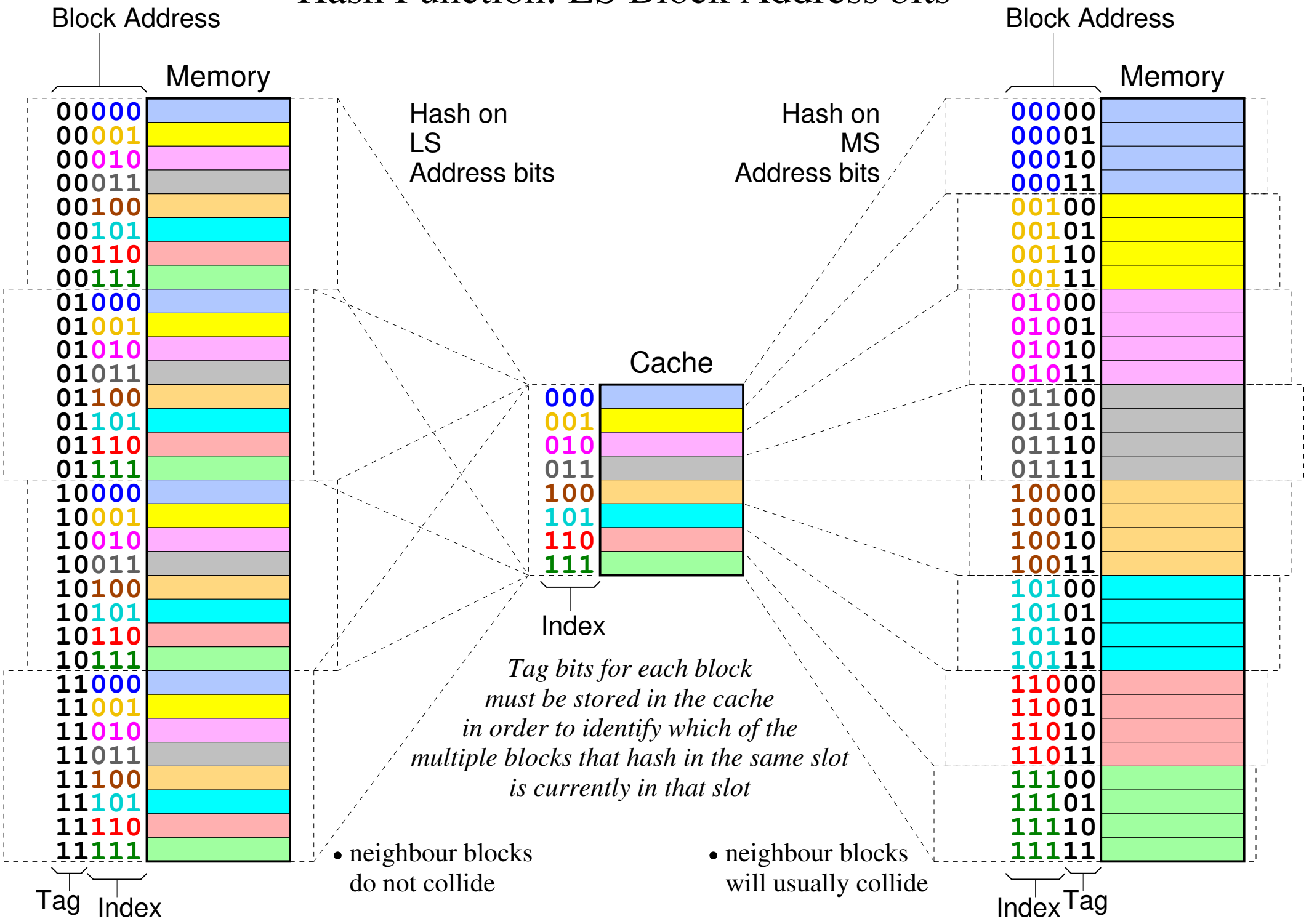
Chapter 5

Large and Fast: Exploiting Memory Hierarchy

Principle of Locality

- Programs access a small proportion of their address space at any time
- Temporal locality — Χρονική Τοπικότητα
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- Spatial locality — Χωρική Τοπικότητα
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

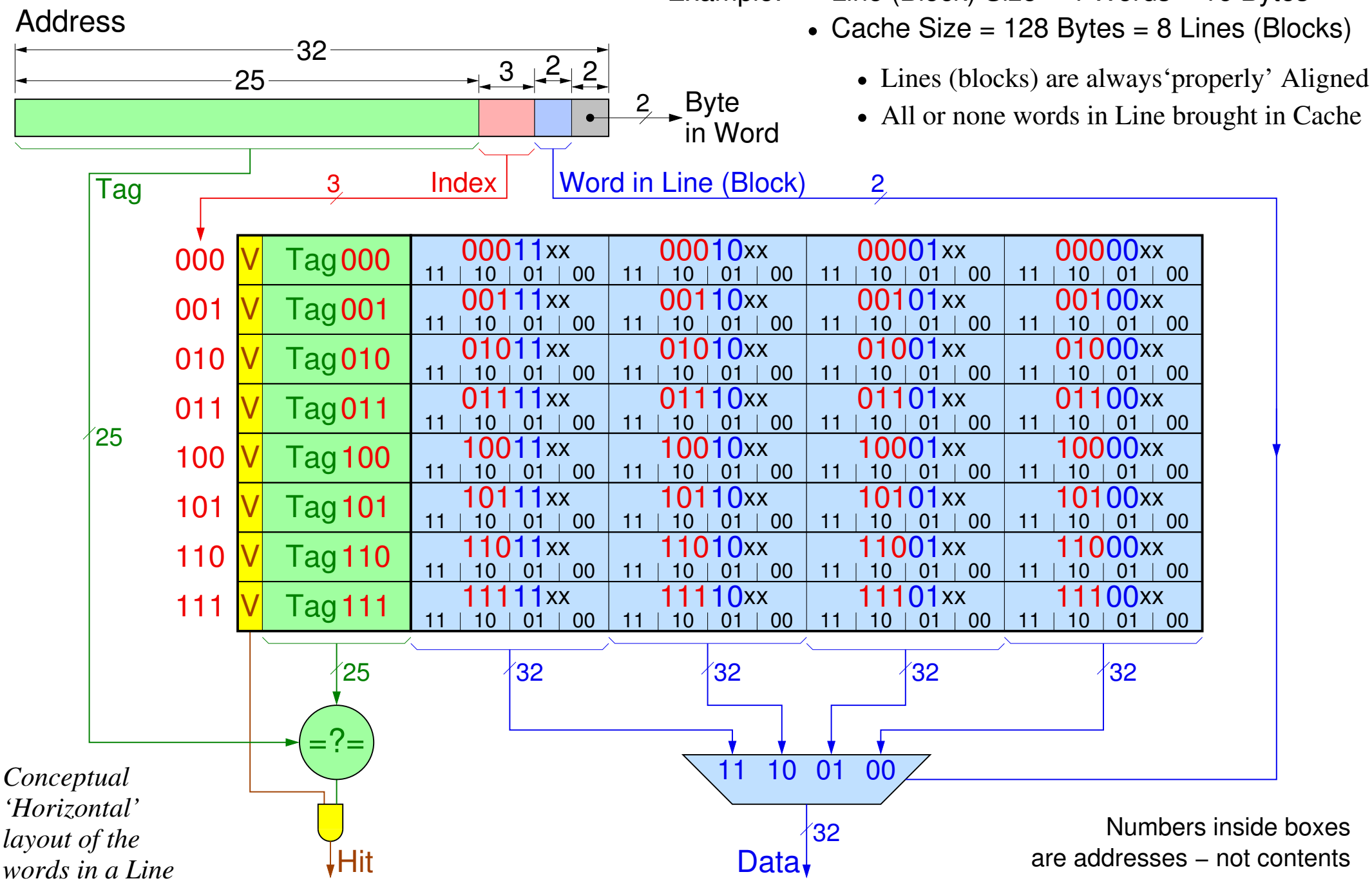
Hash Function: LS Block Address bits



Increased Line (Block) Size, to exploit Spatial Locality

Example:
- Line (Block) Size = 4 Words = 16 Bytes
- Cache Size = 128 Bytes = 8 Lines (Blocks)

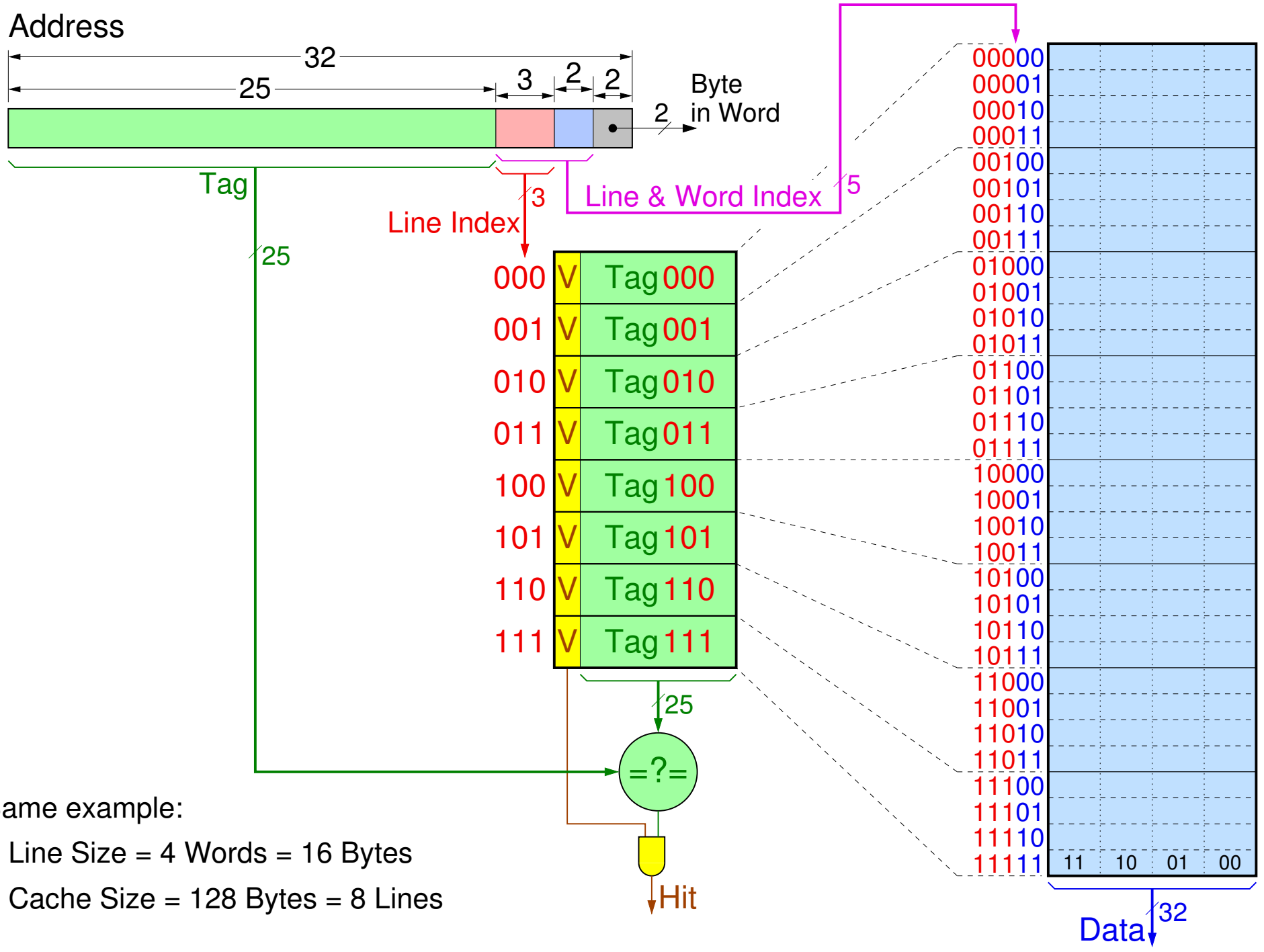
- Lines (blocks) are always 'properly' Aligned
- All or none words in Line brought in Cache



Conceptual 'Horizontal' layout of the words in a Line

Numbers inside boxes are addresses – not contents

‘Vertical’ Layout of the Words in a Line(Block)



Same example:

- Line Size = 4 Words = 16 Bytes
- Cache Size = 128 Bytes = 8 Lines

Write-Through Ταυτόχρονη Εγγραφή

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

Write-Combining: sequential accesses to DRAM take shorter for subsequent words beyond the first one

Write-Back Ετερόχρονη Εγγραφή

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first

Main Memory is inconsistent with Cache

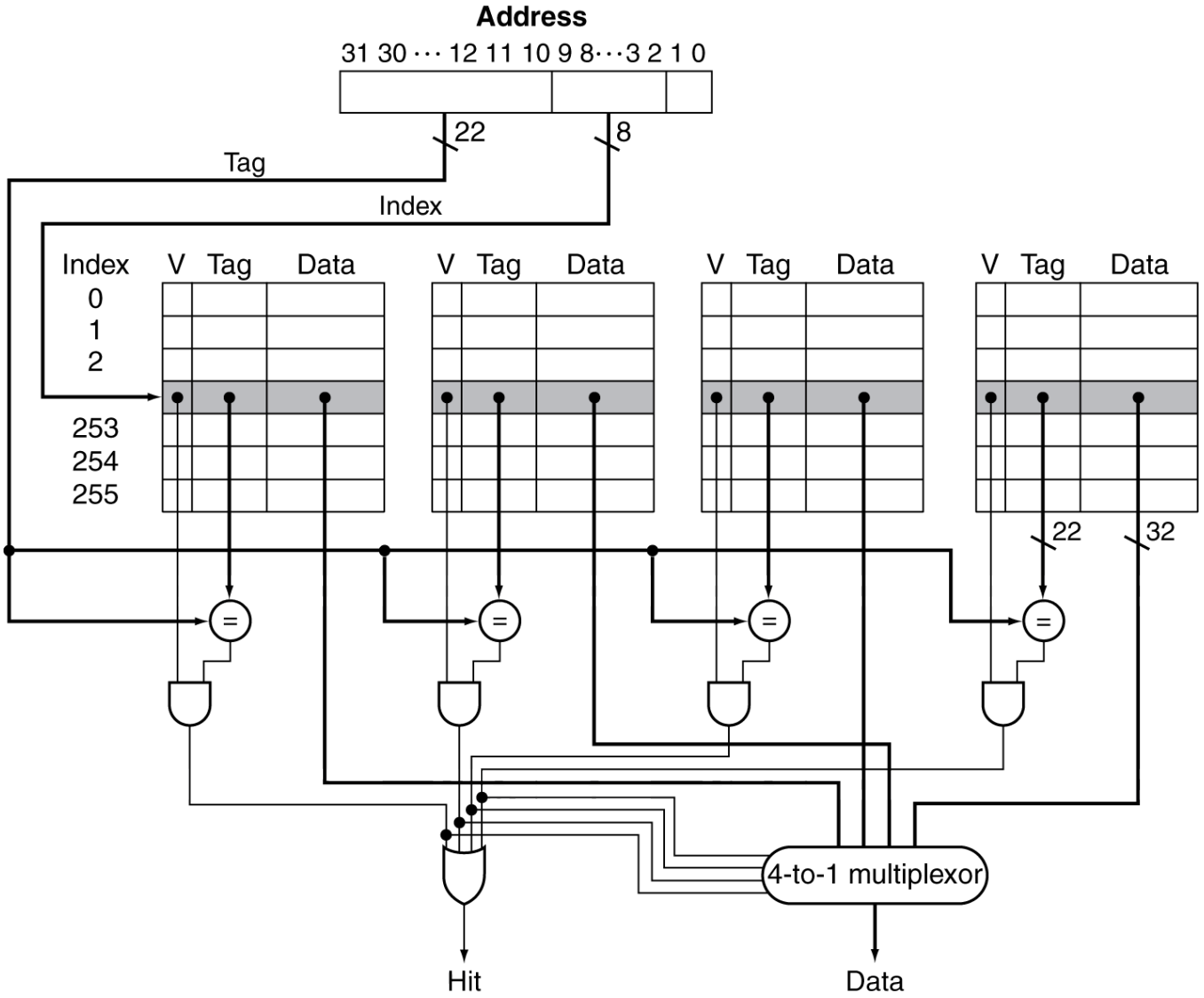
We will revisit this when talking about I/O, then about multicores…

Associative Caches

Μερικώς Προσεταιριστικές Κρυφές Μνήμες

- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- *n*-way set associative
  - Each set contains *n* entries
  - Index portion of address
  - ~~Block number~~ (address) determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

Set Associative Cache Organization



Replacement Policy

Πώς να προβλέψουμε το μέλλον;;
Συχνά, το πρόσφατο παρελθόν αποτελεί καλή ένδειξη γιά το προσεχές μέλλον!...

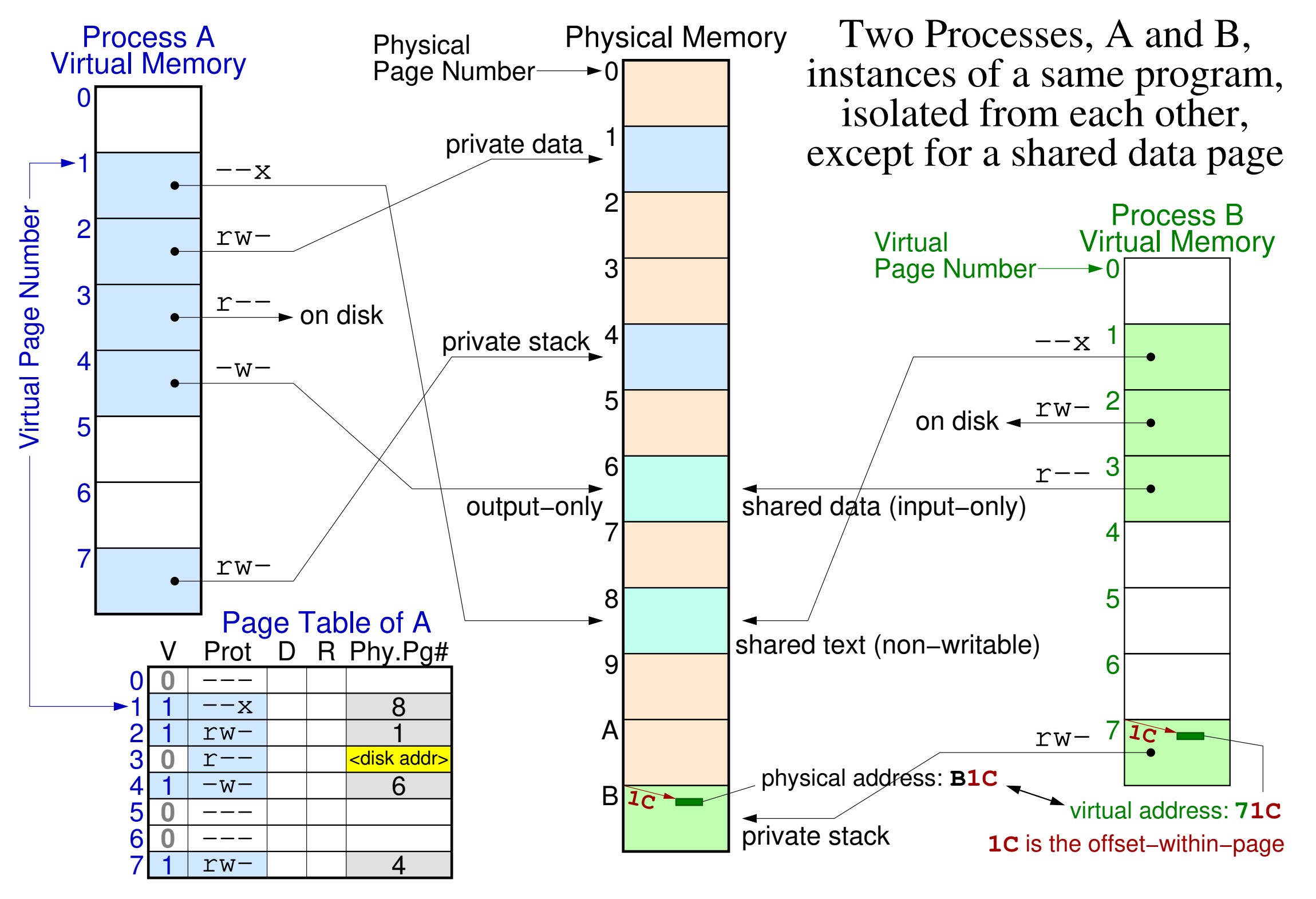
- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately the same performance as LRU for high associativity

Virtual Memory

- Use main memory as a “cache” for secondary (disk) storage
  - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
- CPU and OS translate virtual addresses to physical addresses
  - VM “block” is called a page
  - VM translation “miss” is called a page fault

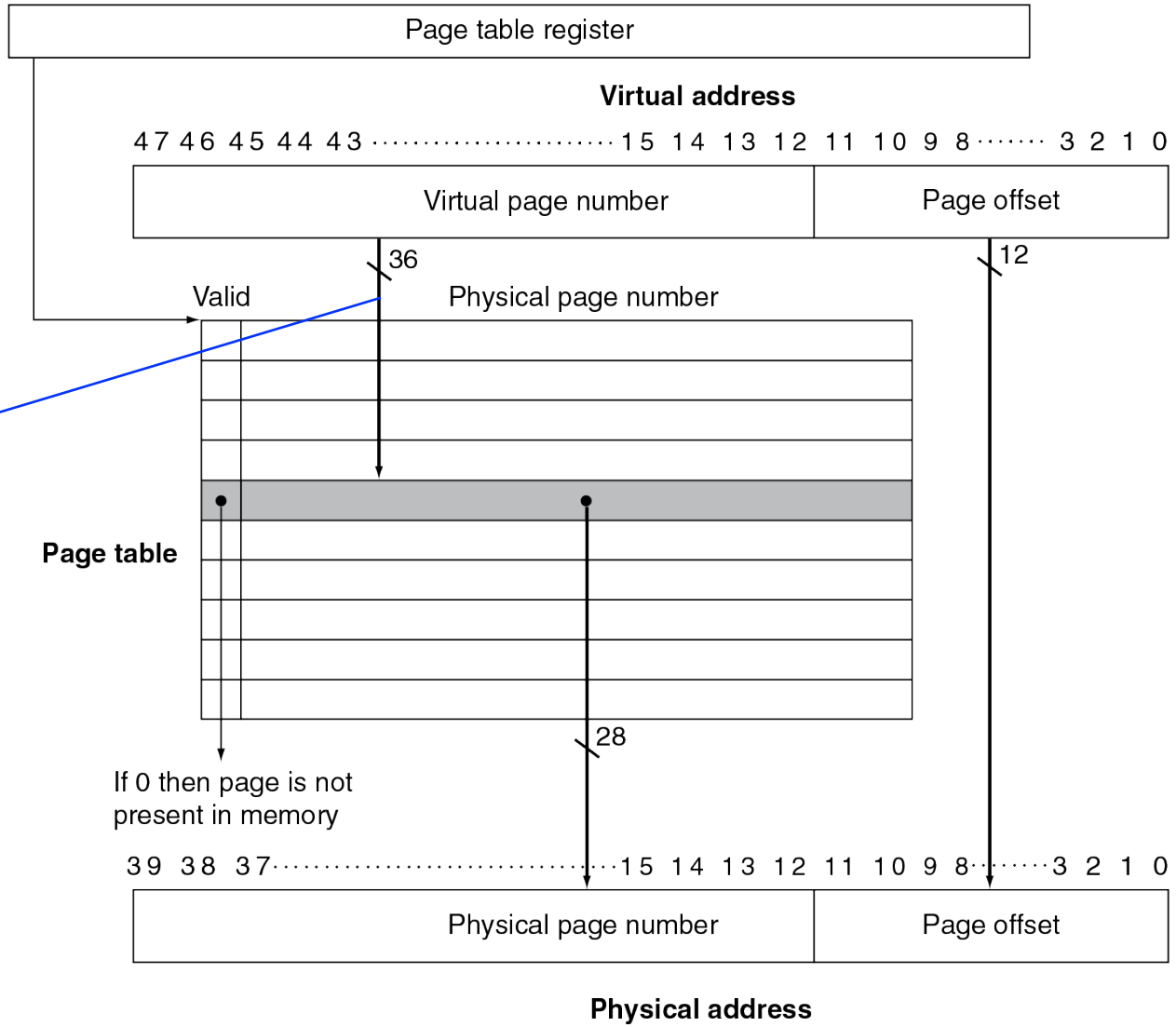
Virtualization & Protection

Also solve the Fragmentation problem: available mem. for new process is fragmented



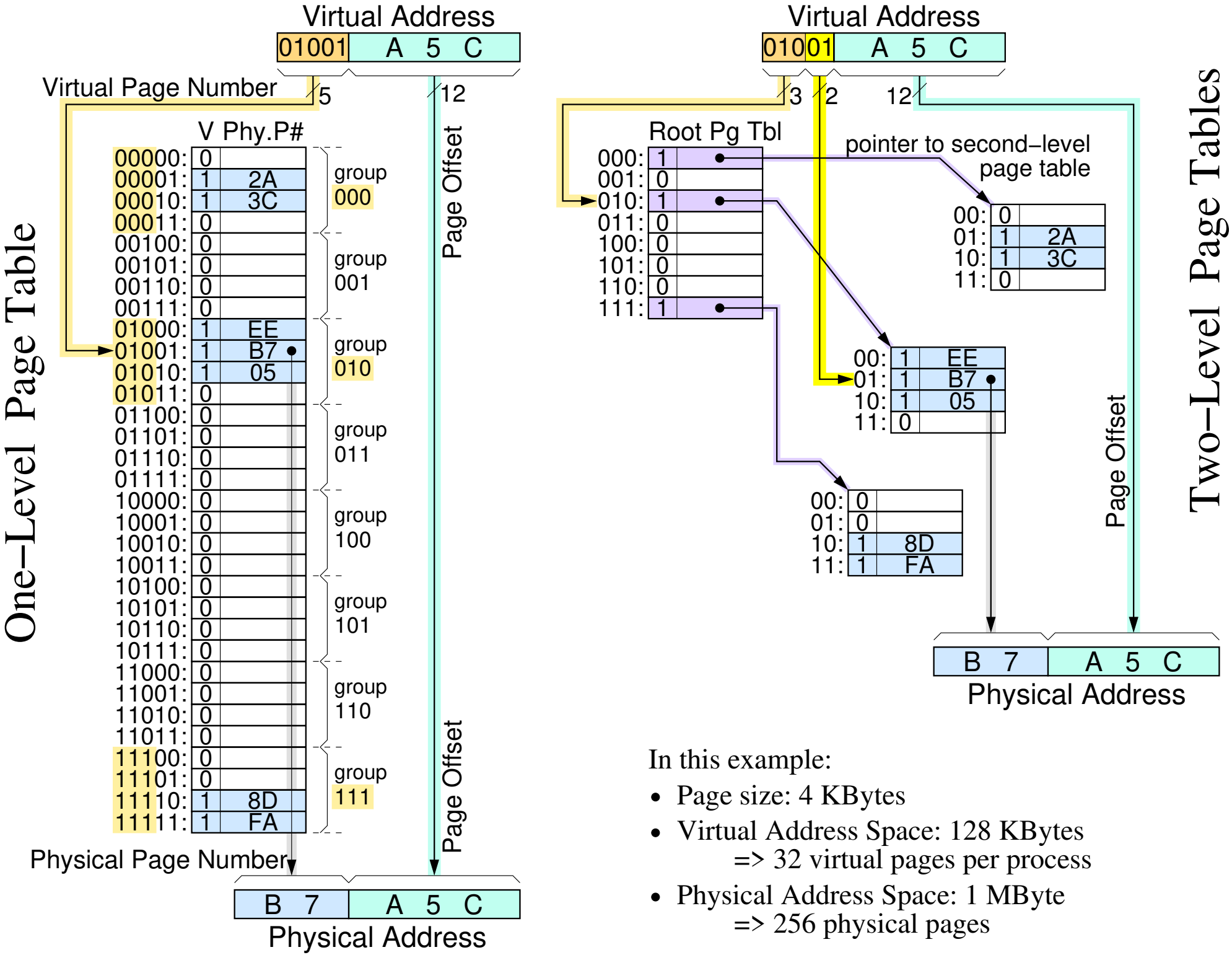
Two Processes, A and B, instances of a same program, isolated from each other, except for a shared data page

Translation Using a Page Table



index within Page Table

Problem: Very Large Size of single-level Page Table; Solution: Multi-Level Page Tables.



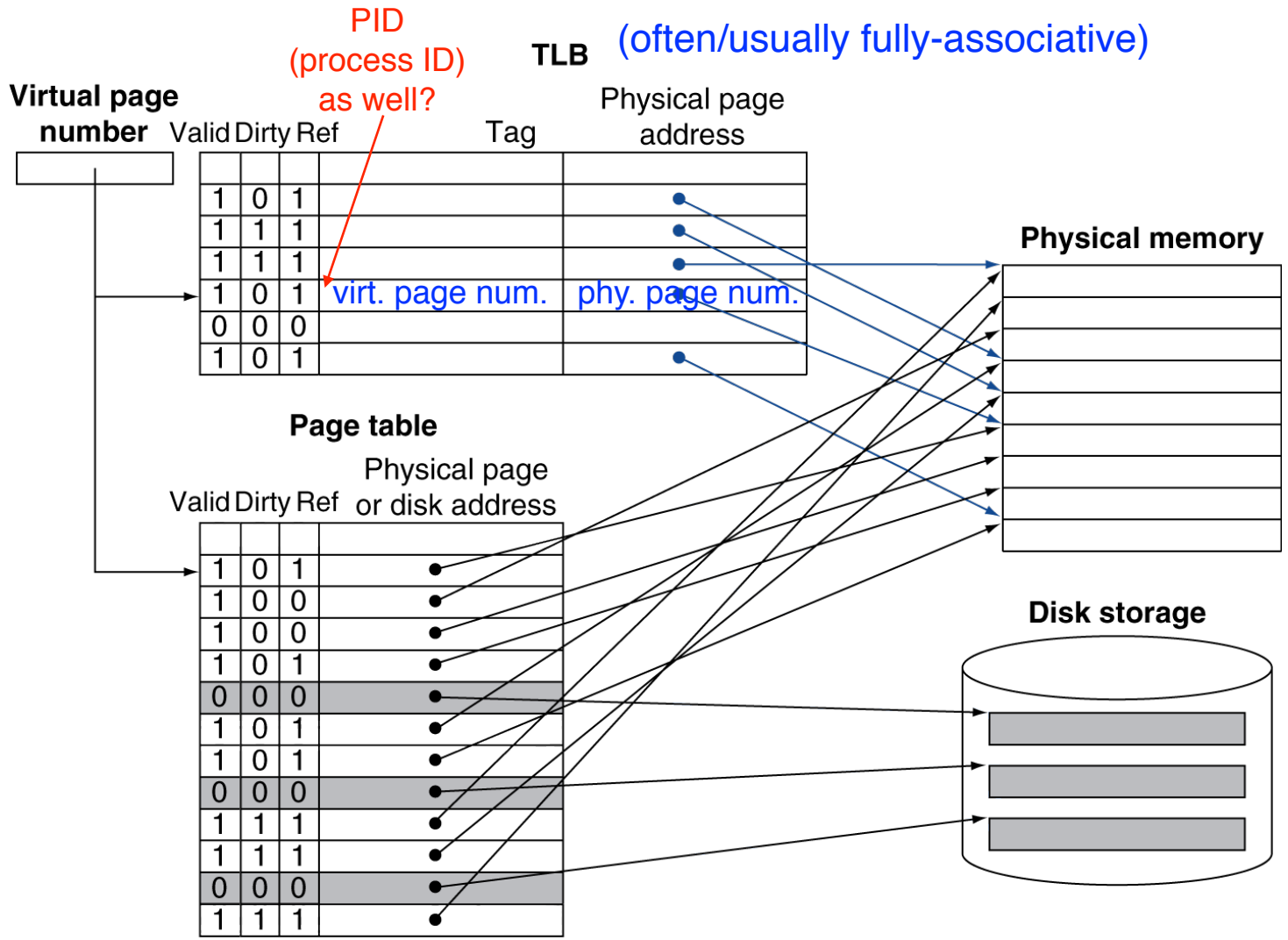
In this example:

- Page size: 4 KBytes
- Virtual Address Space: 128 KBytes
  => 32 virtual pages per process
- Physical Address Space: 1 MByte
  => 256 physical pages

Replacement and Writes

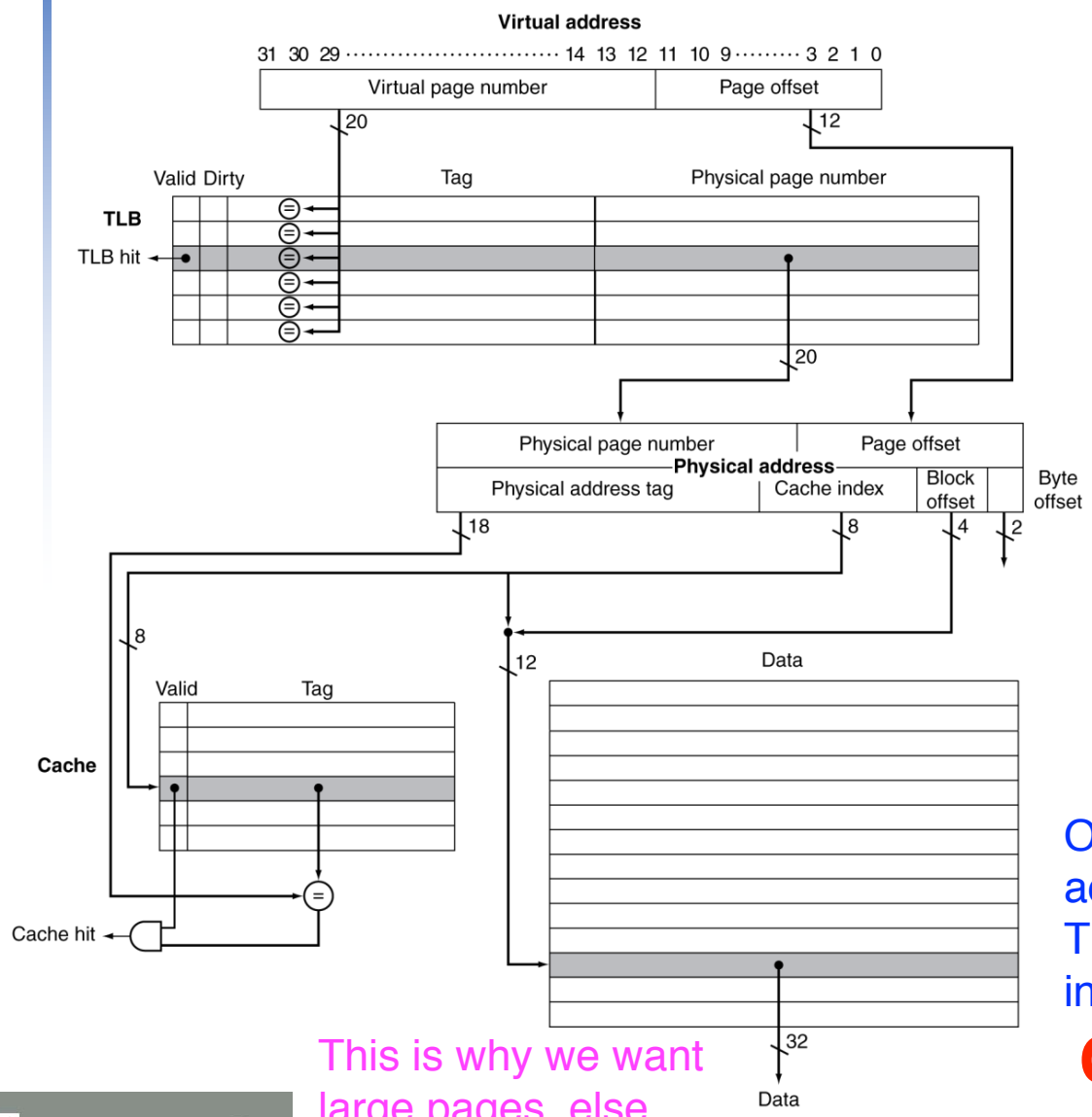
- To reduce page fault rate, prefer least-recently used (LRU) replacement
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 has not been used recently
- Disk writes take millions of cycles
  - Block at once, not individual locations
  - Write through is impractical
  - Use write-back
  - Dirty bit in PTE set when page is written

Fast Translation Using a TLB



TLB= Translation Look-aside Buffer (a cache of Page-Table entries)

TLB and Cache Interaction



- If cache tag uses physical address
  - Need to translate before cache lookup
- Alternative: use virtual address tag
  - Complications due to aliasing
    - Different virtual addresses for shared physical address

Often we want: physical addr. cache, and TLB access in parallel with tag read from cache. This requires cache index to be fully contained in page offset bits, which means:

Cache Way Size ≤ Page Size

This is why we want large pages, else forced to increase associativity of L1

Memory Protection

- Different tasks can share parts of their virtual address spaces
  - But need to protect against errant access
  - Requires OS assistance

Supervisor mode can only be entered at (hardwired) exception handler address, only through ecall or exception

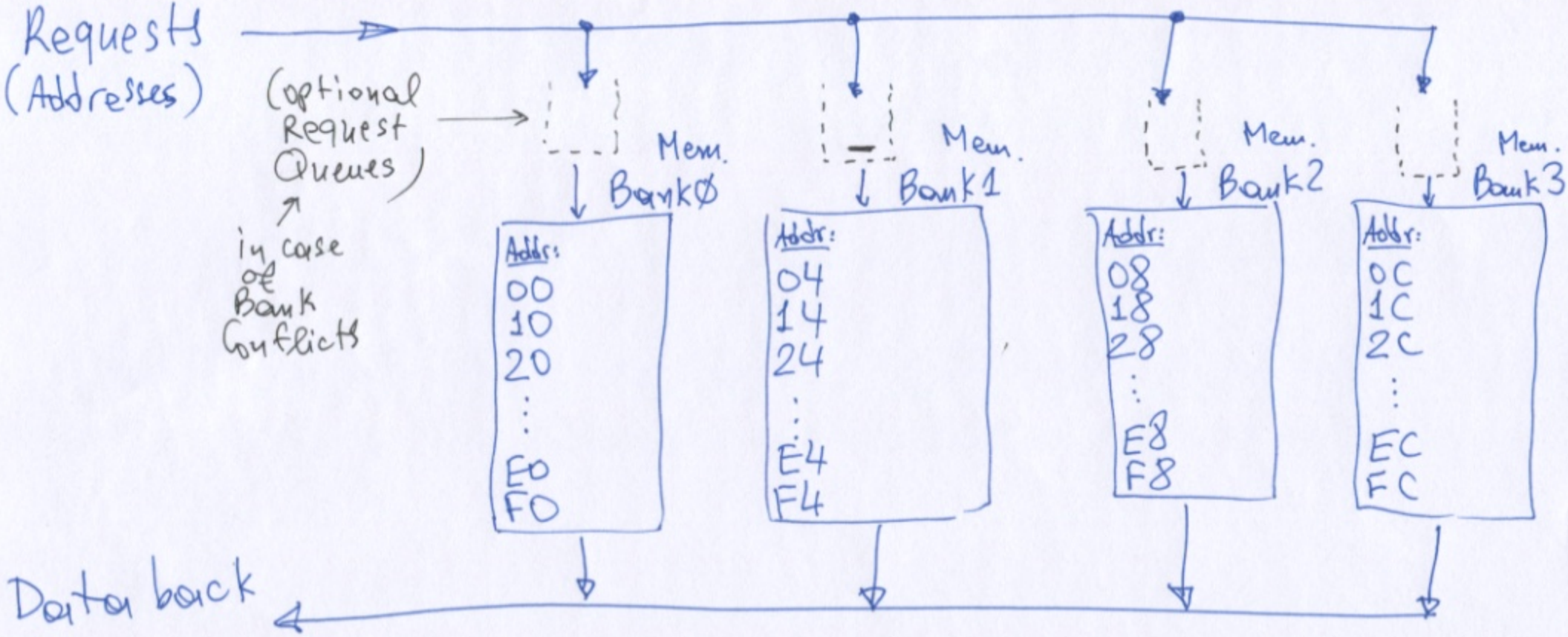
- Hardware support for OS protection
  - Privileged supervisor mode (aka kernel mode)
  - Privileged instructions only executable in supervisor mode
  - Page tables and other state information only accessible in supervisor mode
  - System call exception (e.g., ecall in RISC-V)

Handling Exceptions

- Save PC of offending (or interrupted) instruction
  - In RISC-V: Supervisor Exception Program Counter (SEPC)
- Save indication of the problem
  - In RISC-V: Supervisor Exception Cause Register (SCAUSE)
  - 64 bits, but most bits unused
    - Exception code field: 2 for undefined opcode, 12 for hardware malfunction, …
- Jump to handler
  - Assume at $0000\ 0000\ 1C09\ 0000_{hex}$

Many RISC-V computers store the exception entry address in a special register named Supervisor Trap Vector (STVEC), which the OS can load with a value of its choosing.

Interleaved Memory Banks (Διαφύλλωση Μνήμης)



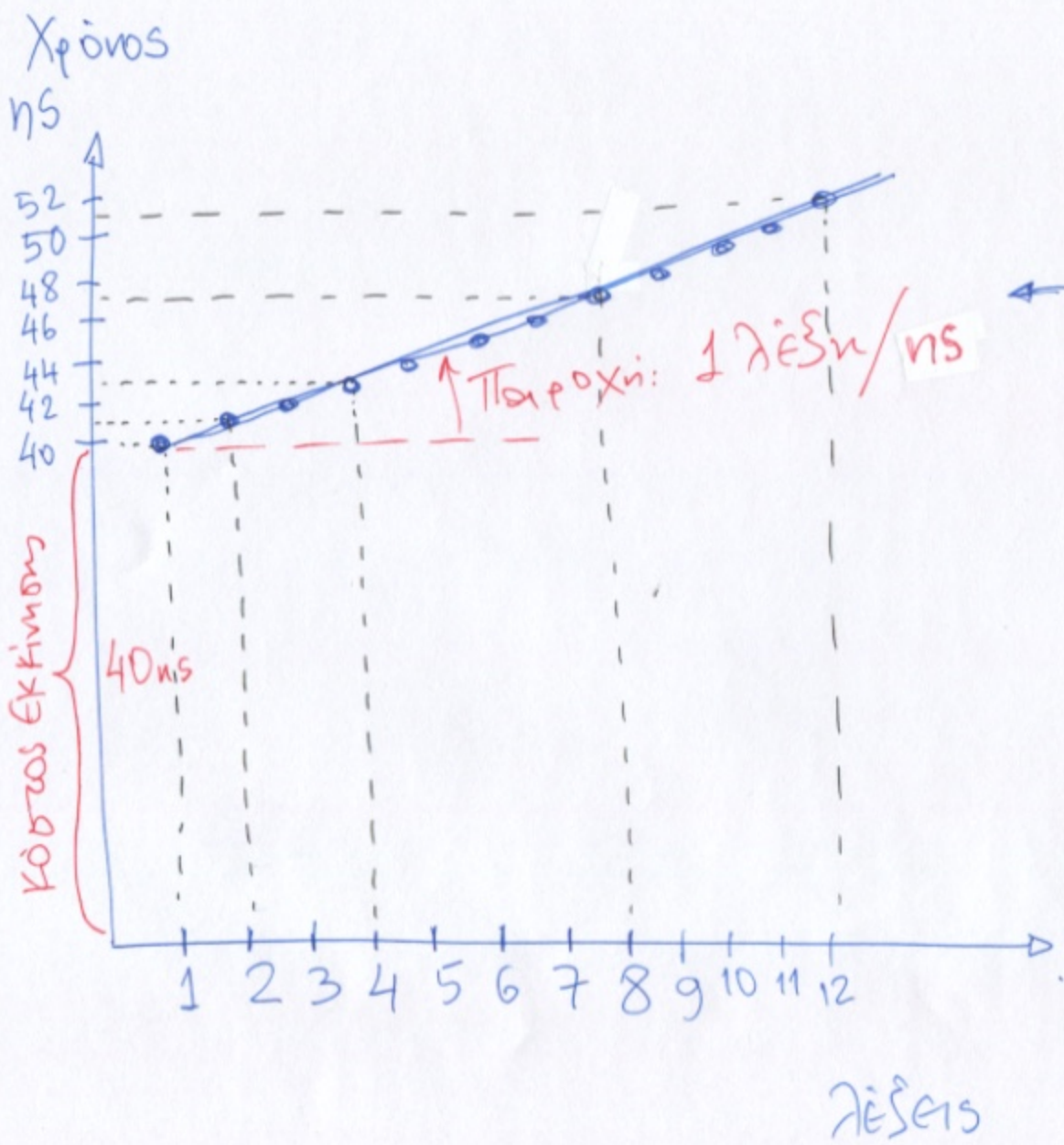
e.g.: 100 ns per Bank Access, one new Request every 25 ns

"Split Transactions" on request/reply "bus" ... pipelining

... multiple transactions interleaved in time

§13.2 Κόστος Εκκίνησης, Παροχή (startup Cost vs. Throughput)

=>Amortize cost over Large data blocks

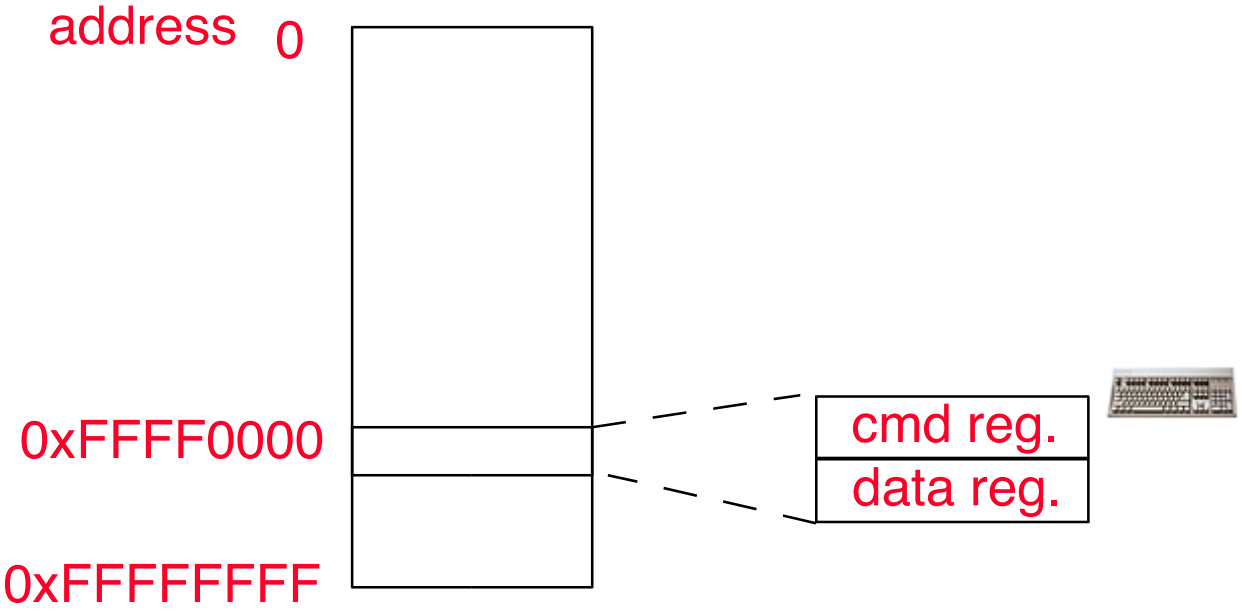


Παραδείγματα:

- η pipeline μας:
  - Κόστος εκκίνησης = 5 κύκλοι ρολ.
  - παροχή ≈ 1 εντολή/κύκλο
- DRAM - προσβ. σε συνεχόμενες λέξεις
  - Κόστος εκ. = (row) access time
  - παροχή - π.χ. 500MHz clock DDR timing
    ⇒ μια λέξη ανά $\frac{1}{2}T_{ck} = \frac{2ns}{2} = 1ns$
- Δίσκος π.χ. (milli 10^{-3})
  - Κόστος εκκίνησης ≈ 10 ms
  - παροχή ≈ 100 MBy/s = 1 Byte ανά 10 ns (nano 10^{-9})
- Δίκτυα π.χ. 20km, 10 Gb/s
  - πρώτο bit: $\frac{20\,km}{200\,\frac{Mm}{s}} \approx 0.1ms = 100\mu s$ (ταχ. φωτός σε οπτική ίνα)
  - $10\,\frac{Gb}{s} = 1$ bit ανά 0.1 ns

Memory Mapped I/O

- Certain addresses are not regular memory
- Instead, they correspond to registers in I/O devices



Processor Checks Status before Acting

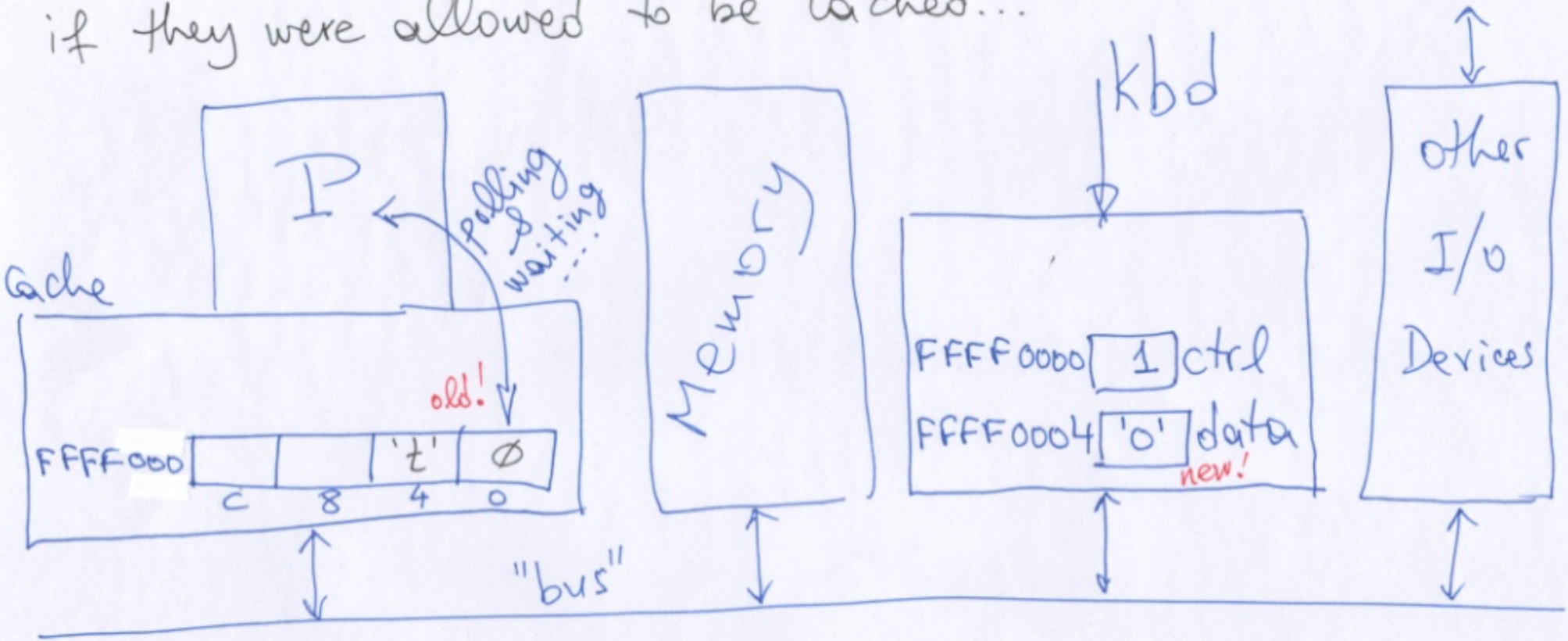
- Path to device generally has 2 registers:
  - 1 register says it's OK to read/write (I/O ready), often called Control Register
  - 1 register that contains data, often called Data Register
- Processor reads from Control Register in loop, waiting for device to set Ready bit in Control reg to say its OK (0 Þ 1)
- Processor then loads from (input) or writes to (output) data register
  - Load from device/Store into Data Register resets Ready bit (1 Þ 0) of Control Register

"Polling"

"Busy wait" if done continuously; else, poll multiple devices on every interrupt from the real-time clock (usu. 50-120 Hz)

I/O Address Pages must be non-cacheable!

if they were allowed to be cached...

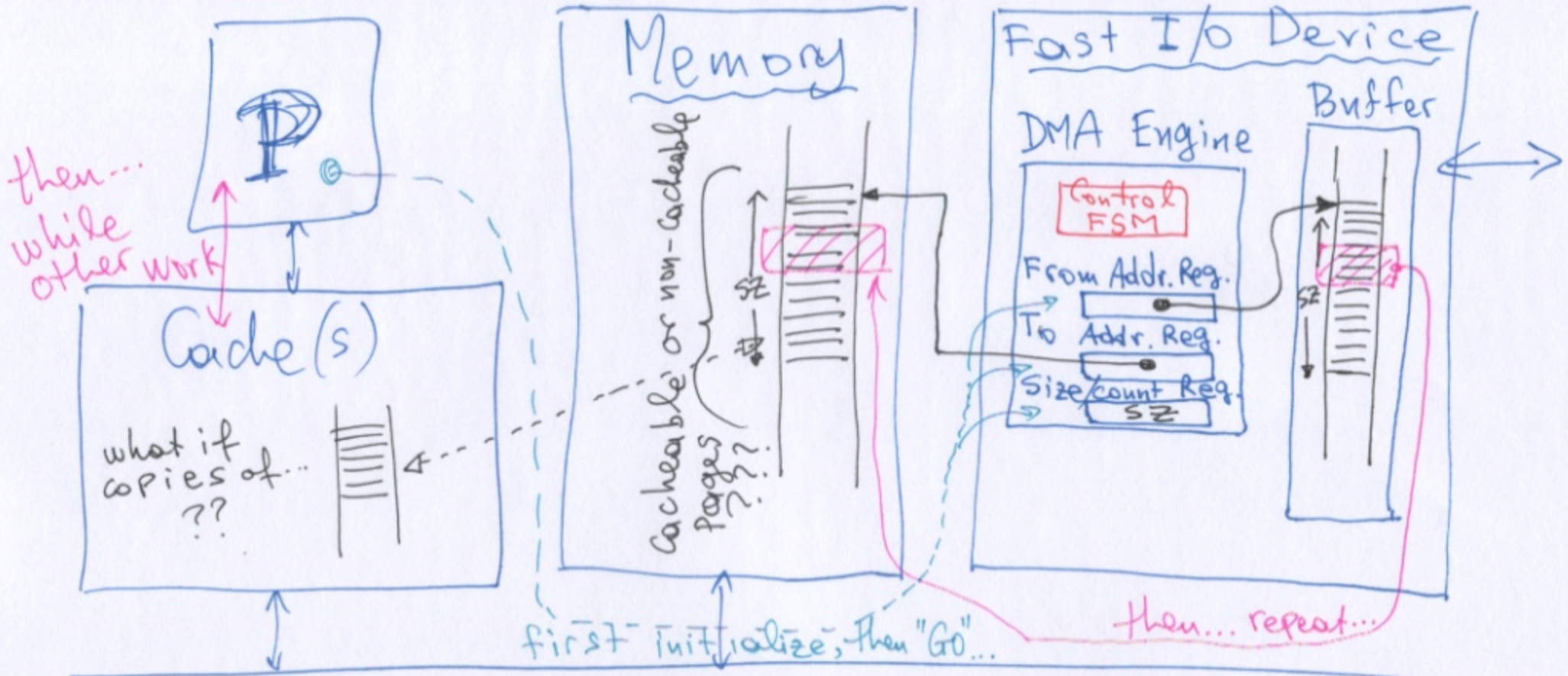


- traditional ("non-coherent"...) caching does NOT work when other devices (I/O, other proc. cores) access memory independently
- note: write-through is a "half-solution": works for output, but not for input...

I/O Interrupt

- An I/O interrupt is like an overflow exceptions except:
  - An I/O interrupt is “asynchronous”
  - More information needs to be conveyed
- An I/O interrupt is asynchronous with respect to instruction execution:
  - I/O interrupt is not associated with any instruction, but it can happen in the middle of any given instruction
  - I/O interrupt does not prevent any instruction from completion

Direct Memory Access (DMA)



Alternatives for cacheability: (write-through only solves half the problem)

- DMA onto non-cacheable memory pages ... too slow when processor processes the I/O data
- Flush the cache before/after I/O DMA ... quite expensive operation (total flush? selective flush? (scan entire cache))
- Cache-Coherent DMA ← good! → next chapter...

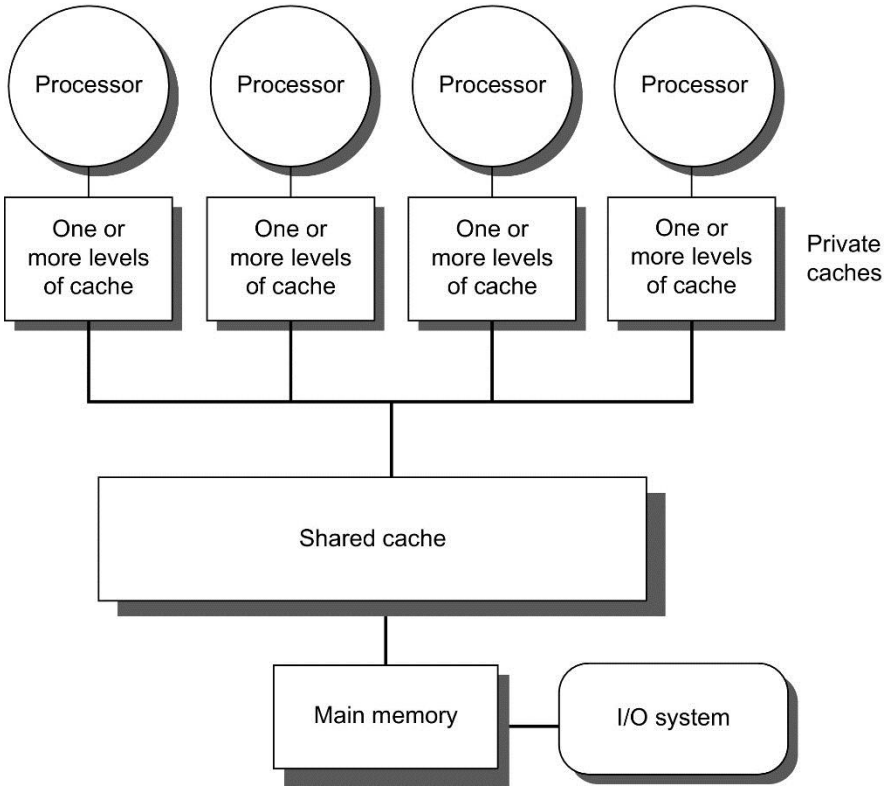
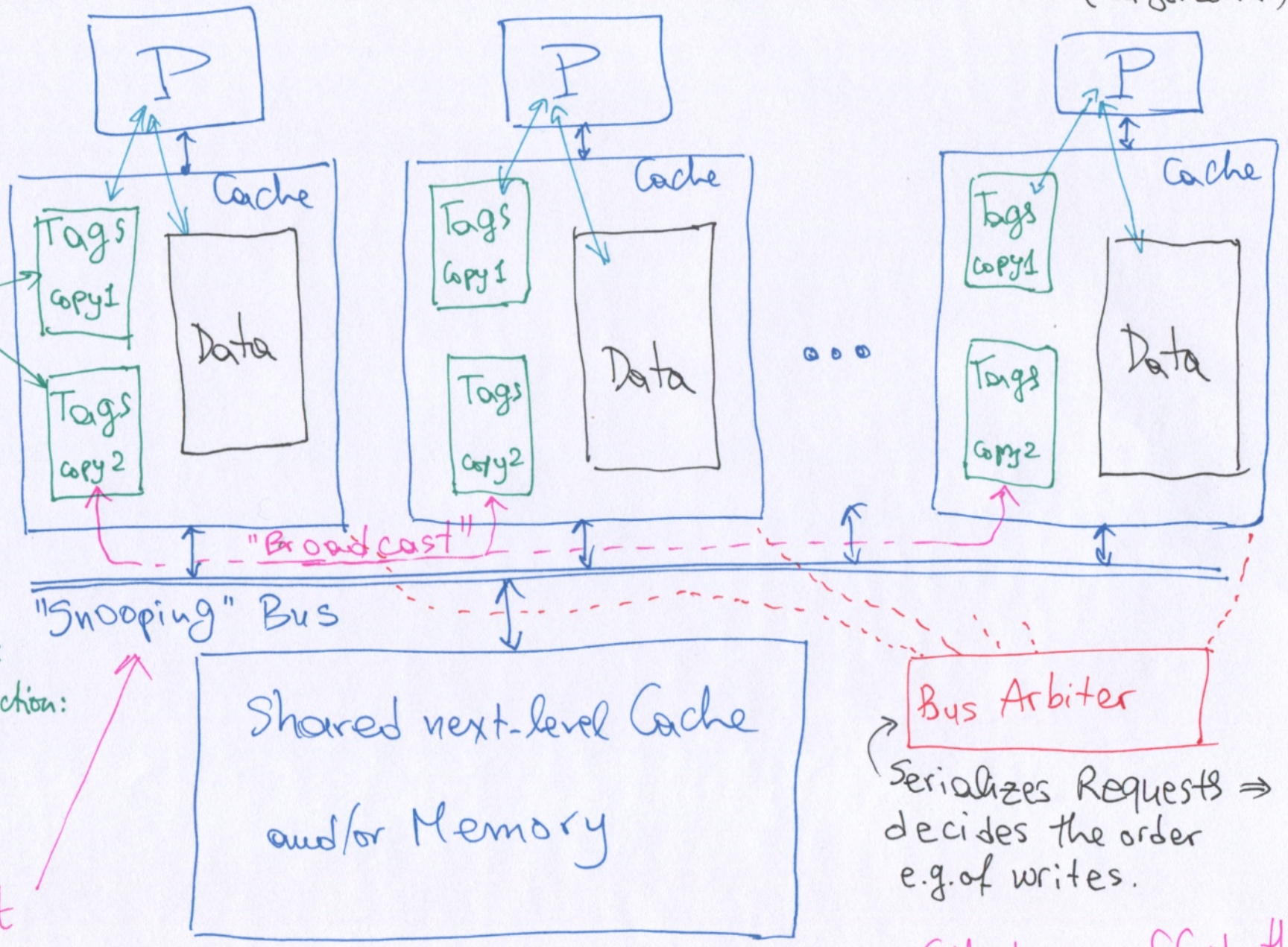


Figure 5.1 Basic structure of a centralized shared-memory multiprocessor based on a multicore chip. Multiple processor-cache subsystems share the same physical memory, typically with one level of shared cache on the multicore, and one or more levels of private per-core cache. The key architectural property is the uniform access time to all of the memory from all of the processors. In a multichip design, an interconnection network links the processors and the memory, which may be one or more banks. In a single-chip multicore, the interconnection network is simply the memory bus.

Snooping Cache Coherence (Συνοχή κρυφής Μνήμης με κρυφοκοίταγμα/καταπολέμηση (κουτσομπολιό στη γειτονιά))



Two copies of tags, or 2-port TagsMemory: mostly read-only; only need to write infrequently: on misses or when affected by bus transaction: then, have to write into both copies.

Broadcast Medium: everybody has to listen to everything going on (that may affect others) — when does it affect others ???

Serializes Requests ⇒ decides the order e.g. of writes.

Upon Writes: to Invalidate or to Update the Others?

- **Invalidate-based Protocols:**

  When I write (modify) something that I have, and there is a danger others may have copies of it, tell them to invalidate their copies!

  (like telling them: "If you ever need it again, come back to me and ask me for its latest value")

  - Advantage: from that point on, I know that I have the only copy, therefore I can freely "play with it", as long as nobody ask me for a copy again.

  (most usual)

- **Update-based Protocols:**

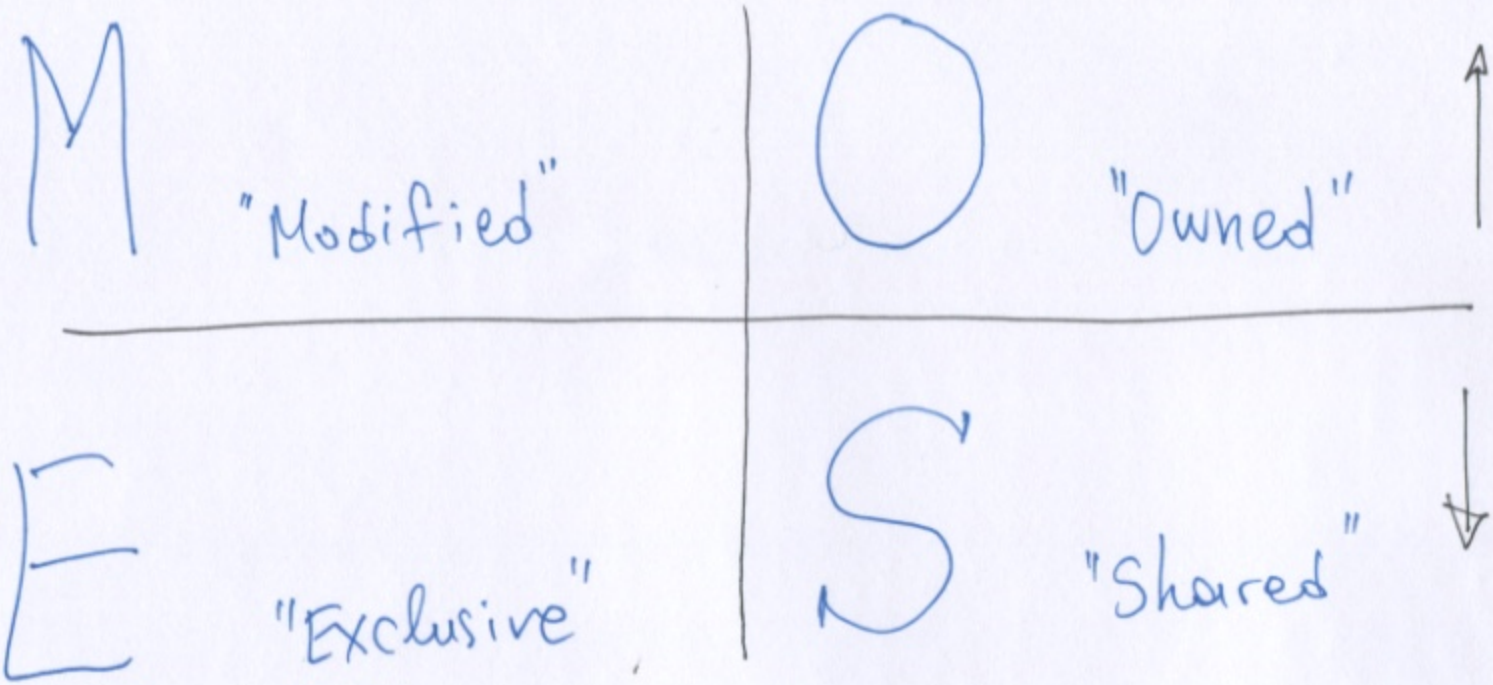
  When I write (modify) something that I have, and there is a danger others may have copies of it, broadcast the new value so as to update the values of all copies!

  - Advantage: if others will need the value again soon, they will have it in their caches
  - Disadvantage: multiple copies will keep existing, hence the need to continuously keep updating them

  (rarely (if ever) used)

(statistics showed that disadvantage is important in the general case (unless there is hint about some data by the algorithm/software)

Cache Line States: what do I know about it? "MOESI"



Dirty: I have modified my copy, and I am responsible for writing it back to memory

Clean: It is guaranteed that the memory (and others) have the same value as I do.

Exclusive:
It is guaranteed that my copy is the ONLY copy currently existing in any cache.

(potentially) Shared:
other caches may have copies of this line (not known for sure)

I "Invalid"
nothing in this Line.

Simplification: MSI Protocol

(No "E" info when Clean; If Dirty, must be Exclusive (no "O" state))

M (Modified): Dirty and Exclusive

- with invalidate-based protocol, when I write into my copy, I have to invalidate all others, hence I am guaranteed to have the ONLY copy ⇒ Exclusive

S (Shared): (potentially) Shared and guaranteed Clean

- when first reading from memory → S (no "E" info maintained)
- for simplicity, I do not keep track whether or not others too may have copies
- if I had the line as M, and another cache misses it, then:
  - I have to write it back to memory (no "O" state, for simplicity)
  - the other cache gets a copy automatically on the bus
- I may have had the line as S, and all other caches may have evicted their copies, so I may be the only one having a copy, but I do NOT know that for sure...
- If I have the line as S, and I want to write into it, I must broadcast an Invalidate command, since I have no "E" info!

I (Invalid)

from MSI to MESI protocol:

- Add and maintain E (exclusive and clean) info:
  - When I read-miss, if no other cache responds (faster than memory) providing me the data, and I have to wait for the memory to bring me the data ⇒ then I know I am "E".
  - Advantage versus MSI: If the line is "E" and I want to write into it, I do NOT need to broadcast any invalidate: save bus traffic.

MOESI protocol:

- Add "O" (Owned) info: Line is shared, Memory is "old", and the responsibility to write back is mine!
  - When the line is "M" (Modified: dirty and exclusive), and another cache read-misses on it, I supply the data to the other cache (faster than memory), but I do not spend the time to write-back to memory (now): save time (for now).
  - The other caches that have copies, have them in S state, hence they are allowed to evict their copies without writing back: I have the (only) "O" copy, hence the responsibility to write-back is mine!

Dynamic Multiple Issue

- "Superscalar" processors
- CPU decides whether to issue 0, 1, 2, … each cycle (checks dependencies and)
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

Allows executables to run on newer processors, with same ISA but different pipeline, without needing to be recompiled

Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
ld   x31,20(x21)
add  x1,x31,x2
sub  x23,x23,x3
andi x5,x23,20
```

  - Can start sub while add is waiting for ld

Out-of-Order (ooo) Execution

In-Order Commit

(so as to flush results of mis-speculated instructions, and also allow precise exceptions)

Multithreading

One "thread of control" = one (traditional) sequential program.
Multiple threads = parallel program.

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads

mimic multiple cores, thus:

but Share the Functional Units and the Caches

- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)