2009

Efficient Implementation of RAID-6 Encoding and Decoding on a Field Programmable Gate Array (FPGA)

David Jacob
Virginia Commonwealth University

Efficient Implementation of RAID-6 Encoding and Decoding on a Field Programmable Gate Array (FPGA)

David Jacob

Director – Dr. James M. McCollum
Assistant Professor of Electrical & Computer Engineering

School of Engineering,
Virginia Commonwealth University,
Richmond, Virginia

December 10, 2009

Abstract

RAID-6 is a data encoding scheme used to provide single drive error detection and dual drive error correction for data redundancy on an array of disks. Here we present a thorough study of efficient implementations of RAID-6 on field programmable gate arrays (FPGAs). Since RAID-6 relies heavily on Galois Field Algebra (GFA), an efficient implementation of a GFA FPGA library is also presented. Through rigorous performance analysis, this work shows the most efficient ways to tradeoff FPGA resources and execution time when implementing GFA functions as well as RAID-6 encoding and decoding.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Redundant Array of Inexpensive Drives (RAID)

RAID is the concept of using an array of storage devices to function as one storage device[1]. This was motivated by the fact that the same capacity and relaibility can be attained in an array of small, cheap hard drives as in a large, expensive hard drive, but at a much lower price. There are 4 commonly used types of RAID: RAID-0, RAID-1, RAID-5, and RAID-6. RAID-0 is a pure data striping scheme with no redundancy. This gives the maximum performance and capacity to the array. RAID-1 is a pure data mirroring scheme. This provides maximum redundancy so that the array can survive the failure of all but one device with no loss of data. This comes at the expense of array capacity and write speed. RAID-5 uses striped data with a rotated parity. This allows near optimal capacity and performance while allowing a single storage device to fail with no loss of data. RAID-6 uses striped parity with two rotated parities. This system is similar to RAID-5 except two devices may be lost with no loss of data from the array. Implementing RAID-6 encoding and decoding in an FPGA is the focus of this work.

1.2 Field Programmable Gate Array (FPGA)

Field Programmable Gate Arrays (FPGAs) are Integrated Circuits (ICs) with various types of reconfigurable hardware blocks and interconnect. These hardware blocks are typically a 4-input look-up table (LUT4) and a flip-flop along with some other logic[2][3]. A single LUT4 can implement any arbitrary 4-input logic function. This general-purpose reconfigurable logic is called the FPGA "fabric". Unless the size or speed constraints are problematic, any design can be implemented by changing the look-up tables and interconnects in the fabric of a modern FPGA. Often, there are also specialized hardware blocks included in the FPGA, such as adders, RAM/ROMs, or even entire hard-core microprocessors[2][3]. These specialized hard-core blocks are generally referred to by their function (e.g. "RAM block" or "adder block") and are not considered "fabric".

FPGAs differ from Application-Specific Integrated Circuits (ASICs) in several ways. ASICs are fabricated circuits that have been designed to perform a specific task[4]. Because ASICs allow the designer more control over the specific implementations, a well-crafted ASIC will almost always execute a task faster than an FPGA designed to do the same task, while still requiring less power and hardware. In addition, because of the mask-based fabrication process, ASICs are less expensive to mass-produce than an FPGA[4]. However, traditional ASICs cannot be changed, so new ASICs would need to be created, along with creating new masks that are needed to fabricate them. This is an expensive and time consuming process, which makes modifying an ASIC design very undesirable[4]. By contrast, many FPGAs do not have this limitation. Though there are some FPGAs that are only programmable a single time (OTP), many FPGAs can be freely reprogrammed whenever desired by simply downloading a different configuration into the FPGA[5]. This reconfigurability allows much more flexibility in updating or improving the design, even after the distribution of the design. Also, each FPGA does not have the start-up cost of the creation of fabrication masks; thus, for low-volume production, it is typically less costly to produce FPGAs compared to ASICs.

FPGA designs also tend to have shorter time to market than ASIC designs. In both ASIC and FPGA development, HDL simulators are used to debug and verify the initial design. After this, the place and route process is many times automated, though in cutting-edge ASIC and FPGA designs, this will be done by hand. Then, an FPGA design can be quickly tested in the actual hardware. ASIC designs must be fabricated to do the equivalent test, and this can take up to 3 months in some cases. Then in either case, if any faults in the design are found, another iteration must be done of the design process. For an FPGA, this can be done very quickly, even dozens of times a day if required. For an ASIC, this is not possible. In addition, for every iteration of an ASIC that is sent to fabrication, there is an additional cost to fabricate the integrated circuit. There is no associated reoccurring cost to reconfigure an FPGA.

1.3 Motivation

This work was inspired by a design needed by NASA. The design was required to be radiation tolerant, and no radiation tolerant ASIC RAID-6 implementation exists. NASA asked that it be developed on a radiation tolerant FPGA. This was primarily because the space-application market is very low volume, so it would not be cost efficient to design and fabricate a radiation tolerant RAID-6 ASIC design. FPGAs are cost effective at this level of production because the non-reoccurring costs are much lower. Also, radiation tolerant FPGAs already exist (of which the Virtex-4 is one of), which mitigate the effects of the space environment. When developing ASICs for harsh environments (either Industrial or Space), the designer needs to take into account environmental factors. FPGA designs require less special consideration by the system designer, as the environmental consideration has already been partially addressed by the FPGA producer.

This work develops and evaluates methods of implementing RAID-6 IP Cores. Galois Field Algebra, which is instrumental in the efficient encoding and decoding of RAID-6 parity blocks, is also developed and evaluated. A novel method of doing division by a constant is developed and used to optimize RAID-6 error correction. Also, a new method for RAID-6 error detection is developed, which reduces hardware requirements for the operation. In addition, an optimized IP Core library using lookup-table and combinational based Galois Field Algebra is created for Galois Field Algebra, RAID-6 encoding, and using RAID-6 decoding to find a single error or correct two known errors.

1.4 Organization

An overview of RAID is given in Section 2.1. In Section 2.2, there is an overview of how mathematics in Galois Field Algebra (which RAID-6 depends on) is calculated. Following this, in Section 2.3, an overview of how to encode a RAID-6 stripe, and then how to use this encoded stripe to correct two errors, or detect a single error is discussed. Then in Section 2.4, there is a brief overview of the prior work in hardware and software GF Algebra implementations.

In Chapter 3, prior work and implementations in Galois Field Algebra are explored in more detail. These implementations are compiled into a VHDL Library of parts, which are evaluated in terms of both resource utilization and speed. Then in Chapter 4, the prior work and implementations in various RAID-6 operations are discussed, implemented in VHDL and evaluated. Also, several novel designs for combinational GF division by a constant, multiplication by a constant using logarithm tables, and RAID-6 error detection are explored and compared to existing work. Finally, Chapter 5 presents the conclusions of this research.

Chapter 2

Background Information and Prior Work

2.1 RAID-6 Overview

RAID-6 can accommodate 257 storage devices, but 2 storage devices worth of capacity is used to store parity information. RAID-6 was designed to be able to tolerate two storage device failures in the array without any corruption of the data. To do this, it has two rotating parity blocks with each stripe, which are called "P" and "Q". This is demonstrated in Figure 2.1. The parity blocks ("P" and "Q") are rotated within the stripe to even the wear on the storage devices when the full stripe is not being read at once.

RAID-6 also has the ability to detect when a block of data is corrupted. RAID-6 can sense and correct one erroneous block. Furthermore, RAID-6 can correct two erroneous blocks if an external source can identify which two

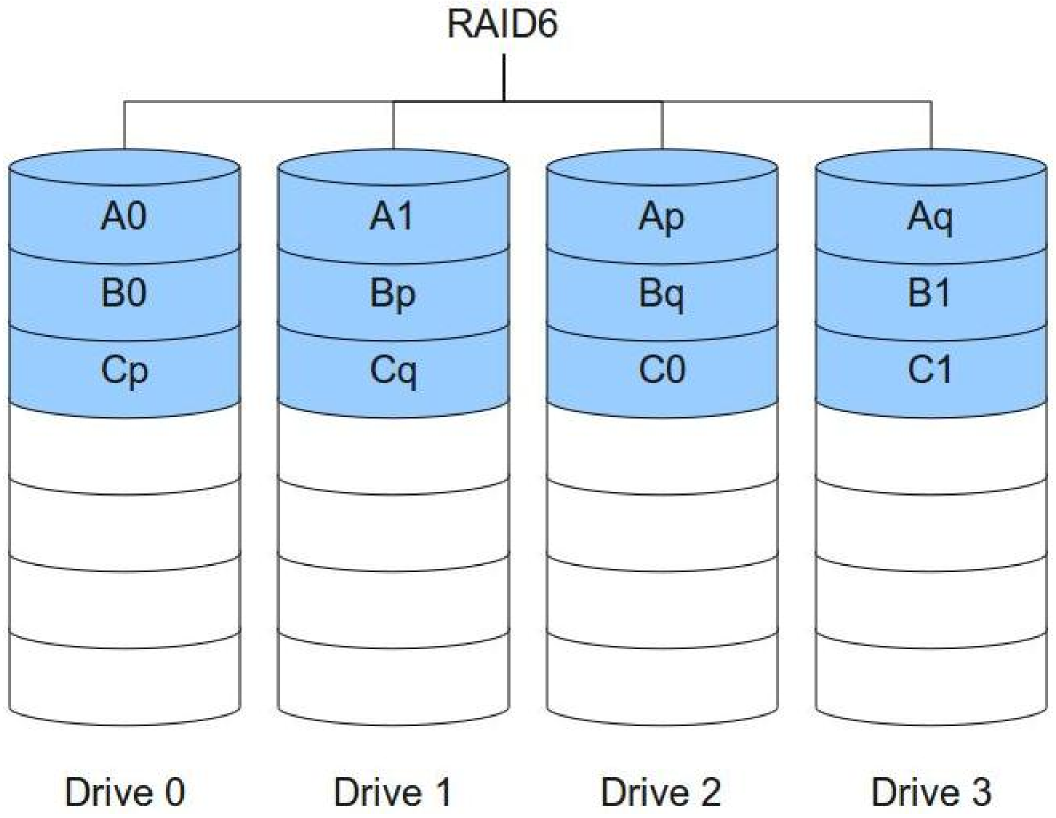

Figure 2.1: Raid-6 Configuration Diagram

blocks are corrupted. In exchange for these properties, RAID-6 loses two storage devices worth of performance and capacity, as they are used to store the parity information.

For RAID-6 encoding, the first parity bock (P) is a simple exclusive-or parity across the data stripe. The second parity block (Q) is encoded using a special form of Abstract Algebra called Galois Field Algebra. In Section 2.2 the background of how Galois Field Algebra is computed will be reviewed. Then in Section 2.3, the different possible operations of RAID-6 will be reviewed.

2.2 Binary Galois Field Algebra

RAID-6 uses a 8-bit Binary Galois Field, which is represented as $GF(2^8)$. This field consists of 8-bit binary values, or "bytes" of data. These bytes can be represented in hexadecimal or binary notation, just like the natural numbers. The concept of fractions and negative numbers do not exist in GF algebra, and all mathematical operations result in another value within this field. There are a number of mathematical operations that can be done in this field, and they listed below with their associated symbol:

1. Addition/Subtraction – $A \oplus B$
2. Multiplication – $A \otimes B$
3. Logarithm – $log(A)$
4. Exponentiation – $exp(A)$
5. Division – $A \oslash B$

2.2.1 GF Addition and Subtraction

GF addition can be computed as a bit-wise exclusive-or operation between the two values. Given two binary values, which are called A and B:

$$\begin{aligned} A &= \text{0x55} = 01010101 \\ B &= \text{0x0F} = 00001111 \end{aligned} \tag{2.1}$$

These values can be added as shown in Equation 2.2, which is equivalent to a bit-wise exclusive-or operation.

$$\begin{aligned} A \oplus B &= 01010101 \oplus 00001111 \\ A \oplus B &= 01011010 = \text{0x5A} \end{aligned} \tag{2.2}$$

GF subtraction is the inverse operation of GF addition. Since the exclusive-or operation is its own inverse, GF addition and GF subtraction are equivalent.

2.2.2 GF Multiplication

Every Galois Field has a specific polynomial that is used to generate the field. This generating polynomial is referred to as F. For the Galois Field that is used in this text:

$$F = x^8 + x^4 + x^3 + x^2 + 1 \rightarrow 100011101 = \text{0x11D} \tag{2.3}$$

GF multiplication is the equivalent to using an LFSR with F as the feedback. Using traditional LFSR notation, this is shown as:

$$A \otimes B \equiv (A * B)\ mod\ F \tag{2.4}$$

For example, when GF multiplying A by 2, it is done in the following manner:

$$\begin{aligned} A \otimes 2 &= \text{0x55} \otimes \text{0x02} \\ A \otimes 2 &= (\text{0x55} * \text{0x02})\ mod\ \text{0x11D} \end{aligned} \tag{2.5}$$

Using standard multiplication, multiplying by 2 is a standard shift operation, which will be represented in the following way:

$$\begin{aligned} &A \otimes 2 = (\text{0x55} << 1) \; mod \; \text{0x11D} \\ &A \otimes 2 = (\text{0x0A}A) \; mod \; \text{0x11D} \end{aligned} \tag{2.6}$$

Since the most significant bit (MSb) is a zero, the feedback will not effect the value. So the result is:

$$A \otimes 2 = \text{0xAA} \tag{2.7}$$

This is equivalent to loading A in the LFSR and shifting once. When GF multiplying A by four, simply shift the LFSR twice, which is the equivalent of multiplying by 2 twice. For example:

$$\begin{aligned} &A \otimes 4 = \text{0x55} \otimes \text{0x04} \\ &A \otimes 4 = ([(\text{0x55} << 1) \; mod \; \text{0x11D}] << 1) \; mod \; \text{0x11D} \\ &A \otimes 4 = (\text{0xAA} << 1) \; mod \; \text{0x11D} \\ &A \otimes 4 = (\text{0x154}) \; mod \; \text{0x11D} \end{aligned} \tag{2.8}$$

This time, the MSb is a one, so the feedback equation is exclusive-ored before the final output.

$$\begin{aligned} &A \otimes 4 = (\text{0x154}) \; mod \; \text{0x11D} \\ &A \otimes 4 = \text{0x154} \oplus \text{0x11D} \\ &A \otimes 4 = \text{0x048} = \text{0x48} \end{aligned} \tag{2.9}$$

This principle can be extended to multiplication by any value that is a power of 2 by simply continuing to shift and apply the feedback until the desired power has been reached. When multiplying by a value that is not a multiple of 2, the distributive property of Galois Fields can be used to compute the product. For example, if multiplying A and B:

$$\begin{aligned} &A = \text{0x55} \\ &B = \text{0x06} \end{aligned} \tag{2.10}$$

Then

$$\begin{aligned} &A \otimes B = \text{0x55} \otimes \text{0x06} \\ &A \otimes B = (\text{0x55} \otimes \text{0x02}) \oplus (\text{0x55} \otimes \text{0x04}) \end{aligned} \tag{2.11}$$

From here, the multiplication values that have already been found in the prior two examples can be substituted in.

$$\begin{aligned} &A \otimes B = (\text{0x55} \otimes \text{0x02}) \oplus (\text{0x55} \otimes \text{0x04}) \\ &A \otimes B = \text{0xAA} \oplus \text{0x48} \\ &A \otimes B = \text{0xE2} \end{aligned} \tag{2.12}$$

This principle can be generalized to multiply any two values in the field by using the distributive property on each bit of the multiplier:

$$A \otimes B = \sum_{i=0}^{n-1} (A \otimes 2^i \otimes B_i) \tag{2.13}$$

So for each bit in B that is a one, that power of 2 is multiplied by A, and then all these products are exclusive-ored together. A last example using this principle is multiplying A and B with the values:

$$\begin{aligned} &A = \text{0x55} \\ &B = \text{0x0B} \end{aligned} \tag{2.14}$$

So, A will be shifted three times, one time, and no times, then the results will be exclusive-ored together.

$$\begin{aligned} &A \otimes B = \text{0x55} \otimes \text{0x0B} \\ &A \otimes B = (\text{0x55} \otimes \text{0x08}) \oplus (\text{0x55} \otimes \text{0x02}) \oplus (\text{0x55} \otimes \text{0x01}) \\ &A \otimes B = [(\text{0x55} << 3) \; mod \; \text{0x11D}] \oplus (\text{0xAA} \oplus \text{0x55}) \end{aligned} \tag{2.15}$$

where A needs to be shifted modulo F three times. The value of A shifted two times was already calculated, so that can be substituted in:

$$\begin{aligned}
&A \otimes B = [(\text{0x48} << 1)\ mod\ \text{0x11D}] \oplus \text{0xFF}\\
&A \otimes B = [(\text{0x090})\ mod\ \text{0x11D}] \oplus \text{0xFF}\\
&A \otimes B = \text{0x90} \oplus \text{0xFF}\\
&A \otimes B = \text{0x6F}
\end{aligned} \tag{2.16}$$

2.2.3 GF Logarithm and Exponentiation

In $GF(2^8)$, GF logarithm and exponentiation are computed with 2 as the base. Exponentiation is defined as repeated multiplication of 0x02 and logarithms are defined as the inverse operation to exponentiation. For example:

$$\begin{aligned}
&\text{0x02}^{\text{0x03}} = \text{0x02} \otimes \text{0x02} \otimes \text{0x02} = \text{0x08}\\
&\text{0x02}^{\text{0xFA}} = \text{0x02} \otimes \ldots \otimes \text{0x02} = \text{0x6C}\\
&log(\text{0x08}) = log(\text{0x02}^{\text{0x03}}) = \text{0x03}\\
&log(\text{0x1D}) = log(\text{0x02}^{\text{0x08}}) = \text{0x08}
\end{aligned} \tag{2.17}$$

Both of these operations are typically found using a lookup table, as there is no simple way to calculate them. It should be noted that GF multiplication can be computed using logarithm and exponentiation, as:

$$A \otimes B = exp([log(A) + log(B)]\ mod\ \text{0xFF}) \tag{2.18}$$

2.2.4 GF Division

In $GF(2^8)$, every number has a unique multiplicative inverse. Division is defined as multiplication by the multiplicative inverse of the divisor:

$$A \oslash B = A \otimes B^{-1} \tag{2.19}$$

The complication with this definition is that finding the GF multiplicative inverse is not easily computable. Alternately, GF division can be calculated using logarithms and exponentiation:

$$A \oslash B = exp[(log(A) - log(B))\ mod\ \text{0xFF}] \tag{2.20}$$

Using logarithm and exponentiation tables, like the ones shown in Appendix B, this becomes a simpler operation. For example:

$$\begin{aligned}
&\text{0x1D} \oslash \text{0x08} = exp[(log(\text{0x1D}) - log(\text{0x08}))\ mod\ 255]\\
&\text{0x1D} \oslash \text{0x08} = exp[(8-3)\ mod\ 255]\\
&\text{0x1D} \oslash \text{0x08} = exp[5\ mod\ 255]\\
&\text{0x1D} \oslash \text{0x08} = exp[5] = exp[\text{0x05}]\\
&\text{0x1D} \oslash \text{0x08} = \text{0x20}
\end{aligned} \tag{2.21}$$

In this example, the difference of the logarithms is still between 0 and 254, so no action needs to be done to enforce the $mod\ 255$. If the dividend and divider are swapped:

$$\begin{aligned}
&\text{0x08} \oslash \text{0x1D} = exp[(log(\text{0x08}) - log(\text{0x1D}))\ mod\ 255]\\
&\text{0x08} \oslash \text{0x1D} = exp[(3-8)\ mod\ 255]\\
&\text{0x08} \oslash \text{0x1D} = exp[-5\ mod\ 255]\\
&\text{0x08} \oslash \text{0x1D} = exp[-5+255]\\
&\text{0x08} \oslash \text{0x1D} = exp[250] = exp[0xFA]\\
&\text{0x08} \oslash \text{0x1D} = 0x6C
\end{aligned} \tag{2.22}$$

In this example, the $mod\ 255$ needs to be enforced, so 255 is added to shift the result back into the range.

2.3 RAID-6 Operations

The actual method of how to recover from two storage device failures is not in the specifications for RAID-6. Many different ways have been developed and tried, but the most standard and prevalent way of recovering data is to use Reed-Solomon Encoding[6]. This is the method of RAID-6 encoding which will be discussed throughout the rest of this work.

RAID-6 has three possible operations to be designed. These are encoding of the data, detection of an erroneous data block, and correction of two know data block errors.

2.3.1 RAID-6 Encoding

RAID-6 encoding uses two rotating parity blocks to create redundancy called "P" and "Q". The first parity block (P) uses a simple exclusive-or parity. Since exclusive-or is the same as GF addition, this can also be described as:

$$P = \sum_{i=0}^{N_D-1} (D_i) \tag{2.23}$$

where D_i is the i^{th} data block in the data stripe that is being encoded and the number of data storage devices in the RAID-6 array will be called N_D. The second parity block (Q) is created in a similar manner, but uses a GF multiplication to multiply a "index" of 2^i to each data block:

$$Q = \sum_{i=0}^{N_D-1} (2^i \otimes D_i) \tag{2.24}$$

where D_i is the i^{th} data block in the data stripe that is being encoded. When the data blocks are encoded, each data block is broken up into 8-bit binary words (bytes), and the correlating word in each data block is called a word stripe. Each word stripe has a P and Q word calculated for it individually and is stored on the array in the same location of the data block on the parity drives. For example, if the first word of each data block in an array with $N_D = 3$ is:

0	1	2
0xAA	0x0F	0xFF

Then the P parity word can be calculated by simply using an exclusive-or operation.

$$\begin{aligned} P &= \sum_{i=0}^{N_D-1} (D_i) \\ P &= \text{0xAA} \oplus \text{0x0F} \oplus \text{0xFF} \\ P &= \text{0x5A} \end{aligned} \tag{2.25}$$

Similarly, the Q parity can be calculated, but the associated 2^i is multiplied before the summation.

$$\begin{aligned} Q &= \sum_{i=0}^{N_D-1} (2^i \otimes D_i) \\ Q &= (2^0 \otimes \text{0xAA}) \oplus (2^1 \otimes \text{0x0F}) \oplus (2^2 \otimes \text{0xFF}) \\ Q &= (\text{0x01} \otimes \text{0xAA}) \oplus (\text{0x02} \otimes \text{0x0F}) \oplus (\text{0x04} \otimes \text{0xFF}) \\ Q &= \text{0xAA} \oplus \text{0x1E} \oplus \text{0xDB} \\ Q &= \text{0x6F} \end{aligned} \tag{2.26}$$

So the encoded word stripe that would be written to the disk would be:

0	1	2	P	Q
0xAA	0x0F	0xFF	0x5A	0x6F

2.3.2 RAID-6 Double Error Correction

If there is an external method of detecting errors on the data blocks in a stripe, then two blocks may be recovered with no loss of data by using the RAID-6 parity blocks. There are various ways to detect errors, either from the file system detecting errors, or the hardware detecting storage device failure, or possibly even embedding a CRC or error detecting code in the data before the RAID-6 encoding. When these errors are detected, there are 4 possible scenarios which would have to be dealt with to fix all possible permutations of where two errors could be:

1. The P and Q blocks
2. The Q block and a Data block
3. The P block and a Data block
4. Two Data blocks

The P and Q blocks

If the two blocks that have been corrupted are the two parity blocks, then these can be regenerated from the original data using the same method as they are normally generated.

The Q block and a Data block

If one of the errors is in a data block and the other is in the Q block, the corrupted data block can be found using the P block and traditional normal exclusive-or parity. This is done in the following way.

$$\begin{aligned}
&P = D_0 \oplus \ldots \oplus D_{L-1} \oplus D_L \oplus D_{L+1} \oplus \ldots \oplus D_n \\
&0 = P \oplus D_0 \oplus \ldots \oplus D_{L-1} \oplus D_L \oplus D_{L+1} \oplus \ldots \oplus D_n \\
&D_L = P \oplus D_0 \oplus \ldots \oplus D_{L-1} \oplus D_{L+1} \oplus \ldots \oplus D_n
\end{aligned} \tag{2.27}$$

Alternately, this can be found by recomputing the P parity with the erroneous data block replaced with a zero. This recomputed parity is called P'. If this value is GF subtracted from the original P, D_L is the result.

$$\begin{aligned}
&P = D_0 \oplus \ldots \oplus D_{L-1} \oplus D_L \oplus D_{L+1} \oplus \ldots \oplus D_n \\
&P' = D_0 \oplus \oplus D_{L-1} \oplus 0 \oplus D_{L+1} \oplus \ldots \oplus D_n \\
&D_L = P \oplus P'
\end{aligned} \tag{2.28}$$

After the data is repaired, the Q block can be regenerated normally. As an example, suppose that there are two known errors in the data stripe that was just encoded in Section 2.3.1, which are marked as 0xXX.

0	1	2	P	Q
0xAA	0xXX	0xFF	0x5A	0xXX

Using Equation 2.27, the erroneous data can be found.

$$\begin{aligned}
&D_1 = P \oplus D_0 \oplus D_2 \\
&D_1 = \text{0x5A} \oplus \text{0xAA} \oplus \text{0xFF} \\
&D_1 = \text{0x0F}
\end{aligned} \tag{2.29}$$

Using the second method, P' is calculated first.

$$\begin{aligned}
&P' = D_0 \oplus D_2 \\
&P' = \text{0xAA} \oplus \text{0xFF} \\
&P' = \text{0x55}
\end{aligned} \tag{2.30}$$

Then this is substituted into Equation 2.28.

$$\begin{aligned}
&D_2 = P \oplus P' \\
&D_2 = \text{0x5A} \oplus \text{0x55} \\
&D_2 = \text{0x0F}
\end{aligned} \tag{2.31}$$

With both methods, the same correct data is found. After this, the Q block can be regenerated, just as it was originally generated in Equation 2.26.

The P block and a Data block

If there is one error in a data block and an error in the P block, the data is found from the Q block by recomputing the Q block (Q') with the bad data block (D_L) as all zeros. From this, the original data can be found in the following way:

$$\begin{aligned}
&Q = 2^0 \otimes D_0 \oplus \ldots \oplus 2^{L-1} \otimes D_{L-1} \oplus 2^L \otimes D_L \oplus 2^{L+1} \otimes D_{L+1} \oplus \ldots \oplus 2^n \otimes D_n \\
&Q' = 2^0 \otimes D_0 \oplus \ldots \oplus 2^{L-1} \otimes D_{L-1} \oplus 2^L \otimes 0 \oplus 2^{L+1} \otimes D_{L+1} \oplus \ldots \oplus 2^n \otimes D_n \\
&Q \oplus Q' = 2^L \otimes D_L \\
&D_L = (Q \oplus Q') \oslash 2^L
\end{aligned} \tag{2.32}$$

Once the data block is found, the P block can be regenerated as usual. As an example, suppose that there are two known errors in the data stripe that was encoded in Section 2.3.1, which are marked as 0xXX.

0	1	2	P	Q
0xAA	0x0F	0xXX	0xXX	0x6F

To find the missing data, first Q' is calculated.

$$\begin{aligned}
&Q = \sum_{i=0}^{N_D-1} (2^i \otimes D_i) \\
&Q' = (2^0 \otimes \text{0xAA}) \oplus (2^1 \otimes \text{0x0F}) \\
&Q' = (\text{0x01} \otimes \text{0xAA}) \oplus (\text{0x02} \otimes \text{0x0F}) \\
&Q' = \text{0xAA} \oplus \text{0x1E} \\
&Q' = \text{0xB4}
\end{aligned} \tag{2.33}$$

Then, using Equation 2.32, the data can be found.

$$\begin{aligned}
&D_2 = (\text{0x6F} \oplus \text{0xB4}) \oslash 2^2 \\
&D_2 = \text{0xDB} \oslash \text{0x04} \\
&D_2 = exp([log(\text{0xDB}) - log(\text{0x04})] \, mod \, 255) \\
&D_2 = exp([\text{0xB1} - \text{0x02}] \, mod \, 255) \\
&D_2 = exp(\text{0xAF} \, mod \, 255) \\
&D_2 = exp(\text{0xAF}) \\
&D_2 = \text{0xFF}
\end{aligned} \tag{2.34}$$

Now that the data is corrected, the P parity can be regenerated, just as it was originally generated in Equation 2.25.

Two Data blocks

The last and most complicated case is when both of the errors are data blocks. The two errors are at known locations, defined as K and L, and it can be assumed (without loss of generality) that $K < L$. Given this, we compute new parity blocks (P' and Q') with the erroneous blocks as zeros. From this we can isolate the original data by using GF subtraction.

$$\begin{aligned}
&P = D_0 \oplus \ldots \oplus D_{K-1} \oplus D_K \oplus D_{K+1} \oplus \ldots \oplus D_{L-1} \oplus D_L \oplus D_{L+1} \oplus \ldots \oplus D_n \\
&P' = D_0 \oplus \ldots \oplus D_{K-1} \oplus 0 \oplus D_{K+1} \oplus \ldots \oplus D_{L-1} \oplus 0 \oplus D_{L+1} \oplus \ldots \oplus D_n \\
&P = P' \oplus D_K \oplus D_L
\end{aligned} \tag{2.35}$$

$$\begin{aligned}
&Q = 2^0 \otimes D_0 \oplus \ldots \oplus 2^{K-1} \otimes D_{K-1} \oplus 2^K \otimes D_K \oplus 2^{K+1} \otimes D_{K+1} \oplus \ldots \\
&\qquad \oplus 2^{L-1} \otimes D_{L-1} \oplus 2^L \otimes D_L \oplus 2^{L+1} \otimes D_{L+1} \oplus \ldots \oplus 2^n \otimes D_n \\
&Q' = 2^0 \otimes D_0 \oplus \ldots \oplus 2^{K-1} \otimes D_{K-1} \oplus 2^K \otimes 0 \oplus 2^{K+1} \otimes D_{K+1} \oplus \ldots \\
&\qquad \oplus 2^{L-1} \otimes D_{L-1} \oplus 2^L \otimes 0 \oplus 2^{L+1} \otimes D_{L+1} \oplus \ldots \oplus 2^n \otimes D_n \\
&Q = Q' \oplus 2^K \otimes D_K \oplus 2^L \otimes D_L
\end{aligned} \tag{2.36}$$

From here, we have a simple set of linear equations with two unknowns. Equation 2.35 is solved for D_L and Equation 2.36 is solved for D_K.

$$\begin{aligned}
&P = P' \oplus D_K \oplus D_L \\
&D_L = P \oplus P' \oplus D_K
\end{aligned} \tag{2.37}$$

$$\begin{aligned} &Q = Q' \oplus 2^{-K} \otimes D_K \oplus 2^L \otimes D_L \\ &D_K = 2^{-K} \otimes (Q \oplus Q') \oplus 2^{L-K} \otimes D_L \end{aligned} \tag{2.38}$$

Then, substituting Equation 2.38 into Equation 2.37 gives:

$$\begin{aligned} &D_K = 2^{-K} \otimes (Q \oplus Q') \oplus 2^{L-K} \otimes (P \oplus P' \oplus D_K) \\ &D_K \otimes (2^{L-K} \oplus 1) = 2^{-K} \otimes (Q \oplus Q') \oplus 2^{L-K} \otimes (P \oplus P') \\ &D_K = (2^{-K} \otimes (Q \oplus Q') \oplus 2^{L-K} \otimes (P \oplus P')) \oslash (2^{L-K} \oplus 1) \end{aligned} \tag{2.39}$$

This gives the solution for D_K. Since $K < L$:

$$2^{L-K} \oplus 1 > 1 \tag{2.40}$$

So there will never be the situation where dividing by 0 will need to be accounted for. Once the correct value of D_K is found, it can be substituted into Equation 2.37 to find D_L. As an example, suppose that there are two known errors in the data stripe that was encoded in Section 2.3.1, which are marked as 0xXX.

0	1	2	P	Q
0xAA	0xXX	0xXX	0x5A	0x6F

For these two missing data words, $K = 1$ and $L = 2$. First, P' and Q' must be calculated.

$$\begin{aligned} &P' = D_0 \\ &P' = \text{0xAA} \end{aligned} \tag{2.41}$$

$$\begin{aligned} &Q' = 2^0 \otimes D_0 \\ &Q' = \text{0xAA} \end{aligned} \tag{2.42}$$

Now D_1 can be found by using Equation 2.39.

$$\begin{aligned} &D_K = (2^{-K} \otimes (Q \oplus Q') \oplus 2^{L-K} \otimes (P \oplus P')) \oslash (2^{L-K} \oplus 1) \\ &D_1 = (2^{-1} \otimes (\text{0x6F} \oplus \text{0xAA}) \oplus 2^{2-1} \otimes (\text{0x5A} \oplus \text{0xAA})) \oslash (2^{2-1} \oplus 1) \\ &D_1 = (2^{-1} \otimes \text{0xC5} \oplus 2 \otimes \text{0xF0}) \oslash (2 \oplus 1) \end{aligned} \tag{2.43}$$

Since "$2^{-1} \otimes X$" is equivalent to "$X \oslash 2$":

$$\begin{aligned} &D_1 = (\text{0xC5} \oslash 2 \oplus 2 \otimes \text{0xF0}) \oslash (2 \oplus 1) \\ &D_1 = (\text{0xEC} \oplus \text{0xFD}) \oslash \text{0x03} \\ &D_1 = \text{0x11} \oslash \text{0x03} \\ &D_1 = exp([log(\text{0x11}) - log(\text{0x03})] \; mod \; \text{0x11D}) \\ &D_1 = exp([\text{0x64} - \text{0x19}] \; mod \; \text{0x11D}) \\ &D_1 = exp(\text{0x4B} \; mod \; \text{0x11D}) \\ &D_1 = exp(\text{0x4B}) \\ &D_1 = \text{0x0F} \end{aligned} \tag{2.44}$$

Now that D_1 has been found, D_2 can be found by using Equation 2.37.

$$\begin{aligned} &D_L = P \oplus P' \oplus D_K \\ &D_2 = \text{0x5A} \oplus \text{0xAA} \oplus \text{0x0F} \\ &D_2 = \text{0xFF} \end{aligned} \tag{2.45}$$

Thus both unknown data blocks are found.

2.3.3 RAID-6 Error Detection

RAID-6 can sense a single error in a word stripe of data by using the embedded parities alone[7]. This allows for many errors in the data to be recognized and corrected, as long as there is only one error per word stripe. For typical applications, the probability of having more than one word corrupted per word stripe is very low. If there is more than one word corruption per word stripe, there is a chance of it being detectable, but there is also a chance of the multiple errors being misdiagnosed as a single bit error. In this last case, using this method of detecting errors would introduce an extra error to the word stripe by "correcting" the wrong problem.

If there are multiple errors in the word stripe, this will always look like a single error at a data location. Once this method is used, the location indicated will be an indeterminate value in the range allowed by the word size. If this value is outside the allowable range of the data, then this can be recognized as multiple errors. Otherwise, it is misdiagnosed as a single bit error. The probability of this misdiagnosis occurring when there are multiple errors on a stripe is:

$$P(\text{undetected error}) = \frac{N_D}{2^n - 1} \tag{2.46}$$

where N_D is the number of data drives in the system. For example, if an 8-bit word system with 6 drives has multiple errors on a word stripe:

$$\begin{aligned}&P(\text{undetected error}) = \tfrac{N_D}{2^n-1}\\&P(\text{undetected error}) = \tfrac{6}{255}\\&P(\text{undetected error}) = 0.02353\end{aligned} \tag{2.47}$$

So, in this example, there is a 2.35% chance of multiple errors going undetected.

Assume that a single word is corrupted after the stripe has been encoded. This corrupted word is at some unknown location L, and the data word D_L has been changed to some value X. This can be found in the following way[7]. First, the two parity values for the corrupted data would be calculated, just as shown in the former section. These new parities will be called P' and Q'. If the value of these are different than the P and Q values that were originally encoded into the stripe, then this is proof that there is an error on the stripe. Then, the error must be isolated. This is done using GF subtraction (which is the same as GF addition) on both the P and Q parity equations.

$$\begin{aligned}&P = D_0 \oplus \ldots \oplus D_{L-1} \oplus D_L \oplus D_{L+1} \oplus \ldots \oplus D_n\\&P' = D_0 \oplus \ldots \oplus D_{L-1} \oplus X \oplus D_{L+1} \oplus \ldots \oplus D_n\\&P \oplus P' = D_L \oplus X\end{aligned} \tag{2.48}$$

$$\begin{aligned}&Q = 2^0 \otimes D_0 \oplus \ldots \oplus 2^{L-1} \otimes D_{L-1} \oplus 2^L \otimes D_L \oplus 2^{L+1} \otimes D_{L+1} \oplus \ldots \oplus 2^n \otimes D_n\\&Q' = 2^0 \otimes D_0 \oplus \ldots \oplus 2^{L-1} \otimes D_{L-1} \oplus 2^L \otimes X \oplus 2^{L+1} \otimes D_{L+1} \oplus \ldots \oplus 2^n \otimes D_n\\&Q \oplus Q' = 2^L \otimes D_L \oplus 2^L \otimes X = 2^L \otimes (D_L \oplus X)\end{aligned} \tag{2.49}$$

Then the value of L can be found by dividing Equation 2.49 by Equation 2.48 and then taking the logarithm:

$$\begin{aligned}&(Q \oplus Q') \oslash (P \oplus P') = 2^L\\&L = log((Q \oplus Q') \oslash (P \oplus P'))\end{aligned} \tag{2.50}$$

Once the value of L has been found, a corrected value can be found using exclusive-or parity correction using the original P value, just as shown in Equation 2.27.

If the word in error is not in the data, but is in either the P or the Q block, then it is trivial to show that this parity value will not pass the check against P' or Q', and the other parity will pass. Fixing this error is as simple as replacing the erroneous parity word with the one that has just been calculated (either P' or Q' depending on which is in error).

Taking the data stripe that we just encoded in Section 2.3.1, what if D_2 got corrupted in the following way:

0	1	2	P	Q
0xAA	0x0F	0xEF	0x5A	0x6F

To detect this error, first the P' and Q' parity must be computed just like was described in Section 2.3.1.

$$\begin{aligned} P &= \sum_{i=0}^{N_D-1} (D_i) \\ P' &= \text{0xAA} \oplus \text{0x0F} \oplus \text{0xEF} \\ P' &= \text{0xDA} \end{aligned} \tag{2.51}$$

$$\begin{aligned} Q &= \sum_{i=0}^{N_D-1} (2^i \otimes D_i) \\ Q' &= (2^0 \otimes \text{0xAA}) \oplus (2^1 \otimes \text{0x0F}) \oplus (2^2 \otimes \text{0xFF}) \\ Q' &= (\text{0x01} \otimes \text{0xAA}) \oplus (\text{0x02} \otimes \text{0x0F}) \oplus (\text{0x04} \otimes \text{0xEF}) \\ Q' &= \text{0xAA} \oplus \text{0x1E} \oplus \text{0xE1} \\ Q' &= \text{0x55} \end{aligned} \tag{2.52}$$

Since P' and Q' are not equal to the P and Q that were stored with the stripe, the error is now detected. Now to locate this error, Equation 2.50 is used.

$$\begin{aligned} L &= log((Q \oplus Q') \oslash (P \oplus P')) \\ L &= log((\text{0x6F} \oplus \text{0x55}) \oslash (\text{0x5A} \oplus \text{0xDA})) \\ L &= log(\text{0x3A} \oslash \text{0x80}) \\ L &= log(exp([log(\text{0x3A}) - log(\text{0x80})] \; mod \; 255)) \end{aligned} \tag{2.53}$$

After substitution in for the division, it is easy to see that the outer logarithm and exponentiation cancel each other out.

$$\begin{aligned} L &= log(exp([log(\text{0x3A}) - log(\text{0x80})] \; mod \; 255)) \\ L &= [log(\text{0x3A}) - log(\text{0x80})] \; mod \; 255 \\ L &= [9 - 7] \; mod \; 255 \\ L &= [2] \; mod \; 255 \\ L &= 2 \end{aligned} \tag{2.54}$$

So the location of the erroneous data was correctly found to be D_2. Now the original data can be recovered using Equation 2.27.

$$\begin{aligned} D_2 &= P \oplus D_0 \oplus D_1 \\ D_2 &= \text{0x5A} \oplus \text{0xAA} \oplus \text{0x0F} \\ D_2 &= \text{0xFF} \end{aligned} \tag{2.55}$$

2.4 Prior Work In Hardware and Software Implementations of GF Algebra

Most of the work done in hardware and software implementations of GF Algebra to date has been in the area of Galois Field exponentiation algorithms and factoring of large numbers, as these have applications in public key cryptography. Berlekamp developed an algorithm for doing polynomial factorization over a Galois Field using matrix reductions and GCD calculations[8]. The concept of normal basis representation of GF elements was developed by Massey and Omura[9]. This allows squaring to be done as a cyclic shift. Considerable work has also been done in quickly finding the the multiplicative inverse with $GF(2^n)$ using normal basis[10][11]. These methods use the fact that exponentiation is a simple cyclic shift in normal basis and are not nearly as efficient in polynomial basis where exponentiation is much more difficult. Normal basis multiplication is more difficult to do than in polynomial basis, so this basis is typically only used when exponentiation is a primary operation, as when doing elliptical-based public-key algorithms[12][13]. Because multiplication is essential to RAID-6 encoding and decoding, normal basis is outside the scope of this work.

Using polynomial basis, Wu proposed complexity bounds on doing GF multiplication based on using standard methods and then reducing the results modulo the generating polynomial[14]. Unfortunately, he proposed no methods on how to obtain these bounds. Anvin showed ways of implementing basic GF arithmetic, with a focus on software implementations[7]. Paar showed methods of developing a bit-parallel combinational multiplier[15]. The concepts of both Anvin and Paar are very similar, but Anvin's focus on software implementations leads the implementations in two different directions. Both of these implementations are implemented in hardware and are compared in Section 3.3 . Guajardo et al. showed that Itoh and Tsujii's method of quickly finding the multiplicative inverse in normal basis can also be applied to polynomial basis, assuming the use of polynomial multiplication and exponentiation[16]. Various methods of finding the multiplicative inverse using the extended Euclidean Algorithm have also been developed[17][18]. Each of these methods will be discussed in the following chapter.

Chapter 3

Efficient Implementation of a GF Algebra FPGA library

The purpose of this work was to create a fast, efficient implementation of RAID-6. Since any RAID-6 implementation strongly relies on the underlying GF Algebra, first an efficient implementation of GF Algebra must be designed. As discussed in Section 2.2, GF Algebra has a number of operations that are defined, as listed below. Multiplicative Inversion is also included in this list, as it will be required for GF Division using some of the methods described here.

1. Addition/Subtraction – $A \oplus B$
2. Logarithm – $log(A)$
3. Exponentiation – $exp(A)$
4. Multiplication – $A \otimes B$
5. Division – $A \oslash B$
6. Inversion – A^{-1}

This work compiles all accepted implementations for each of these operations in a VHDL library of parts and evaluates each for use in an FPGA (specifically, the Virtex-4 FX60). This work also proposes, implements and evaluates several improvements by using the simplifications that constants allow. To allow this library of parts to be more flexible and readable, a number of constants and types were created in the package:

```
constant W: integer := 8;
constant F: std_logic_vector(W downto 0) := "100011101";
subtype word is std_logic_vector(W-1 downto 0);
```

The constant W is the order of the function F used to generate the Galois Field. By definition, this value is also the number of bits in any binary number in the field. The subtype "word" is the type used to represent these numbers. The constant F represents the function F. Other constants and types are also defined in the package. These will be described when the function that uses them are introduced. As shown above, the IP Core Library was evaluated using 8-bit words, or bytes, as the basis of calculations and uses the same generating function, F, as used in the examples above.

3.1 GF Addition/Subtraction

GF addition and subtraction, as discussed in the last chapter, may be computed by using a bit-wise exclusive-or operation. In the Virtex-4, the fabric is made up of 4-input lookup tables. Because of this, the exclusive-or operation can be implemented in a lookup table. No function was designed to do this operation, as VHDL already includes the "xor" operation that computes the bit-wise exclusive-or.

3.2 GF Logarithm and Exponentiation

GF logarithm and exponentiation operations are most simply computed by creating lookup tables of these operations. For this component, another custom VHDL type was created to represent these lookup tables:

```
type LUT is array ((2**W)-1 downto 0) of word;
```

Actually populating these lookup tables can be done in multiple ways. The first option is to generate the lookup tables (by hand or with some external program) and insert them into the VHDL code as an array of values. A more flexible and straightforward method would be to generate the tables from within the VHDL code. The exponentiation table is the easier of the two tables to generate, while the logarithm table is much more difficult to generate.

Since exponentiation and logarithm are inverse operations, the exponentiation table can be generated first and then used to generate the logarithm table by inverting the table. VHDL functions that accomplish this are shown in Appendix C in the functions called `Calc_GFILOG` and `Calc_GFLOG`. In this code, exponentiation is referred to as "ILog" which stands for "Inverse Logarithm".

When these functions are synthesized, each implements a lookup table that consumes 64 slices. FPGAs in general, and Virtex-4 FPGAs in particular, typically include hardcore RAM blocks that are dedicated to storing data called "Block RAM" and can be used as either RAM or ROM memory. For the Virtex-4, these block RAMs may be dual ported, which allows two simultaneous lookups in the same clock cycle from the same table. It would be desirable to use these blocks for these lookup tables, which are dedicated for this purpose. There are several disadvantages to this method. First, these block RAMs are very useful, so they can become a tightly constrained resource in a system if there are multiple components competing for them. The second is that these block RAMs are synchronous devices, so they require a clock cycle to calculate the results. The functions created are not synchronous, so a synchronous system is created to use the lookup tables created above and load them each into a block RAM. This is accomplished by giving the lookup table certain compiler attributes. The VHDL parts that implement this are shown in Appendix D in the entities called `GFILog_BRAM` and `GFLog_BRAM`. "BRAM" is used to refer to a "block RAM".

When these components are synthesized, they are implemented as 1 RAM block each. They have a propagation delay of 2.46 ns. This is shown in Table 3.1.

	Slices	BRAMs	Prop Delay (ns)	cyc/calc	latency	max Freq (MHz)	calc/sec (MHz)
Logarithm	0	1	2.46	1	1	406.504065	406.504065
Exponentiation	0	1	2.46	1	1	406.504065	406.504065

Table 3.1: Logarithm and exponentiation Table Lookup Summary

3.3 GF Multiplication

There are several accepted hardware implementations of GF multiplication in the literature today. These are:

1. Logarithm Table Method [19]
2. LFSR Method [7]
3. Bit-wise Parallel Method [20]

Also, GF Multiplication by a constant allows simplification to the hardware, which will be discussed in Section 3.5.

3.3.1 Logarithm Table Method

One method of calculating the GF product is the logarithm table based method. This uses Equation 2.18 shown in the last chapter. Using this method, the logarithm of each input word is found and are summed modulo $2^n - 1$. The exponentiation of this sum is the GF product of the two input values. This process is shown in Figure 3.1.

Since the RAM blocks each require a clock cycle delay, this system is very naturally pipelined with 3 clock cycle latency. To generalize and simplify this block, a constant call "max" is defined as follows:

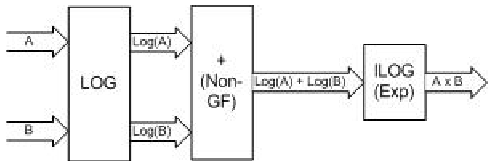

Figure 3.1: Table-Based GF Multiplier

```
constant    max       : integer := (2**W)-1;
```

This constant is used to do addition modulo $2^n - 1$. The VHDL entity to do this is called `GFM_BRAM` and is shown in Appendix D.

When synthesized, this entity uses 2 RAM blocks and 25 slices. This block has a 3-stage pipeline, so the result has a 3 cycle latency. But once the pipeline is loaded, a calculation is finished each clock cycle. The maximum propagation delay for a step this pipeline is 7.576 ns, which gives this block a maximum clock frequency of 132 MHz.

If we assume that we already know the logarithm of one of the inputs, which occurs frequently in the RAID-6 calculations, a slightly modified version of this multiplier can be created that can compute two multiplications simultaneously. This takes advantage of all of the possible bandwidth of the dual-ported RAM blocks in the Virtex-4. This component is almost the same as the standard GF multiplication component above, but with the addition modulo $2^n - 1$ duplicated. This VHDL component is called `GFM_BRAM_double` and is also shown in Appendix D.

This component is almost the same as duplicating two of the `GFM_BRAM` components. The main difference is that rather then two arbitrary variables being passed into the component, you now have an arbitrary variable, and the log of a value. This allows two multiplication operations to occur using only two RAM blocks and 49 slices. As before, the multiplications have a latency of 3 cycles, but a calculation is completed each cycle once the pipeline is loaded. The propagation delay is 7.514 ns, which gives a maximum clock frequency of 133 MHz. This is only two MHz slower then the single arbitrary multiplication version.

3.3.2 LFSR Method

Another method discussed earlier is the use of a LFSR to compute the product. When translated into hardware, this multiplier would successively multiply one input by 0x02 using a LFSR, and each time the value would be masked by the associated bit of the other input. Then the masked value would be added to the running total. After 8 clock cycles, the running total would be the product of the two numbers. A block diagram of this method is shown in Figure 3.2.

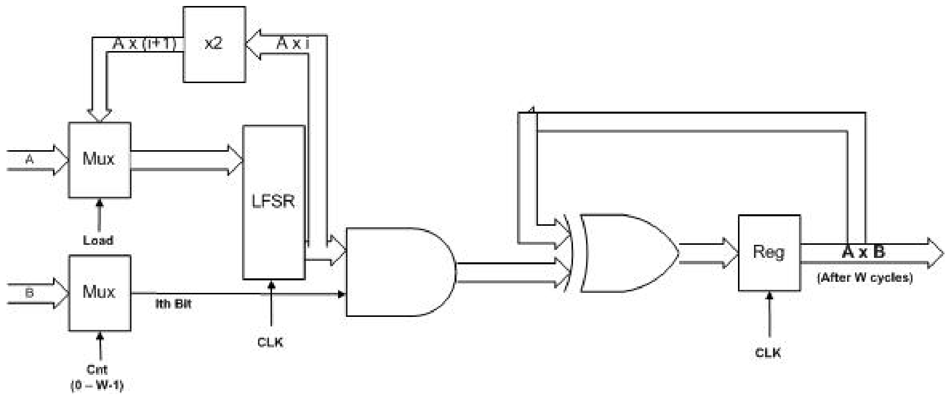

Figure 3.2: LFSR GF Multiplier

Because the LFSR is needed for each multiplication, the LFSR multiplier requires a full 8 clock cycles to compute one product. The VHDL entity which uses this method is called `GFM_LFSR` and is also shown in Appendix D.

The `load` signal is required for the multiplier to know when to start a new computation. The `done` signal is not strictly required since the timing for this component is consistently 8 cycles after it is loaded. But the `done` signal is already generated in the module and can simplify other components by not requiring them to implement their own 8 cycle counter. When synthesized, the LFSR multiplier requires only 14 slices and the propagation delay through the part is only 3.121 ns, which would allow this part to be clocked at up to 320 MHz. This small size and speed is offset by the fact that this component requires 8 clock cycles to complete a computation.

3.3.3 Bit-wise Parallel Multiplier

A third option on GF multiplication is an extension of the LFSR method, which was first presented by Mastrovito[21]. In the LFSR method, when multiplying by 2, the LFSR shift is the equivalent of applying the following transformations to each bit of the input to find the product (assuming the F function given above):

$$\begin{aligned}
&2B_7 \leftarrow B_6 \\
&2B_6 \leftarrow B_5 \\
&2B_5 \leftarrow B_4 \\
&2B_4 \leftarrow B_3 \oplus B_7 \\
&2B_3 \leftarrow B_2 \oplus B_7 \\
&2B_2 \leftarrow B_1 \oplus B_7 \\
&2B_1 \leftarrow B_0 \\
&2B_0 \leftarrow B_7
\end{aligned} \tag{3.1}$$

Where B_i is the i^{th} bit of the input, and $2B_i$ is the i^{th} bit of the the input GF multiplied by 2. This is very simple combinational logic. To multiply by 4, the same transform is done twice, which becomes:

$$\begin{aligned}
&4B_7 \leftarrow B_5 \\
&4B_6 \leftarrow B_4 \\
&4B_5 \leftarrow B_3 \oplus B_7 \\
&4B_4 \leftarrow B_2 \oplus B_6 \oplus B_7 \\
&4B_3 \leftarrow B_1 \oplus B_6 \oplus B_7 \\
&4B_2 \leftarrow B_0 \oplus B_6 \\
&4B_1 \leftarrow B_7 \\
&4B_0 \leftarrow B_6
\end{aligned} \tag{3.2}$$

This multiplying can be repeated to find how to multiply by any power of 2. For multiplying two 8-bit numbers, multiples up to 2^{n-1} or 128 are needed, which is the equivalent of shifting the LFSR 7 times, which is all that is required to create a general-purpose multiply. In many cases these transfer functions can be simplified because:

$$A \oplus A = 0 \tag{3.3}$$

After simplification, the table of each multiplication by a power of 2 is shown in Table 3.2.

All of these transfer functions are combinational logic, and since no function contains more than 4 inputs, each can be calculated using one slice. The output of each multiplication is then bit-wise ANDed with the associated bit of A in parallel, and then the output of each of these can be routed into eight 8-input exclusive-or gates to calculate the final product of A and B. Using this method reduces the entire process of multiplication down to combinational logic. Because of this, no flip-flops or control logic are necessary, since the 8 bits of processing are done in parallel and are completed in a single cycle. In theory, this method requires 41 exclusive-or gates that are 4-input or less (so each can be calculated with one 4-input lookup table), eight 8-input AND gates, and eight 8-input exclusive-or gates. The block diagram of this method is shown in Figure 3.3.

This bit-parallel component can be implemented directly into VHDL code by calculating the transfer functions required for a particular generating function, and then transcribe these as VHDL code. This was implemented in the `GFMe` entity shown in Appendix C. But a much less error-prone and more flexible method of doing this is to have the compiler compute the transfer equations at compile time. Computing these functions in the VHDL code is done by doing the equivalent of the LFSR method in a function using a for loop. The compiler will optimize the loop down to the appropriate exclusive-or equations at compile time. For simplicity, this is coded as three separate functions. The first function implements the GF Multiply by 2 functionality described in Equation 3.1. The VHDL function for this is called `GFM2w` and is shown in Appendix C. A block diagram of this function is shown in Figure 3.4.

B	$B \otimes 2$	$B \otimes 4$	$B \otimes 8$	$B \otimes 16$	$B \otimes 32$	$B \otimes 64$	$B \otimes 128$
B_7	B_6	B_5	B_4	$B_3 \oplus B_7$	$B_2 \oplus B_6 \oplus B_7$	$B_1 \oplus B_5 \oplus B_6 \oplus B_7$	$B_0 \oplus B_4 \oplus B_5 \oplus B_6$
B_6	B_5	B_4	$B_3 \oplus B_7$	$B_2 \oplus B_6 \oplus B_7$	$B_1 \oplus B_5 \oplus B_6 \oplus B_7$	$B_0 \oplus B_4 \oplus B_5 \oplus B_6$	$B_3 \oplus B_4 \oplus B_5$
B_5	B_4	$B_3 \oplus B_7$	$B_2 \oplus B_6 \oplus B_7$	$B_1 \oplus B_5 \oplus B_6 \oplus B_7$	$B_0 \oplus B_4 \oplus B_5 \oplus B_6$	$B_3 \oplus B_4 \oplus B_5$	$B_2 \oplus B_3 \oplus B_4$
B_4	$B_3 \oplus B_7$	$B_2 \oplus B_6 \oplus B_7$	$B_1 \oplus B_5 \oplus B_6 \oplus B_7$	$B_0 \oplus B_4 \oplus B_5 \oplus B_6$	$B_3 \oplus B_4 \oplus B_5$	$B_2 \oplus B_3 \oplus B_4$	$B_1 \oplus B_2 \oplus B_3 \oplus B_7$
B_3	$B_2 \oplus B_7$	$B_1 \oplus B_6 \oplus B_7$	$B_0 \oplus B_5 \oplus B_6$	$B_4 \oplus B_5 \oplus B_7$	$B_3 \oplus B_4 \oplus B_6 \oplus B_7$	$B_2 \oplus B_3 \oplus B_5 \oplus B_6$	$B_1 \oplus B_2 \oplus B_4 \oplus B_5$
B_2	$B_1 \oplus B_7$	$B_0 \oplus B_6$	$B_5 \oplus B_7$	$B_4 \oplus B_6$	$B_3 \oplus B_5 \oplus B_7$	$B_2 \oplus B_4 \oplus B_6 \oplus B_7$	$B_1 \oplus B_3 \oplus B_5 \oplus B_6$
B_1	B_0	B_7	B_6	B_5	B_4	$B_3 \oplus B_7$	$B_2 \oplus B_6 \oplus B_7$
B_0	B_7	B_6	B_5	B_4	$B_3 \oplus B_7$	$B_2 \oplus B_6 \oplus B_7$	$B_1 \oplus B_5 \oplus B_6 \oplus B_7$

Table 3.2: Transfer Functions for multiplying by 2^x for $x = 0 - 7$

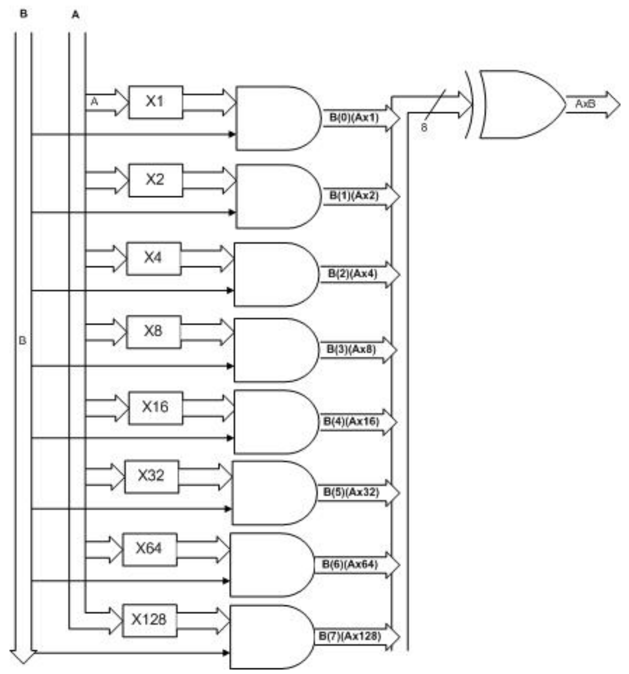

Figure 3.3: Bit Parallel Multiplier

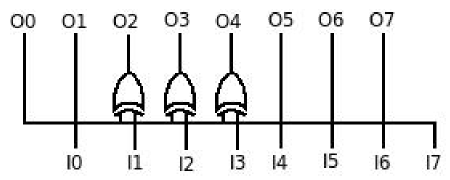

Figure 3.4: Combinational $I \otimes 2$

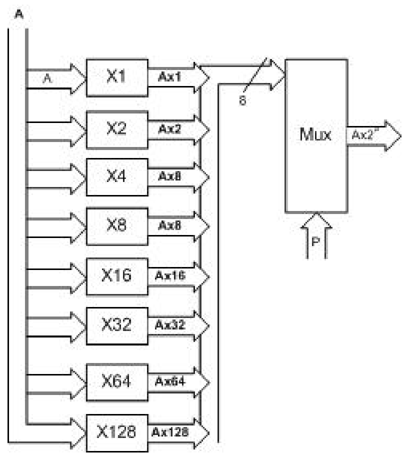

Figure 3.5: Combinational $I \otimes 2^p$

This function is then used in a function to multiply by 2 successively to calculate the input value multiplied by 2^p. The VHDL function for this is called `GFM2p` and is shown in Appendix C. A block diagram of this function is shown in Figure 3.5.

When a constant value for `p` is passed into this function, the compiler unrolls and simplifies the calculations in the `for` loop down to the same equations as found mathematically above. This function is then used to compute the product by using another loop to pass it the required constants and exclusive-or the results together. The function to do this is called `GFMc` and is shown in Appendix C. The block diagram for this is the same as Figure 3.3 shown above.

The `for` loop in this function is also unwrapped at compile time, which reduces this function down to the same transfer functions as would be if the function was hardcoded and then manually exclusive-ored. This also allows much greater flexibility. If the Galois Field needed to change size, or a different generating polynomial F was desirable, then this method would allow everything to be changed by simply changing the global constants. The hardcoded version would need to be completely rewritten with new transfer functions.

When synthesized, this function creates combinational logic that uses 38 slices and has a propagation delay of 6.515 ns. This means that this logic can be run at a maximum of 153.5 MHz. Since these functions simplify down to combinational logic, the calculations are completed in a single clock cycle.

GF Multiplication Summary

A summary of the size and timing of the multipliers presented in this section is shown in Table 3.3. These results will be discussed in detail in Section 3.7.

	Slices	BRAMs	Prop Delay (ns)	cyc/calc	latency	max Freq (MHz)	calc/sec (MHz)
BRAM	25	2	7.576	1	3	131.9957761	131.9957761
BRAM(2x)	49	2	7.514	1	3	133.0849082	133.0849082
LFSR	14	0	3.121	8	8	320.410125	40.051265625
Bit-Parallel	38	0	6.515	1	1	153.4919417	153.4919417

Table 3.3: Galois Field Multiplier Summary

3.4 GF Division and Inversion

Division can be computed in one of two ways.

1. Logarithm Table Method
2. Multiplication by Inverse

The second method requires the creation of an inversion component, which will be discussed here. Also, GF Division by a constant allows simplification of the hardware, which will be discussed in Section 3.6.

3.4.1 Table Method

Using logarithm tables, the quotient is computed as it was in the examples in Section 2.2.4. By taking the log of the two input values, computing the difference modulo $2^n - 1$, and then the exponentiation of this difference is the quotient. The block diagram of this component is shown in Figure 3.6.

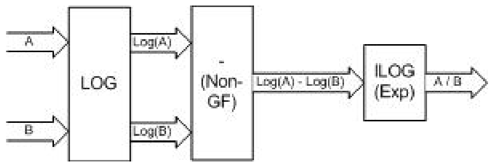

Figure 3.6: Table-Based GF Divider

This component is almost identical to the Table-based multiplier component, except that the division uses subtraction rather than addition. Once again, the constant "max" is used to ease subtraction modulo $2^n - 1$. The entity to implement this component is called `GFD_BRAM` and is shown in Appendix D.

When synthesized, this component uses 2 RAM blocks and 19 slices. The max propagation delay through a pipeline stage is 6.628 ns, which allows this component to be clocked at 150 MHz.

If we assume that we already know the logarithm of the divisor, which occurs sometimes in the RAID-6 calculations, a slightly modified version of this divider can be created that can compute two divisions simultaneously. This takes advantage of all of the possible bandwidth of the RAM blocks.

This component is almost the same as the standard table-based division component, but with the subtraction modulo $2^n - 1$ duplicated. This VHDL component is called `GFD_BRAM_double` and is shown in Appendix D.

This component still only uses 2 RAM blocks, but uses 37 slices. The propagation delay, latency and speed are identical to the standard division component.

3.4.2 Multiplicative Inverse

As discussed in Section 2.2.4, the other method of division utilizes a multiplicative inversion block in conjunction with a multiplication block to circumvent doing actual division in hardware. To do this, only an inversion block needs to be designed. There are several different methods to this:

1. Brute Force Search
2. Fermat's Theorem
3. Modified Euclidean Algorithm
4. Lookup Table

Two of these methods can be done using serial and/or parallel methods. Using serial methods, fewer resources are needed for the blocks, but the computations will require more clock cycles to complete. In the parallel methods, the calculations will complete in a single clock cycle, but more resources will be required for the component.

Brute Force Search

The most simplistic implementation discussed is the brute force method of successively multiplying by each number in the Galois Field and testing to see if the product is equal to one. If it is, then the inverse has been found. The serial implementation would have only one multiply/compare operation, and then iterate through all the possible value, while the parallel version would duplicate this by the number of possible values to allow the computation to complete in one cycle. The parallel version was designed, but would not compile, as the resulting hardware was too large to fit in the

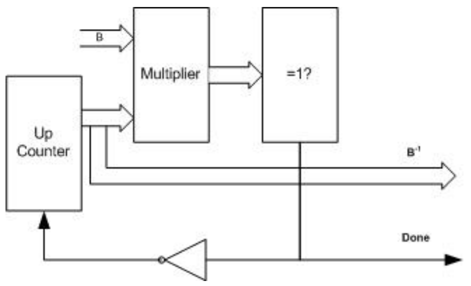

Figure 3.7: Serial Brute Force Inverse Search

largest target FPGA available. The serial version is feasible to implement, and the block diagram for it is shown in Figure 3.7.

The VHDL component for this serial version is called `GFI_Brute` and is shown in Appendix D. This entity uses the `load`/`done` method of handshaking in the same way that the LFSR multiplier does. In this case, the `done` signal really is necessary, as the component can take anywhere from 1 to 255 cycles to find the inverse, depending on the input value. This component requires 57 slices and has a propagation delay of 5.264 ns which allows this component to run at 190 MHz.

Fermat's Theorem

An alternative way of finding the multiplicative inverse in GF(2) is derived from Fermat's Little Theorem which was originally stated by Fermat, and proved later by Euler. This can be stated as[17]:

$$A^{p^n} = A \tag{3.4}$$

for any element A in the field. If this is divided by A on both sides:

$$A^{p^n-1} = 1 \tag{3.5}$$

Then rewriting the previous equation, it turns into:

$$A * A^{p^n-2} = 1 \tag{3.6}$$

Therefore,

$$A^{-1} = A^{p^n-2} \tag{3.7}$$

As there is no established way to do exponentiation without a lookup table, the straightforward method would be to cascade p^{n-2} multipliers that multiply the output of the last multiplier by A. This can be simplified in many ways, but the method in this work uses addition chains as used by Itoh-Tsujii[10]. One method of doing this is shown in Figure 3.8.

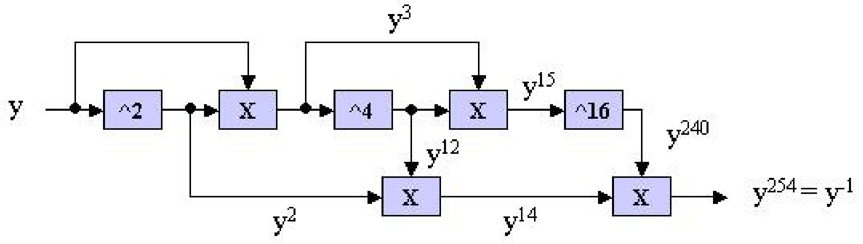

Figure 3.8: Computing Multiplicative Inverse by Computing y^{254}

This translates directly into combinational logic very simply, but does not lend itself to serialization, as the computations are different at each stage. This logic is defined as a VHDL function, which is called `GFIc` and is shown in Appendix C. This compiles into a 123 slice component, with a propagation delay of 5.969 ns. This allows it to be clocked at 168 MHz and compute an inverse in a single clock cycle.

Modified Euclidean Algorithm

Yet another method to implement the inverse function in GF Algebra is to use a modification of the Extended Euclidean Algorithm. The full derivation of this method can be found from the original paper written by Brunner et al[18], but a much abbreviated explanation is provided here.

The original Euclidean Algorithm was developed in order to find the Greatest Common Divisor (GCD) of two different numbers (A and B). The Extended Euclidean Algorithm finds the values of X and Y as well as $GCD(A, B)$, which are defined as:

$$GCD(A, B) = A \cdot X + B \cdot Y \tag{3.8}$$

This algorithm is useful for finding the multiplicative inverse. The primitive generator function (F) is by definition a “prime” in the field that it generates, so the GCD of any number in the field and F is 1. If this algorithm is done in GF(2^n):

$$\begin{aligned} &A \otimes X \oplus F \otimes Y = GCD(A, F) \\ &A \otimes X \oplus F \otimes Y = 1 \\ &A \otimes X = F \otimes Y \oplus 1 \end{aligned} \tag{3.9}$$

Since GF multiplication is done modulo F, any multiple of F is equivalent to $0 \ mod \ F$:

$$\begin{aligned} &A \otimes X = F \otimes Y \oplus 1 \\ &A \otimes X = 0 \oplus 1 \\ &A \otimes X = 1 \end{aligned} \tag{3.10}$$

So the value X is the multiplicative inverse of A. Therefore, implementing the Extended GCD algorithm using GF Algebra would find the GF multiplicative inverse of A. A problem with this method is that the Extended GCD algorithm uses division. Since the purpose of finding the inverse is to use it to implement GF division, there is no method for computing division in this algorithm. Brunner et al. developed a modification of the Extended Euclidean Algorithm that avoids this issue[18]. For this algorithm, several functions are assumed to be available:

$$\begin{aligned} &\texttt{GFM2(A)} \rightarrow A \otimes 2 \\ &\texttt{GFD2(A)} \rightarrow A \oslash 2 \end{aligned} \tag{3.11}$$

Using these, the algorithm can be implemented in hardware using the following logic:

```
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;
        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(B);
```

For our application, N = 8. The loop in this algorithm can either be done in serial or parallel. The serial implementation is called `GFI_Euclid` and is shown in Appendix D. This component also uses the `load`/`done` method of handshaking, though the component requires a consistent 16 cycles to compute the inverse. The component requires 64 slices and has a propagation delay of 3.252 ns, which allows this component to be clocked at 307 MHz.

The parallel version is almost exactly the same as the algorithm given above. Since this component is combinational, it is defined as a VHDL function rather then an entity. The VHDL that defines the component is called `GFIei` and is shown in Appendix C.

This component executes the same computations as `GFI_Euclid`, but in a single cycle. The propagation delay for this component is 5.750 ns, which equates to a 174 MHz clock frequency. This component requires 129 slices. It would seem that this component should be $2 \cdot n$ (or in this case 16) times larger than the serial version. The reason this is not the case is because the control logic in the serial version for looping and handshaking is complex compared to the actual computing logic, which is simply exclusive-or and shift operations. Since this control logic can be omitted in the parallel version, this balances out part of the duplication of the computing logic.

Lookup Table

Another straightforward method of finding the multiplicative inverse of a number is to implement a lookup table of this function. The output of this would then be routed to the multiplier. Assuming a lookup-table based multiply is not used, this method would use half the block RAM required for the pure lookup table division and use more hardware in fabric for the multiplier. This may be an appropriate compromise if not enough block RAM is available for both the logarithm and exponentiation tables. However if block RAM is tightly constrained in the system, then another implementation must be found.

GF Division and Inversion Summary

The methods described in this section are summarized in Tables 3.4 and 3.5.

Table 3.6 summarizes all the methods of division. When the inverters are used to implement division, a bit-parallel multiplier is used along with it.

GF Division	Slices	BRAMs	Prop. Delay (ns)	cycles/calc.	latency	max Freq (MHz)
BRAM	19	2	6.658	1	3	150.1952538
BRAM(2x)	37	2	6.658	1	3	150.1952538

Table 3.4: Galois Field Division Summary

GF Inversion	Slices	BRAMs	Prop. Delay (ns)	cycles/calc.	latency	max Freq (MHz)
Brute Force	57	0	5.264	Variable (1-255)	Variable (1-255)	189.9696049
Euclid (serial)	64	0	3.252	16	16	307.503075
Itoh	123	0	5.969	1	1	167.53225
Euclid (Parallel)	129	0	5.75	1	1	173.9130435
BRAM	0	1	2.46	1	1	406.504065

Table 3.5: Galois Field Inversion Summary

	Slices	BRAMs	Prop Delay (ns)	cyc/calc	latency	max Freq (MHz)	calc/sec (MHz)
BRAM	19	2	6.658	1	3	150.1952538	150.1952538
BRAM(2x)	37	2	6.658	1	3	150.1952538	150.1952538
BRAM (Inver)	38	1	6.515	1	2	153.4919417	153.4919417
Brute Force	95	0	6.515	2 - 256	2 - 256	153.4919417	153.49 - 0.60
Euclid (serial)	102	0	6.515	17	17	153.4919417	9.0289377
Itoh	161	0	6.515	1	2	153.4919417	153.4919417
Euclid (Parallel)	167	0	6.515	1	2	153.4919417	153.4919417

Table 3.6: Galois Field Division Summary

3.5 GF Multiplication by a Constant

There are many cases when GF multiplying when one of the values is a constant. In these cases, the hardware required can be simplified to take advantage of this. These methods require computation of certain values or transfer functions before runtime.

One of the most efficient ways of multiplying by a constant was introduced by Mastrovito[21]. His "Mastrovito Multiplier" can be described in many different ways, but the logic behind it is simple. Since a Galois Field is by definition a polynomial ring, any value in the field can be expressed as a power of 2. Because of this, if $GF(2^n)$ multiplication by a constant (A) is all that is required, this is simply multiplying by a power of 2. The transfer function for multiplying by 2 can be tiled $log(A)$ times to find the transfer function for this particular multiplication. This is the equivalent of "unrolling" the loops of the LFSR required to do these repeated multiplications by 2, similarly as was done to create the general-purpose bit-parallel multiplier. Cascading this logic block $log(A)$ times is not very expensive in terms of hardware, but it can be simplified even further. Initially, the worst case scenario for this calculation is when:

$$log(A) = 254 \tag{3.12}$$

This would mean that the following number of exclusive-or gates would be required:

$$Gates = 3 \cdot 254 = 762 \tag{3.13}$$

But since they are exclusive-or gates that are being cascaded:

$$A \oplus A = 0 \tag{3.14}$$

This can be utilized to greatly simplify the circuit. The true worst case scenario would be when, after simplifying, the function would require eight 8-input exclusive-or gates. An 8-input gate can be made from three 4-input lookup tables (LUT4), so the worst case scenario would only require 24 LUT4. Also, for the only slightly better case of 7-input exclusive-ors being required, this reduces the LUT4 per gate down to 2, and any gate with 4 or less inputs can be done with one LUT4. Because of this, the average case is only 10.7 LUT4. Comparing this to a full multiplier which uses 38 slices, there is a large savings in hardware.

3.6 GF Division by a Constant

One method for division by a constant is to use the combinational division algorithm, with the inverse of the constant pre-calculated. This method does not need to have the block to find the multiplicative inverse. A normal multiplication circuit can be used for this, which greatly simplifies the circuit.

A further improvement on this is to use an unrolled LFSR as is done for multiplying by a constant, but the feedback equation would be changed to do division by 2 modulo F. This can also be viewed as finding the equation that would “undo” the multiplication by 2 modulo F. Looking at it from this point of view, it is easy to derive the transfer function. For multiplying, the function used is:

$$\begin{aligned}
&2B_7 \leftarrow B_6\\
&2B_6 \leftarrow B_5\\
&2B_5 \leftarrow B_4\\
&2B_4 \leftarrow B_3 \oplus B_7\\
&2B_3 \leftarrow B_2 \oplus B_7\\
&2B_2 \leftarrow B_1 \oplus B_7\\
&2B_1 \leftarrow B_0\\
&2B_0 \leftarrow B_7
\end{aligned} \tag{3.15}$$

So for dividing, it would need to go the other direction:

$$\begin{aligned}
&B_6 \leftarrow 2B_7\\
&B_5 \leftarrow 2B_6\\
&B_4 \leftarrow 2B_5\\
&B_3 \leftarrow 2B_4 \oplus B_7\\
&B_2 \leftarrow 2B_3 \oplus B_7\\
&B_1 \leftarrow 2B_2 \oplus B_7\\
&B_0 \leftarrow 2B_1\\
&B_7 \leftarrow 2B_0
\end{aligned} \tag{3.16}$$

Since the function is being reversed, the value of B_7 is one of the unknowns to be found. But B_7 is equal to $2B_0$:

$$\begin{aligned}
&B_6 \leftarrow 2B_7\\
&B_5 \leftarrow 2B_6\\
&B_4 \leftarrow 2B_5\\
&B_3 \leftarrow 2B_4 \oplus 2B_0\\
&B_2 \leftarrow 2B_3 \oplus 2B_0\\
&B_1 \leftarrow 2B_2 \oplus 2B_0\\
&B_0 \leftarrow 2B_1\\
&B_7 \leftarrow 2B_0
\end{aligned} \tag{3.17}$$

Now reorganizing so that they are in order:

$$\begin{aligned}
&B_7 \leftarrow 2B_0\\
&B_6 \leftarrow 2B_7\\
&B_5 \leftarrow 2B_6\\
&B_4 \leftarrow 2B_5\\
&B_3 \leftarrow 2B_4 \oplus 2B_0\\
&B_2 \leftarrow 2B_3 \oplus 2B_0\\
&B_1 \leftarrow 2B_2 \oplus 2B_0\\
&B_0 \leftarrow 2B_1
\end{aligned} \tag{3.18}$$

Using this function, division by any power of 2 uses this transfer function tiled $log(A)$ times and then simplified by canceling exclusive-ors, just like in the case of multiplication by a constant. The worst and average cases are the same as with multiplication by a constant. Comparing this to the combinational division circuit (which uses 161 slices), there is a huge savings in the amount of hardware required.

3.7 GF Algebra Conclusions

The conclusions here are based off of the experimentally found results from the components shown using a 8-bit word size.

3.7.1 GF Addition/Subtraction

As stated earlier, GF addition and subtraction are a bit-wise exclusive-or and do not require any special hardware to calculate on a FPGA.

3.7.2 GF Logarithm and Exponentiation

Similarly, the only practical method of implementing GF logarithm and exponentiation is though lookup tables. The lookup tables are typically stored in RAM blocks. The resource utilization and speed for these components are summarized in Table 3.1 on page 19. The only drawback from this method is the use of block RAM.

3.7.3 GF Multiplication

When calculating GF multiplication, there are a number of established methods. A summary of resource utilization and speed of the different methods is shown in Table 3.3 on page 23.

The LFSR method seems the least practical in real life applications, but if a high clock speed is required, and the cycles required to do the calculations is not a factor, then this would be ideal. In most real world applications, it is better to be able to do the calculations in as few cycles as possible, so for most applications, one of the other methods would be preferable. Compared to the bit-parallel multiplier, using the lookup table method with block RAM will save a few slices, but will slow down the clock and will add latency to the calculations. The bit-parallel multiplier's high speed, small size, and freedom from using block RAMs make it a good choice for most applications.

3.7.4 GF Division and Inversion

For GF Division, the application will determine which method will be optimal. If lookup tables are used, the division can be done directly. Table 3.4 on page 28 shows the results of this method.

The other method of division is to first calculate the inverse of the divisor, and then multiply this by the dividend to find the quotient. Various methods of finding the GF multiplicative inverse were discussed in Section 3.4.2, and their results are summarized in Table 3.5 on page 28.

The inversion methods have a wide range of properties. Except under the most lax of requirements, the variable computation time of the brute force inverse component, of which the worst case is 255 cycles, cannot be recommended, despite the small size and relatively low propagation delay of the component. This component was included in the comparison purely as a baseline to compare other components with. The other serial method requires a consistent 16 cycles to complete the computation and requires 64 slices, which is only 7 slices more them the brute force method.

The high speed inverters all require 1 clock cycle to compute a multiplicative inverse. Both the Itoh-Tsujii and Modified Euclidean methods, despite having entirely different methods of arriving at the inverse, result in components that are roughly the same size and speed. Either can be used for high-speed calculations in systems that are tightly constrained in block RAM.

Under circumstances where there are plenty of RAM blocks available in the FPGA, the lookup table based division seems to be the obvious choice for division. They require fewer slices then the combinational inversion methods and does not require an additional multiplier to complete the division. They do require 3 cycles to complete and need 2 BRAMs per calculation. If RAM blocks are not abundant, then using inversion to do division may be preferable.

An inverter that only requires a single RAM block and is very fast can be created by mapping the iversion function to a block RAM. This block RAM is dual-ported, so it can actually be used to compute two divisions at the same time. If this is used in conjunction with either the combinational multiplication components discussed above, an interesting compromise between the standard GF division component is found. If used along with the two Mastrovito multiplier, two dividers with a latency of two cycles and high speed is created. If combined with two LFSR multipliers, the GF divider with the highest clock rate is attained, though the calculation requires 17 cycles to complete.

Chapter 4

Efficient Implementation of a RAID-6 FPGA library

RAID-6 has three possible operations to be designed, as described in Section 2.3. These are encoding of data, detection of an erroneous data block, and correction of two know data block errors. To ease and generalize the VHDL implementation of RAID-6 operations, a new constant was created in the package:

```
constant N: integer := 6;
```

This constant simply represents the number of data words there are in a stripe, or N_D up to this point in the text. For this work, the constant was set to the value 6, but this could be changed to any positive, even value. Not allowing this to be an odd value simplifies the design. Based off of this constant, several new data types were created to represent the actual stripe when passed in and out of various functions:

```
type parity is array (1 downto 0) of word;
type stripe is array (N-1 downto 0) of word;
type stripewp is array (N+1 downto 0) of word;
```

The `parity` type represents the parity words of a word stripe. the P and Q parity words are word 0 and word 1, respectively. The `stripe` type is the data stripe, and the `stripewp` is the data stripe with the parity concatenated as the N and N+1 words. Once again, evaluation of all of the components was on the Virtex-4 FX60.

4.1 RAID-6 Encoding

As described in Section 2.3.1, RAID-6 encoding uses two rotating parity blocks to create redundancy. These parity blocks (P and Q) are generated using GF algebra in the following manner.

$$P = \sum_{i=0}^{N_D-1} (D_i) \tag{4.1}$$

$$Q = \sum_{i=0}^{N_D-1} (2^i \otimes Di) \tag{4.2}$$

where D_i is the i^{th} data block in the data stripe that is being encoded.

It will be assumed that all of the data is arriving to the encoder block at the same time. If this is not the case, a trivial finite state machine can be made that will pass the incoming data blocks to the appropriate location in a stripe-sized register, so there is no loss of generality. The summation in these equations are simply exclusive-or operations. The only part of this that allows for variation is the multiplication by 2^i that is required when generating the Q parity. Since RAID-6 is typically required to run at high speed, the LFSR method of multiplication was not considered for this application. The two methods evaluated here are:

1. Table-based
2. Combinational (Mastrovito)

4.1.1 Table-Based RAID-6 Encoding

One way of generating the parity words are by using lookup-table based multiplication. Since the multiplication is by a constant, only one logarithm lookup and one exponentiation lookup are required per multiplication. This allows use of the GFM_BRAM_double component designed earlier, which reduces the block RAM requirements to N_D. Using this component, it is simple to generate the encoded parity for a given data stripe. The VHDL component to do this is called RAID6en_bram and is shown in Appendix D. A block diagram demonstrating this method is shown in Figure 4.1.

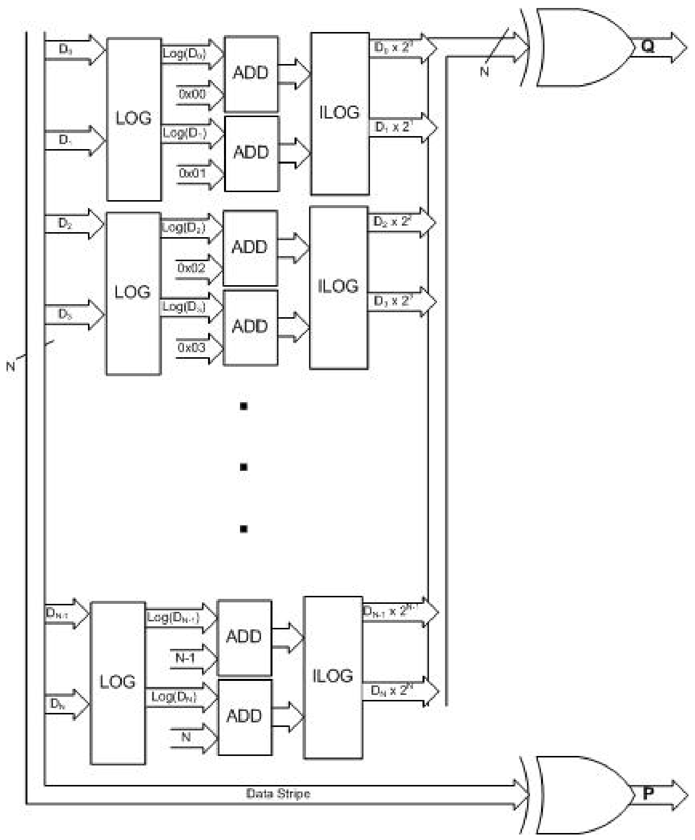

Figure 4.1: Table-based RAID-6 Encoding

This component uses a generate loop to generate the proper number of multiplication by constant components based on the constant N defined in the package. This generate statement will not compile properly if the value of N is not an even value, because it takes for granted that both sides of the multiplication component will be used. The result of the multiplication is loaded into a stripe register called “multiplied”, which is then exclusive-ored together in the process to calculate the Q parity word. Similarly, the P parity word is generated in the process by exclusively-oring the incoming stripe. When this component is compiled with the previously mentioned N=6, the component uses 147 slices and 6 RAM blocks. The propagation delay through the component is 7.452 ns, which allows a clock frequency of 134 MHz. The latency for this component is 4 cycles, and because the process needs to wait for the multiplication to complete before the Q block can be calculated, the latency of the component is also 4 cycle. This could be changed to a pipelined method, but this would further enlarge the component.

4.1.2 Combinational RAID-6 Encoding

The other method of doing the required multiplication by constants to encode the Q parity words is to use Mastrovito multipliers as discussed earlier. Since this method results in an combinational component, the VHDL was done as a function. The VHDL function is called `RAID6en` and is shown in Appendix C. A block diagram of this component is shown in Figure 4.2.

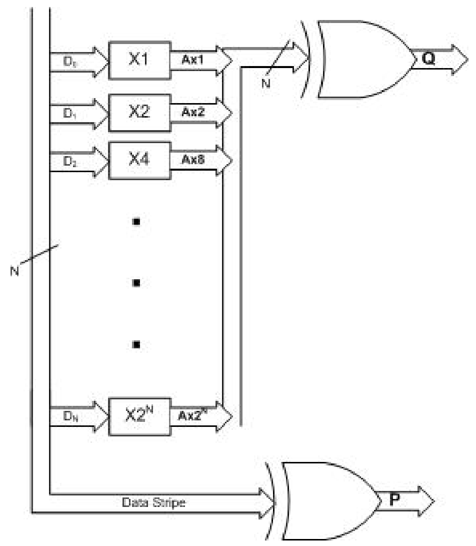

Figure 4.2: Combinational RAID-6 Encoder

The function is very simple. The stripe is stored in the "sin" (which is short for "stripe input"), and the parity is initialized. Then the input values are looped through and the multiplication and exclusive-or operations to generate the parity words are done in this loop. Then the parity is returned. This function compiles into a component that uses only 30 slices. The propagation delay through it is only 3.043 ns, which allows the component to be clocked at 329 MHz. Because the component is combinational, the latency and cycles required to complete the computation are 1 cycle.

4.1.3 Summary

The results of the compiled RAID-6 Encoding components are shown in Table 4.1.

	Slices	BRAMs	Prop Delay (ns)	cyc/calc	latency	max Freq (MHz)	calc/sec (MHz)
BRAM	147	6	7.452	4	4	134.1921632	33.5480408
Bit-Parallel	30	0	3.043	1	1	328.6230693	328.6230693

Table 4.1: RAID-6 Encoding Summary

4.2 RAID-6 Error Detection

As shown in Section 2.3.3, a single corrupted word in a word stripe can be located by applying the following equation:

$$log((Q \oplus Q') \oslash (P \oplus P')) = L \tag{4.3}$$

where P' and Q' are the recalculated parity words, and L is the location of the erroneous data. Because the logarithm is required, it is most efficient to do the division using logarithm and exponentiation lookup tables. With lookup tables, the division calculation is done in the following manner:

$$A \oslash B = exp[(log(A) - log(B)) \; mod \; (2^n - 1)] \tag{4.4}$$

where n is the number of bits in a word. If this is substituted into the error detection equation, this allows the following simplification:

$$\begin{aligned} &log(exp[(log(Q \oplus Q') - log(P \oplus P'))\ mod\ (2^n - 1)]) = L \\ &(log(Q \oplus Q') - log(P \oplus P'))\ mod\ (2^n - 1) = L \end{aligned} \tag{4.5}$$

This simplification reduces the cost of the calculation down to a dual-ported lookup table, and a subtraction modulo $2^n - 1$ circuit. This is used in the VHDL component called `RAIDf1_BRAM`, which is shown in Appendix D.

This component receives a corrupted stripe with parity and outputs the corrected version of that stripe. The logic to find the location of the error is fairly simple and follows Equation 4.5 logically. The first process calculates the correct data for location L (datal) from the P parity word. The second process simply creates a new corrected stripe that is the output of the component. A block diagram of this component is shown in Figure 4.3.

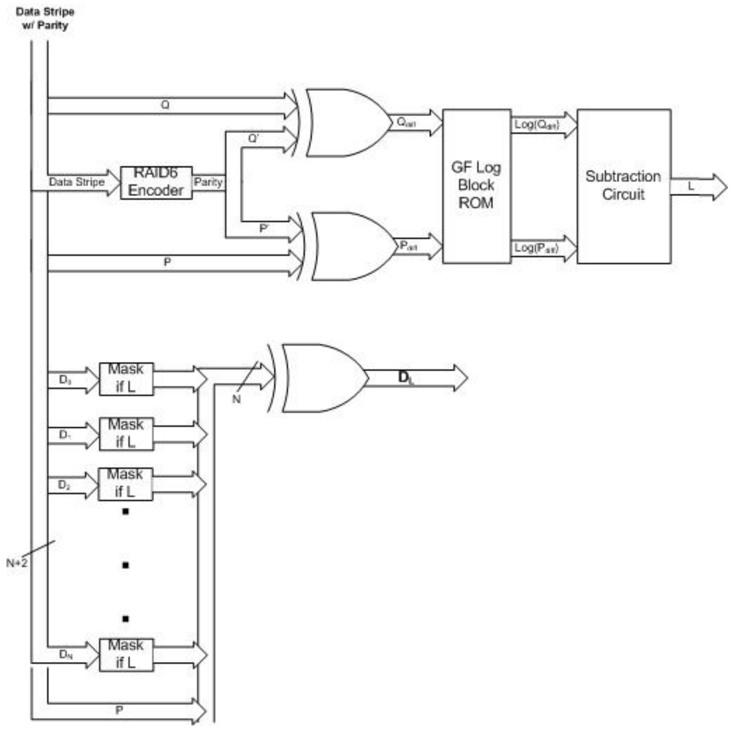

Figure 4.3: Block Diagram of `RAIDf1_BRAM`

The top part finds the location of the data error, and the bottom part uses the output of it to correct the error. This component uses 95 slices and 1 RAM block. The latency for a computation is two cycles. The propagation delay through the device is 7.452 ns, which allows for a clock speed of 103 MHz. This is summarized in Table 4.2. The clock speed could easily be improved through deepening the pipeline, though it would enlarge the component and increase latency.

	Slices	BRAMs	Prop Delay (ns)	cyc/calc	latency	max Freq (MHz)	calc/sec (MHz)
BRAM	95	1	9.715	1	2	102.9336078	102.9336078

Table 4.2: RAID-6 Error Correction Summary

4.3 RAID-6 Error Correction

As shown in Section 2.3.2, if errors can be detected by some other means, two simultaneous errors can be corrected in a stripe. To summarize the method described earlier, there are 4 possible error scenarios that need to be accounted for.

1. The P and Q blocks
2. The Q block and a Data block
3. The P block and a Data block
4. Two Data blocks

When creating hardware to do this, The first 3 cases are simple. The final case requires the following equations to be implemented in hardware:

$$\begin{aligned} D_K &= (2^{-K} \otimes (Q \oplus Q') \oplus 2^{L-K} \otimes (P \oplus P')) \oslash (2^{L-K} \oplus 1) \\ D_L &= P \oplus P' \oplus D_K \end{aligned} \tag{4.6}$$

This case is challenging, as it requires division by a variable value that is not a power of 2. A block diagram of the hardware required to make these calculations is shown in Figure 4.4. Once again, both a table-based version and a combinational version are designed and evaluated here.

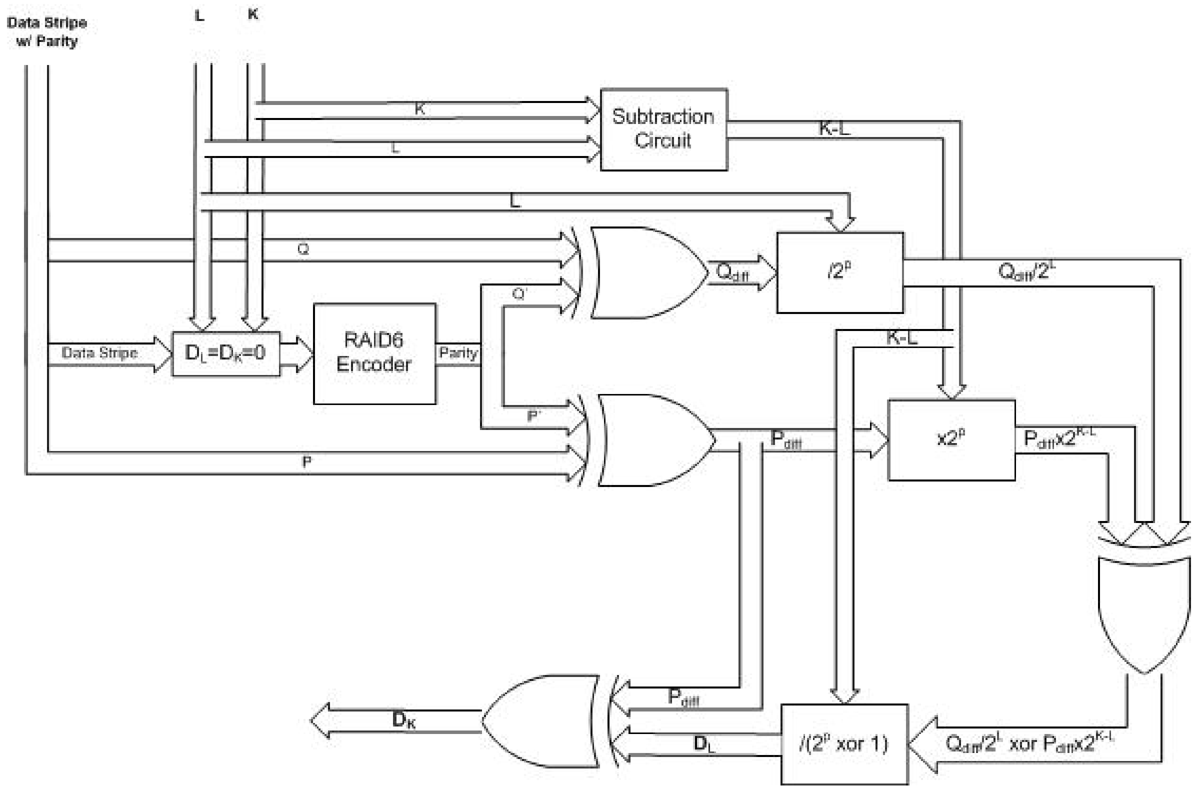

Figure 4.4: RAID-6 Error Correction Block Diagram

4.3.1 Table-Based RAID-6 Error Correction

Since general-purpose GF division is difficult to do efficiently without lookup tables, the lookup table based GFA seems attractive. A VHDL code implementation of this is called `RAID6c2_bram` and is shown in Appendix D.

The first process simply copies over the good data from the input stripe "a" and zeros out the bad data. The second process calculates $2^{L-K} \oplus 1$ by using bit shifts to perform the exponentiation. This saves a table lookup. The P' and Q' parities are found using the good data, and the difference is calculated using exclusive-ors. The division and multiplication in the dividend are calculated using the lookup table based multiplier and divider shown in 3.3.1 and 3.4.1. Some type-conversion is done to allow the integers to be passed into these pre-designed blocks. These values are exclusive-ored together to form the dividend of the end division. Then the final division is formed, an exclusive-or is done to correct the value of D_K, and D_L is calculated by a simple bit-wise exclusive-or.

When compiled, this component requires 12 RAM blocks and 289 slices. The propagation delay through the component is 7.297 ns, which allows a clock frequency of 137 MHz. The calculation has a latency of 9 clock cycles.

4.3.2 Combinational RAID-6 Error Correction

The combinational version of this device is not nearly as straight forward. This calculation requires division by $2^{L-K} \oplus 1$ which is a variable value that is not a power of 2, so the obvious method available is to use a full $GF(2^8)$ divider. A full divider requires an inversion circuit, which is very logic intensive. To avoid this, a simplification is made. Rather than using a general-purpose division block, several division by a constant circuits are used, and then the correct value is multiplexed into the circuit. Since division by a constant is many times more efficient than general-purpose division, this saves slices and computing time for the circuit. But as the number of storage devices increases, the possible values of L and K go up, which requires more multiplication by constant circuits. So with a larger number of storage devices, using the general purpose division circuit will be more efficient. This division by a constant circuit uses the techniques shown in 3.6 to calculate the transfer equations for all values of $2^{L-K} \oplus 1$ for $N_D \leq 8$, which means that $L - K$ will be in the range 1 to 7. If more storage devices are used, simply finding the correct transfer equations is all that is required to extend this function. The function is called `divby_2pxor1` and is shown in Appendix C.

This function is straight forward, but because each bit's transfer equation is manually entered, it is not short. Each division by a constant result is calculated in parallel and stored in an array, and the results are then multiplexed out by means of the array reference in the return statement. Using this specialized division function, correcting two errors becomes much simpler. The function to do the required calculations is called `RAID6c2ecomb` and is shown in Appendix C.

This combinational version works very similarly to the table-based version. First, the data is copied into registers for the output stripe (`sout`), and the good data is copied into a data stripe to compute P' and Q'. Also, the original parity is stored in `Pi` and `Qi`. Then P' and Q' are computed, and the difference between then and the original are calculated. Also, the difference between L and K is computed. These values are then used along with the multiply and divide by a power of 2 functions created earlier and the special `divby_2pxor1` function to calculate D_K. This value is then used to calculate D_L.

This component was compiled and required 273 slices. A calculation is computed each clock cycle. The propagation delay through this combinational circuit was 18.577 ns, which allows a maximum clock frequency of 53.8 MHz. After analysis, 13.331 ns of this propagation delay was caused by the routing delay from the high fan-out of the input word in the `divby_2pxor1` function. This can be mitigated if the calculations are pipelined. Keeping this in mind, an alternate implementation is designed. This implementation uses a 3 stage pipeline, but the pipeline could be deeper or shallower if different clock speeds are required. The component for this is called `RAID6c2e_pipe` and is shown in Appendix D.

The calculations are partitioned so that the generation of the P', Q', and all the differences are done in the first cycle, the GF calculations to find D_K are done in the second cycle, and the exclusive-or (for D_L) and all the output registering is done in the third cycle. So the latency in this device is 3 cycles. After compilation, 255 slices are used, and the new propagation delay is 6.724 ns, which allows a clock frequency of 149 MHz.

4.3.3 Summary

The results of the compiled RAID-6 Error Correction components are shown in Table 4.3.

	Slices	BRAMs	P Delay (ns)	cyc/calc	latency	max Freq (MHz)	calc/sec (MHz)
BRAM	289	12	7.297	1	9	137.0426203	137.0426203
Bit-Parallel	273	0	18.577	1	1	53.83000484	53.8300048
Bit-Parallel (Pipe)	255	0	6.724	1	3	148.7209994	148.7209994

Table 4.3: RAID-6 Double Error Correction Summary

4.4 RAID-6 Conclusions

Chapter 4 discussed the three different operations to implement RAID-6 functionality. After data has been encoded using RAID-6, it can be decoded to either find a single error, or correct two known errors. Each of these operations can be done with or without BRAM, which lend themselves to different systems depending of the requirements. These calculations are based on using a 6-word data stripe, as shown in Chapter 4. The conclusions here are based off of the experimentally found results from the components shown up to this point in Chapters 3 and 4 using a 8-bit word size and 6-word data stripe (or 8-word stripe with parity included).

4.4.1 RAID-6 Encoding

The two methods for encoding are summarized in Table 4.1 on page 33. It is plain that using lookup tables to calculate the RAID-6 encoding is inferior in size and speed to using bit-parallel multipliers to do the calculations.

4.4.2 RAID-6 Error Detection

As discussed earlier, when finding a single corrupted data block in RAID-6 encoded data, the lookup-table based method is the best method to use. Since the logarithm table is required for the calculation anyway, the simplified calculations based on dividing using lookup tables greatly simplifies the calculations. The requirements for this component is shown in Table 4.2 on page 34.

4.4.3 RAID-6 Error Correction

A summary of the different components to use RAID-6 to correct two known errors is shown in Table 4.3 on page 36.

The specialized bit-parallel RAID-6 error correction component uses fewer slices and has no block RAM requirements. It is much slower then any other components described up to this point. The pipelined version of this fixes that shortcoming (and is actually faster than the table-based method) and still manages to be smaller than both of the other components. It is theorized that the pipelining allowed the component to be simpler than its combinational counterpart, because the pipelining allowed built-in registers within the slices to be used which reduced routing requirements in the component.

The resource utilization of this specialized division is highly dependent on the system it is designed for. Unlike methods that use a general-purpose divider, the specialized divider scales in size with N_D, so the more storage devices that are used in the system, the larger this component becomes. For the particular n and $f(x)$ used up to this point, the slices used based off N_D can be approximated as shown in Figure 4.5.

The amount of logic used is relatively linear to the number of storage devices in the system. The variations shown are because it is impossible to use only a fraction of a LUT4. It seems impossible to find a simple equation that summarizes this behavior accurately, particularly at lower values of N_D. However, an approximate relationship between number of storage devices and number of LUT4 is:

$$\text{No. LUT4} = 10.7 \cdot N_D \tag{4.7}$$

This relationship consistently overestimates the number of LUT4 that are necessary up until N_D is over 170, so this should be treated as an upper bound for most practical applications. But even taking this relationship as accurate, the number of LUT4 need for most applications is reasonable. Until N_D is in excess of 60, the special-purpose divider will result in a smaller component then the general-purpose divider. However, for larger values of N_D, the general purpose divider should be considered.

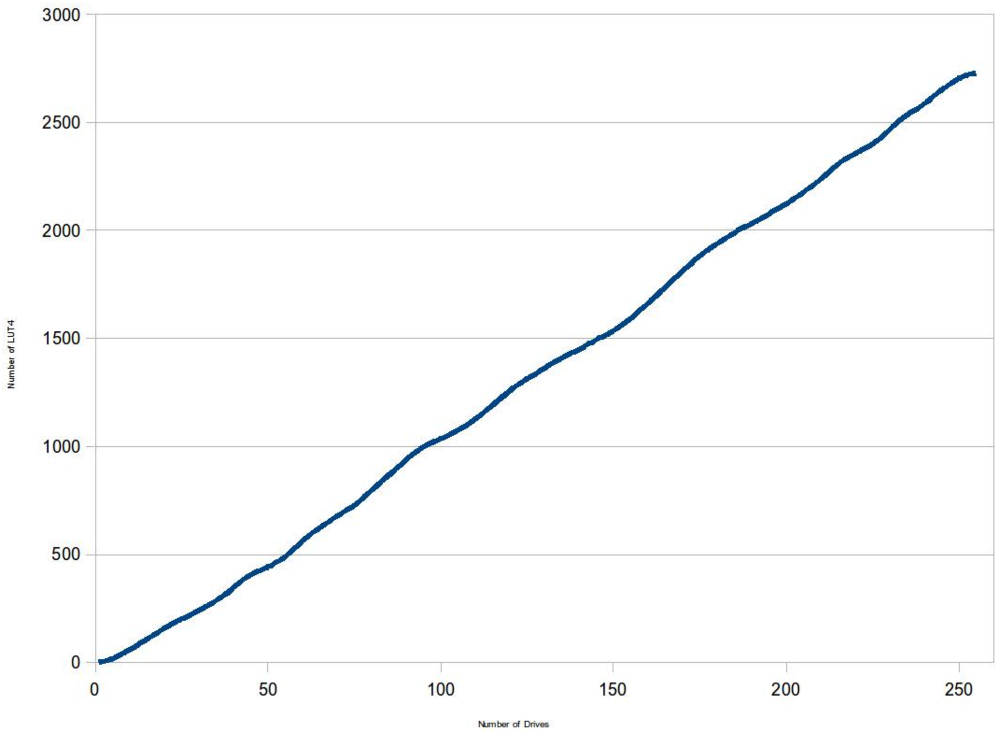

Figure 4.5: Chart of FPGA LUT4 used in a RAID-6 system based on Storage Device Count

Chapter 5

Conclusions and Future Work

5.1 Conclusions

All of the performance and resource utilization in this text were examined with respect to a Virtex-4 FX60.

GF Algebra, which is essential to RAID-6, has various basic operations which have been evaluated here. GF addition and subtraction are a bit-wise exclusive-or and do not require any special hardware to calculate on a FPGA. The only practical method of computing GF logarithm and exponentiation is using lookup tables. Though many different methods exist to calculate GF multiplication, it is best done using the bit-parallel method described in Section 3.3.3. For GF division, the application determines the preferred method, depending on how available block RAMs are in the design, and the amount of time available to do the calculations. Inversion can be used to do GF division. It can either be computed combinationally or with block RAM, but the combinational method is fairly large and significantly slower.

When implementing RAID-6, the three operations are encoding, error detection, and error correction. The results in this work are based off of a six data drive system, but some speculation to how it would scale is also included here. RAID-6 encoding using bit-wise parallel multipliers is faster and consumes fewer fabric slices than the block RAM approach and does not use any block RAM. For the table based method, each additionall 2 data drives added to the system will require an additional double GF multiplier, so this will scale at a rate of about 25 slices and an additional block RAM per drive added to the system. However, the bit-parallel method utilizes optimized transfer equations to multiply by the required constants. As discussed earlier, multiplication by a constant can be predicted to be about 10.7 slices per multiplication. So when more drives are added to the system, the combinational version should always require at least 57% fewer slices then the equivalent table-based version, and still have no block RAM consumption. It is not expected that the speed of either component will significantly degrade as it scales, as the only calculation that is not done in parallel is the final GF summation, which is simply an 8-bit exclusive-or. Because of this, it is expected that the combinational method will maintain a significantly faster speed then it's table-based counterpart.

The most efficient method to use RAID-6 double error correction depends on the size of the system. In general, the effectiveness of using a combinational encoder to generate the P' and Q' parity required for the calculations has been established. As before this will scale with N_D. The rest of the calculations have different trade-offs as N_D grows. This is because both K and L in the calculations can have any value from within the range of 0 to N_D. For $N_D < 8$, the combinational methods of multiplying or dividing by a power of 2 or the special-purpose divider will be much smaller than the general-purpose versions used in the table-based error correction. However, the table-based versions will not scale with the number of drives. They will still require the same 6 bock RAMs and associated slices, assuming all the multiplications and logarithms are done in block RAMs (which is not true with the current incarnation; there are some simplifications based on the current $N_D < 8$). However, the combinational methods will need to account for the larger range of K and L by adding more transfer equations as N_D grows. These transfer equations are approximately 10.7 slices per bit, so the combinational calculations are expected to scale up at about roughly 240 slices per drive. This estimate greatly over estimates at lower values of N_D (as shown by $N_D = 6$ only consuming 273 slices), but instead shows the expected trend over all. Even so, it is estimated that, assumed that the calculations use the same RAID-6 encoding method, when N_D get past 16, the combinational version will consume significantly more slices than the table-based version, and the trade-off between the 6 block RAMs of the table-based version and the number of slices required by the combinational version will become much less one-sided. It is also expected that as N_D gets past

50, the gain from consuming 9 block RAMs will outway the cost. These considerations do not take into account the increased propigation delay that the combinational version would experience. It is difficult to predict how this would scale from current data, but the table-based version would not have this degraded performance as N_D grows.

The most efficient method of detecting a single error is to use the simplified logarithm table approach described here. The caculations used to detect the error are independent of the number of drives in the system. The only thing that would scale up is the generation of the P' and Q' parities for the calculations. Assuming the use of a combinational RAID-6 encoder is used to generate the P' and Q' needed for the calculations, this scaling would also be at about 10.7 slices per drive added to the system. Of course, the subsequent correcting of the detected error would scale with the number of drives in the system as well.

There are a number of hardware improvements made here, that have not been found in the current literature. One of these is exploiting the dual-ported nature of Xilinx block RAM to compute 2 multiplication/divisions per hardware block. This reduces the cost of multiplication and division down to one block RAM ber calculation, assuming the log of one of the inputs is already know. The hardware simplifications possible when calculating GF division by a constant does not seem to be address anywhere in the liturature. This allows division by a constant to be reduced down to the same size as multiplication by a constant, despite the greater difficulty in dividing by a variable value. Also, the mathematical simplifications used to simplify RAID-6 error detection down to a single dual-ported Block RAM and a subtraction modulo F circuit seems novel as well. Finally, the use of a special-purpose divider to ease RAID-6 double error correction has not been attempted up to this point.

In addition to this, a complete performance evaluation of both GF Algebra and RAID-6 was completed on the Virtex-4 FX60, with $N_D = 6$, using both established methods and those that were developed here. It is the hope of the author that this work will motivate further study into high-performance hardware developments for GF Algebra and RAID-6, as well as serve as a basis for further FPGA implementations.

5.2 Future Work

There are many avenues to explore based on this work. Exploring optimized solutions to RAID-6 with a larger N_D would be of great use to the field. While it is expected that the RAID-6 encoding and error detection predictions stated here are accurate, they should be verified. Also, RAID-6 double error correction looks to be an interesting problem, where it is projected that scaling will render solutions used here impractical. The table-based method used here takes advantage of several simplifications that a small N_D allows, but these are not expected to scale well with larger values of N_D and should be replaced with the traditional table methods. Another possible solution to this problem would be a hybrid error correction unit that uses combinational methods for part of the calculations, and uses table methods for places where the combinational methods will not scale well.

Also, all testing in this work has been on the Virtex-4 FX60, and these VHDL components may perform differently on different FPGAs. Porting these designs to other Virtex-4 Xilinx FPGAs should be a simple task, assuming block RAM is available. Porting to other Xilinx FPGAs would require slightly more deliberation, but should still be fairly simple. However, porting these designs to the Altera platform may be more difficult than porting to other Xilinx FPGAs because the block RAM may not be compatible to those used by Xilinx. Altera's FPGAs have more options for radiation tolerant and hardened designs, which is required for the space environment.

To take it a step further, future work would be to evaluate these designs for use in a radiation hardened VLSI design. The VHDL components would need to be placed and routed. In addition, accounting for the space environment may require a redesign to include radiation mitigation technologies and methods. This would allow for a true radiation hardened version of this design, which would be beneficial for prolonged missions into space.

Bibliography

[1] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *the 1988 ACM SIGMOD international conference on Management of data*, 1988.

[2] Altera, "Logic Elements and Logic Array Blocks in Cyclone III Devices." `http://www.altera.com/literature/hb/cyc3/cyc3_ciii51002.pdf?GSA_pos=4&WT.oss_r=1&WT.oss=LE`, 2008. Product Specification.

[3] Xilinx, "Virtex-4 Family Overview." `http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf`, 2007. Product Specification.

[4] M. J. S. Smith, *Application-Specific Integrated Circuits*. Addison-Wesley Longman Inc, 1997.

[5] Xilinx, "Getting Started with FPGAs." `http://www.xilinx.com/company/gettingstarted/index.htm`, 2009. Website.

[6] J. S. Plank, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems." `http://www.cs.utk.edu/~plank/plank/papers/CS-96-332.html`, 1996.

[7] H. P. Anvin, "The mathematics of RAID-6," 2009. Linux RAID Implementation Documentation.

[8] E. R. Berlekamp, "Factoring Polynomials Over Finite Fields," *The Bell System Technical Journal*, 1983.

[9] J. L. Massey and J. K. Omura, "Computational method and apparatus for finite field arithmetic," 1981. U.S. Patent Application.

[10] T. Itoh and S. Tsujii, "A Fast Algorithm for Computing Multiplicative Inverses in GF(2ˆm) Using Normal Bases," *Information and Computation*, vol. 78, no. 3, pp. 171–177, 1988.

[11] C. Wang, T. K. Truong, H. M. Shao, L. J. Deutsch, J. K. Omura, and I. R. Reed, "VLSI Architectures for Computing Multiplications and Inverses in GF(2ˆm)," *IEEE Transactions on Computers*, vol. C-34, no. 8, pp. 709–716, 1985.

[12] A. J. Menezes, T. Okamoto, and S. A. Vanstone, "Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field," *IEEE Transactions on Information Theory*, vol. 39, no. 5, pp. 1639–1646, 1993.

[13] D. V. Bailey and C. Paar, "Optimal Extension Fields for Fast Arithmetic in Public-Key Algorithms," *Lecture Notes in Computer Science*, vol. 1462, pp. 472–485, 1998.

[14] H. Wu, "Low Complexity Bit-Parallel Finite Field Arithmetic Using Polynomial Basis," *Lecture Notes in Computer Science*, p. 727, 1999.

[15] C. Paar and M. Rosner, "Comparison of arithmetic architectures for Reed-Solomon decoders in reconfigurable hardware," in *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[16] J. Guajardo and C. Paar, "Itoh-Tsujii Inversion in Standard Basis and Its Application in Cryptography and Codes," *Designs, Codes and Cryptography*, vol. 25, no. 2, pp. 207–216, 2002.

[17] K. Araki, I. Fujita, and M. Morisue, "Fast inverters over finite field based on Euclid's algorithm," *Transactions IEICE*, vol. E-72, no. 11, pp. 1230–1234, 1989.

[18] H. Brunner, A. Curiger, and M. Hofstetter, “On Computing Multiplicative Inverses in GF(2ˆm),” *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 1010–1015, 1993.

[19] M. DiPaolo, “Hardware Accelerator for RAID6 Parity Generation/Data Recovery Controller.” `http://www.xilinx.com/bvdocs/appnotes/xapp731.pdf`, 2007. Xilinx Application Note for Virtex-4 Family.

[20] E. D. Mastrovito, “Vlsi designs for multiplication over finite fields GF(2m),” *Lecture Notes in Computer Science*, vol. 357, pp. 297–309, 1989.

[21] E. Mastrovito, *VLSI Architectures for Computaions in Galois Fields*. PhD thesis, Linkoping Univiversity, 1991.

Appendix A

Definitions

A.1 Terms and Abreviations

Fabric The general-purpose portion of a FPGA's logic (e.g. FPGA slices).

Bit A single Boolean digit with two states which may be represented as "1 or 0", "True or False", "High or Low", etc.

Word A n-bit bit string

Storage Device Any digital device used to store many strings of binary data (e.g. Hard Drive, Flash Drive, RAID, Tape Drive, etc.)

Capacity The amount of binary data that can be stored in the storage device

Sector The smallest subdivision of data on a storage device

RAID "Redundant Array of Independent/Inexpensive Drives", refer to Section 2.1

Array A number of storage devices configured to act as a single storage device

Sector Stripe The string of N sectors that are (or intended to be) at the same location on each of the N storage devices in the array

Word Stripe The string of N words that are (or intended to be) at the same location in the sector stripe

Redundancy The number of sectors in a sector stripe (or storage devices in an array) that may be lost without corrupting the data on the array

GF see "Galois Field"

Galois Field A mathematical field which is of finite size

Galois Field Algebra A field of abstract algebra which uses Galois Fields (finite fields) to compute the results of algebraic operations, refer to Section 2.2

A.2 Symbols

N_D number of data words in the stripe

N number of words in the stripe/number of drives in the array

$\oplus$ Exclusive-Or Binary Operator or Galois Field Addition/Subtraction

$\otimes$ Galois Field Multiplication

$(x << y) mod F$ y LFSR shifts of x; equivalent to x is shifted left and reduced modulo F, y times

$\oslash$ Galois Field Division

+ Standard Addition

- Standard Subtraction

$\cdot$ Standard Multiplication

/ Standard Division

$\lceil x \rceil$ Ceiling function – the smallest integer value that isn't less than x

$\sum_{i=0}^{n-1} D_i$ Sums array D from 0 to $n-1$ using GF addition

Appendix B

GF Logarithm and Exponentiation Tables

GF log	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	X[1]	0	1	19	2	32	1A	C6	3	DF	33	EE	1B	68	C7	4B
1	4	64	E0	0E	34	8D	EF	81	1C	C1	69	F8	C8	8	4C	71
2	5	8A	65	2F	E1	24	0F	21	35	93	8E	DA	F0	12	82	45
3	1D	B5	C2	7D	6A	27	F9	B9	C9	9A	9	78	4D	E4	72	A6
4	6	BF	8B	62	66	DD	30	FD	E2	98	25	B3	10	91	22	88
5	36	D0	94	CE	8F	96	DB	BD	F1	D2	13	5C	83	38	46	40
6	1E	42	B6	A3	C3	48	7E	6E	6B	3A	28	54	FA	85	BA	3D
7	CA	5E	9B	9F	0A	15	79	2B	4E	D4	E5	AC	73	F3	A7	57
8	7	70	C0	F7	8C	80	63	0D	67	4A	DE	ED	31	C5	FE	18
9	E3	A5	99	77	26	B8	B4	7C	11	44	92	D9	23	20	89	2E
A	37	3F	D1	5B	95	BC	CF	CD	90	87	97	B2	DC	FC	BE	61
B	F2	56	D3	AB	14	2A	5D	9E	84	3C	39	53	47	6D	41	A2
C	1F	2D	43	D8	B7	7B	A4	76	C4	17	49	EC	7F	0C	6F	F6
D	6C	A1	3B	52	29	9D	55	AA	FB	60	86	B1	BB	CC	3E	5A
E	CB	59	5F	B0	9C	A9	A0	51	0B	F5	16	EB	7A	75	2C	D7
F	4F	AE	D5	E9	E6	E7	AD	E8	74	D6	F4	EA	A8	50	58	AF

Table B.1: GF Logarithm Table

[1] GF log(0x00) is not defined

GF exp	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1	2	4	8	10	20	40	80	1D	3A	74	E8	CD	87	13	26
1	4C	98	2D	5A	B4	75	EA	C9	8F	3	6	0C	18	30	60	C0
2	9D	27	4E	9C	25	4A	94	35	6A	D4	B5	77	EE	C1	9F	23
3	46	8C	5	0A	14	28	50	A0	5D	BA	69	D2	B9	6F	DE	A1
4	5F	BE	61	C2	99	2F	5E	BC	65	CA	89	0F	1E	3C	78	F0
5	FD	E7	D3	BB	6B	D6	B1	7F	FE	E1	DF	A3	5B	B6	71	E2
6	D9	AF	43	86	11	22	44	88	0D	1A	34	68	D0	BD	67	CE
7	81	1F	3E	7C	F8	ED	C7	93	3B	76	EC	C5	97	33	66	CC
8	85	17	2E	5C	B8	6D	DA	A9	4F	9E	21	42	84	15	2A	54
9	A8	4D	9A	29	52	A4	55	AA	49	92	39	72	E4	D5	B7	73
A	E6	D1	BF	63	C6	91	3F	7E	FC	E5	D7	B3	7B	F6	F1	FF
B	E3	DB	AB	4B	96	31	62	C4	95	37	6E	DC	A5	57	AE	41
C	82	19	32	64	C8	8D	7	0E	1C	38	70	E0	DD	A7	53	A6
D	51	A2	59	B2	79	F2	F9	EF	C3	9B	2B	56	AC	45	8A	9
E	12	24	48	90	3D	7A	F4	F5	F7	F3	FB	EB	CB	8B	0B	16
F	2C	58	B0	7D	FA	E9	CF	83	1B	36	6C	D8	AD	47	8E	X(1)

Table B.2: GF Exponentiation Table

(1) GF exp(0xFF) is not defined

Appendix C

GF Algebra and RAID-6 Library Package

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

package pkg_RAID6 is
  constant W: integer := 8;
  constant N: integer := 6;
  constant F: std_logic_vector(W downto 0) := "100011101";

  constant WpDW: integer := 4;

  subtype word is std_logic_vector(W-1 downto 0);
  subtype DW is std_logic_vector((WpDW*W)-1 downto 0);

  type parity is array (1 downto 0) of word;
  type stripe is array (N-1 downto 0) of word;
  type stripewp is array (N+1 downto 0) of word;
  type LUT is array ((2**W)-1 downto 0) of word;

  type DWparity is array (1 downto 0) of DW;
  type DWstripe is array (N-1 downto 0) of DW;
  type DWstripewp is array (N+1 downto 0) of DW;

  type divfunc is array (7 downto 1) of word;

  constant  max      : integer := (2**W)-1;
  constant  zero     : word := (others => '0');
  constant  one      : word := (0 => '1',others => '0');
  constant  hi       : word := (others => '1');


  function log2 (x : positive) return natural;
  function highestBit(A: std_logic_vector) return integer;
  function onlyBit(A: std_logic_vector;b: integer) return std_logic;

  function Calc_GFILOG return LUT;
  function Calc_GFLOG( ILOG : LUT ) return LUT;
  function GFLOGlut (A: std_logic_vector) return std_logic_vector;
  function GFILOGlut (A: std_logic_vector) return std_logic_vector;

  function GFM (A, B: std_logic_vector) return std_logic_vector;
  function GFM2 (A: std_logic_vector) return std_logic_vector;
  function GFM2p (A: std_logic_vector;p: integer range 0 to 7) return std_logic_vector;
  function GFMc (A, B: std_logic_vector) return std_logic_vector;
  function GFMe (A, B: std_logic_vector) return std_logic_vector;

  function GFD (A, B: std_logic_vector) return std_logic_vector;
  function GFD2 (A: std_logic_vector) return std_logic_vector;
  function GFD2p (A: std_logic_vector;p: integer range 0 to 7) return std_logic_vector;
  function GFMILOGs (A: std_logic_vector) return std_logic_vector;

  function GFIc (A: std_logic_vector) return std_logic_vector;
  function GFIl (A: std_logic_vector) return std_logic_vector;
  function GFIei (A: std_logic_vector) return std_logic_vector;

  function RAIDenc (A: stripe) return parity;
  function RAID6CompC1 (c1,c2: integer range 0 to N-1) return word;
  function RAID6CompC2 (c1,c2: integer range 0 to N-1) return word;
  function RAID6c2e (A: stripewp;B,C: word;x,y: integer) return stripewp;
  function RAID6c1 (A: stripewp) return stripewp;

  function divby_2pxor1 (input: word; power:integer range 1 to 7) return word;
  function RAID6c2ecomb (A: stripewp;x,y: integer) return stripewp;

  function RAIDencDWstripe (A: DWstripe) return DWparity;

end pkg_RAID6;
package body pkg_RAID6 is

-----------------------------------------------------------------------
--Function: log2 - returns the base 2 logarithm of x
--   Input: x - positive integer
-- Returns: natural number
--  Ussage: Y = highestBit(X);
-----------------------------------------------------------------------
  function log2 (x : positive) return natural is
    variable temp, log: natural;
  begin
    temp := x / 2;
    log := 0;
    while (temp /= 0) loop
      temp := temp/2;
      log := log + 1;
    end loop;
    return log;
  end function log2;

-----------------------------------------------------------------------
--Function: highestBit - returns the bit position of the highest order bit
--   Input: A - 8-bit standard logic vector
-- Returns: integer range 0 to 7.
--  Ussage: Y = highestBit(X);
```

```
-----------------------------------------------------------------------
  function highestBit(A: std_logic_vector) return integer is
    variable ret  : integer range 0 to N+1;
    variable ai   : std_logic_vector(N+1 downto 0);
  begin
    ai := A;
    for i in A'range loop
      ret := i;
      exit when ai(i) = '1';
    end loop;
    return ret;
  end highestBit;

-----------------------------------------------------------------------
--Function: onlyBit - returns '0' if bit "b" is the only bit,
--              otherwise, returns '1'
--   Input: A - 8-bit standard logic vector, b - integer range 0 to 7
-- Returns: std_logic
--  Ussage: Y = onlyBit(X,i);
-----------------------------------------------------------------------
  function onlyBit(A: std_logic_vector;b: integer) return std_logic is
    variable  ret    : std_logic;
  begin
    ret := '0';
    for i in 7 downto 0 loop
      if i = b then
        ret := ret or not(A(i));
      else
        ret := ret or A(i);
      end if;
    end loop;
    return ret;
  end onlyBit;

-----------------------------------------------------------------------
--Function: Calc_GFILOG - creates the inverse log table for W bits
--               and generator F
--   Input: none
-- Returns: LUT
--  Ussage: GFILOG_LUT = Calc_GFILOG;
-----------------------------------------------------------------------
  function Calc_GFILOG return LUT is
    variable Temp : std_logic_vector(W downto 0);
    variable ILOG : LUT;
  begin
  ILOG(0) := (0 => '1', others => '0');
  for i in 1 to (2**W-1) loop
    Temp := ILOG(i-1) & '0';
    if ( Temp(8) = '1' ) then
      Temp := Temp xor F;
    end if;
    ILOG(i) := Temp(W-1 downto 0);
  end loop;
  return ILOG;
  end function;

----------------------------------------------------------------------
--Function: Calc_GFLOG - creates the log table from the inverse log table
--   Input: LUT
-- Returns: LUT
--  Ussage: GFLOG_LUT = Calc_GFLOG(Calc_GFILOG);
----------------------------------------------------------------------
  function Calc_GFLOG( ILOG : LUT ) return LUT is
  variable LOG : LUT;
  begin
    LOG(0) := (others => '1');
    for i in 0 to (2**W-2) loop
      LOG(conv_integer(ILOG(i))) := conv_std_logic_vector(i,W);
    end loop;
    return LOG;
  end function;

----------------------------------------------------------------------
--Function: GFLOGlut - returns the Galois Field LOG of the input
--              from a Look-Up Table
--   Input: A - 8-bit standard logic vector
-- Returns: 8-bit standard logic vector
--  Ussage: Y = GFLOGlut(X);
----------------------------------------------------------------------
  function GFLOGlut (A: std_logic_vector) return std_logic_vector is
    CONSTANT GFLOG: LUT := Calc_GFLOG(Calc_GFILOG);
    variable ai : std_logic_vector(7 downto 0);
  begin
    ai := A;
    return GFLOG(conv_integer(ai));
  end  GFLOGlut;


----------------------------------------------------------------------
--Function: GFILOGlut - returns the Galois Field Inverse LOG of the input
--              from a Look-Up Table
--   Input: A - 8-bit standard logic vector
-- Returns: 8-bit standard logic vector
--  Ussage: Y = GFILOGlut(X);
----------------------------------------------------------------------
  function GFILOGlut (A: std_logic_vector) return std_logic_vector is
    CONSTANT GFILOG: LUT := Calc_GFILOG;
    variable ai : std_logic_vector(7 downto 0);
  begin
    ai := A;
    return GFILOG(conv_integer(ai));
  end  GFILOGlut;


----------------------------------------------------------------------
--Function: GFM - Uses log and ilog lookup tables to compute Galois
--          Field product of X and Y.
--   Input: A, B - 8-bit standard logic vectors.
-- Returns: 8-bit standard logic vector
--  Ussage: Z = GFM(X, Y);
-----------------------------------------------------------------------
  function GFM (A, B: std_logic_vector) return std_logic_vector is
    variable  ai, bi, ci  :std_logic_vector(7 downto 0);
    variable  sum         :integer range 0 to 511 := 0;
    variable  ret         :std_logic_vector(7 downto 0);
  begin
    ai := A;
    bi := B;

    -- add the logs of the inputs
    sum := conv_integer(GFLOGlut(ai)) + conv_integer(GFLOGlut(bi));

    -- ci <= sum modulo 255
    if sum >= hi then
      ci := conv_std_logic_vector((sum - max),W);
    else
      ci := conv_std_logic_vector(sum,W);
    end if;

    -- if one of the inputs is 0 return 0, else return the inverse log of ci
    if ai = 0 or bi = 0 then
      ret := zero;
    else
      ret := GFILOGlut(ci);
    end if;

    return ret;
  end GFM;


-----------------------------------------------------------------------
--Function: GFD - Uses log and ilog lookup tables to compute Galois
--          Field quotient of X and Y.
--   Input: A, B - 8-bit standard logic vectors.
-- Returns: 8-bit standard logic vector
--  Ussage: Z = GFD(X, Y);
-----------------------------------------------------------------------
  function GFD (A, B: std_logic_vector) return std_logic_vector is
    variable  ai,bi,la,lb,ci  :std_logic_vector(7 downto 0);
    variable  ret             :std_logic_vector(7 downto 0);
  begin
    ai := A;
    bi := B;

    -- compute the logs of the inputs
    la := GFLOGlut(ai);
    lb := GFLOGlut(bi);

    -- ci <= difference modulo 255
    if la >= lb then
      ci := la - lb;
    else
```

```
      ci := la + 255 - lb;
    end if;

    -- if one of the inputs is 0 return 0, else return the inverse log of ci
    if ai = 0 or bi = 0 then
      ret := x"00";
    else
      ret := GFILOGlut(ci);
    end if;

    return ret;
  end GFD;


----------------------------------------------------------------------
--Function: GFM2 - returns the GF product of A*2 using combinational logic
--   Input: A - 8-bit standard logic vector
-- Returns: 8-bit standard logic vector
--  Ussage: Y = GFM2(X);
----------------------------------------------------------------------
  function GFM2 (A: std_logic_vector) return std_logic_vector is
    variable ai,ret  : word;
  begin
    ai := A;
    ret(7) := ai(6);
    ret(6) := ai(5);
    ret(5) := ai(4);
    ret(4) := ai(3) xor ai(7);
    ret(3) := ai(2) xor ai(7);
    ret(2) := ai(1) xor ai(7);
    ret(1) := ai(0);
    ret(0) := ai(7);

    return ret;
  end GFM2;

----------------------------------------------------------------------
--Function: GFM2w - returns the GF product of A*2 using combinational logic
--   Input: A - word
-- Returns: word
--  Ussage: Y = GFM2w(X);
----------------------------------------------------------------------
  function GFM2w (A: word) return word is
    variable ret   : word;
    variable ai    : std_logic_vector(W downto 0);
  begin
    ai := A&'0';
    if (ai(W) = '1') then
      ai := ai xor F;
    end if;
    ret := ai(W-1 downto 0);
    return ret;
  end GFM2w;
```

```
----------------------------------------------------------------------
--Function: GFM2p - returns the GF product of A*2^p using combinational logic
--   Input: A - 8-bit standard logic vector; p - integer range 0 to 7
-- Returns: 8-bit standard logic vector
--  Ussage: Y = GFM2(X,i);
----------------------------------------------------------------------
  function GFM2p (A: std_logic_vector;p: integer range 0 to W-1)
        return std_logic_vector is
    variable ret  : std_logic_vector(W-1 downto 0);
  begin
    ret := A;
    for i in 1 to W-1 loop
      exit when i>p;
      ret := GFM2(ret);
    end loop;

    return ret;
  end GFM2p;


----------------------------------------------------------------------
--Function: GFMILOGs - returns the GF product of 2^A using combinational logic
--   Input: A - 8-bit standard logic vector
-- Returns: 8-bit standard logic vector
--  Ussage: Y = GFMILOGs(X);
----------------------------------------------------------------------
  function GFMILOGs (A: std_logic_vector) return std_logic_vector is
    variable ret  : std_logic_vector(7 downto 0);
    variable p    : integer range 0 to 255;
  begin
    ret := x"01";
    p := conv_integer(A);
    for i in 1 to 255 loop
      exit when i>p;
      ret := GFM2(ret);
    end loop;

    return ret;
  end GFMILOGs;


----------------------------------------------------------------------
--Function: GFMc - Uses GFM2p to compute Galois Field product of X and Y.
--   Input: A, B - 8-bit standard logic vectors.
-- Returns: 8-bit standard logic vector
--  Ussage: Z = GFMc(X, Y);
----------------------------------------------------------------------
  function GFMc (A, B: std_logic_vector) return std_logic_vector is
    variable  ai, bi    :std_logic_vector(7 downto 0);
    variable  ret       :std_logic_vector(7 downto 0);
  begin
    ret := x"00";
    ai := A;
    bi := B;
    for i in 0 to W-1 loop
      if bi(i) = '1' then
        ret := ret xor GFM2p(ai,i);
      end if;
    end loop;

    return ret;
  end GFMc;


----------------------------------------------------------------------
--Function: GFM2pe - returns the GF product of A*2^p using combinational logic
--   Input: A - 8-bit standard logic vector; p - integer range 0 to 7
-- Returns: 8-bit standard logic vector
--  Ussage: Y = GFM2(X,i);
----------------------------------------------------------------------
  function GFM2pe (A: std_logic_vector;p: integer range 0 to 7)
        return std_logic_vector is
    type matrix is array (0 to 7) of std_logic_vector(7 downto 0);

    variable  t         :matrix;
    variable  X         :std_logic_vector(7 downto 0);
    variable  y         :integer range 0 to 7;
    variable  ret       :std_logic_vector(7 downto 0);
  begin
    X := A;
    y := p;

    t(0)(7)  := X(7);
    t(0)(6)  := X(6);
    t(0)(5)  := X(5);
    t(0)(4)  := X(4);
    t(0)(3)  := X(3);
    t(0)(2)  := X(2);
    t(0)(1)  := X(1);
    t(0)(0)  := X(0);

    t(1)(7)  := X(6);
    t(1)(6)  := X(5);
    t(1)(5)  := X(4);
    t(1)(4)  := X(3) xor X(7);
    t(1)(3)  := X(2) xor X(7);
    t(1)(2)  := X(1) xor X(7);
    t(1)(1)  := X(0);
    t(1)(0)  := X(7);

    t(2)(7)  := X(5);
    t(2)(6)  := X(4);
    t(2)(5)  := X(3) xor X(7);
    t(2)(4)  := X(2) xor X(6) xor X(7);
    t(2)(3)  := X(1) xor X(6) xor X(7);
    t(2)(2)  := X(0) xor X(6);
    t(2)(1)  := X(7);
    t(2)(0)  := X(6);
    t(3)(7)  := X(4);
    t(3)(6)  := X(3) xor X(7);
    t(3)(5)  := X(2) xor X(6) xor X(7);
    t(3)(4)  := X(1) xor X(5) xor X(6) xor X(7);
    t(3)(3)  := X(0) xor X(5) xor X(6);
    t(3)(2)  := X(5) xor X(7);
    t(3)(1)  := X(6);
    t(3)(0)  := X(5);

    t(4)(7)  := X(3) xor X(7);
    t(4)(6)  := X(2) xor X(6) xor X(7);
    t(4)(5)  := X(1) xor X(5) xor X(6) xor X(7);
    t(4)(4)  := X(0) xor X(4) xor X(5) xor X(6);
    t(4)(3)  := X(4) xor X(5) xor X(7);
    t(4)(2)  := X(4) xor X(6);
    t(4)(1)  := X(5);
    t(4)(0)  := X(4);

    t(5)(7)  := X(2) xor X(6) xor X(7);
    t(5)(6)  := X(1) xor X(5) xor X(6) xor X(7);
    t(5)(5)  := X(0) xor X(4) xor X(5) xor X(6);
    t(5)(4)  := X(3) xor X(4) xor X(5);
    t(5)(3)  := X(3) xor X(4) xor X(6) xor X(7);
    t(5)(2)  := X(3) xor X(5) xor X(7);
    t(5)(1)  := X(4);
    t(5)(0)  := X(3) xor X(7);


    t(6)(7)  := X(1) xor X(5) xor X(6) xor X(7);
    t(6)(6)  := X(0) xor X(4) xor X(5) xor X(6);
    t(6)(5)  := X(3) xor X(4) xor X(5);
    t(6)(4)  := X(2) xor X(3) xor X(4);
    t(6)(3)  := X(2) xor X(3) xor X(5) xor X(6);
    t(6)(2)  := X(2) xor X(4) xor X(6) xor X(7);
    t(6)(1)  := X(3) xor X(7);
    t(6)(0)  := X(2) xor X(6) xor X(7);

    t(7)(7)  := X(0) xor X(4) xor X(5) xor X(6);
    t(7)(6)  := X(3) xor X(4) xor X(5);
    t(7)(5)  := X(2) xor X(3) xor X(4);
    t(7)(4)  := X(1) xor X(2) xor X(3) xor X(7);
    t(7)(3)  := X(1) xor X(2) xor X(4) xor X(5);
    t(7)(2)  := X(1) xor X(3) xor X(5) xor X(6);
    t(7)(1)  := X(2) xor X(6) xor X(7);
    t(7)(0)  := X(1) xor X(5) xor X(6) xor X(7);

    ret := t(y);

    return ret;
  end GFM2pe;


----------------------------------------------------------------------
```

```
--Function: GFMe - Uses direct combinational logic to compute the Galois
--          Field product of X and Y.
--   Input: A, B - 8-bit standard logic vectors.
-- Returns: 8-bit standard logic vector
--  Ussage: Z = GFMe(X, Y);
----------------------------------------------------------------------
  function GFMe (A, B: std_logic_vector) return std_logic_vector is
    type matrix is array (0 to 7) of std_logic_vector(7 downto 0);

    variable  y        :matrix;
    variable  X, bi    :std_logic_vector(7 downto 0);
    variable  ret      :std_logic_vector(7 downto 0);
  begin
    ret := x"00";
    X := A;
    bi := B;

    y(0)(7)  := X(7);
    y(0)(6)  := X(6);
    y(0)(5)  := X(5);
    y(0)(4)  := X(4);
    y(0)(3)  := X(3);
    y(0)(2)  := X(2);
    y(0)(1)  := X(1);
    y(0)(0)  := X(0);

    y(1)(7)  := X(6);
    y(1)(6)  := X(5);
    y(1)(5)  := X(4);
    y(1)(4)  := X(3) xor X(7);
    y(1)(3)  := X(2) xor X(7);
    y(1)(2)  := X(1) xor X(7);
    y(1)(1)  := X(0);
    y(1)(0)  := X(7);

    y(2)(7)  := X(5);
    y(2)(6)  := X(4);
    y(2)(5)  := X(3) xor X(7);
    y(2)(4)  := X(2) xor X(6) xor X(7);
    y(2)(3)  := X(1) xor X(6) xor X(7);
    y(2)(2)  := X(0) xor X(6);
    y(2)(1)  := X(7);
    y(2)(0)  := X(6);

    y(3)(7)  := X(4);
    y(3)(6)  := X(3) xor X(7);
    y(3)(5)  := X(2) xor X(6) xor X(7);
    y(3)(4)  := X(1) xor X(5) xor X(6) xor X(7);
    y(3)(3)  := X(0) xor X(5) xor X(6);
    y(3)(2)  := X(5) xor X(7);
    y(3)(1)  := X(6);
    y(3)(0)  := X(5);

    y(4)(7)  := X(3) xor X(7);
    y(4)(6) := X(2) xor X(6) xor X(7);
    y(4)(5) := X(1) xor X(5) xor X(6) xor X(7);
    y(4)(4) := X(0) xor X(4) xor X(5) xor X(6);
    y(4)(3) := X(4) xor X(5) xor X(7);
    y(4)(2) := X(4) xor X(6);
    y(4)(1) := X(5);
    y(4)(0) := X(4);

    y(5)(7) := X(2) xor X(6) xor X(7);
    y(5)(6) := X(1) xor X(5) xor X(6) xor X(7);
    y(5)(5) := X(0) xor X(4) xor X(5) xor X(6);
    y(5)(4) := X(3) xor X(4) xor X(5);
    y(5)(3) := X(3) xor X(4) xor X(6) xor X(7);
    y(5)(2) := X(3) xor X(5) xor X(7);
    y(5)(1) := X(4);
    y(5)(0) := X(3) xor X(7);


    y(6)(7) := X(1) xor X(5) xor X(6) xor X(7);
    y(6)(6) := X(0) xor X(4) xor X(5) xor X(6);
    y(6)(5) := X(3) xor X(4) xor X(5);
    y(6)(4) := X(2) xor X(3) xor X(4);
    y(6)(3) := X(2) xor X(3) xor X(5) xor X(6);
    y(6)(2) := X(2) xor X(4) xor X(6) xor X(7);
    y(6)(1) := X(3) xor X(7);
    y(6)(0) := X(2) xor X(6) xor X(7);

    y(7)(7) := X(0) xor X(4) xor X(5) xor X(6);
    y(7)(6) := X(3) xor X(4) xor X(5);
    y(7)(5) := X(2) xor X(3) xor X(4);
    y(7)(4) := X(1) xor X(2) xor X(3) xor X(7);
    y(7)(3) := X(1) xor X(2) xor X(4) xor X(5);
    y(7)(2) := X(1) xor X(3) xor X(5) xor X(6);
    y(7)(1) := X(2) xor X(6) xor X(7);
    y(7)(0) := X(1) xor X(5) xor X(6) xor X(7);


    for i in 0 to 7 loop
      if bi(i) = ’1’ then
        ret := ret xor y(i);
      else
        ret := ret xor x"00";
      end if;
    end loop;

    return ret;
  end GFMe;


------------------------------------------------------------------------
--Function: GFD2 - returns the GF quotient of A/2 using combinational logic
--   Input: A - 8-bit standard logic vector
-- Returns: 8-bit standard logic vector
--  Ussage: Y = GFD2(X);
```

```
------------------------------------------------------------------------
  function GFD2 (A: std_logic_vector) return std_logic_vector is
    variable ai,ret  : std_logic_vector(7 downto 0);
  begin
    ai := A;
    ret(7) := ai(0);
    ret(6) := ai(7);
    ret(5) := ai(6);
    ret(4) := ai(5);
    ret(3) := ai(4) xor ai(0);
    ret(2) := ai(3) xor ai(0);
    ret(1) := ai(2) xor ai(0);
    ret(0) := ai(1);

    return ret;
  end GFD2;

------------------------------------------------------------------------
--Function: GFD2p - returns the GF quotient of A/2^p using combinational logic
--   Input: A - 8-bit standard logic vector; p - integer range 0 to 7
-- Returns: 8-bit standard logic vector
--  Ussage: Y = GFD2p(X,i);
------------------------------------------------------------------------
  function GFD2p (A: std_logic_vector;p: integer range 0 to 7)
        return std_logic_vector is
    variable ret     : std_logic_vector(7 downto 0);
  begin
    ret := A;
    for i in 1 to 7 loop
      exit when i>p;
      ret := GFD2(ret);
    end loop;

    return ret;
  end GFD2p;


------------------------------------------------------------------------
--Function: GFIc - returns the GF Multiplicative Inverse using combinational logic
--   Input: A - 8-bit standard logic vector
-- Returns: 8-bit standard logic vector
--  Ussage: Y = GFIc(X);
------------------------------------------------------------------------
  function GFIc (A: std_logic_vector) return std_logic_vector is
    variable ret,p1,p2,p3,p6   : std_logic_vector(7 downto 0);
    variable p12,p14,p15,p240  : std_logic_vector(7 downto 0);
  begin
    p1 := A;
    p2 := GFMe(p1,p1);
    p3 := GFMe(p2,p1);
    p6 := GFMe(p3,p3);
    p12 := GFMe(p6,p6);
    p14 := GFMe(p12,p2);
    p15 := GFMe(p12,p3);
    p240 := p15;
    for i in 1 to 4 loop
      p240 := GFMe(p240,p240);
    end loop;

    ret := GFMe(p240,p14);
    return ret;
  end GFIc;

-----------------------------------------------------------------------
--Function: GFIl - returns the GF Multiplicative Inverse using table lookups
--   Input: A - 8-bit standard logic vector
-- Returns: 8-bit standard logic vector
--  Ussage: Y = GFI(X);
-----------------------------------------------------------------------
  function GFIl (A: std_logic_vector) return std_logic_vector is
    variable ret   : std_logic_vector(7 downto 0);
  begin
    ret := GFD(X"01",A);
    return ret;
  end GFIl;


-----------------------------------------------------------------------
--Function: GFIei - returns the GF Multiplicative Inverse using Euclidean Alg
--   Input: A - 8-bit standard logic vector
-- Returns: 8-bit standard logic vector
--  Ussage: Y = GFIe(X);
-----------------------------------------------------------------------
  function GFIei (A: std_logic_vector) return std_logic_vector is
    constant  M    : integer := 8;
    constant  F    : std_logic_vector(8 downto 0) := "100011101";
    variable   s,r,t  : std_logic_vector(8 downto 0);
    variable  v,u,w  : std_logic_vector(7 downto 0);
    variable  d    : integer range -8 to 8;
  begin
    s := F;
    r := '0'&A;
    v := (others => '0');
    u := (0 => '1',others => '0');
    d := 0;
    for i in 1 to 2*M loop
      if r(8) = '1' and s(8) = '1' then
        t := s xor r;
        w := v xor u;
      else
        t := s;
        w := v;
      end if;

      if r(8) = '0' then
        s := t;
        r := r(7 downto 0)&'0';
        v := w;
        u := GFM2(u);
        d := d+1;
      else
        if d = 0 then
          s := r;
          r := t(7 downto 0)&'0';
          v := u;
          u := GFM2(w);
          d := d+1;
        else
          s := t(7 downto 0)&'0';
          r := r;
          v := w;
          u := GFD2(u);
          d := d-1;
        end if;
      end if;
    end loop;
    return u;
  end GFIei;

-----------------------------------------------------------------------
--Function: RAID6CompC1 - Precomputes constant 1 for RAID error correction
--   Input: x,y - integer range 0 to 7
-- Returns: word
--  Usage: Y = RAID6CompC1(a,b);
-----------------------------------------------------------------------
  function RAID6CompC1 (c1,c2: integer range 0 to N-1) return word is
    variable  x,y    : integer range 0 to N-1;
    variable  a,c,d  : word;
  begin
    x := c1;
    y := c2;
    a := x"01";
    a := GFM2p(a,y-x);
    c := a xor x"01";
    d := GFD(a,c);

    return d;
  end RAID6CompC1;


-----------------------------------------------------------------------
--Function: RAID6CompC2 - Precomputes constant 2 for RAID error correction
--   Input: x,y - integer range 0 to 7
-- Returns: word
--  Usage: Y = RAID6CompC2(a,b);
-----------------------------------------------------------------------
  function RAID6CompC2 (c1,c2: integer range 0 to N-1) return word is
    variable  x,y    : integer range 0 to N-1;
    variable  a,b,c,d  : word;
  begin
    x := c1;
    y := c2;
    a := x"01";
    a := GFM2p(a,y-x);
    c := a xor x"01";
    b := x"01";
    b := GFD2p(b,x);
    d := GFD(b,c);

    return d;
  end RAID6CompC2;


----------------------------------------------------------------------
--Function: RAID6c2e - uses the parity information and precomputed constants
--            to correct 2 known data errors
--   Input: A - stripe w/ parity; B,C - W-bit word;
--      x,y - integer range 0 to 7
-- Returns: stripe w/ parity
--  Usage: Y = RAID6c2e(X,A,B,l,k);
----------------------------------------------------------------------
  function RAID6c2e (A: stripewp;B,C: word;x,y: integer) return stripewp is

    variable  sout        : stripewp;
    variable  data        : stripe;
    variable  Pi,Qi,c1,c2    : word;
    variable  Px,Qx      : word;
    variable  PQ        : parity;
    variable  p1,p2      : integer range 0 to N+1;

  begin
    c1 := B;
    c2 := C;

    p1 := x;
    p2 := y;

    sout := A;
    for i in data'range loop
      data(i) := sout(i);
    end loop;
    Pi := A(N);               -- get initial P
    Qi := A(N+1);               -- get initial Q

    data(p1) := (others => '0');        -- clear out bad data byte
    data(p2) := (others => '0');        -- clear out bad data byte
    PQ := RAIDenc(data);          -- compute new P and Q

    -- use new P and Q to compute difference from original P and Q
    Qx := PQ(1) xor Qi;
    Px := PQ(0) xor Pi;

    -- find correct data values
    sout(p1) := GFMe(c1,Px) xor GFMe(c2,Qx);
    sout(p2) := Px xor sout(p1);
    return sout;
  end RAID6c2e;


----------------------------------------------------------------------
--Function: divby_2pxor1 - returns input/(2^power xor 1)
--   Input: input - W-bit word, power - integer range 1 to 7
-- Returns: W-bit word
--  Ussage: Y = divby_2pxor1(X,a);
----------------------------------------------------------------------
  function divby_2pxor1 (input: word; power:integer range 1 to 7) return word is
    type divfunc is array (7 downto 1) of word;
    variable func    : divfunc;
    variable a    : word;
    variable p    : integer range 1 to 7;
  begin
    a := input;
    p := power;

    func(1)(0) := a(1) xor a(2) xor a(3) xor a(4) xor
                    a(5) xor a(6) xor a(7);
    func(1)(1) := a(2) xor a(3) xor a(4) xor a(5) xor
                   a(6) xor a(7);
    func(1)(2) := a(0) xor a(1) xor a(2);
    func(1)(3) := a(4) xor a(5) xor a(6) xor a(7);
    func(1)(4) := a(0) xor a(1) xor a(2) xor a(3) xor a(4);
    func(1)(5) := a(0) xor a(1) xor a(2) xor a(3) xor
                   a(4) xor a(5);
    func(1)(6) := a(0) xor a(1) xor a(2) xor a(3) xor
                   a(4) xor a(5) xor a(6);
    func(1)(7) := a(0) xor a(1) xor a(2) xor a(3) xor
                   a(4) xor a(5) xor a(6) xor a(7);

    func(2)(0) := a(0) xor a(1) xor a(3) xor a(5) xor a(7);
    func(2)(1) := a(0) xor a(1) xor a(2) xor a(4) xor a(6);
    func(2)(2) := a(0) xor a(2);
    func(2)(3) := a(5) xor a(7);
    func(2)(4) := a(1) xor a(3) xor a(5) xor a(6) xor a(7);
    func(2)(5) := a(0) xor a(2) xor a(4) xor a(6) xor a(7);
    func(2)(6) := a(1) xor a(3) xor a(5) xor a(7);
    func(2)(7) := a(0) xor a(2) xor a(4) xor a(6);

    func(3)(0) := a(0) xor a(1) xor a(4) xor a(7);
    func(3)(1) := a(1) xor a(2) xor a(5);
    func(3)(2) := a(0) xor a(1) xor a(2) xor a(3) xor
                   a(4) xor a(6) xor a(7);
    func(3)(3) := a(0) xor a(2) xor a(3) xor a(5);
    func(3)(4) := a(0) xor a(3) xor a(6) xor a(7);
    func(3)(5) := a(1) xor a(4) xor a(7);
    func(3)(6) := a(2) xor a(5);
    func(3)(7) := a(0) xor a(3) xor a(6);

    func(4)(0) := a(2) xor a(3) xor a(4) xor a(6) xor a(7);
    func(4)(1) := a(0) xor a(3) xor a(4) xor a(5) xor a(7);
    func(4)(2) := a(1) xor a(2) xor a(3) xor a(5) xor a(7);
    func(4)(3) := a(7);
    func(4)(4) := a(0) xor a(2) xor a(3) xor a(4) xor a(6) xor a(7);
    func(4)(5) := a(0) xor a(1) xor a(3) xor a(4) xor a(5) xor a(7);
    func(4)(6) := a(0) xor a(1) xor a(2) xor a(4) xor a(5) xor a(6);
    func(4)(7) := a(1) xor a(2) xor a(3) xor a(5) xor a(6) xor a(7);

    func(5)(0) := a(0) xor a(1) xor a(2) xor a(3) xor a(6) xor a(7);
    func(5)(1) := a(1) xor a(2) xor a(3) xor a(4) xor a(7);
    func(5)(2) := a(0) xor a(1) xor a(4) xor a(5) xor a(6) xor a(7);
    func(5)(3) := a(0) xor a(3) xor a(5);
    func(5)(4) := a(2) xor a(3) xor a(4) xor a(7);
    func(5)(5) := a(0) xor a(3) xor a(4) xor a(5);
    func(5)(6) := a(0) xor a(1) xor a(4) xor a(5) xor a(6);
    func(5)(7) := a(0) xor a(1) xor a(2) xor a(5) xor a(6) xor a(7);

    func(6)(0) := a(0) xor a(2) xor a(4) xor a(5);
    func(6)(1) := a(0) xor a(1) xor a(3) xor a(5) xor a(6);
    func(6)(2) := a(0) xor a(1) xor a(5) xor a(6) xor a(7);
    func(6)(3) := a(0) xor a(1) xor a(4) xor a(5) xor a(6) xor a(7);
    func(6)(4) := a(0) xor a(1) xor a(4) xor a(6) xor a(7);
    func(6)(5) := a(1) xor a(2) xor a(5) xor a(7);
    func(6)(6) := a(0) xor a(2) xor a(3) xor a(6);
    func(6)(7) := a(1) xor a(3) xor a(4) xor a(7);

    func(7)(0) := a(2) xor a(4) xor a(7);
    func(7)(1) := a(3) xor a(5);
    func(7)(2) := a(0) xor a(2) xor a(6) xor a(7);
    func(7)(3) := a(1) xor a(2) xor a(3) xor a(4);
    func(7)(4) := a(0) xor a(3) xor a(5) xor a(7);
    func(7)(5) := a(1) xor a(4) xor a(6);
    func(7)(6) := a(0) xor a(2) xor a(5) xor a(7);
    func(7)(7) := a(1) xor a(3) xor a(6);

    return func(p);

  end divby_2pxor1;


----------------------------------------------------------------------
--Function: RAIDenc - returns the RAID6 parity of the byte stripe
--   Input: A - stripe
-- Returns: parity (2 words)
--  Ussage: Y = GFIe(X);
----------------------------------------------------------------------
  function RAIDenc (A: stripe) return parity is
    variable   sin    : stripe;
    variable  PQ    : parity;

  begin
    sin := A;
    PQ(1) := (others => ’0’);
    PQ(0) := (others => ’0’);
    for i in 0 to N-1 loop
      PQ(0) := PQ(0) xor sin(i);
      PQ(1) := PQ(1) xor GFM2pe(sin(i),i);
    end loop;
    return PQ;
  end RAIDenc;


------------------------------------------------------------------------
--Function: RAID6c2ecomb - uses the parity information
--              to correct 2 known data errors
--   Input: A - stripe w/ parity; x,y - integer range 0 to 7
-- Returns: stripe w/ parity
--  Usage: Y = RAID6c2ecomb(X,l,k);
------------------------------------------------------------------------
  function RAID6c2ecomb (A: stripewp;x,y: integer) return stripewp is

    variable  sout      : stripewp;
    variable  data      : stripe;
    variable  Pi,Qi      : word;
    variable  Px,Qx      : word;
    variable  PQ        : parity;
    variable  p1,p2,diff    : integer range 0 to N+1;

  begin
    p1 := x;
    p2 := y;

    sout := A;
    for i in data'range loop
      data(i) := sout(i);
    end loop;
    Pi := A(N);                 -- get initial P
    Qi := A(N+1);               -- get initial Q

    data(p1) := (others => '0');       -- clear out bad data byte
    data(p2) := (others => '0');       -- clear out bad data byte
    PQ := RAIDenc(data);           -- compute new P and Q

    -- use new P and Q to compute difference from original P and Q
    Qx := PQ(1) xor Qi;
    Px := PQ(0) xor Pi;

    -- find correct data values
    diff := p2-p1;
    sout(p1) := divby_2pxor1(GFD2p(Qx,p1) xor GFM2p(Px,diff),diff);
    sout(p2) := Px xor sout(p1);

    return sout;
  end RAID6c2ecomb;


------------------------------------------------------------------------
--Function: RAID6c1 - uses the parity information find and
```

```
--              correct 1 unknown data error
--   Input: A - (N+2)-byte standard logic vector,
--          B - (N+2)-bit standard logic vector
-- Returns: N-byte standard logic vector
--  Ussage: Y = GFIe(X,Y);
----------------------------------------------------------------------
  function RAID6c1 (A: stripewp) return stripewp is

    variable  sout        : stripewp;
    variable  data        : stripe;
    variable  Pi,Qi      : word;
    variable  Px,Qx,Pd,Qd    : word;
    variable  PQ        : parity;
    variable  z        : integer range 0 to 2**W-1;

  begin
    sout := A;
    for i in data'range loop
      data(i) := sout(i);
    end loop;
    Pi := A(N);               -- get initial P
    Qi := A(N+1);               -- get initial Q

    PQ := RAIDenc(data);          -- compute new P and Q

    -- use new P and Q to compute difference from original P and Q
    Qx := PQ(1) xor Qi;
    Px := PQ(0) xor Pi;

    Pd := Px xor Pi;
    Qd := Qx xor Qi;

    z := conv_integer(GFLOGlut(GFD(Qx,Px)));

    if Pd = zero and Qd >= one then
      sout(N+1) := Qx;
    elsif Qd = zero and not(Pd = zero) then
      sout(N) := Px;
    elsif not(Qd = zero) and not(Pd = zero) then
      if z >=0 and z < N then
        data(z) := (others => '0');
        PQ := RAIDenc(data);          -- compute new P and Q
        Px := PQ(0) xor Pi;
        sout(z) := Pi xor Px;
      end if;
    end if;

    return sout;
  end RAID6c1;

  ----------------------------------------------------------------------
  --Function: RAIDencDWstripe - returns the RAID6 parity of
  --              the Double Word stripe
  --   Input: A - DWstripe
  -- Returns: DWparity (2 words)
  --  Ussage: Y = GFIe(X);
  -----------------------------------------------------------------------
    function RAIDencDWstripe (A: DWstripe) return DWparity is
      variable   sin    : DWstripe;
      variable  PQ    : DWparity;
      variable  bytestripe : stripe;
      variable  byteparity : parity;

    begin
      sin := A;
      for i in 0 to WpDW-1 loop
        for j in 0 to N-1 loop
          bytestripe(j) := sin(j)(((i*W)+(W-1)) downto (i*W));
        end loop;
        byteparity := RAIDenc(bytestripe);
        PQ(0)(((i*W)+(W-1)) downto (i*W)) := byteparity(0);
        PQ(1)(((i*W)+(W-1)) downto (i*W)) := byteparity(1);
      end loop;
      return PQ;
    end RAIDencDWstripe;


end pkg_RAID6;
```

Appendix D

GF Algebra and RAID-6 Library Components

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.all;
use pkg_RAID6.all;
------------------------------------------------------------------------
-- Entity: GFILog_BRAM - computes GF Inverse Log (exponentiation) of inputs
------------------------------------------------------------------------
entity GFILog_BRAM is
  port (
      clk : in  std_logic;
      I1  : in  word;
      O1  : out word;
      I2  : in  word;
      O2  : out word
  );
end GFILog_BRAM;

architecture arch of GFILog_BRAM is
  CONSTANT GFILOG: LUT := Calc_GFILOG;
  attribute ROM_STYLE : string;
  attribute ROM_STYLE of GFILOG: constant is "BLOCK";

begin
  process(clk)
  begin
    if clk'event and clk = '1' then
      O1 <= GFILOG(conv_integer(I1));
      O2 <= GFILOG(conv_integer(I2));
    end if;
  end process;
end architecture;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.all;
use pkg_RAID6.all;
-----------------------------------------------------------------------
-- Entity: GFLog_BRAM - computes GF Log of inputs
-----------------------------------------------------------------------
entity GFLog_BRAM is
  port   (
      clk : in  std_logic;
      I1  : in  word;
      O1  : out word;
      I2  : in  word;
      O2  : out word
  );
end GFLog_BRAM;

architecture arch of GFLog_BRAM is
  CONSTANT GFLOG: LUT := Calc_GFLOG(Calc_GFILOG);
  attribute ROM_STYLE : string;
  attribute ROM_STYLE of GFLOG: constant is "BLOCK";

begin
  process(clk)
  begin
    if clk'event and clk = '1' then
      O1 <= GFLOG(conv_integer(I1));
      O2 <= GFLOG(conv_integer(I2));
    end if;
  end process;
end architecture;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.all;
use pkg_RAID6.all;
-----------------------------------------------------------------------
-- Entity: GFM_BRAM - computes prod, the GF product of a and b
-----------------------------------------------------------------------
entity GFM_BRAM is
  port   (
      clk   : in  std_logic;
      a     : in  word;
      b     : in  word;
      prod  : out word
  );
end GFM_BRAM;

architecture arch of GFM_BRAM is

  component GFLog_BRAM is
  port   (
      clk : in  std_logic;
```

```
        I1  : in  word;
        O1  : out word;
        I2  : in  word;
        O2  : out word
    );
    end component;

    component GFILog_BRAM is
    port   (
        clk : in  std_logic;
        I1  : in  word;
        O1  : out word;
        I2  : in  word;
        O2  : out word
    );
    end component;

    signal loga,logb,c  : word;


begin
  u1: GFLog_BRAM port map(clk,a,loga,b,logb);
  u2: GFILog_BRAM port map(clk,c,prod,(others => '0'), open);

  process(clk)
    variable sum :integer range 0 to 2**(W+1)-1;
  begin
    if clk'event and clk = '1' then
      -- add the logs of the inputs
      sum := conv_integer(loga) + conv_integer(logb);

      -- c <= sum modulo (2^w)-1
      if sum >= max then
        c <= conv_std_logic_vector((sum - max),W);
      else
        c <= conv_std_logic_vector(sum,W);
      end if;

    end if;
  end process;
end architecture;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.all;
use pkg_RAID6.all;
-----------------------------------------------------------------------
-- Entity: GFD_BRAM - computes quot, the GF quotient of a and b
-----------------------------------------------------------------------
entity GFD_BRAM is
  port   (
      clk   : in  std_logic;
      a     : in  word;
      b     : in  word;
      quot  : out word
  );
end GFD_BRAM;

architecture arch of GFD_BRAM is

  component GFLog_BRAM is
  port   (
      clk : in  std_logic;
      I1  : in  word;
      O1  : out word;
      I2  : in  word;
      O2  : out word
  );
  end component;

  component GFILog_BRAM is
  port   (
      clk : in  std_logic;
      I1  : in  word;
      O1  : out word;
      I2  : in  word;
      O2  : out word
  );
  end component;

  signal loga,logb,c  : word;


begin
  u1: GFLog_BRAM port map(clk,a,loga,b,logb);
  u2: GFILog_BRAM port map(clk,c,quot,(others => '0'), open);

  process(clk)
  begin
    if clk'event and clk = '1' then
      -- ci <= difference modulo 255
      if loga >= logb then
        c <= loga - logb;
      else
        c <= loga + max - logb;
      end if;
    end if;
  end process;
end architecture;


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.all;
use pkg_RAID6.all;
-----------------------------------------------------------------------
-- Entity: GFI_BRAM - computes inv, the GF multiplicative inverse of a
-----------------------------------------------------------------------
entity GFI_BRAM is
  port   (
      clk : in  std_logic;
      a   : in  word;
      inv : out word
  );
end GFI_BRAM;

architecture arch of GFI_BRAM is

  component GFLog_BRAM is
  port   (
      clk : in  std_logic;
      I1  : in  word;
      O1  : out word;
      I2  : in  word;
      O2  : out word
  );
  end component;

  component GFILog_BRAM is
  port   (
      clk : in  std_logic;
      I1  : in  word;
      O1  : out word;
      I2  : in  word;
      O2  : out word
  );
  end component;

  signal loga, c  : word;


begin
  u1: GFLog_BRAM port map(clk,a,loga,(others => '0'), open);
  u2: GFILog_BRAM port map(clk,c,inv,(others => '0'), open);

  process(clk)
  begin
    if clk'event and clk = '1' then
      c <= hi - loga;
    end if;
  end process;
end architecture;


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.all;
use pkg_RAID6.all;
-----------------------------------------------------------------------
-- Entity: GFM_LFSR - multiplies a times 2 ^ logb
--  load - loads in a and logb when high, starts calculations when goes low
--  done - signals that the product is valid
-----------------------------------------------------------------------
entity GFM_LFSR is
  port   (
      clk   : in  std_logic;
      load  : in  std_logic;
      a     : in  word;
      b     : in  word;
      done  : out  std_logic;
      prod  : out word := (others => '0')
  );
end GFM_LFSR;

architecture arch of GFM_LFSR is
  signal lfsr,p  : word := (others => '0');
  signal d  : std_logic := '0';
  signal c  : integer range 0 to W := 0;
begin
  d <= '1' when c >= W else '0';
  done <= d;
  prod <= p;
  process(clk)
  begin
    if clk'event and clk = '1' then
      if load = '1' then
        lfsr <= a;
        c <= 0;
        p <= (others => '0');
      elsif d = '0' then
        lfsr <= GFM2(lfsr);
        c <= c + 1;
        if b(c) = '1' then
          p <= p xor lfsr;
        end if;
      end if;
    end if;
  end process;
end architecture;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.all;
use pkg_RAID6.all;
-----------------------------------------------------------------------
-- Entity: GFI_Brute - finds multiplicative inverse of a by brute force search
--  load - loads in a and logb when high, starts calculations when goes low
--  done - signals that the output is valid
```

```
----------------------------------------------------------------------
entity GFI_Brute is
  port   (
      clk  : in  std_logic;
      load : in  std_logic;
      a    : in  word;
      done : out std_logic;
      inv  : out word := (others => '0')
  );
end GFI_Brute;

architecture arch of GFI_Brute is
  signal p,i  : word := (others => '0');
  signal d    : std_logic := '0';
begin
  p <= GFMe(a,i);
  d <= '1' when conv_integer(p) = 1 else '0';
  done <= d;
  inv <= i;
  process(clk)
  begin
    if clk'event and clk = '1' then
      if load = '1' then
        i <= (others => '0');
      elsif d = '0' then
        i <= i + 1;
      end if;
    end if;
  end process;
end architecture;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.all;
use pkg_RAID6.all;
----------------------------------------------------------------------
-- Entity: GFI_Euclid - finds multiplicative inverse of a using
--              Modified Euclidean Algorithm
--  load - loads in a and logb when high, starts calculations when goes low
--  done - signals that the output is valid
----------------------------------------------------------------------
entity GFI_Euclid is
  port   (
      clk  : in  std_logic;
      load : in  std_logic;
      a    : in  word;
      done : out std_logic;
      inv  : out word := (others => '0')
  );
end GFI_Euclid;

architecture arch of GFI_Euclid is
```

```
  signal X,Xn    : word;
  signal R,Rn    : std_logic_vector(W downto 0);
  signal delta   : integer range 0 to W-1 := 0;
  signal cnt     : integer range 0 to 2*W := 0;
begin
  process(clk)
    variable V  : word;
    variable T  : std_logic_vector(W downto 0);
  begin
    if clk'event and clk = '1' then
      if load = '1' then
        Rn <= F;
        R  <= '0' &a;
        Xn <= (others => '0');
        X  <= (0 => '1', others => '0');
        delta <= 0;
        cnt <= 2*W;
      elsif cnt >= 1 then
        cnt <= cnt - 1;
        if R(W) = '1' and Rn(W) = '1' then
          T := R xor Rn;
          V := X xor Xn;
        else
          T := Rn;
          V := Xn;
        end if;

        if R(W) = '0' then
          Rn <= T;
          R <= R(W-1 downto 0)&'0';
          Xn <= V;
          X <= GFM2(X);
          delta <= delta + 1;
        else
          if delta = 0 then
            Rn <= R;
            R <= T(W-1 downto 0)&'0';
            Xn <= X;
            X <= GFM2(V);
            delta <= delta + 1;
          else
            Rn <= T(W-1 downto 0)&'0';
            R <= R;
            Xn <= V;
            X <= GFD2(X);
            delta <= delta - 1;
          end if;
        end if;
      end if;
    end if;
  end process;

  inv <= X;
  done <= '1' when cnt = 0 else '0';
```

```
end architecture;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.all;
use pkg_RAID6.all;
-----------------------------------------------------------------------
-- Entity: GFM_BRAM_double - computes 2 prod, the GF product of a and b with
--              one lookup table
-----------------------------------------------------------------------
entity GFM_BRAM_double is
  port   (
      clk    : in  std_logic;
      a1     : in  word;
      logb1  : in  word;
      prod1  : out word;
      a2     : in  word;
      logb2  : in  word;
      prod2  : out word
  );
end GFM_BRAM_double;

architecture arch of GFM_BRAM_double is

  component GFLog_BRAM is
  port   (
      clk : in  std_logic;
      I1  : in  word;
      O1  : out word;
      I2  : in  word;
      O2  : out word
  );
  end component;

  component GFILog_BRAM is
  port   (
      clk : in  std_logic;
      I1  : in  word;
      O1  : out word;
      I2  : in  word;
      O2  : out word
  );
  end component;

  signal loga1,loga2,c1,c2  : word;

begin
  u1: GFLog_BRAM port map(clk,a1,loga1,a2,loga2);
  u2: GFILog_BRAM port map(clk,c1,prod1,c2,prod2);

  process(clk)
```

```
    variable sum1, sum2 :integer range 0 to 2**(W+1)-1;
  begin
    if clk'event and clk = '1' then
      -- add the logs of the inputs
      sum1 := conv_integer(loga1) + conv_integer(logb1);

      -- c <= sum modulo (2^w)-1
      if sum1 >= max then
        c1 <= conv_std_logic_vector((sum1 - max),W);
      else
        c1 <= conv_std_logic_vector(sum1,W);
      end if;

      -- add the logs of the inputs
      sum2 := conv_integer(loga2) + conv_integer(logb2);

      -- c <= sum modulo (2^w)-1
      if sum2 >= max then
        c2 <= conv_std_logic_vector((sum2 - max),W);
      else
        c2 <= conv_std_logic_vector(sum2,W);
      end if;

    end if;
  end process;
end architecture;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.all;
use pkg_RAID6.all;
----------------------------------------------------------------------
-- Entity: GFD_BRAM_double - computes quot, the GF quotient of a and b twice
--             using one table
----------------------------------------------------------------------
entity GFD_BRAM_double is
  port   (
      clk    : in  std_logic;
      a1     : in  word;
      logb1  : in  word;
      quot1  : out word;
      a2     : in  word;
      logb2  : in  word;
      quot2  : out word
  );
end GFD_BRAM_double;

architecture arch of GFD_BRAM_double is

  component GFLog_BRAM is
  port   (
      clk : in  std_logic;
      I1  : in  word;
      O1  : out word;
      I2  : in  word;
      O2  : out word
  );
  end component;

  component GFILog_BRAM is
  port   (
      clk : in  std_logic;
      I1  : in  word;
      O1  : out word;
      I2  : in  word;
      O2  : out word
  );
  end component;

  signal loga1,loga2,c1,c2  : word;

begin
  u1: GFLog_BRAM port map(clk,a1,loga1,a2,loga2);
  u2: GFILog_BRAM port map(clk,c1,quot1,c2,quot2);


  process(clk)
  begin
    if clk'event and clk = '1' then
      -- ci <= difference modulo 255
      if loga1 >= logb1 then
        c1 <= loga1 - logb1;
      else
        c1 <= loga1 + max - logb1;
      end if;

      -- ci <= difference modulo 255
      if loga2 >= logb2 then
        c2 <= loga2 - logb2;
      else
        c2 <= loga2 + max - logb2;
      end if;
    end if;
  end process;
end architecture;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.all;
use pkg_RAID6.all;
----------------------------------------------------------------------
-- Entity: RAID6en_bram - finds the P and Q values of stripe a
----------------------------------------------------------------------
entity RAID6en_bram is
```

```
  port   (
      clk : in  std_logic;
      a   : in stripe;
      P   : out word;
      Q   : out word
  );
end RAID6en_bram;

architecture arch of RAID6en_bram is
  component GFM_BRAM_double is
    port   (
        clk    : in  std_logic;
        a1     : in  word;
        logb1  : in  word;
        prod1  : out word;
        a2     : in  word;
        logb2  : in  word;
        prod2  : out word
    );
  end component;


  signal  multiplied  : stripe;
begin

  multiplying: for i in 0 to (N-1)/2 generate
    multiplier: GFM_BRAM_double
      port map(
        clk    => clk,
        a1    => a(2*i),
        logb1  => conv_std_logic_vector(2*i,W),
        prod1  => multiplied(2*i),
        a2    => a(2*i+1),
        logb2  => conv_std_logic_vector(2*i+1,W),
        prod2  => multiplied(2*i+1)
      );
  end generate multiplying;

  process(multiplied,a)
    variable  temp_p,temp_q : word;
  begin
    temp_p := (others => '0');
    temp_q := (others => '0');
    for i in a'range loop
      temp_p := temp_p xor a(i);
      temp_q := temp_q xor multiplied(i);
    end loop;
    P <= temp_p;
    Q <= temp_q;
  end process;
end architecture;


library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.all;
use pkg_RAID6.all;
------------------------------------------------------------------------
-- Entity: RAID6c2_bram - fixes errors at locations l and k (assumes l>k)
--D_L = {(P oplus P') oplus 2^K times (Q oplus Q')} over {2^{L-K} oplus 1} newline
--D_K = P oplus P' oplus D_L

------------------------------------------------------------------------
entity RAID6c2_bram is
  port   (
      clk : in  std_logic;
      a   : in stripe;
      P   : in  word;
      Q   : in  word;
      l   : in integer;
      k   : in integer;
      fixed : out stripe
  );
end RAID6c2_bram;

architecture arch of RAID6c2_bram is
  component RAID6en_bram is
    port   (
        clk : in  std_logic;
        a   : in stripe;
        P   : out word;
        Q   : out word
    );
  end component;

  component GFM_BRAM_double is
    port   (
        clk    : in  std_logic;
        a1     : in  word;
        logb1  : in  word;
        prod1  : out word;
        a2     : in  word;
        logb2  : in  word;
        prod2  : out word
    );
  end component;

  component GFD_BRAM_double is
    port   (
        clk    : in  std_logic;
        a1     : in  word;
        logb1  : in  word;
        quot1  : out word;
        a2     : in  word;
        logb2  : in  word;
        quot2  : out word
    );
  end component;

  component GFD_BRAM is
    port   (
        clk   : in  std_logic;
        a     : in  word;
        b     : in  word;
        quot  : out word
    );
  end component;

  signal  good                  : stripe;
  signal  dividend,divider      : word;
  signal  Pt,Qt,Pd,Qd           : word;
  signal  mult1,mult2           : word;
  signal  diff                  : integer range 0 to N;
  signal  dl,dk,kword,diffword  : word;
  signal  temp                  : word;
begin
  process(k,l,a)
  begin
    for i in a'range loop
      if i = l or i = k then
        good(i) <= (others => '0');
      else
        good(i) <= a(i);
      end if;
    end loop;
  end process;

  diff <= l-k;
  process(diff)
    variable t  : word;
  begin
    for i in t'range loop
      if i = diff then
        t(i) := '1';
      else
        t(i) := '0';
      end if;
    end loop;
    t(0) := '1';
    divider <= t;
  end process;

  testEncode: RAID6en_bram
    port map  (
        clk => clk,
        a   => good,
        P   => Pt,
        Q   => Qt
    );
  Pd <= Pt xor P;
  Qd <= Qt xor Q;

  kword <= conv_std_logic_vector(k,W);
  diffword <= conv_std_logic_vector(diff,W);

  dividemult: GFD_BRAM_double
    port map(
        clk    => clk,
        a1     => Qd,
        logb1  => kword,
        quot1  => mult2,
        a2     => (others => ’0’),
        logb2  => (others => ’0’),
        quot2  => open
    );


  multiply: GFM_BRAM_double
    port map(
        clk   => clk,
        a1    => Pd,
        logb1 => diffword,
        prod1 => mult1,
        a2    => (others => ’0’),
        logb2 => (others => ’0’),
        prod2 => open
    );
  dividend <= mult1 xor mult2;

  divide: GFD_BRAM
    port map  (
        clk  => clk,
        a    => dividend,
        b    => divider,
        quot => temp
    );

  dk <= temp xor one;
  dl <= dk xor Pd;

  process(dk,dl,a)
  begin
    for i in a’range loop
      if i = l then
        fixed(i) <= dl;
      elsif i = k then
        fixed(i) <= dk;
      else
        fixed(i) <= a(i);
      end if;
    end loop;
  end process;

end architecture;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.all;
use pkg_RAID6.all;
------------------------------------------------------------------------
-- Entity: RAIDf1_BRAM - finds 1 error and corrects it
------------------------------------------------------------------------
entity RAIDf1_BRAM is
  port   (
      clk    : in  std_logic;
      a      : in stripewp;
      fixed  : out stripewp
  );
end RAIDf1_BRAM;

architecture arch of RAIDf1_BRAM is

  component GFLog_BRAM is
  port   (
      clk : in  std_logic;
      I1  : in  word;
      O1  : out word;
      I2  : in  word;
      O2  : out word
  );
  end component;

  signal PQ,PQ2       : parity;
  signal pd,qd        : word;
  signal logpd,logqd  : word;
  signal l            : word;
  signal datain,data  : stripe;
  signal dl           : word;


begin
  copyin: for i in datain'range generate
    datain(i) <= a(i);
  end generate copyin;

  PQ <= RAIDenc(datain);
  pd <= PQ(0) xor a(N);
  qd <= PQ(1) xor a(N+1);

  u1: GFLog_BRAM port map(clk,pd,logpd,qd,logqd);

  l <= logqd - logpd;

  process(datain,l)
```

```
    variable s, t  : word;
  begin
    t := a(N);
    for i in datain'range loop
      if i = l then
        s := (others => '0');
      else
        s := datain(i);
      end if;
      t := t xor s;
    end loop;
    dl <= t;
  end process;

  process(a,dl,l)
  begin
    for i in fixed'range loop
      if i = l then
        fixed(i) <= dl;
      else
        fixed(i) <= a(i);
      end if;
    end loop;

  end process;
end architecture;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.all;
use pkg_RAID6.all;
----------------------------------------------------------------------
-- Entity: RAID6c2e_pipe - fixes 2 known data errors using pipelined
--              combinational circuit
-- fixed(0) <= correct value of a(x)
-- fixed(1) <= correct value of a(y)
----------------------------------------------------------------------
entity RAID6c2e_pipe is
  port   (
      clk    : in  std_logic;
      a      : in  stripewp;
      x,y    : in  integer range 0 to N+1;
      fixed  : out stripewp
  );
end RAID6c2e_pipe;

architecture arch of RAID6c2e_pipe is

  signal PQin,PQ          : parity := (others => (others => '0'));
  signal Px,Qx,Px1,Px2    : word := (others => '0');
  signal numerator        : word := (others => '0');
  signal fixedx, fixedy   : word := (others => '0');
  signal goodData        : stripe := (others => (others => '0'));
  signal diff, diff2, x1 : integer range 0 to N+1:= 1;


begin
  copyin: for i in goodData'range generate
    goodData(i) <= (others => '0') when (i=x or i=y) else a(i);
  end generate copyin;
  PQin(0) <= a(N);
  PQin(1) <= a(N+1);
  PQ <= RAIDenc(goodData);

  process(clk)
  begin
    if clk'event and clk = '1' then
      Px <= PQin(0) xor PQ(0);
      Qx <= PQin(1) xor PQ(1);
      diff <= y-x;
      x1 <= x;

      fixedx <= divby_2pxor1(GFD2p(Qx,x1) xor GFM2p(Px,diff),diff);
      Px2 <= Px;

      fixedy <= fixedx xor Px2;
    end if;
  end process;

  copyout: for i in a'range generate
    fixed(i) <= fixedx when i=x else
           fixedy when i=y
           else a(i);
  end generate copyout;

end architecture;
```