Hamming Code Error correction implementation and translation to RISC-V Assembly

Othmane AZOUBI

June 1, 2024

othmane.azoubi@um6p.ma

Preface

This report focuses on the second assignment of Computer Organization and Architecture under the supervision of Professor Mohammed ATIBI.
Following thorough research from various sources listed at the end of this report, I have elaborated on Hamming error-correcting codes. This includes defining the tools and algorithms used to support my implementation choices in RISC-V Assembly.

I. Introduction

Definition

Richard Hamming developed the concept of Hamming codes, which are a class of linear error-correcting codes. Even though advanced methods are now available, Hamming's work was revolutionary. He was the first to take on the difficult task of detecting and fixing errors in data storage and transmission. This established the basis of preserving data integrity across noisy channels. In this report, we have the opportunity to look into Hamming code. it's a 1950 technique originally created by Richard Hamming to fix data mistakes.
Throughout this report, I'll explain how it works, and walk through a basic implementation in RISC V Assembly while explaining the algorithms used.

Backstory

When Bell Labs began developing the Bell Model V computer in the late 1940s, Richard Hamming, the creator of the Hamming codes, was a member of the team. The machine's cycle times were measured in seconds and it was powered by electromechanical relays. Punched paper tape, measuring around seven-eighths of an inch in width and with up to six holes per row, was used to insert input. On weekdays, the machine would stop if relay problems were found, alerting operators with flashing lights to fix the problem. But when operators weren't around on the weekends or during off-peak hours, the machine would carry on automatically to the next duty.
Hamming, who worked regularly on the weekends, was getting more and more irritated with having to completely restart his programs every time a mistake was found. "Damn it, if the machine can detect an error, why can't it locate the position of the error and correct it?" he said in an interview, expressing his anger. His determination to find a solution to the error-correction issue was motivated by this dissatisfaction. In the years that followed, he created a number of algorithms to deal with this problem. He presented the technique that is now known as Hamming code in 1950; it is still used in modern technologies like ECC memory.

II. Applications, Advantages and Disadvantages

Applications

1. **Satellites:**
   Hamming codes are essential in satellite communication systems. They ensure that data transmitted through space remains accurate, even when there are potential interferences or distortions. By detecting and correcting errors, they help maintain the integrity of communication signals.

2. **Computer Memory:**
   In computer memory systems, like ECC RAM modules, Hamming codes play a crucial role. They help identify and fix memory errors, making data storage and retrieval more reliable in computer systems.

3. **Modems:**
   Hamming codes are used in modems, especially when transmitting data over noisy channels like telephone lines. They ensure the accuracy of data transfer by detecting and correcting errors caused by signal distortions.

4. **PlasmaCAM:**
   For systems like PlasmaCAM, which rely on precise cutting instructions, Hamming codes can be valuable. They help identify and correct errors in the instructions sent to the system, ensuring accurate cutting operations.

5. **Open Connectors:**
   In systems using open connectors for data transmission, such as industrial automation or networking equipment, Hamming codes maintain data integrity. They detect and correct errors that may occur during data transmission, ensuring reliable communication.

6. **Shielding Wire:**
   Hamming codes find application in systems using shielding wire, like those in aerospace or automotive industries. They detect and correct data corruption or noise introduced along the wire, ensuring signal integrity.

7. **Embedded Processors:**
   Embedded processors, found in various electronic devices, benefit from Hamming codes for error detection and correction. By integrating Hamming code algorithms, data integrity is maintained, enhancing system reliability and performance.

Advantages

- Hamming code is effective on networks where data streams are vulnerable to single-bit errors.

- Besides detecting bit errors, Hamming code also identifies the specific bit containing the error, allowing for correction.

- Hamming codes are well-suited for computer memory and single-error correction applications due to their ease of use.

Limitations

- Hamming code is only capable of detecting and correcting single-bit errors and cannot handle multiple-bit errors.
- For Hamming code to work correctly, a substantial number of extra data bits must be added to the original message, which increases both the transmission time and the bandwidth usage.
- Hamming code lacks security because it can only identify errors that occur accidentally during data transmission, not intentional mistakes.

III. Former definitions

Before we discuss how the Hamming code algorithm works, first we need to define some concepts, helper algorithms and tools to aid us:

1. Parity bits

Parity: If counting the number of ones in your binary sequence results to be an even number, we say that the total parity is even. Otherwise, we say it's odd

1	0	0	1
0	1	0	1
1	1	0	0
0	0	1	1

(a) Example of an even block

0	1	1	0
1	0	1	0
1	0	0	0
0	1	0	1

(b) Example of an odd block

Figure 1: Examples of parity

Parity bit: Is a bit in a block of binary, that helps us detect and correct errors in transmitted data. It always ensures that the total number of bits (including the parity bit itself) is always even.

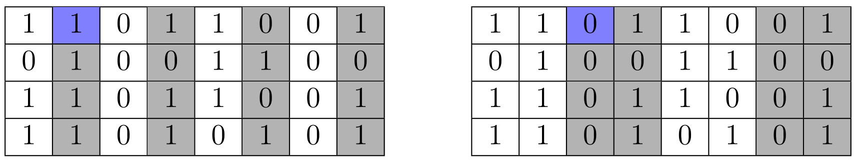
2. Power of two check

A power of two is a number that has only a single bit in it. This could be easily proven by the fact that $2^k = \underbrace{2 \times 2 \times \ldots \times 2}_{k \text{ times}}$ is equivalent to 1 (2^0) being bit shifted k times to the left $\underbrace{10\ldots\ldots\ldots 00_2}_{k\text{times}}$. This observation leads us to conclude that $2^k - 1 = \underbrace{011\ldots\ldots\ldots 11_2}_{k \text{ times}}$.

Thus, $\forall k \in \mathbb{N}\ (2^k)\&(2^k - 1) = 0$. Determining whether a number is a power of two, will help us knowing the positions of parity bits since in Hamming code they are positioned in bits where the position is a power of two.

3. Error Code, Detection and Correction

It's the code that is made out of $C = (C_k \ldots C_2 C_1 C_0)_2$, with C_i $(0 \leq i \leq k)$ being the parity of the region that contains the $i-th$ parity bit (the sub-block that skips i bits checks i bits after the parity bit).



1	1	0	1	1	0	0	1
0	1	0	0	1	1	0	0
1	1	0	1	1	0	0	1
1	1	0	1	0	1	0	1

(a) C_1 sub-block(checks 1 skips 1)

1	1	0	1	1	0	0	1
0	1	0	0	1	1	0	0
1	1	0	1	1	0	0	1
1	1	0	1	0	1	0	1

(b) C_2 sub-block(checks 2 skips 2)

Figure 2: 32-bit blocks with C_1 and C_2 sub-blocks highlighted

A received message is called single-bit erroneous when $C \neq 0$. We can locate the position of such bit by computing $C - 1$, and thus correct a received message X by inverting its $(C-1)$-th bit using: $X \oplus (1 << (C-1))$.

4. Brian Kernighan's Algorithm

First, let's try to turn off the rightmost set bit of a number n. To do this, we will consider two cases:

- n **is even:** In this case, n is divisible by 2^k with $k \in \mathbb{N}$. n is written in the format $n = \ldots\ldots\underbrace{10\ldots00}_{\text{k times}}{}_2$, and $n - 1 = \ldots\ldots\underbrace{01\ldots11}_{k\text{ times}}{}_2$.
  We conclude that $n\&(n-1)$ is n but with the rightmost set bit turned off.
- n **is odd:** In this case, $n - 1$ is n but with the rightmost set bit turned off. This can also be written as $n\&(n-1)$.

Conclusion: The expression $n\&(n-1)$ can be used to turn off the rightmost set bit of a number n.

For example, consider the number $52 = 0011\ 0100_2$:

$$\begin{array}{rl} n & = 0011\ 0100 \\ n-1 & = 0011\ 0011 \\ \hline n\&(n-1) & = 0011\ 0000 \end{array}$$

Figure 3: Example taken from cp-algorithms.com

We can count the number of '1' in n using the above expression. The idea is to keep turning off the rightmost set bit (after counting it), till $n = 0$. Here is an implementation:

```c
int countSetBits(int n) {
    int count = 0;
    while (n) {
        n = n & (n - 1);
        count++;
    }
    return count;
}
```

This algorithm will help us determine the parity of a binary block by checking the parity of the returned "count".

IV. Hamming code algorithm (32bits)

In this section, I will use all the tools and algorithms explained above to construct the Hamming code algorithm. I will first provide the algorithm in pseudo-code, and then write its RISC-V counterpart. But first, I will give the definitions of the *get_message*() and *parity* functions (Section III.4) in RISC-V:

```
parity:
    # n = a0 and return value in a7
    li a7, 0
    loop_parity:
        beqz a0, end_loop_parity
            addi a7, a7, 1
            addi t0, a0, -1
            and a0, a0, t0
        j loop_parity

    end_loop_parity:
    andi a7, a7, 1
    jr ra


get_message:
    #matricule = a0, r = a1 and return value in a7
    xori t0, a0, 0xFFFFFFFF

    sll t1, t0, a1  # (msg << r)
    li t3, 32
    sub t2, t3, a1  # 32 - r
    srl t2, t0, t2 # (msg >> (32 - r))
    or t0, t1, t2  # msg = (msg << r) | (msg >> (32 - r))

    li t3, 0x00FFFFFF
    and a7, t0, t3
    jr ra
```

1. Hamming map:

It's the process of taking 24bits of data and placing them in a 32bits message while keeping away from bits that represents powers of 2 (Sections III.1 and III.2). Let's consider the following example:

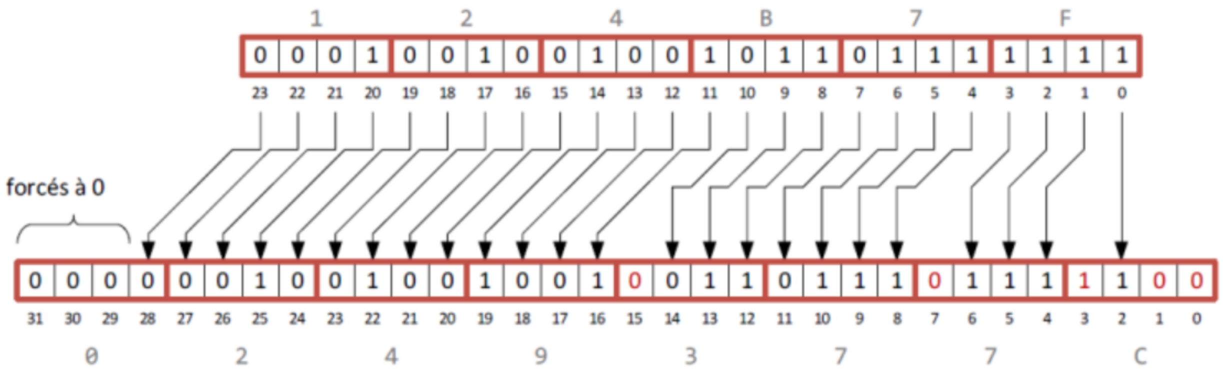


Figure 4: Example hamming map

The algorithm behind it is straight forward:

Algorithm 1 Hamming Map

```
1: function HAMMING_MAP(msg)
2:     $hamming \leftarrow 0$
3:     $bit\_position \leftarrow 0$
4:     for $i \leftarrow 0$ to 31 do
5:         if $i \neq 0$ and $(i\&(i-1)) \neq 0$ then          ▷ skipping parity bit positions
6:             if $msg\&(1 \ll bit\_position)$ then
7:                 $hamming \leftarrow hamming \mid (1 \ll 31)$
8:             end if
9:             $bit\_position \leftarrow bit\_position + 1$
10:        end if
11:        $hamming \leftarrow hamming \gg 1$
12:    end for
13:    return $hamming$
14: end function
```

RISC-V Implementation:

```
hamming_map:
    #a0 = data and return value in a7
    li t0, 0 #current position in the data
    li a7, 0

    #a7 = hamming
    #t0 = bit_position
    #a0 = msg

    li t1 , 0
    loop_map:

        #Parity bits checking:
        beqz t1, map_skip_parity # check if t1 == 0
        addi t2, t1, -1
        and t2, t1, t2
        beqz t2, map_skip_parity # check if (t1 & (t1 - 1)) == 0

        li t2, 1
        sll t2, t2, t0
        and t2, a0, t2 # t2 = msg & (1 << bit_position)
        beq t2, x0, map_zero
            li t2, 1
            slli t2, t2, 31
            or a7, a7, t2 # hamming = hamming | (1 << (bit_position))
        map_zero:
        addi t0, t0, 1  # bit_position++

        j map_skip_parity_end
        map_skip_parity:
            li t2, 0x7FFFFFFF
            and a7, a7, t2 # Forcing parity bits to 0
        map_skip_parity_end:

        srli a7, a7, 1 # Shift hamming to the right so we can process the
            ↪ next bit

    addi t1, t1, 1
    li t2, 32
    bne t1, t2, loop_map

    jr ra
```

2. Hamming encode:

It's the process of setting the parity bits in our previously obtained Hamming map. Where each parity bit is responsible over the parity of it's sub-block (Consider Figure 2 as an example), Here is a quick algorithm to achieve the encoding:

Algorithm 2 Hamming Encode

```
1:  function HAMMING_ENCODE(map)
2:      $code \leftarrow map$                          ▷ Initialize code with map
3:      $p \leftarrow$ PARITY($map$ & 0x55555554)
4:      $code \leftarrow code \mid p$
5:      $p \leftarrow$ PARITY($map$ & 0x66666664)
6:      $code \leftarrow code \mid (p << 1)$
7:      $p \leftarrow$ PARITY($map$ & 0x78787870)
8:      $code \leftarrow code \mid (p << 3)$
9:      $p \leftarrow$ PARITY($map$ & 0x7F807F00)
10:     $code \leftarrow code \mid (p << 7)$
11:     $p \leftarrow$ PARITY($map$ & 0x7FFF0000)
12:     $code \leftarrow code \mid (p << 15)$
13:     return $code$
14: end function
```

RISC-V Implementation:

```
hamming_encode:
    #a0 = map and return value in a7
    mv t6, a0 # return
    mv t5, a0 # map
    mv t0, x0 # Parity bit

    addi sp, sp, -4
    sw ra, 0(sp)

    li t3, 0x55555554
    and a0, t5, t3 # map & 0x55555554
    jal parity # parity(map & 0x55555554)
    or t6, t6, a7

    li t3, 0x66666664
    and a0, t5, t3 # map & 0x66666664
    jal parity # parity(map & 0x66666664)
    slli a7, a7, 1
    or t6, t6, a7

    li t3, 0x78787870
    and a0, t5, t3 # map & 0x78787870
    jal parity # parity(map & 0x78787870)
    slli a7, a7, 3
    or t6, t6, a7

    li t3, 0x7F807F00
    and a0, t5, t3 # map & 0x7F807F00

    jal parity # parity(map & 0x7F807F00)
    slli a7, a7, 7
    or t6, t6, a7

    li t3, 0x7FFF0000
    and a0, t5, t3 # map & 0x7FFF0000
    jal parity # parity(map & 0x7FFF0000)
    slli a7, a7, 15
    or t6, t6, a7

    lw ra, 0(sp)
    addi sp, sp, 4

    mv a7, t6
    jr ra
```

3. Hamming decode

After receiving the message we compute the parity of all sub-blocks to create an Error code C like discussed in Section III.3 the algorithm behind it is also simple:

Algorithm 3 Hamming Decode

```
1:  function HAMMING_DECODE($code$)
2:      $c0 \leftarrow$ PARITY(($code\&0x55555555$))
3:      $c1 \leftarrow$ PARITY(($code\&0x66666666$))
4:      $c2 \leftarrow$ PARITY(($code\&0x78787878$))
5:      $c3 \leftarrow$ PARITY(($code\&0x7F807F80$))
6:      $c4 \leftarrow$ PARITY(($code\&0x7FFF8000$))
7:      $C \leftarrow c0$ OR $(c1 << 1)$ OR $(c2 << 2)$ OR $(c3 << 3)$ OR $(c4 << 4)$
8:      if $C \neq 0$ then
9:          $code \leftarrow code \oplus (1 << (C - 1))$
10:     end if
11:     return $code$
12: end function
```

RISC-V Implementation:

```
hamming_decode:
    #a0 = code and return value in a7
    mv t6, a0
    li t5, 0 # Error code

    li t0, 0x55555555
    li t1, 0x66666666
    li t2, 0x78787878
    li t3, 0x7F807F80
    li t4, 0x7FFF8000

    and t0, t6, t0 # code & 0x55555555
    and t1, t6, t1 # code & 0x66666666
    and t2, t6, t2 # code & 0x78787878
    and t3, t6, t3 # code & 0x7F807F80
    and t4, t6, t4 # code & 0x7FFF8000

    addi sp, sp, -4
    sw ra, 0(sp)

    mv a0, t0
    jal parity
    slli a7, a7, 0
    or t5, t5, a7

    mv a0, t1
    jal parity
    slli a7, a7, 1
    or t5, t5, a7

    mv a0, t2
    jal parity
    slli a7, a7, 2
    or t5, t5, a7

    mv a0, t3
    jal parity
    slli a7, a7, 3
    or t5, t5, a7

    mv a0, t4
    jal parity
    slli a7, a7, 4

    or t5, t5, a7

    lw ra, 0(sp)
    addi sp, sp, 4

    beqz t5, no_error  # if C != 0, there is an error
        li t1, 1
        addi t0, t5, -1 # t0 = C - 1
        sll t1, t1, t0  # 1 << (C - 1)
        xor t6, t6, t1  # code = code ^ (1 << (C - 1))
    no_error:


    mv a7, t6
    jr ra
```

4. Hamming unmap

It’s the process of applying the error correction formula, as seen in Section III.3, and extracting the original 24-bit data from the 32-bit received message after correction. Here is the algorithm behind it:

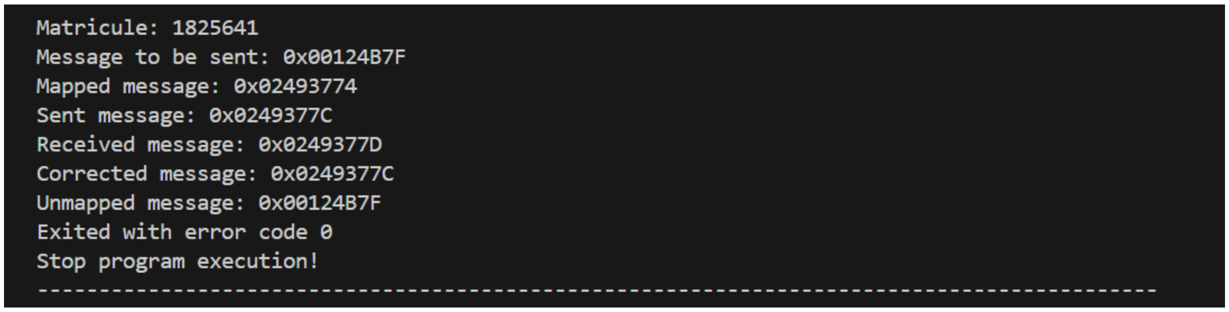
RISC-V Implementation:

Algorithm 4 haming_unmap

```
 1: function HAMING_UNMAP(map)
 2:     $hamming \leftarrow map$                                  ▷ Initialize hamming with map
 3:     $matricule \leftarrow 0$
 4:     $bit\_position \leftarrow 0$
 5:     for $i \leftarrow 0$ to 31 do
 6:         if $((i+1)\&i) == 0$ then
 7:             continue
 8:         else
 9:             if $(hamming \ \& \ (1 \ll i))$ then
10:                 $matricule \leftarrow matricule \mid (1 \ll bit\_position)$
11:             end if
12:             $bit\_position \leftarrow bit\_position + 1$
13:         end if
14:     end for
15:     return $matricule$
16: end function
```

```
hamming_unmap:
    # a0 = map and return value in a7
    li t0, 0 # bit position
    li a7, 0 # matricule
    # a0 = hamming

    li t1, 0 # index
    loop_unmap:

        # Parity bits checking:
        addi t2, t1, 1
        and t2, t1, t2
        beqz t2, unmap_skip_parity # check if (t1 & (t1 - 1)) == 0

        li t2, 1
        sll t3, t2, t1 # t3 = hamming & (1 << i)
        and t3, a0, t3
        beqz t3, unmap_zero
            li t3, 1
            sll t3, t3, t0 # (1 << bit_position)
            or a7, a7, t3 # matricule = matricule | (1 << bit_position)
        unmap_zero:
        addi t0, t0, 1  # bit_position++

        unmap_skip_parity:

        addi t1, t1, 1
        li t2, 32
        bne t1, t2, loop_unmap

    jr ra
```

5. Output



```
Matricule: 1825641
Message to be sent: 0x00124B7F
Mapped message: 0x02493774
Sent message: 0x0249377C
Received message: 0x0249377D
Corrected message: 0x0249377C
Unmapped message: 0x00124B7F
Exited with error code 0
Stop program execution!
-------------------------------------------------------------------------------------------
```

Figure 5: Example hamming map

Sources

- https://cp-algorithms.com/algebra/bit-manipulation.html
- https://www.youtube.com/watch?v=X8jsijhllIA
- https://www.geeksforgeeks.org/hamming-code-in-computer-network/
- https://www.techtarget.com/whatis/definition/Hamming-code
- https://en.wikipedia.org/wiki/Hamming_code
- https://zoo.cs.yale.edu/classes/cs323/doc/Hamming.pdf
- https://www.naukri.com/code360/library/examples-for-hamming-code