



# **Microprocessors & Microcontrollers**

## **: Arm Cortex M0+**

### **(Using RP2040)**

## **Lecture ES11**

### **About Arm Cortex-M0+**

# Focus

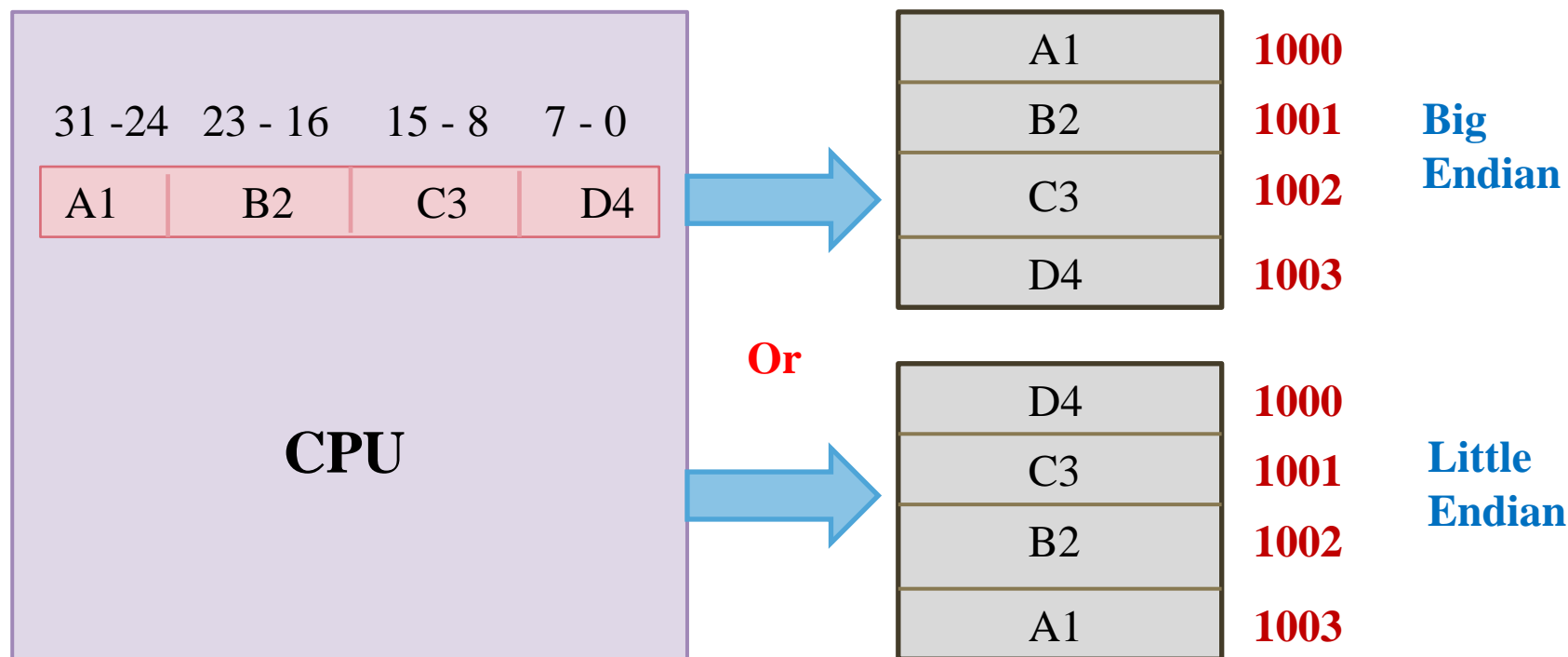
- Endianness (Big Vs Little)
- Arm Cortex-M0+
  - ARMv6-M Architecture Profile
  - Features
  - External Interfaces
  - Configurations
  - Register Set



# Endianness (Big Vs Little)

# Big-Endian vs Little-Endian

- When a register content from a processor is moved to memory, it can be saved in two different ways
  - Big Endian**
  - Little Endian**



# Quiz 1

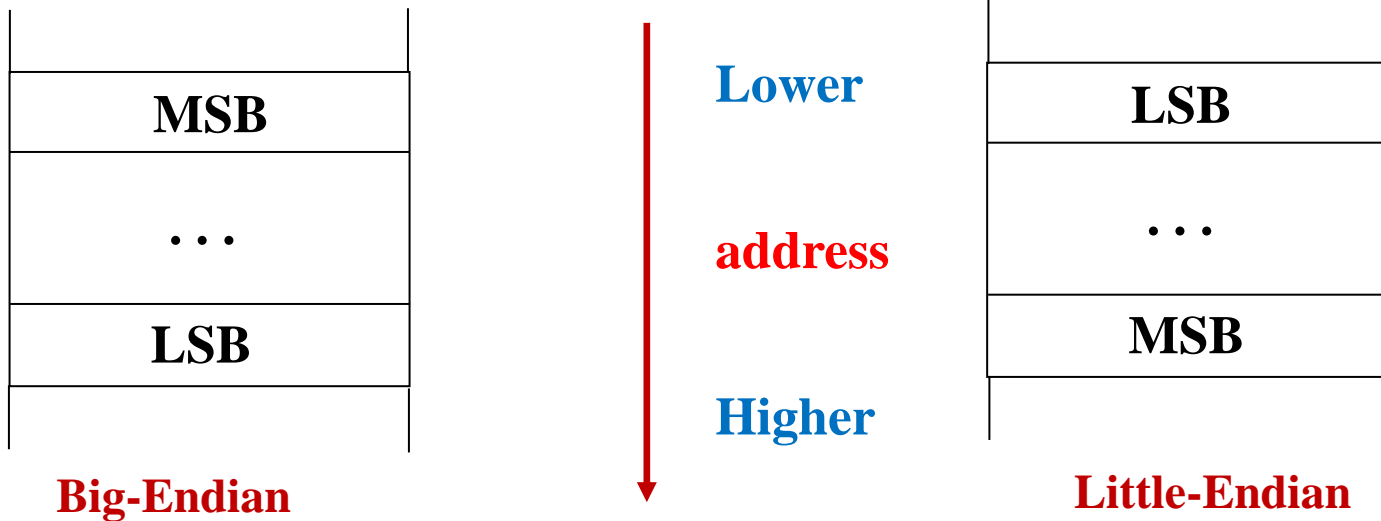
## Choose the correct option:

1. A CPU wrote a register content into memory in a Big-Endian mode. When the same content is read back from the memory into a register, the CPU is reading it in Little-Endian mode.
- a) The new content in the register will be different from what was written.
  - b) The content will be the same.
  - c) The content may be same or different.
  - d) Endianness need not be the same while writing and reading the contents into/from the memory.

**Correct option: a**

**Note:** The endianness needs to be the same. The endianness used while writing a content into the memory **should** be used while reading the same content back into a register.

# Big-Endian and Little-Endian



## Bi-Endian

Motorola 68xx, 680x0

IBM Mainframe

HP PA-RISC

Internet TCP/IP

ARM

Motorola Power PC

Sun SPARC

MIPS

Intel x86

AMD Opteron

DEC VAX

PIC

Microcontroller

# Big-Endian and Little-Endian

- Addition in Assembly

- Example: `ADD r0, r1, r2` (in ARM)

Equivalent to: `a = b + c` (in C)

where ARM registers `r0, r1, r2` are associated with C variables `a, b, c`

- Subtraction in Assembly

- Example: `SUB r3, r4, r5` (in ARM)

Equivalent to: `d = e - f` (in C)

where ARM registers `r3, r4, r5` are associated with C variables `d, e, f`



# Big-Endian and Little-Endian

① Write the Each program line output, and store the data to memory, which is of Big Endian.

MOV R1, #0X85

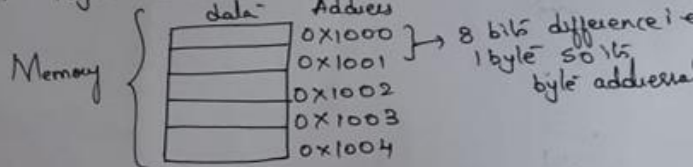
MOV R2, #0X35

ADD R4, R1, R2

STR R4, [R3]

R3 = 0X1000

Case: Byte Addressable



Solution: MOV R1, #0X85 ; Moves the immediate number to GPR R1  $\therefore$  R1 = 0X85

MOV R2, #0X35 ;  $\Rightarrow$  R2 = 0X35 (hexadecimal)

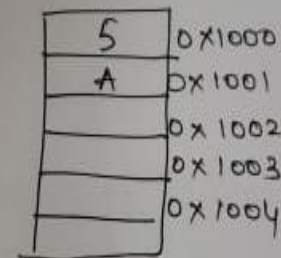
ADD R4, R1, R2 ;  $\Rightarrow$  R4 = R1 + R2  
 $= 0X85 + 0X35$   
 Hexadecimal Add

$\therefore = 0X5A$   
 $\Rightarrow$  R4 = 0X5A

STR R4, [R3] ; Store instruction loads the R4 Register data to memory location pointed by R3 i.e. (1000)

$\therefore$  It's given memory in Big Endian (BE)  
 In BE lower byte in higher address & higher byte in lower address.

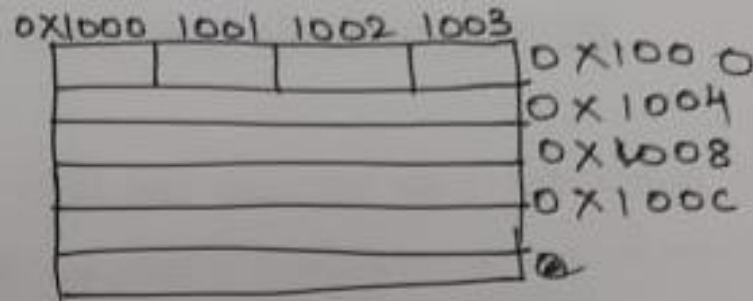
R4 = 0X5A  
 Higher Byte  $\rightarrow$  lower byte



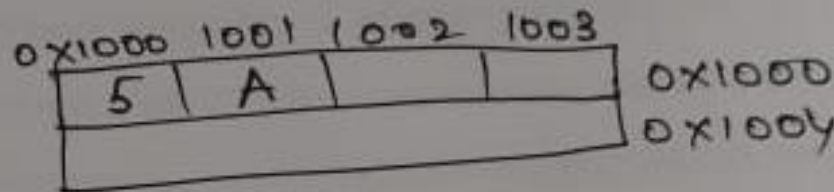


# Big-Endian and Little-Endian

Case ii) Word Addressable: where addresses differ by 1 word (i.e 4 bytes i.e 32 bits)



So in this case result  $R4 = 0x5A$  is stored as



# Endianness Comparison

## Advantages and Disadvantages

### Big-Endian

- Easier to determine a sign of the number
- Easier to compare two numbers
- Easier to divide two numbers
- Easier to print

### Little-Endian

- Easier for addition and multiplication of multi-precision numbers
- It's easy to read the value in a variety of different datatype sizes
- It's easy to cast the value to a smaller type, for example from **int16\_t** to **int8\_t** since **int8\_t** is the byte at the beginning of **int16\_t**

# QUIZ

**1. What is the min. number of assembly instructions needed to perform the following**

$$a = b + c + d - e;$$

- A. Single instruction**
- B. Two instructions**
- C. Three instructions**
- D. Four instructions**

**2.  $f = (g + h) - (i + j);$**

# QUIZ

## 1 solution: Break into multiple instructions

ADD r0, r1, r2 ; a = b + c

ADD r0, r0, r3 ; a = a + d

SUB r0, r0, r4 ; a = a - e

$$a = b + c + d - e;$$

## 2 solution:

ADD r0,r1,r2 ; f = g + h

ADD r5,r3,r4 ; temp = i + j

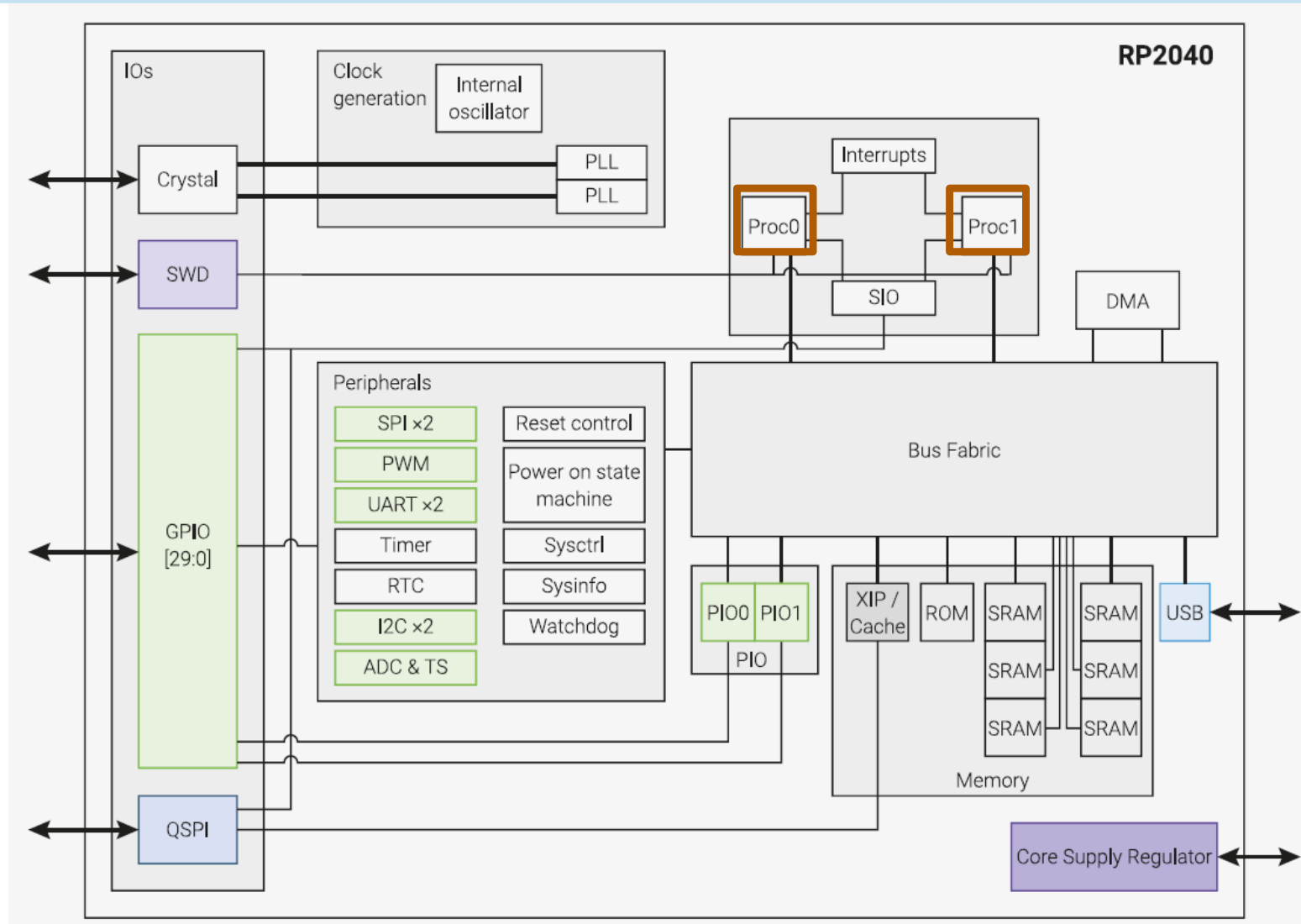
SUB r0,r0,r5 ; f =(g+h)-(i+j)

$$f = (g + h) - (i + j);$$



# **Arm –Cortex-M0+** **(ARMv6-M Profile)**

# Arm-Cortex-M0+ Cores



# Arm Cortex-M0+ (ARMv6-M Profile)

Processor	Arm Architecture	Core Architecture	Thumb®	Thumb®-2	Hardware Multiply	Hardware Divide	Saturated Math	DSP Extensions	Floating Point
Cortex-M0	Armv6-M	Von Neumann	Most	Subset	1 or 32 cycle	No	No	No	No
Cortex-M0+	Armv6-M	Von Neumann	Most	Subset	1 or 32 cycle	No	No	No	No
Cortex-M1	Armv6-M	Von Neumann	Most	Subset	3 or 33 cycle	No	No	No	No
Cortex-M3	Armv7-M	Harvard	Entire	Entire	1 cycle	Yes	Yes	No	No
Cortex-M4	Armv7E-M	Harvard	Entire	Entire	1 cycle	Yes	Yes	Yes	Optional

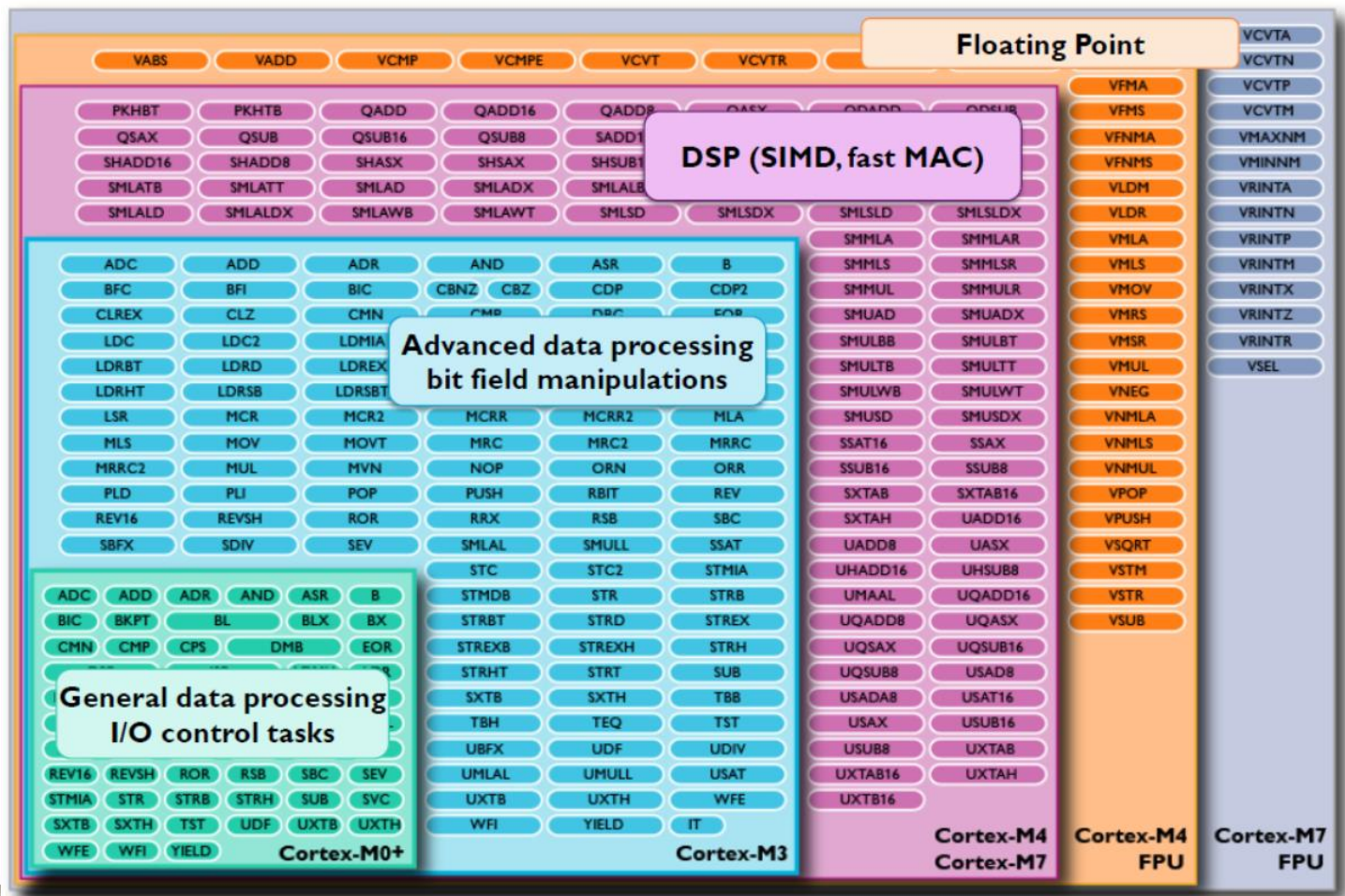
**ARMv7-M:** The microcontroller profile for systems supporting only the Thumb instruction set, and where overall size and deterministic operation for an implementation are more important than absolute performance.

**ARMv6-M** is a **subset** of **ARMv7-M**



# Cortex-M Instruction Set


- ARMv6-M supports the Thumb instruction set, including a small number of 32-bit instructions.
- Hence widely used in mobile devices, such as smartphones and tablets



# Cortex-M Instruction Set

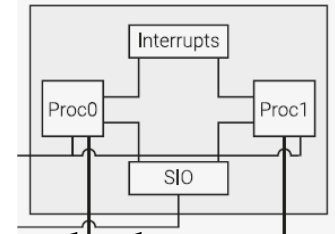
- The Thumb instruction set is a subset of the ARM instruction set architecture developed by ARM Holdings. It is designed to improve code density by using 16-bit instructions instead of the regular 32-bit ARM instructions. The Thumb instruction set allows for more efficient use of memory, which is particularly important in embedded systems with limited resources.
- Key features of the Thumb instruction set include:
- **16-bit Instructions:** Thumb instructions are half the size of regular ARM instructions (32 bits). This reduction in size results in more compact code, which is beneficial for systems with limited memory.
- **Code Density:** By using 16-bit instructions, the Thumb instruction set increases the code density, meaning that more instructions can fit into a given amount of memory.
- **Backward Compatibility:** Processors that support the Thumb instruction set are typically backward compatible with the regular ARM instruction set. This allows for a mix of Thumb and ARM instructions in the same program.

# Cortex-M Instruction Set

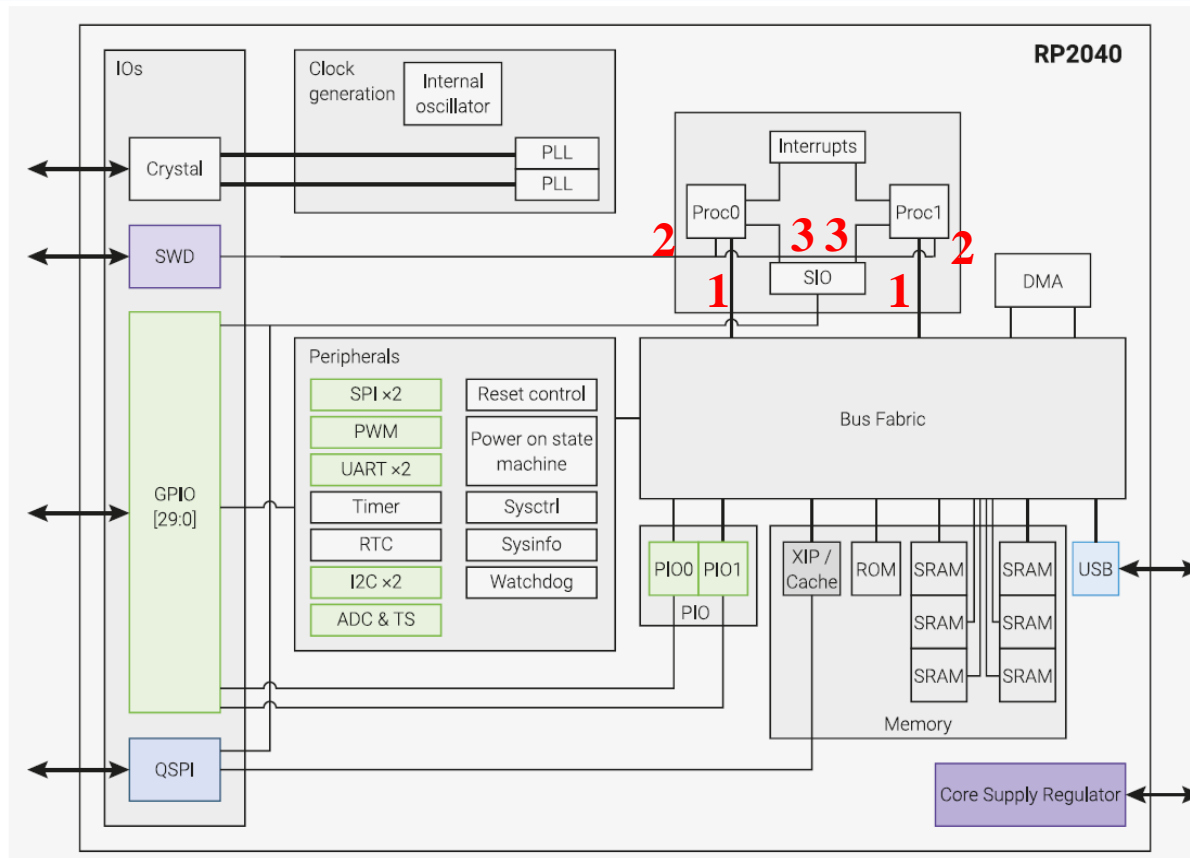
- **Reduced Power Consumption:** The smaller instructions also contribute to reduced power consumption since fetching and decoding smaller instructions require less energy.
  - The Thumb instruction set has been widely adopted in ARM Cortex-M microcontrollers, which are commonly used in embedded systems and microcontroller applications. It allows these systems to achieve better performance and efficiency in terms of code size and power consumption.
  - It's important to note that there are different versions of the Thumb instruction set, such as Thumb, Thumb-2, and Thumb-EE (Enhanced Efficiency). Thumb-2, for example, extends the Thumb instruction set with additional 32-bit instructions to provide a balance between code density and performance. The choice of Thumb or regular ARM instruction set depends on the specific requirements of the target application.
- 

# Arm-Cortex-M0+: Features and Benefits

- Tight integration of system peripherals reduces area and development costs.
- Thumb instruction set combines high code density with 32-bit performance.
- Support for single-cycle I/O access.
- Power control optimization of system components.
- Integrated sleep modes for low-power consumption.
- Fast code execution enables running the processor with a slower clock or increasing sleep mode time.
- Optimized code fetching for reduced flash and ROM power consumption.
- Hardware multiplier.
- Deterministic instruction cycle timing.
- Deterministic, high-performance interrupt handling for time-critical applications.
- Support for system level debug authentication.
- Serial Wire Debug (SWD) reduces the number of pins (5 wires) required for debugging.



# Arm-Cortex-M0+: External Interfaces



- The **three interfaces** provided in the processor for **external accesses** are:
  1. External AHB-Lite interface to bus fabric
  2. Debug Access Port (DAP)
  3. Single-cycle I/O Port to SIO peripherals

# Arm-Cortex-M0+: External Interfaces

- The AHB-Lite (Advanced High-Performance Bus Lite) is a simplified version of the ARM AMBA (Advanced Microcontroller Bus Architecture) AHB (Advanced High-Performance Bus) interface. AHB is a widely used on-chip bus protocol in ARM-based systems for connecting various components such as processors, memory, and peripherals.
- When discussing an "External AHB-Lite interface to bus fabric," it typically means connecting an AHB-Lite interface to the bus fabric, which is the interconnection network that allows different components on a chip to communicate with each other.
- Integrating a DAP with an AHB-Lite interface is common in embedded systems to enable debugging capabilities. The DAP can be connected to the AHB-Lite bus to access and control various components for debugging, such as reading and writing to registers, halting the processor, and inspecting memory.
- A "single-cycle I/O port to SIO (Serial Input/Output) peripherals" suggests a design feature in a digital system where the I/O port can communicate with SIO peripherals in a single clock cycle.

# Arm-Cortex-M0+: External Interfaces

## ➤ **I/O Port:**

- An I/O port, or Input/Output port, is a hardware interface used for communication between a digital system and external devices. It allows data to be transferred into or out of the system.

## ➤ **Single-Cycle:**

- "Single-cycle" indicates that the data transfer between the I/O port and the SIO peripherals can be completed within a single clock cycle. This design choice is often made to achieve high-speed communication.

## ➤ **SIO Peripherals:**

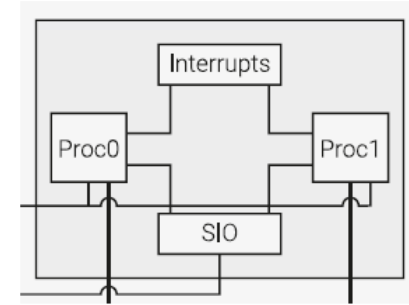
- SIO refers to Serial Input/Output peripherals. These peripherals typically communicate using serial protocols like UART (Universal Asynchronous Receiver/Transmitter), SPI (Serial Peripheral Interface), I2C (Inter-Integrated Circuit), or other serial communication standards.



# Arm-Cortex-M0+: Configurations

Each processor is configured with the following features:

- Architectural clock gating (for power saving)
- Little Endian bus access (code access is always Little endian)
- Data access is configurable (It is fixed as Little in RP2040)
- Four Breakpoints
- Debug support (via 2-wire debug pins SWD/SWCLK)
- 32-bit instruction fetch (to match 32-bit data bus)
- IOPORT (for low latency access to local peripherals (SIO))
- 26 interrupts
- 8 MPU regions
- All registers reset on power up



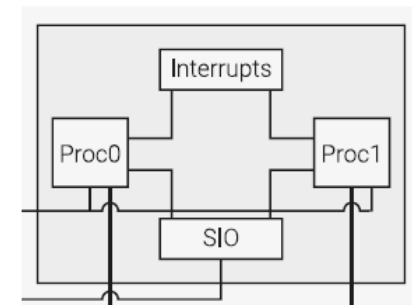
**MPU:** Memory Protection Unit

# Arm-Cortex-M0+: Configurations ... contd.

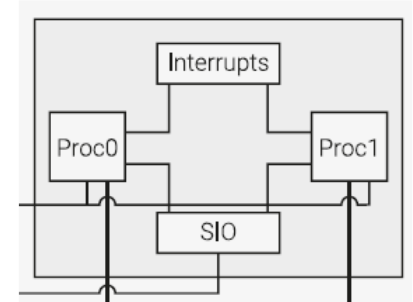
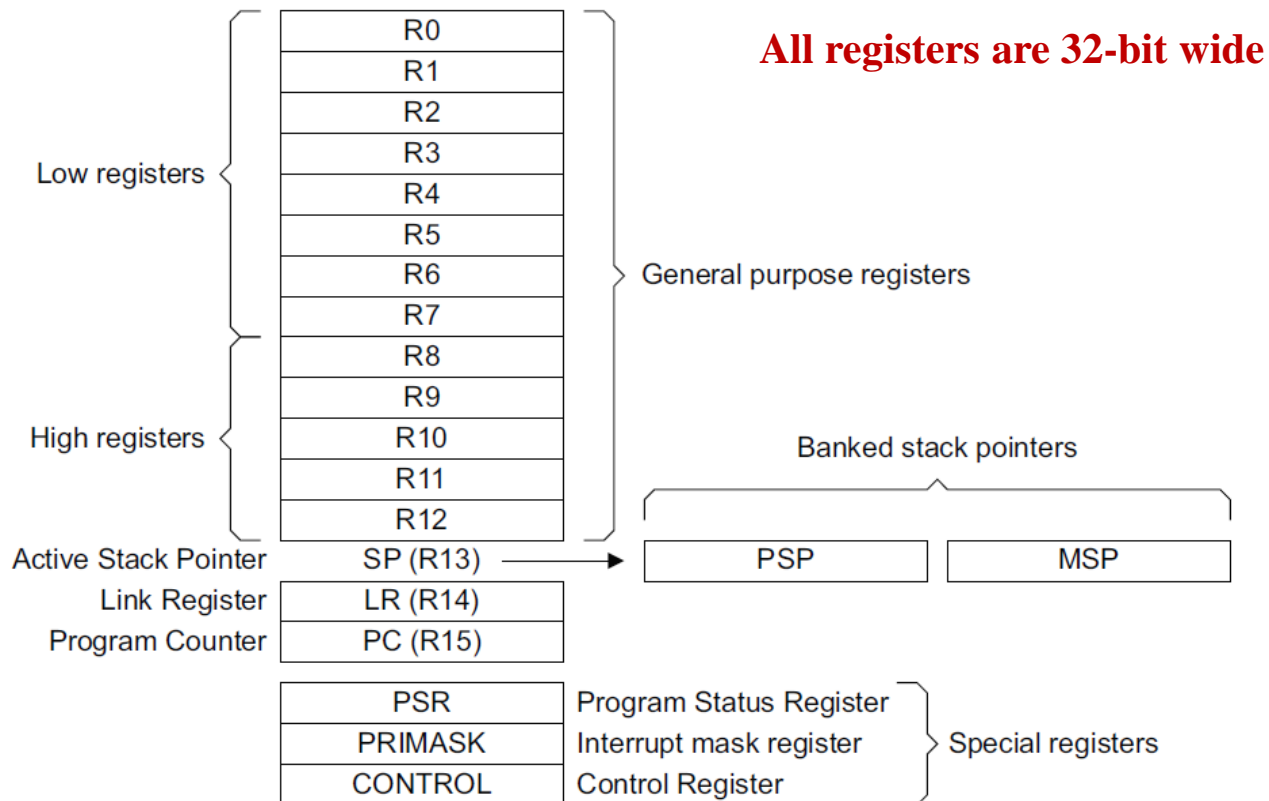
Each processor is configured with the following features:

- Fast multiplier (MULS 32x32 single cycle)
- SysTick timer
- Vector Table Offset Register (VTOR)
- 34 WIC (Wake-up Interrupt Controller) lines (32 IRQ and NMI)
- DAP feature: Halt event support
- DAP feature: SerialWire debug interface (protocol 2 with multidrop support)
- DAP feature: Micro Trace Buffer (MTB) is not implemented

Each M0+ core has its own interrupt controller which can individually mask out interrupt sources as required. The same interrupts are routed to both M0+ cores.



# Arm-Cortex-M0+: Register Set



**PSP:** Process Stack Pointer

**MSP:** Main Stack Pointer (default on reset)

**Ref:**Ref4\_ARM-Cortex-M0+ Devices Generic User Guide, Section 2.1.3 Core Registers

# Arm-Cortex-M0+: Register Set

## General-purpose registers

R0-R12 are 32-bit general-purpose registers for data operations.

## Stack Pointer

The stack pointer (SP) is register R13.

The processor uses a full descending stack, meaning the Stack Pointer holds the address of the last stacked item in memory. When the processor pushes a new item onto the stack, it decrements the Stack Pointer and then writes the item to the new memory location.

## Link Register

The Link Register (LR) is register R14. It stores the return information for subroutines, function calls, and exceptions. On reset, the processor sets the LR value to 0xFFFFFFFF.

## Program Counter

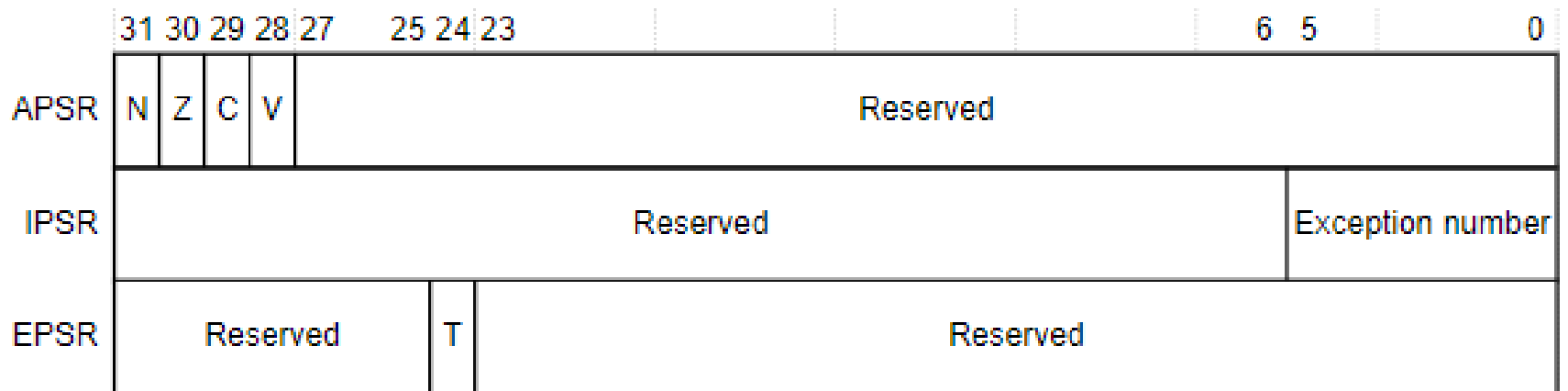
The Program Counter (PC) is register R15. It contains the current program address. On reset, the processor loads the PC with the value of the reset vector defined in the vector table.

## Arm-Cortex-M0+: Register Set

## Combined Program Status Register

The Combined Program Status Register (xPSR) consists of the Application Program Status Register (APSR), Interrupt Program Status Register (IPSR), and Execution Program Status Register (EPSR).

These registers are mutually exclusive bit fields in the 32-bit PSR. The bit assignments are as follows:



# Arm-Cortex-M0+: Register Set

## **Priority Mask Register**

The PRIMASK register is intended to disable interrupts by preventing activation of all exceptions with configurable priority in the current Security state.

## **CONTROL register**

The CONTROL register controls the stack that is used, the privilege level for software execution when the core is in Thread mode and indicates whether the FPU state is active.

# Summary

- Endianness (Big Vs Little)
- Arm Cortex-M0+
  - ARMv6-M Architecture Profile
  - Features
  - External Interfaces
  - Configurations
  - Register Set