



Instituto Tecnológico de Costa Rica

Técnicas de Adquisición y Procesamiento de Datos

Proyecto II

Profesor: Ing. Eduardo Interiano Salguero

Alumnos

Santiago López Rojas

Erick Salas Chaverri

Ronny Zárate Ferreto

I Cuatrimestre 2020

Resumen

El presente es el proyecto final del curso de Adquisición y procesamiento de datos. En el mismo se implementa un codificador de audio basado en el principio de cuantización de coeficientes en el dominio de la frecuencia con FFT, similar al principio de MP3 pero como una prueba de concepto.

La implementación se realizó en un ordenador y en un sistema embebido (se selecciona la Raspberry Pi 4 como plataforma de desarrollo), para este último se le brindó a cada grupo de trabajo una guía para establecer el entorno de desarrollo con un *boot* personalizado. Para los cálculos de la FFT se emplea una aproximación en punto flotante y otra en punto fijo obteniendo resultados muy similares con errores MCE de alrededor de 2.1×10^{-4} . Finalmente se empleó la unidad NEON Single Instruction Multiple Data (SIMD) de la Raspberry Pi con el fin de optimizar la implementación.

Introducción

El documento consta de 3 secciones. La primera sección es una introducción sobre el proceso de *boot* en los sistemas embebidos y la organización del documento. Seguidamente se tiene la sección de desarrollo, divididas en 4 sub-secciones:

- Establecimiento del entorno de trabajo y proceso de *boot* en el *Target*
- Codificador y Decodificador de audio con FFT usando Python y C en *Host*
- Codificador y Decodificador de audio con FFT usando C en *Target* (Fixed-point)
- Optimización de Codificador y Decodificador de audio con FFT usando NEON en *Target*

En cada una de estas se indica el procedimiento realizado, resultados obtenidos y análisis de los mismos. Además, se hace mención de las mayores dificultades que se encontraron cada apartado y la forma en que fueron superadas.

El codificador/decodificador de los últimos 3 puntos de la sección de desarrollo se prueba utilizando archivos de audio con tonos, cuyo *bitdepth* es de 16 bits y poseen una frecuencia de muestreo de 8 kHz. Estos archivos son generados con la herramienta *Audacity* y presentan una duración máxima de 10s.

Como último apartado del documento se tienen las conclusiones donde se discute sobre los resultados obtenidos en la sección de desarrollo y se dan sugerencias para mejorar el desempeño y las pruebas para comprobar el funcionamiento.

Para el desarrollo de la primera etapa del proyecto se basa en el instructivo brindado, a partir del cual se establece el entorno de trabajo, para ello se hace uso de una máquina virtual con Ubuntu 18.04, el cual es referido como **Host**. Como siguiente paso se desarrolla una imagen personalizada de Linux, partiendo desde el *boot*, la compilación del Kernel y la creación del *File System* para el **Target**, es decir la Raspberry Pi 4.

De modo que el desarrollo del proceso de *boot* se realiza siguiendo la guía brindada, este entorno se desarrolla utilizando el software *Oracle VM Virtual Machine*. El establecimiento del proceso de *boot* en el *Target* sigue el flujo presentado en la figura 1. A partir de esto se explicará el proceso utilizado para el sistema embebido.

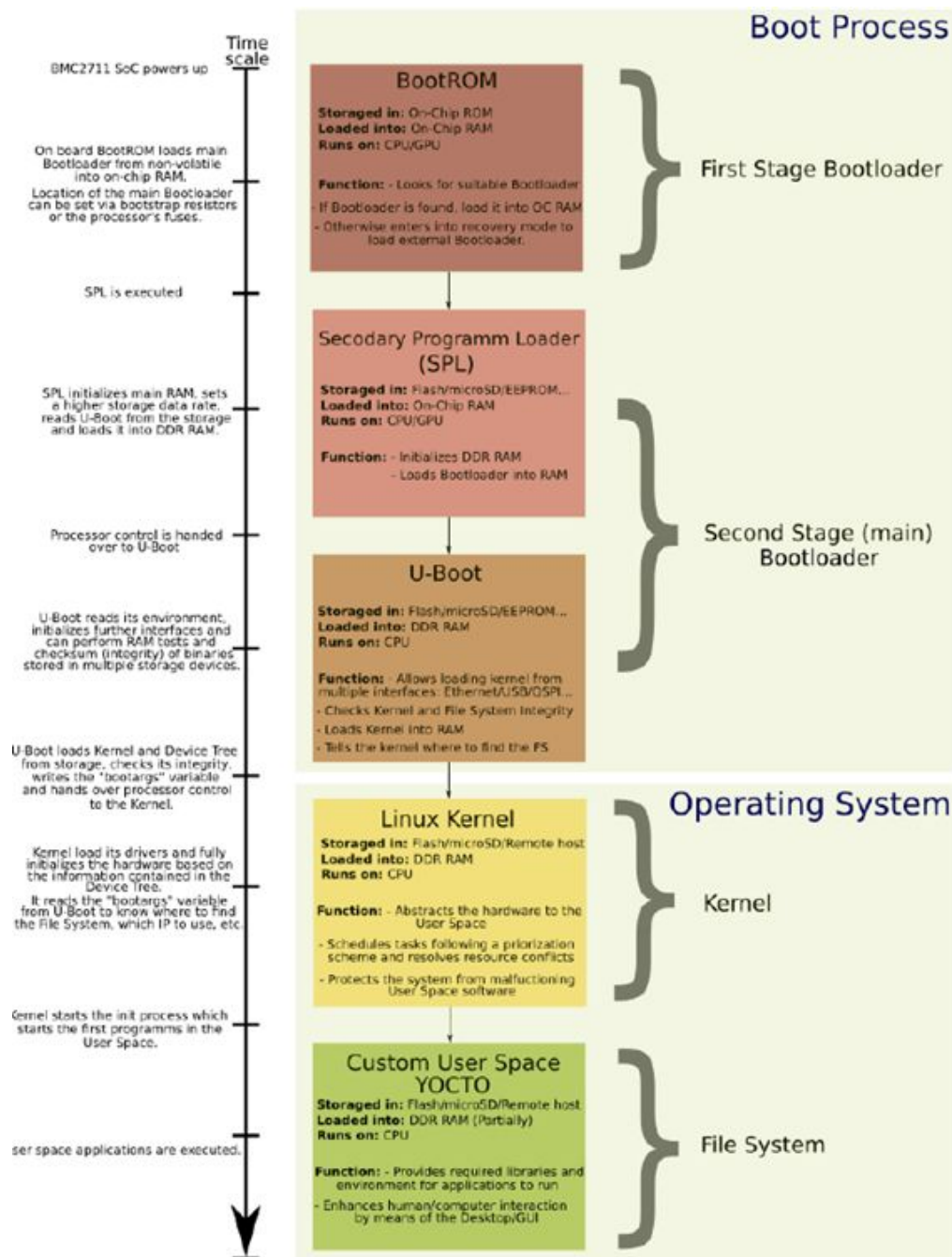


Figura 1. Flujo de un proceso genérico de *boot* en un sistema embebido

La figura 1 explica el proceso de *boot* general para cualquier sistema embebido, el cual consta de cuatro secciones:

1. Primera etapa de la carga del boot (*Bootloader*)

BootROM: Se almacena en **On-Chip ROM** y en este proceso se carga a **On-Chip RAM** y se ejecuta en los CPU/GPU. Esta etapa funciona de la siguiente manera:

- Se busca por un *bootloader* aceptables para ser cargados, cuando este se logra detectar se carga en el **On-Chip RAM**, si no se logra encontrar en el **On-Chip ROM** el sistema pasa en modo de recuperación y busca y carga *bootloader* externos.

2. Segunda etapa de la carga del boot/main (*Bootloader*)

Secondary Program Loader (SPL): Se almacena en la memoria **Flash/microSD/EEPROM** y en este proceso se carga a **DDR RAM** y se ejecuta en los CPU/GPU. Esta etapa funciona de la siguiente manera:

- Inicializa el *main* DDR RAM y carga el *U-Boot* en la DDR RAM, para permitir realizar el siguiente paso (el proceso de control es manejado sobre el *U-Boot*).

U-Boot: Se almacena en la memoria **Flash/microSD/EEPROM** y en este proceso se carga a **DDR RAM** y se ejecuta en los CPU. Esta etapa funciona de la siguiente manera:

- Permite la lectura del **kernel** de múltiples interfaces como por ejemplo Ethernet, USB o QSPI. Revisa la correcta integridad del **kernel** y el **File system**. Carga el kernel en la RAM y le dice al kernel donde se encuentran los **File system**. Luego de realizar esto, se escriben el *bootargs* y se le da el control al **kernel**.

3. Tercera etapa kernel (*Operating System*)

Linux Kernel: Se almacena en la memoria **Flash/microSD/Remote host** y en este proceso se carga a **DDR RAM** y se ejecuta en los CPU. Esta etapa funciona de la siguiente manera:

- Abstrae el hardware hacia el espacio software de usuario, luego organiza las tareas siguiendo un esquema de prioridades y resuelve los conflictos de dependencia de recursos. Evita los malos funcionamientos en el espacio software de usuario.

4. Cuarta etapa File System

Espacio de usuario personalizado YOCTO: Se almacena en la memoria **Flash/microSD/Remote host** y en este proceso se carga a **DDR RAM (parcialmente)** y se ejecuta en los CPU. Esta etapa funciona de la siguiente manera:

- Provee las bibliotecas requeridas y corre las aplicaciones del entorno. Establece las interacciones humano/computadora, en otras palabras, el Escritorio/GUI.

Desarrollo

Establecimiento del entorno de trabajo y proceso de *boot* en el *Target*

Video de implementación :

En la sección anterior se explica cómo se realiza el proceso de boot para un sistema embebido en general; no obstante, existen ciertas limitación en lo que respecta al *Target* que estamos empleando. La primera de estas limitaciones, siendo que no es una plataforma open-source, se presenta la falta de información o manual de referencia que describa la estructura interna del funcionamiento del SoC Broadcom BVM2711 (utilizado en la Raspberry Pi 4).

Esto hace imposible la creación de un *Bootloader* que interactúe con el procesador en las primeras etapas de boot, cuando sus unidades internas no han sido inicializadas.

De este modo el Broadcom BVM2711 tiene originalmente el siguiente proceso de boot:

1. Primera etapa del *Bootloader*

- Siguiendo el mismo proceso que se indica anteriormente, la primera etapa del BootROM se encuentra en el on-chip ROM del procesador y es ejecutado en la GPU, mientras el nucleos ARM se encuentran en etapa de reinicio.
- Luego el BootROM lee el *Bootloader main* de la EEPROM

2. Segunda etapa del *Bootloader*

- *Bootloader* revisa el orden predeterminado de boot que posee, de esta manera revisa en orden cual boot se debe emplear (microSD, Network o USB).
- Luego de mirar en la interfaz seleccionada por el archivo *start.elf* binario y lo carga.
- Finalmente el archivo *start.elf* mira dentro del archivo *config.txt* por información para continuar con el proceso de boot, como por ejemplo, donde encontrar el *Kernel*.

Como segunda limitante se tiene que el *BootROM*, el *bootloader* y el *start.elf* todos son programas de propietario, por lo que no se tiene acceso al código fuente, solo el código binario. Esto se hace con el fin de hacer imposible realizar modificaciones sobre estos. Como tercera limitación tenemos que un *network boot* no existe a la fecha para nuestro *Target*, solo versiones de prueba, siendo poco recomendable el usar estas por la posible aparición de *bugs*.

Entonces, para superar estas 3 limitaciones y lograr un *network boot* estable entre *Host* y *Target*, se seguirán los siguientes pasos:

1. Se usarán los primeros dos pasos del proceso de boot de la RPI 4 como si fueran la primera etapa del nuevo proceso de boot. Esto se consigue modificando el archivo config.txt para que se cargue directamente el U-Boot del segundo paso de boot de la RPI 4.
2. Como en el primer paso del boot se inicializa el LPDDR4 RAM no será necesario usar el SPL, por lo que se omite este paso.

3. Una vez que el U-Boot esta corriendo, se tiene ya toda la flexibilidad de un *bootloader* modificable para lograr *network boot*.

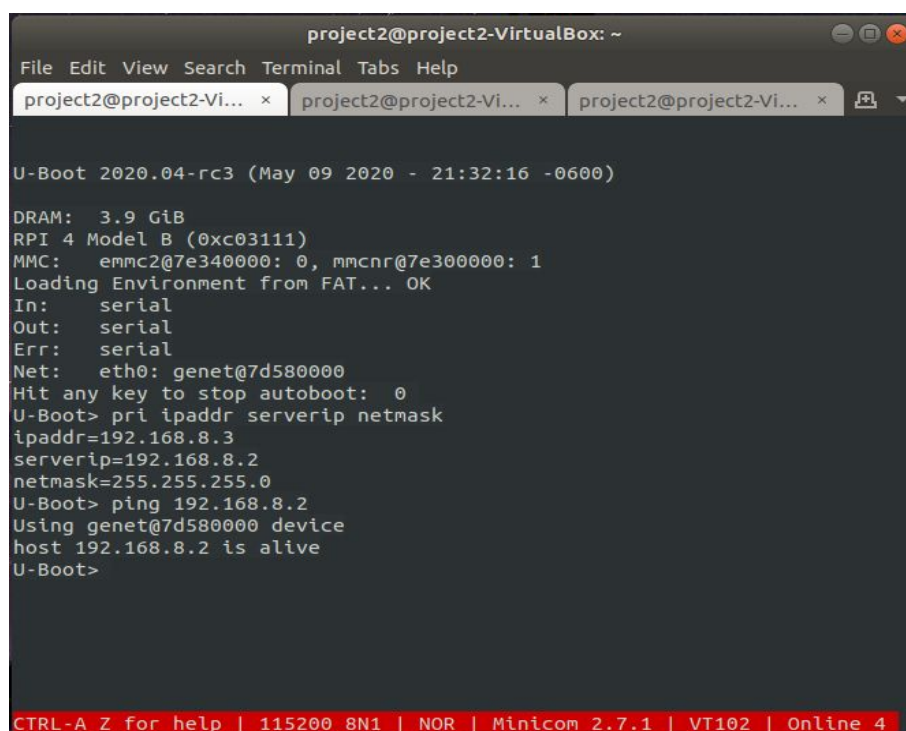
Primera etapa del *bootloader*: BootRom

Como se mencionó, las primeras instrucciones que se ejecutan en el SoC después del encendido son las instrucciones localizadas en el *BootROM*. Dado que este es normalmente un software propietario se utilizó el BootROM brindado por Broadcom.

Se usa el *Das U-Boot* que es un *bootloader* universal que usan muchos sistemas embebidos. Este es configurado y seteado en el entorno Linux Ubuntu 18.04 como se indica en [1], de igual manera este es configurado directamente en el *Host* y además se establece una comunicación Ethernet entre el *Host* y la *Target*.

Siguiendo las instrucciones del instructivo, como primer paso se hizo la Cross-Compilación del Toolchain para la arquitectura del ARM 64-bit utilizando el Host que es un sistema basado en un compilador x86 de 64-bits. Cuando el Cross-compilador estuvo listo se lleva a cabo la Cross-Compilación de U-Boot, con ello se tiene el soporte para el proceso de *Boot* del procesador Broadcom BCM2711, y los *drivers* que permiten la inicialización y uso de los periféricos de la Raspberry Pi 4.

Se configuró el *Target* para que cargue el U-Boot compilado, para ello se modifica el archivo *config.txt* para que cargue el archivo binario recién compilado, y haciendo uso de la comunicación serial mediante el protocolo UART se puede observar que el proceso fue exitoso, tal como se muestra en la figura 2.

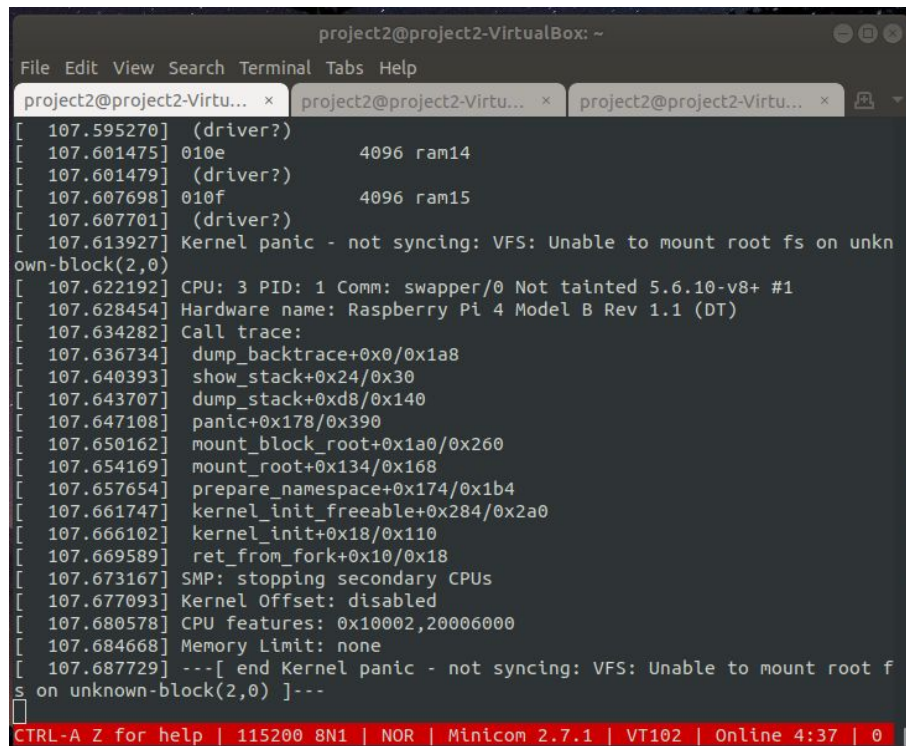


```
project2@project2-VirtualBox: ~  
File Edit View Search Terminal Tabs Help  
project2@project2-Vi... x project2@project2-Vi... x project2@project2-Vi... x  
U-Boot 2020.04-rc3 (May 09 2020 - 21:32:16 -0600)  
DRAM: 3.9 GiB  
RPI 4 Model B (0xc03111)  
MMC: emmc2@7e340000: 0, mmcnr@7e300000: 1  
Loading Environment from FAT... OK  
In: serial  
Out: serial  
Err: serial  
Net: eth0: genet@7d580000  
Hit any key to stop autoboot: 0  
U-Boot> pri ipaddr serverip netmask  
ipaddr=192.168.8.3  
serverip=192.168.8.2  
netmask=255.255.255.0  
U-Boot> ping 192.168.8.2  
Using genet@7d580000 device  
host 192.168.8.2 is alive  
U-Boot>  
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Online 4
```

Figura 2. Etapa de U-Boot en la Raspberry Pi 4.

El siguiente paso es la compilación del Kernel de Linux, el kernel provee una capa de abstracción entre el hardware (drivers) y las aplicaciones del *User-Space*. Para llevar a cabo eso se clona el repositorio de Github: <https://github.com/torvalds/linux.git>. Se configura el Kernel con las definiciones por defecto para el procesador Broadcom BCM2711 y la Raspberry Pi 4.

Una vez compilado el Kernel se configura para cargarlo mediante TFTP, y se realiza el proceso de *Boot* hasta el mensaje de aviso por la ausencia del *File System* esto se muestra en la figura 3.



```
project2@project2-VirtualBox: ~
File Edit View Search Terminal Tabs Help
project2@project2-Virtu... x project2@project2-Virtu... x project2@project2-Virtu... x
[ 107.595270] (driver?)
[ 107.601475] 010e          4096 ram14
[ 107.601479] (driver?)
[ 107.607698] 010f          4096 ram15
[ 107.607701] (driver?)
[ 107.613927] Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(2,0)
[ 107.622192] CPU: 3 PID: 1 Comm: swapper/0 Not tainted 5.6.10-v8+ #1
[ 107.628454] Hardware name: Raspberry Pi 4 Model B Rev 1.1 (DT)
[ 107.634282] Call trace:
[ 107.636734] dump_backtrace+0x0/0x1a8
[ 107.640393] show_stack+0x24/0x30
[ 107.643707] dump_stack+0xd8/0x140
[ 107.647108] panic+0x178/0x390
[ 107.650162] mount_block_root+0x1a0/0x260
[ 107.654169] mount_root+0x134/0x168
[ 107.657654] prepare_namespace+0x174/0x1b4
[ 107.661747] kernel_init_freeable+0x284/0x2a0
[ 107.666102] kernel_init+0x18/0x110
[ 107.669589] ret_from_fork+0x10/0x18
[ 107.673167] SMP: stopping secondary CPUs
[ 107.677093] Kernel Offset: disabled
[ 107.680578] CPU features: 0x10002,20006000
[ 107.684668] Memory Limit: none
[ 107.687729] ---[ end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(2,0) ]---
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Online 4:37 | 0
```

Figura 3. Proceso de *Boot* hasta la búsqueda del *File System*.

Finalmente se emplea Yocto para generar el *File system*, este de la misma manera se descarga, configura y compila directamente en el *Host*, y es copiado a un servidor NFS para comunicarlo con el *Target*. En la figura 4 se muestra el boot exitoso hasta la solicitud de credenciales para el login. El proceso de boot se encuentra ejemplificado en el video de demostración.


```

project2@project2-VirtualBox: ~
File Edit View Search Terminal Tabs Help
project2@project2-VirtualB... x project2@project2-VirtualB... x project2@project2-VirtualB... x
Failed to init entropy source hwrng

Initializing AES buffer

Enabling JITTER rng support

Initializing entropy source jitter

Starting OpenBSD Secure Shell server: sshd
done.
Starting rpcbind daemon...done.
Starting advanced power management daemon: No APM support in kernel
(failed.)
Starting bluetooth: bluetoothd.
Starting syslogd/klogd: done
[ ok ] Starting Avahi mDNS/DNS-SD Daemon: avahi-daemon
Starting Telephony daemon
Starting Linux NFC daemon
Starting tcf-agent: OK
umount: /mnt/.psplash: no mount point specified.

Poky (Yocto Project Reference Distro) 3.0.3 raspberry4-64 ttyAMA0

raspberrypi4-64 login: root
Password:
root@raspberrypi4-64:~#
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Online 4:51 | ttyUSB0

```

Figura 4. Proceso de *Boot* completo.

Para el desarrollo de la aplicación se hace uso de la plataforma eclipse la cual se configura para poder hacer el debug desde la misma aplicación y correr en la Raspberry Pi 4 como se muestra en la figura 5.

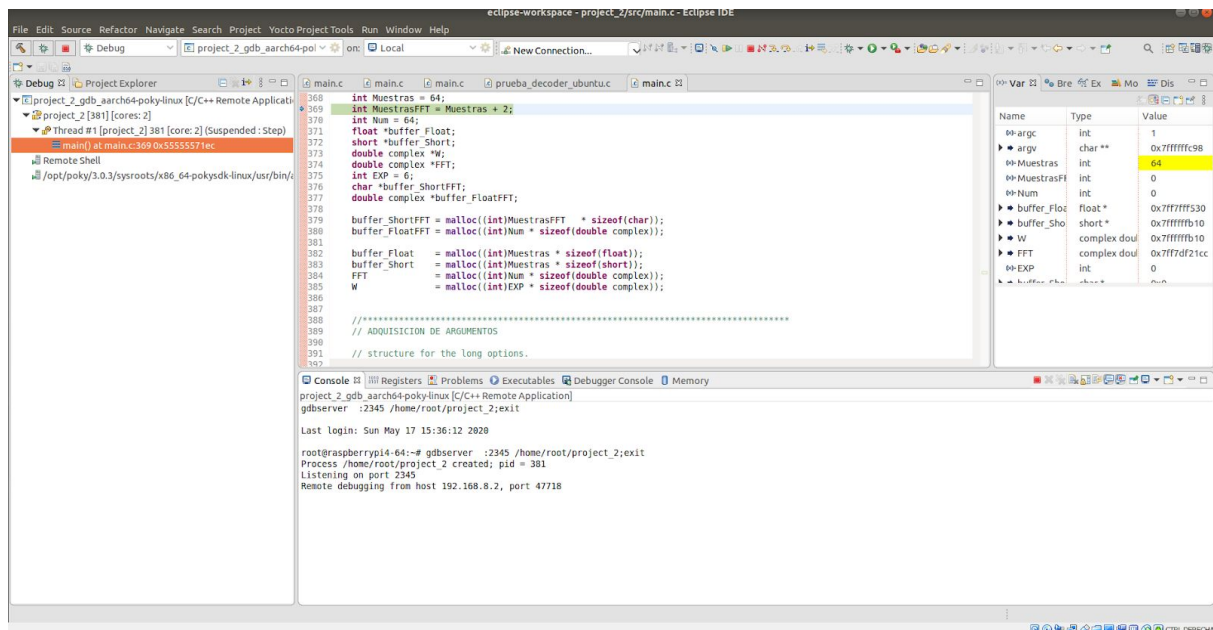


Figura 5. Debug ejecutado en la Raspberry Pi 4 desde eclipse.

Codificador y Decodificador de audio con FFT usando Python y C en *Host*

Aproximación usando Python

Basado en Implementación en Python 3.7 y utilizando bibliotecas de alto nivel para el procesamiento de las muestras de audio provenientes del archivo WAV se presentan de los siguientes elementos.

Un módulo para la lógica general para la construcción del algoritmo tanto para la generación de los vectores como los coeficientes conjugados complejos mediante la aplicación de la transformada rápida de Fourier (FFT), su correspondiente almacenamiento en archivo en binario de 16 bits para finalmente su lectura y ensamblado en sus valores equivalentes. Para finalmente aplicar la inversa de Fourier (IFFT)

1. Obtención de la Transformada rápida de Fourier FFT utilizando la biblioteca `numpy` de 64 bits por medio de vectores de tipo de dato complejo
2. Los elementos de dichos vectores son codificados con una profundidad de 16 bits cada uno
3. De momento la solución actual no contempla disminución de la la profundidad en bits.
4. La lectura del archivo que contiene los coeficientes en binario se pueden recuperar mediante una lectura consecutiva de las partes reales e imaginarias hasta acabar con el buffer.
5. Dicha recuperación desde archivo se la complementa con sus correspondientes vectores de coeficientes conjugados dispuestos de manera simétrica.
6. El script de python finalmente dispone de comandos para la creación de gráficos para visualizar la señal obtenida

Aproximación usando C

Además de la aproximación empleando Python también se optó por usar C para empezar a realizar un puente entre la ejecución en el *Host* en punto flotante y la ejecución en el *Target* con punto fijo. Las bibliotecas más importantes que se agregan son la `complex.h` y la `math.h` que ayudan para el procesamiento de datos y la generación de las constantes W de la FFT.

Siguiendo con las reglas establecidas por el docente, se genera un código con las siguientes características:

1. Generación de constantes para una FFT/IFFT de 64 bits.
2. Posibilidad de codificar con FFT bloques de 64 muestras de un archivo de audio muestreado a una tasa de 8 kHz y con un *bitdepth* de 16 bits .
3. Los coeficientes resultantes de la codificación con FFT deben ser almacenados en variables de 8, 4, 2 o 1 bit según se desee.
4. Posibilidad de decodificar con una IFFT el archivo codificado que se genera siguiendo los puntos anteriores y guardar los resultados en un archivo de audio.
5. **(Extra)** Generar un archivo csv con la salida de la IFFT para graficar los resultados de manera más sencilla.

En el **video de implementación 1 de la sección de anexos** se menciona que los cálculos en esta sección se efectúan con punto flotante; esto hace de este modo para facilitar el paso a punto fijo en el

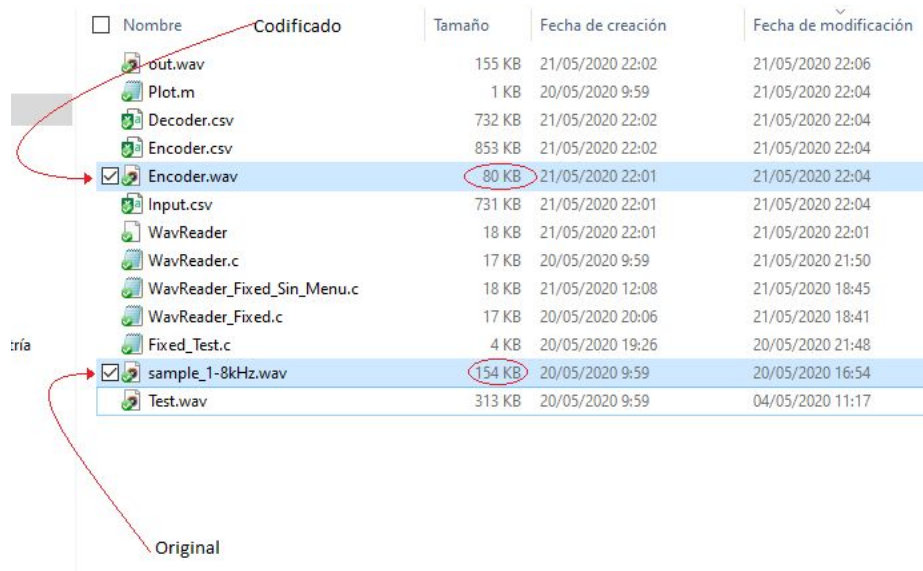
siguiente apartado, luego de lograr resultados satisfactorios en este. No obstante, la escritura del archivo codificado se hace en punto fijo.

Los archivos wav con *bitdepth* de 16 bits por defecto poseen sus variables guardadas por muestra en una configuración $Q_s 0.15$., siendo necesario realizar una conversión de punto fijo a flotante para la lectura de los wav y también para la escritura de estos. Los coeficientes de la FFT se guardan también en punto fijo, pero con la diferencia de que estos coeficientes tienen un tamaño de 8 bits con una configuración $Q_s - 1.8$ aprovechando que los coeficientes escalados no superan el valor de ± 0.5 .

De esta manera se puede ver como cada juego de coeficientes (real + imaginario) tiene un tamaño de 16 bits, con esto podemos inferir que el tamaño final del archivo codificado debería poseer el mismo tamaño que el archivo original, por que ambos poseen 16 bits por muestra. Con esto en mente entonces ¿Qué beneficio posee esta aproximación y por qué se le denomina compresor?

Esta pregunta se responde teniendo en cuenta la propiedad de la transformada de fourier de números completamente reales. Esta dice que las transformaciones de vectores de números completamente reales poseen simetría par. Con esto en cuenta se guardan únicamente la primera mitad de los coeficientes ya que con estos se pueden reconstruir los siguientes aplicando un conjugado a los valores.

En la figura 6 se muestra un ejemplo de la reducción del tamaño de los archivos de audio, siendo en este caso *sample_1-8kHz.wav* el audio original con 154 KB, mientras que su análogo codificado corresponde a *Encoder.wav* con casi la mitad del tamaño (80 KB).



Nombre	Codificado	Tamaño	Fecha de creación	Fecha de modificación
out.wav		155 KB	21/05/2020 22:02	21/05/2020 22:06
Plot.m		1 KB	20/05/2020 9:59	21/05/2020 22:04
Decoder.csv		732 KB	21/05/2020 22:02	21/05/2020 22:04
Encoder.csv		853 KB	21/05/2020 22:02	21/05/2020 22:04
Encoder.wav		80 KB	21/05/2020 22:01	21/05/2020 22:04
Input.csv		731 KB	21/05/2020 22:01	21/05/2020 22:04
WavReader		18 KB	21/05/2020 22:01	21/05/2020 22:01
WavReader.c		17 KB	20/05/2020 9:59	21/05/2020 21:50
WavReader_Fixed_Sin_Menu.c		18 KB	21/05/2020 12:08	21/05/2020 18:45
WavReader_Fixed.c		17 KB	20/05/2020 20:06	21/05/2020 18:41
Fixed_Test.c		4 KB	20/05/2020 19:26	20/05/2020 21:48
sample_1-8kHz.wav		154 KB	20/05/2020 9:59	20/05/2020 16:54
Test.wav		313 KB	20/05/2020 9:59	04/05/2020 11:17

Figura 6. Demostración de compresión de archivos de audio.

Como segunda prueba para comprobar el correcto funcionamiento, se codifica y decodifica un tono sinusoidal para posteriormente graficar el resultado con MATLAB y comparándolo con el tono original, obteniendo los resultados en la figura 7 donde se puede observar un desfase, este se da por que durante la lectura del archivo wav no se hace distinción del encabezado y es tomado en cuenta

como una muestra. Para evitar problemas en el análisis se elimina este *header* y se coloca como valor 0 aunque sigue ocupando espacios de memoria, esto genera este desfase artificial.

Finalmente se calcula el error cuadrático medio (**MSE**) **siendo este de 2.1×10^{-4}** y se realiza una prueba de duración de ejecución en *Host* con un archivo de audio de 313 KB; obteniendo para la codificación una duración de 3.243612s y para la decodificación un tiempo de 4.981087s.

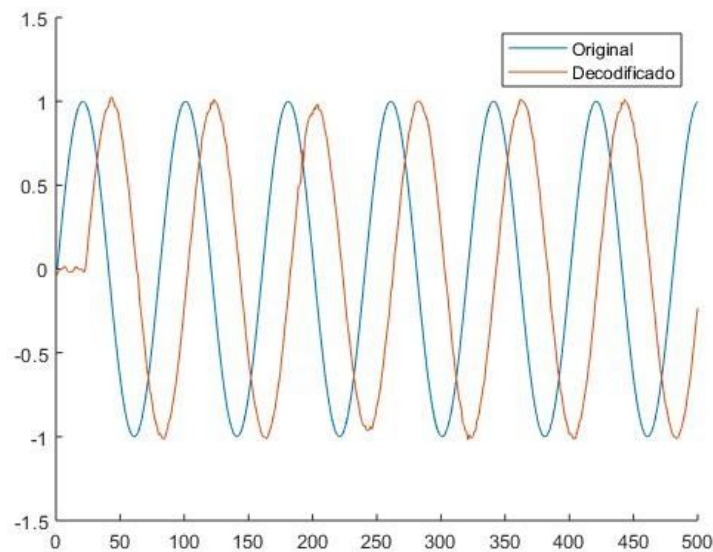


Figura 7. Comparación gráfica entre tono original y descomprimido usando punto flotante.

Durante esta implementación se presentaron las siguientes **dificultades**:

- La inclusión de la biblioteca `math.h` en la compilación del archivo C.
- Tratamiento del encabezado del wav. Esto se soluciona eliminando las muestras y dejando en 0 todo el encabezado.
- Generación del encabezado para el wav generado en la IFFT. Para que los archivos wav sean reproducidos por un lector de audio es necesario el encabezado del archivo, no obstante a falta de tiempo y conocimiento para la generación de este no se emplea. En el video de implementación se explica que una manera tosca de solucionar este problema es copiando el encabezado del wav original y pegando directamente al inicio del wav descomprimido.

Para realizar una comparación con el siguiente sección se incluye el código de la FFT usando operaciones punto flotante como se muestra en la figura 8.

```

void fft(double complex *X, unsigned short EXP, double complex *W, unsigned short SCALE){
    double temp_re;      /* Temporary storage of complex variable */
    double temp_im;

    double U_re;          /* Twiddle factor W^k */
    double U_im;

    unsigned short i,j;
    unsigned short id;     /* Index for lower point in butterfly */
    unsigned short N=1<<EXP; /* Number of points for FFT */
    unsigned short L;      /* FFT stage */

    unsigned short LE;     /* Number of points in sub DFT at stage L and offset
                           to next DFT in stage */

    unsigned short LE1;    /* Number of butterflies in one DFT at stage L.
                           Also is offset to lower point in butterfly at stage L */

    float scale;
    scale = 0.5;
    if (SCALE == 0){
        scale = 1.0;
    }

    /* FFT butterfly */
    for (L=1; L<=EXP; L++){
        LE=1<<L;          /* LE=2^L=points of sub DFT */
        LE1=LE>>1;        /* Number of butterflies in sub-DFT */
        U_re = 1.0;
        U_im = 0.;
        for (j=0; j<LE1;j++){
            /* Do the butterflies */
            for(i=j; i<N; i+=LE) {
                id=i+LE1;
                temp_re = (creal(X[id])*U_re - cimag(X[id])*U_im)*scale;
                temp_im = (cimag(X[id])*U_re + creal(X[id])*U_im)*scale;

                X[id] = (creal(X[i])*scale - temp_re) + (cimag(X[i])*scale - temp_im) * I;
                X[i] = (creal(X[i])*scale + temp_re) + (cimag(X[i])*scale + temp_im) * I;
            }
            /* Recursive compute W^k as U*W^(k-1) */
            temp_re = U_re*creal(W[L-1]) - U_im*cimag(W[L-1]);
            U_im = U_re*cimag(W[L-1]) + U_im*creal(W[L-1]);
            U_re = temp_re;
        }
    }
}

```

Figura 8. Función FFT implementada en punto flotante.

Codificador y Decodificador de audio con FFT usando C en *Target* (Fixed-point)

En este punto se modifica el código planteado en la sección anterior para hacer todas las operaciones matemáticas haciendo uso de punto fijo. Como primer cambio importante para alcanzar este objetivo, se elimina la biblioteca *complex.h*, teniendo en cuenta que las variables y operaciones que provee son en tipo *double* y la constante necesidad de hacer redondeos y truncamientos de manera manual se desea evitar.

Por estos motivos se opta por crear una estructura llamada *Complex* (ver figura 9) la cual contiene dos variables tipo *integer* de 64 bits que corresponden a el factor real e imaginario. Esta estructura sustituirá en el código todas las variables *double complex* de la biblioteca *complex.h*.

```

31  typedef struct{
32      int64_t re;
33      int64_t im;
34  }Complex;

```

figura 9. Estructura *Complex* para sustituir variables *double complex*

Siguiendo con las reglas establecidas por el docente, siguen siendo las mismas con la diferencia de que se tiene que usar punto fijo. Con esto se debe cumplir:

1. Generación de constantes para una FFT/IFFT de 64 bits.
2. Posibilidad de codificar con FFT bloques de 64 muestras de un archivo de audio muestreado a una tasa de 8 kHz y con un *bitdepth* de 16 bits .
3. Los coeficientes resultantes de la codificación con FFT deben ser almacenados en variables de 8, 4, 2 o 1 bit según se desee.
4. Posibilidad de decodificar con una IFFT el archivo codificado que se genera siguiendo los puntos anteriores y guardar los resultados en un archivo de audio.
5. **(Extra)** Generar un archivo csv con la salida de la IFFT para graficar los resultados de manera más sencilla.

El principal reto de esta sección era el decidir el formato de punto fijo que se usarán para los cálculos y generación de los coeficientes de la FFT. Por comodidad se contemplaron solo dos opciones, la primera siendo el formato $Q_s1.15$ por ser el formato en que se encuentra escrito por defecto los archivos wav, y por otra parte se tenía el $Q_s - 1.8$ correspondiente al formato decidido en el punto anterior para el almacenado de los coeficientes de la FFT.

De entre los dos finalmente se opta por emplear el $Q_s1.15$ por poseer más bits de exactitud. De este modo al estar usando registros de 32 bits con expansión a 64 para las multiplicaciones se establece que para todo el código el formato de punto fijo usado es de $Q_s16.15$, siendo empleado el $Q_s - 1.8$ únicamente para la lectura y escritura de los archivos codificados con FFT.

Esta aproximación se ejecuta en el Host como se muestra en el **vídeo implementación 2 de la sección de anexos** y en el Target como se indica en el **vídeo implementación 3 de la sección de anexos**. En ambos casos se utiliza un tono para hacer las pruebas de funcionamiento obteniendo los resultados de las figura 10 y figura 11 respectivamente.

Finalmente se calcula el error cuadrático medio (**MSE**) **siendo este de 2.254×10^{-4}** y se realiza una prueba de tiempo de ejecución en *Host* y en *Target* usando un archivo de audio de 313 KB. El primero dura 1.608404s para la codificación y para decodificación consume 1.500935s; por otra parte, en Target presenta un tiempo de codificaciones de 6.1798s y un tiempo de decodificación es de 4.84227s.

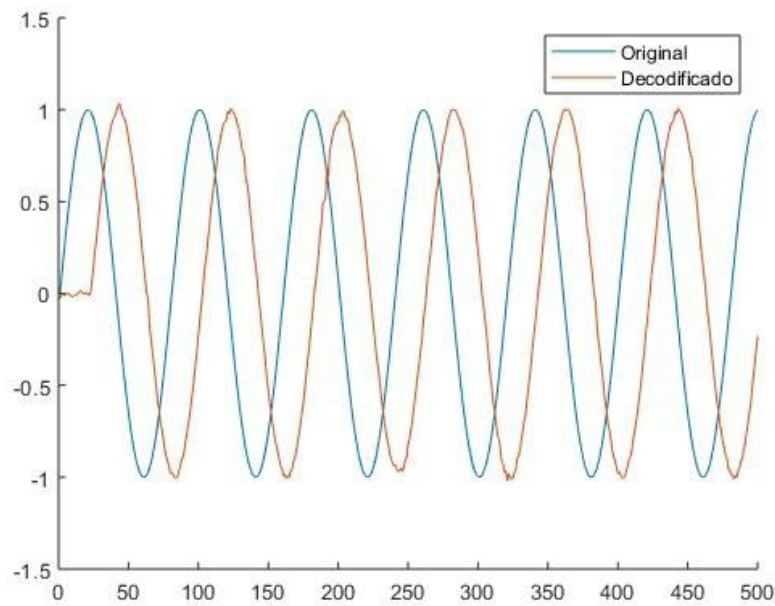


Figura 10. Comparación tono original y descomprimido usando punto fijo, ejecutado en *Host*.

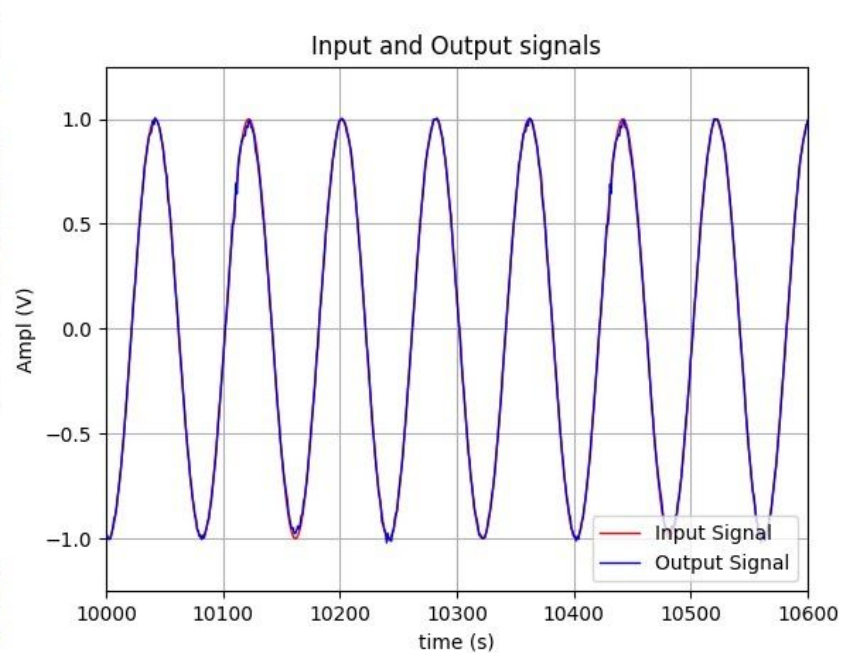


Figura 11. Comparación tono original y descomprimido usando punto fijo, ejecutado en *Target*.

Como diferencias en la función FFT se tienen 3 aspectos importantes. El primero es el cambio en el tipo de variables, todas las que eran de tipo *double* pasan a ser *integer* de 64 bits. Luego tenemos el desplazamiento de 15 bits en los coeficientes constantes para mantener el formato $Q_{16.15}$. Finalmente se tienen las multiplicaciones estas al multiplicar dos variables $Q_{16.15}$ terminan siendo variables tipo $Q_{32.30}$ por lo que se debe hacer un desplazamiento de 15 bits para devolverlas a el formato estándar que se ha estado empleando de $Q_{16.15}$. Todo esto se muestra en el código de la figura 12.


```

307 void fft(Complex *X, unsigned short EXP, Complex *W, unsigned short SCALE, int Fix){
308     int64_t temp_re;          /* Temporary storage of complex variable */
309     int64_t temp_im;
310     int64_t U_re;             /* Twiddle factor W^k */
311     int64_t U_im;
312
313     unsigned short i,j;
314     unsigned short id;        /* Index for lower point in butterfly */
315     unsigned short N=1<<EXP; /* Number of points for FFT */
316     unsigned short L;         /* FFT stage */
317     unsigned short LE;        /* Number of points in sub DFT at stage L and offset
318                                to next DFT in stage */
319     unsigned short LE1;       /* Number of butterflies in one DFT at stage L.
320                                Also is offset to lower point in butterfly at stage L */
321     int64_t scale;
322     scale = 0.5*(1<<Fix);
323     if (SCALE == 0){
324         scale = (1<<Fix);
325     }
326
327     /* FFT butterfly */
328     for (L=1; L<=EXP; L++){
329         LE = 1<<L;           /* LE=2^L=points of sub DFT */
330         LE1 = LE>>1;         /* Number of butterflies in sub-DFT */
331
332         U_re = 1.0*(1<<Fix);
333         U_im = 0;
334
335         for (j=0; j<LE1;j++){
336             /* Do the butterflies */
337             for(i=j; i<N; i+=LE) {
338                 id=i+LE1;
339                 temp_re = X[id].re*U_re/(1<<Fix) - X[id].im*U_im/(1<<Fix);
340                 temp_re = temp_re*scale/(1<<Fix);
341
342                 temp_im = X[id].im*U_re/(1<<Fix) + X[id].re*U_im/(1<<Fix);
343                 temp_im = temp_im*scale/(1<<Fix);
344
345                 X[id].re = X[i].re*scale/(1<<Fix) - temp_re;
346                 X[id].im = X[i].im*scale/(1<<Fix) - temp_im;
347                 X[i].re = X[i].re*scale/(1<<Fix) + temp_re;
348                 X[i].im = X[i].im*scale/(1<<Fix) + temp_im;
349             }
350             /* Recursive compute W^k as U*W^(k-1) */
351             temp_re = (U_re*W[L-1].re/(1<<Fix) - U_im*W[L-1].im/(1<<Fix));
352             U_im = (U_re*W[L-1].im/(1<<Fix) + U_im*W[L-1].re/(1<<Fix));
353             U_re = temp_re;
354         }
355     }
356 }

```

Figura 12. Función FFT implementada en punto fijo.

Las dificultades en esta sección siguen correspondiendo con las mencionadas en la sección anterior y se le suman que durante la implementación en *Target* surgió la peculiaridad de que no permitía hacer el paso de char a int de manera directa, no reconociendo el bit número 8 del char como el bit de signo, por lo que se tuvo que recurrir a hacer el complemento A2 de manera manual con la función de la figura 13.


```

59 float fix_to_flo_Forced(int x, int e){
60     // f es el numero en punto fijo de tipo int
61     // e es la cantidad de bits que se le dio la seccion decimal
62     double f;
63     int sign;
64     int c;
65
66     c = abs(x);
67     sign = 1;
68
69     if (x > 127){
70         /* Las siguientes 3 lineas es para devolverlo del complemento a 2
71            si el numero original era negativo */
72
73         c = abs(c - 256);
74         sign = -1;
75     }
76
77     /*Lo que se hace es simplemente multiplicar el numero
78     en punto fijo dividido por 2 elevado a la e*/
79     f = (1.0 * c)/(1<<e);
80     f = f * sign;
81     return f;
82 }
83

```

Figura 13. Función *fix_to_flo_Forced*.

Optimización de Codificador/Decodificador de audio con FFT usando NEON en *Target*

Habiendo realizado pasado de una programación punto flotante a punto fijo se desea optimizar aún más el tiempo de ejecución del codificador/decodificador de audio con FFT. Para cumplir con este objetivo se opta por emplear las funciones de *arm_neon.h*. La tecnología Arm Neon es una extensión de arquitectura avanzada de datos múltiples de instrucción única (SIMD) para los procesadores de las series Arm Cortex-A y Cortex-R. La tecnología de Neon es una arquitectura SIMD compacta. Los registros de neón se consideran vectores de elementos del mismo tipo de datos, con instrucciones de neón que operan en múltiples elementos simultáneamente. La tecnología admite varios tipos de datos, incluidas las operaciones de punto flotante y entero. Esta biblioteca de C/C++ permite forzar al compilador a realizar operaciones vectoriales; en otras palabras, ejecutar operaciones no dependientes en forma paralela siempre y cuando el hardware lo permita.

Inicialmente se intenta realizar la optimización de la función *fix_to_flo_Forced* (figura 13) que se emplea durante la lectura de datos en el proceso de decodificación.

Se escoge una vectorización de 4 muestras y los cambios en el código inician con la modificación en los tipos de variables usados, todos son sustituidos por sus análogas vectoriales (por ejemplo *int64_t* pasa a ser *int32x4_t*). De igual manera se cambia los operandos de sumas, restas, multiplicaciones y

divisiones escalares por sus respectivas operaciones vectoriales las cuales se pueden encontrar en [2]. Finalmente la función *fix_to_flo_Forced* queda como se muestra en la figura 14.

Desafortunadamente por falta de tipo y por inconvenientes con incompatibilidad en manejo de tipos de datos no fue posible incorporar la función modificada directamente en el Codificador/Decodificador con FFT, aunque se desea recalcar que es completamente realizable. Para probar el funcionamiento y demostrar los alcances de esta optimización se realiza una prueba de concepto probando la función por separado con valores aleatorios.

```
int32x4_t fix_to_flo_Forced(int32x4_t x,int e){
    int32x4_t comp = vdupq_n_s32(127); //Inicializa un vector para la comparación
    int32x4_t c = vabsq_s32(x); //c = abs(x);
    int32x4_t sign = vdupq_n_s32(1); //sign = 1;

    uint32x4_t aux = vcgtq_s32(x,comp); //Se generan mascarar para la ejecución de If
    int32x4_t mask = (int32x4_t)aux;
    int32x4_t mask_neg = vmvnq_s32(mask);

    //no se cumple la condición del if
    int32x4_t f_F = vshrq_n_s32(c,e); //c / 2^e
    f_F = vmulq_s32(f_F,sign); // f = f * sign;
    f_F = vandq_s32(f_F,mask_neg); // Se ejecuta un and bit a bit para dejar solo los valores que no cumplen el if

    //se cumple la condición del if
    comp = vdupq_n_s32(256);
    c = vabdq_s32(vabsq_s32(x),comp); // Se realizan en c las operaciones dentro del if
    int32x4_t f_T = vshrq_n_s32(c,e);
    f_T = vmulq_s32(f_T,sign);
    f_T = vandq_s32(f_T,mask);

    int32x4_t f = vaddq_s32(f_F,f_T); //Se suman los aportes tanto de la condición true como false
    return f;
}
```

Figura 14. Función *fix_to_flo_Forced* modificada con operaciones NEON.

```
<terminated> test_NEON_gdb_aarch64-poky-linux [C/C++ Remote Application] Remote Shell (Terminated N
gdbserver :2345 /home/root/test_NEON;exit

Last login: Sun May 17 15:49:55 2020 from 192.168.8.2

root@raspberrypi4-64:~# gdbserver :2345 /home/root/test_NEON;exit
Process /home/root/test_NEON created; pid = 425
Listening on port 2345
Remote debugging from host 192.168.8.2, port 41102
Inicio de reloj para operaciones no vectoriales...
Tiempo de ejecución tomó 7010 clicks completar el proceso (0.007010 segundos).

Inicio de reloj para operaciones vectoriales...
Tiempo de ejecución tomó 6 clicks completar el proceso (0.000006 segundos).

Child exited with status 0
logout
```

Figura 15. Ejecución comparativa de la función *fix_to_flo_forced* sin optimización vectorial y con optimización vectorial.

Se puede observar en la figura 15 que el tiempo de ejecución requerido para aplicar dichas funciones a un vector de 64000 muestras varía considerablemente con la optimización utilizando la vectorización de NEON.

Conclusiones

Se completo exitosamente el proceso de desarrollo de sistema operativo personalizado para la plataforma de desarrollo seleccionada (Raspberry Pi 4), esto resultó en un aprovechamiento en cuanto a conocimientos adquiridos muy valioso, de igual manera se pudo observar de primera mano cada una de las etapas de dicho proceso, tanto el U-Boot, el Kernel así como el File System, y haciendo uso de herramientas de Cross-compilación desde un host con diferente arquitectura.

Realizando una comparación entre los resultados de la implementación en C de punto flotante y punto fijo se observa cómo los valores de error (MCE) son prácticamente iguales, demostrando que la implementación en punto fijo además de presentar un esfuerzo computacional menor, presentan una exactitud casi idéntica que su análogo en punto flotante pero teniendo el beneficio, como se ve en los resultados del *Host*, de una reducción de más del un 50% en tiempo de ejecución.

No obstante, como se menciona en las dificultades, al usar punto fijo en ocasiones es necesario agregar funciones para mantener la integridad de los números; Como en el caso ocurrido durante la experimentación, en el que el Target no consideraba automáticamente el octavo bit de las variables tipo char como el bit de signo y hubo que hacer la conversión a complemento A2 de manera manual.

Esto aunque no significa mayor complejidad de programación como se puede apreciar en la simplicidad de la función, no obstante, si consume tiempo detectar estos errores y es necesario correr línea por línea el código en función de encontrar estos fallos específico, siendo posible que exista más de uno.

El uso de la tecnología Neon que permite la paralelización de los cálculos mediante operaciones vectoriales mejor considerablemente los tiempos de ejecución, esto sin embargo conlleva un proceso de modificación del código y las funciones implementadas que resulta laborioso y de una complejidad intermedia.

Bibliografía

[1] Araya J. & Interiano E. (2020) *Project 2: Implementation of an Audio Codec in an Embedded Device*. LECTURE: MP-6157 TÉCNICAS DE ADQUISICIÓN Y PROCESAMIENTO DE DATOS. Instituto Tecnológico de Costa Rica

[2] DS-5. (2020) ARM Software development tools. Extraído de:

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0491h/BABIBDGG.html>

Anexos

GitHub:

https://github.com/RonnyZF/project_2_audio_codec

Video con proceso de boot de Target: <https://www.youtube.com/watch?v=3eslimkhSgY>

Video de implementación 1: <https://www.youtube.com/watch?v=UzZJhjFyj6I>

Video de implementación python: <https://www.youtube.com/watch?v=5F4cjOKxMK0>

Video de implementación 2: <https://www.youtube.com/watch?v=tVqrYyEsQhs>

Video de implementación 3: <https://youtu.be/PHYP3oXo0IU>

Video de implementación 4: <https://youtu.be/FIvNUH0lmO8>

PlayList con todos los videos:

<https://www.youtube.com/playlist?list=PL8xuu8gJxQepIFbppUUxMDEHF6oeCLJa2>