Final Project Report: BCH Hard Decision Decoding and Soft Decision Decoding

林裕翔（R14943148）　　李柏輝（R14943144）

December 24, 2025

1 Algorithm Design

1.1 Overall Architecture

The proposed decoder follows the standard algebraic BCH decoding and Chase soft-decision framework introduced in the course notes: syndrome computation, key equation solving (Berlekamp-based algorithm), Chien search, and correlation-based selection for soft decoding. Our architecture supports three code configurations, $(63, 51)$, $(255, 239)$, and $(1023, 983)$, and unifies both hard-decision and soft-decision decoding within a single pipeline.

At the top level, the decoder first receives log-likelihood ratios (LLRs) through a streaming interface. For hard-decision decoding, the LLRs are quantized to bits and directly fed into the hard-decision path. For soft-decision decoding, a Chase-style engine identifies the two least reliable positions, generates four test patterns by flipping these bits, and sends each pattern to a shared hard-decision decoder. Inside the hard-decision core, the *syndrome* block computes $S_1, \ldots, S_{2t}$ over GF(2^m), the *key equation solver* block derives the error-locator polynomial $\sigma(x)$ using an iterative Berlekamp-based algorithm, and the *Chien search* block locates all roots of $\sigma(x)$ to recover the error positions and produce the error-location numbers.

While this overall flow is aligned with the reference algorithm, our main contributions lie in how these four blocks are organized and optimized. The syndrome unit, key equation solver, Chien search, and correlation calculation are all redesigned to better exploit common substructures, reduce the number of finite-field operations, and improve hardware reuse between different code lengths and between hard and soft decoding. Detailed algorithmic refinements for each block are described in the following subsections.

1.2 Syndrome

1.2.1 Base Algorithm

In the baseline BCH decoder, the syndrome block computes the first $2t$ syndromes

$$S_j = \sum_{i=0}^{n-1} c_i \alpha^{ij}, \qquad j = 1, 2, \ldots, 2t, \tag{1}$$

where $c_i \in \{0, 1\}$ is the i-th hard-decision bit of the received codeword, $n \in \{63, 255, 1023\}$ is the code length, and $\alpha \in \mathrm{GF}(2^m)$ is a primitive element for the corresponding BCH code. These syndromes are then passed to the key equation solver.

In hardware, the $2t$ syndromes are accumulated on-the-fly as the codeword bits arrive. The input codeword is provided to the decoder starting from the most significant bit (MSB), i.e., from position $i = n - 1$ down to $i = 0$. To match this streaming order, our baseline implementation uses a "reverse" indexing: at each clock cycle, the new input bit c_i (starting from $i = n-1$) is broadcast to all $2t$ syndrome accumulators, and each S_j is updated according to a standard feedback form

$$S_j \leftarrow S_j \cdot \alpha^j + c_i, \tag{2}$$

so that, after n cycles, all $S_1, \ldots, S_{2t}$ are available at the output of the syndrome unit.

1.2.2 Proposed Improvement

For binary BCH codes over $\mathrm{GF}(2^m)$, the syndromes satisfy the well-known Frobenius relation

$$S_{2i} = S_i^2, \qquad i = 1, 2, \ldots, t, \tag{3}$$

because the mapping $x \mapsto x^2$ is an automorphism of $\mathrm{GF}(2^m)$. This means that the even-indexed syndromes $S_2, S_4, \ldots, S_{2t}$ are completely determined by the odd-indexed syndromes $S_1, S_3, \ldots, S_{2t-1}$.

We exploit this property in our design. Instead of accumulating all $2t$ syndromes directly from the input codeword, the proposed syndrome unit only computes the odd syndromes $S_1, S_3, \ldots, S_{2t-1}$ from the streaming bits, using the same MSB-first feedback structure as in the baseline. After the last codeword bit has been received, a dedicated $\mathrm{GF}(2^m)$ squaring block generates the even syndromes by

$$S_{2i} = \big(S_i\big)^2, \qquad i = 1, 2, \ldots, t. \tag{4}$$

Since squaring in $\mathrm{GF}(2^m)$ can be implemented by a fixed bit-permutation network, this approach avoids additional full multipliers and adds very little area and delay.

1.2.3 Comparison

In the baseline implementation, the input bit c_i is broadcast to all $2t$ syndrome accumulators in parallel. For the largest supported code with $t = 4$, this means each bit on the `i_data` bus must drive up to eight $\mathrm{GF}(2^m)$ accumulation paths (S_1 through S_8), which results in a relatively large capacitive load and can make the input port a critical timing point.

With the proposed architecture, the input codeword bits are only used to update the odd syndromes S_1, S_3, S_5, S_7. The even syndromes are generated afterwards by a local squarer network. As a result, the fanout from the `i_data` port to the syndrome unit is roughly halved, significantly reducing the effective capacitance seen by the input bus. This not only lowers dynamic power on the input signals but also helps to relax the timing constraints at the decoder interface, while the extra squaring logic introduces only modest area overhead.

1.3 Key Equation Solver

1.3.1 Base Algorithm (Berlekamp Algorithm)

In the baseline design, the key equation

$$S_1 + \sigma_1 = 0,\ S_2 + \sigma_1 S_1 + 2\sigma_2 = 0,\ \ldots \tag{5}$$

relates the $2t$ syndromes $\{S_i\}_{i=1}^{2t}$ to the unknown error–locator polynomial

$$\sigma(X) = 1 + \sigma_1 X + \sigma_2 X^2 + \cdots + \sigma_v X^v. \tag{6}$$

Directly solving this system is expensive, so we adopt the classical Berlekamp iterative algorithm used in the course notes.

The algorithm constructs a sequence of intermediate polynomials $\sigma_\mu(X)$, indexed by the iteration number μ, together with their degrees ℓ_μ and discrepancies d_μ. Each $\sigma_\mu(X)$ is the current estimate of the error–locator polynomial after using syndromes up to $S_{\mu+1}$, while d_μ measures how well $\sigma_\mu(X)$ satisfies the key equation at this step. The procedure is initialized with trivial polynomials (e.g., $\sigma_{-1}(X) = 1$, $\sigma_0(X) = 1$) and corresponding (d_μ, ℓ_μ) values derived from S_1.

For each iteration $\mu = 0, 1, \ldots, 2t-1$, a new row $(\sigma_{\mu+1}(X), d_{\mu+1}, \ell_{\mu+1})$ is computed from the previous rows. If the current discrepancy is zero ($d_\mu = 0$), the polynomial is simply propagated,

$$\sigma_{\mu+1}(X) = \sigma_\mu(X), \qquad \ell_{\mu+1} = \ell_\mu, \tag{7}$$

and the new discrepancy $d_{\mu+1}$ is recomputed from the next syndrome using the updated coefficients. If $d_\mu \neq 0$, the algorithm selects a previous row index $\rho < \mu$ with non-zero discrepancy d_ρ and the largest value of $(\rho - \ell_\rho)$. The polynomial is then updated by

$$\sigma_{\mu+1}(X) = \sigma_\mu(X) + d_\mu d_\rho^{-1} X^{\mu-\rho} \sigma_\rho(X), \tag{8}$$

and the degree is set to

$$\ell_{\mu+1} = \max\big(\ell_\mu,\ \ell_\rho + \mu - \rho\big). \tag{9}$$

After each update, the new discrepancy $d_{\mu+1}$ is computed as

$$d_{\mu+1} = S_{\mu+2} + \sigma_{1,\mu+1} S_{\mu+1} + \cdots + \sigma_{\ell_{\mu+1},\mu+1} S_{\mu+2-\ell_{\mu+1}}, \tag{10}$$

where $\sigma_{i,\mu+1}$ is the coefficient of X^i in $\sigma_{\mu+1}(X)$. The iterations terminate after $\mu = 2t-1$, and the final polynomial $\sigma_{2t}(X)$ is taken as the error–locator polynomial $\sigma(X)$ used in the subsequent Chien search stage.

1.3.2 Proposed Improvement (inversionless Berlekamp–Massey Algorithm)

To reduce the complexity of the key equation solver and make the datapath more regular, we adopt the new inversionless Berlekamp–Massey algorithm proposed in [1]. Instead of explicitly computing the inverse of the discrepancy in each iteration, the algorithm works on two auxiliary polynomials $\Phi^{(r)}(z)$

and $\Psi^{(r)}(z)$ and an integer variable $\ell^{(r)}$, and updates them using only finite-field additions and multiplications. This inversionless formulation is well-suited for a systolic hardware implementation and can be shared by different BCH codes.

In our implementation, the inversionless BM algorithm is summarized as follows.

Algorithm 1: Inversionless Berlekamp–Massey algorithm [1]
beginequation 0.3em] *Input:* syndromes $S_0, S_1, \ldots, S_{2t-1}$ and $S(z) = S_0 + S_1 z + \cdots + S_{2t-1} z^{2t-1}$.
Initialization: $\Phi^{(0)}(z) = S(z) + z^{3t}$, $\Psi^{(0)}(z) = 1$, $\ell^{(0)} = 0$.
For $r = 0, 1, \ldots, 2t-1$ *do*

- Let $\Phi_0^{(r)}$ and $\Psi_0^{(r)}$ be the constant terms of $\Phi^{(r)}(z)$ and $\Psi^{(r)}(z)$.
- Update

$$\Phi^{(r+1)}(z) = \frac{\Psi_0^{(r)}\,\Phi^{(r)}(z) - \Phi_0^{(r)}\,\Psi^{(r)}(z)}{z}. \tag{11}$$

- If $\Phi_0^{(r)} \neq 0$ and $\ell^{(r)} \geq 0$, then

$$\Psi^{(r+1)}(z) \leftarrow \Phi^{(r)}(z), \qquad \ell^{(r+1)} \leftarrow -\ell^{(r)} - 1; \tag{12}$$

else

$$\Psi^{(r+1)}(z) \leftarrow \Psi^{(r)}(z), \qquad \ell^{(r+1)} \leftarrow \ell^{(r)} + 1. \tag{13}$$

End for
Output: polynomial $\Phi^{(2t)}(z)$, from which the error-locator polynomial and the complementary error-evaluator polynomial can be extracted.

As shown in [1], $\Phi^{(r)}(z)$ and $\Psi^{(r)}(z)$ are related to the conventional Berlekamp polynomials. In particular, at iteration r we have

$$\Phi^{(r)}(z) = \hat{\Theta}^{(r)}(z) + z^t \hat{\sigma}^{(r)}(z), \qquad \ell^{(r)} = r - 2\hat{v}^{(r)}, \tag{14}$$

where $\hat{\sigma}^{(r)}(z)$ is a scaled version of the error-locator polynomial and $\hat{v}^{(r)} = \deg \hat{\sigma}^{(r)}$. Therefore, after $r = 2t$ iterations the degree of the locator polynomial can be recovered as

$$\hat{v}^{(2t)} = \frac{2t - \ell^{(2t)}}{2}. \tag{15}$$

We exploit this property in our soft-decision decoder. In a Chase decoder, some test patterns may contain more than t errors (for example, three bit errors for a $(255, 239)$ BCH code with $t = 2$). In such cases, the Berlekamp iterations may converge to a polynomial whose degree is larger than t. Using the degree relation above, we can check whether $\hat{v}^{(2t)} \leq t$ after the iterations. If the degree exceeds t, we directly declare that this test pattern has failed and do not send its locator polynomial to the Chien search. This provides a clean, theorem-based failure detection criterion for the soft-decision decoder instead of relying only on correlation values.

In addition, since $\Phi^{(2t)}(z)$ packs both the complementary error-evaluator and the error-locator polynomial into a single length-$(2t+1)$ polynomial, our hardware only needs one register bank for $\Phi(z)$ rather than separate banks for $\sigma(z)$ and $\Theta(z)$. The same datapath is reused for all supported BCH codes and for both hard- and soft-decision modes.

1.3.3 Comparison

Compared with the baseline Berlekamp algorithm described in the previous subsection, the proposed inversionless implementation mainly improves the hardware cost of the key equation solver. In the conventional Berlekamp–Massey algorithm, every nonzero discrepancy requires a multiplication by δ^{-1} in GF(2^m). A straightforward hardware implementation uses a lookup table (LUT) or a dedicated inversion unit. Because our decoder has to support several BCH codes with different field sizes and primitive polynomials, these LUTs cannot be shared efficiently across codes and would consume a large amount of area. The inversionless algorithm [1] completely removes field inversions; each processing element only contains a small number of multipliers and adders, and the same structure can be reused for all supported codes. Overall, this leads to a more hardware-friendly key equation solver that is significantly more resource-efficient when multiple BCH codes must be integrated into a single design.

1.4 Chien Search

The Chien search stage locates all roots of the error–locator polynomial $\sigma(x)$ obtained from the key equation solver. For a length-n BCH code, an error at position i (counting from 0 to $n-1$) corresponds to a root at $x = \alpha^{-i}$, and the conventional Chien search evaluates

$$\sigma(\alpha^0),\ \sigma(\alpha^{-1}),\ \ldots,\ \sigma(\alpha^{-(n-1)}) \tag{16}$$

sequentially until all positions have been checked. This purely serial implementation requires up to n cycles and can easily become the bottleneck of the decoder.

To improve throughput, we adopt a parallel Chien search. Let P denote the Chien search parallelism, i.e., the number of evaluation points processed in one cycle. At cycle k ($k = 0, 1, \ldots$), the Chien search simultaneously evaluates $\sigma(x)$ at

$$x_{k,p} = \alpha^{-(kP+p)}, \qquad p = 0, 1, \ldots, P-1, \tag{17}$$

and packs the results into a P-bit pattern

$$\mathbf{y}^{(k)} = \big(y_0^{(k)}, y_1^{(k)}, \ldots, y_{P-1}^{(k)}\big), \tag{18}$$

where $y_p^{(k)} = 1$ if and only if $\sigma(x_{k,p}) = 0$. Because the decoder corrects at most $t \le 4$ errors, each $\mathbf{y}^{(k)}$ contains at most four ones. Over $\lceil n/P \rceil$ cycles, the entire codeword is covered.

For each window $\mathbf{y}^{(k)}$, we still need to translate the P-bit pattern into the indices of the set bits, i.e., the error positions in the range $kP, \ldots, kP + P - 1$. Instead of scanning $\mathbf{y}^{(k)}$ bit by bit, we use a tree-based index–extraction scheme. Conceptually, the P bits are first partitioned into small groups of size B (e.g., $B = 8$). Each group locally computes

- the number of ones in its B bits (clipped at t), and
- up to t local indices where the bits are equal to one.

These local results are then recursively merged by a binary tree of “merge” operations. Each merge node takes the outputs from two child groups, adds their counts (again clipped at t), shifts the indices of

the upper group by the appropriate offset, and selects the first at most t indices in order. After $\log_2(P/B)$ levels, the root of the tree produces a global list of indices

$$i_1^{(k)},\ i_2^{(k)},\ i_3^{(k)},\ i_4^{(k)} \tag{19}$$

and the total number of roots in the current window.

Figure 1 illustrates a simplified example of this tree-based index extraction for a small P.

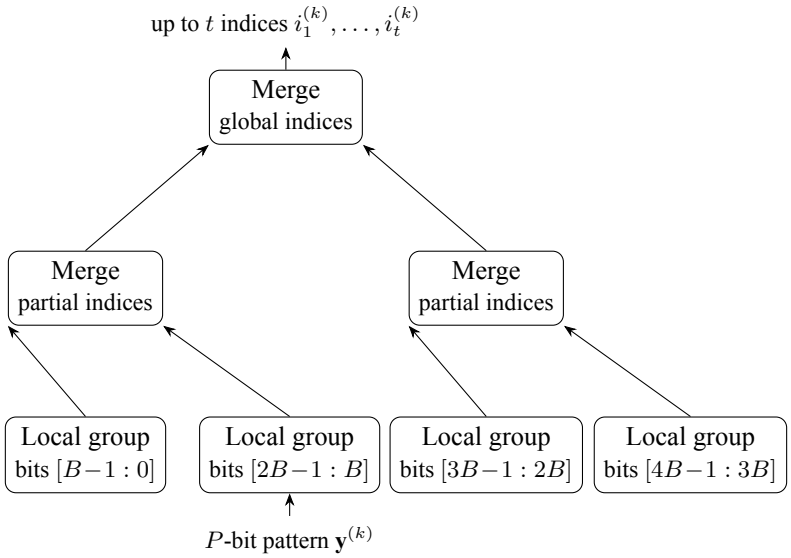


Figure 1: Conceptual tree-based index extraction for the parallel Chien search. The P-bit pattern $\mathbf{y}^{(k)}$ (with P evaluations per cycle) is split into local groups of size B. Each group finds local ones and their positions, and a merge tree combines these partial results to obtain the global indices of the roots in the current window.

Here, P denotes the degree of parallelism in the Chien search (number of evaluation points processed per cycle), and B denotes the local group size used inside the tree. In the implementation, P and B are chosen to balance the trade-off between area, delay, and throughput, but the algorithmic structure follows the same parallel Chien search plus tree-based index extraction described above.

1.5 Correlation Calculation

1.5.1 Base Algorithm

In the course note, the Chase soft-decision decoder evaluates each successfully decoded test pattern by a correlation metric. Given the LLR sequence $\{L_i\}_{i=0}^{n-1}$ of the received word and a decoded candidate codeword $\hat{c}(X) = \hat{c}_0 + \hat{c}_1 X + \cdots + \hat{c}_{n-1} X^{n-1}$, the correlation is defined as

$$\mathrm{Corr}(\hat{c}) = \sum_{i=0}^{n-1} L_i\big(1 - 2\hat{c}_i\big), \tag{20}$$

where $\hat{c}_i \in \{0, 1\}$ is treated as a real-valued number. If $\hat{c}_i = 0$, the factor $(1 - 2\hat{c}_i)$ equals $+1$ and the i-th LLR contributes $+L_i$ to the metric; if $\hat{c}_i = 1$, the factor equals -1 and the contribution is $-L_i$. The Chase algorithm then selects, among all patterns that pass hard-decision decoding, the candidate with

the largest correlation value as the final output.

1.5.2 Proposed Improvement

In our implementation, we exploit the structure of the correlation metric and the relation between the LLR sign and the hard-decision bit to derive a much simpler expression that depends only on the error positions of each test pattern.

Let r_i denote the hard-decision bit of the received word before Chase flipping:

$$r_i = \begin{cases} 0, & L_i > 0, \\ 1, & L_i < 0. \end{cases} \tag{21}$$

Consider a position i where the final decoded bit should be corrected relative to r_i (i.e., there is an error at bit i in the input codeword):

- If $L_i < 0$, the channel suggests a "1" and we have $r_i = 1$ while the correct bit is $\hat{c}_i = 0$. In the correlation expression, changing $\hat{c}_i$ from 1 to 0 changes the term from $-L_i$ to $+L_i$, which can be written as $-|L_i|$ relative to a symmetric reference.
- If $L_i > 0$, the channel suggests a "0" and we have $r_i = 0$ while the correct bit is $\hat{c}_i = 1$. Changing $\hat{c}_i$ from 0 to 1 changes the term from $+L_i$ to $-L_i$, which again corresponds to $-|L_i|$ relative to the same reference.

Therefore, for any position i at which the decoded candidate contains an error relative to the channel observation, flipping that bit to the correct value decreases the correlation by $|L_i|$ in a symmetric sense, irrespective of the sign of L_i. Now consider two test patterns j and k that are identical except for bit i: pattern j has an error at position i while pattern k does not. Then, compared with a common baseline, pattern j effectively suffers a penalty of $-|L_i|$ in the correlation, whereas pattern k gains $+|L_i|$.

Since only the relative ordering of correlation values matters, we can subtract the same constant from all patterns and scale by $1/2$ without changing which pattern is selected. After subtracting $\sum_{l=0}^{n-1} |\ell_l|$ and dividing by 2, the adjusted correlation for a pattern j can be written as

$$\widetilde{\text{Corr}}(j) = -\sum_{i \in \mathcal{E}_j} |L_i|, \tag{22}$$

where $\mathcal{E}_j$ is the set of bit positions that are in error for pattern j (i.e., positions where the decoded bit disagrees with the channel hard decision). Equivalently, we can define a non-negative "cost"

$$\text{Cost}(j) = \sum_{i \in \mathcal{E}_j} |L_i|, \tag{23}$$

and select the pattern with the smallest cost instead of the largest correlation. This transformation is particularly attractive for hardware implementation: each pattern only needs to accumulate the magnitudes $|L_i|$ over its error positions, without revisiting all n bits or handling signed additions.

To avoid signed arithmetic, we further implement the metric directly as $\text{Cost}(j)$: whenever the error locator indicates that bit i belongs to $\mathcal{E}_j$, we simply add $|L_i|$ to the pattern's cost accumulator. The final

decoded result is the pattern with the minimum cost, and its error-location set $\mathcal{E}_j$ is used as the output of the decoder.

1.5.3 Comparison

Compared with the baseline correlation computation, the proposed formulation significantly reduces both arithmetic complexity and storage requirements, especially for the $(1023, 983)$ BCH code. In the original approach, each of the four Chase test patterns must evaluate

$$\text{Corr}(\hat{c}) = \sum_{i=0}^{n-1} L_i(1 - 2\hat{c}_i) \tag{24}$$

over all n bits. For $n = 1023$ and 8-bit signed LLRs, the accumulator must handle a dynamic range on the order of $1023 \times |L_i|$, which requires roughly $\lceil \log_2 1023 \rceil + 8$ bits (about 17 bits) per correlation value and full-length accumulation per pattern.

In contrast, our method uses only the error positions reported by the hard-decision decoder. For the $(1023, 983)$ code with $t = 4$ and 2 flipped least-reliable bits in the Chase algorithm, the number of errors in any pattern is at most six. The cost of a pattern is thus the sum of at most six non-negative magnitudes $|L_i|$, each requiring only 7 bits. The resulting accumulator width is on the order of $\lceil \log_2 6 \rceil + 7$ bits (about 10 bits), and the number of additions per pattern is bounded by the number of error locations instead of the code length.

Furthermore, because the correlation decision depends only on $|L_i|$ in our transformed metric, we can store only the magnitude of each LLR in the on-chip memory, reducing the LLR buffer from 1023×8 bits to 1023×7 bits for the longest code. Overall, the proposed correlation calculation achieves a much more hardware-efficient metric computation while preserving the same soft-decision performance as the original Chase formulation.

2 Hardware Implementation

2.1 Overall Architecture

Figure 2 shows the top-level datapath of the proposed BCH decoder. The core accepts a 64-bit LLR input word `idata` and produces a 10-bit error-location output `odata`. Internally, all arithmetic over $\text{GF}(2^m)$ is implemented using a common finite-field multiplier that is reused in the *Syndrome Unit*, the *Key Equation Solver Unit*, and the *Chien Search Unit*. The figure focuses on how data moves between these blocks; the detailed control FSM and handshake signals are handled in the RTL but are not drawn here.

The green blocks correspond to hardware that is only enabled in the soft-decision decoder, whereas the blue blocks are shared by both hard- and soft-decision modes:

- **smallest LLR** (green) scans the incoming 64-bit LLR word `idata` and finds the positions and magnitudes of the two least-reliable bits (two smallest $|\text{LLR}_i|$). The positions are encoded as two 10-bit indices ("2×10 bit" in Fig. 2). These indices are used both to construct flipped patterns for soft-decision decoding and, in the early-stop case, to directly form error locations without running the key-equation solver and Chien search.

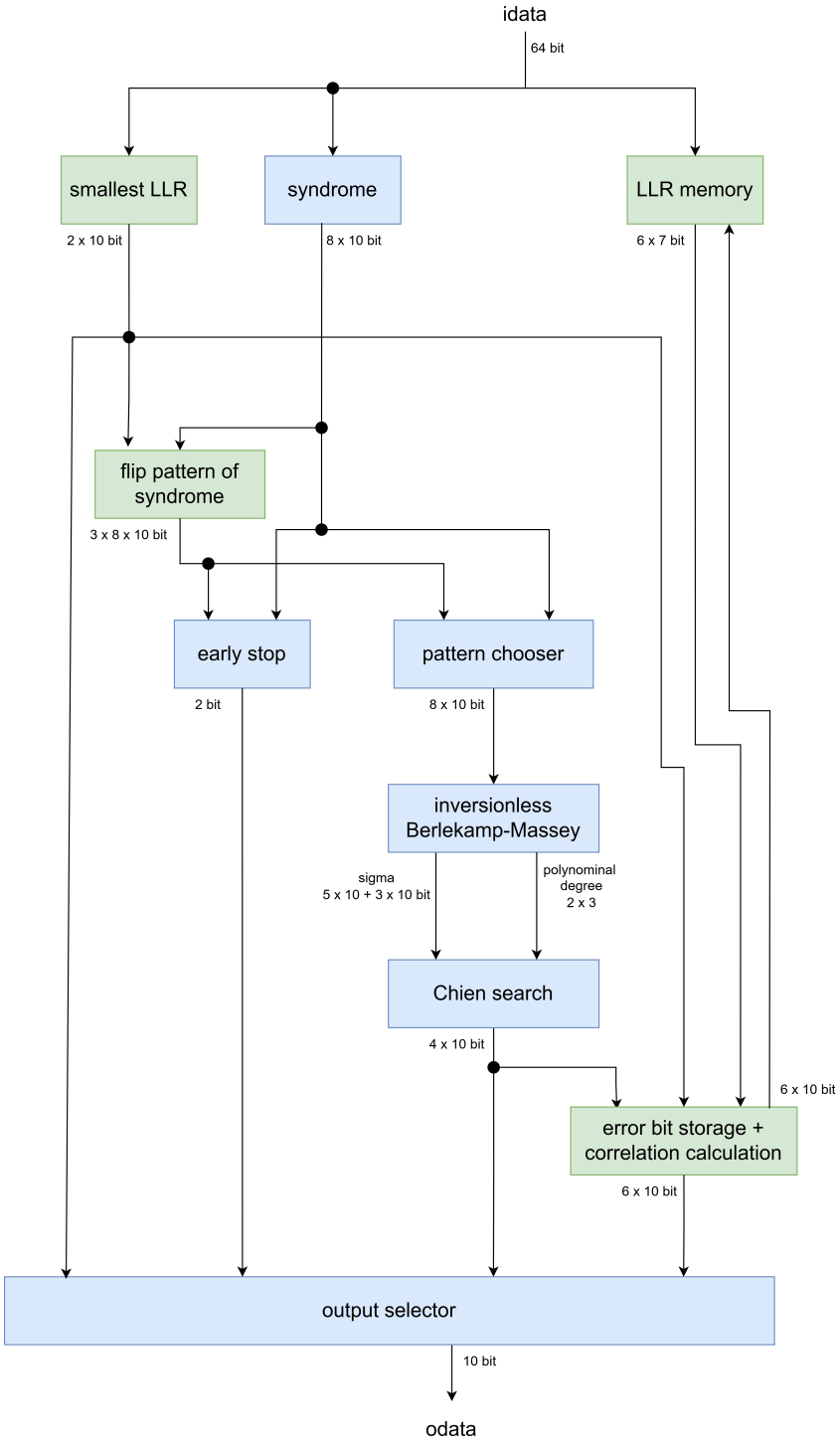


Figure 2: Datapath of the proposed BCH hard/soft-decision decoder. Only dataflow connections are shown; handshake and control signals (`set`, `ready`, `finish`, valid flags, etc.) are omitted for clarity. Blue blocks are shared by both hard- and soft-decision decoding, while green blocks are only used in the soft-decision mode.

- **syndrome** (blue) is the *Syndrome Unit*. It computes the eight syndromes $S_1, \ldots, S_8$ (each 10 bits wide in the unified representation) from `idata` for the selected code $(63, 51)$, $(255, 239)$, or $(1023, 983)$. The 8×10-bit syndrome vector is broadcast to the early-stop logic, the flipped-pattern generator, and the pattern chooser.

- **LLR memory** (green) stores the absolute values of the LLRs $|\mathrm{LLR}_i|$ for the whole codeword. Only the magnitude is kept, encoded as 7-bit words, which reduces the memory size while allowing random access by downstream blocks such as the Chien search and the correlation calculator. In the diagram this interface is summarized as "6×7 bit" and "6×10 bit" buses, corresponding to the number of error locations that may be accessed in parallel.

- **flip pattern of syndrome** (green) uses the original syndromes and the two smallest-LLR positions to generate the syndromes of three additional patterns corresponding to flipping those unreliable bits. Together with the original (unflipped) pattern, up to four candidate patterns are formed for soft-decision decoding. The output is shown as "$3 \times 8 \times 10$ bit" in Fig. 2.

- **early stop** (blue) examines the odd-order syndromes S_1, S_3, S_5, S_7 for each candidate pattern. In hard-decision mode it reduces to the usual no-error test on the original syndrome: if all odd syndromes are zero, the decoder can bypass the key-equation solver and Chien search and directly declare that no error locations need to be output.

  In soft-decision mode, the block simultaneously checks up to four patterns (the original pattern and three flipped patterns generated by **flip pattern of syndrome**). For each pattern it determines whether $S_1 = S_3 = S_5 = S_7 = 0$ holds. If at least one pattern passes this test, **early stop** asserts an early-stop flag and outputs a 2-bit pattern index (labelled "2 bit" in Fig. 2) that encodes which pattern should be used. The actual error positions are then taken from the two 10-bit indices produced by **smallest LLR**: depending on the selected pattern, the output selector will emit no error (original pattern), one error (flip only the first or only the second least-reliable bit), or two errors (flip both positions), while completely bypassing the key-equation solver and Chien search.

- **pattern chooser** (blue) selects which syndrome pattern is fed into the key equation solver. In hard-decision mode, it simply passes the original 8×10-bit syndrome vector from the **syndrome** block. In soft-decision mode, it sequentially outputs the syndrome vectors of the four patterns generated by **flip pattern of syndrome**. For $(63, 51)$ and $(255, 239)$, the first 4×10 bits correspond to pattern 1 and the second 4×10 bits to pattern 2; for $(1023, 983)$, the full 8×10-bit vector is treated as a single pattern.

- **inversionless Berlekamp–Massey** (blue) is the *Key Equation Solver Unit*. It consumes an 8×10-bit syndrome vector from the **pattern chooser** and computes one or two error-locator polynomials in $GF(2^m)$ using an inversionless Berlekamp–Massey recursion. All coefficients are represented in the unified 10-bit field format and the block explicitly outputs the constant terms σ_0 as well, because in the chosen formulation the locator polynomials are *not* normalized to be monic (i.e., $\sigma_0 \neq 1$ cannot be assumed).

  The output interface is shown as "σ-coeff. $5 \times 10 + 3 \times 10$ bit" and "polynomial degree 2×3 bit" in Fig. 2. This corresponds to up to five 10-bit coefficients $(\sigma_0^{(1)}, \ldots, \sigma_4^{(1)})$ for the first polynomial,

up to three 10-bit coefficients $(\sigma_0^{(2)}, \ldots, \sigma_2^{(2)})$ for a second polynomial, and two 3-bit degree fields indicating the actual degrees of the polynomials.

The way these outputs are used depends on the decoding mode and the code:

  - *Hard-decision decoding:* only a single locator polynomial is needed. The decoder uses the first set of five 10-bit coefficients. For the $(63, 51)$ and $(255, 239)$ codes ($t = 2$), the third- and fourth-order coefficients are always zero, but they share the same 5-word interface. For the $(1023, 983)$ code ($t = 4$), all five coefficients may be non-zero.
  - *Soft-decision decoding for* $(63, 51)$ *and* $(255, 239)$*:* the **pattern chooser** provides two syndrome vectors at a time, corresponding to two candidate patterns. The key equation solver reuses the same arithmetic core to solve both key equations in parallel and therefore outputs two polynomials: the first one through the 5×10-bit port and the second one through the 3×10-bit port (only up to degree 2 is needed for these codes). This allows two patterns to share the same hardware without duplicating the entire solver.
  - *Soft-decision decoding for* $(1023, 983)$*:* only one syndrome vector is processed at a time, so effectively only the 5×10-bit polynomial is used; the second (3×10) output is ignored for this code.

- **Chien search** (blue) evaluates the error-locator polynomial(s) and produces up to four error locations for each pattern, summarized as "4×10 bit" in Fig. 2. For the $(63, 51)$ and $(255, 239)$ codes, two polynomials can be processed in parallel at the input, but the error locations are streamed out one by one. For $(1023, 983)$, one full Chien search is executed and the error locations for that pattern are output in a single pass.

- **error bit storage + correlation calculation** (green) stores the error-location sets and correlation metrics for up to four soft-decision patterns. For each candidate pattern, it receives up to six 10-bit error locations from **Chien search** ("6×10 bit"), uses these locations to read the corresponding $|\mathrm{LLR}_i|$ values from the **LLR memory**, and accumulates a correlation metric based only on these error bits. Once all candidate patterns have been processed, it compares their metrics and selects the pattern with the smallest value (i.e., the most likely codeword). The selected pattern's error locations are then forwarded to the output selector.

- **output selector** (blue) is the final interface to the system. It receives three kinds of candidate error-location sets:

  - from **early stop**: a flag and a 2-bit pattern index indicating which pattern (if any) satisfies the zero-odd-syndrome condition; together with the two 10-bit positions from **smallest LLR**, this determines a trivial set of 0, 1 or 2 error locations;
  - directly from **Chien search**: up to four 10-bit error-location indices per pattern ("4×10 bit"), used in hard-decision decoding and when soft-decision logic is disabled;
  - from **error bit storage + correlation calculation**: up to six 10-bit error-location indices ("6×10 bit") of the soft-decision pattern with the best correlation metric.

Based on the decoding mode and the control flags, the output selector chooses one of these sources and serializes the corresponding error locations. The indices are driven on the 10-bit output `odata`, one location per cycle, and the signal `finish` is asserted when the last error location of the chosen set has been emitted.

The following subsections detail the shared arithmetic blocks that dominate area and timing: the finite-field multiplier architecture, the Syndrome Unit, the Key Equation Solver Unit (inversionless Berlekamp–Massey), and the Chien Search Unit.

2.2 Finite Field Multiplier

The decoder requires finite-field multiplication over $\mathrm{GF}(2^m)$ for $m \in \{6, 8, 10\}$, corresponding to $\mathrm{BCH}(63, 51)$, $\mathrm{BCH}(255, 239)$, and $\mathrm{BCH}(1023, 983)$, respectively. To reduce area, we implement a single unified bit-parallel multiplier with a 10-bit interface that is reused across all modes. Elements are represented as binary polynomials

$$A(x) = \sum_{i=0}^{9} a_i x^i, \qquad B(x) = \sum_{j=0}^{9} b_j x^j, \tag{25}$$

where $(a_9, \ldots, a_0)$ and $(b_9, \ldots, b_0)$ are the 10-bit inputs. For $\mathrm{GF}(2^6)$ and $\mathrm{GF}(2^8)$, the most significant bits are always zero:

$$\mathrm{GF}(2^6): \ a_i = b_i = 0, \ \forall i \geq 6, \qquad \mathrm{GF}(2^8): \ a_i = b_i = 0, \ \forall i \geq 8, \tag{26}$$

so the same 10×10 partial-product network can be shared among all three BCH codes.

Stage 1: convolution (unreduced product). The first stage computes the unreduced product

$$R(x) = A(x)\,B(x) = \sum_{k=0}^{18} c_k x^k \tag{27}$$

by a convolution-like AND–XOR array. Each coefficient c_k is the XOR of all partial products $a_i b_j$ such that $i + j = k$:

$$c_k = \bigoplus_{\substack{0 \leq i,j \leq 9 \\ i+j=k}} (a_i b_j), \qquad k = 0, 1, \ldots, 18. \tag{28}$$

This corresponds exactly to the combinational logic that generates `row_xor[0] ... row_xor[18]` in the RTL. Structurally, this stage is a fixed Toeplitz-like convolution network that does not depend on the chosen field degree m; only the input patterns (a_i, b_j) differ between modes due to zero padding in the $\mathrm{GF}(2^6)$ and $\mathrm{GF}(2^8)$ cases.

Stage 2: modular reduction. The second stage performs modular reduction of $R(x)$ with respect to the primitive polynomial $p_m(x)$ of the selected field:

$$p_6(x) = x^6 + x + 1, \quad p_8(x) = x^8 + x^4 + x^3 + x^2 + 1, \quad p_{10}(x) = x^{10} + x^3 + 1. \tag{29}$$

For each mode, we compute

$$P^{(m)}(x) = R(x) \bmod p_m(x), \tag{30}$$

and output the lower m coefficients of $P^{(m)}(x)$ as the finite-field product.

Instead of explicitly computing the polynomial long division at run time, we pre-derive the XOR combinations of $\{c_k\}$ that implement the reduction. Let us denote the output coefficients by $d_i^{(m)}$ for $i = 0, \ldots, m-1$. The reduction logic in the RTL matches the following expressions.

GF(2^6) mode ($m = 6$)

For $p_6(x) = x^6 + x + 1$, the reduced product

$$P^{(6)}(x) = \sum_{i=0}^{5} d_i^{(6)} x^i \tag{31}$$

is obtained from $c_0, \ldots, c_{10}$ as

$$d_0^{(6)} = c_0 \oplus c_6, \tag{32}$$
$$d_1^{(6)} = c_1 \oplus c_6 \oplus c_7, \tag{33}$$
$$d_2^{(6)} = c_2 \oplus c_7 \oplus c_8, \tag{34}$$
$$d_3^{(6)} = c_3 \oplus c_8 \oplus c_9, \tag{35}$$
$$d_4^{(6)} = c_4 \oplus c_9 \oplus c_{10}, \tag{36}$$
$$d_5^{(6)} = c_5 \oplus c_{10}. \tag{37}$$

The higher bits are forced to zero:

$$d_i^{(6)} = 0, \quad i = 6, 7, 8, 9. \tag{38}$$

GF(2^8) mode ($m = 8$)

For $p_8(x) = x^8 + x^4 + x^3 + x^2 + 1$, the reduced product

$$P^{(8)}(x) = \sum_{i=0}^{7} d_i^{(8)} x^i \tag{39}$$

is obtained from $c_0, \ldots, c_{14}$ as

$$d_0^{(8)} = c_0 \oplus c_8 \oplus c_{12} \oplus c_{13} \oplus c_{14}, \tag{40}$$
$$d_1^{(8)} = c_1 \oplus c_9 \oplus c_{13} \oplus c_{14}, \tag{41}$$
$$d_2^{(8)} = c_2 \oplus c_8 \oplus c_{10} \oplus c_{12} \oplus c_{13}, \tag{42}$$
$$d_3^{(8)} = c_3 \oplus c_8 \oplus c_9 \oplus c_{11} \oplus c_{12}, \tag{43}$$
$$d_4^{(8)} = c_4 \oplus c_8 \oplus c_9 \oplus c_{10} \oplus c_{14}, \tag{44}$$
$$d_5^{(8)} = c_5 \oplus c_9 \oplus c_{10} \oplus c_{11}, \tag{45}$$
$$d_6^{(8)} = c_6 \oplus c_{10} \oplus c_{11} \oplus c_{12}, \tag{46}$$
$$d_7^{(8)} = c_7 \oplus c_{11} \oplus c_{12} \oplus c_{13}, \tag{47}$$

and again

$$d_i^{(8)} = 0, \quad i = 8, 9. \tag{48}$$

GF(2^{10}) mode ($m = 10$)

For $p_{10}(x) = x^{10} + x^3 + 1$, the reduced product

$$P^{(10)}(x) = \sum_{i=0}^{9} d_i^{(10)} x^i \tag{49}$$

is obtained from $c_0, \ldots, c_{18}$ as

$$d_0^{(10)} = c_0 \oplus c_{10} \oplus c_{17}, \tag{50}$$
$$d_1^{(10)} = c_1 \oplus c_{11} \oplus c_{18}, \tag{51}$$
$$d_2^{(10)} = c_2 \oplus c_{12}, \tag{52}$$
$$d_3^{(10)} = c_3 \oplus c_{10} \oplus c_{13} \oplus c_{17}, \tag{53}$$
$$d_4^{(10)} = c_4 \oplus c_{11} \oplus c_{14} \oplus c_{18}, \tag{54}$$
$$d_5^{(10)} = c_5 \oplus c_{12} \oplus c_{15}, \tag{55}$$
$$d_6^{(10)} = c_6 \oplus c_{13} \oplus c_{16}, \tag{56}$$
$$d_7^{(10)} = c_7 \oplus c_{14} \oplus c_{17}, \tag{57}$$
$$d_8^{(10)} = c_8 \oplus c_{15} \oplus c_{18}, \tag{58}$$
$$d_9^{(10)} = c_9 \oplus c_{16}. \tag{59}$$

In summary, the multiplier consists of a single shared convolution stage that produces the intermediate coefficients $\{c_k\}$, followed by a small mode-dependent XOR network implementing modular reduction for $p_6(x)$, $p_8(x)$, or $p_{10}(x)$. Because the GF(2^6) and GF(2^8) operands are embedded into the 10-bit interface with fixed zero MSBs, the same hardware can be reused across all BCH modes without duplicating multipliers for different field sizes.

2.3 Syndrome Unit

The Syndrome Unit computes the first eight syndromes $S_1, \ldots, S_8$ required by all supported BCH codes $(63, 51)$, $(255, 239)$, and $(1023, 983)$ over the finite field GF(2^m). The datapath is organized so that one 64-bit word is processed per cycle: each word contains eight LLR values, and only their sign bits are used as hard decisions $r_k \in \{0, 1\}$. Thus the unit effectively processes eight code bits in parallel per clock.

For a received codeword of length n, the odd-order syndromes are defined as

$$S_\ell = \sum_{k=0}^{n-1} r_k \alpha^{\ell k}, \qquad \ell \in \{1, 3, 5, 7\}, \tag{60}$$

where α is a primitive element of GF(2^m). To exploit the eight-bit parallelism, the index k is decomposed as $k = 8i + b$ with block index $i \geq 0$ and intra-block position $b \in \{0, \ldots, 7\}$:

$$S_\ell = \sum_i \sum_{b=0}^{7} r_{8i+b} \alpha^{\ell(8i+b)}. \tag{61}$$

The implementation precomputes suitable powers of α so that the contribution of each group of eight bits can be evaluated with simple bit masking and XOR operations.

At the beginning of each codeword, when a "clear and write" signal is asserted, the unit initializes four arrays of field elements $\{P_b^{(\ell)}(0)\}_{b=0}^{7}$ for $\ell \in \{1, 3, 5, 7\}$. Each $P_b^{(\ell)}(0)$ is a precomputed constant corresponding to $\alpha^{-\kappa_{\ell,b}}$ for the chosen code; these constants are stored as 10-bit vectors ($m = 6, 8, 10$ are all embedded into a unified 10-bit representation). For $(63, 51)$ and $(255, 239)$ only the arrays for $\ell = 1$ and $\ell = 3$ are used; the arrays for $\ell = 5$ and $\ell = 7$ are enabled only for $(1023, 983)$.

In the first active cycle, the odd-order syndromes are formed directly from the current group of eight hard decisions:

$$S_\ell^{(0)} = \sum_{b=0}^{7} r_b \, P_b^{(\ell)}(0), \qquad \ell \in \{1, 3, 5, 7\}. \tag{62}$$

Each product $r_b \, P_b^{(\ell)}(0)$ is implemented as a bitwise AND between the 10-bit constant $P_b^{(\ell)}(0)$ and a replicated hard decision r_b, and the eight terms are accumulated by XOR. This realizes the multiplication by $\alpha^{-\kappa_{\ell,b}}$ for all eight bits in parallel without a general-purpose multiplier.

For every subsequent group of eight bits, the same computation is repeated with updated powers of α. The update obeys the recurrence

$$P_b^{(\ell)}(i+1) = P_b^{(\ell)}(i) \, \alpha^{-8\ell}, \qquad b = 0, \ldots, 7, \tag{63}$$

$$S_\ell^{(i+1)} = S_\ell^{(i)} \oplus \sum_{b=0}^{7} r_{8(i+1)+b} \, P_b^{(\ell)}(i+1), \tag{64}$$

so that each cycle multiplies all basis elements by $\alpha^{-8\ell}$ and adds the contribution of the new eight bits. These updates are implemented by feeding the arrays $P_b^{(\ell)}$ into a bank of shared finite-field multipliers with constants corresponding to α^{-8}, α^{-24}, α^{-40}, and α^{-56} for $\ell = 1, 3, 5, 7$, respectively. In this way, the same structural pattern is reused across all three code lengths: the unit always consumes eight bits per clock and gradually shifts the evaluation point of the syndrome polynomials along the codeword.

The even-order syndromes are not computed by independent polynomial evaluations. Instead, the unit exploits the algebraic relations valid in characteristic two:

$$S_2 = S_1^2, \qquad S_4 = S_2^2, \qquad S_6 = S_3^2, \qquad S_8 = S_4^2. \tag{65}$$

After S_1 and S_3 (and, for the longest code, S_5 and S_7) have been obtained by the streaming accumulation above, the corresponding even-order syndromes are generated by dedicated squaring circuits. Field squaring in GF(2^m) is a linear permutation of the bit positions, so these squarers can be implemented with only wiring and XOR, without general multipliers. This reduces both the area and the critical-path delay of the Syndrome Unit while maintaining full throughput of one 64-bit input word per cycle.

2.4 Key Equation Solver Unit

The key equation solver implements the inversionless Berlekamp–Massey (iBM) algorithm proposed by Chen *et al.* [1]. Starting from Dornstetter's BM variant, their work derives an inversionless formulation whose update equations can be mapped to a very regular systolic array. The algorithm maintains two polynomials $\Phi^{(r)}(z)$ and $\Psi^{(r)}(z)$ and an integer counter $\ell^{(r)}$. At iteration r, the i-th coefficient pair $(\Phi_i^{(r)}, \Psi_i^{(r)})$ is updated using the discrepancy coefficients $\Phi_0^{(r)}$ and $\Psi_0^{(r)}$ and a global control bit $MC^{(r)}$:

$$\Phi_i^{(r+1)} = \Psi_0^{(r)}\, \Phi_{i+1}^{(r)} - \Phi_0^{(r)}\, \Psi_{i+1}^{(r)}, \tag{66}$$

$$\Psi_i^{(r+1)} = \begin{cases} \Phi_i^{(r)}, & \text{if } MC^{(r)} = 1, \\ \Psi_i^{(r)}, & \text{if } MC^{(r)} = 0, \end{cases} \tag{67}$$

where subtraction is identical to addition in $GF(2^m)$. The counter $\ell^{(r)}$ is updated only from $\Phi_0^{(r)}$ and $MC^{(r)}$; after a fixed number of iterations ($2t$ in the original algorithm), the tail of $\Phi^{(r)}(z)$ contains both the error-locator polynomial $\sigma(z)$ and the auxiliary polynomial needed for Forney's formula. Because the coefficients of Φ and Ψ are always handled in a uniform way, the algorithm naturally maps to a linear array of identical processing elements (PEs).

In the reference architecture of Figure 3, each PE stores one coefficient pair (Φ_i, Ψ_i) and realizes the iBM update with two finite-field multipliers and one finite-field adder (XOR). An array of $(3t{+}1)$ PEs therefore contains $(3t{+}1)$ *adders* and $2(3t{+}1)$ *multipliers*. The control block is shared by all PEs; it computes $MC^{(r)}$, updates $\ell^{(r)}$, and does not contribute to the critical arithmetic path.

Unified datapath and polynomial outputs. In our design all coefficients are represented as 10-bit elements so that the same datapath can handle $GF(2^6)$, $GF(2^8)$, and $GF(2^{10})$. The key equation solver always outputs the constant term together with higher-order coefficients; we do *not* normalize to $\sigma_0 = 1$. To keep a uniform interface to the Chien search, the primary locator polynomial is exported as five 10-bit coefficients $\{\sigma_0, \ldots, \sigma_4\}$ even for the $t = 2$ codes, where the degree is at most 2 and the higher coefficients are zero. In soft-decision mode for $(63, 51)$ and $(255, 239)$, a second locator polynomial is produced in parallel with only three useful coefficients, so the combined output width is "$5 \times 10 + 3 \times 10$ bits" as shown in the overall architecture figure.

Shared systolic array across BCH codes. The RTL module `ibm` instantiates 14 identical PEs (`ibm_pe`) and up to two control units (`ibm_control`). By changing how these PEs are enabled and interconnected,

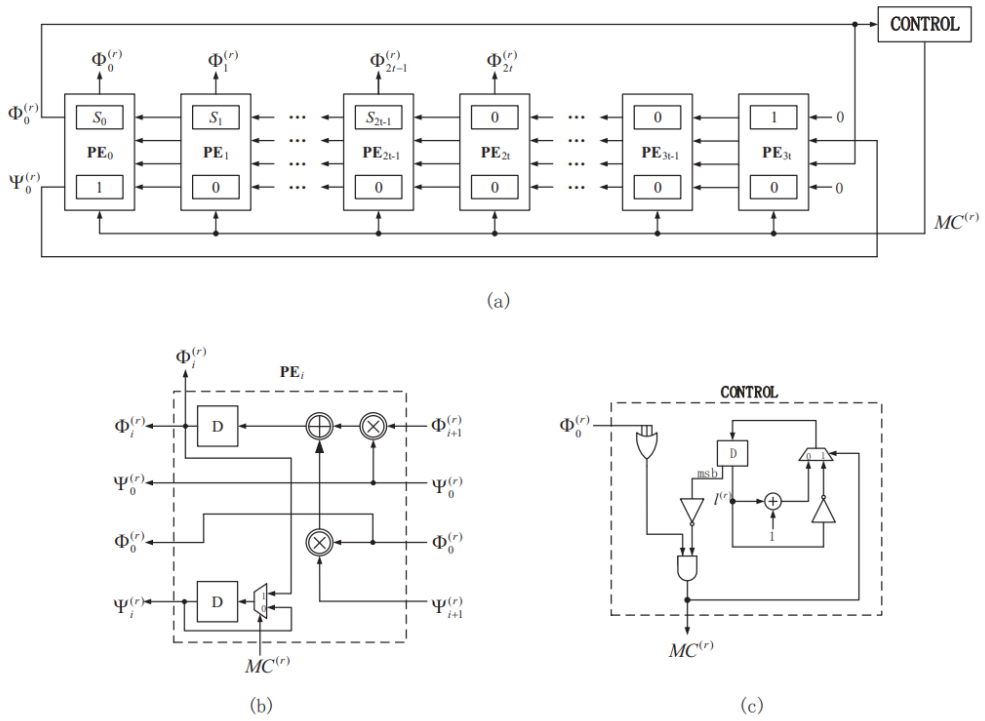


Figure 3: Inversionless Berlekamp–Massey systolic architecture from [1]. (a) Overall array of $3t{+}1$ PEs. (b) Processing element (PE) that updates the coefficients of $\Phi^{(r)}(z)$ and $\Psi^{(r)}(z)$. (c) Control unit that generates the control bit $MC^{(r)}$ and updates the integer counter $\ell^{(r)}$.

the same hardware supports all three BCH codes and both hard- and soft-decision modes:

- $(63, 51)$ / $(255, 239)$, *hard-decision decoding ($t = 2$).* Only PE0–PE6 form an active iBM lane. This lane behaves as a standard $(3t{+}1) = 7$-stage systolic array, controlled by `control1`. The array uses **7 adders** and **14 multipliers**. The internal iteration counter runs for 3 active cycles (counter values 0, 1, 2); when the counter reaches 2, the solver asserts `o_valid` and the 5-coefficient primary polynomial is ready. PE7–PE13 and the second control unit are clock-gated in this mode.

- $(63, 51)$ / $(255, 239)$, *soft-decision decoding ($t = 2$).* In soft-decision mode, two syndrome patterns (original and flipped) must be decoded. We reuse the same 14 PEs by enabling a second lane: PE0–PE6 form "Lane 1" and PE7–PE13 form "Lane 2". Each lane has its own control unit (`control1` and `control2`) and behaves as an independent 7-stage iBM array. The two lanes process their respective syndromes in parallel and both finish after 3 cycles (counter reaching 2). Effectively, we obtain two locator polynomials in **the same latency as a single hard-decision decode**. The arithmetic resources that are actually active in this mode are **14 adders** and **28 multipliers** (two lanes, each with 7 PEs). The outputs are grouped as one 5-coefficient primary polynomial (from Lane 1) and one 3-coefficient auxiliary polynomial (from Lane 2).

- $(1023, 983)$, *hard- and soft-decision decoding ($t = 4$).* For the longer code the error-correction capability is $t = 4$, so an iBM array of size $(3t{+}1) = 13$ is required. Instead of instantiating a third lane, we *concatenate* the two 7-stage lanes into a single chain. In this configuration PE0–PE12

act as a 13-stage iBM array driven by `control1`, while PE13 is disabled. The same 14 PEs are therefore reused as a $3t{+}1$-stage solver for $t = 4$. The internal counter runs for 7 active cycles (counter values 0–6); when it reaches 6, `o_valid` is asserted and a single 5-coefficient locator polynomial (degree ≤ 4) is produced. The effective resources in this mode are **13 adders** and **26 multipliers**. Hard- and soft-decision decoding share exactly the same solver; the difference lies only in how many syndrome patterns are fed to the iBM unit by the pattern chooser.

In summary, the proposed key equation solver keeps a single bank of 14 systolic PEs and reconfigurable control, and exploits the structure of the iBM algorithm to serve all codes and modes:

- For $t = 2$ codes, it delivers *two* locator polynomials in 3 cycles when soft-decision decoding is enabled, or a single polynomial in 3 cycles for hard-decision decoding.
- For the $t = 4$ code, the same hardware is reconfigured as a 13-stage array and delivers one degree-≤ 4 polynomial in 7 cycles.

This sharing strategy significantly reduces area compared with instantiating separate key-equation solvers for each code and each operating mode, while keeping a fixed and short decoding latency.

2.5 Chien Search Unit

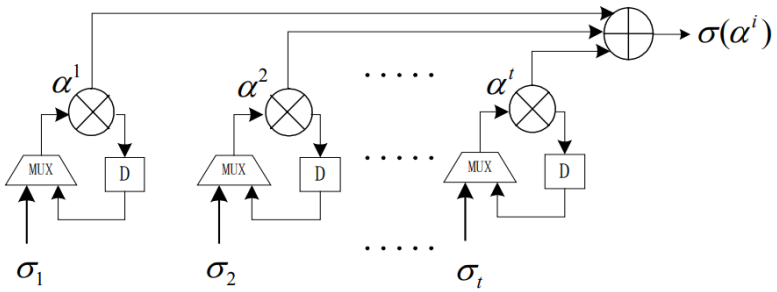


Figure 4: Parallel Chien search datapath with $P = 128$ evaluations per cycle and tree-based index extraction.

The Chien search evaluates the error-locator polynomial(s) over a full codeword and reports the error positions (roots). A locator polynomial is written as

$$\sigma(x) \;=\; 1 + \sigma_1 x + \sigma_2 x^2 + \cdots + \sigma_v x^v, \quad v \leq 4, \tag{68}$$

where v is the degree and $t \leq 4$ is the decoder's error-correction capability. We evaluate $\sigma(\cdot)$ at points $x_{k,p} = \alpha^{-(kP+p)}$ grouped into windows of P consecutive positions (parallel lanes). For window $k \in \{0, \ldots, \lceil n/P \rceil - 1\}$ and lane $p \in \{0, \ldots, P-1\}$,

$$y_p^{(k)} \;\triangleq\; \sigma\big(x_{k,p}\big) \;=\; \sigma\big(\alpha^{-(kP+p)}\big), \tag{69}$$

and a root is detected iff

$$y_p^{(k)} = 0 \iff \text{error at bit position } (kP + p). \tag{70}$$

Because $t \leq 4$, each P-bit vector $y^{(k)}$ contains at most four ones.

Parallel evaluation (fixed hardware parameters). We fix the parallelism to

$$P \;=\; 128, \tag{71}$$

so one Chien window covers 128 bit positions per cycle. The number of evaluation windows for a code of length n is

$$N_{\text{win}}(n, P) \;=\; \left\lceil \frac{n}{P} \right\rceil. \tag{72}$$

For the three target codes:

$$N_{\text{win}}(63, 128) = 1, \tag{73}$$

$$N_{\text{win}}(255, 128) = 2, \tag{74}$$

$$N_{\text{win}}(1023, 128) = 8. \tag{75}$$

In hard-decision mode, we evaluate a single pattern:

$$C_{\text{hard}}(n) \;=\; N_{\text{win}}(n, 128). \tag{76}$$

In soft-decision mode, we evaluate four candidate patterns (original + three flips):

$$C_{\text{soft}}(n) \;=\; 4 \cdot N_{\text{win}}(n, 128). \tag{77}$$

Index extraction via a merge tree. To avoid scanning $y^{(k)}$ bit-by-bit, we partition it into local groups of

$$B \;=\; 8 \quad \Rightarrow \quad G \;=\; \frac{P}{B} \;=\; 16 \text{ groups}, \tag{78}$$

and combine their results through a balanced binary tree of depth

$$D \;=\; \log_2\!\left(\frac{P}{B}\right) \;=\; \log_2(16) \;=\; 4. \tag{79}$$

Each local group produces: (i) a local count $c \in \{0, 1, 2, 3, 4\}$ clipped at $t = 4$, (ii) up to t local indices of ones, and (iii) a validity flag:

$$\text{Local outputs: } \big(c,\ \mathcal{I} = \{i_1, \ldots, i_{\leq t}\},\ v\big). \tag{80}$$

A merge node with upper child $\mathcal{A}$ and lower child $\mathcal{B}$ computes

$$c_{\text{out}} = \min\!\big(t,\ c_{\mathcal{A}} + c_{\mathcal{B}}\big), \tag{81}$$

$$\mathcal{I}_{\text{out}} = \text{first } t \text{ indices from } \Big(\mathcal{I}_{\mathcal{A}} \;\cup\; \{i + \Delta \mid i \in \mathcal{I}_{\mathcal{B}}\} \Big), \tag{82}$$

where Δ is the bit offset of the lower child. After $D = 4$ levels, the root yields

$$\{\, i_1^{(k)},\, i_2^{(k)},\, i_3^{(k)},\, i_4^{(k)} \,\} \quad \text{and the total root count in window } k, \tag{83}$$

with $0 \le i_1^{(k)} < \cdots < i_4^{(k)} \le P-1$.

System scheduling and cycle accounting. Besides Chien evaluation, the rest of the decoder pipeline contributes a fixed latency of

$$C_{\text{other}} \;=\; 34 \text{ cycles per frame,} \tag{84}$$

measured after removing input buffering. With $P = 128$ and ignoring early-stop, BCH(1023,983) requires 8 cycles per pattern (Eq. (75)), thus

$$C_{\text{Chien, 1023}}^{\text{soft}} \;=\; 4 \times 8 \;=\; 32 \text{ cycles.} \tag{85}$$

For BCH(255,239) and BCH(63,51), Chien requires 2 and 1 cycles per pattern, respectively.

Evaluation score and why soft-decision dominates. We adopt the grading metric

$$\text{Score} \;=\; A \cdot \big(C_{\text{hard}}(n) + C_{\text{soft}}(n)\big) \cdot \frac{T_{\text{clk}}}{n}, \tag{86}$$

where: A is chip area, T_{clk} is the clock period, and $n \in \{63, 255, 1023\}$. Because soft-decision evaluates four patterns (Eq. (77)), the term $C_{\text{soft}}(n)$ is typically much larger than $C_{\text{hard}}(n)$,[1] therefore **soft-decision dominates the score**.

Parallelism choice via a simple model. Let x denote the Chien parallelism (number of lanes), with $1 \le x \le 128$ and $x = 128$ used in hardware. We employ a linear area model that separates non-Chien logic and Chien logic:

$$A(x) \;\approx\; \underbrace{50}_{\text{non-Chien}} \;+\; \underbrace{50 \cdot \frac{x}{128}}_{\text{Chien (linear in } x)}, \tag{87}$$

and a cycle model for BCH(1023,983) soft-decision:

$$C_{\text{tot}}(x) \;\approx\; \underbrace{34}_{C_{\text{other}}} + \underbrace{\frac{4 \cdot 1023}{x}}_{\text{four patterns over } n=1023}. \tag{88}$$

Combining,

$$\text{Score}(x) \;\propto\; \Big(50 + 50 \cdot \frac{x}{128}\Big) \cdot \Big(34 + \frac{4092}{x}\Big), \tag{89}$$

which attains a minimum near $x \approx 124$. Choosing $x = P = 128$ achieves essentially the same score while aligning with memory/LLR organization and simplifying control.

Definitions and origin of constants.

- $n \in \{63, 255, 1023\}$: code length for BCH(63,51), BCH(255,239), BCH(1023,983).
- $t \le 4$: maximum number of correctable errors supported by the design (sets list sizes and clip levels).

[1] For BCH(1023,983) at $P = 128$, $C_{\text{soft}} = 32$ vs. $C_{\text{hard}} = 8$.

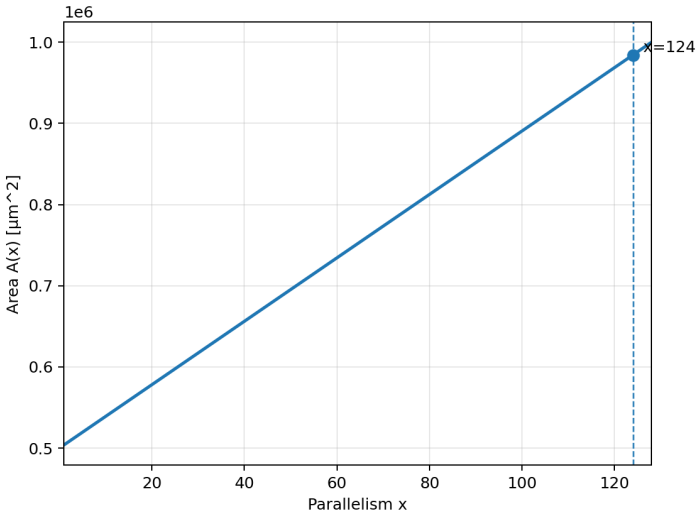


Figure 5: Area model $A(x) = 50 + 50 \cdot (x/128)$ for $x \in [1, 128]$. The score optimum occurs near $x = 124$.

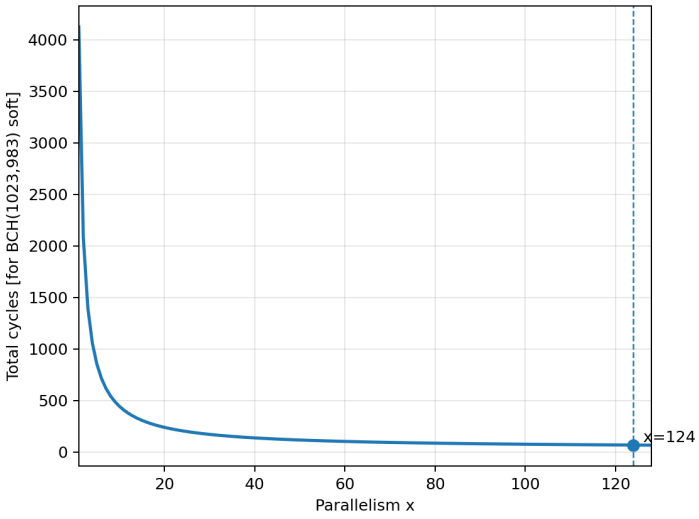


Figure 6: Cycle model $C_{\text{tot}}(x) = 34 + 4092/x$ for $x \in [1, 128]$ (BCH(1023,983), soft decision).

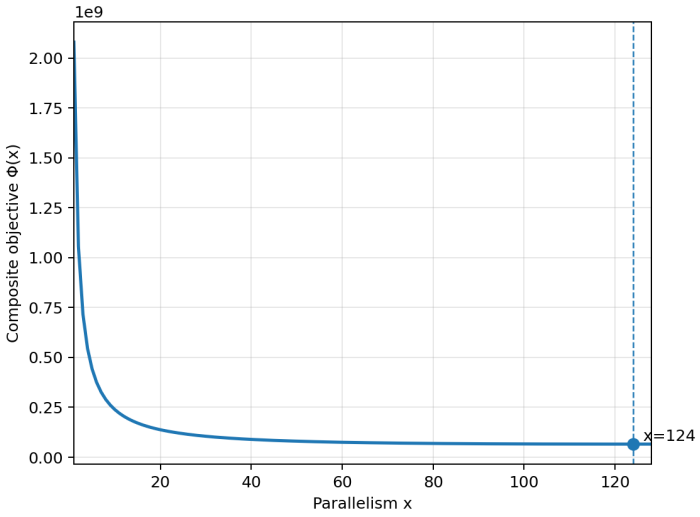


Figure 7: Score proxy $\text{Score}(x) \propto (50+50\cdot x/128)\cdot(34+4092/x)$ for $x \in [1, 128]$, annotated minimum near $x = 124$.

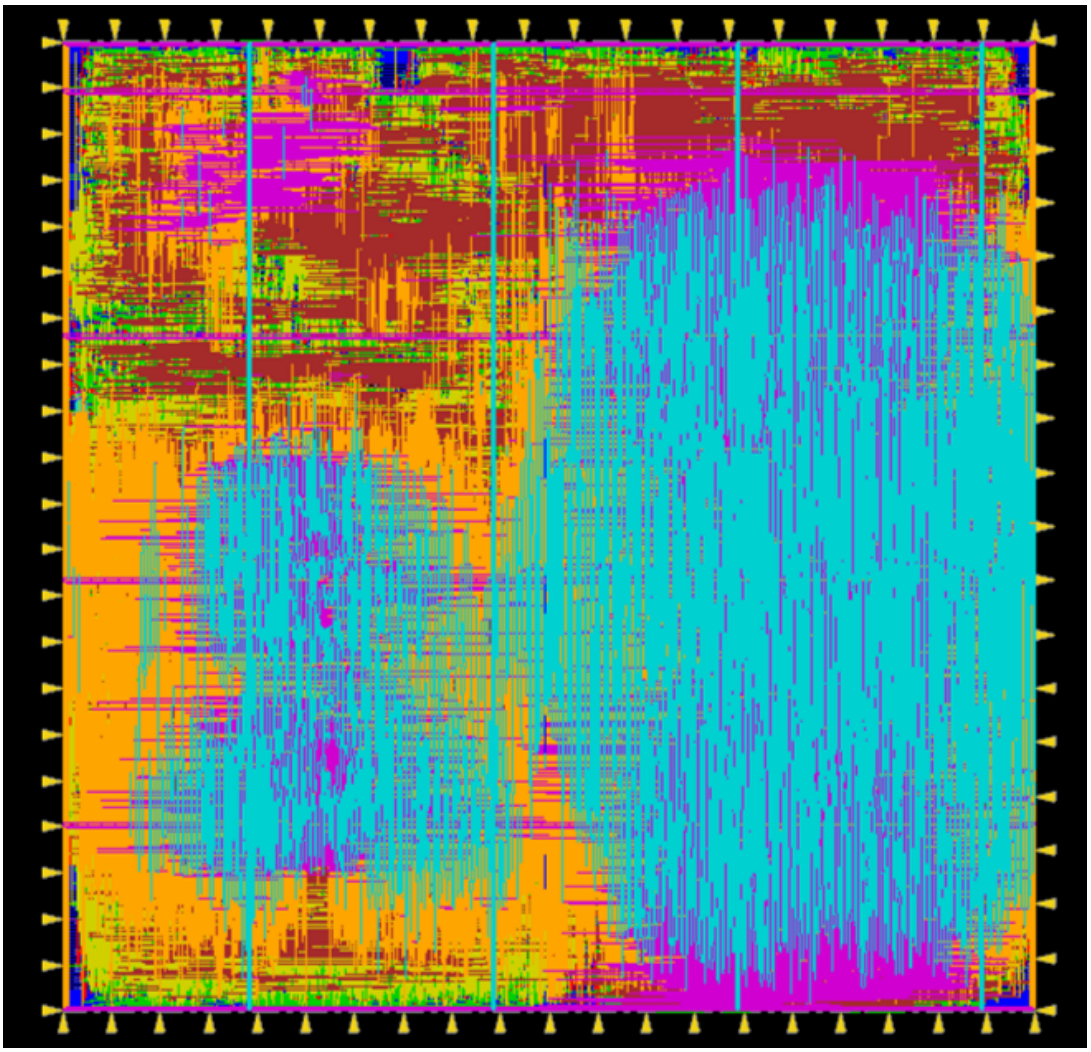
- $P = 128$ (Eq. (71)): degree of Chien parallelism; chosen to align with 64-LLR I/O granularity used by the grading setup (hard/soft both fed by 64 entries) and to minimize the score under Eq. (89).
- $B = 8$ (Eq. (78)): local group size in the index-extraction tree; balances area/timing.
- $G = P/B = 16$, $D = \log_2(16) = 4$ (Eqs. (78)–(79)): number of groups and tree depth.
- $C_{\text{other}} = 34$ cycles (Eq. (84)): measured fixed latency of non-Chien stages per frame after excluding input buffering (syndrome, key-equation solver, buffering, and selector control).
- The factor 4 in Eq. (88): four candidate patterns in soft-decision (original + three flips).
- The constant 1023 in Eq. (88): code length of BCH(1023,983).
- Area model constants in Eq. (87):
  - The term 50 ("non-Chien") represents approximately 500,000 μm^2 of fixed logic, normalized by 10,000 μm^2 per unit.
  - The term $50\cdot(x/128)$ ("Chien") assumes Chien area scales roughly linearly with parallelism x, reaching about 500,000 μm^2 at $x = 128$ under the same normalization.
- T_{clk}: clock period; treated as constant for score comparison across x.

3 APR Results

In this section, we show the physical design results of the proposed BCH decoder. The gate-level netlist is generated by Synopsys Design Compiler, and the layout is implemented using a standard-cell based APR flow in Cadence Innovus.

3.1 Layout Photo

The layout photo of the proposed BCH decoder after APR is shown below. The power ring, power stripes, and signal routing are all contained within the core region, and the final layout passes DRC, LVS, and antenna checks without violations.



3.2 Physical Design Summary

The proposed BCH decoder is implemented using a TSMC 130 nm 8-metal CMOS technology and a standard-cell based APR flow in Cadence Innovus.

The floorplan is generated with a target core utilization of about 0.85. We leave a core-to-boundary spacing of 6 μm on all four sides to reserve area for the power ring and to relax routing congestion near the core boundary.

A core power ring is then built around the standard-cell region. The vertical segments of the ring are routed on Metal 6 and the horizontal segments on Metal 7, so that the power ring can benefit from the wider upper metal layers and provide a low-resistance path for the power delivery network.

Inside the core, we construct a global power stripe mesh. Vertical power stripes are routed on Metal 8, while horizontal stripes are routed on Metal 7. In total, about four groups of power stripes are inserted across the core. Using a relatively small number of stripe groups reduces routing blockages while still

providing sufficient current delivery, so the signal router can find shorter, less congested paths and timing closure becomes easier.

We also refine the I/O placement. In the default setting, the `ioc` block is placed near the upper-left corner just below the top boundary, and the clock pin inside the `ioc` has to be routed through a highly congested region. This leads to dense routing and CTS congestion in that area. We modify the I/O constraints so that the `ioc` is moved to the middle of the left edge of the core. With the clock pin closer to the center of the design, the clock-tree routing becomes more balanced and the overall wire distribution for CTS and signal routing is improved, which helps prevent timing and DRC violations around the I/O region.

3.3 APR Optimization

During the APR stage, we further tuned several Innovus settings to improve runtime and timing convergence.

First, we enable multi-threading by setting `setMultiCpuUsage -localCpu 24`. This significantly reduces the runtime of placement, CTS, and routing, which allows us to iterate more quickly on different APR settings.

For the power delivery network, we keep the power ring and stripe layers as described in Section 3.2, but intentionally limit the number of stripe groups to about four. We observe that using fewer stripe groups leaves more routing resources for signal nets, which alleviates congestion while still keeping IR-drop within an acceptable range.

In the post-route ECO stage, the design often stalled with slightly negative worst negative slack (WNS) and total negative slack (TNS). To improve convergence, we change the ECO optimization mode from the default setting that optimizes only WNS to a mode that simultaneously optimizes both WNS and TNS. When all electrical constraints (maximum capacitance, maximum transition, and maximum fanout) are already clean but WNS and TNS violations still exist, we enable incremental optimization. Incremental ECO only modifies the local nets around the remaining setup/hold timing violations, which speeds up convergence and preserves the quality of already optimized regions. This strategy can be applied at every ECO stage and is effective for closing both setup and hold timing.

3.4 Performance Metrics

We summarize the post-APR metrics of the proposed BCH decoder in the table below. The simulation times are measured in nanoseconds (ns), and the power results are reported in watts (W).

The results for patterns 100, 200, and 300 are obtained from the **public test patterns** provided by the contest. In our setup, these three public patterns correspond to hard-decision decoding of 64-, 255-, and 1023-word BCH codes, respectively. In addition, we generate our own test patterns for both hard-decision and soft-decision decoding at the same three code lengths.

Table 1: Metrics of the proposed BCH decoder after APR.

Metric	Value
Clock Period (ns)	5.7
Core Utilization	0.85
Synthesis Area (μm^2)	985133.614543
Core Area (μm^2)	1085616.30
Die Area (μm^2)	1112009.74
Public test patterns (hard-decision)	
Pattern 100 (hard, 64-word) Time (ns)	390
Pattern 100 (hard, 64-word) Power (W)	0.0112
Pattern 200 (hard, 255-word) Time (ns)	675
Pattern 200 (hard, 255-word) Power (W)	0.0136
Pattern 300 (hard, 1023-word) Time (ns)	1883
Pattern 300 (hard, 1023-word) Power (W)	0.0612
Self-generated patterns (hard-decision)	
Self hard 64-word Time (ns)	287
Self hard 64-word Power (W)	0.0104
Self hard 255-word Time (ns)	675
Self hard 255-word Power (W)	0.0140
Self hard 1023-word Time (ns)	1769
Self hard 1023-word Power (W)	0.0434
Self-generated patterns (soft-decision)	
Self soft 64-word Time (ns)	481
Self soft 64-word Power (W)	0.0185
Self soft 255-word Time (ns)	766
Self soft 255-word Power (W)	0.0318
Self soft 1023-word Time (ns)	2180
Self soft 1023-word Power (W)	0.0918

Note: For each code length, we simulate **only two** self-generated test sequences in both hard-decision and soft-decision modes. The results are mainly used to verify functionality under different data patterns; they should not be interpreted as statistically averaged power numbers. All of these test sequences pass without functional errors under the 5.7 ns clock period.

In addition to the cases listed in Table 1, we also performed more extensive simulations using our self-generated patterns. For both hard-decision and soft-decision decoding modes, we applied multiple

test sequences at each code length, including 64-, 100-, 1000-, and 10000-pattern cases. All of these self-generated test cases passed without functional errors under the 5.7 ns clock period.

3.5 Post-APR Performance Summary

To study the impact of core utilization on area and performance, we also ran APR with a lower target utilization of 0.70. The comparison between the two settings is summarized in Table 2. For each utilization, we report the die/core area and the simulation time and power for the public test patterns 100, 200, and 300.

Table 2: Comparison of different core utilizations (public patterns 100, 200, 300).

Metric	**Util = 0.85**	**Util = 0.70**
Die Area (μm^2)	1112009.74	1852360.74
Core Area (μm^2)	1085616.30	1441474.52
Pattern 100 Time (ns)	390	411
Pattern 100 Power (W)	0.0112	0.0115
Pattern 200 Time (ns)	675	711
Pattern 200 Power (W)	0.0136	0.0140
Pattern 300 Time (ns)	1883	1983
Pattern 300 Power (W)	0.0612	0.0608

From Table 2, the 0.70-utilization case leads to a much larger die area (about 1.5$\times$ compared with 0.85), because more white space is left inside the core. For the public test patterns 100, 200, and 300, the simulation time under 0.70 utilization is slightly longer, while the power is only slightly lower, than those under 0.85 utilization. Considering both speed and area, the 0.85 utilization provides a better overall trade-off and is chosen as our final setting.

We also experimented with a target utilization of 0.90. However, in this case the design can pass the public test patterns only when the clock period is relaxed to about 6.1 ns, which does not meet the original timing requirement of 5.7 ns. Therefore, the 0.90-utilization results are not included in the final comparison.

In addition to the public patterns, we generated our own test patterns, including 64-word hard-decision and soft-decision cases, and longer sequences with 100, 1000, and 10000 patterns. These self-generated patterns further stress the decoder and show consistent trends: the power scales roughly with the amount of switching activity, while the timing remains stable under the chosen 0.85-utilization setting.

References

[1] C. Chen, Y. S. Han, Z. Wang, and B. Bai, “A new inversionless berlekamp-massey algorithm with efficient architecture,” in *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*, pp. 48–53, 2019.