זיכרון מטמון 1

מבנה מחשבים ספרתיים
234267

הבעיה

מהירות הגישה לזיכרון איטית ביחס לביצועי המעבד (עד פי 1000), ככל שהזיכרון גדול יותר הגישה אליו איטית יותר.

ביצועי המעבד נפגעים משמעותית אם בכל קריאה מהזיכרון יש להמתין (ולהשבית את המעבד) מספר רב של מחזורי שעון.

הפתרון

החזקת עותק חלקי של הזכרון "קרוב" למעבד כך שזמן הגישה אליו יהיה קצר בהרבה.

למה הפתרון הזה טוב?

- מקומיות בזמן – אם ניגשנו לאובייקט מסויים, סביר להניח שניגש אליו שוב בקרוב.
  - אנו מבלים 90% מהזמן ב- 10% מהקוד – בעקר עקב לולאות בהן קוראים אותה שורה מספר פעמים.
  - משתנים מסוימים מעודכנים פעם אחר פעם. לדוגמא – מיון.

- מקומיות במקום – אם ניגשנו לאובייקט מסויים, סביר להניח שניגש לאובייקטים סמוכים אליו.
  - קטעי קוד, סביר מאד שנצטרך גם את הפקודה הבאה ואילו שאחריה.
  - נתונים – אם קראנו (או כתבנו) ממקום אחד במערך סביר להניח שנקרא (או נכתוב) גם את סביבתו.

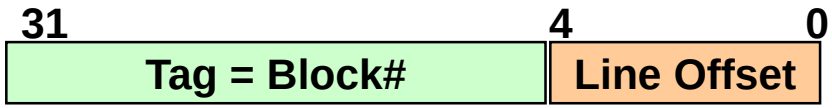
תזכורת לחוק אמדל – עדיף לשפר את מה שקורה רוב הזמן !

טרמינולוגיה

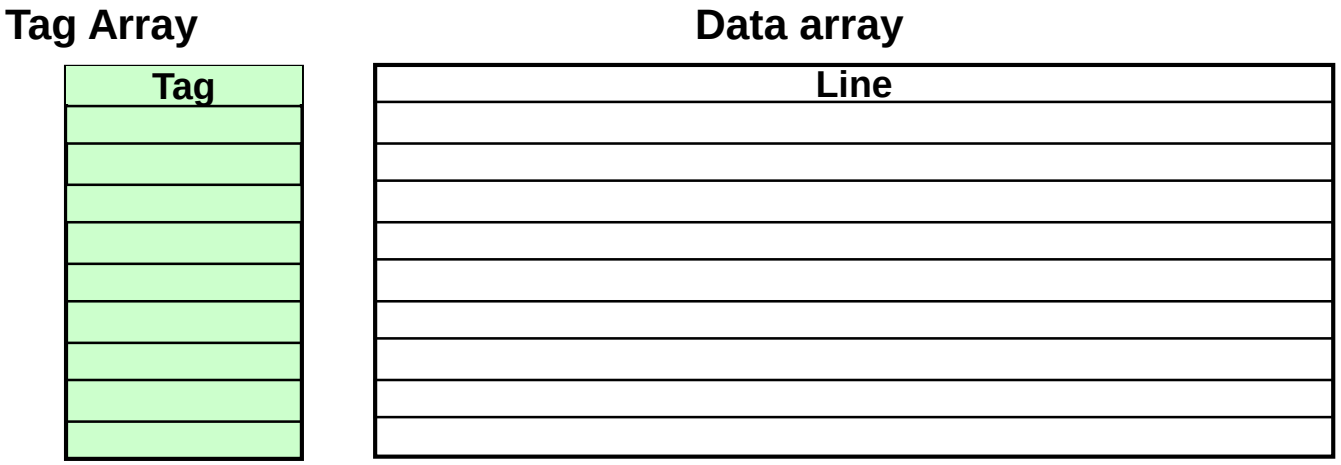
- **פגיעה** (Hit)
  - הנתון מופיע ברמת הזכרון.
- **החטאה** (Miss)
  - הנתון לא מופיע ברמת הזיכרון ולכן צריך להביאו מרמה נמוכה יותר.
- **יחס פגיעה** (hit rate)
  - אחוז הפגיעות מתוך סה"כ הגישות לזכרון.
  - Miss rate = 1 – hit rate
- **בלוק**
  - הזכרון המרכזי מחולק לבלוקים של מספר בתים. בכל פעם שנצטרך להעתיק בית למטמון נעתיק את כל הבלוק בו הוא נמצא.

Fully associative Organization

1. נחלק את שדה כתובת המידע למספר הבלוק (tag – מספר זיהוי) ולמיקום המידע בתוך הבלוק (offset).

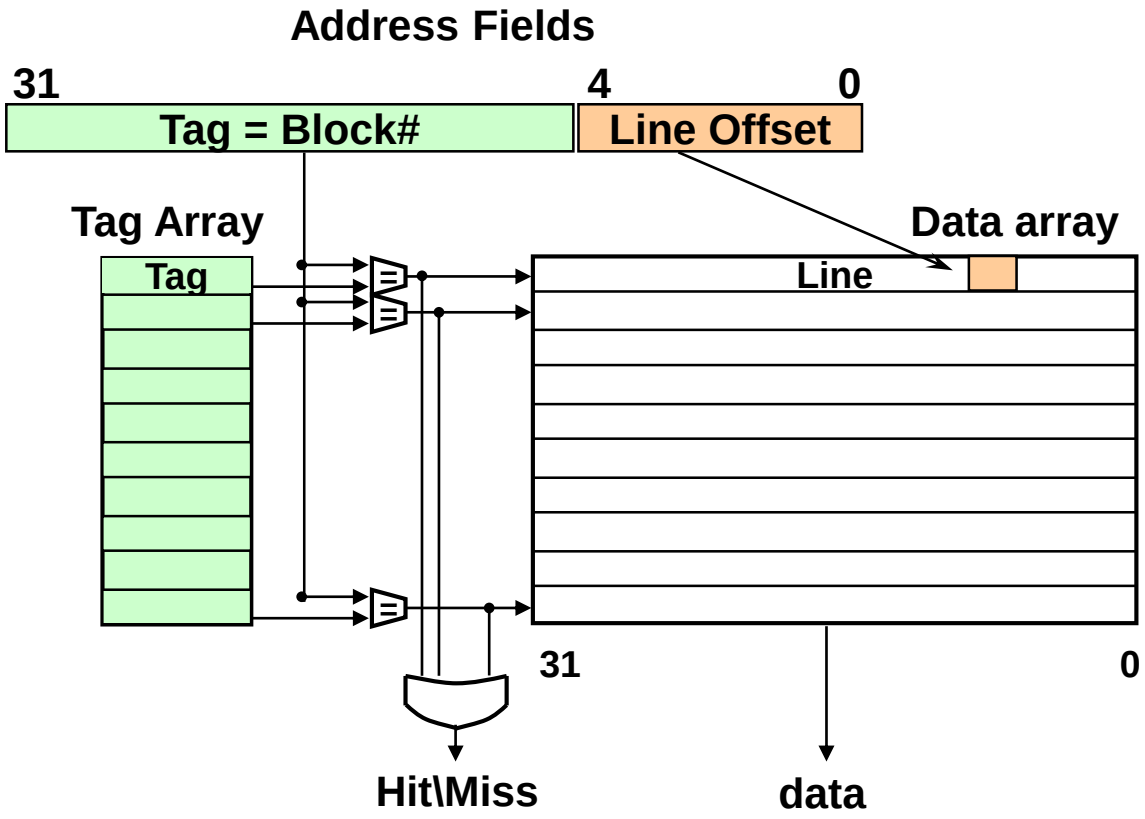


2. המטמון יהיה בנוי כטבלה של **מספר הבלוק** ו- **תוכן הבלוק**.

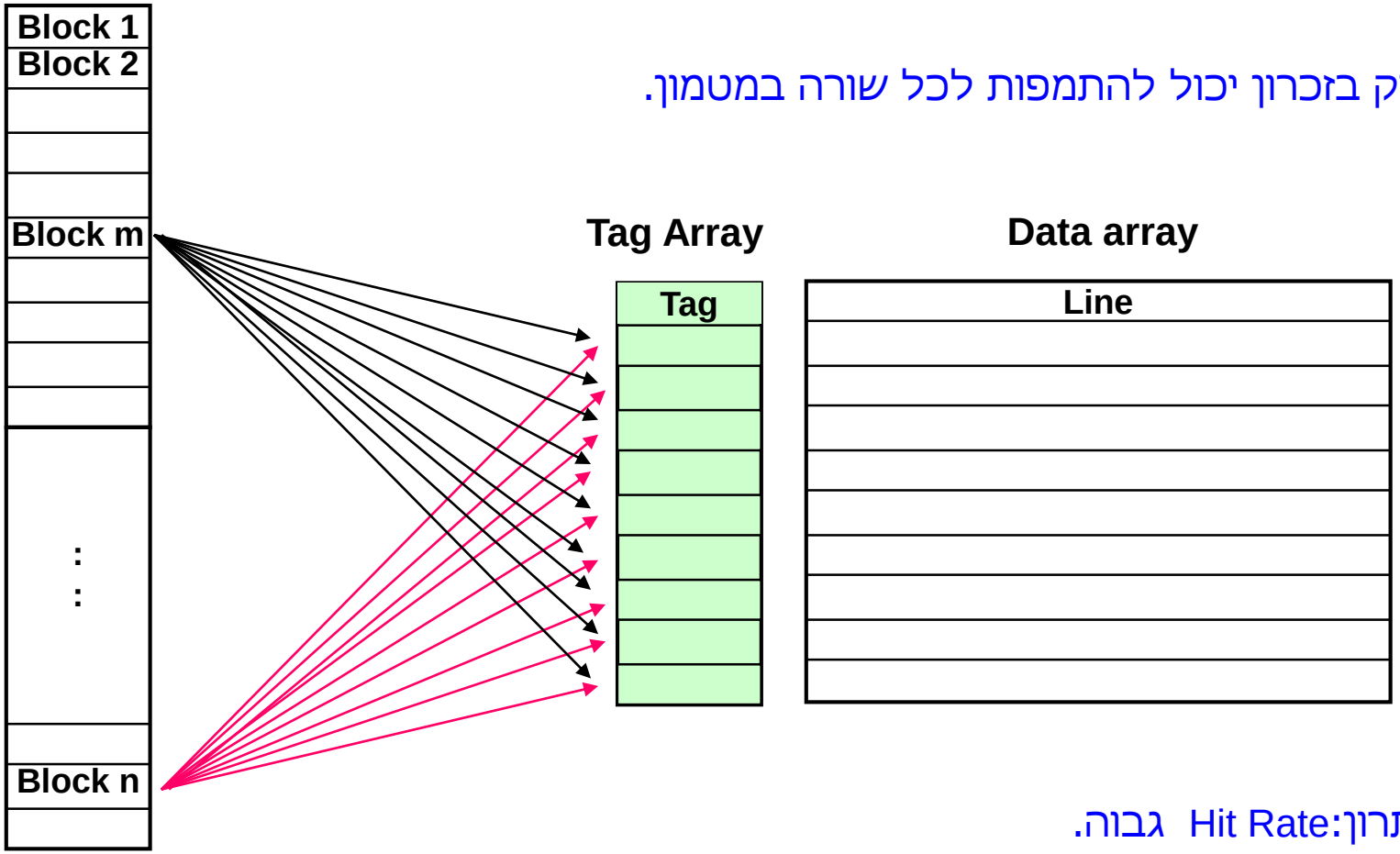


כל בלוק בזכרון יכול להכנס לכל שורה במטמון.

Fully associative Organization



Fully associative Organization



כל בלוק בזכרון יכול להתמפות לכל שורה במטמון.

- יתרון:Hit Rate גבוה.
- חסרון: מימוש ההשוואה המקבילית קשה ויקר.

מדיניות פינוי

במטמון יהיו הרבה פחות שורות מאשר בלוקים בזכרון הראשי

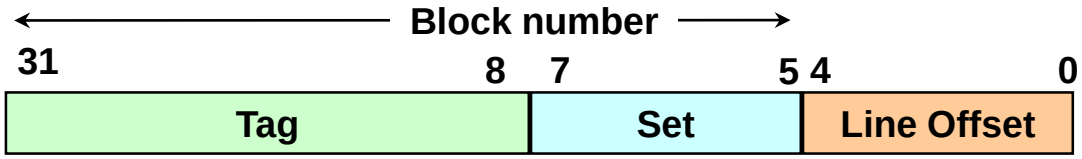
כאשר יש צורך להביא בלוק נוסף למטמון יהיה צורך לזרוק את אחד הבלוקים הקודמים.

יש מספר אפשרויות לבחור מי יהיה הבלוק שייזרק:

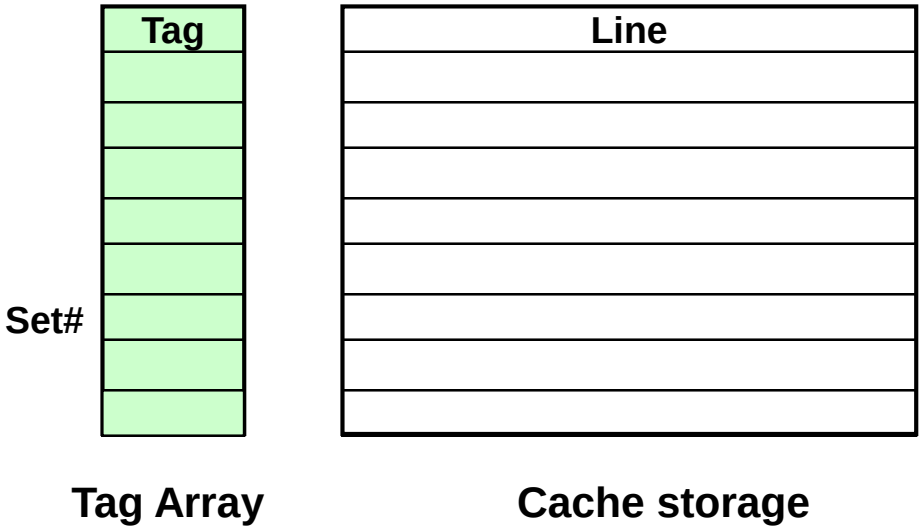
1. LRU - least recently used
   - הבלוק שזמן רב ביותר לא השתמשנו בו (לקריאה או לכתיבה).
2. LRM - least recently modified
   - הבלוק שזמן רב ביותר לא כתבנו אליו.
3. Random
   - בחירה אקראית לחלוטין.

Direct Mapping Organization

1. נחלק את שדה כתובת המידע למיקום המידע בתוך הבלוק (offset) ואת מספר הבלוק נחלק למיקום במטמון (set) ולמספר זיהוי (tag).



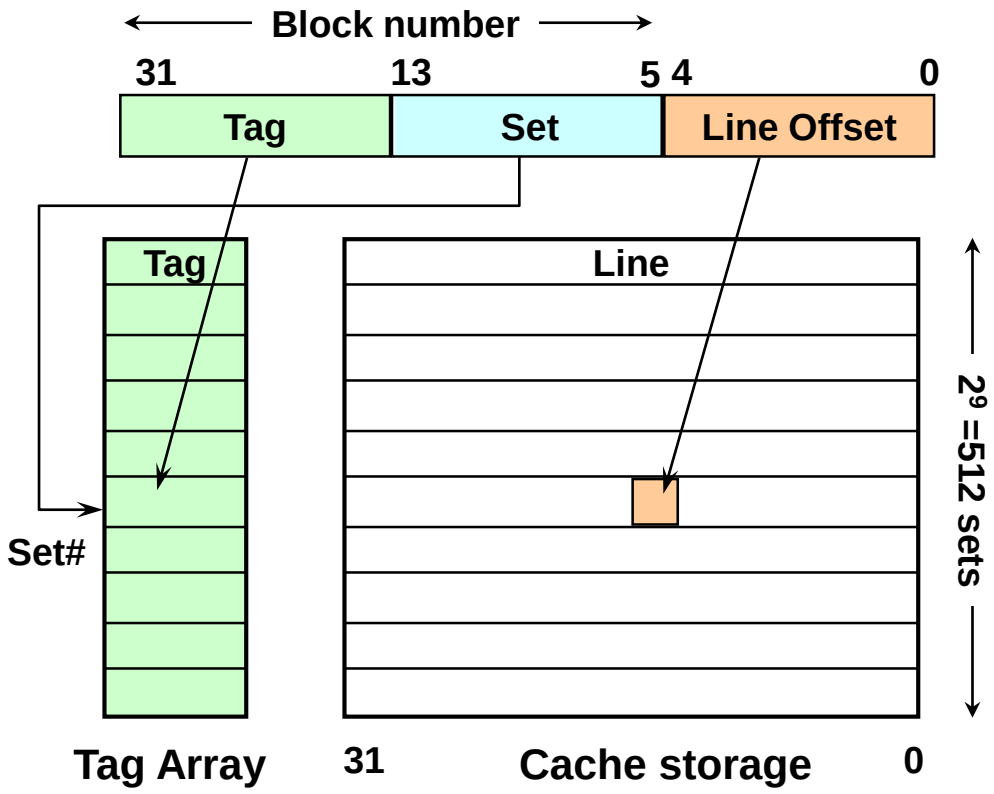
2. המטמון יהיה בנוי כטבלה של **מספר הבלוק** ו- **תוכן הבלוק**.



ההבדל ממבנה של fully associative cache הוא שהפעם לכל שורה במטמון יש מספר מזהה – set.

Direct Mapping Organization

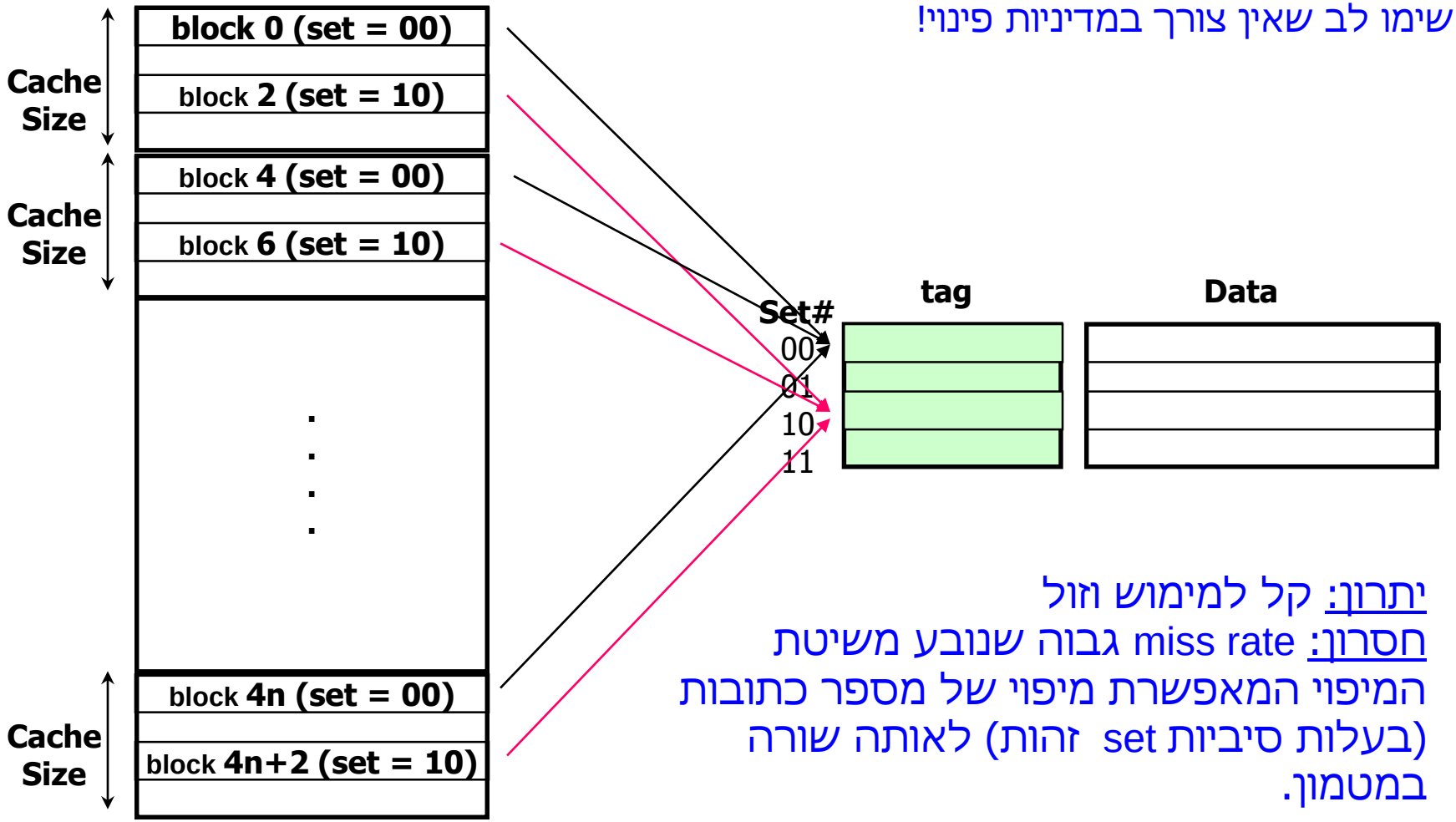
3. מציאת הנתון:
א. נמצא את השורה המתאימה עפ"י ה- set כאינדקס במערך.
ב. נשווה את ה- tag של הכתובת עם ה- tag של השורה במטמון אליה הצביע ה- set.
במידה ויש התאמה הבלוק המבוקש נמצא במטמון. אחרת יש החטאה וצריך להביא את הבלוק.
ג. במידה והנתון נמצא במטמון, ניגש ל- byte המתאים בשורה עפ"י שדה ה- disp.



Direct Mapping Organization

כל שורה בזכרון יכולה להמצא בשורה אחת במטמון.

שימו לב שאין צורך במדיניות פינוי!

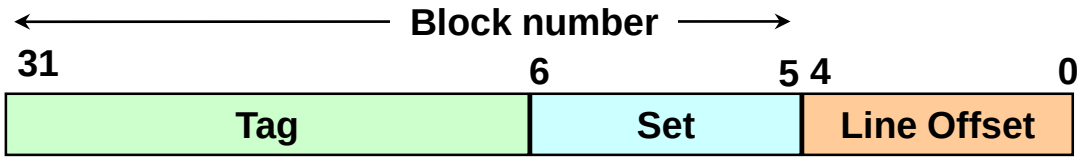


יתרון: קל למימוש וזול

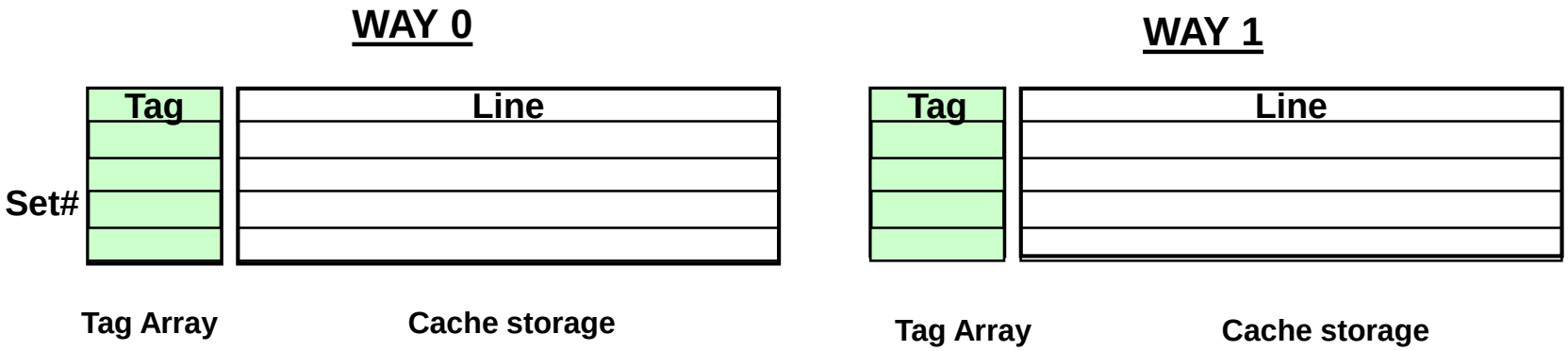
חסרון: miss rate גבוה שנובע משיטת המיפוי המאפשרת מיפוי של מספר כתובות (בעלות סיביות set זהות) לאותה שורה במטמון.

2 way set associative Organization

1. באופן זהה ל- direct mapping נחלק את שדה כתובת המידע למיקום המידע בתוך הבלוק (offset) ואת מספר הבלוק נחלק למיקום במטמון (set) ולמספר זיהוי (tag).



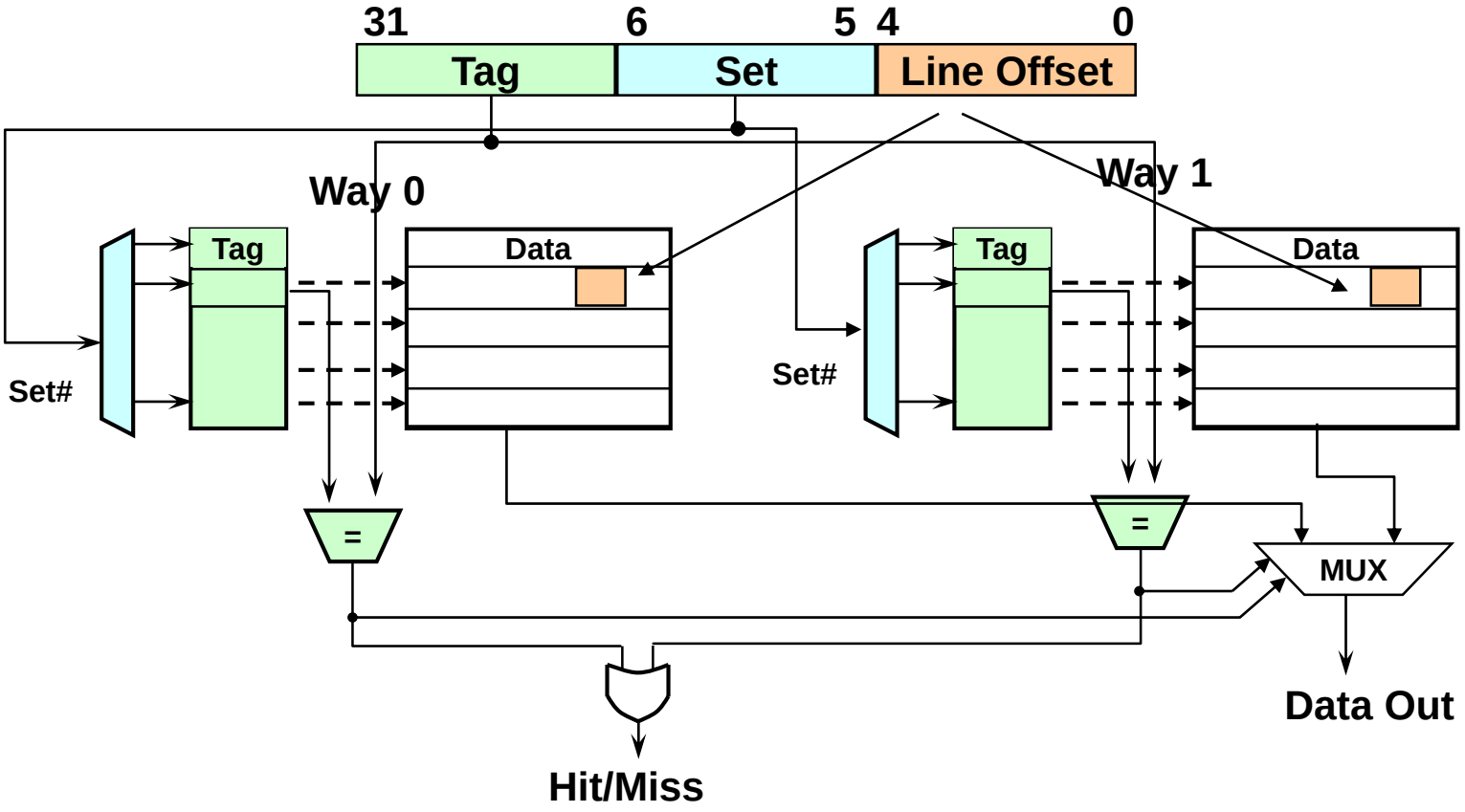
2. המטמון יהיה בנוי **כשתי** טבלאות של **מספר הבלוק** ו- **תוכן הבלוק**.



ההבדל ממבנה של direct mapping הוא שהפעם יש שני ways, כל אחד בנוי כמו direct mapping כאשר הנתון יכול להמצא בכל אחד מהם.

2 way set associative Organization

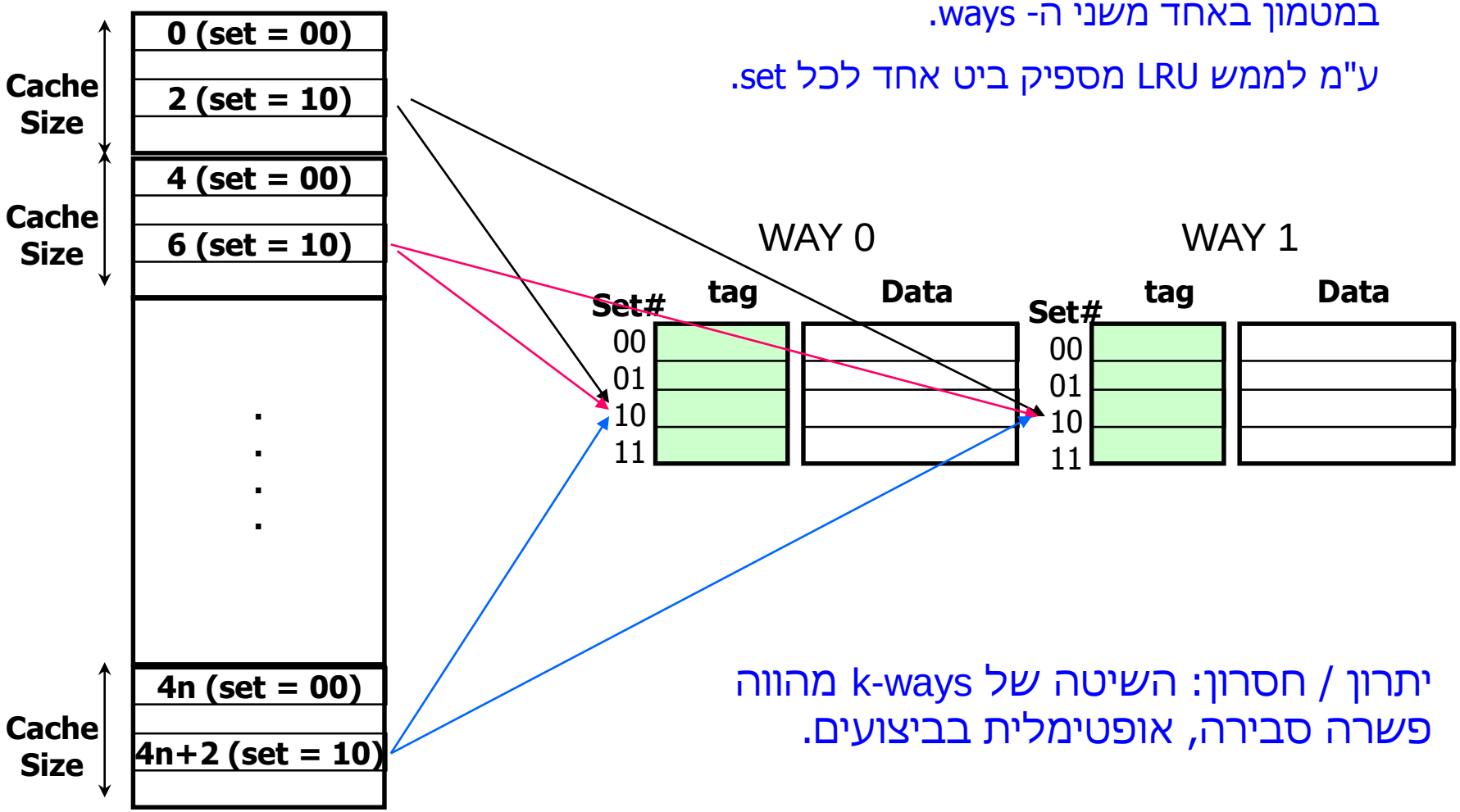
3. מציאת הנתון:
א. נמצא את השורה המתאימה עפ"י ה- set כאינדקס במערך.
ב. נשווה את ה- tag של הכתובת הדרושה עם ה- tag של השורה במטמון אליה הצביע ה- set בכל אחד מה- ways. במידה ויש התאמה הבלוק המבוקש נמצא במטמון. אחרת יש החטאה וצריך להביא את הבלוק.
ג. במידה והנתון נמצא במטמון, ניגש ל- byte המתאים בשורה עפ"י שדה ה- disp.



2 way set associative Organization

כל שורה בזכרון יכולה להמצא בשורה אחת
במטמון באחד משני ה- ways.

ע"מ לממש LRU מספיק ביט אחד לכל set.

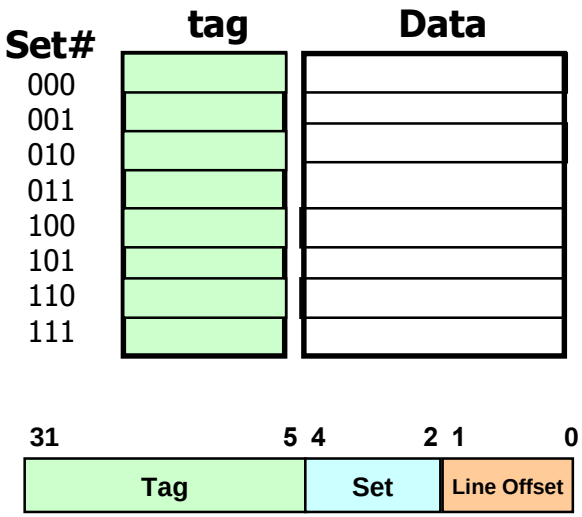


יתרון / חסרון: השיטה של k-ways מהווה
פשרה סבירה, אופטימלית בביצועים.

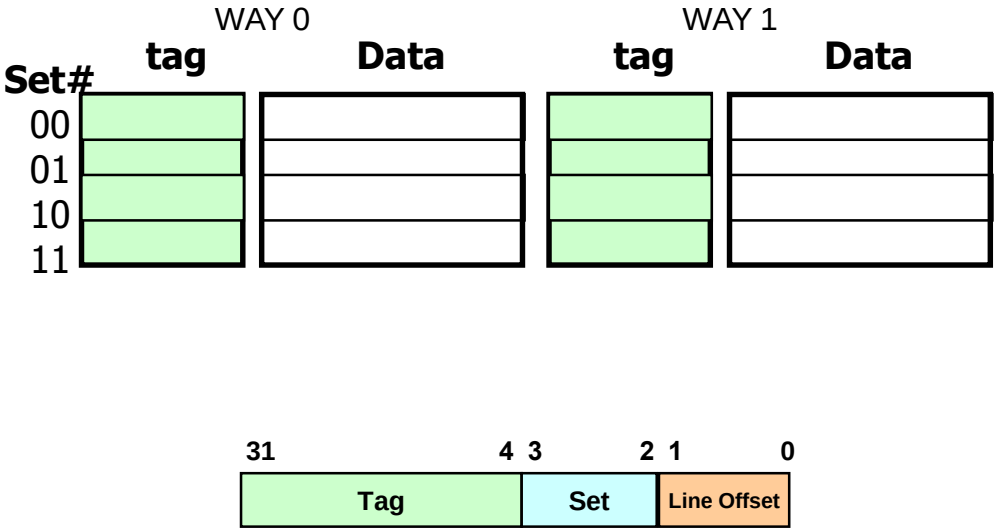
K way set associative Organization

נניח מטמון בעל קיבולת של 8 בלוקים (DATA) 4 בתים
כ"א ואסוציאטיביות משתנה

1 way set associative
(direct mapping)

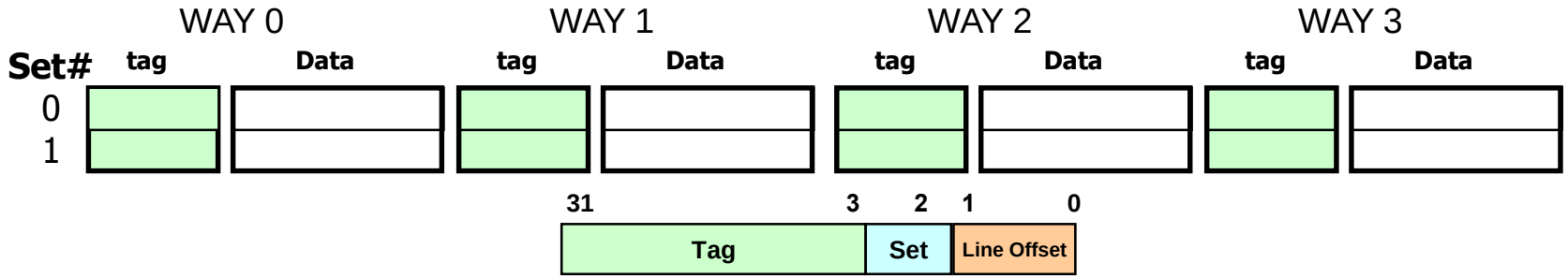


2 way set associative

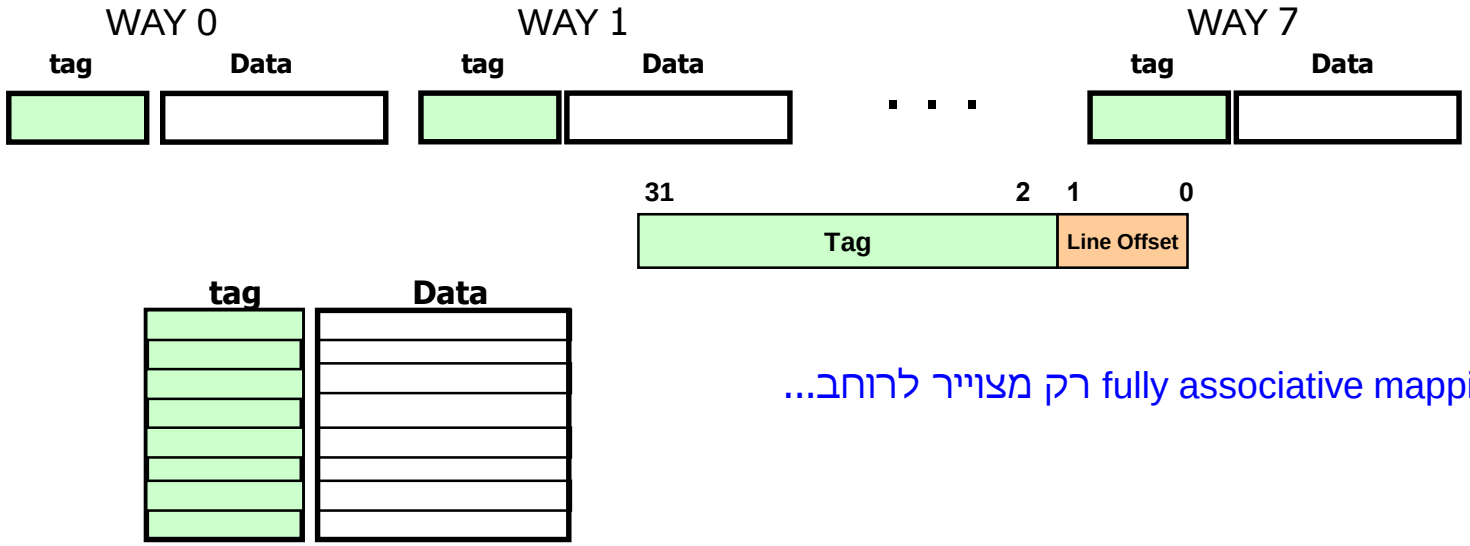


K way set associative Organization (II)

4 way set associative



8 way set associative

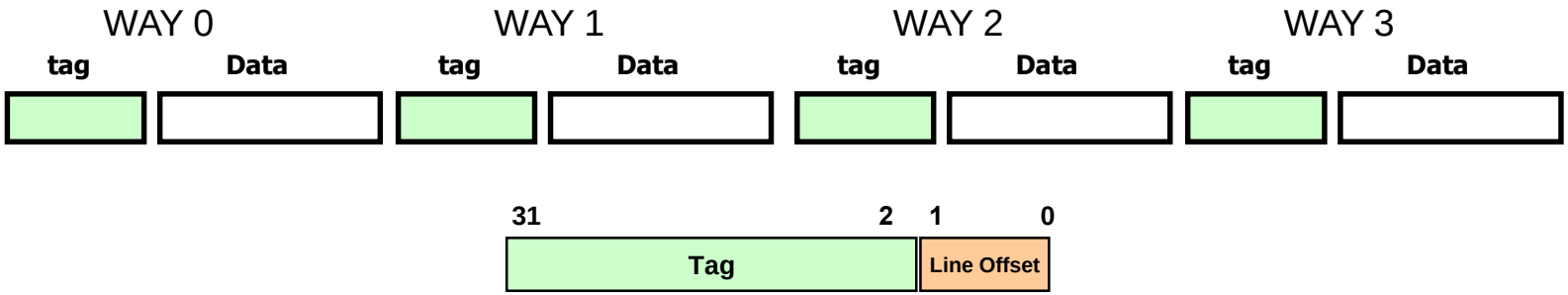


זהו בעצם fully associative mapping רק מצוייר לרוחב...

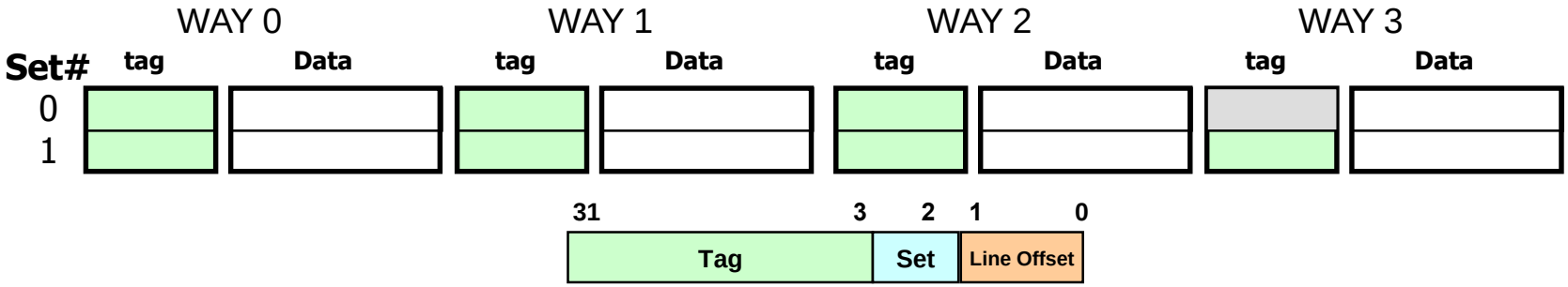
K way set associative Organization (III)

נניח מטמון בעל אסוציאטיביות של 4 ונפח data משתנה

4 בלוקים

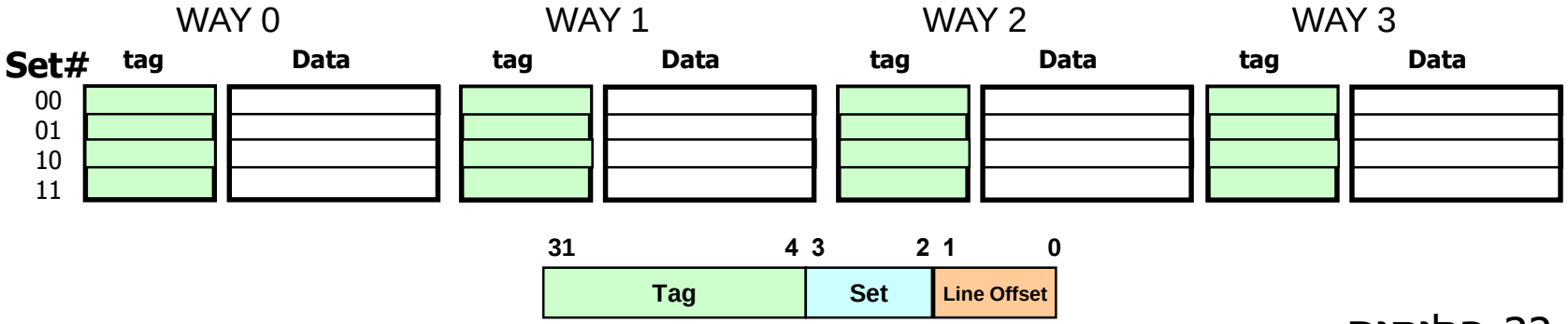


8 בלוקים

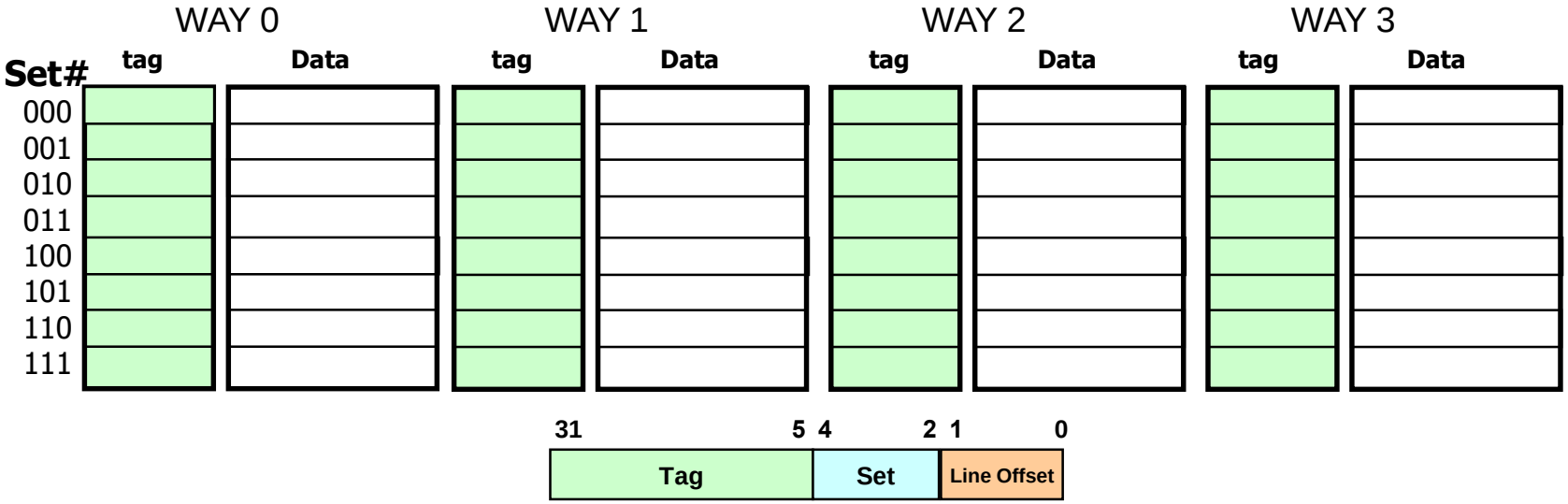


K way set associative Organization (IV)

16 בלוקים



32 בלוקים



עדכון למטמון: Write Back

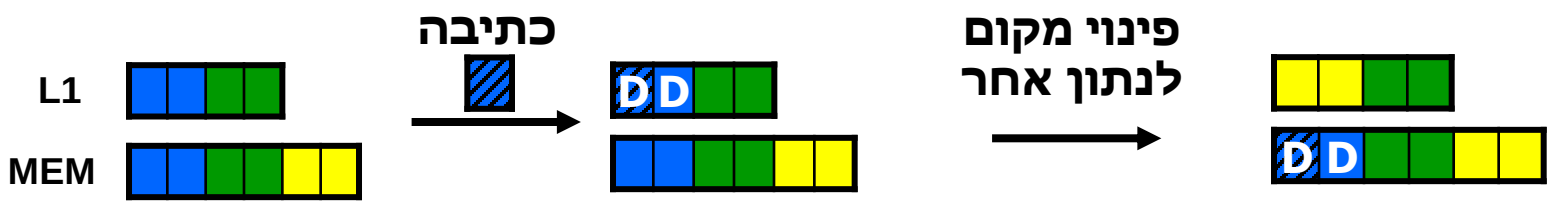
נניח שיש צורך **לעדכן** נתון מסויים. במידה והנתון **נמצא** במטמון, כותבים את הערך החדש לבלוק המתאים במטמון. נותרת השאלה מתי נעדכן את הערך השמור בזיכרון הראשי. הדבר תלוי במדיניות הכתיבה.

במדיניות WB, בזמן כתיבה נכתוב רק למטמון. העדכון לזיכרון יבוצע כאשר הנתון ייזרק מהמטמון.

על מנת לממש מדיניות זאת לכל בלוק תשמר סיבית מיוחדת dirty שתהווה אינדיקציה האם הבלוק עודכן וטרם נכתב לרמת זכרון נמוכה יותר.

כאשר נתבקש לפנות בלוק מסוים, נעדכן בלוק זה ברמת הזכרון הנמוכה יותר במידה וסיבית ה- dirty דולקת.

בכל עדכון נצטרך לעדכן את כל הבלוק מכיוון שסיבית ה- dirty מוחזקת עבור הבלוק כולו ולא עבור byte מסויים.



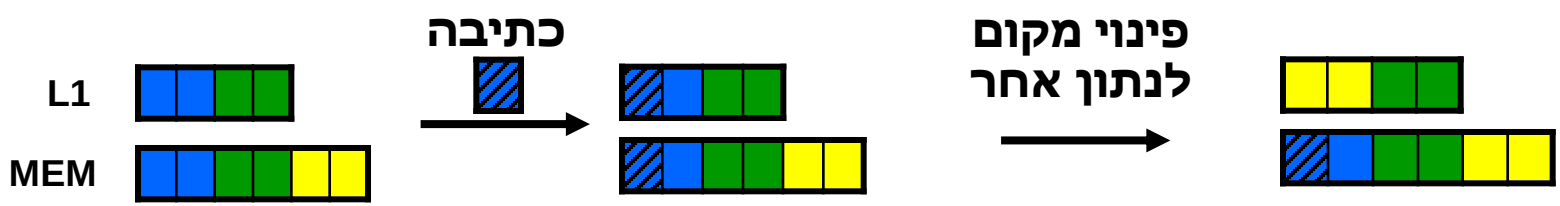
הציור – עבור מטמון (fully associative, 2 מילים בבלוק) בעל רמה אחת שפועלות במדיניות כתיבה writeback.

עדכון למטמון: Write Through

מדיניות כתיבה נוספת היא מדיניות Write through

בזמן כתיבה נכתוב את הערך החדש של הנתון גם למטמון וגם לזיכרון

כאשר הנתון ייזרק מהמטמון, אין צורך לעדכן את הזיכרון.



הציור – עבור מטמון (fully associative, 2 בתים בבלוק) בעל רמה אחת שפועל במדיניות כתיבה write through.

<u>שימו לב!</u>

בשיטת write through אין צורך בסיבית dirty (למה ?)

סיבית סטטוס נוספת שנמצאת עבור כל בלוק במטמון (בד"כ ללא קשר למדיניות) היא סיבית Valid.
בלוק שנמצא במצב I (Invalid) הוא בלוק שבו המידע שנמצא בשורת ה- data אינו משקף את המידע שנמצא בבלוק בעל tag מתאים ברמות נמוכות יותר.

כתיבה למטמון: Write Allocate

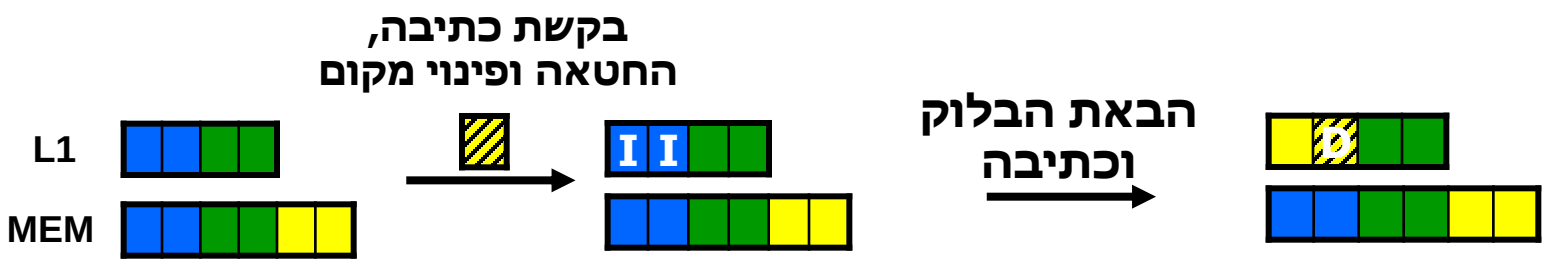
נניח שיש צורך **לעדכן** נתון מסויים **שלא נמצא** במטמון , מצב זה נקראת החטאת כתיבה.

ישנן שתי אפשרויות עיקריות לטיפול במקרה זה:

Write allocate

במקרה של החטאה שולחים לזיכרון בקשה להבאת הבלוק.

טרם הבאת הבלוק מהרמה התחתונה מפנים את המקום המתאים (מבחינת set ומדיניות החלפה).

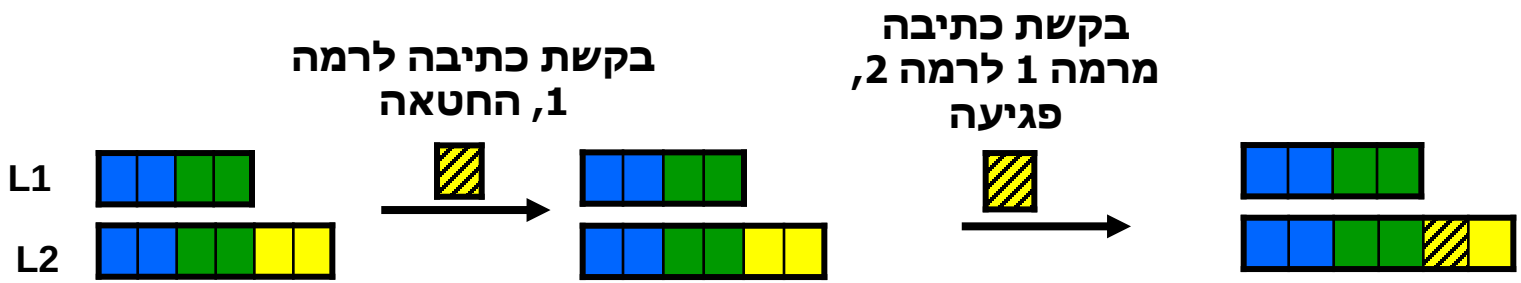


הציור – עבור מטמון (fully associative, 2 בתים בבלוק) בעל רמה אחת שפועל במדיניות כתיבה writeback ומדיניות write miss של Write allocate.

כתיבה למטמון: No Write Allocate

מדיניות נוספת למצב של החטאת כתיבה היא No Write allocate

במקרה של החטאה מורידים לזיכרון את בקשת הכתיבה עצמה. אין הבאה של הבלוק לרמה הנדרשת.



הציור – עבור מטמון (fully associative, 2 בתים בבלוק) בעל שתי רמות שפועלות במדיניות כתיבה write through ומדיניות write miss של No Write allocate ברמה הראשונה.

סוגי החטאות

Compulsory – הבלוק לא היה מעולם בשימוש עד עכשיו, בהכרח הוא לא יימצא במטמו (cold miss)

Conflict – הבלוק כבר היה בשימוש אך היות והמטמון אינו fully associative בלוק אחר תפס את מקומו.

- LRU policy and mapping affects conflict misses

Capacity – הבלוק כבר היה בשימוש אך היות ומאז נקראו יותר בלוקים ממה שהמטמון היה יכול להכיל, בלוק אחר תפס את מקומו. גם אם המטמון היה fully-associative הייתה נוצרת כאן החטאה.

- Associativity can't help

דוגמא

נתון מטמון 2-way set associative במדיניות פינוי LRU . הכתובת מחולקת באופן הבא ,

Tag	Set	offset
29 … 5	4 … 2	1 0

למטמון הנתון נשלחה סדרת הכתובות הבאה (גישה לבתים):

Addresses: 5 7 1 4 36 8 100 6 4 12 36 12 68 5 7

בהנחה שמדיניות הפינוי היא LRU והמטמון מלא ב- data של בלוקים שאינם מופיעים בסדרה דלעיל מצא:

כמה החטאות יווצרו?

מה מכיל המטמון בתום הסדרה (קרי – אילו בלוקים בכל set)?

כדי לפתור את השאלה נצטרך לחשב לאיזה set במטמון נכניס את הבלוק המבוקש מהזכרון הראשי.

את שדה ה- set ניתן לחשב לפי

Set = floor(ADDRESS / line length) mod (lines per way)

הסבר: חלוקת הכתובת ב- 4 (אורך השורה) תוריד את שדה ה- offset, וביצוע mod8 יתן את שלושת סיביות ה- LSB של מספר הבלוק שהן בעצם ה- set.

נימוק	Way /LRU	Hit/ miss	set	tag	כתובת	
				0	**5**	1
				0	**7**	2
				0	**1**	3
				0	**4**	4
				1	**36**	5
				0	**8**	6
				3	**100**	7
				0	**6**	8
				0	**4**	9
				0	**12**	10
				1	**36**	11
				0	**12**	12
				2	**68**	13
				0	**5**	14
				0	**7**	15

	כתובת	tag	set	Hit/ miss	Way /LRU	נימוק
1	**5**	0	1	**M**	0/1	עפ"י הנתון השורה לא נמצאת
2	**7**	0	1	**H**	0/1	בפקודה הקודמת הבאנו שורה שמכילה גם את כתובת 7 (וגם את 4 ו- 6)
3	**1**	0	0	**M**	0/1	עפ"י הנתון השורה לא נמצאת
4	**4**	0	1	**H**	0/1	הנתון הובא ב- 1
5	**36**	1	1	**M**	1/0	הנתון לא נמצא. היות וכבר הובא נתון בעל set=1(ב- 1), way = 1.
6	**8**	0	2	**M**	0/1	עפ"י הנתון השורה לא נמצאת
7	**100**	3	1	**M**	0/1	בפעם האחרונה בה קבלנו set = 1 השורה הובאה ל- way1 ולכן עפ"י LRU צריך לפנות את way0.
8	**6**	0	1	**M**	1/0	אמנם השורה המכילה את כתובת 6 הובאה כבר ב- (2) אבל ב- (7) החלפנו אותה בשורה חדשה ולכן החטאה. בפעם האחרונה בה קבלנו set = 1 השורה הובאה ל- way0 ולכן עפ"י LRU צריך לפנות את way1.
9	**4**	0	1	**H**	1/0	ב- (8) שוב הבאנו את השורה של 4-7
10	**12**	0	3	**M**	0/1	עפ"י הנתון השורה לא נמצאת
11	**36**	1	1	**M**	0/1	בפעם האחרונה בה קבלנו set = 1 השורה הובאה ל- way1 ולכן עפ"י LRU צריך לפנות את way0.
12	**12**	0	3	**H**	0/1	הנתון כבר הובא ב- (10).
13	**68**	2	1	**M**	1/0	בפעם האחרונה בה קבלנו set = 1 השורה הובאה ל- way0 ולכן עפ"י LRU צריך לפנות את way1.
14	**5**	0	1	**M**	0/1	אמנם השורה המכילה את כתובת 5 הובאה שוב ב- (8) אבל ב- (13) החלפנו אותה בשורה חדשה ולכן החטאה. בפעם האחרונה בה קבלנו set = 1 השורה הובאה ל- way1 ולכן עפ"י LRU צריך לפנות את way0.
15	**7**	0	1	**H**	0/1	ב- (14) שוב הבאנו את השורה של 4-7

שאלה

לפניך קטע הקוד הבא, המשמש לאתחול מערך:

```
int array[1024];

for ( int i=0 ; i<1024 ; i++ )
    array[i] = 0;
```

הנחות:

1. המשתנה i והמצביע array מאוחסנים ברגיסטרים.
2. משתנה מסוג int תופס 4 בתים בזיכרון, הוא aligned (מיושר) בזיכרון.
המערך מיושר אף הוא יחסית לשורת מטמון (מתחיל בכתובת שהיא כפולה של 16)
3. למערכת זיכרון מטמון נפרד לקוד, לנתונים וכן TLB
4. גודל מטמון הנתונים 1KB (עבור data) והוא מאורגן בשיטת 4 way-set-associative,
גודל כל בלוק 16 בתים, מדיניות כתיבה write through, מדיניות write-allocate
ומדיניות פינוי random.
5. ניתן להניח כי לא מתבצעת החלפת תהליכים במהלך ריצת התוכנית.
6. גודל כתובת 32 ביט.

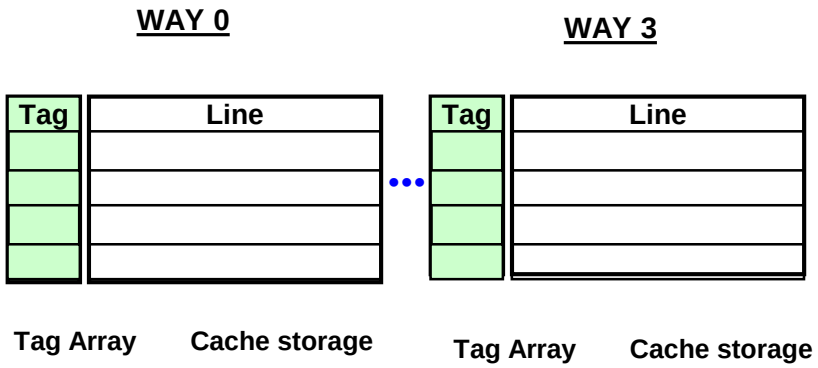
- I, array in register
- int is 4 bytes (aligned)
- iCache:
- dCache: 1KB – 4 way set associative – 16B block
  - write through – write allocate - Random
- 32 Bit address

Cache directory size : שאלה

א. כמה סיביות בסה"כ מכיל ה-cache directory של מטמון הנתונים?

$$\#blocks = \frac{\text{cache size}}{\text{block size}} = \frac{2^{10}}{2^{4}} = 2^{6} blocks$$

$$\#sets = \frac{\#blocks}{\#ways} = \frac{2^{6}}{2^{2}} = 2^{4} sets$$



- 16 byte blocks : 4 bits , 2^4 sets : 4 bits ➔ tag = 32 – 8 = 24 bits

ה- cache directory מכיל את 24 סיביות ה- tag וסיבית valid (אין סיבית modified כי המדיניות היא write-through ואין סיבית LRU כי מדיניות הפינוי היא Random).
מכיוון שמדובר ב- 4 way cache כשבכל way יש 16 sets, גודל ה- cache directory הוא:

Cache Directory Size = (	24	+	1	) *	4	*	16	= 1600
	tag		Valid		ways		sets	

```
int array[1024];
for ( int i=0 ; i<1024 ; i++ )
      array[i] = 0;

- 16 sets / 4 ways = 1KB
```

how many misses: שאלה

ב. נתון כי בתחילת הרצת קטע הקוד מטמון הנתונים ריק.
מהו מספר ה- cache misses המקסימאלי שאירעו
במטמון הנתונים במהלך ריצת קטע הקוד?

כל int תופס בזיכרון 4 בתים, לכן יש צורך לעבור על 4*1024 = 4096 בתים באופן רציף. מספר ה- misses יהיה כמספר הבלוקים שתופס המערך במטמון. לכן ה- miss rate עבור הקוד הזה יהיה 0.25 .

$$\frac{\text{array size}}{\text{block size}} = \frac{2^{12}}{2^{4}} = 2^{8}\,blocks \rightarrow 256\,misses$$

שאלה: how many misses

ג. כיצד הייתה משתנה תשובתך לסעיף ב' אם משתנים מסוג integer לא היו מיושרים(aligned) בזיכרון?

אם המערך לא היה מיושר, אז הוא היה מוסט קדימה או אחורה במספר בתים, ואז מספר הבלוקים שהמערך תופס וההחטאות הוא 256+1=257.

שאלה ב' (6)

ד. איזה עיקרון (בהקשר של זיכרון מטמון) בא לידי ביטוי בקטע הקוד?

1. עיקרון אי הוודאות.
2. עיקרון הלוקאליות במקום.
3. עיקרון הלוקאליות בזמן.
4. עיקרון הלוקאליות במקום ועיקרון הלוקאליות בזמן.
5. אף אחת מתשובות 1-4

עיקרון הלוקאליות במקום בא לידי ביטוי בקטע קוד זה, כיוון שהמערך יושב בזיכרון בצורה רציפה ולכן בכל פעם שניגשנו לבלוק מסוים חסכנו החטאה ל- int-ים באותו בלוק.

עיקרון הלוקאליות בזמן לא בא לידי ביטוי מכיוון שניגשנו לכל נתון פעם אחת בלבד.

What about *i* when it comes from the memory ?

- locality in time for *i*...

שאלה

ה.מהו מספר הבלוקים המקסימאלי שיועברו מהזיכרון הראשי למטמון הנתונים (עבור הרצה יחידה של קטע הקוד) אם נתון כעת, כי משתנה i יושב בכתובת פיזית 0x0001 0000 בזיכרון (ולא ברגיסטר) וכן מדיניות no write allocate ?

- Number of blocks copied from memory to cache?

בביצוע לולאת for אנו ניגשים לכל אחד מהמשתנים במערך לצורך כתיבה בלבד.
המשתנה היחידי שנקרא הוא i (בבדיקת תנאי היציאה).
המטמון עובד במדיניות NO-write allocate ולכן בכל בקשות הכתיבה נעדכן רק את הזיכרון הראשי, והבלוק היחיד שיטען למטמון יהיה של משתנה i.
הכתובת x000010000 יושבת מיושרת בזיכרון ולכן נטען בלוק אחד למטמון.

ו.איזה סוגי החטאות יש בדוגמת הקוד הנתונה?

מכיוון שאנו ניגשים לכל נתון פעם אחת בדיוק, אנו מחטיאים רק כאשר אנו ניגשים לבלוק בפעם הראשונה.
לכן יש בקוד רק שגיאות compulsory

שאלה

ו. עבור השינויים הבאים במטמון, כיצד הם ישפיעו על ה-HR בקוד הנתון ?

1. שינוי מדיניות הפינוי למדיניות מתקדמת יותר
2. הכפלת גודל המטמון כך שמספר הבלוקים יוכפל
3. הכפלת גודל המטמון כך שגודל הבלוק יוכפל

אנו ניגשים לכל נתון פעם אחת. לכן שינוים 1 ו-2 לא ישפיעו על ביצועי התוכנית . כמו שמנו לב סעיף קודם, יש בקוד רק החטאות מסוג compulsory . הכפלת גודל הבלוק תגרום לכך שתהיה החטאה רק פעם ב-8 גישות למערך ולכן תשפר את ה- HR .

שאלה ב' (7) : LRU

ח. תאר (בקצרה) שיטה למימוש מדיניות פינוי LRU במטמון הנתונים.
על המימוש להיות חסכוני בזיכרון ככל הניתן (גם על חשבון יעילותו).
כמה סיביות יש להוסיף למטמון עבור המימוש שהצעת.

- Implement an LRU – how many bits ?

נממש LRU ברמת ה-set ע"י שמירת רשימה ובה ארבעה צמתים (אחד לכל way), השומרת את סדר עדכון ה- ways. בכל גישה ל-way נעדכן את הסדר ברשימה כך שה-way בו השתמשנו לאחרונה יהיה האחרון ברשימה.

כדי לשמור את מספור ה-way נשתמש בשתי סיביות עבור כל way (כי יש ארבעה ways) ולכן בסה"כ נוסיף למטמון 2*4=8 סיביות לכל set.

שאלה ב' (7) : LRU

ניתן לשפר את השיטה הקודמת באופן הבא:
נשמור ברשימה רק שלושה צמתים: נשמיט את הצומת הממופה ל- way בו השתמשנו לאחרונה. בכל גישה ל- way הנמצא ברשימה, נוסיף לרשימה את ה way שהושמט ונוציא מהרשימה את ה-way אליו ניגשנו (הוא כרגע ה way בו השתמשנו לאחרונה).
בדרך זאת נוסיף למטמון 2*3=6 סיביות לכל set.

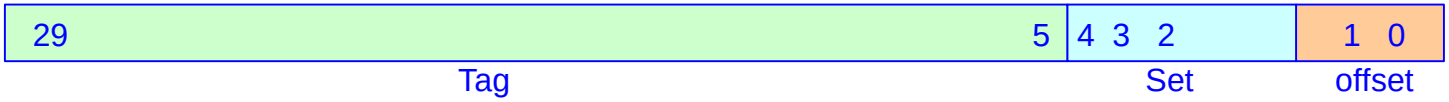
השיטה האופטימאלית (מבחינת חסכון במקום):
בכל רגע יש לנו תור ובו ארבעה ways.
מספר האפשרויות בו יכול התור להימצא הוא 4! = 24
מספיקות 5 סיביות (לכל set) כדי לקודד את כל המצבים.

שאלה II

המטמון ה- 2 way הבא שובץ במחשב בו מרחב הזיכרון הפיסי מחולק לארבעה סגמנטים זרים שווים בגודלם (256 MB כל אחד), A1 B0 A0 B1 לפי סדר זה.

למתכנן המטמון הוצבה דרישה לפיה לא ייוצר מצב שיותר מ- 50% מהמטמון יכיל כתובות של סגמנטי A או סגמנטי B ועם זאת תישמר שיטת 2way.

הצע דרך פשוטה ויעילה למימוש דרישה חדשה זו מה- cache והסבר את פעולתה.



שאלה 2

נשים לב כי ה- MSB של כל סגמנט מוגדר באופן הבא:

A0	00???..
B0	01???..
A1	10???..
B1	11???..

ספרת ה- 2nd MSB עבור סגמנטי A היא תמיד 0, ואילו עבור סגמנטי B היא תמיד 1.

נדאג למפות את המטמון כך שארבע השורות העליונות ימופו תמיד אך ורק לכתובות מסגמנטי A, ואילו ארבע בתחתונות לכתובות מסגמנטי B, ע"י שימוש בביט הנ"ל שנקרא לו ביט הסגמנט.
תוצאה זאת נשיג באמצעות שינוי פונקציית המיפוי (שדה ה- set) במקום שיכיל את 3 סיביות LSB של מספר הבלוק (mod8), נשתמש ב- שתי סיביות LSB בלבד ובסיסית הסגמנט.

מיפוי ישן: SSS DD דדדד דדדד דדדד דדדד דדדד דדדד

מיפוי חדש: TSS DD דדדד דדדד דדדד דדדד דדדד TSדד

שאלה II (5)

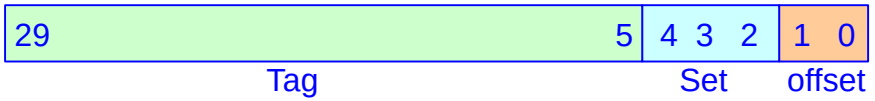
Way 1				LRU	Way 0				#set
				0	0	1	2	3	0
68	69	70	71	0	4	5	6	7	1
				0	8	9	10	11	2
				0	12	13	14	15	3
									4
									5
									6
									7

- Physical memory divided into 4 segments: A0 B0 A1 B1 (256MB each – 28 bit address) -
  - A0 00.... (30 bit address)
  - B0 01....
  - A1 10....
  - B1 11....

- 50% of the cache for segments A – 50% of the cache for segments B

- Solution:
- Change the mapping function so that the sets [0:3] will contains data for segments A, and [4:7], data for segment B

Original mapping:



New mapping: