

# Cache and Locality

# Different types of cache.

- **Fully associative**, no sets. Means that we have to search all tags in our cache and then look at block-offset (where in our block).
  - I. Increased Complexity: Because there is no mapping between memory addresses and cache locations, a fully associative cache must search through all of its entries to determine if a block of memory is already in the cache. This increases the complexity of the cache's design and can lead to longer cache lookup times.
  - II. Large size: A fully associative cache must be large enough to hold all possible blocks of memory. This makes fully associative caches less practical in many systems, especially those with limited space.
  - III. Less efficient use of cache space: Fully associative cache's lack of memory mapping might lead to a lower cache hit rate because a block from memory may be replaced by a block from memory that will be not used in a short time.
- **Direct map** – 1way. Each set has exactly 1 block. We still need to compare the tag, and check valid bit.
- **2way set-associative**– 2 blocks per set
- **4way set-associative**– 4 blocks per set
- ... and so on.

- **Write allocate:** fetch on write. Write value to cache from memory on cache miss.
- **no-write-allocate:** cache miss simply writes the value to main memory
- **Writeback:** not all data are written to memory when changed in cache. Data changed in cache are marked as dirty and changes are pushed to memory later. (can be done eg. When a buffer is full)
  - Downside if crash. Not all changed data are written to memory.

# Row wise and column wise

- In C data is written row wise in memory.
- Eg. If we init a 2d array.
- **Input** :  $mat[][] = \{\{1, 2, 3\},$   
                   $\{4, 5, 6\},$   
                   $\{7, 8, 9\}\}$
- **Output** : Row-wise: 1 2 3 4 5 6 7 8 9  
              Col-wise : 1 4 7 2 5 8 3 6 9
- *So we want to access all elements in a row, then proceed to the next row*

```

40 long** row_vector_add(long row_vector[M], long matrix[M][N]) {
41     int i, j;
42     // For each element for the row_vector
43     for (i = 0; i < M; i++) {
44         // Add the element to each element of a column
45         for (j = 0; j < N; j++) {
46             matrix[i][j] += row_vector[i];
47         }
48     }
49     return matrix;
50 }

55 long** col_vector_add(long row_vector[M], long matrix[M][N]) {
56     int i, j;
57     // For each element for the row_vector
58     for (i = 0; i < N; i++) {
59         // Add the element to each element of a column
60         for (j = 0; j < M; j++) {
61             matrix[j][i] += row_vector[j];
62         }
63     }
64     return matrix;
65 }

```

int M = 2000; // rows  
int N = 500; // columns

Time elapsed:  
0.001601 seconds

Time elapsed: 0.003158  
seconds

Row\_vector\_add is  
approx 2x the speed of  
col\_vector\_add.

Depends on input  
parameters.

See code.zip

# Temporal and spatial locality

- **Spatial and temporal locality** form the principle of '**Locality of Reference**'
- **Spatial locality** refers to the likelihood that memory addresses close to each other in memory will be accessed close together in time.
- **Temporal locality** refers to the likelihood that a memory location will be accessed again in the near future. Replacement policies, such as least recently used (LRU), are used to determine which locations to evict when the cache is full.

### 1.3 Data Cache (about 10 %)

In the following we examine a 2-way set-associative data cache with a total of 32 cache-blocks of 16 bytes each.

**Question 1.3.1:** Describe how an address is partitioned into the three components (tag, index, byte-offset) used when accessing such a cache.

The byte offset must be able to address 16 bytes, so it takes up the lowest 4 bits (0-3). The index must be able to select one of 16 sets, so it take up the next 4 bits (4-7). The tag is the rest of the address (8-whatever the address size)

Ekstra info:

Total cache size allocation: (16B + metadata) \* 32 Blocks

Total cache data size: 16B \* 32 Blocks = 512 B

2way; For each set we have 2 cache blocks/lines.

32 cache blocks: total size of cache

16 bytes each: each block has 16 bytes

Number of sets: 32 blocks / 2 blocks per set = 16 sets

Block offset addr size (in bits):  $\log_2(16) = 4$

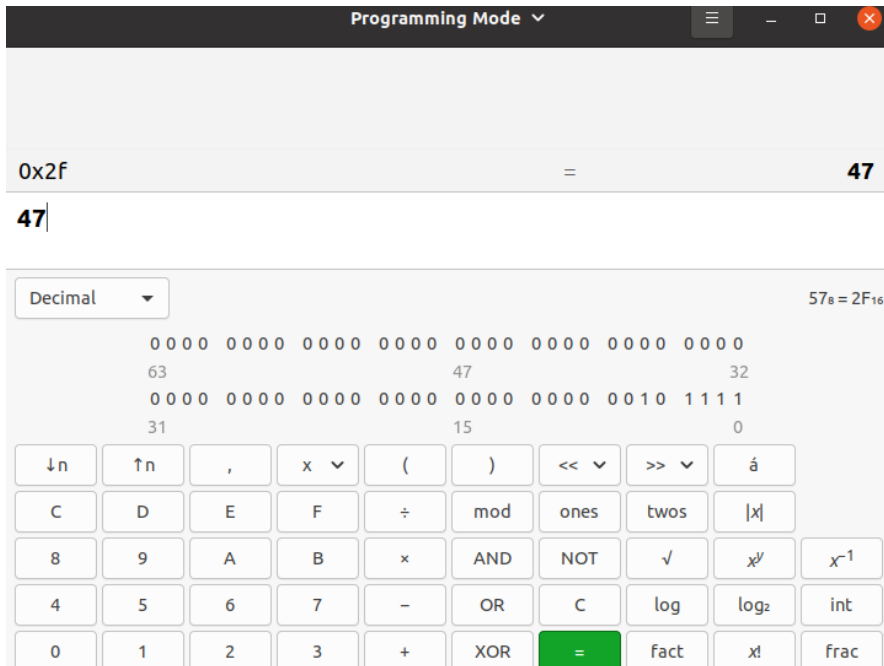
Index (which set) addr size (in bits):  $\log_2(16) = 4$

The tag is just the rest available in the whole addr.  
eg. addr size 16 bits.

0000 0000 0000 0000

# Extract tag and index from addr

Ubuntu has a smart calculator



**Question 1.3.2:** Determine tag and index for each of the addresses in the following list:

0x010, 0x011, 0x02F, 0x112, 0x213, 0x314, 0x11C, 0x715, 0x012, 0x11F, 0x220, 0x020.

Address	tag	index
0x010	0x0	0x1
0x011	0x0	0x1
0x02F	0x0	0x2
0x112	0x1	0x1
0x213	0x2	0x1
0x314	0x3	0x1
0x11C	0x1	0x1
0x715	0x7	0x1
0x012	0x0	0x1
0x11F	0x1	0x1
0x220	0x2	0x2
0x020	0x0	0x2

Look at Hex representation. Each digit in hex is 4 in bit representation.

Or use a calculator to convert.



# Write cache representation

**Question 1.3.3:** Write down a simulation of how the cache state changes as it is accessed according to the above sequence. Assume LRU replacement.

Show the cache content by showing the tags for each valid entry for each index. Ignore invalid entries.

example:

2:{3,5} 3:{1}

means the caches has 3 valid entries, at index 2 (in set nr 2) we have tags 3 and 5, and at index 3 we have the tag 1. keep the tags within each set sorted by time of access, so the newest accessed tag is first. That way you know how to find the least recently used tag for replacement.

Write one line for each access. Show the cache content before and after each access, the accessed address as (index,tag) pair, and whether the access is a hit or miss.

example:

2:{3,5} 3:{1} (1,456) Miss 1:{456}, 2:{3,5}, 3:{1}

	(1,0)	Miss	1:{0}	Index 1 , Tag 0	
1:{0}	(1,0)	Hit	1:{0}	Index 1 , Tag 0	
1:{0}	(2,0)	Miss	1:{0}, 2:{0}	Index 2 , Tag 0	
1:{0}, 2:{0}	(1,1)	Miss	1:{1,0}, 2:{0}	Index 1 , Tag 1	Why not this guy
1:{1,0}, 2:{0}	(1,2)	Miss	1:{2,1}, 2:{0}	Index 1 , Tag 2	
1:{2,1}, 2:{0}	(1,3)	Miss	1:{3,2}, 2:{0}	Index 1 , Tag 3	
1:{3,2}, 2:{0}	(1,1)	Miss	1:{1,3}, 2:{0}	Index 1 , Tag 1	Why not this guy
1:{1,3}, 2:{0}	(1,7)	Miss	1:{7,1}, 2:{0}	Index 1 , Tag 7	
1:{7,1}, 2:{0}	(1,0)	Miss	1:{0,7}, 2:{0}	Index 1 , Tag 0	Why not this guy
1:{0,7}, 2:{0}	(1,1)	Miss	1:{1,0}, 2:{0}	Index 1 , Tag 1	
1:{1,0}, 2:{0}	(2,2)	Miss	1:{1,0}, 2:{2,0}	Index 2 , Tag 2	
1:{1,0}, 2:{2,0}	(2,0)	Hit	1:{1,0}, 2:{0,2}	Index 2 , Tag 0	

# Example direct mapped:

- 8 kibibyte total cache size
- Each block is 8 byte
- Direct mapped and 32-bit addr

Data might be in kibi/kilo-byte


1 kibibyte = 1024 bytes

1 kilobyte = 1000 bytes

- Bitsize of offset =  $\log_2 ( 8 ) = 3$
  - # sets =  $8 \text{ KB} * 1024 \text{ B/KB} / ( 8 \text{ B/block} * 1 \text{ block/set} ) = 1024 \text{ set}$
  - Bitsize of index =  $\log_2 ( 1024 ) = 10$
  - Tag is the rest
- 
- What if fully associative?
  - Its the same, we just dont have a set index.
  - So block-offset is 3 bits, and tag is the rest.

Tag	Set	Offset
000	0 0000 0000	0000


Direct mapped (1-way)

	Address	Index	Tag	Hit/Miss	All cache content after access
0000 0000 0000 0000	0x0000	0	0		
0000 0000 0000 0001	0x0001	0	0		
0000 0000 0000 1111	0x000F	1	0		
0000 0001 0000 0010	0x0102	32	0		
0000 0001 0000 0111	0x0107	32	0		
0000 0000 0000 0010	0x0002	0	0		
1000 0001 0000 0011	0x8103	32	4		
0000 0001 0000 0001	0x0101	32	0		


Tag	Set	Offset
000	0 0000 0000	0000

	Address	Index	Tag	Hit/Miss	All cache content after access
0000 0000 0000 0000	0x0000	0	0	miss	0 : {0}
0000 0000 0000 0001	0x0001	0	0		
0000 0000 0000 1111	0x000F	1	0		
0000 0001 0000 0010	0x0102	32	0		
0000 0001 0000 0111	0x0107	32	0		
0000 0000 0000 0010	0x0002	0	0		
1000 0001 0000 0011	0x8103	32	4		
0000 0001 0000 0001	0x0101	32	0		

Tag	Set	Offset
000	0 0000 0000 0000	0000

	Address	Index	Tag	Hit/Miss	All cache content after access
0000 0000 0000 0000	0x0000	0	0	miss	0 : {0}
0000 0000 0000 0001	0x0001	0	0	hit	0 : {0}
0000 0000 0000 1111	0x000F	1	0		
0000 0001 0000 0010	0x0102	32	0		
0000 0001 0000 0111	0x0107	32	0		
0000 0000 0000 0010	0x0002	0	0		
1000 0001 0000 0011	0x8103	32	4		
0000 0001 0000 0001	0x0101	32	0		

Tag	Set	Offset
000	0 0000 0000 0000	0000

	Address	Index	Tag	Hit/Miss	All cache content after access
0000 0000 0000 0000	0x0000	0	0	miss	0 : {0}
0000 0000 0000 0001	0x0001	0	0	hit	0 : {0}
0000 0000 0000 1111	0x000F	1	0	miss	0 : {0}, 1 : {0}
0000 0001 0000 0010	0x0102	32	0		
0000 0001 0000 0111	0x0107	32	0		
0000 0000 0000 0010	0x0002	0	0		
1000 0001 0000 0011	0x8103	32	4		
0000 0001 0000 0001	0x0101	32	0		

Tag	Set	Offset
000	0 0000 0000 0000	0000

	Address	Index	Tag	Hit/Miss	All cache content after access
0000 0000 0000 0000	0x0000	0	0	miss	0 : {0}
0000 0000 0000 0001	0x0001	0	0	hit	0 : {0}
0000 0000 0000 1111	0x000F	1	0	miss	0 : {0}, 1 : {0}
0000 0001 0000 0010	0x0102	32	0	miss	0 : {0}, 1 : {0}, 32 : {0}
0000 0001 0000 0111	0x0107	32	0		
0000 0000 0000 0010	0x0002	0	0		
1000 0001 0000 0011	0x8103	32	4		
0000 0001 0000 0001	0x0101	32	0		

Tag	Set	Offset
000	0 0000 0000 0000	0000

	Address	Index	Tag	Hit/Miss	All cache content after access
0000 0000 0000 0000	0x0000	0	0	miss	0 : {0}
0000 0000 0000 0001	0x0001	0	0	hit	0 : {0}
0000 0000 0000 1111	0x000F	1	0	miss	0 : {0}, 1 : {0}
0000 0001 0000 0010	0x0102	32	0	miss	0 : {0}, 1 : {0}, 32 : {0}
0000 0001 0000 0111	0x0107	32	0	hit	0 : {0}, 1 : {0}, 32 : {0}
0000 0000 0000 0010	0x0002	0	0		
1000 0001 0000 0011	0x8103	32	4		
0000 0001 0000 0001	0x0101	32	0		



Tag	Set	Offset
000	0 0000 0000 0000	0000

	Address	Index	Tag	Hit/Miss	All cache content after access
0000 0000 0000 0000	0x0000	0	0	miss	0 : {0}
0000 0000 0000 0001	0x0001	0	0	hit	0 : {0}
0000 0000 0000 1111	0x000F	1	0	miss	0 : {0}, 1 : {0}
0000 0001 0000 0010	0x0102	32	0	miss	0 : {0}, 1 : {0}, 32 : {0}
0000 0001 0000 0111	0x0107	32	0	hit	0 : {0}, 1 : {0}, 32 : {0}
0000 0000 0000 0010	0x0002	0	0	hit	0 : {0}, 1 : {0}, 32 : {0}
1000 0001 0000 0011	0x8103	32	4		
0000 0001 0000 0001	0x0101	32	0		

Tag	Set	Offset
000	0 0000 0000 0000	0000

	Address	Index	Tag	Hit/Miss	All cache content after access
0000 0000 0000 0000	0x0000	0	0	miss	0 : {0}
0000 0000 0000 0001	0x0001	0	0	hit	0 : {0}
0000 0000 0000 1111	0x000F	1	0	miss	0 : {0}, 1 : {0}
0000 0001 0000 0010	0x0102	32	0	miss	0 : {0}, 1 : {0}, 32 : {0}
0000 0001 0000 0111	0x0107	32	0	hit	0 : {0}, 1 : {0}, 32 : {0}
0000 0000 0000 0010	0x0002	0	0	hit	0 : {0}, 1 : {0}, 32 : {0}
1000 0001 0000 0011	0x8103	32	4	miss	0 : {0}, 1 : {0}, 32 : {4}
0000 0001 0000 0001	0x0101	32	0		

Tag	Set	Offset
000	0 0000 0000 0000	0000

	Address	Index	Tag	Hit/Miss	All cache content after access
0000 0000 0000 0000	0x0000	0	0	miss	0 : {0}
0000 0000 0000 0001	0x0001	0	0	hit	0 : {0}
0000 0000 0000 1111	0x000F	1	0	miss	0 : {0}, 1 : {0}
0000 0001 0000 0010	0x0102	32	0	miss	0 : {0}, 1 : {0}, 32 : {0}
0000 0001 0000 0111	0x0107	32	0	hit	0 : {0}, 1 : {0}, 32 : {0}
0000 0000 0000 0010	0x0002	0	0	hit	0 : {0}, 1 : {0}, 32 : {0}
1000 0001 0000 0011	0x8103	32	4	miss	0 : {0}, 1 : {0}, 32 : {4}
0000 0001 0000 0001	0x0101	32	0	miss	0 : {0}, 1 : {0}, 32 : {0}