Laboratorio codificación Hamming (7,4)

Belky Valentina Girón López
est.belky.giron@unimilitar.edu.co
Docente: José De Jesús Rúgeles

Resumen — En esta práctica trabajamos con la codificación Hamming (7,4), que sirve para proteger la transmisión de datos corrigiendo errores. La idea fue tomar un dato de 16 bits del acelerómetro dividirlo en 4 grupos de 4 bits y luego aplicar la codificación Hamming a cada grupo.

El resultado fue que cada grupo de 4 bits se convirtió en 7 bits, de modo que la trama completa pasó de 16 a 28 bits. Esto sucede porque el código agrega bits para poder detectar y corregir errores de un solo bit.

Lo más importante de la práctica fue ver cómo, a través de este proceso, se garantiza que los datos transmitidos sean más confiables, incluso si ocurren alteraciones en el camino. Aunque al inicio puede parecer que se gasta espacio al aumentar el número de bits, la ventaja es que la comunicación se vuelve mucho más segura.

Abstract **--** In this lab, we worked with Hamming (7,4) coding, which is used to protect data transmission by correcting errors. The idea was to take a 16-bit data point from the accelerometer, divide it into four groups of four bits, and then apply Hamming coding to each group.

The result was that each group of four bits became seven bits, so the entire frame went from 16 to 28 bits. This happens because the code adds bits to detect and correct single-bit errors.

The most important aspect of the lab was to see how, through this process, the transmitted data is guaranteed to be more reliable, even if alterations occur along the way. Although at first, it may seem that increasing the number of bits wastes space, the advantage is that communication becomes much more secure.

I. Introducción

En este laboratorio exploramos cómo funciona la codificación Hamming (7,4) aplicada a datos de un sensor. El objetivo fue comprender de manera práctica cómo un mensaje puede ser protegido contra errores al transmitirse, agregando bits de paridad que permiten detectar y corregir fallos. A través del ejercicio con la Raspberry Pi Pico 2W y la simulación de datos, pudimos ver de forma sencilla cómo se transforma una señal de 16 bits en una trama más robusta de 28 bits, entendiendo así la importancia de la redundancia en las comunicaciones digitales.

II. Desarrollo de experimentos

Para el montaje del laboratorio se usa una Raspberry Pi Pico 2W conectada al acelerómetro MPU6050 a través del bus I2C.

El microcontrolador tiene la tarea de tomar 16 bits, dividirlos en cuatro grupos de 4 bits, aplicar a cada grupo el algoritmo de Hamming (7,4) para codificarlos, y finalmente generar una trama de 28 bits. Esta trama se transmite por la UART, y podemos visualizarla tanto en un osciloscopio como en la consola del sistema.

Paso 1: Registro inicial

El punto de partida es un registro de 16 bits, al que llamamos AX. En este caso, el valor tomado fue:

AX = 111111110111100

Este registro representa la información tomada del acelerómetro que más adelante vamos a codificar con Hamming.

Paso 2: División

Para poder aplicar el código Hamming (7,4), lo primero que hacemos es dividir la palabra de 16 bits en 4 grupos más pequeños de 4 bits, conocidos como *nibbles*. Así quedan distribuidos:

- D (bits 15..12): 1111
- C (bits 11..8): 1111
- B (bits 7..4): 1011
- A (bits 3..0): 1100

Esta separación facilita el trabajo, ya que el algoritmo solo puede aplicarse a bloques de 4 bits.

Paso 3: Codificación Hamming (7,4)

Luego, a cada bloque de 4 bits se le aplica el algoritmo Hamming (7,4), que convierte los 4 bits originales en 7 bits codificados. Este proceso agrega bits con el fin de detectar y corregir errores en la transmisión:

- D: 1111 → 1111111
- C: 1111 → 1111111
- B: 1011 → 1010101
- A: 1100 → 1100001

Así, cada bloque original de 4 bits ahora ocupa 7 bits, lo cual explica el crecimiento de la trama.

Paso 4: Trama final

Finalmente, al concatenar todos los grupos de bits codificados, se obtiene una nueva palabra de 28 bits:

Trama final = 1111111111110101011100001

Esta es la versión protegida de la información original, con el aumento de bits suficiente para que, si ocurre un error en un solo bit, pueda ser detectado y corregido en el receptor.

Esto lo podemos ver en la consola del Thonny a partir del código ejecutado:

CODIGO:

```
from machine import Pin, I2C, UART
import time
from hamming74 import hamming74_encode,
hamming74_decode

# Configuración I2C con GP4 y GP5
i2c = I2C (1, scl = Pin(5), sda = Pin(4), freq=400000)

# UART para osciloscopio
uart = UART (0, baudrate=9600, tx=Pin(12), rx=Pin(13))

# Dirección MPU6050
MPU6050_ADDR = 0x68

def setup_mpu6050():
    """Configurar el MPU6050"""
    i2c.writeto_mem(MPU6050_ADDR, 0x6B, b'\x00')  #
Despertar
    time.sleep(0.2)

def scan_i2c():
    """Escanear dispositivos I2C"""
    devices = i2c.scan()
    print("Dispositivos I2C encontrados:", [hex(d) for d in
devices])
    return devices

def read_accel():
    """Leer datos del acelerómetro (16 bits por eje)"""
    try:
        # Leer 6 bytes del registro 0x3B
        data = i2c.readfrom_mem(MPU6050_ADDR, 0x3B, 6)

        # Convertir a valores de 16 bits
        accel_x = (data[0] << 8) | data[1]
        accel_y = (data[2] << 8) | data[3]
        accel_z = (data[4] << 8) | data[5]

        # Ajustar complemento a 2
        if accel_x > 32767: accel_x -= 65536
        if accel_y > 32767: accel_y -= 65536
        if accel_z > 32767: accel_z -= 65536

        return accel_x, accel_y, accel_z

    except Exception as e:
        print(f"Error leyendo acelerómetro: {e}")
        return 0, 0, 0

def split_16bit_to_nibbles(value):
    """Dividir valor de 16 bits en 4 nibbles de 4 bits"""
    value = abs(value) & 0xFFFF
    nibbles = [ ]

    for i in range(4):
        shift = 4 * (3 - i)  # 12, 8, 4, 0
        nibble_val = (value >> shift) & 0x0F

        # Convertir a lista de bits [d3, d2, d1, d0] (MSB -> LSB)

        bits = [
            (nibble_val >> 3) & 1,
            (nibble_val >> 2) & 1,
            (nibble_val >> 1) & 1,
            (nibble_val >> 0) & 1
        ]
        nibbles.append(bits)

    return nibbles
```

```
def encode_sample(sample_16bit):
    """Codificar muestra de 16 bits usando Hamming (7,4)"""
    print (f"Valor original: 0x{sample_16bit:04X} =
{sample_16bit:016b}")

    # 1. Dividir en 4 nibbles
    nibbles = split_16bit_to_nibbles(sample_16bit)
    print ("Nibbles obtenidos:")

    for i, nibble in enumerate(nibbles):
        nibble_val = nibble[0]*8 + nibble[1]*4 + nibble[2]*2 +
nibble[3]
        print(f" Nibble {i+1}: {nibble} = 0x{nibble_val:X}")

    # 2. Aplicar Hamming a cada nibble
    encoded_bits = []
    for nibble in nibbles:
        encoded = hamming74_encode(nibble)
        encoded_bits.extend(encoded)
        nibble_val = nibble[0]*8 + nibble[1]*4 + nibble[2]*2 +
nibble[3]
        print(f" Nibble 0x{nibble_val:X} → Hamming:
{encoded}")

    print(f"Bits codificados totales ({len(encoded_bits)}):
{encoded_bits}")
    return encoded_bits


def bits_to_bytes(bits):
    """Convertir lista de bits a bytes para transmisión UART"""
    bytes_list = bytearray()

    for i in range(0, len(bits), 8):
        byte_bits = bits[i:i+8]

        while len(byte_bits) < 8:
            byte_bits.append(0)

        byte_val = 0
        for bit in byte_bits:
            byte_val = (byte_val << 1) | bit

        bytes_list.append(byte_val)

    return bytes_list

def main():
    print("Iniciando Laboratorio Hamming MPU6050...")
    print("Pines I2C: SDA=GP14, SCL=GP15")
    print("UART TX: GP12 (para osciloscopio)\n")
    # Escanear I2C
    devices = scan_i2c()

    if MPU6050_ADDR not in devices:
        print(" MPU6050 no detectado. Verificar conexiones.")
        print(" - VCC → 3.3V")
        print(" - GND → GND")
        print(" - SDA → GP14 (Pin 19)")
        print(" - SCL → GP15 (Pin 20)")
        return

    print("MPU6050 detectado correctamente!")

    # Configurar MPU6050
    setup_mpu6050()
    sample_count = 0

    while True:
        try:
            # Leer acelerómetro
            accel_x, accel_y, accel_z = read_accel()

            # Usar el eje X para codificación
            sample = accel_x

            # Codificar muestra
            encoded_bits = encode_sample(abs(sample) &
0xFFFF)

            # Transmitir por UART
            encoded_bytes = bits_to_bytes(encoded_bits)
            uart.write(encoded_bytes)

            sample_count += 1
            print(f"Muestra #{sample_count} transmitida:
{list(encoded_bytes)}")
            print("-" * 50 + "\n")

            time.sleep(2)

        except Exception as e:
            print(f"Error: {e}")
            time.sleep(1)


if __name__ == "__main__":
    main()
```

Al ejecutar el anterior código obtenemos el siguiente resultado en la consola:

```
==================================
AX (16 bits): 1111111110111100
D (bits15..12) -> 1111
Codificado: 1111111
C (bits11..8) -> 1111
Codificado: 1111111
B (bits7..4) -> 1011
Codificado: 1010101
A (bits3..0) -> 1100
Codificado: 1100001
Trama final (28 bits): 1111111111111110101
011100001
==================================
```

Fig. 1. Resultado de la consola del Thonny.

Luego analizamos los datos obtenidos en la pantalla Olled

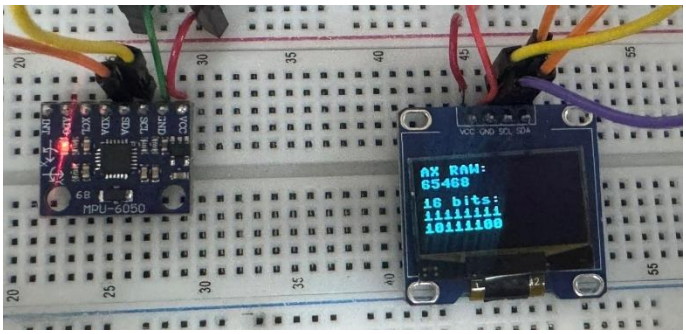

Fig. 2. datos del código ejecutado en la pantalla olled.

AX RAW: 65468

- Aquí se muestra el valor leído directamente desde el acelerómetro MPU6050 en el eje X. Ese número corresponde al dato real tomado en 16 bits, antes de ser procesado o convertido a unidades físicas.

16 bits:

- Justo debajo aparece la representación binaria del mismo dato, es decir, cómo se ve el número almacenado en los 16 bits del registro AX.

El valor se divide en dos líneas para que pueda entrar en la pantalla:

- 1111111110111100
-

Relación con el proceso Hamming

- Ese valor binario de 16 bits es el que más adelante se separa en 4 nibbles.
- Cada nibble pasa por el algoritmo Hamming (7,4), que agrega bits de paridad para generar la trama de 28 bits.
- En la OLED se ve la entrada del proceso, mientras que en la UART y el osciloscopio se observa la trama codificada transmitida.

La pantalla OLED está mostrando el dato real (RAW) del acelerómetro y su equivalente en binario de 16 bits, que es exactamente el punto de partida para el procedimiento de división en nibbles y codificación Hamming que ya analizamos.

Ahora analizaremos la imagen obtenida en el osciloscopio:

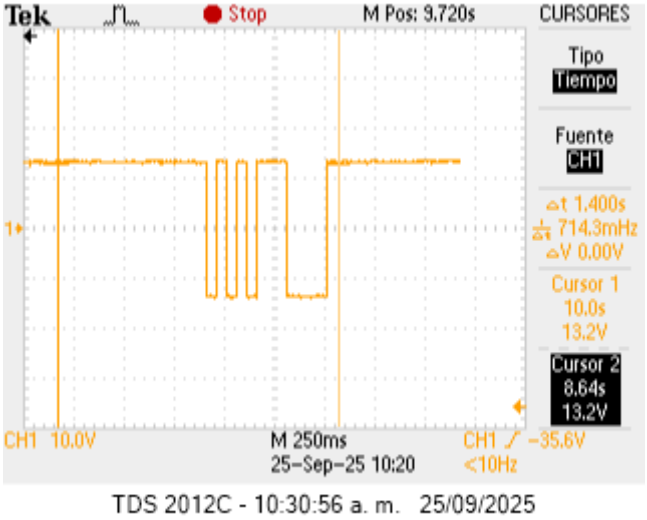

Fig. 3. Datos obtenidos en el osciloscopio.

En la captura del osciloscopio se observa la señal digital transmitida por la UART de la Raspberry Pi Pico 2W.

Lo que vemos son niveles lógicos que cambian entre alto (1) y bajo (0), correspondientes a cada bit de la trama codificada con Hamming (7,4).

- Cada bloque de bits transmitido forma parte de la trama de 28 bits, que proviene de la codificación de los 16 bits iniciales.
- La señal inicia con un bit de inicio (start bit), que baja el nivel para indicar al receptor que comienza la transmisión.
- Luego, bit a bit, se transmiten los datos codificados, que corresponden a los nibbles después de pasar por el algoritmo Hamming.
- Finalmente, aparece el bit de parada (stop bit), que lleva la señal de nuevo al estado alto, marcando el final de la transmisión.

En la pantalla del osciloscopio, cada pulso rectangular representa un bit. Si pausamos la señal y la leemos con cuidado, es posible reconstruir el patrón binario transmitido y verificar que coincide con la trama final de 28 bits calculada:

1111111111111010101100001

Esto confirma que la codificación Hamming se realizó correctamente y que los datos transmitidos pueden observarse tanto en consola como físicamente en la forma de onda.

Link repositorio GitHub:
https://github.com/belkyvalentina11/Comunicaci-n-digital-.git

REFERENCIAS

[1] Invarato, R. (2017, 12 agosto). *Código de Hamming: Detección y Corrección de errores - Jarroba*. Jarroba. https://jarroba.com/codigo-de-hamming-deteccion-y-correccion-de-errores/.