Embedded Systems

Lecture 10
Computational Units

© Michele Magno
D-ITET Center for Project-Based Learning

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

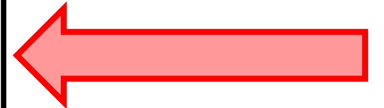
Where We are

Hardware-Software

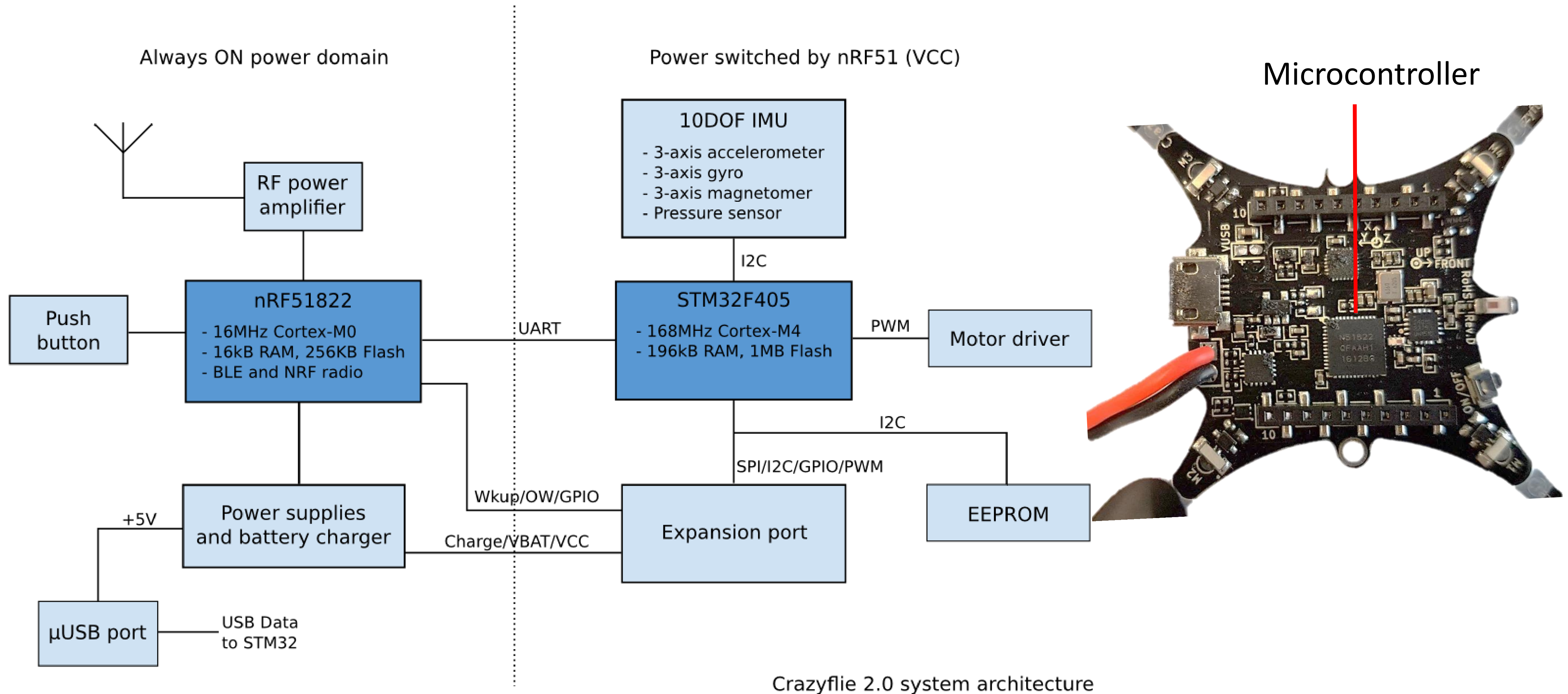
Real-Time

Special

0. Introduction into Embedded Systems
1. Hardware-Software Architecture and Software Development
2. Hardware-Software Interfaces – (GPIO), Interrupt, and Clock
3. Hardware-Software Interfaces - Serial Interfaces
4. No Lecture
5. Hardware-Software Interfaces - Timer, PWM and ADC
6. Real-Time Systems
7. Dynamic Scheduling and Real-Time Operating Systems
8. Deterministic Scheduling
9. Low Power Design
10. Computational Units
11. Implementation Strategies & Project Kick-off
12. Project Q&A

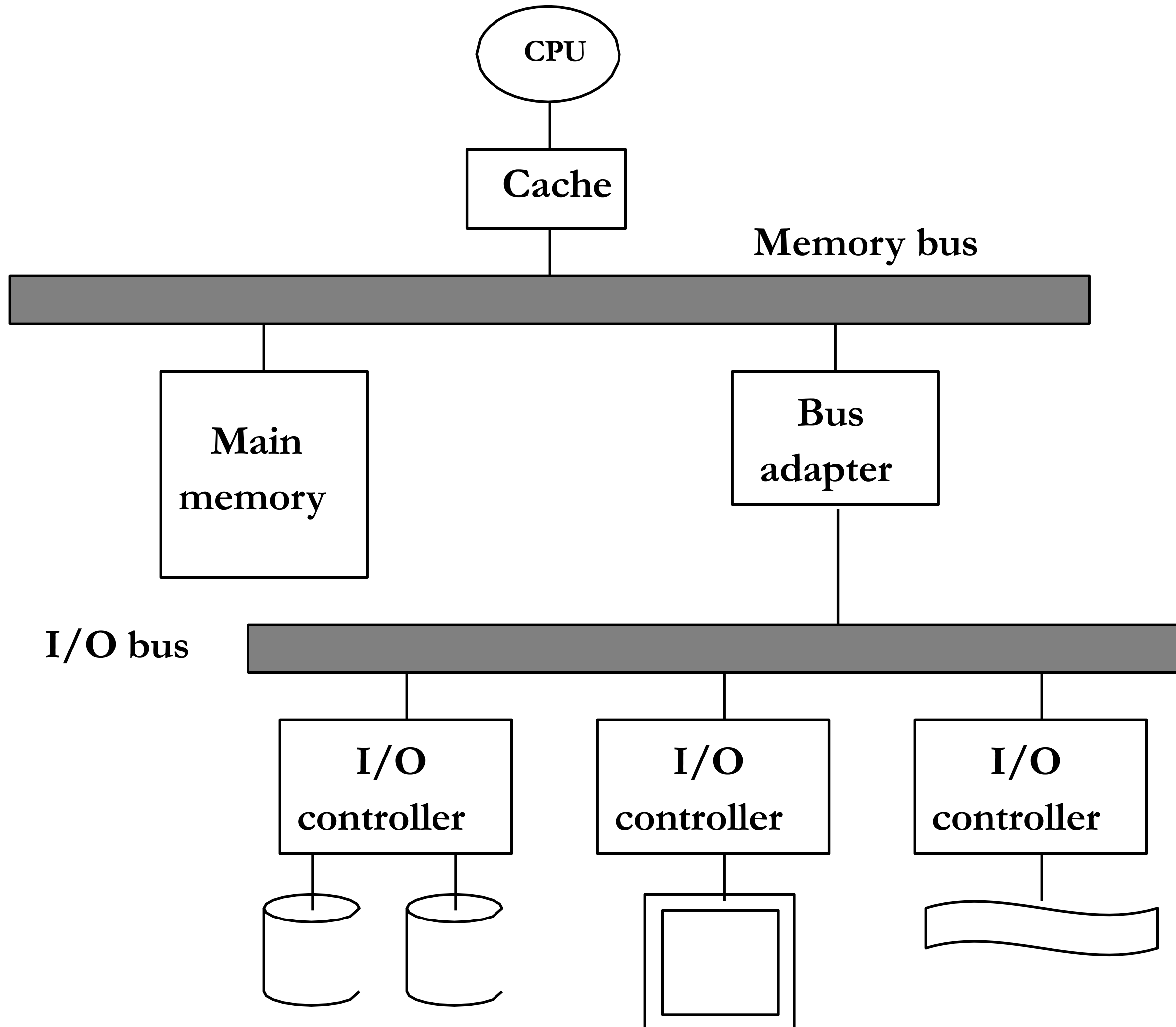


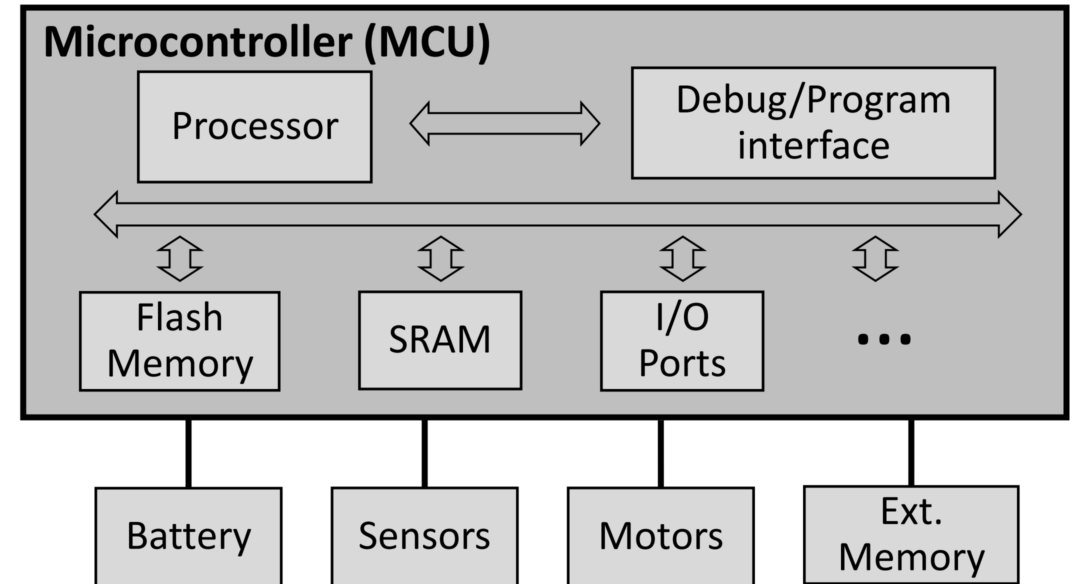
Let’s go back to our Drone example



Crazyflie 2.0 system architecture

I/O Controllers & Devices



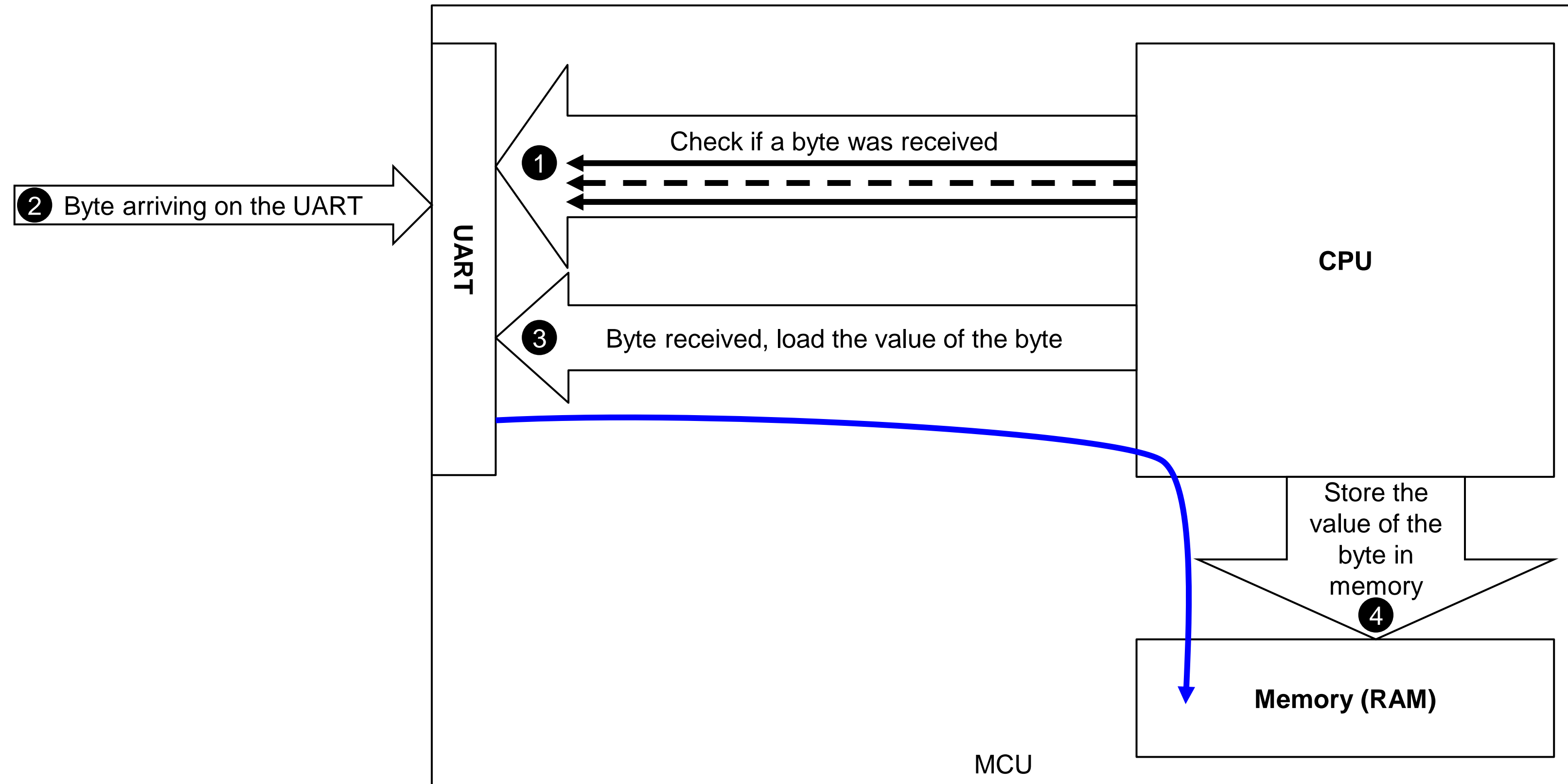


Example with UART communication

Reminder : Polling vs Interrupt

UART Rx message using Polling

Read one byte

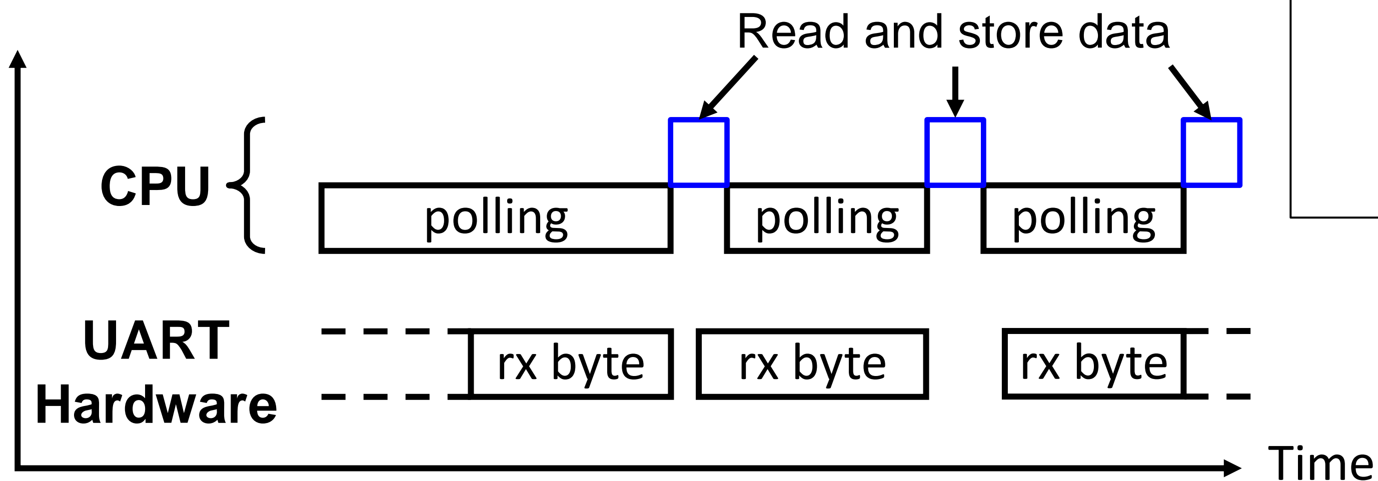


Reminder : Polling vs Interrupt

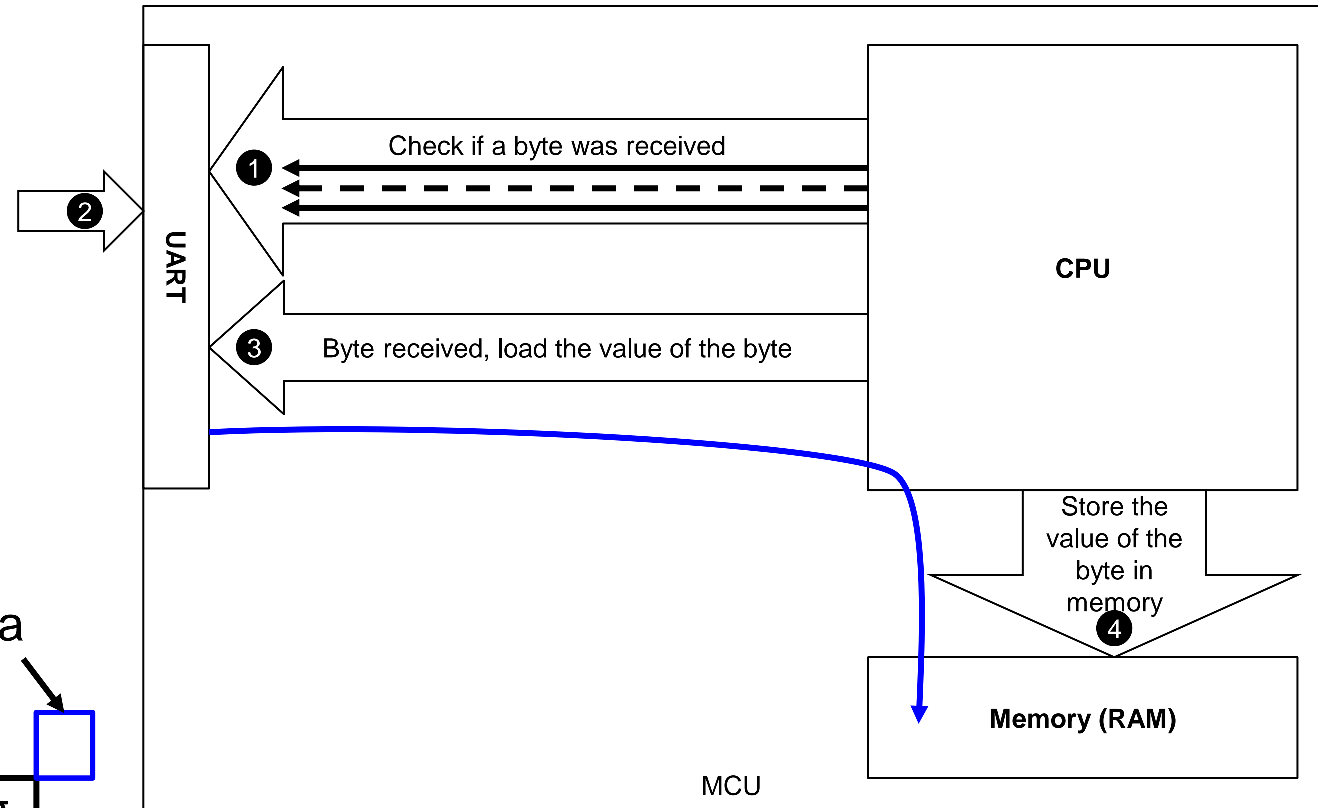
UART Rx message using Polling

Read multiple bytes:

- Repeat the same steps for the expected number of bytes to be received
- Increment the memory address where the byte should be stored



Receiving three UART bytes – CPU and UART Activity

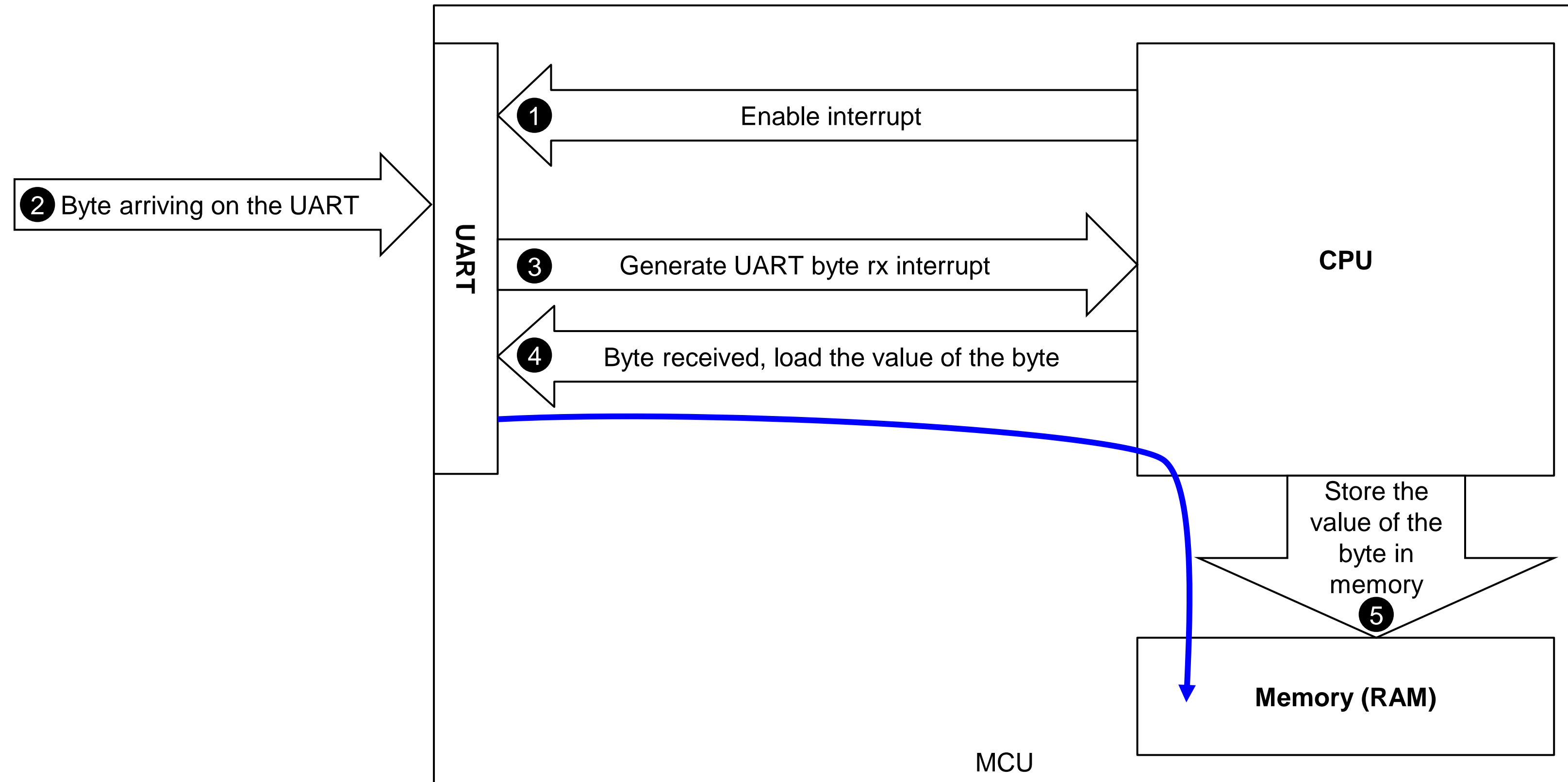


→ 100% of CPU time used for UART reception

Reminder : Polling vs Interrupt

UART Rx message using Interrupt

Read one byte

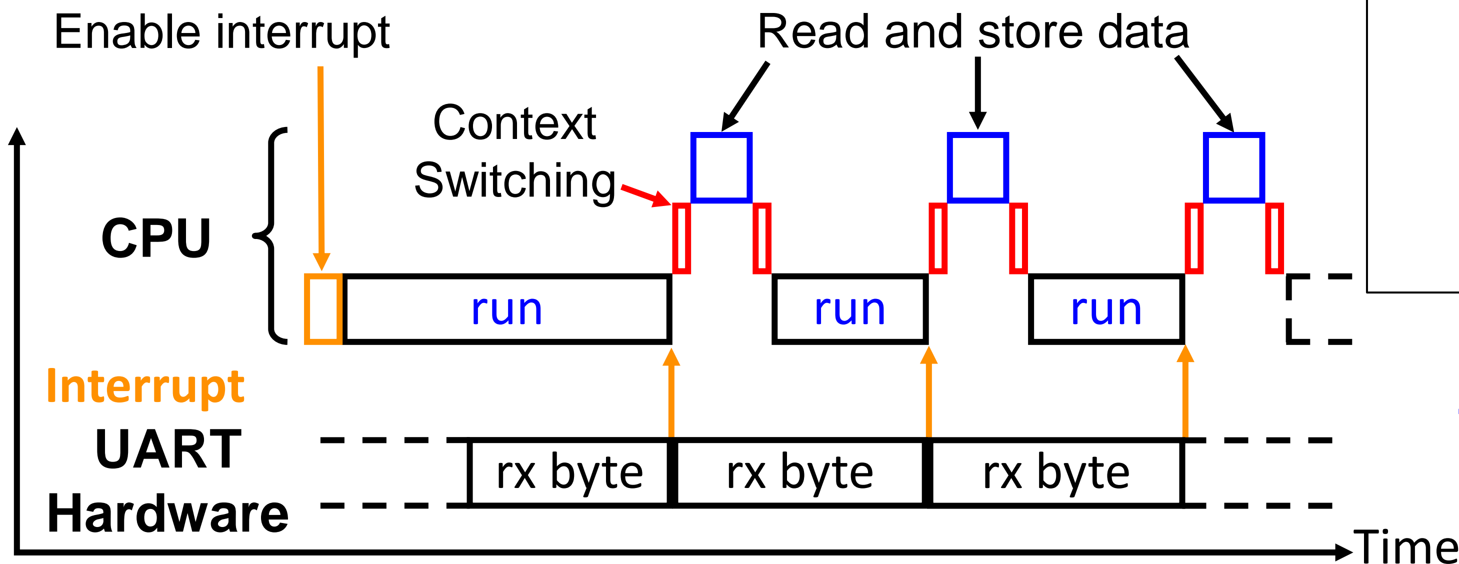


Reminder : Polling vs Interrupt

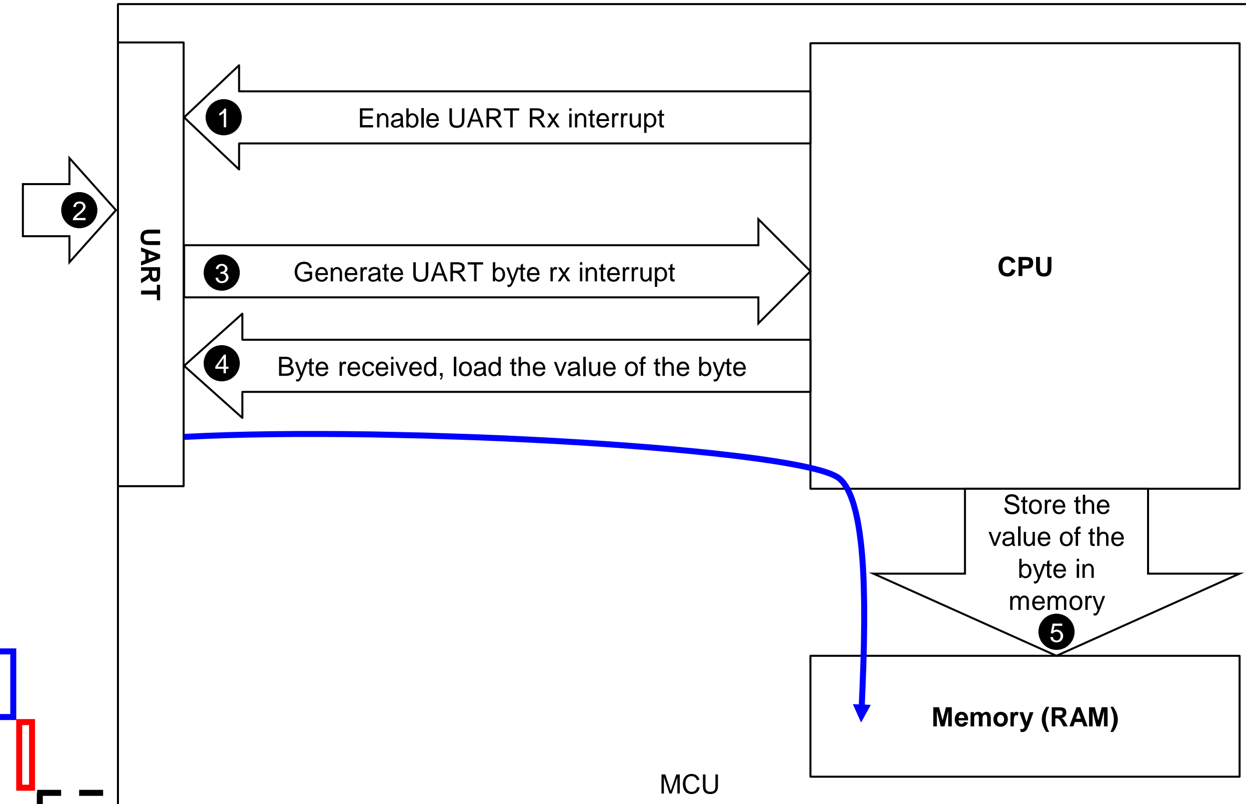
UART Rx message using Interrupts

Read multiple bytes:

- Repeat the same steps (2-5) for the expected number of bytes to be received
- Increment the memory address where the byte should be stored



Receiving three UART bytes – CPU and UART Activity



→ **CPU used for UART Reception only when a byte arrives on the port**

Interrupt Approach – Best solution?

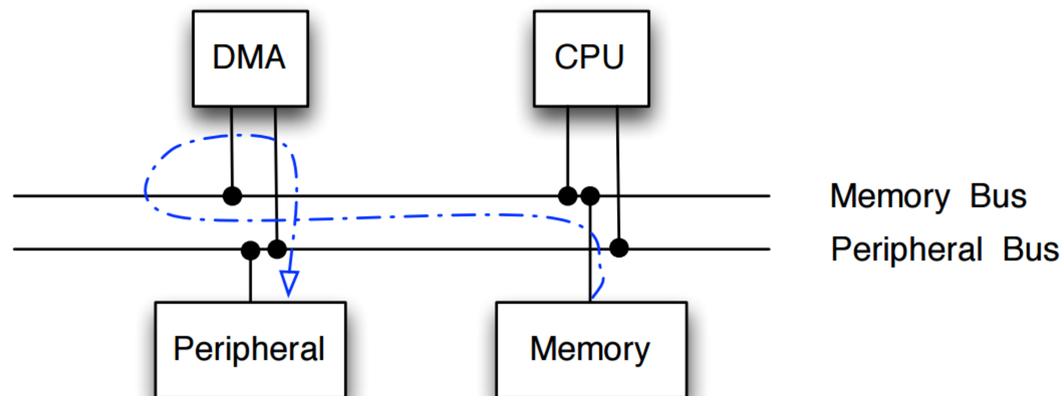
- CPU has **to issue individual commands** – through the OS – to read every sector from disk
- Data transfer is very slow for slow peripherals (~100 μs, vs ns for CPU clock speed)
- The interrupt mechanism can be time-consuming because it involves multiple steps: detecting the interrupt (IRQ), saving the CPU's current state (program counter and registers) to the stack, jumping to the Interrupt Service Routine (ISR), executing the ISR, and finally restoring the CPU's state to resume normal execution
- Time taken by interrupt mechanism makes it difficult to synchronize and quantify (i.e. Nested interrupts)

Can we improve the efficiency? Direct Memory Access (DMA)

DMA Characteristics

Direct memory access (DMA) is used to provide **high-speed data transfer between peripherals and memory** as well as **memory to memory**.

- Data can be quickly moved by DMA without any CPU actions.
- **It keeps CPU resources free for other operations.**



DATA TRANSFERS:

- **With CPU:**
  - ldr
  - str
  - …
  - ldr
  - Str
- **With DMA:**
  - Give the start command
  - Wait the interrupt when transfer is completed

"Direct Memory Access (DMA) is a technique that allows the transfer of bulk data from peripheral to memory, or vice versa, without the CPU's intervention" [1]

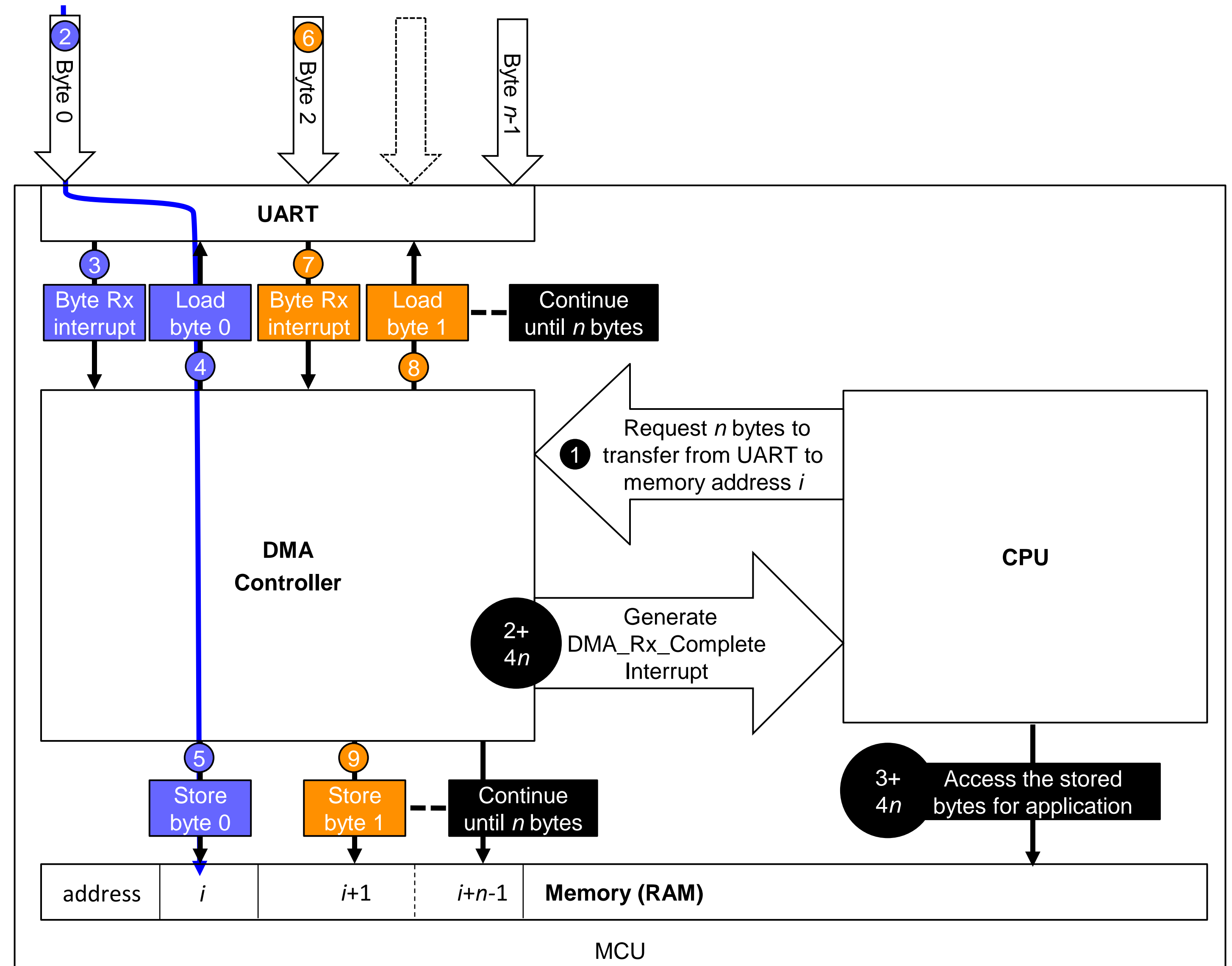
1 - Embedded Systems with ARM Cortex-M Microcontrollers in assembly Language and C, Fourth Edition – Dr. Yifeng Zhu

DMA

UART Rx message using DMA

Read ***n*** bytes

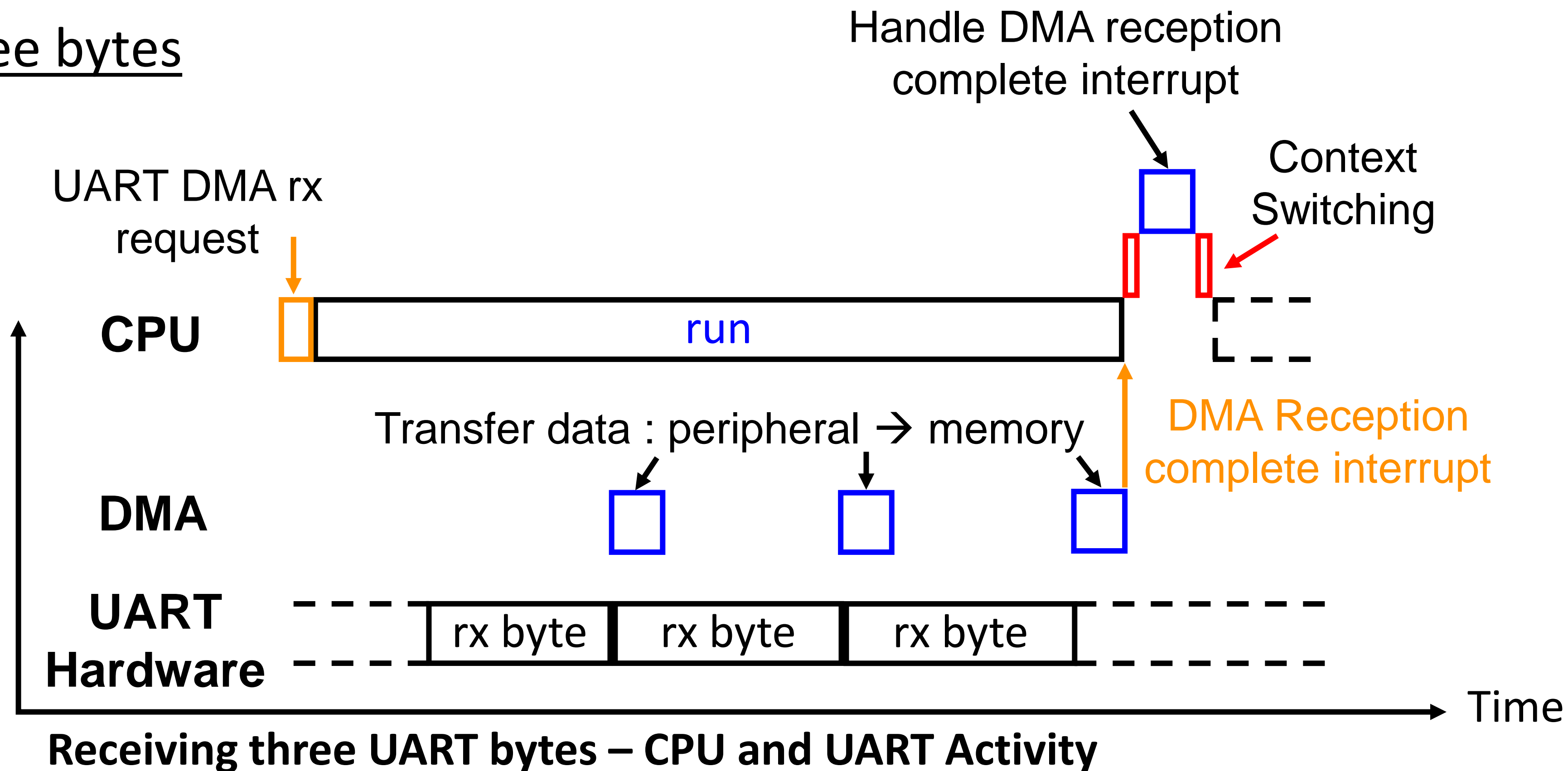
- CPU initiates a UART reception request of *n* bytes to the DMA
- The DMA controller is triggered when a byte is received over UART and stores it to the Memory at address *i + byte_nb*
- The process is repeated *n* times, and the DMA controller generates an interrupt to notify the CPU that the reception is complete, and the data is available in memory
- The CPU can then access the data and use it for its application



DMA

UART Rx message using DMA

Read three bytes



Receiving three UART bytes – CPU and UART Activity

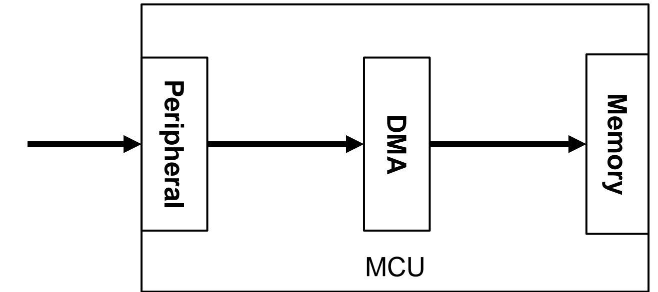
→ **CPU used only when a full frame is available received and does not have to load data from peripheral and store it to memory**

Polling vs Interrupt vs DMA

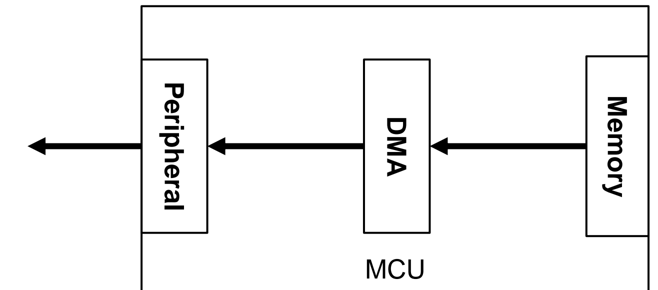
Feature	Polling	Interrupt	DMA
CPU Usage	High (CPU constantly checks status)	Moderate (CPU handles only on interrupt)	Low (CPU only involved after all data is transferred)
Implementation Complexity	Simple (basic status check loop)	Moderate (interrupt setup required)	Complex (DMA and interrupt setup required)
Suitability for Large Data Transfers	Poor (high CPU overhead)	Medium (many interrupts required)	Excellent (transfer complete with minimal CPU involvement)
Transfer speed	Slow/Medium	Fast	Fastest (dedicated hardware)
Interrupt Frequency	None	Every byte	Only at transfer completion
Power Consumption	High, CPU always active	Medium, CPU can be put in a low power mode while waiting for a new byte	Low, CPU can be put in a low power mode while waiting for the whole frame transfer

DMA Features

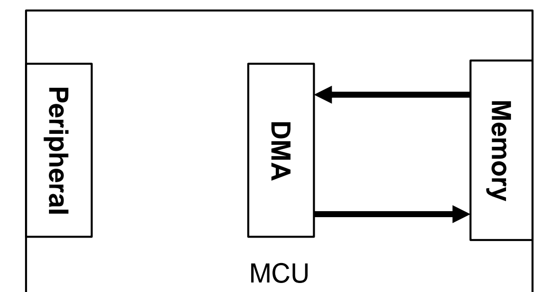
- Peripheral to Memory
  - Read UART/I2C/SPI frames
    → Previous example
  - Sampling data from an ADC



- Memory to Peripheral
  - Write UART/I2C/SPI frames
    → When writing data to a communication peripheral, the three methods (polling, interrupt, and DMA) are applicable:
    - Write the byte to the peripheral
    - Wait until it is sent (as the CPU clock speed is usually way faster than the communication protocol one)
    - Send the next byte etc.
  - Transferring data to DAC for waveform generation

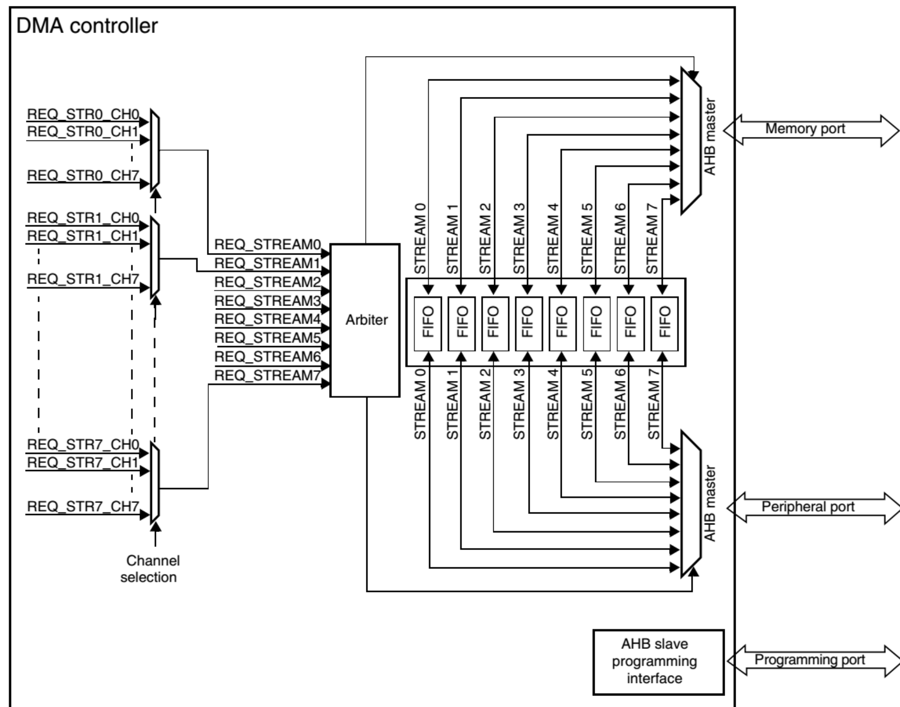


- Memory to Memory
  - Moving large blocks of data in memory
    → Can be used instead of `memcpy()`



DMA in STM32

We have two DMA controllers have 16 streams in total (8 for DMA1 and 8 for DMA2), each stream has 8 different channels **each dedicated to managing memory access requests** from one peripheral. It has an arbiter for handling the priority between DMA requests.



16 streams for dedicated hardware DMA requests, software trigger is also supported. The **configuration is done by software**.

4 priority levels (very high, high, medium, low) or hardware in case of equality (request 1 has priority over request 2, etc.)

Independent source and destination transfer size (byte, half word, word), Source/destination addresses must be aligned on the data size.

5 event flags (DMA Half Transfer, DMA Transfer complete, DMA Transfer Error, DMA FIFO Error, Direct Mode Error) logically ORed together in a single interrupt request for each stream

DMA Memory to memory

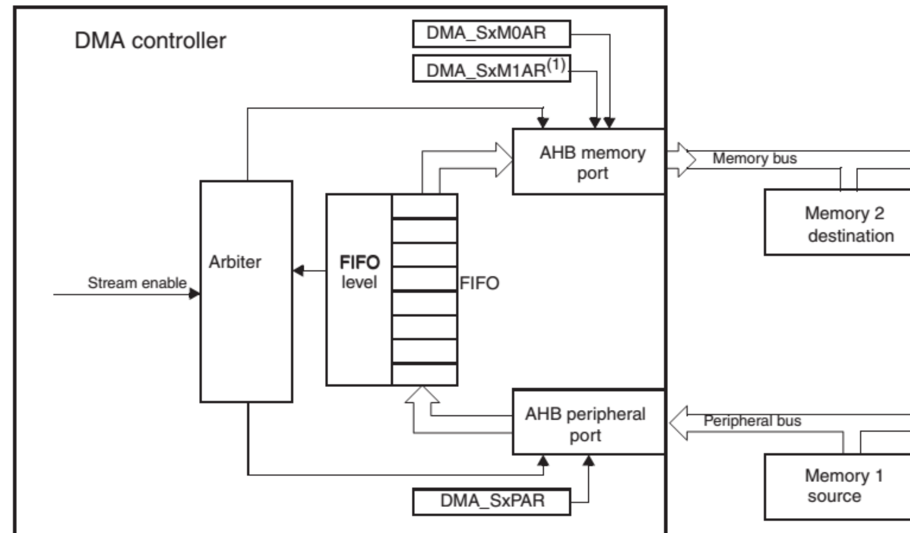
DMA transfer modes: **memory-to-memory**, **peripheral-to-memory**, **memory-to-peripheral**.

- **Memory-to-Memory**:

  Transfer data from one memory buffer to another

  Circular and Direct modes are not allowed.

  Only the DMA2 controller is able to perform memory-to-memory transfers

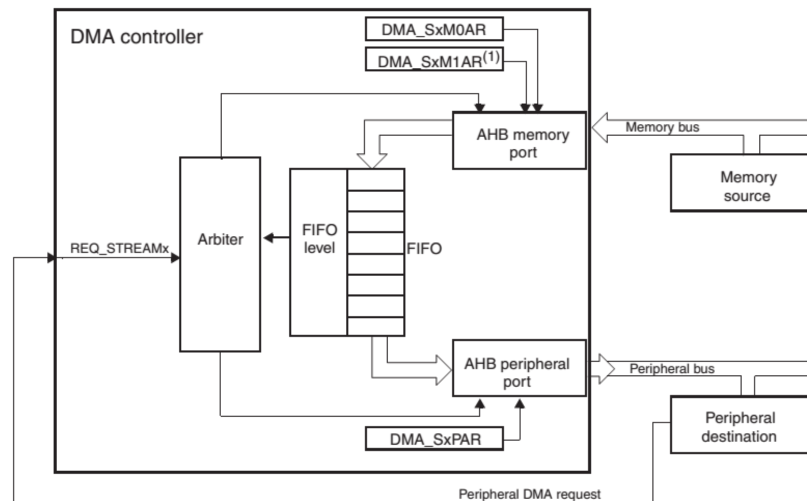


DMA Memory and peripherals

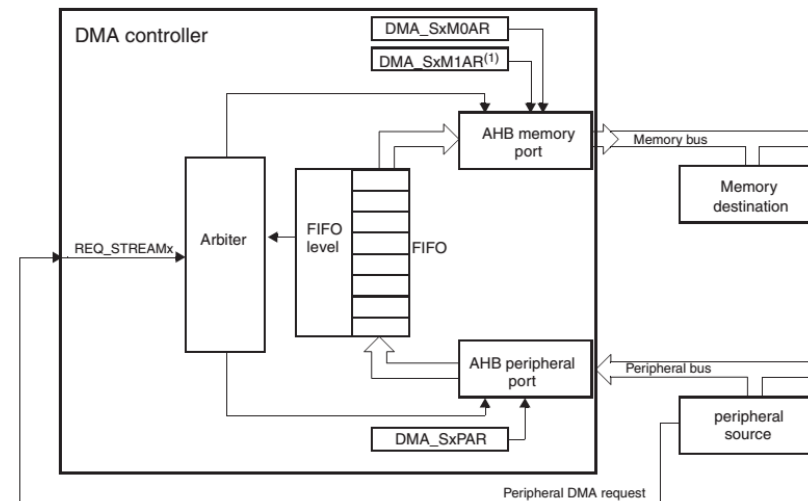
Memory-to-Peripheral or Peripheral-to-Memory:

When the threshold level of the FIFO is reached, its content is drained and stored into the destination.

Direct Mode: the threshold level of the FIFO is not used. After each single data transfer from the peripheral to the FIFO, the data is immediately drained and stored into the destination.



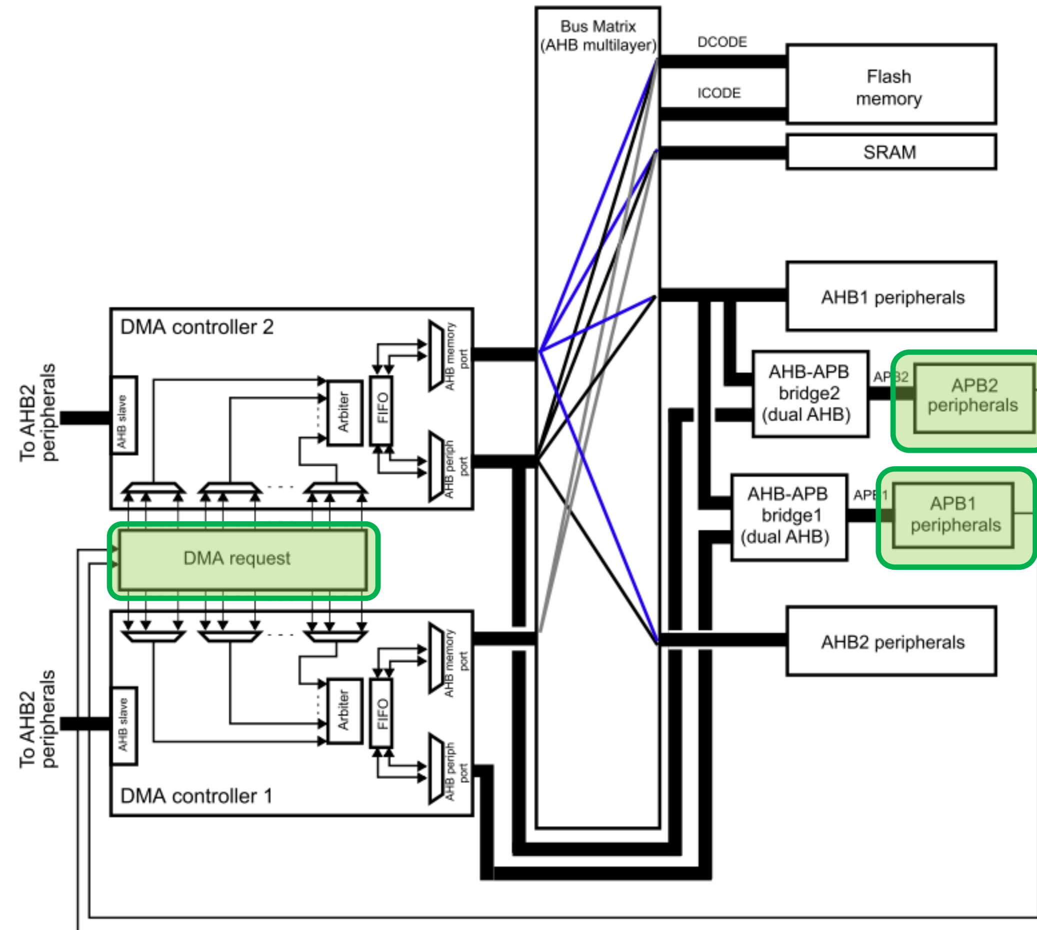
Memory-to-Peripheral



Peripheral-to-Memory

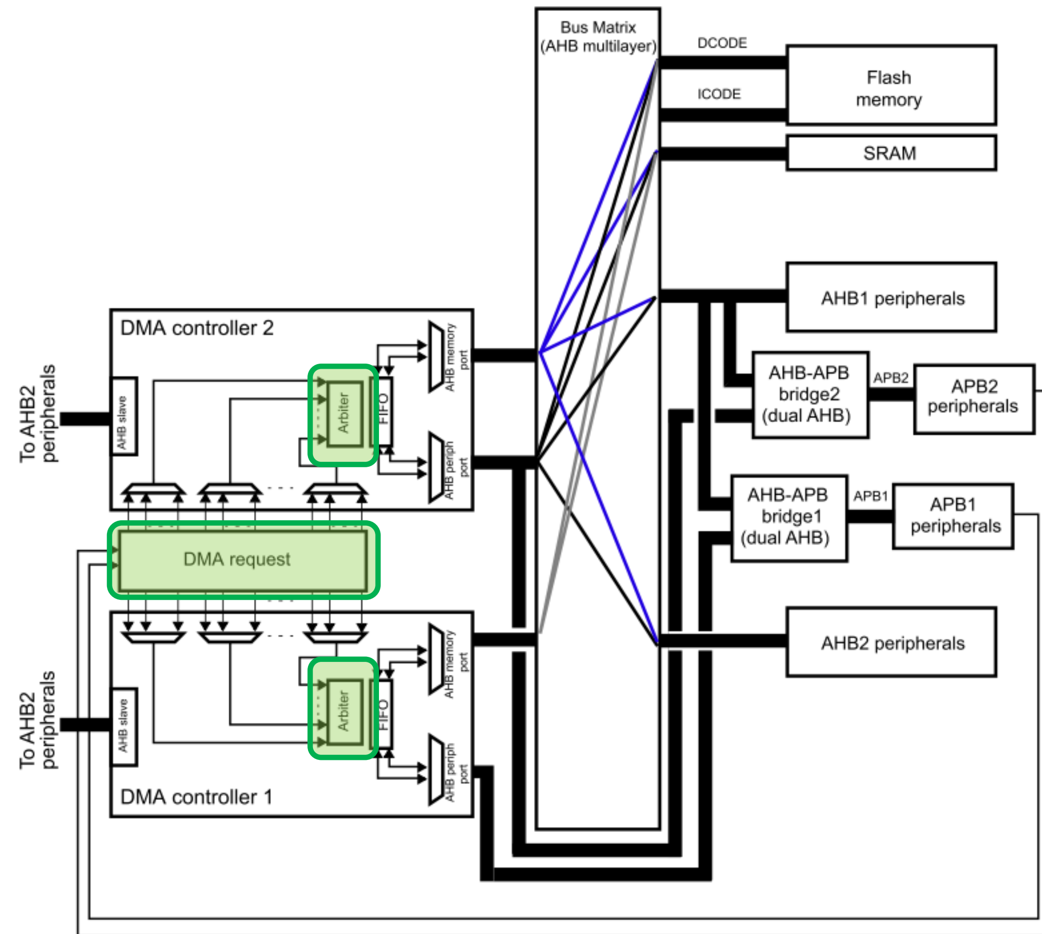
DMA Block scheme (LAB)

- After an event, the **peripheral sends a request signal to the DMA Controller**.
- 8 channels are multiplexed In one stream.
- The DMA controller serves the request depending on the streams priorities.
- The DMA channels can also work without being triggered by a request from a peripheral. This mode is called **Memory to Memory mode**.



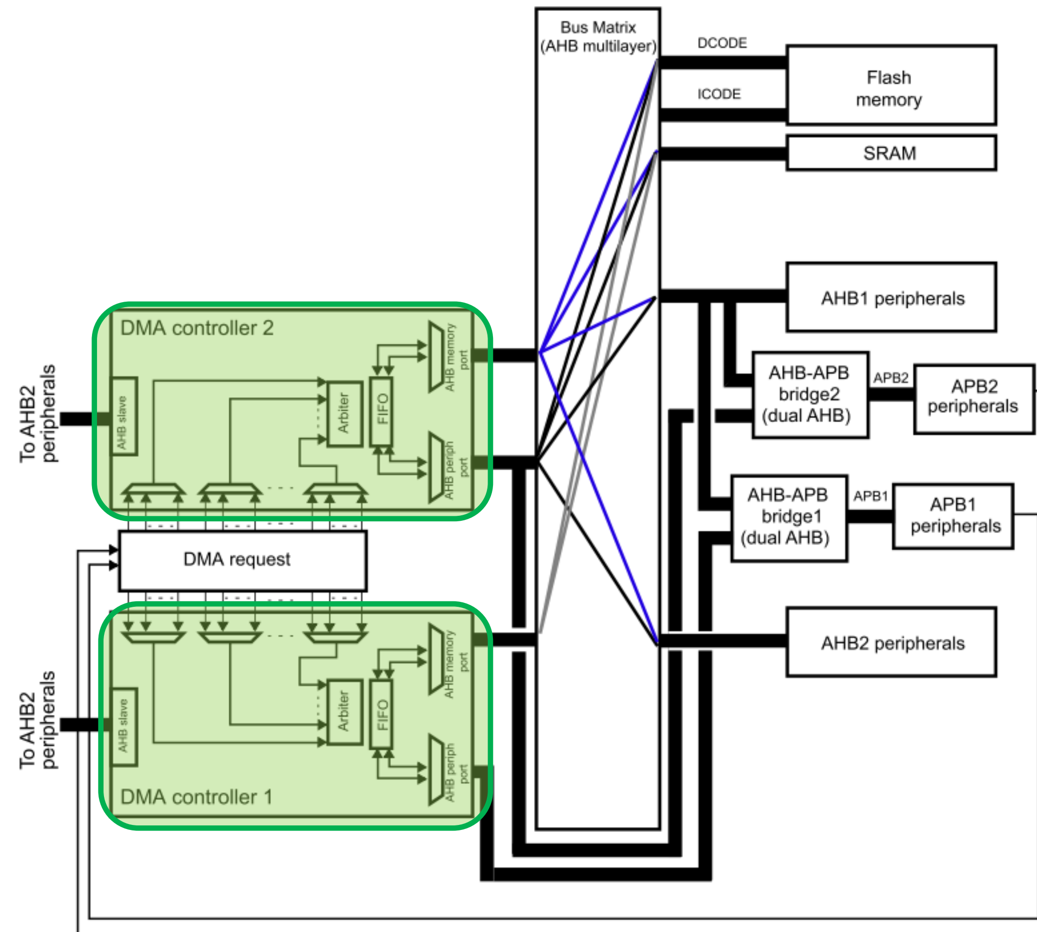
DMA Block scheme (LAB)

- DMA requests are collected and channel multiplexed.
- The arbiter **manages the stream requests** based on their priority and launches the peripheral/memory access sequences.
- The priorities are managed in two stages:
  - **Software** (defined priority)
  - **Hardware** (if 2 requests have the same software priority level, the channel with the lowest number will get priority versus the channel with the highest number.)



DMA Block scheme (LAB)

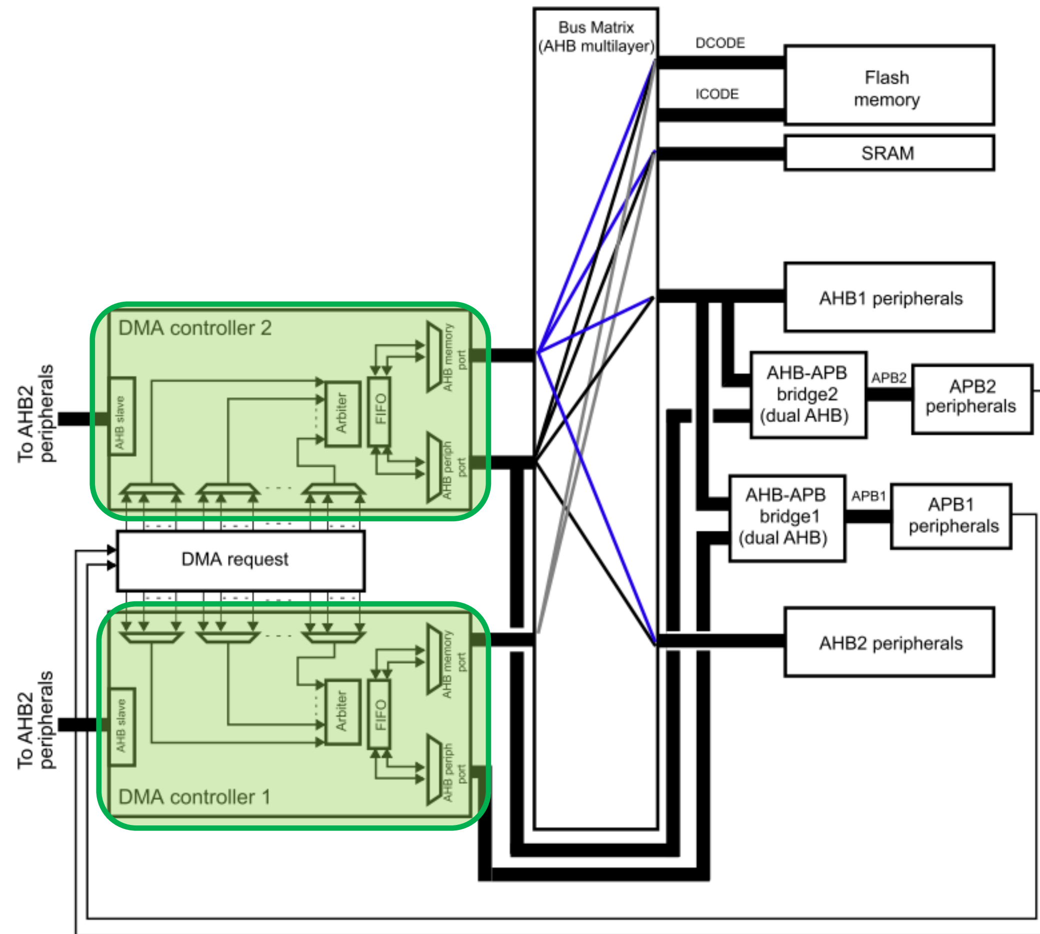
- Each channel can handle DMA transfers **between a peripheral register** located at a fixed address **and a memory address**.
- Peripheral and memory pointers can optionally be **automatically post-incremented** after each transaction.
- If incremented mode is enabled, the **address of the next transfer will be the address of the previous one incremented by 1, 2 or 4** depending on the chosen data size.



DMA Block scheme (LAB)

- **Noncircular mode**: no DMA request is served after the last transfer (that is once the number of data items to be transferred has reached zero).

- **Circular mode**: handle **circular buffers and continuous data** flows (e.g. ADC scan mode). When circular mode is activated, the number of data to be transferred is automatically reloaded with the initial value programmed during the channel configuration phase, and the DMA requests continue to be served.



DMA Channels

- Each Stream is multiplexed up to eight channel, only one channel at a time can be active
- It is not possible to activate two channels on the same stream
- Is possible to perform transfers from all 8 streams

Table 28. DMA1 request mapping (STM32F401xB/C and STM32F401xD/E)

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
Channel 1	I2C1_RX	I2C3_RX				I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
Channel 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4						USART2_RX	USART2_TX	
Channel 5			TIM3_CH4 TIM3_UP		TIM3_CH1 TIM3_TRIG	TIM3_CH2		TIM3_CH3
Channel 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2	I2C3_TX	TIM5_UP	
Channel 7			I2C2_RX	I2C2_RX				I2C2_TX

Instruction Set Architecture

Instruction Set Architecture

An *Instruction Set Architecture (ISA)* is part of the abstract model of a MCU that defines how the CPU is controlled by the software.

The ISA acts as an *interface between the hardware and the software*, specifying both what the processor is capable to do as well as how it is done.

Examples (ARM instructions):

```
ADD{<cond>}{S} <Rd>, <Rn>, <shifter_operator> // Add Rn and <shifter_operator>
AND{<cond>}{S} <Rd>, <Rn>, <shifter_operator> // Bitwise and Rn and <shifter_operator>
LDR{<cond>}    <Rd>, <addressing_operator>    // loads word from mem. and writes to Rd
MOV{<cond>}{S} <Rd>, <shifter_operand>        // moves value to Rd
MUL{<cond>}{S} <Rd>, <Rm>, <Rs>               // multiply signed/unsigned variables
```

Different CPU architectures have different ISA

Different Instruction Set Architectures

Complex Instruction Set Computer (CISC)

- Has many specialized instructions, some of which may only be rarely used in practical programs. E.g., Intel x86

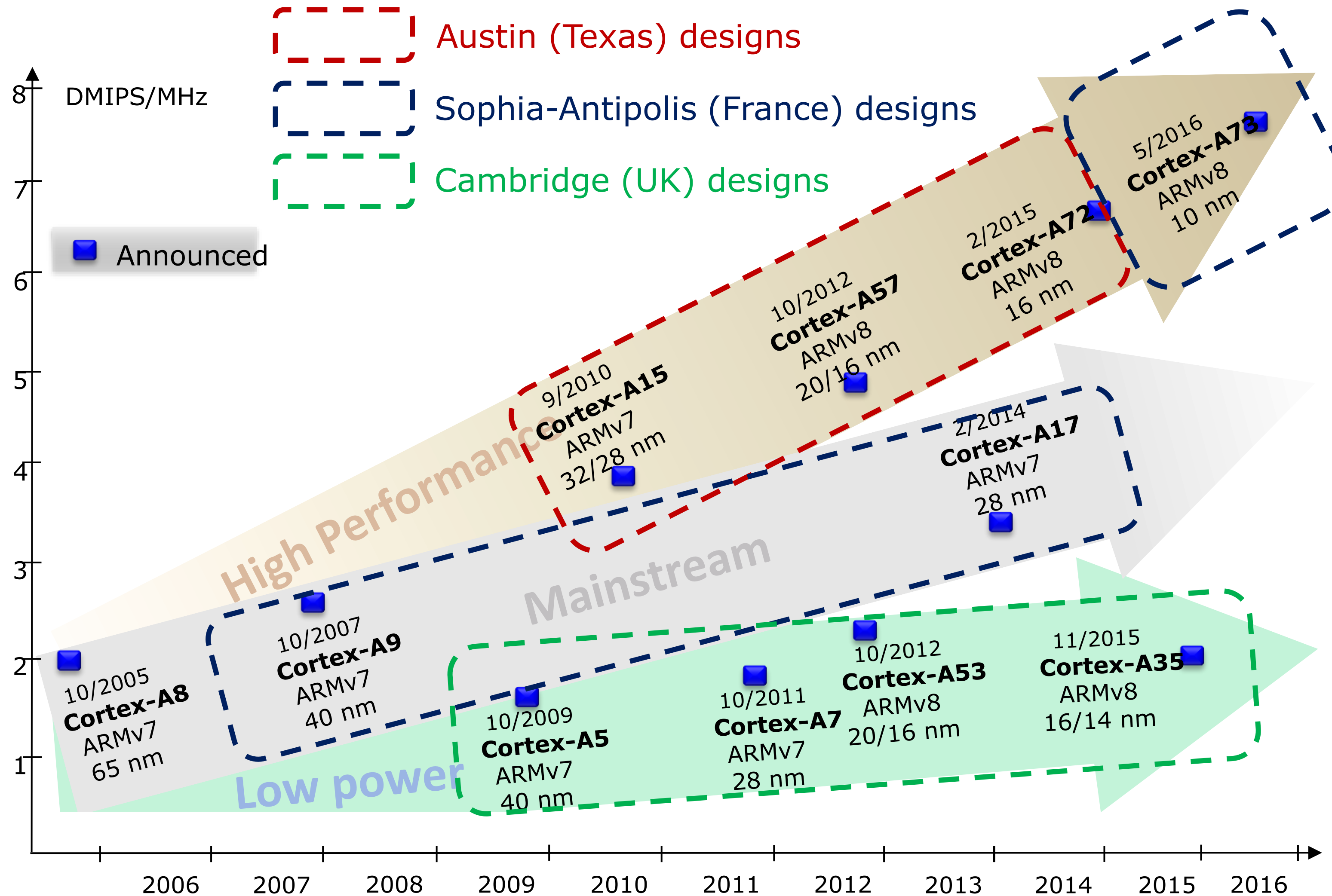
Reduced Instruction set Computer (RISC)

- Simplifies the processor *by efficiently implementing* only the instructions that are frequently used in programs, while the less common operations are implemented as subroutines. E.g., *Arm, RISC-V*

Example: For performing an ADD operation

- $C = A + B$
- CISC may only need one instruction: *add a, b, c*
- RISC may **need 4 instructions**:
  - *load reg*1, a   *load reg*2, b   *add reg*3, *reg*1, *reg*2   *store c, reg3*

ARM is The Leading ISA for Embedded Systems



Cortex-M4 Processor Features – Example

- Cortex-M4 processor is designed to meet the challenges of low dynamic power constraints while retaining light footprints
- ISA M4 will run with the *same instructions per task*
- However, we can get better power-efficient processor thanks to the technology
  - 180 nm process –157 µW/MHz
  - 90 nm process – 33 µW/MHz
  - 40 nm process – 8 µW/MHz

ARM Cortex-M4 Implementation Data			
Process	180ULL (7-track, typical 1.8v, 25C)	90LP (7-track, typical 1.2v, 25C)	40G 9-track, typical 0.9v, 25C)
Dynamic Power	**157 µW/MHz**	**33 µW/MHz**	**8 µW/MHz**
Floorplanned Area	0.56 mm^2	0.17 mm^2	0.04 mm^2

Embedded Systems – Performance Metrics

Performance Metrics

Performance metrics are mensurable features of the system's performance and are used to compare different systems. They often represent conflicting requirements.

Performance perspective; Given a collection of systems, which one has the:

- Best performance?
- Least Cost?
- Best performance/cost ratio?

Lifetime perspective; Given the same of systems, which one has the:

- Best performance?
- Least Cost?
- Best performance/cost ratio?

Performance Metrics

Performance metrics are mensurable features of the system's performance and are used to compare different systems:

- Power Consumption
- CPU clock speed
- Data throughput
- System Response time
- …and more

The goal is to *understand what factors in the architecture contribute to overall system performance* and the relative importance (and cost) of these factors

We need *different performance metrics* as well as a *different set of applications* to benchmark embedded systems.

CPU Execution Time

The *CPU execution time* is defined as the time a CPU needs to complete a specific task. It does not include waiting times (e.g. for interrupts).

$$CPU\ execution\ time = Number\ of\ CPU\ clock\ cycles * t_{clock\ cycle}$$

$$= \frac{Number\ of\ CPU\ clock\ cycles}{f_{clock}}$$

We can *improve the performance* by:

- Reducing the time of a clock cycle (increasing the clock frequency)
- Reducing the number of clock cycles for a specific task

How can We Improve The Computational Performance?

$$CPU\ performance = \frac{1}{execution\ time} = \frac{1}{\text{Number of CPU clock cycles} * \text{t}_{\text{clock cycle}}}$$

	Instruction Count	Clock Cycle
Algorithm	X	
Compiler	X	
ISA	X	X
Core organization		X
Technology		X
Hardware Accelerators	X	

How can We Improve The Computational Performance?

$$CPU\ performance = \frac{1}{execution\ time} = \frac{1}{\text{Number of CPU clock cycles} * \text{t}_{\text{clock cycle}}}$$

- *Algorithm optimizations* reduce the amount of needed instructions
- An *efficient compiler* reduces the amount if instructions by an optimal Instruction set utilization.
- An *extensive ISA* provides the compiler a larger instruction set.
- An *optimized hardware core* organization allows to reduce $t_{clock\ cycle}$ and improves the CPI – Cycles per Instruction
- A better hardware *fabrication technology* allows shorter $t_{clock\ cycle}$
- Specialized *hardware accelerators* reduce the instruction count and improves the CPI

Computational Performance

To maximize the computational performance (operation per second), the *execution time* needs to be minimized

$$performance_X = \frac{1}{execution\ time_X}$$

If X is n-times faster than Y, then:

$$\frac{performance_X}{performance_Y} = \frac{execution\ time_Y}{execution\ time_X} = n$$

We can *improve the performance* by:

- Reducing the time of a clock cycle (increasing the clock frequency)
- Reducing the number of clock cycles for a specific task (*efficient implementation*)

CPU Execution Time – Example

A task runs on microcontroller A with a 100 MHz clock in 10 seconds, *what clock frequency must a microcontroller B have to run the same task in 6 seconds*?

We assume Microcontroller B will require 20% more clock cycles for the same task.

$$CPU\ execution\ time_A * f_{clock_A} = Number\ of\ CPU\ clock\ cycles_A$$

$$10\ seconds * 100\ MHz = 10^9 clock\ cycles$$

$$f_{clock_B} = \frac{Number\ of\ CPU\ clock\ cycles_B}{CPU\ execution\ time_B} = \frac{1.2*10^9 \text{clock cycles}}{6 \text{ seconds}} = 200 \text{ MHz}$$

Computational Performance – Example

If microcontroller A executes a task in 10 seconds and microcontroller B needs 15 seconds for executing the same task, *how much is A faster than B*?

$$\frac{performance_A}{performance_B} = \frac{execution\ time_B}{execution\ time_A} = \frac{15\ seconds}{10\ seconds} = 1.5$$

Low Power Programming Strategies

Use Compiler Optimizations

The fundamental compiler optimization choice: code size vs execution time

- *-Ospace* This option, enabled by default, tells the compiler to apply optimizations for reducing image size, possibly at the expense of speed.
- *-Otime* This option tells the compiler to apply optimizations for execution speed, possibly at the expense of code size.

	Execution Time (ms)	**Program Size (bytes)**
Compiled with –Otime, -O3	2.383	150
Compiled with –Ospace, -O3	2.467	134

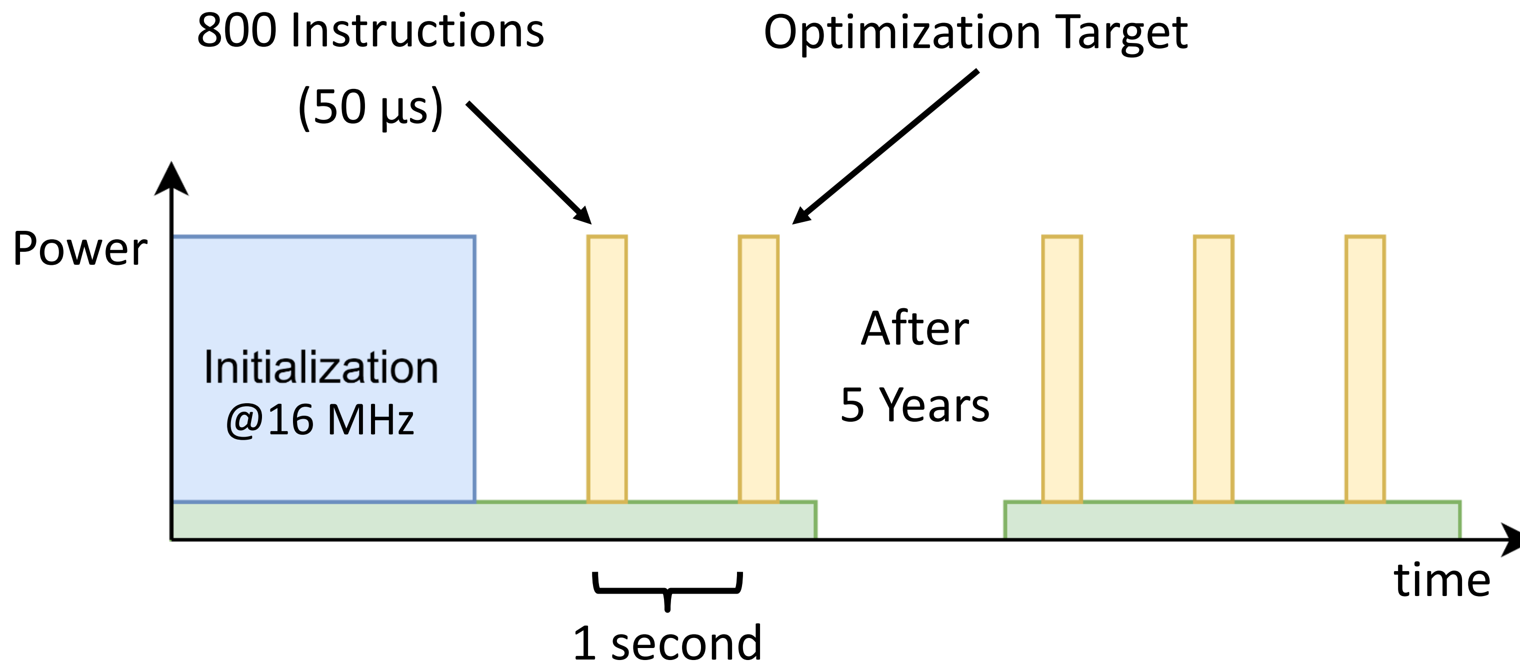
Same code, but different compiler optimizations

- To reduce time spent during execution, we have to select the time optimization compiler target

https://dl.acm.org/doi/fullHtml/10.1145/1452012.1452015

Analyse and Profile Your Application

For long application runtimes, the reduction of only a few instructions can already lead to a significant runtime improvement



Each single instruction is executed 158 million times over 5 years.

Removing only 1 instruction will generate a benefit of 2.3 days over 5 years

Removing 1 instruction = 2.3 days more runtime!

Optimized code has minimal instructions while having the *same functionality*

Use Memory if Available (ARM 32-bit)

```
#include <stdint.h>
unsigned short int counter = 10;

void A (void){
     while(counter){
          counter--;
                    movw    r3, #:lower16:counter
                    movt    r3, #:upper16:counter
                    ldrh    r3, [r3]
                    subs    r3, r3, #1
                    uxth    r2, r3
                    movw    r3, #:lower16:counter
                    movt    r3, #:upper16:counter
                    strh    r2, [r3]
     }
}
```

ARM GCC 13.2.0

UXTH{cond} {Rd}, Rm {,rotation} extends a 16-bit value to a 32-bit value.

8 instructions for *counter--;*

```
#include <stdint.h>
unsigned int counter = 10;

void B (void){
     while(counter){
          counter--;
                    movw    r3, #:lower16:counter
                    movt    r3, #:upper16:counter
                    ldr     r3, [r3]
                    subs    r2, r3, #1
                    movw    r3, #:lower16:counter
                    movt    r3, #:upper16:counter
                    str     r2, [r3]

     }
}
```

ARM GCC 13.2.0

7 instructions for *counter--;*

2 data bytes more but one instruction less!

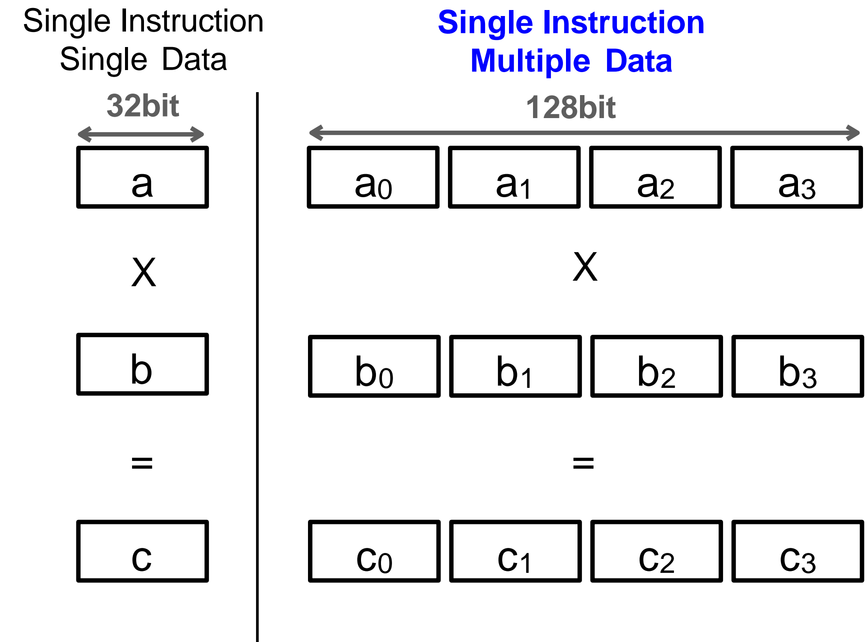
SIMD Programming

SIMD (Single Instructions Multiple Data)

- Perform operations in data vectors with a single instruction
- Available on most processors

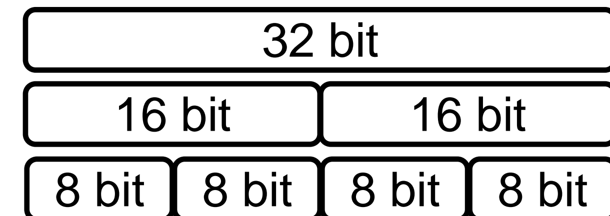
With SIMD, it is possible to:

- *Exploit data-level parallelism* in loops
- Achieve speedup *by quantization* (e.g. int8)



Difference between a conventional and SIMD instructions

The Cortex-M4 and Cortex-M7 provide SIMD instructions that operate on 8- or 16-bit integers. All registers are still 32-bits wide, but the SIMD instructions operate on 2 x 16-bit values or 4 x 8-bit values at a time within a 32-bit register.



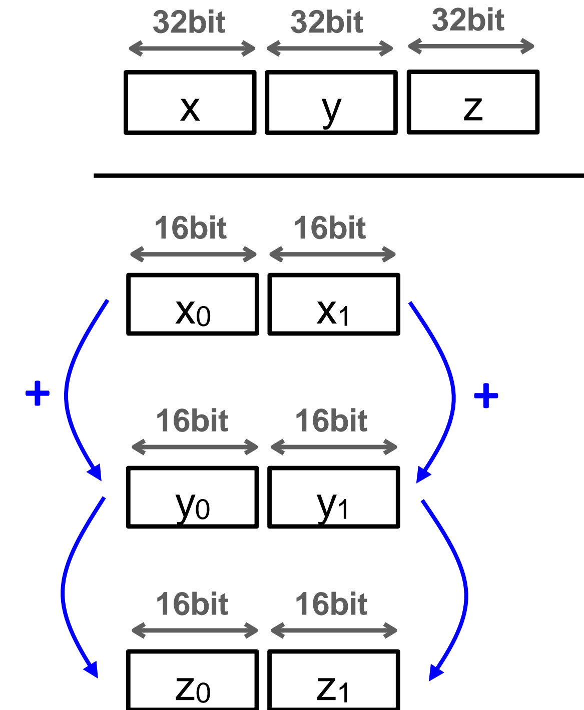
SIMD Programming in ARM Cortex-M

To use the SIMD instructions:

- load values into int32_t variables (i.e. 32.bit register)
- invoke the corresponding *SIMD intrinsic instruction*

Example:

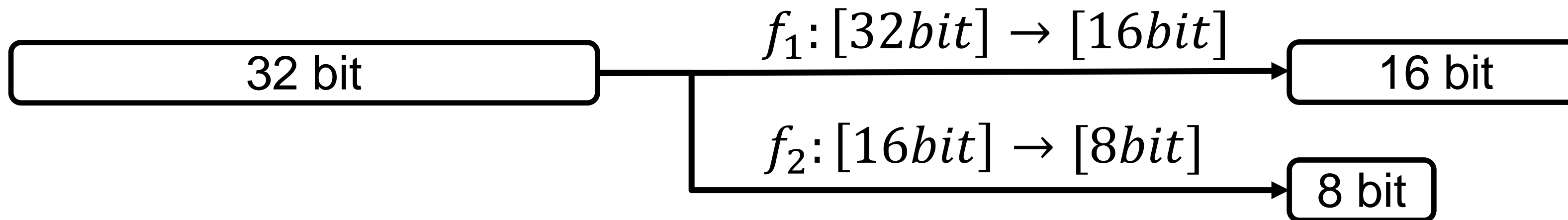
```
Int32_t x, y, z;
y = some value;
x = some value;
Z = __SADD16(x,y);
```



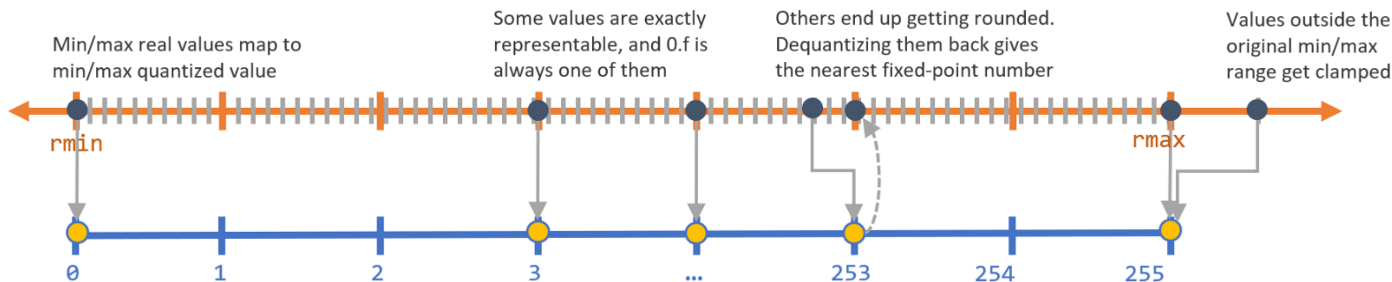
This will *perform two 16-bit signed additions in parallel on the corresponding halfwords of the operands and writes the results into the corresponding halfwords of the destination*, leaving the result in z.

Quantization

Quantization is the process of mapping continuous infinite values to a smaller set of discrete finite values. After quantization, SIMD can be used to speed up the task



f_x is the mapping function for the quantization. Depending on the application, f can be a linear- or non-linear function.



Why do we want to quantize?

Efficiency: Smaller numbers use less memory and allow us to process more data at once, especially with SIMD instructions.

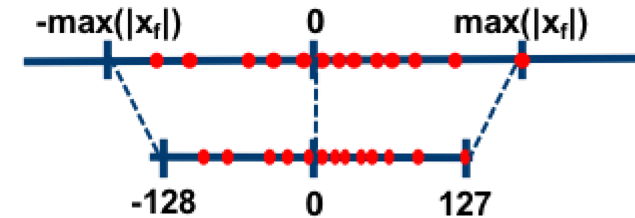
Speed: Integer operations are faster than floating-point operations in many embedded processors, including ARM Cortex-M.

Symmetric quantization

- Symmetric quantization maps floating-point values into a range that is symmetric around zero, like [−128,127] for 8-bit integers (Q7 format).

Formula:

- Xq=round(X × S)



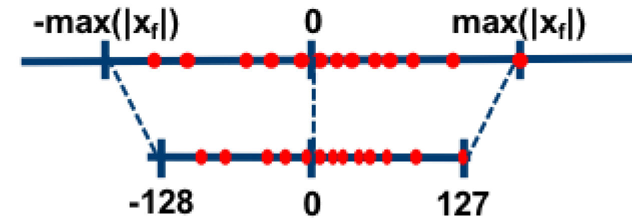
Where:

- X: This is the original floating-point value.
- S: The scaling factor, which is computed as: $S=Q/R$
  - Where :
    - $Q=2^n-1$ The maximum value for the integer representation (e.g., 127 for Q7)
    - R: The maximum absolute value of the floating-point range (e.g., [−1.0,1.0]).

Example

Example:

If we want to quantize $x = 0.5$ into Q7:



1. Compute the scale factor:

$$S = \frac{127}{1.0} = 127$$

2. Multiply and round:

$$x_q = \text{round}(0.5 \times 127) = \text{round}(63.5) = 64$$

3. Saturate (if necessary):
   - Check if x_q is within $[-128, 127]$. If not, clamp it to the nearest boundary.

Key Idea: Symmetric quantization works best when the data is centered around zero, with no offset.

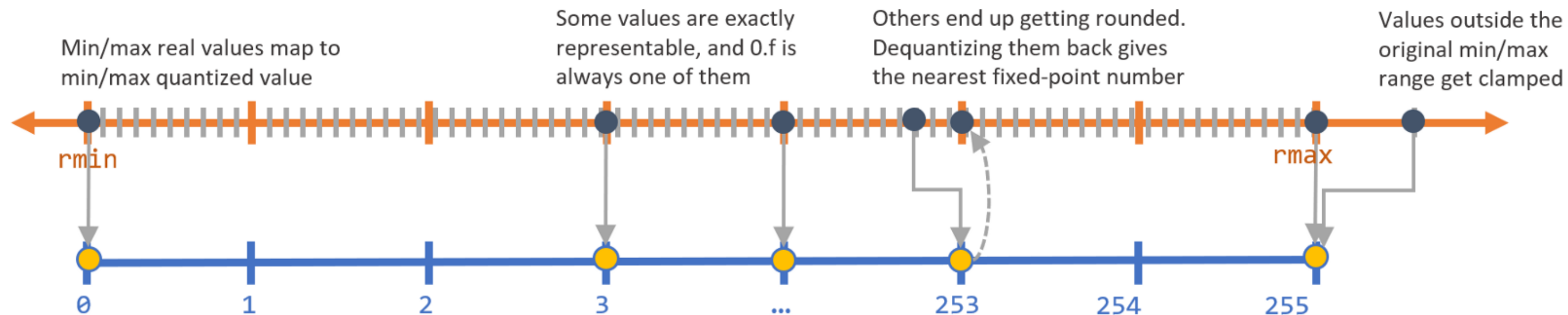
Mathematical Formulation for Quantization

- In **asymmetric quantization**, the integer range is shifted to allow a non-zero zero-point (Z), mapping the floating-point range [rmin , rmax]to the **integer range [0,2^{n-1}] (i.e an 8-bit range [0,255])**. The formula becomes:
- X_q=round(X × S + Z)

Where

S= 2^{n-1} /(rmax−rmin)

Z= − rmin × S = Zero-point offset.



Example

Example:

Let's quantize $x = 0.5$ into an 8-bit range $[0, 255]$, where the floating-point range is $[-1.0, 1.0]$:

1. Compute the scale factor:

$$S = \frac{255}{1.0 - (-1.0)} = \frac{255}{2} = 127.5$$

2. Compute the zero-point:

$$Z = -(-1.0 \times 127.5) = 127.5$$

3. Quantize:

$$x_q = \text{round}(0.5 \times 127.5 + 127.5) = \text{round}(191.25) = 191$$

4. Saturate (if necessary):

   - Check if x_q is within $[0, 255]$. If not, clamp it to the nearest boundary.

Key Idea: Asymmetric quantization introduces a zero-point offset to handle data ranges that don't include zero.

Speed-up going to quantization

We will use two arrays, one of **32-bit floating-point values** and another of **quantized 8-bit values (Q7)**:

- **Floating-point data (32-bit):** $X=[0.5, -0.75, 0.1, -1.0]$
- **Quantized data (8-bit Q7):** $Xq=[64, -96, 13, -127]$

We have also quantized operands

$$Yq=[32, -32, 45, -64]$$

Scalar Operation (32-bit)

In a scalar operation, each addition would process one element at a time and use 32-bit representation for the floating-point values.

- Result=$x1+y1$, $x2+y2$, $x3+y3$, $x4+y4$
- Since we are using floating-point values, each of these additions will use **32-bit precision** and take at least **one clock cycle per operation** (for a simple CPU without optimizations).

Move to use SIMD

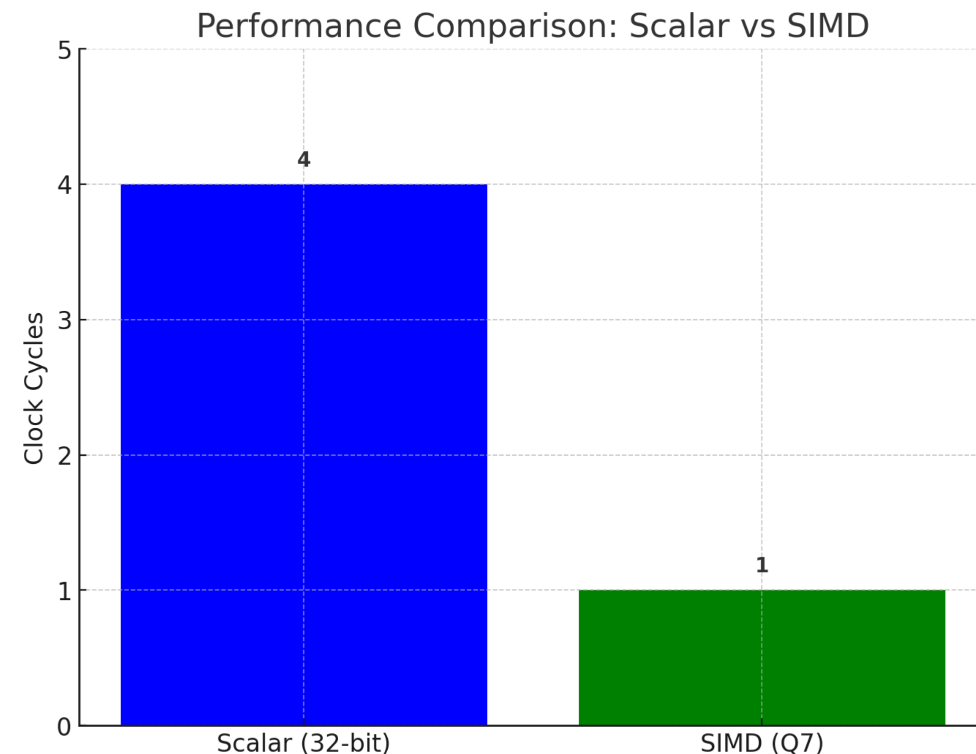
- **SIMD (Q7 quantized addition):** we can perform
- Xq +Yq directly
  - In SIMD, adding quantized values with QADD8\text{QADD8}QADD8 allows us to process all the values simultaneously in one instruction:

- $64+32=96$
- $-96+(-32)=-128$
- $13+45=58$
- **$-127+(-64)=-191$, which saturates to -127 due to Q7 limits.**
  - **OVERFLOW ISSUE**

Let’s see the benefits

- **Scalar Floating-Point Operations (32-bit):**
- Each of the 4 additions uses **32-bit floating-point** precision, taking **1 clock cycle per operation**. So the total time for the scalar operation is:
- Using the **SIMD** instruction QADD8, all 4 additions are performed in **1 clock cycle**.

$$\text{Speedup} = \frac{\text{Scalar Time}}{\text{SIMD Time}} = \frac{4 \text{ clock cycles}}{1 \text{ clock cycle}} = 4\times$$



What Did You Learn?

- ✓ Optimising data transfer using Direct Memory Access (DMA)
- ✓ Instruction Set Architecture (ISA)
  - ✓ ARM
  - ✓ SIMD
- ✓ Quantization and Speed-up Calculation