國 立 交 通 大 學

電 信 工 程 學 系 碩 士 班

碩 士 論 文

具計算效率且節省記憶體的

第三代行動通訊(3GPP)渦輪碼解碼器

Calculation Efficient and Memory Saving

Turbo Decoder for 3GPP

研究生：鍾文狀

指導教授：紀翔峰　博士

中華民國九十三年八月

研 究 生：鍾文狀　　Student: Wen-Choung Chong

指導教授：紀翔峰 博士　　Advisor: Dr. Hsiang-Feng Chi

國立交通大學

電信工程學系碩士班

碩士論文

A Thesis

Submitted to Deparment of Communication Engineering

College of Electrical Engineering and Computer Science

National Chiao-Tung University

for the Degree of Master

in

Communication Engineering

August 2004

Hsinchu, Taiwan, Republic of China

中華民國九十三年八月

具計算效率且節省記憶體的第三代行動通訊(3GPP)渦輪碼解碼器

學生：鍾文狀　　　　　　　　　　　　　　指導教授：紀翔峰　博士

國立交通大學電信工程學系碩士班

摘　　要

目前的無線通訊系統中，資料量的傳輸需求愈來愈大，而在傳輸通道的非理想效應影響下經常使得傳輸資料出現錯誤。為了有效降低錯誤率，第三代行動通訊(3GPP，3GPP2…等)系統均採用了目前更正能力最強的渦輪碼。渦輪碼的硬體實現中最大的難題在於解碼時需要大量的記憶體及大量的運算。一般渦輪碼解碼器中所採用的節省記憶體架構(sliding window)雖可解決記憶體的問題，但同時也會導致更多的運算量。在節省記憶體及運算量的考量下，本論文的目的是以另一種節省記憶體的架構(halfway)實現一個和原始架構比起來可節省記憶體且不增加任何運算量的渦輪碼解碼器。在解碼上，我們採用 Max-Log-MAP algorithm 使得運算複雜度降低。在硬體上，我們使用了只包含一個 Max-Log-MAP 解碼器的硬體架構。

Calculation Efficient and Memory Saving Turbo Decoder for 3GPP

Student: Wen-Choung Chong Advisor: Dr. Hsiang-Feng Chi

Department of Communication Engineering
National Chiao Tung University

ABSTRACT

Turbo codes have become one of the necessary specifications for the state-of-the-art communication systems. The difficulties in implementing turbo decoder are the vast computational complexities and the request for a lot of memories. The most public method for decreasing the need of memories is sliding window method. But using sliding window method will increase the computational complexities. This thesis is purposed to propose a calculation efficient and memory saving turbo decoder. We use another memory saving algorithm – halfway algorithm, in our turbo decoder. This successfully decreases the computational complexities and the need of memory capacity. Besides, we adopt Max-Log-MAP algorithm in our design in order to simplify the hardware.

誌　　　謝

短短二年的研究生生涯即將在本論文的完成下告一段落，首先感謝指導教授紀翔峰老師在學業及生涯規劃上的指導，在學業上，老師根據實驗室同學各人的特質，因材施教，讓每位同學自己學得如何自己從無到有的求得知識，在生涯規劃上，老師也盡力地提供自己過去的生活經驗並分析給我們知道。

感謝實驗室同學賴昭宏、李昭宏、李佳勳，我們一起在老師的帶領下，把實驗室從無到有地建立起來，而在硬體、軟體實作及論文寫作時，每個人都幫我解決了不少問題，也提供了我很多意見。

感謝父母親及哥哥、姊姊讓我在衣食無憂的情況下，得以認真的在實驗室裡做研究，而哥哥有時也會在我的要求下在百忙之中來實驗室教導我們同學如何使用儀器及 FPGA，實在是麻煩他了。

最後要感謝的是我的女朋友，楊正豔，從大學以來陪我至今，在六年的求學過程中，她陪在我身邊，無論在學業或生活上遇到困難，她都給予我支持、鼓勵，因此我才能不畏艱難地考上研究所並完成論文寫作。

Content

List of Figures

List of Tables

Chapter 1
Introduction

In this chapter, we will introduce the basic elements of the digital communication system and the concept of channel coding in the beginning. Then the motivation and the objective of this thesis are presented. Finally we will introduce the organization of this thesis.

1.1 Digital communication system

The basic elements of a digital communication system are shown in Figure 1.1.

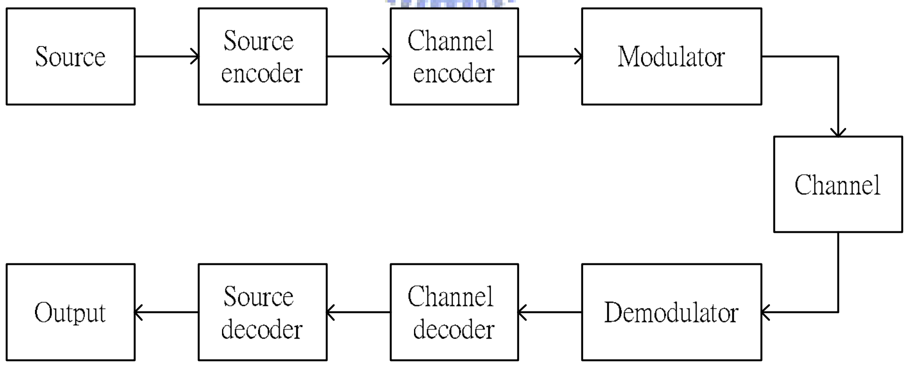


Figure 1.1: basic elements in digital communication system

The messages from the source are converted into a sequence of binary digits by source encoder. The process of efficiently converting the output of the source into a sequence of binary digits is called source encoding. Alternatively speaking, the source encoder compresses the data from source and result in little or no redundancy in the binary representations of the data. Then the sequence of binary digits from the source encoder is passed to the channel encoder. On the contrary, the channel encoder is to introduce some controlled redundant information in the binary information sequence. These added redundancies can help the receiver to overcome the noise and interference encountered in the transmission of the signal through the channel. In effect, redundancy in the information sequence aids the receiver in decoding the information sequence correctly. The main purpose of the modulator is to map the binary information sequence into signal waveforms. We can choose modulator according to different applications and different channels. Usually we use the additive white Gaussian noise channel to simulate the channel block because it can provide precise analyses.

At the receiving end of a digital communication system, the successive three blocks are used to recover the original signals from the noisy receiving sequence. The demodulator processes the noisy waveforms and reduces them to a sequence of numbers that represent estimates of the transmitted symbols. The channel decoder will use these numbers to reconstruct the original information sequence from knowledge of the channel encoder. The source decoder uncompresses the sequence from knowledge of the source encoder and attempts to reconstruct the original signals.

The subject of the channel encoder and channel decoder is called channel codes or error control codes. In this thesis, we focus on this subject, especially the hardware implementation of the channel decoder.

1.2 History of channel coding

The concept of channel coding came from the paper [1] which was published by Claude Shannon in 1948. Shannon's primary result in this area is called the channel capacity theorem or noisy channel coding theorem. This theorem states that there exist error control codes such that information can be transmitted across the channel at rates less than the channel capacity with arbitrarily low bit error rate. Unfortunately, Shannon did not show how to construct the codes which can achieve the channel capacity. Two categories of channel codes, block codes and convolutional codes, were developed and widely used in practical systems.

The first error correcting code was Hamming code [2], which can correct only one error. During the years from 1957 to1959, cyclic codes [3-5] were published in some reports by E. Prange. Cyclic codes led to the development of BCH codes and Reed-Solomon codes a few years later. In 1959 and 1960 [6-8], Bose and Ray-Chaudhuri and Hocquenghem discover the multiple error correcting codes which are later named as Bose-Chaudhuri-Hocquenghem (BCH) codes. Reed-Solomon codes were discovered in 1960 by Reed and Solomon [9] and they were closely related to BCH codes.

In 1955, the first convolutional forward error correction codes were discovered by Elias [10]. In 1961, Wozencraft and Reiffen proposed the sequential decoding algorithm [11, 12] and this decoding algorithm is fast but sub-optimum. In 1967, Viterbi proposed an optimum decoding algorithm [13] which was recognized by Forney [14] as maximum likelihood decoding algorithm in 1973.

In 1987, Ungerboeck proposed trellis coded modulation (TCM) [15, 16] which integrates forward error correcting codes and modulation. TCM can achieve significant coding gains over power and band-limited transmission media.

In 1993, turbo codes [17] were invented by C. Berrou, A. Glavieux and P. Thitimajshima. Turbo codes were a historic breakthrough because they help the communication systems achieve Shannon limit closer than other codes.

1.3 Background of Turbo codes

Since turbo codes were proposed by C. Berrou, A. Glavieux and P. Thitimajshima in 1993 [17], they have been widely studied and discussed. Till now they are known as the best forward error correcting codes. Due to turbo codes' outstanding error correcting performance and their ability to achieve the Shannon capacity limit by 0.7 dB [17], there are many researches on the realizations of turbo codes. Turbo codes outperformed all other known coding schemes. Recently turbo codes have been adopted in several standardized communication systems, such as the third-generation (3G) mobile communication standards: i.e. W-CDMA (Wideband Code Division Multiple Access) in the 3rd Generation Partnership Project (3GPP), cdma2000 in the 3rd Generation Partnership Project 2 (3GPP2), and TD-SCDMA (proposed by China and Japan).

1.4 Motivation and Goal

Turbo codes have become one of the necessary specifications for the state-of-the-art communication systems. How to efficiently realize the turbo decoder in the integrated circuit always cause much research attention.

The difficulties in designing turbo decoders come from the high computational complexity. The challenging tasks are how to reduce the hardware cost and power consumption, the word-length determination in the fixed-point arithmetic, and cost-effective memory allocation/partition. In this thesis, we aim at implementing the turbo decoder of 3GPP/W-CDMA on field-programmable gate arrays (FPGAs) with memory saving methods. We will use Max-Log-MAP algorithm to solve the difficulty of the computational complexity. The ultimate goal is to propose low complexity, calculation efficient and memory-saving architecture.

1.5 Thesis Outline

This thesis is organized into eight chapters and described as follow:

In chapter 2, we would have an overview of entire turbo code system. In chapter 3, we introduce several decoding algorithms, discuss, and compare four decoding methods, including three memory saving schemes. In chapter 4, the 3GPP turbo encoder and interleaver are described. The hardware design considerations are discussed in chapter 5. In chapter 6, we describe the hardware architecture in detail. The ASIC and FPGA implementation and verification processes are presented in chapter 7. The conclusion and the future works are presented in Chapter 8.

Chapter 2
Overview of Turbo Code System

Turbo codes use concatenated schemes with the interleavers/de-interleavers placed between the constituent encoders/decoders. The standard turbo encoder structure uses the recursive systematic convolutional codes and parallel concatenated convolutional codes. In order to achieve good BER performance, we need the decoding algorithms which can accept soft input and produce soft outputs and can work iteratively.

2.1 Concatenated Codes

Turbo codes are usually composed of several concatenated convolutional codes. There are two kinds of concatenated convolutional codes, one is parallel concatenated convolutional codes (PCCCs) and the other is serial concatenated convolutional codes (SCCCs). PCCCs are often constituted by two or more recursive systematic convolutional (RSC) encoders joined in parallel by one or more pseudo-random interleavers, furthermore, the encoders encode the same information bits besides the information bits are scrambled by the interleaver. SCCCs also use the constituent convolutional encoder and the interleavers as PCCCs but differ from their connection method. The encoders used in SCCCs are connected serially and inserted by the pseudo-random interleaver. Figure 2.1 shows the encoder diagram of PCCCs and SCCCs.

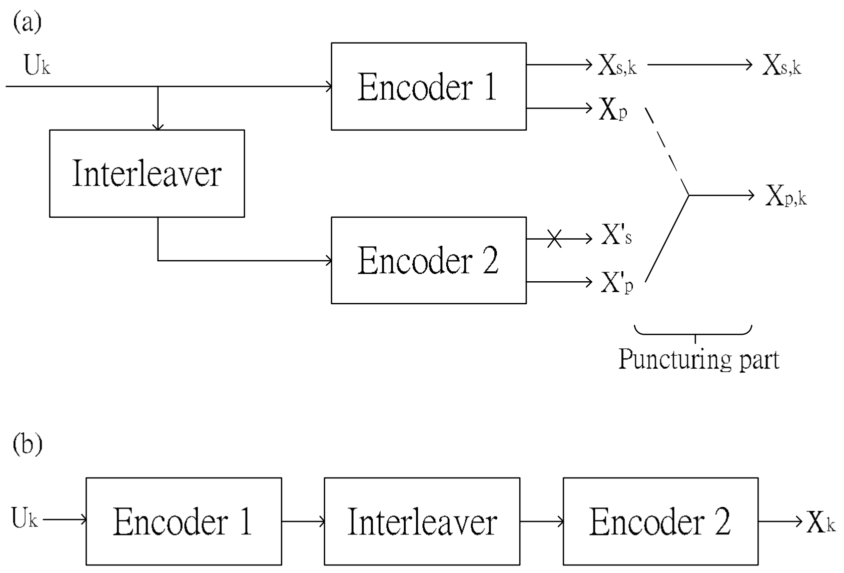


Figure 2.1: Turbo encoder diagrams of (a) PCCCs (b) SCCCs

The advantage of SCCCs is that: for a fixed frame size N, the slope of BER curve is inversely related to N^2 or N^3 but BER curve for PCCCs is only inversely related to N. Beside, SCCCs do not suffer from error floor but PCCCs do. The problem of error floor is caused by the poor interleaver design and truncation in the decoding procedure. But it was shown that both SCCCs and PCCCs could be designed without suffering from error floor no matter what BER requirement is [18].

Although SCCCs have the merits mentioned above, we often choose PCCCs in turbo code due to PCCCs' less computational complexity given the same constituent encoders and their better BER performance at low SNRs. Throughout the rest of this thesis, "turbo code" is referred to use PCCCs.

2.2 Recursive Systematic Convolutional (RSC) Encoder

Turbo codes use two or more RSC encoders as their component encoder. Although the encoders need not to be the same, we often use identical encoders in practice due to the low complexity of decoding. The term "recursive" means the encoder has a feedback loop; therefore, the output of this encoder is affected by the preceding output bit. And the term “systematic” means the encoder has one of its outputs identical to its input bit. Figure 2.2 shows the conventional convolutional encoder.

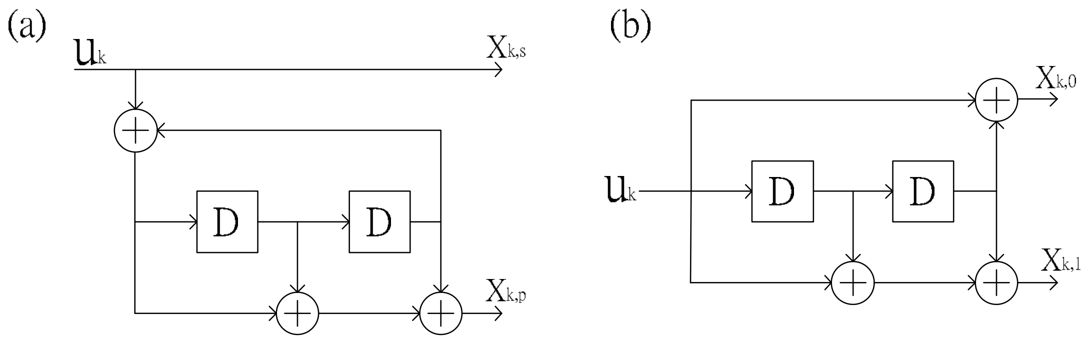


Figure 2.2: (a) RSC encoder with constraint length =3, generator matrix $G=[5,7]_{octal}$ (b)Non-recursive non-systematic encoder with constraint length =3, generator matrix $G=[5,7]_{octal}$

It can be proved that the recursive systematic convolutional code is code-equivalent to the non-systematic non-recursive convolutional code [19]. That is the sets of the codewords that they define are the same and for any codeword of the recursive systematic convolutional encoder, we can find the input stream for the non-systematic non-recursive convolutional code such that it produces the same codeword, vice versa. Although their codewords are identical, they behave differently. It is also shown that the RSC encoder tends to produce codewords with more weights than the code-equivalent non-recursive encoder [20]. This behavior causes the RSC encoder produce fewer codewords with lower weights and makes the error correcting performance better. This is the main reason to use the RSC encoders as turbo codes’ constituent encoders. Additionally, when we use the RSC encoder as constituent encoder, we only need to transmit the systematic output bits from the first one encoder because their systematic bits are alike except the order. Then the code rate of the encoder increases, bandwidth efficiency improves without degrading the performance since we still transmit all the information produced by the encoder.

2.3 Interleavers

The interleavers placed between the encoders are going to make the code more random in order to improve the burst error correction capability and they play a key rule in turbo code. What affect the interleaver are how random the interleaver is and how big the size of the interleaver is. As the size of the interleaver grows, the performance of the turbo code usually becomes better. But there is a tradeoff between the decoding latency and the BER performance. When the interleaver is more likely random, the performance of the turbo code also becomes better due to this kind of interleaver can make the correlation of the information bits decrease more. There are several kinds of interleavers, e.g. column-row interleaver, helical interleaver, odd-even interleaver, simile interleaver, frame interleaver, pseudo-random interleaver, S-type interleaver…etc. As long as we use the interleaver we proposed, the performance of the turbo code will suffer and we need to use different kind of interleaver according to the system requirement.

2.4 Decoders

Although the constituent encoders for turbo code belong to convolutional encoders, the decoding scheme for turbo codes is different from the pure convolutional decoding scheme. As mentioned above, turbo codes use the parallel concatenated encoding scheme. The turbo decoder would be constructed on the serial concatenated scheme because the performance of serial concatenated decoding scheme is better than that of parallel concatenated decoding scheme. The reason is the serial concatenated decoder will provide some extra information (or we call extrinsic information in turbo codes) to another decoder as its a-prior information. In turn, the latter decoder will also provide extra information to the former one. Contrarily the parallel concatenated decoders decode the information independently. Figure 2.3 shows the conventional turbo decoder's diagram.

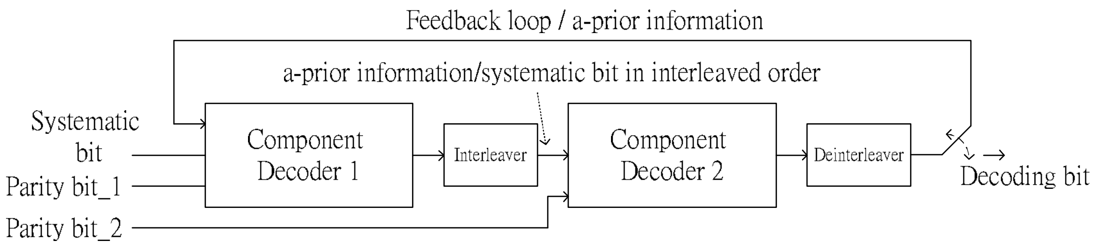


Figure 2.3: Conventional turbo decoder's diagram

Because each component decoder must provide the a-prior information to the other, they must have soft outputs. Since they have the soft inputs, we call them soft-input soft-output (SISO) decoders.

Chapter 3
Turbo Decoding

Nowadays we have two categories of algorithms to decode turbo codes, one originates from Maximum a posteriori (MAP) algorithm [21] proposed by Bahl et al. and another is Soft-Output Viterbi algorithm (SOVA) [22] proposed by Hagenauer and Hoeher.

Their evolutional histories are shown in Figure 3.1.

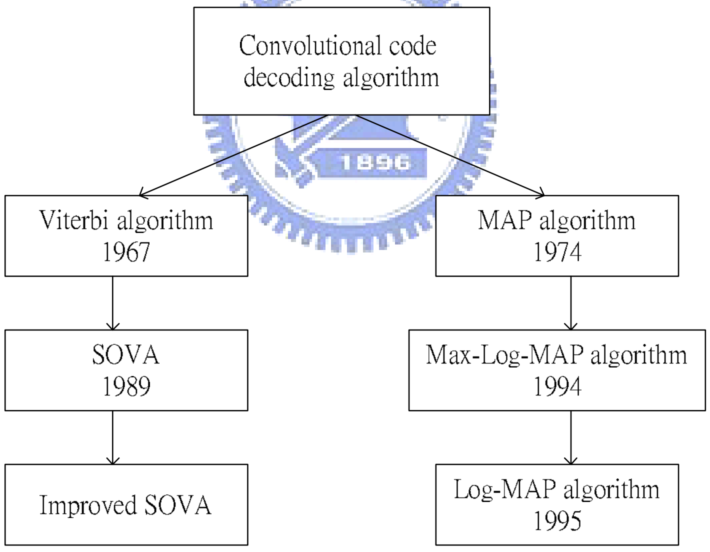


Figure 3.1: Evolution of soft-input soft-output (SISO) decoding algorithms

3.1 Decoding Algorithms

3.1.1 Maximum-a-posteriori (MAP) Algorithm

The Log Likelyhood Ratios (LLRs) L(u_k) of a data bit u_k is defined to be the log of the ratio of the probabilities of the bit taking its two possible values:

$$L(u_k) \triangleq \ln\left(\frac{P(u_k=+1)}{P(u_k=-1)}\right) \tag{1}$$

where $P(u_k=\pm 1)$ is the probability of the data bit u_k equals to ± 1.

After encoding the data bit u_k and transmitting the encoding bits through the channel and the matched filter, we received the sequence $\underline{y}$. Therefore we get the conditional LLR defined as:

$$L(u_k \mid \underline{y}) \triangleq \ln\left(\frac{P(u_k=+1 \mid \underline{y})}{P(u_k=-1 \mid \underline{y})}\right) \tag{2}$$

These conditional probabilities $P(u_k=\pm 1 \mid \underline{y})$ are the a-posteriori probabilities of the decoded bit u_k. The goal of the MAP algorithm is to estimate the decoded bit sequence and provide the probabilities of the correctness of every decoded bit given the received sequence $\underline{y}$ and it aims at minimizing the decoded bit error rate (BER). This means the MAP algorithm is correspondent with finding the a-posteriori LLR $L(u_k \mid \underline{y})$. By using Baye's rule and its derivation,

$$P(a \wedge b) = P(a|b)\cdot P(b) \tag{3}$$

$$P(\{a \wedge b\}|c) \equiv P(a|\{b \wedge c\})\cdot P(b|c) \tag{4}$$

the a-posteriori LLR $L(u_k \mid \underline{y})$ can be rewritten as:

$$L(u_k \mid \underline{y}) = \ln\left(\frac{P(u_k=+1 \wedge \underline{y})}{P(u_k=-1 \wedge \underline{y})}\right) \tag{5}$$

Figure 3.2 is the possible trellis for K=3 RSC code.

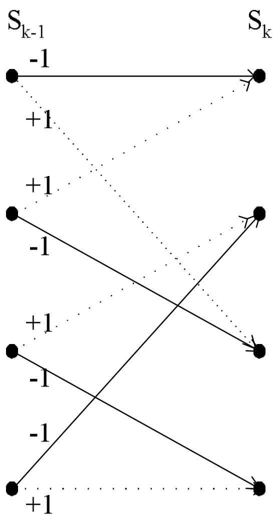


Figure 3.2: possible transitions in K=RSC code

If the previous state $S_{k-1}=s'$ and the present state $S_k=s$ are known then the input bit u_k will be known. The transitions which occur when $u_k=+1$ and those which occur when $u_k=-1$ are mutual exclusive so that the probability that any one of them occurs is equal to the sum of their individual probabilities. Equation (5) can be written as:

$$L(u_k\,|\,\underline{y}) \triangleq \ln\left(\frac{\sum\limits_{\substack{(s',s)\Rightarrow\\ u_k=+1}} P(S_{k-1}=s'\wedge S_k=s\wedge \underline{y})}{\sum\limits_{\substack{(s',s)\Rightarrow\\ u_k=-1}} P(S_{k-1}=s'\wedge S_k=s\wedge \underline{y})}\right) \tag{6}$$

Assume the channel is memoryless and using the Bayes' rule, we can write the individual probabilities $P(S_{k-1}=s'\wedge S_k=s\wedge \underline{y})$ from the numerator and denominator as:

$$\begin{aligned} P(s'\wedge s\wedge \underline{y}) &= P(s'\wedge s\wedge \underline{y}_{j<k}\wedge \underline{y}_k\wedge \underline{y}_{j>k}) \\ &= P(s'\wedge s\wedge \underline{y}_{j<k}\wedge \underline{y}_k)\cdot P(\underline{y}_{j>k}\,\big|\,s'\wedge s\wedge \underline{y}_{j<k}\wedge \underline{y}_k) \\ &= P(s'\wedge s\wedge \underline{y}_{j<k}\wedge \underline{y}_k)\cdot P(\underline{y}_{j>k}\,\big|\,s) \\ &= P(s'\wedge \underline{y}_{j<k})\cdot P(\{\underline{y}_k\wedge s\}\,\big|\,\{s'\wedge \underline{y}_{j<k}\})\cdot P(\underline{y}_{j>k}\,\big|\,s) \\ &= P(s'\wedge \underline{y}_{j<k})\cdot P(\{\underline{y}_k\wedge s\}\,\big|\,s')\cdot P(\underline{y}_{j>k}\,\big|\,s) \\ &= \alpha_{k-1}(s')\cdot \gamma_k(s',s)\cdot \beta_k(s) \end{aligned} \tag{7}$$

where $P(s' \wedge s \wedge \underline{y})$ represents $P(S_{k-1} = s' \wedge S_k = s \wedge \underline{y})$ for simplicity, and $\alpha_{k-1}(s')$, $\beta_k(s)$, $\gamma_k(s', s)$ are shown below:

$$\alpha_{k-1}(s') = P(S_{k-1} = s' \wedge \underline{y}_{j<k}) \tag{8}$$

$$\beta_k(s) = P(\underline{y}_{j>k} \mid S_k = s) \tag{9}$$

$$\gamma_k(s', s) = P(\{\underline{y}_k \wedge S_k = s\} \mid S_{k-1} = s'). \tag{10}$$

Using Bayes' rule and the assumption that channel is memoryless, $\alpha_k(s)$ can be written as:

$$\begin{aligned}
\alpha_k(s) &= P(S_k = s \wedge \underline{y}_{j<k+1}) \\
&= P(s \wedge \underline{y}_{j<k} \wedge \underline{y}_k) \\
&= \sum_{\text{all } s'} P(s' \wedge s \wedge \underline{y}_{j<k} \wedge \underline{y}_k) \\
&= \sum_{\text{all } s'} P(\{s \wedge \underline{y}_k\} \mid \{s' \wedge \underline{y}_{j<k}\}) \cdot P(s' \wedge \underline{y}_{j<k}) \\
&= \sum_{\text{all } s'} P(\{s \wedge \underline{y}_k\} \mid s') \cdot P(s' \wedge \underline{y}_{j<k}) \\
&= \sum_{\text{all } s'} \gamma_k(s', s) \cdot \alpha(s')
\end{aligned} \tag{11}$$

Assuming the trellis has the initial state $S_0 = 0$, the initial conditions for $\alpha_k(s)$ are:

$$\begin{aligned}
\alpha_0(S_0 = 0) &= 1 \\
\alpha_0(S_0 = s) &= 0 \quad \text{for all } s \neq 0
\end{aligned} \tag{12}$$

Similar to the derivation of $\alpha_k(s)$, $\beta_{k-1}(s')$ can also be written as:

$$\beta_{k-1}(s') = \sum_{\text{all } s} \beta_k(s) \cdot \gamma_k(s', s) \tag{13}$$

If the trellis is terminated in the all-zero state, the initial conditions for $\beta_k(s)$ are:

$$\begin{aligned}
\beta_N(0) &= 1 \\
\beta_N(s) &= 0 \quad s \neq 0
\end{aligned} \tag{14}$$

If the trellis is not terminated, then the initial conditions for $\beta_k(s)$ are:

$$\beta_N(s) = 1 \quad \text{for all } s \tag{15}$$

where N is the number of the stages in the trellis.

Thus, once the $\gamma_k(s', s)$ values are known, $\alpha_k(s)$ and $\beta_{k-1}(s')$ values can be calculated recursively.

Using the derivation from Bayes' rule, $\gamma_k(s',s)$ can be written as:

$$\begin{aligned}\gamma_k(s',s) &= P(\{\underline{y}_k \wedge s\} \mid s') \\ &= P(\underline{y}_k \mid \{s' \wedge s\}) \cdot P(s \mid s') \\ &= P(\underline{y}_k \mid \{s' \wedge s\}) \cdot P(u_k) \\ &= P(\underline{y}_k \mid \underline{x}_k) \cdot P(u_k)\end{aligned} \tag{16}$$

where u_k is the input bit which would cause the transition from state $S_{k-1} = s'$ to state $S_k = s$ and $\underline{x}_k$ is the corresponding transmitted codeword. $P(u_k)$ is the a-prior probability of this input bit u_k.

Assuming the channel is Gaussian and using BPSK modulation, $\gamma_k(s',s)$ can be written as:

$$\begin{aligned}\gamma_k(s',s) &= P(u_k) \cdot P(\underline{y}_k \mid \{s' \wedge s\}) \\ &= C \cdot e^{(u_k \cdot L(u_k)/2)} \cdot \exp(\frac{E_b}{2\sigma^2} \cdot 2a \cdot \sum_{l=1}^{n} x_{kl} y_{kl}) \\ &= C \cdot e^{(u_k \cdot L(u_k)/2)} \cdot \exp\left(\frac{L_c}{2} \sum_{l=1}^{n} x_{kl} y_{kl}\right)\end{aligned} \tag{17}$$

where C is the term does not depend on the sign of the bit u_k and the transmitted codeword $\underline{x}_k$, n is the number of the bits in codeword $\underline{x}_k$. L_c is called channel reliability value and defined as:

$$\mathrm{L_c} = \frac{E_b}{2\sigma^2} \cdot 4a \tag{18}$$

where E_b is the transmitted energy per bit, σ^2 is the noise variance and a is the fading amplitude($a = 1$ for non-fading AWGN channels).

Finally the a-posteriori LLR $L(u_k \mid \underline{y})$ in equation (6) can be rewritten as:

$$L(u_k \mid \underline{y}) = \ln\left(\frac{\sum_{\substack{(s',s)\Rightarrow \\ u_k=+1}} \alpha_{k-1} \cdot \gamma_k(s',s) \cdot \beta_k(s)}{\sum_{\substack{(s',s)\Rightarrow \\ u_k=-1}} \alpha_{k-1} \cdot \gamma_k(s',s) \cdot \beta_k(s)}\right) \tag{19}$$

This conditional LLR $L(u_k \mid \underline{y})$ is what MAP algorithm wants to get.

Because the turbo codes use RSC, we can separate $\gamma_k(s',s)$ into two parts. One has relationship with the systematic bit and the other does not. When we assume the systematic bit is the first bit of n transmitted bits, $x_{k1}=u_k$, we get:

$$\begin{aligned}\gamma_k(s',s) &= C\cdot e^{(u_k\cdot L(u_k)/2)}\cdot \exp\left(\frac{L_c}{2}\sum_{l=1}^{n}x_{kl}y_{kl}\right)\\ &= C\cdot e^{(u_k\cdot L(u_k)/2)}\cdot \exp\left(\frac{L_c}{2}u_k y_{ks}\right)\cdot \exp\left(\frac{L_c}{2}\sum_{l=2}^{n}x_{kl}y_{kl}\right)\\ &= C\cdot e^{(u_k\cdot L(u_k)/2)}\cdot \exp\left(\frac{L_c}{2}u_k y_{ks}\right)\cdot \chi_k(s',s)\end{aligned} \tag{20}$$

where $\chi_k(s',s)$ is the part uncorrelated with the systematic bit and it is shown below:

$$\chi_k(s',s)=\exp\left(\frac{L_c}{2}\sum_{l=2}^{n}x_{kl}y_{kl}\right) \tag{21}$$

Then we can separate the a-posteriori LLR $L(u_k|\underline{y})$ into three parts and rewrite it as follows:

$$\begin{aligned}L(u_k|\underline{y}) &= \ln\left(\frac{\sum\limits_{\substack{(s',s)\Rightarrow\\ u_k=+1}}\alpha_{k-1}\cdot e^{(+L(u_k)/2)}\cdot e^{(+L_c y_{ks}/2)}\cdot\chi_k(s',s)\cdot\beta_k(s)}{\sum\limits_{\substack{(s',s)\Rightarrow\\ u_k=-1}}\alpha_{k-1}\cdot e^{(-L(u_k)/2)}\cdot e^{(-L_c y_{ks})}\cdot\chi_k(s',s)\cdot\beta_k(s)}\right)\\ &= L(u_k)+L_c y_{ks}+\ln\left(\frac{\sum\limits_{\substack{(s',s)\Rightarrow\\ u_k=+1}}\alpha_{k-1}\cdot\chi_k(s',s)\cdot\beta_k(s)}{\sum\limits_{\substack{(s',s)\Rightarrow\\ u_k=-1}}\alpha_{k-1}\cdot\chi_k(s',s)\cdot\beta_k(s)}\right)\\ &= L(u_k)+L_c y_{ks}+L_e(u_k)\end{aligned} \tag{22}$$

where:

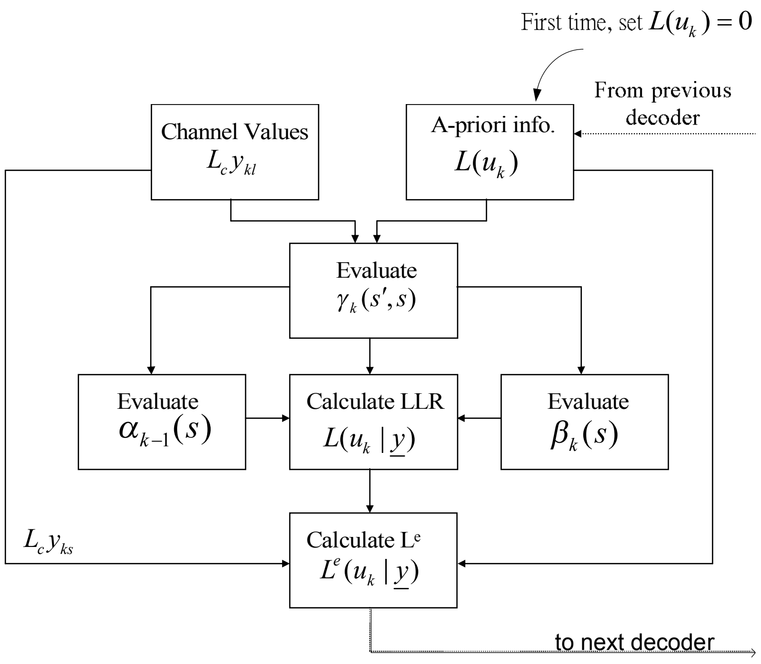
$$L_e(u_k)=\ln\left(\frac{\sum\limits_{\substack{(s',s)\Rightarrow\\ u_k=+1}}\alpha_{k-1}\cdot\chi_k(s',s)\cdot\beta_k(s)}{\sum\limits_{\substack{(s',s)\Rightarrow\\ u_k=-1}}\alpha_{k-1}\cdot\chi_k(s',s)\cdot\beta_k(s)}\right) \tag{23}$$

The first term $L(u_k)$ is the a-prior LLR which can be derived from $P(u_k)$ and it is usually unknown at the decoder. Because we usually assume $P(u_k = \pm 1) = 0.5$ at first time, the initial conditions of $L(u_k)$ are all zero in the logarithm domain. But when we use iterative turbo decoder, each component decoder can provide the other one with the a-prior LLRs.

The second term $L_c y_{ks}$ stands for the soft output of the channel when the input systematic bit u_k transmitted through the channel and received as y_{ks}. Because the channel reliability value L_c is directly relative to the channel SNR, the received systematic bit y_{ks} will have a large impact on the a-posteriori LLR $L(u_k|\underline{y})$ if the channel SNR is high and vice versa.

The third term $L_e(u_k)$ is referred to as the extrinsic LLR for the bit u_k because it uses the values of the branch transition probabilities $\gamma_k(s', s)$ for all the branches except for the k-th branch. Then it will be sent to the next decoder as the a-prior information.

The flowchart of all the operations involved in MAP algorithm and iterative decoding process is shown in Figure 3.3.



First time, set $L(u_k) = 0$
From previous decoder
Channel Values $L_c y_{kl}$
A-priori info. $L(u_k)$
Evaluate $\gamma_k(s', s)$
Evaluate $\alpha_{k-1}(s)$
Calculate LLR $L(u_k | \underline{y})$
Evaluate $\beta_k(s)$
$L_c y_{ks}$
Calculate L^e $L^e(u_k | \underline{y})$
to next decoder

Figure 3.3: MAP iterative decoding flow chart

The structure of the turbo decoder is shown in Figure 3.4. It is constituted by two component decoders, one interleaver and one deinterleaver and the decoders will work iteratively. Each component decoder has three inputs: 1. the systematic information 2. The parity information associated the component encoder and 3. The information provided by the other component decoder which was referred to as a-prior information.

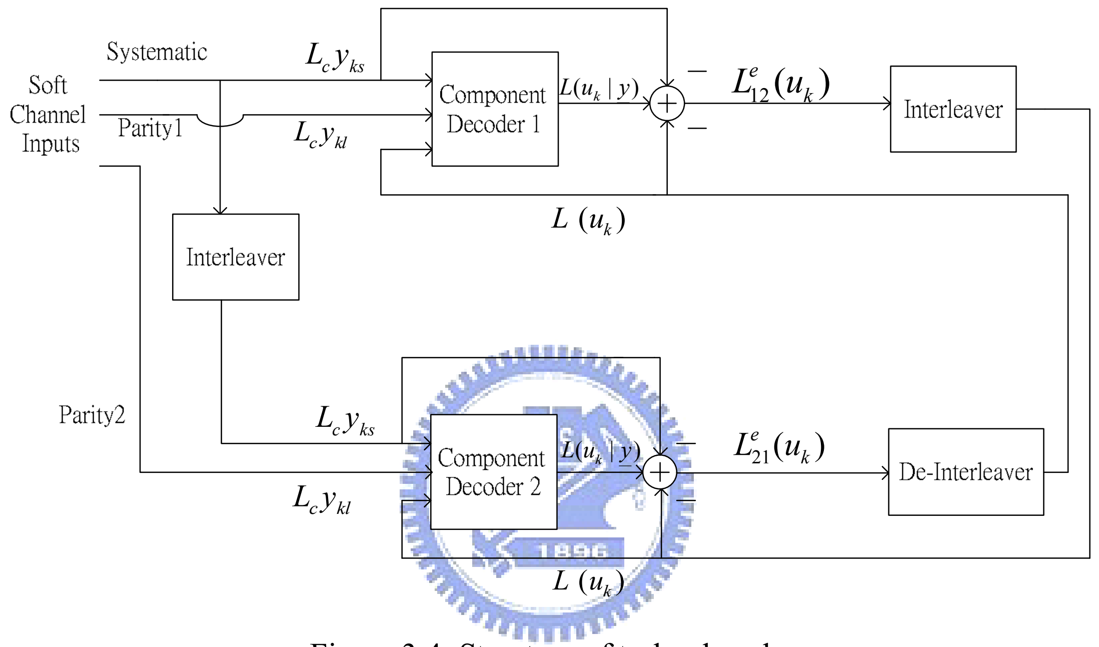


Figure 3.4: Structure of turbo decoder

We describe the iterative decoding process as follow:

Firstly the component decoder 1 takes the systematic bits in natural order and the parity bits transmitted by the encoder 1 as its input signals but take the a-prior information, which should get from component decoder 2, as 0 since the component decoder 2 does not take action. After finish the decoding of the decoder 1, the decoding result or the a-prior information should be transferred to the decoder 2 in interleaving order.

Secondly the decoder 2 takes the parity bits transmitted by the encoder 2, the systematic bits in interleaving order and the a-prior information provided by the decoder 1 in interleaving order as its input signals. When the decoder 2 finishes its decoding process, it also produces the a-prior information for the decoder 1 but in the interleaving order, then transferring the a-prior information with the aid of the de-interleaver to the decoder 1.

The first iteration completes after these steps and we can repeat again besides the decoder 1 has the a-prior information this time. Usually after 5 to 10 times iterations, the decoder will output the decoding results. Because the iterative decoding process is similar to the cyclic feedback mechanism of the turbo engine, we name the code “turbo code”.

3.1.2 Max-Log-MAP Algorithm

The Max-Log-MAP algorithm simplifies the calculations of $\alpha_k(s)$, $\beta_k(s)$ and $\gamma_k(s',s)$ which are needed by MAP algorithm by transferring these calculations into the log arithmetic domain and using the Jacobian logarithm approximation loosely:

$$\ln\left(\sum_i e^{x_i}\right) \approx \max_i(x_i) \tag{24}$$

where $\max_i(x_i)$ is the maximum value of x_i.

By defining $A_k(s)$, $B_k(s)$ and $\Gamma_k(s',s)$ as the logarithm of $\alpha_k(s)$, $\beta_k(s)$ and $\gamma_k(s',s)$, we can rewrite the equations as follows:

$$\begin{aligned} A_k(s) &\triangleq \ln\left(\alpha_k(s)\right) \\ &= \ln\left(\sum_{\text{all } s'} \alpha_{k-1}(s')\gamma_k(s',s)\right) \\ &= \ln\left(\sum_{\text{all } s'} \exp\left[A_{k-1}(s') + \Gamma_k(s',s)\right]\right) \\ &\approx \max_{s'}\left(A_{k-1}(s') + \Gamma_k(s',s)\right) \end{aligned} \tag{25}$$

$$\begin{aligned} B_{k-1}(s') &\triangleq \ln\left(\beta_{k-1}(s')\right) \\ &\approx \max_{s}\left(B_k(s) + \Gamma_k(s',s)\right) \end{aligned} \tag{26}$$

Equation (25) is calculated in a forward recursive manner and equation (26) is

$A_k(s) = \max_{s'}\left(A_k(s') + \Gamma_k(s', s)\right)$ calculated in a backward recursive manner but they are both equivalent to the recursion used in the Viterbi algorithm – for the merging paths the survivor is found by using additions and comparison. Then the new branch metric $\Gamma_k(s', s)$ can be written as:

$$\begin{aligned}
\Gamma_k(s', s) &\triangleq \ln\left(\gamma_k(s', s)\right) \\
&= \ln\left(C \cdot e^{(u_k \cdot L(u_k)/2)} \exp\left[\frac{E_b}{2\sigma^2} 2a \sum_{l=1}^{n} y_{kl} x_{kl}\right]\right) \\
&= \ln\left(C \cdot e^{(u_k \cdot L(u_k)/2)} \exp\left[\frac{L_c}{2} \sum_{l=1}^{n} y_{kl} x_{kl}\right]\right) \\
&= \hat{C} + \frac{1}{2} u_k \cdot L(u_k) + \frac{L_c}{2} \sum_{l=1}^{n} y_{kl} x_{kl}
\end{aligned} \tag{27}$$

where $\hat{C} = \ln(C)$ does not have any relationship with the data bit, u_k ,or the codeword, $\underline{x}_k$, and so can be considered a constant and ignored.

From equation (19), the a-posteriori LLRs $L(u_k | \underline{y})$ for Max-Log-MAP algorithm can be calculated as:

$$\begin{aligned}
L(u_k | \underline{y}) &= \ln\left(\frac{\sum\limits_{\substack{(s',s)\Rightarrow \\ u_k=+1}} \alpha_{k-1}(s') \cdot \gamma_k(s', s) \cdot \beta_k(s)}{\sum\limits_{\substack{(s',s)\Rightarrow \\ u_k=-1}} \alpha_{k-1}(s') \cdot \gamma_k(s', s) \cdot \beta_k(s)}\right) \\
&= \ln\left(\frac{\sum\limits_{\substack{(s',s)\Rightarrow \\ u_k=+1}} \exp\left(A_{k-1}(s') + \Gamma_k(s', s) + B_k(s)\right)}{\sum\limits_{\substack{(s',s)\Rightarrow \\ u_k=-1}} \exp\left(A_{k-1}(s') + \Gamma_k(s', s) + B_k(s)\right)}\right) \\
&\approx \max_{\substack{(s',s)\Rightarrow \\ u_k=+1}} \left(A_{k-1}(s') + \Gamma_k(s', s) + B_k(s)\right) \\
&\quad - \max_{\substack{(s',s)\Rightarrow \\ u_k=-1}} \left(A_{k-1}(s') + \Gamma_k(s', s) + B_k(s)\right)
\end{aligned} \tag{28}$$

The transitions from the trellis stage S_{k-1} to the stage S_k are grouped into two groups. One contains those might happen if $u_k = +1$ and the other contains those might happen if $u_k = -1$. In each group, we only want the maximum value of $\left(A_{k-1}(s') + \Gamma_k(s',s) + B_k(s)\right)$ and the a-posteriori LLRs $L(u_k \mid \underline{y})$ can be calculated as their difference.

3.1.3 Log-MAP Algorithm

It was found by Robertson et al. [23] Max-Log-MAP algorithm would result in worse performance than MAP algorithm when used iterative decoding due to the rough approximation. But the approximation can be made exact by using the Jacobian logarithm:

$$\begin{aligned}\ln(e^{x_1} + e^{x_2}) &= \max(x_1, x_2) + \ln\left(1 + e^{-|x_1 - x_2|}\right) \\ &= \max(x_1, x_2) + f_c(|x_1 - x_2|) \\ &= g(x_1, x_2)\end{aligned} \tag{29}$$

where $f_c(\sigma)$ stands for a correction term and σ equals to the magnitude of the difference between x_1 and x_2. $f_c(\sigma)$ need not be computed for every value of σ, but instead can be stored in a look-up table. There are several ways to implement the look-up table and make the algorithm have other names such as constant-log-MAP, linear-log-MAP algorithms.

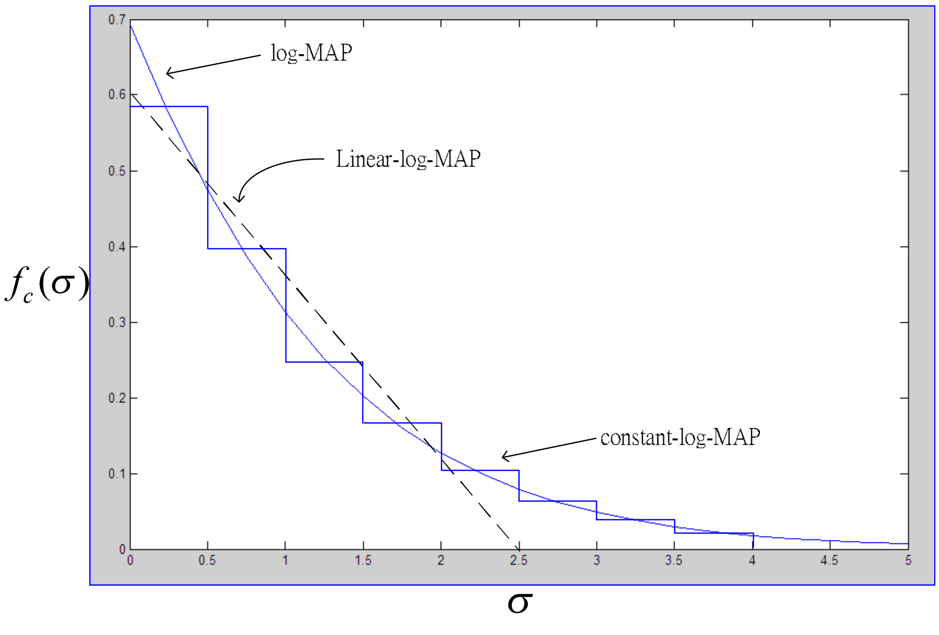


Figure 3.5: Various look-up table for Log-MAP

For binary trellises $A_k(s)$ and $B_{k-1}(s')$ can be written as:

$$\begin{aligned} A_k(s) &\triangleq \ln\left(\alpha_k(s)\right) \\ &= \ln\left(\sum_{\text{all } s'} \exp\left[A_{k-1}(s') + \Gamma_k(s',s)\right]\right) \\ &\approx \max\left(\left(A_{k-1}(s') + \Gamma_k(s',s)\right), \left(A_{k-1}(s'') + \Gamma_k(s'',s)\right)\right) \\ &\quad + f_c\left(\left|\left(A_{k-1}(s') + \Gamma_k(s',s)\right) - \left(A_{k-1}(s'') + \Gamma_k(s'',s)\right)\right|\right) \end{aligned} \tag{30}$$

$$\begin{aligned} B_{k-1}(s') &\triangleq \ln\left(\beta_{k-1}(s')\right) \\ &\approx \max\left(\left(B_k(s) + \Gamma_k(s',s)\right), \left(B_k(s'') + \Gamma_k(s',s'')\right)\right) \\ &\quad + f_c\left(\left|\left(B_k(s) + \Gamma_k(s',s)\right) - \left(B_k(s'') + \Gamma_k(s',s'')\right)\right|\right) \end{aligned} \tag{31}$$

Because there will be $2 \cdot 2^{K-1}$ transitions at each stage of the trellis for binary trellis, there will be 2^{K-1} transitions in each of the maximizations in equation (30) (31), where K is the constraint length of the convolutional code. If we want to apply the Jacobian logarithm to it, we need to nest the $g(x_1, x_2)$ operations. Then we should use the nesting equation shown below:

$$\ln\left(\sum_{i=1}^{I} e^{x_i}\right) = g\left(x_I, g\left(x_{I-1}, \cdots, g\left(x_3, g\left(x_2, x_1\right)\right)\right)\cdots\right) \tag{32}$$

3.1.4 SNR mismatch

According to [24] [25], the BER performance of the Log-MAP algorithm would decrease if the channel's SNR ratio estimation is not estimated correctly. As the frame size of Turbo code increases, the effect on BER performance would become more severe. Contrarily the BER performance of Max-Log-MAP will not be affected by the mismatched SNR.

The reason for BER performance affected by SNR mismatch is the non-linear character of Log-MAP algorithm. The difference between Max-Log-MAP algorithm and Log-MAP algorithm is the correction term on the right hand side in equation (29). The correction term results in non-linear character of Log-MAP algorithm. When we calculate the branch metrics, state metrics, a-posterior LLR and extrinsic information iteratively, their values will be affected by the non-linear term. Since the approximation used by Max-Log-MAP algorithm is linear, the branch metrics, state metrics, a-posterior LLR and extrinsic information all will be scaled by L_c simultaneously. Therefore, we can let L_c equal to one in the calculations.

3.1.5 Conclusion

As mentioned, there are two kinds of SISO decoding algorithms could be adopted in the turbo decoder. One is the family of MAP algorithms and the other is SOVA. Although [23] claims that the SOVA has only half the complexity of the Max-Log-MAP, there are other researches [26] find SOVA is more complex than Max-Log-MAP unless the decoder using SOVA is designed carefully. No matter how the decoder using SOVA is implemented, the BER performance is worse than or equal to (at most) the performance of Max-Log-MAP. Therefore we do not discuss about SOVA in this thesis.

The original MAP algorithm does not suit to be implemented on the hardware due to it needs many multiplications and exponential calculations. Therefore, the most popular Turbo decoding algorithms derived from MAP algorithm and have been adopted in the hardware implementations are Log-MAP and Max-Log-MAP algorithms. As we described, Max-Log-MAP algorithm is a simplified version of Log-MAP algorithm and the former BER performance is slight worse than the latter one. But according to the analyses from [26], the computational complexity of Log-MAP algorithm is 2 to 3 times as complex as Max-Log-MAP.

According to section 3.1.4, Log-MAP algorithm suffers from SNR mismatch problem but Max-Log-MAP algorithm does not. Even if the channel SNR could be estimated correctly real time, Log-MAP algorithm still needs several lookup tables in the hardware implementation but Max-Log-MAP does not. In fact, the channel varies at any time and on-line SNR estimation is impracticable to some degree. Therefore we implement Max-Log-MAP algorithm on our hardware.

3.2 Memory saving methodologies

In turbo decoding, the memory part always plays an important rule because it occupies most of the area of the decoder. In this chapter, we will introduce the original decoder structure and there kinds of saving memory decoding method, including preprocessing over whole block method, preprocessing over window method and halfway method. Finally we will compare these methods in memory capacity aspect. When we say memory capacity in this section, we mean those used to store the state metrics.

From equation (28) the a-posteriori LLRs $L(u_k|\underline{y})$ are calculated as:

$$L(u_k|\underline{y}) = \max_{\substack{(s',s)\Rightarrow \\ u_k=+1}} \left(A_{k-1}(s') + \Gamma_k(s',s) + B_k(s) \right) - \max_{\substack{(s',s)\Rightarrow \\ u_k=-1}} \left(A_{k-1}(s') + \Gamma_k(s',s) + B_k(s) \right)$$

That means we must have the values of $A_{k-1}(s')$, $\Gamma_k(s',s)$ and $B_k(s)$ at first. As we know, $A_{k-1}(s')$ is calculated in forward recursive manner and $B_k(s)$ is calculated in backward recursive manner.

Assume the inputs to the encoders are binary, the component encoder has constraint length K and the data need to be decoded have a frame length N. We do the backward recursion first due to be capable of making decisions in the usual order of the data. In order to make decisions over the whole frame, the state metrics calculated during the first processing (backward) must be memorized. Then the required memory size for a received frame length N is $M_r = N \cdot 2^{K-1} \cdot q$, where q is the number of quantization binary digits. The operations flow and the memory required are shown in Figure 3.6.

We take the specification from 3GPP turbo code for example, the maximum frame length (N_{max}) is 5114 and the constraint length (K) is 4 so that if we set the number of quantization bits (q) equal to 10, we will need about 410 Kbits. In most case, reducing the size of the memory is necessary.

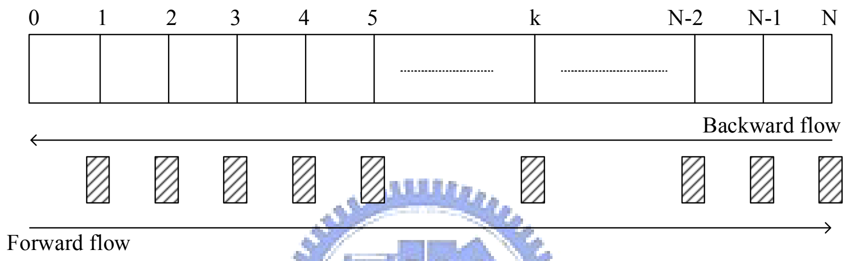


Figure 3.6: Operations on a frame of size N

(The rectangles with gray lines are the memories required during the processing)

3.2.1 Preprocessing over Whole Block Method

The first method for reducing the memory size uses the concept of initialization. The initialization process precedes the first processing in the same order (backward). Choose a number L as the length of a block and calculate the number p by $p = \left\lfloor \frac{N}{L} \right\rfloor$, then the forward flows and backward flows are subdivided into p sub-process. The operations flow and the memory required are shown in Figure 3.7.

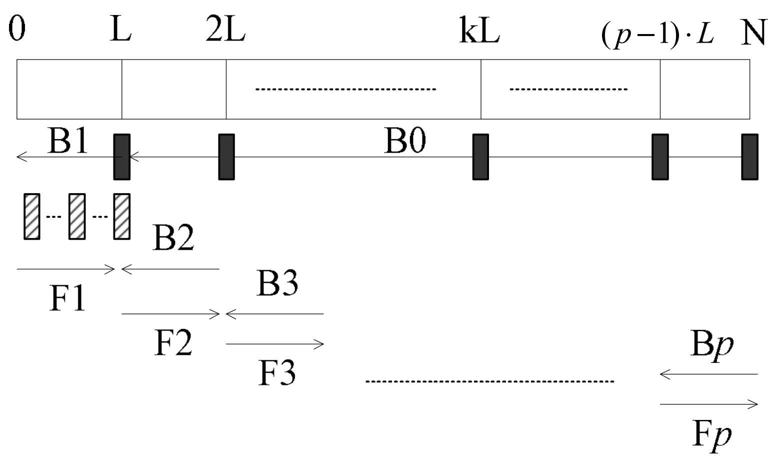


Figure 3.7: Operations on a frame of size N for preprocessing over whole block (The rectangles with gray lines are the memories required during the processing of making decision and the black rectangles represent the memories for initialization)

In the beginning, we perform the backward calculations and store the backward state metrics in the memories (which are indicated as black rectangles in Figure 3.7 periodically (period=*L*). The stored values will serve as initialization metrics for the backward sub-processes. So the backward flow is carried out on successive windows of size *L*, where the starting state metrics are known.

The capacity of the memory for initialization is $M_{ri} = (p-1)\cdot 2^{K-1}\cdot q$. The capacity of the memory for making decisions is $M_{rd} = L\cdot 2^{K-1}\cdot q$ for using only one ACS processor. If using two ACS processors, it can be shown that the required memory size can decrease as $M_{rd} = (L-1)\cdot 2^{K-1}\cdot q$. So the overall required memories are $M_{total} = (L+p-1)\cdot 2^{K-1}\cdot q$ for using one ACS processor and $M_{total} = (L+p-2)\cdot 2^{K-1}\cdot q$ for using two ACS processors. In general, we choose $p = L \approx \left\lceil \sqrt{N} \right\rceil$ because this choice can offer the minimal memory capacity.

We take the 3GPP Turbo encoder for example again and let $q = 10$, we get:

$$p = L \approx \left\lceil \sqrt{N} \right\rceil = \left\lceil \sqrt{5114} \right\rceil = 72$$

the total memory capacity is $(72 + 72 - 1) \cdot 2^3 \cdot 10 = 11440 = 11.44\ \text{Kbits}$. This number is 36 times smaller than direct decoding method.

3.2.2 Sliding Window Method

The sliding window method, proposed by [27], is based on the trellis convergence property of convolutional code. That is, if the Viterbi decoder started in unknown state, the state metrics generated initially are useless. But after a few constraint length (usually five to ten times constraint length), the set of the state metrics are as reliable as if the process had been started at the initial node. This fact can also apply to the backward and forward recursive calculations in turbo codes. Now the initialization state metrics for backward recursive calculations ($B_k(s)$ or B_k) do not need to wait until finishing pre-processing over almost the whole block. The operations flow is shown in Figure 3.8. The pre-processing length is b bits and the whole frame is divided into p blocks. Each block is L bits long except the last one is $(N \bmod L)$. It is apparent that the memories needed are fewer than the fore-method if L is small. The total capacity of the memories required to make decisions is $M = L \cdot 2^{K-1} \cdot q$.

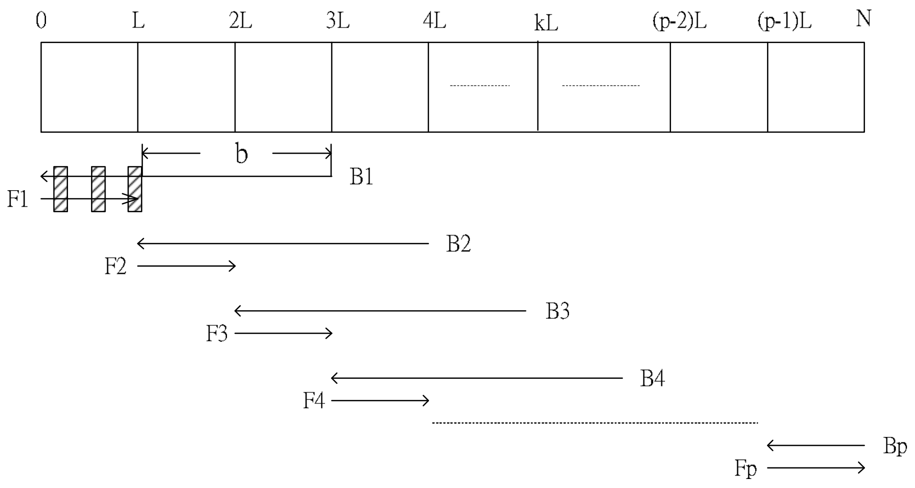


Figure 3.8: Operation flow for sliding window method

(The rectangles with gray lines are the memories required during making decisions)

Generally speaking, $L = cK \quad c = 5 \sim 10$, K is the constraint length, and $b = dL, d \in \mathrm{N}$.

We take 3GPP Turbo code as an example and assume $L = 5 \cdot K = 20$ and $b = L$ and $q = 10$. The required memory capacity is:

$$M = L \cdot 2^{K-1} \cdot q = 20 \cdot 8 \cdot 10 = 1600 = 1.6 \text{ Kbits}$$

This number is 256 times smaller than the direct decoding method.

3.2.3 Halfway Method

Halfway method was originally proposed by [28]. In this thesis, we make some modifications on the original version. The original version is applied to the data frame which is made up of the received data of frame size N followed by the one of these same data in the interleaved order. Therefore, the data frame is $2N$ bits long. We make modifications so that this method can be applied to decode the data in natural order and in the interleaved order respectively. This method is kindly like the first method, preprocessing over the whole block method which needs to use periodic memorizing. Backward sub-processes are carried out successively on blocks of data which is L bits long and the metrics calculated are need to be memorized for making the decisions with forward sub-processes. Each backward sub-process is followed by a forward sub-process on the same data block. The required memories for calculation of the decisions are $M_{rd} = L \cdot 2^{K-1} \cdot q$.

Different from the first method, the initialization metrics for each backward sub-process are set in a uniform and arbitrary way. The 2^{K-1} calculated metrics on the first data of the interval of size L in each backward sub-process are needed to be stored. They will serve as the initialization metrics for the sub-processes starting from the next iteration. The memory capacity for these kind initialization metrics is $M_{ri} = D \cdot (p-1) \cdot 2^{K-1} \cdot q$ where the first term in right hand side, D, represents the number of the component encoders. Usually, we use two RSC encoders in turbo code, that is $D = 2$. The overall required memories are $M_{total} = (L + D \cdot (p-1)) \cdot 2^{K-1} \cdot q$ for using only one ACS processor and $M_{total} = (L + D \cdot (p-2)) \cdot 2^{K-1} \cdot q$ for that with two processors. This method is most effective and fastest in these three memory saving algorithms because there are no initialization processing and processing forcing the convergence of the trellis needed in the operations.

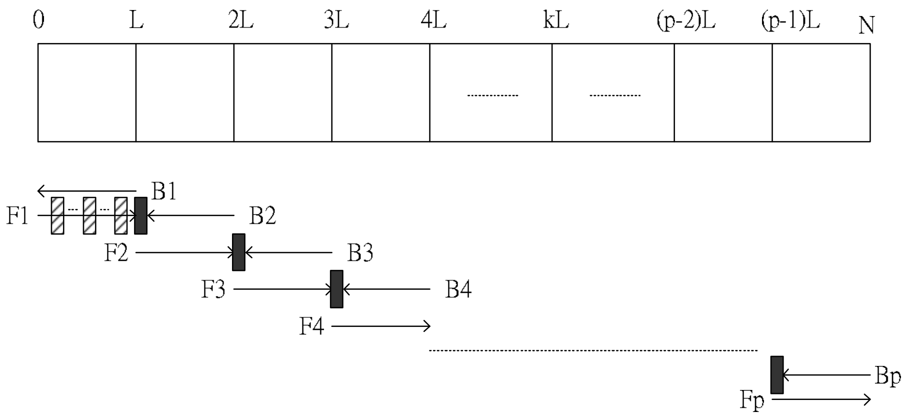


Figure 3.9: Operations flow for Halfway

(The rectangles with gray lines are the memories required during making decisions and the black rectangles represent the memories for next-iteration initialization)

We take 3GPP Turbo code as an example and assume $L = 93$, $p = 55$ and $q = 10$, the memory capacity is

$$\left(93 + 2 \cdot (55 - 1)\right) \cdot 8 \cdot 10 = 16080 = 16.08 \text{ Kbits}$$

This number is 25 times smaller than the direct decoding method.

3.2.4 Comparisons

The memory capacities for each method are listed in Table 3.1. Because preprocessing over whole block method needs an initialization process over whole block, its speed is slower than those who do not need initializations. The sliding window method also needs several initialization processes over some small windows. According to the length of the block (L) and the length of pre-processing (b), the sliding windows method may be slower or faster than the first memory saving method but never be faster than Halfway method. If the length of pre-processing is bigger than L, the overlapping calculations of backward metrics occur more times and it will make the decoding speed slow down. Because halfway method needs no initializations, it can perform as faster as the original Max-Log-MAP algorithm.

Using one ACS processor	**Memory capacity**
Max-Log-MAP	$N \cdot 2^{K-1} \cdot q$
Preprocessing over whole block	$\left(L + p - 1\right) \cdot 2^{K-1} \cdot q$
Sliding window	$L \cdot 2^{K-1} \cdot q$
Halfway	$\left(L + D \cdot (p - 2)\right) \cdot 2^{K-1} \cdot q$

Table 3.1: Comparisons of saving memory decoding methods

Assume using only one decoder with only one ACS processor. That is the decoder can only deal with one trellis stage at one time regardless of forward recursion calculations or backward recursion calculations. The following symbols will be used:

N	the length of one data frame
K	the constraint length of the convolutional encoder
q	the number of quantization binary digits
L	block length
p	the number of block
b	the length for convergence when using sliding window, $b = dL, d \in \mathrm{N}$
D	the number of the encoders

We will use the subscript "wb" as "whole block", the subscript "sw" as "sliding window", the subscript "hw" as "halfway", POWB as "preprocessing over whole block method", SW as "sliding window method" and HW as "halfway method".

The comparison of the complexities bases on the same decoding algorithm but different memory saving method. We only need to consider only the numbers of trellis stages required processing. The numbers of stages required processing per half iteration for each method is listed below:

POWB: $2 \cdot N + (N - L_{wb}) = 3 \cdot N - L_{wb}$

SW: $N \bmod L_{sw} = 0$

$$\begin{cases} 2 \cdot N + (p_{sw} - 1) \cdot b & \text{if } b < L_{sw} \\ 2 \cdot N + (p_{sw} - d) \cdot b & b = d \cdot L_{sw} \quad d \in N \end{cases}$$

HW: $2 \cdot N$

The hypothesis $N \bmod L_{sw} = 0$ for SW is an assumption without losing generality.

For POWB, the advantages are that it needs fewer memories than direct decoding without using any memory saving skills and it also provides the same performance as direct decoding. The disadvantages are the decoding latency and the need for many calculations to initialize the backward initialization memory.

For SW, the advantage is that the memory capacity needed is smaller than other methods if let $L_{sw} < L_{hw}, L_{wb}$. The disadvantage is the need for initializations. If $L_{sw} < L_{wb}$ and $b \ge L_{sw}$, it will need more calculations of initialization than POWB. When $b \ge L_{sw}$, there will be $(p-d)\cdot(b-L_{sw})$ overlapping calculations for initializations.

For HW, the advantage is the lack of the initialization; therefore it needs as many calculations as direct decoding does. This is very helpful in using only one ACS processor. The simulation performance of HW in our test is equal to SW. The disadvantage is the memory capacity compared to other memory saving methods.

From another aspect, decoding one bit will need to calculate m trellis stages, where m is:

$$m_{POWB} = \frac{(3\cdot N - L_{wb})}{N} = 3 - \frac{L_{wb}}{N} \approx 3$$

$$m_{SW} \approx \frac{2\cdot L_{sw} + b}{L_{sw}} = 2 + \frac{b}{L_{sw}}$$

$$m_{HW} = \frac{2\cdot N}{N} = 2$$

We define efficiency as follows:

$$\begin{aligned} efficiency &= \frac{decode\ one\ bit's\ information}{number\ of\ forward \text{ and } backward\ state\ metrics\ calculated} \\ &= m^{-1} \end{aligned}$$

Theoretically, decoding one bit will require one forward state metrics calculation and one backward state metrics calculation. By observing the above definition, we know halfway method provide the same efficiency as the theoretic value and it is the most efficient calculation in these three memory-saving methods. Furthermore, the redundant calculations will consume unnecessary power.

The comparison curves of the BER performances of the SW and HW method are shown in Figure 3.10. Assume using 3GPP turbo coder, the data frame size $N = 500$, $L_{HW} = 256$, $L_{SW} = 24$ and $b = 24$. The word “SW-#A-#B” in Figure means the curve uses sliding window method with $L_{SW} = \#A$ and $b = \#B$; the word “HW-#C” means the curve uses halfway method with $L_{HW} = \#C$.

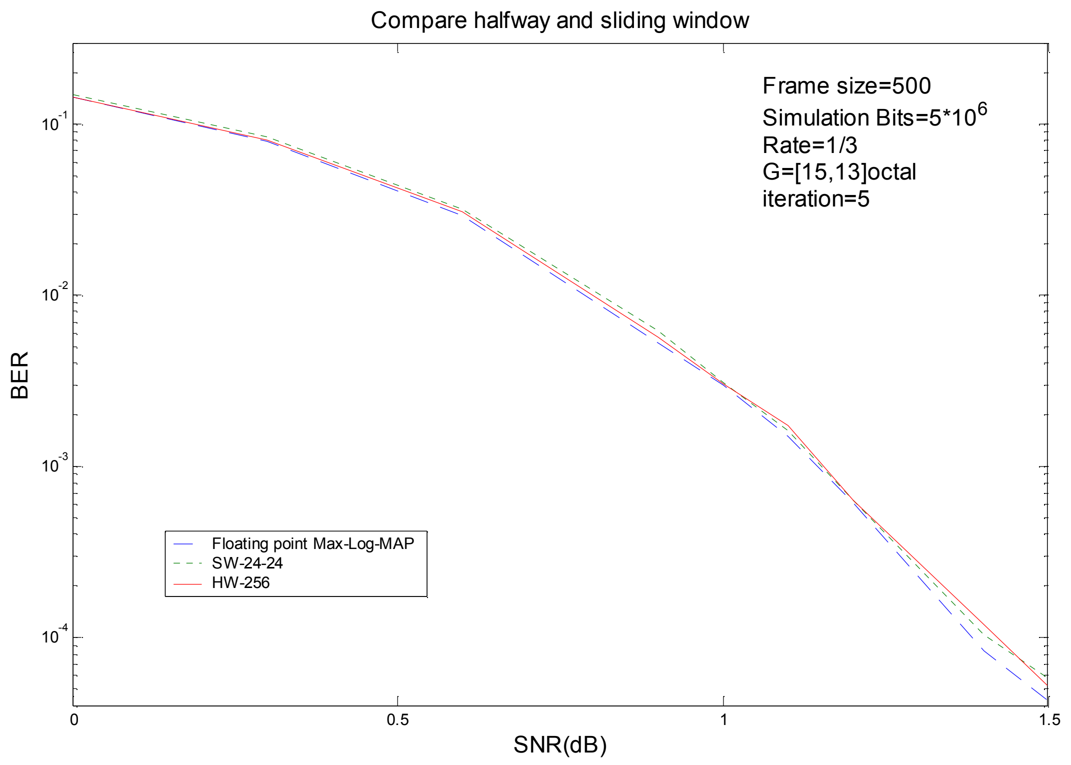


Figure 3.10: Compare the performances of halfway and sliding window

If using sliding window method with fewer than $(d+1)$ ACS processors, it will lead to an additional memory of $2^{K-1} \cdot q$ and decrease the decoding speed. The problem of the slow decoding speed of sliding windows method could be solved by using $(d+1)$ ACS processors. However, the decoder using halfway method only needs an additional ACS processor, totally two ACS processors, can achieve the same decoding speed as the sliding window method with $(d+1)$ ACS processors.

Chapter 4
3GPP Turbo Encoder

3GPP Turbo coder [29] uses Parallel Concatenated Convolutional Code (PCCC) with two 8-state constituent encoders and one internal interleaver. The code rate of Turbo coder is 1/3. The structure of 3GPP Turbo coder is shown in Figure 4.1.

The requests of the encoder are as follows:

1. The initial value of the shift registers of the constituent encoders shall be all zeros.
2. Outputs from the Turbo coders are

$$x_{s1}, x_{p1}, x'_{p1}, \ldots, x_{sK}, x_{pK}, x'_{pK}$$

where $x_{s1}, x_{s2}, \ldots, x_{sK}$ are the systematic bits which equal to the input bits u_k to the Turbo encoder, and K is the number of a block of input bits, and $x_{p1}, x_{p2}, \ldots, x_{pK}$ and $x'_{p1}, x'_{p2}, \ldots, x'_{pK}$ are the bits output from first and second constituent encoders, respectively. The bits output from Turbo code internal interleaver are denoted by $u'_1, u'_2, \ldots, u'_k$ and these bits are to be input to the second constituent encoder.

4.1 Constituent Encoder

3GPP constituent 8-state encoder and its corresponding trellis diagram are shown in Figure 4.1. The transfer function of the 8-state constituent code for PCCC is:

$$G(D) = \left[1, \frac{1 + D^2 + D^3}{1 + D + D^3} \right],$$

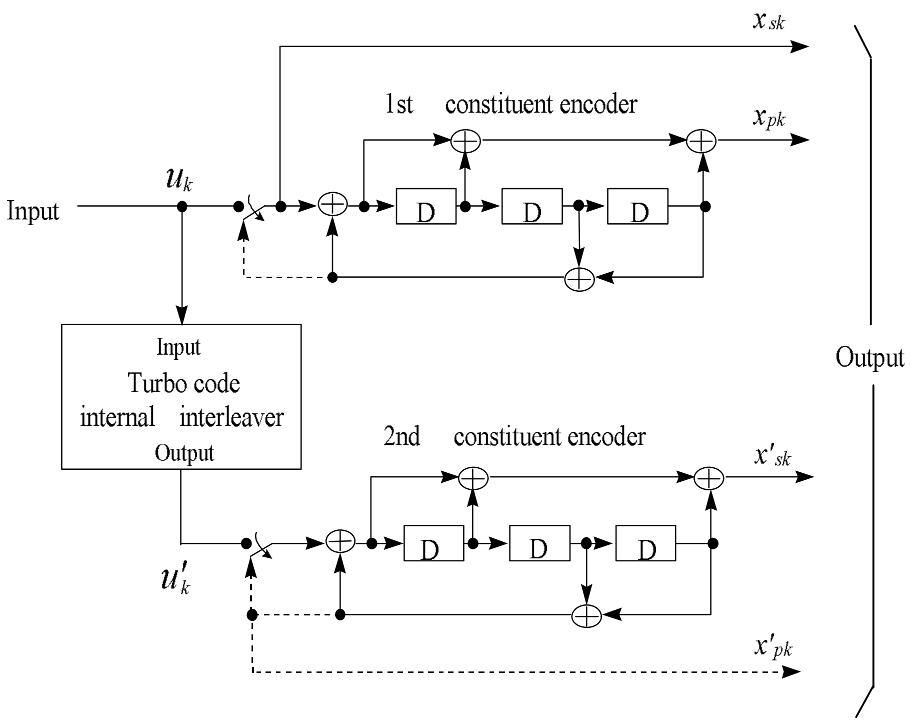


Figure 4.1: Structure of rate 1/3 Turbo coder

(dotted lines apply for trellis termination)

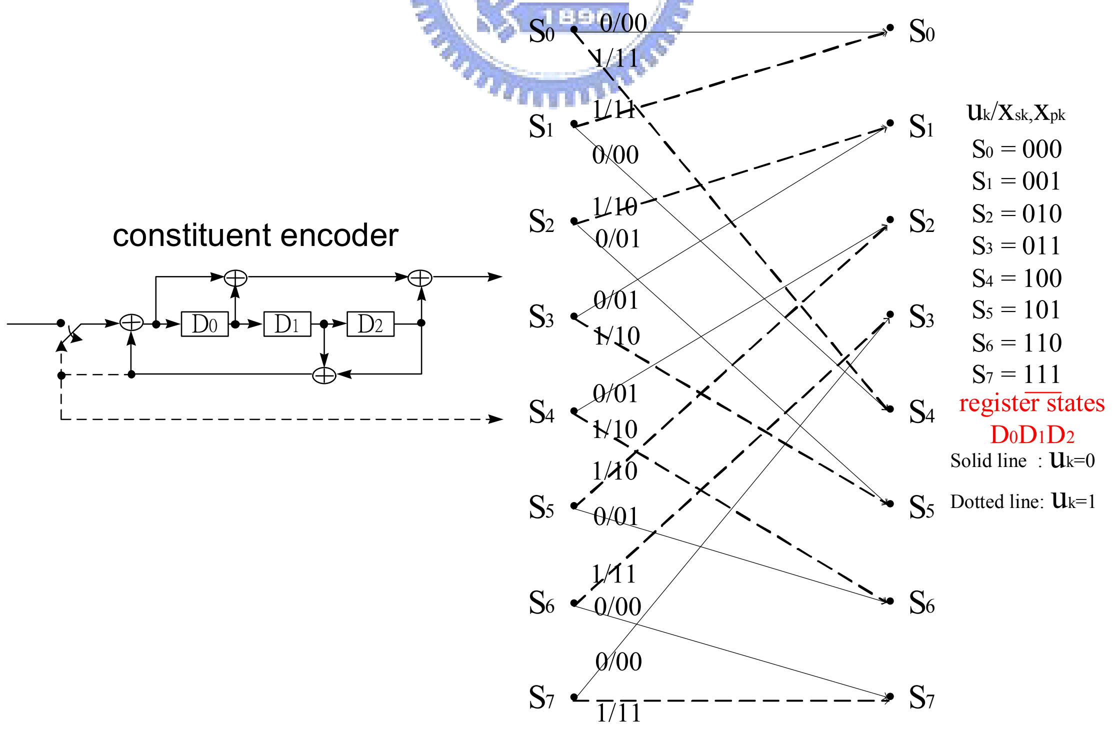


Figure 4.2: Constituent encode of 3GPP turbo encoder and its trellis

4.2 Trellis Termination

Because the first request, the initial value of the shift registers of the encoders shall be all zeros, both the constituent encoders need to perform trellis termination after encoding one block of input bits. The action of terminating the trellis is performed by taking the last $K-1$ bits from the shift register of each encoder feedback to their selves after all information bits are encoded then all shift registers will return to zero. The switch in each constituent encoder should be switched to the lower position when terminating and the structure of each encoder is shown in Figure 4.3. These encoded tail bits are padded after the encoded information bits, and the transmitted bits for trellis termination shall be:

$$x_{s(K+1)}, x_{p(K+1)}, x_{s(K+2)}, x_{p(K+2)}, x_{s(K+3)}, x_{p(K+3)}, x'_{s(K+1)}, x'_{p(K+1)}, x'_{s(K+2)}, x'_{p(K+2)}, x'_{s(K+3)}, x'_{p(K+3)}$$

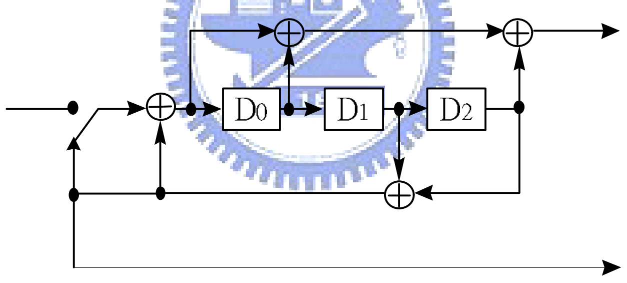


Figure 4.3: Constituent encoder for terminating the trellis

4.3 Interleaver

The 3GPP Turbo code internal interleaver is a block interleaver consisting of a rectangular matrix and its size is decided by the frame size of the input bits, *K*. The original message bits input to the interleaver row by row. If the input bits are not enough to filling the matrix, we need to add some redundant bits to fill it. Then we perform intra-row permutations and inter-row permutations of the rectangular matrix. Finally, the bits in the matrix are read out column by column and pruning the redundant bits we added before. We denote the bits input to the internal interleaver by $u_1, u_2, u_3, \dots u_K$, where K is the integer number of the bits and takes one value of 40 $\leq$ K $\leq$ 5114.

4.3.1 Deciding the size of the rectangular matrix

First to all, we need to decide the number of the rows and the columns of the rectangular matrix according the following process:

(1) According to the equation (33), determining the number of rows of the rectangular matrix, R. The rows of rectangular matrix are numbered 0, 1, …, R - 1 from top to bottom

$$R = \begin{cases} 5, \text{if } (40 \leq K \leq 159) \\ 10, \text{if } ((160 \leq K \leq 200) \text{ or } (481 \leq K \leq 530)) \\ 20, \text{if } (K = \text{any other value}) \end{cases} \tag{33}$$

(2) Along with Table 4.1 and relationship shown below, we can determine the prime number, p, used in the intra-permutation and the number of columns of rectangular matrix, C. The columns of rectangular matrix are numbered 0, 1, …, C - 1 from left to right

```
if (481 ≤ K ≤ 530) then
    $p = 53$ and $C = p$.
else
    Find minimum number $p$ from Table 4.1 such that
        $K \leq R \times (p + 1)$,
    and determine $C$ such that
```

$$C = \begin{cases} p-1 & \text{if} \quad K \le R \cdot (p-1) \\ p & \text{if} \quad R\cdot(p-1) < K \le R \cdot p \\ p+1 & \text{if} \quad R \cdot p \le K \end{cases}$$

end if

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

Table 4.1: List of prime number p and associated primitive root v

(3)Write the input bit sequence $u_1, u_2, u_3, \ldots u_K$ into the $R \times C$ rectangular matrix row by row:

$$\begin{bmatrix} y_1 & y_2 & y_3 & \cdots & y_C \\ y_{(C+1)} & y_{(C+2)} & y_{(C+3)} & \cdots & y_{2C} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ y_{((R-1)\cdot C+1)} & y_{((R-1)\cdot C+2)} & y_{((R-1)\cdot C+3)} & \cdots & y_{R\cdot C} \end{bmatrix}$$

where $y_k = u_k$ for k = 1, 2, …, K and if $R \times C > K$, the dummy bits are added to the tail of the input sequence such that $y_k = 0$ or 1 for $k = K + 1, K + 2, \ldots, R \times C$. These dummy bits will be discarded when read the bits from the rectangular matrix after intra-row and inter-row permutations.

After the $R \times C$ rectangular matrix is filled with the input and dummy bits, we perform the intra-row permutations and inter-row permutations in turn.

Number of input bits K	Number of rows R	Inter-row permutation patterns <T(0), T(1), …, T(R - 1)>
(40≦K≦159)	5	<4, 3, 2, 1, 0>
(160 ≦ K ≦ 200) or (481≦K≦530)	10	<9, 8, 7, 6, 5, 4, 3, 2, 1, 0>
(2281 ≦ K ≦ 2480) or (3161≦K≦3210)	20	<19, 9, 14, 4, 0, 2, 5, 7, 12, 18, 16, 13, 17, 15, 3, 1, 6, 11, 8, 10>
K = any other value	20	<19, 9, 14, 4, 0, 2, 5, 7, 12, 18, 10, 8, 13, 17, 3, 1, 16, 6, 15, 11>

Table 4.2: Inter-row permutation patterns for Turbo code internal interleaver

4.3.2 Intra-row and inter row permutations

After the bits input to the $R \times C$ rectangular matrix, the intra-row and inter-row permutations for the $R \times C$ rectangular matrix are performed stepwise by using the following algorithm with steps (1) – (6):

(1) Select a primitive root v from Table 4.1, which is indicated on the right side of the prime number p.

(2) Construct the base sequence $\langle s(j) \rangle_{j \in \{0,1,\cdots,p-2\}}$ for intra-row permutation as:

$s(j) = (v \cdot s(j-1)) \bmod p$, $j = 1, 2, \cdots, (p-2)$, and s(0) = 1.

(3) Assign q_0 = 1 to be the first prime integer in the sequence $\langle q_i \rangle_{i \in \{0,1,\ldots,R-1\}}$, and determine the prime integer q_i in the sequence $\langle q_i \rangle_{i \in \{0,1,\ldots,R-1\}}$ to be a least prime integer such that $g.c.d(q_i, p-1) = 1$, $q_i > 6$, and $q_i > q_{i-1}$ for each $i = 1, 2, \cdots, R-1$. Here *g.c.d.* is greatest common divisor.

(4) Permute the sequence $\langle q_i \rangle_{i \in \{0,1,\ldots,R-1\}}$ to make the sequence $\langle r_i \rangle_{i \in \{0,1,\cdots,R-1\}}$ such that

$r_{T(i)} = q_i$, $i = 0, 1, \cdots, R-1$

where $\langle T(i) \rangle_{i \in \{0,1,\cdots,R-1\}}$ is the inter-row permutation pattern defined as the one of the four kind of patterns, which are shown in Table 4.2, depending on the number of input bits K.

(5) Perform the i-th (i = 0, 1, …, R - 1) intra-row permutation as:

```
if ($C = p$) then
 $U_i(j) = s((j \times r_i) \bmod (p-1))$,   $j$ = 0, 1, …, ($p$ - 2), and $U_i(p - 1) = 0$,
 where $U_i(j)$ is the original bit position of $j$-th permuted bit of $i$-th row.
end if
if ($C = p + 1$) then
  $U_i(j) = s((j \times r_i) \bmod (p-1))$,   $j$ = 0, 1, …, ($p$ - 2).  $U_i(p - 1) = 0$, and $U_i(p) = p$,
  where $U_i(j)$ is the original bit position of $j$-th permuted bit of $i$-th row, and
   if ($K = R \times C$) then
      Exchange $U_{R-1}(p)$ with $U_{R-1}(0)$.
   end if
end if
if ($C = p - 1$) then
```

$$U_i(j) = s((j \times r_i) \bmod (p-1)) - 1, \qquad j = 0, 1, \ldots, (p - 2),$$

where $U_i(j)$ is the original bit position of j-th permuted bit of i-th row.

end if

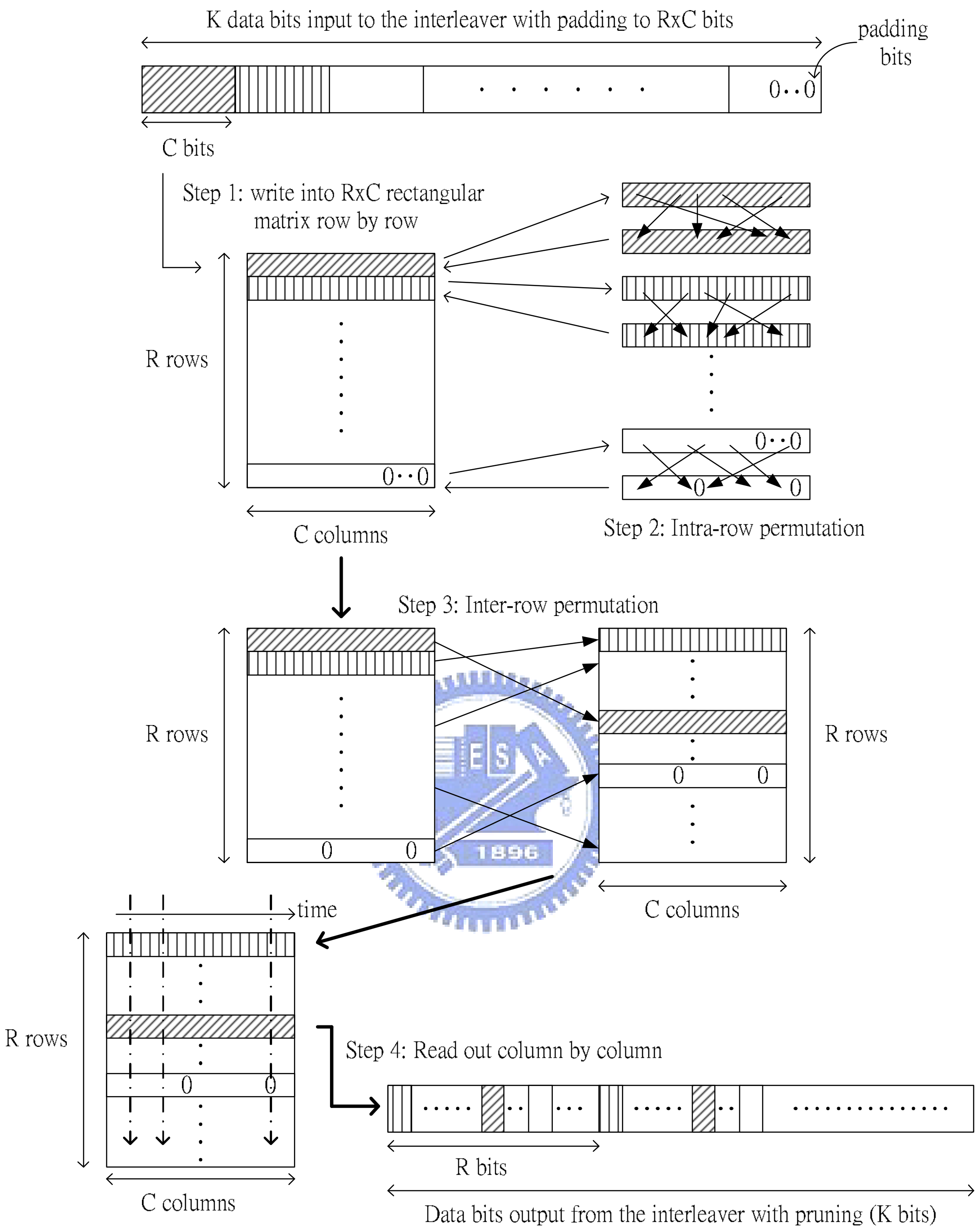
(6) Perform the inter-row permutation for the rectangular matrix based on the pattern $\langle T(i) \rangle_{i \in \{0,1,\cdots,R-1\}}$, where $T(i)$ is the original row position of the i-th permuted row.

4.3.3 Output the bits from the rectangular matrix with pruning

After intra-row and inter-row permutations, the bits of the permuted rectangular matrix are denoted by y'_k:

$$\begin{bmatrix} y'_1 & y'_{(R+1)} & y'_{(2R+1)} & \cdots & y'_{((C-1)\cdot R+1)} \\ y'_2 & y'_{(R+2)} & y'_{(2R+2)} & \cdots & y'_{((C-1)\cdot R+2)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ y'_R & y'_{2R} & y'_{3R} & \cdots & y'_{C\cdot R} \end{bmatrix}$$

The output of the Turbo code internal interleaver is the bit sequence read out column by column from the intra-row and inter-row permuted $R \times C$ rectangular matrix starting with bit y'_1 in row 0 of column 0 and ending with bit y'_{CR} in row R - 1 of column C - 1. The output is pruned by deleting dummy bits that were padded to the input of the rectangular matrix before intra-row and inter row permutations, i.e. bits y'_k that corresponds to bits y_k with $k > K$ are removed from the output. The bits output from Turbo code internal interleaver are denoted by $x'_1, x'_2, \ldots, x'_K$, where x'_1 corresponds to the bit y'_k with smallest index k after pruning, x'_2 to the bit y'_k with second smallest index k after pruning, and so on. The number of bits output from Turbo code internal interleaver is K and the total number of pruned bits is: $R \cdot C - K$. The interleaving flow chart after deciding R, C is shown in Figure 4.4.



K data bits input to the interleaver with padding to RxC bits
padding bits
0··0
C bits
Step 1: write into RxC rectangular matrix row by row
R rows
0··0
C columns
Step 2: Intra-row permutation
0··0
0
0
Step 3: Inter-row permutation
R rows
0
0
R rows
0
0
C columns
time
R rows
0
0
C columns
Step 4: Read out column by column
R bits
Data bits output from the interleaver with pruning (K bits)

Figure 4.4: Interleaving flow chart

Chapter 5
Design Considerations

The 3GPP turbo encoders are constructed by two identical encoders; therefore, we can use only one decoder to decode the received sequence serially and iteratively. In this thesis, the turbo decoder uses only one decoder and only one ACS processor to calculate the forward and backward state metrics for low complexity. When designing the hardware of the decoder, we need to discuss and consider about several issues as follows:

1. Decoding algorithm selection.
2. Memory saving method selection.
3. Decision of the block length.
4. The analyses of fixed-point representations for calculations.

5.1 Decoding algorithm selection

Due to Max-Log-MAP algorithm's low complexity and only minor performance loss comparing with Log-MAP and the poor SNR sensitivity which means we will not need any multiplications in decoding process, we implement our hardware by Max-Log-MAP algorithm.

5.2 Memory saving method selection

Because we use only one decoder for decoding, we cannot tolerate the redundant calculations for initialization processes. Since halfway method does not have the redundant calculations and needs fewer calculations than the other memory saving methods we will adopt this method to implement the hardware for saving memory capacity and reducing the power consumption.

5.3 Decision of the block length

After choosing the memory saving method, we need to decide the block length. Although the memory depth can be set arbitrary in the format of power of 2 on FPGA, it is somewhat impractical in ASIC. Here we assume the minimum memory depth is 32 then we can derive the memory capacity for the initialization memory and state metrics memory respectively for different block length. The results are shown in Table 5.1, the forth column means the original initialization memory needed. The second column is the actual memory depth we implement on hardware since we assume the smallest memory depth is 32.

block length / state metric memory depth	initialization memory depth	Total memory depth	*p*D (D=2 for 3GPP)*
32 = 32 + 0	320 = 256 + 64	352	320
64 = 64 + 0	160 = 128 + 32	224	160
96 = 64 + 32	128 = 128 + 0	224	108
128 = 128 + 0	96 = 64 + 32	224	80
160 = 128 + 32	64 = 64 + 0	224	64
192 = 128 + 64	64 = 64 + 0	256	54
256 = 256 + 0	64 = 64 + 0	320	40
288 = 256 +32	64 = 64 + 0	352	36

Table 5.1: various memory depths

We simulate the decoder of different block length for three minimum memory requirements. The performances for various block length are shown in Figure 5.1. From the Figure 5.1, we can observe that the performance gets better as the block length increases. BER_{256} is better than BER_{128} by 1 dB and BER_{192} by 0.5 dB so that we choose 256 as the block length for the turbo decoder in order not to degrade the decoder's BER performance too much.

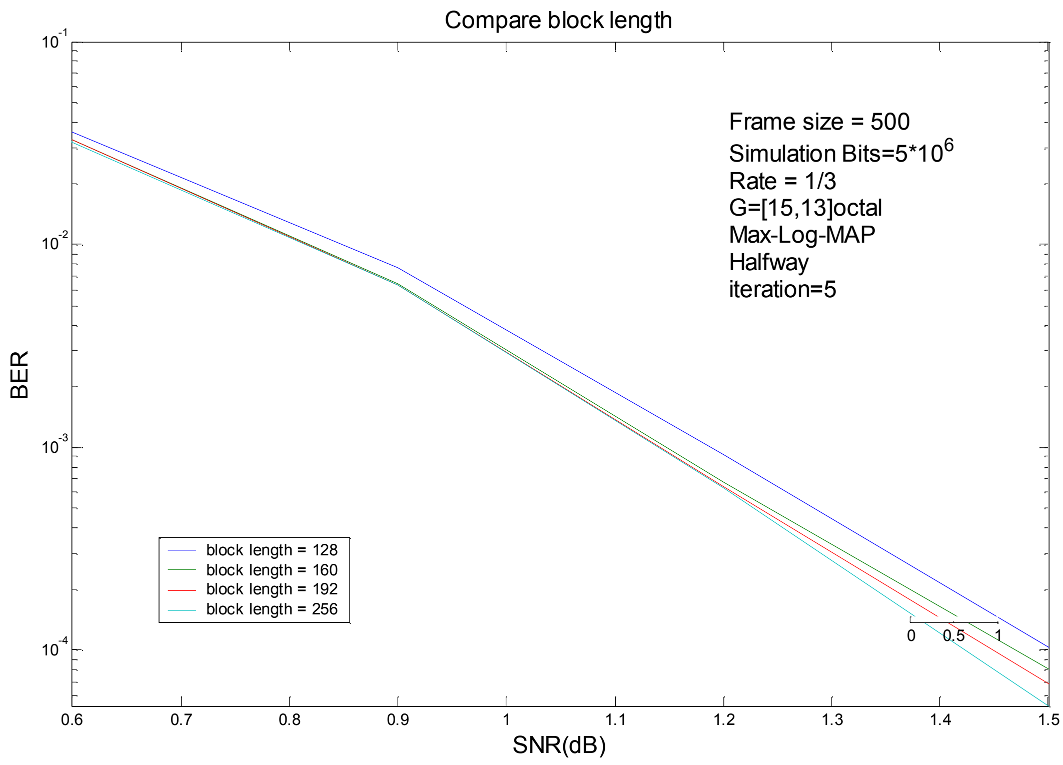


Figure 5.1: Simulations for various block length

5.4 Analyses of fixed-point representations for calculations

In the fixed-point implement of turbo decoder the word length will affect the performance of the decoder. Unnecessary bits would waste the memory space, increase the computational complexity of the hardware, consume more power and decrease the hardware speed.

The data required in Max-Log-MAP decoding process are: the received sequence y_{ks}, y_{kp}, y'_{kp}, the branch metrics $\Gamma_k(s',s)$, the forward state metrics $A_k(s)$, the backward state metrics $B_k(s)$, the a-priori information $L(u_k)$ and the extrinsic information $L_e(u_k)$, a-posterior LLR $L(u_k|\underline{y})$.

First of all, we decide the number of the fractional bits of all quantities. The simulation results are shown in Figure 5.2. By comparing the results, it is apparent that the performances are poor when we set the number of the fractional bits to 1 or 2. But when the number of the fractional bits equals to 3, the performance gets very close to Max-Log-MAP in floating point. Increasing the number of the fractional bits to 4, the performance will not get much improvement. Therefore we will choose 3 as the number of the fractional bits of all quantities.

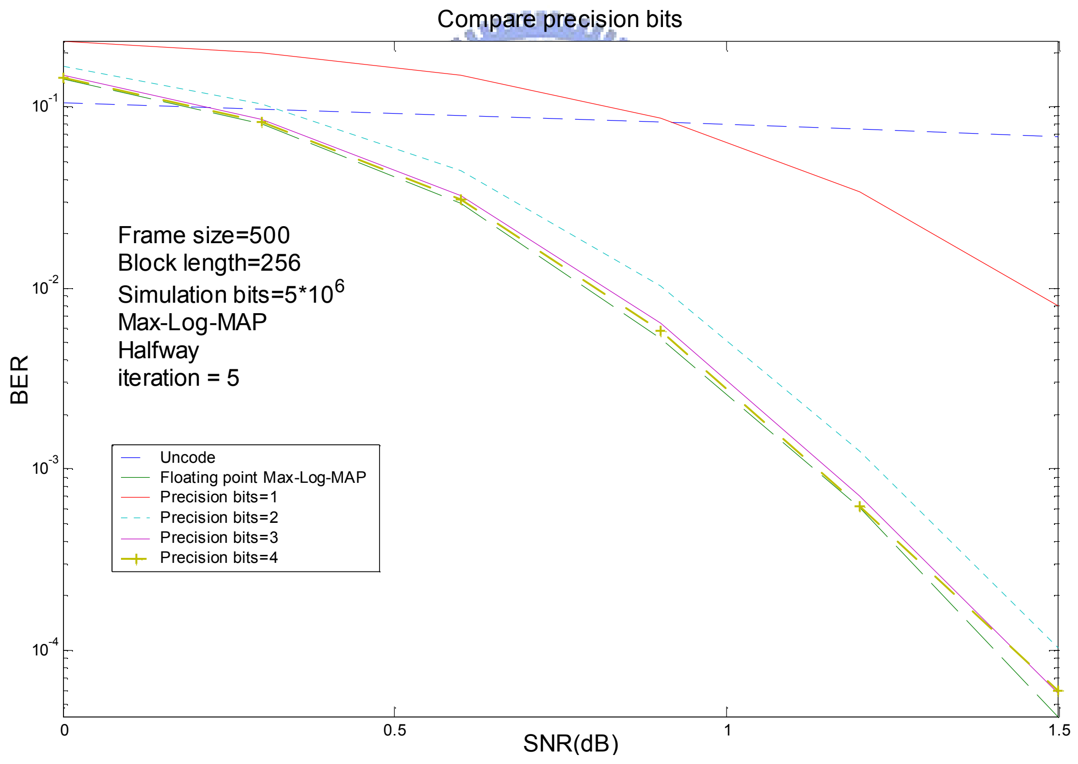


Figure 5.2: Simulations for various precision

After deciding the precision, we consider the dynamic range of all quantities to decide the number of the integer bits then we could decide the fixed point representation of all. We will use *FP(q,f)* to represent the fixed point representation, where q is the total number of the bits and f is the number of the fractional bits, then (q-f) is the number of the integer bits and that is the dynamic range.

5.4.1 Received sequence y_{ks}, y_{kp}, y'_{kp}

The received sequence mainly relates to the modulation/demodulation and the transmit channel. Assume the channel is AWGN non-fading and using BPSK modulation to transmit the encoded information and the channel's SNR at least equal to 0 dB, the value of the received sequence will distribute over 7 to -7 through the MATLAB simulations. Consequently, we need 3 bits to represent the integer value and 1 bit to represent the sign and totally we need 7 bits, *FP(7,3)* for the received sequence.

5.4.2 Branch metrics $\Gamma_k(s',s)$

The branch metrics are calculated by equation (27) and we rewrite it here

$$\Gamma_k(s',s) = \hat{C} + \frac{1}{2}u_k \cdot L(u_k) + \frac{L_c}{2}\sum_{l=1}^{n} y_{kl}x_{kl}$$

As we discussed in section 3.1.4, Max-Log-MAP is SNR independent and we can let $L_c = 1$ and discard the constant term $\hat{C}$, this equation can be rewritten as:

$$\Gamma_k(s',s) = \frac{1}{2}u_k \cdot L(u_k) + \frac{1}{2}\sum_{l=1}^{n} y_{kl}x_{kl} \tag{34}$$

where n is the number of the encoded bits and equals to 2 for 3GPP. Because it is relative to a-priori LLR, $L(u_k)$, we will decide it's fixed point representation, *FP(q_{bm},3)*, after deciding $L(u_k)$. The discuss following will assume the branch metrics are big enough to store a-priori LLR $L(u_k)$ until deciding $L(u_k)$.

5.4.3 Forward state metrics $A_k(s)$ and Backward state metrics $B_k(s)$

Because 3GPP encoder starts to encode the information from state S_0, ideally the forward recursive calculations has the initialization conditions as follows:

$$\begin{cases} A_0(S_0=0)=\ln(\alpha_0(S_0=0))=\ln(1)=0 \\ A_0(S_0=s)=\ln(\alpha_0(S_j=s))=\ln(0)=-\infty \quad for\ s\neq 0 \end{cases}$$

Since 3GPP encoder provides the trellis termination, the backward recursive calculations has the initialization conditions as follows:

$$\begin{cases} B_N(S_0=0)=\ln(\alpha_0(S_0=0))=\ln(1)=0 \\ B_N(S_0=s)=\ln(\alpha_0(S_0=s))=\ln(0)=-\infty \quad for\ s\neq 0 \end{cases}$$

In hardware design, there is no infinity value. Thus we set $A_0(S_0=0)$ and $B_N(S_0=0)$ equal to the maximum value that can be represented by *FP(q_{sm} ,3)* where q_{sm} will be decided later. And we set $A_0(S_0=s)$, $B_N(S_0=s)$ $for\ s\neq 0$ equal to zero. Generally this setting would not affect the LLR calculations except the maximum value is not infinity anymore because we only need to know their relative difference value but not their exact values. The simulation results are shown in Figure 5.3. According to the simulation results, we let q_{sm} equal to 6

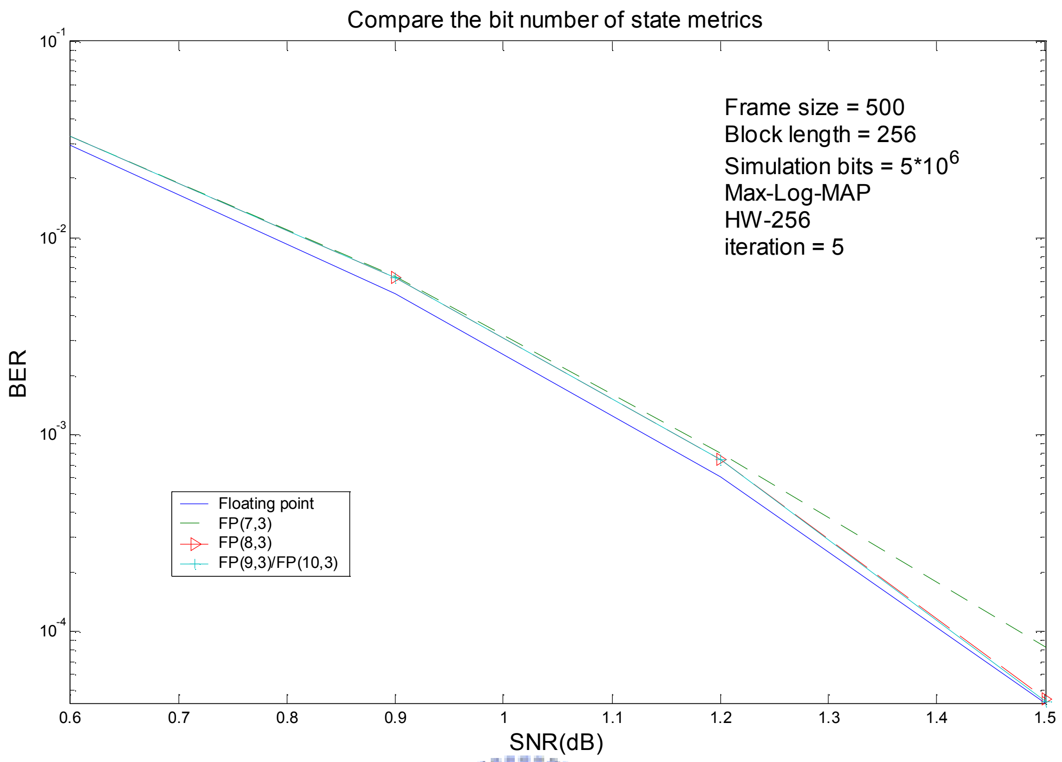


Figure 5.3: Simulations for deciding the integer bits of state metrics

5.4.4 a-priori information LLR $L(u_k)$, extrinsic information $L_e(u_k)$

From equation (1)

$$L(u_k) \triangleq \ln\left(\frac{P(u_k=+1)}{P(u_k=-1)}\right)$$

we can derive $P(u_k=\pm 1)$ as:

$$P(u_k=\pm 1)=\left(\frac{e^{-L(u_k)/2}}{1+e^{-L(u_k)}}\right)\cdot e^{\pm L(u_k)/2}$$

and the corresponding probabilities for $L(u_k)=11 \sim -12$ are tabulated in Table 5.2. Thus we can see when $L(u_k)\geq 7$, $P(u_k=+1)\geq 0.999$ and $L(u_k)\leq -8$, $P(u_k=+1)\geq 0.9996$, that is a-prior LLR for the decoded bit is highly believable. The simulation results for different integer bits of $L(u_k)$ are shown in the Figure 5.4.

$L(u_k)$	11	10	9	8	7	6	5	4
$P(u_k=+1)$	0.99998	0.99995	0.99988	0.99966	0.99909	0.99753	0.99331	0.98201
$P(u_k=-1)$	1.7E-05	4.5E-05	0.00012	0.00034	0.00091	0.00247	0.00669	0.01799
$L(u_k)$	3	2	1	0	-1	-2	-3	-4
$P(u_k=+1)$	0.95257	0.8808	0.73106	0.5	0.26894	0.1192	0.04743	0.01799
$P(u_k=-1)$	0.04743	0.1192	0.26894	0.5	0.73106	0.8808	0.95257	0.98201
$L(u_k)$	-5	-6	-7	-8	-9	-10	-11	-12
$P(u_k=+1)$	0.00669	0.00247	0.00091	0.00034	0.00012	4.5E-05	1.7E-05	6.1E-06
$P(u_k=-1)$	0.99331	0.99753	0.99909	0.99966	0.99988	0.99995	0.99998	0.99999

Table 5.2: Relationship between $L(u_k)$ and $P(u_k)$

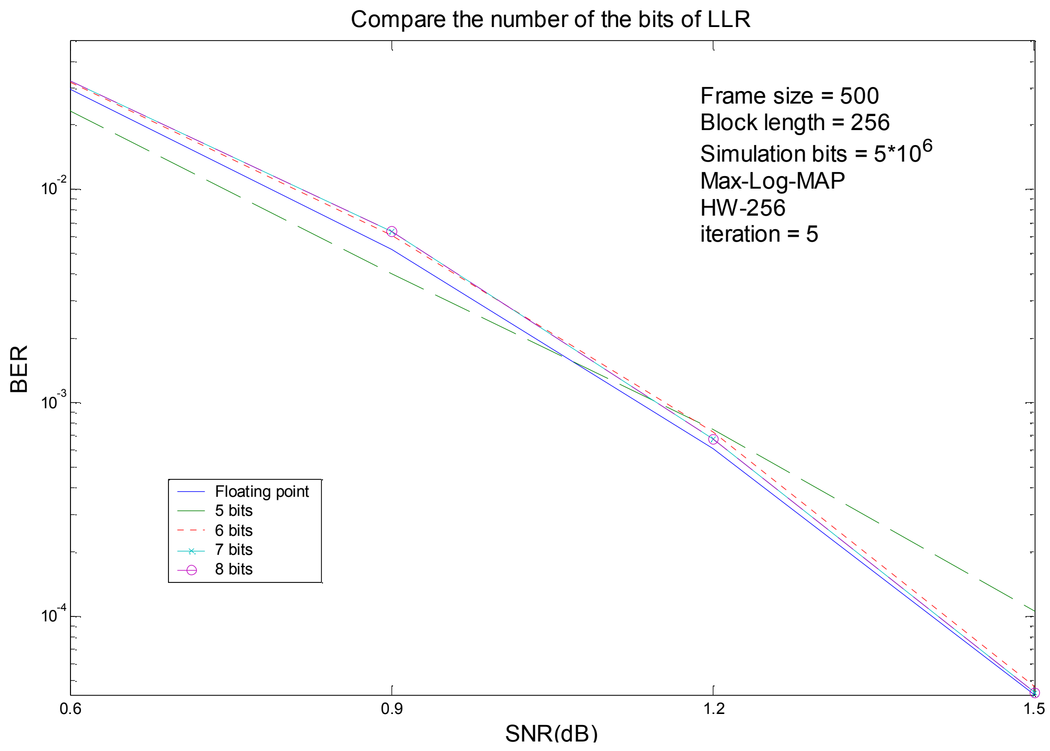


Figure 5.4: Simulations of various bit length of LLR

When the number of the integer bits exceeds 4, it does not improve the performance significantly. Therefore we use 4 bits to represent the integer part of the a-prior information and 3 bits for fractional part. Since the fixed point representation of $L(u_k)$ is decided, we can decide the branch metric q_{sm} as said in section 5.4.2. From the three terms on the right hand side in equation (34), their integer bits are all 4 bits but they are all divided by 2, therefore we only need to set q_{sm} to 5.

The extrinsic information $L_e(u_k)$ is used as the a-priori information for next decoder so that its fixed point representation is identical to the a-priori LLR, that is *FP(7,3)*.

5.4.5 a-posterior LLR $L(u_k|\underline{y})$

The equation of a-posterior LLRs $L(u_k|\underline{y})$ is rewritten below:

$$L(u_k|\underline{y}) \approx \max_{\substack{(s',s)\Rightarrow \\ u_k=+1}} \left(A_{k-1}(s') + \Gamma_k(s',s) + B_k(s) \right) - \max_{\substack{(s',s)\Rightarrow \\ u_k=-1}} \left(A_{k-1}(s') + \Gamma_k(s',s) + B_k(s) \right)$$

Each term on the RHS is computed as the sum of two state metrics and one branch metric and a-posterior LLRs equal to the difference of the two terms. Thus we can give it one more bit than the state metrics in order to prevent the occurrence of overflow. Then we use *FP(10,3)* to represent a-posterior LLR $L(u_k|\underline{y})$.

Through all the analyses, we arrange all the numbers in Table 5.3. By comparing to the fixed point analyses of [30], we know our design is a little bit conservative. Our design uses extra one fractional bit.

	integer bits (including sign bit)	fractional bits	total bits
Received value	4	3	7
Branch metric	5	3	8
Forward state metric	6	3	9
Backward state metric	6	3	9
LLR(u_k)	7	3	10
a-priori information	4	3	7
extrinsic information	4	3	7

Table 5.3: word length of our design

Using the word length in Table 5.3, we simulate the performance of our design and compare them to the performance of floating point.

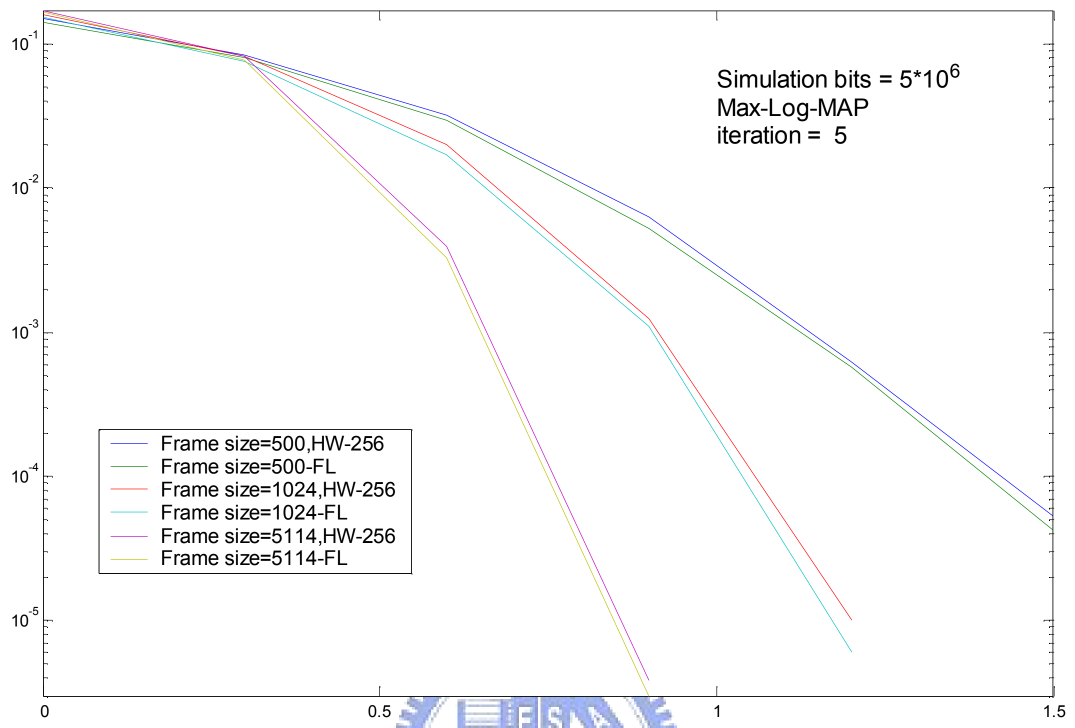


Figure 5.5 performance of our design comparing to floating point

Chapter 6
Hardware Architecture

In this chapter, we introduce our turbo decoder hardware architecture from the computational core of Max-Log-MAP decoding algorithm and discuss about all the units inside. The overall hardware architecture will be shown in the end of the first section. Then the decoding process will be presented.

6.1 Hardware architecture

The computational core of Max-Log-MAP decoder is composed of branch calculation unit, add-compare-select unit, a-posterior LLRs calculation unit and permutation units. The block diagram of the computational core is shown in Figure 6.1.

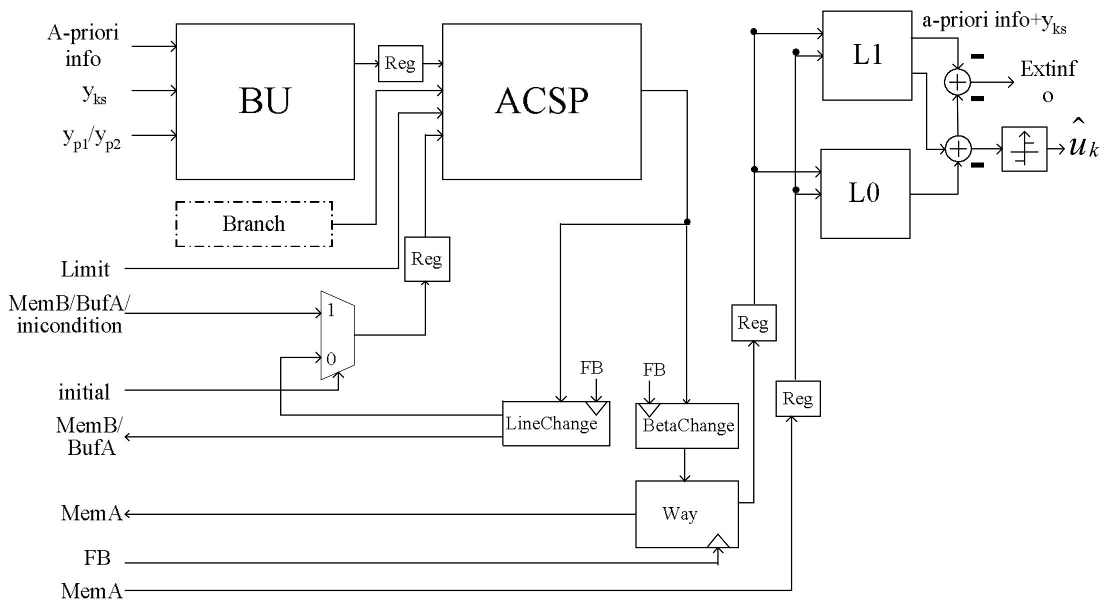


Figure 6.1: Computation core of the turbo decoder

All units will be discussed in detail as follows:

6.1.1 Branch Metrics Unit (BMU)

The branch metrics are calculated according to equation (34):

$$\Gamma_k(s',s) = \frac{1}{2}u_k \cdot L(u_k) + \frac{1}{2}\sum_{l=1}^{n} y_{kl}x_{kl}$$

The hardware for calculating branch metrics is shown in Figure 6.2. S0,S1,S2 in Figure 6.2 are relative to the sign of u_k, x_{k1}, x_{k2}, respectively. Because u_k, x_{k1}, x_{k2} have four combination i.e.000, 001,110,111, we need four BMUs.

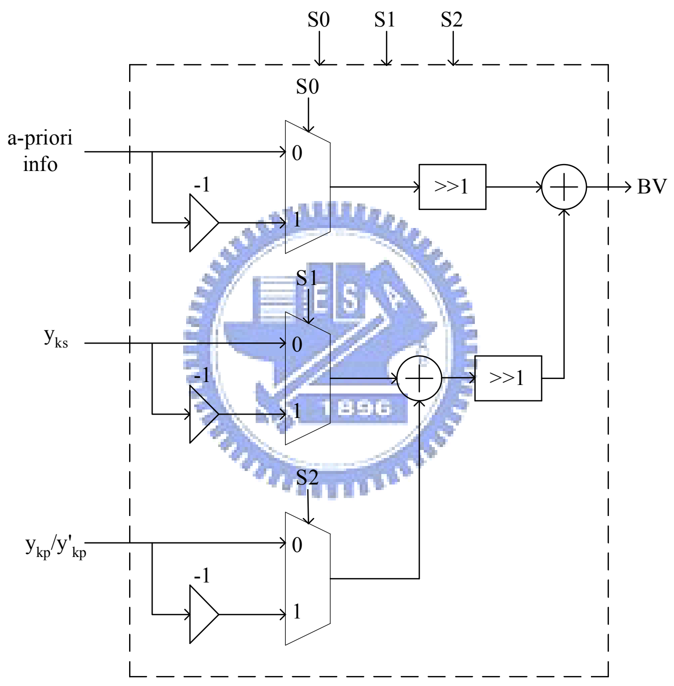


Figure 6.2: Branch metric unit

6.1.2 Add-Compare-Select Processor (ACSP)

The equations of forward state metrics $A_k(s)$ and the backward state metrics $B_{k-1}(s')$ are rewritten below. From the equations, we know that they are both calculated through adding, comparing and selecting the maximum value computations. The ACS processing element (ACSPE) hardware is shown in Figure 6.3.

$$A_k(s) = \max_{s'}\left(A_{k-1}(s') + \Gamma_k(s', s)\right)$$

$$B_{k-1}(s') = \max_{s}\left(B_k(s) + \Gamma_k(s', s)\right)$$

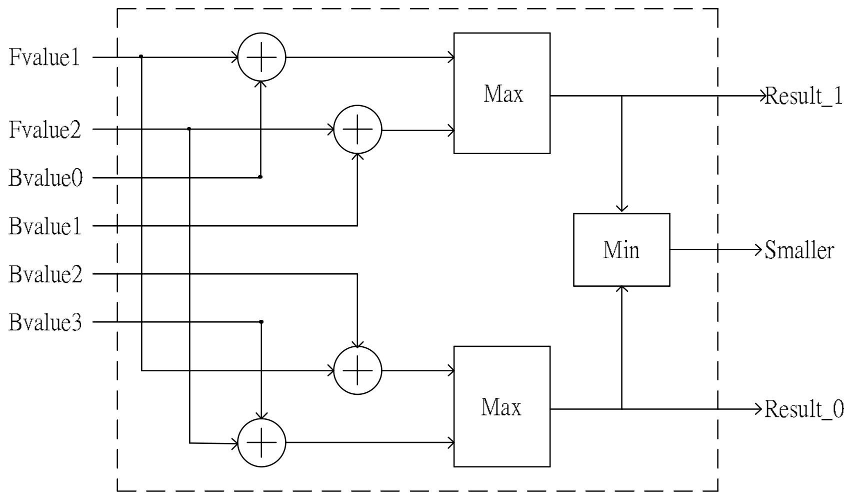


Figure 6.3: ACS processing element

Since 3GPP encoder has eight states, we need to combine eight ACSPEs to form an ACS processor in order to calculate one stage in one clock cycle. The eight ACSPEs and the corresponding trellis diagram are shown in Figure 6.4

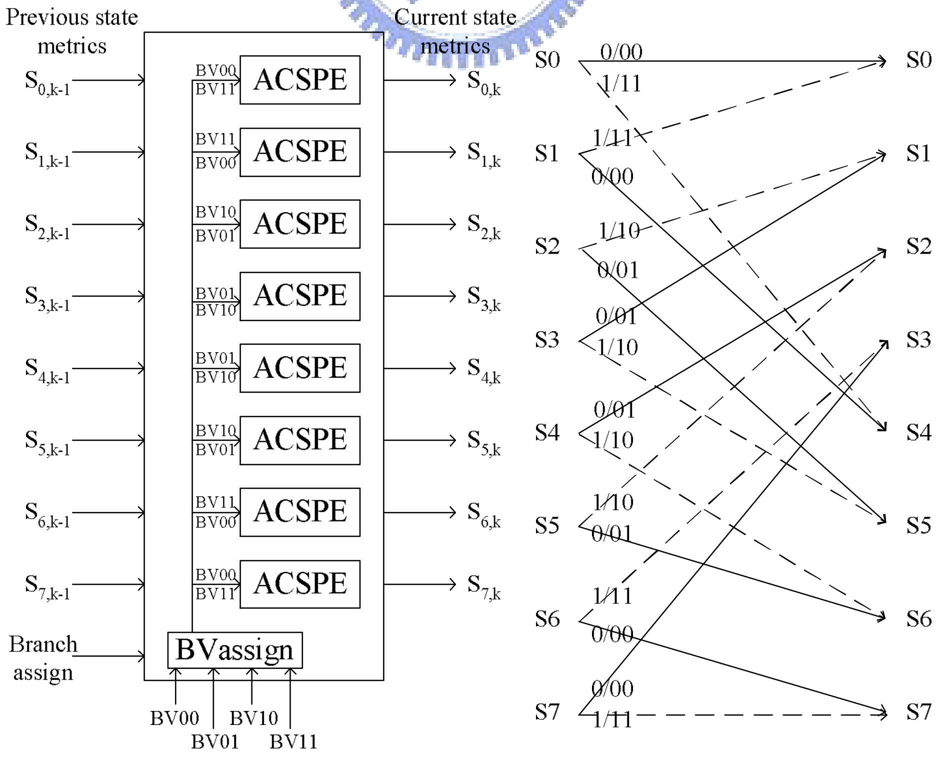


Figure 6.4: Bundled ACSPEs and the corresponding trellis

The ACS processor uses the initial value to compute the state metrics at first time. Then it will calculate by the value in the register on the feedback loop recursively. The value in the registers would increase time after time and the overflow will happen soon. Thus we place hardware to subtract the minimum value of the eight state metrics produced every time. In order to prevent the overflow, a saturation unit is placed behind it. The whole ACS processor (ACSP) and the feedback loop are shown in Figure 6.5.

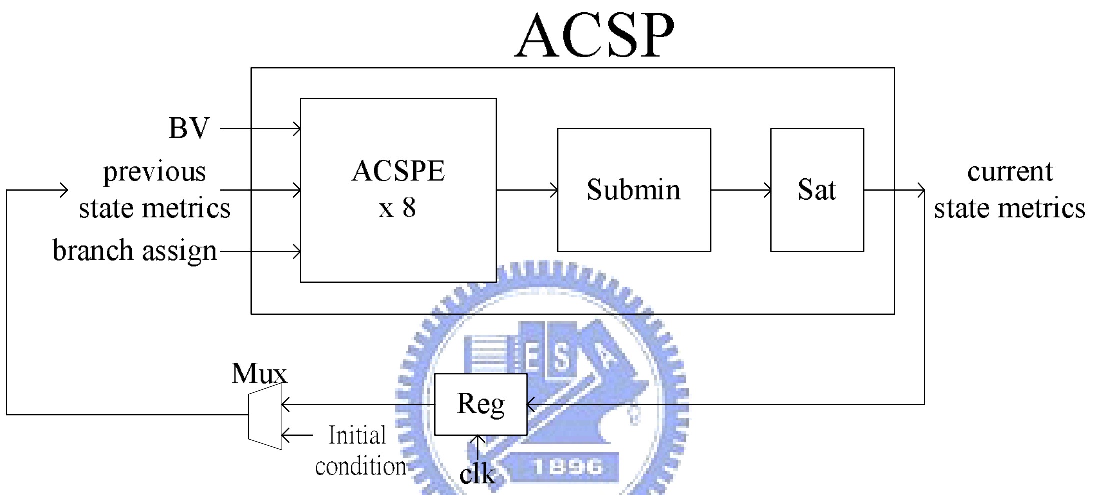
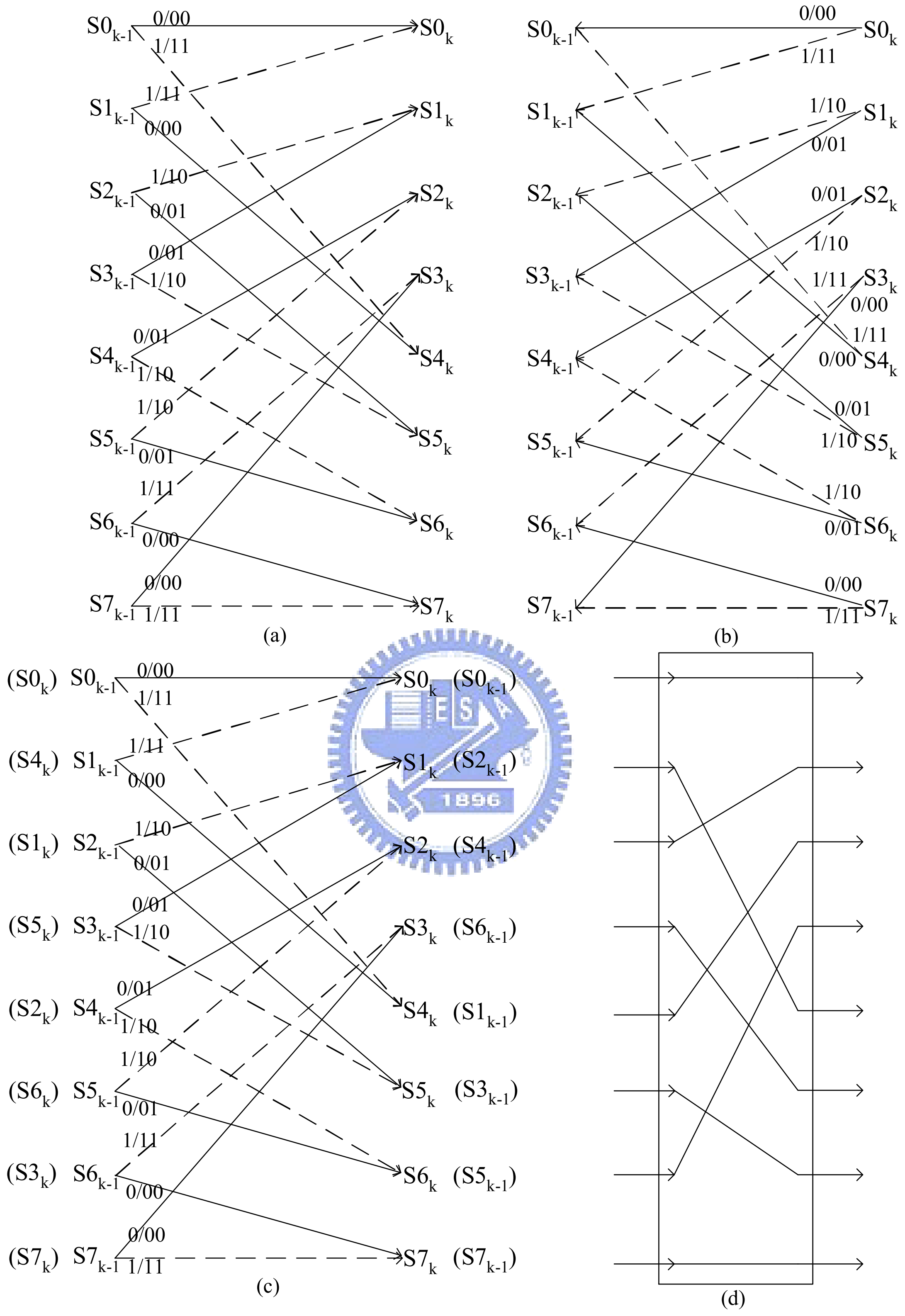


Figure 6.5: ACS processor and feedback loop

6.1.3 LineCchange unit and BetaChange unit

Using only one ACS processor means we need to calculate the forward metrics and backward metrics by the same ACS processor successively. Observing Figure 6.6(a)(b), we can find the only one difference between them is the direction so that we can design a permutation unit to rearrange the input addresses and output addresses for one ACS processor to compute for the forward metrics and backward metrics. In Figure 6.6(c), the addresses for calculating backward metrics are shown in the parenthesis and the permutation rules are shown in Figure 6.6(d).



(a)
(b)
(c)
(d)
S0k-1 S1k-1 S2k-1 S3k-1 S4k-1 S5k-1 S6k-1 S7k-1
S0k S1k S2k S3k S4k S5k S6k S7k
0/00 1/11 1/10 0/01
(S0k) (S4k) (S1k) (S5k) (S2k) (S6k) (S3k) (S7k)
(S0k-1) (S2k-1) (S4k-1) (S6k-1) (S1k-1) (S3k-1) (S5k-1) (S7k-1)

Figure 6.6: (a) trellis diagram for forward state metrics calculations. (b) trellis diagram for backward state metrics calculations. (c) trellis diagram for mapping backward trellis to forward trellis. (d) permutation rules

According to Figure 6.6(d), the permutation unit is shown in Figure 6.7(a); we call it “BetaChange” because it’s used to permute the address while computing backward metrics. We need to permute the input and output when calculating backward state metrics. Thus we design “LineChange” unit to perform these two permutations at one time and it is shown in Figure 6.7(b).

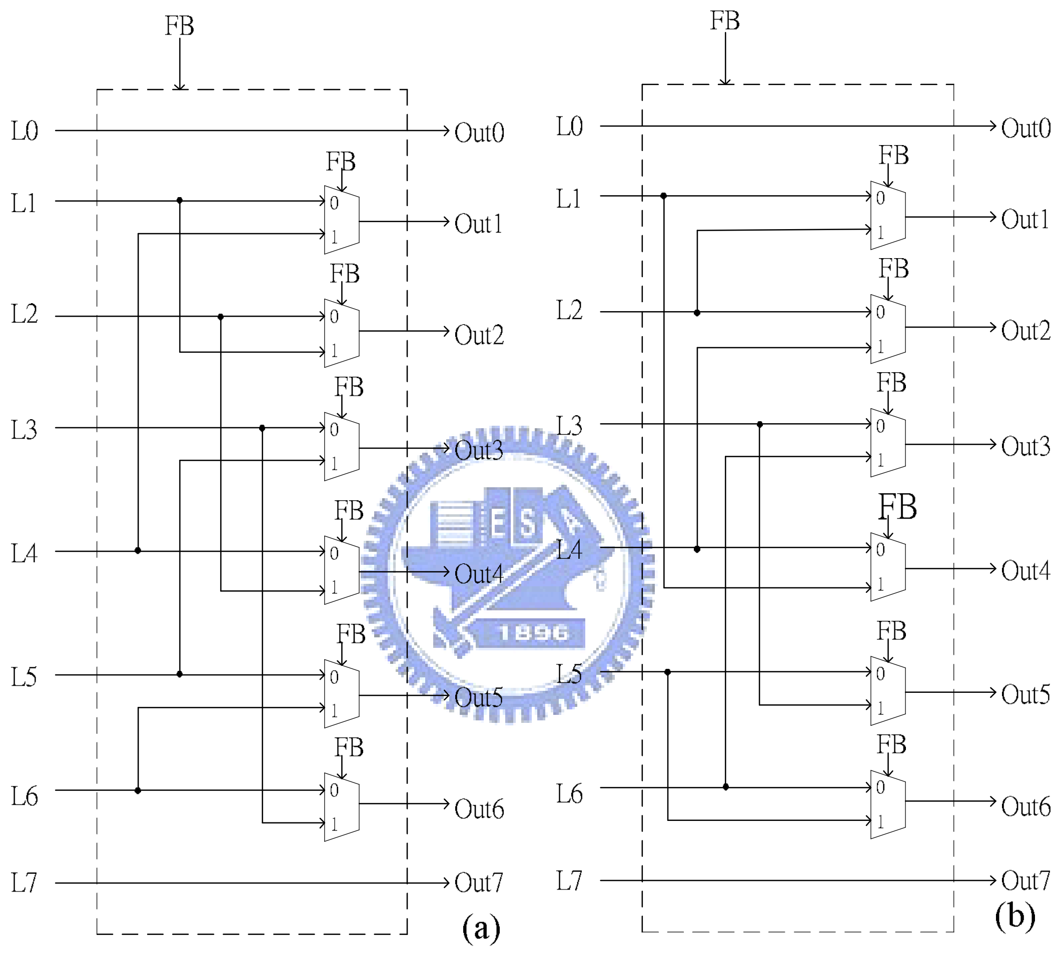


Figure 6.7: (a) BetaChange hardware (b) LineChange hardware

6.1.4 L0,L1 unit

The equation of a-posterior LLR $L(u_k|\underline{y})$ is rewritten below:

$$L(u_k|\underline{y}) = \max_{\substack{(s',s)\Rightarrow \\ u_k=+1}}\left(A_{k-1}(s') + \Gamma_k(s',s) + B_k(s)\right) - \max_{\substack{(s',s)\Rightarrow \\ u_k=-1}}\left(A_{k-1}(s') + \Gamma_k(s',s) + B_k(s)\right)$$
$$= L1 - L0$$

where $L1 = \max\limits_{\substack{(s',s)\Rightarrow \\ u_k=+1}} \left(A_{k-1}(s') + \Gamma_k(s',s) + B_k(s)\right)$, $L0 = \max\limits_{\substack{(s',s)\Rightarrow \\ u_k=-1}} \left(A_{k-1}(s') + \Gamma_k(s',s) + B_k(s)\right)$

Because there are eight paths correspondent to $u_k = \pm 1$ respectively, the max operations on the right-hand side means to select the maximum value from eight sums. We will use nesting max operations to implement. The hardware architecture of $L1$ processor is shown in Figure 6.8. The architecture of $L0$ processor is the same with $L1$.

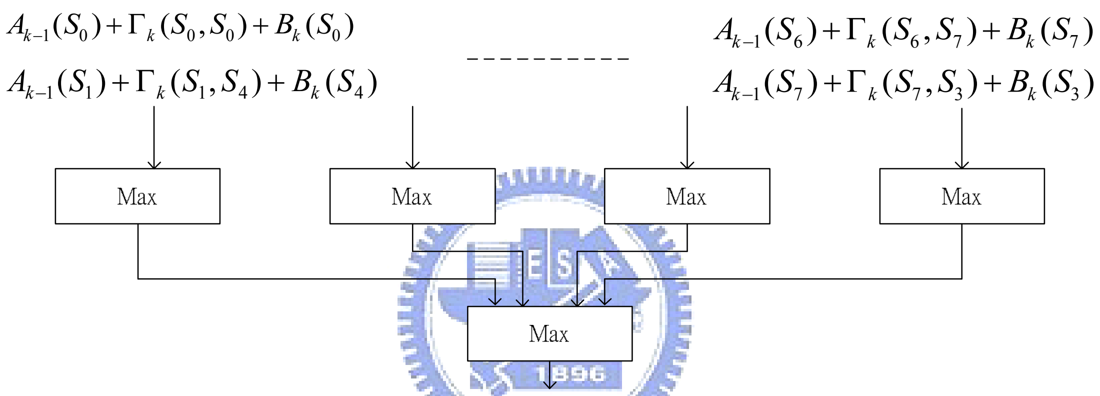

$A_{k-1}(S_0)+\Gamma_k(S_0,S_0)+B_k(S_0)$
$A_{k-1}(S_1)+\Gamma_k(S_1,S_4)+B_k(S_4)$
$A_{k-1}(S_6)+\Gamma_k(S_6,S_7)+B_k(S_7)$
$A_{k-1}(S_7)+\Gamma_k(S_7,S_3)+B_k(S_3)$
Max
Max
Max
Max
Max

Figure 6.8: Nesting max operations for L1 hardware

When $L1, L0$ are calculated, we can compute $L(u_k|\underline{y})$. If $L(u_k|\underline{y})$ is bigger than or equal to zero, the decoded bit is "1". If $L(u_k|\underline{y})$ is smaller than to zero, the decoded bit is "0".

6.1.5 Complete turbo decoder architecture

The block diagram of entire turbo decoder hardware is shown in Figure 6.9.

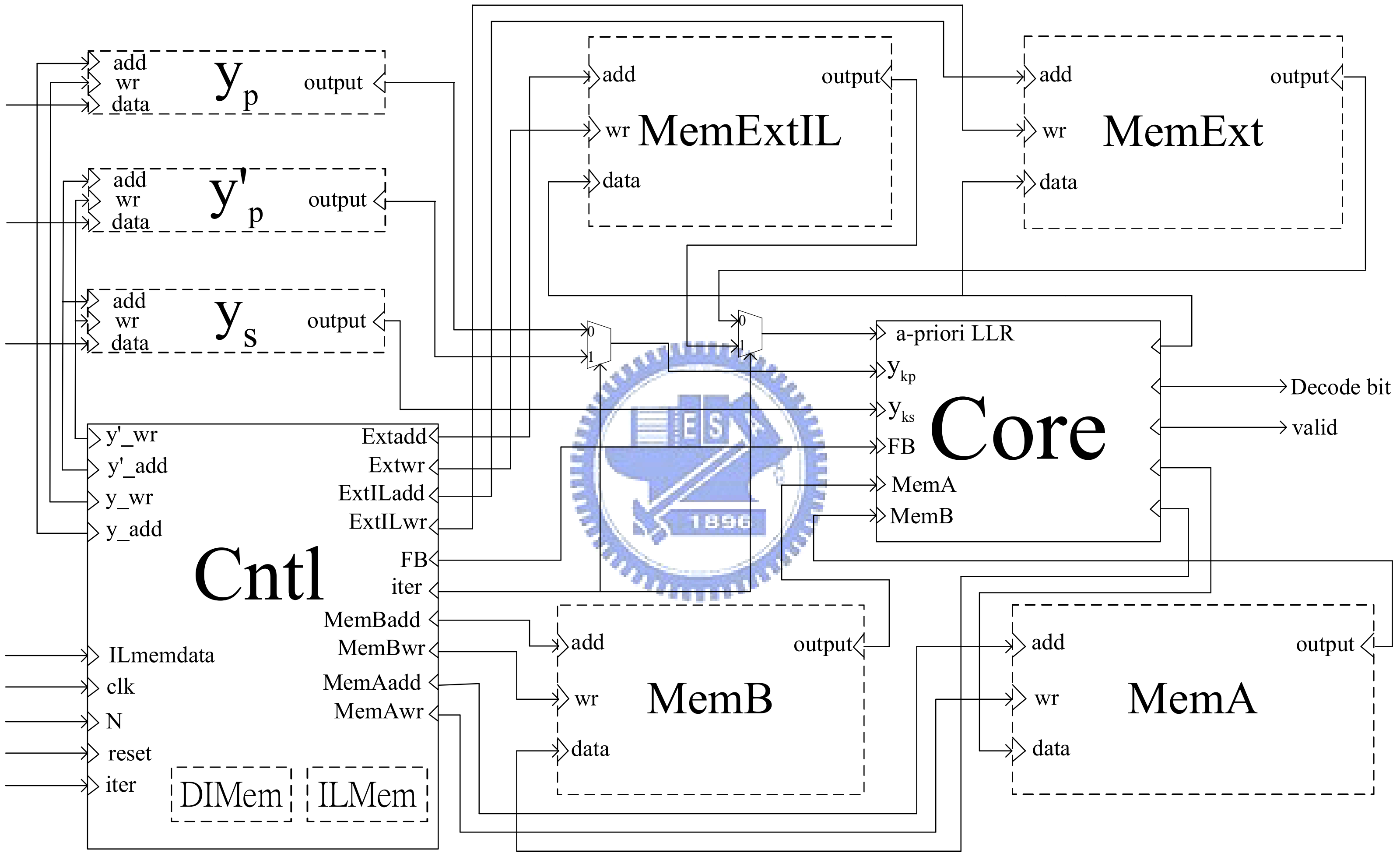


Figure 6.9: The block diagram of total turbo decoder hardware

The solid line rectangles are computational unit or control unit and the dotted line rectangles are memories. The functions of all blocks are as follows:

1. Cntl unit: Cntl unit is used to control where to write or read and when to write or read of all memories. We pack the interleaver/deinterleaver memory into it because the functions of them are to provide the addresses for mapping the normal order systematic bits y_{ks} to interleaved order systematic bits y'_{ks} and the address for writing extrinsic LLRs.

2. y_s,y_p,y'_p memories: These memories are used to store the transmitted systematic bits, the parity bits corresponding to the normal order information bits, the interleaved systematic bits, the parity bits corresponding to the interleaved order information bits, respectively. Their depths all equal to L_{HW} .

3. MemExtIL, MemExt memories: These two memories are used to provide and store a-priori LLR and the extrinsic information. In first half iteration, the decoder deal with the natural order received bits, y_s,y_p, and MemExt provides a-priori LLRs and MemExtIL receives the extrinsic information produced in this half iteration. In the last half iteration, the decoder deal with the interleaving order received bits y'_s,y'_p, MemExILt provides a-priori LLRs now and MemExt receives the extrinsic information produced in this half iteration.

4. MemA memory: MemA memory is used to store the backward state metrics when calculating backwardly and to provide the state metrics to L1, L0 processors which are in the computational core when the decoder calculates the forward state metrics.

5. MemB: MemB memory is the initialization memory required for halfway. It stores the backward state metrics periodically and provides these state metrics as the initialization state metrics for the corresponding backward calculations at next iteration.

6.2 Memory requirements

In previous section, we list all the memories needed in the decoder hardware. Along with the specification decided in section 5.3 and section 5.4, we can calculate all the memory capacities. We arrange all the numbers in Table 6.1.

		Bit width	memory capacity (bits) (depth*width*banks)	memory capacity (bytes)
Input memory_b	*ys*,*yp*,*y'p*	7	5120*7*3=107520	13440
Input memory_d	*ys*,*yp*,*y'p*	7	5120*7*3=107520	13440
Initialization memory	MemB	9	64*8*9=4608	576
Backward metrics memory	MemA	9	256*8*9=18432	2304
interleaver/deinterleaver	ILMem /DIMem	13	5114*13*2=132964	16620.5
Extrinsic info. Memory	MemExt /MemExtIL	7	5114*7*2=71596	8949.5
Total			442640	55330
Total (excluding input memory_b)			335120	41890

Table 6.1: memory capacity

The first column is the memory name with respect to their function and the second column is the corresponding name in the Figure 6.9. In the second and third row, there are two sets of memories for y_s, y_p, y'_p. The first memory, input memory_b, is used as buffer which stores data for the next block while decoding the data in memory_d. When finish decoding the data in memory_d, the roles of these two memories exchange. Usually the data buffer (memory_b) is not counted in the decoder hardware due to it does not provide data to the decoder in decoding process though it is essential. Therefore we list the memory capacity without buffer in the last row for reference.

6.3 Decoding Process

Before the decoding process starts, we need to initialize the interleaver and de-interleaver memory. Because generating the interleaving sequence needs many multiplications and divisions and look-up tables, we do not implement it in the hardware. Instead, we use software to calculate the interleaving sequence and input the sequence into the interleaver memory and de-interleaver memory before start to decode. So we input the interleaving sequence to the interleaver memory first. At the same time, we take the data input to the interleaver as the writing address and take the writing address of the interleaver as the input data for the deinterleaver memory. After initializing the interleaver/de-interleaver memories, the decoding process begins.

Assume the received data comprising one frame of information are in the input memory_d, frame size $= N$, block length $= L_{HW}$, block number $p = \lceil N / L_{HW} \rceil$. If $N \le L_{HW}$, this decoder works as the normal Max-Log-MAP decoder does. In order to explain the halfway decoding process, we further assume $N = p \cdot L_{HW}$ where $p \in \mathbb{N}$ then we can divide one frame into block-1, block-2… block-*p*, denoted as sb_i where $i \in 1, 2, ..., p$. Because 3GPP encoder has trellis termination, the encoded code has 12 tail bits corresponding to 6 trellis stages as stated in section 4.2. Hence the decoder will calculate backward state metrics by these tail bits first in order to process regularly hereafter. When we finish computing the tail bits, the backward state metrics are stored to MemB as the initialization state metrics for sb_p.

Now the first half iteration decoding begins. We use y_s, y_p to calculate backward state metrics first by for sb_i from $s_{i \cdot L_{HW}-1}$ to $s_{(i-1)L_{HW}}$ where $i = 1, 2, ..., p$. The initialization state metrics for each block are all set to zeros at this iteration except the last block. The state metrics which input to ACSP are saved to MemA and the last calculated state metrics are stored to MemB at address $(63 - i + p)$ when $i \neq 1$. Afterwards the forward state metrics are calculated for sb_i from $s_{(i-1)L_{HW}}$ to $s_{i \cdot L_{HW}-1}$ where $i = 1, 2, ..., p$. The state metrics which input to ACSP are sent to $L0$, $L1$ units with the relative backward state metrics stored in MemA. The extrinsic information can be computed and saved to MemExtIL according to the relative interleaving address. The decoding process in second half iteration is similar to first half iteration but is different from: using y'_s, y'_p to calculate the backward/forward state metrics; the last calculated state metrics of each block are stored to MemB at address $(31 - i + p)$ when $i \neq 1$; the computed extrinsic information is saved to MemExt according to the relative de-interleaving address.

When first iteration completes, MemB will have the initialization state metrics for backward state metrics for sb_i where $i = 1, 2, ..., p$. The graphic representation of the halfway SISO algorithm is shown in Figure 6.10.

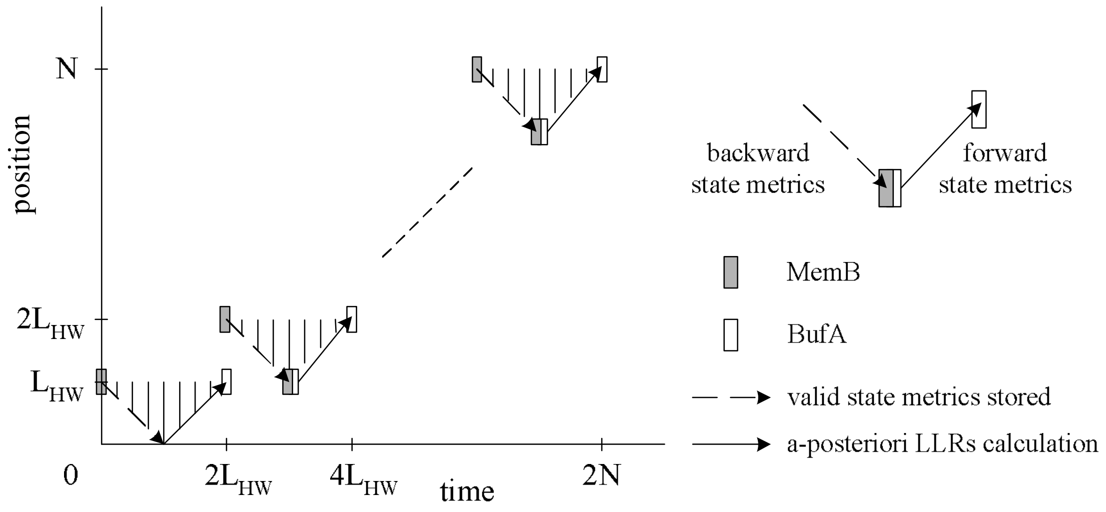


Figure 6.10: graphic representation of the halfway SISO algorithm

Although we assume $N = p \cdot L_{HW}$ where $p \in \mathbb{N}$ in the decoding process discussed above, N would not equal to multiples of L_{HW} in general. The derivations can easily be modified to apply to the general cases.

Chapter 7
Hardware implementation

With modern VLSI technology, we can design the hardware with high clock rate and complicated functions. There are two design abstractions: Bottom-up and top-down. By using the abstractions, the designer can collapse details and arrive at a simpler concept with which to deal.

In the design process of integrated circuit, the layout techniques are very amateur so that we can use Computer-aided design (CAD) tool to help us to place and route. Nowadays most of the digital communication integrated circuits adopt the standard cell design instead of full custom design. Therefore the emphasis is put on the algorithms and the hardware architectures. In this thesis we also adopt the standard cell to design the hardware.

7.1 Design and verify process

First we write a C program to simulate the decoding algorithm so that we can understand the flow of the decoding process. And we can verify the C program by examining a lot of data.

Second we plan the hardware architecture. In this thesis, we implement the decoder by halfway memory saving method. Thus we can achieve the 3GPP requirement by using only one ACS processor without high operation frequency. Then we develop a bit-accurate C model according to the above architecture. Because we use fixed-point implementation, we could analyze the word length of the quantities by this bit-accurate C model. It is easier to modify the word length and the architecture in C code than in HDL code. If we find the specification can not satisfy our objective, we could redesign the architecture or change the word length easily and quickly. Besides, C model can help us to process HDL debugging easily.

Third we can proceed to RTL verification. When the functions of the RTL code work correctly, we can synthesize the code with synthesis tools. If the synthesis result could not satisfy our requirement, we need to modify the architecture and repeat the flow from bit-accurate C model.

Finally, if the synthesis result achieves the requirement, we can download the RTL code to FPGA develop board. Afterward we verify the hardware circuit by inputting a lot of data.

In summary, our develop and design flow is shown in Figure 7.1

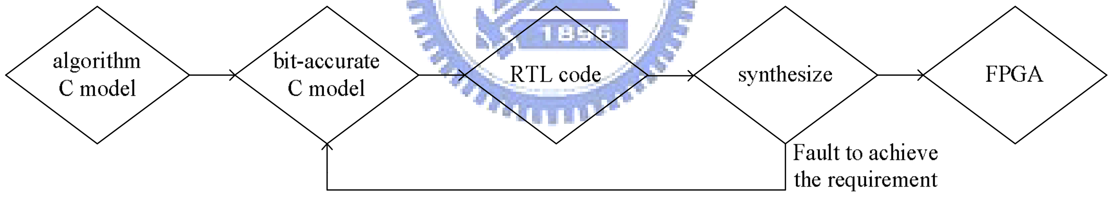


Figure 7.1: develop and design flow

7.2 Hardware specification

In this section, we will describe the clock cycles for decoding one block of data first. Then we define the hardware input and output ports clearly.

7.2.1 Clock cycles for decoding one data frame

The clock cycles for decoding one data frame are dependent on the frame size. Our hardware is pipelined into five stages. Thus the internal latency is 5 clock cycles.

The total required clock cycles for decoding one data frame are calculated as follows:

$$clock\_cycles_{N,Iter} = 2 \cdot \left(2 \cdot frame\_size + internal\_delay\right) \cdot Iter + 6 \qquad (35)$$

The subscript of clock cycles "*N*" stands for the frame size of the data for simplicity. The first term in the inner parentheses "*frame_size*" is also the frame size of the data. The term "Iter" is the number of complete decoding iteration. The last term "6" is the clock cycles for calculating the tail bits.

Since the frame size of 3GPP turbo code ranges from 40 to 5114, we list some examples as follows:

Iteration = 5

Frame size	40	500	1024	5114
clock cycles	856	10056	20536	102336

Iteration = 10

Frame size	40	500	1024	5114
clock cycles	1706	20106	41006	204666

Table 7.1: decoding clock cycles for different frame size

When frame size is small, the internal delay will affect the decoding cycles severely.

7.2.2 Hardware interface

For convenience, we pack the decoder as a processing core and indicate the input/output ports in Table 7.2. When this processing core is used, we only need to configure the pins adequately. The I/O diagram of this processing core is shown in Figure 7.2.

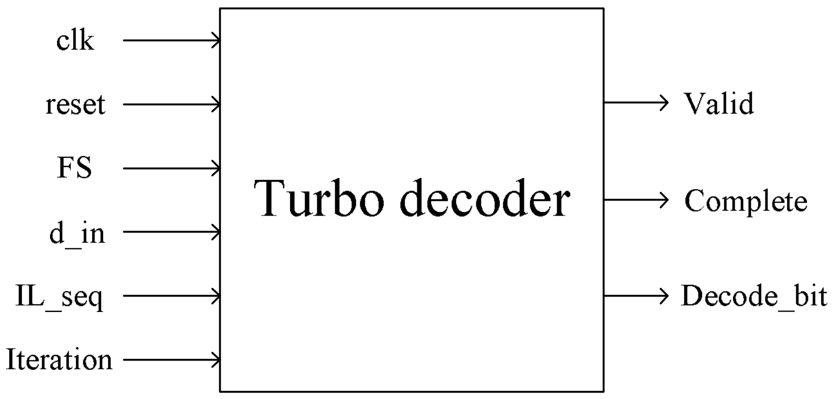


Figure 7.2: Turbo deocder I/O diagram

Port	i/o	bit width	description
clk	input	1	system clock
reset	input	1	reset the register contents
FS	input	13	configure the frame size of data
d_in	input	7	received data input
IL_seq	input	13	interleave sequence input
Iteration	input	5	configure the iteration number
Valid	output	1	indicate the decode bit valid
Decode_bit	output	1	decode bit output
Complete	output	1	indicate finish decoding one block of data

Table 7.2: I/O ports definition

7.3 ASIC performance

We are interested in how many gate counts are in the turbo decoder hardware. So we will divide the turbo decoder into two part, one is memory part and the other is control and computation part. The ASIC verification flow is shown in Figures 7.3. We use MATLAB to generate the encoded sequence and the additive white Gaussian noise and write the information into test bench. We can compare the results with the decoding bits by bit-accurate C decoding program. If “Out_cp” outputs “1”, there should be something wrong in the decoder hardware.

The ASIC simulation environment is as follows:

HDL: verilog

Compiler tool: verilog-XL

Debug tool: Debussy

Synthesis tool: synopsys

Process: TSMC 0.25 μm

The simulation results are listed in Table 7.3. The maximum clock rate for this decoder is 102.56MHz.

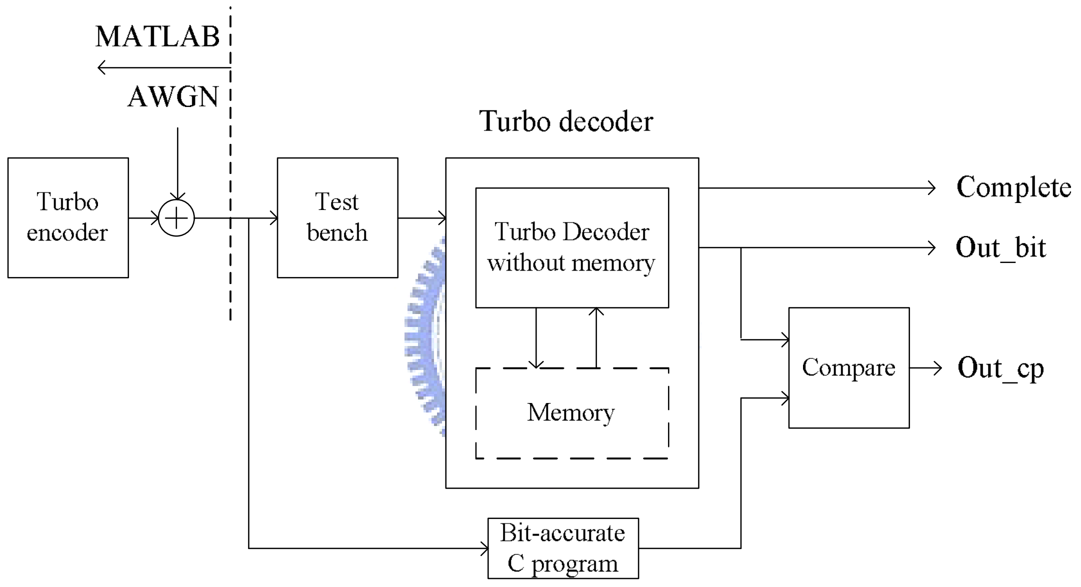


Figure 7.3: ASIC verification flow

Constraint	9.75ns	10ns	12.5ns	25ns
Clock rate	102.56MHz	100MHz	80MHz	40MHz
Gate counts	28.7k	28.1k	24.8k	15.1k

Table 7.3: ASIC simulation results

Along with equation (35) in section 7.2.1, we can calculate the clock rate required for decoding the data. Assume required output data rate = R_d, frame size = N and iteration number = $Iter$, we can get:

$$required\ clock\ rate_{N,Iter} = \frac{R_d}{N} \cdot clock\_cycles_{N,Iter}$$

In 3GPP, maximum R_d is 2 Mbps, thus required clock rate is:

clock rate	Iter=5	Iter=10
N=40	42.8	85.3
N=5114	40.02	80.04

Table 7.4: required clock rate for decoding different frame size and iteration

Because our hardware has maximum operation frequency 102.56 MHz, it can meet 3GPP requirement.

7.4 FPGA verification

We use MATLAB to generate the encoded sequence and the additive white Gaussian noise. We use the bit-accurate C decoder to decode the received sequence and write the decoding results into a file. Then we put the received information into ROM of the turbo decoder and compare the decoding results with those generating by the bit-accurate C decoder. The output bit and the comparison results are displayed in the seven-segment display. The FPGA verification flow is shown in Figure 7.4.

The simulation environment is as follows:

FPGA development board: Altera stratix II EP1S25780C5

Simulation software: Quartus II 4.0

HDL: verilog

Max. clock rate = 40.2 MHz

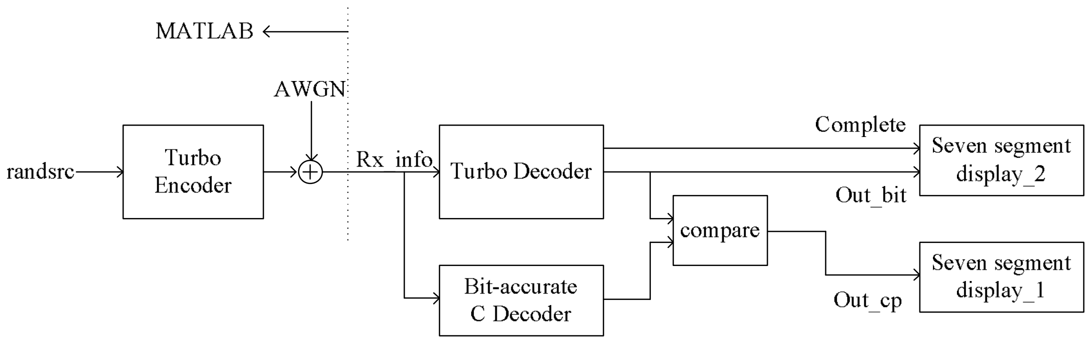


Figure 7.4: FPGA verification flow

Chapter 8
Conclusion and Future works

8.1 Conclusion

In this thesis, we implement an efficient and memory saving 3GPP turbo decoder which uses the halfway method. This decoder bases on Max-Log-MAP algorithm and uses only one ACS processor. This successfully decreases the memory capacity which is the critical design problem for turbo decoders. It also discards the redundant calculations for initializations which are required for other decoding methods. As a result, using only one ACS processor in our decoder will not slow down the decoding speed. Furthermore, using halfway memory saving method in the decoder can decrease the decoding latency. By use of the computer simulation and the analyses, we decide the fixed point representations and the block length for halfway method in order to obtain a cost-effective turbo decoder. We compare the BER performance of halfway with the commonly-used sliding window schemes and confirm that our approach does not sacrifice any performance.

8.2 Future works

Our hardware design still can be improved in 3 aspects:

1. Decoding speed: Though our decoder hardware can satisfy the maximum decoding speed of 3GPP specification, 2M bits/s, by 5 iterative decoding at 40.2 MHz operation frequency, the need for more iterations and faster decoding speed

will still exist in the future. Therefore we can use one more ACS processor to calculate forward state metrics when the original ACS processor calculates backward state metrics at the same time. This will boost the decoding speed by a little overhead and hardware requirement.

2. Stopping criterion: we do not implement any stopping criterion on our decoder, thus the decoder will decode for fixed number of iterations. This results in consuming energy unnecessary and wasting the decoding time.
3. Embedded interleaver/de-interleaver generator: At the moment we assume the interleave/de-interleaver data are stored to the memory and these will cost a lot of memory. If we can design the hardware for generating interleaving/de-interleaving sequence when needed immediately, it will decrease the memory capacity needed by decoder significantly. More exactly, that is 2x13x5114 = 132964 =132.9 K bits = 16.6125 Kbytes.

References

[1] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, pp. 379-427, 1948.

[2] R. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technical Journal*, vol. 29, pp. 147-160, 1950.

[3] E. Prange, "Cyclic Error-Correcting Codes in Two Symbols," *Air Force Cambridge Research Center*-TN-57-103, Cambridge, MA: September 1957.

[4] E. Prange, "Some Cyclic Error-Correcting Codes with Simple Decoding Algorithms," *Air Force Cambridge Research Center*-TN-57-103, Cambridge, MA: September 1957.

[5] E. Prange, "The Use of Coset Equivalence in the Analysis and Decoding of Group Codes," *Air Force Cambridge Research Center*-TN-57-103, Cambridge, MA: September 1957.

[6] A. Hocquenghem, "Codes correcteurs d'erreurs," *Chiffres* (*Paris*), vol. 2, pp. 147-156, September 1959.

[7] R. Bose and D. Ray-Chaudhuri, "On a Class of Error Correcting Binary Group Codes," *Information and Control*, vol. 3, pp. 68-79, March 1960.

[8] R. Bose and D. Ray-Chaudhuri, "Further Results on Error Correcting Binary Group Codes," *Information and Control*, vol. 3, pp. 279-290, September 1960.

[9] I. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *Journal of the Society of Industry and Applied Mathematics*, vol. 8, pp. 300-304, June 1960.

[10] P. Elias, "Coding for Noisy Channels," *IRE Convention Record*, pt. 4, pp. 37-47, 1955.

[11] J. Wozencraft, "Sequential Decoding for Reliable Communication," *IRE Natl.*

Conv. Rec., vol. 5, pt.2, pp. 11-25, 1957.

[12] J. Wozencraft and B. Reiffen, “Sequential Decoding,” *Cambridge, MA, USA: MIT Press*, 1961.

[13] A. Viterbi, “Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm,” *IEEE Transactions on Information Theory*, vol. IT-13, pp. 260-269, April 1967.

[14] G. Forney, “The Viterbi Algorithm,” *Proceedings of the IEEE*, vol. 61, pp. 268-278, March 1973.

[15] G. Ungerboeck, “Trellis-Coded Modulation with Redundant Signal Sets part I: Introduction,” *IEEE Communications Magazine*, vol. 25, pp. 5-11, February 1987.

[16] G. Ungerboeck, “Trellis-Coded Modulation with Redundant Signal Sets part II: State of the art,” *IEEE Communications Magazine*, vol. 25, pp. 12-21, February 1987.

[17] C. Berrou, A. Glavieus, and P. Thitimajshima, “Near Shannon Limit Error-Correcting Coding and Decoding: Turbo Codes,” *Proceedings of the International Conference on Communications*, (Geneva, Switzerland), pp. 1064-1070, May 1993.

[18] C. Berrou, C. Douillard, M. Jezequel, “Designing Turbo Codes for Low Error Rates", *IEE Workshop*, London, UK, December 1999.

[19] David J. C. MacKay, “Information Theory, Inference and Learning Algorithms,” *Cambridge University Press*, pp. 576, September 2003.

[20] Divsalar, D. and Pollara, F., “Turbo Codes for PCS Applications,” *Proceedings of International Conference on Communications*, Seattle, WA., pp. 54-59, June 1995.

[21] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, “Optimal Decoding of Linear Codes for Minimising Symbol Error Rate,” *IEEE Transactions on Information*

Theory, vol. 20, pp. 284-287, March 1974.

[22] J. Hagenauer and P. Hoher, "A Viterbi Algorithm with Soft-decision Outputs and its applications," *IEEE Globe-com*, pp. 1680-1686, 1989.

[23] P.Robertson, E. Villebrun, and P. Hoher, "A Comparison of Optimal and Sub-Optimal MAP Decoding Algorithms Operating in the Log Domain," *Proceedings of the International Conference on Communications*, (Seattle, USA), pp. 1009-1013, June 1995.

[24] A. Worm, P. Hoeher, and N. Wehn, "Turbo-Decoding Without SNR Estimation," *IEEE Communications Letter*, vol. 4, pp. 193-195, June 2000.

[25] T. A. Summers and S. G.Wilson, "SNR Mismatch and Online Estimation in Turbo Decoding," *IEEE Trans. Communications.*, vol. 46, pp. 421–423, April 1998.

[26] Peter H-Y Wu, "On the Complexity of Turbo Decoding Algorithms," *IEEE Vehicular Technology Conference*, spring 2001.

[27] A. Viterbi, "An Intuitive Justification and Simplified Implementation of MAP Decoder for Convolutional Codes," *IEEE Select. Areas in Communication*, vol. 16, pp. 260-264, February 1998.

[28] F. Raouafi, A. Dingninou, C. Berrou, "Saving Memory in Turbo Decoders using the Max-Log-MAP Algorithm," *IEE Colloquium*, pp. 14/1-14/4, November 1999.

[29] http://www.3gpp.org, "Multiplexing and channel coding," TS 25.212 v 6.2.0.

[30] T. K. Blankenship, B. Classon, "Fixed-point Performance of Low-complexity Turbo Decoding Algorithms," *IEEE Vehicular Technology Conference*, spring 2001.