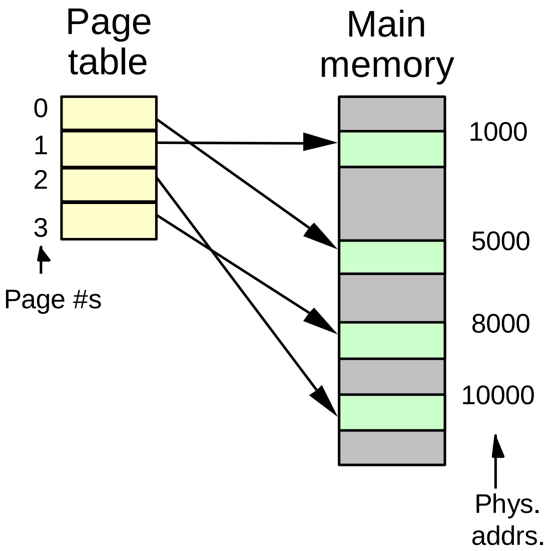
Translation Lookaside Buffers

The CPU generates *virtual* addresses, which correspond to locations in virtual memory.

In principle, the virtual addresses are translated to physical addresses using a page table.

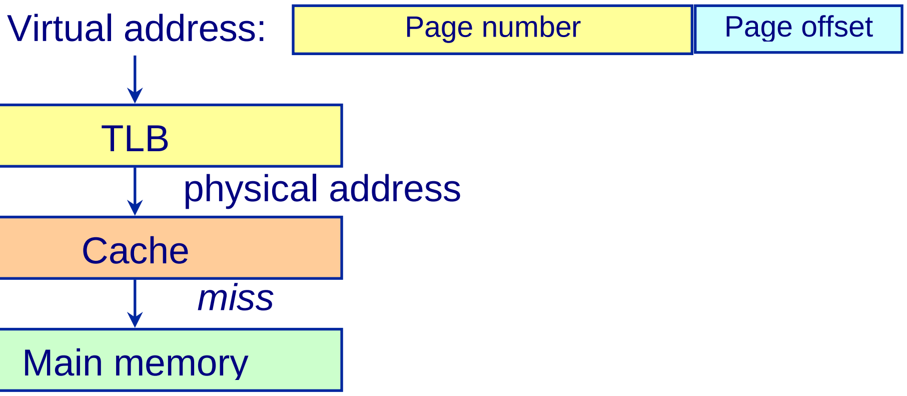


But this is too slow, so in practice, a *translation lookaside buffer* (TLB) is used.

It is like a special cache that is indexed by page number.

If there is a hit on a page number, then the address of the page in memory (called the *page-frame address*) is immediately obtained.

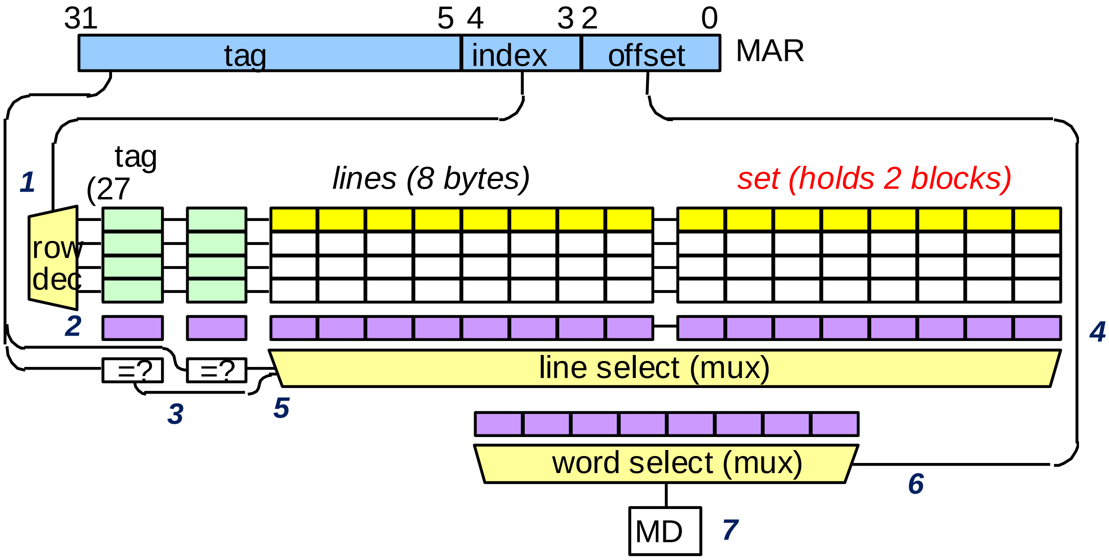
Therefore, the TLB and the cache must be accessed sequentially.



This adds an extra cycle in case of a hit.

How can we avoid wasting this time?

Let’s look at what happens when a memory address is accessed.



What are the steps in cache access?

1.
2.
3.
4.
5.
6.
7.

We always need to read lines into the sense amplifiers and then select the word (cf. the direct-mapped cache diagram in Lecture 10).

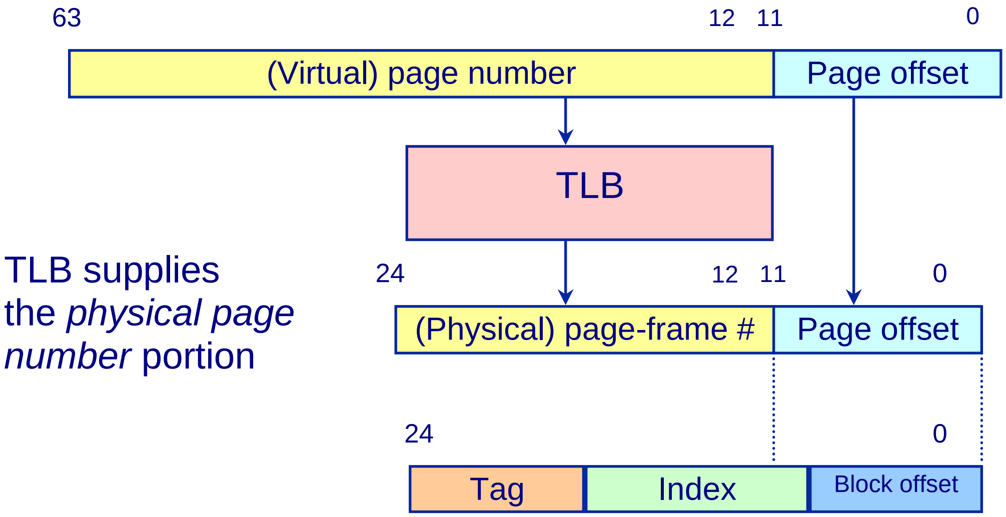
Now, if we know the index *before* address translation takes place, we can perform steps ________ while address translation is occurring.

There is a tradeoff between speed and power efficiency.

- For power efficiency, which order should should steps 1 through 4 be performed in?

- For maximum speed, which of steps 1 through 4 can be performed in parallel?

Let’s take a look at address translation.

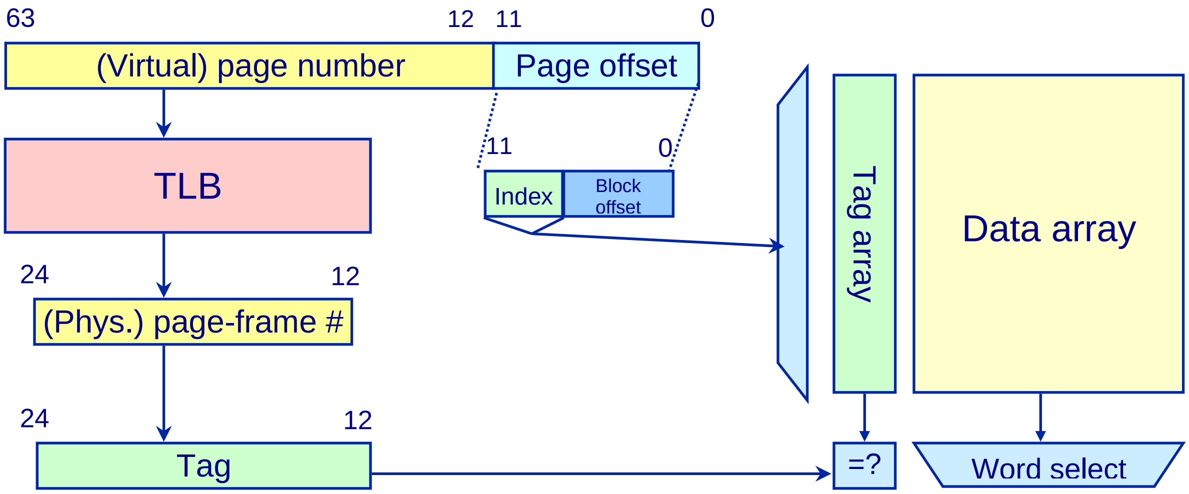


In this example, what is the page size?

How much physical memory is there?

Our goal is to allow the cache to be indexed before address translation completes.

In order to do that, we need to have the index field be *entirely contained* within the page offset.



Cache hit time reduces from two cycles to one!

… because the cache can now be *indexed* in parallel with TLB (although the tag match uses output from the TLB).

But there are some constraints...

- Suppose our cache is direct mapped. Then the index field just contains the line number. So, (line number || block offset) must fit inside the page offset.

  What is the largest the cache can be?

- If we want to increase the size of the cache, what can we do?

Options:

- For new machines, select page size such that—

$$\text{page size} \geq \frac{\text{cache size}}{\text{associativity}}$$

- If page size is fixed, select associativity so that—

$$\text{associativity} \geq \frac{\text{cache size}}{\text{page size}}$$

Example: MC88110

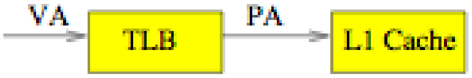
- Page size = 4KB
- I-cache, D-cache are both: 8KB, 2-way set-associative (4KB = 8KB / 2)

Example: VAX series

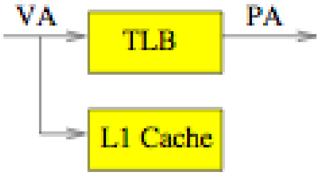
- Page size = 512B
- For a 16KB cache, need assoc. = (16KB / 512B) = 32-way set. assoc.!

The textbook gives these three alternatives for cache indexing and tagging. Answer some questions about them.

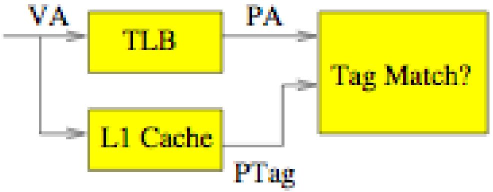
Physically Indexed and Tagged



Virtually Indexed and Tagged



Virtually Indexed but Physically Tagged



What’s the main disadantage of physically indexed and tagged?

What is the organization we have just been discussing (in the last diagram)?

What is the main disadvantage of virtually indexed and tagged?

Multilevel cache design

What are distinguishing features of the different cache levels of the four-level design (from 2013) illustrated on p. 135 of the textbook?

	Distinguish-ing feature	Size	Access time	Implement’n techology
L1 cache				
L2 cache				
L3 cache				
L4 cache				
Main mem.				

What are some advantages of a centralized cache?

What are some advantages of a banked structure?

Inclusion in multilevel caches

Answer these questions about inclusion policies.

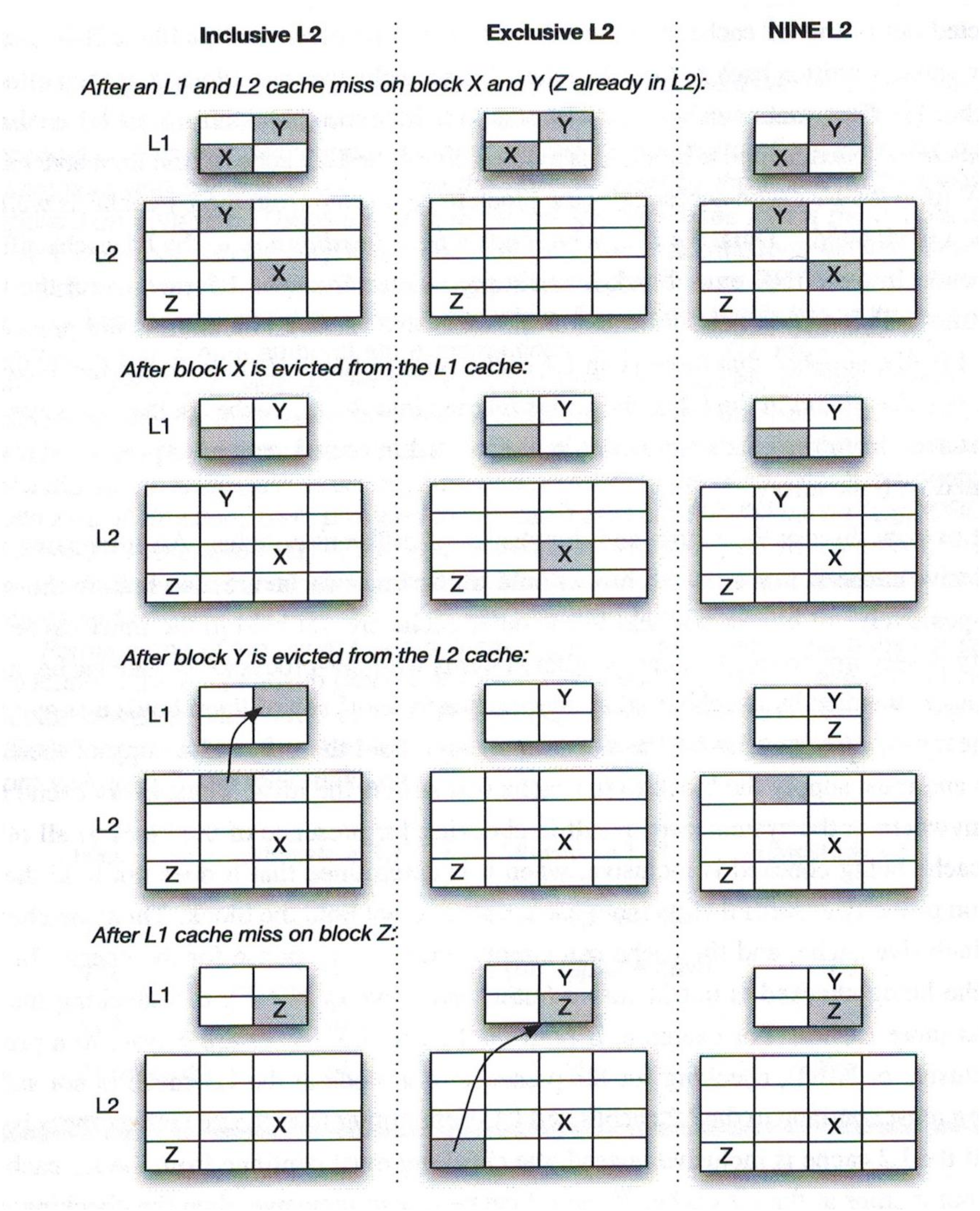
Which kind(s) of caches move a block from one level to the other?

Which kind(s) of caches propagate up an eviction from the L2 to the L1?

Which kind(s) of caches have to inform the L2 about a write to the L1?

In an inclusive cache, can L2 associativity be greater than L1 associativity?

Find and describe the typo in this diagram.


Inclusive L2
Exclusive L2
NINE L2
After an L1 and L2 cache miss on block X and Y (Z already in L2):
L1
L2
X
Y
Z
After block X is evicted from the L1 cache:
After block Y is evicted from the L2 cache:
After L1 cache miss on block Z:

Replacement policies

LRU is a good strategy for cache replacement.

In a set-associative cache, LRU is reasonably cheap to implement. Why?

With the LRU algorithm, the lines can be arranged in an *LRU stack*, in order of recency of reference. Suppose a string of references is—

a b c d a b e a b c d e

and there are 4 lines. Then the LRU stacks after each reference are—

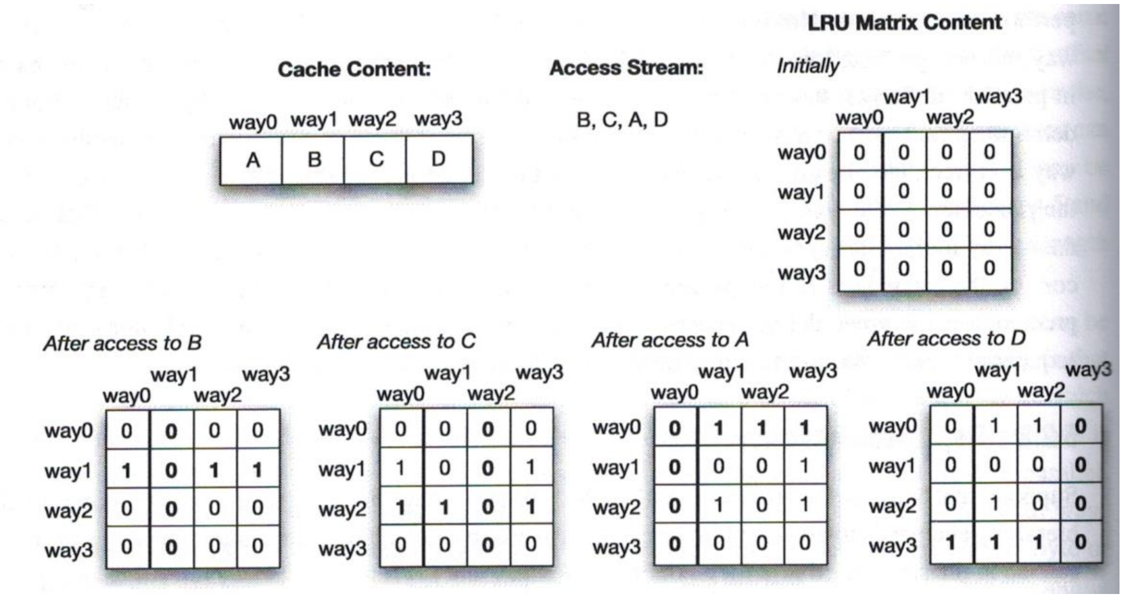
a	*b*	*c*	*d*	*a*	*b*	*e*	*a*	*b*	*c*	*d*	*e*
	a	*b*	*c*	*d*	*a*	*b*	*e*	*a*	*b*	*c*	*d*
		a	*b*	*c*	*d*	*a*	*b*	*e*	*a*	*b*	*c*
			a	*b*	*c*	*d*	*d*	*d*	*e*	*a*	*b*
*	*	*	*			*			*	*	*

Notice that at each step:

- The line that is referenced moves to the top of the LRU stack.
- All lines below that line keep their same position.
- All lines above that line move down by one position.

How many bits per set are required to keep track of LRU status in both of the implementations described in the text?

- Matrix
- Pseudo-LRU



Initially

	way0	way1	way2	way3
way0	0	0	0	0
way1	0	0	0	0
way2	0	0	0	0
way3	0	0	0	0

After access to B

	way0	way1	way2	way3
way0	0	0	0	0
way1	1	0	1	1
way2	0	0	0	0
way3	0	0	0	0

After access to C

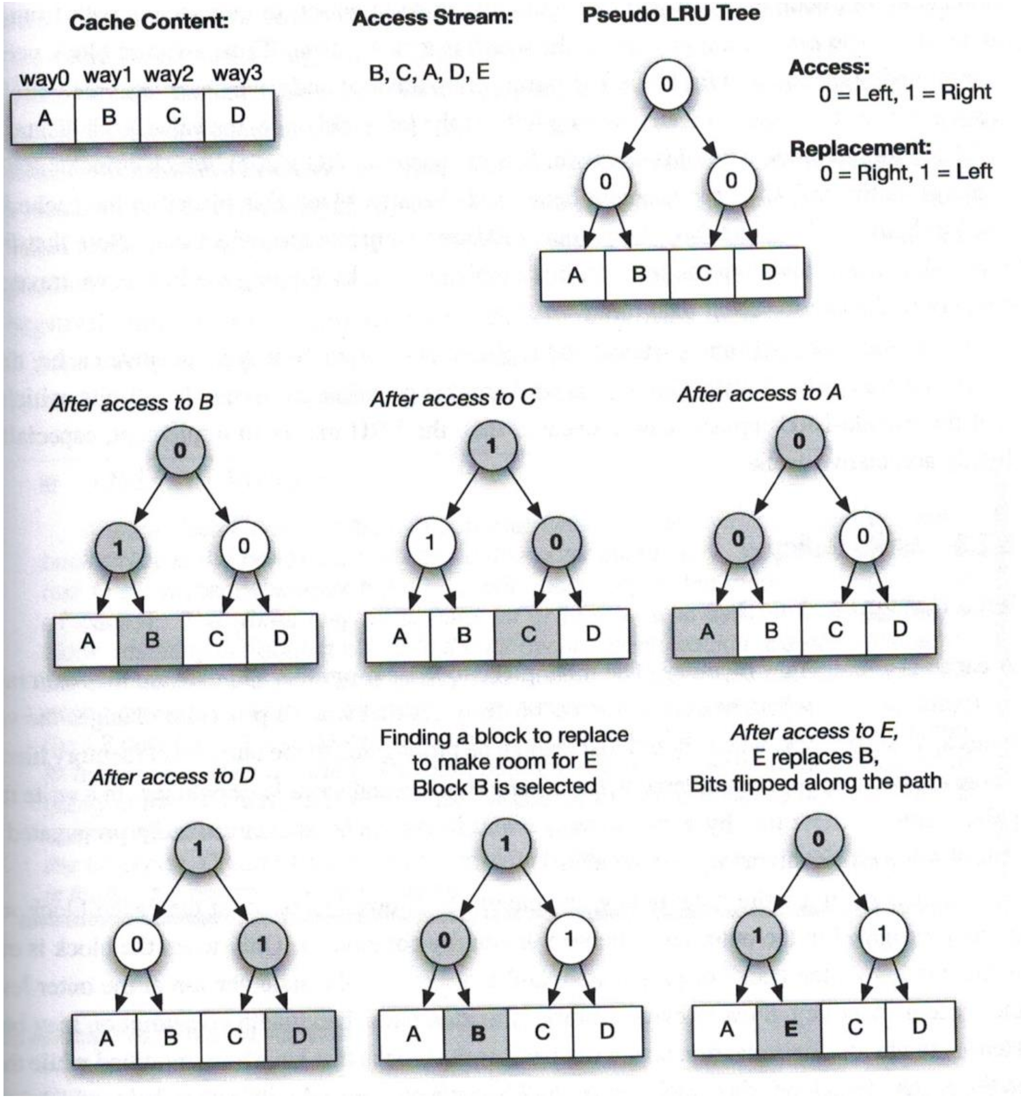
	way0	way1	way2	way3
way0	0	0	0	0
way1	1	0	0	1
way2	1	1	0	1
way3	0	0	0	0

After access to A

	way0	way1	way2	way3
way0	0	1	1	1
way1	0	0	0	1
way2	0	1	0	1
way3	0	0	0	0

After access to D

	way0	way1	way2	way3
way0	0	1	1	0
way1	0	0	0	0
way2	0	1	0	0
way3	1	1	1	0


Cache Content:
way0 way1 way2 way3
A B C D
Access Stream:
B, C, A, D, E
Pseudo LRU Tree
0
0 0
A B C D
Access:
0 = Left, 1 = Right
Replacement:
0 = Right, 1 = Left
After access to B
0
1 0
A B C D
After access to C
1
1 0
A B C D
After access to A
0
0 0
A B C D
After access to D
1
0 1
A B C D
Finding a block to replace
to make room for E
Block B is selected
1
0 1
A B C D
After access to E,
E replaces B,
Bits flipped along the path
0
1 1
A E C D