



NCDC

NUST CHIP DESIGN CENTRE

Digital Logic Design

Error Detection and Correction

Error Detection and Correction



Part I: Simple Algorithms

- Parity: Single bit detection
- Hamming Codes: Simple error correction

Part II: Detecting Multiple Errors

- Theory: Modulo 2 arithmetic
- Practice Cyclic redundancy codes

The Big Idea: Redundant Encoding



- When storing or sending information, add extra bits (called redundancy)

Error Detection

- When accessing or receiving information, use the extra bits to deduce if corruption occurred.
- Trivial example: send each bit twice, if the values don't match, there was corruption.

Error Correction

- When accessing or receiving information, use the extra bits to fix errors in the data.
- Trivial example: send each bit three times, use a voting scheme to correct single bit errors.
- All error correction has its limits. Trivial example: what if noise corrupted two of the three copies?

Detection Versus Correction



- What are the tradeoffs between detection and correction?

Error Detection

- + Needs fewer redundancy bits.
- If an error is detected, receiver requests a resend of the data. This adds latency and complexity.
- If information was destroyed (like a DRAM flipping a bit) there is no data to resend.

Error Correction

- + Low Latency, correction happens as a part of the logic for receiving the data.
- Uses more redundancy bits.
- If too many errors happened, data cannot be corrected, and a resend will be needed anyways.



Simple Error Detection

Problem: Send an N-bit quantity $b_0 \dots b_{n-1}$.

Give the receiver a way to detect a single bit error.

Solution: Parity

- Send N+1 bits ($b_0 \dots b_{n-1}$).
- Bit b_n is one if there is an odd number of ones amongst $b_0 \dots b_{n-1}$.
- The extra bit is called **parity bit**.

Computing Parity Bit

- $b_n = b_0 \oplus b_1 \oplus \dots \oplus b_{n-1}$
- Where \oplus is the XOR function.
- Parallel data: combinatorial XOR tree.
- Serial data: XOR gate and register bit.



Simple Error Correction

Problem: Send an N -bit quantity $b_0 \dots b_{n-1}$. Give the receiver a way to detect and correct a single-bit error.

Hamming Codes

- Take overlapping subsets of $b_0 \dots b_{n-1}$.
- Compute one parity bit for each subset.
- Send M bits: N data bits and K parity bits.
- With enough parity bits, you can ID a flipped data bit.
- Receiver computes a checkword that is 0 (no errors) or whose value is the bit position of the error.
- Complement the bit position of the error to fix.
- R. Hamming, 1950.

Hamming Codes in Detail



Definition

- N data bits $b_0 \dots b_{n-1}$.
- K parity bits $p_0 \dots p_{k-1}$.
- M total bits $c_1 \dots c_M$.
- The c 's are numbered starting from **one**.

Step 1: Assign b_i 's and p_i 's to c_i

- Assign each b_i and p_i only **once**.
- First assign c_1 , then c_2 , etc.
- Assign c_i to a parity bit if i a power of two.
- Elsewise assign c_i to a data bit.
- Stop when no more data bits left to assign.



Assignment Example

Given $N = 4$: $b_0 \dots b_3$

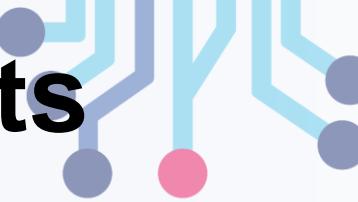
- $c_1 = p_0$. ($2^0 == 1$).
- $c_2 = p_1$. ($2^1 == 2$).
- $c_3 = b_0$.
- $c_4 = p_3$. ($2^2 == 4$).
- $c_5 = b_1$.
- $c_6 = b_2$.
- $c_7 = b_3$. (last *bi* used)

Bit Position	Binary Index	P1	P2	P4
1 (p_0)	0001	✓	✗	✗
2 (p_1)	0010	✗	✓	✗
3 (b_0)	0011	✓	✓	✗
4 (p_3)	0100	✗	✗	✓
5 (b_1)	0101	✓	✗	✓
6 (b_2)	0110	✗	✓	✓
7 (b_3)	0111	✓	✓	✓

Observations

- $N = 4$, $K = 3$, $M = 4 + 3 = 7$
- For large N , $K \approx \log_2(N)$
- This is a good thing.

Hamming Codes - Computing Parity Bits



Step 2: Compute Parity bits

- Recall from introduction ...
- Take overlapping subsets of the bits.
- Compute one parity bit for each subset.

The Question:

- How to take the subsets from $c_1 \dots c_M$?

Algorithm:

- Write c subscripts in binary.
- Subsets have a 1 in a particular bit position.
- Each subset has only one p_i .
- Set p_i to XOR of subset data bits.



Parity Computation Example

Recall $M = 7$

- $c_1 = c_{001} = p_0$
- $c_2 = c_{010} = p_1$
- $c_3 = c_{011} = b_0$
- $c_4 = c_{100} = p_3$
- $c_5 = c_{101} = b_1$
- $c_6 = c_{110} = b_2$
- $c_7 = c_{111} = b_3$

Subsets

- $(c_4, c_5, c_6, c_7) == (p_3, b_1, b_2, b_3)$
- $(c_2, c_3, c_6, c_7) == (p_1, b_0, b_2, b_3)$
- $(c_1, c_3, c_5, c_7) == (p_0, b_0, b_1, b_3)$

To Compute Parity

- XOR data bits of the subset.
- Assign to parity bit of the subset.
- Each subset has one parity bit by design.

Bit Position	Binary Index	P1	P2	P4
1 (p_0)	0001	✓	✗	✗
2 (p_1)	0010	✗	✓	✗
3 (b_0)	0011	✓	✓	✗
4 (p_3)	0100	✗	✗	✓
5 (b_1)	0101	✓	✗	✗
6 (b_2)	0110	✗	✓	✓
7 (b_3)	0111	✓	✓	✓

Hamming Codes – Correcting Error



Step 3: Correct Errors on Reception

- With enough parity bits, you can ID a flipped data bit.
- Receiver computes a checkword that is 0 (no errors) or whose value is bit position of the error.

The Question:

- How to compute checkword?

Algorithm:

- Break data into subsets.
- If subset contains p_i , set w_i to XOR of all subset members.
- Checkword = $w_0 + 2 * w_1 + 4 * w_2 + \dots$
- If checkword is zero, no errors.
- If checkword is non-zero, flip bit $c_{checkword}$.

Checkword Computation Example



$$c_1 = c_{001} = p_0$$

$$c_2 = c_{010} = p_1$$

$$c_3 = c_{011} = b_0$$

$$c_4 = c_{100} = p_3$$

$$c_5 = c_{101} = b_1$$

$$c_6 = c_{110} = b_2$$

$$c_7 = c_{111} = b_3$$

Subsets

$$(c_4, c_5, c_6, c_7) == (p_3, b_1, b_2, b_3)$$

$$(c_2, c_3, c_6, c_7) == (p_1, b_0, b_2, b_3)$$

$$(c_1, c_3, c_5, c_7) == (p_0, b_0, b_1, b_3)$$

To Compute Checkword

$$w_2 = c_4 \oplus c_5 \oplus c_6 \oplus c_7 = p_3 \oplus b_1 \oplus b_2 \oplus b_3$$

$$w_1 = c_2 \oplus c_3 \oplus c_6 \oplus c_7 = p_1 \oplus b_0 \oplus b_2 \oplus b_3$$

$$w_0 = c_1 \oplus c_3 \oplus c_5 \oplus c_7 = p_0 \oplus b_0 \oplus b_1 \oplus b_3$$

$$\text{Checkword} = w_0 + 2 * w_1 + 4 * w_2$$

Bit Position	Binary Index	P1	P2	P4
1 (p_0)	0001	✓	✗	✗
2 (p_1)	0010	✗	✓	✗
3 (b_0)	0011	✓	✓	✗
4 (p_3)	0100	✗	✗	✓
5 (b_1)	0101	✓	✗	✗
6 (b_2)	0110	✗	✓	✓
7 (b_3)	0111	✓	✓	✓

Hamming Code Epilogue



Why Does it Work?

- Uses binary code to advantage.
- If checkword not zero, a bit must be wrong.
- Each $w_0 = 0$ removes some bits from suspicion.
- Whichever bit is left must be wrong.
- If a bit is wrong, only one way to make it right!

Help! I'm Confused!

- It's normal to be confused after seeing it just once.
- Print these slides, reread.

Activity



Identify the corrupted transmissions and rectify the error.

Bit position	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	1	0	0	1	0	1	0	0	0
1	0	1	1	1	0	0	1	0	1	0	0	0
0	0	1	1	1	0	0	0	1	0	1	0	0

Detecting Multiple Errors



For Use When:

- Sending packets of data over an unreliable medium.
- Most of the time, bits arrive perfectly.
- Sometimes, many bits are corrupted.
- Need to detect corruption, request a resend.
- Parity too weak: multi-bit corruption is common.

Example Applications:

- Ethernet Packets
- Internet Protocol Datagrams



Basic Idea: Checksums

- Given a data packet of N bits.
- Compute an M bit **checksum** word on data.
- Send M checksum bits and N data bits.
- Receiver also computes checksum on packet.
- Parity is a very weak 1-bit checksum.

Desirable Checksum Properties

- N may vary packet by packet, M is fixed.
- Algorithm should be easy to compute.
- Algorithm and M chosen to match application.

Example: Cyclic Redundancy Check (CRC)



- Used in Ethernet Packets ($M = 32$).
- Ethernet P (undetected error) = $1/2^{32}$
- Less than one in a billion.

The Basic Idea

- Treat data packet as an N bit number.
- Divide number by a constant.
- The “remainder” of the division is the checksum.

But Division is Slow!

- Solution: use a number system where division is fast.
- Modulo 2 Arithmetic.



Modulo 2 Arithmetic

- $+$ is bit-wise XOR (no carries).
- $-$ is bit wise XOR (no borrows).
- $*$ is bit wise AND (no carries).
- Division uses $-$ and $*$ above (no borrows)

Not an Ad-hoc Scheme!

- Operations define a finite (Galois) field.
- Associative and distributive properties.
- $+$ and $*$ identities and inverses.
- Ops map one N -bit word to another N -bit word.

Modulo 2 Division



Generator

Data

Reminder

A binary division diagram illustrating the division of the dividend **101010** by the divisor **1001**. The quotient is **101** and the remainder is **10**.

The diagram shows the following steps:

- The dividend is **101010**.
- The divisor is **1001**.
- The quotient is **101**.
- The remainder is **10**.

Annotations highlight the first step of the division:

- A blue bracket underlines the first four bits of the dividend: **1001**.
- A green bracket underlines the first four bits of the divisor: **1001**.
- A red arrow points from the remainder **10** towards the bottom right corner.



CRC Algorithm in Detail

- D is N -bit data word to be sent.
- We compute M -bit checksum word R .
- Send concatenation: $2^M * D + R$

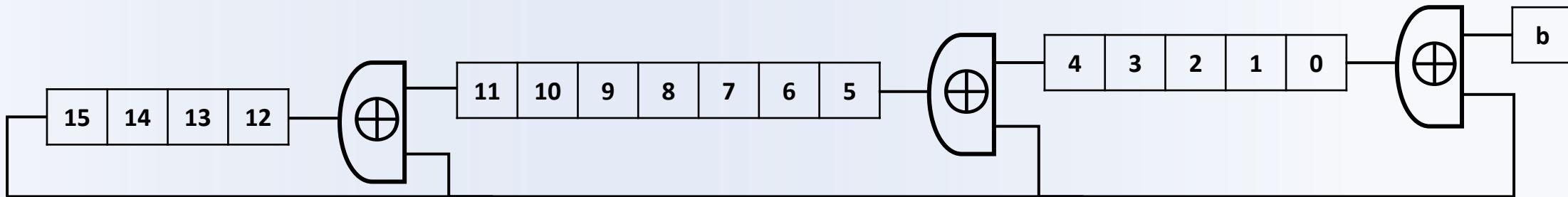
Computing Checksum

- Compute $(2^M * D)/G = Q + R/G$
- Ethernet $G = 100000100110000010001110110110111$
- Checksum is remainder R .

On Receipt

- Divide $(2^M * D + R)$ by G
- $(2^M * D) / G + (R/G)$
- From above: $Q + (R/G) + (R/G)$
- Rewrite: $Q + (R/G) - (R/G)$
- Remainder should be zero! If not, corrupt.

CRC Hardware



- Serial dividend flows from b box.
- Flip Flops hold 16-bit remainder R .
- 17-bit divisor $G = 10001000000100001$
- Input XOR is first and last 1.
- Two middle 1's map to middle XORs.

Preparing Checksum

- Clear flip-flops.
- Append 16 zeros to N -bit data D .
- $16 + N$ clocks to enter data.
- Remainder R sits in flip-flops.

Upon Receipt of $2^{16} * D + R$

- Clear flip-flops.
- $16 + N$ clocks to enter data.
- A non-zero flip-flop flags an error.

CRC – How does it Work?



- The Secret: it doesn't work for all divisors.
- Only for numbers with “cyclic” property.
- Theoretical background needed to go further.
- Linear Feedback Shift Registers
- Also used for random number generation.

In Conclusion

- Error coding is where match meets logic.
- Modulo 2 bit-serial circuits are powerful.
- Deep mathematics behind the gates.

Thank You

