Error Correction and Detection in Digital Circuits

Tai Williams[1]

[1] *Spring 2025 COT3100H Final Project*

Digital circuits and hardware in general are vulnerable to errors from many different sources. To be specific, the incursion of flipped bits during data transmission or storage is the subject of focus. There have been many proposed solutions to help improve data integrity and resilience. Each of these solutions all involves the inclusion of extra bits outside of the intended data message to produce the ability to detect errors. In this work, the intention is to analyze some of these developed solutions and consider the usage of some of these approaches. Through this analysis, conclusions are drawn as to the efficiency of different encodings of redundant bits in terms of space complexity, error-correction capabilities, computational complexity, and more.

I. INTRODUCTION

Data integrity is an ever-important subject in any capacity involving data and electronics. Without the presence of countermeasures, a simple bit error, i.e., a 1 being flipped to a 0, can propagate through our systems and lead to critical errors. Such an error can come from a variety of sources, such as stray electromagnetic noise, physical defects, thermal fluctuations, and poor software. The sheer randomness and potential of these errors being incurred poses a genuine threat to the systems that we develop. Thus, this led to the development of countermeasures that improve data integrity. The countermeasures of focus in this work are:

- Brute Force Approach
- Hamming Codes
- Cyclic Redundancy Checks
- Bose-Chaudhuri-Hocquenghem Codes

II. APPROACH

A. Implementation

For the purposes of this work, the four methods above have all been implemented in Verilog Hardware Description Language and compiled using Icarus Verilog. Testbenches have been written that use the multiple modules which comprise each method discussed in Section I. The modules are all capable of taking in parameters but, for the sake of analysis and comparison, all data streams will be around 16 bits.

A linear feedback shift register is used to generate the data stream on a clocked signal with a period (T) of twenty nanoseconds. A second module known as **Flipper** uses multiple linear feedback shift registers which are comprised into a single signal using bitwise operations (Typically AND to reduce the number of ones). **Flipper** also outputs a *flips* vector which indicates which bits were flipped. The output signal of **Flipper** is XORed with the encoded data stream to randomly flip bits in the stream.

A final separate module **Checker** is used to compare the original data stream to the now decoded stream from a given method. **Checker** outputs two signals. The first is a one bit signal called *error* which returns true when there is a difference encountered between the streams and false otherwise. The second is a N-bit signal, N is the length of our data stream, called *BitError* that carries ones in places where there is a difference encountered in the data stream, and zeroes if the streams are the same at the given bit.

These modules form the basis for the testbench through which the methods were analyzed.

B. Comparison Analysis

To compare these methods against one another, the number of parity bits relative to the number of data bits will be the main consideration. Additionally, the probability of a given method to catch errors within a certain scale is of interest. Some systems require more precision and, by extension, are more vulnerable to errors than others so the determination of the best possible response to given conditions is important. Minimizing computations is also an essential feature to keep in mind.

III. BRUTE FORCE METHOD

A. Overview

The brute force approach is simple in both its implementation and computation. By placing duplicate bits surrounding each data bit, a layer of protection is created. The number of duplicate bits added (t) per data bit is typically even. The length of the codeword (N) is given by N = n(t+1) where n is the length of the message. If we take t to be 2, minimal size needed for the code word. We would have the following example:

$$11010 \longrightarrow 111111000111000$$

The method checks the value that makes up the majority of (t+1) blocks in the code word to decode back into the original data stream. I.E. 110 would treated as 1. An even number t is chosen to avoid the ambiguous case present in even sized data chunks.

B. Performance

The benefits of this approach is it's simplicity to implement and operate. For the case of random single bit errors inflicted on a data stream, this method can always correct them. To be more specific, this module can always correct a string of errors that are $t-1$ long given that the distance between these errors is $\geq t$. Depending on the placement of errors of length t, these can be corrected as well but not always. Errors of length $\geq t+1$ cannot be corrected by this module.

The main drawback is that error correcting capabilities are linearly correlated to the increasing size of the codeword. While effective, this method lacks the finesse of others especially when it comes to minimizing the size of codewords.

IV. HAMMING CODES

A. Overview

Hamming codes are linear error-correcting codes that are capable of correcting one-bit errors. This minimal error correction is made up for by the shorter codewords especially compared to the Brute force method above. A codeword made from a hamming code can have a maximum length of computed from the equation:

$$2^r \geq n + r + 1$$

Where r is the number of parity bits, and n is the number of data bits. The length of the codeword is given by $(n+r)$. By placing parity bits at bit positions that are powers of 2, a decoder module can check the validity of a codeword for errors by comparing expected parity values to that of what is present.

B. Math Background

The parity bits are placed in positions 2^i where $0 \leq i \leq r-1$. It is important to note that in this use case we count indices starting at 1. These parity bits are responsible for overseeing data bits whose index, in binary, has a one present in the same place. To be more specific, $2^i \ \& \ p \neq 0$ where p is the bit position. The following table displays some of the bit coverage provided by each parity bit:

Parity Bit	Bit Positions Covered
1	**1**, 3, 5, 7, 9, 11, 13, 15...
2	**2**, 3, 6, 7, 10, 11, 14, 15...
4	**4**, 5, 6, 7, 12, 13, 14, 15...
8	**8**, 9, 10, 11, 12, 13, 14, 15...

The value of a parity bit p_i is determined based on the bits which fall under its coverage. In this work, even parity is used to determine value of p_i. The value of p_i is such that the number of ones under its coverage is even. If the number of ones in the range is odd, p_i will take on the value of one to create an even number. Conversely if the number of ones is even, p_i will become zero to maintain the even parity.

Since any given data bit position is covered by 2 or more p_i, a change in one bit position will create a discrepancy in multiple p_i, which can be tracked and mapped. A syndrome vector is computed via XOR operation between the p_i in the codeword and p'_i calculated from the codeword. The numeric value of this syndrome vector gives the position of the error. If syndrome is 0, this means that no error has occurred. Consider the following example:

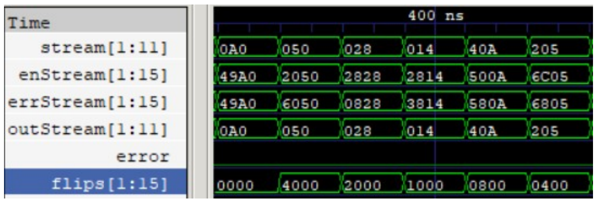
Consider the following codeword generated from **1101**:
1010101 with parity bits: $p_1 = 1,\ p_2 = 0,\ p_4 = 0$
During transmission, the bit in position 5 is flipped:
1010001 with parity bits: $p_1 = 1,\ p_2 = 0,\ p_4 = 0$
The receiver would recompute the parity bits to find the following: $p'_1 = 0,\ p'_2 = 0,\ p'_4 = 1$
A XOR operation between the p_i in the codeword and the recomputed p'_i yields: $100 \wedge 001 = 101 \rightarrow 5$. With this information, bit position 5 can be flipped restoring the message.

C. Implementation

Hamming Codes were implemented through two seperate modules: **HammingEncode** and **HammingDecode**. Each takes in parameters N and r, defining the codeword.

HammingEncode starts by assigning the data bits into a register with length $(N+r)$. In this assignment, bit positions that are a power of two in the codeword are skipped over and filled with zero. This results in a codeword filled with our N data bits, and zeroed out parity bits. Next, the module iterates through the codeword r times to compute each parity bit. By adding the bits under the given parity bit's coverage, the value necessary to maintain even parity is computed. To finish, the module outputs the value from the register.

HammingDecode takes in the encoded stream and iterates through it r times as well. This portion is the same as **HammingEncode** in the calculation of determining the necessary parity value. However at the end of each iteration, a bit of the syndrome vector is computed



```
Time                   400 ns
    stream[1:11]  0A0   050   028   014   40A   205
  enStream[1:15]  49A0  2050  2828  2814  500A  6C05
 errStream[1:15]  49A0  6050  0828  3814  580A  6805
 outStream[1:11]  0A0   050   028   014   40A   205
           error
      flips[1:15] 0000  4000  2000  1000  0800  0400
```

FIG. 1. Hamming Code Implementation

based on a XOR between the newly computed parity versus what is present in the stream. The syndrome's numeric value gives the position to be XORed. To finish its job, the module reconstructs the original datastream from the codeword, skipping indices that are powers of 2, only reading the data bits.

D. Performance

As mentioned above, Hamming Codes are particularly proficient at correcting only one error at a time. However, this is made up for in the reduced amount of space they take compared to the Brute Force Method. The computation is also not strenuous and can be implemented on hardware relatively simply.

V. CYCLIC REDUNDANCY CHECKS

A. Overview

Cyclic Redundancy Checks (CRC) are another form of error-management code used to protect data integrity. Compared to the previous methods, it cannot correct errors on its own. However, this does not devalue the use of this method given its strength in identifying any changes within a data stream.

The goal of CRCs is to use a given polynomial key to generate a remainder which can be used to validate data. This polynomial key is shared between both the transmitter and receiver. Both systems use the polynomial to calculate what the remainder of the data block should be and any difference indicates an error.

CRCs when used in this way are particularly useful in data transmission as if an error is detected, likely caused by noise, a signal can be sent to retransmit the data.

B. Math Background

The polynomial to be used in the CRC algorithm should be chosen with care as it dictates the error catching abilities of a system. The length and thereby degree of the polynomial has a direct influence on the length of the remainder. In this iteration, a polynomial $g(x)$ is defined as follows:

$$g(x) = p(x)(1 + x)$$

where p(x) is a primitive polynomial. A primitive polynomial is irreducible and in this case over a finite field $GF(2)$ thus only coefficients of 0 and 1 are allowed. $g(x)$ has a degree of r and thus $p(x)$ has a degree of r-1.

For this application I have chosen $p(x) = x^5 + x^3 + 1$ (Used in some RFID applications) and by extension $g(x) = x^6 + x^5 + x^4 + x^3 + x + 1$ represented in binary as 1111011. This polynomial will produce a remainder with a maximum degree of r-1. The code word generated by the CRC has a length of $C = N + r - 1$.

To compute the remainder, polynomial long division takes place in the form of binary operations. Rather than subtracting as division takes place on each level, a XOR operation takes place with the divisor polynomial $g(x)$.

Long Division Example:

$$\begin{array}{r l}
 & 1010111 \\
1001 & \overline{)\,10111011 \mid 000} \\
 & \underline{\wedge 1001} \\
 & 1010 \\
 & \underline{\wedge \ldots\ldots.} \\
 & \boxed{0110}
\end{array}$$

Since both the encoder and decoder modules compute the remainder, and the remainder is based on the entire data stream, any discrepancies indicate an error. By using the polynomials indicated, the code is capable of detecting single, double, triple, and any odd number of errors given the irreducibility of the polynomial.

C. Implementation

Like the Hamming Codes, CRC implementation has been split up into two modules: **crcEncode** and **crcDecode**.

crcEncode takes in three parameters: the size of the data stream (N), The degree of the polynomial + 1 (R), and an R-bit vector (DIV) that is g(x) in binary form. 2 registers are named to contain the codeword *mes* and the current remainder *crc*. The first step of logic is to load the original data stream into the first N bits of *mes* followed by appending R-1 zeroes. To establish the initial phase of long division, a slice of R bits is taken from the front of *mes* and assigned to *crc*.

In a loop that runs N times, if the most significant bit (MSB) of *crc* is 1, then *crc* is assigned to its XOR with DIV. If the MSB of *crc* is 0 then crc remains unchanged in this stage. At the end of each iteration crc is shifted left one bit and the next bit of *mes*, starting from the slice, is inserted as the least signficant bit (LSB). The module finishes by assigning a concatenation of the original data stream, and crc (excluding the MSB).

crcDecode carries out the same procedure as **crcEncode** but has different inputs and outputs. This

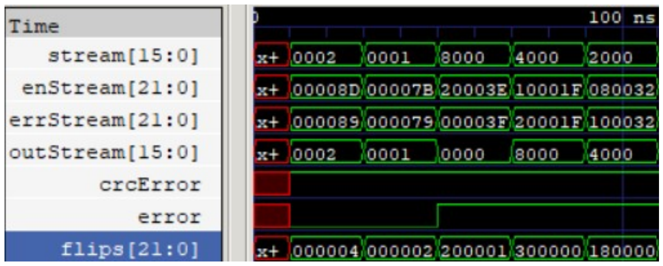

Time
100 ns
stream[15:0]
x+ 0002 0001 8000 4000 2000
enStream[21:0]
x+ 00008D 00007B 20003E 10001F 080032
errStream[21:0]
x+ 000089 000079 00003F 20001F 100032
outStream[15:0]
x+ 0002 0001 0000 8000 4000
crcError
error
flips[21:0]
x+ 000004 000002 200001 300000 180000

FIG. 2. CRC Implementation || note the false positive

module receives the N+R-1 long codeword *stream* and outputs the decoded data stream *outStream* and a one bit signal *crcError*. The same long division procedure is run on *stream* and a remainder *rem* is computed. Given no changes have occurred, the *rem* should hold the value 0. Otherwise, this indicates that an error has occurred since *stream* is no longer divisible by DIV.

D. Performance

This module performs exceptionally well in detection of errors. Under even severe cases of multiple errors within small ranges of each other, there were no false negatives when it came to catching errors. One element of note is the existence of inconsequential catches. For example, consider a single bit error in the LSB of the codeword encoded by the **crcEncode** module. When processed by **crcDecode** module, the *crcError* signal would be triggered however, the data bits of the codeword have not been altered and still hold value. This means that the *outStream* of **crcDecode** is actually correct creating a psuedo false positive in some cases.

VI. BOSE-CHAUDHURI-HOCQUENGHEM CODES

A. Overview

Bose-Chaudhuri-Hocquenghem Codes (BCH codes) are a special class of CRCs. BCH codes like the cyclic redundancy check use generator polynomials $g(x)$ defined over a finite field to find errors. Compared to the previous methods discussed, BCH codes are proficient in correcting multiple random errors while also attempting to minimize codeword length.

The usage of BCH codes takes the form of multiple steps. To generate the codeword, the original data $\mu(x)$ is shifted left to make space for an additional remainder $r(x)$ computed using a generator polynomial. This portion is very similar to CRCs.

Once the codeword has been computed, syndrome vectors are calculated that are used to detect and eventually correct errors. These syndrome vectors are then processed through the Berlekamp-Massey algorithm forming the error-locator polynomial $\Gamma(x)$. The roots of $\Gamma(x)$ at a given value indicates the position of an error that can then be flipped. Once these flips have been made, the original message is then reconstructed by reversing the encoding process.

B. Math Background

All calculations are done over $GF(2^m)$ created by a primitive polynomial of degree m. Roots of this polynomial are given as α^i. A CRC is defined by its length n, the number of data bits k, and the number of bits to correct T.

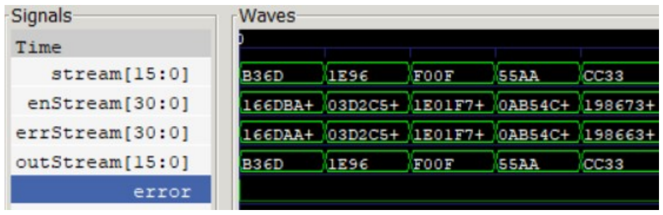
Encoding Process:
The original message $\mu(x)$ is shifted as follows $\mu'(x) = x^{n-k}\mu(x)$. Then a remainder is computed through long division like that of the CRCs. $r(x) = \mu'(x) \bmod g(x)$. The final code is computed by appending the remainder. $c(x) = \mu'(x) + r(x)$

Let $c'(x)$ represent the codeword after accumulating some errors $e(x)$. That is $c'(x) = c(x) + e(x)$.

Error Detection through Syndromes:
Each syndrome vector S_j is computed through the following $\sum_{i=0}^{N} S_j \wedge \alpha^{i*j \bmod (2^m-1)}$. To clarify, a given syndrome S_j is initialized to 0 and then continuously XORed with $\alpha^{i*j \bmod (2^m-1)}$ where i is incremented from 0 to N-1. The result is a set of $2T$ syndrome vectors. If each of these vectors is zero, this indicates that no error has occurred.

Error Locator Polynomial:
The Berlekamp-Massey algorithm is used to form the polynomial $\Gamma(x) = 1 + \Gamma_1 x + \Gamma_2 x^2 \ldots \Gamma_T x^T$ The goal of the algorithm is to determine these coefficients. All values are initialized to one prior to their introduction, excluding S_j. To compute the value of each coefficient, each syndrome value is used. On a given iteration with S_j, d is computed as follows: $d = S_j \wedge \sum_{i=1}^{L} \Gamma_i * S_{j-i}$. In the next step, for each Γ_i the following occurs: $\Gamma_i = \Gamma_i \wedge d \times B_{i-m}$. $B(x)$ serves as a helper polynomial throughout this process. Following this, $B(x)$ is set to be $\frac{\Gamma_{old}(x)}{d}$ where Γ_{old} is the value of Γ before the XOR operations. The following process repeats for each S_j until $\Gamma(x)$ has been computed.

Identifying Errors:
The roots of $\Gamma(x)$ hold information on $e(x)$. A root at α^{-i} indicates an error at the i-th postion of $c'(x)$. In hardware, these roots are found via Chien Search instead of being computed by hand. Once the locations of errors has been identified they can be flipped via a XOR operation yielding $c''(x)$.



```
stream[15:0]       B36D     1E96     F00F     55AA     CC33
enStream[30:0]     166DBA+  03D2C5+  1E01F7+  0AB54C+  198673+
errStream[30:0]    166DAA+  03D2C5+  1E01F7+  0AB54C+  198663+
outStream[15:0]    B36D     1E96     F00F     55AA     CC33
error
```

FIG. 3. BCH Implementation: Testing 1 bit Errors

Reconstructing Original Message:
After the appropriate XOR operations have occurred, $\mu(x)$ can be obtained by the following operation: $\mu(x) = \frac{c''(x)}{x^{n-k}} \bmod x^k$.

C. Implementation

To implement this method in Verilog, a modular approach is taken by separating each step into its own module. Inside of a top level wrapper module, an **encoder** and **decoder** module are instantiated.

The **encoder** module has three sub-modules: **shifter**, **poly_divider**, and **remainder_adder**. **shifter** is responsible for taking in a k-bit message and left shift it by n-k positions and output a n-bit message. **poly_divider** is responsible for dividing the message from **shifter** by the generator polynomial $g(x)$ and outputs the (n-k)-bit remainder. Finally **remainder_adder** concatenates the remainder and message from the previous two sub-modules to produce the final codeword.

The **decoder** module has five sub-modules: **syndrome_calc**, **berlMass**, **chien_search**, **error_corrector**, **message_extractor**. **syndrome_calc** takes in the n-bit codeword, and outputs $2T$ syndrome vectors following the process in section B. **berlMass** takes in the syndrome vectors and outputs the coefficients of the $\Gamma(x)$ error locator polynomial. **chien_search** receives the coefficients of $\Gamma(x)$ and produces an n-bit error vector E by testing $i \in [0, n-1]$ in the expression $\Gamma(a^{-i}) = 0$ to find roots. **error_corrector** takes in the codeword and the E vector and outputs the XOR result between the two vectors. **message_extractor** takes in the corrected codeword produced by **error_corrector** and outputs only the top k-bits in a *msg_out* vector.

To perform operations over the given field $GF(2^m)$ a Look-Up Table is used to store computations ahead of time.

D. Performance

The BCH codes are an effective approach to error correction that excel at both correction capabilities while also minimizing the space used in storing parity bits. A set up with a given T value is capable of resolving T errors within a distance of $2T+1$. There is a large amount of flexibility to be had in choosing the parameters of a BCH approach as long as they fall in the restrictions established by BCH. Additionally, the cyclic nature of BCH codes helps simplify their implementation in hardware whether that be through Look-Up Tables or using feedback shift registers to compute values. Overall, BCH codes are an exceptional way of improving data integrity.

VII. ANALYSIS

Each of the methods discussed in this work have their own specific use cases in which they excel. The most important consideration in deciding which to use is the application. While the Brute Force Approach can very quickly take up a large amount of space with redundant bits, it has large error-correcting capabilities. The hamming codes excel in single-bit error correction with minimal parity bits. CRCs are specialized in detection of errors through relatively simple computations. BCH codes add the error correction that CRCs are missing at the expense of more parity bits being required.

Given 16 data bits within optimal positioning:

Method	Parity Bits	Average Error Correction
BF	32	9
HC	5	1
CRC	6	0
BCH	15	3

It is important to note the respective integrity provided by each approach. In this analysis of the correction algorithms, consider a transmission in a noisy environment. This noisy environment depending on conditions may impart an error on any given bit position with probabilities of 5%, 15%, and 30%. The following table details the success rates of correction algorithms from this work:

Error Success Rates

Method	5%	15%	30%
BF	0.999	0.826	0.057
HC	0.717	0.155	0.006
BCH	0.933	0.296	0.007

Computed by the following equation:

$$\sum_{k=0}^{n} \binom{l}{k} p^k (1-p)^{l-k}$$

Where n is the number of errors correctable, l is the length of the code word, and p is the probability of an error occurring.

Consider the following table as well that computes a score that considers the required number of parity bits as well. Scores are computed via the following: $\frac{c'}{p} \times 100$ where c' is success rate shown in the previous table and p is the number of parity bits.

Error Correction Scores			
Method	5%	15%	30%
BF	3.12	2.58	0.178
HC	14.34	3.10	0.120
BCH	6.22	1.97	0.046

Interesting trends can be seen when considering the success rates against the number of parity bits used. In this instance with only 16 data bits, the hamming codes despite only being able to correct one error per data block has the highest score or close to it under each percentage. This is a clear indication of the importance of also considering the needed number of parity bits compared to the success rates. If the number of computations required were quantified and included more variation would occur, especially in the case of BCH codes that when compared to the other methods, require significantly more operations.

Data integrity is an essential field and by extension performing these probability tests especially based on the expected environment of deployment is essential. If the chances of error are minimal then a simple hamming code setup may be most effective. As the chances of error increase then a brute force or BCH approach may be needed depending on the size of codeword allowable. Additionally, if data can be resent, using a CRC may be the best approach rather than attempting to correct a corrupted data block.

VIII. Conclusion

This work served as an exploration into various error detection and correction methodologies used in devices throughout the world. Through the analysis of those discussed, it is clear that there is no one favorite that is supremely above any other. In reality many of technologies use multiple forms of these approaches in combination to further ensure data integrity. In the future, research into the further integration of these tactics would prove beneficial. The implementation of these systems into digital circuits simplifies some aspects given the limited field but also complicate some computations as vectors are unrolled for implementation. Overall, digital systems serve as a great platform for these methods in protecting data integrity. Research into this area is important for the functionality and safety of technology used everyday.

[1] Low-Complexity error Correction Algorithm for cyclic redundancy check codes. (2021, December 10). IEEE Conference Publication — IEEE Xplore. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp= &arnumber=9674684&tag=1

[2] Walters, M., Sinha Roy, S., & School of Computer Science, University of Birmingham, United Kingdom. (2021). Constant-Time BCH Error-Correcting code. In Constant-time BCH Error-Correcting Code [Journal-article]. https://eprint.iacr.org/2019/155.pdf

[3] Muneeb, M. A., S, N., Guru Nanak Dev Engineering College, Guru Nanak Dev Engineering College, & M A Muneeb. (2022). Verilog Implementation of hamming code for error control coding. In International Journal of Research in Engineering and Science (IJRES) (Vols. 10–10, Issue 1, pp. 69–73). https://www.ijres.org/papers/Volume-10/Issue-1/Ser-2/L10016973.pdf

[4] Koopman, P. & Carnegie Mellon University. (2004). Cyclic Redundancy Code (CRC) polynomial selection for embedded networks. In The International Conference on Dependable Systems and Networks & DSN-2004, Preprint. https://users.ece.cmu.edu/ ̃koopman/roses/dsn04/koopman04_crc_poly_embedded.pdf