Escuela de Ingeniería en Computadores

CE 4301 — Arquitectura de Computadores I

Diseño e Implementación de un ASIP para reverberación de audio

Proyecto Individual o Parejas

Profesor: Luis Chavarria Zamora

Estudiantes:

Carlos Andrés Mata Calderón

Felipe Vargas Jiménez

I Semestre 2024

Listado de requerimientos del sistema

1. **Cálculo de Reverberación**: Existe un algoritmo en assembly que es capaz de calcular para cada x[n] de entrada obtener su y[n] para aplicar el efecto de reverberación a la señal de audio.
2. **Cálculo Reverberación Inversa**: Existe un algoritmo en assembly que es capaz de calcular para cada x[n] de entrada obtener su y[n] para aplicar el efecto de reverberación a la señal de audio.
3. **Lectura y Escritura de Datos**: Uso de un buffer en assembly para la lectura y escritura de los datos del audio en crudo (sin comprensión) en .bin
4. **Memoria del Programa**: Uso de un buffer circular en assembly para gestionar los datos necesarios para el cálculo del y[n] dado un x[n]. Este buffer circular almacena y[n] anteriormente calculados para el caso de Reverberación o x[n] anteriormente utilizados para el caso de Reverberación Inversa.
5. **Procesamiento en Punto Fijo**: Implementación de aritmética (suma, resta, y multiplicación) en formato punto fijo Q. Donde la aritmética es independiente de los decimales de la representación.
6. **Generador de Audio a Formato Crudo**: Se emplea un lenguaje de programación de alto nivel para transformar un archivo de audio en formato .wav en un archivo crudo en formato binario (.bin), utilizando la representación en punto fijo Q. Este proceso permite especificar la cantidad de decimales utilizados para la representación a través de un argumento. Además, los tres primeros números binarios del archivo resultante se reservan para almacenar los datos de modo de operación, k y alpha, necesarios para la ejecución de un algoritmo específico en lenguaje ensamblador.

7. **Generador de Formato Crudo a Audio**: Se emplea un lenguaje de programación de alto nivel para procesar archivos en formato binario (.bin) y convertirlos en archivos de audio en formato .wav. Este proceso interpreta cada número binario utilizando el formato de punto fijo Q, y permite especificar la cantidad de decimales utilizados para la representación mediante un argumento. Adicionalmente, se descartan los primeros tres números binarios del archivo, ya que estos no forman parte de la composición musical.

Elaboración de opciones de solución al problema

- ***Emulador:***

QEMU es un emulador de fuente abierta y una solución de virtualización que brinda soporte para múltiples conjuntos de instrucciones arquitectónicas, incluyendo RISC-V. Su amplia adopción en el ámbito del desarrollo y la verificación de software se debe a su versatilidad y capacidad para soportar una variedad de sistemas operativos este posibilita la simulación de entornos completos así como de aplicaciones individuales en diversas configuraciones de RISC-V.

Asimismo para la arquitectura ARM, QEMU también se destaca como una opción preferente para la simulación. Es compatible con un extenso espectro de variantes ARM facilitando así el desarrollo y las pruebas de software en un ambiente simulado. QEMU es capaz de replicar sistemas integrales, que incluyen el procesador central, periféricos y conectividad de red.

- ***Cantidad de Registros***

En cuanto a los registros en ARM, los registros se dividen en volátiles y no volátiles, determinando su papel en las llamadas a funciones. Los registros volátiles (r0 a r3, r12) se usan para parámetros y resultados temporales, y no se preservan entre llamadas de función. Los no volátiles (r4 a r11, r13 a r15), incluyendo el puntero de marco (r11), puntero de pila (r13), registro de vínculo (r14), y contador de programas (r15), mantienen su valor a través de llamadas, asegurando la continuidad del estado del programa.

Name	Use
R0	Argument / return value / temporary variable
R1–R3	Argument / temporary variables
R4–R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Imagen 1: Registros en ARM

Por otra parte tenemos a los registros en RISCV se dividen en 32 registros de propósito general, de punto flotante, de control y estado donde cada uno posee 32 bits esto para RV32, en la imagen se puede apreciar que del registro x4 al registro x7 son de variable temporal, al igual que del registro x28 al registro x31, en cuanto al registro que si se almacena en

memoriatenemos que va del x18 al x27

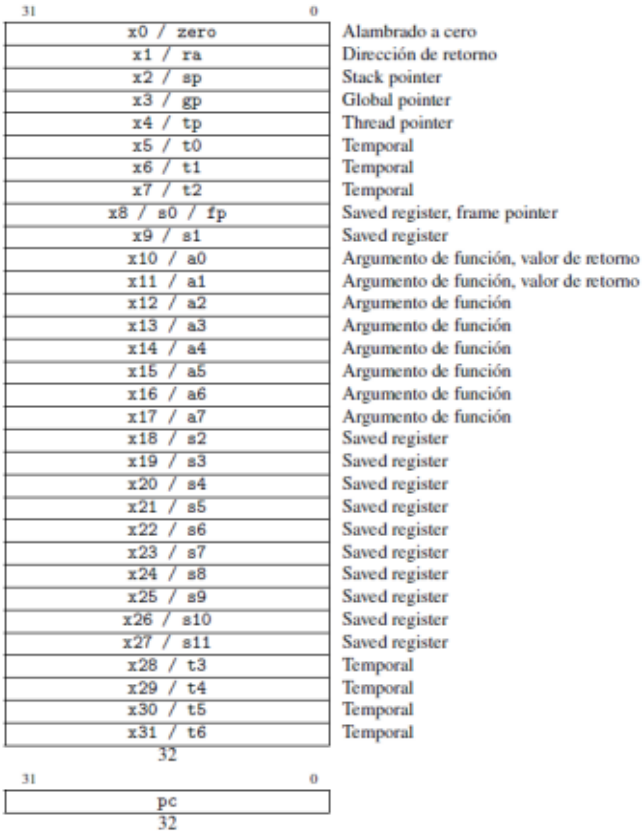


Imagen 2:Registros de RISCV

- **Formato de Instrucciones**

En la arquitectura ARM, las instrucciones se caracterizan por su simplicidad y claridad. Por ejemplo, para incrementar el valor del registro `r0` en 4, simplemente escribiremos la instrucción de la siguiente manera:

```
add r0, #4
```

Este ejemplo destaca la facilidad con la que se pueden expresar operaciones en ARM, gracias a su diseño intuitivo y a la flexibilidad de su conjunto de instrucciones.

Por otro lado, en la arquitectura RISC-V, la sintaxis para operaciones similares requiere una especificación más detallada. Para realizar la misma operación de sumar 4 al valor de un registro, tendríamos que utilizar la siguiente instrucción:

```
addi t0, t0, 4
```

Este nivel de detalle se debe a que RISC-V adopta un enfoque más estricto en cuanto a la estructura de sus instrucciones. En RISC-V, es necesario especificar explícitamente tanto el registro objetivo (`t0`) como el registro fuente (`t0`) en este caso, aun cuando la operación se realice sobre el mismo registro. Esta característica refleja la filosofía de diseño de RISC-V, que prioriza la uniformidad y la simplicidad en el nivel de sintaxis de las instrucciones, a costa de requerir una especificación más explícita por parte del programador.

- **Los System Calls**

Los System Calls son cruciales en este proyecto de reverberación, ya que permiten la interacción con el sistema operativo para realizar operaciones esenciales como abrir, cerrar, leer y escribir archivos. Comprender cómo las Arquitecturas de Conjunto de Instrucciones (ISA) gestionan estos llamados al sistema es fundamental.

En la arquitectura ARM, los llamados al sistema (system calls) se realizan a través de la instrucción SWI (Software Interrupt), seguida por un identificador numérico que indica la operación específica solicitada. Por ejemplo, SWI 0 puede ser utilizado para invocar una operación de lectura o escritura. La documentación y guías detalladas sobre cómo ARM maneja estos llamados al sistema se pueden encontrar en ARM System Calls, un recurso que explica claramente los argumentos requeridos por cada función del sistema operativo.

Para la arquitectura RISC-V, el proceso es similar pero utiliza la instrucción ECALL (Environment Call) para los llamados al sistema. ECALL sirve como un punto de interrupción que transfiere el control al manejador del sistema operativo con información sobre la operación deseada. Un recurso útil para comprender los system calls en RISC-V es RISC-V System Calls, complementado por tutoriales prácticos como RISC-V Linux System Calls Tutorial, que ofrece una guía paso a paso sobre cómo emplear ECALL en contextos específicos.

Dado que ambos, ARM y RISC-V, son emuladores que operan en entornos Linux, la referencia a las páginas de manual de Linux para system calls (Linux System Calls Man Pages) es invaluable. Estas páginas ofrecen una comprensión profunda de los valores y argumentos que deben pasarse a los registros al realizar llamadas al sistema, asegurando que las operaciones de archivo necesarias para el proyecto de reverberación se realicen correctamente.

Validación de opciones de solución

QEMU resulta ser más sencillo de utilizar en comparación con la librería de emulación de RISC-V, principalmente debido a la mayor disponibilidad de documentación. Además, al emular ARM, se utiliza GDB, una herramienta estándar para la depuración de programas, facilitando el acceso a información relevante a través de fuentes como Ubuntu's documentation. Este factor convierte al emulador en un aspecto crucial a considerar.

En cuanto a la cantidad de registros, a pesar de que RISC-V dispone de un número mayor, se prefiere ARM. La limitación en la cantidad de registros en ARM exige una utilización más estratégica de estos, obligando al programador a planificar cuidadosamente la estructura de su programa. Esta restricción, lejos de ser un impedimento, potencia una mayor comprensión y control sobre el código.

La estructura de las instrucciones en ARM, al ser más legible y contar con una sintaxis más sencilla, facilita significativamente la depuración de errores en el programa. Aunque RISC-V está ganando popularidad por ser una arquitectura completamente abierta, ARM cuenta con la ventaja de estar ampliamente adoptado en la industria. La familiaridad con ARM abre puertas al estudio de código publicado por otros programadores, gracias a una comunidad más establecida en comparación con RISC-V.

Por estas razones, se decidió que el desarrollo del código en lenguaje ensamblador se llevaría a cabo en ARM, basado en los análisis previamente mencionados. Esta elección se fundamenta en la facilidad de uso, las herramientas de depuración disponibles, la gestión óptima de los registros, y la legibilidad de las instrucciones, así como la sólida comunidad y presencia en la industria de ARM.

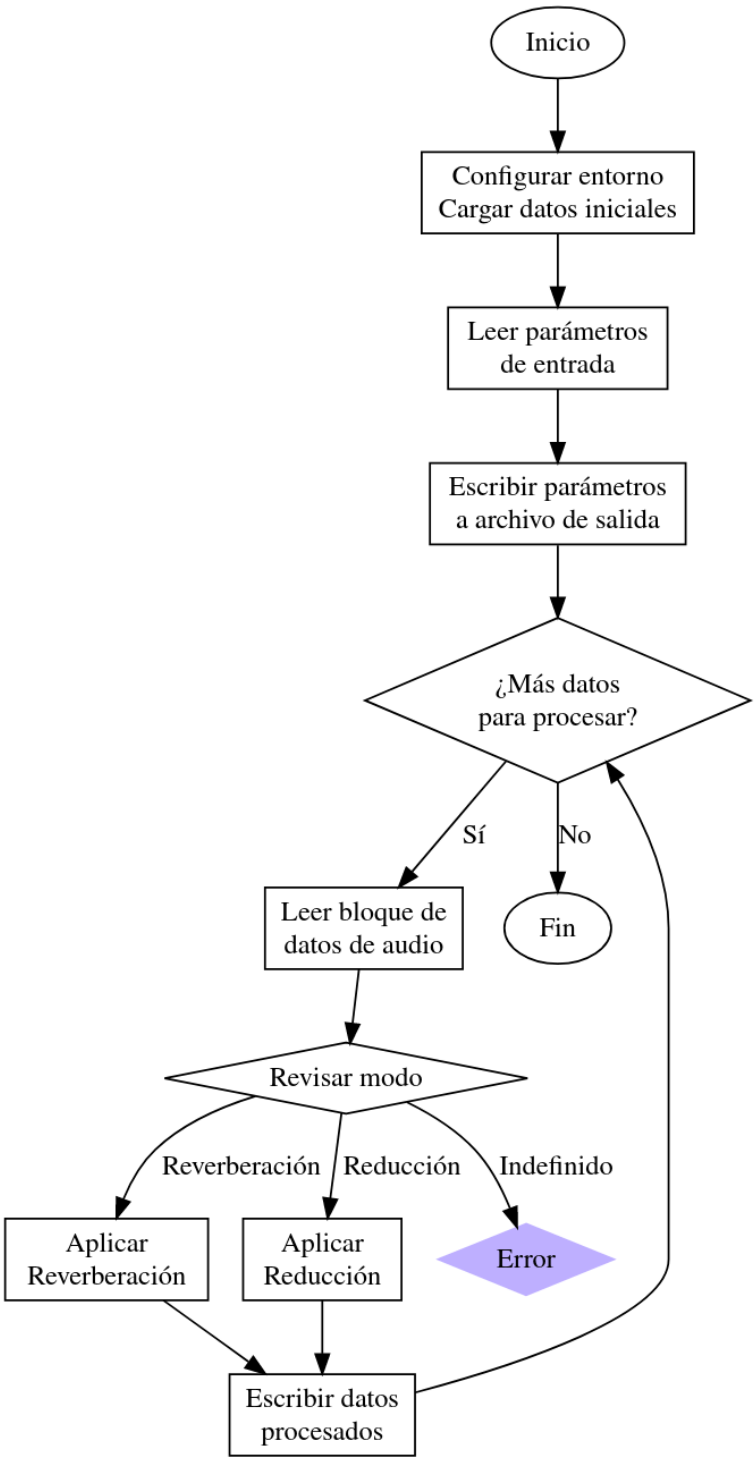
Anexo: Implementación de la Solución Escogida de Aplicación de Reverberación

El programa en ASSEMBLY se divide en cuatro grandes módulos: el módulo de lectura y escritura de archivos, el módulo de reverberación, el módulo de reverberación inversa y el módulo de multiplicación en formato Q.

Lectura y Escritura del Buffer

Se implementó un buffer especializado que administra tanto la lectura de los valores de entrada x[n] desde el archivo input.bin, como la selección del filtro adecuado para aplicar los efectos de reverberación o reverberación inversa. Este diseño separa claramente las tareas de lectura y escritura dentro del buffer, permitiendo una gestión eficiente del espacio de memoria. A través de este enfoque, el tamaño del buffer se mantiene independiente del volumen de los datos procesados, utilizando solo 1 KiB para cargar y almacenar tanto los datos de entrada x[n] como los resultados procesados y[n]. Esta optimización asegura una manipulación eficiente de los datos en el buffer, facilitando un procesamiento ágil sin comprometer la capacidad de almacenamiento o la velocidad de ejecución.

```
Inicio
  Preparar registros y entorno
  Cargar datos de entrada (1024 bytes en buffer de lectura)

  Leer parámetros de entrada desde archivo
  Escribir parámetros a archivo de salida

  Establecer variables para el procesamiento de audio
  Iniciar bucle de carga de datos
      Leer datos de audio en buffer desde archivo de entrada
      Si no se leyeron datos o hay un error, terminar el bucle

      Determinar si aplicar reverberación o reducción basado en el
modo
      Si el modo es reverberación
      Llamar a función de reverberación con parámetros necesarios
      Sino si el modo es reducción
      Llamar a función de reducción con parámetros necesarios
      Sino
      Salir con error

      Escribir los datos procesados al archivo de salida
      Repetir bucle con nueva carga de datos
  Fin del bucle

  Restaurar entorno y finalizar programa
Fin
```

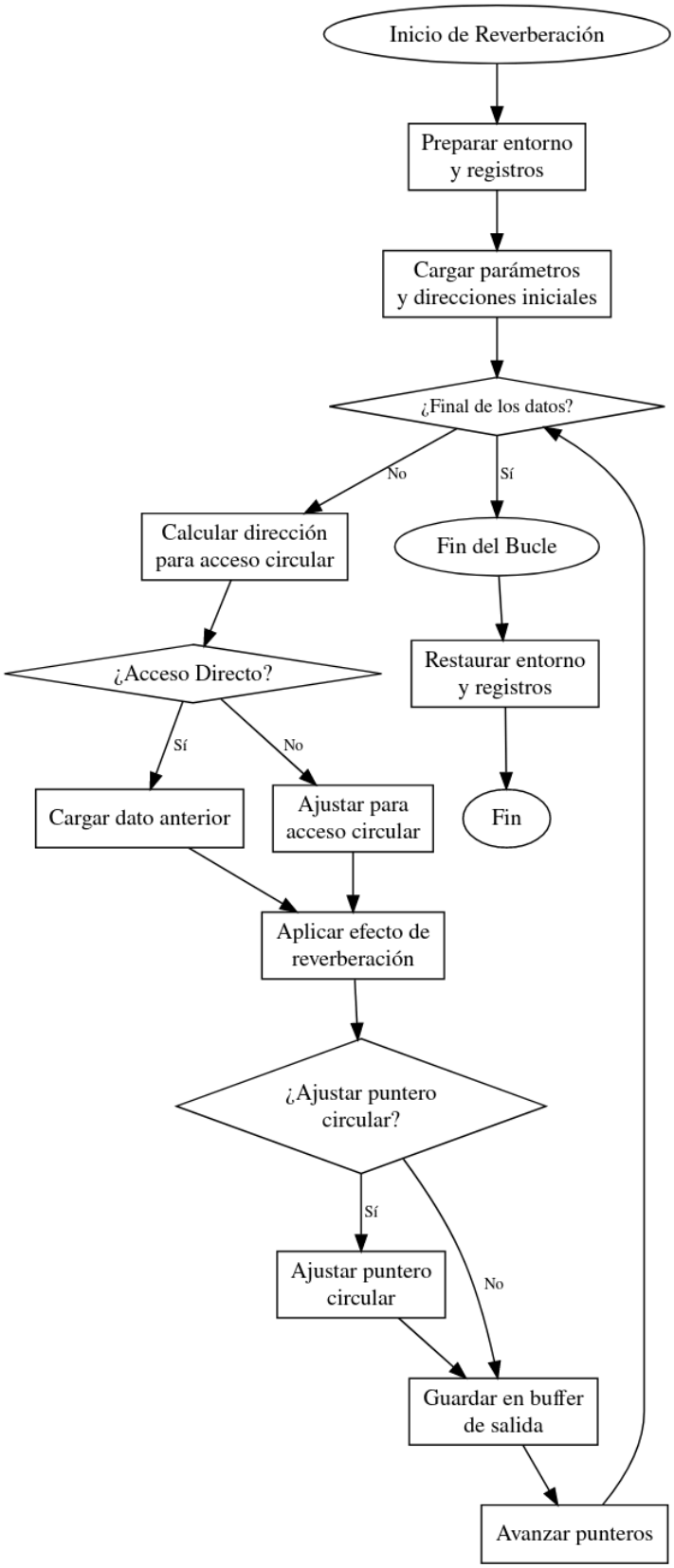

Inicio
Configurar entorno
Cargar datos iniciales
Leer parámetros
de entrada
Escribir parámetros
a archivo de salida
¿Más datos
para procesar?
Sí
No
Leer bloque de
datos de audio
Fin
Revisar modo
Reverberación
Reducción
Indefinido
Aplicar
Reverberación
Aplicar
Reducción
Error
Escribir datos
procesados

Circular Buffer

Se hace uso de un buffer circular principalmente para almacenar los 22,205 valores anteriores. Esta cifra específica se deriva de la necesidad de replicar un efecto de reverberación de 500 ms, considerando una frecuencia de muestreo de 44,100 Hz. Para lograr esto, es imprescindible retroceder 22,205 valores en el tiempo, razón por la cual el buffer circular debe tener, como mínimo, la capacidad para almacenar esa cantidad de valores.

Tanto la reverberación como su proceso inverso emplean algoritmos similares, diferenciándose principalmente en el cálculo de y[n] y en la información que se almacena en el buffer circular. Para el caso de la reverberación, se almacena el valor de y[n] en el buffer, mientras que para la inversa, lo que se guarda es el valor de x[n]. Esta distinción es fundamental para entender cómo se manipulan los datos en cada proceso, permitiendo que ambos efectos se implementen de manera eficiente utilizando una estructura común.

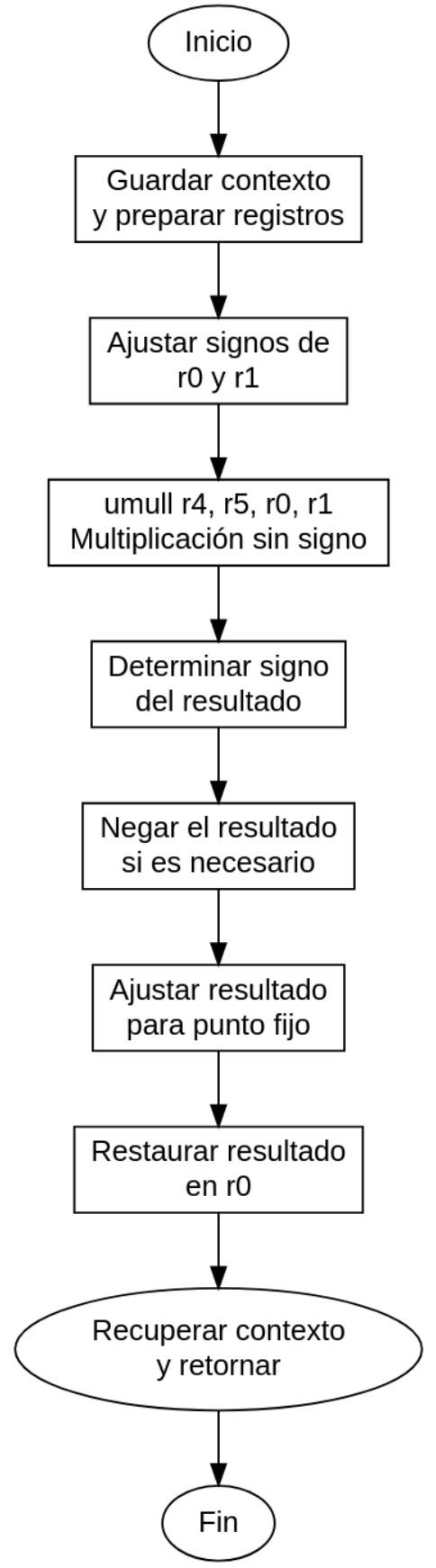
```
Inicio de reverberación
  Preparar entorno y registros

  Cargar parámetros y direcciones iniciales

  Bucle de reverberación:
     Si se alcanza el final de los datos, terminar bucle

     Calcular dirección para acceso circular
     Si el acceso es directo:
     Cargar dato anterior directamente
     Sino:
     Ajustar y cargar dato anterior con lógica circular

     Aplicar efecto de reverberación
     Cargar valor de entrada actual
     Calcular nuevo valor de salida
     Guardar nuevo valor en el buffer circular

     Verificar y ajustar puntero circular si es necesario
     Guardar nuevo valor en el buffer de salida
     Avanzar punteros
     Repetir bucle

  Terminar reverberación
  Restaurar entorno y registros
Fin
```


Inicio de Reverberación
Preparar entorno y registros
Cargar parámetros y direcciones iniciales
¿Final de los datos?
No
Sí
Calcular dirección para acceso circular
Fin del Bucle
¿Acceso Directo?
Restaurar entorno y registros
Sí
No
Cargar dato anterior
Ajustar para acceso circular
Fin
Aplicar efecto de reverberación
¿Ajustar puntero circular?
Sí
No
Ajustar puntero circular
Guardar en buffer de salida
Avanzar punteros

Multiplicación formato Q

El algoritmo empleado para multiplicar dos números en el formato Q es independiente de la cantidad de bits que conforman la parte fraccionaria. Esto se debe a que la cantidad de bits de la parte fraccionaria se especifica como un argumento dentro del algoritmo.

```
Inicio de multiply_fixed_point
  Guardar contexto
  Preparar registros para operación

  Ajustar signos de los operandos
  Si r0 < 0, hacer r0 = -r0
  Si r1 < 0, hacer r1 = -r1

  Realizar multiplicación sin signo
  Determinar si el resultado debe ser negativo
  Si es necesario, negar el resultado
  Ajustar el resultado para punto fijo

  Restaurar resultado en r0
  Recuperar contexto y retornar
Fin
```


Inicio
Guardar contexto y preparar registros
Ajustar signos de r0 y r1
umull r4, r5, r0, r1 Multiplicación sin signo
Determinar signo del resultado
Negar el resultado si es necesario
Ajustar resultado para punto fijo
Restaurar resultado en r0
Recuperar contexto y retornar
Fin

Referencias

ARM System Calls. Recuperado de https://arm.syscall.sh/

Linux Man Pages. Sección 2. Recuperado de https://man7.org/linux/man-pages/dir_section_2.html

Scott, W. (n.d.). RISC-V Linux System Calls. GitHub. Recuperado de https://github.com/scotws/RISC-V-tests/blob/master/docs/riscv_linux_system_calls.md

Harris, S., & Harris, D. (2015). Digital Design and Computer Architecture: ARM Edition. Morgan Kaufmann Publishers Inc., 340 Pine Street, Sixth Floor, San Francisco, CA, Estados Unidos. ISBN: 978-0-12-800056-4.