC</td>
<td>Forward-Backward</td>
<td>100</td>
<td>115</td>
<td>130nm CMOS</td>
<td>2.57 mm²</td>
<td>116 mW</td>
</tr>
<tr>
<td>Martina et al [21]</td>
<td>2008</td>
<td>UMTS/3GPP and WiMAX CTC</td>
<td>Forward-Backward</td>
<td>200</td>
<td>12 (UMTS) 90 (WiMAX)</td>
<td>130nm CMOS</td>
<td>204k gates</td>
<td>N/A</td>
</tr>
<tr>
<td>Sun et al [22]</td>
<td>2008</td>
<td>UMTS/3GPP CTC</td>
<td>Forward-Backward</td>
<td>200</td>
<td>711</td>
<td>65nm CMOS</td>
<td>4.7M gates</td>
<td>N/A</td>
</tr>
<tr>
<td>Benkeser et al [23]</td>
<td>2008</td>
<td>UMTS/3GPP CTC</td>
<td>Forward-Backward</td>
<td>246</td>
<td>20 (UMTS)</td>
<td>130nm CMOS</td>
<td>1.2 mm²</td>
<td>58 mW</td>
</tr>
</tbody>
</table>

Table 1.1: Summary of Turbo Decoders

1.3 Thesis Outline

The remainder of this thesis is divided into four principal sections. In chapter 2, we will discuss the theoretical aspects of turbo decoding and construct a software model to simulate the decoder.
Chapter 3 describes the practical and architectural issues involved in defining the digital hardware architecture, based on the results of chapter 2. Chapter 4 examines the theoretical and practical digital design issues in creating the decoder ASIC. The conclusions of this thesis and a discussion of lessons learned are presented in Chapter 5.
Chapter 2: Turbo Decoding of Product Codes

The objective of this chapter is to demonstrate the theoretical ideas behind turbo decoding of product codes. We will start with a review of block codes in section 2.1 and show how they can be decoded using the Forward-Backward algorithm. In section 2.2 we will show how block codes can be combined to form a product code, and how they can be iteratively decoded using the Forward-Backward algorithm. Section 2.3 will present simulation results.

2.1 Block Codes and the Forward-Backward Algorithm

2.1.1 A (15,11) Block Code

A (15,11) cyclic code with generator \( g(x) = x^4 + x + 1 \) has been selected as our reference block code, since it is well studied in the literature and BER curves are readily available for it [24] [25] [26] [27].

This code is a Hamming code with Hamming distance 3, which means it is capable of correcting a single error.

The encoder begins operation at time \( t = 0 \) with \( \text{cntrlM}=0 \), \( \text{cntrlS}=1 \) and the flip flops \( S_0S_1S_2S_3 \) reset to 0000. This allows the input bits to pass though to the output while simultaneously being fed into the shift register. After 11 clock cycles, at time \( t = 10 \), all the input bits have been transmitted and the parity bits have been generate inside the shift register. The shift register is then reconfigured to transmit the parity bits by setting \( \text{cntrlM}=1 \) and \( \text{cntrlS}=0 \). After 4 additional clock cycles, at time \( t = 14 \), all the parity bits have also been transmitted. This leads to a
codeword of the format \( u_0u_1u_2u_3u_4u_5u_6u_7u_8u_9u_{10}p_0p_1p_2p_3 \) where \( u_k \) are the information bits and \( p_k \) are the parity check bits. We will refer to this string of bits as the transmitted bits, \( c_k \).

The contents of the shift register represent the state \((S_0S_1S_2S_3)\) of the encoder, which always starts and ends in the 0000 state [24]. Using the procedure in [24], we can construct a state transition table, and hence a trellis for this code. The state transition table is given by:

<table>
<thead>
<tr>
<th>Current State</th>
<th>Next State Given Input = '0'</th>
<th>Next State Given Input = '1'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 2.1: State Transition Table Based On Figure 2.1

Using the state transition table, we can determine the path through the trellis for any given codeword. As an example, the trellis path for codeword 101001011101101 is shown in Figure 2.2.
After encoding, the codeword is transmitted over an additive white Gaussian noise (AWGN) channel. At the receiver we wish to decode the codeword, minimizing the number of bit errors. This is accomplished through the use of the Forward-Backward algorithm.

2.1.2 The Forward-Backward Algorithm for Block Codes

The Forward-Backward Algorithm or BCJR algorithm was discovered by Bahl, Cocke, Jelinek, and Raviv [28] in 1974. It is a maximum a posteriori (MAP) soft output algorithm for trellis codes that computes a posteriori probabilities for each transmitted bit and selects the most probable, or maximal bit. It was adapted in [26] to operate with linear block codes, such as the previously mentioned (15,11) code.

In order to apply the Forward-Backward algorithm to a code, it must have a trellis representation. The Forward-Backward algorithm processes channel information and any a priori information to produce extrinsic information for the received bits.

2.1.3 The Forward-Backward Algorithm and the Log Likelihood Ratio

The Forward-Backward algorithm calculates the log likelihood ratio (LLR) for each bit of received data. The LLR is defined as the ratio of two conditional probabilities:
The sign of the LLR is the hard decision for each bit, $u_k$, and the magnitude of the LLR is the reliability estimate or certainty of that bit and is referred to as the soft-output estimate.

The Forward-Backward Algorithm operates on the trellis, in blocks of $N$ bits. It works by first calculating branch metrics for each state in the trellis, based on the received bits. These branch metrics are then used to recursively calculate state metrics. The final calculation sums the state metrics to determine the LLR.

2.1.4 Branch Metrics

We can write the branch metric, $y_k$, from state $s$ to $s'$ at stage $k$ in the trellis as:

$$y_k(s', s) = \exp \left[ \left( \frac{2}{\sigma^2} y_k + Apr(u_k) \right) \frac{x_k(s', s)}{2} \right]$$

(2.2)

where $x_k(s', s)$ is the expected bit between state $s$ and $s'$, $\sigma^2$ is the variance of the channel, and $Apr(u_k)$ is any a priori information in the form of bit estimates about $u_k$. For the purpose of decoding a single independent code word, $Apr(u_k) = 0$. This fact will be used in a later section as part of the turbo decoder, when a priori information is available. The branch metric is evaluated for each branch leaving a state in the trellis, which means that two branch metrics are computed per state corresponding to each $x_k(s', s) = \pm 1$. Using this equation, we can define a forward recursion and backward recursion.

2.1.5 Forward Recursion State Metrics

We wish to calculate the state metrics, $\alpha_k$, for each state of the trellis at stage $k$, based on the past state metrics of the trellis at stage $k - 1$ and branch metrics $y_k$. We can define this recursively as:

$$\alpha_k(s) = \sum_{s'} \left( a_{k-1}(s') y_k(s', s) \right), \forall k \in 1, ..., N - 1$$

(2.3)
At stage $k = 0$, we are in the starting state of the trellis. Since we know the encoder starts in the 0000 state, we are 100% certain the first state in the trellis is state $s = 0$. Hence, we set $\alpha_0(0) = 1$. From this, we know that the probability of being in any other state must be 0, and therefore we set $\alpha_0(s) = 0, \forall s \neq 0$.

$$\alpha_0(0) = 1 \quad (2.4)$$

$$\alpha_0(s) = 0, \forall s \neq 0 \quad (2.5)$$

### 2.1.6 Backward Recursion State Metrics

We now process the trellis in the opposite direction to calculate the state metrics at each stage $k - 1$, based on the state metrics of future states of the trellis at stage $k$ and branch metrics $\gamma_k$. We can define this recursively as

$$\beta_{k-1}(s') = \sum_s (\beta_k(s)\gamma_k(s, s')), \forall k \in N, ..., 1 \quad (2.6)$$

At stage $k = N$, we are in the ending state of the trellis. Since we know the encoder ends in the 0000 state, we are 100% certain the final state in the trellis is state $s = 0$. Hence, we set $\beta_N(0) = 1$. From this, we know that the probability of being in any other state must be 0, and therefore we set $\beta_N(s) = 0, \forall s \neq 0$.

$$\beta_N(0) = 1 \quad (2.7)$$

$$\beta_N(s) = 0, \forall s \neq 0 \quad (2.8)$$

Note that in order to perform the backward recursions, all bits in the block must be received.

### 2.1.7 Log Likelihood Ratio Calculation

Once the Forward and Backward recursions have been computed, we can calculate the log likelihood ratio of each bit of received data at stage $k$. This is given by:
\[ \text{LLR}(u_k) = \ln \frac{P(u_k = +1 | y_k)}{P(u_k = -1 | y_k)} \]  

\[ = \ln \frac{\sum_{(s', s) \nu u_k = +1} \alpha_{k-1}(s') \beta_k(s)}{\sum_{(s', s) \nu u_k = -1} \alpha_{k-1}(s') \beta_k(s)} \]  

The upper summation is calculated for all branches in the trellis at stage \( k \) where any “+1” transition occurs. Likewise, the lower summation is calculated for all branches in the trellis at stage \( k \) where any “-1” transition occurs.

2.1.8 The Forward-Backward Algorithm in the Log Domain

The standard Forward-Backward Algorithm is computationally complex since it involves a large number of multiplications and exponentiations. This makes it difficult to realize in hardware. However, the Forward-Backward Algorithm can be transformed into the log domain, which removes the exponentiations and replaces the multiplications and divisions by additions and subtractions. This idea of log transformation was first used in [29], where the exponentiations are replaced by the Jacobi logarithm, which states that:

\[ \ln(e^x + e^y) = \text{MAX}(x, y) + \ln(1 + e^{-|x-y|}). \tag{2.10} \]

In [30] the idea was applied specifically to the Forward-Backward algorithm for turbo decoding, which led to equation (2.11) being referred to as:

\[ \text{MAX}^*(x, y) = \text{MAX}(x, y) + \ln(1 + e^{-|x-y|}). \tag{2.11} \]

since it is essentially a maximum operator with an additional correcting factor. This can be implemented as a comparison operator with the correcting factor, \( \ln(1 + e^{-|x-y|}) \), coming from a look up table. It is only necessary to compute \(|x - y|\) and use this result as an input to the lookup table.

Using equation (2.11), we can transform the Forward-Backward Algorithm into the Log domain.

2.1.9 Branch Metrics in the Log Domain

The branch metrics become:
\begin{align*}
G_k(s', s) &= \ln(y_k(s', s)) \quad (2.12) \\
&= \left( \frac{2}{\sigma^2} y_k + Apr(u_k) \right) x_k(s', s)
\end{align*}

Notice the exponential of equation (2.2) is no longer present in (2.12). This greatly simplifies the hardware implementation.

### 2.1.10 Forward Recursion State Metrics in the Log Domain

The forward recursion becomes:

\[ A_k(s) = \ln(\alpha_k(s)) \]
\[ = \text{MAX}_{s'}(A_{k-1}(s') + G_k(s', s)) \]

For \( k = 1 \) to \( k = N - 1 \), with the initial conditions:

\[ A_0(0) = 0 \quad (2.14) \]
\[ A_0(s) = -\infty , \forall s \neq 0 \quad (2.15) \]

### 2.1.11 Backward Recursion State Metrics in the Log Domain

The backward recursion becomes:

\[ B_{k-1}(s') = \ln(\beta_{k-1}(s')) \]
\[ = \text{MAX}_{s'}(B_k(s) + G_k(s', s)) \]

with the initial conditions:

\[ B_N(0) = 0 \quad (2.17) \]
\[ B_N(s) = -\infty , \forall s \neq 0 \quad (2.18) \]

In both cases, the conversion to the log domain has removed the multiplications and replaced them with additions.
2.1.12 Log Likelihood Ratio Calculation in the Log Domain

The standard LLR has both multiplications and divisions. These are transformed into additions and subtractions in the log domain. The LLR can be restated as:

\[
L(u_k) = \text{MAX}^*_{(s', s), u_k = +1} \left( A_{k-1}(s') + B_k(s) \right) - \text{MAX}^*_{(s', s), u_k = -1} \left( A_{k-1}(s') + B_k(s) \right)
\]

This is essentially a hierarchical \( \text{MAX}^* \) over the positive transitions and negative transitions, with the results being subtracted from each other.

2.1.13 A Comparison of Forward-Backward Algorithms

It is also possible to further simplify the Forward-Backward algorithm by approximating \( \text{MAX}^*(x, y) \) as simply the maximum of \( x \) and \( y \). This yields the MAX Forward-Backward algorithm, which is less hardware intensive than the previous two versions. This gives a total of 3 different versions of the Forward-Backward algorithm.

Robertson et al [30] compared the conventional Forward-Backward algorithm, the Log Forward-Backward algorithm and the MAX Forward-Backward algorithm. They concluded that the Log Forward-Backward algorithm performed identically to the conventional Forward-Backward algorithm, and that they both performed significantly better than the MAX Forward-Backward algorithm, which was commonly used at this time.

Furthermore, they concluded that the \( \text{MAX}^* \) correction factor does not have to be very accurate to yield good performance. They demonstrated this by simulating a soft output decoder using ideal mathematics and quantizing \( \text{MAX}^* \) to 8 levels. The BER curves from this simulation were almost identical to those from the ideal simulation.

Therefore, for practical implementation with modern hardware, the Log Forward-Backward is the best choice in most circumstances.

2.1.14 A Forward-Backward Decoder

A simplified block diagram of a Forward-Backward decoder is show in Figure 2.3.
To create a product code, the following process is used:

1. Select a \((N,K)\) block code that will be used for both encodings.

2. Create a data block, \(U_k\), of size \(K \times K\).

3. Encode each row of the \(K \times K\) data block to produce a set of parity bits, of total size \(K \times (n - k)\). This set of parity bits is known as \(P_0\).

We will use this Forward-Backward decoder to construct a turbo decoder in section 2.2.

2.2 Turbo Decoding of Product Codes using the Forward-Backward Algorithm

2.2.1 Product Codes

Product codes allow two relatively simple codes to be combined into a more powerful code. The classic product code is formed by the double encoding of a data block using a block code.

In the following sections we will use capital letters to denote product code related variables, while in the previous sections we have used lower case variables that are related to a single block code. For example, we will use the notation \(Y_k\) to denote the received bits of the entire product code, while \(y_k\) represents the received bits of a single block code within the product code \((y_k \in Y_k)\).

Figure 2.3: Simplified Block Diagram of a Forward-Backward Decoder
4. Encode each column of the $K \times K$ data block to produce a set of parity bits, of total size $K \times (n - k)$. This set of parity bits is known as $P_1$.

5. Transmit the $K \times K$ data block $U_k$ and the two sets of $K \times (n - k)$ parity bits over the channel. These three sets of bits are serially concatenated and mapped to either ‘+1’ or ‘-1’ for a logical 1 or 0, respectively. They are referred to as $C_k$.

This yields a $(K^2 + 2K(N - K), K^2)$ product code of rate $\frac{K^2}{K^2 + 2K(N - K)}$. In our particular case, $N = 15$ and $K = 11$, resulting in a $(209, 121)$ product code of rate 0.58. The code is illustrated in Figure 2.4.

![Figure 2.4: Row-Column Interleaver](image)

More generally, we do not have to limit the encoding to rows and columns. We can use a variety of permutations to interleave the data bits between encodings. In this thesis, we focus on row-column interleaving, but it is also possible to use random interleaving or other deterministic interleaver functions.

The purpose of the product encoding is to increase the minimum distance and spread out bursts of errors into more easily correctable, single bit errors [31]. This is because the constituent codes can correct only a finite number of errors in each code word.
For example, if a burst of errors occurs in three consecutive data bits, they will appear as consecutive errors when decoded using the first set of parity bits $P_0$. The $(15,11)$ constituent code can only correct a single bit error, and as such, this pattern is uncorrectable. However, when decoded with the second set of parity bits $P_1$ the three consecutive errors will be spread out into three single bit errors, which can be corrected by the $(15,11)$ code.

A limitation of row-column interleaving is that it cannot correct a rectangular error pattern since two errors occur no matter which set of parity bits is used for decoding. However, by using a larger data block size of $i \cdot K \times K$ it is possible to correct these errors [26].

### 2.2.2 Turbo Decoding

The general idea of turbo decoding is to use two Forward-Backward decoders to repeatedly decode the product code, with soft information about data bits being passed between the decoders. This soft information is known as extrinsic information. The turbo decoding procedure for product codes presented here was derived in [26]. Consider the decoder shown in Figure 2.5.

![Figure 2.5: Block diagram of a Turbo Decoder](image)

At the receiver, we observe $Y_k$ which is the serial concatenation of the data bits and parity bits. Before decoding can begin, the received bits $Y_k$, must be scaled by $2/\sigma^2$, so that they may be
combined with the extrinsic information \[2\]. The multiplication of \(Y_k\) by \(2/\sigma^2\) is removed from equation (2.12) to compensate for this prescaling.

Once prescaling has been performed, decoding proceeds as follows:

1. The received bits \(Y_k\) are split into the data block, \(Y_D\), and two parity blocks, \(Y_{P0}\) and \(Y_{P1}\).
   \((Y_D = Y_{0...120}, Y_{P0} = Y_{121...164}, \text{and } Y_{P1} = Y_{165...209} \text{ due to the serial concatenation.})\)

2. The data block, \(Y_D\), and the first set of parity bits, \(Y_{P0}\), are demultiplexed so that the 11 (15,11) code words corresponding to the original horizontal encoding can be reconstructed. These 11 words are then processed by the first Forward-Backward decoder, FB\(_0\) to produce extrinsic information, \(L_0(U_k)\) for the entire received data block. During this initial run, no \textit{a priori} information is available and hence \(Apr(U_k) = 0\).

3. The data block, \(Y_D\), is then interleaved and combined with the second set of parity bits, \(Y_{P1}\), so that the 11 (15,11) code words corresponding to the original vertical encoding can be reconstructed. These 11 words are then processed by the second Forward-Backward decoder, FB\(_1\) to produce extrinsic information, \(L_1(U_k)\) for the entire received data block. However, during this decoding cycle, \textit{a priori} information is available in the form of the extrinsic information, \(L_0(U_k)\) from the previous cycle, and hence \(Apr(U_k) = L_0(U_k)\).
   This \textit{a priori} information is used by the Forward-Backward decoder as an additional estimate on the received bits, which improves the accuracy of the algorithm. Note that no \textit{a priori} or extrinsic information is available for the parity bits, since there is only a single check on each set of parity bits. As such, there is no second check, and hence it is impossible to apply iterative decoding techniques to them.

4. The extrinsic information from the previous cycle, \(L_1(U_k)\), is now used as \textit{a priori} information by FB\(_0\), and hence \(Apr(U_k) = L_1(U_k)\). \(L_1(U_k)\), the data block \(Y_D\), and parity bits \(Y_{P0}\) are now processed again by the first Forward-Backward decoder. This produces an updated set of extrinsic information which replaces \(L_0(U_k)\) and acts as \textit{a priori} information to the second decoder.

5. The algorithm can now repeat, cycling between steps 3 and 4 until stopped.

At any step, the best estimate of the received data bits is given by:
\[ L(\hat{U}_k) = \frac{2}{\sigma^2} Y_k + L_0(U_k) + L_1(U_k), \]  

(2.20)

where \( L_1(U_k) \) is the LLR from the second Forward-Backward decoder. The hard decision output, \( \hat{U}_k \), is equal to the polarity of \( L(\hat{U}_k) \).

 Turbo decoding is able to correct errors that conventional coding cannot because it makes use of two independent sets of parity bits. This gives the algorithm two independent checks to utilize when decoding. However, with each passing iteration, the extrinsic information becomes more correlated since it is based on results from the previous iterations. This means that there is a diminishing return with every iteration beyond the first iteration, and with sufficient iterations, the final result will converge and this will be the best possible result. Practically, the algorithm is usually stopped either after some preselected number of iterations, or when the hard decisions no longer change between iterations, or when the absolute values of the LLRs exceed a certain threshold [32]. Usually, only a small number of iterations are required to obtain most of the improvements in error correction.

For a detailed tutorial of the turbo decoding process, the interested reader can refer to [26], [33], [34] and [35] where the decoding process is described. Reference [33] has an excellent example of a turbo decoder going through each iteration of the process, while [35] is a textbook on turbo decoding. In 2007, [36] was published which was a special issue of the Proceedings of the IEEE dedicated to turbo and turbo-like codes.

2.3 Software Simulation

The algorithms described in the previous sections have been implemented in C using floating-point arithmetic.¹ Figure 2.6 shows an overview of the simulation process.

¹ At the time this simulation was performed, the only option was to write custom code. However, in 2005 an open source Coded Modulation Library was released that can simulate many types of iterative codes. It is developed by West Virginia University and Iterative Solutions and is regularly updated to support the latest industry standard codes. [76]
The simulation proceeds as follows:

1. Gen_data() generates \( U_k \), which is a pseudorandom data pattern of 121 bits with seed \( R_D \).

2. Prod_encode() produces \( C_k \), by encoding \( U_k \), and generating parity sets \( P_0 \) and \( P_1 \), for a total of 209 total bits.

3. Modulate() maps the logical 0’s and 1’s of \( C_k \) to “-1” and “+1”, respectively.

4. Add_noise() adds additive white Guassian noise (AWGN) to \( C_k \). This noise is generated according to the procedure in [37] with standard deviation \( \sigma \) and seed \( R_N \).

5. Prescale() multiplies the 209 noisy bits, \( Y_k \), by \( 2/\sigma^2 \).
6. Demod() reconstructs the data array $Y_D$ and two parity arrays, $Y_{P0}$ and $Y_{P1}$. It also reconstructs the horizontal and vertical (15,11) code words.

7. Prod_decode() is the actual turbo decoding engine that calculates estimates $\hat{U}_k$ based on the code bits.

8. Quantize and compare, compares the generated data to the best estimates $\hat{U}_k$ and calculates the BER.

9. This process is repeated until the total number of data packets, as specified by the user has been transmitted through the channel. On each run, $R_D$ and $R_N$ are incremented so that new data and noise are generated each time.

2.3.1 Baseline Results

1000 data packets were simulated at every SNR producing the BER curves shown in Figure 2.7.

Figure 2.7: Bit Error Rate for Floating-Point Simulation
As can be seen from the BER curves, BER improves with each additional iteration, but the majority of the improvement has happened by the fourth iteration. Therefore, the hardware design will focus on the case of four iterations, since six iterations gives little improvement in BER and consumes 50% more power.

In [26] the identical (15,11)x(15,11) product code is simulated with similar results. This verifies the decoding procedure.

2.3.2 Coding Performance and Constituent Codes

Instead of using a (15,11)x(15,11) Hamming code, it is possible to use other Hamming codes that have a higher rate. In [26], results are presented for a (31,26)x(31,26) and (63,57)x(63,57) product code. Decoding these codes yields a BER of $2 \times 10^{-3}$ and $4 \times 10^{-3}$ after 6 iterations, respectively, at an SNR of 3dB.

In [38] results are presented for a (1023,1013)x(1023,1013) product code, which was simulated on a supercomputer due to its computational complexity. However, the authors were able to approach the channel capacity by 0.27 dB at a BER of $10^{-5}$ using the algorithm of [26]. In [11], it is stated that the Chase algorithm would be approximately 0.1dB worse for this particular code.

As previously mentioned, the trellis for any of these codes will have $2^{N-K}$ states, which works out to 32 states for the (31,26) code and 64 states for the (63,57) code. The (1023,1013) code would have 1024 states. As such, these codes require more hardware to implement than smaller 16 or 8 state codes.

2.4 Comparison with Convolutional Turbo Codes

Compared with convolutional turbo codes, product turbo codes are generally higher rate, which yields higher efficiency at the cost of increased BER. Convolutional codes are able to achieve better BER performance since they have more redundant information and because there is an implicit check on the redundant bits due to the constituent codes.

In terms of implementation, the branch metric and state metric calculations are similar so most of the decoding hardware can be shared. Furthermore, techniques such as Radix-4 decoding as used in [39] can also be applied to product turbo codes.
2.5 Industry Standard Product Turbo Codes

2.5.1 IEEE 802.16 WiMAX

As was mentioned in the introduction, IEEE 802.16d WiMAX supports a code very similar to the code studied in this thesis. It is an optional part of the standard [5].

The constituent code it uses is either a (16,11), (32,26) or a (64,57) extended Hamming code. By following the procedure in [40] the H matrix can be derived, and from this the trellis can be deduced [24]. The trellis can then be processed by the Forward-Backward algorithm as has been previously described.

To form the product code, WiMAX starts with a $K_1 \times K_2$ block of data bits. The horizontal encoding happens first with an $(N_1, K_1)$ code. Once this is complete, the vertical encoding with an $(N_2, K_2)$ code is performed. Note that the parity bits from the horizontal encoding are also encoded during the vertical encoding, which allows errors in these bits to be corrected. The product code is shown in Figure 2.8.

![Figure 2.8: WiMAX Product Turbo Code](image)

We can decode this code by following a similar procedure to that of 2.2.2, with the small change that we will now have a priori information available for the first set of parity bits. This should slightly improve the BER since more errors can now be corrected.
In [7] the (64,57)x(64,57) WiMAX product code was simulated using the algorithm described in [41], which yielded a BER of $2 \times 10^{-5}$ at 3dB SNR after 32 iterations.

2.6 Summary

This chapter has outlined the theoretical ideas behind the turbo decoding of product codes.

1. Any block code with a trellis representation can be decoded by the Forward-Backward algorithm to produce a soft output estimate.

2. By combining two block codes using an interleaver, a product code can be formed.

3. This product code has greater error correcting capability than the constituent codes.

4. The product code can be decoded by iteratively applying the Forward-Backward algorithm.

5. The majority of the errors will be corrected during the first few iterations.

The practical implementation of a turbo decoding system is described in the following chapters.
Chapter 3: Architecture

This chapter discusses the digital architecture and hardware considerations of the product turbo code decoder. In section 3.1, we begin with a discussion of fixed-point mathematics and the impact this has upon the decoder. In section 3.2 we modify the C simulation to use fixed-point calculations and analyze the results. A detailed architecture for the Forward-Backward decoder is presented in 3.5, and the extensions to it necessary to implement the full turbo decoder are presented in 3.6. The memory needed to implement the decoder and its scalability to other codes is discussed in sections 3.7 and 3.8, respectively.

3.1 Fixed-Point and Quantization Effects

The initial design and simulation of the decoder was done using floating-point arithmetic. Unfortunately, the hardware needed to implement floating-point arithmetic occupies a large area in silicon and consumes a great deal of power. As such, this design relies on the use of fixed-point arithmetic which consumes far less area and power, but is potentially less accurate.

3.1.1 Fixed-Point Arithmetic

Fixed-point arithmetic is identical to two's complement arithmetic, with the additional constraint that the bits used to represent a number are divided into integer and fractional components. As long as the position of the decimal point is accurately tracked, standard arithmetic operations can be used [42].

An example fixed-point number is shown below:

$$10.625 = 1010.101$$

The bits to the right of the decimal are used to represent fractional quantities, and hence, the more bits we have, the more accurately we can represent fractional numbers. The bits to the left of the decimal are used to represent large magnitudes, which are needed so that there is sufficient dynamic range to prevent overflow. Our objective in using fixed-point arithmetic is to find the minimum number of bits needed to represent each parameter in the decoder, while still retaining enough accuracy so that the BER is not significantly degraded. Fewer bits translate into smaller circuits and less power usage. In the following sections we will examine some of the
complications of fixed-point arithmetic that must be dealt with in order to preserve as much accuracy as possible.

3.1.2 Truncation Effects

When arithmetic operations are performed on fixed-point values, it is possible that there will be too few fractional bits to accurately represent the new quantity. For example, consider the case of \( \frac{1}{4} \times \frac{1}{4} = \frac{1}{16} \). The quantity \( \frac{1}{4} \) can be exactly represented by 2 fractional bits, but \( \frac{1}{16} \) requires 4 fractional bits for exact representation. If the decoder uses 3 fractional bits, then it is necessary to approximate the 4 bit value using only 3 bits. There are several possible solutions, including conventional rounding, always rounding up (ceiling), or always rounding down (floor). In [43] this problem was studied in the context of turbo decoders and the authors concluded that the floor function gives the best BER performance. Hardware naturally implements the floor function since extra bits can simply be dropped.

3.1.3 Overflow and Saturation Effects

Since fixed-point can only represent a limited arithmetic range, it is possible to perform calculations that produce results larger than the most positive or negative representable numbers. When a value in the fixed-point system overflows, either positively or negatively, it is necessary to saturate that value to the most positive or most negative number representable. If this is not done, the two's complement number will wrap around, which will reverse its sign and change its magnitude. Therefore, after each operation, the most significant bit of the inputs and outputs must be compared to determine if a sign change has occurred. If it has, then the output must be saturated. From this point forward, all arithmetic operations are performed using saturation arithmetic.

3.1.4 Normalization

A further complication of overflow is that even with saturation it limits dynamic range. For example, if we have several sets of 100 numbers each and we wish to compare the sum of each set, we will likely have saturated all the sums by the time they are compared. This would render any comparison useless and is a known problem with fixed-point. It has been noted in [44], [45] and [43], and was first studied in the context of the Viterbi algorithm in [46]. The Forward-
Backward algorithm is also vulnerable to this problem due to the repeated summations in the forward and backward recursions, which are used to calculate the state metrics.

One possible solution is to normalize the state metrics every cycle by subtracting the largest metric from the others. It was shown in [44] that the LLR operation is independent of subtracting a constant from the magnitudes of the metrics. This preserves dynamic range without loss of accuracy.

The other solution is to use modulo normalization, which is a normalization system in which values are allowed to overflow, as long as the difference between values is accurately tracked. This can be used because the MAX* operation of the Forward-Backward algorithm is dependent on only the difference between its inputs and not on their absolute value. However, sufficient bits must be allocated such that the differences do not overflow. This requires adding one or two extra bits to the branch metrics, but eliminates the need for finding the maximum metric and subtracting it from the others [47].

In this work, we have chosen to use the subtraction normalization method because it keeps the state metrics at the fewest possible bits and saves state metric memory.

A block diagram of a Forward-Backward decoder that includes the normalization circuit is shown in Figure 3.1.

![Figure 3.1: Block Diagram of a Forward-Backward Decoder](image)
3.1.5 Analysis of the Floating-Point Simulation

The C simulation was instrumented to gather statistical information of all primary variables within the decoder. This involved finding the maximum absolute value for each parameter to determine the number of whole bits to allocate, and finding the minimum absolute value and minimum fraction for each parameter to determine the number of fractional bits to allocate.

The state metrics, $A_k(s)$ and $B_{k-1}(s')$ of (2.13) and (2.16), and the branch metrics $G_k(s', s)$ of (2.12) must be expressed in fixed-point. These metrics are used to calculate the LLRs and as such accuracy is very important. Furthermore, the state metrics are used in recursive calculations, where a small error can be multiplied and propagated many times. Due to the normalization of the state metrics, the pre and post normalization values will both have to be examined to ensure that there is sufficient dynamic range in both cases.

At the turbo decoder level, the received bits $\frac{2}{\sigma^2}Y_k$ and extrinsic / a priori information, $L_0(U_k)$ and $L_1(U_k)$ must also be expressed in fixed-point. This can be shortened to fewer bits without a significant loss in coding performance [48].

Statistical analysis was run on the turbo decoder simulation over 1, 2, 3 and 4 iterations with at least 1000 product code blocks simulated at each SNR. The results after 4 iterations are shown in Table 3.1. The “bits” column shows the number of bits needed to approximately represent the corresponding value in the “values” column. With fewer iterations there was little decrease in the values and bit widths, and hence after 4 iterations the maximum values are observed. Results for all the iterations are given in Appendix 1.

These results are critical in formulating an estimate of the number of bits that must be allocated to each parameter for fixed-point simulations.
### Quantization Statistics

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Property</th>
<th>Value</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>((2/\sigma^2)Y_k)</td>
<td>Max. Absolute Value</td>
<td>18.18538</td>
<td>5</td>
</tr>
<tr>
<td>((2/\sigma^2)Y_k)</td>
<td>Min. Absolute Value</td>
<td>7.48E-07</td>
<td>21</td>
</tr>
<tr>
<td>((2/\sigma^2)Y_k)</td>
<td>Min. Fraction</td>
<td>7.48E-07</td>
<td>21</td>
</tr>
<tr>
<td>(L_0(U_k), L_1(U_k))</td>
<td>Max. Absolute Value</td>
<td>24.95822</td>
<td>5</td>
</tr>
<tr>
<td>(L_0(U_k), L_1(U_k))</td>
<td>Min. Absolute Value</td>
<td>2.86E-07</td>
<td>22</td>
</tr>
<tr>
<td>(L_0(U_k), L_1(U_k))</td>
<td>Min. Fraction</td>
<td>2.86E-07</td>
<td>22</td>
</tr>
<tr>
<td>(G_k(s',s))</td>
<td>Max. Absolute Value</td>
<td>17.55799</td>
<td>5</td>
</tr>
<tr>
<td>(G_k(s',s))</td>
<td>Min. Absolute Value</td>
<td>3.74E-07</td>
<td>22</td>
</tr>
<tr>
<td>(G_k(s',s))</td>
<td>Min. Fraction</td>
<td>2.13E-07</td>
<td>22</td>
</tr>
<tr>
<td>(A_{k\text{pre}}(s), B_{k-1\text{pre}}(s'))</td>
<td>Max. Absolute Value</td>
<td>94.77869</td>
<td>7</td>
</tr>
<tr>
<td>(A_{k\text{pre}}(s), B_{k-1\text{pre}}(s'))</td>
<td>Min. Absolute Value</td>
<td>1.92E-08</td>
<td>26</td>
</tr>
<tr>
<td>(A_{k\text{pre}}(s), B_{k-1\text{pre}}(s'))</td>
<td>Min. Fraction</td>
<td>7.94E-11</td>
<td>34</td>
</tr>
<tr>
<td>(A_k(s), B_{k-1}(s'))</td>
<td>Max. Absolute Value</td>
<td>103.998</td>
<td>7</td>
</tr>
<tr>
<td>(A_k(s), B_{k-1}(s'))</td>
<td>Min. Absolute Value</td>
<td>6.36E-07</td>
<td>21</td>
</tr>
<tr>
<td>(A_k(s), B_{k-1}(s'))</td>
<td>Min. Fraction</td>
<td>1.55E-08</td>
<td>26</td>
</tr>
</tbody>
</table>

**Table 3.1: Analysis of Floating-Point Simulation**

#### 3.1.6 MAX* In Fixed-Point

As was stated in equation (2.11), we must calculate \(\ln(1 + e^{-|x-y|})\) as part of the MAX* function. Due to the complexity of the \(\ln\) function, it is not practical to calculate it directly with fixed-point arithmetic. Instead, the value \(|x - y|\) is calculated and is used as an entry into a look up table (LUT), to determine the value of the expression. This approach is commonly used in turbo decoders.

Alternatively, based on [49], an approximation for MAX* can be used. They show that \(f(x) = \ln(1 + e^{-|x-y|})\) can be approximated by:

\[
f(x) = \begin{cases} 
\frac{3}{8} & \text{for } -2 \leq x \leq 2 \\
0 & \text{otherwise.} 
\end{cases}
\]  
\[
(3.1)
\]
By using this approximation, the lookup table can be eliminated, which results in significant savings at the hardware level. Coding performance is barely affected with only a 0.03 dB loss at high bit error rates and virtually no loss otherwise [49]. This is in complete agreement with the previously mentioned results by [30] which stated that MAX* could be approximated with no significant loss in performance.

3.2 A Software Fixed-Point Turbo Decoder

To determine the optimal bit widths, the C simulation was modified to account for fixed-point math, by using a custom fixed-point library.\(^2\) Only 3 functions in the C program needed significant changes. Prescale() was changed so that it prescaled and quantized the channel values using fixed-point arithmetic. Prod_decode() was heavily modified to use the fixed-point library and the approximate MAX* function, which was the bulk of the re-coding effort. The quantization was removed from the final comparison stage of the program since it was no longer needed.

This new simulation flow is shown in Figure 3.2.

\(^2\) For a modern design, SystemC could be used instead of ISO C to run fixed point simulation. This would have the added benefit that the SystemC can also be synthesized directly.
These changes allowed a variable number of bits to be allocated to each parameter, so that experiments could be run with different bit widths for the various decoder parameters. The decoder parameters were merged into two categories: Forward-Backward parameters, known as FB parameters, which included $G_k(s',s)$, $A_{k_{(pre)}}(s)$, $B_{k-1_{(pre)}}(s')$, $A_k(s)$ and $B_{k-1}(s')$ and turbo parameters, which included $\frac{2}{\sigma^2} y_k$, $L_0(U_k)$ and $L_1(U_k)$. For each category, whole bits and fractional bits were considered.

For each SNR, over 1000 blocks were decoded using 4 iterations (for a total of 209k bits decoded) and a summary of the results is presented in Table 3.2. The loss relative to the floating point simulation at a BER of $10^{-2}$ was calculated by interpolation from the BER curves in Figure
2.7. This simulation was run multiple times by sweeping the SNR in 0.5 dB steps from 0 dB to 3 dB.

The choice of 4 whole turbo bits, 2 fractional turbo bits, 5 whole Forward-Backward bits and 3 fractional Forward-Backward bits was selected as being the best overall, since it represents a good compromise between accuracy and bit width. Other cases where 1 or 2 additional bits were used improved accuracy by only 0.011 dB. The 2 cases listed represent the biggest improvement in accuracy, with other uses of 1 or 2 more bits having less effect. Similarly, the cases where 1 or 2 bits are removed decrease accuracy. The 2 cases listed resulted in the lowest overall decrease in BER when bits were removed. Also, the case of using 8 bits for each parameter was run since this represents a reasonable upper bound on the accuracy of the fixed-point simulation. In practice, the hardware to implement more than 8 bits per parameter would be costly.

<table>
<thead>
<tr>
<th>Turbo Whole Bits</th>
<th>Turbo Fractional Bits</th>
<th>FB Whole Bits</th>
<th>FB Fractional Bits</th>
<th>Loss rel. to Floating-Point @ BER=10^{-2} (dB)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>0.014</td>
<td>16 bits for each value does not improve BER dramatically.</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>0.018</td>
<td>Best way to use 2 more bits than used in this thesis.</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>0.021</td>
<td>Best way to use 1 more bit than used in this thesis.</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>0.029</td>
<td>This thesis</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>0.044</td>
<td>Best way to save 1 more bit than used in this thesis.</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>0.068</td>
<td>Best way to save 2 more bits than used in this thesis.</td>
</tr>
</tbody>
</table>

Table 3.2: Fixed-Point Simulation Comparison After 4 Iterations

From the discussion in the previous section, we can now redraw the block diagram of the turbo decoder to include the bit widths of the various busses. This becomes the basis for the fixed-point decoder. In Figure 3.3, a Prescaler is shown that is used to calculate \( \frac{2}{\sigma^2} Y_k \). In hardware this prescaling will be done off chip, and we will assume that \( Y_k \) has been scaled ideally based on the SNR. However, for production chips the prescaler is usually integrated and it must make certain assumptions about the SNR and prescale by the appropriate amount. (In some designs a fixed prescaler for a certain range of SNRs is used, while in others a variable prescaler that is based on the SNR of the received data is used.)
3.3 Fixed-Point Simulation Results

Based on the selected bit widths, a comparison between floating-point and fixed-point was run for 1, 2, 4 and 6 iterations. The results from the simulation are shown in Figure 3.4. After 4 iterations, at a BER of $10^{-3}$ there is a loss of 0.04 dB.
With this information, the objective of the remainder of the thesis is to show how to implement a hardware turbo decoder that performs identically to the fixed-point simulation.

### 3.4 Forward-Backward Decoder Architecture

Our objective is to define a Forward-Backward decoder architecture that minimizes area, while having the maximum possible throughput and lowest possible power consumption. Before this can be explored in detail, we must first discuss the fundamental building block of the Forward-Backward decoder: the Add-Compare-Select (ACS) Unit.

#### 3.4.1 Add-Compare-Select (ACS) Unit

The Add-Compare-Select unit, or ACS, is the computational kernel of the Forward-Backward algorithm. It calculates the state metrics for the alpha and beta recursions as in (2.13) and (2.16). In hardware, a circuit to implement the ACS is shown in Figure 3.5.
The circuit starts by summing the state metrics from the previous state and the branch metrics, to form intermediate values $X$ and $Y$. The values $X$ and $Y$ are then used in a $\text{MAX}^*$ operation, which works by calculating $X - Y$ and using the result to determine the correction factor. If the difference is less than positive 2, the upper 4 bits will all be one. If the difference is less than or equal to negative 2, the upper 4 bits will all be one. From this, the multiplexer M1 can select either zero or $\frac{3}{8}$ depending on whether the difference is within $-2 \leq x \leq 2$. Multiplexer M2 selects the maximum of the sums and passes this value through to the final adder, where the correction factor is added. This $\text{MAX}^*$ circuit is taken from [49].

Notice that the circuit is purely combinational, which means no memory or control logic is needed for a lookup table to implement $\text{MAX}^*$. According to [49], this represents an area savings of approximately 40% relative to a conventional implementation using a lookup table. Since $\text{MAX}^*$ can be used repeatedly throughout the Forward-Backward decoder the total hardware savings are significant.

### 3.4.2 Trellis Processing

A single ACS circuit calculates a single state metric for either the forward or backward recursions. In this work, we use $2^{N-K} = 16$ ACS units in parallel, which allows for an entire stage of the trellis to be calculated in one cycle. A memory is used to store the results of the current stage of the trellis, so that these results can be used as an input for the next stage. This is shown in Figure 3.6.
3.4.3 Trellis Processing Schedule

Consider the operation of the 16 ACS units as they process the trellis. The challenge is to find the optimal order in which to calculate the alpha and beta recursions, such that the memory and computational requirements are minimized, and that memory contention is avoided.

3.4.4 Conventional Trellis Processing Schedule

In the conventional algorithm, the forward recursion is performed first and the backward recursion second. Since we can calculate all the metrics of one time interval in parallel, it takes 30 clock cycles to decode the block, because there are 15 trellis stages in both the forward and backward recursions. (LLR calculations happen concurrently with the backward recursion and are therefore included within the 30 clock cycles.)
However, this necessitates that the results of the entire forward recursion are stored in memory. Therefore, we require

\[8 \text{ bits/metric} \times 16 \text{ metrics/time interval} \times 15 \text{ time intervals} = 1920 \text{ bits of memory.}\]

### 3.4.5 Modified Trellis Processing Schedule

![Figure 3.8: Modified Schedule](image)

For product turbo codes we are only interested in decoding the data bits, since any estimate of parity bits is discarded on the next iteration. This is because it is not possible to pass extrinsic information about the parity bits between Forward-Backward decoders. This gives rise to a new schedule where the backward recursion is performed first, and only the state metrics for states \( k = 0 \) to \( k = 10 \) are stored in memory. The forward recursion is performed second, but it need only calculate metrics for the first 11 stages since any stage after that corresponds to a parity bit. This reduces the decode time to \( 15 + 11 = 26 \) clock cycles, and reduces the memory requirement to \( 8 \text{ bits/metric} \times 16 \text{ metrics/time interval} \times 11 \text{ time intervals} = 1408 \text{ bits.} \) Furthermore, by not fully decoding the parity bits, significant power is conserved.

### 3.5 Forward-Backward Decoder Implementation

Based on the results of the previous sections, we can define hardware to implement a Forward-Backward decoder. This is shown in Figure 3.9.
At every stage through the trellis, the branch metric unit and ACS units calculate the state metrics. These metrics must be normalized to prevent overflow. This is done by a normalization circuit, known as Maxtree. The normalized state metrics are then stored by the registers into memory. The LLR unit is then used to calculate soft bit estimates based on the normalized metrics.

Each component will now be explained in detail.

3.5.1 Branch Metric Calculator

The branch metric calculator implements equation (2.12) and processes received channel values so that they can be used by the ACS units. The outputs of the branch metric calculator are registered before they are connected to the ACS.

3.5.2 Modified ACS

The ACS unit of Figure 3.5 must be modified such that it can be process both alpha and beta recursions. The modified ACS unit is shown in Figure 3.10.
3.5.3 Initial Conditions

Recall from section 2.1 that the initial conditions on the alpha and beta recursions are:

\[ A_0(0) = 0 \]
\[ A_0(s) = -\infty, \forall s \neq 0 \]
\[ B_N(0) = 0 \]
\[ B_N(s) = -\infty, \forall s \neq 0 \]

These conditions are implemented by two multiplexors at the input of the ACS circuit. At the start of a forward or backward recursion, the select signal to these multiplexors is asserted to force the initial conditions into the ACS core. The multiplexor on the first ACS unit (ACS\(_0\)) is hard coded to output 00000.000 when select is asserted, while the multiplexors on the other ACS (ACS\(_{1\ldots15}\)) output 100000.000. These represent 0 and the most negative possible number in fixed-point, respectively.

3.5.4 Processing the Forward and Backward Recursions

In order to use the same hardware for both forward and backward recursions, we must connect the ACS units in a flexible manner. The interconnect between ACS units is driven by the trellis.
and state transition table. From Table 2.1, we know how to interconnect the ACS units for both forward and backward recursion. However, we must add an additional set of multiplexors at the ACS inputs such that we may switch between forward and backward recursions.

As an extension of this concept, if instead of connecting the multiplexors based on the state transition table, we connected them through a software configurable means (such as via a memory), then this design could support codes with a time variant trellis.

### 3.5.5 Maxtree Normalization Circuit

There are 16 modified ACS units which compute 16 state metrics per cycle, each of which has 8 bits of precision. As was previously discussed, we must normalize these metrics every cycle by subtracting the largest metric from all the others to prevent overflow.

This is done using a normalization circuit, known as Maxtree, which finds the biggest state metric and subtracts it from all other state metrics.

![Maxtree Normalization Circuit](image)

**Figure 3.11: Maxtree Normalization Circuit**

Sixteen metrics are taken as inputs, and the tree of comparators processes each value until the largest one is found. Each comparator uses an 8 bit adder and checks the sign bit of the result to determine which input was larger. Once the largest metric has been found, it is subtracted from all other metrics using the column of 16 adders near the output. This adds 5 full adder delays to the metric calculation time. Unfortunately, there is no efficient way around this, since modulo
normalization would increase the state metrics to approximately 10 bits, which would consume approximately 20% to 30% more area for logic and 25% more area for memory.

3.5.6 SRAM and State Metric Register

Memory, in the form of static random access memory (SRAM) is needed to store the state metrics during each cycle of the decode. $2^{n-k} = 16$ state metrics, at a width of 8 bits, for a total of 128 bits, must be stored each cycle. This requires a 128 bit wide memory to store all the results in one memory clock cycle. The state metrics are registered before they are sent to the memory.

3.5.7 Log Likelihood Ratio (LLR) Calculator

The log likelihood ratio unit is intended to calculate the log likelihood ratio of (2.19), which is restated here for convenience.

$$L(u_k) = MAX_{(s', s), u_k = +1}^*(A_{k-1}(s') + B_k(s))$$
$$- MAX_{(s', s), u_k = -1}^*(A_{k-1}(s') + B_k(s))$$

For an N state trellis, the LLR requires $2N$ adders in the first stage, followed by $J = 2^{N-K}$ stages of MAX* units arranged in a tree structure. The LLR unit can be divided into two logical halves: the upper half is intended to calculate the likelihood of a “+1”, while the lower half calculates the likelihood of a “-1”. In the final stage, the likelihood of a “+1” and a “-1” are subtracted, yielding the log likelihood ratio.

For this particular code, we have a 16 state trellis, which results in an LLR unit with 32 adders in the first stage and 4 stages of MAX* units. A simplified version of an LLR unit with only 2 stages is shown in Figure 3.12.
3.5.8 Control Unit (CU)

A finite state machine is used to control the Forward-Backward decoder. It is responsible for state metric memory control signals, setting initial conditions and reading estimates of decoded bits out of memory for computing the LLR.

3.5.9 Critical Path

The critical path through the decoder is contained within the ACS feedback loop. It is highlighted in Figure 3.13.

---

**Figure 3.12: Log Likelihood Ratio Calculation Circuit**

![Log Likelihood Ratio Calculation Circuit](image)
The critical path is the slowest path from the Current State Metric Register to the Next State Metric Register. It can go through any of the ACS units since they all receive the branch metrics and inputs from the Current State Metric Register at the same time. This takes approximately 3 full adder delays. The newly computed state metrics from the modified ACS are then normalized by the Maxtree and latched into the Next State Metric Register. This takes approximately 5 full adder delays in addition to the setup and hold time of the register.

In total, the delay is approximately 8 full adders delays, plus some additional time for multiplexors and nand/nor gates in the ACS unit. At 3 full adder delays, the state metric calculations are relatively quick, but a large amount of time is spent normalizing the state metrics since it takes 5 full adder delays.

### 3.6 Turbo Decoder Architecture

With the architecture for the Forward-Backward decoder defined, we can now use it to construct the full turbo decoder. The turbo decoder is composed of one Forward-Backward decoder, two memories (one for the channel memory, which stores the received data block and the other for the extrinsic / a priori information), and a circuit to generate addresses for the interleaver / inverse interleaver.
3.6.1 Operation of the Turbo Decoder

The turbo decoder operates as described in section 2.2.2, with the exception that there is now only a single Forward-Backward decoder. This decoder is reused for both the horizontal and vertical decoding cycle, with the extrinsic information being stored in memory.

This results in an area efficient design that requires 26 cycles/FB decode * 22 decodes = 572 cycles per iteration.

3.6.2 Memory for Channel and Extrinsic / A Priori Information

Two memories are used in the turbo decoder, one for channel memory and one for extrinsic / a priori information. The 6 bit words that represent the received values are stored in the channel memory. There are 209 values * 6 bits/value = 1254 bits, which means that the ideal memory would be 6 bits wide with 209 addresses. The extrinsic / a priori information that flows between decoders is also 6 bits wide, but we need only store this for the first 121 values. This is because
the first 121 values represent information bits, while the latter 88 values represent parity bits. Therefore, this memory should be 6 bits wide * 121 addresses = 726 bits.

3.6.3 Address Generator

The address generator creates a series of addresses that are used to load and store information from the memories into the Forward-Backward decoder. It contains a memory cell to store the interleaver permutations. This could have been implemented as combinational logic for the row-column interleaver, but the hardware was designed to support any arbitrary interleaver.

![Address Generator Circuit](image)

**Figure 3.15: Address Generator Circuit**

Two counters are used to track the address of information and parity bits. When set to interleaved mode, the value of the data counter is used as an address to the interleaver memory. The memory then returns the position of the interleaved bits as its output. By multiplexing these counters and memory outputs together, it is possible to generate the sequence of addresses.

To write back to memory, a buffer of the last K addresses is kept. This allows any interleaver function to be used, since there is no need for a symmetric interleaver. For a row-column interleaver, this could also be done with combinational logic.

3.6.4 Calculation of Best Estimate

The best estimate of each bit is calculated according to equation (2.20), which is essentially an addition of 3 values. This is implemented as two full adders.

3.6.5 Control Unit

The control unit coordinates the generation of addresses with the other memory control signals and the operation of the Forward-Backward decoder. It is implemented as two state machines.
3.7 Memory Requirements

The total memory requirements of the (15,11) decoder are:

<table>
<thead>
<tr>
<th>Component</th>
<th>Words</th>
<th>Width</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward-Backward Decoder Memory FB0</td>
<td>11</td>
<td>128</td>
<td>1408</td>
</tr>
<tr>
<td>Channel Memory CH0</td>
<td>209</td>
<td>6</td>
<td>1254</td>
</tr>
<tr>
<td>Extrinsic / A Priori Memory EA0</td>
<td>121</td>
<td>6</td>
<td>726</td>
</tr>
<tr>
<td>Interleaver Memory IN0</td>
<td>121</td>
<td>8</td>
<td>968</td>
</tr>
<tr>
<td><strong>Total Memory</strong></td>
<td></td>
<td></td>
<td><strong>4356</strong></td>
</tr>
</tbody>
</table>

Table 3.3: Memory Requirements

3.8 Scalability

This architecture can be extended to decode (31,26), (63,57) and higher rate product codes. It can also be extended to support the WiMAX codes. The case of conventional product codes will be considered first.

For the Forward-Backward decoder, each trellis has $2^{N-K}$ states which will require $2^{N-K}$ ACS units with $N - K + 1$ layers of adders in the normalization circuit. This increases the area of the Forward-Backward decoder and decreases its speed. Memory requirements are also increased as $N - K$ grows, needing a memory that is 8 bits $\times 2^{N-K}$ wide and $K$ words deep.

At the turbo decoder level, the channel and extrinsic / a priori memories must be larger to store the greater number of channel bits. The channel memory must now store $K^2 + 2K(N-K)$ received bits $\times$ 6 bits/received bit, while the extrinsic / a priori must store $K^2$ received bits $\times$ 6 bits/received bit. For practical designs, the interleaver memory would be implemented as combinational logic, which would not require memory, and hence we will ignore the scalability of the interleaver memory.

In the particular case of WiMAX codes, most parameters will scale as above, except that now the Forward-Backward decoder must store $N$ words, while the channel memory must store $N \times N$ received bits.
The following table shows the number of ACS units and size of memory needed to decode various product codes.

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>Type</th>
<th>ACS Units</th>
<th>Depth of Maxtree</th>
<th>Total Memory Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>4</td>
<td>Conventional</td>
<td>8</td>
<td>4</td>
<td>592</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>Conventional</td>
<td>16</td>
<td>5</td>
<td>3388</td>
</tr>
<tr>
<td>31</td>
<td>26</td>
<td>Conventional</td>
<td>32</td>
<td>6</td>
<td>16328</td>
</tr>
<tr>
<td>63</td>
<td>57</td>
<td>Conventional</td>
<td>64</td>
<td>7</td>
<td>72276</td>
</tr>
<tr>
<td>1023</td>
<td>1013</td>
<td>Conventional</td>
<td>1024</td>
<td>11</td>
<td>20734084</td>
</tr>
<tr>
<td>16</td>
<td>11</td>
<td>WiMAX</td>
<td>32</td>
<td>6</td>
<td>5408</td>
</tr>
<tr>
<td>32</td>
<td>26</td>
<td>WiMAX</td>
<td>64</td>
<td>7</td>
<td>24448</td>
</tr>
<tr>
<td>64</td>
<td>57</td>
<td>WiMAX</td>
<td>128</td>
<td>8</td>
<td>104832</td>
</tr>
</tbody>
</table>

Table 3.4: Scalability to Other Product Codes
(The (15,11) code used in this thesis is highlighted.)

3.9 Summary

This chapter has described the architecture and operations of the turbo decoder.

1. The optimal bit widths were selected for the fixed-point representation of the internal quantities of the decoder.

2. Normalization of state metrics is necessary to prevent overflow. This can be accomplished by periodically subtracting the largest metric.

3. The Add-Compare-Select unit forms the core of the decoder.

4. The ACS and Maxtree normalization circuit are on the critical path of the decoder.

5. The turbo decoder uses several memories to store extrinsic information between iterations.

In the following chapters, the synthesis and physical design of the decoder will be discussed.
Chapter 4: ASIC Design

This chapter will discuss the hardware implementation of the turbo decoder. In section 4.1 the available technology options will be discussed, which will form the basis of the design flow described in section 4.2. In sections 4.3 to 4.11 the various stages of the design flow will be examined in detail. The simulated results of the flow will be discussed in 4.12 and the test results from the fabricated ASIC will be presented in section 4.14. Alternative technologies and technology scaling will be examined in section 4.15.

4.1 Implementation Technology

For any digital IC design, a fabrication technology must be selected. For this thesis, the best available process with an SRAM option was a 0.18µm CMOS process that had one poly layer and six metal layers (1P6M). This makes it possible to integrate logic and memory into a single IC.

The decoder is based on third party standard cells, which include a logic cell library and a memory cell library. As such, specific information about the cells and foundry process is protected under non-disclosure agreement. This limits the level of detail that specific aspects of the decoder can be discussed in this thesis. However, as a reference point the area of a minimum inverter is approximately 9 µm² with a FO4 delay of approximately 80ps. The library has an ideal placement density of approximately 82k minimum strength 2-input NAND gates per mm².

4.2 EDA Methodology

ASICs and SoCs are designed by performing a series of operations using various Electronic Design Automation (EDA) tools. These steps form the basis of an EDA tool flow. The optimization settings and design constraints used in the flow can have an enormous impact on the final results. Therefore, significant planning and research was done to determine the most appropriate optimizations and constraints to apply to the design. The design flow is illustrated in Figure 4.1 and was based on a standard digital design flow [50], with several modifications due to the integrated memories.

---

The FO4 (fan-out four) delay is the average propagation delay through an inverter that is driving four identical inverters.
Figure 4.1: Digital Design Flow for 0.18\textmu m CMOS
4.3 HDL Design and Verification

The first stage of the design cycle was to implement a software model of the decoder using a hardware description language (HDL). Verilog was selected over VHDL as the standard HDL since it is better supported by the EDA tools. The code was written in a hierarchical manner, which allows for flexibility and reuse of the components. It was simulated using the NC-Verilog and Verilog-XL simulators.

The memory cells selected were dual port 256x32 SRAM cells. These were chosen because they were the smallest available cells in the cell library at the time. The state metric memory was composed of 4 memories in parallel to give a 128 bit data bus. Three other memories were used to implement the channel, interleaver and extrinsic / a priori memories.

The fixed-point C simulation was used to generate test vectors that were fed into the Verilog model. The output of both simulations should be identical, and after several design iterations, the outputs of both simulators agreed in all cases. To verify the results, over 2M bits were simulated in both C and Verilog (10000 product code packets at 209 bits each, for a total of 2,090,000 total bits). No discrepancies were found between the simulations.

4.4 ASIC Design Objectives

4.4.1 Maximizing Operating Frequency

In order to have the highest throughput, we must maximize the frequency at which we can clock the design. This is done by making the slowest path in the design, usually referred to as the critical path, as fast as possible. Once the critical path is identified, the EDA tools will optimize it as their first priority. This is usually done by a combination of logic optimization, upsizing gates and by placing the cells of the critical path close to each other, so that routing is minimal.

4.4.2 Minimizing Area

Since the cost of an ASIC is directly proportional to the area it consumes, we wish to minimize area. The EDA tools were set to first optimize for speed, and then to optimize for area. Most area savings will come from non-critical paths since we can afford extra delay on these paths.
4.4.3 Minimizing Power Consumption

Turbo decoder ASICs are often used in mobile applications, such as radios and cellular telephones. Power consumption is therefore important, and should be minimized once performance and area constraints are met. Power consumption is made up of two components: dynamic and static.

Dynamic power is the power consumed whenever the ASIC switches states. This includes the power of the logic cells, memory cells and IO cells. The total power consumption of the cells is given by:

$$P \propto fCV^2$$  \hspace{1cm} (5.1)

where $f$ is the switching frequency, $C$ is the total capacitance of gates switching and $V$ is the power supply voltage. The higher the frequency, the higher the number of gates or the higher the voltage, the more power is consumed. To reduce dynamic power consumption, we must reduce $f$, $C$ and most importantly $V^2$. There is little that can be done about frequency since we wish to maximize performance, and the supply voltage $V$ is 1.8V for this semiconductor process and cell library. However, the EDA software can be configured to attempt to minimize the number of gates that switch for each logic function since this reduces the effective capacitance.

Static power is consumed whenever the device is powered up. It is primarily due to subthreshold conduction and reverse bias PN junction leakage. It is important to minimize static power consumption since turbo decoder ASICs are often used in mobile applications where they may wait in the stand-by state for long periods of time. In 0.18$\mu$m CMOS transistors do not suffer a significant amount of subthreshold conduction or junction leakage, which means that static power consumption is not a significant problem. However, with more advanced technologies, both of these will become more significant. This will be discussed in a later section.

4.5 Synthesis

The objective of synthesis is to take the Verilog model and map it into physical hardware. In a cell-based design, such as this turbo decoder, a library of standard cells is used as the fundamental building blocks of the hardware. Each individual cell implements a simple function, such as NAND or NOR, but they can be connected together to build complex logical functions.
The optimization goals of synthesis are threefold: maximizing operating frequency, minimizing area and minimizing power consumption.

For this design, maximum clock frequency was the primary synthesis goal, with area and power being secondary goals. The synthesis tool, Synopsys Design Compiler, was used to compile the design and to correct any timing violations, such as setup time and hold time violations. For setup time calculations and timing estimation the worst case process, voltage and temperature (PVT) models were used, while for hold time calculations the best case PVT models were used.

Since this design was under 100,000 gates, a top down, flat style synthesis was possible. There was no need for a hierarchical compile of the individual blocks of the decoder. Hierarchical synthesis strategies allow for multi-million gate designs and several strategies and guidelines for hierarchical synthesis are discussed in [51] and [52].

Initial synthesis of the Verilog yielded a design that performed correctly, but that had a minimum clock period of over 30 ns. This large delay was due to the LLR, which had a delay greater than that of the ACS critical path. Therefore, the LLR was pipelined into 3 stages, which reduced its delay below that of the ACS critical path. The modified LLR is shown in Figure 4.2.

Pipelining the LLR required changing the original Verilog code which was resimulated to verify correctness. After synthesis was complete the design had a critical path of 13.15 ns (as determined by static timing analysis), and a total of 18341 cells. Extensive simulations (over 2M bits) were run to ensure the synthesized netlist gave the same results as the C simulation.
The results from synthesis are summarized in Table 4.1.

<table>
<thead>
<tr>
<th></th>
<th>Original LLR</th>
<th>3 Stage Pipelined LLR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers of Cells</td>
<td>18341</td>
<td></td>
</tr>
<tr>
<td>Area of Cells</td>
<td>1.52 mm²</td>
<td></td>
</tr>
<tr>
<td>Critical Path Delay</td>
<td>13.15 ns</td>
<td></td>
</tr>
<tr>
<td>Frequency</td>
<td>76.04 MHz</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Summary of Synthesis Results
4.6 Design for Test Features (DFT)

A major issue in semiconductor design and manufacturing is how to test an IC after it is fabricated to ensure it has been constructed correctly. During fabrication it is possible that certain structures and devices are not formed correctly, which can lead to various problems with the circuits, such as internal nodes being stuck at a logical zero or one. To check for this, there are various design-for-test features that can be added. The goal is to achieve the maximum possible fault coverage, which is the percentage of manufacturing faults that DFT can detect. The target fault coverage was set to 100% during synthesis so that Design Compiler uses DFT features wherever possible. It is not always possible to get 100% coverage, but 90%+ should be obtainable. For typical commercial ICs, fault coverage is usually greater than 98%.

4.6.1 Scan Chain Registers

One commonly used DFT feature is the scan chain. This works by adding a scan in and scan out port to every register in the netlist. Synopsys Test Compiler can then connect these scan ports to form a scan chain. An example scan chain is shown in Figure 4.3

![Scan Chain Registers for Testing Combinational Logic](image)

Figure 4.3: Scan Chain Registers for Testing Combinational Logic

Notice in Figure 4.3 that the combinational logic is surrounded by scan registers. This allows a test vector to be shifted into the scan chain so that it acts as an input to the combinational logic. The shifting is done by driving the test data onto the scan in (SI) port while clocking the registers
using the scan clock (sclk). Once all the registers in the scan chain are loaded, the system clock is
cycled once and the outputs of the combinational logic are latched into the scan registers. The
outputs can then be shifted out to the scan out (SO) port where they are read by a tester.

This is a very effective way to test combinational logic and registers in a design. The
disadvantage to this is that special scan registers must be used instead of normal registers. Scan
registers are larger, consume more power and have a greater delay due to the scan circuitry. In
the original synthesis run, Design Compiler was instructed to only use scan registers. This meant
that when the scan chain was added, the timing did not degrade from its original value. However,
the critical path would have been shortened by 1 ns to 2 ns if no scan chain registers had been
used. (from 16.7 ns to between 15.7 ns and 14.7 ns) The final design utilized a single scan chain,
composed of approximately 1150 registers and required 3 extra IO pins for scan signals.

In addition to the scan ports, the registers also have a reset port, which is used to force the IC into
a known state. This is useful when powering up and when recovering from any unknown states.
The distribution of the reset signal is handled via the physical design tools, and for a design of
this size no special care is required on the reset net. For larger chips, a tree structure can be used
to buffer the reset signal.

4.6.2 Black Box Cells

Any black box cells, such as SRAMs or analog blocks, are not testable using the scan chain
technique. This is because the Test Compiler does not understand the function of these blocks
and therefore is not able to generate appropriate test vectors.

In the case of the turbo decoder, there are 7 black box memory cells that need to be tested for
manufacturing faults. The most straightforward way to do this is to make the memory control
signals externally accessible. This is done by adding IO pins for the memory clock, memory
address and memory data and multiplexing them with the memory signals coming from the
decoder core. The chip can then be set to a memory test mode that connects memory IO signals
directly to the IO pins. It is then possible to use an external tester to run arbitrary memory test
patterns. This arrangement is depicted in Figure 4.4.
Since there are 7 memories, their external outputs are multiplexed together to reduce pin count, with a 3 bit select signal to choose which memory to connect to the IO pins when in test mode.

An additional benefit of the memory test circuitry is that the memories can be accessed when the chip is operating. This is done by holding the core clock low and switching the chip into test mode. The contents of the memories can then be inspected to aid in debugging.

An alternative strategy is to use memory built in self test (MEMBIST) which is discussed in [53].

4.6.3 Shadow Logic

In a design that uses both standard cells and black box cells, it is possible to have standard cells that are not testable. If the inputs to, or outputs from the standard cells are not connected to a register, then they are not observable by the scan chain, and hence not testable. This hidden combinational logic is known as shadow logic [54]. Consider the circuits in Figure 4.5.1
Figure 4.5: Untestable Shadow Logic

The combinational logic in both of these circuits is not testable by the scan chain. In the first case, there are no scan registers at the input to the logic, which means there is no way to apply test vectors to the logic. In the second case, there are no scan registers connected to the outputs of the logic, which means the outputs are not observable.

There are several possible solutions to this problem.

4.6.4 Registering the Inputs and Outputs

The most obvious solution is to register the inputs and outputs to all SRAMs, thereby eliminating the shadow logic. The problem here is that the extra registers add latency and this requires retuning of the state machine controllers. Code must be rewritten and resimulated. Furthermore, latency is slightly increased due to the extra registers on the data path.

4.6.5 Multiplexing the Inputs and Outputs

An alternate solution is to implement the circuit shown in Figure 4.6.
If the multiplexors are set to the test position, the scan registers are connected to the combinational logic which allows it to be tested. In normal operation, no additional registers delays are added to the circuit, which means control logic does not need to be changed. However, the delay through the circuit does increase due to the multiplexers.

### 4.6.6 Synopsys Shadow Logic DFT

The Synopsys Test Compiler has a shadow logic test feature that will add the circuit of Figure 4.6 automatically to the design. However, there were problems encountered when using this feature since the circuits were not always added correctly. To solve this problem, the multiplexers and scan registers were manually added to the Verilog code.

### 4.6.7 Automatic Test Pattern Generation (ATPG) for Logic Cells

Once the scan chain was implemented, Synopsys TetraMAX was used to generate the scan chain test vectors. Using this tool on our design, we were able to achieve 95% fault coverage, meaning that 95% of any possible faults in the combinational and sequential logic were detectable by the scan chain.

### 4.6.8 Memory Testing

Test patterns for the memories are handled manually, and since all relevant memory IO signals are available externally, it is possible to get 100% fault coverage. The subject of memory testing is extensive [53] and there are many possible test strategies, but a simple test strategy involves walking a 1-0-1 pattern through the memory. This test does not cover all possible faults, but it
still gives good fault coverage and a relatively quick run time. More extensive tests can cover 100% of memory faults, but they have large run times.

4.7 Physical Design

Physical design takes the gate level netlist from the synthesizer and builds circuits by placing standard cells and routing wires between them. During this process, there are several challenges to be overcome. The most obvious of these is to implement the required circuit correctly. Furthermore, we wish to maximize the operating frequency and minimize the power consumption and area. This necessitates that we add as little parasitic resistance and capacitance as possible. We must also ensure that cells receive adequate power and that wires exhibit minimal crosstalk.

4.7.1 Power and Ground Distribution

To supply an adequate power and ground net, a robust power distribution grid is needed that is capable of supplying high peak currents while having minimal resistance and inductance. If the grid does not meet these criteria, various problems, such as supply drop and ground bounce can occur. The grid encompasses the IO pads, the power rings, power stripes and the routing to each individual cell. These structures will be covered in greater detail in later sections.

4.7.2 Signal Integrity

To connect the various logic cells, many individual wires are needed. Wires are non-ideal, since they have an innate resistance, innate inductance and capacitance to the power and ground nets and to the substrate. Furthermore, the signals carried by these wires can interfere with each other due to mutual capacitance and inductance. Long wires are exceedingly vulnerable to this phenomenon and care must be taken to avoid using long wires whenever possible. The traditional fix of increasing the width of a wire to alleviate resistance problems is no longer effective since wider wires capacitively couple more strongly than narrower wires. Signal integrity issues were dealt with by floorplanning to avoid long parallel wires and by resizing buffers as needed. This is more fully described in the following sections.
4.8 Floorplanning

The goal of floorplanning is to create the overall structure for the chip. We start with an empty piece of silicon and we allocate space for logic cells, memory cells, IO cells and power and ground buses. The primary objective in floorplanning is to minimize the distance between blocks, such that the length of wires is minimized. The secondary objective is to minimize the total area of the ASIC. Floorplans can be either created manually, or automatically using an EDA tool. In this case, the floorplans were created manually since the automatically generated floorplans were unrouteable.

The floorplan that was used is shown in Figure 4.7:

![Floorplan Diagram](image)

**Figure 4.7: Floorplan**

Starting at the outermost edge, we have the IO ring, the power distribution rings, and then finally the core.
The IO ring is composed of unidirectional signal pads, core power and ground pads, and IO power and ground pads. Separate core and ring supply pads are used because the ring runs at a higher voltage than the core. Furthermore, the IO circuits can generate switching noise which can couple into the core. The converse is also possible. Where possible, power and ground pads are placed in pairs to maximize capacitance from the bond wires.

Multiple power pads are uniformly distributed around the IO ring, so that a continual low resistance and low inductance path is available from the external power connections to the power distribution rings. The power distribution rings are rings of thick metal (metal five and six) that completely encircle the core, such that the core power connections also have a low resistance path to the supply. The rings are spaced closely together to maximize side-wall capacitance from the metal lines. There is also an additional power bus that runs between the logic cells and memory cells. Vertical power stripes in metal three are connected from the area reserved for logic cells to the supply rings. These stripes must be sufficiently wide to ensure adequate power is available to the logic cells.

The core is the area in the center of the diagram which contains the logic and memory cells. The memory cells are placed on three sides of the logic cells with their IO pins facing the core. This minimizes signal integrity problems since short wires can be used to connect the memory to the logic cells.

4.9 Placement

The logic cells in the core consist of approximately 18,900 gates from the standard cell library. Thin power and ground lines divide the block into rows of approximately 6 µm height as shown in Figure 4.8. These thin lines connect to the larger power stripes that run vertically though the region.
The logic cells are placed in rows, with even rows being placed normally and odd rows being mirrored vertically. This allows for the sharing of the horizontal power and ground lines between rows. A placer (in this case QPlace) was used to calculate the best position for each cell amongst the rows, based on its timing requirements and proximity to other cells. QPlace is aware of cells that are part of the critical path, and it attempts to give these cells priority over other cells in placement. It is also aware of signal integrity and will place cells so that signal integrity problems are minimized.

The initial placement density was 70-75%, but this resulted in an area that did not entirely fill the cavity created by the memory cells. The area for logic cells was manually expanded to a placement density of approximately 50% such that it included the entire area bounded by power rings. This was done because of signal integrity concerns, since it places the logic cells that communicate with memory much closer to the memories themselves. No extra die area was used, however, since the area allocated for the logic cells is determined by the dimensions of the memories and the bounding box defined by the memory block placement.
Since there was some unused area, an attempt was made to place spare logic cells and VDD-VSS capacitance cells into the extra spaces between cells. Spare cells, such as inverters, multiplexors and registers, can sometimes be used to fix logical problems with the IC after fabrication. This is done by modifying the metal layers of the IC without changing the base layers. Metal and via masks (metal 2, via 2 and higher) are relatively inexpensive and can be changed quickly, while base layer and contact, metal 1 and via 1 masks are very costly and take more time to change.

Capacitance cells provide local repositories of charge that are very close to the switching circuits and this helps resolve many IR drop issues. Capacitors that are placed further away, such as on the package or on the board suffer from too much parasitic inductance and resistance to be of much help in high speed designs. This is not an issue for this IC, but it would become an issue for a higher speed design.

Unfortunately, the EDA software encountered numerous problems when trying to place spare cells and capacitor cells, and hence this idea was not utilized in this chip.

4.9.1 Clock Distribution via a Clock Tree

Every sequential element in the ASIC, such as the registers and memories, must have a connection to the clock network. The purpose of the clock network is to distribute a uniform clock signal with minimal delay and skew.

Due to the large capacitive load on the clock network, a balanced tree of buffers is used to provide adequate drive strength while minimizing skew. CTGen is used on the placed design to generate the clock tree. Based on the position of the cells and the number of connections to the clock signal, it strategically places clock buffers throughout the rows. The total delay through the clock tree was 2.42 ns with 26 ps of skew, which is relatively small when compared to the 16.7 ns critical path. The transition times were less than 50ps which yielded good performance at the expense of switching power in the clock tree. In this design there is only a single clock domain, since there is no need for multiple clocks. Information about the clock network is summarized below:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum transition time at leaf pins</td>
<td>0.046ns</td>
</tr>
<tr>
<td>Minimum insertion delay to leaf pins</td>
<td>2.393ns</td>
</tr>
<tr>
<td>Maximum insertion delay to leaf pins</td>
<td>2.420ns</td>
</tr>
<tr>
<td>Maximum skew between leaf pins</td>
<td>0.026ns</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>Depth of clock tree</td>
<td>22</td>
</tr>
<tr>
<td>Total buffers in clock</td>
<td>1036</td>
</tr>
<tr>
<td>tree</td>
<td></td>
</tr>
<tr>
<td>Leaf pins driven by clock</td>
<td>1273</td>
</tr>
</tbody>
</table>

**Table 4.2: Clock Tree Summary**

In a future version of this IC, the clock tree constraints could likely be relaxed such that skew and transition time were both less than 200ps, which would save power without reducing performance significantly. It would also reduce area since fewer buffers would be needed to meet these constraints.

### 4.10 Routing

Once the position of the logic and clock cells has been finalized, an automated router is used to connect them together using the available metal layers. Silicon Ensemble is the Cadence router, and it was used to do both routing and signal integrity analysis.

Power and ground nets are routed first since these require larger than normal metal sizes. The power pads are connected to the power rings, which are connected to the power stripes. The clock net is routed second since this is the most skew sensitive net in the system. With the fundamental nets routed, the signal nets are then routed, with the most critical nets being routed first.

During routing, the wires used to interconnect cells add additional load, and hence delay to the system. Silicon Ensemble calculates these loads and resizes gates as needed to meet delay constraints, assuming sufficient area is available in the rows. During synthesis, Design Compiler attempts a similar optimization, but it is inferior to the Silicon Ensemble optimization since placement and parasitic data are not available at that time.

Crosstalk is a serious signal integrity problem that is caused by capacitive coupling between wires. Long wires, especially several wires in a parallel bus are most vulnerable to these problems. In the best case, it can result in delay as the line driver struggles to overcome the interference source. In the worst case, it can become strong enough to temporarily alter the logical value on the line. If this occurs near a clock edge, incorrect data can be sampled. As such,
the router must resolve any signal integrity problems. Silicon Ensemble has two methods of resolving this: shielded routing and buffer insertion. [55]

In shielded routing, a ground cage is constructed around vulnerable signal lines using adjacent metal layers and vias. This provides a ground path for the interfering signals. Silicon Ensemble is able to route shielded nets, but due to a problem with the Cadence LEF/DEF import flow, designs with shielded nets cannot be imported. [50]

As an alternative, a long line can be broken into two or more smaller lines by strategically placing buffers. By reducing the length of the line, capacitance and coupling are reduced. Unfortunately, due to a bug in Silicon Ensemble, buffer insertion does not function correctly. [50]

As such, there are few options to correct signal integrity problems. However, by floorplanning strategically, the number of long lines can be reduced, which resolves most signal integrity problems.

4.11 Layout - LVS and DRC

After the physical design is complete, it is processed by Calibre to perform Layout vs Schematic (LVS) verification. LVS proves that the implemented circuits match those that were designed and optimized by Synopsys and Silicon Ensemble. After LVS, the design is sent to a Design Rule Check (DRC) service to verify that it passes all the technology specific rules. There are usually some DRC errors, most of which are easily corrected. In this case, there were two difficult DRC errors. They were both metal spacing errors involving routing metal and metals inside the black box cells. Trial and error was the only way to resolve this since we do not have layouts for the black box cells.

4.12 Post Layout Results

4.12.1 Area and Cell Statistics

The final dimensions of the ASIC were 1.9 mm x 2.7 mm, for a total area of 5.13 mm$^2$ with a core area of 2.56 mm$^2$. There were 18908 logic cells, 7 memory SRAM blocks comprising a total of 57344 bits (of which 4356 bits were actively used), 35 IO cells and 31 power and ground IO cells.
The cell usage statistics for each different category of logic cell is shown in Figure 4.9.

![Bar Chart: Cell Statistics]

**Figure 4.9: Cell Statistics**

The percentage area allocated to each different element of the design is shown in Figure 4.10. The area for logic is relatively high since the region for logic cells was expanded to encompass the entire area bounded by memory, even though the core logic cells were routable in far less area. (The logic cells in the core were routable with 75% placement density.)
To evaluate the effectiveness of the EDA tools, it is useful to examine a table of buffer sizes for all logic cells in the design. This table categorizes each cell by its driving ability and groups them into categories based on their drive strength. Typically low drive strength cells are used for short distance local routes between cells, while high drive strength cells are used for long distance global routes. This means that if the EDA software has done a good job with placement and routing, then there should be many low drive strength cells and few high drive strength cells.

A table of buffer sizes is presented below.

<table>
<thead>
<tr>
<th>Buffer Size</th>
<th>Quantity</th>
<th>Percentage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0X</td>
<td>1736</td>
<td>9.2</td>
</tr>
<tr>
<td>1X</td>
<td>6001</td>
<td>31.7</td>
</tr>
<tr>
<td>2X</td>
<td>5961</td>
<td>31.5</td>
</tr>
<tr>
<td>4X</td>
<td>3780</td>
<td>20.0</td>
</tr>
<tr>
<td>8X</td>
<td>208</td>
<td>1.1</td>
</tr>
<tr>
<td>16X</td>
<td>212</td>
<td>1.1</td>
</tr>
<tr>
<td>32X</td>
<td>1032</td>
<td>5.6</td>
</tr>
</tbody>
</table>

Table 4.3: Buffer Sizes
From this it is seen that only 7.8% of the logic cells are 8X or stronger cells, which indicates that the placement and routing were relatively good. 73% of the high drive strength cells (16X and 32X) are used in the clock tree and not in the data path. 9.2% of cells are low drive strength 0X cells. Care must be taken with the placement and routing of these cells since they are very sensitive to their load capacitance.

4.12.2 Critical Path

Silicon Ensemble was used to extract the parasitics and determine the final timing through the critical path. For this extraction, worst case timing libraries and pessimistic assumptions were used. The analysis was done both with and without signal integrity and crosstalk analysis.

The critical path in this design is the slowest register to register path through the ACS feedback loop, with a total delay of 15.12 ns without crosstalk analysis and 16.70 ns with it. Based on the crosstalk analysis, this should theoretically allow an operating frequency of 59.8 MHz. The other 255 paths though the ACS feedback loop had delays of 16.70 ns to 15.64 ns.

The critical path and the other nine most critical paths are summarized in Table 4.3.

<table>
<thead>
<tr>
<th>Path</th>
<th>Start</th>
<th>End</th>
<th>Delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>current_state_reg_5</td>
<td>norm_res10_q_reg1</td>
<td>16.70</td>
</tr>
<tr>
<td>2</td>
<td>current_state_reg_5</td>
<td>norm_res10_q_reg0</td>
<td>16.65</td>
</tr>
<tr>
<td>3</td>
<td>current_state_reg_5</td>
<td>norm_res7_q_reg0</td>
<td>16.49</td>
</tr>
<tr>
<td>4</td>
<td>current_state_reg_5</td>
<td>norm_res3_q_reg5</td>
<td>16.49</td>
</tr>
<tr>
<td>5</td>
<td>current_state_reg_5</td>
<td>norm_res10_q_reg2</td>
<td>16.38</td>
</tr>
<tr>
<td>6</td>
<td>current_state_reg_5</td>
<td>norm_res10_q_reg5</td>
<td>16.38</td>
</tr>
<tr>
<td>7</td>
<td>current_state_reg_5</td>
<td>norm_res3_q_reg6</td>
<td>16.38</td>
</tr>
<tr>
<td>8</td>
<td>current_state_reg_5</td>
<td>norm_res10_q_reg6</td>
<td>16.38</td>
</tr>
<tr>
<td>9</td>
<td>current_state_reg_5</td>
<td>norm_res10_q_reg4</td>
<td>16.37</td>
</tr>
<tr>
<td>10</td>
<td>current_state_reg_5</td>
<td>norm_res10_q_reg3</td>
<td>16.36</td>
</tr>
</tbody>
</table>

Table 4.4: Top Ten Most Critical Paths

A table of buffer strengths for the critical path is presented below. The critical path is primarily composed of low strength buffers which imply that it was placed efficiently.

---

4 Various cell libraries use different nomenclature to denote the weakest drive cell. 0X, 0.5X and 1X are often used.
Table 4.5: Buffer Strength of Gates on the Critical Path

<table>
<thead>
<tr>
<th>Buffer Strength</th>
<th>Quantity</th>
<th>Percentage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0X</td>
<td>1</td>
<td>1.3</td>
</tr>
<tr>
<td>1X</td>
<td>9</td>
<td>12.0</td>
</tr>
<tr>
<td>2X</td>
<td>39</td>
<td>52.0</td>
</tr>
<tr>
<td>4X</td>
<td>23</td>
<td>30.7</td>
</tr>
<tr>
<td>8X</td>
<td>2</td>
<td>2.7</td>
</tr>
<tr>
<td>16X</td>
<td>1</td>
<td>1.3</td>
</tr>
</tbody>
</table>

4.12.3 Throughput

A single iteration of the turbo decoder takes 572 clock cycles since there are 22 Forward-Backward decode operations performed, each of which takes 26 clock cycles. At 16.7 ns per cycle (in the worst case corner), the total time required is 9552.4 ns, resulting in a throughput of 12.7 Mbps (megabits/second). For two iterations, the throughput decreases by a factor of 2 to 6.35 Mbps, at 4 iterations it is 3.175 Mbps. This calculation can be easily scaled with frequency.

Since 0.18µm technology is mature, using the worst case models is very pessimistic. Foundries have had time to optimize the process to increase yields of typical performance parts. However, due to variations in the fabrication process, some of parts will perform either more quickly or more slowly than the average. Production chips are often bin sorted and the higher speed parts are sold at a premium, while the lower speed parts are sold at a discount.

The critical path has been recalculated using typical and best case standard cell timing libraries, and the table below summarizes the throughput for each process corner (SS, TT, FF):

<table>
<thead>
<tr>
<th>Corner</th>
<th>Period (ns)</th>
<th>Frequency (MHz)</th>
<th>Throughput 1 Iteration (Mbps)</th>
<th>Throughput 4 Iterations (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS (Slow NMOS / Slow PMOS, VDD=1.62V, 110°C)</td>
<td>16.70</td>
<td>59.8</td>
<td>12.7</td>
<td>3.1</td>
</tr>
<tr>
<td>TT (Typical NMOS / Typical PMOS, VDD=1.8V, 25°C)</td>
<td>13.68</td>
<td>73.1</td>
<td>15.4</td>
<td>3.8</td>
</tr>
<tr>
<td>FF (Fast NMOS / Fast PMOS, VDD=1.92V, -40°C)</td>
<td>10.79</td>
<td>92.7</td>
<td>19.6</td>
<td>4.9</td>
</tr>
</tbody>
</table>

Table 4.6: Frequency and Throughput
4.12.4 Power Estimates and IR Drop Analysis

Based on the final layout, power simulations were performed assuming 50% switching activity. An attempt was made to use the actual test vectors from the Verilog simulation, but due to the way that Physical Design Planner and Silicon Ensemble add and rename nets, difficulties were encountered. Accurate load and crosstalk information was available, as was accurate slew rate information. However, estimates had to be made for the timing behaviour of the standard cells since the power analyzer required timing information in TLF 4.1 format, and only TLF 3.0 models were available\(^5\). Based on these assumptions, power consumption values were calculated at a variety of frequencies. They are shown below:

<table>
<thead>
<tr>
<th>Frequency (MHz)</th>
<th>Power (mW)</th>
<th>Core Current @ 1.8V (mA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>59.8</td>
<td>84.58</td>
<td>46.98</td>
</tr>
<tr>
<td>73.1</td>
<td>103.39</td>
<td>57.43</td>
</tr>
<tr>
<td>92.7</td>
<td>131.11</td>
<td>72.83</td>
</tr>
</tbody>
</table>

Table 4.7: Power Consumption

At the time of tape out, IR drop tools were unavailable. As an approximation, the current consumption was calculated and the power grid stripes were sized to accommodate this current plus 100% margin, while keeping IR drop under 3%. Detailed IR drop calculations are presented in Appendix 3.

4.12.5 Adjusted Area for Custom SRAM Blocks and IO Ring

As mentioned in Table 3.3, 4356 bits of memory were needed in total for both the Forward-Backward and turbo sections of the decoder. However, only 32x256=8192 bit memories were available. Due to the required bit widths and the need to access memories simultaneously, 7 memory blocks were used, for a total of 57344 bits, of which only 4356 bits were actively used. The 8192 bit memory cells were dual-ported, but they were used in a single-ported configuration.

After the completion of this work, a custom memory compiler was made available. By using this compiler and assuming the memories scale linearly with their logical size, the area of the

---

\(^5\) TLF format, or Timing Library Format is a special file type that was used by Silicon Ensemble and Physical Design Planner. This file contains information about the power consumption and delays of each of the standard cells.
memories could be reduced by a factor of 13. Practically, smaller memories are less efficient than larger memories since the utilities (sense amps, decoders) consume a proportionally greater area. Therefore, a scaling factor of 10 is reasonable. Since the floorplan was limited primarily by the size of the memories, it can now be compacted and the core shrunk for additional savings. (The core was originally enlarged to fill the entire area between memory cells, even though it was routable with less area.)

If we consider the product turbo decoder to be a hard macro that will be integrated into a larger ASIC, then we can also discard the IO ring. Using these assumptions, it would be possible to shrink the die to approximately 1.7 mm$^2$ in 0.18µm CMOS.

4.13 Fabrication and Packaging

The finished design was fabricated by the Canadian Microelectronics Corporation (CMC) and packaged into 85 pin PGA packages. The package was a generic package that was not customized for this design, and as such it had no internal power rings or capacitors between power and ground.

4.14 Testing and Test Results

The finished ASIC was tested at the CMC Advanced Digital Systems Laboratory using a Verigy 93000 tester. This is an advanced mixed mode tester with a maximum digital test frequency of 1.25 GHz.

CMC provides a standard digital load board that connects to the tester, while the IC designer usually supplies an interface board that connects between the load board and the IC. Due to costs, a high speed interface board could not be fabricated, and instead a lower speed board was used. This allowed for full functional verification, but limited maximum speed.

Test vectors were prepared based on the results of the bit accurate fixed-point C program. These were used to generate VCD format test vectors, which were translated into a format the tester could understand using a conversion utility [1]. 100 test vectors, each corresponding to a code block of 209 bits were run through the IC and the results compared to their expected values. 80 of the vectors were standard vectors generated at a 0 to 3dB SNR, while 20 of the vectors were noiseless vectors. The noiseless vectors were used to stress the arithmetic circuits since they
would involve higher absolute magnitudes which would be more stressful on the normalization and saturation circuits. No anomalies were found during these tests.

At 20MHz all 100 vectors passed functional testing. At speeds above this, intermittent failures were observed. This was due to the limitations of the interface board, since raising or lowering the core voltage had no effect. Average power was measured during the tests and leakage power was measured at 25°C with the IC idle. The results of the testing are summarized in Table 4.8.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Operating Frequency</td>
<td>20 MHz</td>
</tr>
<tr>
<td>Average Dynamic Power</td>
<td>36.6 mW</td>
</tr>
<tr>
<td>Average Static Leakage Power @ 25°C</td>
<td>7.2 µW</td>
</tr>
</tbody>
</table>

**Table 4.8: Test Results**

In total, 5 ICs were fabricated of which all 5 were functionally correct. The 100 test vectors were only run on a single IC, while a limited set of 10 test vectors were run on the other 4. The purpose of this test was to determine the approximate yield. Since the tests were run manually, it was not feasible to run 100 test vectors on all 5 ICs.
A die is shown in Figure 4.11.

Figure 4.11: Die Micrograph
A final summary of design and test parameter is provided in Table 4.9.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMOS Technology</td>
<td>0.18µm</td>
</tr>
<tr>
<td>Metal and Poly Stack</td>
<td>1P6M</td>
</tr>
<tr>
<td>Core Voltage Supply (V)</td>
<td>1.8</td>
</tr>
<tr>
<td>IO Voltage Supply (V)</td>
<td>3.3</td>
</tr>
<tr>
<td>Die Area (mm$^2$)</td>
<td>5.13</td>
</tr>
<tr>
<td>Core Area (mm$^2$)</td>
<td>2.56</td>
</tr>
<tr>
<td>Area of Hard Macro (mm$^2$)</td>
<td>1.73</td>
</tr>
<tr>
<td>Total Number of Memories (32x256)</td>
<td>7</td>
</tr>
<tr>
<td>Total Number of Standard Cells</td>
<td>18908</td>
</tr>
<tr>
<td>Total Number of Registers</td>
<td>1145</td>
</tr>
<tr>
<td>Total Number of IO Cells</td>
<td>66</td>
</tr>
<tr>
<td>Total Transistors (estimate)</td>
<td>155k</td>
</tr>
<tr>
<td>Transistor Density (transistors per mm$^2$, estimate)</td>
<td>52k</td>
</tr>
<tr>
<td>Target Clock Frequency (MHz)</td>
<td>59.8</td>
</tr>
<tr>
<td>Target Throughput After 4 Iterations @ Target Clock Frequency (Mbps)</td>
<td>3.1</td>
</tr>
<tr>
<td>Estimated Power Dissipation @ Target Clock Frequency (mW)</td>
<td>84.58</td>
</tr>
<tr>
<td>Tested Clock Frequency (MHz)</td>
<td>20</td>
</tr>
<tr>
<td>Measured Throughput After 4 Iterations @ Tested Clock Frequency (Mbps)</td>
<td>1.0</td>
</tr>
<tr>
<td>Measured Power Dissipation @ Tested Clock Frequency (mW)</td>
<td>36.6</td>
</tr>
<tr>
<td>Measured Normalized Power At 4 Iterations (mW/Mbps)</td>
<td>36.6</td>
</tr>
<tr>
<td>Measured Energy per Bit At 4 Iterations (nJ)</td>
<td>34.6</td>
</tr>
<tr>
<td>Measured Leakage Power (µW) @ 25°C</td>
<td>7.2</td>
</tr>
</tbody>
</table>

Table 4.9: Summary of Design and Performance Parameters

4.15 Technology Scaling

This turbo decoder was fabricated in 0.18µm 1P6M CMOS technology. However, process technology is continually advancing which makes it useful to examine how the decoder will scale to other technologies. With every generation of process technology propagation delay decreases, area decreases and supply voltage decreases.

A standard method of process performance estimation with technology scaling is to compare fan out four (FO4) delays of an inverter since this is a good metric of overall process performance. The FO4 delay is the average propagation delay through an inverter that is driving four identical inverters. The delay through the critical path can be scaled according to this metric. Area can be estimated by comparing the sizes of the inverter cell between the two processes and scaling
accordingly. This assumes that the utilization percentage for the core area remains constant, which depends upon the number of standard cell routing tracks and the number of metal layers in the original and scaled technologies. Dynamic power can be estimated by comparing the supply voltages and the total load capacitance at the output of the FO4 inverter. However, estimating the load capacitance is difficult since wire capacitance dominates at finer geometries, and the wire capacitance is highly dependent on the placement and routing. Therefore, dynamic power estimation should be seen as only a coarse approximation.

Unfortunately, leakage also increases with smaller gate lengths and this increases standby power dissipation, which is a serious problem for mobile devices. This will be discussed in further detail in the next section.

Currently we are on the cusp of 32nm technology and 45nm is available at premium foundries. This makes 45nm an obvious scaling target. We will also consider scaling to 65nm since this is an interesting intermediate point and because 65nm processes are inexpensive and readily available from almost all foundries.

<table>
<thead>
<tr>
<th>CMOS Process Type</th>
<th>Core Area (mm²)</th>
<th>VDD (V)</th>
<th>Frequency (MHz)</th>
<th>Throughput 1 Iteration (Mbps)</th>
<th>Throughput 4 Iterations (Mbps)</th>
<th>Normalized Power @ 4 Iterations (mW/Mbps)</th>
<th>Leakage Power @25C (uW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.18µm</td>
<td>1.73</td>
<td>1.8</td>
<td>59.8</td>
<td>12.7</td>
<td>3.1</td>
<td>11.7</td>
<td>7.2</td>
</tr>
<tr>
<td>65nm</td>
<td>0.23</td>
<td>1.2</td>
<td>220</td>
<td>46.7</td>
<td>11.7</td>
<td>2.1</td>
<td>55</td>
</tr>
<tr>
<td>45nm</td>
<td>0.11</td>
<td>1.0</td>
<td>300</td>
<td>63.7</td>
<td>15.9</td>
<td>1.9</td>
<td>395⁶</td>
</tr>
</tbody>
</table>

**Table 4.10: Predicted Performance Parameters with Technology Scaling**

**4.15.1 Advanced Techniques For Minimizing Power Consumption**

With more modern technologies, static power consumption will be a significant problem. It is primarily due to subthreshold conduction which occurs when a transistor is in the logical “off” state since it can still conduct a small amount of current [56]. This is due to the shorter gate lengths and lower threshold voltages (V<sub>t</sub>) of modern transistors.

If this design were ported to either 65nm or 45nm, there would be significant design changes to reduce static power consumption. Firstly, during the ASIC design flow, the EDA tools would be

---

⁶ Assuming high V<sub>t</sub> cells used on all paths other than top 10 critical paths.
instructed to use standard $V_t$ devices only for the critical path (or top few most critical paths), while using high $V_t$ devices everywhere else. This does not affect performance, but it reduces leakage since high $V_t$ devices leak less than standard $V_t$ devices. Secondly, the ASIC would be divided into independent functional blocks which would be separated into independent voltage islands. By using a power or ground gating circuit, blocks can then be powered on and off as they are needed. The voltage supply to each block can also be reduced if that block does not need to run at its full speed. This is most commonly done at the system level, and would be of use if the turbo decoder were used as a hard macro in a larger communications ASIC.

Foundries have also started to introduce low power processes that are optimized for mobile designs. On these processes, some performance is lost, but leakage power is greatly reduced.

4.16 Hardware Scaling to Support Other Product Codes Including WiMAX

4.16.1 Area Estimates

Based on the results of chapters 3 and 4, we can estimate the area of a modified decoder to support various product codes including WiMAX. We assume that the core scales linearly with the number of ACS units, the memory scales linearly with the number of bits to be stored, and the interleaver will be implemented as combinational logic. Furthermore, we assume that this would be implemented as a hard macro and as such, we do not include the area for the IO ring.
Table 4.11: Area Estimates for Conventional and WiMAX Product Code Support
(The (15,11) code used in this thesis is highlighted in gray.)

With either 65nm or 45nm process, it is reasonable to support any of the WiMAX product codes. It is also possible to support larger conventional codes, including the (1023, 1013) product code.

4.16.2 Throughput Estimates

It is possible to estimate throughput by using the results of the previous section and adjusting for the extended parity information of the WiMAX code. Throughput results are presented for both conventional and WiMAX codes in Table 4.12.

Table 4.12: Throughput Estimates for Conventional and WiMAX Product Code Support
(The (15,11) code used in this thesis is highlighted in gray.)

WiMAX supports a theoretical maximum data rate of 75Mbps, but the standards committee has stated that 45Mbps will be the practical maximum. As of this writing, interest is primarily in the lower bit rates, with Sprint being the first provider to deploy WiMAX at a rate of 2Mbps to
4Mbps. [57] A scaled version of this decoder could easily meet a 4Mbps WiMAX bit rate with less power than listed since it could be down clocked.

4.17 Summary

This chapter has described the ASIC design, implementation and testing of the turbo decoder. The main points of this chapter were:

1. The overall EDA Flow was presented.

2. The synthesis process was described.

3. Special attention was given to design for test features to get the highest possible fault coverage.

4. Floorplanning and physical design was studied in detail to ensure that any signal integrity and power distribution problems were resolved.

5. According to post layout simulations, the decoder should run at 59.8 MHz with 84.58 mW power consumption at 1.8 V.

6. A test strategy was implemented to determine the performance of the ASIC. The chip as designed was functionally correct. An operating frequency of 20 MHz at 36.6 mW was obtained. The ASIC could likely operate at higher speed, but it was limited by the test fixture.
Chapter 5: Conclusions

5.1 Summary and Conclusions

Turbo decoding is an ongoing area of interest in communications and VLSI. With the advent of wireless networks, and their rapid growth and adoption for consumer applications, the demand for error correcting ICs will only increase.

The thesis began with a discussion of the Forward-Backward algorithm for the turbo decoding of product codes. From this, a floating-point and fixed-point C simulation was created. The design of the digital architecture was then discussed, and was implemented using Verilog. The Verilog description was synthesized, placed and routed to yield an integrated circuit. Test results for the final design were presented.

5.2 Contributions

As a result of this thesis, the following contributions were made:

1. A C program to simulate turbo decoding of block codes using the Forward-Backward algorithm was developed. It can perform floating or fixed-point decoding of a (209,121) product code.

2. A digital architecture for turbo decoders was defined, using the optimal bit width for fixed-point quantities. The scalability of this architecture was studied for varying length codes.

3. A Verilog model that implements the architecture was written for the case of the (209,121) code.

4. A 0.18µm SoC implementation of the turbo decoder was designed, fabricated and tested. From simulation, an operating frequency of 73.1 MHz at typical conditions was obtained, yielding a throughput of 3.8 Mbps with 4 decoding iterations, while consuming 103.4 mW. The total area was 5.13 mm². Assuming the ASIC would be used as a hard macro,
the area could be reduced to 1.7 mm\(^2\). The ASIC was tested at 20 MHz under typical conditions, which resulted in a throughput of 1.0 Mbps at 36.6 mW and 1.8V supply.

5.3 Future Work

Several valuable lessons were learned during the preparation of this thesis. In light of this, there are several modifications that could be made to improve a second version of the turbo decoder.

5.3.1 Alternatives to the Forward-Backward Algorithm

As an alternative to the Forward-Backward algorithm, the algorithm of [15] could also be used to decode these codes. It has decoding complexity \(n \log n\), and even though it makes extensive use of exponentiations and natural logarithms, it may be possible to optimize or approximate the algorithm to simplify hardware implementation. If this can be done, then it may yield more efficient hardware for large values of \(n\).

5.3.2 Normalization of State Metrics

In this work the state metrics are normalized by subtracting the largest state metric from the others at every clock cycle. This approach allows the state metrics to be represented with 8 bits, at the cost of the 5 adder delays through the Maxtree normalization circuit. A simple improvement to this scheme would be to increase the state metrics by 1 or 2 bits, and then to normalize every 3 or 4 clock cycles. This would increase net throughput, since the ASIC could be clocked at twice the existing frequency, although this would necessitate waiting an extra clock cycle while the normalization was performed every third or fourth cycle.

A more efficient approach, would be to modify the decoder to use modulo normalization as discussed in [47]. 8 bits will be insufficient for the state metrics in this case, so it may have to use 10 bits. This will increase the delay though each adder, the area of each adder and the size of memory for the state metrics. However, Maxtree can then be removed, which saves 31 adders, and reduces the critical path by 5 adder delays. It will be a tradeoff between fewer but slower elements in the critical path vs. faster elements, but in greater quantity. It is estimated that the operating frequency could be increased by approximately two times by making this change. The power consumption would also be reduced since Maxtree would not be switching every clock cycle.
5.3.3 Parallel Decoding of Forward and Backward Recursions

It would be possible to modify this design so that the Forward and Backward recursions are performed in parallel. This would require 16 additional ACS units, but would increase throughput since the number of cycles need to decode a single (15,11) Hamming code will be reduced from 26 to 15, resulting in a factor of 1.73 throughput increase.

5.3.4 Radix-4 Trellis Processing

It is also possible to implement the Forward-Backward algorithm using Radix-4 trellis processing. Using this approach, two stages of a binary trellis are calculated per cycle in a radix-4 trellis, at the expense of a four-way comparison per state. This means that the trellis can now be processed in \( \frac{N}{2} \) cycles, however each radix-4 cycle now results in 50% greater delay compared to a radix-2 cycle [39] [58]. Since each evaluation of a radix-4 stage yields sufficient information to decode 2 bits, net throughput is improved by a factor of 1.33. [39] Most modern Forward-Backward decoders take this approach [20] [22].

5.3.5 Interleaver Design

For this design, the interleaver permutations were stored in SRAM. As an alternative, it is possible to use combinational logic to generate permutations instead. Most industry specifications take this approach since it lowers overall system cost.

5.3.6 Semicustom Design

This design is well suited for a semicustom design flow, since it is composed primarily of saturation adders and modified ACS units. The schematics and layout of these blocks could be designed by hand, allowing a wider range of optimizations. Area savings would also be significant since the custom saturation adder and modified ACS unit would not have the overhead that accompanies standard cell designs.

These two blocks could then be characterized using a combination of HSpice and Synopsys, and used as black box cells in the Verilog code. The overall design would still be written in Verilog

---

\[ \text{Originally the decoding operation required 30 cycles, but with the revised decoding schedule of section 3.X we can reduce this to 26 cycles. Alternatively, by using 16 Forward ACS units and 16 Backward ACS units this can be reduced to 15 cycles.} \]
and synthesized. This would give performance close to a full custom design, since the critical path would be mostly custom blocks, while still maintaining the flexibility and time to market benefits that come from synthesis flows.

By moving to a semicustom flow, the frequency could likely be increased by a factor of 2.

5.3.7 EDA Software

Over the past few years, new EDA tools, such as Encounter and a SRAM memory compiler have become available. These tools are more efficient than previous tools and moving to this new design flow may yield significant savings. As was previously discussed, a memory compiler would reduce memory area by a factor of 10. Furthermore, smaller memories mean shorter wires, and fewer signal integrity problems. Encounter is better able to resolve these issues than the older software, which will mean less deviation between synthesis results and post layout results. The software used to prepare this thesis (Physical Design Planner and Silicon Ensemble) are both considered obsolete by Cadence and have been replaced by Encounter.

5.3.8 Summary Impact

By using modulo normalization, parallel Forward-Backward decoding, radix-4 trellis processing and a semicustom design flow, throughput would be improved by a factor of 9.2. This would allow for a 28.54 Mbps decoder in 0.18µm at 160 MHz or 145.7 Mbps decoder in 45nm CMOS at 800 MHz.
Appendix 1: Analysis of Floating-Point Simulation

Statistical analysis was run on the turbo decoder simulation over 1, 2, 3 and 4 iterations with at least 1000 product code blocks simulated at each SNR. The results of this analysis are presented here. For all key variables, maximum absolute value, minimum absolute value and minimum fraction are presented. The “bits” column shows the number of bits needed to approximately represent the corresponding value in the “values” column. With fewer iterations there was little decrease in the values and bit widths, and hence after 4 iterations the maximum values are observed. These values are used to establish an estimate of the number of bits required to approximate each variable, which will be used in hardware design.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>1 Iteration</th>
<th></th>
<th></th>
<th>2 Iterations</th>
<th></th>
<th></th>
<th>3 Iterations</th>
<th></th>
<th></th>
<th>4 Iterations</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Property</td>
<td>Values</td>
<td>Bits</td>
<td>Values</td>
<td>Bits</td>
<td>Values</td>
<td>Bits</td>
<td>Values</td>
<td>Bits</td>
<td>Values</td>
<td>Bits</td>
<td>Values</td>
</tr>
<tr>
<td>$\frac{2}{\sigma^2}Y_k$</td>
<td>Maximum Absolute Value</td>
<td>18.18538</td>
<td>5</td>
<td>18.18538</td>
<td>5</td>
<td>18.18538</td>
<td>5</td>
<td>18.18538</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\frac{2}{\sigma^2}Y_k$</td>
<td>Minimum Absolute Value</td>
<td>7.48E-07</td>
<td>21</td>
<td>7.48E-07</td>
<td>21</td>
<td>7.48E-07</td>
<td>21</td>
<td>7.48E-07</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\frac{2}{\sigma^2}Y_k$</td>
<td>Minimum Fraction</td>
<td>7.48E-07</td>
<td>21</td>
<td>7.48E-07</td>
<td>21</td>
<td>7.48E-07</td>
<td>21</td>
<td>7.48E-07</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L_0(U_k), L_1(U_k)$</td>
<td>Maximum Absolute Value</td>
<td>11.78039</td>
<td>4</td>
<td>19.55475</td>
<td>5</td>
<td>24.84615</td>
<td>5</td>
<td>24.95822</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L_0(U_k), L_1(U_k)$</td>
<td>Minimum Absolute Value</td>
<td>3.08E-07</td>
<td>22</td>
<td>3.08E-07</td>
<td>22</td>
<td>3.08E-07</td>
<td>22</td>
<td>2.86E-07</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L_0(U_k), L_1(U_k)$</td>
<td>Minimum Fraction</td>
<td>3.08E-07</td>
<td>22</td>
<td>3.08E-07</td>
<td>22</td>
<td>3.08E-07</td>
<td>22</td>
<td>2.86E-07</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$G_k(s,'s)$</td>
<td>Maximum Absolute Value</td>
<td>11.97644</td>
<td>4</td>
<td>14.92226</td>
<td>4</td>
<td>17.48478</td>
<td>5</td>
<td>17.55799</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$G_k(s,'s)$</td>
<td>Minimum Absolute Value</td>
<td>3.74E-07</td>
<td>22</td>
<td>3.74E-07</td>
<td>22</td>
<td>3.74E-07</td>
<td>22</td>
<td>3.74E-07</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$G_k(s,'s)$</td>
<td>Minimum Fraction</td>
<td>3.64E-07</td>
<td>22</td>
<td>3.64E-07</td>
<td>22</td>
<td>3.64E-07</td>
<td>22</td>
<td>2.13E-07</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{k</td>
<td>\text{pre}}(s), B_{k-1</td>
<td>\text{pre}}(s,')$</td>
<td>Maximum Absolute Value</td>
<td>50.82247</td>
<td>6</td>
<td>75.93362</td>
<td>7</td>
<td>92.20393</td>
<td>7</td>
<td>94.77869</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>$A_{k</td>
<td>\text{pre}}(s), B_{k-1</td>
<td>\text{pre}}(s,')$</td>
<td>Minimum Absolute Value</td>
<td>6.53E-08</td>
<td>24</td>
<td>3.82E-08</td>
<td>25</td>
<td>2.39E-08</td>
<td>25</td>
<td>1.92E-08</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>$A_{k</td>
<td>\text{pre}}(s), B_{k-1</td>
<td>\text{pre}}(s,')$</td>
<td>Minimum Fraction</td>
<td>3.91E-08</td>
<td>25</td>
<td>7.94E-11</td>
<td>34</td>
<td>7.94E-11</td>
<td>34</td>
<td>7.94E-11</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>$A_k(s), B_{k-1}(s,')$</td>
<td>Maximum Absolute Value</td>
<td>57.75759</td>
<td>6</td>
<td>86.49009</td>
<td>7</td>
<td>103.4451</td>
<td>7</td>
<td>103.998</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_k(s), B_{k-1}(s,')$</td>
<td>Minimum Absolute Value</td>
<td>6.50E-07</td>
<td>21</td>
<td>6.50E-07</td>
<td>21</td>
<td>6.40E-07</td>
<td>21</td>
<td>6.36E-07</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_k(s), B_{k-1}(s,')$</td>
<td>Minimum Fraction</td>
<td>4.04E-08</td>
<td>25</td>
<td>4.04E-08</td>
<td>25</td>
<td>2.82E-08</td>
<td>25</td>
<td>1.55E-08</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix 2: Fixed-Point Simulation Results

This table shows the percentage increase in BER relative to the floating-point simulation for various bit widths. All fixed-point simulations use the combinational MAX* circuit.

The decoder parameters were merged into two categories: Forward-Backward parameters, known as FB parameters, which included \((G_k(s', s), A_{k\text{pre}}(s), B_{k-1\text{pre}}(s'), A_k(s), B_{k-1}(s'))\) and turbo parameters, which included \((\frac{2}{\sigma^2} Y_k, L_0(U_k) \text{ and } L_1(U_k))\). For each category, whole bits and fractional bits were considered.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Turbo Whole Bits</th>
<th>Turbo Fractional Bits</th>
<th>FB Whole Bits</th>
<th>FB Fractional Bits</th>
<th>Avg BER Increase wrt Floating-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0.0 dB E_b/N_0</td>
<td>4.44%</td>
<td>6.03%</td>
<td>6.05%</td>
<td>6.56%</td>
<td>5.51%</td>
</tr>
<tr>
<td>0.5 dB E_b/N_0</td>
<td>5.78%</td>
<td>6.27%</td>
<td>6.32%</td>
<td>7.37%</td>
<td>7.90%</td>
</tr>
<tr>
<td>1.0 dB E_b/N_0</td>
<td>4.91%</td>
<td>5.48%</td>
<td>5.40%</td>
<td>5.80%</td>
<td>8.56%</td>
</tr>
<tr>
<td>1.5 dB E_b/N_0</td>
<td>5.48%</td>
<td>5.66%</td>
<td>5.57%</td>
<td>7.91%</td>
<td>9.78%</td>
</tr>
<tr>
<td>2.0 dB E_b/N_0</td>
<td>1.66%</td>
<td>2.83%</td>
<td>3.41%</td>
<td>4.98%</td>
<td>8.69%</td>
</tr>
<tr>
<td>2.5 dB E_b/N_0</td>
<td>5.34%</td>
<td>2.93%</td>
<td>3.73%</td>
<td>5.60%</td>
<td>10.91%</td>
</tr>
<tr>
<td>3.0 dB E_b/N_0</td>
<td>3.12%</td>
<td>1.65%</td>
<td>2.39%</td>
<td>7.71%</td>
<td>7.71%</td>
</tr>
<tr>
<td>Average BER</td>
<td>4.39%</td>
<td>4.41%</td>
<td>4.70%</td>
<td>6.56%</td>
<td>8.44%</td>
</tr>
<tr>
<td>This Thesis</td>
<td>4.70%</td>
<td>4.70%</td>
<td>4.70%</td>
<td>6.56%</td>
<td>8.44%</td>
</tr>
<tr>
<td>Comment</td>
<td>8 bits each</td>
<td>Best way to use 2 more bits</td>
<td>Best way to use 1 more bit</td>
<td>This Thesis</td>
<td>Best way to save 1 more bit</td>
</tr>
</tbody>
</table>

82
Appendix 3: IR Drop Analysis

Referring to Table 4.7, the peak operating current is 72.83mA at 1.8V supply. We will base IR drop analysis on 150mA to allow for 100% margin due to the inaccuracy of calculating IR drop without dedicated IR drop analysis software.

We will begin by modeling the current flow of the core, with the assumption that current is consumed uniformly by all the standard cells and that they are placed evenly across the core. Ten pairs of power and ground stripes were inserted into the core, which partitioned it into 9 sections, each 200µm wide by 1800µm tall. Each one of these sections will consume 16.67mA (150mA / 9). Furthermore, each section consisted of 133 rows of standard cells, yielding an average current consumption of 125µA (16.67mA / 133) per row. We will assume that the entire 125µA flows through a current source placed at center of each row.

Figure A3.1: Representative Core Power Grid

To estimate the resistance of the core grid we will consider both the horizontal and vertical power stripes. We will also assume that there is no IR drop through the power pads and the power rings since they are both made of very wide metal. Within the core, stripes of metal 1 run horizontally across each section with a total resistance of 134.6Ω. Stripes of metal 5 run vertically with a resistance of 0.202Ω between rows. Each via array that connects metal 5 to
metal 1 has approximately $1.7\Omega$ resistance. These resistance calculations are based on worst case resistance models.

From this, we can build a large network of resistors to model the grid and place current sources within it to model the logic cells. Spice simulation showed a maximum IR drop of 50mV. Three percent of 1.8V is 54mV, which means that the worst case IR drop should be less than 3% with 100% margin. These are acceptable margins for a production quality IC.
References


[38] H. Nickl, J. Hagenauer, and F. Burkert, "Approaching Shannon’s Capacity Limit by 0.27 dB Using Simple Hamming Codes," *IEEE Communications Letters*, vol. 1, p. 12, Sep.


[72] C. Berrou, "The Ten-Year-Old Turbo Codes are Entering into Service," *IEEE


