Error Detection and Correction

Hui Chen [a]

[a]CUNY Brooklyn College, Brooklyn, NY, USA

September 11, 2023

Outline

Acknowledgement

The content of most slides come from the authors of the textbook:

Null, Linda, & Lobur, Julia (2018). The essentials of computer organization and architecture (5th ed.). Jones & Bartlett Learning.

Table of Contents

Lesson Objectives

Students are expected to be able to

1. describe the fundamentals of numerical data representation and manipulation in digital computers;
2. convert between various radix systems;
3. convert and perform arithmetic in signed integer representations;
4. explain how errors can occur in computations because of overflow and truncation;
5. express floating numbers in floating-point representation;
6. recognize the most popular character codes; and
7. *describe the concepts of error detecting and correcting codes*.

Table of Contents

Need for Error Detection and Correction

It is physically impossible for any data recording or transmission medium to be 100% perfect 100% of the time over its entire expected useful life.

As more bits are packed onto a square centimeter of disk storage, as communications transmission speeds increase, the likelihood of error increases – sometimes geometrically.

Thus, error detection and correction is critical to accurate data transmission, storage and retrieval.

Table of Contents

Detecting Errors

Bit error: $0 \rightarrow 1$ or $1 \leftarrow 0$

How to detect if there is an error?

Check digits, appended to the end of a long number, can provide some protection against data input errors, e.g.,

- The last characters of UPC barcodes and ISBNs are check digits (how are they computed?).
- Parity Bit. A bit computed and added to a sequence of bits
  - Even parity: even number of 1's including the parity bit
  - Odd Parity: odd number of 1's including the parity bit
  - What kind of bit errors can a single parity bit detect?

Longer data streams require more economical and sophisticated error detection mechanisms.

- Cyclic redundancy checking (CRC) codes provide error detection for large blocks of data.

Introduction to CRC

CRCs are widely adopted error detection code, in particular, in data communication networks, to detect transmission errors.

- transmit a 1 but receive a 0, or transmit a 0 but receive a 1

CRCs are polynomials over the modulo 2 arithmetic field.

The mathematical theory behind modulo 2 polynomials is beyond our scope. However, we can easily work with it without knowing its theoretical underpinnings.

CRC Algorithm: Setup

Represent n-bit data (message M) as n-1 degree polynomial, e.g., 5-bit data 11011 as

$$M(x) = 1x^4 + 1x^3 + 0x^2 + 1x^1 + 1x^0 = x^4 + x^3 + x + 1$$

Sender and receiver agrees on a divisor polynomial $C(x)$ of degree k, e.g., a degree 3 divisor polynomial,

$$C(x) = x3 + x2 + 1$$

which corresponds to 4-bit data: 1101

CRC is to generate an error detection code (denoted as E) consisting of $k - 1$ bits.

CRC Algorithm: Generating CRC Code

Algorithm generating M//E when given message M(x), divisor C(x)

1. Left shift M by k bits, i.e., $T(x) = M(x)x^k$
   - 11011 becomes 11011000
2. Compute remainder $T(x)/C(x)$
   - $(x4 + x3 + x + 1)x3/(x3 + x2 + 1)$, we get quotient $Q(x) = x^4 + 1$, remainder $R(x) = x^2 + 1$
3. Subtract $R(x)$ from $T(x)$, the result is M//E
   - $(x^4 + x^3 + x + 1)x^3 - (x^2 + 1) = x^7 + x^6 + x^4 + x^3 + x^2 + 1$

CRC: Error Detection

Algorithm verifying received message where received message represented as polynomial T(x)

1. Calculate remainder of $T(x)/C(x)$
2. If the remainder is not 0, an error
3. Otherwise, no errors detected (which does not mean there is no error)

Table of Contents

Need for Error Correction

Data transmission errors are easy to fix once an error is detected, i.e.,

- Just ask the sender to transmit the data again, in particular, when retransmission is cheap.

However these might be what we desire

- when retranmission is expensive (like in wireless networks), or
- in computer memory and data storage, this cannot be done because too often the only copy of something important is in memory or on disk.

Thus, to provide data integrity over the long term, we want to detect and correct errors, i.e., error correcting codes are required.

Error Correction Codes

Hamming codes and Reed-Solomon codes are two important error correcting codes.

Reed-Solomon codes are particularly useful in correcting burst errors that occur when a series of adjacent bits are damaged.

Because CD-ROMs are easily scratched, they employ a type of Reed-Solomon error correction.

Because the mathematics of Hamming codes is much simpler than Reed-Solomon, we discuss Hamming codes in detail.

Hamming Distance

Hamming codes are code words formed by adding redundant check bits, or parity bits, to a data word.

The Hamming distance between two code words is the number of bits in which two code words differ.

- The following two words have a Hamming distance of 3 because 3 bits are different 1000 1001
  1011 0001

Minimum Distance of a Code

The minimum Hamming distance for a code, D_{min} is the smallest Hamming distance between all pairs of words in the code. It determines its error detecting and error correcting capability.

- For any code word, X, to be interpreted as a different valid code word, Y, at least D_{min} single-bit errors must occur in X.
- Example. to detect k (or fewer) single-bit errors, a code must have a Hamming distance of $D_{min} = k + 1$.

Minimum Distance and Error Correction

Hamming code is designed with $D_{min} = 2k + 1$

- detect $D_{min} - 1$ errors, and
- correct $\lfloor \frac{D_{min}-1}{2} \rfloor$ errors

where k is the number of errors that the code can correct in any data word.

Hamming distance is provided by adding a suitable number of parity bits to a data word.

Design Consdieration

Suppose

- that we have a set of n-bit code words consisting of m data bits and r (redundant) parity bits, also
- that we wish to detect and correct one single bit error only.

An error could occur in any of the n bits, so each code word can be associated with n invalid code words at a Hamming distance of 1.

Therefore, we have n + 1 bit patterns for each code word: one valid code word, and n invalid code words.

Determining r

Using n bits, we have 2^n possible bit patterns. We have 2^m valid code words with r check bits (where $n = m + r$).

For each valid code word, we have $(n + 1)$ bit patterns (1 legal and n illegal).

This gives us the inequality:

$$(n + 1) \times 2^m \leq 2^n$$

Because $n = m + r$, we can rewrite the inequality as:

$$(m + r + 1) \times 2^m \leq 2^{m+r}$$

or

$$(m + r + 1) \leq 2^r$$

This inequality gives us a lower limit on the number of check bits that we need in our code words.

Determining r: Example 1

Suppose we have data words of length $m = 4$. Then:

$$(4 + r + 1) \leq 2^r$$

which implies that r must be greater than or equal to 3.

- We should always use the smallest value of r that makes the inequality true.

which means to build a code with 4-bit data words that will correct single-bit errors, we must add 3 check bits.

You have just done the hard part, i.e., to find the number of check bits. The rest is easy.

Determining r: Example 2

Suppose we have data words of length $m = 8$. Then:

$$(8 + r + 1) \leq 2^r$$

which implies that r must be greater than or equal to 4. Thus, to build a code with 8-bit data words that will correct single-bit errors, we must add 4 check bits, creating code words of length 12.

Again, you have just done the hard part, i.e., to find the number of check bits. The rest is easy.

Assigning Check Bits

So how do we assign values to these check bits? With code words of length 12, we observe that each of the bits, numbered 1 though 12, can be expressed in powers of 2. Thus:

$$1 = 2^0 \qquad 5 = 2^2 + 2^0 \qquad 9 = 2^3 + 2^0$$
$$2 = 2^1 \qquad 6 = 2^2 + 2^1 \qquad 10 = 2^3 + 2^1$$
$$3 = 2^1 + 2^0 \qquad 7 = 2^2 + 2^1 + 2^0 \qquad 11 = 2^3 + 2^1 + 2^0$$
$$4 = 2^2 \qquad 8 = 2^3 \qquad 12 = 2^3 + 2^2 \ldots$$

- $1 = 2^0$ contributes to all of the odd-numbered digits.
- $2 = 2^1$ contributes to the digits, 2, 3, 6, 7, 10, and 11.
- and so forth ...

We can use this idea in the creation of our check bits.

Assigning Check Bits: Approach

1. Using our code words of length 12, number each bit position starting with 1 in the low-order bit.
2. Each bit position corresponding to a power of 2 will be occupied by a check bit.
3. These check bits contain the parity of each bit position for which it participates in the sum.

$\overline{12}$	$\overline{11}$	$\overline{10}$	$\overline{9}$	$\overline{8}$ P	$\overline{7}$	$\overline{6}$	$\overline{5}$	$\overline{4}$ P	$\overline{3}$	$\overline{2}$ P	$\overline{1}$ P
				2^3				2^2		2^1	2^0

Assigning Check Bits: Example

Since $1 = 2^0$ contributes to the values 1, 3 , 5, 7, 9, and 11, bit 1 will check parity over bits in these positions.

Since $= 2^1$ contributes to the values 2, 3, 6, 7, 10, and 11, bit 2 will check parity over these bits.

...

For the word 11010110, assuming even parity, we have a value of 1 for check bit 1, and a value of 0 for check bit 2, which results in

1	1	0	1	P	0	1	1	P	0	P	P
12	11	10	9	8	7	6	5	4	3	2	1
				2^3				2^2		2^1	2^0

Assigning Check Bits: Example

The completed code word is shown above.

- Bit 1 checks the bits 1, 3, 5, 7, 9, and 11, so its value is 1 to ensure even parity within this group.
- Bit 2 checks the bits 2, 3, 6, 7, 10, and 11, so its value is 0.
- Bit 4 checks the bits 4, 5, 6, 7, and 12, so its value is 1.
- Bit 8 checks the bits 8, 9, 10, 11, and 12, so its value is also 1.

1	1	0	1	P=1	0	1	1	P=1	0	P=0	P=1
12	11	10	9	8	7	6	5	4	3	2	1
				2^3				2^2		2^1	2^0

Using the Hamming algorithm, we can not only detect single bit errors in this code word, but also correct them!

Error Detection: Example

Suppose an error occurs in bit 5, as shown below. Our parity bit values are:

- Bit 1 checks 1, 3, 5, 7, 9, and 11. This is incorrect as we have a total of 3 ones (which is not even parity).
- Bit 2 checks bits 2, 3, 6, 7, 10, and 11. The parity is correct.
- Bit 4 checks bits 4, 5, 6, 7, and 12. This parity is incorrect, as we have 3 ones.
- Bit 8 checks bit 8, 9, 10, 11, and 12. This parity is correct.

1	1	0	1	P=1	0	1	0	P=1	0	P=0	P=1
12	11	10	9	8	7	6	5	4	3	2	1
				2^3				2^2		2^1	2^0

Error Correction: Example

Suppose an error occurs in bit 5, as shown below.

- We have erroneous parity for check bits 1 and 4.
- Bit 1 checks 1, 3, 5, 7, 9, and 11. Bit 4 checks bits 4, 5, 6, 7, and 12. The intersection of checked bits are 5, 7. If we flip any bits other than 5 or 7, we make one parity correct, but not the other. We can not flip 7, because 7 is also checked by Bit 4. Correct bit 5. More generally, ...
- Which data bits are in error? We find out by adding the bit positions of the erroneous bits.
- Simply, $1 + 4 = 5$. This tells us that the error is in bit 5. If we change bit 5 to a 1, all parity bits check and our data is restored.

$\frac{1}{12}$	$\frac{1}{11}$	$\frac{0}{10}$	$\frac{1}{9}$	$\frac{\text{P=1}}{8}$	$\frac{0}{7}$	$\frac{1}{6}$	$\frac{0 \rightarrow 1}{5}$	$\frac{\text{P=1}}{4}$	$\frac{0}{3}$	$\frac{\text{P=0}}{2}$	$\frac{\text{P=1}}{1}$
				2^3				2^2		2^1	2^0

Table of Contents

Summary and Q&A

You are expected to be able to

1. *describe the concepts of error detecting and correcting codes*;

Any questions on:

- Data Corruption
- Error Detection
- Parity (even and odd parity)
- Cyclic redundancy checking (CRC)
- Error Correction and Hamming Code