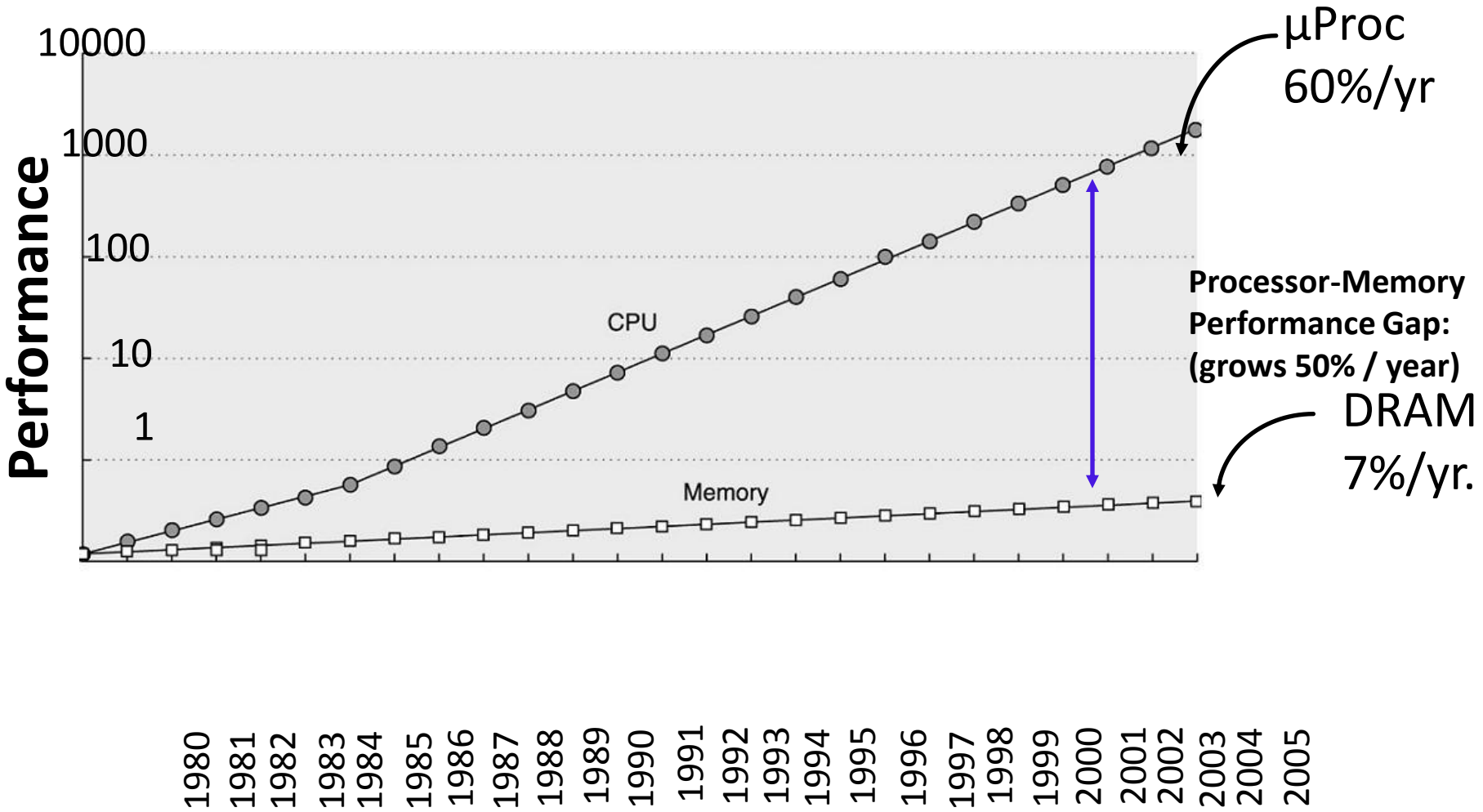
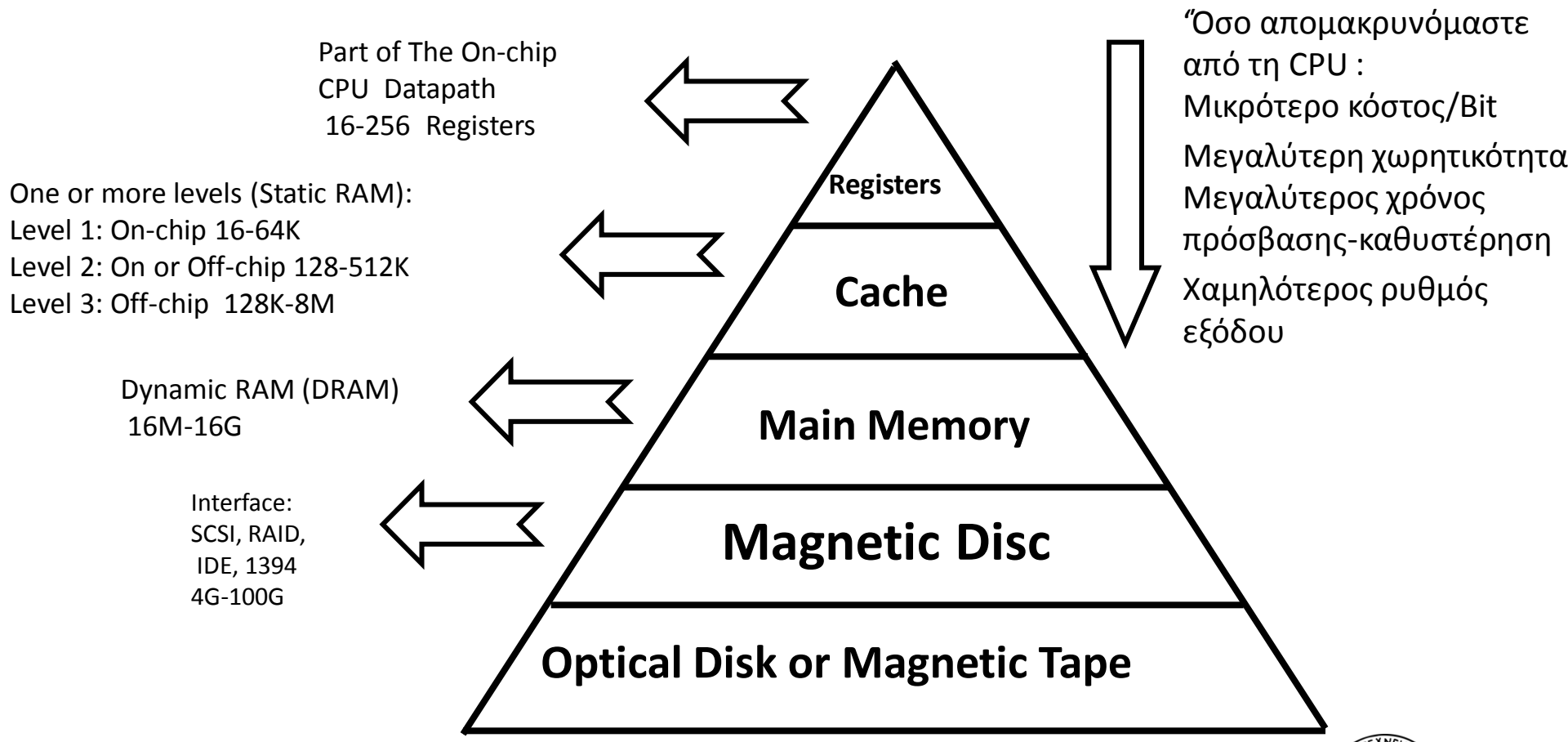
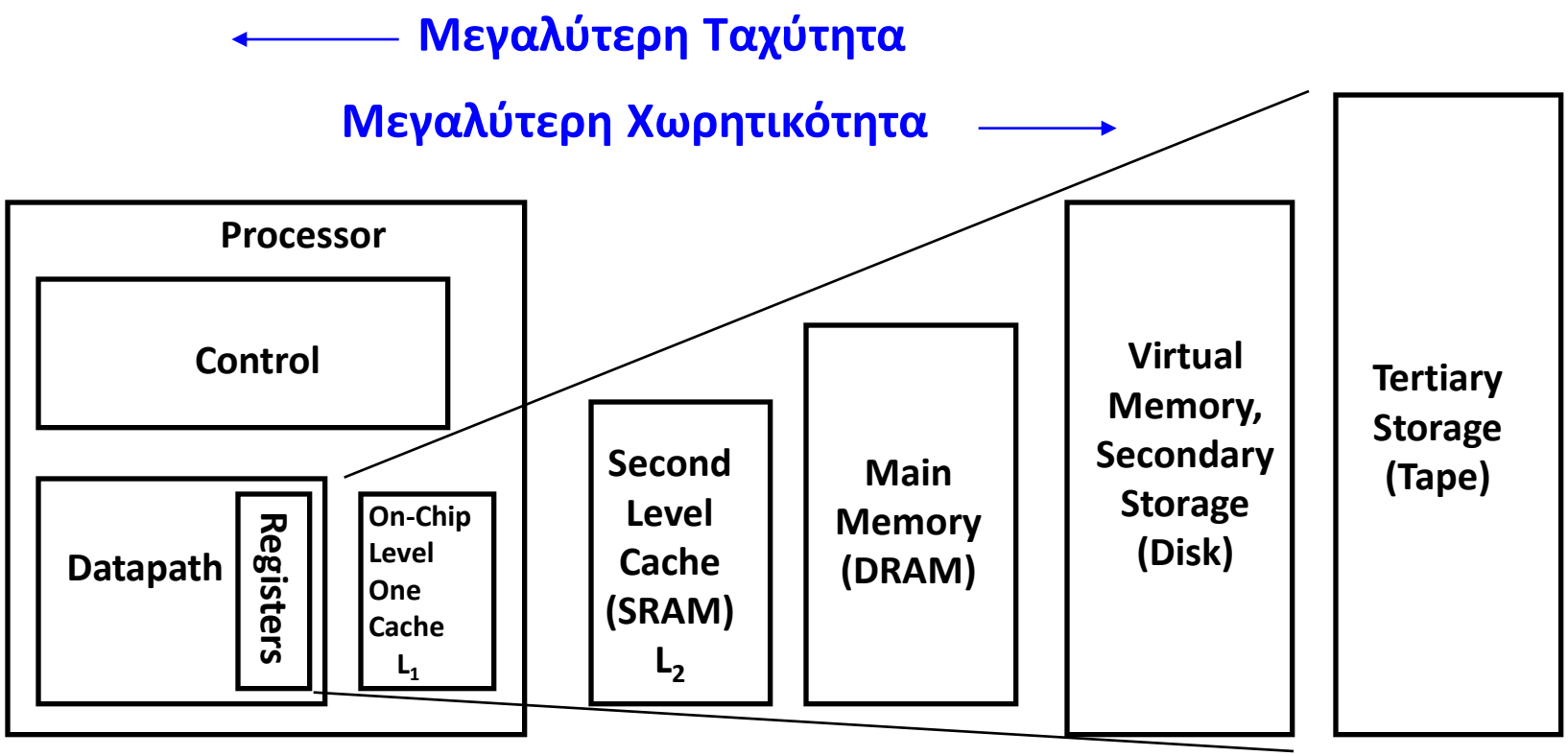
Processor-Memory (DRAM) Διαφορά επίδοσης



Ιεραρχία μνήμης

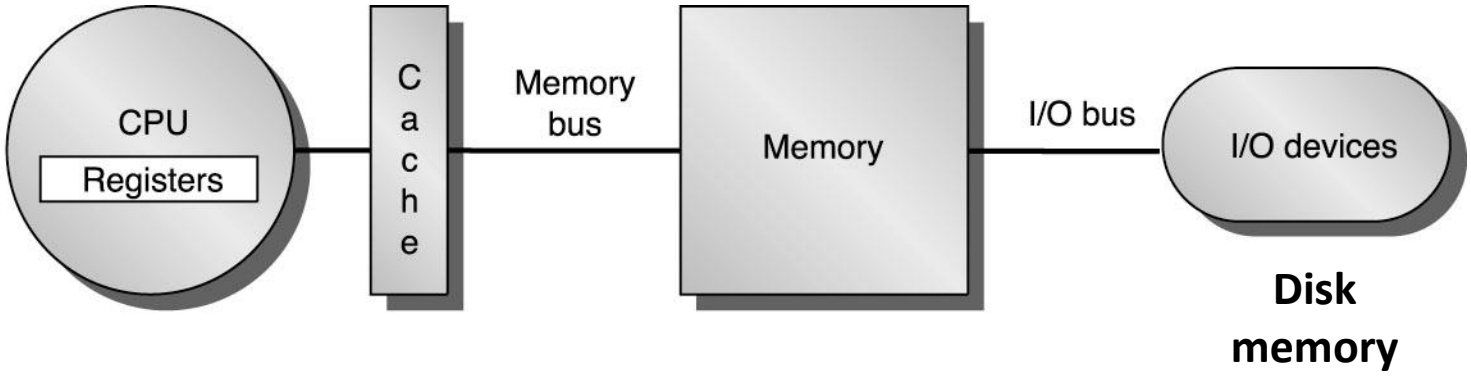


Παράδειγμα Ιεραρχίας Μνήμης (με 2 επίπεδα cache)



Ταχύτητα :	1-5ns	10ns	30ns	50ns	10ms	10sec
Μέγεθος :	<1KB	<256KB	<8MB	<4GB	>1GB	TB
Bandwidth :	150GB/s	50GB/s	25GB/s	4GB/s	10MB/s	

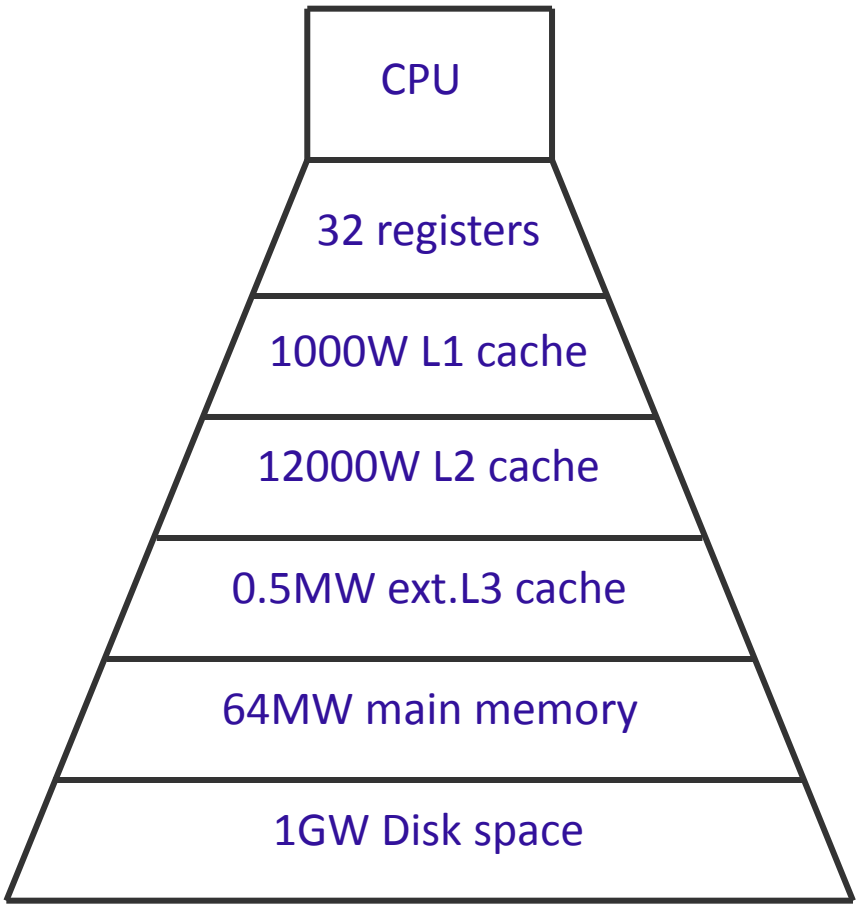
Το μοντέλο της Ιεραρχίας Μνήμης



μέγεθος :	**500bytes**	**64KB**	**512MB**	**100GB**
ταχύτητα :	**0,25ns**	**1ns**	**100ns**	**5ms**

Παράδειγμα Ιεραρχίας μνήμης

Digital PWS 600 au - Alpha 21164 CPU – 600MHz



Level	Capacity	Throughput	Latency
Register	512B	24GB/sec	2ns
L1 cache	8KB	16GB/sec	2ns
L2 cache	96KB	8GB/sec	6ns
L3 cache	4MB	888MB/sec	24ns
Main Mem	512MB	1GB/sec	112ns

Τυπικές Αρχιτεκτονικές

- **IBM Power 3 (1998):**
  - L1 = 64 KB, 128-way set associative
  - L2 = 4 MB, direct mapped, line size = 128, write back
- **Compaq EV6 (Alpha 21264):**
  - L1 = 64 KB, 2-way associative, line size= 32
  - L2 = 4 MB (or larger), direct mapped, line size = 64
- **HP PA:** no L2
  - PA8500, PA8600: L1 = 1.5 MB
  - PA8700: L1 = 2.25 MB
- **AMD Athlon:** L1 = 64 KB, L2 = 256 KB
- **Intel Pentium 4:** L1 = 8 KB, L2 = 256 KB
- **Intel Itanium:**
  - L1 = 16 KB, 4-way associative
  - L2 = 96 KB, 6-way associative
  - L3 = off chip, size varies
- **IBM Power 8 (2014):**
  - 4, 6, 8, 10 or 12 chiplets (core+L1+L2)
  - L1 = 32KB + 64 KB
  - L2 = 512KB, direct mapped, line size = 128, write back
  - L3 = #chiplets x 8MB
  - up to 1 TB of memory per socket
- **AMD Jaguar (2013):**
  - L1 = 32KB + 32KB per core
  - L2 = 1-2MB, shared by 2-4 cores
- **Intel Haswell (2013):**
  - 2-4, 6-8, 8+ cores
  - L1 = 32KB + 32KB per core
  - L2 = 256KB per core
  - L3 = 4 – 20MB

Γιατί είναι ωφέλιμη η Ιεραρχία Μνήμης;

- Κατά κανόνα τα προγράμματα προσπελαύνουν ένα μικρό μόνο μέρος του συνόλου των διευθύνσεων (εντολές/δεδομένα) κατά την εκτέλεση ενός συγκεκριμένου τμήματός τους

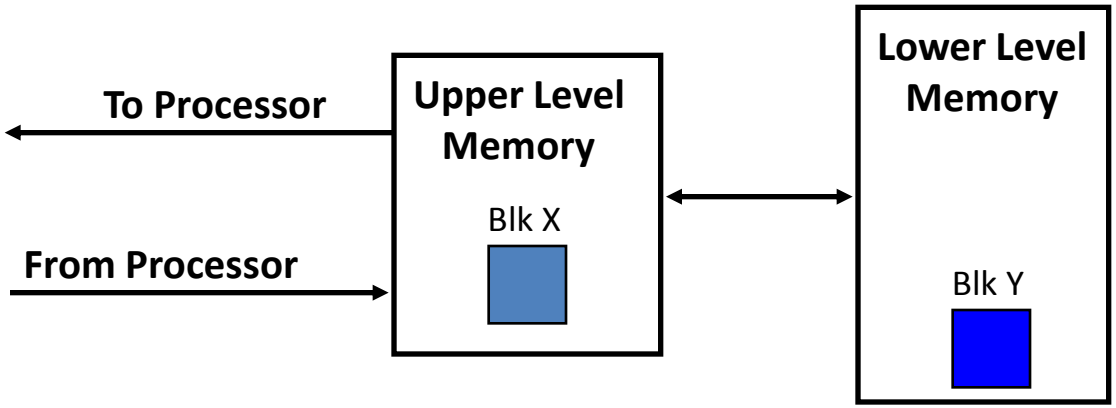
- Δύο είδη τοπικότητας δεδομένων:

  - Temporal Locality: Στοιχεία που έχουν πρόσφατα προσπελαστεί τείνουν να προσπελαύνονται ξανά στο άμεσο μέλλον
  - Spatial locality: Γειτονικά στοιχεία όσων έχουν ήδη προσπελαστεί, έχουν αυξημένη πιθανότητα να προσπελαστούν στο άμεσο μέλλον

- Η ύπαρξη τοπικότητας στις αναφορές ενός προγράμματος, καθιστά εφικτή τη δυνατότητα να ικανοποιούνται η αίτηση για δεδομένα από *επίπεδα μνήμης* που βρίσκονται *ιεραρχικά ανώτερα*

Ορολογία

- **block – line - page** : η μικρότερη μονάδα μεταφοράς δεδομένων μεταξύ των επιπέδων μνήμης



Ορολογία

- **hit** : το block *βρίσκεται* σε κάποια θέση του εξεταζόμενου επιπέδου μνήμης
  - **hit rate** : hits/συνολικές προσπελάσεις μνήμης
  - **hit time** : χρόνος προσπέλασης των δεδομένων

- **miss** : το block *δεν υπάρχει* στο εξεταζόμενο επίπεδο μνήμης
  - **miss rate** : 1 – (hit rate)
  - **miss penalty** : (χρόνος μεταφοράς των δεδομένων ενός block στο συγκεκριμένο επίπεδο μνήμης) + (χρόνος απόκτησης των δεδομένων από την CPU)
    - **access time** : χρόνος απόκτησης της 1ης λέξης
    - **transfer time** : χρόνος απόκτησης των υπόλοιπων λέξεων

Η Βάση της Ιεραρχίας Μνήμης

- Οι δίσκοι περιέχουν όλα τα δεδομένα
- Όταν ο επεξεργαστής χρειάζεται κάποιο στοιχείο, αυτό ανεβαίνει σε ανώτερα επίπεδα μνήμης
- Η cache περιέχει αντίγραφα των στοιχείων της μνήμης που έχουν χρησιμοποιηθεί
- Η μνήμη περιέχει αντίγραφα των στοιχείων του δίσκου που έχουν χρησιμοποιηθεί

4 Ερωτήσεις για τις caches

- Πού μπορεί να τοποθετηθεί ένα block σε ένα ψηλότερο επίπεδο στην ιεραρχία μνήμης;
  - Τοποθέτηση block :
    - *direct-mapped, fully associative, set-associative*
- Πώς βρίσκουμε ένα block στα διάφορα επίπεδα μνήμης;
  - Αναγνώριση ενός block :
    - *Tag / Block*
- Ποιο από τα ήδη υπάρχοντα block της cache πρέπει να αντικατασταθεί σε περίπτωση ενός miss;
  - Μηχανισμός αντικατάστασης block :
    - *Random, Least Recently Used (LRU), FIFO*
- Τι συμβαίνει όταν μεταβάλλουμε το περιεχόμενο ενός block;
  - μηχανισμοί εγγραφής :
    - *write-through ή write-back*
    - *write-allocate ή no-write-allocate*

Οργάνωση της Cache

Τοποθέτηση ενός block μνήμης στην cache

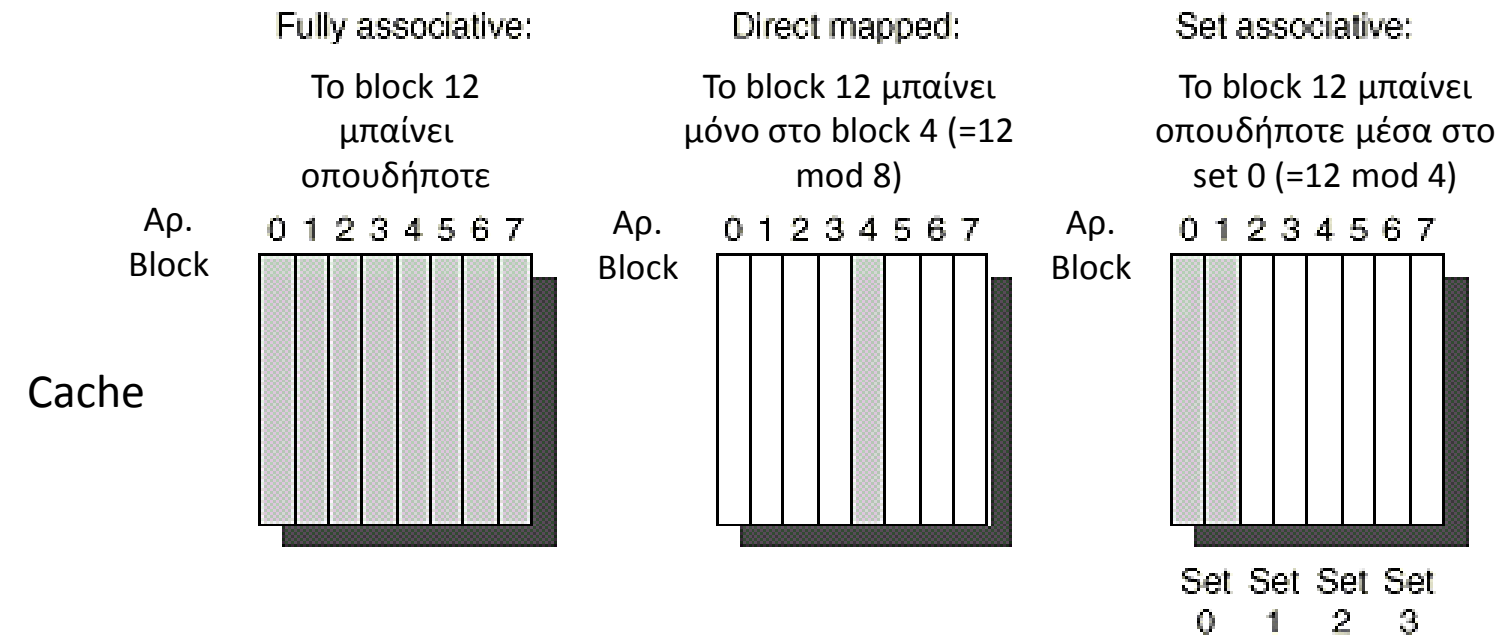
- Direct mapped :

  (διεύθυνση block) mod (αρ. block στην cache)
- Set associative :

  (διεύθυνση block) mod (αρ. sets στην cache)
- Fully associative :

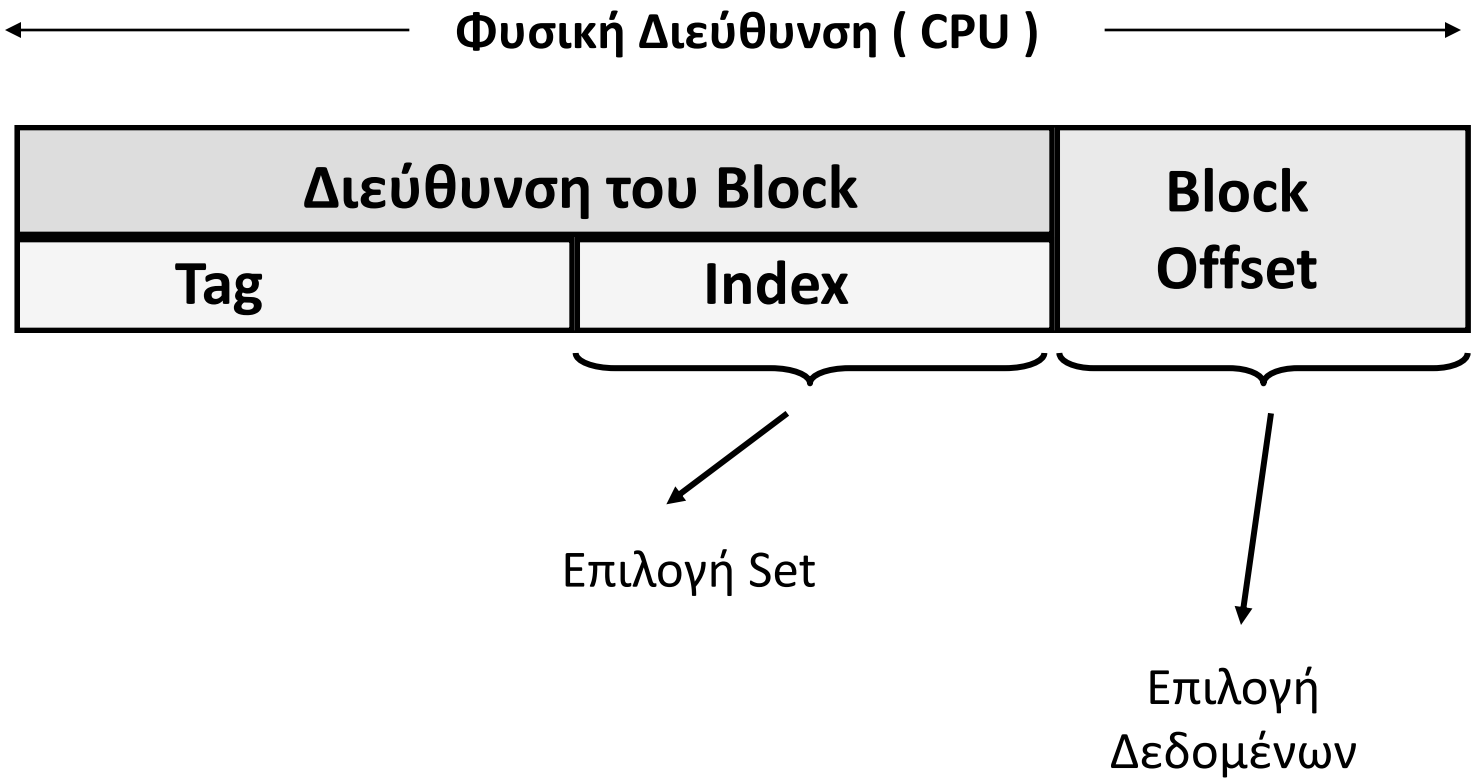
  οπουδήποτε!

Οργάνωση της Cache

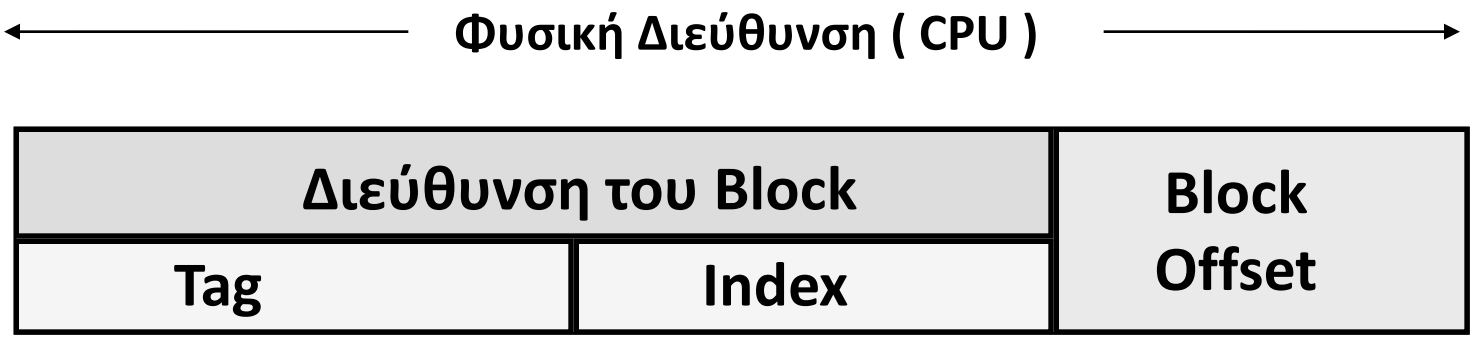


Cache με 8 blocks και μνήμη με 32 blocks

Τα πεδία διεύθυνσης



Τα πεδία διεύθυνσης



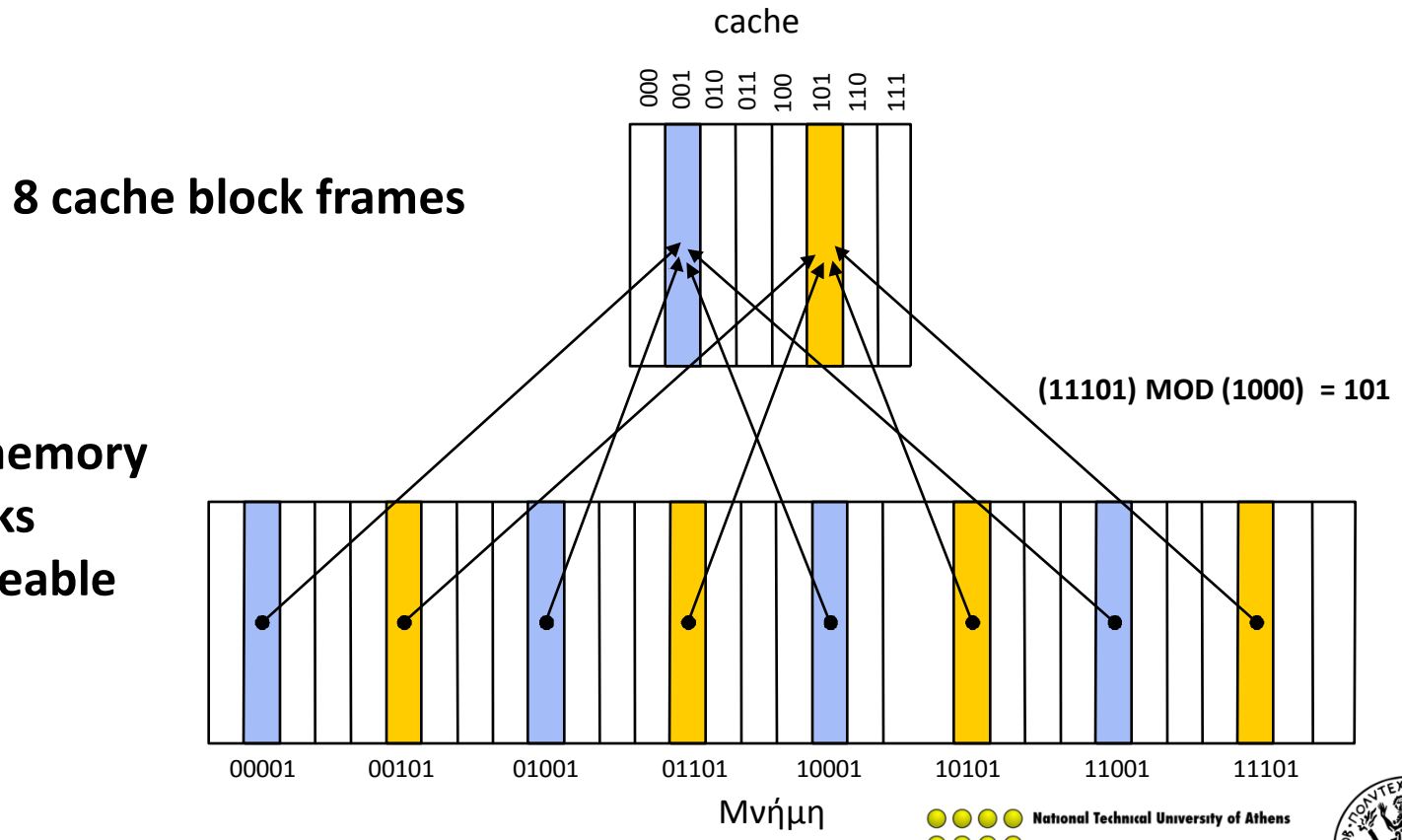
Μέγεθος block offset = log2(μέγεθος block)

Μέγεθος Index = log2(Συνολικός αριθμός blocks/associativity)

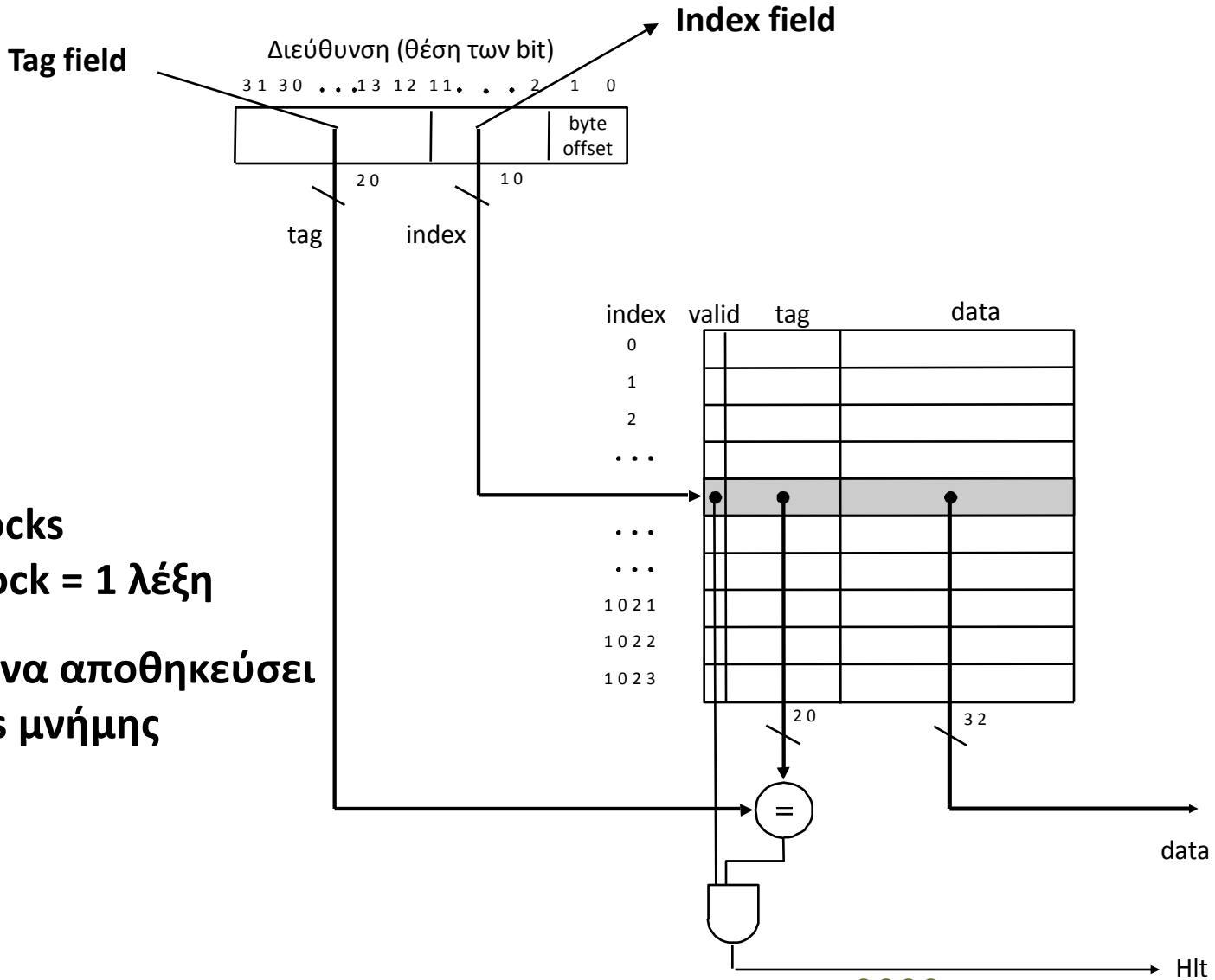
Μέγεθος tag = μέγεθος address - μέγεθος index - μέγεθος offset

Οργάνωση της Cache : Direct Mapped Cache

Κάθε block μπορεί να αποθηκευθεί μόνο σε μία θέση :
(διεύθυνση block) MOD (Αρ. blocks στην cache)
στο παράδειγμά μας: (διεύθυνση block address) MOD (8)



Παράδειγμα : Direct Mapped Cache



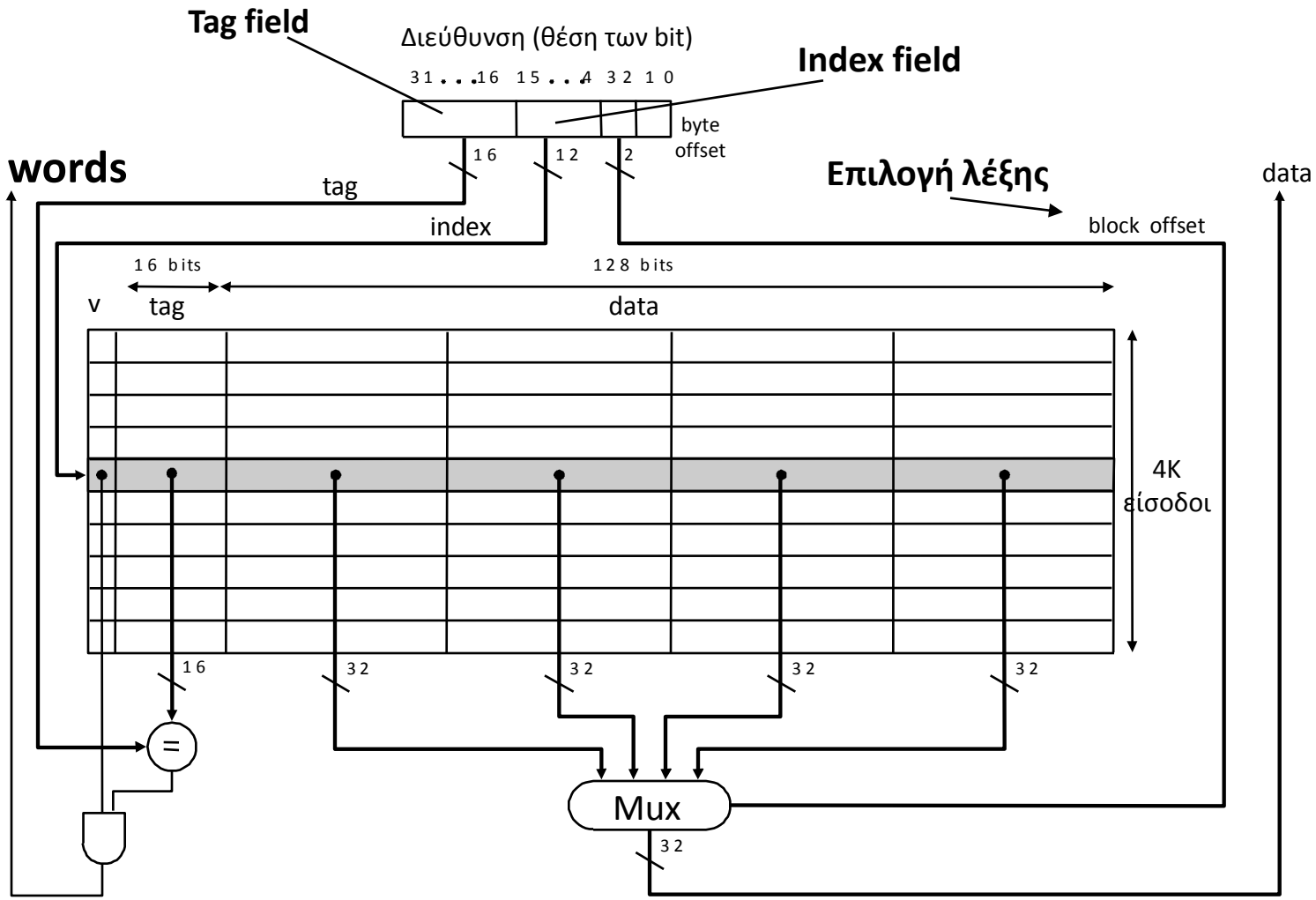
1024 Blocks
Κάθε block = 1 λέξη

Μπορεί να αποθηκεύσει
2^{32} bytes μνήμης

Παράδειγμα : Direct Mapped Cache

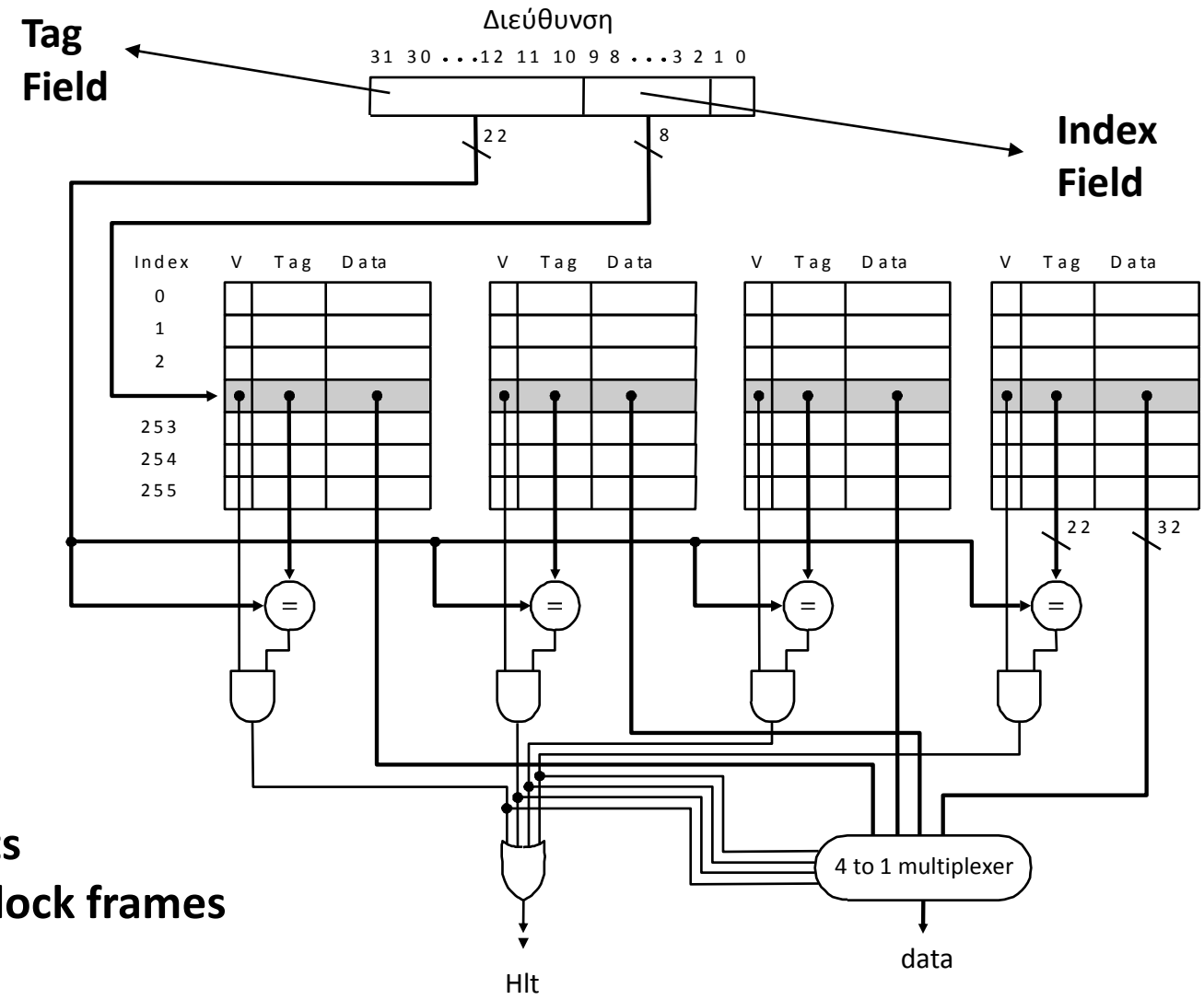
4K blocks

Κάθε block = 4 words



Καλύτερη αξιοποίηση της spatial locality

4-Way Set Associative Cache: (MIPS)



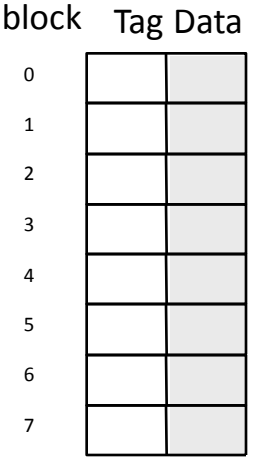
256 sets

1024 block frames

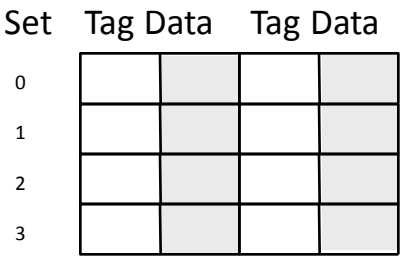
Οργάνωση της Cache : Set Associative Cache

Χωρητικότητα cache : 8 words

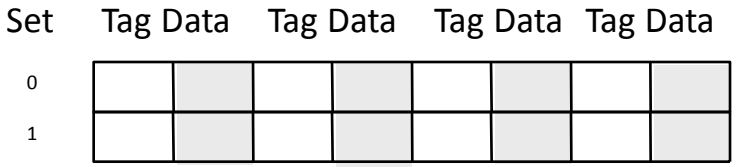
1-way set associative (direct mapped)



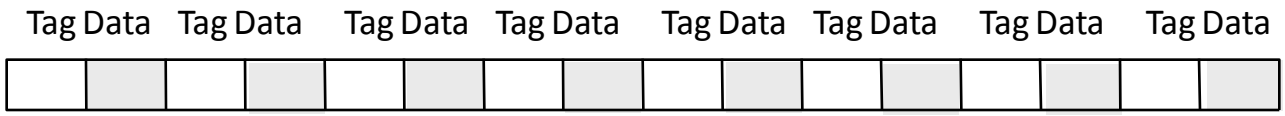
2-way set associative



4-way set associative



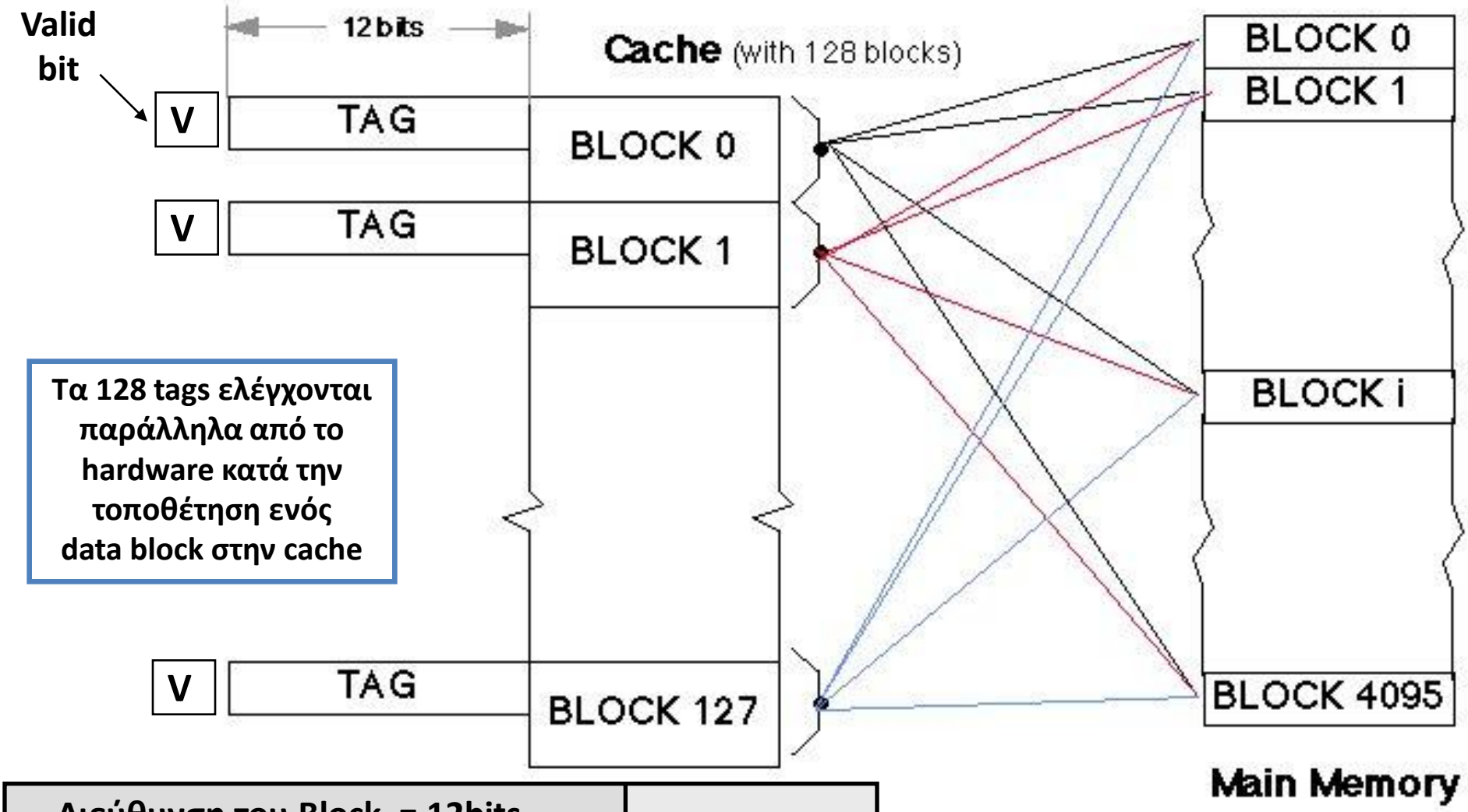
8-way set associative (fully associative)



Παράδειγμα οργάνωσης cache-διευθυνσιοδότηση

- L_1 cache με 128 cache block frames
- Κάθε block frame περιέχει 4 λέξεις (16 bytes)
- 16-bit διευθύνσεις μνήμης στην cache
  (64Kbytes κύρια μνήμη ή 4096 blocks μνήμης)
- Δείξτε την οργάνωση της cache (mapping) και τα πεδία διευθύνσεων της cache για:
  - Fully Associative cache.
  - Direct mapped cache.
  - 2-way set-associative cache.

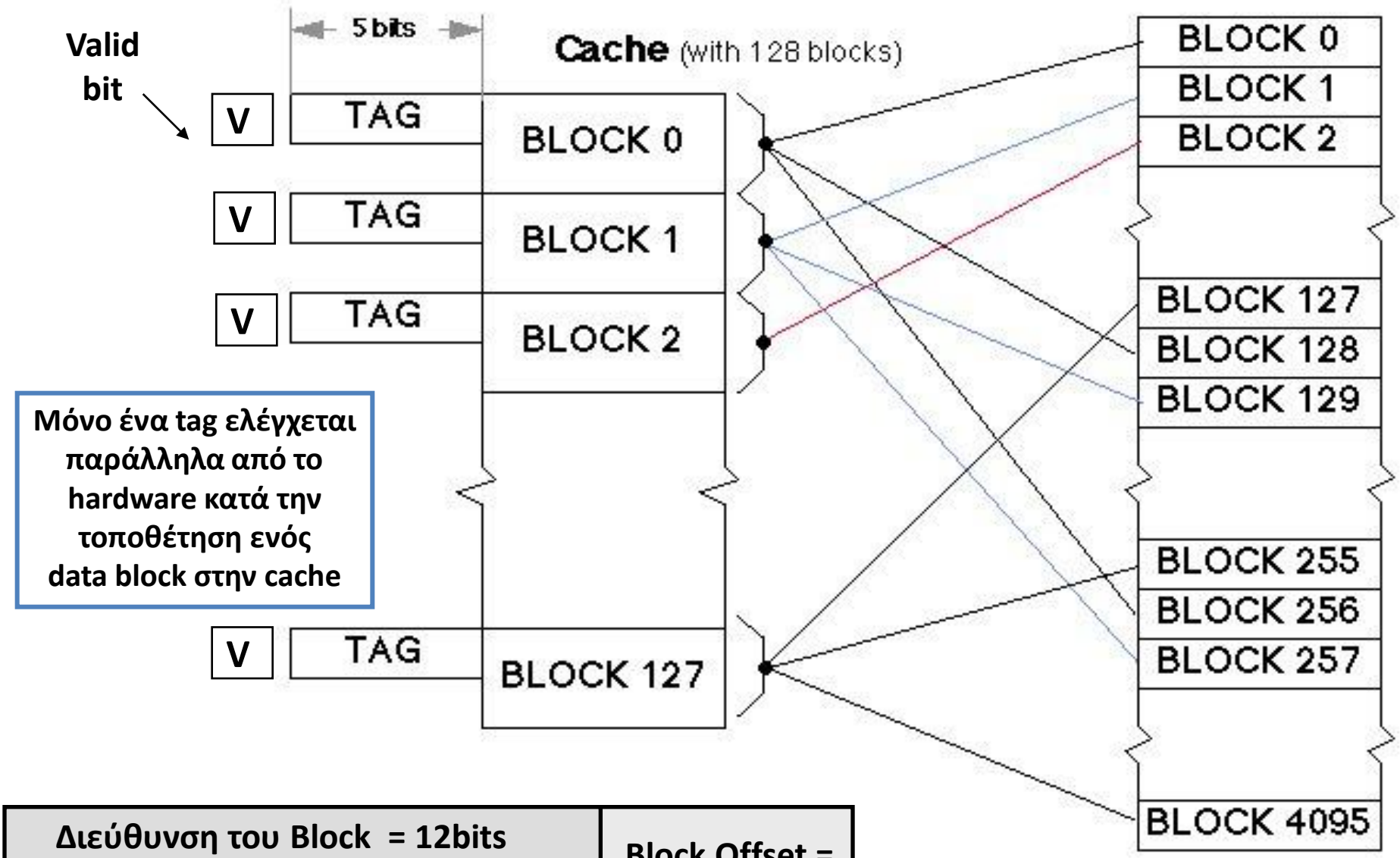
Fully Associative Case



Τα 128 tags ελέγχονται παράλληλα από το hardware κατά την τοποθέτηση ενός data block στην cache

Διεύθυνση του Block = 12bits	Block Offset = 4bits
Tag = 12bits	

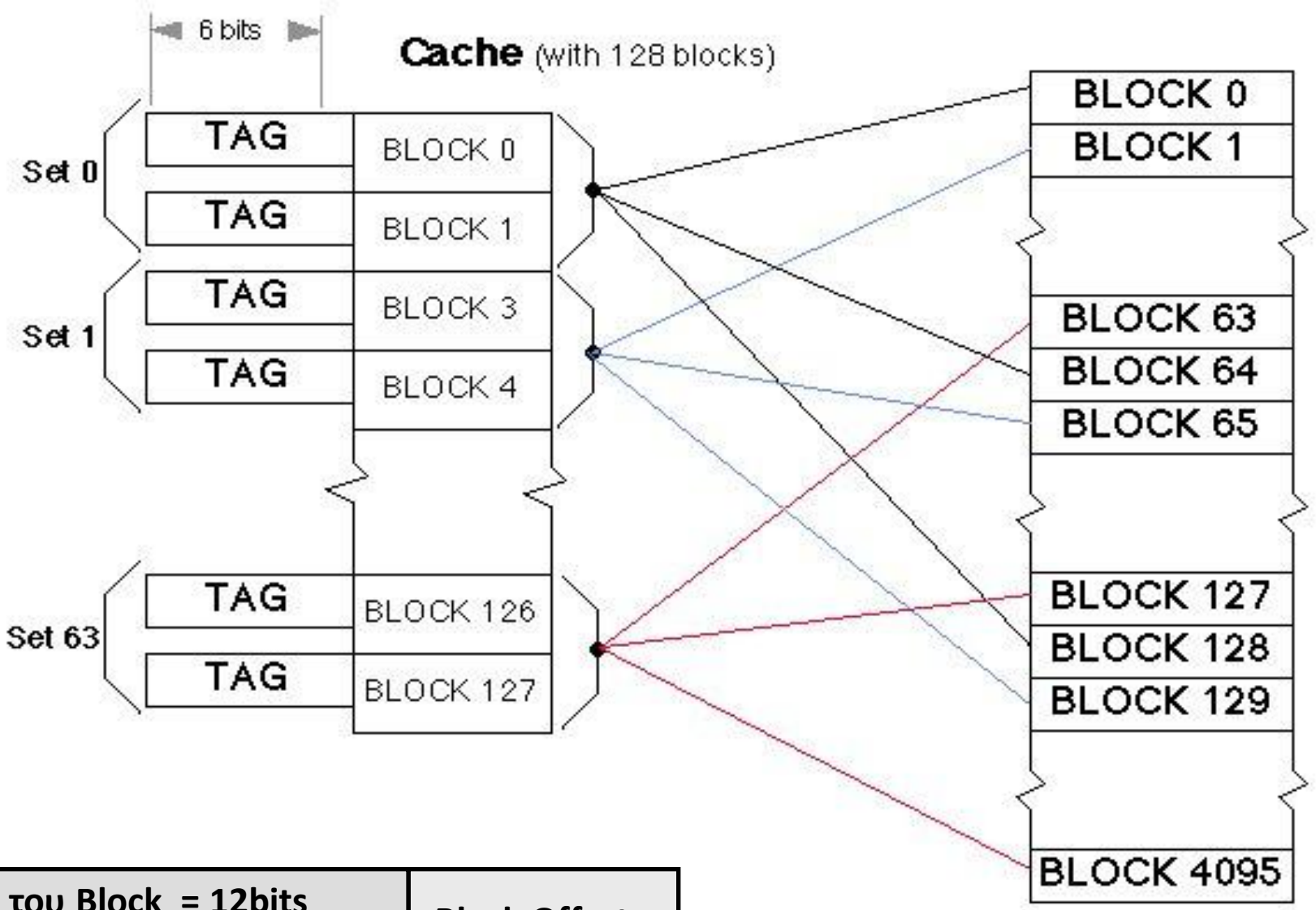
Direct Mapped Cache



Μόνο ένα tag ελέγχεται παράλληλα από το hardware κατά την τοποθέτηση ενός data block στην cache

Διεύθυνση του Block = 12bits		Block Offset = 4bits
Tag = 5bits	Index = 7bits	

2-Way Set-Associative Cache



Δύο tags σε ένα set ελέγχονται παράλληλα από το hardware κατά την τοποθέτηση ενός data block στην cache

Διεύθυνση του Block = 12bits		Block Offset = 4bits
Tag = 6bits	Index = 6bits	

Προσπέλαση δεδομένων σε Direct Mapped Cache

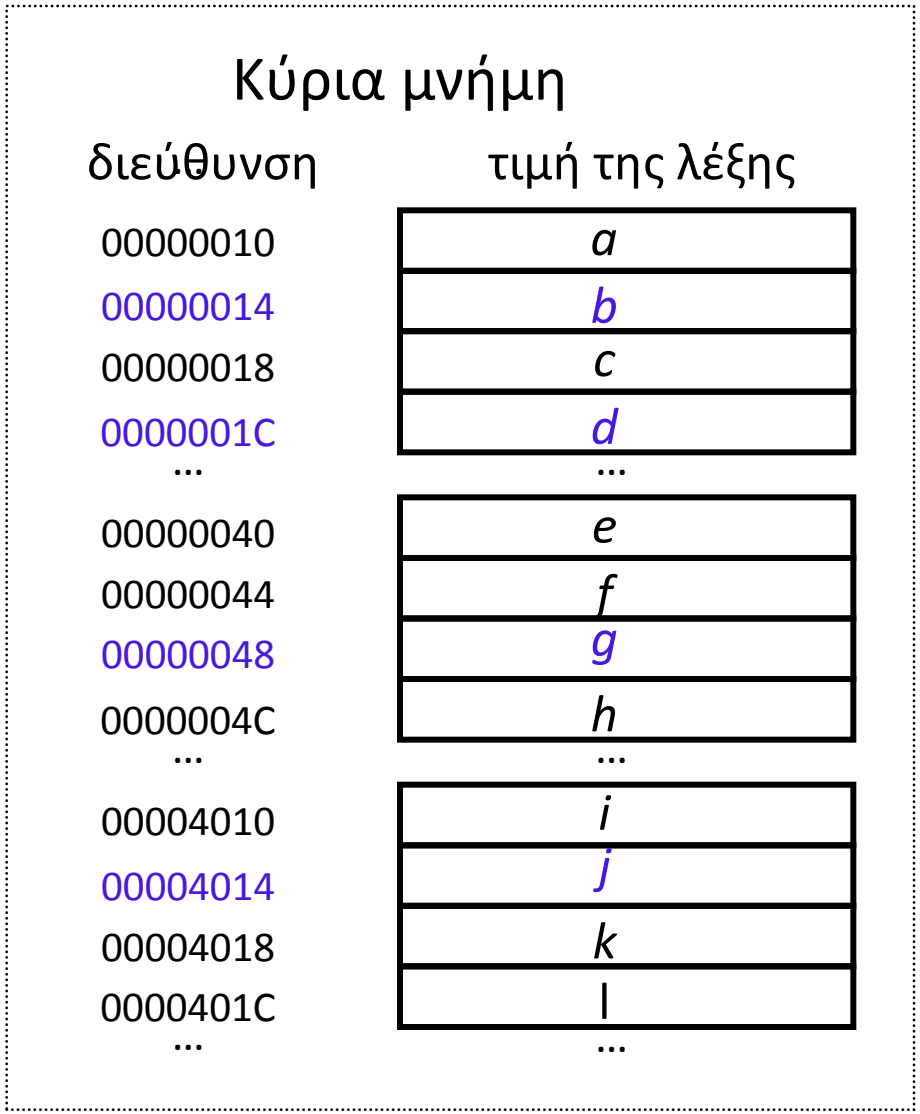
Η CPU καλεί προς ανάγνωση τις εξής διευθύνσεις:

0x00000014

0x00000048

0x0000001C

0x00004014



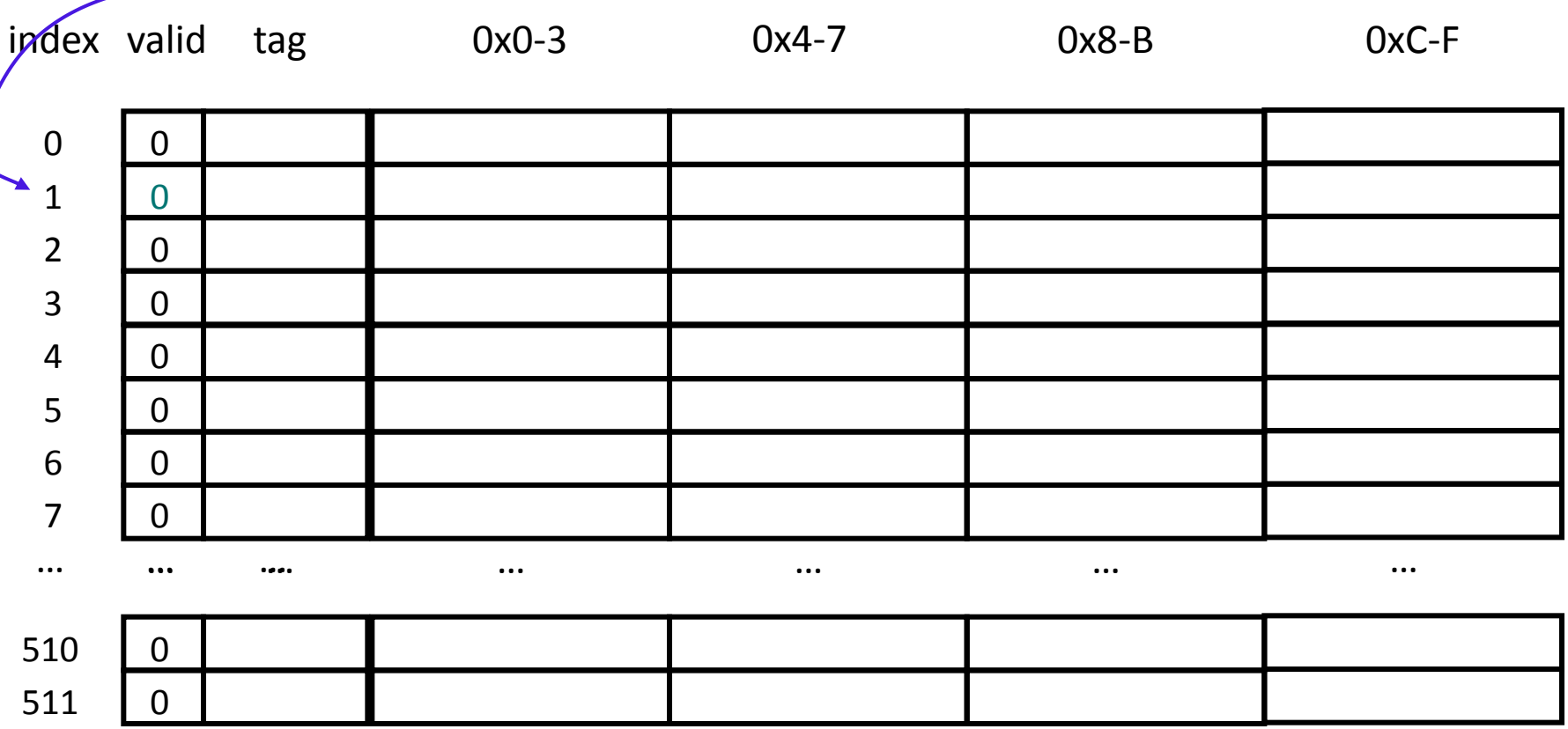
8KB Direct-mapped cache 4W blocks

Αρχικά όλες οι θέσεις invalid

index	valid	tag	0x0-3	0x4-7	0x8-B	0xC-F
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
…	…	…	…	…	…	…
510	0					
511	0					

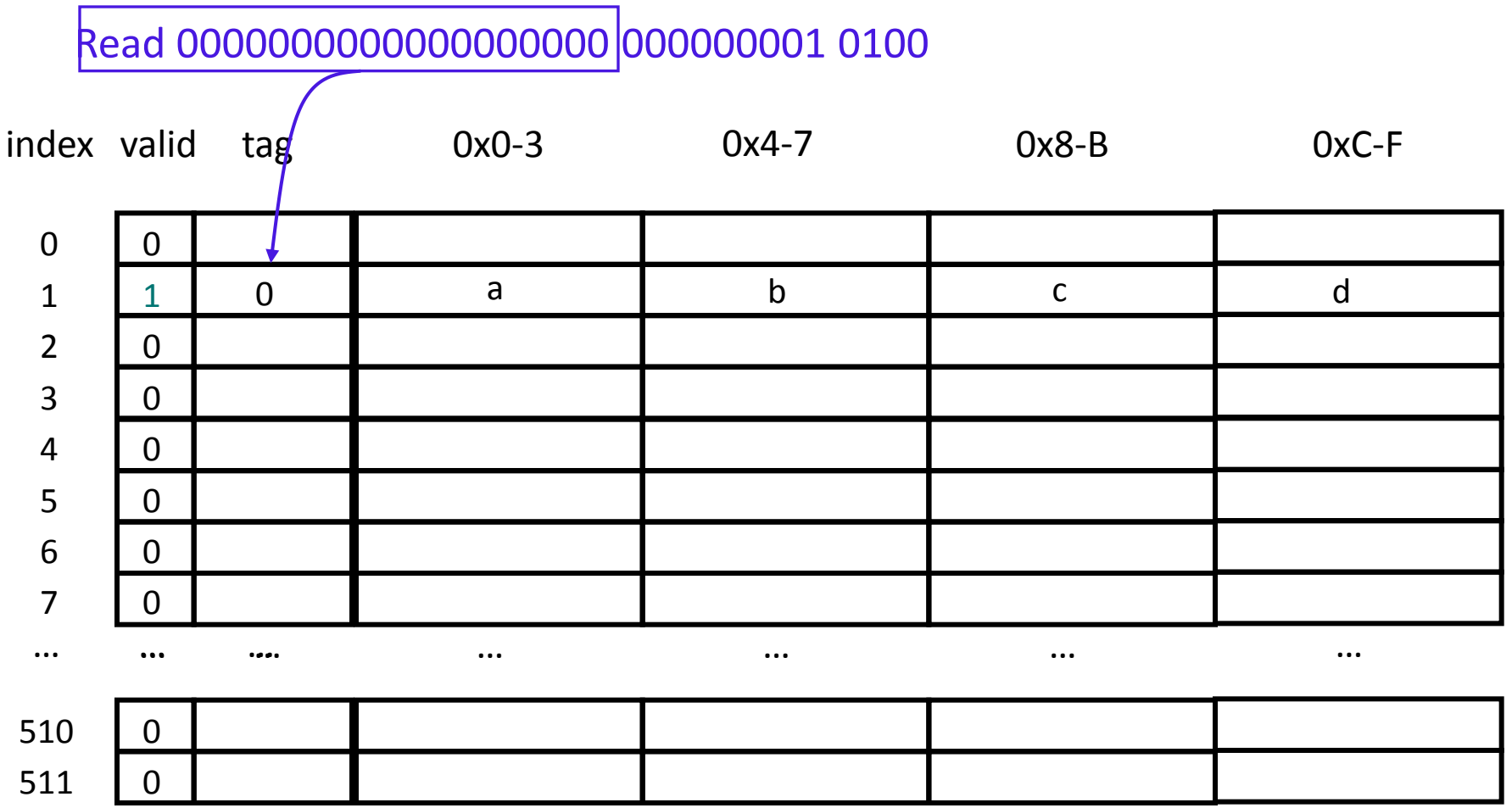
8KB Direct-mapped cache 4W blocks

Read 00000000000000000000 00000001 0100 (0x00000014)



Read block 1 : invalid data στο block 1 !

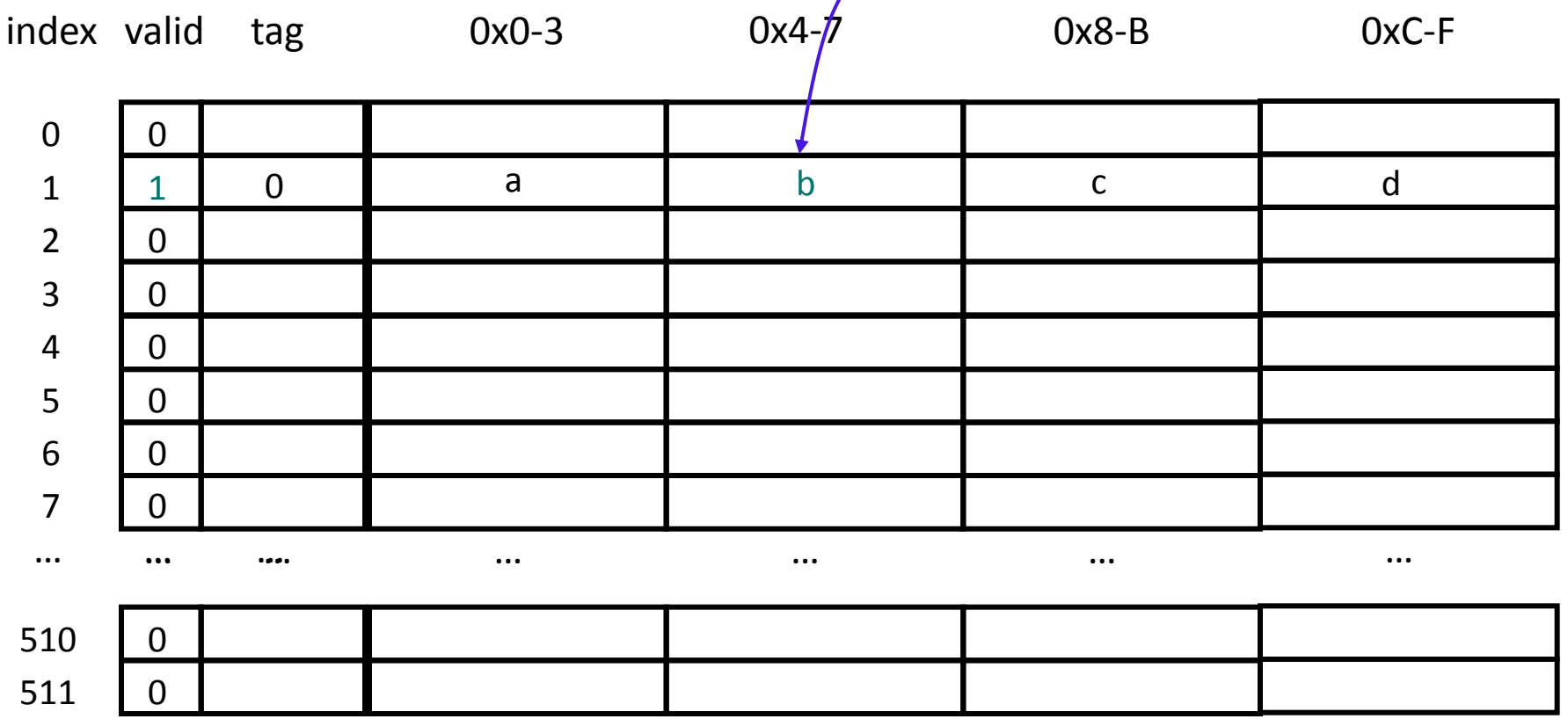
8KB Direct-mapped cache 4W blocks



Φόρτωσε τα ζητούμενα δεδομένα στην cache !

8KB Direct-mapped cache 4W blocks

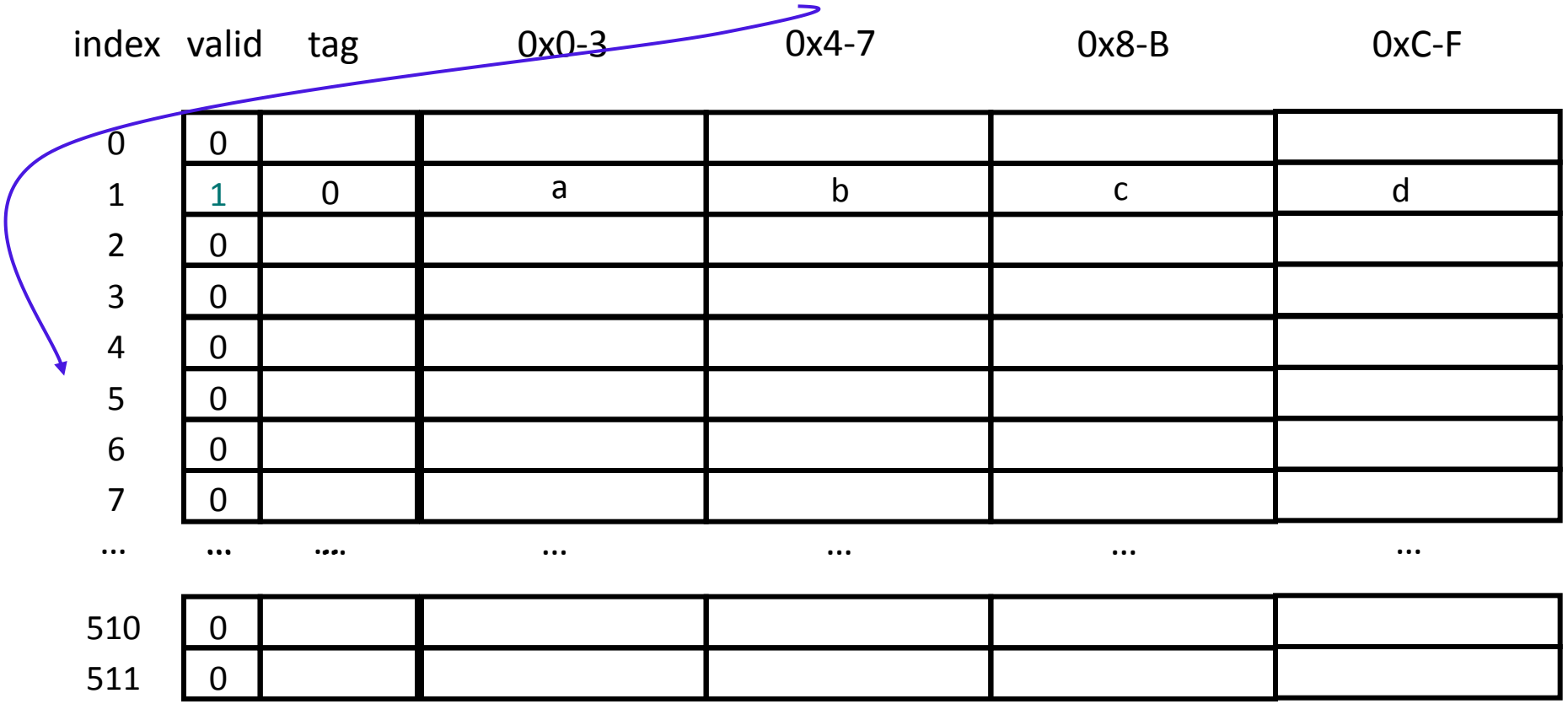
Read 00000000000000000000 00000001 0100



Επέστρεψε το b (θέση 0100) στην CPU

8KB Direct-mapped cache 4W blocks

Read 00000000000000000000 000000100 1000 (0x00000048)



Read block 4 : invalid data στο block 4 !

8KB Direct-mapped cache 4W blocks

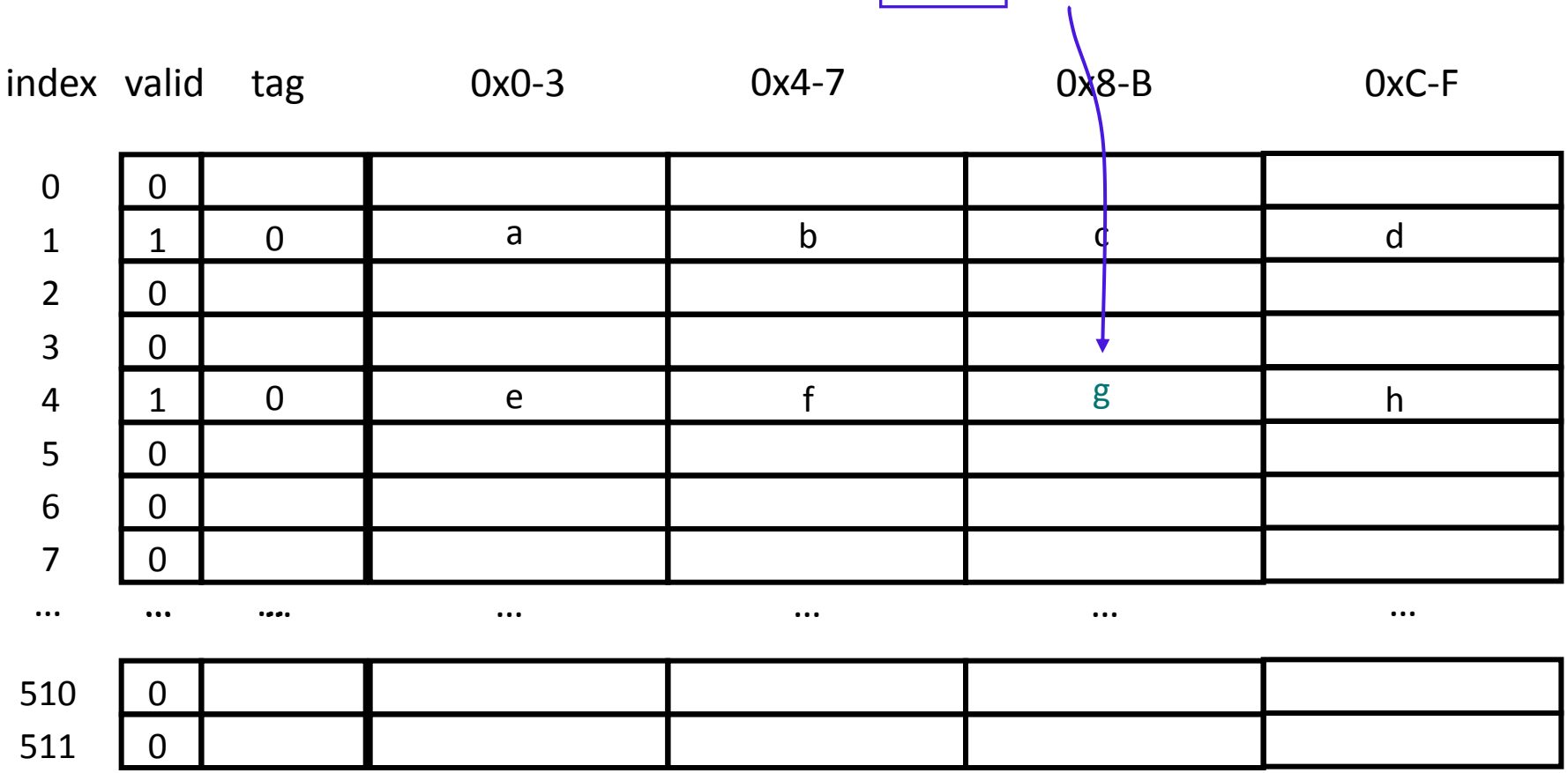
Read 00000000000000000000 000000100 1000

index	valid	tag	0x0-3	0x4-7	0x8-B	0xC-F
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	1	0	e	f	g	h
5	0					
6	0					
7	0					
...	...	....	...	...	...	...
510	0					
511	0					

Φόρτωσε τα ζητούμενα δεδομένα στην cache και κάνε το block valid !

8KB Direct-mapped cache 4W blocks

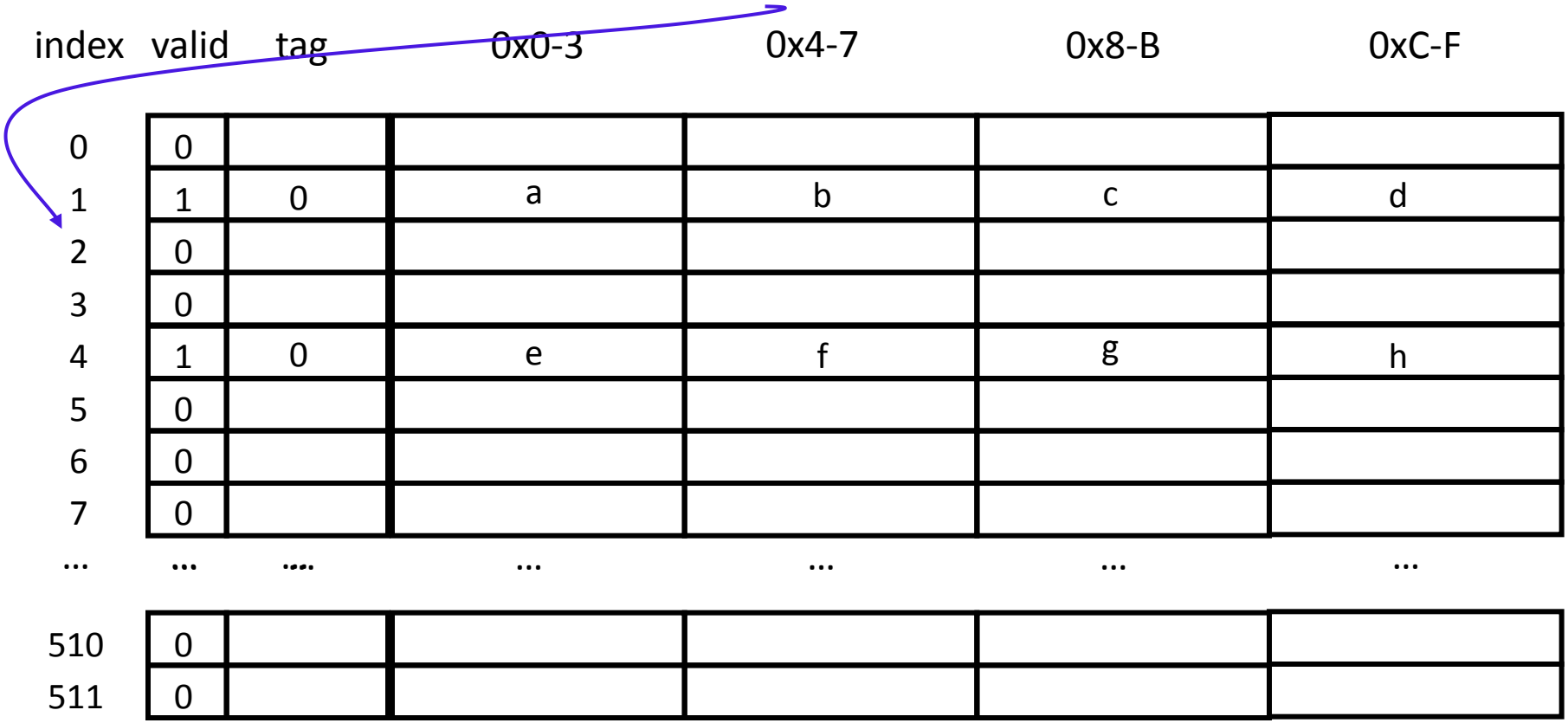
Read 00000000000000000000 000000100 1000



Επέστρεψε στην CPU την τιμή g !

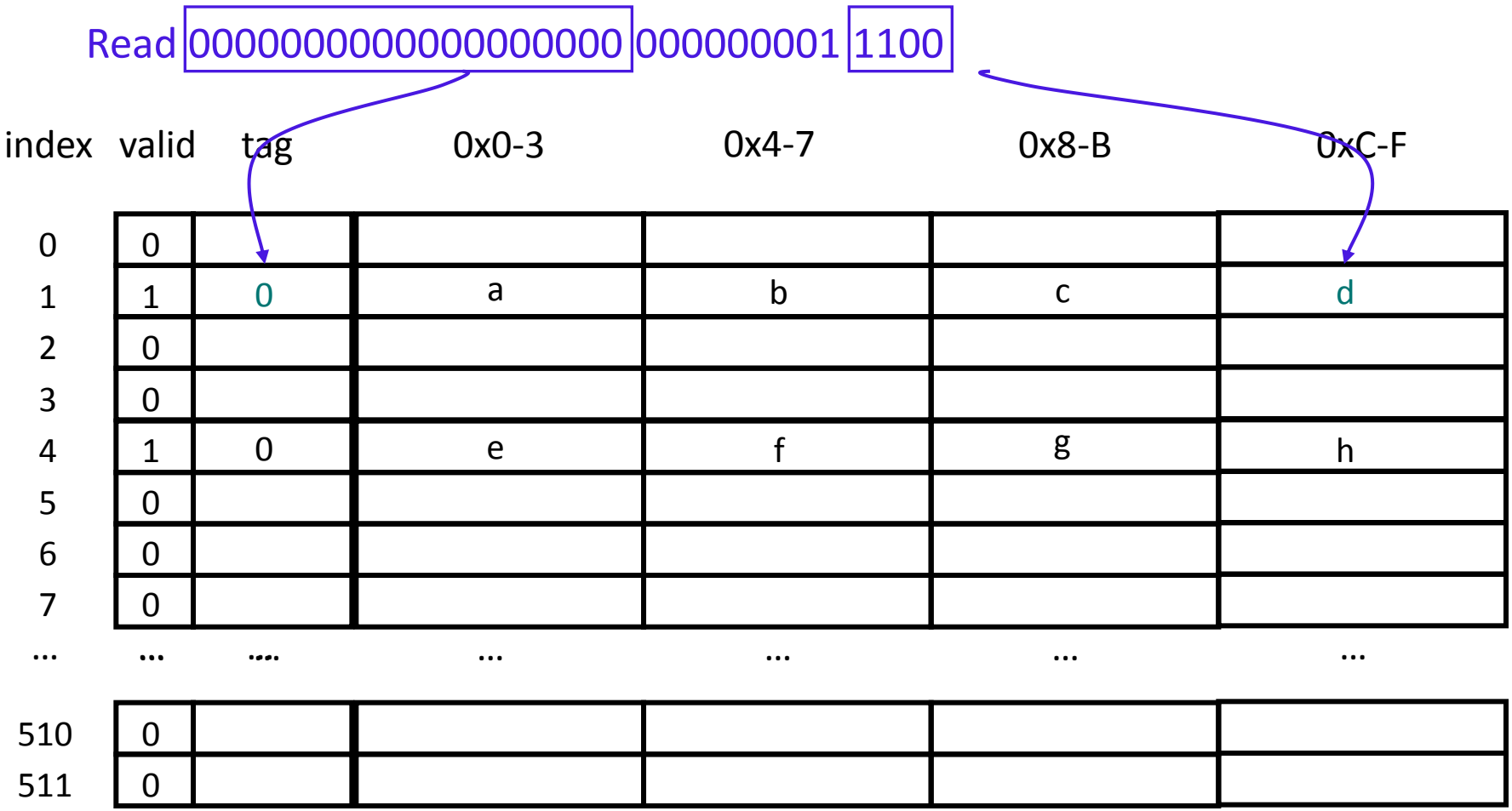
8KB Direct-mapped cache 4W blocks

Read 00000000000000000 000000001 1100 (0x0000001C)



Read block 1 !

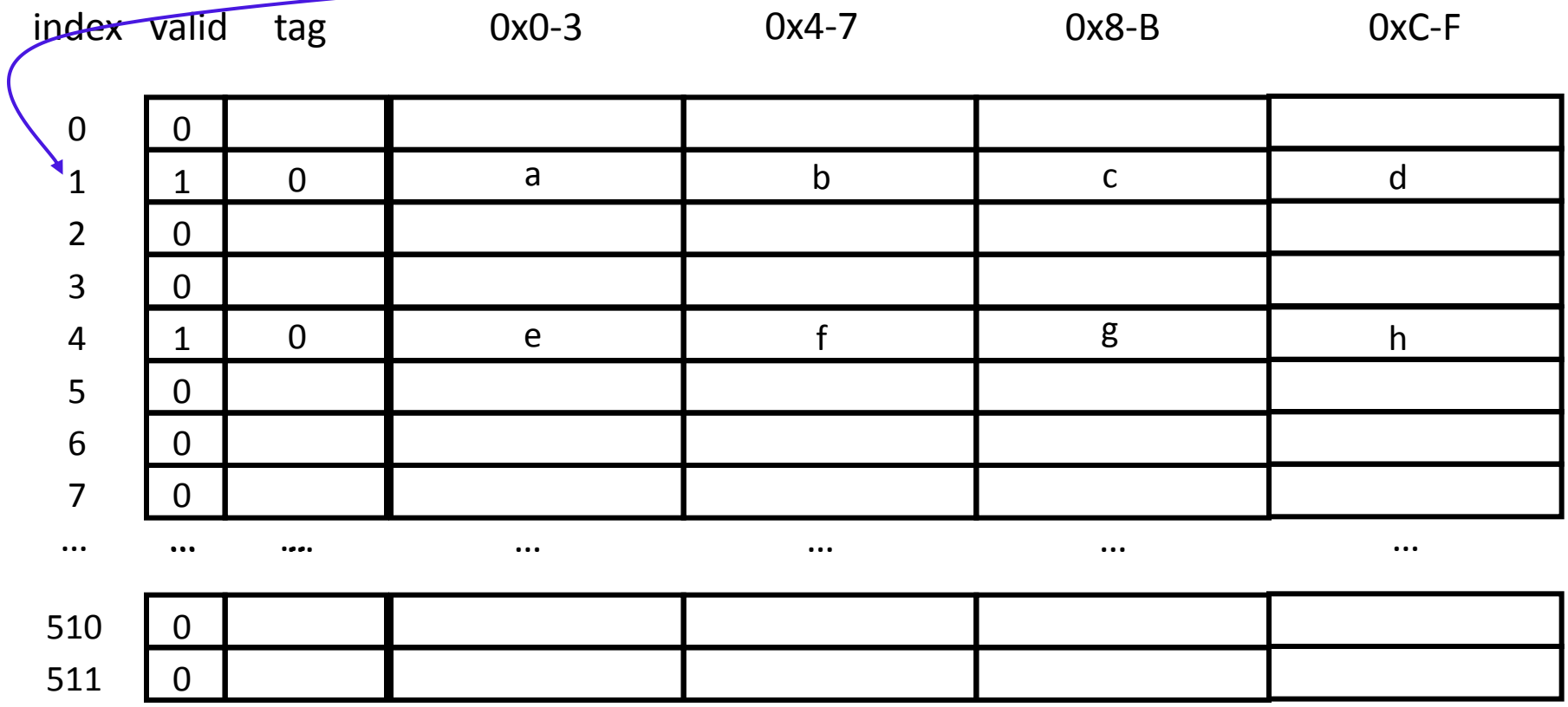
8KB Direct-mapped cache 4W blocks



Το πεδίο tag έχει τη σωστή τιμή ! Άρα επιστρέφεται η τιμή d

8KB Direct-mapped cache 4W blocks

Read 00000000000000010 000000001 0100 (0x00004014)



index	valid	tag	0x0-3	0x4-7	0x8-B	0xC-F
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	1	0	e	f	g	h
5	0					
6	0					
7	0					
...	...	....	...	...	...	...
510	0					
511	0					

Read block 1 !

8KB Direct-mapped cache 4W blocks

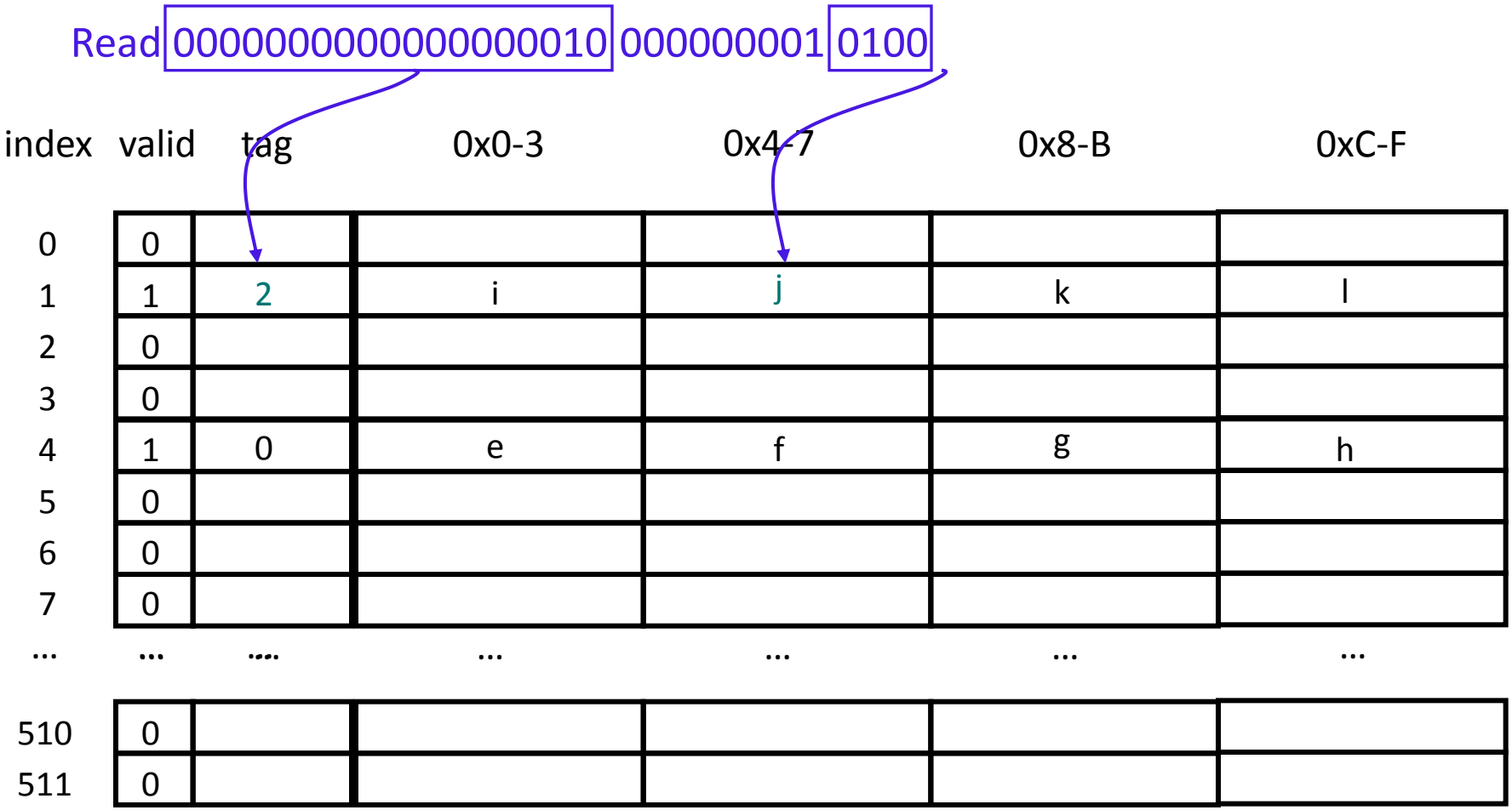
Read 00000000000000010 000000001 0100

index	valid	tag	0x0-3	0x4-7	0x8-B	0xC-F
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	1	0	e	f	g	h
5	0					
6	0					
7	0					
…	…	…	…	…	…	…
510	0					
511	0					

Valid data αλλά το πεδίο tag δεν είναι το σωστό 2!=0

Miss : πρέπει να αντικατασταθεί το block 1 με νέα δεδομένα

8KB Direct-mapped cache 4W blocks



Φόρτωσε το σωστό περιεχόμενο και στείλε το j στην CPU

Υπολογισμός του αριθμού των bits που χρειάζονται

- Πόσα bits συνολικά χρειάζονται σε μία direct-mapped cache με 64 KBytes data και blocks της 1 λέξης, για 32-bit διευθύνσεις;
  - 64 Kbytes = 16 Kwords = 2^{14} words = 2^{14} blocks
  - Block size = 4 bytes => offset size = 2 bits,
  - #sets = #blocks = 2^{14} => index size = 14 bits
  - Tag size = address size - index size - offset size = 32 -14 - 2 =16 bits
  - Bits/block = data bits + tag bits + valid bit = 32 + 16 + 1 = 49
  - Bits της cache = #blocks x bits/block = 2^{14} x 49 = 98 Kbytes
- Πόσα bits συνολικά χρειάζονται σε μία 4-way set associative cache για την αποθήκευση των ίδιων δεδομένων;
  - Block size και #blocks δεν αλλάζει.
  - #sets = #blocks/4 = (2^{14})/4 = 2^{12} => index size = 12 bits
  - Tag size = address size - index size - offset = 32 - 12 - 2 = 18 bits
  - Bits/block = data bits + tag bits + valid bit = 32 + 18 + 1 = 51
  - Bits της cache = #blocks x bits/block = 2^{14} x 51 = 102 Kbytes
- Αύξηση του associativity => Αύξηση των bits της cache

Υπολογισμός του αριθμού των bits που χρειάζονται

- Πόσα bits συνολικά χρειάζονται σε μία direct- mapped cache με 64KBytes data και blocks των 8 λέξεων, για 32-bit διευθύνσεις (2^{32} bytes μπορούν να αποθηκευθούν στη μνήμη);
  - 64 Kbytes = 2^{14} words = $(2^{14})/8 = 2^{11}$ blocks
  - block size = 32 bytes
    => offset size = block offset + byte offset = 5 bits
  - #sets = #blocks = 2^{11} => index size = 11 bits
  - tag size = address size - index size - offset size = 32 - 11 - 5 = 16 bits
  - bits/block = data bits + tag bits + valid bit = 8 x 32 + 16 + 1 = 273 bits
  - bits in cache = #blocks x bits/block = 2^{11} x 273 = 68.25 Kbytes

- Αύξηση του μεγέθους του block => Μείωση των bits της cache.

Μηχανισμοί αντικατάστασης ενός block της cache

- **Random** (τυχαία) – επιλογή ενός τυχαίου block με βάση κάποια ψευδοτυχαία ακολουθία
  - απλή υλοποίηση στο hardware
- **LRU** (least recently used) – αντικαθιστάται το block που δεν έχει χρησιμοποιηθεί για περισσότερη ώρα
  - ακριβή υλοποίηση στο hardware
  - είναι η τεχνική που χρησιμοποιείται συνήθως (ή προσεγγίσεις)
- **FIFO** (first in - first out) - αντικαθιστάται το block που έχει εισαχθεί πρώτο στην cache

Μηχανισμοί εγγραφής σε block (σε περίπτωση write hit/miss)

- Σε περίπτωση write-hit, γνωστοποιείται η αλλαγή στην κύρια μνήμη ;

  ναι : **write-through**

  όχι : **write-back**

- Σε περίπτωση write-miss, τοποθετείται το block στην cache;

  ναι : **write-allocate** (συνήθως με write-back)

  όχι : **write-no-allocate** (συνήθως με write-through)

Write Policies: Write-Back & Write-Through

- **write-back**: ενημέρωση της μνήμης μόνο κατά την απομάκρυνση του block από την cache
  - οι εγγραφές πραγματοποιούνται με την ταχύτητα της cache
  - dirty bit κατά την τροποποίηση – αντικατάσταση των “clean” block χωρίς ενημέρωση της μνήμης
    - Χαμηλό ποσοστό misses
    - Πολλές εγγραφές ενός block σε μία ενημέρωση
- **write-through**: ενημέρωση της μνήμης σε κάθε εγγραφή
  - το κατώτερο ιεραρχικά επίπεδο περιέχει τα εγκυρότερα δεδομένα
  - εύκολη υλοποίηση (εξασφάλιση data coherency)
  - αυξημένη μετακίνηση δεδομένων προς τη μνήμη
  - συχνά χρησιμοποιείται ένας write buffer για αποφυγή καθυστερήσεων όσο ενημερώνεται η μνήμη

Write Through vs Write Back

- **Write Through** - the information is written to both the block in the cache and to the block in the lower-level memory.

Pros	Cons
- read miss never results in writes to main memory - easy to implement - main memory always has the most current copy of the data (consistent)	- write is slower - every write needs a main memory access - as a result uses more memory bandwidth

- **Write Back** - the information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced. To reduce the frequency of writing back blocks on replacement, a dirty bit is commonly used. This status bit indicates whether the block is dirty (modified while in the cache) or clean (not modified). A clean block is not written on a miss.

Pros	Cons
- writes occur at the speed of the cache memory - multiple writes within a block require only one write to main memory - as a result uses less memory bandwidth	-harder to implement - main memory is not always consistent with cache - reads that result in replacement may cause writes of dirty blocks to main memory

Write-Allocate & Write-no-allocate (στη περίπτωση write-miss)

- **Write-allocate**: το block ενημερώνεται στη μνήμη και μετά μεταφέρεται από τη μνήμη στη cache
- **Write-no-allocate**: το block ενημερώνεται στη μνήμη και δεν μεταφέρεται στη cache

"Allocate" a cache line to store the memory block !

Read hit / miss

- **read hit** : ανάγνωση των δεδομένων από την cache
- **read miss** : μεταφορά ολόκληρου του block που περιέχει τα δεδομένα που αναζητάμε στην cache και στη συνέχεια όπως στο read hit

Write hit / miss

- **Write-back & Write-allocate**
  - **write hit**:
    - Εγγραφή των δεδομένων στην cache (μόνο).
    - Το block είναι dirty
    - Η κύρια μνήμη ενημερώνεται μόνο όταν απομακρυνθεί το block από την cache
  - **write miss**:

    Το block:
    - Ενημερώνεται στη μνήμη
    - μεταφέρεται στην cache
    - Στη συνέχεια όπως στο write hit

Write hit / miss

- **Write-through & write-no-allocate**
  - **write hit**:
    - Εγγραφή των νέων δεδομένων στην cache
    - Ενημέρωση της κύρια μνήμης
  - **write miss**:
    - Εγγραφή μόνο στην κύρια μνήμη
    - Δεν εμπλέκεται καθόλου η cache

Write Back with Write Allocate:

- on hits it writes to cache setting “dirty” bit for the block, main memory is not updated;
- on misses it updates the block in main memory and brings the block to the cache;
- Subsequent writes to the same block, if the block originally caused a miss, will hit in the cache next time, setting dirty bit for the block. That will eliminate extra memory accesses and result in very efficient execution compared with Write Through with Write Allocate combination.

Write Back with No Write Allocate:

- on hits it writes to cache setting “dirty” bit for the block, main memory is not updated;
- on misses it updates the block in main memory not bringing that block to the cache;
- Subsequent writes to the same block, if the block originally caused a miss, will generate misses all the way and result in very inefficient execution.

Write Through with Write Allocate:

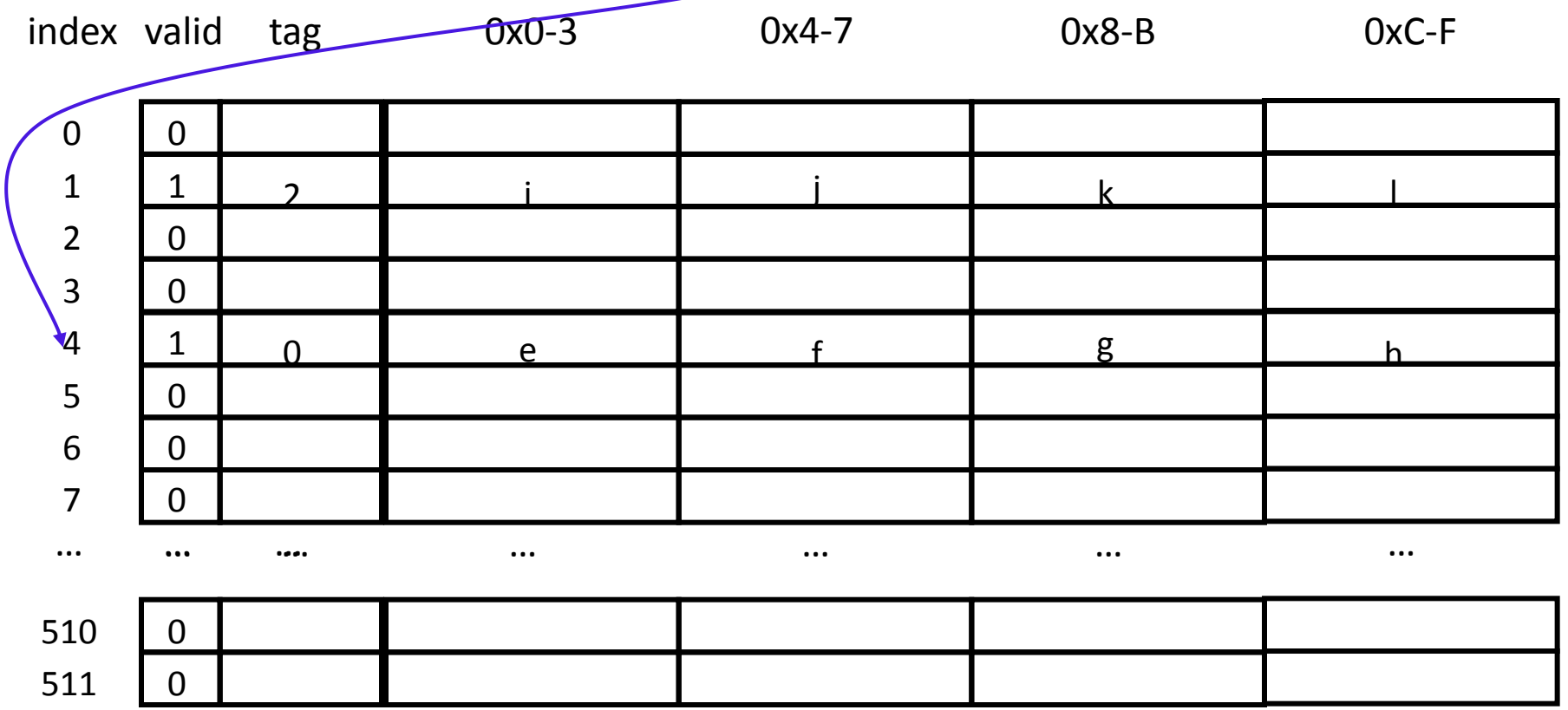
- on hits it writes to cache and main memory
- on misses it updates the block in main memory and brings the block to the cache
- Bringing the block to cache on a miss does not make a lot of sense in this combination because the next hit to this block will generate a write to main memory anyway (according to Write Through policy)

Write Through with No Write Allocate:

- on hits it writes to cache and main memory;
- on misses it updates the block in main memory not bringing that block to the cache;
- Subsequent writes to the block will update main memory because Write Through policy is employed. So, some time is saved not bringing the block in the cache on a miss because it appears useless anyway.

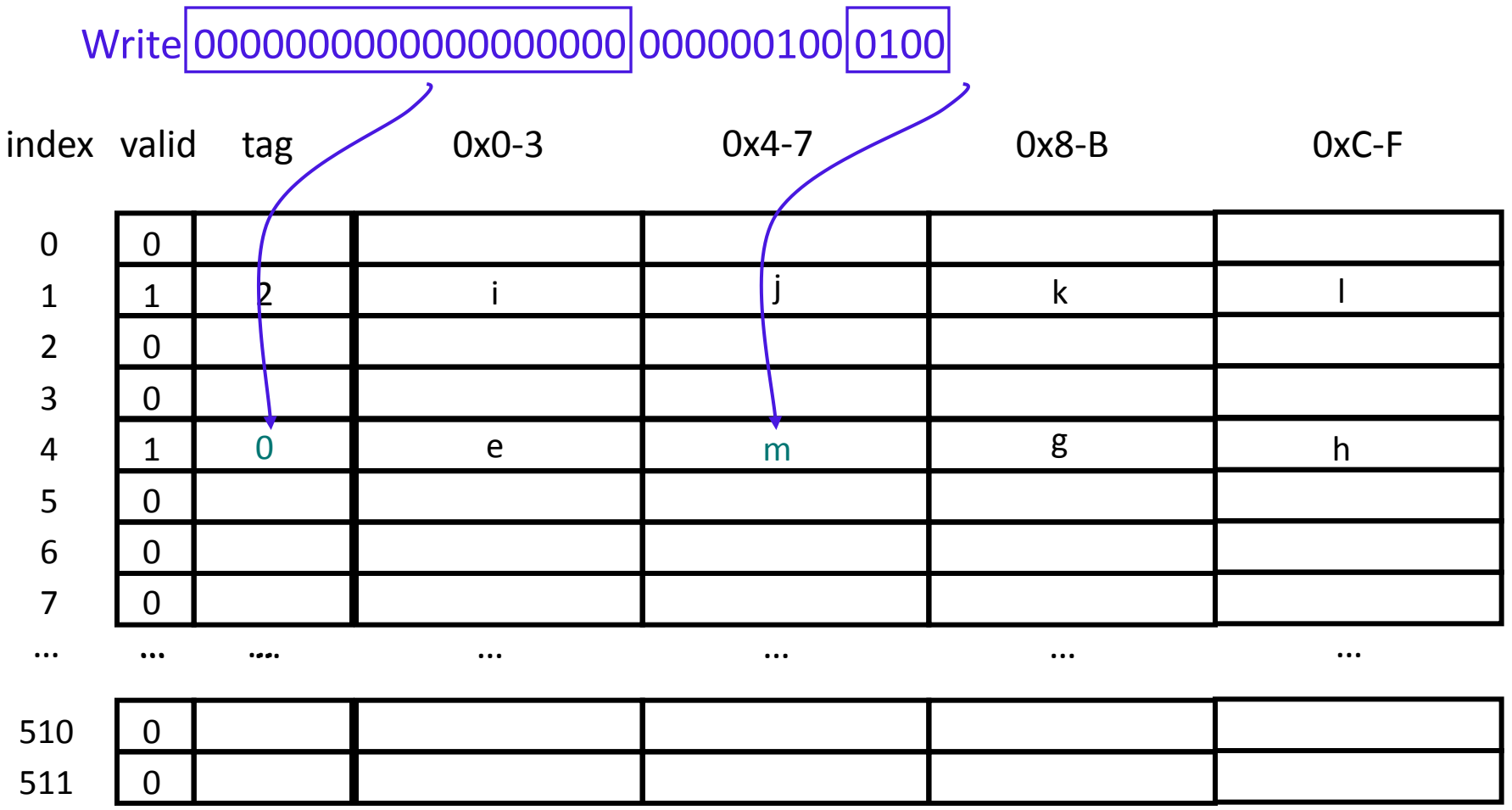
8KB Direct-mapped cache - 4W blocks write through

Write 00000000000000000000 000000100 0100 (0x00000044) , m



Read block 4 !

8KB Direct-mapped cache - 4W blocks write through



Valid data - σωστό tag – εγγραφή στο πεδίο 0100 της cache και ενημέρωση της κύριας μνήμης !

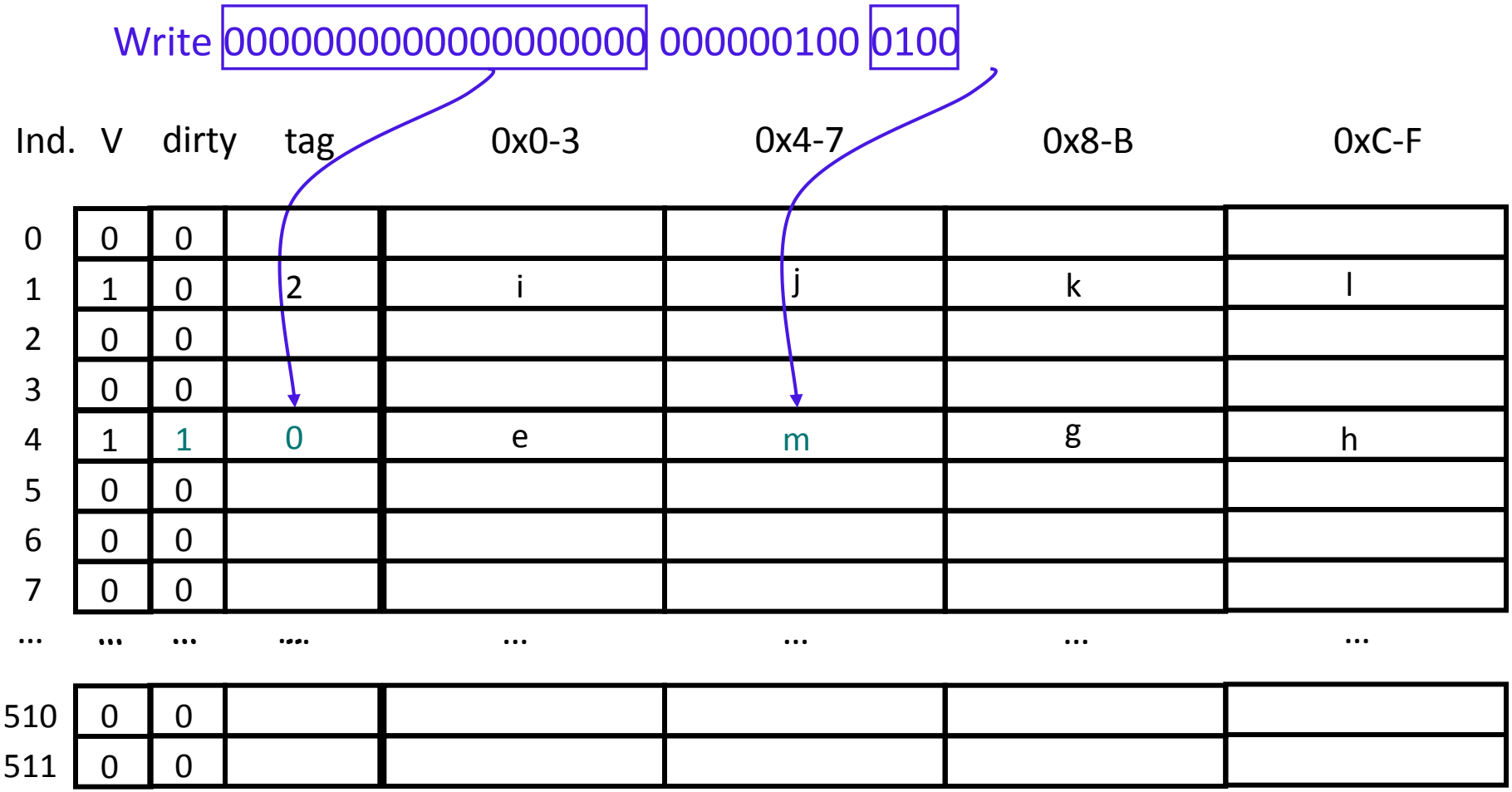
8KB Direct-mapped cache - 4W blocks write back

Write 00000000000000000000 000000100 0100 (0x00000044) , m

Ind.	V	dirty	tag	0x0-3	0x4-7	0x8-B	0xC-F
0	0	0					
1	1	0	2	i	j	k	l
2	0	0					
3	0	0					
4	1	0	0	e	f	g	h
5	0	0					
6	0	0					
7	0	0					
...	...	...	....	...	...	...	...
510	0	0					
511	0	0					

Read block 4 !

8KB Direct-mapped cache - 4W blocks write back



Valid data - σωστό tag – εγγραφή στο πεδίο 0100 της cache και ενημέρωση του dirty bit !

8KB Direct-mapped cache - 4W blocks write back

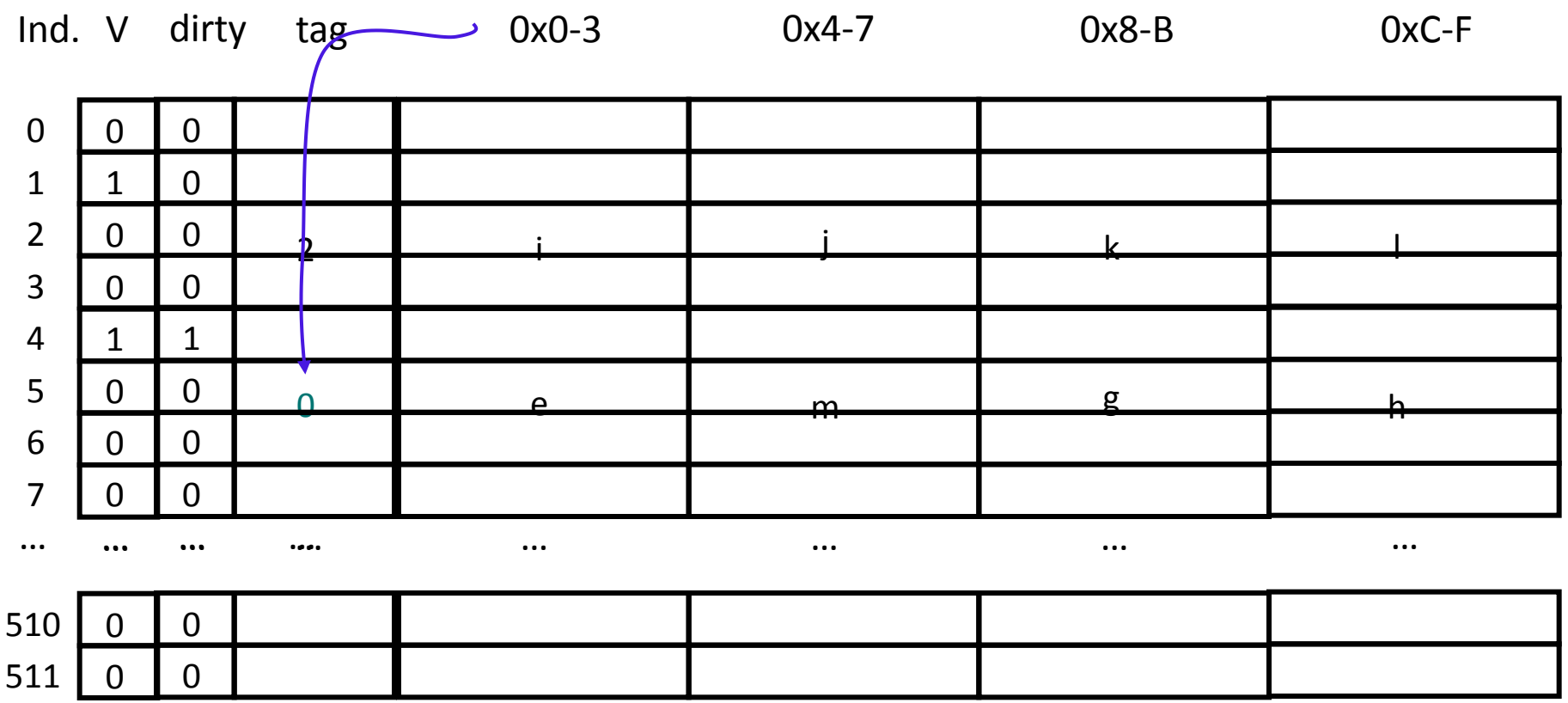
Read 00000000000000000100 000000100 1100 (0x0000804C)

Ind.	V	dirty	tag	0x0-3	0x4-7	0x8-B	0xC-F
0	0	0					
1	1	0	2	i	j	k	l
2	0	0					
3	0	0					
4	1	1	0	e	m	g	h
5	0	0					
6	0	0					
7	0	0					
...	...	...	....	...	...	...	...
510	0	0					
511	0	0					

Read block 4 !

8KB Direct-mapped cache - 4W blocks write back

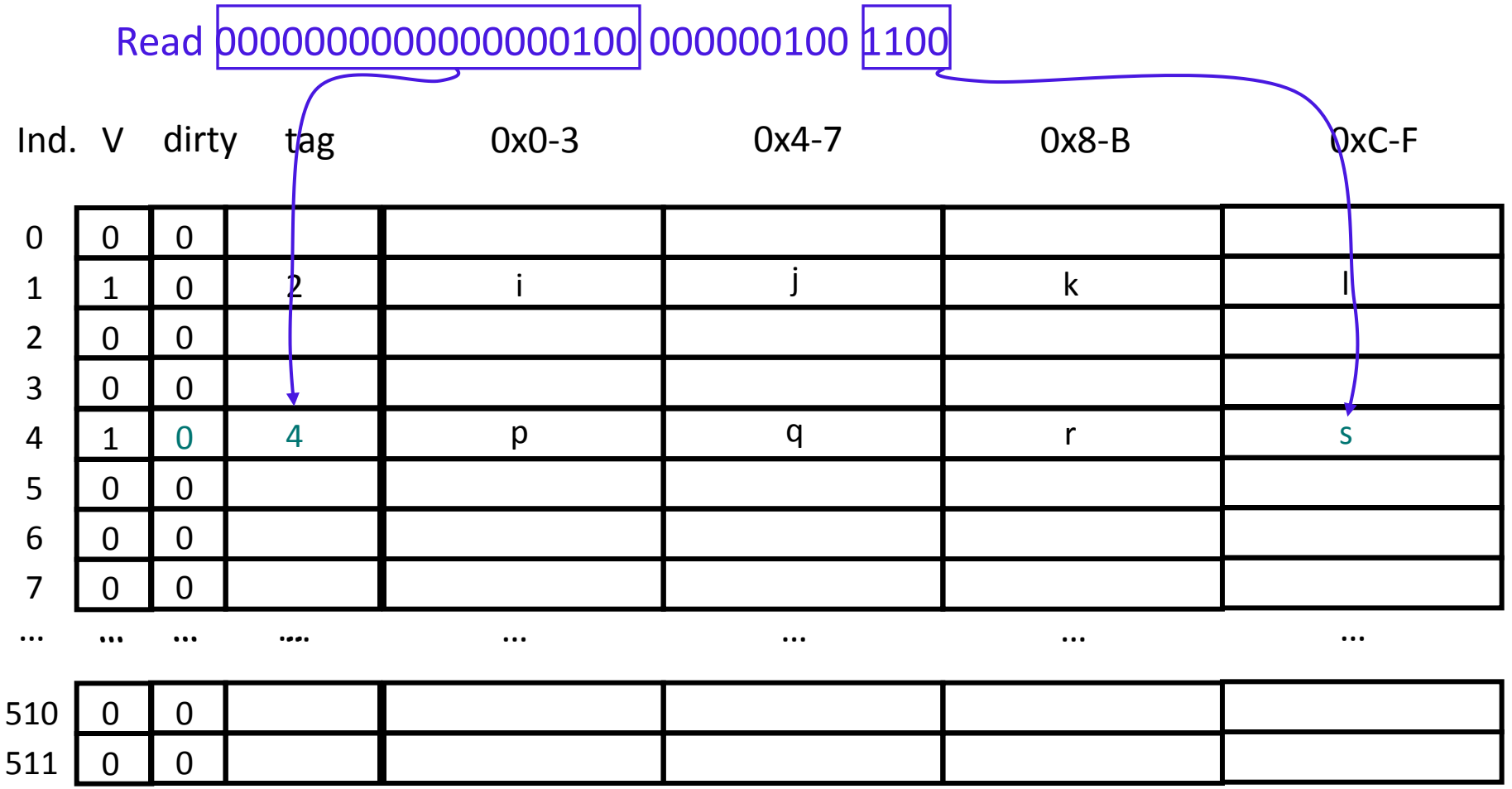
Read 00000000000000100 000000100 1100



Valid data – το πεδίο tag όμως δεν ταιριάζει : 0!=4

Το dirty bit είναι 1 : Ενημερώνεται η μνήμη (0x00000040-0x0000004F) και στη συνέχεια φορτώνεται η σωστή διεύθυνση

8KB Direct-mapped cache - 4W blocks write back



Φορτώνεται η σωστή διεύθυνση - ενημερώνονται τα πεδία tag - dirty

Επιστρέφεται η τιμή r στη CPU

Επίδοση των επιπέδων μνήμης (performance)

- μέσος χρόνος προσπέλασης των δεδομένων (access time)

$$t_{avg} = t_{hit} + miss\ \ rate \cdot t_{miss\ \ penalty}$$

Cache : ενοποιημένη ή όχι;

- Ενοποιημένη για εντολές και δεδομένα (unified)
  - Μικρότερο κατασκευαστικό κόστος
  - Καλύτερο ισοζύγισμα του χώρου που καταλαμβάνεται από εντολές/δεδομένα
  - Επιπλέον misses λόγω διεκδίκησης κοινών θέσεων στην cache (conflict misses)
- Δύο διαφορετικές caches για εντολές και δεδομένα (data cache & instruction cache)
  - 2-πλάσιο εύρος ζώνης
  - όχι conflict misses

Παράδειγμα

- Σε ποια περίπτωση έχουμε καλύτερη επίδοση;

  Σε σύστημα με 16KB instruction cache και 16KB data cache ή σε σύστημα με 32KB unified cache;

  Υποθέτουμε ότι το 36% των εντολών είναι εντολές αναφοράς στη μνήμη (load/store).

  hit time = 1 clock cycle

  miss penalty = 100 clock cycles

  στη unified cache είναι: hit time = 2 clock cycles

Χρησιμοποιείστε τα δεδομένα του ακόλουθου πίνακα (αναφέρονται σε 1000 εντολές):

	Instr.cache	**data cache**	**unified cache**
16KB	3.82	40.9	51.0
32KB	1.36	38.4	43.3

Παράδειγμα (συνέχεια…)

- Λύση

$$miss\ \ rate = \frac{misses}{mem\ \ accesses}$$

$$miss\ \ rate_{16\ KB\ \ instr\ \ cache} = \frac{3.82}{1000} = 0.0038$$

$$miss\ \ rate_{16\ KB\ \ data\ \ cache} = \frac{40.9}{360} = 0.114$$

74%·0.0038 + 26%·0.114 = 0.0324

$$miss\ \ rate_{32\ KB\ \ unif\ \ cache} = \frac{43.3}{1000 + 360} = 0.0318$$

miss rate (unified cache) < miss rate (instr + data cache)

Παράδειγμα (συνέχεια…)

- Λύση

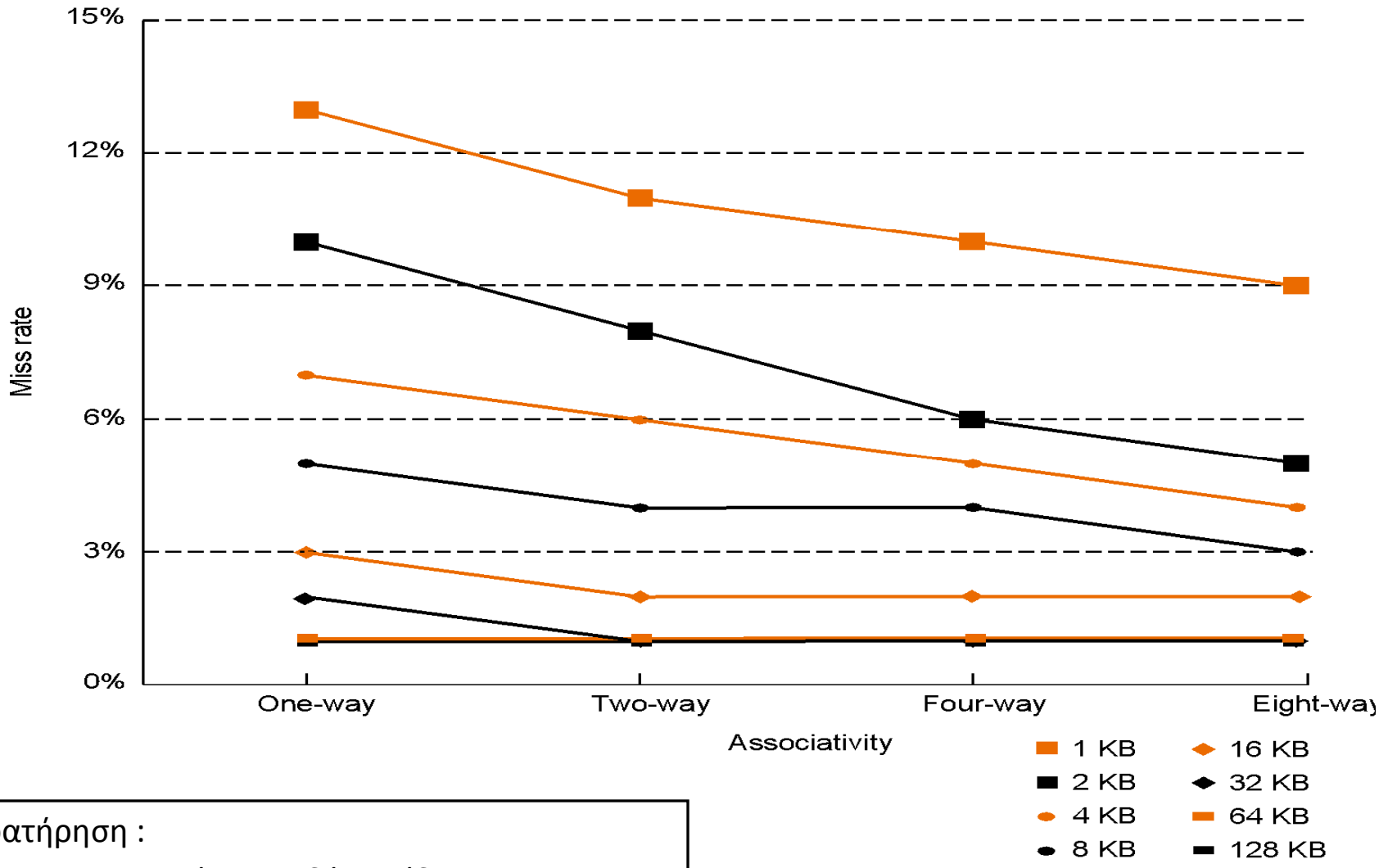
$$t_{avg} = t_{instr} + t_{data} = 74\,\%\,(1 + 0.004 \cdot 100) + 26\,\%\,(1 + 0.114 \cdot 100)$$

$$= 4.26$$

$$t_{avg} = 2 + 0.0318 \cdot 100 = 5.18$$

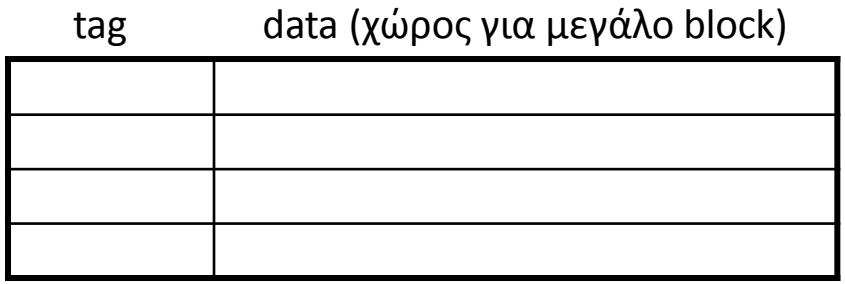
μέσος χρόνος/access (instr+data cache) < μέσος χρόνος/access (unified cache)

Cache Associativity



Παρατήρηση :
Μια 4-way cache έχει σχεδόν το ίδιο hit rate
με μια direct-mapped cache διπλάσιου μεγέθους

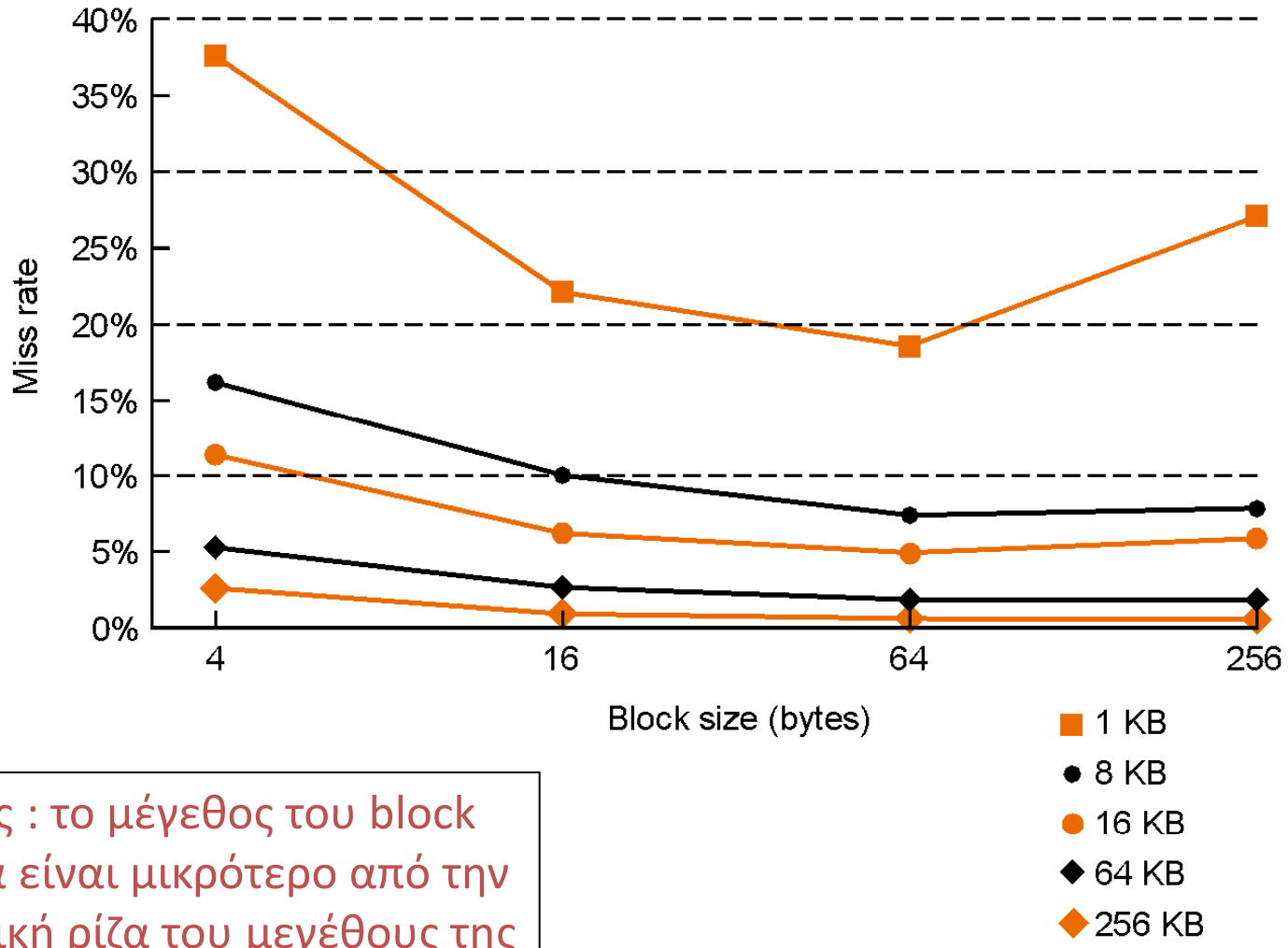
Μέγεθος των Cache Blocks



- Σε μεγάλα cache blocks επωφελούμαστε από την spatial locality.
- Λιγότερος χώρος απαιτείται για tag (με δεδομένη χωρητικότητα της cache)
- Υπερβολικά μεγάλο μέγεθος block σπαταλάει το χώρο της cache
- Τα μεγάλα blocks απαιτούν μεγαλύτερο χρόνο μεταφοράς (transfer time).

Ένας καλός σχεδιασμός απαιτεί συμβιβασμούς!

Μέγεθος Block και Miss Rate



Κανόνας : το μέγεθος του block πρέπει να είναι μικρότερο από την τετραγωνική ρίζα του μεγέθους της cache.

Miss Rates

- Caches διαφορετικού μεγέθους, Associativity & αλγορίθμους αντικατάστασης block

Associativity:	2-way		4-way		8-way	
Μέγεθος	LRU	Random	LRU	Random	LRU	Random
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Επίδοση των caches

Για CPU με ένα μόνο επίπεδο (L1) cache και καθόλου καθυστέρηση όταν έχουμε cache hit:

Με ιδανική μνήμη

Χρόνος CPU = (κύκλοι ρολογιού κατά τη λειτουργία της CPU + κύκλοι ρολογιού λόγω καθυστέρησης από προσπέλαση της μνήμης(Mem stalls))
x χρόνος 1 κύκλου ρολογιού

Mem stalls =
(Αναγνώσεις x miss rate αναγνώσεων x miss penalty αναγνώσεων) +
(Εγγραφές x miss rate εγγραφών x miss penalty εγγραφών)

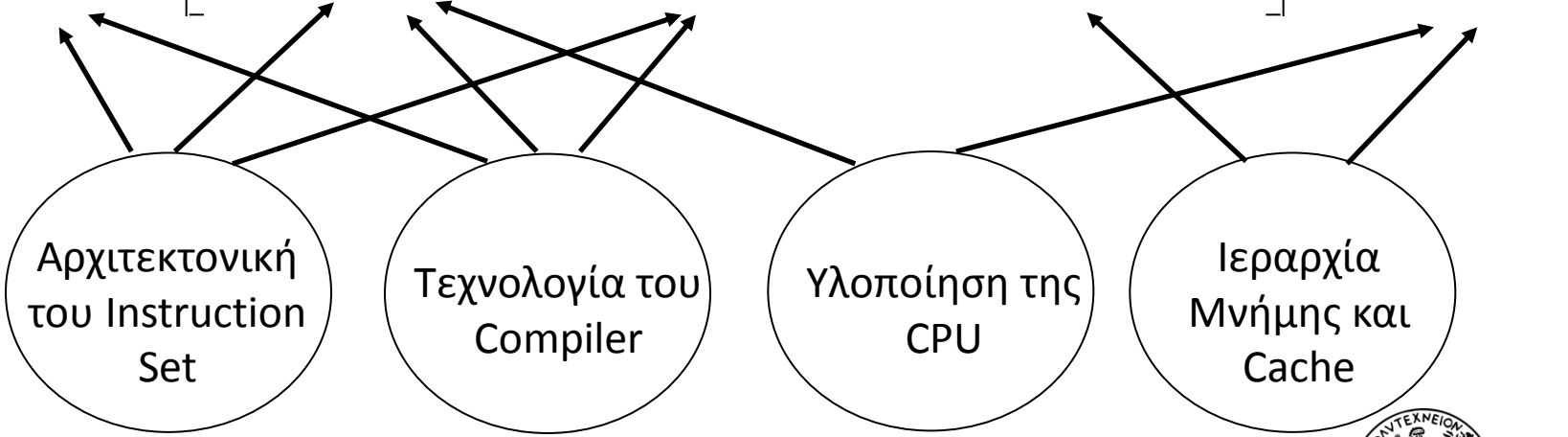
Αν τα miss penalties των αναγνώσεων και των εγγραφών είναι ίδια:
Mem stalls = Προσπελάσεις μνήμης x Miss rate x Miss penalty

Χρόνος εκτέλεσης

$$\text{χρόνος εκτέλεσης} = \text{αριθμός εντολών} \times \frac{\text{κύκλοι}}{\text{εντολή}} \times \text{χρόνος 1 κύκλου}$$

$$= \text{αριθμός εντολών} \times \left(\frac{\text{κύκλοι CPU}}{\text{εντολή}} + \frac{\text{κύκλοι μνήμης}}{\text{εντολή}} \right) \times \text{χρόνος 1 κύκλου}$$

$$= \text{αριθμός εντολών} \times \left[\frac{\text{κύκλοι CPU}}{\text{εντολή}} + \left(\frac{\text{αναφορές}}{\text{εντολή}} \times \frac{\text{κύκλοι μνήμης}}{\text{αναφορά}} \right) \right] \times \text{χρόνος 1 κύκλου}$$



Επίδοση των caches

CPUtime = Instruction count x CPI x Χρόνος 1 κύκλου ρολογιού

$CPI_{execution}$ = CPI με ιδανική μνήμη

CPI = $CPI_{execution}$ + Mem stalls/εντολή

CPUtime = Instruction Count x ($CPI_{execution}$ + Mem stalls/εντολή)
x χρόνος 1 κύκλου ρολογιού

Mem stalls/εντολή =
Προσπελάσεις μνήμης/εντολή x Miss rate x Miss penalty

CPUtime = IC x ($CPI_{execution}$ + Προσπελάσεις μνήμης ανά εντολή x
Miss rate x Miss penalty) x Χρόνος 1 κύκλου ρολογιού

Misses/εντολή = Προσπελάσεις μνήμης ανά εντολή x Miss rate

CPUtime = IC x ($CPI_{execution}$ + Misses/εντολή x Miss penalty)
x Χρόνος 1 κύκλου ρολογιού(C)

Παράδειγμα

- Έστω μία CPU λειτουργεί με ρολόι 200 MHz (5 ns/cycle) και cache ενός επιπέδου.
- $CPI_{execution}$ = 1.1
- Εντολές: 50% arith/logic, 30% load/store, 20% control
- Υποθέτουμε cache miss rate = 1.5% και miss penalty = 50 cycles.

$$CPI = CPI_{execution} + \text{Mem stalls/εντολή}$$

Mem Stalls/εντολή = Mem accesses /εντολή x Miss rate x Miss penalty

Mem accesses /εντολή = 1 + 0.3 = 1.3

Instruction fetch **Load/store**

Mem Stalls /εντολή = 1.3 x 0.015 x 50 = 0.975

CPI = 1.1 + 0.975 = 2.075

Η ιδανική CPU χωρίς misses είναι 2.075/1.1 = 1.88 φορές γρηγορότερη

Παράδειγμα

- Στο προηγούμενο παράδειγμα υποθέτουμε ότι διπλασιάζουμε τη συχνότητα του ρολογιού στα 400 MHZ. Πόσο γρηγορότερο είναι το μηχάνημα για ίδιο miss rate και αναλογία εντολών;

Δεδομένου ότι η ταχύτητα της μνήμης δεν αλλάζει, το miss penalty καταναλώνει περισσότερους κύκλους CPU:

Miss penalty = 50 x 2 = 100 cycles.

CPI = 1.1 + 1.3 x .015 x 100 = 1.1 + 1.95 = 3.05

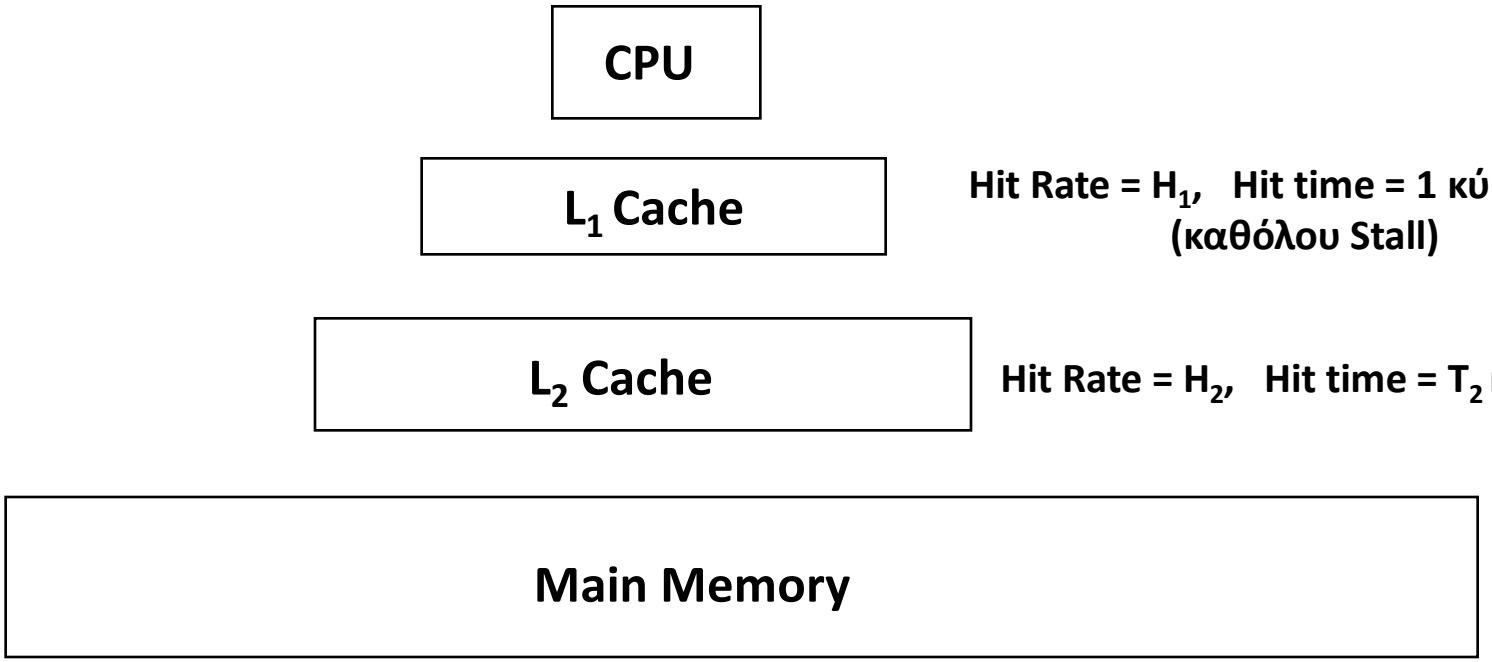
Speedup = $(CPI_{old} \times C_{old}) / (CPI_{new} \times C_{new})$

= 2.075 x 2 / 3.05 = 1.36

Το νέο μηχάνημα είναι μόνο 1.36 φορές ταχύτερο και όχι 2 φορές γρηγορότερο λόγω της επιπλέον επιβάρυνσης των cache misses.

→ *CPUs με μεγαλύτερη συχνότητα ρολογιού, έχουν περισσότερους κύκλους/cache miss και μεγαλύτερη επιβάρυνση της μνήμης στο CPI.*

2 επίπεδα Cache: L_1, L_2



Cache 2 επιπέδων

CPUtime = IC x ($CPI_{execution}$ + Mem Stalls/εντολή) x C

Mem Stalls/εντολή = Mem accesses/εντολή x Stalls/access

- Για ένα σύστημα με 2 επίπεδα cache, χωρίς penalty όταν τα δεδομένα βρεθούν στην L_1 cache:

Stalls/memory access =

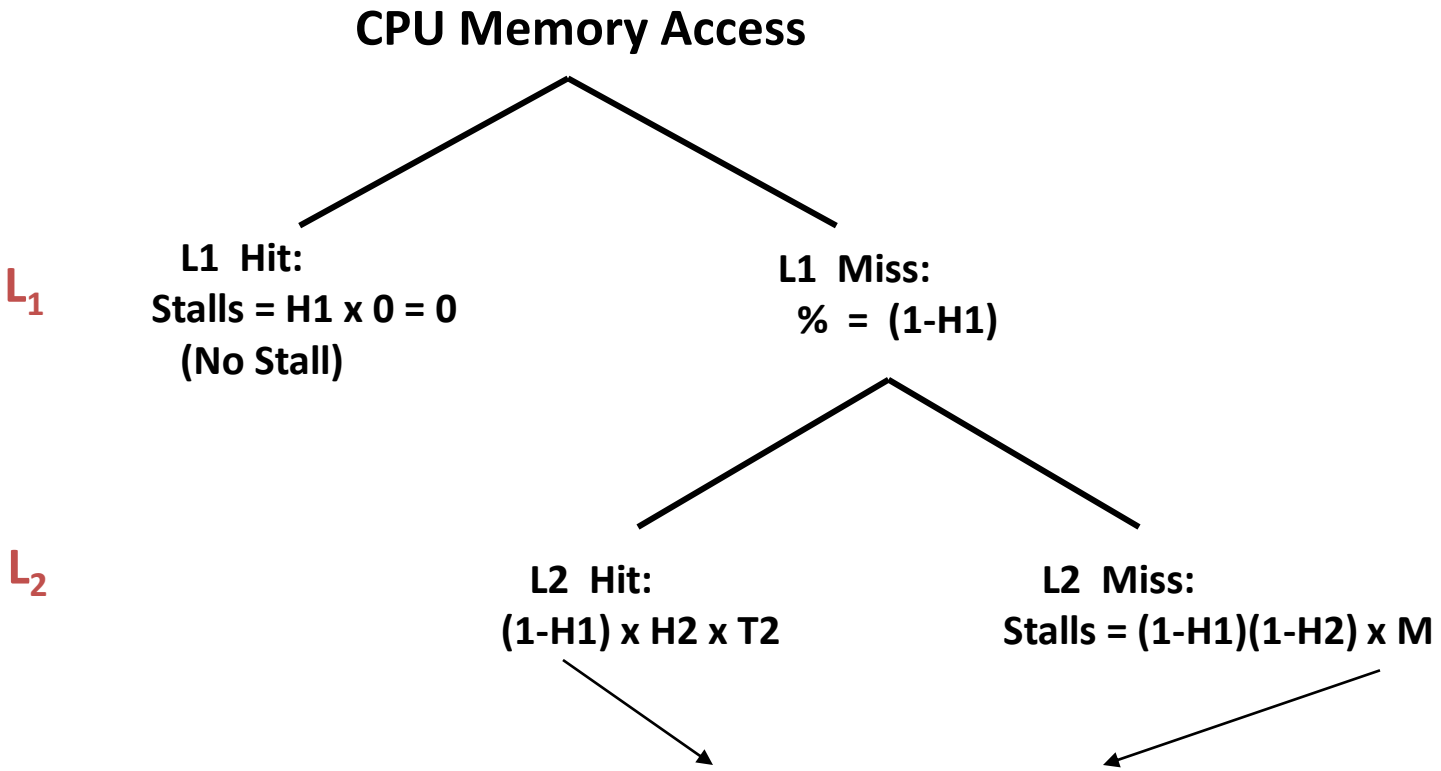
[miss rate L_1] x [Hit rate L_2 x Hit time L_2
+ Miss rate L_2 x Memory access penalty)]

= (1-H1) x H2 x T2 + (1-H1)(1-H2) x M

L1 Miss, L2 Hit

L1 Miss, L2 Miss:
Προσπέλαση της Main Memory

Επίδοση της L2 Cache Memory Access Tree (CPU Stalls/Memory Access)



Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1)(1-H2) x M

Παράδειγμα L2 Cache

- CPU με $CPI_{execution}$ = 1.1 και συχνότητα 500 MHZ
- 1.3 memory accesses/εντολή.
- L_1 cache : στα 500 MHZ με miss rate 5%
- L_2 cache : στα 250 MHZ με miss rate 3%, (T_2 = 2 κύκλοι)
- M (Memory access penalty) = 100 κύκλοι. Να βρεθεί το CPI.

CPI = $CPI_{execution}$ + Mem Stalls/εντολή

Χωρίς Cache, CPI = 1.1 + 1.3 x 100 = 131.1

Με L_1 Cache, CPI = 1.1 + 1.3 x 0.05 x 100 = 7.6

Mem Stalls/εντολή = Mem accesses/εντολή x Stalls/access

Stalls/memory access = (1-H1) x H2 x T2 + (1-H1)(1-H2) x M

= 0.05 x 0.97 x 2 + 0.05 x 0.03 x 100

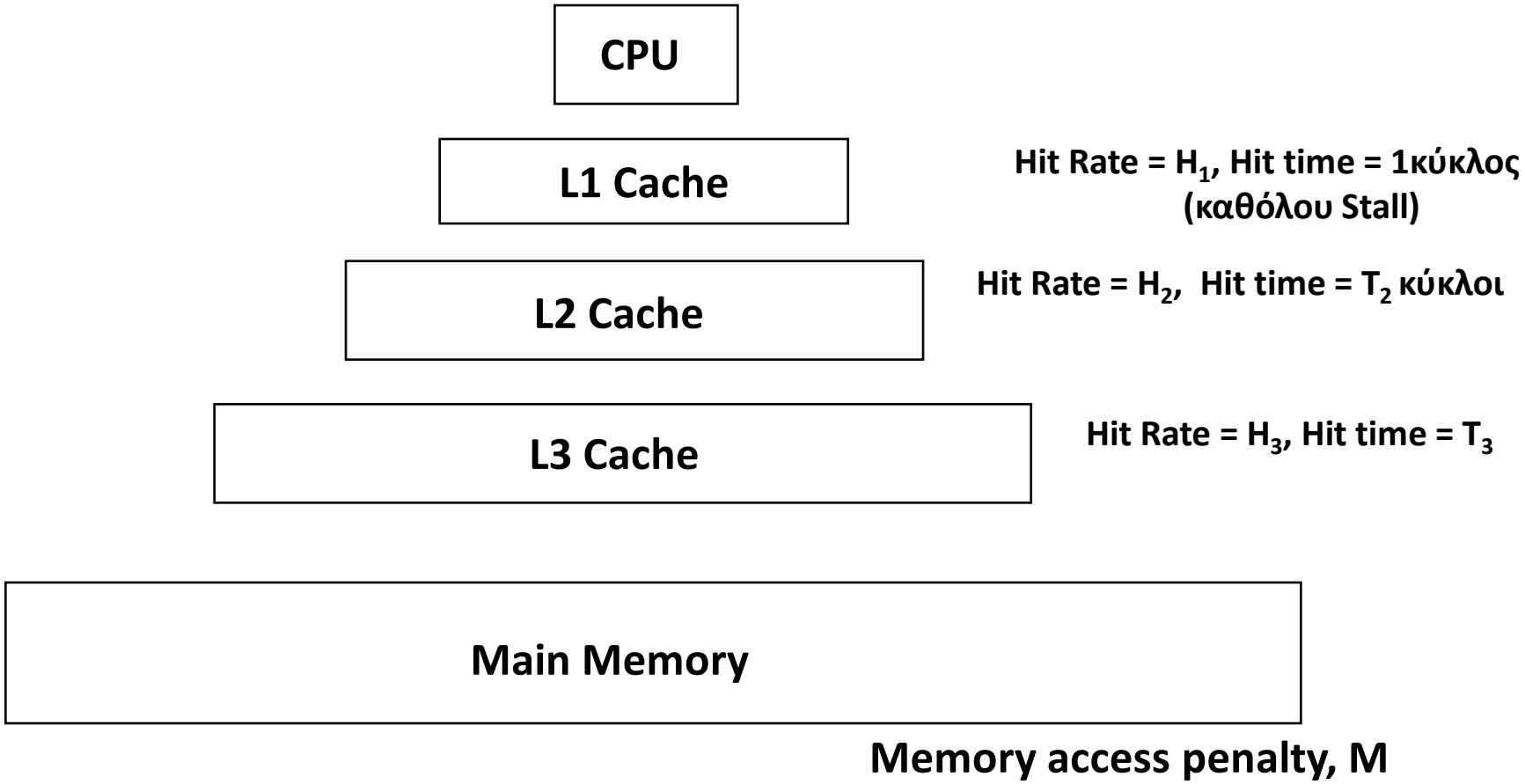
= 0.097 + 0.15 = 0.247

Mem Stalls/εντολή = Mem accesses/εντολή x Stalls/access = 0.247 x 1.3 = 0.32

CPI = 1.1 + 0.32 = 1.42

Speedup = 7.6/1.42 = 5.35

3 επίπεδα Cache



Επίδοση της L3 Cache

CPUtime = IC x ($CPI_{execution}$ + Mem Stalls/εντολή) x C

Mem Stalls/εντολή = Mem accesses /εντολή x Stalls/access

- Για ένα σύστημα με 3 επίπεδα cache, χωρίς penalty όταν τα δεδομένα βρεθούν στην L_1 cache:

Stalls/memory access =

[miss rate L_1] x [Hit rate L_2 x Hit time L_2

+ Miss rate L_2 x (Hit rate L3 x Hit time L_3

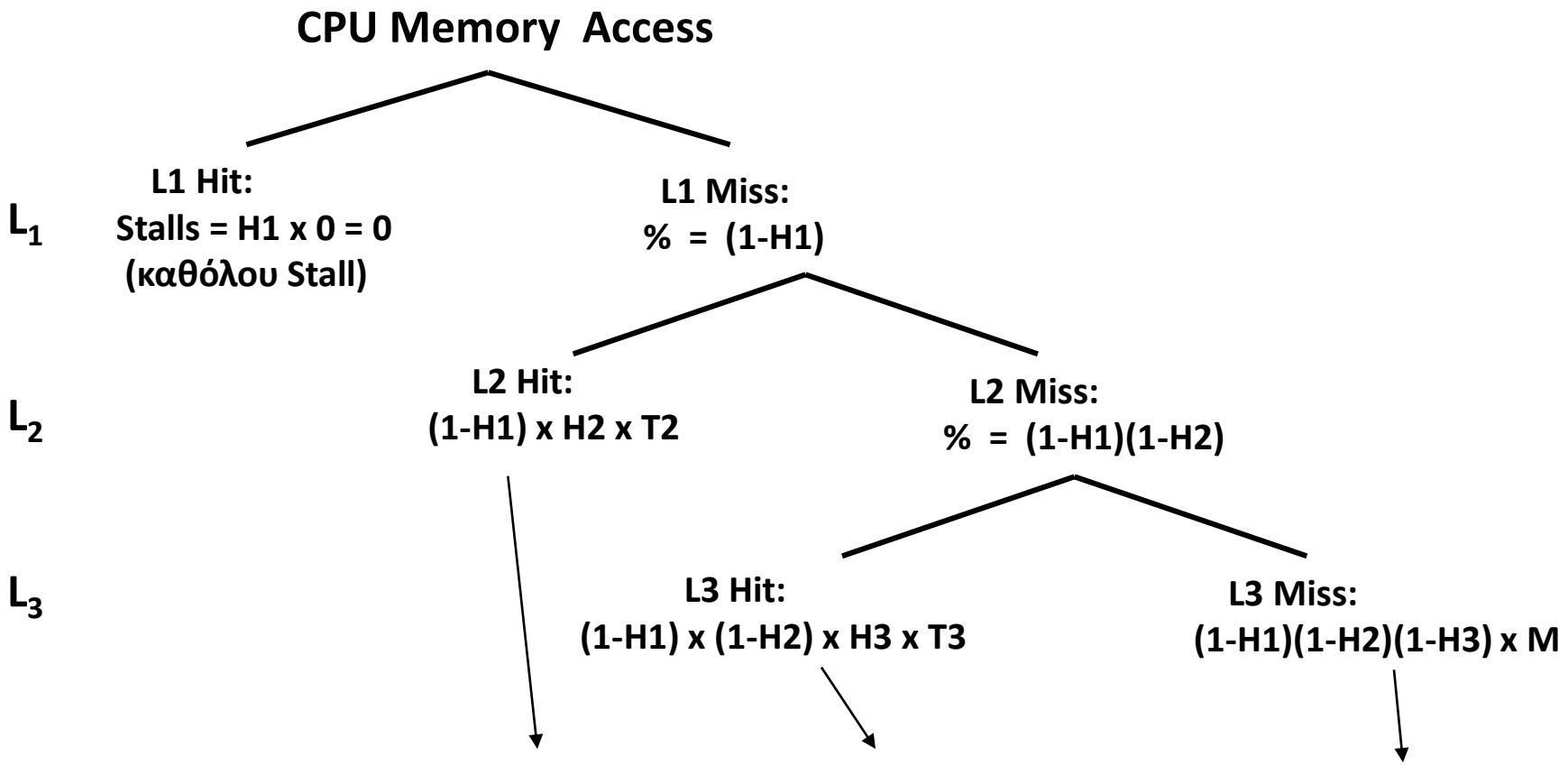
+ Miss rate L_3 x Memory access penalty)]

= (1-H1) x H2 x T2

+ (1-H1) x (1-H2) x H3 x T3

+ (1-H1)(1-H2) (1-H3)x M

Επίδοση της L3 Cache Memory Access Tree (CPU Stalls/Memory Access)



Stalls/memory access = (1-H1) x H2 x T2 + (1-H1) x (1-H2) x H3 x T3 + (1-H1)(1-H2) (1-H3) x M

Παράδειγμα L3 Cache

- CPU με $CPI_{execution}$ = 1.1 και συχνότητα 500 MHZ
- 1.3 memory accesses/εντολή.
- L_1 cache : στα 500 MHZ με miss rate 5%
- L_2 cache : στα 250 MHZ με miss rate 3%, (T_2 = 2 κύκλοι)
- L_3 cache : στα 100 MHZ με miss rate 1.5%, (T_3 = 5 κύκλοι)
- Memory access penalty, M= 100 cycles. Να βρείτε το CPI.

χωρίς Cache, CPI = 1.1 + 1.3 x 100 = 131.1

Με L_1 Cache, CPI = 1.1 + 1.3 x 0.05 x 100 = 7.6

Με L_2 Cache, CPI = 1.1 + 1.3 x (0.05 x 0.97 x 2 + 0.05 x 0.03 x 100) = 1.42

$$CPI = CPI_{execution} + \text{Mem Stalls/εντολή}$$

Mem Stalls/εντολή = Mem accesses/εντολή x Stall cycles/access

Stalls/memory access = (1-H1) x H2 x T2 + (1-H1) x (1-H2) x H3 x T3 + (1-H1)(1-H2) (1-H3) x M

= 0.05 x 0.97 x 2 + 0.05 x 0.03 x 0.985 x 5 + 0.05 x 0.03 x 0.015 x 100

= 0.097 + 0.0075 + 0.00225 = 0.107

$$CPI = 1.1 + 1.3 \times 0.107 = 1.24$$

Speedup σε σχέση με L1 μόνο = 7.6/1.24 = 6.12

Speedup σε σχέση με L1, L2 = 1.42/1.24 = 1.15