How Pseudo LRU Cache Replacement works

When all the *ways* in a *congruence class* (aka "*set*") in a cache are in use, a very popular method of choosing which *way* to reclaim/replace is pseudoLRU.

In layman-terms:

Whenever there is a need to pick one of many possible choices usually the choice is guided by a certain reasoning.

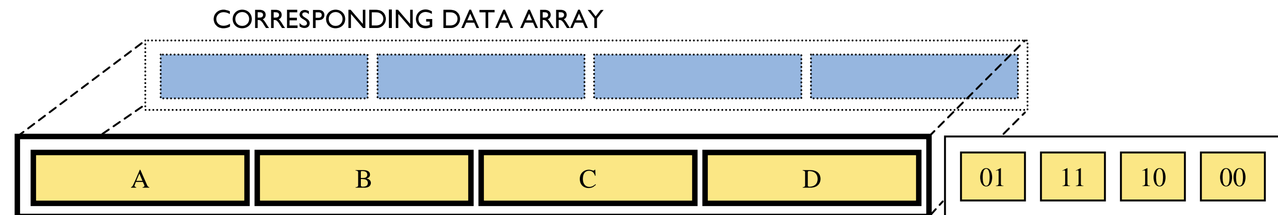
Here is a situation from real life. Say you are donating clothes to charity. You might ask yourself, "Which clothes to donate?". One way to choose might be random (and it may be argued whether there is any reasoning involved here). Another way might be "Let me choose the clothes that I have not worn in the longest time". Another might be, "Let me give away the oldest clothes I have." The second scheme in the example above is like the LRU scheme, which stands for least recently used. The third is like FIFO (First In First Out), and the first is, unsurprisingly, like the random replacement policy. The funny thing is, sometimes instead of trying to be smart/logical about it, if we just used random policy, we end up doing almost as well as the other policies (at least with caches), with very little overhead (as in the overhead to keep track of which is the oldest or least worn cloth you own).

First lets look at two forms of implementing true LRU in a cache, the latter using fewer bits than the former. Then we will look at pseudoLRU which uses even fewer bits to maintain the ordering information between the various "ways" of a set-associative cache. It has been observed that in spite of keeping limited ordering information, pseudoLRU does almost as well as true LRU in terms of overall microprocessor performance.

TRUE LRU

1st method

- $N*log_2(N)$ LRU bits per set, where N is number of ways
- The recency order between the various ways is maintained



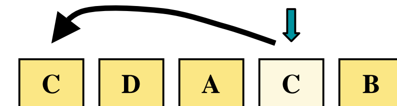
TAG ARRAY: n tags for an n-way set associative cache. Usually n is a power of 2 (though not necessary)

LRU BITS: $\log_2(n)$ bits **per** tag, to identify its recency of use

Current LRU bits imply the current order is

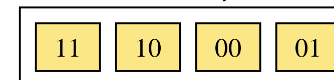
MRU LRU

Say there is a hit to C. C becomes the MRU. D, A, B maintain their relative order



MRU LRU

Final LRU bits: very different



- **WHAT THE LRU BITS MEAN**

The LRU bits show that the order of access in the past with the most recently accessed element (LRU bits 00) first is

D, A, C, B

corresponding to LRU bits indicating 0,1,2 and 3. So at this time if a replacement candidate is to be selected B would be chosen, since is was the least recently used (LRU).

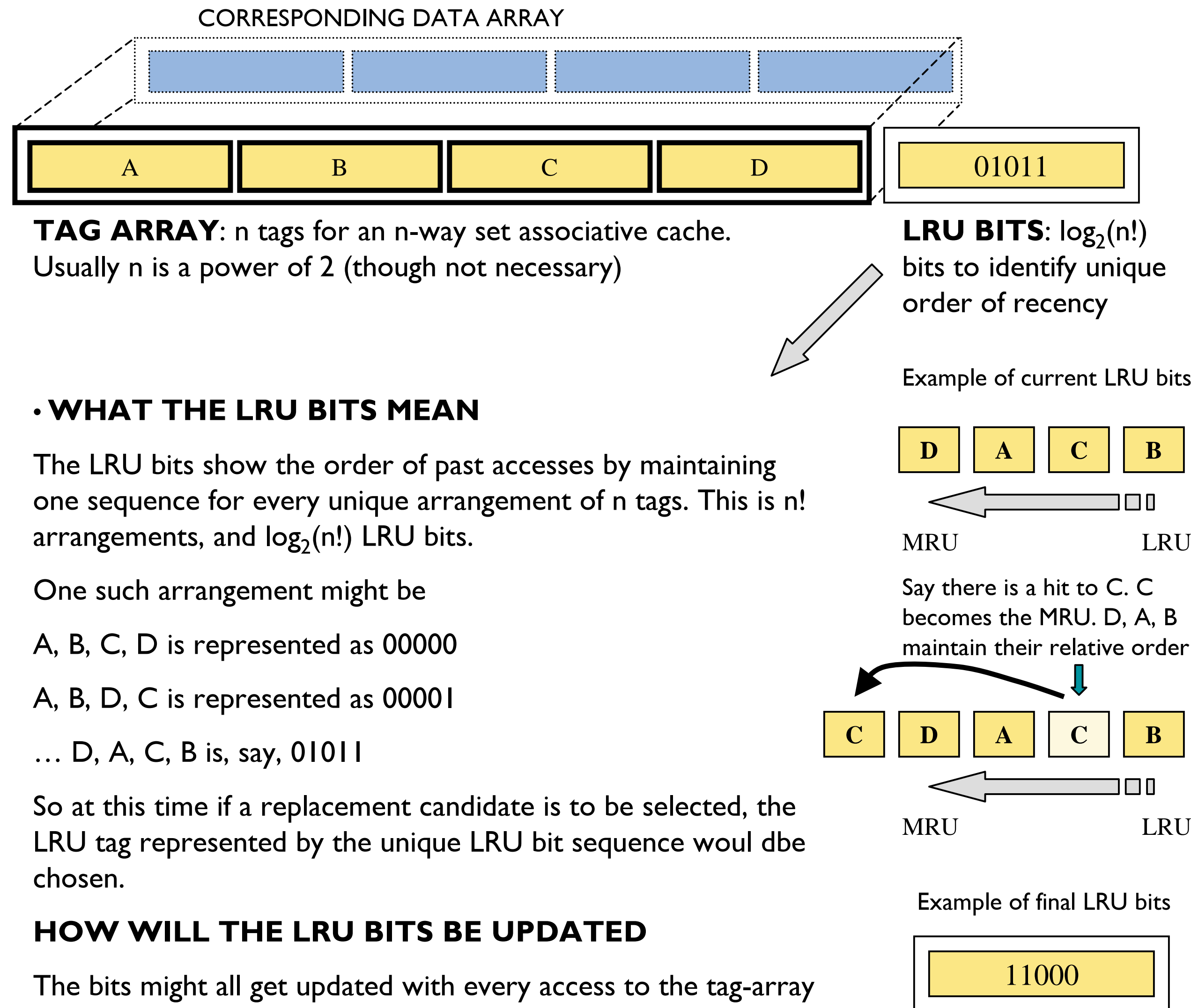
- **HOW WILL THE LRU BITS BE UPDATED**

The bits might all get updated with every access to the tag-array to indicate the new order of recency. The only time bits will not get updated is if there is a hit to the most recently used tag, in which case the new recency order is same as it was. For example if there is a hit to tag C, then the order changes as shown to the right and the LRU bits change to show that new order.

TRUE LRU

2nd method

- **$\log_2(N!)$** LRU bits per set, where N is number of ways. Therefore this uses fewer bits to maintain ordering information. As an example, for an 8-way set associative cache, we need 16 bits compared to 24 required by the 1st method.
- The recency order between the various ways is maintained



TAG ARRAY: n tags for an n-way set associative cache. Usually n is a power of 2 (though not necessary)

LRU BITS: $\log_2(n!)$ bits to identify unique order of recency

- **WHAT THE LRU BITS MEAN**

The LRU bits show the order of past accesses by maintaining one sequence for every unique arrangement of n tags. This is n! arrangements, and $\log_2(n!)$ LRU bits.

One such arrangement might be

A, B, C, D is represented as 00000

A, B, D, C is represented as 00001

… D, A, C, B is, say, 01011

So at this time if a replacement candidate is to be selected, the LRU tag represented by the unique LRU bit sequence woul dbe chosen.

HOW WILL THE LRU BITS BE UPDATED

The bits might all get updated with every access to the tag-array to indicate the new order of recency. The only time bits will not get updated is if there is a hit to the most recently used tag, in which case the new recency order is same as it was.

PSEUDO LRU

N-1 LRU bits per set, where N is number of ways. 7 bits for an 8 way set-associative cache compared to 24 or 16 by the two True LRU methods discussed.

• The recency order between the various ways is **not** fully maintained. It is partially maintained…and that seems good enough for most workloads.

Remembers which *way between a pair of consecutive ways* is LRU.

Example: For an 8-way there are 4 such pairs:

Way0-Way1, Way2-Way3, Way4-Way5 and Way6-Way7 (need 4 bits)

Remembers which *pair between consecutive pairs of the above pairs* is LRU.

Example: For an 8-way there are 2 such pairs:

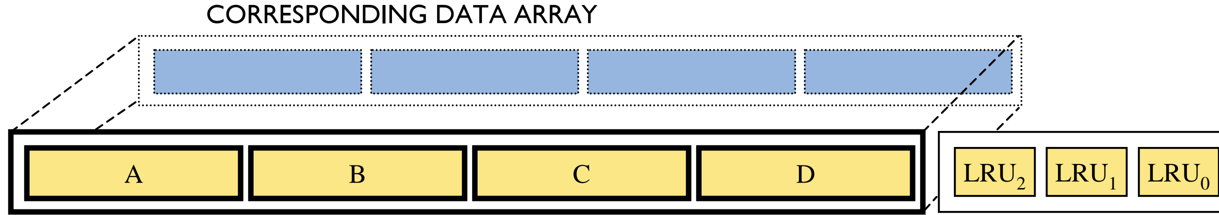
Ways01-Ways23, Ways45-Ways67 (need 2 bits)

Keeps doing that for *larger and larger consecutive groups*.

Example: For an 8-way there is 1 such pair:

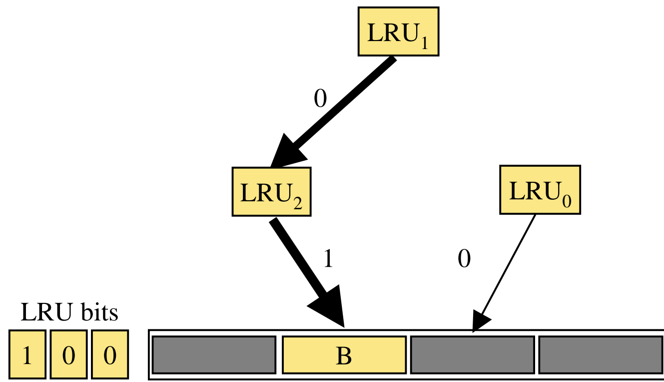
Ways0123-Ways4567 (need 2 bits)

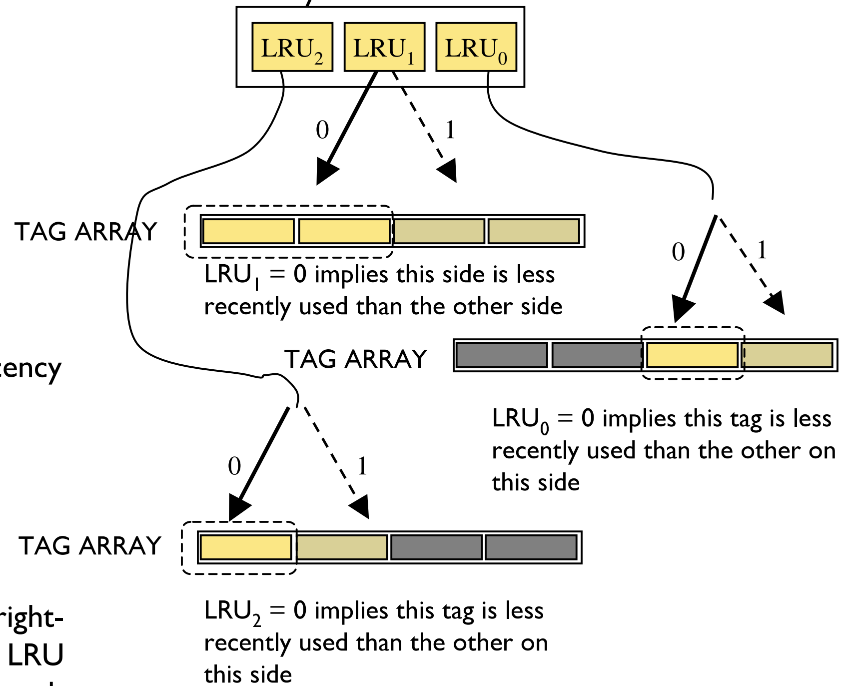
So for an 8-way we need 7 bits in all.



TAG ARRAY: n tags for an n-way set associative cache. Usually n is a power of 2 (though not necessary)

LRU BITS: n-1 bits, each bit indicates recency between a pair of consecutive regions of the tag array





LRU bits are 100 i.e. B is the least recently used tag

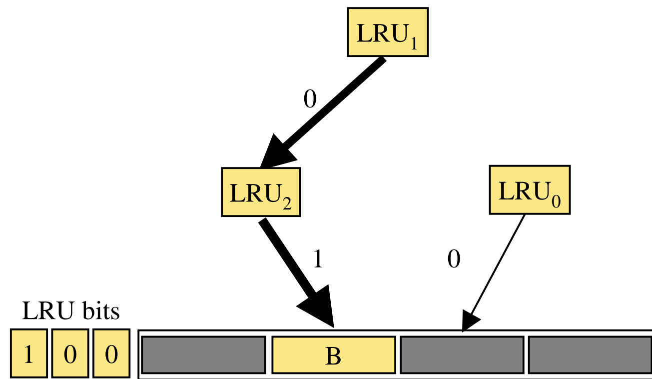
Pseudo LRU only keeps partial information about the recency order between tags.

In this example, bit LRU1 says the left two tags are less recently used than the right two

bit LRU2 says between A and B, B is less recently used.

bit LRU0 keeps some record of what is going on on the right-side, even though it is not required in finding the current LRU tag, by remembering on the right side, C is less recently used than D

EXAMPLE 1 – HIT: more recently used tag touched



Tag A hits, making it the most recently used

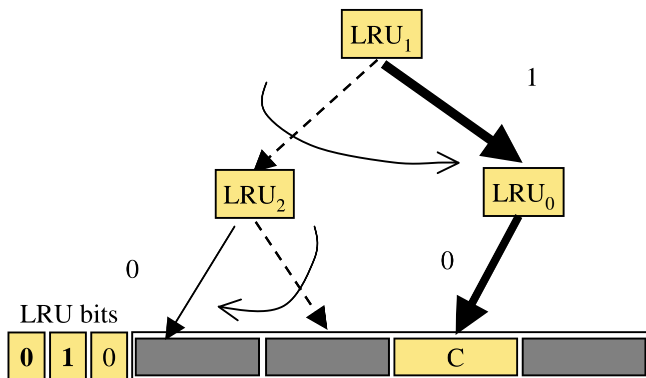
No change to the LRU bits!

LRU bits do not care about more recently used tags, they only care about least/less recently used tags.

In this case B is least recently used and C is less recently used (compared to D)

Since those are not touched, the LRU bits do not change.

EXAMPLE 2 – MISS: LRU bits have to be updated



Miss! Requires a replacement

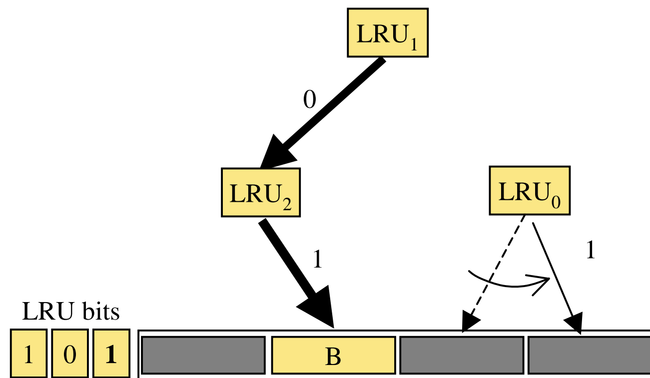
Easy. Choose B, since that is where the arrows lead you to.

But now position B has become most recently used. The LRU bits need to point to the LRU. So they will change.

Action: Flip all bits (swing all arrows) that lead to B.

Logic behind this: At the level that chooses between A and B, B is more recent, so point to A; i.e. in this example toggle bit LRU_2. At the next higher level, since B is on the left side, the "left side" has become more recently used as a group, so switch to the right side; i.e. in this example toggle bit LRU_1.

EXAMPLE 3 – HIT: less recently used tag touched



Tag C hits. This is not *the* least recently used but it is a less recently used tag, which needs to change since it became the most recently used just now.

The LRU bits will change.

Action: Flip all bits (swing all arrows) that lead to C.

Logic behind this: At the level that chooses between C and D, C became more recent and since it was being pointed to by LRU_0, that pointer must swing to point to D now. At the next higher level, since LRU_1 is pointing to the left side as being less recently used and since that remains true even now, nothing needs to be done to LRU_1.

Therefore a simple way to look at pseudoLRU is -

For tag hits flip/*swing* all the bits/*arrows* that lead to it. If the bits do not lead to the tag then the pseudoLRU bits do not change upon that hit.

For tag misses follow the bits/arrows to figure out the least recently used tag. This is the replacement candidate. Then, flip/*swing* all the bits/*arrows* that lead to it.

And oh by the way, the reason pseudoLRU is not as rigorous as true LRU is that when a tag position is used (thus making it most recent), pseudo LRU policy refuses to even look at that half of the set for the next LRU tag to point to. Instead it immediately swings over to the other half. In other words, a tag just touched and the resultant pseudoLRU bits can never be in the same half of the set!