|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Information Theory and Coding Techniques** | **Semester:** | **V** |
| **Date of Performance:** | **3/09/2024** | **Batch No:** | **B1** |
| **Faculty Name:** |  | **Roll No:** | **16014022050** |
| **Faculty Sign & Date:** |  | **Grade/Marks:** | **/ 25** |

**Experiment No: -5**

**Title:** Write C/ MATLAB Program for Hamming Codes

|  |
| --- |
| **Aim and Objective of the Experiment:** |
| 1. To generate 11 bit hamming code using 7 bit message bit. 2. To detect 1 bit error and correct it. |

|  |
| --- |
| **COs to be achieved:** |
| **CO1:** Use basic Concept of Probability Theory, Information Theory and Source Coding in communication.  . |

|  |  |
| --- | --- |
| **Theory:** | |
| In [telecommunication](http://en.wikipedia.org/wiki/Telecommunication), Hamming codes are a family of [linear error-correcting codes](http://en.wikipedia.org/wiki/Linear_code) that generalize the [Hamming (7, 4)-code](http://en.wikipedia.org/wiki/Hamming(7,4)) invented by [Richard Hamming](http://en.wikipedia.org/wiki/Richard_Hamming) in 1950. Hamming codes can detect up to two-bit errors or correct one-bit errors without detection of uncorrected errors. By contrast, the simple [parity cod](http://en.wikipedia.org/wiki/Parity_bit)e cannot correct errors, and can detect only an odd number of bits in error. Hamming codes are [perfect codes](http://en.wikipedia.org/wiki/Perfect_code), that is, they achieve the highest possible [rate](http://en.wikipedia.org/wiki/Block_code#The_rate_R) for codes with their [block length](http://en.wikipedia.org/wiki/Block_code#The_block_length_n) and [minimum distance](http://en.wikipedia.org/wiki/Block_code#The_distance_d) 3. Due to the limited redundancy that Hamming codes add to the data, they can only detect and correct errors when the error rate is low. | |
| **Stepwise-Procedure:** |
| **Steps of algorithm:**  1) Start  2) Take the 7 bit message bit (input)  3) Generate the 4 bit (parity) bits by using the message bit [Even Parity]  a] R0=D2+D4+D6+D8+D10  b] R1=D2+D5+D6+D9+D10  c] R2= D4+D5+D6  d] R3= D8+D9+D10  4) Place this 4 bit in the position of power of 2 i.e.:  R0 at 2 = 1st  R1 at 2 = 2nd  R2 at 2 = 4th  R3 at 2=8th  5) Hence the 11 bit Hamming Code is generated. i.e.:  D10 D9 D8 R3 D6 D5 D4 R2 D2 R1 R0  **Correction:**  6) Input the received 11 bit Hamming code  7) Check for the Parity bits. i.e.: R0, R1, R2, R3 [Even Parity] i.e.:  R0’=R0+D2+D4+D6+D8+D10  R1’= R1+D2+D5+D6+D9+D10  R2’= R2+D4+D5+D6  R3’= R3+D8+D9+D10  8) calculate the position of error bit by the formula;  h=8 x R3’ + 4 x R2’ + 2 x R1’ + 1 xR0’  9) Invert the bit at the hth position  10) Display the corrected 11 bit Hamming Code  11) Stop.  **Calculation**:  1) Enter the 7 bit message bits: 1 1 1 1 \_\_ 1 1 \_\_ 1 \_\_ \_\_  Even Parity D10 D9 D8 D7 R3 D5 D4 R2 D2 R1 R0  R0: D2+D4+D6+D8+D10: 1+1+1+1+1: R0 = 1  R1: D2+D5+D6+D9+D10: 1+1+1+1+1: R1 = 1  R2: D4+D5+D6: 1+1+1: R2 = 1  R3: D8+D9+D10: 1+1+1: R3 = 1  2) Generated 11 bits Hamming Code is: 1 1 1 1 1 1 1 1 1 1 1  3) Enter the received code: D10 D9 D8 D7 R3 D5 D4 R2 D2 R1 R0  Even Parity: 1 0 1 1 1 1 1 1 1 1 1  R0= R0+D2+D4+D6+D8+D10: 1+1+1+1+1+1: R0 = 0  R1= R1+D2+D5+D6+D9+D10: 1+1+1+1+0+1: R1 = 1  R2= R2+D4+D5+D6: 1+1+1+1: R2 = 0  R3= R3+D8+D9+D10: 1+1+0+1: R3 = 1  :. h = 8 x R3 + 4 x R2 + 2 x R1 + 1 x R0  = 8 x 1 + 4 x 0 + 2 x 1 + 1 x 0  = 10  :. Error is at position 10 from R.H.S  ie: D9 = 0 should be 1  :. Corrected Hamming Code: 1 1 1 1 1 1 1 1 1 1 1  **SIMULATION:** a. Draw the flowchart.  b. Attach the C/MATLAB simulation results for the same.  c. Prove the results with theoretical solutions. |

|  |
| --- |
| **Observations:** |
| # Function to calculate parity bits for Hamming Code generation  def calculate\_parity\_bits(data\_bits):      # Extract data bits (D2, D4, D5, D6, D8, D9, D10)      D2, D4, D5, D6, D8, D9, D10 = data\_bits      # Calculate parity bits using even parity      R0 = D2 ^ D4 ^ D6 ^ D8 ^ D10  # R0 = D2 + D4 + D6 + D8 + D10      R1 = D2 ^ D5 ^ D6 ^ D9 ^ D10  # R1 = D2 + D5 + D6 + D9 + D10      R2 = D4 ^ D5 ^ D6              # R2 = D4 + D5 + D6      R3 = D8 ^ D9 ^ D10             # R3 = D8 + D9 + D10      return [R0, R1, R2, R3]  # Function to generate 11-bit Hamming Code  def generate\_hamming\_code(data\_bits):      # Calculate parity bits      R0, R1, R2, R3 = calculate\_parity\_bits(data\_bits)      # Place the parity bits at positions 1, 2, 4, 8      hamming\_code = [R0, R1, data\_bits[0], R2, data\_bits[1], data\_bits[2], data\_bits[3], R3, data\_bits[4], data\_bits[5], data\_bits[6]]      return hamming\_code  # Function to check and correct errors in the received Hamming code  def correct\_hamming\_code(received\_code):      # Extract parity and data bits from the received code      R0, R1, D2, R2, D4, D5, D6, R3, D8, D9, D10 = received\_code      # Recalculate parity bits for error detection      R0\_check = R0 ^ D2 ^ D4 ^ D6 ^ D8 ^ D10  # R0'      R1\_check = R1 ^ D2 ^ D5 ^ D6 ^ D9 ^ D10  # R1'      R2\_check = R2 ^ D4 ^ D5 ^ D6             # R2'      R3\_check = R3 ^ D8 ^ D9 ^ D10            # R3'      # Calculate the error position      h = 8 \* R3\_check + 4 \* R2\_check + 2 \* R1\_check + 1 \* R0\_check      if h == 0:          print("No error in the received code.")      else:          print(f"Error detected at position: {h}")          # Correct the error by flipping the bit at the h-th position          received\_code[h - 1] = 1 - received\_code[h - 1]  # Flip the bit      return received\_code  # Example Usage  if \_\_name\_\_ == "\_\_main\_\_":      # Input 7-bit message: D2, D4, D5, D6, D8, D9, D10      data\_bits = [1, 1, 1, 1, 1, 1, 1]      # Generate Hamming code      hamming\_code = generate\_hamming\_code(data\_bits)      print("Generated 11-bit Hamming code:", hamming\_code)      # Simulate an error in the received code      received\_code = hamming\_code.copy()      received\_code[9] = 0  # Introduce an error at position 10 (D9)      print("Received code with error:", received\_code)      # Correct the error      corrected\_code = correct\_hamming\_code(received\_code)      print("Corrected Hamming code:", corrected\_code)  Output:    Theory: |

|  |
| --- |
| **Post Lab Subjective/Objective type Questions:** |
| 1. Classify the source coding techniques into character oriented and bit oriented protocol. Among this which one is effective technique for data compression and why?   Source coding techniques can be classified into two categories:   1. **Character-Oriented Protocols**:    * **Definition**: In character-oriented protocols, data is processed in terms of characters, each represented by a fixed number of bits. Examples include ASCII, where each character is represented by 7 or 8 bits, and UTF-8, which uses variable-length encoding but typically operates with 1 to 4 bytes per character.    * **Example**: ASCII encoding. 2. **Bit-Oriented Protocols**:    * **Definition**: In bit-oriented protocols, data is treated as a continuous stream of bits, not confined to character boundaries. These protocols can operate with variable-length encoding, which allows for better data compression by optimizing the number of bits used to represent different elements based on their frequency.    * **Example**: Huffman coding, Run-Length Encoding (RLE), Arithmetic coding.   **Effective Technique for Data Compression:**   * **Bit-oriented protocols** are generally more effective for data compression compared to character-oriented protocols. This is because:   1. **Variable-Length Encoding**: Bit-oriented techniques like Huffman coding assign shorter bit strings to frequently occurring symbols and longer bit strings to less frequent ones, optimizing the number of bits used and reducing redundancy.   2. **Flexibility**: They allow for a more fine-grained approach to encoding data, treating it at the bit level rather than being constrained by character boundaries.   3. **Higher Compression Ratio**: Bit-oriented methods are better at eliminating redundancies, resulting in a higher compression ratio, which is crucial for applications that need to reduce file size or transmission bandwidth.   Therefore, **bit-oriented protocols** are more efficient for data compression due to their adaptability in representing data using variable-length encoding schemes that minimize the number of bits required for frequently occurring data. |

|  |
| --- |
| **Conclusion:** |
| In this experiment, we successfully generated a 11-bit Hamming code from a 7-bit message and demonstrated error detection and correction for a single-bit error. Hamming codes prove to be an efficient technique for detecting and correcting single-bit errors in data transmission. |

|  |
| --- |
| **Signature of faculty in-charge with Date:** |