

# Introducción a la arquitectura ArmV8-M

Resumen

Public - Cortex-M Profile Software Training

10/07/2020

# Índice

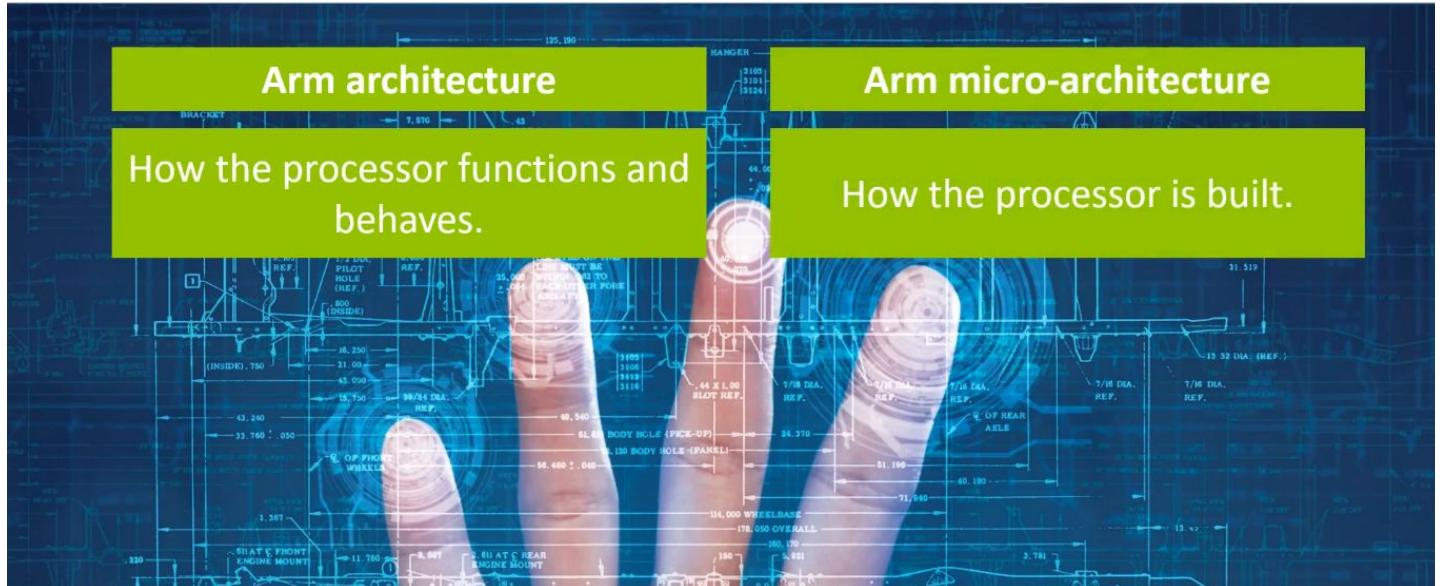
<b>Introducción</b>	<b>2</b>
Que es la arquitectura ARMv8-M	2
¿ Que hace a un Cortex-M perfil M ?	3
Evolución de la arquitectura Armv8-M	3
Que se incluye en ARMv8-M	4
Otras arquitecturas de Arm	5
<b>Descripción general del conjunto de instrucciones</b>	<b>6</b>
Tipos de datos Armv8-M (Data types)	6
Conjunto de registro (register set) del perfil Armv8-M	7
Armv8-M Instruction set	7
Armv8-M load / store architecture	8
Modos de direccionamiento Armv8-M	9
Instrucciones de procesamiento de datos Armv8-M	9
Armv8-M data processing reference	10
Instrucciones de control de flujo (flow control) Armv8-M	10
<b>Modos, privilegios y pilas</b>	<b>12</b>
Descripción general (modos Armv8-M, privilegios y pilas)	12
Modos, privilegios y pilas	12
Ejemplo de uso del modo procesador (Processor mode)	13
Armv8-M - MSPLIM y PSPLIM	13
<b>Arm Memory Model and Protection</b>	<b>14</b>
Arm Memory Model and Protection Overview	14
Armv8-M System address map part 1	14
Armv8-M System address map part 2	15
Armv8-M memory types and properties	15
Armv8-M - Normal memory	16
Cacheability and Shareability	17
Armv8-M - Device memory	18
Atributos de Device memory	19
Descripción general Armv8-M Memory Protection Unit (MPU)	20
Programando la MPU	21
<b>Excepciones</b>	<b>23</b>
Descripción general de las excepciones de Armv8-M	23
Descripción general del manejo de excepciones de Armv8-M	23
Special Program Status Registers (SPSR)	24
Modelo de prioridad Armv8-M	25
Armv8-M exception priorities	26
Comportamientos Armv8-M	27
Armv8-M stacking exception entry	28
Comportamiento de entrada de excepción Armv8-M	29
Comportamiento de retorno de excepción Armv8-M	30
Ejemplo de entrada de excepción Armv8-M NMI	32
Ejemplo de retorno de excepción de Armv8-M NMI	34
Armv8-M fault exceptions	35
Resumen de Armv8-M	37

# Introducción

## Que es la arquitectura ARMv8-M



What is architecture?



What is architecture?

**arm CORTEX-M**



M3



M7



M33

Armv7-M architecture

Armv8-M architecture

3-stage pipeline

6-stage pipeline

DSP extension

## ¿ Que hace a un Cortex-M perfil M ?

Las 2 cualidades más importantes son **Eficiencia** y **Simplicidad**.

Se puede programar una aplicación enteramente en **C**, por lo que no es necesario usar **Assembly**.



### What makes a Cortex-M processor M?



## Evolución de la arquitectura Armv8-M

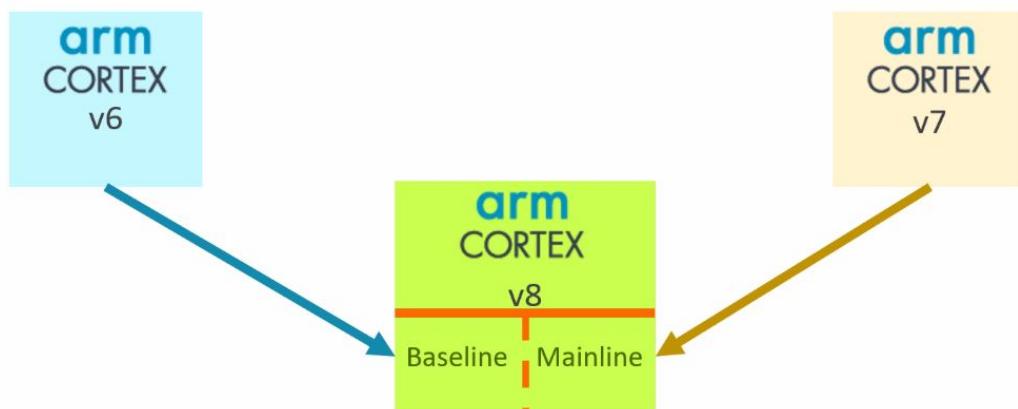
**ARMv8-M** es una evolución directa de **ARMv6-M** y **ARMv7-M**.

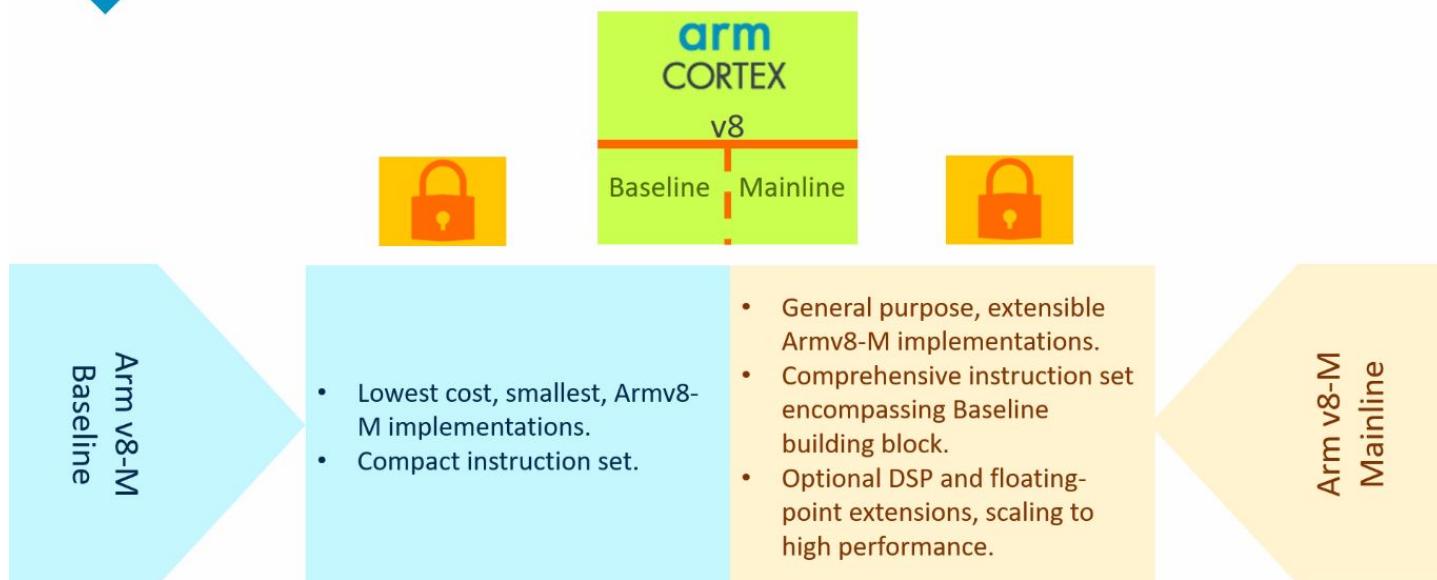
**ARMv7-M** no ha evolucionado de la arquitectura de **ARMv6-M**. La diferencia entre estas es que **ARMv6-M** fue diseñada para ser la versión más eficiente del perfil M, a cambio de la capacidad de ejecutar la mayoría de las instrucciones de 32 bits y algunas otras características incluidas en **ARMv7-M**.

ARMv8-M se divide en 2 sub arquitecturas.



### Evolution of the Arm-M architecture





## Que se incluye en ARMv8-M

Arm se ha basado en las arquitecturas **Armv6-M** y **Armv7-M** para producir la arquitectura **Armv8-M**.

**Armv8-M** es altamente compatible con las arquitecturas **Armv6-M** y **Armv7-M** para permitir la migración fácil de software a través de la familia de procesadores Cortex-M.

**Armv8-M** sigue siendo una arquitectura de **32 bits** y, para admitir la diversidad de aplicaciones integradas entre las arquitecturas v6-M y v7-M, se divide en dos perfiles, **Baseline** y **Mainline**.

### Arquitectura Baseline:

- Extiende **Armv6-M** al agregar large move-immediate, divide, acquire/release and exclusives
- Adición de semáforos: **LDREX**, **STREX** para Byte, Word y Halfword según **Arm v7-M**.
- Adición de C11 / C++ 11 atomics: **LDA**, **STL** y variantes exclusivas para accesos de Bytes, Words y Halfwords.
- Adición de soporte de sólo ejecución: inclusión de movimientos inmediatos **MOVW** y **MOVT** de **16 bits** según **Armv7-M**.
- Mejoras de rendimiento sobre v6-M: división de enteros con signo / sin signo, bifurcación de **32 bits** de largo, Comparar y bifurcar si es cero / no es cero. (Compare and branch if zero / non-zero)
- Opcional Embedded Trace Macrocell (**ETM**) y Trace Port Interface Unit (**TPIU**).

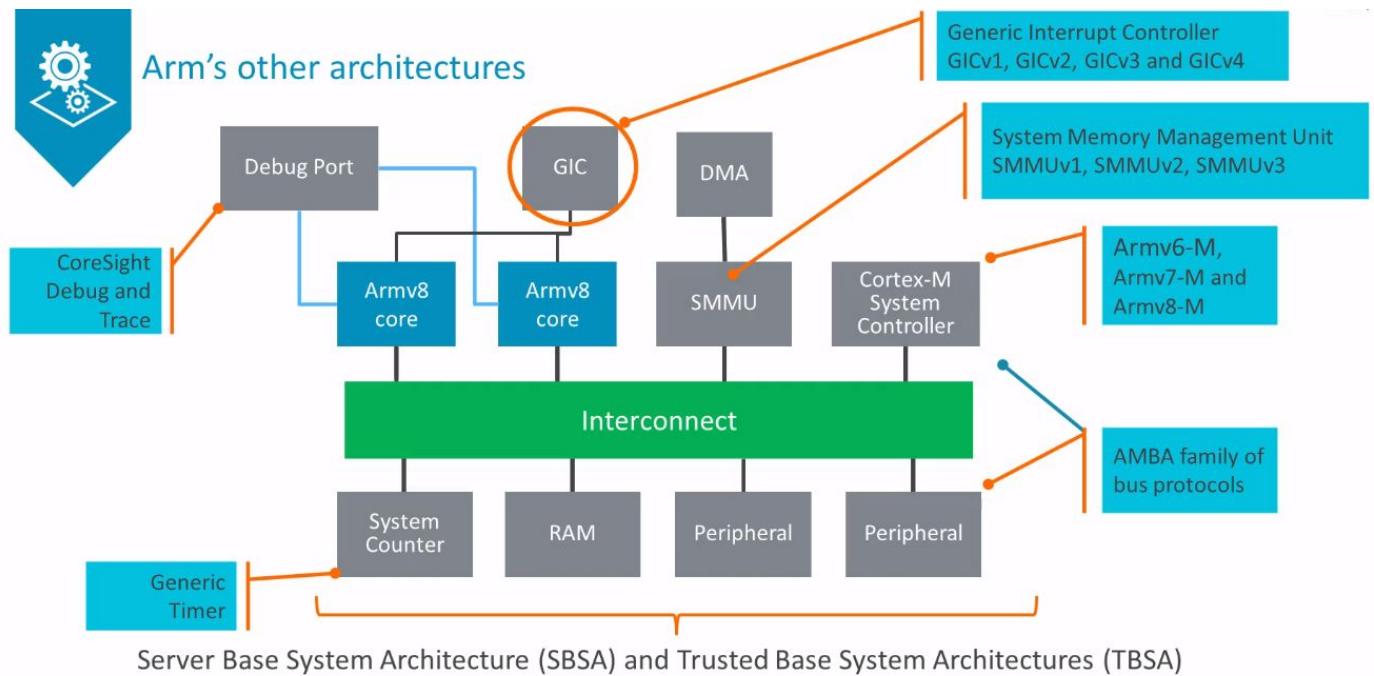
### Arquitectura Mainline:

- Conserva Baseline y todos los fundamentos de **v7-M**.
- Agrega aproximadamente **80** operaciones aritméticas y SIMD de saturación.
- Agrega aprox. **45** instrucciones de punto flotante de precisión simple compatibles con **IEEE754**.
- Agrega soporte de comprobación de límite de pila para el estado No seguro.
- Opcional Instrumentation Trace Macrocell (**ITM**).
- Extensión opcional de procesamiento de señal digital (**DSP**).
- Extensión opcional de punto flotante (**FP**).
- Soporte opcional para la operación de doble precisión.

### Ambos:

- Extensión de seguridad opcional: **TrustZone** para **Armv8-M**.

# Otras arquitecturas de Arm



# Descripción general del conjunto de instrucciones

Este capítulo presenta el conjunto de instrucciones **T32**, que es el único conjunto de instrucciones para **Armv8-M**.

**T32** es una combinación mejorada de los conjuntos de instrucciones originales de **Thumb** y **Thumb2**. En otras palabras, **T32** no solo contiene las instrucciones de **16 y 32 bits** que estaban disponibles en los conjuntos de instrucciones originales de **Thumb**, sino que también incluye algunas instrucciones nuevas que son exclusivas de **Armv8-M**.

A continuación, observamos estos tres grupos de instrucciones con más detalle:

- Cargar y almacenar (Load and store).
- Procesamiento de datos (Data-processing).
- Instrucciones de control de flujo (Flow control instructions).

Para recapitular, **Cortex-M** se puede programar únicamente en **C**. Entonces:

¿Por qué todavía necesitamos saber sobre el conjunto de instrucciones y las instrucciones de **Assembly**?

La respuesta a esta pregunta es que a veces todavía se necesita comprender **Assembly** para el trabajo de validación, testear el comportamiento de corner-case, reiniciar controladores, sistemas operativos, controladores de dispositivos, código crítico optimizado a mano o leer el desmontaje durante las sesiones de depuración. Además, es posible que algunas características de la arquitectura no estén disponibles a través del compilador y, en tales casos, deberá comprender **Assembly** para utilizar las llamadas intrínsecas correctas.

Aquí hay algunos puntos clave a tener en cuenta. En la arquitectura Arm, un **Byte** se refiere a **8 bits**, una **Halfword** es de **16 bits**, una **Word** es de **32 bits** y una **Doubleword** es de **64 bits**. Esto es consistente sin importar el perfil de la arquitectura de brazo del que estamos hablando, ya sea **A**, **R** o **M**.

## Tipos de datos Armv8-M (Data types)



### Data types

Arm v8-M is a 32-bit load/store architecture.

Byte = 8 bits

Halfword = 16 bits

Word = 32 bits

Doubleword = 64 bits

These data types are used for all data operations in the Armv8-M architecture

## Conjunto de registro (register set) del perfil Armv8-M

Hay **16** registros de **propósito general**, denominados **R0 a R15**.

Solo los registros **R0 a R7** son accesibles para todas las instrucciones ya sea de **16 y 32 bits**.

**R8 a R12** no están disponibles para todas las instrucciones de **16 bits**, pero si para las de **32 bits**.

También hay registros de **propósito especial** que no forman parte de los anteriores.

**xPSR**: controla y muestra el entorno de ejecución actual del procesador.

**Control register**: selecciona la pila actual del procesador, el uso de punto flotante y el privilegio de ejecución.

**PRIMASK, FAULTMASK y BASEPRI register**: tienen un significado para el manejo de las excepciones.

Los procesadores **ARMv8-M Mainline** también implementan **MSPLIM** (Límite de MSP) y **PSPLIM** (Límite de PSP) que actúan como protectores del límite inferior de su **Stack Space**.

Los corchetes indican que los registros están almacenados. El banco de registros se refiere a proporcionar múltiples copias de un registro que pueden no estar disponibles en un mismo tiempo. Sin embargo, dependiendo del contexto actual del procesador, el procesador puede estar accediendo a una versión particular del banco de registros.



### Armv8-M profile register set



## Armv8-M Instruction set

Debido a que **Baseline** y **Mainline** incluyen diferentes capacidades y características. Hay diferentes instrucciones disponibles para cada uno.

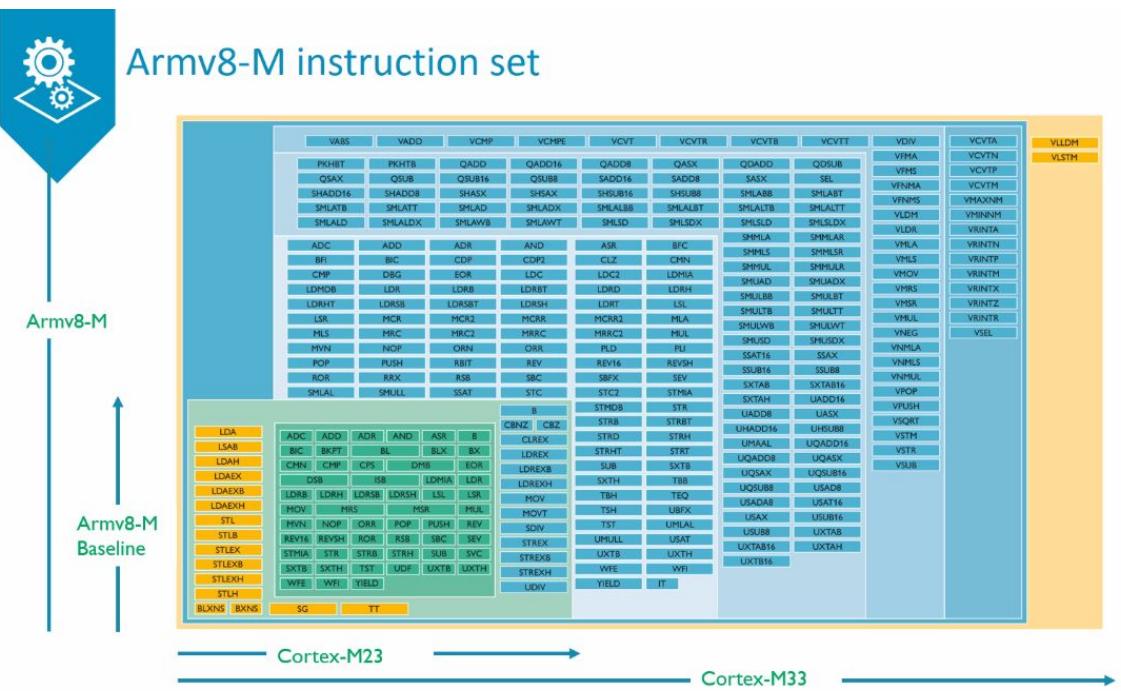
**ARMv8-M Baseline** puede utilizar todas las instrucciones de **16 bits** y algunas de **32 bits**.

**ARMv8-M Mainline** puede utilizar todas las instrucciones de **16 y 32 bits**.

Las instrucciones amarillas indican las nuevas instrucciones agregadas en ARMv8-M que anteriormente no estaban disponibles en **Thumb** o **Thumb2**.

Los tonos diferentes de azul indican las instrucciones incluidas en las extensiones opcionales, por ejemplo: la extensión de seguridad **TrustZone**.

El **Cortex-M23** es un ejemplo de implementación **ARMv8-M Baseline** que utiliza sólo un subconjunto de las instrucciones, y el **Cortex-M33** es un ejemplo de implementación **ARMv8-M Mainline** que puede utilizar completamente todas las extensiones opcionales e instrucciones relacionadas con la extensión disponible en la arquitectura



## Armv8-M load / store architecture

ARM es una arquitectura de carga y almacenamiento, lo que significa que solo hay unas pocas instancias en las que una instrucción puede modificar directamente el contenido de la memoria.

Un valor se carga primero desde la memoria en un registro, se modifica y luego se almacena de nuevo en la memoria, dandonos el nombre de arquitectura de carga y almacenamiento.

Existen algunas variaciones de las instrucciones de carga y almacenamiento, según el tamaño de los datos que están siendo operados.

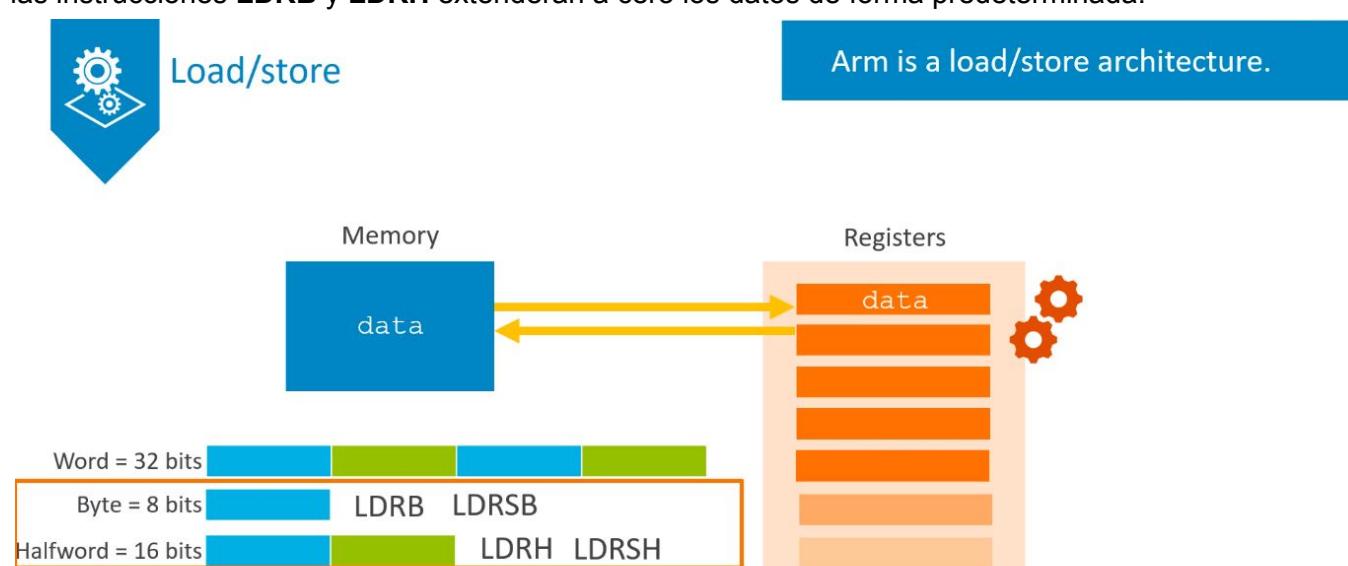
Una instrucción **LDR**, cargará **32 bits**, o **1 Word** desde una ubicación de memoria al registro destino. Una instrucción **STR** hace lo contrario: almacena **1 Word** desde el registro al destino en memoria.

Las instrucciones **LDRB** y **STRB** tienen el mismo mecanismo como las instrucciones **LDR** y **STR**, respectivamente, pero se mueven solo **8 bits** o **1 byte** de datos.

**LDPH y STPH** también son similares pero mueven 16 bits o 1 **Halfword** de datos.

También hay variantes de carga en la que los datos tienen **Signo Extendido** o **Cero Extendido**. Como los registros de propósito general en los que estamos cargando son de **32 bits**, las instrucciones de carga de **bytes** y **halfword** no cargarán datos que completen la totalidad del registro. En estos casos, es posible que queramos decidir si extender con signo o con ceros el ancho de los datos del registro.

Utilizamos **LDRSB** y **LDRSH** para cargas de **bytes** y **halfword** de extensión de signo. Tenga en cuenta que las instrucciones **LDRB** y **LDRH** extenderán a cero los datos de forma predeterminada.



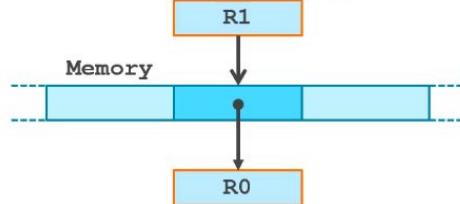
# Modos de direccionamiento Armv8-M



## Addressing modes

### Simple

`LDR R0, [R1]`

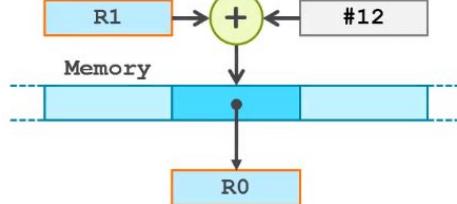


`LDR{<size>} {<cond>} Rd, <address>`

`STR{<size>} {<cond>} Rd, <address>`

### Offset

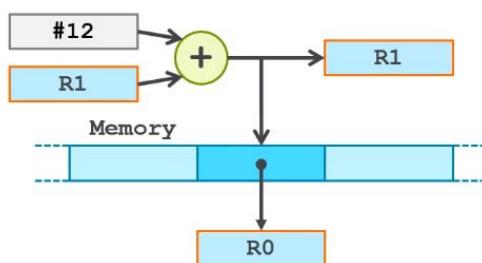
`LDR R0, [R1, #12]`



## Addressing modes

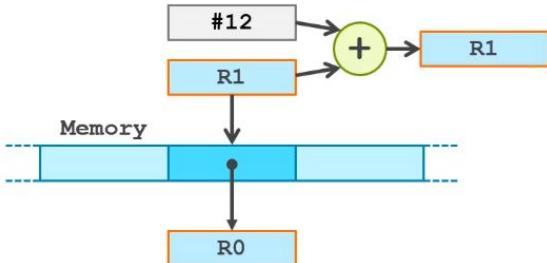
### Pre-indexed

`LDR R0, [R1, #12]!`



### Post-indexed

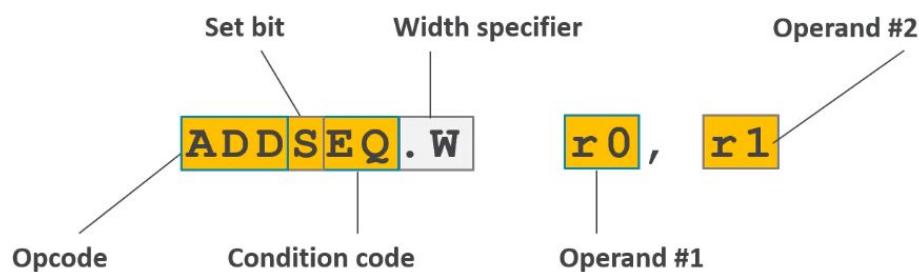
`LDR R0, [R1], #12`



# Instrucciones de procesamiento de datos Armv8-M



## Data processing instructions



	arithmetic		logical		move
manipulation (has destination register)	<b>ADD</b>	<b>SBC</b>	<b>BIC</b>	<b>EOR</b>	<b>MVN</b>
comparison (set flags only)	<b>CMN</b> (ADDS)	<b>CMP</b> (SUBS)	<b>TST</b> (ANDS)	<b>TEQ</b> (EORS)	<b>MOV</b>

## Arm v8-M data processing reference

Arithmetic instruction	Operation	Flags set?	Result saved?
<b>ADDS</b> r0, r1, r2	$r0 = r1 + r2$	Yes, all	Yes
<b>ADCS</b> r0, r1	$r0 = r0 + r1 +$	Yes, all	Yes
<b>SUB</b> r3, r1, r7	$r3 = r1 - r7$	No	Yes
<b>RSB</b> r3, r3, r7	$r3 = r7 - r3$	No	Yes
<b>CMP</b> r1, r7	$r1 - r7$	Yes, all	No
<b>ANDS</b> r0, r1, #0xA0	$r0 = r1 \& 0xA0$	Only N, Z and C	Yes
<b>BIC</b> r0, r1, #0xA0	$r0 = r1$ with bits 5 and 7 cleared	No	Yes
<b>ORRS</b> r0, r1, #0xA0	$r0 = r1   #0xA0$	Only N, Z and C	Yes
<b>TST</b> r1, #0xAB	$r1 \& #0xAB$	Only N, Z and C	No

## Instrucciones de control de flujo (flow control) Arm v8-M

Operaciones como Llamadas a funciones, Devoluciones y Bucles afectan el flujo de un programa, es decir requieren que un procesador pueda ejecutar instrucciones de manera no secuencial.

**B:(Branch):** Bifurca a una dirección diferente sin afectar ningún registro.

**B<cond>:** Es como la instrucción Branch anterior, excepto que puede ser ejecutado condicionalmente.

**BL:** Actualiza el valor del Link Register cuando se toma el Branch. Permite llamadas a función y devoluciones.

**BX:** (Branch and Exchange): Se usa principalmente para retorno de funciones

**BLX:** (Branch with Link and Exchange): Combina la funcionalidad de **BX** y **BL**

**CBZ:** (Compare and Branch): Combinan la instrucción de comparación y la instrucción de bifurcación en una sola instrucción probando el indicador de condición **Z**.



### Flow Control Instructions

Branch instructions vary in size and range

Instruction	Branch Range	
	16-bit	32-bit
<b>B</b>	+/- 2KB	+/- 16MB
<b>B&lt;cond&gt;</b>	-256 to +254 bytes	+/- 1MB

```

B    start
MOVS r0, r1
lab1
    ADDS r0, #1
    ..
    ..
start
    CMP r0, r1
    ..

```

Perform PC-relative branch to label "start"

Continue execution from here

También hay instrucciones no seguras de **BX** y **BLX** que tienen un significado especial en la seguridad, si se implementa la extensión de seguridad **TrustZone**.



## Flow Control Instructions

Branch instructions vary in size and range

Instruction	Branch Range	
	16-bit	32-bit
<b>B</b>	+/- 2KB	+/- 16MB
<b>B&lt;cond&gt;</b>	-256 to +254 bytes	+/- 1MB
<b>BL</b>		+/- 16MB
<b>BX</b>	Any <sup>1</sup>	
<b>BXNS</b>	Any <sup>2</sup>	
<b>BLX</b>	Any <sup>1</sup>	
<b>BLXNS</b>	Any <sup>2</sup>	
<b>CBZ</b>	+4 to 130 bytes	
<b>CBNZ</b>	+4 to 130 bytes	
<b>TBB</b>		+512 bytes
<b>TBH</b>		+128KB

1 Register-based address anywhere in 4GB address space

2 Available if Security Extension is present.  
(must be executed from Secure state)

# Modos, privilegios y pilas

## Descripción general (modos Armv8-M, privilegios y pilas)

Este capítulo presenta la parte Modos, privilegios y pilas del modelo del programador Armv8-M. En comparación con el modelo anterior del programador Armv7-M, es casi lo mismo con la pequeña adición de los registros **MSPLIM** y **PSPLIM**.

## Modos, privilegios y pilas

Con **ARMv8-M** hay 2 modos de operación **Thread Mode** y **Handler Mode**, que definen si el programa que se ejecuta es una aplicación o un controlador de excepciones. También hay 2 niveles de privilegio, estos niveles proporcionan un mecanismo para controlar el acceso a recursos críticos.

**Thread Mode:** tiene un nivel de privilegio programable si se implementa la extensión principal, si la extensión principal no está implementada, entonces la capacidad de programar el nivel privilegiado para **Thread Mode** es la implementación definida.

El nivel de ejecución privilegiado requerido para una ubicación de memoria determinada se puede controlar utilizando la **Memory Protection Unit** o **MPU**.

Los procesadores **ARMv8-M** tienen **2** pilas (Stack) con los **SP** correspondientes, pero pueden tener **4** si se implementa la extensión de seguridad.

El **SP** siempre está alineado con **Words**. Los **SP** se denominan **MSP** y **PSP**.

Si se implementa la extensión de seguridad, los **SP** se duplican para los estados de seguridad, **Secure** and **Non-secure**: **MSP\_S**, **PSP\_S** y **MSP\_NS**, **PSP\_NS**.

**Handler Mode** solo usará el **MSP**, mientras que **Thread Mode** puede usar **MSP** y **PSP** dependiendo de la selección del **SP** en el registro especial de **Control**.



### Modes, privilege, and stacks

Thread	Handler
Normal application reset and processes	Exceptions
Privileged and Unprivileged	Privileged only
Main stack (MSP) Optional Process Stack (PSP)	Main stack (MSP) only

Cuando se ejecuta en **Thread Mode** sin privilegios, el código puede ejecutar una instrucción **Supervisor Call (SVC)** para generar una excepción **SVC** para solicitar acceso a recursos protegidos del **Handler Mode**.

Thread		Handler
MSP + PRIV	PSP + PRIV	MSP + PRIV
MSP + UNPRIV	PSP + UNPRIV	

## Ejemplo de uso del modo procesador (Processor mode)

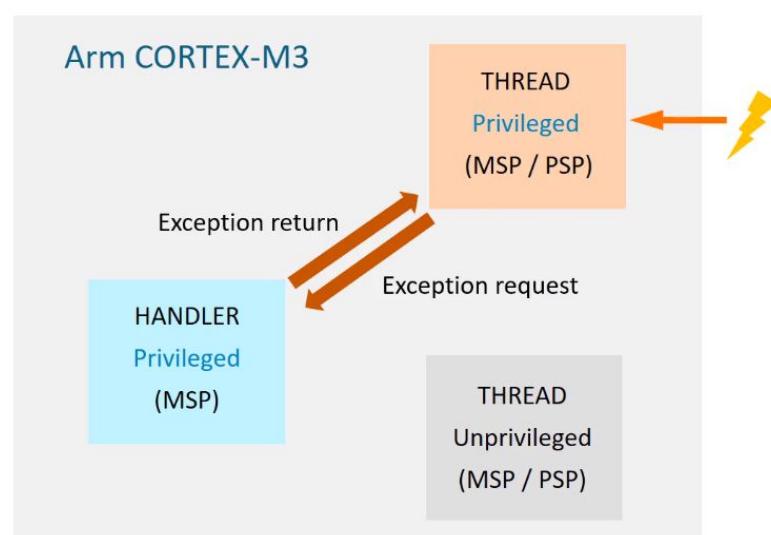
Cuando se produce una excepción interna (o externa), el procesador cambia a **Handler Mode** para ejecutar el código de controlador de excepción (**exception handler code**). Una vez que **exception handler code** finaliza la ejecución, el procesador vuelve al lugar donde estaba realizando una operación de retorno de excepción.



### Processor mode usage example

Processor mode changes  
when an exception occurs in  
Thread mode

Mode does not change when  
an exception occurs in  
Handler mode



## Armv8-M - MSPLIM y PSPLIM

Una nueva adición a la arquitectura Armv8-M son los registros **MSPLIM** de límite de **MSP** y el límite de **PSP** **PSPLIM**. Estos registros son solo de acceso privilegiado y están almacenados entre estados de seguridad. En otras palabras, si se implementa la Extensión de seguridad, habrá cuatro registros de límite de puntero de pila, **MSPLIM\_S**, **PSPLIM\_S**, **MSPLIM\_NS** y **PSPLIM\_NS**

### Descripciones de campo

Las asignaciones de bits de **MSPLIM** son:



Las asignaciones de bits son exactamente las mismas para **MSPLIM** y **PSPLIM**. El campo LIMIT contiene la dirección del límite inferior del registro **SP**. El procesador tomará una excepción si el **SP** apropiado se actualiza a un valor inferior a este límite.

# Arm Memory Model and Protection

## Arm Memory Model and Protection Overview

Este capítulo presenta el modelo de memoria para Armv8-M.

Primero echaremos un vistazo a la disposición de memoria del mapa de memoria predeterminado. Luego, discutiremos los diferentes tipos de memoria disponibles en la arquitectura. Finalmente, aprenderemos sobre la Unidad de Protección de Memoria, o **MPU**.

A diferencia de **ARMv7-M**, en **ARMv8-m** ya no hay un tipo de memoria fuertemente ordenado y el tipo de memoria del dispositivo también es diferente. Además, la interfaz de programación **MPU** también ha cambiado desde **Armv7-M**.

### Armv8-M System address map part 1

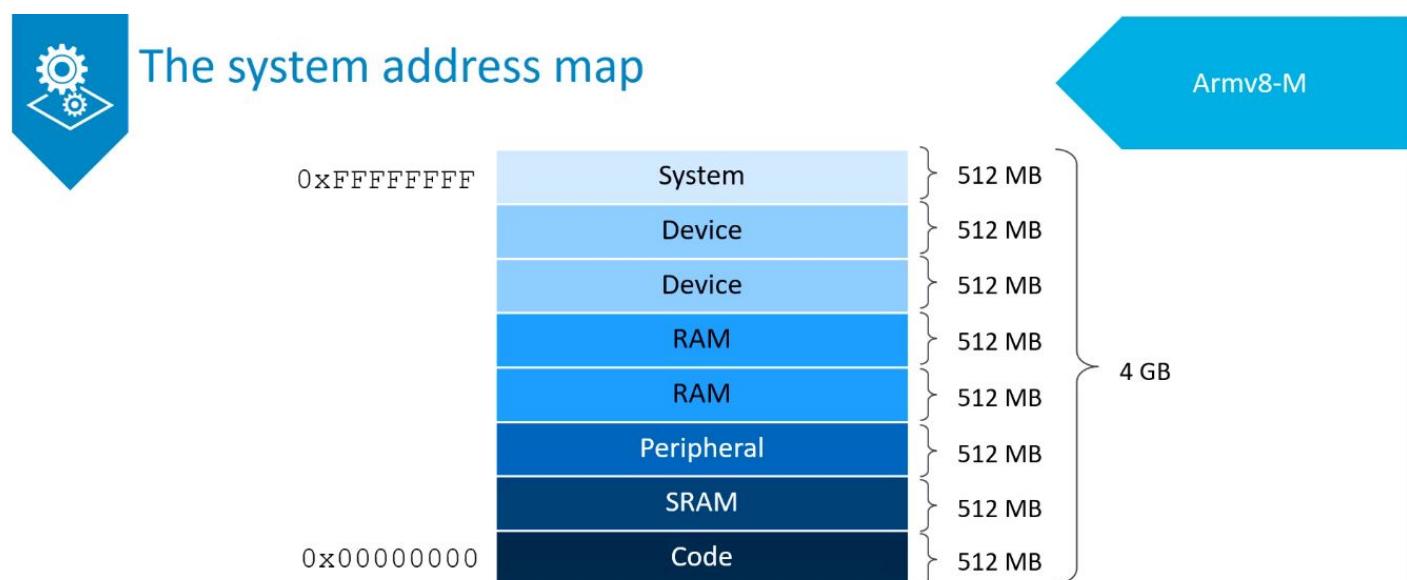
El espacio de direcciones **ARMv8-M** es un espacio de direcciones único y plano de **4 GB**.

**ARMv8-M** implementa la Protected Memory System Architecture (**PMSAv8**), lo que significa todas las direcciones de memorias utilizadas por los procesadores v8-M son direcciones físicas. No hay direccionamiento virtual, a diferencia de los procesadores de perfil A.

El memory map predeterminado se divide en 8 segmentos primarios de **512 MB**.

Los **Peripheral register** en **ARMv8-M** se mapean en memoria para residir en direcciones específicas y, por lo tanto, se puede acceder a través de operaciones estándar de carga y almacenamiento.

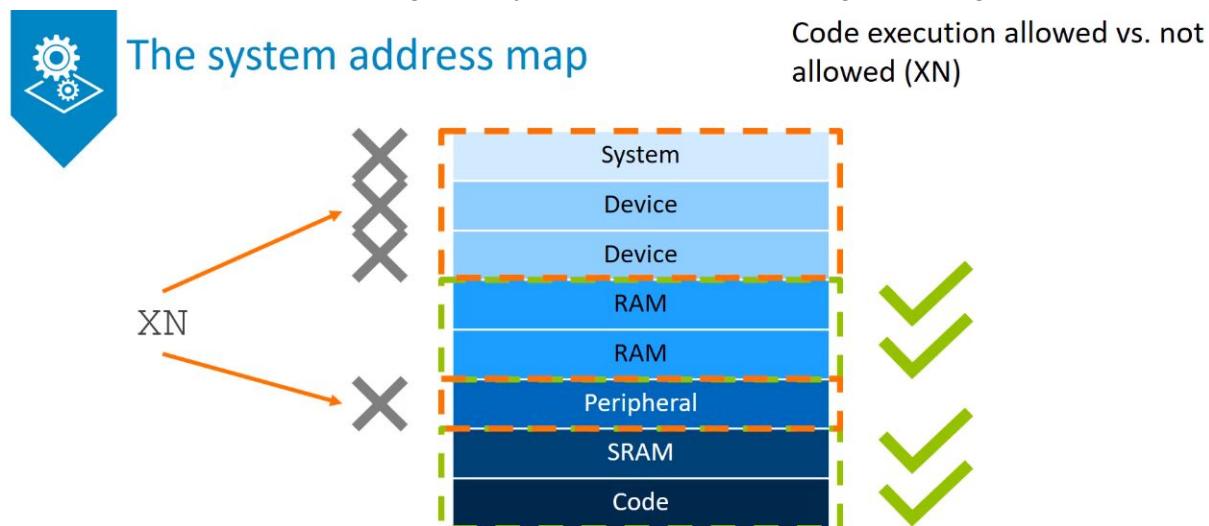
El modelo de memoria de **ARMv8-M** es compatible con **v7-M** y **v6-M**, facilitando la transferencia de software antiguo. Tenga en cuenta que un **ARMv8-M** de implementación **Baseline** solo será compatible con versiones anteriores de **ARMv6-M**, mientras que un **ARMv8-M Mainline** será compatible con ambos modelos de memoria **ARMv6-M** y **ARMv7-M**.



## Arm v8-M System address map part 2

Cada región tiene su propio conjunto de atributos. Un atributo importante informa al procesador si está permitido o no ejecutar código desde una dirección física particular. Por defecto la ejecución de código está permitida desde el **Code**, **SRAM**, y 2 regiones **RAM**.

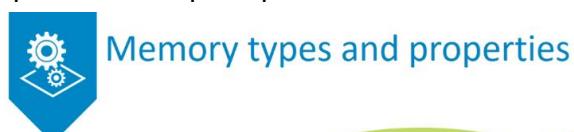
Todas las demás regiones **Peripheral**, las 2 regiones **Device** y la región **System** son tratados por defecto como **eXecute Never (XN)**. Cualquier intento de ejecutar código desde una región marcada como **XN** desencadenaría una falla, el tipo de falla podría ser una **Secure Fault** o **MemManage Fault** dependiendo de si se implementa la extensión de seguridad y si el acceso viola las reglas de seguridad.



## Arm v8-M memory types and properties

Cada región de memoria definida tiene un tipo de memoria especificado. ARMv8-M tiene 2 tipos de memoria mutuamente excluyente: **Normal memory** y **Device Memory**.

En **ARMv8-M** no existe el tipo de memoria **Strongly ordered memory**. Esto se debe a que ahora hay varias versiones de **Device memory** dependiendo de la combinación específica de los atributos de memoria del dispositivo. Un tipo específico de **Device memory** es equivalente a la antigua **Strongly ordered memory**.



Hay otros atributos que definen aún más la región de memoria. Estos son **Access Permission**, **Execute Permission**, **Shareability** y **Cacheability**. Hasta ahora hemos hablado de los permisos de acceso (**Access permissions**) y los permisos de ejecución (**Execute permissions**).

**Access permissions** define si una región de memoria requiere acceso con privilegios. De lo contrario **non-privileged code** también puede acceder a esta región.

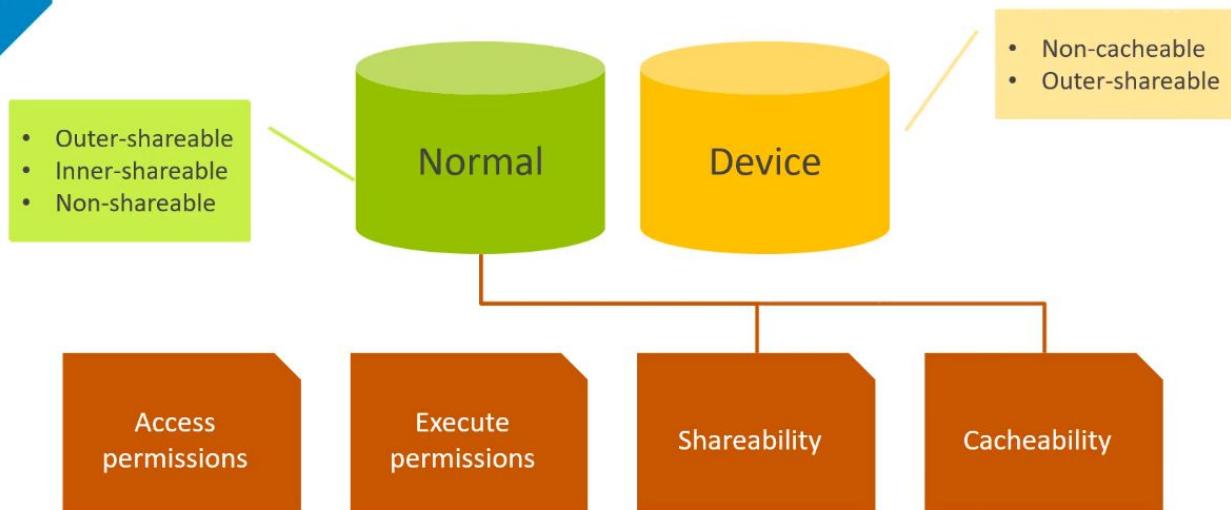
**Execute permissions** se refiere a si el código se puede ejecutar en esta región o si está marcado como **XN**. Una región **Normal memory** también se puede marcar como una **Outer-Shareable**, **Inner-shareable**, o **Non-shareable**. Cada uno tiene un significado especial para definir la observabilidad de esa región particular de memoria dentro del sistema.

El atributo **Cacheability** determina si una región **Normal memory** puede residir dentro de un cache.

**Shareability** y **Cacheability** solo se aplica realmente a la **Normal memory**, como regiones de memoria del dispositivo (**Device memory regions**) que siempre son tratadas como **Non-Cacheable** y **Outer-Shareable**.



## Memory types and properties



### Armv8-M - Normal memory

**Normal memory** tiene varias cualidades importantes que la hacen flexible y adecuada para la ejecución de código. Los accesos a la **Normal memory** se pueden reordenar por el procesador. En otras palabras, significa que el procesador puede emitir accesos a regiones de **Normal memory** fuera del orden del programa. **Normal memory** puede leerse y escribirse repetidamente sin efectos secundarios.

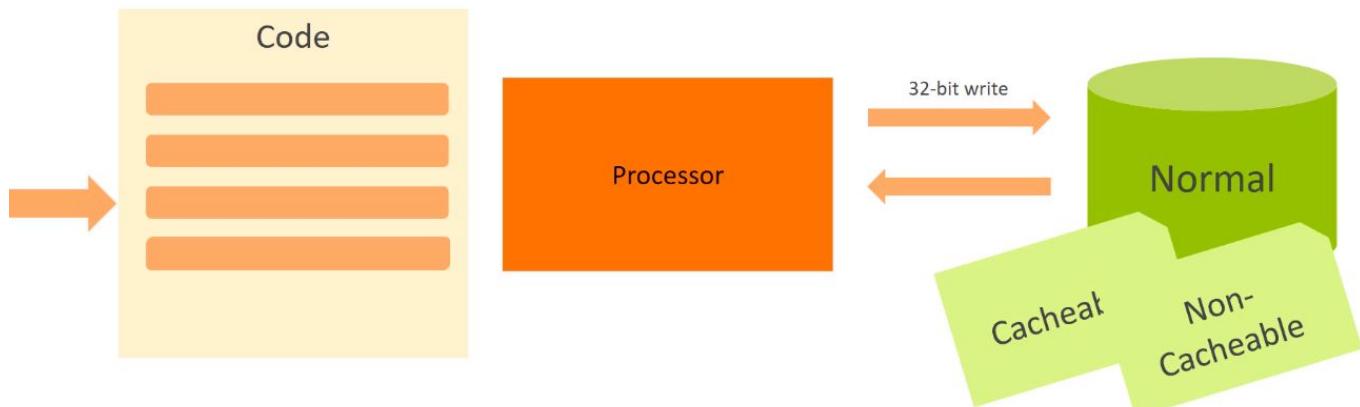
Los accesos a la **Normal memory** también se pueden combinar.

Los accesos no alineados también pueden ser compatibles con la **Normal memory**, como en el caso de **Cortex-M33**.

La **Normal memory** también se puede marcar como **Cacheable** o **Non-Cacheable** y atributos **Cacheability** asignados. Finalmente, se puede acceder a la **Normal memoria** de forma especulativa. Esto significa que los datos o instrucción fetches puede ocurrir en una **Normal memory** sin tener que ejecutar instrucciones haciendo referencia explícita a la operación.



### Normal memory



# Cacheability and Shareability

Aquí explicamos los conceptos de **Cacheability** y **Shareability** en Armv8-M. Para obtener todos los detalles, consulte el capítulo Modelo de memoria en el [Manual de referencia de arquitectura Armv8-M](#)

## Cacheability

La arquitectura proporciona dos niveles conceptuales de caché:

- El caché interno (Inner cache).
- El caché externo (Outer cache).

Arm Architecture no define formalmente que debe ser la **caché interna** u **externa**. Sin embargo, normalmente la caché interna es pequeña y está definida por la implementación "lo más cerca posible del procesador". Y, por lo general, el caché externo es mucho más grande y contiene información que debe almacenarse en caché, pero no necesariamente tiene que ser así.

Los atributos de **Cacheability** son:

- Non-cacheable.
- Write-Through Cacheable.
- Write-Back Cacheable.

Los atributos **Write-through** y **Write-back** también pueden tener las siguientes sugerencias de asignación de caché, que son independientes para los accesos de lectura y escritura:

- Read-Allocate, Transient Read-Allocate, o No Read-Allocate.
- Write-Allocate, Transient Write-Allocate, o No Write-Allocate.

Tenga en cuenta que la arquitectura no requiere una implementación para hacer uso de las sugerencias de asignación de caché.

## Shareability

Hay dos dominios de **Shareability** conceptual:

- The Inner Shareability domain.
- The Outer Shareability domain.

Al igual que con **Cacheability**, la arquitectura nuevamente no define formalmente que **Inner** y **Outer** tiene que ser. De nuevo, está definido por la implementación.

La arquitectura requiere que los accesos a una ubicación de shareable memory sean coherentes dentro del dominio de Shareability de esa ubicación. Esto significa que todos los observadores en un dominio de **Inner Shareability** son datos coherentes para los accesos de datos a la memoria que tienen el atributo Inner-shareable Shareability. Del mismo modo, todos los observadores en un dominio de Outer Shareability son datos coherentes para los accesos de datos a la memoria que tienen el atributo Outer-shareable Shareability.

Cada observador sólo puede ser miembro de un solo dominio de **Inner Shareability** y un solo dominio de **Outer Shareability**

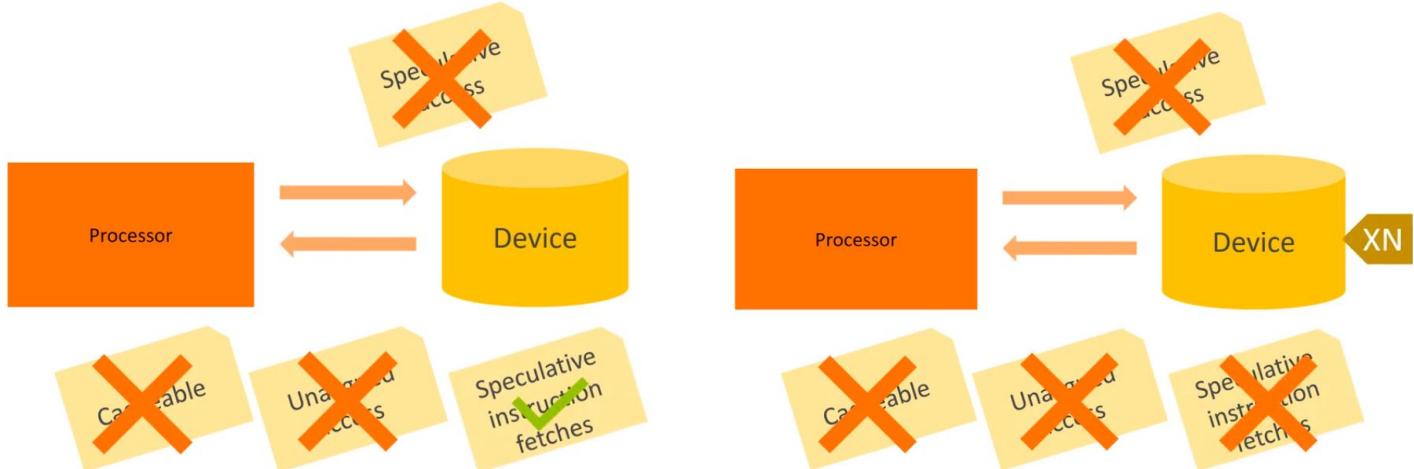
El mapa de memoria predeterminado contiene normal cacheable regions que ya tienen asignado el atributo de shareability. Pero si utiliza la **MPU** para definir sus propias regiones de **Normal memory** personalizadas, debe asignar a cada región uno de estos atributos de Shareability:

- Non-shareable.
- Inner-shareable.
- Outer-shareable.

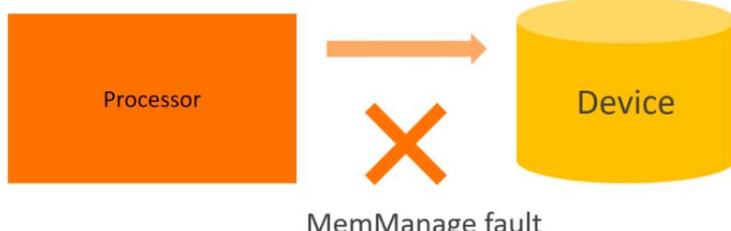
## Arm v8-M - Device memory

Los accesos a la **Device memory** del dispositivo pueden tener efectos secundarios, lo que hace adecuado para periféricos y dispositivos de entrada y salida. La **Device memory** no es **cacheable** y los accesos **unaligned** no son compatibles y generarán una excepción unaligned **UsageFault**.

No se pueden realizar acceso de datos especulativos a la **Device memory**. Sin embargo se pueden realizar búsquedas de instrucciones especulativas, a menos que la región de memoria también sea marcada como **execute never (XN)**.



Una de dos cosas pueden suceder cuando se realiza por error una búsqueda (fetch) de instrucciones desde una región **Device memory**. Dependiendo de la implementación, el procesador podría tratar la búsqueda (fetch) como si fuera una ubicación en la **Normal Non-cacheable memory**, o tomar una **MemManage fault**.



Hay algunas versiones de la **Device memory** definidas aún más por los **Device memory attributes**.

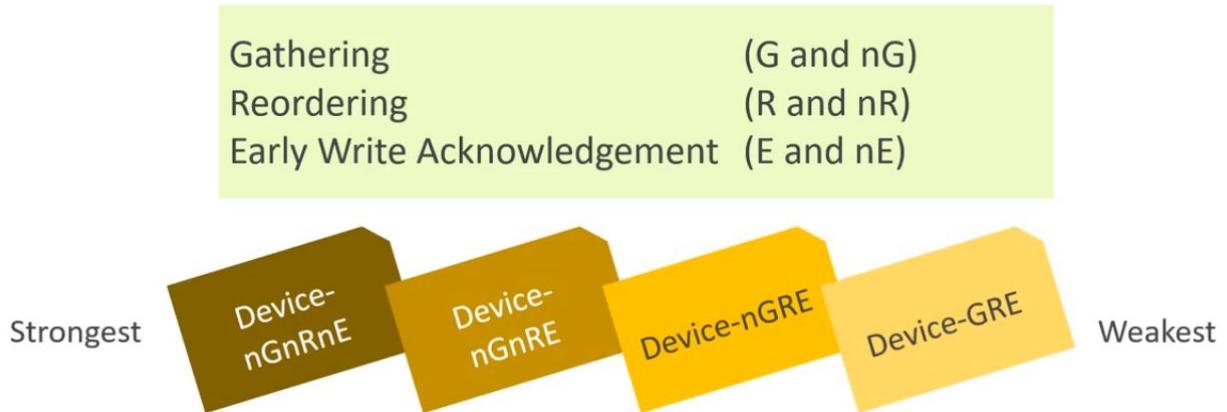
Estos **Device memory attributes** incluyen **Gathering** (recopilacion) (**G** y **nG**), **Reordering** (reordenamiento) (**R** y **nR**) y **Early Write Acknowledgement** (reconocimiento temprano) (**E** y **nE**).

Las posibles combinaciones de estos 3 atributos se enumeran aquí desde los tipos de orden más fuerte hasta los más débiles.

Se puede pensar en el antiguo tipo **Strongly ordered memory** siendo reemplazada por el tipo **Device memory** más fuerte **Device nGnRnE** que no permite **gathering**, **reordering** y **Early Acknowledgement**.

La versión **Arm v6-M** y **Arm v7-M** de la **Device memory** se refiere al tipo de memoria **Device-nGnRE**.

Como se puede ver, 2 versiones de **Device memory** más flexibles, se han agregado en **Arm v8-M**



# Atributos de Device memory

A cada región de memoria del dispositivo se le asigna una combinación de atributos de **Device memory**. Como hemos visto, los atributos son **Gathering (G y nG)**, **Reordering (R y nR)** y **Early Write Acknowledgement (E y nE)**. El prefijo **n** indica que el atributo no se aplica.

Aquí se describen las cualidades más importantes para cada atributo. Sin embargo, para ver sus descripciones completas, consulte el capítulo **Memory model** en el [Manual de referencia de arquitectura Armv8-M](#).

## Atributo G

Si múltiples accesos de lectura o escritura del mismo tipo de memoria son para:

- La misma ubicación, entonces se pueden fusionar (gathered) en una sola transacción si todos tienen el atributo G.
- Diferentes ubicaciones, entonces también pueden fusionarse (gathered) en una sola transacción si todas tienen el atributo G

## Atributo nG

Los accesos múltiples a una ubicación de memoria con el atributo **nG** no se pueden fusionar (gathered) en una sola transacción.

## Atributo R

El atributo **Reordering** no impone restricciones ni relajaciones.

## Atributo nR

Con este atributo, si el acceso es a:

- Periférico, luego llega al periférico en orden de programa. Si hay una combinación de accesos a **Device nGnRE** y **Device-nGnRnE** en el mismo periférico, estos accesos ocurren en orden de programa.
- No periférico, entonces este atributo no impone restricciones ni relajaciones.

## Atributo E

Este atributo no impone restricciones ni relajaciones.

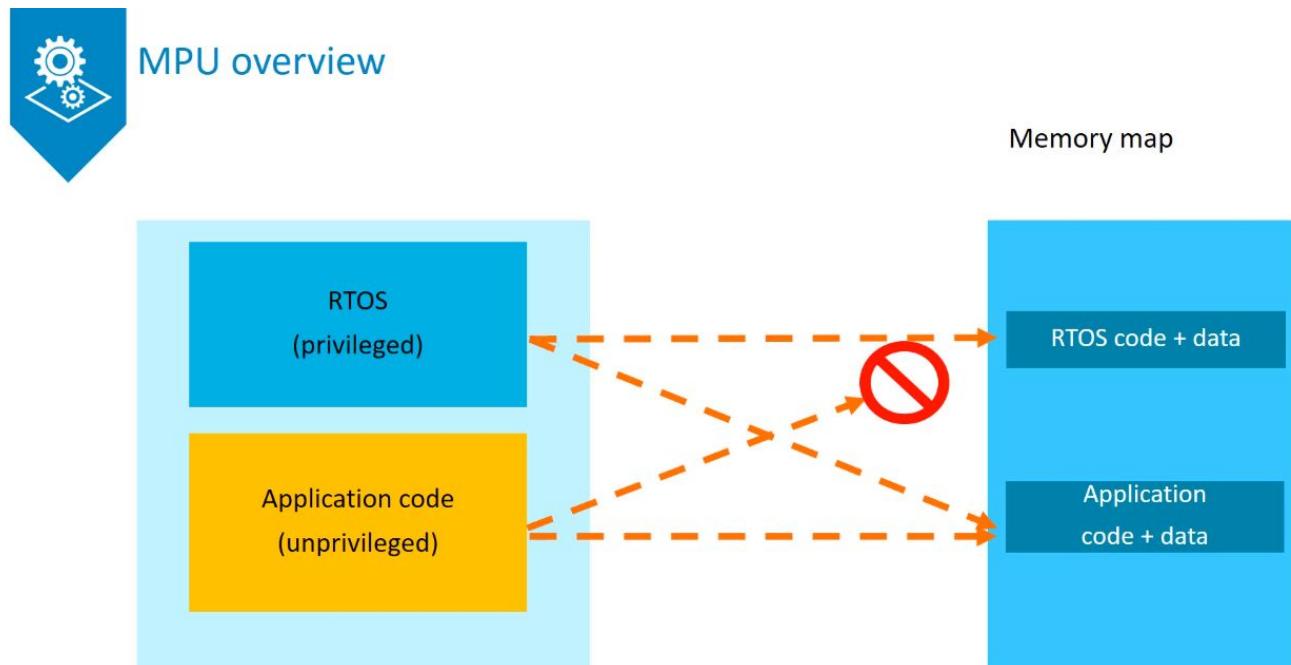
## Atributo nE

La asignación del atributo **nE** recomienda que solo el **endpoint** del acceso de escritura retorne un **write acknowledgement** del acceso, y que ningún **earlier point** en el sistema de memoria retorne un **write acknowledgement**.

El atributo **E** se trata como una sugerencia. Arm recomienda encarecidamente que esta sugerencia no sea ignorada por un processing element (**PE**), sino que esté disponible para su uso por el sistema.

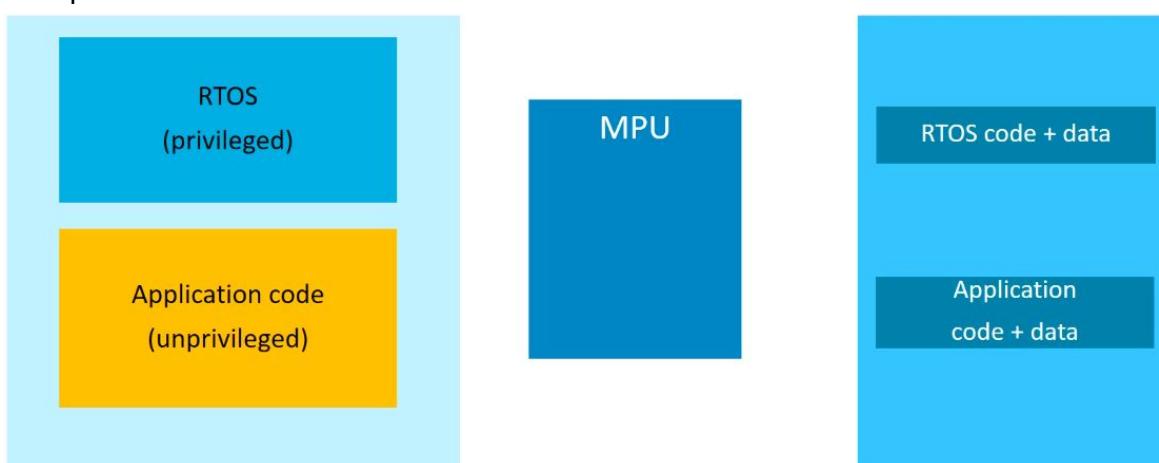
# Descripción general Armv8-M Memory Protection Unit (MPU)

El mapa de direcciones del sistema predeterminado puede ser adecuado para algunas aplicaciones. Sin embargo los sistemas más complejos pueden requerir flexibilidad adicional y protección adicional que puede ofrecer un mapa personalizado. Por ejemplo Un sistema Operativo corriendo en ejecución privilegiada, necesitará acceso tanto al Sistema Operativo como a los códigos y datos de la Aplicación. Pero mientras que una aplicación debe tener acceso libre a su propio código asociado y regiones de datos, no se le debe dar acceso a códigos y datos asociados con el sistema operativo u otras aplicaciones.



ARMv8-M soporta Protected Memory System Architecture (**PMSAv8**) donde está el espacio de direcciones protegido por una Unidad de Protección de Memoria (**MPU**).

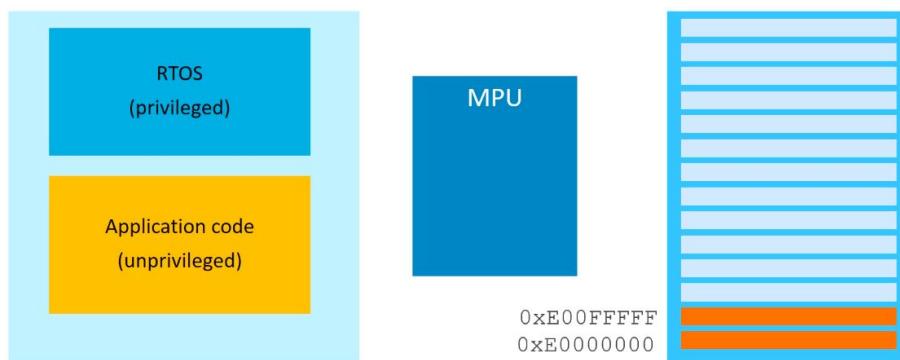
La **MPU** es una característica opcional de la arquitectura, se puede tener hasta dos **MPU** separadas si la **Security Extension** está implementada; uno para cuando el procesador está en **Non-secure state** y otro para cuando el procesador está en **Secure state**.



La **MPU** se utiliza para definir un nuevo mapa de direcciones para reemplazar el mapa de direcciones predeterminado. Por ejemplo: con la **MPU** se puede partitionar la memoria en regiones configurables. Dentro de cada región, puede definir su tipo, atributos y permisos de acceso.

Sin embargo la **MPU** no puede reasignar la región del **Private Peripheral Bus (PPB)** de este rango de direcciones.

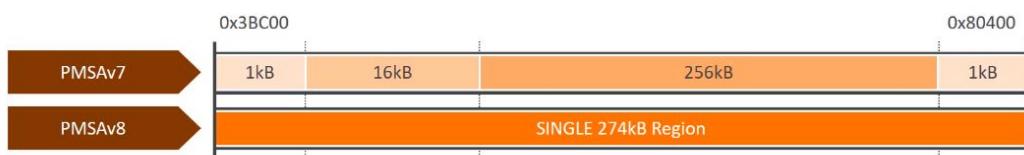
Esta región se tratará como una como se define en el mapa de direcciones del sistema predeterminado, y los accesos a esta región no están sujetos a verificación de **MPU**. También tenga en cuenta que la **MPU** no realiza ninguna verificación de seguridad. Esto se realiza a través de la **Security Attribution Unit (SAU)**.



Una diferencia importante para la **MPU** de **ARMv7-M** y **ARMv8-M** es que las regiones superpuestas y las subregiones ya no son soportadas. También hay una usabilidad y flexibilidad mejorada en **PMSAv8** versus **PMSAv7** en que las regiones definidas por **MPU** ya no necesitan estar alineadas por su tamaño y como una potencia de 2. Las regiones definidas por **MPU** en **ARMv8-M** simplemente deben alinearse con un múltiplo de 32 bytes.



### MPU regions



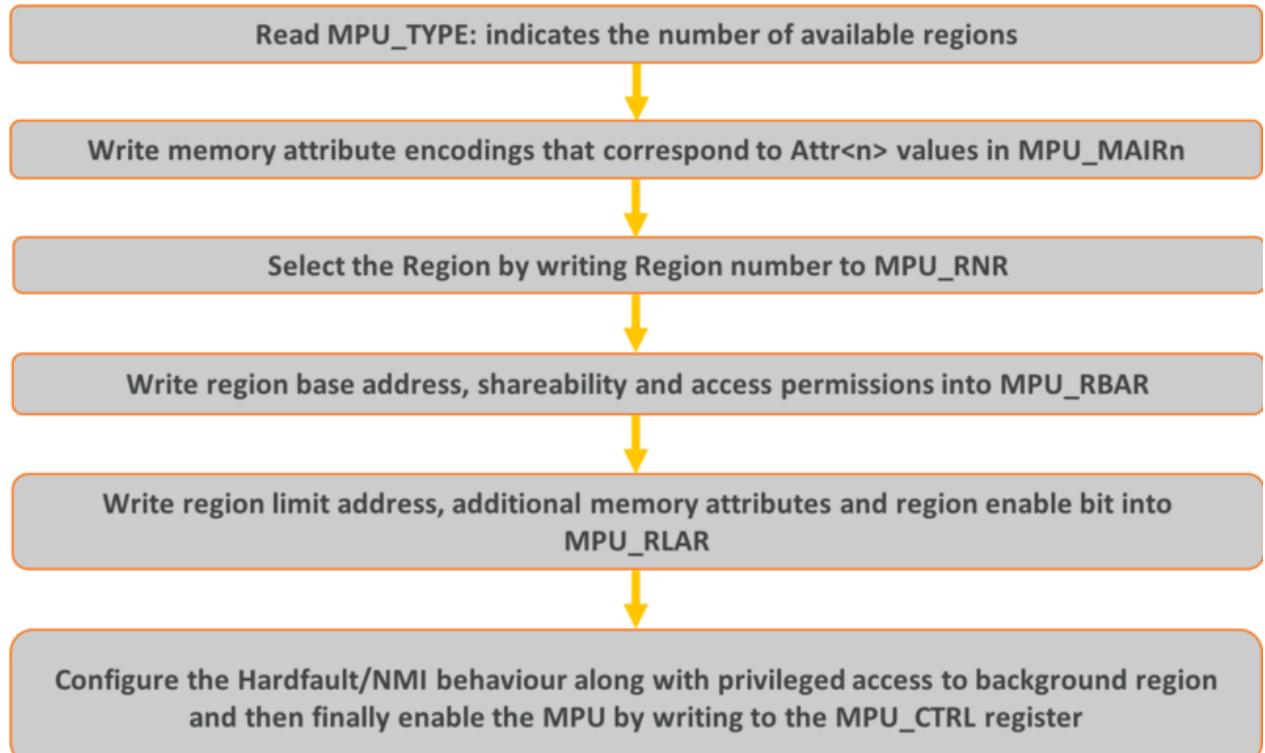
## Programando la MPU

En resumen, la interfaz de programación para la **MPU** se ha simplificado. En esta tabla, puede ver los registros de **MPU** que deben programarse para crear un mapa de memoria personalizado utilizando el software para habilitarlo.

Address	Name	Type	Description
0xE000ED90	MPU_TYPE	RO	MPU Type Register
0xE000ED94	MPU_CTRL	RW	MPU Control Register
0xE000ED98	MPU_RNR	RW	MPU Region Number Register
0xE000ED9C	MPU_RBAR	RW	MPU Region Base Address Register
0xE000EDA0	MPU_RLAR	RW	MPU Region Limit Address Register
0xE000EDA4	MPU_RBAR_A1	RW	MPU Region Base Address Register Alias 1
0xE000EDA8	MPU_RLAR_A1	RW	MPU Region Limit Address Register Alias 1
0xE000EDAC	MPU_RBAR_A2	RW	MPU Region Base Address Register Alias 2
0xE000EDB0	MPU_RLAR_A2	RW	MPU Region Limit Address Register Alias 2
0xE000EDB4	MPU_RBAR_A3	RW	MPU Region Base Address Register Alias 3
0xE000EDB8	MPU_RLAR_A3	RW	MPU Region Limit Address Register Alias 3
0xE000EDC0	MPU_MAIR0	RW	MPU Memory Attribute Indirection Register 0
0xE000EDC4	MPU_MAIR1	RW	MPU Memory Attribute Indirection Register 1

Para obtener más información sobre lo que controla cada registro, consulte la sección Especificación de registro en el [Manual de referencia de arquitectura Armv8-M](#). La tabla muestra que la **MPU Region Base Address** y los registros **Region Limit Address** tienen un alias un total de 4 veces. Esto es para agilizar el proceso de programación de modo que pueda usar una instrucción **Store Multiple** o una **memcpy** (memory copy) para escribir los 8 registros a la vez. En otras palabras, el programador puede programar 4 regiones con 1 instrucción **STM**.

Aquí hay un diagrama de flujo de una secuencia de programación de ejemplo para la **MPU**:



Tenga en cuenta que debe deshabilitar **MPU** al cambiar las configuraciones de **MPU**.

# Excepciones

## Descripción general de las excepciones de Armv8-M

Una cosa que diferencia a la arquitectura de perfil **M** de los perfiles **A** y **R** es su modelo de excepciones único y eficiente. Este capítulo trata en detalle el modelo de excepciones, incluidos los diferentes tipos de excepciones disponibles, estados de excepción, prioridades, apilamiento y fallas.

El modelo de excepciones no ha cambiado en su mayoría de las arquitecturas **M** anteriores, pero hay algunas adiciones debido a la **Security Extension**. Además, tenga en cuenta que existen ligeras variaciones dependiendo de si está utilizando **Main Extension** o no.

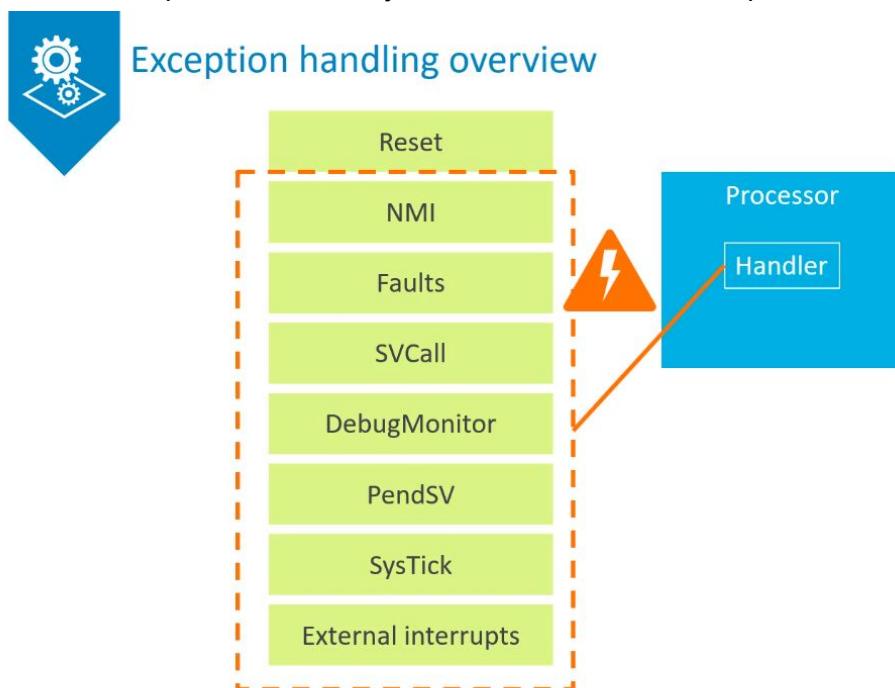
### ¿Qué es una excepción?

En palabras simples, las excepciones son eventos que causan cambios en el flujo del programa fuera de la secuencia de código normal. Cuando sucede, el programa que se está ejecutando actualmente se suspende y se ejecuta el controlador de excepciones asociado con el evento. Los eventos pueden ser externos o internos. Cuando un evento es de una fuente externa, se conoce comúnmente como interrupción o **IRQ**. Casi todos los procesadores modernos admiten excepciones e interrupciones. En microcontroladores típicos, las interrupciones también se pueden generar utilizando periféricos en chip o por software.

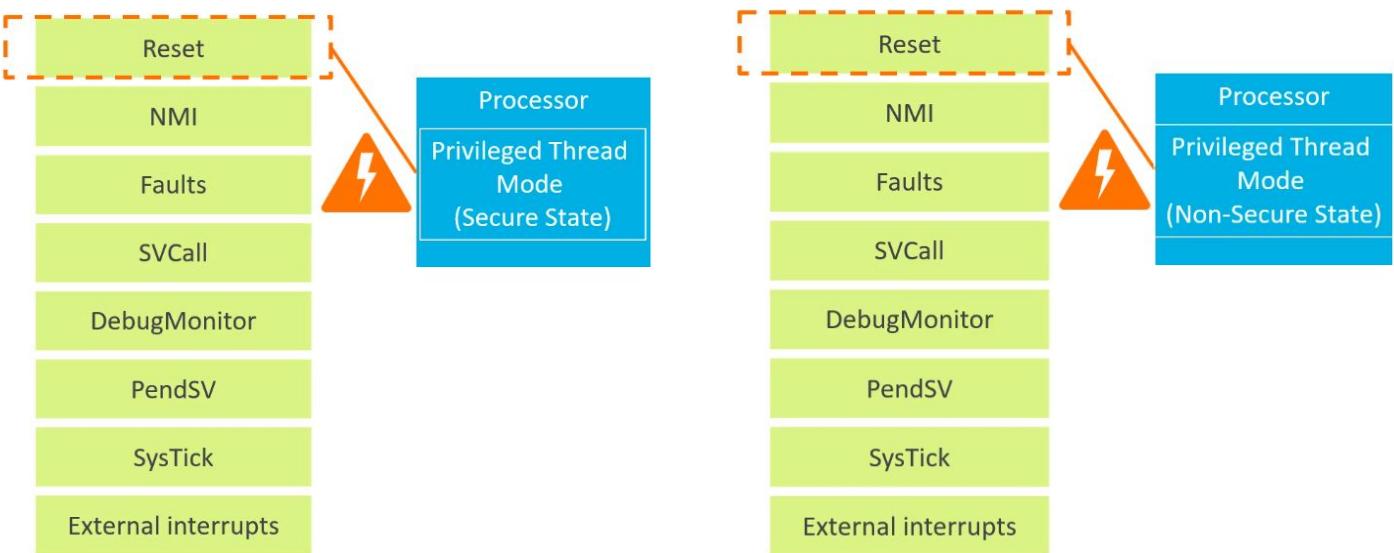
## Descripción general del manejo de excepciones de Armv8-M

En **ARMv8-M** las excepciones se incluyen en los siguientes tipos. Desde la prioridad más alta hasta la más baja, tenemos: **Reset**, **Non-Maskable Interrupt (NMI)**, **Faults**, **SVcall**, **DebugMonitor** (Main Extension Only), **PendSV**, **SysTick** y **External Interrupts (IRQ)**.

Todas las excepciones se manejan en **Handler Mode**, excepto **Reset**.

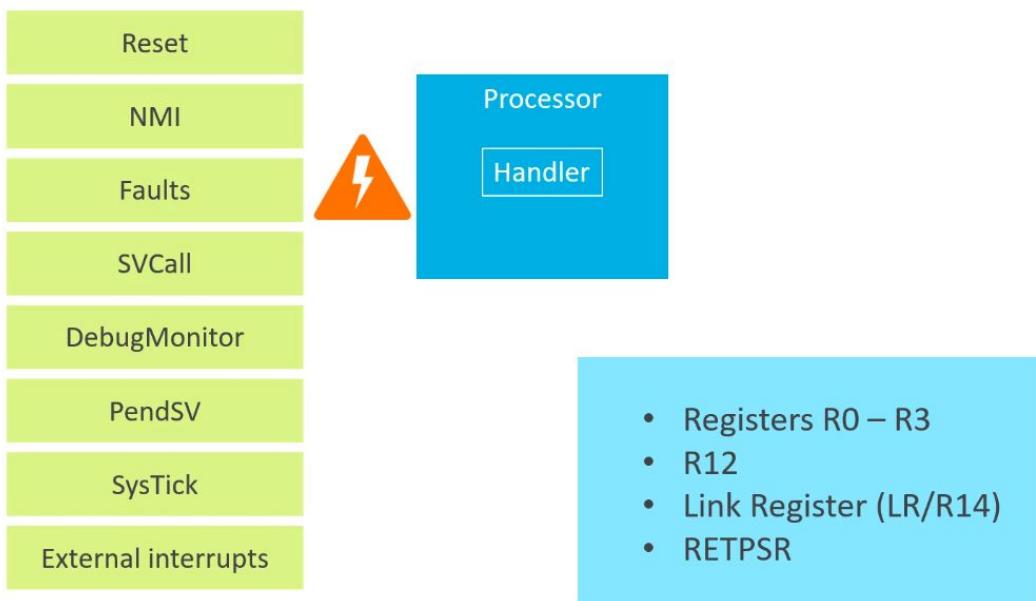


El procesador se establece en **Privileged Thread Mode** en estado **Seguro** si la **Security Extension** es implementada, o **Privileged Thread Mode** en estado **No seguro** si no ha implementado la **Security Extension**.



En todos los demás casos, una excepción lleva al procesador al **Handler Mode** si el procesador se está ejecutando actualmente en **Thread Mode**, o permanece en **Handler Mode** si este es el modo actual del procesador.

Como en las arquitecturas M anteriores, **ARMv8-M** guarda y restaura automáticamente el contexto de estado del procesador en la pila, en la entrada excepción y en el retorno de excepción. El **state context** son **8 Words** de **32 bits** que incluyen: Registros **R0-R3, R12, LR** y **RETPSR**. El **RETPSR** es un poco diferente del **xPSR** en **ARMv7-M**. El **RETPSR** consta no solo del compuesto de **APSR, EPSR e IPSR** sino también el **SFPA**, bit del registro de **CONTROL**. Además **ARMv8-M** ya no da la opción de alinear la pila a **4 bytes** en la entrada de excepción. El **SP** se vuelve a alinear automáticamente a una alineación de 8 bytes en una entrada de excepción si aun no lo ha hecho.



## Special Program Status Registers (SPSR)

El **Combined Program Status Register**, o **XPSR**, contiene información sobre el entorno actual del procesador y consta de **APSR, EPSR e IPSR**. Acceder al **XPSR** dará una vista compuesta de **32 bits** de los **3** registros en lugar de tener que acceder a cada registro individualmente.

### APSR – Application Program Status Register

El **APSR** contiene los indicadores de condición de la unidad lógica aritmética (**ALU**). Estas son las banderas de condición **N, Z, C, V, Q**, así como las banderas **GE** (mayores o iguales), que se utilizan para la extensión **DSP**. Recuerde que los indicadores de condición se establecen mediante instrucciones durante la ejecución del código y se utilizan para operaciones condicionales y ramas condicionales.

## IPSR – Interrupt Program Status Register

El campo de excepción en el **IPSR** contendrá el número de excepción o la excepción que se está ejecutando actualmente, o **cero** para el **Thread Mode**.

## EPSR – Execution Program Status Register

El **EPSR** contiene la información del entorno de ejecución, incluidos el bit **T** y los bits **IT / ICI**. El bit **T** indica que el proceso está en estado de ejecución **T32**. Dado que los procesadores de perfil Cortex-M no admiten el estado de ejecución de Arm, este bit siempre debe establecerse. De lo contrario, la ejecución de cualquier instrucción generará un estado inválido **INVSTATE UsageFault**. El campo **IT / ICI** se usa para la ejecución condicional (**if-then**) o la interrupción de continuación (**ICI**). Dado que las instrucciones **T32** permiten la ejecución condicional con la ayuda de instrucciones **If-then**, el campo **IT** se utiliza para realizar un seguimiento de la condición actual y la posición de una secuencia de bloque **IT**.

Las instrucciones de carga y almacenamiento multiciclo se pueden interrumpir y continuar después de que se maneja la excepción. El campo de bits se utilizará como bits **ICI** en este caso.

Tenga en cuenta que, dado que tanto la funcionalidad de **IT** como **ICI** comparten el mismo campo en el **EPSR**, las cargas y almacenamiento multiciclo no pueden reanudarse dentro de un bloque de **IT**.

## Modelo de prioridad Armv8-M

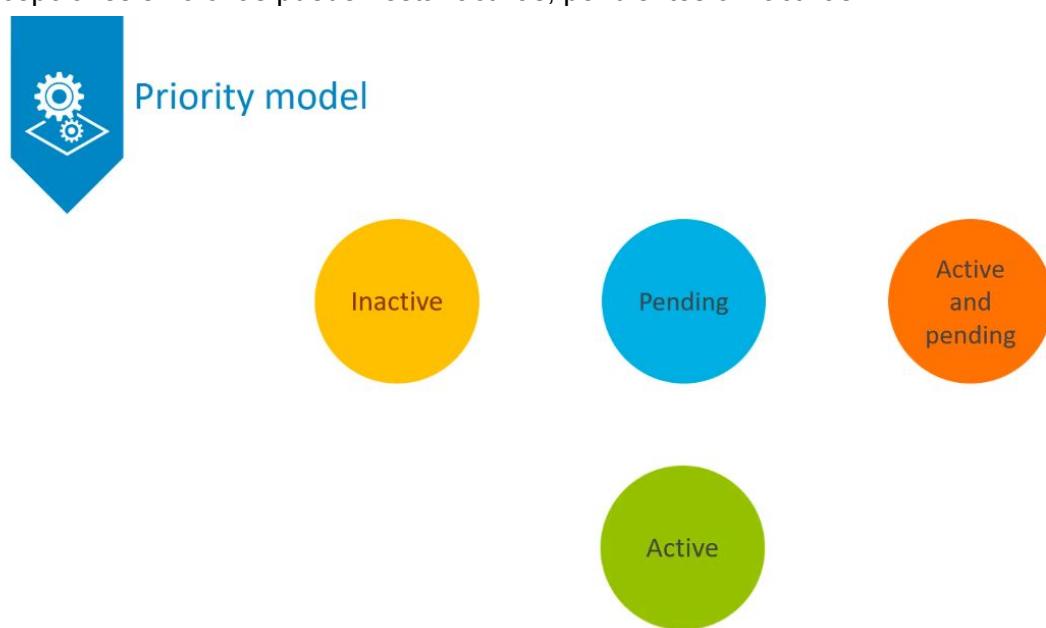
Una excepción, que no sea **Reset** tiene los siguientes estados posibles:

**Active**: que es una excepción que está siendo manejada o fue manejada. Si fue manejada, el **handler** fue reemplazado por **handler** para una excepcion de mayor prioridad.

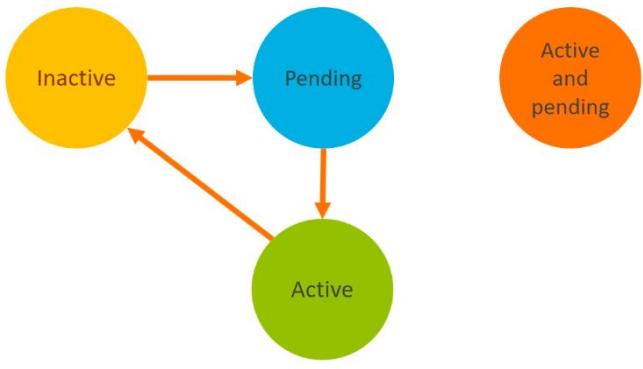
**Pending**: una excepción pendiente ha sido generada pero no está activa.

**Inactive**: la excepción no se ha generado.

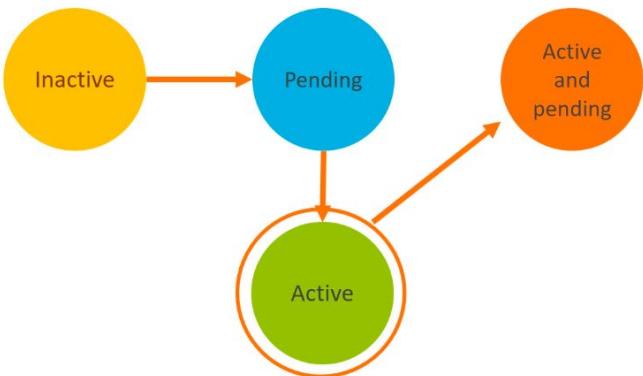
**Active and Pending**: una instancia de la excepción está activa, y la segunda instancia de la excepción está pendiente. Solo las excepciones asíncronas pueden estar activas y pendientes (Active and Pending). Las excepciones síncronas pueden estar activas, pendientes o inactivas.



Por lo general, una excepción pasa de un estado **Inactivo** a un estado **Pendiente**, y luego inmediatamente al estado **Activo**, si no hay actualmente excepciones activo o pendiente de mayor prioridad. En este simple caso, el estado **Activo** vuelve a **Inactivo** al salir del Exception handler.

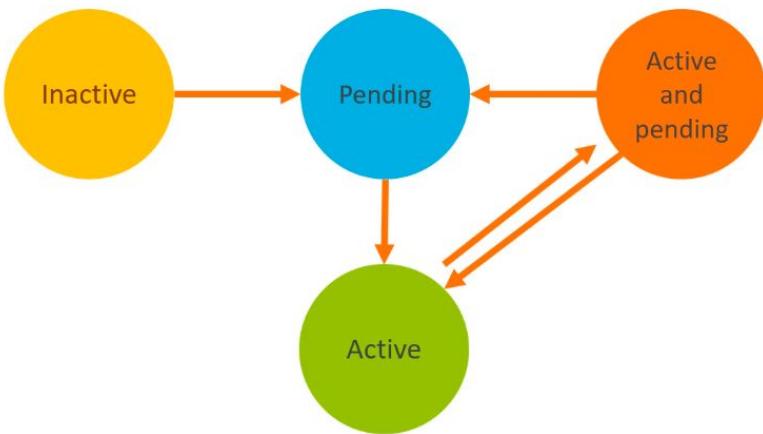


Si se solicita nuevamente una interrupción mientras está actualmente activo, se vuelve activo y pendiente. Cuando una excepción está en el estado **Activo y Pendiente**, el handler software puede borrar la bandera pendiente para mover la excepción al estado **Activo**.



De lo contrario, al salir del handler, ya no está activo, sino pendiente.

Es probable que esto vuelva a pasar de inmediato a **Activo** a menos que llegue otra interrupción de mayor prioridad.



## Armv8-M exception priorities

Name	Exception Number	Exception Priority No.
Interrupts #0 - #495 (N interrupts)	16 to 16 + N	0-255 (programmable)
SysTick	15	0-255 (programmable)
PendSV	14	0-255 (programmable)
SVCall	11	0-255 (programmable)
SecureFault	7	0-255 (programmable)
Usage Fault	6	0-255 (programmable)
Bus Fault	5	0-255 (programmable)
Memory Management Fault	4	0-255 (programmable)
Hard Fault (Secure HardFault)	3	-1 (programmable -1 or -3)
Non Maskable Interrupt (NMI)	2	-2
Reset	1	-4

Lowest  
↑  
Highest

¿Cómo determina el procesador qué excepción manejar primero o si una excepción activa se adelanta por otra con una prioridad más alta?

En Armv8-M Architecture, cada tipo de excepción tiene un número de excepción asociado y un número de prioridad de excepción. Las excepciones con el número de prioridad más baja indican que tiene la prioridad más alta. Notarás que en este caso, **Reset** tiene el número de prioridad de excepción de **-4**, lo que hace que tenga la máxima prioridad de todas las excepciones. **Reset, NMI y Hard Fault** son las excepciones de mayor prioridad y las únicas excepciones con prioridad fija.

**Secure HardFault** puede ser "**-1**" o "**-3**", controlado por el campo **BFHFNMINS** en el registro **AIRCR**.

La arquitectura admite **3 - 8 bits** para **Mainline** o **4 bits** para **Baseline** para que el usuario establezca la prioridad de excepción para todas las demás excepciones que no tienen números de prioridad predeterminados. En otras palabras, el programador puede definir la prioridad para todas las excepciones marcadas como programables.

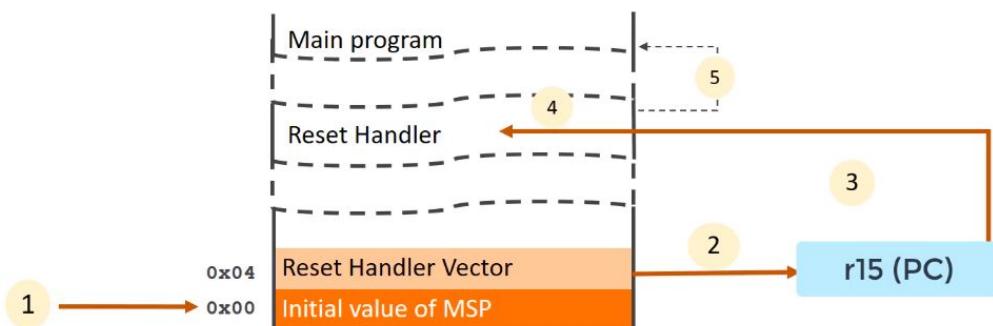
Las excepciones que terminan teniendo el mismo nivel de prioridad siguen un orden de prioridad fijo usando el número de excepción; cuanto menor sea el número de excepción, mayor será la prioridad. Por ejemplo, si el usuario ha definido **Bus Fault** y **Usage Fault** para que tengan el mismo nivel de prioridad, **Bus Fault** tendrá mayor prioridad y puede evitar la **Usage Fault**, ya que tiene un número de excepción más bajo.

## Comportamientos Armv8-M

Veamos que debería suceder fuera del **Reset** cuando la entrada de reinicio del procesador está impuesta. Como parte de la secuencia de reinicio en cualquier sistema de perfil M, el valor inicial de **MSP** se carga desde esta dirección (0x00) y la dirección **reset handler** es cargado en el **PC** desde esta dirección (0x04). El **Reset Handler** se ejecuta en **Modo Thread privilegiado** y probablemente bifurcará a un **Main program** escrito en C o C++.



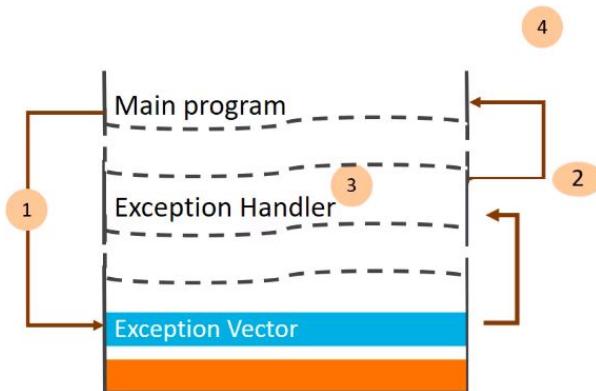
### Reset behavior



Veamos el Exception Behavior. Si se produce una excepción y tiene una prioridad suficiente para ser atendida, el flujo de instrucciones actual se detiene, el estado del procesador se almacena en el pila actual, y el procesador accede a la tabla de vectores para leer la dirección del vector para la excepción asociada. El **Exception Handler** se ejecuta en **Modo Handler**. Una vez que la excepción ha sido atendida, su controlador puede volver al Thread interrumpido, asumiendo que no hay más excepciones pendientes con suficiente prioridad que podrían ser atendidas primeras. Otra cosa que sucede en la entrada de excepción es que el **LR** se modifica hacia el valor **EXC\_RETURN** especial, que indica como el procesador debe volver de forma segura al Thread previamente interrumpido.



## Exception behavior

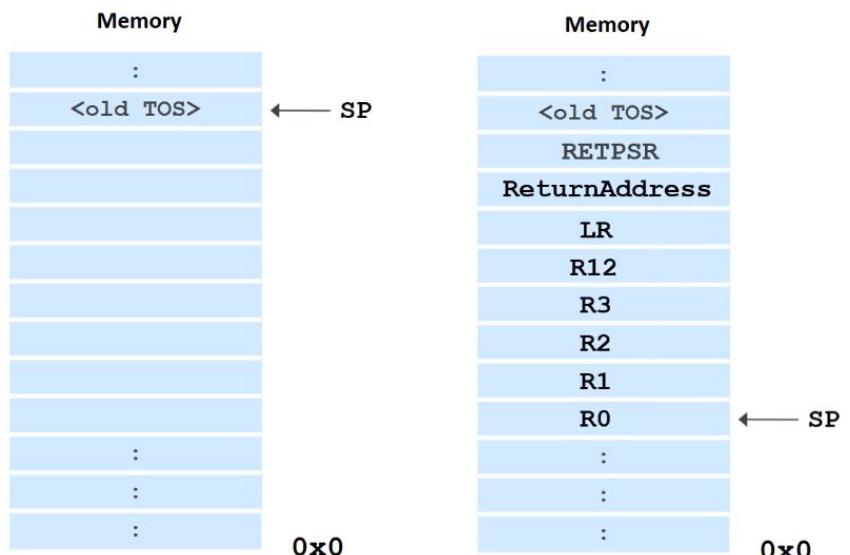


## Armv8-M stacking exception entry

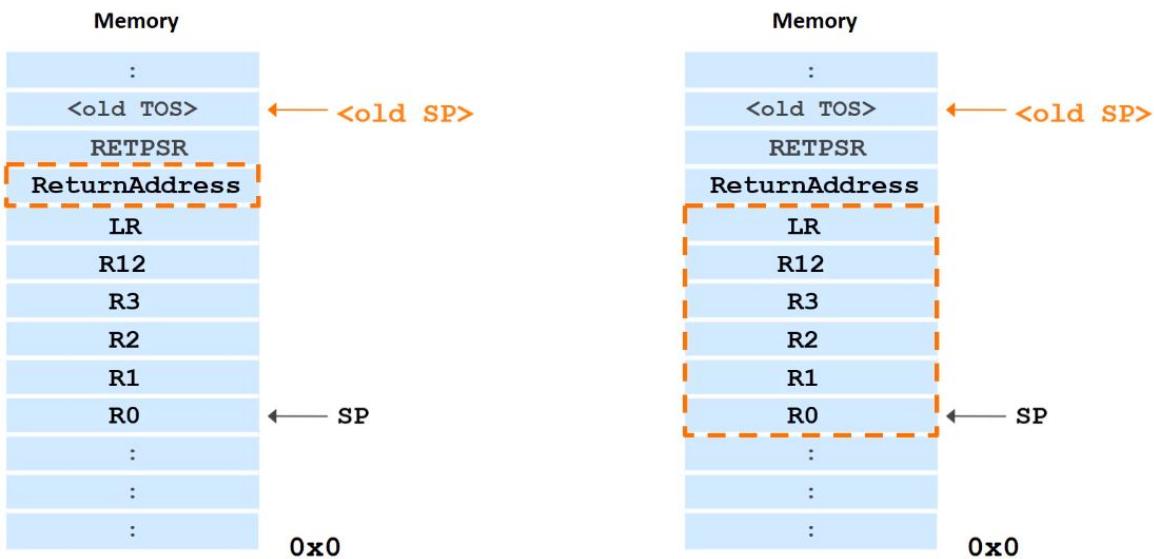
Veamos la información que el procesador introduce en la pila durante la entrada de excepción. La pila es **Full Descending** lo que significa que el hardware disminuye el **SP** al final del nuevo stack frame antes que almacene datos en la pila. Full significa que el **SP** apunta al último elemento conocido de la pila, que se muestra como "**old Top Of Stack**". Cuando ocurre una excepción, el hardware guarda 8 **Words** de 32 bits, que comprenden **RETPSR**, **Return Address**, **LR**, **R12**, **R3**, **R2**, **R1** y **R0**.



## Stacking on exception entry



El **RETPSR** contiene el estado anterior del procesador, por lo tanto, es vital que esta información se conserve. También es vital guardar la dirección de donde el procesador debería retornar después de reparar la excepción. Para las mayorías de las excepciones, como las interrupciones y **SVCALL**, **ReturnAddress** es la dirección de la instrucción que se habría ejecutado a continuación si no hubiera ocurrido la excepción. Sin embargo, para algunas excepciones como **UsageFault**, **ReturnAddress** es la dirección de la instrucción que causó la excepción, para permitir que la instrucción que causó la falla se vuelva a ejecutar, después de que la falla haya sido reparada por el **handler code**.



Los registros restantes **R0-R3, R12** y **LR** deben guardarse de acuerdo con el **AAPCS**, para que **handler code** de excepciones no los corrompa. Al guardar estos registros en la pila automáticamente, el **handler code** de excepciones no necesita preocuparse por corromper el contenido de estos registros, reduciendo la sobrecarga del software y permitiendo que manipuladores de excepciones sean escritos como funciones normales de **C**.

## Comportamiento de entrada de excepción Armv8-M

Este artículo describe el comportamiento de entrada de excepción con más detalle. Consulte la sección "Manejo de excepciones" en el [Manual de referencia de arquitectura Armv8-M](#) para obtener la información completa.

Cuando ocurre una excepción, se da preferencia a la ejecución actual y causa la siguiente secuencia básica:

1. **R0-R3, R12, LR, RETPSR** y **CONTROL.SFPA** están apilados.
2. La return address se determina y apila.
3. Apilamiento opcional del **Floating-point context**, que puede ser cualquiera de los siguientes:
  - Sin apilamiento o preservación del **Floating-point context**.
  - Apilar el básico **Floating-point context**.
  - Apilar el básico **Floating-point context** y el adicional **Floating-point context**.
  - Conservación del estado **Lazy Floating-point**.
4. **LR** está seteado en **EXC\_RETURN**.
5. Borrado opcional de registros, dependiendo de la transición del **Security state**.
6. También se borran las siguientes banderas:
  - El **IT State** se borra si se implementa la **Main Extension**.
  - **CONTROL.FPCA** se borra si se implementa la **Floating-point Extension**.
  - **CONTROL.SFPA** se borra si se implementan la **Floating-point Extension** y la **Security Extension**.
7. Se activa una transición al **Security state** de la excepción.
8. Se elige la excepción que se debe tomar y se establece **IPSR.Exception** en consecuencia. La configuración de **IPSR.Exception** en un valor distinto de cero (non-zero) hace que el **PE** cambie al **Handler mode**
9. **CONTROL.SPSEL** se establece en **0**, para indicar la selección de la main stack, dependiendo del **Security state** al que se dirige.
10. El bit **Pending** de la excepción a tomar se establece en **0**. El bit **Active** de la excepción a tomar se establece en **1**.
11. El **Security state** se cambia al **Security state** de la excepción que se está activando.

12. Los registros se borran, dependiendo de la transición del **Security state**. Los registros se dividen entre **the caller** y **callee**. Si la transición del **Security state** es de estado **Secure** a **Non-secure**, todos los registros se borran a **0**. En todos los demás casos, **the caller registers** se establecen en un valor **UNKNOWN** y los **the callee registers** permanecen sin cambios y no se apilan.
13. **EPSR.T** se establece en el bit [0] del vector de excepción para que se tome la excepción.
14. El **PC** está configurado en el vector de excepción para que se tome la excepción.

## Comportamiento de retorno de excepción Armv8-M

Una vez que los 8 registros se han apilado, el **LR** se modifica a un valor especial conocido como **EXC\_RETURN**. **EXC\_RETURN** no es un registro real. **EXC\_RETURN** es un valor especial definido arquitectónicamente por el mecanismo de retorno de excepción.

Se produce un retorno de excepción cuando una de las siguientes instrucciones carga un valor **EXC\_RETURN** en el **PC**. A continuación podemos verlo en la siguiente imagen resaltada en verde.

Cargar el **PC** con una dirección como esta, provocaría un error ya que todos los valores **EXC\_RETURN** son direcciones ilegales para la ejecución de instrucciones. En cambio, cuando el procesador ve que los bits **24-31** de la dirección están configurados, se da cuenta de que necesita realizar un retorno de excepción y, en su lugar, lee los bits **0 a 6** de **EXC\_RETURN**, para determinar qué hacer exactamente.



### Exception return behavior

- POP/LDM that includes loading the program counter (PC).
- LDR with PC as a destination.
- BX with any register (for example, BX LR, or BXNS).



Las asignaciones de bits para la carga útil **EXC\_RETURN** en ARMv8-M son diferentes de los valores **EXC\_RETURN** en ARMv7-M y ARMv6-M.

En un valor válido **EXC\_RETURN**, el campo **PREFIX** de los bits **24 a 31** debería leerse como todos los **1**, o **0xFF** en hexadecimal.



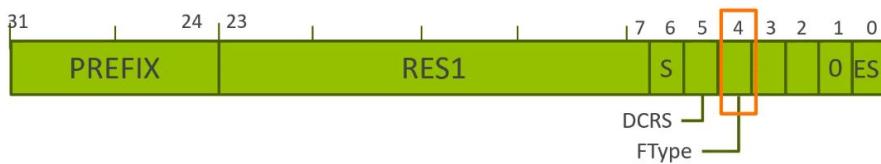
El **bit 6** indica si una pila **Secure** o **Non-secure** es usado para restaurar el stack frame en el retorno de excepción (bit 5).



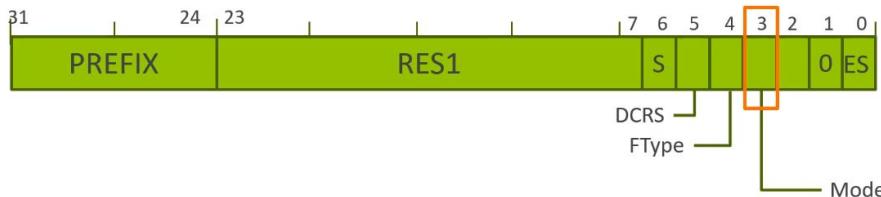
El **default calle register stacking bit (DCRS)** indica si las reglas predeterminadas para apilar los **calle registers** han sido seguidos u omitidos.



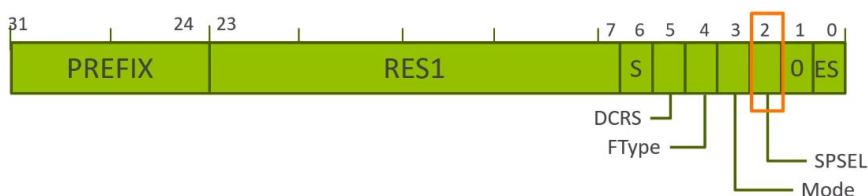
El bit 4 indica si la **stack frame** usada es una **standard integer only frame** o un **extended floating-point stack frame**, si la **Floating-point Extension** es implementada.



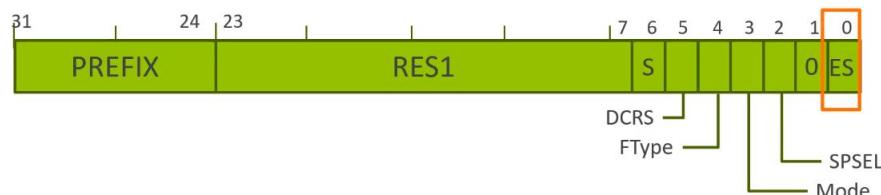
El bit 3 indica el modo desde el que se apiló, ya sea **Thread** o **Handler**.



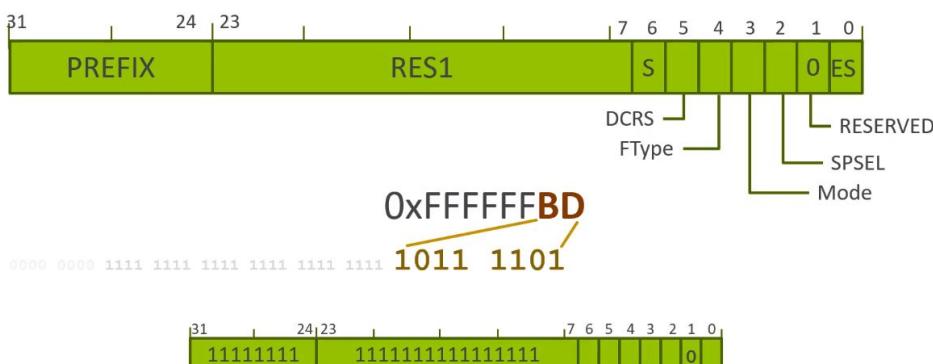
El bit 2 indica si **PSP** o **MSP** se seleccionó en **CONTROL.SPSEL** durante el apilamiento.



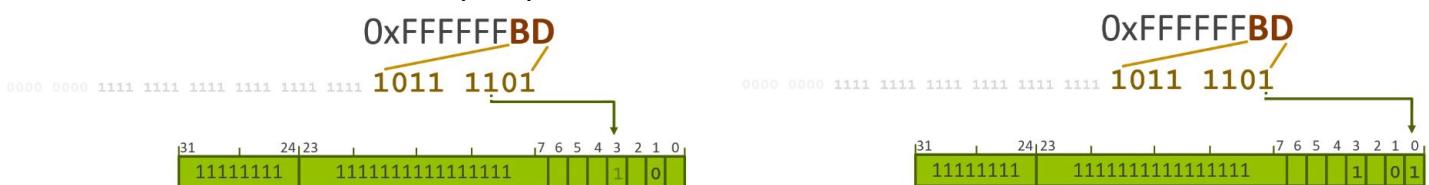
El bit 0 es el bit de **Exception Security** que indica si la excepción se toma a un estado **Secure** o **Non-secure**



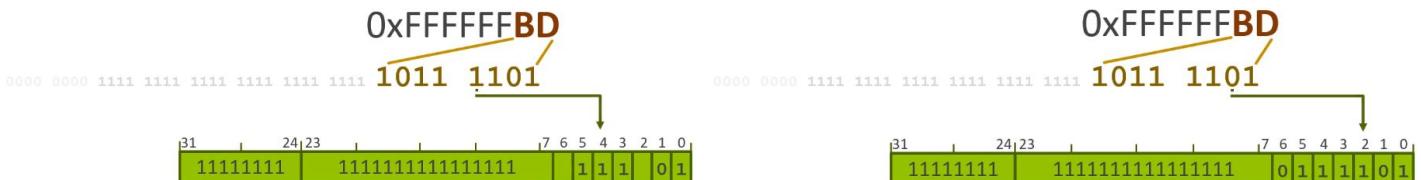
El bit 1 es un campo reservado.



Si tomamos el valor **EXC\_RETURN** que vimos en el ejemplo anterior, podemos ver que indica que la excepción ocurrió cuando el procesador estaba en **Thread Mode** que se muestra en el **bit 3** y se lleva al **Secure state** en el **Handler Mode** (**bit 0**).



Se utilizó un esquema de apilamiento de frame estándar predeterminado (bit 5 y bit 4) para los almacenar registros en la **Non-secure** (bit 6) **Process Stack** (bit 2).



# Ejemplo de entrada de excepción Armv8-M NMI

Que ocurre en la entrada de excepción (exception entry) utilizando una excepción **NMI** como ejemplo.

Para empezar el procesador está en **Modo Thread** y está ejecutando una instrucción **ADD** cuando un **NMI** se impone. La instrucción **ADD** de ciclo único se completa antes de que el procesador tome las siguientes acciones principales.

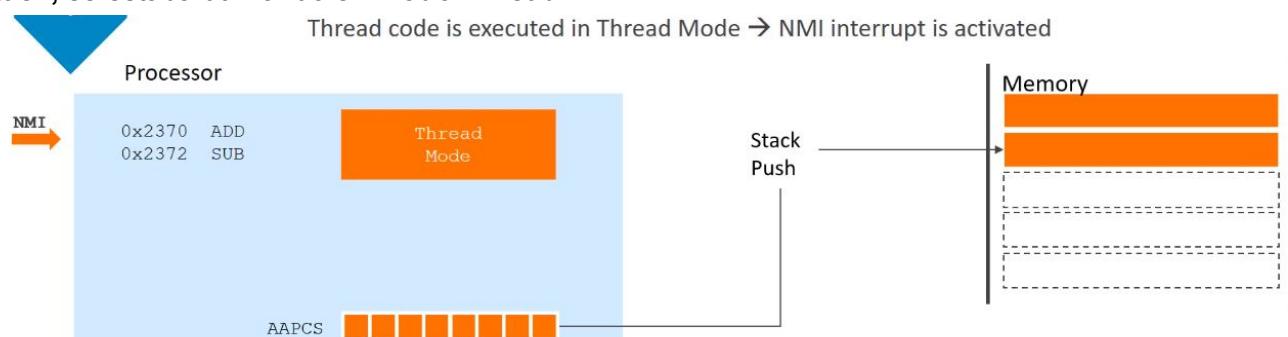


## NMI exception entry example

Thread code is executed in Thread Mode → NMI interrupt is activated

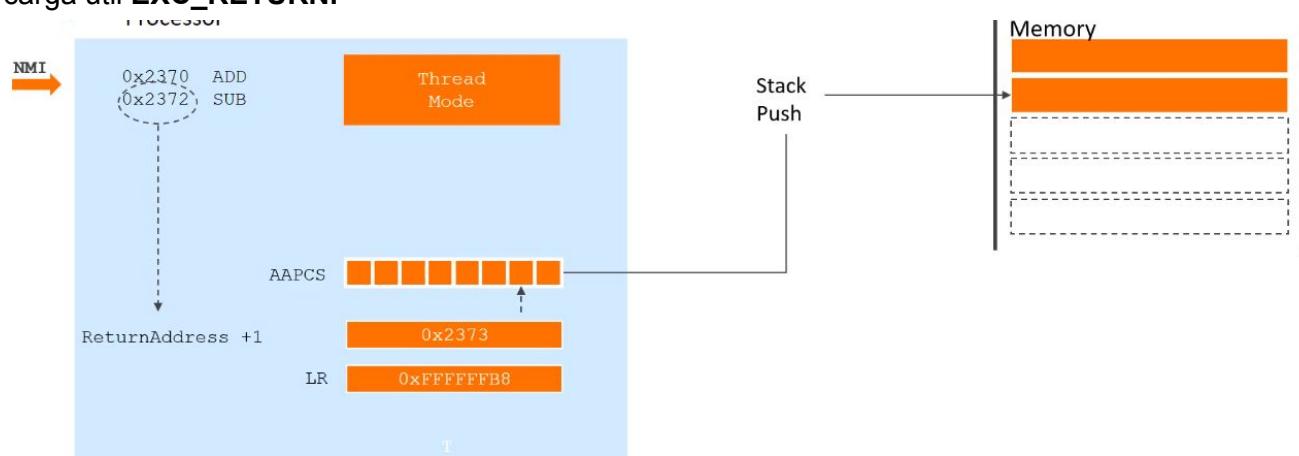


8 registros se insertan en la pila actual. El registro de **CONTROL** indica que pila, **Main Stack** o **Process Stack**, se estaba utilizando en **Modo Thread**.

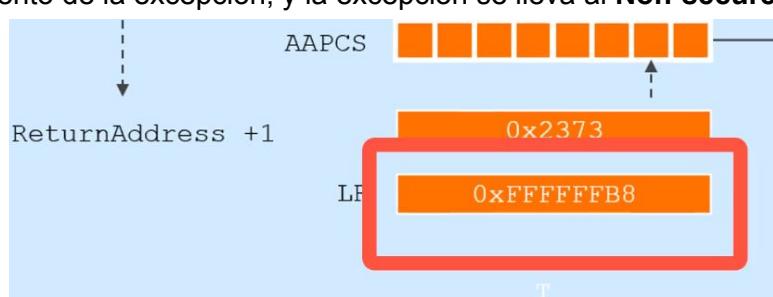


**ReturnAddress** se establece en la dirección de las subsiguiente instrucción **SUB + 1**.

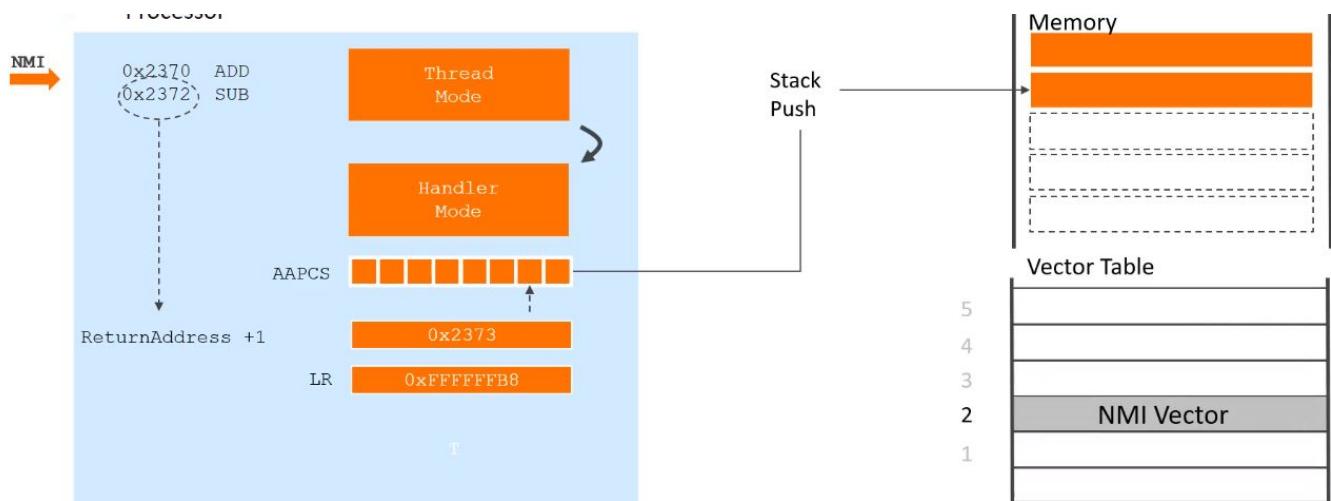
El **+1** no indica que la dirección de retorno sea una dirección impar, no alineada; más bien, el bit menos significativo impar es copiado en el bit **T** **EPSR** para que el procesador retorne al estado de ejecución **Thumb**. Esto es diferente a las arquitecturas de perfil M anteriores en que el estado de la **CPU** ya no es contenido en la carga útil **EXC\_RETURN**.



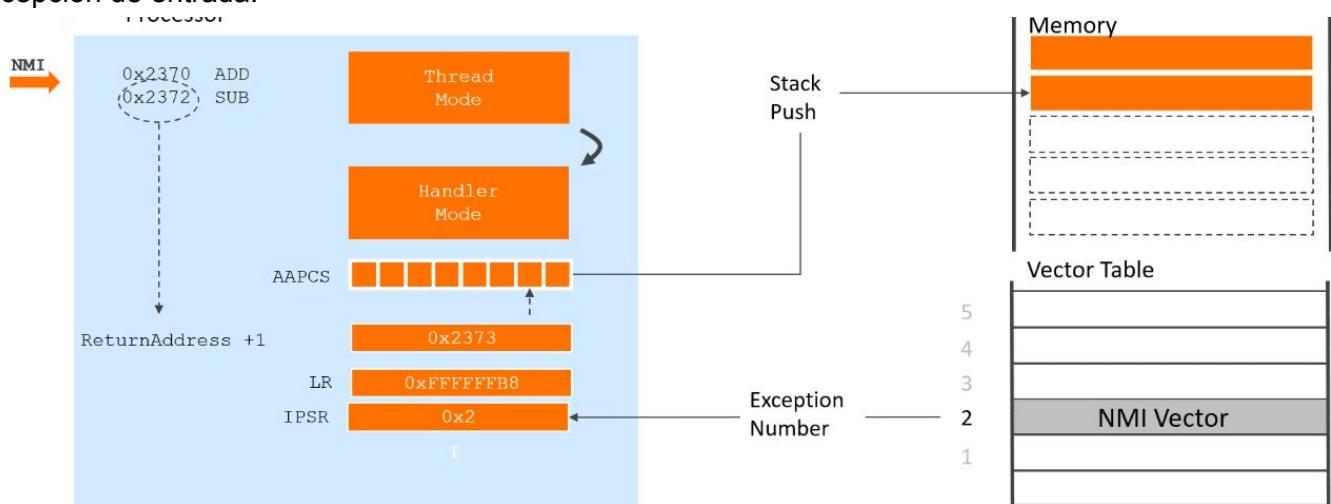
El **LR** se modifica a esta dirección [0xFFFFFFF8] para la excepción de retorno, que indica que la **Main Stack** estaba activa en el momento de la excepción, y la excepción se lleva al **Non-secure state**.



Durante (o después) el estado de guardado, la dirección del **NMI handler** se lee de la tabla de vectores. El procesador cambia a **Modo Handler**.



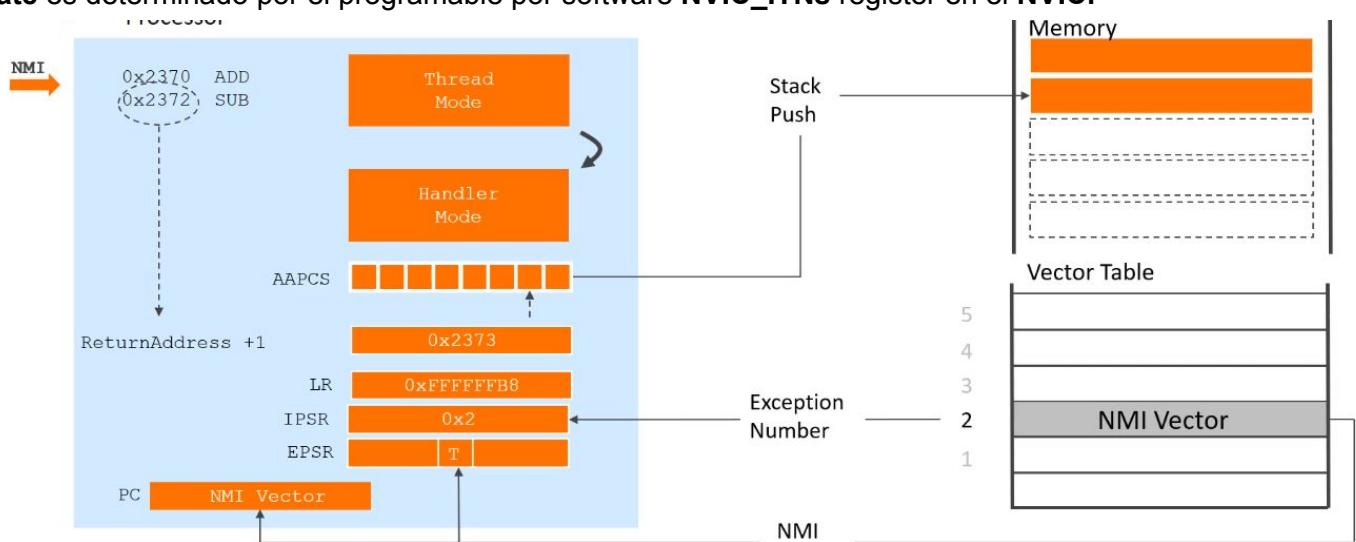
El **IPSR** se actualiza con la número de excepción correspondiente al **NMI**. El **EPSR T-bit** se establece de acuerdo con el bit menos significativo del vector **NMI**. Si no se establece el bit **T** en 1, se producirá un error de excepción de entrada.



Se ejecuta la primera instrucción de la rutina del controlador NMI (**NMI handler**).

La **NMI** handler se ejecuta en **Modo Handler** utilizando la **Main Stack**.

Tenga en cuenta de que si la excepción **NMI** y otras excepciones se manejan en **Secure state or Non-secure state** es determinado por el programable por software **NVIC\_ITNs** register en el **NVIC**.



# Ejemplo de retorno de excepción de Armv8-M NMI

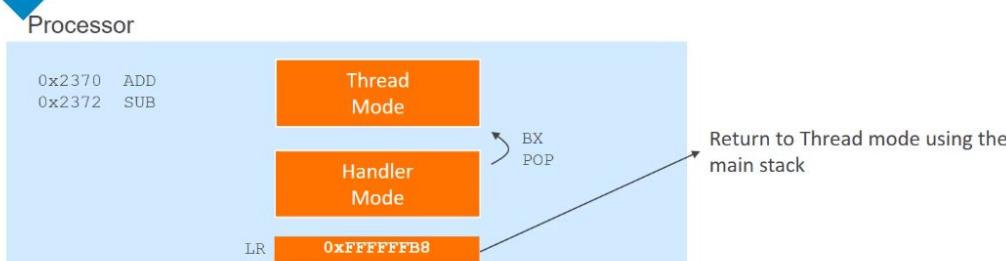
Es simple para el software volver a la tarea en primer plano desde el **NMI handler**.

El **LR**, **EXC\_RETURN** que contiene este valor [0xFFFFFB8] indica que un retorno de excepción, retornara el procesador a **Modo Thread** utilizando **Main Process Stack**.

Si el valor **EXC\_RETURN** se mantiene en el **LR** al final de **NMI handler** simplemente puede ejecutar una operación **BX LR** para volver al contexto anterior.



## NMI exception return example

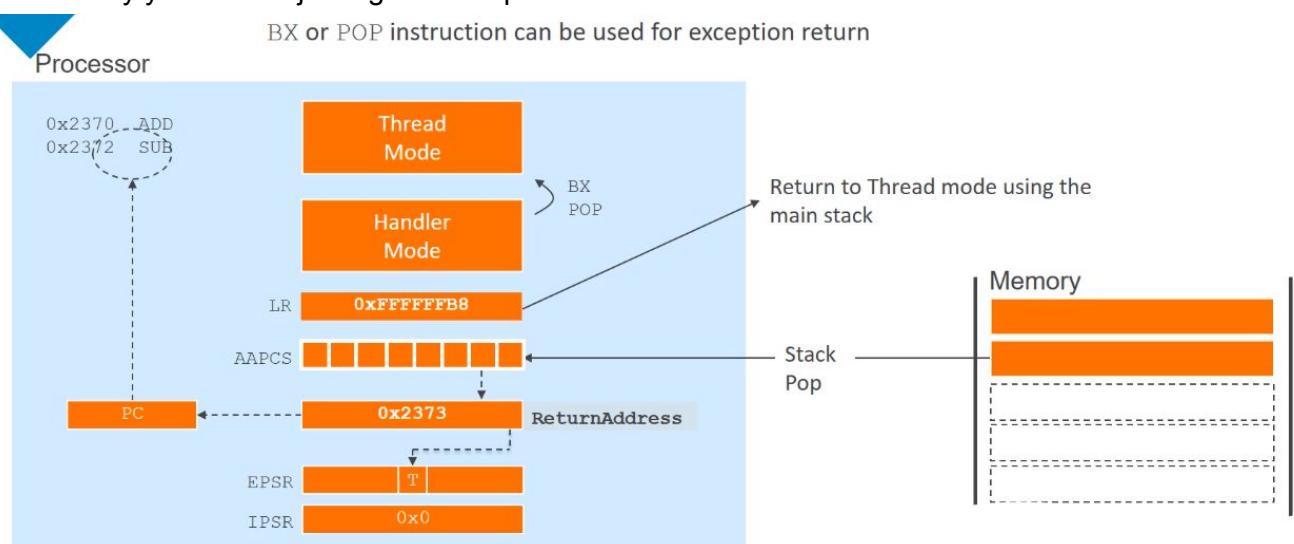


Si el **LR** se guardó en la pila al comienzo de la rutina de servicio de interrupción el software simplemente puede usar una operación **POP** para cargar el valor **EXC\_RETURN** de la pila en el **PC**.

Al ejecutar una operación de retorno de excepción válida, el procesador hará **POP** el contexto anterior de la Main Stack en los 8 registros antes mencionado.

Configurar **PC** en **ReturnAddress** para recuperar la operación **SUB** y siguiendo las instrucciones en el **Processor's Pipeline**.

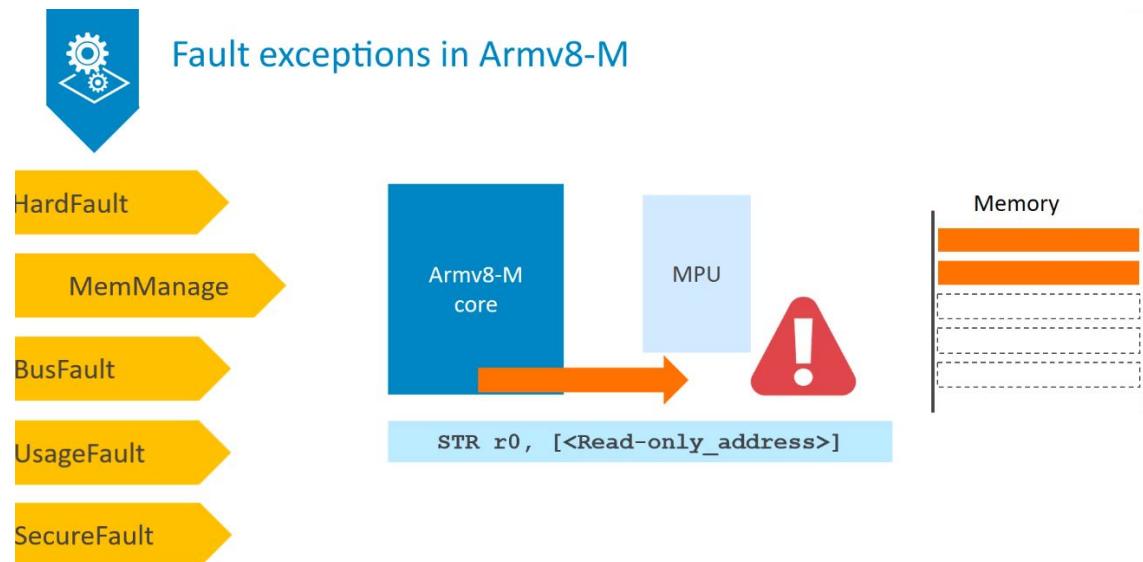
Y finalmente establecer el **IPSR** en este valor (0x0) que indica que el procesador está de nuevo en **Modo Thread** y ya no maneja ninguna excepción.



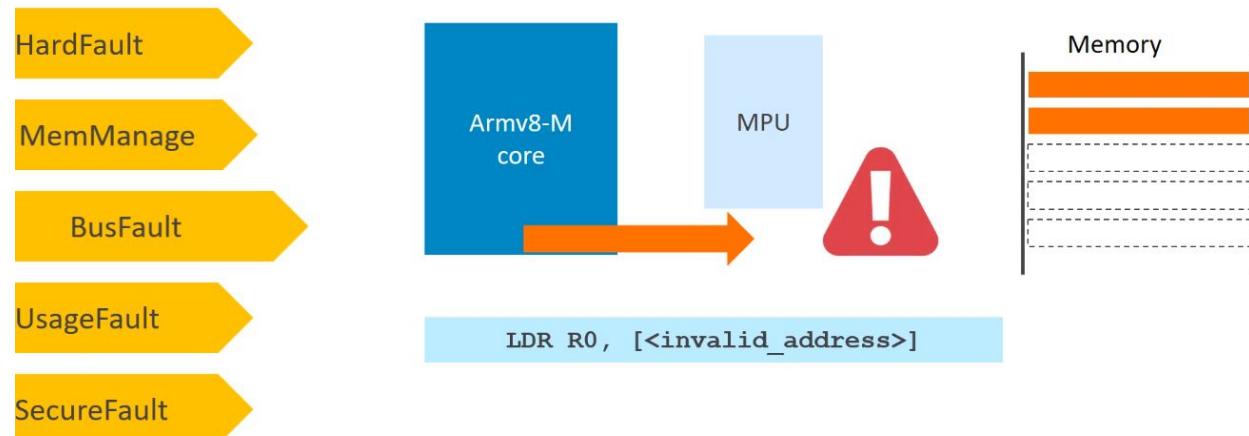
# Arm v8-M fault exceptions

Hay 5 clases de fallas en ARMv8-M: **HardFault**, **MemManage**, **BusFault**, **UsageFault** y **SecureFault**.

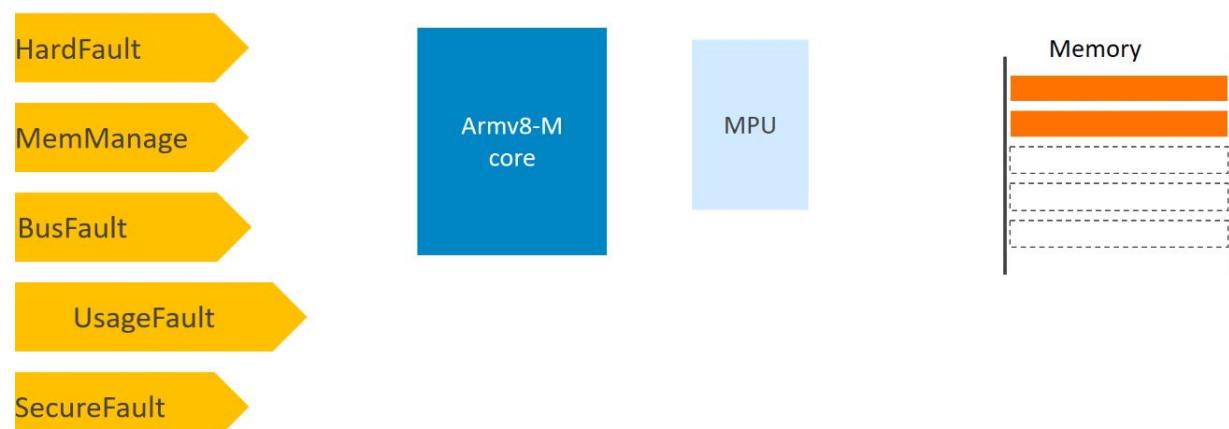
**MemManage Fault:** Las fallas **MemMange** ocurren debido a violaciones de permisos de la **MPU**, ya sea en una búsqueda de instrucciones, un acceso a datos, un proceso de apilamiento y desapilamiento de hardware o preservación del contexto **Floating point**.



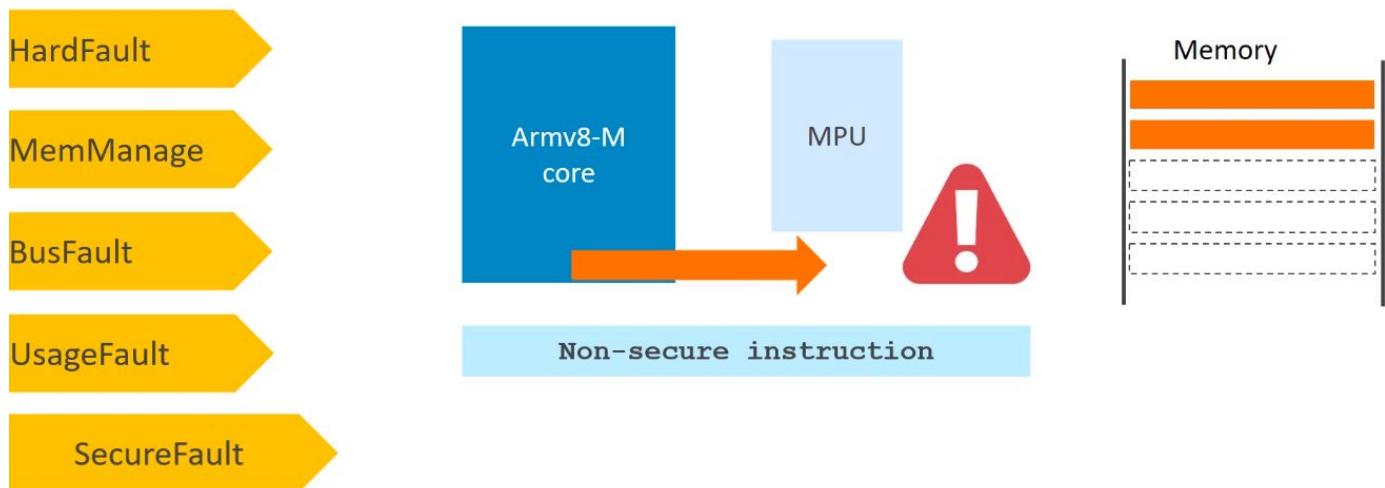
**BusFault:** puede ocurrir debido a un error de acceso a la memoria, después de que el acceso ha pasado la verificación de **MPU** y la transacción se ha ido al **Bus**



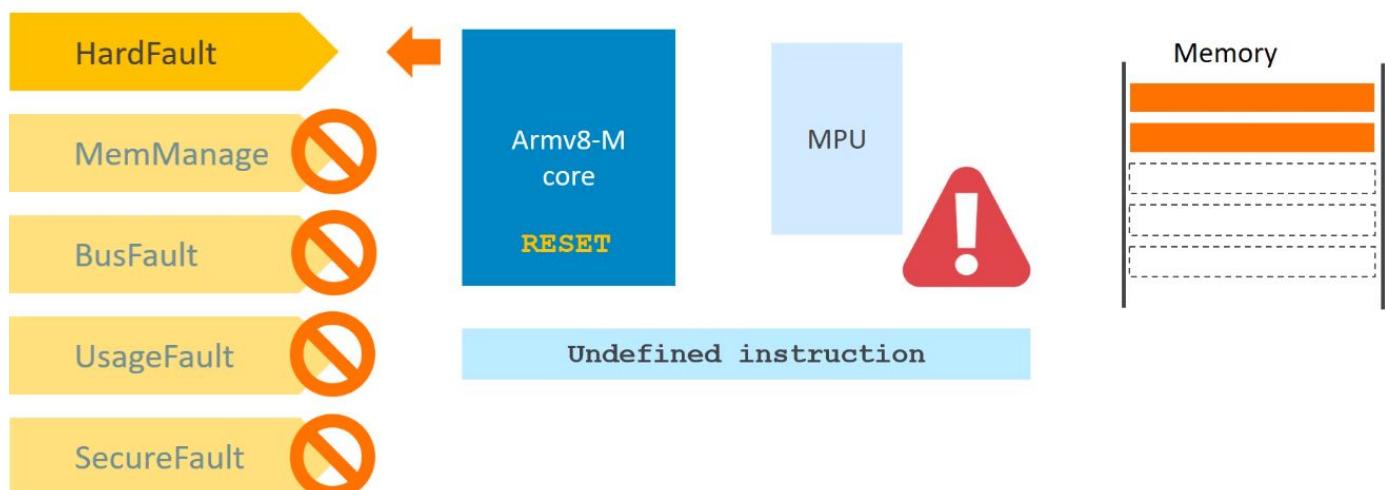
**UsageFault:** puede ocurrir debido a una variedad de errores relacionados con la ejecución, como ejecutar una instrucción indefinida, poner el procesador en un estado ilegal (bit T no establecido), desbordando la pila, accesos unaligned no permitido, etc.



**SecureFault:** es nuevo en la arquitectura ARMv8-M y único para implementaciones con **Security Extension**. **SecureFault** puede ocurrir debido a una variedad de errores relacionados con la seguridad.



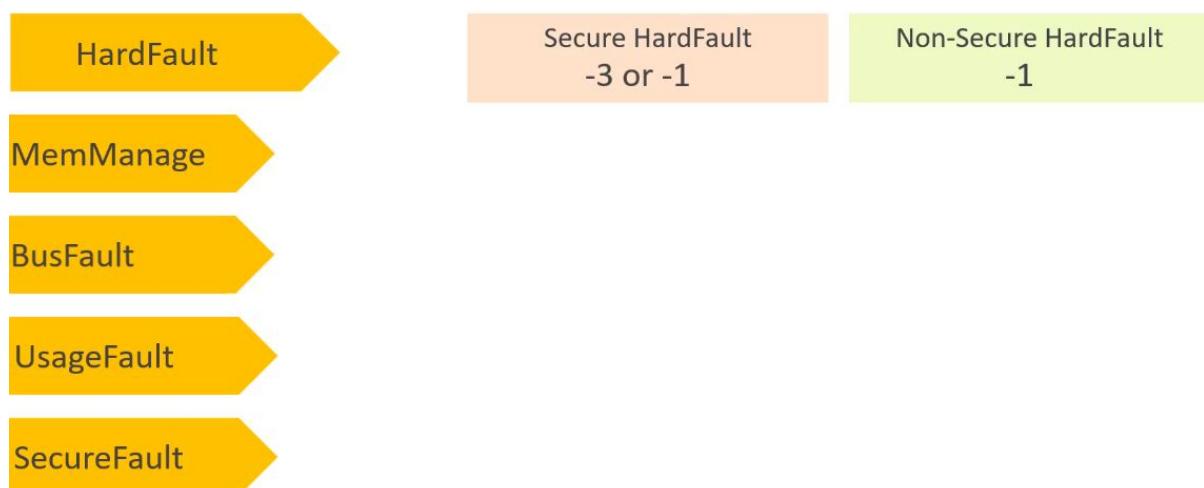
En **Reset**, la falla **MemManage**, **BusFault**, **UsageFault** y **SecureFault** están todas deshabilitadas. Si algunas de estas fallas ocurren cuando están deshabilitados, el procesador las escala a **HardFault**, que siempre está habilitado con un nivel de prioridad -1.



Esto nos lleva finalmente a **HardFault**.

**HardFault** se descompone en **Secure HardFault** y **Non-secure HardFault**.

**Non-secure HardFault** siempre tiene un número de prioridad de -1, pero **Secure HardFault** tiene una prioridad configurable de -3 o -1. **Secure HardFault** tambien solo está disponible si se implementa **Security Extension**.



## Resumen de Armv8-M

Esto concluye la Introducción al módulo de arquitectura **Armv8-M**. En resumen, vimos una breve introducción a qué es Arm Architecture y de qué trata el Perfil Cortex-M. Luego se presentó el conjunto de instrucciones T32 y los diferentes tipos de instrucciones disponibles en la arquitectura **Armv8-M**. Luego, pasamos a vistas detalladas del modelo del programador, el modelo de memoria y el modelo de excepción.

Haga clic aquí para obtener más información:

<https://developer.arm.com/support/training/arm-trustzone-for-armv8-m-online-introduction>

### Certificado de Examen



Matias Vironi

Has passed

Introduction to Armv8-M

Scoring:

100% (150 points)

on 7/10/2020

**arm**  
Training