GF(2^8) Degree-255 Polynomial — Hardware Summary

Goal: Evaluate the polynomial

$$P(x) = c_{255}x^{255} + c_{254}x^{254} + \cdots + c_1x + c_0$$

in GF(2^8), where:

- Each coefficient `c_k` is given as an **8-bit index** (not raw byte) into the field table (you told me coefficients are *already indices*).
- `x` is provided as an **8-bit index** into the same field table.

This document summarizes the data encodings, BRAM layouts, Horner-in-index-domain algorithm, hardware FSM, timing, and verification steps.

---

1. Assumptions and notation

- You gave four BRAMs / LUTs (we expand them for GF(2^8)):

- `alpha[0..254]` : 8-bit field value `α(k)` for exponent index `k` (maps index → byte).

- `mul[0..254][0..254]` : 8-bit **index** `k` such that `α(k) = α(i)*α(j)` for nonzero operands.
- `xor[0..254][0..254]` : 8-bit **index** `k` such that `α(k) = α(i) xor α(j)` for nonzero operands.

- `inv[0..254]` : 8-bit **index** of multiplicative inverse `α(-i)` (optional but handy).

- **Index encoding (recommended):** 1 byte (8 bits) per index.

- `0 .. 254` = exponent indices for nonzero field elements (`α^0` .. `α^{254}`).

- `255` (`8'hFF`) = special **ZERO** code representing field element `0`.

- Your inputs: `C[0..255]` and `x` are already **indices** in this encoding (so no value→index conversion required). If you ever get raw bytes (α-values), you will need a reverse map `rev[value] -> index`.

---

2. Key arithmetic rules (index domain)

Use index-domain lookup tables to make arithmetic trivial:

- **Addition (field XOR):**
- If either operand is `ZERO (255)`, `v xor 0 = v` (so return the other index).
- If both are `ZERO` → `ZERO`.
- Else (both nonzero) → use `xor[idx_a][idx_b]` to get result index.
- **Multiplication:**
- If either operand is `ZERO` → `ZERO`.
- Else → `mul[idx_a][idx_b]` returns the result index.
- **Inverse / division (if needed):** use `inv[idx]`.

This allows pure table lookups; no bitwise polynomial multiply logic required.

---

3. Algorithm — Horner in the index domain

Horner's rule (works efficiently with LUTs):

```
// inputs: x_idx (8-bit index), coeff_idx[0..255] (each 8-bit index)
ZERO = 8'hFF
acc = coeff_idx[255]   // highest-degree coefficient (index-coded)
for i = 254 downto 0:
    // multiply acc by x
    if (acc == ZERO || x_idx == ZERO) then acc = ZERO
    else acc = mul[ acc ][ x_idx ]

    // add next coefficient
    if (acc == ZERO) acc = coeff_idx[i]
    else if (coeff_idx[i] == ZERO) acc = acc
    else acc = xor[ acc ][ coeff_idx[i] ]
endfor
// now acc is the result index; map to actual byte
result_idx = acc
result_val = (acc == ZERO) ? 8'h00 : alpha[ acc ]
```

Operations per coefficient: 1 table multiply + 1 table add (or some short special-case checks). Total operations for degree-255: 255 multiplies + 255 adds.

---

4. Suggested Verilog module I/O (top level)

- `clk, rst` — standard clock/reset.
- `start` — start evaluation pulse.
- `x_idx [7:0]` — x index (0..254 or 255 for ZERO).
- `C_in [7:0]` interface: many options:
- Option A: Provide coefficients already loaded in an internal ROM/BRAM `coeff[0..255]` (8-bit each).
- Option B: Stream coefficients in on start (less common here since user has them available).
- `busy` / `done` signals.
- `result_idx [7:0]` — index result (0..254 or 255=ZERO).
- `result_val [7:0]` — actual mapped byte `alpha[result_idx]` (or `0x00` for ZERO).

Example ports (compact):

```
input clk, rst, start;
input [7:0] x_idx;
input [7:0] coeffs [0:255]; // ROM/BRAM or input bus
output reg done, busy;
output reg [7:0] result_idx, result_val;
```

---

5. BRAM / memory layout & sizes

- `mul` table: 255×255 entries. If stored as 1 byte per entry → 65025 bytes (\~63.47 KiB).
- `xor` table: same size as `mul` → \~63.47 KiB.
- `alpha` table: 255 × 1 byte ≈ 255 bytes (0.25 KiB).
- `inv` table: 255 bytes.

Total LUT storage \~127 KiB + small tables. This fits comfortably in BRAM on typical mid-sized FPGAs but you should check available BRAM blocks on your target (or split tables across BRAMs).

Implementation note: Use block RAM (BRAM) primitives or `$readmemh` for simulation and synthesize to memory. If BRAM read latency > 1 cycle, adjust FSM accordingly (pipeline the reads or add wait states).

---

6. FSM / timing (one-cycle per Horner iteration option)

A simple FSM that does *one Horner iteration per clock*:

1. **IDLE** — wait `start`.
2. **LOAD** — load `x_idx` into `x_reg`, load `acc = coeff[255]`, set `i = 254`.
3. **ITERATE** — each clock:
4. read `mul[acc][x_reg]` and set `acc_tmp` (if acc or x is ZERO, `acc_tmp = ZERO`).
5. read `coeff[i]`.
6. compute `acc_next` via XOR-LUT (with ZERO checks). This can be combinationally done from the LUT outputs.
7. `acc <= acc_next; i <= i-1`.
8. **DONE** when `i < 0` → map `result_val = (acc==ZERO) ? 0 : alpha[acc]`, assert `done` / deassert `busy`.

Latency / throughput: If each iteration takes 1 cycle and there are 255 iterations, latency ≈ 255 cycles (plus a few overhead cycles). Example: at 100 MHz → \~2.55 µs per evaluation.

If BRAMs have read latency of 2 cycles, the FSM must be adjusted (pipeline/coalesce reads) — typically add 1–2 wait states or design a small pipeline that prefetches LUT outputs.

---

7. Zero / special cases (explicit)

- If `x_idx == 255` (x = 0): polynomial simplifies — every multiplication yields zero, so Horner reduces to `acc` being the highest nonzero constant reached by scanning coefficients: effectively `P(0) = c_0` if you evaluate in standard order; but with Horner starting from highest degree you can short-circuit: multiplying any `acc` by 0 gives ZERO, then `acc` becomes next coefficient, etc. The FSM logic already handles this via the `ZERO` checks.

- If coefficients are indices and one equals 255, treat it as 0.

---

8. Implementation / synthesis tips

- **ROM style:** store `mul` and `xor` in BRAMs as 2-D arrays or flattened 1-D memory with address computed by `addr = i * 255 + j`.
- **BRAM read ports:** you may need two read ports (one for mul, one for xor) or time-multiplex single-port BRAM (but that increases cycles). Use dual-port BRAMs where possible.
- **Packing indices:** using 8 bits per index is convenient. Reserve `8'hFF` as ZERO.
- **Coefficient input:** since you said coefficients are passed as indices, simply load them into a `coeff_rom` before `start`, or provide a separate write interface to update coefficients.
- **Verification:** compare hardware output vs. software implementation using the same LUTs. Generate random `x_idx` and `coeff_idx` and check equality.

---

9. Testbench plan

1. Create or import the `alpha`, `mul`, `xor` tables (same content as the hardware BRAMs).
2. Write a software helper (Python / C / SystemVerilog DPI) that performs the same Horner algorithm using these LUTs and produces expected `result_idx` and `result_val`.
3. In the Verilog testbench: load `coeff_rom`, poke `x_idx`, assert `start`, wait for `done`, compare outputs to the software reference.
4. Randomize coefficients and x, test many vectors, include edge cases with many zeros.

---

10. Checklist before synthesis

- Confirm the **index encoding** (`0..254` + `255` → ZERO) is exactly what your BRAMs expect.
- Confirm BRAM read latency (and adjust FSM/wait states accordingly).
- Verify `mul` / `xor` tables cover only `0..254` indices and that you handle ZERO separately in logic.
- Decide whether you want to preconvert raw coefficient bytes into indices (if you were ever to accept byte values rather than indices).

---

11. Example small walkthrough (degree 3 toy example)

Given indices: `C3=5, C2=10, C1=255 (zero), C0=7`, and `x=3`:

```
acc = C3 = 5
acc = mul[5][3] -> idxA
acc = xor[idxA][C2=10] -> idxB
acc = mul[idxB][3] -> idxC
acc = xor[idxC][C1=255] (C1 is ZERO) -> acc stays idxC
acc = mul[idxC][3] -> idxD
acc = xor[idxD][C0=7] -> result_idx
result_val = alpha[result_idx]
```

This exactly mirrors the full 255-degree flow but with fewer iterations.

---

12. Next steps / Offer

I can:

- Produce the **Verilog HDL** for the sequential Horner evaluator (degree-255) that uses these BRAMs (with realistic BRAM read-latency handling and a `coeff_rom`).
- Produce a **SystemVerilog testbench** + small Python reference model that validates the design.

If you'd like the Verilog and testbench, tell me whether:

1. Your BRAMs are synchronous single-cycle read (combinational) or they have 1+ cycle latency, and
2. You prefer `255` or `0xFF` to denote ZERO (I used `8'hFF` here but can change).

End of summary.

(If your BRAM indexes really are `0..16` like the earlier GF(2^4) example, this same document applies conceptually — only the index range and table sizes change. Tell me which target field you want and I will produce the Verilog + testbench.)