On a turbo decoder design for low power dissipation

By
Jia Fei

Dissertation submitted to the Faculty
of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Electrical Engineering

Dr. Dong S. Ha, Chairman
Dr. F. Gail Gray
Dr. Joseph G. Tront
Dr. Brian D. Woerner

6 July 2000
Blacksburg, Virginia

Keywords: Turbo decoder, Log-MAP, Log-likelihood ratio,
Branch metric, State metric, Low power, Synopsys
Copyright 2000, Jia Fei

On a turbo decoder design for low power dissipation

Jia Fei
Dr. Dong S. Ha, Chairman
Bradley Department of Electrical and Computer Engineering.

(Abstract)

A new coding scheme called "turbo coding" has generated tremendous interest in channel coding of digital communication systems due to its high error correcting capability. Two key innovations in turbo coding are parallel concatenated encoding and iterative decoding. A soft-in soft-out component decoder can be implemented using the maximum *a posteriori* (MAP) or the maximum likelihood (ML) decoding algorithm. While the MAP algorithm offers better performance than the ML algorithm, the computation is complex and not suitable for hardware implementation. The log-MAP algorithm, which performs necessary computations in the logarithm domain, greatly reduces hardware complexity. With the proliferation of the battery powered devices, power dissipation, along with speed and area, is a major concern in VLSI design. In this thesis, we investigated a low-power design of a turbo decoder based on the log-MAP algorithm. Our turbo decoder has two component log-MAP decoders, which perform the decoding process alternatively. Two major ideas for low-power design are employment of a variable number of iterations during the decoding process and shutdown of inactive component decoders. The number of iterations during decoding is determined dynamically according to the channel condition to save power. When a component decoder is inactive, the clocks and spurious inputs to the decoder are blocked to reduce power dissipation. We followed the standard cell design approach to design the proposed turbo decoder. The decoder was described in VHDL, and then synthesized to measure the performance of the circuit in area, speed and power. Our decoder achieves good performance in terms of bit error rate. The two proposed methods significantly reduce power dissipation and energy consumption.

Acknowledgements

I would like to first thank my committee chairman and advisor, Dr. Dong S. Ha. It was through his patience and invaluable guidance that this work was accomplished. I would also like to express my appreciation for Dr. F. Gail Gray, Dr. Joseph G. Tront, and Dr. Brian D. Woerner for serving as my committee members and commenting on this work.

I am extremely grateful to Dr. James R. Armstrong for his expert advice on this work. I would like to acknowledge the support of our department system administrator Mr. Harris. Without his help none of this would have been finished on time.

Next, I would like to thank Jos Sulistyo, Carrie Aust, Suk Won Kim and all the other students of VTVT (Virginia Tech VLSI for Telecommunications) for their guidance and help. Special thank should be given to my coworker and good friend Meenatchi Jagasivamani. During the past two years, we had a good time working together in the lab and having fun after work.

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

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Third generation (3G) mobile communication systems aim to provide a variety of different services including multimedia communication. This requires digital data transmission with low bit error rates. However, due to the limitation of the battery life of wireless devices, transmitted power should remain as low as possible. Low power makes the system more susceptible to noise and interference. Error control coding is thus used to increase the noise immunity of the communication systems. As a result, the transmitted power can be lowered to obtain the same bit error rate compared with a system without an error correction coding scheme. The difference in power is called *coding gain*, which is a measure of the effectiveness of the coding scheme. In a power-limited circumstance, such as mobile communication, the error control coding scheme with high coding gain is preferred to enable the system to work at a relatively low energy to noise ratio (E_b/N_0). One-dimensional convolutional encoding and soft-decision Viterbi decoding are often applied in wireless communication systems [9]. Recently, a new coding scheme called " turbo coding", which stems from convolutional coding, has been adopted in the 3G mobile communication system due to its high coding gain and reasonable computation complexity.

Turbo coding was initially introduced by Berrou, Glavieux and Thitimasjshima in 1993 [1]. It was reported to provide error control performance within a few tenths of a dB of the Shannon's limit. There are two key innovations in turbo coding: parallel concatenated encoding and iterative decoding (p. 7 of [48]). Parallel concatenated encoders consist of two or more component encoders for convolutional codes. Decoding is performed iteratively. The output of the first decoder is permutated and fed to the second decoder to form one cycle of the iteration. Each systematic code is decoded using a soft-in soft-out (SISO) decoder. A SISO decoder can be implemented using the maximum *a posteriori* (MAP) or maximum likelihood (ML) decoding algorithm. While the MAP algorithm is optimal, the computation is complex and not suitable for hardware

implementation. By performing the MAP algorithm in the logarithm domain, the complexity is greatly reduced and made it amenable for hardware implementation.

With its high BER versus E_b/N_0 performance, turbo codes attract much attention since its introduction in 1993. Many researchers have investigated to understand the structure and principle of turbo coding, including the performance of SISO decoders, interleaving methods, and error probability bounds [20] - [25]. Besides that, some hardware implementations either on FPGAs or ASICs have also reported in [7], [8], [10], [14], [16].

In digital VLSI design, speed and silicon area used to be the two most important concerns. Recently, with the booming of portable devices such as cellular phones, camcodecs and laptop computers, power consumption becomes another important factor. Power dissipation can be classified into two categories, static power and dynamic power dissipation. Typically, dynamic power counts for 80%-90% of the total power consumption in a full static CMOS circuit. Numerous techniques have been proposed to reduce dynamic power dissipation [40], [48]. The techniques can be applied at different levels of digital design, from system level, down to switch level. In [8], Hong, Yi, and Stark proposed a way to stop the iteration of turbo decoding by comparing the outputs of the two component decoders and incorporated a power-down mode to save power. Many other proposals employed in low-power Viterbi decoder design such as [17] - [19] may be also applicable to turbo decoder design for low power dissipation.

In this thesis, a low-power design of a turbo decoder at the gate level in standard cell design environment is proposed. The standard cell design procedure starts from a behavioral VHDL description of the circuit. It is synthesized to generate a gate level design using Synopsys tools. The gate-level design can be imported into a place and route tool to generate a layout of the design. The advantages of a standard cell based design over full custom design are faster turn around time for the design, ease in design verification and more accurate modeling of the circuit. In order to achieve low power dissipation, we introduce low-power techniques into the behavioral description of a turbo

decoder. We add small control logic to realize flexible number of iterations in turbo decoding based on the channel condition. When channel condition is good, a smaller number of iterations may be sufficient to save power. Since two serially concatenated log-MAP decoders take turn to generate estimate information, there is always one idle decoder. Clock gating technique is added to minimize the signal switching activity. For our turbo decoder, we also incorporated blocking of spurious inputs to save power. In our experiments, power dissipation was estimated on the basis of the switching activity measured through logic simulation. Experimental results indicate that a variable number of iterations and clock gating reduces power dissipation significantly.

The organization of the thesis is as follows: Background on the operation of turbo encoders and decoders is provided in Chapter 2. A brief description of low-power design techniques investigated in this thesis is also covered in this chapter. Chapter 3 proposes a low-power design for turbo decoders. Design of a component decoder is described first, and then the overall design of a turbo decoder is discussed. Several relevant low power design techniques are presented last. Chapter 4 presents experimental results on our turbo decoder. Bit error rates for different numbers of iterations at different E_b/N_0s are presented, and observations made from the experiment are discussed. The area and speed of the synthesized turbo decoder circuits are reported. Chapter 5 concludes the thesis.

Chapter 2

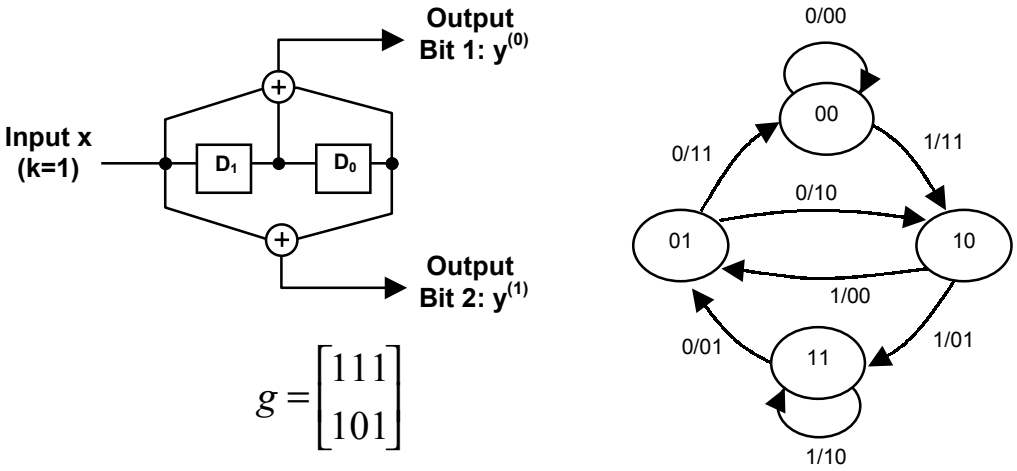
Turbo Encoding and Decoding Algorithm and Low-power Design Techniques

In this section, we provide necessary background on our research for a low-power design of Turbo decoders based on recursive systematic convolutional (RSC) codes. First, the concept of the RSC code is explained. Then, we present the turbo encoding and decoding algorithm and the hardware implementation of the algorithm. The goal of our design is to achieve low-power dissipation; therefore techniques for low-power design are also included.

2.1 Recursive Systematic Convolutional Codes

Turbo code was originally introduced by Berrou, Glavieux, and Thirimajshima in their paper "Near Shannon Limit Error Correction Coding and Decoding: Turbo-Codes" in 1993 [1]. In this paper, they quoted a bit error rate (BER) performance very close to Shannon's theoretical limit, generating tremendous interest in the field. The parallel concatenation is one of the two concepts critical to this new coding scheme. At the transmitter side, a parallel concatenated encoder uses two or more component encoders in conjunction with an interleaver to realize some long codes in an efficient manner. In this section, we review the concept of convolutional codes and discuss RSC encoders that are employed in turbo encoding.

Convolutional coding can be applied to a continuous input stream, offering an alternative to block codes for transmission over a noisy channel. An encoder with feedback loop generates recursive code, which has infinite impulse response (IIR), while a feedback free encoder has a finite impulse response (FIR). A systematic code is that one of the outputs is the input itself. A commonly used convolutional encoder is a binary non-recursive non-systematic convolutional encoder, such as the one shown in Figure 2.1. The outputs are computed as a linear combination of the current input and a finite number of past inputs.



$$g = \begin{bmatrix} 111 \\ 101 \end{bmatrix}$$

Figure 2.1: Example convolutional encoder

The encoder in Figure 2.1 has two memory elements (flip-flops) and produces two bits of encoded information for each bit of input information, so it is called binary rate ½ convolutional encoder with constraint length K=3. A convolutional encoder is generally characterized in (n, k, K) format, where

n is the number of outputs of the encoder;

k is the number of inputs of the encoder;

K is constraint length, which is the number of memory elements + 1;

The rate of an (n, k, K) encoder is k/n. For example, the encoder shown in Figure 2.1 is a (2,1,3) encoder with rate ½.

A convenient means to relate the output of a convolutional encoder to its input is through the generator matrix G. For the example encoder in Figure 2.1, the generator matrix is

$$\boldsymbol{G} = \begin{bmatrix} g^{(0)} \\ g^{(1)} \end{bmatrix} = \begin{bmatrix} 111 \\ 101 \end{bmatrix}$$

The two elements of the generator matrix $g^{(0)}$ and $g^{(1)}$ are also known as interconnection function as they represent the connections between the memory cells and the XOR gates.

The code word Y corresponding to an information sequence x is obtained through the generator sequence:

$$y_i^{(j)} = \sum_{l=0}^{K-1} x_{i-l} g_l^{(j)} \bmod 2,$$

where $Y = (y^{(0)}, y^{(1)})$ and k is the constraint length.

A convolutinional encoder is a Mealy machine, where the output is a function of the current state and the current input. Graphical techniques, which include the state diagram and the trellis diagram, can be used to analyze convolutional codes. A state machine shown in Figure 2.1 depicts state transitions and the corresponding encoded outputs. Since there are two memory elements in the encoder, there exist 4 states in the state diagram. A trellis diagram is an extension of a state diagram that explicitely shows the passage of time. We describe the trellis diagram in conjunction with the recursive systematic code in the following section.

As a nonrecursive systematic convolutional encoder results in poor distance properties [1], the RSC code is usually used in turbo code. Figure 2.2 illustrates an example (2,1,3) RSC code with rate 1/2. In the figure, x denotes the input sequence, and one of the outputs which is different from x is represented as p, denoting a parity bit. p can be recursively calculated as:

$$p_i = p_i + \sum_{l=0}^{K-1} p_{i-l} g_l^{(1)} \bmod 2,$$

where $g^{(1)}$ is the generator sequence for output y. The generator matrix G for the RSD encoder in Figure 2.2 is:

$$\boldsymbol{G} = \begin{bmatrix} 111 \\ 101 \end{bmatrix}$$

Note that the generator matrixes of the two encoders in Figure 2.1 and Figure 2.2 are identical.

All the techniques used to analyze the nonrecursive systematic code can be applied to RSC codes. With the same generator matrix G, the trellis structure is identical

for the RSC code and the nonrecursive systematic code except that inputs and outputs corresponding to specific branches in the trellis may differ. A trellis diagram for the RSC encoder of Figure 2.2 is given in Figure 2.3. In the trellis diagram, nodes represent the states of the encoder. Directed branches show the possible state transitions due to different inputs. The labeled trellises define the encoded symbol generated by the transitions.

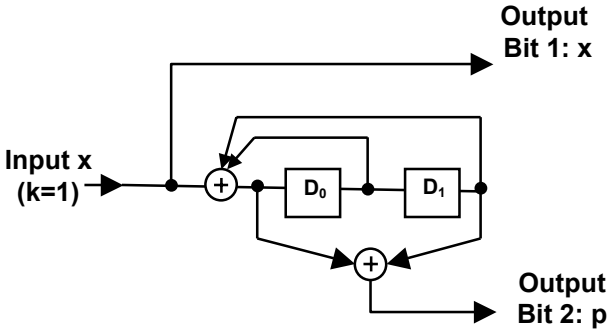


Figure 2.2: (2,1,3) recursive systematic convolutional encoder with rate ½ and generator matrix G = $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$

The usual scheme is to initialize the encoder to the all-zeros state S_0. Therefore, starting from the initial state S_0 at time t=0, the trellis records the possible transitions. After the entire inputs of a frame are encoded, a number of tail bits are appended (and encoded) to force the encoder back to state S_0. For a nonrecursive convolutional encoder, a sequence of '0' can force the final state to S_0. However, due to the presence of feedback in a RSC code, we need to solve state equations to find out the proper sequence of the padding bits for a RSC code [11].

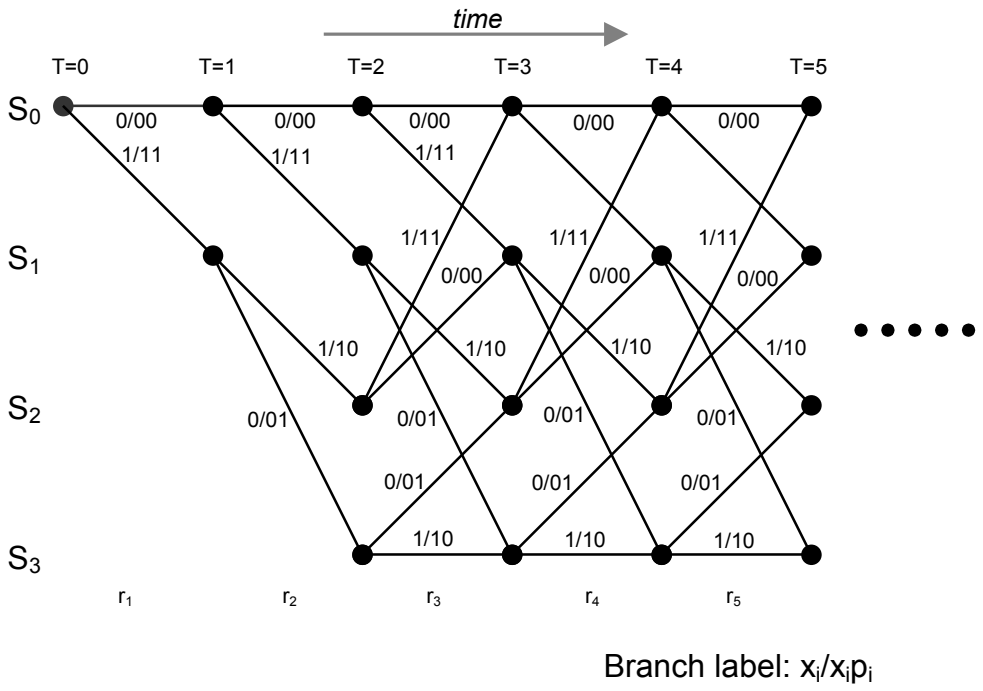


Figure 2.3: Trellis diagram for a (2,1,3) RSC code with rate ½

2.2 Turbo Encoder

Code sequences generated through parallel concatenation achieve error rate performance within a few tenths of a dB of the Shannon's limit (p. 77 of [48]). A typical turbo encoder composed of two identical parallel concatenated RSC encoders in conjunction with an interleaver is shown in Figure 2.4. Each RSC encoder is called a component encoder of the turbo encoder.

A binary input sequence x is applied to a component encoder, which is a RSC encoder with rate ½, to create a parity sequence p_1. Meanwhile, x is interleaved and subsequently encoded by a second component encoder to obtain a second parity sequence p_2. The information bit and the two parity streams are then multiplexed to create a coded output sequence Y, which results in a rate 1/3 turbo code. If a higher data rate such as ½ is required for the example decoder, it is possible to puncture the code streams. The puncture is beyond the scope of this thesis.

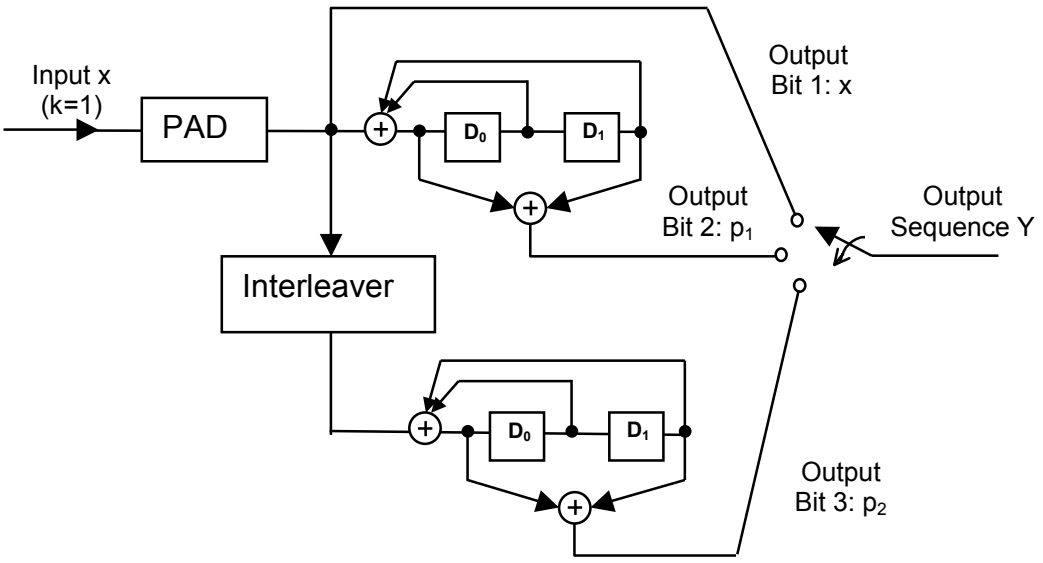


Fig. 2.4: Example rate 1/3 turbo encoder

The interleaver is a single input, single output logic block that receives a sequence of symbols from a fixed alphabet at the input and produces the identical symbols at the output in a different temporal order (p. 34 of [48]). When the encoded words are decoded at the receiver side, the reordering of the information bits can avoid some error patterns such as burst errors. It was reported that the construction and the length of the interleaver affect the performance of the code [4]. The performance improves as the interleave length grows (p. 33 of [4]). It is shown that some interleavers perform better than others [3], and there has been intensive research on interleavers [1], [4], [48], [49]. In terms of hardware design, a ROM can be used to store the chosen interleaver sequence. A more hardware efficient method is to use a pseudo noise (PN) generator using a primitive feedback polynomial. The data is written to a memory in the sequential order and read out in the pseudo-random order with the address generated by the PN generator. A PN generator is used in the interleaver for our turbo decoder. Details on the implementation of a PN generator will be discussed in Section 3.3.3.

The interleaver of a turbo encoder causes a problem to force state S_0 for the two RSC encoders at the end of each frame. By padding tail bits to the input sequence, the

state of the first encoder alone can be forced to state S_0. However, the interleaver imposes a constraint to simultaneously terminate the trellises of both RSC encoders to state S_0. Some special interleavers [11] or precursor bits are necessary to address the problem [4]. Hence, it is typical that the trellis starts and ends at state S_0 for encoder 1, while the trellis starts from state 0 for encoder 2 but ends up at an arbitrary state with equal probability.

2.3 Turbo Decoder Algorithm

2.3.1 Overview

Turbo decoding is an iterative application for the convolutional decoding algorithm to successively generate an improved retrieval of the transmitted data (p. 119 of [48]). In this section, we introduce notations and the turbo decoding algorithm briefly. A more detailed turbo decoding algorithm is explained later in Section 2.4.4.

First, we consider a binary digital communication system over an additive white Gaussian noise (AWGN) channel as shown in Figure 2.5. In this figure and for the later discussions, the following definitions and expressions hold:

1. x is the sequence of information bits with components over the finite field GF(2).
2. y is the sequence of encoded symbols resulting from the turbo encoder. Each symbol includes three bits: the systematic bit x, the parity bit p_1 from encoder 1, and the parity bit p_2 from encoder 2.
3. r is the received sequence of symbols with $r_k = [x_k', p_{1k}', p_{2k}']$, where r_k is the symbol received at time k in a frame. r_k is corrupted by Gaussian noise. Assume n_k, n_{p1}, n_{p2} are independent Gaussian random variables with zero mean and variance σ^2. Under a binary phase shift keying (BPSK) modulation scheme with antipodal signals ±1, the components x_k', p_{1k}', p_{2k}' are Gaussian random variables with mean $2x_k - 1$, $2p_{1k} - 1$ and $2p_{2k} - 1$, respectively, and variance σ^2. Then the three values x_k', p_{1k}', p_{2k}' can be represented as:

$x_k' = (2x_k - 1) + n_k$

$p_{1k}' = (2p_{1k} - 1) + n_{p1}$

$p_{2k}' = (2p_{2k} - 1) + n_{p2}$, where $x_k, p_{1k}, p_{2k} \in \{0, 1\}$.

4. As the three term $2x_k - 1$, $2p_{1k} - 1$ and $2p_{2k} - 1$ can only take on two values, 1 and –1, with equal probability, the Gaussian probability density function (pdf) for x_k' can be represented as:

$$f(x_k' \mid 2x_k - 1 = 1) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(x_k' - 1)^2}{2\sigma^2}) \tag{2.1}$$

$$f(x_k' \mid 2x_k - 1 = -1) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(x_k' + 1)^2}{2\sigma^2}) \tag{2.2}$$

and likewise for p_{1k}', p_{2k}'.

5. $r_{a \le k \le b}$ represent the sequence of the received symbols from time index k = a through b, that is, $r_{a \le k \le b} = (r_a, r_{a+1}, r_{a+2}, \ldots, r_{b-1}, r_b)$.

6. $\hat{x}$ is an estimate for the information sequence x. A Log-likelihood ratio (LLR) is a useful tool to make the estimation. Assume x_k is a bit that we want to determine the value of. The LLR of estimated x_k denoted as $\Lambda(\hat{x}_k)$ is defined as the logarithm of the ratio of the *a posteriori probability* (APP) of each information bit x_k being 1 to the APP of it being 0. We can use an arbitrary positive number ε as the base of the logarithm.

$$\Lambda(\hat{x}_k) \equiv \log_\varepsilon \frac{\Pr[x_k = 1 \mid r]}{\Pr[x_k = 0 \mid r]}$$

7. For a sequence x, the interleaved sequence is $\alpha(x)$, and the de-interleaved sequence of x is $\alpha^{-1}(x)$.

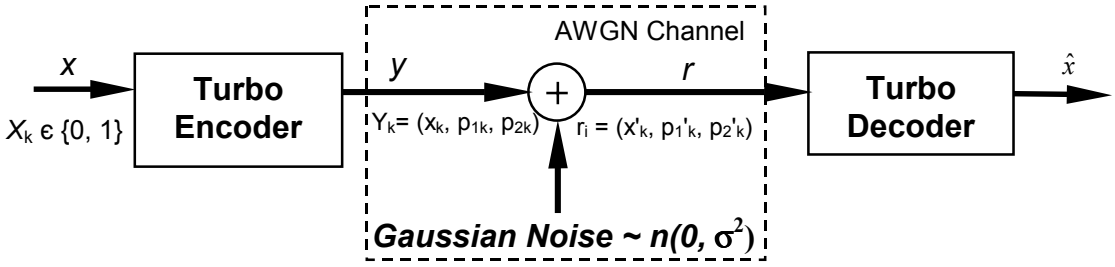


Fig. 2.5: The channel encoding and decoding system over an AWGN channel

A block diagram of a turbo decoder is shown in Figure 2.6. For each component encoder in the turbo encoder, there is one corresponding component decoder at the turbo decoder side. The decoding process is as follows. A component decoder DEC1 receives the systematic input x' the parity input p_1' and the de-interleaved *a priori* information $\alpha^{-1}(L_e^{(2)})$. The *a priori* information is produced by a component decoder DEC2 and is initialized to 0 during the first iteration. After the decoding process, DEC1 generates an LLR denoted as $\Lambda^{(1)}$ and the extrinsic information $L_e^{(1)}$. Then, DEC2 performs decoding process for the interleaved systematic input $\alpha(x')$, the parity p_2', and the interleaved *a priori* extrinsic information $\alpha(L_e^{(1)})$. As the result of the decoding process, DEC2 generates an LLR denoted as $\Lambda^{(2)}$, and the extrinsic information $L_e^{(2)}$. The extrinsic information $L_e^{(2)}$ is fed back to DEC1 after de-interleaving, and DEC1 repeats the same process. It should be noted that only when the extrinsic information generated by a decoder is uncorrelated with the *a priori* information of the decoder, then BER performance can be improved [4]. So it subtracts *a priori* information for calculation of the extrinsic information. The quality of the estimated sequence $\hat{x}$ improves for each iteration, owing to the improved *a priori* information from the previous coding stage. The iteration repeats until some predetermined stopping condition is met. The BER decreases from one iteration to the next according to a law of diminishing returns [4].

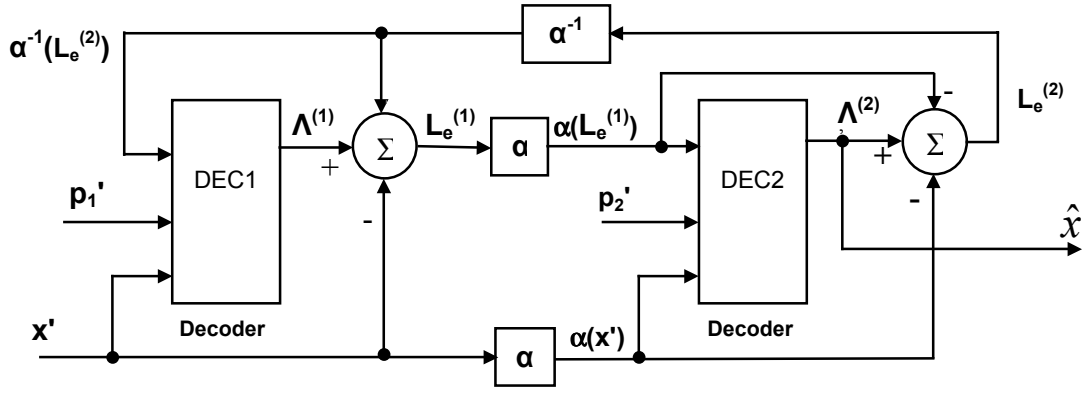


Fig. 2.6: Iterative turbo decoder

The interleaved signal $\alpha(x')$ is usually eliminated in actual implementation as shown in Figure 2.7. The elimination of the signal is possible as the the extrinsic information $L_e^{(1)}$ of DEC1 includes the information of x' provided we subtract the *a priori* information only, but not the systematic bit x' from the log-likelihood ration $\Lambda^{(1)}$. Our implementation (to be given in Chapter 3) is based on the block diagram shown in Figure 2.7.

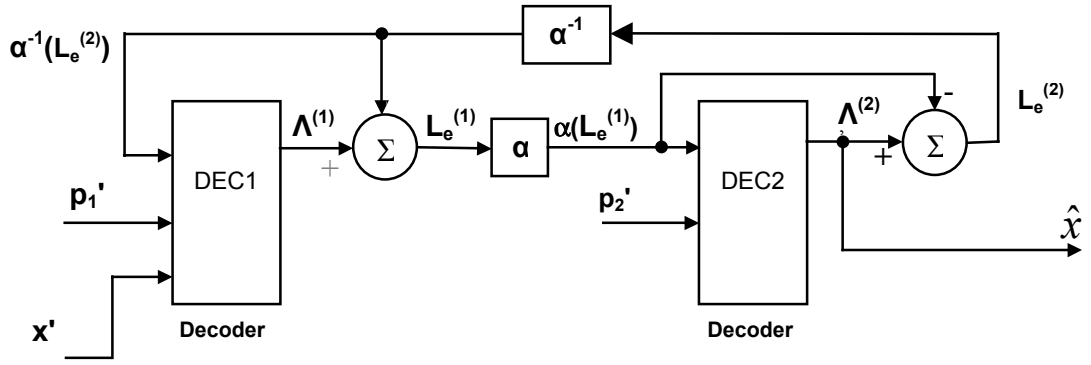


Fig. 2.7: Simplified Iterative turbo decoder

2.3.2 MAP algorithm and SOVA algorithm

The decoding algorithm for a component decoder of a turbo decoder should be able to handle LLRs at both inputs and outputs [4]. Two suitable candidate algorithms are soft-in soft-out Viterbi algorithm (SOVA) and *Bahl-Cocke-Jelinek-Raviv* (BCJR) algorithm. SOVA tries to minimize the code word error by maximizing the probably $p(r|\hat{x})$, which is a **maximum likelihood (ML)** algorithm; The BCJR algorithm attempts to maximize the *a posteriori* probabilities (APP) $p(\hat{x}|r)$ of the individual bits, and it is also known as **maximum a posteriori (MAP)** algorithm [2].

MAP algorithm is optimal for estimating an input information bit x_i in terms of BER. Extensive works have been reported on the MAP algorithm [1] - [4], [20] - [25], [29] - [38]. We present basic concepts and results of the MAP algorithm below.

As noted earlier, the LLR of the k^{th} input bit of an input sequence x is defined as:

$$\Lambda(\hat{x}_k) \equiv \log_{\varepsilon} \frac{\Pr[x_k = 1 \mid r]}{\Pr[x_k = 0 \mid r]} \tag{2.3}$$

$\Pr[x_k = 1|r]$ is the APP in which the information bit equals to '1'. Considering the state transition in the trellis structure, we can express $\Pr[x_k = 1|r]$ as follows:

$$\Pr[x_k = 1|r] = \sum_{(s',s)\in S^+} \Pr(S_{k-1} = s', S_k = s \mid r) = \sum_{(s',s)\in S^+} \Pr(S_{k-1} = s', S_k = s, r) / \Pr(r), \tag{2.4}$$

where S^+ is the set of all pairs of states which transient from a state s' at time k-1 to a state s at time k under $x_k = 1$. Similarly,

$$\Pr[x_k = 0|r] = \sum_{(s',s)\in S^-} \Pr(S_{k-1} = s', S_k = s \mid r) = \sum_{(s',s)\in S^-} \Pr(S_{k-1} = s', S_k = s, r) / \Pr(r), \tag{2.5}$$

where S^- is the set of all pairs of states which transient from a state s' at time k-1 to a state s at time k under $x_k = 0$. Hence, the LLR of the k^{th} input bit of an input sequence x is obtained as:

$$\Lambda(\hat{x}_k) \equiv \log_{\varepsilon} \frac{pr(x_k = 1 \mid r)}{pr(x_k = 0 \mid r)} = \log_{\varepsilon} \left[\frac{\sum_{(s',s)\in S^+} \Pr(S_{k-1} = s', S_k = s, r)}{\sum_{(s',s)\in S^-} \Pr(S_{k-1} = s', S_k = s, r)} \right] \tag{2.6}$$

If $\Lambda(\hat{x}_k) > 0$, we decode the input bit x_k as 1; otherwise, the input bit as 0.

$$x_k = \begin{cases} 1 & if \ \Lambda(\hat{x}_k) \geq 0 \\ 0 & if \ \Lambda(\hat{x}_k) < 0 \end{cases}$$

For convenience, we denote "$S_{k-1} = s'$" and "$S_k = s$" as S_{k-1} and S_k, respectively thereafter. We partition the joint probability of $\Pr(S_{k-1} = s', S_k = s, r)$ into three parts using Bayes' rule (p. 17 of [4]):

$$\begin{aligned} &\Pr(S_{k-1} = s', S_k = s, r) \equiv \Pr(S_{k-1}, S_k = s, r) \\ &= \Pr(S_{k-1}, r_{1\leq j<k}) \Pr(r_k, S_k \mid S_{k-1}) \Pr(r_{k<j\leq n} \mid S_k) \end{aligned} \tag{2.7}$$

Let us define the three probabilities as follows:

$$\alpha_{k-1}(S_{k-1}) \equiv \Pr(S_{k-1}, r_{1\leq j<k}) \tag{2.8}$$

`

$$\gamma_k(S_{k-1}, S_k) \equiv \Pr(r_k, S_k \mid S_{k-1}) \tag{2.9}$$

$$\beta_k (S_k) \equiv \Pr(r_{k<j\leq n} \mid S_k) \tag{2.10}$$

where $\alpha_{k-1}(S_{k-1})$ is the function of the received information prior to the stage k, $\gamma_k(S_{k-1}, S_k)$ is the function of the received information for stage k, and $\beta_k (S_k)$ is the function of the received information after stage k. This is illustrated in Figure 2.8.

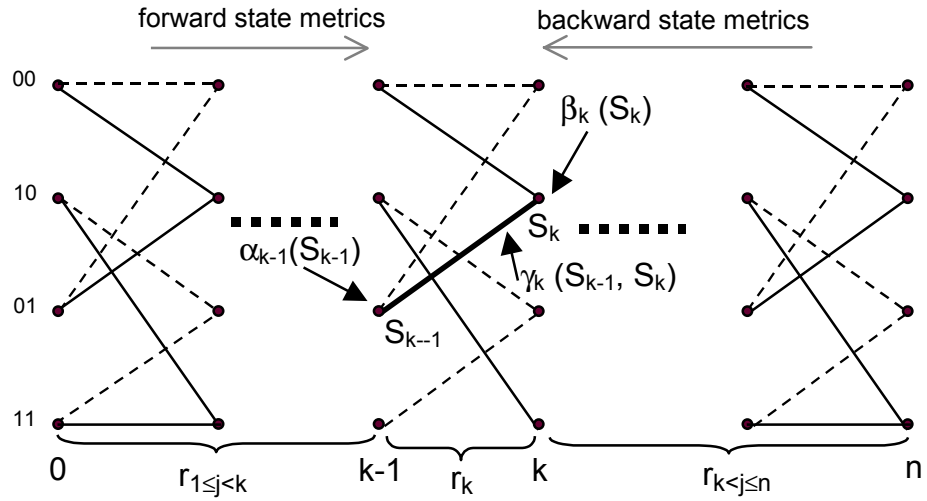


Fig. 2.8: Illustration of the probability functions that partition the received sequence

In order to maximize the APP of an estimated information bit $\hat{x}_k$, all the possible states through the trellis should be considered. Therefore, the probability functions α, γ and β must be computed for every state at every stage [4]. We use three parameters, *forward state metric* (FSM) α_k^m, *branch metric* (BM) $\gamma_k^{i,m}$ and *backward state metric* (BSM) β_k^m to characterize the node at stage k and state m in the trellis with incoming information bit i being either '1' or '0'. These notations are often simplified as $\alpha_k(S_k)$, $\gamma_k (S_{k-1}, S_k)$, $\beta_k (S_k)$.

The calculation of $\gamma_k^{i,m}$ involves estimation of the Gaussian probability density function (pdf) with noise variance σ^2. The function $\gamma_k^{i,m}$ can be expressed as:

$$\boldsymbol{\gamma_k^{i,m}} \equiv \Pr(r_k, S_k \mid S_{k-1}) = p(x_k=i)\, p(x_k' \mid x_k)\, p(p_k' \mid p_k) \tag{2.11}$$

In order to calculate $p(x_k' \mid x_k)$ and $p(p_k' \mid p_k)$, we define LLRs of the channel measurements of a received signal x_k' as $Lc(x_k')$ and p_k' as $Lc(p_k')$ as follows:

$$Lc(x_k') \equiv \log_{\varepsilon} \frac{p(x_k' \mid x_k = 1)}{p(x_k' \mid x_k = 0)} \tag{2.12}$$

$$Lc(p_k') \equiv \log_{\varepsilon} \frac{p(p_k' \mid p_k = 1)}{p(p_k' \mid p_k = 0)} \tag{2.13}$$

As mentioned before x_k' and p_k' are Gaussian random variables with mean $\pm$ 1 and variance σ^2. Referring to equation (2.1) and using the fact that

$$P(x_k' \mid x_k = 1) \cong f(x_k' \mid 2x_k\text{-}1 = 1) * \Delta$$

where the approximation becomes an equality in the limit as Δ --> 0.

$Lc(x_k')$ can be expressed as:

$$Lc(x_k') \equiv \log_{\varepsilon} \frac{p(x_k' \mid x_k = 1)}{p(x_k' \mid x_k = 0)} \cong \log_{\varepsilon} \frac{f(x_k' \mid 2x_k - 1 = 1)\Delta}{f(x_k' \mid 2x_k - 1 = -1)\Delta}$$

As Δ -> 0,

$$L_c(x_k') = \frac{1}{\ln \varepsilon} \frac{2}{\sigma^2} x_k' \tag{2.14}$$

Likewise,

$$Lc(p_k') \equiv \log_{\varepsilon} \frac{p(p_k' \mid p_k = 1)}{p(p_k' \mid p_k = 0)} = \frac{1}{\ln \varepsilon} \frac{2}{\sigma^2} p_k' \tag{2.15}$$

We can see that the LLRs of the channel measurements of received signals x_k', and p_k' can be obtained by scaling each input information by $\frac{1}{\ln \varepsilon} \frac{2}{\sigma^2}$ [4], [32]. Solving Equation (2.12) and (2.13) using the relations $p(x_k' \mid x_k = 0) = 1 - p(x_k' \mid x_k = 1)$, we obtain:

$$p(x_k' \mid x_k = 1) =, \text{ and}$$

$$p(x_k' \mid x_k = 0) = \left[\frac{1}{1 + \varepsilon^{Lc(x_k')}} \right]$$

We use x_k to represent a binary signal. The above two equations can be merged and expressed as:

$$p(x_k' \mid x_k) = \left[\frac{\varepsilon^{x_k Lc(x_k')}}{1 + \varepsilon^{Lc(x_k')}} \right] \tag{2.16}$$

$$p(p_k' \mid p_k) = \left[\frac{\varepsilon^{p_k Lc(p_k')}}{1 + \varepsilon^{Lc(p_k')}} \right] \tag{2.17}$$

Meanwhile, we use $L_a(x_k)$ to denote the *a priori* LLR of the information bit as:

$$L_a(x_k) \equiv \log_{\varepsilon} \frac{p(x_k = 1)}{p(x_k = 0)} \tag{2.18}$$

Solving the above equation using the relation $p(x_k = +1) = 1 - p(x_k = 0)$

$$p(x_k=i) = \frac{\varepsilon^{x_k L_a(x_k)}}{\varepsilon^{L_a(x_k)} + 1}, \; i=0 \text{ or } 1 \tag{2.19}$$

Combining equation (2.16) – (2.19), $\gamma_k^{i,m}$ becomes [4]:

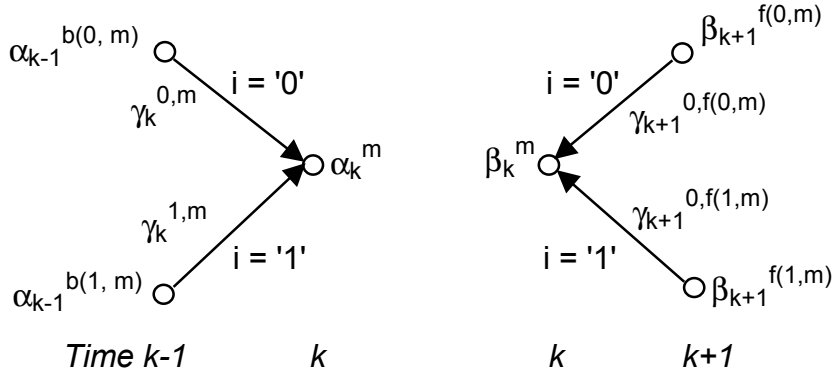
$$\boldsymbol{\gamma_k^{i,m}} = p(x_k=i)p(x_k' \mid x_k)p(p_k' \mid p_k) = \left[\frac{\varepsilon^{x_k L_a(x_k)}}{1+\varepsilon^{L_a(x_k)}}\right]\left[\frac{\varepsilon^{x_k Lc(x_k')}}{1+\varepsilon^{Lc(x_k')}}\right]\left[\frac{\varepsilon^{p_k Lc(p_k')}}{1+\varepsilon^{Lc(p_k')}}\right] \tag{2.20}$$

where (x_k, p_k) is the expected symbol of the specific transition from state b(i,m) at time k-1 to state m at time k, where b(i, m) is the previous state from current state m on the previous branch corresponding to input i.

The FSM α_k^m can be calculated by a forward recursion through the trellis with the knowledge of the initial states at time 0 and γ_k (S_{k-1}, S_k). Based on the probability calculation, α_k^m is expressed as:

$$\boldsymbol{\alpha_k^m = \alpha_{k-1}^{b(0,m)} \gamma_k^{0,m} + \alpha_{k-1}^{b(1,m)} \gamma_k^{1,m}} \tag{2.21}$$

As the encoder starts at the initial state 0, the FSM at time 0 can be initialized to $\alpha_0^0 = 1$ and $\alpha_0^m = 0$ for $m \neq 0$. A graphical representation of the calculation of α_i^m is shown in Figure 2.9 (a).



(a) Calculation of α_k^m (b) Calculation of β_k^m

Fig. 2.9: Graphical representation of the calculation of α_k^m and β_k^m

The BSM β_k^m is obtained in a similar way by a backward recursion through the trellis after all the branch metrics $\gamma_k^{i,\,m}$ are computed. A mathematical expression of the backward state metric is:

$$\boldsymbol{\beta_k^m = \beta_{k+1}^{f(0,m)}\,\gamma_{k+1}^{0,\,f(0,m)} + \beta_{k+1}^{f(1,m)}\,\gamma_{k+1}^{1,\,f(1,m)}} \tag{2.22}$$

where f(i, m) is defined as the next state given an input i and current state m. Figure 2.9 (b) illustrates the computation of β_i^m. As Decoder 1 terminates at state 0, $\beta_n^o = 1$ and $\beta_n^m = 0$ for $m \neq 0$. However, if no special termination scheme is used, the final state of Decoder 2 is unknown. Hence $\beta_n^m = 1$ for every state m. In fact, β_n^m can be assigned any number, but 0 is desirable to prevent overflow. Hence we have the following initialization condition:

Decoder 1: $\beta_n^o = 1$ and $\beta_n^m = 0$ for $m \neq 0$;

Decoder 2: $\beta_n^m = 0$ for every m;

We obtain the following Equation (2.23) from Equation (2.6) and (2.8)-(2.10):

$$\Lambda(\hat{x}_k) \equiv \log_\varepsilon \frac{p(x_k = 1 \mid r)}{p(x_k = 0 \mid r)} = \log_\varepsilon \left[\frac{\sum_{m,k} \alpha_{k-1}^{b(1,m)} \gamma_k^{1,m} \beta_k^m}{\sum_{m,k} \alpha_{k-1}^{b(0,m)} \gamma_k^{0,m} \beta_k^m} \right] \tag{2.23A}$$

Equation (2.23A) can be computed using the Equation (2.20) – (2.22) derived above. It is shown in the original turbo decoder paper [1] and later in [20] that the LLR $\Lambda(\hat{x}_k)$ can be partitioned into three terms as:

$$\Lambda(\hat{x}_k) = Lc(x_k') + L_a(\mathrm{x_k}) + L_{ek} \tag{2.23B}$$

where

$$\mathrm{Lc}(\mathrm{x_k}') = \frac{1}{\ln \varepsilon} \frac{2}{\sigma^2} \mathrm{x_k}' \tag{2.14}$$

$$\mathrm{L_a}(\mathrm{x_k}') = \log_\varepsilon \frac{p(x_k = 1)}{p(x_k = 0)} \tag{2.18}$$

In terms of hardware implementation, the two equations (2.23A) and (2.23B) are interpreted as follows. A MAP decoder outputs LLR $\Lambda(\hat{x}_k)$. The output of the decoder is partitioned into three parts: the LLR of channel estimation of received signal $\mathrm{x_k}'$, the *a priori* LLR of $\mathrm{x_k}$ and the *extrinsic* information $\mathrm{L_{ek}}$ [20]. In contrast, the first two terms

$Lc(x_k')$ and $L_a(x_k')$ combined are called *intrinsic* information. The MAP decoder can be configured to receive the sequence information in the form of log-likelihood ratios and to produce an estimate for the information in the form of LLRs too. The sign bit of $\Lambda(\hat{x})$ denotes the hard decision, and the magnitude of $\Lambda(\hat{x})$ represents the reliability of that hard decision.

The other candidate of the decoding algorithm of turbo decoding is soft output Viterbi algorithm (SOVA). While details of the algorithm are availible in [6], [26]-[29], we briefly describe the main idea of the SOVA. The SOVA algorithm considers only two paths, the ML path and one competing path that merge at the same state as shown in Figure 2.10. The state metric calculation involves the summation of the *a priori* information coming from the other decoder and the received channel information. The two inputs are called *soft-in* information. After all the state metrics are computed, traceback is performed, and the ML path is found using a conventional hard decision Viterbi decoder traceback scheme. The SOVA also finds a competitive path along with the ML path and generates the *soft output* by taking the minimum of the metric differences between the ML and competitive path along the ML-path. The SOVA takes the form of log-likelihood ratio, and it also preserves the additive structure similar to Equation (2.23B), that is:

$$\Lambda(\hat{x}_k) = Lc(x_k') + L_a(x_k) + L_{ek} \tag{2.24}$$

The last term is the extrinsic information L_e, which is the soft-output of a SOVA decoder. Therefore, it can be used as a component decoder of a turbo decoder.

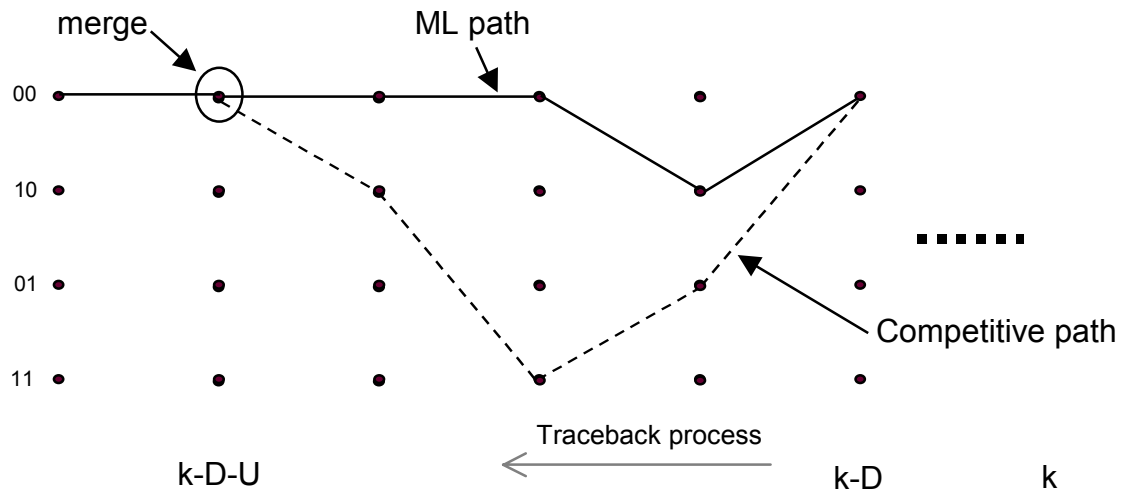


Fig. 2.10: Example of ML path and competitive path

The following diagram shows a generalized component decoder interface. It follows the Equation (2.24) and issuitable for both MAP and SOVA.

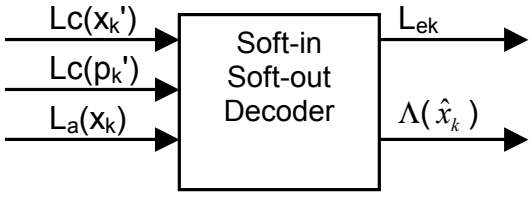


Figure 2.11: Soft-in / Soft-out decoder

The MAP algorithm involves extensive multiplications and logarithm computations, which are complicated in hardware implementation. The advantage of the MAP algorithm is that it takes all paths into consideration and generates the sum of the probabilities of all paths in the estimation of x_k. As the extrinsic LLR and the intrinsic LLR are uncorrelated (p. 103 of [4]) at least in theory, the performance of the MAP is optimal in terms of bit error rate (BER) [2].

In contrast, the SOVA produces the soft output by considering only two paths: the maximum likelihood path (ML) and the strongest competitor. Hence, the extrinsic

information depends strongly on the choice of the two paths. It may happen that the best competing path is eliminated before merging with the ML path. Therefore, the SOVA yields an inferior soft-output than the MAP algorithm at the advantage of relatively low computational complexity.

2.3.3 Log-MAP Algorithm

In order to avoid non-linear calculations in the MAP algorithm, it is reconfigured so that all data are expressed in negative logarithms. Then all the multiplications are converted to addition. Before we review the log-MAP algorithm, we introduce a binary operator called "E-operator". The E-operator is based on the Jacobian logarithm and is defined as follows [2]:

$$E(a, b) \equiv - \log_{\in}(\varepsilon^{-a} + \varepsilon^{-b}) = \min (a, b) - f(|a-b|); \tag{2.25}$$

$$f(z) = c \ln (1 + e^{-z/c}), z \geq 0 \tag{2.26}$$

$$c = \log_{\varepsilon} e \tag{2.27}$$

where the function f(z) is a nonlinear correction function and $\in$ can be an arbitrary positive number in theory. However, the limited bit width in hardware needs a careful choice of $\log_{\in} e$. Let us consider the adjust function f(z). If a look-up table (LUT) with 64 entries and 4-bit width are chosen to implement it, the input z should be quantized to 6 bits, ranging from 0 to 63. As f(z) is a monotonically decreasing function, the maximum value of f(z) is obtained for z = 0. Hence, $f(z)_{max} = c \ln (1 + e^{-z/c}) \mid_{z=0} = c \ln 2$. Since the output of f(z) is represented by only 4 binary bits, $f(z)_{max} = 15$. The equation $c \ln 2 = 15$ yields c = 21.64. Therefore, $\in$ is 1.047 from equation (2.27).

The E-operation can be extended to handle multiple terms. It can be expressed as:

$$E(a_1, a_2, \ldots, a_n) \equiv -\log_{\varepsilon}(\varepsilon^{-a_1} + \varepsilon^{-a_2} + \ldots + \varepsilon^{-a_n}) \tag{2.28}$$

The E-operation with multiple terms can be performed recursively by calculating two terms at a time as shown below:

$$\varepsilon^{-a_1} + \varepsilon^{-a_2} = \varepsilon^{\log_{\varepsilon}(\varepsilon^{-a_1} + \varepsilon^{-a_2})}$$

$$= \varepsilon^{-E(a_1, a_2)}$$

Similarly,

$$\begin{aligned}\varepsilon^{-a_1} + \varepsilon^{-a_2} + \varepsilon^{-a_3} &= (\varepsilon^{-a_1} + \varepsilon^{-a_2}) + \varepsilon^{-a_3} \\ &= \varepsilon^{-E(a_1, a_2)} + \varepsilon^{-a_3} \\ &= \varepsilon^{-E(-E(a_1, a_2), a_3)}\end{aligned}$$

By repeating the same procedure, we obtain

$$E(a_1, a_2, \ldots, a_n) = -\log_\varepsilon (\varepsilon^{-E(-E(-E(a_1, a_2), a_3))\ldots a_n)}) \tag{2.29}$$

Hence, the E-operation with multiple terms can be calculated using Equation (2.25) recursively.

By taking the negative logarithms on the three parameters, we have:

$$\Gamma_k^{i,m} \equiv -\log_\varepsilon \gamma_k^{i,m} \tag{2.30}$$

$$A_k^m \equiv -\log_\varepsilon \alpha_k^m \tag{2.31}$$

$$B_k^m \equiv -\log_\varepsilon \beta_k^m \tag{2.32}$$

$$L_k \equiv -\Lambda(\hat{x}_k) \tag{2.33}$$

The MAP algorithm becomes the log-MAP algorithm:

$$\begin{aligned}A_k^m &\equiv -\log_\varepsilon \alpha_k^m \\ &= -\log_\varepsilon (\alpha_{k-1}^{b(0,m)} \gamma_k^{0,m} + \alpha_{k-1}^{b(1,m)} \gamma_k^{1,m})\end{aligned}$$

After some calculation,

$$A_k^m = E(\Gamma_k^{0,m} + A_{k-1}^{b(0,m)}, \Gamma_k^{1,m} + A_{k-1}^{b(1,m)}) \tag{2.34}$$

$$= \min(\Gamma_k^{0,m} + A_{k-1}^{b(0,m)}, \Gamma_k^{1,m} + A_{k-1}^{b(1,m)}) - f(z) \tag{2.35}$$

where $z = |(\Gamma_k^{0,m} + A_{k-1}^{b(0,m)}) - (\Gamma_k^{1,m} + A_{k-1}^{b(1,m)})|$

Similarly,

$$B_k^m = E(\Gamma_{k+1}^{0,f(0,m)} + B_{k+1}^{f(0,m)}, \Gamma_{k+1}^{1,f(1,m)} + B_{k+1}^{f(1,m)}) \tag{2.36}$$

$$= \min(\Gamma_{k+1}^{0,f(0,m)} + B_{k+1}^{f(0,m)}, \Gamma_{k+1}^{1,f(1,m)} + B_{k+1}^{f(1,m)}) - f(z) \tag{2.37}$$

where $z = |(\Gamma_{k+1}^{0,f(0,m)} + B_{k+1}^{f(0,m)}) - (\Gamma_{k+1}^{1,f(1,m)} + B_{k+1}^{f(1,m)})|$

The branch metric can be expressed as:

$$\begin{aligned}\Gamma_k^{i,m} &= -\log_\varepsilon \varepsilon^{x_k Lc(x_k') + p_k Lc(p_k')} - \log_\varepsilon \varepsilon^{x_k L_a(x_k)} - C_k \\ &= -(x_k Lc(x_k') + p_k Lc(p_k') + x_k L_a(x_k) - C_k\end{aligned} \tag{2.38}$$

C_k is common for all the branches at stage k in the trellis, therefore it can be ignored for the calculation of $\Gamma_k^{i,m}$.

Finally, the LLR of the estimated information bit is obtained as:

$$L_k = \underset{m=0}{\overset{2^v-1}{E}}(A_{k-1}^{b(0,m)} + \Gamma_k^{0,m} + B_k^m) - \underset{m=0}{\overset{2^v-1}{E}}(A_{k-1}^{b(1,m)} + \Gamma_k^{1,m} + B_k^m) \quad (2.39)$$

Note that the E-operation for multiple terms can be applied to Equation (2.39). While all multiplications and exponential calculations are eliminated, only additions and subtractions are necessary for the calculation of LLRs. Hence, the log-MAP algorithm minimizes the computational complexity.

2.3.4 Turbo Decoding Algorithm Based on the Log-MAP Algorithm

In this section, the turbo decoding algorithm based on the log-MAP is explained. Refer to the structure of a turbo decoder shown in Figure 2.6. We use α to represent the interleaving and α^{-1} for de-interleaver. The first log-MAP decoder accepts three inputs: the received systematic bit sequence x', the parity bit sequence p_1' and a soft input $\alpha^{-1}(L_e^{(2)})$ from decoder DEC2. The two sequences x' and p_1' are then converted to log-likelihood ratios of the channel measurements of the received signals $Lc(x_k')$ and $Lc(p_k')$ by scaling the received signals with $\frac{1}{\ln \varepsilon}\frac{2}{\sigma^2}$. The log-MAP algorithm is performed by decoder DEC1. The log-MAP decoder generates a LLR $\Lambda(x_k')$ of the estimated sequence. $\Lambda(x_k')$ is given in Equation (2.39), which is denoted as $\Lambda^{(1)}$ in the block diagram. This process is represented as:

$$[\Lambda(x_k'), \Lambda(p_{1k}'), \alpha^{-1}(L_e^{(2)}{}_k)] \Rightarrow \Lambda^{(1)}{}_k$$

Base on Equation (2.23B), the extrinsic information $L_e^{(1)}$ from DEC1 is:

$$L_e^{(1)}{}_k = \Lambda^{(1)}{}_k - \Lambda(x_k') - \alpha^{-1}(L_e^{(2)}{}_k) \quad (2.40)$$

Equation (2.40) is an extrinsic LLR information for an estimate of the information sequence x. Note that it is a function of only the parity sequence p_1' for the given input sequence x. Therefore, the error in the estimation is independent of the error in the parity sequence p_2' [4]. So the extrinsic information generated by DEC1 and the intrinsic information from p_2' is uncorrelated.

Next, the extrinsic information $L_e^{(1)}$ of decoder DEC1 and the received information bit sequence x' are permutated to randomize the burst error in the sequence. The two interleaved sequences $\alpha(x')$ and $\alpha(L_e^{(1)})$ and the second parity bit sequence p_2' are applied to decoder DEC2. Similarly, the process of DEC2 is represented as:

$$[\Lambda(\alpha(x_k')), \Lambda(p_{2k}'), \alpha(L_e^{(1)}{}_k)] \Rightarrow \Lambda^{(2)}{}_k$$

The LLR of the estimated information bit of DEC2 is subtracted by the log-likelihood ratio of interleaved information bit x and the *a priori* LLR soft input to yield the extrinsic information of DEC2:

$$L_e^{(2)}{}_k = \Lambda^{(2)}{}_k - \Lambda(\alpha(x_k')) - \alpha(L_e^{(1)}{}_k) \tag{2.41}$$

After $L_e^{(2)}{}_k$ is de-interleaved, it becomes a new *a priori* information for DEC1. The reliability of the *a priori* LLR increases with an increased number of iterations, and the decoder generates the estimate with higher confidence. The loop continues for a fixed number of times, and the sign of the final LLR of the estimated information bit $\Lambda(\hat{x}_k)$ is examined to decide the value information bit, either '1' or '0'.

2.4 Implementation of the Log-MAP Algorithm

We give an overview of the implementation in this section and discuss the details of the proposed turbo decoder in Chaper 3. The major tasks of a log-MAP decoder are as follows:

1. Quantization: conversion of the analog inputs into digital.
2. Synchronization: Detection of the boundaries of frames and of code symbols
3. Branch metric computation, forward state metric computation, and data storage
4. Backward state metric computation and calculation of LLRs of the estimated information bit sequence
5. Generation of the output and iteration control

Figure 2.12 shows the flow of the log-MAP decoding algorithm, which performs the above tasks in the specific order.

Quantization is an important issue to achieve low bit error rate (BER) in turbo decoding [50]. In order to fit the received signal into the full scale range of the quantizer, adjustment of the received signal is necessary. Optimal scaling factors can be obtained as

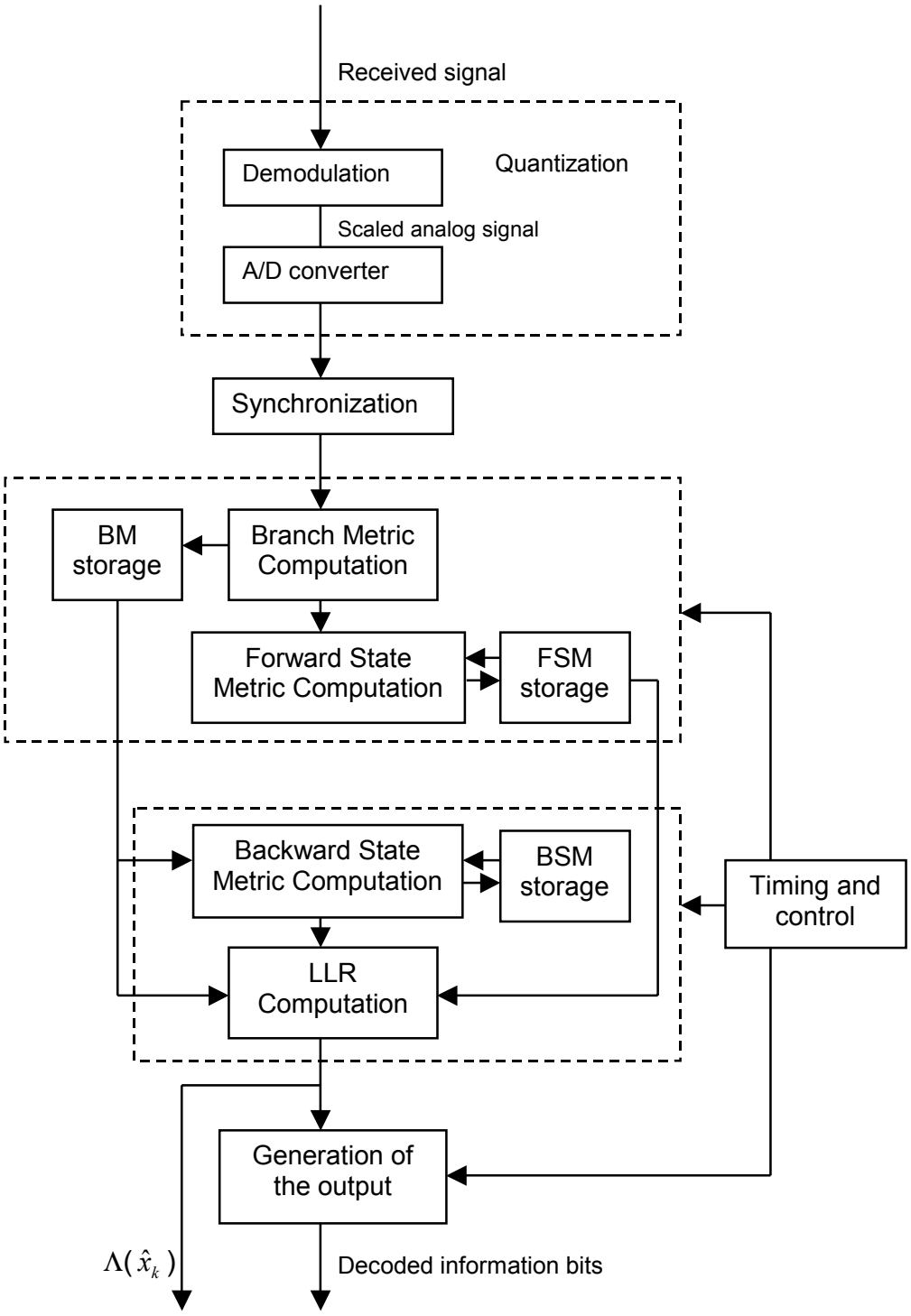


Fig. 2.12: The flow in soft-in soft-out log-MAP decoder

suggested in [50]. A lookup table stores the optimal scaling factor corresponding to a certain level E_b/N_0. Optimal scaling factors depend on frame size and the quatization level as well as E_b/N_0 of signals [50]. Optimal scaling factors lie between 0.21A and 0.26Q for E_b/N_0 = 0 dB ~ 2dB and the frame size = 1023, where Q is the number of quantization levels [50]. For example, if signals are quantized into 7 bits, Q becomes 127.

A demodulator in Figure 2.12 scales the input by a certain scaling factor, and the scaled signal is passed onto an analog-to-digital converter (ADC) to digitize the analog signal. The synchronization block detects the frame boundaries of code words and symbol boundaries in a code word. In our turbo decoder, *we assume that the inputs of a component decoder are already demodulated and quantized to 7 bit signed data. We also assume that the boundaries of the symbols and the frames have been identified.*

The branch metric given in equation (2.38) can be simplified based on the folowing equality [32]:

$$-br = |r|\,(b \oplus u(r)) - (r + |r|)/2 \tag{2.42}$$

where for a real number r

$$u(r) \equiv \begin{cases} 1 & \text{when } r \geq 0 \\ 0 & \text{when } r < 0, \end{cases}$$

and for binary numbers b_1 and b_2,

$$b_1 \oplus b_2 \equiv \begin{cases} 1 & \text{when } b_1 \neq b_2 \\ 0 & \text{when } b_1 = b_2 \end{cases}$$

Note that u(r) is the unit step function and $\oplus$ is XOR operation in logic design.

Let $A \equiv \frac{1}{\ln \varepsilon} \frac{2}{\sigma^2}$, Equation (2.14) and (2.15) can be rewritten as:

$$Lc(x_k) = Ax_k'$$

and

$$Lc(p_k) = Ap_k',$$

then Equation (2.38) becomes:

$$\Gamma_k^{i,m} = - x_k(Ax_k' + x_kL_a(x_k)) - p_kAp_k' - C_k$$

Using the equality of Equation (2.42) and noting that x_k and p_k are binary numbers, the branch metric can be expressed as:

$$\Gamma_k^{i,m} = | L_a(x_k) + A^*x_k' | (x_k \oplus u (L_a(x_k) + A^*x_k'))$$
$$- ((L_a(x_k) + A^*x_k') + | L_a(x_k) + A^*x_k' |) / 2$$
$$+| A^*p_k'| (p_k \oplus u (A^*p_k')) – (A^*p_k' + | A^*p_k' |) / 2 - C_k$$

Since only two variables x_k and p_k depend on the state m, it can be simplified as [10],[32]:

$\Gamma_k^{i,m} = |L_a(x_k) + A^*x_k' | (x_k \oplus u (L_a(x_k) + A^*x_k')) +| A^*p_k'| (p_k \oplus u (A^*p_k')) - C_k - C_k^*$

Two constant terms are common for all states of a stage k, and only the relative values of branch metrics at a stage are significant. Hence, the two constant terms, C_k and C_k^* can be dropped in implementation to yield the final equation given below:

$$\Gamma_k^{i,m} = |L_a(x_k) + A^*x_k' | (x_k \oplus u (L_a(x_k) + A^*x_k')) +| A^*p_k'| (p_k \oplus u (A^*p_k')) \quad (2.43)$$

The block diagram shown in Figure 2.13 implements Equation (2.43). The ABS modules calculate the absolute value of the channel information. Based on the outputs of the XOR gates, a multiplexer chooses the corresponding branch metric to the output.

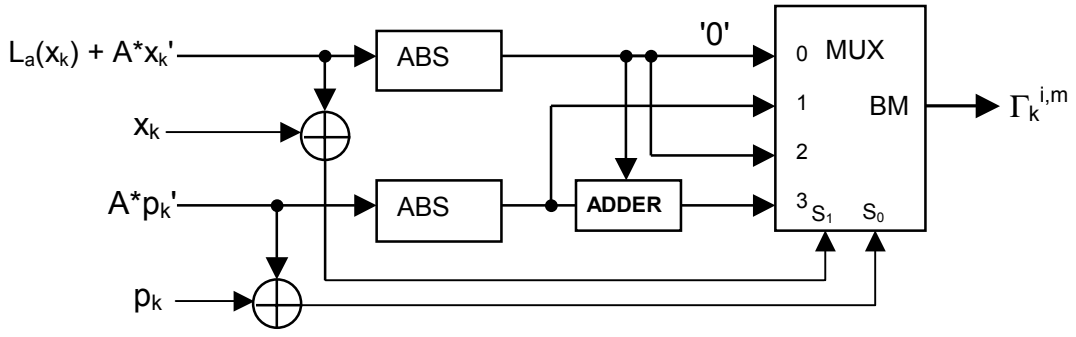


Figure 2.13: A branch metric computation block

Once the branch metric is computed for a branch, it is stored in a RAM for future use and passed onto the FSM calculation block at the same time. Since FSM calculation and BSM calculation are similar, we explain calculation of a FSM using a trellis diagram in Figure 2.14. The trellis diagram shown in Figure 2.14 consists of a butterfly structure, which contains a pair of origin and destination states, and four interconnecting branches. Refer to Section 2.4.3 for the notations used in the figure.

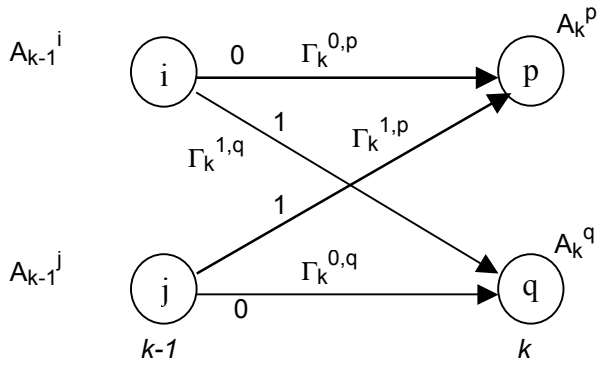


Fig. 2.14: Computation of FSMs

In the figure, the branches labeled '1' ('0') corresponds to input information bit '1' ('0'). The following relations hold for an (n, 1, K) RSC encoder:

$$j = i + 2^{K-2}$$

$$p = 2i$$

$$q = 2i+1$$

The key operation in the SM block is the E-operation. We name it Add-Compare-Select (ACS) operation as used in a Viterbi decoder. The block adds the state metrics of the two previous states with the corresponding branch metrics and obtains two path metrics of the current state. Up to this point, it is identical to the Viterbi algorithm. These two path metrics are compared and the minimum of them is selected. However, the calculation of the FSM is not finished yet. The correction function, which is the absolute difference between the two path metrics, should be calculated and used to adjust the state metric. The block diagram of an E-operation is shown in Figure 2.15. The f(z) module in the figure can be implemented using a look-up table (LUT).

Each butterfly wing in Figure 2.14 needs one E-operation. In order to reduce the decoder gate count, a serial implementation is usually used, that is, one FSM is computed during one clock period. In our design, there are $2^4 = 16$ states at one stage in the trellis, hence, we adopted 16 clocks to calculate the 16 FSMs for each stage.

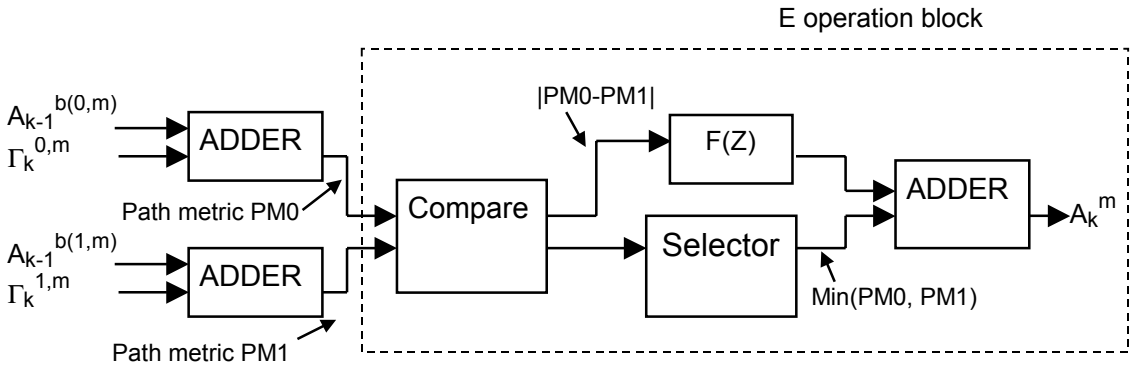


Fig 2.15: E-operation module in the FSM calculation block

After the FSMs are calculated for each node in the trellis, BSMs are computed in backward. Consider the butterfly given in Figure 2.16. The goal of the BSM calculation is to compute B_k^i and B_k^j for the given state metrics B_{k+1}^p and B_{k+1}^q, and branch metrics $\Gamma_{k+1}^{0,p}$, $\Gamma_{k+1}^{1,p}$, $\Gamma_{k+1}^{0,q}$ and $\Gamma_{k+1}^{1,q}$. The branch metrics are obtained from the RAM which stores the value during BM calculations. The ACS procedure and E-operations are the same as the FSM calculation.

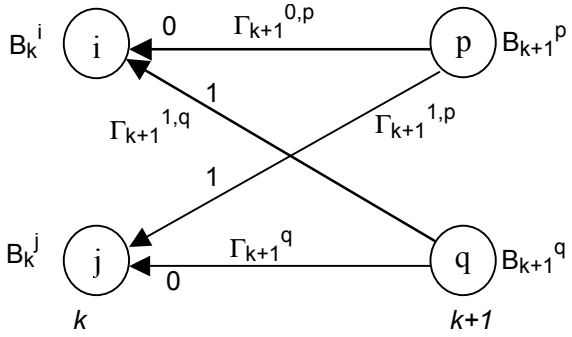


Fig. 2.16: Computation of BSMs

While BSM calculations are performed, the LLRs are computed at the LLR module simultaneously. Consider state i at stage k in Figure 2.16 as example: During the calculations of B_k^i, the intermediate backward path metrics $B_{k+1}^p + \Gamma_{k+1}^{0,p}$ and $B_{k+1}^q + \Gamma_{k+1}^{1,q}$ are applied to the LLR module. Meanwhile, the FSM A_k^i for state i at stage k is read from a RAM. The LLR module adds A_k^i to the two backward path metrics respectively and performs E-operations individually. Since there are 16 states at one

stage, the E-operation in the LLR module contains 16 terms. As stated in Section 2.4.3, the E-operation with multiple terms can be calculated recursively using Equation (2.29). The circuit block diagram is shown in Figure 2.17. During each clock cycle, the newly calculated metric and the current register output are fed back to the E-circuit and the result is stored in the same register for use in the next clock period. Each LLR module takes one path metric at a time, hence two parallel LLR modules are needed to calculate $\underset{m=0}{\overset{2^{v}-1}{E}}(\mathrm{A}_{\mathrm{k-1}}^{\mathrm{b(0,m)}}+\Gamma_{\mathrm{k}}^{\mathrm{0,m}}+B_k^m)$ and $\underset{m=0}{\overset{2^{v}-1}{E}}(\mathrm{A}_{\mathrm{k-1}}^{\mathrm{b(1,m)}}+\Gamma_{\mathrm{k}}^{\mathrm{1,m}}+B_k^m)$ respectively. There are $2^{\mathrm{v}} = 16$ states at one stage, hence, the LLR output of a log-MAP decoder is updated on every 16 clock cycles.

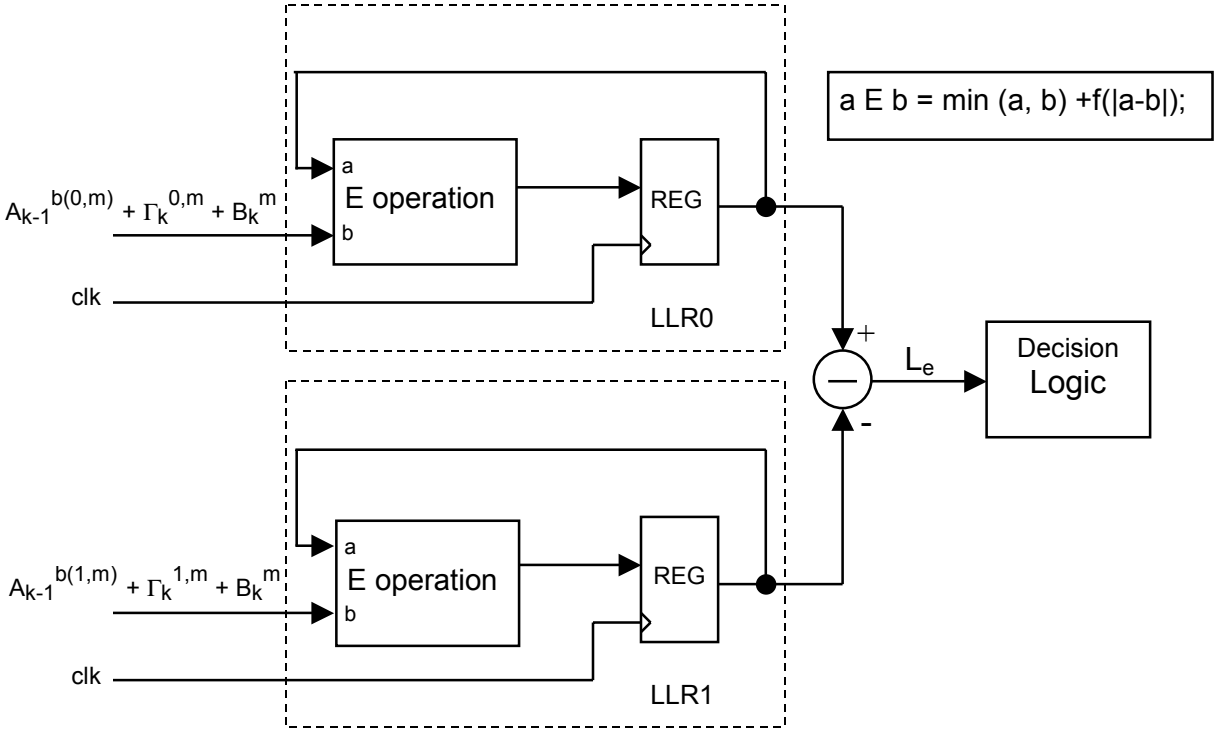


Fig. 2.17: Block diagram for the LLR

2.5 Implementation of a turbo decoder

The hardware implementation of a turbo decoder can follow the basic struture shown in Figure 2.7. Besides the two identical log-MAP decoders, the structure of the interleaver and the de-interleave should be decided. In this section, we discuss structure

of a commonly used interleaver and de-interleaver structure briefly. Different architectures of a turbo decoder are discussed next.

An interleaver plays a key role in the performance of the turbo coding algorithm. As stated earlier, an interleaver/de-interleaver based on a PN sequence generator is an efficient hardware. An interleaver/de-interleaver based on a PN generater writes a sequence of data into a RAM in the order whose address is generated by a PN generator and reads the data in the ascending order. A PN generator is a linear feedback shift register (LFSR), as shown in Figure 2.18 and can be expressed as a connection polynomial.

$$p(x) = x^k + a_{k-1} * x^{k-1} + \ldots + a_2 * x^2 + a_1 * x + x^0$$

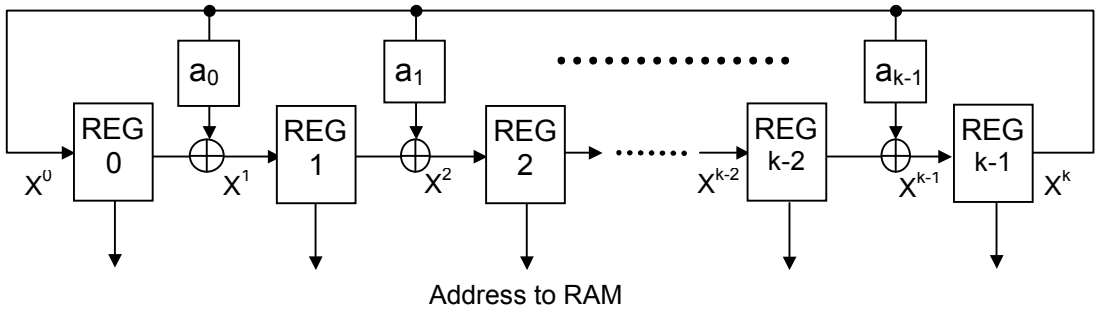


Fig. 2.18: PN generator structure

The coefficients a_k in the connection polynomial are in finite field GF(2). If p(x) is primitive with degree m, it generates all patterns except the all-0 pattern, called a *maximum length sequence*. It is shown that some interleavers perform better than others, a procedure for eliminating poor performance interleaver design is proposed in the paper [3]. A de-interleaver simply recovers the data sequence to its original order. A de-interleaver can be implemented by reversing the order of read and write.

A more flexible method is to store the order in a RAM and to interleave the data according to the order stored in the RAM. However, the size of RAMs is large for a large frame size. In order to save silicon area, interleavers and deinterleavers are implemented using PN generators in our design.

Iterative decoding can be realized with a single processor that performs all necessary computations or with a pipelined architecture that allows data to flow through a series of discrete computational engines. A typical method is to use two component decoders and have them take turns generating the APP of the information symbols. While one decoder is working, the other is in idle state. Since multiple iterations should be performed and finished in one frame time, the component decoders should work under a fast clock. The timing of the two decoders is show in Figure 2.19.

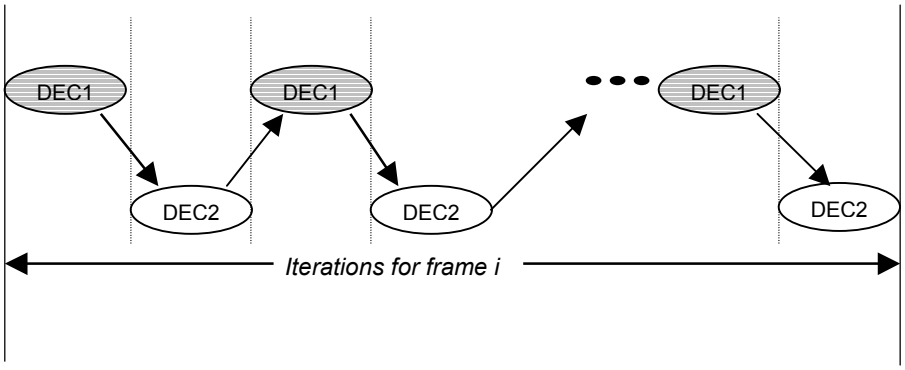


Fig. 2.19: Pipelined architecture of a turbo decoder in the serial mode

Another architecture of a turbo decoder is to employ the parallel mode. A parallel turbo decoding algorithm is illustrated in [48]. The parallel mode of decoding is shown in Figure 2.20. It is reported that the parallel mode of deceding eliminates the inherent bias toward the component decoder that is activated first in the iterative decoding process [48].

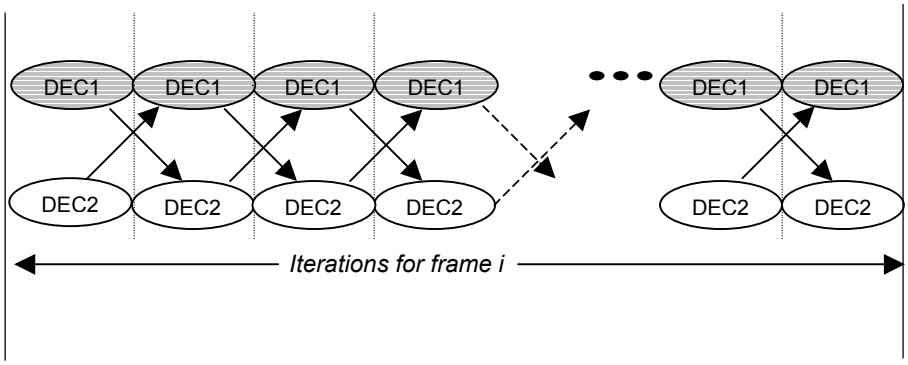


Fig. 2.20: Pipelined architecture of turbo decoder

In order to save power consumption and make the design simple, the serial utilization of the component decoders is chosen for our turbo decoder.

2.6 Low Power Design Techniques

The design of low power circuits can be tackled at different levels from the system level, the gate level to the switch level. At the system level, several schemes such as the selection of parallel and pipelined architecture or dynamic adjustment of the operations may be applied for low power design. The reduction of switching activity at nodes of a circuit, which directly affects the power dissipation, is the major focus at the gate level [52]. The optimization of the transistor sizing and parameter adjustment is another degree of freedom to minimize the power at the switch level. Voltage scaling and different transistor configurations in forming cells can also be applied at this level. For a standard cell approach, there is no control over internal circuitry (i.e. transistor sizes) of a cell. Therefore, in our low power design, the approaches to reduce the switching activities are mainly at the gate level and the system level.

The main source for a full static CMOS circuits to dissipate power is the dynamic power dissipation, caused by capacitance switching. Dynamic power dissipation is responsible for usually over 85 to 90 percent of the total poser dissipation [52]. The average switching power for a CMOS gate with a load capacitor, C_L, is given by:

$$P = \alpha C_L V^2 f$$

where V is the supply voltage, f is the clock frequency, and α is the signal activity. For each gate charging and discharging cycle, $C_L V^2$ Joul energy is dissipated by the load capacitance. The rate at which a gate switches in a circuit depends on the clock frequency f. The signal activity α is defined as the probability that the signal transits from 0 to 1 (or 1 to 0) for a clock cycle.

Given the formula of power dissipation, we can manipulate the four parameters to reduce the power dissipation. The supply voltage and the clock frequency are determined

at the system level, and they are beyond the control of a circuit designer. It is possible to reduce C_L by using less hardware and reducing the wire capacitance during place and routing. However, a major reduction in power dissipation can be achieved by reducing the switching activities where a designer has more control. A careful description of the circuit in a high-level description language (such as VHDL) can yield a circuit with a lower switching activity. Several techniques that may be employed for low power dissipation under the standard cell design approach are listed below:

- Elimination of redundant logic.

  A redundant logic, which does not contribute to the funtion of the circuit, dissipates power and should be eliminated. Usually a logic synthesis tool can optimise the circuit at the gate level and the redundant circuit is removed.

- Clock Gating.

  Clock gating is the most popular method for power reduction of clock signals. The clock signal is disabled for a functional unit which is not used for some extended period. In the pipeline design, when the registers stop functioning, the internal signal transitions in the combinational logic between registers are also eliminated and low power can be achieved. Figure 2.21 shows a clock gating method to disable a functional unit.

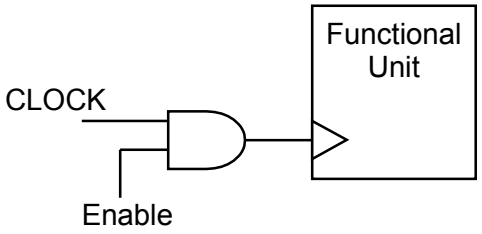


Fig. 2.21: Clock gating scheme

Clock gating requires additional logic to generate enable signals. The overhead dissipates power and may incur performance degradation. Therefore, clock gating should be employed only when benefits are greater than the cost.

- Variable number of iterations

  According to the level of channel noise, the number of iterations in the decoding is dynamically adjusted. When the noise level is low, the number of iterations can be reduced to save power.

- Blocking of floating inputs

  The outputs of RAMs are floating (high impedance) except during read or write operation. Floating inputs may incur short circuit current, hence, it should be avoided to reduce static power dissipation. We can block floating inputs using AND (or OR) gates with control signals at 0.

Employment of appropriate low-power design techniques depends on the characteristic of the circuit. In our turbo decoder, we consider all the above techniques for low-power design.

2.7 Review of Previous Work on Turbo Decoder Design

The turbo code was introduced by Berou et. Al [1] in 1993, and the turbo code combined with iterative decoding is the most powerful form of error control coding known so far. The algorithm has been investigated intensively [1] - [5], [20] - [38], and some hardware implementations of turbo decoder have already been reported [7], [8], [10], [16]. We will review some of the relevant works for turbo decoder designs.

Pietrobon proposed a turbo decoder with a serial block-type MAP decoder operating in the logarithm domain [10] [15]. Programmable gate arrays and EPROMs allow the decoder to be programmed for almost any code from 4 to 512 states, rate 1/3 to rate 1/7. Since the decoder requires large memory, he also presented an efficient implementation of a continuous decoder [10]. A sliding window technique that is similar, in principle, to of the traceback algorithm used in Viterbi decoders was used to reduce the block size at the expense of extra computations. The smaller block size results in less memory storage requirements and reduced delay [10], [13], [37], [38].

Hong and Stark presented a low-power multi-stage pipeline turbo decoder design [8], [12]. The number of iterations determined dynamically by comparing the Hamming

distance between the decoder bits from the two consecutive iteration stages. When the Hamming distance between the data from two consecutive iterations is smaller than a threshold value, the iteration is stopped, and the power down mode is incorporated.

Halter, Oberg, Chau and Siegel suggested an area and computational-time efficient turbo decoder implementation on a reconfigurable processor [7]. The turbo decoder takes advantage of the sliding window algorithms to achieve minimal storage requirements. The key system parameters can also be reconfigured via software. The flexibility and easy prototyping of FPGA technology makes this decoder suitable in a research environment.

All the above works aim to build up a turbo decoder using the log-MAP algorithm. Our design is also based on the same algorithm with emphasis on low-power design.

Chapter 3

Proposed Turbo Decoder Design

3.1 Overview

In this chapter, we describe the implementation of the iterative turbo decoder based on the log-MAP algorithm. First, the structure of component decoders implementing the log-MAP algorithm is introduced. Second, implementation of the overall turbo decoder is illustrated in detail. Finally, we propose a low-power design of a turbo decoder with variable iterations and the power shutdown mode.

3.2 Hardware Implementation of the Log-MAP Algorithm

The previous chapter described the log-MAP algorithm, which is the core of the turbo decoder. In this section, we discuss the hardware implementation of this algorithm. The top-level functional block diagram of the log-MAP decoder is presented first. Then, the design of the major components in a log-MAP decoder is illustrated in detail in the subsequent sections.

The system specifications for our turbo encoder and turbo decoder are given below:

- System
  Bit rate (input rate of the encoder): 125 kbps
  Frame length: 1024 symbols/frame (1023 data and one synchronization symbol)
  Period of a frame: 8.192 ms
  System clock: 40 MHZ
  Gate count: about 13,000 equivalent NAND2 gates

- Encoder: (3,1,5) RSC encoder
  Rate: 1/3
  Constraint length: 5

Generator matrix: $G = \begin{bmatrix} 10011 \\ 11101 \end{bmatrix}$

Interleaver length: 1023 bits

- Decoder

  Type: log-MAP decoder

  No. of component decoders: 2

  Interleaver length: 1023 bits

  Clocks: system clock (CLK), 40 MHz

  data clock (DCLK), 2.5 MHz

  CLK1, 20 MHz

  CLK2, 20 MHz

  The clocks used for the log-MAP component decoders are shown in Figure 3.1.

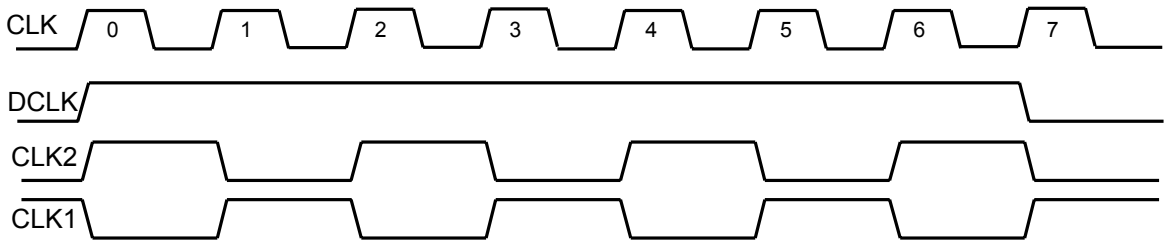


Fig. 3.1: Clocks used in the log-MAP decoder

3.2.1 Top Level Log-MAP Decoder

The major building blocks of our log-MAP turbo decoder are shown in Figure. 3.2. The four major components are the branch metric calculation (BMC) unit, forward state metric calculation (FSMC) unit, backward state metric calculation (BSMC) unit and the log-likelihood ratio calculation (LLRC) unit. A control circuit is also part of the decoder. Besides these blocks, 2 16kx8 bits RAMs are used to store FSMs and BSMs respectively, and 2 2kx8 bits RAM stores BMs.

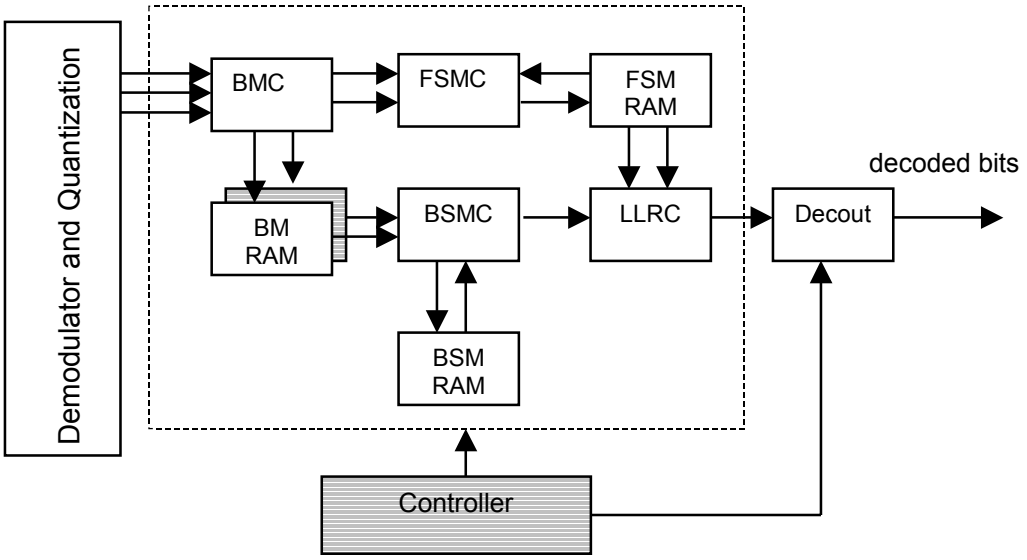


Fig. 3.2: The block diagram of a log-MAP decoder

All the notations used in Chapter 2 are held except the following definitions:

1. The received sequence passes through a demodulator and an ADC. The information bit sequences x_i', parity bits $p1_i'$ and $p2_i'$ are quantized into q = 7 bits in 2's complement form ranging from –64 to 63. The quantized data sequences are denoted as sys, par1 and par2, which correspond to x_i', $p1_i'$ and $p2_i'$ respectively. Our design does not implement a demodulator, an AGC and an ADC.
2. L_a is the *a priori* probability (APrP) from the previous stage decoder. It is quantized into 8 bits and ranges from –128 to 127.
3. A_k^m, B_k^m, and $\Gamma_k^{i,m}$ used in Chapter 2 are replaced by FSM_k^m, BSM_k^m and $BM_k^{i,m}$.

The BMC unit receives three inputs: L_a, sys and par1 (par2) for Decoder 1 (Decoder 2). It generates branch metrics (BMs), which are routed to the FSMC unit and stored in the BM RAMs at the same time. The FSM unit computes new FSMs using the existing FSMs stored in the FSM RAMs and the newly obtained BMs. After all the FSMs are calculated for the current frame, the BSMC unit starts to compute BSMs using the BMs and the existing BSMs read from the BSM RAMs. Meanwhile, the LLR unit generates a log-likelihood ratio based on the current FSM, and the backward path metrics

(obtained from BMs and BSMs). If this is the last iteration of the turbo decoding, the estimated information bits are generated by the DECOUT block.

In order to reduce hardware complexity, we employ a serial SM (state metric) calculation. Therefore, a decoder clock (CLK), which is the system clock, should be 2^v times faster than the data clock (DCLK), where v is the number of states of the corresponding encoder. As v = 4 for our encoder, the decoder clock CLK is 16 times faster than the data clock DCLK. Two other clocks, CLK1 and CLK2, are used in our decoder and shown in Figure 3.1. Note that CLK1 and CLK2 are half the system clock frequency and are complement with each other.

3.2.2 Branch Metric Calculation (BMC) Module

The branch metric is computed based on the received input values and the expected code symbols on a trellis. The 16 states at time k in the trellis transient to another 16 states at time k+1 according to the input information bit x_i'. The state transition from the stage k-1 to the next stage k can be described using 8 butterflies, where each butterfly corresponds to two input states and two output states as shown in Figure 3.3. The expected symbol for each possible state transition is listed on the branch of each butterfly with format "ip", where i is the input information bit, and p is the parity bit corresponding to the component decoder.

In Section 2.5, Equation (2.43) shows the calculation of branch metrics, and it is given again.

$$BM_k^{i,m} = | L\_a_k + sys_k | (i \oplus u (L\_a_k + sys_k)) + | par_k| (p^{i,m} \oplus u (par_k)) \qquad (3.1)$$

The branch metric $BM_k^{i,m}$ is the metric for the branch whose destination state is m at the stage k under the information bit i. For example, $BM_k^{0,12}$ corresponds to the branch between S_{14} at stage k-1 and S_{12} at stage k in Figure 3.3. The above equation is

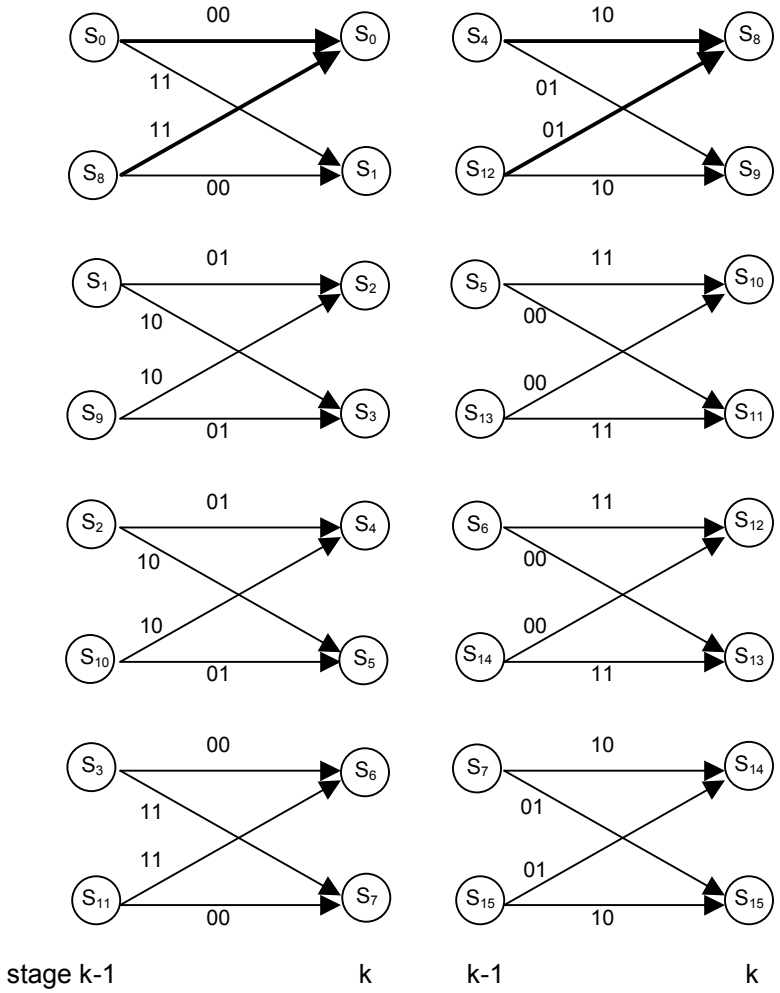


Fig. 3.3: State transition from time *k-1* to time *k*

represented in the truth table form in Table 3.1, which reflects the actual implementation of the BM block.

Table 3.1 Implementation of the SM block

$i \oplus u\ (L\_a_k + sys_k)$	$p^{i,m} \oplus u\ (par_k)$	$BM_k^{i,m}$
0	0	0
0	1	$\lvert par_k \rvert$
1	0	$\lvert L\_a_k + sys_k \rvert$
1	1	$\lvert L\_a_k + sys_k \rvert + \lvert par_k \rvert$

In the above table, u(x) is the unit step function which returns '1' ('0') if $x \geq 0$ ($x < 0$). As noted earlier, *a priori* probability L_a_k of stage k is a signed 8-bit number ranging from

–128 to 127. Both sys_k and par_k of a stage k are 7 bits and range from –64 to 63. Only the information bit i and the parity bit p are specific to the branch, and the other parameters are common to all the 32 branches of the stage k. In order to limit the number of bits to express a SM, an unsigned 7-bit is used to represent a BM, and hence, the maximum value of a BM is limited to 127. Note that BM is always positive, and a larger value of BM implies that the path is less likelihood in the trellis. Due to the limitation, the result of the summation is truncated if $L\_a_k + sys_k$ is greater than 127 or less than –128. If both $(i \oplus u (L\_a_k + sys_k))$ and $(p^{i,m} \oplus u (par_k))$ are '1', the sum of $| L\_a_k + sys_k |$ and $| par_k|$ are again limited within 127. A ripple carry adder (RCA) with the bit width of 7 is used to perform the addition. When the carry-out bit is '0', the sum of the two data is less than or equal to 127, and no limitation on the bits is necessary. However, if the carry-out bit is '1', which implies that the addition causes an overflow, all the 7 bits are set to '1' to limit the resultant output to 127. Seven 2-input OR logical gates are used to implement the limiting circuit. A block diagram shown in Figure 3.4 shows the above operation.

The LIM block limits the bit width within the desired range. The ABS block calculates the absolute value of the incoming data. A multiplexer determines to output the 2's complement of the input or the input itself based on the input's signed bit MSB. If the MSB is '0', the output of the ABS circuit is the input itself, otherwise, the 2's complement of the input is output. During one period of the data clock DCLK, the four inputs of the MUX are held because the three input data L_a, sys and par remain the same. One of the four MUX inputs of each MUX is selected as the output of the BMC module based on the expected symbols read from the ROM.

The expected symbols "ip" for each state transition are stored in ROMs. Only p is different among the expected symbols of the branch under i = '0', hence, we can use two 16x1 bit ROM to store p under i = '0' and i = '1', respectively. The parity bits p are stored in ROM1 at address 0 to 15 under i = '0'. p under i = '1' are stored at address 0 through address 15 in ROM2. The access address of the ROM is controlled by an up-counter counting from 0 to 15 working under decoder clock CLK.

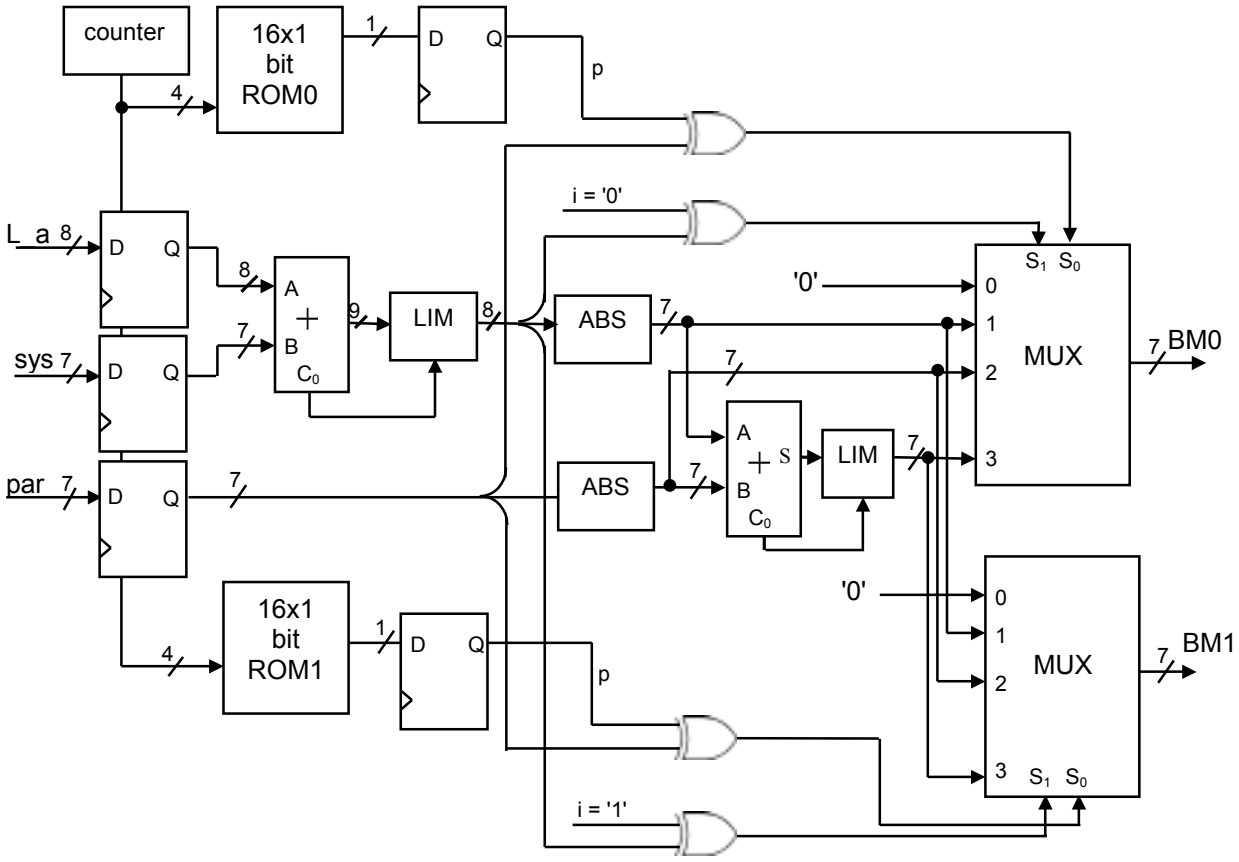


Fig. 3.4: Block diagram of BMC module

Since the branch metrics of a given stage depend only on the *a priori* information, (i.e., the systematic and the parity bits), there are only four different branch metrics for each stage k corresponding to ip = 00, 01, 10 and 11. However, since branch metrics are used for the forward state metric calculation at the same time, a pair of BMs, $BM_k^{0,m}$ and $BM_k^{1,m}$ needs to be calculated for every state during the forward state metric calculation. Close examination of equation (3.1) and the butterfly structure in Figure 3.3 reveals that every pair of branches ending at state 2m and state 2m+1 has the same BMs. Take the top butterfly structure in Figure 3.3 as example. The input states are S_0 and S_8, and the output states are S_0 and S_1. The state transitions from S_0 to S_0 and S_8 to S_1 correspond to input data ip = 01. Since the two state transitions have the same expected symbol, their BMs are the same. So do the two BMs for ip= 10. Therefore, we need to calculate only the BMs for the branches terminate at states 2m, where m = 0, 1, 2, … , 7. In order to

simplify the design, a pair of BMs, $BM_k^{0,2m}$ and $BM_k^{1,2m}$, whose branches terminate at the same state 2m at stage k are calculated in parallel. The 7 pairs of BMs are calculated in sequence from m = 0 through m = 7.

As branch metrics are also necessary for backward state metric calculations, the calculated BMs are stored into two RAMs. In order to save silicon area, only four branch metrics are stored. $BM_k^{0,0}$ and $BM_k^{0,8}$, corresponding to ip = 00, 01, are stored in the first RAM. $BM_k^{1,0}$, $BM_k^{1,8}$ are stored in the second RAM, corresponding to ip = 10 and 11, respectively. These four branch metrics are highlighted in Figure 3.3. Totally, 2N storage elements are required in total for each BM RAM, where N is the frame length. As N = 1023 for our decoder, it needs 4096 bytes of memory to store BMs for our decoder. In order to access $BM_k^{0,m}$ and $BM_k^{1,m}$ at the same time, we employed two 2K byte asynchronous single port static RAMs in our design.

A timing diagram of the BM calculations is shown in Figure 3.5. The input data L_a_k, sys_k and par_k are sampled at the rising edge of the data clock DCLK. Before a pair of BMs is calculated, the expected symbols should be read out from the ROM. The read addresses are 2m and 2m + 16, corresponding to the pair of branches that terminate at the same state 2m. The expected symbols are then available to use at the rising edge of CLK1. The pair of BMs is computed, and available at the next rising edge of CLK1. Meanwhile, the expected symbols for the next pair of BMs are ready to be read out. As a result, the first pair of BMs is available after 3 CLK cycle delay, where the first cycle is to read symbols from the ROM and the next two CLK cycles to compute BMs. , A new pair of BMs is generated and passed onto FSMC module at every rising edge of CLK1.

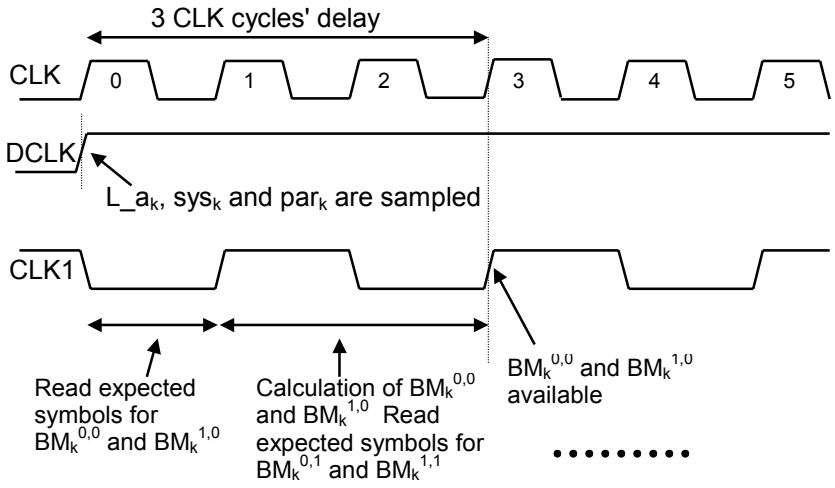


Fig. 3.5: Timing diagram of BM calculations

3.2.3 Forward and Backward State Metric (SM) Calculation module

The structure of a forward state metric calculation (FSMC) module and that of the backward state metric calculation (BSMC) module are very similar. We explain the operation of the FSMC module in detail and describe the differences for the BSMC module.

Each node needs two state metrics, the forward state metric (FSM) and the backward state metric (BSM). After considering the precision of the calculation and the hardware complexity, we assign an 8 bit unsigned number to each state metric. Hence, the range of the state metric is from 0 to 255.

- **FSM Calculation**

For each node S_m, there exist two incoming branches b_i (corresponding to information bit '0') and b_j (corresponding to information bit '1') departing from S_i and S_j, respectively. The FSM of S_m is the E-operation of the two path metrics (the FSM of S_i + the BM of b_i) and (the FSM of S_j + the BM of S_j). We illustrate the process using a butterfly shown in Figure 3.6. Suppose that we want to compute the FSM of state S_6 at

time k, FSM_k^6. From the relationship of a butterfly wing given in Section 2.5, we can identify two incoming branches and their departing states, S_3 and S_{11}. The FSM is computed as

$$FSM_k^6 = E\,(FSM_{k-1}^3 + BM_k^{0,6},\ FSM_{k-1}^{11} + BM_k^{1,6})$$

According to the definition of the E-operation described in Section 2.5, the above equation is expressed as:

$$FSM_k^6 = \min\,(FSM_{k-1}^3 + BM_k^{0,6},\ FSM_{k-1}^{11} + BM_k^{1,6}) - f(|\,(FSM_{k-1}^3 + BM_k^{0,6}) - (FSM_{k-1}^{11} + BM_k^{1,6})\,|)$$

and

$$f(z) = c\ \mathrm{In}\,(1 + e^{-z/c}),\ z \geq 0$$

where c is determined according to the size of the LUT. Considering the trade-off between the precision and hardware complexity, we have decided the size of the LUT be 64x4 bits. Hence, c is found to be 21.6. (Refer to Section 2.5 regarding the determination of the value of c.)

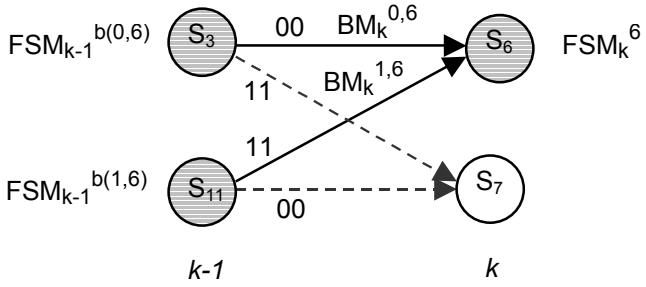


Fig. 3.6: Partial trellis for the computation of FSM

If we denote the path metric of a branch that arrives at a state m at time k under input information i as $PM_k^{i,m}$, we have:

$$PM_k^{i,m} \equiv FSM_{k-1}^{b(i,m)} + BM_k^{i,m} \qquad (3.2)$$

Note that $FSM_{k-1}^{b(i,m)}$ denotes the FSM of the state at time k-1 which transient to state m under input information i. Thus, in general, a FSM for a node m at time k is:

$$FSM_k^m = \min\,(PM_k^{0,m},\ PM_k^{1,m}) - f(|PM_k^{0,m} - PM_k^{1,m}|) \qquad (3.3)$$

where

$$f(z) = 21.6\ \ln(1 + e^{-z/21.6}),\ z \geq 0 \qquad (3.4)$$

As an addition of two positive numbers is simpler than a subtraction, we make f(z) absorb the "minus" symbol. Hence, the FSM calculation becomes:

$$FSM_k^m = \min (PM_k^{0,m}, PM_k^{1,m}) + f(|PM_k^{0,m} - PM_k^{1,m}|) \quad (3.5)$$

where

$$f(z) = - 21.6 \ln(1 + e^{-z/21.6}), z \geq 0 \quad (3.6)$$

f(z) in Equation (3.6) is always negative with the minimum value of –21.6*ln2 at z = 0. In order to circumvent the usage of a signed data to represent negative f(z) in the LUT, a bias of 21.6*ln2 is added to f(z). Thus, Equation (3.6) is modified as:

$$f(z) = - 21.6 \ln(1 + e^{-z/21.6}) + 21.6 \ln 2, z \geq 0 \quad (3.7)$$

The resultant f(z) is then equal to or greater than 0, where a 4-bit unsigned data is used to represent f(z). This is feasible as the relative, not absolute, values of FSMs are important. A f(z) LUT is shown in Figure 3.7.

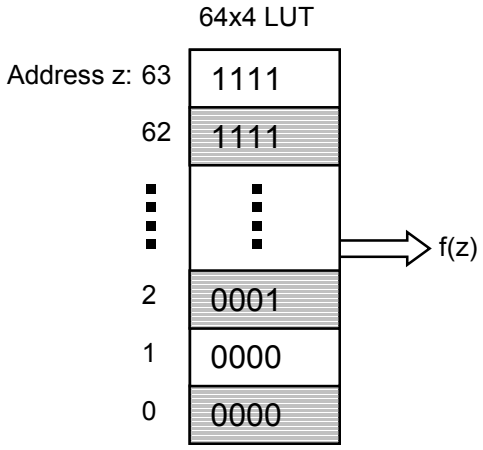


Fig. 3.7: LUT of adjustment function f(z)

The first step of the calculation is to determine the addresses to read out $FSM_{k-1}^{b(0,m)}$ and $FSM_{k-1}^{b(1,m)}$. Noting that $BM_k^{0,m}$ and $BM_k^{1,m}$ are calculated in the ascending order of the state and the newly calculated $BM_k^{0,m}$ and $BM_k^{1,m}$ are to be used in the calculation directly, FSMs should also be calculated in the same order as for BMs. Hence, FSM_k^0, FSM_k^1, … , FSM_k^{15} are computed one at a time in the order. Between the two possible previous states for each state, the state corresponding to the input '0' is read out first. Therefore, the addresses to read the previous stage's FSMs are not continuous,

but in the order of 0, 8, 1, 9, 2, 10, 3, 11, 12, 4, 13, 5, 14, 6, 15, 7. (Refer to Figure 3.3 for the sequence.) As a data is read out from a RAM serially, a serial-to-parallel operation is necessary to compute a new FSM. The serial-to-parallel circuit is included in the log-MAP controller and shown in Figure 3.8.

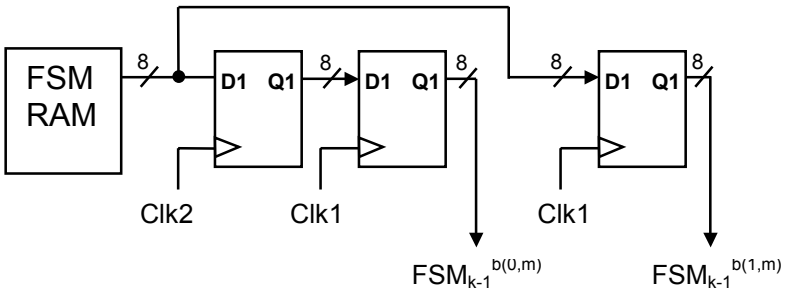


Fig. 3.8: Serial-to-parallel circuit to input FSMs read from the RAM

Consider the timing diagram of the FSM calculation shown in Figure 3.9. At a rising edge of CLK1, two FSMs, $FSM_{k-1}^{b(0,m)}$ and $FSM_{k-1}^{b(1,m)}$ in the previous stage are available at the input of FSMC module. $BM_k^{0,m}$ and $BM_k^{1,m}$ are passed to the FSMC module at the same time. As mentioned in Section 3.2.2, BMs of every pair of branches ending at state 2m and state 2m+1 are exactly the same. Hence, with the two FSMs in the previous stage and two corresponding BMs, two consecutive FSMs of the same butterfly of the current stage can be computed based on Equation (3.5). Noting that the period of CLK1 is twice the period of CLK and they have the same rising edge, the two consecutive FSMs can be calculated in two CLK cycles. During the first CLK cycle, path metrics $PM_k^{0,m}$ and $PM_k^{1,m}$ are calculated by adding $BM_k^{0,m}$ to $FSM_{k-1}^{b(0,m)}$ and $BM_k^{1,m}$ to $FSM_{k-1}^{b(1,m)}$, respectively. Next, the E-operation is performed on $PM_k^{0,m}$ and $PM_k^{1,m}$ to calculate FSM_k^m. In the following CLK cycle, $BM_k^{0,m}$ and $BM_k^{1,m}$ are reversed to compute FSM_k^{m+1}. A newly computed FSM is latched by a register at the rising edge of CLK and written to a FSM RAM using the address produced by an up-counter in the order of 0, 1, 2, 3, …, 15. A finite state machine with two states, S0 and S1, is implemented to decide whether BMs should be reversed or not. The state of the machine switches between S0 and S1 at each rising edge of CLK. BMs are not reversed at the state of S0 while reversed BMs are used to compute FSM_k^{m+1} at state S1.

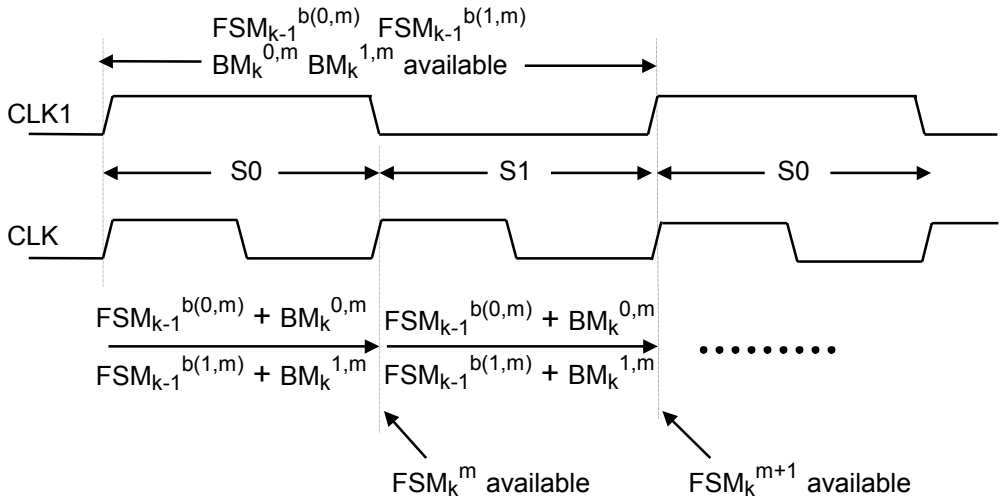


Fig. 3.9: Timing diagram of FSM calculations

Path metrics should be calculated before the E-operation is performed. Figure 3.10 illustrates how to compute a path metric PM_k by adding a BM_k to a FSM_k. As we add two positive numbers directly, the resultant FSM_k may overflow soon. To circumvent the problem, we subtract the minimum FSM of the stage k-1 from $FSM_{k-1}^{b(i,m)}$ before the addition is performed. This is possible as only the relative value of FSMs are important. After the addition, a limit circuit limits the sum to 8 bits. If there is an overflow, the sum is set to 255.

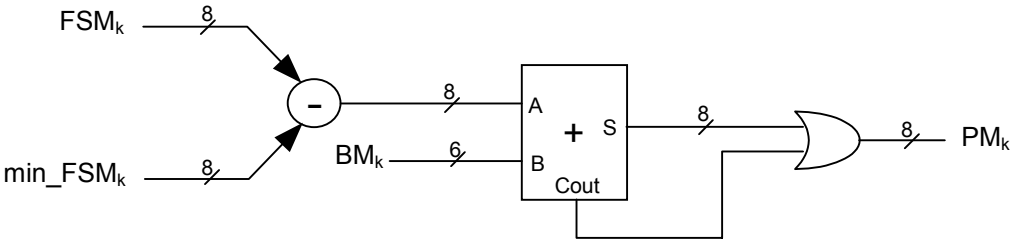


Fig. 3.10: SM adder

The E-operation is a key point in the state metric calculations. For convenience, we denote PM0 and PM1 to represent the path metrics $PM_k^{0,m}$ and $PM_k^{1,m}$, respectively. As shown in Equation (3.3), the E-operation includes finding the minimum of the two path metrics and calculation of the absolute difference of PM0 and PM1. Then through a

look up table (LUT), the adjustment function f(z) is added to the minimum value of the two path metrics to obtain an FSM. Figure 3.11 illustrates the E-operation [10]. Both PM0 and PM1 are 8-bit unsigned numbers. Adder 1 compares the magnitude of the two data as follows.

Cout1 = '1' if PM0 > PM1

Cout2 = '0' if PM0 ≤ PM1

Adder2 performs a subtraction operation, PM0 – PM1, in 1's complement form with auxiliary sign bit '0' for PM0 and PM1. The XNOR gate changes (complements) the sign if the result of (PM0 – PM1) is negative. Hence, the node U is equal to |PM0 – PM1|. The circuit LIM limits the input data within $2^6 - 1 = 63$, and the output of LIM serves as the address z of the LUT. The MUX chooses the minimum of PM0 and PM1, and the smaller one is added to the adjustment function f(z). Finally, the maximum of the sum is limited to 255 due to the OR gate, and the resultant FSM is latched.

The several truncation steps involved in the process reduce the accuracy and cause some degradation in performance. However, simulation results of MATLAB verify that the degradation is insignificant.

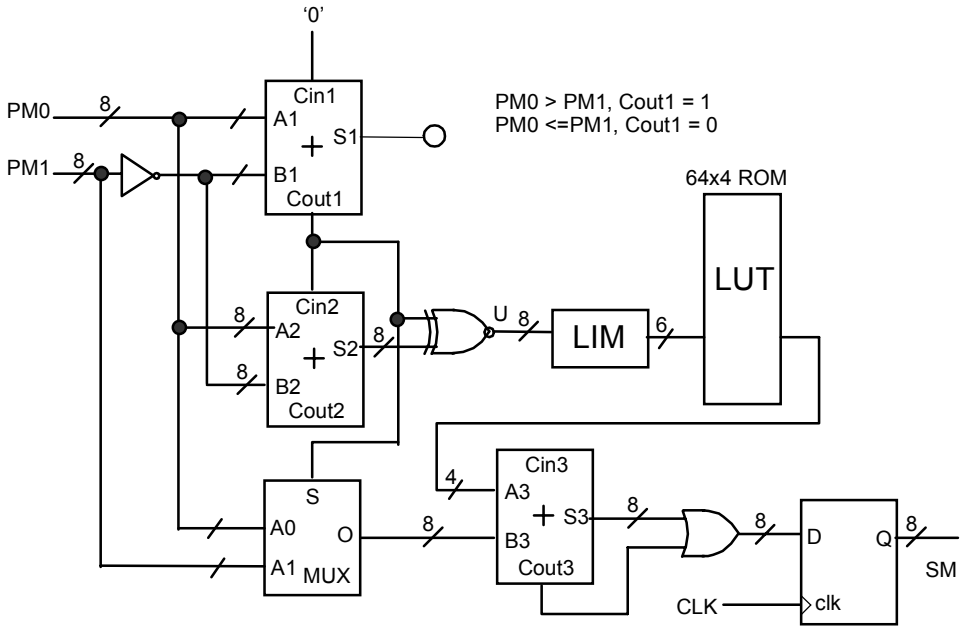


Fig. 3.11: E-operation circuit

- **Minimum SM calculation**

When a new FSM is calculated, it is applied to the FSM RAM and to the minimum state metric calculation (MinSMC) block shown in Figure 3.12. The new FSM is compared with the current minimum FSM held at DFF2 flip-flops. The smaller one between them becomes the new minimum current FSM and is stored back in DFF2 flip-flops. The set of DFF1 flip-flops are incorporated to store the minimum FSM of the previous stage k-1 while the minimum FSM is calculated for the current stage k. DFF1 flip-flops are updated only once when the computation of current FSMs are finished on every 16 CLK cycles. Initially, DFF2 flip-flops are reset to the largest number 255 to be ready for the minimum FSM calculation of the next stage.

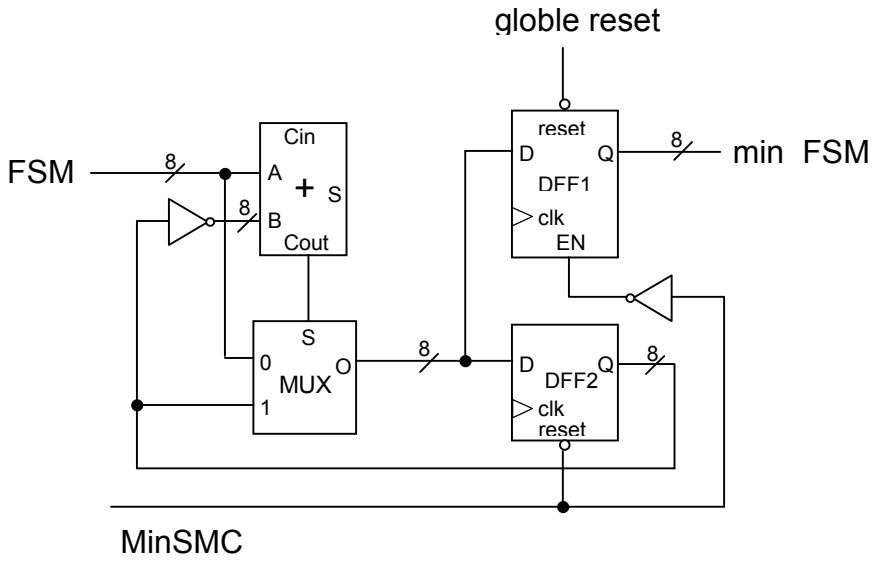


Fig. 3.12: Minimum FSM calculation block

- **Initialization for FSMs**

The initial state of a trellis at the beginning of a new frame is S_0. As the FSMs of stage 1 are based on the FSMs at stage 0, it is necessary to initialize the FSMs at stage 0. FSM_0^0 of state 0 at time k=0 should be initialized to 0. FSMs of the other 15 state FSM_0^m, m = 1, 2, ... , 15, are set to the largest number 255.

- **Storage RAM for FSMs**

A $FSM_{k-1}^{b(i,m)}$ of stage k-1 needs to be read out from a FSM RAM to calculate the FSM_k^m at stage k, while the newly obtained FSM_k^{m-1} at stage k should be written back to the same RAM at a different location for future use at the same time. Hence, a dual port static RAM that can be read and written simultaneously is used. The addresses of the memory elements are shown in Figure 3.13. As 16 FSMs are calculated for each symbol, 16(N+1) storage elements are required to store the FSMs, where N is the frame length. As N = 1023 for our decoder, it needs 16, 384 bytes of memory for the storage of FSMs. As a result, the FSM RAM in our design is a 16k x 8 bits asynchronous dual port SRAM.

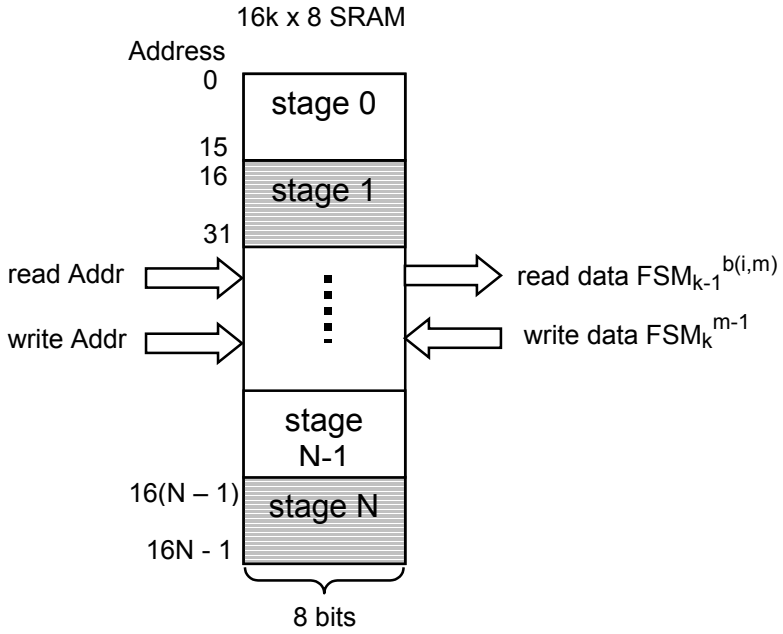


Fig. 3.13: Storage RAMs for FSMs

- **BSM Calculation**

The calculation of BSMs (Backward State Metrics) is the same as that of FSMs except the direction of the time index. A BSM_k^m of state m at time k is computed using two BSMs at time k+1 ($BSM_{k+1}^{f(0,m)}$ and $BSM_{k+1}^{f(1,m)}$) and two BMs ($BM_{k+1}^{0,m}$ and

$BM_{k+1}^{1,m}$). The equation to calculate a BSM for a partial trellis given in Figure 3.14 is as follows:

$$BSM_k^m = E\left(BM_{k+1}^{0,f(0,m)} + BSM_{k+1}^{f(0,m)}, BM_{k+1}^{1,f(1,m)} + BSM_{k+1}^{f(1,m)}\right) \quad (3.8)$$

Let us define a reverse path metric (RPM) of a transition from a state m at time k to a state with state number b(i, m) at time k-1 under input information i as $RPM_k^{i,m}$, we have:

$$RPM_k^{i,m} \equiv BSM_k^m + BM_k^{i,m} \quad (3.9)$$

Thus, Equation (3.8) can be represented using RPM:

$$BSM_k^m = E\left(RPM_{k+1}^{0,f(0,m)}, RPM_{k+1}^{1,f(1,m)}\right)$$

$$= \min\left(RPM_{k+1}^{0,f(0,m)}, RPM_{k+1}^{1,f(1,m)}\right) + f\left(\left|RPM_{k+1}^{0,f(0,m)} - RPM_{k+1}^{1,f(1,m)}\right|\right) \quad (3.10)$$

where $$f(z) = -21.6 \ln(1 + e^{-z/21.6}) + 21.6 \ln 2, \; z \geq 0 \quad (3.7)$$

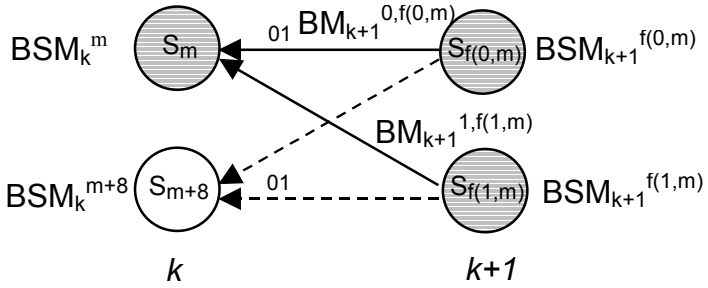


Fig. 3.14: Partial trellis for the calculation of BSM_k^m

In the equation, f(i, m) is the next possible state number for the current state m under input bit i. It is important to note that the same branch metrics are used for FSM and BSM calculations. The adjust function f(z) is the same as for FSM calculation, hence, the same LUT can be used.

From Equation (3.5) and (3.10), we can tell that BSM and FSM units can have exactly the same circuit structure except that the ROM for a BSM unit should store f(i,m)s and BMs are read from a RAM. The control logic has to decide the correct address to retrieve $BSM_{k+1}^{f(0,m)}$ and $BSM_{k+1}^{f(1,m)}$ to compute BSM_k^m at stage k from the BSM RAM. According to the partial trellis shown in Figure 3.3, two new BSMs with the state index m and m+8 at time k can be calculated by reading two consecutive BSMs and

two consecutive BMs at time k+1 from the RAMs. Therefore the control unit should generate the following read and write address sequences for BSM RAM:

Table 3.2 Read/write address sequences for the BSM RAM

Read Addr. at time k+1	0	1	2	3	4	5	6	7	9	8	11	10	13	12	15	14
Write Addr. at time k	0	8	1	9	2	10	3	11	4	12	5	13	6	14	7	15

Readers may refer to the state transition diagram in Figure 3.3 to verify the sequences.

As only four branch metrics are stored in two BM RAMs, BMs need to be read out from two RAMs based on the expected symbol "ip" to perform BSM calculation. When the BSM under calculation is BSM_k^m, the expected symbols of the two branches departuring from state m can be read out from the ROMs that store parity bit p as shown in Figure 3.4. Under i = '0', the output p of ROM0 serves as the address to read a BM from the memory BM RAM0 at stage k+1. The BM under i = '1' is read out from the memory BM RAM1 at stage k+1 based on the output of ROM1. Similar to the FSM calculation, every pair of branches ending at state m and state m+8 has exactly the same BMs. Hence, only BMs starting from state S_0 though state S_7 are necessary to be read out to perform the BSM calculation. The read operation of BMs from BM RAMs is illustrated in Figure 3.15.

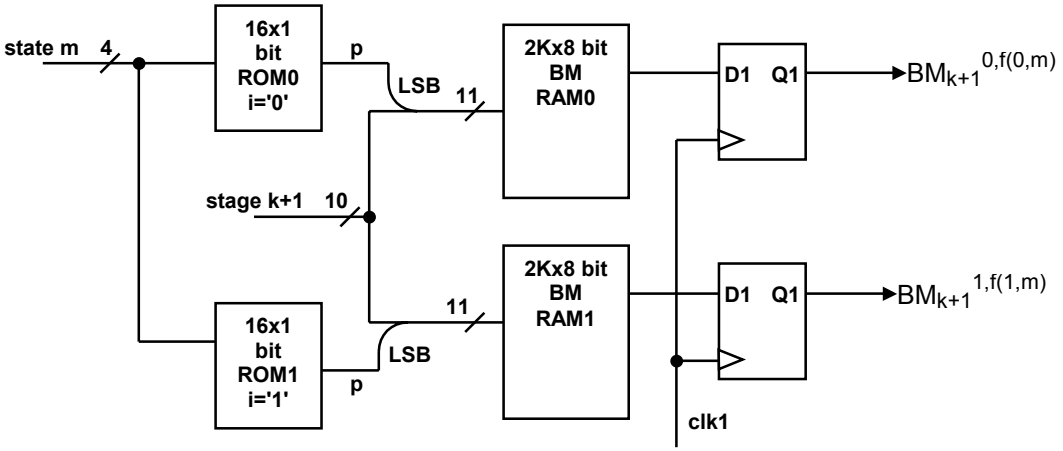


Fig. 3.15: Read operation of BM RAMs to calculate BSM

- **Initialization for BSMs**

Initialization of BSMs is different for the two decoders. The incoming input sequence is not interleaved for Decoder 1, so that the final state of Decoder 1 is S_0 with the tail bits that force the state to S_0. Hence, the BSMs in the final stage 1023 can be set as follows:

$$BSM_{1023}{}^{0} = 0, \text{ and } BSM_{1023}{}^{m} = 255, m = 1, 2, \ldots, 15$$

For Decoder 2, the final state is unknown due to the interleaved sequences. Hence, we set all state to 0. Theoretically, it can be set any value, but 0 is preferred to minimize the chances of being overflow.

- **Storage RAM for BSMs**

The type of the RAM used for storage of BSMs is identical to that of the FSM RAM, and it has the same organization of the addresses as the FSM RAM. A 16k x 8 bits asynchronous dual port SRAM is used for the BSM RAM.

3.2.4 Log-likelihood Ratio Calculation (LLRC) Module

The computation of the log-likelihood ratio LLR_{k+1} of the estimated information bit at stage k follows the equations given below:

$$LLR_{k+1} = LLR_{(k+1)0} - LLR_{(k+1)1} \tag{3.11}$$

$$LLR_{(k+1)0} = \underset{m=0}{\overset{15}{E}}(FSM_{k}^{m} + BM_{k+1}^{0,f(0,m)} + BSM_{k+1}^{0,f(0,m)}) \tag{3.12}$$

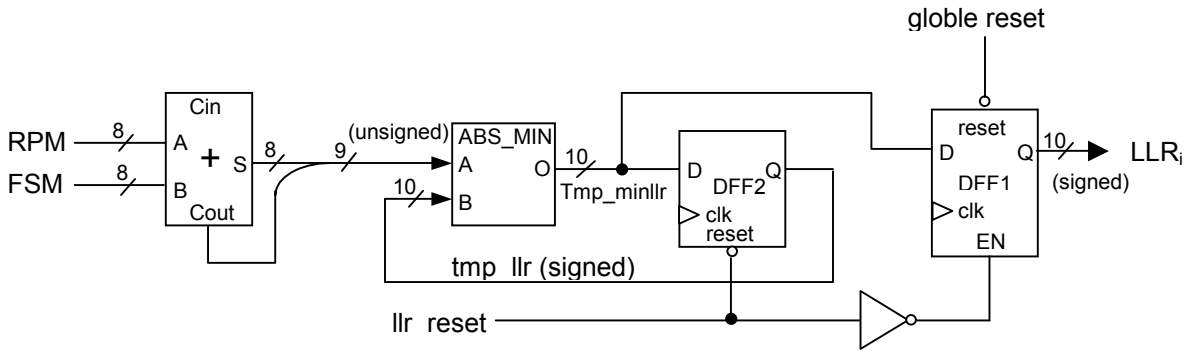
$$LLR_{(k+1)1} = \underset{m=0}{\overset{15}{E}}(FSM_{k}^{m} + BM_{k+1}^{1,f(1,m)} + BSM_{k+1}^{1,f(1,m)}) \tag{3.13}$$

Where $LLR_{(k+1)0}$ is *a posteriori* probability of each information bit i being '0' at time k+1, and $LLR_{(k+1)1}$ is for i being '0' at time k+1. Note that the E-operation involves 16 terms. For illustration purpose, we use a log-likelihood ratio metric (LLRM) to express a term in $LLR_{(k+1)i}$ calculation:

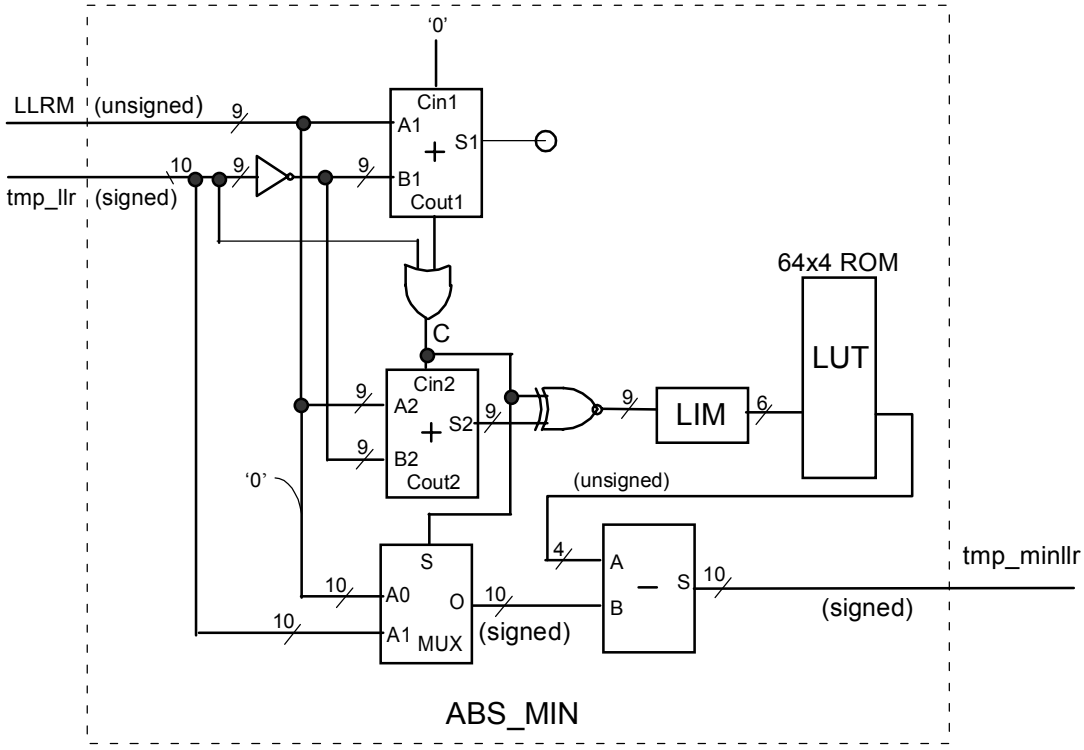
$$\begin{aligned} LLRM_{k+1}^{i,f(i,m)} &\equiv FSM_{k}^{m} + BM_{k+1}^{i,f(i,m)} + BSM_{k+1}^{i,f(i,m)} \\ &\equiv FSM_{k}^{m} + RPM_{k+1}^{i,f(i,m)} \end{aligned} \tag{3.14}$$

The computation of an LLRM involves an addition of a RPM and an FSM. Two reverse path metrics $RPM_{k+1}^{0,f(0,m)}$ and $RPM_{k+1}^{1,f(1,m)}$ (Refer to Equation (3.9).) are available as we move backward to calculate the BSM_k^m (Refer to Equation (3.10).) Hence, only an FSM_k^m of state m at time k is necessary to be read from the FSM RAM on each CLK cycle. The state number m follows the sequence 0, 8, 1, 9, …, 7, 15, which is of the same order to compute the BSM. As the two RPMs are obtained at the same time, $LLRM_{k+1}^{0,f(0,m)}$ and $LLRM_{k+1}^{1,f(1,m)}$ need to be computed in parallel on each CLK. Hence, two identical LLRC modules are used to calculate the 2 LLRMs and perform the E-operations separately. One of the two modules is illustrated in Figure 3.16.

As mentioned in Section 2.4.3, the E-operation with multiple terms can be calculated accurately by recursively calculating two terms each time using Equation (2.29). Therefore, the E-operation of LLR calculation is performed by a module that is similar to the one used in the FSMC module (Refer to Figure 3.11) as shown in Figure 3.16(b)): Once an LLRM is available, we compare it with the current term held at DFF2 flip-flops. Select the smaller one between them, adjust the selected one according to the adjustment function f(z) and store the intermediate data back into DFF2 flip-flops on each CLK cycle. Since one LLRM is available at one time for an LLRC module and there are totally 16 LLRMs for each $LLR_{(k+1)i}$ calculation, the final $LLR_{(k+1)i}$ for stage k+1 is available at every 16 CLK cycles. Refer to the timing diagram shown in Figure 3.17. In this way, we recursively calculate $LLR_{(k+1)i}$. The set of DFF1 flip-flops in Figure 3.16(a) are incorporated to store the previously calculated $LLR_{(k+2)i}$ for stage k+2. DFF1 flip-flops are updated only once when the computation of the final $LLR_{(k+1)i}$ at stage k+1 is finished. It should be noted that the set of DFF2 flip-flops are initialized to the maximum value 511.



(a) Overall block diagram



(b) Difference and minimum calculation circuit of the LLR module

Fig. 3.16: LLR calculation module

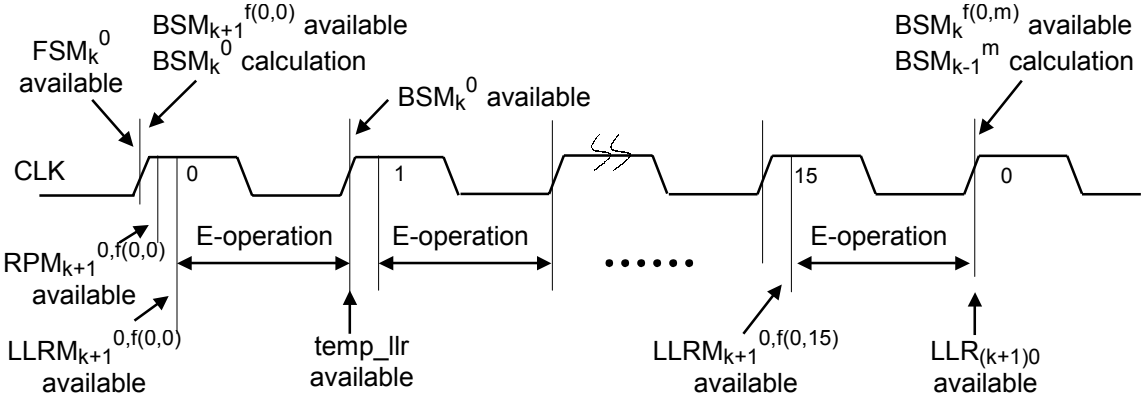


Fig. 3.17: Timing diagram of LLR_0 calculation

However, the LUT for f(z) is different from that used in the SMC. Since we are performing E-operations on more than two terms, we cannot add the bias cln2 to Equation (3.6) as we did in the state metric calculation. In order to keep the elements in the f(z) LUT positive, we adopt the original form of the E-operation as given in Equations from (2.25) to (2.27). For convenience, they are listed again as follows:

$$E(a, b) \equiv - \log_{\in}(\in^{-a} + \in^{-b}) = \min(a, b) - f(|a-b|); \quad (2.25)$$

$$f(z) = c \ln(1 + e^{-z/c}), z \geq 0 \quad (2.26)$$

$$c = \log_{\in} e \quad (2.27)$$

As the size of the LUT is the same as in the state metric calculation, c is kept at 21.8. The resultant output of the LUT is subtracted from the minimum term during the E-operation. The subtraction may result in a negative number. Therefore 10-bit signed number is used to represent the output of the E-operation. Note that LLRM is always positive while the intermediate term in the E-operation can be positive or negative. The output C of an OR gate whose inputs are Cout1 and the sign bit of the intermediate term determines the minimum value between LLRM and tmp_llr. Similar to the criteria shown in the FSM calculation, we have:

C = '1' if LLRM > tmp_llr

C = '0' if LLRM ≤ tmp_llr

When $LLR_{(k+1)0}$ and $LLR_{(k+1)1}$ are available after 16 CLK cycles, we subtract $LLR_{(k+1)1}$ from $LLR_{(k+1)0}$ based on the Equation (3.11) to obtain LLR_{k+1}. The subtraction circuit handles 2 10-bit signed number and outputs the 10-bit signed number LLR_{k+1}. Note that LLRs are calculated with the backward state metrics, so the LLRs generated by the decoder are in reverse order. The design of the interleaver and de-interleaver should take this factor into consideration in the turbo decoder.

3.2.5 Control Module for the log-MAP Decoder

The control module of our log-MAP decoder generates reset signals, control signals and addresses to read and write the RAMs. We have already described part of the control logic for read/write address generation in previous sections. Each decoding process can be partitioned into two parts: The first half of the decoding process is for the calculation of both BMs and FSMs; the second half calculation is to obtain BSMs and LLRs. The timing of all the read/write operations in one decoding stage is shown in Figure 3.18.

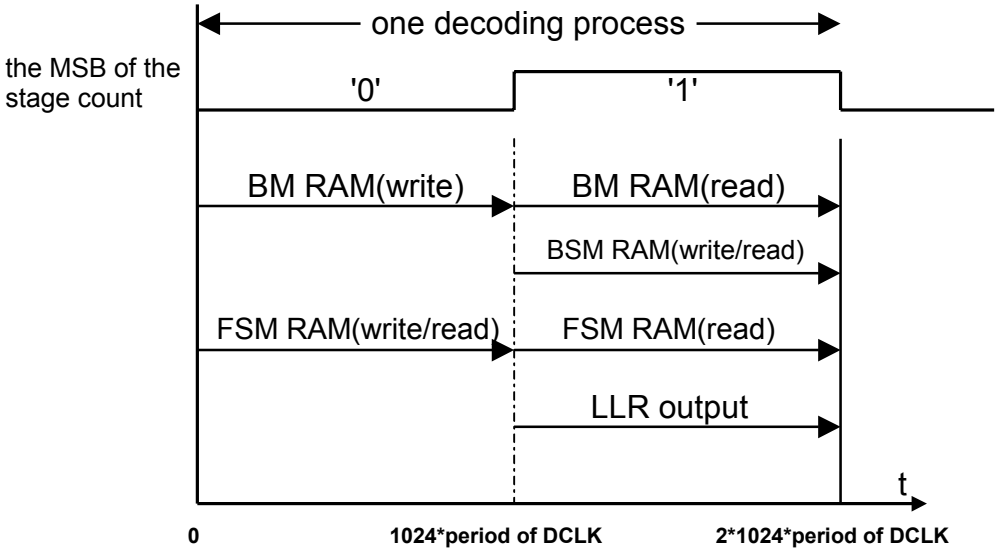


Fig. 3.18: The memory access timing diagram of a log-MAP decoder

During the first half of the process, the controller provides a write enable signal and addresses to write the newly calculated BMs into the two 2K bytes BM RAMs. The FSM calculation needs to read the FSMs of the previous stage from the FSM RAM and to write back into the same RAM simultaneously. Hence a dual port 16K bytes RAM is used (refer to Section 3.2.3). A timing diagram of the FSM calculation is shown in Figure 3.19. An FSM is read from the RAM and a calculated FSM is stored back on every clock cycle. Note that two FSMs are read and two FSMs are computed and stored back for each butterfly. There is latency of 3 CLK cycles between the read of the first FSM of the previous stage k-1 in a butterfly and the write of the first new FSM at stage k in the same butterfly back into the RAM.

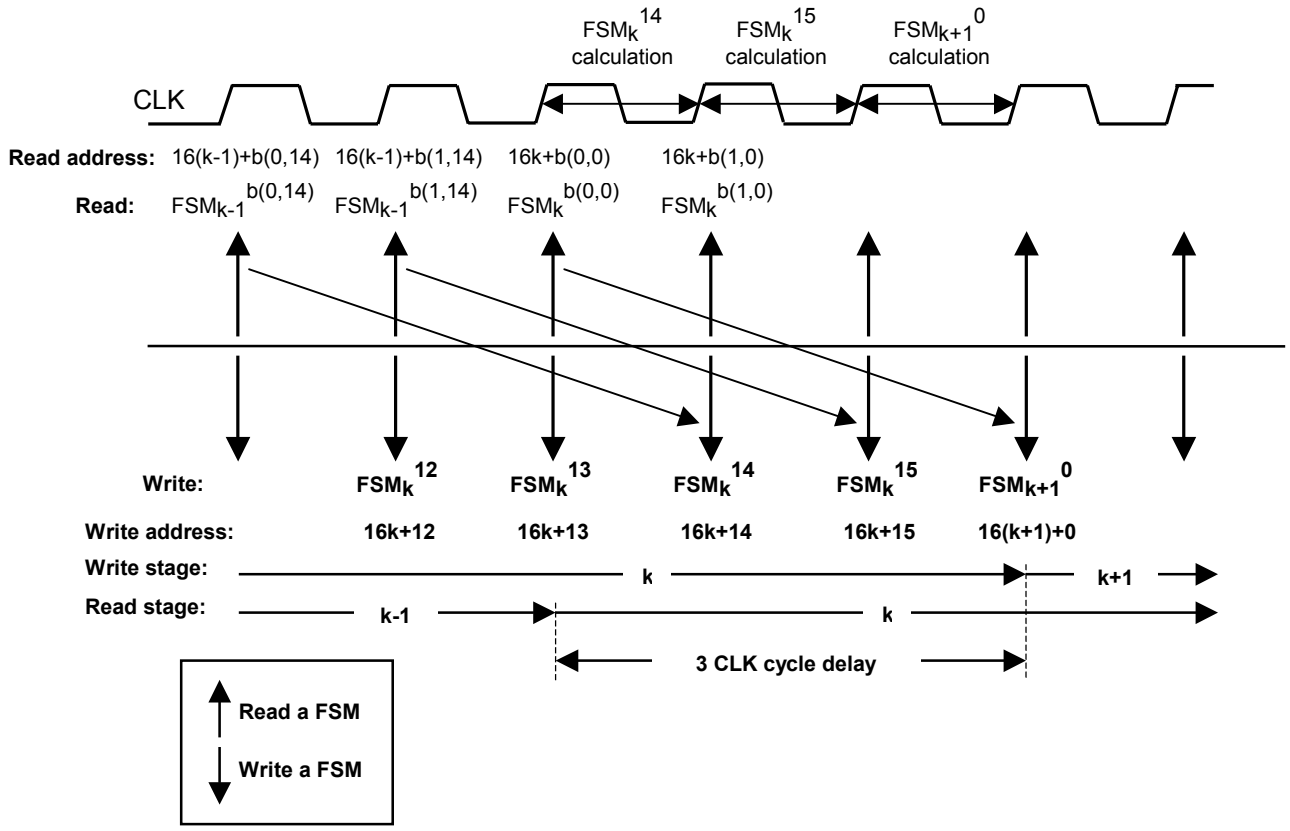


Fig. 3.19: Timing diagram of the read and write of the FSM RAM in FSM calculations

During the second half of the process, we trace the trellis backward from the last stage to calculate both of the BSMs and the LLRs. The read enable signal and addresses are provided by the controller to read the BMs from the BM RAM to calculate BSMs. Similar to the FSM calculation, the BSMs of the previous stage are read from the BSM RAM at the same time as the newly computed BSMs are written back into the same RAM. Hence, the BSM RAM is also a 16K bytes dual port RAM. There is also latency of 3 CLK cycles between the read of the first BSM of the previous stage in a butterfly and the write of the first new BSM in the same butterfly back into RAM. Meanwhile, the reverse path metrics generated during the calculation of BSMs are used to compute the LLR. To complete the LLR calculation, the corresponding FSMs need to be read out from the FSM RAM on every CLK cycle. Since the data for the FSM were written in the ascending address, the addresses for the read and write operations of the RAM in the backward processing should be in the descending order.

We explain the address generation for the FSM RAM in the controller, which is more complex than that of the BM RAM and the BSM RAM. The read address of the FSM RAM is supplied by an 11-bit stage counter and a 4-bit free running up-counter. The most significant bit (MSB) of the state up-counter indicates if the current operation belongs to the first half (in which FSMs are computed) or the second half (in which BSMs are computed) processing. A block diagram of the address generator for the FSM RAM is shown in Figure 3.20.

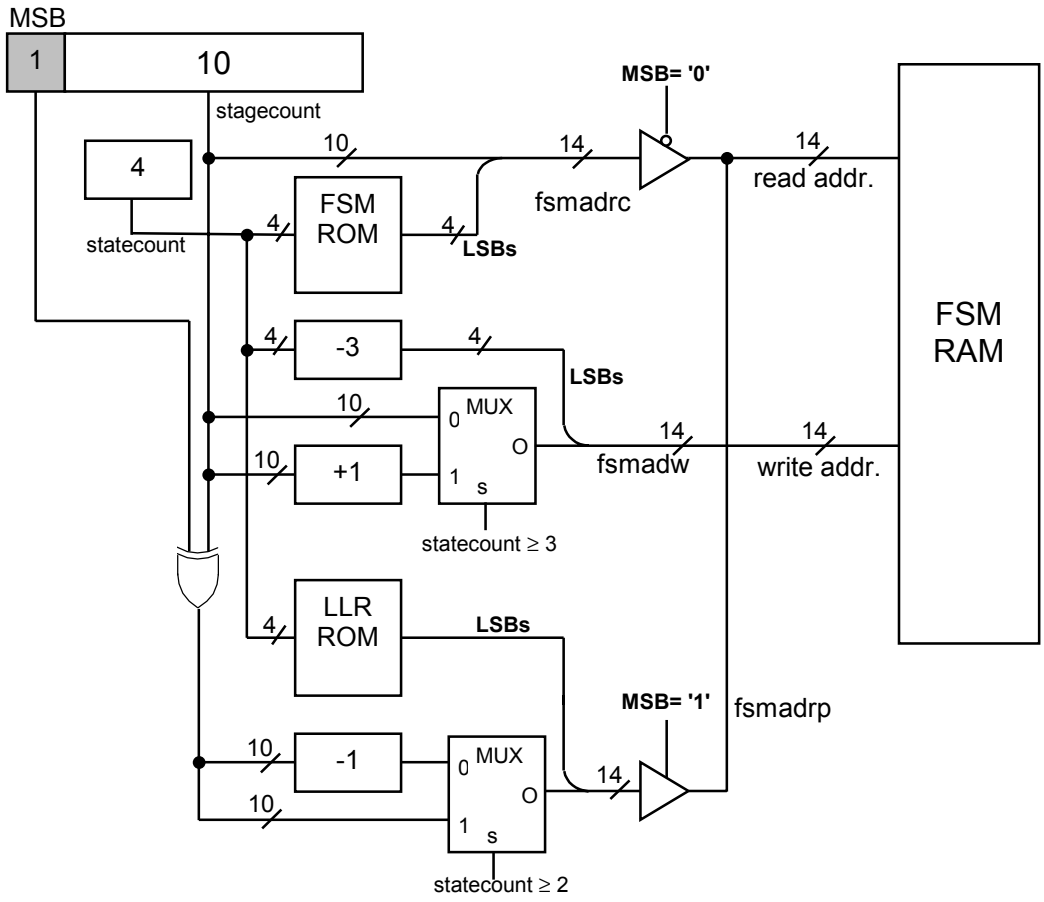


Fig. 3.20 FSM RAM address generation

In the figure, the FSM ROM (We call it last_state ROM in our VHDL code.) specifies the order of address sequence in the FSM calculations within one stage. The LLR ROM (we call it next_state ROM in our VHDL code.) specifies the order of addresses to read FSMs during the calculation of LLRs in the second half decoding process. During the FSM calculation, the read address is denoted as *fsmadrc* in the figure. *fsmadrc* is a combination of the output of the stage count and the output of the FSM RPM. For convenience, we call the output of the stage count and the state count as *stagecount* and *statecount*, respectively. The write address FSM RAM is denoted as *fsmadw* in Figure 3.20.

Suppose that we start to compute FSMs at stage k+1. So we read FSMs at stage k for the computation. However, due to the latency we mentioned previously (Refer to Fig.

3.8.), the last three FSMs of stage k are still being written to the memory bank of the FSM RAM of stage k. This means that read and write operations happen simultaneously at the memory bank of stage k for the first three CLK cycles. In terms of write address, the first three FSMs should be written into the memory bank of k and the remaining FSMs (which belong to stage k+1) written into memory bank of (k+1). Note that FSMs should be read from the memory bank of (k+1) as we are computing FSMs of stage k+1. The write address is generated as follows.

i) state counter < 3

write address = stagecount & (statecount – 3)

ii) state counter ≥ 3

write address = (stagecount + 1) & (statecount – 3)

In the above, the symbol "&" is concatenation operation. The write address is a concatenation of 10 bits of the stagecount and 4 bits of (statecount-3). The multiplexer MUX1 in Figure 3.20 performs the operation.

During the backward operation, FSMs need to be read out again. The read address of FSMs for the LLR calculations is denoted as *fsmadrp*. Since the stage number should decrease for LLRs, XOR operations are performed with the MSB of the stage counter and every bit of the other 10 bits of the stage counter so that the output of the stage counter (denoted as *rev_stagecounter*) is complemented. Hence, the stage number decreases. There is a delay of 2 CLK cycles between the read of FSMs and the write of the corresponding BSMs. The multiplexer MUX2 takes care of the delay. If statecount ≥ 2, *fsmadrp* is the combination of *rev_stagecounter* and the output of the LLR ROM. Otherwise, *fsmadrp* is the concatenation of (*rev_stagecounter* – 1) and the output of the LLR ROM. Finally, the two read addresses, *fsmadrc* and *fsmadrp*, are resolved using two tri-state buffers to generate the read address of the FSM RAM. Generation of the read and write addresses for BSMs is similar.

Besides the control of the RAMs, the controller also produces reset signals for the different modules to synchronize with each other. The reset signals come from the combination of the stage counter and the state counter and refresh the registers in the

modules such as the minimum SMC and LLRs on every 16 CLK cycles. It also generates the signal to shut down an inactive component decoder in the turbo decoder to be explained in Section 3.3.3.

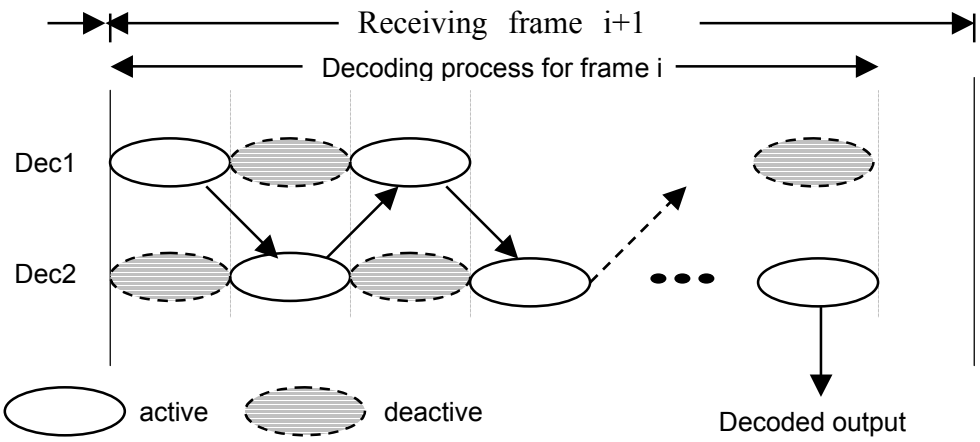
3.3 Turbo Decoder Implementation

In this section, we describe the implementation of our iterative turbo decoder based on the component log-MAP decoder presented in the previous section.

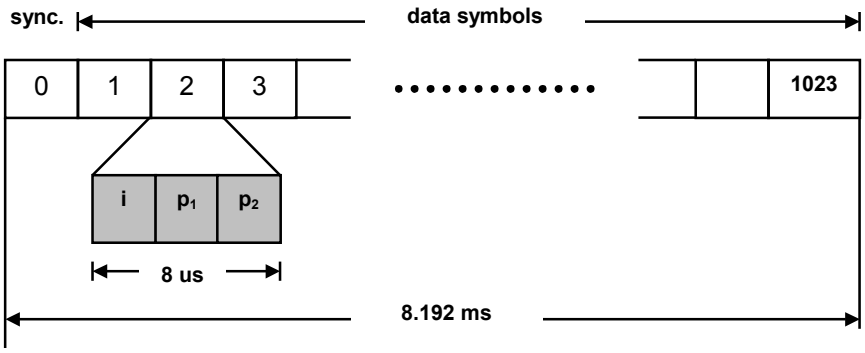
3.3.1 Overview of the Overall Operation of the Turbo Decoder

We explain the overall operation of the turbo decoder given in Figure 3.21(a) briefly. The decoder receives the first frame of data and stores them in SRAMs. Both component decoders are idle during the time. In the second frame, the channel data is received and stored in another SRAMs. At the same time, Decoder 1 starts to decode the data of the first frame, while Decoder 2 is idle, waiting for *a priori* probability (AprP) from Decoder 1. As Decoder 1 finishes the process, the extrinsic information is passed to Decoder 2. Decoder 2 starts processing, while Decoder 1 is now idle. When Decoder 2 finishes the process, it completes one iteration. When the decoder receives the third frame of data, the two component decoders start to decode the second frame of data in the same manner. When n iterations are necessary, the same process repeats n times. However, in order to limit the latency of decoding within one time frame, the n iterations should be finished within one frame time.

The format of a frame is shown in Figure 3.21(b). The first symbol is for synchronization and is ignored by component decoders. The remaining 1023 symbols bear data information, and each symbol consists of an information bit and two parity bits. Each symbol is 8 μs long (which is 125 KHz) and each frame is 8.192 ms.



(a) Pipelined architecture for turbo decoder



(b) Format of a frame

Fig. 3.21: Overall operation of the decoder

3.3.2 Top level turbo decoder structure

In our design, we adopt the serial implementation, which was discussed in Chapter 2. In addition to the two log-MAP decoders described in the previous section, the turbo decoder needs an interleaver, a de-interleaver, subtraction/limit blocks, and the RAMs to store the received signals for later processing. Fig 3.22 illustrates a block diagram of an iterative turbo decoder. The three inputs of the turbo decoder are *sys*, *par1*, *par2*, where *sys* is the uncoded information bit, *par1* is the non-interleaved parity bit, and *par2* is the interleaved parity bit. The output of Decoder 1 (2) is the log-

likelihood ratio L_all_1 (L_all_2), which becomes extrinsic information L_e1 (L_e2) after the subtraction and limit operation. After the extrinsic information L_e_n, n=1,2, is interleaved or de-interleaved, it becomes the *a priori* probability (AprP) information L_a_n for the next stage decoder.

Before we describe the operation of the turbo decoder, it is necessary to explain major clocks employed in the system. The clocks are summarized in Table 3.3. The prefix "F" denotes "fast", and the prefix "S" denotes "slow". It should note that four F-clocks, FCLK, FCLK1, FCLK2, and FDCLK are mapped to CLK, CLK1, CLK2, and DCLK of a component decoder, respectively.

Table 3.3 Clock signals used in the top level design

Name	frequency	phase	Remark
FCLK	40 MHz	0°	System clock
FCLK1	20 MHz	180°	
FCLK2	20 MHz	0°	
FDCLK	2.5MHz	0°	The data clock of component decoders
SDCLK	125 KHz	arbitrary	The data clock of the input signals

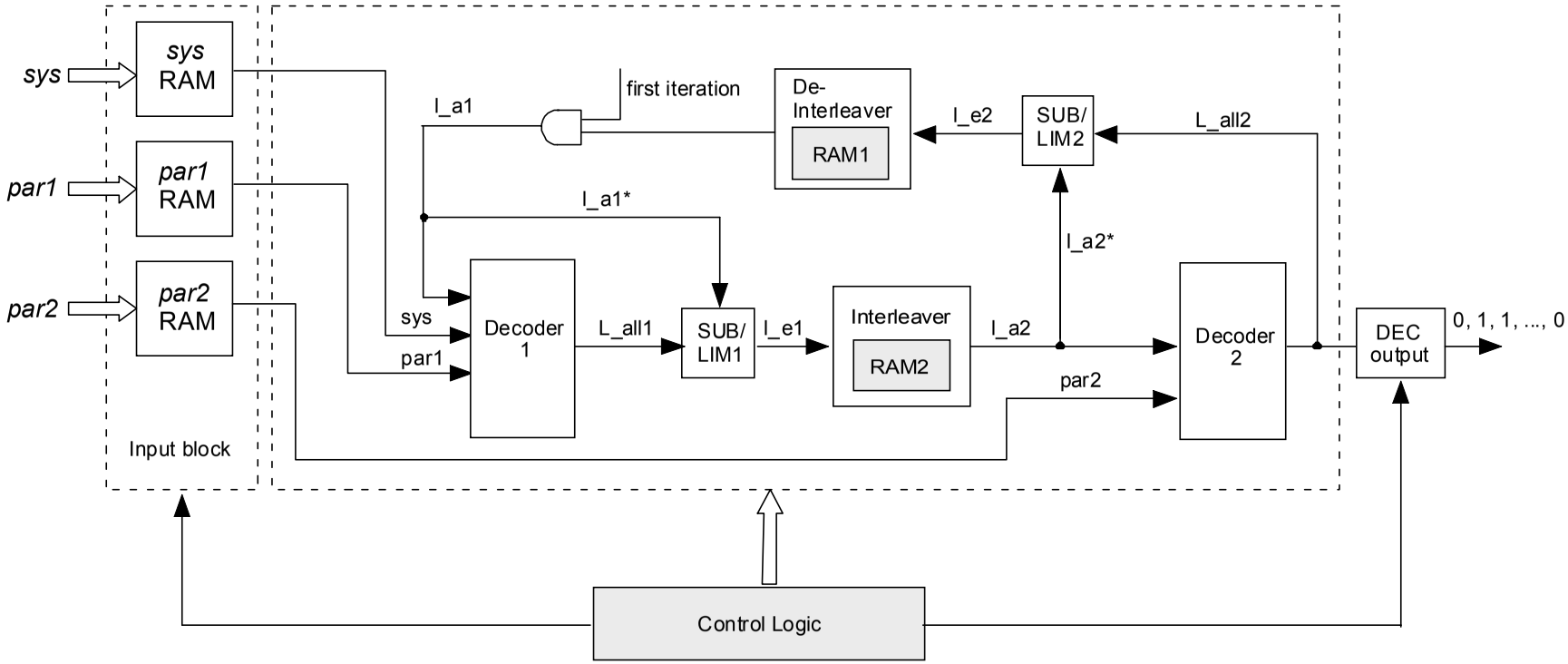


Fig 3.22 Top level turbo decoder block diagram

The operation of the turbo decoder shown in Figure 3.22 is described briefly.

- **Input blocks**:

The encoded symbol is transmitted serially through a channel. At the receiver side, the three elements, sys, par1, par2, arrive serially, and they are converted into three parallel sequences of symbols (sys, par1, par2) by a serial-to-parallel converter. The parallel signals, sys, par1 and par2 are quantized to 7 bit signed number ranging from –64 to 63, and the clock frequency of the three signals is SDCLK. Our turbo decoder receives 1023 input symbols in the first frame and stores the 3 parallel sequences in three 1K x 7-bit SRAMs. We call the group of the three RAMs that store sys, par1 and par2 as "Group A RAMs". In the second frame, the data is received and stored in the other group of RAMs called "Group B RAMs". At the same time, the first component decoder, Decoder 1, starts to decode the data of the first frame by reading sys and par1 under a fast data clock FDCLK.

- **Log-MAP decoders**:

This is the core components of our turbo decoder. There are two log-MAP decoders connected in a pipelined structure. Decoder 1 accepts data sequences *sys* and *par1* and produces a log-likelihood ratio L_all1. We represent the decoding procedure as follows:

$$[\text{sys, par1}] \Rightarrow \text{L\_all1} \tag{3.15}$$

where the arrow represents the log-MAP decoding algorithm. Initially, the *a priori* probability L_a1 from Decoder 2 is reset to 0. After Decoder 1 finishes forward processing, it starts to output the LLRs (denoted L_all1) during the backward processing. As mentioned in Section 3.2.4, the LLRs from a component decoder is represented as 10-bit signed numbers. The extrinsic information, which is the improved estimate of the information bits, is obtained by a subtraction operation:

$$\text{L\_e1} = \text{L\_all1} - \text{L\_a1} \tag{3.16}$$

In the circuit shown in Figure 3.22, the SUB-LIM block performs this operation. After we subtract L_a1 from L_all1 to obtain a 10-bit signed number L_e1, the limit circuit truncates and limits the value in the range of –128 ~ 127 by using an 8-bit signed number. Before L_e1 is applied to Decoder 2 as the *a priori* values, the sequence has to be

interleaved to match the order of the sequence *par2*, which is the interleaved parity sequence. We denote the interleaved sequence as l_a2. Following the express in Equation (3.15), the decoding process of Decoder 2 is:

$$[l\_a2, par2] \Rightarrow L\_all2 \tag{3.17}$$

where L_all2 is the *a posteriori* probability (APP) LLR of the interleaved information sequence. The extrinsic information of Decoder 2 is obtained by subtracting L_a2 from L_all2:

$$L\_e2 = L\_all2 - L\_a2 \tag{3.18}$$

The following de-interleaver block reorders the sequence of L_e2 to generate the APrP information L_a1. L_a1 is stored in a RAM temporarily and is applied to Decoder 1 along with two input sequences *sys* and *par1* to start the second iteration. As we can see, due to de-interleaving, L_a1 is weakly correlated with *sys* and *par1*. Therefore, when it is passed onto the next iteration, it becomes an additional information that has weak correlation with the other inputs *sys* and *par1*. In this manner, the turbo decoder eliminates the effect of burst errors.

Each frame has one synchronization symbol followed by 1023 data symbols. Each symbol arrives on every SDCLK of 125 KHz and Decoder1 and Decoder2 operate at a higher clock frequency, FDCLK of 2.5 MHz. It takes (16*1023 + 3) FCLK clock cycles for each forward or backward operation. Note that the frame length is 1023 data symbols and the number of states is 16. The three FCLK clock cycles are due to the latency of each pipelined operation. To synchronize the forward and backward operations with FDCLK, which is 16 times slower than FCLK, the latency is increased to 1 FDCLK cycle. (So that a component decoder is idle for 13 FCLK cycles at the end of each operation.) Under the provision, each forward or backward operation takes 1024 FDCLK cycles. It implies that each iteration takes 4*1024 = 4096 FDCLK cycles, which is the minimum time required to decode one frame of 1023 symbols running at SDCLK. To enable n iterations with one time frame, the following equation holds:

$$\text{Period of a frame} = \frac{1}{f_{SDCLK}} \times 1024 \geq 4096 \times \frac{1}{f_{FDCLK}} \times n \tag{3.19}$$

Hence, the clock frequency of input data f_{SDCLK} is obtained as:

$$f_{SDCLK} \leq \frac{f_{FDCLK} \times 1024}{4096 \times n} \tag{3.20}$$

The maximum allowable input data clock frequency, SDCLK, for n= 1, 2, …, 10 iterations for FDCLK = 2.5 MHz is given in table 3.4. Note that the input data frequency is in fact the input data rate of the turbo decoder. For our turbo decoder, the data clock frequency SDCLK is set to 125 KHz. Hence, it can have up to five iterations.

Table 3.4 Input data frequency for n iterations under FDCLK = 2.5MHz

No. of iterations	Frequency of SDCLK
1	625.0000 KHz
2	312.5000 KHz
3	208.3333 KHz
4	156.2500 KHz
5	125.0000 KHz
6	104.1667 KHz
7	89.2857 KHz
8	78.1250 KHz
9	69.4444 KHz
10	62.5000 KHz

3.3.3 Interleaver and De-interleaver in Turbo Decoder

The concept of interleaving was introduced in Chapter 2. In this section, we illustrate the implementation of an interleaver and a deinterleaver for our turbo decoder. Only an interleaver is used to encode the information bits to generate parity bits *par2* at the encoder side. Both an interleaver and a de-interleaver are necessary at the decoder side. Let us describe an interleaver design at the encoder side first.

The interleaver used in our design consists of a pseudo-random noise (PN) sequence generator, a synchronous up-counter and an SRAM. We illustrate our design

with a smaller size frame, which is $2^3 - 1 = 7$ bits/frame. Suppose we choose a primitive polynomial of degree 3 (any primitive polynomial can be used). The connection polynomial chosen and the structure of PN are shown in Equation (3.21) and Figure 3.23, respectively. The initial state of the PN generator is the all-1 state.

$$P1(x) = x^3+x+1 \tag{3.21}$$

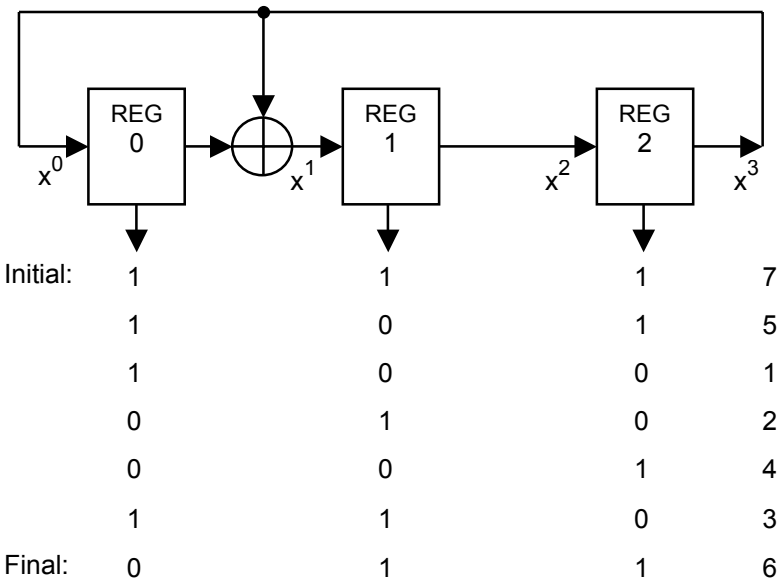


Fig. 3.23: PN generator structure

The input sequence is stored in the pseudo-random order determined by the PN generator and read in an ascending order using an up-counter. An example of how to obtain an interleaved sequence is illustrated in Figure 3.24.

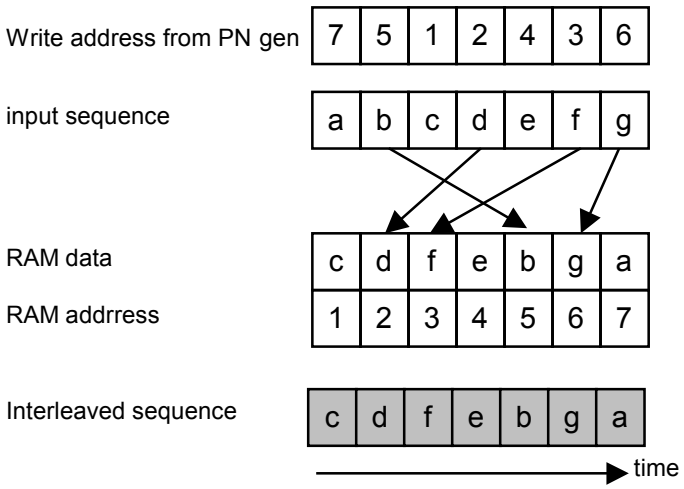


Fig. 3.24: Example interleaver at the encoder side

Now, let us consider an interleaver design of the turbo decoder. The APP (*a posteriori* probability) information from Decoder 1 is interleaved before it is applied to Decoder 2. The order of the interleaved sequence should match with that of the interleaver of the encoder, i.e., the connection polynomials of the two interleavers should be identical. However, as the log-likelihood ratios (LLRs) are generated in the reverse order of the stage index, the LLRs should be reversed in time before they are applied to the interleaver. The reverse operation incurs the decoding latency and hardware complexity. Hence, we propose design of an interleaver to incorporate the reverse order of LLRs. The key idea is to use the reciprocal polynomial of the PN generator.

The reciprocal polynomial f*(x) of f(x) with order n is defined as

$$f^*(x) = x^n f(1/x)$$

The reciprocal polynomial P2(x) of P1(x) in Equation (3.21) is obtained as

$$\begin{aligned} P_2^*(x) &= x^3 P(\frac{1}{x}) \\ &= x^3(\frac{1}{x^3} + \frac{1}{x} + 1) \\ &= 1 + x^2 + x^3 \\ &= x^3 + x^2 + 1 \end{aligned} \tag{3.22}$$

The PN generator of the reciprocal polynomial generates exactly the reverse sequence of the original generator. A reciprocal PN generator can also be obtained by simply reversing the signal flow of the original PN generator. The reciprocal PN generator of the PN generator in Figure 3.23 is obtained through the reversal of the signal and is shown in Figure 3.25. Note that the initial state of the reciprocal PN generator is the final state (=011) of the original PN generator. *In summary, the PN generator for the interleaver of the turbo decoder should be the reciprocal of the PN generator for the interleaver of the encoder.*

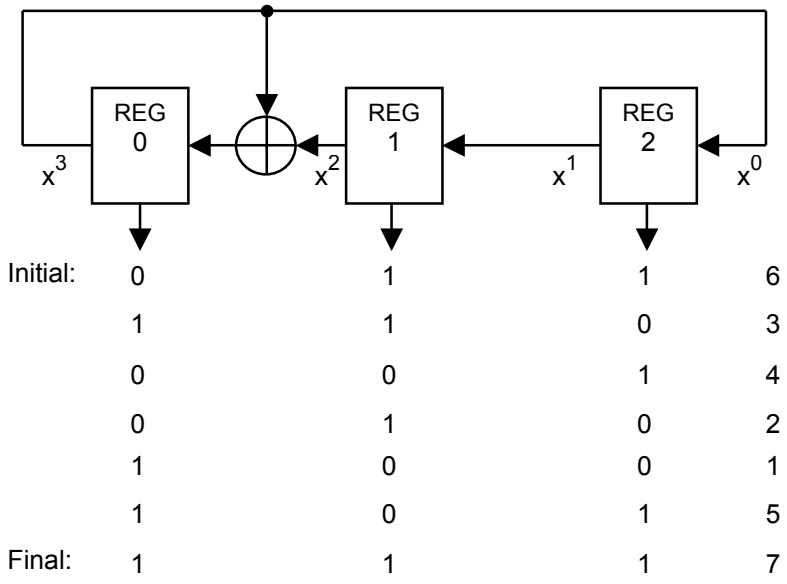
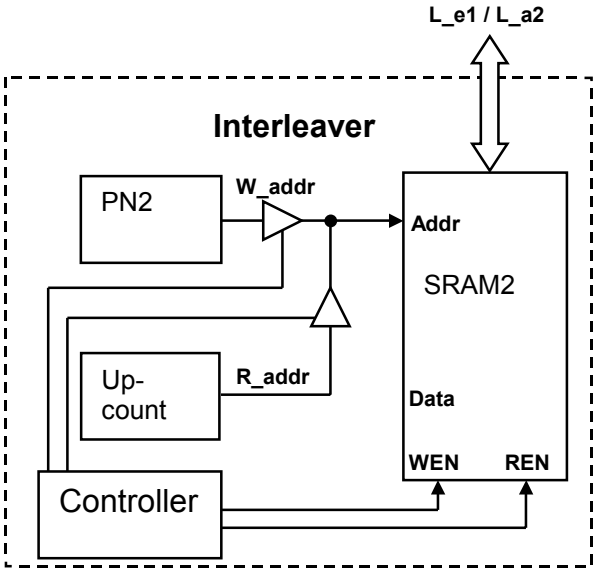


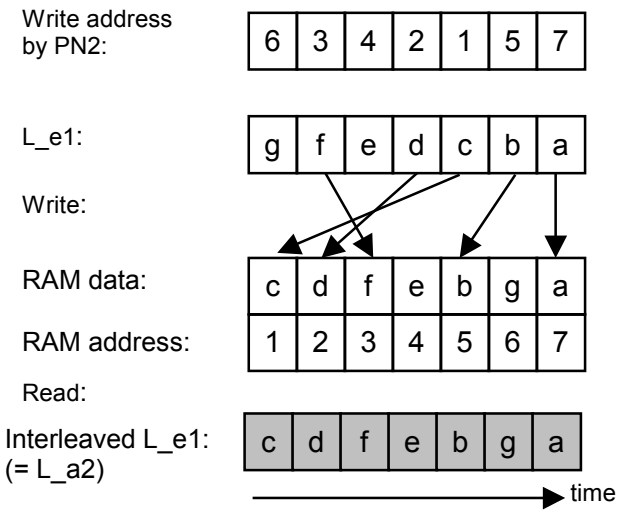
Fig. 3.25: Reversed signal flow PN generator structure

The architecture of the interleaver in our turbo decoder is shown in Figure 3.26(a). We name the reciprocal PN generator used in the interleaver as PN2. Once the extrinsic information L_e1 from Decoder 1 is calculated, it is written to RAM2 based on the pseudo-random address generated by PN2. When Decoder 2 starts the decoding process, an up-counter produces the address to read the extrinsic information L_e1 from RAM2, which yields an interleaved sequence of L_e1 denoted as L_a2. Because the read and the write addresses of RAM2 are from different sources: one from PN generator, the other from up-counter, tri-state buffers are used to resolve the RAM addresses. The control signals from the system controller control the write/read enable signals and the

time to enable the tri-state buffers. As the LLRs are generated under clock FDCLK, the up-counter and PN2 also run under the same FDCLK. Figure 3.26(b) illustrates an interleaving operation with the polynomial given in Equation (3.23). We find the order of the interleaver L_a1 in Figure 3.26(b) match with the order of the interleaved *par2* in Figure 3.24. Hence, the problem of reversed LLRs is solved by using a reciprocal polynomial function.



(a) Interleaver



(b) Example interleaver

Fig. 3.26 The structure of the interleaver and its functionality in turbo decoder

After Decoder 2 finishes decoding, the extrinsic information L_e2 should be de-interleaved, and its original order (identical to the order of *sys*) needs to be recovered. The order of L_a2 (which is the interleaved L_e1 sequence) to Decoder 2 is "cdfebga" as shown in Fig 3.26(b). We need to recover it to the sequence "abcdefg" after the de_interleaving. The LLRs are generated in the order of "agbefdc", which is the reverse of the input sequence. (This necessitates the use of a down-counter to read L_a2 for the "SUB/LIM" block in Figure 3.22.) The LLRs are written into RAM1 using the address generated by a down-counter. The contents of the RAM are "cdfebga" in the ascending order of the address after the write operation. Since the PN2 is used to interleave the extrinsic information of L_a2, we should used the reciprocal PN generator of PN2 for the de-interleaving. The reciprocal of PN2, called PN1, is used to read RAM2 of the deinterleaver. The de-interleaved sequence of L_e2, which is L_a1, is "abcdefg". The circuit sturcture and the initialization of PN1 is the same as shown in Figure 3.23.

The de-interleaving process is illustrated in Figure 3.27(b): Once the extrinsic signal from Decoder 2 is available, it is written to an 1k x 8-bit SRAM (denoted as RAM1) with the address generated by the down-counter. For example, the first decoded data "a" is written to the last address, which is 7 in the this example, the second data "b" is written to address 6 and so on. After all the extrinsic information for a frame is stored into RAM1, PN1 provides the addresses to read out the data from RAM1. Note that the order of the data read from the RAM is the same to the order of *sys*.

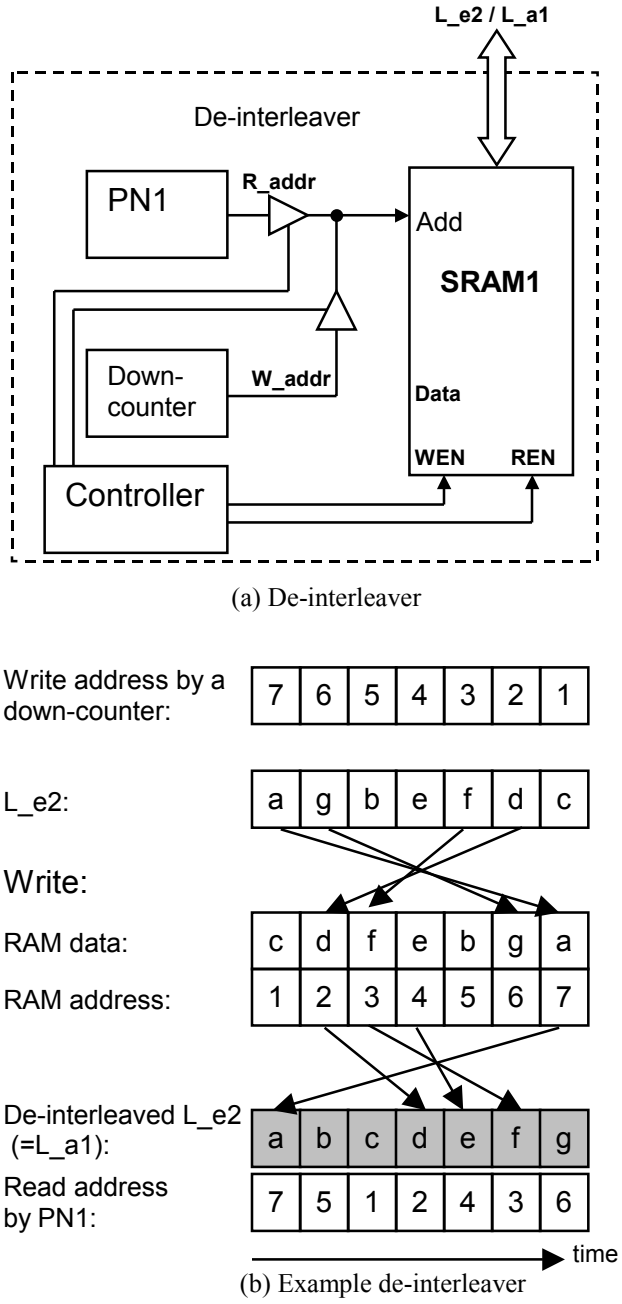


Fig. 3.27 The structure of the de-interleaver and its functionality in turbo decoder

In our turbo decoder design, the polynomial function P1(x) and P2(x) are of degree 10 to generate 1023 pseudorandom addresses. P1(x) and P2(x), which are used in PN1 and PN2, respectively, are given in Table 3.5. Note that PN1 is used both in the de-

interleaver of the turbo decoder and the interleaver in the turbo encoder, and PN2 is employed in the interleaver of the turbo decoder.

Table 3.5 Primitive polynomial function used in interleaver/de-interleaver

	Polynomial function	Initial state	Final state	Remark
P1(x)	$P1(x) = x^{10} + x^7 + x^3 + x + 1$	"1010111011" = 699	"1111111111" = 1023	Employed in the de-interleaver in the turbo decoder and in the interleaver of the encoder
P2(x)	$P2(x) = x^{10} + x^9 + x^7 + x^3 + 1$	"1111111111" = 1023	"1010111011" = 699	Employed in the interleaver of the turbo decoder

Finally, we should mention the address generations for the interleaver and the de-interleaver for SUB/LIM blocks. The address of the de-interleaver in Figure 3.22 is generated by PN1 for the extrinsic information L_a1 of Decoder 1. As LLRs of Decoder 1, L_all1, are generated in the reverse order, the extrinsic information L_a1 should be also reversed before it is applied to SUB/LIM block. This means we need PN2 to read the extrinsic information for the SUB/LIM1 block. The notation L_a1* in Figure 3.22 denotes the reverse order. Similarly, the extrinsic information L_a2 for Decoder 2 is read by a down_counter. Therefore, an up_counter should be used for the SUB/LIM2 block to reverse the order of L_a2.

3.3.4 System Control Logic

As mentioned in Section 3.3.2, we use two groups of RAMs called Group A RAMs and Group B RAMs to store the channel information *sys*, *par1* and *par2*. During the first frame, which is frame 0, three signals sys, par1 and par2 are stored in Group A RAMs. During the second frame, the three signals are now stored in Group B RAMs. In general, Group A RAMs (Group B RAMs) store data of an even (odd) number frame, and data in Group B RAMs (Group A RAMs) are read out for decoding at the same time. The relationship between the frame index number and the groups of RAMs is shown in Figure 3.28.

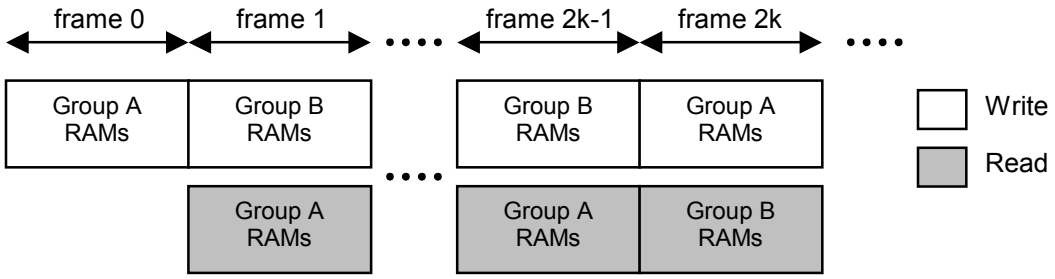


Fig. 3.28: Storage of channel information

Consider the timing diagram of an odd frame shown in Figure 3.28. The forward operation and the backward operation are represented with separated rectangular boxes during a decoding process in the figure. The information in the box shows if the data should be read from or written to RAM1 or RAM2 and the sources of the address generators. A list of address generation blocks used in Figure. 3.29 is given below:

- u_cnt: up_counter
- d_cnt: down_counter
- Dd_cnt: down_counter delayed by one FDCLK
- PN1: Pseudorandom noise generator with polynomial function P1(x) in table 3.5
- PN2: Pseudorandom noise generator with polynomial function P2(x) in table 3.5
- DPN2: PN2 delayed by one FDCLK

Suppose that Decoder 1 is processing data, while Decoder 2 is idle during a certain iteration, but not the first iteration. Decoder 1 reads the extrinsic information of Decoder 2 from RAM1 with the address generated by the PN1 generator during the forward operation. (Refer to Figure 3.22.) During the backward operation, Decoder 1 reads from RAM1 again to compute new extrinsic information, but with the address generated by the PN2 generator to match the reverse order of LLRs calculated by Decoder 1. Decoder 1 writes the extrinsic information into RAM2 with the same PN2 address generator but delayed by one FDCLK cycle, which is denoted as DPN2 address generator.

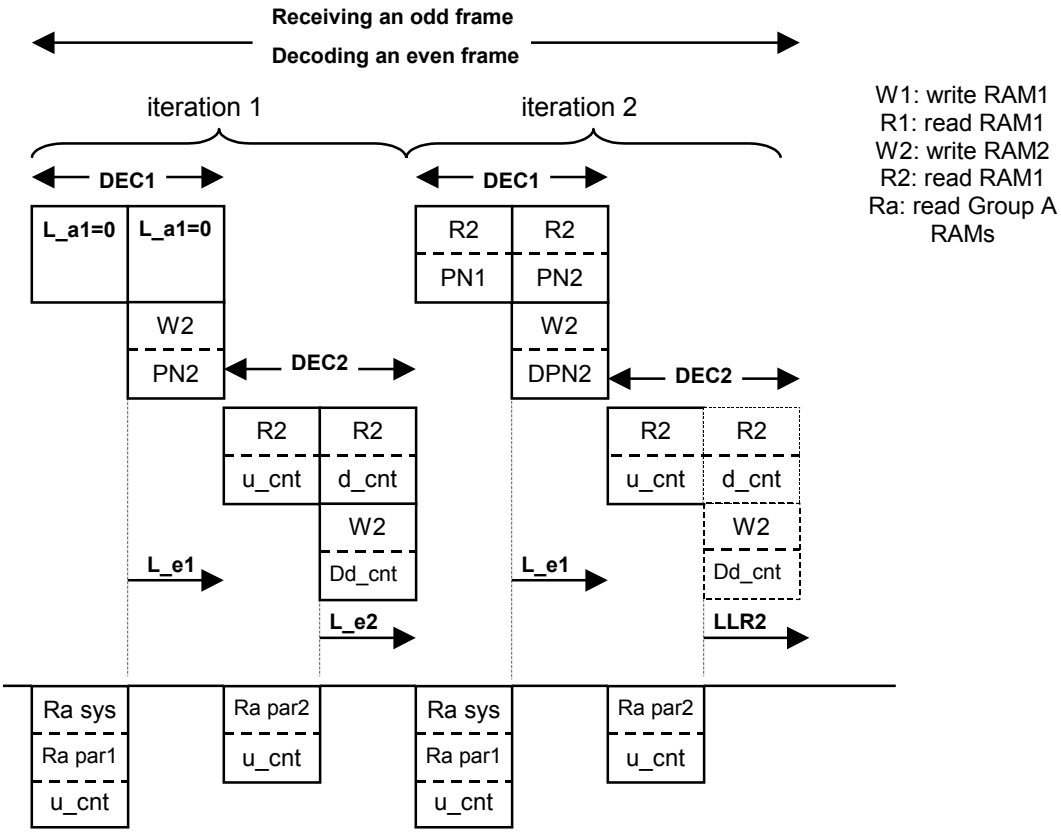


Fig 3.29: Timing diagram of RAM control signals and addresses

Suppose that Decoder 2 is processing data now, while Decoder 1 is idle. Decoder 2 reads RAM2 with the address generated by the up_counter in the forward operation. During the backward operation, Decoder 2 reads RAM2 again (to compute the extrinsic information) but with the address generated by the down counter. The calculated extrinsic information is written into RAM2 with the address generated by the same down counter but delayed by one clock (FDCLK) denoted as Dd_cnt counter. If it is the last iteration, it is necessary to compute only LLRs, but not the extrinsic information, during the backward operation. So write operation of RAM2 is disabled.

3.4 Proposed Low-power Design for Turbo Decoder

Given the background about the implementation of our turbo decoder, we propose low-power design of the turbo decoder through reduction of the switching activities during decoding. We propose several methods for power reduction in this section. Although the log-MAP algorithm reduces the complexity of the computation greatly, a log-MAP decoder is still computationally intensive, and each decoding stage consumes a large amount of power. Hence, if we adjust the number of iterations dynamically according to the channel condition, the power consumption can be reduced. After reaching the necessary number of iterations, both component decoders are shut down. The next method is to shut down an inactive component decoder during the decoding process. As noted earlier, only one of the two decoders is working at a time during decoding while the other is idle. We shut down the idle component decoder to reduce the power dissipation. The last method is to block spurious signals to eliminate unnecessary switches. We explain these methods below.

3.4.1 Implementation of a Variable Number of Iterations

In order to ensure a reliable communication, a communication system should be designed for the worst channel condition. However, the worst channel condition occurs seldom, so that the performance of a communication system exceeds the required one during the non-worst channel condition to result in waste of power dissipation.

The necessary number of iterations for a turbo decoder depends on the channel condition, specifically E_b/N_0, where E_b is the transmitted power of each bit and N_0 is the power of the channel noise. If E_b/N_0 increases, it needs a less number of iterations to achieve a desired BER (bit error rate). The energy dissipated during the decoding process is proportional to the number of iterations. Hence, we propose the number of iterations be determined dynamically according to the channel condition.

Considering the speed of circuit and the improvement of the BER through an additional iteration, we have chosen the maximum number of iterations of our decoder to be four. Therefore, we classify the channel condition into four categories based on E_b/N_0, and the decoder performs a different number of iterations accordingly as shown in Table 3.6. Our simulation result to be presented in Chapter 4 shows that the proposed number of iterations achieves the BER less than $7.8*10^{-5}$ at E_b/N_0 =1.0 dB after 5 iterations and $1.96*10^{-5}$ at E_b/N_0 =1.2 dB after 3 iterations.

Table 3.6 Channel condition classification

Channel condition	E_b/N_0	Number of iterations
Excellent	$E_b/N_0 > 2$dB	1
Good	$2\text{dB} \geq E_b/N_0 \geq 1.3$dB	2
Fair	$1.3\text{dB} > E_b/N_0 \geq 0.9$dB	3
Poor	$E_b/N_0 < 0.9$ dB	4
Very poor	E_b/N_0	5

The necessary number of iterations is given as an input signal of the turbo decoder. After the necessary iterations have been performed, a finite state machine (FSM) generates output signals to shut down both Decoder 1 and Decoder 2. The FSM works under clock FDCLK. The inputs of the FSM include control signals (*en_dec1* and *en_dec2*), a reset signal (res_FSM), and the desired number of iterations (*NUM_iter*) for the current channel condition. The two control signals, *en_dec1* and *en_dec2*, are alternatively active (high), indicating the status of the component decoders. When *en_dec1* (*en_dec2*) is '1', Decoder 1 (Decoder 2) is active.

Consider the state diagram of the FSM given in Figure 3.30. The initial state of the FSM is denoted as S_0, where no decoding process is performed during the time. Once a reset signal res_FSM transits to '0', the desired number of iterations NUM_iter determines the next state to move from the initial state S_0. Suppose that NUM_iter = "001" implying the necessary number of iterations is 2 from Table 3.7. The next state is S_4, and it stays at the state until Decoder 1 finishes its current processing. It transits to S_3

as *en_dec1* becomes '0', indicating that Decoder 2 starts processing, and stays at S_3 until *en_dec1* changes to '1', which means the processing is done. When the FSM reaches state S_2, it has finished the first iteration, and it enters the last iteration. After transition to S_1 state, a signal iter_sign is set to '0', and the signal deactivates the blocks that write the extrinsic information L_e2 into RAM1. (Refer to Figure 3.22.) After Decoder 2 finishes the processing, it goes back to S_0, which indicates the completion of the necessary iterations. The signal res_FSM is an indicator of the beginning of a new frame. It is a combination of the outputs of a 11-bit slow stage counter working under SDCLK and a 4-bit slow state counter working under SCLK. We called them slow counters in contrast to the counters working under fast clocks, FDCLK and FCLK. When the least significant ten bits of the slow stage counter counts to 1 and the slow state counter outputs 0, res_FSM is set to '0', indicating that the decoding process of a new frame starts. The signal stop_iter becomes '1' in state S_0, and it forces en_dec1 and en_dec2 to stay at '0' until the signal res_FSM = '0', indicating a new frame. When it moves to other states than state S_0, stop_iter is '0'. The timing diagram of the state transition is given in Figure 3.31. Since there are 11 states involved, 4 D-FFs working with fast data clock DCLK are used. We use the state coding as listed in Table 3.7. Note that the least three significant bits of the five starting states (S_{10}, S_8, S_6, S_4, S_2) are identical to NUM_iter, indicating the number of iterations for the frame.

It is obvious that the power consumption overhead incurred by the registers and the combinational circuit for the FSM is much less than the power consumed by a fixed number of iterations considering the worst channel condition.

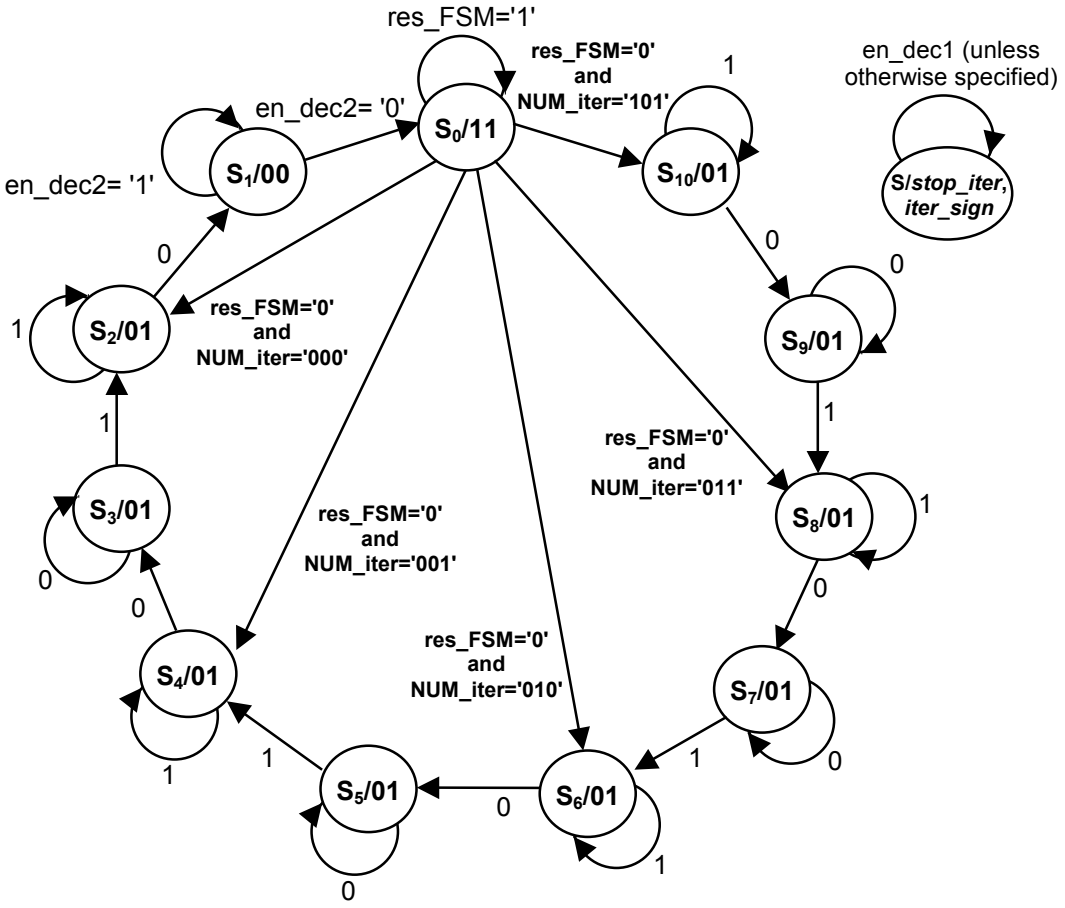


Fig. 3.30 State diagram of the finite state machine to control the number of iterations

Table 3.7 State encoding of the finite state machine

State Number	NUM_iter	state encoding
S_0		1100
S_1		0100
S_2	000 (1)	0000
S_3		1000
S_4	001 (2)	1001
S_5		0001
S_6	010 (3)	0010
S_7		1010
S_8	011 (4)	1011
S_9		1111
S_{10}	101 (5)	1101

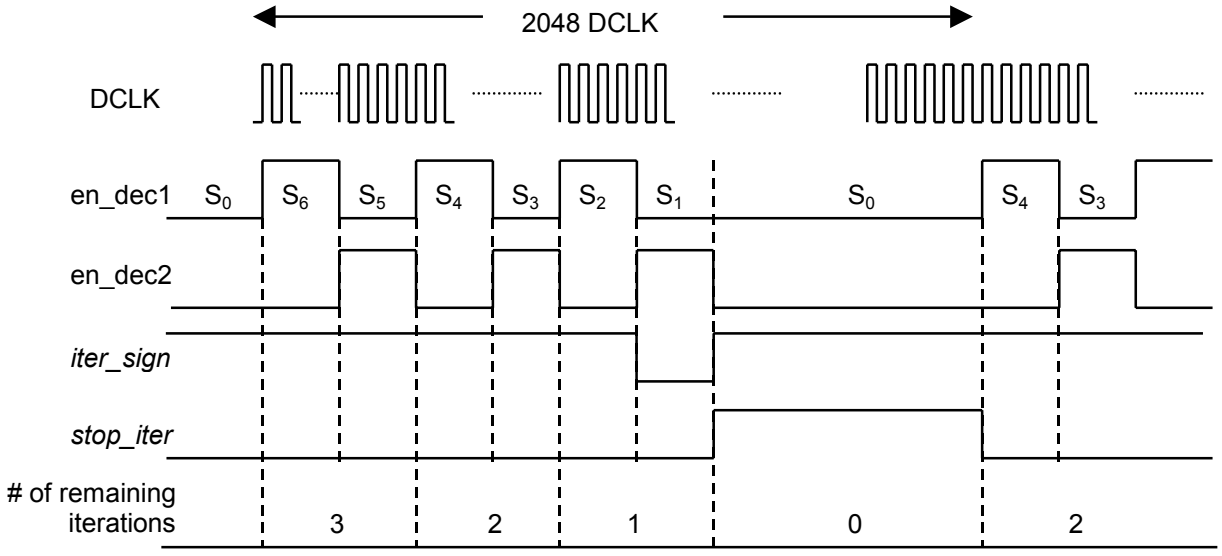


Fig. 3.31: Timing diagram of the state machine

3.4.2 Shutdown of Component Decoders

In our turbo decoder, one of the two component decoders can be shut down alternatively when it is idle during the decoding process and both of them can be shut down after the decoding process is over for the frame. We employed a clock gating scheme and block of input signals to shut down an inactive component decoder. Two enable signals, *en_dec1* and *en_dec2*, indicate the status of two decoders as mentioned in the previous section.

When a component decoder is inactive, the clock FDCLK is blocked using the corresponding control signal, en_dec1 or en_dec2. The AND operation of FDCLK and a control signal to block the clock may create a timing problem as shown in Figure 3.32(a). The control signal en_deci, i = 1 or 2, is generated in synchronous to FDCLK, and the control signal may be delayed by Δ. If we perform the AND operation of the two signals, the period of the first clock is shorter by Δ. It may cause a problem if it is on the critical path. The problem is addressed by the OR operation of FDCLK and the complement of the control signal, and it is shown in Figure 3.32(b). The cost for this design is that the component decoder is enabled at t_2 instead of t_1. However, the latter design is more prudent, as it avoids any potential timing problem and is employed in our design.

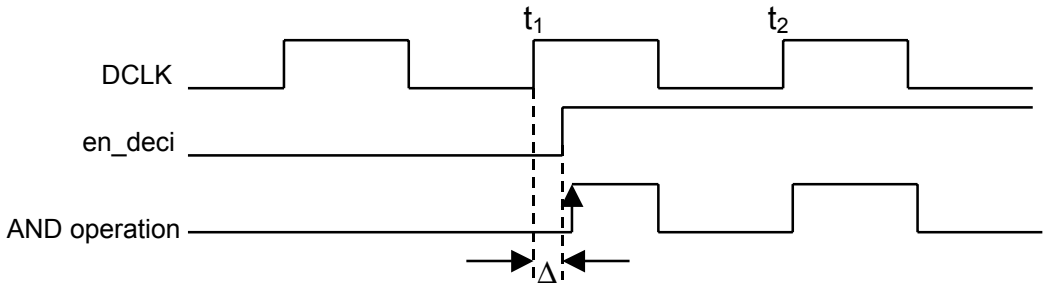


Figure 3.32(a): Block the clock using AND operation

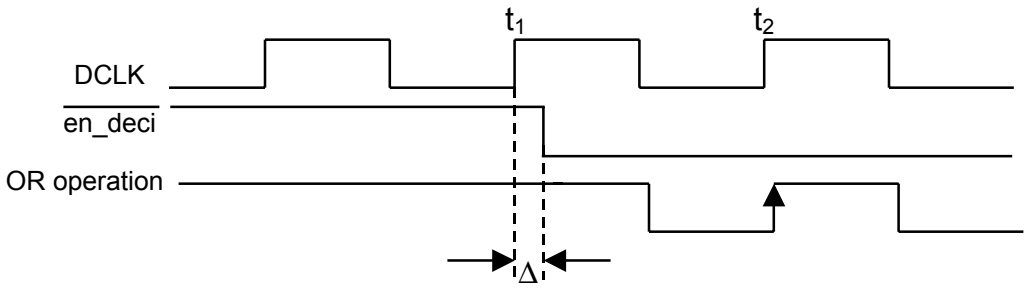
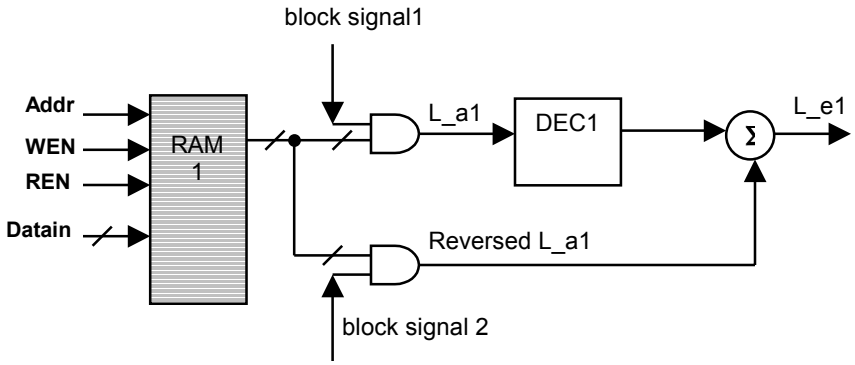


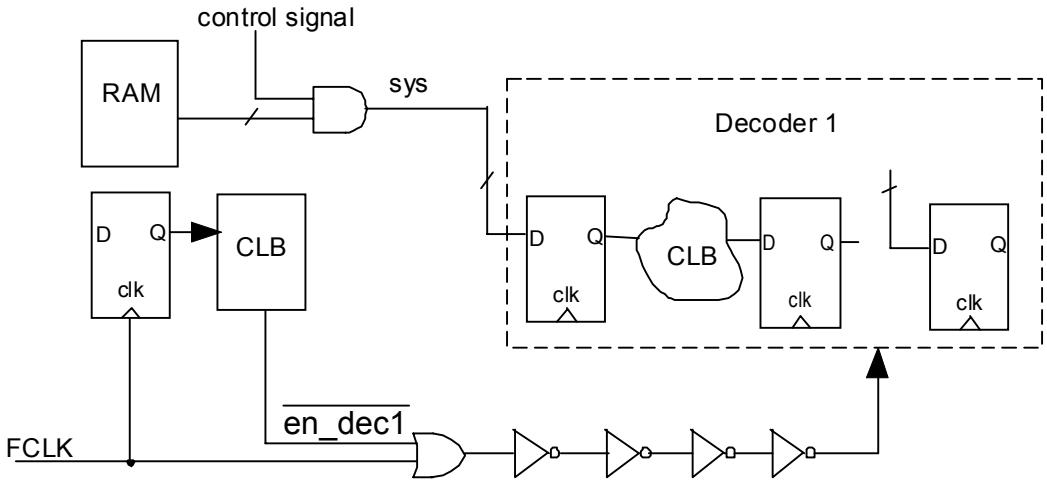
Figure 3.32(b): Block the clock using AND operation

In order to completely shutdown the inactive component decoder, the inputs of the decoders are also blocked by some control signals to prevent the component decoder from receiving spurious inputs when it is in idle. For Decoder 1, two channel inputs, sys and par1, are read from Group A or Group B RAMs, and the priori information, L_a1, is obtained from RAM1. Decoder 1, when active, needs only sys, par1 and L_a1 at the input of the Decoder 1 during the forward operation, while Decoder 2 needs only par2 and L_a2 during the forward operation when active. Other than these periods of time, all the inputs of the decoders should not change to avoid unnecessary signal transitions. IN fact, the output data bus of a RAM is floating when the RAM is disabled, which is not desirable in low power design because it may cause short circuit. Hence, control signals are generated from the controller to block the RAM output data bus from the decoders. AND operations of a control signal and the RAM output data bus are applied to prevent the circuit from never floating.

In the case of L_a1 and L_a2, they are also read out during both the forward and the backward operation to calculate the extrinsic information. Hence, the output data buses are always active. However, the outputs should also be blocked from the component decoders and the SUB/LIM blocks, so that spurious inputs should not be applied to the two blocks. It is shown in Figure 3.33(a). Figure 3.33(b) shows the gated clock and blocked inputs of Decoder 1.



(a) Block diagram of the implementation to shut down Decoder 1



(b) Gated clock and blocked inputs in Decoder 1

Figure 3.33: Shutdown of Decoder 1

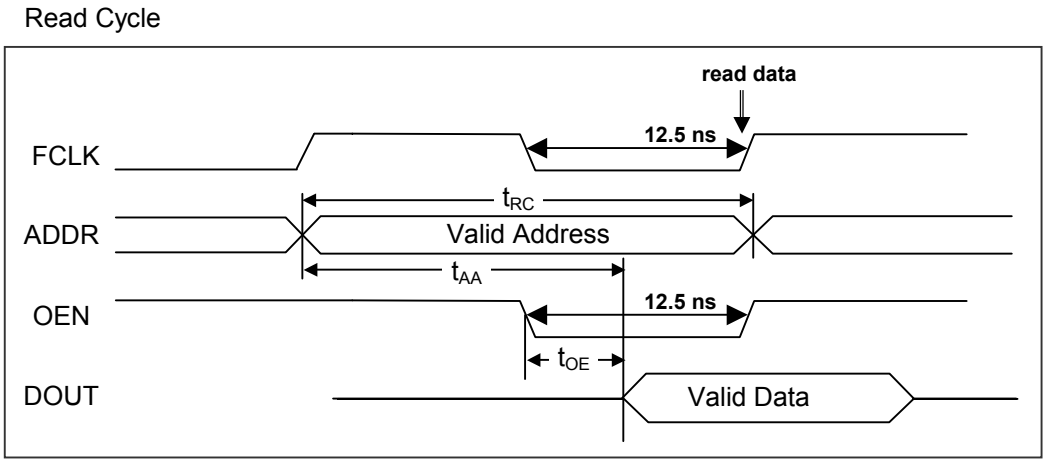
3.5 Control Signals of RAMs and Clock Generation

Two types of asynchronous SRAMs, single port SRAMs and dual port SRAMs, are used in our turbo decoder. Each single port RAM has one read enable signal OEN, one write enable signal WEN, one address bus and one input/output data bus. Single port RAM can only perform either read operation or write operation at a time. A dual port SRAM also has one OEN, one WEN, but it has two address buses and two data buses,

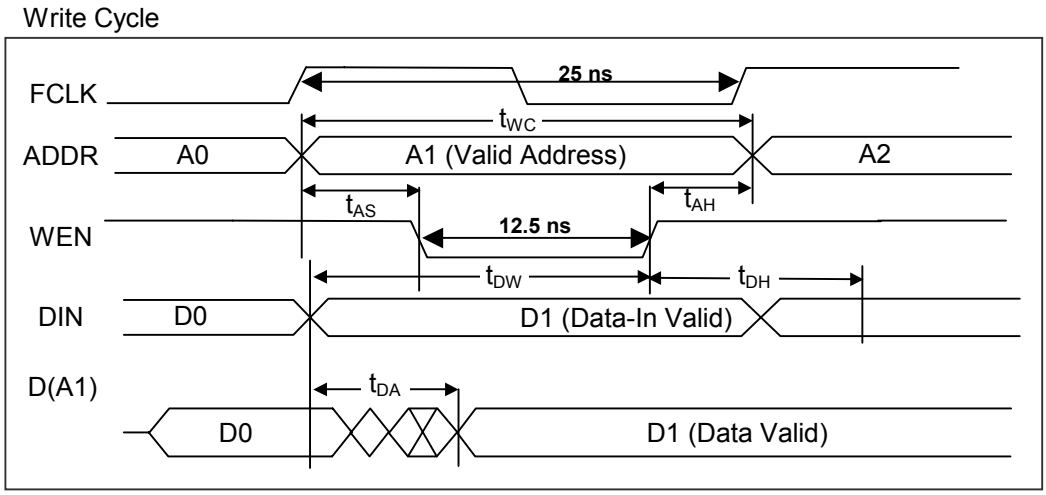
where read and write operation can be performed at the same time using one pair of address and data bus respectively. The enable signals OEN and WEN are active low. We use FCLK of 40 MHz (with period 25 ns) as the fastest read enable signal and the write enable signal is a FCLK delayed by one quarter of the period. The FSM and BSM RAMs need the fastest OEN and WEN, and the read operation and the write operation of these RAMs are illustrated in Figure 3.34.

During the read cycle, an address ADDR is applied at the rising edge of FCLK, and the output enable OEN is active at the falling edge of FCLK. The data is read at the rising edge of the following clock. Our turbo decoder requires the output enable time, t_{OE}, to be less than 12.5 ns, which is easily met for contemporary SRAMs.

During the write cycle, an address ADDR and date DIN are applied at the rising edge of FCLK. Then write enable WEN becomes low after 6.3 ns (a quarter of the period of the clock FCLK) and lasts for half of the period of FCLK. As a result, the data-in setup time t_{DW}, the data-in hold time, t_{DH}, and the data-in access time t_{DA} are sufficiently long to perform a write operation successfully. Hence, the timing requirements for a RAM are met under the system clock of 40 MHz.



(a) Read cycle



(b) Write cycle

Fig. 3.34: Timing Parameters of RAMs [51]

There are two external clocks, FECLK of 80 MHz and SECLK of 4 MHz. The two clocks are asynchronous. SECLK and its derivatives are synchronized to the data rate rather than the decoders.

In the turbo decoder design, except the slow clock signals, SCLK, SDCLK, and the write enable signals for Group A and B RAMs, all the other clocks, read and write enable signals are generated by an external clock FECLK running under 80MHz. This is illustrated in Figure 3.35. Other clocks, FCLK, FCLK2 and FDCLK, are generated by

dividing the FECLK by 2, 4 and 32, respectively, and FCLK1 is the compliment of FCLK2. FCLK is also the OEN signal for FSM and BSM RAMs. FCLK1 is the OEN signal for BM RAMs and FDCLK is used as OEN signal for Group A and B RAMs. A compliment of a write enable signal is a clock signal delayed by one quarter of the clock period. The write enable signals can be obtained by dividing the compliment of FECLK. For example, we compliment FECLK, and divide the complimented signal by 2, then compliment the signal again to generate WEN for BM, FSM and BSM RAMs.

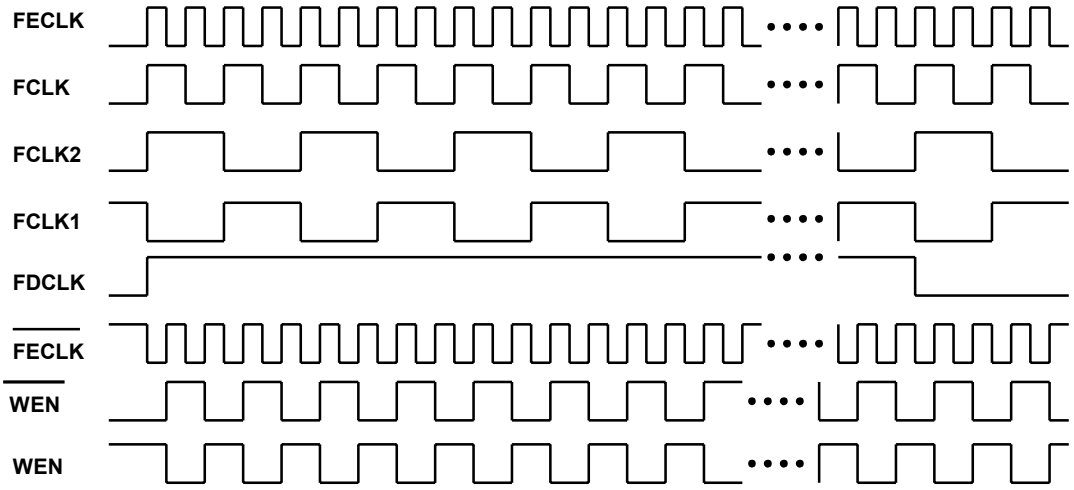


Fig. 3.35: Clock and WEN signals generation based on FECLK

The slow clocks SDCLK and SCLK are produced by the other external clock SECLK of 4 MHz. Similarly, SCLK and SDCLK are obtained by dividing SECLK by 2 and 32, respectively. The write enable signals of Group A and B RAMs are based on the compliment of SECLK, and it is illustrated in Fig. 3.36. Note that there is still one quarter of SCLK delay between the rising edge of SCLK and the falling edge of the write enable signal.

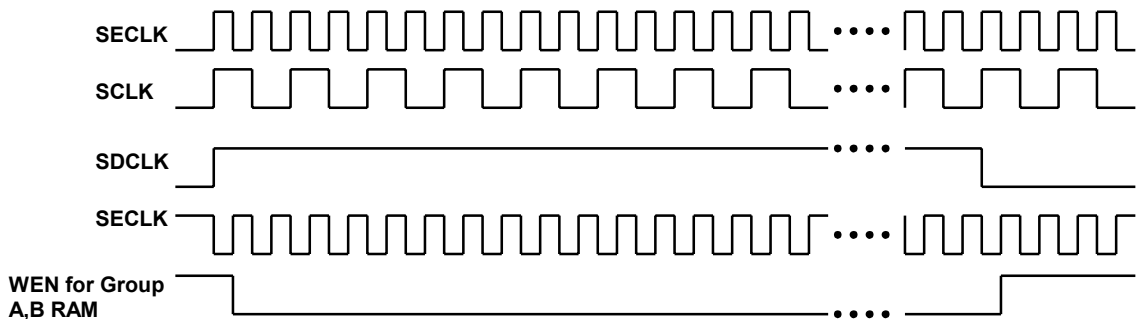


Fig. 3.36: Clock and WEN signals generation based on SECLK

3.6 Initialization

A global reset signal "reset" is active low and resets the turbo decoder at the rising edge of slow data clock SDCLK. This implies that the reset signal should remain low for at least one SCDLK clock cycle.

Two FSM RAMs and two BSM RAMs of the two decoders should be initialized at the beginning of the decoding process. The first 16 elements (which corresponds to stage 0) of each FSM RAM are initialized. 0 is written to address 0 and 255's are written to addresses 1 to 15.

The last 16 elements (which correspond to stage 1023) of the BSM RAM for Decoder 1 should also be initialized to 0 followed by 15 255's. In other words, the location 3FF0 (= 16368) should be 0 and the remaining 15 addresses should be 255. 0's should be written to the last 16 elements of the BSM RAM for Decoder 2.

The initialization of the four RAMs is performed during the first frame of the decoding process.

Chapter 4

Experimental Results

In this chapter, we describe the verification of our turbo decoder first. Then we present the experimental results including area, timing and power dissipation of our turbo decoder. The turbo decoder was described in VHDL at behavioral level. A logic synthesis tool of Synopsys, called Design Compiler, was used to obtain a gate level circuit from the behavioral level description. The technology used in our experiments is TSMC 0.35 μm CMOS with supply voltage of 3.3V.

4.1 Verification of the Design

After the operations of individual blocks were tested, the verification was performed by comparing the outputs of the synthesized turbo decoder and the results from MATLAB simulation. In the hardware implementation of the turbo decoder, limited numbers of bits were used to represent data. In order to verify the design, MATLAB codes were written to reflect the actual implementation. For this purpose, we used integers with specific ranges in MATLAB codes instead of real numbers. We simulated 6000 random inputs for the MATLAB codes and for the synthesized gate level circuits. The two results match with each other. In addition to the verification using MATLAB simulation, we also compared the final decoded output bits obtained from the gate level simulation with the input information bits.

4.2 Performance of the Turbo Decoder

Since the discrepancy between the results obtained from the MATLAB simulation and that from the synthesized gate level simulation is negligible, we measured the performance of the turbo decoder through MATLAB simulation. The simulation follows the procedure listed below:

1. Generate the information bits randomly.
2. Encode the information bits using a turbo encoder with the specified generator matrix.
3. Use BPSK modulation to convert the binary bits, 0 and 1, into antipodal signals. In our simulation, signals of 1 and –1 are transmitted.
4. Introduce noise to simulate channel errors. We assume that the signals are transmitted over an AWGN (Additive White Gaussian Noise) channel. The noise is modeled as a Gaussian random variable with zero mean and variance σ^2. The variance of the noise is obtained as $\sigma^2 = \frac{1}{2 \times r \times E_b / N_0}$, where r is the code rate (Refer to Section 2.1). In our design, r = 1/3. We use a built-in MATLAB function *randn* to generate a sequence of normally distributed random numbers, where *randn* has zero mean and 1 variance. Thus the received signal at the decoder side is:

   $$x' = x + \sigma * randn,$$

   where $x \in \{+1,-1\}$ and x' is the received signal.
5. Simulate the automatic gain control (AGC) and an ADC (analog to digital converter) to scale the received signals and convert them to integers. According to [50], there is an optimal gain to scale the received signal before it is applied to a quantizer. Hence, we used a lookup table to store the near optimal scaling factor.
6. Decode the information and write all the LLR outputs to a file for the purpose of comparison against the gate level simulation.
7. Count the number of erroneous bits by comparing the decoded bit sequence with the original one.
8. Calculate the BER and plot it.

We summarize the specification of our turbo decoder used for the performance measurement.

- Frame length: 1024

- Component encoder: (2,1,5)
- Generator polynomial: $g_0 = 23$ (octal), $g_1 = 35$ (octal)
- Quantization level of the channel information sys, par1 and par2: 7 bit signed data ranging from –64 to 63
- BM: 7 bit unsigned data ranging from 0 to 63
- FSM, BSM: 8 bit unsigned data ranging from 0 to 255
- LLR: 10 bit signed data ranging from –512 to 511
- *a priori* information L_a: 8 bit signed data ranging from –128 to 127

The BER of the decoder was measured at ten different E_b/N_0 points such that E_b/N_0 (dB) = 0, 0.2, 0.5, 0.6, 0.8, 1.0,1.2,1.5, 1.7, 2. We experimented with two sets of scaling factors listed in Table 4.1. Note that the scaling factor B is double the scaling factor A. We obtained the scaling factors based on [10]. The received signal x' is multiplied by the scaling factor of the corresponding E_b/N_0 before quantization into a 7-bit signed number. We assume that E_b/N_0 is available to the turbo decoder.

Table 4.1 Two sets of scaling factor used before quantization

E_b/N_0(dB)	0	0.2	0.4	0.5	0.6	0.8	1.0	1.2	1.5	1.7	2.0
Scaling factor A	31	30	29	29	28	28	28	27	26	25	24
Scaling factor B	62	60	NA	58	NA	56	56	54	52	50	48

Stopping criteria for the simulation were determined as follows. An error frame means a frame with at least one bit error in it.

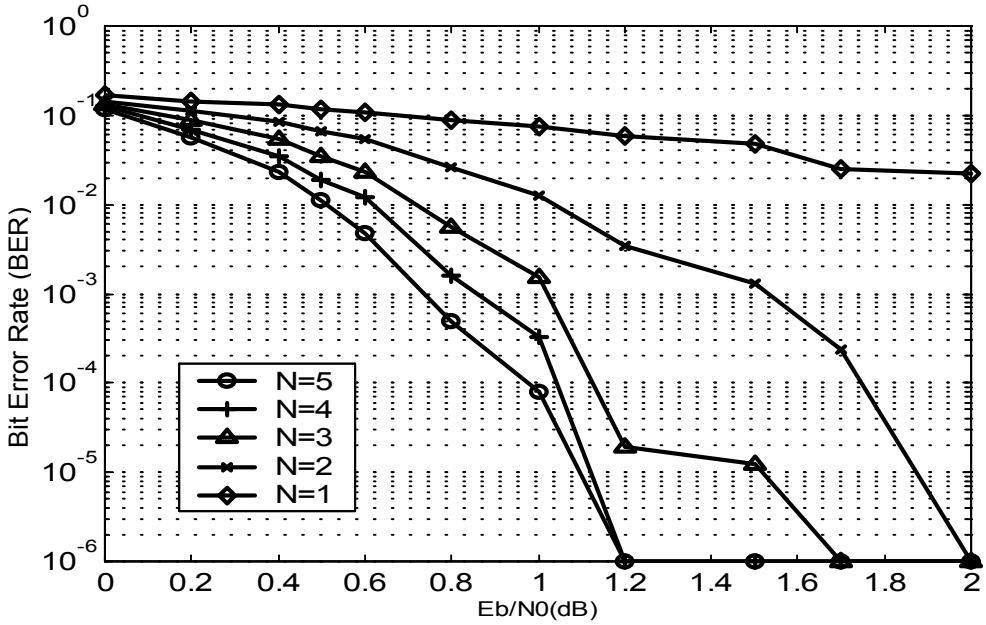
a) $E_b/N_0 \geq 1$ dB

Error frames appear infrequently. Therefore, the simulation stops if the number of error frames under five iterations is 5 or the total number of frame processed so far is 150, whichever comes first. If a BER is 0, it is plotted as 10^{-6} in the graph.

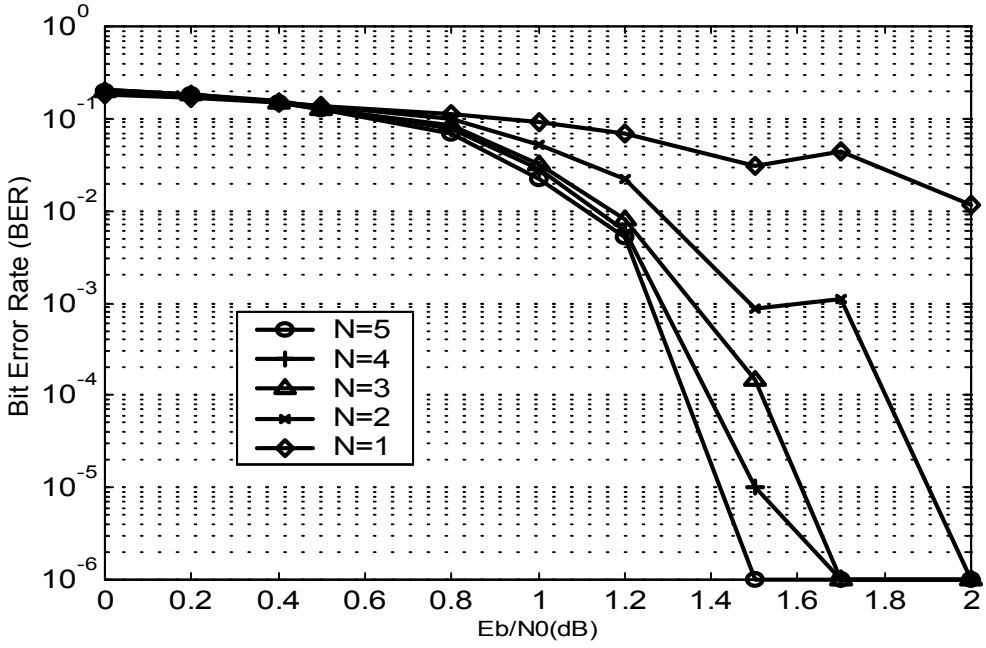
b) $E_b/N_0 < 1$ dB

The simulation stops if the number of error frames under five iterations is 50.

The simulation results for the two different sets of scaling factors are plotted in Figure 4.1. Figure 4.1(a) is the performance of our turbo decoder with the scaling factor A under five different numbers of iteration. Figure 4.1(b) is the performance with the scaling factor B. In the two figures, N is the number of iterations.



(a) Performance under the scaling factor A

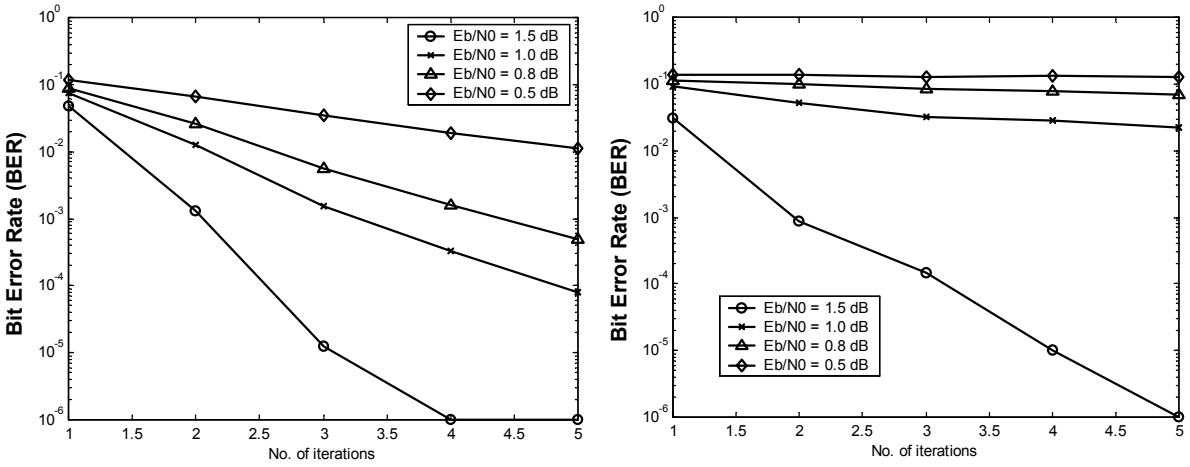


(b) Performance under the scaling factor B

Fig. 4.1: Performance of a rate 1/3, 16 states, frame length 1023 turbo decoder

As we compare the two graphs shown in Figure 4.1, the scaling factor is sensitive to the BER performance against E_b/N_0. After 5 iterations, the BER of the scaling factor A is more than 65 times lower than that of the scaling factor B under E_b/N_0 = 0.8 dB. The difference in performance at E_b/N_0 = 1.0 dB and 1.2 dB is even larger. However, the difference becomes smaller for larger E_b/N_0. We can also notice that under low E_b/N_0, such as E_b/N_0 = 0.5 dB, the BER decreases with the increase of the number of iterations under the scaling factor A. However, the trend does not hold for the scaling factor B. It is explained as if the scaling factor is too high, the data are more opt to overflow during the calculation. According to [50], a severe overflow in the decoding process significantly degrades the performance of a turbo decoder with the fixed point implementation. The research in [50] also shows that in order to achieve higher performance under E_b/N_0 less than 0.5 dB, the number of bits of the internal data needs to be at least 14. Note that the number of bits is 10 in the design. Overall, our turbo decoder achieves reasonably good performance with the scaling factor A.

Figure 4.2 shows BER versus the number of iterations for the two sets of scaling factors. A general trend is that BERs decrease rapidly for each additional iteration with the scaling factor A, but the BERs decrease rather slowly or do not decrease with the scaling factor B except for high E_b/N_0 of 1.5 dB. This means that a proper scaling factor is important for low E_b/N_0 signals.



(a) using scaling factor A (b) using scaling factor B

Fig. 4.2: rate 1/3, 16 states, frame length 1023 turbo decoder BER performance versus No. of iterations under certain Eb/N0 with defferent scaling factor

The performance graph with the scaling factor A shows that a variable number of iterations can be employed to achieve a certain level of BER. For example, CDMA standard IS-95 requires BER = 10^{-3} for voice communication. The number of iterations to achieve the expected BER is summarized in Table 4.2.

Table 4.2 No. of iterations to achieve BER = 10^{-3} under variance E_b/N_0

E_b/N_0	Necessary No. of iterations
[0.6 dB, 0.75 dB]	5
(0.75 dB, 0.9 dB]	4
(0.9 dB, 1.3dB]	3
≥1.3 dB	2
very high E_b/N_0	1

Finally, the performance of our decoder can be further improved through employment of larger number of bits for internal data [50] and large size/sophisticated interleavers at the cost of more complex hardware.

4.3 Area and Delay of the Turbo Decoder

We employ the bottom-up approach to synthesize the decoder. Sub-modules were synthesized individually and integrated at the top level. RAMs were modeled in behavioral level and were not synthesized. We have put a load capacitance equivalent to ten inverter gates for the delay measurement. The equivalent NAND2 gate counts and critical path delays of sub-modules are listed in Table 4.2. The module names in the table are given below:

- BM1: branch metric calculation block in Decoder 1
- BM2: branch metric calculation block in Decoder 2
- BSMC: backward state metric calculation block in Decoder 1 or Decoder 2
- CTRL: control logic in Decoder 1 or Decoder 2
- FSMC: forward state metric calculation block in Decoder 1 or Decoder 2
- SYS_CTRL: turbo decoder system control logic block
- LLR: log-likelihood ratio calculation block in Decoder 1 or Decoder 2
- MINSM: minimum state metric search block in Decoder 1 or Decoder 2
- SUB-LIM: subtraction circuit and limit circuit in Decoder 1 or Decoder 2 (refer to Figure 3.21.)
- TOPCOUNT: state counter and stage counter in Decoder 1 or Decoder 2, working under FCLK and FDCLK, respectively.
- Slow_TOPCOUNT: state counter and stage counter at the turbo decoder system level, working under SCLK and SDCLK, respectively.

The area of the component decoders, Decoder 1 and Decoder 2, is 5972 and 5778 NAND2 gates (excluding the RAMs), respectively. The two component decoders are moderate in size. The reason for smaller gate count for Decoder 2 over Decoder 1 is that Decoder 2 does not contain a circuit to sum up the *a priori* information and the *sys* input. We noticed that the critical path delay of Decoder 1 and Decoder 2 are identical, which is the critical path of the top level turbo decoder. The output of the FSMC module is applied directly (without going through registers) to the MINSM module. Close examination reveals that the critical delay is the sum of the delay of the FSMC module

and part of the MINSM module. The maximum allowable system speed is 57MHz based on the critical delay. Currently, the system clock FCLK is set to 40 MHz for our decoder.

Table 4.3 Area and critical path delay in each sub-module and top-level circuit

Module	Equivalent NAND2 gate counts	Critical path delay (ns)
BM1	729	1.54
BM2	542	1.49
BSMC*	750.67	15.71
CTRL*	2026	6.79
FSMC*	638	15.77
SYS_CTRL	1309	4.62
LLR**	528	16.06
MINSM*	187.32	3.62
SUB_LIM*	189.68	5.62
TOPCOUNT	224.66	1.82
Slow_TOPCOUNT	206.33	1.82
Decoder 1	5972	17.34
Decoder 2	5778	17.34
Turbo Decoder (total)	13877	17.34

Note: Two units are necessary for a block marked with "*" and four units for the block marked with "**".

RAMs are modeled in behavioral level. As mentioned in Section 3.5, the timing requirements of the RAMs are easily met for contemporary SRAMs.

4.4 Power Dissipation Measurement

Power dissipation can be measured at the gate level or at the switch level (extracted from the layout). Although gate level measurement is less accurate than at the switch level, it is faster and it provides the designer with enough information about the

power dissipation. Thus the design can be adjusted at an earlier stage to save the design cost. As our goal is to obtain a power efficient turbo decoder, absolute values are not critical. Instead, the relative accuracy suffices our needs. Hence, we opted to measure the power at the gate level.

The flow of power measurement at the gate level is shown in Figure 4.4. Design compiler provides an option to measure the power consumption at the gate level. The power dissipation is the sum of static and dynamic power dissipation. The design compiler refers to the characterization of the technology library to estimate the static power dissipation. For the computation of dynamic power dissipation, the tool follows the formula $P=\alpha C_L V^2 f$, where α is the switching activity, C_L is the parasitic capacitance, V is the supply voltage, and f is the clock frequency. The capacitance for the basic cells and the supply voltage are defined in the technology library. The major task of the power estimation at the gate level is to measure the switching activity α. There are two major approaches: probabilistic estimation and simulation based estimation. The probabilistic approach is faster, but it turns out that the accuracy is too poor to be accepted. Hence, we adopted the simulation-based approach, where approximately 5,000 random input patterns (generated from MATLAB) were applied. During the simulation of the gate level VHDL file, the switching activity of each node was captured and annotated, and the power dissipation was estimated based on the power dissipation formula.

The flow chart in Figure 4.3 shows the procedure to measure power in Synopsys environment. In the figure, the Designer Compiler of Synopsys reads an RTL design and compiles it to generate a technology-dependant gate level design. After changing the name rules for VHDL in the gate level design, the design is sent to the VSS simulator. The VSS simulator performs logic simulation according to the test bench file for the design. The test bench includes more than 5,000 random patterns for our design. The gate level simulation generates a toggle file containing all the switching activity information of all the nodes. Such kind of toggle file can be converted to a script file, and can be read by Design Compiler to annotate the switching activity. Power compiler

performs a power analysis in the design using the annotated switching activity and generates reasonably accurate power dissipation results.

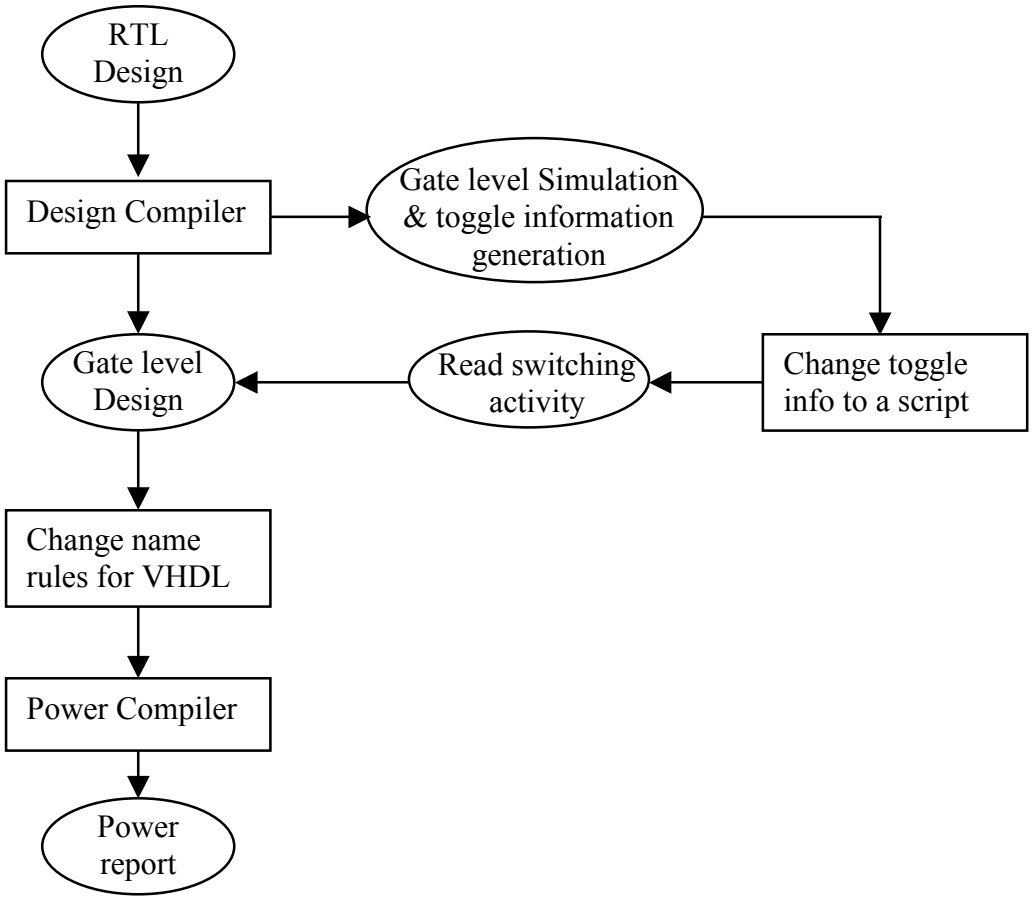


Fig. 4.3: Methodology for Gate-level power estimation. [Synopsys manual]

4.5 Power Dissipation in Turbo Decoders with Low-power Design

Three low power design techniques are proposed in Chapter 3. Since the blocking of floating inputs reduces the static power dissipation, the power measurement method described above cannot be used. Hence, we consider two low-power design techniques, flexible number of iterations and shut down of component decoders, in our experiments.

We measured power dissipation of three different modes of operation for the turbo decoder during a typical (not the first) frame. The result is shown in Figure 4.4.

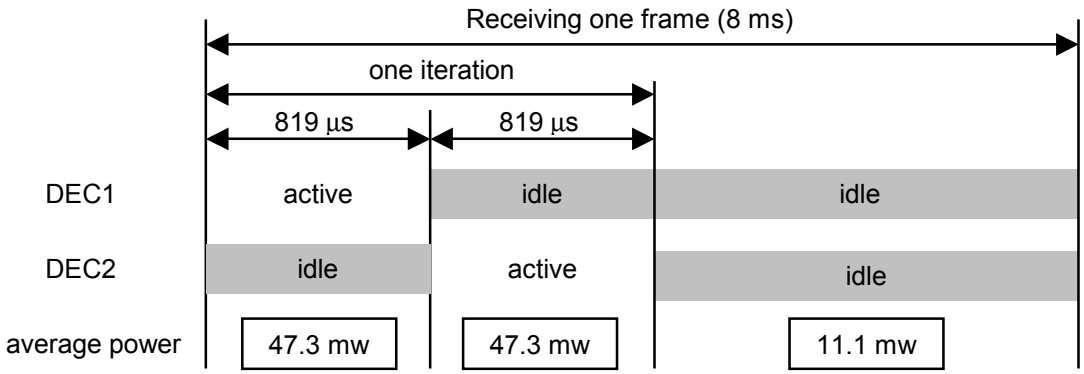


Fig. 4.4: Power consumption of three modes in turbo decoder

From the figure, the power dissipation of the two component decoders are similar. Each component decoder dissipate about (47.3 – 11.1) = 36.2 mw. When both component decoders are shut down (idle), the power dissipation is reduce to 11.1 mw. Hence, a substantial power is saved during the idle state.

The total amount of energy consumed for each iteration is obtained as:

$$47.3 \text{ mw} * 819 \text{ μs} + 47.3 \text{ mw} * 819 \text{ μs} = 77.5 \text{ μJ} \quad (4.3)$$

The amount of energy consumed within one time frame for various numbers of iterations is listed in Table 4.4. A portable device operated by a battery draws the energy from the battery. Hence, the variable number of iterations proposed in the thesis prolongs the battery life as manifested in Table 4.4.

Table 4.4 Energy consumed for various number of iterations in one time frame

Iteration	Energy consumed
1	148 μJ
2	207 μJ
3	266 μJ
4	326 μJ
5	385 μJ

Clocks gating is employed to shut down inactive component decoders. Data inputs of inactive component decoders are also blocked in our design. Therefore, there is no dynamic power dissipation for inactive component decoders. It means that the power saving through shutting down an inactive component decoder is 36.2 mw for both Decoder 1 and Decoder 2.

Finally, the outputs of RAMs are floating (high impedance) except during read or write operation. Floating inputs may incur short circuit current, hence, it should be avoided to reduce static power dissipation. We block floating inputs as shown in Figure 3. 33 when a component decoder is inactive.

In summary, a variable number of iterations and clock gating significantly reduce the power dissipation of our turbo decoder. Blocking of floating inputs of component decoders would further reduce the static power reduction of component decoders.

Chapter 5

Conclusion

A class of novel error correction codes, which is known as turbo codes, generates tremendous interest in channel coding of digital communication systems recently due to its high error correcting capability. Parallel concatenated encoding and iterative decoding with interleaving are two key design innovations of turbo codes. We investigated a low power implementation of a turbo decoder in this thesis.

A turbo decoder consists of five major blocks, an input interface block, two component decoder blocks, interleaving/de-interleaving blocks, and control logic blocks. The input interface block stores incoming data. Two component decoders are connected in a pipelined manner and take turn to estimate the maximum *a posteriori* probability of the information bits iteratively. Each decoder improves the estimation based on the *a posteriori* information from the previous decoder. Before the output of one decoder is applied to the other one, the output sequence is interleaved or de-interleaved to match the order of the input sequence. We used a pseudorandom sequence generator and RAMs to implement an interleaver and a de-interleaver. System controllers are responsible for timing and reset signals.

The log-MAP algorithm is adopted to implement component decoders. It is equivalent to the original MAP algorithm that is optimal for estimating the information bits. The complexity of the hardware implementation of the log-MAP algorithm is significantly reduced through computing in the logarithm domain. A log-MAP decoder consists of six parts: a branch metric (BM) calculation block, a forward state metric (FSM) calculation block, a backward state metric (BSM) calculation block, log-likelihood ratio (LLR) calculation block, a control logic and several static RAMs. An operation called "E-operation", which is similar to Add-Compare-Select operation in a Viterbi decoder, is involved during the calculation of FSMs, BSMs and LLRs. One decoding process is partitioned into two steps: first, we calculate BMs and FSMs

simultaneously in the forward direction. All BMs and FSMs are stored into SRAMs. Second, BSMs and LLR are computed simultaneously in the backward direction.

Our research in the low-power design of turbo decoders to is to reduce the dynamic power dissipation in the standard cell design environment. We considered two methods, variable number of iterations and clock gating. The number of iterations is decided dynamically according to the channel condition. The clock-gating method is applied to the two component decoders. When one of the decoders is working, the other is idle. The clock of the idle decoder is gated to save power. We also block inputs of the idle decoder to prevent propagation of spurious inputs.

The design flow that we followed is similar to the one used in industry. The behavior of a turbo decoder was described in VHDL and then the description was modified to embed the low-power techniques. Then the design is synthesized to generate a gate level circuit. The gate-level circuit is simulated again to verify the functionality. Later we can use place and routing tool from Cadence to generate the layout of the turbo decoder.

We wrote MATLAB codes that match the hardware implementation of the turbo decoder to obtain the reference data to verify the correctness of the synthesized circuit. We verified that the results from MATLAB simulation and synthesized circuit simulation match well. We obtained the BER performance of our turbo decoder through MATLAB simulations. After five iterations, the BER of our turbo decoder reaches $7.8*10^{-5}$. Although direct comparison with other turbo decoders is difficult, we believe that the performance of our turbo decoder is reasonable good.

We listed the delay and area of individual blocks of our turbo decoder. The critical delay in our turbo decoder is 17.34 ns, and the total number of NAND2 gates is 13877. The system clock of the circuit is set to 40 MHz and the maximum clock frequency is 57 MHz. The dynamic power dissipation of the circuit was measured based on the annotated switching activities from the gate level simulation. The dynamic power

dissipation of RAMs is not included due to Unavailability of a RAM power model. The average power dissipation during one decoding processing is 47.3 mw. The system consumes 11.1 mw during the shutdown mode. The energy consumed in one time frame is 148 μJ, 207 μJ, 266 μJ, 326 μJ and 385 μJ for one to five iterations, respectively.

In summary, we investigated a low-power design of a turbo decoder. The variable number of iterations and clock-gating of idle component decoders were proposed to reduce power dissipation. The experimental results show that methods save the power significantly.

Bibliography

[1] C. Berrou, A. Glavieux, and P. Thitimasjshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes," in *Proc., IEEE Int. Conf on Commun.,* (Geneva, Switzerland), pp.1064-1070, May 1993.

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

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

[4] William J. Ebel, "Turbo-Codes: Algorithms and Implementation," contract report submitted to Texas Instruments.

[5] J. Hagenauer, P. Robertson, and L. Papke, "Iterative (turbo) decoding of systematic convolutional codes with the MAP and SOVA algorithms," in *Proc., ITG Conf,* pp.1-9, Sept., 1994.

[6] L. Papke, P. Robertson, and E. Villebrun, "Improved decoding with the SOVA in a parallel concatenated (turbo-code) scheme," in *Proc., IEEE Int. Conf on Commun.,* pp.102-106,1996.

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

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

[9] L. Bomer, F. Burkert, J. Eichinger, R. Halfmann, W. Liegl, M. Werner, " A CDMA radio link with 'turbo-decoding': concept and performance evaluation," in *Sixth IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, vol.2, pp.788-793, 1995.

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

[11] Matther C. Valenti, "An Introduction to Turbo Codes," class project paper.

[12] S. Hong, W.E. Stark, "VLSI circuit complexity and decoding performance analysis for low-power RSC turbo-code and iterative block decoders design," in *Proc., IEEE Military Commun. Conf.*, vol. 3, pp.708-712, 1998.

[13] S. S. Pietrobon, "Efficient Implementation of Continuous MAP Decoders and a Synchronisation Technique for Turbo Decoders," in *Int. Symp. on Information Theory and its Applications*, pp.586-589, Sept. 1996.

[14] G. Masera, G. Piccinini, M. R. Roch, M. Zamboni, "VLSI Architectures for Turbo Codes," *in IEEE Trans. On VLSI systems*, vol. 7, No.3, Sept. 1999.

[15] S. S. Pietrobon, "Implementation and performance of a serial MAP decoder for use in an iterative turbo decoder," in *Proc., IEEE Int. Symp. on Inform. Theory,* pp. 471, 1995.

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

[17] K. Seki; S. Kubota; M. Mizoguchi and S. Kato, “Very low power consumption Viterbi decoder LSIC employing the SST (scarce state transition) scheme for multimedia mobile communications,” *Electronics-Letters, IEE*, Vol.30, no.8, p.637-639, 14 April 1994.

[18] I. Kang, A. N. Willson Jr, " A low-power state-sequential Viterbi decoder for CDMA digital cellular applications," *Conference-Paper, ISCAS 96. IEEE, New York, NY, USA*, vol.4, pp.272-275, 1996.

[19] L. Lang, C. Y. Tsui, R. S. Cheng, " Low power soft output Viterbi decoder scheme for turbo code decoding," *Conference-Paper, ISCAS 97. IEEE, New York, NY, USA*, vol.2, pp.1369-1372, 1997.

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

[21] L. R. Bahl, J. Cocke, F. Jelinek, J, Rajiv, " Optimal decoding of linear codes for maximum symbol error rate," *IEEE Trans. Info. Theory*, March 1974.

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

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

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

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

[26] L. Lin and R. S. Cheng, "Improvements in SOVA-based decoding for turbo codes," in *Proc., IEEE Int. Conf. on Commun.,* 1997.

[27] J. Hagenauer, P. Hoeher, " A Viterbi Algorithm with Soft-Decision Outputs and its Applications," IEEE GLOBECOM, vol.3, pp1680-1686, 1989.

[28] O. J. Joeressen, M. Vaupel, H. Meyr, " Soft-Output Viterbi Decoding: VLSI Implementation Issues," in *Proc., IEEE 43rd Veh. Tech, Conf.,* pp.941-944, 1993.

[29] J. Hagenauer, " Source-Controlled Channel Decoding," in *IEEE Tran. On Communications*, vol.43, No. 9, Sept. 1995.

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

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

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

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

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

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

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

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

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

[39] T. Lang, E. Musoll, J. Cortadella, " Individual Flip-Flops with Gated Clocks for Low Power Datapaths," in *Proc. IEEE Tran. On Circuits and Systems – II: Analog and Digital Signal Processing,* vol. 44, No. 6, pp.507-515, Jun. 1997.

[40] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, R. W. Brodersen, "Optimizing Power Using Transformations," in *IEEE Tran. On Computer-aided design of intergrated circuits and systems*, vol. 14, No. 1, pp. 12-31, 1995.

[41] E. K. Hall, S. G. Wilson, " Stream-Oriented Turbo Codes," in *IEEE Veh. Tech. Conf.* vol.3, pp.2542-2546, 1998.

[42] R. Subramanian, M. Barberis, F. Gerbig, B, Ghosh, " Design and Implementation of All-Digital Receivers for Mobile Communications," in *IEEE Veh. Tech. Conf.,* vol. 2, pp.1043-1047, 1996.

[43] S. Benedetto and G. Montorsi, "Design of parallel concatenated convolutional codes," *IEEE Trans. Commun.,* vol.44, pp.591-600, May 1996.

[44] C. Berrou, "Some clinical aspects of turbo codes," in *Proc., Int. Symp. on Turbo Codes and Related Topics,* (Brest, France), pp.26-31, Sept.1997.

[45] F. Burkert and J. llagenauer, "A serial concatenated coding scheme with iterative 'turbo' and feedback decoding," in *Proc., Int. Symp. on Turbo Codes and Related Topics,* (Brest, France), pp.227-230, Sept.1997.

[46] G. K. Yeap, *Practical Low Power Digital VLSI Design*, Kluwer Academic Publishers, 1998.

[47] S. B. Wicker, *Error Control Systems for Digital Communication and Storage*, Prentice Hall, 1995.

[48] C. Heegard, S. B. Wicker, *Turbo Coding*, Kluwer Academic Publishers, 1999.

[49] S. Dolinar, D. Divsalar, "Weight distributions for turbo codes using random and nonrandom permutations," in *TDA Progress Report*, pp.42-121, JPL, Aug. 1995

[50] Y. Wu, "Implementation of Parallel and Serial Concatenated Convolutional Codes," Ph. D thesis, ECPE Dept., Virginia Tech, 2000.

[51] M. Jagasivamani, "Development of a Low Power SRAM Compiler," MS thesis, ECPE Dept., Virginia Tech, 2000.

[52] S. Ranpara, "On a Viterbi decoder design for low power dissipation," MS thesis, ECPE Dept., Virginia Tech, 2000.

Vita

Jia Fei was born in Shanghai, China. In July 1993, she was admitted to Electronic Engineering Department of Fudan University in Shanghai, China. She finished her B.S. requirement and became a graduate student in Shanghai Institute of technical physics, China Academic Institute in September 1997, being excused from the admission tests.

She joined the Electrical and Computer Engineering at Virginia Polytechnic Institute and State University in August 1998, where she works with Dr. Dong. S. Ha. She got her MS degree in July 2000. After graduation, she will begin the employment with Qualcomm, San Diego as an IC design engineer.