Adaptive Forward Error Correcting Decoders: A Survey of Architectures and Implementations

Sergei Sawitzki
FH Wedel (University of Applied Sciences)
Wedel, Germany
e-mail: Sergei.Sawitzki@fh-wedel.de

***Abstract*—This work describes adaptive designs of decoders for several forward error correction codes. After a brief introduction to the corresponding code family and basic decoder architecture, several possibilities of multi-standard implementation of such decoders are discussed. In the first scenario, decoders capable of decoding the codes with different parameters belonging to the same family are introduced. The results suggest, that the hardware overhead caused by the additional flexibility is – in most cases – as low as 5 – 10% additional silicon footprint while the throughput is reduced by 2 – 26 % compared to the implementation based on the fixed code parameters. In the second scenario, reconfigurable designs of multi-family decoders are discussed. Since some parts of the data-path and internal memory can be reused for different decoders, silicon area savings of more than 40 % are achievable compared to the overall chip area costs of the individual decoder implementation per code family. The overall usefulness of such reconfigurable decoder designs depends on the application case, for instance, the throughput requirements or the necessity to process data streams encoded with different codes in parallel. The figures reported herein summarize and extend some previous work known from literature, as well as research carried out by the author.**

Keywords-forward error correction; FEC decoders; adaptive decoder architectures; convolutional codes; Viterbi algorithm; Reed-Solomon codes; turbo codes; LDPC codes; multi-standard FEC architectures.

I. Introduction and Related Work

Most of the modern digital communication and data storage standards enforce the use of different error correcting codes. Since in most cases the error correction process is carried out by the receiver – without resending or rereading the data – these codes are usually referred to as Forward Error Correcting (FEC) codes and the respective hard- or software units are called FEC encoders and decoders. Since the code used in particular storage or communication system is usually selected in such a way that it addresses the specific noise characteristics of the channel (memoryless vs. with memory, single vs. burst errors etc.), the more general term *channel code* is often used in the same context synonymously.

The FEC codes most widely used nowadays are belonging to one of the following code families: convolutional codes, Reed-Solomon (RS) codes, Low-Density Parity-Check (LDPC) codes, and turbo codes. Some standards specify varying codes for different data rates or combine codes from different families within one run to address more of the specific channel properties like cross-talk and fading improving the overall bit error rate.

Over the last few years, mobile devices got significantly more flexible. Modern mobile phones and other wearable devices support numerous communication interfaces providing the user with almost unlimited connectivity. Even the low-price devices nowadays support wireless LAN, different mobile internet standards, and Bluetooth, just to name a few. At least some of them rely on the same kind of FEC algorithms, although the specific parameters like code generator polynomials, coding rates, constraint length, block sizes, and so on may vary even within the same standard or application. To address this issue, adaptive Viterbi and RS decoders will be discussed below and compared to the straight-forward implementation for the single code instance. Such comparison will provide a glimpse of the overhead introduced by the additional flexibility.

On the other side, some computations carried out during the decoding process of different code families are quite similar. In addition, some data need to be temporarily stored either throughout the complete decoding procedure or, in case of LDPC and turbo codes, between adjacent decoding iterations resulting in specific memory requirements. This poses the question, if at least some parts of the silicon dedicated to the data-path and memory can be reused among different standards reducing the silicon footprint and improving the power balance compared to the straight-forward implementation of every single decoder. This question will be addressed in the discussion of combined Viterbi-turbo and LDPC-turbo decoders.

The applicability and advantages of the adaptive FEC decoder implementation depend on the use case, e.g., the necessity to process several data streams encoded with the same code or with different codes in parallel or the option to switch between different data rates. This should be kept in mind during the discussion of the different design options as will be addressed below in more detail. In general, this contribution is the revised and significantly extended version of the conference paper [1] published previously by the same author.

In principle, FEC decoding algorithms can be implemented both in software and hardware. However, given the fact that most of them are quite computationally intensive, software-based solutions are usually lagging behind the high throughput requirements imposed by the majority of contemporary communication and data storage standards. For instance, iterative algorithms involved in the decoding process of turbo and LDPC codes may require as many as 1 500 machine instructions per bit (or even more depending on the number of decoding iterations) [2]. Running such decoders at the data rates of several hundreds of megabits per second requires computational power, which is

beyond the capabilities of even most powerful microprocessors. For this reason, the scope of this paper is limited to the dedicated hardware architectures, i.e., direct mapping of the FEC decoders to silicon. Alternative approaches, which are less flexible than the software solution but still offer at least some properties of the architectures built around the instruction set paradigm, are Application-Specific Instruction-set Processors (ASIP) [3][4] and Networks-on-Chip (NoC) [5]. These are beyond the scope of this contribution. A comprehensive overview concerning the scalability issues and multi-standard capabilities of different FEC decoders is provided in [2][6][7]. Based upon these studies, some of the results are re-evaluated, supported by new findings and extended herein.

The rest of this paper is organized as follows. Section II introduces an adaptive implementation of the Viterbi decoder and compares several designs known from literature. In Section III, the additional costs of a reconfigurable design for a Reed-Solomon decoder compared to the appropriate single-code implementation are discussed. Section IV summarizes the general options of combining decoders for different code families within one design and analyzes the architectures of reconfigurable Viterbi-turbo and LDPC-turbo decoders in more detail. In Section V, architectures of all presented decoder families are briefly reviewed. Finally, Section VI provides a summary of this work and draws some conclusions concerning adaptive and multi-family FEC decoder implementation showing some directions for future research.

The discussion of every code family starts with the description of the key aspects relevant for the decoder implementation. For the readers interested in more detailed theoretical background, some references are provided where appropriate.

II. Viterbi Decoders

Convolutional codes were introduced in 1955 [8] and became one the most widely applied FEC code family. The following subsections describe the structure and the parameters of the code as well as the mapping of the decoding algorithm to hardware finishing with the discussion of the additional costs imposed by an adaptive implementation.

A. Code Structure

Convolutional encoder passes the incoming data stream through a shift register. The input of the first storage element as well as the outputs of several storage elements of this shift register are combined using an XOR operator. Particular XOR operators are selected to produce the output data streams. The number of delay stages in the shift register including the input of the first delay stage is referred to as *constraint length* K. Larger K values provide better error correction capabilities (at the cost of the increasing decoder complexity). Constraint lengths between 5 and 11 are common in real-world applications.

If the encoder does not have any feedback structures, i.e., no outputs of the delay stages are fed back to the input via XOR operators, it is called non-recursive. Such convolutional encoder can be seen as Finite Impulse Response (FIR) filter. The output

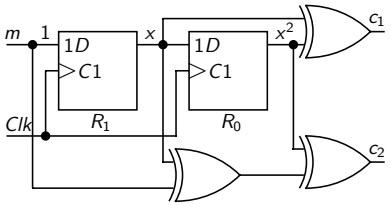


Figure 1. Simple convolutional encoder (non-recursive, non-systematic).

is the discrete convolution of the input data with the impulse responses (one for every output stream) of this FIR in the time domain, hence the name of the code [9, pp. 455 – 456]. In almost all applications of the non-recursive codes, the original input does not appear in the encoded message. In terms of the coding theory such code is classified as non-systematic. Recursive systematic convolutional codes will be discussed in the Subsection IV-A in the context of the turbo codes.

The number n of the output data streams defines the code rate $R = 1/n$. For the codes used in most communication standards 2 to 4 output bits are produced for every input bit, thus the code rate R lies between 1/2 and 1/4. Depending on the required error correction performance, it is possible to adapt the code rate $1/n$ of the initial code (called *mother code*) to different values by puncturing, i.e., removing some encoded bits before transmission and replacing them afterwards with dummy bits at the receiver before decoding. Higher code rates reduce the performance of the code.

The particular codes differ depending on which bits from the sequence stored in the shift register are involved in the XOR calculation. This information is provided in the form of the so-called generator polynomials g, one for every output. Figure 1 shows an example of a simple convolutional encoder with $K = 3$, $R = 1/2$, $g_1 = 011$ and $g_2 = 111$. As can be seen, the generator polynomials are often specified by the binary or octal representation of their coefficients. The encoder is a Finite-State Machine (FSM) whose output depends on g. Recording all possible state transitions of this FSM over time creates a directed graph called *trellis diagram*. As illustrated in Figure 2 (for the code produced by the encoder in Figure 1), every input sequence corresponds to the unique path through this graph, while every state transition on this path generates one output symbol. Since the decoder only sees the output sequence, while the input sequence and the corresponding state transitions remains inaccessible, the underlying abstraction is called Hidden Markov Model (HMM).

B. Decoding Algorithm

The decoding process is based on the maximum-likelihood (MLL) principle. The decoder's task is to find the state transition sequence which the encoder FSM most likely underwent, based on its output sequence, which in most cases is corrupted by noise during the transmission. Afterwards, the state transitions can be mapped to the corresponding input data sequence restoring the original message. In graph theory, this

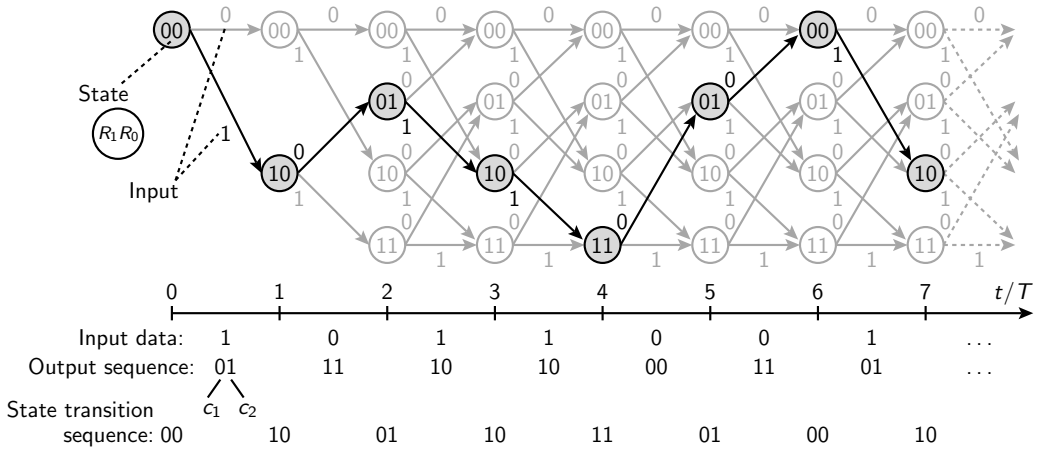


Figure 2. Trellis diagram.

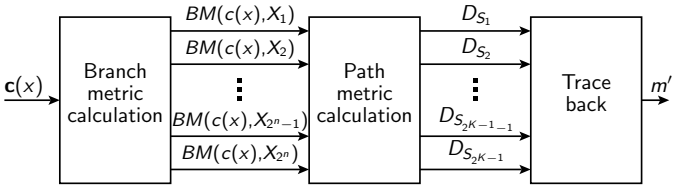


Figure 3. Top view architecture of the Viterbi decoder.

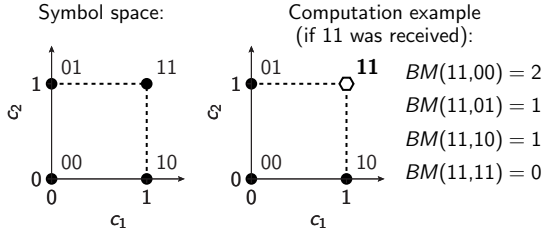


Figure 4. Branch metric example.

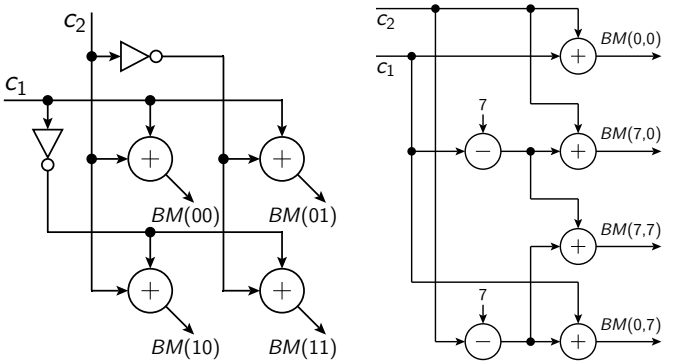
problem can be framed as finding the path through the trellis diagram that minimizes the cumulative error metric.

The most widely used decoding algorithm for convolutional codes was introduces in 1967 by Andrew Viterbi and is based on dynamic programming [10]. As shown in Figure 3, a Viterbi decoder consists of the three major parts, which are executed consecutively and are called branch metric calculation, path metric calculation, and trace-back.

In the first stage of the decoding process, every received symbol is compared to all legal output symbols of the encoder. The result is called Branch Metric (BM). In the simplest case of the so-called hard-input decoding, where every input symbol to the decoder is considered to be n bits wide, BM is the Hamming distance as shown in Figure 4 for the case $n = 2$. However, after the modulation and analog-digital conversion every bit composing any code symbol is mapped to an integer number represented by several bits. This case is referred to as soft-input decoding with the input bit-width b (resulting in $b \times n$ bits per code symbol). Table I shows different representation

TABLE I. INPUT FORMATS FOR SOFT-INPUT DECODING WITH $b = 3$.

Interpretation	Two's complement	Signed magnitude	Unsigned magnitude
Strongest 0	011	011	000
	010	010	001
	001	001	010
Weakest 0	000	000	011
Weakest 1	111	100	100
	110	101	101
	101	110	110
Strongest 1	100	111	111

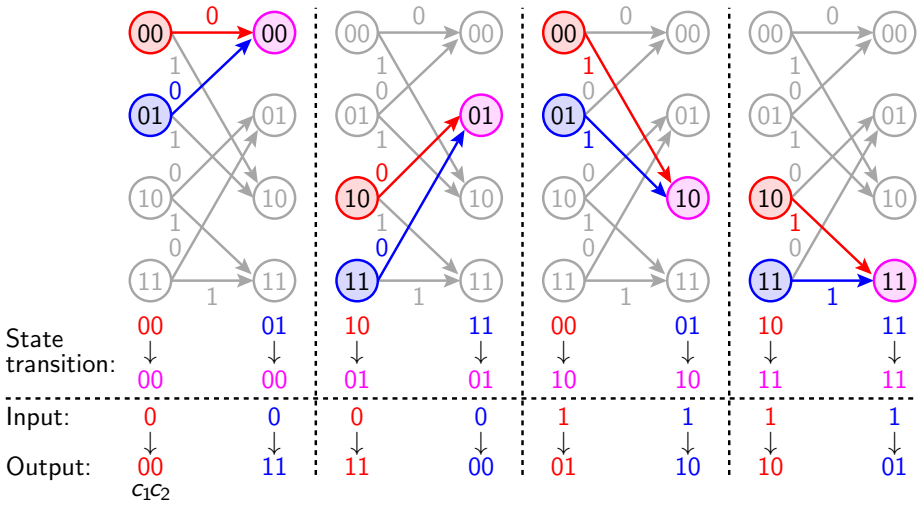


(a) Hard-input (Hamming distance) [11]

(b) Soft-input for $b = 3$ (Manhattan distance, unsigned magnitude)

Figure 5. Branch metric computation.

conventions for soft-bits. Note that independent of the format, hard-input symbols can be generated from the soft-inputs by keeping the Most Significant Bit (MSB) and removing the rest. Soft-input decoding provides better error correction performance at the cost of implementation complexity. For example, other measures like Manhattan or (squared) Euclidean distance are needed to represent the branch metrics. The number of BM values to be calculated per received symbol depends on the code rate, since every additional output bit at the encoder doubles the number of legal output symbols, and the decoder must compare each received symbol against all 2^n possibilities.



$$
\begin{aligned}
PM_{t+1}(00) &= \min(PM_t(00) + BM(x, \overbrace{(0,0)}^{c_1,c_2}), PM_t(01) + BM(x, \overbrace{(1,1)}^{c_1,c_2})) \\
PM_{t+1}(01) &= \min(PM_t(10) + BM(x,(1,1)), PM_t(11) + BM(x,(0,0))) \\
PM_{t+1}(10) &= \min(PM_t(00) + BM(x,(0,1)), PM_t(01) + BM(x,(1,0))) \\
PM_{t+1}(11) &= \min(PM_t(10) + BM(x,(1,0)), PM_t(11) + BM(x,(0,1)))
\end{aligned}
$$

Figure 6. Path metric calculation.

Figure 5 shows the branch metric calculation for hard- and soft-input decoding.

In the second stage, branch metrics are used to calculate the Path Metrics (PM). One PM needs to be calculated for every state of the encoder. All path metrics are set to zero in the beginning and updated with every processed symbol. The corresponding operation is called Add-Compare-Select (ACS) and is the computational core of the decoding process. Every ACS unit consists of two adders, one comparator and one multiplexer. In addition, one register is required per unit to store the actual PM value. As the number of ACS units is equal to the number of states of the encoder, it can be calculated as 2^{K-1}, i.e., it increases exponentially with the constraint length of the code. The result of the ACS calculation is the new path metric and the binary decision, which state transition the encoder underwent while the corresponding output symbol was generated. This decision is stored for every processed symbol and every ACS unit in form of a single bit called *decision bit*. The decision bits accumulated over time represent several paths through the possible state transition sequences of the encoder with every path corresponding to the certain input sequence. At the same time, the actual path metric of every state stores the cumulative error of the current path leading to this state as the sum of single BM values calculated in the previous stage. Figure 6 illustrates the path metric calculation for the code produced by the encoder in Figure 1.

With every processed symbol, new BM values are added to the actual PM values, thus path metrics are monotonically increasing, requiring normalization since the bit-width of the registers storing the actual values is limited. This can be achieved by shifting every PM to the right by one or more

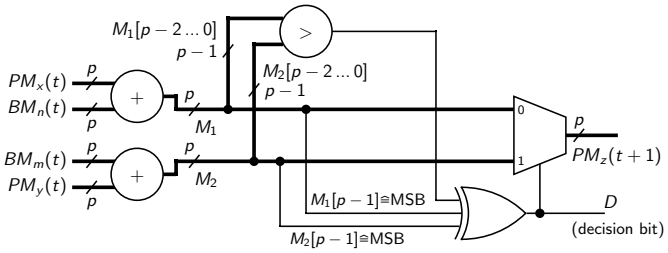


Figure 7. ACS unit with modulo-based normalization (adapted from [13]).

bits once any PM reaches the maximum value representable within the reserved bit-width, thus avoiding overflow. The more efficient approach, however, is based on modulo normalization. The comparison operation within the ACS unit calculates the difference between two sums of a path metric with a branch metric. It can be proven, that this difference has the upper bound determined by the input bit-width, the constraint length, and the coding rate [12][13]:

$$P_{\max} = \left(2^b - 1\right) \cdot \frac{K}{R}. \tag{1}$$

The essence of the modulo normalization is setting the bit-width p for the PM representation to $\lceil \log_2(2 \cdot P_{\max}) \rceil = \lceil \log_2(P_{\max}) + 1 \rceil$. By doing this, the overflow can be ignored. Instead, the result of the comparison is inverted, if the MSBs of the compared values differ (which means that the smaller value underwent an overflow and is actually greater). Hardware implementation of the ACS unit with modulo-based normalization logic is shown in Figure 7.

The final stage of the Viterbi decoding algorithm is called trace-back. During this stage the decision bits produced by the

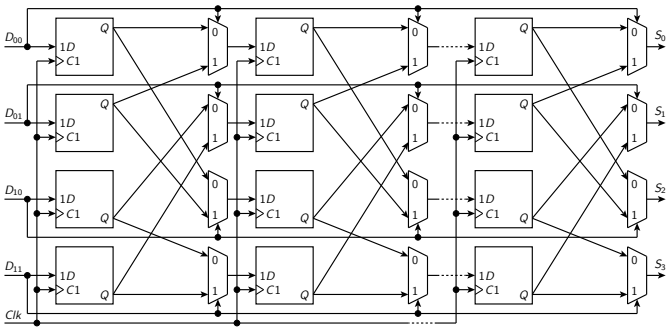


Figure 8. Trace-back by register exchange.

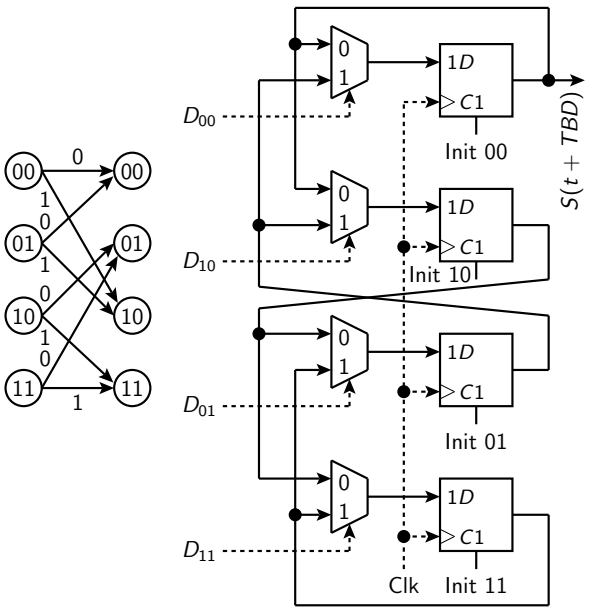


Figure 9. Trace forward unit (adapted from [14]).

preceding PM calculation are used to follow the path with the smallest cumulative error from the current state back to the initial state of the decoder. It can be shown that under normal conditions all paths through the trellis merge to a single one after a certain number of steps which is called Trace-Back Depth (TBD). This path represents the most probable sequence of the encoder's state transitions. As a consequence, decoder needs to store the decision bits for at least the number of symbols equal to TBD. The exact value of the TBD depends on the channel conditions. For Additive White Gaussian Noise (AGWN) channels TBD values of $5 \times K$ are sufficient, but they may be higher than $20 \times K$ for channels with fading and multi-path propagation.

The most obvious implementation of the trace-back step is to map the trellis diagram to hardware directly in form of the network of storage elements connected by multiplexers. This approach is called register exchange and is depicted in the Figure 8 for the trellis diagram shown in the Figure 2. If the number of trace-back stages is large enough and the channel conditions do not overstress the error correction performance of the code, all S_x outputs of the last stage should be the same after at most TBD clock cycles, so any of these outputs can be used as the decoded message.

The major disadvantage of the register exchange approach is its power inefficiency for higher values of the constraint length. In the extreme case of $K = 11$ on a multi-path channel, decoding of one single bit would require more than 200 000 accesses to the storage elements. Even for smaller K values and under the assumption that not every flipflop changes its value at every clock cycle, the power consumption remains quite high. For this reason, memory-based trace-back techniques are used in most real-world Viterbi decoders for $K > 3$.

Given the fact that one decision bit is produced by every ACS unit, the memory capacity for a single trace-back run equals to $\mathrm{TBD} \times 2^{K-1}$. As the decision bits cannot be overwritten during the trace-back, this amount needs to be at least doubled. Many Viterbi decoder implementations use three memory banks: one for storing the actual decision bits, one for merging the different paths to the single one determining the starting point for the decoding, and one for reading the decoded bits starting with the state from the merging step. As soon as one iteration is completed, i.e., one memory bank was completely read of written, the banks change their roles. This approach requires the memory capacity of $3 \times \mathrm{TBD} \times 2^{K-1}$. However, it has been shown that the merging step can be replaced by the single stage using 2^{K-1} registers of $K-1$ bits connected by a simple multiplexer network. This technique is called *trace forward* and allows to save one memory bank reducing the overall memory capacity to $2 \times \mathrm{TBD} \times 2^{K-1}$ [14]. Figure 9 shows the trace-forward hardware for $K = 3$ (note that every register is $K - 1 = 3 - 1 = 2$ bits wide). After TBD decoding steps the output $S(t+\mathrm{TBD})$ provides the bit pattern, which encodes the starting state for the actual trace-back run.

As already mentioned, the decision bits corresponding to the most probable path are traversed in the reverse order during the trace-back. Thus, the corresponding decoded input bits must be stored in a Last-In First-Out (LIFO) buffer. This LIFO memory is finally read to produce the decoded input sequence in the right order (which, in optimal case, will be the same as the original input to the encoder).

C. Adaptive Implementation

Table II summarizes some applications of convolutional codes in different communication standards. As already mentioned, even within a single standard code parameters may vary. While a Viterbi decoder for a specific convolutional code has fixed design parameters like the number of the BM and the ACS units, TBD, and so on, it is quite different in the case of the adaptive decoder. Concerning the most settings, the decoder must be dimensioned for the worst case, for instance, the largest value of the trace-back depth or the number of quantization levels for the soft-bit input.

As the decoder should be capable of processing both hard- and soft-input bits, the first design decision is the maximum input bit-width. As has been shown by previous research, increasing the soft-bit resolution beyond 5 bits has almost

TABLE II. CONVOLUTIONAL CODES IN DIGITAL COMMUNICATION SYSTEMS.

Standard	Constraint length K	Generator polynomials	Code rate R (mother code)
IEEE 802.11a/g/n (wireless LAN)	7	$g_1 = 133_8, g_2 = 171_8$	1/2
ECMA-368 (wireless USB)	7	$g_1 = 133_8, g_2 = 165_8, g_3 = 171_8$	1/3
DAB (Digital Audio Broadcasting)	7	$g_1 = g_4 = 133_8, g_2 = 171_8, g_3 = 145_8$	1/4
DVB (Digital Video Broadcasting), CCSDS 131.0-B (Space Data Systems, telemetry)	7	$g_1 = 171_8, g_2 = 133_8$	1/2
UMTS (Universal Mobile Telecommunications System)	9	$g_1 = 561_8, g_2 = 753_8$	1/2
		$g_1 = 557_8, g_2 = 663_8, g_3 = 771_8$	1/3
WiMAX (Worldwide interoperability for Microwave Access)	7	$g_1 = 171_8, g_2 = 133_8$	1/2
GSM (Global System for Mobile Communications)	5	$g_1 = 31_8, g_2 = 33_8$	1/2
		$g_1 = 33_8, g_2 = 25_8, g_3 = 37_8$	1/3
CDMA2000 (Code Division Multiple Access)	11	$g_1 = 4656_8, g_2 = 4726_8, g_3 = 5562_8, g_4 = 6372_8$	1/4
GMR-1 (GEO Mobile Radio interface), satellite phone	5	$g_1 = 31_8, g_2 = 27_8$	1/2
		$g_1 = 25_8, g_2 = 33_8, g_3 = 37_8$	1/3
		$g_1 = 31_8, g_2 = 27_8, g_3 = 25_8, g_4 = 37_8$	1/4
		$g_1 = 25_8, g_2 = 33_8, g_3 = 37_8, g_4 = 35_8, g_5 = 27_8$	1/5
	7	$g_1 = 133_8, g_2 = 171_8$	1/2
GMPRS (GEO Mobile Packet Radio Service), in addition to GMR-1	6	$g_1 = 45_8, g_2 = 55_8, g_3 = 73_8, g_4 = 77_8$	1/4
	9	$g_1 = 435_8, g_2 = 657_8$	1/2
		$g_1 = 755_8, g_2 = 633_8, g_3 = 443_8$	1/3
		$g_1 = 671_8, g_2 = 645_8, g_3 = 473_8, g_4 = 537_8$	1/4

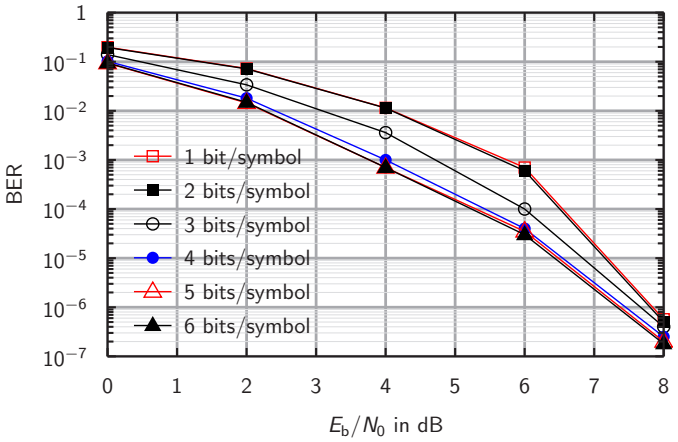
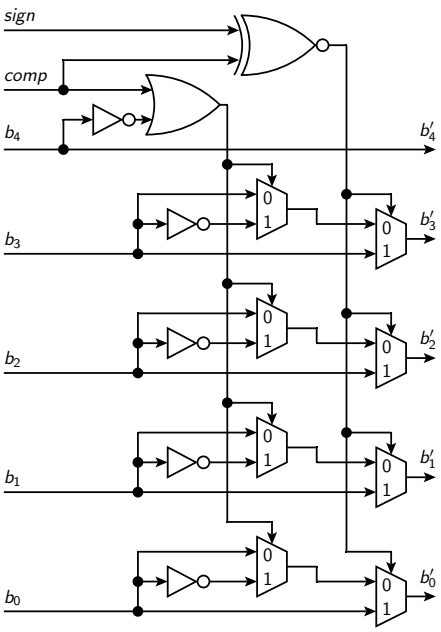


Figure 10. Effect of increasing the input bit width of a Viterbi decoder on BER (adapted from [11]).



(a) Circuit

b	$c_x^{(i)} \mapsto b_y$
1	$c_x^{(0)} \mapsto b_4$
2	$c_x^{(1)} \mapsto b_4$
	$c_x^{(0)} \mapsto b_0$
3	$c_x^{(2)} \mapsto b_4$
	$c_x^{(1)} \mapsto b_1$
	$c_x^{(0)} \mapsto b_0$
4	$c_x^{(3)} \mapsto b_4$
	$c_x^{(2)} \mapsto b_2$
	$c_x^{(1)} \mapsto b_1$
	$c_x^{(0)} \mapsto b_0$
5	$c_x^{(4)} \mapsto b_4$
	$c_x^{(3)} \mapsto b_3$
	$c_x^{(2)} \mapsto b_2$
	$c_x^{(1)} \mapsto b_1$
	$c_x^{(0)} \mapsto b_0$

(b) Input mapping

Figure 11. Unsigned magnitude input converter ($c_x^{(i)}$ ist the ith bit of the input symbol c_x)

no impact on the Bit Error Ratio (BER) [11]. Figure 10 summarizes the corresponding dependencies. According to these findings, the b parameter will be limited to the range from 1 (hard-input) to 5. To unify the BM calculation, the input is converted to the unsigned magnitude format. The corresponding circuit is shown in Figure 11. Depending on the value of the *sing* and *comp* inputs, the soft-bits are considered to be in the two's complement ($sign = 0$, $comp = 1$), signed magnitude ($sign = 1$, $comp = 0$) or unsigned magnitude ($sign = comp$) format (see Table I for explanation). The output always represents a corresponding unsigned magnitude value. Note that the size of the converter circuit does not depend on any other code parameter except the input bit-width, so for the fixed b value it remains constant and introduces almost no overhead to the overall silicon footprint. If the input bit-width is less than 5, then the corresponding bits are padded with zeros.

The number of BM values depends on the code rate R and is equal to 2^n. For instance, if the decoder should be able to decode any code up to $R = 1/5$, 32 branch metrics need to be calculated. For this purpose the circuit shown in Figure 5b can be scaled accordingly, while the constant values have to be adapted to the input bit-width b, e.g., 16 in case of $b = 4$ or 32 in case of $b = 5$. For other metrics than Manhattan distance, the data path has to be slightly extended. For instance, calculating squared Euclidean distance require an additional multiplier or

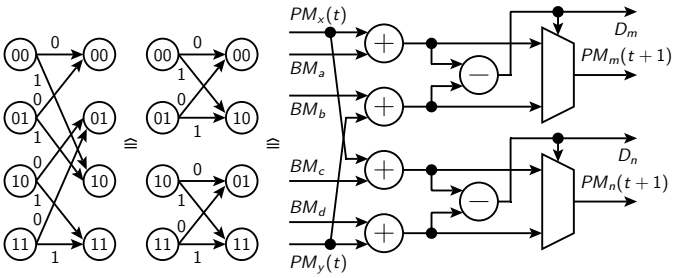


Figure 12. Butterfly structure of the path metric calculation. Comparators are represented by subtractors.

TABLE III. OUTPUT OF DIFFERENT CONVOLUTIONAL ENCODERS WITH $K = 3$.

State transition	Output c_1c_2 for $g_2 = 111$ and	
	$g_1 = 011$	$g_1 = 101$
$00 \rightarrow 00$	00	00
$00 \rightarrow 10$	01	11
$01 \rightarrow 00$	11	11
$01 \rightarrow 10$	10	00
$10 \rightarrow 01$	11	01
$10 \rightarrow 11$	10	10
$11 \rightarrow 01$	00	10
$11 \rightarrow 11$	01	01

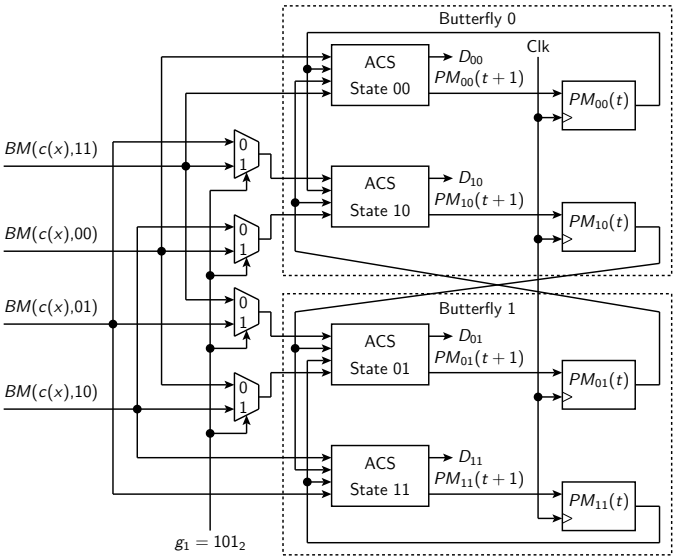


Figure 13. Path metric calculation for a reconfigurable Viterbi decoder.

a Look-Up-Table (LUT) per symbol. Again, once dimensioned for the lowest code rate (the largest value of n), the silicon area required to implement the adaptable branch metric unit does not change with any other code parameter.

The design of an adaptable ACS unit is determined be the number of codes with different generator polynomials as well as the maximal constraint length. For $K = 11$, the number of ACS calculations per decoded bit is equal to 1024. Mapping all of them to hardware so that the decoder will be able to achieve maximal degree of parallelism would require far too much silicon area. A reasonable approach is to implement 256 ACS units, so that all codes up to $K = 9$ can be decoded fully parallel, while for $K = 10$ and $K = 11$ two or four clock cycles are required to decode a single bit, respectively. Considering current CMOS technology nodes, even using this time-multiplexed mode for higher constraint lengths provides throughputs of several hundreds of megabits per second, which should be enough for most applications.

Changing the generator polynomials of the code does not affect the stage transitions of the encoder. In fact, due to the binary nature of the input, every state has two predecessors and two successor states while the edges connecting these states are following the same pattern, independent of the number of states. Figure 12 shows how these patterns can be used to combine pairs of states within the so-called *butterfly* structure. If generator polynomials are replaced by the new ones, the encoder, however, produces different output, since different bits (compared to the initial code) are passed to the XOR gates. Table III indicates how the output symbols produced by the same state transition change, if the generator polynomial $g_1 = 011$ of the encoder represented in Figure 1 is replaced by $g_1 = 101$ (while g_2 remains the same). Consequently, every new set of generator polynomials requires a different interconnect pattern between the branch metrics and the inputs of the butterfly units.

Figure 13 shows the ACS calculation unit for a decoder capable of decoding the output produced by the encoder in Figure 1, as well as the code generated by $g_1 = 101$ and $g_2 = 111$. While for each particular code the interconnect between the BM calculation outputs and PM calculation inputs is fixed, the adaptive decoder needs to switch between two different patterns depending on the code to be decoded. This functionality is realized by the multiplexers placed at the inputs of the particular ACS units. If the decoder has to process codes with different constraint lengths, some additional flexibility is required in the connections between the registers storing the actual PM values and the inputs of the ACS units as well. However, if the ACS units are grouped to the butterflies as shown above, the interconnect between them show a very regular pattern, so the overall area overhead is limited. Still, with larger number of different polynomials and larger values of the constraint length more and larger multiplexers at the inputs of the adaptive PM calculation unit are required reducing the clock frequency and throughput. A detailed design of the interconnect network for an adaptable Viterbi decoder including the particular interconnect patterns for different settings is described in [15].

In general, varying the constraint length and/or the generator polynomials has the strongest impact on the additional chip area and the critical path of the decoder. For the trace-back stage, the memory capacity has to be calculated for the worst case, i.e., the largest K and TBD values. In addition, the addressing scheme has to be adapted to every single case, which imposes the need for programmable address counters. Compared to other modifications this overhead is almost negligible. Figure 14 shows the trace-back unit of the reconfigurable Viterbi decoder

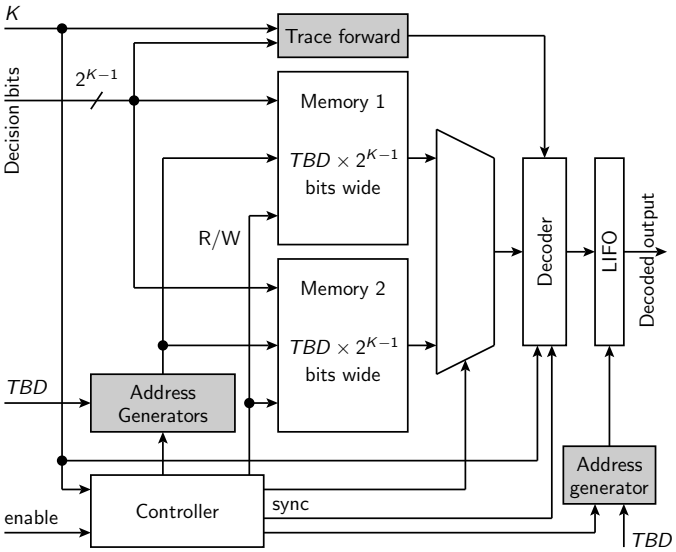


Figure 14. Trace-back unit for a reconfigurable Viterbi decoder with clock and reset signals omitted for the sake of clarity.

for variable values of the constraint length. The building blocks affected by the reconfigurable nature of the design are shaded. In principle, the address generators could be dimensioned for the worst case, just like the memories. However, larger TBD increases the latency of the decoder, which would mean an unnecessary overhead for smaller K values. At the same time, making the address generators pre-loadable and scalable adds almost nothing to the overall silicon footprint of the decoder.

The trace forward block discussed in the previous subsection needs to be dimensioned for the largest K value, while superfluous registers are bypassed when decoding codes with smaller constraint lengths. Again, some additional multiplexers are needed to implement this feature. Since trace forward unit is not on the critical path of the decoder, it does not introduce a major burden. One memory block is used to store the actual decision bits (write access) while the other is used to trace back the decision bits from the previous cycle (read access). The memories swap their roles after one cycle is completed, which explains the need for the multiplexer at their read ports.

Table IV summarizes the area and timing figures of different adaptable Viterbi decoder designs known from literature [16][17][18]. The major findings can be generalized as follows:

- The overhead increases with the degree of flexibility. The Viterbi decoders designed to support a (small) fixed amount of codes introduce the least – in some cases even negligible – overhead. In contrast, the full flexibility regarding the constraint length, the code rate as well as the generator polynomials requires up to 50 % more silicon area and reduces the throughput by about 26 %. Although looking diminishing at the first glance, it may still be quite a good price to pay considering the alternative of implementing a dedicated decoder for every single code.
- Varying the constraint length K has the highest impact on the area overhead of the adaptive Viterbi decoder implementation. Two factors can be identified as being responsible for this matter. On one hand, the number of ACS units increases exponentially with K, which means, the number and the size of the multiplexers required to implement different interconnect patterns increase accordingly. On the other hand, the TBD value grows linearly with K, so the memory depth of the trace-back grows accordingly. At the same time, the memory width grows exponentially with K. These consequences become crucial for higher values of K: the area overhead for $K = 9$ is about 9 %, while for $K = 11$ it is more than 50 %. For Viterbi decoders with smaller constraint lengths the area overhead is almost negligible.

TABLE IV. COMPARISON OF ADAPTABLE AND SINGLE-CODE VITERBI DECODERS.

Architecture	Metric	Reference	Result
[16] $K = 3 \dots 7$, g variable, b fixed	silicon area	$K = 7$, g fixed	$+2,9\,\%$
	throughput	$K = 7$, g fixed	$-1,5\,\%$
VITURBO [17] $K = 3 \dots 9$, g variable, b fixed	silicon area	$K = 9$, g variable	$+9\,\%$
	max. clock frequency	$K = 9$, g variable	$-30\,\%$
[18] $K \in \{7, 9\}$, g according to EDGE, CMDA2000 and WCDMA, b fixed	silicon area	$K = 9$, g fixed (CDMA2000)	overhead is negligible
	critical path	$K = 9$, g fixed (CDMA2000)	$+4\,\%$
[7] $K = 3 \dots 11$, g variable, $n = 2 \dots 4$, $b = 1 \dots 5$	silicon area	$K = 11$, $n = 4$, $b = 5$, g fixed	$+50,1\,\%$
	throughput	$K = 11$, $n = 4$, $b = 5$, g fixed	$-26,1\,\%$

TABLE V. ELEMENTS OF THE FINITE FIELD $\mathrm{GF}(2^3)$ GENERATED BY $\mathbf{f}(x) = 1 + x + x^3$ IN DIFFERENT REPRESENTATIONS.

Symbolic	Binary	Polynomial	α-polynomial
0	000	0	$0 = 0 + 0 \cdot \alpha + 0 \cdot \alpha^2$
α^0	100	1	$1 = 1 + 0 \cdot \alpha + 0 \cdot \alpha^2$
α^1	010	x	$\alpha = 0 + 1 \cdot \alpha + 0 \cdot \alpha^2$
α^2	001	x^2	$\alpha^2 = 0 + 0 \cdot \alpha + 1 \cdot \alpha^2$
α^3	110	$1 + x$	$1 + \alpha = 1 + 1 \cdot \alpha + 0 \cdot \alpha^2$
α^4	011	$x + x^2$	$\alpha + \alpha^2 = 0 + 1 \cdot \alpha + 1 \cdot \alpha^2$
α^5	111	$1 + x + x^2$	$1 + \alpha + \alpha^2 = 1 + 1 \cdot \alpha + 1 \cdot \alpha^2$
α^6	101	$1 + x^2$	$1 + \alpha^2 = 1 + 0 \cdot \alpha + 1 \cdot \alpha^2$

III. REED-SOLOMON DECODERS

The basic idea of the error-correcting linear cyclic codes, nowadays known as the Reed-Solomon codes, can be traced back to 1952 [19][20]. In their current form, the codes were introduced in the seminal paper by Irving S. Reed and Gustave Solomon, hence the name [21].

A. Code Structure

RS codes are defined over a finite field $\mathrm{GF}(q^m)$, which is an extension field of $\mathrm{GF}(q)$, where q is a prime number and m is a natural number different from zero. All practically applied codes are based upon $q = 2$, which implies that the field size is a power of two. Every finite field is generated by an irreducible polynomial $\mathbf{f}(x)$ of the degree m, whereby the elements of the field can be themselves represented as

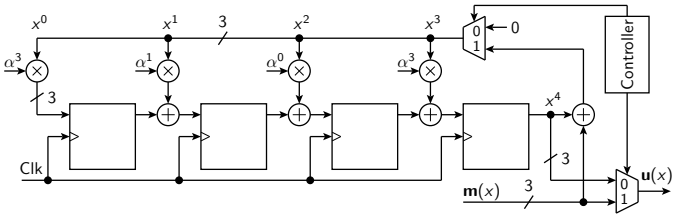


Figure 15. Reed-Solomon encoder for a (7,3) code over $\mathrm{GF}(2^3)$.

polynomials of the degree $m-1$ with the coefficients from $\mathrm{GF}(q)$ as shown in Table V. Choosing a different polynomial as the generator of the field results in different representation of its elements, however, it can be shown, that all finite fields for the fixed q und m values are isomorphic to each other [20]. If the generator polynomial of the field is primitive, its roots (called primitive elements) can be used as the generators of the multiplicative group of the field, i.e., every non-zero element can be derived from any root α by raising it to consecutive powers $\alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{q^m-2}$.

Binary representation of the elements of $\mathrm{GF}(2^m)$ requires m bits per element. This is the reason, why almost all RS codes used in communication and storage systems are defined over $\mathrm{GF}(2^8)$: each symbol can be encoded by exactly one byte. In contrast to the most other code families, the error correcting properties of the RS codes are defined symbol-wise, i.e., it does not matter, if a single bit or any other number of bits within one symbol are corrupted – the decoder treats it as a single symbol error. This explains, why RS codes are very powerful at correcting burst errors. A (255,239) RS code has the block size of 255 bytes, with 239 bytes of original message data and $2t = 255 - 239 = 16$ bytes of the checksum added during encoding. $t = 16/2 = 8$ is the number of corrupted symbols which can be corrected by the code. If 8 adjacent symbols (64 adjacent bits in the worst case) would be corrupted during the transmission, the decoder would still be able to correct all of them.

The general notation used to specify RS codes is (n, k), where n is the block size and k is the original message size before encoding (both specified in symbols of m bits width). As stated in the example above $n - k = 2t$, where t is the number of correctable symbol errors and $R = k/n$ is the code rate. The t parameter can be chosen depending on the required error correction capacity of the code. Once fixed, it defines the generator polynomial for the code, which has the degree $2t$:

$$\mathbf{g}(x) = g_0 + g_1 x + g_2 x^2 + \ldots + g_{2t-1} x^{2t-1} + x^{2t}. \quad (2)$$

The coefficients $g_0, \ldots, g_{2t-1}$ are all elements of $\mathrm{GF}(2^m)$ and the polynomial itself is defined in such a way, that its $2t$ roots correspond to the $2t$ consecutive powers of the single element of $\mathrm{GF}(2^m)$:

$$\mathbf{g}(x) = (x - \alpha) \cdot (x - \alpha^2) \cdot (x - \alpha^3) \cdot \ldots \cdot (x - \alpha^{2t}). \quad (3)$$

Although in principle any $2t$ consecutive powers can be chosen to define the generator polynomial, for practical reasons, the primitive element $\alpha = \alpha^1$ of the extension field $\mathrm{GF}(q^m)$ is chosen as the first root in most cases (being the generator of the multiplicative group of this field). Equation (3) follows this convention as well.

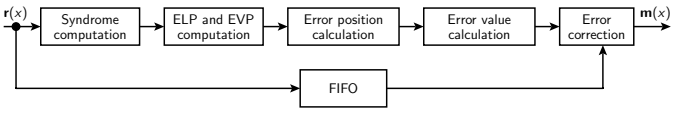


Figure 16. Top view architecture of the Reed-Solomon decoder.

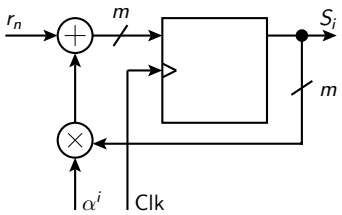


Figure 17. Basic cell for the syndrome computation.

RS codes are block-codes, i.e., they extend messages of the fixed size k by adding $n-k$ redundant symbols, so the encoded information block consists of n symbols. It is possible to vary the block size without changing the error correction capability of the code by replacing a part of the original message by z zeros before encoding. This z zero symbols are not transmitted. Instead, the receiver inserts the corresponding number of zeros before decoding, so effectively, the message size is reduced to $k - z$, while the difference $(n - z) - (k - z) = n - k = 2t$ determining the number of errors correctable by the code remains the same. This procedure is called shortening and the resulting code shortened, while the original code is referred to as mother code, just like in case of puncturing discussed in Subsection II-A.

The encoding procedure appends the rest $\mathbf{p}(x)$ of the division of the polynomial representing the original message $\mathbf{m}(x)$ shifted to the left by $n-k$ symbols by the generator polynomial:

$$x^{n-k}\mathbf{m}(x) = \mathbf{q}(x)\mathbf{g}(x) + \mathbf{p}(x), \quad (4)$$

which is equivalent to

$$\mathbf{p}(x) = x^{n-k}\mathbf{m}(x) \bmod \mathbf{g}(x). \quad (5)$$

This operation can be easily carried out using feedback shift-register as illustrated in Figure 15 for a very simple (7,3) RS code. Note that addition and multiplication operations are defined within the corresponding finite field (i.e., they differ from standard arithmetic). Since the original message is not changed by the encoding procedure, RS code is classified as systematic in coding theory (non-systematic RS codes exist as well but are not as widely used as their systematic counterparts).

B. Decoding Algorithm

Figure 16 shows the global architecture of the RS decoder ($\mathbf{r}(x)$ and $\mathbf{m}(x)$ stand for received and decoded message respectively). In a nutshell, the decoding process consists of five consecutive steps called syndrome computation, Error Locator Polynomial (ELP) and Error Value Polynomial (EVP)

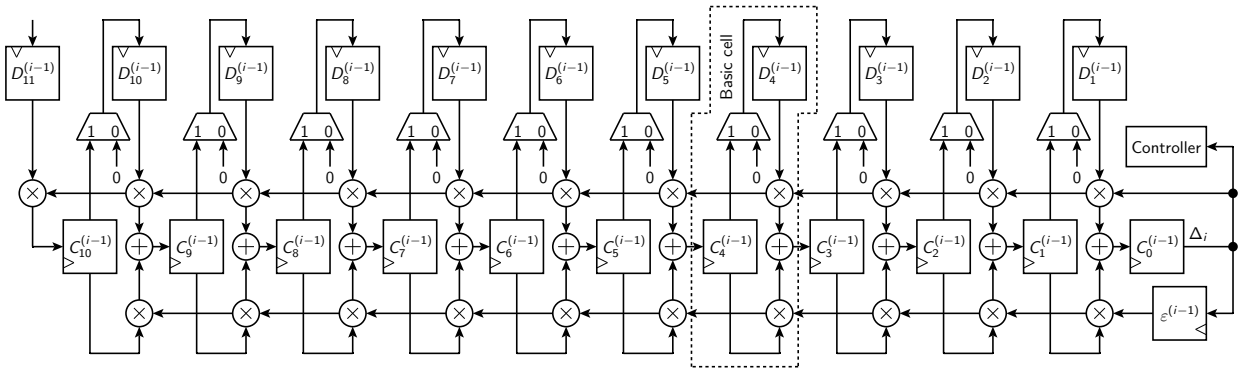


Figure 18. Dual-line implementation of the Berlekamp-Massey algorithm for $t = 5$ in hardware with clock and reset signals as well as multiplexer selector inputs omitted for the sake of clarity (adapted from [22]).

computation, error position calculation, error value calculation and error correction. Compared to the rest of the decoding algorithm, the first and the last step are rather trivial. Since decoding requires a number of clock cycles to accomplish, the First-In First-Out (FIFO) memory block is used to shift the received message accordingly, so the symbols arrive at the error correction stage at the same time as the corresponding correction values are provided.

Given that the encoding process can be represented by polynomial multiplication over $\mathrm{GF}(q^m)$, it becomes clear how transmission errors can be detected (and corrected). Since the multiplication operation preserves the roots of the generator polynomial, the decoder can verify the received message by checking whether $2t$ consecutive powers of the primitive element of $\mathrm{GF}(q^m)$ are roots of the polynomial $\mathbf{r}(x)$. This is the first step of the decoding process called syndrome computation. If at least one of the syndromes is not equal to zero, then the message was corrupted. Depending of the latency requirements, the syndromes can be calculated sequentially or in parallel by inserting the consecutive powers of α (elements of the field $\mathrm{GF}(2^m)$) into $\mathbf{r}(x)$. Figure 17 (adapted from [23]) shows the hardware used for the syndrome computation with r_n representing the nth symbol of the received message and S_i standing for the ith syndrome value.

After the syndromes are known, they are used in the next step of the decoding process to compute the number and the positions of the corrupted symbols. The most intuitive way to accomplish this is – in the first iteration – to assume the number of errors to be one and try to solve the corresponding system of equations $\mathbf{\Lambda}_l \cdot \mathbf{S}^\top = 0$ based on the vector of syndrome values $\mathbf{S}$ and the error locator polynomial

$$\begin{aligned}\mathbf{\Lambda}_l(x) &= x^l \cdot \prod_{i \in I_{\mathrm{FP}}} \left(x - \alpha^i\right) \\ &= x^l \cdot \left(\lambda_0 + \lambda_1 x + \ldots + \lambda_{r-1} x^{r-1} + x^r\right)\end{aligned} \tag{6}$$

with l as the (assumed) number of corrupted symbols, and $I_{\mathrm{FP}} = \{i : e_i \neq 0, 0 \leq i < n\}$ as the set of positions of all error values e_i. For $i \in I_{\mathrm{FP}}$, $x = \alpha^i$ is a root of ELP meaning that $e_i \neq 0$ (ith symbol of the message is corrupted).

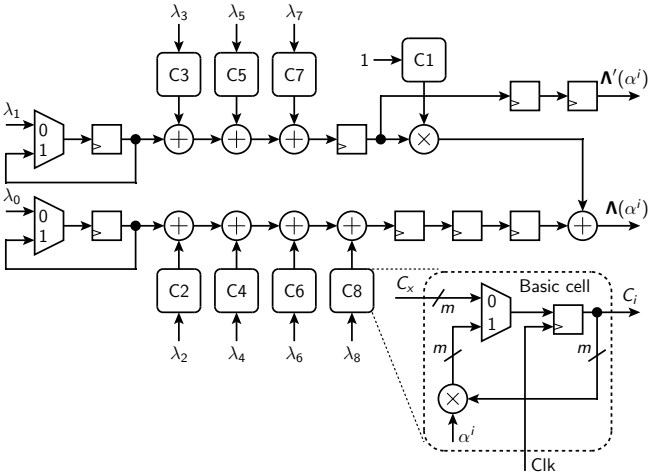


Figure 19. Hardware implementation of the Chien search for a (255,239) RS decoder. Clock and reset signals are omitted for the sake of clarity (adapted from [24]).

Accordingly if $i \notin I_{\mathrm{FP}}$, then $x = \alpha^i$ is not a root of ELP meaning that $e_i = 0$ (ith symbol of the message is correct).

The solution for $l = 1$ indicates the position of the corrupted symbol. In case the system of equations for a single error is not solvable, the assumption of two corrupted symbols ($l = 2$) is made resulting in a different system of equations. The process is repeated until a solvable system of equations is found and solved delivering the ELP coefficients $\lambda_0 \ldots \lambda_{l-1}$. This iterative procedure of finding the number of the corrupted symbols is called Peterson-Zierler-Gorenstein algorithm [25][26].

When ELP coefficients are known, finding the positions of the erroneous symbols can be easily accomplished by Chien search [27]. It checks which symbols of $\mathrm{GF}(2^m)$ (except zero) are roots of the $\mathbf{\Lambda}_l(x)$. As already mentioned, ELP is constructed in such a way, that the equation $\mathbf{\Lambda}_l(\alpha^i) = 0$ is valid, if ith symbol of the received message is corrupted. Consequently, the power i of every root found by the Chien search indicates the position of one corrupted symbol.

Since RS codes work on symbols of m bits width, knowing the error position of the symbol is not enough. An additional

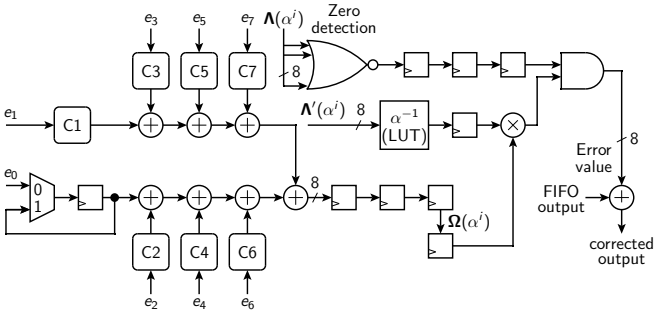


Figure 20. Hardware implementation of the Forney algorithm for a (255,239) RS decoder. Basic cells are the same as ones used for the Chien search. Clock and reset signals are omitted for the sake of clarity (adapted from [24]).

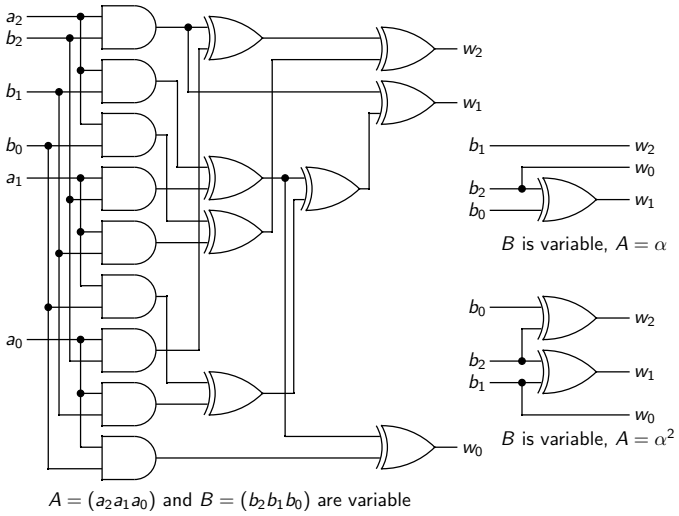


Figure 21. Variable-variable versus constant-variable multiplier for $\mathrm{GF}(2^3)$.

step is required to determine the error values. This can be done by using the syndromes computed in the first step, since every set of syndromes can be mapped to certain errors. Alternatively, the error value polynomial

$$\mathbf{\Omega}(x) = 1 + e_1 x + e_2 x^2 + \ldots + e_{e-1} x^{e-1} \qquad (7)$$

with e_x representing the error values can be used [28]. EVP stands in relation to the ELP and the syndrome vector via the so-called *key equation*

$$\mathbf{\Lambda}(x)\mathbf{S}(x) \bmod x^{2t} = \mathbf{\Omega}(x). \qquad (8)$$

Since Peterson-Zierler-Gorenstein algorithm does not scale very well, Berlekamp-Massey algorithm is more frequently used for the hardware implementation of the RS decoders [29]. It can be parallelized and modified to reuse the same hardware to calculate both error location and error value polynomials by solving the key equation. Figure 18 shows the hardware implementation of the Berlekamp-Massey algorithm using the so-called dual-line architecture [22]. Its detailed description, however, would go far beyond the scope of this paper. Again, after the coefficients of the ELP and EVP are calculated, Chien search must be carried out to determine the error locations. Figure 19 shows the corresponding circuit, where the notation Cx within the basic cell means, that α^x value is used for the calculation. In parallel to the error locations the so called *formal derivative* $\mathbf{\Lambda}'$ of the ELP is calculated:

$$\mathbf{\Lambda}'(x) = \sum_{i=1}^{t} i \cdot \lambda_i x^{i-1} \;\text{ with }\; i \cdot \lambda_i = \underbrace{\lambda_i + \lambda_i + \cdots + \lambda_i}_{i \text{ times}}. \qquad (9)$$

Assuming that the sequence of the $2t$ consequtive powers of the GF elements used as the roots of the generator polynomial $\mathbf{g}(x)$ starts with $\alpha^1 = \alpha$, the error values can be calculated with Lagrange interpolation using the formal derivative and the (already caluculated) EVP [30][31]:

$$e_i = -\frac{\mathbf{\Omega}(\alpha^{-i})}{\mathbf{\Lambda}'(\alpha^{-i})} \;\text{ with }\; \alpha^{-i} = \frac{1}{\alpha^i} \;\equiv\; \alpha^i \cdot \alpha^{-i} = 1, \qquad (10)$$

whereby the same basic cell as used for the Chien search can be applied and the multiplicative inverse values α^{-i} can be pre-calculated and stored in a LUT. This method of determining the error values are known as Forney algorithm [28]. As can be seen from Figure 20, which shows the corresponding hardware, the data path resembles some of the structure used for the Chien search. The inputs of the basic cells are now the coefficients of the EVP (instead of ELP) and both ELP $\mathbf{\Lambda}(x)$ and its formal derivative $\mathbf{\Lambda}'(x)$ are used as inputs.

In the final decoding step, the error values are added to the received symbols at the calculated error positions restoring the original message. As already mentioned, a FIFO memory is used to make sure the right values appear at the output of the decoder at the right time. The finite filed addition corresponds to a bitwise XOR operation, which scales well and is trivial from the implementation point of view.

C. Adaptive Implementation

Table VI summarizes some applications of RS codes in different communication standards. Depending on the required error correction performance, various finite fields, primitive polynomials and t values are used in practice.

As can be seen from the previous subsection, decoding the RS codes requires numerous calculations based on the finite field arithmetic. While the finite field addition is trivial and scales seamlessly with the bit-width m, efficient implementation of the finite field multiplication has some issues. In an RS decoder designed for a specific code, many of the GF-multipliers have one constant input (variable-constant multipliers). For adaptive decoding of different codes, a variable-variable multiplier is required. The corresponding overhead is quite high as Figure 21 illustrates for a very simple case of $\mathrm{GF}(2^3)$. This becomes even more critical with larger values of m. Luckily, only the multipliers involved in the syndrome and error location computation are affected, so the impact on the area of the whole RS decoder is limited, though not negligible.

The traditional way to implement a variable-variable GF-multiplier is based on the polynomial modulo reduction. If the

TABLE VI. REED-SOLOMON CODES IN DIGITAL COMMUNICATION SYSTEMS.

Standard	Galois-Field	Primitive polynomial $\mathbf{f}(x)$	(Mother) code
ITU-T J.83 (digital cable TV, USA)	$\mathrm{GF}(2^7)$	x^7+x^3+1	(128,122)
	$\mathrm{GF}(2^8)$	$x^8+x^4+x^3+x^2+1$	(204,188), (207,187)
ECMA-368 (wireless USB)	$\mathrm{GF}(2^8)$	$x^8+x^4+x^3+x^2+1$	(23,17), shortened from (255,249)
DAB (Digital Audio Broadcasting)	$\mathrm{GF}(2^8)$	$x^8+x^4+x^3+x^2+1$	(204,188) shortened from (255,239)
DVB (Digital Video Broadcasting)	$\mathrm{GF}(2^8)$	$x^8+x^4+x^3+x^2+1$	(204,188) shortened from (255,239)
DOCSIS 3.0 and 3.1 (Data Over Cable Service Interface Specifications)	$\mathrm{GF}(2^8)$	$x^8+x^4+x^3+x^2+1$	$(255,k)$ with $k=223,225,227,\ldots,251,253$, also shortened
WiMAX (Worldwide interoperability for Microwave access)	$\mathrm{GF}(2^8)$	$x^8+x^4+x^3+x^2+1$	(255,239)
VDSL (Very high speed Digital Subscriber Line)	$\mathrm{GF}(2^8)$	$x^8+x^4+x^3+x^2+1$	(144,128), (240,224), and others
GSM (Global System for Mobile communications)	$\mathrm{GF}(2^8)$	$x^8+x^4+x^3+x^2+1$	(85,73) shortened from (255,243)
IEEE802.3bp (automotive Ethernet)	$\mathrm{GF}(2^9)$	x^9+x^4+1	(450,406)
IEEE802.3bj (100Gb/s Ethernet)	$\mathrm{GF}(2^{10})$	$x^{10}+x^3+1$	(528,514), (544,514)
IEEE802.11ad (directional multi-gigabit wireless LAN)	$\mathrm{GF}(2^8)$	$x^8+x^4+x^3+x^2+1$	(224,208) shortened from (255,239)
GMR-1 (GEO Mobile Radio interface), satellite phone GMPRS (GEO Mobile Packet Radio Service)	$\mathrm{GF}(2^4)$	x^4+x+1	(15,9)
CCSDS 131.0-B (Space Data Systems, telemetry)	$\mathrm{GF}(2^8)$	$x^8+x^7+x^2+x+1$	(255,223), (255,239)

elements of the finite field are written in the polynomial form, their product can be calculated as follows [24]:

$$W = A \cdot B = \sum_{k=m}^{2m-2} d_k h^k + \sum_{k=0}^{m-1} d_k h^k \;\text{ with }\; d_k = \sum_{i=0}^{k} a_i b_{k-i}, \tag{11}$$

where a_x and b_x are the corresponding coefficients of the polynomial representation of A and B. Since a_x and b_x are both elements of $\mathrm{GF}(2)$, $d_k \in \mathrm{GF}(2)$ is valid as well. Given $h^k \in \mathrm{GF}(2^m)$ and $m \leq l \leq 2m-2$, h^k can be rewritten as

$$h^k = \sum_{i=0}^{m-1} g_i^k h^i, \tag{12}$$

while g_i^k with $0 \leq i \leq m-1$ is uniquely defined. Using the substitution $w_k = d_k + \sum_{j=m}^{2m-2} d_j g_k^j$, the product can be rewritten as

$$W = \sum_{k=0}^{m-1} w_k h^k, \;\text{ i.e., }\; W(x) = A(x) \cdot B(x) \bmod \mathbf{f}(x) \tag{13}$$

with $\mathbf{f}(x)$ as the generator polynomial of the finite field (not of the RS code) [24]. Thus, finite field multiplication can be divided into two steps: polynomial multiplication according to the equation (11) and modulo reduction according to the equation (13).

For $\mathrm{GF}(2^3)$, the polynomial representation of the elements looks like $A = a_2\alpha^2 + a_1\alpha + a_0$ and $B = b_2\alpha^2 + b_1\alpha + b_0$, so the first step delivers

$$\begin{aligned} A \cdot B &= (\alpha A b_2 + A b_1)\alpha + A b_0 = (\alpha(a_2\alpha^2 + a_1\alpha + a_0)b_2 \\ &+ (a_2\alpha^2 + a_1\alpha + a_0)b_1)\alpha + (a_2\alpha^2 + a_1\alpha + a_0)b_0 \\ &= a_2 b_2 \alpha^4 + (a_1 b_2 + a_2 b_1)\alpha^3 \\ &+ (a_0 b_2 + a_1 b_1 + a_2 b_0)\alpha^2 + (a_0 b_1 + a_1 b_0)\alpha + a_0 b_0. \end{aligned} \tag{14}$$

The higher powers $\alpha^4 = a^2 + \alpha$ and $\alpha^3 = \alpha + 1$ are reduced modulo 3 in the second step

$$\begin{aligned} A \cdot B &= a_2 b_2 \alpha^4 + (a_1 b_2 + a_2 b_1)\alpha^3 \\ &+ (a_0 b_2 + a_1 b_1 + a_2 b_0)\alpha^2 + (a_0 b_1 + a_1 b_0)\alpha + a_0 b_0 \\ &= a_2 b_2(\alpha^2 + \alpha) + (a_1 b_2 + a_2 b_1)(\alpha + 1) \\ &+ (a_0 b_2 + a_1 b_1 + a_2 b_0)\alpha^2 + (a_0 b_1 + a_1 b_0)\alpha + a_0 b_0 \\ &= (a_2 b_2 + a_0 b_2 + a_1 b_1 + a_2 b_0)\alpha^2 \\ &+ (a_2 b_2 + a_1 b_2 + a_2 b_1 + a_0 b_1 + a_1 b_0)\alpha \\ &+ a_1 b_2 + a_2 b_1 + a_0 b_0. \end{aligned} \tag{15}$$

The expression (15) reflects exactly the structure of the variable-variable multiplier depicted in the Figure 21, where nine AND gates represent the multiplication and eight XOR gates the addition over $\mathrm{GF}(2)$. Since the polynomial multiplication and modulo reduction have the same logic depth, a pipeline stage can be added between them for larger values of m to improve the clock frequency.

Although providing reasonable performance for a fixed m value, the polynomial modulo reduction approach cannot be applied for an adaptive architecture. Some solutions exists for specific cases, e.g., if $\mathrm{GF}(2^l)$ is a sub-field of $\mathrm{GF}(2^k)$ while l is an integer divider of k, a multiplication in $\mathrm{GF}(2^k)$ can be implemented by several multiplications in $\mathrm{GF}(2^l)$ [9][32].

A better approach is based on the Montgomery reduction [33], which uses two identical multiplier circuits to implement a multiplication over $\mathrm{GF}(2^m)$, while the same hardware can be reused for the multiplication over $\mathrm{GF}(2^l)$ for $l \leq m$. Figure 22 (adapted from [34]) shows the top-view architecture of the Montgomery multiplier with an example implementation for $\mathrm{GF}(2^4)$. Note that m_i are the (binary) coefficients of the generator polynomial $\mathbf{f}(x)$ of the finite field and $K(x)$ is a constant for every fixed $\mathbf{f}(x)$. The correspondence $m_0 = 1$ can be used to simplify the sub-circuits which use m_0 as an input.

Having a suitable GF multiplier in place, the implementation of an adaptable RS decoder is quite straightforward. The hardware components introduced in the Subsection III-B must be dimensioned for the largest envisioned values of t and m.

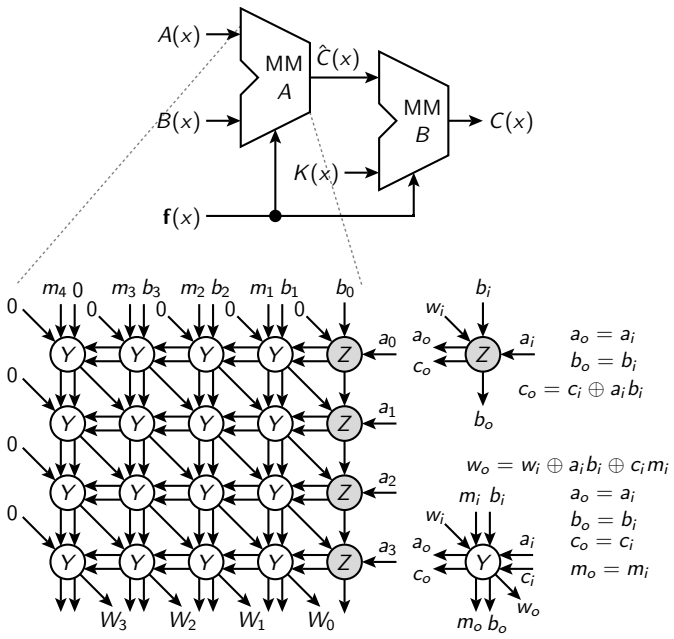


Figure 22. Montgomery multiplier for $\mathrm{GF}(2^m)$ with $m \leq 4$.

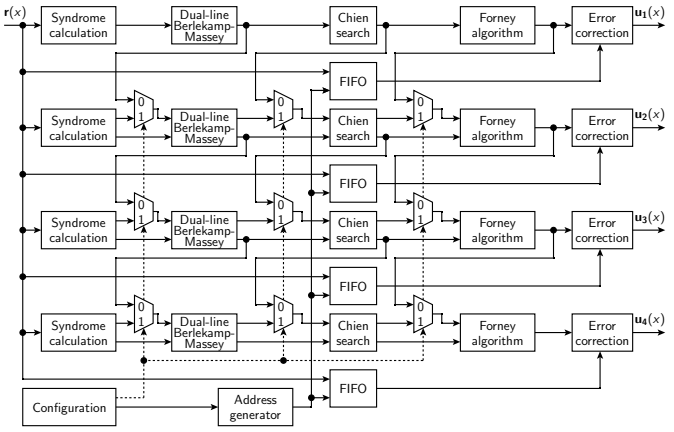


Figure 23. Top-level architecture of a multi-standard multi-stream Reed-Solomon decoder.

For smaller values of m, corresponding bits in the symbolic representation of the finite field elements must be filled with zeros. The same applies to the syndrome computation when decoding codes with smaller t parameters, i.e., the "nonexistent" roots are set to zero in advance. Alternatively, the inactive hardware can be used to decode one or more additional data streams in parallel. This can be done as long as the condition

$$\sum_{i=1}^{w} t_i \leq t_{\max} \tag{16}$$

holds (t_i represents the error correction capacity of the code for the data stream i). The corresponding architecture is shown in Figure 23 and can, for instance, be used to decode a single stream with $t = 16$ or up to four data streams encoded using a code with $t = 4$. The configuration block stores the control information to direct the data stream(s). In addition, the Montgomery multipliers must be configured depending on the chosen generator polynomials.

TABLE VII. COMPARISON OF ADAPTABLE AND SINGLE-CODE REED-SOLOMON DECODERS.

Comparison metric	Decoder architecture				
	[37]	[38]	[39]	[40] 1st variant	[40] 2nd variant
Code parameter	fixed	n and t variable	Universal decoder: n, t, m and $\mathbf{f}(x)$ variable		
m	8	8	$1 \ldots 8$	$1 \ldots 10$	$1 \ldots 8$
t	8	$1 \ldots 8$	$1 \ldots 8$	$1 \ldots 8$	$1 \ldots 16$
Erasure correction	no	no	no	≤ 16	≤ 16
Throughput in Gb/s	1,6	0,8	0,048	2,2	2,4
Gate equivalents	21 000	34 000	44 000	75 000 + 35 Kbit RAM	39 000 + 15 Kbit RAM

The last element affected by the adaptable implementation is the FIFO memory. Since its capacity does not depend on t, it has to be dimensioned for the largest envisioned block size and use programmable address generators to decode smaller blocks. The resulting overhead is almost negligible given the fact, that – depending on the implementation – the FIFO buffer can consume up to 75 % – 90 % of the overall chip area [35][36].

Table VII summarizes the area and throughput figures of different RS decoder designs known from literature [37][38][39][40]. For the adaptive decoding, n, t, and m parameters can be changed, though in practice m is usually kept constant, since almost all real-world RS codes are based on $m = 8$ (ITU-T J.83, IEEE802.3bp, and IEEE802.3bj standards being an exception by specifying codes based on $\mathrm{GF}(2^7)$, $\mathrm{GF}(2^9)$, and $\mathrm{GF}(2^{10})$ correspondingly). In addition, different polynomials can be chosen as the finite field generators. As mentioned in Subsection III-A, varying the GF generator polynomial $\mathbf{f}(x)$ results in a different encoding of the particular elements of the finite field, however, all finite fields for the fixed values of q and m are isomorphic to each other.

The conclusions from the study of the different RS decoder designs are as follows:

- The area overhead of the adaptive RS decoder implementation can be quite significant. Compared to the fixed-code decoder with $m = t = 8$, a fully reconfigurable design can consume about 85 % more silicon area in terms of gate equivalents in addition to 15 KBit more SRAM. Just as in the case of an adaptive Viterbi decoder this is still quite a low price to pay, since the error correction capabilities are much better and the additional erasure correcting feature is useful in many application scenarios (erasures are errors whose position is known in advance).
- Throughput is usually not an issue with RS decoding, since the symbol-per-second decoding rate is multiplied by m to obtain the bit-per-second values. For adaptive decoding, this means that the slower clock rates, caused by the additional overhead of the adaptive implementation, are not significant in most cases.

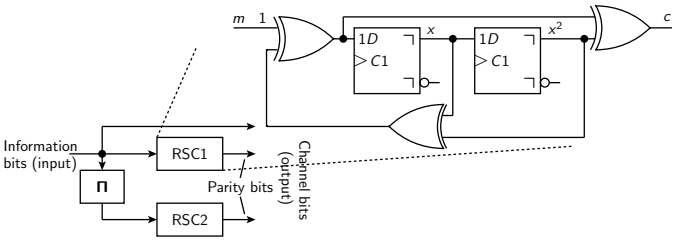


Figure 24. Simple turbo encoder.

TABLE VIII. TURBO CODES IN DIGITAL COMMUNICATION SYSTEMS.

Standard	Maximum constraint length K	Maximum block size (bits)	Code rate (mother code)
LTE (Long Term Evolution)	4	6 144	1/3
WiMAX (Worldwide inter-operability for Microwave access)	4	4 800	1/3
UMTS (universal mobile telecommunications system)	4	5 144	1/3
CDMA2000 (Code division multiple access)	4	20 730	1/5
EV-DO (Evolution data optimized)	4 4	4 096 1024	1/3 1/5
DVB-RCS (digital video broadcasting return channel satellite)	4	1 728	1/3
DVB-RCS2 (digital video broadcasting return channel satellite, second generation)	5	4 792	1/3
CCSDS 131.0-B (Space Data Systems, telemetry)	5	8 920	1/4

IV. MULTI-FAMILY DECODERS

As can be seen from the previous sections, the arithmetic involved in the decoding of convolutional and RS codes is quite different, which limits the possibilities of meaningful hardware reuse. However, it may be an option to combine decoders for the other code families within one design with Viterbi decoder or with each other.

A. Turbo Codes

Turbo codes were introduced in 1993 [41]. The basic idea is to use several (usually two) recursive convolutional encoders to process the same input data. First encoder receives the data directly, the second one gets an interleaved data stream as illustrated in Figure 24 (adapted from [42]). The size of the interleaver is usually in the order of several kilobits. Pseudo-random address generators are commonly used to implement the interleaving operation, which ensures that the same data appears as several statistically independent messages.

In contrast to the non-recursive non-systematic encoders discussed in the Subsection II-A, turbo encoders are usually built based upon Recursive Systematic Codes (RSC) due to their ability – especially if several encoders are concatenated – to produce codewords of higher weight, which has advantages concerning the recursive decoding procedure. The outputs of the encoders are transmitted as parity information together with the original message, so the resulting code is systematic. Due to the feedback within the data path, the RSC can be seen as an Infinite Impulse Response (IIR) filter [9, pp. 453–454]. Just like in case of conventional convolutional codes, puncturing can be applied to turbo codes to change the code rate.

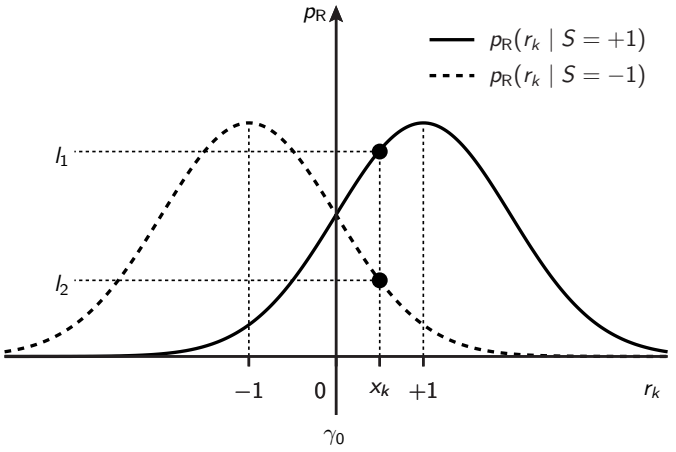


Figure 25. Likelihood functions for a BPSK symbols transmitted over AWGN channel [43].

Table VIII summarizes some applications of the turbo codes in different communication standards with their mother code rates (some standards specify additional code rates, which are not included in the table). The probably simplest explanation of the decoding principles assumes the Binary Phase Shift Keying (BPSK) signal s_k transmitted over an AGWN channel with noise specified as n_k [2][43, Section 8.1]. The received signal is the sum of the transmitted signal and noise

$$r_k = s_k + n_k \text{ with } p_{\mathrm{N}}(n_k) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{1}{2}\cdot\left(\frac{n_k}{\sigma}\right)^2} \quad (17)$$

as probability density with variance σ^2. The likelihood of the received signal to have a certain value under the assumption that s_k was sent can be calculated as follows:

$$\begin{aligned} p_{\mathrm{R}}(r_k \mid S = s_k) &= p_{\mathrm{N}}(r_k - s_k \mid S = s_k) \\ &= \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{1}{2}\cdot\left(\frac{r_k - s_k}{\sigma}\right)^2}. \end{aligned} \quad (18)$$

Since for a BPSK signal the only possible values for s_k are $+1$ and -1, these likelihood functions can be easily visualised as shown in Figure 25. A hard-decision maximum-likelihood decoder would compare the received value x_k against the likelihoods l_1 and l_2 and produce the value with the higher likelihood as the output. In other terms, such a decoder returns $+1$, if x_k lies on the right side of the decision threshold γ_0 and -1, if x_k lies to the left of γ_0 [43, Section 8.1.1.2].

Turbo decoder, however, works with soft decisions (soft output), i.e., instead of ± 1, the decoded value is the likelihood for a bit to be zero or one. A common way to represent this likelihood is to encode the bit decision as a sign bit (MSB) and the reliability of this decision as a magnitude. Log-Likelihood-Ratio (LLR) is an example of a metric constructed this way [44]:

$$LLR_{\mathrm{R}} = \ln \frac{p_{\mathrm{R}}(r_k \mid S = +1)}{p_{\mathrm{R}}(r_k \mid S = -1)}, \quad (19)$$

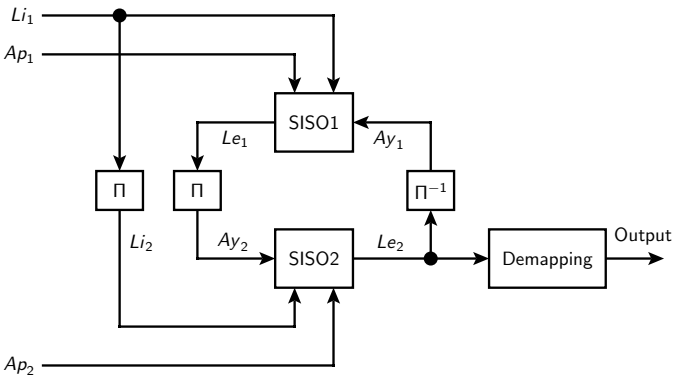


Figure 26. Top view architecture of the turbo decoder [2]. Li_x represent systematic bits (original message).

which is frequently used in soft-output decoding due to its numerical advantages (no floating-point multiplicaion required).

Figure 26 shows the global structure of the turbo decoder. The decoding is done iteratively using two decoders that process the received data and pass their outputs to each other, with interleaving and deinterleaving steps between them. Since each decoder uses the soft metrics produced by the other (called a-priori information Ay_x) as an own input and produces soft output decisions (called extrinsic bits Le_x) itself, the corresponding process is called Soft-Input Soft-Output (SISO) decoding.

In contrast to the hard-decision MLL decoding, each SISO module uses the a-priori information to improve its own decisions, which are then passed back to the first decoder and so on. During this process, the initial likelihood values associated with x_k are improved and shifted towards a more reliable decision after each iteration, so that the decoding process usually can be stopped after several iterations [43, Subsection 8.2.5]. This feedback loop gave the name to the code family in analogy to the turbocharger principle used in combustion engines. In the systems with fixed maximal latency requirements, the decoding process is stopped after several iterations. In addition, the decoding process can be stopped, if the results of subsequent iterations do not change (so-called *early stopping*), which can be easily implemented in LLR representation by comparing the MSBs of the corresponding soft-decisions. In the final step, a block called a demapper converts soft decisions into hard bits based on their probability metrics. Since both SISO modules are identical, common implementations of the turbo decoders often have only one SISO block in place and reuse it in a time-multiplexing manner as SISO1 and SISO2 (Figure 26) interchangeably to save the silicon area.

The error-correcting capabilities of the turbo codes depend on the constraint length and the size of the interleaver. Increasing both parameters improve the performance but has negative impact on the silicon footprint. As can be seen from Table VIII, the common range for K is between 3 and 5 and for block size between 1 and 21 kbit.

TABLE IX. LDPC CODES IN DIGITAL COMMUNICATION SYSTEMS.

Standard	Maximum block size (bits)	Code rate
IEEE 802.11n (wireless LAN)	1 944	1/2, 2/3, 3/4, 5/6
IEEE 802.11ad (WiGig, 60 GHz WLAN)	672	1/2, 3/4, 5/8, 13/16
IEEE 802.11.3c (WiGig, 60 GHz WLAN)	672	1/2, 3/4, 5/8, 7/8
IEEE 802.11ay (WiGig, 60 GHz WLAN)	1 344	1/2, 2/3, 3/4, 5/8, 5/6, 7/8, 13/16
WiMAX (Worldwide interoperability for Microwave Access)	2 304	1/2, 2/3, 3/4, 5/6
DVB-T2 and -S2 (Digital Video Broadcasting Terrestrial and Satellite)	64 800	1/2, 3/5, 2/3, 3/4, 4/5 5/6 (both), 1/4, 1/3, 2/5, 8/9, 9/10 (S2 only)
GMR (GEO mobile radio interface, satellite phone)	11 136	1/2, 2/3, 3/4, 4/5, 9/10
GMPRS (GEO Mobile Packet Radio Service)	8 880	1/2, 2/3, 3/4, 4/5, 9/10
IEEE 802.3an (10 Gigabit Ethernet)	2 048	7/8
IEEE 802.22 (WRAN, Wireless Regional Area Networks)	480	1/2, 2/3, 3/4, 5/6
CCSDS 131.0-B (Space Data Systems, telemetry)	8 160	1/2, 2/3, 4/5, 7/8
CCSDS 231.0-B (Space Data Systems, telecommand)	512	1/2

B. LDPC Codes

LDPC codes were described in detail in the dissertation of Robert Gallager in 1963 [45]. Due to the high computational requirements of the decoding process, the first practical applications emerged more than 30 years later, when LDPC codes were proposed as channel codes for several communication standards (see Table IX for examples). In a nutshell, LDPC codes are linear block codes with very large (up to several hundreds rows and columns), very sparsely filled generator matrix. As for other block codes, the encoding process can be described by the vector-matrix multiplication of the message block $\mathbf{m}$ by the generator matrix $\mathbf{G}$ producing the encoded message $\mathbf{c}$:

$$\mathbf{c} = \mathbf{m} \times \mathbf{G}. \tag{20}$$

The major difference to the "normal" block codes is the large block size (in the order of $10^4 \ldots 10^5$ bits) with the small number of parity bits (i.e., number of ones in the generator matrix), hence the name. For a systematic block code, $\mathbf{G}$ has the following form

$$\mathbf{G} = \left(\mathbf{P} \vdots \mathbf{I}_k \right) \tag{21}$$

with $\mathbf{P}$ as the parity matrix and $\mathbf{I}_k$ as the identity matrix of size k (the length of the message block $\mathbf{m}$). Since the parity matrix determines which message bits are involved in the corresponding parity equations, the correctness of the received information block of length n can be checked using the parity check matrix $\mathbf{H}^\top$ defined as

$$\mathbf{G} \times \mathbf{H}^\top = \mathbf{0} \text{ with } \mathbf{H}^\top = \begin{pmatrix} \mathbf{I}_{n-k} \\ \cdots \\ \mathbf{P} \end{pmatrix}. \tag{22}$$

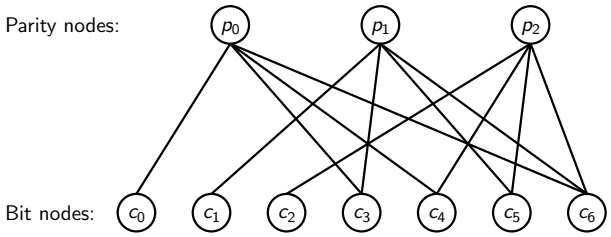


Figure 27. Tanner graph for a (7,4) Hamming code.

If the received codeword $\mathbf{c}'$ was affected by the error vector $\mathbf{e}$, the syndrome computation (multiplying $\mathbf{c}'$ by $\mathbf{H}^\top$) will deliver a non-zero value:

$$\begin{aligned}\mathbf{s} &= \mathbf{c}' \times \mathbf{H}^\top = (\mathbf{c}+\mathbf{e}) \times \mathbf{H}^\top = (\mathbf{m}\times\mathbf{G}+\mathbf{e}) \times \mathbf{H}^\top \\ &= \mathbf{m} \times \underbrace{\mathbf{G} \times \mathbf{H}^\top}_{=\mathbf{0} \text{ see } (22)} + \mathbf{e} \times \mathbf{H}^\top = \mathbf{e} \times \mathbf{H}^\top. \end{aligned} \quad (23)$$

According to the equation (23), the syndrome vector depends on the error vector, so it can be used to determine the bits affected by noise and restore the original message. The same principle applies to the non-systematic codes as well, though their generator and parity check matrix do not include the identity matrix. LDPC codes can be defined either as systematic or non-systematic with latter being used much more often in practice. The code rate of an LDPC code can easily be changed by adjusting the size of $\mathbf{G}$ as well as by applying puncturing.

In case of linear codes, if $\mathbf{e}$ is a valid codeword itself, its addition to the submitted message would result in a valid codeword, so the decoder would not notice the problem. However, due to the sparse filling of the generator matrix with ones, LDPC codes have a very large code distance making the latter case extremely unlikely to appear.

Due to the large block size, direct decoding of the received codeword by inspecting the syndrome vector is not practical. Instead, just like in case of turbo codes, the decoding process is carried out in an iterative manner. Bipartite graphs known as Tanner graphs are used to represent the parity check matrix as shown in Figure 27 for a simple Hamming code defined by the parity check matrix

$$\mathbf{H} = \left(\begin{array}{cccc:cccc} 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{array}\right). \quad (24)$$

For a $m \times n$ generator matrix, Tanner graphs consists of two disjoint set of nodes: $C = \{c_0, \ldots, c_{n-1}\}$ representing the bits of the encoded message and $P = \{p_0, \ldots, c_{m-1}\}$ representing the parity equations. A node $c_x \in C$ is connected to a node $p_y \in P$ if xth bit is involved in the yth parity check. An LDPC code is called regular, if the weight w_r of every row as well as the weight w_c of every column of its parity check matrix is the same, i.e., the sum of ones in every row is constant and the sum of ones in every column is constant (though w_r and w_c do not have to be equal). Such a code can be characterized by a triple (n, w_c, w_r).

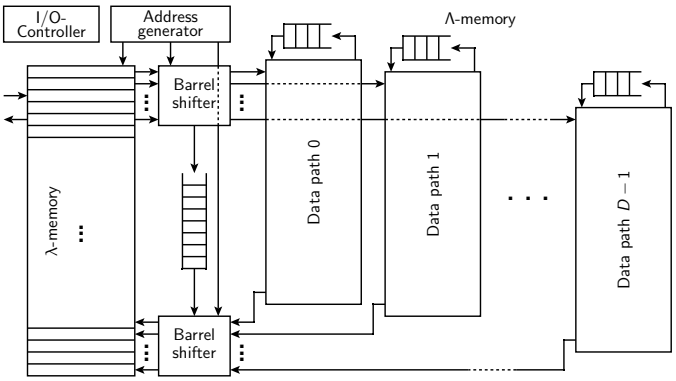


Figure 28. Top view architecture of the LDPC decoder [2].

The simplest way to decode the LDPC codes is to apply Gallager's hard decision algorithm. In the first iteration, parities are checked to determine, which bits of the information block are responsible for most unsatisfied parity equations. These bits are inverted and the next iteration starts. The decoding ends, as soon as all equations are satisfied or a certain number of iterations is reached. For a very simple regular (16,3,4) LDPC code with the parity check matrix

$$\mathbf{H} = \left(\begin{array}{cccc:cccc:cccc:cccc} 1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1 \\ \hdashline 1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0 \\ 0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0 \\ 0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0 \\ 0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1 \\ \hdashline 1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1 \\ 0&1&0&0&0&0&1&0&0&0&0&1&1&0&0&0 \\ 0&0&1&0&0&0&0&1&1&0&0&0&0&1&0&0 \\ 0&0&0&1&1&0&0&0&0&1&0&0&0&0&1&0 \end{array}\right) \quad (25)$$

this process looks as follows [20, Section 15.6].

Assume, the codeword

$$\mathbf{c}' = (1,1,0,0,1,0,1,0,0,0,0,0,0,0,0,0) \quad (26)$$

is received (with bits of $\mathbf{c}'$ numbered in ascending order from left to right and rows of $\mathbf{H}$ numbered in ascending order from above starting with 1). After calculating $\mathbf{c}' \times \mathbf{H}^\top = (0,0,0,0,0,1,1,0,1,0,0,1)$ it becomes clear, that the parity equations 6, 7, 9, and 12 are violated while the bits 6, 10, 11, and 15 are involved in the most (respectively two) violated equations. Flipping these bits results in

$$\mathbf{c}'' = (1,1,0,0,1,1,1,0,0,1,1,0,0,0,1,0). \quad (27)$$

After the product $\mathbf{c}'' \times \mathbf{H}^\top = (0,1,0,1,0,1,1,0,1,0,0,1)$ is calculated once again, the parity equations 2, 4, 6, 7, 9, and 12 appear violated, with bits 6 and 15 involved in the most

TABLE X. COMBINATION OPTIONS OF DIFFERENT DECODERS (BASED ON DATA FROM [2]).

Decoder	REED-SOLOMON	VITERBI	Turbo	LDPC
REED-SOLOMON	=	−	−	−
VITERBI	−	=	+	−
Turbo	−	+	=	+
LDPC	−	−	+	=

equations (three each). Finally, flipping these bits results in the codeword

$$\mathbf{c} = (1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0), \qquad (28)$$

which, according to the condition $\mathbf{c} \times \mathbf{H}^\top = \mathbf{0}$, was the original message [20, p. 600].

This process can be visualized using the Tanner graph of the code: Every parity node passes the modulo-2 sum (XOR) of the bits involved in the corresponding parity equation to every bit node within its network (the bit receiving the sum is not included in its calculation). The bit nodes with the largest amount of received ones are those violating the most parity equations and must be flipped [43, Subsection 8.2.4.2]. Better error correction performance is achieved by using the soft-decision approach, which – instead of flipping bits – propagates probabilities through the Tanner graph trying to maximize the a-posteriori probability for every bit to satisfy all parity equations [9, Section 15.5]. The bit probabilities based on the soft decision are assigned to the corresponding bit nodes and sent to the parity nodes, which use them to calculate the probabilities that the corresponding parity equations are satisfied. These probabilities are then sent back to the bit nodes according to their involvement in the respective parity equations and used to refine the initial assumptions. This procedure called *belief propagation* is repeated several times and can be seen as a kind of message passing algorithm [43, Subsection 8.2.4.4]. At the end, either the codeword is decoded or the decoding process stops after a fixed number of iterations without producing a solution. As in the case of turbo codes, logarithmic metrics like LLR can be used to represent the likelihood values, which are converted to the hard bit decisions at the end.

Figure 28 shows the global architecture of the LDPC decoder with soft-decision memories referred as λ, and parity node message memories referred as Λ. The number of data paths can be scaled according to the throughput requirements, while the barrel shifter take care of providing the correct values for the computation.

C. Combining Different Decoders

The conclusion from the discussion of the different decoding algorithms in the previous sections is that advantages of multi-family FEC decoder implementation result from memory and data-path reuse. Table X summarizes the potential of different combinations. As already mentioned, RS decoders have quite different arithmetic requirements, which explains, why the first row and the first column of the table are filled with "−" limiting the options to Viterbi, turbo, and LDPC decoders.

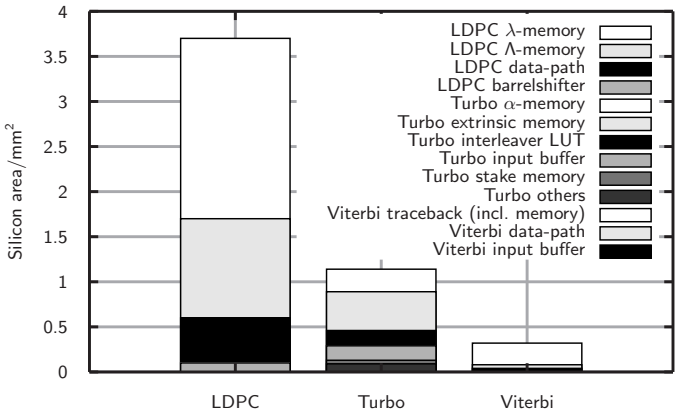


Figure 29. Chip area distribution for different FEC decoder designs (adapted from [2]).

According to the brief description of the iterative decoding, it should be clear that a lot of local memory is required to store the information bits, soft decisions, and intermediate results. Figure 29 shows the silicon area distribution between different parts of the design for LDPC, turbo, and Viterbi decoders. The extreme disproportion in the required silicon area between LDPC and Viterbi decoders suggests, that a combination of both within the same design with the goal to reuse some of the resources does not make sense. Even if the complete Viterbi decoder area could be reused for LDPC (which is virtually impossible), the overall gain would sum up to less than 10 % of the silicon footprint [2].

The SISO module required for turbo decoding usually implements the Bahl-Cocke-Jelinek-Raviv (BCJR) algorithm, which ensures the minimal (a-posteriori) symbol error probability [46]. BCJR is often used as a synonym for Maximum-a-Posteriori (MAP) approach. Depending on the probability metrics, modified versions of the same algorithm are known as Log-MAP and Max-Log-MAP (ML-MAP) [47]. An alternative SISO implementation is based on the Soft-Output Viterbi-Algorithm (SOVA), which can also be used to decode the convolutional codes. In contrast to the hard decision Viterbi decoder, SOVA uses a different path metric, which is based on the a-priory probabilities of the input symbols and produces a reliability metric instead of the hard bit decision as an output [9, Subsection 14.3.17]. The reliability could, for instance, be based on the difference between the branch metrics of the kept and discarded branches accumulated over the length of the TBD. SOVA minimizes the cumulative error for a sequence of symbols, which – unlike the BCJR approach – does not necessary result in a minimal symbol error probability.

The difference between two algorithms is that BCJR calculates both forward and backward iterations for every state transition (following the path to the actual and succeeding state from different directions), while Viterbi follows only one path [48]. The BCJR algorithm assigns three values to every state transition. For the transition from x to y (assuming x is reached at time t), the probability of the sequence up to time t ending at state x is first calculated by the forward iteration

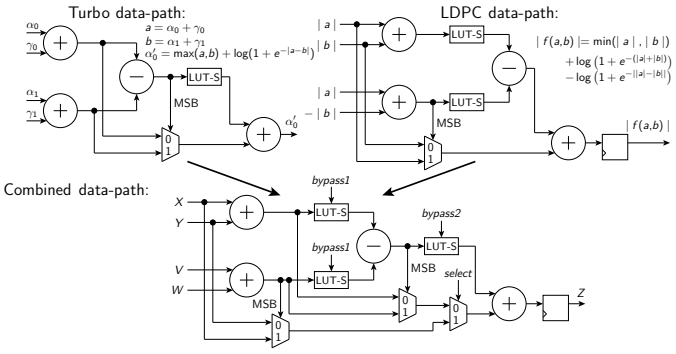


Figure 30. Combining Turbo and LDPC data-paths within a single FFU (adapted from [50], LUT-S are Look-Up-Tables storing the approximated logarithmic values).

of the BCJR algorithm and is commonly designated as $\alpha_t(x)$. Next, the probability of the future sequence from time $t+1$ to the end, given the state y at time $t+1$, is calculated by the backward iteration of the BCJR algorithm and is usually designated as $\beta_{t+1}(y)$. Finally, the branch metric of the state transition between x and y is calculated as $\gamma(x, y)$. Since the values of the previous iteration are required to compute the next one, they need to be stored. The corresponding memories are called α, β (or stake), and γ (or extrinsic), which explains the chart legend related to the turbo decoder in Figure 29.

Using SOVA instead of BCJR increases the potential for hardware reuse in a combined Viterbi-turbo decoder at the cost of slightly reduced bit error ratio for the turbo decoder compared to the SISO module based on the Max-log-MAP algorithm. In the case of an ML-MAP BCJR implementation, at least the adders of the Viterbi's ACS units can be reused in the LLR metric calculation. The same hardware can be reused for the forward and backward computation in a time-multiplexed manner as long as the throughput requirements are met [49].

A closer look at the internal data-path of the turbo and LDPC decoders reveals some similarities as well. Multi-family decoder can be built either reusing some of the LDPC data-path to decode turbo codes [51] or vice versa [52][53]. Figure 30 illustrates how the computational steps for both decoding algorithms can be unified within the same Flexible Functional Unit (FFU) [50]. As in the case of multi-standard Viterbi decoder, some additional multiplexers are required to switch between the codes, which slightly increases the critical path. However, significant reuse options are only valid for high-throughput turbo decoders, which require several parallel SISO units. Otherwise, the area percentage of the data-path compared to the memory is almost negligible, as shown in Figure 29. The memory reuse potential of about 10 % can be achieved if the access bandwidths of the turbo and LDPC parts are comparable [2].

The conclusion from the previous discussion leaves Viterbi-turbo and LDPC-turbo designs as the only options for a multi-family decoder implementation. Some case studies of such designs are known from literature [2][17][49][50][54]. Table XI summarizes the results. As expected, some portions of the data-path and memory can be reused resulting in silicon area savings of 10–42 %, while throughput is not affected in most cases of the combined Viterbi-turbo decoder. If adaptable designs are compared with a single family decoder, the overhead of introducing an additional code family is very low. For instance, adding the LDPC functionality to the existing turbo design comes at only 10–20 % additional silicon area [50].

TABLE XI. COMPARISON OF MULTI-FAMILY FEC DECODERS.

Architecture	Metric	Reference	Results
Viterbi-Turbo Decoders			
VITURBO [17]	silicon area	Viterbi decoder ($K = 3 \ldots 9$)	+5 % logic +25 % memory
[49] (fully parallel)	silicon area	separate turbo and Viterbi decoder	$-14,5\,\%$
	throughput	separate turbo and Viterbi decoder	no change
[49] (time-multiplex)	silicon area	separate turbo and Viterbi decoder	$-28,1\,\%$
	throughput	separate turbo and Viterbi decoder	same for Viterbi 1/2 for turbo
[54]	silicon area	turbo decoder	$+20\,\%$
LDPC-Turbo Decoders			
[2]	silicon area	separate turbo and LDPC decoder	$-10\,\%$
[50]	silicon area	turbo decoder LDPC decoder reused vs. separate data-paths	$+10 \ldots +20\,\%$ $+15 \ldots +20\,\%$ $-39 \ldots -42\,\%$

It should be noted that reusing the same hardware for different decoders limits the possibilities for parallel decoding of differently encoded data streams. Depending on the throughput requirements, either a time-multiplexed (temporal scheduling) or a multi-data-path (spatial scheduling) solution can be applied to address this problem. Since the application scenarios are known in advance, these constraints can be taken into account before the decision on the final decoder architecture is made.

V. COMPARATIVE ANALYSIS OF ADAPTIVE DECODER ARCHITECTURES

The findings from the last three sections provide enough data for a generalized analysis of all presented architectures. Although it is clear that adaptive decoder implementation comes at an additional cost, the figures vary depending on the code family, the code parameters involved, and the degree of flexibility.

Viterbi decoders can be enabled to decode a few additional codes with almost negligible overhead, as long as the constraint length is kept constant. Otherwise, the increase in the silicon footprint and the drop in throughput can be quite high. Generally speaking, adding several generator polynomials to an existing architecture or changing the code rate has much less impact on the area and performance figures than increasing the constraint length. From an implementation perspective, major changes in the overall decoder architecture are limited to making the data path configurable by adding multiplexers and inserting adaptable address generators instead of simple counters into the trace-back stage, while the memory capacity of the trace-back unit must be dimensioned for the worst case.

Throughput is more severely affected compared to the other discussed decoder families, especially if all code parameters are variable.

Changing the generator polynomials of the code and the underlying finite field for Reed-Solomon decoders has a much greater impact than in the Viterbi case, as it introduces major changes to the GF-multipliers used for syndrome and error location computation. However, changing the degree of the GF generator polynomial (which equals the symbol bit-width) is even more critical. In general, the modifications required to enable the basic RS decoder to process different codes are much more profound. For larger block sizes, the impact on the chip area of the overall decoder is limited, as the FIFO memory becomes the dominating factor. Reduced clock frequency is usually not an issue, since the decoding chain relies on symbols composed of multiple (usually 8) bits. For very small values of m, throughput plays a more important role. However, such RS codes are rarely used in real-world applications.

For multi-family decoders, selecting the proper code combinations is the key factor. Because the goal of such combined implementations is to reuse parts of the data-path and memory, adaptable Viterbi-turbo and LDPC-turbo architectures are the only meaningful alternatives. Changes to the data-path are comparable to the Viterbi decoder case, provided the same decoding algorithms and likelihood metrics are used (which may slightly degrade the BER figures). Otherwise, the potential for sharing silicon area across different decoders is limited to memory, which does not deliver a significant gain. Throughput is only slightly affected – or even completely unaffected – when multi-family decoders are compared to their single-family counterparts. However, the throughput requirements of the decoders should be closely matched to maximize the benefits of a combined implementation. In general, finding the optimal settings for a feasible multi-family decoder architecture is far more difficult than tuning a single-family decoder to process codes with variable parameters.

VI. Conclusion and Future Work

This paper discusses several options for the adaptive FEC decoder design. The findings based on literature study and own research suggest that the overhead of extending a decoder towards other codes of the same family comes at a moderate cost. As expected, this overhead (compared to a decoder implementation for a fixed parameter set) increases with the amount of flexibility and code complexity, but can be justified by the fact that separate designs per code instance are no longer necessary in most cases.

At the same time, decoders covering several code families can take advantage of resource sharing, reducing the overall silicon footprint compared to a dedicated implementation of one particular decoder per family. The potential advantages, however, strongly depend on the use cases and the envisioned set of standards. In particular, combining iterative decoders with significantly different throughput requirements will not result in noticeable area savings.

One aspect not discussed herein is the impact of the adaptive decoder implementation on power consumption. On one hand, reconfigurability comes at the cost of additional area, which can lead to higher energy-per-bit figures. On the other hand, reusing parts of the data-path and memory for different decoding algorithms is expected to yield overall energy savings. Supporting these assumptions with concrete metrics is a promising direction for future research.

References

[1] S. Sawitzki, "Adaptive Architectures for Forward Error Correction", in *CENICS 2025: The Eighteenth International Conference on Advances in Circuits, Electronics and Micro-Electronics*, T. Bostelmann, Ed., Barcelona, Spain: International Academy, Research, and Industry Association, Oct. 2025, pp. 1–7.

[2] J. T. M. H. Dielissen, N. Engin, S. Sawitzki, and C. H. van Berkel, "FEC Decoders for Future Wireless Devices: Scalability Issues and Multi-Standard Capabilities", in *Circuits and Systems for Future Generations of Wireless Communications*, A. Tasić, W. A. Serdijn, L. E. Larson, and G. Setti, Eds., ser. Series on Integrated Circuits and Systems, Berlin: Springer, 2009, pp. 271–297.

[3] T. Vogt and N. Wehn, "A Reconfigurable ASIP for Convolutional and Turbo Decoding in an SDR Environment", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 10, pp. 1309–1320, Oct. 2008.

[4] P. M. Velayuthan, "Towards Optimized Flexible Multi-ASIP Architectures for LDPC/Turbo Decoding", PhD thesis, Université de Bretagne-Sud, Dec. 2012.

[5] M. Scarpellino, A. Singh, E. Boutillon, and G. Masera, "Reconfigurable Architecture for LDPC and Turbo Decoding: A NoC Case Study", in *Proceedings of IEEE 10th International Symposium on Spread Spectrum Techniques and Applications*, Bologna, Italy: IEEE, Aug. 2008, pp. 671–676.

[6] J. T. M. H. Dielissen, N. Engin, S. Sawitzki, and C. H. van Berkel, "Multi-Standard FEC Decoders for Wireless Devices", *IEEE Transactions on Circuits and Systems II*, vol. 55, no. 3, pp. 284–288, May 2008.

[7] S. Sawitzki, "Embedded Reconfigurable Computing: Architekturen, Anwendungen und Entwurfswerkzeuge (Architectures, Applications, Tools)", Habilitation Thesis, Dresden University of Technology, 2024.

[8] P. Elias, "Coding for Noisy Channels", *IRE Convention Record*, vol. 3, no. 4, pp. 37–46, 1955.

[9] T. K. Moon, *Error Correction Coding. Mathematical Methods and Algorithms*. John Wiley & Sons, 2005.

[10] A. J. Viterbi, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm", *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, Apr. 1967.

[11] I. Habib, Ö. Paker, and S. Sawitzki, "Design Space Exploration of Hard-Decision Viterbi Decoding: Algorithm and VLSI Implementation", *IEEE Transactions on VLSI Systems*, vol. 18, no. 5, pp. 794–807, May 2010.

[12] A. P. Hekstra, "An Alternative to Metric Rescaling in Viterbi Decoders", *IEEE Transactions on Communications*, vol. 37, no. 11, pp. 1220–1222, Nov. 1989.

[13] J. Dragaš, "Design Trade-offs in the VLSI Implementation of High-Speed Viterbi Decoders and their Application to MLSE in ISI Cancellation", Master's Thesis, Integrated Systems Laboratory, Swiss Federal Institute of Technology Zurich, Mar. 2011.

[14] P. J. Black and T. H. Meng, "Hybrid Survivor Path Architectures for Viterbi Decoders", in *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, Minneapolis, MN, USA, Apr. 1993, pp. 433–436.

[15] W. Tang, N. Engin, F. A. Steenhof, M. Klaassen, A. Hekstra, and S. Sawitzki, *Multi-Standard Viterbi Processor*, US Patent 8,904,266 B2, NXP B. V., Eindhoven, The Netherlands, Dec. 2014.

[16] K. Chadha and J. R. Cavallaro, "A Reconfigurable Viterbi Decoder Architecture", in *Conference Record of the Thirty-Fifth Asilomar Conference on Signals, Systems & Computers*, M. B. Matthews, Ed., vol. 1, Pacific Grove, CA, USA: IEEE, Nov. 2001, pp. 66–71.

[17] J. R. Cavallaro and M. Vaya, "VITURBO: A Reconfigurable Architecture for Viterbi and Turbo Decoding", in *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, Hong Kong: IEEE, Apr. 2003, pp. 497–500.

[18] T. Vogt, N. Wehn, and P. Alves, "A Multi-Standard Channel-Decoder for Base-Station Applications", in *17th Symposium on Integrated Circuits and Systems Design (SBCCI 2004)*, Porto de Galinhas, Brazil: IEEE, Sep. 2004, pp. 192–197.

[19] K. A. Bush, "Orthogonal Arrays of Index Unity", *The Annals of Mathematical Statistics*, vol. 23, no. 3, pp. 426–434, Sep. 1952.

[20] W. C. Huffman and V. Pless, *Fundamentals of Error-Correcting Codes*. Cambridge, NY, USA: Cambridge University Press, 2003.

[21] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields", *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, Jun. 1960.

[22] H.-J. Kang and I.-C. Park, "A High-Speed and Low-Latency Reed-Solomon Decoder based on a Dual-Line Structure", in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Orlando, FL, USA, May 2002, pp. 3180–3183.

[23] A. Kumar, "High-Throughput Reed-Solomon Decoder for Ultra Wide Band", Master's Thesis, Technische Universiteit Eindhoven and National University of Singapore, Dec. 2004.

[24] H. Lee, "High-Speed VLSI Architecture for Parallel Reed-Solomon Decoder", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 2, pp. 288–294, Apr. 2003.

[25] W. Peterson, "Encoding and Error-Correction Procedures for the Bose-Chaudhuri Codes", *IRE Transactions on Information Theory*, vol. 6, no. 4, pp. 459–470, Sep. 1960.

[26] D. E. Gorenstein and N. Zierler, "A Class of Error Correcting Codes in p^m Symbols", *Journal of the Society of Industrial and Applied Mathematics*, vol. 9, no. 2, pp. 207–214, Jun. 1961.

[27] R. T. Chien, "Cyclic Decoding Procedures for the Bose-Chaudhuri-Hocquenghem Codes", *IEEE Transactions on Information Theory*, vol. 10, no. 4, pp. 357–363, Oct. 1964.

[28] G. D. Forney, "On Decoding BCH Codes", *IEEE Transactions on Information Theory*, vol. 11, no. 4, pp. 549–557, Oct. 1965.

[29] E. R. Berlekamp, *Algebraic Coding Theory*, revised. Aegean Park Press, Jun. 1984.

[30] R. D. Kudryavtsev and M. K. Samarin, "Lagrange Interpolation Formula", in *Encyclopedia of Mathematics*, M. Hazewinkel, Ed., London, UK: Springer, 2001.

[31] C. K. P. Clarke, "Reed-Solomon Error Correction", British Broadcasting Corporation, White paper WPH 031, Jul. 2002.

[32] A. Vourdas, "Harmonic Analysis on a Galois Field and Its Subfields", *Journal of Fourier Analysis and Applications*, vol. 14, no. 1, pp. 102–123, Feb. 2008.

[33] P. L. Montgomery, "Modular Multiplication without Trial Division", *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, Apr. 1985.

[34] C.-C. Lin, F.-K. Chang, H.-C. Chang, and C.-Y. Lee, "An Universal VLSI Architecture for Bit-Parallel Computation in $\mathrm{GF}(2^m)$", in *The 2004 IEEE Asia-Pacific Conference on Circuits and Systems*, Piscataway, NJ, USA: IEEE, Dec. 2004, pp. 125–128.

[35] H.-C. Chang, C. B. Shung, and C.-Y. Lee, "A Reed-Solomon Product-Code (RS-PC) Decoder Chip for DVD Applications", *IEEE Journal of Solid-State Circuits*, vol. 36, no. 8, pp. 229–238, Feb. 2001.

[36] A. Kumar and S. Sawitzki, "High Throughput and Low Power Reed Solomon Decoder for Ultra Wide Band", in *Intelligent Algorithms in Ambient and Biomedical Computing*, W. Verhaegh, E. Aarts, and J. Korst, Eds., ser. Philips Research Book Series, vol. 7, Dordrecht, The Netherlands: Springer, 2006, pp. 299–316.

[37] A. G. M. Strollo, N. Petra, D. De Caro, and E. Napoli, "An Area-Efficient High-Speed Reed-Solomon Decoder in $0.25\,\mu$m CMOS", in *Proceedings of the IEEE 30th European Solid State Circuits Conference*, Leuven, Belgium: IEEE, Sep. 2004, pp. 479–482.

[38] H.-Y. Hsu and A.-Y. Wu, "VLSI Design of a Reconfigurable Multi-Mode Reed-Solomon Codec for High-Speed Communication Systems", in *IEEE Asia-Pacific Conference on ASIC Proceedings*, Taipei, Taiwan: IEEE, Aug. 2002, pp. 359–362.

[39] J.-C. Huang, C.-M. Wu, M.-D. Shieh, and C.-H. Wu, "An Area-Efficient Versatile Reed-Solomon Decoder for ADSL", in *Proceedings of the 1999 IEEE International Symposium on Circuit and Systems (ISCAS'99)*, vol. 1, Orlando, FL, USA: IEEE, May 1999, pp. 517–520.

[40] F.-K. Chang, C.-C. Lin, H.-C. Chang, and C.-Y. Lee, "Universal Architectures for Reed-Solomon Error-and-Erasure Decoder", in *Asian Solid-State Circuits Conference*, Hsinchu, Taiwan: IEEE, Nov. 2005, pp. 229–232.

[41] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo Codes", in *Proceedings of the IEEE International Conference on Communication*, Geneva, Switzerland: IEEE, May 1993, pp. 1064–1070.

[42] J. T. M. H. Dielissen, "VLSI Design and Implementation of a Turbo Decoder", Master's Thesis, Eindhoven University of Technology, May 2000.

[43] B. Sklar and F. J. Harris, *Digital Communications: Fundamentals and Applications*, third edition. Hoboken, NJ, USA: Pearson Education, Inc., 2021.

[44] J. Hagenauer, E. Offer, and L. Papke, "Iterative Decoding of Binary Block and Convolutional Codes", *IEEE Transactions on Information Theory*, vol. 42, no. 2, pp. 429–445, Mar. 1996.

[45] R. G. Gallager, "Low-Density Parity-Check Codes", Sc. D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1963.

[46] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate", *IEEE Transactions on Information Theory*, vol. 20, no. 2, pp. 284–287, Mar. 1974.

[47] P. Robertson, E. Villebrun, and P. Hoeher, "A Comparison of Optimal and Sub-Optimal MAP Decoding Algorithms Operating in the Log Domain", in *IEEE International Conference on Communications*, Seattle, WA, USA: IEEE, Jun. 1995, pp. 1009–1013.

[48] A. J. Viterbi, "An Intuitive Justification and a Simplified Implementation of the MAP Decoder for Convolutional Codes", *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 2, pp. 260–264, Feb. 1998.

[49] G. Krishnaiah, N. Engin, and S. Sawitzki, "Scalable Reconfigurable Channel Decoder Architecture for Future Wireless Handsets", in *Design, Automation and Test in Europe Conference and Exhibition, DATE 2007*, Nice, France: European Design and Automation Association, Apr. 2007, pp. 1563–1568.

[50] Y. Sun and J. R. Cavallaro, "A Flexible LDPC/Turbo Decoder Architecture", *Journal of Signal Processing Systems*, vol. 64, no. 1, pp. 1–16, Jul. 2011.

[51] D. J. C. MacKay, "Good Error-Correcting Codes Based on Very Sparse Matrices", *IEEE Transactions on Information Theory*, vol. 45, no. 2, pp. 399–431, Mar. 1999.

[52] M. M. Mansour and N. R. Shanbhag, "High-Throughput LDPC Decoders", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 6, pp. 976–996, Dec. 2003.

[53] J. Lu and J. M. F. Moura, "Turbo-like Decoding of LDPC Codes", in *IEEE International Magnetics Conference (Intermag 2003)*, Boston, MA, USA: IEEE, Mar. 2003, DT–11.

[54] I. Ahmed and T. Arslan, "A Reconfigurable Viterbi Decoder for a Communication Platform", in *Proceedings of the 2005 International Conference on Field Programmable Logic and Applications (FPL)*, T. Rissa, S. Wilton, and P. Leong, Eds., Tampere, Finland: IEEE, Aug. 2005, pp. 435–440.