

Instituto Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Maestría en Electrónica

Curso MP-6157 Técnicas de Adquisición y Procesamiento de Datos

Proyecto 2: Implementation of an Audio Codec in an Embedded Device.

Brandon Varela

Steven Rojas

Andrés López Gómez

I Cuatrimestre 2022

1. Introducción.

En la actualidad, con la continua generación de nuevas aplicaciones de software con mayores niveles de desempeño, se ha aumentado constantemente la exigencia del hardware en cual se ejecutan tales aplicaciones, tratando de aprovechar al máximo la potencia y los recursos del sistema. Por tal motivo se han alcanzado sistemas embebidos cada vez más potentes con hardware más reducido.

Los sistemas embebidos son sistemas electrónicos encapsulados, donde se almacena en una sola tarjeta, todo el hardware y software necesario, para realizar un propósito o función específicas. Están orientados a resolver problemas en tiempo real, por lo que su sistema operativo va personalizado a este fin.

Para el siguiente informe se desarrolló un programa en lenguaje C, “Codificador – Decodificador de Audio”, el cual se utiliza para evaluar la velocidad de procesamiento de un sistema embebido personalizado, generado por medio de la herramienta Yocto, e implementado para una tarjeta de desarrollo Raspberry Pi 4 Model B.

Asimismo, se implementó una red LAN con IPs estáticas, para generar un entorno Linux por medio de una máquina virtual, que permite la comunicación con el sistema embebido Raspberry Pi 4 (Ethernet o UART) y se utiliza para la depuración y verificación de la funcionalidad del sistema. En la siguiente figura 1 se muestra un diagrama general del entorno de desarrollo.

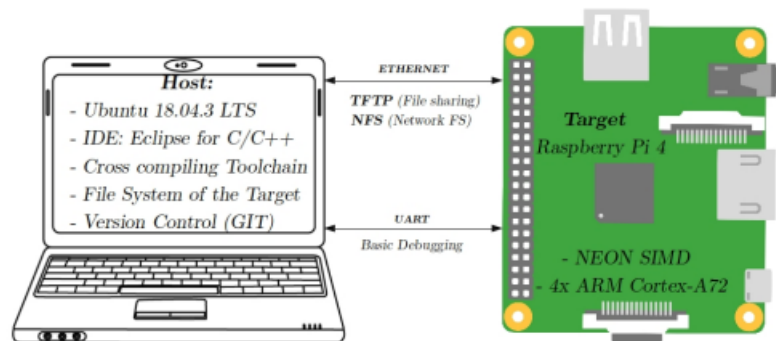


Figura 1. Diagrama general del entorno de desarrollo.

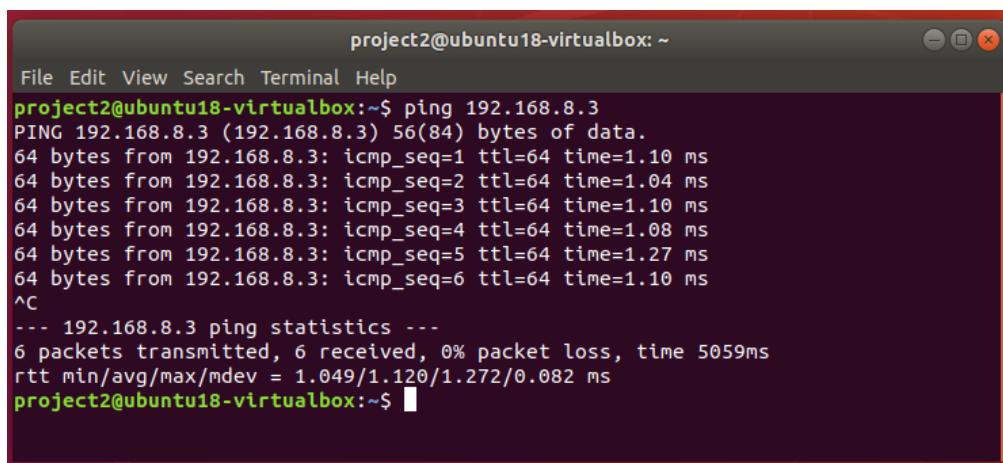
2. Setting up the Host

Por medio del instructivo y tiempo de investigación se implementa una red LAN con las IPs estáticas según la figura 2.

| Device | Static IP | Netmask | Gateway |
|-------------------------------|-------------|---------------|---------|
| PC (laptop) | 192.168.8.1 | 255.255.255.0 | - |
| Virtual Machine (Ubuntu host) | 192.168.8.2 | 255.255.255.0 | - |
| Target (RPI 4) | 192.168.8.3 | 255.255.255.0 | - |

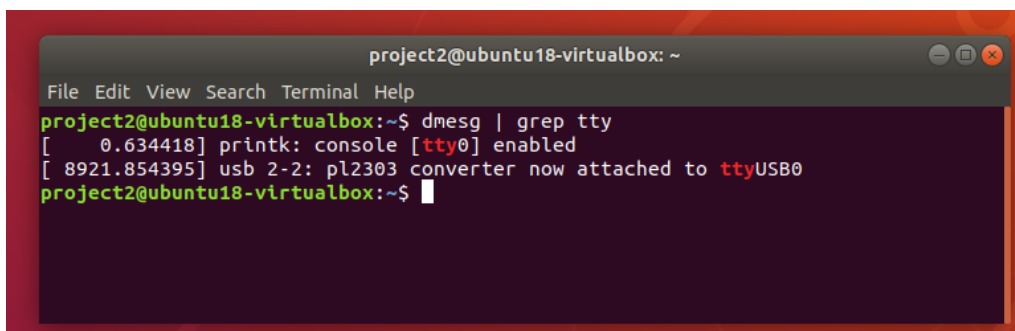
Figura 2. Asignación de IPs estáticas.

Se prueban las interfaces de comunicación Ethernet y UART entre la Raspberry Pi4 y la máquina virtual con Ubuntu, tal y como se muestra en la figura 3. De igual forma se verifica la correcta instalación de Eclipse IDE C/C++ Developers y la conexión TTL, tal y como se muestra en la figura 4.



```
project2@ubuntu18-virtualbox: ~  
File Edit View Search Terminal Help  
project2@ubuntu18-virtualbox:~$ ping 192.168.8.3  
PING 192.168.8.3 (192.168.8.3) 56(84) bytes of data.  
64 bytes from 192.168.8.3: icmp_seq=1 ttl=64 time=1.10 ms  
64 bytes from 192.168.8.3: icmp_seq=2 ttl=64 time=1.04 ms  
64 bytes from 192.168.8.3: icmp_seq=3 ttl=64 time=1.10 ms  
64 bytes from 192.168.8.3: icmp_seq=4 ttl=64 time=1.08 ms  
64 bytes from 192.168.8.3: icmp_seq=5 ttl=64 time=1.27 ms  
64 bytes from 192.168.8.3: icmp_seq=6 ttl=64 time=1.10 ms  
^C  
--- 192.168.8.3 ping statistics ---  
6 packets transmitted, 6 received, 0% packet loss, time 5059ms  
rtt min/avg/max/mdev = 1.049/1.120/1.272/0.082 ms  
project2@ubuntu18-virtualbox:~$
```

Figura 3. Comunicación Ethernet Máquina Virtual - RPi4.



```
project2@ubuntu18-virtualbox: ~  
File Edit View Search Terminal Help  
project2@ubuntu18-virtualbox:~$ dmesg | grep tty  
[ 0.634418] printk: console [tty0] enabled  
[ 8921.854395] usb 2-2: pl2303 converter now attached to ttyUSB0  
project2@ubuntu18-virtualbox:~$
```

Figura 4. Comunicación TTL.

3. Setting up our Embedded System and its Board Support Package

Para el desarrollo del sistema operativo encargado de ejecutar el codec de audio en la placa de desarrollo Raspberry Pi 4 se hace uso de Yocto Project, esta herramienta nos permite desarrollar sistemas operativos basados en Linux hechos a la medida. Para así aprovechar los recursos de nuestro hardware al máximo y contar únicamente con los paquetes necesarios para ejecutar nuestras aplicaciones.

El flujo de trabajo de Yocto consiste en la utilización de Bitbake, una herramienta de construcción de código para sistemas basados en diferentes arquitecturas, el proceso de compilación de código para sistemas de arquitecturas diferentes a la máquina de desarrollo se conoce como compilación cruzada. Bitbake utiliza archivos de configuración que permiten especificar y configurar diferentes parámetros al momento de la construcción, así como archivos fuente que contienen el código necesario para el desarrollo de nuestros programas. Los archivos fuente presentados como recetas, las cuales representan un paquete o aplicación capaz de ser compilada e instalada en nuestro sistema, además de esto, los archivos fuente se presentan como capas, las cuales son un conjunto de recetas que contienen diferentes paquetes que comparten características entre sí, o que contienen los paquetes necesarios para permitir el uso de una plataforma de hardware, mejor conocido como Board Support Package.

Bitbake utiliza como entrada todos los paquetes requeridos por el sistema para la ejecución, esto abarca los paquetes de configuración de hardware, paquetes de aplicaciones, archivos de configuración, entre otros. Además, como salida Bitbake permite la generación de dos artefactos, una imagen ejecutable por nuestro sistema y un kit de desarrollo de software, que contiene el toolchain necesario para el desarrollo y compilación cruzada de aplicaciones para nuestro sistema embebido. En la figura 5 se muestra el flujo de trabajo de Yocto y los diferentes componentes de su arquitectura.

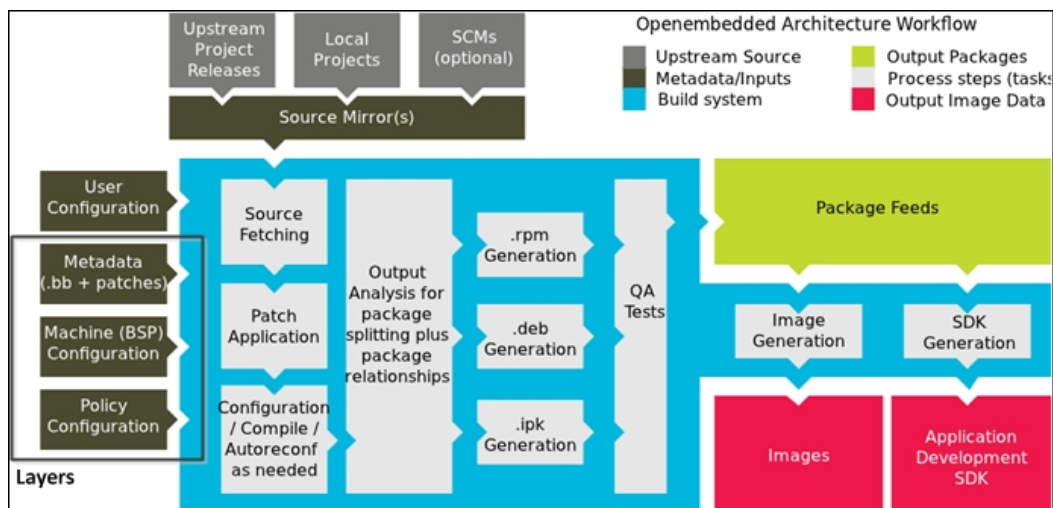


Figura 5. Flujo de trabajo de Yocto.

Para el desarrollo de nuestro sistema, se utilizaron diferentes paquetes como OpenSSH para ejecutar un servidor SSH en la Raspberry Pi y ser capaces de conectarnos de manera remota desde nuestro computador a través de la interfaz ethernet y el protocolo SSH y el layer meta-raspberrypi para configurar nuestro sistema y tener acceso a todos los componentes de hardware presentes en el mismo. También fue necesaria la generación de una imagen booteable por la raspberry pi y un kit de desarrollo de software para realizar la compilación cruzada de nuestra aplicación y realizar las pruebas en nuestro hardware de manera remota utilizando una sesión ssh.

4. The Application: Audio CODEC for an embedded Device

4.1. Encoder

Un audio codec consiste de 2 partes o componentes, los cuales son un codificador y un decodificador. La función principal del codificador es procesar las muestras de audio que van a tener X cantidad de bits y con estas producir un archivo de salida que utiliza una menor cantidad de bits. Esto se logra por medio de distintos algoritmos que pueden ser lossy (con pérdidas), lossless (sin pérdidas) o una combinación de ambos [1]. Sin importar el algoritmo elegido el objetivo primordial es conseguir un buen factor de compresión, pero sin perder mucha calidad de audio.

En la siguiente figura se muestra el diagrama de bloques que describe el funcionamiento del codificador diseñado en el presente proyecto.

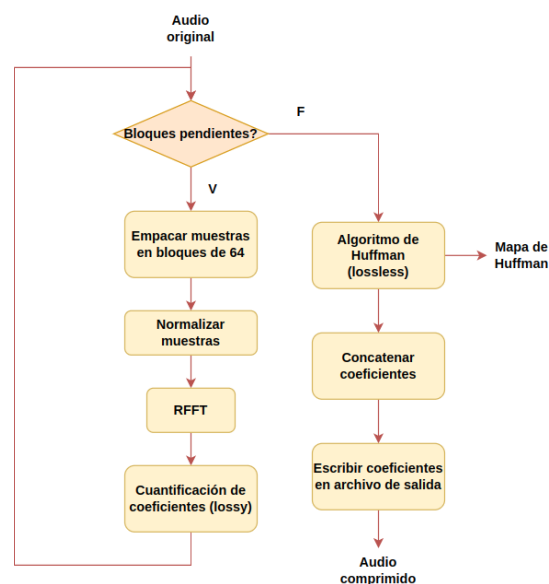


Figura 6. Diagrama de bloques del codificador.

Como se puede ver en la figura anterior el algoritmo del encoder diseñado utiliza algoritmos lossy y lossless para obtener mejores resultados como se verá en las secciones posteriores. Además, del archivo de audio comprimido también se genera un archivo con el árbol o mapa de huffman ya que este es necesitado por el decoder.

4.2. Decoder

Como se mencionó en la sección anterior el decoder es una parte fundamental del audio codec. Su función principal es realizar el proceso inverso del encoder, por esta razón este debe de saber exactamente qué hizo el encoder. Por lo tanto, el decoder va a recibir un archivo comprimido y con este va a generar el archivo de audio original, que como se mencionó anteriormente puede que hayan pérdidas en el proceso de compresión, con lo que este nuevo archivo creado tendría ciertas variaciones comparado al archivo original.

En la siguiente figura se muestra el diagrama de bloques que describe el funcionamiento del decodificador diseñado en el presente proyecto.

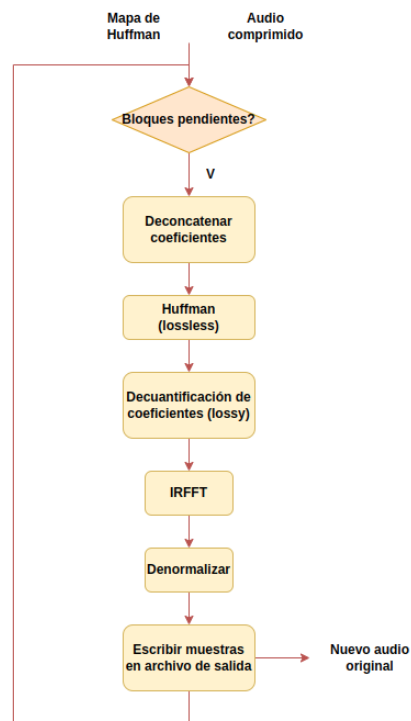


Figura 7. Diagrama de bloques del decodificador.

4.3. First Deliverable: Prototype and Test Vectors

Para la primera implementación del audio codec en punto flotante se decidió utilizar C como el lenguaje de programación. Esto se hizo pensando en poder reutilizar gran parte de este código cuando se fuera a realizar la implementación en punto fijo.

Para probar este algoritmo se tomó un audio de 33 segundos y utilizando Audacity se recortó a 9 segundos y se muestreó a 8 KHz utilizando esta misma herramienta.

Para comprobar el funcionamiento de esta implementación se tomó en cuenta el factor de compresión y el porcentaje de error de las muestras generadas al comprimir y descomprimir el audio en comparación con el audio original. Estos se pueden ver a continuación.

```
-rw-rw-r-- 1 bvarela bvarela 345097 may  1 23:50 decoder_output.txt  
-rw-rw-r-- 1 bvarela bvarela  65738 may  1 23:50 encoder_output.txt  
-rw-rw-r-- 1 bvarela bvarela 345229 may  1 23:49 samples.txt
```

Figura 8. Factor de compresión en punto flotante.

Como se puede ver en la figura anterior el tamaño del archivo original era 345229 y se comprime a 65738, con lo cual se tiene un factor de compresión de 5.25.

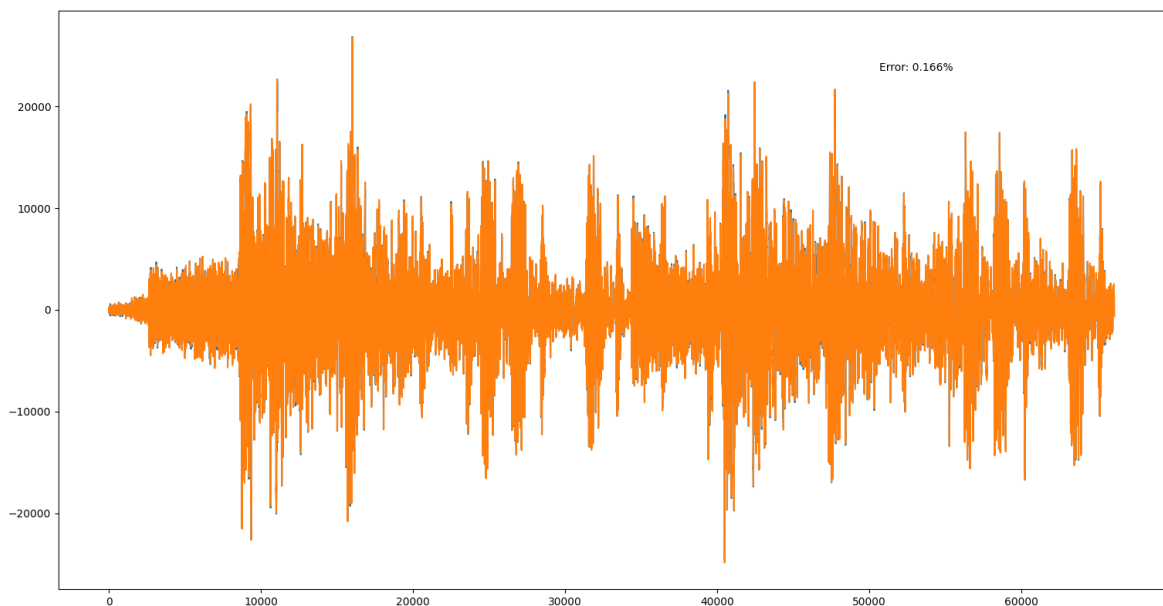


Figura 9. Audio original vs nuevo audio.

En la gráfica anterior se muestra en celeste el audio original y en anaranjado el nuevo audio después de comprimir y descomprimir. Como se puede ver hay pequeñas variaciones y el error promedio fue de 0.166% para la implementación en punto flotante.

4.4. Second Deliverable: Fixed-point Implementation

Los procesadores de señales digitales o DSP, son procesadores específicamente diseñados para procesar datos en tiempo real. Algunos DSP tienen unidades punto flotante ya que tiene la ventaja de ser más fácil y rápido de programar, sin embargo son menos eficientes en costo y potencia. Por tal motivo es importante diseñar o implementar algoritmos usando números enteros. La mayoría de los DSP operan en punto fijo.

En esta sección se evaluará la aplicación de la sección anterior con aritmética punto fijo, la cual usa un factor de escalamiento de 2^{14} , lo que implica 14 bits para la parte fraccionaria.

Al igual que el primer entregable se utilizó el mismo audio de 9 segundos y para comprobar el funcionamiento de esta implementación se tomó en cuenta el factor de compresión y el porcentaje de error de las muestras generadas al comprimir y descomprimir el audio en comparación con el audio original. Estos se pueden ver a continuación.

```
-rw-rw-r-- 1 bvarela bvarela 345255 may  2 18:16 decoder_output.txt  
-rw-rw-r-- 1 bvarela bvarela  64094 may  2 18:16 encoder_output.txt  
-rw-rw-r-- 1 bvarela bvarela 345229 abr 22 22:13 samples.txt
```

Figura 10. Factor de compresión en punto fijo.

Como se puede ver en la figura anterior el tamaño del archivo original era 345229 y se comprime a 64094, con lo cual se tiene un factor de compresión de 5.38. Si se compara este con el factor de compresión del entregable 1 se puede ver que es muy similar.

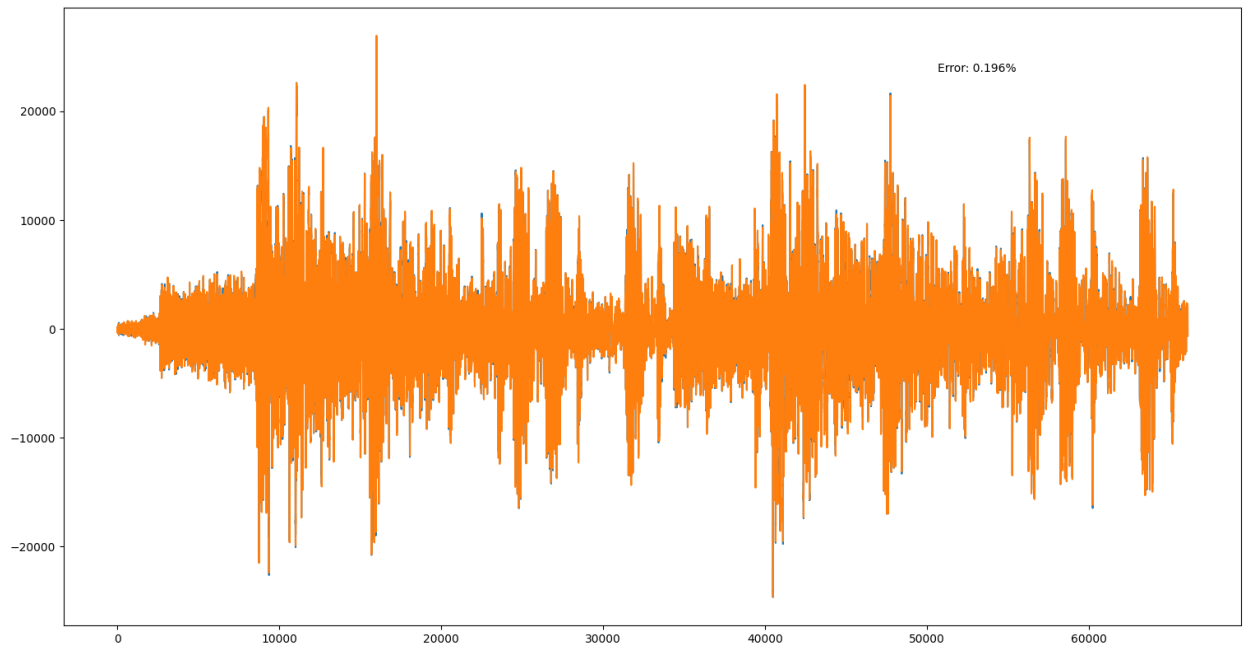


Figura 11. Audio original vs nuevo audio.

En la gráfica anterior se muestra en celeste el audio original y en anaranjado el nuevo audio después de comprimir y descomprimir. Como se puede ver hay pequeñas variaciones y el error promedio fue de 0.196% para la implementación en punto fijo. Este a su vez es muy similar al error que se obtuvo en la implementación en punto flotante.

En la tabla 1 y 2, se observan las muestras tomadas de los tiempos de ejecución para el Encoder y el Decoder de la aplicación en lenguaje C, respectivamente, dividiendo de igual forma el comportamiento para los diferentes niveles de optimización para el compilador de IDE Eclipse.

Tabla 1. Muestras del tiempo de ejecución Encoder.

| Encoder. Tiempo de ejecución con diferentes niveles de optimización [s] | | | | | |
|--|------------------|------------------|------------------|------------------|------------------|
| Muestra | -O0 | -O1 | -O2 | -O3 | -Os |
| 1 | 0,671904 | 0,655032 | 0,662297 | 0,663802 | 0,677107 |
| 2 | 0,670065 | 0,683784 | 0,695423 | 0,650768 | 0,694466 |
| 3 | 0,662851 | 0,65764 | 0,68527 | 0,675157 | 0,653121 |
| 4 | 0,655183 | 0,650425 | 0,642575 | 0,673715 | 0,656082 |
| 5 | 0,665327 | 0,673098 | 0,668125 | 0,652378 | 0,672813 |
| 6 | 0,668692 | 0,673917 | 0,666952 | 0,659815 | 0,676797 |
| 7 | 0,69048 | 0,673721 | 0,671095 | 0,654819 | 0,669763 |
| 8 | 0,671686 | 0,653593 | 0,674039 | 0,670966 | 0,662542 |
| 9 | 0,663407 | 0,677308 | 0,654435 | 0,663067 | 0,688947 |
| 10 | 0,663009 | 0,682329 | 0,68095 | 0,683615 | 0,69143 |
| Promedio | 0,6682604 | 0,6680847 | 0,6701161 | 0,6648102 | 0,6743068 |

Tabla 2. Muestras del tiempo de ejecución Decoder.

| Decoder. Tiempo de ejecución con diferentes niveles de optimización [s] | | | | | |
|--|------------------|------------------|-------------------|------------------|------------------|
| Muestra | -O0 | -O1 | -O2 | -O3 | -Os |
| 1 | 2,669431 | 2,734873 | 2,808799 | 2,750054 | 2,708661 |
| 2 | 2,78047 | 2,817548 | 2,780859 | 2,863084 | 2,659536 |
| 3 | 2,79586 | 2,835596 | 2,858349 | 2,687771 | 2,828232 |
| 4 | 2,650288 | 2,681332 | 2,84,0973 | 2,638534 | 2,848938 |
| 5 | 2,844635 | 2,737157 | 2,71,7831 | 2,739254 | 2,736247 |
| 6 | 2,657441 | 2,690559 | 2,586316 | 2,718284 | 2,921882 |
| 7 | 2,689696 | 2,642315 | 2,791975 | 2,730145 | 2,772247 |
| 8 | 2,855382 | 2,676518 | 2,782103 | 2,874975 | 2,765439 |
| 9 | 2,723569 | 2,789092 | 2,703973 | 2,816933 | 2,647534 |
| 10 | 2,718345 | 2,855247 | 2,732753 | 2,84525 | 2,807288 |
| Promedio | 2,7385117 | 2,7460237 | 2,75564088 | 2,7664284 | 2,7696004 |

4.4.1. Análisis y resultados: Tiempos Encoder y Decoder

Con los datos obtenidos, se procede a centralizar los valores máximos, mínimos y promedios para cada nivel de optimización del compilador. El resultado se muestra en la tabla 3 y 4.

Tabla 3. Tiempo promedio para ejecución del Encoder.

| Encoder | | | |
|------------------------------|-----------------------------------|-----------------------------------|--|
| Nivel de Optimización | Tiempo Ejecución mayor [s] | Tiempo Ejecucion menor [s] | Tiempo Promedio 10 muestras [s] |
| -O0 | 0,69048 | 0,662851 | 0,6682604 |
| -O1 | 0,683784 | 0,650425 | 0,6680847 |
| -O2 | 0,695423 | 0,642575 | 0,6701161 |
| -O3 | 0,683615 | 0,650768 | 0,6648102 |
| -Os | 0,694466 | 0,653121 | 0,6743068 |

Tabla 4. Tiempo promedio para ejecución del Decoder.

| Decoder | | | |
|------------------------------|-----------------------------------|-----------------------------------|--|
| Nivel de Optimización | Tiempo Ejecución mayor [s] | Tiempo Ejecucion menor [s] | Tiempo Promedio 10 muestras [s] |
| -O0 | 2,855382 | 2,650288 | 2,7385117 |
| -O1 | 2,855247 | 2,642315 | 2,7460237 |
| -O2 | 2,858349 | 2,586316 | 2,755640875 |
| -O3 | 2,874975 | 2,638534 | 2,7664284 |
| -Os | 2,921882 | 2,647534 | 2,7696004 |

Debido a la optimización y flujo de la aplicación, los tiempos de ejecución en los diferentes niveles de optimización son similares para el Encoder, sin embargo, se muestra una disminución del tiempo entre el nivel -O0 y el -O3, siendo el nivel de optimización -O3 el más óptimo para el compilador.

Asimismo, los tiempos de ejecución en los diferentes niveles de optimización son similares para el Decoder, pero en este caso mostrando un ligero incremento entre el nivel -O0 y el -Os. Al ser programas separados e independientes, los resultados predicen que el decodificador de la aplicación puede ser más rápido con el nivel de

optimización -O0, por la estructura y diseño de programación, así como las técnicas de optimización en lo interior del IDE Eclipse.

Lo anterior se puede apreciar de mejor forma en la figura 12 y 13, donde se muestra gráficamente los tiempos promedio de cada nivel de optimización, para el Encoder y Decoder respectivamente.

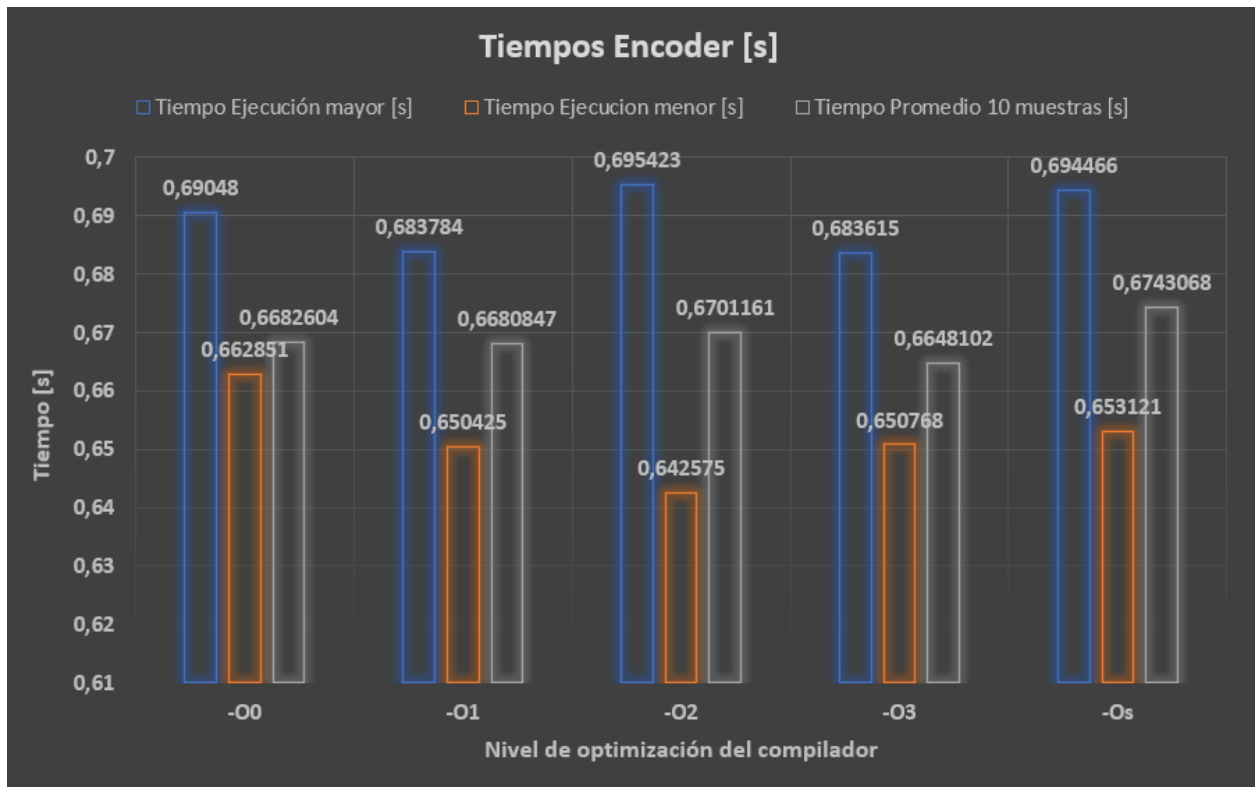


Figura 12. Tiempos de Encoder.

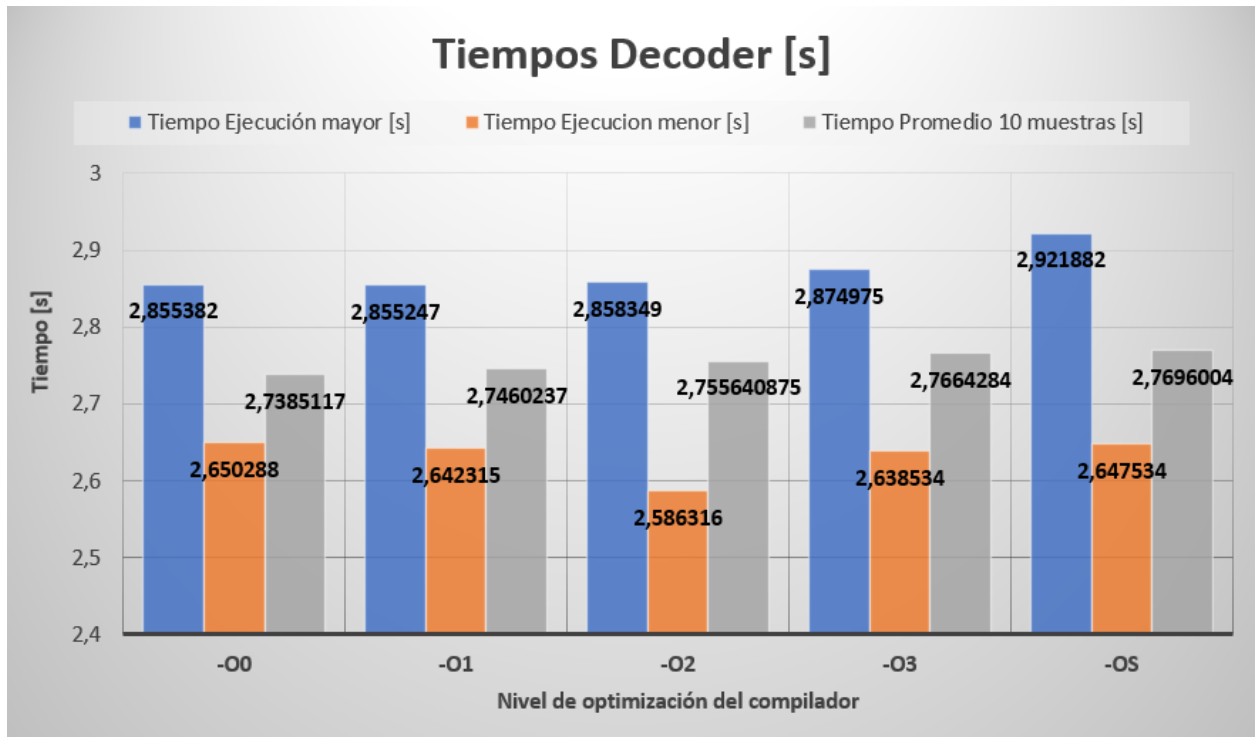


Figura 13. Tiempos de Encoder.

4.5. Third Deliverable: Optimized Implementation for ARM and NEON

Los DSP tienen la cualidad de tener interacción eficiente con periféricos, accesos rápidos a memoria, consta con modos de direccionamiento optimizados y poseen arquitecturas avanzadas con soporte de punto flotante y múltiples unidades funcionales (VLIW), los cuales pueden enviar múltiples instrucciones por ciclo. También cuentan con operaciones SIMD, las cuales son utilizadas para operar instrucciones en paralelo, en algunos casos muy utilizadas en algoritmos de estimación de movimiento para compresión de video o bien en algoritmos de compresión de audio, como el utilizado en la aplicación implementada, el cual se basa en un algoritmo de Hoffman.

A nivel de procesamiento de datos, es importante utilizar técnicas de optimización de software en el algoritmo. Algunas técnicas son Loop Unrolling, Software Pipelining y Caché Prefetching.

El dispositivo Raspberry Pi4 Model B, cuenta con un procesador Quad core Cortex-A72 (ARM v8) de 64 bits con un reloj de 1.5GHz. Los procesadores de la serie ARM Cortex-A pueden usar el procesador multimedia NEON para acelerar el programa. En la siguiente sección se comentarán y mostrarán los resultados al utilizar técnicas de optimización con NEON.

4.5.1. Análisis y resultados: Tiempos Encoder y Decoder por bloque

ARM NEON se introduce con el fin de mejorar la codificación y decodificación multimedia, la interfaz de usuario, los gráficos y las funciones relacionadas con los juegos que se ejecutan en dispositivos móviles o inteligentes. Se utiliza para acelerar los algoritmos y capacidades de procesamiento de señales.

Para efectos de la implementación de la aplicación de Codificación y Decodificación de audio de este proyecto, la diferencia entre tiempos de ejecución con NEON y sin NEON fué despreciable. Sin embargo se utiliza el *“Intrinsic vaddq_s64”* con el cual se optimiza la aplicación.

Se registran los tiempos de ejecución por bloque del Encoder y Decoder a la hora procesar los datos, sin embargo se descartan tiempos mínimos como la concatenación a la hora de procesar los datos. Los resultados se observan en la tabla 5 y 6. De forma similar, se arrojan los datos para cada nivel de optimización del compilador.

Tabla 5. Tiempos de ejecución por bloque del Encoder.

| Encoder por bloque | | | |
|-----------------------|----------------------------|----------------------------|---------------------|
| Nivel de Optimización | Tiempo Ejecución mayor [s] | Tiempo Ejecucion menor [s] | Tiempo Promedio [s] |
| -O0 | 0,000179 | 0,000077 | 0,000087 |
| -O1 | 0,000173 | 0,000077 | 0,000087 |
| -O2 | 0,000182 | 0,000077 | 0,000087 |
| -O3 | 0,000177 | 0,000078 | 0,000088 |
| -Os | 0,00018 | 0,000083 | 0,000093 |

Tabla 6. Tiempos de ejecución por bloque del Decoder.

| Decoder por bloque | | | |
|------------------------------|-----------------------------------|-----------------------------------|----------------------------|
| Nivel de Optimización | Tiempo Ejecución mayor [s] | Tiempo Ejecucion menor [s] | Tiempo Promedio [s] |
| -O0 | 0,004695 | 0,001549 | 0,002658 |
| -O1 | 0,005458 | 0,001605 | 0,002729 |
| -O2 | 0,005471 | 0,00161 | 0,002739 |
| -O3 | 0,005757 | 0,001515 | 0,002616 |
| -Os | 0,006151 | 0,001615 | 0,002792 |

Para el caso del Encoder, se muestra una uniformidad de tiempos para los diferentes niveles de optimización y se debe nuevamente por el diseño y optimización de los algoritmos utilizados en la aplicación. El tiempo por bloque del Encoder hace referencia a un promedio de 0.000087s.

En el caso del Decoder, de igual forma se muestra poca variación, sin embargo se aprecia que el valor con menor tiempo y el menor promedio para el tiempo de ejecución por bloque es por parte del nivel de optimización -O3, con un tiempo promedio de 0.002616s.

Para visualizar de mejor forma los datos, en la figura 14 y 15 se muestran las gráficas de los datos obtenidos.

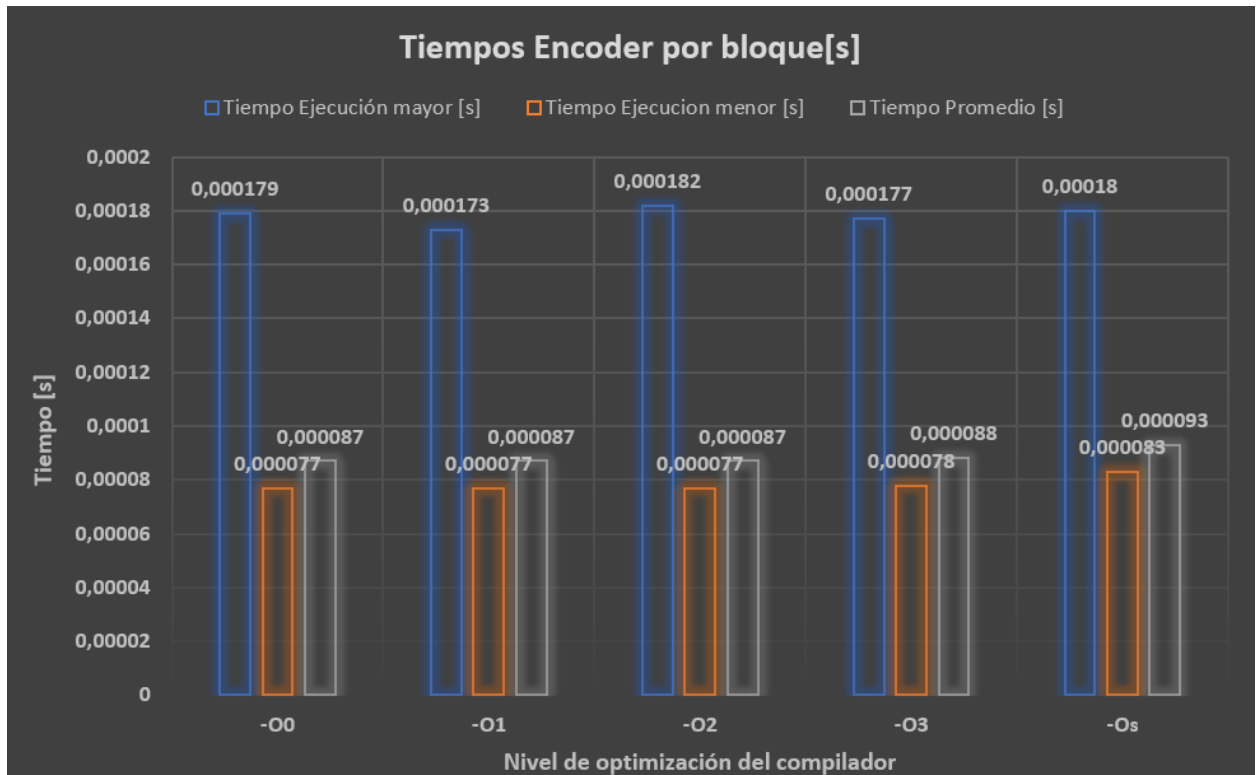


Figura 14. Tiempos por bloque del Encoder.

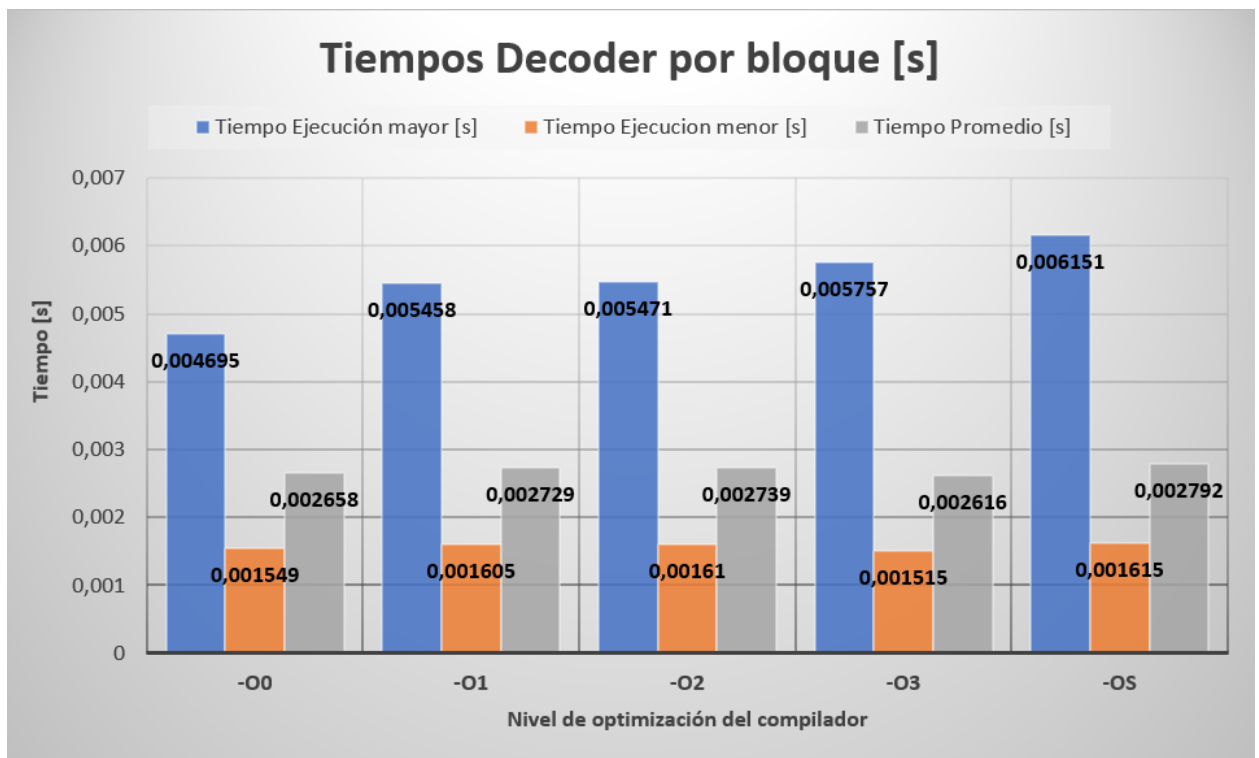


Figura 15. Tiempos por bloque del Decoder.

5. Conclusiones.

- 5.1. La implementación de un algoritmo que utiliza compresión de datos lossy y lossless ayuda en gran medida para aumentar el factor de compresión y a su vez mantener una calidad de audio buena.
- 5.2. Para la implementación con punto fijo, el Encoder tiene un mejor rendimiento con el nivel de optimización -O3 del compilador.
- 5.3. Los DSP programados en punto fijo son más eficientes en costo y potencia.
- 5.4. Los procesadores de la serie ARM Cortex-A pueden usar el procesador multimedia NEON para acelerar el programa.
- 5.5. Con ARM NEON el decodificador se muestra más óptimo para el nivel de optimización -O3, con un tiempo promedio de 0.002616 segundos.
- 5.6. El uso de “intrinsics” permite optimizar el uso de las características de cada arquitectura.
- 5.7. Las técnicas de optimización de software ayudan a mejorar el paralelismo a nivel de instrucción (ILP).
- 5.8. El error promedio fue de 0.166% para la implementación en punto flotante y 0.196% para la implementación en punto fijo.

6. Recomendaciones

- Utilizar el mismo archivo de salida del encoder “encoder_output.txt” para guardar el mapa o árbol de Huffman.
- Mejorar el algoritmo de Huffman para utilizar un solo bit para representar el valor más repetido, en lugar de 2 como lo hace ahora.
- Utilizar más intrinsics en la implementación para NEON.
- Cambiar algunas funciones para que las optimizaciones del compilador tengan mejores resultados.
- Utilizar archivos .wav como entrada del encoder, en lugar de archivos de texto.

- Generar archivos .wav en el decoder en lugar de archivos de texto.

7. Bibliografía

[1] International Journal of Science and Research (IJSR), & Tewari, P. (2017, febrero). Audio Compression Using Fourier Transform. <https://www.ijsr.net/archive/v6i2/ART2017951.pdf>

[2] Kuo, Sen M., Bob H. Lee & Wenshun Tian. Real-Time Digital Signal Processing: Implementation and Applications. Second Edition, John Wiley & Sons, 2007.

[3] Kumar, Vipin, et al. Introduction to Parallel Computing: Design and Analysis of Algorithms. Second Edition, Addison Wesley, 2003.