Armv8-M Mainline Exception Handling

Learning objectives (1)

At the end of this module you will be able to:

Describe the M-profile exception model including:

- The built-in Nested Vectored Interrupt Controller (NVIC) and how it is optimised for embedded systems
- Exception types
- Vector table
- Reset behaviour
- Basic exception handling mechanism

Describe reset behaviour

Describe what happens on exception entry and exception return

Discuss expected exception latency and architectural optimizations that can help speed up exception handling

Learning objectives (2)

Enable, disable, and pend exceptions, and find out what state an exception is in

Describe the prioritization scheme for exceptions

Describe the different types of interrupt sensitivity

Develop CMSIS-Core compliant software for:

- Vector table
- Exception handler routines
- Enabling/disabling/pending exceptions and reading the exception state
- Working with operating system related exceptions such as SVC and SysTick
- Managing exception/interrupt prioritization
- Advanced exception handling features

Describe and handle different types of fault exceptions

Agenda

Introduction

Exception Model

- Exception Entry and Exit Behaviour
- Prioritization and Control
- Interrupt Sensitivity

Writing the Vector Table and Interrupt Handlers

Internal Exceptions and RTOS Support

Fault Exceptions

References

Overview

The exception architecture in Arm Microcontroller profile architecture versions is very different from other Arm architectures

Designed for microcontroller applications

- Built-in support for many interrupt sources
- Efficient handling of nested interrupts
- Flexible interrupt architecture (highly configurable)
- Built in RTOS support

Main Features

- Nested Vectored Interrupt Controller (NVIC)
- Micro-coded architecture handles the “dirty work”
  - No software overhead for Interrupt Entry/Exit and Interrupt Nesting
- Only one mode and one stack for all exception handling
  - Handler mode / Main stack
- Easy to program
  - Not necessary to write handler code in Assembler

Micro-coded interrupt mechanism

Interrupt architecture designed for low latency

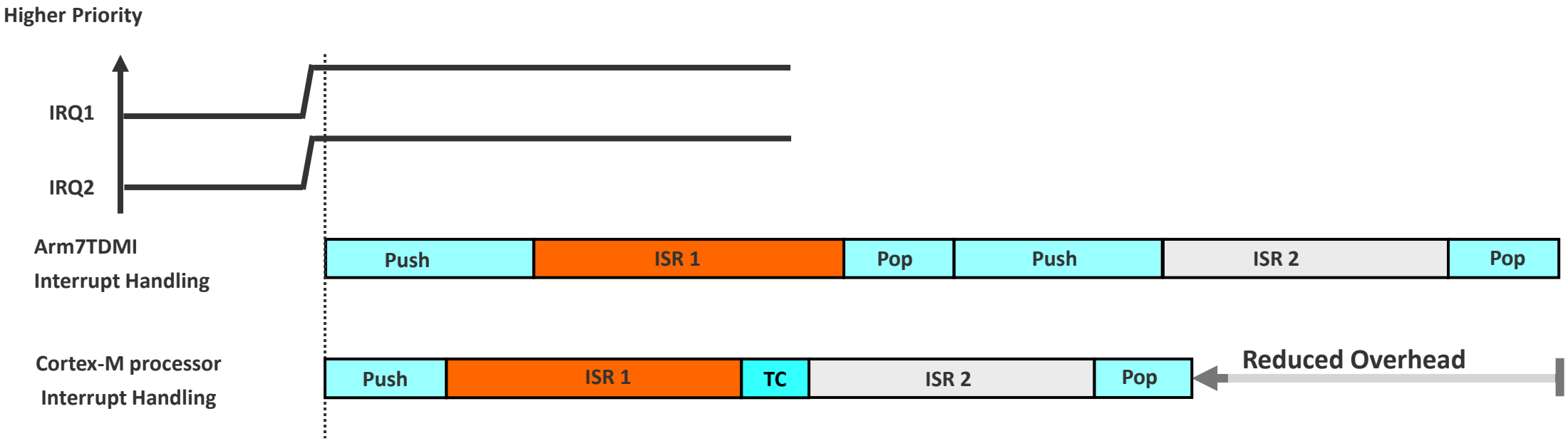
Interrupt prioritization mechanism is built into NVIC

Interrupt entry/exit is “micro-coded” – controlled by hardware

- Automatically saves and restores processor context
- Allows late determination of highest priority pending interrupt
- Allows another pending interrupt to be serviced without a full restore/save of processor state (tail-chaining)
- Interrupt Service Routines (ISRs) can be written entirely in C

Some multi-cycle instructions are interruptible for improved interrupt latency

Interrupt overheads



Security Extension – TrustZone for Armv8-M

This module introduces the underlying exception handling behavior of Armv8-M Mainline

- Similar to Armv7-M

Armv8-M architecture supports an optional security extension

- “Arm TrustZone for Armv8-M”

Inclusion of the Security extension modifies some aspects of exception handling

- Some exception handling resources are programmable to be Secure or Non-secure
- Some resources are banked (duplicated) to have both a Secure and a Non-secure version
- Additional memory-mapped registers are introduced in the System Control Space
- Additional bits populated in some control / status registers
- Additional registers may be included in context saving stack frames
  - Register values may be zeroed out when changing security state

Security extension features are described separately

1200 Armv8-M Mainline Security Extension

Agenda

Introduction

Exception Model

- Exception Entry and Exit Behaviour
- Prioritization and Control
- Interrupt Sensitivity

Writing the Vector Table and Interrupt Handlers

Internal Exceptions and RTOS Support

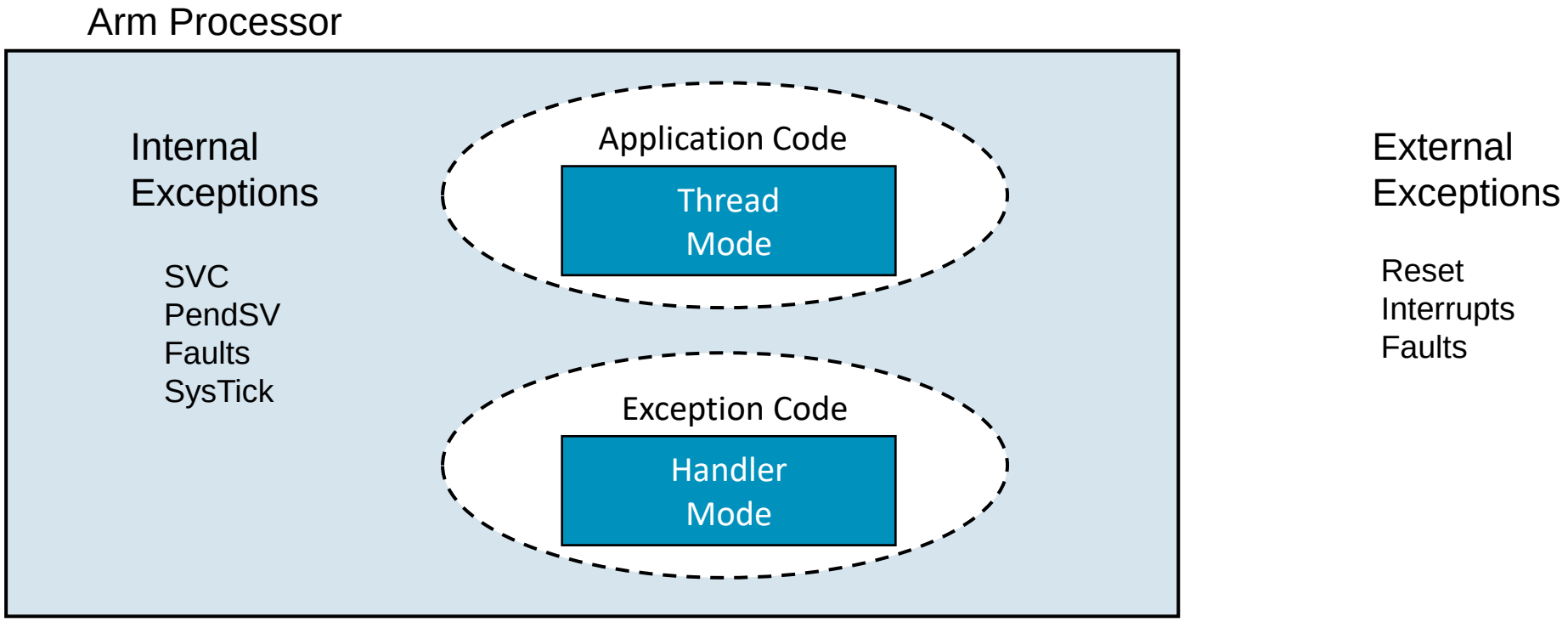
Fault Exceptions

References

Exception types

Exceptions can be caused by various events

- Internal
- External



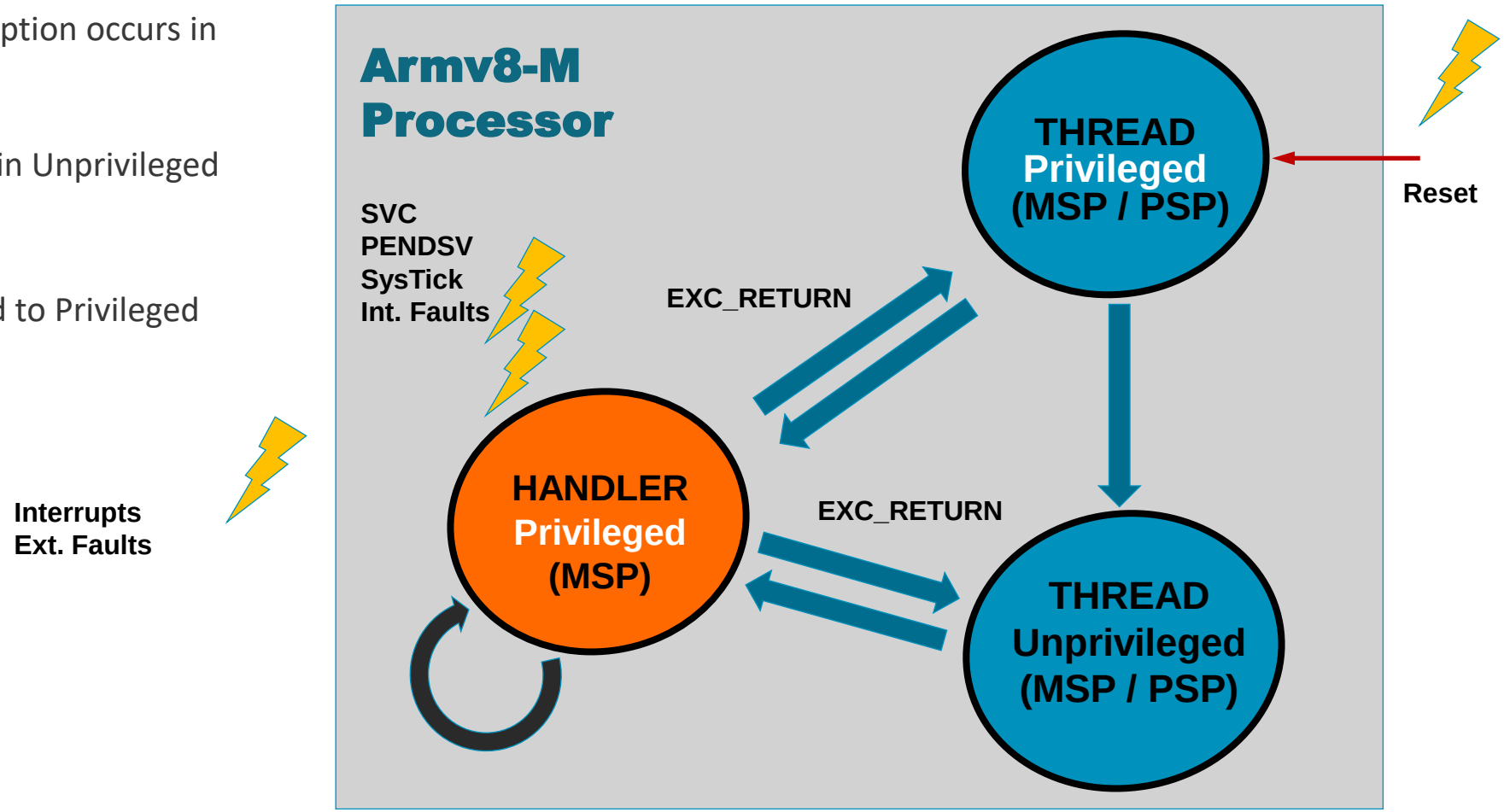
Processor mode usage example

Processor mode changes when an exception occurs in Thread mode

Mode does not change when an exception occurs in Handler mode

How do you switch from MSP to PSP in Unprivileged Thread mode?

How do you switch from Unprivileged to Privileged Thread mode?



External interrupts

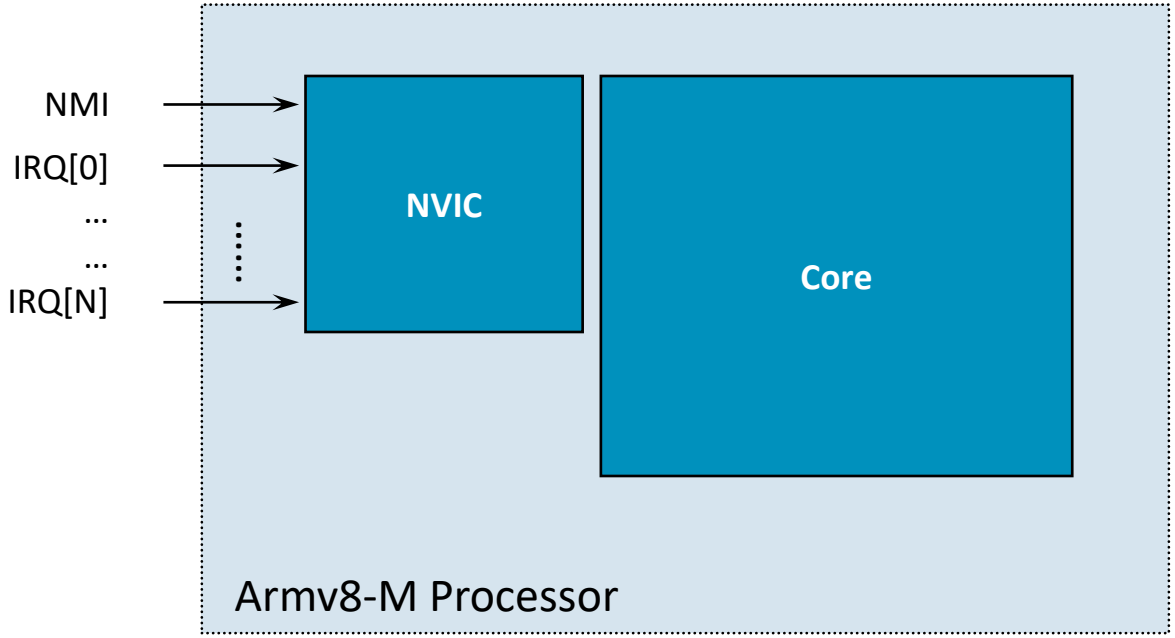
External Interrupts handled by Nested Vectored Interrupt Controller (NVIC)

- Tightly coupled with processor core

One Non-Maskable Interrupt (NMI) supported

Number of external interrupts is implementation-defined

- Armv8-M Mainline supports up to 496 interrupts



Pre-emption

Pre-emption occurs when a task is suspended in order to handle an exception

The currently running instruction stream is pre-empted

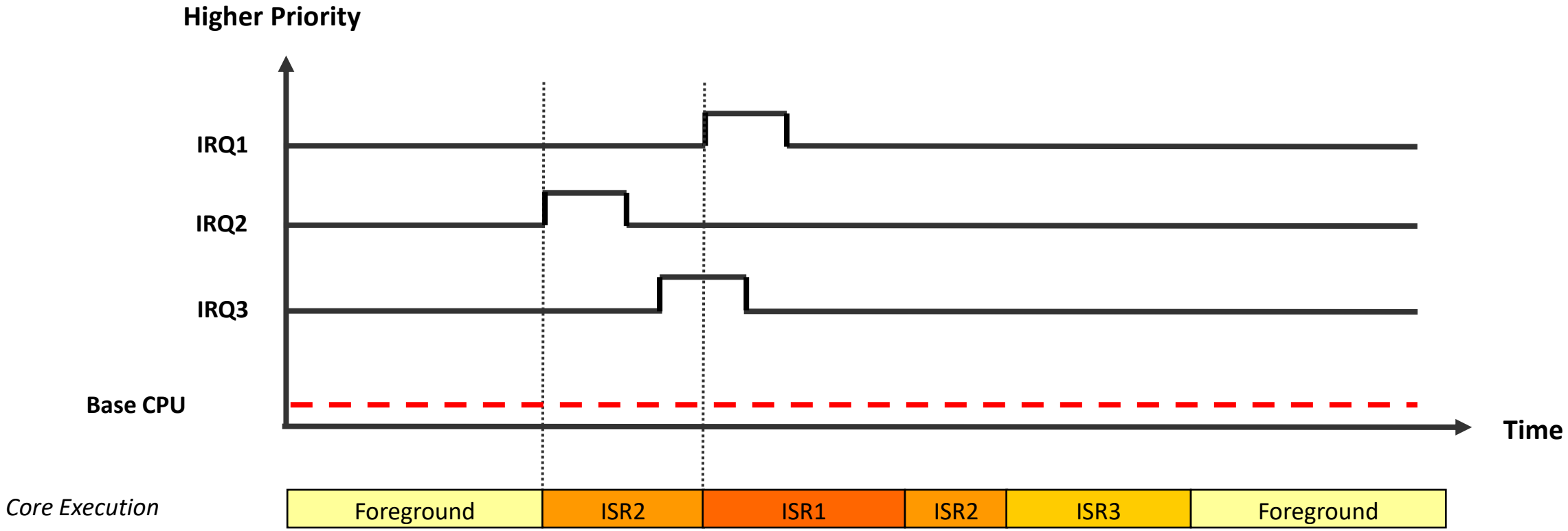
Pre-emption rules

- When multiple exceptions with the same priority are pending, they are handled in fixed order
  - Defined by the order of the vectors in the vector table
- Once an exception is active, only exceptions with a higher priority can pre-empt it

“Late arrival” is a little different

- In this case, a higher-priority exception takes the place of a lower one after the exception entry sequence has started
- More later...

Exception handling example



(ISR 2 resumes)

Exception model

Core begins execution with a “base level of execution priority”

- At reset all interrupts are disabled
- The base level of execution priority has a lower priority than the lowest programmable priority level, so any enabled interrupt will pre-empt the core

When an enabled interrupt is asserted

- Interrupt is taken (serviced by interrupt handler)
- Core runs the handler at the execution priority level of the interrupt
- When interrupt handler completes, original priority level is restored

When an enabled interrupt is asserted with lower or equal priority

- Interrupt is “pended” to run (when it has sufficient priority)

When a disabled interrupt is asserted

- Interrupt is “pended” to run (when it is enabled and has sufficient priority)

Interrupt nesting is always enabled

- To avoid nesting set all interrupts to the same priority

Exception properties

Each exception has associated properties

Exception Property	Description
Exception number	Identification for the exception
Vector address	Exception entry point in memory
Priority level	Determines the order in which multiple pending exceptions are handled

Vector table for Armv8-M Mainline

First entry contains initial Main SP

All other entries are addresses for exception handlers

- Must have LSB = 1 (for Thumb)

Table has up to 496 external interrupts

- Implementation-defined
- Maximum table size is 2048 bytes

Table may be relocated (next slide)

Each exception has an exception number

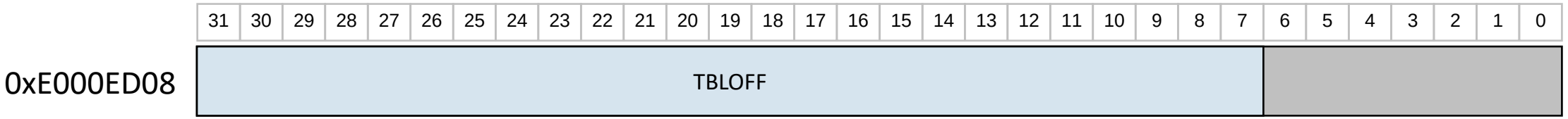
- Used in Interrupt Control and State Register to indicate the active or pending exception type

Table can be generated using C code

Address offset		Exception #
0x40 + 4*N	External interrupt N	16 + N
…	…	…
0x40	External interrupt 0	16
0x3C	SysTick	15
0x38	PendSV	14
0x34	Reserved	13
0x30	Debug Monitor	12
0x2C	SVC	11
0x20 to 0x28	Reserved (x3)	8-10
0x1C	SecureFault*	7
0x18	UsageFault	6
0x14	BusFault	5
0x10	MemManage	4
0x0C	HardFault	3
0x08	NMI	2
0x04	Reset	1
0x00	SP_main	N/A

* Reserved if Security Extension not implemented

Vector Table Offset Register (VTOR)



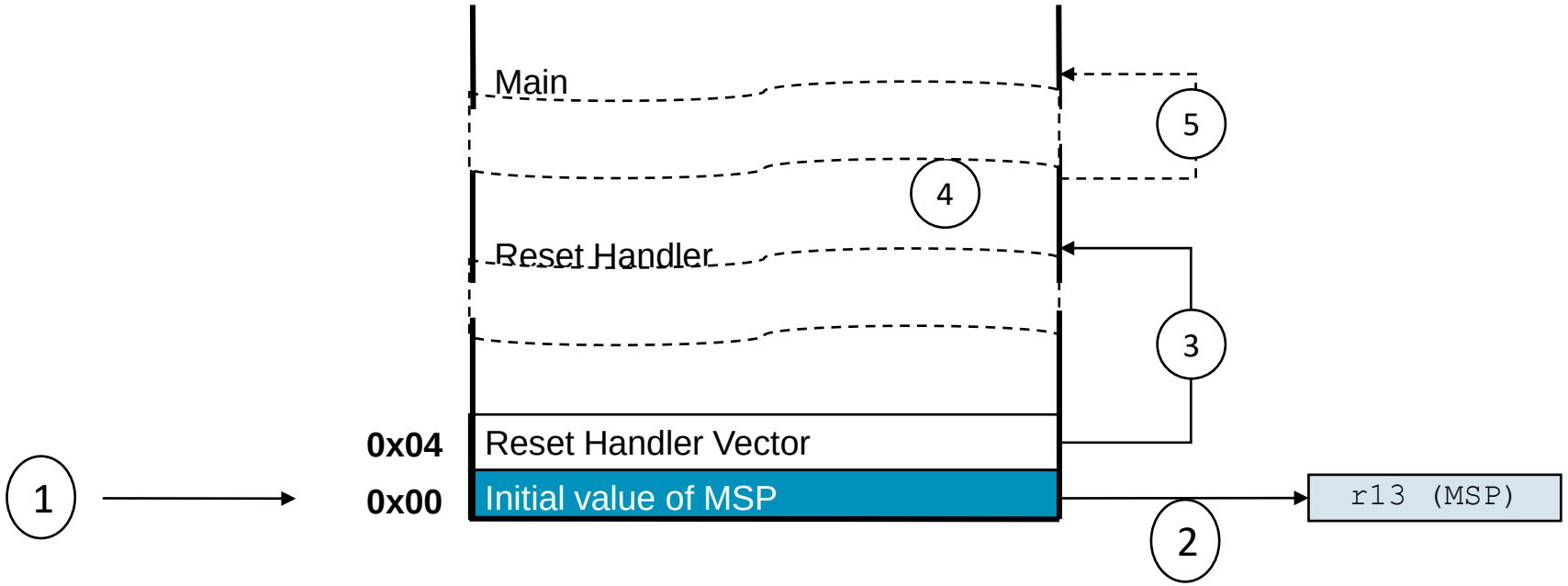
VTOR specifies the base address of the vector table

- Offset from address 0x0

At reset the `TBLOFF` field is fixed to 0x00000000 unless an implementation includes configuration input signals that determine the reset value

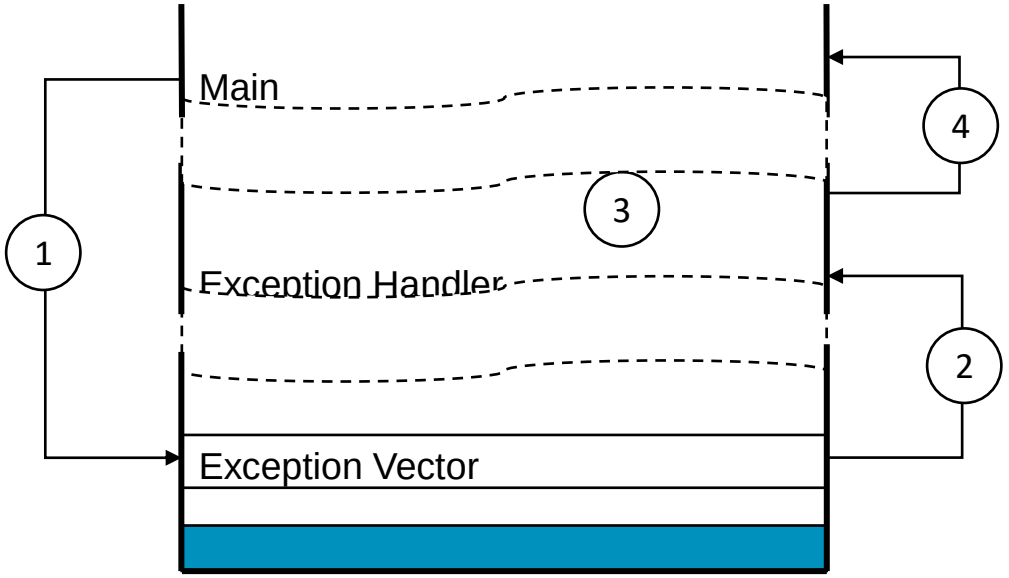
- An implementation might further restrict alignment of the VTOR
- Vector Table alignment should match or exceed table size for configured number of interrupts

Reset behavior



- A reset exception occurs (Reset input was asserted)
- Load MSP (Main Stack Pointer) register initial value from address in VTOR
- Load reset handler vector address from address VTOR + 0x04
- Reset handler executes in Thread Mode
- Optional: Reset handler branches to the main program

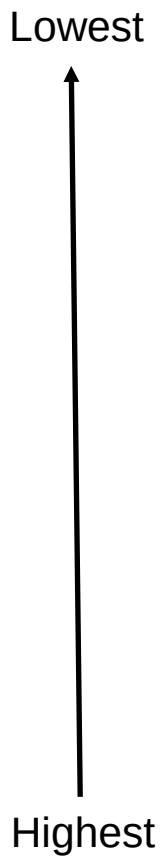
Exception behavior



1. **Exception occurs**
   - Current instruction stream stops
   - Processor accesses vector table
2. **Vector address for the exception loaded from the vector table**
3. **Exception handler executes in Handler Mode**
4. **Exception handler returns to main (assuming no nesting – more later)**

Exception priorities overview

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



The lower the priority number, the higher the priority level

Priority level is stored left-justified in a byte-wide register, which is set to 0x0 at reset

- Armv8-M Mainline supports 3-8 bits of priority physically implemented

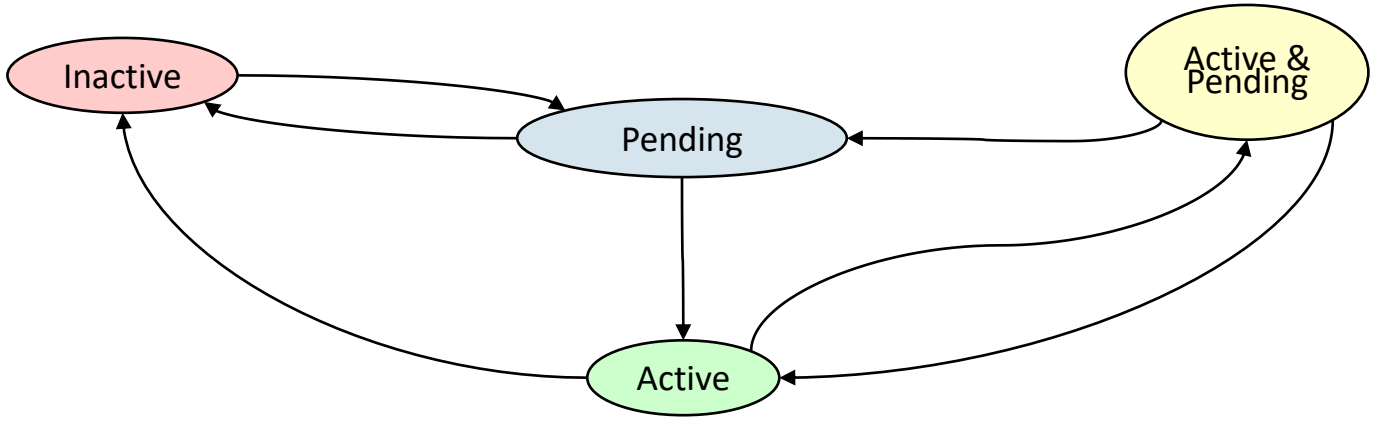
Exceptions with the same priority level follow a fixed priority order using the exception number

- Lower exception number has higher priority level

Exception states

Exceptions can be in the following states

- **Inactive:** Not Pending or Active
- **Pending:** Exception event has been generated but processing has not started yet
- **Active:** Exception processing has been started but is not complete yet
  - An exception handler can interrupt the execution of another exception handler. In this case both exceptions are in the active state.
- **Active and Pending:** The exception is being serviced by the processor and there is a pending exception from the same source



Agenda

Introduction

Exception Model

- **Exception Entry and Exit Behaviour**
- Prioritization and Control
- Interrupt Sensitivity

Writing the Vector Table and Interrupt Handlers

Internal Exceptions and RTOS Support

Fault Exceptions

References

Exception entry behavior (1)

When receiving an exception the processor will finish the current instruction for most instructions

- To minimize interrupt latency, the processor can take an exception during the execution of a multi-cycle instruction - see next slide

The processor state is automatically saved to the current stack

- 8 registers are pushed: R0-R3, R12, LR, ReturnAddress, RETPSR
- Follows Arm Architecture Procedure Call Standard (AAPCS)

During (or after) state saving, the address of the ISR is read from the Vector Table

The Link Register is modified for interrupt return

The first instruction of the exception handler routine executed

- Interrupt late-arrival and interrupt tail-chaining can improve IRQ latency

The exception handler routine executes in Handler mode using the Main stack

Exception entry behavior (2)

On taking an exception during a multi-cycle instruction the processor either:

- Continues execution of the instruction after returning from the exception
- Abandons the instruction and restarts the instruction on returning from the exception
- Completes the instruction, affecting interrupt latency

For multi-cycle instructions:

- Divide and floating point square root instructions are abandoned and restarted
- Load/store multiple instructions are treated as “exception-continuable instructions”, if possible
  - The ICI bits in the EPSR are used to hold the continuation state
- However, in some situations `LDM`/`STM` instructions are abandoned and restarted
  - For example, An `LDM`/`STM` instruction within an If-Then block

Architecturally software must not use an LDM/STM to access a volatile location if

- it executes inside an IT block
- it loads the same register as the base register
- it loads the PC

Stacking on exception entry

Foreground stack (Main or Process) used to store the pre-interrupt state

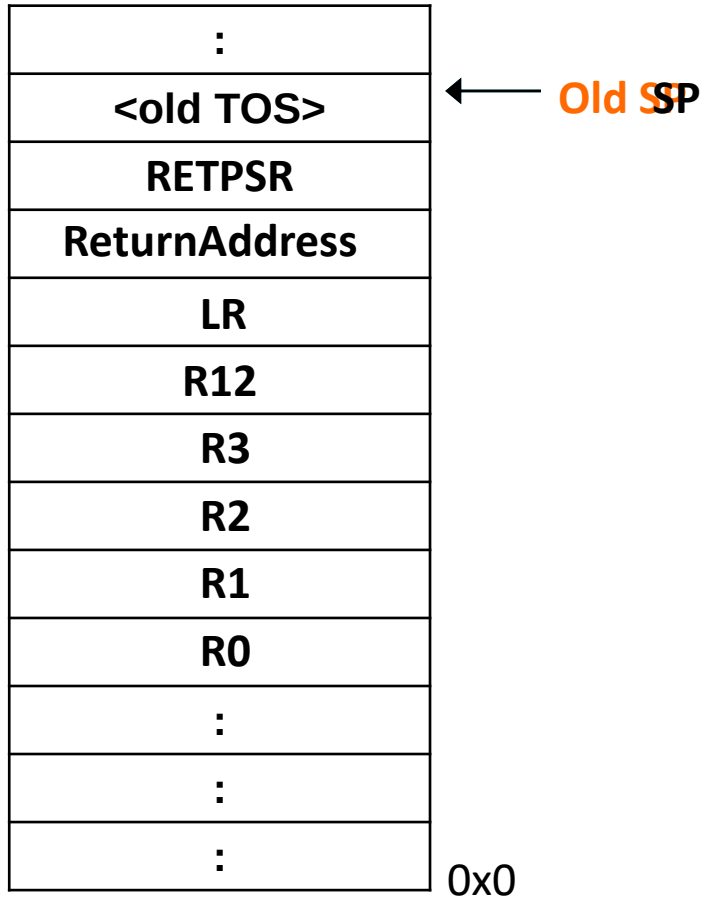
Stack is Full-Descending

- SP automatically decremented on push
- SP always points to non-empty value
- Same behavior as other Arm cores

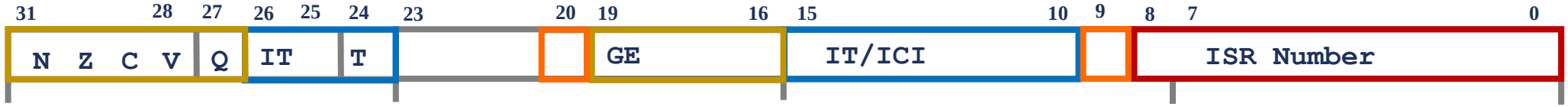
The stack frame must be 8-byte aligned

- AAPCS compliant
- Extra word padded above RETPSR if required

Main stack active after state is saved



RETPSR



Combined Exception Return Program Status Registers

- ALU flags [31:27], [19:16]
- T bit [24]
- SFPA [20] – Secure Floating Point Active
- IT / ICI [26:24], [15:10]
- SPREALIGN [9] – Stack Pointer Re-align
- Exception Number [8:0]

ReturnAddress values

The value loaded into the ReturnAddress depends on the pre-empting exception

	Value of ReturnAddress
BusFault (precise), HardFault (precise)	Instruction causing the fault (Usually fatal error so return is unlikely)
BusFault (imprecise), HardFault (imprecise)	Next instruction to be executed (Usually fatal error so return is unlikely)
MemManage, UsageFault	Instruction causing the fault (Instruction may be re-executed after the fault cause is fixed)
IRQ, NMI, PendSV, SVCall, SysTick, DebugMonitor	Next instruction to be executed (Allows return after handling the exception)

The least significant bit of ReturnAddress is set (T-bit)

Returning from an exception

The processor returns from an exception with the following instructions when the PC is loaded with “magic” value of 0xFFFF_FFXX (EXC_RETURN)

- BX LR
- LDR/LDM/POP which includes loading the PC

If no interrupts are pending, foreground state is restored

- Stack and mode specified by EXC_RETURN is used

If other interrupts are pending, the highest priority may be serviced

- Serviced if interrupt priority is higher than the foreground’s base priority
- Process is called tail-chaining as foreground state is not yet restored

If state restore is interrupted, it is abandoned

- New ISR executed without state saving (original state still intact and valid)
- Must still fetch new vector and refill pipeline

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

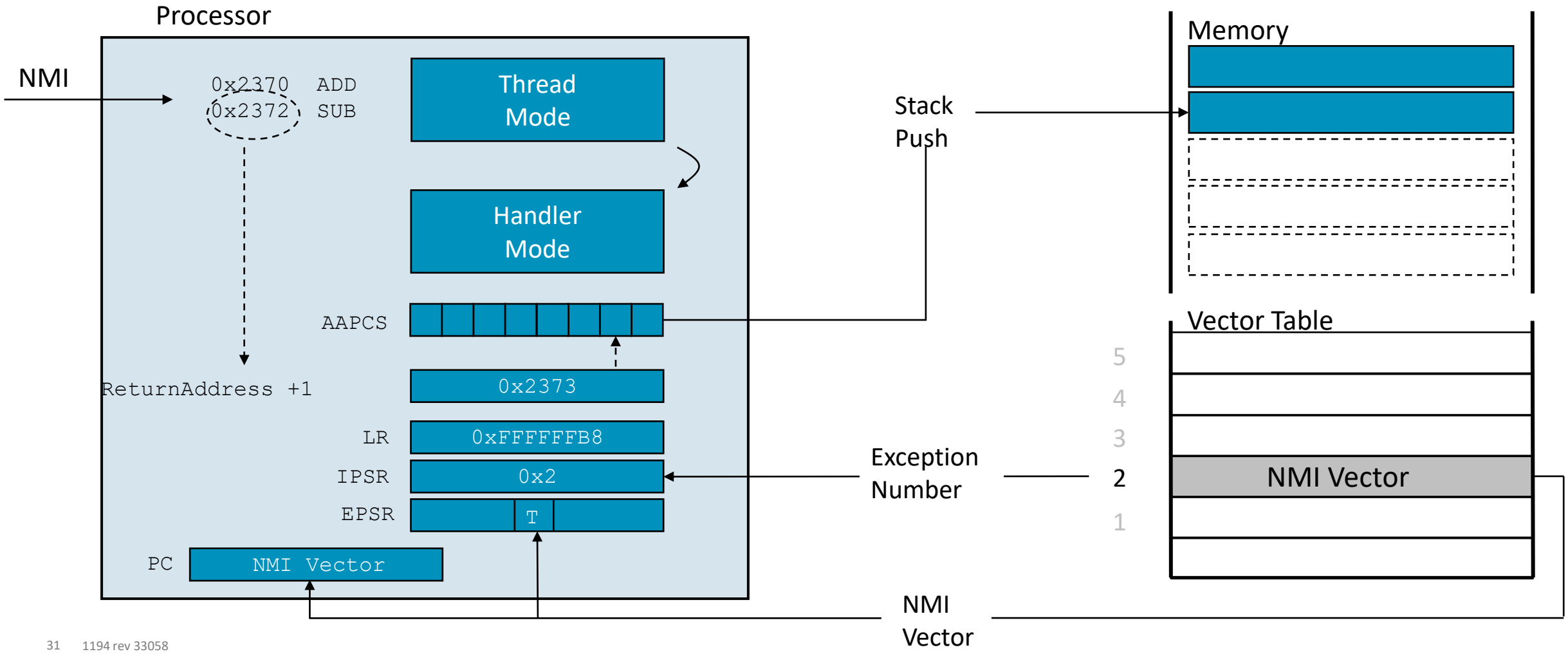
Bits	31:28	27:7	6:5	4	3	2	1	0
Description	EXC_RETURN indicator	Reserved	Security stack status	FPU Frame Type	Return mode	Return stack	Reserved	Exception Secure
Value	0xF	0x1FFFFF	01 (NS) else (Secure)	1 (Basic) 0 (Extended)	1 (Thread) 0 (Handler)	1 (process stack) 0 (main stack)	0	1 (Secure) 0 (NS)

For implementations without Security Extension:

EXC_RETURN	**Condition**
0xFFFFFFA0	Return to Handler Mode and restore extended FP stack frame
0xFFFFFFA8	Return to Thread mode using the main stack and restore extended FP stack frame
0xFFFFFFAC	Return to Thread mode using the process stack and restore extended FP stack frame
0xFFFFFFB0	Return to Handler Mode and restore basic stack frame
0xFFFFFFB8	Return to Thread mode using the main stack and restore basic stack frame
0xFFFFFFBC	Return to Thread mode using the process stack and restore basic stack frame

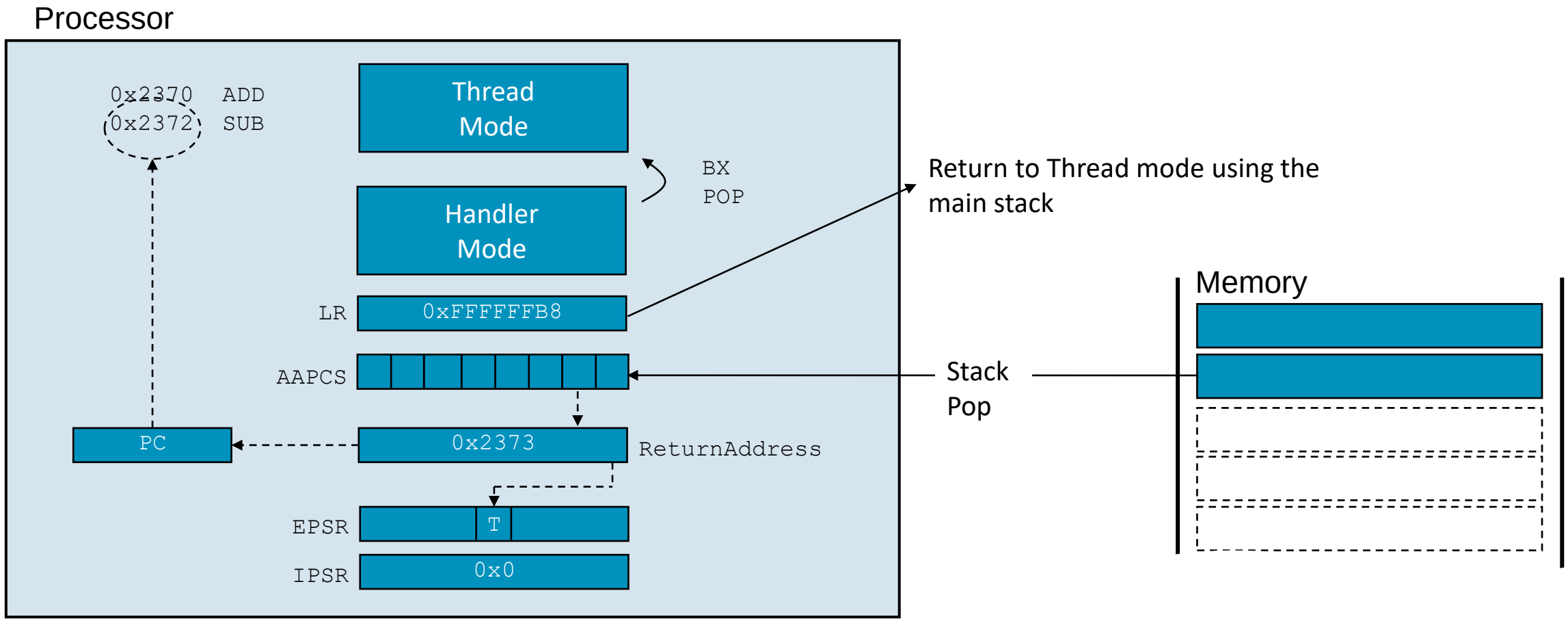
NMI exception entry example

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

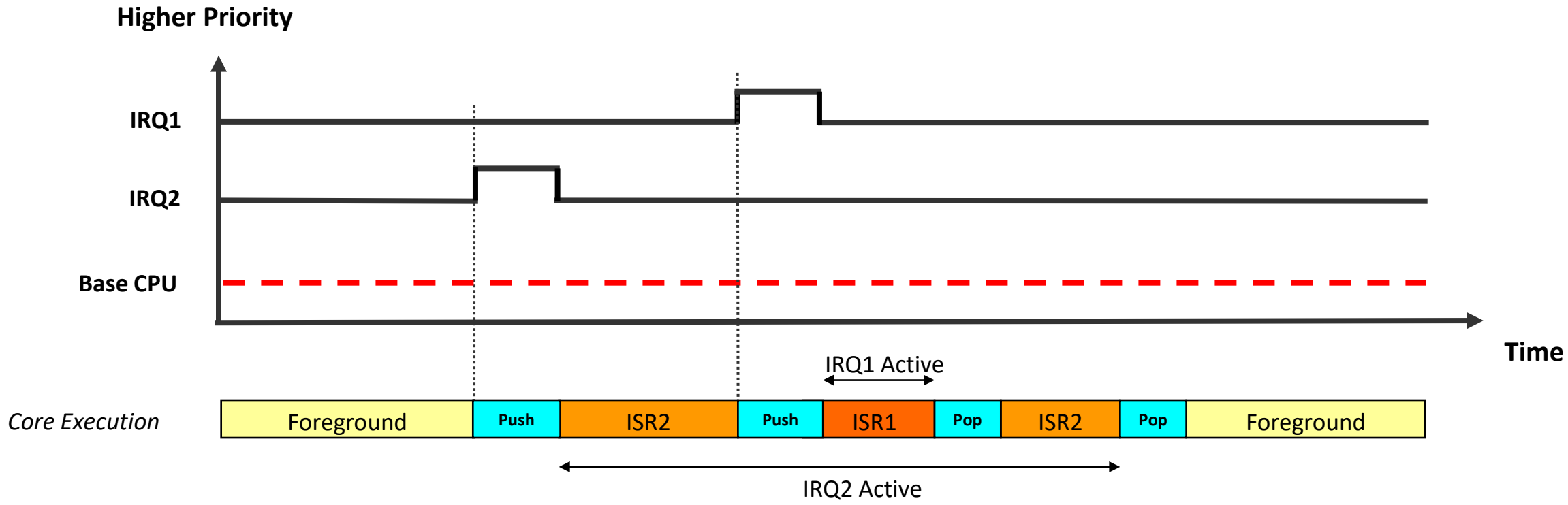


NMI exception return example

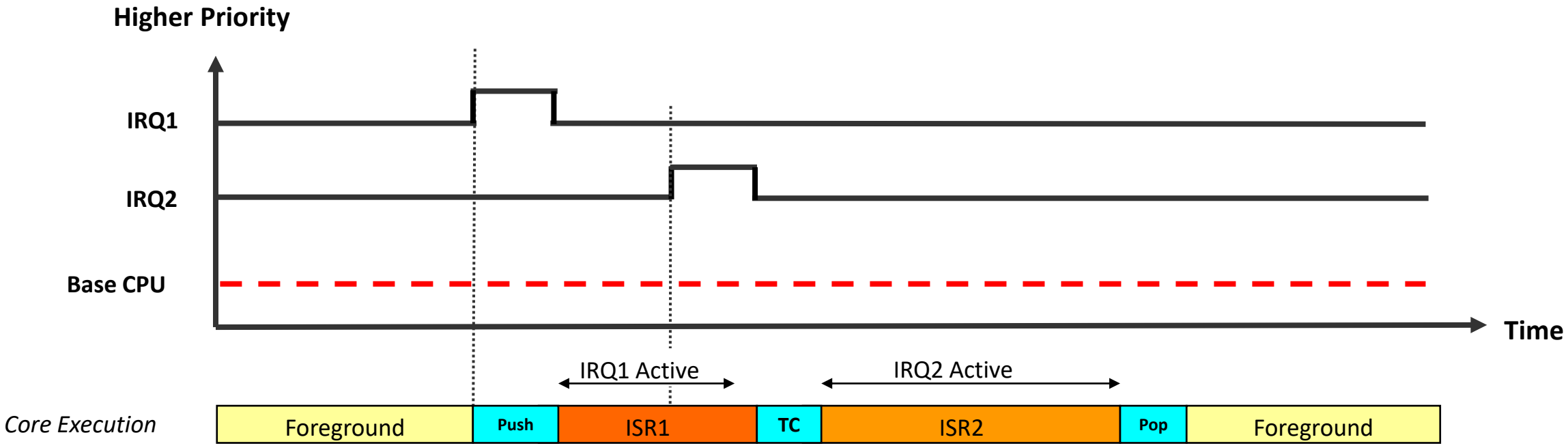
BX or POP instruction can be used for exception return



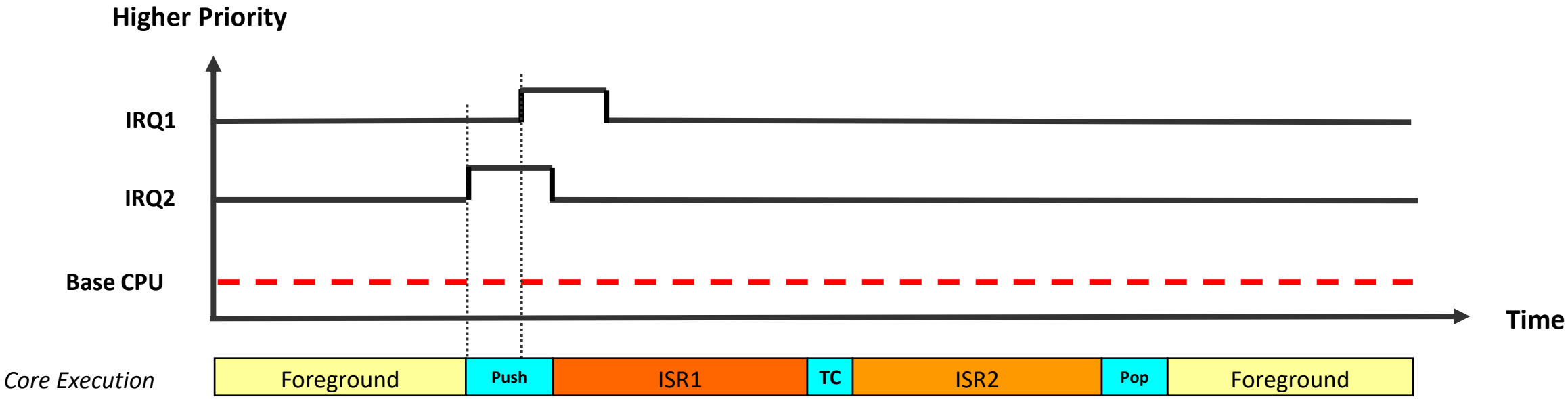
Nesting example



Tail-chaining example



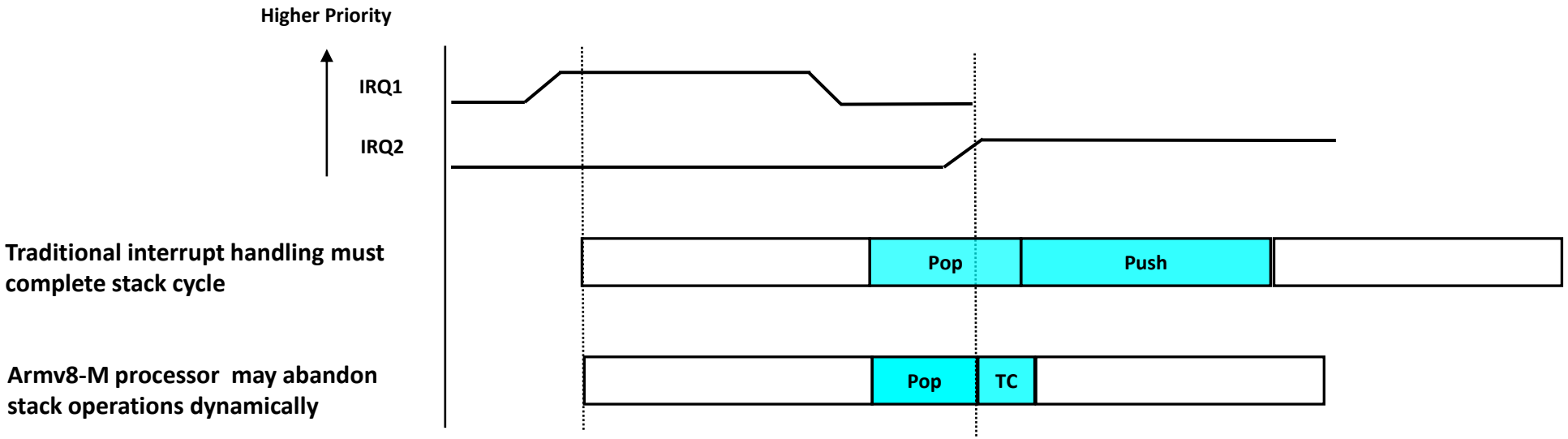
Late-arriving example



A pending higher priority exception is handled before an already pending lower priority exception even after exception entry sequence has started

The lower priority exception is handled after the higher priority exception

Exceptions during state restore



ARM7TDMI

- Load Multiple uninterruptible, and hence the core must complete the POP and then full stack PUSH

Armv8-M processor

- POP may be abandoned early if another interrupt arrives
- If POP is interrupted, the new handler can be fetched directly

Agenda

Introduction

Exception Model

- Exception Entry and Exit Behaviour
- **Prioritization and Control**
- Interrupt Sensitivity

Writing the Vector Table and Interrupt Handlers

Internal Exceptions and RTOS Support

Fault Exceptions

References

Execution priority and priority boosting

An Armv8-M Mainline processor provides three special-purpose mask registers for execution priority boosting

BASEPRI (the base priority mask) sets the execution priority to a value between 1 and 255

- Acts as a general interrupt mask
- Example
  - BASEPRI = 16 → only interrupts with priority levels (-4 to 15) can pre-empt
- If BASEPRI = 0 masking is disabled (any enabled interrupt pre-empts Thread mode)

PRIMASK (the exception mask register) can be used to raise the execution priority level to 0

- Only allows HardFault and NMI to pre-empt

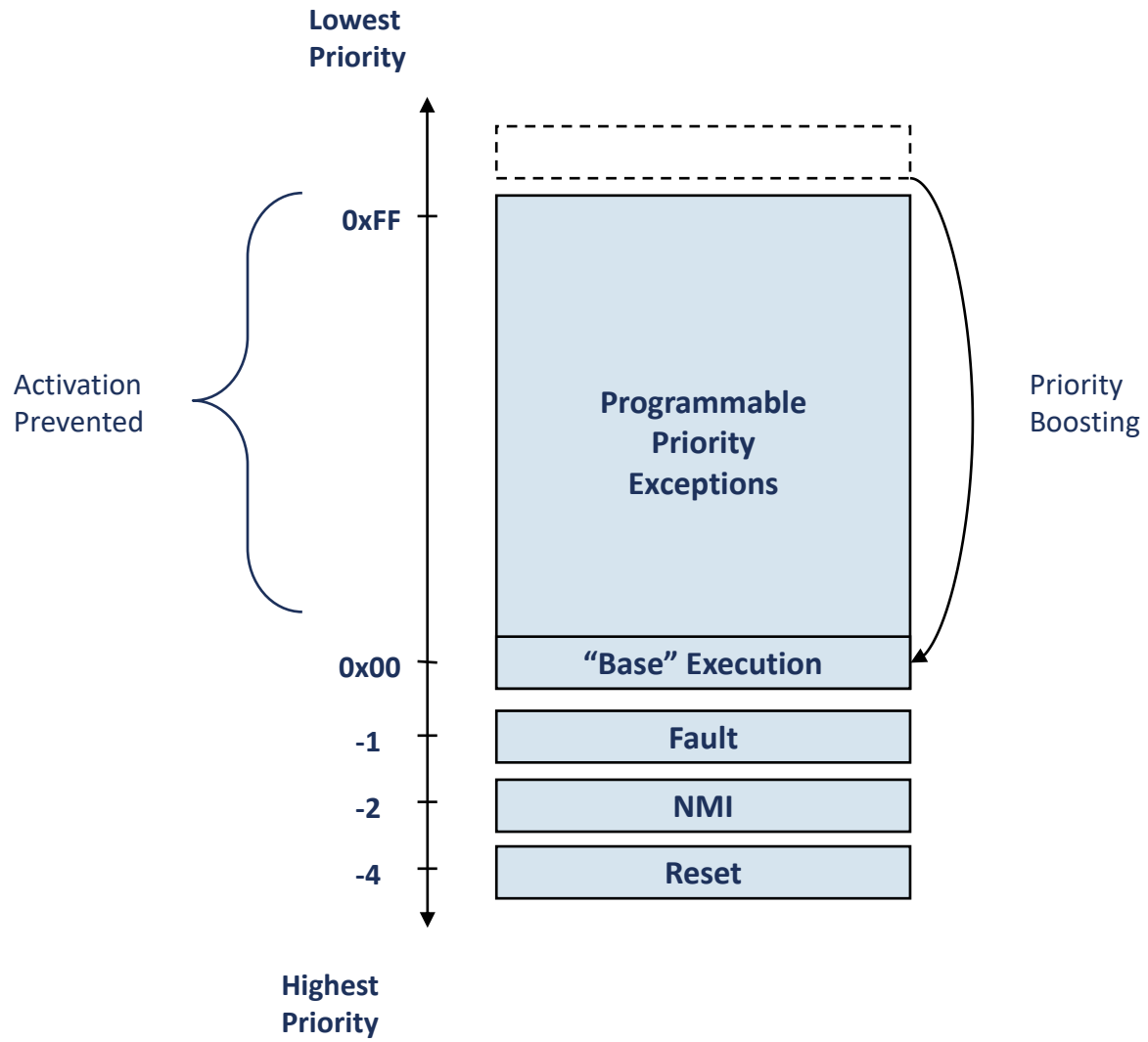
FAULTMASK (the fault mask register) can be used to raise the execution priority level to -1

- Can only be set by an exception handler (automatically clears on exception return)
- Only allows NMI to pre-empt (and secure HardFault if its priority is -3)

Priority boosting example - PRIMASK

PRIMASK can be used to disable interrupts by raising the base level of execution priority level to 0

All programmable exceptions are prevented from preempting the current program



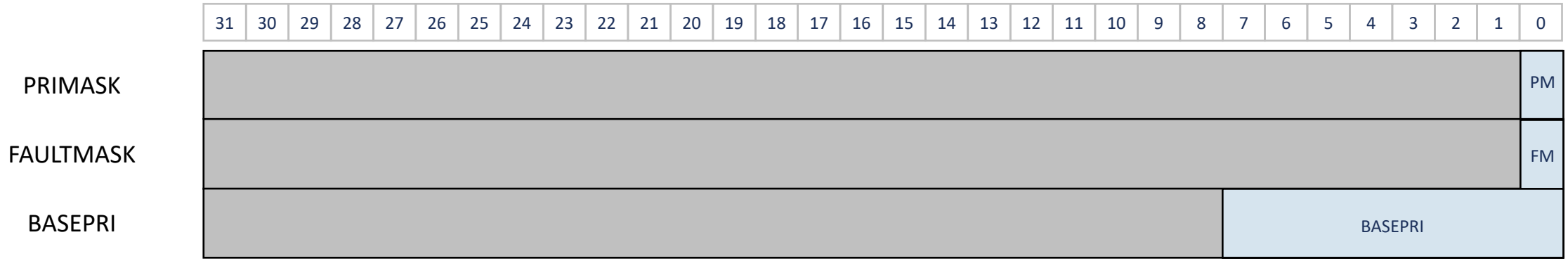
Special-purpose mask registers

A reset clears all the mask registers to zero

The processor ignores unprivileged writes to the mask registers

Software can access these registers using the `MRS` and `MSR` instructions

- PRIMASK and FAULTMASK may also be accessed using the **`CPS`** instruction
- The **`MSR`** instruction accepts a register masking argument, BASEPRI_MAX
  - Updates BASEPRI only if BASEPRI masking is disabled, or the new value increases the BASEPRI priority level



Priority boosting instructions

BASEPRI is accessed like a Program Status Register

```
MSR{cond} BASEPRI, Rm   ; write BASEPRI
MRS{cond} Rd, BASEPRI   ; read BASEPRI
```

PRIMASK and FAULTMASK map to the classic “I” and “F” flags

```
CPSID I  ; set PRIMASK (disable interrupts)
CPSIE I  ; clear PRIMASK (enable interrupts)
CPSID F  ; set FAULTMASK (disable faults and interrupts)
CPSIE F  ; clear FAULTMASK (enable faults and interrupts)
```

CMSIS Functions:

- PRIMASK: `__enable_irq()`, `__disable_irq()`
- FAULTMASK: `__enable_fiq()`, `__disable_fiq()`
- BASEPRI: `__get_BASEPRI()`, `__set_BASEPRI(uint32_t value)`

Must be in privileged mode to modify execution priority level

Armv8-M Mainline priority grouping

The 8-bit Priority Level is divided into 2 fields

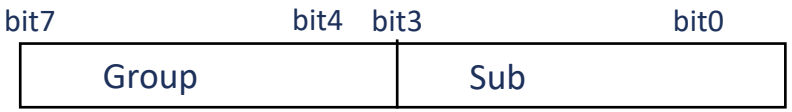
- Group Priority and Sub-Priority [GROUP.SUB]

Group Priority is the pre-empting priority level

Sub-Priority is used only if the Group Priority is the same

Exception Number is final tie-breaker

- If two interrupts have identical priority levels, lowest vector number takes priority



Example

- 4-bit Group Priority / 4-bit Sub-Priority
- SysTick is enabled with priority = 0x30
- Timer1 is enabled with priority = 0x30
- Timer2 is enabled with priority = 0x31
  - If Timer1 & Timer2 assert simultaneously → Timer1 serviced, Timer2 pended
  - If Timer2 is being serviced and Timer1 asserts → Timer1 pended
  - If SysTick & Timer1 assert simultaneously → SysTick serviced, Timer1 pended

Group priority / sub-priority selection

PRIGROUP	Binary Point (group.sub)		Group Priority (Pre-empting)		Sub-Priority	
			Bits	Levels	Bits	Levels
000	7.1	gggggggs	7	128	1	2
001	6.2	gggggggss	6	64	2	4
010	5.3	ggggggsss	5	32	3	8
011	4.4	ggggssss	4	16	4	16
100	3.5	gggsssss	3	8	5	32
101	2.6	ggssssss	2	4	6	64
110	1.7	gsssssss	1	2	7	128
111	0.8	ssssssss	0	0	8	256

Selection made by PRIGROUP in Application Interrupt and Reset Control Register

System-wide configuration applies to all interrupt priorities

If fewer than 8 priority bits implemented, fewer Sub-Priority bits are available

- Example: 6 priority bits and PRIGROUP = 001 → 6 Group Bits / 0 Sub-Priority Bits

Interrupt control and status bits

ENABLE bit (R/W) – is interrupt enabled or disabled?

PENDING bit (R/W) – is interrupt waiting to be serviced?

- Interrupt is pending if priority is too low or if interrupt is disabled
- Can set bit to generate an interrupt (if enabled and sufficient priority)
- Can clear bit to clear a pending interrupt (interrupt will not be serviced)

ACTIVE bit (R only) – is interrupt active or inactive?

- If interrupts are nested, more than one active bit will be set

PRIORITY bits (R/W) – set the interrupt priority

- Lower value has higher priority (priority 0 is highest)
- Armv8-M Mainline: 3 - 8 bits (8 - 256 levels)
  - May be divided into a group (pre-empt) priority and a sub-group priority (user defined)

Interrupt Enable Registers

SETENAx Registers – Write “1” to set the interrupt enable bit

	Bit 31				Bit 0
0xE000E100	SETE31	…	SETE2	SETE1	SETE0
0xE000E104	SETE63	…	SETE34	SETE33	SETE32

CLRENAx Registers – Write “1” to clear the interrupt enable bit

	Bit 31				Bit 0
0xE000E180	CLRE31	…	CLRE2	CLRE1	CLRE0
0xE000E184	CLRE63	…	CLRE34	CLRE33	CLRE32

Reading SETENAx or CLRENAx

- If interrupt enabled, bit position will return “1”
- If interrupt disabled, bit position will return “0”

Interrupt Pending Registers

SETPENDx Registers – Write “1” to set the pending bit

	Bit 31				Bit 0
0xE000E200	SETP31	…	SETP2	SETP1	SETP0
0xE000E204	SETP63	…	SETP34	SETP33	SETP32

CLRPENDx Registers – Write “1” to clear the pending bit

	Bit 31				Bit 0
0xE000E280	CLRP31	…	CLRP2	CLRP1	CLRP0
0xE000E284	CLRP63	…	CLRP34	CLRP33	CLRP32

Reading SETPENDx or CLRPENDx returns the pending status

- Hardware automatically clears pending bit when interrupt serviced
- Only the minimum number of bits/registers is implemented

Interrupt Active Registers

ACTIVEx Registers - show which interrupts are active (read only)

	Bit 31				Bit 0
0xE000E300	ACT31	…	ACT2	ACT1	ACT0
0xE000E304	ACT63	…	ACT34	ACT33	ACT32
⋮	⋮				

When interrupt handler starts execution, ACTx is set

When interrupt handler returns, ACTx is cleared

May have more than 1 active bit set if interrupts are nested

- Example
  - INT0 is running and is pre-empted by INT4
  - ACTIVE0 = 0x00000011

Only the minimum number of bits/registers is implemented

Interrupt Priority Registers

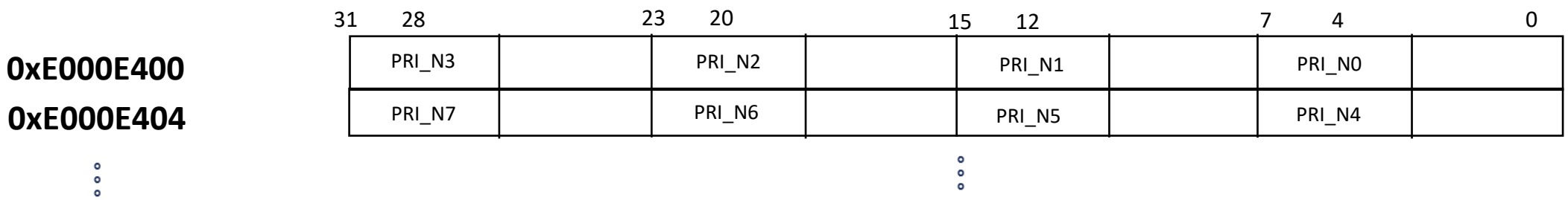
Each priority register holds the priority for 4 interrupts (1 byte / interrupt)

Priority is always LEFT justified in the byte

- Provides best compatibility between cores and within core implementations

Armv8-M Mainline: 3-8 priority bits

- 4-bit example shown below



Reading the unused (reserved) space returns 0 and writes are ignored

Only the minimum number of registers is implemented

Agenda

Introduction

Exception Model

- Exception Entry and Exit Behaviour
- Prioritization and Control
- **Interrupt Sensitivity**

Writing the Vector Table and Interrupt Handlers

Internal Exceptions and RTOS Support

Fault Exceptions

References

Interrupt sensitivity

Interrupts are active high inputs

Two sensitivity modes are supported for interrupts

- Pulse-sensitive Interrupts
- Level-sensitive Interrupts

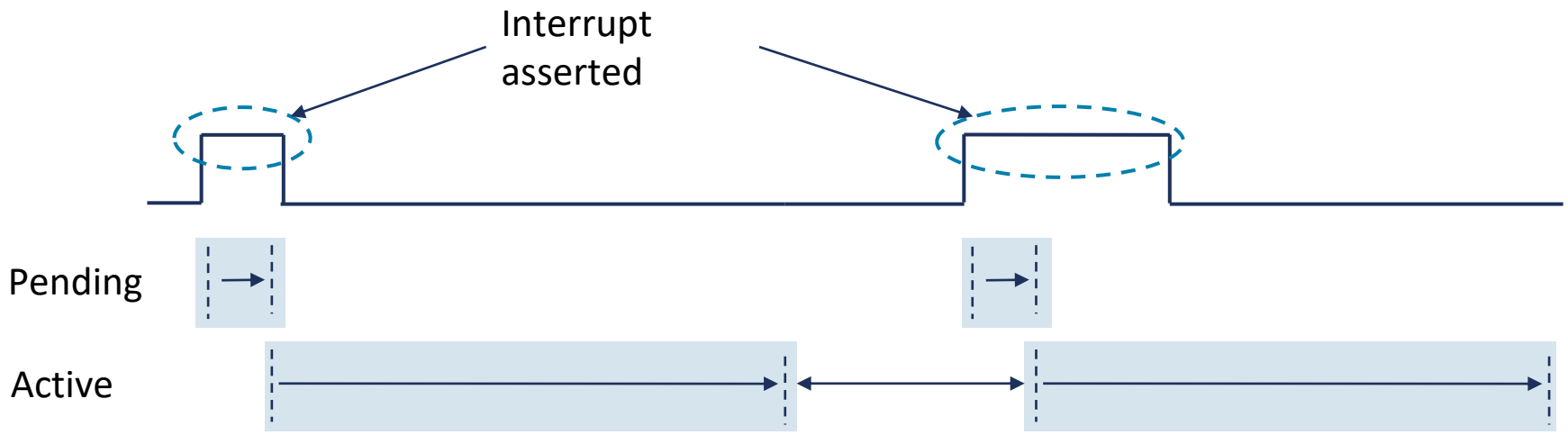
Interrupt Service Routine (ISR) requirements are dependent on sensitivity configuration

Pulse-sensitive Interrupts – one pulse

Pulse can be asserted for one, or more clock cycles

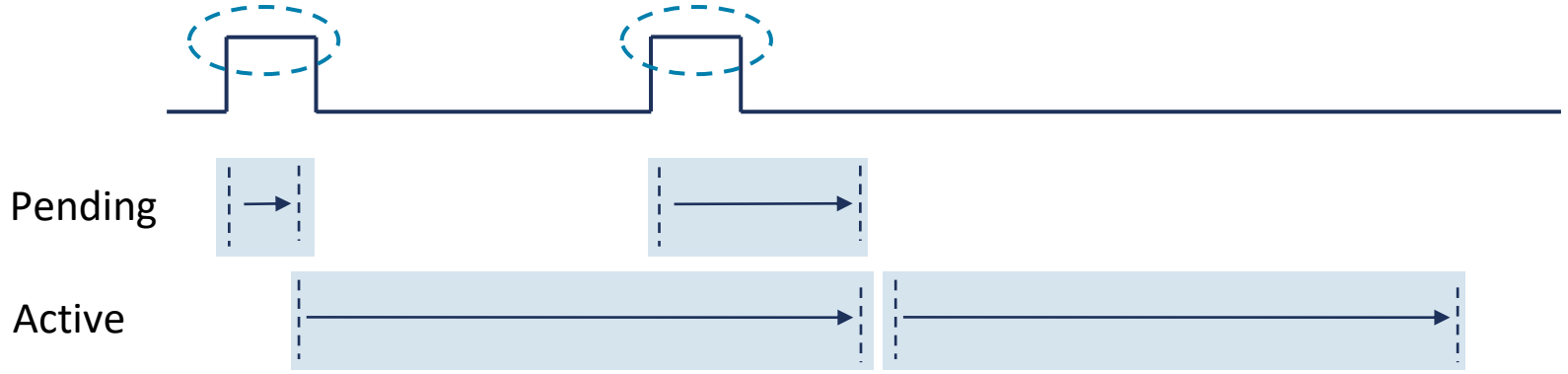
Interrupt is cleared directly by the source

- No software overhead for the ISR

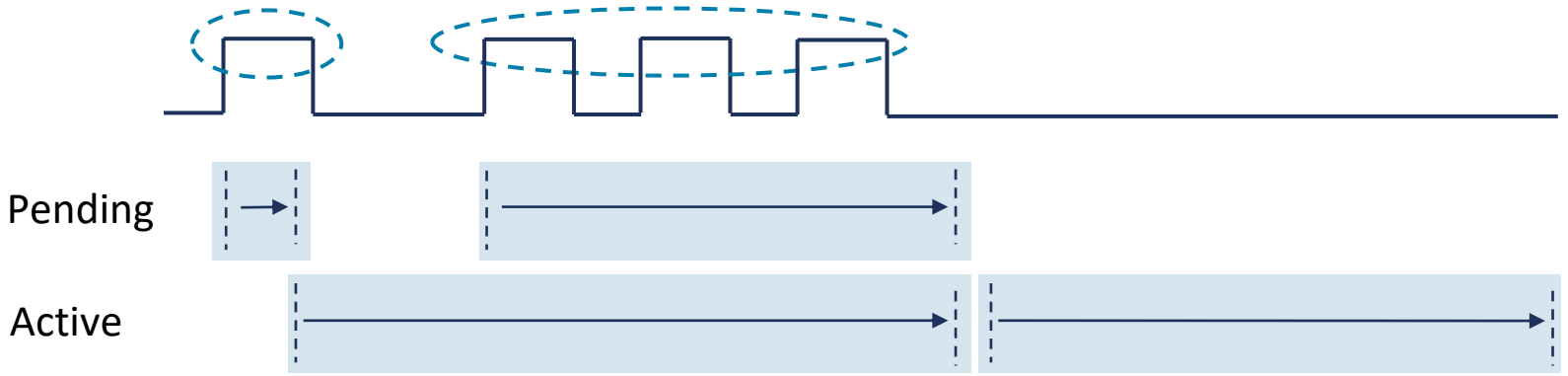


Pulse interrupts – multiple pulses

A second pulse-sensitive interrupt can be pended while the first is still active



Multiple pulse-sensitive interrupts asserted while the first interrupt is still active

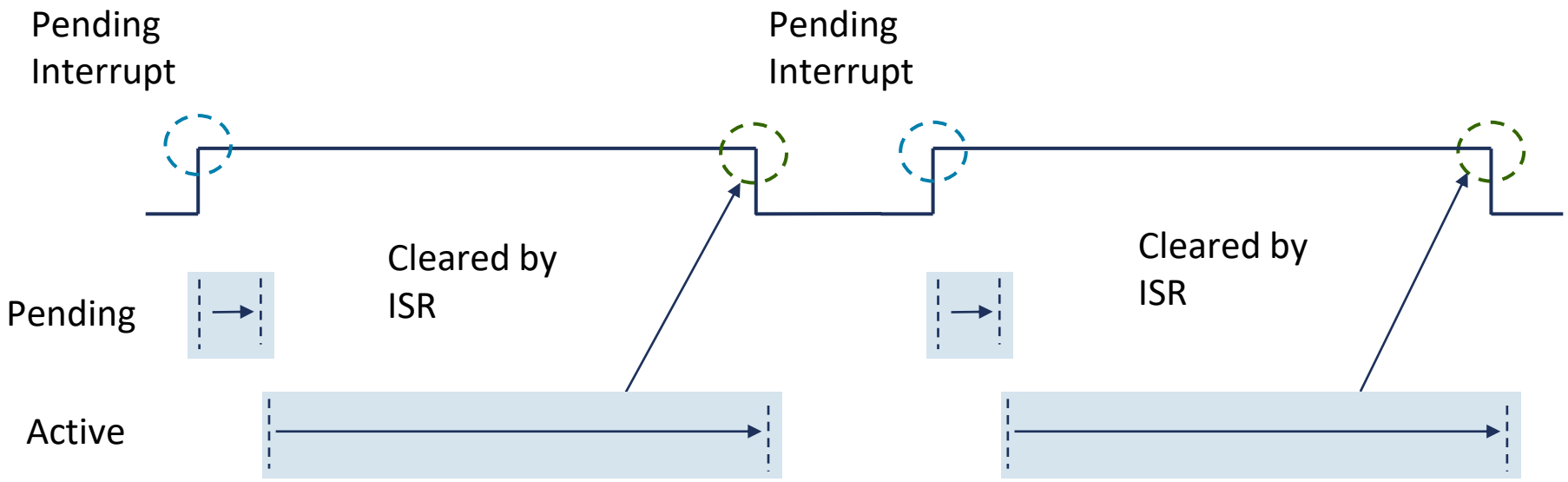


Level-sensitive interrupts

A level-sensitive interrupt is asserted until cleared by the ISR

- Usually done near the end of the ISR
  - Nested interrupts for the same interrupt source are not supported

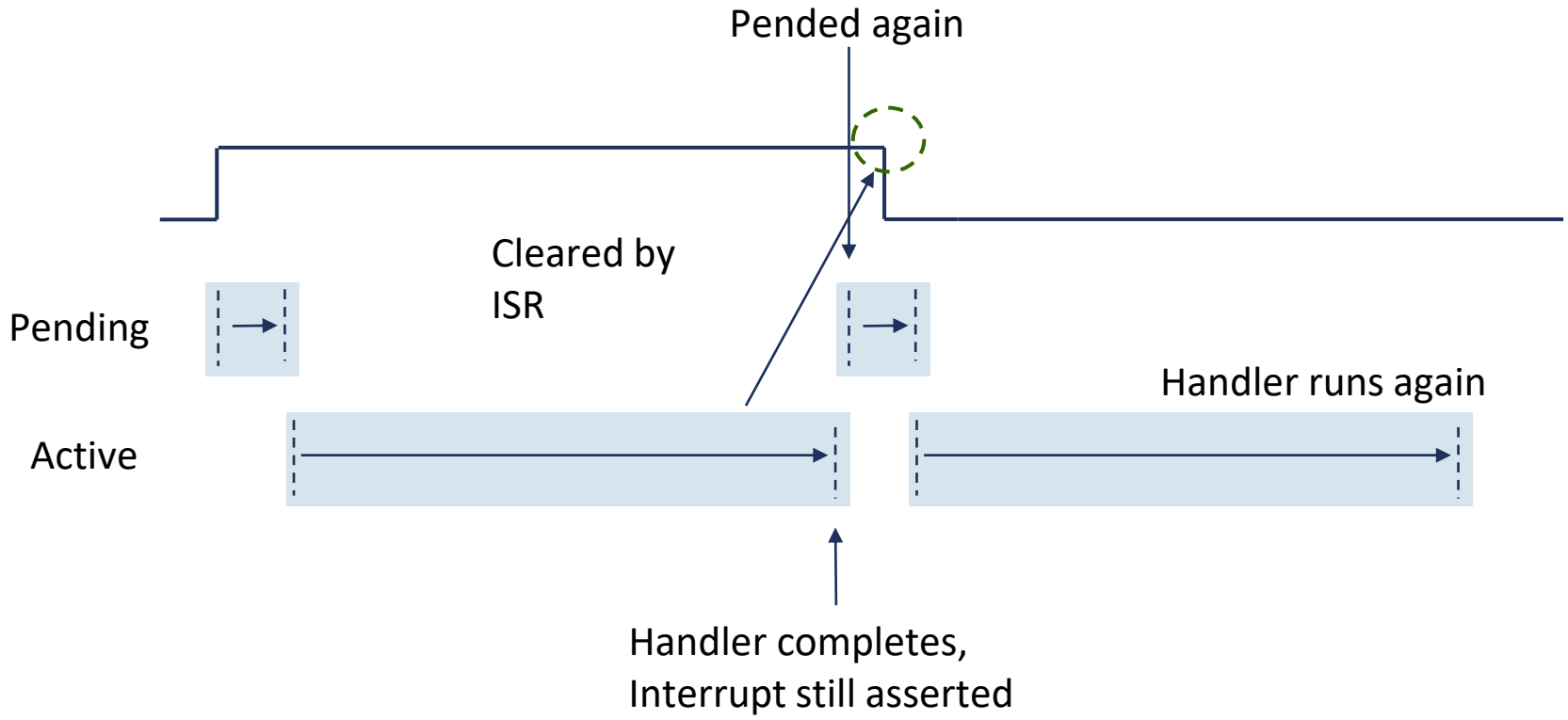
Example: ISR clears the interrupt source at the end of handling the interrupt



Pending the same interrupt again

An interrupt is pended again when a new low-to-high transition is detected

It is also pended again if the signal remains asserted after the handler exits



Agenda

Introduction

Exception Model

- Exception Entry and Exit Behaviour
- Prioritization and Control
- Interrupt Sensitivity

Writing the Vector Table and Interrupt Handlers

Internal Exceptions and RTOS Support

Fault Exceptions

References

CMSIS-CORE: vector table

```
; Vector Table Mapped to Address 0 at Reset

                AREA    RESET, DATA, READONLY, ALIGN=8
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors       DCD     __initial_sp
                DCD     Reset_Handler
                DCD     NMI_Handler
                DCD     HardFault_Handler
                DCD     MemManage_Handler
                DCD     BusFault_Handler
                DCD     UsageFault_Handler
                DCD     0, 0, 0, 0
                DCD     SVC_Handler
                DCD     DebugMon_Handler
                DCD     0
                DCD     PendSV_Handler
                DCD     SysTick_Handler

                ; External Interrupts
; ToDo: Add here the vectors for the device specific external interrupts handler
                DCD <DeviceInterrupt>_IRQHandler ; 0: Default
__Vectors_End
__Vectors_Size  EQU __Vectors_End - __Vectors
```

The vector table at boot is minimally required to have 4 values: stack top, reset routine location, NMI ISR location, HardFault ISR location

The SVCall ISR location must be populated if the SVC instruction will be used

Once interrupts are enabled, the vector table must then contain pointers to all enabled (by mask) exceptions

Writing interrupt handlers (1)

Clear the interrupt source if required

- Some may not need to be cleared (like SysTick)

Interrupt nesting always enabled by default

- No modification needed to interrupt handlers
- Just be careful of Main stack overflows

No need to set STKALIGN bit in Armv8-M

- Bit 9 in Configuration and Control Register is a RES1 bit

Writing interrupt handlers (2)

C Code is recommended for all handlers

- ISR function must be type void and can not accept arguments
  - The `__irq` keyword is optional (for clarity)

```
__irq void SysTickHandler (void) {
}
```

- No special assembly language entry and exit code required

Writing handlers in assembly

- Must manually save and restore any non-scratch registers which are used
  - R4, R5, R6, R7, R8, R9, R10, R11, LR
- Must maintain 8-byte stack alignment if making function calls
- Must return from interrupt using EXC_RETURN with proper instruction
  - LDR PC, …..
  - LDM/POP which includes loading the PC
  - BX LR

Interrupt management

Read from CLRPENDx / SETPENDx to determine whether IRQ is pended

Consider writing to CLRPENDx, for example, before enabling an IRQ

Read from ACTIVEx to determine if a specific ISR is active

Interrupt Program Status Register (IPSR) shows current active ISR

IPSR

Interrupt Control and State Register contains more information

0xE000ED04

P = ISRPENDING	Set if an interrupt is Pending and enabled
VECTPENDING	Interrupt number of highest pending enabled ISR
R = RETTOBASE	Return to Base, set if number of active exceptions = 1
VECTACTIVE	Interrupt number of the current executing ISR - Same as Interrupt Program Status Register (IPSR)

Agenda

Introduction

Exception Model

- Exception Entry and Exit Behaviour
- Prioritization and Control
- Interrupt Sensitivity

Writing the Vector Table and Interrupt Handlers

Internal Exceptions and RTOS Support

Fault Exceptions

References

Internal exceptions

SysTick Timer

- Provides system heart-beat for RTOS (or custom program executive)
- Periodic interrupt drives system task scheduling
- Better than a standard Timer Interrupt
  - Integrated into the NVIC
  - Does not vary between vendor implementation and between Cortex-M families

Supervisor Call (SVC)

- Similar to “SWI” instruction on older Arm cores
- Allows non-privileged software to make system calls
  - RTOS services requests from non-privileged mode
  - Provides protection to important system functionality
  - Examples: Open Serial Port, Remap Memory, Enter Low Power Mode, etc.

Pended System Call (PendSV)

- Operates with SVC to ease RTOS development
- Intended to be an interrupt for RTOS use

Supervisor Call (SVC)

Can be used from Thread or Handler mode (even if privileged)

Assembly Syntax

```
SVC #immediate8
```

C Syntax uses `__svc` keyword

```
return-type __svc (int svc_num) function-name ([arguments])
```

Example:

```
void __svc(42) terminate_procedure (int proc_number)
```

Restrictions

- Cannot be called from SVCall or any handler with same or higher priority (causes a further Fault)
  - Including NMI or HardFault Handler

SVC handlers

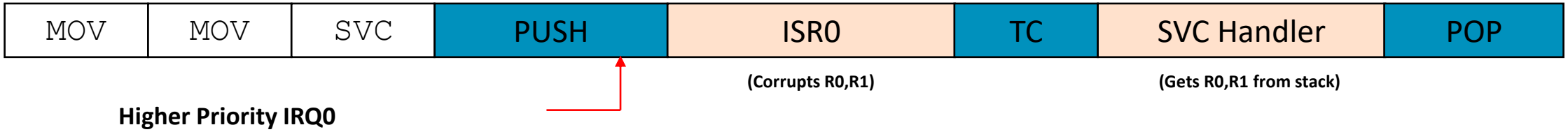
SVC Handler may have to determine SVC number for processing

- The stacked return address is used to determine location of SVC instruction
- Then the lower byte of the opcode is extracted

Any arguments to the SVC are passed in R0-R3 (like a normal call)

- Registers may be corrupted by a late arriving interrupt
- Arguments must be retrieved from stack

```
__svc (0x5) void OpenPort (0x20, 0xF)
// MOV r1, #0xF  ; port to open
// MOV r0, #0x20 ; port mode
// SVC #0x5      ; OpenPort SVC number
```



Pended system call (PendSV)

Permanently enabled interrupt

- Avoids need for long periods of interrupts disabled

Not an instruction – but instigated via Interrupt Control and State Register

- May only be written in privileged mode
- Set by PENDSVSET bit
- Cleared by PENDSVCLR bit

Intended usage

- As a private, low-level interrupt for the RTOS
- SVC cannot be nested, so RTOS uses PendSV to perform specific tasks
- PendSV can be given same or lower interrupt priority versus SVC
  - PendSV will only run after SVC completes (and before foreground is returned)
  - PendSV can then handle low-priority RTOS “cleanup” or perform context switch

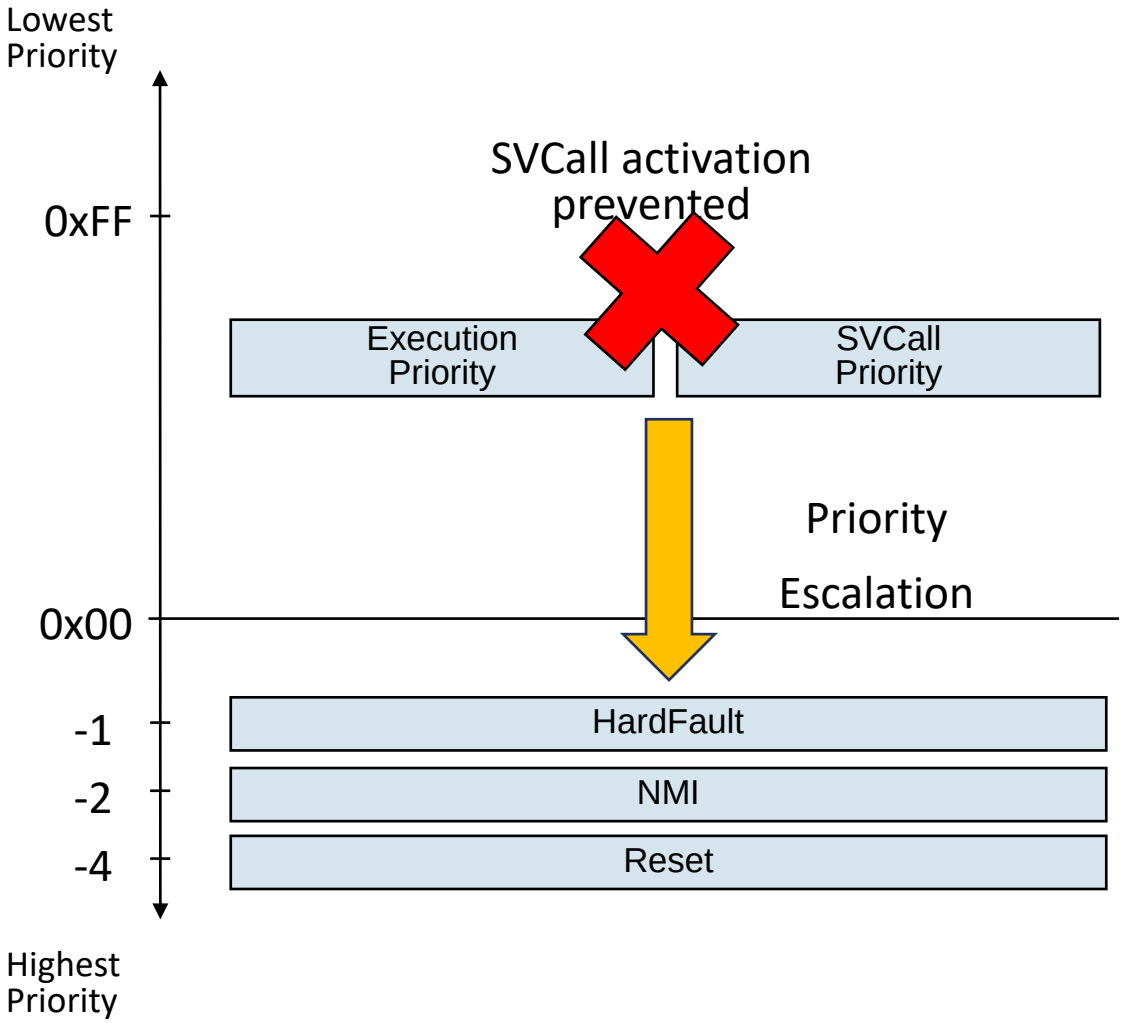
Priority Escalation

If an SVCall exception is lower than or equal to the currently executing group priority, preventing normal preemption, a HardFault is taken

Priority escalation applies when the currently executing group priority is less than HardFault

- If an exception occurs at a currently executing group priority of HardFault or higher, the behavior is entirely unexpected and fatal

A fault that is escalated to a HardFault retains the ReturnAddress behavior of the original fault



Internal interrupt registers

SysTick, SVC and PendSV are managed through dedicated registers

Interrupt Control and State Register (0xE000ED04)

- Contains the pending bits – for setting and clearing

System Handler Control and State Register (0xE000ED24)

- Contains the active bits

System Handler Priority Register 2 (0xE000ED1C)

- Contains priority for SVCall

System Handler Priority Register 3 (0xE000ED20)

- Contains priority for PendSV and SysTick

Agenda

Introduction

Exception Model

- Exception Entry and Exit Behaviour
- Prioritization and Control
- Interrupt Sensitivity

Writing the Vector Table and Interrupt Handlers

Internal Exceptions and RTOS Support

Fault Exceptions

References

Fault exceptions in Armv8-M

Three classes of general faults

- BusFault – Memory access error (Prefetch or Data Access)
- MemManage – MPU permissions mismatch
- UsageFault – Undefined instruction, CP access, illegal state transition (i.e. T=0)
  - Can optionally trap divide by zero and unaligned memory accesses

All three faults are disabled by default

- Only enable if required
- Manually enabled in the System Handler Control and State Register
- It is possible to set the priority level for each fault

HardFault – “fall back” fault

- Always enabled
- Priority level “-1”, but Secure version can be programmed to priority -3
- Executes through “Fault Escalation”

SecureFault

- Security violation (where Secure Extension is implemented)

Fault escalation

Fault Escalation to HardFault occurs when a fault occurs but…

- The fault is not enabled or…
- The handler doesn't have enough priority to run or…
- The fault handler encounters the same fault

Examples

- An undefined instruction is decoded
  - Processor tries to signal a UsageFault, but the UsageFault is not enabled
- With the MPU enabled an interrupt handler makes an illegal memory access
  - Processor tries to signal a MemManage fault, but the fault priority is lower than the priority of the running interrupt
- An undefined instruction is encountered in a UsageFault handler
  - An enabled fault handler causes the same kind of fault as is being serviced

Fault handling

A simple system would only use the HardFault handler

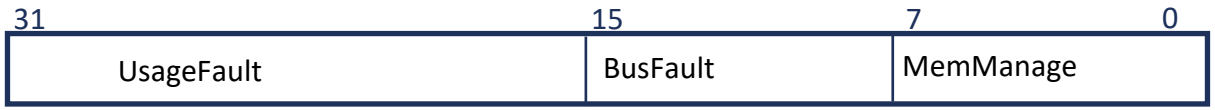
- No other fault handlers enabled
- Enable separate fault handlers only when you have a need to use them

MemManage and BusFault have a Fault Address Register

Each fault has an associated Status Register

- Interrogate Status Register to determine fault source
- Take “appropriate action”

0xE000ED28



Same rules apply for writing fault handler routines as for ISRs

A fault inside a HardFault handler will lock up the core

- LOCKUP output asserted, perhaps used to inform watchdog to trigger reset

The Lockup state

In unrecoverable exception cases Lockup state is entered

- Vector table faults
  - Fault on reading MSP initial value or Reset vector
  - Fault on reading NMI vector
  - Fault on reading HardFault vector

- Any Fault when executing at a negative priority number (NMI or HardFault)
  - Examples: Memory faults, Undefined instruction or execution of SVC instruction

Lockup state behavior

In lockup state:

- DHCSR.S_LOCKUP reads as 1
- The PC reads as 0xEFFFFFFE - this is an XN address
- The PE stops fetching and executing instructions
- If the implementation provides an external LOCKUP signal, LOCKUP is asserted HIGH

Processor can lockup at different priority levels

- Lockup at -2
  - NMI vector fault
  - Handling NMI exception
  - HardFault while saving context on HardFault-to-NMI pre-emption
- Lockup at -1 or -3
  - On all other vector table faults
  - Handling HardFault exception

NMI can pre-empt the processor when in lockup -1 state

Exit the Lockup state

Lockup state can be exited only in the following cases:

- NMI exception when the locked up priority is -1
- Debug event (Debug extensions)
- Reset

Implementation might provide LOCKUP output

Synchronous exceptions

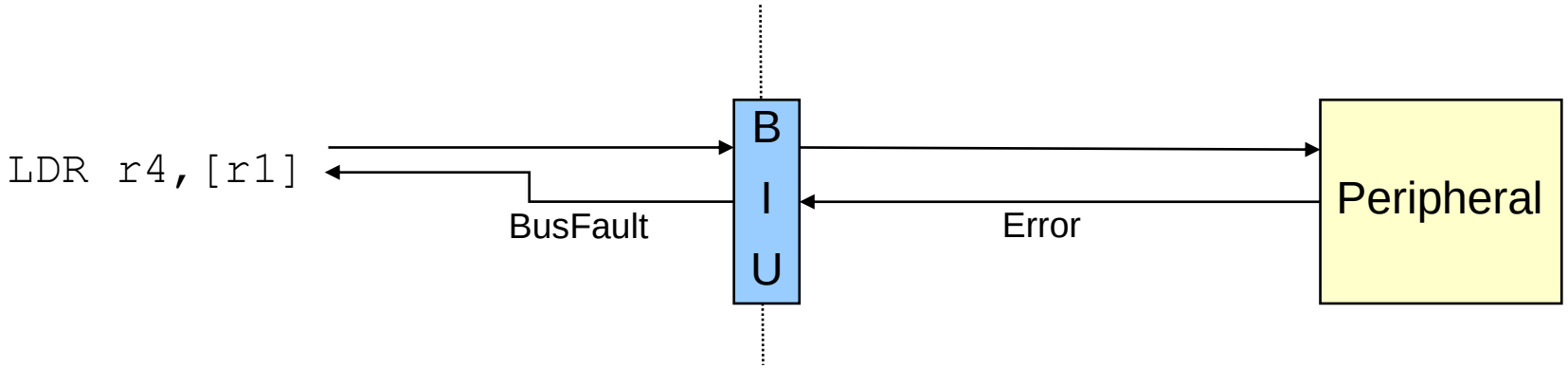
Synchronous exceptions have a fixed relationship to the instruction stream, e.g.

- Bus error on an instruction fetch
- Bus error on a load/store instruction
- Execution of an undefined instruction

Synchronous exceptions are always precise

Example:

- Precise BusFault due to data abort on bus



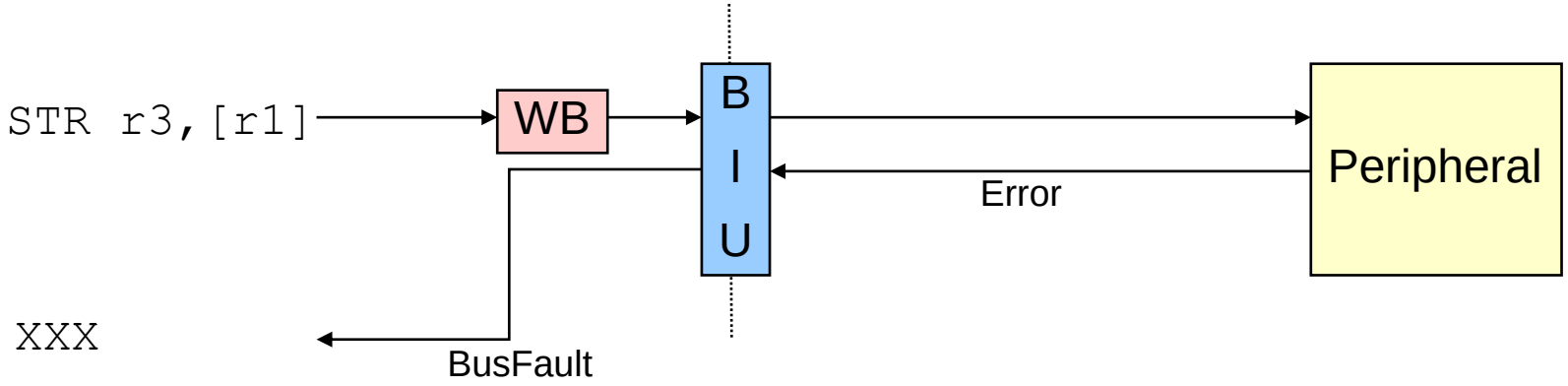
Asynchronous exceptions

Asynchronous exceptions have no fixed relationship to the instruction stream, e.g.

- Bus error on an buffered store operation
- Interrupts

Imprecise Asynchronous faults are treated as unrecoverable

- BusFault due to an erroneous access to memory using a Write Buffer



Agenda

Introduction

Exception Model

- Exception Entry and Exit Behaviour
- Prioritization and Control
- Interrupt Sensitivity

Writing the Vector Table and Interrupt Handlers

Internal Exceptions and RTOS Support

Fault Exceptions

References

References

Armv8-M Architecture Reference Manual

- Chapter B11: Nested Vectored Interrupt Controller

Application Note:

- AN209 - Using Cortex-M3/M4/M7 Fault Exceptions

Books

- The Definitive Guide to the Cortex-M3 and Cortex-M4 Processors (ISBN: 978-0124080829) - Yiu

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