J Sign Process Syst
DOI 10.1007/s11265-010-0477-6

A Flexible LDPC/Turbo Decoder Architecture

Yang Sun · Joseph R. Cavallaro

Received: 21 November 2009 / Revised: 11 March 2010 / Accepted: 12 March 2010

Abstract Low-density parity-check (LDPC) codes and convolutional Turbo codes are two of the most powerful error correcting codes that are widely used in modern communication systems. In a multi-mode baseband receiver, both LDPC and Turbo decoders may be required. However, the different decoding approaches for LDPC and Turbo codes usually lead to different hardware architectures. In this paper we propose a unified message passing algorithm for LDPC and Turbo codes and introduce a flexible soft-input soft-output (SISO) module to handle LDPC/Turbo decoding. We employ the trellis-based maximum *a posteriori* (MAP) algorithm as a bridge between LDPC and Turbo codes decoding. We view the LDPC code as a concatenation of *n* super-codes where each super-code has a simpler trellis structure so that the MAP algorithm can be easily applied to it. We propose a flexible functional unit (FFU) for MAP processing of LDPC and Turbo codes with a low hardware overhead (about 15% area and timing overhead). Based on the FFU, we propose an area-efficient flexible SISO decoder architecture to support LDPC/Turbo codes decoding. Multiple such SISO modules can be embedded into a parallel decoder for higher decoding throughput. As a case study, a flexible LDPC/Turbo decoder has been synthesized on a TSMC 90 nm CMOS technology with a core area of 3.2 mm^2. The decoder can support IEEE 802.16e LDPC codes, IEEE 802.11n LDPC codes, and 3GPP LTE Turbo codes. Running at 500 MHz clock frequency, the decoder can sustain up to 600 Mbps LDPC decoding or 450 Mbps Turbo decoding.

Keywords SISO decoder · LDPC decoder · Turbo decoder · Error correcting codes · MAP algorithm · Reconfigurable architecture

Y. Sun (✉) · J. R. Cavallaro
Department of Electrical and Computer Engineering Rice University, 6100 Main Street, Houston, TX 77005, USA
e-mail: ysun@rice.edu

J. R. Cavallaro
e-mail: cavallar@rice.edu

1 Introduction

Practical wireless communication channels are inherently "noisy" due to the impairments caused by channel distortions and multipath effect. Error correcting codes are widely used to increase the bandwidth and energy efficiency of wireless communication systems. As a core technology in wireless communications, forward error correction (FEC) coding has migrated from basic convolutional/block codes to more powerful Turbo codes and LDPC codes. Turbo codes, introduced by Berrou et al. in 1993 [4], have been employed in 3G and beyond 3G wireless systems, such as UMTS/WCDMA and 3GPP Long-Term Evolution (LTE) systems. As a candidate for 4G coding scheme, LDPC codes, which were introduced by Gallager in 1963 [13], have recently received significant attention in coding theory and have been adopted by some advanced wireless systems such as IEEE 802.16e WiMAX system and IEEE 802.11n WLAN system. In future 4G networks, internetworking and roaming between different networks would require a multi-standard FEC decoder. Since Turbo codes and LDPC codes are widely used in many different 3G/4G systems, it is important to design a configurable decoder to support multiple FEC coding schemes.

In the literature, many efficient LDPC decoder VLSI architectures have been studied [6, 9, 12, 14, 18, 24, 27, 29, 35, 37, 39, 45, 47]. Turbo decoder VLSI architectures have also been extensively investigated by many researchers [5, 8, 20, 21, 25, 30, 33, 41, 44]. However, designing a flexible decoder to support both LDPC and Turbo codes still remains very challenging. In this paper, we aim to provide an alternative to dedicated silicon that reduces the cost of supporting both LDPC and Turbo codes with a small additional overhead. We propose a flexible decoder architecture to meet the needs of a multi-standard FEC decoder.

From the theoretical point of view, there are some similarities between LDPC and Turbo codes. They can both be represented as codes on graphs which define the constraints satisfied by codewords. Both families of codes are decoded in an iterative manner by employing the sum-product algorithm or belief propagation algorithm. For example, MacKay has related these two codes by treating a Turbo code as a low-density parity-check code [23]. On the other hand, a few other researchers have tried to treat a LDPC code as a Turbo code and apply a turbo-like message passing algorithm to LDPC codes. For example, Mansour and Shanbhag [24] introduce an efficient turbo message passing algorithm for architecture-aware LDPC codes. Hocevar [18] proposes a layered decoding algorithm which treats the parity check matrix as horizontal layers and passes the soft information between layers to improve the performance. Zhu and Chakrabarti [50] looked at the super-code based LDPC construction and decoding. Zhang and Fossorier [46] suggest a shuffled belief propagation algorithm to achieve a faster decoding speed. Lu and Moura [22] propose to partition the Tanner graph into several trees and apply the turbo-like decoding algorithm in each tree for faster convergence rate. Dai et al. [12] introduce a turbo-sum-product hybrid decoding algorithm for quasi-cyclic (QC) LDPC codes by splitting the parity check matrix into two sub-matrices where the information is exchanged.

In our early work [38], we have proposed a super-code based decoding algorithm for LDPC codes. In this paper, we extend this algorithm and present a more generic message passing algorithm for LDPC and Turbo decodings, and then exploit the architecture commonalities between LDPC and Turbo decoders. We create a connection between LDPC and Turbo codes by applying a super-code based decoding algorithm, where a code is divided into multiple super-codes and then the decoding operation is performed by iteratively exchanging the soft information between super-codes. In the LDPC decoding, we treat a LDPC code as a concatenation of n super-codes, where each super-code has a simpler trellis structure so that the maximum *a posteriori* (MAP) algorithm can be efficiently performed. In the Turbo decoding, we modify the traditional message passing flow so that the proposed super-code based decoding scheme works for Turbo codes as well.

Contributions of this paper are as follows. First, we introduce a flexible soft-input soft-output (Flex-SISO) module for LDPC and Turbo codes decoding. Second, we introduce an area-efficient flexible functional unit (FFU) for implementing the MAP algorithm in hardware. Third, we propose a flexible SISO decoder hardware architecture based on the FFU. Finally, we show how to enable parallel decoding by using multiple such Flex-SISO decoders.

The remainder of the paper is organized as follows. Section 2 reviews the super-code based decoding algorithm for LDPC codes. Section 3 presents a Flex-SISO module for LDPC/Turbo decoding. Section 4 introduces a flexible functional unit (FFU) for LDPC and Turbo decoding. Based on the FFU, Section 5 describes a dual-mode Flex-SISO decoder architecture. Section 6 presents a parallel decoder architecture using multiple Flex-SISO cores. Section 7 compares our flexible decoder with existing decoders in the literature. Finally, Section 8 concludes the paper.

2 Review of Super-code Based Decoding Algorithm for LDPC Codes

By definition, a Turbo code is a parallel concatenation of two super-codes, where each super-code is a constituent convolutional code. Naturally, Turbo decoding procedure can be partitioned into two phases where each phase corresponds to one super-code processing. Similarly, LDPC codes can also be partitioned into super-codes for efficient processing as previously mentioned in Section 1. Before proceeding with a discussion of the proposed flexible decoder architecture, it is desirable to review the super-code based LDPC decoding scheme in this section.

2.1 Trellis Structure for LDPC Codes

A binary LDPC code is a linear block code specified by a very sparse binary $M \times N$ parity check matrix:

$$\mathbf{H} \cdot \mathbf{x}^T = \mathbf{0}, \quad (1)$$

where $\mathbf{x}$ is a codeword ($\mathbf{x} \in C$) and $\mathbf{H}$ can be viewed as a bipartite graph where each column and row in $\mathbf{H}$ represent a variable node and a check node, respectively. Each element of the parity check matrix is

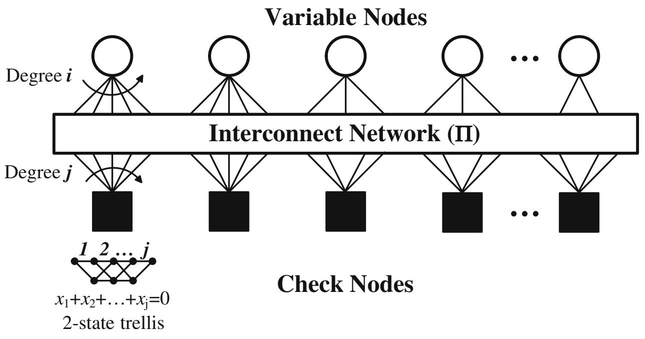


Figure 1 Trellis representation for LDPC codes where a two-state trellis diagram is associated with each check node.

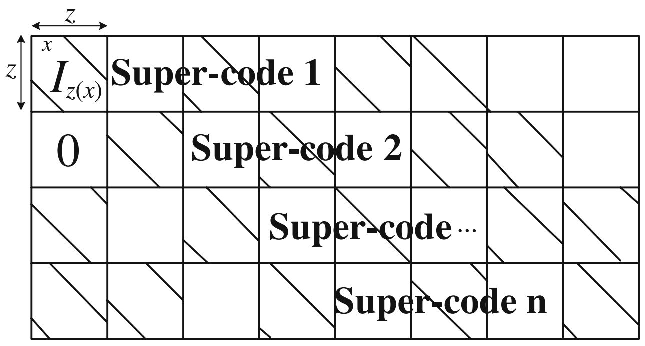


Figure 3 A block-structured parity check matrix, where each block row (or layer) defines a super-code. Each sub-matrix of the parity check matrix is either a zero matrix or a $z \times z$ cyclically shifted identity matrix.

either a zero or a one, where nonzero elements are typically placed at random positions to achieve good performance. The number of nonzero elements in each row or each column of the parity check matrix is called check node degree or variable node degree. A regular LDPC code has the same check node and variable node degrees, whereas an irregular LDPC code has different check node and variable node degrees.

The full trellis structure of an LDPC code is enormously large, and it is impractical to apply the MAP algorithm on the full trellis. However, alternately, a (N, M-N) LDPC code can be viewed as M parallel concatenated single parity check codes. Although the performance of a single parity check code is poor, when many of them are sparsely connected they become a very strong code. Figure 1 shows a trellis representation for LDPC codes where a single parity check code is considered as a low-weight two-state trellis, starting at state 0 and ending at state 0.

2.2 Layered Message Passing Algorithm for LDPC Codes

The main idea behind the layered LDPC decoding is essentially the Turbo message passing algorithm [24]. It has been shown that the layered message passing algorithm can achieve a faster convergence rate than the standard two-phase message-passing algorithm for structured LDPC codes [18, 24]. To be more general, we can divide the factor graph of an LDPC code into several sub-graphs [38] as illustrated in Fig. 2. Each sub-graph corresponds to a super-code. If we restrict that each sub-graph is loop-free, then each super-code has a simpler trellis structure so that the MAP algorithm can be efficiently performed.

As a special example, the block-structured Quasi-Cyclic (QC) LDPC codes used in many practical communication systems such as 802.16e and 802.11n can be easily decomposed into several super-codes. As shown in Fig. 3, a block structured parity check matrix can be viewed as a 2-D array of square sub-matrices. Each sub-matrix is either a zero matrix or a z-by-z cyclically shifted identity matrix $I_{z(x)}$ with random shift value x. The parity check matrix can be viewed as a concatenation of n super-codes where each block row or layer defines a super-code. In the layered message passing algorithm, soft information generated by one super-code can be used immediately by the following super-codes which leads to a faster convergence rate [24].

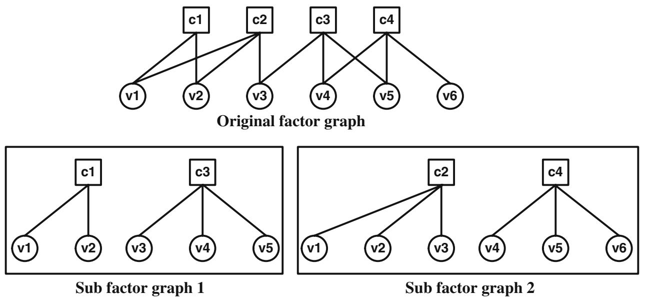
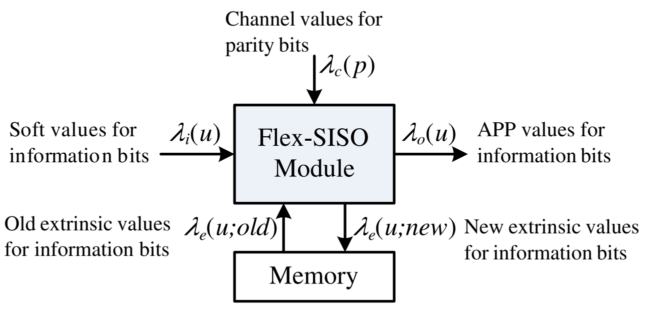


Figure 2 Dividing a factor graph into sub-graphs.

3 Flexible SISO Module

In this section, we propose a flexible soft-input soft-output (SISO) module, named Flex-SISO module, to decode LDPC and Turbo codes. The SISO module is based on the MAP algorithm [3]. To reduce complexity, the MAP algorithm is usually calculated in the log domain [31]. In this paper, we assume the MAP algorithm is always calculated in the log domain.

The decoding algorithm underlying the Flex-SISO module works for codes which have trellis representations. For LDPC codes, a Flex-SISO module was used



Channel values for parity bits
$\lambda_c(p)$
Soft values for information bits
$\lambda_i(u)$
Flex-SISO Module
$\lambda_o(u)$
APP values for information bits
Old extrinsic values for information bits
$\lambda_e(u;old)$
$\lambda_e(u;new)$
New extrinsic values for information bits
Memory

Figure 4 Flex-SISO module.

to decode a super-code. For Turbo codes, a Flex-SISO module was used to decode a component convolutional code. Iteration performed by the Flex-SISO module is called sub-iteration, and thus one full iteration contains n sub-iterations.

3.1 Flex-SISO Module

Figure 4 depicts the proposed Flex-SISO module. The output of the Flex-SISO module is the *a posteriori* probability (APP) log-likelihood ratio (LLR) values, denoted as $\lambda_o(u)$, for information bits. It should be noted that the Flex-SISO module exchanges the soft values $\lambda_o(u)$ instead of the extrinsic values in the iterative decoding process. The extrinsic values, denoted as $\lambda_e(u)$, are stored in a local memory of the Flex-SISO module. To distinguish the extrinsic values generated at different sub-iterations, we use $\lambda_e(u; old)$ and $\lambda_e(u; new)$ to represent the extrinsic values generated in the previous sub-iteration and the current sub-iteration, respectively. The soft input values $\lambda_i(u)$ are the outputs from the previous Flex-SISO module, or other previous modules if necessary. Another input to the Flex-SISO module is the channel values for parity bits, denoted as $\lambda_c(p)$, if available. For LDPC codes, we do not distinguish information and parity bits, and all the codeword bits are treated as information bits. However, in the case of Turbo codes, we treat information and parity bits separately. Thus the input port $\lambda_c(p)$ will not be used when decoding of LDPC codes. At each sub-iteration, the old extrinsic values, denoted as $\lambda_e(u; old)$, are retrieved from the local memory and should be subtracted from the soft input values $\lambda_i(u)$ to avoid positive feedback.

A generic description of the message passing algorithm is as follows. Multiple Flex-SISO modules are connected in series to form an iterative decoder. First, the Flex-SISO module receives the soft values $\lambda_i(u)$ from upstream Flex-SISO modules and the channel values (for parity bits) $\lambda_c(p)$ if available. The $\lambda_i(u)$ can be thought of as the sum of the channel value $\lambda_c(u)$ (for information bit) and all the extrinsic values $\lambda_e(u)$ previously generated by all the super-codes:

$$\lambda_i(u) = \lambda_c(u) + \sum \lambda_e(u). \qquad (2)$$

Note that prior to the iterative decoding, $\lambda_i(u)$ should be initialized with $\lambda_c(u)$. Next, the old extrinsic value $\lambda_e(u; old)$ generated by this Flex-SISO module in the previous iteration is subtracted from $\lambda_i(u)$ as follows:

$$\lambda_t(u) = \lambda_i(u) - \lambda_e(u; old). \qquad (3)$$

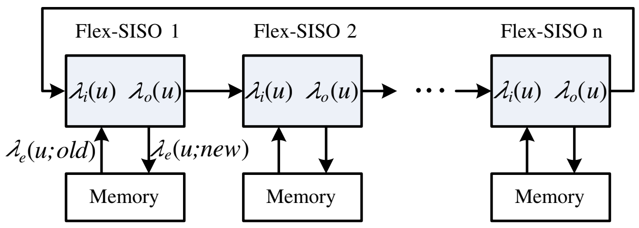
Then, the new extrinsic value $\lambda_e(u; new)$ can be computed using the MAP algorithm based on $\lambda_t(u)$, and $\lambda_c(p)$ if available. Finally, the APP value is updated as

$$\lambda_o(u) = \lambda_i(u) - \lambda_e(u; old) + \lambda_e(u; new). \qquad (4)$$

Then this updated APP value is passed to the downstream Flex-SISO modules. This computation repeats in each sub-iteration.

3.2 Flex-SISO Module to Decode LDPC Codes

In this section, we show how to use the Flex-SISO module to decode LDPC codes. Because QC-LDPC codes are widely used in many practical systems, we will primarily focus on the QC-LDPC codes. First, we decompose a QC-LDPC code into multiple super-codes, where each layer of the parity check matrix defines a super-code. After the layered decomposition, each super-code comprises z independent two-state single parity check codes. Figure 5 shows the super-code based, or layered, LDPC decoder architecture using the Flex-SISO modules. The decoder parallelism at each Flex-SISO module is at the level of the sub-matrix size z, because these z single parity codes have no data dependency and can thus be processed simultaneously. This architecture differs than the regular two-phase LDPC decoder in that a code is partitioned into multiple sections, and each section is processed by a same processor. The convergence rate can be twice faster than that of a regular decoder [18].



Flex-SISO 1
Flex-SISO 2
Flex-SISO n
$\lambda_i(u)$ $\lambda_o(u)$
$\lambda_i(u)$ $\lambda_o(u)$
$\lambda_i(u)$ $\lambda_o(u)$
$\lambda_e(u;old)$
$\lambda_e(u;new)$
Memory
Memory
Memory

Figure 5 LDPC decoding using Flex-SISO modules where a LDPC code is decomposed into n super-codes, and n Flex-SISO modules are connected in series to decode.

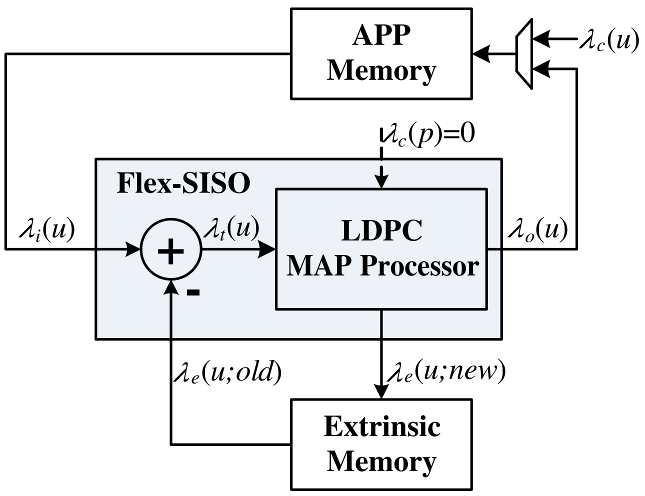


Figure 6 LDPC decoder architecture based on the Flex-SISO module.

Since the data flow is the same between different sub-iterations, one physical Flex-SISO module is instantiated, and it is re-used at each sub-iteration, which leads to a partial-parallel decoder architecture. Figure 6 shows an iterative LDPC decoder hardware architecture based on the Flex-SISO module. The structure comprises an APP memory to store the soft APP values, an extrinsic memory to store the extrinsic values, and a MAP processor to implement the MAP algorithm for z single parity check codes. Prior to the iterative decoding process, the APP memory is initialized with channel values $\lambda_c(u)$, and the extrinsic memory is initialized with 0.

The decoding flow is summarized as follows. It should be noted that the parity bits are treated as information bits for the decoding of LDPC codes. We use the symbol u_k to represent the k-th data bit in the codeword. For check node m, we use the symbol $u_{m,k}$ to denote the k-th codeword bit (or variable node) that is connected to this check node m. To remove correlations between iterations, the old extrinsic message is subtracted from the soft input message to create a temporary message λ_t as follows

$$\lambda_t(u_{m,k}) = \lambda_i(u_k) - \lambda_e(u_{m,k}; old), \tag{5}$$

where $\lambda_i(u_k)$ is the soft input log likelihood ratio (LLR) and $\lambda_e(u_{m,k}; old)$ is the old extrinsic value generated by this MAP processor in the previous iteration. Then the new extrinsic value can be computed as:

$$\lambda_e(u_{m,k}; new) = \sum_{j: j \neq k}\boxplus \lambda_t(u_{m,j}), \tag{6}$$

where the $\boxplus$ operation is associative and commutative, and is defined as [15]

$$\lambda(u_1) \boxplus \lambda(u_2) = \log \frac{1 + e^{\lambda(u_1)} e^{\lambda(u_2)}}{e^{\lambda(u_1)} + e^{\lambda(u_2)}}. \tag{7}$$

Finally, the new APP value is updated as:

$$\lambda_o(u_k) = \lambda_t(u_{m,k}) + \lambda_e(u_{m,k}; new). \tag{8}$$

For each sub-iteration l, Eqs. (5)–(8) can be executed in parallel for check nodes $m = lz$ to $lz + z - 1$ because there are no data dependency between them.

3.3 Flex-SISO Module to Decode Turbo Codes

In this section, we show how to use the Flex-SISO module to decode Turbo codes. A Turbo code can be naturally partitioned into two super-codes, or constituent codes. In a traditional Turbo decoder, where the extrinsic messages are exchanged between two super-codes, the Flex-SISO module can not be directly applied, because the Flex-SISO module requires the APP values, rather than the extrinsic values, being exchanged between super-codes. In this section, we made a small modification to the traditional Turbo decoding flow so that the APP values are exchanged in the decoding procedure.

3.3.1 Review of the Traditional Turbo Decoder Structure

The traditional Turbo decoding procedure with two SISO decoders is shown in Fig. 7. The definitions of the symbols in the figure are as follows. The information bit and the parity bits at time k are denoted as u_k and $(p_k^{(1)}, p_k^{(2)}, ..., p_k^{(n)})$, respectively, with $u_k, p_k^{(i)} \in \{0, 1\}$. The channel LLR values for u_k and $p_k^{(i)}$ are denoted as $\lambda_c(u_k)$ and $\lambda_c(p_k^{(i)})$, respectively. The *a priori* LLR, the extrinsic LLR, and the APP LLR for u_k are denoted as $\lambda_a(u_k)$, $\lambda_e(u_k)$, and $\lambda_o(u_k)$, respectively.

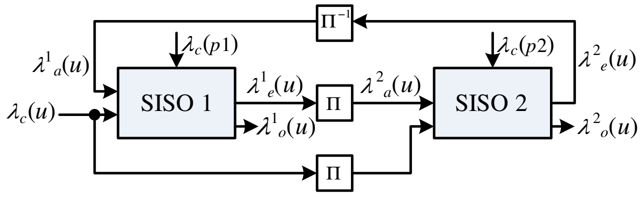


Figure 7 Traditional Turbo decoding procedure using two SISO decoders, where the *extrinsic* LLR values are exchanged between two SISO decoders.

In the decoding process, the SISO decoder computes the extrinsic LLR value at time k as follows:

$$\lambda_e(u_k) = \max^*_{\mathbf{u}:u_k=1}\{\alpha_{k-1}(s_{k-1}) + \gamma_k^e(s_{k-1}, s_k) + \beta_k(s_k)\} \\ - \max^*_{\mathbf{u}:u_k=0}\{\alpha_{k-1}(s_{k-1}) + \gamma_k^e(s_{k-1}, s_k) + \beta_k(s_k)\}. \tag{9}$$

The α and β metrics are computed based on the forward and backward recursions:

$$\alpha_k(s_k) = \max^*_{s_{k-1}}\{\alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k)\} \tag{10}$$

$$\beta_k(s_k) = \max^*_{s_{k+1}}\{\beta_{k+1}(s_{k+1}) + \gamma_k(s_k, s_{k+1})\}, \tag{11}$$

where the branch metric γ_k is computed as:

$$\gamma_k = u_k \cdot (\lambda_c(u_k) + \lambda_a(u_k)) + \sum_i^n p_k^{(i)} \cdot \lambda_c(p_k^{(i)}). \tag{12}$$

The *extrinsic* branch metric γ_k^e in Eq. 9 is computed as:

$$\gamma_k^e = \sum_i^n p_k^{(i)} \cdot \lambda_c(p_k^{(i)}). \tag{13}$$

The max$^*(\cdot)$ function in Eqs. 9–11 is defined as:

$$\max^*(a, b) = \max(a, b) + \log(1 + e^{-|a-b|}). \tag{14}$$

The soft APP value for u_k is generated as:

$$\lambda_o(u_k) = \lambda_e(u_k) + \lambda_a(u_k) + \lambda_c(u_k). \tag{15}$$

In the first half iteration, SISO decoder 1 computes the extrinsic value $\lambda_e^1(u_k)$ and pass it to SISO decoder 2. Thus, the extrinsic value computed by SISO decoder 1 becomes the *a priori* value $\lambda_a^2(u_k)$ for SISO decoder 2 in the second half iteration. The computation is repeated in each iteration. The iterative process is usually terminated after certain number of iterations, when the soft APP value $\lambda_o(u_k)$ converges.

3.3.2 Modified Turbo Decoder Structure Using Flex-SISO Modules

In order to use the proposed Flex-SISO module for Turbo decoding, we modify the traditional Turbo decoder structure. Figure 8 shows the modified Turbo decoder structure based on the Flex-SISO modules.

It should be noted that the modified Turbo decoding flow is mathematically equivalent to the original Turbo decoding flow, but uses a different message passing method. The modified data flow is as follows. In the first half iteration, Flex-SISO decoder 1 receives soft LLR value $\lambda_i^1(u_k)$ from Flex-SISO decoder 2 through de-interleaving ($\lambda_i^1(u_k)$ is initialized to channel value $\lambda_c(u_k)$ prior to decoding). Then it removes the old extrinsic value $\lambda_e^1(u_k; old)$ from the soft input LLR $\lambda_i^1(u_k)$ to form a temporary message $\lambda_t^1(u_k)$ as follows (for brevity, we drop the superscript "1" in the following equations)

$$\lambda_t(u_k) = \lambda_i(u_k) - \lambda_e(u_k; old). \tag{16}$$

To relate to the traditional Turbo decoder structure, this temporary message is mathematically equal to the sum of the channel value $\lambda_c(u_k)$ and the *a priori* value $\lambda_a(u_k)$ in Fig. 7:

$$\lambda_t(u_k) = \lambda_c(u_k) + \lambda_a(u_k). \tag{17}$$

Thus, the branch metric calculation in Eq. 12 can be rewritten as:

$$\gamma_k = u_k \cdot \lambda_t(u_k) + \sum_i^n p_k^{(i)} \cdot \lambda_c(p_k^{(i)}). \tag{18}$$

The extrinsic branch metric (γ_k^e) calculation, and the extrinsic LLR ($\lambda_e(u_k)$) calculation, however, remain the same as Eqs. 13 and 9–11, respectively. Finally, the soft APP LLR output is computed as:

$$\lambda_o(u_k) = \lambda_t(u_k) + \lambda_e(u_k; new). \tag{19}$$

In the Flex-SISO based iterative decoding procedure, the soft outputs $\lambda_o^1(u)$ computed by Flex-SISO decoder 1 are passed to Flex-SISO decoder 2 so that

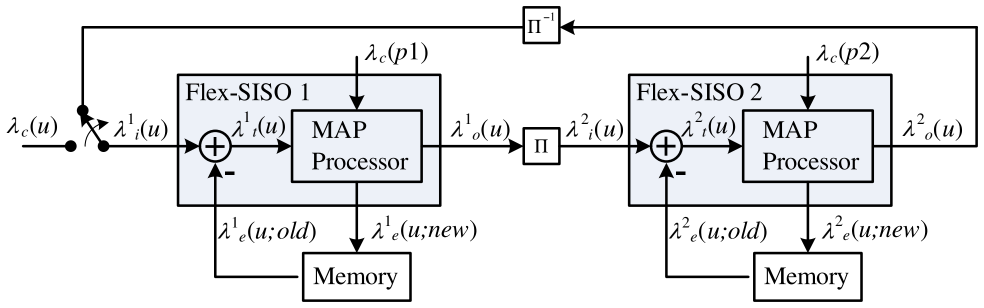


Figure 8 Modified Turbo decoding procedure using two Flex-SISO modules. The *soft* LLR values are exchanged between two SISO modules.

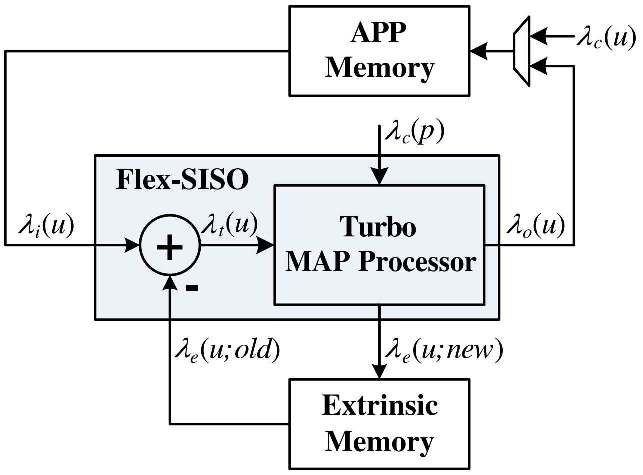


Figure 9 Turbo decoder architecture based on the Flex-SISO module.

they become the soft inputs $\lambda_i^2(u)$ for Flex-SISO decoder 2 in the second half iteration. The computation is repeated in each half-iteration until the iteration converges. Since the operations are identical between two sub-iterations, only one physical Flex-SISO module is instantiated, and it is re-used for two sub-iterations.

Figure 9 shows an iterative Turbo decoder architecture based on the Flex-SISO module. The architecture is very similar to the LDPC decoder architecture shown in Fig. 6. The main differences are: 1) the Turbo decoder has separate parity channel LLR inputs whereas the LDPC decoder treats parity bits as information bits, 2) the Turbo decoder employs the MAP algorithm on an N-state trellis whereas the LDPC decoder applies the MAP algorithm on z independent two-state trellises, and 3) the interleaver/permuter structures are different (not shown in the figures). But despite these differences, there are certain important commonalities. The message passing flows are the same. The memory organizations are similar, but with a variety of sizes depending on the codeword length. The MAP processors, which will be described in the next section, have similar functional unit resources that will be configured using multiplexors for each algorithm. Thus, it is natural to design a unified SISO decoder with configurable MAP processors to support both LDPC and Turbo codes.

4 Design of a Flexible Functional Unit

The MAP processor is the main processing unit in both LDPC and Turbo decoders as depicted in Fig. 6 and Fig. 9. In this section, we introduce a flexible functional

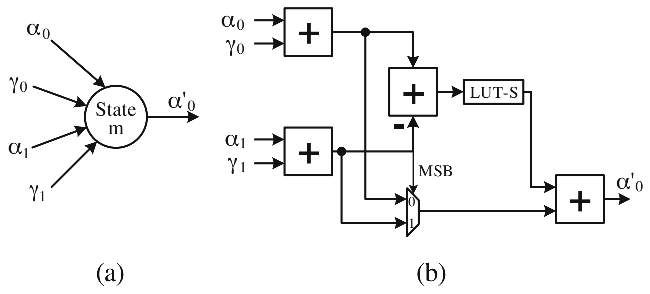


Figure 10 Turbo ACSA structure. **a** Flow of state metric calculation. **b** Circuit diagram for the Turbo ACSA unit.

unit to decode LDPC and Turbo codes with a small additional overhead.

4.1 MAP Functional Unit for Turbo Codes

In a Turbo MAP processor, the critical path lies in the state metric calculation unit which is often referred to as add-compare-select-add (ACSA) unit. As depicted in Fig. 10, for each state m of the trellis, the decoder needs to perform an ACSA operation as follows:

$$\alpha'_0 = \overset{*}{\max}(\alpha_0 + \gamma_0, \alpha_1 + \gamma_1), \tag{20}$$

where α_0 and α_1 are the previous state metrics, and γ_0 and γ_1 are the branch metrics. Figure 10b shows a circuit implementation for the ACSA unit, where a signed-input look-up table “LUT-S" was used to implement the non-linear function $\log(1 + e^{-|x|})$. This circuit can be used to recursively compute the forward and backward state metrics based on Eqs. 10 and 11.

4.2 MAP Functional Unit for LDPC Codes

In the layered QC-LDPC decoding algorithm, each super-code comprises z independent single parity check codes. Each single parity check code can be viewed as a terminated two-state convolutional code. Figure 11 shows an example of the trellis structure for a single parity check node.

An efficient MAP decoding algorithm for single parity check code was given in [16]: for independent

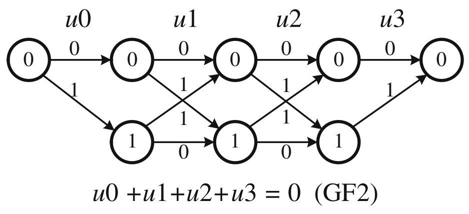


Figure 11 Trellis structure for a single parity check code.

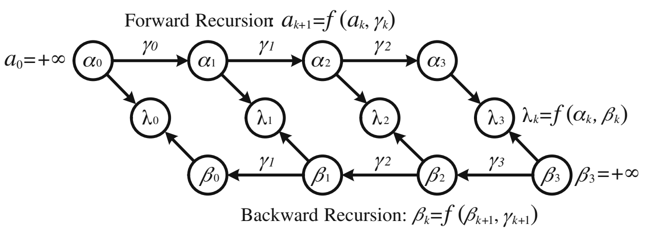


Figure 12 A forward–backward decoding flow to compute the extrinsic LLRs for single parity check code.

random variables $u_0, u_1, ..., u_l$ the extrinsic LLR value for bit u_k is computed as:

$$\lambda(u_k) = \sum_{\sim\{u_k\}} \boxplus \lambda_i(u_i), \tag{21}$$

where the compact notation $\sim\{u_k\}$ represents the set of all the variables with u_k excluded. For brevity, we define a function $f(a, b)$ to represent the operation $\lambda_i(u_1) \boxplus \lambda_i(u_2)$ as follows

$$f(a, b) = \log \frac{1 + e^a e^b}{e^a + e^b}, \tag{22}$$

where $a \triangleq \lambda_i(u_1)$ and $b \triangleq \lambda_i(u_2)$. Figure 12 shows a forward–backward decoding flow to implement Eq. 21. The forward (α) and backward (β) recursions are defined as:

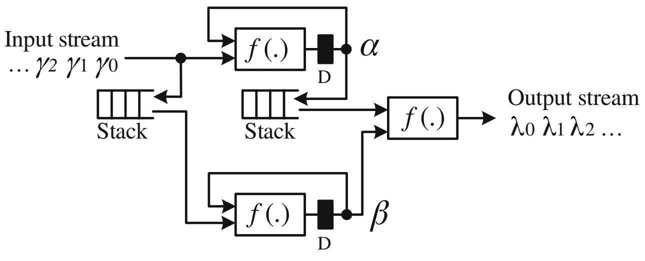
$$\alpha_{k+1} = f(\alpha_k, \gamma_k) \tag{23}$$

$$\beta_k = f(\beta_{k+1}, \gamma_{k+1}), \tag{24}$$

where $\gamma_k = \lambda_i(u_k)$ and is referred to as the branch metric as an analogy to a Turbo decoder. The α and β metrics are initialized to $+\infty$ in the beginning. Based on the α and β metrics, the extrinsic LLR for u_k is computed as:

$$\lambda(u_k) = f(\alpha_k, \beta_k). \tag{25}$$

Figure 13 shows a MAP processor structure to decode the single parity check code. Three identical $f(a, b)$ units are used to compute α, β, and λ values. To relate to the top level LDPC decoder architecture



Input stream
$\ldots \gamma_2\ \gamma_1\ \gamma_0$
$f(.)$
D
α
Stack
Stack
$f(.)$
Output stream
$\lambda_0\ \lambda_1\ \lambda_2\ \ldots$
$f(.)$
D
β

Figure 13 MAP processor structure for single parity check code.

Table 1 LUT approximation for $g(x) = \log(1 + e^{-|x|})$.

$\lvert x\rvert$	$\lvert x\rvert = 0$	$0 < \lvert x\rvert \le 0.75$	$0.75 < \lvert x\rvert \le 2$	$\lvert x\rvert > 2$
$g(x)$	0.75	0.5	0.25	0

as shown in Fig. 6, the inputs to this MAP processor are the temporary metrics $\lambda_t(u_{m,k})$, and the outputs from this MAP processor are the extrinsic metrics $\lambda_e(u_{m,k}; new)$.

To compute Eq. 22 in hardware, we separate the operation into sign and magnitude calculations:

$$\begin{aligned} \text{sign}(f(a, b)) &= \text{sign}(a)\,\text{sign}(b), \\ |f(a, b)| &= \min(|a|, |b|) + \log(1 + e^{-(|a|+|b|)}) \\ &\quad - \log\left(1 + e^{-\left||a|-|b|\right|}\right). \end{aligned} \tag{26}$$

Compared to the classical "tanh" function used in LDPC decoding $\Psi(x) = -\log(\tanh(|x/2|))$, the $f(\cdot)$ function is numerically more robust and less sensitive to quantization noise. Due to its widely dynamic range (up to $+\infty$), the $\Psi(x)$ function has a high complexity and is prone to quantization noise. Although many approximations have been proposed to improve the numerical accuracy of $\Psi(x)$ [26, 29, 48], it is still expensive to implement the $\Psi(x)$ function in hardware. However, the non-linear term in the $f(\cdot)$ function has a very small dynamic range:

$$0 < g(x) \triangleq \log(1 + e^{-|x|}) < 0.7,$$

thus the $f(\cdot)$ function is more easily to be implemented in hardware by using a low complexity look-up table (LUT). To implement $g(x)$ in hardware, we propose to use a four-value LUT approximation which is shown in Table 1. For fixed point implementation, we propose to use $Q.2$ quantization scheme (Q total bits with 2 fractional bits). Table 2 shows the proposed LUT implementation for $Q.2$ quantization. It should be noted that $g(x)$ is the same as the non-linear term in the Turbo max$^*(\cdot)$ function (c.f. Eq. 14). Thus, the same look-up table configuration can be applied to the Turbo ACSA unit. In Section 4.4, we will show the decoding performance by using this look-up table.

Figure 14 depicts a circuit implementation for the LDPC $|f(a, b)|$ functional unit using two look-up tables "LUT-S" and "LUT-U", where LUT-S and LUT-U implement $\log(1 + e^{-\left||a|-|b|\right|})$ and $\log(1 + e^{-(|a|+|b|)})$,

Table 2 LUT implementation for $Q.2$ quantization.

$\lvert x\rvert$	0	1	2	3	4	5	6	7	8	> 8
$g(x)$	3	2	2	2	1	1	1	1	1	0

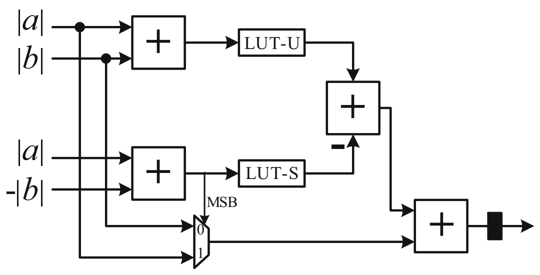


Figure 14 Circuit diagram for the LDPC $|f(a,b)|$ functional unit.

respectively. The difference between LUT-S and LUT-U is that: LUT-S is a signed-input look-up table that takes both positive and negative data inputs whereas LUT-U is an unsigned-input look-up table (half size of LUT-S) that only takes positive data inputs.

4.3 Proposed Flexible Functional Unit (FFU)

If we compare the LDPC $|f(a,b)|$ functional unit (c.f. Fig. 14) with the Turbo ACSA functional unit (c.f. Fig. 10), we can see that they have many commonalities except for the position of the look-up tables and the multiplexor. To support both LDPC and Turbo codes with minimum hardware overhead, we propose a flexible functional unit (FFU) which is depicted in Fig. 15. We modify the look-up table structure so that each look-up table can be bypassed when the *bypass* control signal is high. A *select* signal was used to switch between the LDPC mode and the Turbo mode. The functionality of the proposed FFU architecture is summarized in Table 3.

The word lengths for X, Y, V, and W are all 9 bits. To evaluate the area efficiency of the proposed FFU, we have described the LDPC $f(a,b)$ unit, the Turbo ACSA unit, and the proposed FFU in Verilog HDL, and synthesized them on a TSMC 90 nm CMOS technology. The maximum achievable frequency (assuming no clock skews) and the synthesized area at two frequencies (400 and 800 MHz) are summarized in Table 4. As can be seen, the proposed flexible functional unit FFU has only about 15% area and timing overhead compared to the dedicated functional units. The area efficiency is achieved because many logic gates can be shared between LDPC and Turbo modes.

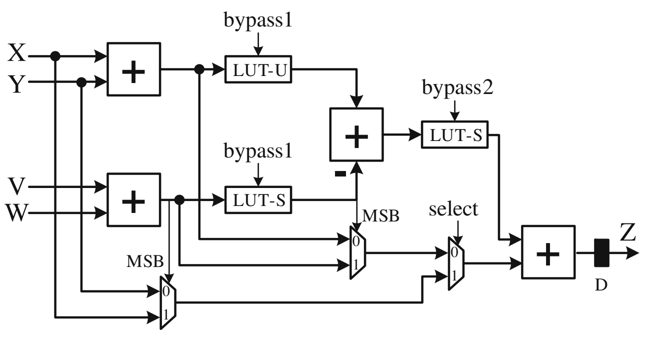


Figure 15 Circuit diagram for the flexible functional unit (FFU) for LDPC/Turbo decoding.

Table 3 Functional description of the FFU.

Signals	LDPC Mode	Turbo Mode
select	1	0
bypass1	0	1
bypass2	1	0
X	$\|a\|$	α_0
Y	$\|b\|$	γ_0
V	$\|a\|$	α_1
W	$-\|b\|$	γ_1
Z	$\|f(a,b)\|$	$\max^*(\alpha_0+\gamma_0, \alpha_1+\gamma_1)$

4.4 Fixed Point Decoding Performance

To evaluate the fixed-point decoding performance using the look-up table based FFU, we perform float-point and bit-accurate fixed-point simulations for LDPC and Turbo codes using BPSK modulation over an AWGN channel. As a good trade-off between complexity and performance, we use 6.2 quantization scheme for channel LLR inputs for fixed-point LDPC and Turbo decoders.

Figure 16 shows the bit error rate (BER) simulation result for a WiMAX LDPC code with code-rate = 1/2, and code-length = 2,304. The maximum number of iterations is 15. As can be seen from Fig. 16, the fixed-point FFU solution has a very small performance degradation (< 0.05 dB) at BER level of 10^{-6} compared to the floating point solution. We also plot a BER curve for the scaled minsum solution [11], which is a sub-optimal approximation algorithm without using the look-up tables. As can be seen from the figure, the look-up table based FFU solution can deliver a better decoding performance than the scaled minsum solution. The complexity of adding the look-up tables is relatively small because the word length of the data in

Table 4 Synthesis results for different functional units.

Functional unit	$\|f(a,b)\|$	ACSA	FFU
Max frequency	920 MHz	885 MHz	815 MHz
Area (400 MHz)	1,192 μm^2	1,263 μm^2	1,419 μm^2
Area (800 MHz)	1,882 μm^2	2,086 μm^2	2,423 μm^2

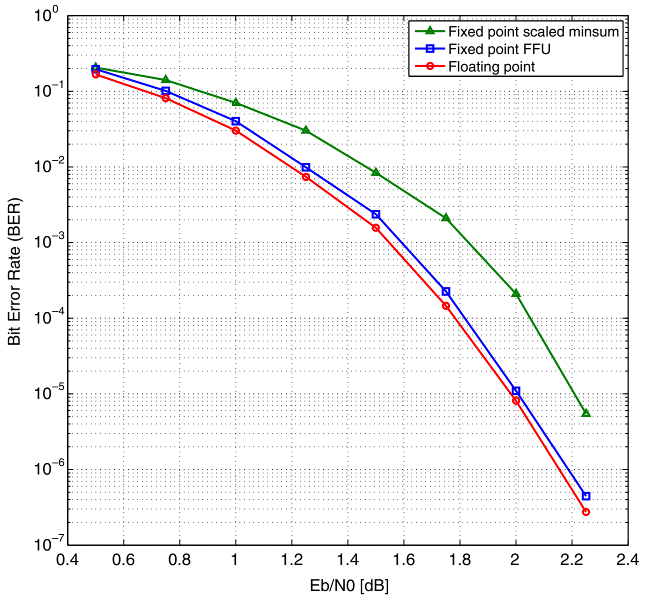


Figure 16 Simulation results for a rate 1/2, length 2304 WiMAX LDPC code.

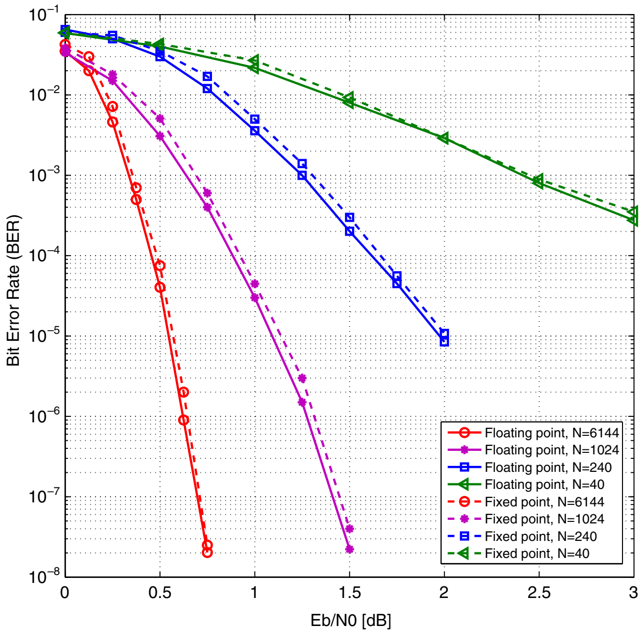


Figure 18 Simulation results for 3GPP-LTE Turbo codes with a variety of block sizes.

the look-up table is only 2-bit. Figure 17 compares the convergence speed of the layered decoding algorithm with the standard two-phase decoding algorithm.

Figure 18 shows the BER simulation result for 3GPP-LTE Turbo codes with block sizes of 6,144, 1,024, 240, and 40. The maximum number of Turbo iterations is 6 (12 half iterations). The sliding window length is 32. As can be seen from the figure, the FFU based fixed-point decoder has almost no performance loss compared to the floating point case. The proposed FFU solution will deliver a better decoding performance than the sub-optimal max-logMAP solution.

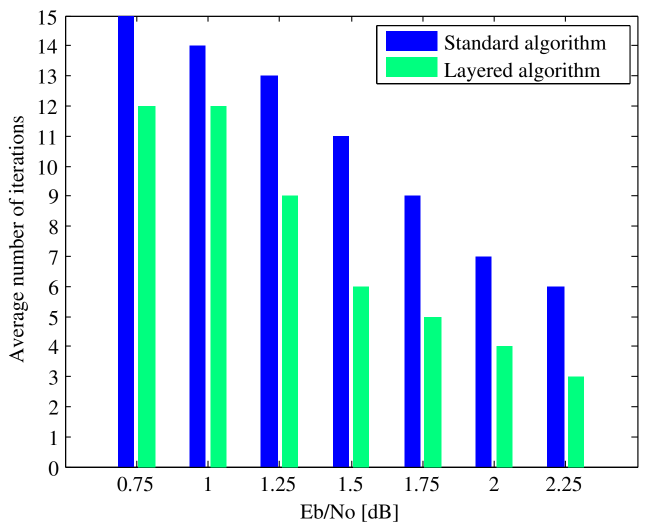


Figure 17 Comparison of the convergence speed.

From these simulation results, we conclude that the proposed look-up table based FFU is a good solution for supporting high performance LDPC and Turbo decoding requirements.

5 Design of A Flexible SISO Decoder

Built on top of the FFU arithmetic unit, we introduce a flexible SISO decoder architecture to handle LDPC and Turbo codes. Figure 19 illustrates the proposed dual-mode SISO decoder architecture. The decoder comprises four major functional units: alpha unit (α), beta unit (β), extrinsic-1 unit, and extrinsic-2 unit. The decoder can be reconfigured to process: i) an eight-state convolutional Turbo code, or ii) 8 single parity check codes.

5.1 Turbo Mode

In the Turbo mode, all the elements in the Flex-SISO decoder will be activated. For Turbo decoding, we use the *Next Iteration Initialization* (NII) sliding window algorithm as suggested in [1, 19]. The NII approach can avoid the calculation of training sequences as initialization values for the β state metrics, instead the boundary metrics are initialized from the previous iter-

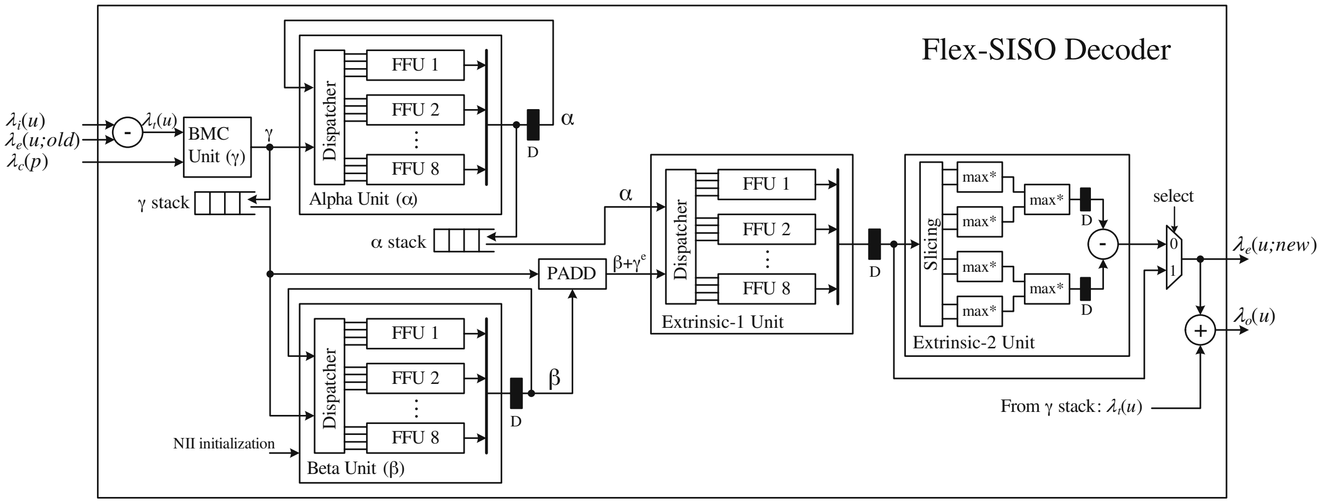


Figure 19 Flexible SISO decoder architecture.

ation. As a result, the decoding latency is smaller than the traditional sliding window algorithm which requires a calculation of training sequences [25, 43], and thus only one β unit is required. Moreover, this solution is very suitable for high code-rate Turbo codes, which require a very long training sequence to obtain reliable boundary state metrics. Note that this scheme would require an additional memory to store the boundary state metrics.

A dataflow graph for NII sliding window algorithm is depicted in Fig. 20, where the X-axis represents the trellis flow and the Y-axis represents the decoding time so that a box may represent the processing of a block of L data in L time steps, where L is the sliding window size. In the decoding process, the α metrics are computed in the natural order whereas the β metrics and the extrinsic LLR (λ_e) are computed in the reverse

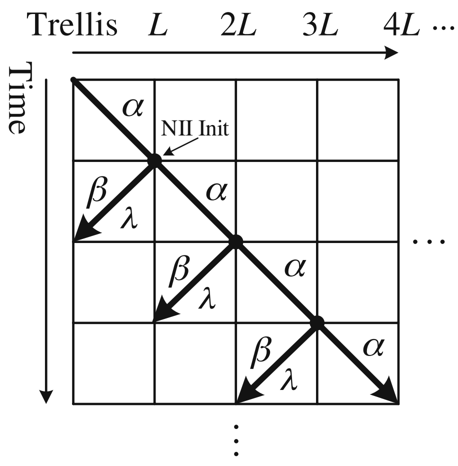


Figure 20 Data flow graph for Turbo decoding.

order. By using multiple FFUs, the α and β units are able to compute the state metrics in parallel, leading to a real time decoding with a latency of L.

The decoder works as follows. The decoder uses soft LLR value $\lambda_i(u)$ and old extrinsic value $\lambda_e(u; old)$ to compute $\lambda_t(u)$ based on Eq. 16. A branch metric calculation (BMC) unit is used to compute the branch metrics $\gamma(u, p)$ based on Eq. 18, where $u, p \in \{0, 1\}$. Then the branch metrics are buffered in a γ stack for backward (β) metric calculation. The α and β metrics are computed using Eqs. 10 and 11. The boundary β metrics are initialized from an NII buffer (not shown in Fig. 19). A dispatcher unit is used to dispatch the data to the correct FFUs in the α/β unit. Each α/β unit has fully-parallel FFUs (eight of them), so the eight-state convolutional trellis can be processed at a rate of one-stage per clock cycle.

To compute the extrinsic LLR as defined in Eq. 9, we first add β metrics with the extrinsic branch metrics $\gamma^e(p)$, where $\gamma^e(p)$ is retrieved from the γ stack, as $\gamma^e(0) = 0$, $\gamma^e(1) = \gamma(0, 1) = \lambda_c(p)$. The extrinsic LLR calculation is separated into two phases which is shown in the right part of Fig. 19. In phase 1, the extrinsic-1 unit performs eight ACSA operations in parallel using eight FFUs. In phase 2, the extrinsic-2 unit performs 6 max*(a, b) operations and 1 subtraction. Finally, the soft LLR $\lambda_o(u)$ is obtained by adding $\lambda_e(u; new)$ with $\lambda_t(u)$, where $\lambda_t(u)$ is also retrieved from the γ stack, as $\lambda_t(u) = \gamma(1, 0)$.

5.2 LDPC Mode

In the LDPC mode, a substantial subset (more than 90%) of the logic gates will be reused from the Turbo

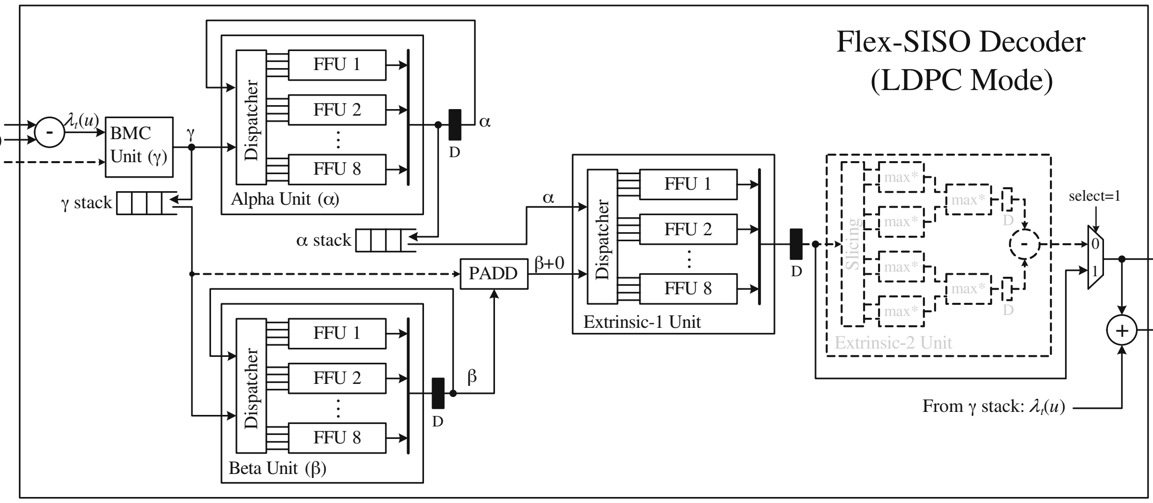


Figure 21 Flexible SISO decoder architecture in LDPC mode.

mode. As shown in Fig. 21, three major functional units (α unit, β unit, and the extrinsic-1 unit) and two stack memories are reused in the LDPC mode. The extrinsic-2 unit will be de-activated in the LDPC mode. The decoder can process 8 single parity check codes in parallel because each of the α unit, β unit, and extrinsic-1 unit has eight parallel FFUs.

The dataflow graph of the LDPC decoding (c.f. Fig. 12) is very similar to that of the Turbo decoding (c.f. Fig. 20). The decoder works as follows. The decoder first computes $\lambda_t(u)$ based on Eq. 5. In the LDPC mode, the branch metric γ is equal to $\lambda_t(u)$. Prior to decoding, the α and β metrics are initialized to the maximum value. Assuming the check node degree is L. In the first L cycles, the α unit recursively computes the α metrics in the forward direction and store them in an α stack. In the next L cycles, the β unit recursively computes the β metrics in the backward direction. At the same time, the extrinsic-1 unit computes the extrinsic LLRs using the α and β metrics. While the β unit and the extrinsic-1 unit are working on the first data stream, the α unit can work on the second stream which leads to a pipelined implementation.

Table 5 Flex-SISO decoder area distribution.

Unit	Area (mm^2)
α-unit	0.014
β-unit	0.014
Extrinsic-1 unit	0.014
Extrinsic-2 unit	0.004
α and γ stack memories	0.045
Control logic & others	0.007
Total	0.098

5.3 Performance

The proposed Flex-SISO decoder has been synthesized on a TSMC 90 nm CMOS technology. Table 5 summarizes the area distribution of this decoder. The maximum clock frequency is 500 MHz and the synthesized area is 0.098 mm^2. The Flex-SISO is a basic building block in a LDPC decoder or a Turbo decoder, and can be reconfigured to process an eight-state trellis for a Turbo code, or eight check rows for a LDPC code. As the baseline design, a single Flex-SISO decoder can approximately support 30–40 Mbps (LTE) Turbo decoding, or 40–50 Mbps (802.16e or 802.11n) LDPC decoding. In a parallel processing environment, multiple SISO decoders can be used to increase the throughput.

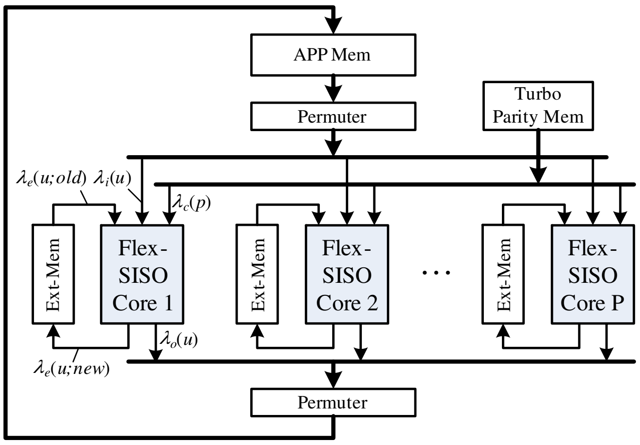


Figure 22 Parallel LDPC/Turbo decoder architecture based on multiple Flex-SISO decoder cores.

Table 6 Performance of the proposed parallel decoder (3.2 mm^2 core area, 500 MHz clock frequency, TSMC 90 nm technology).

Supported codes	Code size (bit)	Parallelism	Quantization	Max. iteration	Max. throughput (Mbps)	Latency
LDPC 802.16e	576–2,304	$z = 24$–96	6.2	15	600	1,590 cycles
LDPC 802.11n	648–1,944	$z = 27$–81	6.2	15	500	1,620 cycles
Turbo 3GPP-LTE	40–6,144	Sub-block = 1–12	6.2	6	450	6,822 cycles

6 Parallel Decoder Architecture Using Multiple Flex-SISO Decoder Cores

For high throughput applications, it is necessary to use multiple SISO decoders working in parallel to increase the decoding speed. For parallel Turbo decoding, multiple SISO decoders can be employed by dividing a codeword block into several sub-blocks and then each sub-block is processed separately by a dedicated SISO decoder [7, 20, 30, 41, 42]. For LDPC decoding, the decoder parallelism can be achieved by employing multiple check node processors [10, 14, 32, 40, 49].

Based on the Flex-SISO decoder core, we proposed a parallel LDPC/Turbo decoder architecture which is shown in Fig. 22. As depicted, the parallel decoder comprises P Flex-SISO decoder cores. In this architecture, there are three types of storage. Extrinsic memory (Ext-Mem) is used for storing the extrinsic LLR values produced by each SISO core. APP memory (APP-Mem) is used to store the initial and updated LLR values. The APP memory is partitioned into multiple banks to allow parallel data transfer. Turbo parity memory is used to store the channel LLR values for each parity bit in a Turbo codeword. This memory is not used for LDPC decoding (parity bits are treated as information bits for LDPC decoding). Two permuters are used to perform the permutation of the APP values back and forth.

As a case study, we have designed a high-throughput, flexible LDPC/Turbo decoder to support the following three codes: 1) 802.16e WiMAX LDPC code, 2) 802.11n WLAN LDPC code, and 3) 3GPP-LTE Turbo code. Table 6 summarizes the performance and design parameters for this decoder. The number of the Flex-SISO decoders is chosen to be 12.

For LDPC decoding, with 12 available Flex-SISO cores the decoder can process up to $12 \times 8 = 96$ check nodes simultaneously. Because the sub-matrix size z is between 24 to 96 for 802.16e LDPC codes, and 27 to 81 for 802.11n, the proposed decoder always guarantees that all of the z check nodes within a layer can be processed in parallel.

For 3GPP-LTE Turbo decoding, the codeword can be partitioned into M sub-blocks for parallel processing. LTE Turbo code uses a quadratic permutation polynomial (QPP) interleaver [36] so that it allows conflict free memory access as long as M is a factor of the codeword length. There are 188 different codeword sizes defined in LTE. For LTE Turbo codes, all of the codewords can support a parallelism level of 8, some of the codewords can support parallelism level of 10 or 12. Because we have 12 Flex-SISO cores available, we will dynamically allocate the maximum possible number of Flex-SISO cores ($8 \leq M \leq 12$) constrained on the QPP interleaver parallelism. As an example, for the maximum codeword size of 6144, we can allocate all of the 12 Flex-SISO cores to work in parallel. It should be noted that the parallelism level has some impact on the error performance of the decoder due to the edge effects caused by the sub-block partitioning [17].

This parallel and flexible decoder has been implemented in Verilog HDL and synthesized on a TSMC 90 nm CMOS technology using Synopsys Design Compiler. The maximum clock frequency of this decoder is 500 MHz. The synthesized core area is 3.2 mm^2, which includes all of the components in this decoder. Table 6 summarizes the features of this decoder. The decoder can be configured to support IEEE 802.16e LDPC codes, IEEE 802.11n LDPC codes, and 3GPP LTE Turbo codes. Compared to a dedicated LDPC

Table 7 Turbo decoder architecture comparison with existing solutions.

	This work	[2]	[34]	[28]
Modes	Turbo, LDPC	Viterbi, Turbo, LDPC	Turbo, LDPC	Viterbi, Turbo, LDPC, RS
Technology	90 nm	65 nm	130 nm	90 nm
Clock frequency	500 MHz	400 MHz	200 MHz	NA
Core area	3.2 mm^2	0.62 mm^2	NA	NA
Throughput (LDPC)	600 Mbps (@15 iter.)	257 Mbps (@10 iter.)	11.2 Mbps (@10 iter.)	70 Mbps
Throughput (Turbo)	450 Mbps[a] (@6 iter.)	18.6 Mbps[a] (@5 iter.)	86.5 Mbps[b] (@8 iter.)	14 Mbps[a]

[a]Binary Turbo code
[b]Double-binary Turbo code

decoder solution [37], this flexible decoder has only about 15–20% area overhead when normalized to the same throughput target (with the same number of iterations). Compared to a dedicated Turbo decoder solution [30], our flexible decoder shows only about 10–20% area overhead when normalized to the same technology and the same throughput and code length.

7 Related Work and Architecture Comparison

Multi-mode Turbo decoders are an increasingly important component in mobile wireless devices. To support multi-mode decoding, the ASIC/ASIP/MPSoC/SIMD architectures have been recently proposed [2, 28, 34]. In [2], a reconfigurable application-specific instruction-set processor (ASIP) architecture is presented for convolutional, Turbo, and LDPC code decoding. In [34], a multi processor system on chip (MPSoC) architecture is described for LDPC and Turbo code decoding. In [28], a SIMD-like processor architecture is proposed for Viterbi, Turbo, Reed-Solomon, and LDPC decoding. Table 7 shows the architecture comparison and tradeoff analysis of these decoders. Each approach has different benefit in terms of flexibility. Our focus is to achieve highest throughput for both LDPC and Turbo codes. As can be seen from the table, the proposed decoder can support very high throughput LDPC/Turbo decoding at a small silicon area cost.

8 Conclusion

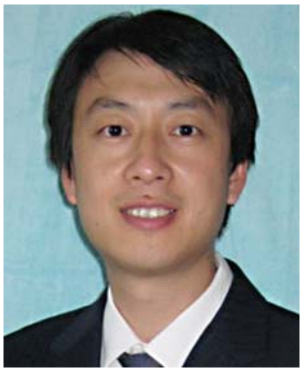
In this work, we present a flexible decoder architecture to support LDPC and Turbo codes. We propose a dual-mode Flex-SISO decoder as a basic building block in LDPC and Turbo decoders. Our study has been focused on the Flex-SISO decoder architecture design and implementation. We unify the decoding process for LDPC and Turbo codes so that the same Flex-SISO decoder can be re-used for both cases resulting in more than 80% resource sharing. To increase decoding throughput, we propose a parallel LDPC/Turbo decoder using multiple Flex-SISO cores. With a core area of 3.2 mm^2, the decoder is able to sustain 600 Mbps 802.11e LDPC decoding, 500 Mbps 802.11n LDPC decoding, or 450 Mbps 3GPP LTE Turbo decoding. The proposed architecture can significantly reduce the cost of a multi-mode receiver.

Acknowledgements The authors would like to thank Nokia, Nokia Siemens Networks (NSN), Xilinx, Texas Instruments (TI), and US National Science Foundation (under grants CCF-0541363, CNS-0551692, CNS-0619767, CNS-0923479, and EECS-0925942) for their support of the research.

References

1. Abbasfar, A., & Yao, K. (2003). An efficient and practical architecture for high speed turbo decoders. *IEEE Vehicular Technology Conference, 1*, 337–341.
2. Alles, M., Vogt, T., & Wehn, N. (2008). FlexiChaP: A reconfigurable ASIP for convolutional, turbo, and LDPC code decoding. In *2008 5th International symposium on turbo codes and related topics* (pp. 84–89).
3. Bahl, L., Cocke, J., Jelinek, F., & Raviv, J. (1974). Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Transactions on Information Theory IT-20*, 284–287.
4. Berrou, C., Glavieux, A., & Thitimajshima, P. (1993). Near Shannon limit error-correcting coding and decoding: Turbo-codes. In *IEEE Int. conf. commun.* (pp. 1064–1070).
5. Bickerstaff, M., Davis, L., Thomas, C., Garrett, D., & Nicol, C. (2003). A 24Mb/s radix-4 logMAP turbo decoder for 3GPP-HSDPA mobile wireless. In *IEEE Int. solid-state circuit conf. (ISSCC)*.
6. Blanksby, A. J., & Howland, C. J. (2002). A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder. *IEEE Journal of Solid-State Circuits, 37*, 404–412.
7. Bougard, B., Giulietti, A., Derudder, V., Weijers, J. W., Dupont, S., Hollevoet, L., Catthoor, F., et al. (2003). A scalable 8.7-nJ/bit 75.6-Mb/s parallel concatenated convolutional (turbo-) codec. In *IEEE International solid-state circuit conference (ISSCC)*.
8. Bougard, B., Giulietti, A., Van der Perre, L., & Catthoor, F. (2002). A class of power efficient VLSI architectures for high speed turbo-decoding. In *IEEE conf. global telecommunications* (Vol. 1, pp. 549–553).
9. Brack, T., Alles, M., Kienle, F., & Wehn, N. (2006). A synthesizable IP core for WIMAX 802.16e LDPC code decoding. In *IEEE 17th Int. symp. personal, indoor and mobile radio communications* (pp. 1–5).
10. Brack, T., Alles, M., Lehnigk-Emden, T., Kienle, F., Wehn, N., L'Insalata, N., et al. (2007). Low complexity LDPC code decoders for next generation standards. In *Design, automation, and test in Europe* (pp. 331–336). New York: ACM
11. Chen, J., Dholakia, A., Eleftheriou, E., Fossorier, M., & Hu, X. (2005). Reduced-complexity decoding of LDPC codes. *IEEE Transactions on Communications, 53*, 1288–1299.
12. Dai, Y., Yan, Z., & Chen, N. (2006). High-throughput turbo-sum-product decoding of QC LDPC codes. In *40th Annual conf. on info. sciences and syst.* (Vol. 11, pp. 839– 8446).
13. Gallager, R. (1963). *Low-density parity-check codes*. Cambridge: MIT.
14. Gunnam, K. K., Choi, G. S., Yeary, M. B., & Atiquzzaman, M. (2007). VLSI architectures for layered decoding for irregular LDPC codes of WiMax. In *IEEE International Conference on Communications (ICC)* (pp. 4542–4547).
15. Hagenauer, J., Offer, E., & Papke, L. (1996). Iterative decoding of binary block and convolutional codes. *IEEE Transactions on Information Theory, 42*(2), 429–445.

16. Hagenauer, J., Offer, E., & Papke, L. (1996). Iterative decoding of binary block and convolutional codes. *IEEE Transactions on Information Theory, 42*, 429–445.
17. He, Z., Fortier, P., & Roy, S. (2006). Highly-parallel decoding architectures for convolutional turbo codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 14*(10), 1147–1151.
18. Hocevar, D. (2004). A reduced complexity decoder architecture via layered decoding of LDPC codes. In *IEEE workshop on signal processing systems (SIPS)* (pp. 107–112).
19. Dielissen, J., & Huisken, J. (2000). State vector reduction for initialization of sliding windows MAP. In *2nd International symposium on turbo codes and related topics*.
20. Lee, S. J., Shanbhag, N., & Singer, A. (2005). Area-efficient high-throughput MAP decoder architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 13*, 921–933.
21. Lin, Y., Mahlke, S., Mudge, T., & Chakrabarti, C. (2006). Design and implementation of turbo decoders for software defined radio. In *IEEE SIPS* (pp. 22–27).
22. Lu, J., & Moura, J. (2003). Turbo like decoding of LDPC codes. In *IEEE Int. conf. on magnetics* (pp. DT-11).
23. MacKay, D. J. C. (1998). Turbo codes are low density parity check codes. Available online, http://www.inference.phy.cam.ac.uk/mackay/turbo-ldpc.pdf.
24. Mansour, M. M., & Shanbhag, N. R. (2003). High-throughput LDPC decoders. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 11*, 976–996.
25. Masera, G., Piccinini, G., Roch, M., & Zamboni, M. (1999). VLSI architecture for turbo codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 7*, 369–3797.
26. Masera, G., Quaglio, F., & Vacca, F. (2005). Finite precision implementation of LDPC decoders. In *IEEE proc. commun.* (Vol. 152, pp. 1098–1102).
27. Mohsenin, T., Truong, D., & Baas, B. (2009). Multi-split-row threshold decoding implementations for LDPC codes. In *IEEE International symposium on circuits and systems (ISCAS'09)* (pp. 2449–2452).
28. Niktash, A., Parizi, H., Kamalizad, A., & Bagherzadeh, N. (2008). RECFEC: A reconfigurable FEC processor for Viterbi, turbo, Reed-Solomon and LDPC coding. In *IEEE Wireless communications and networking conference (WCNC)* (pp. 605–610).
29. Oh, D., & Parhi, K. (2006). Low complexity implementations of sum-product algorithm for decoding low-density parity-check codes. In *IEEE Workshop on signal processing systems (SIPS)* (pp. 262–267).
30. Prescher, G., Gemmeke, T., & Noll, T. (2005). A parametrizable low-power high-throughput turbo-decoder. In *IEEE Int. conf. acoustics, speech, and signal processing* (Vol. 5, pp. 25–28).
31. Robertson, P., Villebrun, E., & Hoeher, P. (1995). A comparison of optimal and sub-optimal MAP decoding algorithm operating in the log domain. In *IEEE Int. conf. commun. (ICC)* (pp. 1009–1013).
32. Rovini, M., Gentile, G., Rossi, F., & Fanucci, L. (2007). A scalable decoder architecture for IEEE 802.11n LDPC codes. In *IEEE global telecommunications conference* (pp. 3270–3274).
33. Salmela, P., Sorokin, H., & Takala, J. (2008). A programmable Max-Log-MAP turbo decoder implementation. *Hindawi VLSI Design, 2008*, 636–640.
34. Scarpellino, M., Singh, A., Boutillon, E., & Masera, G. (2008). Reconfigurable architecture for LDPC and turbo decoding: A NoC case study. In *IEEE 10th International symposium on spread spectrum techniques and applications* (pp. 671–676).
35. Shih, X. Y., Zhan, C. Z., Lin, C. H., & Wu, A. Y. (2008). An 8.29 mm^2 52 mW multi-mode LDPC decoder design for mobile WiMAX system in 0.13 m CMOS process. *IEEE Journal of Solid-State Circuits, 43*, 672–683.
36. Sun, J., & Takeshita, O. (2005). Interleavers for turbo codes using permutation polynomials over integer rings. *IEEE Transactions on Information Theory, 51, 101–119*.
37. Sun, Y., & Cavallaro, J. R. (2008). A low-power 1-Gbps reconfigurable LDPC decoder design for multiple 4G wireless standards. In *IEEE International SOC conference* (pp. 367–370).
38. Sun, Y., & Cavallaro, J. R. (2008). Unified decoder architecture For LDPC/Turbo codes. In *IEEE Workshop on Signal Processing Systems (SIPS)* (pp. 13–18).
39. Sun, Y., Karkooti, M., & Cavallaro, J. R. (2006). High throughput, parallel, scalable LDPC encoder/decoder architecture for OFDM systems. In *IEEE workshop on design, applications, integration and software* (pp. 39–42).
40. Sun, Y., Karkooti, M., & Cavallaro, J. R. (2007). VLSI decoder architecture for high throughput, variable block-size and multi-rate LDPC codes. In *IEEE International symposium on circuits and systems (ISCAS)* (pp. 2104–2107).
41. Sun, Y., Zhu, Y., Goel, M., & Cavallaro, J. R. (2008). Configurable and scalable high throughput turbo decoder architecture for multiple 4G wireless standards. In *IEEE International conference on application-specific systems, architectures and processors (ASAP)* (pp. 209–214).
42. Thul, M. J., Gilbert, F., Vogt, T., Kreiselmaier, G., & Wehn, N. (2005). A scalable system architecture for high-throughput turbo-decoders. *Journal of VLSI Signal Processing, 39*, 63–77.
43. Viterbi, A. (1998). An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes. *IEEE Journal on Selected Areas in Communications, 16*, 260–264.
44. Wang, Z., Chi, Z., & Parhi, K. (2002). Area-efficient high-speed decoding schemes for turbo decoders. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 10*, 902–912.
45. Wang, Z., & Cui, Z. (2007). Low-complexity high-speed decoder design for quasi-cyclic LDPC codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 15*, 104–114.
46. Zhang, J., & Fossorier, M. (2002). Shuffled belief propagation decoding. In *Asilomar Conference on signals, systems and computers* (Vol. 1, pp. 8–15).
47. Zhang, K., Huang, X., & Wang, Z. (2009). High-throughput layered decoder implementation for quasi-cyclic LDPC codes. *IEEE Journal on Selected Areas in Communications, 27*(6), 985–994.
48. Zhang, T., Wang, Z., & Parhi, K. (2001). On finite precision implementation of low density parity check codes decoder. In *IEEE Int. symposium on circuits and systems (ISCAS)* (Vol. 4, pp. 202–205).
49. Zhong, H., & Zhang, T. (2005). Block-LDPC: A practical LDPC coding system design approach. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, 52*(4), 766–775 (see also IEEE Transactions on Circuits and Systems I: Regular Papers).
50. Zhu, Y., & Chakrabarti, C. (2009). Architecture-aware LDPC code design for multiprocessor software defined radio systems. In *IEEE transactions on signal processing* (Vol. 57, pp. 3679–3692).



Yang Sun received the B.S. degree in Testing Technology & Instrumentation in 2000 and the M.S. degree in Instrument Science & Technology in 2003, from Zhejiang University, Hangzhou, China. From 2003 to 2004, he was with S3 Graphics Co. Ltd. as an ASIC design engineer, developing Graphics Processing Unit (GPU) cores for graphics chipsets. From 2004 to 2005, he was with Conexant Systems Inc. as an ASIC design engineer, developing video decoder cores for set-top box (STB) chipsets. During the summer of 2007 and 2008, he worked at Texas Instruments - R&D center as an intern, developing LDPC and Turbo error-correcting decoders.

He is currently a PhD student in the Department of Electrical and Computer Engineering at Rice University, Houston, Texas. His research interests include parallel algorithms and VLSI architectures for wireless communication systems. He received the 2008 IEEE SoC Conference Best Paper Award, the 2008 IEEE Workshop on Signal Processing Systems Bob Owens Memory Paper Award, and the 2009 ACM GLSVLSI Best Student Paper Award.

Joseph R. Cavallaro received the B.S. degree from the University of Pennsylvania, Philadelphia, Pa, in 1981, the M.S. degree from Princeton University, Princeton, NJ, in 1982, and the Ph.D. degree from Cornell University, Ithaca, NY, in 1988, all in electrical engineering. From 1981 to 1983, he was with AT&T Bell Laboratories, Holmdel, NJ. In 1988, he joined the faculty of Rice University, Houston, TX, where he is currently a Professor of electrical and computer engineering. His research interests include computer arithmetic, VLSI design and microlithography, and DSP and VLSI architectures for applications in wireless communications. During the 1996–1997 academic year, he served at the National Science Foundation as Director of the Prototyping Tools and Methodology Program. He was a Nokia Foundation Fellow and a Visiting Professor at the University of Oulu, Finland in 2005 and continues his affiliation there as an Adjunct Professor. He is currently the Associate Director of the Center for Multimedia Communication at Rice University. He is a Senior Member of the IEEE. He was Co-chair of the 2004 Signal Processing for Communications Symposium at the IEEE Global Communications Conference and General Co-chair of the 2004 IEEE 15th International Conference on Application-Specific Systems, Architectures and Processors (ASAP).