High-Speed VLSI Architectures for Turbo Decoders

Zhongfeng Wang[1] and Xinming Huang[2]
[1]*Broadcom Corporation, 5300 California Avenue, Irvine, CA 92617, USA,*
[2]*Department of ECE, Worcester Polytechnic Institute, Worcester, MA 01609, USA.*

Turbo code, being one of the most attractive near-Shannon limit error correction codes, has attracted tremendous attention in both academia and industry since its invention in early 1990's. In this chapter, we will discuss high-speed VLSI architectures for Turbo decoders. First of all, we will explore joint algorithmic and architectural level optimization techniques to break the high speed bottleneck in recursive computation of state metrics for soft-input soft-output decoders. Then we will present area-efficient parallel decoding schemes and associated architectures that aim to linearly increase the overall decoding throughput with sub-linearly increased hardware overhead.

Keywords: Turbo code, MAP algorithm, parallel decoding, high speed, VLSI.

1. Introduction

Error correction codes are an essential component in digital communication and data storage systems to ensure robust operation of digital applications, wherein, Turbo code, invented by Berrou (1993), is among the two most attractive near-optimal (i.e., near-Shannon limit) error correction codes. As a matter of fact, Turbo codes have been considered in several new industrial standards, such as 3rd and post-3rd generation cellular wireless systems (3GPP, 3GPP2, and 3GPP LTE), Wireless LAN (802.11a), WiMAX (broadband wireless, IEEE 802.16e) and European DAB and DVB (digital audio broadcasting and digital video broadcasting) systems.

One key feature associated with Turbo code is the iterative decoding process, which enables Turbo code to achieve outstanding performance with moderate complexity. However, the iterative process directly leads to low throughput and long decoding latency. To obtain a high decoding throughput, a large amount of computation units have to be instantiated for each decoder, and this results in a large chip area and high power consumption. In contrast, the growing market of wireless and portable computing devices as well as the increasing desire to reduce packaging costs have directed industry to focus on compact low-power circuit implementations. This tug-of-war highlights the challenge and calls for innovations on Very Large Scale Integration (VLSI) design of high-data rate Turbo decoders that are both area and power efficient.

For general ASIC design, there are two typical ways to increase the system throughput: 1) raise the clock speed, and 2) increase the parallelism. In this chapter, we will tackle the high speed Turbo decoder design in these two aspects. Turbo code decoders can be based on either *maximum-a-posterior* probability (MAP) algorithm proposed in Bahl (1974) (or any variants of approximation) or soft-output Viterbi algorithm (SOVA) proposed in Hagenauer (1989) (or any modified version). However, either algorithm involves recursive computation of state metrics, which forms the bottleneck in high speed integrated circuit design since conventional pipelining techniques cannot be simply applied for raising the effective clock speed. Look-ahead pipelining in Parhi (1999) may be applicable. But the introduced hardware overhead can be intolerable. On the other hand, parallel processing can be effective in increasing the system throughput. Unfortunately, direct application of this technique will cause hardware and power consumption to increase linearly, which is against the requirement of modern portable computing devices.

In this chapter, we will focus on MAP-based Turbo decoder design since MAP-based Turbo decoder significantly outperforms SOVA-based Turbo decoders in terms of Bit-Error-Rate (BER). In addition, MAP decoders are more challenging than SOVA decoders in high speed design (Wang 2007). Interested readers are referred to Yeo (2003) and Wang (2003c) for high data-rate SOVA or Turbo/SOVA decoder design. The rest of the chapter is organized as follows. In Section 2, we give background information about Turbo codes and discuss simple serial decoder structure. In Section 3, we will address high speed recursion architectures for MAP decoders. Both Radix-2 and Radix-4 recursion architectures are investigated. In Section 4, we present area-efficient parallel processing schemes and associated parallel decoding architectures. We conclude the chapter in Section 5.

2. Background of Turbo Codes

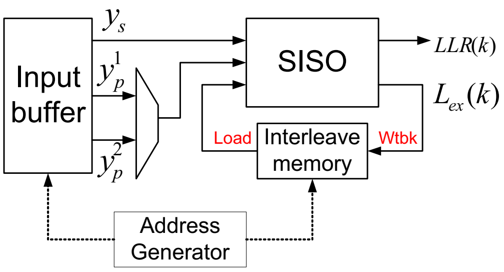


Fig. 1. A serial Turbo decoder architecture.

A typical Turbo encoder consists of two recursive systematic convolutional (RSC) encoders and an interleaver between them (Wang 1999). The source data are encoded by the first RSC encoder in sequential order while its interleaved sequence is encoded by the second RSC encoder. The original source bits and parity bits generated by two RSC encoders are sent out in a time-multiplexed way. Interested reader are referred to Berrou (1993) for details. Turbo code usually works with large block sizes for the reason that the larger the block size, the better the performance in general. In order to facilitate iterative decoding, the received data of a whole decoding block have to be stored in a memory, whose size is proportional to the Turbo block size. Hence, Turbo decoders usually require large memory storage. Therefore serial decoding architectures are widely used in practice.

A typical serial Turbo decoder architecture is shown in Fig. 1. It has only one soft-input soft-output (SISO) decoder, which works in a time-multiplexed way as proposed in Suzuki (2000). Each iteration is decomposed into two decoding phases, i.e., *the sequential decoding phase*, in which the data are processed in sequential order, and *the interleaved decoding phase*, in which the source data are processed in an interleaved order. Both probability MAP algorithm proposed in Berrou (1993) and SOVA proposed in Hagenauer (1989) can be employed for the SISO decoding.

The serial Turbo decoder includes two memories: one is used for received soft symbols, called the input buffer or the receiver buffer, the other is used to store the extrinsic information, denoted as the interleaver memory. The extrinsic information is feedback as the *a priori* information for next decoding. The input buffer is normally indispensable. With regard to the interleaver memory, either two ping-pong buffers can be used to complete one Load and one Write operations required to process each information bit within one cycle or one single-port memory can be employed to fulfil the two required operations within two clock cycles. A memory-efficient architecture was presented by Wang (2003a) and Parhi, which can process both Read and Write operations at the same cycle using single-port memories with the aid of small buffers.

Turbo decoder works as follows. The SISO decoder takes soft inputs (including the received systematic bit y_s and the received parity bit y_p^1 or y_p^2) from the input buffer and the *a priori* information from the interleaver memory. It outputs the log likelihood ratio $LLR(k)$, and the extrinsic information, $L_{ex}(k)$, for the k-th information bit in the decoding sequence. The extrinsic information is sent back as the new *a priori* information for next decoding. The interleaving and de-interleaving processes are completed in an efficient way. Basically the data are loaded according to the current decoding sequence. For instance, the extrinsic information is loaded in sequential order at sequential decoding phase while being loaded in the interleaved order at the interleaved decoding phase. After processing, the new extrinsic information is written back to the original places. In this way, no de-interleave pattern is required for Turbo decoding.

3. High Speed Log-MAP Decoder Design

3.1 Maximum A Posterior (MAP) algorithm

As discussed in Section 2, practical Turbo decoders usually employ serial decoding architectures, such as Suzuki (2000), for area-efficiency. Thus, the throughput of a Turbo decoder is highly limited by the clock speed and the maximum number of iterations to be performed. To facilitate iterative decoding, Turbo decoders require soft-input soft-output decoding algorithms, among which the probability MAP algorithm proposed in Bahl (1974) is widely adopted for its excellent performance.

The MAP algorithm is commonly implemented in log domain, thus called Log-MAP (Wang 1999). The Log-MAP algorithm involves recursive computation of forward state metrics (simply called α metrics) and backward state metrics (simple called β metrics). The log-likelihood-ratio is computed based on the two types of state metrics and associated branch metrics (denoted as γ metrics). Due to different recursion directions in computing α and β metrics, a straightforward implementation of Log-MAP algorithm will not only consume large memory but also introduce large decoding latency. The sliding window approach was proposed by Viterbi (1998) to mitigate this issue. In this case, pre-backward recursion operations are introduced for the warm-up process of real backward recursion. For clarity, we denote the pre-backward recursion computing unit as **β0** unit and denote the real (effective or valid) backward recursion unit as **β1** unit. The details of Log-MAP algorithm will not be given in the chapter. Interested readers are referred to Wang (1999).

The timing diagram for typical sliding-window-based Log-MAP decoding is shown in Fig. 2, where SB1, SB2, etc, stand for consecutive sub-blocks (*i.e.*, sliding windows), the branch metrics computation was computed together with pre-backward recursion, though not shown in the figure for simplicity and clarity.

The structure of a typical serial Turbo decoder based on log-MAP algorithm is shown in Fig. 3, where the soft-output unit is used to compute LLR and extrinsic information (denoted as Lex), the interleaver memory is used to store the extrinsic information for next (phase of) decoding. It can be seen from the figure that both the branch metric unit (BMU) and soft-output unit (SOU) can be pipelined for high speed applications. However, due to recursive computation, three state metrics computation units form the high-speed bottleneck. The reason is that the conventional pipelining technique is not applicable for raising the effective processing speed unless one MAP decoder is used to process more than one Turbo code blocks or sub-blocks as discussed in Lee (2005). Among various high-speed recursion architectures in the literature such as Lee (2005), Urard (2004), Boutillon (2003), Miyouchi (2001) and Bickerstaff (2003), the designs presented in Urard (2004) and Bickerstaff (2003) are most attractive. In Urard (2004), an offset-add-compare-select (OACS) architecture is proposed to replace the traditional add-compare-select-offset (ACSO) architecture. In addition, the look-up table (LUT) is simplified with only 1-bit output, and the computation of absolute value is avoided through introduction of the reverse difference of two competing path (or state) metrics. An approximate 17% speedup over the traditional Radix-2 ASCO architecture was reported. With one-step look-ahead operation, a Radix-4 ACSO architecture can be derived. Practical Radix-4 architectures such as Miyouchi (2001) and Bickerstaff (2003) always involve approximations in order to achieve higher effective speedup. For instance, the following approximation is adopted in Bickerstaff (2003):

$$\max^*(\max^*(A, B), \max^*(C, D))=\max^*(\max(A, B), \max(C, D)), \tag{1}$$

where

$$\max^*(A, B)= \max(A,B)+\log(1+e^{-|A-B|}). \tag{2}$$

This Radix-4 architecture can generally improve the processing speed (equivalent to twice of its clock speed) by over 40% over the traditional Radix-2 architecture, and it has *de facto* the highest processing speed among all existing (MAP decoder) designs found in the literature.

However, the hardware will be nearly doubled compared to the traditional ACSO architecture presented in Urard (2004).

In this section, we will first present an advanced Radix-2 recursion architecture based on algorithmic transformation, approximation and architectural level optimization, which can achieve comparable processing speed as the state-of-the-art Radix-4 design while having significantly lower hardware complexity. Then we discuss an improved Radix-4 architecture that is 32% faster than the best existing approach.

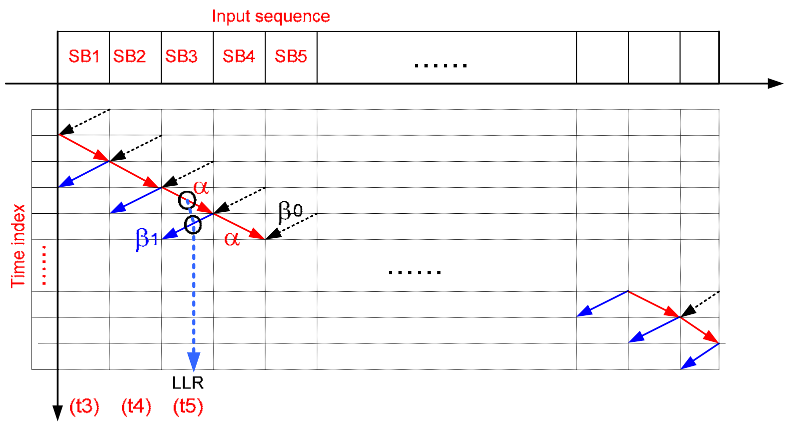


Fig. 2. The timing diagram for Log-MAP decoding.

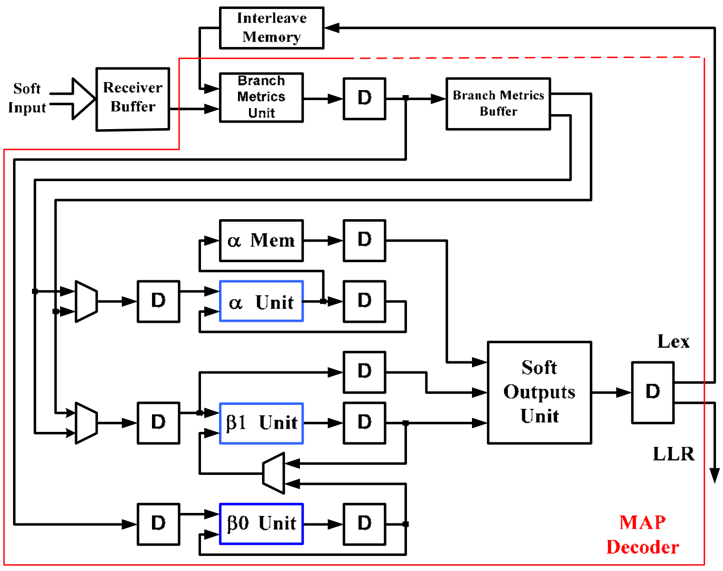


Fig. 3. A Log-MAP Turbo decoder structure.

3.2 An Advanced High-Speed Radix-2 Recursion Architecture for MAP Decoders

For convenience in later discussion, we first give a brief introduction to MAP-based Turbo decoder structure. As shown in Fig. 3, the branch metrics unit (**BMU**) takes inputs from the receiver buffer and the interleaver memory. The outputs of BMU are directly sent to the pre-backward recursion unit (*i.e.*, β0 unit). The previously stored branch metrics for consecutive sliding windows are input to the forward recursion unit (*i.e.*, α unit) and the effective backward recursion unit (*i.e.*, β1 unit), respectively. The soft output unit (**SOU**) that is used to compute the log likelihood ratio (**LLR**) and the extrinsic information (Lex) takes inputs from the previously stored α metrics, the currently computed β metrics and the previously stored branch metrics (γ). The **SOU** starts to generate soft outputs after the branch metrics have been computed for the first two sliding windows. It can be observed that the high-speed bottleneck of a Log-MAP decoder lies in the three recursive computation units since both BMU and SOU can be simply pipelined for high-speed applications. Thus, this section is dedicated to the design of high-speed recursive computation units, *i.e.*, α, β0 and β1 units shown in Fig. 3.

It is known from Log-MAP algorithm that all the three recursion units have similar architectures. So we will focus our discussion on design of α units. The traditional design for α computation is illustrated in Fig. 4, where the **ABS** block is used to compute the absolute value of the input and the **LUT** (*i.e.*, look-up table) block is used to implement a nonlinear function $\log(1+e^{-x})$, where x > 0. For simplicity, only one branch (*i.e.*, one state) is drawn. The overflow approach (Wu 2001) is assumed for normalization of state metrics as used in conventional Viterbi decoders.

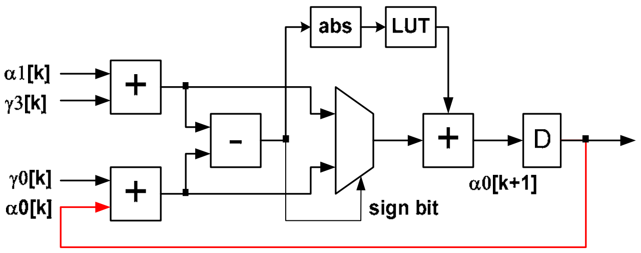


Fig. 4. The original recursion architecture: Arch-O.

It can be seen that the computation of the recursive loop consists of three multi-bit additions, the computation of absolute value and a random logic to implement the LUT. As there is only one delay element in each recursive loop, the traditional retiming technique in Denk (1998) can not be used to reduce the critical path.

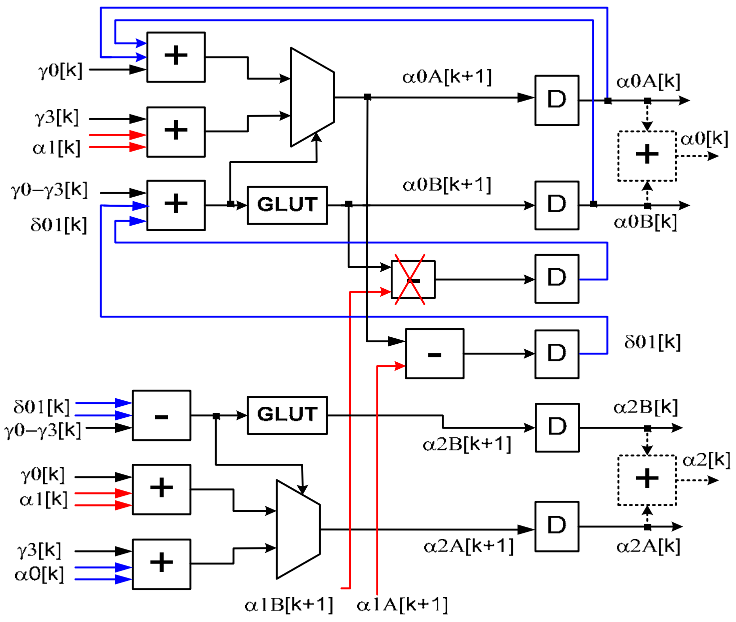


Fig. 5. An advanced Radix-2 fast recursion architecture.: Arch-A

In this work, we propose an advanced Radix-2 recursion architecture shown in Fig. 5. Here we first introduce a difference metric for each competing pair of states metrics (*e.g.*, **α0** and **α1** in Fig. 4) so that we can perform the front end addition and the subtraction operations simultaneously in order to reduce the computation delay of the loop. Secondly, we employ a generalized LUT (see GLUT in Fig. 5) that can efficiently avoid the computation of absolute value instead of introducing another subtraction operation as in Urard (2004). Thirdly, we move the final addition to the input side as with the OACS architecture in Boutillon (2003) and then utilize one stage carry-save structure to convert a 3-number addition to a 2-number addition. Finally, we make an intelligent approximation in order to further reduce the critical path.

The following equations are assumed for the considered recursive computation shown in Fig. 5:

$$\alpha 0[k+1] = \max{}^*(\alpha 0[k]+\gamma 0[k],\ \ \alpha 1[k]+\gamma 3[k]), \tag{3}$$

$$\alpha 2[k+1] = \max{}^*(\alpha 0 k]+\gamma 3[k],\ \ \alpha 1[k]+\gamma 0[k]),$$

where max* function is defined in (2).

In addition, we split each state metrics into two terms as follows:

$$\alpha 0[k] = \alpha 0A[k] + \alpha 0B[k],$$
$$\alpha 1[k] = \alpha 1A[k] + \alpha 1B[k],$$
$$\alpha 2[k] = \alpha 2A[k] + \alpha 2B[k]. \tag{4}$$

Similarly, the corresponding difference metric is also split into two terms:

$$\delta_{01A}[k] = \alpha_{0A}[k] - \alpha_{1A}[k],$$
$$\delta_{01B}[k] = \alpha_{0B}[k] - \alpha_{1B}[k]. \quad (5)$$

In this way, the original add-and-compare operation is converted as an addition of three numbers, *i.e.*,

$$(\alpha_0 + \gamma_0) - (\alpha_1 + \gamma_3) = (\gamma_0 - \gamma_3) + \delta_{01A} + \delta_{01B} \quad (6)$$

where $\gamma 0 - \gamma 3$ is computed by BMU, the time index [k] is omitted for simplicity. In addition, the difference between the two outputs from two GLUTs, *i.e.*, δ_{01B} , can be neglected. From extensive simulations, we found that this small approximation doesn't cause any performance loss in Turbo decoding with either AWGN channels or Raleigh fading channels. This fact can be simply explained in the following. If one competing path metrics (*e.g.*, $p0 = \alpha 0 + \gamma 0$) is significantly larger than the other one (*e.g.*, $p1 = \alpha 1 + \gamma 3$), the GLUT output will not change the decision anyway due to their small magnitudes. On the other hand, if the two competing path metrics are so close that adding or removing a value from one GLUT may change the decision (*e.g.*, from *p0>p1* to *p1>p0*), picking any survivor (*p0 or p1*) should not make big difference.

At the input side, a small circuitry shown in Fig. 6 is employed to convert an addition of 3 numbers to an addition of 2 numbers, where FA and HA represents full-adder and half-adder respectively, XOR stands for exclusive OR gate, d0 and d1 correspond to the 2-bit output of GLUT. The state metrics and branch metrics are represented with 9 and 6 bits, respectively in this example. The sign extension is only applied to the branch metrics. It should be noted that an extra addition operation (see dashed adder boxes) is required to integrate each state metric before storing it into the α memory.

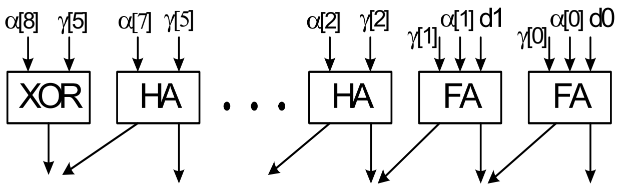


Fig. 6. A carry-save structure in the front end of Arch-A.

The generalized LUT (GLUT) structure is shown in Fig. 7, where the computation of absolute value is eliminated by including the sign bit into 2 logic blocks, *i.e.*, **Ls2** and **ELUT**, where the Ls2 function block is used to detect if the absolute value of the input is less than 2.0, and the ELUT block is a small LUT with 3-bit inputs and 2-bit outputs. It can be derived that $Z = \bar{S}\overline{b_7 \ldots + b_4 + b_3} + S(b_7 \ldots b_4 b_3)$. It was reported in Gross (1998) that using two output values for the LUT only caused a performance loss of 0.03 dB from the floating point simulation for a 4-state Turbo code. The approximation is described as follows:

If $|x|<2$, f(x) = 3/8; else f(x) = 0; (7)

where x and f(x) stands for the input and the output of the LUT, respectively. In this approach, we only need to check if the absolute value of the input is less than 2 or not, which can be performed by the **Ls2** block in Fig. 7. A drawback of this method is that its performance would be significantly degraded if only two bits are kept for the fractional part of the state metrics, which is generally the case.

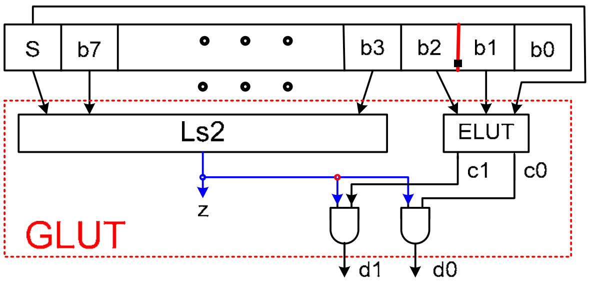


Fig. 7. The structure of GLUT used in Arch-A.

|x|	0.0	0.50	1.0	1.50
f(x)	¾	2/4	¼	¼

Table 1. The proposed LUT approximation

In our design, both the inputs and outputs of the LUT are quantized with 4 levels. The details are shown in Table 1. The inputs to ELUT are treated as a 3-bit signed binary number. The outputs of ELUT are *AND*ed with the output of **Ls2** block. This means, if the absolute value of the input is greater than 2.0, the output from the GLUT is set as 0. Otherwise the output from ELUT will be the final output.

The ELUT can be implemented with combinational logic for high speed applications. Its computation latency is smaller than the latency of **Ls2** block. Therefore, the overall latency of the GLUT is almost the same as the above-discussed simplified method, where the total delay consists of 1 MUX delay and the computation delay of **Ls2**.

After all the above optimization, the critical path of the recursive architecture is reduced to 2 multi-bit additions, one 2:1 MUX operation and 1-bit addition operation, which saves nearly 2 multi-bit adder delay compared to the traditional ACSO architecture. We will show detailed comparisons in subsection 3.4.

3.3 An Improved Radix-4 Architecture for MAP Decoders

In the following, we discuss an improved Radix-4 recursion architecture. The computation for $\alpha 0[k+2]$ is expressed as follows:

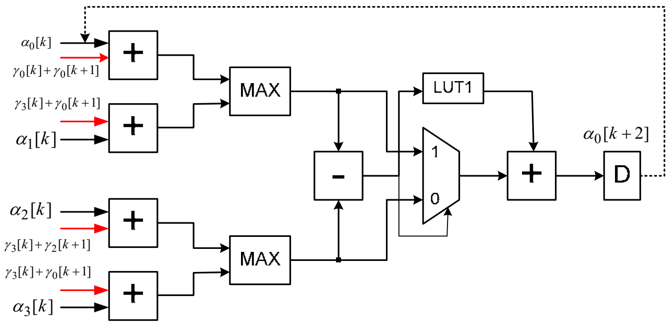
$$\begin{aligned}\alpha 0[k+2] &= \max{}^*(\alpha 0[k+1]+\gamma 0[k+1],\ \ \alpha 1[k+1]+\gamma 3[k+1]) \\ &= \max{}^*(\max{}^*(\alpha 0[k]+\gamma 0[k],\ \ \alpha 1[k]+\gamma 3[k])+\gamma 0[k+1], \\ &\qquad \max{}^*(\alpha 2k]+\gamma 2[k],\ \ \alpha 3[k]+\gamma 1[k])+\gamma 3[k+1]),\end{aligned} \tag{8}$$

where

$$\alpha 1[k+1] = \max{}^*(\alpha 2[k]+\gamma 2[k],\ \ \alpha 3[k]+\gamma 1[k]). \tag{9}$$

In Bickerstaff (2003), Lucent Bell Labs proposed the following approximation:

$$\begin{aligned}\alpha 0[k+2] &\approx \max{}^*(\max(\alpha 0[k]+\gamma 0[k],\ \ \alpha 1[k]+\gamma 3[k])+\gamma 0[k+1], \\ &\qquad \max(\alpha 2k]+\gamma 2[k],\ \ \alpha 3[k]+\gamma 1[k])+\gamma 3[k+1]).\end{aligned} \tag{10}$$


$\alpha_0[k]$
$\gamma_0[k]+\gamma_0[k+1]$
$\gamma_3[k]+\gamma_0[k+1]$
$\alpha_1[k]$
$\alpha_2[k]$
$\gamma_3[k]+\gamma_2[k+1]$
$\gamma_3[k]+\gamma_0[k+1]$
$\alpha_3[k]$
+
MAX
-
LUT1
1
0
$\alpha_0[k+2]$
D
Fig. 8. The Radix-4 architecture proposed by Lucent Bell Labs: Arch-L.

This approximation is reported to have a 0.04 dB performance loss compared to the original Log-MAP algorithm. The architecture to implement the above computation is shown in Fig. 8 for convenience in later discussion. As it can be seen, the critical path consists of 4 multi-bit adder delay, one generalized LUT delay (Note: the LUT1 block includes absolute value computation and a normal LUT operation) and one 2:1 MUX delay.

Similarly, we can take an alternative approximation as follows:

$$\begin{aligned}\alpha 0[k+2] &\approx \max(\max{}^*(\alpha 0[k]+\gamma 0[k],\ \ \alpha 1[k]+\gamma 3[k])+\gamma 0[k+1], \\ &\qquad \max{}^*(\alpha 2k]+\gamma 2[k],\ \ \alpha 3[k]+\gamma 1[k])+\gamma 3[k+1]).\end{aligned} \tag{11}$$

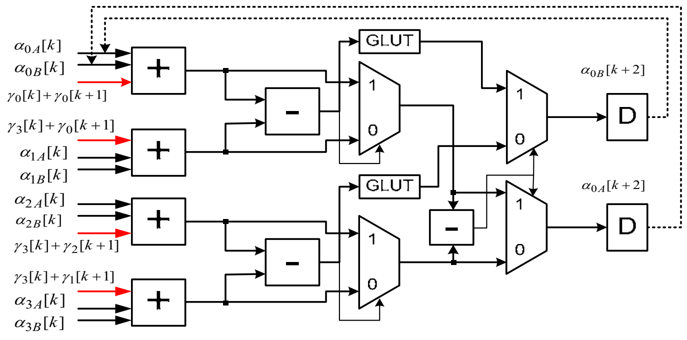


Fig. 9. The improved Radix-4 recursion arch.: Arch-B.

Intuitively, Turbo decoder employing this new approximation should have the same decoding performance as using equation (7). While directly implementing (11) does not bring any advantage to the critical path, we intend to take advantages of the techniques that we developed in subsection 3.2. The new architecture is shown in Fig. 9. Here we split each state metric into two terms and we adopt the same GLUT structure as we did before. In addition, a similar approximation is incorporated as with Arch-A. In this case, the outputs from GLUT are not involved in the final stage comparison operation. It can be observed that the critical path of the new architecture is close to 3 multi-bit adder delay. To compensate for all the approximation introduced, the extrinsic information generated by the MAP decoder based on this new Radix-4 architecture should be scaled by a factor around 0.75.

3.4 Performance Comparisons

For quantitative comparison in hardware complexity and processing speed, we used TSMC 0.18 um standard cells to synthesize one α unit for 5 different recursion architectures, *i.e.*, 1) the traditional ACSO architecture: Arch-O, 2) the reduced-precision Radix-2 recursion architecture presented in Urard (2004): Arch-U, 3) the Radix-4 architecture proposed by Lucent: Arch-L, 4) the advanced Radix-2 recursion architecture presented in this work: Arch-A, and 5) the improved Radix-4 recursion architecture: Arch-B. All the state metrics are quantized as 9 bits while the branch metrics are represented using 6 bits. The detailed synthesis results are listed in Table 2.

It can be observed that the proposed Radix-2 architecture has comparable processing speed as the Radix-4 architecture proposed by Bell Labs with significantly lower complexity, while the improved Radix-4 architecture is 32% faster with only 9% extra hardware. This amount of hardware overhead should be negligible compared to an entire Turbo decoder. It can also be seen that the new Radix-4 architecture achieves twice speedup over the traditional Radix-2 recursion architecture.

	Max clock Frequency (Mhz)	Relative Area	Relative Processing Speed
Arch-O	241	1.0	1.0
Arch-U	355	1.14	1.39
Arch-L	182	1.82	1.51
Arch-A	370	1.03	1.54
Arch-B	241	1.99	2.0

Table 2. Comparison for various recursion architectures

We have performed extensive simulations for Turbo codes using the original MAP and using various approximations. Fig. 10 shows the BER (bit-error-rate) performance of a rate-1/3, 8-state, block size of 512 bits, Turbo code using different MAP architectures. The simulations were undertaken under the assumption of AWGN channel and BPSK signaling. A maximum of 8 iterations was performed. More than 40 million random information bits were simulated for both Eb/No=1.6 dB and Eb/No=1.8dB cases. It can be noted from Fig. 10 that there is no observable performance difference between the true MAP algorithm and two approximation methods associated with the proposed recursion architectures while the approximation employed in Urard (2004) caused approximately 0.2 dB performance degradation in general.

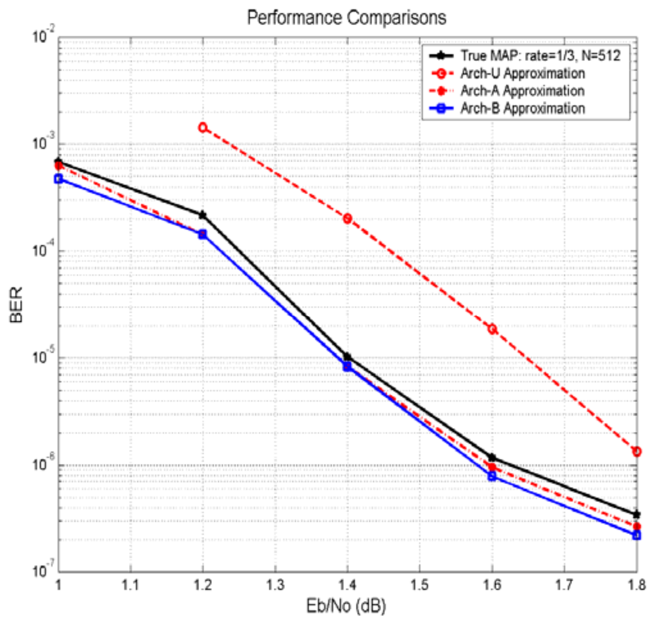


Fig. 10. Performance comparisons between the original MAP and some approximations.

We argue that the proposed Radix-4 recursion architecture is optimal for high-speed MAP decoders. Any (significantly) faster recursion architecture (*e.g.*, a possible Radix-8 architecture) will be at the expense of significantly increased hardware. On the other hand, when the target throughput is moderate, these fast recursion architectures can be used to reduce power consumption because of their dramatically reduced critical paths.

In general, the throughput of a serial Turbo decoder is significantly limited by recursive computation and VLSI technology used to implement it. On the other hand, increasing the clock frequency would cause the dynamic power dissipation to increase linearly. Therefore, it is not a good option to increase the clock frequency for high throughput applications.

4. Efficient Parallel Turbo Decoding Architectures

In this section, we first introduce an area-efficient parallel Turbo decoding scheme, where a data frame is partitioned into several equivalent segments with proper overlap between adjacent segments, and sliding-window-based Turbo decoding is then applied to each segment. The goal is to maintain the same memory requirement as serial decoding case while increasing the complexity of the logic components only, thus linearly increase the system throughput with sub-linearly increased hardware.

The major challenge lies in real implementation. As each memory (receiver memory or extrinsic information memory) needs to support multiple data (for multiple component soft-input soft-output decoders) at the same cycle, memory access conflicts are inevitable unless M-port (M = parallelism level) memory is used, which contradicts the original low-complexity design objective. Two different solutions are proposed in this work. First of all, if interleave patterns are free to design, we can adopt dividable interlavers, which inherently ensure no memory access conflict after proper memory partition. For practical applications wherein Turbo code interleave patterns are fixed, e.g., 3GPP and 3GPP2, a more generic solution is introduced in this chapter. By introducing some small buffers for data and addresses as well, we are able to avoid memory access conflict under any practical random interleavers. Combining all the proposed techniques, it is estimated that 200 Mb/s Turbo decoder is feasible with current CMOS technology with moderate complexity.

4.1 Area-efficient parallel decoding schemes

Parallel processing has been widely used in low power and high throughput circuit design. Multiple serial Turbo decoders can be concatenated in a parallel way (see Fig. 11 A) or in a serial way. In either case, both the area and power are increased linearly as the throughput increases. The area-efficient parallel Turbo decoding schemes were first proposed in Wang (2001). The idea is to let multiple SISO decoders work on the same data frame simultaneously. In this case, only the hardware of the SISO decoder part needs to be increased while the total hardware in the memory part remains unchanged on a large extent (see Fig. 11 B). As the memory part usually dominates the overall hardware of a Turbo decoder, the area-efficient parallel Turbo decoding architectures are expected to achieve multiple times the throughput of a serial decoding decoder with a small fraction of hardware overhead. Unfortunately, the real implementation issues were not covered in the original work.

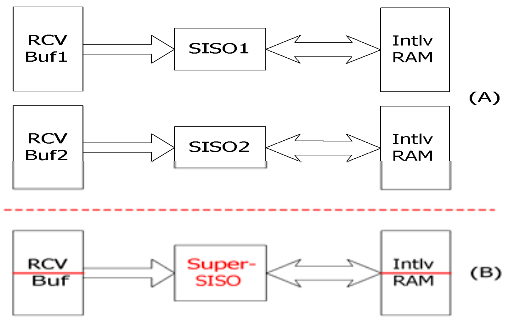
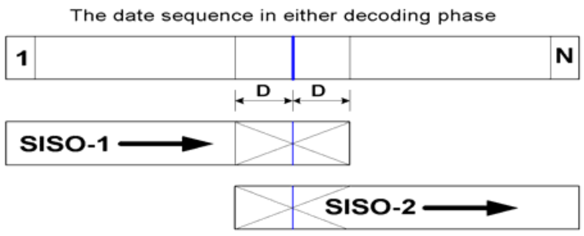


Fig. 11. Traditional vs. area-efficient parallel Turbo decoding.



a) A simple area-efficient 2-level parallel Turbo decoding scheme.

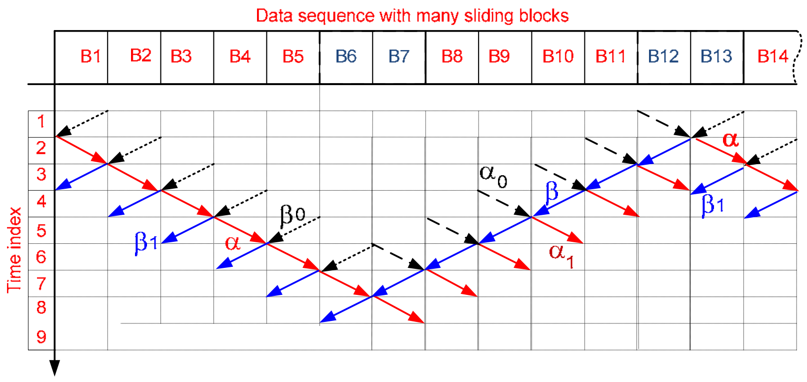


Fig. 12. b) Multi-level parallel Turbo decoding scheme based on sliding-window approach.

A simple area-efficient 2-level parallel Turbo decoding scheme is shown in Fig. 12 a), where a sequence of data is divided into two segments with equivalent length. There is an overlap with length of 2D between two segments, where D equals the sliding window size if Log-MAP (or MAX-Log-MAP) algorithm is employed in the SISO decoder or the survivor length if SOVA is employed, as proposed in Wang (2003c). The two SISO decoders work on two different data segments (with overlap) at the same time. Fig. 12 b) shows (a portion of) a sliding-window-based area-efficient multi-level parallel Turbo decoding data flow, where

SISO decoders responsible for two adjacent data segments are processing data in reverse directions in order to reuse some of the previously computed branch metrics and state metrics. For other parallel decoding schemes with various trade-offs, interested reader are referred to Wang (2001) and Zhang (2004).

In this section, we will focus on area-efficient two-level parallel decoding schemes. It can be observed from Fig. 12.a) that two data accesses per cycle are required for both the receiver buffer and the interleaver memory (assuming two ping-pong buffers are used for the interleaver memory). Using a dual-port memory is definitely not an efficient solution since a normal dual-port memory consumes as much area as two single-port memories with the same memory size.

4.2 Area-efficient parallel decoding architectures

In order to maintain the total hardware of these memories, we choose to partition each memory into multiple segments to support multiple data accesses per cycle.

Memory partitioning can be done in various ways. An easy and yet effective way is to partition the memory according to a number of least significant bits (lsb's) of the memory address. To partition a memory into two segments, it can be done according to the least significant bit (lsb). For the resultant two memory segments, one contains data with even addresses and the other contains data with odd addresses.

The memory partitioning can also be done in a fancy way, *e.g.*, to partition the memory into 2 segments, let the 1st memory segment contain data with addresses $b_2b_1b_0$ ={0, 2, 5, 7}, and the 2nd segment contain data with addresses $b_2b_1b_0$ ={1, 3, 4, or 6}, where $b_2b_1b_0$ denotes the 3 lsb's. Depending on the applications, different partitioning schemes may lead to different hardware requirements.

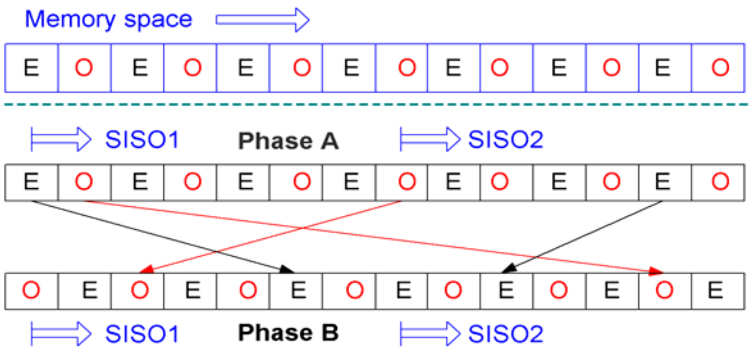
For the two memory segments, it is possible, in principle, to support two data accesses within one cycle. However, it is generally impossible to find an efficient partitioning scheme to ensure the target addresses (for both Load and Write operations) are always located in different segments at each cycle because the Turbo decoder processes the data in different orders during different decoding phases. Consider a simple example, given a sequential data sequence {0, 1, 2, 3, 4, 5, 6, 7}, the interleaved data sequence is assumed as {2, 5, 7, 3, 1, 0, 6, 4}. If we partition the memory into two segments according to the sequential decoding phase, *i.e.*, one segment contains data sequence {0, 1, 2, 3} and the other has {4, 5, 6 7}. During the sequential phase, SISO-1 works on data set {0, 1, 2, 3} and SISO-2 works on {4, 5, 6, 7}. So there is no data conflict (note: the overlap between two segments for parallel decoding is ignored in this simple example). However, during the interlaved decoding phase, SISO-1 will process data in set {2, 5, 7, 3} and SISO-2 will process data in set {1, 0, 6, 4}, both in sequential order. It is easy to see that, at the first cycle, both SISO decoders require data located in the first segment (1st and 2nd data in the input data sequence). Thus a memory access conflict occurs. More detailed analysis can be found in Wang (2003b). The memory access issue in parallel decoding can be much worse when multiple rates and multiple block sizes of Turbo codes are supported in a specific application, *e.g.*, in 3GPP CDMA systems.

In principle, the data access conflict problem can be avoided if the Turbo interleaver is specifically designed. A generalized even-odd interleave is defined below:

- All even indexed data are interleaved to odd addresses,
- All odd indexed data are interleaved to even addresses.

Back to the previous example, an even-odd interleaver may have an interleaved data sequence as {3, 6, 7, 4, 1, 0, 5, 2}.

With an even-odd interleaver, we can partition each memory into two segments: one contains even-addressed data and the other with odd-addressed data.


Memory space
E O E O E O E O E O E O E O
SISO1
Phase A
SISO2
E O E O E O E O E O E O E O
O E O E O E O E O E O E O E
SISO1
Phase B
SISO2
Fig. 13. Data processing in either decoding phase with an even-odd interleaver.

As shown in Fig. 13, the data are processed in an order of {even address, odd address, even , odd, …} in Phase A (*i.e.*, the sequential phase) and an order of {odd address, even address, odd, even, ..} in Phase B (*i.e.*, the interleaved phase). If we let SISO-1 and SISO-2 start processing from different memory segments, then there would be no data access conflict during either decoding phase. In other words, it is guaranteed that both SISO decoders always access data located in different memory segments.

More complicated interleavers can be designed to support multiple (M>=2) data accesses per cycle. The interested readers are referred to Wang (2003c), He (2005), Giulietti (2002), Bougard (2003), and Kwak (2003) for details.

In real applications, the Turbo interleaver is usually not free to design. For example, they are fixed in WCDMA and CDMA 2000 systems. Thus, a generic parallel decoding architecture is desired to accommodate all possible applications. Here we propose an efficient memory arbitration scheme to resolve the problem of data access conflict in parallel Turbo decoding.

The fundamental concept is to partition one single-port memory into S (S>=2) segments and use B (B >=2) small buffers to assist reading from or writing data back to the memory, where S does not have to be the same as B. For design simplicity, both S and B are normally chosen to be a power of 2. S is better chosen to be a multiple of B. In this chapter, we will consider only one simple case, *i.e.*, S=2, B=2. For all other cases of B=2, they can be easily extended from the illustrated case. However, for cases with B>2, the control circuitry will be much more complicated.

For the receiver buffer, only the Load operation is involved while bother Load and Write operations are required for the interleaver memory. So we will focus our discussion on the interleaver memory.

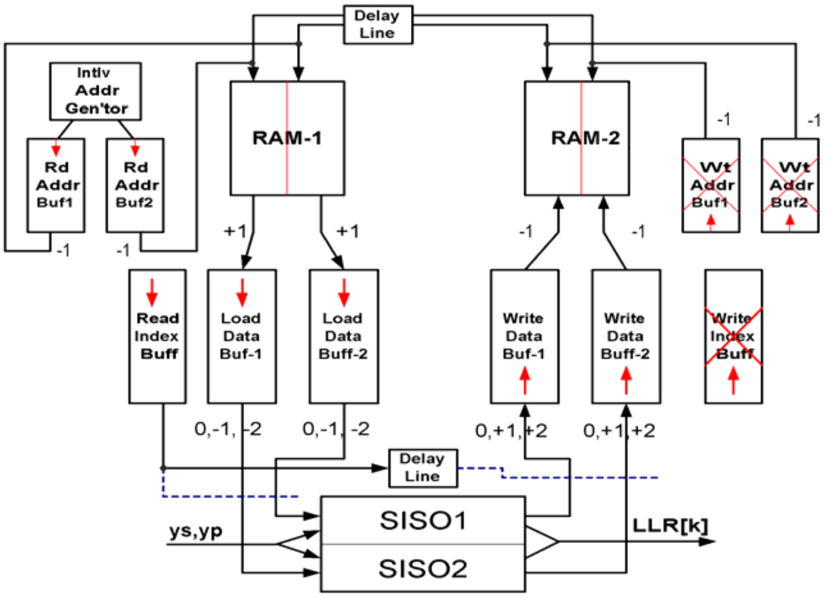


Fig. 14. The area-efficient parallel decoding architecture.

With regard to the interleaver memory, we assume two ping-pong buffers (i.e., RAM1 and RAM2 shown in Fig.14) are used to ensure that one Load and one Write operations can be completed within one cycle. Each buffer is partitioned into two segments: Seg-A contains even addressed data and Seg-B contains odd-addressed data. For simplicity, we use Seg-A1 to represent the even-addressed part of RAM1, Seg-B1 to represent the odd-addressed part of RAM1. The similar representations are used for RAM2 as well.

An area-efficient 2-parall Turbo decoding architecture is shown in Fig. 14. The interleaver address generator (IAG) generates two addresses for two SISO decoders respectively at each cycle. They must belong to one of the following cases: (1) two even addresses, (2) two odd addresses and (3) one even and one odd address. In Case 1, two addresses are put into Read Address Buffer 1 (RAB1), In Cases 2, two addresses are put in Read Address buffer 2 (RAB2). In Cases 3, the even address goes to RAB1 and the odd address goes to RAB2.

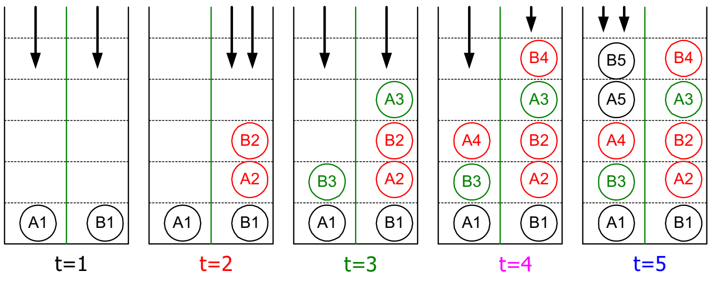
A small buffer called Read Index Buffer (RIB) is introduced in this architecture. This buffer is basically a FIFO (first-in first-out). Two bits are stored at each entry. The distinct four values represented by the two bits have following meanings:

- 00: 1st and 2nd even addresses for SISO-1 and SISO-2 respectively,
- 11: 1st and 2nd odd addresses for SISO-1 and SISO-2 respectively,
- 01: the even address is used for SISO-1 and the odd address for SISO-2,
- 10: the even address is used for SISO-1 and the odd address for SISO-2.

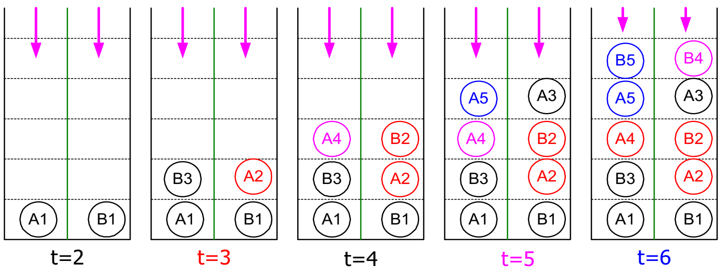
After decoding for a number of cycles (*e.g.*, 4 ~ 8 cycles), both RAB1 and RAB2 will have a small amount of data. Then the data from the top of RAB1 and RAB2 are used respectively as the addresses to Seg-A1 and Seg-B1 respectively. After one cycle, two previously stored extrinsic information symbols are loaded to Load Data Buffer 1 (LAB1) and Load Data Buffer 2 (LDB2). Then the data stored in RIB will be used to direct the outputs from both load buffers to feed both SISO decoders. For example, if the data at the top of RIB is 00, then LDB1 output two data to SISO-1 and SISO-2 respectively. The detailed actions of address buffers and data buffers controlled by RIB for loading data are summarized in Table 3.

Value of RIB	Action of address buffers	Action of data buffers
0	RAB1 load nothing, RAB2 loads 2 addresses, 1st for Seg-1 and 2nd for Seg-2	LDB1 output nothing, LDB2 out 2 data, 1st for SISO1 and 2nd for SISO2
1	RAB1 load 1 address for Seg-1, RAB2 load 1 address for Seg-2	LDB1 output 1 for SISO1 and LDB2 out 1 for SISO2
2	RAB2 load nothing, RAB1 load 2 addresses, 1st for Seg-1 and 2nd for Seg-2	LDB1 output 2 data, 1st for SISO1 and 2nd for SISO2
3	RAB1 load 1 address for Seg-1, RAB2 load 1 for Seg-2	LDB1 output 1 for SISO2 and LDB2 out 1 for SISO1

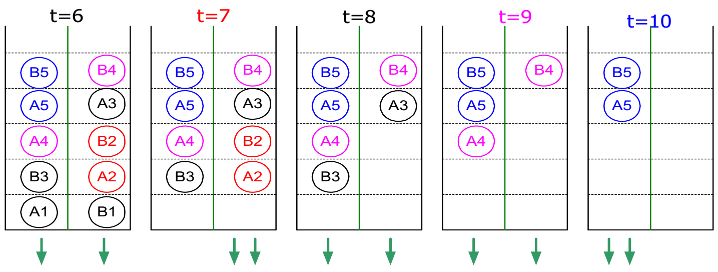
Table 3. Summary of loading data operations.



(a) load addresses to RAB1 and RAB2



(b) load data to RDB1 and RDB2



(c) output data to SISO1 and SISO2

Fig. 15. Procedure of loading data from memory to SISO decoders.

A simple example is shown in Fig. 15 to illustrate the details of loading data from memory to SISO decoders, where, for instance, A3 (B5) denotes the address of data to be loaded or the data itself corresponding to memory segment A (segment B) for the 3rd (5th) data in the processing sequence. Fig. 15 (a) shows the details of feeding two dada addresses to two address buffers at every cycle. Fig. 15 (b) shows the details of feeding one data to each data buffer at every cycle. The details of outputting the required two data to both SISO decoders are shown in Fig. 15 (c). As can be seen, both SISO1 and SISO2 can get their required data that may be located in the same memory segment at the same cycle (*e.g.*, t=7 in this example). A tiny drawback of this approach is that a small fixed latency is introduced. Fortunately this latency is negligible compared with the overall decoding cycles for a whole Turbo code block in most applications.

In general (*e.g.*, when sliding-window-based MAP algorithm is employed for SISO decoders), the write sequence for each SISO decoder is the delayed version of the read sequence. So the write index buffer and write address buffers can be avoided by using delay lines as shown in Fig. 14.

At each cycle, two extrinsic information symbols are generated from the two SISO decoders. They may both feed Write Data Buffer 1 (WDF1), or both feed Write Data Buffer 2 (WDF2), or one feeds WDF1 and the other feeds WDF2. The actual action is controlled by the delayed output from RIB.

Similar to loading data, after the same delay (*e.g.*, 4 ~ 8 cycles) from the first output of either SISO decoder, the data from WDB1 and WDB2 will be written back to Seg-B1 and Seg-B2 respectively. In the next decoding phase, RAM-1 and RAM-2 will exchange roles. So some extra MUXs must be employed to facilitate this functional switch.

5.3 Implementation issues and simulation results

The integer numbers shown in Fig. 14 indicated the data flow associated with each buffer. For example, {0, -1, -2} is shown along the output line of LDB1, which means LDB1 may output 0, 1, or 2 data at each cycle. It can be observed that all data and address buffers work in a similar way. One common feature of these buffers is that they have a constant data flow (+1 or –1) at one end (Load or Write) while having an irregular data flow ({0, -1, -2} or {0, +1, +2}) at the other end (Write or Load). On the average, incoming and outgoing data are largely balanced. So the buffer sizes can be very small.

The RIB can be implemented with a shift register as it has a regular data flow in both ends while a 3-port memory suffices to fulfill the functions of the rest buffers.

It has been found from our cycle-accurate simulations that it is sufficient to choose a buffer length of 25 for all buffers if the proposed 2-parallel architecture is applied in either WCDMA or CDMA2000 systems. The maximum Turbo block size for WCDMA system is approximately 5K bits. The lowest code rate is 1/3. Assume both the received soft inputs and the extrinsic information symbols are expressed as 6 bits per symbol.

- The overall memory requirement is 5K*(2+3)*6=150K bits.
- The total overhead of small buffers is approximately 25*3*2*(3*6+2*6) + 25*3*13*2~= 6K bits, where the factor 3 accounts for 3 ports of memory, 13 represents each address contains 13 binary bits.

It can be seen that the overhead is about 4% of total memory. With CDMA2000 systems, the overhead occupies an even smaller percentage in the total hardware because the maximum Turbo block size is several times larger.

For WCDMA systems, it is reasonable to assume the total area of a Log-MAP decoder consumes less than 15% of total hardware of a serial Turbo decoder. Thus, we can achieve twice the throughput with the proposed 2-parallel decoding architecture while spending less than 20% hardware overhead. If SOVA is employed in the SISO decoder, the overhead could be less than 15%. In either case, the percentage of the overhead will be even smaller in CDMA2000 systems.

Assume 6 iterations are performed at the most, the proposed 2-level parallel architecture, if implemented with TSMC 0.18 um technology, can achieve a minimum decoding throughput of 370 M*2/(6*2) > 60 Mbps. If state-of-the-art CMOS technology (*e.g.*, 65nm CMOS) is used, we can easily achieve 100Mbps data-rate with the proposed 2-parallel decoding architecture.

If an 4-parallel architecture is employed, over 200Mbps data rate can be obtained, though a direct extension of the proposed 2-paralell architecture will significantly complicate the control circuitry.

It is worthwhile to note, when the target throughput is moderately low, the proposed area-efficient parallel Turbo decoding architecture can be used for low power design. The reason is that the former architecture can work at a much lower clock frequency and the supply voltage can thus be reduced significantly.

6. Conclusions

Novel fast recursion architecture for Log-Map decoders have been presented in this chapter. Experimental results have shown that the proposed fast recursion architectures can increase the process speed significantly over traditional designs while maintaining the decoding performance. As a more power-efficient approach to increase the throughput of Turbo decoder, area-efficient parallel Turbo decoding schemes have been addressed. A hardware-efficient 2-parallel decoding architecture for generic applications is presented in detail. It has been shown that twice the throughput of a serial decoding architecture can be obtained with an overhead of less than 20% of an entire Turbo decoder. The proposed memory partitioning techniques together with the efficient memory arbitration schemes can be extended to multi-level parallel Turbo decoding architectures as well.

7. References

3rd Generation Partnership Project (3GPP), Technical specification group radio access network, multiplexing and channel coding (TS 25.212 version 3.0.0), http://www.3gpp.org.

3rd Generation Partnership Project 2 (3GPP2), http://www.3gpp2.org.

A. Giulietti *et al.* (2002), Parallel Turbo code interleavers: Avoiding collisions in accesses to storage elements, *Electron. Lett.*, vol. 38, no. 5, pp. 232–234, Feb. 2002.

A. J. Viterbi. (1998). An intuitive justification of the MAP decoder for convolutional codes, *IEEE J. Select. Areas Commun.*, vol.16, pp. 260-264, February 1998.

A. Raghupathy. (1998). Low power and high speed algorithms and VLSI architectures for error control coding and adaptive video scaling, Ph.D. dissertation, *Univ. of Maryland, College Park*, 1998.

B. Bougard *et al.* (2003). A scalable 8.7 nJ/bit 75.6 Mb/s parallel concatenated convolutional (Turbo-) codec, in *IEEE ISSCC Dig. Tech. Papers*, 2003, pp. 152–153.

C. Berrou, A. Clavieux & P. Thitimajshia. (1993). Near Shannon limit error correcting coding and decoding: Turbo codes, *ICC'93*, pp 1064-70.

E. Boutillon, W. Gross & P. Gulak. (2003). VLSI architectures for the MAP algorithm, *IEEE Trans. Commun.*, Volume 51, Issue 2, Feb. 2003, pp. 175 – 185.

E. Yeo, S. Augsburger, W. Davis & B. Nikolic. (2003). A 500-Mb/s soft-output Viterbi decoder, IEEE Journal of Solid-State Circuits, Volume 38, Issue 7, July 2003, pp:1234 – 1241.

H. Suzuki, Z. Wang & K. K. Parhi. (2000). A K=3, 2Mbps Low Power Turbo Decoder for 3rd Generation W-CDMA Systems, in *Proc. IEEE 1999 Custom Integrated Circuits Conf. (CICC)*, 2000, pp 39-42.

J. Hagenauer & P. Hoher. (1989). A Viterbi algorithm with soft decision outputs and its applications, *IEEE GLOBECOM*, Dallas, TX, USA, Nov. 1989, pp 47.1.1-7.

J. Kwak & K Lee (2002). Design of dividable interleaver for parallel decoding in turbo codes. *Electronics Letters*, vol. 38, issue 22, pp. 1362-64, Oct. 2002.

J. Kwak, S. M. Park, S. S. Yoon & K Lee (2003). Implementation of a parallel Turbo decoder with dividable interleaver, *ISCAS'03*, vol. 2, pp. II-65- II68, May 2003.

K. K. Parhi. (1999). VLSI Digital signal Processing Systems, *John Wiley & Sons*, 1999.

L.Bahl, J.Jelinek, J.Raviv & F.Raviv. (1974). Optimal Decoding of Linear Codes for minimizing symbol error rate, *IEEE Trans.s Inf. Theory*, vol. IT-20, pp.284-287, March 1974.

M. Bickerstaff, L. Davis, C. Thomas, D. Garret & C. Nicol. (2003). A 24 Mb/s radix-4 LogMAP Turbo decoder for 3 GPP-HSDPA mobile wireless, in *Proc. IEEE ISSCC Dig. Tech. Papers*, 2003, pp. 150–151.

P. Urard et al. (2004). A generic 350 Mb/s Turbo codec based on a 16-state Turbo decoder, in *Proc. IEEE ISSCC Dig. Tech. Papers*, 2004, pp. 424–433.

S. Lee, N. Shanbhag & A. Singer. (2005). A 285-MHz pipelined MAP decoder in 0.18 um CMOS, *IEEE J. Solid-State Circuits*, vol. 40, no. 8, Aug. 2005, pp. 1718 – 1725.

T. Miyauchi, K. Yamamoto & T. Yokokawa. (2001). High-performance programmable SISO decoder VLSI implementation for decoding Turbo codes, in *Proc. IEEE Global Telecommunications Conf.*, vol. 1, 2001, pp. 305–309.

T.C. Denk & K.K. Parhi. (1998). Exhaustive Scheduling and Retiming of Digital Signal Processing Systems, *IEEE Trans. Circuits and Syst. II*, vol. 45, no.7, pp. 821-838, July 1998

W. Gross & P. G. Gulak. (1998). Simplified MAP algorithm suitable for implementation of Turbo decoders, *Electronics Letters*, vol. 34, no. 16, pp. 1577-78, Aug. 1998.

Y. Wang, C. Pai & X. Song. (2002). The design of hybrid carry-lookahead/carry-select adders, *IEEE Trans. Circuits and Syst. II*, vol.49, no.1, Jan. 2002, pp. 16 -24

Y. Wu, B. D. Woerner & T. K. Blankenship, Data width requirement in SISO decoding with module normalization, *IEEE Trans. Commun.*, vol. 49, no. 11, pp. 1861–1868, Nov. 2001.

Y. Zhang & K. Parhi (2004). Parallel Turbo decoding, Proceedings of the 2004 International Symposium on Circuits and Systems (ISCAS04), Volume 2, 23-26 May 2004, pp: II - 509-12 Vol.2.

Z Wang, Y. Tan & Y. Wang. (2003b). Low Hardware Complexity Parallel Turbo Decoder Architecture, *ISCAS'2003*, vol. II, pp. 53-56, May 2003.

Z. He, S. Roy & Fortier (2005). High-Speed and Low-Power Design of Parallel Turbo Decoder Circuits and Systems, *IEEE International Symposium on Circuits and Systems (ISCAS'05)*, 23-26 May 2005, pp. 6018 – 6021.

Z. Wang & K. Parhi. (2003a). Efficient Interleaver Memory Architectures for Serial Turbo Decoding, *ICASSP'2003*, vol. II, pp. 629-32, May 2003.

Z. Wang & K. Parhi. (2003c). High Performance, High Throughput Turbo/SOVA Decoder Design, *IEEE Trans. on Commun.*, vol. 51, no 4, April 2003, pp. 570-79.

Z. Wang, H. Suzuki & K. K. Parhi. (1999). VLSI implementation issues of Turbo decoder design for wireless applications, in *Proc. IEEE Workshop on Signal Process. Syst. (SiPS)*, 1999, pp. 503-512.

Z. Wang, Z. Chi & K. K. Parhi. (2001). Area-Efficient High Speed Decoding Schemes for Turbo/MAP Decoders, in *Proc. IEEE ICASSP'2001*, pp 2633-36, vol. 4, Salt Lake City, Utah, 2001.

Z. Wang. (2000). Low complexity, high performance Turbo decoder design, Ph.D. dissertation, *University of Minnesota*, Aug. 2000.

Z. Wang. (2007). High-Speed Recursion Architectures for MAP-Based Turbo Decoders, in *IEEE Trans. on VLSI Syst.*, vol. 15, issue 4, pp: 470-74, Apr. 2007.

VLSI
Edited by Zhongfeng Wang

ISBN 978-953-307-049-0
Hard cover, 456 pages
Publisher InTech
Published online 01, February, 2010
Published in print edition February, 2010

The process of Integrated Circuits (IC) started its era of VLSI (Very Large Scale Integration) in 1970’s when thousands of transistors were integrated into one single chip. Nowadays we are able to integrate more than a billion transistors on a single chip. However, the term “VLSI” is still being used, though there was some effort to coin a new term ULSI (Ultra-Large Scale Integration) for fine distinctions many years ago. VLSI technology has brought tremendous benefits to our everyday life since its occurrence. VLSI circuits are used everywhere, real applications include microprocessors in a personal computer or workstation, chips in a graphic card, digital camera or camcorder, chips in a cell phone or a portable computing device, and embedded processors in an automobile, et al. VLSI covers many phases of design and fabrication of integrated circuits. For a commercial chip design, it involves system definition, VLSI architecture design and optimization, RTL (register transfer language) coding, (pre- and post-synthesis) simulation and verification, synthesis, place and route, timing analyses and timing closure, and multi-step semiconductor device fabrication including wafer processing, die preparation, IC packaging and testing, et al. As the process technology scales down, hundreds or even thousands of millions of transistors are integrated into one single chip. Hence, more and more complicated systems can be integrated into a single chip, the so-called System-on-chip (SoC), which brings to VLSI engineers ever increasingly challenges to master techniques in various phases of VLSI design. For modern SoC design, practical applications are usually speed hungry. For instance, Ethernet standard has evolved from 10Mbps to 10Gbps. Now the specification for 100Mbps Ethernet is on the way. On the other hand, with the popularity of wireless and portable computing devices, low power consumption has become extremely critical. To meet these contradicting requirements, VLSI designers have to perform optimizations at all levels of design. This book is intended to cover a wide range of VLSI design topics. The book can be roughly partitioned into four parts. Part I is mainly focused on algorithmic level and architectural level VLSI design and optimization for image and video signal processing systems. Part II addresses VLSI design optimizations for cryptography and error correction coding. Part III discusses general SoC design techniques as well as other application-specific VLSI design optimizations. The last part will cover generic nano-scale circuit-level design techniques.