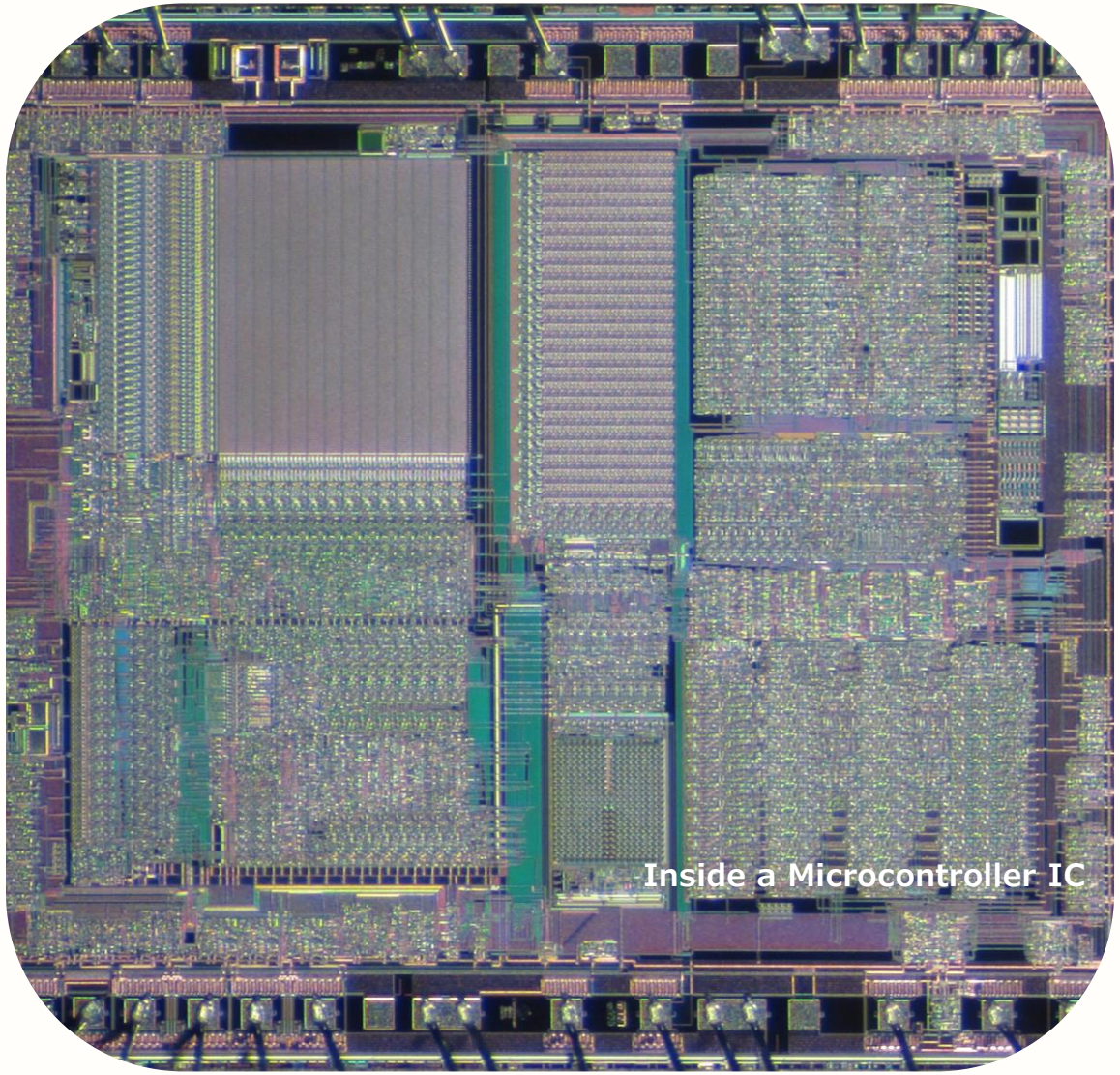


Evolution of Memory Architecture and internal blocks of MCUs

Session #2

A Journey within a Microcontroller IC

Doc Version : V1.0 Date : Apr 2026

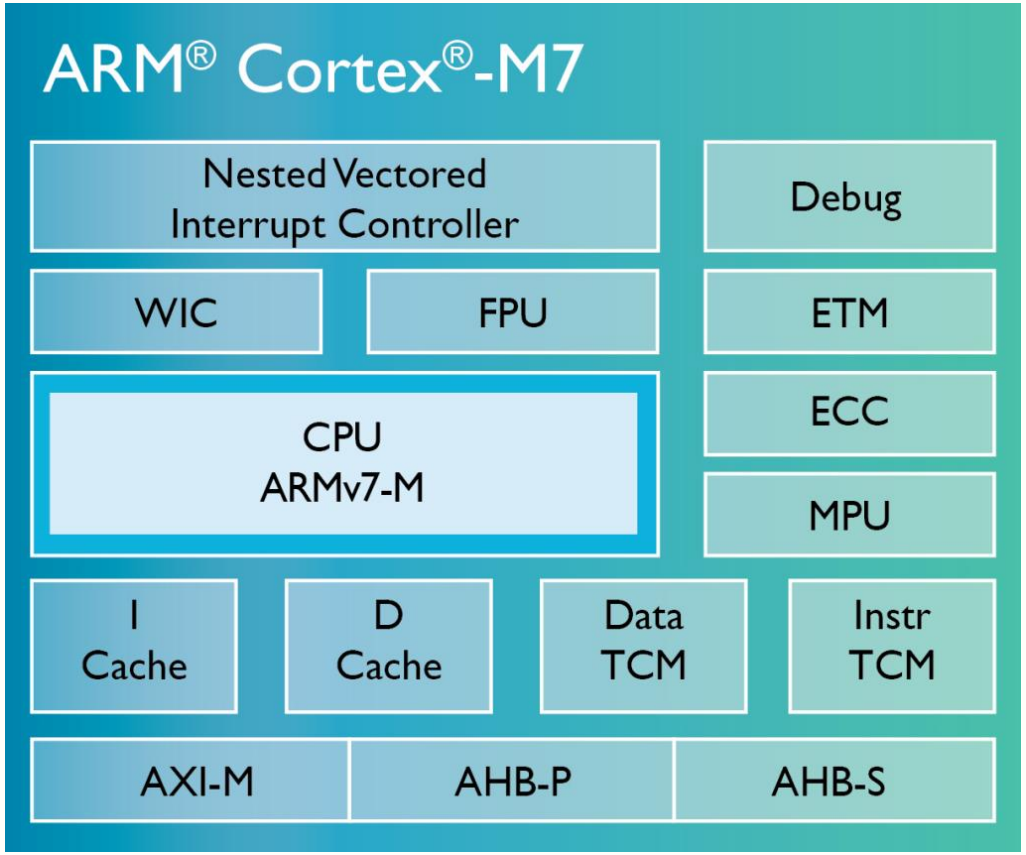
What we covered in our Session #1?

- Journeyed through how technology evolved in last 80 years (From Mechanical computing to today's computing using ICs)
- Learnt how the progression happened from electronic valves to Semiconductor, Diode, Transistors, MOSFETs, LSI Logic ICs, ALUs, Memory, First processor and First MCU

What are we going to cover in Session #2

Like we journeying through the evolution of IC, we will now journey through the blocks inside the MCU

- How the Memory Architecture for MCUs evolved
- What are the key blocks besides the CPU and why they are there



ARM Cortex-M7: One of several CPUs designed by "ARM" that can be licensed and used to create a Microcontroller IC

Agenda

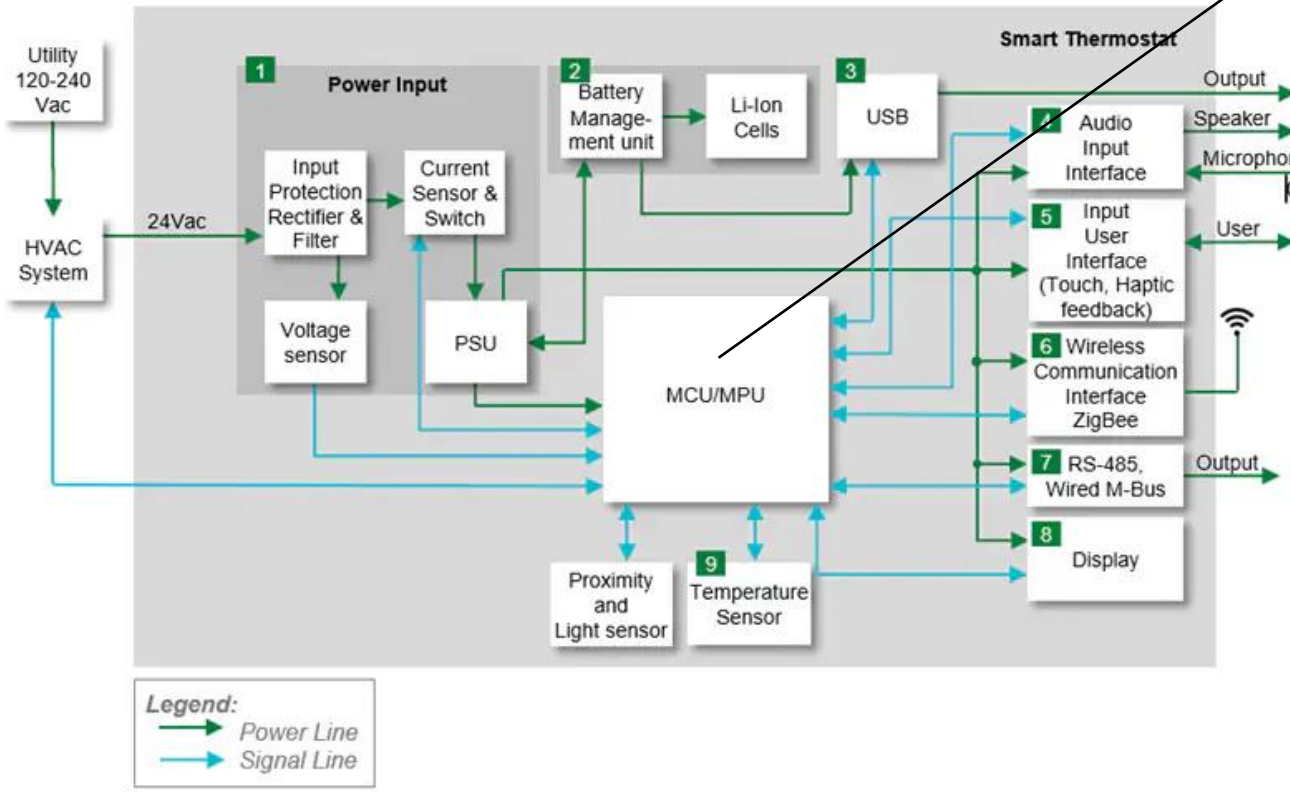
- MCU Memory Architecture
- Evolution of memory Architecture: Von Neumann, Harvard, Modified Harvard
- Memory space, addressing and partitions example from ARM Cortex M
- DMA
- DSP, FPU
- NVIC
- Instruction pipelining
- Flash instruction Fetch Accelerator, Cache Memory, TCM
- Bus Protocols and Bus Matrix

Before go into the MCU..

Where and How an MCU is used?

MCU is at the core of many embedded applications

Smart thermostat block diagram



	Technology	Series
1	Chip Fuse (24V AC)	437, 468
	PPTC	2920L,SMDC
	TVS Diode	SACB, SMAJ, SMF3.3
	Latching Relay Driver	CPC1600
2	TVS Diode, MLV	MLA, SMF
	PPTC	femtoSMD, nanoSMD, picoSMD
	Strap PPTC	MXP, SL
3	TVS Array	SESD, SPXX
	PPTC	0402L, femtoSMD
4	TVS Array	SACB, SMAJ, SMBJ
5	TVS Array	PESD, SP3213-01UTG
6	TVS Array	SP3213-01UTG
	Polymer ESD	PESD
7	TVS Array	SM712
8	TVS diode, MLV	MLA, PLED, SMF
9	NTC	SM Series, RB, DO-35

https://www.mouser.in/new/littelfuse/littelfuse-smart-thermostat/

What Does different Embedded Systems do in Real Life?

🌡️ Smart Thermostat	🚗 ABS Brake Controller	⌚ Fitness Wristband
Senses temperature · Runs PID control algorithm · Drives touchscreen & Wi-Fi stack	Reads wheel speed sensors, 6 axis sensors · Decides in microseconds · Modulates brake pressure	Runs on coin cell battery · Senses heart rate · Transmits data via BLE 5.0
🤖 Industrial Robot Arm	🔋 EV Battery BMS	📡 Wi-Fi Router
Hard real-time motion control · Multi-axis coordination · Safety-critical firmware	Monitors 100s of cells simultaneously · Balances charge · Prevents thermal runaway	Packet forwarding at line rate · QoS scheduling · Over-the-air firmware updates

1. Smart Watch
2. Anti Skid Braking System (ABS)
3. EV Battery Management System (BMS)
4. EV Motor Control

Amongst the above applications

- Where the low power is most critical? (If multiple, list it in an order)
- Where the Safety is most critical?
- Where the performance requirement is the highest?

As an Engineer

What do I need to know, “to select the Microcontroller for any given end application?” and..

Why should I learn about how to select an MCU?

What Most Engineers commonly do?..

→“Google” the application

→Find a reference board/schematic/application note/popular tutorial

→ Select that MCU…

This works Perfectly – Until it does not and when it does not, they don’t know why it does not work…

What if the engineer consciously understands the MCU selection?

→Builds products that work first time (Saves time), avoids costly board re-spins (Saves cost)

→ Good at system-level problem debug as they understand the architecture well (Creates a product, that doesn’t return)

→ Can defend their MCU choice to their manager & organization with technical evidence (Makes Manager’s job easier in getting the necessary funds and attention from management for project execution).

What do I need to know for choice of MCU (Broader Level)

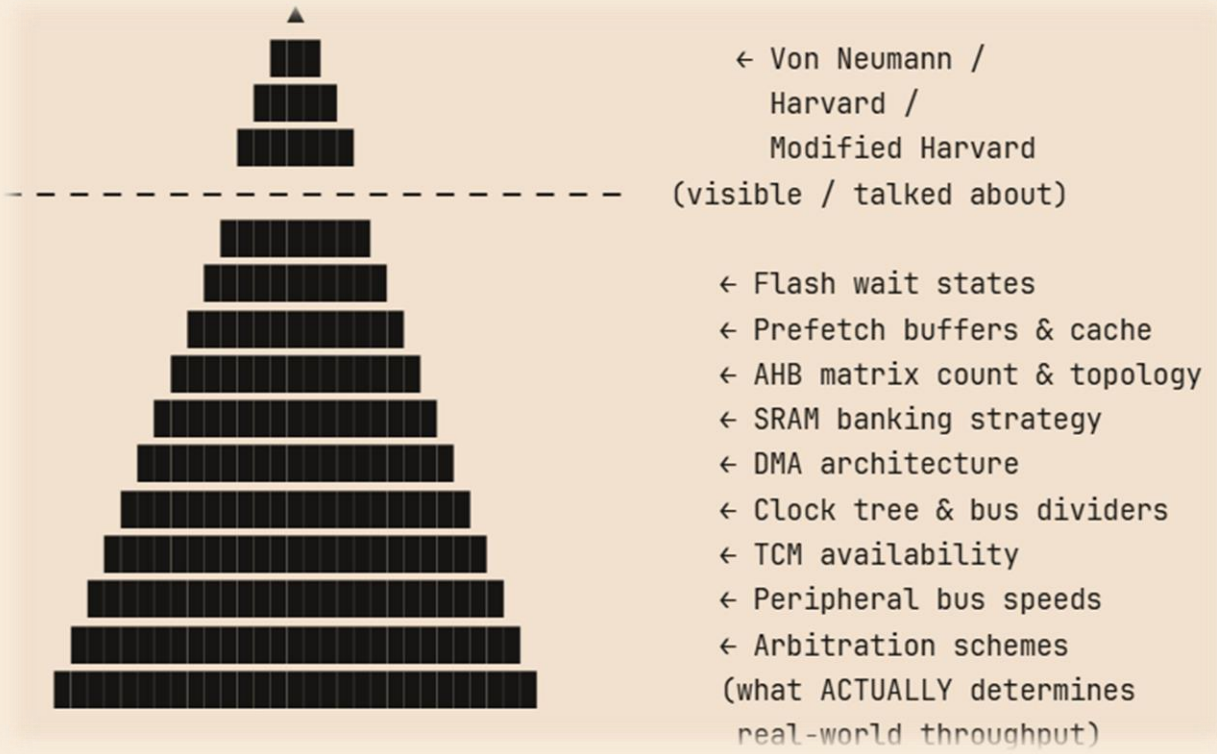
From Silicon to Software Making the Right MCU Decisions

1) The Application Requirements	2) The MCU Architecture Knowledge	3) The Ecosystem and Production Factors
• What does the system need to compute? • What does the system need to communicate? • What does the system need to sense and actuate? • What are the real-time requirements? • What is the power budget? • What is the memory requirement? • What is the cost target? • What is the operating environment? • Are there safety or security requirements?	• Memory architecture • Bus Architecture • Peripheral set • Processing Capability • Power Architecture • Security Architecture	• Tool chain and IDE • Community and Documentation • Reference designs and Evaluation boards • Long term availability • Price and supply chain

“MCU selection depends on 3 things: 1) End Application 2) MCU Architecture 3) MCU Ecosystem”

MCU Memory Architecture

Von Neumann .
Harvard . Bus Matrix .
TCM . Cache . DMA



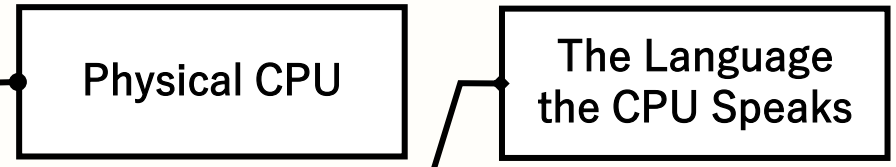
Session #2

MCU ARCHITECTURE KNOWLEDGE (At a deeper level)

1	2	3	4
Memory Architecture choices for MCU Core	Instruction Set Architecture (ISA) choices for MCU core	MCU Core choices and configuration	Benchmarking MCU
How does the CPU access Instruction/Code space and Data space?	What kind of Instruction set would suit for this MCU design?	Do we design a custom CPU or RISC-V ISA based or ARM ISA based?	So many MCU options out there. How to pick the best for my requirement?

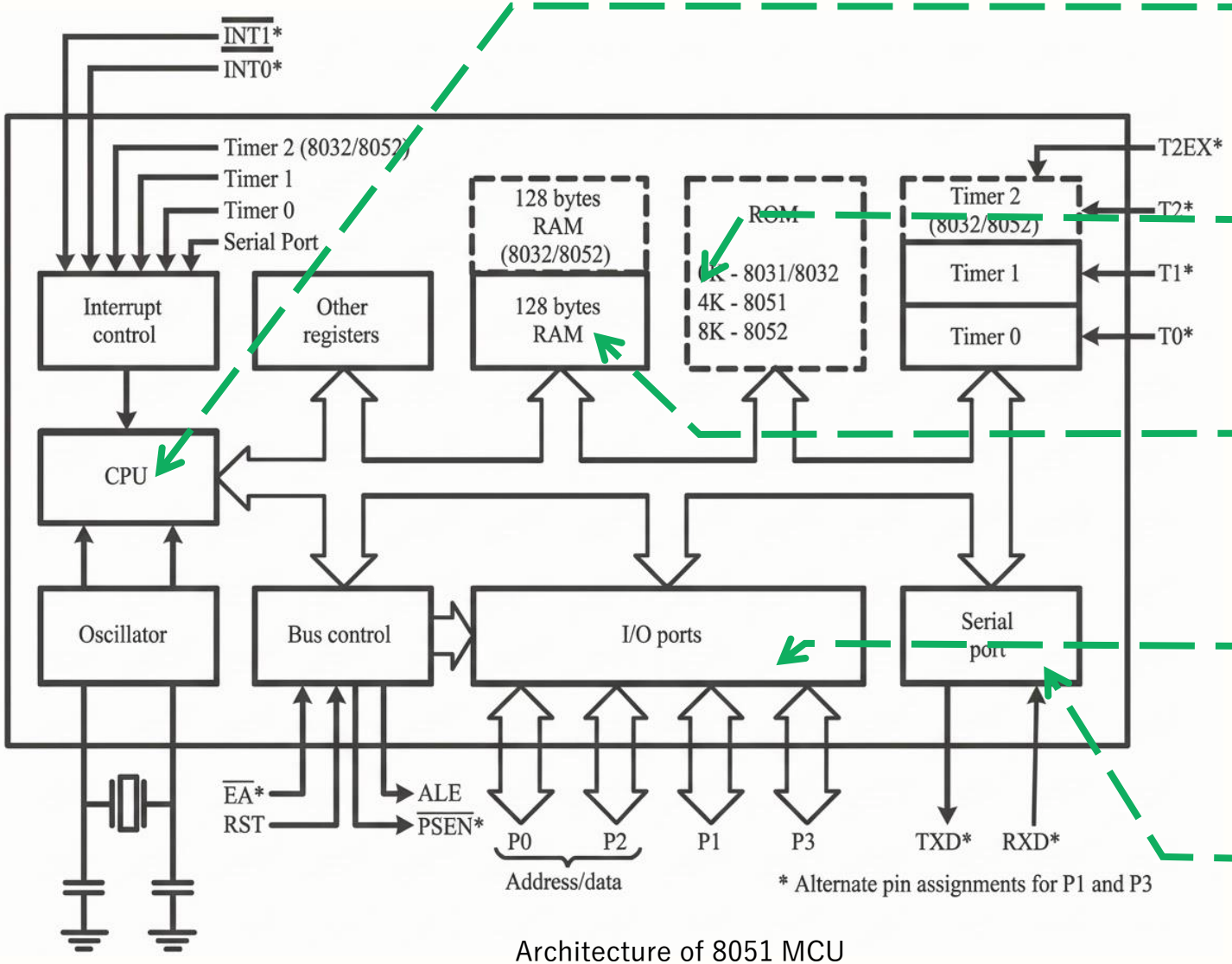
By the end of this section, you will understand not just what the specs on a datasheet mean — but what is important for your system design and how to find out the MCU that meets.

Your Decision-Making Framework



#	Section Title	What You Will Learn
01	MCU Memory Architecture	Von Neumann vs Harvard vs Modified Harvard · Bus matrix · TCM · Cache · DMA — why memory determines real throughput, not clock speed
02	Instruction Set Architecture (ISA)	CISC vs RISC vs RISC-V · ARM ISA · Thumb · Thumb-2 · DSP extensions — what ISA means for your code, compiler, and performance
03	ARM Cortex-M Cores & Vendor Ecosystem	M0 to M85 · ARM's IP licensing model · How ST, NXP, Nordic, Microchip differentiate around the same CPU core
04	Benchmarking — Choosing with Evidence	EEMBC CoreMark · ULPMark · FPMark · Reading vendor benchmarks honestly · When to run your own tests

A refresh question… What Is Actually Inside an MCU?



Architecture of 8051 MCU

Where code is executed
CPU

Where your code lives
ROM (Flash)

Where your data lives at runtime
RAM

External interface
I/O Ports

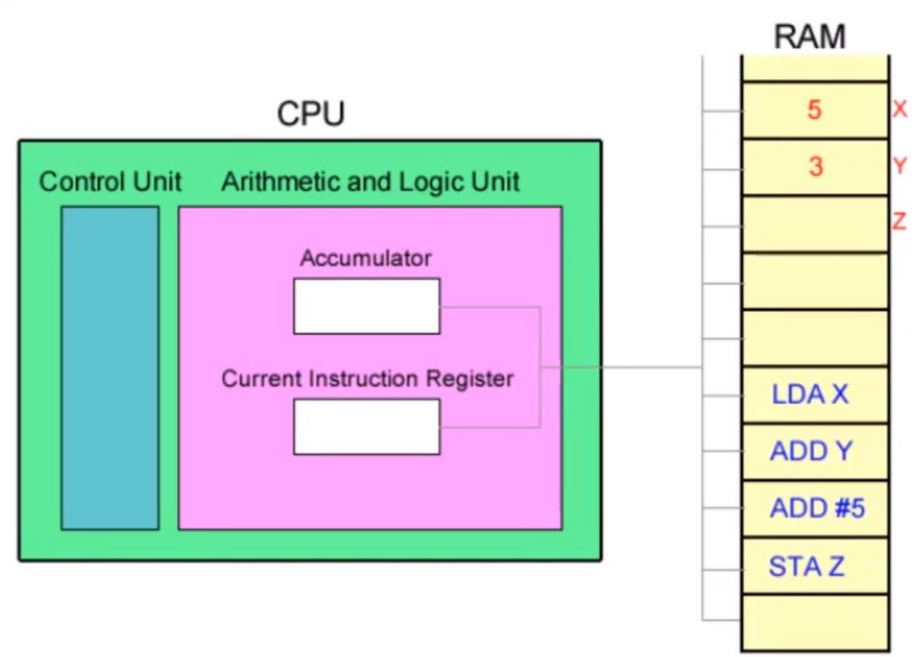
External data Exchange
Serial Port:

We have seen an MCU needs two types of memory areas.

1) A memory area to store the **program / instructions**.
2) Another memory area to store the **data**

Do you think the CPU accesses the program memory (Flash) first and then data or

Does it access both Program Memory and Data Memory at a time?

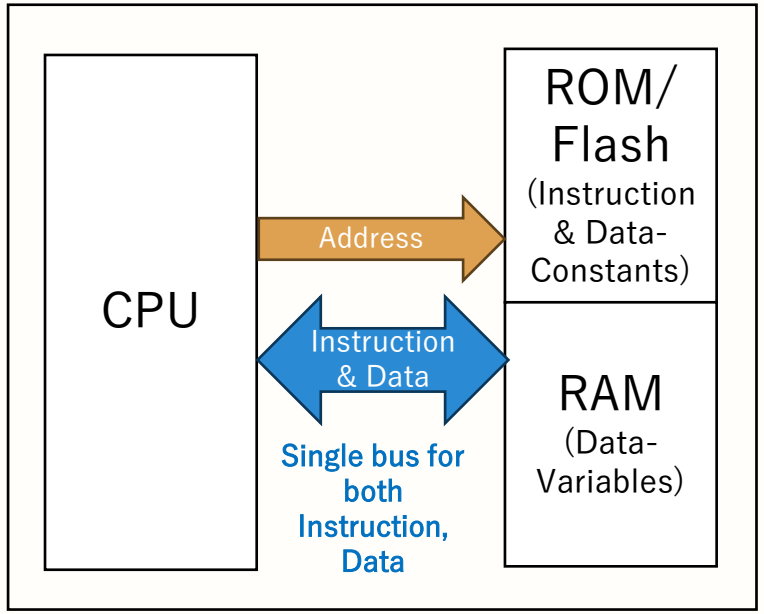


https://www.youtube.com/watch?v=RRU_LU5XyQg

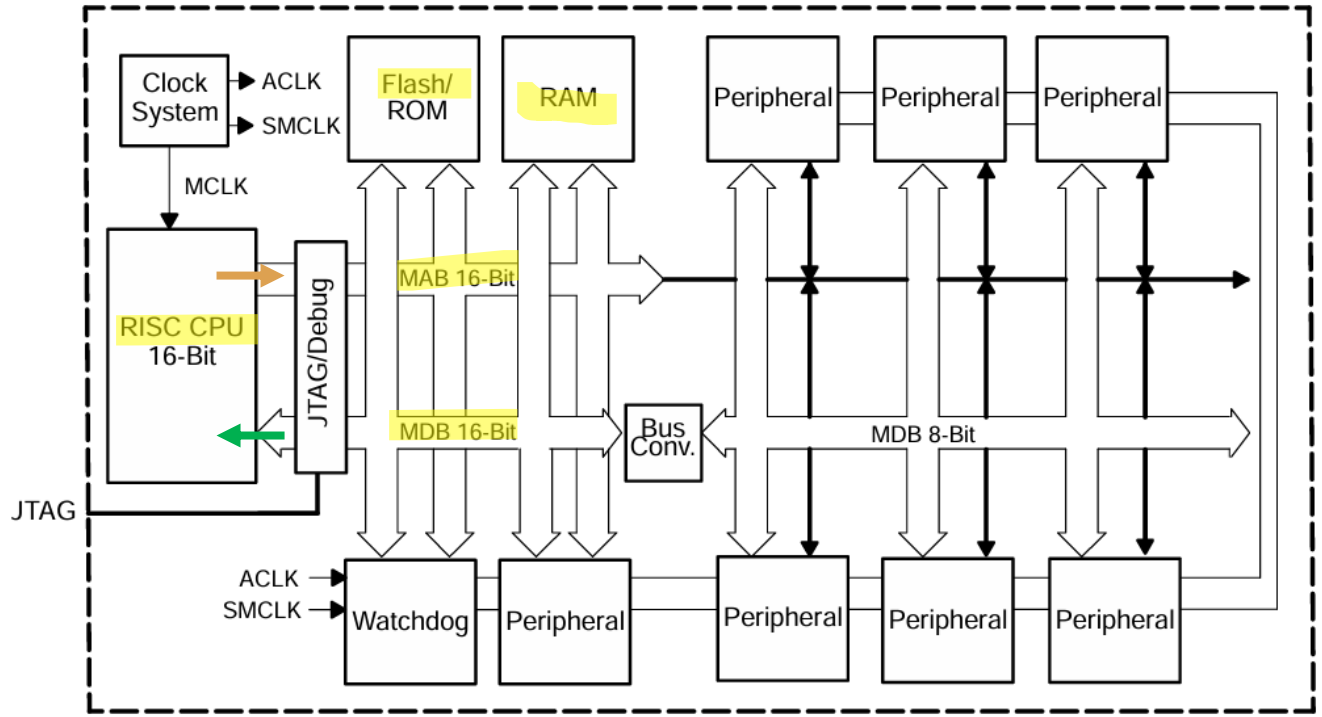
Von Neumann Architecture

MCU Memory Architecture - *Von Neumann*

Von Neumann Architecture



MSP430F4xx MCU with Von Neumann Architecture



⚠ **The Von Neumann Bottleneck**

Stores both the program instructions and the user data in one memory address area and use a single bus to carry both into the CPU → As there is only one bus “common for Instruction & Data”, the CPU can either fetch an instruction OR read some data — but not “both” at the same time. → **performance ceiling**

✅ Advantages

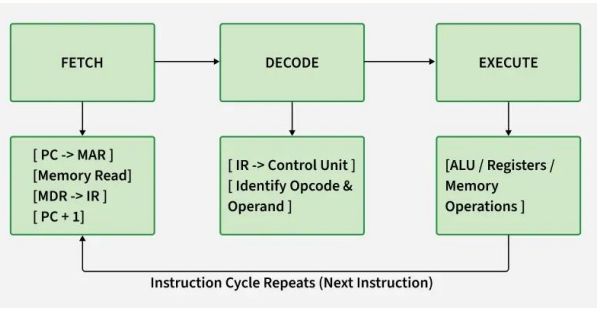
Simple silicon implementation;

Unified memory, software friendly;

❌ Disadvantages

Bus bottleneck;

CPU stalls on every data access;



Pros of Von Neumann Architecture

- **Simpler Hardware Design implementation**: Only one memory space and one set of address/data buses are needed, which reduces chip complexity, silicon area, and overall cost.
- **Flexible Memory Usage**: Program code and data share the same memory address space, allowing dynamic allocation and easier modification of memory usage at runtime. Programmers don't have to worry about separate code and data memory limits.
- **Easier Programming and Debugging**: Unified memory model simplifies software development, compiler design, and debugging tools. Self-modifying code and loading programs into memory are straightforward.
- **Better for General-Purpose Applications**: Ideal for tasks where code and data sizes vary significantly or where frequent data access to code memory (e.g., constants stored in Flash) is needed. Most general-purpose MCUs and Desktop, Laptop processors still prefer this approach due to its simplicity and flexibility.

In short, Von Neumann's biggest advantage is **simplicity and flexibility**, making it the preferred choice for cost-sensitive and moderately performing embedded systems, even though it introduces the well-known memory bottleneck.

Summary - Von Neumann Architecture

Role in MCU : Single shared memory space and bus for both instructions and data. Simpler hardware and flexible memory usage.

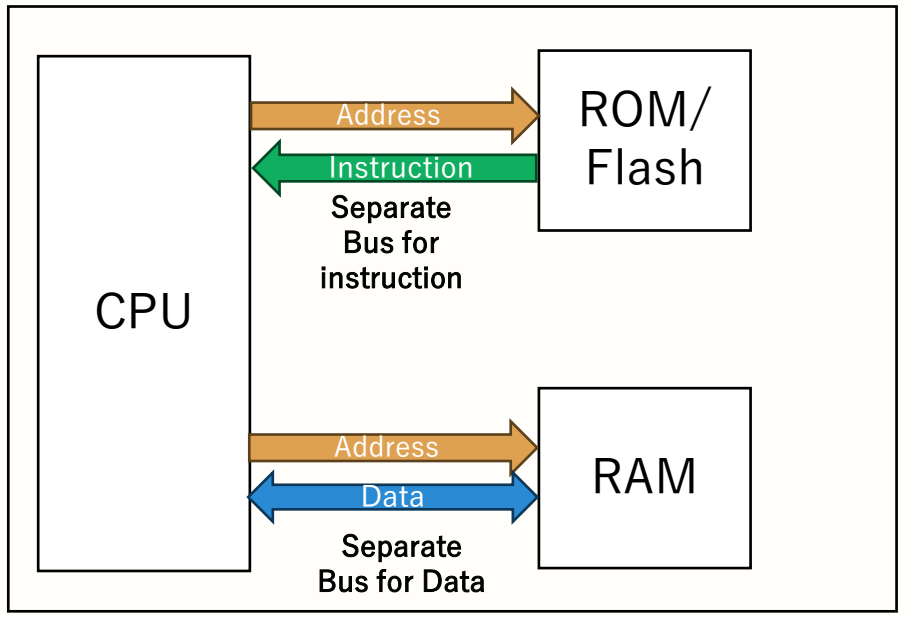
Impact on Overall MCU Computing Performance

- While being a simplest possible Memory Architecture approach, Creates the classic von Neumann bottleneck — instruction fetches and data accesses compete for the same bus/memory bandwidth.
- Limits peak throughput and increases latency in compute-intensive tasks. Best suited for cost-sensitive, general-purpose applications.

Harvard Architecture

MCU Memory Architecture - *Harvard*

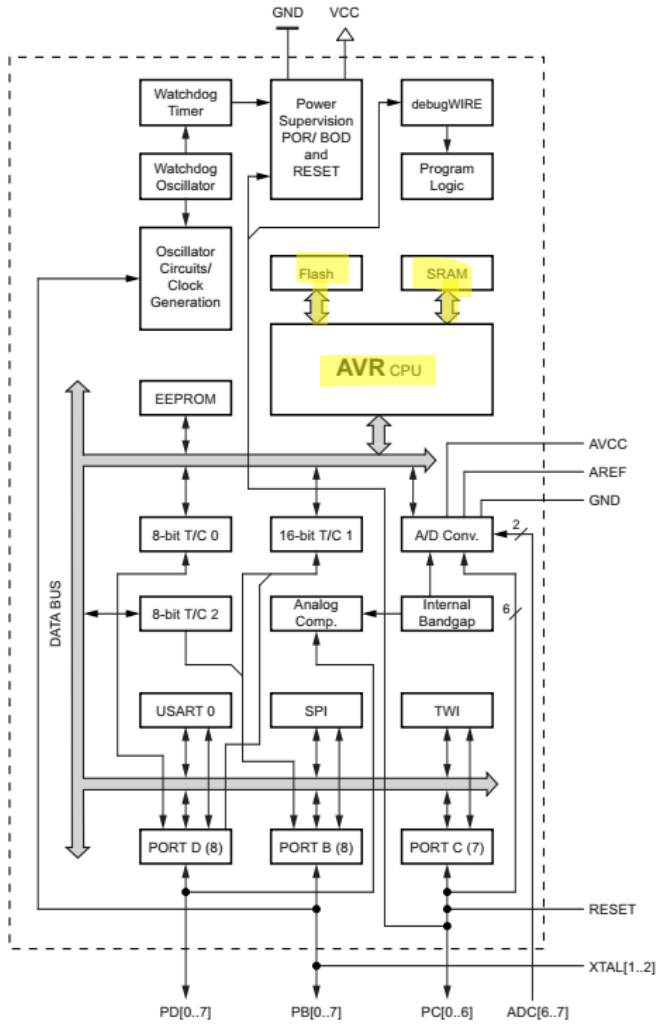
Harvard Architecture



Harvard achieves up to 2× the throughput of Von Neumann on memory-intensive workloads

- **"Completely separate" memory spaces** — one for instructions, one for data
- **"Completely separate" buses** — instruction bus and data bus are independent
- CPU can fetch the **next instruction** while simultaneously **reading or writing data**
- Eliminates the Von Neumann bottleneck entirely. Higher throughput
- Used extensively in **Digital Signal Processors (DSPs)**
- Code **cannot** be executed from data memory (RAM) — fixed separation (Why there is a need for running code from RAM?. We will see later)

Atmega328P Datasheet (Strict Harvard Architecture)



Von Neumann Vs Harvard Architecture – Clock cycles

Example sequence of Instruction Execution:

```
Instruction 1:   LOAD  R0, [0x200]    ; Load value from memory address 0x200 into R0
Instruction 2:   LOAD  R1, [0x204]    ; Load value from memory address 0x204 into R1
Instruction 3:   ADD   R2, R0, R1     ; Add R0 and R1, store result in R2
```

Von Neumann: 9 Cycles

```
        Clk1    Clk2    Clk3    Clk4    Clk5    Clk6    Clk7    Clk8    Clk9
        ____    ____    ____    ____    ____    ____    ____    ____    ____
I1:   [FETCH] [DECOD] [EXEC ]
I2:                   ▒▒▒▒▒▒  [FETCH] [DECOD] [EXEC ]
I3:                                   ▒▒▒▒▒▒  [FETCH] [DECOD] [EXEC ]

▒▒▒ = Waiting — bus occupied by previous instruction's data read
```

Harvard: 5 Cycles

```
        Clk1    Clk2    Clk3    Clk4    Clk5
        ____    ____    ____    ____    ____
I1:   [FETCH] [DECOD] [EXEC ]
I2:           [FETCH] [DECOD] [EXEC ]
I3:                   [FETCH] [DECOD] [EXEC ]

No gaps. No waiting. Natural overlap from independent buses.
```

For the same clock frequency, which is faster?

Harvard : Improved throughput for the same clock frequency,

Easier to implement pipelines for further throughput enhancement

PROs - Harvard Architecture

- Enables **simultaneous access** to instructions and data via separate buses and memory spaces, eliminating the von Neumann bottleneck
- Delivers **higher throughput** and faster overall processing speed, ideal for real-time embedded applications
- Provides **predictable and deterministic performance** — critical for hard real-time systems in MCUs
- Improves **memory efficiency** by allowing different word sizes and optimized memory types (e.g., Flash for code, SRAM for data)
- Enhances **system security and stability** — instructions and data cannot easily corrupt each other
- Supports efficient **pipelining** and parallel operations, boosting performance without increasing clock frequency

CONs - Harvard Architecture

- Requires **more complex hardware** — duplicate memory units, separate buses, and additional control logic
- Increases **chip area and manufacturing cost** compared to simpler Von Neumann design
- Offers **limited flexibility** — fixed separation of code and data memory makes dynamic memory allocation harder
- Can lead to **under-utilization** of one memory space if code or data size is imbalanced
- More difficult **programming and debugging** in some cases due to separate address spaces
- Less suitable for general-purpose computing (PC, Laptops) where code and data sizes vary widely at runtime

Summary – Harvard Architecture

Role in MCU :

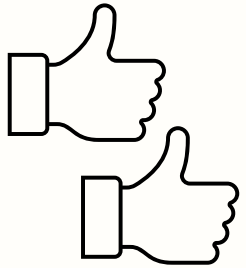
- Separate memory spaces and dedicated buses for instructions and data. Allows simultaneous access.

Impact on Overall MCU Computing Performance

- Eliminates the bottleneck by enabling parallel instruction fetch and data access.
- Significantly boosts throughput, pipeline efficiency, and determinism.
- Excellent for real-time and signal-processing workloads.

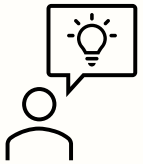
Von Neumann – Unified memory space

Harvard – Better throughput of code execution



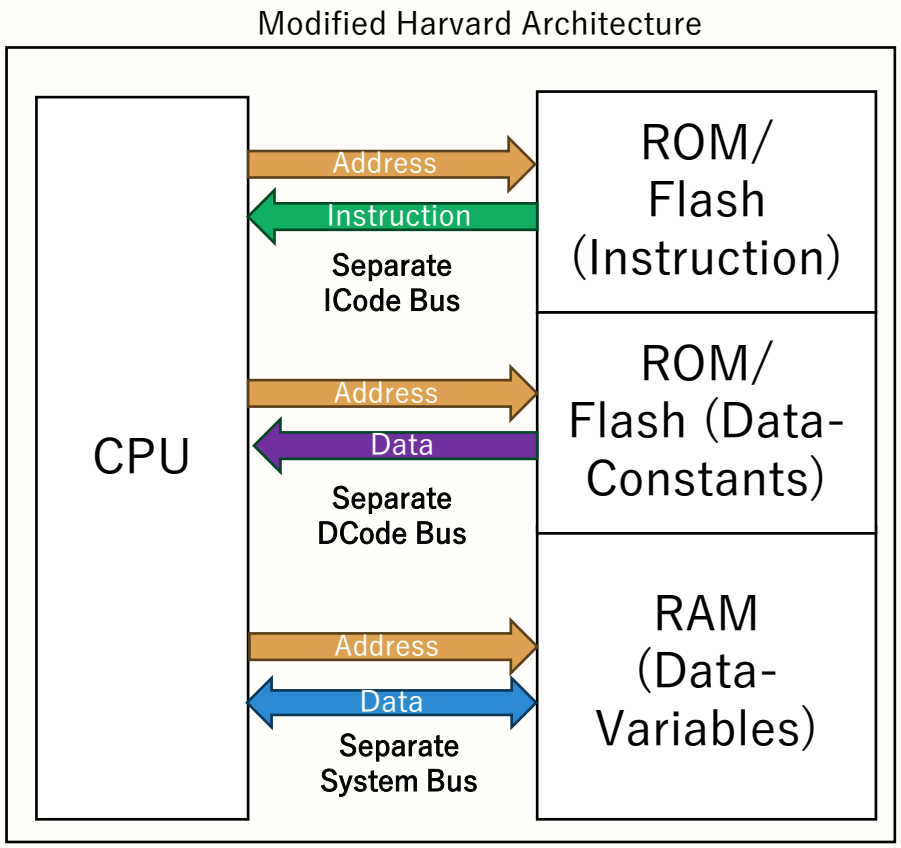
What would you prefer for an MCU?

What if we can get both the benefits?



Modified Harvard Architecture

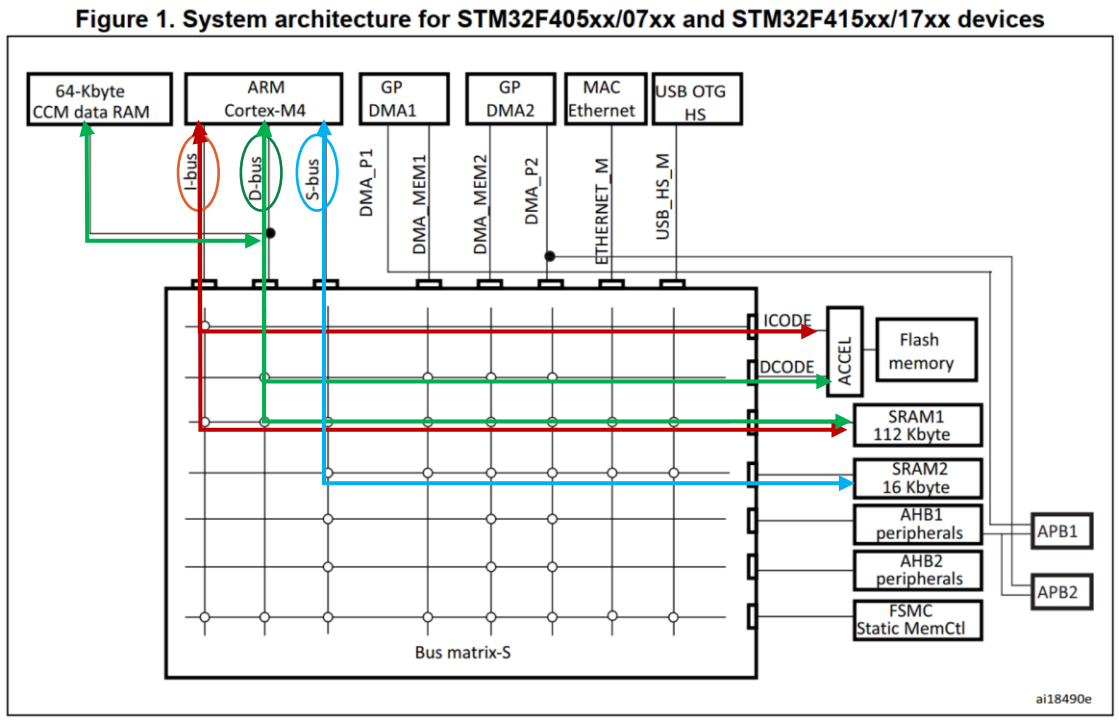
MCU Memory Architecture – *Modified Harvard*

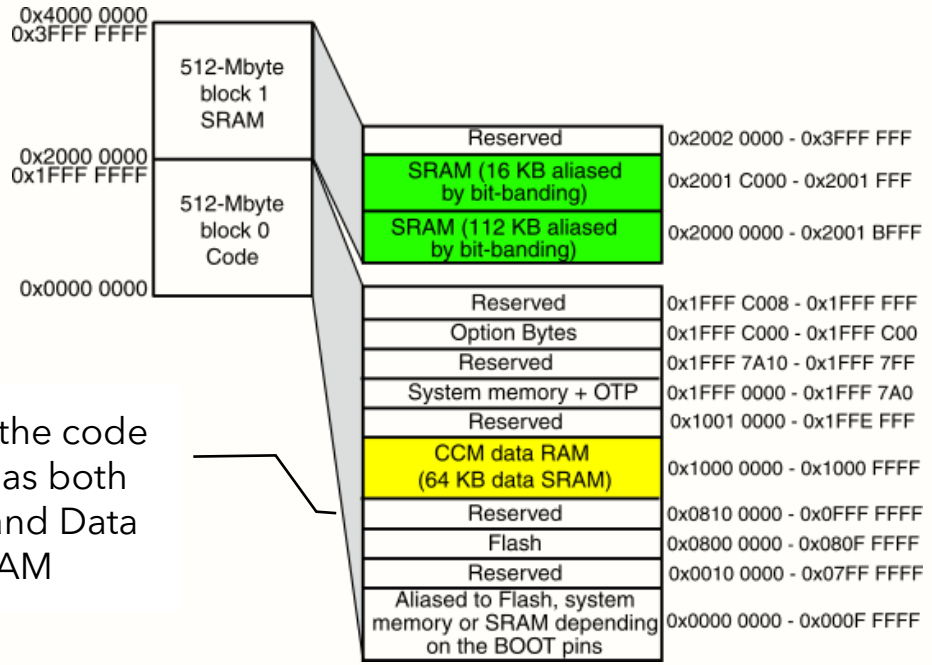


Three dedicated bus interfaces

- **I-Code bus:** Primarily for instruction fetches from non-volatile memory (e.g., Flash/ROM)
- **D-Code bus:** For data accesses (literal loads/constants) from the code/non-volatile memory region
- **System bus:** For data accesses to volatile memory (SRAM), peripherals, and the rest of the memory map

Modified Harvard Architecture :
ARM Cortex M4 MCU bus implementation (STM32F4)





Notice the code area has both Flash and Data RAM

Two separate buses (ICode, DCode) for instruction and data fetch from the Code space.

Third separate System Bus (S-Bus) to access SRAM.

— exactly like Harvard, but a single unified address space, exactly like Von Neumann.

The best of both worlds.

Note: CCM Data RAM implementation is done by ST and not by ARM. We will discuss later the TCM implementation by ARM.

The Modified Harvard Solution – Three Key Properties

Property 1 — Separate Buses (kept from Harvard):

- The CPU has a dedicated **ICode bus** for instruction fetches and a dedicated **DCode bus** for data accesses within the code region. These two buses operate independently and simultaneously — the Harvard performance advantage is fully preserved.

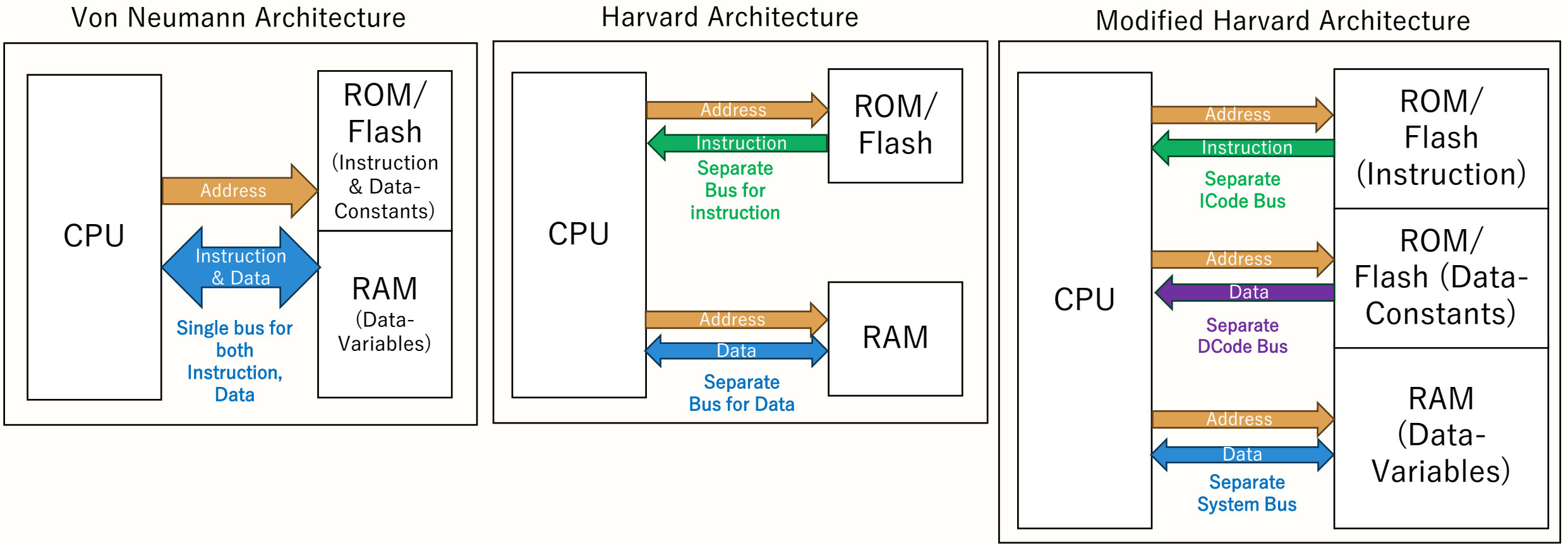
Property 2 — Unified Address Space (taken from Von Neumann):

- There is one single 4 GB address map shared by everything — Flash, SRAM, peripherals, external memory. Code and data live in the same address space — the linker works with one unified memory map.

Property 3 — Flexibility (new — not in either pure architecture):

- Because address space is unified, code can be copied to SRAM and executed from there. Constants and lookup tables in Flash can be read as data through the DCode bus. A bootloader can receive new firmware over UART, write it to RAM, and execute it immediately.

Comparison of Three Architectures



Summary – Modified Harvard Architecture

Role in MCU :

- Hybrid design: Separate fast paths for instructions and data at core level, but unified main memory.

Impact on Overall MCU Computing Performance

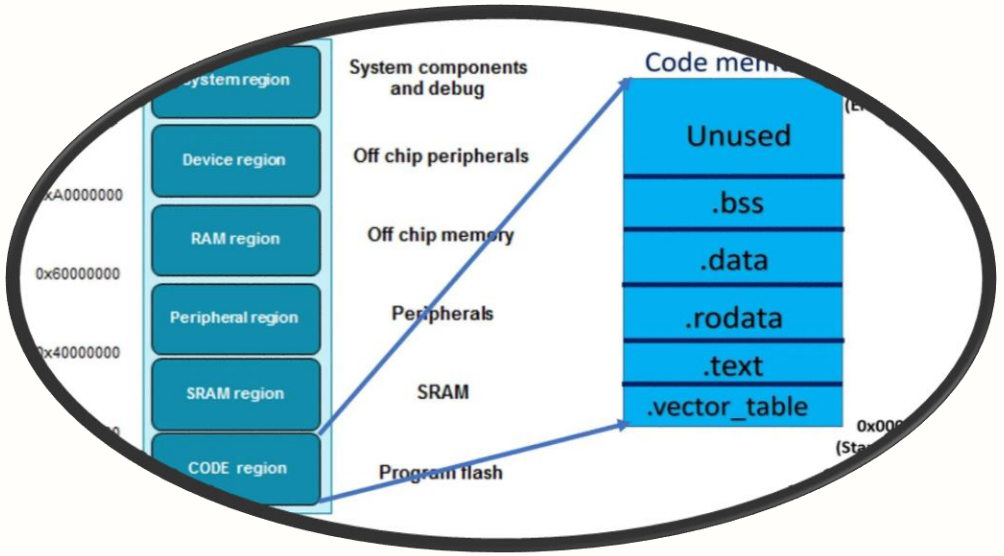
- Delivers the best balance — high performance from Harvard-style parallelism while retaining programming simplicity and flexibility of unified memory.
- Forms the foundation of nearly all modern high-performance MCUs.

Are we clear where Von Neumann is still used today, why

and

what is the default MCU architecture these days to go to and why?

Memory Address space and Memory map in ARM M CPU



Unified 4GB Address space in ARM Cortex-M

Arm® Cortex®-M processors support a 4GB (4 Gigabytes) address space.

This is based on the 32-bit architecture, which uses 32-bit memory addressing, allowing the processor to address 2^{32} unique locations, typically defined as 4 gigabytes (0x00000000 to 0xFFFFFFFF).

Key Details:

Byte-Addressable: While the memory system is typically 32-bit wide (4 bytes/word), the Cortex-M architecture is byte-addressable, meaning each address refers to a single byte. The 4GB address space is divided into 0.5GB regions, each assigned for specific function.

Unified Memory Map: Instructions and data share the same 4GB address space, which is partitioned into regions for code, SRAM, peripherals, and system components. Internal peripherals and debug components have fixed memory locations.

Word Alignment: Although byte-addressable, memory accesses, particularly peripheral registers, are usually 32-bit aligned (addresses divisible by 4) to improve efficiency.

The Full Memory Map Table - Example

Region	Start Address	End Address	Size	Bus Used	What Lives Here
Code	0x00000000	0x1FFFFFFF	512 MB	ICode + DCode	Flash, ROM, Bootloader, Constants
SRAM	0x20000000	0x3FFFFFFF	512 MB	System Bus	On-chip RAM — variables, stack, heap
Peripheral	0x40000000	0x5FFFFFFF	512 MB	System Bus	Memory-mapped peripheral registers
External RAM	0x60000000	0x9FFFFFFF	1 GB	System Bus	Off-chip SDRAM via FSMC / FMC
External Device	0xA0000000	0xDFFFFFFF	1 GB	System Bus	Off-chip peripherals, NOR Flash
Private Peripheral Bus	0xE0000000	0xFFFFFFFF	512 MB	PPB	NVIC, SysTick, CoreSight Debug

The Bus Routing Rule
How Cortex-M Decides Which Bus to Use

When the CPU generates a memory address, it routes it to ICode/Dcode/System Bus automatically:

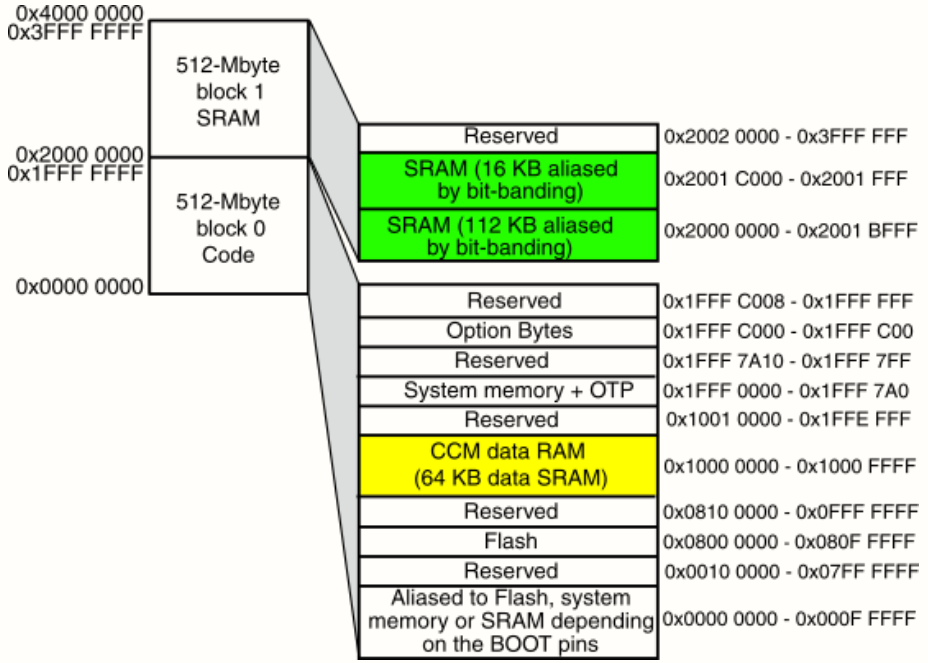
- Address 0x00000000 – 0x1FFFFFFF → ICode bus (if instruction fetch)

  → DCode bus (if data access)

- Address 0x20000000 – 0xDFFFFFFF → System bus (always)

Address 0xE0000000 – 0xFFFFFFFF → Private Peripheral Bus (always)

- The routing is HARDWARE — happens in one cycle, zero overhead. The CPU does not decide — the address itself decides.



ARM Cortex M Memory Map – Key Nuances

While the boundaries are architecturally defined, there is flexibility in the implementation:

Remapping: Some Cortex-M microcontrollers allow the vector table (and sometimes flash/SRAM) to be remapped to different locations, particularly during startup, allowing, for example, the RAM to appear at address 0x00000000.

Vendor Customization: While the memory map is standardized, silicon vendors can add their own peripheral sets in the designated peripheral regions

Memory Protection Unit (MPU): All Cortex-M cores (except the M0) offer optional MPUs. While the overall boundaries of the memory map (e.g., SRAM region) are fixed, the MPU allows you to partition these regions, change access permissions, or define custom memory attributes.

Memory Mapped I/O in ARM Cortex M

On x86 (desktop CPUs):

Separate IN / OUT instructions to talk to hardware. Hardware lives in a separate I/O address space

On ARM Cortex-M:

NO separate I/O instructions exist . Peripheral registers are just memory addresses. You read/write them exactly like RAM

Example — Toggle a GPIO pin on STM32F4:

```
// GPIOA Output Data Register is at address 0x40020014
*((volatile uint32_t*)0x40020014) = 0x00000001;
```

That is a normal memory write — to an address that happens to control a physical pin on the chip.

What the journey we just completed is the “Evolution of Memory Architecture for MCU”

Von Neumann → Harvard → Modified Harvard

Is the choice of Memory architecture the only way to decide the processing power of an MCU?

Quite a many things indeed..

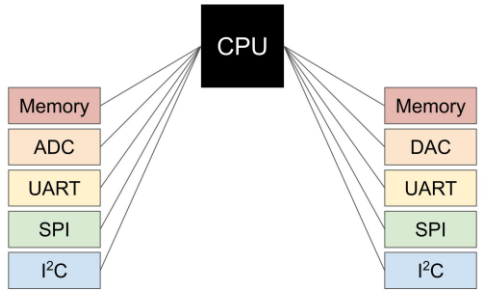
DMA, DSP & FPU, NVIC, Flash fetch Buffers, Cache, TCM, Bus Matrix.

DMA

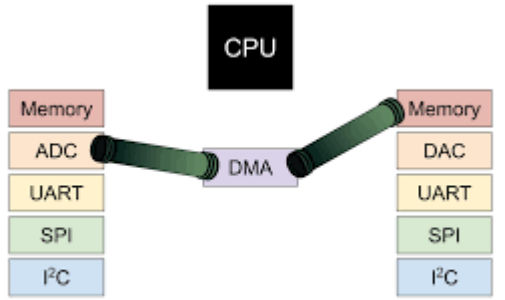
DMA — The Hidden throughput Multiplier

- When there is no DMA feature, the CPU must manage
  - “Data to” and “Data from” the peripherals like UART, CAN, Ethernet,
  - Array of Data from a high sampling speed ADC
  - Sending data to high resolution TFT
  - Memory to Memory data transfer operations
  - Enabling the peripherals can be one solution, but when there are too many interrupts, the CPU gets stuck and it affects the processing power.
- Direct Memory Access (DMA) is a hardware feature in a Microcontroller that acts as a configurable agent for data transfer.
- There can one or many DMAs available in an MCU
- Thus the MCUs with DMA Improves overall performance and efficiency of the CCU.

Without DMA



With DMA



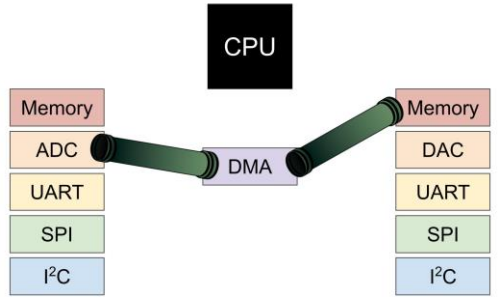


Image Courtesy: https://www.digikey.com/en/maker/projects/getting-started-with-stm32-working-with-adc-and-dma/f5009db3a3ed4370acaf545a3370c30c

With and Without DMA - Example

Without DMA at 168 MHz:

- ADC 3-channel at 1 Msps — 15% CPU.
- LCD 320×240 at 60 Hz — 23% CPU.
- UART logging at 115200 — 0.03% CPU.
- Total consumed by data movement — 38% CPU.
- Remaining for FOC algorithm — 62% CPU.

With DMA at 168 MHz:

- ADC DMA circular — 0% CPU.
- LCD DMA circular — 0% CPU.
- UART DMA normal mode — 0% CPU.
- Total consumed by data movement — ~0% CPU.
- Remaining for FOC algorithm — ~100% CPU.

The Real-World Impact:

- Having 100% CPU available for the FOC algorithm lets engineers run more complex algorithms, increase PWM frequency, add extra control loops, or handle Ethernet processing.
- DMA doesn't speed up the CPU but frees it up, making it more valuable.
- Not using DMA on high-data peripherals wastes CPU capacity already built into your system.

How DMA Works — Configure Once, Transfer Autonomously

- **Five Step DMA Configuration:**
  - Step 1 — **Set source address** — where data comes from. ADC data register, SRAM buffer, Flash address.
  - Step 2 — **Set destination address** — where data goes. SRAM buffer, UART data register, LCD frame buffer.
  - Step 3 — **Set transfer count** — how many bytes, halfwords, or words to move.
  - Step 4 — **Set transfer options** — increment source and/or destination, circular mode, burst size, priority level.
  - Step 5 — **Enable DMA stream** — DMA controller becomes an independent bus master. CPU walks away.
- **What Happens After Enable:**
- DMA controller generates AHB-Lite transactions on the bus matrix independently. Reads from source address. Writes to destination address. Decrements counter. When counter reaches zero — raises Transfer Complete interrupt. **CPU resumes only to process the completed buffer.**
- **CPU Activity During DMA Transfer:**
- CPU is completely free — running FOC algorithm, processing Ethernet packets, updating display logic — anything useful. Zero instructions consumed by data movement.
- **DMA turns the CPU from a data courier into a data processor — which is what you bought it for.**

Different modes of operation of DMA

Normal Mode: DMA performs the configured number of transfers then stops and raises a transfer complete interrupt. CPU must reconfigure and restart for the next batch. Used for one-shot transfers — firmware update, single block copy, one-time peripheral read.

Circular Mode: DMA automatically restarts from the beginning when it reaches the end of the buffer. Continuous operation — no CPU intervention needed between transfers. Used for continuous ADC sampling, continuous UART receive, continuous audio playback.

Circular mode with double buffer — the most powerful pattern for real-time systems:

Buffer A and Buffer B — same size. DMA fills Buffer A while CPU processes Buffer A's previous contents — then DMA switches to Buffer B while CPU processes Buffer A — then back to Buffer A while CPU processes Buffer B. Perfectly interleaved — zero gaps — CPU always has fresh data and always has time to process it.

FIFO Mode: DMA has an internal 4-word FIFO buffer. Instead of transferring immediately on each peripheral request — it accumulates data in the FIFO and transfers in a burst when FIFO reaches threshold. Reduces bus matrix occupancy because bursts are more efficient than single transfers. Used when bus bandwidth must be conserved — multiple DMA streams competing on same matrix.

Direct Mode: No FIFO — each peripheral request immediately triggers one transfer. Lower latency but higher bus overhead. Used when latency matters more than bus efficiency.

DMA Transfer Types

Transfer Type	Source	Destination	Real-World Example
Peripheral → Memory	ADC data register	SRAM sample buffer	Motor phase current sampling at 1 Msps
Memory → Peripheral	SRAM TX buffer	UART data register	Sending diagnostic log over serial port
Memory → Memory	SRAM source	SRAM destination	Frame buffer copy, memcpy acceleration
Peripheral → Peripheral	Timer output compare	DAC input register	Sine wave generation synchronized to PWM timer

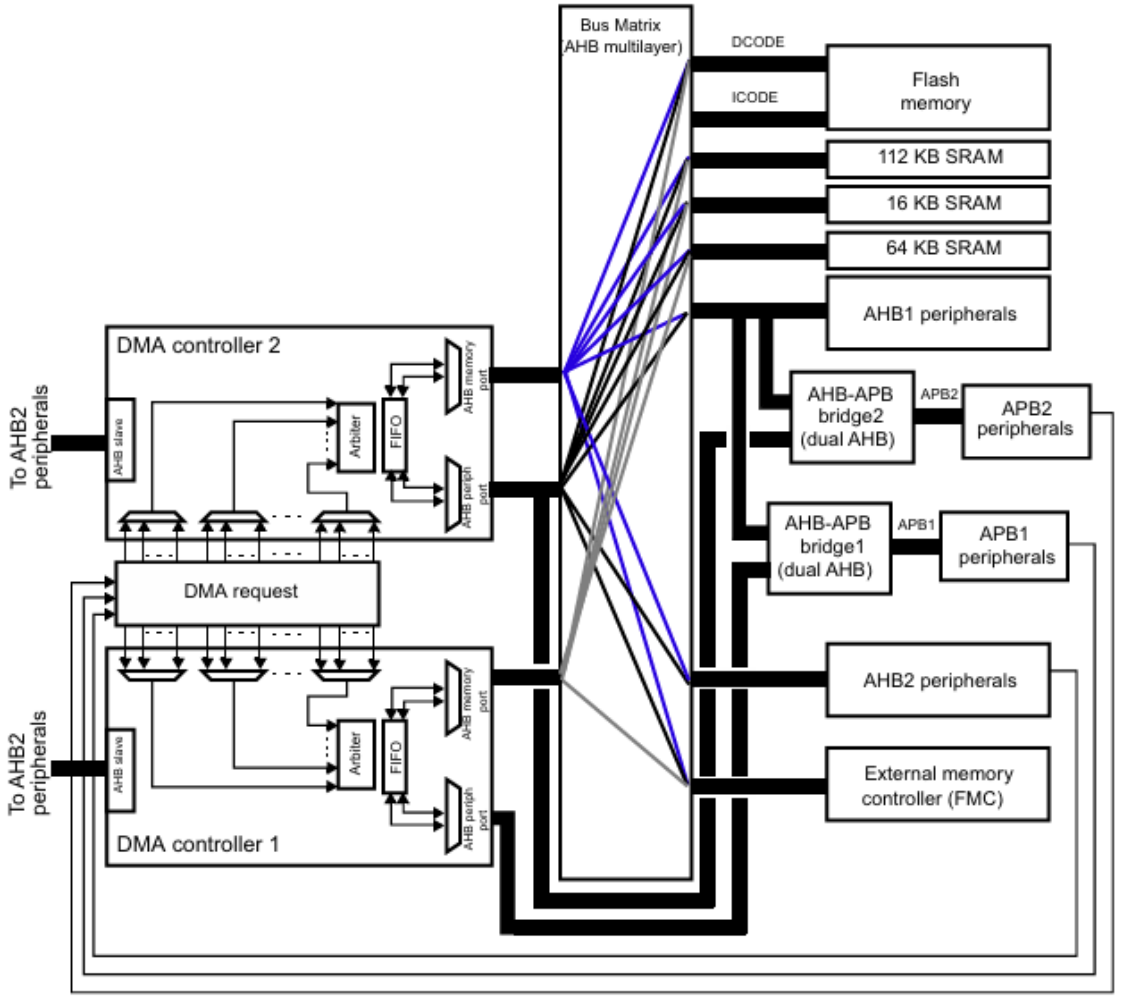
- **Key Constraints to Know:**
- Memory-to-memory transfers — only available on DMA2 on STM32F4. DMA1 cannot perform memory-to-memory.
- Peripheral-to-peripheral transfers — only possible when both peripherals are connected to the same DMA controller and are mutually compatible trigger sources.
- Flash as source — only DMA2 can access Flash on STM32F4. If copying lookup tables or font data from Flash to SRAM at startup — must use DMA2.

Always verify which DMA controller can reach both your source and destination before assigning peripherals — the mapping is fixed in silicon and cannot be changed in software.

Example of DMA Hardware on STM32F4

DMA1 vs DMA2 — Critical Differences:

Capability	DMA1	DMA2
Access SRAM1, SRAM2	✅ Yes	✅ Yes
Access Flash	❌ No	✅ Yes
Access AHB2 (USB OTG HS)	❌ No	✅ Yes
Memory-to-memory transfers	❌ No	✅ Yes
Access APB1, APB2 peripherals	✅ Yes	✅ Yes



This mapping is fixed in silicon. Full mapping in RM0090 Figures 33, 34.
https://www.st.com/resource/en/reference_manual/rm0090-stmicroelectronics.pdf

Summary – DMA

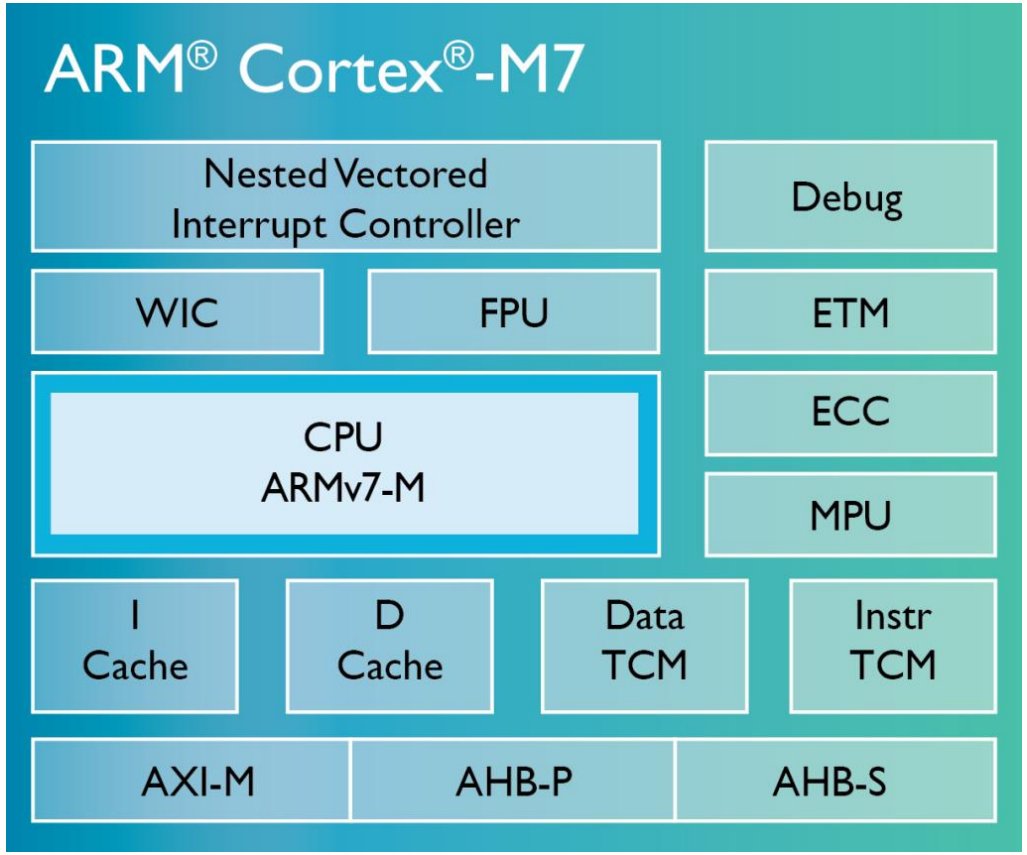
Role in MCU :

- Controller that performs memory/peripheral data transfers independently of the CPU.

Impact on Overall MCU Computing Performance

- Frees the CPU from repetitive data movement tasks (often saving 40–70% CPU cycles).
- Improves overall system throughput, reduces power consumption, and enables efficient handling of high-bandwidth peripherals (ADC, USB, Ethernet, audio).

DSP Extension & Floating Point Unit



Additional Hardware enhancements in ARM Cortex-M7

The ARM Cortex-M7 FPU and DSP extension are primarily hardware features that implement new instructions (instruction set extensions). The FPU is a hardware unit for floating-point calculations, while the DSP extension adds SIMD instructions and a single-cycle MAC (Multiply-Accumulate) unit.

Key Details

FPU (Floating Point Unit): This is a dedicated hardware module that supports single and optional double-precision IEEE 754 arithmetic.

DSP Extensions: These are specialized hardware features, such as single-cycle 16/32-bit MAC (Multiply-Accumulate) and SIMD instructions, which speed up algorithms.

MAC (Multiply-Accumulate): The MAC is a core part of the DSP hardware extension, enabling single-cycle multiply and accumulate operations.

The FPU offers optional single/double precision, while the DSP extension is standard, **both designed to speed up numeric calculations without external DSP chips**

Summary – DMA

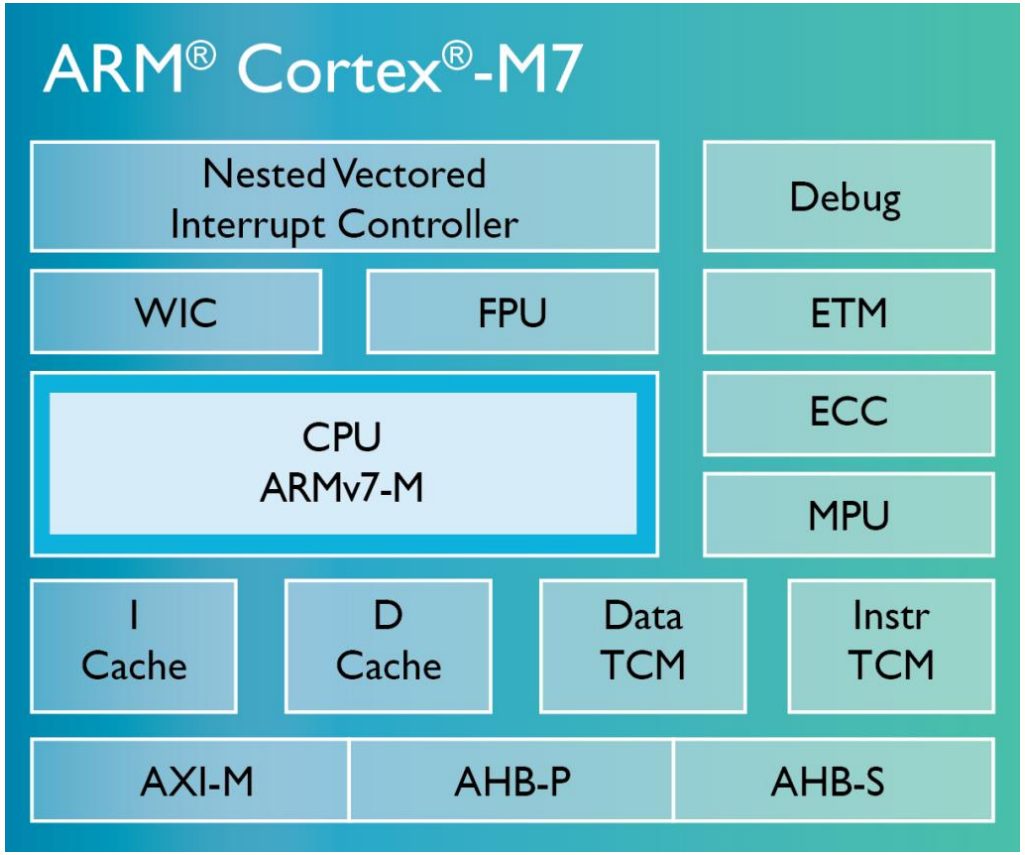
Role in MCU :

- DSP: Specialized instructions and hardware (MAC, SIMD, saturation) for signal processing.
- FPU: Dedicated hardware for floating-point arithmetic.

Impact on Overall MCU Computing Performance

- Provides order-of-magnitude speedup (10–20× or more) for math-heavy and signal-processing tasks compared to software emulation.
- Enables real-time audio, motor control, image processing, and complex algorithms on a single MCU.

NVIC


ARM® Cortex®-M7
Nested Vectored Interrupt Controller
Debug
WIC
FPU
ETM
CPU ARMv7-M
ECC
MPU
I Cache
D Cache
Data TCM
Instr TCM
AXI-M
AHB-P
AHB-S

NVIC Controller in ARM

The Nested Vectored Interrupt Controller (NVIC) is a mandatory, closely integrated hardware block within the ARM Cortex-M processor core, not a separate external peripheral. It handles all interrupts and system exceptions, supporting low-latency response, nested interrupts, prioritization, and efficient tail-chaining.

Key Hardware Features of the NVIC:

- **Low-Latency Exception Handling:** Tightly coupled to the CPU core, it provides fast interrupt response and handles state saving (stacking) and restoring automatically, reducing overhead.
- **Interrupt Nesting:** Supports nested interrupts, allowing higher-priority interrupts to preempt currently running lower-priority Interrupt Service Routines (ISRs).
- **Programmable Priority Levels:** Allows assigning priorities and sub-priorities, determining which interrupt to handle first if multiple occur simultaneously.
- **Vector Table Management:** Manages the vector table, which is a lookup table of memory addresses for interrupt handlers.
- **Tail-Chaining:** Optimizes the transition between two pending interrupts, avoiding unnecessary stack operations.
- **Support for Many Interrupts:** Depending on the Cortex-M implementation it can support up to 240+ interrupt sources.

Summary – NVIC

Role in MCU :

- ARM’s advanced interrupt controller supporting prioritized, nested interrupts with automatic vectoring and tail-chaining.

Impact on Overall MCU Computing Performance

- Dramatically reduces interrupt latency and context-switch overhead.
- Enables highly responsive, deterministic real-time behavior with minimal CPU overhead for interrupt handling.

Instruction Pipelining

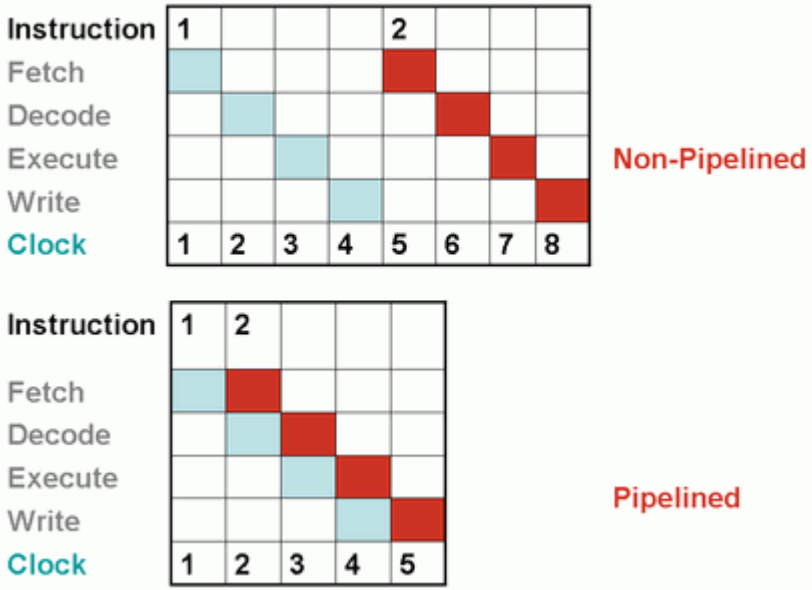
Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Instruction Pipelining

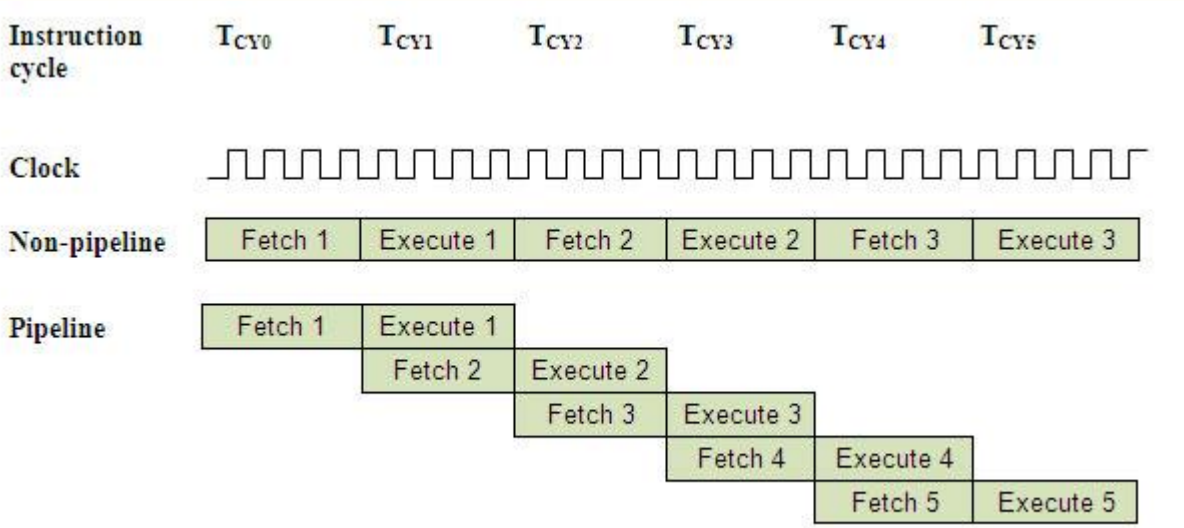
Instruction Pipelining is a method of improving the CPU's execution throughput by overlapping instruction fetch and execution.

Von Neumann Pipeline support : Possible, but suffers from Von Neumann Bottleneck (fetch & data access compete)
Harvard Pipeline support : Excellent — simultaneous instruction fetch + data access

2 Stage Pipeline example in Von-Neumann



2 Stage Pipeline example in PIC16 Harvard Architecture



Instruction Pipelining in ARM Cortex M MCUs

Cortex-M Core	Architecture	Pipeline Stages	Note
Cortex-M0	Armv6-M	3	Fetch, Decode, Execute
Cortex-M0+	Armv6-M	2	Fetch/Pre-decode, Decode/Execute (Reduced power)
Cortex-M1	Armv6-M	3	Designed specifically for FPGAs
Cortex-M3	Armv7-M	3	Standard 3-stage Harvard architecture
Cortex-M4	Armv7E-M	3	Same as M3 but adds DSP/FPU units
Cortex-M7	Armv7E-M	6	Dual-issue superscalar pipeline
Cortex-M23	Armv8-M Baseline	2	TrustZone equivalent of M0+
Cortex-M33	Armv8-M Mainline	3	TrustZone equivalent of M4
Cortex-M35P	Armv8-M Mainline	3	M33 with physical security (anti-tamper)
Cortex-M52	Armv8.1-M	4	Optimized for Helium (MVE)
Cortex-M55	Armv8.1-M	4 / 5	4-stage Integer; 5-stage Floating Point/Vector
Cortex-M85	Armv8.1-M	7 / 9	7-stage Integer; 9-stage Floating Point/Vector

Things to Remember:

- Pipelining increases CPU throughput by overlapping instruction execution, but it does not fix the speed disparity between the fast CPU and slow main memory (Flash).
- Cache memory is essential to reduce memory latency by storing frequently used data/instructions close to the core.
- ARM Cortex M MCU CPU uses Modified Harvard Architecture.

Summary – Instruction Pipelining

Role in MCU :

- Overlaps multiple stages of instruction processing (fetch, decode, execute, etc.) so several instructions advance in parallel.

Impact on Overall MCU Computing Performance

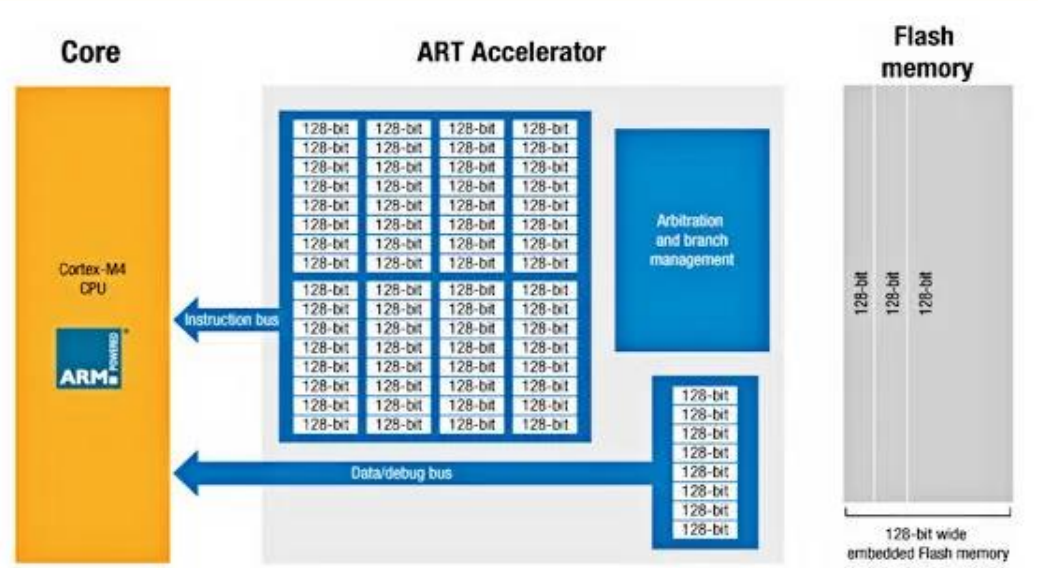
- Increases instruction throughput (often 1–2+ instructions per cycle) without increasing clock frequency. Major contributor to overall CPU performance; pipeline hazards are managed with forwarding and branch prediction.

We just discussed Flash is slow…

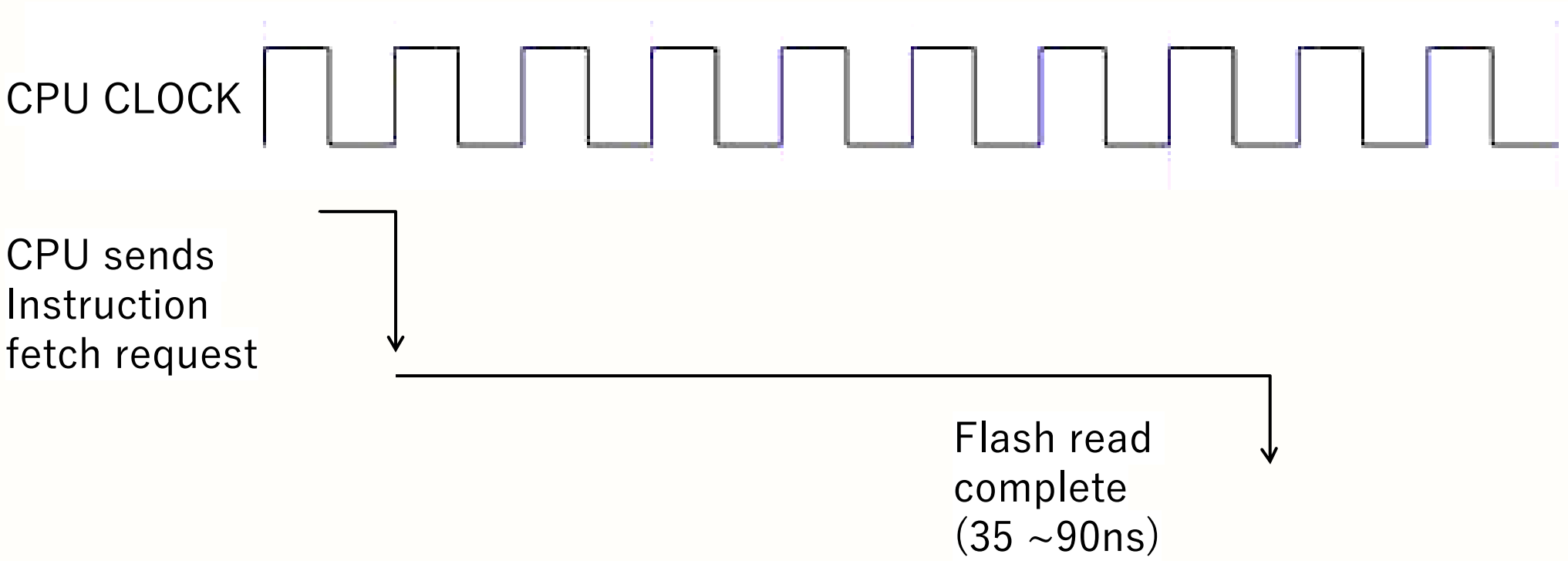
What does it mean?

Let’s try to understand the issue and the way it is mitigated.

Flash Wait states and Instruction fetch buffers



Why Memory – Not Clock Speed – Is the Bottleneck?



The Flash Wait State Calculation (STM32F4 Example)

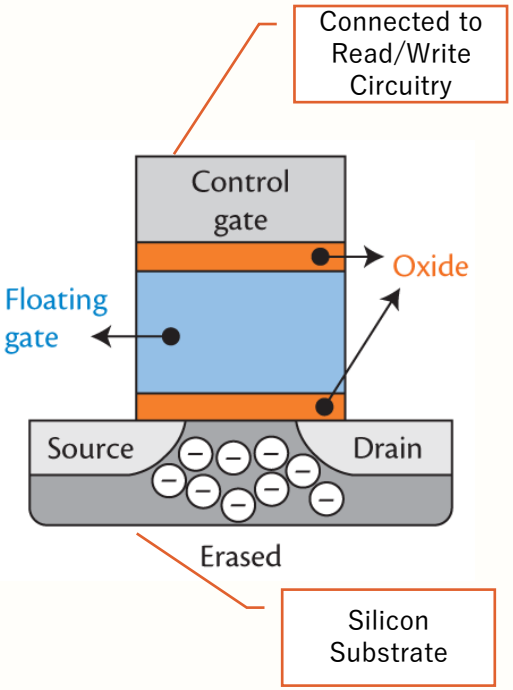
CPU Clock	CPU Cycle Time	Flash Access Time	Wait States Required	Efficiency
24 MHz	41.7 ns	35–90 ns	0 WS	CPU slightly slower than Flash
48 MHz	20.8 ns	35–90 ns	1 WS	One extra cycle to wait
84 MHz	11.9 ns	35–90 ns	2–3 WS	Flash takes 3–8 CPU cycles
168 MHz	5.95 ns	35–90 ns	5 WS	Flash takes 6–15 CPU cycles

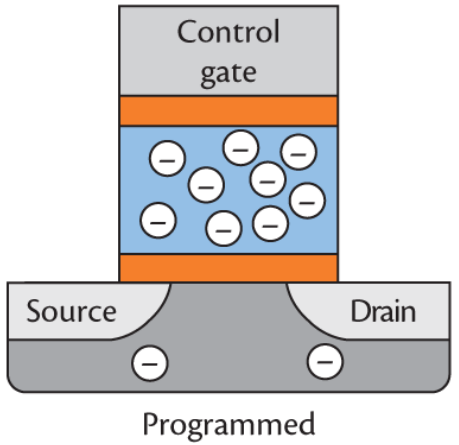
The faster you clock the CPU, the more wait states you need.

The physics of Flash sensing does not change — only the CPU gets faster.

Why Flash is slower than CPU

Reason	Explanation
Different Technology	• CPU is built from SRAM-based flip-flops and logic gates — switching in < 1 ns. • Flash uses floating-gate transistors that **require charge tunneling to read (35 ns to 90 ns)** — fundamentally slower physics.
Read Access Time	• Modern NOR Flash (used in MCUs) has a typical read access time of **35 ns to 90 ns.** • At 168 MHz the CPU needs a new value every **5.95 ns** — Flash is 6 to 15× too slow.
Why Not Just Use Fast SRAM?	• SRAM is 10–50× more expensive per bit and consumes significantly more silicon area and power. • **A 1 MB SRAM Flash replacement would make the MCU impractically large and expensive.**





- Clock speed = how fast instructions can be ISSUED
- Memory architecture = how fast they ARRIVE at CPU
- Flash wait states + bus contention = real bottleneck
- Same MHz, different memory = very different results

The Three Flash Operations and timings

ERASE

Apply HIGH voltage (~12–20V) between control gate and substrate

Electrons tunnel OUT of floating gate back to substrate (Fowler-Nordheim tunneling — reverse direction)

Result: Floating gate empty = bit is 1

Time: 1 ms to 100 ms per sector

Granularity: Entire SECTOR at once (cannot erase single byte)

Endurance: ~10,000 to 100,000 cycles

PROGRAM

Apply HIGH voltage (~12V) between control gate and channel

Electrons tunnel FROM channel ONTO floating gate (Fowler-Nordheim tunneling)

Result: Floating gate charged = bit is 0

Time: 10 µs to 100 µs per word

Granularity: Word or page at a time

Constraint: Can only write to ERASED (all 1s) locations

READ

Apply READ voltage to control gate (~3.3V or 5V depending on process)

Sense amplifier detects whether current flows through transistor:

Current flows → Bit = 1 (erased)

No current → Bit = 0 (programmed)

Time: 35 ns to 90 ns per access

← THIS is what causes wait states

Granularity: Typically 64-bit or 128-bit wide read bus

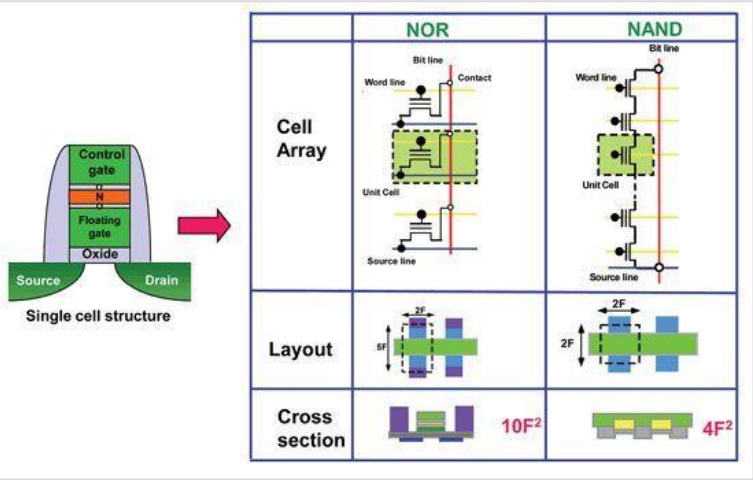
Note that, in a primary battery operated application, every firmware upgrade consumes energy from the battery, reducing the overall life time for a given battery.

NOR vs NAND Flash – Which is used in MCUs?

NOR Flash	NAND Flash
Random access Any address directly	Sequential / page access Must read entire page (~512B–4KB)
Fast read (35–90ns) Slow write/erase	Slower read (~25µs page read) Fast write/erase
Execute-in-place (XIP) possible	Cannot execute directly Must copy to RAM first
Used in MCUs for program storage	Used in SSDs, USB drives, smartphones for bulk storage
64 KB – 2 MB	8 GB – 2 TB

MCUs use NOR Flash, as Random access and execute-in-place are non-negotiable requirements for running firmware.

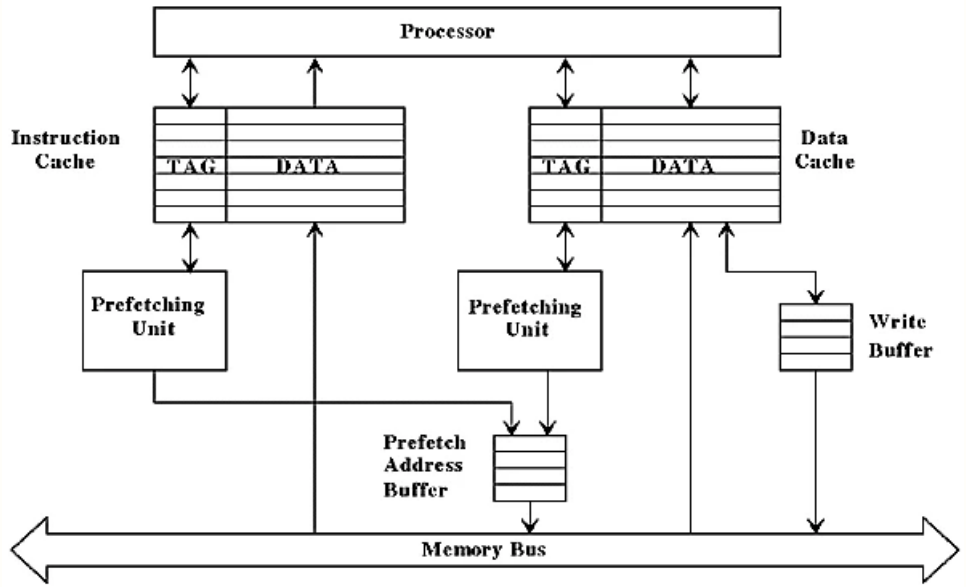
NOR flash connects cells in parallel (like a NOR gate) for fast random access, while NAND flash connects cells in series (like a NAND gate) for higher density, faster write/erase, and lower cost.



How do vendors mitigate the wait state penalty?

Solution 1 — Prefetch Buffer Read ahead of the CPU — ex: fetch the next 128 bits (16bytes) while the CPU is still executing the current ones. Zero cost when code runs sequentially. (In ARM CPU, ARM instructions are 32 bits wide and Thumb instructions are 16 bits wide. So, the prefetch buffer essentially helps to hold 4/8 instructions)

Solution 2 — Instruction Cache (I-Cache) Store recently fetched Flash lines in fast SRAM. On cache hit → zero wait states. On cache miss → full wait state penalty.



Solution 3 — Tightly Coupled Memory (TCM) Copy critical code to zero-wait-state SRAM. Execute from SRAM instead of Flash. Zero wait states — always — for that code.

ST uses Solution 1 + 2 (ART Accelerator) in some and adds solution 3 as well in M7, M85 cores

Infineon uses Solution 1 + 2 (Prefetch + Cache) in XMC4000

ARM provides Solution 2, 3 (TCM) on M7 and M85

Case Study: How ST Mitigates Flash Latency
The ART Accelerator (STM32F4)

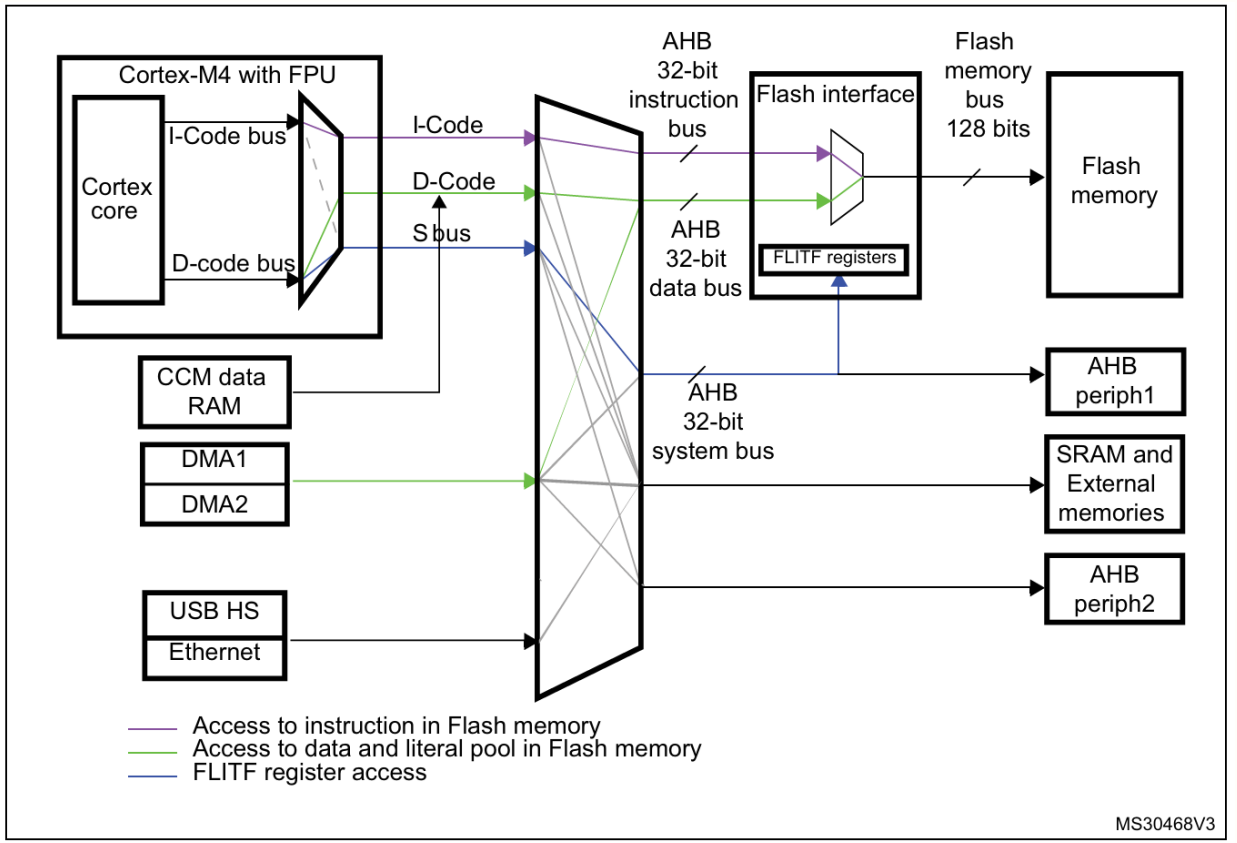
ST's Adaptive Real-Time (ART) Accelerator is a combined prefetch buffer and instruction cache built into the STM32F4 Flash controller

What ART Stands For and Where It Lives

ART = Adaptive Real-Time Accelerator

It is not inside the Cortex-M4 CPU core. It sits between the CPU's ICode bus and the physical Flash array — inside ST's Flash interface controller.

ARM does not provide it — ST designed and added it themselves.



What ST's ART contains?

Component	Size	Purpose
Instruction Prefetch Buffer	128 bits (4 × 32-bit words)	Fetches next Flash line before CPU needs it
Instruction Cache Lines	64 lines × 128 bits = 1 KB	Stores recently fetched instruction lines
Data Prefetch Buffer	64 bits	Prefetches literal pool and constant data
ART Control Logic	—	Decides when to prefetch, manages cache lines

Without ART Accelerator:

CPU ICode Bus → Flash Array (35–90 ns access) → CPU gets instruction → 5 wait states at 168 MHz

With ART Accelerator:

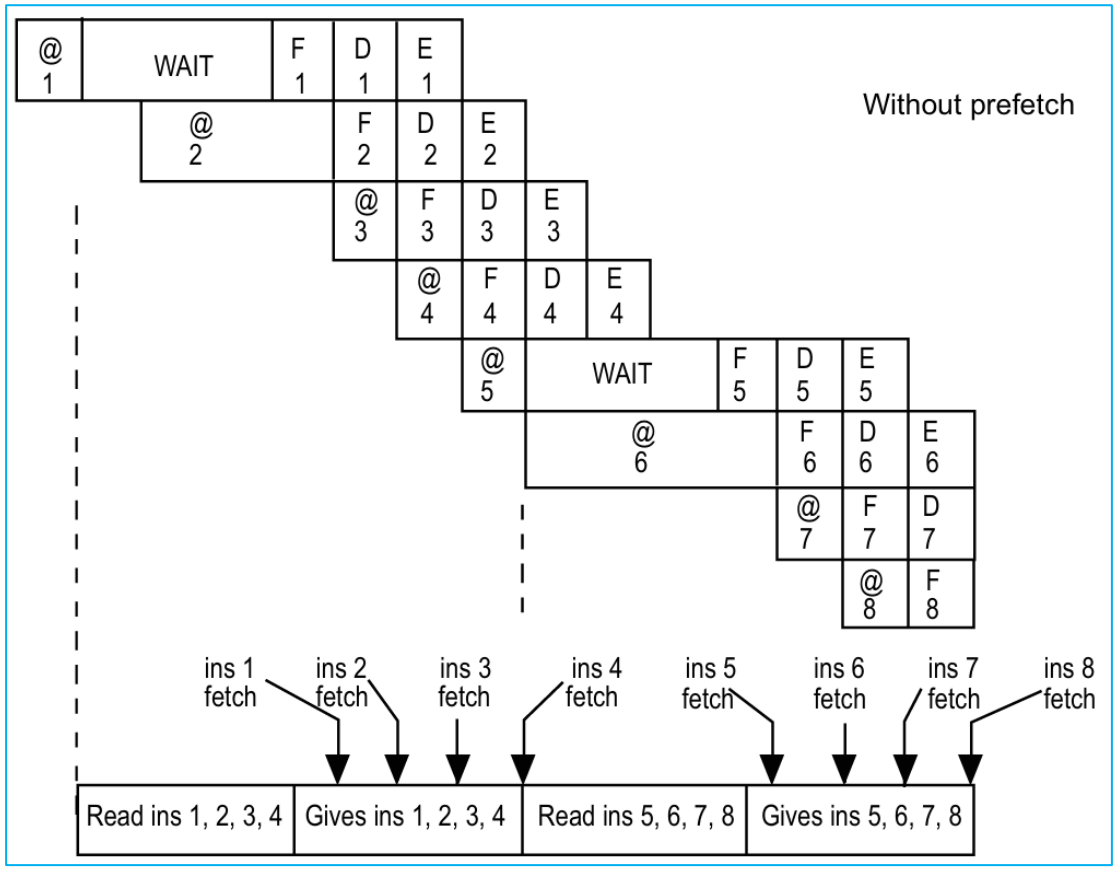
CPU ICode Bus → ART Accelerator → IF hit: instruction delivered in 1 cycle, zero wait states → IF miss: ART fetches from Flash (5 WS), stores in buffer, delivers to CPU

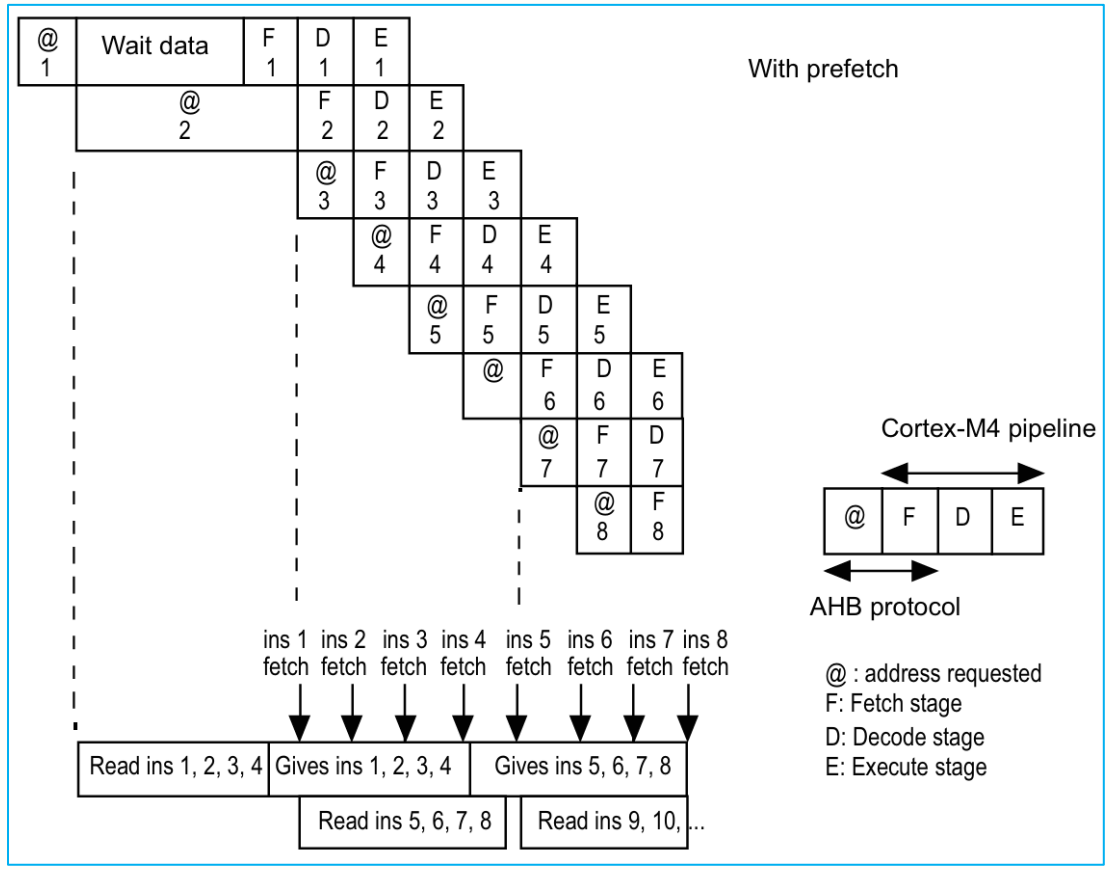
The ART Accelerator sits in the middle and acts as a speed adapter between the fast CPU and the slow Flash.

Along with Instruction Cache, total ART Accelerator storage = effectively 1 Kb of zero-wait-state instruction buffer

How ST's ART works?

While the CPU is executing the current 128-bit block of instructions, the ART Accelerator is already reading the next 128-bit block from Flash in the background. By the time the CPU finishes the current block and asks for the next one — it is already waiting in the buffer. Zero wait states experienced by the CPU.





ART Accelerator Performance Impact

Condition	Clock	CoreMark	CoreMark/MHz	Source
ART fully ON — Flash execution	168 MHz	566	3.37	ST official datasheet
ART fully ON — Flash execution	84 MHz	285	3.39	ST official presentation
ART fully ON — Flash execution	180 MHz	608	3.38	ST official website
ART OFF — Flash execution	168 MHz	~280–300 (estimated)	~1.7–1.8	Community reports — up to 2× degradation — no official number
SRAM execution — no Flash	168 MHz	~570	~3.39	Theoretical — no ART needed

ART Accelerator Limitations – When it does NOT help?

“ART Accelerator = Prefetch Buffer + Instruction Cache + Data Cache”

Situation 1 — Dense branch code: Code with many if/else branches, switch/case statements, or function pointer calls frequently jumps to new addresses. Every jump that misses the 8 cache lines pays the full 5 WS penalty. Cache hit rate drops.

Situation 2 — Large code with poor locality: If the active code footprint exceeds 64 cache lines × 128 bits = 1 KB, cache lines get evicted and refetched repeatedly. Hit rate degrades.

Situation 3 — Interrupt-heavy systems: Every interrupt entry jumps to the ISR vector — a cache miss. Every interrupt return jumps back to interrupted code — potentially another miss. High interrupt rate = high cache miss rate.

Situation 4 — Data cache coherency with DMA: If DMA writes to Flash-mapped constants that are cached in the data cache, the CPU reads stale data. Must invalidate data cache after DMA operations. A subtle but serious bug source.

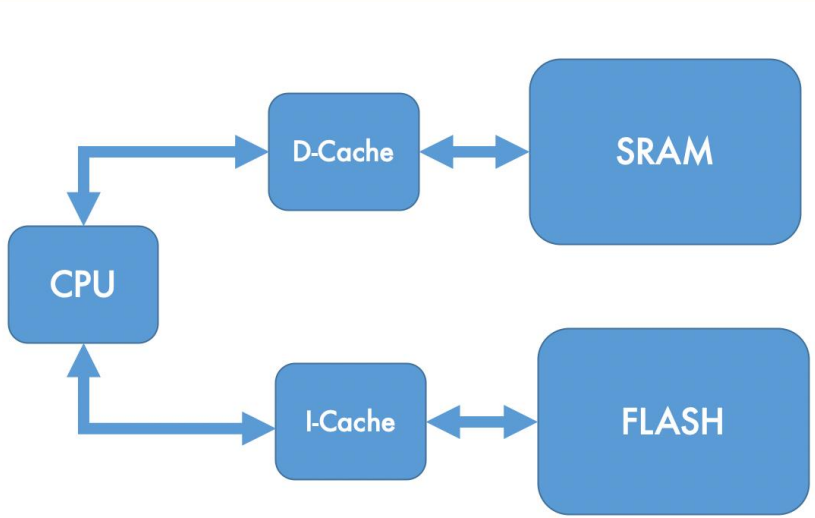
We learnt that MCU vendors have their own way of dealing with Flash wait cycle issue in architectures like ARM Cortex M3.

What is the solution from ARM for this problem?

It is Cache and TCM.

Let’s investigate the details.

Cache Memory



Cache memory

Cache memory is a small, high-speed storage buffer (SRAM) located between the CPU and slower main memories like Flash or external RAM.

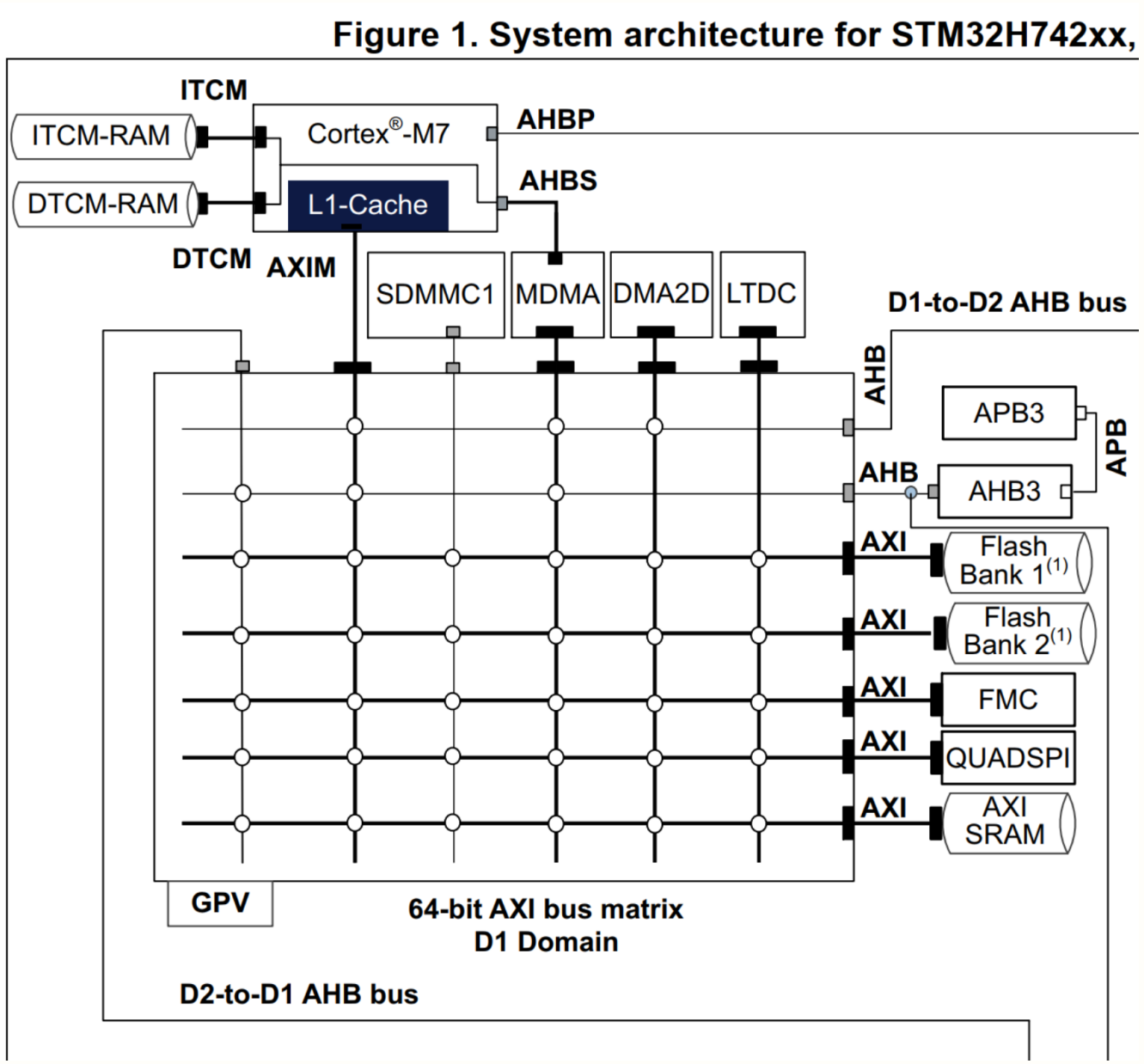
It is designed to bridge the speed gap, allowing the CPU to execute code at its maximum frequency without waiting for slow Flash memory accesses, which often require multiple "wait states".

The Cortex-M7 features a Harvard Architecture cache, and is split into two independent parts:

Instruction Cache (I-Cache): Stores recently fetched instructions to speed up code execution.

Data Cache (D-Cache): Stores recently accessed data to accelerate reads and writes.

Typical sizes range from 4KB to 64KB, organized into 32-byte cache lines.



How Cache gets filled, Refreshed, How CPU uses it

How it Gets Filled

The cache is filled through a process called Read-Allocate on a cache miss:

1. Request: The CPU tries to read an address.
2. Miss: The cache controller checks its "Tags" (stored addresses) and finds the data is not present.
3. Fetch: The controller fetches an entire 32-byte block (a "line") from Flash or RAM into a Linefill Buffer.
4. Fill: This entire block is then written into the cache. This follows the principle of spatial locality—if you need one instruction, you'll likely need the next few soon.

How it Gets Refreshed

The cache does not "refresh" automatically like DRAM; it must be managed by the developer or hardware policies:

1. Eviction: When the cache is full and new data is needed, an existing line is kicked out (evicted) based on a "Least Recently Used" (LRU) algorithm.
2. Invalidation: You manually tell the CPU that the cache content is no longer valid (e.g., after a DMA transfer updates main memory) so the CPU fetches fresh data.
3. Cleaning: For the D-Cache, "cleaning" writes "dirty" data (modified by the CPU but not yet in RAM) back to the main memory.

How the CPU Knows Where to Go

The CPU always asks the Cache Controller first. It does not "choose" between Flash and Cache in a manual sense; rather:

1. The CPU issues a memory address.
2. The Cache Controller performs a Lookup by comparing the address against its internal Tags.
3. Hit: If the tag matches, the controller intercepts the request and provides the data instantly from the cache.
4. Miss: If no tag matches, the controller transparently routes the request to the AXI/AHB bus to fetch it from Flash.

Key Details on Cortex-M7 Cache Management:

- **Memory Mapped Registers:** Cache control occurs via memory-mapped control registers in the PPB (Private Peripheral Bus) space, not by mapping the actual cache RAM lines into the general memory space.
- **Cache Maintenance:** Software uses specialized CMSIS functions to perform invalidation (for I-cache/D-cache) and cleaning (D-cache) by writing to these registers.
- **MPU Configuration:** The Memory Protection Unit (MPU) defines which regions are cacheable (Write-Through or Write-Back).
- **Harvard Architecture:** It has separate Instruction Cache and Data Cache (optional ECC).
- **Cache vs. TCM:** While the cache is managed memory-mapped, it differs from Tightly Coupled Memory (TCM), which is physically mapped to high-speed address space.

Tightly Coupled Memory (TCM)

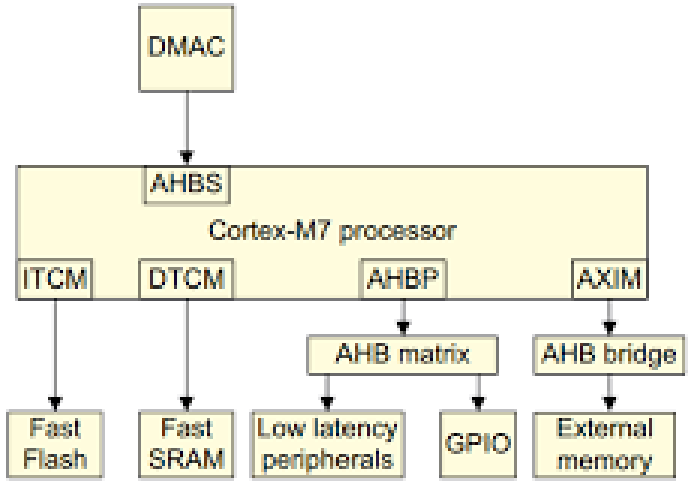


Figure 1-1 Example Cortex-M7 system

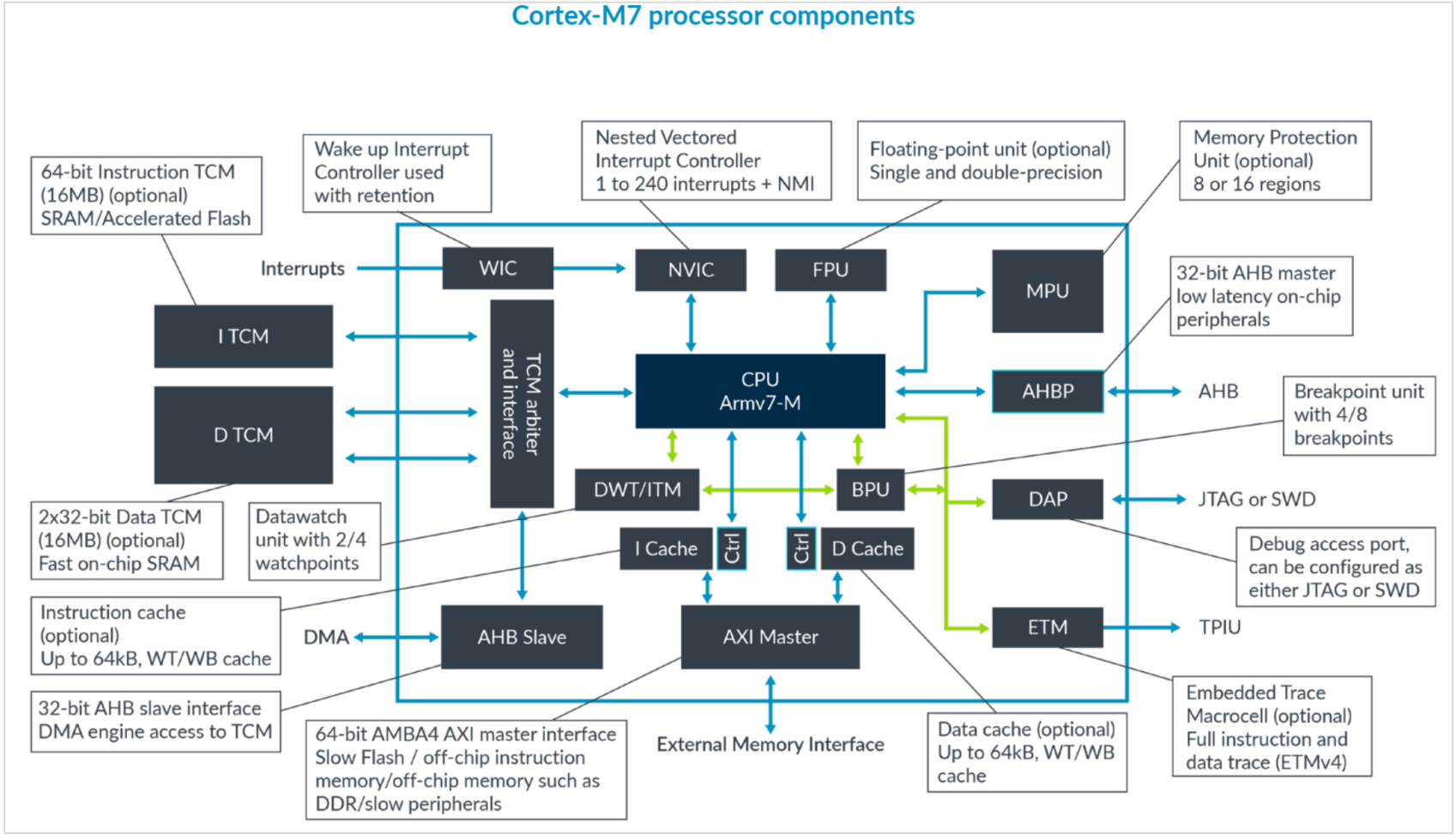
Tightly Coupled Memory (TCM)

Zero Wait State Performance

TCM is dedicated memory wired **directly into the CPU core pipeline — bypassing the bus matrix, bypassing the cache, bypassing the Flash controller entirely.** Zero wait states. Always. No exceptions.

ITCM — Instruction Tightly Coupled Memory: (Best used for: Time-critical interrupt service routines, motor control inner loops, PID computation routines, real-time signal processing kernels.)

DTCM — Data Tightly Coupled Memory: (Best used for: Real-time data buffers, motor control state variables, PID coefficients, DMA transfer staging areas, interrupt handler local data.)



Ex: STM32H743: ITCM – 64KB; DTCM – 128 KB; I-Cache – 16KB; D-Chache – 16KB

How to Use TCM in Practice

Using TCM requires two steps — telling the linker where to place the code, and copying it there at startup.

- In the linker script — add a ITCM region: ITCM (rwx) : ORIGIN = 0x00000000, LENGTH = 64K
- In your C code — mark the function with a section attribute:
  attribute((section(".itcm_code"))) void FOC_ControlLoop(void) { ... }
- At startup in your initialization code — copy the ITCM section from Flash to TCM:
  memcpy((void*)0x00000000, &_itcm_load_start, &_itcm_size);

After this the CPU executes FOC_ControlLoop from ITCM at zero wait states regardless of what the rest of the system is doing.

This is standard practice in high-performance motor control firmware. ST's Motor Control SDK (MCSDK) does this automatically for the FOC kernel when targeting STM32H7.

TCM Vs Cache differences

- **Determinism:** TCM guarantees fixed-latency access (1 cycle), essential for real-time applications. L1-Cache latency varies (1-3+ cycles) based on hits or misses, which is non-deterministic.
- **Management:** TCM is static; developers manage its contents via the linker script. Cache is dynamic; the controller automatically moves frequently accessed data from flash/SRAM into cache.
- **Mapping:** TCM has a fixed memory address space, often used for interrupt routines (ISRs) or critical data structures. Cache does not have a fixed, software-assigned mapping.
- **Architecture:** TCM (ITCM/DTCM) allows simultaneous access by the CPU and DMA, which is critical for moving data without halting the CPU. L1 Cache is purely for CPU acceleration.

Which ARM Cortex-M cores Have TCM

Core	ITCM	DTCM	Typical Size	Notes
Cortex-M0	❌	❌	—	No TCM
Cortex-M0+	❌	❌	—	No TCM
Cortex-M3	❌	❌	—	No TCM
Cortex-M4	❌	❌	—	No TCM — ART or cache only
Cortex-M7	✅	✅	16–512 KB each	TCM + I-Cache + D-Cache
Cortex-M23	❌	❌	—	No TCM
Cortex-M33	❌	❌	—	No TCM
Cortex-M55	❌	❌	—	No TCM — Helium + cache
Cortex-M85	✅	✅	Implementation defined	TCM + cache + Helium

Summary – Instruction Fetch Buffers, Cache, TCM

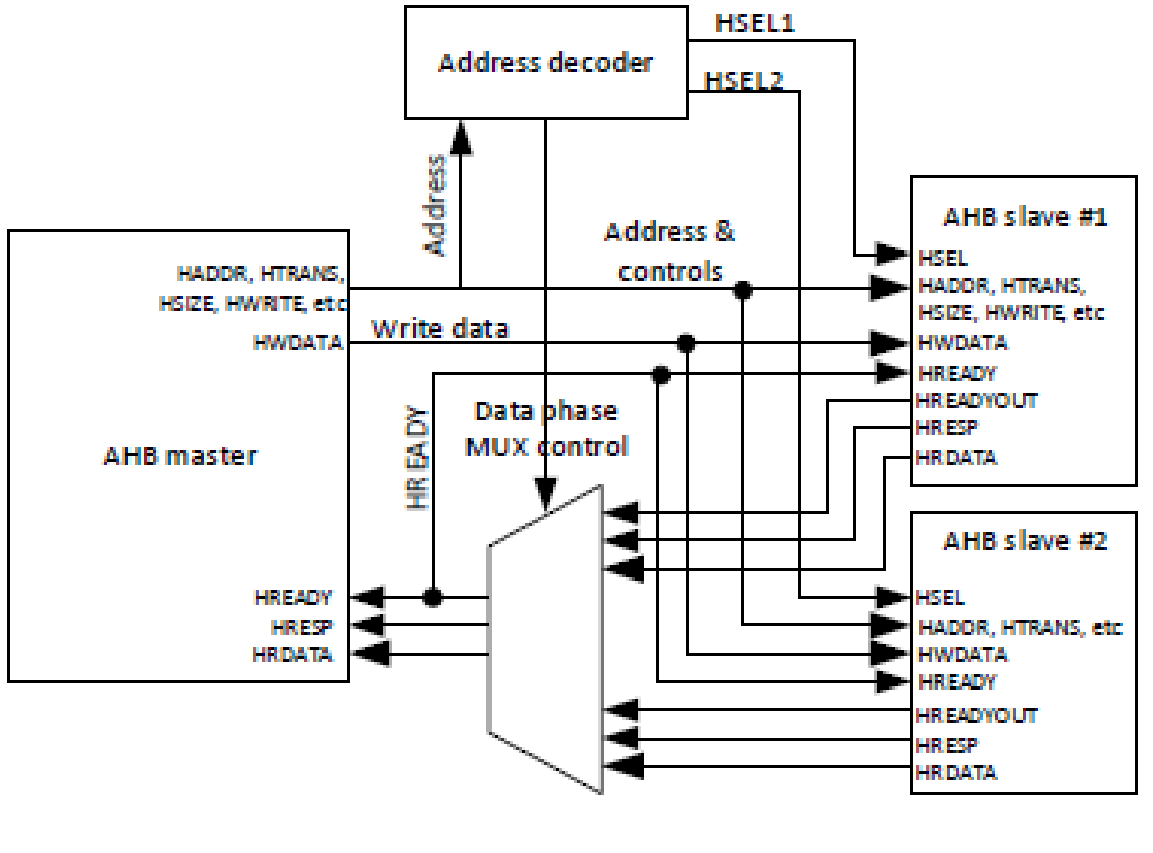
Role in MCU :

- Fetch buffers/prefetch: Pre-load upcoming instructions.
- Cache: Fast on-chip memory for frequently used code/data.
- TCM: Tightly Coupled Memory – dedicated single-cycle SRAM directly connected to the core.

Impact on Overall MCU Computing Performance

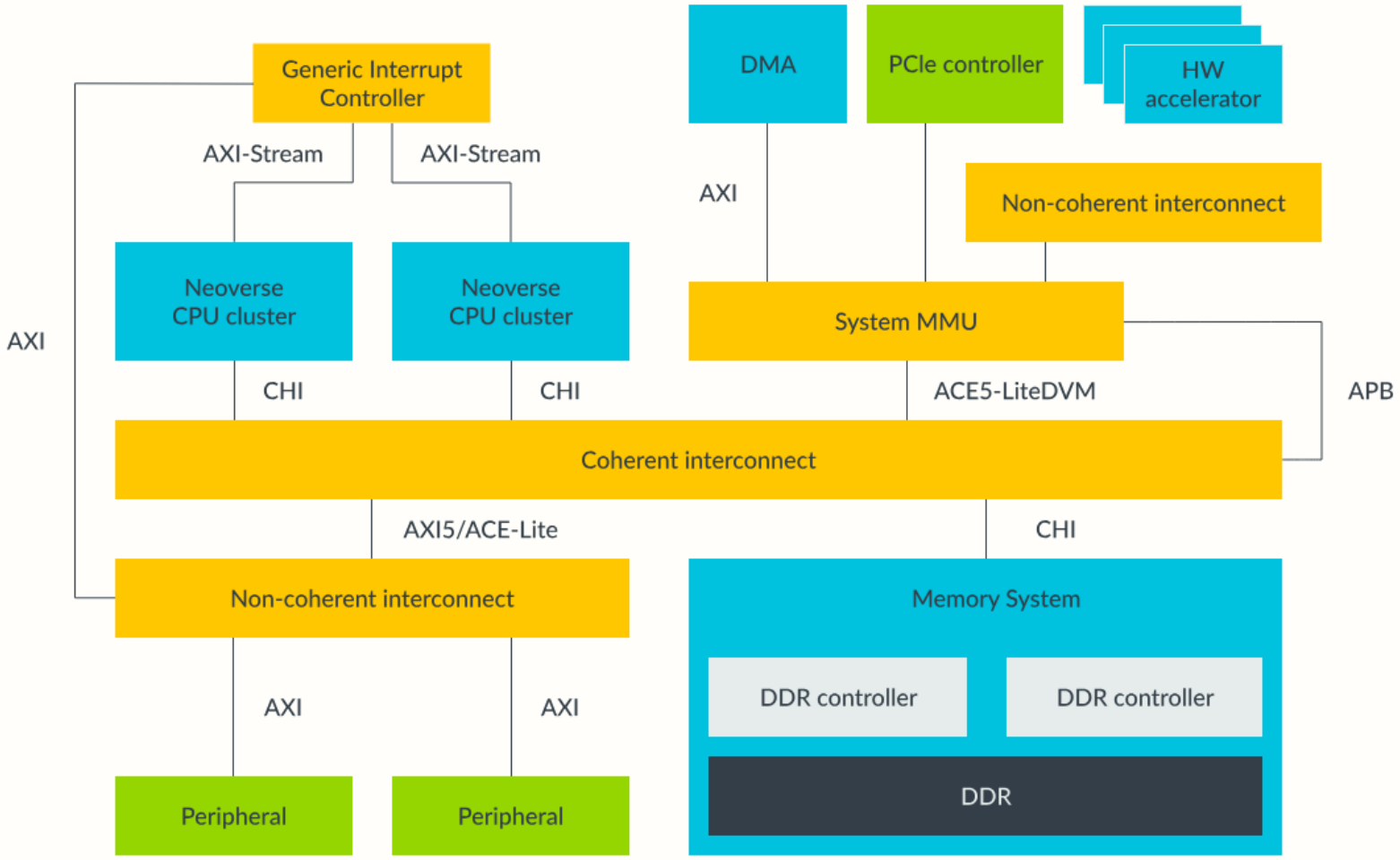
- Greatly reduces memory access stalls.
- Cache improves average-case speed via locality.
- TCM guarantees deterministic, single-cycle access for critical code and data.
- Together they maximize effective performance and predictability in real-time systems.

Bus Protocol



AMBA Bus Protocols — The On-Chip Communication Fabric

- The Advanced Microcontroller Bus **Architecture (AMBA) is an ARM-developed, open-standard on-chip interconnect specification** for managing connections between functional blocks in SoC designs.
- AMBA protocols define communication methods between these blocks.
- The diagram illustrates an SoC with multiple functional blocks using AMBA protocols such as AXI, CHI, and APB to interact.



How has AMBA evolved?

Key AMBA specifications		AMBA	AMBA 2	AMBA 3	AMBA 4	AMBA 5
CHI Coherent Hub Interface	A credited coherency protocol Layered architecture for scalability					CHI
ACE AXI Coherency Extensions	A superset of AXI — system-wide coherency across multicore clusters				ACE +Lite	ACE5 +Lite
AXI Adv. eXtensible Interface	AXI supports separate A/D phases, bursts, multiple outstandings addresses, and OoO responses			AXI3	AXI4 +Lite, +Stream	AXI5
AHB Adv. High-performance Bus	AHB suports 64 and 128 bit multi-manager AHB-Lite is for single managers		AHB	AHB +Lite		AHB5 +Lite
APB Advanced Peripheral Bus	System bus for low bandwidth peripherals	APB	APB2	APB3	APB4	

List of AMBA family protocols

Protocol	Full Name	Introduced	Key Characteristic
APB	Advanced Peripheral Bus	AMBA 1.0 — 1996	Simple, low power, for slow peripherals
AHB	Advanced High-performance Bus	AMBA 2.0 — 1999	Pipelined, burst transfers, multiple masters
AHB-Lite	AHB Lite	AMBA 3 — 2003	Simplified AHB — single master only
AXI	Advanced eXtensible Interface	AMBA 3 — 2003	High performance, separate read/write channels
AXI4-Lite	AXI4 Lite	AMBA 4 — 2010	Simplified AXI — no burst, simple register access
ACE	AXI Coherency Extensions	AMBA 4 — 2010	Cache coherent multi-core
CHI	Coherent Hub Interface	AMBA 5 — 2014	High performance multi-core coherency

For ARM Cortex-M MCUs the relevant protocols are APB, AHB-Lite, and AXI.

ACE and CHI are for multi-core application processors — outside our scope today.

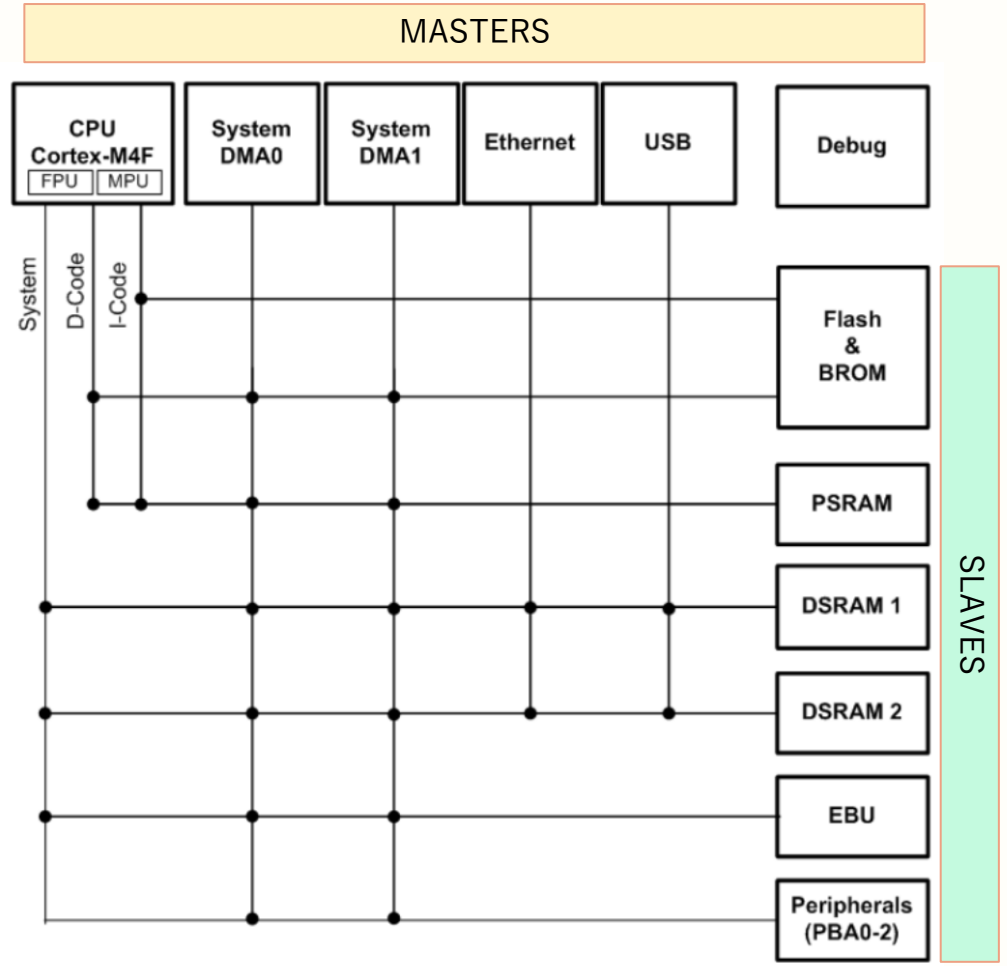
Bus Matrix

What is Bus Matrix?

ARM bus matrices, based on AMBA standards, connect multiple masters (like CPUs or DMA) to multiple slaves (such as memory or peripherals) simultaneously.

Key types include

- the Multi-layer AHB Bus Matrix for parallel access,
- the simpler AHB-Lite Bus Matrix for single-master systems,
- the high-performance AXI Interconnect for advanced designs, and
- the Multi-AHB Matrix commonly used in Cortex-M microcontrollers for handling multiple masters.

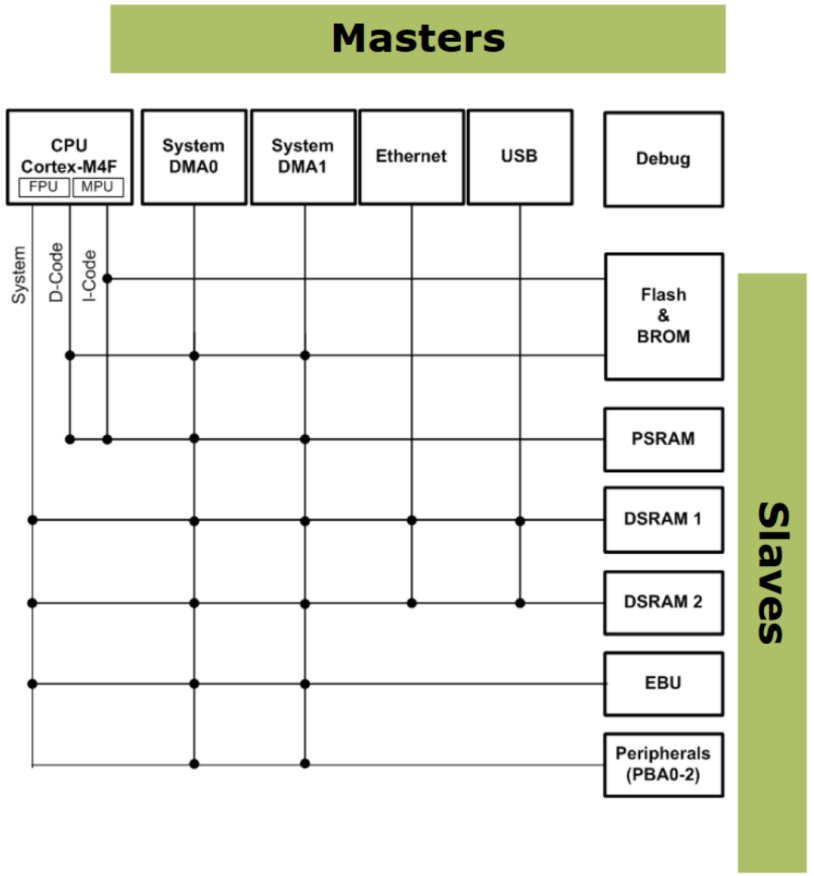


Bus Matrix: Ex – Infineon XMC4000 (AHB Lite (System & Code), APB Bus, External Bus Unit, Multi Layer Bus Matrix

Bus Matrix Characteristics

Bus Matrix Characteristics:

- **Arbitration:** The matrix manages which master gains access to a slave, often using a "round-robin" or fixed-priority arbitration scheme.
- **Latency:** While providing parallel access, the matrix can introduce a latency of at least one cycle at each new access for some slaves.
- **Configurability:** Bus matrices are generated based on the number of masters, slaves, and data width needed, typically using Perl scripts to generate Verilog RTL.



Access Priorities per Slave

Examples of simultaneous accesses:
1. Ethernet-DMA to DSRAM2 while
2. CPU to PSRAM while
3. DMA0 to DSRAM1

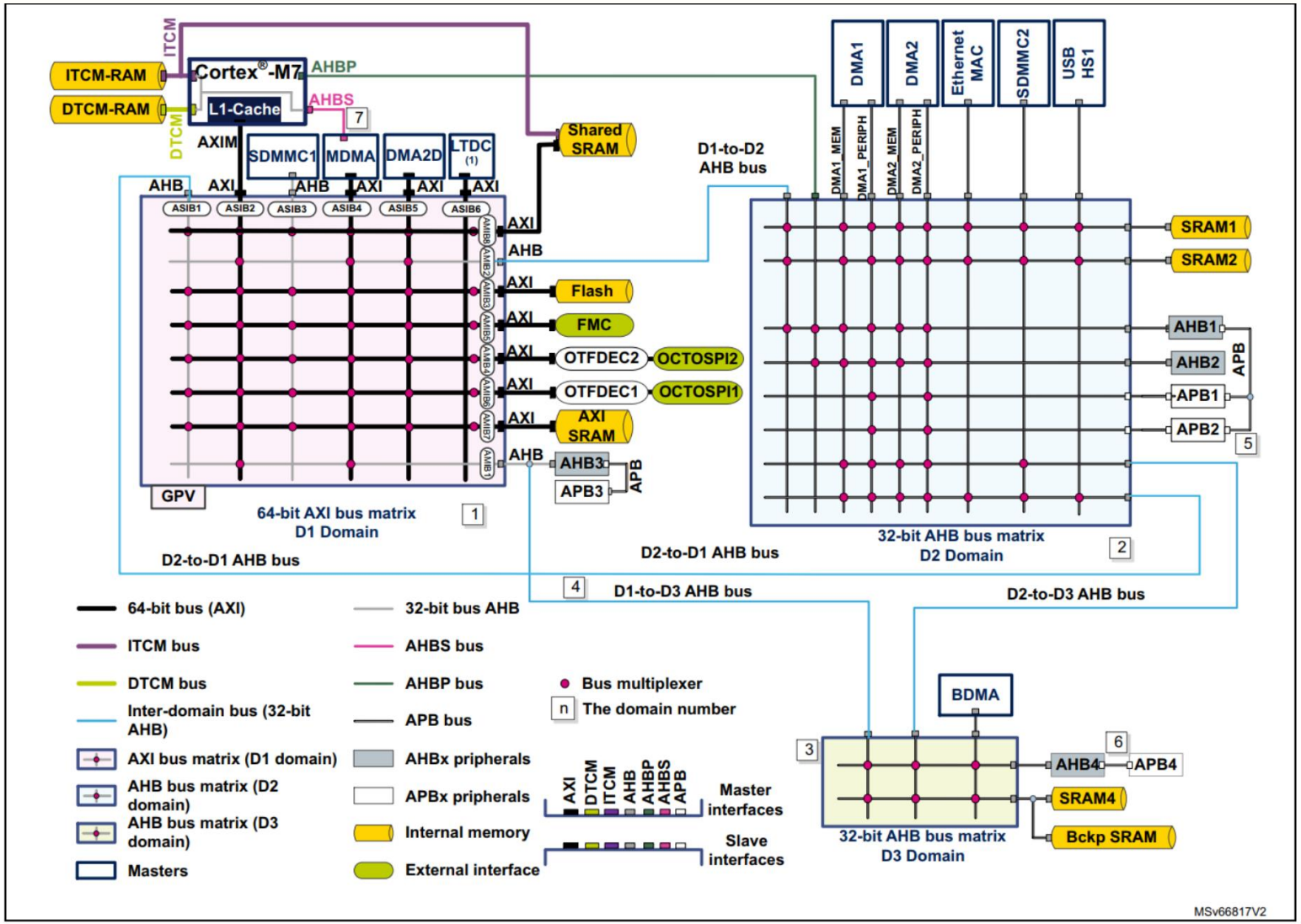
	CPU	GPD MA0	GPD MA1	ETH	USB
FLASH	1	2	3	-	-
PSRAM	1	2	3	-	-
DSRAM 1	1	2	3	4	5
DSRAM 2	1	4	5	2	3
EBU	1	2	3	-	-
PBA0-2	1	2	3	-	-

Bus Matrix - STM32H743 example

STM32H7 series is the first series of Arm® Cortex®-M7-based 32-bit microcontrollers able to run at up to 550 MHz, reaching new performance records of 1177 DMIPS and 2777 CoreMark®.

- STM32H743 has 3 interconnect fabrics in total:
  - 1 × 64-bit AXI Matrix — in the D1 domain
  - 2 × 32-bit AHB Bus Matrices — in the D2 and D3 domains
  - D1 Domain — CPU Domain:
  - D2 Domain — Peripheral Domain:
  - D3 Domain — Low-power Domain:

Why the Three Domain Architecture Matters Beyond Performance? The domain split is not just about bandwidth — it is also about power management:D3 domain can remain active while D1 and D2 are completely powered off. A low-power sensor acquisition running on BDMA in D3 — reading an ADC and storing to SRAM4 — can operate without waking the CPU or any D1/D2 peripheral. The Cortex-M7 stays in deep sleep. This is critical for battery-powered industrial IoT applications using the STM32H7.



https://www.st.com/resource/en/application_note/an4891-stm32h72x-stm32h73x-and-singlecore-stm32h74x75x-system-architecture-and-performance-stmicroelectronics.pdf

How High Bandwidth Peripherals Stress the Bus Matrix

Adding a

TFT LCD controller,

Gigabit Ethernet,

USB High Speed, or

external SDRAM

to your MCU design does not just add functionality

— it adds aggressive DMA masters that compete for the same bus matrix bandwidth your CPU depends on.

Understanding this competition is essential for system-level MCU selection.

The Bandwidth Demand of High-Speed Peripherals

Peripheral	Data Rate	DMA Burst Size	Bus Matrix Load	Typical Use Case
TFT LCD — 320×240 @ 60 Hz	320×240×2×60 = ~9.2 MB/s	32–64 bytes	Continuous — never stops	Instrument display
TFT LCD — 800×480 @ 60 Hz	800×480×2×60 = ~46 MB/s	32–64 bytes	Continuous — never stops	HMI panel
Ethernet 100 Mbps	~12.5 MB/s peak	64–1518 bytes	Bursty — packet arrival	Industrial IoT gateway
Ethernet 1 Gbps	~125 MB/s peak	64–1518 bytes	Bursty but intense	High speed data logging
USB HS — 480 Mbps	~60 MB/s peak	512 bytes per packet	Bursty — transfer dependent	USB data acquisition
USB FS — 12 Mbps	~1.5 MB/s peak	64 bytes per packet	Low — manageable	HID, CDC comm
External SDRAM — read/write	Up to 200 MB/s	16–256 bytes burst	Continuous when active	Frame buffer storage
ADC — 12-bit @ 1 Msps	2 MB/s	8–16 bytes	Moderate — DMA driven	Motor phase current sensing
SD Card — SDIO	Up to 25 MB/s	512 bytes per block	Bursty — logging bursts	Data logging to SD

Bus Matrix Stress

Four Scenarios From Comfortable to Collapse

Scenario	System Configuration	Bus Matrix Load	Result
1 — Comfortable	STM32F407 + Motor FOC + UART @ 115200 baud	UART DMA = 14.4 KB/s — negligible	CPU sees zero additional latency. System works perfectly.
2 — Mild Stress	Add 320×240 TFT LCD @ 60 Hz	LCD DMA = 9.2 MB/s continuous — persistent SRAM competitor	Motor control jitter increases 1–3 µs per loop. Still manageable.
3 — Serious Stress	Add 800×480 LCD + 100 Mbps Ethernet	LCD DMA = 46 MB/s + Ethernet DMA = 12.5 MB/s — both targeting SRAM	CPU occasionally starved. Motor control misses 50 µs deadline intermittently. Display tears during Ethernet bursts.
4 — Collapse	Add USB HS data acquisition to Scenario 3	USB DMA adds 60 MB/s peak — three aggressive DMA masters plus CPU all on one SRAM	Display corrupts. Ethernet drops packets. USB errors. Motor control jitter becomes catastrophic.

The system did not fail because the CPU was too slow. It failed because the bus matrix could not serve four masters simultaneously targeting the same SRAM slave.

The Solution — Bus Matrix Aware System Design

Principle 1 — Separate masters onto separate matrices:

- On STM32H743 — place the CPU's critical code and data on ITCM and DTCM accessed via the AXI matrix in D1 domain.
- Place DMA1 and DMA2 operations — Ethernet, USB — on the AHB matrix in D2 domain. Place BDMA operations — low-speed sensor reading — on D3 domain. Three domains — three matrices — zero cross-domain contention on the CPU's critical path.

Principle 2 — Separate frame buffers onto separate SRAM banks:

- Place the LCD frame buffer in a dedicated SRAM bank that no other master uses. On STM32H743 — SRAM4 in D3 domain is an excellent frame buffer location — the MDMA can transfer from SRAM4 to LCD without touching D1 or D2 SRAM at all.
- CPU never competes with LCD DMA.

Principle 3 — Use external SDRAM for frame buffers where possible:

- Move the LCD frame buffer to external SDRAM via the FMC controller.
- The FMC is its own slave on the AXI matrix. LCD DMA accesses SDRAM — CPU accesses internal SRAM — different slaves — zero contention. This is the correct architecture for 800×480 and larger displays.

Principle 4 — Place critical code in TCM:

- Motor control FOC kernel in ITCM — accessed via dedicated ITCM port bypassing the AXI matrix entirely. Even when the AXI matrix is saturated with Ethernet and USB DMA — the CPU's instruction fetch is unaffected because it uses a completely separate port. This is the ultimate isolation.

Summary – Bus Protocols and Bus Matrix

Role in MCU :

- Bus Protocols (AHB, AXI, APB): Standardized on-chip communication.
- Bus Matrix: Multi-layer interconnect allowing concurrent master-slave accesses.

Impact on Overall MCU Computing Performance

- Reduces contention and enables parallel data movement between multiple masters (CPU, DMA) and slaves (memory, peripherals).
- Significantly improves system-level bandwidth, scalability, and overall throughput in complex MCUs.

Final Summary

Quick Takeaways

- **These blocks work together synergistically:** Modified Harvard + TCM/Cache + Pipelining deliver high core performance, while DMA + Bus Matrix + NVIC ensure the rest of the system doesn't become a bottleneck.
- Modern high-end MCUs (e.g., Cortex-M7) combine **all** of these features to achieve both high throughput and hard real-time determinism.

Final: The Iceberg Reality of MCU Architecture

CPU Core — Cortex-M4

Clock Speed — 168 MHz

Flash Size — 1 MB

SRAM Size — 192 KB

CoreMark Score — 566

Flash wait states at operating frequency

Prefetch buffer presence and size

Instruction cache size and hit rate

TCM presence and direct CPU connection

Number of AHB bus matrices

SRAM banking across separate slave ports

DMA controller count and stream mapping

AHB-Lite vs AXI bus width and channel separation

APB clock division ratio

Bus arbitration scheme — fixed priority vs round robin

D-Cache coherency requirements with DMA

Domain architecture — D1 D2 D3 isolation

Easily visible in marketing and Datasheets

What determines real throughput..

Seven Key Takeaways from this session

#	Takeaway	One Line Proof
1	Modified Harvard gives performance without sacrificing flexibility	Separate ICode and DCode buses — unified 4 GB address space
2	Flash wait states are the single biggest performance tax	STM32F4 at 168 MHz — CPU idle 83% of fetch cycles without ART
3	Vendors mitigate Flash latency differently	ST uses 1 KB ART cache — Infineon XMC4800 uses 8 KB cache plus BTC
4	TCM is the only deterministic true zero-wait-state solution	No cache misses — ever — at any frequency — on any code pattern
5	Bus matrix topology determines parallel throughput	One matrix — masters take turns. Three matrices — masters work in parallel.
6	DMA recovers CPU capacity consumed by data movement	ADC plus LCD plus UART without DMA consumes 38% CPU before algorithm runs
7	High bandwidth peripherals must be bus-matrix-budgeted before MCU selection	800×480 LCD plus Ethernet plus USB HS on one matrix causes system collapse

The Architecture Evaluation Checklist

When evaluating any MCU — ask these questions:

Question 1 — Is there a prefetch buffer or instruction cache or TCM — and how large?

Question 2 — Is D-Cache present — and do I need a cache coherency strategy?

Question 3 — How many bus matrices — and which masters are on each?

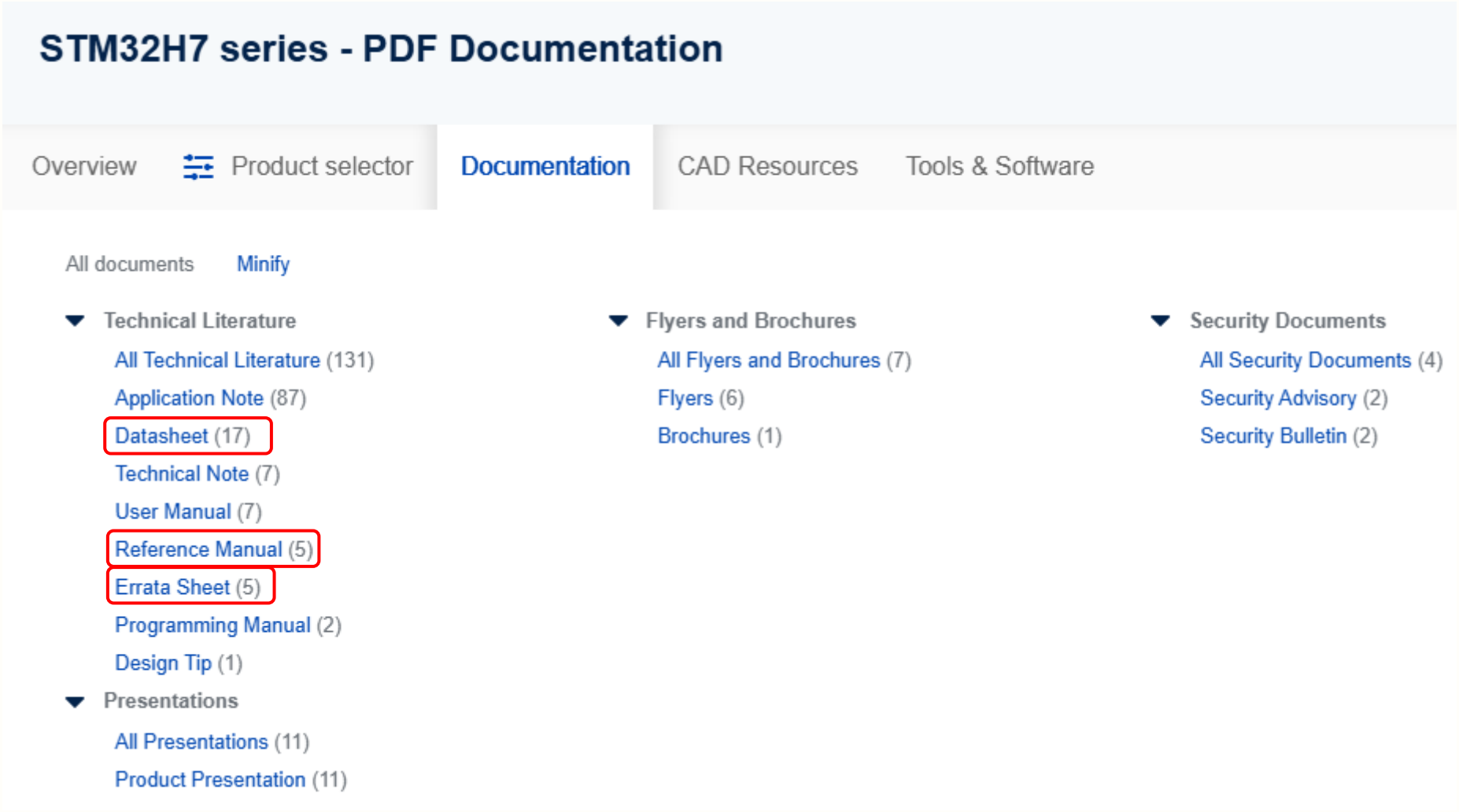
Question 4 — How is SRAM banked across slave ports?

Question 5 — How many DMA controllers and streams available?

Question 6 — What is the APB clock ratio for my peripheral speed requirements?

Question 7 — Does it have DSP, FPU to meet my signal processing and fast control needs?

Key Documents to look for during selection of MCU



https://www.st.com/en/microcontrollers-microprocessors/stm32h7-series/documentation.html

What is an Errata Sheet for an MCU?

An **Errata Sheet** (also called **Device Errata**, **Silicon Errata**, or **Limitations Document**) is an official document released by the MCU manufacturer (e.g., STMicroelectronics, NXP, Texas Instruments, Microchip) that lists **known bugs, limitations, and deviations** from the expected behavior described in the datasheet and reference manual.

It details issues found in specific **silicon revisions** (die/stepping versions) of the MCU after the device was released or during production.
For each issue, it typically includes:

- Description of the problem
- Conditions under which it occurs
- Affected silicon revisions or part numbers
- **Workarounds** (hardware or software solutions) whenever available
- Status (open, fixed in newer revision, etc.)

Errata sheets are updated over time as new issues are discovered or fixed in later silicon revisions. So, look for the revision corresponding to the silicon rev.

Why is it Necessary to Read the Errata Sheet FIRST?

Reading the errata **first** (or at least early) is a critical best practice in embedded development for these reasons:

- **Avoids nasty surprises later** — The datasheet describes how the MCU *should* work ideally. The errata tells you how it *actually* behaves in reality on the physical silicon you will use. Many seemingly straightforward features can have hidden limitations.
- **Influences MCU selection** — Some critical bugs (e.g., in timers, ADC, USB, Ethernet, or DMA) can make a particular MCU revision unsuitable for your application. Discovering this after PCB design or prototype is expensive.
- **Helps you design robust workarounds from the beginning** — Knowing the limitations upfront allows you to implement safe code or hardware solutions right from the start, instead of debugging mysterious issues months later.
- **Ensures realistic expectations** — Datasheets are marketing + ideal specs; errata provides the practical truth about the silicon you are buying.

What's the next session about?

In the next session we shall dive deep into another interesting topic – Instruction Set Architecture (ISA)