Galois Field Algebra and RAID6

By David Jacob

Overview

- Galois Field
  - Definitions
  - Addition/Subtraction
  - Multiplication
  - Division
  - Hardware Implementation
- RAID6
  - Definitions
  - Encoding
  - Error Detection
  - Error Correction
  - Hardware Implementations

Galois Field (GF)

- A finite field with integer elements
- All GF operations are closed
  - Operations on a element give another element in the field
- The field is generated using a generating polynomial, F
  - All math is done modulo F

GF Notation

- GF(p^n)
  - p = prime that defines number of numbers per digit
    - Ex. GF(2) = binary
  - n = highest order of generating polynomial; also the number of digits for each number in the field
    - E.g. GF(2^8) = 8-bit binary field (aka: every element is a byte) This is the field that will be used throughout the rest of this discussion
  - For GF(2^8), F = $x^8+x^4+x^3+x^2+1$

Addition/Subtraction

- Defined as addition/subtraction modulo p.
  – In GF(2), this is the XOR operation

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	0

Multiplication

- Multiplication modulo F
- Ex.
  - F = 100011101, A = 10101010, B = 00000010
  - AxB = (A*B) mod F

    = (101010100) mod 100011101

    = 01001001

Multiplication by 2^x

- As shown before, this is equivalent to a LFSR with a feedback of F that is shifted x times.
- Since fields are also mathematical rings, all elements are a power of 2, so this can be used to multiply any numbers A and B if you know what $\log_2(B)$ is
- If you are multiplying by a constant, this LFSR can be unrolled and combined to reduce time and logic

$$
\begin{aligned}
2B_7 &\Leftarrow B_6 \\
2B_6 &\Leftarrow B_5 \\
2B_5 &\Leftarrow B_4 \\
2B_4 &\Leftarrow B_3 \oplus B_7 \\
2B_3 &\Leftarrow B_2 \oplus B_7 \\
2B_2 &\Leftarrow B_1 \oplus B_7 \\
2B_1 &\Leftarrow B_0 \\
2B_0 &\Leftarrow B_7
\end{aligned}
$$

Multiplication by 2^x

X^0	X^1	X^2	X^3	X^4	X^5	X^6	X^7
X^7	X^0	X^1+X^7	X^2+X^7	X^3+X^7	X^4	X^5	X^6
X^6	X^7	X^0+X^6	$X^1+X^6+X^7$	$X^2+X^6+X^7$	X^3+X^7	X^4	X^5
X^5	X^6	X^5+X^7	$X^0+X^5+X^6$	$X^1+X^5+X^6+X^7$	$X^2+X^6+X^7$	X^3+X^7	X^4
X^4	X^5	X^4+X^6	$X^4+X^5+X^7$	$X^0+X^4+X^5+X^6$	$X^1+X^5+X^6+X^7$	$X^2+X^6+X^7$	X^3+X^7
X^3+X^7	X^4	$X^3+X^5+X^7$	$X^3+X^4+X^6+X^7$	$X^3+X^4+X^5$	$X^0+X^4+X^5+X^6$	$X^1+X^5+X^6+X^7$	$X^2+X^6+X^7$
$X^2+X^6+X^7$	X^3+X^7	$X^2+X^4+X^6+X^7$	$X^2+X^3+X^5+X^6$	$X^2+X^3+X^4$	$X^3+X^4+X^5$	$X^0+X^4+X^5+X^6$	$X^1+X^5+X^6+X^7$
$X^1+X^5+X^6+X^7$	$X^2+X^6+X^7$	$X^1+X^3+X^5+X^6$	$X^1+X^2+X^4+X^5$	$X^1+X^2+X^3+X^7$	$X^2+X^3+X^4$	$X^3+X^4+X^5$	$X^0+X^4+X^5+X^6$

Fast General-Purpose Multiplication

- If you want to multiply by a number that isn't a power of 2, use Distributive property.

$$A \times B = \sum_{i=0}^{n} (A_i \times 2^i \times B)$$

  - Multiplying 2^ixB can be done using unrolled LFSRs
  - A(i)x(2^ixB) is done with AND gates
  - Addition is XOR gates
- This results in general purpose multiplication being done in combinational time

GF Division

- Defined as multiplication by the multiplicative inverse
  - $A/B = AxB^{-1}$
- The multiplicative inverse is unique for every element in the field
- Multiplicative inverse defined as:
  - $AxA^{-1} = 1$

Multiplicative Inverse

- There are 3 ways of finding multiplicative inverse: Brute Force, Fermat's Little Theorem, and Extended Euclidean Algorithm
- Brute Force method of multiplying by each possible element until one of the products is 1 is obviously very expensive in either time or hardware

Fermat's Little Theorem

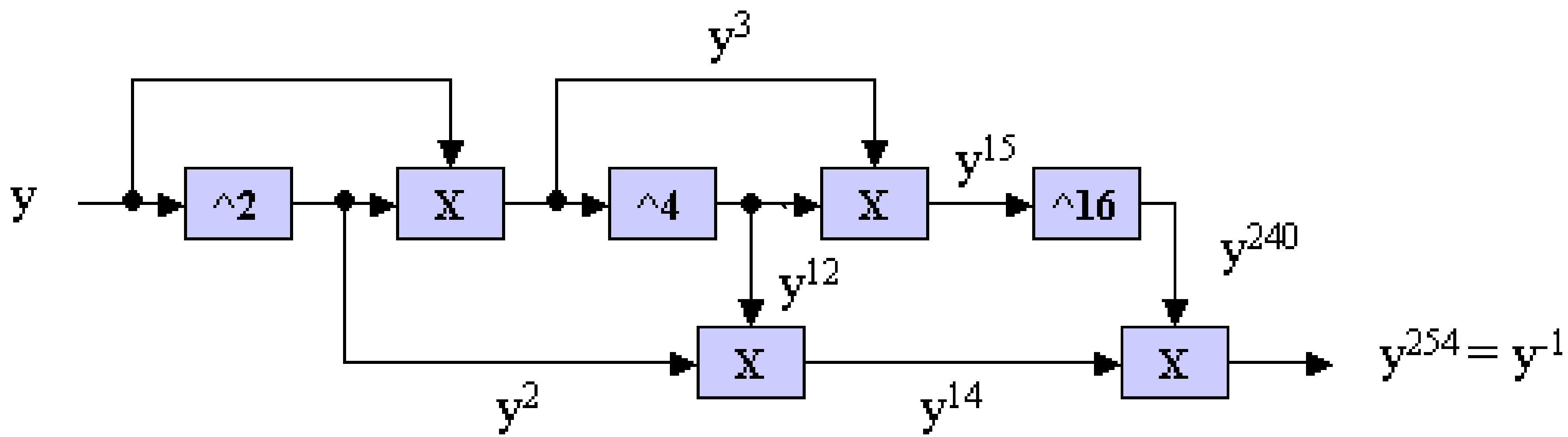
- Fermat's little theorem involves math modulo F, and can be used like this:

$$A^{p^n} = A \, mod \ F$$
$$A^{p^{n-1}} = 1 \, mod \ F$$
$$A \cdot A^{p^{n-2}} = 1 \, mod \ F$$

- Therefore, in GF(2^8): $A^{254} = A^{-1}$

Fermat's Little Theorem (Tom Wada, 2003)

- Using some "tricks" this can be calculated much easier than it would seem
  - This still requires the equivalent of 11 general-purpose multipliers



Euclidean Algorithm

- The Euclidean Algorithm is used to find the Greatest Common Denominator (GCD) of two numbers.
- If you are trying to find the GCD(A,B), and assuming A>=B
  - Q = A/B (integer division), R = A mod B

$$
\begin{aligned}
A &= Q \cdot B + R \\
R &= A - Q \cdot B \\
R &= m \cdot GCD(A, B) - Q \cdot n \cdot GCD(A, B) \\
R &= GCD(A, B) \cdot (m - Q \cdot n) \\
R &= GCD(A, B) \cdot p
\end{aligned}
$$

- So, R is also a multiple of the GCD(A,B), so GCD(A,B) = GCD(B,R)
- This can be continued until there is no remainder, in which case, the last value divided by is the GCD(A,B)

Euclidean Algorithm

```
GCD(A,B):
   // initialize
   Rn := A;        R  := B;
   repeat
    // shift the values back for the next reduction
    Rm := Rn;
    Rn := R;
    // reduce
    Q := Rm/Rn;          //this is integer division
    R := Rm - Q * Rn;
   until R = 1;
   return Rn;
end GCD(A,B);
```

Extended Euclidean Algorithm

- Not only find GCD, but constants of multiplication

$$GCD(A, B) = A \cdot X + B \cdot Y$$

- Uses the quotients that are thrown away in the normal Euclidean Algorithm to find X and Y

Extended Euclidean Algorithm

- This is is found by assuming:

$$R_i = A \cdot X_i + B \cdot Y_i$$

- So:

$$R_i = R_{i-2} - \frac{R_{i-2}}{R_{i-1}} \cdot R_{i-1}$$

$$R_i = (A \cdot X_{i-2} + B \cdot Y_{i-2}) - \frac{R_{i-2}}{R_{i-1}} \cdot (A \cdot X_{i-1} + B \cdot Y_{i-1})$$

$$R_i = A \cdot X_{i-2} + B \cdot Y_{i-2} - \frac{R_{i-2}}{R_{i-1}} \cdot A \cdot X_{i-1} + \frac{R_{i-2}}{R_{i-1}} \cdot B \cdot Y_{i-1}$$

$$R_i = A \cdot (X_{i-2} - \frac{R_{i-2}}{R_{i-1}} \cdot X_{i-1}) + B \cdot (Y_{i-2} + \frac{R_{i-2}}{R_{i-1}} \cdot Y_{i-1})$$

Extended Euclidean Algorithm

- Since X and Y are defined recursively, starting points are needed
- Consider that the first two “remainders” are A and B

$$R_{-2} = A = A \cdot 1 + B \cdot 0$$
$$R_{-1} = B = A \cdot 0 + B \cdot 1$$

Extended Euclidean Algorithm

```
Ext_GCD(A,B):
   //initialize
   Rn := A;         R  := B;
   Xn := 1;         X  := 0;
   Yn := 0;         Y  := 1;
   repeat
        // shift the values back for the next reduction
        Rm := Rn;         Rn := R;
        Xm := Xn;         Xn := X;
        Ym := Yn;         Yn := Y;
        // reduce
        Q := Rm/Rn;       //this is integer division
        R := Rm - Q * Rn;
        // update X and Y
        X := Xm - Q * Xn;         Y := Ym - Q * Yn;
   until R = 1;
   return Rn,X,Y;
end Ext_GCD(A,B);
```

How does Extended Euclidean Algorithm Help?

- In GF algebra, F is coprime with all elements in the field and multiplication is done modulo F so:

$$A \times X \oplus F \times Y = GCD(A, F)$$
$$A \times X \oplus F \times Y = 1$$
$$A \times X = F \times Y \oplus 1$$
$$A \times X = 0 \oplus 1$$
$$A \times X = 1$$

- So X is the multiplicative inverse of A

Improving Ext. Euclidean Algorithm for GF(2)

- First, the Y is not important, so don't keep track of it
- Second, since the point of finding the multiplicative inverse is to implement division, finding Q = Rn/Rm is impossible.
  - Q isn't important either, just finding the remainder after the division

Finding the GF(x) Remainder (Brent et. al, 1984)

- Basically do binary "long division" until the remainder is found

```
MOD(A,B)
      delta := deg A - deg B;
      repeat
            // scale A and X
            Bs := x^delta * B;          Xs := x^delta * X;
            // reduce
            A := A – Bs;                Y := Y - Xs;
            // recalculate degree
            delta := deg A - deg B;
      until delta < 0;
      return A, Y;
end MOD(A,B);
```

Finding the GF(2) Remainder (Brunner et. al, 1993)

- How to do “x^{delta} * B” efficiently?
  - Could shift both values until the Msb are high
  - Then when subtraction is done, the top bit of A is 0, so it can be shifted, and delta decremented
- Remember that the result must be in the Galois Field, so math on it should be GF Algebra!
  - GFM2(A) = returns A times 2 (GF Multiplication)
  - GFD2(A) = returns A divided by 2 (GF Division)

Finding the GF(2) Remainder (Brunner et. al, 1993)

```
MOD(A,B)
    delta := 0;
    repeat
     if R(N) = 0 then            // scale up B and X and increment delta
          B := B << 1;           X := GFM2(X);        delta := delta + 1;
     else
          if A(N) = 0 then       // scale up A and scale down X
               A := A << 1;      X := GFD2(X);
          else
          // if both MSb's are high, reduce B and Y and scale A and X
               A := A – B;       Y := Y xor X;
               A := A << 1;      X := GFD2(X);
          end if;
          delta := delta - 1;
     end if;
    while delta >= 0;
    return A and Y;
end MOD(A,B);
```

GF(2) Multiplicative Inverse (Brunner et. al, 1993)

- Combining this method of finding the remainder with the original Extended Euclidean Algorithm gives a usable implementation
- Since the order of F is N, and worst case, the order of A can be of order N, the loop needs to be done 2*N times
- To save registers, X and A can be used as temporary registers, since the final value of them is unimportant anyway

GF(2) Multiplicative Inverse (Brunner et. al, 1993)

```
GF_Inversion(A)
   Rn := F;          R  := A;
   Xn := 1;          X  := 0;
   delta := 0;

   for i = 1 to 2*N
    if R(N) = 0 then                    // scale up B and X and increment delta
        Rn := Rn << 1;      X := GFM2(X);
        delta := delta + 1;
    else
        if Rn(N) = 1 then
            R := R – Rn;         X := X xor Xn;
        end if;
        R := R  << 1;
        if delta = 0 then                    // division is done, so swap variables for new division
            swap(R,Rn);          swap(X,Xn);
            X := GFM2(X);
        else
            X := GFD2(X);
            delta := delta - 1;
        end if;
    end if;
   end loop;
   return R;
end GF_Inversion(B);
```

GF(2) Multiplicative Inverse In Hardware (Brunner et. al, 1993)

- To implement things in hardware, concurency can be taken advantage of
- To simplify hardware design, signals T and W are added

GF(2) Multiplicative Inverse In Hardware (Brunner et. al, 1993)

```
GF_Inversion(B)
   Rn := F;       R  := B;
   Xn := 1;  X  := 0;
   delta := 0;

   for i = 1 to 2*N
    if R(N) = 1 and Rn(N) = 1 then
        T := R xor Rn;
        W := X xor Xn;
    else
        T := R;
        W := X;
    end if;
```

GF(2) Multiplicative Inverse In Hardware (Brunner et. al, 1993)

```
      if R(N) = 0 then
            R := R << 1;                    Rn := T;
            X := GFM2(X);             Xn := W;
            delta := delta + 1;
      else
            if delta = 0 then
                  Rn := R;                  R := T << 1;
                  Xn := X;            X := GFM2(W);
                  delta := delta + 1;
            else
                  Rn := T << 1;             R := R;
                  Xn := W;                  X := GFD2(X);
                  delta := delta - 1;
            end if;
      end if;
    end loop;
    return R;
GF_Inversion(A);
```

Division by 2^x

- Dividing by 2 is the inverse of multiplying by 2, so a LFSR which reverses the multiply by 2 LFSR would divide by 2.
- This can once again be expanded to multiply by any constant.

$$
\begin{aligned}
2B_7 &\Leftarrow B_6 \\
2B_6 &\Leftarrow B_5 \\
2B_5 &\Leftarrow B_4 \\
2B_4 &\Leftarrow B_3 \oplus B_7 \\
2B_3 &\Leftarrow B_2 \oplus B_7 \\
2B_2 &\Leftarrow B_1 \oplus B_7 \\
2B_1 &\Leftarrow B_0 \\
2B_0 &\Leftarrow B_7
\end{aligned}
\qquad \Rightarrow \qquad
\begin{aligned}
B_7 &\Leftarrow 2\mathrm{B}_0 \\
B_6 &\Leftarrow 2\mathrm{B}_7 \\
B_5 &\Leftarrow 2\mathrm{B}_6 \\
B_4 &\Leftarrow 2\mathrm{B}_5 \\
B_3 &\Leftarrow 2\mathrm{B}_4 \oplus 2\mathrm{B}_0 \\
B_2 &\Leftarrow 2\mathrm{B}_3 \oplus 2\mathrm{B}_0 \\
B_1 &\Leftarrow 2\mathrm{B}_2 \oplus 2\mathrm{B}_0 \\
B_0 &\Leftarrow 2\mathrm{B}_1
\end{aligned}
$$

Multiplication/Division with Lookup Tables

- Multiplication and Division can also be done w/ lookup tables

$$A \times B = \exp(\log(A) + \log(B))$$
$$A / B = \exp(\log(A) - \log(B))$$

- Requires 256X8 lookup tables
  - Typically done in hard RAM blocks, so as not to use up fabric resources
  - The lookup tables are at most dual ported, so 2 RAM blocks are needed per pair of inputs

RAID

- Redundant Array of Independent (Inexpensive) Drives
- RAID comes in 4 common “varieties”
  - RAID0 - data striped across the array
  - RAID1 - data mirrored across the array
  - RAID5 - data striped across the array with one parity block
  - RAID6 - data striped across the array with two parity blocks

RAID 6

- RAID6 uses GF(2^8) Algebra to create 2 redundant parity blocks
  - Data is striped in data blocks of 1 sector
  - 2 blocks are used for parity information so usable array space is N – 2 drives
  - Can detect 1 corrupt data bock
  - Can recover 2 corrupt data blocks (assuming some other method of detecting the error exists)

RAID6 Parity

- The P block is: $P = \sum_{i=0}^{n-2} (D_i)$
  - This is the same as RAID5 parity
  - Allows for easy generation and recovery
- The Q block is: $Q = \sum_{i=0}^{n-2} (2^i \times D_i)$
  - More complicated generation, but allows for error detection

RAID6 Error Detection

- If the data at (unknown) location L is corrupted to X, then:

$$P = D_0 \oplus \ldots \oplus D_{L-1} \oplus D_L \oplus D_{L+1} \oplus \ldots \oplus D_n$$
$$P' = D_0 \oplus \ldots \oplus D_{L-1} \oplus X \oplus D_{L+1} \oplus \ldots \oplus D_n$$
$$P \oplus P' = D_l \oplus X$$

$$Q = 2^0 \times D_0 \oplus \ldots \oplus 2^{L-1} \times D_{L-1} \oplus 2^L \times D_L \oplus 2^{L+1} \times D_{L+1} \oplus \ldots \oplus 2^n \times D_n$$
$$Q' = 2^0 \times D_0 \oplus \ldots \oplus 2^{L-1} \times D_{L-1} \oplus 2^L \times X \oplus 2^{L+1} \times D_{L+1} \oplus \ldots \oplus 2^n \times D_n$$
$$Q \oplus Q' = 2^L \times D_L \oplus 2^L \times X = 2^L \times (D_L \oplus X)$$

$$(P \oplus P') / (Q \oplus Q') = 2^L$$
$$\log((P \oplus P') / (Q \oplus Q')) = L$$

RAID6 Error Correction

- If 2 errors exist, there are 4 options of what they could be:
  - The two parity blocks
    - If this is the case, just recompute them
  - One data block and P
  - One data block and Q
  - Two data blocks

One Corrupted Data Block

- If only one data block is corrupted, and one of the parity is corrupted, then the data can be recreated from the good parity
  - If P is good than:

$$P = D_0 \oplus \ldots \oplus D_{L-1} \oplus D_L \oplus D_{L+1} \oplus \ldots \oplus D_n$$
$$0 = P \oplus D_0 \oplus \ldots \oplus D_{L-1} \oplus D_L \oplus D_{L+1} \oplus \ldots \oplus D_n$$
$$D_L = P \oplus D_0 \oplus \ldots \oplus D_{L-1} \oplus D_{L+1} \oplus \ldots \oplus D_n$$

  - If Q is good than recompute Q (called Q') with the bad data as zeros:

$$Q = 2^0 \times D_0 \oplus \ldots \oplus 2^{L-1} \times D_{L-1} \oplus 2^L \times D_L \oplus 2^{L+1} \times D_{L+1} \oplus \ldots \oplus 2^n \times D_n$$
$$Q' = 2^0 \times D_0 \oplus \ldots \oplus 2^{L-1} \times D_{L-1} \oplus 2^L \times 0 \oplus 2^{L+1} \times D_{L+1} \oplus \ldots \oplus 2^n \times D_n$$
$$Q \oplus Q' = 2^L \times D_L$$
$$(Q \oplus Q')/2^L = D_L$$

Two Data Drives Corrupted

- Data is corrupted on drives L and K (assuming K<L), recalculate P and Q (P' and Q') with erroneous data blocks as zeros:

$$P = D_0 \oplus \ldots \oplus D_{K-1} \oplus D_K \oplus D_{K+1} \oplus \ldots \oplus D_{L-1} \oplus D_L \oplus D_{L+1} \oplus \ldots \oplus D_n$$

$$P' = D_0 \oplus \ldots \oplus D_{K-1} \oplus 0 \oplus D_{K+1} \oplus \ldots \oplus D_{L-1} \oplus 0 \oplus D_{L+1} \oplus \ldots \oplus D_n$$

$$P = P' \oplus D_K \oplus D_L$$

$$Q = 2^0 \times D_0 \oplus \ldots \oplus 2^{K-1} \times D_{K-1} \oplus 2^K \times D_K \oplus 2^{K+1} \times D_{K+1} \oplus \ldots$$
$$\oplus 2^{L-1} \times D_{L-1} \oplus 2^L \times D_L \oplus 2^{L+1} \times D_{L+1} \oplus \ldots \oplus 2^n \times D_n$$

$$Q' = 2^0 \times D_0 \oplus \ldots \oplus 2^{K-1} \times D_{K-1} \oplus 2^K \times 0 \oplus 2^{K+1} \times D_{K+1} \oplus \ldots$$
$$\oplus 2^{L-1} \times D_{L-1} \oplus 2^L \times 0 \oplus 2^{L+1} \times D_{L+1} \oplus \ldots \oplus 2^n \times D_n$$

$$Q = Q' \oplus 2^K \times D_K \oplus 2^L \times D_L$$

Two Data Drives Corrupted

- Then solve the first equation for D_L and the second for D_K and plug the in for D_K:

$$P = P' \oplus D_K \oplus D_L$$
$$D_L = P \oplus P' \oplus D_K$$

$$Q = Q' \oplus 2^K \times D_K \oplus 2^L \times D_L$$
$$D_K = 2^K \times (Q \oplus Q') \oplus 2^{L-K} \times D_L$$

$$D_L = P \oplus P' \oplus 2^K \times (Q \oplus Q') \oplus 2^{L-K} \times D_L$$
$$D_L \oplus 2^{L-K} \times D_L = P \oplus P' \oplus 2^K \times (Q \oplus Q')$$
$$(2^{L-K} \oplus 1) \times D_L = P \oplus P' \oplus 2^K \times (Q \oplus Q')$$
$$D_L = \frac{P \oplus P' \oplus 2^K \times (Q \oplus Q')}{2^{L-K} \oplus 1}$$

Two Data Drives Corrupted

- Since K<L, it can be assumed that $2^{L-K} \oplus 1 > 1$
  - No division by zero possible
- After D_L is found, plug back in for D_K in the P equation solved for D_K:

$$D_k = P \oplus P' \oplus D_L$$

Cost of Implementing in FPGA

- FPGAs use 4 input lookup tables (LUT4) in the fabric to implement logic
  - 2-input AND has same logic cost as 2-input XOR
  - 2-input XOR has same logic cost as 4-input XOR
- If more than 4 inputs are needed, another LUT4 is cascaded to make a 7-input gate
  - This can be repeated many times in a tree (with a branching factor of 4), until required number of inputs is supplied:
  - Hardware cost is: $LUT4 / N - input\ gate = \lceil (N-1)/3 \rceil$
  - Speed cost is: $delay = Depth\ of\ LUT4\ tree = \lceil \log_4 N \rceil$

What is the Best way to do RAID6 in Hardware?

- With various ways, which is the best?
- 3 different things to be discussed
  - Encoding
  - Decoding to detect error
  - Decoding to correct errors

FPGA Hardware Encoding

$$Q = \sum_{i=0}^{N} (2^i \times Di)$$

- Can be done with 3 different methods:
  - Lookup Tables
    - Requires N 256x8 lookup tables to be done (assuming N is even)
    - Good for when slice count becomes an issue and timing constraints are relaxed
  - Hardware General-Purpose Multipliers
    - Easily expandable and requires no block RAM
  - Hardware Special-Purpose Multipliers
    - Uses multiplication by 2^x multipliers to multiply by the required constants
    - Requires very few slices and no block RAM

FPGA Error Detection

$$\log((P \oplus P')/(Q \oplus Q')) = L$$

- Requires a log table, so only sensible way of doing it is with lookup tables
- This also allows for simplified logic

$$\log(\exp(\log(P \oplus P') - \log(Q \oplus Q'))) = L$$
$$\log(P \oplus P') - \log(Q \oplus Q') = L$$

  - Only requires one dual-ported log table, and no exponentiation table this way

FPGA 2 Error Correction

$$D_L = \frac{P \oplus P' \oplus 2^K \times (Q \oplus Q')}{2^{L-K} \oplus 1}$$

- Can be done 3 different ways:
  - Lookup tables
    - Requires 4 lookup tables, or 2 if no pipelining is required
  - General-Purpose multiplication and Division
    - Quite a lot of hardware required
  - Special-Purpose Multiplication and Division
    - Use multiply/divide by constant circuits w/ multiplexer to use the proper one for the desired values of L and K
    - Need at most N-1 multiply by constants, and N-1 Divide by constants and 2 (N-1)-input Muxes

Conclusion

- Multiply/Divide by constant combinational circuits can be used to greatly reduce the complexity of RAID6 encoding and decoding

Any Questions?