Armv8-M Mainline Security Extension

Course 4 Advanced Armv8-M Features

Agenda

Overview

Memory Configuration

Function Calls & Toolchain Support

Exceptions

Learning Objectives

By the end of this session you should be able to:

- Describe the need for secure and Non-secure partitioning in Arm embedded system
- Describe the key different in the programmer's model with and without the Security Extension
- Describe the Armv8-M instructions used to call between Secure and Non-secure code
- Describe the architectural and hardware features that support Secure/Non-secure memory partitioning and signalling
- Partition Secure and Non-secure memory regions using CMSIS-Core
- Build secure applications and libraries using CMSE-compliant compiler tools
- Describe how the exception handling mechanism behaves with the Security Extension configured
- Target configurable interrupts as Secure or Non-secure and prioritize secure exceptions

Introduction to TrustZone for Armv8-M

Armv8-M architecture includes optional Security Extension

- Branded as Arm TrustZone for Armv8-M

Similar in concept to TrustZone for Armv8-A

- Implementation is optimized for microcontrollers

System may be partitioned between Secure and Non-secure software

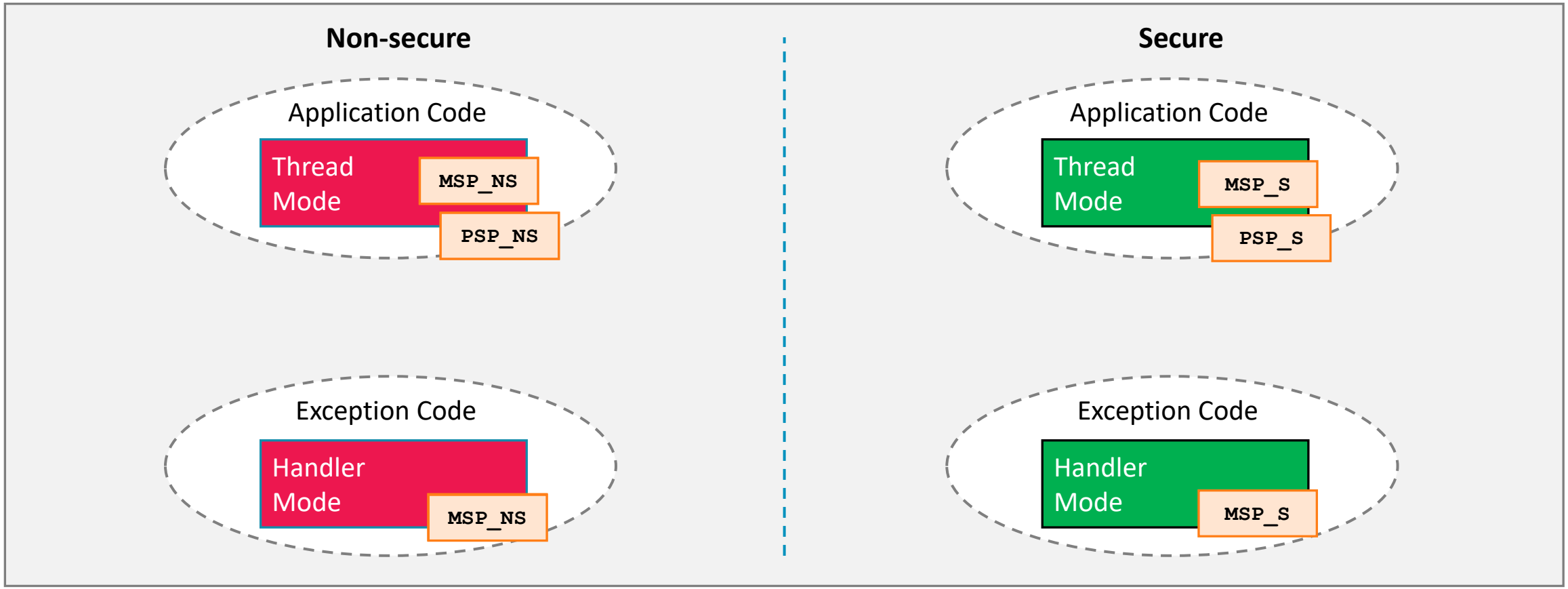
Secure software is highly trusted

- Has access to more system resources
- Protected from access by non-trusted code

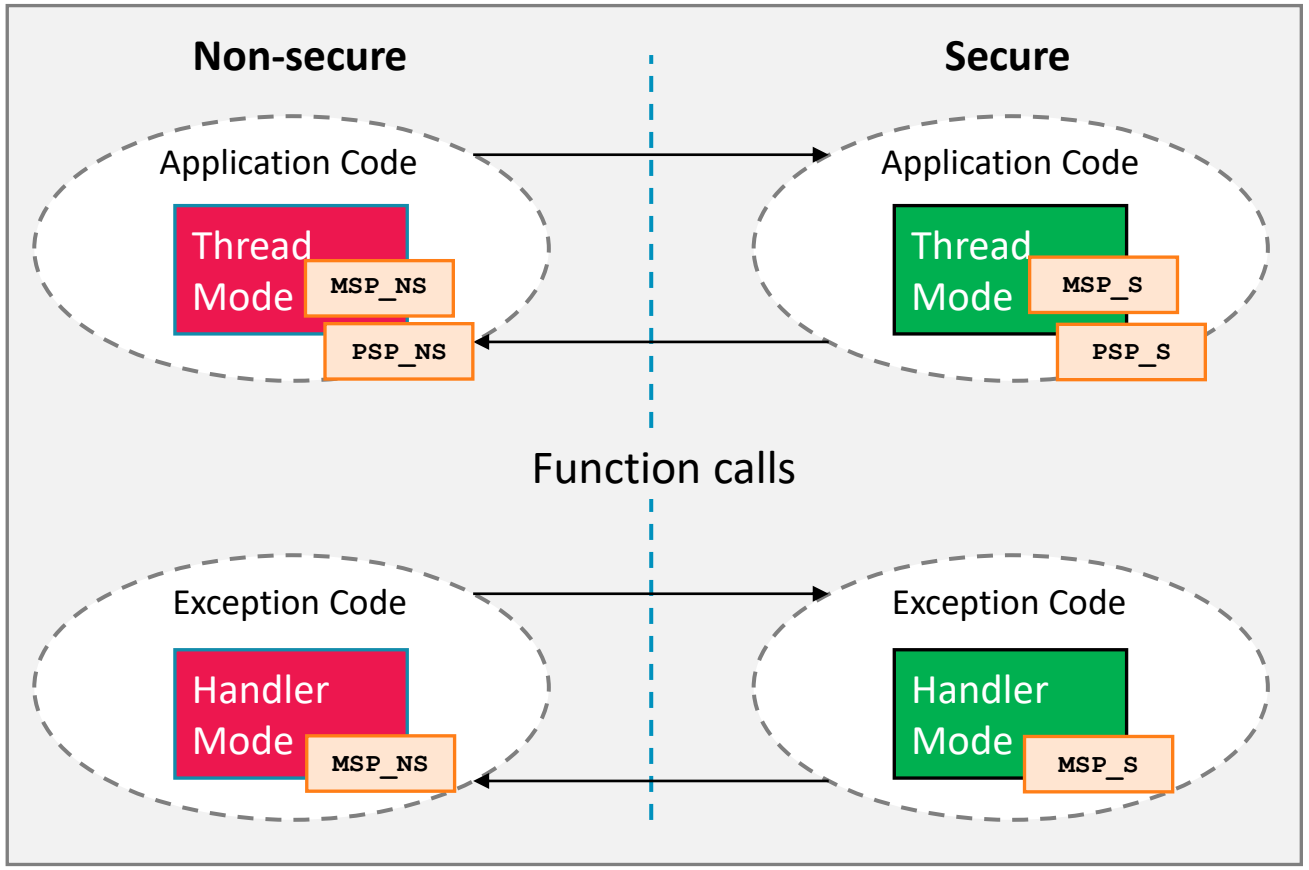
To protect the secure software the security extensions provide:

- Isolated Secure memory for code and data
- Secure execution state to run Secure code

Secure and Non-secure states



Calling between security states



Secure code can call Non-secure functions

- Non-secure functions and data should not be trusted

Non-secure code can call into Secure libraries

- Only a sub-set of the Secure code is callable
- Secure entry points are limited
- Non-secure code does not need to know it is calling a Secure function

This is different from Armv8-A TrustZone

- Where changing security state can only occur on an exception boundary

General-purpose register banking

Most general-purpose registers are common to both security states

- Registers **R0**-**R7**
  - Accessible to all instructions
- Registers **R8**-**R12**
  - Accessible to a few 16-bit instructions
  - Accessible to all 32-bit instructions
- **R14** is the link register (**LR**)
- **R15** is the program counter (**PC**)

R13 is the stack pointer (SP)

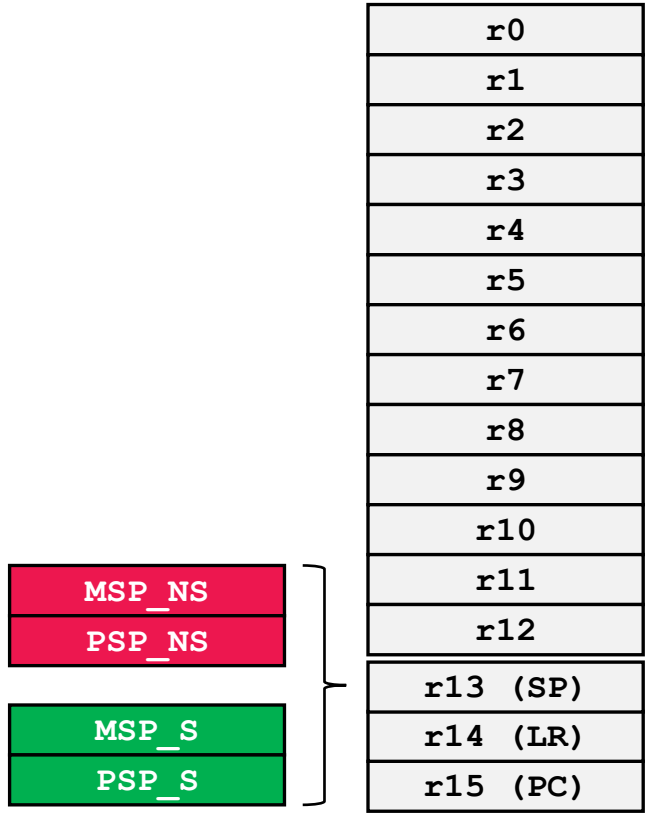
- Banked by security state

CONTROL.SPSEL selects between MSP and PSP

- Secure code can access **MSP_S** or **PSP_S**
- Non-secure code access **MSP_NS** or **PSP_NS**

Floating-point registers D0-D15 are not banked

CONTROL and some other special-purpose registers are also banked by security ...



Special-purpose register banking

Special-purpose registers are accessed using special instructions

- `MSR/MRS/CPS`

Some registers are security banked

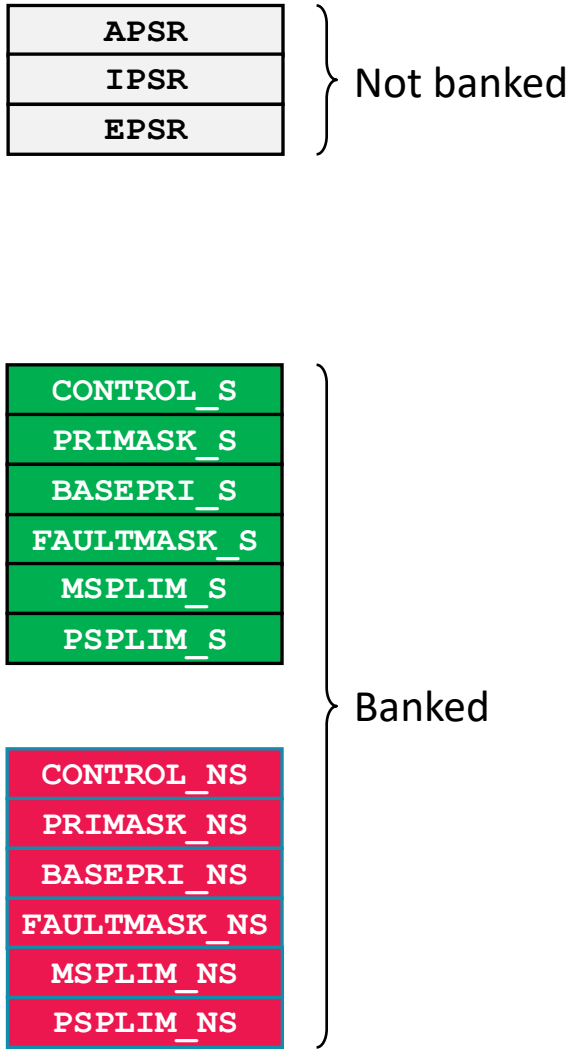
- `<register_name>_NS` - Non-secure instance of the register
- `<register_name>` - Current Security state

Non-secure code can only access Non-secure registers

Secure code can access Secure and Non-secure instances

```
MRS r0, CONTROL         ; Access current security
                        ; state's CONTROL register

MRS r0, CONTROL_NS      ; Access Non-secure CONTROL register
```



Agenda

Overview

Memory Configuration

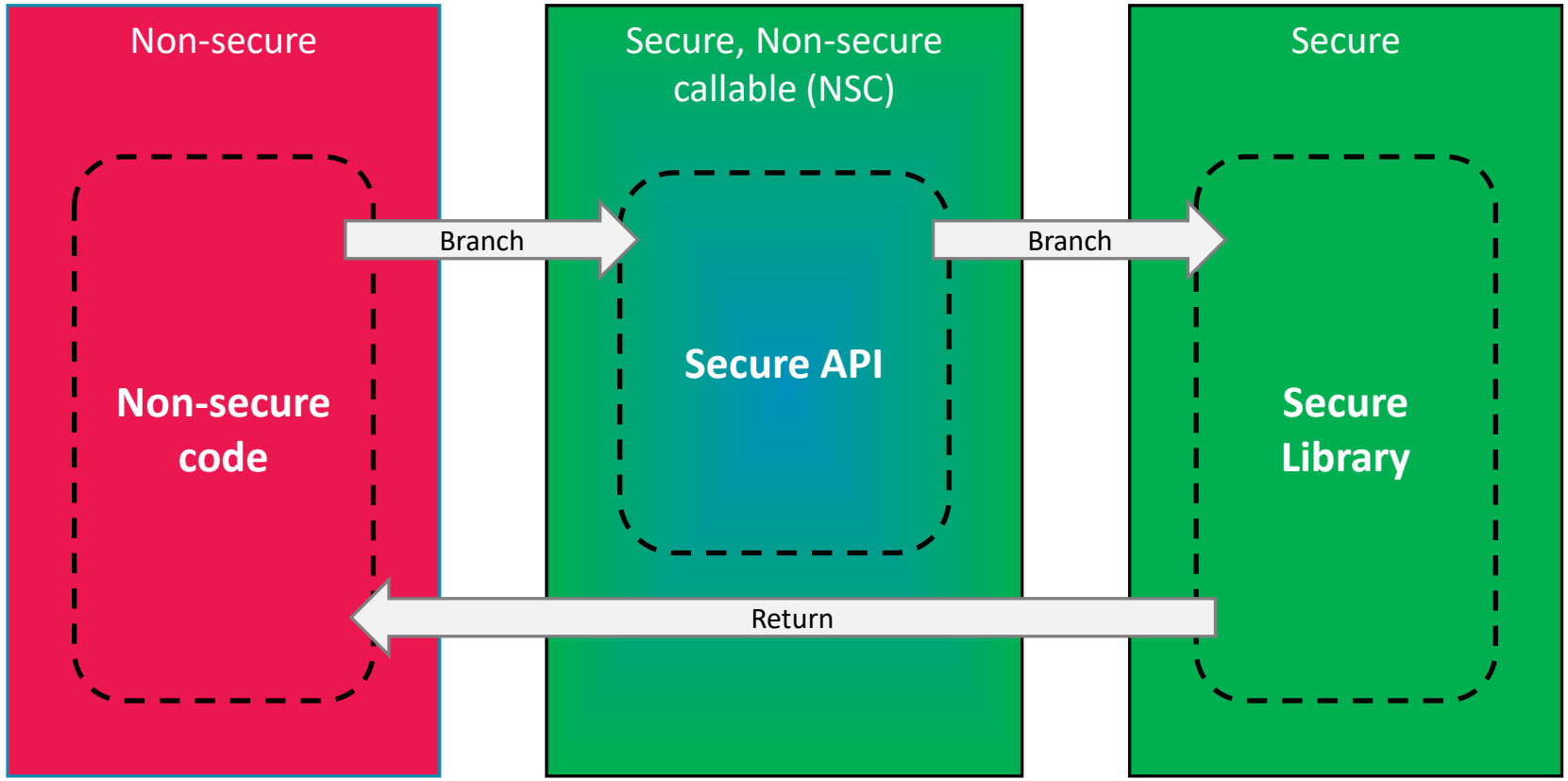
Function Calls & Toolchain Support

Exceptions

Memory security

Physical memory is split into Secure and Non-secure regions

- A Secure region can also be Non-Secure Callable (NSC)



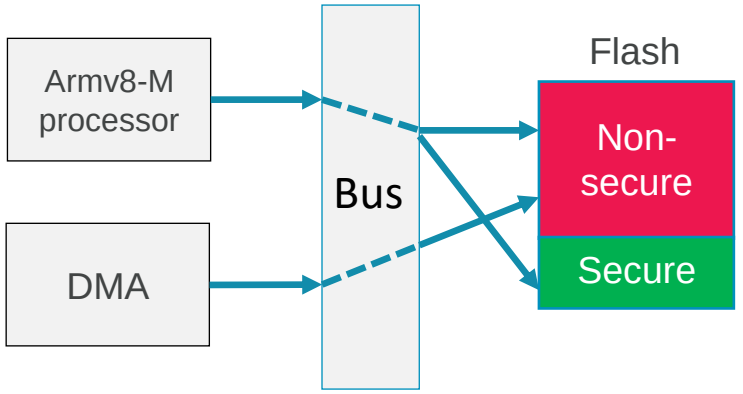
Secure memory rules

Rules enforced within the processor:

- Non-secure state cannot access Secure memory
- Secure code cannot be executed from Non-secure memory
- Secure code can access Non-secure data

Rule enforced by the memory system:

- AMBA 5 AHB (AHB5) adds HNONSEC signal
- AXI has security as part of AxPROT

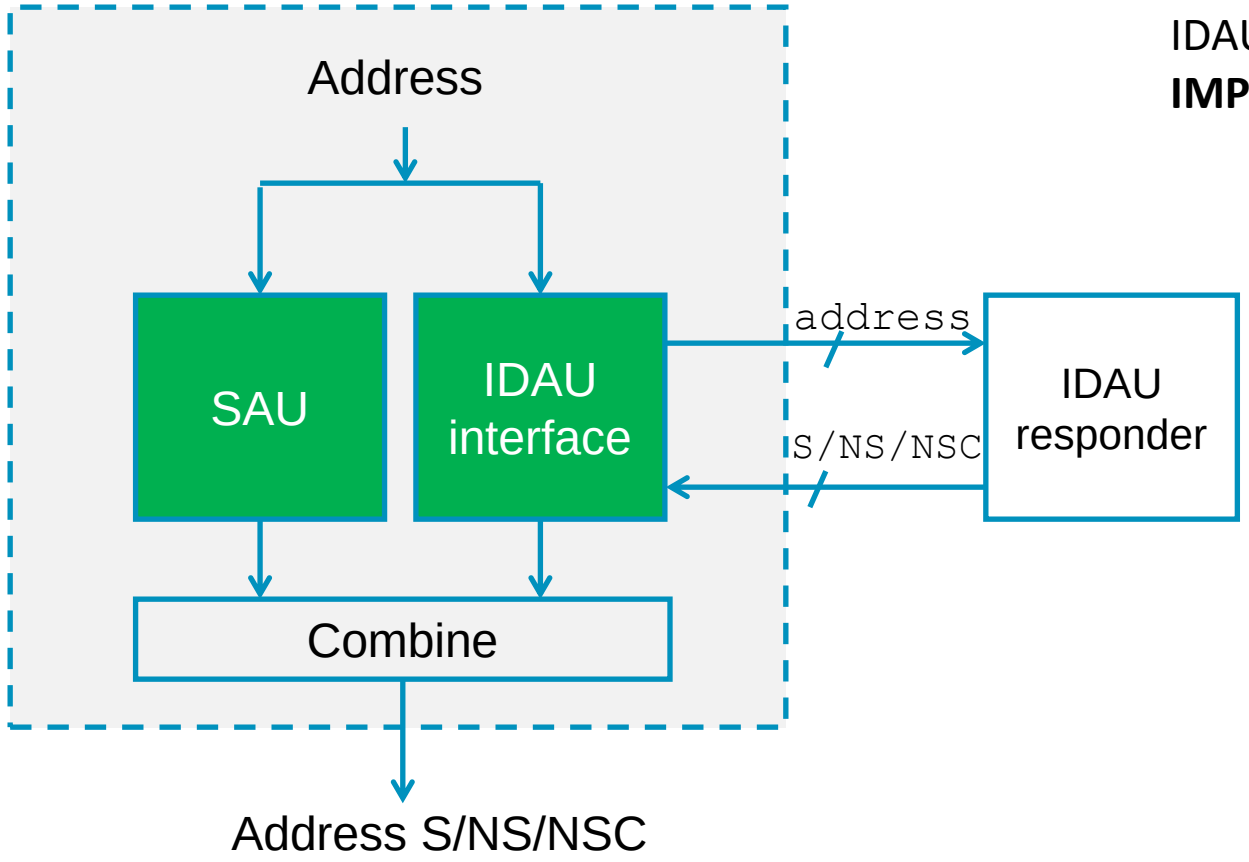


Bus level security signals can be used to protect memory from other masters

Memory security determination

The security state of a memory region is controlled by the combination of two values

- Security Attribution Unit (SAU)
- Implementation Defined Attribution Unit (IDAU)

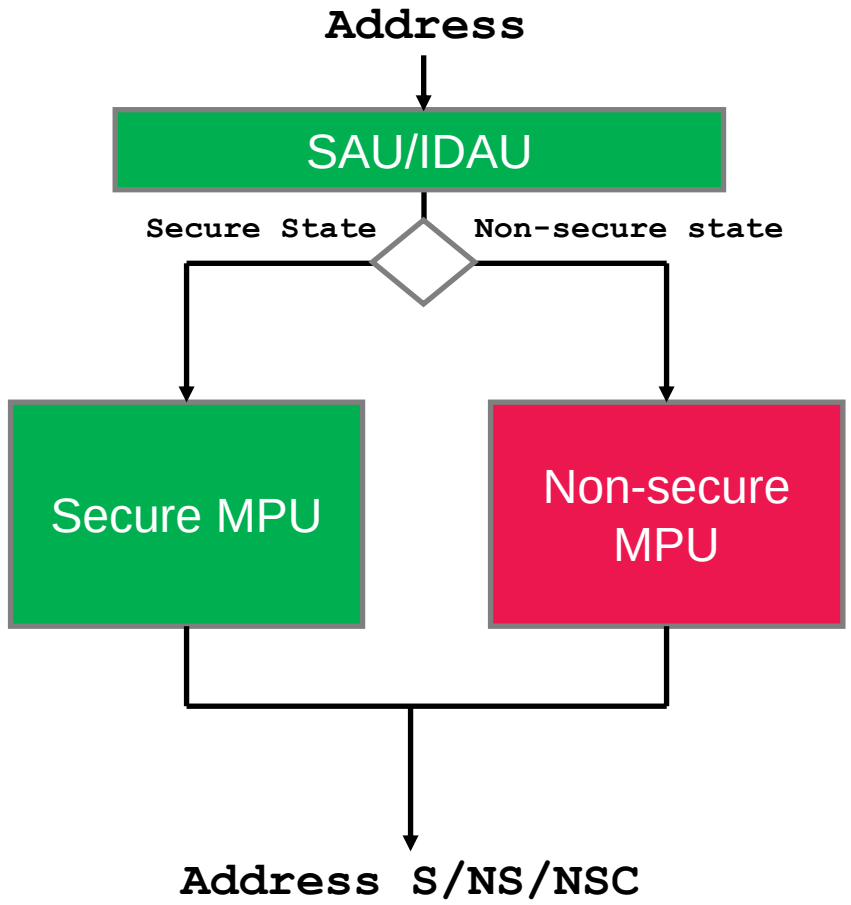


IDAU is optional and its details are **IMPLEMENTATION DEFINED**

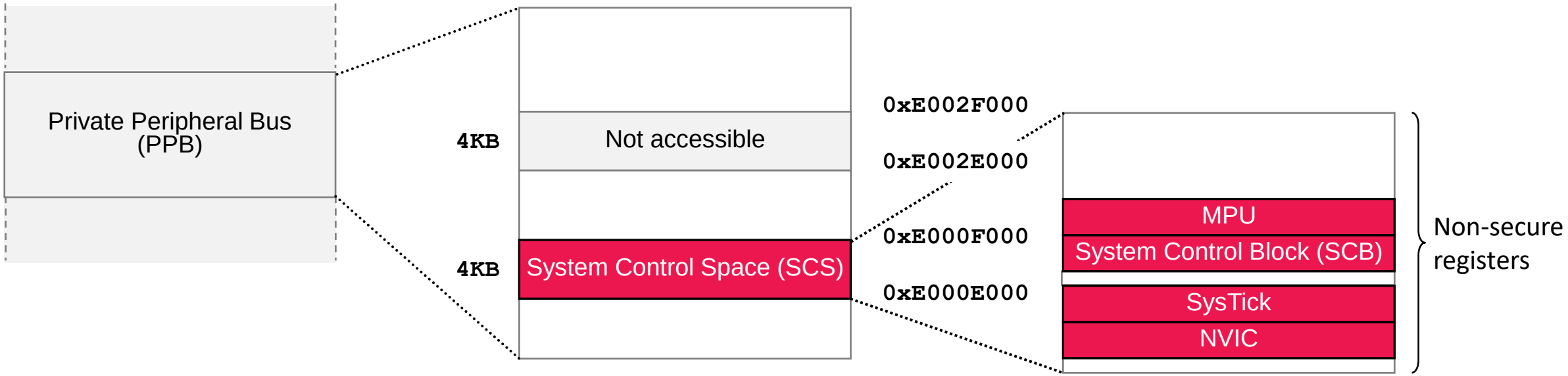
Memory Protection Unit

Independent MPU settings for Secure and Non-secure code

- **`MPU_*`** registers are banked across security states



Non-secure view of SCS



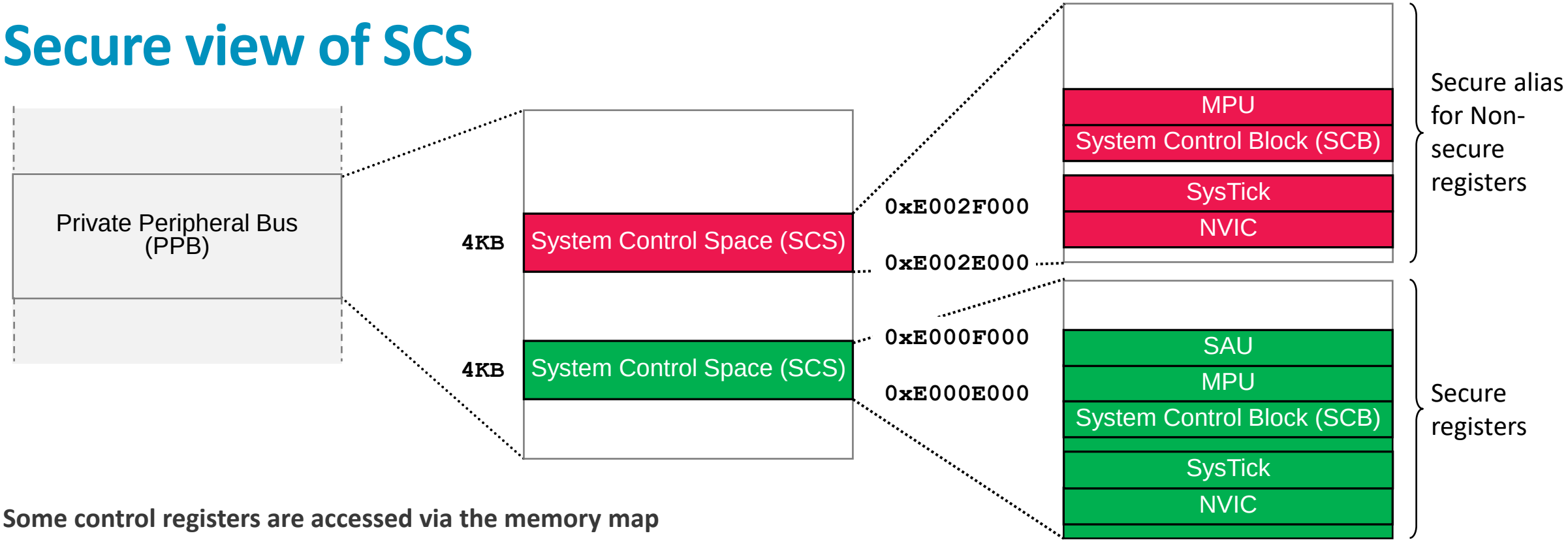
Non-secure code accesses SCS at `0xE000E000` – `0xE000F000`

- Views Non-secure banked register
- No Non-secure access to SAU control registers

Addresses `0xE002E000` – `0xE002F000` behave as if the address doesn't exist

- **`RAZ/WI`** if the access is privileged
- BusFaults if the access is unprivileged

Secure view of SCS



Some control registers are accessed via the memory map

- Reserved addresses in the Private Peripheral Bus

Secure accesses to 0xE000E000 – 0xE000F000 access Secure banked registers

- Security Attribution Unit (SAU) control registers are only available to Secure code

Secure accesses to 0xE002E000 – 0xE002F000 access Non-secure banked registers

- Registers behave as if they were accessed by Non-secure code

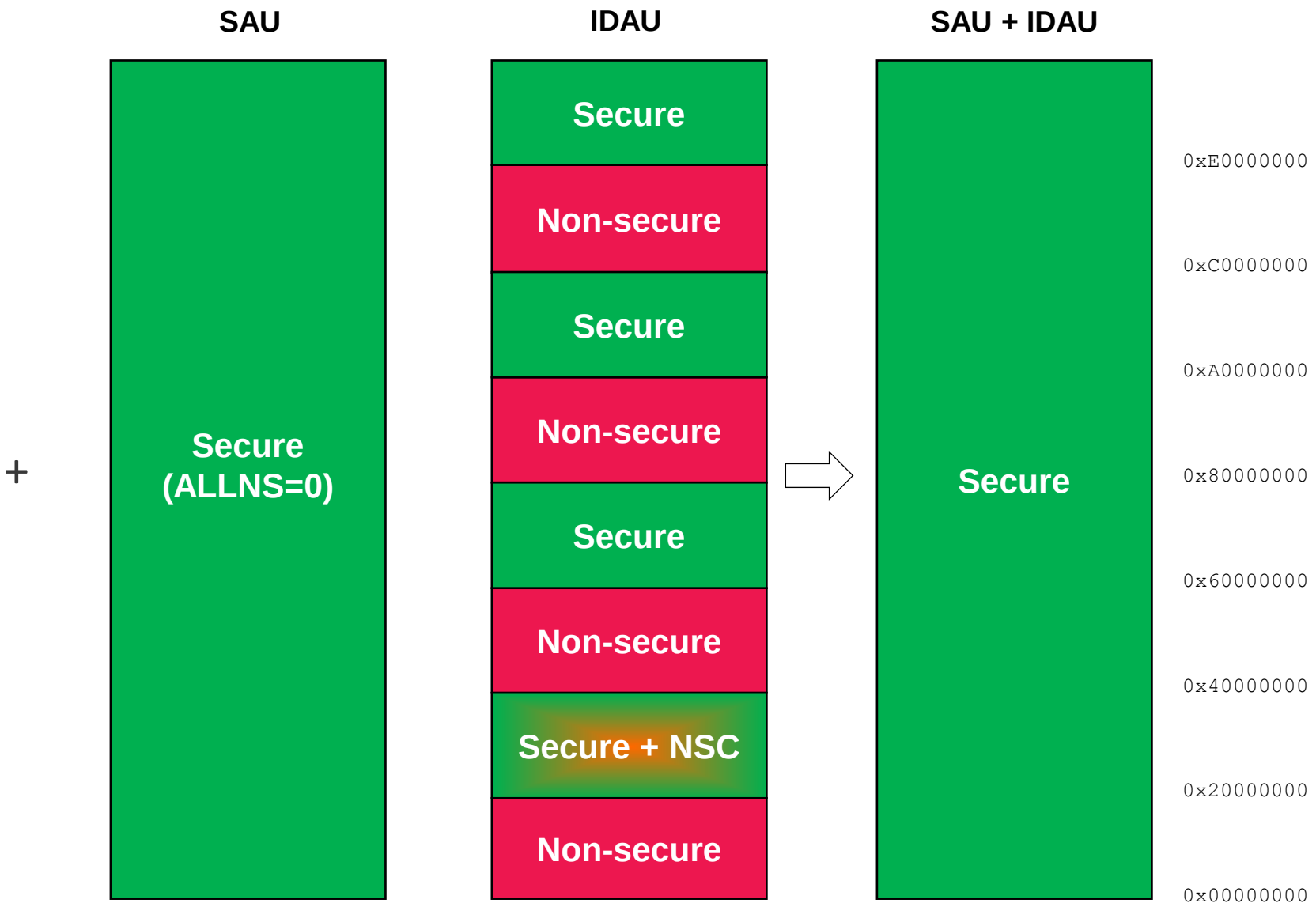
SAU registers

SAU registers are only in Secure SCS

- Configured in a similar way to the MPU
- The number of regions is implementation defined

Address	Name	Description
0xE000EDD0	SAU_CTRL	SAU Control Register
0xE000EDD4	SAU_TYPE	SAU Type Register
0xE000EDD8	SAU_RNR	SAU Region Number Register
0xE000EDDC	SAU_RBAR	SAU Region Base Address Register
0xE000EDE0	SAU_RLAR	SAU Region Limit Address Register

Boot security map – example configuration (1)



IDAU provides default security mappings

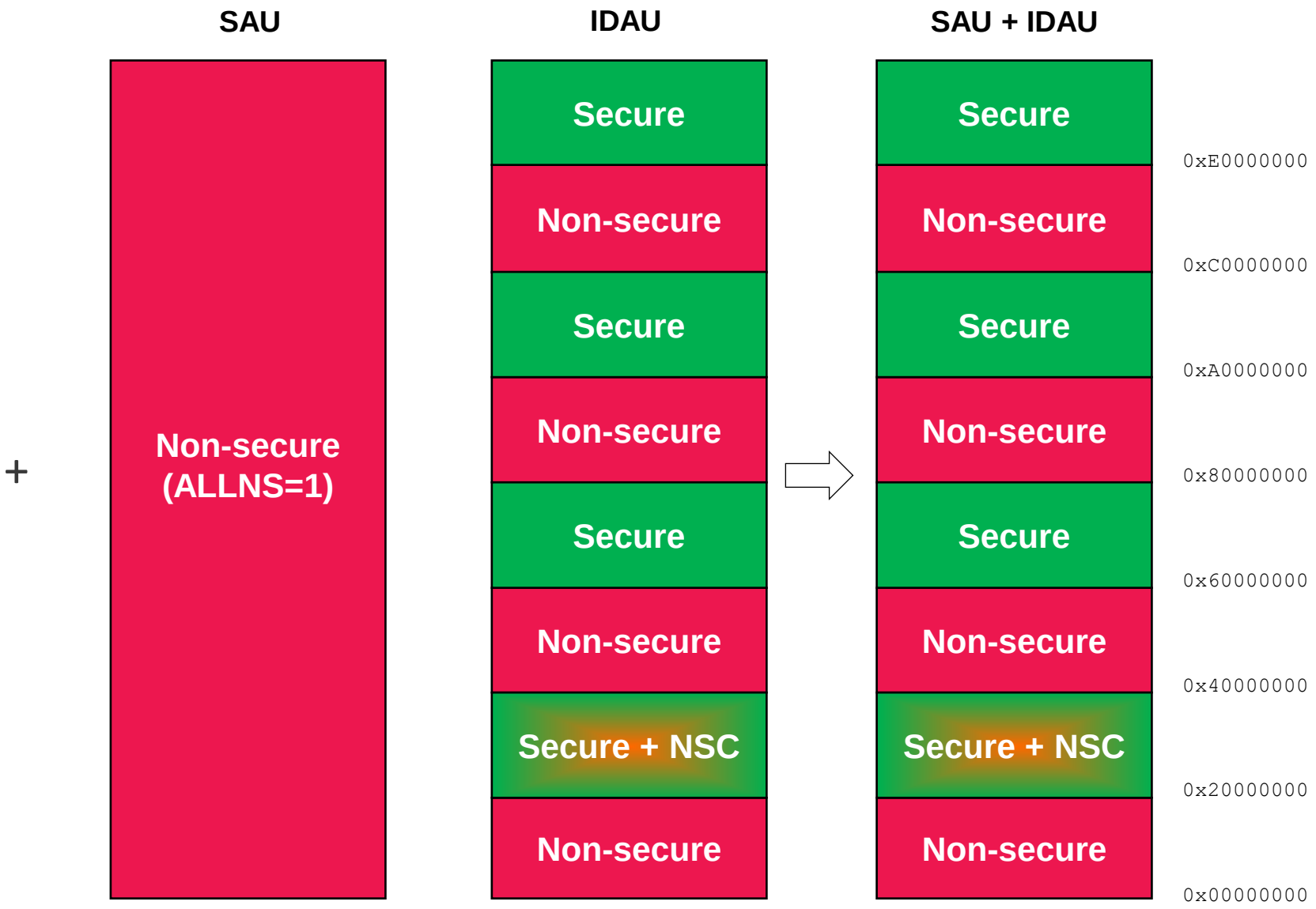
SAU is disabled at reset

- Default security is IMPLEMENTATION DEFINED

SAU marks all memory as Secure by default

- Controlled by SAU_CTRL.ALLNS

Boot security map – example configuration (2)



IDAU provides default security mappings

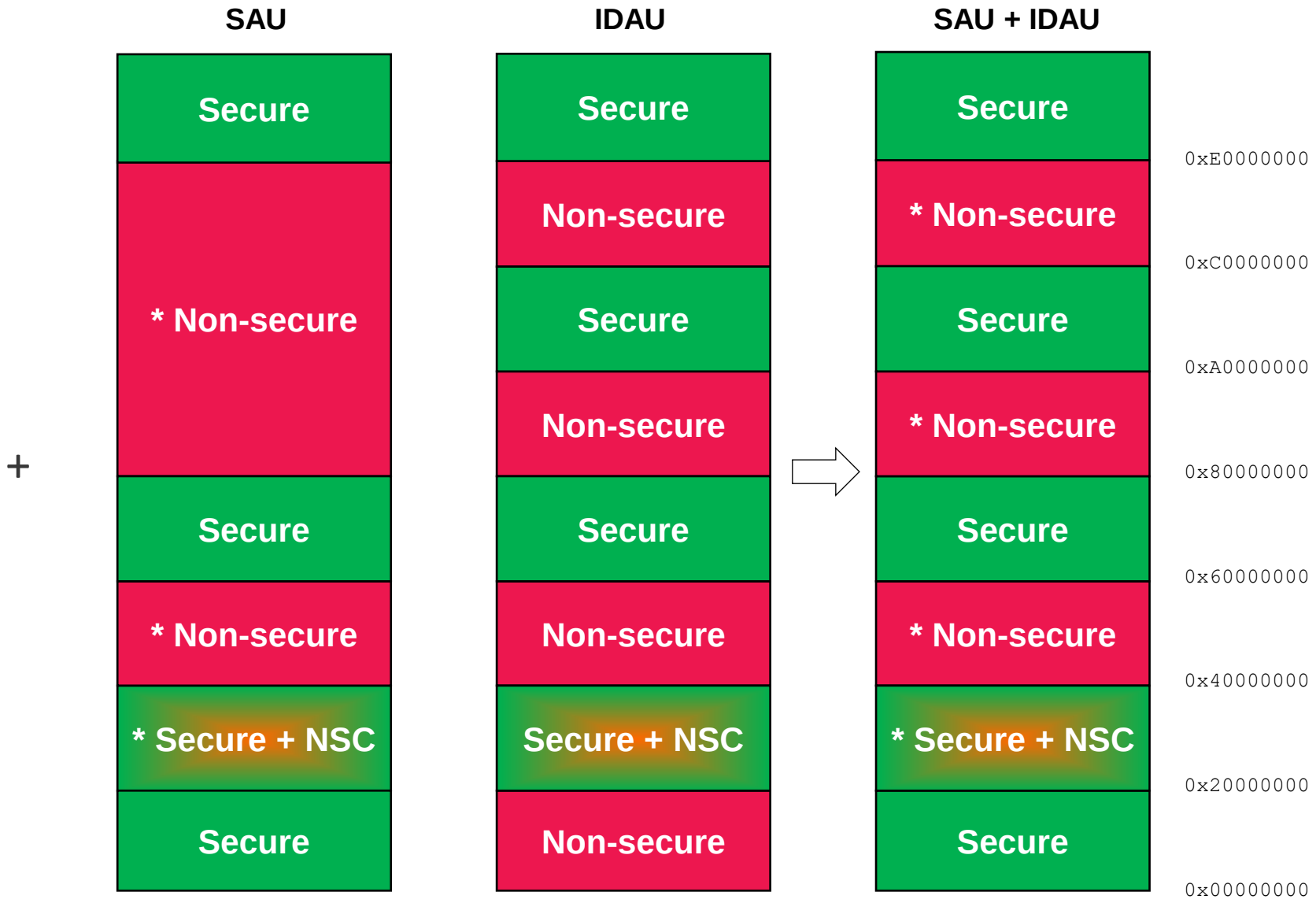
SAU is disabled at reset

- Default security is IMPLEMENTATION DEFINED

SAU marks all memory as Non-secure by default

- Controlled by SAU_CTRL.ALLNS

Runtime security map example configuration



The SAU can reconfigure attributes

The Armv8-M architecture permits a maximum of 256 SAU regions

When the SAU is enabled, memory not covered by an enabled SAU region is Secure

SAU region configuration

`SAU_TYPE` indicates the number of available regions

- Implementation with a maximum of 8 regions

31 … 8	7 … 4	3 … 2	1 … 0
RES0			SREGION

`SAU_RNR` selects the region to be programmed

31 … 8	7 … 4	3 … 2	1 … 0
RES0			REGION

`SAU_RBAR` selects the base address of a region

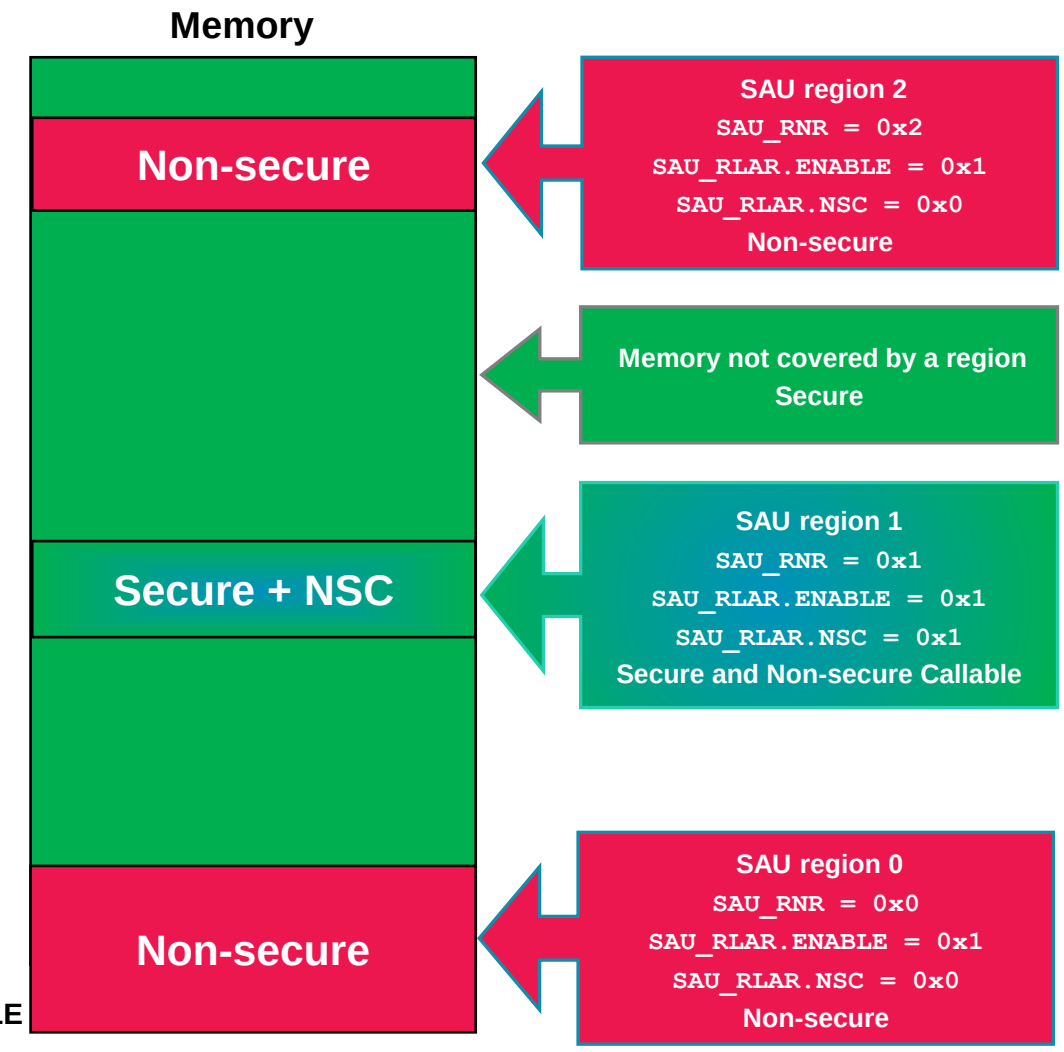
31 … 5	4 … 3	2 … 1	0
BADDR		RES0	

`SAU_RLAR` selects the limit address of a region

31 … 5	4 … 3	2 … 1	0
LADDR		RES0	

- **`NSC`** bit marks region as Non-secure or NSC
- **`Enable`** bit to enable/disable a region

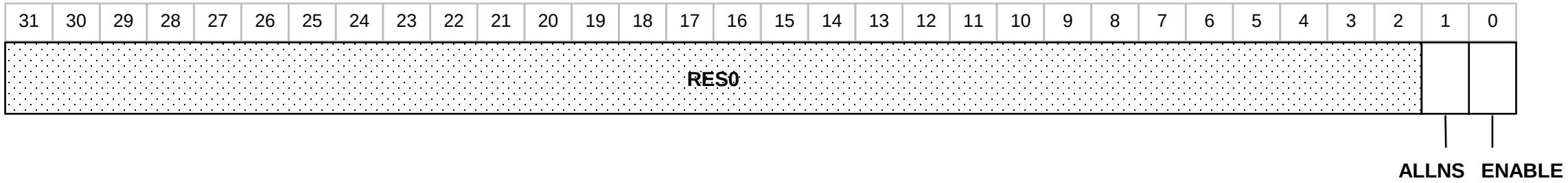
NSC ENABLE



Enabling the SAU

SAU_CTRL contains two programmable bits

- **ENABLE**
  - Set to 1 to enable the SAU, set to 0 to disable the SAU
  - Resets to an IMPLEMENTATION DEFINED value on a Warm reset
  - If this register resets to 1, the SAU region registers also reset to an IMPLEMENTATION DEFINED values
- **ALLNS**
  - Determines whether the SAU treats all memory as secure or Non-secure when SAU_CTRL.ENABLE is set to 0
  - Resets to an IMPLEMENTATION DEFINED value on a Warm reset



On Cortex-M23, Cortex-M33 and Cortex-M55, SAU_CTRL resets to 0

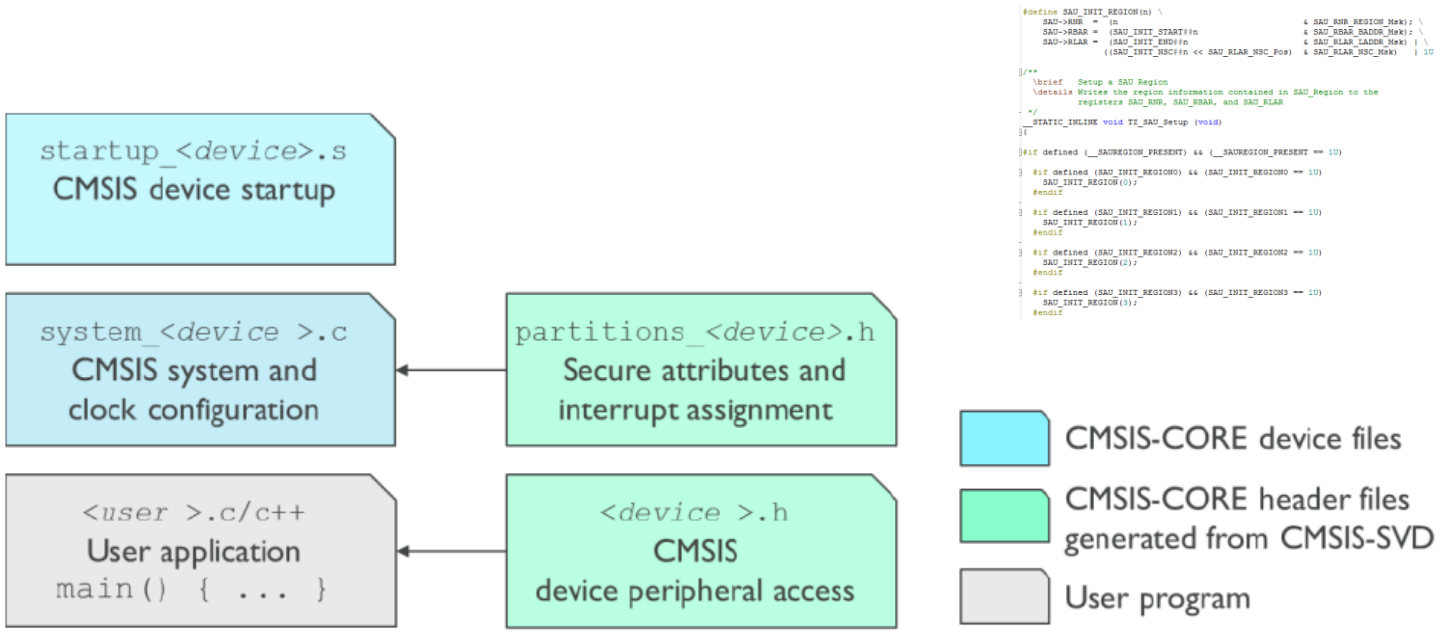
- All memory is secure out of reset

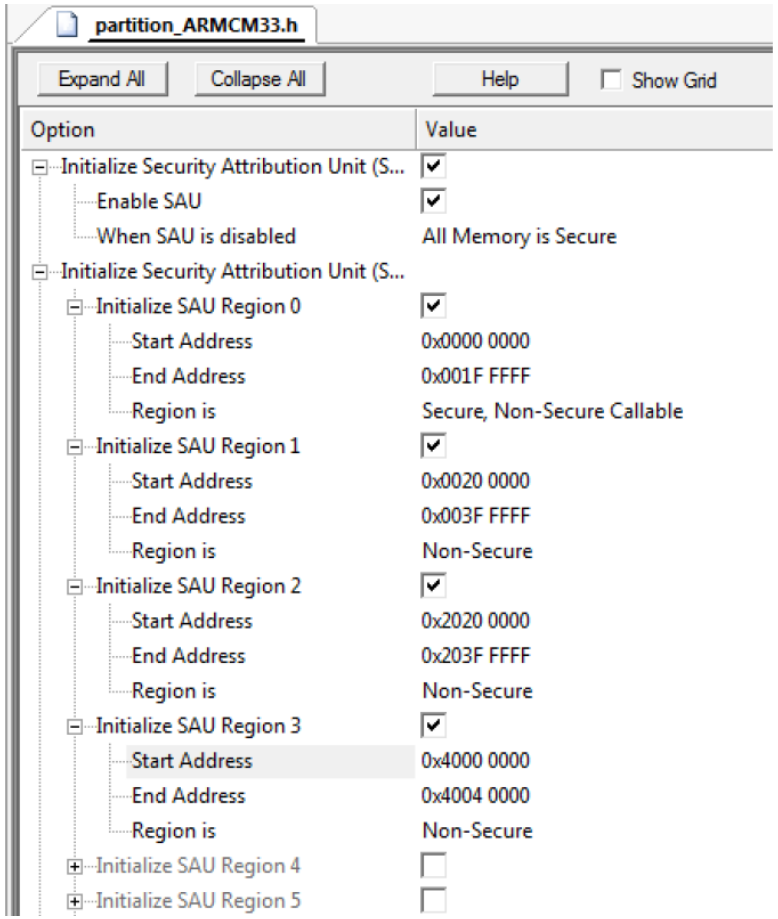
Configuring the SAU with CMSIS

CMSIS-CORE now provides partition_<device>.h

- `TZ_SAU_Setup()` used to configure SAU regions

Some software tools provide SAU configuration wizards





Agenda

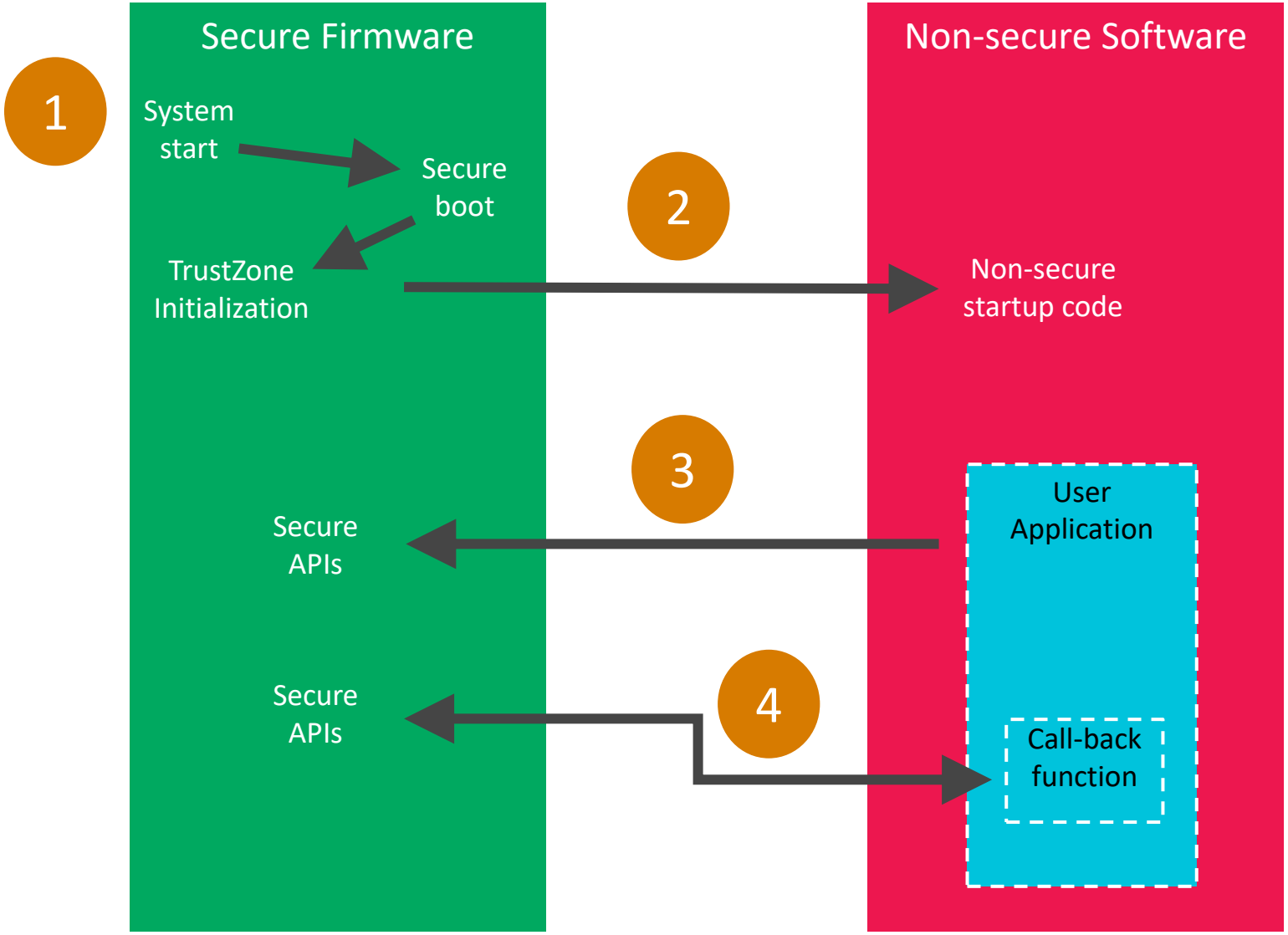
Overview

Memory Configuration

Function Calls & Toolchain Support

Exceptions

Branching between Secure and Non-secure states



Function calls using branch instructions

Instruction	Function	Description
B <imm>	Branch	Branches to PC-relative offset specified by the <imm> field.
BL <imm>	Branch with Link	Branches to PC-relative offset specified by the <imm> field and updates the Link Register (LR) to contain the return address.
BX <Rn>	Branch and eXchange	Branches to an address pointed to by Rn. Can used at the end of a subroutine to return to caller (BX lr).
BLX <Rn>	Branch with Link and eXchange	Branches to an address pointed to by Rn and updates the LR with the return address.
BXNS <Rn>	BX to Non-secure state	Works like the BX instruction but also causes a security state transition from Secure to Non-secure. If Rn[0] = 0.
BLXNS <Rn>	BLX to Non-secure state	Works like the BLX instruction but also causes a security state transition from Secure to Non-secure. If Rn[0] = 0.

Note:

The eXchange in BLX/BX was for switching between Arm and Thumb state by toggling the least significant bit (LSB) of branch address contained in Rn.

- Rn[0] = 0 : A32 state
- Rn[0] = 1 : T32 state

This transition is not available in Armv8-M as it only supports Thumb state and LSBs for branch target addresses are always set to 1.

The LSB of branch address has now been re-purposed to indicate transition from Secure to Non-secure state.

Arm C Language Extensions (ACLE)

Arm C Language Extensions (ACLE) specifies source language extensions

The Armv8-M Security Extension is sometimes known as the Cortex-M Security Extension (CMSE)

- The “Armv8-M Security Extensions: Requirements on Development Tools” document lists requirements that development tools must satisfy in order to work with CMSE
  http://infocenter.Arm.com/help/topic/com.Arm.doc.ecm0359818/ECM0359818_Armv8m_security_extensions_reqs_on_dev_tools_1_0.pdf

CMSE supports the generation of

- New instructions: `SG`, `BXNS`, `BLXNS`, `TT`
- Secure software
- Import libraries to allow Non-secure software to call into secure software

Compiler toolchains like Arm Compiler 6 and GCC are CMSE compliant

- Include `<Arm_cmse.h>` header
- Compile with `-mcmse`

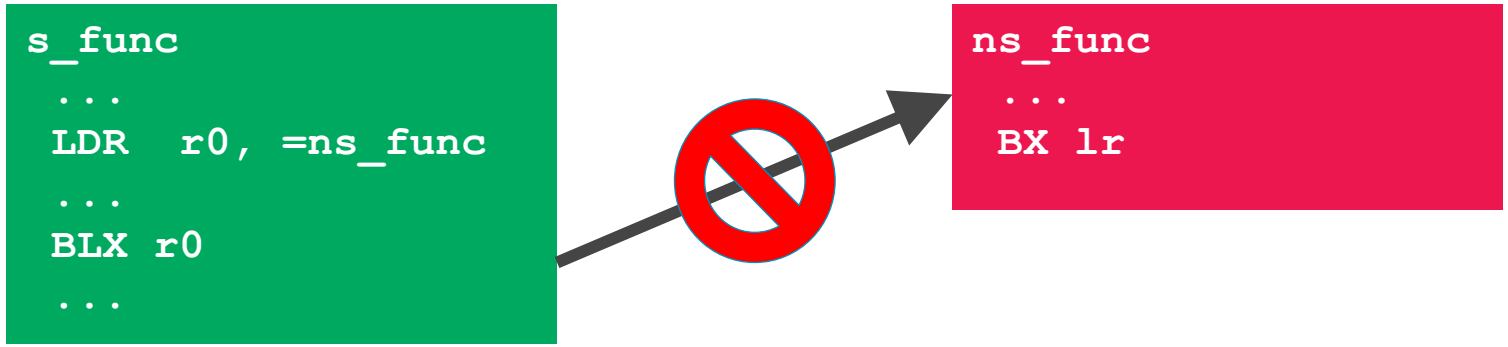
Calling Non-secure code from secure code (1)

Secure code must call Non-secure code using an interstating branch instruction

- `BXNS` or `BLXNS`

Attempting to switch to Non-secure state via any other branch will result in

- An INVTRAN SecureFault Armv8-M Mainline implementations
- A Secure HardFault in Armv8-M Baseline implementations



Calling Non-secure code from secure code (2)

Before executing a `BXNS` or `BLXNS` instruction

1. **Save all active non-banked registers by copying them to secure memory**
2. **The branch target address must have the LSB set to 0 and reside in Non-secure memory**
3. **Clear all non-banked registers except:**
   - Link register (BLXNS only)
   - Registers that hold arguments for the call
   - Registers that do not hold secret information

C language extensions can be used by secure software to create a function pointer to a Non-secure function

- ACLE defines a CMSE function attribute:
  `__attribute__((cmse_nonsecure_call))`

```
typedef void __attribute__((cmse_nonsecure_call)) ns_func(void);
ns_func *FunctionPointer;
FunctionPointer = (ns_func *)(0x21000000u);
FunctionPointer();
```

Example disassembly for CMSE-compliant Non-secure to secure function call

```
PUSH    {r0-r12}
MOVW    r0,#0x0
MOVT    r0,#0x2100
MOV     r1,r0
MOV     r2,r0
MOV     r3,r0
MOV     r4,r0
MOV     r5,r0
MOV     r6,r0
MOV     r7,r0
MOV     r8,r0
MOV     r9,r0
MOV     r10,r0
MOV     r11,r0
MOV     r12,r0
MSR     APSR_nzcvq,r0
BLXNS   r0
```

Push r0-r12 onto secure stack

Move address into r0 (LSB=0)

Clear registers (overwrite with Non-secure branch address)

Non-secure branch

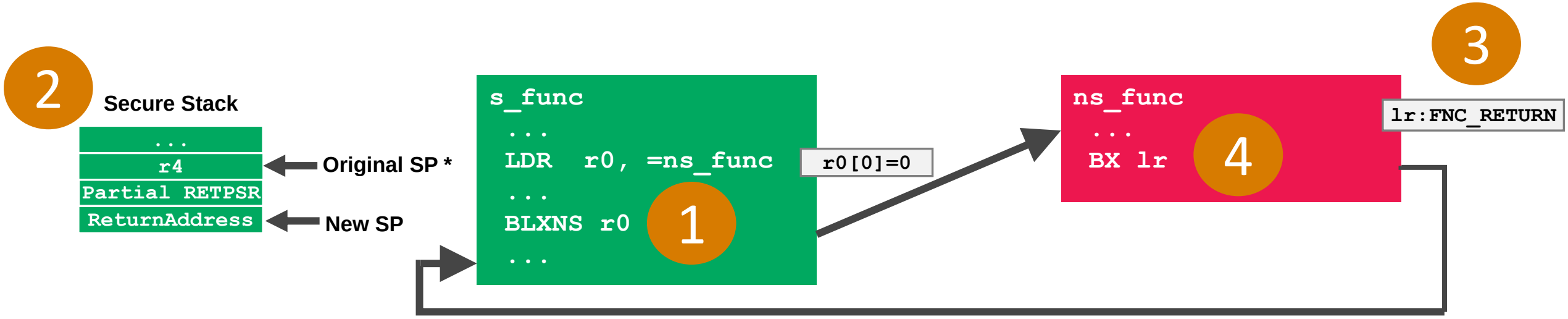
Calling Non-secure code from secure code (3)

`BLXNS` pushes the function return address onto the secure stack to hide it from Non-secure code

- The special value `FNC_RETURN` is stored in `lr`
- The `IPSR` and `CONTROL.SFPA` are also stacked as the “Partial RETPSR”
- * Also, if necessary, a word of padding is added to align the stack to an 8 byte boundary

Non-secure code branching to `FNC_RETURN` will:

- Unstack the real return address from the secure stack
- Branch back into Secure state



Calling secure code from Non-secure code (1)

Non-secure code can branch into secure code

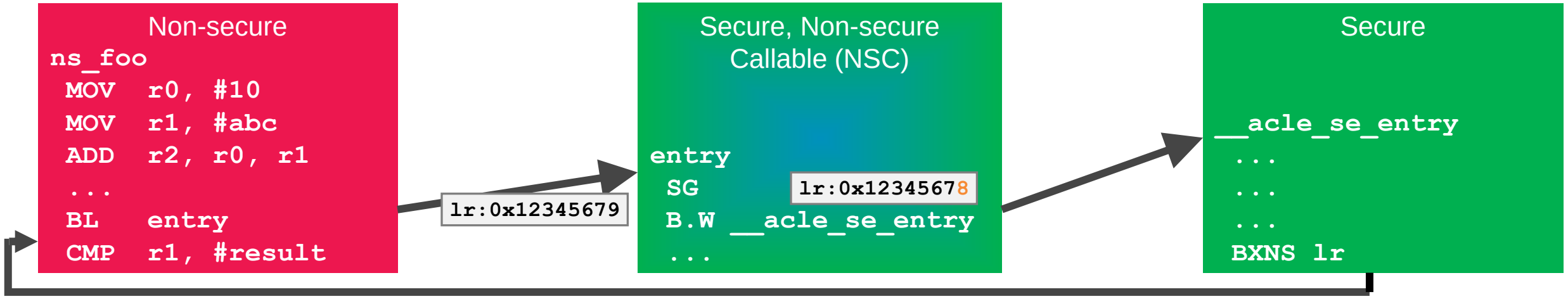
- This allows secure libraries to be used by Non-secure applications

The branch target address:

- Must be mapped as **Secure and Non-secure Callable** by the SAU/IDAU
- Must contain a **Secure Gateway** (**SG**) instruction

When executed from Secure, NSC memory, the SG instruction

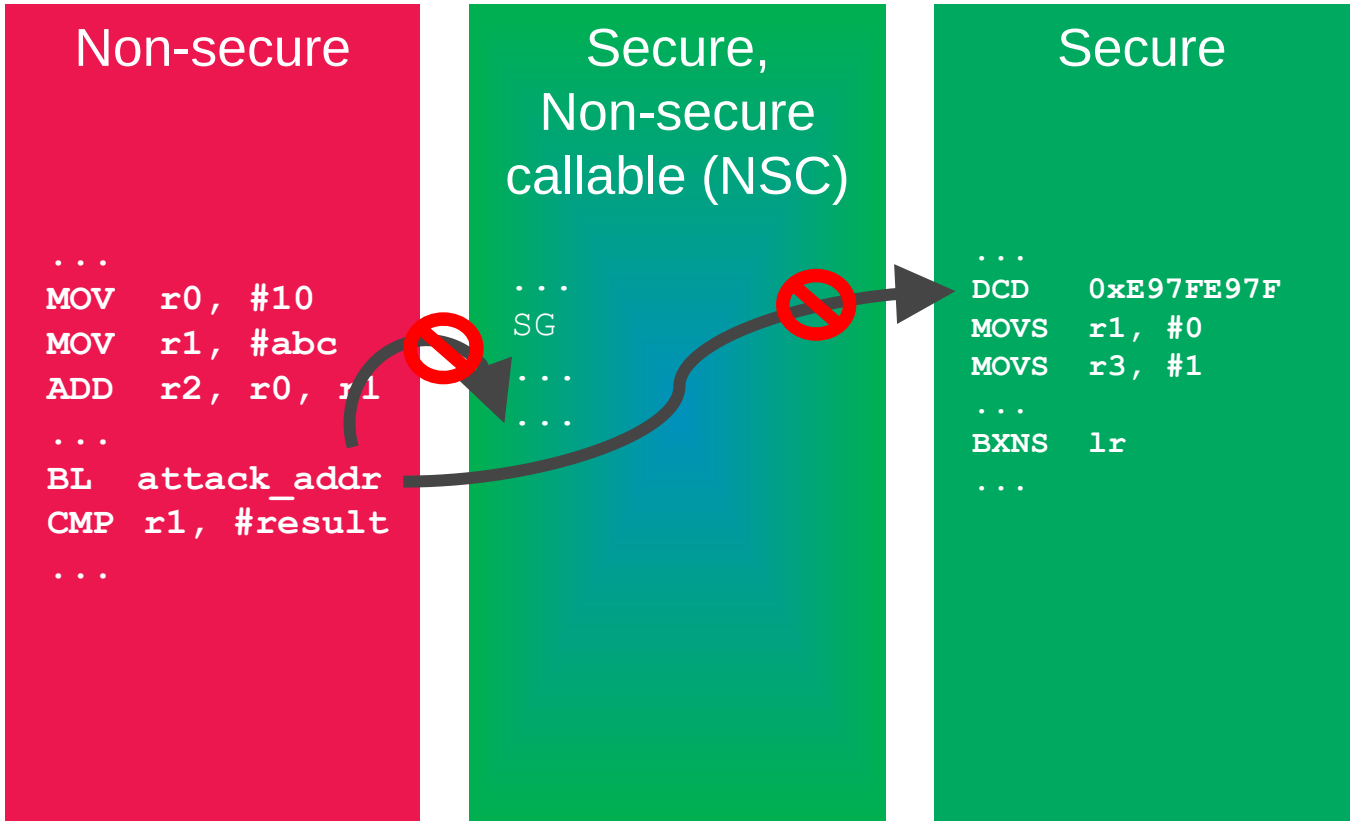
- Changes the security state to Secure
- Sets `lr[0]` to `0` to allow the secure code to return using a `BXNS` instruction



Calling secure code from Non-secure code (2)

Non-secure branches to unauthorized secure addresses will cause a fault

- SecureFault exception in Armv8-M Mainline
- HardFault in Armv8-M Baseline



- Branches to an address in Secure, NSC memory that do not contain an SG instruction will fault
- Branches to ANY address in Secure memory will fault
  - Even SG instructions
- SG instructions are not expected in Secure memory but the bit pattern could be there because of:
  - Uninitialized memory
  - Literal data in executable memory

Creating an import library in Arm Compiler 6 (1)

Entry functions are prototyped as normal in an interface header

secure_interface.h

```
int entry1(int x);
int entry2(int x);
```

Function body is decorated with `__attribute__((cmse_nonsecure_entry))`

secure_library.c

```
#include <Arm_cmse.h>
int __attribute__((cmse_nonsecure_entry)) entry1(int x) {
  ...
}
int __attribute__((cmse_nonsecure_entry)) entry2(int x) {
  ...
}
```

An import library is generated at the link stage

```
Armclang -c --target=Arm-Arm-none-eabi -march=Armv8-m.base -mcmse secure.c
Armlink --import-cmse-lib-out=export\secure_library.o --scatter=secure.scf -o secure.axf secure.o
```

Using the import library

Non-secure code includes the interface header and calls the functions as usual

non_secure_main.c

```
#include "secure_interface.h"

int main(void) {
    int val1, x;
    val1 = entry1(x);
    return 0;
}
```

Non-secure code links against the import library

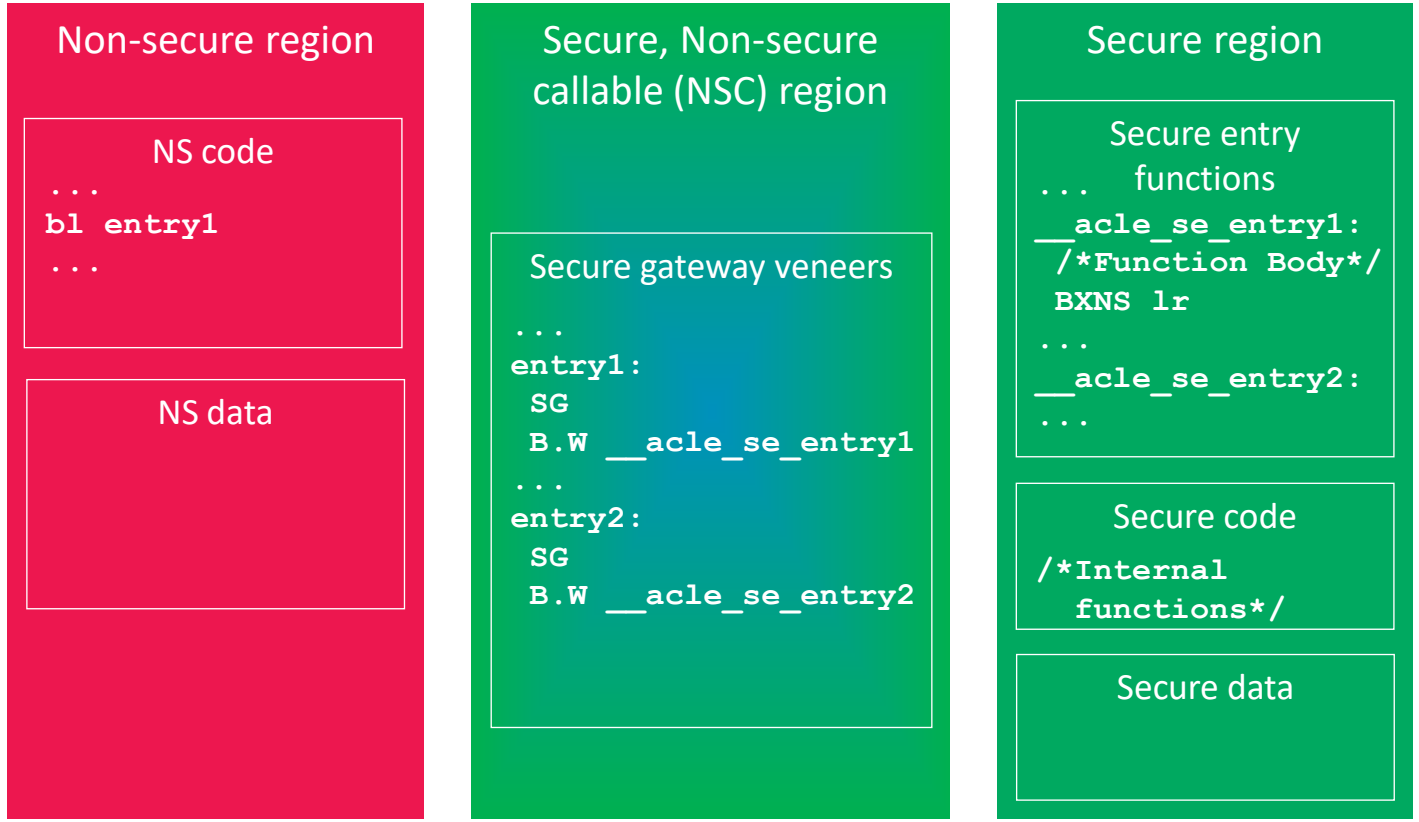
```
Armlink -o nonsecure.axf --cpu=8-M.base --scatter nonsecure_scf.txt nonsecure.o
system_Armv8-M.o startup_Armv8-M.o ..\export\secure_library.o
```

Non-secure code calls the function using the standard calling mechanism

```
main
...
        0x21000208:    f002f944    ..D.    BL        entry1
...
```

Secure gateway veneers

The toolchain can generate “secure gateway veneers” to allow access to secure code via SG instructions



NSC veneers in Arm Compiler 6 (1)

Body of secure entry functions will be marked with two labels

```
__acle_se_entry1:
entry1:
    .fnstart
@ BB#0:
    .save    {r7, lr}
    push {r7, lr}
    ...
    bl func1
    pop.w {r7, lr}
    ...
    bxns lr
```

Placed in secure memory

```
LOAD_REGION 0x0 0x3000
{
    EXEC_SECURE 0x0 ANY_SIZE 0xF00 0x1000
    {
        *(+RO,+RW,+ZI)
    }
...
}
```

NSC veneers in Arm Compiler 6 (2)

Veneers are generated by the secure link step

- Labeled using a symbol derived from the function name
- Call `__acle_se_<name>` label

```
entry1
        0x30000000:    e97fe97f    ....    SG          ; [0x2ffffe08]
        0x30000004:    f000b80c    ....    B.W         __acle_se_entry1 ; 0x30000020
entry2
        0x30000008:    e97fe97f    ....    SG          ; [0x2ffffe10]
        0x3000000c:    f000b80e    ....    B.W         __acle_se_entry2 ; 0x3000002c
```

Veneer code is placed separately in the scatter file into Non-secure Callable memory

```
LOAD_NSCR 0x30000000 0x1000
{
    EXEC_NSCR 0x30000000 0x1000
    {
        *(Veneer$$CMSE)
    }
}
```

TT instruction

New Test Target (TT) instruction returns the MPU and SAU configuration for an address

- MPU, SAU and IDAU region details

`TT{cond}{q} Rd, Rn`

- **Rn** contains the address to be tested
- **Rd** returns the attributes of the address

Secure functions may be passed pointers when called from Non-secure code

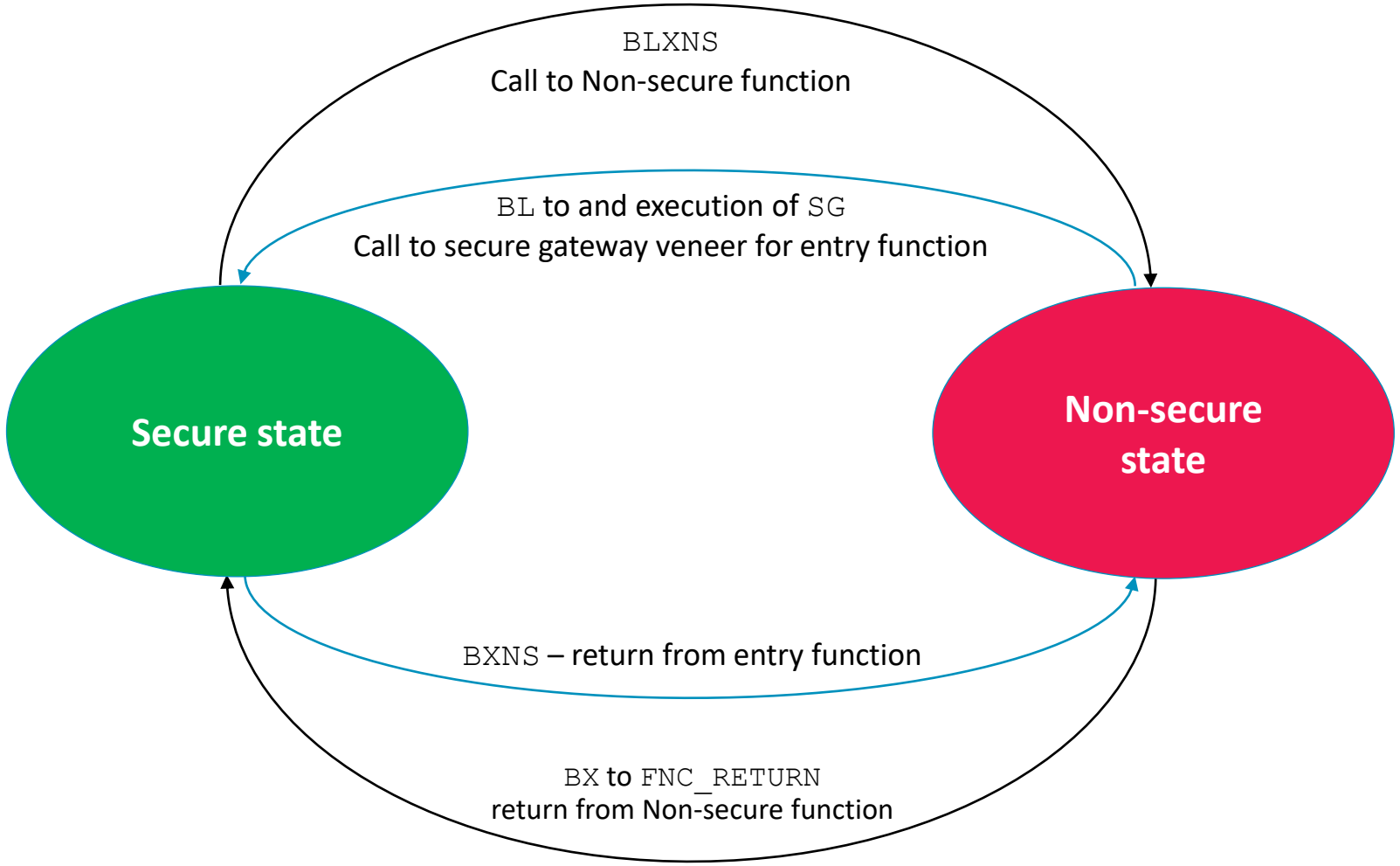
- **TT** can validate that the calling function has the rights to access the memory
  - Doesn't break privilege
- **TT** can validate that the address is Non-secure
  - Isn't an attack on Secure memory

[7:0]	MREGION	MPU region
[15:8]	SREGION	SAU region
[16]	MRVALID	Is MREGION valid?
[17]	SRVALID	Is SREGION valid?
[18]	R	Is address readable?
[19]	RW	Is address RW?
[20]	NSR	Is Non-secure && readable?
[21]	NSRW	Is Non-secure && RW?
[22]	S	Is address Secure?
[23]	IRVALID	Is IREGION valid?
[31:24]	IREGION	IDAU region

CMSE support for TT

- `cmse_address_info_t cmse_TT(void *p)`
- `void *cmse_check_address_range(void *p, size_t size, nt flags)`

Security state changes using software



Agenda

Overview

Memory Configuration

Function Calls & Toolchain Support

Exceptions

Interrupts and exceptions

Interrupts can be programmed as secure or Non-secure interrupts

Some system exceptions are banked (e.g. SysTick)

New SecureFault exception

Banked System Control Block (SCB) registers

- Two VTOR – Separate vector tables for Secure exceptions and Non-Secure exceptions
- Non-Secure SCB registers can be accessed from Secure side via alias addresses

Priority of Secure exceptions/interrupts can share same levels as Non-secure's, or can be higher priority (programmable)

Exception priorities overview

Name	Exception #	Exception Priority #	Security
Interrupts #0 - N	16 to 16 + N	0-255 (programmable)	**Configurable**
SysTick	15	0-255 (programmable)	**Banked**
PendSV	14	0-255 (programmable)	**Banked**
DebugMonitor	12	0-255 (programmable)	**Configurable**
SVCall	11	0-255 (programmable)	**Banked**
SecureFault	**7**	**0-255 (programmable)**	**Secure**
UsageFault	6	0-255 (programmable)	**Banked**
BusFault	5	0-255 (programmable)	**Configurable**
MemManage	4	0-255 (programmable)	**Banked**
Non-secure HardFault	**3**	**-1**	**Non-secure**
Secure HardFault	**3**	**-3 or -1 (programmable)**	**Secure**
Non Maskable Interrupt (NMI)	2	-2	**Configurable**
Reset	1	**-4**	**Secure**

The lower the priority number, the higher the priority level

NMI, HardFault and BusFault default to be Secure, but can be set to NS

System handler priority

The priorities of the configurable system exceptions are configured by the System Handler Priority Registers

- **SHPR1** – MemManage, BusFault, UsageFault, SecureFault
- **SHPR2** – SVCall
- **SHPR3** – PendSV, SysTick, DebugMonitor

BusFault, HardFault, and NMI can be made Non-secure using AIRCR.BFHFNMINS

- "BusFault, HardFault, and NMI Non-secure enable"

SHPRn registers are banked by security state

- Secure code can access Non-secure aliases, **SHPRn_NS**, at 0xE002ED18 - 0xE002ED20

The 8-bit Priority Level is divided into two fields

- Group Priority and Sub-Priority

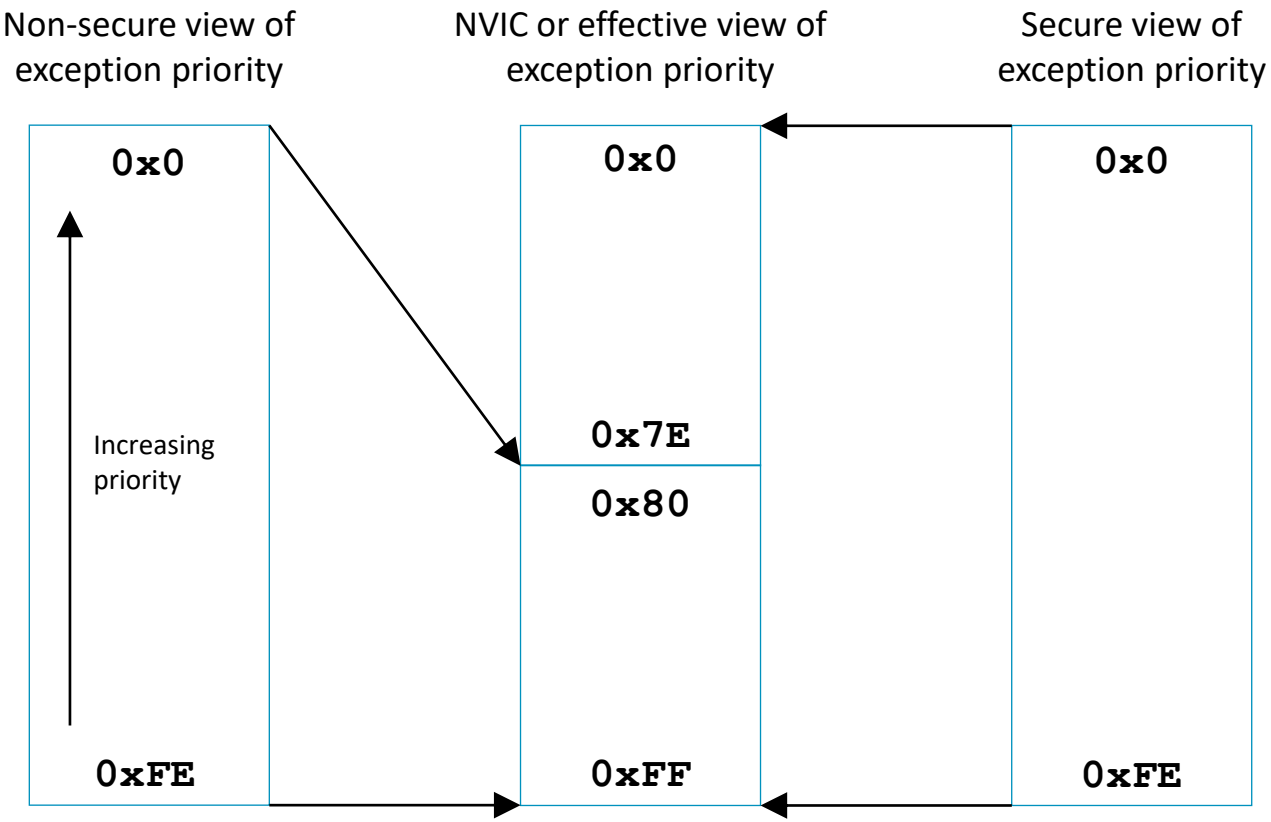
Secure grouping configured by AIRCR_S.PRIGROUP

Non-secure grouping configured by AIRCR_NS.PRIGROUP

Secure exception prioritization

Non-secure exceptions can be forced into the lower half of the priority range

- Using **`AIRCR_S.PRIS`**



Gives one less bit of priority to Non-secure code; e.g.

Write **`0x0`** to **`SHPR2_NS`**

- Set NS SVCall to 'highest' priority

Effective priority in NVIC is **`0x80`**

- Non-secure view **`0x0`**
- NVIC view **`0x80`**

Write **`0x0`** to **`SHPR2_S`**

- Secure SVCall priority is **`0x0`**

Configuring the NVIC

NVIC control registers are security banked

- **`NVIC_*`** at address **`0xE000E100`**
- Secure code can access **`NVIC_*_NS`** alias from **`0xE002E100`**

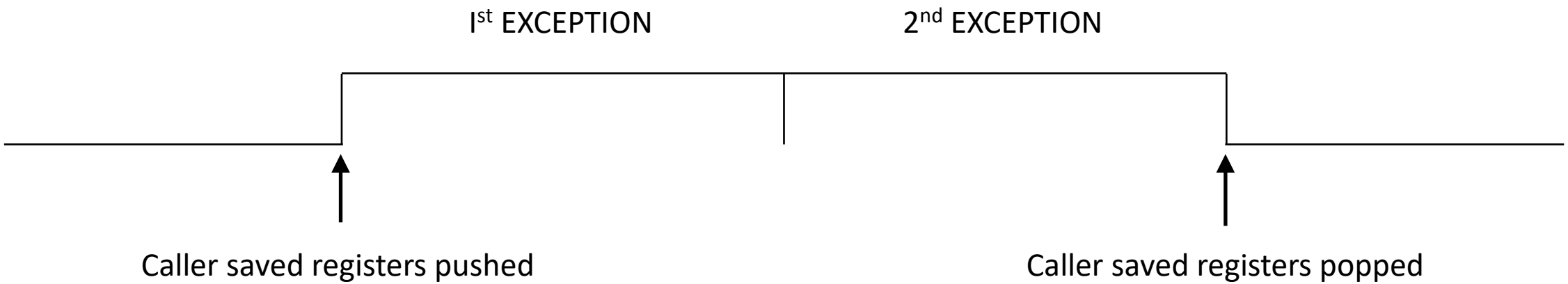
`NVIC_ITNSn` Registers – Write `0b1` to target Non-secure state

	Bit 31				Bit 0
`0xE000E380`	**`ITNS31`**	…	**`ITNS2`**	**`ITNS1`**	**`ITNS0`**
`0xE000E384`	**`ITNS63`**	…	**`ITNS34`**	**`ITNS33`**	**`ITNS32`**

Configuring an interrupt as Secure makes NS accesses to its NVIC configuration bits RAZ/WI

Registers reset to zero (i.e. all interrupts are Secure)

Exceptions in Armv7-M



Hardware automatically pushes / pops caller saved registers

- ISR's are just C functions
- No assembly wrappers needed
- Low interrupt latency

Pending exception tail-chained without unstacking / restacking

EXC_RETURN

On interrupt entry, the LR stores the EXC_RETURN value

- Previous LR is saved to stack so it can properly be restored

EXC_RETURN is used to correctly return from the interrupt

Bits	31:24	23:7	6	5	4	3	2	1	0
FIELD	0xFF	RES1	S	DCRS	FType	Mode	SPSEL	RES0	ES
Notes			Return stack (1=Secure, 0=NS)	Default Callee Registers Stacking	Stack Frame Type (0=Extended Stack Frame, 1=Standard Stack Frame – Integer only)	The mode that was stacked from 1 (Thread) 0 (Handler)	0 (main stack) 1 (process stack)		Exception State. Security state taken to (1=Secure, 0=NS)

Extended to indicate:

- Which stack was used (Secure/Non-secure)
- If registers were already stacked
- The security state that handled the exception

Taking an exception

The processor automatically transitions to the security state targeted at an exception

- Secure exception handlers do not need to start with **`SG`**
- Exception vector must be in the correct memory type– otherwise a SecureFault is raised
  - E.g. Vector for a Secure exception must be in Secure memory

VTOR register is banked

- Secure exceptions use **`VTOR_S`**
- Non-secure exceptions use **`VTOR_NS`**

On Non-secure exception entry if transitioning Secure to Non-secure

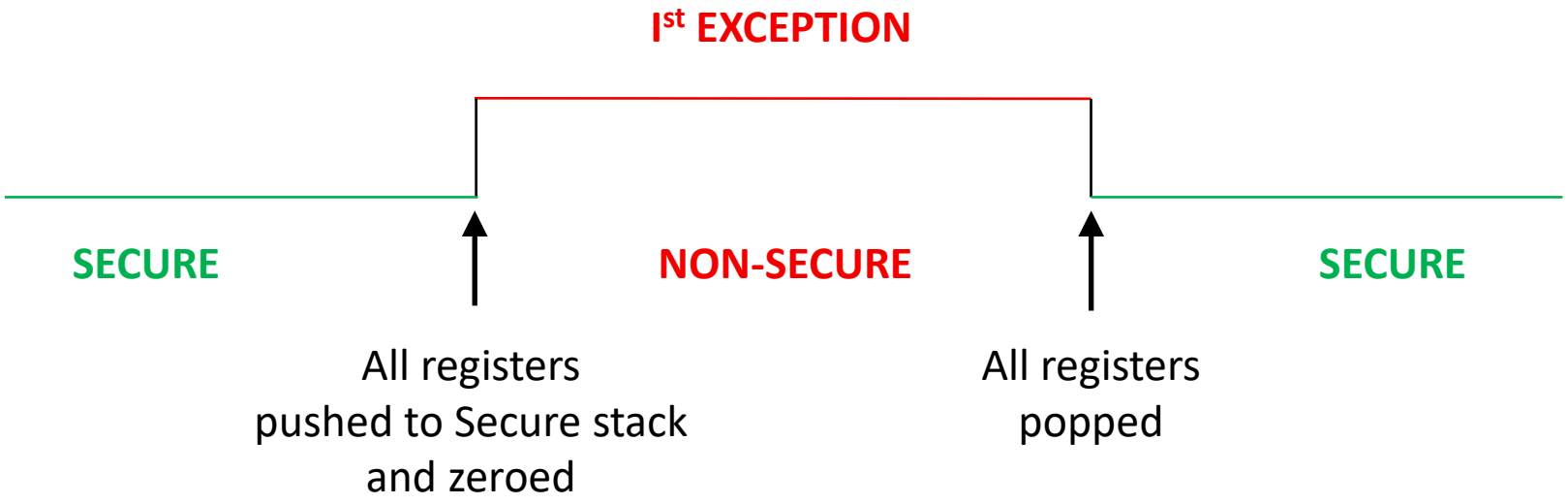
- Callee saved integer registers are also pushed to the stack (r4-r11)
- Callee saved registers are zeroed (r4-r11)
- Caller saved registers are zeroed (r0-r3, r12, APSR and EPSR)

Stack frame can be extended to hold floating-point registers

- Lazy Context preservation can be used as in Armv7-M
- If FP data is secure both callee and caller saved FP registers are pushed to the stack, and then cleared

Extended `EXC_RETURN` indicates which registers need to be popped

Secure → Non-secure exceptions



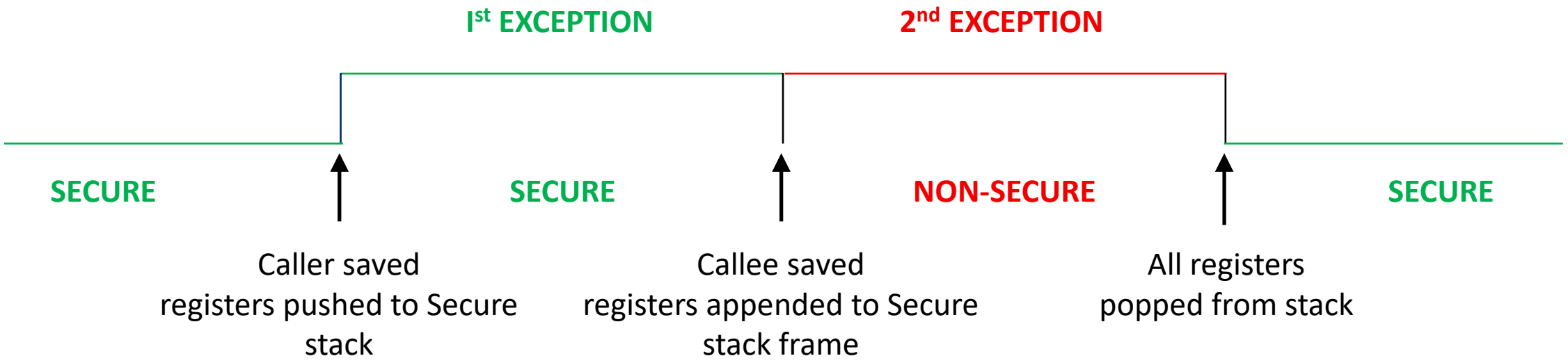
Hardware detects transition from secure → Non-secure

- All registers stacked
- Register values cleared

Registers saved on Secure stack

Non-secure code can't see or alter any Secure registers

Chaining secure and Non-secure exceptions



Latency of 1st exception not affected

Stack frame extended when 2nd exception taken

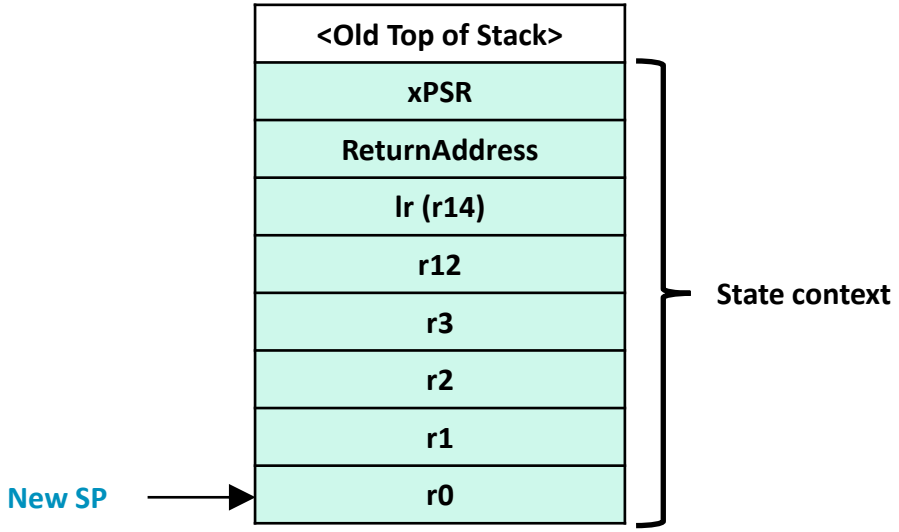
- Approximately the same latency as saving basic stack frame

All stacking / unstacking done by hardware

Stack frame layout (no FP)

On taking an exception from Non-secure state

- Hardware saves state context onto the stack pointed to by `sp`



Stack Frame layout (no FP)

On taking an exception from Non-secure state

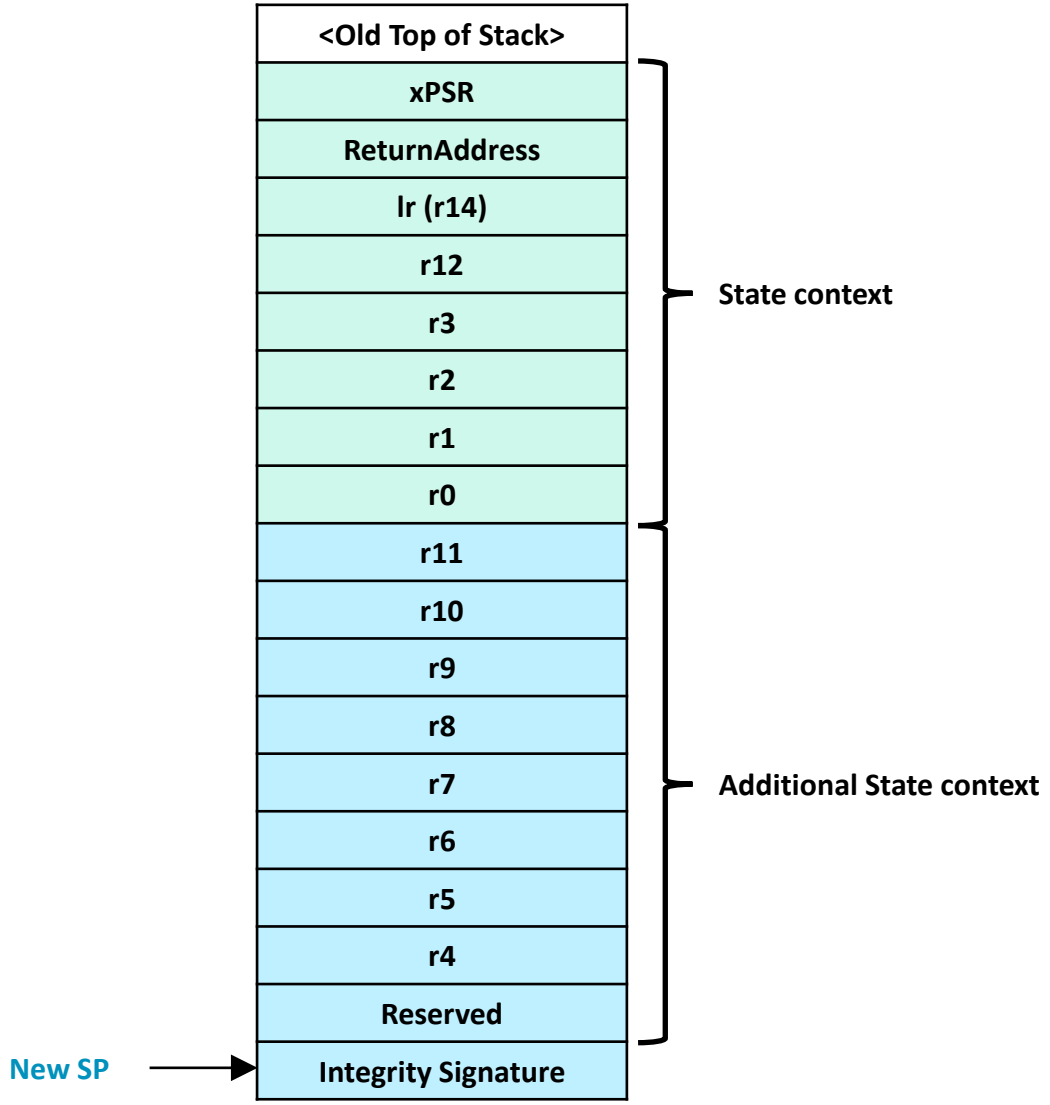
- Hardware saves state context onto the stack pointed to by `sp`

On taking an exception from Secure to Non-secure state

- The stack frame is extended
- Hardware also saves Additional state context

Additional state context includes:

- r4-r11
- An Integrity Signature
- A reserved word of padding to maintain alignment



Register values after context stacking (no FP)

After context stacking r0-r3, r12, xPSR become:

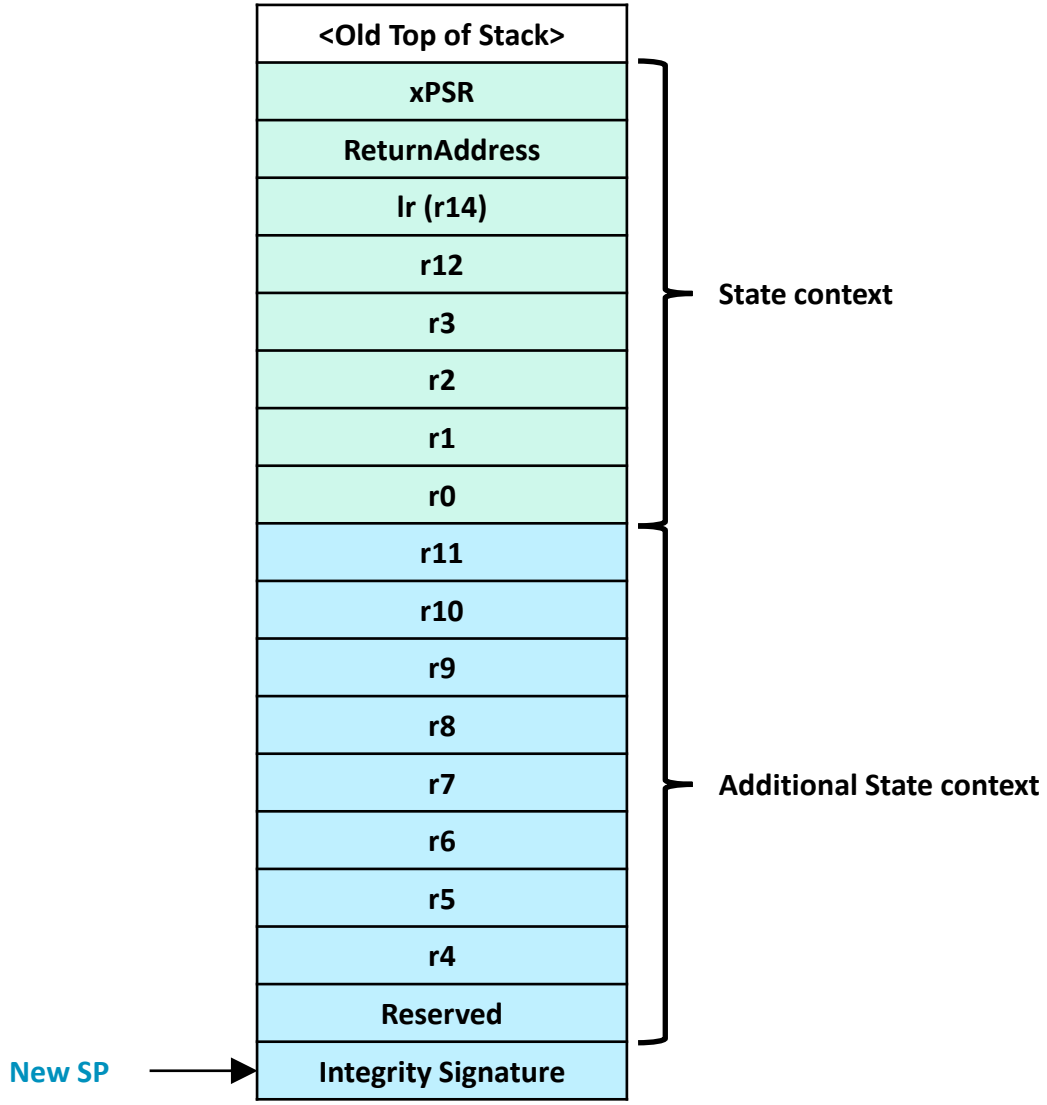
- **Zero** if exception is taken to Non-secure state
- Otherwise UNKNOWN

If the stack frame is extended and the Background Security state is Non-secure

- r4-r11 remain unchanged

If the stack frame is extended and the Background Security state is Secure:

- r4-r11 become **Zero** if exception is taken to Non-secure state
- Otherwise r4-r11 are UNKNOWN



Integrity signature

Callee saved register section of the stack frame contains an integrity signature

- If upon unstacking such a frame the value is incorrect a SecureFault is raised

Stack Frame Type Check (`SFTC`) flag must match the `FType` field in `EXC_RETURN`

- Non-secure code cannot influence how the stack frame is interpreted

This value also prevents use of an exception frame as a function return frame

- If software branches to **`FNC_RETURN`** instead of **`EXC_RETURN`** the Integrity Signature will be interpreted as the return address
- Top byte of the signature is **`0xFE`** which will cause a MemManage fault if used as a branch target

31	1	0
1111 1110 1111 1010	0001 0010 0101 101	SFTC

Stack Frame layout with Floating-point Extension

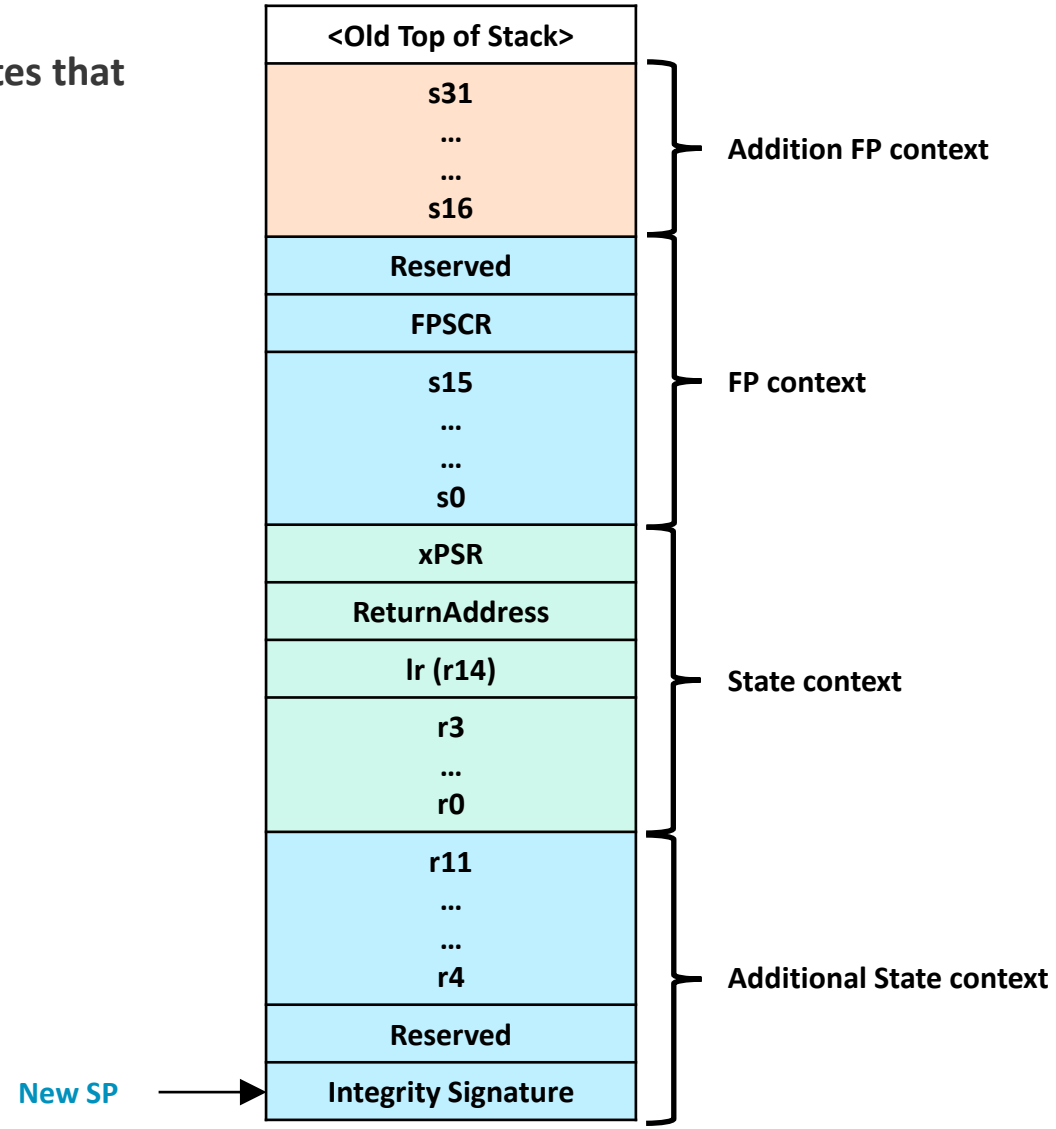
If the Floating-point Extension is implemented `CONTROL.FPCA` indicates that floating point context is active

- Only active context needs to be stacked
- `CONTROL_S.SFPA` indicates that the FP context is Secure data

Standard FP context is pushed to the current stack

If exception entry is to Non-secure state:

- The stack frame is extended
- Additional FP context is pushed to the stack



Initializing secure stacks

In the normal course of execution on entry into Non-secure state Secure stacks will be topped with either

- The Integrity Signature or
- A valid Secure return address

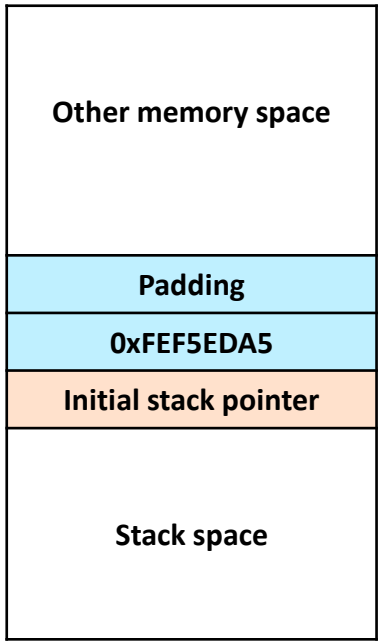
This prevents malicious use of FNC_RETURN or EXC_RETURN

If a Secure stack is not initialized before entering the Non-secure state this protection might not be present

The problem can be mitigated by *sealing* the stack

- This involves placing a special value at the address above the initial stack pointer
- Arm recommends using the value 0xFEF5EDA5

Both MSP_S and PSP_S should be sealed



References

Armv8-M Architecture Reference Manual:

https://developer.Arm.com/products/architecture/m-profile/docs/ddi0553/latest/Armv8-m-architecture-reference-manual

Armv8-M Security Extensions: Requirements on Development Tools - Engineering Specification:

https://developer.Arm.com/products/architecture/m-profile/docs/ecm0359818/latest/Armv8-m-security-extensions-requirements-on-development-tools-engineering-specification

Secure software guidelines

https://developer.Arm.com/products/architecture/m-profile/docs/100720/latest/secure-software-guidelines

TrustZone for Armv8-M Connected Community

https://community.Arm.com/processors/trustzone-for-Armv8-m/

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה