CVSD Final Project: BCH Decoder

Team 020
r14943062 邱郁喆, r14k41019 郭格均

Abstract

This report details our group's design and hardware implementation of the CVSD final project: a BCH decoder supporting three code lengths—(63, 51), (255, 239), and (1023, 983). The decoder implements hard-decision decoding via the Berlekamp Algorithm and soft-decision decoding via the Chase Algorithm. In this project, we address several practical hardware design challenges—including GF arithmetic optimization, syndrome evaluation, error-location computation, etc., to achieve a decoder that balances accuracy, performance, and silicon area.

1 Algorithm Design and Analysis

In this project, we designed a BCH decoder supporting both hard-decision and soft-decision decoding across three code lengths. The algorithm design was strongly influenced by our choice of field representation and our decision to avoid lookup tables (LUTs), which impacted how we implemented GF arithmetic, the Berlekamp-Massey Algorithm (BMA), and the overall decoding flow.

1.1 Field Representation Without Lookup Tables

A common hardware-friendly approach for BCH decoders is to precompute a lookup table mapping each exponent α^i to its vector (tuple) representation, enabling constant-time multiplication, division, and exponent stepping during Chien search. However, supporting three different BCH fields ($m = 6$, 8, and 10) would require three large LUTs, significantly increasing area.

To minimize memory footprint, we chose to store only the *tuple representation* of each GF element and performed all arithmetic through combinational logic. This means that for every GF operation, especially multiplication, we must compute

$$(a \cdot b) \bmod p(x)$$

directly using bit-level polynomial operations rather than via exponent arithmetic.

1.2 Parallel Polynomial-Based GF Multiplication

Since no LUT is available, exponent-domain multiplication (i.e., addition of exponents) cannot be used. Instead, we implement a fully combinational *parallel GF multiplier* for each field size. The multiplier performs:

1. Bitwise polynomial multiplication in GF(2),
2. Modular reduction using the corresponding primitive polynomial.

This architecture produces a one-cycle GF multiplication, which is crucial because:

- Syndrome computation requires multiplying by α^k every cycle,
- BMA requires multiple multiplications per iteration,
- Chien search requires repeated multiplication by α^{-1}.

Parallel GF multiplication is therefore a key enabler of our LUT-free architecture.

1.3 Syndrome Computation

During the input phase, the decoder receives eight LLR values per cycle. Syndrome computation is performed *on-the-fly* while streaming in these inputs, so no additional latency is introduced after the last input sample arrives. For each received bit r_i, the syndromes are updated using:

$$S_i = r(\alpha^i) = r_0 + r_1\alpha^i + r_2\alpha^{2i} + ... + r_{n-1}\alpha^{(n-1)i}$$

Because our architecture processes 8 bits per cycle, each cycle performs eight consecutive syndrome updates. The GF multipliers required for the α^k scaling are reused across iterations, and no lookup tables are used; instead, all updates are computed with parallel polynomial-based multipliers. For $t = 2$ cases (i.e., $(63, 51)$, $(255, 239)$), the decoder must compute syndromes S_1 through S_4. These four syndromes are fully accumulated by the time all input bits have been streamed into the decoder. For $t = 4$ cases (i.e., $(1023, 983)$), the same architecture naturally extends to compute syndromes S_1 through S_8 within the same input window.

As a result, syndrome computation overlaps entirely with the input loading phase. Once the final 8-bit chunk of the received word has been processed, all required syndromes are immediately available for the following BMA stage, ensuring a fully pipelined and low-latency decoding flow.

1.4 Hard-Decision Decoding via Inversionless Berlekamp–Massey Algorithm

A major consequence of removing LUTs is that *GF division becomes expensive.* The classical Berlekamp–Massey algorithm updates the connection polynomial using

$$\sigma^{(\mu+1)} = \sigma^{(\mu)} - \frac{d_\mu}{d_\rho} X^{\mu-\rho}\sigma^{(\rho)},$$

which requires computing the multiplicative inverse of d_L in GF(2^m). Computing GF inverses without lookup tables is costly and slow, since inversion requires running the extended Euclidean algorithm or exponentiating to $2^m - 2$, both of which are unsuitable for a pipelined hardware decoder.

Therefore, to eliminate all divisions, we adopt an inversionless variant of the Berlekamp–Massey Algorithm (BMA), derived from the serial architecture proposed in [1] by Hsie-Chia Chang and C. B. Shung. This architecture removes costly GF inversions and restructures the recurrence into a fully multiplicative form suitable for efficient hardware mapping. The inversionless Berlekamp-Massey algorithm is as followed:

Initial condition :
$D^{(-1)} = 0, \quad \delta = 1$
$\sigma^{(-1)}(x) = \tau^{(-1)}(x) = 1$
$\Delta^{(0)} = S_1$
for $i = 0$ **to** $2t - 1$
$\begin{cases} \sigma^{(i)}(x) = \delta \cdot \sigma^{(i-1)}(x) + \Delta^{(i)} x\tau^{(i-1)}(x) \\ \Delta^{(i+1)} = S_{i+2}\sigma_0^{(i)} + S_{i+1}\sigma_1^{(i)} + \cdots + S_{i-\nu_i+2}\sigma_{\nu_i}^{(i)} \end{cases}$
If $\Delta^{(i)} = 0$ **or** $2D^{(i-1)} \geq i + 1$
$D^{(i)} = D^{(i-1)}, \quad \tau^{(i)}(x) = x\tau^{(i-1)}(x);$
else
$D^{(i)} = i + 1 - D^{(i-1)}, \quad \delta = \Delta^{(i)}$
$\tau^{(i)}(x) = \sigma^{(i-1)}(x)$

Figure 1: Inversionless Berlekamp-Massey Algorithm

where all operations use only multipliers and XORs, no inverses are needed. We implemented the algorithm for getting the error-location polynomial in this project.

1.5 Chien Search

The Chien search stage find the roots of error locator polynomial by substituting α^{-j} into $\sigma(X)$ at all field elements corresponding to codeword positions. In the classical formulation, the decoder tests

$$x = \alpha^{-1},\ \alpha^{-2},\ \alpha^{-3}, \ldots,$$

because bit position i corresponds to evaluating $\sigma(\alpha^{-i})$. This requires repeatedly multiplying by α^{-1}, which is trivial if exponent-domain lookup tables are available.

However, in our LUT-free architecture, field elements are stored only in tuple (polynomial) representation, and GF division or exponent negation is not supported. As a result, computing α^{-1} or performing repeated inverse-based updates is not practical.

To overcome this limitation, we reformulate the search direction. Since

$$\alpha^{j} = \alpha^{-(n-j)}$$

in a length-n BCH code, we instead start the search at

$$x_0 = \alpha^0, x_1 = \alpha^1 = \alpha^{-62}$$

The remaining field elements are generated using only forward multiplication:

$$x_{i+1} = x_i \cdot \alpha^1.$$

Thus, the search sequence becomes:

$$\alpha^0,\ \alpha^1,\ \alpha^2,\ \ldots,\ \alpha^{n-1},$$

which is mathematically identical to evaluating $\sigma(\alpha^0), \sigma(\alpha^{-(n-1)}), \sigma(\alpha^{-(n-2)}), \ldots, \sigma(\alpha^{-1})$ but is fully compatible with a multiplier-only GF implementation. By scanning the roots in this order, we eliminate the need for inversion while maintaining full decoding correctness and preserving low area cost.

1.6 Soft-Decision Decoding via Chase Algorithm

Chase algorithm is a soft-decision decoding algorithm, which tests multiple r(X) candidates to find the most probable result. Below shows the overall flow of how we implement it in our design.

Overall Flow

Soft decoding in our design works as follows:

1. During the input phase, we receive 8-bit LLRs and store them into an internal LLR memory while simultaneously computing hard-decision bits.

2. In the same streaming process, we track the p = 2 least reliable positions (e.g., the smallest-magnitude LLRs) using a small comparator network and a few registers to record their indices.

3. After the full codeword is loaded, we generate a small, fixed set of test patterns by selectively flipping these least-reliable bits on top of the hard-decision vector.

4. For each pattern, we run the same hard-decision BCH pipeline (syndrome computation, inversionless BMA, Chien search), and then evaluate a correlation metric between the candidate codeword and the stored LLRs.

5. We keep track of the best candidate in terms of correlation and output its decoded bits and error locations as the final result.

2 Hardware Implementation

The BCH decoder is implemented as a fully parameterized hardware architecture supporting three code lengths $(63, 51)$, $(255, 239)$, and $(1023, 983)$ and their corresponding Galois fields $m = 6, 8, 10$. All major modules share common GF arithmetic units to reduce area, and the decoder follows a streaming pipeline in which syndrome computation, soft-decision processing, BMA, and Chien search are orchestrated by a global controller.

2.1 Overall Architecture

Our design architecture consists of:

- LLR input buffer and least-reliable tracker
- GF($GF(2^M)$) multiplier modules
- 8 syndrome calculator modules
- 4 parallel inversionless BMA modules
- 4 parallel chien search modules
- Soft-decision controller and correlation calculating unit
- A finite-state machine controlling the full decoding sequence

All communication between blocks is performed through registers and simple handshake signals, ensuring predictable timing and making the modules reusable across all supported BCH codes.

2.2 Key Module Implementation Details

The decoder receives 8-bit LLR inputs one per cycle. These are written sequentially into an internal memory implemented as an array of registers. During this same input phase, we also:

2.2.1 Galois Field Multiplier (`gf_mul`)

This combinatorial module is the foundational arithmetic unit, implemented using a **bit-parallel multiplication architecture** based on the chosen irreducible polynomial for the field size 2^m.

- **Parallel Kernels:** The module contains three distinct, dedicated combinatorial circuits for $GF(2^6)$, $GF(2^8)$, and $GF(2^{10})$
- **Irreducible Polynomial Mapping:** The multiplication and subsequent modular reduction for each field size are hardcoded into the connections and XOR logic of the circuit (e.g., the large XOR logic trees in the $GF(2^{10})$ multiplier implementation). The final output is selected via a multiplexer controlled by the input m.

2.2.2 Syndrome Calculator Modules (`S1_calc` to `S8_calc`)

The syndrome bank is a collection of eight parallel, identical Linear Feedback Shift Register (LFSR) circuits.

- **Structure:** Each calculator (`S_calc` module) is an 8-bit or 10-bit shift register where the feedback is determined by the GF multiplication by α^8 and the incoming data r_data.
- **Hardcoded Feedback:** The field multiplication ($\cdot\alpha^8$) for each of the three code rates is implemented as a **deep combinatorial XOR network** within the `always @(*)` block of each module, optimized for the specific irreducible polynomial of that field.

The syndrome calculator is a fully streaming design. When one LLR value and its corresponding hard-decision bit enter the system, the module performs eight consecutive updates using the parallel GF multiplier. All GF multiplications by α^k share the same polynomial-based multiplier; the exponentiation is handled by shift-register–like rotation of precomputed α^k tuples, eliminating the need for lookup tables. Once the input phase ends, all required syndromes $S_1 \dots S_{2t}$ are available immediately, allowing the BMA stage to start without any stall cycles.

2.2.3 Inversionless BMA Module

The BMA module implements the inversionless version of the Berlekamp–Massey algorithm for $t = 2$ and $t = 4$.

- **Parallel Instantiation:** The top-level `bch` module instantiates four separate `BMA` units (`u_bma`, `u_bma_f1`, `u_bma_f2`, `u_bma_f3`) to process the original syndrome vector (S) and the three modified syndrome vectors (S_{f1}, S_{f2}, S_{f3}) concurrently during Soft-Decision decoding.
- **Iteration Count:** The iteration proceeds for $2t$ steps, controlled by the `iteration_cnt` register, which runs from 0 up to $2t - 1$.
- **Critical Path:** Each iteration calculates the next error locator polynomial coefficients (σ_{next}) and the discrepancy (Δ_{next}). This calculation involves multiple Galois Field multiplications (`gf_mul`) and XOR additions, forming the most time-critical combinatorial path in the design.
- **Input Interface:** The module receives the flattened syndrome vector, `syndromes_flat`, which includes S_1 through S_{11} (11 coefficients of 10 bits each).
- **Outputs:** The module outputs the final error locator polynomial coefficients (σ_{out}) and the degree of the polynomial (D), which corresponds to the number of errors found.

2.2.4 Chien Search Module

The `chien_search` module is responsible for finding the roots of the error locator polynomial $\sigma(x)$ generated by the BMA, thus determining the error location numbers.

- **Parallel Instantiation:** Similar to BMA, four parallel `chien_search` units (`u_search` through `u_search_f3`) are instantiated in the top module for concurrent root finding in Soft-Decision mode.
- **Evaluation:** The search iteratively evaluates $\sigma(\alpha^i)$ for $i = 0$ to $n-1$, controlled by `search_cnt`.
- **Root Check:** If the evaluation result (`eval`) is zero, then $n - i$ (where i = search_cnt) corresponds to an error location.
- **Power Calculation:** Each cycle, the required powers of α ($\alpha^i, \alpha^{2i}, \alpha^{3i}, \alpha^{4i}$) are computed using GF multipliers (`gf_mul`) to prepare for the next iteration's evaluation.
- **Error Location Storage:** The module stores up to t error locations in the `err_loc` array.

2.2.5 Soft-Decision Controller and Correlation Calculating Unit

This unit manages the final decision process in Soft-Decision mode, which selects the most probable decoding result from the four possible test patterns (P_1 to P_4).

- **Test Patterns:** The four patterns are: P_1 (original S), P_2 (S_{f1}, with $\min_2$ flipped), P_3 (S_{f2}, with $\min_1$ flipped), and P_4 (S_{f3}, with both $\min_1$ and $\min_2$ flipped).

- **Correlation Calculation:** The `bch` module uses four parallel signed 13-bit registers (`correlation`, `correlation_f1`, `correlation_f2`, `correlation_f3`) to accumulate the correlation score for each pattern.
- **Correlation Update Logic:** The score is updated based on the LLR value of the current symbol:
  - If the bit is an error location (hit) or a minimum location ($\min_1, \min_2$), the signed LLR is subtracted from the score.
  - Otherwise, the signed LLR is added to the score.
- **Decision Logic:** After iterating through all n codeword bits, the unit uses three parallel comparators to find the maximum correlation score (`max_correlation`) and the index of the best pattern (`idx_max_correlation`).
- **FSM Control:** The FSM transitions out of the `S_CORRELATION` state only when all bits have been correlated ($correlation\_cnt == n$).

The soft-decision unit is tightly integrated into the hardware pipeline. After the input phase finishes, the module already knows the positions of the least reliable bits. These indices determine the flipping patterns for soft decoding. Rather than storing multiple full-length candidate vectors, we generate each candidate *on the fly*, running it through the hard-decision BCH pipeline. For each candidate, the correlation score is accumulated serially:

- the LLR memory is shifted or indexed sequentially,
- one signed multiply-accumulate structure collects the correlation value,
- only the final score is stored, minimizing data-path width and register count.

After finishing all candidates, the best one (maximum correlation) is selected and ready for output.

2.2.6 Global Control FSM

The FSM that controls our whole design contains 7 states:

1. S_IDLE,
2. S_SYNDROME_CALC,
3. S_SOFT_WAIT,
4. S_BMA,
5. S_SEARCH,
6. S_CORRELATION,
7. S_OUTPUT

Each stage asserts `done` signals which trigger transitions. Because each module has fixed and predictable latency, the control logic remains simple and timing-robust.

2.3 Hardware Optimization Techniques

The decoder adopts several architectural optimizations that balance performance and hardware efficiency. Rather than relying on pipelining or folding transformations, we focus on three key strategies: controlled parallelism, elimination of lookup tables, and extensive resource sharing.

2.3.1 Parallel Processing for Soft-Decoding Candidates

To accelerate the Chase soft-decision flow, the design instantiates:

- **four parallel BMA modules**, and
- **four parallel Chien search modules**,

allowing all candidate patterns to proceed through the error-location pipeline concurrently. This reduces the soft-decoding latency to nearly that of a single hard-decision decode, providing substantial speed improvement at a moderate area cost.

Parallelism is applied selectively—only to the stages where candidate-level concurrency yields significant throughput benefits.

2.3.2 Elimination of Lookup Tables and GF Division

The architecture avoids all lookup tables, including exponent, logarithm, and inverse tables. All GF elements are stored only in tuple form, and all arithmetic is realized through a polynomial-based parallel GF multiplier.

Eliminating LUTs not only reduces memory overhead but also removes the need for GF division. This choice directly enables the adoption of the inversionless Berlekamp–Massey algorithm, which removes division entirely and provides a uniform, hardware-friendly update structure across $m = 6, 8, 10$ fields.

By combining targeted parallelism with extensive resource sharing, the decoder achieves a compact yet high-throughput implementation supporting both hard- and soft-decision BCH decoding.

3 APR Results

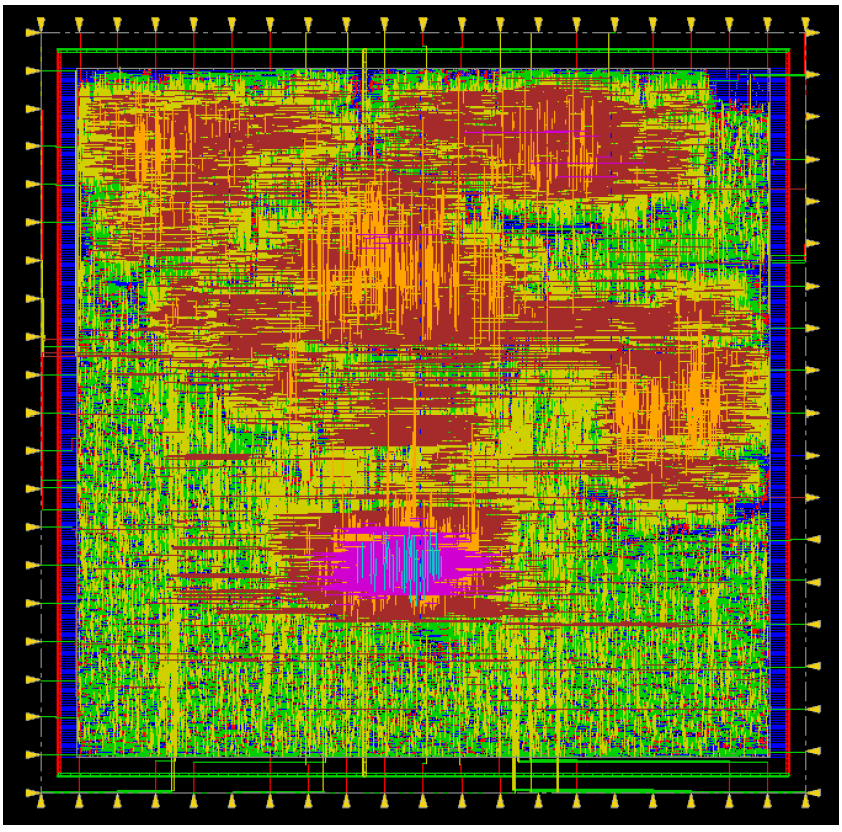

Figure 2: Final Layout of Our Design

Our APR result met all specified DRC/LVS requirements. Figure 2 shows the physical view of our final layout. The core utilization we used for the final APR result is 0.8.

4 Performance Evaluation

4.1 Area & Timing Performance Metric

The final AT performance metric for each version of our experiment is as follows:

Design Version	Pre-APR		Post-APR		Compile Command
	gatesim CT (ns)	Area (um^2)	postsim CT (ns)	Area (um^2)	
v1 (gatesim failed)	6.6	803919.187327	X	X	compile_ultra
v2 (syn with CT 6.6 ns)	7.0	840235.062995	7.0	1200808.656	compile_ultra -noautoungroup
v3 (syn with CT 7.0 ns)	7.0	829112.001309	7.0	1105849.310	compile_ultra -noautoungroup
v4 (optimized RTL code)	7.0	747124.186160	7.0	934332.133	compile_ultra

Table 1: Area & Timing Metric for Pre-APR and Post-APR

Table 1 summarizes the area and timing results of three design versions evaluated during the implementation process.

In version 1 (v1), the design was synthesized using the `compile_ultra` command. Although synthesis completed successfully, the resulting netlist failed to pass gate-level simulation due to excessive simulation time, eventually leading to a timeout. This behavior suggests that `compile_ultra` may generate an **over-optimized netlist** with aggressive logic restructuring, which significantly increases simulation complexity and causes gate-level simulation to become impractical.

To mitigate this issue, versions 2 and 3 (v2 and v3) were synthesized using `compile_ultra -noautoungroup`. By disabling automatic ungrouping, the hierarchical structure is better preserved, and both versions successfully passed gate-level simulation without timeout issues.

In v2, the synthesis constraint file specified a target clock period of 6.6 ns. However, gate-level simulation could only pass when the clock period was relaxed to 7.0 ns, indicating that the design was marginally unable to meet the tighter timing constraint in practice. In contrast, v3 directly specified a clock period of 7.0 ns during synthesis, which reduced timing pressure and resulted in a slightly smaller synthesized area compared to v2.

During the APR stage, different core utilization targets were applied. Version 2 used a core utilization of 0.7, while version 3 increased the utilization to 0.75. Although the synthesized areas of v2 and v3 are similar, the post-layout area of v3 is approximately 100,000 units smaller. This observation indicates that higher core utilization leads to more compact placement and reduced routing overhead, thereby improving area efficiency after physical design.

In version 4 (v4), further improvements were achieved through RTL-level optimization rather than synthesis option tuning. Hardware sharing was enhanced across multiple modules, and the correlation computation in the soft-decision decoder was optimized from processing one index per cycle to processing 32 indices per cycle. These changes significantly reduced control overhead and register usage while improving datapath efficiency.

As a result, v4 shows a substantial reduction in both synthesized and post-layout area compared to previous versions. In addition, the APR stage of v4 adopts an increased core utilization of 0.8, further improving layout compactness without sacrificing timing closure. Unlike v1, v4 is synthesized using `compile_ultra` without simulation issues, demonstrating that RTL-level optimization is more effective and robust than relying solely on aggressive synthesis optimizations.

Overall, v4 achieves the best balance among area efficiency, timing robustness, simulation stability, and physical design quality, and is therefore selected as the final implementation.

4.2 Power & Timing Performance Metric

Below are the PT performance metric for v4 postsim, which is our last design version. Here we use the public test pattern p100, p200, p300 for simulation. Note that the postsim cycle time is 7.0 ns.

Table 2: Power & Timing Metric for Postsim

Test Data	**Time (ns)**	**Power (STA, W)**
p100	1138	0.0274
p200	2608	0.0277
p300	14711	0.0283

5 References

[1] Hsie-Chia Chang and C. B. Shung, "New serial architecture for the Berlekamp-Massey algorithm," in IEEE Transactions on Communications, vol. 47, no. 4, pp. 481-483, April 1999.