2CS202CC23

Data Communication

B.Tech. Semester IV

Dhrumil Sheth 23BCE317

Parth Parmar 24BCE527

School of Engineering

Institute of Technology

Nirma University

Ahmedabad 382481

1. Introduction

The Hamming Code Visualizer provides an intuitive platform for understanding error-correction methodologies in digital communications. By implementing Hamming codes, the application demonstrates how parity bits strategically inserted within data sequences enable both the detection and correction of transmission errors. This visualization tool serves as a valuable resource for educational institutions, providing students and professionals with hands-on experience in error-correction techniques without requiring extensive theoretical background.

2. Project Overview

The Hamming Code Visualizer encompasses a comprehensive set of features that facilitate understanding of error-correction principles. Users can input binary data ranging from 4 to 16 bits, which the application then encodes using Hamming code algorithms. The encoded data can be manipulated to introduce errors at specific bit positions, allowing users to observe how the system identifies and corrects these errors.

The application's step-by-step visualization approach breaks down complex operations into digestible segments, enabling users to track how each parity bit contributes to error detection and correction. This methodical demonstration enhances comprehension of the mathematical relationships underlying Hamming codes and their practical application in ensuring data integrity during transmission.

3. Theoretical Background

Hamming codes represent a sophisticated approach to error management in digital communications. These codes systematically add redundancy through parity bits positioned at strategic locations within a data sequence. For a data sequence of length m, the number of required parity bits (r) is determined by the inequality $2^r \geq m + r + 1$, resulting in an encoded message of length m + r bits.

The structural organization of Hamming codes places parity bits at positions corresponding to powers of 2 (positions 1, 2, 4, 8, etc.), with data bits occupying all remaining positions. Each parity bit monitors a specific subset of bits within the encoded message, creating overlapping check patterns that enable precise error localization.

When transmission errors occur, the system rechecks each parity bit against its expected value. Parity bits that fail verification contribute to a binary number called the syndrome, which directly indicates the position of the erroneous bit. By flipping the bit at this position, the system automatically corrects the error, restoring data integrity with minimal computational overhead.

4. Algorithm Analysis

Hamming Encoding Algorithm

- Input: data - An array of binary digits (0 or 1)
- Output: encoded - An array representing the Hamming-encoded data

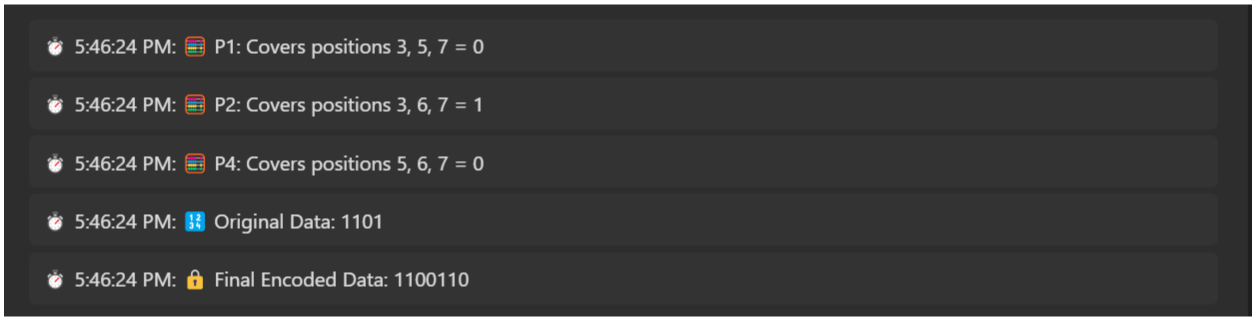
```
1. Calculate the number of parity bits needed (r):
  Set r = 1
  While 2^r < m + r + 1 do
     Increment r by 1
  End While


2. Initialize the encoded array of length m + r:
  For i = 1 to m + r do
     If i is a power of 2 then
        encoded[i-1] = 0 (reserve for parity bit)
     Else
        encoded[i-1] = next data bit
```

```
    End If
  End For

3. Calculate each parity bit:
  For i = 0 to r-1 do
    Set parityPos = 2^i
    Set parity = 0

    For each position j where bit parityPos in j's binary representation is 1:
      parity = parity XOR encoded[j-1]
    End For

    encoded[parityPos-1] = parity
  End For

4. Return encoded array
```

Original Data: 1101

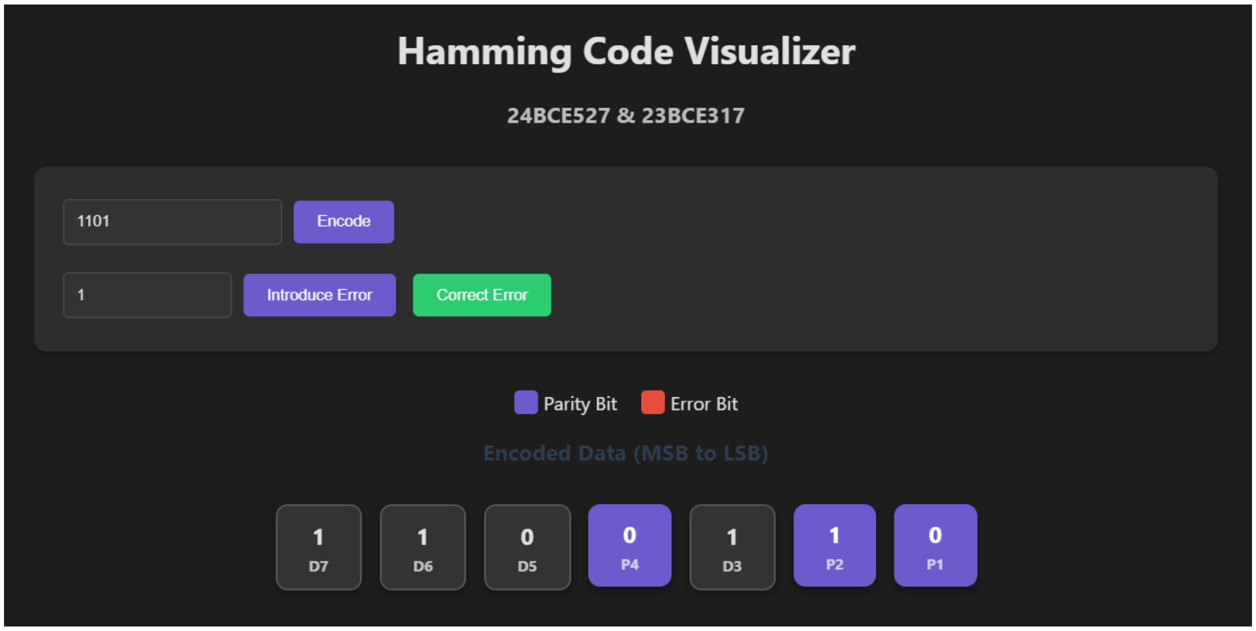
1. Data length m = 4, we need r = 3 parity bits (since 2^3 > 4+3+1)
2. Total encoded length = 7 bits
3. Calculate parity bits:
    - P1: Checks bits 1,3,5,7 → parity of (0,1,0,1) = 0
    - P2: Checks bits 2,3,6,7 → parity of (0,1,1,1) = 1
    - P4: Checks bits 4,5,6,7 → parity of (0,0,1,1) = 0

4. Final encoded message: 0110011



```
⏰ 5:46:24 PM: P1: Covers positions 3, 5, 7 = 0
⏰ 5:46:24 PM: P2: Covers positions 3, 6, 7 = 1
⏰ 5:46:24 PM: P4: Covers positions 5, 6, 7 = 0
⏰ 5:46:24 PM: Original Data: 1101
⏰ 5:46:24 PM: 🔒 Final Encoded Data: 1100110
```

Fig 1: Data Encoding



Hamming Code Visualizer

24BCE527 & 23BCE317

🟪 Parity Bit 🟥 Error Bit

Encoded Data (MSB to LSB)

1	1	0	0	1	1	0
D7	D6	D5	P4	D3	P2	P1

Fig 2: Data Visualization

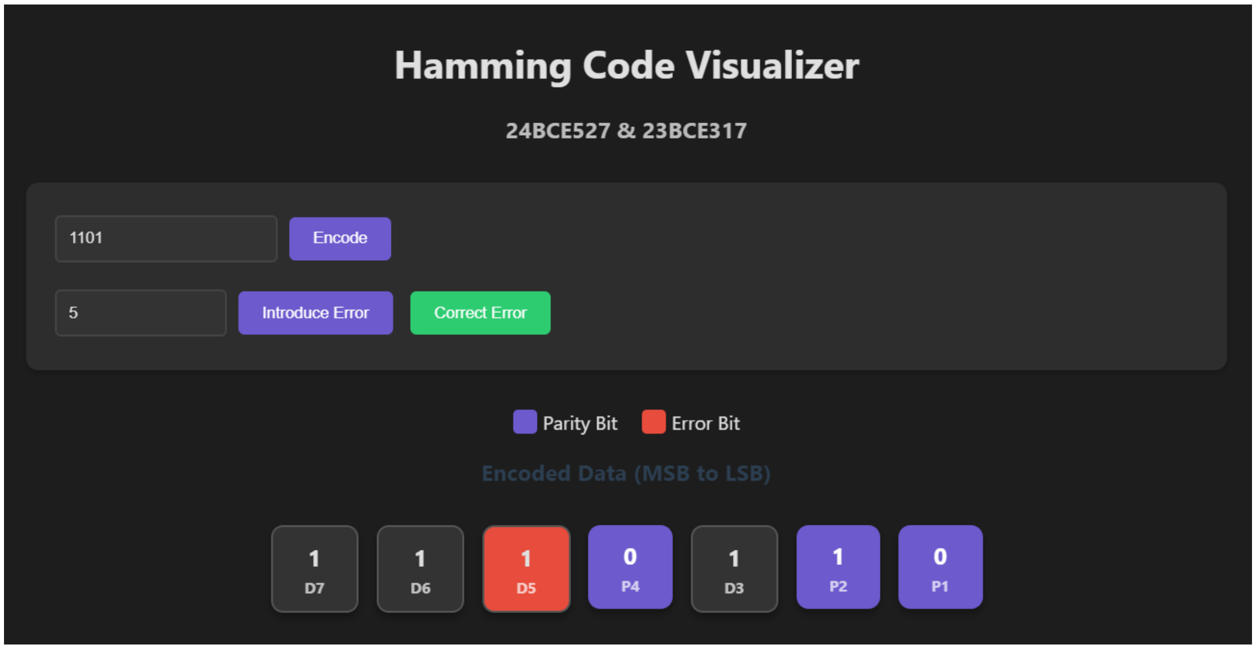
Error Detection Algorithm

- Input: encodedData - Hamming-encoded data that may contain an error
- Output: syndrome - Position of the error (0 if no error)

```
1. Initialize syndrome = 0


2. For each parity bit position i (where i is a power of 2):
 a. Calculate expected parity of all bits covered by this parity bit
 b. If calculated parity doesn't match the stored parity bit:
   syndrome = syndrome + i


3. Return syndrome
```

Example

- 7-bit Hamming code: 1100110 (right-to-left, 0-indexed)
- Error introduced at position 5: 1110110



Hamming Code Visualizer

24BCE527 & 23BCE317

1101

5

Fig 3: Error Introduction

Parity Check 1

- Checks bits 1,3,5,7 → values 0,1,1,1
- XOR = 1 (not 0) → Add 1 to syndrome

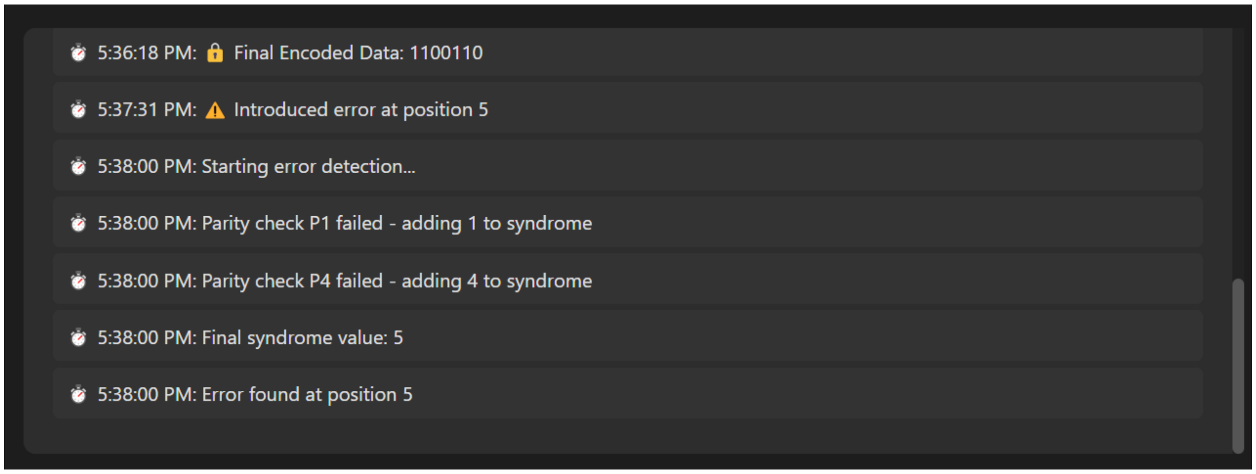
Parity Check 2 (position 1)

- Checks bits 2,3,6,7 → values 1,1,1,1
- XOR = 0 → No error detected

Parity Check 4 (position 3)

- Checks bits 4, 5, 6, 7 → values 0,1,1,1
- XOR = 1 (not 0) → Add 4 to syndrome

Final syndrome = Error position = 5



```
5:36:18 PM: 🔒 Final Encoded Data: 1100110
5:37:31 PM: ⚠ Introduced error at position 5
5:38:00 PM: Starting error detection...
5:38:00 PM: Parity check P1 failed - adding 1 to syndrome
5:38:00 PM: Parity check P4 failed - adding 4 to syndrome
5:38:00 PM: Final syndrome value: 5
5:38:00 PM: Error found at position 5
```

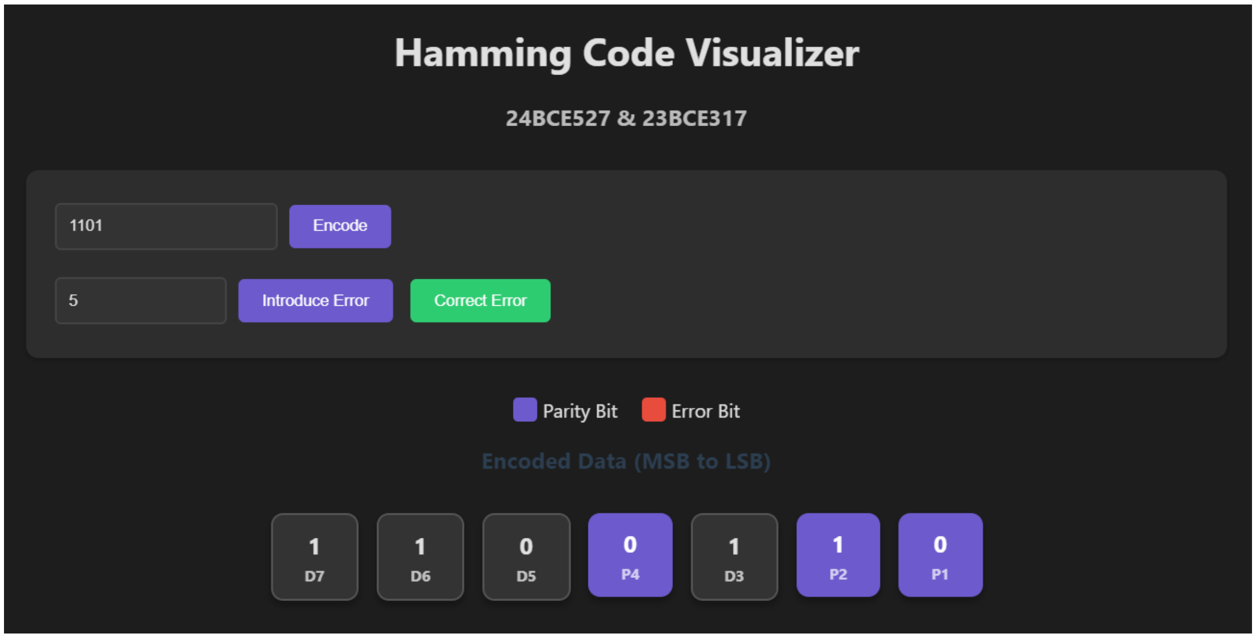
Fig 4: Error Correction Process

Error Correction Algorithm

- ➢ Input: encodedData - Hamming-encoded data with a possible error
- ➢ Output: correctedData - Error-corrected encoded data

```
1. Calculate syndrome using the calculateSyndrome algorithm


2. If syndrome is 0:
   a. No error detected, return encodedData unchanged


3. If syndrome is not 0:
  a. Flip the bit at position (syndrome-1)
  b. Return the modified encodedData
```

Final Corrected Data: 1100110


Hamming Code Visualizer

24BCE527 & 23BCE317

1101

5

1	1	0	0	1	1	0
D7	D6	D5	P4	D3	P2	P1

Fig 5: Error Corrected

5. Limitations and Alternatives

The current implementation of the Hamming Code Visualizer, while effective for educational purposes, has several inherent limitations. The fixed bit-length requirement restricts input to between 4 and 16 binary digits, limiting application to small data chunks. Additionally, the system only corrects single-bit errors; multiple bit flips may lead to incorrect corrections or undetected errors. The implementation also does not optimize redundancy beyond standard Hamming Code requirements, potentially increasing overhead for larger datasets.

Several alternative approaches could address these limitations. Enhanced Hamming Code (SEC-DED) extends standard Hamming Code with an additional parity bit, enabling both single error correction and double error detection. This approach provides improved reliability with minimal additional complexity, making it suitable for applications requiring higher data integrity.

Cyclic Redundancy Check (CRC) offers more robust error-detection through polynomial division to generate checksums. While CRC excels at detecting various error patterns, including burst errors, it typically lacks intrinsic correction capabilities. This approach is particularly valuable in network transmission scenarios where error detection takes precedence over automatic correction.

Reed-Solomon Codes represent a more sophisticated alternative, operating on blocks of data rather than individual bits. These codes can correct multiple errors within a block, making them ideal for storage media, telecommunications, and other applications susceptible to burst errors. While offering superior error management, Reed-Solomon Codes require more complex implementation and greater computational resources.

6. Implementation Considerations

The current implementation prioritizes educational clarity over performance optimization. In production environments, several enhancements could improve efficiency and scalability. Greater utilization of bitwise operations could accelerate parity calculations, while pre-computed syndrome tables would enable faster error correction. Parallel processing of multiple data blocks could significantly increase throughput in high-volume applications.

Scalability improvements could extend the implementation to support variable-length encoding beyond the current 16-bit limit. Block-based processing would enable handling of streaming data, while configurable redundancy levels could optimize performance based on channel reliability characteristics.

The user interface could be enhanced with advanced visualization modes illustrating bit relationships and dependencies. Real-time transmission simulation with configurable error rates would provide more realistic scenarios, while comparative analysis between different error correction schemes would offer valuable insights into their relative performance characteristics.

7. Conclusion

The Hamming Code Visualizer successfully demonstrates the principles of error detection and correction through interactive visualization and step-by-step explanations. Users gain practical understanding of parity bit calculation and positioning, syndrome-based error detection, and automatic single-bit error correction.

Through efficient algorithmic implementation and intuitive visualization, the application transforms abstract mathematical concepts into tangible processes that illustrate the mechanisms ensuring reliable data transmission in digital systems. The visualizer bridges theoretical knowledge and practical implementation, providing valuable educational resources for understanding error-correction methodologies.