Ensuring Data Integrity in Digital Communications

An In-Depth Exploration of Hamming Code and Its Implementation in RISC-V Assembly

Made by **Youness Anouar**

Under the supervision of **Mr. ATIBI Mohamed**

Mohammed 6 Polytechnic university

College Of Computing

Computer Organization and Architecture Module

I. Introduction to Hamming Code:

In the realm of digital communications, data integrity is paramount. Hamming code, named after its inventor, Richard Hamming, is a sophisticated error-detecting and correcting code that plays a crucial role in ensuring that data transmitted across noisy channels remains accurate and reliable. Developed in the late 1940s, Hamming code was one of the first error-correcting codes and continues to be relevant today due to its efficiency and effectiveness.

II. Objective of Hamming Code

The primary objective of Hamming code is to detect and correct single-bit errors in data bytes transmitted over unreliable or noisy communication channels. This is achieved by adding extra "parity bits" to the data bits. The positions of these parity bits are cleverly calculated to enable the detection and correction of errors in the specific bits they monitor.

III. Applications of Hamming Code

Hamming code is widely used in various applications where data integrity is critical. Some of the key applications include:

1. **Wireless Communications:** In wireless communication systems, Hamming code helps in maintaining the accuracy of data transmitted over airwaves that are susceptible to interference and noise.
2. **Data Transmission Systems:** It is also employed in data transmission systems like modems, where the integrity of received data needs to be verified and maintained.
3. **Satellite Communication:** Due to the long distances involved and the presence of cosmic interference, satellite communications systems rely on error-correcting codes like Hamming to preserve data integrity.

IV. The core algorithm

To delve deeper into the algorithms and mathematics behind Hamming code, let's examine how it detects and corrects errors.

1. Message Preparation

Before transmission, the original data bits are augmented with redundant parity bits at specific positions. These positions are typically powers of two (i.e., 1st, 2nd, 4th, 8th positions, etc.).

2. Parity Bit Calculation

Parity bits are calculated to make the total count of 1's in certain groups of bits (including the parity bit itself) either even or odd. This parity is determined by the parity checking method (even or odd) used in the algorithm.

The general formula to calculate parity for a set of bits is :

$$pi = \bigoplus j \in Si \; bj$$

where:

- pi is the i-th parity bit,
- Si is the set of positions that include the i-th bit in their binary representation.
- bj the j-th bit of the message including data and parity bits.
- $\oplus$ denotes the XOR operation.

3. Code Formation

Once the parity bits are calculated, they are inserted into the frame at their

4. Error Detection and Correction at the Receiver

Upon reception, the receiver recalculates the parity bits using the same groups and parity checking method as the sender. If the calculated parity matches the received parity for all parity bits, it is assumed that there is no error. Otherwise,

the position of the error is determined by the parity bits that are incorrect. This position is given by:

$$C = \sum_{i=0}^{n-1} ci \cdot 2^i$$

where *ci* are the recalculated parity bits and *n* is the number of parity bits.

5. Error Correction

If an error is detected, it is corrected by flipping the bit at the error position identified by *C-1*.

- If all parities are correct (i.e., even), there is no error, or there is an even number of errors, which Hamming code cannot detect.
- If the parity check results in an odd value, the positions of the parity bits that detect an error can be summed (using XOR) to determine the exact location of the error.

Let us take an example :

- **Step 1: Message Preparation**

  Assume we have a 24-bit message that we want to transmit securely. For simplicity, let's denote this message as: M=1101 0110 1001 1010 1110 0011M=1101 0110 1001 1010 1110 0011

- **Step 2: Parity Bit Placement**

  For Hamming Code, parity bits are placed at positions that are powers of two (1, 2, 4, 8, 16,...). Since we are dealing with a 24-bit message, we will calculate the number of parity bits needed and their positions. The total number of bits (message + parity bits) should satisfy the condition $2r \geq m + r + 12r \geq m + r + 1$ where m is the original number of bits (24 in this case) and r is the number of parity bits.

  For m=24:

  r=5 (since $2^5 = 32$ is the smallest power of 2 greater than 24+5+1=30)

Thus, parity bits are placed at positions 1, 2, 4, 8, and 16.

- **Step 3: Message with Parity Bits :**

  Inserting parity bits into the original message: _ _ 1 _ 101 011 010 01 _ 101 011 100 111 _ 001 1_ _ 1 _ 101 011 010 01 _ 101 011 100 111 _ 001 1

  In this representation, underscores (_) represent positions of the parity bits (yet to be calculated).

- **Step 4: Parity Bit Calculation**

  Each parity bit covers specific bits in the message:

  + $p1$ covers all bits whose positions have the lowest bit in binary representation set (positions 1, 3, 5, 7, 9,...).
  + $p2$ covers bits whose second-lowest bit is set (positions 2, 3, 6, 7, 10, 11,...).
  + $p4$ covers bits in positions 4 through 7, 12 through 15, and so on.
  + $p8$ and $p16$ similarly cover increasingly larger blocks of bits.

  Using XOR operations, we calculate each parity bit to ensure even parity:

$$p1 = M1 \oplus M3 \oplus M5 \oplus \dots$$

$$p2 = M2 \oplus M3 \oplus M6 \oplus M7 \oplus \dots$$

$$p4 = M4 \oplus M5 \oplus M6 \oplus M7 \oplus M12 \oplus M13 \oplus M14 \oplus M15 \oplus \dots$$

$$p8 = M8 \oplus M9 \oplus M10 \oplus M11 \oplus M12 \oplus M13 \oplus M14 \oplus M15 \oplus M24 \dots$$

$$p16 = M16 \oplus M17 \oplus M18 \oplus M19 \oplus M20 \oplus M21 \oplus M22 \oplus M23 \oplus M24 \dots$$

- **Step 5: Insert Calculated Parity Bits**

  After calculating the parity bits, they are inserted at their respective positions in the message: M' = $p1$ $p2$ 1 $p4$ 101 011 010 01 $p8$ 101 011 100 111 $p16$ 001 1

- **Step 6: Transmission**

  The completed Hamming code, including the original data bits and the newly inserted parity bits, is transmitted to the receiver.

- **Step 7: Error Detection at the Receiver**

- Upon reception, the receiver recalculates the parity for the received message using the same groups and checks: $c1$, $c2$, $c4$, $c8$, $c16$...
  - If the recalculated parity bits (ci) match the received parity bits, the message is assumed to be correct. Otherwise, an error has been detected.

- **Step 8: Error Position Calculation**

  To locate the error, the receiver computes: $C = \sum^{i} ci \cdot 2^{i}$ where ci are the recalculated parity bits and the sum is taken over all recalculated parity bits.

- **Step 9: Error Correction**

  If an error is detected ($C \neq 0$), the bit at position $C-1$ is flipped (inverted) to correct the error.
  - Let's say upon reception, the recalculated parities are as follows: $c1 = 1$, $c2 = 0$, $c4 = 0$, $c8 = 1$, $c16 = 0$ This indicates an error at position:

$$C = 1.2^0 + 0.2^1 + 0.2^2 + 1.2^3 + 0.2^4 = 1 + 8 = 9$$

  - Therefore, the bit at position 8 is incorrect and must be flipped to correct the error.

V. Assembly Implementation in RISC-V :

The main function orchestrates the entire process of generating, encoding, transmitting, decoding, and correcting the message. It performs the following steps:

- Loads the matricule and the parameter r.
- Calls get_message_asm to generate the message to be sent.
- Calls hamming_map_asm to map the message into the Hamming code format.
- Calls hamming_encode_asm to encode the mapped message with parity bits.
- Simulates the reception of data (potentially with errors) and calls hamming_decode_asm to correct any errors.
- Finally, it calls hamming_unmap_asm to retrieve the original message from the corrected data.

1. `get_message_asm` Function :

Purpose: This function is designed to generate the initial message sent across the communication channel. It performs several operations on an input integer, representing a "matricule" or a unique identifier.

Working Steps:

- **Bit inversion**: Initially, all bits of the 'matricule' are inverted. This is done using the bitwise NOT operation.
- **rotation**: The inverted bits are then rotated left by a specified number of bits (r). This is achieved by shifting left (sll) and shifting right (srl) and then combining the two shifted values using or.

```
not t0, a0              # t0 = ~a0, bitwise NOT of a0
not t2, a0              # t2 = ~a0, duplicate for the second part of rot
sll t0, t0, t1          # t0 = t0 << t1, shift left logical by t1 places

neg t1, t1              # t1 = -t1, negate t1
addi t1, t1, 32         # t1 = t1 + 32, adjust t1 for right shift amount
srl t2, t2, t1          # t2 = t2 >> t1, shift right logical
or t0, t0, t2           # t0 = t0 | t2, combine the shifted bits
```

- **Masking**: Finally, only the least significant 24 bits are kept by applying a bitmask, effectively trimming the result to the required message size.

```
li t2, 0xFFFFFF         # t2 = 0xFFFFFF, load mask into t2
and t0, t0, t2          # t0 = t0 & t2, apply mask
mv a0, t0               # Move t0 into return register a0
```

Role: This function prepares the raw data (matricule) by manipulating it into a form that will be further processed for error encoding (mapping and parity bit addition). This is a preliminary step before applying Hamming code.

2. `hamming_map_asm` Function :

The purpose of hamming_map_asm is to map a given message using the Hamming code. This involves rearranging and adding parity bits to the message to facilitate error detection and correction.

- **Allocate Space:** The function begins by allocating space on the stack to store temporary registers.

```
addi sp,sp, -12: Allocates space on the stack for three temporary registers.
```

- **Save Registers:** Saves the values of registers t0, t1, and t2 onto the stack to preserve their original values during execution.

```
sw t0,8(sp), sw t1, 4(sp), sw t2, 0(sp): Stores the values of t0, t1, and t2 onto the stack.
```

- **Bit Manipulation Operations**: Extracting specific bits from the input message and rearranging them to form the mapped message:

```
andi t0,a0,1: Extracts the least significant bit of the input message (a0) using a bitwise AND operation and stores it in t0.

slli t0,t0,2: Shifts the extracted bit two positions to the left to position it correctly in the mapped message.

andi t1,a0,14: Extracts the three most significant bits of the input message using a bitwise AND operation and stores them in t1.

slli t1,t1,3: Shifts the extracted bits three positions to the left to position them correctly in the mapped message.
```

```
xor t0, t0, t1: XORs the bits extracted from steps 1
and 2 to combine them into the mapped message.

li t2,0x7F0: Loads a mask into t2 to isolate the bits
in the middle section of the input message.

and t1, a0, t2: Extracts the middle bits of the input
message using a bitwise AND operation and stores them
in t1.

slli t1,t1,4: Shifts the extracted bits four positions
to the left to position them correctly in the mapped
message.

xor t0,t0,t1: XORs the bits extracted from steps 4 and
5 to combine them into the mapped message.

li t2 , 0xFFF800: Loads a mask into t2 to isolate the
bits in the last section of the input message.

and t1,a0,t2: Extracts the last bits of the input
message using a bitwise AND operation and stores them
in t1.

slli t1,t1,5: Shifts the extracted bits five positions
to the left to position them correctly in the mapped
message.

xor t0,t0,t1: XORs the bits extracted from steps 7 and
8 to combine them into the final mapped message.
```

- **Return Result:** Moves the mapped message from t0 back to a0.

```
mv a0,t0: Moves the mapped message from t0 to a0.
```

- **Restore Registers and Stack**: Restores the original values of registers t0, t1, and t2 from the stack and deallocates the allocated space on the stack.

```
lw t0,8(sp), lw t1,4(sp), lw t2,0(sp): Loads the
original values of t0, t1, and t2 from the stack.

addi sp, sp , 12: Deallocates the space allocated on
the stack.
```

- **Return**: Returns from the function.

```
    jr ra: Jumps to the return address stored in ra to
    return from the function.
```

3. parity Function :

The purpose of the parity function is to calculate the parity of a given number (in this case, the bits extracted from the message).

- **Allocate Space**: Allocate space on the stack to store temporary registers.

```
addi sp,sp, -12: Allocates space for three
temporary registers on the stack.
```

- **Save Registers:** Save the values of registers t0, t1, and t2 onto the stack to preserve their original values during execution.

```
    sw t0,8(sp), sw t1, 4(sp), sw t2, 0(sp): Stores the
values of t0, t1, and t2 onto the stack.
```

- **Copy Argument:** Copy the argument (a1) to t0 for manipulation.

```
mv t0,a1: Copies the argument to t0.
```

- **Initialize Parity Count**: Initialize t1 to zero, which will be used to count the number of set bits.

```
    li t1,0: Loads zero into t1.
```

- **Calculate Parity:**

  Loop through each bit of the input number (t0) until all bits are processed:
  Check if the least significant bit of t0 is zero. If it is, jump to the exit.
  Extract the least significant bit of t0 (t2) using a bitwise AND operation.
  Add the extracted bit (t2) to the parity count (t1).
  Right shift t0 to process the next bit.
  Repeat the loop until all bits are processed.
  Exit the loop when all bits have been processed.

```
loop:
      beqz t0, exit      # If t0 is 0, exit the loop
      andi t2, t0, 1     # Isolate the least significant bit
      add t1, t2, t1     # Add it to t1, which accumulates the count of '1'bits
      srli t0, t0, 1     # Shift t0 right by one to process the next bit
      j loop             # Jump back to the start of the loop
exit:
      mv a1, t1          # Move the count of '1' bits into a1
```

- **Return Parity:** Move the calculated parity count (t1) to the argument register (a1) to return it to the caller.

```
    mv a1,t1: Copies the calculated parity count to the
argument register a1.
```

- **Restore Registers:** Restore the original values of registers t0, t1, and t2 from the stack.

```
    lw t0,8(sp), lw t1,4(sp), lw t2,0(sp): Loads the
original values of t0, t1, and t2 from the stack.
```

- **Deallocate Space:** Deallocate the space allocated on the stack.

```
    addi sp, sp , 12: Deallocates the space allocated
for temporary registers on the stack.
```

- **Return:** Return from the function.

```
    jr ra: Jumps to the return address stored in ra to
return from the function.
```

4. `hamming_encode_asm` Function:

The purpose of hamming_encode_asm is to encode the given message using the Hamming code by adding parity bits at specific positions.

- **Allocate Space**: Allocate space on the stack to store temporary registers.

```
addi sp,sp, -12: Allocates space for three
temporary registers on the stack.
```

- **Save Registers:** Save the values of registers t0, t1, and t2 onto the stack to preserve their original values during execution.

```
sw t1,8(sp), sw t0, 4(sp), sw t2, 0(sp): Stores the
values of t1, t0, and t2 onto the stack.
```

- **Save Return Address:** Save the return address to the caller (stored in ra) in register t1 to be restored later.

```
mv t1,ra: Copies the return address from ra to t1.
```

- **Encode Message**: For each parity bit position:

  Load the corresponding mask into t2.

  Extract bits from the message using a bitwise AND operation with the mask.

Call the parity function to calculate parity for the extracted bits.

Extract the calculated parity bit (least significant bit) and add it to the encoded message using a bitwise OR operation.

Shift the calculated parity bit to the correct position and add it to the encoded message.

Repeat this process for all parity bit positions.

- **Restore Registers:** Restore the original values of registers t0, t1, and t2 from the stack.

```
lw t1,8(sp), lw t0,4(sp), lw t2,0(sp): Loads the original values of t1, t0, and t2 from the stack.
```

- **Restore Return Address:** Restore the return address from t1 back to ra.

```
mv ra,t1: Copies the return address from t1 back to ra.
```

- **Deallocate Space:** Deallocate the space allocated on the stack.

```
addi sp, sp , 12: Deallocates the space allocated for temporary registers on the stack.
```

- **Return:** Return from the function.

```
    jr ra: Jumps to the return address stored in ra to return from the function.
```

```
    mv t1, ra           # Save return address
```

```
li t2, 0x55555554     # Apply mask for first parity calculation
and a1, t2, a0        # Apply bitwise AND to isolate bits for parity calcls
jal parity            # Call parity function to calculate parity bit
andi a1, a1, 1        # Isolate the lowest bit
or t0, t0, a1         # Combine the parity bit into the encoded output

# Repeats similar steps for other parity bits with different masks
```

- **Restore Registers:** Restore the original values of registers t0, t1, and t2 from the stack.

  `lw t1,8(sp), lw t0,4(sp), lw t2,0(sp): Loads the original values of t1, t0, and t2 from the stack.`

- **Restore Return Address:** Restore the return address from t1 back to ra.

  `mv ra,t1: Copies the return address from t1 back to ra.`

- **Deallocate Space:** Deallocate the space allocated on the stack.

  `addi sp, sp , 12: Deallocates the space allocated for temporary registers on the stack.`

- **Return:** Return from the function.

  `jr ra: Jumps to the return address stored in ra to return from the function.`

5. `hamming_decode_asm` Function

Error Detection and Correction: This function detects and corrects errors in a received Hamming code by recalculating parity bits and comparing them with received parities to identify errors.

- **Allocate Space**: Allocate space on the stack to store temporary registers.

```
addi sp,sp, -12: Allocates space for three
temporary registers on the stack.
```

- **Save Registers:** Save the values of registers t0, t1, and t2 onto the stack to preserve their original values during execution.

```
sw t1,8(sp), sw t0, 4(sp), sw t2, 0(sp): Stores the
values of t1, t0, and t2 onto the stack.
```

- **Save Return Address:** Save the return address to the caller (stored in ra) in register t1 to be restored later.

```
mv t1,ra: Copies the return address from ra to t1.
```

- **Decode Message:**

  For each parity bit position:

  1. Load the corresponding mask into t2.
  2. Extract bits from the received data using a bitwise AND operation with the mask.
  3. Call the parity function to calculate parity for the extracted bits.
  4. Extract the calculated parity bit (least significant bit) and add it to t0 using a bitwise OR operation. This step identifies the position of the error bit, if any.

5. Left shift the calculated parity bit to position it correctly.
6. Add the shifted parity bit to t0.
7. Repeat this process for all parity bit positions.

- **Correct Error (if any):**
  1. If t0 is non-zero, indicating an error was detected:
  2. Calculate the bit position of the error by converting t0 to a mask.
  3. XOR the received data with the error mask to correct the erroneous bit.

```
mv t1, ra             # Save return address

li t2, 0x55555555     # Mask for parity bits
and a1, t2, a0        # Isolate relevant bits for parity calculation
jal parity            # Compute parity
andi a1, a1, 1        # Isolate lowest bit
or t0, t0, a1         # Store computed parity bit

# Additional parity calculations follow similar steps

beqz t0, correct      # Check if t0 (error position) is zero; if not, correct error
correct:
       li t2, 1
       t2, t2, t0
       srli t2, t2, 1
       xor t0, a0, t2     # Correct error by flipping the erroneous bit

mv ra, t1             # Restore return address
```

6. `hamming_unmap_asm` Function

Reversing Bitwise Mapping: After decoding, this function reverses the mapping done before encoding to restore the original message bits to their correct positions

```
andi t1, t0, 4        # Isolate specific bits
srli t1, t1, 2        # Shift right to move bits back to their original position

srli t0, t0, 3        # Shift the entire register to handle other bits
or t0, t0, t1         # Combine bits to their original positions

# Additional operations handle other sections of the message
```

Restoring Original Message: The operations logically reverse the transformations applied during the mapping phase, using similar bit manipulation techniques to ensure the final data represents the original message.

VI. Conclusion: Advantages and Disadvantages of Hamming Code :

Advantages

1. **Error Detection and Correction**:
    - **Single-Bit Error Correction**: Hamming code can detect and correct single-bit errors, which significantly enhances data integrity in communication systems. This ability is crucial in applications where even a single bit error can lead to significant issues, such as in satellite communications and data transmission systems .
    - **Multiple Error Detection**: While Hamming code primarily corrects single-bit errors, it can also detect two-bit errors, providing an additional layer of data reliability.
2. **Efficiency**:
    - **Low Redundancy**: Compared to other error-correcting codes, Hamming code introduces relatively low redundancy. This efficiency makes it suitable for systems where bandwidth and storage are limited.
    - **Speed**: The simplicity of Hamming code's algorithm allows for fast encoding and decoding processes. This speed is beneficial in real-time applications and systems with limited processing power.
3. **Versatility**:
    - **Wide Application Range**: Hamming code is used across various fields, from wireless communications to computer memory systems, due to its robustness and simplicity. Its ability to maintain data integrity over noisy channels is a key advantage in these diverse applications .

4. **Ease of Implementation**:
    - **Hardware and Software**: Hamming code is relatively straightforward to implement in both hardware and software, making it accessible for many different types of systems. This ease of implementation helps in designing reliable digital communication and storage systems without extensive complexity .

Disadvantages

1. **Limited Error Correction Capability**:
    - **Single-Bit Focus**: Hamming code can only correct single-bit errors and detect double-bit errors. It is not effective for correcting burst errors or multiple random errors, limiting its usefulness in environments with high error rates .
2. **Redundancy Overhead**:
    - **Additional Bits**: Although Hamming code introduces less redundancy than some other methods, it still adds extra parity bits to the original data. This overhead can be a disadvantage in systems where every bit of data and parity is critical due to extremely limited bandwidth or storage capacity .
3. **Error Detection Limitations**:
    - **Multiple Error Detection**: While Hamming code can detect two-bit errors, it cannot correct them. This limitation means that in cases where multiple errors occur, Hamming code may not provide the necessary correction, requiring more advanced or additional error-correcting schemes .
4. **Complexity for Large Data Blocks**:
    - **Increased Complexity**: For large blocks of data, the number of parity bits required increases, which can complicate the encoding and decoding process. This added complexity can be a drawback in systems where simplicity and minimal resource usage are paramount .

In summary, Hamming code is a robust and efficient error-correcting code widely used in various digital communication and storage systems. Its primary strengths lie in its ability to correct single-bit errors, low redundancy, and ease of implementation. However, its limitations in handling multiple errors and the overhead of additional parity bits must be considered when deciding its suitability for specific applications.