82.1 Introduction

The publication of Shannon’s ground breaking paper [1,2] in 1948, identifying the fundamental limits of information transfer capacity of noisy communication channels, resulted in the birth of information theory and coding theory as two distinct areas of research. Coding theory involves the investigation...
of efficient error control codes (ECC), the study of their performance bounds, and the determination of computationally efficient decoding algorithms. In fact, the present moment is historic in another sense as the tremendous advances in the theory and practice of coding theory converges with the semiconductor industry trend embodied by Moore's Law \[3\]. The latter has enabled the implementation of decoders that were considered too complex to implement just a few years ago. As a result, it is not uncommon to see codes such as low-density parity check (LDPC) codes \[4\] being "rediscovered" and incorporated in next generation communications standards.

A well-designed code can enable orders-of-magnitude reduction in the bit error-rate (BER) with a nominal cost in terms of power and latency. ECC can be said to have singularly resulted in the tremendous growth in communication infrastructure that has transformed modern society. This is evident in the pervasiveness of the Internet, cellular phones, modems, and wireless systems today and the continuing emergence of newer generations of communications standards ranging from digital video broadcast (DVB-H, T-DMB \[5,6\]), wireless metropolitan area network (MAN) (IEEE 802.16 \[7\]), wireless local area network (LAN) (IEEE 802.11g \[8\]), wireless personal area network (Bluetooth, \[9\]), 4G \[10\], fiber communications (OC-192 \[11\]), asymmetric digital subscriber lines (ADSL \[12\]), backplane/storage area networks (ANSI X3.230 \[13\]), and many others. The availability of a reliable, low-cost implementation substrate such as silicon has made integrated circuits (ICs) for communications an important area of research in industry as well as in academia.

A key component of any communications standards mentioned above is the ECC and a key requirement in being able to deliver low-cost silicon is a deep understanding of the fundamentals of ECC performance and their very large-scale integrated circuit (VLSI) architectures. This chapter presents the basics of the theory underlying some of the commonly employed ECCs as well as some of the more advanced ones such as LDPC along with discussion of issues involved in a VLSI implementation of ECC decoders. This chapter is organized as follows: in Section 82.2, we present Reed–Solomon (RS) decoders followed by convolutional codes, the Viterbi algorithm, and Viterbi decoder architectures in Section 82.3. In Section 82.4, we present turbo decoders that were discovered in the early 1990s and are considered to be a major breakthrough in coding theory, and since then have been incorporated in many wireless communication standards. Finally, in Section 82.5, we present LDPC decoders which are considered as a competitor to turbo codes and are being evaluated for inclusion in next-generation communication standards.

### 82.2 RS Decoder Architectures

RS codes \[14–16\] are employed in numerous communications systems. These include deep space, digital subscriber loops, data storage, wireless, and more recently, optical systems. The drive toward higher data rates makes it necessary to devise very high-speed implementations of decoders for RS codes.

An \((n,k)\) RS code is a mapping from \(k\) data symbols to \(n\) code-symbols. The \((255, 239)\) RS code is a good example. A data or code-symbol is an \(m\)-bit word where \(m = 8\) (i.e., a byte) is typical. Such a symbol is said to lie in a finite field or a Galois field of size \(2^m\) (GF(\(2^m\))), e.g., GF(256) is the set of all 8-bit numbers. Usually, \(n = 2^m - 1\) and the construction of an \((n,k)\) RS code guarantees that up to \(t = \left\lfloor (n - k)/2 \right\rfloor\) symbol errors are correctable. For example, up to 8 symbol errors are correctable in a \((255, 239)\) code.

Inclusion of an \((n,k)\) RS code in a communication system results in code rate \(R = k/n\) and an increase in line rate by a factor of \(1/R\). For example, earlier in this decade, optical links at OC-192 data rates \((9.953\, \text{Gb/s})\) needed to increase their data rates to 10.6 Gb/s to include the overhead owing to the inclusion of \((255, 239)\) RS code. This coding overhead is considered quite acceptable because the code provides a few orders-of-magnitude reduction in the BER for the same signal-to-noise ratio (SNR).

#### 82.2.1 Finite-Field Arithmetic

An RS encoder and decoder is composed of finite-field arithmetic units such as the finite-field adder, multiplier, and divider. Finite-field arithmetic units map operands from GF(\(2^m\)) back into GF(\(2^m\)) and thus differ from conventional integer arithmetic such as two's complement. Finite-field computations are elegantly described if we view elements of GF(\(2^m\)) as polynomials with degree at most \(m - 1\), e.g., the element 10000101
82.2.2 RS Encoder

The encoding and decoding processes are elegantly described if we view the \( k \) data symbols and \( n \) code symbols as polynomials over GF\( (2^m) \), i.e., polynomials whose coefficients are in GF\( (2^m) \).

The encoding process is best described in terms of the data polynomial \( D(z) = d_{k-1}z^{k-1} + d_{k-2}z^{k-2} + \cdots + d_2z + d_0 \), where \((d_{k-1}, d_{k-2}, \ldots, d_2, d_0)\) are \( k \) \( m \)-bit data symbols, being transformed into a codeword polynomial \( C(z) = c_{n-1}z^{n-1} + c_{n-2}z^{n-2} + \cdots + c_2z + c_0 \), where \((c_{n-1}, c_{n-2}, \ldots, c_2, c_0)\) are \( n \) \( m \)-bit code symbols. These code symbols are transmitted over the communication channel (or stored in memory).

All codeword polynomials \( C(z) \) are polynomial multiples of \( G(z) \), the generator polynomial of the code, which is defined as

\[
G(z) = \prod_{i=0}^{2t-1} (z - \alpha^{m^i})
\]  

(82.2)

where \( m_0 \) is typically 0 or 1 and \( \alpha \) is the primitive element. However, other choices sometimes simplify the decoding process slightly. All communication standards specify \( G(z) \), e.g., \( G(z) = \prod_{i=0}^{2t-1} (z - \alpha^i) \) for wireless MAN systems [18].

Since \( 2t \) consecutive powers \( \alpha^{m^i}, \alpha^{m^{i+1}}, \ldots, \alpha^{m^{2t-1}} \) of \( \alpha \) are roots of \( G(z) \), and \( C(z) \) is a multiple of \( G(z) \), it follows that

\[
C(\alpha^{m^i}) = 0, \quad 0 \leq i \leq 2t - 1
\]  

(82.3)

for all codeword polynomials \( C(z) \). In fact, an arbitrary polynomial of degree less than \( n \) is a codeword polynomial if and only if it satisfies Eq. (82.3).

A systematic encoder produces codewords that are comprised of data symbols followed by parity-check symbols, and is obtained as follows: Let \( Q(z) \) and \( P(z) \) denote the quotient and remainder, respectively, when the polynomial \( z^{-k}D(z) \) of degree \( n-1 \) is divided by \( G(z) \) of degree \( 2t = n-k \). Then, the codeword \( C(z) \) is given by

\[
(c_{n-1}, c_{n-2}, \ldots, c_2, c_0) = (d_{k-1}, d_{k-2}, \ldots, d_2, d_0, -p_{n-k-1}, -p_{n-k-2}, \ldots, -p_t, -p_0)
\]

and consists of data symbols followed by parity-check symbols. The architecture of a systematic RS encoder is shown in Figure 82.1.
In broadband communication systems, the RS encoder is typically followed by a convolutional encoder. In the absence of a convolutional encoder, the code symbols are input to a channel modulator, which maps the code symbols onto a signal constellation [19]. The choice of the signal constellation depends on the transmit power, the channel frequency response and SNR, the data rate, and also the required BER at the receiver. Typical signal constellations include BPSK, QPSK, and QAM [19].

82.2.3 RS Decoder

RS decoders are classified into two categories: hard-decision RS decoders and soft-decision RS decoders. A hard-decision RS decoder receives one symbol per transmitted symbol and no other information. In contrast, a soft-decision decoder may receive one or more symbols for each transmitted code symbol in addition to information regarding the reliability of these symbols. We focus on hard-decision decoders in this section as they constitute the majority of decoders in current implementations. A brief introduction to soft-decision decoders is provided at the end of this section.

The input to the hard-decision decoder for each transmitted codeword is the set of received symbols: \( r_n, r_{n-2}, \ldots, r_0 \) termed the received word. Analogous to the transmitted codeword polynomial \( C(z) \), the received word can also be viewed as a polynomial \( R(z) \). Any of the received symbols in \( R(z) \) may be in error. Hence, the received word \( R(z) \) can be written as

\[
R(z) = C(z) + E(z)
\]  

(82.4)

where the number of nonzero coefficients in the error-polyomial \( E(z) \) depends on the number of erroneous symbols \( v \) within the received word \( R(z) \). Since the maximum number of decoder correctable errors \( t \) in a received word is usually much smaller than the code polynomial degree \( n - 1 \), for typical high rate codes, \( E(z) \) can be compactly expressed as

\[
E(z) = Y_1 z^i_1 + Y_2 z^i_2 + \cdots + Y_v z^i_v
\]  

(82.5)

where \( Y_1, Y_2, \ldots, Y_v \) are the error values and \( X_1 = \alpha^i_1, X_2 = \alpha^i_2, \ldots, X_v = \alpha^i_v \) are the error locations. A hard-decision decoder can detect and correct all the erroneous symbols as long as \( v \leq t \). An efficient RS decoder determines the error values and error locations in four stages, which are listed below:

1. Syndrome polynomial computation
2. Key equation solution
3. Chien formula-based error location search
4. Forney formula-based error value evaluation

The architectures for each of these steps will be discussed in the remainder of this section.
(1) **Architectures for syndrome polynomial computation.** Syndrome computation is the evaluation of the received word at the roots of the transmitted codeword or the generator polynomial \( G(z) \). The syndromes \( s_i \) for \( 0 \leq i < 2t \) are given by

\[
s_i = R(\alpha^{m_0 + i})
\]

From Eq. (82.3) and Eq. (82.4), it follows that \( R(\alpha^{m_0 + i}) = E(\alpha^{m_0 + i}) \). Hence, syndrome computation is nothing but the evaluation of the error polynomial at \( 2t \) finite-field locations. Syndrome computation is typically performed using Horner’s rule which computes \( R(\alpha^{m_0 + i}) \) as

\[
R(\alpha^{m_0 + i}) = ((r_{m_0} \alpha^{m_0 + i} + r_{m_1}) \alpha^{m_0 + i} + \ldots + r_{l}) \alpha^{m_0 + i} + r_0
\]  

(82.6)

The \( n - 1 \) multiplications and \( n \) additions in Eq. (82.6) are recursively computed using a multiplier–accumulator (MAC). The architecture for computing \( 2t \) syndromes in parallel using \( 2t \) multipliers and adders with Horner’s rule is shown in Figure 82.2(a).

This architecture requires \( n \) clock cycles to complete the computation. It should be noted that one of the operands to the multiplier is a constant. The complexity of constant input multipliers is roughly half that of a regular multiplier. One can further reduce area by pipelining the multipliers and folding more than one syndrome value computation onto the same MAC.

High-throughput syndrome computation architectures are obtained by using a two-stage implementation as shown in Figure 82.2(b). In the first stage, \( l \) input received symbols are passed through each MAC to compute a partial product of \( R(\alpha^{m_0 + i}) \). In the second stage, another multiplier–adder pair multiplies and adds the partial products together. Both stages are based on Horner’s rule.

Given the syndrome values \( s_0, s_1, \ldots, s_{2t-1} \), the syndrome polynomial is defined as

\[
S(z) = s_0 + s_1 z + \ldots + s_{2t-1} z^{2t-1}
\]

The syndrome polynomial is input to the key equation solver which is discussed next.

(2) **Architectures for key equation solver.** The key equation solver block contains recursion and hence determines the throughput of the entire RS decoder. The outputs of the key equation solver are the error
locator polynomial $\Lambda(z)$, and the error evaluator polynomial $\Omega(z)$ defined as

$$\Lambda(z) = \prod_{j=1}^{r} (1 - X_j z) = 1 + \lambda_1 z + \lambda_2 z^2 + \cdots + \lambda_r z^r$$  \hspace{1cm} (82.7)

$$\Omega(z) = \sum_{j=1}^{r} Y_j X_j^{-m} \prod_{j' \neq j} (1 - X_{j'} z) = \omega_0 + \omega_1 z + \cdots + \omega_{t-1} z^{t-1}$$  \hspace{1cm} (82.8)

where $v \leq t$ are the number of symbol errors and the $r$ roots of the error locator polynomial are the error locations obtained via Chien search (described in Section 82.1.4). The error evaluator polynomial contains information regarding the error values that are obtained using Forney’s formula as shown in Section 82.1.4.

The key equation relates the syndrome polynomial $S(z)$, the error locator polynomial $\Lambda(z)$, and the error evaluator polynomial $\Omega(z)$ as follows:

$$\Lambda(z)S(z) = \Omega(z) \text{mod } z^{2t}$$  \hspace{1cm} (82.9)

The key equation solver determines the polynomials $\Lambda(z)$ and $\Omega(z)$ using $S(z)$. From Eq. (82.9), it is easily seen that once $\Lambda(z)$ is determined, $\Omega(z)$ is obtained as the coefficients of the first $2t$ terms in the product $\Lambda(z)S(z)$. In addition, we have from the linear complexity property [16] of the finite-field Fourier transform that the error locator satisfies the following linear recursion:

$$s_k = - \sum_{j=1}^{r} \hat{\lambda}_j s_{k-j}, \quad k = v, \ldots, 2t - 1$$  \hspace{1cm} (82.10)

In other words, the error locator polynomial $\Lambda(z)$ is a minimal degree predictor of the syndrome sequence $\{s_v, \ldots, s_{2t-1}\}$. Hence, $\Lambda(z)S(z)$ has zero coefficients for terms with degree $v$ to $2t - 1$. Consequently, $\Omega(z)$ is also guaranteed to have zero coefficients for degrees $v$ to $2t - 1$.

Hence using Eq. (82.10), the key equation can be rewritten as:

$$\Lambda(z)S(z) = \Omega(z) + z^{2t} \Omega^{(1)}(z) \text{ with } \deg(\Omega(z)) \leq v$$  \hspace{1cm} (82.11)

Solving the key equation comprises two separate problems: determining the least degree polynomial $\Lambda(z)$ (with degree $v$) such that the product $\Lambda(z)S(z)$ has zero coefficients from degree $v$ to degree $2t - 1$. Note that, since the maximum correctable errors is limited to $t$, all valid error locators have degree $v \leq t$. This is followed by determining the product $\Lambda(z)S(z)$ up to degree $v - 1$. Typically, both the problems of determining the error locator and the error evaluator are solved together in an iterative process.

Two main iterative algorithms available for solving the key equation are: the Berlekamp–Massey algorithm (BMA) [16] and the Euclid’s algorithm [20]. Architectures for both these algorithms are discussed next.

The BMA algorithm solves the modified problem of determining a minimum degree polynomial $\Lambda(z)$ that satisfies Eq. (82.10) under the relaxed condition: $\deg(\Lambda(z)) \leq 2t$. This relaxation ensures that the BMA always has a solution. When the solution obtained has a degree $v \leq t$, the BMA solution is in fact the error locator polynomial $\Lambda(z)$ of Eq. (82.9).

The BMA is an iterative algorithm. In the $r$th of the $2t$ iterations, it generates the minimum degree polynomial $\Lambda(r,z)$ of degree $v$, that satisfies the following linear recursion up to the current iteration index $r$:

$$s_k = - \sum_{j=1}^{r} \hat{\lambda}_{jr} s_{k-j}, \quad k = v, \ldots, r$$

In other words, $\Lambda(r,z)$ is a predictor of the syndrome sequence $s_v$ to $s_r$. In each iteration $r$, a predicted syndrome $\hat{s}_r$ is computed as

$$\hat{s}_r = - \sum_{j=1}^{v-1} \hat{\lambda}_{jr} s_{r-j}$$  \hspace{1cm} (82.12)
This is subtracted from the actual syndrome value $s_i$ to obtain the *discrepancy coefficient* $\Delta_i = s_i - \hat{s}_i$. If $\Delta_i \neq 0$, then the error locator is updated by addition of a scaled version of the error locator from a previous iteration called the *scratch polynomial*. The scaling constant is chosen to reduce the discrepancy coefficient to zero. The scratch polynomial is updated every iteration to ensure that the minimum degree property of the error locator is maintained.

From an architectural perspective, the throughput bottleneck occurs in computing the discrepancy coefficient. This is because the critical path in computing Eq. (82.12) passes through at least one multiplier and a logarithmic number of adders implementing Eq. (82.12). Updating the error locator and error evaluator polynomials has a fixed complexity and involves multiplication of a polynomial with a scalar constant followed by component-wise addition. These operations have a critical path passing through a multiplier and an adder.

A systolic architecture to reduce the critical path has been described in [21]. The main idea behind this architecture is to iterate the discrepancy coefficients in a manner similar to the computation of the error locator polynomial. With this modification, the architecture becomes extremely regular. The critical path within the iteration is then reduced to a multiplier followed by an adder. This increases the throughput of the RS decoder.

The pseudocode of the reformulated inversionless Berlekamp–Massey (RiBM) algorithm from Ref. [20] is given in Figure 82.3. The systolic architecture implementing the pseudocode is shown in Figure 82.4. In the pseudocode, $\delta_i(0)$ (for $0 \leq i < 2t$) denotes the initial value of the discrepancy polynomial and $\theta_i(0)$ (for $0 \leq i < 2t$) is the initial value of the corresponding scratch polynomial. Note that the $\delta_0(0)$ corresponds to the initial error locator polynomial (set to unity), and $\theta_0(0)$ corresponds to its scratch polynomial (also initialized to unity). The architecture shown in Figure 82.4(a) consists of an array of $3t$ identical processing elements. The details of the processing elements are shown in Figure 82.4(b). As seen in this figure, the critical path of the architecture passes through one multiplier and an adder. With each iteration, all the terms of the error locator and the discrepancy polynomial move one position to the left. Hence, the discrepancy coefficient required in the current iteration (denoted by $\delta_i(r)$ in Figure 82.4) is always available at the leftmost processing element. At the end of $2t$ iterations, the error locator is available at processing elements from $t$ to $2t$ and $\Omega^\infty(z)$ from Eq. (82.11) at processing elements numbered $0$ to $t - 1$. More details on the reformulation leading to this algorithm and further architectural details are available in Ref. [21].

In addition to BMA, the key equation can also be solved by using the well-known Euclid’s algorithm [20] for determining the greatest common divisor (GCD) of two polynomials. RS decoder implementation

---

**VLSI Architectures for Forward Error-Control Decoders**

**Initialization**: $\delta_0(0)=1, \theta_0(0)=1; \quad \delta_i(0)=0 \quad \text{for} \quad i=2t, \ldots, 3t-1 \quad k(0)=0 \quad \gamma(0)=1$

**Input**: $s_i, i=0,1, \ldots, 2t-1$

$\delta_0(0)=\theta_0(0)=s_i, (i=0,1, \ldots, 2t-1)$

for $r=0$ to $2t-1$ do

**Step RiBM.1**

$\delta_i(r+1) = \gamma(r) \delta_{i-1}(r) - \delta_i(r) \cdot \theta_i(r), (i=0,1, \ldots, 3t)$

**Step RiBM.2**

if $\delta_i(r) \neq 0$ and $k(r) \geq 0$

$\theta_i(r+1) = \delta_{i-1}(r), \quad (i=0,1, \ldots, 3t)$

$\gamma(r+1) = \delta_i(r)$

$k(r+1) = -k(r)-1$

else

$\theta_i(r+1) = \theta_{i-1}(r), \quad (i=0,1, \ldots, 3t)$

$\gamma(r+1) = \gamma(r)$

$k(r+1) = k(r)+1$

end if

end for

**Output**: $\lambda_i = \delta_{i+1}, (i=0,1, \ldots, t); \quad \omega_i = \delta_i(2t), \quad (i=0,1, \ldots, t-1)$

---

© 2006 by CRC Press LLC
based on the Euclid's algorithm is a well-researched problem \[22–24\]. In general, if \(A(z)\) and \(B(z)\) are two polynomials with coefficients over a field, and if \(G(z)\) is the GCD of the two polynomials, then Euclid's algorithm expresses \(G(z)\) as a function of \(A(z)\) and \(B(z)\), i.e.,

\[
G(z) = R(z)A(z) + T(z)B(z) \tag{82.13}
\]

Comparing Eq. (82.11) and Eq. (82.13), we see that the key equation is equivalent to finding the GCD \(G(z)\) (error evaluator \(\Omega(z)\)) and \(R(z)\) (error locator \(\Lambda(z)\)) given the syndrome polynomial \(S(z) = A(z)\) and \(B(z) = z^n\). Euclid's algorithm finds the GCD through a process of iterative division of two polynomials and their remainders. This is followed by a traceback of the iterations to find \(R(z)\). However, a modified form of the Euclid's algorithm called the extended Euclid's algorithm determines both the GCD (error evaluator) and the coefficient polynomial (error locator) in the same set of iterations. The pseudocode of the extended Euclid's algorithm, based on Ref. [22] is shown in Figure 82.5. In this figure, \(a_i\) and \(b_i\) are the leading coefficients of \(R_i(z)\) and \(Q_i(z)\), respectively. This pseudocode can be implemented on a systolic array of \(2t\) processing elements similar to that of the systolic BMA. The
VLSI Architectures for Forward Error-Control Decoders

structure of one processing element [22] is shown in Figure 82.6. Further details on this implementation can be found in Ref. [22].

Both the extended Euclid’s architecture and the RiBM architecture have the same critical path and require the same number of iterations for processing a syndrome polynomial. If the number of multipliers is taken as a representative of the hardware complexity, then the systolic RiBM architecture requires 6t multipliers whereas the systolic extended Euclid’s architecture requires 8t multipliers. In addition, the control circuitry of the Euclid’s architecture is more complex compared to the RiBM architecture. Pipelining the BM architecture and the extended Euclid’s architecture is discussed in Ref. [21] and Ref. [23], respectively.

Initialization: $R_0(z) = S(z)$; $Q_0(z) = z^{2^t}$; $\Theta(z) = 0$; $\Phi(z) = 1$

for $i = 0$ to $2t-1$ and $\text{deg}(R_i) > t$

$\ell_i = \text{deg}(R_i(z)) - \text{deg}(Q_i(z))$

if ($\ell_i \geq 0$)

$R_{i+1}(z) = b_i R_i(z) - z^{\ell_i} a_i Q_i(z)$

$Q_{i+1}(z) = Q_i(z)$

$\Theta_{i+1}(z) = b_i \Theta_i(z) - z^{\ell_i} a_i \Phi_i(z)$

$\Phi_{i+1}(z) = \Phi_i(z)$

else

$R_{i+1}(z) = a_i Q_i(z) - z^{\ell_i} b_i R_i(z)$

$\Theta_{i+1}(z) = R_i(z)$

$\Theta_{i+1}(z) = a_i \theta_i(z) - z^{\ell_i} b_i \Theta_i(z)$

$\Phi_{i+1}(z) = \Phi_i(z)$

end if

end for

Output: $\Lambda(z) = \Theta(z)$; $\Omega(z) = R_{i+1}(z)$.

FIGURE 82.5 Pseudocode of the extended Euclid’s algorithm.

FIGURE 82.6 Processing element of the extended Euclid’s architecture.
82.2.4 Chien Search and Forney’s Formula

The roots of the error locator polynomial $\Lambda(z)$ can be obtained from the Chien search procedure [25]. The error values at the error locations can be determined from the Forney formula [26]. Chien search involves the evaluation of the error locator polynomial at all the elements of the finite field and the flagging of those elements at which the polynomial evaluated to zero. Note that, from Eq. (82.7), the roots of the error locator are $X_i^{-1}$ for each error location $X_i = \alpha$. Since the error locator degree is constrained to a maximum of $t$, we require $t$ multiplications and $t + 1$ additions to evaluate the error locator at one finite-field symbol value. Hence the complexity of Chien search is $nt$ multiplications and $n(t + 1)$ additions. Note that Chien search can be implemented via the Horner’s rule as discussed in connection with syndrome computation. The highest throughput is obtained when all the evaluations are performed in parallel using $n$ MACs while minimum area implementations perform all the evaluations serially using one MAC. The Chien search architecture is chosen on the basis of the throughput of the other blocks such as the key equation solver architecture.

If $\deg(\Lambda(z)) \leq t$ and all the roots of $\Lambda(z)$ lie in the Galois field of operation, then the error values at these error locations are found using the Forney algorithm. Given the error location $X_i$, the error locator $\Lambda(z)$, and the error evaluator $\Omega(z)$, the error value $Y_i$ is given by Forney’s formula [26] as

$$Y_i = \frac{z^{m-1}\Omega(z)}{\Lambda(z)} \bigg|_{z = X_i^{-1}} \quad (82.14)$$

The denominator in Forney’s formula is obtained as part of the Chien search procedure. The Forney formula still requires an inverter. Efficient inverter architectures are provided in Ref. [17]. The numerator in Eq. (82.14) is computed using the Horner’s rule. The computed error values are subtracted from the received word to obtain the transmitted codeword.

82.2.5 Advanced RS Decoders: Soft-Decision Decoders

Reliability information, either at the symbol level or at the bit level can be used to improve on the bit-error performance obtained from hard-decision decoders described thus far. The simplest soft-decision decoder is an erasure decoder [16] that employs received symbol reliability information and a preset threshold to declare certain received symbols to be erasures. Each erasure reduces the minimum distance of the code by unity. With $\mu$ erasures, the maximum number of correctable errors $t'$ among the unerased symbols is given by

$$t' = \left\lfloor \frac{n-k-\mu}{2} \right\rfloor$$

A hard-decision decoder can perform erasure decoding with modification to the initialization of the error locator and error evaluator. Hence, the complexity of an erasure decoder is similar to that of a hard-decision decoder. If the erased locations contain some of the erroneous received symbols, then the total number of errors that can be corrected is greater than that of the hard-decision decoder. The maximum number of received symbols that can be erased is however limited by the minimum distance of the code.

An extension of the erasure decoding idea is the generalized minimum distance (GMD) algorithm [27], that iteratively increases the number of erasures and checks for decodability at each iteration. This algorithm performs $t$ erasure decoding trials with the number of erasures increased by two for each trial. The complexity of GMD decoding can be reduced by using the results of the previous trial as an initialization of the current trial. This version of the GMD decoding algorithm is discussed in Ref. [28].

Another soft-decision decoding algorithm primarily employed for RS codes over small fields is the ordered statistics decoding (OSD) algorithm [29,30]. OSD uses bit-level reliabilities to reduce the parity
check matrix of the code to a form having unit weight columns for low-reliability bits. This is followed
by flipping low-reliability bits that do not satisfy the binary parity check equations thereby providing an
initial estimate of the transmitted codeword. A search is conducted over sets of high-reliability bits to
obtain a codeword closer to the received word. The size of the search set of high-reliability bits is typically
limited to two or three as the search complexity is exponential in the number of these bits.

Algebraic soft-decision decoding [31] is a recently developed algorithm for soft-decoding RS codes.
This technique assumes more than one received symbol and corresponding reliability information for
each transmitted symbol. Coordinate pairs are then formed using the code-evaluation point and the
received symbol. A bivariate (two-dimensional) finite-field polynomial is generated that passes through
each of these coordinate points. The number of times the polynomial passes through a particular
coordinate point is determined by the reliability of the received symbol. The curve is then factorized to
obtain the transmitted data polynomial. Architectures for algebraic soft-decoding have been proposed
in Refs. [32,33] and are presently an active area of research.

82.3 Viterbi Decoder Architectures

Convolutional codes find use in a wide range of applications such as magnetic disk-drive channels,
satellite, and in wireless communications. These codes are typically low-rate codes, i.e., $R = 1/2$ or $R = 1/3$
for most commonly employed codes. The Viterbi algorithm [34] is an efficient algorithm for decoding
convolutional codes. The complexity of a Viterbi decoder is inversely proportional to the code rate unless
the code rate is increased via puncturing.

Convolutional codes can be viewed as linear block codes with memory, i.e., the output of the linear
encoder depends not only on the present information symbols but also on previous information symbols.
Thus, a convolutional encoder is a finite-state machine (FSM) and the Viterbi algorithm is a particularly
efficient method of estimating the state sequence of an FSM from a sequence of noisy observations of
its output.

82.3.1 Convolutional Encoder

Consider an $(n_o, k_o, K)$ convolutional code where the encoder is an FSM with $K−1$ memory elements
that encodes a $k_o$-bit data symbol into an $n_o$-bit code symbol as shown in Figure 82.7(a). The parameter $K$
is called the constraint length of the code. At time index $k$, the $N = 2^k−1$ possible encoder states are
denoted as $s_i(k)$ where $i = 0, \ldots , N−1$ and the branch connecting states $s_i(k)$ with $s_j(k + 1)$ is denoted
as $b_{ij}(k)$. Note that $s_i(k)$ and $b_{ij}(k)$ are labels and not variables.

Given the current state $s_i(k)$ and an input symbol $u(k)$, the encoder transitions to a unique next state
$s_j(k + 1)$ and generates a unique output symbol $c(k)$. Hence, for an input sequence of $L$ data symbols
$u = (u(0), u(1), \ldots , u(L−1))$, the encoder starts from an initial state $s_0(0)$ and performs $L$ state transitions

FIGURE 82.7 Example of a convolutional encoder: (a) a (2,1,3) convolutional encoder with 2 memory elements
and modulo 2 adders, and (b) a trellis section resulting from state transitions of the encoders in (a).
while generating \( L \), \( n_x \)-bit code symbols \( c \triangleq (c(0), c(1), \ldots, c(L-1)) \). An efficient method of describing all possible state transitions in an encoder is by using a \textit{trellis diagram}. A section of the trellis of the encoder of Figure 82.7(a) is depicted in Figure 82.7(b), where the solid edges correspond to \( u(k) = 1 \) and dashed edges correspond to \( u(k) = 0 \). Note that the number of branches emerging from or ending in any state is \( 2^n \).

The received sequence \( y = (y(0), y(1), \ldots, y(L-1)) \) at the receiver is simply a noisy version of the transmitted code sequence \( c \). Thus, the convolutional decoding problem can be defined as estimating the input sequence \( u \) given the noisy sequence \( y \). Two major approaches for estimating the transmitted bits \( u(k) \) from \( y \) are: (1) maximum likelihood (ML) decoding, and (2) maximum a posteriori probability (MAP) decoding. MAP is known to be much more complex than ML decoding. Furthermore, the results of MAP and ML decoding are identical when the probability distribution of the input \( u(k) \) is uniform.

In 1966, Andrew Viterbi [34] proposed a computationally efficient procedure to implement ML decoding of convolutional codes. As a result, the Viterbi algorithm is almost universally applied whenever there is a convolutional code in a communication link. Hence, we will focus on the Viterbi algorithm and the ML decoding approach in this section.

Next, we provide a brief description of the ML decoding principle and then describe the Viterbi algorithm.

### 82.3.2 Maximum Likelihood Decoding and the Viterbi Algorithm

Maximum likelihood decoding involves the estimation of the transmitted symbol sequence \( u \) by determining the path in the trellis which maximizes the probability \( p(y | u) \). Hence, this approach is also called ML sequence estimation (MLSE). The ML principle is quite general though and can be applied equally well to the problem of estimating individual symbols, i.e., for symbol-by-symbol detection. For example, consider the case of ML symbol-by-symbol detection when the transmitted symbols \( c \) are \( \pm 1 \), and the received symbols are \( y = c + \eta \), where \( \eta \) is a zero mean additive white Gaussian noise (AWGN) with a probability density function (PDF) given by

\[
p(\eta) = \frac{1}{\sqrt{2\pi\sigma_n^2}} \exp\left(-\frac{\eta^2}{2\sigma_n^2}\right)
\]

where \( \sigma_n^2 \) is the noise variance. Clearly, \( p(y | c) \) is given by

\[
p(y | c = 1) = \frac{1}{\sqrt{2\pi\sigma_n^2}} \exp\left(-\frac{(y - 1)^2}{2\sigma_n^2}\right)
\]

\[
p(y | c = -1) = \frac{1}{\sqrt{2\pi\sigma_n^2}} \exp\left(-\frac{(y + 1)^2}{2\sigma_n^2}\right)
\]

Hence, to determine if \( p(y | c = 1) \geq p(y | c = -1) \), all that needs to be done is to check if \( (y - 1)^2 \leq (y + 1)^2 \). In other words, as \( y \) can be positive or negative, one needs to compute the Euclidean distance between the observation \( y \) and all possible transmitted symbol values (\( \pm 1 \) in this case) and choose as the ML result the transmitted symbol value that results in the smallest Euclidean distance.

Note that the appearance of the Euclidean distance metric in the example above is a direct consequence of the AWGN assumption with signal-independent noise. In some applications such as optical communications employing nonreturn to zero (NRZ) modulation, the noise variance for a “1” and a “0” are significantly different. In such cases, the distance metric needs to be modified accordingly. Furthermore, in situations where the noise is not Gaussian, there may not exist a distance metric and instead the PDFs themselves need to be estimated and employed in the decoding process.
The above description can be extended to the observed sequence $y$ and the code sequence $c$, where the latter is known to be the output of the encoder FSM that takes $u$ as the input. The decoder knows the number of states in the FSM and the state-transition matrix, but it does not know what state sequence has been traversed at any given time index. The job of the ML decoder is to determine the most likely state sequence traversed based on the observation $y$ from which it can then determine the most likely code sequence $c$ and more importantly, the most likely input sequence $u$. The ML decoder is said to execute an MLSE procedure.

As mentioned earlier, the Viterbi algorithm [34] was first proposed as an efficient procedure to compute the ML solution to the problem of decoding convolutional codes.

### 82.3.3 Viterbi Decoder Architectures

The Viterbi decoding algorithm determines the ML transmitted information sequence $u$. Recall that the receiver knows the encoder trellis and the noisy observations $y$. The Viterbi decoder consists of three types of computations: (1) branch metric computation implemented in a branch metric unit (BMU), (2) state/path metric computation implemented in a path metric unit (PMU), and (3) survivor path computation implemented in the survivor memory unit (SMU). Thus, a generic Viterbi decoder architecture will take the form shown in Figure 82.8.

We now describe each of the computations in more detail.

1. **Branch Metric Computation.** We compute the branch metric $B_{ij}(k)$ for the branch connecting states $s_i(k)\text{ and } s_j(k+1)$ and do so for every branch in the trellis. The branch metric is a measure of the probability of traversing the branch $b_{ij}(k)$ given that the encoder is in state $s_i(k)$ and the received value is $y(k)$.

   If the demodulator provides hard decisions such that $y(k)$ is binary, then the branch metric is the Hamming distance between $y(k)$ and the code symbol $c(k)$ associated with the branch. If $y(k)$ is a real number and the noise in the channel is AWGN, then the ML principle described in Section 82.3.2 results in the branch metric computation to be identical to the computation of the Euclidean distance given by

   $\sum_{l=1}^{n} (y_l(k) - c_l(k))^2$ \hspace{1cm} (82.18)

   where $l$ indexes the individual components of $y(k)$ and $c(k)$, i.e., the samples and bits of $y(k)$ and $c(k)$, respectively. The BMU has a feed-forward architecture and hence can be pipelined easily and therefore is usually not in the critical path.

2. **State/Path Metric Computation.** We compute the state/path metric $S_i(k)$ for each state in the trellis. The state metric $S_i(k)$ is a measure of the probability of the encoder arriving at state $s_i(k)$ given the observations $\{y(0), y(1), \ldots, y(k-1)\}$. In computing $S_i(k)$, the Viterbi algorithm employs the branch metrics. Hence, it has knowledge of the most likely path that leads to the state $s_i(k)$. This path is called the survivor path and hence the state metric is also called the survivor path metric or simply the path metric. The survivor path needs to be stored in memory.

   To understand the state metric computation, consider the two-state trellis shown in Figure 82.9 where the state metrics $S_0(k)$ and $S_1(k)$ corresponding to states $s_0(k)$ and $s_1(k)$, respectively, are employed as labels for the trellis nodes. The branches corresponding to $u(k)=0$ are dotted and the solid branches correspond to $u(k)=1$. The trellis branches are labeled with the corresponding branch
metrics. From Figure 82.9 we see that

\[
S_k(k + 1) = \min [S_k(k) + a + c, S_{k}(k - 1) + b + g, S_{k}(k - 1) + c + e, S_{k}(k - 1) + d + g]
\]

\[
= \min [\min (S_{k}(k) + a, S_{k}(k - 1) + c) + e, \min (S_{k}(k) + b, S_{k}(k - 1) + d)] + g
\]

\[
= \min [S_{k}(k) + e, S_{k}(k) + g]
\]

(82.19)

where \( S_{k}(k) = \min [S_{k}(k) + a, S_{k}(k) + c] \) and \( S_{k}(k) = \min [S_{k}(k) + b, S_{k}(k) + d] \). From Eq. (82.19), it is clear that the basic operation in determining the state metrics involves addition of the branch metric to the previous state metric followed by a comparison with other such products, and finally followed by a selection of the minimum of all possible path metrics. Therefore, the state metrics are computed as

\[
S_{k+1} = \min \{ S_{k}(k) + B_{j_{\text{dec}}}(k) \}
\]

\[
d_{j_{\text{dec}}}(k) = u(k - K + 1)
\]

(82.20)

where the min is over all \( 2^k \) states at time index \( k \) that are connected to state \( s_{k}(k + 1) \) and \( d_{j_{\text{dec}}}(k) \) referred to as the decision, is the value held by the earliest bit in the encoder shift-register. Note that the availability of the decision (earliest) bit makes it possible to determine the state at time index \( k - 1 \) from which the survivor path was extended to state \( s_{k}(k) \). This information is useful in tracing back the survivor path to generate the decoded bits. For each state at time \( k \), indexing the \( 2^k \) incoming branches with the earliest encoder bits at time index \( k - 1 \) enables the decision bit to also indicate the branch of the survivor path. Eq. (82.20) is referred to as the add-compare-select (ACS) operation and is implemented in an ACS unit (ACSU) that forms a key processing kernel in the PMU. The PMU is usually the throughput determining block of a Viterbi decoder and hence we will discuss the design of this block in somewhat greater detail.

The PMU can be designed in a state-parallel implementation where all the \( N \) states metrics are computed in parallel using \( N \) ACSUs, or serially or a hybrid. A direct implementation of an ACSU is shown in Figure 82.10(a) which includes a \( 2^k \)-input CS block. In this figure, \( S_{j}(k) \) for \( 0 \leq i < 2^{k-1} \) are the state metrics of the \( 2^k \) states connected to state \( j \). The \( 2^k \)-input CS block can be implemented as a tree of two-input CS blocks as shown in Figure 82.10(b) (for \( k_0 = 2 \)). The two input CS block compares the MSB of the two inputs and proceeds down to the LSB. If any two input bits differ, then the CS block flags a decision and selects the smaller of its inputs as the output. The critical path of the ACSU consists of a carry ripple in the adder (LSB to MSB) followed by a compare ripple (MSB to LSB). The critical path of the ACSU can be reduced by using redundant carry-save MSB first computation [35].

Two algorithmic techniques for increasing throughput are parallel processing and higher radix processing [36]. In parallel processing, the input sequence is divided into subblocks and more than one subblock is processed at a time. The other option for increasing throughput is to process more than one trellis section at a time. This process is termed higher radix processing. Figure 82.11, shows the concept of merging two trellis sections of the encoder shown in Figure 82.7(a). This leads to radix-4 processing.
The output code symbols on the branches going to the zeroth state are shown in the radix-4 trellis in Figure 82.11. It is clear that the complexity of BMU doubles in going from conventional radix-2 to radix-4 processing; twice as many branches now exist and each branch metric computation requires the processing of twice the number of inputs. Further, the number of comparisons in the PMU also doubles. However, the additions are performed outside the ACS loop leading to a net speed up of the PMU which in turn leads to a throughput increase compared to radix-2 processing. The achievable throughput increase has been observed to lie between 1 and 2 for radix-4 processing.

(3) Survivor path computation. Each path metric update in the PMU results in the extension of the survivor paths to each of the \( N \) states \( S_j(k) \) from one of the \( N \) states \( S_i(k) \). These survivor paths need to be stored and processed to determine the decoder output. Typically, as a rule of thumb, the survivor path length or the survivor memory depth \( L \) is chosen to be four to five times the constraint length \( K \). Doing so results in the survivor paths of all the states converging to a single path for time index smaller than \( k \) with a probability approaching one. This unique path is the maximum likelihood path and the input labels or the decisions \( d_{S_j}(k - l) \) \((l \geq 1)\) (see Eq. (82.20)) are the decoded outputs of the Viterbi decoder.

There are two main approaches to implementing the SMU: (1) trace-back [37] and (2) register-transfer [38]. Figure 82.12(a) shows a 4-state trellis with the branch labels indicating the decisions and the states being marked as shown. In this figure, the MSB of the state-index represents the earliest encoder bit and the LSB represents the most recent encoder bit. It is seen that the branch labels correspond to the MSB of the state from which the branch originates.

The survivor path memory in Figure 82.12(b) is organized in an \( N \) row-by \( L \) column format where each row corresponds to a state and each column corresponds to a time index in the range \( k \) to \( k - L \). Thus, in each time index, the vector of decisions from Eq. (82.20) are stored in the corresponding column. In trace-back, the survivor path of any state at time \( k \) is chosen to be traced back. The trace-back procedure involves recursively reading the contents of the survivor path memory shown in Figure 82.12(b) from time index \( k \) to \( k - L \). The address, which is essentially the row index of the memory, for time index...
The address sequence generated by this procedure is identical to the time-reversed state sequence generated by the forward recursion of the ACS unit.

Figure 82.12(c) shows the architecture of a register-transfer architecture of the SMU. In this approach, the survivor paths and not the vector of decisions, are stored in an L bit register (path register) for every state. At each time index \( k \), the survivor path gets updated based on the decision made by the ACSU. The most recent bit in the register for a state \( s_i(k) \) is always equal to \( d_s(k) \). However, the other bits in the path register may need to be replaced entirely with the contents of the path register of another state. This occurs if the survivor path for state \( s_i(k) \) is obtained by extending the survivor path from another state \( s_j(k-1) \). Register transfer is fast, but power hungry owing to the extensive data movement needed to update the survivor path register.

Hybrid approaches for designing the SMU [39] exist where trace-back and register-exchange approaches are combined. In general, the trace-back approach is suitable for decoders with a large number of states \( (N \geq 8) \) and low data rates (<2 Mb/s) while the register-transfer works well when the number of states is small and data rates are high (>10 Mb/s). Though most applications in the past fell neatly into these two categories, many modern-day applications such as ultra-wideband [40] have data rates in the 100’s of Mb/s along with a large number of states such as \( N = 64 \). Thus, the design of high-speed, low-power, large-state Viterbi decoders continues to be an important area of research [41,42].

82.3.4 The Soft-Output Viterbi Algorithm

The Viterbi decoder described so far generates the most likely transmitted bit sequence based on channel observations and the knowledge of the encoder state transition table. The soft-output Viterbi algorithm (SOVA) [43] provides the reliability of each bit in the decoded sequence in addition to the decoded sequence. Reliability of a bit \( u(k) \) is quantified by the log-likelihood ratio (LLR) which is given by

\[
L(u(k)) = \ln \left[ \frac{p_e(u(k))}{1 - p_e(u(k))} \right]
\]

where \( p_e(u(k)) \) is the probability of error of bit \( u(k) \). The LLRs are referred to as soft outputs. These soft outputs improve decoding performance in both serial and parallel concatenated decoders such as turbo decoders, which will be described in Section 82.4.
It is easy to show that the probability of the encoder state sequence being the same as the survivor path of state $s_i(k)$ is given by

$$p(s_i(k)) = e^{-S_i(k)}, \quad 0 \leq i \leq N$$

For $k_0 = 1$, the ACS unit decides between two paths, path 1 and path 2. Assume that path 1 is chosen. The probability of selecting the wrong survivor path, i.e., the path error probability is then given by

$$p_e(s_i(k)) = \frac{e^{-S_i(k)} + e^{-S'_i(k)}}{1 + e^{S'_i(k) - S_i(k)}}$$

where $\delta = S'_i(k) - S_i(k)$, $S'_i(k)$ is the state metric corresponding to path 2 in Eq. (81.20). Thus, $S_i(k) \leq S'_i(k)$. Hence, with a probability $p_e(s_i(k))$, the Viterbi algorithm can make errors in all the positions where the information bits $u(k)$ of path 2 differ from those of path 1. Using the path-error probability $p_e(s_i(k))$, the formula for bit-error probability $p_e(u(l), k)$ for bit $u(l)$ at time index $k$ is given by the recursion

$$p_e(u(l), k) = p_e(u(l), k - 1) (1 - p_e(s_i(k)) + (1 - p_e(u(l), k - 1)) p_e(s_i(k))$$

From Eq. (82.21) and Eq. (82.22), the LLR of $u(l)$ can be refined until $l \leq k - L$, i.e., the bit is decoded. The above recursion can also be applied directly in the log-likelihood domain in which case the recursion is then given by

$$L(u(l), k) \leftarrow f(L(u(l), k - 1), \delta)$$

where the function $f(L(u(l), k - 1), \delta) = \min(L(u(l), k - 1), \delta)$. It has been shown [44] that the hardware complexity of SOVA is roughly twice that of the Viterbi decoder.

### 82.4 Turbo Decoder Architectures

Turbo decoders are composed of two or more constituent soft-input soft-output (SISO) decoders, which correspond to the component codes employed in the transmitter, and an interconnection of these constituent decoders through an interleaver/deinterleaver [45,46]. The decoding algorithm employed in the constituent decoders is the MAP algorithm or SOVA, but it is well known that MAP-based turbo decoders outperform SOVA-based turbo decoders. The MAP algorithm provides the LLR of the transmitted code symbols $u(k)$. The LLRs are iteratively updated by the constituent decoders. However, the use of iterative processing results in a large computational and storage complexity, and hence high-power dissipation in the receiver. Therefore, low-power and high-throughput architectures for turbo decoders have recently been investigated [36, 47–64] for wireless and broadband applications. In this section, we first describe the algorithm and VLSI architecture of SISO MAP decoding and then present high-throughput and low-power turbo decoder architectures.

#### 82.4.1 MAP Decoder

In this section, the algorithm for the SISO MAP decoder is described, followed by a description of a baseline VLSI architecture for a SISO MAP decoder.

1. **The MAP Algorithm.** Unlike the Viterbi algorithm, the MAP algorithm determines each of the transmitted information symbols $u(k)$ independently by maximizing the a posteriori probability $P(u(k) | \gamma)$, where $\gamma$ denotes the received encoded bits used to determine $u_i$. The BCJR algorithm [48]
solves the MAP decoding problem efficiently and its log-domain version, i.e., the log-MAP algorithm has the advantage that it can be formulated in terms of additions instead of multiplications.

A log-MAP algorithm estimates the LLR of data symbol \( u(k) \) denoted as \( \Lambda(u(k)) \) defined below

\[
\Lambda(u(k)) = \ln \frac{p(u(k) = 1 | y)}{p(u(k) = 0 | y)} = \ln \frac{\sum_{u(k+1) = 1} p(u, y)}{\sum_{u(k+1) = 0} p(u, y)}
\]  

(82.23)

where the summations in the numerator and the denominator are over all trellis branches where \( u(k) = 1 \) and \( u(k) = 0 \), respectively, and the observation is made over \( L \) trellis sections, i.e., the sequence \( y = \{y(0), y(1), \ldots, y(L - 1)\} \). We can rewrite \( \Lambda(u(k)) \) as

\[
\Lambda(u_k) = \ln \sum_{u(k-1) = 1} e^{h_p(u, y)} - \ln \sum_{u(k-1) = 0} e^{h_p(u, y)}
\]  

(82.24)

Typically, one term will dominate in each summation leading to the following approximation

\[
\Lambda(u(k)) = \max_{u(k-1)} \ln p(u, y) - \max_{u(k-1)} \ln p(u, y)
\]  

(82.25)

The log-domain probability \( \ln p(u, y) \) can be further decomposed [49] as follows:

\[
\ln p(u, y) = \ln[p(s_i(k-1), y(0 : k-1))p(y(k), s_j(k)|s_i(k-1))p(y(k+1 : L-1)|s_j(k))]
\]

\[
= \ln p(s_i(k-1), y(0 : k-1)) + \ln p(y(k), s_j(k)|s_i(k-1))
\]

\[
+ \ln p(y(k+1 : L-1)|s_j(k))
\]

\[
= \alpha(s_i(k-1)) + \lambda(s_i(k-1), s_j(k)) + \beta(s_j(k))
\]  

(82.26)

where \( y(0:k-1) \) and \( y(k+1 : L-1) \) are the observed sequence before and after the \( k \)th trellis section. The first term \( \alpha(s_i(k-1)) \) is referred to as the forward metric and is equal to the probability that the trellis reaches state \( s_i(k-1) \) given the past observation \( y(0 : k - 1) \). The forward metric \( \alpha(s_i(k)) \) plays the same role as the state metric \( S_i(k) \) in a Viterbi decoder and is computed recursively as

\[
\alpha(s_i(k)) = \max_{s_j(k-1)} \{\alpha(s_i(k-1)) + \lambda(s_i(k-1), s_j(k))\}
\]  

(82.27)

where the max operation is performed over all states \( s_i(k-1) \) that are connected to state \( s_i(k) \). The \( \alpha(s_i(k)) \) update is performed recursively starting from \( s_i(0) \) and is referred to as forward iteration. The second term in Eq. (82.26) \( \lambda(s_i(k-1), s_j(k)) \), is referred to as the branch metric and is related to the probability that a transition from \( s_i(k-1) \) to \( s_j(k) \) occurs. The branch metric \( \lambda(s_i(k-1), s_j(k)) \) is computed from the channel output, noise statistics, and the error-free output of the branch connecting \( s_i(k-1) \) and \( s_j(k) \) at time \( k \). Note that each trellis section has \( 2^k \) possible transitions. The third term \( \beta(s_j(k)) \), is referred to as the backward metric and is equal to the probability that the trellis reaches state \( s_j(k) \) given the future observations \( y(k+1 : L-1) \). The backward metric \( \beta(s_j(k)) \) is computed recursively as

\[
\beta(s_j(k)) = \max_{s_j(k+1)} \{\beta(s_j(k+1)) + \lambda(s_j(k), s_j(k+1))\}
\]  

(82.28)

where the max operation is performed over all states \( s_j(k+1) \) that are connected to state \( s_j(k) \). The \( \beta(s_j(k)) \) update is performed recursively starting from \( s_j(L - 1) \) and is referred to as backward iteration.
**VLSI Architectures for Forward Error-Control Decoders**

Hence, the decoding process consists of three steps. First, the branch metrics in each trellis section are computed. Second, the forward and backward metrics \( \alpha(s_j(k)) \) and \( \beta(s_i(k)) \) are computed recursively via Eq. (82.27) and Eq. (82.28). Third, the LLR \( \Lambda(u(k)) \) is computed as

\[
\Lambda(u(k)) = \max_{u_{k-1}=0} \{ \alpha(s_j(k-1) + \lambda(s_j(k), s_i(k)) + \beta(s_i(k)) \} - \max_{u_{k-1}=0} \{ \alpha(s_j(k-1) + \lambda(s_j(k), s_i(k)) + \beta(s_i(k)) \} 
\]

(82.29)

It is known that the function \( \max \) defined as

\[
\max \{x, y\} = \max \{x, y\} + \ln(1 + e^{x-y}),
\]

(82.30)

can be employed instead of the max, in which case the performance of the log-domain MAP algorithm approaches that of the BCJR algorithm [49] to within 0.05 dB. The second term in Eq. (82.30) is referred to as the correction factor.

(2) MAP decoder architectures. Numerous architectures can be employed to implement the log-MAP algorithm [50]. However, the trellis sweep over all \( L \) observed symbols requires large memory to hold the forward and backward metrics until they are used in the LLR computation in Eq. (82.29). Hence, the sliding window log-MAP algorithm has become popular as it minimizes the metric storage requirements [49].

The sliding window log-MAP decoding algorithm is derived via the property that the forward and backward metrics \( \alpha \) and \( \beta \) converge after a few constraint lengths have been traversed in the trellis, independent of the initial conditions [49]. We refer to this property as the warm-up property and the warm-up period is assumed to have a duration of \( L \) symbols. Owing to the warm-up property, the state metrics (\( \alpha \) and \( \beta \)) computation can be partitioned into windows of size \( L \). Further, the computations in each window can be done in parallel. Figure 82.13 shows an example of a decoding flow where the warm-up property is employed only for computing backward metrics. The warm-up or initialization period is depicted by dashed lines and the computation period by solid lines. This warm-up property will be exploited later in deriving parallel and block-interleaved pipelined (BIP) architectures.

**FIGURE 82.13** Scheduling of state metric recursions for \( \alpha \) and \( \beta \) in the sliding window log-MAP. For simplicity, it is assumed that the computation and warm-up period equals \( L \). Shadowed region indicates the computation and storage of \( \beta \). The gridded region indicates the computation of \( \alpha \) followed by the computation of \( \Lambda(u(k)) \). Here, \( \beta^1 \) and \( \beta^2 \) are the first and second \( \beta \)-recursion outputs, respectively.
Figure 82.14 shows the VLSI architecture of a decoder whose data-flow graph is shown in Figure 82.13. The architecture has units for the computation of branch metrics (\( \lambda \)-unit), one forward recursion (\( \alpha \)-unit), two backward recursions and a buffer to store backward metrics \( \beta^1 \) and \( \beta^2 \) (\( \beta \)-unit), and the \( \Lambda \) metric processing unit (\( \Lambda \)-MPU). The computations in \( \lambda \)-MPU and \( \Lambda \)-MPU can be implemented in a feedforward manner and thus these do not limit the throughput. However, the forward and backward recursions are computed via an array of ACS kernels in a state-parallel manner. The ACS kernel for a MAP decoder in Figure 82.15 shows that the correction factor in Eq. (82.30) is implemented via a look-up-table (LUT) and state metric rescaling is employed to avoid overflows [47]. As is the case in a Viterbi decoder, it is the critical path delay of the ACS unit in Figure 82.15 that limits the throughput.

### 82.4.2 High-Speed MAP Decoder Architectures

In this section, we present techniques for improving the throughput of recursive data-flow graph present in the MAP decoder, in particular in the ACS kernel. First, we review existing techniques of parallel processing [51,52] and look-ahead transform [36,53]. Then, we present the block-interleaved pipelining (BIP) technique [54–56].

In general, pipelining or parallel processing becomes difficult for a recursive data-path. However, if the data is being processed in blocks and the processing satisfies the following two properties: (1) computation
between blocks are independent and (2) computation within a block is recursive, then, a block parallel processing architecture can be achieved. Further, if a block can be segmented into computationally independent subblocks, parallel processing can be applied at the subblock level. This leads to the high-throughput MAP decoder architectures presented in Refs. [51,52,54–56].

(1) Parallel Processing. Consider the recursive architecture in Figure 82.16(a). Note that the architecture in this figure cannot be easily pipelined or parallelized owing to the presence of the feedback loop. However, if a data block of length $B$ is processed independently of other blocks and the computations within a block can be segmented into computationally independent subblocks, then one can parallelize the architecture as shown in Figure 82.16(b), where the parallelization factor $M = 2$ and a block $X$ is divided into $M = 2$ subblocks, $X_1$ and $X_2$. It is obvious that the critical path is not affected and the throughput is increased by a factor of $M$ at the expense of a factor of $M$ increase in hardware complexity.

(2) Look-ahead transform. Another transform to achieve high-throughput for recursive data-flow graphs is look-ahead computation [53]. Look-ahead leads to an increase in the number of symbols processed at each time step as shown in Figure 82.17, where two symbols are processed in one clock cycle. If $x(n) = [x(2n - 1), x(2n)]$ and $y(n) = [y(2n - 1), y(2n)]$, then look-ahead results in the output being expressed as $y(n) = F(x(n), y(2(n - 1)))$. Note that $F(\cdot)$ will have a longer critical path delay than the original computation of Figure 82.16(a). However, it has been shown that the function $F(\cdot)$ can be optimized via logic minimization so that an overall increase in throughput can be achieved. For example, as mentioned earlier, in the context of Viterbi decoding, it has been shown that a $1.7 \times$ increase in throughput is feasible via radix-4 computation [36].

(3) BIP. The parallel architecture in Figure 82.16(b), where the level of parallelism is equal to 2, has two identical computation units processing two independent input symbols. Therefore, the hardware complexity increases linearly with the level of parallelism $M$. A comparatively area-efficient architecture is obtained by using the BIP technique proposed in Ref. [58]. First, the data-flow of Figure 82.16(b) is folded [53] on to a single computation unit as shown in Figure 82.18(a), where two independent computations are carried out in a single computational unit. Note that the resulting BIP architecture in this figure is inherently pipelined. Therefore, an application of retiming [53] (see Figure 82.18(b)) results in reduction of the critical path delay by a factor of two over that of the original architecture in Figure 82.16(a). It is clear that the retimed BIP architecture in Figure 82.18(b) leads to high throughput at the cost of a marginal increase in memory owing to pipelining latches when compared to the architecture in Figure 82.16(a).
82.4.3 High-Speed Turbo Decoder Architectures

The turbo code considered in this section is made up of two recursive systematic convolutional (RSC) encoders concatenated in parallel as shown in Figure 82.19. The bit sequences transferred from one encoder to the other are permuted by an interleaver. The decoder contains two SISO MAP decoders which are associated with the two RSC encoders as depicted in this figure. The decoding of the observed sequences is performed iteratively via the exchange of soft output information \( \Lambda(n(k)) \) between the constituent decoders. The decoding process is repeated iteratively until an appropriate stopping criterion is satisfied.

The turbo decoder can be implemented via a serial architecture as shown in Figure 82.20, where one SISO MAP decoder is time-shared. Hence, increasing the throughput of the SISO MAP decoder directly leads to an overall improvement in throughput of the turbo decoder.

High-throughput turbo decoder architectures are designed using the MAP decoder architectures presented in Section 82.4.2. Table 82.1 summarizes the key parameters for the parallel, BIP, and look-ahead BIP architectures for two codes with encoder polynomials \([5,7]\) \(L = 16\) and \([13,15]\) \(L = 32\), with constraint length (encoder memory) \(K = 3\) and 4, respectively. These parameters are obtained from
TABLE 82.1 Turbo Decoder Application Results

<table>
<thead>
<tr>
<th>Parameter</th>
<th>K</th>
<th>Parallel</th>
<th>BIP</th>
<th>Look-ahead BIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical path delay (ns)</td>
<td>3</td>
<td>13.994</td>
<td>7.126</td>
<td>13.011</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>17.767</td>
<td>9.793</td>
<td>15.517</td>
</tr>
<tr>
<td>Speed-up (η)</td>
<td>3</td>
<td>2</td>
<td>1.96</td>
<td>2.15</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>1.81</td>
<td>2.29</td>
</tr>
<tr>
<td>Normalized area</td>
<td>3</td>
<td>2</td>
<td>1.63</td>
<td>2.03</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>1.76</td>
<td>1.83</td>
</tr>
<tr>
<td>Measured area (mm²)</td>
<td>3</td>
<td>5.3(2.87/2.43)</td>
<td>4.32(2.87/1.45)</td>
<td>5.4(2.36/3.04)</td>
</tr>
<tr>
<td>(Memory/logic)</td>
<td>4</td>
<td>10.9(7.46/3.44)</td>
<td>9.62(7.46/2.16)</td>
<td>9.99(5.58/4.41)</td>
</tr>
</tbody>
</table>

82.4.4 Low-Power Turbo Decoder Architectures

The application of BIP, folding, and retiming reduces the critical path delay in the ACS kernel of MAP decoders with marginal area overhead. Subsequent application of voltage scaling can result in savings in power keeping the same throughput. A low-power turbo decoder architecture can be obtained by processing M subblocks of size B/M bits via subblock interleaved computations, where B denotes the block length of information bits. In this section, we present results related to achievable power savings in turbo decoders by combining voltage scaling and BIP.

To reduce power, we scale the supply voltage of the block-interleaved architecture such that the block processing time is made equal to that of the conventional architecture. The conventional architecture requires \(B + 2L\) cycles to process a block, where \(L\) is the warm-up depth. The proposed architecture requires

\[
M \times \left( \frac{B}{M} + 2L \right) = B + 2ML
\]

(82.31)
cycles for processing one block. Equating the block processing times of the two architectures we get

\[
\tau_{cri,p} = \tau_{cri,s} \times \frac{(B + 2L)}{(B + 2ML)}
\]

(82.32)

where \(\tau_{cri,p}\) and \(\tau_{cri,s}\) are the critical path delays of the block-interleaved and conventional architectures, respectively. Thus, we can reduce the supply voltage such that \(\tau_{cri,p}\) is equal to \(\tau_{cri,s} \times B + 2L/B + 2ML\).

Figure 82.21 depicts the critical path delay ratio, area overhead ratio, and power savings for \(K = 3\), \(B = 1024\), and \(L = 16\) for velocity saturation index values of \(a = 1\) and \(2\). It is observed that at a certain point there is no further power savings owing to the area overhead. Further, as the number of states \(\left(=2^{K-1}\right)\) is increased, the power savings decrease because the area overhead increases rapidly for large values of \(K\) (see Figure 82.22).

82.5 Low-Density Parity Check (LDPC) Decoder Architectures

LDPC codes were introduced by Gallager in his seminal work in 1963 [4], and have largely been ignored since then until the successful introduction of turbo codes by Berrou et al., in 1993 [65]. With this renewed interest, several researchers rediscovered LDPC codes and began to investigate codes on graphs in conjunction with...
iterative decoding. Long LDPC codes with iterative decoding have been shown to almost achieve capacity within a fraction of decibels [66], [67]. These discoveries have promoted LDPC codes as strong competitors to turbo codes in many communication and storage systems where high reliability is required.

This section focuses on the VLSI design aspects of LDPC decoders. After introducing LDPC codes in Section 82.5.1, two iterative decoding message-passing algorithms for LDPC codes are discussed in Section 82.5.2. In Section 82.5.3, several architectures for the message computation kernels employed by the decoding algorithms are presented and used in the decoder architectures presented in Section 82.5.4. Finally, Section 82.5.5 extends the discussion to repeat-accumulate codes.

82.5.1 LDPC Codes

An LDPC code is a linear block code defined by a sparse parity-check matrix $H_{m \times n}$ with $m$ parity-check equations on $n$ codeword bits with $k = n - m$ information bits. An LDPC code is typically described by
a bipartite Tanner graph [68] whose adjacency matrix is $H$ having $n$ bit nodes and $m$ check nodes corresponding to the $n$ columns and $m$ rows of $H$, respectively (see Figure 82.23). A bit node $u_i$ is connected to $r_j$ check nodes and a check node $v_i$ is connected to $c_i$ bit nodes, where $r_j$ and $c_i$ are called the node degrees. Equivalently, the $j$th column of $H$ has $r_j$ ones, and the $i$th row has $c_i$ ones. If all bit-node degrees and all check-node degrees are uniform, the code is said to be regular, otherwise it is called an irregular LDPC code. In general, the graph edges are defined randomly by a $p \times p$ “edge” permuter, where $p = \sum_{j=1}^{n} r_j$. Two graph parameters relevant to performance are the girth or length of the shortest cycle in the graph and diameter or maximum length of the shortest path in the graph. The code length $n$ and the randomness of the edge permuter play an important role in the decoder design of an LDPC code. The encoding complexity of LDPC codes is quadratic in the code length $n$.

Much of the research on LDPC codes has focused on explicit construction methods of bipartite Tanner graphs with large girth. Architecture-aware LDPC (AA-LDPC) codes are a class of structured LDPC codes with large girth oriented toward an efficient decoder implementation [69]. The parity-check matrix of an AA-LDPC code is divided into $S \times S$ submatrices, where each submatrix is either the zero matrix or a binary permutation matrix as shown in Figure 82.24(a). Equivalently, the bit nodes and check nodes in an AA-LDPC Tanner graph are grouped into clusters of size $S$ such that if one node from a cluster connects to a node from another cluster, then there exist distinct connections between all nodes from the same cluster.
both clusters (see Figure 82.24(b)). The connections between two clusters are done according to the permutation of the corresponding submatrix in \( H \).

### 82.5.2 LDPC Decoding Algorithms

LDPC codes are decoded iteratively using a suboptimal message-passing algorithm, called the sum–product algorithm [70–72], which closely approximates maximum likelihood decoding. The algorithm operates on the Tanner graph of an LDPC code by computing messages for every node and communicating these messages along the edges in the graph. These messages represent estimates of the transmitted bits together with a measure of the reliability of these estimates. The message associated with a transmitted bit \( u_j \) with noisy received bit \( \tilde{u}_j \) is given by the log-likelihood ratio, and is called the channel observation or intrinsic message. The bit message \( \mu_{u_j \rightarrow v_j} \) passed from bit node \( u_j \) to a neighboring check node \( v \in \mathcal{N}(u_j) \), where \( \mathcal{N}(u) \) is the set of all check nodes connected to \( u_j \), is proportional to the likelihood of bit \( u_j \) in favor of zero given the information obtained from check nodes other than \( v \) and is given by

\[
\mu_{u_j \rightarrow v} = \delta_{u_j} + \sum_{w \in \mathcal{N}(u_j) \setminus v} \mu_{w \rightarrow u_j}, \quad \text{for all } v \in \mathcal{N}(u_j) \tag{82.33}
\]

Similarly, the check message \( \mu_{v_j \rightarrow u_j} \) passed from check node \( v_j \) to a neighboring bit node \( u \) is proportional to the likelihood with which bit \( u \) satisfies check \( v_j \) given the likelihood in favor of zero of the remaining bits checked by \( v \), and is given by

\[
\mu_{v_j \rightarrow u} = \psi^{-1} \left( \sum_{w \in \mathcal{N}(v_j) \setminus u} \psi(\mu_{w \rightarrow v_j}) \right), \quad \text{for all } u \in \mathcal{N}(v_j) \tag{82.34}
\]

where \( \psi(x) = (\text{sgn}(x), |\psi(x)|) = (\text{sgn}(x), -\log \tanh(\frac{|x|}{2})) \), and \( \text{sgn}(x) = -1 \) if \( x < 0 \), +1 otherwise. The sum \( \psi(x) + \psi(y) \) is defined as \( \psi(y) \cdot \psi(x) = (\text{sgn}(xy), |\psi(x)| + |\psi(y)|) \).

Owing to the presence of cycles in a Tanner graph, different scheduling of node computations yield nonequivalent message-passing algorithms. The two main scheduling techniques for message-passing are the two-phase message-passing (TPMP) and the turbo-decoding message-passing (TDMP) algorithms. In the TPMP algorithm [4], a decoding iteration is divided into two rounds of computations, one pertaining to bit nodes and the other to check nodes. In each round, all nodes are activated in parallel, sending new outgoing messages to all neighboring nodes using messages received in the previous round. Decoding proceeds until the codeword has zero syndrome or a predefined number of iterations is reached. The algorithm requires saving all intermediate messages (equal to twice the number of nonzero entries in \( H \)) at both rounds every iteration. Moreover, newly computed messages in a round of computations do not participate in further message computations until the decoding iteration is over. The advantage of the TPMP algorithm is that it is inherently parallel.

In the TDMP algorithm [73], updated messages are used directly within an iteration to compute new messages, and hence refined estimates spread faster among neighboring nodes in the graph, speeding up the convergence rate of the algorithm. Further, new check messages become directly bit messages within the same iteration, hence both bit and check messages merge into a single type of messages resulting in significant memory savings. The algorithm is based on decoding the rows of \( H \) sequentially. Extrinsic messages generated from decoding earlier rows are used as input prior messages in decoding subsequent rows. Each row \( i \) in \( H \) is associated with a vector of extrinsic messages \( \lambda_i \) corresponding to the nonzero entries in that row. For each bit the sum of all messages generated by the rows in which that bit participates are stored in the vector \( \gamma \) of \( n \) posterior messages. Decoding the \( \delta \)th parity-check row involves first reading
the extrinsic messages $\lambda_i$ and the posterior messages $\gamma(I_i)$, where $I_i$ denotes the set of indices of the ones in row $i$. Next, prior messages $\rho = \{\rho_1, \ldots, \rho_n\} = \gamma(I) - \lambda$ are computed by subtracting $\lambda$ from $\gamma(I)$ to reduce correlation between messages generated from decoding earlier rows. Output messages $\Lambda' = [\Lambda'_1, \ldots, \Lambda'_n]$ are then computed using Eq. (82.34) with $\rho$ as input

$$\Lambda'_i = \psi^{-1} \left( \sum_{j=0}^{w_i} \psi(\rho_j) \right), \quad i = 1, \ldots, n, \quad j = 1, \ldots, w_i$$  \hspace{1cm} (82.35)$$

where $w_i$ is the number of ones in row $i$. Equivalently, $\Lambda'_i$ can be computed more efficiently as

$$\Lambda'_i = Q(\rho_1, \rho_2, \ldots, \rho_{j(i-1)}, \rho_{j(i)}, \ldots, \rho_n), \quad i = 1, \ldots, n, \quad j = 1, \ldots, w_i$$  \hspace{1cm} (82.36)$$

The $Q$-function is defined recursively as

$$Q(x_1, x_2, \ldots, x_n) \triangleq Q(...(Q(Q(x_1, x_2), x_3), \ldots), x_n)$$  \hspace{1cm} (82.37)$$

and $Q(x_1, x_2)$ approximates the difference between two logsum operations [74]

$$Q(x_1, x_2) = \log(e^{x_1} + e^{x_2}) - \log(e^{x_1} + e^{x_2} + 1)$$

$$= \max(x_1 + x_2, 0) - \max(x_1, x_2) + \max \left( \frac{5}{8} - \frac{1}{4}, 0 \right) - \max \left( \frac{5}{8} - \frac{1}{4}, 0 \right)$$

Finally, the vector $\Lambda'$ replaces the old extrinsic messages $\lambda_i$ and the posterior messages for the bits located at positions indicated by $I_i$ are updated by adding $\Lambda'$ to $\rho$: $\gamma(I_i) = \rho + \Lambda'$. These steps constitute a decoding subiteration, while a round of subiterations over all rows of $H$ constitutes a decoding iteration.

The TPMP algorithm requires memory storage for $2\sum_{j=1}^{n} r_j + n$ messages, while the TDMP algorithm requires storage for only $\sum_{j=1}^{n} r_j + n$ messages, resulting in a memory savings of $\sum_{j=1}^{n} r_j / (2\sum_{j=1}^{n} r_j + n) \times 100\%$. For uniform column degrees $r$, this amounts to $r/(2r + 1) \times 100\%$. Moreover, the TDMP algorithm requires on average 50% less iterations to converge for moderate to high SNR compared to the TPMP algorithm as determined by simulations [73].

### 82.5.3 Architectures for Message Computation Kernels

The message computation kernels of Eq. (82.33) and Eq. (82.34) compute distinct messages to all the neighbors of each node in a Tanner graph. These kernel computations from a source node to a neighboring destination node can be implemented by first computing the total sums in Eq. (82.33) and Eq. (82.34) and then subtracting out the component pertaining to the destination node to reduce correlation between messages. Figure 82.25(a) and 82.25(b) show serial dataflow graphs implementing Eq. (82.33) and Eq. (82.34) assuming a node having six neighbors. The latency of these serial architectures for a general node degree $d$ is equal to $d + 1$. These message computation kernels are sometimes referred to as constituent SISO decoders. Figure 82.25(c) shows a two-way stack-based architecture implementing kernel equations (82.36) and (82.37) of the TDMP algorithm using the $Q$-function whose logic schematic is shown in Figure 82.25. A pair of $Q$-function blocks compute the recursions in Eq. (82.37) both from the left ($\rho_j$ up to $\rho_i$) and the right ($\rho_j$ down to $\rho_i$). Intermediate results from the left and the right are pushed onto a pair of stacks until both recursions have reached the half way point. After that, the stored results from the left (up to $\rho_i - 1$) are popped from the stack and fed together with the results of the running recursion from the right into a new $Q$-function block that computes the output messages $\Lambda_3, \Lambda_2,$
Similarly, the stored results in the right stack are combined with the results of the running recursion from the left to generate output messages \( \Lambda_4, \Lambda_5, \Lambda_6 \). The latency of the architecture in Fig. 82.25(c) is 6 clock cycles. In general, the latency is \( d \) clock cycles.

Figure 82.25(d) shows a parallel dataflow graph using 12 \( Q \)-function blocks. The upper blocks process recursion (82.37) from the left, while the lower blocks process the recursion from the right. The middle row of blocks combines the results from both recursions to generate output messages. Note that the inputs must be skewed for proper operation. In general, this parallel architecture requires \( 3(d - 1) \) \( Q \)-function blocks.

### 82.5.4 LDPC Decoder Architectures

LDPC decoder architectures are typically classified as parallel, serial, or partly-parallel (scalable) architectures with respect to allocating resources for message computation and message communication. A parallel LDPC TPMP-decoder simply implements the Tanner graph of the code, with bit function units (BFUs) allocated to compute bit messages, check function units (CFUs) to compute check messages, and a physical interconnect of wires that connect these units according to the edge permuter as shown in Figure 82.26(a). To sustain the high bandwidth of message communication between function units, the edge permuter must be implemented as a physical interconnect of wires. While such decoder fully exploits the inherent parallelism of the TPMP algorithm achieving high-throughput with low-power consumption, it is constrained by chip area and interconnect complexity, and hence does not scale gracefully with code length. Long on-chip interconnect wires present implementation challenges in terms of placement, routing, and buffer-insertion to achieve timing closure. For example, the average interconnect wire length of the rate-1/2, 1024-bit LDPC decoder of Ref. [75] is 3 mm using 0.16 \( \mu \)m CMOS technology, and has a chip area of \( 7.5 \) mm \( \times \) 7 mm of which only 50% is utilized owing to routing congestion.
VLSI Architectures for Forward Error-Control Decoders

In contrast, in serial decoders [76] computations are folded onto a small set of bit and CFUs that communicate through memory instead of a complex interconnect as shown in Figure 82.26(b). Each function unit requires separate read/write networks with complex control to access messages in memory. The BFUs read check messages and generate updated bit messages, while CFUs read bit messages and generate updated check messages. Computations proceed until all bit node and check node operations complete. To avoid overlapping messages in memory, two memory modules are used for storing bit messages where one module is read while the other is being written, and similarly for check messages. At the end of every iteration, the read and write order from/to the memory modules is swapped. Serial decoders require a substantial memory overhead that amounts to four times the number of nonzero elements in $H$, and their throughput is constrained by the number of function units and read/write networks to access memory. The advantage of serial decoders is their small area.

Partly-parallel or scalable decoders combine the high-throughput characteristics of parallel decoders and the area efficiency of serial decoders. In the following, partly-parallel TDMP decoders for AA-LDPC codes will be described assuming a regular LDPC code of length $n$ having check and bit node degrees $r$ and $c$, respectively, and whose $H$ matrix is decomposed into $S \times S$ submatrices with $D$ block rows and $B$ block columns. Referring to Figure 82.24(a), the $S$ rows of ones in each row of submatrices in an AA-LDPC matrix $H$ are nonoverlapping, and hence can be processed in parallel using $S$ SISO decoders. Decoder $s$ processes row $s$ in each row of submatrices, for a total of $D$ rows, and maintains its extrinsic messages in a local $\lambda$-memory as shown in Figure 82.27. Posterior messages are stored in a global $\gamma$-memory and communicated in parallel to and from the decoders using a network that implements the factored edge permuter. The advantage of the AA structure is evident in that it enables an efficient implementation of the permuter using a programmable multistage interconnection network of switches (e.g., Benes network [77]). The switches are programmed to route the various permutations when processing the rows of submatrices in $H$ using $\pi$-memory. The network in general has $2 \log(S) - 1$ stages, and each stage contains $S/2 \times 2 \times 2$ switches. The parameter $S$ is a parallelism factor that determines the throughput $\Theta$ of the decoder. The total number of $\lambda$-messages that need to be stored is $cSD = c(n - k)$, while the total size of $\gamma$-memory is $n = BS$ messages. The size of $\pi$-memory is $\frac{1}{2}(2 \log S - 1)$ bits. The SISO decoders complete one pass over all rows of $H$ (one iteration) in $cD$ clock cycles assuming serial SISO decoders similar to Figure 82.25(c). For $I$ iterations and clock frequency $f$, the throughput attained by the decoder is

$$\Theta = \frac{nf}{cD} = \frac{fS}{rI} \text{ bits/s}$$

Note that the architecture shown in Figure 82.27 can decode any AA-LDPC code with the same parameters $n, r, c$, and $S$, but having different structure and location of permutation submatrices in $H$. © 2006 by CRC Press LLC
82.5.5 Extensions to Repeat-Accumulate (RA) Codes

RA codes [78] are a subclass of LDPC codes that can be encoded in linear time. Formally, an RA code is a linear block code composed of a repetition code concatenated through a permuter with a rate-1 memory-1 convolutional code. An information message is encoded by repeating each bit \( r \) times, and then accumulating the repeated bits as shown in Figure 82.28(a). Typically, the information bits are transmitted together with the parity bits in an RA code. Equivalently, an RA code can also be defined algebraically as the null-space of a sparse parity-check matrix as shown in Figure 82.28(b). The left portion of \( H \) corresponds to the information bits with ones distributed according to the repetition code. The right portion of \( H \) represents the parity bits with ones randomly distributed depending on the permuter.

\[
H = \begin{bmatrix}
\mathbf{H}_{n \times k} & \mathbf{H}_{n \times n} \\
\end{bmatrix}
\]

FIGURE 82.27 TDMP decoder architecture.

FIGURE 82.28 RA codes: (a) encoder, and (b) parity-check matrix.
to the permutation structure. The right portion corresponds to the parity bits (accumulator) having ones on the main diagonal and the first lower off-diagonal. Given the sparse-matrix representation of RA codes, the previous discussion on LDPC codes and their decoder architectures applies to RA codes as well. Similarly, AA RA codes can be defined by decomposing the parity-check matrix into permutation submatrices and the TDMP algorithm naturally applies. However, the parity portion needs some modification. Instead of accumulating parities using adjacent check nodes in the Tanner graph, parities are accumulated using checks from adjacent clusters in parallel by connecting the rightmost check node cluster to the leftmost parity node cluster. The reader is referred to Ref. [79] for more details.

82.6 Conclusions

The subject of coding theory and ECC architectures is an active area of research today and is expected to remain so for the foreseeable future. New applications of communication systems combined with the unrelenting scaling of feature sizes in modern semiconductor processes makes for a rich interplay between the areas of coding theory and VLSI architectures.

While Moore’s Law has enabled the widespread deployment of communications systems, there are questions regarding the ability of the semiconductor industry to scale feature sizes well into the nanometer regime in a cost-effective manner. These questions arise from the emergence of nonidealities such as noise, process variations, soft errors, and many others, in modern silicon process technology. Hence, system-on-a-chip subsystems such as datapath and control (computation), buses (communication) and memory (storage) are beginning to resemble noisy communication networks. This fact points to a unique opportunity for extending and applying ECC and ECC-type techniques to enable the design of reliable and efficient ICs in general and communication ICs in particular. Examples of such research can be found in [80]–[82] where communication-centric concepts such as equalization, coding, and joint equalization and coding have been applied to on-chip computation and communication. This is an area replete with possibilities and can lead to an era where coding theory extends if not rescues Moore’s Law from obsolescence.

References


