SOFT DECODING OF CONVOLUTIONAL PRODUCT CODES ON AN FPGA PLATFORM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MUSTAFA SANLI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

SEPTEMBER 2005

Approval of Graduate School of Natural and Applied Sciences.

_____

Prof. Dr. Canan ÖZGEN

Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

_____

Prof. Dr. İsmet ERKMEN

Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

_____

Asst. Prof. Dr. A. Özgür YILMAZ

Supervisor

Examining Committee Members:

Prof. Dr. Yalçın TANIK (METU, EE) _____

Asst. Prof. Dr. A. Özgür YILMAZ (METU, EE) _____

Assoc Prof. Dr. T. Engin TUNCER (METU, EE) _____

Asst. Prof. Dr. Cüneyt BAZLAMAÇCI (METU, EE) _____

Hacer SUNAY (TÜBİTAK, BİLTEN) _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Mustafa SANLI

ABSTRACT

SOFT DECODING OF CONVOLUTIONAL PRODUCT CODES ON AN FPGA PLATFORM

Sanlı, Mustafa
M.Sc., Department of Electrical and Electronics Engineering
Supervisor: Asst. Prof. Dr. Ali Özgür YILMAZ

September 2005, 79 pages

In today's world, high speed and accurate data transmission and storage is necessary in many fields of technology. The noise in the transmission channels and read-write errors occurring in the data storage devices cause data loss or slower data transmission. To solve these problems, the error rate of the received information must be minimized. Error correcting codes are used for detecting and correcting the errors.

Turbo coding is an efficient error correction method which is commonly used in various communication systems. In turbo coding, some redundancy is added to the data to be transmitted. The redundant data is used to recover original data from the received data. MAP algorithm is one of the most commonly used soft decision decoding algorithms.

In this thesis, hardware implementation of the MAP algorithm is studied. MAP decoding is verified on an FPGA. Virtex2Pro is the platform of choice in this study. The algorithm is written in the VHDL language. A MAP decoder is designed and its operation is verified. Using many MAP decoders concurrently, a convolutional product decoder is implemented as well. Area and speed limitations are discussed.

Keywords: MAP Algorithm, Turbo Coding

ÖZ

EVRİŞİMSEL ÇARPIM KODLARININ FPGA ÜZERİNDE YUMUŞAK ÇÖZÜMÜ

Sanlı, Mustafa
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Yard. Doç. Dr. Ali Özgür YILMAZ

Eylül 2005, 79 sayfa

Bugünün dünyasında, yüksek hızda hatasız veri aktarımı ve depolama, teknolojinin birçok alanında gereklidir. Veri yollarında yer alan gürültü ve depolama araçlarındaki okuma-yazma hataları veri kaybına ve düşük hızda veri aktarımına sebep olmaktadır. Bu sorunları çözebilmek için alınan bilgideki hata oranı en aza indirilmelidir. Hataları algılamak ve düzeltmek için hata düzeltme kodları kullanılmaktadır.

Turbo kodlama, çeşitli komünikasyon sistemlerinde yaygın olarak kullanılan, etkili bir hata düzeltme metodudur. Turbo kodlamada, aktarılacak olan veriye bazı fazlalıklar eklenir. Bu eklenen fazla bilgi, orijinal verinin alınan veriden elde edilmesinde kullanılır. MAP algoritması en yaygın kullanılan yumuşak çözüm algoritmalarından biridir.

Bu tezde, MAP algoritmasının donanımsal gerçeklenmesi yapılmıştır. MAP çözümü, FPGA üzerinde doğrulanmıştır. Bu çalışma Virtex2Pro üzerinde yapılmıştır. Algoritma VHDL diliyle yazılmıştır. Bir MAP çözücüsü tasarlanmış ve çalışması doğrulanmıştır. Ayrıca, birçok MAP çözücüsü aynı anda çalıştırılarak bir evrişimsel çarpım çözücü de gerçeklenmiştir. Alan ve hız sınırları tartışılmıştır.

Anahtar Kelimeler: MAP Algoritması, Turbo Kodlama

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor Asst. Prof. Dr. Ali Özgür YILMAZ for his guidance and friendly attitude. Next, I would like to thank Mr. Orhan GAZİ for his support in developing the algorithm and testing. I would also like to thank my colleagues Mr. Hüseyin YÜRÜK and Mr. Eren BABATAŞ for their help in the redaction of the thesis.

Special thank should be given to Aselsan Inc. for providing me time for my graduate studies.

Finally, I would like to thank my family for their love and encouragement throughout my life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Today, there is a need for high speed and accurate data storage and transmission. Improvements in technology enable high speed data transfer and processing. But, there is noise on transmission channels and there is also possibility of technical mistakes. These factors introduce errors to the received data. Hence, the aim must be to reduce and minimize the error rate in the received information with reasonably high data rates. Error correcting codes are used for detecting and correcting the errors.

Although error control coding is used in a variety of systems, it is especially useful in wireless communications systems. Such systems typically operate at low signal-to-noise ratios (SNR). The multipath behavior of the wireless channels also causes problems. As a result of the wireless environment, the received signal possibly contains many errors. Due to ubiquitous

communication desires of people, demand for wireless communication is growing rapidly each day. Hence, there is now a widespread need for fast and efficient error detecting and correcting systems.

The main idea of channel coding is to add redundancy to the data to be transmitted where the redundant data is used to recover the original data from the received data. Turbo codes constitute a special class of concatenated codes which were discovered in 1993 [1]. Turbo coding is a comparably recent advance in channel coding which proved itself in the last decade to be commonly used in various new communication systems and standards. Turbo codes perform quite well with reasonable complexity very close to the capacity especially in power limited channels such as wireless channels [1]. Turbo codes are composed of convolutional codes and their decoding is done by information exchange between the decoders of convolutional codes. That's why, maximum a posteriori (MAP) decoding of convolutional codes received large interest since the discovery of turbo codes.

MAP algorithm is one of the most commonly used soft decision decoding algorithms [2], [15-26]. MAP is an algorithm to estimate random parameters with prior distributions. In communications framework it is used to estimate the most likely information bit transmitted in a codeword. The plain form of the MAP algorithm is complex and not suitable for hardware implementation. The BCJR algorithm [2] offers an efficient tool to implement the MAP algorithm. The complexity of BCJR algorithm is linear with the length of the sequence input to it and it can be easily implemented in hardware [3]. Many hardware implementations are available [9] – [13].

In this thesis, hardware implementation of the MAP algorithm is studied. MAP decoding is verified on an FPGA. Virtex2Pro is the platform of choice in this study. The algorithm is written in the VHDL language. A MAP decoder is designed and its operation is verified. Based on the implemented MAP

decoder, a parallel decoder structure for a product code is constructed. Although product codes are usually built by using block codes, we constructed a product code structure with convolutional codes. Many MAP decoders will be run concurrently in this parallel implementation. Area and speed limitations will be discussed.

The outline of this script is as follows. A brief theoretical background in the area of error correcting codes is given in Chapter 2. Block codes and convolutional codes are explained and basic definitions are presented. Chapter 3 is devoted to the explanation of the MAP algorithm in detail. The VHDL implementation of the MAP algorithm and the corresponding test results are given in Chapter 4. Computational complexity calculations of the implemented algorithm are given in Chapter 5. In this chapter, area and speed limitations are discussed as well. Finally, we summarize and conclude the thesis in Chapter 6.

CHAPTER 2

ERROR CORRECTING CODES

This chapter gives general information on error correcting codes. Firstly, digital communication systems and types of codes are explained. Then, basic definitions are given. Finally, convolutional codes are explained and convolutional encoder representations are given.

2.1 Digital Communication Systems

Typical communication systems use several error correcting codes that are used to correct different types of errors. The physical medium which is used to transmit messages is called a channel. It can be a telephone line, a satellite link, a wireless channel etc. There are different sources of noise in channels. This introduces errors in the received message.

The basic idea of error correcting codes is to add some redundancy to a message before its transmission through a noisy channel. At the receiver, the original message can be recovered from the corrupted one if the number of errors is within the error correction capability of the code. Figure 2.1 shows a simplified model for a digital communication system. In the figure, *u* denotes the information sequence, *v* denotes the codeword and *y* denotes the received message.

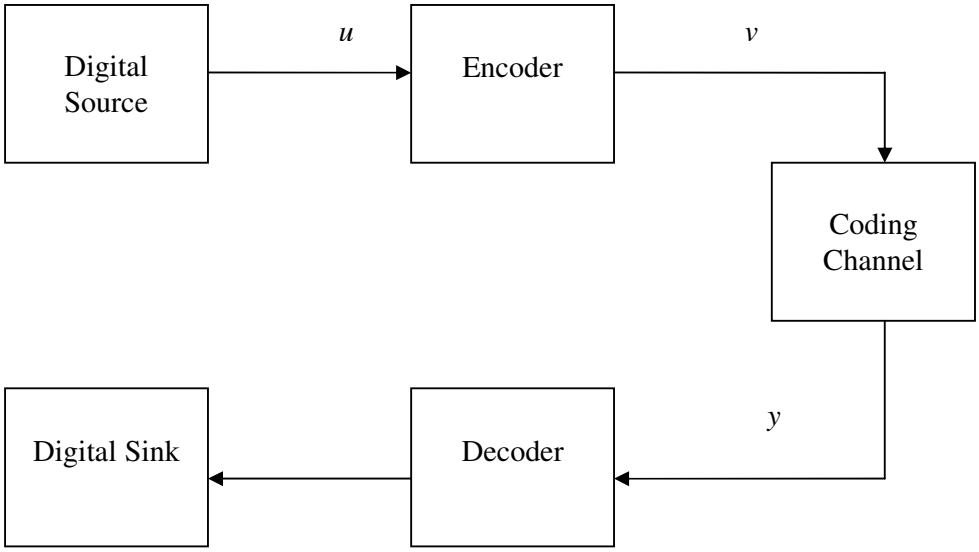

Figure 2.1: Simplified model of a digital communication system

In order to introduce error correcting capability to a digital communication system, redundancy must be added in a controlled manner. However, extra redundancy means a lower information transfer rate. Also, as the coding strategies become more complicated for correcting larger number of errors, fast and efficient encoding and decoding are difficult to achieve.

The mathematical description of a channel can be given as follows. A communication channel consists of an input alphabet A, an output alphabet B, and a real number $P(b \mid a)$ for each pair $a \in A,\ b \in B$. $P(b \mid a)$ is the probability that b is received, given that a is transmitted.

2.2 Types of Codes [31]

Mainly, there are two types of codes in channel coding: Block codes and convolutional codes. The properties of these types of codes are given in the following paragraphs.

2.2.1 Block Codes

In a block code, the encoder divides the incoming information sequence into message blocks of k symbols with symbol alphabet size q. This length-k information block is called a message. There are q^k possible different messages. The encoder transforms this message into an n-symbol codeword. Hence, there are q^k different codewords. The set of q^k codewords of length n is called an (n, k) block code. The ratio k/n is called the code rate. Each message is encoded independently based on a generator matrix. The encoder has no memory. It can be implemented with a combinational circuit. Block codes are codes that are constructed based on algebraic structures. There are many efficient optimal and suboptimal decoding algorithms for block codes; however few of them work directly on the channel observation. In general, every block code has a trellis (which will be defined under convolutional codes discussion) through which optimal decoding can be done. Though, trellises of block codes are time-varying and their state complexity can grow quite large especially when the code rate is neither very small nor very large [27].

2.2.2 Convolutional codes

Each encoded block depends not only on the length-k message but also on the m previous message blocks. Hence, the encoder has a memory of order m as opposed to the no-memory property in block encoders. It can be implemented with a sequential circuit. More explanation will be provided in Section 2.4.

2.3 Basic Definitions

- A word is a sequence of symbols.

- A code is a set of vectors called codewords.

- A block code is a set of fixed length codewords. The fixed length of these codewords is called the block length and it is denoted by n.

- A block code of size M defined over an alphabet with q symbols is a set of M q-ary sequences, each of length n. When $q = 2$, the symbols are called bits and the code is a binary code. Generally,

$$M = q^k \tag{2.1}$$

  for some integer k, and such a code is called an (n, k) code.

- The code rate of an (n, k) code is the ratio (k/n). Smaller code rate means greater redundancy. A code with greater redundancy is able to detect and correct more erroneous symbols, but this reduces the rate of information transmission.

2.4 Convolutional Codes

In convolutional codes, each block of k symbols is mapped into a block of n symbols. These n symbols are not only determined by the present information symbols but also by the previous information bits. This can be represented by a finite state machine.

Convolutional codes are usually generated by shift registers. Figure 2.2 shows an example of a convolutional encoder where the shift registers are shown with the empty rectangulars which act as delay elements. The adders perform modulo-2 addition. We will only consider binary codes in the remainder of this thesis.

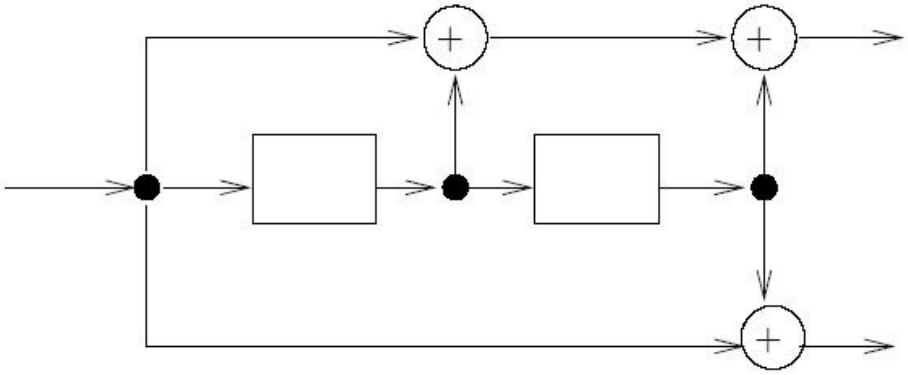

Figure 2.2: A convolutional encoder

The code rate for a convolutional code is defined as

$$code\ rate = \frac{k}{n}, \tag{2.2}$$

where k is the number of parallel input information bits and n is the number of parallel output encoded bits at one time interval. The constraint length K for a convolutional code is defined as

$$K = m+1, \tag{2.3}$$

where m is the maximum number of stages (memory size) in any shift register.

2.4.1 Convolutional Encoder Representations

Convolutional encoders can be implemented by finite state machines. With this idea, convolutional encoders are generally shown in three ways:

- Tree diagram representation
- State diagram representation
- Trellis diagram representation

2.4.1.1 Tree Diagram Representation

The tree diagram representation shows all possible sequences. Figure 2.3 shows an example of the tree diagram representation. In the tree diagram, lines represent inputs. A solid line represents 0 and a dashed line represents 1. Outputs are shown on the branches of the tree. An input information sequence defines a specific path through the tree diagram. For example, if the input sequence is given as “0101”, then looking at the figure, it is possible to find the outputs as “00 11 10 00”.

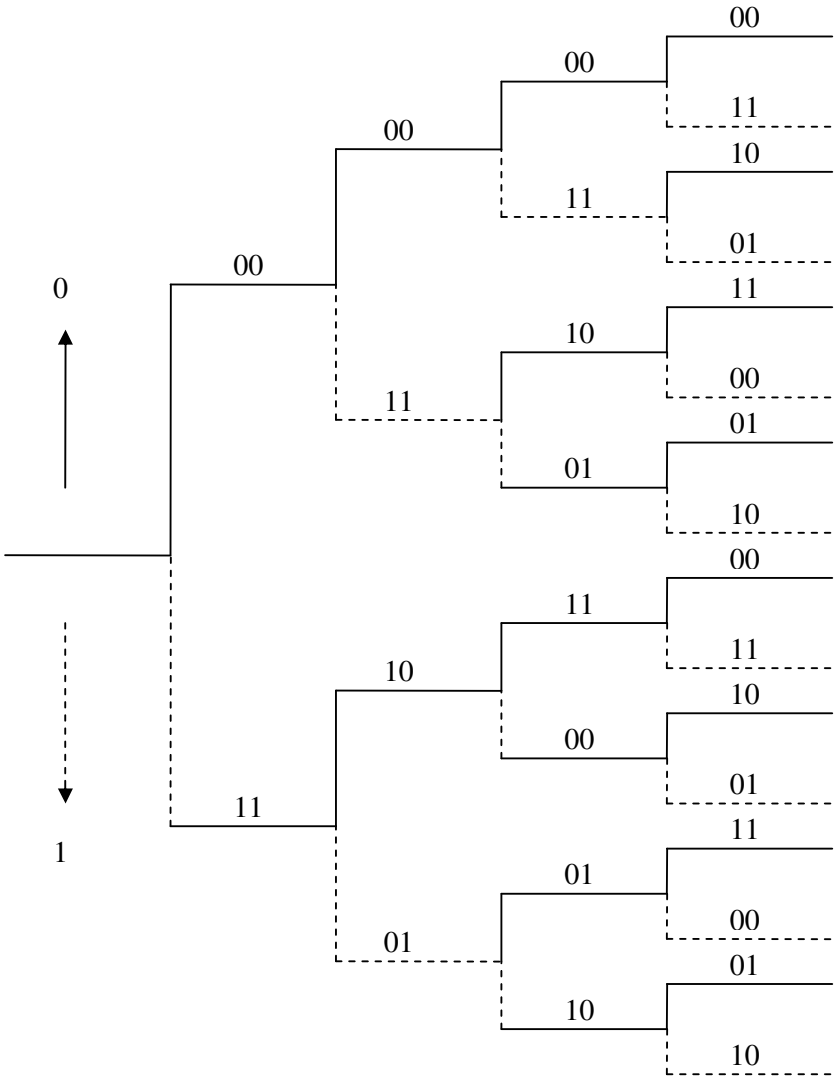

Figure 2.3: The tree diagram representation of a convolutional code

The tree diagram shows the passage of time as we move on the tree branches. When compared to a state diagram representation, instead of jumping between the states, we traverse the branches of the tree depending on whether a 1 or 0 is received.

2.4.1.2 State Diagram Representation

The state diagram shows the state information of the encoder on a diagram. Figure 2.4 shows the state diagram of a convolutional encoder. Each node represents a particular state of a convolutional encoder. The state of a convolutional encoder is defined by the current content in the shift registers. At any time, the encoder is in any one of the states shown in the figure. The lines going to and from the nodes show possible state transitions. At any time, the input may be 0 or 1 for the convolutional encoder considered here. Each of these inputs causes the encoder to jump into a different state. The output of the encoder is also given in the state diagram. In the notation *i/jk* of the figure, *i* denotes the input causing the transition and *jk* denotes the corresponding output. The state diagram contains the same information as the state table but this is a graphical representation. The state diagram does not have the time dimension.

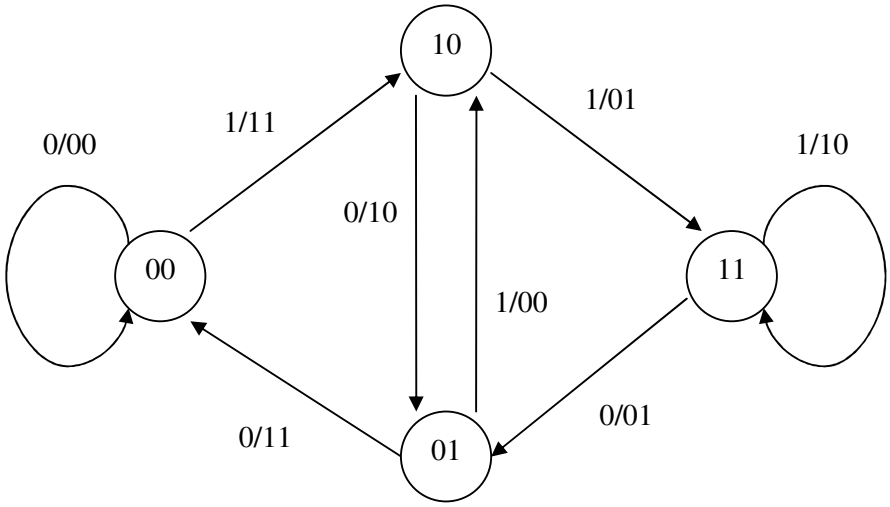

Figure 2.4: An example state diagram of a convolutional encoder

2.4.1.3 Trellis Diagram Representation

Trellis diagram shows all possible state transitions at each time step. It is generated by adding a time index to a state diagram. Figure 2.5 shows the trellis diagram of a 4-state convolutional code.

Trellis diagrams are preferred over both the tree and the state diagrams because they represent linear time sequencing of the events. One of the axes is discrete time and the other contains all possible states. As time passes, we move horizontally on the trellis diagram. In each transition, new bits arrive.

To draw the trellis diagram, all the possible states are marked in the *x*-axis. Then, each state is connected to the next state by the corresponding input and output for that transition. For example, in each state a 0 or a 1 may arrive. In the figure, the arrows going upward represent a 0 and the ones going downward represents a 1. The diagram always begins at state 00, and continues.

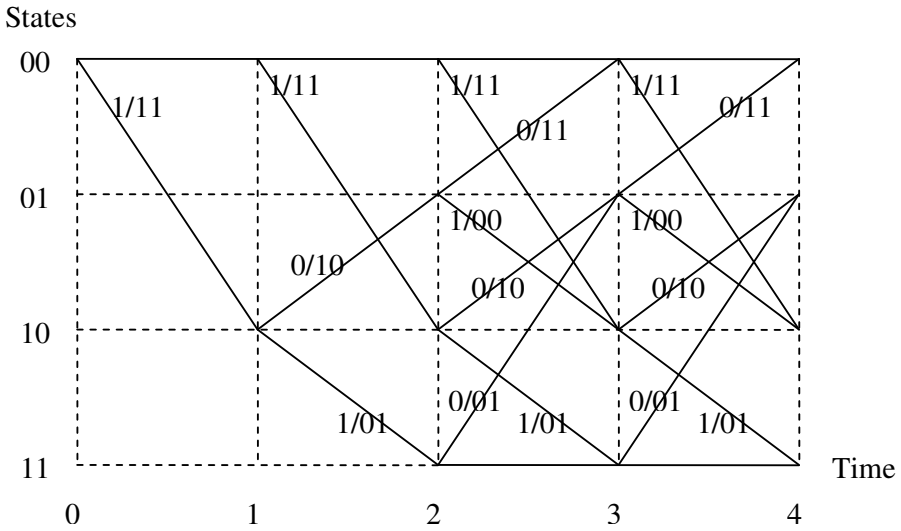

Figure 2.5: The trellis diagram of a 4-state convolutional code

CHAPTER 3

THE MAP ALGORITHM [32]

In the past, wireless systems were used mainly for voice communication. But today, they transmit data, audio, video, and voice on the same line. In order to build reliable wireless systems, designers must use advanced error correction mechanisms. Turbo coding is an efficient error correction approach which is proposed to be used in many modern wireless systems. BCJR algorithm is one of the most commonly used algorithms in turbo decoding [2], [15-26]. It is an efficient algorithm through which one can obtain a posteriori probabilities of individual bits.

In this chapter, BCJR-based MAP algorithm which is the algorithm used in this thesis is explained in detail.

3.1 MAP Decoding of Convolutional Codes

Figure 2.1 shows a simplified model of a digital communication system. A data symbol u_k is the collection of all the s data bits (u_1^k, u_2^k,..., u_s^k) that makes a symbol out of bits. By the finite state representation explained in Chapter 2, the output and the next state of a convolutional code can be determined by the current state and the data. Hence, it can be written as

$$P(u_k = i|\mathrm{y}) = \sum_{(m',m):\, u_k = i} P(S_{k-1} = m', S_k = m|\mathrm{y}), \tag{3.1}$$

where S_k is the encoder's state at discrete time index k, y is all the channel observation, i is a symbol from the alphabet of data symbols.

We define a function $M_k(m',m)$ such that

$$M_k(m',m) = P(S_{k-1} = m', S_k = m|\mathrm{y}). \tag{3.2}$$

Then, (3.1) can be written as

$$P(u_k = i|\mathrm{y}) = \sum_{(m',m):\, u_k = i} M_k(m',m). \tag{3.3}$$

For the paths having the transition $S_{k-1} = m' \to S_k = m$, $M_k(m',m)$ can be written as

$$M_k(m',m) = \sum_{u \in D_k(m',m)} P(u|y), \tag{3.4}$$

where u is the specific input sequence to the encoder, $D_k(m',m)$ is the set of all input sequences which traverse the code's trellis graph through state m' at time $k-1$ and state m at time k.

Using Bayes' theorem, the following expression can be written as

$$P(u \mid y) = \frac{f(y \mid u)P(u)}{f(y)}. \tag{3.5}$$

Recall that $f(y)$ is constant for all k and thus can be disregarded. (3.4) can be rewritten as

$$M_k(m',m) = \sum_{u \in D_k(m',m)} f(y|u)P(u). \tag{3.6}$$

As the symbols in the data sequence are assumed to be independent,

$$p(u) = \prod_{t=1}^{N} P(u_l). \tag{3.7}$$

We know that for each u, there is a corresponding c_u. Then,

$$f(y \mid u) = f(y \mid c_u), \tag{3.8}$$

where c_u is the encoded sequence corresponding to u.

We define a state variable for the past data and the output sequence c_u:

$$\sigma_1(c_u) = [y_1, ..., y_l; c_u], \tag{3.9}$$

where $\sigma_1(c_u)$ is the state for the l^{th} symbol in the codeword c_u and y_k denotes the channel signal for the k^{th} symbol interval.

Applying the chain rule, the following is obtained

$$f(y \mid c_u) = \prod_{l=1}^{N} f(y_l \mid \sigma_{l-1}(c_u)). \tag{3.10}$$

When we substitute (3.7), (3.8) and (3.10) into (3.6), we get the following expression:

$$M_k(m', m) = \sum_{u \in D_k(m', m)} \prod_{l=1}^{N} P(u_l) f(y_l \mid \sigma_{l-1}(c_u)). \tag{3.11}$$

When memoryless channels are considered and the convolutional code's encoder is a FSM,

$$f(y_l \mid \sigma_{l-1}(c_u)) = f(y_l \mid y_{1,\ldots,} y_{l-1}; c_u), \tag{3.12}$$

$$= f(y_l \mid c_u), \tag{3.13}$$

$$= f(y_l \mid S_{l-1}(u), S_l(u)), \tag{3.14}$$

where $S_l(u)$ is the state of the encoder at time l when the input to the encoder is u.

Then, (3.10) can be simplified to

$$M_k(m',m) = \sum_{u \in D_k(m',m)} \prod_{l=1}^{N} \gamma_1(S_{l-1}(u), S_l(u)), \tag{3.15}$$

where

$$\gamma_l(m',m) = P(u_l) f(y_l \mid m',m). \tag{3.16}$$

(3.15) can be computed using the BCJR algorithm. This algorithm is used to sum up the metrics of all paths passing through a fixed transition in a given trellis. The algorithm calculates (3.15) recursively. Two functions $\alpha_k(m)$ and $\beta_k(m)$ are used. First one is used for the calculations that are in the forward direction and the other is used for calculations in the backward direction. The sum of all path metrics ending up state *m* at time *k* is calculated by $\alpha_k(m)$. The sum of all path metrics starting at state *m* at time *k* is calculated by $\beta_k(m)$.

The following recursive calculations are used to find $\alpha_k(m)$ and $\beta_k(m)$:

$$\alpha_k(m) = \sum_{m'} \alpha_{k-1}(m') \gamma_k(m',m), \tag{3.17}$$

$$\beta_k(m) = \sum_{m'} \beta_{k+1}(m') \gamma_{k+1}(m,m'). \tag{3.18}$$

The initial values required for the recursive calculations are supplied to the encoder. The encoder starts calculation at zero state. If no termination strategy is used, the final state can be any of the states with equal likelihood. $M_k(m',m)$ can be written as

$$M_k(m',m) = \alpha_{k-1}(m')\gamma_k(m',m)\beta_k(m)\,. \tag{3.19}$$

In this case, (3.4) can be written as:

$$P(u_k = i \mid y) = \sum_{(m',m):\,u_k = i} \alpha_{k-1}(m')\gamma_k(m',m)\beta_k(m)\,. \tag{3.20}$$

CHAPTER 4

IMPLEMENTATION OF THE MAP ALGORITHM AND TEST RESULTS

In this chapter, implementation of the MAP algorithm is explained and test results are given. First, log domain MAP algorithm is explained. Then, main parts of the code such as data storage, max* operations, and recursive calculations are explored. In the recursive calculations part, branch metric calculations, alpha and beta calculations, LL calculations and decoded data calculations are given. Then, implementation of the convolutional product decoder is explained where many decoders work in paralel. Finally, test results are presented.

4.1 Log Domain MAP Algorithm

In Chapter 3, the MAP algorithm is explained in detail. We use this algorithm to find out which probability is bigger: $P(u_k=0|y)$ or $P(u_k=1|y)$. This information is used to guess the original codeword by looking at the received one.

To achieve this, we define $LL(u_k)$. $LL(u_k)$ is defined as

$$LL(u_k) = \ln \frac{P(u_k = 1 | y)}{P(u_k = 0 | y)}. \tag{4.1}$$

If $LL(u_k) > 0$, this means $P(u_k=1|y) > P(u_k=0|y)$. Hence, we may conclude that $u_k=1$. $u_k=0$ otherwise. Equation (3.20) tells that, the only variables necessary for the calculation of $LL(u_k)$ are $\alpha_{k-1}(m')$, $\gamma_k(m',m)$, and $\beta_k(m)$.

From Chapter 3, we know that

$$\alpha_k(m) = \sum_{m'} \alpha_{k-1}(m')\gamma_k(m',m), \tag{4.2}$$

$$\beta_k(m) = \sum_{m'} \beta_{k+1}(m')\gamma_{k+1}(m,m'). \tag{4.3}$$

Hence, once we know the values of $\gamma_k(m',m)$, both $\alpha_k(m')$ and $\beta_k(m)$ can be calculated recursively. At this point, all that remains is the computation of $\gamma_k(m',m)$. The probability domain version of the MAP algorithm is numerically unstable for long and even moderate codeword lengths. It involves many multiplications and logarithmic operations that require large

look-up tables. Hence, it is not suitable for VHDL coding. As a result of this, we will use the log domain version of it.

In the log domain MAP algorithm, $\alpha_k(m)$ is replaced by the forward metric

$$\tilde{\alpha}_k(m) = \ln(\alpha_k(m)). \tag{4.4}$$

Using equation (4.2), we can write

$$\tilde{\alpha}_k(m) = \ln\left(\sum_{m'} \alpha_{k-1}(m')\gamma_k(m',m)\right), \tag{4.5}$$

$$= \log\left(\sum_{m'} \exp\left(\tilde{\alpha}_{k-1}(m') + \tilde{\gamma}_k(m',m)\right)\right), \tag{4.6}$$

where the branch metric $\tilde{\gamma}_k(m',m)$ is given by

$$\tilde{\gamma}_k(m,m') = \ln(\tilde{\gamma}_k(m,m')). \tag{4.7}$$

The probability $\beta_k(m')$ is replaced by the backward metric.

$$\tilde{\beta}_k(m') = \ln(\beta_k(m')). \tag{4.8}$$

Using equation (4.3), we can write

$$\tilde{\beta}_k(m') = \ln\left(\sum_{m} \beta_{k+1}(m)\gamma_{k+1}(m',m)\right), \tag{4.9}$$

$$= \ln\left(\sum_{m} \exp\left(\tilde{\beta}_{k+1}(m) + \tilde{\gamma}_{k+1}(m',m)\right)\right). \tag{4.10}$$

Now, using equations (4.1) and (3.20), $LL(u_k)$ can be written as

$$LL(u_k) = \ln \frac{\sum_{(m',m):u_k=1} \alpha_{k-1}(m')\gamma_k(m',m)\beta_k(m)}{\sum_{(m',m):u_k=0} \alpha_{k-1}(m')\gamma_k(m',m)\beta_k(m)}, \tag{4.11}$$

$$= \ln\left[\sum_{(m',m):u_k=1} \exp\left(\tilde{\alpha}_{k-1}(m') + \tilde{\gamma}_k(m',m) + \tilde{\beta}_k(m)\right)\right] - \ln\left[\sum_{(m',m):u_k=0} \exp\left(\tilde{\alpha}_{k-1}(m') + \tilde{\gamma}_k(m',m) + \tilde{\beta}_k(m)\right)\right]. \tag{4.12}$$

At this point, to make the above calculations simpler and more suitable for VHDL coding, a new function max* is defined as [27]

$$\max^*(x, y) = \ln(e^x + e^y). \tag{4.13}$$

We know that the maximum of two numbers, max(x,y), can be calculated as

$$\max(x, y) = \ln\left(\frac{e^x + e^y}{1 + e^{-|x-y|}}\right). \tag{4.14}$$

Then, max*(x,y) can be written as

$$\max^*(x, y) = \max(x, y) + \ln(1 + e^{-|x-y|}). \tag{4.15}$$

It is clear and can easily be proven that

$$\max^*(x, y, z) = \max^*\left[\max^*(x, y), z\right]. \tag{4.16}$$

Now, we can rewrite equations (4.6), (4.10) and (4.13) using the max* function.

$$\tilde{\alpha}_k(m) = \max_{m'}{}^*\left(\tilde{\alpha}_{k-1}(m') + \tilde{\gamma}_k(m', m)\right), \tag{4.17}$$

$$\tilde{\beta}_k(m') = \max_{m}{}^*\left(\tilde{\beta}_{k+1}(m) + \tilde{\gamma}_{k+1}(m', m)\right), \tag{4.18}$$

$$LL(u_k) = \left[\max_{(m',m):u_k=1}{}^*\left(\tilde{\alpha}_{k-1}(m') + \tilde{\gamma}_k(m', m) + \tilde{\beta}_k(m)\right)\right] - \left[\max_{(m',m):u_k=0}{}^*\left(\tilde{\alpha}_{k-1}(m') + \tilde{\gamma}_k(m', m) + \tilde{\beta}_k(m)\right)\right]. \tag{4.19}$$

Equations (4.17), (4.18) and (4.19) show that the log domain computation of $LL(u_k)$ is very suitable for logic design. The max* function involves only a two-input max function and a lookup table for the term $\ln(1+e^{-|x-y|})$.

We know that

$$0 < \ln(1 + \exp(-|a-b|)) \leq \ln(2) \cong 0.693. \tag{4.20}$$

A small sized look-up table can be used for the $\ln(1 + \exp(-|a-b|))$ values. A table of size 8 is usually sufficient [30]. For a precision of 0.1, the values of (a-b) can be calculated for 8 different values of the log operation with step sizes of 0.1 and this look-up table can be used in the calculations.

4.2 Implementation of the Log Domain MAP Algorithm

The VHDL code mainly consists of three parts: Data storage, max* operations, and recursive calculations. Using the code, a sequential logic circuit is generated. All the operations take place in the rising edge of the clock. For generating the clock, the local oscillator of the FPGA Virtex2Pro is used. The clock frequency of the FPGA device is 150 MHz. In the VHDL code an asynchronous reset is provided. For data storage, registers and a RAM is used. Timing of the signals is very important in reading data from RAM and in making recursive calculations. To achieve this, a clock counter is used. The sequence of the events is defined according to this counter.

4.2.1 Data Storage

Received data is kept on the block RAM of the FPGA Virtex2Pro. Each data is accepted to have m decimal digits to the left of the decimal point and n digits to the right. There is no complex component in received data since binary shift keying is assumed. In the case of other modulation schemes, the code should be rewritten correspondingly. To overcome the difficulties of dealing with the decimal part, each data is multiplied by 10^n before storing into the RAM. Hence, in the RAM, there are $(m+n)$ digit integers. To reduce the complexity and save area, these data are entered into RAM as binary vectors.

For example, if data words have 3 decimal digits to the left of the decimal point and 1 digit to the right, each data is multiplied by 10 before they are stored into the RAM. In this case, to represent 4 decimal digits, 14-bit binary vectors are used. Hence, each word of the RAM is a 14-bit binary vector.

The channel information is stored in the vector Y. This vector can be written as

$$Y = [\, y_1\, y_{1p}\;\; y_2\, y_{2p}\;\; y_3\, y_{3p} \ldots\, y_N\, y_{Np}\,], \tag{4.21}$$

where y_k and y_{kp} are the k^{th} data and parity words. Hence, if the vector Y has N data words and N parity words, to store this data, a RAM size of $2N$ words is needed to store this data. In each clock cycle, only one word can be read from the RAM (unless special techniques are utilized). At the start of the program, all the received data is read from the RAM and written to registers. Hence, for a vector Y of N data words, a reading time of $2N$ clock cycles are required.

In the VHDL code, it is tried to use registers instead of RAMs whenever possible. This is due the fact that the contents of a group of registers can be read at a single clock cycle but only one word can be read from the RAM in each clock cycle. Hence, registers are used to store matrices and vectors.

4.2.2 Max* Operations

In the implementation, we use two max* functions: max2 and max4. Max2 and max4 operations are written as separate entities. They are mapped to the main code where necessary. The max2 entity has data, reset, and clock inputs. Its clock is synchronous with the master clock. It makes the calculation which is given in the equation (4.13).

A small sized look-up table is used for the $\ln(1 + \exp(-|a-b|))$ values. Max4 entity makes the calculation

$$\max4(a,b,c,d) = \max2(\max2(\max2(a,b),c),d). \tag{4.22}$$

where a, b, c, d are binary vectors. This calculation needs to run the max2 block 3 times. Hence, it needs 3 clock cycles. It is possible to make the same calculation in 2 clock cycles. Equation (4.22) can be written as

$$\max4\ (a,b,c,d) = \max2(\max2(a,b),\ \max2(c,d)). \tag{4.23}$$

Since max2 function was designed as an entity, it is used as a block in the max4 design. The max4 device is shown in Figure 4.1.

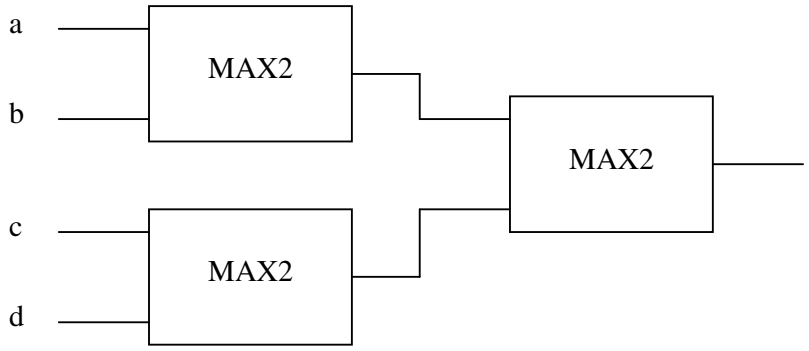

Figure 4.1: The structure of the max4 device

4.2.3 Recursive Calculations

The encoder can be thought of as a finite state machine. In the VHDL code, the state information is kept in the vector *S*. This vector has the form

$$S = [S_0,\ S_1,\ S_2,\ \ldots], \tag{4.24}$$

$$S_j = [S_{j0}\ S_{j1}], \tag{4.25}$$

where S_j is the j^{th} state and S_{j0}, S_{j1} denotes the outputs (data and parity bits) when the encoder is in the j^{th} state and a 0 or 1 is received respectively.

In this thesis, a (1, 5/7) systematic recursive convolutional code is implemented [31]. The algorithm will be explained assuming that this code is used. The encoder is shown in Figure 4.2. The corresponding trellis diagram is drawn in Figure 4.3.

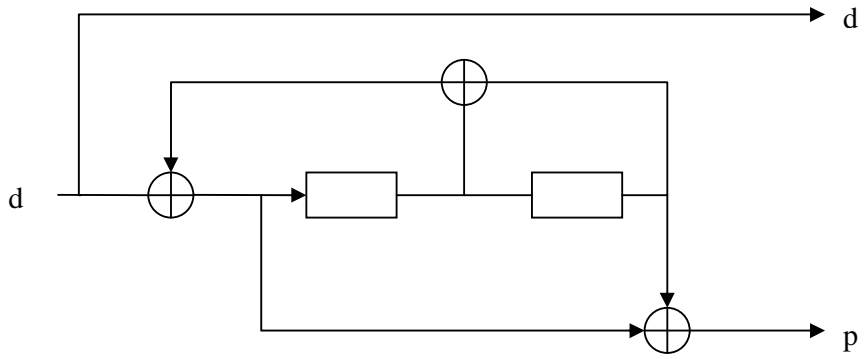

Figure 4.2: The (1, 5/7) convolutional encoder

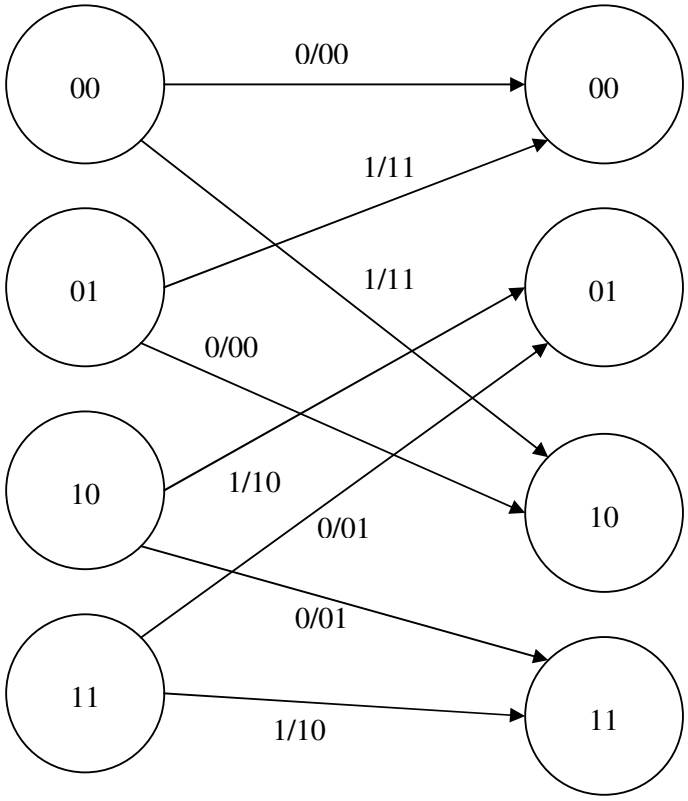

Figure 4.3: The one-step trellis diagram of the (1, 5/7) encoder.

Looking at Figure 4.3, the vector *S* is formed as follows. It must be noted that (2*u*-1) BPSK coding is used. Hence, binary 1 is represented by 1 and binary 0 is represented by -1.

$$S = [\ S_{00}\quad S_{01}\quad S_{10}\quad S_{11}\quad S_{20}\quad S_{21}\quad S_{30}\quad S_{31}], \tag{4.26}$$

$$S = [-1\ -1\quad 1\ \ 1\quad -1\ -1\quad 1\ 1\quad -1\ 1\quad 1\ -1\quad -1\ 1\quad 1\ -1]. \tag{4.27}$$

State information is used to calculate BM_k, branch metric vector for the received data pair. This branch metric is some constant times the logarithm of the conditional channel observation probability multiplied by the a priori probability [27]. This vector is formed as

$$BM^k = [BM_{00}{}^k \; BM_{01}{}^k \; BM_{10}{}^k \; BM_{11}{}^k \; BM_{20}{}^k \; BM_{21}{}^k \; BM_{30}{}^k \; BM_{31}{}^k \;], \quad (4.28)$$

where BM_k is the transition metric of each branch on the state diagram. It can be written as

$$BM_{ij}{}^k = K \; [\, y_k \, y_{kp}] \; S_{ij}{}^T + \; [\, p_{k0} \, p_{k1}] \quad (4.29)$$

where $K=1/\sigma^2$. σ is the noise standard deviation of the additive white Gaussian noise that is present in the channel. The values of k are between 1 and N. Hence, the values BM_k are kept in the matrix BM. To keep these data, $8N$ registers are required.

p_{kj} is a priori probability of the bit k being equal to j. p_{kj} is kept in the vector P. P is the a priori information vector for the data bits.

$$P = [\, p_{10} \, p_{11} \, p_{20} \, p_{21} \ldots \, p_{N0} \, p_{N1}], \quad (4.30)$$

$$p_{k0} = - \max^*(0, \, LL(k)), \quad (4.31)$$

$$p_{k1} = \max^*(0, \, LL(k)). \quad (4.32)$$

The values of $\tilde{\alpha}_k(m)$ required for the calculation of equation (4.19) are kept in the matrix alpha. This matrix consists of 4 vectors because our encoder has 4 states. Components of this matrix are in the form $\tilde{\alpha}_k(m)$ where the values of k are between 0 and N-1 and the values of m are between 0 and 3. $\tilde{\alpha}_k(m)$ is the logarithm of the probability of being at state m after the arrival of k^{th} data in the forward direction.

The values of $\tilde{\alpha}_k(m)$ are calculated recursively. We assume that the encoder is initially in the state 0. Hence, the probability of being in the state 0 is 1 and the

probability of being in all other states is 0. As we implement the algorithm in the log domain, the probability of being in the state 0 is log(1) which is equal to 0 and the probability of being in all other states is log(0). Due to the computational limitations, log(0) is taken to be the smallest number to be represented by the available number of bits. This smallest number is -800 when 14 bits are used for quantization.

Hence, the starting values are

$$\tilde{\alpha}_0(0) = 0, \tag{4.33}$$

$$\tilde{\alpha}_0(1) = -800, \tag{4.34}$$

$$\tilde{\alpha}_0(2) = -800, \tag{4.35}$$

$$\tilde{\alpha}_0(3) = -800. \tag{4.36}$$

For all the values of k between 1 and N-1, the values of $\tilde{\alpha}_k(m)$ are calculated recursively using equation (4.17). It is clear that the probability of being in a state can be calculated by summing the probabilities of traversals through branches leading to that state. The probability of each traversal is equal to the multiplication of the probability of being in a state that the traversal begins with and the branch metric leaving the state. The $\tilde{\alpha}_k(m)$ values for the code used in this study are obtained by the following equations

$$\tilde{\alpha}_k(0) = \max^*(\tilde{\alpha}_{k-1}(0) + \mathrm{BM}_{00}{}^{\mathrm{k}},\ \tilde{\alpha}_{k-1}(1) + \mathrm{BM}_{11}{}^{\mathrm{k}}), \tag{4.37}$$

$$\tilde{\alpha}_k(1) = \max^*(\tilde{\alpha}_{k-1}(3) + \mathrm{BM}_{30}{}^{\mathrm{k}},\ \tilde{\alpha}_{k-1}(2) + \mathrm{BM}_{21}{}^{\mathrm{k}}), \tag{4.38}$$

$$\tilde{\alpha}_k(2) = \max^*(\tilde{\alpha}_{k-1}(1) + \mathrm{BM}_{10}{}^{\mathrm{k}},\ \tilde{\alpha}_{k-1}(0) + \mathrm{BM}_{01}{}^{\mathrm{k}}), \tag{4.39}$$

$$\tilde{\alpha}_k(3) = \max^*(\tilde{\alpha}_{k-1}(2) + \mathrm{BM}_{20}{}^{\mathrm{k}},\ \tilde{\alpha}_{k-1}(3) + \mathrm{BM}_{31}{}^{\mathrm{k}}). \tag{4.40}$$

After the calculation of the matrix alpha, normalization of the newly alpha values is necessary. Otherwise, overflow or underflow problems might occur.

We will utilize an intermediate variable called the nonterm to perform normalization. Since $\tilde{\alpha}_k(m)$ constitute a probability measure, their sum over the states should add up to 1.

$$\text{nonterm } (k) = \max^* (\tilde{\alpha}_k(0), \tilde{\alpha}_k(1), \tilde{\alpha}_k(2), \tilde{\alpha}_k(3)). \tag{4.41}$$

Then, nonterms are subtracted from the $\tilde{\alpha}_k(m)$.

$$\tilde{\alpha}_k(0) = \tilde{\alpha}_k(0) - \text{nonterm}(k), \tag{4.42}$$

$$\tilde{\alpha}_k(1) = \tilde{\alpha}_k(1) - \text{nonterm}(k), \tag{4.43}$$

$$\tilde{\alpha}_k(2) = \tilde{\alpha}_k(2) - \text{nonterm}(k), \tag{4.44}$$

$$\tilde{\alpha}_k(3) = \tilde{\alpha}_k(3) - \text{nonterm}(k). \tag{4.45}$$

To store the matrix alpha, 4*N* registers are used.

The values of $\tilde{\beta}_k(m)$ are kept in the matrix beta. Components of this matrix are in the form $\tilde{\beta}_k(m)$ where the values of *k* are between 1 and *N* and the values of *m* are between 0 and 3. $\tilde{\beta}_k(m)$ is the logarithm of the probability of being at state *m* after the arrival of k^{th} data in the backward direction. This logarithm is calculated after all the data is received in the backward direction.

The values of $\tilde{\beta}_k(m)$ are calculated recursively. As trellis termination is used, the final state of the encoder is 0. Hence, the probability of being in the state 0 is 1 and the probability of being in all other states is 0. When these values are transferred to the log domain, the probability of being in the state 0 becomes log(1) which is equal to 0 and the probability of being in all other states becomes log(0) which is again taken to be -800 (for quantization with 14 bits).

Hence, the starting values are

$$\tilde{\beta}_N(0) = 0, \tag{4.46}$$

$$\tilde{\beta}_N(1) = -800, \tag{4.47}$$

$$\tilde{\beta}_N(2) = -800, \tag{4.48}$$

$$\tilde{\beta}_N(3) = -800. \tag{4.49}$$

If the trellis termination were not used, we wouldn't know the final state of the encoder. The final state could be any of the four states with equal probability. Then, the starting values would be all zero.

For all the values of k between N-1 and 1, the following calculations are made using equation (4.18):

$$\tilde{\beta}_k(0) = \max^*\left(\tilde{\beta}_{k+1}(0) + BM_{00}{}^{k+1}, \tilde{\beta}_{k+1}(2) + BM_{01}{}^{k+1}\right), \tag{4.50}$$

$$\tilde{\beta}_k(1) = \max^*\left(\tilde{\beta}_{k+1}(2) + BM_{10}{}^{k+1}, \tilde{\beta}_{k+1}(0) + BM_{11}{}^{k+1}\right), \tag{4.51}$$

$$\tilde{\beta}_k(2) = \max^*\left(\tilde{\beta}_{k+1}(3) + BM_{20}{}^{k+1}, \tilde{\beta}_{k+1}(1) + BM_{21}{}^{k+1}\right), \tag{4.52}$$

$$\tilde{\beta}_k(3) = \max^*\left(\tilde{\beta}_{k+1}(1) + BM_{30}{}^{k+1}, \tilde{\beta}_{k+1}(3) + BM_{31}{}^{k+1}\right). \tag{4.53}$$

After the calculation of the matrix beta, nonterms are calculated just as in the case of calculation of alphas.

$$\text{nonterm } (k) = \max^* \left(\tilde{\beta}_k(0), \tilde{\beta}_k(1), \tilde{\beta}_k(2), \tilde{\beta}_k(3)\right). \tag{4.54}$$

Then, nonterms are subtracted from the $\tilde{\beta}_k(m)$.

$$\tilde{\beta}_k(0) = \tilde{\beta}_k(0) - \text{nonterm}(k), \tag{4.55}$$

$$\tilde{\beta}_k(1) = \tilde{\beta}_k(1) - \text{nonterm}(k), \tag{4.56}$$

$$\tilde{\beta}_k(2) = \tilde{\beta}_k(2) - \text{nonterm}(k), \tag{4.57}$$

$$\tilde{\beta}_k(3) = \tilde{\beta}_k(3) - \text{nonterm}(k). \tag{4.58}$$

To store the matrix beta, 4N registers are used.

To simplify the LL calculations, we define two functions; sum0(k) and sum1(k). In the calculations, sum0(k) represents log($P(u_k=0|y)$) and sum1(k) represents log($P(u_k=1|y)$). Now, using the new variables, equation (4.19) can be written as

$$LL(k) = \text{sum1}(k) - \text{sum0}(k). \tag{4.59}$$

For all the values of k between 1 and N, the values of sum0(k) and sum1(k) are calculated using equation (4.19).

$$sum0(k) = \max^*\begin{pmatrix} \tilde{\alpha}_{k-1}(0) + BM_{00}{}^k + \tilde{\beta}_k(0), \\ \tilde{\alpha}_{k-1}(1) + BM_{10}{}^k + \tilde{\beta}_k(2), \\ \tilde{\alpha}_{k-1}(2) + BM_{20}{}^k + \tilde{\beta}_k(3), \\ \tilde{\alpha}_{k-1}(3) + BM_{30}{}^k + \tilde{\beta}_k(1) \end{pmatrix}, \tag{4.59}$$

$$sum1(k) = \max^*\begin{pmatrix} \tilde{\alpha}_{k-1}(0) + BM_{01}{}^k + \tilde{\beta}_k(2), \\ \tilde{\alpha}_{k-1}(1) + BM_{11}{}^k + \tilde{\beta}_k(0), \\ \tilde{\alpha}_{k-1}(2) + BM_{21}{}^k + \tilde{\beta}_k(1), \\ \tilde{\alpha}_{k-1}(3) + BM_{31}{}^k + \tilde{\beta}_k(3) \end{pmatrix}. \tag{4.60}$$

The values $LL(k)$ are kept in the vector LL. To store the vector LL, N registers are used.

The vector LL is also calculated for parity bits. The values LL for parity bits are kept in the vector LLp. This vector can be written as $LLp(k)$ where the values of k are between 1 and N.

For all the values of k between 1 and N, the values of *LLp* for parity bits can be calculated as

$$sum0(k) = \max^*\begin{pmatrix} \tilde{\alpha}_{k-1}(0) + BM_{00}{}^{k} + \tilde{\beta}_k(0), \\ \tilde{\alpha}_{k-1}(1) + BM_{10}{}^{k} + \tilde{\beta}_k(2), \\ \tilde{\alpha}_{k-1}(2) + BM_{21}{}^{k} + \tilde{\beta}_k(1), \\ \tilde{\alpha}_{k-1}(3) + BM_{31}{}^{k} + \tilde{\beta}_k(3) \end{pmatrix}, \tag{4.61}$$

$$sum1(k) = \max^*\begin{pmatrix} \tilde{\alpha}_{k-1}(0) + BM_{01}{}^{k} + \tilde{\beta}_k(2), \\ \tilde{\alpha}_{k-1}(1) + BM_{11}{}^{k} + \tilde{\beta}_k(0), \\ \tilde{\alpha}_{k-1}(2) + BM_{20}{}^{k} + \tilde{\beta}_k(1), \\ \tilde{\alpha}_{k-1}(3) + BM_{30}{}^{k} + \tilde{\beta}_k(2) \end{pmatrix}, \tag{4.62}$$

$$LLp(k) = \text{sum1}(k) - \text{sum0}(k). \tag{4.45}$$

We use the BCJR algorithm to find out the probabilities: $P(u_k=0|y)$ or $P(u_k=1|y)$. This information is used to guess the original codeword looking at the received one. Once we calculate the values of $LL(k)$ for all k, we can guess the original codeword using equation (4.1). The decoder accepts these guesses as the decoded data.

Decoded data is kept in the vector *DecDat*. The elements of this vector can be written as $DecDat(k)$ where the values of k are between 1 and N.

Decoded parity forms the vector *DecDatP*. Similarly, the elements of this vector can be written as $DecDatP(k)$ where the values of k are between 1 and N.

The decoded data are calculated using equation (4.1). A positive $LL(k)$ means that $P(u_k=1|y)$ is greater than $P(u_k=0|y)$. In this case, the decoded data is

guessed as 1. This information is kept in the vector *DecDat* as *DecDat*(*k*)=1. Similarly, a negative value of *LL*(*k*) means that *DecDat*(*k*) is 0. The vector *DecDatP* is also calculated in the same manner. These vectors are kept in 2*N* registers.

4.3 Implementation of the Convolutional Product Decoder

We explained the VHDL implementation of the BCJR decoder for a convolutional code up to this point. We will now use this as a component in a convolutional product decoder. Although product codes are usually constructed with block codes [27], they can be constructed with convolutional codes as well. We will study such a construction in this section.

In the encoder side of a product code, encoding is performed with the help of a matrix. This matrix determines how each encoder works. Convolutional codes are used to encode rows and columns in our construction.

First, the data to be sent is stored into a matrix. Then each row is encoded. Although different convolutional codes can be used to encode each row, the same systematic recursive convolutional code is used here. The data matrix dimension is $k \times k$, and encoded data matrix dimension is $n \times n$. Hence, our code is an ($n \times n$, $k \times k$) code. Then, this matrix is coded columnwise. The rate 1/2 systematic convolutional code is used to encode each row and column. Hence the overall data rate is ¼ when no trellis termination is employed.

The convolutional product coded data matrix is binary phase shift key (BPSK) modulated. Then, this signal is passed through an additive white Gaussian noise (AWGN) channel with double-sided power spectral density $N_0/2$ (noise variance σ^2).

The log-MAP soft decoding algorithm is used to iteratively decode the convolutional product code. First, column decoder decodes each column one by one, then the information obtained from the column decoder is given to the row decoder. Then, row decoding is performed and the information is given to the column decoder. This procedure is repeated many times.

In the convolutional product decoder design, a single RAM keeps the received data information. The decoders that are used in parallel processing are modified such that they have no RAMs. Data is directly read into registers. Hence, all the data is accessible in a single clock cycle. This saves operation time.

Figure 4.4 shows the convolutional product decoder. In the figure, y_c is the matrix that contains the channel information. To decode $2N$ codewords of length $2N$, a matrix y_c of size $2N \times 2N$ is required.

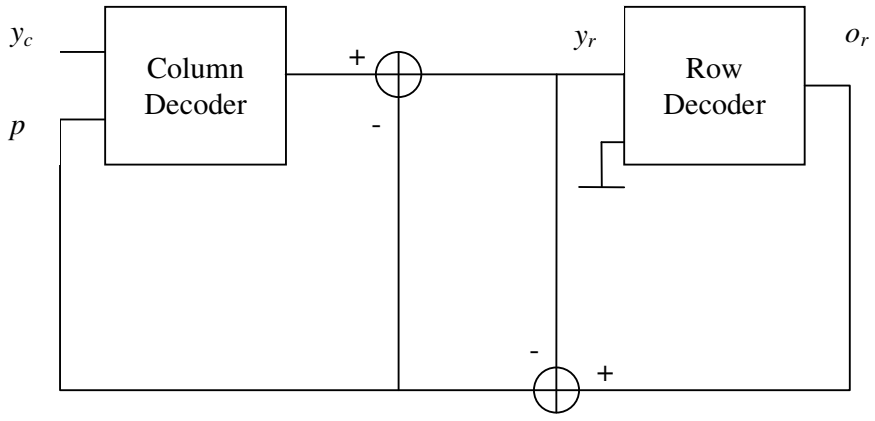

Figure 4.4: Convolutional product decoder

A priori information is kept in the matrix p. The matrix p has a size of $N \times 2N$. Column decoder takes each column of the y_c and p, then, writes the decoded

data to a column of the matrix y_r. To implement the column decoder, $2N$ decoders work in paralel. After that, the matrix p is subtracted from the y_r.

Row decoder takes all rows of the matrix y_r and writes the decoded results to a $N \times N$ matrix o_r. This time, all elements in the matrix p are equal to 0. To implement the row decoder, N decoders are used. the matrix p is obtained by subtracting y_r from the matrix o_r.

These operations are performed continuously. And, the matrix p is refreshed simultaneously.

4.4 Test Results

To verify the operation of the MAP decoder for N =5 and r=14, a data sequence "10101" is used. First, this data is encoded using the (1, 5/7) convolutional code. This gives the encoded sequence "1101100111". After BPSK (2u-1) mapping, (σ^2=0.2) noise is added to this sequence. Hence, the following input sequence is obtained:
0.673421 1.43184 -1.47894 1.64453 0.495061 -0.481105 -1.70511 1.05269 1.38844 1.16594

Finally, this sequence is given to the decoder. The decoder output and the critical variables are compared to the theoretical results. This comparison is given in Appendix A. It is observed that the decoded data is same as the original data. Also, the variables are very close to the theoretical results. The maximum difference between the implementation results and the hardware results is less than 10%.

Then, the convolutional product decoder is tested. We used the MAP decoders in parallel, to iteratively decode the convolutional product code. The decoding algorithm is explained in Section 4.3 in detail.

First, a 5×5 data matrix is selected and encoded according to the procedure given in Section 4.3. Then, a BPSK (2u-1) modulation is employed. After that, (σ^2=0.2) noise is added to this sequence. The 10×10 received data matrix which is obtained after these steps is given in Appendix B.

In this test, 10 decoders are used in paralel as column decoders and 5 decoders are used as row decoders. The test results are given in Appendix B. The test shows that, after the first iteration, 24 of the 25 data are decoded correctly and 1 of the data is erronous. After the second iteration, all the data are decoded correctly. The reliability of the decoder can be increased by increasing the number of iterations. In our test, 2 iterations were enough to correctly decode the received data.

4.5 Performance of The Implemented Decoders

To observe the performance of the decoders under different noise conditions, Bit Error Rate (BER) tests are performed. Bit error rate (BER) is defined as the number of erroneous bits divided by the total number of bits transmitted, received, or processed. The logarithm of this ratio is the BER of the system given in the figures. E_b/N_0 is defined as the ratio of energy per data bit (Eb) to the power spectral density of the noise (N0). E_b/N_0 is a measure of signal to noise ratio (SNR) for a digital communication system. It is used as the basic measure of how strong the signal is when compared to noise.

The single MAP decoder and the convolutional product decoder are tested with 6 different noise levels and with 4 different quantization cases. These quantization cases use different number of bits in the recursive calculations. In the tests, quantization is made with 14, 11, 9 and 4 bits. The noise level of AWGN channel is adjusted by using different values of E_b/ N_0.

Quantization is made after the BM matrix calculation. Up to this point, 14 bits are used. After BM matrix calculation, quantization is made for 4 different cases. In the quantization with 14 bits, all the elements of BM matrix are multiplied by 10, the values larger than 8000 are taken as 8000 and the values smaller than -8000 are taken as -8000. In the recursive calculations, the values between -8000 and 8000 are used. In the quantization with 11 bits, the elements of BM matrix are multiplied by 10 and and the values between -800 and 800 are used. The larger values are taken as 800 and the smaller values are taken as -800. In the quantization with 9 bits, the elements of the BM matrix are multiplied by 3 and and the values between -240 and 240 are used. In the quantization with 4 bits, quantization step is selected as 2. The values between -13 and 15 are used. In each quantization case, a look-up table of size 8 is used for max operations. The table is refreshed according to the bits available in each quantization case.

The code rate of the single MAP decoder is 0.5 and the code rate of the convolutional product decoder is 64/400. The convolutional product decoder is tested with 12 iterations. The values of K used in the BER tests are given in Table 4.1.

Table 4.1: The values of K used in the BER tests.

		K	
		Single MAP Decoder	Convolutional Product Decoder
E_b/N_0 (dB)	1	1.2590	0.4029
	2	1.5848	0.5072
	3	1.9952	0.6385
	4	2.5119	0.8038
	5	3.1626	1.0121
	6	3.9809	1.2755

The tests are made using 10000 random data sets and the test results are compared with the floating point simulation results. Table 4.2 gives the test results for the single MAP decoder and Figure 4.5 shows the test and floating point results of the single MAP decoder.

Table 4.2: The BER test results fort the single MAP decoder.

		BER			
		r=14	r=11	r=9	r=4
E_b/N_0 (dB)	1	-1.328	-1.309	-1.291	-1.217
	2	-1.863	-1.838	-1.805	-1.678
	3	-2.197	-2.105	-2.013	-1.841
	4	-3.225	-3.078	-2.885	-2.591
	5	-3.435	-3.271	-3.126	-2.837
	6	-4.417	-4.204	-3.903	-3.515

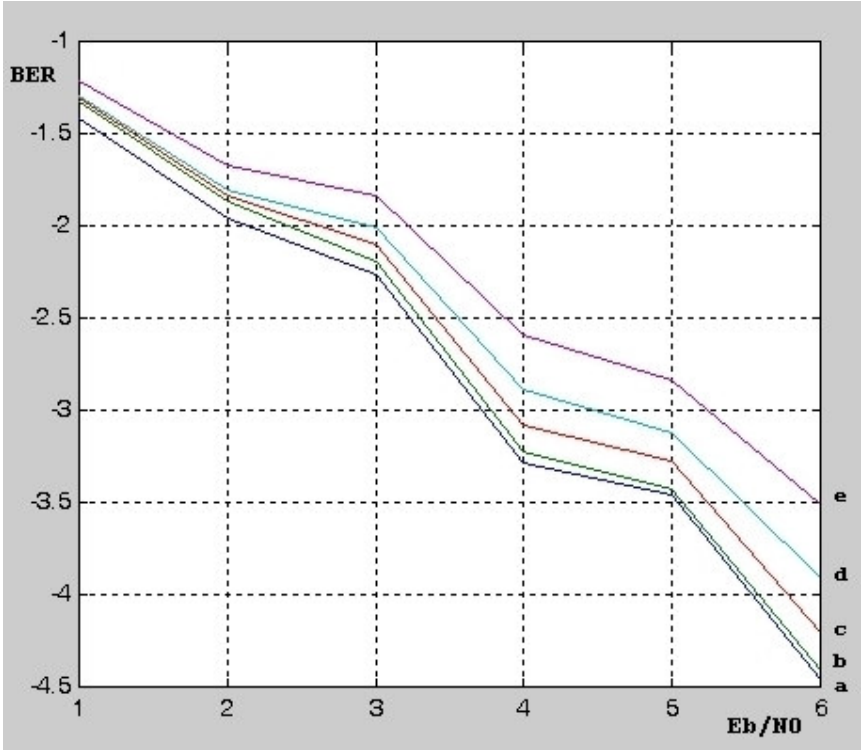

Figure 4.5: The test and floating point results of single MAP decoder. (a: floating point simulation, b: r=14 quantization, c: r=11 quantization, d: r=9 quantization, e: r=4 quantization, BER in logarithm)

Table 4.3 gives the test results fort the convolutional product decoder and Figure 4.6 shows the test and floating point results of the convolutional product decoder.

Table 4.3: The BER test results fort the convolutional product decoder.

		B E R			
		r=14	r=11	r=9	r=4
E_b/N_0 (dB)	1	-1.445	-1.431	-1.412	-1.363
	2	-2.123	-2.087	-2.004	-1.912
	3	-2.845	-2.761	-2.673	-2.541
	4	-3.714	-3.583	-3.456	-3.251
	5	-4.661	-4.498	-4.345	-4.084
	6	-5.653	-5.452	-5.234	-4.873

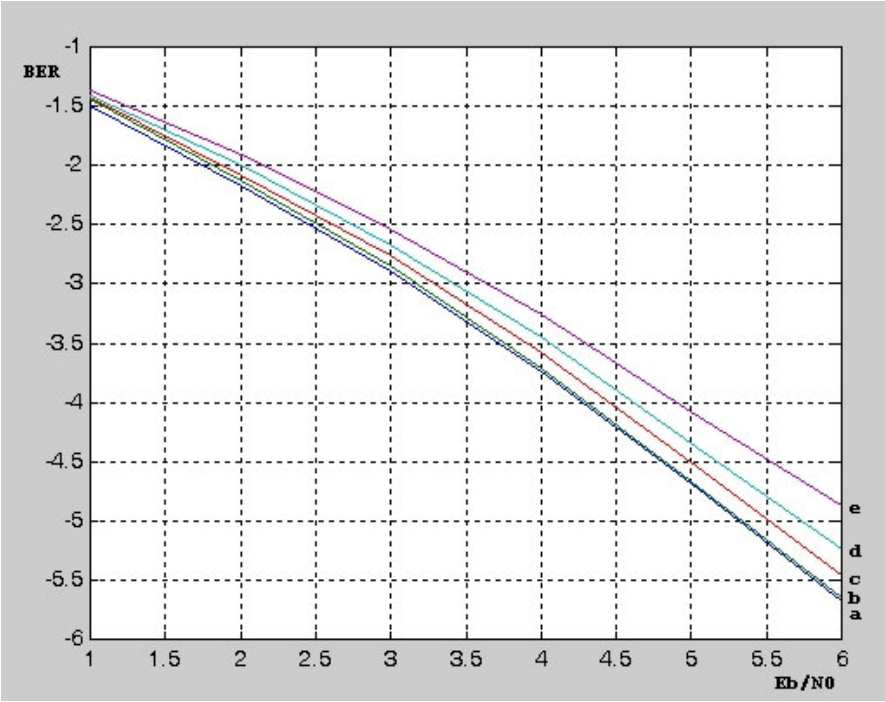

Figure 4.6: The test and floating point results of the convolutional product decoder. (a: floating point simulation, b: r=14 quantization, c: r=11 quantization, d: r=9 quantization, e: r=4 quantization, BER in logarithm)

The tests show that the quantization with 14 bits give results very close to the floating point simulations. It is also seen that the quantizations with 11 and 9 bits are very also very close to the floating point simulations. But the quantization with 4 bits causes a loss of 0.5 dB when compared to the quantization with 9 bits when the value of E_b/N_0 is 6.

CHAPTER 5

COMPUTATIONAL COMPLEXITY

In this chapter, computational complexity of the VHDL code is explained. First, operation time analysis of a single MAP decoder is given. In this section, operation time required for each part of the circuit is explained. Then, data storage requirements in the MAP decoder are investigated. The required RAM size is calculated. Also, the number of registers required in each section of the code is given. Later, the operation time and data storage requirements are investigated for the convolutional product decoder. Finally, area limitations of the FPGA are investigated. Looking at the area occupied by the designed circuits, maximum decoding capacity of the FPGA is estimated.

5.1 Operation Time Analysis of the MAP Decoder

Hardware implementation of the MAP algorithm is achieved using an FPGA. Virtex2Pro is the platform of choice in this study. The logic circuit is built using the VHDL language. A master clock of 150 MHz is used. All clock signals used in the circuit are synchronous with the master clock. The master clock signal is produced using the local oscillator of the FPGA. Rising edge of the clock is used.

Generally, in the area of turbo decoding, the most time consuming process is the data read and write cycles [27]. This fact also holds for our case. Channel information can be stored in a RAM. In this study, Xilinx IP core "Single-Port Block Memory v5.0" is the selected RAM structure.

As it was explained in Section 4.2.1, the received signal consists of words corresponding to data and parity bits. The vector Y is used to denote the received signal. For N data words, N parity words are used. Hence, the vector Y has a size of $2N$. The vector Y has the form

$$Y = [\, y_1\, y_{1p}\;\; y_2\, y_{2p}\;\; y_3\, y_{3p} \ldots\ldots\ldots y_N\, y_{Np}\,]. \tag{5.1}$$

At the start of the program, the whole Y must be read from the RAM and written to a register. In each clock cycle, only a single word can be read from the RAM. If each data and parity word is stored as a separate word in the RAM, a reading time of $2N$ clock cycles is required for N data words. In the cases when an input RAM is not necessary, data can be read directly into registers. This saves $2N$ clock cycles. In the calculation of the overall operation time, it is assumed that the data is read directly into registers.

Another possibility to minimize the time used for reading from RAM is to store more than one data in a memory word of the block RAM. For example, if we

have data words of 4 bits, then it is possible to store three data words in a RAM word as a 12 bit logic vector. Once the data is read, 3 data words can be transferred to corresponding registers. This enables us to read 3 data in a single clock cycle. Hence, the data reading process takes 1/3 of the previous time. It must be noted that, in the Single-Port Block Memory v5.0 which is used in this study, the memory word cannot be longer than 256 bits.

There are other data that must be read and written during the operation of the circuit. These data are required for the calculation of necessary variables. These data are stored not into RAM but into registers. Consequently, different data sources are accessible at the same time. Hence, many processes can run simultaneously. This is very effective in decreasing the operation time.

But, when registers are used, these memory elements occupy some FPGA area. This consumes some area which can be used in implementing logical circuits. When block RAMs are used, they are placed in an area of the FPGA which is reserved for block RAMs. Hence, no useful FPGA area is consumed. When the values N and r are not very large, the area loss caused by storing the data into registers is not significant when compared to the speed improvement. Hence, in this study, registers are the preferred memory elements.

As it was explained in Section 4.2.3, the branch metric matrix is an $N\times8$ matrix. The calculation of each element of this matrix requires 2 multipliers and 2 adders. This makes a total of $16N$ multipliers and $16N$ adders. As a result of the parallel operation, 3 clock cycles are dedicated to BM matrix calculation. 2 of the 3 clock cycles are required for multiplications and the remaining 1 clock cycle is used for addition. This time guarantees proper operation for a long range of K values.

If the calculations were not made in parallel, we would need only 2 multipliers and 2 adders. But, this time, the calculation would take $24N$ clock cycles. This

long operation time is not tolerable. As a result, the elements of the matrix are calculated in parallel.

For the vectors in alpha, *N*-1 recursive calculations are needed. Each calculation in equations (4.37) – (4.40) requires 1 clock cycle. After that, each nonterm calculation requires 2 clock cycles. So, 3 clock cycles are required for an $\tilde{\alpha}_k(m)$ calculation. Hence, totally, 3(*N*-1) clock cycles are required. The vectors $\tilde{\alpha}_k(0), \tilde{\alpha}_k(1), \tilde{\alpha}_k(2), \tilde{\alpha}_k(3)$ are calculated at the same time. In each clock cycle, 8 adders and 4 max2 blocks operate in parallel.

The values $\tilde{\beta}_k(m)$ are calculated simultaneously with $\tilde{\alpha}_k(m)$, no extra time is required. This saves us 3(*N*-1) clock cycles. Generally, $\tilde{\alpha}_k(m)$ and $\tilde{\beta}_k(m)$ calculations are very similar. So, it is possible to use the same circuit components both in $\tilde{\alpha}_k(m)$ and $\tilde{\beta}_k(m)$ calculations. This reduces the number of total circuit components required for $\tilde{\alpha}_k(m)$ and $\tilde{\beta}_k(m)$ calculations and saves area. But, as we prefer calculating the values $\tilde{\alpha}_k(m)$ and $\tilde{\beta}_k(m)$ at the same time, the circuit components cannot be shared. To achieve simultaneous operation, new adders, subtractors and max2 components must be used for $\tilde{\beta}_k(m)$ calculations. We can conclude that, with simultaneous operation, we decrease the operation time of the decoder but we waste some FPGA area. Hence, it can be stated that there is a trade off between computation time and circuit area. The trade off between computation time and circuit area is discussed in Section 5.3. In this case, we save 3(*N*-1) clock cycles by using only 8 more adders and 4 more max2 blocks.

In producing the vector *LL*, calculation of each *sum0* and *sum1* takes 2 clock cycles and subtraction of sum0 from sum1 requires 1 clock cycle. In equations (4.59) and (4.60), it can be seen that for each $LL(u_k)$ calculation, 16 parallel adders, 1 subtractor and 2 max4 blocks are required. The *LL* vector needs *N*

$LL(u_k)$ calculations. As these calculations are performed in parallel, total calculation time is equal to a single calculation time of 3 clock cycles. This parallel operation requires a total of 16*N* parallel adders, *N* subtractors and 2*N* max4 blocks. *LLp* calculations are made using the same adders and subtractors as the *LL* uses. Again, *LLp* calculation is made in 3 clock cycles.

The vector *LL* can also be calculated serially. This time, it would be possible to use the same adders, subtractors and max4 blocks in all calculations. But, the operation time would increase. As we found above, 3 clock cycles are needed to make a single $LL(u_k)$ calculation. For *N* serial $LL(u_k)$ calculations, 3*N* clock cycles would be required. But this time, 16 parallel adders, 1 subtractor and 2 max4 blocks would be enough to make all the calculations.

The vector decoded data is produced by comparing the values $LL(u_k)$ with 0. As it was explained in Section 4.2.6, if $LL(u_k)$ is larger than zero, *DecDat*(*k*) is 1, otherwise 0. This comparison and the formation of the vector *DecDat* is completed in a single clock cycle. To achieve this, *N* comparators are used. Similarly, *DecDatP* is calculated in a single clock cycle using the same comparators. We can conclude that, the decoding time of *N* data words and *N* parity words takes a total of 3*N*+8 clock cycles. Table 5.1 shows the operation time required for each part of the circuit.

There are some ways for further decreasing the operation time. One of them is pipelining. When the values alpha and beta are being calculated recursively, after a certain time (when half the alpha and beta values are calculated), there is no need to wait until the finalization of alpha and beta calculations. The *LL* calculations can start immediately. This saves a clock time of 3*N*/2. This one and similar advanced techniques that will decrease the calculation time are left as future work..

Table 5.1: The operation time required for each part of the circuit for N data words and N parity words

Operation	**Required Time** (clock cycles)
BM matrix calculation	3
Alpha-Beta and Nonterm calculations	3*N*-3
LL and *LLp* calculations	6
Decoded data and decoded parity calculations	2
Total	**3*N*+8**

5.2 Data Storage in the MAP Decoder

In each decoder, received data may be kept in a RAM or the data can be read directly into registers. Received data is in the form of vectors of length 2*N*.

Hence, to store the received data, a block RAM of 2*N* words or 2*N* registers are needed. In the below calculations, it is assumed that the received data is kept in registers. The length of the memory words and the size of the registers depend on the length of the received data. If the received data has *r* bits, then all the memory elements should store *r* bits. If the data is to be stored on a block RAM of the FPGA, it must be noted that the Xilinx IP core "Single-Port Block Memory v5.0" cannot store memory words longer than 256 bits.

For recursive calculations, many registers are used. From equation (4.28), it is seen that the matrix *BM* is composed of 8*N* words of *r* bits. Hence, the matrix *BM* is stored in 8*N* registers of length *r* bits.

A priori information for the data words is stored in the vector *p*. From equation (4.30), this vector has 2*N* words of *r* bits. Since each $p_{k0} = - p_{k1}$, *N* registers are required to store the a priori information data.

If the a priori information is not used in the decoder which is the case in the column decoders, then some simplifications can be made to decrease the number of registers. For this decoder, each line of the matrix *BM* has 4 entries because the remaining 4 entries are equal to former 4 entries. Hence, instead of 8*N* registers, 4*N* registers are enough. Also in this decoder, there is no need to use *N* registers to keep a priori information. As a result, if the a priori information is not used, then the number of registers will be decreased by 5*N*.

Section 4.2.3 tells that the elements of alpha are kept in the matrix alpha. This matrix consists of 4 vectors. Each vector has *N* words. To store the matrix alpha, 4*N* registers are required. Similarly, the values of beta are stored in the matrix beta of size 4*N*. To store the matrix beta, 4*N* registers of size *r* bits are required.

In *LL* calculation, the calculations are made according to the equations (4.40) and (4.41). For *sum0* and *sum1* calculations, a total of 2*N* registers are used. The values *LL* are stored into *N* registers. A total of 3*N* registers are required. So, for *LL* and *LLp*, a total of 6*N* registers of *r* bits are used. It is also possible to write the difference of *sum0* and *sum1* directly into a register. In this case, no registers are required to store *sum0* and *sum1*. This means, instead of 6*N* registers, 2*N* registers are enough. But, because of the implementation difficulties, 6*N* registers are used in this study.

As we explained in Section 4.2.3, the decoded data is kept in the vector *DecDat*. This vector has *N* words. That is, *N* registers of size 1 bit are required. Similarly, *N* registers are needed for the vector *DecDatP*. This means, for decoded data, a total of 2*N* registers of size 1 bit are required.

Totally, a single map decoder uses 25*N* registers of size *r* bits and 2*N* registers of 1 bit. Table 5.2 shows the number of registers required for storing signals. Other logic components that are used in the implementation of the decoder are summarized in Table 5.3.

Table 5.2: The number of registers required for storing signals

Stored signals	**Number of 14-bit registers**
Received data	2*N*
Branch-metric matrix	8*N*
A priori information	*N*
Alpha matrix	4*N*
Beta matrix	4*N*
LL vector	3*N*
LLp vector	3*N*
Total	**25*N***

Table 5.3: The logic components that are used in the implementation of the decoder

Stored signals	Number of logic components
Branch-metric matrix	16*N* adders 16*N* multipliers
Alpha matrix	8 adders 4 max2 blocks
Beta matrix	8 adders 4 max2 blocks
LL vector	16*N* adders 4*N* max2 blocks *N* subtractors
DecDat vector	*N* comparators

5.3 Minimization of Time and Area

The MAP decoder works on matrices and vectors. It makes many recursive calculations. Hence, many operations are repeated several times for a large number of signals. It is possible to use the same circuit components for different signals in different clock cycles. For example, in the calculation of the vector *LL*, the same operations are repeated for all the vector elements. As the operations are the same, the arithmetic and logic blocks may be shared between the operations. This minimizes the area. But, in this case, simultaneous operation of the components is not possible. All the elements of the vector cannot be calculated at the same time. As the arithmetic and logic blocks are shared, circuit must wait until the required component is idle. This

means that the speed is degraded. On the other hand, when separate components are used for the signals that can be calculated at the same time, the time required for the operation is minimized. But, for parallel processing, each signal needs separate arithmetic and logic blocks. This approach increases the number of circuit components but decreases the operating time of a single MAP decoder.

For example, when the vector *LL* is calculated in parellel, 16*N* adders, 4*N* max2 blocks and *N* subtractors are required. The calculation time takes 3 clock cycles. On the other hand, when the calculation is made serially, 16 adders, 4 max2 blocks and 1 subtractor is used. This time, the operation takes 3*N* clock cycles. The rise in decoding time affects the decoding performance significantly. The serial calculation results in a decoding rate of 32.6 Mbps. With the parallel calculation, the decoding performance is calculated as 19.7 Mbps.

When many decoders are placed on a FPGA for parallel processing, the relation between the area and the speed becomes more complex. As we want to reach high speeds, we want to minimize the operation time of a single MAP decoder. But, the operation time of a decoder can only be minimized by consuming large areas. This prevents us from employing maximum number of decoders in parallel. So, the overall speed is degraded.

If we minimize the area of a single decoder, then it is possible to maximize the number of decoders working in parallel. But, this time, operation time of a single decoder becomes very long. This also degrades the overall speed. Hence, when designing the decoder, both speed and area limitations must be considered.

In this study we tried to achieve maximum speed in a reasonable area. Hence, for the calculations that occupy large area and have short operation times, we

tried to use the same components for many operations. For example, *LL* and *LLp* calculations need large area, so they are calculated using the same logic components. On the other hand, for the calculations that occupy small area, we tried to use separate logic components for each calculation. For example, *DecDat* vector is calculated by using *N* comparators in parallel. This way, parallel or serial operation is selected looking at the time and area needs of the calculation.

5.4 Convolutional Product Decoder Performance

In convolutional product decoder, a single block RAM keeps the received data information. The decoders that are used in the convolutional product decoder have no RAMs. Data is read directly from the block RAM into registers.

This RAM has $2N$ lines of length $2N$. Each word is accepted to have r bits. Hence, to store the received data, a block RAM of size $r \times 4N^2$ is needed. A priori information is stored in the matrix p. As it was explained in Section 4.2, this matrix has a size of $N \times 2N$. Each word in the matrix has r bits. Hence, $2N^2$ registers of r bits are required.

Section 4.2 tells that each of the matrices y_r and o_r has $2N^2$ words. Similarly, for each one, $2N^2$ registers of r bits are required. At the beginning, data is read from the RAM into registers in $4N^2$ clock cycles.

The column decoder is composed of $2N$ MAP decoders. The operation time of the column decoder is equal to that of a single MAP decoder. This is equal to $3N+8$ clock cycles. After that, $2N^2$ subtractors are used in parallel to make the following subtraction:

$$y_r = y_r - p. \tag{5.2}$$

The row decoder consists of N MAP decoders. The operation time of the column decoder is equal to that of a single MAP decoder. This is followed by the following subtraction:

$$p = o_r - y_{r.} \tag{5.3}$$

$2N^2$ subtractors are used in parallel to make the calculation given in equation (5.3). This parallel subtraction takes a single clock cycle.

Hence, a decoding cycle takes a total of $4N^2+6N+18$ clock cycles. When multiple iterations are desired, there is no need to read the data from the RAM again, because the data is already transferred to registers. For every iteration, the parallel decoders perform their operations over and over. Hence, for m iterations, the operation time can be given as $4N^2+m(6N+18)$.

5.5 Area Limitations

In this thesis, the VHDL code is first written for N=5 and r=14. The circuit is implemented and its operation is verified with tests. It is seen that the generated logic circuit occupies 2% of the FPGA area. It consumes 278 slices out of 13696.

We can assume that all the register sizes change linearly with r. It can be assumed that the size of adders and multipliers also change nearly-linearly with r. Hence the area consumption can be assumed to be directly proportional to r. To verify this fact, the same decoder is designed for r=4. This circuit occupies 0.6% of the FPGA area. It consumes 89 slices out of 13696. The size of this decoder verifies the linear relationship prediction. Looking at these facts, we

can conclude that it is possible to decrease the decoder size in a nearly linear manner by selecting small values of *r*.

As a result of the recursive structure of the MAP algorithm, the number of arithmetic operations performed in the circuit is linearly proportional to the number of the received data words *N*. In addition to that, Section 5.2 tells that the number of registers in the circuit is also linearly proportional on *N*. In the light of these, we can conclude that the occupied area is linearly proportional to the number of received data words N and the number of bits in each word *r*.

In convolutional product decoder test, the circuit is implemented for a 10×10 received data vector which corresponds to *N*=5 and data words of 14 bits. After the implementation, it is seen that the circuit occupies 22% of the FPGA area. It uses 3018 out of 13696 slices.

Section 5.4 tells that in parallel processing, the number of registers that are used to keep p, o_r and y_r matrices are linearly proportional to N^2. But, the number of MAP decoders is linearly proportional with *N*. The area occupied by the registers that keep p, $o_{r,}$ and y_r matrices are negligible compared to the area occupied by the decoders. Hence, it can be accepted that the occupied area is proportional to N^2. With the same algorithm, it is possible to reach higher speeds by decreasing *N* and *r*.

When *N*=5 and *r*=14 are used in the design, a single MAP decoder reaches a decoding rate of 32.6 Mbps on a Virtex2Pro xc2vp30. The Rocket I/O technology of the FPGA enables the high speed serial interfaces. Hence, these decoding rates can be reached without I/O limitations. In the light of the operation time calculation given in Section 5.1, the performance of the decoder can be calculated for different values of *N*. It must be noted that *r* does not affect the decoding speed of the decoder directly. But it increases the area

required to implement the decoder. Table 5.4 gives the decoding rate of a single MAP decoder for different values of *N*.

Table 5.4: The decoding rate of a single MAP decoder for different value of *N*

N	**Decoding rate (Mbps)**
5	32.6
10	39.5
25	45.2
50	47.5
100	48.7

The convolutional product decoder is designed using the above decoders in parallel. With N=5 and r=14, the data rate is found to be 19.1 Mbps for 2 iterations. Following the explanations in Section 5.4, the decoding performance of the convolutional product decoder can be calculated for different values of *N*. Table 5.5 shows the decoding performance of the convolutional product decoder for different values of *N* and different number of iterations.

Table 5.5: The decoding performance of the parallel decoder for different values of *N* and different number of iterations

N	**Decoding Rate for 2 Iterations (Mbps)**	**Decoding Rate for 3 Iterations (Mbps)**
5	19.1	15.4
10	27.0	23.7
25	33.1	31.2
50	35.3	34.2

It can be seen on Table 5.5 that the decoding rate increases with the increasing values of *N*. This follows from the fact that, when *N* increases, the number of decoders working in parallel also increases. It must be noted that, again *r* has not a direct effect on the decoding rate. But it affects the area that the decoder occupies. When *r* is selected as 14, the maximum value that *N* can take is 250 for a single MAP decoder. The decoders for larger values of *N* cannot fit on the surface of the FPGA Virtex2Pro. But, when *r* is selected as 4, *N* can have values up to 900. Hence, when *r* is selected as 4, the maximum speed achievable on the FPGA is 37.4 Mbps for 2 iterations. Also, it must be noted that as larger values of *N* are selected, the $4N^2$ clock cycles that is spent for reading the data from the RAM becomes dominant over the total operation time and the number of iterations does not affect the operation time drastically. In conclusion, it can be said that high decoding rates are achievable by selecting high values of *N* and small values of *r*.

By using a two level buffer strategy, it is possible to further increase the decoding rate. This time, we don't need a RAM to store the channel information. This information is stored directly into buffers. When one level

of buffers is full, then the other level accepts the channel information. In this case, some area is consumed for the implementation of the input registers, but decoding rate is improved significantly. With this strategy, for N=25 and for 5 iterations, it is possible to reach a decoding speed of 111 Mbps.

CHAPTER 6

SUMMARY AND CONCLUSIONS

Today, in many fields of technology, high speed and accurate data storage and transmission is required. Usually, the main difficulty that we face at high data rates is the errors in the received signal. Error correcting codes are used for detecting and correcting the errors. Turbo coding is an efficient error correction method and it is commonly used in wireless systems. Maximum a posteriori (MAP) decoding of convolutional codes received large interest since the discovery of turbo codes.

In this thesis, MAP algorithm is implemented on an FPGA. The decoders are designed using the VHDL language. First, a MAP decoder is implemented and tested. Then, a convolutional product decoder is designed using the MAP decoders in parallel. The outputs of decoders were compared with the theoretically calculated output. We further ran simulations to compare the

performance of the designed decoders to the performance of decoders implemented in software with floating point numbers. All these studies revealed the proper operation of our design.

With the basic design values N=5 and r=14, a single MAP decoder reaches a decoding rate of 32.6 Mbps. In the light of the operation time calculation given in Section 5.1, the performance of the decoder can be calculated for different values of N. It must be noted that r does not affect the decoding speed of the decoder directly, but it increases the area required to implement the decoder. The convolutional product decoder is designed using the above decoders in parallel. With N=5 and r=14, the data rate is found to be 19.1 Mbps for 2 iterations. The decoding rate increases with the increasing values of N. This follows from the fact that, when N increases, the number of decoders working in parallel also increases.

When r is selected as 14, the maximum value that N can take is 250. The decoders for larger values of N cannot fit on the surface of the FPGA Virtex2Pro. But, when r is selected as 4, N can have values up to 900. Hence, when r is selected as 4, the maximum speed achievable on the FPGA is 37.4 Mbps for 2 iterations. Also, it must be noted that as larger values of N are selected, the $4N^2$ clock cycles that is spent for reading the data from the RAM becomes dominant over the total operation time and the number of iterations does not affect the decoding rate drastically. In conclusion, it can be said that high decoding rates are achievable by selecting high values of N and small values of r.

To observe the performance of the decoders under different noise conditions, Bit Error Rate (BER) tests are performed. The single MAP decoder and the convolutional product decoder are tested with 6 different noise levels and with 4 different quantization cases. These quantization cases use different number

of bits in the recursive calculations. In the tests, quantization is made with 14, 11, 9 and 4 bits. The noise level of AWGN channel is adjusted by using different values of Eb/ N0. The tests show that the quantization with 14 bits give results very close to the floating point simulations. It is also seen that the other quantization cases also give results very close to the floating point simulations.

As a conclusion, in this thesis, MAP algorithm is successfully transferred to a logic circuit. A decoder circuit is produced and using these decoders, convolutional product decoder is implemented and tested. The limitations in the hardware world are investigated. The trade offs between the resources are explained.

Further research can be carried on the minimization of the operation time. Some calculations may be forced to be simultaneous with the introduction of new registers and arithmetic units. Some new techniques like pipelining may be used. For the convolutional product decoder, more efficient data reading techniques may be investigated, e.g., more than one data may be stored in a RAM word.

APPENDIX A

TEST RESULTS OF THE MAP DECODER

To verify the operation of the MAP decoder, a data sequence "10101" is used. First, this data is encoded using BPSK ($2u$-1) mapping. This gives the encoded sequence "1101100111". Then, (σ^2=0.2) noise is added to this sequence. Hence, the following input sequence is obtained:

0.673421 1.43184 -1.47894 1.64453 0.495061 -0.481105 -1.70511 1.05269 1.38844 1.16594

Finally, this sequence is given to the decoder. The obtained results are compared with theoretical results. The table A.1 summarizes the comparison of the decoder results with the theoretical ones. The results show that hardware implementation of the MAP decoded the received data sequence successfully.

Table A.1: The comparison of the decoder results with the theoretical ones

	Decoder Test Results	**Theoretical Results**
Alfa0	0 -21.5 -37.5 -36.5 -29.5	0 -21.05 -37.50 -36.04 -26.81
Alfa1	-800 800 -31.5 -10.5 -1.5	-800 -800 -31.23 -9.76 0
Alfa2	-800 -0.5 -36.5 -36.5 -21.5	-800 0 -35.84 -36.18 -20.29
Alfa3	-800 -800 -0.5 -0.5 -29.5	-800 -800 0 0 -27.58
Beta0	-26 -33.5 -36.5 -26.5 0	-24.55 -32.39 -36.07 -25.54 0
Beta1	-27 -33.5 -44.5 -0.5 -800	-26.21 -32.53 -42.59 -1.38 -800
Beta2	-1 -10.5 -28.5 -800 -800	0 -9.76 -27.58 -811.53 -800
Beta3	-32 -0.5 -0.5 -800 -800	-30.92 0 -1.88 -811.53 -800
LL	46 -46 45.5 -54 54	45.60 -45.60 45.60 -52.36 52.36
Decoded Data	10101	10101

APPENDIX B

TEST RESULTS OF THE CONVOLUTIONAL PRODUCT DECODER

In the convolutional product decoder test, first a 5×5 data matrix is selected. The data matrix selected for the test is given in Table A:2.

Table A.2: The data matrix selected for the convolutional product decoder test

1	1	1	1	1
0	0	0	1	0
1	0	0	1	1
1	0	0	0	0
0	0	0	1	1

This matrix is encoded according to the procedure given in Section 4.3. Then BPSK ($2u$-1) modulation is employed. After that, (σ^2=0.2) noise is added to this sequence. The 10×10 received data matrix which is obtained after these steps is given in Table A.3.

Table A.3: The encoded data matrix that is used in the convolutional product decoder test

0.982	0.943	0.242	-1.881	1.160	-0.507	1.110	0.704	-0.149	-2.786
0.702	0.535	-0.327	-1.689	1.428	1.297	1.792	-0.220	0.087	-0.305
0.421	-1.628	-0.250	0.302	-1.946	-0.234	1.383	0.817	1.237	0.674
0.082	-0.136	2.001	-1.083	0.888	0.757	-0.063	-1.902	0.925	1.499
-0.157	2.232	-0.102	2.074	0.662	1.281	0.406	-0.518	0.096	0.888
-2.002	-1.228	1.060	1.786	2.080	-1.831	1.983	0.478	-2.023	-0.043
0.446	1.315	-2.000	1.800	-1.200	0.184	-1.667	-0.980	-1.828	1.838
-1.681	0.439	-1.281	-2.137	-1.219	-0.687	-0.955	1.806	1.072	1.027
0.369	-0.395	-0.797	-2.337	-0.828	-1.351	1.320	-0.011	-1.397	-1.343
-0.511	0.308	1.919	-0.552	0.750	0.90	0.124	-0.131	0.824	-0.131

After the first iteration, that is, after the first decoding cycle, the obtained matrix o_r is shown in Table A.4. In the table, the dark cells contain the values of *LL* for data bits and the white cells contain the values of *LL* for parity bits. The decoded data is given in Table A.5. In this table, it is possible to see that after the first iteration, 24 of the 25 data are decoded correctly and one of the data is erronous.

Table A.4: The matrix o_r after the first iteration

11.7	11.7	11.6	-15.3	10.9	10.8	10.1	10.2	7.5	-8.3
-5.4	-5.4	-5.3	-8.5	-7.1	-7.3	3.9	3.5	-2.1	2.7
6.4	6.4	-6.4	10.9	-7.9	7.8	3.1	3.2	2.5	4.2
13.4	13.5	-13.6	15.8	-11.2	11.3	-10.2	-10.2	-9.1	-9.2
-3.1	-3.1	-3.1	-8.5	-5.4	-5.4	1.4	0.8	-1.3	-0.8

Table A.5: The decoded data after the first iteration

1	1	1	1	1
0	0	0	1	0
1	0	0	1	1
1	0	0	0	0
0	0	0	1	0

After the second iteration, the matrix o_r is given in Table A.6. The corresponding decoded data is given in Table A.7. After the second iteration, all the data are decoded correctly. The reliability of the decoder can be further increased by increasing the number of iterations. In our test, 2 iterations were enough to correctly decode the received data.

Table A.6: The matrix o_r after the second iteration

22.8	22.8	22.7	-28.3	23.4	23.5	12.1	12.1	8.3	-8.2
-22.3	-22.3	-22.5	-27.4	-19.5	-19.5	8.3	8.3	-7.4	7.8
15.8	15.8	-15.4	20.1	-14.2	14.2	3.1	3.1	4.2	3.5
16.1	16.1	-16.1	18.3	-12.8	-12.8	-8.2	-8.3	-8.9	16.9
-4.7	-4.7	-4.7	-7.5	-6.8	-6.8	2.1	1.8	0.5	-1.5

Table A.7: The decoded data after the second iteration

1	1	1	1	1
0	0	0	1	0
1	0	0	1	1
1	0	0	0	0
0	0	0	1	1

APPENDIX C

THE PROPERTIES OF THE FPGA VIRTEX-II PRO

The FPGA Virtex-II Pro can be used in many applications such as

- optical networking,
- wireless infrastructure,
- storage area Networks(SANs),
- industrial control and image processing.

The FPGA family Virtex-II Pro depends on the Virtex-II architecture and has two new structures that enhance the application capabilities of this family. These new structures are:

- Rocket I/O™ Multi-Gigabit Transceivers (MGT) with transfer rates from 622Mb/s to 3.125 Gb/s.
- IBM PowerPC™ Embedded Processor Cores with performance of 300+ MHz core frequency.

With its Rocket I/O MGT technology, it is suitable for applications that require fast serial data transfer from chip-to-chip, across a backplane, and to an optical transponder.

With its Embedded PowerPC core, it transfers complex embedded system designs to FPGA logic or PowerPC core.

In addition to these new structures, the family Virtex-II Pro uses the new structures that are introduced in the family Virtex-II. These structures are given as follows:

- Digitally Controlled Impedance (DCI) technology for signal integrity management,
- Digital Clock Managers (DCM) for creating advanced clocking domains,
- Dedicated XtremeDSP Multipliers for high-performance DSP applications,
- System I/O technology for support of multiple different single-ended and differential I/O standards.

The Family Virtex-II Pro is composed of 10 members. Each member has logic cells (between 3000 and 125000), high capacity block RAM's, Digital Clock Managers (between 4 and 12), Rocket I/O MGT blocks (between 0 and 24),

and PowerPC processor cores (between 0 and 4). Table A.9 summarizes the properties of the members of this family.

Table A.8: The properties of the members of the family Virtex-II Pro.

Device	RocketIO Transceiver Blocks	PowerPC Processor Blocks	Logic Cells(1)	CLB (1 = 4 slices = max 128 bits)		18 X 18 Bit Multiplier Blocks	Block SelectRAM+		DCMs	Maximum User I/O Pads
				Slices	Max Distr RAM (Kb)		18 Kb Blocks	Max Block RAM (Kb)		
XC2VP2	4	0	3,168	1,408	44	12	12	216	4	204
XC2VP4	4	1	6,768	3,008	94	28	28	504	4	348
XC2VP7	8	1	11,088	4,928	154	44	44	792	4	396
XC2VP20	8	2	20,880	9,280	290	88	88	1,584	8	564
XC2VP30	8	2	30,816	13,696	428	136	136	2,448	8	644
XC2VP40	0(2) or 12	2	43,632	19,392	606	192	192	3,456	8	804
XC2VP50	0(2) or 16	2	53,136	23,616	738	232	232	4,176	8	852
XC2VP70	16 or 20	2	74,448	33,088	1,034	328	328	5,904	8	996
XC2VP100	0(2) or 20	2	99,216	44,096	1,378	444	444	7,992	12	1,164
XC2VP125	0(2), 20, or 24	4	125,136	55,616	1,738	556	556	10,008	12	1,200

The Rocket I/O technology enables the high speed serial interfaces. The serial I/O standards supported by Virtex-II Pro Rocket I/O technology are summarized in Table A.10.

Table A.9: The serial I/O standards supported by Virtex-II Pro Rocket I/O technology.

Serial Standard	Data Rate Per Channel	Baud Rate Per Channel
InfiniBand	2.0 Gbps	2.5 Gbps
1 Gb Ethernet 1000 Base-CX/SX/LX	1.0 Gbps	1.25 Gbps
10 Gb Ethernet (XAUI)	2.5 Gbps	3.125 Gbps
Fibre Channel	0.85/1.7 Gbps	1.06/2.12 Gbps
Serial ATA	1.2 Gbps	1.5 Gbps
Serial RapidIO	2.5 Gbps	3.125 Gbps
PCI Express (3 GIO)	2.0 Gbps	2.5 Gbps

The properties of PowerPC 405 can be summarized as follows:

- Embedded 300+ MHz Harvard Architecture Block
- Low Power Consumption: 0.9 mW/MHz
- Five-Stage Data Path Pipeline
- Hardware Multiply/Divide Unit
- Thirty-Two 32-bit General Purpose Registers
- 16 KB Two-Way Set-Associative Instruction Cache
- 16 KB Two-Way Set-Associative Data Cache
- Memory Management Unit (MMU)
- Dedicated On-Chip Memory (OCM) Interface
- Supports IBM CoreConnect™ Bus Architecture
- Debug and Trace Support

The functional blocks of PowerPC 405 are

- Cache units,
- Memory Management unit,
- Fetch Decode unit,
- Execution unit,
- Timers,
- Debug logic unit.

BIBLIOGRAPHY

[1] C. Rrou, A. Glavieux, and P. Thitimajshima, “Near Shannon limit error-correcting coding and decoding: Turbo codes,” *Proc. IEEE Int. Conf. Communications,* pp. 1064-1070, 1993.

[2] L. R. Bahl, J. Cocke, E. Jelinek, and J. Raviv, “Optimal decoding of linear codes for minimising symbol error rate,” *IEEE Trans. ZnJ Theory,* pp. 284-287, 1974.

[3] P. Robertson, "Illuminating the structure of parallel concatenated recursive systematic (turbo) codes," *Proc., IEEE GLOBECOM,* 1994, pp.1298-1303.

[4] J. Fei, "On a turbo decoder design for low power dissipation," *PhD Thesis,* Virginia Polytechnic Inst. And State Univ., 2000.

[5] J. Hagenauer, "The turbo principle: Tutorial introduction and state of the art," *Proc., Int. Symp. on Turbo Codes and Related Topics,* (Brest, France), pp.1-l1, Sept.1997.

[6] S. Benedetto and G. Montorsi, "Unveiling turbo codes: some results on parallel concatenated codes," *IEEE Trans. Inform. Theory*, vol. 42, pp.409-429, Mar. 1996.

[7] S. Benedetto and G. Montorsi, "Generalized concatenated codes with interleavers," *Proc., Int. Symp. on Turbo Codes and Related Topics,* (Brest, France), pp. 32-39, Sept.1997.

[8] H. Koorapaty, Y. P. E. Wang, and K. Balachandran, "Performance of turbo codes with short frame sizes," *Proc., IEEE Veh. Tech. Conf.,* pp.329-333, 1997.

[9] S. Halter, M. Oberg, P. M. Chau, P. H. Siegel, "Reconfigurable signal processor for channel coding & decoding in low SNR wireless communications," *IEEE Workshop on Signal Processing Systems*, pp.260-274, 1998.

[10] S. Hong, J. Yi, W. E. Stark, "VLSI design and implementation of low-complexity adaptive turbo-code encoder and decoder for wireless mobile communication applications," *IEEE Workshop on Signal Processing Systems*, pp.233-242, 1998.

[11] S. S. Pietrobon, "Implementation and performance of a turbo/MAP decoder," *Int. J. Satell. Commun*., vol.16, pp.23-46, 1998.

[12] G. Masera, G. Piccinini, M. R. Roch, M. Zamboni, "VLSI architectures for turbo codes," *IEEE Trans. On VLSI systems*, vol. 7, No.3, Sept. 1999.

[13] Z. Blazek, V. Z. Bhargava, "A DSP-based implementation of a turbo-decoder," *Global Telecommunications Conference*, vol. 5, pp.2751-2755, 1998.

[14] P. Robertson, P. Hoeher, and E. Villebrun, "Optimal and sub-optimal maximum a posteriori algorithms suitable for turbo decoding," *European Trans. on Telecommun.,* vol.8, pp.119-125, Mar./Apr. 1997.

[15] P. Robertson, "Improving decoder and code structure of parallel concatenated recursive systematic (turbo) codes," *Proc., IEEE Int. Conf. on Universal Personal Communications*, pp. 183-187, 1994.

[16] William J. Ebel, "Turbo-codes: Algorithms and implementation," contract report submitted to Texas Instruments.

[17] A. J. Viterbi, "An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes," *IEEE J. Select. Areas Commun.,* vol. 16, pp.260-264, 1998.

[18] W. J. Gross, P. G. Gulak, "Simplified MAP algorithm suitable for implementation of turbo decoders," *Electronics Letters*, vol. 34, No. 16, pp.1577-1578, 1998.

[19] B. Sklar, "Turbo code concepts made easy, or how I learned to concatenate and reiterate," *Proc. MILCOM 97*, vol.1, pp.20-26, 1997.

[20] K. H. Tzou, J. G. Dunham, "Sliding block decoding of convolutional codes," *IEEE Trans. Commun.,* COM-29, pp.1401-1403,1981.

[21] S. Benedetto, G. Montorsi, D. Divsalar, F. Pollara, "Soft-output decoding algorithms in iterative decoding of turbo codes," *JPL TDA Prog. Rep. 42*, pp.63-87, 1996.

[22] J. Hagenauer, "Source-controlled channel decoding," *IEEE Tran. On Communications*, vol.43, No. 9, 1995.

[23] O. M. Collins, "The subtleties and intricacies of building a constraint length 15 convolutional decoder," *IEEE Trans. Commun.*, COM-40, pp.1810-1819, 1992.

[24] S. S. Pietrobon, J. J. Kasparian, P. K. Gray, "A multi-D trellis decoder for a 155 Mbit/s concatenated codec," *Int. J. Satell. Commun.,* vol.12, pp.539-553,1994.

[25] S. S. Pietrobon and S. A. Barbulescu, "A simplification of the modified Bahl decoding algorithm for systematic convolutional codes," *Int. Symp. On Information Theory and Its Applications*, Sydney, pp.1073-1077, 1994..

[26] X. Wang, S. B. Wicker, "A soft-output decoding algorithm for concatenated codes," *IEEE Trans. Info. Theory*. Pp.543-553, 1996.

[27] S. Lin, D. J. Costello, *Error Control Coding*, Pearson Prentice Hall, 2004

[28] J. Hagenauer, E. Offer and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Trans. Info. Theory*, vol. 42, No. 2, 1996

[29] S. Benedetto, D. Divsalar, G. Montorsi and F. Pollara, "Serially concatenation of interleaved codes: Design and performance analysis," *IEEE Trans. Info. Theory*, vol. 44, pp. 909-926, 1998.

[30] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal MAP algorithms operating in the log domain," *Proc. 1995 Int. Conf. on Commun.* pp. 1009-1013.

[31] J. G.Prokis, *Digital Communications*, McGraw-Hill, 1995.

[32] A. O. Yilmaz, "Performance of Turbo Codes: The Finite Length Case," *PhD Thesis*, Univ. Of Michigan, 2004.