M-profile Cache Management

Course 4 Advanced Armv8-M Features

Learning objectives

- Describe fundamental cache principles, structure and terminology
- Compare cache configuration options in Cortex-M implementations
- Discuss architectural cache policies and behaviour
- Discuss when and when not to mark memory regions as cacheable
- Enable/disable/maintain caches using CMSIS
- Introduce Error Correcting Code (ECC) for cache reliability and fault detection/correction

Agenda

Caches Fundamentals

Example Cortex-M Cache Subsystems

Cache Programming Model

System Considerations

Error Correcting Code (ECC) for Caches

What is a cache?

Small fast memory, local to the processor

- Not RAM or TCM, and not directly addressed

Automatically holds copies of recently accessed memory locations

- Which memory locations are cached is controlled via MMU or MPU

Relies on memory re-use to improve performance

Improves performance for slow and narrow memory

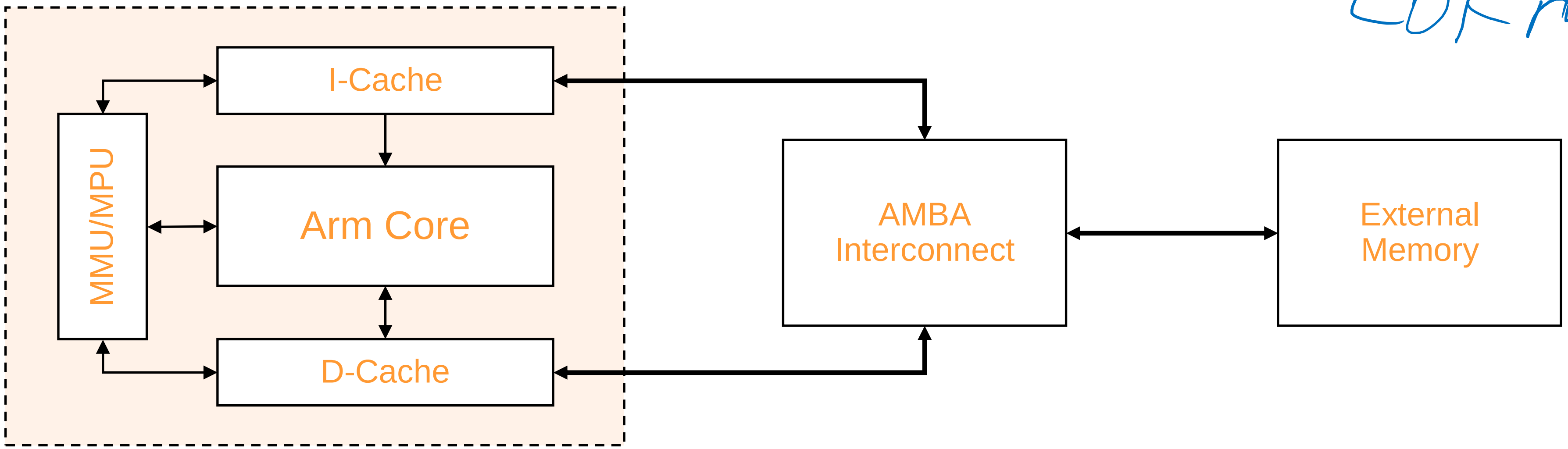
Reduces bus bandwidth requirements and so power consumption

LDR r0, [r1]

Temporal locality

Spatial locality

LDR r2, [r1, #4]



How is a cache accessed by the core?

When the core requests an access to cacheable memory, the cache will be searched first

- A Cache Lookup is performed on the requested memory address

The requested address is split into a Tag, Index and Offset field

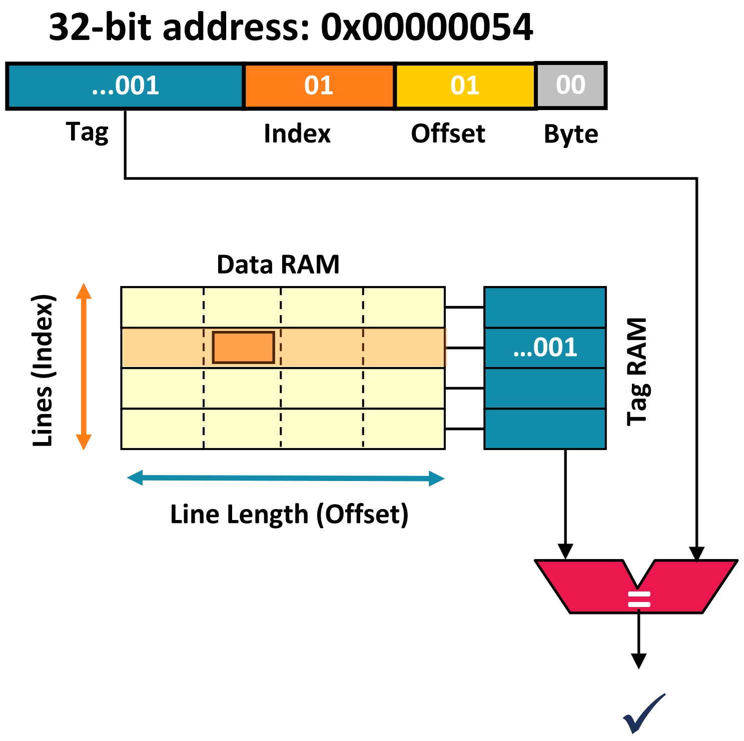
- The Index is used to locate the Cache Line of interest
- The Tag is compared against the saved Tag for that particular line

Tag match indicates a Cache Hit

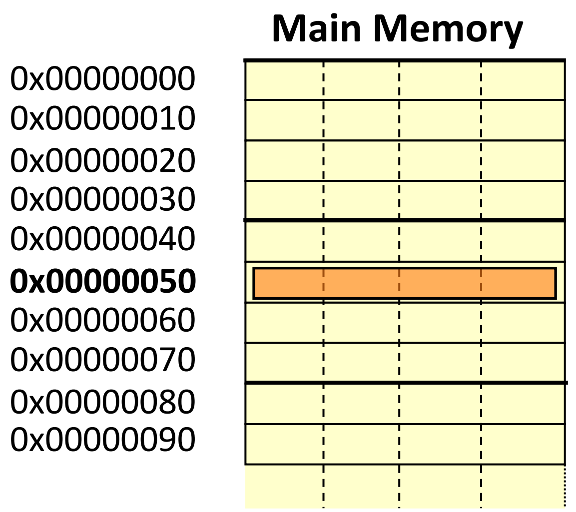
- The Offset field will specify a particular word in the cache line

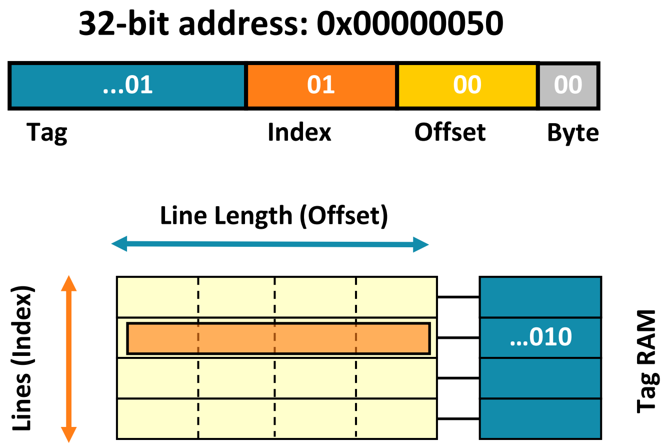
Tag mismatch indicates a Cache Miss

- Will result in an access to external memory
- Access will be delayed
- A Cache Miss typically results in a Cache Linefill



How is a cache populated?





Data is copied into the cache one line at a time (Cache Linefill)

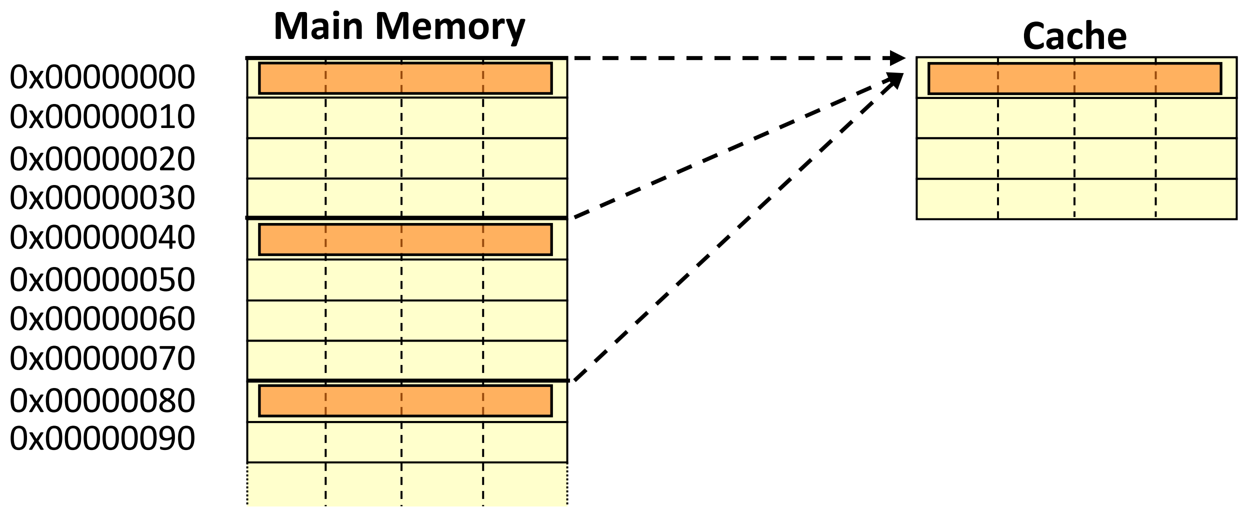
- A cache line is always aligned to the cache line size (Offset is zero)

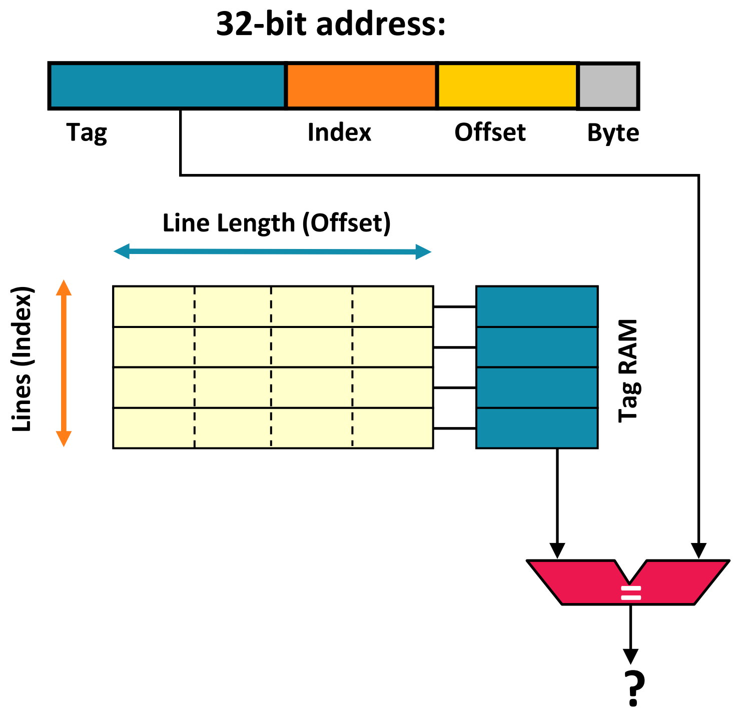
A Cache Linefill is triggered by a Cache Miss

- Data previously requested is now loaded from external memory into the cache line selected by the Index
- The associated cache Tag is updated

Allocated bits for each field depend on the structure and size of the cache

Direct mapped cache





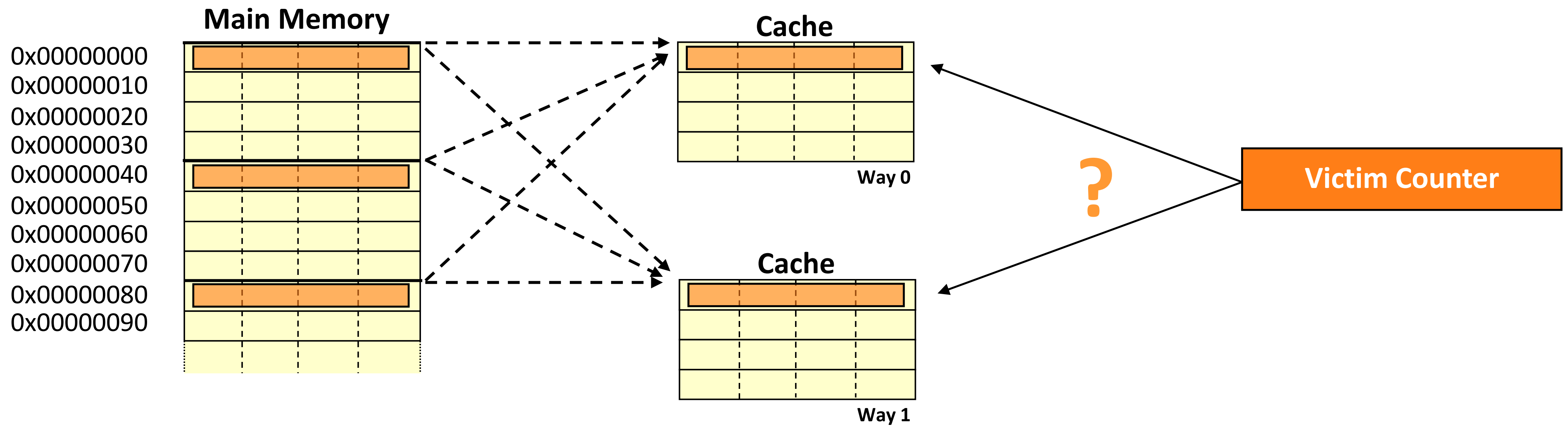
Memory space maps to a single block of cache (Way)

- Multiple memory locations will contend for the same cache line (same Index)
- The saved Tag will identify which memory location the line contains

Data can easily get replaced before re-use (Eviction)

- For example, a processing loop that is twice the size of the cache?

Set associative cache



Multiple cache Ways work in parallel

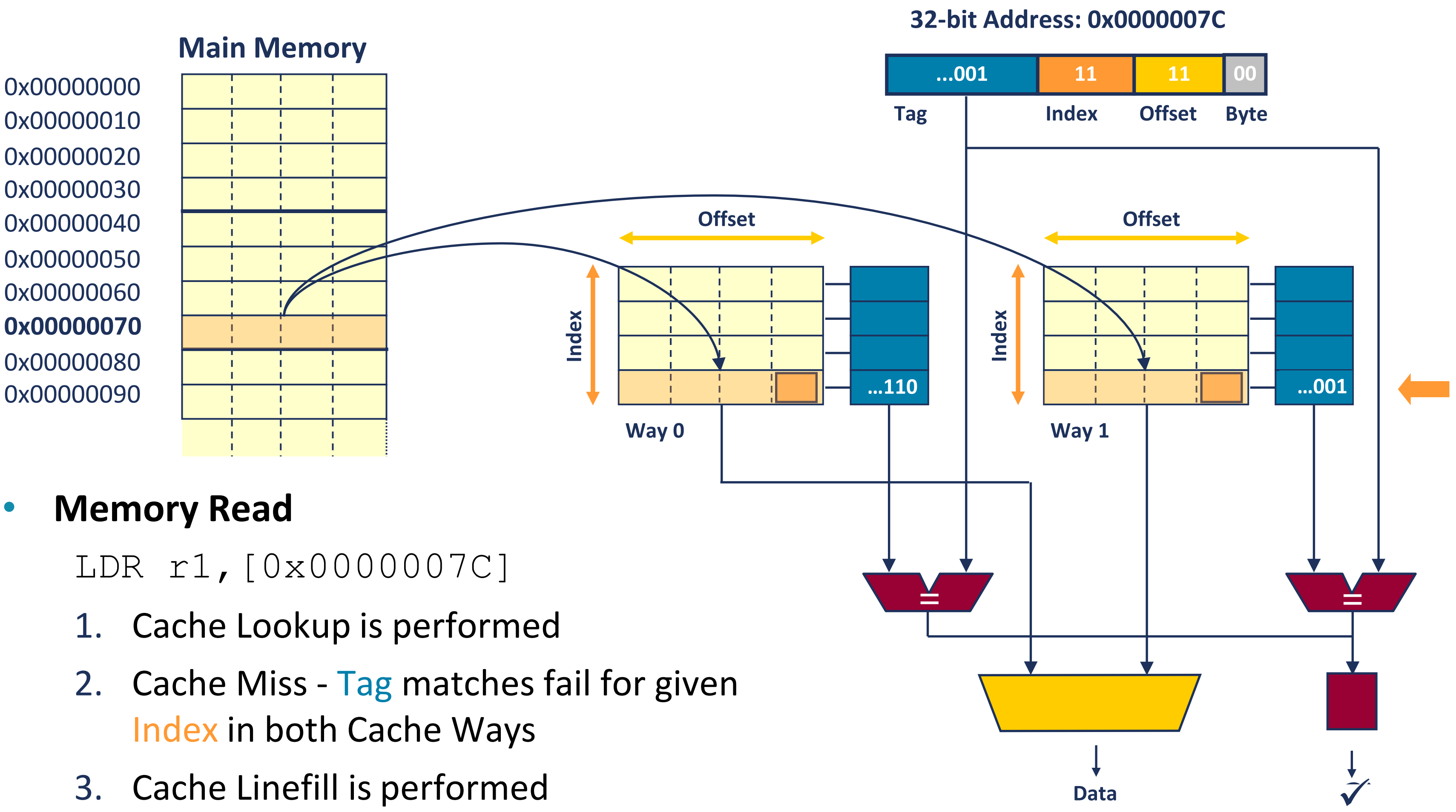
There are now multiple possible cache locations for any given address

- One in each cache Way

Victim counter decides which cache Way will be used for the Linefill

- The previous data will get evicted from the selected cache line
- Evicted data may have to update external memory (*Dirty* data)

Example memory access



- **Memory Read**

  `LDR r1,[0x0000007C]`

  1. Cache Lookup is performed
  2. Cache Miss - Tag matches fail for given Index in both Cache Ways
  3. Cache Linefill is performed
  4. Victim counter specifies which cache Way to use (will Evict previous data)
  5. Cache returns requested word to the core

Set associative cache - summary

Set Associative caches reduce contention problems

- Consists of Directly Mapped Caches in parallel (each referred to as a Way)
- More complex to implement and requires more comparison logic

Each memory location now has ‘n’ possible locations

- Where ‘n’ is determined by the number of Cache Ways implemented

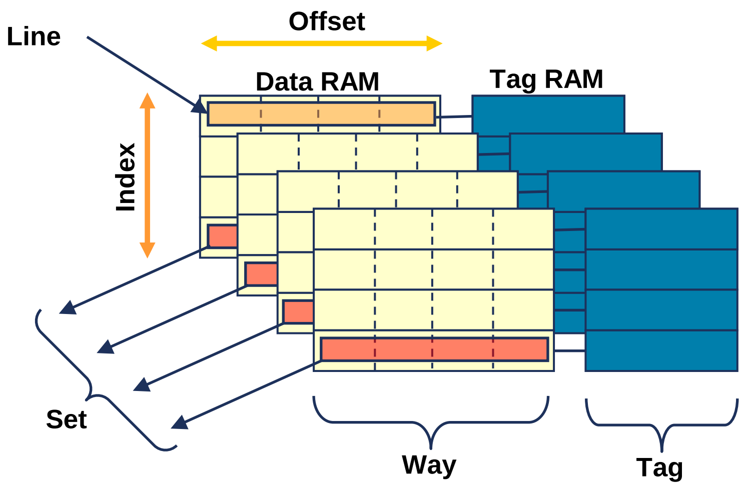
Victim counter selects which Cache Way to use for an Eviction/Linefill

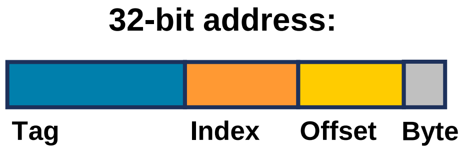
- Arm processors support one or more of the following strategies:
  - Random
  - Round robin (cyclic)

Unoccupied lines are used in preference for the Eviction scheme

- Supported by some of Arm cores

Cache terminology





Line	Smallest loadable unit of a cache - always a block of contiguous words in memory
Way	Refers to a particular cache 'page' - example above has 4 Ways
Set	The same line from each cache Way grouped together forms a Set
Tag	The portion of a memory address which is stored within the cache to identify the particular physical address located there
Index	The portion of the memory address which determines the Set in which the cache line may be stored

Agenda

Caches Fundamentals

Example Cortex-M Cache Subsystems

Cache Programming Model

System Considerations

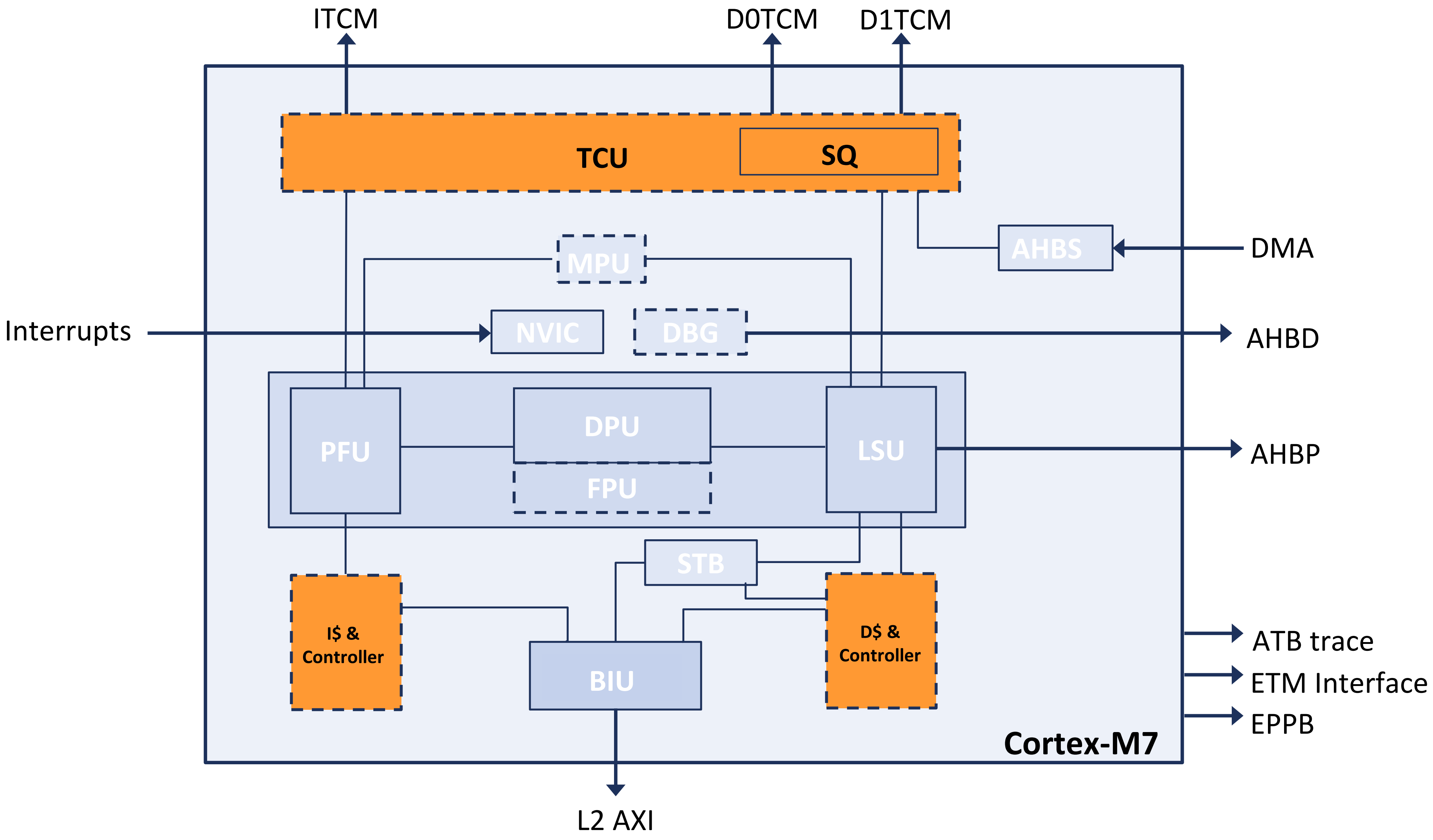
Error Correcting Code (ECC) for Caches

Cortex-M implementations with caches

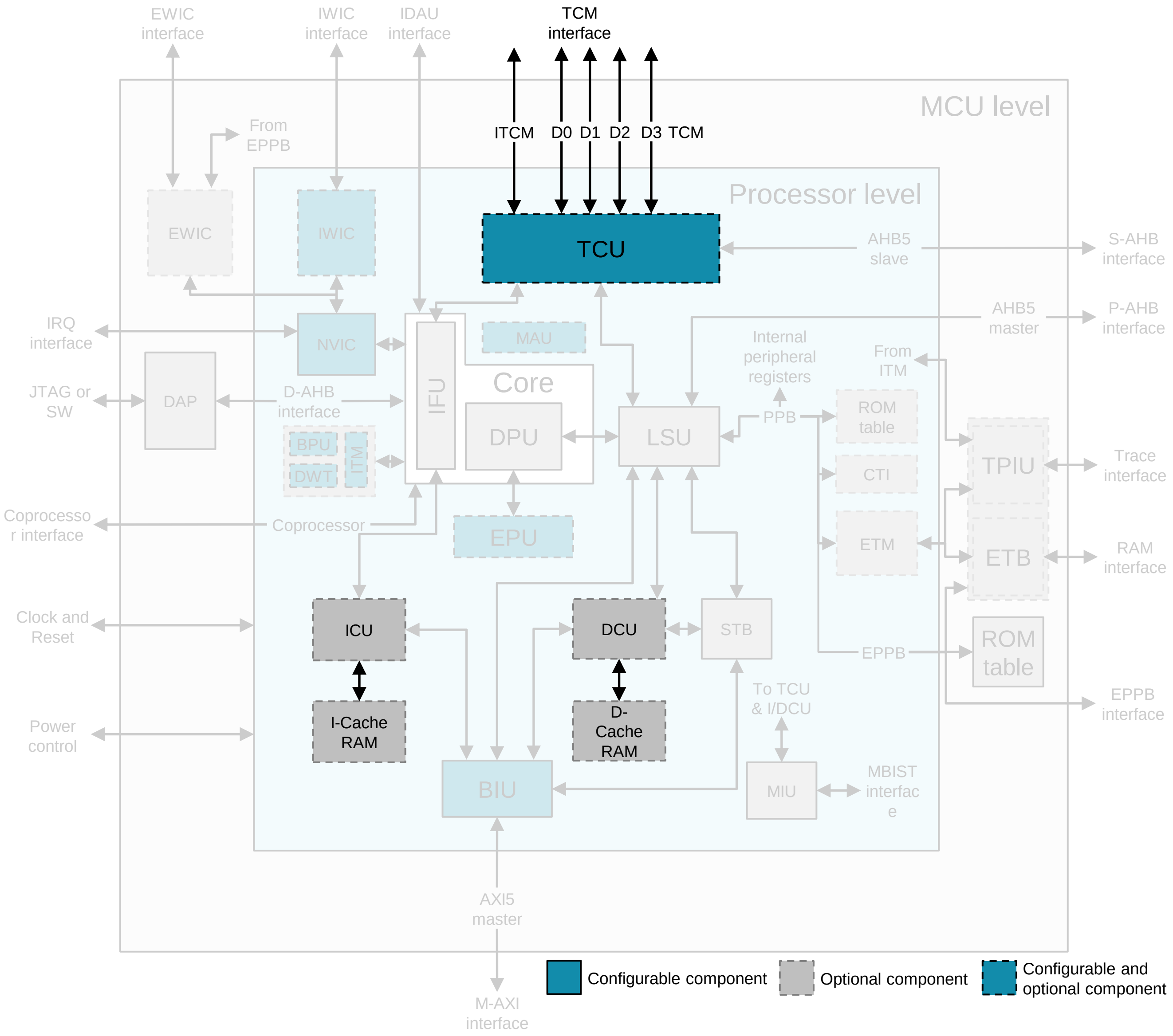
Cortex-M7 and Cortex-M55 support internal caches

All Cortex-M processor have the option to support system caches

Cortex-M7 processor block diagram



Cortex-M55 Processor



Cortex-M7 and Cortex-M55 L1 caches

Separate instructions and data caches (4KB, 8KB, 16KB, 32KB & 64KB)

- Controlled by PPB registers

I-cache

- 2-way set-associative with a pseudo-random replacement policy
  - Lower-cost than 4-way associative with almost identical performance

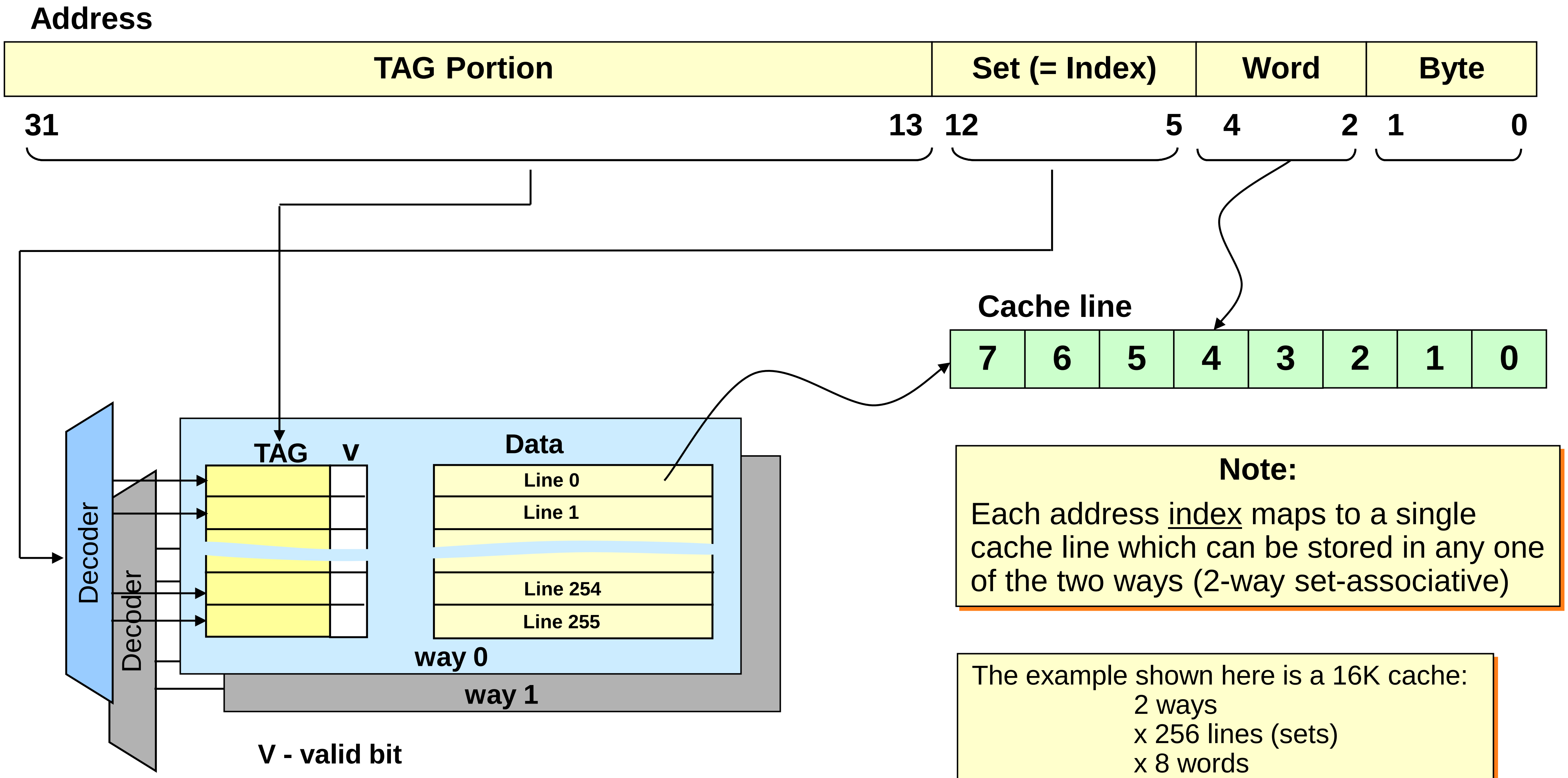
D-cache

- 4-way set-associative with a pseudo-random replacement policy
- Support for dual-issue of loads without use of dual-ported memories

Both caches support ECC

- SEC-DED on Tag and Data RAMs
- Error bank registers – hard-error support, containment, diagnostics

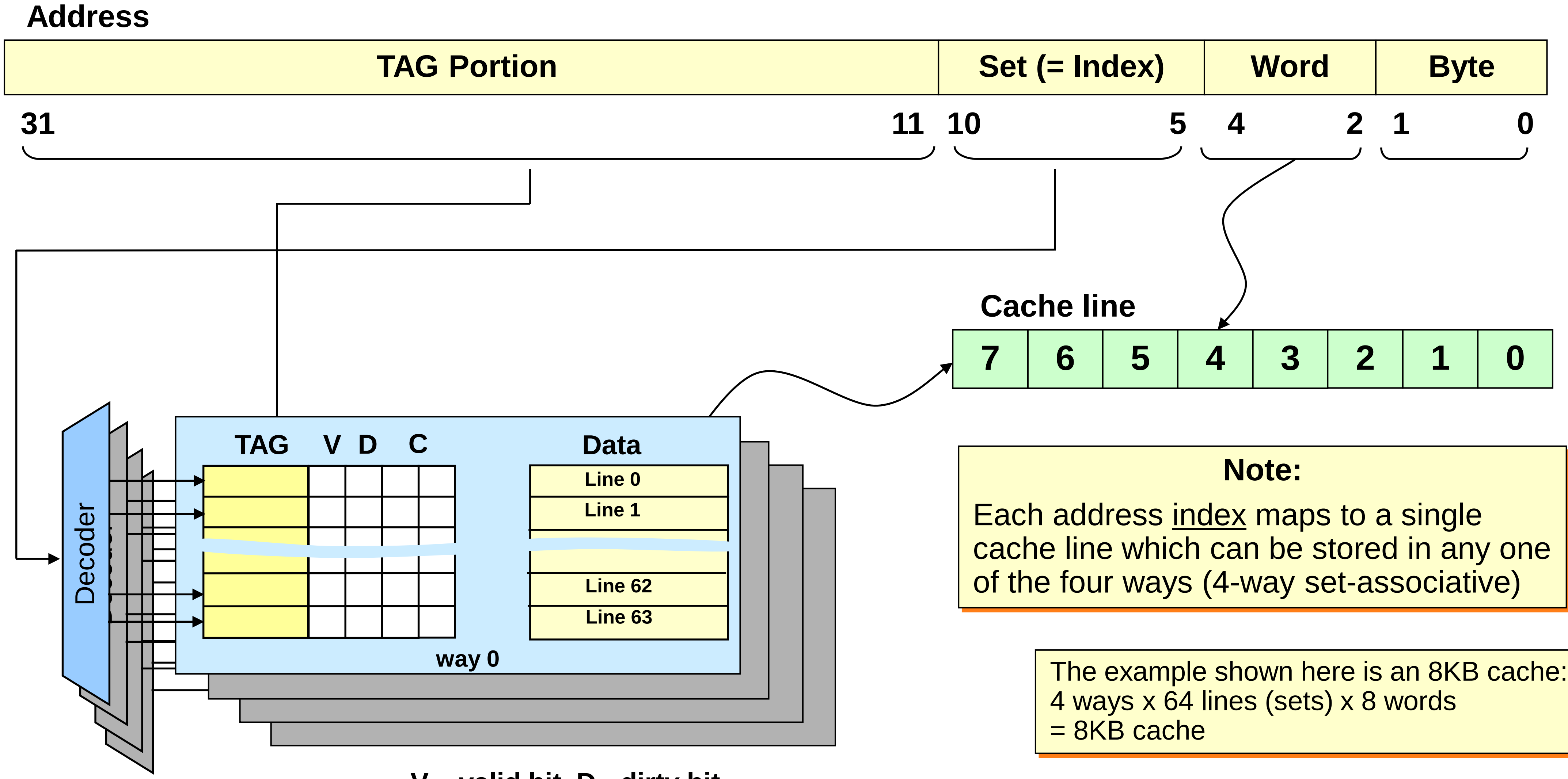
I-Cache 2-Way Set-Associative



Note:
Each address index maps to a single cache line which can be stored in any one of the two ways (2-way set-associative)

The example shown here is a 16K cache:
2 ways
x 256 lines (sets)
x 8 words
= 16KB cache

D-Cache 4-Way Set-Associative



V – valid bit, D - dirty bit,

C – 2 bits outer cacheability attributes

Note:

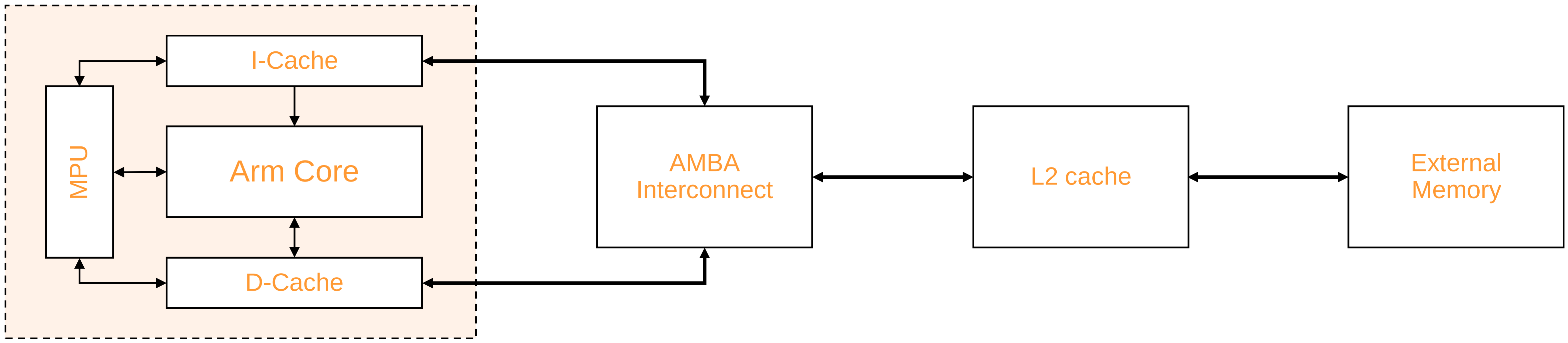
Each address index maps to a single cache line which can be stored in any one of the four ways (4-way set-associative)

The example shown here is an 8KB cache:
4 ways x 64 lines (sets) x 8 words
= 8KB cache

Level 2 caches

A Cortex-M processor may implement two levels of cache

- Internal L1 cache
  - Relatively small, providing fast access inside the L1 sub-system
- External L2 cache (system cache)
  - Relatively large, with access times slower than L1 memory accesses
  - Other Arm architectures permit internal L2 caches



Agenda

Caches Fundamentals

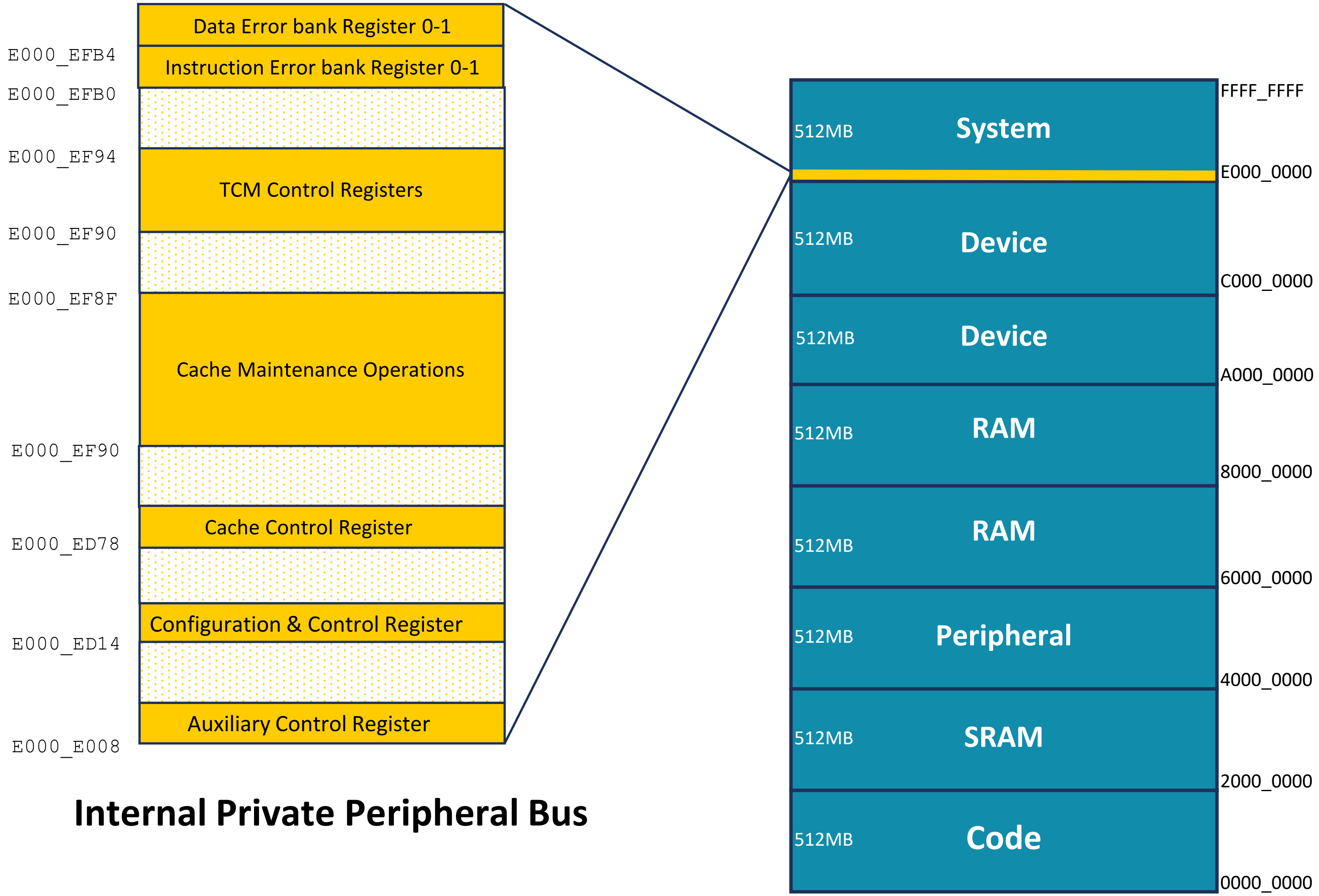
Example Cortex-M Cache Subsystems

Cache Programming Model

System Considerations

Error Correcting Code (ECC) for Caches

PPB L1 sub-system related regions



Internal Private Peripheral Bus

L1 data cache policies

Read-Allocate (RA):

- Only cache misses due to reads cause cache linefills
- Can be configured in Write-Through or Write-Back mode

Write-Allocate (WA):

- Cache misses due to reads and writes cause cache linefills
- Can be configured as Write-Back only

Write-Through(WT)

- Write updates both the cache and the external memory system

Write-Back (WB)

- Write updates the cache only (and cache line is marked as dirty)

Transient and Non-transient (not available in Armv7-M)

- Clean cache lines that are associated with Transient memory are prioritized for eviction over lines that are associated with Non-transient memory

Caching and memory attributes

Memory attributes are defined by the default memory map or MPU settings

- These attributes affect the bus protocol/cache memory system behavior significantly

Attributes:

- Memory type:
  - Normal – can be cached (depending on cacheability/shared attributes)
  - Device/Strongly-ordered – use only for Devices – not cached
- Allocate policy
  - Write allocate – best for overall performance
  - Read allocate – specific use cases, e.g., `memset()`
  - Write-through – lower performance that WB. Useful for safety-critical
  - For I-cache, all allocate policies are treated as read-allocate (no instruction writes)
- Shared
  - Only use for memory that is shared with another processor (coherency guarantees)
  - Will by default ensure data accesses are NOT cached in the D-cache
  - Has no effect on I-cache – shareable regions will be cached

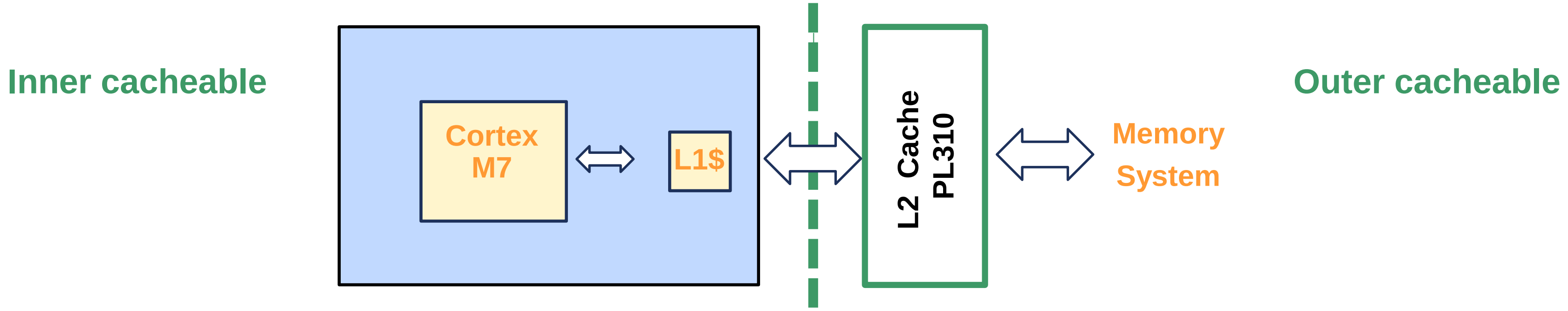
Inner and outer cache policies

Policies can be defined separately for Inner and Outer cache regions

- **Inner** cacheable: data may be stored by caches inside the Cortex-M7 processor
- **Outer** cacheable: generally means access can be stored in an external cache

Region definition depends on the MPU settings

- L2 cache policies on Cortex-M7 and Cortex-M55 are described by the outer cache region attributes



Cache coherency

There is no hardware coherency support

Two options to make coherency work:

- Use the Shared attribute for all shared regions of memory
  - This will by default prevent these regions from being cached in D-cache
  - Lower performance since all accesses need to go to L2
  - Easiest for software since caches are transparent for these regions of memory

- Software cache maintenance
  - Writes need to be made globally visible
    - Use Write-Through
    - D-cache clean or D-cache clean and invalidate for all updated locations
    - The Cortex-M7 L1 Cache Control Register also supports a SIWT field to treat Normal, Cacheable and Shared memory as Write Through
  - Other master writes need to be visible
    - Invalidate updated locations in Cortex-M processor’s D-cache

L1 memory system buffers

L1 caches utilize internal buffers to ensure that they function efficiently

For example, inside the Cortex-M7 there are three types of buffers (each 32 bytes in size)

- 1 x Instruction Linefill Buffer
- 2 x Data Linefill Buffers
  - Only present if a D-cache is implemented
  - Used for both load-initiated and store-initiated linefills
  - Stores from store buffer can merge into linefill buffer
  - Also used for non-cacheable read bursts
- 1 x Store Buffer
  - Used for evictions
  - Used for write-through, read-allocate and non-cacheable write bursts

Internal buffers are transparent to the programmer

Cache Control & Identification Registers

Address	Register Name	Function
0xE000EF9C	CACR	L1 Cache Control Register *
0xE000ED84	CSSELR	Cache Size Selection Register
0xE000ED80	CCSIDR	Cache Size ID Register
0xE000ED7C	CTR	Cache Type Register
0xE000ED78	CLIDR	Cache Level ID Register
0xE000ED14	CCR	Configuration and Control Register
0xE000E008	ACTLR	Auxiliary Control Register

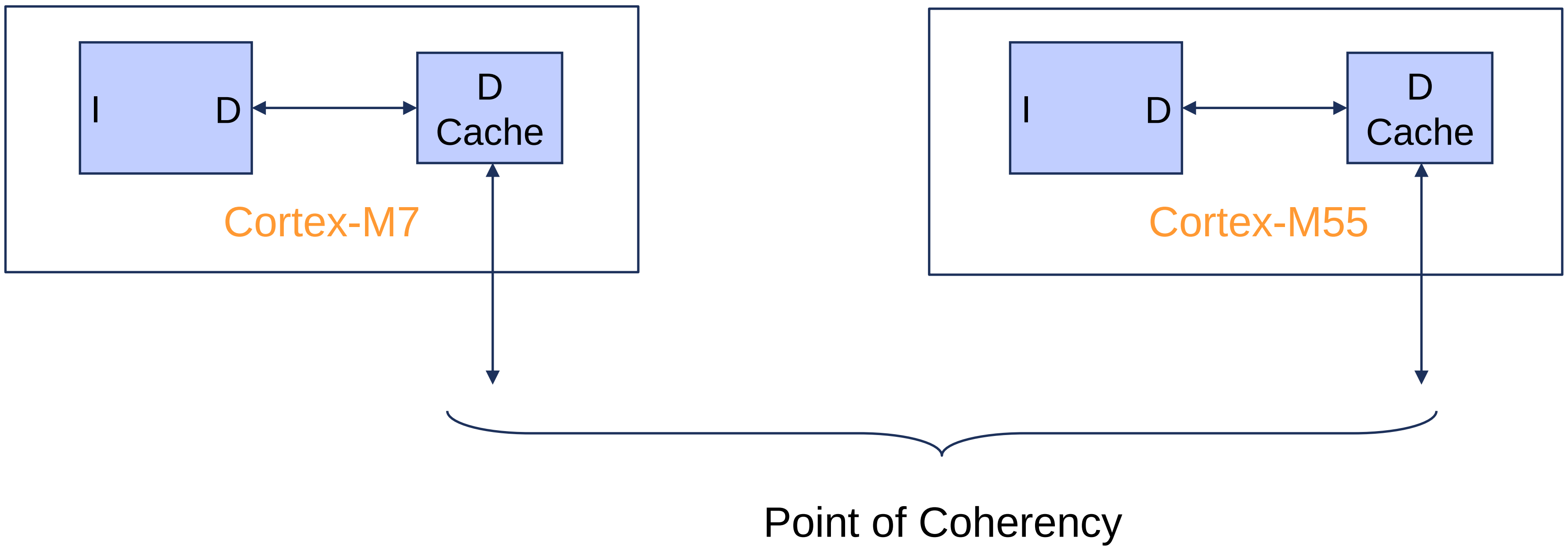
*** The CACR is a non-architectural register**

Point of Coherency (PoC)

All agents see the same copy of memory

For Cortex-M7 and Cortex-M55:

- PoC is 1 – Data accesses are coherent at L2 or beyond

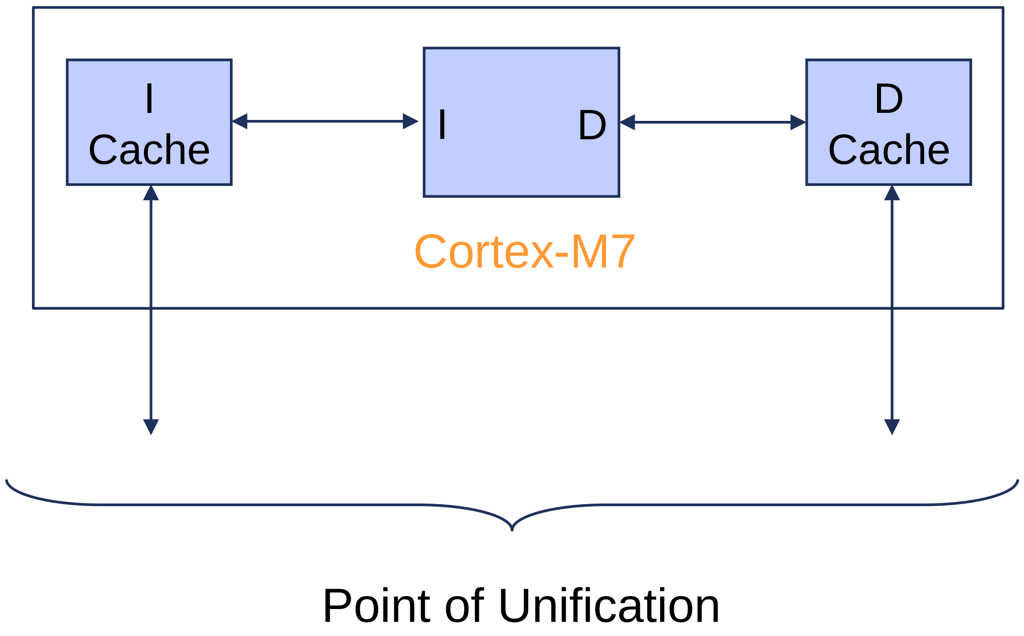


Point of Unification (PoU)

Instruction and data caches of the PE see the same copy of memory

For Cortex-M7:

- PoU is 1 – D-cache and I-cache accesses are unified at L2



Cache maintenance operations (1)

All cache maintenance operations are through the memory mapped System Control Space (SCS) region of the internal PPB memory space

Address	Register	Type	Function	Data
0xE000EF50	ICIALLU	WO	I-Cache invalidate all to PoU *	Ignored
0xE000EF58	ICIMVAU	WO	I-Cache invalidate by address to PoU *	Address
0xE000EF5C	DCIMVAC	WO	D-Cache invalidate by address to PoC	Address
0xE000EF60	DCISW	WO	D-Cache invalidate by set/way	Set/Way
0xE000EF64	DCCMVAU	WO	D-Cache clean by address to PoU	Address
0xE000EF68	DCCMVAC	WO	D-Cache clean by address to PoC	Address
0xE000EF6C	DCCSW	WO	D-Cache clean by set/way	Set/Way
0xE000EF70	DCCIMVAC	WO	D-Cache clean & invalidate by address to PoC	Address
0xE000EF74	DCCISW	WO	D-Cache clean & invalidate by set/way	Set/Way
0xE000EF78	BPIALL	RAZ/WI	Branch predictor invalidate all	Ignored

The data specifies an address or set/way, otherwise it is irrelevant/ignored

* Only applies to separate I-Caches, does not apply to unified caches

Cache maintenance operations (2)

For data cache maintenance operations by set/way (DCISW, DCCSW and DCCISW) the following bit assignments are used so that software can specify which set/way to clean/invalidate

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Way																	Set														0

- Way that the operation applies to
- For the data cache, values 0, 1, 2 and 3 are supported

- Set/index that the operation applies to
- The number of indices in a cache depends on the configured cache size

Cache maintenance operations (3)

Architecturally:

- DSB is required to ensure completion of all previous cache maintenance operations
- ISB is required to ensure that I-cache maintenance operations are visible to subsequent instruction fetches

These barriers are a requirement on Cortex-M7

- Cache maintenance operations can continue in the background without stalling the pipeline
- Therefore, software needs to correctly use barriers

Note that you can use a single DSB after a sequence of cache maintenance operations - you do not need a barrier after each

Cache maintenance operations can continue in the background without stalling the pipeline

- You must execute a DSB if you want to ensure all previous cache maintenance operations have completed

Initializing and enabling L1 caches (1)

Both the I-cache and D-cache must be completely invalidated before they are enabled on power-on reset

- Cache line valid bits are held in Tag RAM (for area/power)
- Failure to do this could cause UNPREDICTABLE behavior
- This is an architectural requirement

I-cache:

- Invalidate the entire I-cache using the ICIALLU register

D-cache

- Recommended method is to iterate through the entire D-cache and invalidate

Initializing and enabling L1 caches (2)

Note that on soft resets, it is possible to avoid invalidating the caches if the contents of the RAMs before reset were reliable

- This is useful for coming out of dormant state for example.

To ensure data coherency, the D-cache should be cleaned before it is disabled

- Required only if Write-Back caching is used
- Failure to do so could result in loss of data

The Cortex-M7 and Cortex-M55 Technical Reference Manuals contain example code

- CMSIS-Core provides L1 cache access functions

Initializing and enabling L1 caches with CMSIS (1)

CMSIS functions are available to invalidate and clean the entire data cache

```
__STATIC_INLINE void  SCB_InvalidateDCache (void)      // The function invalidates the entire data cache. After reset,
                                                          you must invalidate each cache before enabling it.


__STATIC_INLINE void  SCB_CleanDCache (void)           // The function cleans the entire data cache.


__STATIC_INLINE void  SCB_CleanInvalidateDCache (void) // The function cleans and invalidates the entire data cache.


/* The function invalidates a memory block of size dsize [bytes] starting at address address.
   The address is aligned to 32-byte boundry. */

__STATIC_INLINE void  SCB_InvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize)


/* The function cleans a memory block of size dsize [bytes] starting at address address.
   The address is aligned to 32-byte boundry. */
__STATIC_INLINE void  SCB_CleanDCache_by_Addr (uint32_t *addr, int32_t dsize)


/* The function invalidates and cleans a memory block of size dsize [bytes] starting at address address.
   The address is aligned to 32-byte boundary. */

__STATIC_INLINE void SCB_CleanInvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize)
```

Parameters

[in] `addr` address (aligned to 32-byte boundary)

[in] `dsize` size of memory block (in number of bytes)

Initializing and enabling L1 caches with CMSIS (2)

Invalidate instruction cache

```
__STATIC_INLINE void SCB_InvalidateICache (void)      // The function invalidates the entire instruction cache
                                                      // Before enabling the instruction cache, you must invalidate
                                                      // the entire instruction cache if external memory might have
                                                      // changed since the cache was disabled.
```

Enable data and instruction cache

```
__STATIC_INLINE void  SCB_EnableDCache (void)          // The function turns on the entire data cache.

__STATIC_INLINE void  SCB_EnableICache (void)          // The function turns on the instruction cache.
```

Agenda

Caches Fundamentals

Example Cortex-M Cache Subsystems

Cache Programming Model

System Considerations

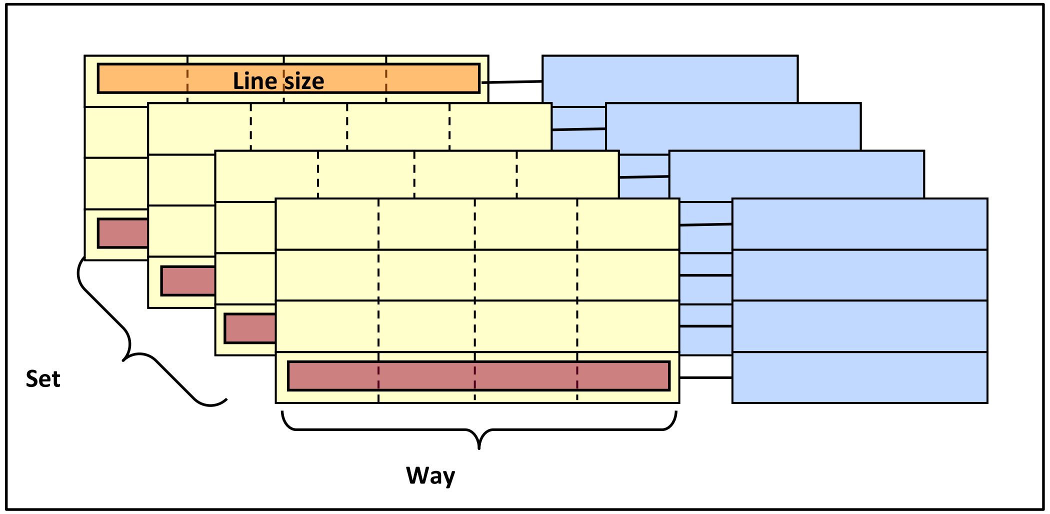
Error Correcting Code (ECC) for Caches

Cache discovery code (1)

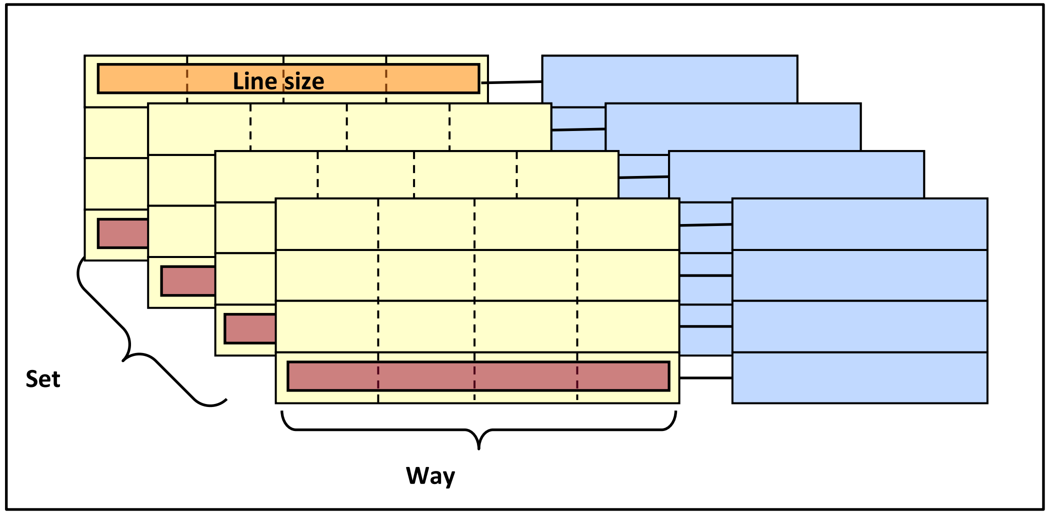
When writing code to clean/invalidate data in the caches, you need to know a few things:

- How many levels of cache are there?
- How big is a cache line?
- How many sets and ways are in the cache?

Level 1



Level 2



…

Cache discovery code (2)

The number of cache levels is listed in the Cache Level ID Register (`CLIDR`)

The cache line size is listed in the Cache Type Register (`CTR`)

Two register accesses are needed to determine the number of sets and ways:

- Write to the Cache Size Selection Register (`CSSELR`) to select which cache you want the information for
- Read of the Cache Size ID Register (`CCSIDR`)

What should I cache?

Object	Cache attribute
Global variables and structures	Yes
Application and exception stacks	Yes Consider TCM for ISR stacks
Heap	Yes
Linked Lists	Possibly In general these data structures cache poorly
DMA regions	Possibly Coherency issues must be managed, but burst read and re-use could have performance benefits
Memory mapped peripherals	No Peripheral registers updated externally
Application code	Yes
Interrupt Service Routines	Yes Consider TCM for improved realtime behavior
Overlaid regions and self modifying code	Yes Coherency issues must be managed

Non-deterministic cache behavior

Code	Timing	Worst case cache impact
`0x091C    ADD r0,r1,r2`	; 1 cycle	; I$ miss, 8 word I$ line fill
`0x0920    MOV r12,#0xA000`	; 1 cycle	; I$ miss, 8 word I$ line fill
`0x0924    LDR r3,[r12,#0]`	; 1 cycle	; D$ miss, 8 word D$ line fill ; D$ 8 word dirty data eviction
`0x092C    LDM sp!,{r0-r9}`	; 10 cycles	; 3 x D$ miss, 24 word D$ line fill ; 3 x 8 word dirty data eviction

Other considerations

- ISR routine could evict foreground task's cached data
- In multi-processor systems, the core could stall waiting for the bus to be granted

Cache optimizations

Cached processors generally provide an excellent compromise between cheap memory systems and good application performance

- But …

Real-time systems require predictable real-time performance

- Cached processors do not exhibit deterministic real-time behavior
- A cache miss cannot easily be predicted and has a large time penalty
- Allowances must be made in system design to deal with this
  - One solution is to use on-chip SRAM (TCM) for critical real-time routines
  - Alternatively, and as a second best, a cache lockdown strategy can be used

Cached processors are designed to operate with caches enabled

Significant performance penalties can occur if all accesses are to external memory

Cortex-M7 and Cortex-M55 processors supports the `PLD` instruction for preloading data into the cache

- Reduces latency, e.g., can help optimize the performance of functions like `memcpy()`

Agenda

Caches Fundamentals

Example Cortex-M Cache Subsystems

Cache Programming Model

System Considerations

Error Correcting Code (ECC) for Caches

Cache ECC – overview

All cache ECC considered and handled internally

ECC scheme is fixed:

- SEC-DED ECC32 for D-cache and SEC-DED ECC64 for I-cache
- Protection of address decoders in Tag RAMs

On an ECC error (either correctable or fatal):

- Affected cache line is evicted
  - Dirty D-cache lines will be cleaned, other lines are just invalidated
  - ECC errors on data during evictions will be corrected inline
- Affected access is then retried
  - Line may be re-loaded from L2 and then written into L1 (i.e 'repaired')
- Error is logged (dedicated, limited storage) for system analysis
- Recorded error locations are removed from cache line allocation pool

Cache ECC – considerations

Fatal errors on dirty cache lines can result in loss of data

To guarantee against this, must use WT

- Performance implications: WT < WB

All correctable errors are transparent to software

- Assuming L2 has correct data

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