

# Ασκήσεις Caches

# Άσκηση 1η

Θεωρήστε ένα σύστημα μνήμης με μία cache:

- 4-way set associative
- μεγέθους 256KB,
- με cache line 8 λέξεων.

Χαρακτηριστικά συστήματος μνήμης:

- μέγεθος της λέξης είναι 32 bits.
- 1 byte η μικρότερη μονάδα δεδομένων που μπορεί να διευθυνσιοδοτηθεί
- 64 bit εύρος διευθύνσεων μνήμης



# Ζητούμενο

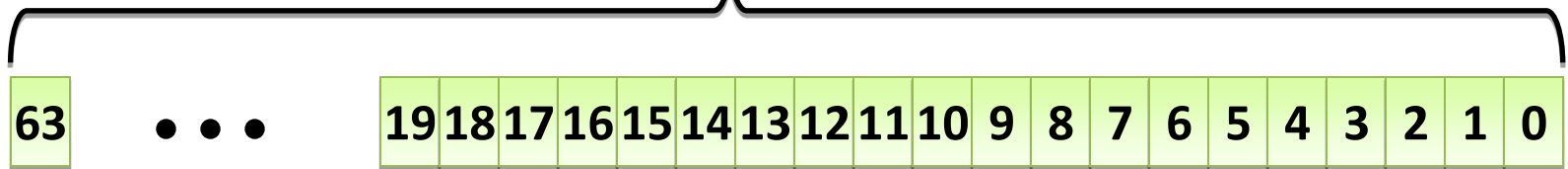
Για τα **επιμέρους πεδία** στα οποία χωρίζεται μία **διεύθυνση μνήμης** σε μία τέτοια οργάνωση cache, **υπολογίστε τον αριθμό των bits του καθενός.**

**Παρουσιάστε ένα διάγραμμα** που να δείχνει πώς διαχωρίζεται η **διεύθυνση** στα πεδία αυτά, και **εξηγήστε τη σημασία του καθενός.**

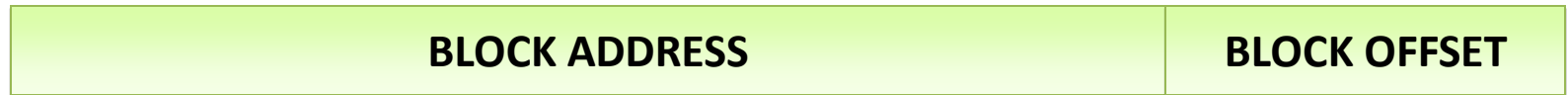
# ΣΚΕΠΤΙΚΟ

Μία διεύθυνση

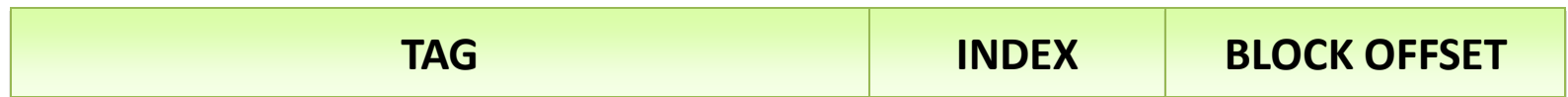
64 bits



χωρίζεται στα πεδία



ή



Πόσα bits είναι το εύρος κάθε πεδίου;

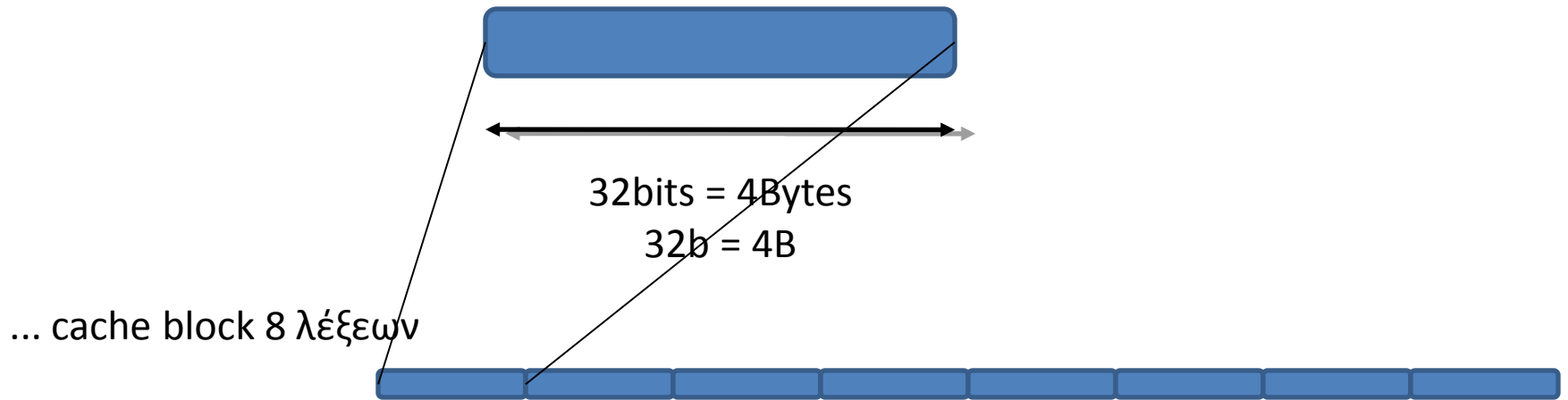
Το μέγεθος της λέξης είναι 32 bits



32bits = 4Bytes

32b = 4B

Το μέγεθος της λέξης είναι 32 bits



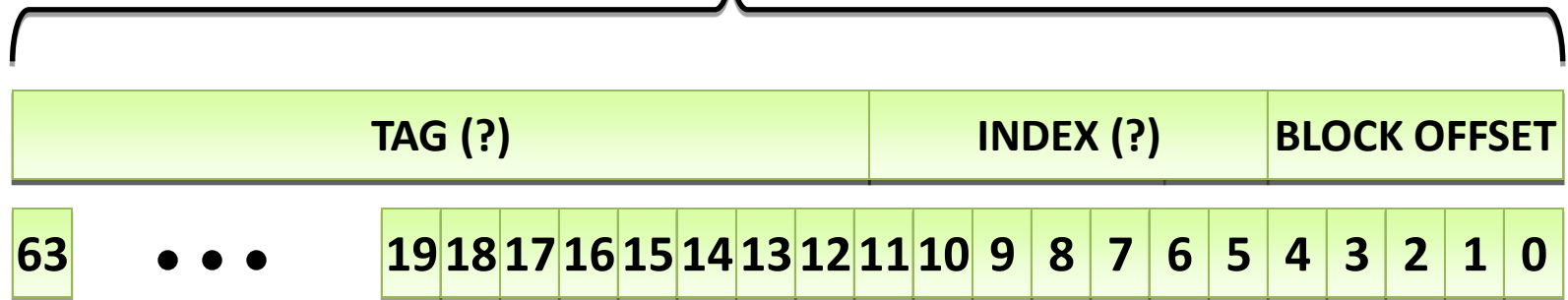
---

ένα cache block αποτελείται από 8 words



Μία διεύθυνση

64 bits



8 words    4 words    2 words    1 word    2 bytes

32 bytes    16 bytes    8 bytes    4 bytes

ένα cache block αποτελείται από 8 words



**BLOCK OFFSET = 5 bits**

ή

**$\log_2 32 = 5$  bits**



Το μέγεθος της λέξης είναι 32 bits



$$32\text{bits} = 4\text{Bytes}$$

$$32\text{b} = 4\text{B}$$

ένα cache block αποτελείται από 8 words



Το μέγεθος της λέξης είναι 32 bits



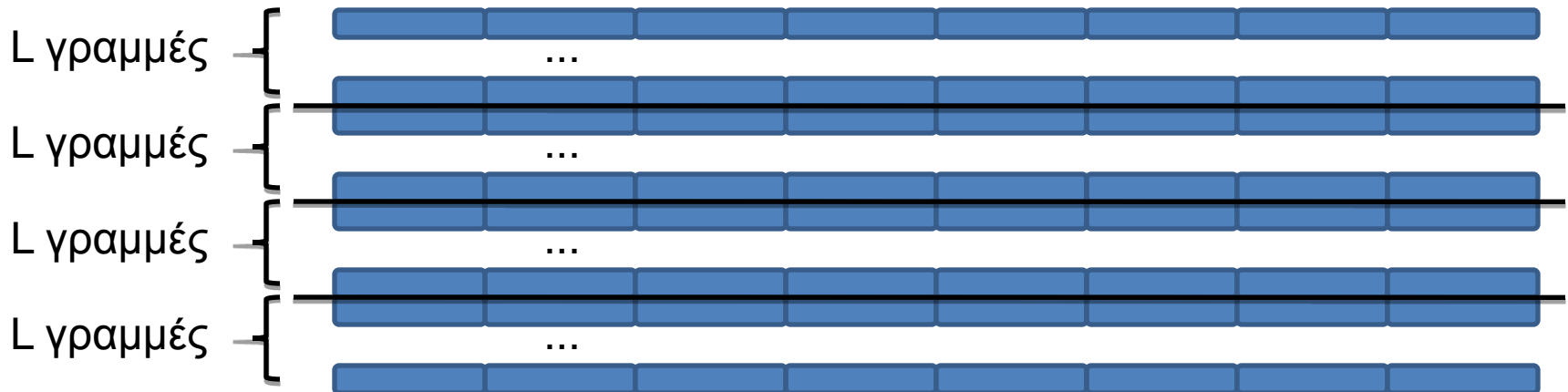
32bits = 4Bytes

32b = 4B

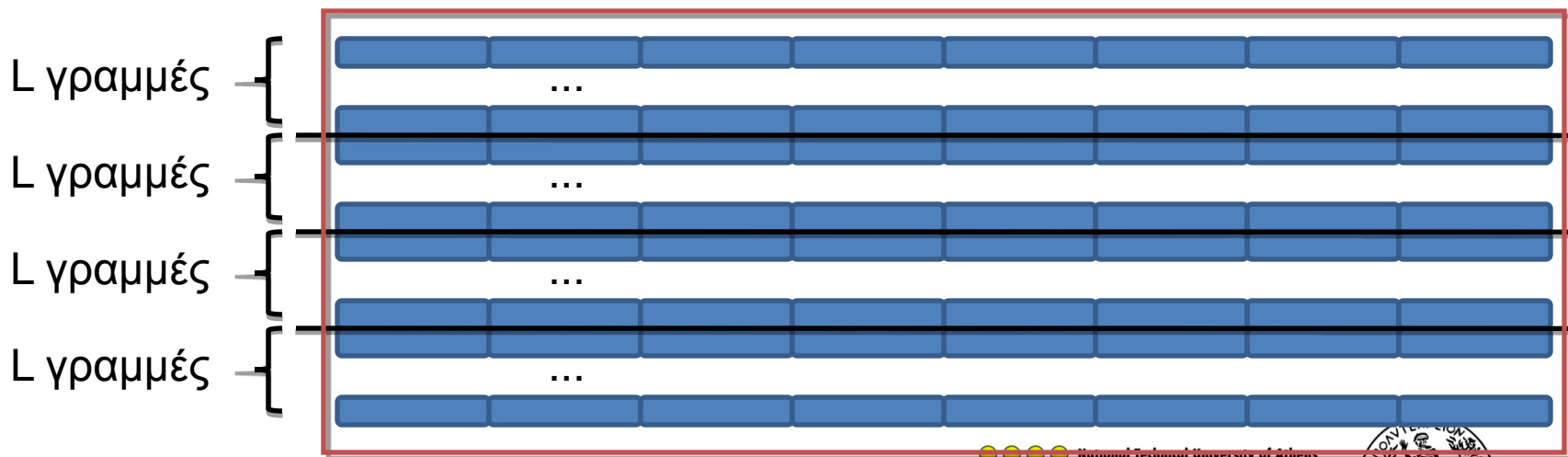
... 4-way set associative



... 4-way set associative

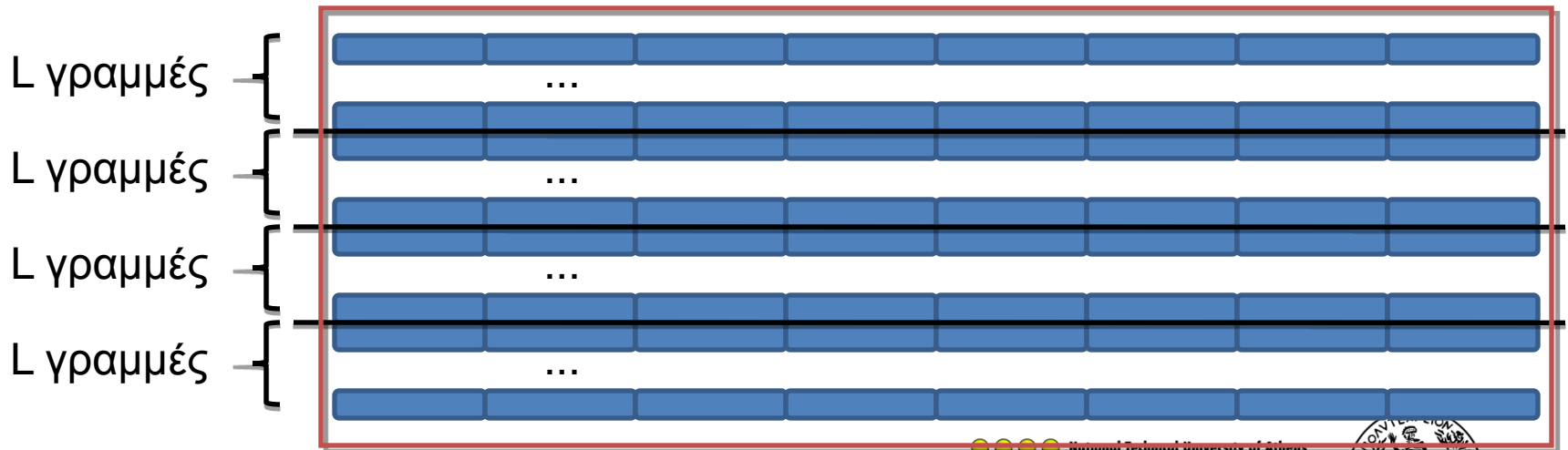


... μεγέθους 256KBytes



... μεγέθους 256KBytes

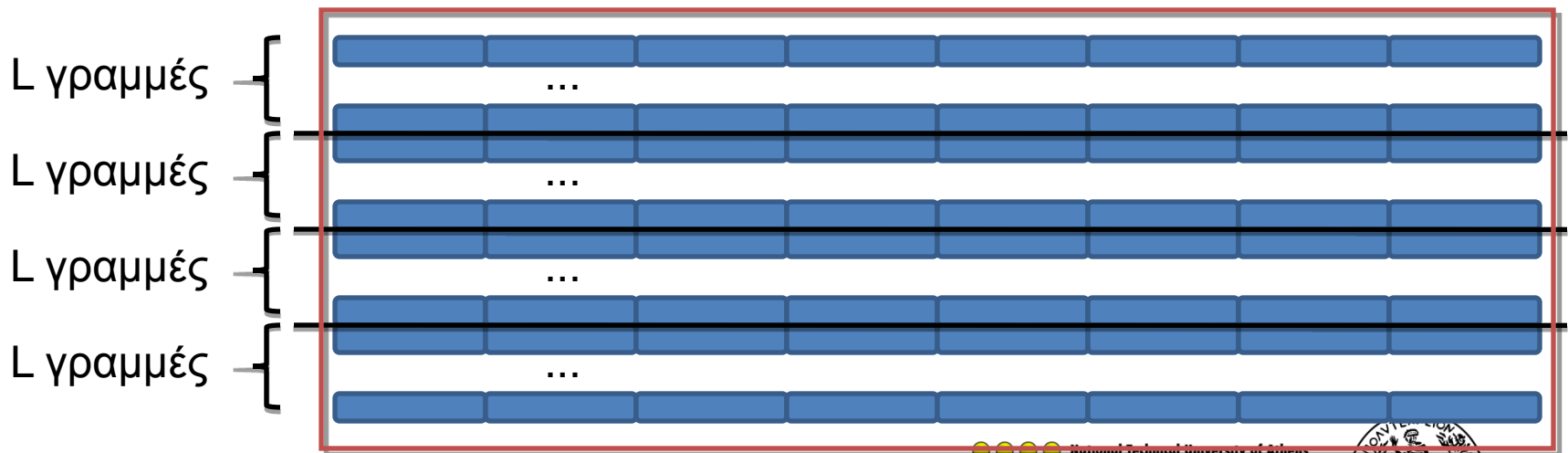
Δηλ.  $4 \times L \times 8 \times 4\text{Bytes} = 256\text{KBytes}$



... μεγέθους 256KBytes

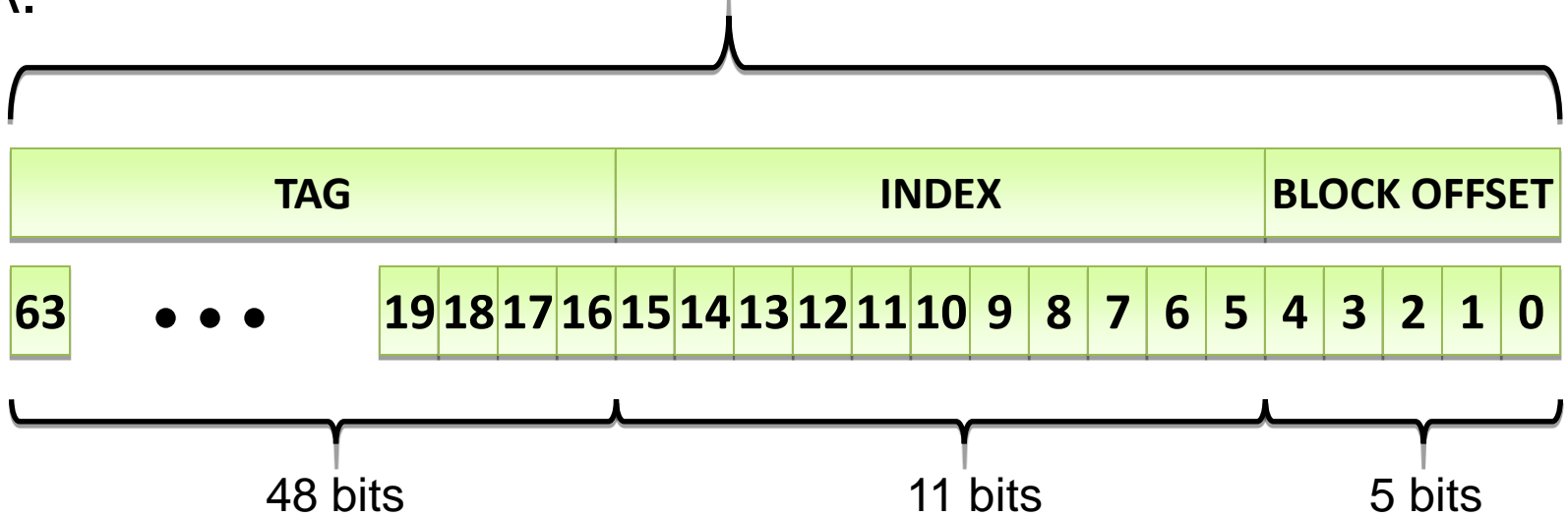
Δηλ.  $4 \times L \times 8 \times 4\text{Bytes} = 256\text{KBytes}$

Άρα,  $L = 2\text{K} = 2048$ , δηλ. κάθε way έχει 2048 γραμμές (blocks).  
Συνεπώς χρειάζονται **11 bits για το index**.



Δηλ.

64 bits



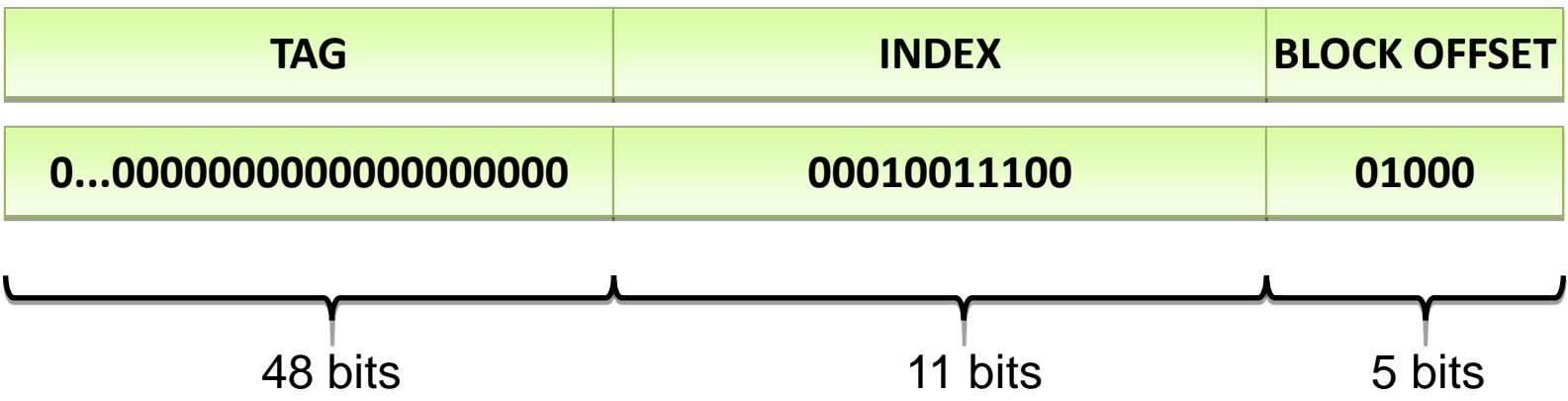
# Ζητούμενο (2)

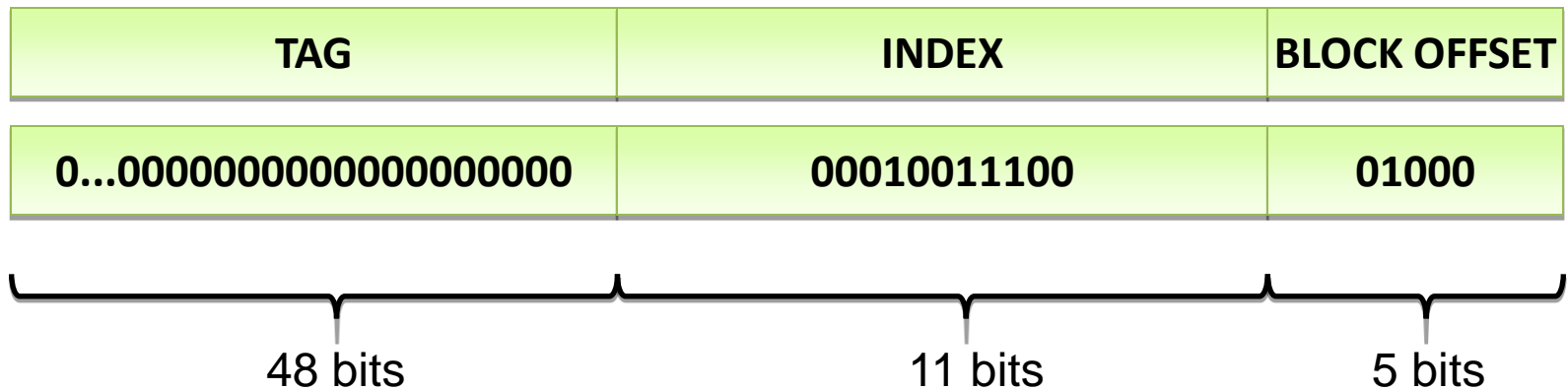
---

Σε ποιες θέσεις της cache μπορεί να απεικονιστεί το byte στη διεύθυνση μνήμης  $5000_{10}$  ;



$5000_{10} = 1001110001000_2$  η οποία διασπάται στα επιμέρους πεδία:

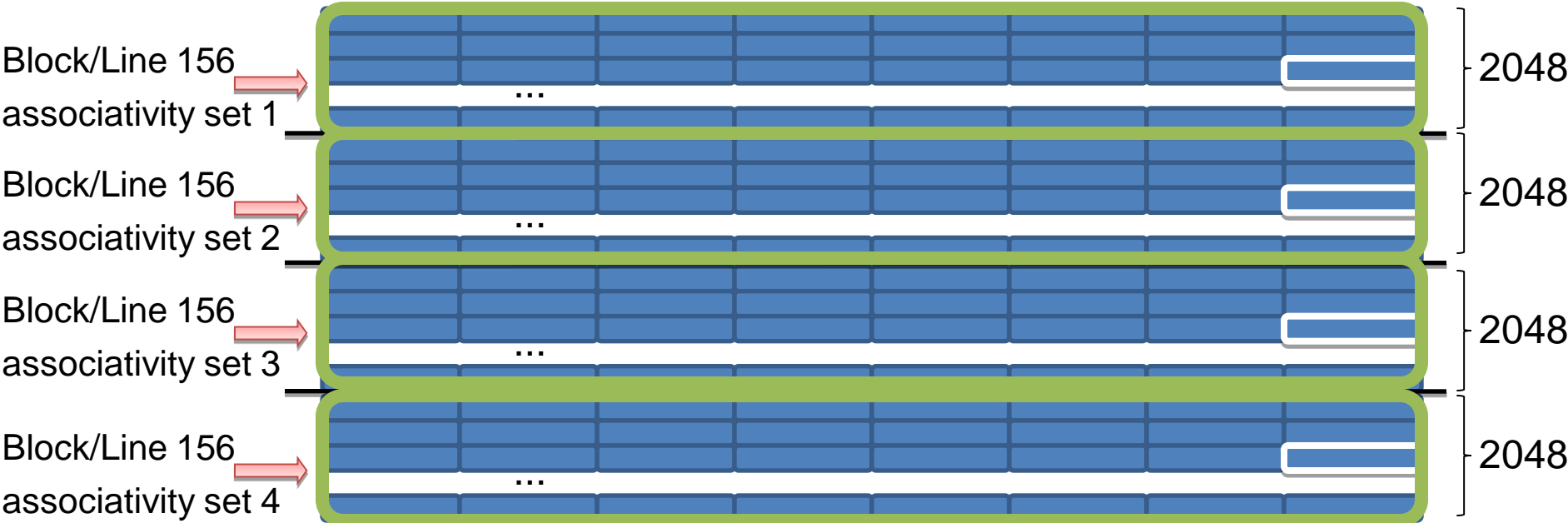




Το byte θα βρίσκεται στην 8η (01000<sub>2</sub>) θέση του block.

Το block αυτό, μπορεί να απεικονιστεί σε οποιαδήποτε από τις 4 θέσεις (4 way-set associative) με τιμή 156 (10011100<sub>2</sub>) της cache.

«Το block αυτό, μπορεί να απεικονιστεί σε οποιαδήποτε από τις 4 θέσεις (4 way-set associative) με τιμή 156 (10011100<sub>2</sub>) της cache.»

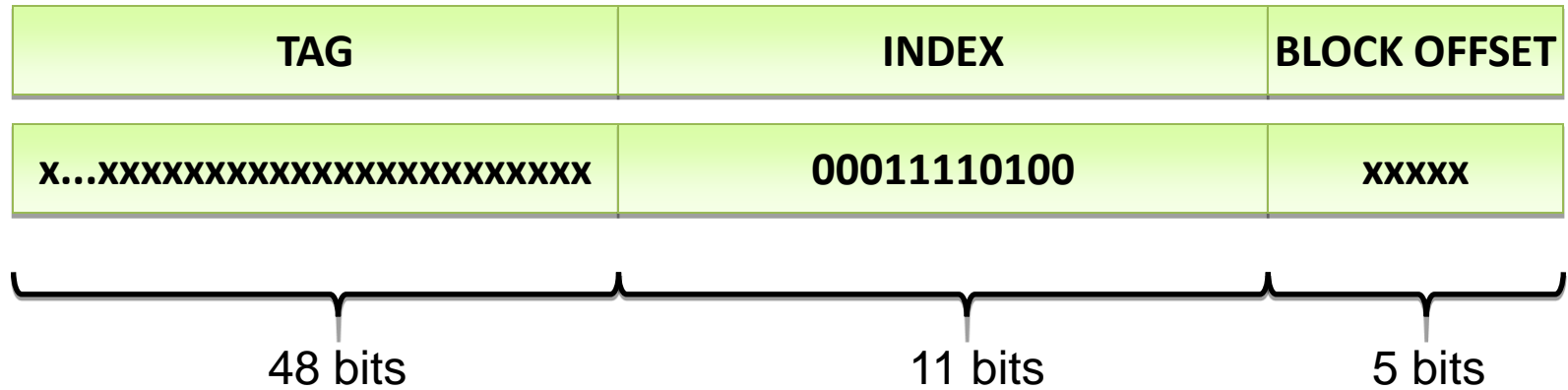


# Ζητούμενο (3)

---

**Ποιες θέσεις μνήμης μπορούν να απεικονιστούν στο σύνολο 244 της cache;**

# Απάντηση



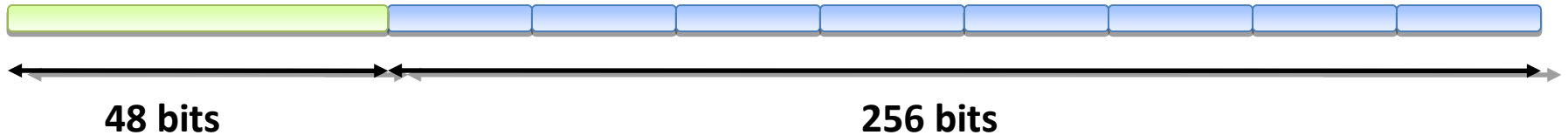
Όλες οι διευθύνσεις μνήμης που τα index bits είναι 244  
(00011110100<sub>2</sub>)

# Ζητούμενο (4)

---

**Τι ποσοστό του συνολικού μεγέθους της cache αφιερώνεται για τα bits του tag;**

# Απάντηση



Ένα cache block (cache line) αποτελείται από 256 bits δεδομένων, και του αντιστοιχεί 1 tag.

Επομένως (έστω ότι δε λαμβάνουμε υπόψη το valid bit), το ποσοστό του μεγέθους της cache που αφιερώνεται για τα bits του tag είναι  $48/(48+256) = 15.78\%$ .

# Άσκηση 2η

Θεωρούμε το ακόλουθο κομμάτι κώδικα

```
#define N 4
#define M 8
double c[N], a[N][M], b[M];
int i, j;

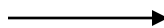
for(i = 0; i < N; i++)
    for(j = 0; j < M; j++)
        c[i] = c[i] + a[i][j] * b[j];
```

- Κάθε στοιχείο του πίνακα έχει μέγεθος 8 bytes
- Υπάρχει 1 επίπεδο κρυφής μνήμης, πλήρως συσχετιστικής, με LRU πολιτική αντικατάστασης, αποτελούμενη από 4 blocks δεδομένων μεγέθους 16 bytes
- Όλες οι μεταβλητές αποθηκεύονται σε καταχωρητές εκτός από τα στοιχεία των πινάκων
- Οι πίνακες αποθηκεύονται κατά γραμμές
- Η σειρά με την οποία γίνονται οι αναφορές είναι c, a, b, c.
- Αρχικά η cache είναι άδεια

*Βρείτε το συνολικό ποσοστό αστοχίας (miss rate) για τον παραπάνω κώδικα*



- 1 block = 16 bytes
- 1 στοιχείο = 8 bytes
- πίνακας αποθηκευμένος κατά γραμμές



σε 1 block της cache θα απεικονίζονται 2 διαδοχικά στοιχεία του πίνακα, π.χ.  $a[i][j]$ ,  $a[i][j+1]$

## αναφορά στη μνήμη

$i=0, j=0$

$c[0]$

compulsory miss

δηλαδή,  
αναφερόμαστε για  
1<sup>η</sup> φορά στα  
αντίστοιχα *blocks*...

## περιεχόμενα cache

$c[0]$	$c[1]$

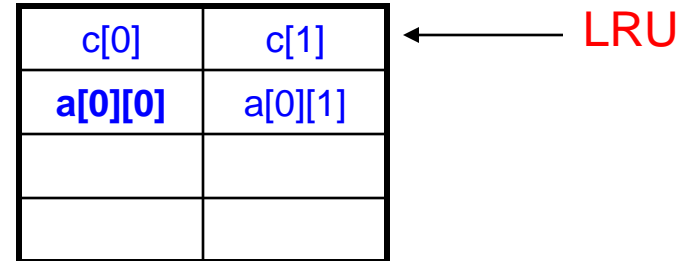
## αναφορά στη μνήμη

**$i=0, j=0$**

$c[0]$       **compulsory miss**

$a[0][0]$       **compulsory miss**

## περιεχόμενα cache



## αναφορά στη μνήμη

**i=0, j=0**

c[0]      **compulsory miss**

a[0][0]    **compulsory miss**

b[0]      **compulsory miss**

## περιεχόμενα cache

c[0]	c[1]	← LRU
a[0][0]	a[0][1]	
b[0]	b[1]	

## αναφορά στη μνήμη

**i=0,j=0**

c[0]      compulsory miss

a[0][0]    compulsory miss

b[0]      compulsory miss

c[0]      hit

## περιεχόμενα cache

c[0]	c[1]
a[0][0]	a[0][1]
b[0]	b[1]

← LRU

## αναφορά στη μνήμη

$i=0, j=1$

$c[0]$

hit

## περιεχόμενα cache

$c[0]$	$c[1]$
$a[0][0]$	$a[0][1]$
$b[0]$	$b[1]$

← LRU

## αναφορά στη μνήμη

**$i=0, j=1$**

$c[0]$  hit

$a[0][1]$  hit

## περιεχόμενα cache

$c[0]$	$c[1]$
$a[0][0]$	<b><math>a[0][1]</math></b>
$b[0]$	$b[1]$

← LRU

## αναφορά στη μνήμη

**i=0,j=1**

c[0]      hit

a[0][1]    hit

b[1]      hit

## περιεχόμενα cache

c[0]	c[1]
a[0][0]	a[0][1]
b[0]	<b>b[1]</b>

← LRU



## αναφορά στη μνήμη

**i=0, j=1**

c[0] hit

a[0][1] hit

b[1] hit

c[0] hit

## περιεχόμενα cache

c[0]	c[1]
a[0][0]	a[0][1]
b[0]	b[1]

← LRU

## αναφορά στη μνήμη

$i=0, j=2$

$c[0]$

hit

## περιεχόμενα cache

$c[0]$	$c[1]$
$a[0][0]$	$a[0][1]$
$b[0]$	$b[1]$

← LRU

## αναφορά στη μνήμη

**i=0, j=2**

c[0] hit

a[0][2] compulsory miss

## περιεχόμενα cache

c[0]	c[1]
a[0][0]	a[0][1]
b[0]	b[1]
a[0][2]	a[0][3]

← LRU

## αναφορά στη μνήμη

**i=0,j=2**

c[0] hit

a[0][2] compulsory miss

b[2] compulsory miss

## περιεχόμενα cache

c[0]	c[1]
b[2]	b[3]
b[0]	b[1]
a[0][2]	a[0][3]

← LRU

## αναφορά στη μνήμη

**i=0,j=2**

c[0] hit

a[0][2] compulsory miss

b[2] compulsory miss

c[0] hit

## περιεχόμενα cache

c[0]	c[1]
b[2]	b[3]
b[0]	b[1]
a[0][2]	a[0][3]

← LRU

## αναφορά στη μνήμη

**$i=0, j=3$**

$c[0]$

hit

## περιεχόμενα cache

$c[0]$	$c[1]$
$b[2]$	$b[3]$
$b[0]$	$b[1]$
$a[0][2]$	$a[0][3]$

← LRU

## αναφορά στη μνήμη

**i=0, j=3**

c[0] hit

a[0][3] hit

## περιεχόμενα cache

c[0]	c[1]
b[2]	b[3]
b[0]	b[1]
a[0][2]	<b>a[0][3]</b>

← LRU

## αναφορά στη μνήμη

**i=0,j=3**

c[0] hit

a[0][3] hit

b[3] hit

## περιεχόμενα cache

c[0]	c[1]
b[2]	<b>b[3]</b>
b[0]	b[1]
a[0][2]	a[0][3]

← LRU



## αναφορά στη μνήμη

**i=0, j=3**

c[0] hit

a[0][3] hit

b[3] hit

c[0] hit

## περιεχόμενα cache

c[0]	c[1]
b[2]	b[3]
b[0]	b[1]
a[0][2]	a[0][3]

← LRU

## αναφορά στη μνήμη

$i=0, j=4$

$c[0]$

hit

## περιεχόμενα cache

$c[0]$	$c[1]$
$b[2]$	$b[3]$
$b[0]$	$b[1]$
$a[0][2]$	$a[0][3]$

← LRU

## αναφορά στη μνήμη

**$i=0, j=4$**

$c[0]$  hit

$a[0][4]$  compulsory miss

## περιεχόμενα cache

$c[0]$	$c[1]$
$b[2]$	$b[3]$
$a[0][4]$	$a[0][5]$
$a[0][2]$	$a[0][3]$

← LRU

## αναφορά στη μνήμη

**i=0,j=4**

c[0]      hit

a[0][4]      compulsory miss

b[4]      compulsory miss

## περιεχόμενα cache

c[0]	c[1]
b[2]	b[3]
a[0][4]	a[0][5]
b[4]	b[5]

← LRU

## αναφορά στη μνήμη

**i=0, j=4**

c[0] hit

a[0][4] compulsory miss

b[4] compulsory miss

c[0] hit

## περιεχόμενα cache

c[0]	c[1]
b[2]	b[3]
a[0][4]	a[0][5]
b[4]	b[5]

← LRU

## αναφορά στη μνήμη

**i=0,j=5**

c[0] hit

a[0][5] hit

b[5] hit

c[0] hit

## περιεχόμενα cache

c[0]	c[1]
b[2]	b[3]
a[0][4]	a[0][5]
b[4]	b[5]

← LRU

## αναφορά στη μνήμη

c0 a00 b0 c0  
c0 a01 b1 c0  
c0 a02 b2 c0  
c0 a03 b3 c0  
c0 a04 b4 c0  
c0 a05 b5 c0  
c0 a06 b6 c0  
c0 a07 b7 c0  
c1 a10 b0 c1  
c1 a11 b1 c1  
c1 a12 b2 c1  
c1 a13 b3 c1  
c1 a14 b4 c1  
c1 a15 b5 c1  
c1 a16 b6 c1  
c1 a17 b7 c1

## αποτέλεσμα

m m m h  
h h h h  
h m m h  
h h h h  
h m m h  
h h h h  
h h h h  
h m m h  
h h h h  
h m m h  
h h h h  
h m m h  
h h h h

9 misses  
23 hits

8 misses  
24 hits

## αναφορά στη μνήμη

c2 a20 b0 c2  
c2 a21 b1 c2  
c2 a22 b2 c2  
c2 a23 b3 c2  
c2 a24 b4 c2  
c2 a25 b5 c2  
c2 a26 b6 c2  
c2 a27 b7 c2  
c3 a30 b0 c3  
c3 a31 b1 c3  
c3 a32 b2 c3  
c3 a33 b3 c3  
c3 a34 b4 c3  
c3 a35 b5 c3  
c3 a36 b6 c3  
c3 a37 b7 c3

## αποτέλεσμα

m m m h  
h h h h  
h m m h  
h h h h  
h m m h  
h h h h  
h m m h  
h h h h  
h m m h  
h h h h  
h m m h  
h h h h  
h m m h  
h h h h

9 misses  
23 hits

8 misses  
24 hits



## Συνολικά

**Accesses** : 128  
**Misses** : 34



Miss rate = 0.266

# Άσκηση 3η

Θεωρούμε το ακόλουθο κομμάτι κώδικα

```
int i, j;
double result, a[110][4];

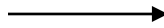
for(i=0; i<4; i++)
for(j=0; j<100; j++)
    result += a[j][i]*a[j+1][i] + 0.5;
```

Υποθέσεις:

- κάθε στοιχείο του πίνακα έχει μέγεθος 8 bytes
- υπάρχει 1 επίπεδο κρυφής μνήμης, πλήρως συσχετιστικής, με LRU πολιτική αντικατάστασης, αποτελούμενη από 100 blocks δεδομένων
- το μέγεθος του block είναι 32 bytes
- ο πίνακας είναι αποθηκευμένος στην κύρια μνήμη κατά γραμμές, και είναι «ευθυγραμμισμένος» ώστε το 1<sup>ο</sup> στοιχείο του να απεικονίζεται στην αρχή μιας γραμμής της cache
- αρχικά η cache είναι άδεια

- Βρείτε ποιες από τις αναφορές στα στοιχεία του πίνακα *a* για όλη την εκτέλεση του παραπάνω κώδικα καταλήγουν σε *misses* στην *cache*.
- Υποδείξτε ποια είναι *compulsory*, ποια είναι *capacity*, και ποια *conflict*.
- Δώστε τον συνολικό αριθμό των *misses*.

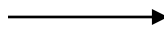
- 1 block = 32 bytes
- 1 στοιχείο = 8 bytes
- πίνακας αποθηκευμένος κατά γραμμές



σε 1 block της cache θα απεικονίζονται 4 διαδοχικά στοιχεία του πίνακα, π.χ.  
 $a[i][j]$ ,  $a[i][j+1]$ ,  $a[i][j+2]$ ,  $a[i][j+3]$

## ΕΠΙΠΛΕΟΝ

- πίνακας ευθυγραμμισμένος



σε 1 block της cache θα απεικονίζεται 1 ολόκληρη γραμμή του πίνακα, δηλαδή  
 $a[i][j]$ ,  $a[i][j+1]$ ,  $a[i][j+2]$ ,  $a[i][j+3]$   
όπου  $j\%4=0$

αναφορά στη μνήμη

περιεχόμενα cache

$i=0, j=0$   
 $a[0][0]$

compulsory miss

δηλαδή,  
αναφερόμαστε για 1<sup>η</sup>  
φορά στα αντίστοιχα  
blocks...

$a[0][0]$	$a[0][1]$	$a[0][2]$	$a[0][3]$



αναφορά στη μνήμη

περιεχόμενα cache

**i=0,j=0**

a[0][0]      **compulsory miss**

a[1][0]      **compulsory miss**

a[0][0]	a[0][1]	a[0][2]	a[0][3]
<b>a[1][0]</b>	a[1][1]	a[1][2]	a[1][3]

## αναφορά στη μνήμη

**i=0,j=0**

a[0][0]      **compulsory miss**

a[1][0]      **compulsory miss**

**i=0,j=1**

a[1][0]      **hit**

## περιεχόμενα cache

a[0][0]	a[0][1]	a[0][2]	a[0][3]
<b>a[1][0]</b>	a[1][1]	a[1][2]	a[1][3]

## αναφορά στη μνήμη

**i=0,j=0**

a[0][0]      **compulsory miss**

a[1][0]      **compulsory miss**

**i=0,j=1**

a[1][0]      **hit**

a[2][0]      **compulsory miss**

## περιεχόμενα cache

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
<b>a[2][0]</b>	a[2][1]	a[2][2]	a[2][3]



## αναφορά στη μνήμη

**$i=0, j=0$**

$a[0][0]$  **compulsory miss**

$a[1][0]$  **compulsory miss**

**$i=0, j=1$**

$a[1][0]$  **hit**

$a[2][0]$  **compulsory miss**

Έτσι για  $i=0, j=1 \dots 99$  θα έχουμε 1 hit + 1 comp. miss για κάθε επανάληψη του  $j$

## περιεχόμενα cache

$a[0][0]$	$a[0][1]$	$a[0][2]$	$a[0][3]$
$a[1][0]$	$a[1][1]$	$a[1][2]$	$a[1][3]$
$a[2][0]$	$a[2][1]$	$a[2][2]$	$a[2][3]$

## αναφορά στη μνήμη

**$i=0, j=0$**

$a[0][0]$       **compulsory miss**

$a[1][0]$       **compulsory miss**

**$i=0, j=1$**

$a[1][0]$       **hit**

$a[2][0]$       **compulsory miss**

Έτσι για  $i=0, j=1 \dots 99$  θα έχουμε 1  
hit + 1 comp. miss για κάθε  
επανάληψη του  $j$



για  $i=0$  θα έχουμε  $2+99=101$   
misses, όλα compulsory

## περιεχόμενα cache

$a[0][0]$	$a[0][1]$	$a[0][2]$	$a[0][3]$
$a[1][0]$	$a[1][1]$	$a[1][2]$	$a[1][3]$
$a[2][0]$	$a[2][1]$	$a[2][2]$	$a[2][3]$

## αναφορά στη μνήμη

**$i=0, j=99$**

$a[99][0]$       **hit**

## περιεχόμενα cache

$a[0][0]$	$a[0][1]$	$a[0][2]$	$a[0][3]$
$a[1][0]$	$a[1][1]$	$a[1][2]$	$a[1][3]$
$a[2][0]$	$a[2][1]$	$a[2][2]$	$a[2][3]$
$a[3][0]$	$a[3][1]$	$a[3][2]$	$a[3][3]$
$a[4][0]$	$a[4][1]$	$a[4][2]$	$a[4][3]$
...	...	...	...
$a[98][0]$	$a[98][1]$	$a[98][2]$	$a[98][3]$
<b><math>a[99][0]</math></b>	$a[99][1]$	$a[99][2]$	$a[99][3]$

αναφορά στη μνήμη

περιεχόμενα cache

**i=0,j=99**

a[99][0]

hit

a[100][0]

compulsory miss



αντικατέστησε το LRU  
block που υπήρχε  
στην cache

<b>a[100][0]</b>	a[100][1]	a[100][2]	a[100][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]
a[3][0]	a[3][1]	a[3][2]	a[3][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
...	...	...	...
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]

## αναφορά στη μνήμη

$i=1, j=0$

$a[0][1]$

capacity miss

διότι αντικαταστάθηκε  
λόγω έλλειψης χώρου το  
block που είχε έρθει στην  
cache κατά το παρελθόν  
και το περιείχε

## περιεχόμενα cache

$a[100][0]$	$a[100][1]$	$a[100][2]$	$a[100][3]$
$a[0][0]$	<b><math>a[0][1]</math></b>	$a[0][2]$	$a[0][3]$
$a[2][0]$	$a[2][1]$	$a[2][2]$	$a[2][3]$
$a[3][0]$	$a[3][1]$	$a[3][2]$	$a[3][3]$
$a[4][0]$	$a[4][1]$	$a[4][2]$	$a[4][3]$
...	...	...	...
$a[98][0]$	$a[98][1]$	$a[98][2]$	$a[98][3]$
$a[99][0]$	$a[99][1]$	$a[99][2]$	$a[99][3]$

## αναφορά στη μνήμη

**i=1,j=0**

a[0][1]      capacity miss

a[1][1]      capacity miss

## περιεχόμενα cache

a[100][0]	a[100][1]	a[100][2]	a[100][3]
a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	<b>a[1][1]</b>	a[1][2]	a[1][3]
a[3][0]	a[3][1]	a[3][2]	a[3][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
...	...	...	...
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]

## αναφορά στη μνήμη

**i=1,j=0**

a[0][1]      **capacity miss**

a[1][1]      **capacity miss**

**i=1,j=1**

a[1][1]      **hit**

## περιεχόμενα cache

a[100][0]	a[100][1]	a[100][2]	a[100][3]
a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	<b>a[1][1]</b>	a[1][2]	a[1][3]
a[3][0]	a[3][1]	a[3][2]	a[3][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
...	...	...	...
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]

## αναφορά στη μνήμη

## περιεχόμενα cache

**i=1,j=0**

a[0][1]      capacity miss

a[1][1]      capacity miss

**i=1,j=1**

a[1][1]      hit

a[2][1]      capacity miss

a[100][0]	a[100][1]	a[100][2]	a[100][3]
a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	<b>a[2][1]</b>	a[2][2]	a[2][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
...	...	...	...
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]



## αναφορά στη μνήμη

## περιεχόμενα cache

**i=1,j=0**

a[0][1]      capacity miss

a[1][1]      capacity miss

**i=1,j=1**

a[1][1]      hit

a[2][1]      capacity miss

a[100][0]	a[100][1]	a[100][2]	a[100][3]
a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	<b>a[2][1]</b>	a[2][2]	a[2][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
...	...	...	...
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]

Στις επόμενες επαναλήψεις θα έχουμε κυκλικές αντικαταστάσεις blocks, οπότε τα misses και τα hits θα ακολουθούν το ίδιο μοτίβο όπως και για i=0.

## αναφορά στη μνήμη

## περιεχόμενα cache

**i=1,j=0**

a[0][1]      **capacity miss**

a[1][1]      **capacity miss**

**i=1,j=1**

a[1][1]      **hit**

a[2][1]      **capacity miss**

a[100][0]	a[100][1]	a[100][2]	a[100][3]
a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	<b>a[2][1]</b>	a[2][2]	a[2][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
...	...	...	...
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]

Στις επόμενες επαναλήψεις θα έχουμε κυκλικές αντικαταστάσεις blocks, οπότε τα misses και τα hits θα ακολουθούν το ίδιο μοτίβο όπως και για i=0.



συνολικά θα έχουμε  $4 \cdot 101 = 404$  misses

## αναφορά στη μνήμη

## περιεχόμενα cache

**i=1,j=0**

a[0][1]      **capacity miss**

a[1][1]      **capacity miss**

**i=1,j=1**

a[1][1]      **hit**

a[2][1]      **capacity miss**

a[100][0]	a[100][1]	a[100][2]	a[100][3]
a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	<b>a[2][1]</b>	a[2][2]	a[2][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
...	...	...	...
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]

Conflict misses δεν έχουμε διότι η cache είναι fully associative → τα blocks δεδομένων μπορούν να απεικονιστούν οπουδήποτε στην cache

Θεωρείστε πάλι τον αρχικό κώδικα. Ποια γνωστή τεχνική βελτιστοποίησης θα εφαρμόζατε στον κώδικα ώστε να μειωθούν τα misses;

```
for(i=0; i<4; i++)  
  for(j=0; j<100; j++)  
    result += a[j][i]*a[j+1][i] + 0.5;
```

στο σώμα του loop δεν υπάρχουν εξαρτήσεις, επομένως μπορούμε να εφαρμόσουμε αναδιάταξη βρόχων (*loop interchange*)

```
for(j=0; j<100; j++)  
  for(i=0; i<4; i++)  
    result += a[j][i]*a[j+1][i] + 0.5;
```

τώρα ο πίνακας προσπελαύνεται όπως είναι αποθηκευμένος → καλύτερη τοπικότητα αναφορών, αφού γειτονικά στοιχεία προσπελούνται σε διαδοχικές επαναλήψεις του εσωτερικού loop

misses συμβαίνουν όταν αναφερόμαστε στο πρώτο στοιχείο κάθε γραμμής ( $i=0$ ) → συνολικά έχουμε 101 misses

# Άσκηση 4η

Θεωρούμε το ακόλουθο κομμάτι κώδικα:

```
#define N 1024
float A[N], B[N];
for(i=0; i<N; i+=1)
B[i] += 2*A[i];
```

Κάνουμε τις εξής υποθέσεις:

- Το πρόγραμμα εκτελείται σε έναν επεξεργαστή με μόνο ένα επίπεδο κρυφής μνήμης δεδομένων, η οποία αρχικά είναι άδεια. Η κρυφή μνήμη είναι *direct mapped*, *write-allocate*, και έχει μέγεθος 2KB. Το μέγεθος του *block* είναι 16 bytes.
- Το μέγεθος ενός *float* είναι 4 bytes.
- Δήλωση διαδοχικών μεταβλητών (βαθμωτών και μη) στο πρόγραμμα συνεπάγεται αποθήκευσή τους σε διαδοχικές θέσεις στη μνήμη.

## Βρείτε το συνολικό ποσοστό αστοχίας (miss rate) για τις αναφορές που γίνονται στην μνήμη στον παραπάνω κώδικα

```
#define N 1024
float A[N], B[N];
for(i=0; i<N; i+=1)
    B[i] += 2*A[i];
```

- #blocks= 2048/16 = 128
- σε κάθε block => 16/4 = 4 στοιχεία ενός πίνακα
- Πού απεικονίζονται τα  $A[i]$ ,  $B[i]$ ; στο ίδιο block... *γιατί;*
  - Άρα: read B[0] (m), read A[0] (m), write B[0] (m)  
read B[1] (h) *γιατί;*, read A[1] (m), write B[1] (m)  
read B[2] (h), read A[2] (m), write B[2] (m)  
read B[3] (h), read A[3] (m), write B[3] (m)  
read B[4] (m), read A[4] (m), write B[4] (m)  
...
- Ανά 4 επαναλήψεις: 9 misses, 3 hits => miss rate = 9/12 = 75%

Ποιες από τις παρακάτω τεχνικές βελτιστοποίησης **επιπέδου λογισμικού** θα ακολουθούσατε προκειμένου να βελτιώσετε την απόδοση του κώδικα;

- *Loop unrolling*
- *Merging arrays*
- *Loop blocking*
- *Loop distribution*

# Loop unrolling

```
for(i=0; i<N; i+=1) {  
    B[i] += 2*A[i];  
    B[i+1] += 2*A[i+1];  
    B[i+2] += 2*A[i+2];  
    B[i+3] += 2*A[i+3];  
}
```

- Βοηθάει;
  - *Πού στοχεύει η τεχνική αυτή;*



# Loop blocking

```
for(i=0; i<N; i+=bs)
    for(ii=i; ii<min(i+bs,N); ii++)
        B[ii] += 2*A[ii];
```

- Βοηθάει;
  - Πού στοχεύει η τεχνική αυτή;
  - Ποιο το πρόβλημα απόδοσης του αρχικού κώδικα όσον αφορά τα *misses*;

# Loop distribution

```
for(i=0; i<N; i+=1) {  
    B[i] += 2*A[i];    =>    ???  
}
```

- Σε τι στοχεύει η τεχνική αυτή;
- Υπό ποιες προϋποθέσεις θα βοηθούσε;

# Merging arrays

```
float A[N], B[N];  
for(i=0; i<N; i+=1)  
    B[i] += 2*A[i];
```

*ο κώδικας γίνεται:*

```
struct merge {  
    float a;  
    float b; };  
struct merge merge_array[1024];  
for(i=0; i<N; i+=1)  
    merge_array[i].b += 2*merge_array[i].a
```

- Σε κάθε block τώρα έχουμε 2 στοιχεία του A και 2 του B
  - Για ζυγά i: 1 miss σε 3 accesses
  - Για μονά i: κανένα miss σε 3 accesses
- Άρα miss rate =  $1/6 = 16.67\%$

Ποιες από τις παρακάτω τεχνικές βελτιστοποίησης **επιπέδου υλικού** θα ακολουθούσατε προκειμένου να βελτιώσετε την απόδοση του κώδικα;

- αύξηση *block size* σε 32 bytes (με διατήρηση της χωρητικότητας)
- αύξηση *associativity* σε 2-way (με διατήρηση της χωρητικότητας της cache)
- προσθήκη *victim cache*
- χρήση μηχανισμού *hardware prefetching*

Ποιο το πρόβλημα απόδοσης του αρχικού κώδικα όσον αφορά τα *misses*, και πώς μπορεί να βοηθήσει η κάθε τεχνική;

# Άσκηση 5η

Εξετάζουμε την εκτέλεση του ακόλουθου βρόχου (αντιμετάθεση πίνακα):

```
for(i=0; i<256; i++)  
    for(j=0; j<256; j++)  
        b[i][j] = a[j][i];
```

- *στοιχεία κινητής υποδιαστολής διπλής ακρίβειας (8 bytes)*
- *ένα επίπεδο data cache: fully associative, write-allocate, 16 KB, LRU πολιτική αντικατάστασης*
- *block size = 64 bytes*
- *οι πίνακες είναι αποθηκευμένοι κατά γραμμές, και “ευθυγραμμισμένοι” ώστε το πρώτο στοιχείο τους να απεικονίζεται στην αρχή μιας γραμμής της cache*

## Βρείτε το συνολικό miss rate.

```
for(i=0; i<256; i++)
    for(j=0; j<256; j++)
        b[i][j] = a[j][i];
```

- cache line 64 bytes => 8 στοιχεία πίνακα σε 1 cache line
- Για την αποθήκευση 1 γραμμής του πίνακα =>  $8 \cdot 256 = 2048$  bytes, ή 32 cache lines
- Για την αποθήκευση 1 στήλης =>  $64 \cdot 256$  bytes (*γιατί;*) = 16KB ή 256 cache lines
  - Τα στοιχεία μιας στήλης δεν μπορούν να επαναχρησιμοποιηθούν

```

for(i=0; i<256; i++)
    for(j=0; j<256; j++)
        b[i][j] = a[j][i];

```

Για την 1η επανάληψη του εξωτερικού loop:

a0,0	b0,0	m	m	
a1,0	b0,1	m	h	
a2,0	b0,2	m	h	
...				
<u>a7,0</u>	<u>b0,7</u>	<u>m</u>	<u>h</u>	
a8,0	b0,8	m	m	(νέα cache line για τον b)
a9,0	b0,9	m	h	
...				
<u>a15,0</u>	<u>b0,15</u>	<u>m</u>	<u>h</u>	
a16,0	b0,16	m	m	
...				

Το miss pattern επαναλαμβάνεται ανά 8 επαναλήψεις του εσωτερικού loop

Άρα συνολικά θα έχουμε  $256/8 * 9 \text{ misses} = 288$  για 1 επανάληψη του εξ. loop

```
for(i=0; i<256; i++)
    for(j=0; j<256; j++)
        b[i][j] = a[j][i];
```

Για την 1η επανάληψη του εξωτερικού loop:

a0,0	b0,0	m	m	
a1,0	b0,1	m	h	
a2,0	b0,2	m	h	
...				
<u>a7,0</u>	<u>b0,7</u>	<u>m</u>	<u>h</u>	
a8,0	b0,8	m	m	(νέα cache line για τον b)
a9,0	b0,9	m	h	
...				
<u>a15,0</u>	<u>b0,15</u>	<u>m</u>	<u>h</u>	
a16,0	b0,16	m	m	
...				

Στην επόμενη επανάληψη του εξ. loop δεν υπάρχει επαναχρησιμοποίηση για κανέναν από τους 2 πίνακες:

ο b διατρέχεται ούτως ή άλλως κατά γραμμές...

ο a;



```

for(i=0; i<256; i++)
    for(j=0; j<256; j++)
        b[i][j] = a[j][i];

```

Για την 1η επανάληψη του εξωτερικού loop:

a0,0	b0,0	m	m	
a1,0	b0,1	m	h	
a2,0	b0,2	m	h	
...				
<u>a7,0</u>	<u>b0,7</u>	<u>m</u>	<u>h</u>	
a8,0	b0,8	m	m	(νέα cache line για τον b)
a9,0	b0,9	m	h	
...				
<u>a15,0</u>	<u>b0,15</u>	<u>m</u>	<u>h</u>	
a16,0	b0,16	m	m	

**Συμπέρασμα:**

Το miss pattern είναι το ίδιο για κάθε επανάληψη του εξωτερικού loop

Misses =  $256 \cdot 288 = 73728$  , σε σύνολο  $2 \cdot 256 \cdot 256$  αναφορών => miss rate = 56.25%

# Εφαρμόστε blocking δίνοντας τον βελτιστοποιημένο κώδικα. Ποιο block size θα επιλέγατε ως καταλληλότερο και γιατί;

```
for(i=0; i<256; i++)  
    for(j=0; j<256; j++)  
        b[i][j] = a[j][i];
```

=>

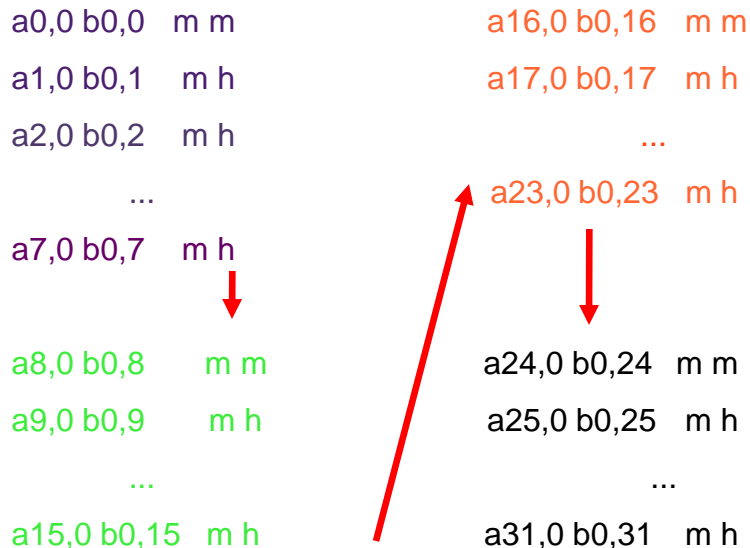
```
for(i=0; i<256; i+=bs)  
    for(j=0; j<256; j+=bs)  
        for(ii=i; ii<i+bs; ii++)  
            for(jj=j; jj<j+bs; jj++)  
                b[ii][jj] = a[jj][ii];
```

- Ποιο block size;
  - Τα 2 blocks πρέπει να χωράνε στην cache => κάθε block 8KB ή 1024 στοιχεία = 32x32 στοιχεία => bs =32

# Ποιο το ποσοστό αστοχίας για τον blocked κώδικα;

```
for(i=0; i<256; i+=32)
  for(j=0; j<256; j+=32)
    for(ii=i; ii<i+32; ii++)
      for(jj=j; jj<j+32; jj++)
        b[ii][jj] = a[jj][ii];
```

Για την 1η επανάληψη του loop “ii”:

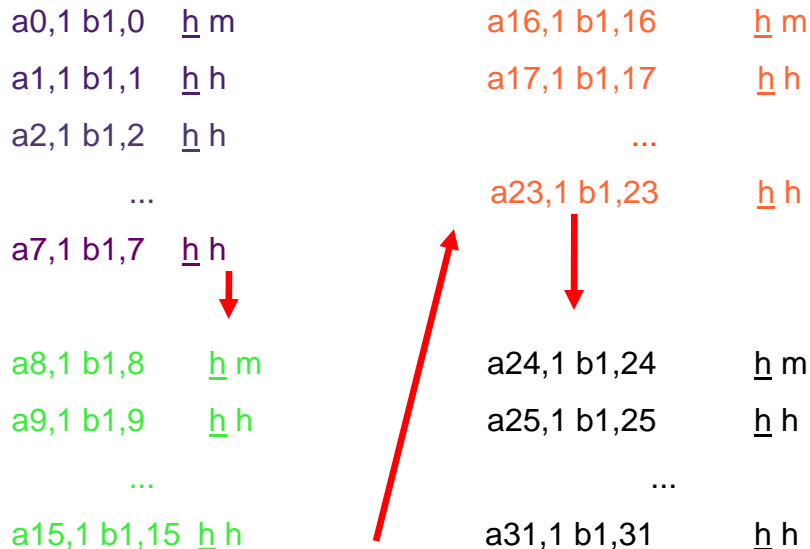


- ανά 8 επαναλήψεις του “jj” το miss pattern επαναλαμβάνεται =>  $4 \cdot 9 = 36$  misses συνολικά

# Ποιο το ποσοστό αστοχίας για τον *blocked* κώδικα;

```
for(i=0; i<256; i+=32)
  for(j=0; j<256; j+=32)
    for(ii=i; ii<i+32; ii++)
      for(jj=j; jj<j+32; jj++)
        b[ii][jj] = a[jj][ii];
```

Στην 2η επανάληψη του loop “ii”, θα έχουμε επαναχρησιμοποίηση στα στοιχεία του a:

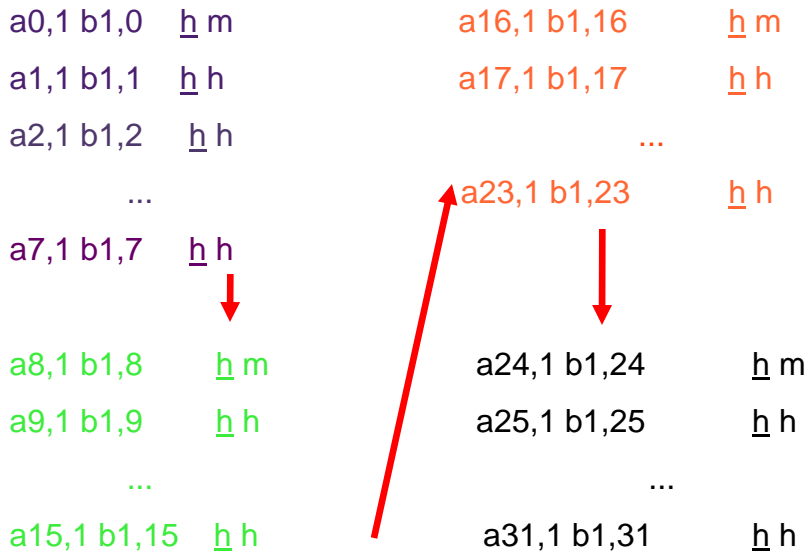


- Σε αυτήν την περίπτωση έχουμε συνολικά  $4*1=4$  misses

# Ποιο το ποσοστό αστοχίας για τον blocked κώδικα;

```
for(i=0; i<256; i+=32)
  for(j=0; j<256; j+=32)
    for(ii=i; ii<i+32; ii++)
      for(jj=j; jj<j+32; jj++)
        b[ii][jj] = a[jj][ii];
```

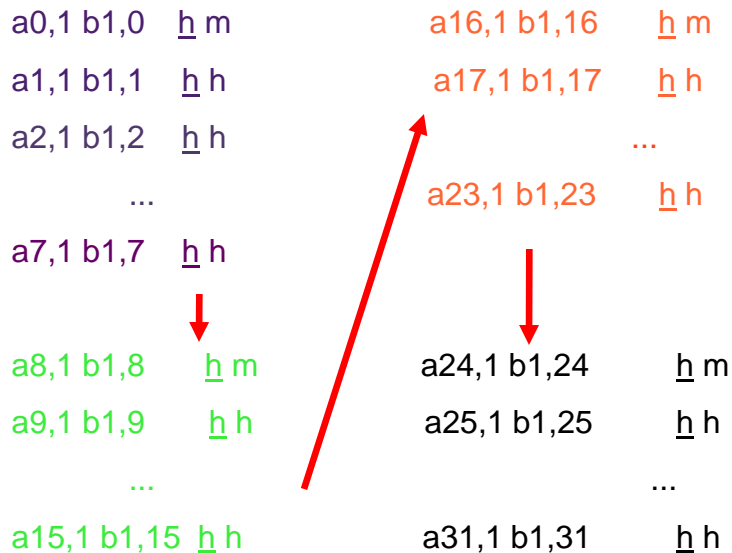
Η επαναχρησιμοποίηση στα στοιχεία του a θα συνεχιστεί για  $ii=2,3,\dots,7$  (γιατί;)



- Συνολικά, για  $ii=0,\dots,7$  θα έχουμε  $36+7*4=64$  misses
- ...το pattern αυτό θα επαναλαμβάνεται για  $ii=8\dots 15$ ,  $ii=16\dots 23$ ,  $ii=24\dots 31$  (όπου αλλάζει η στήλη για τον a ώστε να φορτώνεται καινούρια cache line)

# Ποιο το ποσοστό αστοχίας για τον *blocked* κώδικα;

```
for(i=0; i<256; i+=32)
  for(j=0; j<256; j+=32)
    for(ii=i; ii<i+32; ii++)
      for(jj=j; jj<j+32; jj++)
        b[ii][jj] = a[jj][ii];
```



- Συνολικά επομένως, για μια πλήρη εκτέλεση των 2 εσωτερικότερων *loops* (για την εκτέλεση δηλαδή του αλγορίθμου σε επίπεδο *block* πίνακα), θα έχουμε  $4 \cdot 64 = 256$  *misses* σε σύνολο  $2 \cdot 32 \cdot 32 = 2048$  αναφορών.
- Το *miss pattern* αυτό επαναλαμβάνεται για όλες τις επαναλήψεις των 2 εξωτερικότερων *loops*, για όλους δηλαδή τους συνδυασμούς *blocks* των πινάκων *a* και *b* (δεν υπάρχει επαναχρησιμοποίηση σε επίπεδο *block*, μόνο σε επίπεδο στοιχείων εντός του *block*).
- Επομένως, το *miss rate* =  $256/2048 = 12.5\%$

# Άσκηση 6η

Δίνεται η παρακάτω ακολουθία προσπελάσεων

Διεύθυνση (hex)	Αποτέλεσμα
0x0D8	Miss
0x0C8	Miss
0x0DC	Hit

- Μήκος διεύθυνσης 9 bits
- Συνολικό μέγεθος tag array 48 bits
- Ελάχιστη μονάδα δεδομένων που μπορεί να διευθυνσιοδοτηθεί το 1 byte
- 2-way set associative, πολιτική αντικατάστασης LRU
- Αρχικά η cache είναι άδεια

# Άσκηση 6η

(i) Βρείτε το μέγεθος της cache

•Tag array 48 bits	48 bits x 1 block	X
	24 bits x 2 blocks	X
	16 bits x 3 blocks	X
	12 bits x 4 blocks	X
	8 bits x 6 blocks	X
	6 bits x 8 blocks	
	4 bits x 12 blocks	X
	3 bits x 16 blocks	
	2 bits x 24 blocks	X
1 bit x 48 block	X	



# Άσκηση 6η

- Για tag μήκους 6 bits και 8 blocks (index = 2)

	TAG	I
0x0D8 : miss	0 1101	1000
0x0C8 : miss	0 1100	1000
0x0DC : <del>hit</del>	0 1101	1100

} ≠

- Για tag μήκους 3 bits και 16 blocks (index = 3)

	TAG	IND
0x0D8 : miss	0 11	01 1000
0x0C8 : miss	0 11	00 1000
0x0DC : hit	0 11	01 1100

} ✓

- Συνολικά 16 blocks μεγέθους 8 bytes (?) = 128 bytes cache

# Άσκηση 6η

- 0x151 → 1 0101 0001 → miss
- 0x155 → 1 0101 0101
- 0x020 → 0 0010 0000
- 0x191 → 1 0001 0001
- 0x1d4 → 1 1101 0100
- 0x153 → 1 0101 0011
- 0x123 → 1 0010 0011
- 0x021 → 1 0010 0001

set	tag
000	
001	
010	101
011	
100	
101	
110	
111	

# Άσκηση 6η

- 0x151 → 1 0101 0001 → miss
- 0x155 → 1 0101 0101 → hit
- 0x020 → 0 0010 0000
- 0x191 → 1 0001 0001
- 0x1d4 → 1 1101 0100
- 0x153 → 1 0101 0011
- 0x123 → 1 0010 0011
- 0x021 → 1 0010 0001

set	tag
000	
001	
010	<b>101</b>
011	
100	
101	
110	
111	

# Άσκηση 6η

- 0x151 → 1 0101 0001 → miss
- 0x155 → 1 0101 0101 → hit
- 0x020 → 0 0010 0000 → miss
- 0x191 → 1 0001 0001
- 0x1d4 → 1 1101 0100
- 0x153 → 1 0101 0011
- 0x123 → 1 0010 0011
- 0x021 → 1 0010 0001

set	tag
000	
001	
010	101
011	
100	<b>000</b>
101	
110	
111	

# Άσκηση 6η

- 0x151 → 1 0101 0001 → miss
- 0x155 → 1 0101 0101 → hit
- 0x020 → 0 0010 0000 → miss
- 0x191 → 1 0001 0001 → miss
- 0x1d4 → 1 1101 0100
- 0x153 → 1 0101 0011
- 0x123 → 1 0010 0011
- 0x021 → 1 0010 0001

set	tag	
000		
001		
010	101	<b>100</b>
011		
100	000	
101		
110		
111		

# Άσκηση 6η

- 0x151 → 1 0101 0001 → miss
- 0x155 → 1 0101 0101 → hit
- 0x020 → 0 0010 0000 → miss
- 0x191 → 1 0001 0001 → miss
- 0x1d4 → 1 1101 0100 → miss
- 0x153 → 1 0101 0011
- 0x123 → 1 0010 0011
- 0x021 → 1 0010 0001

set	tag	
000		
001		
010	111	100
011		
100	000	
101		
110		
111		

# Άσκηση 6η

- 0x151 → 1 0101 0001 → miss
- 0x155 → 1 0101 0101 → hit
- 0x020 → 0 0010 0000 → miss
- 0x191 → 1 0001 0001 → miss
- 0x1d4 → 1 1101 0100 → miss
- 0x153 → 1 0101 0011 → miss
- 0x123 → 1 0010 0011
- 0x021 → 1 0010 0001

set	tag	
000		
001		
010	111	101
011		
100	000	
101		
110		
111		

# Άσκηση 6η

- 0x151 → 1 0101 0001 → miss
- 0x155 → 1 0101 0101 → hit
- 0x020 → 0 0010 0000 → miss
- 0x191 → 1 0001 0001 → miss
- 0x1d4 → 1 1101 0100 → miss
- 0x153 → 1 0101 0011 → miss
- 0x123 → 1 0010 0011 → miss
- 0x021 → 1 0010 0001

set	tag	
000		
001		
010	111	101
011		
100	000	<b>100</b>
101		
110		
111		



# Άσκηση 6η

- 0x151 → 1 0101 0001 → miss
- 0x155 → 1 0101 0101 → hit
- 0x020 → 0 0010 0000 → miss
- 0x191 → 1 0001 0001 → miss
- 0x1d4 → 1 1101 0100 → miss
- 0x153 → 1 0101 0011 → miss
- 0x123 → 1 0010 0011 → miss
- 0x021 → 1 0010 0001 → hit

set	tag	
000		
001		
010	111	101
011		
100	000	<b>100</b>
101		
110		
111		