Armv8-M Mainline Programmers' Model

Learning objectives

- Define the base data types used by Armv8-M
- Describe the Arm register bank
- Demonstrate how the stack pointer, link register and program counter are used
- Describe the different processor modes of operation
- Explain the difference between privileged versus unprivileged execution
- Introduce the exception handling mechanism
- Introduce the T32 Instruction Set
- Describe common programming standards such as the AAPCS and CMSIS-Core

Agenda

Introduction

Data Types

Core Registers

Modes, privilege and stacks

Exceptions

Instruction Set Overview

Programming standards

Armv8-M profile overview

Armv8-M processors are designed to support the microcontroller market

- Based on a RISC architecture and most instructions execute in a single cycle
- Simpler to program – entire application can be programmed in C
- Fewer features needed than in application processors

Register and ISA changes from other Arm cores

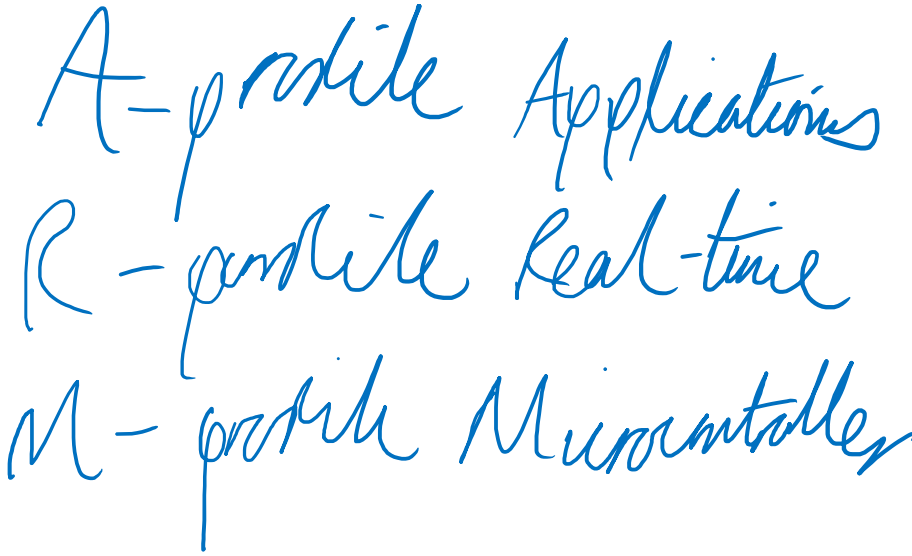
- No Arm instruction set support
- Armv8-M only banks the stack pointer
- Two modes of operation: Thread mode & Handler mode

Different modes and exception models

- Only two execution modes: Thread and Handler mode
- Vector table consists of addresses, not instructions
- Exceptions automatically save state (R0-R3, R12, LR, ReturnAddress, RETPSR) on the stack

Different system control/memory layout

- Cores have a fixed memory map
- No coprocessor 15 – controlled through memory mapped control registers



Agenda

Introduction

Data Types

Core Registers

Modes, privilege and stacks

Exceptions

Instruction Set Overview

Programming standards

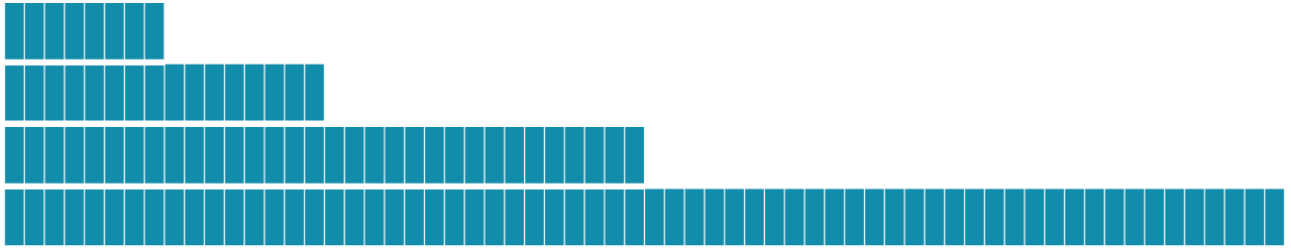
Data types

Armv8-M is a 32-bit load / store architecture

- The only memory accesses allowed are loads and stores
- Most internal registers are 32 bits wide

When used in relation to Arm architectures the following data types are used

- **Byte** 8 bits
- **Halfword** 16 bits
- **Word** 32 bits
- **Doubleword** 64 bits



Agenda

Introduction

Data Types

Core Registers

Modes, privilege and stacks

Exceptions

Instruction Set Overview

Programming standards

Armv8-M Mainline registers

Registers R0-R7

- Accessible to all instructions

Registers R8-R12

- Not available to all (16-bit) instructions

R13 is the stack pointer (SP)

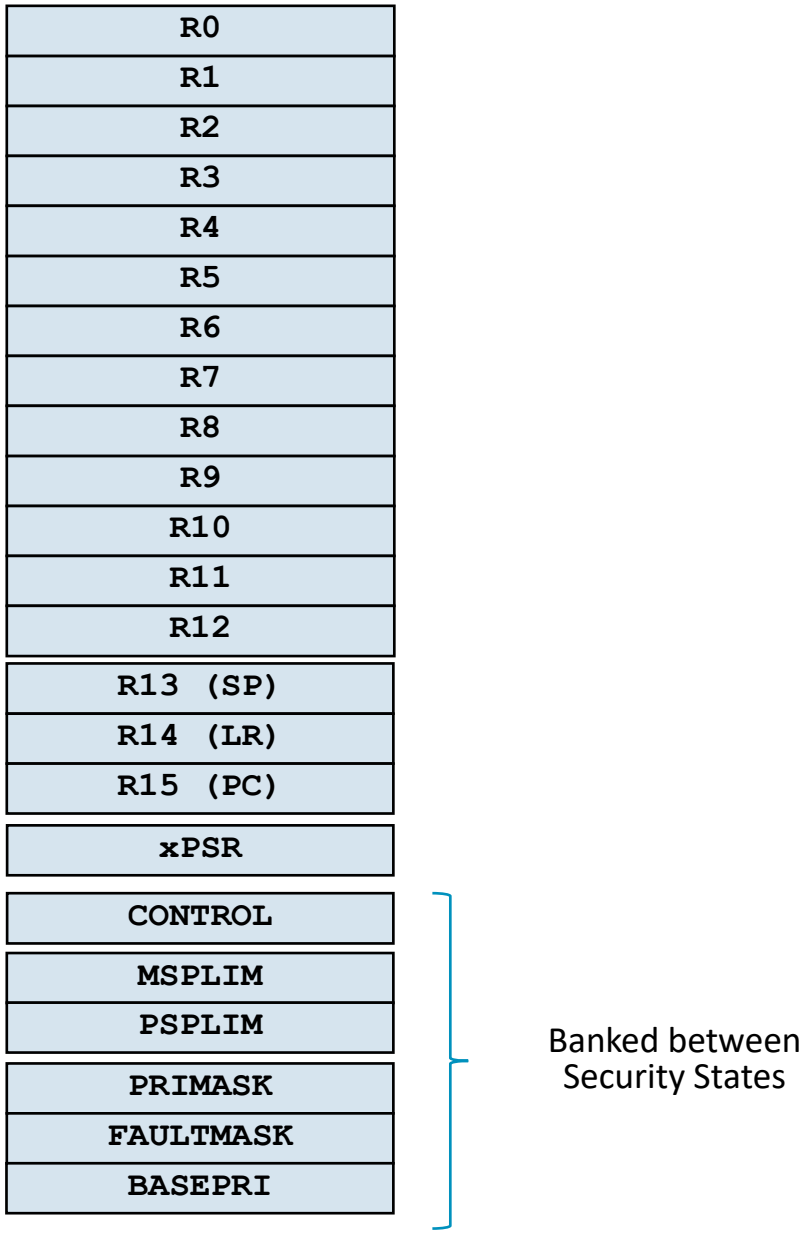
- Armv8-M PEs have two banked versions

R14 is the link register (LR)

R15 is the program counter (PC)

Special-purpose Registers

- Program Status Registers (xPSR)
  - APSR, EPSR, IPSR
- CONTROL
  - FP usage, Stacks and Privilege, PACBTI enable bits
- MSPLIM, PSPLIM
  - Stack Pointer limit registers
- PRIMASK, FAULTMASK, BASEPRI
  - Exception Handling

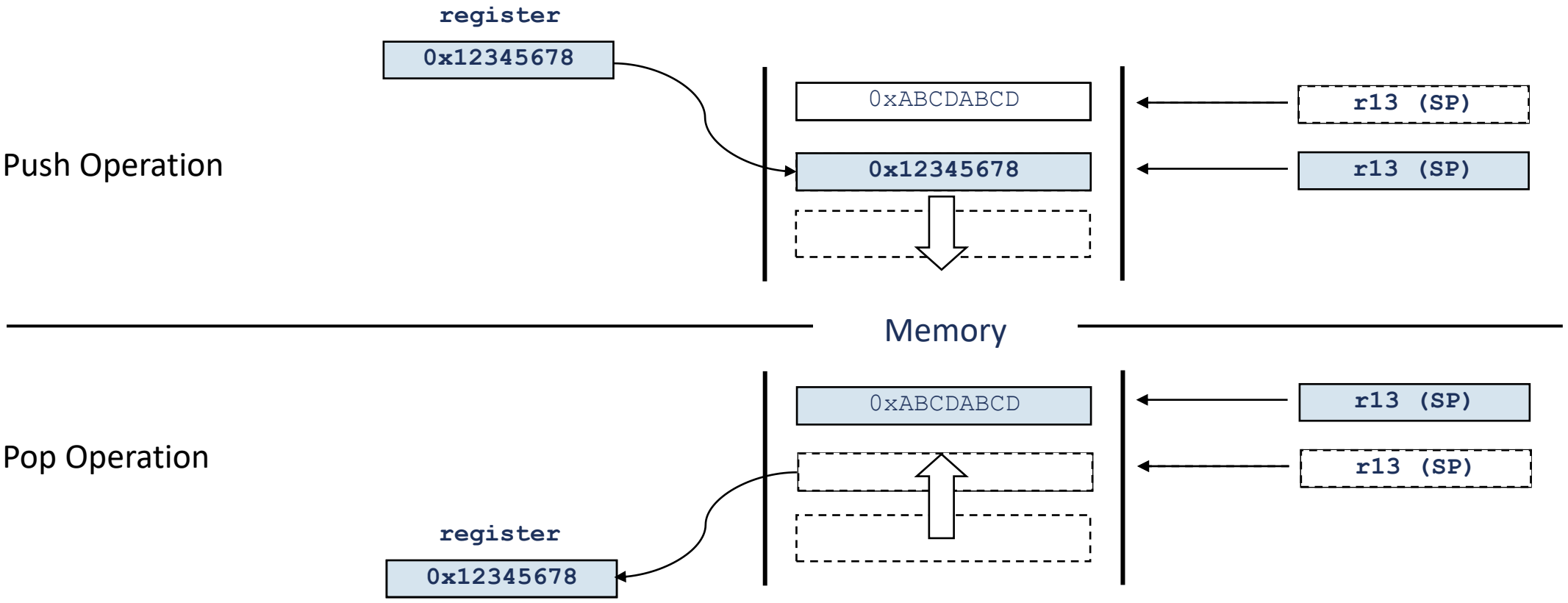


Stack Pointer (SP)

Usage of a stack is to save register contents in memory

- Armv8-M only supports Full Descending Stack

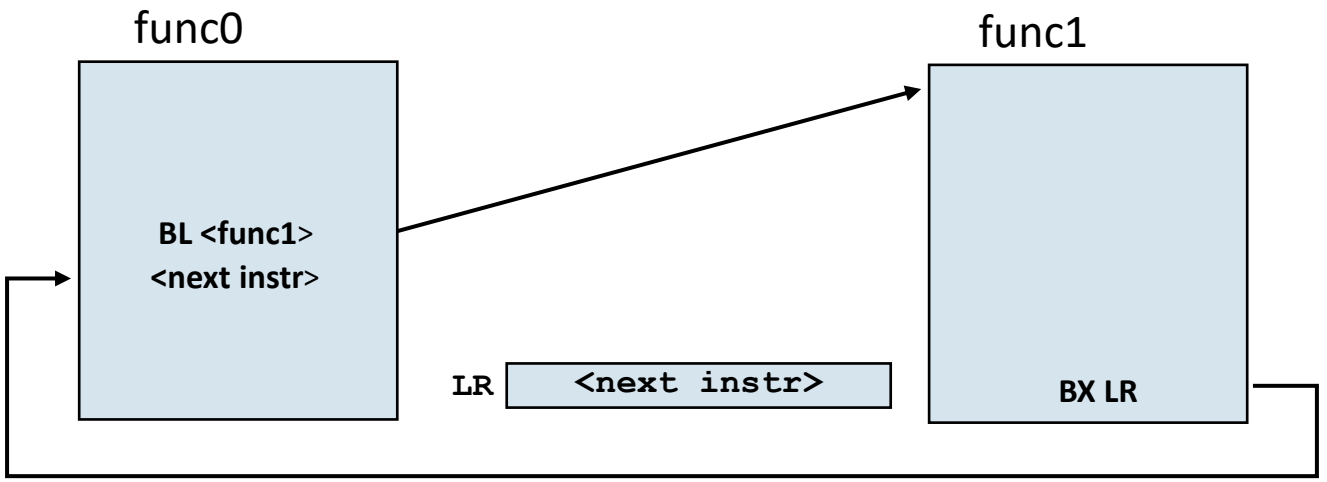
The Stack Pointer (SP) points to a memory location – the stack



Link Register (LR)

The Link Register (LR) is used to enable returns from subroutines

```
void func0 (void)
{
        :
        func1();
        :
}
```



Further usage of the LR

- It has a special function for exception handling

Program Counter (PC)

The program counter can be used for different purposes such as changing the program flow

Some instructions are allowed to read from the PC

- For example, PC-relative data addressing

Some instructions are allowed to write to the PC

- For example, a load instruction with the PC as the destination register
- Bit [0] of any loaded value must be set to 1

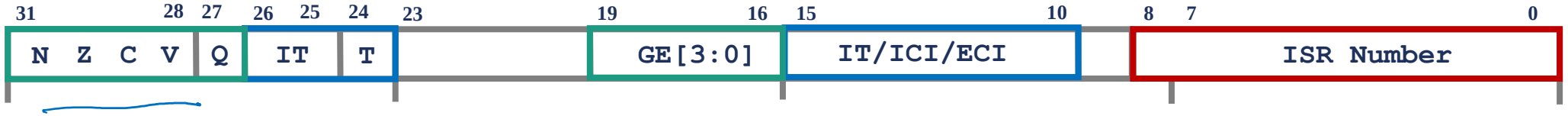
Some instructions implicitly access the PC

- For example, a branch instruction (`B<cond>`) reads from and writes to the PC

When it is permitted to specify the PC, its value is interpreted differently depending on the instruction

- For example, when read implicitly using a BL instruction, the PC points to the address of the current instruction + 4

Program Status Registers



Application Program Status Register [APSR]

- Only ALU flags

Interrupt Program Status Register [IPSR]

- Interrupt/Exception Number

Execution Program Status Register [EPSR]

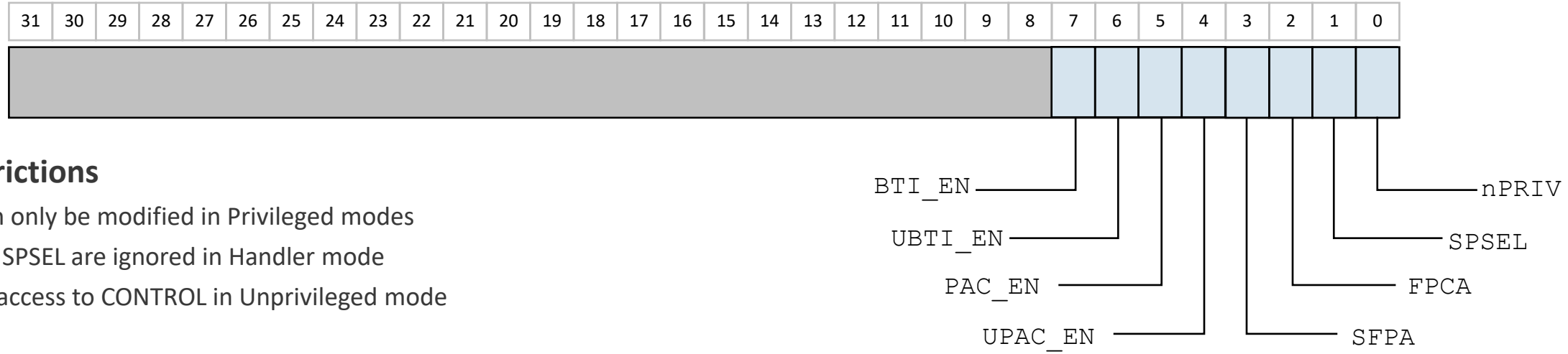
- IT field – If/Then block information
- ICI field – Interruptible-Continuable Instruction information
- T bit (s/b =1, to show core is in Thumb state)

xPSR

- Composite of the 3 PSRs
- Stored on the stack on exception entry

CONTROL Register

The special purpose CONTROL Register (in Mainline) is a 2, 3, 4, or 8-bit register



Access restrictions

- nPRIV can only be modified in Privileged modes
- Writes to SPSEL are ignored in Handler mode
- No write access to CONTROL in Unprivileged mode

nPRIV – Defines the execution privilege in Thread mode

SPSEL – Defines the stack to be used in Thread mode

FPCA[1] – Defines whether the FP extension is active in the current context

SFPA[2] – Indicates the active use of FP registers in Secure State

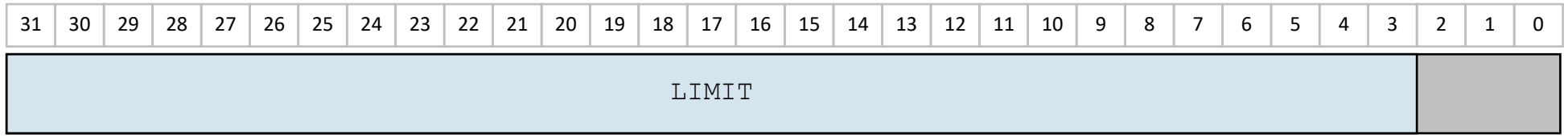
BTI_EN, UBTI_EN, PAC_EN and UPAC_EN[3] – Enable bits for the PACBTI Extension

[1] Available only if Floating Point Extensions is enabled
[2] Available only if Security Extension is enabled
[3] Available only if Pointer authentication branch target identification Extension is enabled

Stack pointer limit registers

MSPLIM – Main Stack Pointer limit register

PSPLIM – Process Stack Pointer limit register



LIMIT value is the bits[31:3] of the lowest limit of the defined stack

I_{KDPG} A stack can descend to its stack limit value. Any attempt to descend the stack further than its stack limit value is a violation of the stack limit.

Applies to an implementation of the architecture ***Armv8.0-M*** *onward.*

The registers are privileged access only

An implementation with the Security Extension provides four stack limit registers

- MSPLIM_S, MSPLIM_NS, PSPLIM_S and PSPLIM_NS

Other special-purpose registers

Other special purpose registers are used in managing the prioritization scheme for exceptions

- PRIMASK – Exception Mask Register
- FAULTMASK – Fault Mask Register
- BASEPRI – Base Priority Mask Register

Discussed in more detail in Armv8-M Exception Handling

Agenda

Introduction

Data Types

Core Registers

Modes, privilege and stacks

Exceptions

Instruction Set Overview

Programming standards

Privileged execution

Privileged/non-privileged operation

- Handler mode is always privileged
- Thread mode can be privileged or non-privileged, depending on the value of CONTROL.nPRIV

Privileged execution is normally required to manage system resources

- The memory mapped control registers in the System Control Space (SCS) require privileged access
- Privileged execution will generate a different memory access to non-privileged access
- When code is executing unprivileged, Thread mode can execute an `SVC` instruction to generate a supervisor call exception

Stacks

Two run-time models supported

- Single Stack Pointer – MSP for entire application
- Two Stack Pointers
  - MSP for Handler Mode (Exception Handling)
  - PSP for Thread Mode (Application Code)

Main Stack Pointer (MSP)

- Used by Thread Mode out of reset
  - Initial MSP value is taken from first entry of Vector Table
- Always used by Handler Mode

Process Stack Pointer (PSP)

- Optionally used for Thread Mode
- PSP is enabled using CONTROL.SPSEL
  - Must be initialized by user before being used

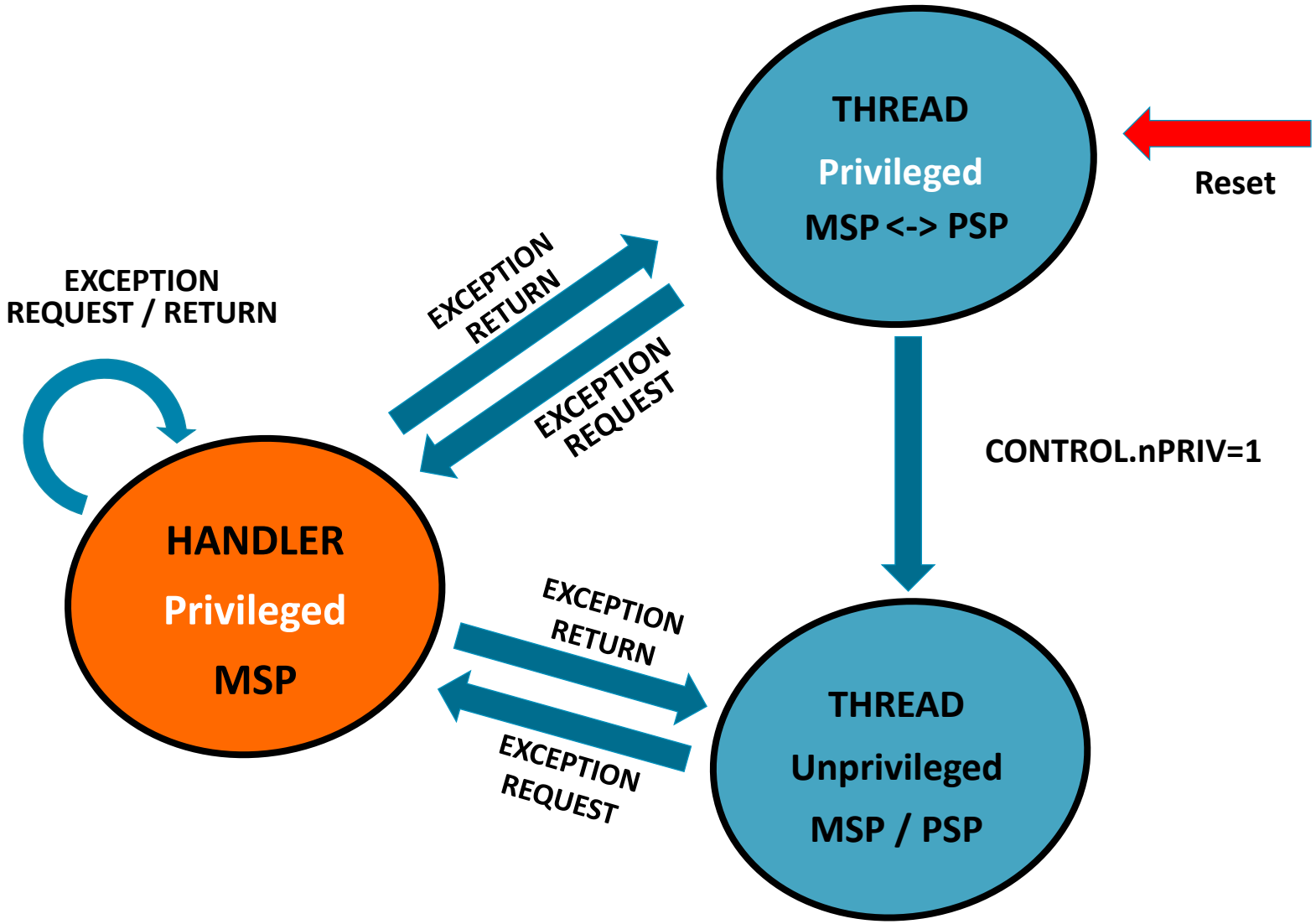
Modes overview

Thread Mode

- Used for application execution
- Can use either
  - Main Stack Pointer (MSP)
  - Process Stack Pointer (PSP)
- Can be in either
  - Privileged mode
  - Unprivileged mode
- At reset: MSP + Privileged mode

Handler Mode

- Used for exception handling
- Entered on an exception
- Always in Privileged mode
- Only uses Main Stack Pointer



Agenda

Introduction

Data Types

Core Registers

Modes, privilege and stacks

Exceptions

Instruction Set Overview

Programming standards

Exception handling

Exception types:

- Reset
- Non-maskable Interrupt (NMI)
- Faults
  - HardFault, UsageFault, MemManage, BusFault & SecureFault
- PendSV
- SVCall
- External Interrupts (IRQs)
- SysTick Interrupt

Exceptions processed in Handler mode (except Reset)

- Exceptions always run as privileged

Armv8-M exception handling reduces software overhead

- Automatic save and restore of processor registers (R0-R3, R12, LR, ReturnAddress, RetPSR)
- Allows handler to be written entirely in ‘C’
- Automatic 8-byte stack alignment on exception entry

Agenda

Introduction

Data Types

Core Registers

Modes, privilege and stacks

Exceptions

Instruction Set Overview

Programming standards

Instruction set support

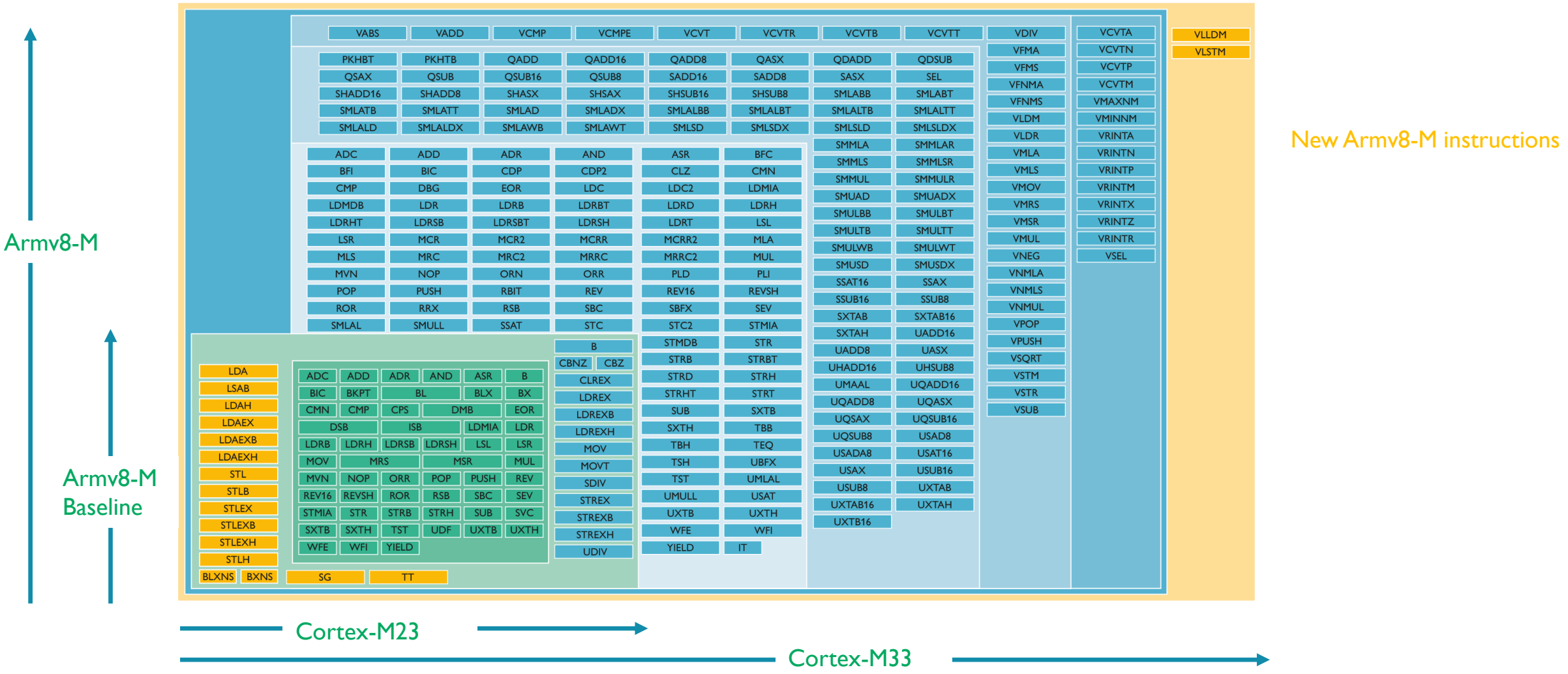
Armv8-M Mainline PEs implement the T32 instruction set

- Derived from the Armv7-M
  - Mix of 16-bit and 32-bit instructions
  - Instruction length can vary, depending on functionality
- Load/Store instruction set; no direct manipulation of memory contents
- Extends Armv7-M by adding Load-Acquire, Store-Release

Optional Security Extension is available

- Additional instructions:
  - Function calls between security states (SG, BXNS, BLXNS)
  - Memory protection unit address queries (TT)

Armv8-M instruction set



Instruction set examples:

Data Processing:

```
MOVS    r2, r5                  ; r2 = r5
ADDS    r5, #0x24               ; r5 = r5 + 36
LSLS    r2, #3                  ; r2 = r2 * 8
MOVT    r9, #0x1234             ; upper halfword of r9 = #0x1234
MULS    r0, r1, r2              ; r0 = r1 * r2
```

ADD r5, r5, #0x24

Memory Access:

STR r2, [r11] LDR r1, [r4, #4]

```
STRB    r2, [r10, r1]           ; store lower byte in r2 at
                                  address {r10 + r1}
LDR     r0, [r1, r2, LSL #2]    ; load r0 with data at address
                                  {r1 + r2 * 4}
LDAH    r3, [r2]                ; Load Acquire halfword with data
                                  at address (r2)
STLEX   r1, r3, [r5]            ; Store and release if PE has
                                  exclusive access
```

Instruction set examples:

Program Flow:

```
B       <label>          ; PC relative branch to <label>
                           location

BL      <label>          ; PC relative branch to <label>
                           location, and return address
                           stored in LR (r14)

BX      R2               ; Branch to address in R2
```

Security:

```
BLXNS    R6              ; Branch and exchange to Non-secure
                           state function - address in r6

SG                       ; Secure gateway to mark a valid
                           branch target for Non-secure calls
```

Agenda

Introduction

Data Types

Core Registers

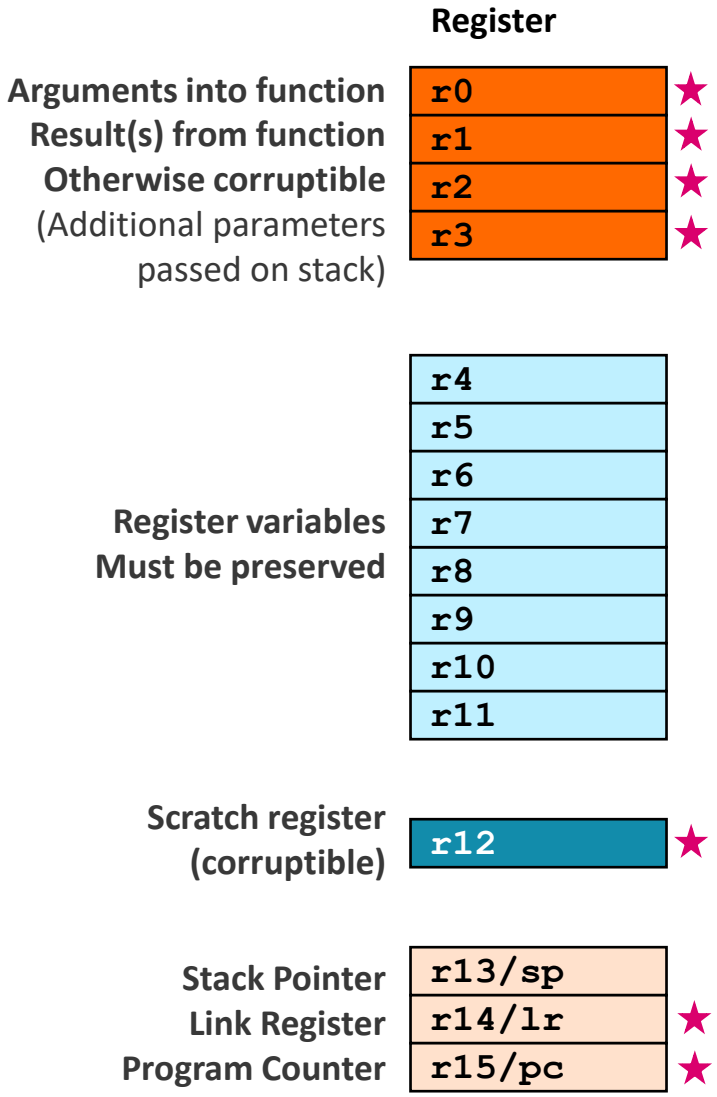
Modes, privilege and stacks

Exceptions

Instruction Set Overview

Programming standards

Arm Procedure Call Standard



The compiler has a set of rules known as a Procedure Call Standard that determine how to pass parameters to a function (see AAPCS)

xPSR flags may be corrupted by function call

Assembler code which links with compiled code must follow the AAPCS at external interfaces

The AAPCS is part of the ABI for the Arm Architecture

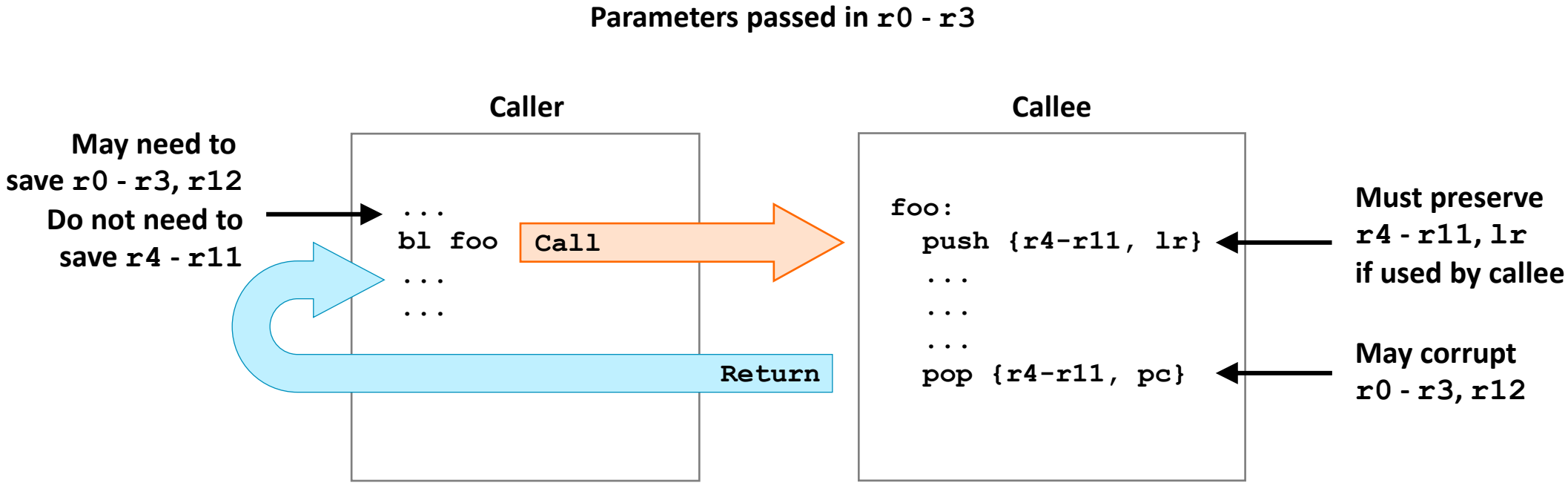
AAPCS requires that SP be 8-byte (2 word) aligned at externally visible boundaries

r14 can be used as a temporary register once value stacked

In Armv8-M, registers marked with a star ★ are automatically pushed on to the stack when an exception occurs

The xPSR ★ (processor state) is also pushed to the stack

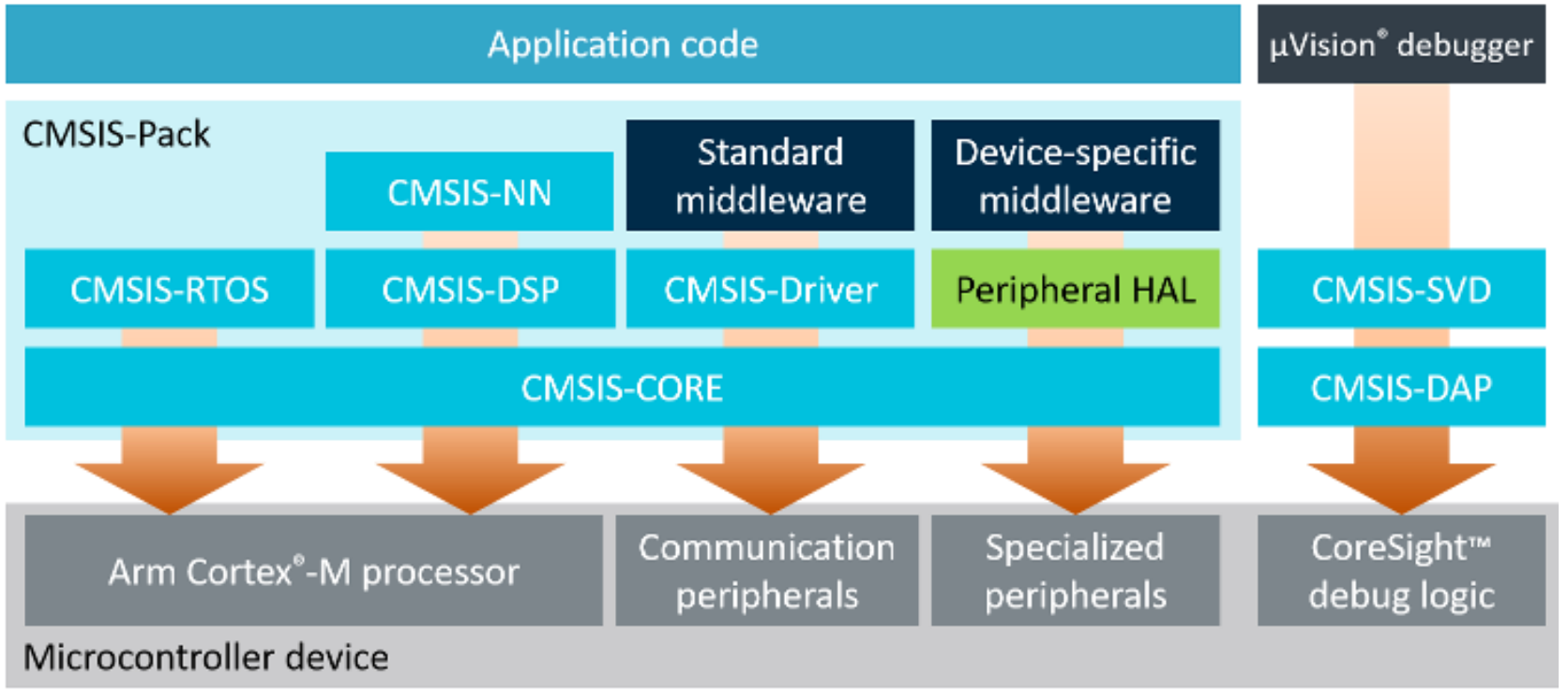
Arm Procedure Call Standard (2)



Value returned in `r0` for `int` / `short` / `char` ;
in `r0` and `r1` for `long long`

AAPCS – Procedure Call Standard for Arm Architecture

Cortex Microcontroller Software Interface Standard



CMSIS-Core overview

CMSIS-Core is an API for the Cortex-M processor core and device peripherals

- Hardware Abstraction Layer (HAL) for Cortex-M processor registers with standardized definitions for the SysTick, NVIC, System Control Block registers, MPU registers, FPU registers, and core access functions
- Intrinsic functions for instructions that cannot be generated directly from C/C++
- Organization and standard naming conventions

Tested and verified toolchains

- Arm Compiler
- GNU Tools ARM Embedded
- IAR Embedded Workbench Kickstart Edition

CMSIS-Core also supports a selected subset of Cortex-A devices

CMSIS-Core examples

Instruction access

```
void __NOP (void)
void __DMB (void)
void __WFI (void)
uint32_t __REV (uint32_t value)
```

Device start-up and vector table

- Reset handler, initial MSP value
- Device specific interrupt configuration

DSP/SIMD instructions

```
uint32_t __SADD8(uint32_t val1, uint32_t val2)
uint32_t __SMUAD (uint32_t val1, uint32_t val2)
```

Special register access

```
void __set_CONTROL (uint32_t value)
uint32_t __get_PSP (void)
void __set_FPSCR (uint32_t value)
```

NVIC access

```
__set_PRIMASK(uint32_t priMask)
void NVIC_DisableIRQ(IRQn_Type IRQn)
```

SysTick access

```
SysTick_Config(uint32_t ticks)
```

Debug access

- Instrumentation Trace Macrocell (ITM)

  `ITM_SendChar`

  `ITM_ReceiveChar`

Reference material

All Arm product documentation available through Arm Developer

- https://developer.arm.com/

Arm Architecture Reference Manual (“Arm Arm”)

- Armv8-M Architecture Reference Manual

Arm Compiler toolchain

- armasm User Guide

Technical Reference Manuals for processor core being used

- More recent cores may only be available to direct licensees

Arm Application Binary Interface (ABI)

- https://developer.arm.com/architectures/system-architectures/software-standards/abi

CMSIS bundle and documentation

Blog: Which CMSIS components should I care about?

- https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/which-cmsis-components-should-i-care-about

CMSIS files and documentation (html) can be downloaded from the ARM website:

- https://github.com/ARM-software/CMSIS_5 (login/registration required)

CMSIS files and documentation are part of a Keil MDK-ARM installation:

- C:\Keil_v5\ARM\PACK

CMSIS documentation is also available from the GitHub:

- http://arm-software.github.io/CMSIS_5/General/html/index.html

CMSIS-DAP firmware and documentation is available as a separate download:

- https://silver.arm.com/browse/CMSISDAP

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