The Camera Simulator
User Guide

ANCA Apatean
- UTCN -

CAMERA – CACHE Memory

Illustrate the concepts of CACHE MEMORY with :

(3 mapped schemes)

1. **Direct mapped** cache
2. **Fully Associative** cache
3. **Set Associative (with m-sets) or n-ways** cache

When the user launches one of the 3 applications for cache memory, the ***Update Progress*** field displays the specifications according to the system which was modeled in the application

- the size of the cache memory and
  the size of the main memory
- the size of pages / slots and of proper blocks
- how the address of the main memory is partitioned in binary

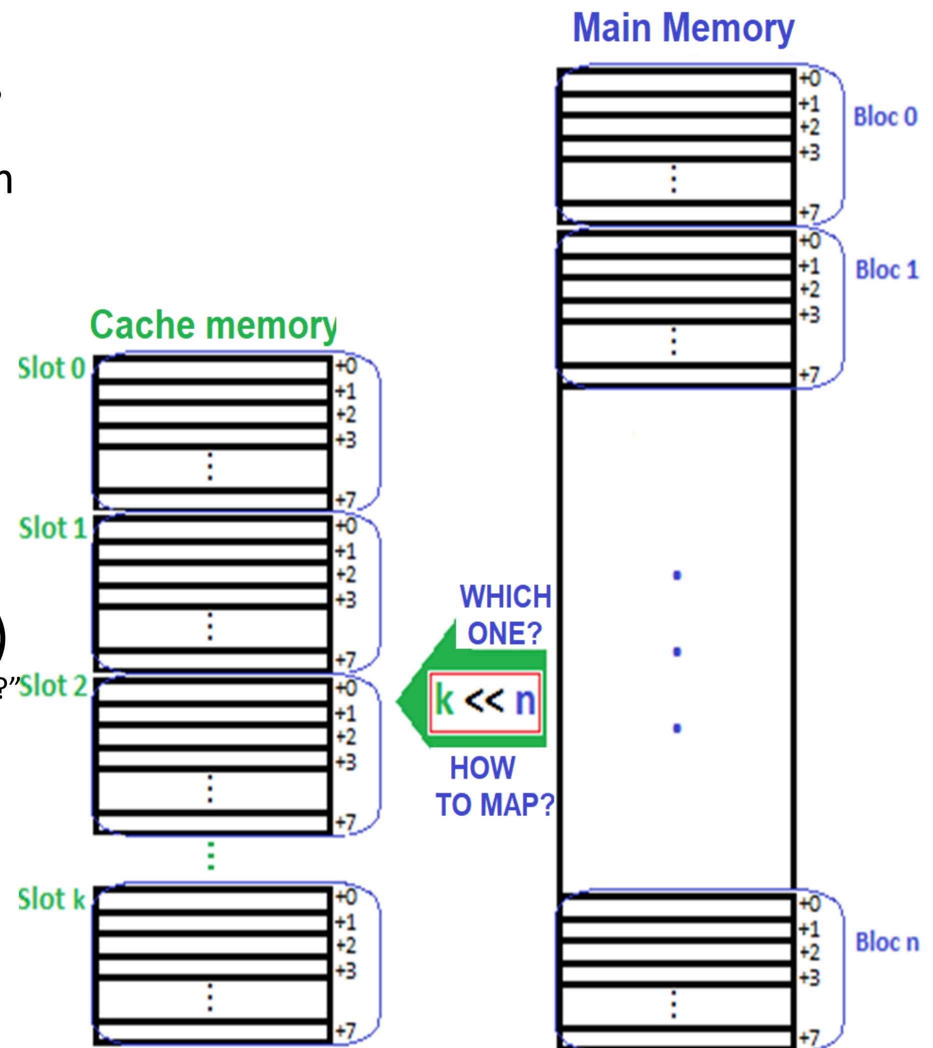
(answer the questions: **WHICH**? **HOW**?)

WHICH? :"**Which** blocks from the main memory in cache will be mapped?"

HOW? "**How** the blocks will be arranged in the cache memory?

Cache memory has smaller size(<<) than the main memory

MAPPING = "the way in which some blocks from the main memory are chosen to be copied in cache memory"



Introduction

The purpose of using cache memory: to increase the speed with which the CPU access the information inside the system

=> the cache memory will store ***the most recent*** or ***the most frequent*** used information

- It is placed closer to CPU (comparing to RAM) and it has higher speed (proved by the memory hierarchy concept)

=> The CPU will receive the requested information more rapidly (being in cache and not in RAM)

One possible problem: **the cache memory capacity is much smaller than that of RAM**

Example: 64k B cache L1 vs 2GB RAM (main memory)

2^{16} B vs 2^{21} B => $2^{21} / 2^{16} = 2^5 = 32$

=> 32 times the cache L1 memory is smaller than main memory !!!

- generally, the cache memory is ***content-addressable***, not by its address like main memory

=> the cache memory is also called CAM-type (content-addressable memory)

- the "content" being verified is in fact a subset of bits (a field) from the data location from the main memory

- More blocks from the main memory will attempt to the same slot (many-to-one mapping)
- The tag field will discriminate between them (within a set or within all of them)

The CAMERA simulator

Install the CAMERA – skip this step in case you already installed it

I. CAMERA is a software packet for the **cache memory** and **virtual memory**

- JAVA coded, belongs to [*ref1*]

II. **Install CAMERA**

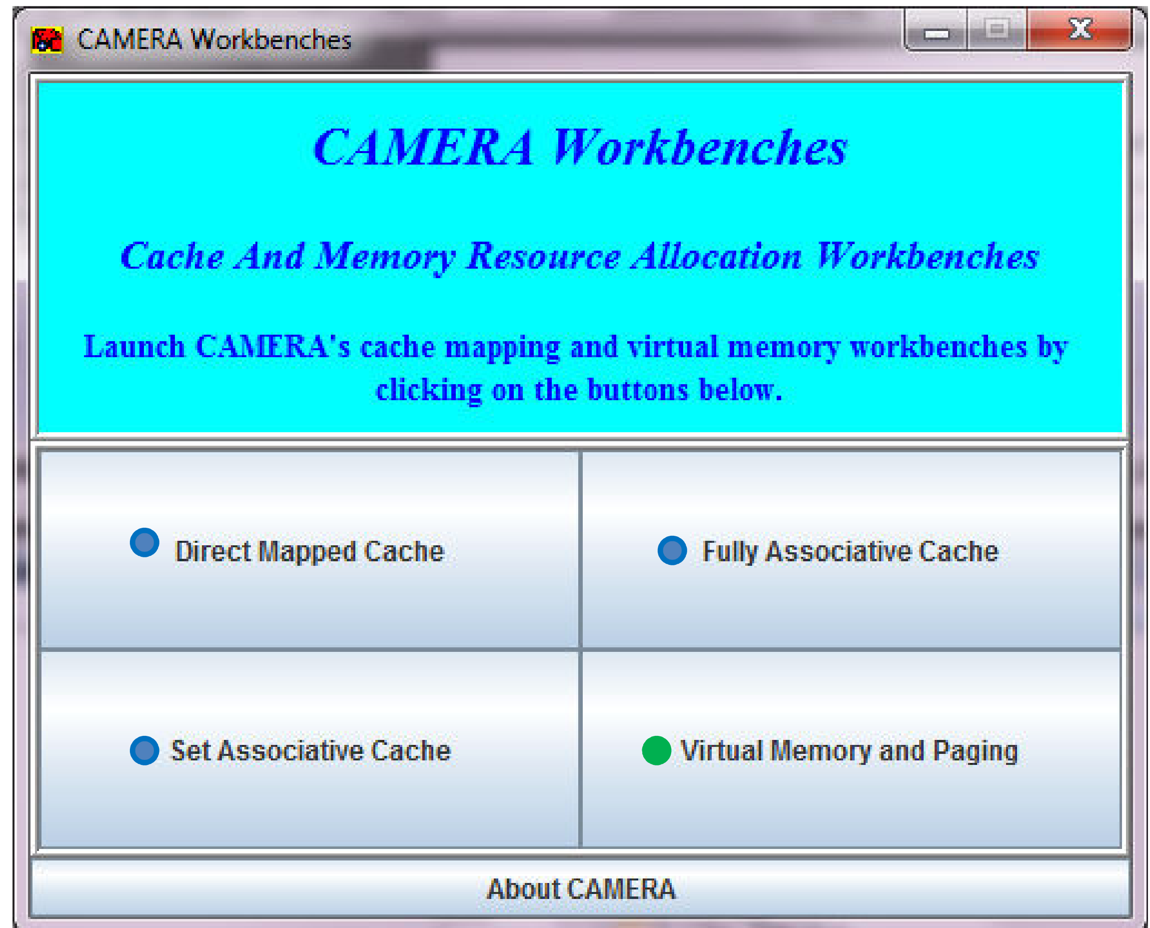
Using the archive ***Camera.zip***

IIa. Un-archive and double-click on ***Camera.jar*** in order to install

IIa'. or you can follow the installing instructions from the ***user guide*** from [*ref1*]

III. **Open/start CAMERA**

- Any of the 4 applications are Standalone -> they can be run independently;



- and they either depend on an external application

Only to CAMERA …

- The following specific attributes are only **for the Camera simulator**:
  - size of the cache: 16 slots* and each slot has 8 “words”**
  - size of the main memory : 32 blocks and each block has 8 words

=> in CAMERA the cache is only 2 times smaller: 32*8B/16*8B= 2 times only !!!

- But in a real system they may be different
  - Examples :
    - in a PC or PC-XT system: there is a **1MB of main memory** due to the 20 address lines
    - in a PC-AT system: there is **16MB of main memory** due to the 24 address lines
- Check with your system by running the **cpu-z** application how much main memory and cache L1 is in your system

*We will refer to cache structures as SLOTS and main memory structures as BLOCKS

** word will be the generic term for the addressable content from the memory (which in byte in our case)

The cache memory has **128 locations**

8 locations per slot

Main memory has 100h locations numbered from 00h to FFh => **256 locations**

each BLOCK contains **8 locations**

Direct Mapped Cache

Cache

Blk0 (Slot 0), Blk1 (Slot 1), Blk2, Blk3, Blk4, Blk5, Blk6, Blk7, Blk8, Blk9, Blk10, Blk11, Blk12, Blk13, Blk14, Blk15 (Slot 15)

Cache Hits 0 Cache Misses 0

Memory

	+0	+1	+2	+3	+4	+5	+6	+7	
00	B0 W0	B0 W1	B0 W2	B0 W3	B0 W4	B0 W5	B0 W6	B0 W7	Block 0
08	B1 W0	B1 W1	B1 W2	B1 W3	B1 W4	B1 W5	B1 W6	B1 W7	Block 1
10	B2 W0	B2 W1	B2 W2	B2 W3	B2 W4	B2 W5	B2 W6	B2 W7	
18	B3 W0	B3 W1	B3 W2	B3 W3	B3 W4	B3 W5	B3 W6	B3 W7	
20	B4 W0	B4 W1	B4 W2	B4 W3	B4 W4	B4 W5	B4 W6	B4 W7	
28	B5 W0	B5 W1	B5 W2	B5 W3	B5 W4	B5 W5	B5 W6	B5 W7	
30	B6 W0	B6 W1	B6 W2	B6 W3	B6 W4	B6 W5	B6 W6	B6 W7	
38	B7 W0	B7 W1	B7 W2	B7 W3	B7 W4	B7 W5	B7 W6	B7 W7	
40	B8 W0	B8 W1	B8 W2	B8 W3	B8 W4	B8 W5	B8 W6	B8 W7	
48	B9 W0	B9 W1	B9 W2	B9 W3	B9 W4	B9 W5	B9 W6	B9 W7	
50	B10 W0	B10 W1	B10 W2	B10 W3	B10 W4	B10 W5	B10 W6	B10 W7	
58	B11 W0	B11 W1	B11 W2	B11 W3	B11 W4	B11 W5	B11 W6	B11 W7	
60	B12 W0	B12 W1	B12 W2	B12 W3	B12 W4	B12 W5	B12 W6	B12 W7	
68	B13 W0	B13 W1	B13 W2	B13 W3	B13 W4	B13 W5	B13 W6	B13 W7	
70	B14 W0	B14 W1	B14 W2	B14 W3	B14 W4	B14 W5	B14 W6	B14 W7	
78	B15 W0	B15 W1	B15 W2	B15 W3	B15 W4	B15 W5	B15 W6	B15 W7	
80	B16 W0	B16 W1	B16 W2	B16 W3	B16 W4	B16 W5	B16 W6	B16 W7	
88	B17 W0	B17 W1	B17 W2	B17 W3	B17 W4	B17 W5	B17 W6	B17 W7	
90	B18 W0	B18 W1	B18 W2	B18 W3	B18 W4	B18 W5	B18 W6	B18 W7	
98	B19 W0	B19 W1	B19 W2	B19 W3	B19 W4	B19 W5	B19 W6	B19 W7	
A0	B20 W0	B20 W1	B20 W2	B20 W3	B20 W4	B20 W5	B20 W6	B20 W7	
A8	B21 W0	B21 W1	B21 W2	B21 W3	B21 W4	B21 W5	B21 W6	B21 W7	
B0	B22 W0	B22 W1	B22 W2	B22 W3	B22 W4	B22 W5	B22 W6	B22 W7	
B8	B23 W0	B23 W1	B23 W2	B23 W3	B23 W4	B23 W5	B23 W6	B23 W7	
C0	B24 W0	B24 W1	B24 W2	B24 W3	B24 W4	B24 W5	B24 W6	B24 W7	
C8	B25 W0	B25 W1	B25 W2	B25 W3	B25 W4	B25 W5	B25 W6	B25 W7	
D0	B26 W0	B26 W1	B26 W2	B26 W3	B26 W4	B26 W5	B26 W6	B26 W7	
D8	B27 W0	B27 W1	B27 W2	B27 W3	B27 W4	B27 W5	B27 W6	B27 W7	
E0	B28 W0	B28 W1	B28 W2	B28 W3	B28 W4	B28 W5	B28 W6	B28 W7	
E8	B29 W0	B29 W1	B29 W2	B29 W3	B29 W4	B29 W5	B29 W6	B29 W7	
F0	B30 W0	B30 W1	B30 W2	B30 W3	B30 W4	B30 W5	B30 W6	B30 W7	
F8	B31 W0	B31 W1	B31 W2	B31 W3	B31 W4	B31 W5	B31 W6	B31 W7	Bloc 31

Address Reference String

the hex "values" appearing here are actually the addresses of the data requested by the CPU

Auto Generate Add. Ref. Str.

Self Generate Add. Ref. Str.

Main Memory Address

TAG BLOCK WORD

Memory Block and Word Bits

BLOCK WORD

how the address is partitioned in binary

PROGRESS UPDATE

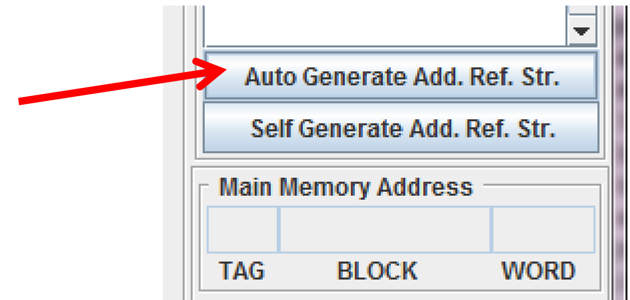
8 Words per Block
Please generate the Address Reference String.
Then click on "Next" to continue.

Restart Next Back Quit

Exercise 1: understand how the Camera simulator is working

1. Choose a "**Direct Mapped Cache**" by pressing the corresponding button.

Select the option "***AutoGenerate Add.Ref.Str.*** "



a) Analyse the connection between the **main memory ("Memory")** and the **cache memory ("Cache")**

(0.1p) a1) Specify what size has the main memory and how it is organised :
How many locations (called generically "words") can be inside a "Block" ? How are the blocks numbered ? Follow the contents of the locations . What did you noticed?

(0.1p) a2) Specify the size of the cache memory and how is this organised:
How many words can be inside a "Slot"(block) ? Would all the words from the main memory fit in the cache memory? How are the slots numbered?

b) Analyse the auto generated sequence, step by step:

b1) Notice how each location (1C on the next figure) is partitioned in fields ("***Main Memory Address***" 1Ch=0001 1100b, i.e. tag=0, block=0011, word=100)

(0.1p) b2) Press ***Next*** and analyse the message appearing at the "**progress update**";
Repeat all the points from b) and explain the ***cache hits*** and ***cache misses*** values

Direct Mapped Cache

Cache

Block	Tag								
Blk0									
Blk1									
Blk2									
Blk3	0	B3 W0	B3 W1	B3 W2	B3 W3	B3 W4	B3 W5	B3 W6	B3 W7
Blk4	1	B20 W0	B20 W1	B20 W2	B20 W3	B20 W4	B20 W5	B20 W6	B20 W7
Blk5									
Blk6									
Blk7									
Blk8									
Blk9									
Blk10									
Blk11									
Blk12	0	B12 W0	B12 W1	B12 W2	B12 W3	B12 W4	B12 W5	B12 W6	B12 W7
Blk13									
Blk14									
Blk15									

Cache Hits 0 Cache Misses 4

Memory

	+0	+1	+2	+3	+4	+5	+6	+7
00	B0 W0	B0 W1	B0 W2	B0 W3	B0 W4	B0 W5	B0 W6	B0 W7
08	B1 W0	B1 W1	B1 W2	B1 W3	B1 W4	B1 W5	B1 W6	B1 W7
10	B2 W0	B2 W1	B2 W2	B2 W3	B2 W4	B2 W5	B2 W6	B2 W7
18	B3 W0	B3 W1	B3 W2	B3 W3	B3 W4	B3 W5	B3 W6	B3 W7
20	B4 W0	B4 W1	B4 W2	B4 W3	B4 W4	B4 W5	B4 W6	B4 W7
28	B5 W0	B5 W1	B5 W2	B5 W3	B5 W4	B5 W5	B5 W6	B5 W7
30	B6 W0	B6 W1	B6 W2	B6 W3	B6 W4	B6 W5	B6 W6	B6 W7
38	B7 W0	B7 W1	B7 W2	B7 W3	B7 W4	B7 W5	B7 W6	B7 W7
40	B8 W0	B8 W1	B8 W2	B8 W3	B8 W4	B8 W5	B8 W6	B8 W7
48	B9 W0	B9 W1	B9 W2	B9 W3	B9 W4	B9 W5	B9 W6	B9 W7
50	B10 W0	B10 W1	B10 W2	B10 W3	B10 W4	B10 W5	B10 W6	B10 W7
58	B11 W0	B11 W1	B11 W2	B11 W3	B11 W4	B11 W5	B11 W6	B11 W7
60	B12 W0	B12 W1	B12 W2	B12 W3	B12 W4	B12 W5	B12 W6	B12 W7
68	B13 W0	B13 W1	B13 W2	B13 W3	B13 W4	B13 W5	B13 W6	B13 W7
70	B14 W0	B14 W1	B14 W2	B14 W3	B14 W4	B14 W5	B14 W6	B14 W7
78	B15 W0	B15 W1	B15 W2	B15 W3	B15 W4	B15 W5	B15 W6	B15 W7
80	B16 W0	B16 W1	B16 W2	B16 W3	B16 W4	B16 W5	B16 W6	B16 W7
88	B17 W0	B17 W1	B17 W2	B17 W3	B17 W4	B17 W5	B17 W6	B17 W7
90	B18 W0	B18 W1	B18 W2	B18 W3	B18 W4	B18 W5	B18 W6	B18 W7
98	B19 W0	B19 W1	B19 W2	B19 W3	B19 W4	B19 W5	B19 W6	B19 W7
A0	B20 W0	B20 W1	B20 W2	B20 W3	B20 W4	B20 W5	B20 W6	B20 W7
A8	B21 W0	B21 W1	B21 W2	B21 W3	B21 W4	B21 W5	B21 W6	B21 W7
B0	B22 W0	B22 W1	B22 W2	B22 W3	B22 W4	B22 W5	B22 W6	B22 W7
B8	B23 W0	B23 W1	B23 W2	B23 W3	B23 W4	B23 W5	B23 W6	B23 W7
C0	B24 W0	B24 W1	B24 W2	B24 W3	B24 W4	B24 W5	B24 W6	B24 W7
C8	B25 W0	B25 W1	B25 W2	B25 W3	B25 W4	B25 W5	B25 W6	B25 W7
D0	B26 W0	B26 W1	B26 W2	B26 W3	B26 W4	B26 W5	B26 W6	B26 W7
D8	B27 W0	B27 W1	B27 W2	B27 W3	B27 W4	B27 W5	B27 W6	B27 W7
E0	B28 W0	B28 W1	B28 W2	B28 W3	B28 W4	B28 W5	B28 W6	B28 W7
E8	B29 W0	B29 W1	B29 W2	B29 W3	B29 W4	B29 W5	B29 W6	B29 W7
F0	B30 W0	B30 W1	B30 W2	B30 W3	B30 W4	B30 W5	B30 W6	B30 W7
F8	B31 W0	B31 W1	B31 W2	B31 W3	B31 W4	B31 W5	B31 W6	B31 W7

Address Reference String

A1
65
9C
1C
F0
93
06
34
02
E8

Auto Generate Add. Ref. Str.

Self Generate Add. Ref. Str.

Main Memory Address

TAG	BLOCK	WORD
0	0011	100

Memory Block and Word Bits

BLOCK	WORD
00011	100

PROGRESS UPDATE

Now that the required memory block is in cache we note the following 3 things:
1. The cache block has a tag associated with it and the tag, as specified by the Tag bit in the Main Memory Address, has a value of 0

Restart | Next | Back | Quit

Each location (1C on the figure) is divided in fields

"Main memory address" **1Ch**=0001 1100b, => tag=0, block=0011, word=100

Address partitioning

Why the memory address is divided in this way ?
(Why the address partitioning is made in this way?)
Or: Why there is a **single bit for tag**, **4 bits for block and 3 bits for word** ?

Answer:
- Because the number of blocks or **slots** in cache is 16 = 2^4
  =>it needs 4 bits to specify the "***block***" *field*
- Because the number of words or **locations** from each slot in cache is 8 = 2^3
  =>it needs 3 bits to specify the "***word***" field

(it shows on which position inside of blocks we find the CPU-requested word
- inside the slot)

- The total main memory size gives the number of blocks necessary for a correct writing of memory address
  there are 256 locations in Camera = > memory address is written on **8 bits;**

$$\mathbf{8 - 4 - 3 = 1}$$

$\Rightarrow$ From those **8 bits : 3bits** were used to specify the **word** field, **4 bits** for the **block** field and only **1 bit** for **tag**

Which of those 3 mapping schemes is ***best*** ?

- "Best" = efficiency – ensure a higher number of ***cache hits*** *and*
  a lower effective access time (***EAT****)*
- The 3 cache mapping schemes illustrate the mapping process starting from the generation of the memory address to its mapping in cache, inside a slot
- The difference between the 3 schemes:
  -a different way of partitioning the address
  -cache memory organization

Example: the address 1Ch in an ***fully associative mapping scheme*** will be mapped in set number 3, into any available block (straight) identified by tag: 00, on field: 4

Mapping scheme	Its feature	Cache memory organization	Address partition	Example: 1Ch=0001.1100b
Direct Mapped	**Restrictive**	Blocks/slots, with direct link	**Tag-bloc-word**	**00011100b**
Fully Associative	**Permissive**	Anywhere, without a link	**Tag-word**	**00011100b**
Set Associative 2-ways or 8-sets	**Restrictive-Permissive**	Slot sets ,any slot	**Tag-set-word**	**00011100b**

1. Direct Mapped Cache

Restrictive :

Each block of memory is mapped to exactly one cache block in a modular fashion. The problem arises when the block will be overwritten and the previous block will be lost The tag field is stored with each memory block when it is placed in cache = > the block can be uniquely identified. When cache is searched for a specific memory block, the CPU knows exactly where to find the block just by looking at the main memory address bits.

If we have a scheme with N blocks, the X block will be mapped in the slot Y,
Where **Y=X mod N**

Example:

If there is a 10 slots cache, the slot number 3 has links with blocks 3,13,23,33,43,... from the main memory

Once a memory block is copied into a cache slot, a ***valid bit*** will be set for this cache slot for letting the system knows that slot contains valid data.

2. Fully Associative Cache

Fully permissive

Instead of specifying a unique location for each main memory block, we can look at the opposite extreme: allowing a block of memory to be placed anywhere in cache. While there are empty blocks in cache, there are no problems.
To implement this mapping scheme, we require associative memory so it can be searched in parallel => all the tags will be searched in parallel for a faster searching of data => special hardware, higher costs.
When cache is searched for a specific memory block, the tag field of the main memory block is compared to all the valid tags in cache and, if a match is found, the block is found. **If there is no match**, the memory block needs to be brought into the cache.
When we have Fully Associative scheme and Set Associative scheme, once the cache is full, a **replacement algorithm** is used to evict an existing memory block from cache and place the new memory block instead of the removed one (called ***"the victim block***")

There are more possibilities of replacement policies: LRU, LFU, FIFO, random, etc
In CAMERA, the replacement algorithm used is **Least Recently Used (LRU)**

3. Set Associative Cache

***A combination: Restrictive-* Permissive**

The third mapping scheme is N-way set associative cache mapping and is similar to direct mapped cache because we use the memory address to map to a cache location. The difference is that *an address maps to a set of cache blocks* instead of a single cache block.

When cache is searched for a specific memory block, the **CPU knows to look in a specific cache set** with the help of the main memory address bits. The tag bits then identify the memory block. A replacement algorithm is needed here too, to determine the "victim" block that will be removed from cache to make available free space for a new memory block.

In CAMERA, the replacement algorithm used is the LRU algorithm.

Replacement policy

The chosen replacement policy depends on the locality *that will be exploited,* generally it is used ***temporal locality*** (in time: the one that was least recently used)

Frequently used policies: **LRU, LFU, FIFO, random**

A *least recently used* (**LRU**) policy type: keeps track of what was used when, which is expensive if one wants to make sure the algorithm always discards *the* least recently used item.

Disadvantage: high complexity – it needs to keep a history of visits per block - slows cache function.

A least frequently used (**LFU**) policy type: counts how often an item is needed. Those that are used least often are discarded.

Disadvantage: high complexity - it needs to keep a history of visits per block - slows cache function.

First-in, first-out (**FIFO**): it's a popular policy

- It will replace the block which came first, then second, etc.

A policy ***random*** type: will remove a random block

Which block will be replaced if we have a miss in cache ?

Studies: comparison between LRU and random [ref2]

- Easy to notice for **Direct Mapped**
- Difficult for **Set Associative** or **Fully Associative**: **Random vs LRU**

Associativity:	**2-way**		**4-way**		**8-way**	
Size	**LRU**	**Random**	**LRU**	**Random**	**LRU**	**Random**
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	**1.15%**	1.17%	**1.13%**	**1.13%**	**1.12%**	**1.12%**

- the smaller the memory size, the higher the percentage,
- the smaller the number of ways, the higher the percentage
- random is weaker than LRU (the percentage is higher)

The best way: LRU with 8- ways and larger cache memory

Data passing from one level to another at memory reading