Rensselaer

why not change the world?®

Ultra Low Latency 64B
BCH encoder/decoder

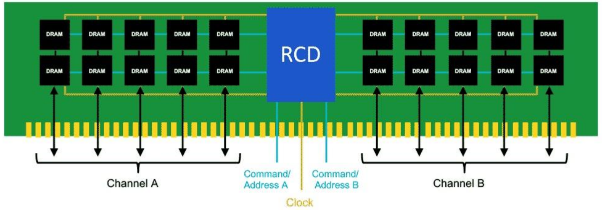
Semester Project: Advanced VLSI Design | 04/18/2024

Introduction: *BCH Codes.*

- A class of **Linear, Cyclic** codes (From the family of **LDPC**, **Hamming**).
- Mathematically, they are also classified among **Polynomial Codes** (akin to CRC).
- Generalization of SEC-DED **Hamming**, can correct/detect **more than one error**.

- Used in **main memory systems**, **storage media** and **communication**.
- Implementation is well-studied from the perspective of **serial communication** => *Usually incur a **high latency cost** in VLSI implementation.*

- **Goal**: **Ultra Low Latency BCH codecs** at **Cacheline Granularities.**



BCH Codes: *The Theory*

- Imagine that binary numbers were **polynomials.**
- **101101 =>** $1x^5+0x^4+1x^3+1x^2+0x^1+1 = m(x)$
- How can the receiver **verify**?

$m_t(x)$ = **101101**

$m_r(x)$ = **100101**

BCH Codes: *The Theory*

- Imagine you have another **number/polynomial** $g(x)$ with roots $\{\alpha, \alpha^2, \alpha^3\}$.
  - Send a coded message $c(x) = m(x) \,.\, g(x) = \mathbf{1}\, x^7 + \mathbf{1}\, x^6 + \mathbf{1}\, x^3 + \mathbf{1}\, x^1 + \mathbf{1}$
  - Receiver evaluates $c(x)$ at the roots of $g(x)$ => IF $g(\alpha) == 0$ THEN $c(\alpha) == 0$.

$c(x) = m(x) \,.\, g(x) =$

11001011

$c(\alpha) == 0\ ?$

$c(\alpha^2) == 0\ ?$

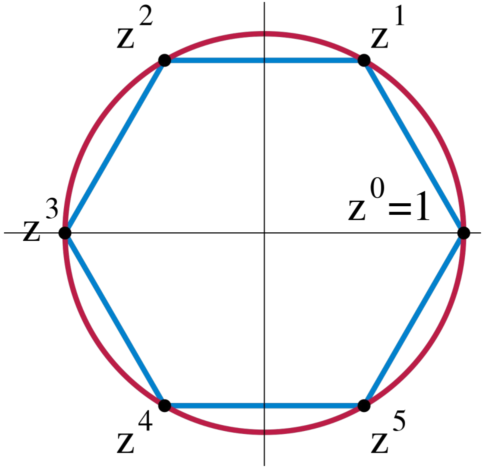
$c(\alpha^3) == 0\ ?$

$\boldsymbol{m(x) = c(x) / g(x)}$

BCH Codes: *A crash course in* ***Galois Fields****.*

- Conventional polynomials are not adaptable to **binary** number system.
  - Where can I find polynomials with **binary coefficients**?
  - **Multiplications** are hard, **divisions** are worse.

- We need a different **polynomial arithmetic**. Enter Galois Fields.



$$\mathrm{GF}(q) = \mathrm{GF}(p)[X]/(P)$$

BCH Codes: *A crash course in* ***Galois Fields***.

- We need a different **polynomial arithmetic**. Enter Galois Fields.
- Where, every number is a polynomial => $6 = 1\,x^2 + 1\,x^1 + 0\,x^0$

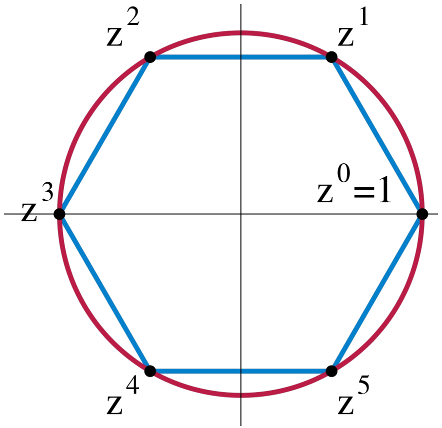
- Addition is over **Modulo 2** for each power, i.e. **XOR:**

  $6 + 5 = (1\,x^2 + 1\,x^1 + 0\,x^1) + (1\,x^2 + 0\,x^1 + 1\,x^1) = 3 =$

  $110_2 \wedge 101_2 = 011_2$

- Multiplication is **Polynomial Product** modulo ***reducing polynomial:***

  $6 \times 5 = (1\,x^2 + 1\,x^1 + 0\,x^1) \times (1\,x^2 + 0\,x^1 + 1\,x^1) = 13$



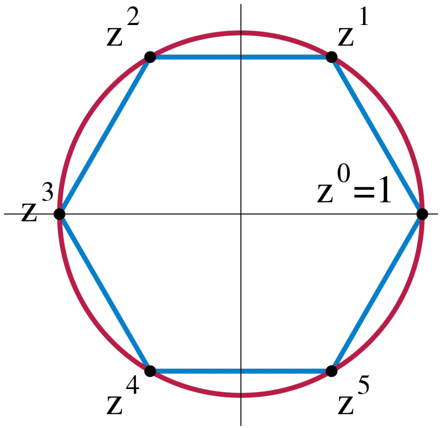
$$\mathrm{GF}(q) = \mathrm{GF}(p)[X]/(P)$$

BCH Codes: *A crash course in* ***Galois Fields***.

GF(2^3) = GF(8) based on the primitive P(x) = $x^3 + x^2 + 1$ = (1101) = 13 (decimal)

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

×	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	4	6	5	7	1	3
3	3	6	5	1	2	7	4
4	4	5	1	7	3	2	6
5	5	7	2	3	6	4	1
6	6	1	7	2	4	3	5
7	7	3	4	6	1	5	2



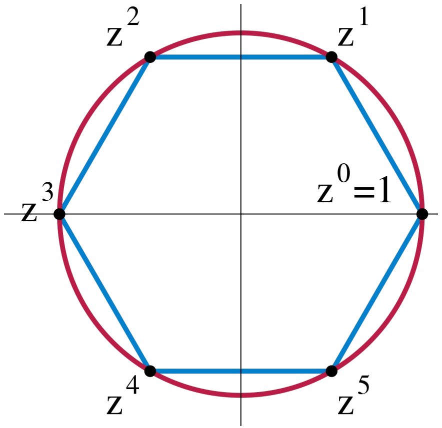
$$\mathrm{GF}(q) = \mathrm{GF}(p)[X]/(P)$$

BCH Codes: *A crash course in* ***Galois Fields***.

$GF(2^3) = GF(8)$ based on the primitive $P(x) = x^3 + x^2 + 1 = (1101) = 13$ (decimal)

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

×	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	4	6	5	7	1	3
3	3	6	5	1	2	7	4
4	4	5	1	7	3	2	6
5	5	7	2	3	6	4	1
6	6	1	7	2	4	3	5
7	7	3	4	6	1	5	2



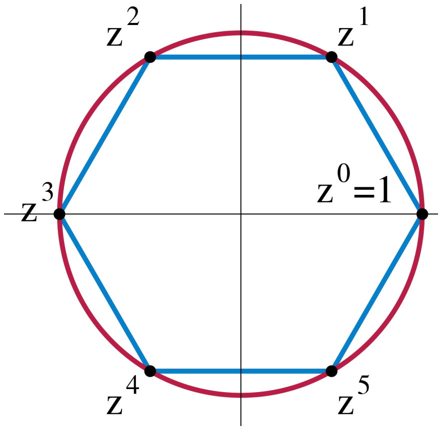
$$GF(q) = GF(p)[X]/(P)$$

BCH Codes: *A crash course in* ***Galois Fields***.

$GF(2^3) = GF(8)$ based on the primitive $P(x) = x^3 + x^2 + 1 = (1101) = 13$ (decimal)

The primitive element $\boldsymbol{\alpha}$: Cyclically spans the entire field

$\alpha^0 = 1$	$\alpha^8 = 5$
$\alpha^1 = 2$	$\alpha^9 = 10$
$\alpha^2 = 4$	$\alpha^{10} = 7$
$\alpha^3 = 8$	$\alpha^{11} = 14$
$\alpha^4 = 3$	$\alpha^{12} = 15$
$\alpha^5 = 6$	$\alpha^{13} = 13 = \mathtt{0xD} = \alpha^3 + \alpha^2 + 1$
$\alpha^6 = 12$	$\alpha^{14} = 9$
$\alpha^7 = 11$	$\alpha^{15} = 1$



$$\mathrm{GF}(q) = \mathrm{GF}(p)[X]/(P)$$

BCH Encoder Design

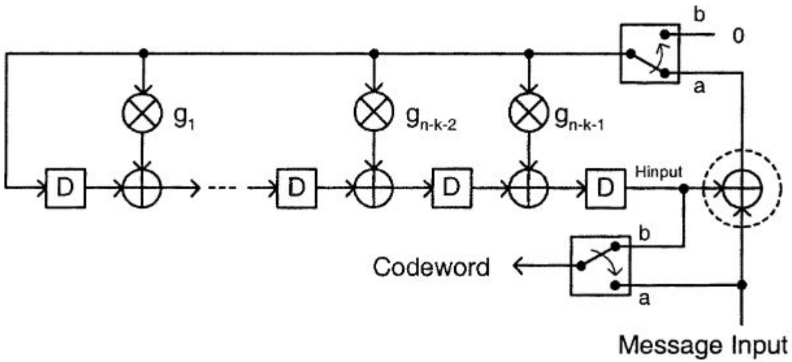
Can we achieve **low latency**?

Design Challenges:

- Implement $c(x) = m(x) \, . \, g(x)$
- Requires **polynomial multiplication**.
- Sometimes, even **division**
- Has an **obstinate** loop bound.

BCH Encoder Design

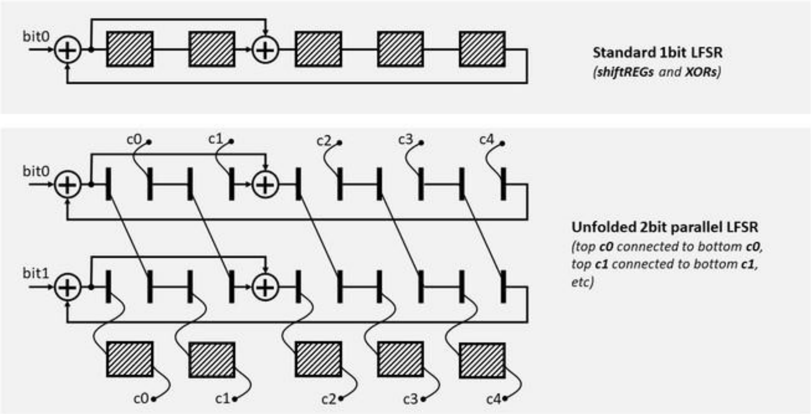
- Implement the **systematic** form $c(x) = q(x) \, . \, g(x) = m(x).x^{n\text{-}k} + r(x)$
- Traditionally done using **LFSR**s.
- Has the disadvantage of serial **input** and **output**, i.e.,
  Latency = code-length clock cycles.



BCH Encoder Design

- **Look-ahead** and ***J*-unfolding** can be used to implement.
- ***J-unfolding*** alone does not change the loop-bound. So one can completely unroll the loop (get a loop-bound of infinity).
- Interestingly, **systematic** encoding is exactly the same as **CRC** encoding.

Parallel CRC circuit example (poly : 1 + X^2 + X^5)



BCH Decoder Design

Philosophy:
Accelerate the common case.

Decoding Steps:

- Calculate **Syndromes**:
  Evaluate $c(x) = m(x) \,.\, g(x)$ at α^n .
- Ascertain **Error Count**
- Obtain **Error Locator Polynomial**
- Use **Chien Search** to find error locations.

BCH Decoder: *The basics*

- Suppose the transmitter sent $c(x) = m(x) \, . \, g(x)$.
- Suppose an error causes bit corruption. Then the **received polynomial** is:

$$r(x) = c(x) + e(x).$$

- Evaluating at k roots of $g(x)$ we have:

$$r(\alpha^k) = c(\alpha^k) + e(\alpha^k)$$

$$r(\alpha^k) = 0 + e(\alpha^k)$$

- $r(\alpha^k)$ is the k^{th} syndrome S_k.

For v errors:

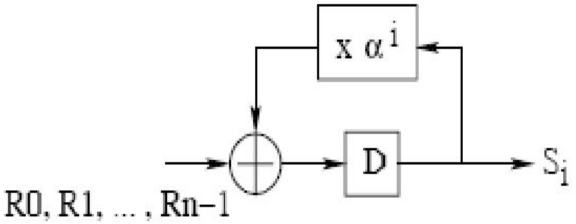
$$\begin{aligned} S_1 &= X_1 + X_2 + \cdots + X_\nu \\ S_2 &= X_1^2 + X_2^2 + \cdots + X_\nu^2 \\ &\vdots \\ S_{2t} &= X_1^{2t} + X_2^{2t} + \cdots + X_\nu^{2t}. \end{aligned}$$

BCH Decoder: *Calculating the syndromes*

- **Syndrome** calculation is the evaluation of received polynomial.
- A **non-zero** syndrome hints towards errors.
- Traditionally, recursive methods have been used. Rely on:

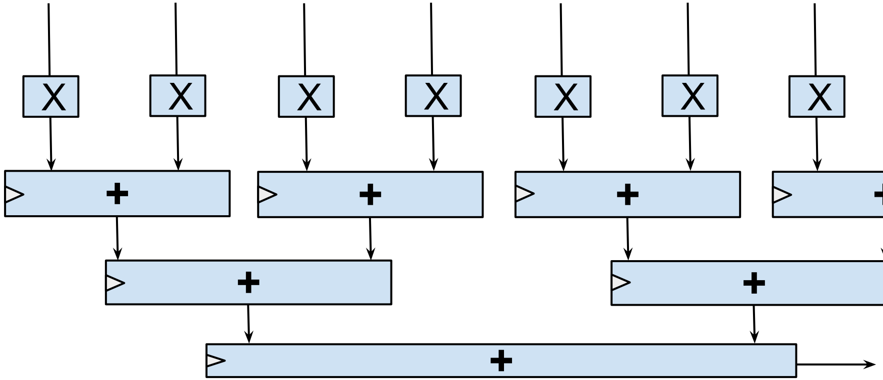
$$S_k = m_{k+3}\alpha^{k+3} + m_{k+2}\alpha^{k+2} + m_{k+1}\alpha^{k+1} + m_k\alpha^k = \alpha\,(\,\alpha\,(\,\alpha\,(\,m_k\alpha^k + m_{k+1})+m_{k+2})+m_{k+3})$$

- Instead, I have opted to use **unrolled xor-sum calculator**.



BCH Decoder: *Calculating the syndromes*

- Unrolled **Log-pipelined** xor-sum calculator:
  - **Access** all the powers of α for the received polynomial.
  - **Mask** the ones that coincide with 0 coefficients.
  - Perform **XOR sum** in parallel.
  - Insert *log(n)* **registers** at the end.
  - **Re-time** using synopsys design compiler.



$\alpha^0 = 1$

$\alpha^1 = 2$

$\alpha^2 = 4$

$\alpha^3 = 8$

$\alpha^4 = 3$

$\alpha^5 = 6$

$\alpha^6 = 12$

$\alpha^7 = 11$

BCH Decoder: *Ascertain Error Count*

- Recall that $S_1 = \alpha^i + \alpha^j + \alpha^k \ldots$ if i, j, k are the error locations.
- Hardwired **power** and **log** tables are used to calculate powers or indices of **roots**. They occur so often, that hardwiring is justified.
- For **no errors**, $S_1 = S_3 = 0$.
- For **single errors**, $S_1^3 = S_3$, or else we have **2 or more errors**. We can use these fact to get the error count.
- Hardwired **power** and **log** tables are used to calculate powers or indices of **roots**. They occur so often, that hardwiring is justified.

BCH Decoder: *Finding Error Locations (Single Errors)*

- Traditionally, error locations are found using **Chien Search** Algorithm, which is iterative, and costly in time.
- For single errors, which account for most of the errors, we have a much faster option:
  - Given that $S_1 = \alpha^i$ if we have a single error at $i\text{-}th$ location.
  - We can simply calculate the error location as $log(\alpha^i)$.
  - Again, hardwired log tables are used for this.

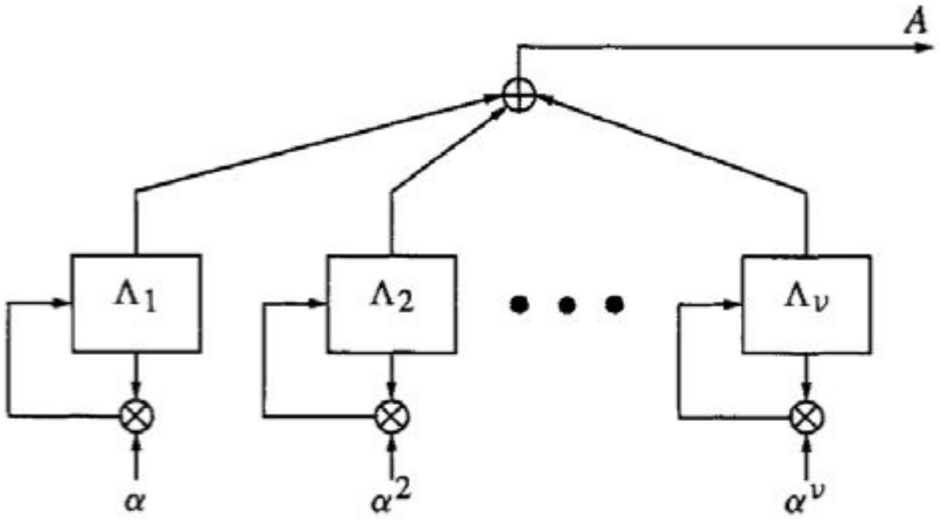
BCH Decoder: *Finding Error Locations (2 Errors)*

- Assuming the case of a 2-error correct and 1-error detect, let's imagine a polynomial:

$$\Lambda(x) = (1- \alpha^i x^{-1})\times(1- \alpha^j x^{-1})$$

- The above is called a **Reverse Error Locator Polynomial**. It is **zero**, whenever we put in an α^i , if is i the index of an error.
- $\Lambda(x) = (1- \alpha^i x^{-1})\times(1- \alpha^j x^{-1}) = 1 - (\alpha^i+\alpha^j)\, x^{-1} + (\alpha^i.\alpha^j)\, x^{-2}$
- The **first** highlighted term is S_1, while the **second** is a function of S_1 and S_3.
- We can use **Chien Search** here to find one index, let's say i . Then j can be found using $S_1 = \alpha^i+\alpha^j$, therefore, $\alpha^j = S_i+\alpha^i$.

BCH Decoder: *Chien Search*



Simulations and Synthesis Results

Implementation Specs:

- $(n, k) = (255 - 111, 239 - 111) = (144, 128)$.
- Message: **16** Bytes x 4 = **64** Bytes
- Codeword: **18** Bytes x 4 = **72** Bytes
- Can correct upto **8** errors: **2** errors per block.

MATLAB Simulation:

Editor - C:\Users\HP\REPOS\AdvancedVLSIfinalProject\BCH_decoder.m

code.m | BCH_decoder.m | eval_poly.m | log_table_generator.m | check_poly.m | includable_table.sv

```
function [idx, error_count, corrected_codeword] = BCH_decoder(input_codeword)
    idx = 15;
    a = gf([1:15], 4, 19);
    alpha = a(1+1);

    syndrome_1 = eval_poly(input_codeword, alpha.^1);
    syndrome_3 = eval_poly(input_codeword, alpha.^3);

    single_bit_error_flag = syndrome_1.^3 + syndrome_3;

    % determine error count
    error_count = 0;

    if syndrome_1 == 0 && syndrome_3 == 0
        error_count = 0;
    elseif (single_bit_error_flag == 0)
        error_count = 1;
    elseif (single_bit_error_flag ~= 0)
        error_count = 2;
    end
```

Command Window

New to MATLAB? See resources for Getting Started.

```
error_count: 2

Original : 1 1 1 0 1 0 0 0 1 0 0 0 0 0 0
Received : 1 1 1 1 1 0 0 0 1 0 0 0 0 1 0
Corrected: 1 1 1 0 1 0 0 0 1 0 0 0 0 0 0
```

Verilog Simulation:

Design Compiler Report: *Low Latency (2 ns) Decoder at 1 GHz.*

```
U2509/Y (INVX1)                            0.02       0.29 f
U2752/Y (AND2X2)                           0.03       0.33 f
U2530/Y (OAI21X1)                          0.04       0.37 r
U846/Y (OR2X1)                             0.04       0.41 r
U2166/Y (OR2X2)                            0.04       0.44 r
U1050/Y (INVX1)                            0.02       0.46 f
U1051/Y (NOR2X1)                           0.03       0.49 r
U4077/Y (INVX1)                            0.02       0.51 f
U4613/Y (NAND3X1)                          0.03       0.54 r
U1407/Y (BUFX2)                            0.04       0.58 r
U1913/Y (AOI21X1)                          0.02       0.59 f
U1576/Y (INVX1)                            0.01       0.60 r
U1264/Y (AND2X2)                           0.04       0.64 r
U1094/Y (OR2X2)                            0.07       0.71 r
U1245/Y (INVX2)                            0.03       0.74 f
U1436/Y (AND2X2)                           0.04       0.78 f
U3885/Y (INVX1)                            0.02       0.79 r
U1633/Y (AND2X1)                           0.04       0.83 r
U3676/Y (INVX1)                            0.03       0.86 f
U5147/Y (OAI21X1)                          0.04       0.91 r
U1681/Y (AND2X1)                           0.03       0.93 r
U2796/Y (INVX1)                            0.02       0.95 f
R_24/D (DFFPOSX1)                          0.00       0.95 f
data arrival time                                     0.95

clock clk (rise edge)                      1.00       1.00
clock network delay (ideal)                0.01       1.01
R_24/CLK (DFFPOSX1)                        0.00       1.01 r
library setup time                        -0.06       0.95
data required time                                    0.95
-----------------------------------------------------------
data required time                                    0.95
data arrival time                                    -0.95
-----------------------------------------------------------
slack (MET)                                           0.00


***** End Of Report *****
```

Rensselaer

why not change the world?®

Thank You!

Asad Ul Haq | **RIN: 662067056** | 04/18/2024