

Memoria Virtuale - Protection

Andrea Bartolini – a.bartolini@unibo.it

Memory Protection

Memory Protection

→ x problemi di Sicurezza e privacy
la protezione consente di garantire che alcune pagine di memoria fisica non
vengano toccate da chi non ha diritto di farlo.
c'è un meccanismo che fa questo.

- Multiple programs (**processes**) run at once
 - Each process has its own page table
 - Each process can use entire virtual address space without worrying about where other programs are

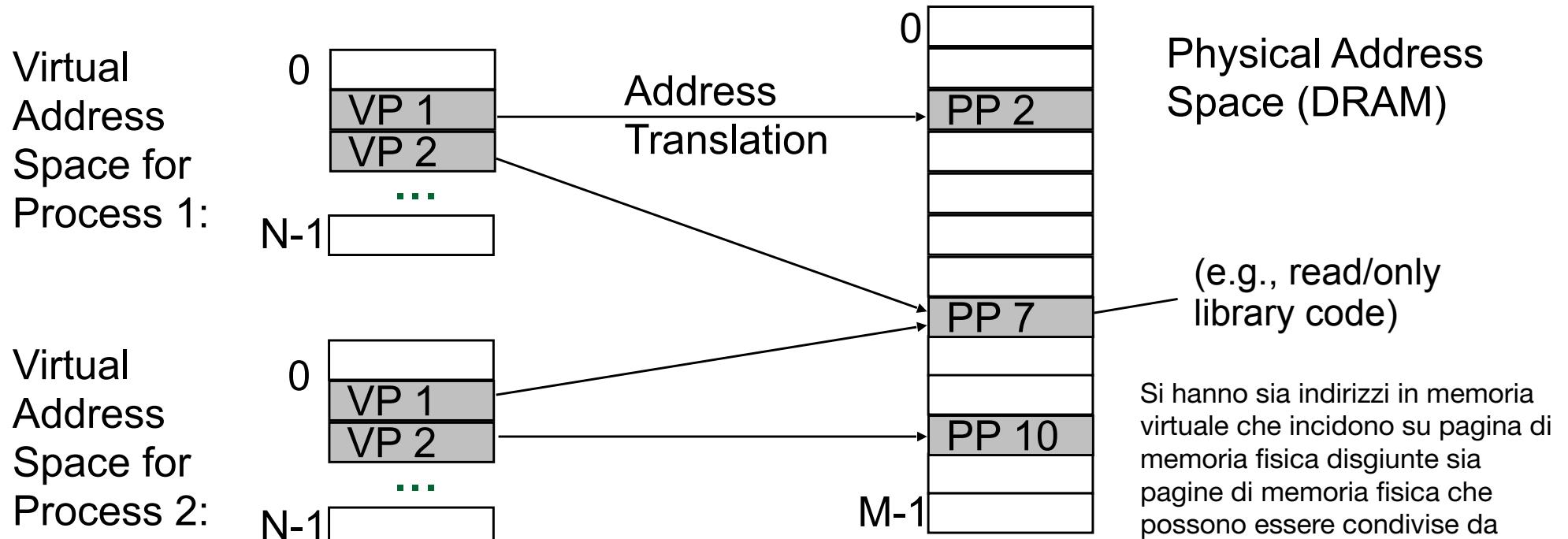
- A process can only access physical pages mapped in its page table – cannot overwrite memory of another process
 - Provides protection and isolation between processes
 - Enables access control mechanisms per page

Ogni processo fra le sue page table e può usare l'intero spazio di indirizzamento virtuale

Ci sono dei bit che identificano se è permesso di accedere o meno (bit di protezione).

Page Table is Per Process

- Each process has its own virtual address space
 - Full address space for each program
 - Simplifies memory allocation, sharing, linking and loading.



Una pagina può avere come etichette dei diritti di accesso, cioè nella page table entry associata al virtual page number n per il processo 2 e alla page table entry associata al virtual page number 0 per il processo 1, sono contenuti dei bit che contengono due diritti di accesso che codificano read e write.

Si hanno sia indirizzi in memoria virtuale che incidono su pagina di memoria fisica disgiunte sia pagine di memoria fisica che possono essere condivise da diverse pagine di memoria virtuale.

Ad esempio se pp7 è una libreria vogliamo che sia read only.

Access Protection/Control via Virtual Memory

Page-Level Access Control (Protection)

- Not every process is allowed to access every page
 - E.g., may need supervisor level privilege to access system pages
 - Idea: Store access control information on a page basis in the process's page table
 - Enforce access control at the same time as translation
- Virtual memory system serves two functions today
- Address translation (for illusion of large physical memory)
 - Access control (protection)

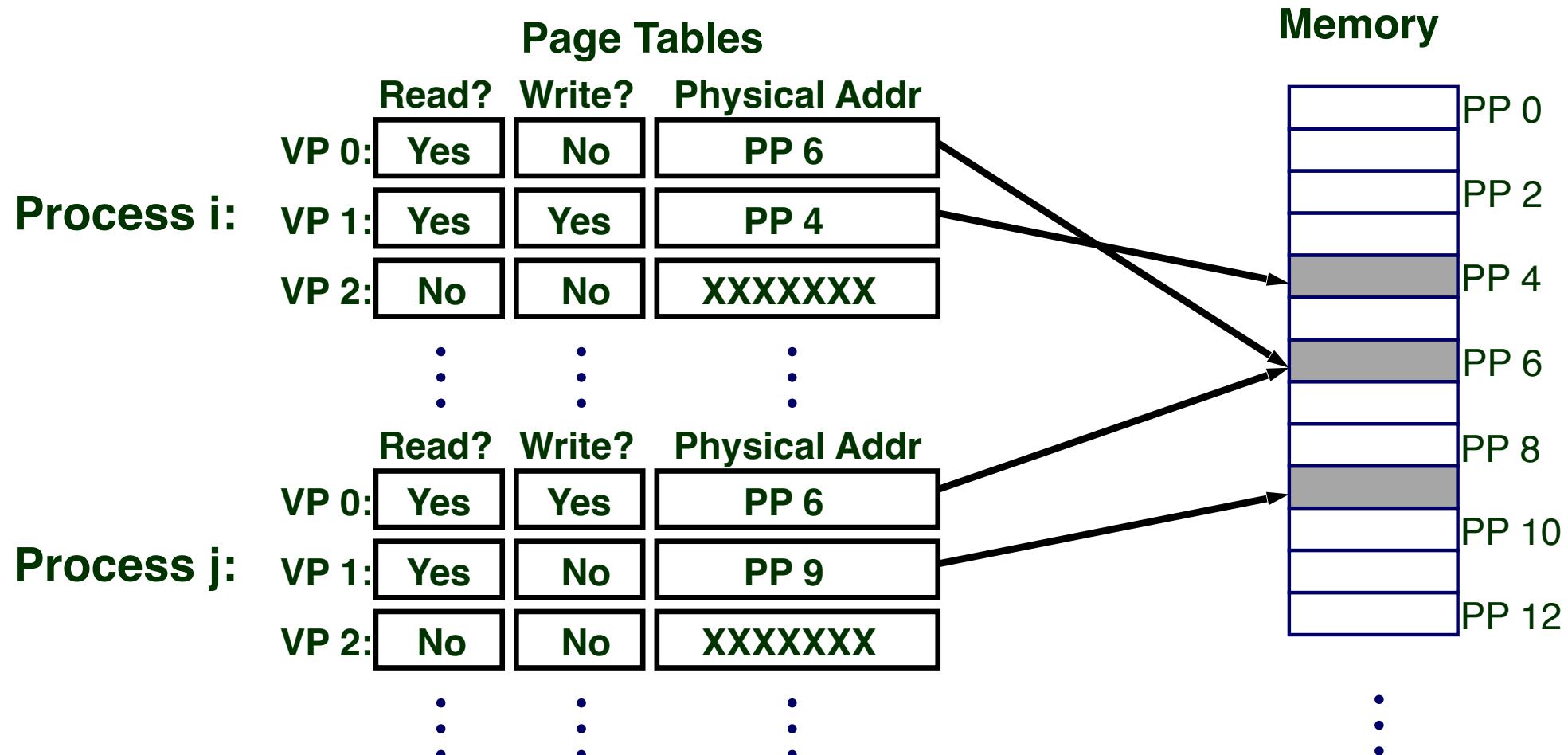
VM as a Tool for Memory Access Protection

ci evita di entrare nella page table

TLB: cache che contiene dei virtual page number e physical page number associato.

E. TLB con 16 traduzioni.

- Extend Page Table Entries (PTEs) with permission bits
- Check bits on each access and during a page fault
 - If violated, generate exception (Access Protection exception)



Essendo che il mapping tra virtual page number e physical page number è proprietà del processo, quindi dovrei invalidare completamente il tlb quando switcho contesto (programma).

La page table entry è indirizzata al virtual page number

All'interno della page table entry si accede con il virtual page number. Con il virtual page number si identifica una page table entry e lo si fa moltiplicando il virtual page number per la page table entry size e il risultato si somma al PTBR . Questa cosa la fa L hardware che traduce.

Nella page table entry c'è il Valid bit il dirty bit e dei bit di accesso (uno per la read e uno per la write , che ci dice se sono abilitate o no).

Anche lo stadio di fetch passa dall' MMU per accedere alla memoria, anche le pagine che contengono istruzioni sono pagine di memoria virtuale che saranno mappate in pagine di memoria fisica tramite una page table. Quindi anche gli accessi verso la memoria delle istruzioni avvengono attraverso una mmu , quindi ci sarà un tlb anche per le istruzioni oltre che per i dati.

Comunque la page table sarà sempre nella memoria dati, quindi quando la mmu cerca la traduzione , sia che sia per un indirizzo di messo dalla fetching unit che dalla load store unit, va in memoria dati a cercarla . Se la traduzione è già nel tlb penso nom si vada in memoria

Privilege Levels in x86

Ma

Il livello di privilegio totale, che è concesso al s.o. Non è effettivamente totale al 100%.

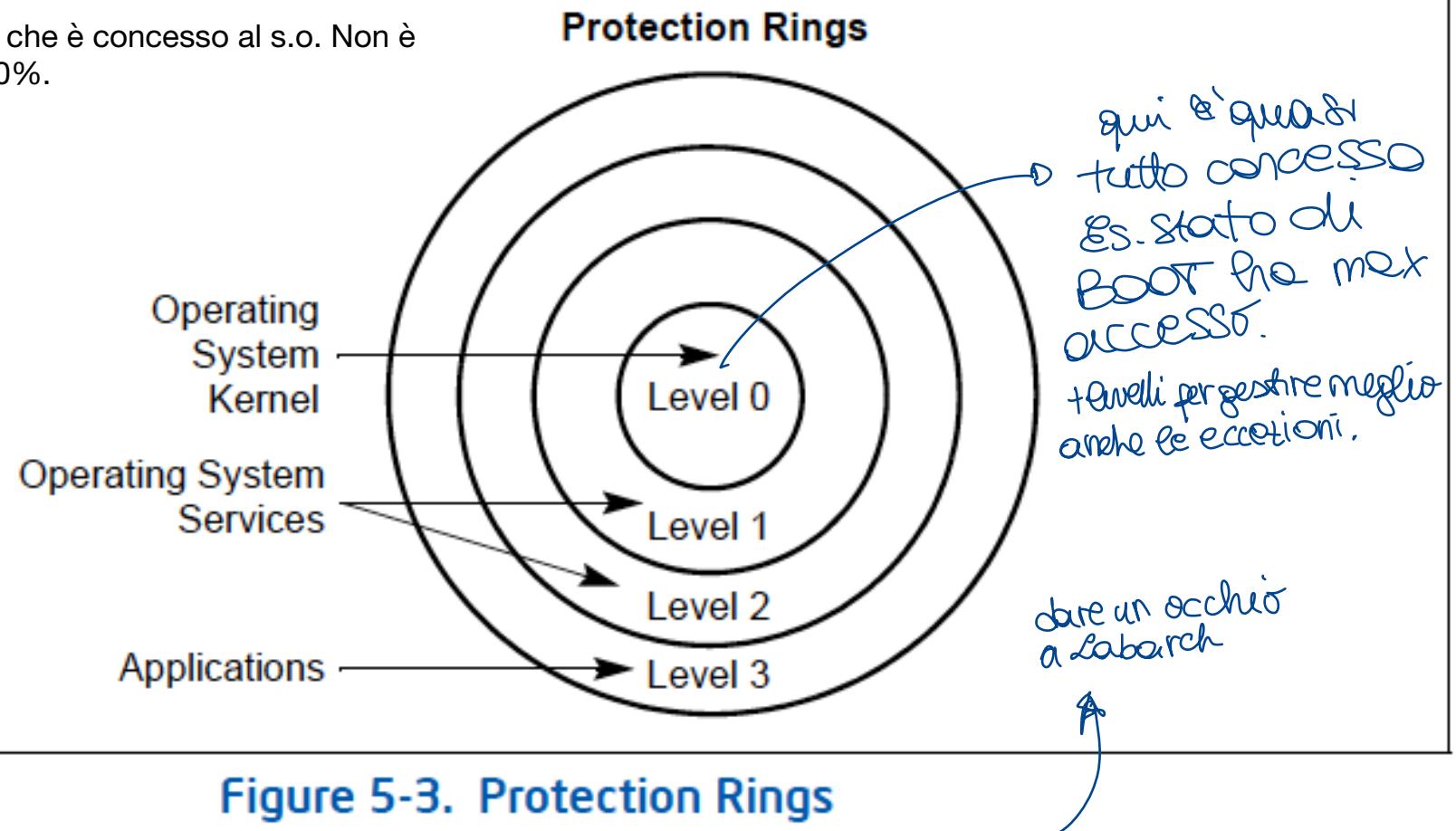


Figure 5-3. Protection Rings

La tabella delle pagine è modificabile dall'utente? Se si potrebbe modificare i bit di protezione da lettura o scrittura e fare quello che vuole. C'è bisogno che quindi la tabella non possa essere modificata dall'utente generico mentre si fa dall'ipervisor. Questo concetto è mappato nei concetti di privilegio. Il livello di privilegio più alto equivale ad avere restrizioni maggiori (level 3). Quando si esegue il codice di una applicazione lo si fa a un livello di privilegio più basso rispetto al codice del codice operativo.

Some Issues in Virtual Memory

Omonimia: stesso virtual page number che identifica due pagine di memoria fisica differente.

siccome possono esserci più processi, e ciascuno può usare l'intero spazio di memoria virtuale, allora probabilmente avrò lo stesso virtual page number che farà riferimento a physical page number di processi diversi.

Sinonimia: quando ho differenti indirizzi di memoria virtuale, quindi diversi virtual page number che fanno riferimento allo stesso physical page number . Questo avviene dove ho pagine di memoria virtuale che sono proprietà di processi diversi che condividono la stessa pagina di memoria fisica. (Es. librerie di sistema).

Three Major Issues



-
1. How large is the page table and how do we store and access it? Quanto grande una pagina è quanto grande la tabella?
 2. How can we speed up translation & access control check?
 3. When do we do the translation in relation to cache access?
-
- There are many other issues we will not cover in detail
 - What happens on a context switch?
 - How can you handle multiple page sizes?
 - ...

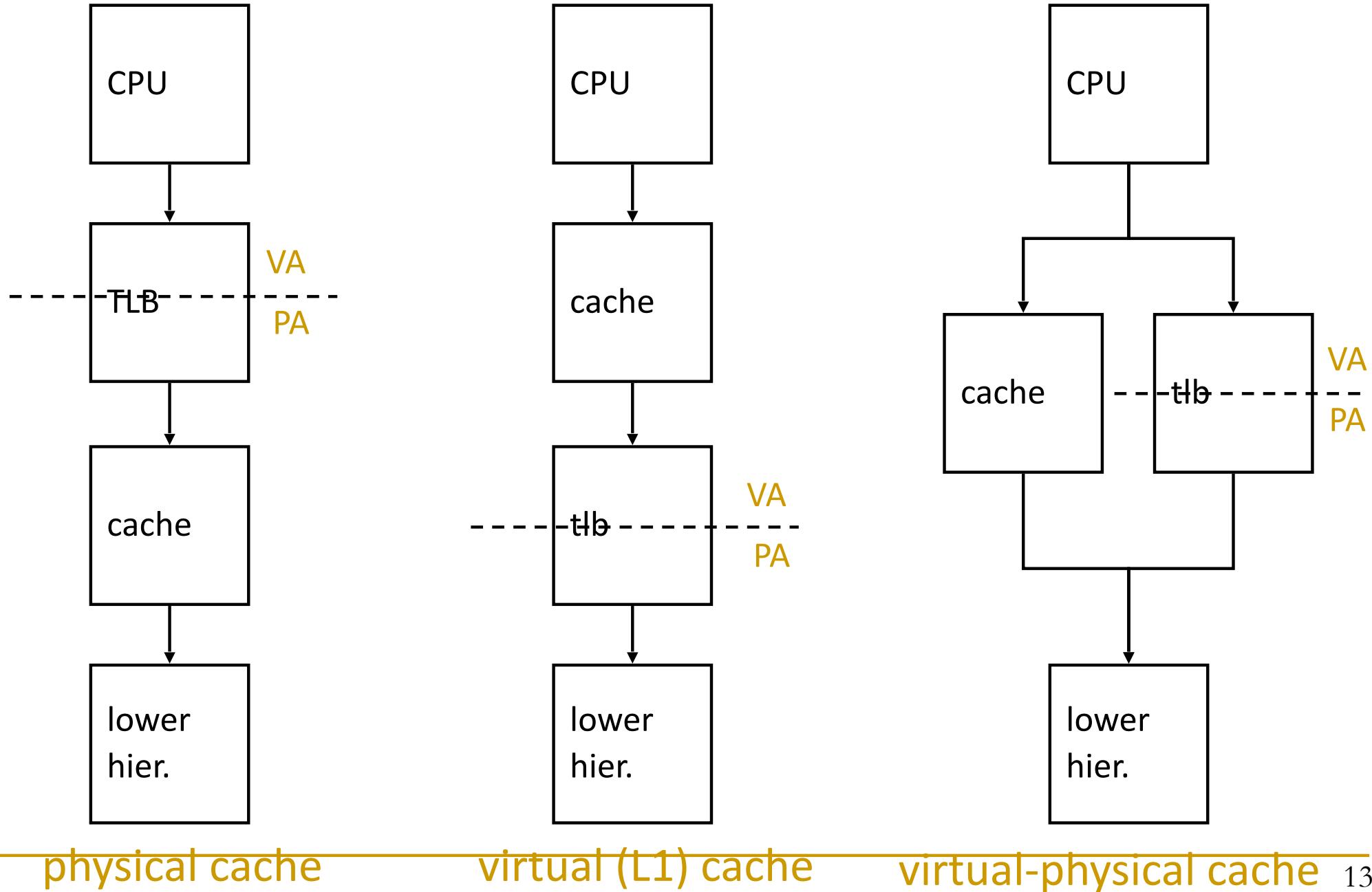
Teaser: Virtual Memory Issue III

- When do we do the address translation?
 - Before or after accessing the L1 cache?

Homonyms and Synonyms

- Homonym: Same VA can map to two different PAs
 - Why?
 - VA is in different processes
- Synonym: Different VAs can map to the same PA
 - Why?
 - Different pages can share the same physical frame within or across processes
 - Reasons: shared libraries, shared data, copy-on-write pages within the same process, ...
- Do homonyms and synonyms create problems when we have a cache?
 - Is the cache virtually or physically addressed?

Cache-VM Interaction



Cache fisica:

accediamo alla cache dopo aver fatto la traduzione degli indirizzi. Prima la cpu emette un indirizzo che viene tradotto dalla mmu (se siamo fortunati dal tlb), e l'indirizzo fisico viene emesso verso la cache. In questo caso si posso avere problemi di omonimi e sinonimi?

No perché in cache ci sono tag e indici che sono relativi a dati di memoria fisica, perché è una copia della dram. Il problema è superato perché la traduzione è stata fatta a monte.

Inoltre se cambio programma in esecuzione non fa niente, non devo invalidare nulla perché ho copie di indirizzi fisici in cache, non virtuali.

Il fatto che però va fatta la traduzione prima di accedere in cache rende il processo più lento.

Cache virtuale:

Metto il tlb dopo la cache .

Questo vuol dire che se fatto hit in cache non pago la traduzione da indirizzo virtuale ad indirizzo fisico. Quindi se la cache avesse gli indirizzi virtuali nel campo tag, quando faccio hit in cache evito di fare la traduzione, questo è buono dal punto di vista della latenza.

C'è però il problema dei sinonimi e degli omonimi, tutte le volte che cambio processo dovrei quindi invalidare la cache. Il problema è che potrei avere un campo tag che contiene bit di indirizzo virtuale, quindi quando cambio processo se non avessi invalidato completamente la cache potrei trovare corrispondenza con l'indirizzo virtuale del processo precedente, e quindi accedere alla memoria fisica del processo precedente, quindi accedo ad sti di qualcuno altro.

Cache che fa contemporaneamente traduzione tra indirizzo virtuale e fisico e accesso in cache:

È la più utilizzata.

Quando accedo alla cache faccio due operazioni accedo a un determinato set con alcuni bit e con altri bit cerco la corrispondenza tra il campo tag dell'indirizzo e il campo tag contenuto in un determinato blocco di memoria.

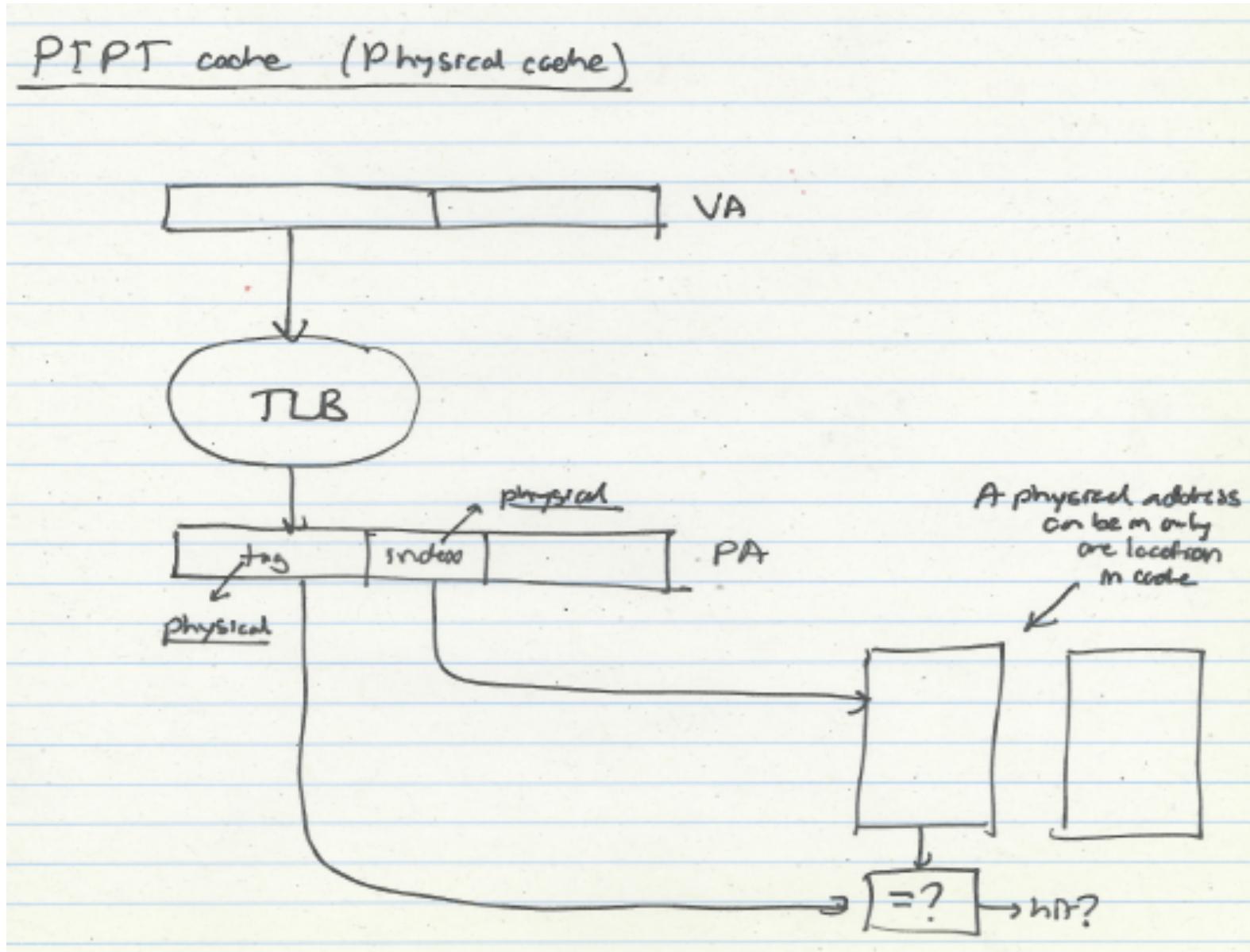
Mentre identifico il set,, quindi mentre precarico i bit dei comparatori che devono comparare il campo tag dei blocchi di memoria con i bit del campo tag del mio indirizzo, quindi identifico un set, ma la comparazione del tag lo faccio con l'indirizzo fisico.

Quindi accedo ai bit di set id con bit virtuali, e ai bit di tag accedo con indirizzo fisico.

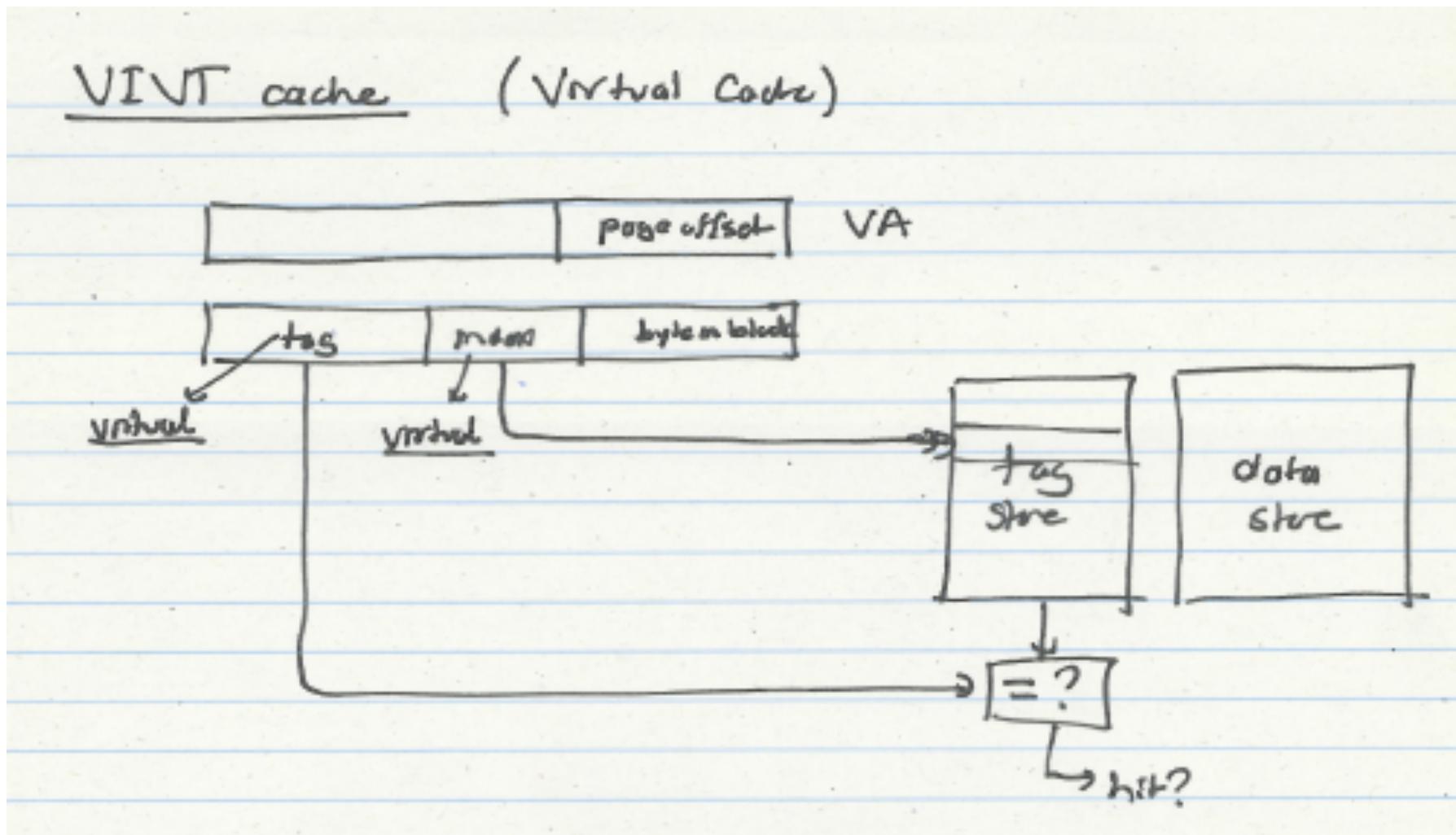
Questa cache è detta virtual index e physically tag. È una cache che funziona bene solo con certe dimensioni di cache.

I campi di indice sono presi dall'indirizzo virtuale ma se sono fortunati cadono nel page offset, quindi nella regione di indirizzi che è uguale tra indirizzi fisici e virtuali. E uso invece i bit di indirizzo fisico per compararmi con il tag contenuto nella cache. Quindi prelevo direttamente i bit dal indirizzo virtuale e li uso per indirizzare il set all'interno della cache. Parallelamente faccio la traduzione da vpn a ppn e con il ppn vado ad accedere al campo tag. Voglio che il tag sia fisico. Così a tutti gli effetti sto indirizzando la cache con un indirizzo fisico ma nascondo la latenza di accesso e. Traduzione degli indirizzi, facendo contemporaneamente l'operazione di accedere al set.

Physical Cache

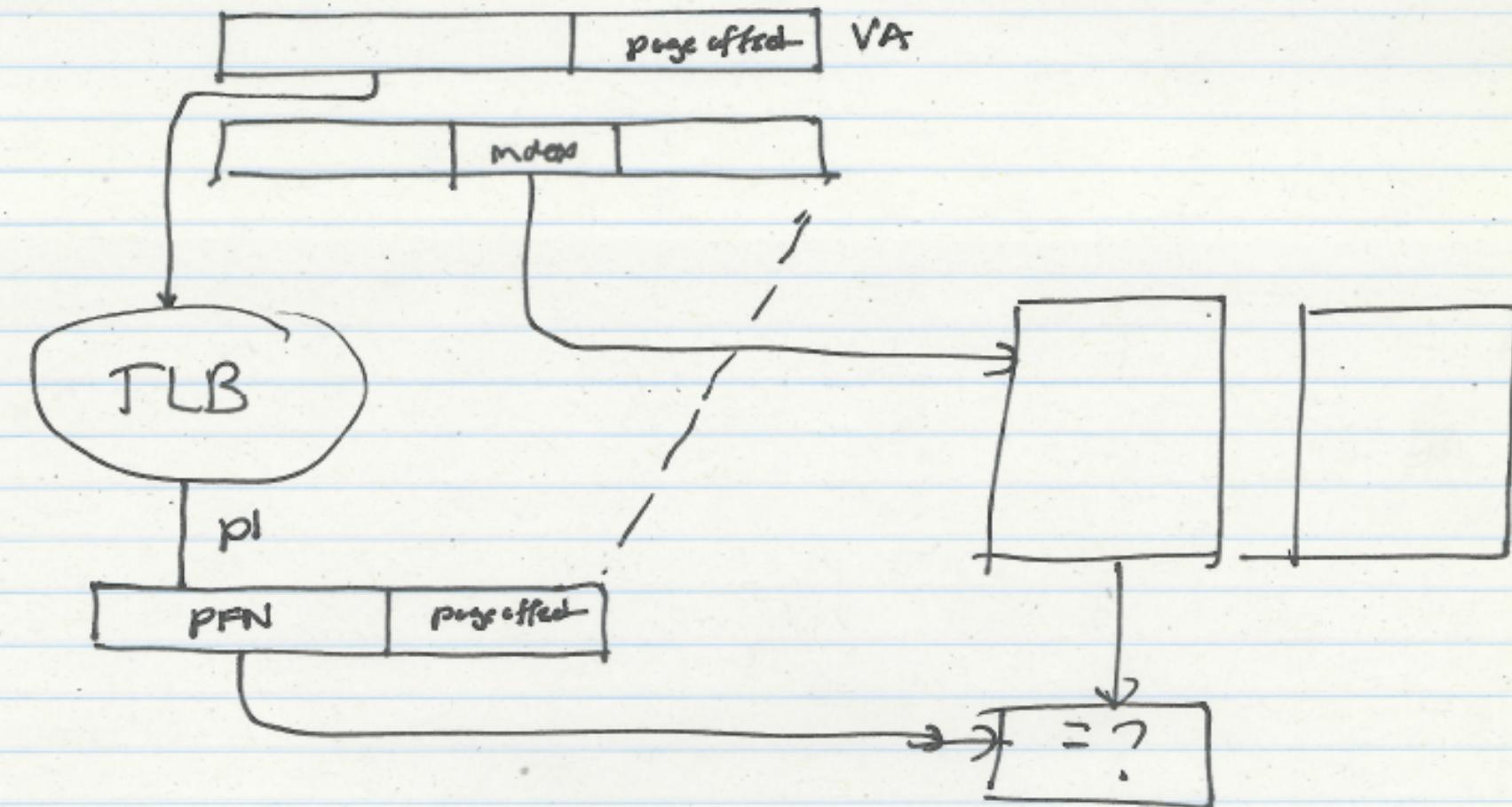


Virtual Cache



Virtual-Physical Cache

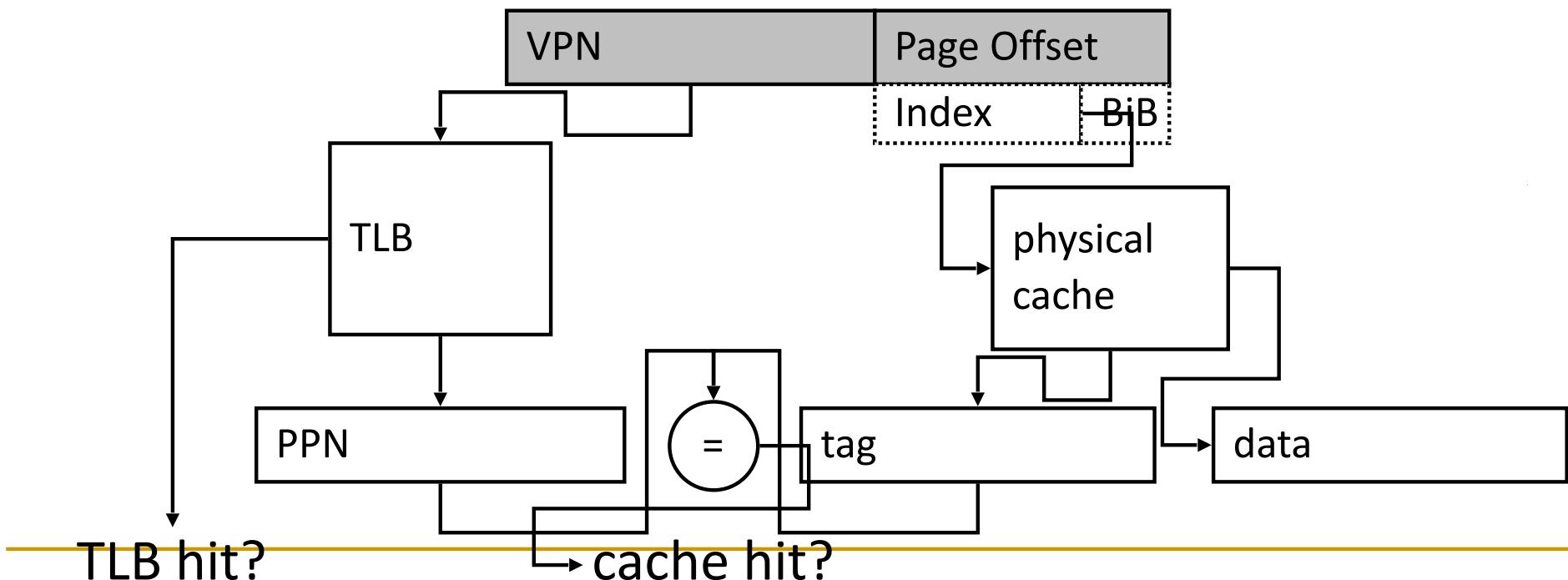
VIPT cache



Where can the same physical address be in the cache?

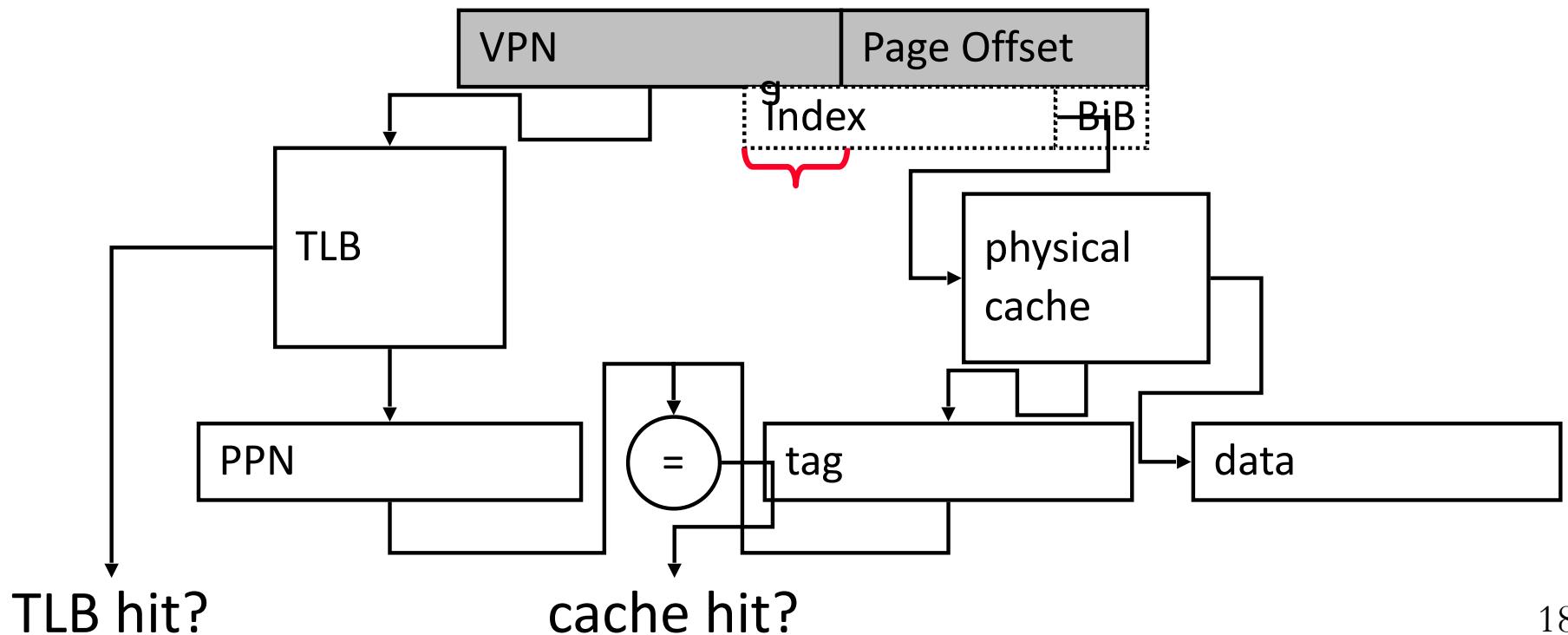
Virtually-Indexed Physically-Tagged

- If $C \leq (\text{page_size} \times \text{associativity})$, the cache index bits come only from page offset (same in VA and PA)
- If both cache and TLB are on chip
 - index both arrays concurrently using VA bits
 - check cache tag (physical) against TLB output at the end



Virtually-Indexed Physically-Tagged

- If $C > (\text{page_size} \times \text{associativity})$, the cache index bits include VPN
 - ⇒ Synonyms can cause problems
 - The same physical address can exist in two locations
- Solutions?



Virtual Memory Summary

Virtual Memory Summary

- Virtual memory gives the illusion of “infinite” capacity
- A subset of virtual pages are located in physical memory
- A page table maps virtual pages to physical pages – this is called address translation
- A TLB speeds up address translation
- Multi-level page tables keep the page table size in check
- Using different page tables for different programs provides memory protection

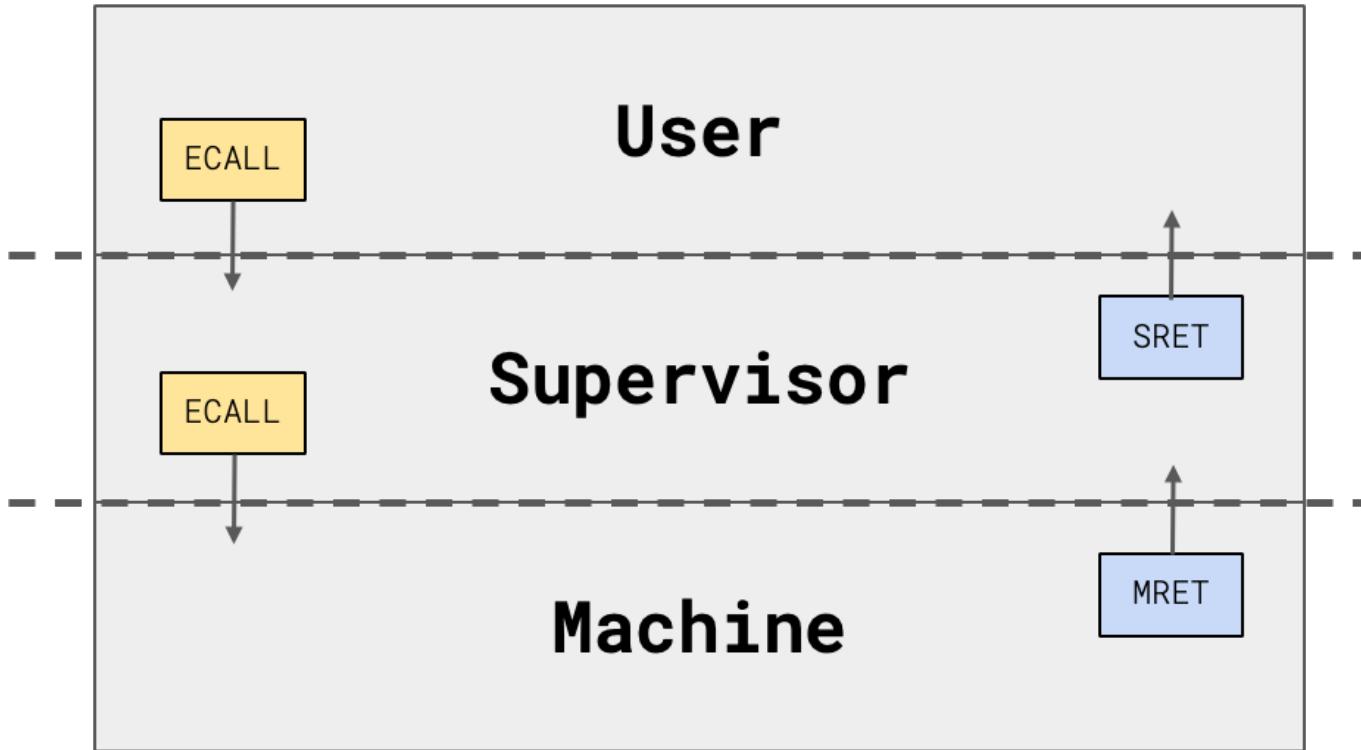
Multilevel On-Chip Caches

Characteristic	ARM Cortex-A53	Intel Core i7
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	Configurable 16 to 64 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	Two-way (I), four-way (D) set associative	Four-way (I), eight-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, variable allocation policies (default is Write-allocate)	Write-back, No-write-allocate
L1 hit time (load-use)	Two clock cycles	Four clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 2 MiB	256 KiB (0.25 MiB)
L2 cache associativity	16-way set associative	8-way set associative
L2 replacement	Approximated LRU	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	12 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

2-Level TLB Organization

Characteristic	ARM Cortex-A53	Intel Core i7
Virtual address	48 bits	48 bits
Physical address	40 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 2 MiB, 1 GiB	Variable: 4 KiB, 2/4 MiB
TLB organization	<p>1 TLB for instructions and 1 TLB for data per core</p> <p>Both micro TLBs are fully associative, with 10 entries, round robin replacement</p> <p>64-entry, four-way set-associative TLBs</p> <p>TLB misses handled in hardware</p>	<p>1 TLB for instructions and 1 TLB for data per core</p> <p>Both L1 TLBs are four-way set associative, LRU replacement</p> <p>L1 I-TLB has 128 entries for small pages, seven per thread for large pages</p> <p>L1 D-TLB has 64 entries for small pages, 32 for large pages</p> <p>The L2 TLB is four-way set associative, LRU replacement</p> <p>The L2 TLB has 512 entries</p> <p>TLB misses handled in hardware</p>

RV Privilege Levels

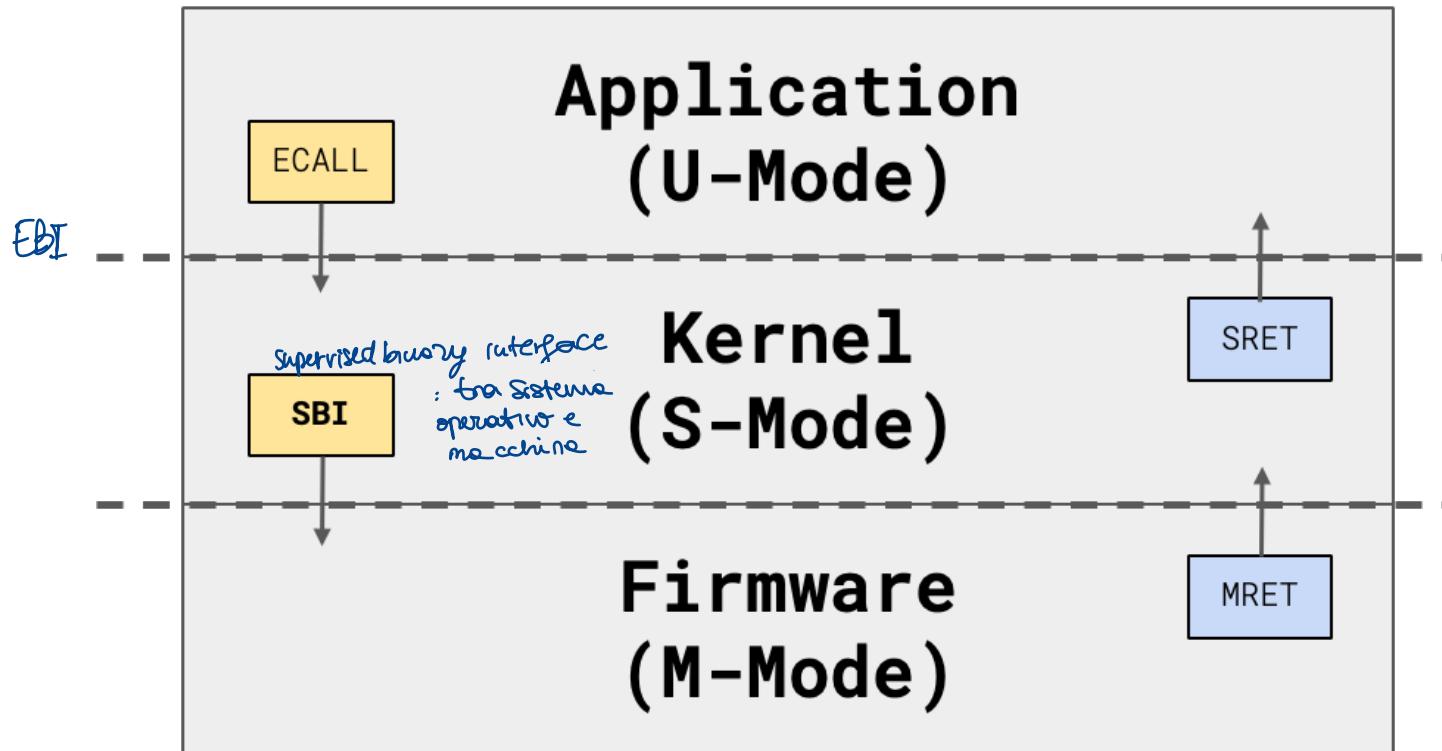


RISC-V Privilege Modes

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	
3	11	Machine	M

Table 1.1: RISC-V privilege levels. tutte le architetture RISC-V devono implementare almeno il livello di privilegio M.

RV Privilege Levels

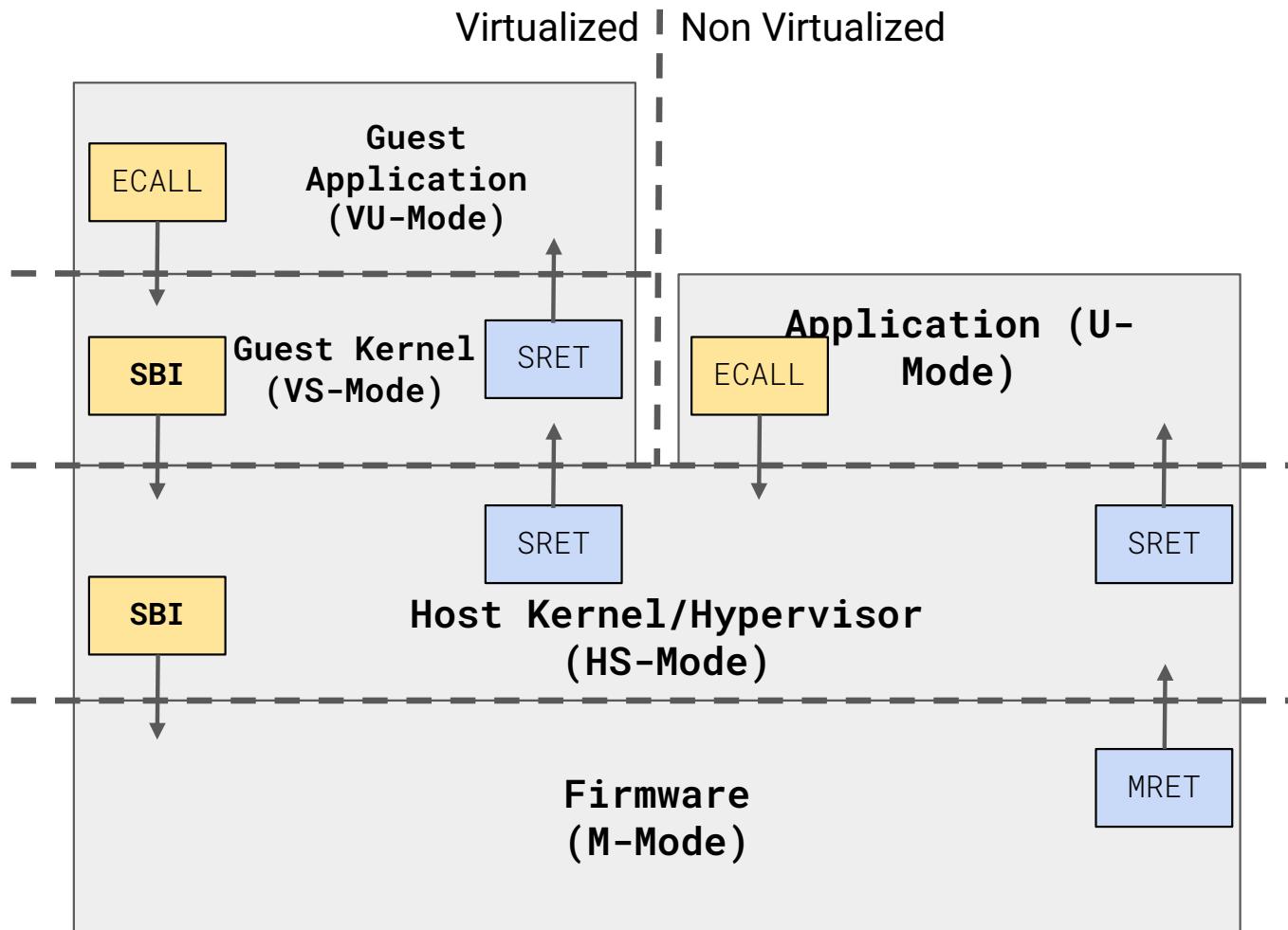


RISC-V Privilege Modes

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

Table 1.2: Supported combinations of privilege modes.

RV Privilege Levels



RISC-V Privilege Modes with Hypervisor Extension

RV Privilege Levels - CSRs

CSR Address			Hex	Use and Accessibility
[11:10]	[9:8]	[7:4]		
Unprivileged and User-Level CSRs				
00	00	XXXX	0x000-0x0FF	Standard read/write
01	00	XXXX	0x400-0x4FF	Standard read/write
10	00	XXXX	0x800-0x8FF	Custom read/write
11	00	OXXX	0xC00-0xC7F	Standard read-only
11	00	10XX	0xC80-0xCB	Standard read-only
11	00	11XX	0xCC0-0xCFF	Custom read-only
Supervisor-Level CSRs				
00	01	XXXX	0x100-0x1FF	Standard read/write
01	01	OXXX	0x500-0x57F	Standard read/write
01	01	10XX	0x580-0x5BF	Standard read/write
01	01	11XX	0x5C0-0x5FF	Custom read/write
10	01	OXXX	0x900-0x97F	Standard read/write
10	01	10XX	0x980-0x9BF	Standard read/write
10	01	11XX	0x9C0-0x9FF	Custom read/write
11	01	OXXX	0xD00-0xD7F	Standard read-only
11	01	10XX	0xD80-0xDBF	Standard read-only
11	01	11XX	0xDC0-0xDFF	Custom read-only
Hypervisor and VS CSRs				
00	10	XXXX	0x200-0x2FF	Standard read/write
01	10	OXXX	0x600-0x67F	Standard read/write
01	10	10XX	0x680-0x6BF	Standard read/write
01	10	11XX	0x6C0-0x6FF	Custom read/write
10	10	OXXX	0xA00-0xA7F	Standard read/write
10	10	10XX	0xA80-0xABF	Standard read/write
10	10	11XX	0xAC0-0xAF	Custom read/write
11	10	OXXX	0xE00-0xE7F	Standard read-only
11	10	10XX	0xE80-0xEBF	Standard read-only
11	10	11XX	0xEC0-0xEFF	Custom read-only
Machine-Level CSRs				
00	11	XXXX	0x300-0x3FF	Standard read/write
01	11	OXXX	0x700-0x77F	Standard read/write
01	11	100X	0x780-0x79F	Standard read/write
01	11	1010	0x7A0-0x7AF	Standard read/write debug CSRs
01	11	1011	0x7B0-0x7BF	Debug-mode-only CSRs
01	11	11XX	0x7C0-0x7FF	Custom read/write
10	11	OXXX	0xB00-0xB7F	Standard read/write
10	11	10XX	0xB80-0xBBF	Standard read/write
10	11	11XX	0xBC0-0xBFF	Custom read/write
11	11	OXXX	0xF00-0xF7F	Standard read-only
11	11	10XX	0xF80-0xFB	Standard read-only
11	11	11XX	0xFC0-0xFFFF	Custom read-only

11	10	11	XXXX	Use and Accessibility
Hypervisor and VS CSRs				
00	10	XXXX	0x200-0x2FF	Standard read/write
01	10	OXXX	0x600-0x67F	Standard read/write
01	10	10XX	0x680-0x6BF	Standard read/write
01	10	11XX	0x6C0-0x6FF	Custom read/write
10	10	OXXX	0xA00-0xA7F	Standard read/write
10	10	10XX	0xA80-0xABF	Standard read/write
10	10	11XX	0xAC0-0xAF	Custom read/write
11	10	OXXX	0xE00-0xE7F	Standard read-only
11	10	10XX	0xE80-0xEBF	Standard read-only
11	10	11XX	0xEC0-0xEFF	Custom read-only
Machine-Level CSRs				
00	11	XXXX	0x300-0x3FF	Standard read/write
01	11	OXXX	0x700-0x77F	Standard read/write
01	11	100X	0x780-0x79F	Standard read/write
01	11	1010	0x7A0-0x7AF	Standard read/write debug CSRs
01	11	1011	0x7B0-0x7BF	Debug-mode-only CSRs
01	11	11XX	0x7C0-0x7FF	Custom read/write
10	11	OXXX	0xB00-0xB7F	Standard read/write
10	11	10XX	0xB80-0xBBF	Standard read/write
10	11	11XX	0xBC0-0xBFF	Custom read/write
11	11	OXXX	0xF00-0xF7F	Standard read-only
11	11	10XX	0xF80-0xFB	Standard read-only
11	11	11XX	0xFC0-0xFFFF	Custom read-only

RV Privilege Levels - CSRs

Number	Privilege	Name	Description
Unprivileged Floating-Point CSRs			
0x001	URW	fflags	Floating-Point Accrued Exceptions.
0x002	URW	frm	Floating-Point Dynamic Rounding Mode.
0x003	URW	fcsr	Floating-Point Control and Status Register (frm + fflags).
Unprivileged Counter/Timers			
0xC00	URO	cycle	Cycle counter for RDCYCLE instruction.
0xC01	URO	time	Timer for RDTIME instruction.
0xC02	URO	instret	Instructions-retired counter for RDINSTRET instruction.
0xC03	URO	hpmcounter3	Performance-monitoring counter.
0xC04	URO	hpmcounter4	Performance-monitoring counter.
		:	
0xC1F	URO	hpmcounter31	Performance-monitoring counter.
0xC80	URO	cycleh	Upper 32 bits of cycle , RV32 only.
0xC81	URO	timeh	Upper 32 bits of time , RV32 only.
0xC82	URO	instreth	Upper 32 bits of instret , RV32 only.
0xC83	URO	hpmcounter3h	Upper 32 bits of hpmcounter3 , RV32 only.
0xC84	URO	hpmcounter4h	Upper 32 bits of hpmcounter4 , RV32 only.
		:	
0xC9F	URO	hpmcounter31h	Upper 32 bits of hpmcounter31 , RV32 only.

c'è un registro che conta il n° di cicli passati dall'inizio dell'esecuzione ad esempio.

RV Privilege Levels - CSRs

Number	Privilege	Name	Description
Supervisor Trap Setup			
0x100	SRW	sstatus	Supervisor status register.
0x104	SRW	sie	Supervisor interrupt-enable register.
0x105	SRW	stvec	Supervisor trap handler base address.
0x106	SRW	scounteren	Supervisor counter enable.
Supervisor Configuration			
0x10A	SRW	senvcfg	Supervisor environment configuration register.
Supervisor Trap Handling			
0x140	SRW	sscratch	Scratch register for supervisor trap handlers.
0x141	SRW	sepc	Supervisor exception program counter.
0x142	SRW	scause	Supervisor trap cause.
0x143	SRW	stval	Supervisor bad address or instruction.
0x144	SRW	sip	Supervisor interrupt pending.
Supervisor Protection and Translation			
0x180	SRW	satp	Supervisor address translation and protection.
Debug/Trace Registers			
0x5A8	SRW	scontext	Supervisor-mode context register.

RV Privilege Levels - CSRs

Number	Privilege	Name	Description
Machine Information Registers			
0xF11	MRO	<code>mvendorid</code>	Vendor ID.
0xF12	MRO	<code>marchid</code>	Architecture ID.
0xF13	MRO	<code>mimpid</code>	Implementation ID.
0xF14	MRO	<code>mhartid</code>	Hardware thread ID.
0xF15	MRO	<code>mconfigptr</code>	Pointer to configuration data structure.
Machine Trap Setup			
0x300	MRW	<code>mstatus</code>	Machine status register.
0x301	MRW	<code>misa</code>	ISA and extensions
0x302	MRW	<code>medeleg</code>	Machine exception delegation register.
0x303	MRW	<code>mdeleg</code>	Machine interrupt delegation register.
0x304	MRW	<code>mie</code>	Machine interrupt-enable register.
0x305	MRW	<code>mtvec</code>	Machine trap-handler base address.
0x306	MRW	<code>mcounteren</code>	Machine counter enable.
0x310	MRW	<code>mstatush</code>	Additional machine status register, RV32 only.
Machine Trap Handling			
0x340	MRW	<code>mscratch</code>	Scratch register for machine trap handlers.
0x341	MRW	<code>mepc</code>	Machine exception program counter.
0x342	MRW	<code>mcause</code>	Machine trap cause.
0x343	MRW	<code>mtval</code>	Machine bad address or instruction.
0x344	MRW	<code>mip</code>	Machine interrupt pending.
0x34A	MRW	<code>mtinst</code>	Machine trap instruction (transformed).
0x34B	MRW	<code>mtval12</code>	Machine bad guest physical address.
Machine Configuration			
0x30A	MRW	<code>menvcfg</code>	Machine environment configuration register.
0x31A	MRW	<code>menvcfgh</code>	Additional machine env. conf. register, RV32 only.
0x747	MRW	<code>mseccfg</code>	Machine security configuration register.
0x757	MRW	<code>mseccfgh</code>	Additional machine security conf. register, RV32 only.
Machine Memory Protection			
0x3A0	MRW	<code>pmpcfg0</code>	Physical memory protection configuration.
0x3A1	MRW	<code>pmpcfg1</code>	Physical memory protection configuration, RV32 only.
0x3A2	MRW	<code>pmpcfg2</code>	Physical memory protection configuration.
0x3A3	MRW	<code>pmpcfg3</code>	Physical memory protection configuration, RV32 only.
		:	
0x3AE	MRW	<code>pmpcfg14</code>	Physical memory protection configuration.
0x3AF	MRW	<code>pmpcfg15</code>	Physical memory protection configuration, RV32 only.
0x3B0	MRW	<code>pmpaddr0</code>	Physical memory protection address register.
0x3B1	MRW	<code>pmpaddr1</code>	Physical memory protection address register.
		:	
0x3EF	MRW	<code>pmpaddr63</code>	Physical memory protection address register.

Number	Privilege	Name	Description
Machine Non-Maskable Interrupt Handling			
0x740	MRW	<code>mnscratch</code>	Resumable NMI scratch register.
0x741	MRW	<code>mnepc</code>	Resumable NMI program counter.
0x742	MRW	<code>mncause</code>	Resumable NMI cause.
0x744	MRW	<code>mnstatus</code>	Resumable NMI status.
Machine Counter/Timers			
0xB00	MRW	<code>mcycle</code>	Machine cycle counter.
0xB02	MRW	<code>minstret</code>	Machine instructions-retired counter.
0xB03	MRW	<code>mhpmcOUNTER3</code>	Machine performance-monitoring counter.
0xB04	MRW	<code>mhpmcOUNTER4</code>	Machine performance-monitoring counter.
		:	
0xB1F	MRW	<code>mhpmcOUNTER31</code>	Machine performance-monitoring counter.
0xB80	MRW	<code>mcycleh</code>	Upper 32 bits of <code>mcycle</code> , RV32 only.
0xB82	MRW	<code>minstreh</code>	Upper 32 bits of <code>minstret</code> , RV32 only.
0xB83	MRW	<code>mhpmcOUNTER3h</code>	Upper 32 bits of <code>mhpmcOUNTER3</code> , RV32 only.
0xB84	MRW	<code>mhpmcOUNTER4h</code>	Upper 32 bits of <code>mhpmcOUNTER4</code> , RV32 only.
		:	
0xB9F	MRW	<code>mhpmcOUNTER31h</code>	Upper 32 bits of <code>mhpmcOUNTER31</code> , RV32 only.
Machine Counter Setup			
0x320	MRW	<code>mcountinhibit</code>	Machine counter-inhibit register.
0x323	MRW	<code>mhpmevent3</code>	Machine performance-monitoring event selector.
0x324	MRW	<code>mhpmevent4</code>	Machine performance-monitoring event selector.
		:	
0x33F	MRW	<code>mhpmevent31</code>	Machine performance-monitoring event selector.
Debug/Trace Registers (shared with Debug Mode)			
0x7A0	MRW	<code>tselect</code>	Debug/Trace trigger register select.
0x7A1	MRW	<code>tdata1</code>	First Debug/Trace trigger data register.
0x7A2	MRW	<code>tdata2</code>	Second Debug/Trace trigger data register.
0x7A3	MRW	<code>tdata3</code>	Third Debug/Trace trigger data register.
0x7A8	MRW	<code>mcontext</code>	Machine-mode context register.
Debug Mode Registers			
0x7B0	DRW	<code>dcsr</code>	Debug control and status register.
0x7B1	DRW	<code>dpc</code>	Debug program counter.
0x7B2	DRW	<code>dscratch0</code>	Debug scratch register 0.
0x7B3	DRW	<code>dscratch1</code>	Debug scratch register 1.

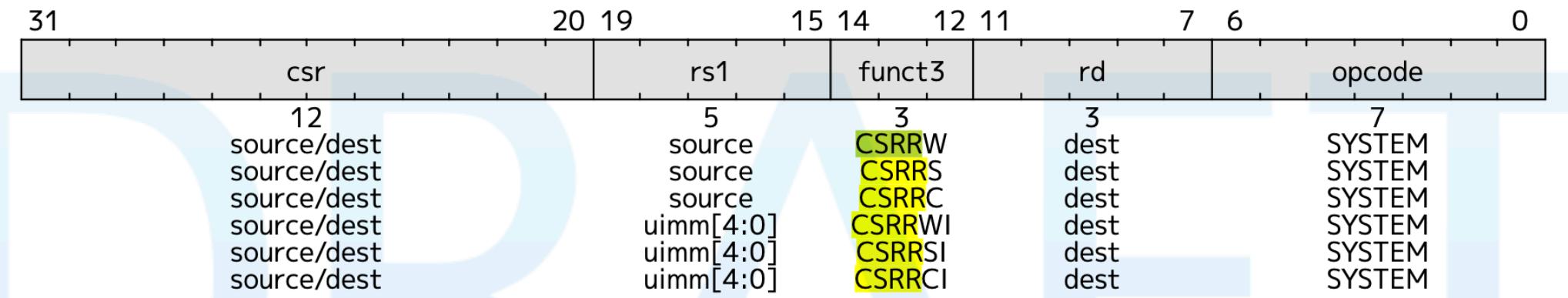
Table 2.6: Currently allocated RISC-V machine-level CSR addresses.

RV Privilege Levels - CSRs

11.1. CSR Instructions

Istruzioni dedicate che possono leggere/modificare specifici bit di un registro che ha un significato particolare

All CSR instructions atomically read-modify-write a single CSR, whose CSR specifier is encoded in the 12-bit *csr* field of the instruction held in bits 31-20. The immediate forms use a 5-bit zero-extended immediate encoded in the *rs1* field.



CSRRW (Atomic Read/Write CSR)

CSRRS (Atomic Read and Set Bits in CSR)

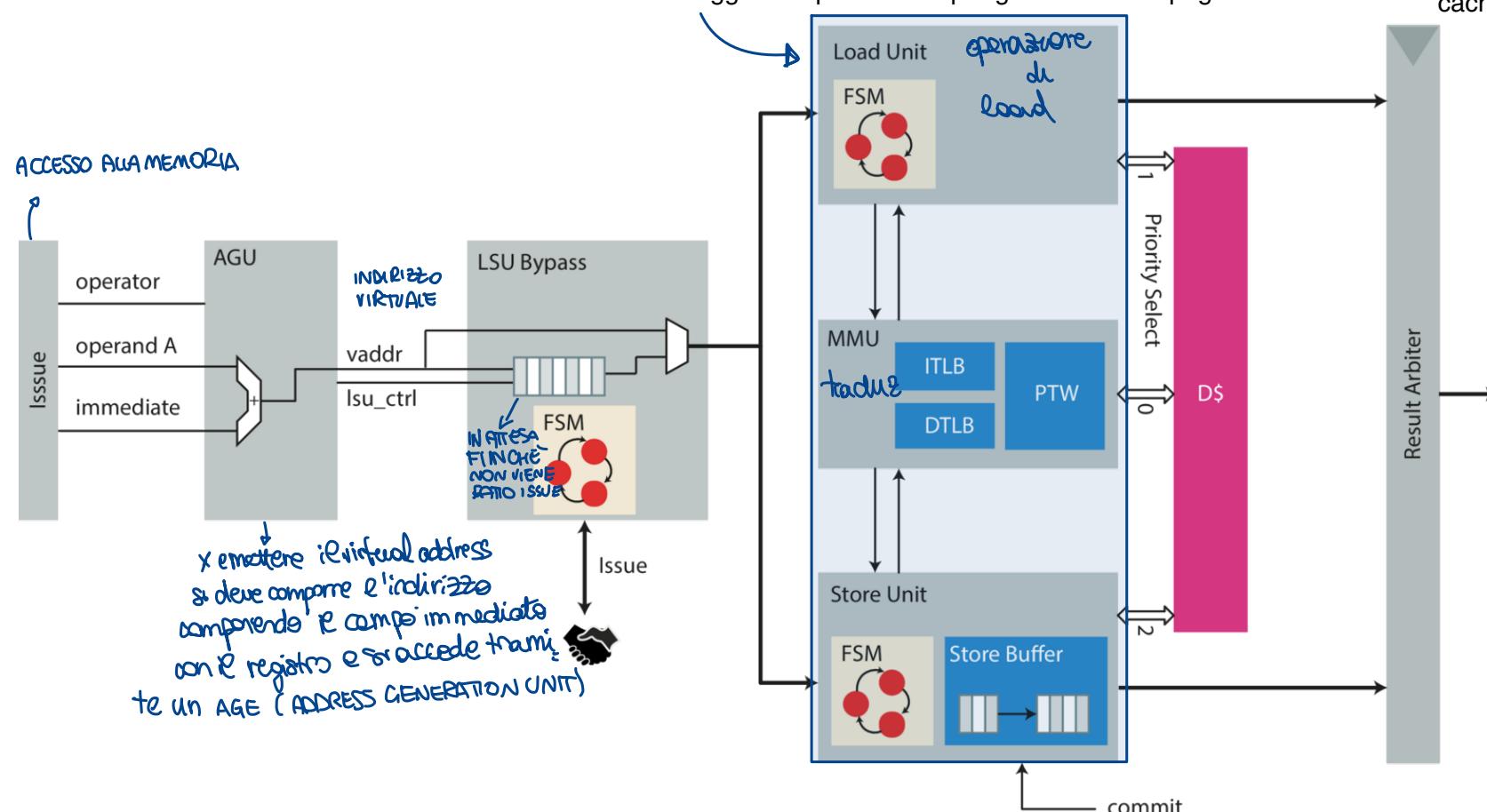
CSRRC (Atomic Read and Clear Bits in CSR)

CVA6 - Load and Store Unit

In un determinato ciclo, potrei avere 3 accessi concorrenti verso la cache.
In un caso potrei avere la load, in un altro la store oppure una traduzione di indirizzo, quindi accesso alla page table.

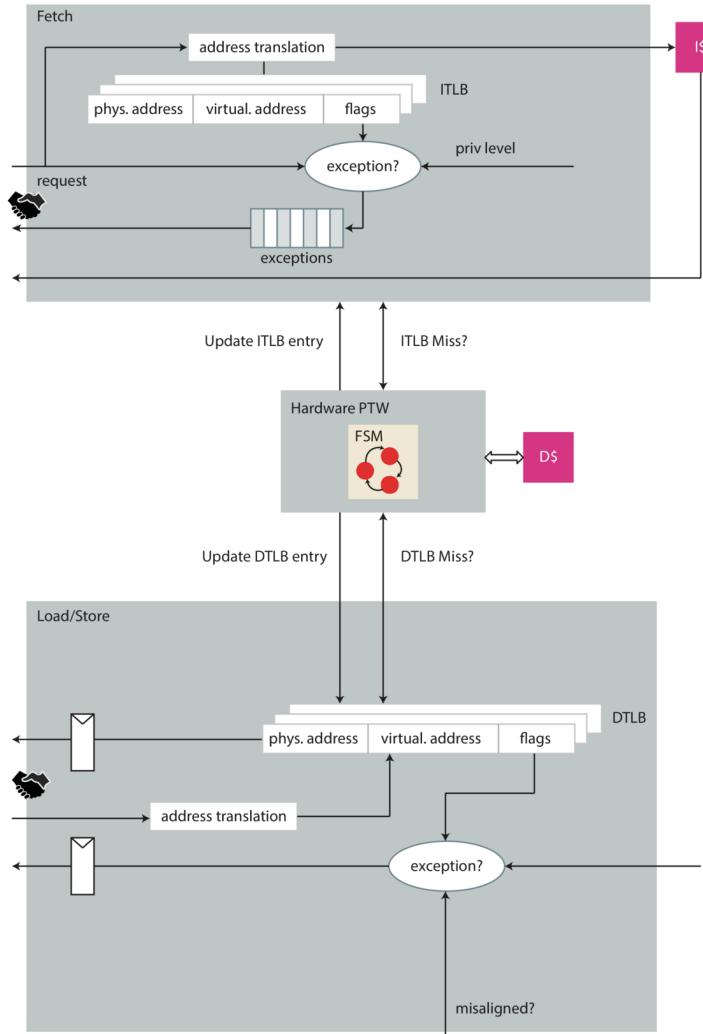
All interno della mmu abbiamo 2tlb, uno per le istruzioni e uno per il dati e anche il page table walker, che ci permette di navigare nella page table.
C'è anche l'arbitraggio che premia sempre gli accessi alla page table.

La load unit emette verso la mmu un indirizzo virtuale e lei le restituirà l'indirizzo fisico con cui poi la load unit andrà a fare accesso verso la data cache.



https://cva6.readthedocs.io/en/latest/03_cva6_design/ex_stage.html

CVA6 - MMU



All' interno di una MMU'

Ci sono due logiche, una parte alta ed una bassa che fanno adddeess translation.
Si ha una logica che fa la traduzione dell'indirizzo per la fetch unit è una parte che fa traduzione per la load/store unit (entrambi questi stadi possono richiedere la traduzione degli indirizzi).

Quando la fetch unit esegue una richiesta di accesso, quindi emette un indirizzo, questo indirizzo passa attraverso L address translation che fa richiesta direttamente al tlb della memoria istruzioni per sapere se c'è già una traduzione tra virt e fisico.

All interno del tlb ci sono anche i flag che codificano indirizzi di accesso su una determinata pagina. La dove i flag non siano verificati viene triggerata un eccezione che riferisce al livello di privilegio. Le eccezioni vengono gestite con Logica FIFO e vengono inviate verso una logica che gestisce le eccezioni.

La logica interposta tra la traduzione degli indirizzi (quindi dal momento in cui si inserisce il programm counter) e l'accesso verso l'istruzione cache è totalmente combinatorio (non ci sono registri), quindi si riesce a fare in un singolo ciclo di ck.

Diversamente per la logica di traduzione (sotto) legata alla load/store unit.

Nel caso di L/s unit, dato che la cache è physically index e virtually tag, la traduzione degli indirizzi passa attraverso uno stadio di registri, si paga un costo aggiuntivo verso la traduzione degli indirizzi. Mentre sopra la logica era più semplice e si riusciva inserire registri tra la traduzione dell'indirizzo e lo stadio di fetch; nel caso della load/store unit è necessario inserirlo perché il path è molto più critico. Esattamente come prima la traduzione avviene accedendo al tlb e poi l'indirizzo che viene emesso viene passato verso l'accesso alla cache e quindi passa attraverso un altro registro di uscita.

Entrambe le due logiche di traduzione degli indirizzi hanno in comune la logica di page table walking: Sia la parte che gestisce la traduzione degli indirizzi per lo stadio di fetch e per lo stadio di memory.

In un ciclo può essere servita una Miss nel tlb dati è una nel tlb istruzioni.

La mmu è condivisa tra lo stadio di fetch e di memory e si fanno istruzioni alla memoria sia con le load/ store sia con accesso al pc, si hanno due tlb disgiunti uno per i dati e uno per le istruzioni.

Se si fa hit in tlb si trova già la traduzione e si può accedere contemporaneamente in cache dati e in cache istruzioni e non avere stalli di tipo strutturale. Avendo una singola mmu è una singola logica di page table walking che è condivisa dalla parte di traduzione degli indirizzi relativa alle istruzioni e quella relativa ai dati, in quel caso se si ha nello stesso ciclo Miss nel tlb istruzioni e Miss nel tlb dati, la ricerca della pagina di mem fisica per l'istruzione sarà serializzata rispetto alla ricerca per la pagina di memoria fisica associata alla memoria virtuale emessa nell'indirizzo verso la main memory. Questo perché in questo caso si ha una sola logica di page table walking.

Siccome la page table di trova nella memoria dati, le Miss di entrambi i tlb devono accedere alla stessa page table, quindi alla stessa regione di memoria e gli accessi devono essere necessariamente serializzati.

https://cva6.readthedocs.io/en/latest/03_cva6_design/ex_stage.html