Proyecto 2: Codificador y decodificador MP3

Estudiante: Daniel Rojas Marín
Carnet: 2016089821

Estudiante: Pamela Salazar Espinoza
Carnet: 2022438314

1. Introducción

En el presente documento se muestra la implementación al alto nivel de un sistema de codificación y descodificación para audio MP3 con los siguientes requisitos de diseño:

a) El codificador se alimentará con muestras de audio de 16 bits, muestreadas a 8 kHz. Las muestras deben procesarse en bloques de 8 ms (64 muestras por bloque). Cada bloque se transformará en frecuencia utilizando una FFT de tamaño 64. Dado que sus entradas son valores reales, la mitad superior del espectro es simétrica y no es necesario para calcularlo. Puede utilizar una FFT para valores reales (conocida como RFFT), que sólo calcula la parte inferior del espectro. En este caso, desea tener 32 complejos coeficientes como una salida de su FFT. Los coeficientes se cuantificarán en valores de 8, 4, 2 o 1 bit. Estos coeficientes se empaquetarán en bytes y se escribirán en una salida expediente. Se recomienda diseñar un mapeo de coeficientes que resulte en un número entero de bytes para cada bloque.
b) El decodificador debe leer archivos comprimidos por su codificador y regenerar muestras de audio con una profundidad de bits de 16 bits y una frecuencia de 8 kHz. La primera etapa será extraer los coeficientes, y escalarlos a la escala lineal, para generar los coeficientes (complejos) de cada frecuencia. Entonces un IFFT de tamaño 64 será ejecutado. Se puede usar un IFFT para valores reales (RIFFT), por lo que solo necesita especificar los coeficientes complejos para la parte inferior del espectro. Debería generar 64 reales muestras. Las muestras generadas por la IFFT se concatenan para generar la señal de salida, que debe guardarse en un archivo.

El diseño se dividió en tres partes, primero se creó un prototipo en Python en el cual se obtuvo los vectores para usar como referencia para las siguientes etapas, el segundo paso fue la implementación de punto fijo y finalmente la implementación en Neon.

2. Prototipo y Vector de Prueba

2.1. Codificador

El prototipo fue creado en Python 3, la librería numpy cuenta con una función fft.rfft la cual devuelve la RFFT de la entrada. Primero se carga el archivo de audio a procesar, de forma que se obtiene el vector al que se le aplicara la rfft:

```
archivo = 'test_audio.wav'

faudio, audio = waves.read(archivo)
```

El vector obtenido se divide en muestras de 64 valores y se le aplica la rfft a cada una de ellas. Esto optimiza tiempo de ejecución con respecto a procesar la rfft para el vector complento. Como se utiliza la rfft la salida para un bloque de 64 muestras es de tamaño 32, se descarta la última frecuencia (la frecuencia de Nyquis):

```
def FFT_64(x):
   audio_len=len(x)
   samples_in=[0]*int(np.ceil(audio_len/64)*64)
   samples_in[0: audio_len]=x
   bloque=[0]*64
   fft_out=[0]*int(64/2)
   for bloque_index in range(0, len(samples_in),64):
      bloque=samples_in[bloque_index: bloque_index+64]
      bloque_temp=np.fft.rfft(bloque)
      if bloque_index==0:
         fft_out=bloque_temp[0:32]
      else:
         fft_out=np.concatenate((fft_out,bloque_temp[0:32]))
   ….
```

Para cuantizar la salida obtenida se utilizan 16 bits, los primeros 8 bits para la parte real y los otros 8 bits para la imaginar. De los 8 bits correspondientes, el primero es para el signo y los 7 restantes para la magnitud. Para convertir las magnitudes en valores que puedan ser representados en 7 bits se uso un facto de 1/406:

```
def FTT_64(x):
    …..
   real=np.asarray((np.real(fft_out))/406, dtype = int)
   imag=np.asarray((np.imag(fft_out))/406, dtype = int)
   output_vec=[0]*len(fft_out)
```

```
    for index in range(len(real)):
        output_vec[index]=complex_to_binary(real[index],imag[index])
    return  output_vec
```

```
def decimal_to_binary(numero_decimal):
    modulos = [0]*8
    a=numero_decimal
    if numero_decimal>0:
        bit_signo=0
    else:
        bit_signo=1
    modulos[0]=bit_signo
    numero_decimal=abs(numero_decimal)
    pos=7
    num_bin=0
    while numero_decimal != 0:
        modulo = numero_decimal % 2
        cociente = numero_decimal // 2
        modulos[pos]=modulo
        numero_decimal = cociente
        pos=pos-1
    return modulos


def complex_to_binary(real,imag):
    num_bin=0
    modulo_real=decimal_to_binary(real)
    modulo_imag=decimal_to_binary(imag)
    modulos=np.concatenate((modulo_real,modulo_imag))
    pos=0
```

```
    for digito in modulos[::-1]:
        num_bin=num_bin+digito*(2**pos)
        pos=pos+1
    return num_bin
```

Finalmente se guarda el vector obtenido en un archivo .txt para ser usado como

2.2. Decodificador

El decodificador lee el vector generado por el codificador. Primero recupera los valores de obtenidos por la rfft, recordando que los primeros 8 bits corresponde a la parte real y los restantes a la imaginaria, donde el primer bit corresponde al signo y los 7 restantes a la magnitud:

```
def binary_to_decimal(numero_decimal):
    a=numero_decimal
    modulos = [0]*16
    pos=7
    real=0
    imag=0
    while numero_decimal != 0:
        modulo = numero_decimal % 2
        cociente = numero_decimal // 2
        modulos[pos]=modulo
        numero_decimal = cociente
        pos=pos-1


    bit_signo=modulos[0]
    numero=modulos[2:8]
    bit_signo_imag=modulos[8]
    numero_imag=modulos[9:16]
```

```
    if bit_signo==0:
        factor=1
    else:
        factor=-1
    if bit_signo_imag==0:
        factor_imag=1
    else:
        factor_imag=-1
    pos=0
    for digito in numero[::-1]:
        real=real+digito*(2**pos)
        pos=pos+1
    pos=0
    for digito in numero_imag[::-1]:
        imag=imag+digito*(2**pos)
        pos=pos+1
    real=factor*real
    imag=factor_imag*imag
    complex=real+imag*1j
    return complex
```

Al vector descodificado se le aplica la ifft para recuperar la señal original:

```
def IFFT_64(x):
    fft_size=len(x)//2
    ifft_out=[0]*64
    pos=0
    samples=[0]*len(x)
```

```
for number in x:
    samples[pos]=binary_to_decimal(number)
    pos=pos+1
for bloque_index in range(0,fft_size,int(64/2)):
    bloque_temp=samples[bloque_index:bloque_index+int(64/2)]
    output_temp=np.fft.irfft(bloque_temp)
    if bloque_index==0:
        ifft_out=output_temp
    else:
        ifft_out=np.concatenate((ifft_out,output_temp))
ifft_out=np.asarray(np.around(ifft_out), dtype = int)
ifft_out=ifft_out*406
return ifft_out
```

En la figura 1 se muestra la señal de audio original y la recuperada por el decoder:

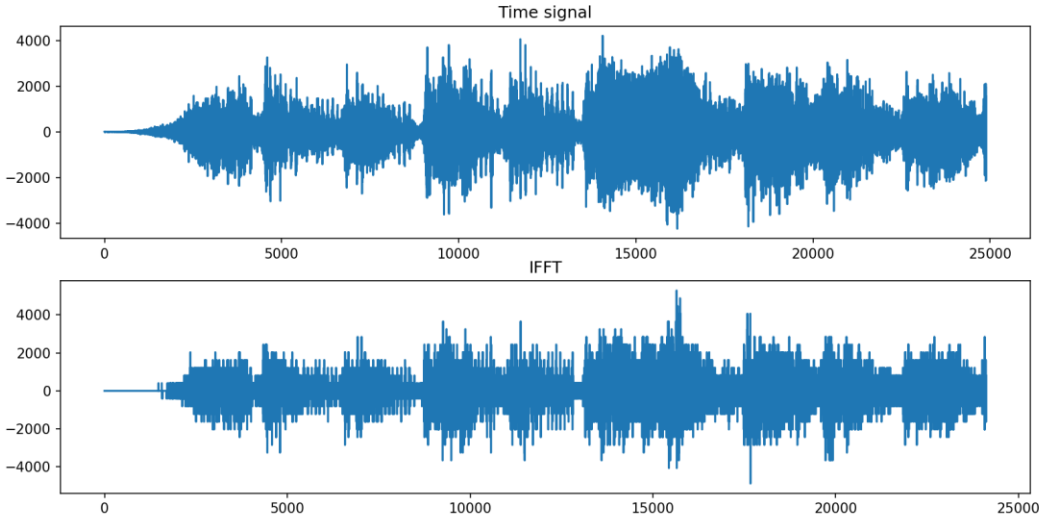

Figura 1. Señal de audio original y señal recuperada por el descodificador.

Como se puede ver en la figura 1 aunque al comprimir los datos se pierde algo de información con la compresión de los datos, se logra recupera una señal muy similar a la original.

3. Implementación de Punto Fijo

En esta esta etapa de implemento el codificado y descodificadores diseñados en la primera etapa, el código usado se encuentra el git del proyecto. Para el encoder se obtuvo un error absoluto promedio de 14223. 23. En la figura 2 se ve los resultados obtenidos por el encoder en Python y el encoder de punto fijo. Como se observa en el grafico el codificador de punto fijo pierde varios de los valores los que aumenta el error, sin embargo la forma de señal de frecuencia obtenida se mantiene.

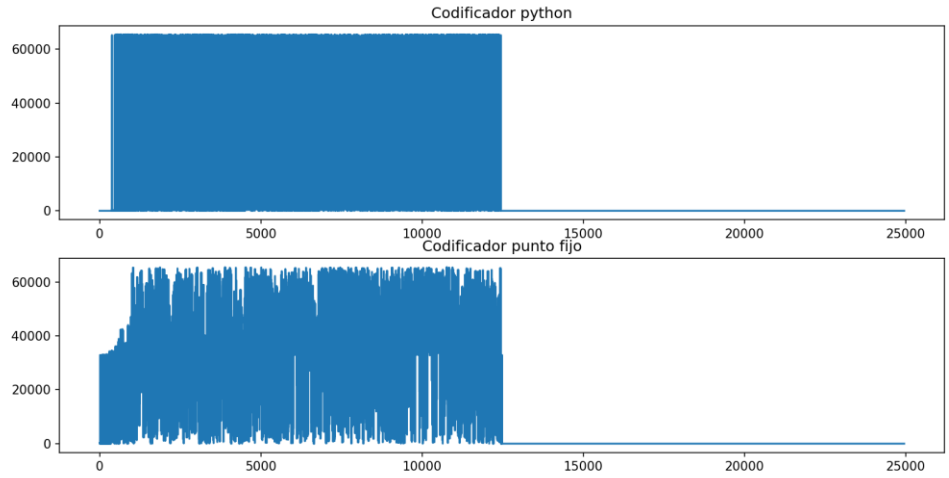

Figura 2. Resultados de codificador implementado en Python vs codificador implementado con aritmética de punto fijo en C.

En el caso del decoder el error absoluto promedio fue 616.56. En la figura 3 se presenta la diferencia entre los vectores obtenidos.

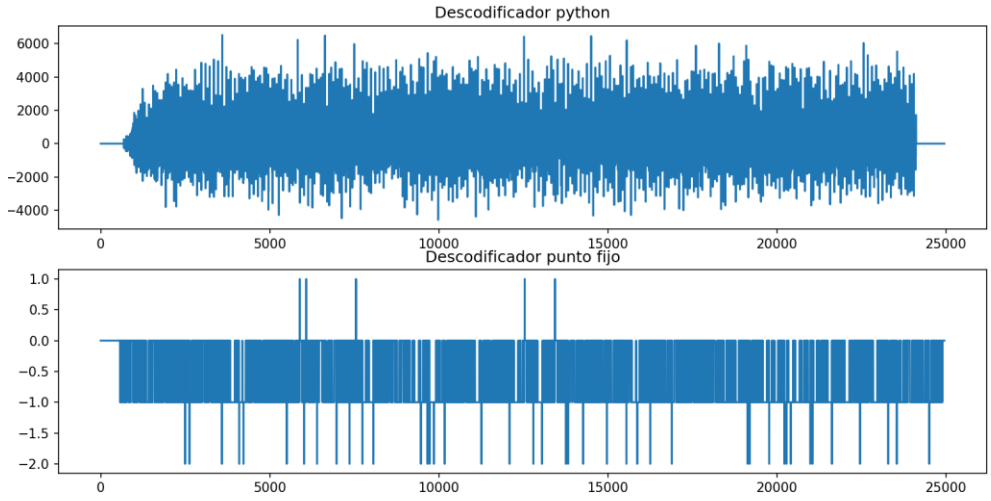

Figura 3. Resultados de descodificador implementado en Python vs codificador implementado con aritmética de punto fijo en C.

Las diferencias y errores entre los procesos codificación y decodificación se deben a la estrategia de cuantización elegida: las operaciones de bits y reducir valores de 16 bits a 8 bits máximo mediante una división entera provocarán perdida de datos que se ve reflejada durante el escalamiento en el decodificador. Además, el factor de escalamiento de la FFT con punto fijo ira reduciendo más precisión entre más etapas se añadan. También por errores de programación es posible perder bits durante los métodos de punto fijo y reconstrucción de las muestras.

Por otra parte, se midió el tiempo de ejecución del codificador y descodificador, los resultados se muestran en la tabla 1.

Tabla 1. Medición de tiempo de ejecución para la implementación de punto fijo de codificador y descodificador.

Nivel Opt.	Tiempo (s)					
	Encoder			Decoder		
O0	0.016412	0.016394	0.016348	0.017300	0.017363	0.017393
O1	0.003557	0.003551	0.003587	0.003567	0.003591	0.003645
O2	0.003353	0.002898	0.002933	0.003385	0.002929	0.002956
O3	0.002927	0.002923	0.002991	0.002967	0.002971	0.002993
Os	0.003320	0.003306	0.003297	0.003353	0.003365	0.003338

4. Neon

En esta etapa de optimizo en el código para su implementación en Neon. En la figura 4 se muestra los resultados para el codificador diseñado en Python y el implementado en Neon, igualmente en la figura 5 se muestran los resultados para el descodificador. Para el caso del codificador se obtuvo un error promedio absoluto de 14886.29 y 2470.82 para decodificador.

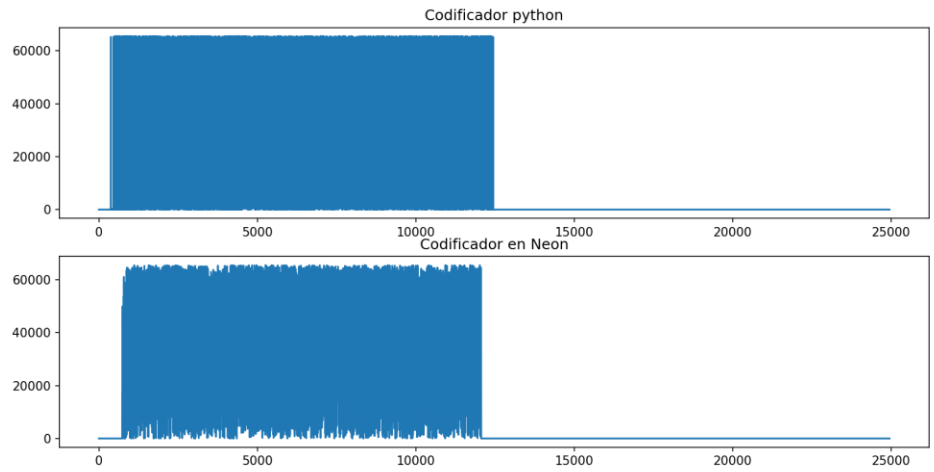

Figura 4. Resultados de codificador implementado en Python vs codificador implementado en Neon.

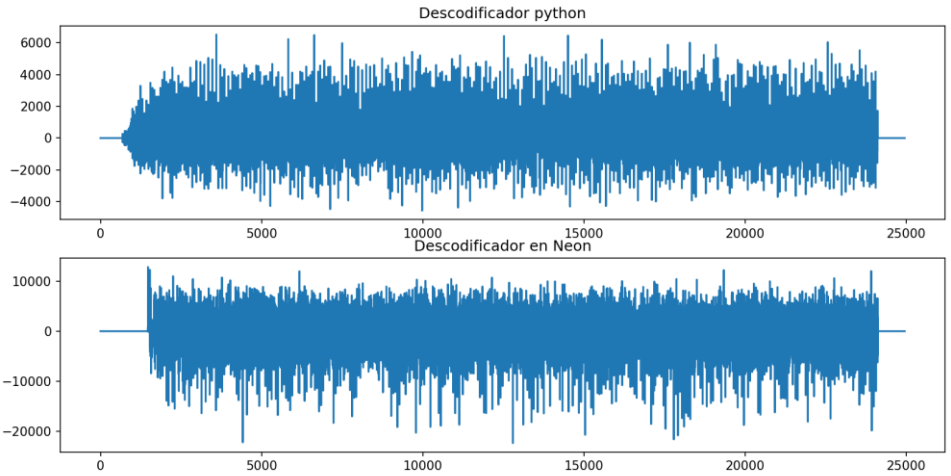

Figura 5. Resultados de decodificador implementado en Python vs codificador implementado Neon.

Como se puede ver la optimización usada en Neon permite obtener mejores resultados. Para este caso también se midió los valores de ejecución los cuales disminuyeron como se muestra en la tabla2.

Tabla 2. Medición de tiempo de ejecución para la implementación de punto fijo de codificador y descodificador.

Nivel Opt.	Tiempo (s)					
	Encoder			Decoder		
O0	0.005077	0.005034	0.005062	0.004857	0.004931	0.004883
O1	0.004011	0.004153	0.004078	0.003922	0.003955	0.003930
O2	0.004081	0.003940	0.004096	0.003905	0.004035	0.003893
O3	0.003970	0.003992	0.003988	0.003981	0.003925	0.003891
Os	0.003994	0.003979	0.003949	0.004000	0.003929	0.003868

5. Conclusiones

La aritmética de punto fijo representa una solución eficaz para realizar tareas computacionales exigentes en sistemas limitados y como una alternativa amigable con lenguajes de bajo nivel como C en directa comparación con los valores de punto flotante, los cuales, aunque precisos y ventajosos exigen mucho de los procesadores. Sin embargo, la efectividad y precisión de las operaciones matemáticas depende principalmente de bibliotecas robustas y aprovechamiento de las estructuras de datos múltiples que cada procesador pueda disponer. En el caso del cortex A72, las operaciones de punto fijo deben ser realizadas

por sobre la unidad aritmética del mismo como cualquier otro tipo de operación, pero otros sistemas como procesadores exclusivos para DSP pueden disponer de funciones intrínsecas que permitan aprovechar al máximo las unidades dedicadas del hardware.