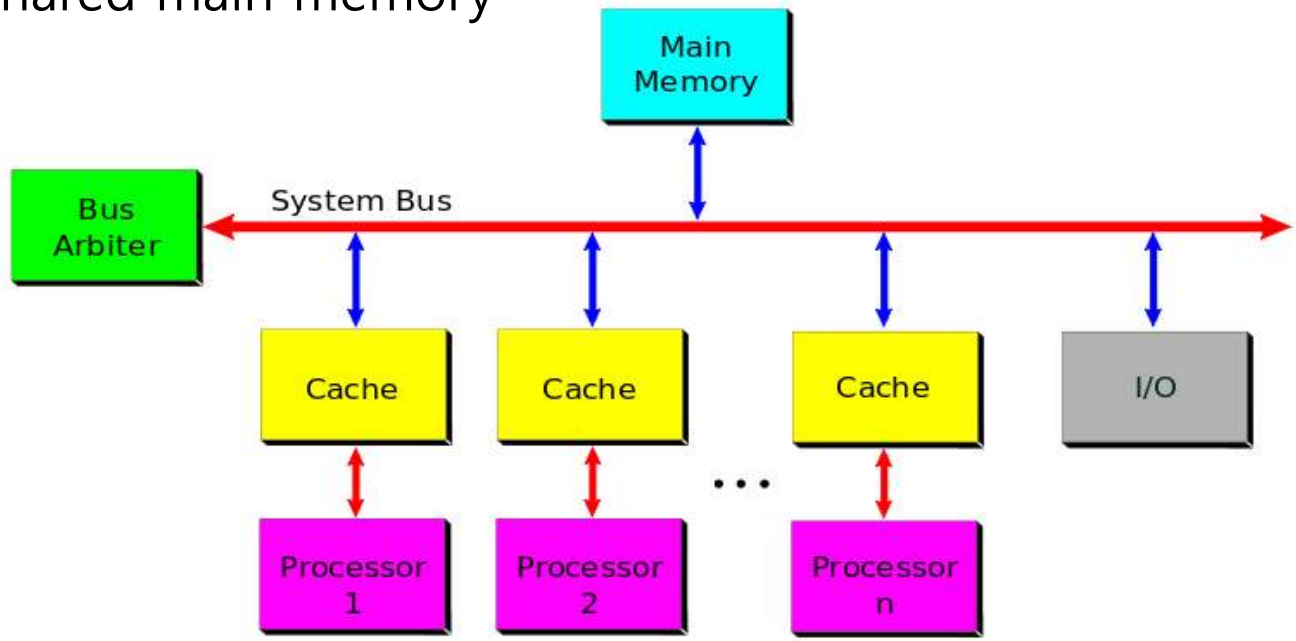
ARM 구조

- ARM v7 중심으로 -

민 홍

Memory-consistency models

- SMP (Symmetric Multiprocessor) System
  - a multiprocessor computer hardware and software architecture where two or more identical processors are connected to a single shared main memory



<출처: Wikipedia>

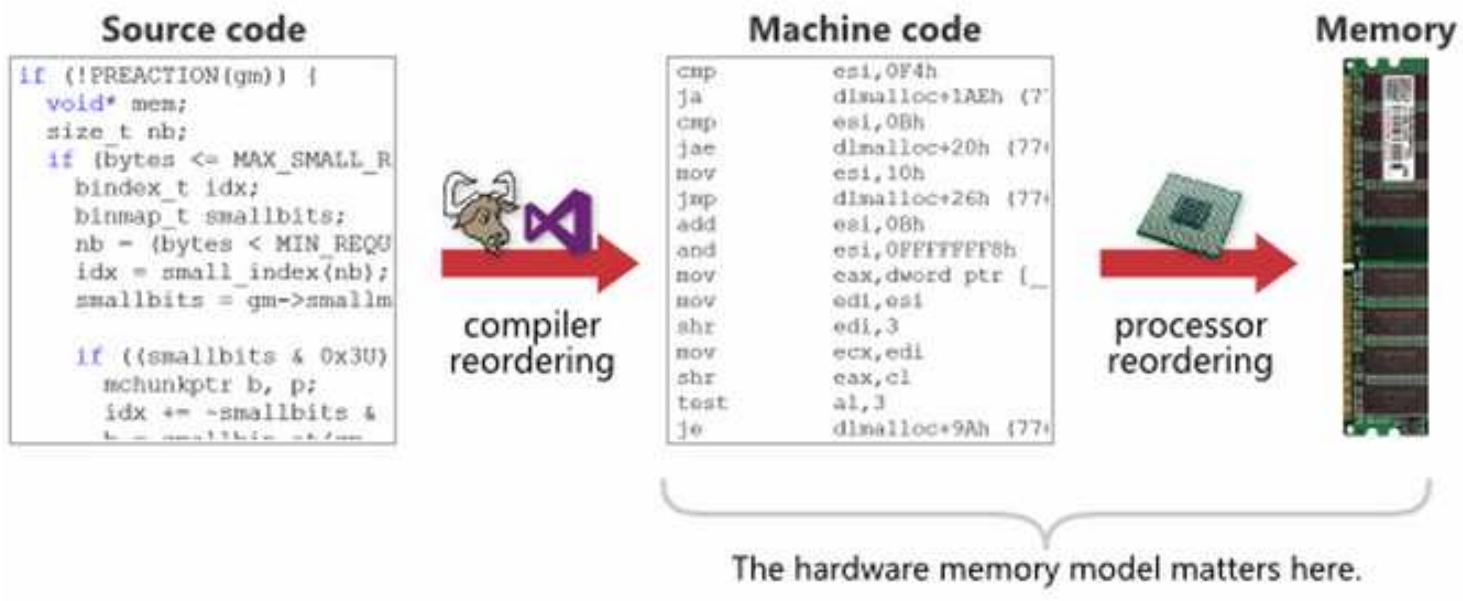
Memory-consistency models

- Memory model
  - What types of memory reordering to expect at runtime relative to a given source code listing

<출처: preshing on programming>

Memory-consistency models

- Level of reordering



<출처: preshing on programming>

Memory-consistency models

- Weak Memory Models
  - Any load or store operation can effectively be reordered with any other load or store operation
  - The reordering may be due to either compiler reordering of instructions, or memory reordering on the processor
- Weak With Data Dependency Ordering
  - If you write A->B, it always guarantees to load a value of B after loading a value of A.
- Strong Memory Models
  - When one CPU core performs a sequence of writes, every other CPU core sees those values change in the same order that they were written.
- Sequential Consistency
  - There is no memory reordering

Exception

Exception?

- Any condition that needs to halt normal execution and instead run software associated with each exception type, known as an exception handler
- The unprivileged User mode can switch to another mode only by generating an exception

Exception mode

Vector address	Exception	Mode	Event	CPSR	Return instruction
0x0	Reset	Supervisor	Reset input asserted	F = 1 I = 1	Not applicable

0x4	Undefined instruction	Undefined	Executing undefined instruction	I = 1	MOVS PC, LR
					(if emulating the instruction)
					SUBS PC, LR, #4
					(if re-executing after for example enabling VFP)

0x8	Supervisor call	Supervisor	SVC instruction	I = 1	MOVS PC,LR
0xC	Prefetch Abort	Abort	Instruction fetch from invalid address	I = 1	SUBS PC, LR, #4
0x10	Data Abort	Abort	Data Read/Write to invalid address	I = 1	SUBS PC, LR, #8 (if retry of the aborting instruction is wanted)
0x14	Yes[a]	HYP	Hypervisor entry	-	ERET
0x18	Interrupt	IRQ	IRQ input asserted	I = 1	SUBS PC, LR, #4
0x1C	Fast Interrupt	FIQ	FIQ input asserted	F = 1 I = 1	SUBS PC, LR, #4

ARM Exceptions Types

- Reset
  - Occurs when the processor reset pin is asserted
    - For signaling Power-up
    - For resetting as if the processor has just powered up
  - Software reset
    - Can be done by branching to the reset vector (0x0000)
- Undefined instruction
  - Occurs when the processor or coprocessors cannot recognize the currently execution instruction

ARM Exceptions Types

- Software Interrupt (SWI)
  - User-defined interrupt instruction
  - Allow a program running in User mode to request privileged operations that are in Supervisor mode
- Prefetch Abort
  - Fetch an instruction from an illegal address, the instruction is flagged as invalid
  - However, instructions already in the pipeline continue to execute until the invalid instruction is reached and then a Prefetch Abort is generated

ARM Exceptions Types

- Data Abort
  - A data transfer instruction attempts to load or store data at an illegal address
- IRQ
  - The processor external interrupt request pin is asserted and the I bit in the CPSR is set
- FIQ
  - The processor external interrupt request pin is asserted and the I and F bit in the CPSR are set
- Breakpoint (BKPT)
  - Similar to prefetch abort
  - Prefetch abort occurs at Execute stage of the pipeline

Vector Table

- At the low or high of the memory map
  - Low: 0x00000000
  - High: 0xFFFF0000
- Each entry has only 32 bit
  - Not enough to contain the full code for a handler
  - Branch instruction or load PC instruction to the actual handler

Vector Table

Exception type	Mode	Normal address	High vector address
Reset	Supervisor	0x00000000	0xFFFF0000
Undefined instructions	Undefined	0x00000004	0xFFFF0004
Software interrupt (SWI)	Supervisor	0x00000008	0xFFFF0008
Prefetch Abort (instruction fetch memory abort)	Abort	0x0000000C	0xFFFF000C
Data Abort (data access memory abort)	Abort	0x00000010	0xFFFF0010
IRQ (interrupt)	IRQ	0x00000018	0xFFFF0018
FIQ (fast interrupt)	FIQ	0x0000001C	0xFFFF001C

Exception priorities

Priority		Exception
Highest	1	Reset
	2	Precise data abort
	3	FIQ
	4	IRQ
	5	Prefetch abort
	6	Imprecise data abort
Lowest	7	BKPT Undefined instruction SVC SMC

Exception handling

- Entering an exception handler (Hardware)
  - The processor saves the current status and the return address
  - Enters a specific mode
  - Disables hardware interrupts
  - Execution is then forced from a fixed memory address called an exception vector
- Exit from an exception handler (Software)
  - CPSR = SPSR_[exception_mode]
  - PC = R14

SWI and Undefined Instruction

- SWI and undefined instruction exceptions are generated by the instruction itself
  - lr_mode = pc + 4 //next instruction
- Restoring the program counter
  - If not using stack: MOVS pc, lr //pc = lr
  - If using stack to store the return address
    - STMFD sp!, {reglist, lr} //when entering the handler

      …

      LDMFD sp!, {reglist, pc}^ //when leaving the handler

FIQ and IRQ

- FIQ and IRQ are generated only after the execution of an instruction
- The program counter has been updated

- *lr_mode* = PC + 4
  - Point to one instruction beyond the end of the instruction in which the exception occurred

FIQ and IRQ

- Restoring the program counter
  - If not using stack: SUBS pc, lr, #4 //pc = lr-4
  - If using stack to store the return address
    - SUB lr, lr, #4 //when entering the handler

      STMFD sp!, {reglist, lr}

      …

      LDMFD sp!, {reglist, pc}^ //when leaving the handler

Prefetch Abort

- If the processor supports MMU (Memory Management Unit)
  - The exception handler loads the unmapped instruction into physical memory
  - Then, uses the MMU to map the virtual memory location into the physical one
- The handler must return to retry the instruction that caused the exception
- The *lr_ABT* points to the instruction at the address *following* the one that caused the abort exception

Prefetch Abort

- The address to be restored is at *lr_ABT – 4*
  - If not using stack: SUBS pc,lr,#4
  - If using stack to store the return address
    - SUB lr,lr,#4 ;handler entry code

      STMFD sp!,{reglist,lr}

      …

      LDMFD sp!,{reglist,pc}^ ; handler exit code

Data Abort

- lr_ABT points *two instructions beyond* the instruction that caused the abort (lr_mode = pc + 4)
  - When a load or store instruction tries to access memory, the program counter has been updated
  - The instruction caused the data abort exception is at *lr_ABT – 8*
- The address to be restored is at *lr_ABT – 8*

Data Abort

- The address to be restored is at *lr_ABT – 8*
  - If not using stack: SUBS pc,lr,#8
  - If using stack to store the return address
    - SUB lr,lr,#8 ;handler entry code

      STMFD sp!,{reglist,lr}

      …

      LDMFD sp!,{reglist,pc}^ ; handler exit code

Caches

Caches

- Pros
  - Speed things up
- Cons
  - program execution time can become non-deterministic
  - lack coherence
  - some extra work to manage

Cache types

- Level 1 (L1) caches
  - connected directly to the processor logic
  - Size: 16KB or 32KB
- Level 2 (L2) caches
  - It can be inside the processor itself (Cortex-A8 and A15) or be implemented as an external block
  - Size: 256KB, 512KB or 1MB

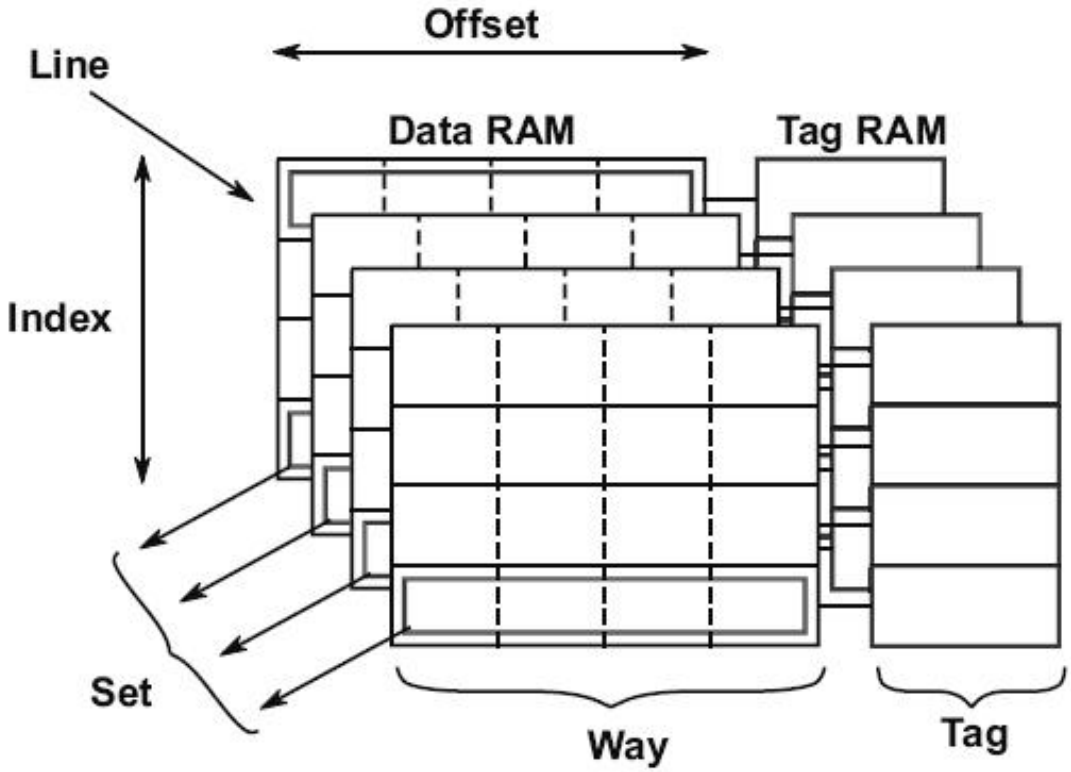
Cache terminology

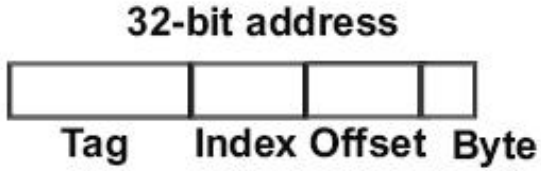
- *Line*
  - the smallest loadable unit of a cache, a block of contiguous words from main memory
- *Index*
  - the part of a memory address which determines in which line(s) of the cache the address can be found
- *Way*
  - a subdivision of a cache, each way being of equal size and indexed in the same fashion

Cache terminology

- *Set*
  - The line associated with a particular index value from each cache way grouped together
- *Tag*
  - the part of a memory address stored within the cache which identifies the main memory address associated with a line of data

Cache terminology





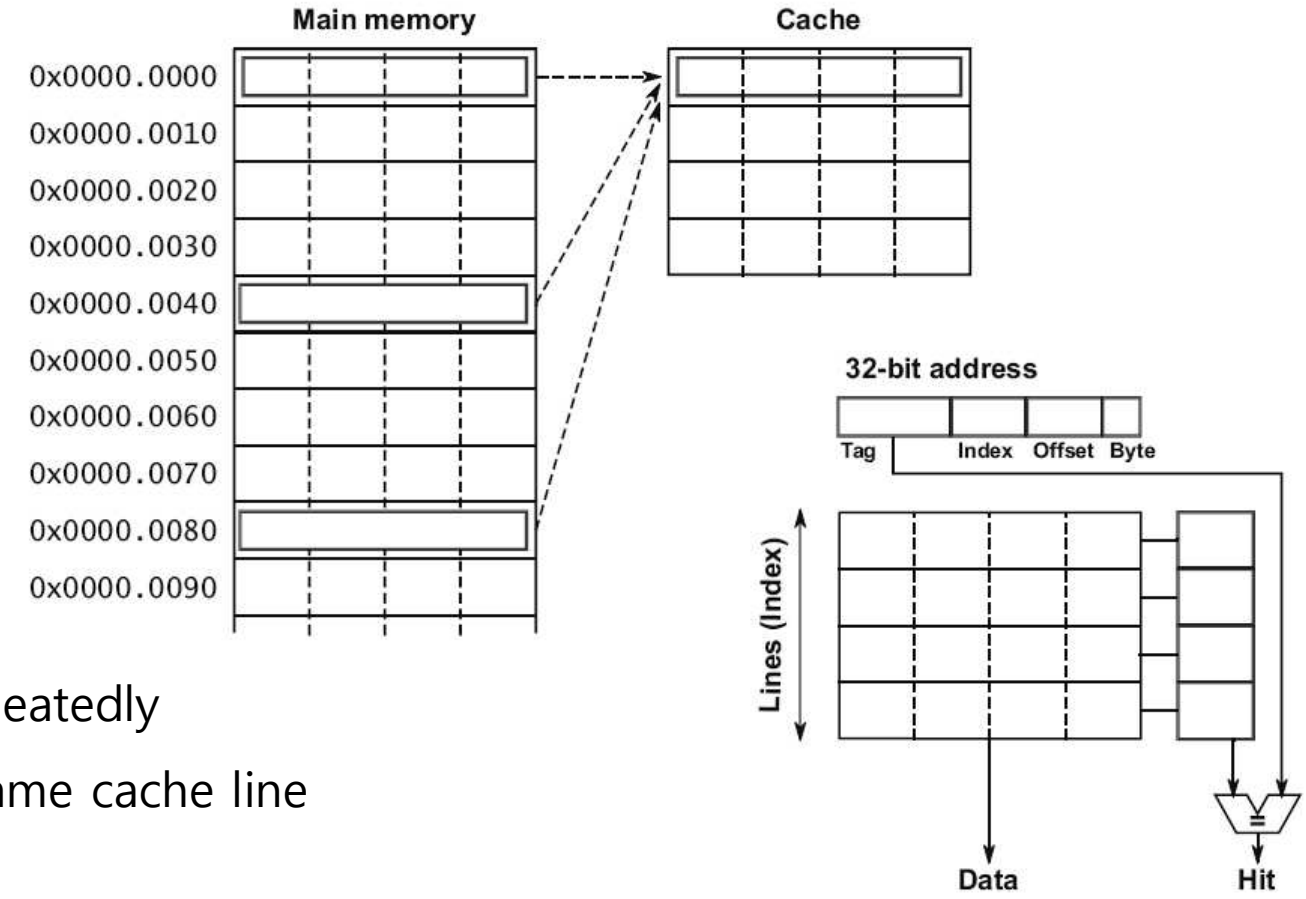
Implementing caches

- Direct mapped caches
- Pros
  - Simple
- Cons
  - Thrashing
    - Load

      and

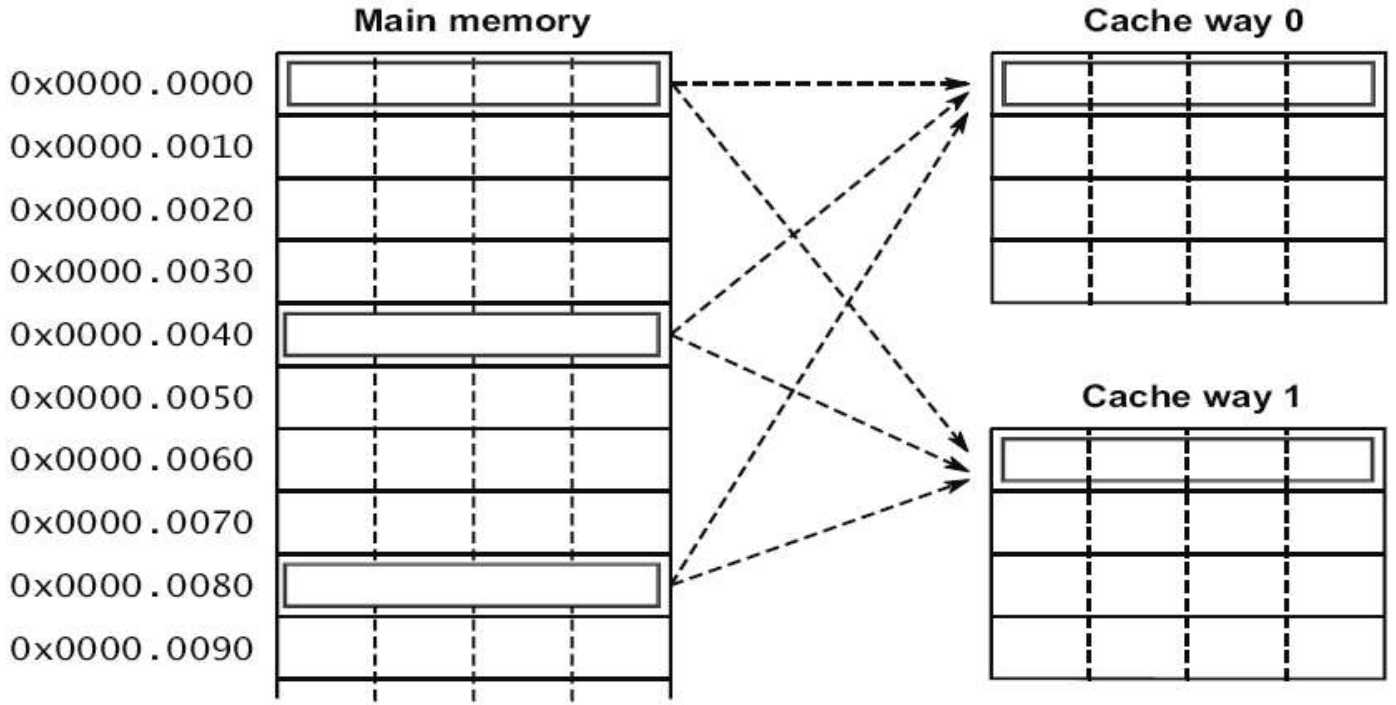
      evict repeatedly

      at the same cache line



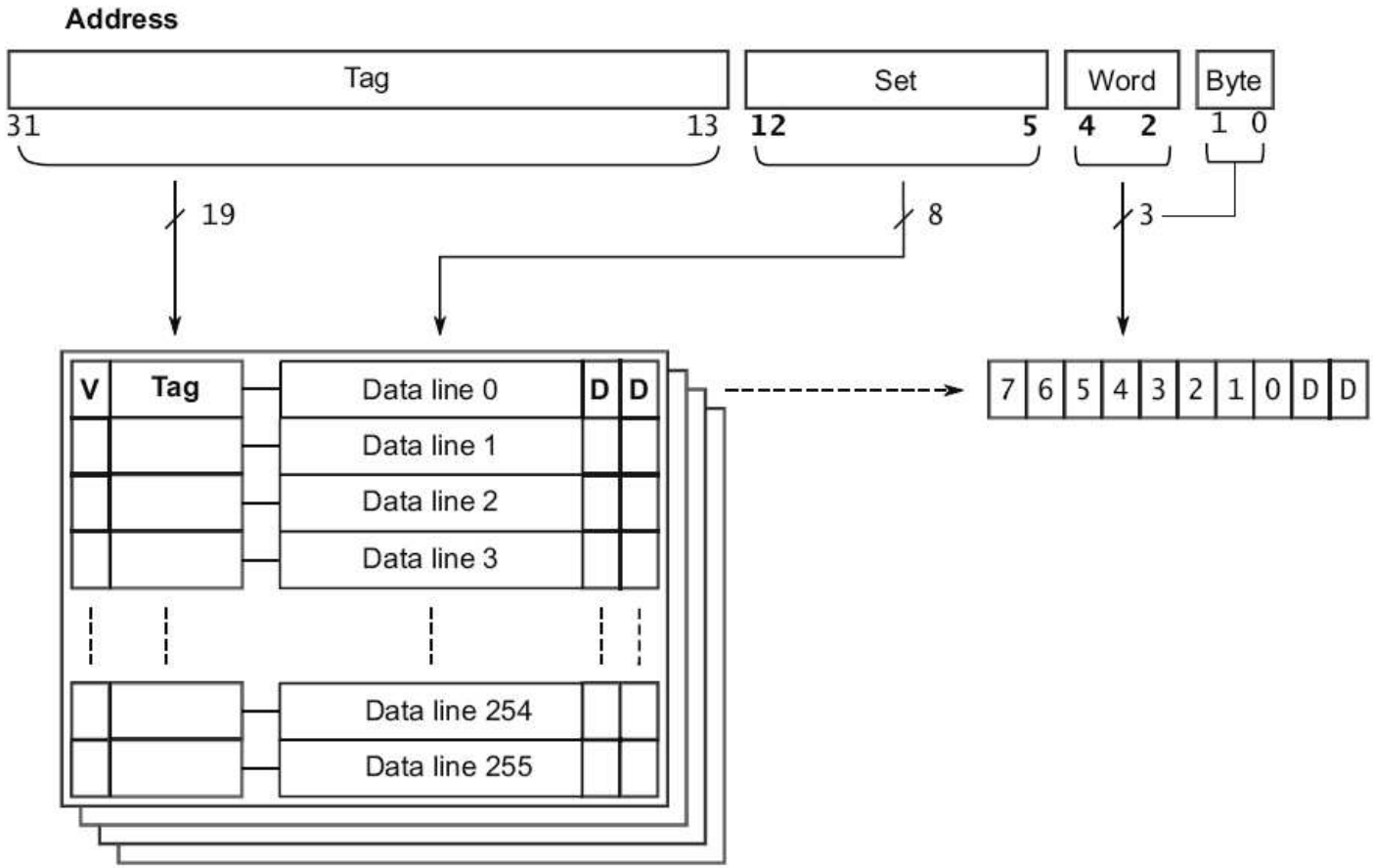
Set associative caches

- To reduce thrashing problem
  - Tag + set index + offset

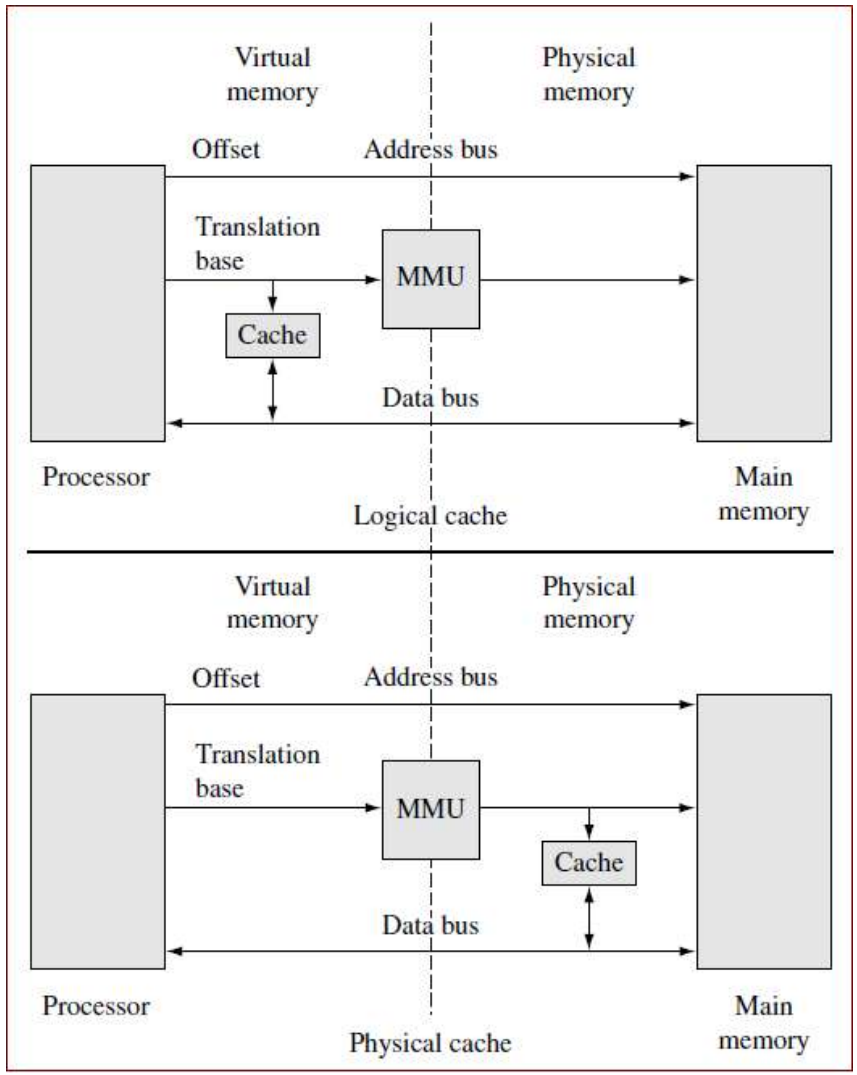


Set associative caches

- A real-life example - 32KB 4-way set associative cache



Physical Cache vs. Logical Cache



Virtual and physical tags and indexes

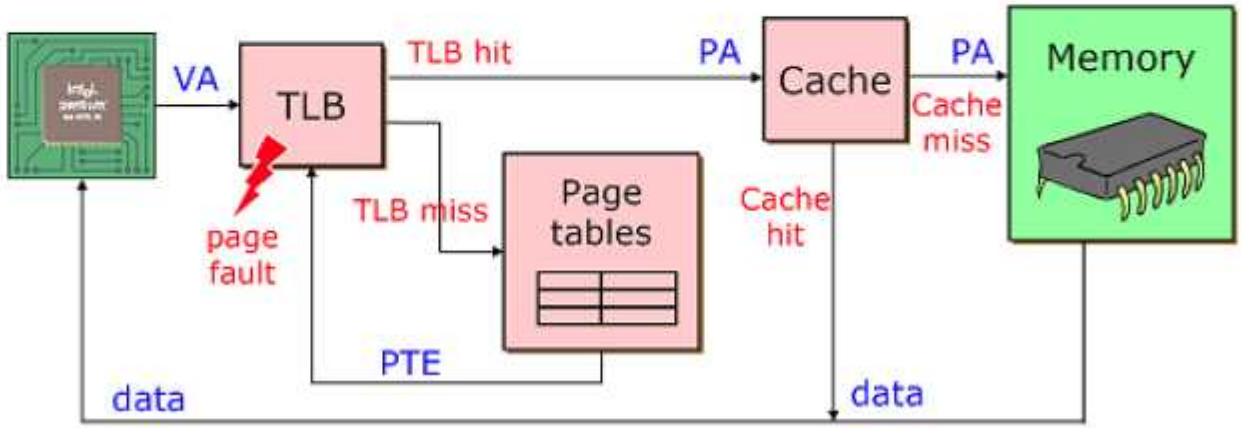
- Early ARM processors
  - used virtual addresses to provide both the index and tag values
  - +: the processor can do a cache look-up without the need for a virtual to physical address translation
  - -: changing the virtual to physical mappings in the system means that the cache must first be cleaned and invalidated
- ARM11 family processors
  - Index: Virtual address / Tag: Physical address
  - Virtually Indexed, Physically Tagged (VIPT)
  - +: Reduce cache clean and flush
  - -: During cache lookup, it should access MMU

Virtual and physical tags and indexes

- Cortex-A8 process
  - a VIPT implementation in its instruction cache, but not its data cache
- Recent Cortex-A series
  - Physically Indexed, Physically Tagged (PIPT) cache

PIPT
(Physically indexed, physically tagged)

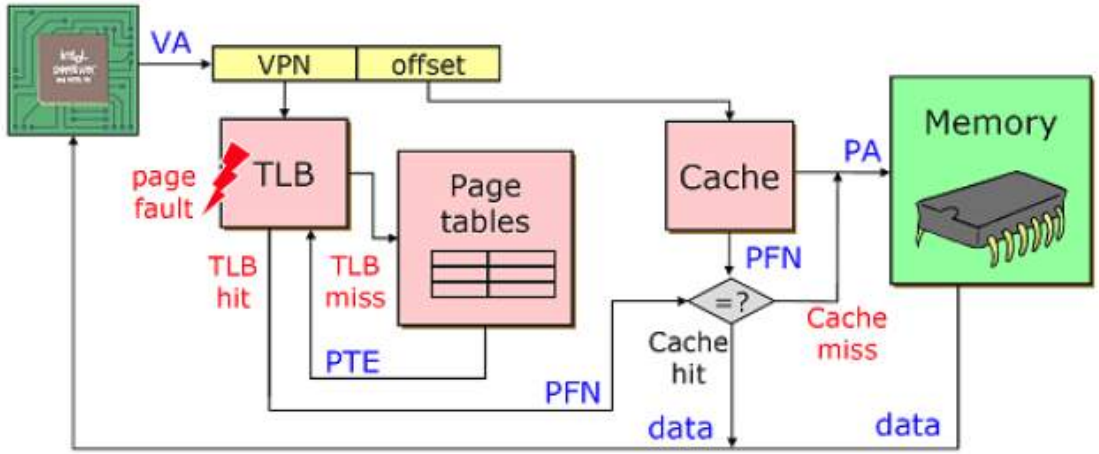
index와 tag에 모두 physical address가 사용됨. 가장 간단한 방법이고 physical address는 유일하기 때문에 address aliasing문제가 없지만, physical address로 변경이 필요하게 됨. 그 의미는 TLB의 개입이 필요하고, TLB miss가 발생할 시에 속도가 떨어짐. TLB miss가 발생하지 않는다 하더라도, TLB lookup 후에 Cache lookup을 sequential하게 이루어져야 함. 따라서 전체적으로 속도가 slow.



<출처: iamroot Kernel 6차 Members>

VIPT
(Virtually indexed, physically tagged)

index에는 virtual address를 사용하고 tag에는 physical address를 사용함. PIPT 대비 low latency (physical address를 찾는데 걸리는 시간)의 장점이 있음. TLB를 통해 cache line을 parallel 하게 찾을 수 있음. 하지만, physical address를 찾을 때까지 tag를 비교할 수는 없음. 왜냐하면 tag가 physical address를 사용하기 때문에. processor가 참조하는 Index는 virtual address이기 때문에 이를 physical address로 변경이 필요하다는 의미임. VIVT 대비 장점은 tag가 physical address이기 때문에 cache가 aliasing을 검출할 수 있음. VIPT는 약간의 tag bit가 좀 더 필요한데, 왜냐하면 index bit가 더이상 동일한 address를 표현하지 않기 때문임.



<출처: iamroot Kernel 6차 Members>

VIVT
(Virtually indexed, virtually tagged)

index와 tag에 모두 virtual address가 사용됨. virtual address가 사용되기 때문에 physical address를 찾기 위해 MMU가 개입될 필요가 없음. 하지만, 서로다른 virtual addresses가 같은 physical address를 가리키는 aliasing문제가 발생함. 이는 physical address에 서로다른 virtual address가 cache될 때 발생하는데, coherency problem을 발생시킴. 예를 들면, 서로다른 사용자 application이 동일한 file을 mmap할 때 사용자 application의 virtual address는 다르나 실제 physical address는 갖게됨. 다른 문제점으로는 V->P mapping (virtual to physical)이 바뀔 수 있다는 점이며, 이경우 TLB entry의 변경에 주의해야하고 entry가 변경되기 전에 해당 cache line들을 flushing해야 함. 왜냐하면, virtual address가 더이상 유효하지 않기 때문에.

<출처: iamroot Kernel 6차 Members>

Cache policies

- Allocation policy
- Replacement policy
- Write policy

Allocation policy

- read allocate
  - allocates a cache line only on a read
  - If a write is performed by the processor which misses in the cache, the cache is not affected and the write goes to main memory
- read-write cache allocate
  - allocates a cache line for either a read or write which misses in the cache

Replacement policy

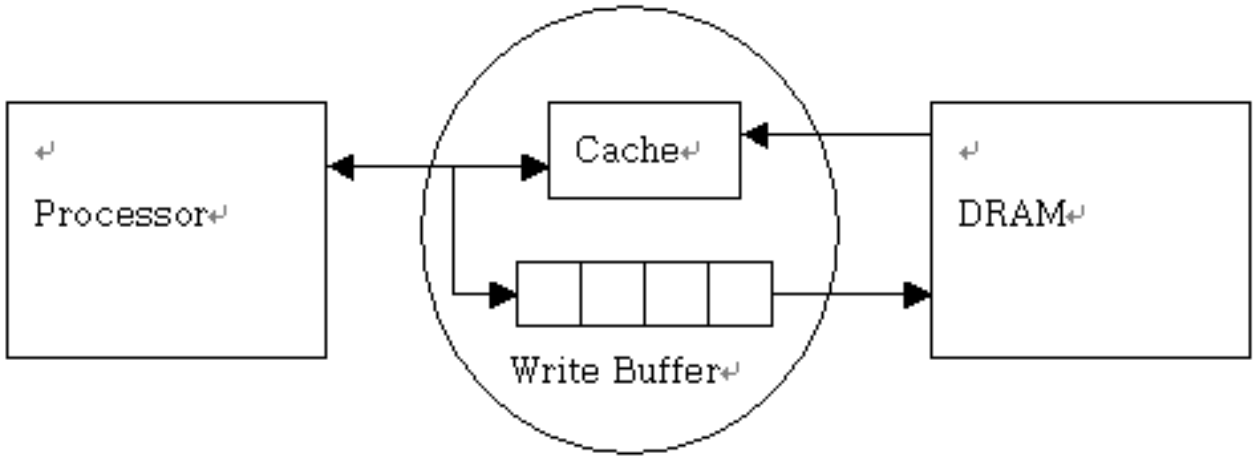
- Round-robin or cyclic replacement
  - Increment the victim counter by one
- Pseudo-random replacement
  - randomly selects the next cache line in a set to replace

Write policy

- Write-through
  - With this policy writes are performed to both the cache and main memory
  - the cache and main memory are kept coherent
- Write-back
  - writes are performed only to the cache, and not to main memory
  - cache lines and main memory can contain different data
  - a dirty line should be written to main memory before eviction

Write buffer

- implemented using a number of FIFOs
- The processor does not have to wait for the write to be completed to main memory
- increase the performance of the system



Invalidating cache memory

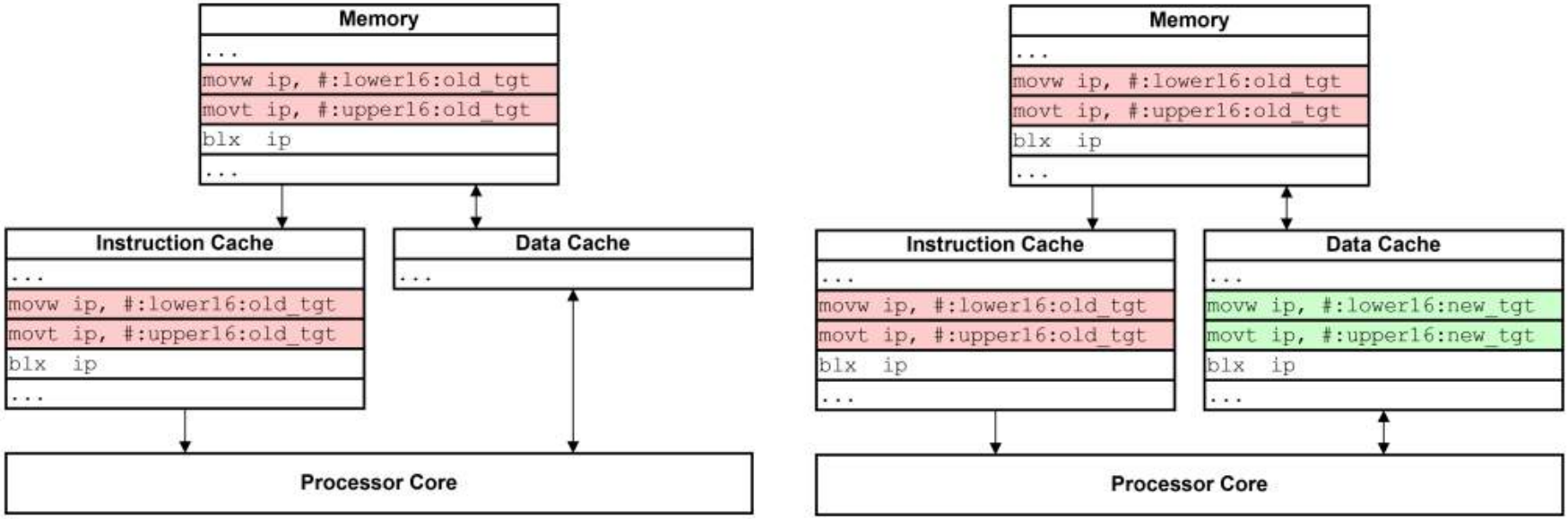
- Invalidation of a cache (or cache line) means to clear it of data
- The cache always needs to be invalidated after reset automatically except Cortex-A9 processor
- write-back cacheable regions would be lost by simple invalidation

Cleaning cache memory

- write the contents of dirty cache lines out to main memory and clear the dirty bit(s) in the cache line
- It is only applicable for data caches in which a write-back policy is used

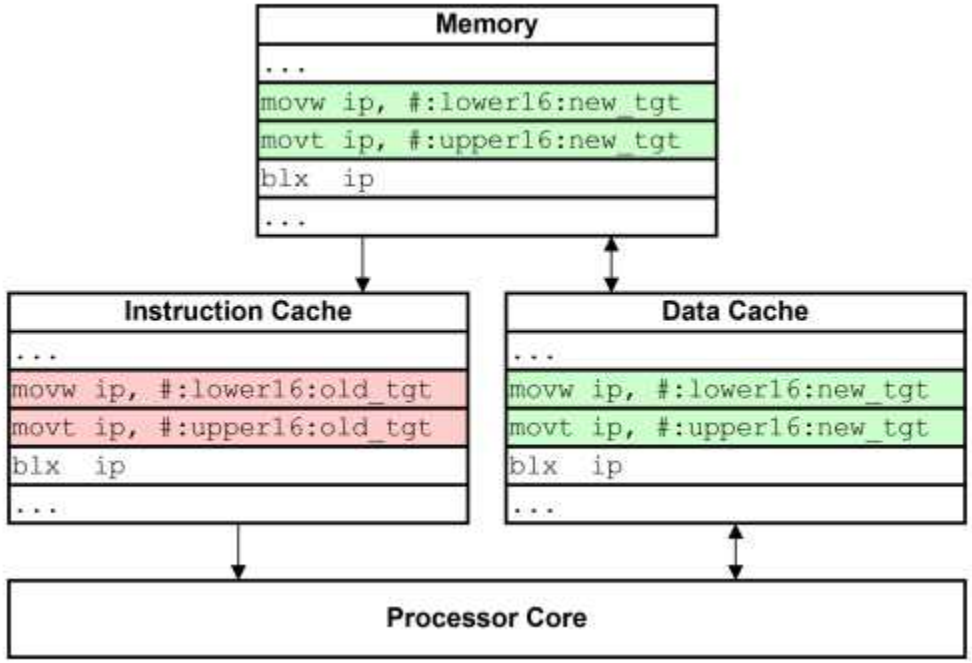
Self-modifying code

- Problem
  - Some existing JIT-compiled code was generated at run-time to load a function address into a register and then branch to it
  - How to execute the modified code?



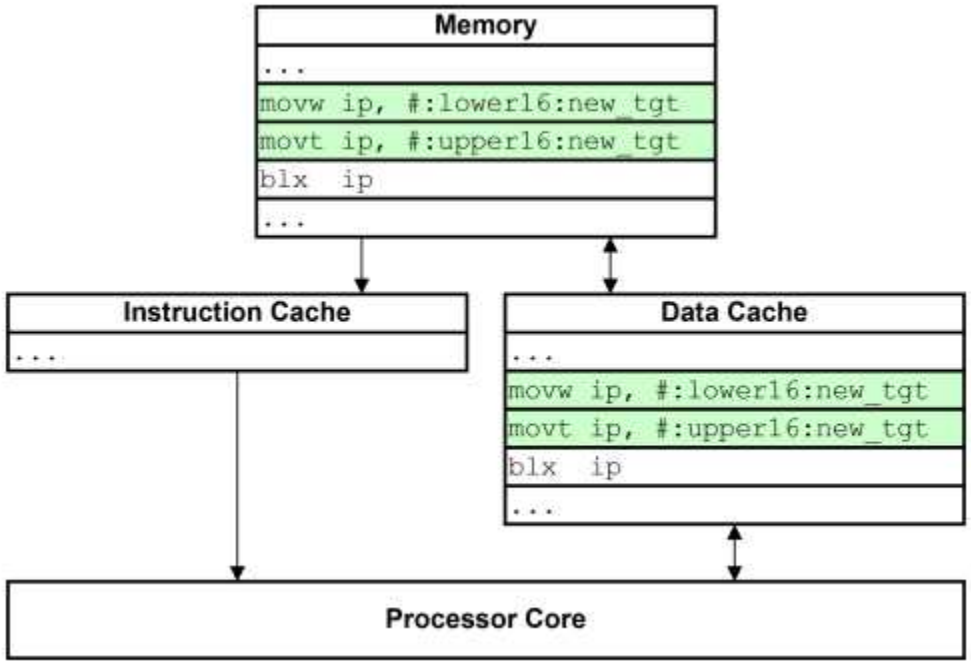
Self-modifying code

- Solution
  - get data from the D-cache into the I-cache
  - to push the data out to memory => clearing D-cache lines
  - wait for the write to complete



Self-modifying code

- Solution
  - tell the processor that the contents of the I-cache are invalidated and need to be re-loaded from memory
  - The commands to clean and/or invalidate the cache are CP15 operations



Cache lockdown

- the programmer to place critical code/or and data into cache memory and protect it from eviction
- For avoiding any cache miss penalty
- Ex> code and data of a critical interrupt handler

Tightly Coupled Memory

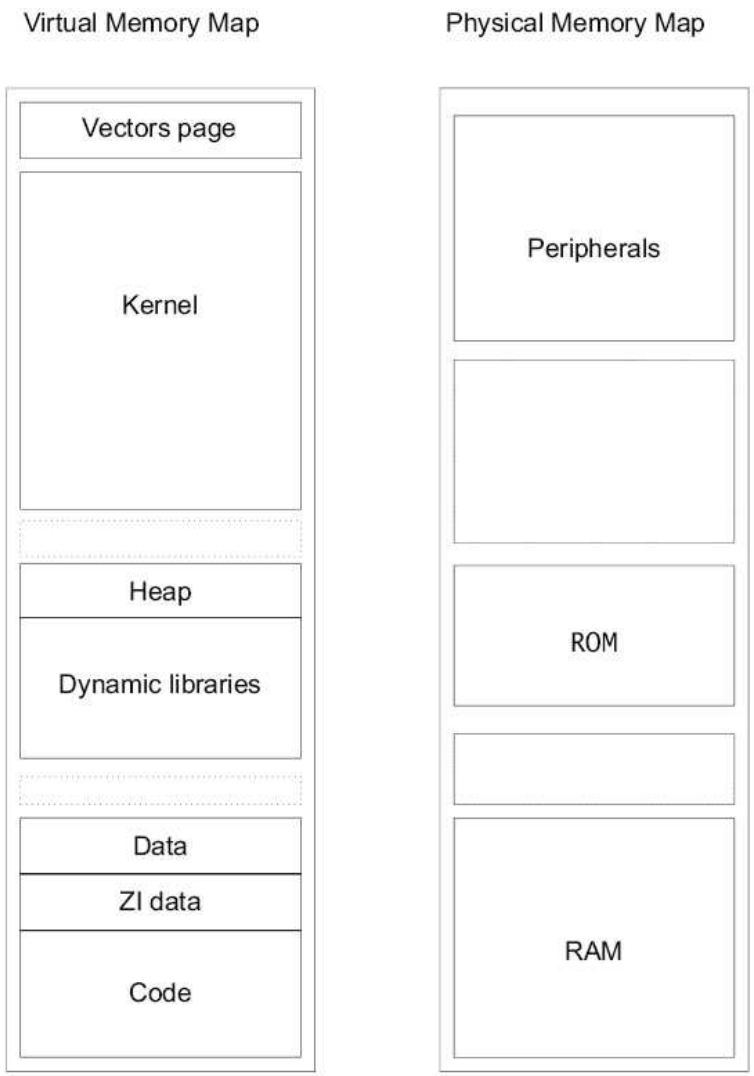
- An alternative to using cache lockdown (H/W approach)
- a block of fast memory located next to the processor core
- Hold instructions or data required for real-time code which needs deterministic behavior
- However, TCM is not supported in Cortex-A series processors

Memory Management Unit

What for?

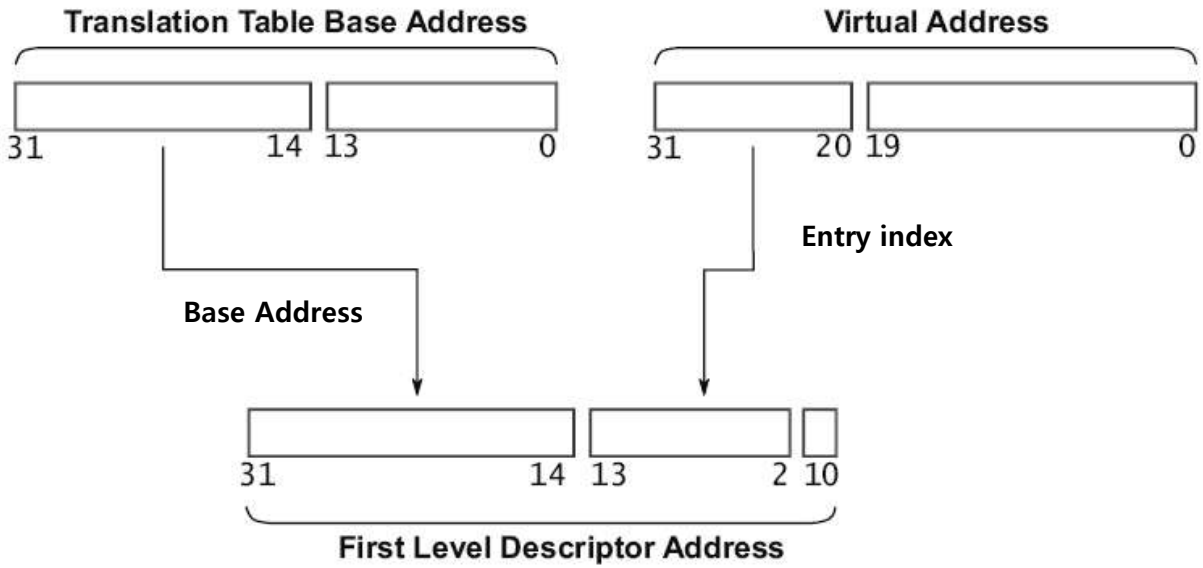
- need a way to partition the memory map and assign permissions and memory attributes to these regions of memory
- programming of applications much simpler
- virtual address space is separate from the actual physical map of memory in the system
- translation of virtual addresses into physical addresses

Virtual and physical memory



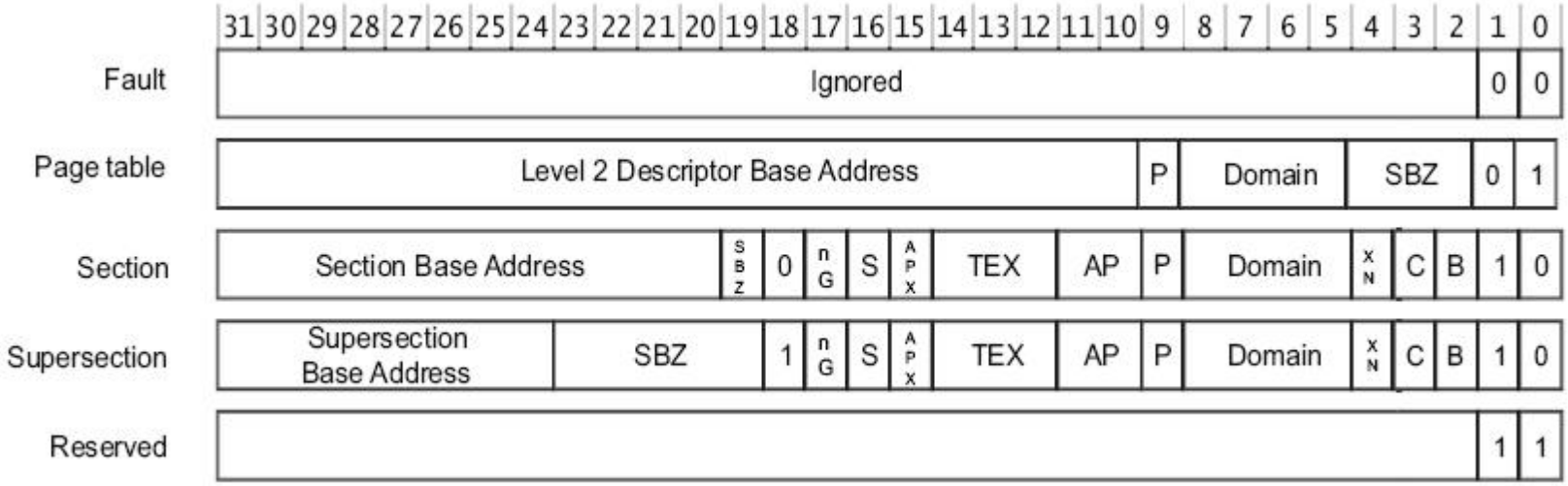
Level 1 page tables

- The base address of the L1 page table is known as the Translation Table Base Address and is held within a register in CP15 c2

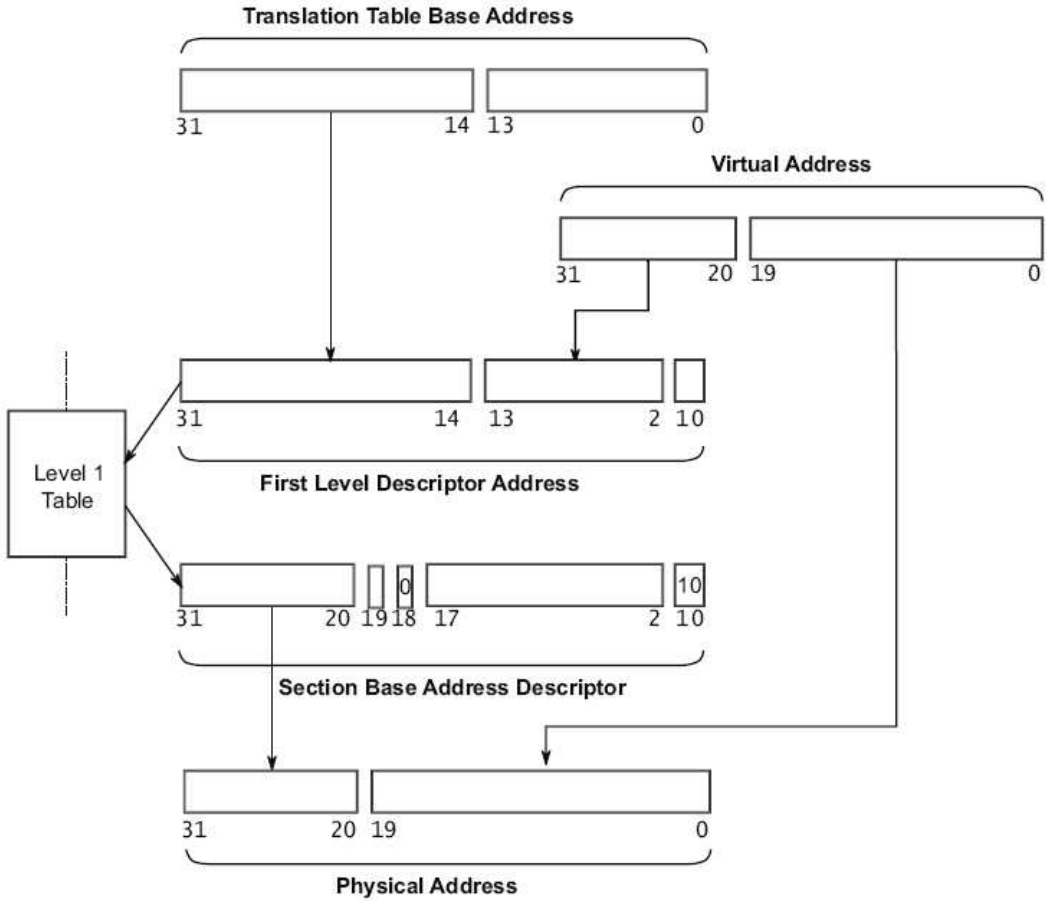


Level 1 page table entry format

- A fault entry that generates an abort exception
- An entry that points to an L2 page table
- A 1MB section translation entry
- A 16MB supersection

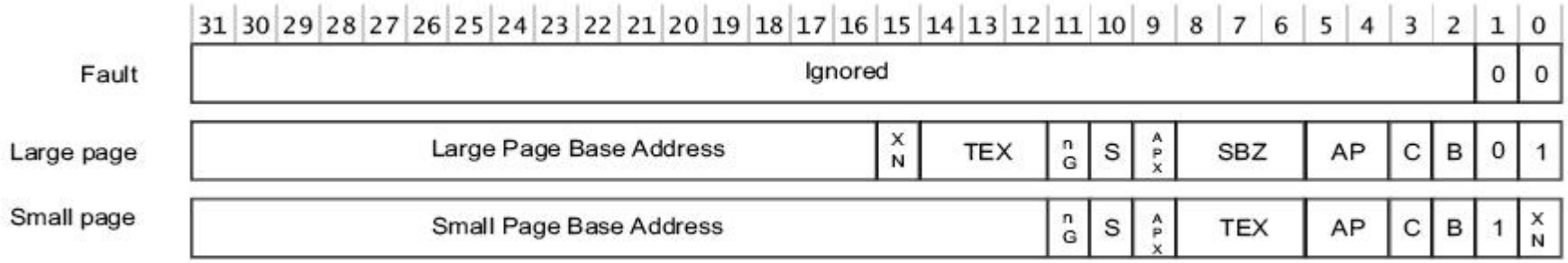


Generating a physical address from a level 1 page table entry

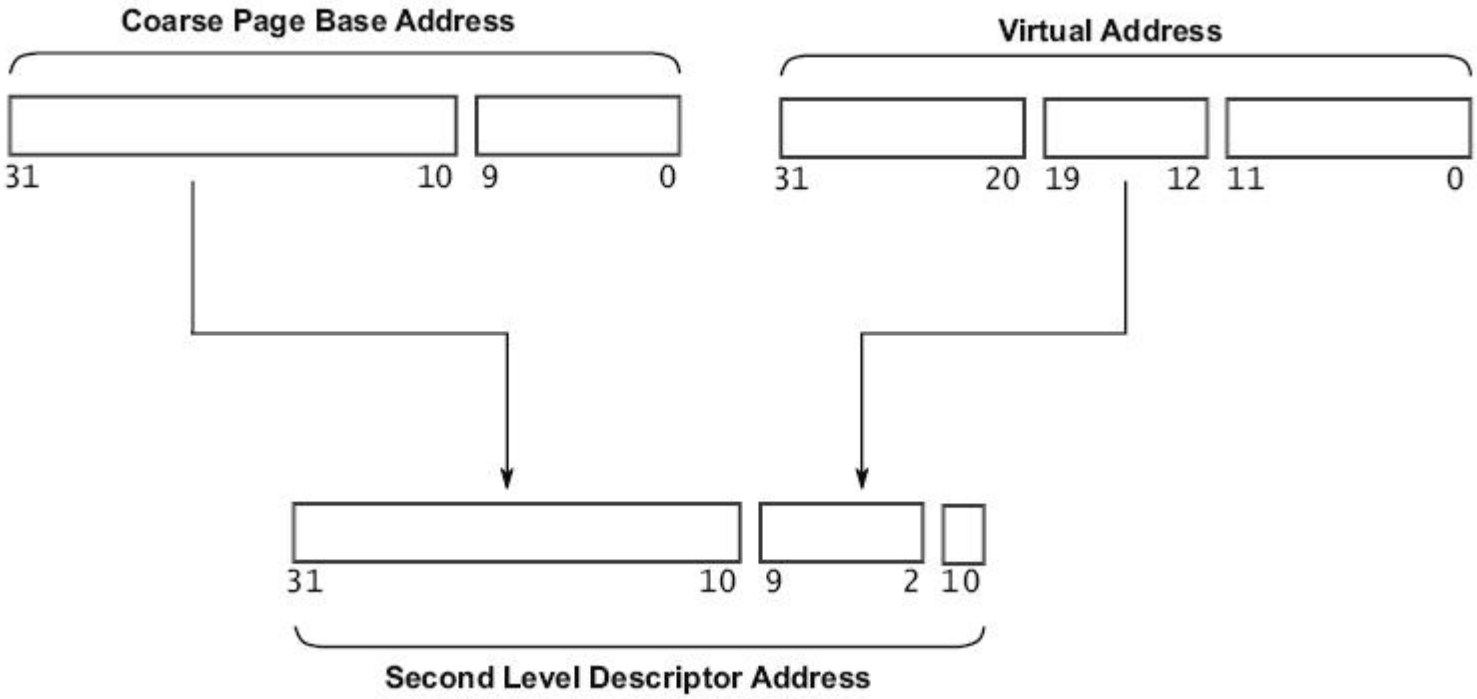


Level 2 page tables

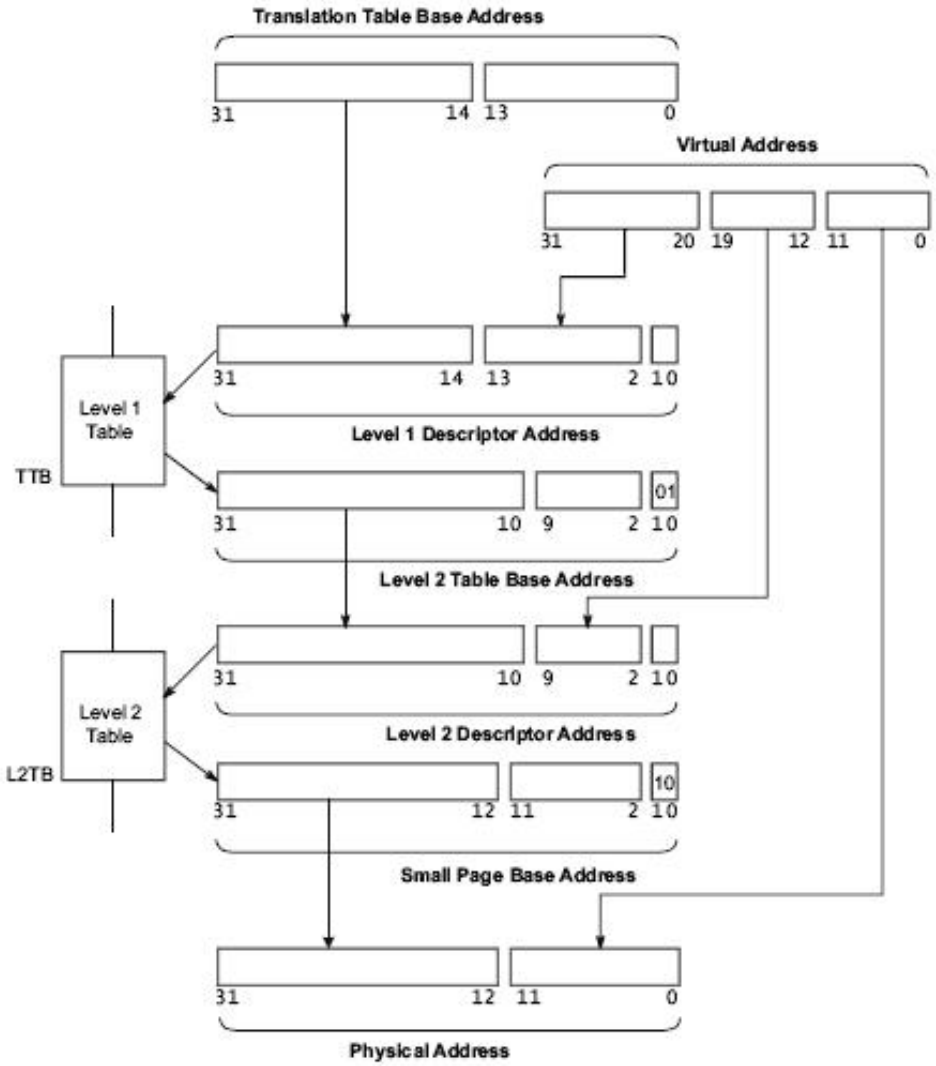
- L2 page table has 256 word-sized entries
  - A fault page entry generates an abort exception if accessed
  - A large page entry points to a 64KB page
  - A small page entry points a 4KB page



Generating the address of the level 2 page table entry



Summary of generation of physical address using the L2 page table entry

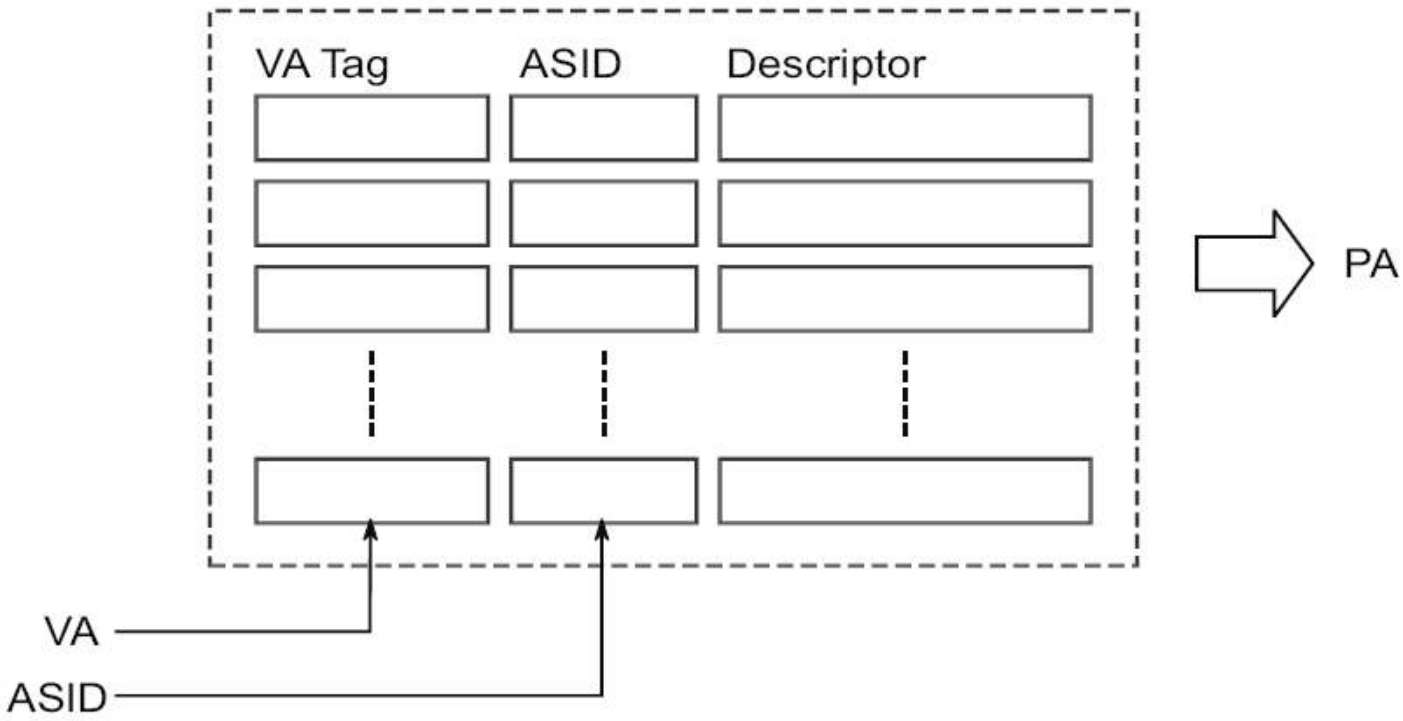


Translation Look-aside Buffer

- a cache of page translations within the MMU
- On a memory access, the MMU first checks whether the translation is cached in the TLB
- TLB hit, and the TLB provides the translation of the physical address immediately

TLB structure

- Attributes: memory type, cache policies, access permissions
- ASID value – Address Space ID



TLB operation

- There are several CP15 operations available
  - It is essential to invalidate the TLB when a valid page table entry is changed
  - a global invalidate of the TLB
  - removal of specific entries
- Support for locking individual entries into the TLB

Choice of page sizes

- Small-size page
  - Smaller page sizes allow finer control of a block of memory and potentially can reduce the amount of unused memory in a page
  - Smaller page sizes also allow increased precision of control over permissions
- Large-size page
  - Each entry in the TLB holds a reference to a larger piece of memory (increase TLB hit ratio)
  - fewer page table walks to slow external memory

Memory attributes

- Memory Access Permissions

APX	AP	Privileged	Unprivileged	Description
0	00	No access	No access	Permission Fault
0	01	Read/Write	No access	Privileged Access only
0	10	Read/Write	Read	No user-mode write
0	11	Read/Write	Read/Write	Full access
1	00	-	-	Reserved
1	01	Read	No access	Privileged Read only
1	10	Read	Read	Read only
1	11	-	-	Reserved

Cache and Write Buffer Bit

I-Cache		D-Cache		
C-bit	Page property	C-Bit	B-Bit	Page property
0	Not cached	0	0	Not cached, Not buffered
1	cached	0	1	Not Cached, Buffered
		1	0	Cached, Write-through
		1	1	Cached, Write-back

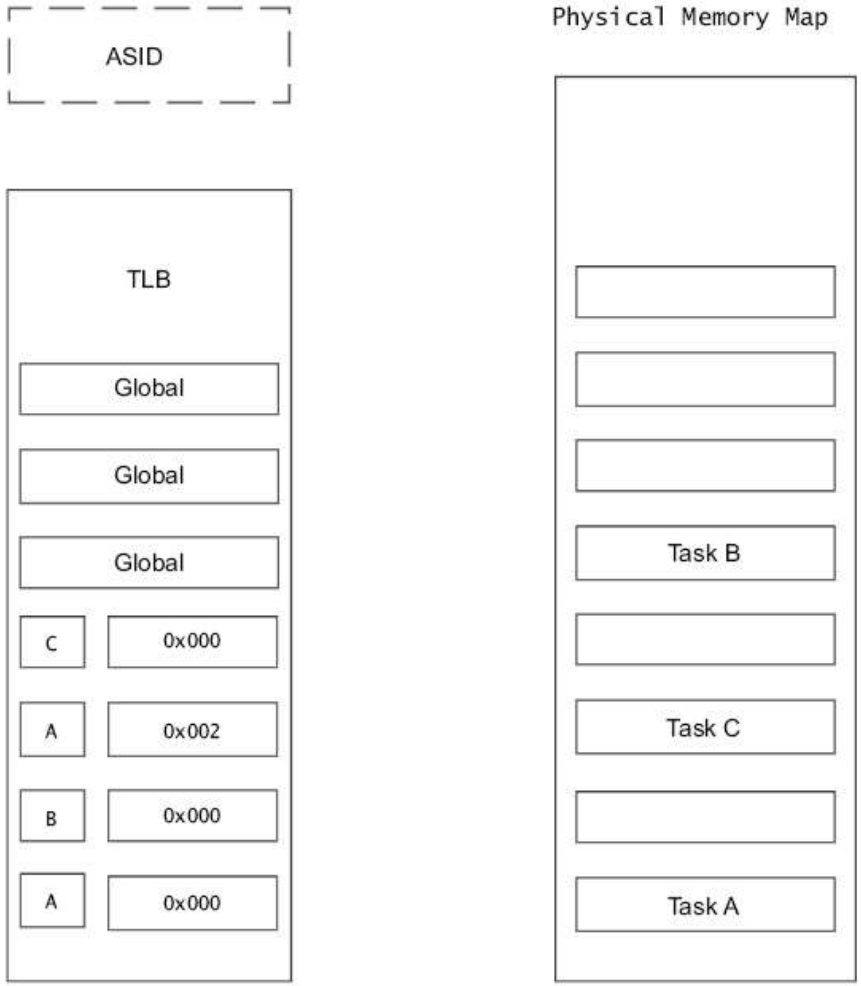
Domains

- The ARM architecture has an unusual feature which enables regions of memory to be tagged with a domain ID
- There are 16 domain IDs provided by the hardware
- CP15 c3 contains the Domain Access Control Register which holds a set of 2-bit permissions for each domain number
- Modes
  - No-access
    - an abort on any access to a page in this domain
  - *Manager*
    - ignores all page permissions and enables full access
  - *Client*
    - the permissions of the pages tagged with the domain

Address Space ID

- If the nG bit is set for a particular page, it means that the page is associated with a specific application and is not global
- This means that when the MMU performs a translation, it uses both the virtual address and an ASID value
- The ASID is a number assigned by the OS to each individual task
- This value is in the range
- 0-255 and the value for the current task is written in the ASID register

ASIDs in TLB mapping same virtual address



Features of Cortex-A15 MMU

- Large Physical Address Extensions (LPAE)
  - Address mapping for a 40-bit address space, 4GB -> 1024GB
  - A new page table entry format - the “long-descriptor format” is added
  - This is set to show that the entry is one of 16 contiguous entries which point to a contiguous output address range. If set, the TLB need cache only one translation for the group of 16 pages
  - “privileged execute never” (PXN) is added. This marks a page as containing code that can be executed only in a non-privileged (user) mode
  - supports virtualization