Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería Mecánica Eléctrica

DESCRIPCIÓN DE LAS MANERAS DE EMPLEAR LA EXTENSIÓN AVANZADA SIMD EN LAS ARQUITECTURAS ARMV7-A Y ARMV8-A DE LOS PROCESADORES ARM

Manuel Estuardo Osorio Chamam
Asesorado por la Inga. Ingrid Salomé Rodríguez de Loukota

Guatemala, abril de 2022

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

DESCRIPCIÓN DE LAS MANERAS DE EMPLEAR LA EXTENSIÓN AVANZADA SIMD EN LAS ARQUITECTURAS ARMV7-A Y ARMV8-A DE LOS PROCESADORES ARM

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

MANUEL ESTUARDO OSORIO CHAMAM
ASESORADO POR LA INGA. INGRID SALOMÉ RODRÍGUEZ DE LOUKOTA

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN ELECTRÓNICA

GUATEMALA, ABRIL DE 2022

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANA	Inga. Aurelia Anabela Cordova Estrada
VOCAL I	Ing. José Francisco Gómez Rivera
VOCAL II	Ing. Mario Renato Escobedo Martínez
VOCAL III	Ing. José Milton De León Bran
VOCAL IV	Br. Kevin Vladimir Armando Cruz Lorente
VOCAL V	Br. Fernando José Paz González
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

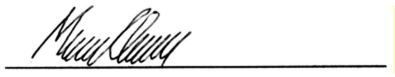
DECANA	Inga. Aurelia Anabela Cordova Estrada
EXAMINADOR	Ing. Christian Antonio Orellana López
EXAMINADOR	Ing. Sergio Leonel Gómez Bravo
EXAMINADOR	Ing. Hugo Leonel Tiul Valenzuela
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

HONORABLE TRIBUNAL EXAMINADOR

En cumplimiento con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

DESCRIPCIÓN DE LAS MANERAS DE EMPLEAR LA EXTENSIÓN AVANZADA SIMD EN LAS ARQUITECTURAS ARMV7-A Y ARMV8-A DE LOS PROCESADORES ARM

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería Mecánica Eléctrica, con fecha 2 de septiembre de 2020.



Manuel Estuardo Osorio Chamam

Guatemala 25 de febrero 2022

Ingeniero
Julio César Solares Peñate
Coordinador del Área de Electrónica
Escuela de Ingeniería Mecánica Eléctrica
Facultad de Ingeniería, USAC.

Apreciable Ingeniero Solares,

Me permito dar aprobación al trabajo de graduación titulado ¨**Descripción de las maneras de emplear la extensión avanzada SIMD en las arquitecturas ARMV7-A y ARMV8-A de los procesadores ARM**¨, del señor **Manuel Estuardo Osorio Chamam**, por considerar que cumple con los requisitos establecidos.

Por tanto, el autor de este trabajo de graduación y, yo, como su asesora, nos hacemos responsables por el contenido y conclusiones de este.

Sin otro particular, me es grato saludarle.

Atentamente,

Inga. Ingrid Rodríguez de Loukota
Colegiada 5,356
Asesora

Ingrid Rodríguez de Loukota
Ingeniera en Electrónica
colegiado 5356

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERIA

Guatemala, 21 de marzo de 2022

Señor director
Armando Alonso Rivera Carrillo
Escuela de Ingeniería Mecánica Eléctrica
Facultad de Ingeniería, USAC

Estimado Señor director:

Por este medio me permito dar aprobación al Trabajo de Graduación titulado: **DESCRIPCIÓN DE LAS MANERAS DE EMPLEAR LA EXTENSIÓN AVANZADA SIMD EN LAS ARQUITECTURAS ARMV7-A Y ARMV8-A DE LOS PROCESADORES ARM,** desarrollado por el estudiante **Manuel Estuardo Osorio Chamam**, ya que considero que cumple con los requisitos establecidos.

Sin otro particular, aprovecho la oportunidad para saludarlo.

Atentamente,

ID Y ENSEÑAD A TODOS

Ing. Julio César Solares Peñate
Coordinador de Electrónica

ESCUELA DE
INGENIERÍA MECÁNICA ELÉCTRICA
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

REF. EIME 37.2022.

El Director de la Escuela de Ingeniería Mecánica Eléctrica, después de conocer el dictamen del Asesor, con el Visto Bueno del Coordinador de Área , al trabajo de Graduación del estudiante Manuel Estuardo Osorio Chamam: DESCRIPCIÓN DE LAS MANERAS DE EMPLEAR LA EXTENSIÓN AVANZADA SIMD EN LAS ARQUITECTURAS ARMV7-A Y ARMV8-A DE LOS PROCESADORES ARM, procede a la autorización del mismo.

Ing. Armando Alonso Rivera Carrillo

Guatemala, 3 de mayo de 2022.

Decanato
Facultad de Ingeniería
24189101- 24189102
secretariadecanato@ingenieria.usac.edu.gt

LNG.DECANATO.OI.304.2022

La Decana de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Ingeniería Mecánica Eléctrica, al Trabajo de Graduación titulado: **DESCRIPCIÓN DE LAS MANERAS DE EMPLEAR LA EXTENSIÓN AVANZADA SIMD EN LAS ARQUITECTURAS ARMV7-A Y ARMV8-A DE LOS PROCESADORES ARM,** presentado por: **Manuel Estuardo Osorio Chamam,** después de haber culminado las revisiones previas bajo la responsabilidad de las instancias correspondientes, autoriza la impresión del mismo.

IMPRÍMASE:

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DECANA
FACULTAD DE INGENIERÍA

Inga. Aurelia Anabela Cordova Estrada
Decana

Guatemala, abril de 2022

AACE/gaoc

ACTO QUE DEDICO A:

Mis amigos y compañeros de la Universidad	Axl Galicia, Félix Herrera, Fernando Montenegro, Jorge Paredes, Heber García, entre otros. Por su ayuda incondicional durante mi tiempo en la facultad. Sin su apoyo, la universidad hubiera sido más difícil.
Mi familia	Brenda Chamam, Alejandro Osorio y Chaim Osorio; por alentarme durante el transcurso de mi carrera.
Karla de León	Por estar a mi lado.

AGRADECIMIENTOS A:

Departamento de Matemática	Por confiar en mi persona.
Inga. Ingrid Rodríguez	Por su apoyo tanto como catedrática como asesora.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

TABLAS

LISTA DE SÍMBOLOS

Símbolo	Significado
/	División
f(x)	Función en términos de la variable x.
=	Igualdad
→	Implica
∞	Infinito
<=	Menor o igual que
<	Menor que
*****	Multiplicación
!=	No igual
0x	Número hexadecimal
√	Raíz cuadrada

GLOSARIO

AARCH32	Estado de ejecución de 32 bits de la arquitectura ARMV8-A de los procesadores ARM.
AARCH64	Estado de ejecución de 64 bits de la arquitectura ARMV8-A de los procesadores ARM.
ABI	Application Binary Interface.
ALU	*Arithmetic logic unit.*
APSR	*Application program status register.*
Armasm	*ARM assembler,* ensamblador de la arquitectura ARM.
Arreglo	Conjunto de elementos de datos.
Binario	Sistema de numeración que utiliza dos cifras.
Bit	Medida de la cantidad de información.
CISC	*Complex instruction set computer.*
Concatenar	Acción de unir dos elementos.

CPSR	*Current program status register.*
ELR	*Exception link register.*
FPCR	*Floating point control register.*
FPSCR	*Floating point status and control register.*
FPSR	*Floating point status register.*
Gas	*GNU assembler,* ensamblador utilizado por GNU.
GCC	GNU Compiler Collection.
GNU	GNU's Not Unix! Sistema operativo que consiste de software libre.
IEEE	Institute of Electrical and Electronics Engineers.
ISA	*Instruction set architecture.*
LLVM	Conjunto de tecnologías para el desarrollo de compiladores.
LPAE	*Large physical address extension.*
LR	*Link register.*
Matriz	Arreglo bidimensional de datos.

MIMD	*Multiple instruction, multiple data.*
MISD	*Multiple instruction, single data.*
Mnemónico	Tipo de lenguaje de programación de bajo nivel. El mnemotécnico representa instrucciones básicas.
NaN	*Not a number.*
NEON	Alias de la unidad avanzada SIMD, una extensión de los procesadores ARM.
PC	*Program counter*.
Permutación	Acción de variar el orden de un conjunto de datos o elementos.
Polinomio	Suma que consiste en el producto de elementos variables y constantes.
Puntero	Es un valor que consiste en la dirección de memoria en la que se almacena otro elemento.
QNaN	*Quiet not a number.*
RISC	*Reduced instruction set computer.*
SCR	*Secure configuration register.*
SIMD	*Single instruction, multiple data*.

Sintaxis	Regla que define la secuencia correcta de los elementos pertenecientes a un lenguaje de programación.
SISD	*Single instruction single data.*
SNaN	*Signaling not a number.*
SP	*Stack pointer.*
SPSR	*Saved program status register.*
SVE	*Scalable vector extension.*
Thumb	Conjunto de instrucciones de los procesadores ARM, codificadas a 16 bits.
VFP	*Vector floating point.*

RESUMEN

Existen tres métodos principales para utilizar la unidad avanzada SIMD en los procesadores ARM: mediante el uso del lenguaje ensamblador, utilizando técnicas de programación en alto nivel para que el compilador utilice las instrucciones de ensamblador o mediante las funciones intrínsecas que ARM proporciona. Estos métodos se desarrollan en un capítulo individual, los cuales proporcionan una base para el uso de la unidad avanzada SIMD.

Se inicia con un capítulo de conceptos fundamentales, cuyo fin es dar a conocer la estructura interna de la arquitectura ARMV7-A y ARMV8-A. Seguido, se presenta la unidad avanzada SIMD con el fin de conocer la forma en que trabaja la unidad, los registros internos y las formas de dividirlos, el conjunto de instrucciones de la unidad y los tipos de datos que se pueden utilizar para las operaciones.

El tercer capítulo se trata de las optimizaciones que se pueden realizar con los compiladores. Inicia con una descripción de todas las opciones de optimización existentes en los compiladores más utilizados seguido de técnicas de programación para facilitar al compilador realizar optimizaciones al código.

Finalmente se da una descripción de las funciones intrínsecas que se pueden utilizar para la unidad avanzada SIMD. Esta descripción incluye tanto la estructura de estas funciones como las construcciones que se pueden realizar y como estas construcciones corresponden a instrucciones en ensamblador.

OBJETIVOS

General

Realizar una descripción sobre los diferentes métodos de utilizar la extensión avanzada SIMD en las arquitecturas ARMv7-A y ARMv8-A de los procesadores ARM.

Específicos

1. Explicar las características, los registros internos y las instrucciones de la extensión avanzada SIMD para su uso en lenguaje ensamblador.

2. Explicar las convenciones de programación en lenguaje de alto nivel para que el compilador sea capaz de inferir y optimizar el uso de la extensión avanzada SIMD.

3. Describir las extensiones de lenguaje C de ARM para realizar un código portable y que utilice los recursos de la extensión avanzada SIMD.

INTRODUCCIÓN

El procesamiento digital de señales es un área que se ha desarrollado rápidamente en los últimos años. Comúnmente los dispositivos especializados al procesamiento constan de dos partes principales: un procesador convencional y un núcleo para el procesamiento digital de señales. Esto se debe a la diferencia que existe entre un procesador para un sistema computacional y un procesador dedicado al procesamiento digital de señales. Ambas partes del dispositivo son esenciales ya que se necesita utilizar sistemas operativos, almacenar información e interactuar con otros dispositivos como también la capacidad de procesar datos en tiempo real. La separación de estas partes implica conocer a detalle las especificaciones de ambos procesadores, junto con sus instrucciones individuales, y lidiar con los problemas que surjan de la interacción entre ambos.

La compañía ARM Holdings lanzó para su arquitectura una extensión, llamada unidad avanzada SIMD, que permite realizar operaciones en paralelo. Lo novedoso de esta extensión es que se encuentra incorporada dentro del núcleo. La extensión cuenta con recursos individuales y sus propias instrucciones dedicadas. El uso efectivo de la unidad avanzada SIMD requiere el conocimiento de la forma en que esta almacena y extrae la información, los registros internos y el conjunto de instrucciones que utiliza. Cuando se necesita un nivel de optimización considerable, es usual que se realice en un lenguaje de bajo nivel ya que los compiladores presentan problemas para optimizar el código que se ejecuta en el procesador en paralelo.

1. MARCO TEÓRICO

1.1. Sistema binario

Es un sistema de numeración el cual posee únicamente dos cifras, un 0 y un 1. Los números representados en un sistema binario consisten en una serie de coeficientes cuyos valores dependen de la posición individual de cada coeficiente dada por la distancia de estos a un punto base y un signo al principio de estos para indicar si el número es negativo o positivo.

Figura 1. **Partes que componen un número binario**

$$-c_4c_3c_2c_1c_0.c_{-1}c_{-2}c_{-3}$$

Fuente: elaboración propia, realizado con Texstudio.

Se puede determinar el valor decimal del coeficiente multiplicándolo por una función exponencial cuya base es 2 (la cantidad de cifras) y cuyo exponente es la posición del coeficiente. La suma de todos estos valores da como resultado el valor que representa en el sistema decimal. Ver figura 2.

Figura 2. **Conversión de un número binario a uno decimal**

$$-101.1_2 \implies -\left(1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1}\right) \implies -5.5$$

Fuente: elaboración propia, realizado con Texstudio.

"La cantidad de información que un número puede transmitir depende de la cantidad de dígitos que posee. La forma más básica de información es aquel número que solo posee un coeficiente, y a este se le conoce como bit."[1] Las agrupaciones de bits mostradas en la tabla I, comúnmente se les da los nombres específicos.

Tabla I. **Nombres de agrupaciones de bits**

Cantidad de bits	Nombre
8	Byte
16	Media palabra
32	Palabra
64	Palabra doble
128	Palabra cuádruple

Fuente: elaboración propia, realizado con Microsoft Word.

Al bit que representa el coeficiente con mayor posición se le conoce como bit más significativo. Similarmente al bit con el coeficiente de menor posición en toda la serie se le conoce como bit menos significativo.

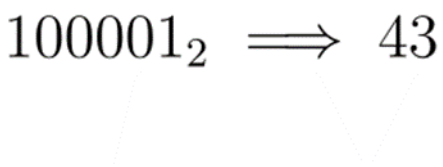
[1] HOHL, William; HINDS, Christopher. *Arm assembly language fundamentals and techniques.* p. 33.

El sistema binario es ampliamente utilizado en electrónica debido a que es fácil representar los números binarios mediante niveles de voltaje o corriente. Para que la representación de los números binarios sea coherente en diversos dispositivos, existen diversos estándares para representar diferentes tipos de números.

1.1.1. Números enteros sin signo

Esta representación omite tanto el punto base, el cual se asume que se asigna al final de los coeficientes, y el signo, que se asume positivo. El resultado de omitir estos elementos es la representación de todos los números enteros positivos. Ver figura 3.

Figura 3. **Representación de un número entero**



Fuente: elaboración propia, realizado con Texstudio.

La cantidad de números que se pueden representar, si se están utilizando n bits, se calcula utilizando una función exponencial de base 2 con exponente n. Por ejemplo, si se tienen 8 bits, se pueden representar hasta 128 números, es decir los números desde el 0 hasta el 127.

1.1.2. Números enteros con signo

Existen diversas maneras de representar el conjunto de números enteros en su totalidad. La forma más fácil es utilizando un bit, usualmente el último, para representar el signo del número. Comúnmente si el último bit es cero, implica un signo positivo y si este es un uno, implica un signo negativo. A esta representación se le conoce como signo con magnitud, ver figura 4.

Figura 4. **Representación signo con magnitud**

$$c_s c_3 c_2 c_1 c_0 \implies (-1)^{c_s} \left(c_3 * 2^3 + c_2 * 2^2 + c_1 * 2^1 + c_0 * 2^0\right)$$

Fuente: elaboración propia, realizado con Texstudio.

Otra forma de representarlos es utilizando la llamada representación en exceso k. Esta especifica que a cada número codificado como entero sin signo se le tiene que restar un número k. Así el cero representado como entero sin signo es equivalente a representar el número –k en esta notación, ver figura 5.

Figura 5. **Representación exceso k**

$$c_3 c_2 c_1 c_0 \implies c_3 * 2^3 + c_2 * 2^2 + c_1 * 2^1 + c_0 * 2^0 - k$$

Fuente: elaboración propia, realizado con Texstudio.

Una de las representaciones más utilizadas para representar a los números enteros con signo es la llamada complemento a dos. Esta representación parte de buscar un número negativo representado por una serie

de coeficientes que sumado al mismo número positivo de como resultado cero. “El resultado es que los coeficientes del número negativo son el producto de invertir los coeficientes del número positivo (cambiar cada coeficiente de 0 a 1 o viceversa) y sumarle la unidad.”[2] A este proceso se le conoce como obtener el complemento a dos de un número. Esta representación es muy útil ya que se pueden utilizar las mismas reglas para realizar operaciones aritméticas, ver figura 6.

Figura 6. **Proceso de obtener el complemento a dos de un número**

$$\begin{aligned} & 01111110_2 \implies +126 \\ Inversión \quad & 10000001_2 \\ Adición \quad & 10000010_2 \implies -126 \end{aligned}$$

Fuente: elaboración propia, realizado con Texstudio.

1.1.3. Números reales

La manera más fácil de representar los números reales es utilizando una posición ya establecida para el punto base. Así, se pueden especificar la cantidad de bits que se utilizarán para la parte entera del número y para la parte fraccionaria. Estos pueden ser sin signo o con signo dependiendo de lo deseado. A esta representación se le conoce como punto fijo ya que el punto base siempre se encontrará en la misma posición.

[2] VALVANO, Jonathan. *Embedded systems: introduction to arm cortex-m microcontrollers.* p. 42.

Una forma más práctica es utilizando una representación conocida como punto flotante. En esta existe un compromiso entre la magnitud del número y la exactitud de este, debido a que solamente es una aproximación a los números reales. Esta se basa en la notación científica en la que un número se representa como se muestra en la figura 7, donde **s** es el signo, **c** es el coeficiente, **b** es la base y **e** es el exponente. Debido a que en esta notación existen muchas maneras de representar el mismo número, usualmente el coeficiente solamente puede contener un dígito como parte entera, es decir que no puede sobrepasar la base del exponente.

Figura 7. **Números en notación científica**

$$s * c * b^{e}$$

Fuente: elaboración propia, realizado con Texstudio.

1.1.3.1. El estándar IEEE 754

El estándar realizado por el Instituto de Ingenieros Eléctricos y Electrónicos es utilizado para representar los números reales en una computadora. “La forma general de estos es similar a la notación científica y también utiliza los 4 parámetros. Este estándar especifica diferentes codificaciones para una base binaria (de 16, 32, 64 y 128 bits) y para una base decimal (de 64 y 128 bits). A las codificaciones de 16, 32 y 64 bits se les conoce

como números de precisión media, precisión simple y precisión doble, respectivamente.”[3]

El más utilizado es el de 32 bits de base binaria. En esta codificación, un bit es utilizado para el signo, 23 bits para el coeficiente y 8 bits para el exponente. Aunque solo se utilizan 23 bits para el coeficiente, en realidad este es de 24 bits porque el primer bit del coeficiente (la parte entera) está dada implícitamente como un uno. Los bits del exponente son números enteros con exceso de 127. Entonces la forma de un número de punto flotante está dada en la figura 8. Otras codificaciones cuentan con una distribución de bits similares a la de los números de precisión simple.

Figura 8. **Forma de un número binario de precisión simple**

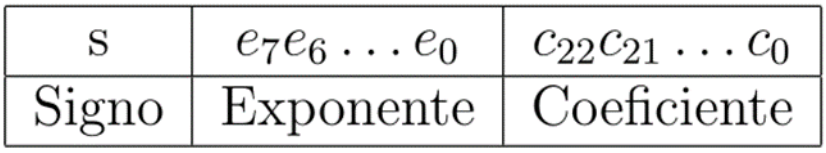
$$(-1)^s * (1.c)_2 * 2^{e-127}$$

Fuente: elaboración propia, realizado con Texstudio.

Además de representar una aproximación hacia los números reales también el estándar especifica otros tipos de valores. Para la representación del cero todos los bits del exponente y del coeficiente tienen que ser igual a cero. El bit del signo indica el signo del cero. Este se utiliza en circunstancias especiales, como la división por cero, pero usualmente ambas representaciones del cero se pueden considerar como iguales.

[3] RAJARAMAN, V. *Ieee standard for floating point numbers*. https://www.ias.ac.in/public/Volumes/reso/021/01/0011-0030.pdf. Consulta: enero de 2022.

Figura 9. **Distribución de bits para números de precisión simple**



Fuente: elaboración propia, realizado con Texstudio.

Para representar una cantidad infinita, todos los bits del exponente tienen que ser igual a uno y los del coeficiente igual a cero. El bit de signo especifica el tipo de cantidad infinita que se representa.

También se pueden representar cantidades más pequeñas que las permitidas por los límites normales del coeficiente. Cuando todos los bits del exponente son cero, pero por lo menos un bit del coeficiente es uno, da como resultado un número denominado subnormal. En esta representación la parte entera del coeficiente (que implícitamente era 1) cambia a un cero. Esto permite representar números que son más cercanos al cero.

Figura 10. **Forma de un número binario subnormal de precisión simple**

$$(-1)^s * (0.c)_2 * 2^{-126}$$

Fuente: elaboración propia, realizado con Texstudio.

El estándar especifica dos representaciones de cantidades indefinidas, llamadas NaN (*not a number*). Existen dos tipos de NaN, uno denominado silencioso (QNaN, *quiet not a number*) y otro de señalización (SNaN, *signaling not a number*). El QNaN se utiliza cuando las operaciones entre dos números

son indefinidas pero que no representan un error en sí, por ejemplo 0/0. El SNaN se utiliza cuando ha ocurrido un error en la operación de un número, por ejemplo, si el resultado de una operación es muy grande para ser representada.

El estándar también define cinco métodos de redondeo cuando se realizan operaciones entre dos números:

- Redondeo hacia abajo: el resultado de una operación siempre será redondeado hacia el número más cercano que no se exceda del resultado.

- Redondeo hacia arriba: el resultado de una operación siempre será redondeado hacia el número más cercano que sea mayor al resultado.

- Redondeo hacia el cero: es una implementación del redondeo hacia abajo para los números positivos y el redondeo hacia arriba para los números negativos. El resultado se sustituye por el número consecutivo que sea más cercano al cero.

- Redondeo hacia el más cercano: el resultado se sustituye por el número que se encuentre más cercano a este. Si la distancia entre el resultado y el número mayor y menor más cercanos a este es la misma, entonces puede haber dos opciones: redondeo hacia un número par y redondeo alejándose del cero.

1.1.4. Polinomios

El sistema binario también puede utilizarse para representar diferentes tipos de información que carecen de sentido en un contexto numérico. Tal es el

caso de los polinomios de una variable, dados de la forma mostrada en la figura 11. Los coeficientes usualmente son limitados a los valores que puede tomar un bit, es decir solamente pueden valer 0 o 1. Esto es de utilidad en aplicaciones como criptografía, corrección de errores, entre otros. Para su codificación, cada coeficiente del polinomio corresponde al bit que se encuentra en la misma posición y las potencias de x vienen dadas implícitamente en la posición del bit.

Figura 11. **Polinomio de una variable x**

$$f(x) = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_1 x + c_0$$

Fuente: elaboración propia, realizado con Texstudio.

Figura 12. **Codificación de un polinomio en un número binario**

$$101110_2 \implies x^5 + x^3 + x^2 + x$$

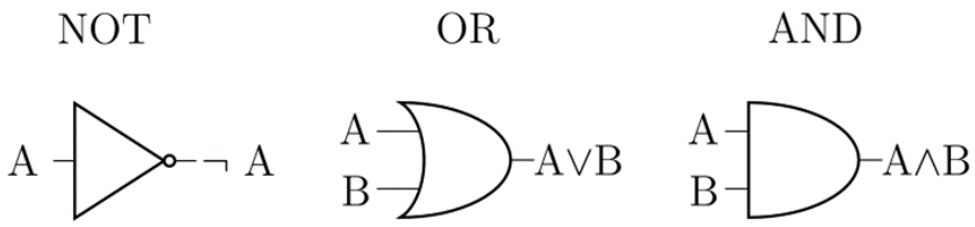
Fuente: elaboración propia, realizado con Texstudio.

1.2. Lógica binaria

Es una manera de describir las operaciones lógicas utilizando álgebra de Boole. En esta, cada valor lógico es representado como una relación uno a uno hacia los valores de un bit en el sistema binario. Usualmente el valor de verdadero es representado con un bit cuyo valor es 1 y el valor de falso con un

bit cuyo valor es 0. Esto es útil porque proposiciones con las operaciones básicas lógicas se pueden emplear físicamente utilizando un sistema binario y circuitos electrónicos. A este tipo de dispositivos se les conoce como compuertas lógicas. Las operaciones lógicas básicas se muestran en la figura 13 junto con los nombres utilizados para las compuertas que realizan estas operaciones.

Figura 13. **Operaciones lógicas básicas**



Fuente: elaboración propia, realizado con Texstudio.

1.3. Aritmética modular

A grandes rasgos y en términos de computadoras se puede pensar en la aritmética modular como aquella donde los dígitos para la representación de los números se encuentran limitados a algún valor definido. Cuando se desea representar un valor mayor al máximo posible por los dígitos, la representación retorna desde el primer valor posible y viceversa. Por ejemplo, si el resultado de una operación es 130 esta es equivalente al número 2 utilizando una representación de números enteros de 8 bits, porque solamente permite representar 128 números diferentes. En general, a la cantidad máxima de números posibles que se pueden representar se le conoce como módulo y en sistemas de computación estos dependen de la cantidad de bits utilizados. Este

comportamiento es común en computadoras debido a que los bits empleados para la información se encuentran limitados físicamente.

1.4. Aritmética de saturación

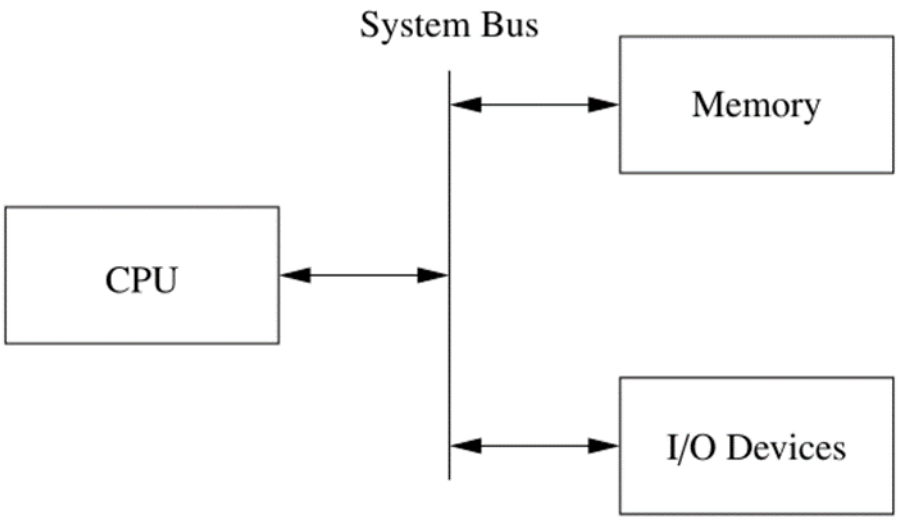
Es un tipo de aritmética cuyos extremos están limitados por un valor máximo y un valor mínimo. Por ejemplo, en cualquier operación cuyo valor exceda 127, si se emplean 8 bits, el resultado sería 127 independientemente de la operación. En términos de la información que contienen los números, este comportamiento provoca un menor error respecto al valor verdadero de la operación. Por esto algunos procesadores implementan la aritmética de saturación, ya que usualmente se desconoce si cierta operación rebasará los límites de la cantidad de bits utilizados.

1.5. Sistemas de computación

En su forma más básica, un sistema de computación puede considerarse que contiene tres componentes principales. “Una unidad central de procesamiento, una memoria y dispositivos de entrada y salida. Todos estos dispositivos se encuentran interconectados por un bus cuya función es transmitir instrucciones, datos y direcciones.”[4]

[4] PYEATT, Larry; UGHETTA, William. *Arm 64-bit assembly language.* p. 1.

Figura 14. **Representación básica de un sistema de computación**



Fuente: PYEATT, Larry. *ARM 64-bit Assembly Language*. p.1.

La unidad central de procesamiento se encarga de realizar todas las operaciones del sistema. La unidad está compuesta por registros, la unidad aritmética lógica y la unidad de control. Los registros sirven para almacenar datos que serán utilizados por el procesador. Estos datos pueden ser direcciones de memoria, operandos, resultados de operaciones, estados del procesador, entre otros. Normalmente, el sistema solo puede realizar operaciones sobre los registros y no sobre la memoria. La unidad aritmética lógica contiene todas las operaciones que se pueden realizar entre conjuntos de bits. La entrada de datos de esta unidad está ligada a registros que pueden contener información que proviene de la memoria o los dispositivos externos. Además de los datos, también es necesario especificar a la unidad la operación a realizar con estos. Por último, la unidad de control recibe las operaciones, denominadas instrucciones, y en base a esto, especifica a los registros y a la unidad aritmética lógica las acciones a realizar para llevar a cabo la instrucción. Una instrucción puede ser de diversos tipos: almacenamiento de datos,

extracción de datos, saltos en el programa, operaciones matemáticas, entre otros.

La memoria almacena las instrucciones y los datos a utilizar por el procesador. Para almacenar información es necesario especificar una dirección, que corresponde al espacio físico donde se almacenará, y un conjunto de bits, que corresponden a la representación de las instrucciones o los datos. Al conjunto de instrucciones almacenadas en la memoria para realizar una tarea indicada se le conoce como programa.

Finalmente, los dispositivos de entrada y salida proporcionan al procesador la interacción entre el medio externo a este. Los dispositivos de entrada envían datos a la unidad central de procesamiento, y esta a su vez realiza lo especificado en el programa para enviar el resultado hacia los dispositivos de salida.

1.5.1. Arquitectura Von Neumann

"Es una implementación de un sistema de computación en el cual la memoria de las instrucciones y los datos es compartida, solamente utilizan el mismo bus para la transmisión de información hacia los demás componentes del sistema."[5] Esta arquitectura posee cierta desventaja ya que no puede realizar una transferencia de instrucciones y datos al mismo tiempo, siendo esto usualmente lo que limita el rendimiento del procesador.

[5] ARM, Ltd. Ltd. *General: harvard vs von neumann architectures. https://developer.arm.com/documentation/ka002816/latest.* Consulta: enero 2022.

1.5.2. Arquitectura Harvard

En esta arquitectura, “la memoria utilizada para el almacenamiento de instrucciones se encuentra separada de la memoria utilizada para el almacenamiento de datos.”[6] Además de estar físicamente separadas, ambas memorias también cuentan con un bus independiente que se encuentra conectado al procesador. Debido a que es más compleja que la arquitectura Von Neumann, esta presenta ciertas ventajas, entre estas se encuentra que es posible diseñar diferentes tipos de memorias que se adecúen a la necesidad de la información y que permite al procesador la transmisión de la siguiente instrucción mientras se ejecuta la instrucción actual.

1.5.3. Arquitectura Harvard Modificada

Se puede pensar en esta arquitectura como una mezcla de las arquitecturas Harvard y Von Neumann. “El sistema almacena las instrucciones y los datos en una misma memoria, pero puede acceder a estas de manera individual. Esto permite más flexibilidad que la arquitectura Von Neumann manteniendo las ventajas de la arquitectura Harvard. En este sistema no existe una diferencia entre instrucciones y datos. Un conjunto de bits almacenados en la memoria puede ser interpretado por la unidad de control como una instrucción y al mismo tiempo como un dato.”[7] Modificar un programa se reduce a almacenar datos en la memoria, algo que el mismo procesador puede realizar con las instrucciones indicadas.

---

[6] ARM, Ltd. Ltd. *General: harvard vs von neumann architectures. https://developer.arm.com/documentation/ka002816/latest.* Consulta: enero 2022.

[7] LEDIN, Jim. *Modern computer architecture and organization: learn x86, arm, and risc-v.* p. 175.

1.5.4. Clasificación por la cantidad de instrucciones y datos

Los sistemas de computación también se suelen clasificar por el número de instrucciones que ejecutan concurrentemente y la cantidad de datos que operan. Se pueden especificar cuatro diferentes tipos de arquitecturas:

- SISD (*single instruction, single data*): El procesador solamente puede realizar una instrucción a la vez sobre un conjunto de datos (generalmente dos). Este es el sistema de computación más simple y funciona de forma secuencial ya que realiza una instrucción a la vez.

- SIMD (*single instruction, multiple data*): El procesador puede realizar una instrucción sobre múltiples conjuntos de datos simultáneamente. Esto permite simplificar las instrucciones repetidas en un programa para que se realicen al mismo tiempo.

- MISD (*multiple instruction, single data*): El sistema posee múltiples procesadores que pueden realizar instrucciones diferentes sobre un mismo conjunto de datos a la vez. Las aplicaciones de este tipo de sistema son muy especializadas debido a que todos los procesadores comparten la misma unidad de memoria.

- MIMD (*multiple instruction, multiple data*): El sistema posee múltiples procesadores y cada uno actúa sobre un conjunto de datos individuales. Cada uno de los procesadores cuenta con su unidad de memoria individual. Se puede pensar en estos sistemas como un conjunto de unidades SISD y SIMD. Por este hecho, es el sistema más potente de las cuatro clasificaciones.

1.5.5. Conjunto de instrucciones

A todas las instrucciones que el sistema de computación puede realizar, se le conoce como conjunto de instrucciones. Usualmente estas se encuentran conformadas por un operador base y varios modificadores. El operador le permite seleccionar al procesador una operación en específico de todas las posibles. Los modificadores alteran el comportamiento de la operación base dependiendo de lo especificado. El conjunto de instrucciones de un procesador se puede clasificar de varias formas que dependen de la naturaleza de las instrucciones.

1.5.5.1. RISC (*reduced instruction set computer*)

Un conjunto de instrucciones reducido se caracteriza porque las instrucciones solamente realizan operaciones básicas, una a la vez. Para realizar una operación más compleja es necesario dividirla en un conjunto de pasos más simples. Usualmente en estos sistemas el tiempo de ejecución de cada instrucción es el mismo.

1.5.5.2. CISC (*complex instruction set computer*)

En un conjunto de instrucciones complejo, las instrucciones especifican usualmente varias operaciones a realizar. Una única instrucción, por ejemplo, es capaz de especificarle al procesador de realizar una extracción de datos, operarlos y almacenarlos. Debido a la variedad de operaciones, el tiempo en que el procesador ejecuta las instrucciones usualmente es variable.

1.6. Arquitecturas ARM

Es una serie de arquitecturas de sistemas de computación diseñados por la compañía ARM Holdings. “Son sistemas con un conjunto de instrucciones reducido y usualmente con una arquitectura Harvard modificada. Debido a su bajo consumo de energía, su bajo costo y su flexibilidad para efectuar diversas aplicaciones, son ampliamente utilizados en los dispositivos móviles. La compañía se encarga de licenciar sus diseños a otras compañías para que fabriquen sus propios dispositivos.”[8]

A partir de la séptima versión de la arquitectura, se ha dividido en tres perfiles enfocados hacia diferentes segmentos del mercado.

El perfil A se enfoca en aplicaciones de alto nivel que requieren grandes cantidades de memoria y alta potencia de procesamiento. Usualmente se pueden encontrar en dispositivos que comprenden desde teléfonos móviles hasta servidores. Soportan sistemas operativos de escritorio y contienen unidades adicionales para los requerimientos de estos, como procesamiento de gráficas y operaciones con punto flotante. Es común que estos procesadores contengan múltiples núcleos por lo que su consumo de energía también es más elevado comparado con las otras opciones.

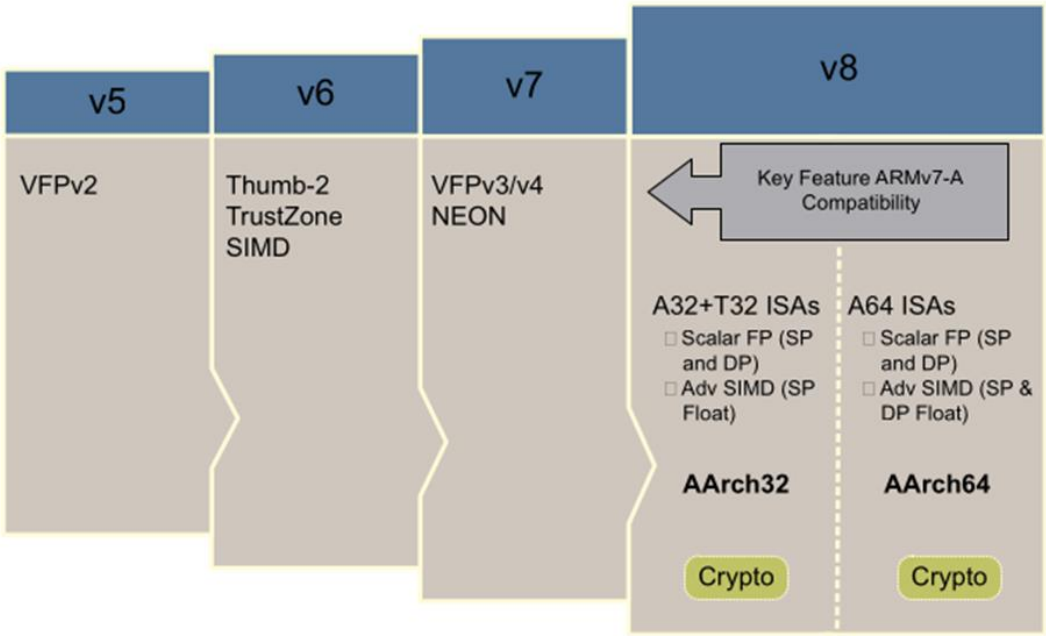
El perfil R está diseñado para aplicaciones donde el tiempo de ejecución es crítico. Es similar al perfil A, pero con módulos adicionales para aumentar la tolerancia a fallos. El tiempo de ejecución de cada instrucción es determinista y no existen retardos impredecibles. Este tipo de arquitectura es de utilidad en dispositivos de seguridad como en el ámbito de medicina, robótica, aviación,

[8] LANGBRIDGE, James. *Professional embedded arm development.* p. 6.

entre otros. Usualmente se necesita utilizar algún tipo de sistema operativo en estos dispositivos, por lo que también cuentan con la memoria necesaria.

Finalmente, el perfil M está enfocado hacia los microcontroladores. Las características deseadas de estos dispositivos son un bajo costo, tamaño reducido y bajo consumo de potencia. Son utilizados para realizar trabajos en específico, por lo que no requieren de una gran capacidad de memoria y tampoco se suele utilizar un sistema operativo para estos. El éxito de los dispositivos basados en este perfil se debe a su flexibilidad, ya que se pueden encontrar desde los aparatos electrodomésticos más comunes hasta en aplicaciones industriales.

Figura 15. **Desarrollo de la arquitectura ARM a lo largo de sus versiones**



Fuente: ARM Ltd. *ARM Cortex-A Series Programmer's Guide for ARMv8-A*. p. 22.

El lanzamiento de la octava versión de la arquitectura representa un cambio drástico. Enfocada principalmente al perfil A, el objetivo es aumentar las capacidades del sistema para fomentar su uso en computadoras de escritorio y servidores. Para ello cuenta con un nuevo conjunto de instrucciones con la capacidad de realizar operaciones de 64 bits. También cuenta con extensiones para realizar operaciones en paralelo de diversos conjuntos de datos. Para evitar una separación entre versiones, el sistema se puede ejecutar en un estado que permite la compatibilidad con la versión séptima de la arquitectura y también realiza mejoras a esta. Los perfiles R y M no poseen la capacidad de realizar operaciones de 64 bits, limitándose a ejecutar en el estado de compatibilidad de 32 bits.

1.6.1. ARMV7-A

Es una arquitectura de 32 bits cuyas implementaciones han sido ampliamente utilizadas en los dispositivos móviles. Es la primera versión de la arquitectura en introducir la unidad NEON.

1.6.1.1. Conjuntos de instrucciones

La arquitectura soporta principalmente dos conjuntos de instrucciones: ARM y Thumb. Ambos conjuntos de instrucciones solamente pueden realizar operaciones directamente en los registros del procesador y, por lo tanto, tienen instrucciones dedicadas para el almacenamiento y la extracción de datos. El procesador puede trabajar con ambos conjuntos de instrucciones libremente. Aunque la funcionalidad de ambos conjuntos es similar, la codificación de ambas es diferente.

El conjunto de instrucciones ARM se encuentra codificado en 32 bits de longitud. Permite ejecutar la mayoría de sus instrucciones de forma condicional dependiendo si los valores cumplen ciertos parámetros. Debido a que todas las instrucciones contienen la misma longitud, la decodificación de las instrucciones es simple. Sin embargo, esto también provoca que un programa ocupe grandes cantidades de memoria aun cuando no sean necesarias todas las funciones del conjunto de instrucciones.

El conjunto Thumb funciona como contraparte al conjunto ARM. Las instrucciones en este conjunto se encuentran codificadas en 16 bits de longitud. Este permite escribir un programa más compacto, pero realiza ciertos sacrificios en las opciones que se pueden utilizar en las instrucciones. La ejecución condicional se limita a la instrucción de saltos en el programa y pocas instrucciones pueden acceder a todos los registros del procesador. La decodificación de estas instrucciones es más compleja debido a su longitud. Una extensión al conjunto de instrucciones original proporciona algunas instrucciones de 32 bits. Esta extensión se llama Thumb-2 y su objetivo es proporcionar la funcionalidad del conjunto de instrucciones ARM, pero reduciendo la densidad del código. Algunas de las nuevas instrucciones se encuentran en ambas codificaciones, 16 y 32 bits, y se puede escoger cualquiera dependiendo de la función a realizar. También la mayoría de las instrucciones se han expandido para permitir su ejecución condicional.

1.6.1.2. Modos del procesador

La arquitectura define nueve modos de ejecución del procesador. Dependiendo del procesador, algunos modos son opcionales y dependen de las extensiones a implementar. Cada modo de ejecución permite ciertos privilegios y restringe acceso a ciertas partes del procesador. Junto con los modos de

ejecución, también se define un nivel de privilegio que permite acceder a los recursos del sistema. Los modos de ejecución son:

- Modo de usuario (USR, *user mode*): en este modo se ejecutan usualmente todas las aplicaciones. Su nivel de privilegio es el más bajo (PL0, *privilege level 0*) y solamente se puede acceder a los recursos del sistema y a la parte de la memoria que tengan el mismo nivel de privilegio. Las aplicaciones se ejecutan en este modo para restringir el uso de los recursos del sistema y evitar modificaciones a partes críticas. Solamente se puede cambiar el modo cuando sucede una excepción como un reinicio, una interrupción, fallos en el acceso de la memoria, entre otros.

- Modo de sistema (SYS, *system mode*): es el modo de ejecución donde los sistemas operativos usualmente se encuentran. Está a un nivel de privilegio más alto que el modo de usuario (PL1, *privilege level* 1) y ambos comparten los registros. No se puede acceder a este modo por alguna excepción.

- Modo de supervisor (SVC, *supervisor mode*): este modo funciona como un intermediario entre el modo de sistema y el usuario. Sirve para que el usuario pueda solicitar realizar operaciones con un nivel de privilegio más alto o algún recurso del sistema operativo. Para realizar la solicitud existe una instrucción dedicada cuyo argumento queda a interpretación del sistema operativo. Al realizar un reinicio del sistema, el procesador entra automáticamente a este modo.

- Modo de abortar (ABT, *abort mode*): el sistema entra a este modo cuando ocurre una falla al acceso de las instrucciones o de la

información. Dependiendo del tipo de falla, el sistema puede obtener la dirección donde ocurrió el problema y corregirlo. Si no se puede obtener la dirección donde ocurrió este problema, el sistema también es capaz de terminar la aplicación. Cuando ocurre una falla con el acceso de las instrucciones, el procesador entra a este modo si intenta ejecutar dicha instrucción.

- Modo indefinido (UND, *undefined mode*): sirve cuando suceden errores relacionados con las instrucciones. Normalmente suceden cuando se intenta ejecutar una instrucción indefinida. Se encuentra en el modo de privilegio 1 (PL1).

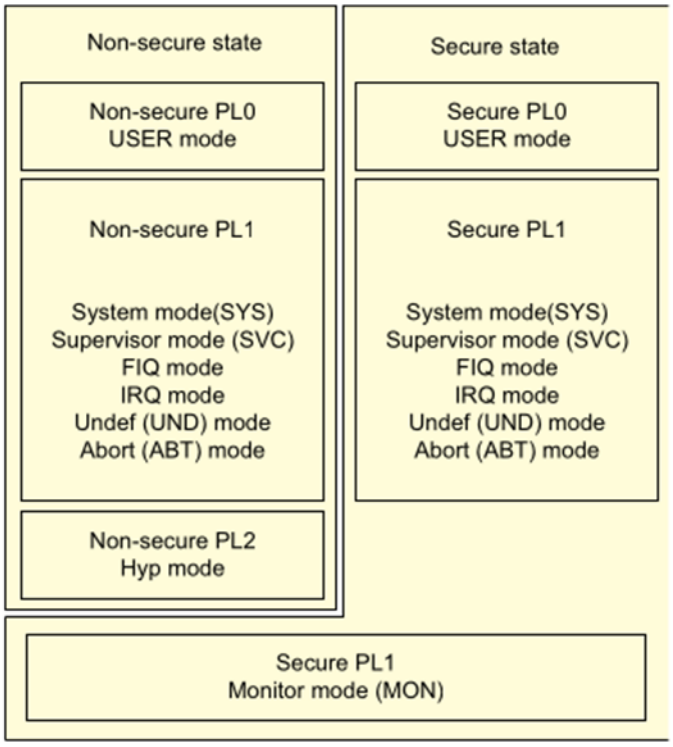
- Modo de interrupción rápida (FIQ, *fast interrupt request*): el procesador ingresa a este modo cuando sucede un pedido de interrupción rápida. Este tipo de interrupción se caracteriza por su baja latencia. Generalmente solamente se utiliza una fuente de interrupción para asegurarse de que se ejecute lo más rápido posible. Contiene ocho registros dedicados para evitar el almacenamiento de la información actual en otros registros. Este modo comparte el nivel de privilegio con el modo de sistema.

- Modo de interrupción (IRQ, *interrupt request*): es similar al modo de interrupción rápida solo que el retardo para estas es mayor debido a que el sistema se asegura de guardar toda la información del programa actual antes de ejecutar la interrupción. Las interrupciones en este modo tienen menor prioridad que las del modo FIQ, pero el nivel de privilegio es el mismo.

- Modo de hipervisor (HYP, *hyp mode*): este modo se incluye como parte de una extensión de virtualización. Permite al sistema ejecutar varios sistemas operativos al mismo tiempo. Debido a que en este modo se controlan los sistemas operativos para que estos puedan ejecutarse en el mismo procesador, tiene un privilegio mayor que el modo de sistema (PL2).

- Modo monitor (MON, monitor mode): un modo que solo se encuentra disponible cuando se implementan las extensiones de seguridad. Se utiliza principalmente como un método para cambiar entre un estado seguro a uno no seguro o viceversa. Usualmente los sistemas operativos se ejecutan en un estado no seguro, mientras que programas que realizan operaciones más sensibles se ejecutan en un estado seguro.

Figura 16. **Jerarquía de los modos del procesador**



Fuente: ARM Ltd. *ARM Cortex-A Series Programmer's Guide*. p. 39.

1.6.1.3. Registros

Dependiendo de las extensiones implementadas en el procesador, este puede llegar a tener hasta 43 registros para el uso de los programas. En cualquier modo del sistema, el procesador solamente puede acceder a 16 de estos registros. Algunos de estos registros son exclusivos para cada modo y otros son compartidos con el modo de usuario. "Los primeros 13 registros, nombrados R0-12, sirven para almacenar cualquier tipo de información para su uso en el programa. Los registros R13-14 se pueden utilizar para este mismo fin, pero cuentan con funciones dedicadas si se desean utilizar. El registro R13, llamado también SP (*stack pointer*), se utiliza para guardar la dirección del

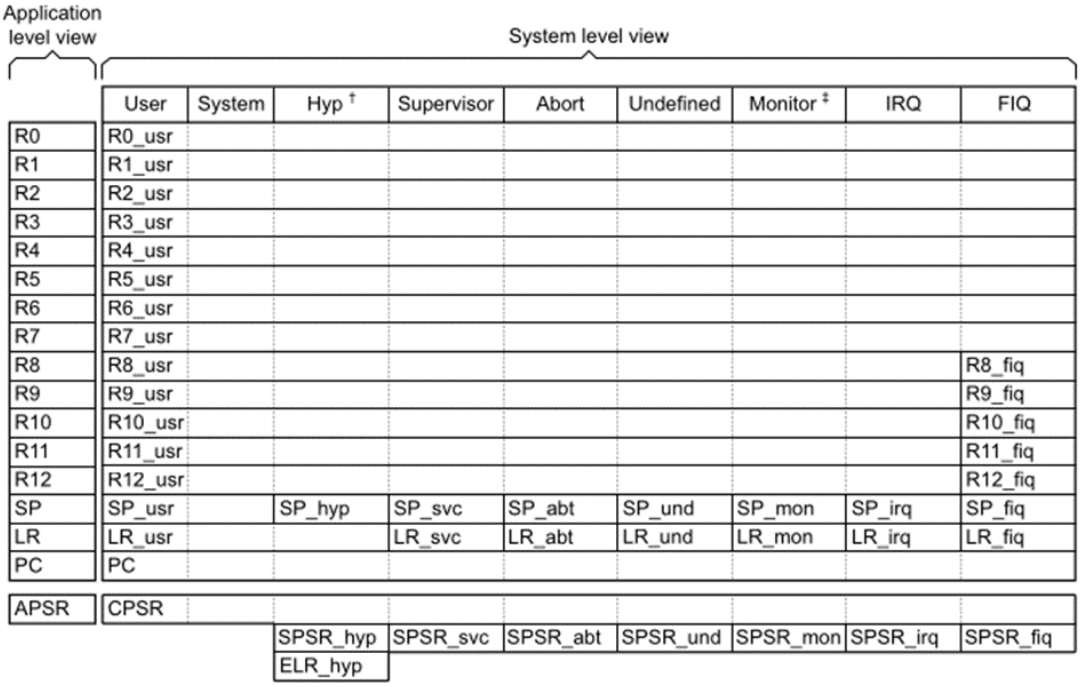
último dato que ha sido almacenado en la pila y el registro R14, llamado también LR (*link register*) se utiliza para almacenar la dirección de retorno cuando se realizan saltos en el programa. El registro R15, llamado PC (*program counter*) almacena la dirección de la instrucción que el sistema se encuentra ejecutando."[9]

Además de los 16 registros, también se cuenta con un registro especial, llamado CPSR (*current program status register*), que almacena información del programa actual. Este registro contiene información del conjunto de instrucciones que se está ejecutando, el modo actual del procesador, la habilitación de interrupciones, parámetros de los resultados de las operaciones efectuadas por la unidad aritmética lógica, entre otros.

En el modo de usuario, el registro se encuentra restringido para que no se puedan efectuar cambios en el sistema. El registro restringido se conoce como APSR (*application program status register*) y contiene únicamente los parámetros de los resultados de las operaciones. En este registro se encuentran los bits N, Z, C, V, Q, y cada uno especifica si de la operación se obtuvo resultado negativo, nulo, acarreo, desbordamiento y saturación, respectivamente. Normalmente no se accede a este registro por separado, sino que se utiliza implícitamente cuando se realizan operaciones condicionales mediante alguna instrucción.

[9] ARM, Ltd. *Arm architecture reference manual armv7-a and armv7-r edition. https://developer.arm.com/documentation/ddi0406/cd. Consulta: enero 2022.*

Tabla II. **Registros de la arquitectura ARMv7-A**



Fuente: ARM Ltd. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition*. p. 1 144.

También se cuenta a su vez con un registro llamado SPSR (*saved program status register*). Éste se utiliza para guardar una copia del CPSR cuando el procesador recibe una excepción y necesita cambiar de modo. Esto permite evaluar los parámetros que el procesador contenía antes de cambiar de modo y que, cuando retorne de la excepción, el programa pueda restaurar el CPSR para continuar con su ejecución normal.

El modo de hipervisor cuenta con un registro especial, llamado ELR (*exception link register*), que se utiliza para almacenar la dirección de retorno de la excepción. El registro R13 lo comparte con el modo de usuario.

1.6.1.4. Extensiones

La arquitectura define el soporte para varias extensiones que aumentan las capacidades del procesador. Estas extensiones quedan a discreción de la implementación del procesador y la mayoría de los aspectos de estas también se pueden modificar. Algunas de estas extensiones son:

- Extensiones de seguridad: permiten al procesador la ejecución de aplicaciones seguras. Incorporan el modo monitor en el procesador. La extensión define un estado seguro y uno no seguro del procesador y divide la memoria para cada estado. Emplea los mecanismos necesarios para evitar que un estado no seguro pueda acceder a la región de memoria que corresponde al estado seguro. El acceso hacia un estado seguro desde uno no seguro se encuentra limitado a pocas excepciones. Emplea un registro, llamado SCR (*secure configuration register*), para especificar el estado de seguridad del sistema y si este se puede modificar. El modo monitor es el intermediario entre estos dos estados.

- Extensiones de memoria: llamada LPAE (*large physical address extension*), permite al sistema acceder a más direcciones de memoria de las permitidas. Utiliza direcciones de 40 bits, lo que permite acceder a 1 TB de memoria a diferencia de los 4 GB que se pueden acceder sin esta extensión.

- Extensiones de virtualización: permiten al sistema ejecutar varios sistemas operativos al mismo tiempo sin que existan dependencias entre ellos. Para esto se crea el modo hipervisor, que funciona en un nivel de privilegio más alto que el de todos los otros modos del sistema. En este modo se ejecuta el programa que se encarga de administrar los sistemas

operativos y los recursos que se le asignan a cada uno de estos. Las interrupciones también se pueden enviar al modo hipervisor y este será el encargado de enviarlas al sistema operativo correspondiente.

- Extensiones de punto flotante: sirven para ayudar al procesador a realizar operaciones con números codificados en punto flotante. En la arquitectura se le conoce como la extensión VFP (*vector floating point*). Dependiendo de la versión a implementar, esta puede soportar operaciones con codificaciones de punto flotante de precisión media, simple y doble. Implementa su propio banco de registros exclusivos para el uso de estas operaciones junto con su registro de control, llamado FPSCR (*floating point status and control register*).

- Extensiones de procesamiento SIMD: llamada por la arquitectura como la extensión avanzada SIMD o también como la unidad NEON, permite al sistema realizar operaciones SIMD. Puede llegar a tener hasta 32 registros de 64 bits, dependiendo de la implementación. Para ahorrar los costos y la potencia utilizada, la unidad NEON comparte los recursos de procesamiento y los registros con la unidad VFP. Soporta operaciones entre números enteros y de punto flotante, aunque no cumple en su totalidad con el estándar IEEE 754.

1.6.2. ARMV8-A

Esta arquitectura presenta un gran cambio respecto a la séptima versión. El fin de la arquitectura es aumentar la potencia de procesamiento mientras se conserva la compatibilidad con sistemas anteriores. Para eso incluye dos estados de ejecución en el sistema: AARCH64 y AARCH32. El estado AARCH64 utiliza direcciones de 64 bits y las instrucciones pueden realizar

operaciones en registros de 64 bits. El estado AARCH32 es el estado de compatibilidad con la séptima versión de la arquitectura, manteniendo las direcciones y los registros de 32 bits. Además de ser compatible, también incluye algunas mejoras del estado AARCH64.

1.6.2.1. Conjuntos de instrucciones

La arquitectura soporta tres conjuntos de instrucciones: A64, A32 y T32. El conjunto de instrucciones que se utiliza depende del estado de ejecución del sistema y de los registros internos de este. El conjunto A64 es exclusivo y no puede trabajar en conjunto con los otros dos. A32 y T32 se pueden utilizar en el mismo programa dependiendo de las funciones que se deseen en el programa.

El conjunto de instrucciones A64 se utiliza cuando el sistema se encuentra en el estado de ejecución AARCH64. La principal característica es que permite al procesador realizar operaciones sobre registros de 64 bits. Las instrucciones presentan una codificación más uniforme y permite mayores saltos en el programa. En general todas las instrucciones que realizan operaciones presentan tres operandos y pueden acceder a todos los registros del sistema para el procesamiento de datos. Cada instrucción tiene dos formas que operan sobre valores de 64 o 32 bits. Sin embargo, todos los valores se guardan en los registros de 64 bits, la instrucción se encarga de realizar los cambios en el registro para almacenar el valor correcto. Todas las instrucciones se encuentran codificadas en 32 bits

Los conjuntos A32 y T32 se utilizan en el estado de ejecución AARCH32. El conjunto A32 es equivalente al conjunto ARM y T32 es equivalente al conjunto Thumb. Ambos presentan las mismas características que en la séptima versión y son generalmente compatibles, aunque presentan

pequeñas diferencias. También se han expandido con nuevas instrucciones de algunas propiedades del conjunto A64.

1.6.2.2. Niveles de excepción

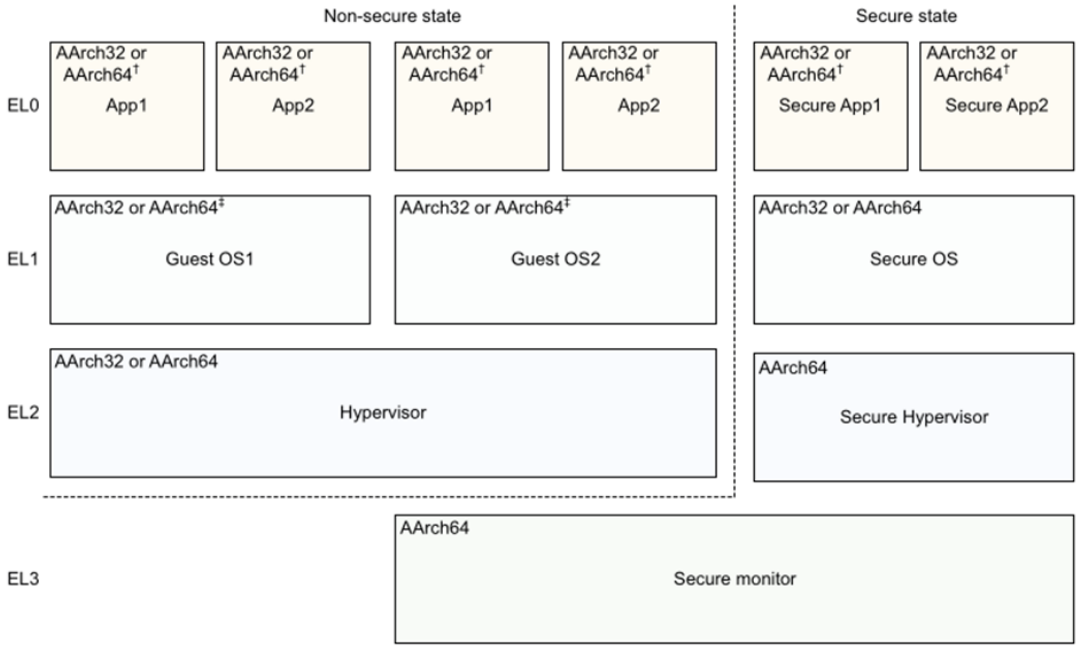
La arquitectura presenta los niveles de excepción (EL, *exception level*) para indicar el privilegio que tiene un programa para acceder a los recursos del sistema. Estos niveles son equivalentes a los niveles de privilegio de la séptima versión, aunque con ciertas diferencias. Los niveles pueden ser EL0, EL1, EL2, EL3, y mientras mayor sea el número del nivel, mayor será el privilegio que este posee. Los niveles de excepción se pueden ejecutar en ambos estados de ejecución del sistema con ciertas restricciones. Estos se pueden ejecutar en un estado seguro o no seguro y son equivalentes a los estados de seguridad de la séptima versión.

En el estado AARCH64, debido a que los modos del procesador ya no son soportados, la arquitectura no proporciona información sobre el tipo de programas que se utilizan en cada nivel de excepción, aunque usualmente se utiliza la misma jerarquía que en la séptima versión: en EL0 se ejecutan aplicaciones del usuario, en EL1 el sistema operativo, en EL2 el hipervisor y en EL3 el monitor de seguridad.

Cuando sucede una excepción, esta se puede ejecutar en el nivel actual o en un nivel superior. Los niveles de excepción, junto con el estado de seguridad, determinan los recursos del sistema que se pueden acceder. Un nivel de excepción solamente se puede ejecutar en el estado AARCH64 si todos los niveles superiores también se encuentran ejecutando el mismo estado. Particularmente si los niveles EL3 y EL2 se encuentran en AARCH64, el

procesador puede ejecutar sistemas operativos y aplicaciones de 32 bits, correspondientes a los niveles EL1 y EL0, utilizando el estado AARCH32.

Figura 17. **Combinaciones de niveles de excepción en ARMv8-A**



Fuente: ARM Ltd. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*. p. 2 319.

El estado AARCH32 continúa utilizando los modos del procesador, pero estos se ejecutan a niveles de excepción determinados. El modo monitor solamente se puede ejecutar en EL3. Los demás modos, exceptuando el de usuario y el de hipervisor, se ejecutan al mismo nivel de excepción. Pueden existir dos posibilidades: que los modos se ejecuten en EL3, y por lo tanto se utilice el modo monitor, o que se ejecuten en EL1 sin el modo monitor. El modo hipervisor y el de usuario se ejecutan en EL2 y EL0 respectivamente. El nivel de privilegio que se utiliza en la séptima versión se sigue utilizando en AARCH32,

pero ligado al nivel de excepción. En particular el primer nivel de privilegio, PL1, se puede ligar a EL1 o EL3 dependiendo del nivel en que se ejecuten los modos.

1.6.2.3. Registros

Los registros del sistema también dependen del estado actual del procesador. Aunque en general los registros para el procesamiento de datos son compartidos entre ambos estados, estos no cumplen las mismas funciones y se nombran de manera diferente.

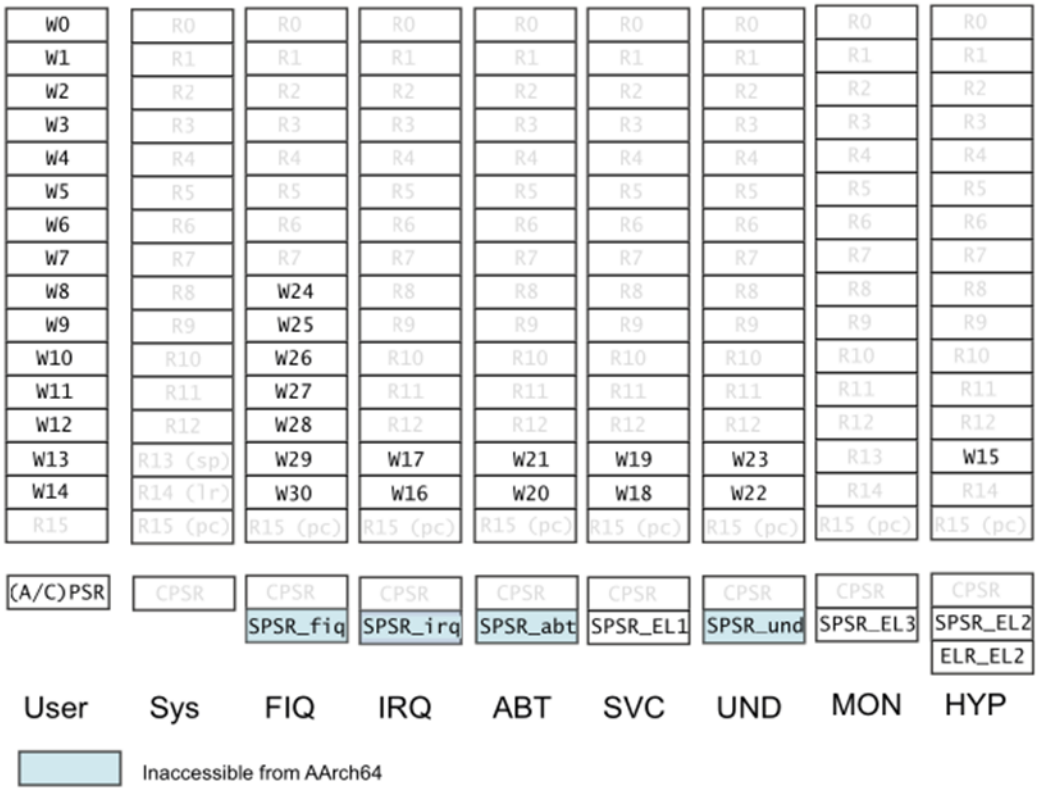
"Existen dos formas de acceder a estos dependiendo si se desean realizar operaciones de 32 bits o 64 bits. Cuando se acceden como X0-30, se utilizan los 64 bits del registro, mientras que cuando se acceden como W0-30, se utilizan los 32 bits menos significativos. Al utilizarlos de la forma W0-30, las instrucciones descartan los bits más significativos o los escriben como cero dependiendo de la situación."[10] Además de estos, el sistema también cuenta con los registros SP, ELR y SPSR, que cumplen con las mismas funciones que en la arquitectura anterior. En vez de contar con registros individuales para cada modo del procesador ahora se dividen por los niveles de excepción, por ejemplo, existe un registro SPSR para los niveles EL1, EL2, EL3, y se utiliza por defecto el registro correspondiente al nivel actual. También cuenta con un registro, llamado ZR (*zero register*), que almacena la constante cero. Esto es de utilidad porque evita utilizar los registros de uso general para almacenar una constante que es ampliamente utilizada en los programas. El registro CPSR no cuenta con su equivalente en el estado AARCH64, en vez de esto se proporcionan mediante campos que se pueden acceder de forma

[10] ARM Ltd. *Arm architecture reference manual armv8, for armv8-a architecture profile.* https://developer.arm.com/documentation/ddi0553/br/. Consulta: enero de 2022

independiente. Al conjunto de todos los campos se le conoce como PSTATE (*processor state*) y contiene información similar a la contenida en el registro CPSR junto con información adicional.

El estado AARCH32 cuenta con los mismos registros que en la séptima versión de la arquitectura y cumplen con la misma función. La mayoría de los registros son compartidos con los 31 registros del estado AARCH64, la diferencia consiste en la función de cada registro y el nombre para acceder a estos. El registro CPSR se presenta como un conjunto de campos del estado del procesador (PSTATE) que incluye todos los campos exclusivos para AARCH32.

Tabla III. **Registros por estado de ejecución de ARMv8-A**



Fuente: ARM Ltd. *ARM Cortex-A Series Programmer's Guide for ARMv8-A.* p. 52.

1.6.2.4. Extensiones

La arquitectura cuenta con extensiones que amplían las funciones del procesador. Entre estas se encuentran las siguientes:

- Extensiones de procesamiento SIMD: la octava versión de la arquitectura amplía la unidad NEON con más registros y mayor funcionalidad en sus instrucciones. Continúa compartiendo los recursos con la unidad de punto flotante. Esta unidad puede ser utilizada en ambos estados de ejecución del sistema. El estado AARCH64 presenta mejoras a las

instrucciones, como el soporte de operaciones con números complejos, multiplicación de matrices, operaciones de criptografía, entre otros. A diferencia de AARCH32, cuando se utiliza la unidad en el estado AARCH64 esta cumple con el estándar IEEE 754 y soporta la elección del método de redondeo, el comportamiento de las cantidades NAN y operaciones con números subnormales. El registro FPSCR se sigue utilizando en AARCH32 mientras que en AARCH64 se divide en dos registros individuales que almacenan información similar: FPCR (*floating point control register*) y FPSR (*floating point status register*).

- Extensión vectorial escalable (SVE, *scalable vector extension*): amplía las capacidades del procesador para operar sobre vectores de mayor tamaño de los que permite la unidad NEON. "El tamaño de los registros sobre los cuales se realizan las operaciones depende de la implementación del procesador y pueden ser desde 128 bits hasta un máximo de 2048 bits, en incrementos de 128 bits. La extensión está enfocada hacia computadoras de alto rendimiento y por esto sirve como un complemento a la unidad NEON y no para sustituirla."[11] Los primeros 128 bits de los registros se comparten con los registros de la unidad NEON para ahorrar espacio. La extensión solamente se puede utilizar en el estado AARCH64 e incluye una ampliación al conjunto de instrucciones A64 para su uso.

1.7. El lenguaje ensamblador y el ensamblador

El lenguaje ensamblador es un lenguaje de programación de bajo nivel utilizado para describir las instrucciones que el procesador ejecutará de una

[11] ARM Ltd. *Arm architecture reference manual supplement the scalable vector extension (sve), for armv8-a.* https://developer.arm.com/documentation/ddi0584/ba/. Consulta: enero de 2022.

forma que sea comprensible. Normalmente el lenguaje ensamblador se realiza de forma que a cada instrucción del procesador le corresponda una instrucción en lenguaje ensamblador. Los diseñadores de cada arquitectura definen la sintaxis a utilizar para las instrucciones, es por esto por lo que cada arquitectura tiene su propio lenguaje ensamblador, sin embargo, existen elementos en común en la mayoría de estos.

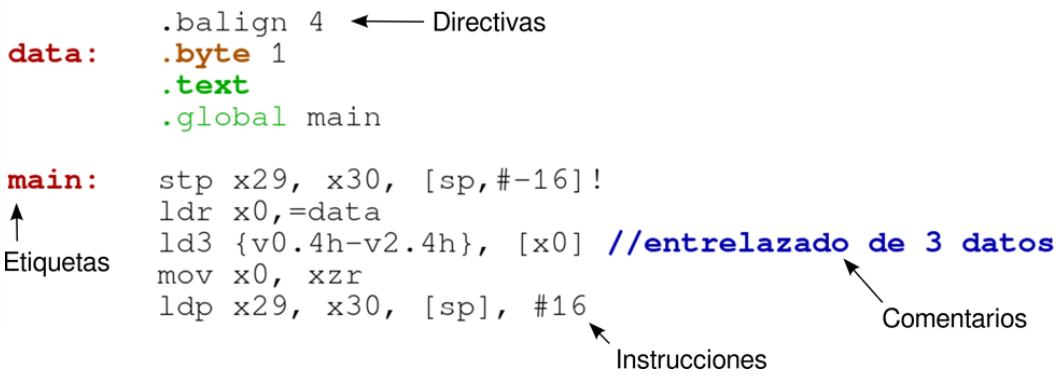
El ensamblador es un programa encargado de transformar las instrucciones escritas en lenguaje ensamblador a un código que pueda ser ejecutado por el procesador (cadenas de bits). Además, también es necesario asignarle a cada instrucción una dirección de memoria donde se almacenará la instrucción, de forma que se ejecuten de acuerdo con el orden establecido por el usuario.

"Independientemente del ensamblador utilizado, la mayoría de los programas contienen cuatro elementos: directivas, etiquetas, instrucciones y comentarios."[12] Las directivas son un conjunto de instrucciones que modifican diversas operaciones del ensamblador como el almacenamiento de memoria, la asignación de símbolos, la separación de secciones, entre otros. Las etiquetas le asignan un nombre a la dirección donde está contenida una instrucción, para poder referenciar esta instrucción sin especificar explícitamente la dirección. Las instrucciones son las operaciones por realizar con el procesador. Generalmente se nombran de acuerdo con una regla mnemotécnica que describe la instrucción a realizar, además también es necesario especificar los datos de entrada y salida, los cuales pueden ser nombres de registros o direcciones de memoria. Finalmente, los comentarios sirven para describir el

[12] PYEATT, Larry. *Modern assembly language programming with the arm processor*. p. 36.

código utilizado, siendo la tarea del ensamblador obviar el texto utilizado para ellos.

Figura 18. **Partes de un programa en lenguaje ensamblador**



Fuente: elaboración propia, realizado con Notepadqq.

Para las arquitecturas ARM se puede utilizar el ensamblador de ARM (armasm) o el ensamblador de GNU (gas), cada ensamblador utiliza una sintaxis diferente para las instrucciones, pero tienen bastantes similitudes entre ellos.

1.8. Compiladores

"Un compilador es un programa capaz de realizar la transformación de un lenguaje de programación a otro."[13] Generalmente son utilizados para traducir un programa realizado en un lenguaje de programación de nivel alto a un código que pueda ser ejecutado por el procesador. El compilador realiza el

[13] AHO, Alfred; LAM, Monica; SETHI, Ravi; ULLMAN, Jeffrey. *Compiladores principios, técnicas y herramientas.* p. 1.

análisis del programa original y genera el programa en el lenguaje de salida a partir de este análisis.

Las directivas del compilador son construcciones que determinan el comportamiento del compilador. El uso de las directivas puede ser muy variado y depende de las opciones del compilador, algunos de sus usos son incluir funciones de librerías comunes, crear una compilación condicional, realizar optimizaciones dependiendo del equipo destino, entre otros.

Algunos compiladores utilizados para la arquitectura ARM son:

- El compilador ARM: es el compilador oficial de la arquitectura ARM. Basado en LLMV y Clang, el compilador soporta todas las arquitecturas junto con las extensiones de estas y todos los procesadores cortex, incluyendo aquellos que se encuentran en desarrollo. Está diseñado para ser compatible con código escrito para el compilador GCC y soporta la sintaxis de ensamblador GNU.

- El compilador GCC (GNU Compiler Collection): es un compilador realizado por el proyecto GNU que soporta diferentes arquitecturas de computadoras y lenguajes de programación y se puede utilizar en varios sistemas operativos. El compilador permite generar código para diferentes arquitecturas o procesadores y seleccionar el nivel de optimización a realizar en el código.

1.8.1. Funciones intrínsecas

Una función intrínseca del compilador es aquella función donde el compilador conoce el comportamiento de la función y es capaz de generar el código de salida directamente. Estas funciones usualmente se utilizan para realizar optimizaciones al código fuente. Uno de sus usos es para generar instrucciones específicas del conjunto de instrucciones que el compilador no es capaz de generar por su cuenta.

2. USO DEL LENGUAJE ENSAMBLADOR EN LA UNIDAD NEON

2.1. La unidad NEON en la arquitectura ARMV7-A

La unidad avanzada SIMD fue introducida en esta arquitectura como un procesador para expandir las instrucciones SIMD de arquitecturas anteriores. Cuenta con un banco de registros independientes en los cuales se pueden ejecutar operaciones de su propio conjunto de instrucciones, el cual se encuentra codificado tanto en formato ARM como en THUMB.

2.1.1. Registros

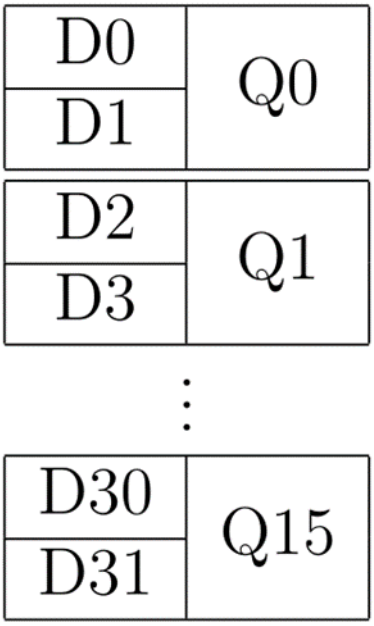
"La unidad NEON cuenta con un conjunto de registros independientes de los utilizados por el procesador, aunque estos registros se encuentran compartidos con la unidad de punto flotante. En su forma más general, la unidad posee 16 registros de 128 bits, nombrados Q0-Q15; o de forma equivalente 32 registros de 64 bits, nombrados D0-31."[14] A pesar de que los registros Q son de 128 bits, no se pueden utilizar para almacenar datos del mismo tamaño que el registro, se utilizan para almacenar un vector que contenga múltiples elementos de un tamaño inferior.

Los registros Q y D se pueden utilizar de forma simultánea, pero hay que tomar en cuenta que cada registro Q se puede dividir en dos registros D. Por

[14] ARM Ltd. *Neon programmer's guide*. https://developer.arm.com/documentation/den0018/latest/. Consulta: enero de 2022.

ejemplo, si se utiliza el registro Q0, esto implica que los registros D0 y D1 ya contienen información. De forma sucesiva se asigna a cada registro Q, dos registros D. Otro aspecto por tomar en cuenta al utilizar los registros es que estos son compartidos con la unidad de punto flotante. Entonces cada registro D se puede dividir en dos registros S (utilizados por la unidad VFP) de 32 bits.

Figura 19. **Composición de los registros de la unidad NEON**

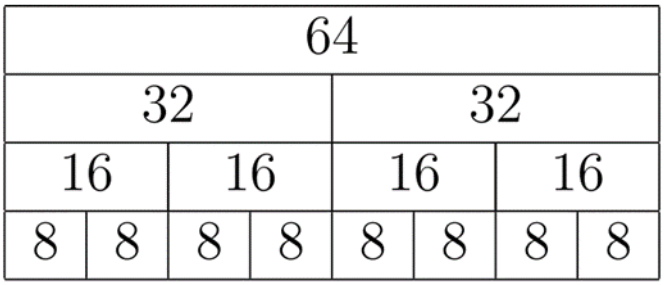


Fuente: elaboración propia, realizado con Texstudio.

2.1.1.1. Registros como vectores

Cada registro de la unidad NEON se puede utilizar como un vector de n elementos del mismo tipo, donde la cantidad de elementos resulta de dividir el registro en n partes iguales. Así, un registro Q puede contener: dos elementos de 64 bits, 4 elementos de 32 bits, 8 elementos de 16 bits y 16 elementos de 8 bits. De forma equivalente un registro D puede contener: un elemento de 64 bits, 2 elementos de 32 bits, 4 elementos de 16 bits y 8 elementos de 8 bits.

Figura 20. **Formas de dividir un registro D**



Fuente: elaboración propia, realizado con Texstudio.

Algunas instrucciones permiten utilizar un elemento individual del vector. Estos elementos se pueden acceder de la forma Qn[x] o Dn[x], donde n es el número del registro y x el elemento a acceder.

2.1.1.2. ABI (Application Binary Interface)

"Para crear funciones que utilicen la unidad NEON y que sean compatibles con diferentes programas, es necesario utilizar los registros siguiendo ciertas convenciones."[15] La ABI especifica el uso de los registros de la siguiente forma:

- Los registros D0-D7 o Q0-Q3 se pueden utilizar para los argumentos de las funciones o para retornar los resultados.

- La información de los registros D8-D15 o Q4-Q7 tiene que ser guardada por la función antes de que se puedan utilizar y se tiene que restablecer la información original después de utilizar los registros.

[15] ARM Ltd. *Application binary interface for the arm architecture - the base atandard.* https://developer.arm.com/documentation/ihi0036/latest. Consulta: enero de 2022.

- La información de los registros D16-D31 o Q8-Q15 tiene que ser guardada antes de utilizar la función y se restablecen los valores después de utilizar la función. Adentro de la función estos registros se consideran volátiles y se pueden utilizar sin alguna restricción.

2.1.2. Conjunto de instrucciones

La unidad NEON posee sus propias instrucciones específicas, las cuales se pueden ejecutar tanto en el conjunto THUMB como en ARM. Ya que la unidad NEON es una extensión de la arquitectura ARM, muchos principios que se utilizan para la programación en ensamblador del procesador se aplican también a la extensión. Por ejemplo, utiliza las mismas extensiones mnemotécnicas para la ejecución condicional y los mismos modos de direccionamiento.

Tabla IV. **Mnemónicos de ejecución condicional**

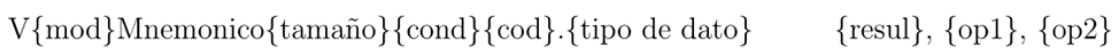
Mnemónico	Significado
EQ	Igual
NE	No igual
GT	Mayor que
GE	Mayor que o igual
LT	Menor que
LE	Menor que o igual

Fuente: elaboración propia, realizado con Microsoft Word.

2.1.2.1. Sintaxis general

La sintaxis de todas las instrucciones de la unidad NEON sigue el formato general mostrado en la figura 21.

Figura 21. **Formato general de las instrucciones NEON**



Fuente: elaboración propia, realizado con Texstudio.

Todas las instrucciones de la unidad NEON inician con una V para distinguirlas de las instrucciones del procesador. Cada instrucción general presenta diferentes variaciones dependiendo del fin deseado. Entre estas variaciones se encuentra el modificador de resultado, el tamaño del resultado, la opción de ejecución condicional, elección de codificación de 16 o 32 bits para el conjunto THUMB y finalmente el tipo de datos a utilizar. Cada instrucción tiene limitaciones específicas sobre el uso de estas variantes, por ejemplo, no todas las instrucciones pueden operar sobre polinomios.

Los datos de entrada y salida pueden ser especificados como los registros de la unidad NEON, registros del procesador, constantes o listas de registros. Normalmente se especifican tres elementos: el resultado, el primer operando y el segundo operando. Dependiendo del tipo de operación de cada instrucción se puede utilizar un tercer operando o solamente uno. Cada una de las instrucciones tiene diferentes limitaciones sobre el uso de cada registro, por ejemplo, no todas las instrucciones pueden acceder a los registros ARM.

2.1.2.2. Modificadores

Las instrucciones permiten realizar modificaciones al resultado de una operación. Entre las posibles modificaciones se encuentran: Q, para utilizar aritmética de saturación; R, para redondear los resultados; D, para duplicar el resultado y H, para dividir el resultado en dos.

2.1.2.3. Tamaños

Los modificadores de tamaño permiten variar la cantidad de bits utilizados para el resultado. Entre estos se encuentran: L, para un resultado que utilice el doble de bits de los operandos; N, para un resultado que utilice la mitad de los bits de los operandos; W, para que el resultado y el primer operando utilicen el doble de bits que el segundo operando.

Tabla V. **Modificadores de tamaño y forma típica de registros**

Tamaño	Registros
Ninguno	Dx, Dy, Dz y Qx, Qy, Qz
L	Qx, Dy, Dz
N	Dx, Qy, Qz
W	Qx, Qy, Dz

Fuente: elaboración propia, realizado con Microsoft Word.

2.1.2.4. Tipos de datos

Los vectores de la unidad NEON pueden contener elementos de diferentes tipos: enteros con signo (S), enteros sin signo (U), enteros sin especificar (I), números de punto flotante (F) y polinomios (P). Los tipos de datos usualmente se especifican con el tipo de dato seguido del tamaño del dato. Cada tipo de elemento posee diferentes tipos de tamaños que son soportados por la unidad NEON, como se muestra en la tabla IV.

Tabla VI. **Tipos de datos soportados por la unidad NEON**

Tamaño de dato	Tipo de dato
.8	.I8 .S8 .U8 .P8
.16	.I16 .S16 .U16 .P16 .F16
.32	.I32 .S32 .U32 .F32
.64	.I64 .S64 .U64

Fuente: elaboración propia, realizado con Microsoft Word.

El tipo de dato P16 solamente es un dato de salida, no se puede utilizar como entrada en una instrucción. El tipo de dato F16 solamente es soportado si se ha implementado la extensión de media precisión. Cuando el tipo de datos es irrelevante para la instrucción, solamente se especifica el tamaño de estos. Algunas instrucciones utilizan diferentes tipos de datos en sus registros. Cuando sucede esto, se pueden añadir los tipos de datos uno seguido después del otro.

2.1.2.5. Listas de registros

Estas listas se utilizan cuando una instrucción realizará operaciones sobre múltiples registros, como en el caso del entrelazado de datos. Las listas se

especifican escribiendo cada registro que estará contenido en la lista separados por comas y encerrándolos en llaves. Hay cierta flexibilidad para la escritura de listas, por ejemplo, si se utilizan registros consecutivos se puede utilizar un guion para especificar la lista ({D0-D3}). También se pueden crear listas con elementos individuales de varios registros ({D0[2], D1[2]}). Cada instrucción tiene restricciones respecto a los registros que pueden aparecer en una lista.

2.1.2.6. Limitaciones de los números en punto flotante

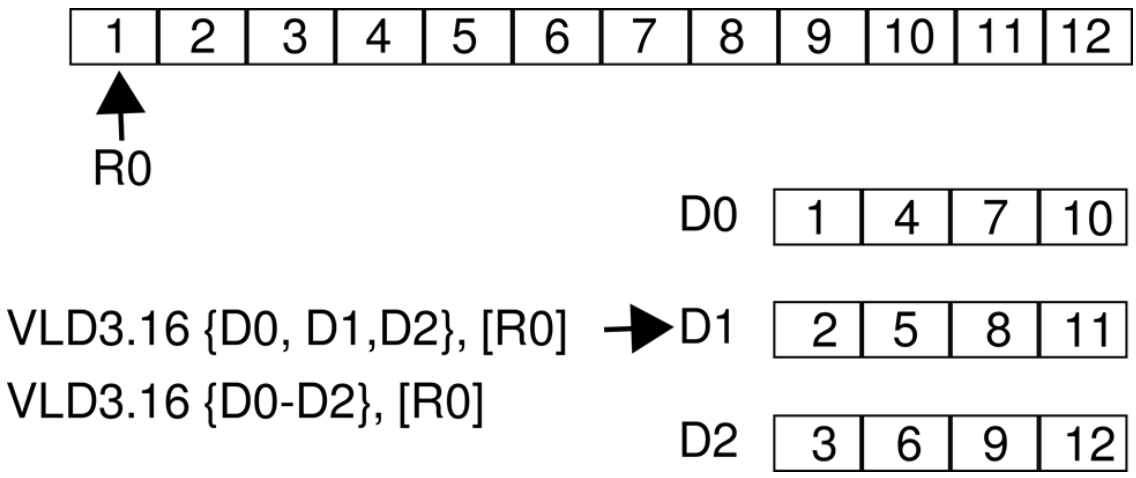
La unidad NEON no cumple con su totalidad con el estándar IEEE 754. La primera característica es que la mayoría de las instrucciones redondean los números subnormales a cero. Solamente un número limitado de instrucciones, generalmente aquellas que realizan el almacenamiento y extracción de datos, pueden utilizar los números subnormales sin redondearlos a cero. Además de esto, la unidad solamente puede realizar operaciones de vectores con números de precisión simple, si se desea operar con valores escalares se puede utilizar la unidad VFP. El redondeo de los números también se encuentra limitado al método de redondeo hacia el más cercano.

2.1.2.7. Almacenamiento y extracción de datos

La instrucción general para el almacenamiento y extracción de datos utilizada por la unidad NEON es VLDn y VSTn, donde n puede ser desde 1 hasta 4 dependiendo del patrón de entrelazado que se utilice. Los modificadores y los tamaños no se utilizan ya que carecen de sentido. Estas instrucciones presentan cierta flexibilidad dependiendo del fin a realizar.

Una gran diferencia que presenta la unidad NEON es que las instrucciones de almacenamiento y extracción de datos pueden entrelazar y desentrelazar a estos para sus operaciones. Esto es conveniente cuando se quieren realizar operaciones sobre ciertos tipos de datos, por ejemplo, para una imagen almacenada en formato RGB es más conveniente realizar un entrelazado para tener un vector que contenga cada color por separado. La instrucción VLD4 representa un entrelazado de cuatro datos y así sucesivamente. Un valor de n igual a 1 significa que no se realizará un entrelazado.

Figura 22. **Proceso de entrelazado de la instrucción VLD3**



Fuente: elaboración propia, realizado con Inkscape.

Las instrucciones permiten especificar hasta 4 registros D como fuente o destino de la operación, pero tienen que ser congruentes con el entrelazado especificado:

- Con n igual a 4 se tiene que especificar una lista de cuatro registros D.

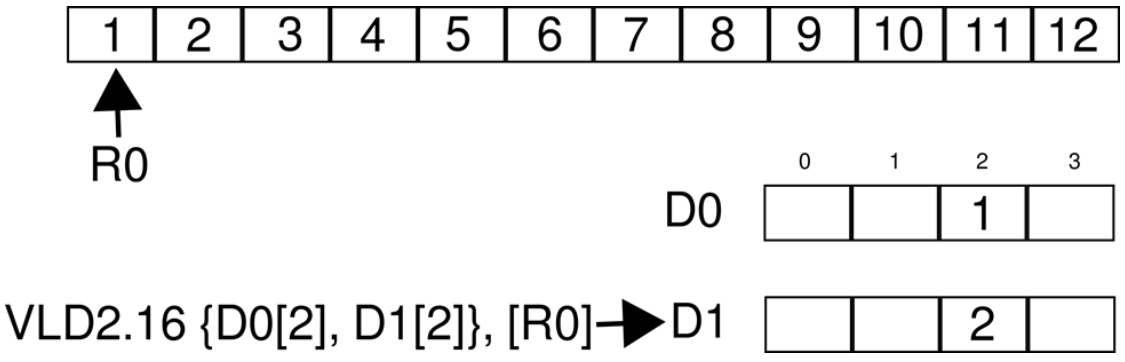
- Con n igual a 3 se tiene que especificar una lista de tres registros D.

- Con n igual a 2 se tiene que especificar una lista de dos o cuatro registros D.

- Con n igual a 1 se puede especificar una lista desde uno hasta cuatro registros D.

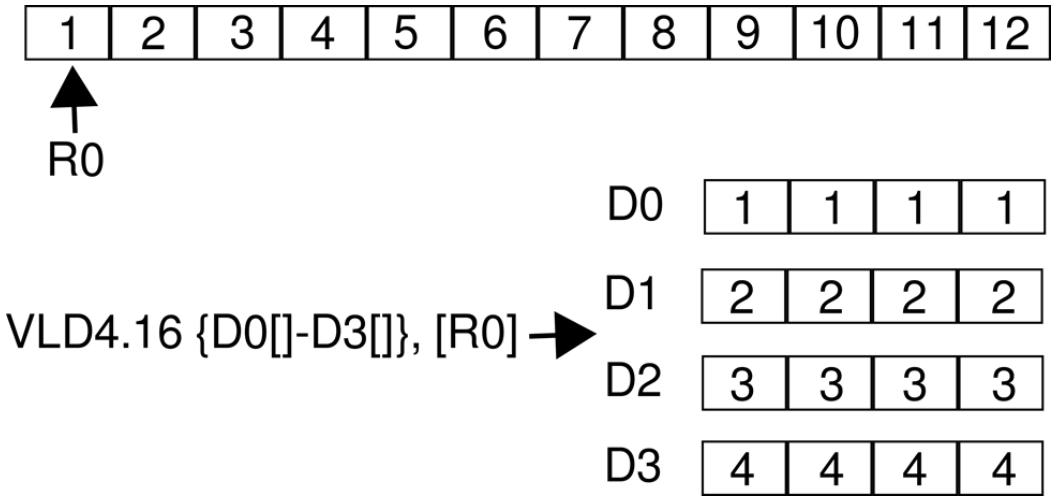
También se puede copiar un elemento individual de memoria a un elemento en específico del vector o viceversa. Para ello se utiliza una lista con el índice del elemento donde se desea almacenar el dato. Si se desea copiar el mismo elemento en todo el vector se utiliza una lista, pero sin especificar el índice del elemento del vector (D2[]). Esta última operación solamente es válida para la extracción de datos.

Figura 23. **Ejemplo de copiar a un elemento en específico del vector**



Fuente: elaboración propia, realizado con Inkscape.

Figura 24. **Ejemplo de copiar un mismo elemento a todo el vector**



Fuente: elaboración propia, realizado con Inkscape.

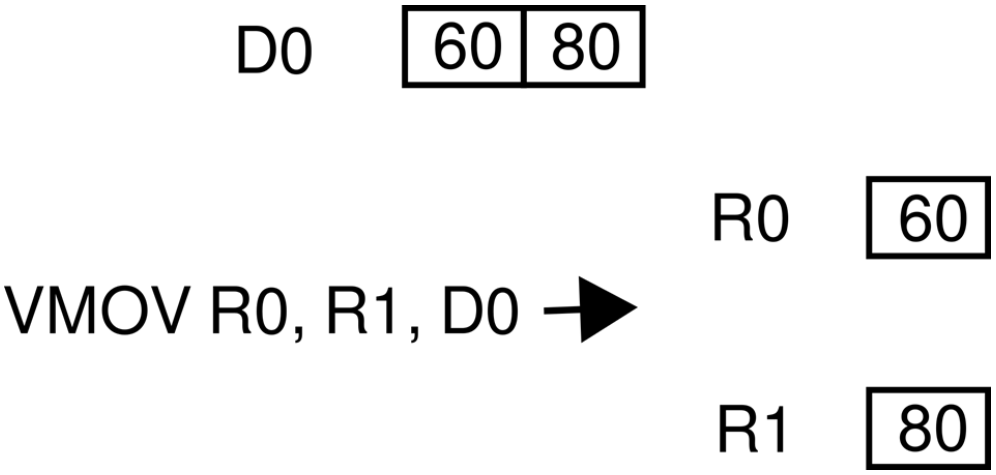
2.1.2.8. Movimiento de datos

La instrucción VMOV permite realizar diversos tipos de movimiento de datos, ya sea de un registro NEON a uno del procesador o viceversa, o entre los mismos registros de la unidad NEON. La primera forma de utilizar la instrucción es especificar un registro de destino (Qn o Dn) y una constante inmediata a cargar en el registro. Esta forma copia la constante en cada elemento del registro.

Si se especifican dos registros, uno de destino y uno de origen, se pueden realizar las siguientes operaciones: el movimiento de datos entre dos registros NEON del mismo tamaño, el movimiento de un registro ARM a un elemento de un vector NEON, y el movimiento de un elemento del vector a un registro ARM. También se pueden mezclar registros Q y D como registros de destino u origen, si se utilizan los modificadores L, N y Q (VMOVL, VMOVN, VQMOVN).

Otra forma de mover datos es especificar dos registros de origen o dos registros de destino. Para el primer caso se especifican dos registros ARM como origen y un registro D como destino, llenando así los 64 bits del vector. Mientras que para el segundo caso se especifican dos registros ARM como destino y un registro D como origen, moviendo así los primeros 32 bits al primer registro ARM y los últimos al segundo registro.

Figura 25. **Mover un registro D a dos registros ARM**

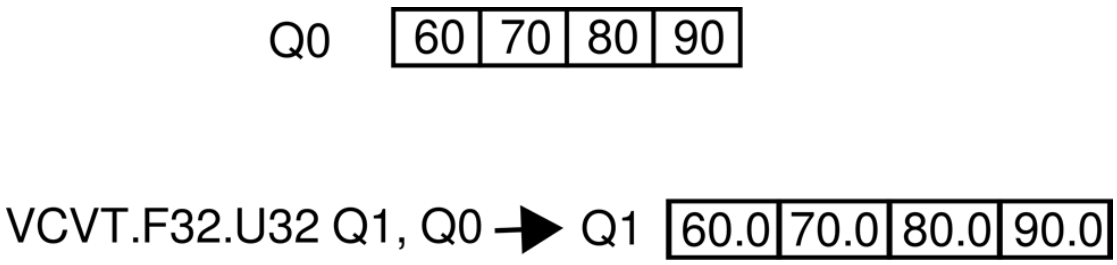


Fuente: elaboración propia, realizado con Inkscape.

2.1.2.9. Conversiones de datos

La unidad permite tres tipos de conversiones de datos: entre números de punto flotante de precisión simple y enteros, entre números de punto flotante de precisión simple y enteros de punto fijo, y entre números de punto flotante de precisión media y simple. La instrucción VCVT realiza todas las conversiones de datos, solamente es necesario especificar los tipos de datos y los registros que contienen estos. Para el caso de conversión de punto fijo también se especifica como el tercer operando el número de bits fraccionarios.

Figura 26. **Ejemplo de conversión de datos U32 a F32**

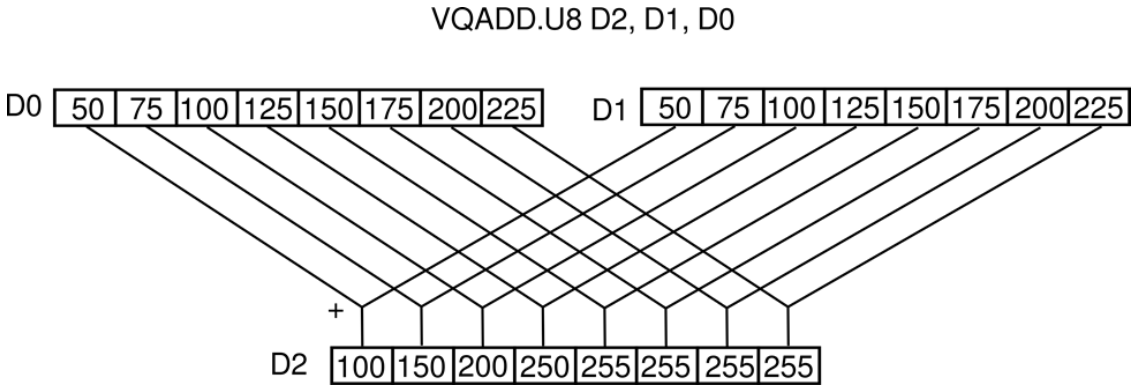


Fuente: elaboración propia, realizado con Inkscape.

2.1.2.10. Operaciones aritméticas

Las operaciones de suma y resta entre vectores se pueden realizar con las instrucciones VADD y VSUB. Los modificadores permitidos para estas instrucciones son L, W y Q. También existe una instrucción especial para utilizar el modificador N: VADDHN y VSUBHN. Esta instrucción realiza la suma entre los vectores y los recorta tomando la mitad superior de los bits resultantes. Si se desea realizar un redondeo se puede agregar el modificador R: VRADDHN y VRSUBHN.

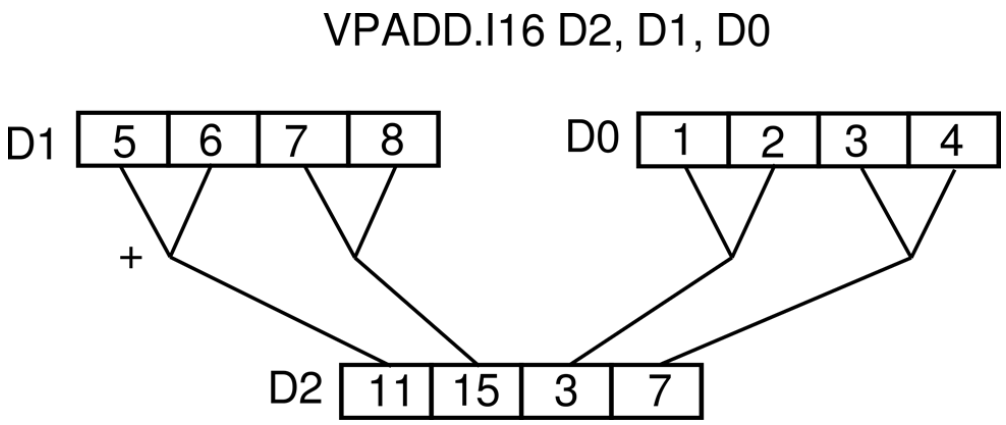
Figura 27. **Ejemplo de suma con saturación**



Fuente: elaboración propia, realizado con Inkscape.

También existen algunas instrucciones especiales que realizan operaciones de suma y resta. Por ejemplo, para realizar una suma o resta y dividir el resultado por la mitad se utiliza la instrucción VHADD o VHSUB, para sumar elementos adyacentes entre dos vectores y guardarlos en un tercer vector se utiliza la instrucción VPADD *(vector pairwise add*), para realizar una resta entre vectores y guardar el valor absoluto de esta operación en otro vector se utiliza la instrucción VABD, entre otros.

Figura 28. **Ejemplo de uso de instrucción VPADD**

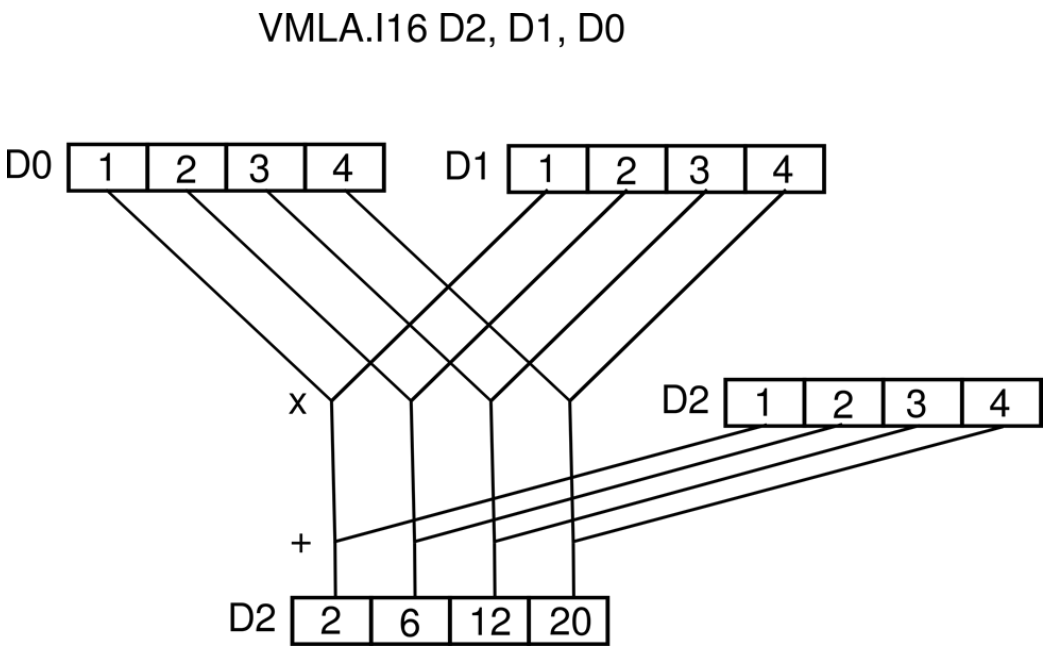


Fuente: elaboración propia, realizado con Inkscape.

Para realizar multiplicación entre vectores se pueden utilizar las instrucciones VMUL (*vector multiply*), VMLA *(vector multiply accumulate*), VMLS (*vector multiply substract*). VMUL realiza la multiplicación entre los elementos del vector correspondiente y los guarda en el vector de destino. VMLA y VMLS realizan la misma operación, pero suman o restan el resultado de la operación a los elementos del vector de destino. Estas instrucciones permiten el uso del modificador L, por ejemplo, VMULL. También se pueden utilizar los modificadores Q y D en conjunto: VQDMULL, VQDMLAL o VQDMLSL.

Cuando se utilizan números de punto flotante, las operaciones de multiplicación y suma o resta realizan el redondeo al finalizar cada operación, es decir realizan dos redondeos por instrucción. Para mitigar los efectos del redondeo existen las instrucciones VFMA (*vector fused multiply accumulate*) y VFMS (*vector fused multiply substract*) que fusionan las operaciones en una misma, realizando el redondeo hasta el final.

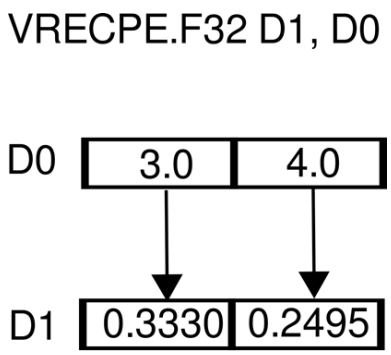
Figura 29. **Ejemplo del uso de la instrucción VMLA**



Fuente: elaboración propia, realizado con Inkscape.

La unidad NEON no cuenta con una instrucción específica para realizar la división entre vectores. Existen dos métodos para realizar la división: el primero consiste en utilizar la unidad VPF, pero esto implica realizar operaciones sobre escalares y no vectores; el segundo método es utilizar la instrucción de multiplicación pero que un operando sea el recíproco del valor que se desea dividir. Para este fin se cuenta con las instrucciones VRECPE *(vector reciprocal estimate*) y VRSQRTE (*vector reciprocal square root estimate*), que realizan, para cada elemento del vector, una estimación del recíproco o una estimación del recíproco de la raíz cuadrada respectivamente.

Figura 30. **Ejemplo del uso de la instrucción VRECPE**



Fuente: elaboración propia, realizado con Inkscape.

Para mejorar la estimación se pueden utilizar las instrucciones VRECPS (*vector reciprocal step*) y VRSQRTS (*vector reciprocal square root step*). Estas instrucciones calculan los pasos del método de Newton-Raphson para estimar los valores recíprocos correspondientes, como se muestra en las tablas VII y VIII. Si se utilizan estas instrucciones junto con VMUL, se puede calcular la siguiente estimación del método de Newton-Raphson a utilizar.

Tabla VII. **Método estimar el recíproco y la raíz del recíproco**

Método de Newton-Raphson	**Converge a**
$x_{n+1} = x_n(2 - dx_n)$	$\frac{1}{d}$
$x_{n+1} = \frac{x_n(3 - dx_n^2)}{2}$	$\frac{1}{\sqrt{d}}$

Fuente: elaboración propia, realizado con Microsoft Word.

Tabla VIII. **Operaciones de las instrucciones VRECPS y VRSQRTS**

Instrucción	Operación
VRECPS dest, op1, op2	$dest = 2 - op1 * op2$
VRSQRTS dest, op1, op2	$dest = \frac{3 - op1 * op2}{2}$

Fuente: elaboración propia, realizado con Microsoft Word.

2.1.2.11. Operaciones lógicas

Las instrucciones que realizan operaciones bit a bit son las siguientes: VAND, VBIC, VEOR, VORN y VORR. Estas instrucciones realizan las operaciones lógicas de AND, AND NOT, XOR, OR NOT y OR respectivamente, entre dos pares de vectores y almacenan el resultado en un tercer vector. Las instrucciones VAND, VBIC, VORN y VORR también permiten como segundo operando un valor inmediato.

Para aplicar un corrimiento por una cantidad variable se puede utilizar la instrucción VSHL. Esta instrucción toma como operandos un vector al que se le desea realizar el corrimiento y un vector con los valores de los corrimientos. La instrucción utiliza el byte menos significativo de todos los elementos del segundo vector para realizar el corrimiento. Si la cantidad es negativa, realiza un corrimiento a la derecha y viceversa. Se pueden utilizar los modificadores R y Q: VRSHL, VQSHL, VQRSHL. Si se desea aplicar un corrimiento por un valor inmediato se puede utilizar la instrucción VSHL para corrimientos a la izquierda y VSHR para corrimientos a la derecha. La instrucción VSHL soporta los modificadores Q y L, mientras que la instrucción VSHR soporta los modificadores R y N.

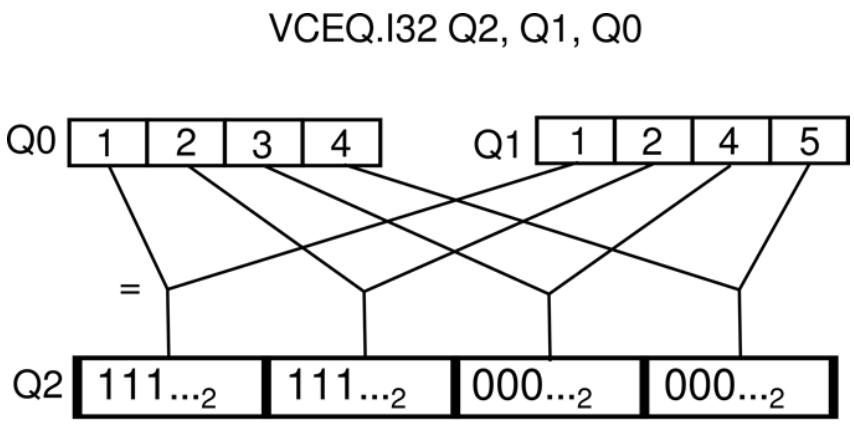
2.1.2.12. Polinomios

La suma y multiplicación de polinomios se puede realizar con las instrucciones de la unidad NEON. Para poder realizar una suma se puede utilizar la instrucción VEOR (*vector bitwise exclusive OR*) que es la operación equivalente para la suma de polinomios. Para la multiplicación se pueden utilizar las instrucciones VMUL y VMULL, especificando el tipo de dato como P8.

2.1.2.13. Operaciones de comparación

Para realizar comparaciones entre dos vectores se pueden utilizar las siguientes instrucciones: VCEQ (*vector compare equal*), VCGE (*vector compare greater than or equal*), VCGT (*vector compare greater than*), VCLE (*vector compare less than or equal*), VCLT (*vector compare less than*). Estas instrucciones realizan la comparación entre cada elemento individual del vector, si esta comparación es verdadera todos los bits del elemento se almacenan como un 1, de lo contrario se almacenan como un 0.

Figura 31. **Ejemplo de la instrucción VCEQ**



Fuente: elaboración propia, realizado con Inkscape.

Se puede agregar un modificador A (*absolute*) para realizar las comparaciones con el valor absoluto: VACGE, VACGT, VACLE, VACLT. Estas instrucciones solamente pueden operar en datos de punto flotante de precisión simple.

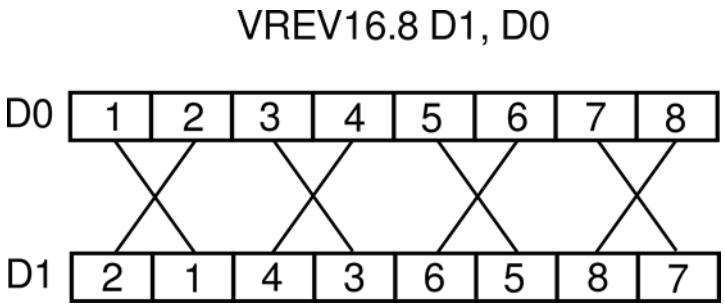
2.1.2.14. Permutación de vectores

La unidad NEON contiene diferentes instrucciones que permiten reordenar el contenido de un vector o crear nuevos vectores a partir de los ya existentes. La instrucción más simple de estas es VSWP, que intercambia el contenido entre dos vectores. Usualmente estas instrucciones consumen muchos recursos, por lo que se recomienda solamente utilizarlas cuando sea necesario, por ejemplo, cuando se esté trabajando con datos que no tengan el mismo formato que las instrucciones.

La instrucción VREVn invierte el orden de los elementos de un vector. Existen tres variantes correspondientes para n igual a 64, 32 y 16. El valor de n

corresponde al número de bits que se agruparan para realizar la inversión. Por ejemplo, VREV32 realiza la inversión, dependiendo del tamaño especificado, cada dos elementos de 16 bits o cada cuatro elementos de 8 bits.

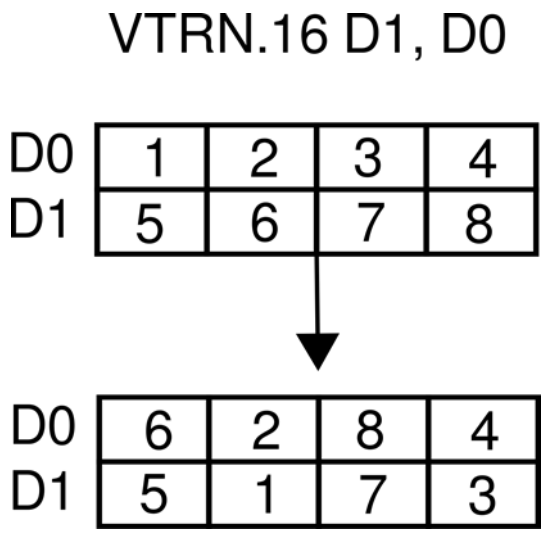
Figura 32. **Ejemplo de uso de la instrucción VREV16**



Fuente: elaboración propia, realizado con Inkscape.

Para obtener la transpuesta de una matriz se puede utilizar la instrucción VTRN. Esta instrucción trata los vectores de entrada como un arreglo de matrices de 2x2 y obtiene la transpuesta de cada matriz. Para obtener transpuestas de matrices más grandes se puede utilizar múltiples veces la instrucción.

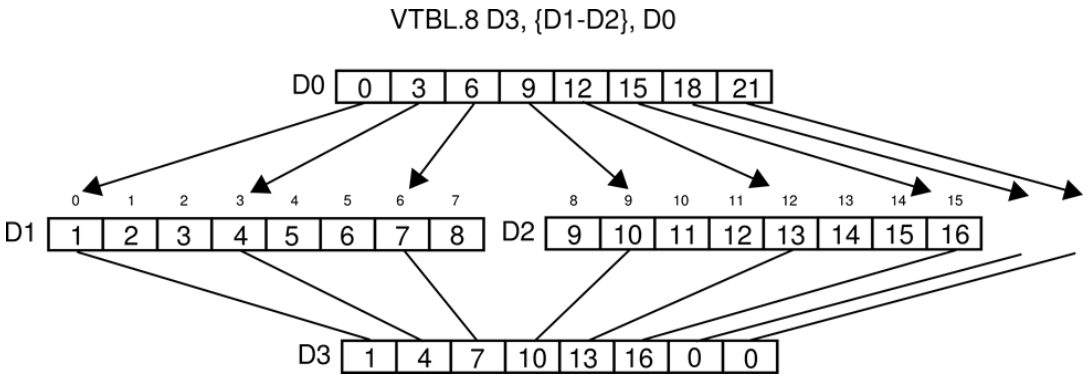
Figura 33. **Ejemplo de uso de la instrucción VTRN**



Fuente: elaboración propia, realizado con Inkscape.

La instrucción VTBL sirve para crear un nuevo vector a partir de una lista de vectores. Toma como datos de entrada una lista de vectores y un vector con los índices de los elementos de la lista a almacenar en el nuevo vector. Si un índice se encuentra fuera de rango la instrucción almacenará un 0 en el elemento correspondiente. Si se desea que no se modifiquen los elementos del vector destino cuando un índice se encuentra fuera de rango, se puede utilizar la instrucción VTBX. Estas instrucciones solamente trabajan con elementos de 8 bits, tanto en la lista de vectores como en el vector de índices.

Figura 34. **Ejemplo de uso de la instrucción VTBL**



Fuente: elaboración propia, realizado con Inkscape.

La instrucción VEXT sirve para concatenar dos vectores. La instrucción utiliza tres valores de entrada: dos vectores y un valor inmediato. La operación extrae la cantidad de elementos inferiores del segundo vector especificados por el valor inmediato y los junta con los elementos superiores del primer vector para almacenarlos en el vector de destino. Cada elemento es tratado como si fuera de 8 bits.

2.2. La unidad NEON en la arquitectura ARMV8-A (AARCH64)

La unidad avanzada SIMD cuenta con mejoras en la arquitectura ARMV8-A. Algunos de los cambios más importantes son que ahora se cuenta con un banco de registros más grande y que la sintaxis de las instrucciones se ha simplificado y ampliado en funcionalidad.

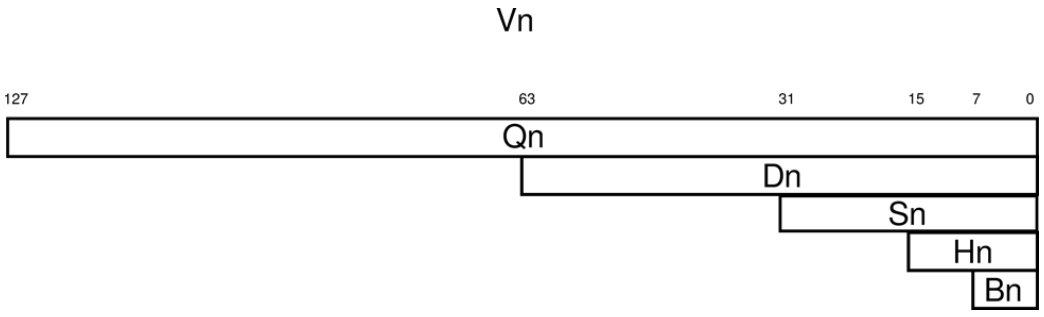
2.2.1. Registros

La unidad NEON en la arquitectura ARMV8-A cuenta con 32 registros de 128 bits cada uno, nombrados de la forma V0 - V31. Al igual que con los

registros de propósito general, el uso de estos depende del estado de ejecución del sistema. En el estado AARCH32 solamente se pueden acceder a los primeros 16 registros, que corresponden a los registros Q0 – Q15 o D0- D31 de ARMV7-A.

En el estado AARCH64 se pueden acceder a cada uno de los 32 registros de diferente forma: como un registro de 128 bits, Q0 - Q31; como un registro de 64 bits, D0 – D31; como un registro de 32 bits, S0 – S31; como un registro de 16 bits, H0 – H31 y como un registro de 8 bits, B0 – B32. Cuando no se utiliza el registro completo, por ejemplo, se accede como D0, se utilizan los bits menos significativos del registro para almacenar la información. Los bits restantes se comportan como los registros generales, es decir se ignoran cuando se realiza la lectura de datos y se almacenan como cero cuando se realizan operaciones. Una diferencia importante con la arquitectura ARMV7-A es que no existe solapamiento entre registros, por ejemplo, si el registro D1 tiene información, se puede utilizar el registro Q0 sin modificar la información de D1.

Figura 35. **Formas de dividir un registro V**



Fuente: elaboración propia, realizado con Inkscape.

2.2.1.1. Registros como vectores

Cada registro de la unidad NEON se puede utilizar como un vector de 128 o 64 bits. Los elementos individuales de cada vector pueden ser de diferentes tamaños, como se muestra en la tabla IX.

Tabla IX. **Forma de nombrar a un vector en AARCH64**

Forma	Nombre
8 bits x 8 elementos	Vn.8B
8 bits x 16 elementos	Vn.16B
16 bits x 4 elementos	Vn.4H
16 bits x 8 elementos	Vn.8H
32 bits x 2 elementos	Vn.2S
32 bits x 4 elementos	Vn.4S
64 bits x 1 elementos	Vn.1D
64 bits x 2 elementos	Vn.2D

Fuente: elaboración propia, realizado con Microsoft Word.

Para especificar el contenido de cada vector se nombran de la forma Vd.T, donde Vd es el registro a utilizar y T es el conjunto de la cantidad de elementos y el tamaño de cada elemento. Para que T represente un vector válido, el tamaño total tiene que ser de 128 o 64 bits. Por ejemplo, V0.16B representa un vector de 128 bits que contiene 16 elementos de 8 bits cada uno, mientras que V0.2S representa un vector de 64 bits que contiene 2 elementos de 32 bits cada uno.

Algunas instrucciones permiten utilizar un elemento individual del vector. Estos elementos se pueden acceder de la forma Vd.Ts[x], donde Ts representa el tamaño del elemento (B, H, S o D) y x el número del elemento a acceder.

2.2.1.2. ABI

La ABI especifica el uso de los registros de la siguiente forma:

- Los registros V0 - V7 se pueden utilizar para los argumentos de las funciones o para retornar los resultados.

- La información de los registros V8 – V15 tiene que ser guardada por la función antes de que se puedan utilizar y se tiene que restablecer la información original después de utilizar los registros.

- La información de los registros V16 – V31 tiene que ser guardada antes de utilizar la función y se restablecen los valores después de utilizar la función. Adentro de la función estos registros se consideran volátiles y se pueden utilizar sin alguna restricción.

2.2.2. Conjunto de instrucciones

La unidad NEON de la arquitectura ARMV8-A se puede utilizar tanto en el estado AARCH32 como AARCH64. En el estado AARCH32 la unidad es compatible con la arquitectura ARMV7-A, utilizando la misma sintaxis de instrucciones, registros, tipos de datos, entre otros. Para el estado AARCH64 el conjunto de instrucciones para la unidad NEON está incorporada en el conjunto A64 presentando ciertas simplificaciones y mejoras a las versiones anteriores.

2.2.2.1. Sintaxis general

Aunque la sintaxis general de las instrucciones de la unidad NEON para el estado AARCH64 sigue manteniendo una estructura similar a las versiones anteriores, estas han sido simplificadas respecto a la arquitectura ARMV7-A.

El prefijo V de todas las instrucciones se ha eliminado, ahora el ensamblador es el encargado de escoger la instrucción adecuada dependiendo de los datos a operar. A la mayoría de las instrucciones se les puede agregar prefijos y sufijos que modifican el resultado de dichas operaciones. Ahora el tipo de dato se especifica como un prefijo en las instrucciones (P, F, entre otros) y el tamaño de los vectores se especifica en el registro individual (V0.4S). Los datos de entrada y salida siguen manteniendo el mismo esquema: pueden ser listas, registros, constantes, entre otros. Las instrucciones siguen utilizando tres elementos de datos: el resultado, el primer operando y el segundo operando. Las modificaciones que se pueden realizar con las instrucciones dependen mucho sobre el tipo de instrucción a utilizar, ya que no todas las modificaciones representan una operación válida.

2.2.2.2. Tipos de datos y prefijos

Los tipos de datos soportados por la unidad NEON son los siguientes:

- Números de punto flotante de 16, 32 y 64 bits (F).
- Enteros con signo y sin signo (S, U).
- Polinomios (P).

Además de estos, también existe el soporte para utilizar números complejos en punto flotante, si se ha implementado la extensión ARMv8.3-CompNum (FC, *floating complex*).

Para especificar el tipo de dato a utilizar, se agrega como un prefijo a la instrucción. Por ejemplo, PMULL especifica la multiplicación entre polinomios. Otros prefijos disponibles son las operaciones con aritmética de saturación. Existen dos prefijos dependiendo si se utilizan elementos con signo (SQ) o sin signo (UQ). Los límites de saturación dependen del tipo de dato utilizado en la instrucción.

Tabla X. **Rango de saturación para diferentes tipos de datos**

Tipo de Dato	Rango de saturación
S8	$-2^7 \leq x < 2^7$
S16	$-2^{15} \leq x < 2^{15}$
S32	$-2^{31} \leq x < 2^{31}$
S64	$-2^{63} \leq x < 2^{63}$
U8	$0 \leq x < 2^8$
U16	$0 \leq x < 2^{16}$
U32	$0 \leq x < 2^{32}$
U64	$0 \leq x < 2^{64}$

Fuente: elaboración propia, realizado con Microsoft Word.

2.2.2.3. Sufijos

El estado AARCH64 también implementa los modificadores de tamaño de ARMv7-A. “El modificador L (Long) opera sobre vectores de 64 bits y produce un vector de 128 bits, duplicando el tamaño de cada elemento del vector. El modificador N (Narrow) opera sobre vectores de 128 bits y produce un vector de 64 bits, esto significa que recorta el tamaño de cada elemento del vector por la

mitad. El modificador W (Wide) opera sobre un vector de 64 bits y otro de 128 bits, y da como resultado un vector de 128 bits."[16]

También se ha agregado un sufijo de 2 para especificar la parte superior de los registros cuando se utilizan los modificadores L, N y W. El modificador L2 toma como entrada los 64 bits más significativos de dos registros y da como resultado un vector de 128 bits. El modificador N2 guarda los elementos en los 64 bits más significativos de un vector. El modificador W2 opera sobre los 64 bits más significativos de un vector y otro de 128 bits, y da como resultado un vector de 128 bits.

Existen dos sufijos para realizar diferentes operaciones: P y V. El sufijo P se utiliza para instrucciones que realizan operaciones en pares de elementos adyacentes de un vector. El sufijo V se utiliza para realizar operaciones sobre todos los elementos del vector.

2.2.2.4. Listas de registros

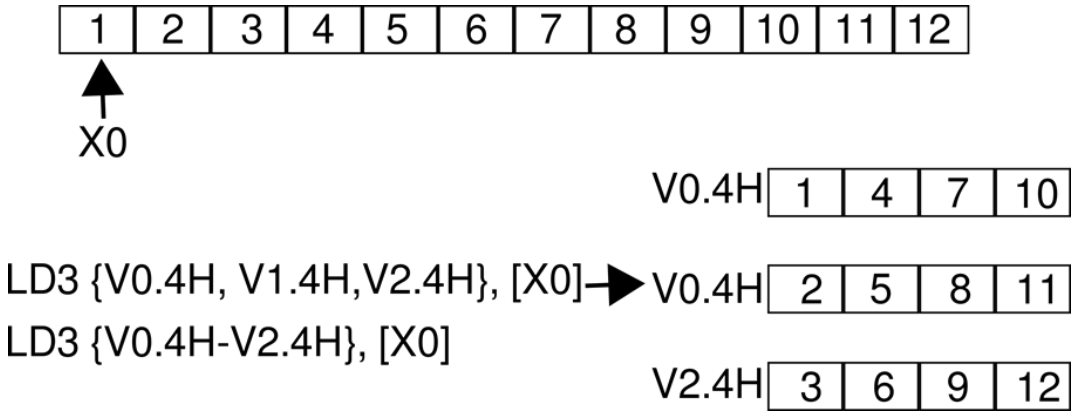
Las listas siguen manteniendo un formato similar: se utiliza un conjunto de registros separados por una coma o un rango de registros separados por un guión. Este conjunto tiene que estar encerrado por llaves ({V0.16B – V3.16B}). Si la instrucción requiere una lista con elementos individuales, se puede especificar el número del elemento a acceder: ({V0.B – V3.B}[2]).

[16] ARM Ltd. *Arm cortex-a series programmer's guide for armv8-a.* https://developer.arm.com/documentation/den0024/latest/. Consulta: enero de 2022.

2.2.2.5. Almacenamiento y extracción de datos

El estado AARCH64 utiliza las mismas instrucciones que la arquitectura ARMV7-A: LDn y STn, con n igual a 1, 2, 3 o 4. Estas instrucciones presentan las mismas características, es decir se puede realizar un entrelazado de datos dependiendo del valor de n, utilizan una lista de registros como vectores a almacenar o extraer los datos y un registro ARM con la dirección en memoria.

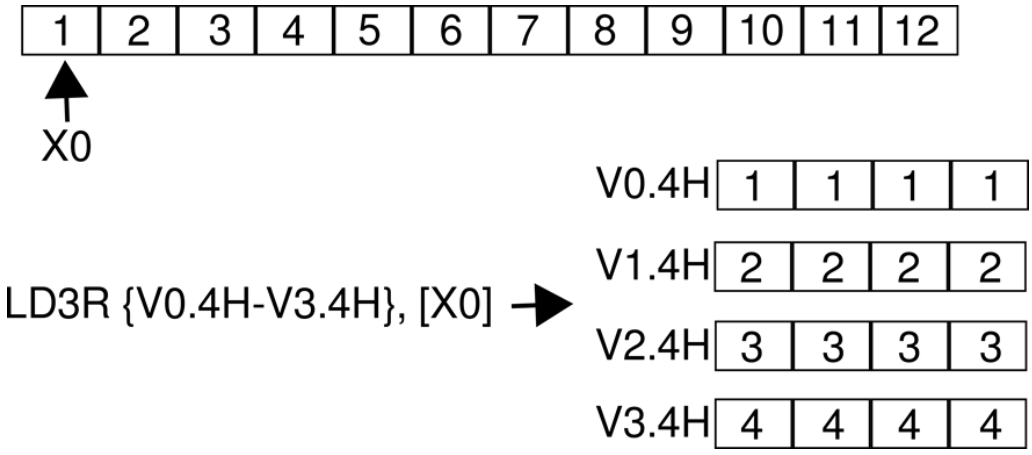
Figura 36. **Ejemplo del uso de la instrucción LDn**



Fuente: elaboración propia, realizado con Inkscape.

De la misma forma si se desea extraer un conjunto de datos a un elemento en específico del vector o viceversa, se puede especificar dicho elemento en la lista. Para copiar un mismo dato a todos los elementos del vector se utiliza la nueva instrucción LDnR, con n igual a 1, 2, 3 o 4.

Figura 37. **Ejemplo del uso de la instrucción LDnR**



Fuente: elaboración propia, realizado con Inkscape.

2.2.2.6. Movimiento de datos

Se utiliza la misma instrucción MOV para copiar datos ya sea de un registro ARM a un registro NEON o para mover elementos de un vector a otro vector. La instrucción permite utilizar los sufijos U, S y F. También existe la instrucción MOVI (*move immediate*) para copiar un valor inmediato a todos los elementos del vector.

2.2.2.7. Conversiones de datos

Para convertir datos enteros o de punto fijo a números en punto flotante, se utiliza la instrucción UCVTF o SCVTF para tipos de datos U o S respectivamente. Solamente es necesario especificar el vector de salida, de entrada y, si se utilizan datos de punto fijo, la cantidad de bits de la parte fraccionaria. El método de redondeo de esta instrucción está especificado por el FPCR.

Para convertir datos de punto flotante a números enteros se utiliza la instrucción FCVTnU o FCVTnS, utilizando cada instrucción si se desean números sin signo o con signo, respectivamente. En estas instrucciones n representa el tipo de redondeo a utilizar, como se especifica en la tabla XI. Las instrucciones que soportan números de punto fijo son FCVTZU y FCVTZS.

Tabla XI. **Redondeos que se pueden utilizar para la instrucción FVCT**

n	Significado
N	Redondeo hacia el más cercano con distancias iguales hacia un número par.
A	Redondeo hacia el más cercano con distancias iguales alejándose del cero.
P	Redondeo hacia arriba.
M	Redondeo hacia abajo.
Z	Redondeo hacia el cero.

Fuente: elaboración propia, realizado con Microsoft Word.

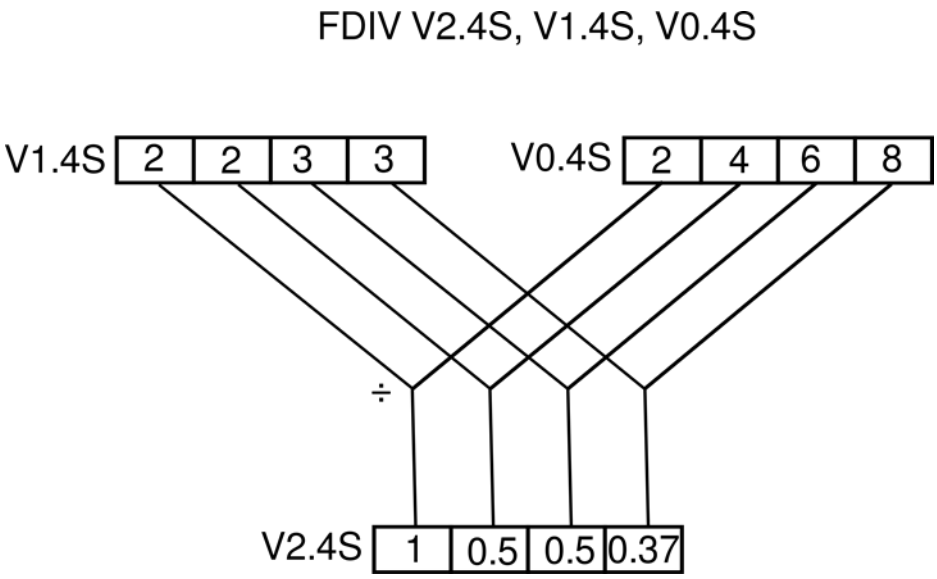
2.2.2.8. Operaciones aritméticas

La suma y resta de vector continúa utilizando las instrucciones ADD y SUB. Si se desea utilizar números en punto flotante se puede agregar el prefijo F (FADD y FSUB). Estas instrucciones tienen variantes con álgebra de saturación (Q) y también permiten utilizar los sufijos L, W y 2; por ejemplo, SQADD, USUBL, SADDW2, entre otros. Si se desea utilizar el sufijo N se pueden utilizar las instrucciones ADDH y SUBH. Estas permiten el uso del prefijo R para especificar un redondeo y el sufijo 2 para especificar los bits a utilizar por ejemplo ADDHN, RADDHN, RSUBHN2, entre otros.

La operación de multiplicación sigue manteniendo el mismo esquema, es decir se siguen utilizando las instrucciones MUL, MLA y MLS. La diferencia es que ahora se puede utilizar el sufijo 2 para las operaciones que utilicen el operando L y hay que especificar el tipo de dato (S, U o F) cuando sea relevante para la instrucción.

El estado AARCH64 soporta una instrucción para realizar la división entre números de punto flotante (FDIV). Esta instrucción soporta realizar la división en números de precisión media, simple o doble. Para realizar la división entre números enteros se puede seguir utilizando el mismo procedimiento: multiplicar por el recíproco. Para este fin se siguen utilizando las instrucciones RECPE y RSQRTE para estimar el valor recíproco, y las instrucciones RECPS y RSQRTS para mejorar la estimación mediante el método de Newton-Raphson.

Figura 38. **Ejemplo del uso de la instrucción FDIV**



Fuente: elaboración propia, realizado con Inkscape.

2.2.2.9. Operaciones lógicas

El estado AARCH64 soporta realizar las operaciones lógicas de AND, AND NOT, OR, NOR y XOR, correspondientes respectivamente a las instrucciones AND, BIC, ORR, ORN, EOR. Estas instrucciones realizan las operaciones bit a bit independientemente del contenido de los vectores. Para especificar si se trata de un vector de 64 bits o de 128 bits se utilizan los tamaños 8B y 16B respectivamente.

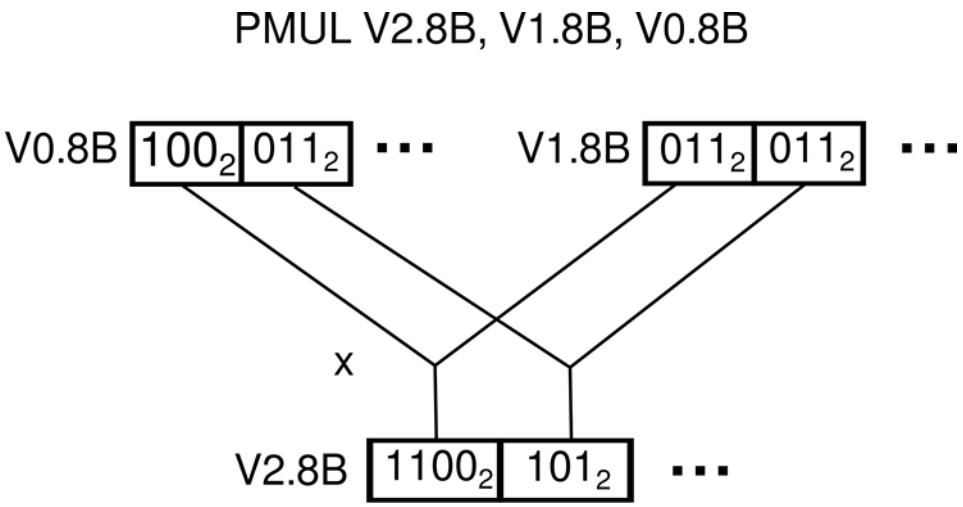
Si se desea hacer un corrimiento a la izquierda se puede utilizar la instrucción SHL. Esta instrucción puede realizar un corrimiento por un valor inmediato de bits especificado como segundo operando. Se pueden utilizar los tipos de datos S y U junto con el sufijo Q o los prefijos L y 2 (SQSHL, UQSHL, SSHLL, SSHLL2, USHLL, USHLL2). Si se desea realizar un corrimiento por una cantidad variable se puede agregar los tipos de datos S y U en la instrucción (SSHL y USHL). En estas, el segundo operando es un vector cuyos 8 bits menos significativos de cada elemento especifican el corrimiento a realizar en el elemento equivalente del primer operando. Dependiendo del signo es el tipo de corrimiento que se realizará. Los sufijos que se pueden utilizar para corrimientos variables son R y Q.

Los corrimientos hacia la derecha funcionan de la misma forma que los corrimientos hacia la izquierda, solamente se utiliza la instrucción SHR. Esta instrucción solamente permite realizar el corrimiento por un valor inmediato ya que la instrucción SHL ya permite realizar corrimientos con un vector. Se pueden utilizar los tipos de datos S y U junto con los prefijos R y Q y los sufijos N y 2.

2.2.2.10. Polinomios

Para la suma de polinomios se sigue utilizando la instrucción EOR, mientras que para la multiplicación de polinomios se puede utilizar la instrucción PMUL, que permite los sufijos L y 2 (PMULL y PMULL2). Los polinomios por utilizar tienen que ser de 8 bits como datos de entrada (los polinomios de 16 bits, que se obtienen al utilizar la instrucción PMULL, solamente son de salida). Si se han implementado las extensiones de criptografía, la instrucción PMULL soporta un polinomio de entrada de 64 bits y un polinomio resultante de 128 bits.

Figura 39. **Ejemplo del uso de la instrucción PMUL**



Fuente: elaboración propia, realizado con Inkscape.

2.2.2.11. Matrices

La extensión ARMv8.2-I8MM provee a la unidad de instrucciones para realizar multiplicación entre matrices de 2x8 y 8x2. Las instrucciones

específicas son SMMLA *(signed matrix multiply-accumulate*), UMMLA (*unsigned matrix multiply-accumulate*) y USMMLA (*signed and unsigned matrix multiply-accumulate*). Estas instrucciones realizan operaciones con números enteros de 8 bits con signo, sin signo y mezclados (la primera matriz es de números sin signo y la segunda de números con signo), respectivamente. El primer vector de datos está organizada fila por fila, mientras que el segundo vector de datos está organizada columna por columna. El resultado de la operación se suma con una matriz de 2x2 en el registro de destino, organizado fila por fila.

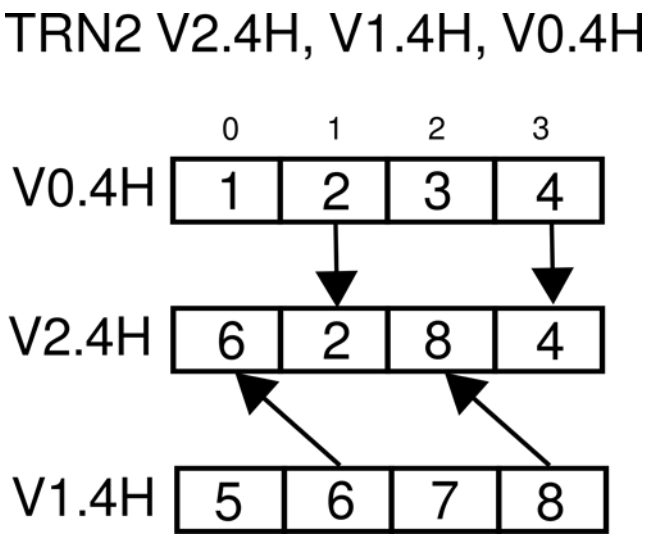
2.2.2.12. Operaciones de comparación

Las instrucciones de comparación siguen funcionando de la misma forma: si la comparación es verdadera se almacenan todos los bits del elemento resultante como uno y si la condición es falsa se almacenan todos los bits del elemento como cero. Estas instrucciones son de la forma CMn siendo n un mnemónico condicional. Se puede agregar el prefijo F a la instrucción para realizar comparaciones con números de punto flotante.

2.2.2.13. Permutación de vectores

Las instrucciones REVn, TBL, TBX y EXT siguen manteniendo la misma funcionalidad que en la arquitectura ARMV7-A. La instrucción TRN se ha expandido a dos variantes: TRN1 y TRN2. Estas variantes funcionan de forma diferente. La instrucción TRN1 toma los elementos pares de ambos operandos y almacena los elementos del primer operando en los índices pares del vector destino y los elementos del segundo operando en los índices impares del vector destino. La instrucción TRN2 funciona de forma similar solamente que utiliza los elementos impares de ambos operandos. Para obtener la traspuesta de matrices se pueden utilizar estas instrucciones en conjunto.

Figura 40. **Ejemplo del uso de la instrucción TRN2**



Fuente: elaboración propia, realizado con Inkscape.

2.2.2.14. Números complejos

La extensión ARMv8.3-CompNum permite realizar operaciones aritméticas con números complejos almacenados en un vector. Los números complejos se almacenan de forma que, para cada par de elementos, el número menos significativo es la parte real y el número más significativo es la parte imaginaria. La extensión solamente permite realizar operaciones con números de punto flotante de precisión simple o doble.

La instrucción FCADD *(floating point complex add*) realiza la suma de cada número complejo del primer vector con cada número complejo del segundo vector rotado por 90 o 270 grados en el plano complejo. Este ángulo de rotación se especifica como un tercer operando.

La instrucción FCMLA (*floating point complex multiply accumulate*) hace la siguiente operación: realiza una rotación de 0, 90, 180 o 270 grados a los números complejos del segundo vector y los multiplica por la parte real del primer vector, si la rotación fue de 0 o 180 grados, o por la parte imaginaria del primer vector, si la rotación fue de 90 o 270 grados. El resultado de la multiplicación es sumado con los números complejos existentes del vector destino. Para realizar una multiplicación de números complejos se pueden utilizar dos instrucciones FCMLA primero con una rotación de 0 y después con una rotación de 90. Se pueden utilizar diferentes conjuntos de rotaciones para realizar diversas operaciones con números complejos, por ejemplo, la multiplicación del complejo conjugado del primer vector por el segundo, que corresponde a dos instrucciones FCMLA con rotaciones de 0 y 270 grados.

2.2.2.15. Criptografía

La arquitectura ARMV8-A provee diversas extensiones para la unidad NEON para el uso de algoritmos de encriptación y funciones hash. Dependiendo de las extensiones utilizadas, la unidad puede presentar instrucciones para la encriptación por medio de AES y SM4. También presenta diversas instrucciones para realizar funciones hash, entre las cuales se encuentran SHA1, SHA2-256, SHA2-512, SHA3 y SM3.

3. OPTIMIZACIONES REALIZADAS POR EL COMPILADOR

Los compiladores pueden optimizar el código escrito en un lenguaje de alto nivel para hacer un uso eficiente de la unidad NEON. Cada compilador tiene diferentes opciones para realizar dicha optimización que van desde especificar el procesador de destino y las extensiones a utilizar, realizar simplificaciones a los ciclos, habilitar optimizaciones para números en punto flotante que puedan violar el estándar IEEE 754, entre otras.

Si bien los compiladores pueden realizar una optimización sustancial del código para realizar un gran nivel de paralelismo, no siempre es posible debido a restricciones inherentes a los tipos de variables, tamaños de los vectores o el tipo de funciones utilizadas. Un código capaz de ser optimizado por el compilador para que el procesador utilice de forma eficiente la unidad NEON tiene que ser escrito de forma que el compilador pueda saber cuáles partes del código son seguras de realizarlas de forma paralela.

3.1. Opciones de optimización del compilador GCC

Las opciones del compilador GCC se utilizan en forma de comandos que inician con un guión seguido de la opción a utilizar. El compilador soporta opciones generales que modifican el comportamiento de este y opciones específicas para cada procesador. En general el orden en que se utilicen los comandos no tiene importancia a menos que se utilicen opciones del mismo tipo. Muchos de estos comandos inician con una f seguido de la descripción de la opción (*ftree-vectorize*). Un comando que realiza lo contrario a la descripción inicia con fno (*fno-merge-constants*). Si el comando necesita algún argumento

de entrada, estos se pueden ingresar seguidos de un espacio o un símbolo de igualdad del nombre de la opción.

Se puede utilizar el comando o para controlar el nivel de optimización a realizar. Un nivel cero (o0) implica el nivel por defecto de compilación. Los niveles 1 al 3 (o1, o2, o3) implican cada vez un nivel mejor de optimización realizada por el compilador, pero sacrificando a su vez el tiempo de compilación y la generación de un código que sea fácil de depurar. Estos comandos habilitan diferentes opciones del compilador para realizar las optimizaciones necesarias, por ejemplo, el nivel -o3 activa la opción *ftree-loop-vectorize.* Por último, el nivel *ofast* habilita las opciones que descartan la adición estricta a los estándares. Por ejemplo, habilita la opción *ffast-math* que permite al compilador realizar la simplificación de expresiones en punto flotante, reordenarlas, asumir que las operaciones serán finitas, entre otras.

3.1.1. Opciones de ARM

Las opciones específicas de ARM que tiene GCC funcionan tanto para la arquitectura ARMV7-A como para el estado AARCH32 de la arquitectura ARMV8-A.

“Se puede especificar la arquitectura deseada con la opción *march* seguido de los argumentos. Esta opción sirve para que el compilador sea capaz de determinar el tipo de instrucciones que el procesador puede utilizar.”[17] Entre las arquitecturas disponibles se encuentran: armv7-a, armv8-a, armv8.1-a, armv8.2-a, armv8.3-a, armv8.4-a, armv8.5-a y armv8.6-a. Además de especificar la arquitectura a utilizar también se puede especificar las

[17] ARM Ltd. *Arm cortex-a series programmer’s guide.* https://developer.arm.com/documentation/den0013/d/. Consulta: enero de 2022.

extensiones contenidas en esta. Las opciones disponibles para las extensiones junto con su descripción se encuentran en la tabla XII.

Tabla XII. **Argumentos disponibles para la opción *march***

Argumento	Descripción
+simd	Unidad NEON V1 y las instrucciones VFPv3.
+neon-fp16	Unidad NEON V1 y las instrucciones VFPv3 con las operaciones de precisión media.
+neon-vfpv4	Unidad NEON V2 y las instrucciones VFPv4.
+crypto	Instrucciones de criptografía
+fp16	Las instrucciones de precisión media. También habilita la unidad NEON y las instrucciones de punto flotante.
+dotprod	Habilita la extensión de producto punto.
+i8mm	Habilita las instrucciones de multiplicación de matrices de enteros de 8 bits.

Fuente: elaboración propia, realizado con Microsoft Word.

Si se desea afinar el aún más el nivel de optimización, se puede utilizar la opción *mtune* para especificar el procesador para el cual se deberá adecuar el código. Algunos de los argumentos permitidos por esta opción son: cortex-a5, cortex-a7, cortex-a8, cortex-a9, cortex-a12, cortex-a15, cortex-a17, cortex-a32, cortex-a35, cortex-a53, cortex-a55, cortex-a57, cortex-a72, cortex-a73, cortex-a75, cortex-a76, cortex-a76ae, cortex-a77, cortex-a78, cortex-a78ae y cortex-a78c. Si se tiene un procesador en configuración big.LITTLE se puede

especificar el segundo procesador seguido de un punto, por ejemplo, cortex-a57.cortex-a53.

También se puede generar un código optimizado para un procesador en específico con la opción *mcpu*. Los argumentos permisibles son los mismos a la opción *mtune.* Adicionalmente se puede utilizar el argumento *+native* para detectar el procesador de la computadora. Por lo general el uso de esta opción habilita todas las extensiones que el procesador ya posee, pero se pueden modificar estas con los argumentos mostrados en la tabla XIII. Las opciones *march* y *mtune* siempre toman precedencia si se especifican junto con *mcpu*. Si el código a compilar únicamente se utilizará en el mismo procesador, el uso de la opción *mcpu* es la forma más fácil de optimizar el código. En cambio, si el código a compilar será utilizado en una familia de procesadores o en diferentes arquitecturas es mejor utilizar una combinación de *march* y *mtune.*

Tabla XIII. **Argumentos de la opción de mcpu**

Argumento	Descripción
'+nofp'	Deshabilita la extensión de punto flotante y la unidad SIMD en el procesador
'+nosimd'	Deshabilita la unidad SIMD
'+crypto'	Habilita las instrucciones de criptografía.

Fuente: elaboración propia, realizado con Microsoft Word.

Otras dos opciones que se pueden utilizar son *mfloat-abi* y *mfpu.* La primera permite especificar el tipo de ABI que se utilizará con las instrucciones en punto flotante. El argumento *hard* permite al procesador realizar las operaciones en punto flotante por medio del hardware del procesador haciendo uso de las convenciones del uso de funciones. La opción *mfpu* especifica al

compilador el tipo extensión que el procesador tiene para realizar operaciones en punto flotante. Entre los argumentos permitidos se encuentran: neon-vfpv3, neon-fp16, neon-vfpv4, neon-fp-armv8 y crypto-neon-fp-armv8. Si no se utiliza el nivel de optimización *ofast* (o se especifica la bandera *funsafe-math-optimizations* contenida en este nivel) el compilador no utilizará la unidad NEON para realizar operaciones en punto flotante. Esto es debido a que la unidad NEON en la arquitectura ARMV7-A no cumple con el estándar IEEE 754 en su totalidad.

3.1.2. Opciones de AARCH64

Al igual que para las opciones de ARM, para el estado AARCH64 se cuentan con las opciones *march, mtune* y *mcpu*. Estas opciones tienen el mismo uso y funcionalidad que las opciones de ARM. Las arquitecturas y los procesadores por especificar en las opciones tienen que poder ejecutarse en el estado AARCH64, por ejemplo, la arquitectura armv8.3-a o el procesador cortex-a73. Por defecto, con cualquier procesador o arquitectura, la unidad NEON y la unidad de punto flotante ya se encuentran habilitadas, pero se pueden habilitar o deshabilitar (al insertar un no antes de la opción) las extensiones de estas agregando los argumentos mostrados en la tabla XIV.

Tabla XIV. **Argumentos de las opciones *march, mtune* y *mcpu*.**

Argumento	Descripción
+crypto	Habilita la extensión de criptografía.
+fp	Habilita la extensión de punto flotante.
+simd	Habilita la extensión NEON.
+rdma	Enable Round Double Multiply Accumulate instructions.
+fp16	Habilita la extensión de precisión media.
+fp16fml	Habilita las instrucciones de multiplicación y adición conjunta para números de precisión media.
+dotprod	Habilita la extensión de producto punto.
+aes	Habilita las instrucciones para AES y multiplicación de polinomios de 64 bits.
+sha2	Habilita las instrucciones para SHA2.
+sha3	Habilita las instrucciones para SHA512 y SHA3.
+sm4	Habilita las instrucciones para SM3 y SM4.
+i8mm	Habilita las instrucciones de multiplicación de matrices de enteros de 8 bits.

Fuente: elaboración propia, realizado con Microsoft Word.

La opción *mfpu* ya no se utiliza porque la extensión NEON y la unidad de punto flotante son mandatorias en la arquitectura ARMV8-A. Sin importar el nivel de optimización, el compilador podrá utilizar la unidad NEON para realizar cálculos con números en puntos flotante, ya que, en el estado AARCH64, esta cumple con el estándar IEEE 754. Sin embargo, si se utiliza el nivel ofast también se pueden especificar las opciones *mlow-precision-div, mlow-precision-sqrt* y *mlow-precision-recip-sqrt* para reducir la precisión a utilizar en las operaciones de división, raíz cuadrada y la estimación del reciproco de la raíz cuadra, respectivamente.

3.2. Opciones de optimización realizadas por otros compiladores

Se pueden utilizar otros compiladores, tanto para la arquitectura ARMV7-A como la arquitectura ARMV8-A. Entre estos se encuentran el compilador Arm Compiler 6 (*armclang*), que está diseñado para el desarrollo de aplicaciones embebidas enfocadas a procesadores que corren sin un sistema operativo; el compilador ARM C/C++ (armcc), que está diseñado para su uso en sistemas operativos Linux y el compilador LLMV-clang, cuya funcionalidad es similar a GCC.

Cada compilador cuenta con sus propias opciones para optimizar el código dependiendo del procesador utilizado. Usualmente estas opciones incluyen, aunque no necesariamente se nombran de la misma forma que en GCC, los niveles de optimización dados por el comando o, opciones para especificar la arquitectura o el procesador como *march* y *mcpu* y opciones específicas de la unidad de punto flotante.

3.3. Consideraciones generales

La estructura del programa también influye en el nivel de optimización que el compilador es capaz de realizar. Para que el código pueda utilizar la unidad NEON es necesario considerar la estructura general del programa, en especial la de los ciclos porque son aquellas partes del programa que más recursos consumen.

3.3.1. Manejo de constantes y variables

El uso eficiente de las constantes y variables depende de la arquitectura del procesador. Dado que la arquitectura ARM utiliza registros de 32 o 64 bits,

dependiendo de la versión de esta, se aprovechan mejor los recursos especificando el mismo tamaño para los datos. También se pueden utilizar los registros de la unidad NEON para almacenar datos de diferentes tamaños, mostrados en las figuras 19 y 35.

3.3.2. Manejo de arreglos

Para poder utilizar los registros de manera eficiente, los arreglos tienen que ser tanto múltiplo de alguno de los tamaños soportados por los registros NEON como también pertenecer a los tipos de datos que se pueden almacenar en los registros. También es necesario tomar en cuenta el tipo de operaciones a realizar ya que no todas las instrucciones son compatibles con los tipos de datos que se pueden almacenar en un registro.

Si se utilizan estructuras es preferible que todos los elementos de la estructura tengan el mismo tamaño para poder utilizar un solo registro para su almacenamiento.

Figura 41. **Ejemplo de una estructura con elementos del mismo tamaño**

```
struct ejemplo {
    uint8_t R;
    uint8_t G;
    uint8_t B;
    };
```

Fuente: elaboración propia, realizado con Notepadqq.

El uso de los arreglos también es un factor que puede cambiar el nivel de optimización. Los elementos de un arreglo tienen que utilizarse de una forma ordenada y accederse a manera que se genere un patrón. Por ejemplo, un arreglo en el que se acceden los elementos uno a uno tiene un patrón de acceso lineal. Sin un patrón de acceso el compilador no puede utilizar las instrucciones de entrelazado para el almacenamiento y extracción de datos.

Figura 42. **Ejemplo de un patrón de acceso lineal**

```
for (i=0; i<4; i++){
    a[i] = 2*i;
    };
```

Fuente: elaboración propia, realizado con Notepadqq.

3.4. Consideraciones sobre ciclos

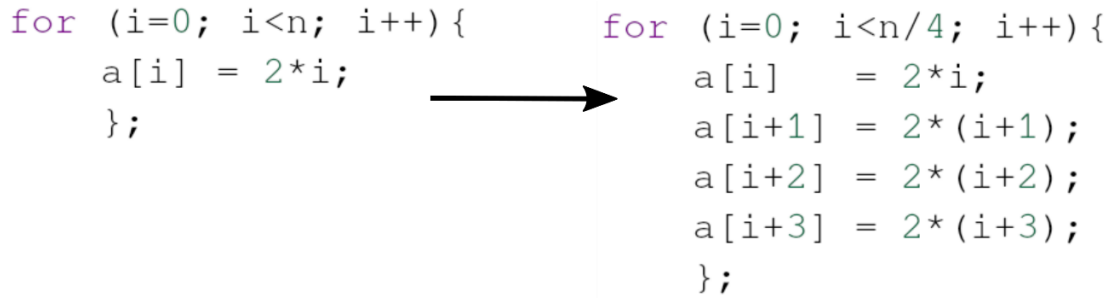
Los ciclos son la parte principal de un programa que puede ser ejecutada de forma paralela por la unidad NEON. Los compiladores son los responsables de asegurarse de que las optimizaciones realizadas a los ciclos den el mismo resultado que el código original. Tanto el contenido del ciclo, el número de iteraciones que se realizarán y el tipo de arreglos que se utilizarán dentro del ciclo afectan la posibilidad de que este pueda ser optimizado.

3.4.1. Reducción de ciclos

Cada iteración de un ciclo implica el uso de instrucciones condicionales, saltos en el código y variables que se necesitan actualizar. Dependiendo del

contenido del ciclo se pueden reducir el número de iteraciones a realizar o incluso a eliminar el ciclo completamente. Existen diversas formas de hacer esto, por ejemplo, si se opera sobre un vector, se puede utilizar en la misma iteración cuatro elementos del vector, logrando así una reducción del ciclo por un factor de cuatro.

Figura 43. **Ejemplo de una reducción por un factor de cuatro**



Fuente: elaboración propia, realizado con Notepadqq.

"El compilador *armcc* utiliza el pragma *unroll* antes de un ciclo *for* para reducirlo por un factor dado."[18] Otros compiladores utilizan diferentes métodos para hacer esta reducción, por ejemplo, en GCC esto se cumple utilizando los niveles de optimización.

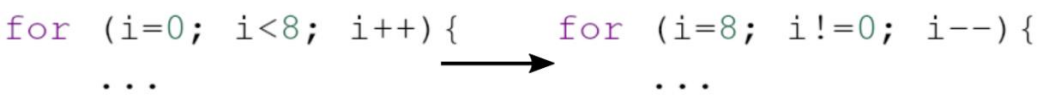
3.4.2. Finalización del ciclo y número de iteraciones

Las condiciones para finalizar el ciclo tienen que ser lo más simples posibles, ya que estas son instrucciones que se realizarán para cada iteración. La mejor forma para cumplir este requerimiento es utilizar una variable de

[18] ARM Ltd. *Arm compiler armcc user guide.* https://developer.arm.com/documentation/dui0472/m/. Consulta: enero de 2022.

control que cuente hacia cero y que el condicional del ciclo se pruebe con la variable cero. Esto es debido al bit Z del registro APSR que permite realizar comparaciones con valores nulos.

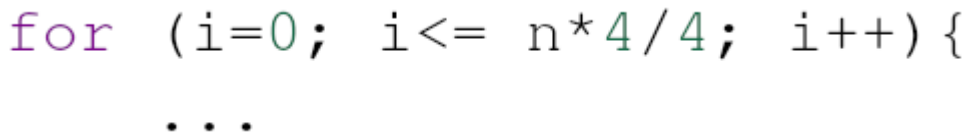
Figura 44. **Ejemplo de optimización de la terminación del ciclo**



Fuente: elaboración propia, realizado con Notepadqq.

Si no es posible cumplir con estas condiciones, se le puede dar al compilador más información sobre la cantidad de iteraciones a realizar. Idealmente las iteraciones tienen que ser un múltiplo del tamaño del vector a utilizar, es decir tienen que ser múltiplos de 2. Si la cantidad de iteraciones se desconoce, pero se sabe que cumple con el requisito anterior, se pueden realizar operaciones sobre la variable de control para indicarle al compilador la naturaleza de la variable. Por ejemplo, se puede descartar el último bit de la variable para indicarle al compilador que esta es un múltiplo de dos o dividirla por dos y multiplicarla por dos.

Figura 45. **Forma de indicar que la variable es múltiplo de cuatro**



Fuente: elaboración propia, realizado con Notepadqq.

En el compilador *armcc* se puede utilizar el pragma *promise* para darle cierta información al compilador. Por ejemplo, se le puede indicar que cierta variable siempre será un múltiplo de cuatro para que el compilador realice las optimizaciones necesarias.

Figura 46. **Ejemplo de uso del pragma *promise***

```
_promise((n % 8) == 0);
for (i = 0; i <= n; i++){
    ...
```

Fuente: elaboración propia, realizado con Notepadqq.

3.4.3. Contenido del ciclo

El contenido del ciclo también es un factor que puede afectar la optimización del código. Es preferible que las operaciones internas del ciclo sean simples y no contengan código condicional complicado, llamadas a otras funciones, o declaraciones para terminar el ciclo antes de las iteraciones indicadas.

Otro factor que también limita las optimizaciones a realizar es la dependencia entre iteraciones de un ciclo. Si un dato calculado en un ciclo se utilizará en un ciclo posterior se dice que existe una relación de dependencia entre las iteraciones del ciclo. Para que el compilador pueda optimizar el ciclo, no tienen que existir relaciones de dependencia entre las iteraciones.

Figura 47. **Ejemplo de un ciclo con dependencia entre iteraciones**

```
for (i = 0; i <= n; i++){
    a[i] = x+y;
    x = a[i]/2 + y;
    };
```

Fuente: elaboración propia, realizado con Notepadqq.

Si se utilizan punteros para acceder a arreglos en la memoria, se puede utilizar la clave *restrict* para especificarle al compilador que este puntero no accede a la misma parte de la memoria que otro puntero. Esto permite al compilador realizar las instrucciones de acceso y almacenamiento de una forma más eficiente sin tener que tomar en cuenta que los datos se puedan sobrescribir. El programador es el encargado de verificar que se cumpla el requerimiento con todos los punteros utilizados.

Figura 48. **Ejemplo de uso de la clave *restrict***

```
void ejemplo(int * restrict a, int * restrict b, int * restrict c) {
     for (i=n; i<0; i--) {
          ...
```

Fuente: elaboración propia, realizado con Notepadqq.

3.4.4. Agrupación de ciclos

Cuando se utilizan estructuras o arreglos es conveniente escribir los ciclos de forma que utilicen todo el conjunto de datos. Si se utilizan diferentes partes

de los datos en otros ciclos esto puede afectar la optimización que es posible realizar. Si se escriben múltiples ciclos para operar diferentes partes de una estructura, es mejor reestructurar el código para utilizar en un solo ciclo todas las partes de la estructura.

Figura 49. **Ciclo que utiliza todos los elementos de una estructura**

```
for (...){
    Ejemplo[i].R ...
    Ejemplo[i].G ...
    Ejemplo[i].B ...
}
```

Fuente: elaboración propia, realizado con Notepadqq.

3.5. Consideraciones sobre números en punto flotante

La optimización del código, cuando se utilizan números en punto flotante, necesita de especial consideración debido a que estos se comportan de una forma especial en sus operaciones. Esto es correcto tanto para la arquitectura ARMV8-A y ARMV7-A, sin embargo, hay que tomar en cuenta que la unidad NEON de esta última no cumple con el estándar IEEE754. Por ejemplo, esto puede afectar los resultados cuando se presenten números subnormales.

3.5.1. Diferencias entre números reales y de punto flotante

Al trabajar números en punto flotante, la principal consideración a tomar es que estos números son solamente una aproximación a los números reales. Las operaciones que son válidas para los números reales no necesariamente son válidas para números en punto flotante. Por ejemplo, las leyes asociativas y

distributivas no se cumplen con los números flotantes porque los números son redondeados por cada operación realizada.

Figura 50. **Ejemplo de leyes no válidas para números en punto flotante**

$$\begin{array}{c|c} \text{Asociativa} & \text{Distributiva} \\ (a+b)+c = a+(b+c) & (a+b)*c = a*c+b*c \end{array}$$

Fuente: elaboración propia, realizado con Texstudio.

Otro efecto para tomar en cuenta es la resta entre números. Cuando se realiza una resta entre dos números muy cercanos se eliminan los números más significativos, dejando solamente los números menos significativos y que son más propensos a ser erróneos. También puede suceder que el resultado de la resta sea un número muy pequeño y que este sea incapaz de ser representado por la precisión utilizada. Esto es más probable que suceda con la arquitectura ARMV7-A ya que la unidad NEON de esta arquitectura no es capaz de representar los números subnormales.

Este comportamiento influye en las optimizaciones que el compilador tiene permitido realizar, ya que al modificar el orden de las operaciones se pueden obtener resultados diferentes. Si se considera que estas limitaciones no son críticas para el algoritmo a utilizar, se pueden habilitar las opciones de optimización de números en punto flotante. Por ejemplo, en GCC se tendría que habilitar la opción *ffast-math*, mientras que en *armcc* se habilitaría la opción *fpmode=fast.*

4. USO DE LAS FUNCIONES INTRÍNSECAS DE NEON

La unidad NEON cuenta con funciones intrínsecas para ayudar al usuario a generar código que utilice los recursos del procesador de manera eficiente. Estas funciones están disponibles para los compiladores armcc, GCC y LLMV, siendo la misma sintaxis independientemente del compilador utilizado. Generalmente estas funciones intrínsecas se clasifican en dos: para la creación de variables y para la operación entre estas. “Las variables creadas con las funciones intrínsecas permiten asignar variables en C que corresponden a un registro NEON, ya sea D, Q o V. Las funciones intrínsecas para realizar operaciones entre variables están escritas como una llamada a una subrutina que utiliza como argumentos las variables ya creadas.”[19] La diferencia con estas funciones es que el compilador ya conoce por cual instrucción se debe de substituir esta subrutina.

Figura 51. **Sustitución de función intrínseca**

Fuente: elaboración propia, realizado con Texstudio.

El uso de las funciones intrínsecas para generar código de la unidad NEON trae ciertos beneficios. Uno de ellos es que el compilador se encarga de asignar los recursos de registros y de memoria. El compilador también puede realizar optimizaciones al código, ya sea por un reordenamiento o por un

[19] ARM Ltd. *Arm c language extensions.* https://developer.arm.com/documentation/101028/latest. Consulta: enero de 2022.

reemplazo de secuencias dependiendo del procesador utilizado. Esto permite tener un código más flexible que en el lenguaje ensamblador.

Las funciones intrínsecas están modeladas con base en el conjunto de instrucciones NEON por lo que la mayoría de las funciones se nombran de una manera similar a su contraparte en lenguaje ensamblador. Algo común en la sintaxis de las funciones intrínsecas es que existen dos variantes: una para operar registros de 64 bits y otra para operar registros de 128 bits. Cuando se utilizan elementos de 64 bits solamente se utiliza el nombre normal de la función, pero cuando se operan sobre elementos de 128 bits se agrega al final del nombre una q. También para cada función intrínseca hay que especificar el tipo de dato sobre el que se realizará la operación.

Figura 52. **Uso de una función de 64 y 128 bits**

```
c = vmul_s8(a, b)          c = vmulq_s8(a, b)
```

Fuente: elaboración propia, realizado con Texstudio.

4.1. Funciones intrínsecas para el uso de variables

Estas funciones permiten declarar variables en el lenguaje de programación, que correspondan al registro físico que se utilizará para el almacenamiento de los datos.

4.1.1. Vectores

Las funciones para declarar vectores se nombran de acuerdo con el patrón mostrado en la figura 53.

Figura 53. **Estructura general de las funciones intrínsecas**

[dato][tamaño]x[elementos]_t

Fuente: elaboración propia, realizado con Texstudio.

Los tipos de elementos son los mismos soportados por las instrucciones de la unidad NEON, pero nombrados de acuerdo con la tabla XV. El tamaño y número de elementos del vector tiene que ser consistente con los soportados por los registros, por ejemplo, no se puede declarar un número flotante de 8 bits ni un vector cuyo tamaño en total sea diferente a 64 o 128 bits.

Tabla XV. **Tipos de datos de las funciones intrínsecas**

Tipo de Dato	**Significado**
int	Entero con signo
uint	Entero sin signo
poly	Polinomio
float	Punto flotante (se utiliza también para operaciones con números complejos)

Fuente: elaboración propia, realizado con Microsoft Word.

También se pueden especificar las listas de registros de la forma mostrada en la figura 54. Si se desea operar en un vector individual de esta lista se puede realizar de la misma forma en la que se accede a un elemento de una estructura, por ejemplo, nombre.val[n].

Figura 54. **Estructura general para la asignación de listas**

Fuente: elaboración propia, realizado con Texstudio.

Un aspecto para tomar en cuenta es que estas funciones solamente sirven para inicializar la variable, no se le puede asignar valores de la misma forma que a una variable de C. Se tienen que utilizar las funciones intrínsecas de extracción de datos para esto.

4.1.2. Extracción y almacenamiento de datos

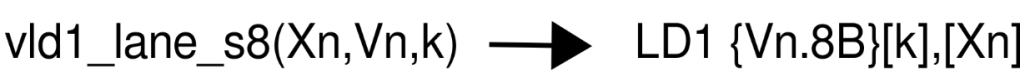
Al igual que en la sintaxis de ensamblador, los intrínsecos de extracción y almacenamiento de memoria utilizan el mnemónico ldn y stn respectivamente, siendo n el patrón de entrelazado que se desee. Existen tres variantes de esta instrucción dependiendo del tipo de operación que se desee realizar. Estas variantes se muestran en la tabla XVI.

Tabla XVI. **Funciones para la extracción y almacenamiento de datos**

Intrínseco	**Argumentos**	**Función**
vldn o vstn	Un puntero o un arreglo	Almacenamiento o extracción de datos con un patrón de entrelazado de n.
vldn_lane o vstn_lane	Un puntero o un arreglo, un vector existente y el índice k del elemento destino	Almacena o extrae n elementos de un vector existente en la memoria o en el vector destino. K es el índice del vector destino donde se almacenará o extraerá el elemento.
vldn_dup	Un puntero o un arreglo	Extrae n datos de la memoria y copia el primero a todos los elementos del primer vector y así sucesivamente.

Fuente: elaboración propia, realizado con Microsoft Word.

Figura 55. **Instrucción equivalente de la función *vldn_lane***



Fuente: elaboración propia, realizado con Texstudio.

Cuando se utiliza alguna de estas funciones intrínsecas con un valor de n igual a 1, solamente es necesario especificar un vector normal. En caso contrario es necesario especificar una lista de vectores para que la cantidad de datos sea suficiente. Además, también es necesario especificar el tipo de dato con el que se realizará la operación. Esto se realiza escribiendo la función intrínseca y, seguido de un guion, el tipo de dato a utilizar, por ejemplo, vld2_u16.

Figura 56. **Ejemplo de uso de la función *vldn***

```
#include <arm_neon.h>

unsigned int A[] = {1,2,3,4,5,6,7,8};
int main(void)
{
        uint32x4x2_t v;
        uint32x4_t k;
        v = vld2q_u32(A);
        k = vaddq_u32(v.val[0],v.val[1]);
        return 0;
}
```

Fuente: elaboración propia, realizado con Notepadqq.

Si se necesita extraer o almacenar un elemento en específico de un vector, también se cuenta con las funciones intrínsecas *vget_lane* y *vset_lane*. Estos intrínsecos son de especial utilidad cuando se necesita regresar ciertos valores a variables normales de C o viceversa. La instrucción equivalente de ensamblador es MOV.

Figura 57. **Ejemplo de uso de la función *vget_lane***

```
#include <stdio.h>
#include <arm_neon.h>

unsigned char A[] = {1,2,3,4,5,6,7,8};

int main(void)
{
        char c = 0;
        uint8x8_t v;
        v = vld1_u8(A);
        c = vget_lane_u8(v,4);
        return 0;
}
```

Fuente: elaboración propia, realizado con Notepadqq.

4.1.3. Conversión entre tipos de vectores

A veces es necesario hacer operaciones entre vectores que contienen diferentes datos. Las funciones intrínsecas no permiten realizar este tipo de operaciones ya que cada vector se declara con el tipo de dato a utilizar y la cantidad de elementos. Esto es diferente que en el lenguaje ensamblador porque a cada instrucción se le proporciona el tipo de dato a utilizar, independientemente de cómo se ha utilizado anteriormente el registro. “Para evitar esta limitación existe la función *vreinterpret* que permite utilizar el vector

como si se hubiera declarado de un diferente tipo de dato."[20] Para esto hay que especificar de primero el tipo de dato de destino y el tipo de dato fuente, como se muestra en la figura 58. Esta función no tiene su contraparte en lenguaje ensamblador debido a que no existen estas limitaciones.

Figura 58. **Ejemplo de uso de la función *vreinterpret***

```
#include <stdio.h>
#include <arm_neon.h>

unsigned char A[] = {1,2,3,4,5,6,7,8};

int main(void)
{
        int c = 0;
        uint8x8_t v;
        uint16x4_t k;
        v = vld1_u8(A);
        k = vreinterpret_u16_u8(v);
        return 0;
}
```

Fuente: elaboración propia, realizado con Notepadqq.

La función *vreinterpret* no realiza ningún cambio en los registros ni en la estructura de los bits de estos. Si se desea, en cambio, realizar una conversión del tipo de dato en un vector, se puede utilizar la función vcvt especificando de primero el tipo de dato de destino seguido del tipo de dato fuente.

[20] ARM Ltd. *Arm neon intrinsics reference.* https://developer.arm.com/documentation/ihi0073/latest. Consulta: enero de 2022.

Figura 59. **Ejemplo de uso de la función vcvt**

```
#include <stdio.h>
#include <arm_neon.h>

unsigned int A[] = {1,2,3,4};
float B[] = {0,0,0,0};

int main(void)
{
        uint32x4_t v;
        float32x4_t k;
        v = vld1q_u32(A);
        k = vcvtq_f32_u32(v);
        vst1q_f32(B,k);
        return 0;
}
```

Fuente: elaboración propia, realizado con Notepadqq.

4.1.4. Creación de vectores

La función *vcreate* permite crear un vector a partir de una variable de 64 bits. Dependiendo del tipo de dato utilizado, divide la variable de 64 bits en la cantidad de elementos del vector y le asigna el valor correspondiente a cada

uno. Esta función es de utilidad cuando no se disponen de los datos en la memoria.

Figura 60. **Ejemplo de uso de la función *vcreate***

```
#include <arm_neon.h>

int main(void)
{
        uint8x8_t v;
        v = vcreate_u8(0x01020304);
        return 0;
}
```

Fuente: elaboración propia, realizado con Notepadqq.

4.2. Operaciones con funciones intrínsecas

Las funciones intrínsecas que realizan operaciones sobre vectores siguen la misma convención establecida para la declaración de variables. Es decir, primero se escribe el nombre de la función intrínseca (usualmente el mismo nombre que la instrucción de ensamblador) seguido del tipo de dato de la operación y los argumentos de la función. Los argumentos de cada función son vectores del mismo tamaño y tipo de dato especificado en la función intrínseca. El vector resultado también tiene que estar de acuerdo con el tamaño y tipo esperado de la instrucción.

Figura 61. **Ejemplo de operaciones con funciones intrínsecas**

```
#include <arm_neon.h>

unsigned char A[] = {1,2,3,4,5,6,7,8};
unsigned char B[] = {9,10,11,12,13,14,15,16};

int main(void)
{
        uint8x8_t v1;
        uint8x8_t v2;
        uint8x8_t v3;

        v1 = vld1_u8(A);
        v2 = vld1_u8(B);

        v3 = vmul_u8(v1, v2);
        v3 = vadd_u8(v3, v1);
        return 0;
}
```

Fuente: elaboración propia, realizado con Notepadqq.

Existen ciertas variaciones respecto al nombramiento de las funciones. Por ejemplo, para la arquitectura ARMV-8 se han eliminado los prefijos de tipo de dato, entonces la función intrínseca *vadd* se puede utilizar como la instrucción FADD al especificar el tipo de dato apropiado.

Figura 62. **Ejemplo de uso de la función *vadd***

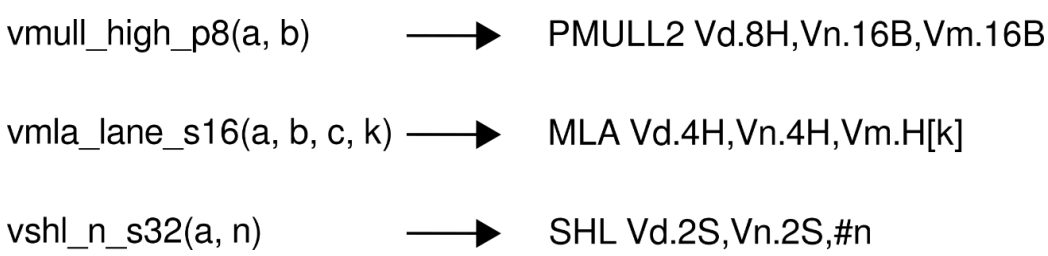
```
#include <arm_neon.h>

float A[] = {1, 2};
float B[] = {3, 4};

int main(void)
{
        float32x2_t v1;
        float32x2_t v2;
        float32x2_t v3;

        v1 = vld1_f32(A);
        v2 = vld1_f32(B);

        v3 = vadd_f32(A, B);
        return 0;
}
```

Fuente: elaboración propia, realizado con Notepadqq.

Otras variaciones se presentan cuando se utiliza el sufijo 2. Para utilizar una instrucción con este sufijo se agrega el texto *high* a la función intrínseca, como se muestra en la figura 63. De la misma forma si en los operandos de la instrucción a utilizar se encuentra un elemento especifico de un vector o un

operando es un valor inmediato se agrega *lane* o n, respectivamente, al nombre de la función intrínseca.

Figura 63. **Variaciones de las funciones intrínsecas**



Fuente: elaboración propia, realizado con Notepadqq.

CONCLUSIONES

1. La unidad avanzada SIMD es una extensión la cual, mediante su correcto uso, es capaz de optimizar el código realizando una cantidad determinada de operaciones en paralelo.

2. La programación de la unidad avanzada SIMD en lenguaje ensamblador proporciona el mejor nivel de optimización, sin embargo, requiere de un alto conocimiento tanto de su estructura interna como del conjunto de instrucciones de la unidad y el código final carece de portabilidad.

3. El uso de un compilador para la optimización del código proporciona al usuario final accesibilidad. El usuario final no necesita conocer la estructura interna de la unidad avanzada SIMD para aprovechar sus beneficios, solamente es necesario seguir las convenciones de programación y escoger las opciones del compilador deseadas.

4. Las funciones intrínsecas sirven como un compromiso entre optimización y portabilidad. Se pueden utilizar las opciones del compilador para optimizar el código a diferentes arquitecturas o procesadores, como también llamar instrucciones específicas del lenguaje ensamblador.

RECOMENDACIONES

1. Definir el alcance del proyecto antes de iniciar la programación de la unidad NEON, para seleccionar el método a utilizar.

2. Utilizar herramientas para el perfilado de funciones para permitir al usuario verificar cuales de estas son las que consumen más recursos. Se puede considerar el uso del lenguaje ensamblador para optimizar estas funciones.

3. Seguir la ABI de la unidad NEON para que las funciones creadas sean compatibles con diferentes programas.

4. Evaluar la precisión necesaria al realizar la operación de división cuando se utiliza la arquitectura ARMV8-A, ya que la diferencia entre la multiplicación por el reciproco o el uso de la instrucción de división puede llegar a ser considerable.

5. Verificar que los algoritmos que utilizan números en punto flotante no se vean afectados por las optimizaciones cuando se utiliza la bandera ofast o equivalentes en el compilador designado.

6. Utilizar la bandera mcpu=native para obtener el mejor rendimiento en el compilador, siempre y cuando el programa este destinado a ejecutarse en el mismo dispositivo.

7. Considerar que no todas las instrucciones en ensamblador tienen una función intrínseca equivalente antes de utilizar las funciones intrínsecas de la unidad NEON.

BIBLIOGRAFÍA

1. AHO, Alfred; LAM, Monica; SETHI, Ravi; ULLMAN, Jeffrey. *Compiladores principios, técnicas y herramientas*. 2a ed. México: PEARSON EDUCACIÓN, 2008. 1 037 p.

2. ARM Ltd. *Application binary interface for the arm architecture - the base atandard.* [en línea]. <https://developer.arm.com/documentation/ihi0036/latest>. [Consulta: enero de 2022].

3. _____. *ARM architecture reference manual ARMv7-A and ARMv7-R edition.* [en línea]. <https://developer.arm.com/documentation/ddi0406/cd>. [Consulta: enero de 2022].

4. _____. *Arm architecture reference manual armv8, for armv8-a architecture profile.* [en línea]. <https://developer.arm.com/documentation/ddi0553/br/>. [Consulta: enero de 2022].

5. _____. *Arm architecture reference manual supplement the scalable vector extension (sve), for armv8-a.* [en línea]. <https://developer.arm.com/documentation/ddi0584/ba/>. [Consulta: enero de 2022].

6. _____. *Arm c language extensions.* [en línea]. <https://developer.arm.com/documentation/101028/latest>. [Consulta: enero de 2022].

7. _____. *Arm compiler armasm user guide.* [en línea]. <https://developer.arm.com/documentation/dui0801/k/>. [Consulta: enero de 2022].

8. _____. *Arm compiler armcc user guide*. [en línea]. <https://developer.arm.com/documentation/dui0472/m/>. [Consulta: enero de 2022].

9. _____. *Arm compiler armclang reference guide.* [en línea]. <https://developer.arm.com/documentation/100067/0610>. [Consulta: enero de 2022].

10. _____. *Arm compiler software development guide.* [en línea]. <https://developer.arm.com/documentation/100066/0610>. [Consulta: enero de 2022].

11. _____. *Arm compiler user guide.* [en línea]. <https://developer.arm.com/documentation/100748/0617/>. [Consulta: enero de 2022].

12. _____. *Arm cortex-a series programmer's guide.* [en línea]. <https://developer.arm.com/documentation/den0013/d/>. [Consulta: enero de 2022].

13. _____. *Arm cortex-a series programmer's guide for armv8-a.* [en línea]. <https://developer.arm.com/documentation/den0024/latest/>. [Consulta: enero de 2022].

14. _____. *Arm c/c++ compiler developer and reference guide.* [en línea]. <https://developer.arm.com/documentation/101458/2110/>. [Consulta: enero de 2022].

15. _____. *Arm neon intrinsics reference.* [en línea]. <https://developer.arm.com/documentation/ihi0073/latest>. [Consulta: enero de 2022].

16. _____. *Arm reliability, availability, and serviceability (ras) specification armv8, for the armv8-a architecture profile.* [en línea]. <https://developer.arm.com/documentation/ddi0587/latest>. [Consulta: enero de 2022].

17. _____. *Neon programmer's guide.* [en línea]. <https://developer.arm.com/documentation/den0018/latest/>. [Consulta: enero de 2022].

18. HIGHAM, Nicholas. *Accuracy and stability of numerical algorithms.* 1a ed. Estados Unidos: SIAM, 1996. 718 p.

19. HOHL, William; HINDS, Christopher. *Arm assembly language fundamentals and techniques.* 2a ed. Estados Unidos: CRC Press, 2015. 448 p.

20. LANGBRIDGE, James. *Professional embedded arm development*. 1a ed. Estados Unidos: John Wiley & Sons, Inc, 2014. 288 p.

21. LEDIN, Jim. *Modern computer architecture and organization.* 1a ed. Reino Unido: Packt Publishing Ltd. 561 p.

22. PYEATT, Larry. *Modern assembly language programming with the arm processor*. 1a ed. Estados Unidos: Elsevier, 2016. 508 p.

23. PYEATT, Larry; UGHETTA, William. *Arm 64-bit assembly language.* 1a ed. Estados Unidos: Elsevier, 2020. 498 p.

24. RAJARAMAN, V*. Ieee standard for floating point numbers* [en línea]. <https://www.ias.ac.in/public/Volumes/reso/021/01/0011-0030.pdf>. [Consulta: enero de 2022].

25. VALVANO, Jonathan. *Embedded systems: introduction to arm cortex-m microcontrollers.* 5a ed. Estados Unidos: 2014. 594 p.

APÉNDICES

Apéndice 1. **Instrucciones de la unidad NEON de ARMV7-A**

Instrucción	**Significado**
VABA, VABAL	Resta, valor absoluto y acumulación
VABD, VABDL	Resta y valor absoluto
VABS	Valor absoluto
VACGE, VACGT, VACLE, VACLT	Comparación entre el primer operando y el valor absoluto del segundo operando
VADD	Suma de vectores
VADDHN	Suma de vectores, truncando la mitad menos significativa
VADDL, VADDW	Suma de vectores, con el resultado del doble de tamaño
VAND	Operación AND bit a bit
VBIC	Operación AND entre el primer elemento y el complemento del segundo bit a bit
VBIF, VBIT, VBSL	Insertar elementos bit a bit
VCEQ	Comparación de igualdad
VCGE	Comparación de mayor o igual que
VCGT	Comparación de mayor que
VCLE	Comparación de menor o igual que cero
VCLS	Conteo de bits consecutivos iguales al signo
VCLT	Comparación de menor que cero
VCLZ	Conteo de bits consecutivos iguales a cero
VCMP, VCMPE	Comparación entre dos registros de punto flotante
VCNT	Conteo de bits iguales a 1
VCVT, VCVTR	Conversión de número entero a punto flotante o viceversa
VCVTB, VCVTT	Conversión de números de precisión simple a precisión media o viceversa
VDIV	División de números de punto flotante
VDUP	Duplicación de elemento de vector
VEOR	Operación XOR bit a bit
VEXT	Extracción de elementos de vector
VFMA, VFMS	Multiplicación y suma con redondeo hasta el final
VHADD, VHSUB	Suma o resta y división por la mitad
VLD1, VLD2, VLD3, VLD4	Extracción de elementos

Continuación del apéndice 1.

Instrucción	**Significado**
VLDM	Extracción de elementos consecutivos
VLDR	Extracción de un registro SIMD
VMAX, VMIN	Comparación de valor máximo o mínimo
VMLA, VMLAL, VMLS, VMLSL	Multiplicación y suma o resta
VMOV	Mover elemento de vector
VMOVL	Mover elemento de vector, con el resultado del doble de tamaño
VMOVN	Mover elemento de vector, con el resultado de la mitad de tamaño
VMRS	Mover a registro ARM de registro NEON
VMSR	Mover a registro NEON de registro ARM
VMUL, VMULL	Multiplicación
VMVN	Operación NOT bit a bit
VNEG	Negación
VNMLA, VNMLS, VNMUL	Multiplicación, suma y negación
VORN	Operación OR NOT bit a bit
VORR	Operación OR bit a bit
VPADAL	Suma de par de elementos adyacentes y acumulación, con el resultado del doble de tamaño
VPADD	Suma de par de elementos adyacentes
VPADDL	Suma de par de elementos adyacentes, con el resultado del doble de tamaño
VPMAX, VPMIN	Comparación de valor máximo o mínimo entre par de elementos adyacentes
VPOP	Extracción de registros consecutivos de la pila
VPUSH	Almacenamiento de registros consecutivos en la pila
VQABS	Valor absoluto con saturación
VQADD	Suma con saturación
VQDMLAL, VQDMLSL	Multiplicación y suma con saturación, el resultado se duplica junto con el tamaño
VQDMULH	Multiplicación con saturación, el resultado se duplica y se retorna la mitad más significativa.
VQDMULL	Multiplicación con saturación, el resultado se duplica junto con el tamaño
VQMOVN, VQMOVUN	Mover elemento de vector con saturación, el resultado de la mitad del tamaño
VQNEG	Negación con saturación

Continuación del apéndice 1.

Instrucción	**Significado**
VQRDMULH	Multiplicación con redondeo y saturación, el resultado se duplica y se retorna la mitad más significativa
VQRSHL	Corrimiento hacia la izquierda con redondeo y saturación
VQRSHRN, VQRSHRUN	Corrimiento hacia la derecha con redondeo y saturación
VQSHL, VQSHLU	Corrimiento hacia la izquierda con saturación
VQSHRN, VQSHRUN	Corrimiento hacia la derecha con saturación
VQSUB	Resta con saturación
VQRADDHN	Suma con redondeo, con el resultado de la mitad del tamaño
VRECPE	Estimación del reciproco
VRECPS	Cálculo del paso para la estimación del reciproco
VREV16, VREV32, VREV64	Copiar bits al revés en grupos de n bits
VRHADD	Suma y redondeo, el resultado se divide por la mitad
VRSHL	Corrimiento hacia la izquierda con redondeo y saturación
VRSHR	Corrimiento hacia la derecha con redondeo
VRSHRN	Corrimiento hacia la derecha con redondeo y resultado truncado a la mitad
VRSQRTE	Estimación del reciproco de la raíz cuadrada
VRSQRTS	Cálculo del paso para la estimación del reciproco de la raíz cuadrada
VRSRA	Suma y corrimiento hacia la derecha con redondeo
VRSUBHN	Resta con redondeo y el resultado truncado a la mitad
VSHL	Corrimiento hacia la izquierda
VSHLL	Corrimiento hacia la izquierda con el resultado del doble de tamaño
VSHR	Corrimiento hacia la derecha
VSHRN	Corrimiento hacia la derecha con el resultado de la mitad de tamaño
VSLI	Corrimiento hacia la izquierda e insertar
VSQRT	Raíz cuadrada
VSRA	Suma y corrimiento hacia la izquierda
VSRI	Corrimiento hacia la derecha e insertar
VST1, VST2, VST3, VST4	Almacenamiento con entrelazado
VSTM	Almacenamiento de elementos consecutivos
VSTR	Almacenar registro SIMD

Continuación del apéndice 1.

Instrucción	Significado
VSUB	Resta con redondeo y el resultado truncado a la mitad
VSUBHN	Resta con el resultado siendo la mitad más significativa
VSUBL, VSUBW	Resta de números, con el resultado del doble de tamaño
VSWP	Intercambiar el contenido de dos vectores
VTBL, VTBX	Tabla de búsqueda de vector
VTRN	Transponer vectores
VTST	Comparación entre elementos de vectores bit a bit
VZIP, VUZIP	Entrelazado de vectores

Fuente: elaboración propia, realizado con Microsoft Word.

Apéndice 2. **Instrucciones de la unidad NEON de ARMV8-A**

Instrucción	Significado
ABS	Valor absoluto
ADD	Suma de vectores
ADDHN, ADDHN2	Suma de vectores, truncando la mitad menos significativa
ADDP	Suma de cada par de elementos adyacentes
ADDV	Suma de todos los elementos de un registro
AESD	Descifrado AES de una ronda.
AESE	Encriptado AES de una ronda
AESIMC	Mezcla inversa de columnas AES
AESMC	Mezcla de columnas AES
AND	Operación AND bit a bit
BCAX	Operación AND y XOR bit a bit
BFCVT	Conversión de punto flotante de precisión simple a formato Bfloat16
BFCVTN, BFCVTN2	Conversión de punto flotante de precisión simple a formato Bfloat16, truncando la mitad menos significativa
BFDOT	Producto punto en formato Bfloat16
BFMLALB, BFMLALT	Multiplicación y suma con ensanchamiento en formato BFLOAT16. Ensancha los elementos pares (B) o los impares (T)
BFMMLA	Multiplicación y suma de matrices de 2x2 en formato Bfloat16
BIC	Operación AND entre el primer elemento y el complemento del segundo bit a bit

Continuación del apéndice 2.

Instrucción	Significado
BIF	Insertar bit del primer registro si el bit del segundo registro es 0
BIT	Insertar bit del primer registro si el bit del segundo registro es 1
BSL	Insertar bit del primer registro si el bit destino es 1, de lo contrario insertar bit del segundo registro
CSL	Conteo de los bits consecutivos iguales al bit de signo
CLZ	Conteo de los ceros consecutivos empezando del bit más significativo
CMEQ	Comparación de igualdad bit a bit
CMEG	Comparación de mayor o igual que
CMGT	Comparación de mayor que
CMHI	Comparación de mayor que para elementos sin signo
CMHS	Comparación de mayor que o igual para elementos sin signo
CMLE	Comparación menor que o igual.
CMLT	Comparación menor que
CMTST	Comparación bit a bit si el resultado no es cero
CNT	Conteo de bits iguales a 1
DUP	Duplicación de elemento de vector
EOR	Operación XOR bit a bit
EOR3	Operación XOR de tres elementos bit a bit
EXT	Extracción de elementos de vector
FABD	Diferencia absoluta en punto flotante
FABS	Valor absoluto en punto flotante
FACGE	Comparación del valor absoluto mayor que o igual en punto flotante
FACGT	Comparación del valor absoluto mayor que en punto flotante
FADD	Suma en punto flotante
FADDP	Suma de cada par de elementos adyacentes en punto flotante
FCADD	Suma números complejos en punto flotante
FCCMP, FCCMPE	Comparación condicional en punto flotante
FCMEQ	Comparación de igualdad en punto flotante
FCMGE	Comparación de mayor que o igual en punto flotante
FCMGT	Comparación de mayor que en punto flotante
FCMLA	Multiplicación y suma de números complejos en punto flotante
FCMLE	Comparación menor que o igual a cero en punto flotante
FCMLT	Comparación de menor que cero en punto flotante
FCMP, FCMPE	Comparación en punto flotante
FCSEL	Selección condicional en punto flotante

Continuación del apéndice 2.

Instrucción	Significado
FCVT	Conversión de precisión de números de punto flotante
FCVTAS	Conversión de punto flotante a enteros con signo, redondeo alejándose del cero
FCVTAU	Conversión de punto flotante a enteros sin signo, redondeo alejándose del cero
FCVTL, FCVTL2	Conversión a valor del doble de precisión
FCVTMS	Conversión de punto flotante a entero con signo, redondeo hacia menos infinito
FCVTMU	Conversión de punto flotante a entero sin signo, redondeo hacia menos infinito
FCVTN, FCVTN2	Conversión a valor de la mitad de precisión
FCVTNS	Conversión de punto flotante a entero con signo, redondeo hacia número par
FCVTNU	Conversión de punto flotante a entero sin signo, redondeo hacia número par
FCVTPS	Conversión de punto flotante a entero con signo, redondeo hacia más infinito
FCVTPU	Conversión de punto flotante a entero sin signo, redondeo hacia más infinito
FCVTXN, FCVTXN2	Conversión a valor de la mitad de precisión, redondeo hacia impares
FCVTZS	Conversión de punto flotante a entero o punto fijo con signo, redondeo hacia el cero.
FCVTZU	Conversión de punto flotante a entero o punto fijo sin signo, redondeo hacia el cero.
FDIV	División en punto flotante
FJCVTZS	Conversión de punto flotante Javascript a punto fijo con signo, redondeo hacia el cero
FMADD	Multiplicación y suma en punto flotante
FMAX, FMAXNM	Comparación de valor máximo en punto flotante
FMAXV, FMAXNMV	Comparación de valor máximo en un vector
FMAXP, FMAXNMP	Comparación de valor máximo entre par de elementos
FMIN, FMINNMV	Comparación de valor mínimo en punto flotante
FMINP, FMINNMP	Comparación de valor mínimo entre par de elementos
FMINV, FMINNMV	Comparación de valor mínimo en un vector
FMLA, FMLAL, FMLAL2	Multiplicación y suma en punto flotante
FMLS, FMLSL, FMLSL2	Multiplicación y resta en punto flotante
FMOV	Mover de o hacia un registro de propósito general
FMSUB	Multiplicación y resta en punto flotante

Continuación del apéndice 2.

Instrucción	Significado
FMUL, FMULX	Multiplicación en punto flotante
FNEG	Negación en punto flotante
FNMADD	Multiplicación y suma con negación en punto flotante
FNMSUB	Multiplicación y resta con negación en punto flotante
FNMUL	Multiplicación y negación en punto flotante
FRECPE	Estimación del reciproco en punto flotante
FRECPS	Cálculo del paso para la estimación del reciproco
FRECPX	Estimación del exponente reciproco
FRINT32X	Redondeo de punto flotante a entero de 32 bits
FRINT32Z	Redondeo hacia el cero de punto flotante a entero de 32 bits
FRINT64X	Redondeo de punto flotante a entero de 64 bits
FRINT64Z	Redondeo hacia el cero de punto flotante a entero de 64 bits
FRINTA	Redondeo alejándose del cero de punto flotante a entero.
FRINTI	Redondeo de punto flotante a entero.
FRINTM	Redondeo hacia menos infinito de punto flotante a entero
FRINTN	Redondeo hacia número par de punto flotante a entero
FRINTP	Redondeo hacia el infinito de punto flotante a entero
FRINTX	Redondeo de punto flotante a entero exacto
FRINTZ	Redondeo hacia el cero de punto flotante a entero
FRSQRTE	Estimación del reciproco de la raíz cuadrada en punto flotante
FRSQRTS	Cálculo del paso para la estimación del reciproco de la raíz cuadrada
FSQRT	Raíz cuadrada en punto flotante
FSUB	Resta en punto flotante
INS	Insertar elemento de otro vector
LD1, LD2, LD3, LD4	Extracción de elementos
LD1R, LD2R, LD3R, LD4R	Extracción de un mismo elemento a todo el vector
LDP, LDNP	Extracción de par de registros SIMD y FP
LDR, LDUR	Extracción de un registro SIMD
MLA	Multiplicación y suma
MLS	Multiplicación y resta
MOV	Mover elemento de vector
MOVI	Mover valor inmediato
MUL	Multiplicación
MVN	Operación NOT bit a bit

Continuación del apéndice 2.

Instrucción	Significado
MVNI	Mover valor inmediato negado
NEG	Negación
NOT	Operación NOT
ORN	Operación OR NOT bit a bit
ORR	Operación OR bit a bit
PMUL	Multiplicación de polinomios
PMULL, PMULL2	Multiplicación de polinomios con el resultado del doble de tamaño
RADDHN, RADDHN2	Suma con redondeo, el resultado es la mitad más significativa
RAX1	Rotar y operación XOR
RBIT	Copiar bits al revés
REV16	Copiar bits al revés en grupos de 16 bits
REV32	Copiar bits al revés en grupos de 32 bits
REV64	Copiar bits al revés en grupos de 64 bits
RSHRN, RSHRN2	Corrimiento hacia la derecha con redondeo y resultado truncado a la mitad
RSUBHN, RSUBHN2	Resta con redondeo y el resultado truncado a la mitad
SABA	Resta, valor absoluto y suma
SABAL, SABAL2	Resta, valor absoluto y suma, con el resultado del doble de tamaño
SABD	Resta con valor absoluto de enteros con signo
SABDL, SABDL2	Resta con valor absoluto de enteros con signo, el resultado del doble de tamaño
SADALP	Suma de cada par de elementos adyacentes de enteros con signo, el resultado del doble de tamaño
SADDL, SADDL2	Suma de enteros con signo, el resultado del doble de tamaño
SADDLP	Suma de cada par de elementos adyacentes de enteros con signo, el resultado del doble de tamaño
SADDLV	Suma de todos los elementos de un registro de enteros con signo, el resultado del doble de tamaño
SADDW, SADDW2	Suma de números enteros con signo, el resultado del doble de tamaño que el segundo registro
SCVTF	Conversión de número con signo a punto flotante
SDOT	Producto punto de números enteros con signo
SHA1C, SHA1M, SHA1P	Actualización de hash SHA1
SHA1H	Rotación fija SHA1
SHA1SU0, SHA1SU1	Programación de actualización SHA1
SHA256H, SHA256H2	Actualización de hash SHA256

Continuación del apéndice 2.

Instrucción	Significado
SHA256SU0, SHA256SU1	Programación de actualización SHA256
SHA512H, SHA512H2	Actualización de hash SHA512
SHA512SU0, SHA512SU1	Programación de actualización SHA512
SHADD	Suma y división por la mitad de enteros con signo
SHL	Corrimiento hacia la izquierda
SHLL, SHLL2	Corrimiento hacia la izquierda con el resultado del doble de tamaño
SHRN, SHRN2	Corrimiento hacia la derecha con el resultado de la mitad de tamaño
SHSUB	Resta y división por la mitad de enteros con signo
SLI	Corrimiento hacia la izquierda e insertar
SM3PARTW1, SM3PARTW2	Rotación y XOR para SM3
SM3SS1	Rotación para SM3
SM3TT1A, SM3TT1B	Suma y XOR para SM3
SM3TT2A, SM3TT2B	Suma y XOR para SM3
SM4E	Encriptación SM4
SM4EKEY	Llave para SM4
SMAX	Comparación de valor máximo en números enteros con signo
SMAXP	Comparación de valor máximo entre par de elementos en números enteros con signo
SMAXV	Comparación de valor máximo en un vector de números enteros con signo
SMIN	Comparación de valor mínimo en números enteros con signo
SMINP	Comparación de valor mínimo entre par de elementos en números enteros con signo
SMINV	Comparación de valor mínimo en un vector de números enteros con signo
SMLAL, SMLAL2	Multiplicación y suma de números enteros con signo, el resultado del doble de tamaño
SMLSL, SMLSL2	Multiplicación y resta de números enteros con signo, el resultado del doble de tamaño
SMMLA	Multiplicación matricial de enteros con signo
SMOV	Mover número entero con signo de vector
SMULL, SMULL2	Multiplicación de números enteros con signo, el resultado del doble de tamaño

Continuación del apéndice 2.

Instrucción	Significado
SQABS	Valor absoluto con saturación de números enteros con signo
SQADD	Suma con saturación de números enteros con signo
SQDMLAL, SQDMLAL2	Multiplicación y suma con saturación de números enteros con signo, el resultado se duplica junto con el tamaño
SQDMLSL, SQDMLSL2	Multiplicación y resta con saturación de números enteros con signo, el resultado se duplica junto con el tamaño
SQDMULH	Multiplicación con saturación de números enteros con signo, el resultado se duplica y se retorna la mitad más significativa.
SQDMULL, SQDMULL2	Multiplicación con saturación de números enteros con signo, el resultado se duplica junto con el tamaño
SQNEG	Negación con saturación de números enteros con signo
SQRDMLAH	Multiplicación y suma con redondeo y saturación de números enteros con signo, el resultado se duplica y se retorna la mitad más significativa
SQRDMLSH	Multiplicación y resta con redondeo y saturación de números enteros con signo, el resultado se duplica y se retorna la mitad más significativa
SQRDMULH	Multiplicación con redondeo y saturación de números enteros con signo, el resultado se duplica y se retorna la mitad más significativa
SQRSHL	Corrimiento hacia la izquierda con redondeo y saturación de números enteros con signo
SQRSHRN, SQRSHRN2	Corrimiento hacia la derecha con redondeo y saturación de números enteros con signo
SQRSHRUN, SQRSHRUN2	Corrimiento hacia la derecha con redondeo y saturación de números enteros sin signo
SQSHL	Corrimiento hacia la izquierda con saturación de números enteros con signo
SQSHLU	Corrimiento hacia la izquierda con saturación de números enteros sin signo
SQSHRN, SQSHRN2	Corrimiento hacia la derecha con saturación de números enteros con signo
SQSHRUN, SQSHRUN2	Corrimiento hacia la derecha con saturación de números enteros sin signo
SQSUB	Resta con saturación de números enteros con signo
SQXTN, SQXTN2	Extracción y saturación de enteros con signo
SQXTUN, SQXTUN2	Extracción y saturación de enteros sin signo
SRHADD	Suma y redondeo de enteros con signo, el resultado se divide por la mitad
SRI	Corrimiento hacia la derecha e insertar
SRSHL	Corrimiento hacia la izquierda con redondeo de números enteros con signo

Continuación del apéndice 2.

Instrucción	Significado
SRSHR	Corrimiento hacia la derecha con redondeo de números enteros con signo
SRSRA	Suma y corrimiento hacia la derecha con redondeo de números enteros con signo
SSHL	Corrimiento hacia la izquierda de números enteros con signo
SSHLL, SSHLL2	Corrimiento hacia la izquierda de números enteros con signo, con el resultado del doble de tamaño
SSHR	Corrimiento hacia la derecha de números enteros con signo
SSRA	Suma y corrimiento hacia la izquierda de números enteros con signo
SSUBL, SSUBL2	Resta de números enteros con signo, con el resultado del doble de tamaño
SSUBW, SSUBW2	Resta de números enteros con signo, con el resultado del doble de tamaño del segundo operando
ST1, ST2, ST3, ST4	Almacenamiento con entrelazado
STP, STNP	Almacenamiento de par de registros SIMD y FP
STR, STUR	Almacenar registro SIMD
SUB	Resta con redondeo y el resultado truncado a la mitad
SUBHN, SUBHN2	Resta con el resultado siendo la mitad más significativa
SUBDOT	Producto punto de números enteros con signo y sin signo
SUQADD	Suma con saturación de número entero con signo y sin signo
SXTL, SXTL2	Extensión de vector mediante el signo
TBL	Tabla de búsqueda de vector
TBX	Extensión de tabla de búsqueda de vector
TRN1, TRN2	Transponer vectores
UABA	Resta y valor absoluto con suma de números enteros sin signo
UABAL, UABAL2	Resta y valor absoluto con suma de números enteros sin signo, el resultado del doble de tamaño
UABD	Resta y valor absoluto de números enteros sin signo
UABDL, UABDL2	Resta y valor absoluto de números enteros sin signo, el resultado del doble de tamaño
UADALP	Suma y acumulación de cada par de elementos adyacentes, el resultado del doble de tamaño
UADDLV	Suma de todos los elementos de un registro de enteros sin signo, el resultado del doble de tamaño
UADDW, UADDW2	Suma de números enteros sin signo, el resultado del doble de tamaño que el segundo registro
UCVTF	Conversión de número sin signo a punto flotante

Continuación del apéndice 2.

Instrucción	Significado
UDOT	Producto punto de números enteros sin signo
UHADD	Suma y división por la mitad de enteros sin signo
UHSUB	Resta y división por la mitad de enteros sin signo
UMAX	Comparación de valor máximo en números enteros sin signo
UMAXP	Comparación de valor máximo entre par de elementos en números enteros sin signo
UMAXV	Comparación de valor máximo en un vector de números enteros sin signo
UMIN	Comparación de valor mínimo en números enteros sin signo
UMINP	Comparación de valor mínimo entre par de elementos en números enteros sin signo
UMINV	Comparación de valor mínimo en un vector de números enteros sin signo
UMLAL, UMLAL2	Multiplicación y suma de números enteros sin signo, el resultado del doble de tamaño
UMLSL, UMLSL2	Multiplicación y resta de números enteros sin signo, el resultado del doble de tamaño
UMMLA	Multiplicación matricial de enteros sin signo
UMOV	Mover número entero sin signo de vector
UMULL, UMULL2	Multiplicación de números enteros sin signo, el resultado del doble de tamaño
UQADD	Suma con saturación de números enteros sin signo
UQRSHL	Corrimiento hacia la izquierda con redondeo y saturación de números enteros sin signo
UQRSHRN, UQRSHRN2	Corrimiento hacia la derecha con redondeo y saturación de números enteros sin signo
UQSHL	Corrimiento hacia la izquierda con saturación de números enteros sin signo
UQSHRN, UQSHRN2	Corrimiento hacia la derecha con saturación de números enteros sin signo
UQSUB	Resta con saturación de números enteros sin signo
UQXTN, UQXTN2	Extracción y saturación de enteros sin signo
URECPE	Estimación del reciproco de números enteros sin signo
URHADD	Suma y redondeo de enteros sin signo, el resultado se divide por la mitad
URSHL	Corrimiento hacia la izquierda con redondeo de números enteros sin signo
URSHR	Corrimiento hacia la derecha con redondeo de números enteros sin signo
URSQRTE	Estimación del reciproco de la raíz cuadrada de números enteros sin signo

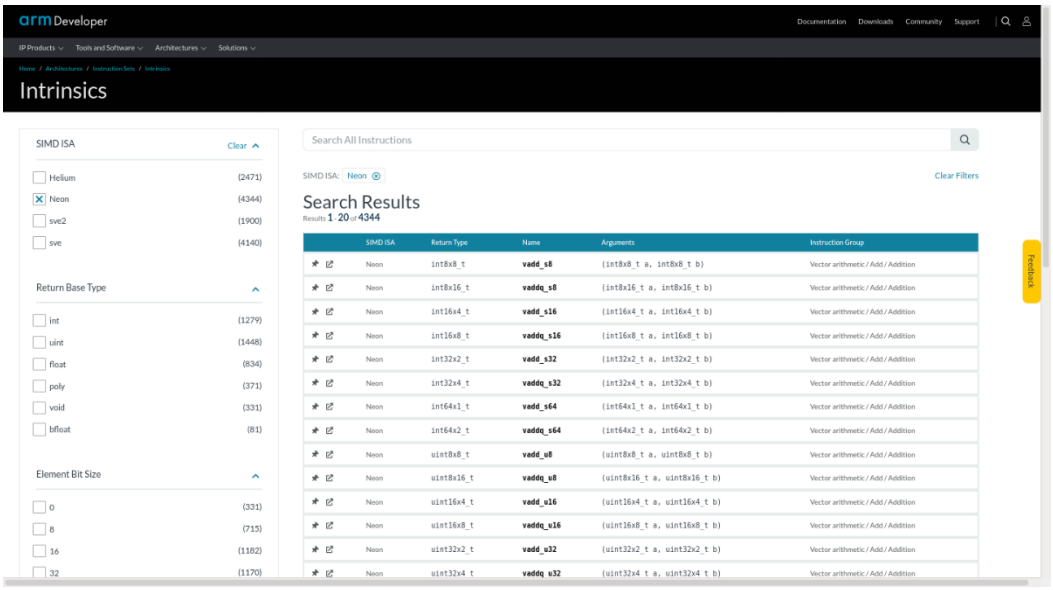
Continuación del apéndice 2.

Instrucción	Significado
URSRA	Suma y corrimiento hacia la derecha con redondeo de números enteros sin signo
USDOT	Producto punto de números enteros con signo y sin signo
USHL	Corrimiento hacia la izquierda de números enteros sin signo
USHLL, USHLL2	Corrimiento hacia la izquierda de números enteros sin signo, con el resultado del doble de tamaño
USHR	Corrimiento hacia la derecha de números enteros sin signo
USMMLA	Multiplicación matricial de enteros con signo y sin signo
USQADD	Suma y saturación de número entero con signo, resultado sin signo
USRA	Suma y corrimiento hacia la izquierda de números enteros sin signo
USUBL, USUBL2	Resta de números enteros sin signo, con el resultado del doble de tamaño
USUBW, USUBW2	Resta de números enteros sin signo, con el resultado del doble de tamaño del segundo operando
UXTL, UXTL2	Extensión de vector sin signo
UZP1, UZP2	Entrelazado de vectores
XAR	Operación XOR y rotación
XTN, XTN2	Extracción de elementos, con el resultado de la mitad del tamaño
ZIP1, ZIP2	Entrelazado de vectores

Fuente: elaboración propia, realizado con Microsoft Word.

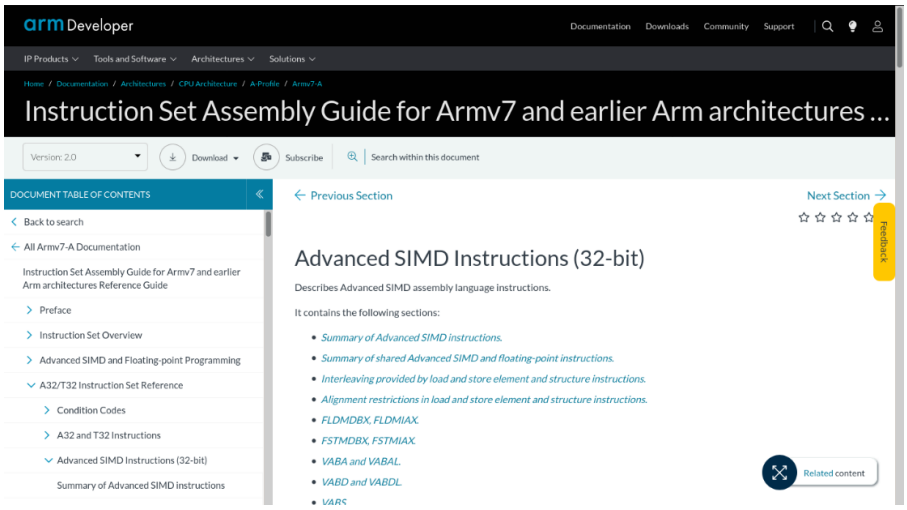
ANEXOS

Anexo 1. **Búsqueda de funciones intrínsecas de la unidad NEON**



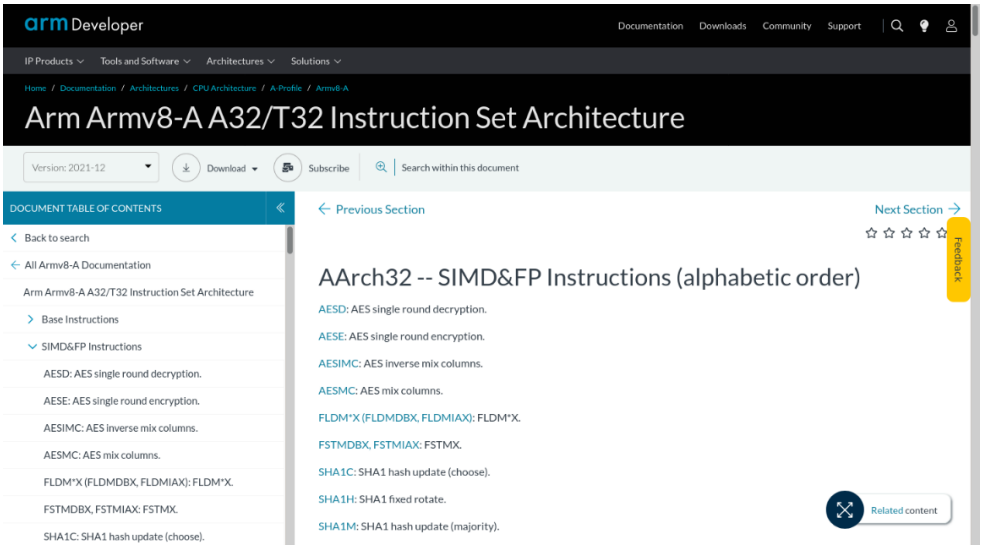
Fuente: ARM Ltd. *Instrinsics*. https://developer.arm.com/architectures/instruction-sets/intrinsics/. Consulta: 12 de enero de 2022.

Anexo 2. **Búsqueda de instrucciones NEON de ARMV7-A**



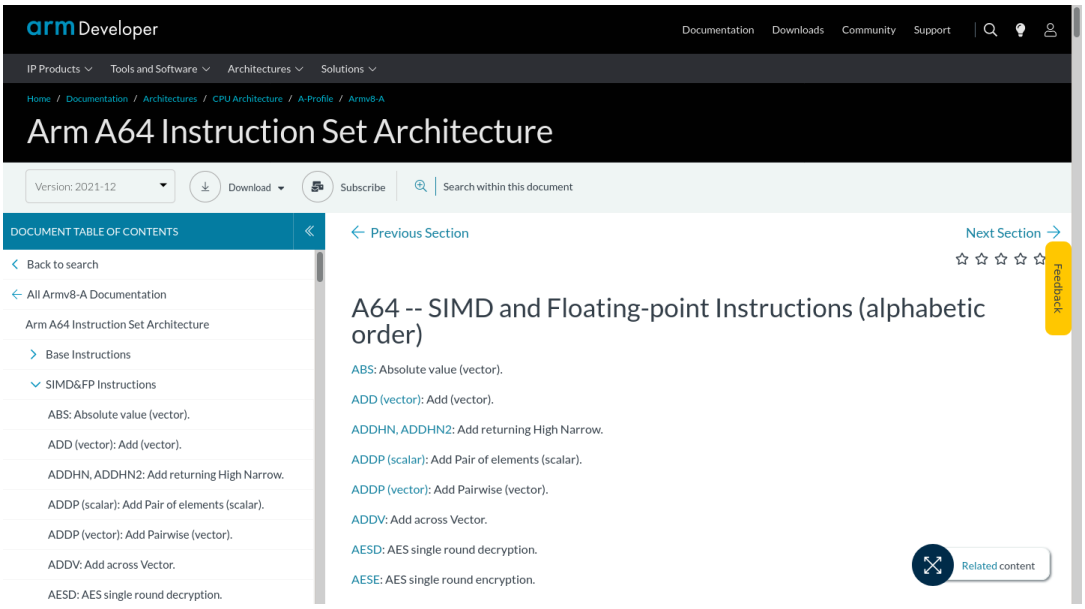
Fuente: *ARM Ltd. Advanced SIMD instructions.* https://developer.arm.com/documentation/100076/0200/a32-t32-instruction-set-reference/advanced-simd-instructions--32-bit-/. Consulta: 12 de enero de 2022.

Anexo 3. **Búsqueda de instrucciones NEON de ARMV8-A A32**



Fuente: ARM Ltd. *SIMD & FP instructions.* https://developer.arm.com/documentation/ddi0597/2021-12/SIMD-FP-Instructions. Consulta: 12 de enero de 2022.

Anexo 4. **Búsqueda de instrucciones NEON de ARMV8-A A64**



Fuente: *ARM Ltd. SIMD and floating-point instructions*. https://developer.arm.com/documentation/ddi0596/2021-12/SIMD-FP-Instructions. Consulta: 12 de enero de 2022.