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(x):
    N=len(x)
    #print(N)
    angulo1=0
    angulo2=0
    z=0
    z1=0
    delta=4*pi/N
    yr=[0]*N
    yi=[0]*N
    y=[0]*N
    for k in range(0,N//2,1):
        for i in range(N//2):
            angulo1=i*delta*k  # n*k*2*pi/(N/2)
            angulo2=angulo1+(delta/2)*k #n*k*2*pi/(N/2)  + 2*pi/N
            z=cos(angulo1)
            z1=cos(angulo2)
            z=z*x[2*i] #posición par
            z1=z1*x[2*i+1] #posición impar
            yr[k]=yr[k]+z+z1 #parte real transformada posición k
            yr[k+N//2]= yr[k+N//2]+z-z1 #parte real transformada posición k + N/2
            z=sin(-angulo1)
            z1=sin(-angulo2)
            z=z*x[2*i] #posición par
            z1=z1*x[2*i+1] #posición impar
```

```
        yi[k]=yi[k]+z+z1 #parte imaginaria transformada posición k
        yi[k+N//2]= yi[k+N//2]+z-z1 #parte imaginaria transformada posición k + N/2
    y[k]=yr[k]+1j*yi[k]
    y[k+N//2]=yr[k+N//2]+1j*yi[k+N//2]
    #print(k,i)
  return y
```

Para cuantizar se utilizo una cuantización de huffman. El cual estima la posibilidades de cada carácter de aparecer y crea un árbol de mapeo para representar cada carácter con numero entero.

```
def tohex(val, nbits):
  return hex((val) & (2**nbits -1))



def FFT_64(x):
  audio_len=len(x)
  samples_in=[0]*int(np.ceil(audio_len/64)*64)
  #output_vector=[0]*int(np.ceil(audio_len/64))
  #print(len(samples_in))
  samples_in[0:audio_len]=x
  #print(len(samples_in))
  #print(samples_in)
  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)
    #print(len(bloque_temp[0:32]))
    if bloque_index==0:
```

```
            fft_out=bloque_temp[0:32]
        else:
            fft_out=np.concatenate((fft_out,bloque_temp[0:32]))

    fft_out = fft_out/416
    return fft_out

def get_probabilities(content):
    total = len(content) + 1 # Agregamos uno por el caracter FINAL
    print("content",len(content))
    c = Counter(content)
    res = {}
    for char,count in c.items():
        res[char] = float(count)/total
    res['end'] = 1.0/total
    return res

def make_tree(probs):
    q = []
    for ch,pr in probs.items():
        # La fila de prioridad está ordenada por prioridad y PROFUNDIDAD
        heapq.heappush(q,(pr,0,ch))

    while len(q) > 1:
        e1 = heapq.heappop(q)
        e2 = heapq.heappop(q)
        nw_e = (e1[0]+e2[0],max(e1[1],e2[1])+1,[e1,e2])
        heapq.heappush(q,nw_e)
    return q[0]
```

```
def make_dictionary(tree):
    res = {}
    search_stack = []
    search_stack.append(tree+("",)) # El último elemento de la lista es el prefijo!
    while len(search_stack) > 0:
        elm = search_stack.pop()
        if type(elm[2]) == list:
            prefix = elm[-1]
            search_stack.append(elm[2][1]+(prefix+"0",))
            search_stack.append(elm[2][0]+(prefix+"1",))
            continue
        else:
            res[elm[2]] =str(int('1'.join(elm[-1]),2))
            #print(a)
        pass
    return res


def compress(dic,content):
    compressed=[]
    for ch in content:
        code = dic[ch]
        compressed.append(code)
        #print(ch,code)
    return compressed


def store(compressed,dic,outfile):
    # Lo guardamos en un archivo
    outf = open(outfile+".txt",'w')
```

```
    for element in compressed:
        outf.write(element + "\n")
    outf.close()
    # Guardamos el diccionario en otro archivo
    outf = open(outfile+".dic",'w')
    json.dump(dic,outf)
    outf.close()
    pass
```

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

2.2. Decodificador

El decodificador promero rescontruye la fft obtenida en el codificador, según el mapeo generado en el mismo.

```
def decode(val,dic):
    for key, value in dic.items():
        if int(val) == int(value):
            return key


def huffman_decode(codes,dic):
    fft=[0]*12448*2
    index=0
    complejo=[]
    for code in codes:
        char=decode(code,dic)
        if int(code)==427 or int(code)==7935:
            complejo=complex(''.join(complejo))
```

```
            fft[index]=complejo
            complejo=[]
            index=index+1
        else:
            complejo.append(char)
    return fft
```

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

    for bloque_index in range(0,fft_size,int(64/2)):
        bloque_temp=x[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=ifft_out*416
    return ifft_out

man=IFFT_64(fft)
```

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

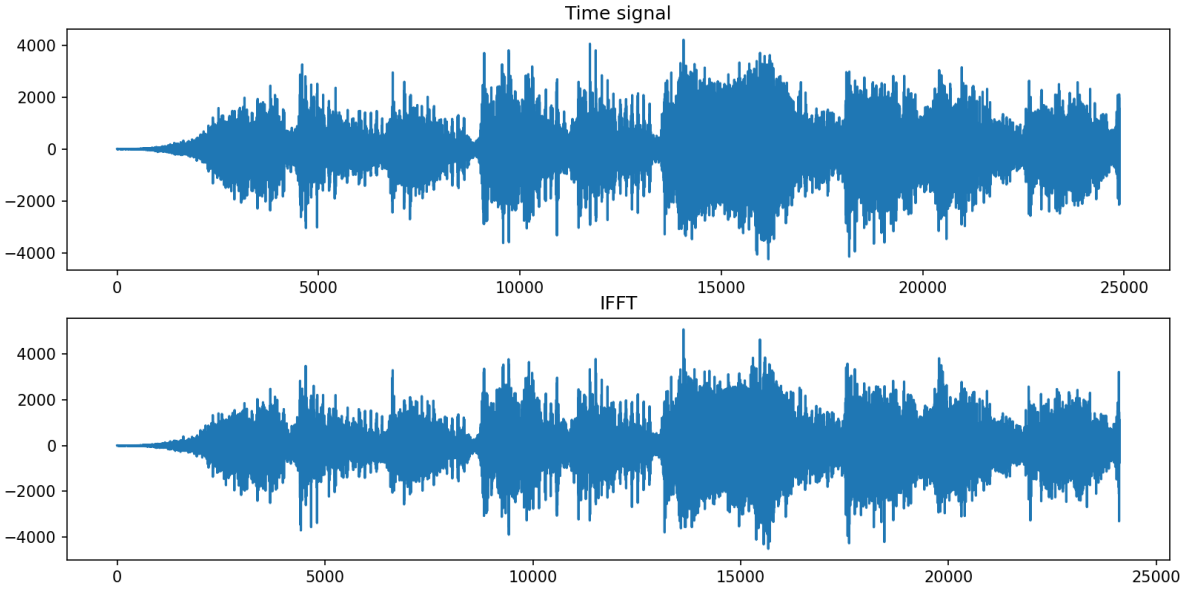

Figura 1. Señal de audio original y señal recuperada por el descodificador.

Como se puede ver en la figura 1 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. El error absoluto promedio fue 20. En la figura 2 se presenta la diferencia entre los vectores obtenidos.

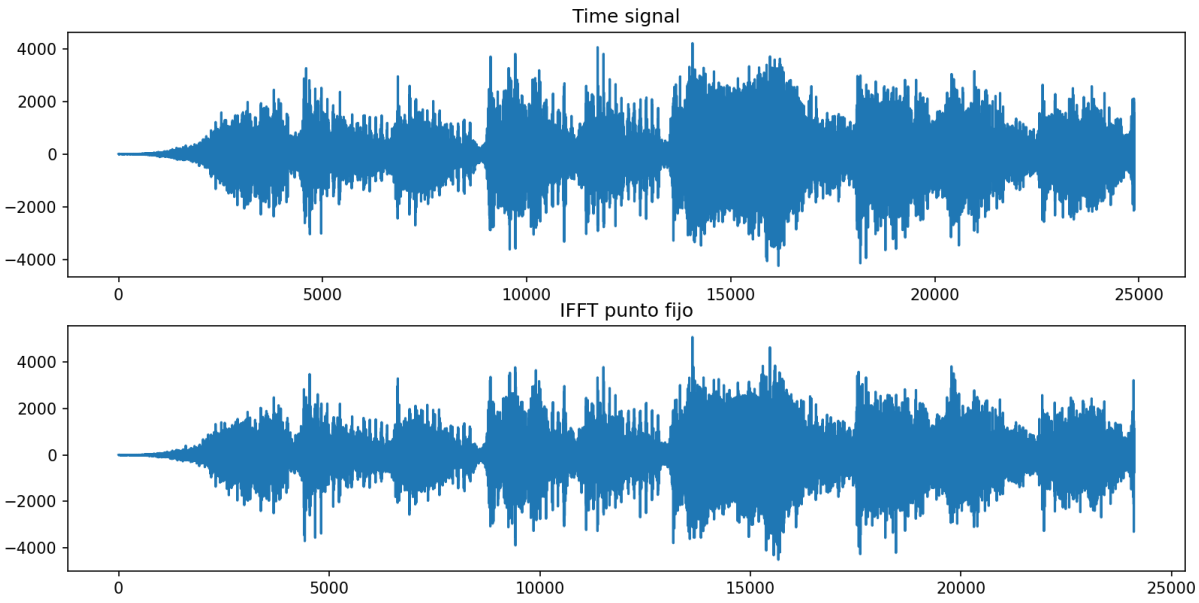

Figura 3. Señal de audio original vs señal obtenida implementadon con aritmética de punto fijo en C.

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	0.034663	0.034631	0.034613	0.035610	0.035586	0.035666
O1	0.010415	0.010485	0.010437	0.011089	0.010964	0.011000
O2	0.007477	0.007367	0.007383	0.007895	0.007944	0.007972
O3	0.007368	0.007378	0.007318	0.007789	0.007798	0.007376
Os	0.010889	0.010864	0.010932	0.011276	0.011324	0.011372

4. Neon

En esta etapa de optimizo en el código para su implementación en Neon. En la figura 3 se muestra los resultados para la implementación en Neon se muestran los resultados para el descodificador. Para obtuvo un error promedio absoluto de 15.7

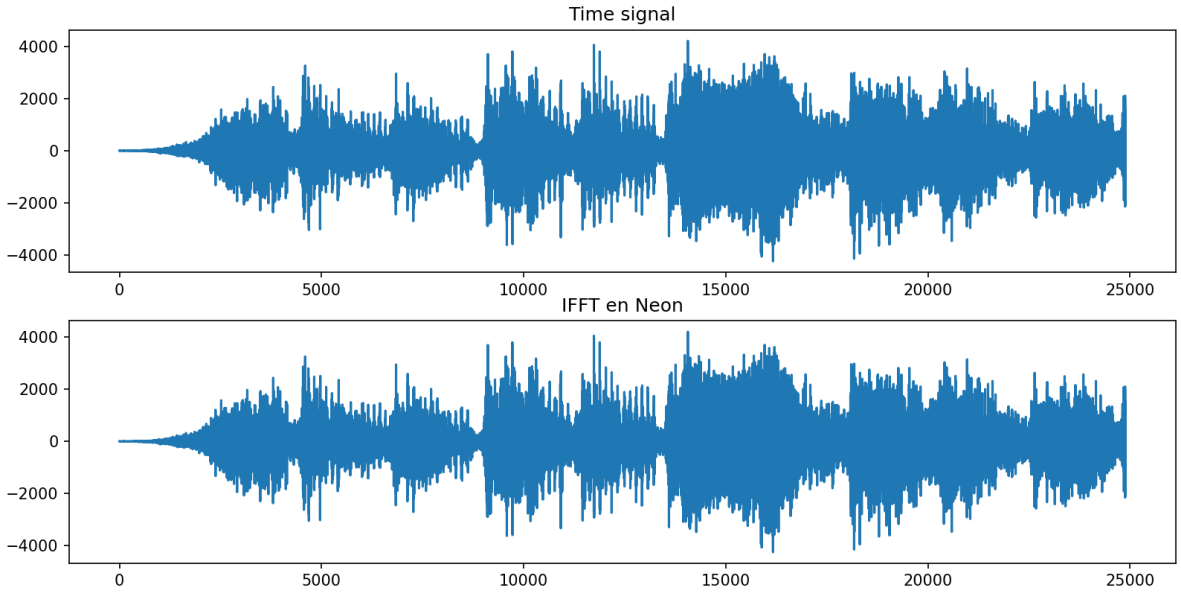

Figura 3. Señal de audio original vs señal obtenida implementando en Neon.

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	0.001235	0.001207	0.001139	0.001306	0.001333	0.001398
O1	0.000468	0.000438	0.000434	0.000465	0.000469	0.000493
O2	0.000437	0.000437	0.000459	0.000478	0.000465	0.000496
O3	0.000434	0.000434	0.000453	0.000467	0.000493	0.000464
Os	0.000562	0.000458	0.000451	0.000491	0.000513	0.000489

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.

6. Apéndice

En la carpeta llamada “Archivos de audio” se encuentran los archivos .wav , el archivo original llamado “test_audio.wav”, el los generados por Python, punto fijo y neon.