Module 4. Memory Hierarchy
Cache Memory

Taken from:

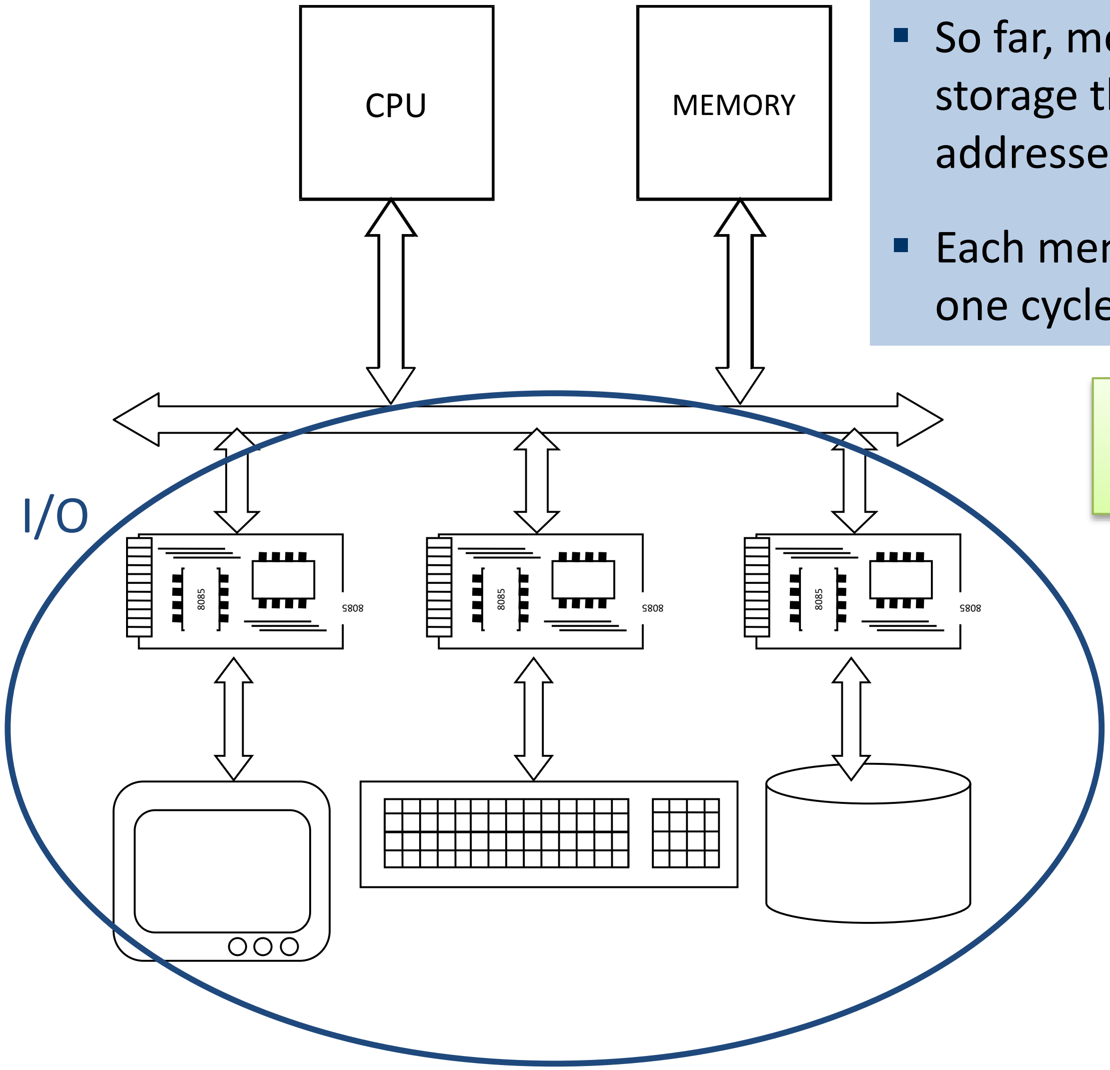
Patterson, D . Hennessy, J. L. “Computer Organization and Design. The hardware/software interface” , 5th ed. Chapter 5

Hennessy, J. L., Patterson, D. “Computer Architecture: A Quantitative Approach”, 5th ed. Chapter 2 and Apendix B.1,2,3

Module 4.1. Cache memory

- Introduction **Patterson. Sections 5.1,2 or Hennessy. Section 2.1**
  - Locality principle
  - Memory Hierarchy
- Cache memories **Patterson. Sections 5.3,4,8**
  - Placement policies
  - Update policies
  - Replacement policies
- Performance **Patterson. Section 5.4**
- How can we improve the cache performance? **Patterson. Section 5.4 or Hennessy. Sections 2.2 and Apendix B.3**
  - Reducing miss rate
  - Reducing miss penalty
  - Reducing hit time
  - Increasing bandwidth

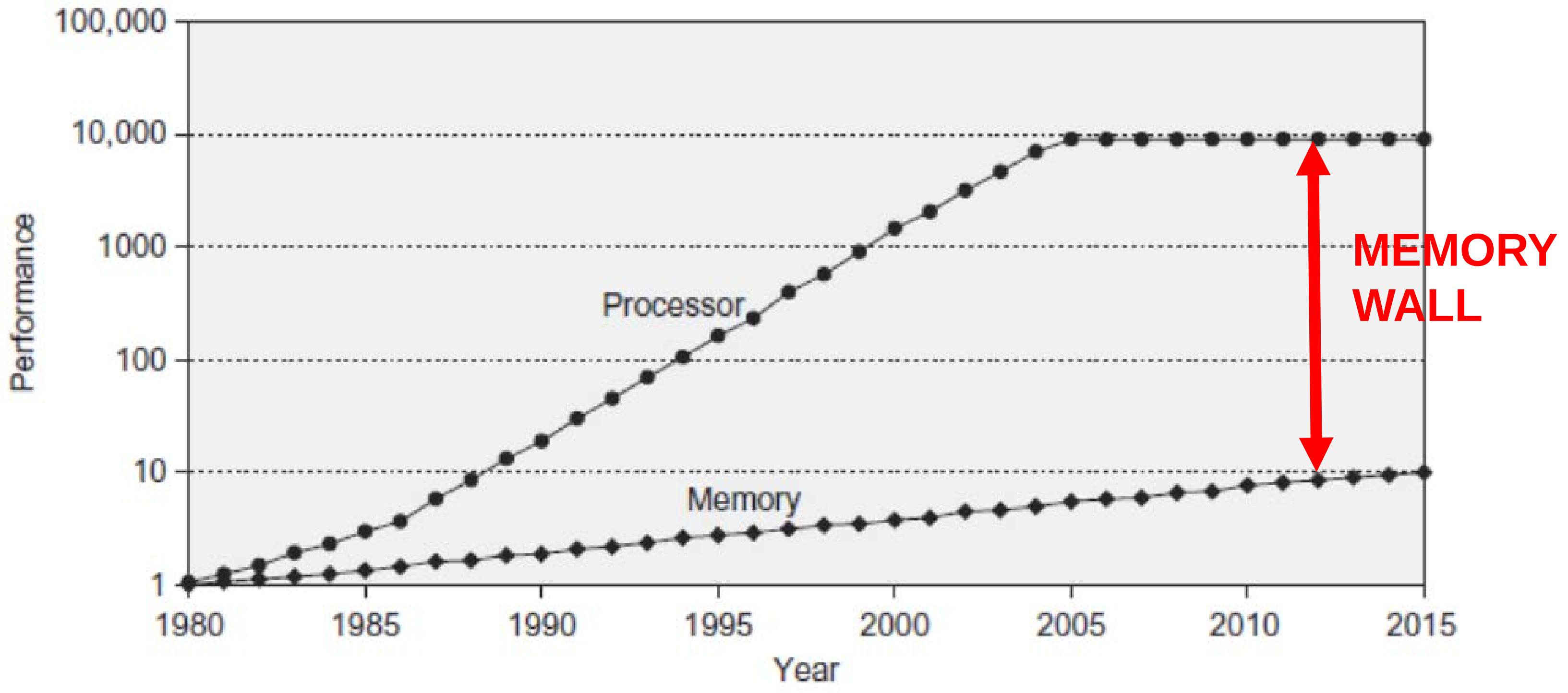
Where are we?



- So far, memory is a linear array of storage that is referenced using the addresses.
- Each memory access is performed in one cycle (IF or M stage)

Programmers want UNLIMITED amounts of FAST memory

First problem: memory wall



The rate of improvement in microprocessor speed exceeds the rate of improvement in DRAM memory speed. (Hitting the memory Wall: implications of the obvious, W. Wulf and S. McKee, 1994).

Second problem: Different technologies for storage

- Typical values for access time/price per GByte (2012)

Memory type	Access Time (ns)	$ per Gbyte (2008)	$ per Gbyte (2012)
SRAM	0.5-2.5	2000-5000	500-1000
DRAM	50-70	20-75	10-20
Flash	5.000-50.000		0,75-1,00
Magnetic Disc	5.000.000-20.000.000	0.20-2	0,05-0,10

Patterson & Hennessy

There is no winner!

Solution: Memory Hierarchy

- Goal:
  – Large and fast memory system, at minimum cost
  – Transparent to the programmer
- Basics:
  – Principle of Locality
- How should we organize the memory?
  – Memory hierarchy

Principle of Locality (of references)

- Most programs exhibit locality, which the cache can take advantage of:
  - The principle of temporal locality says that if a program accesses one memory address, it will tend to access the same address again.
  - The principle of spatial locality says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses soon.

Principle of Locality: example

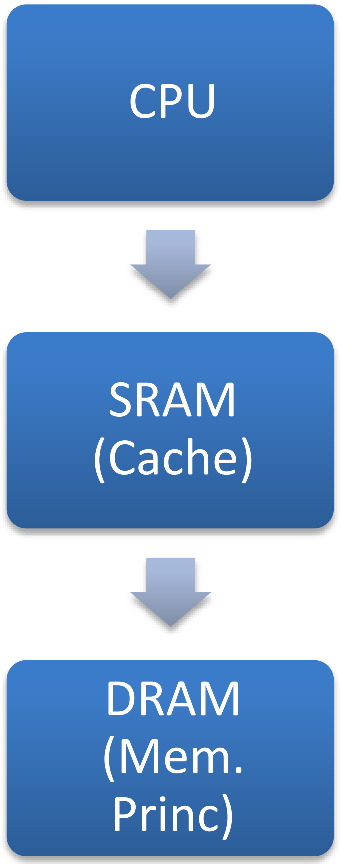
```
int A[16][128]={all values};
int B[16][128]={all values};
int C[16];

int main() {
        int tmp,i,j ;

        for (i=0;i<16;i++) {
                tmp=0;
                for (j=0;j<128;j++)
                        tmp+=A[i][j]*B[i][j];
                C[i]=tmp;
        }
return 0;
}
```

Spatial locality:
After referencing A[0][0] the program will reference A[0][1]

Temporal locality:
This code will be referenced 128 consecutive times

How can locality be exploited?

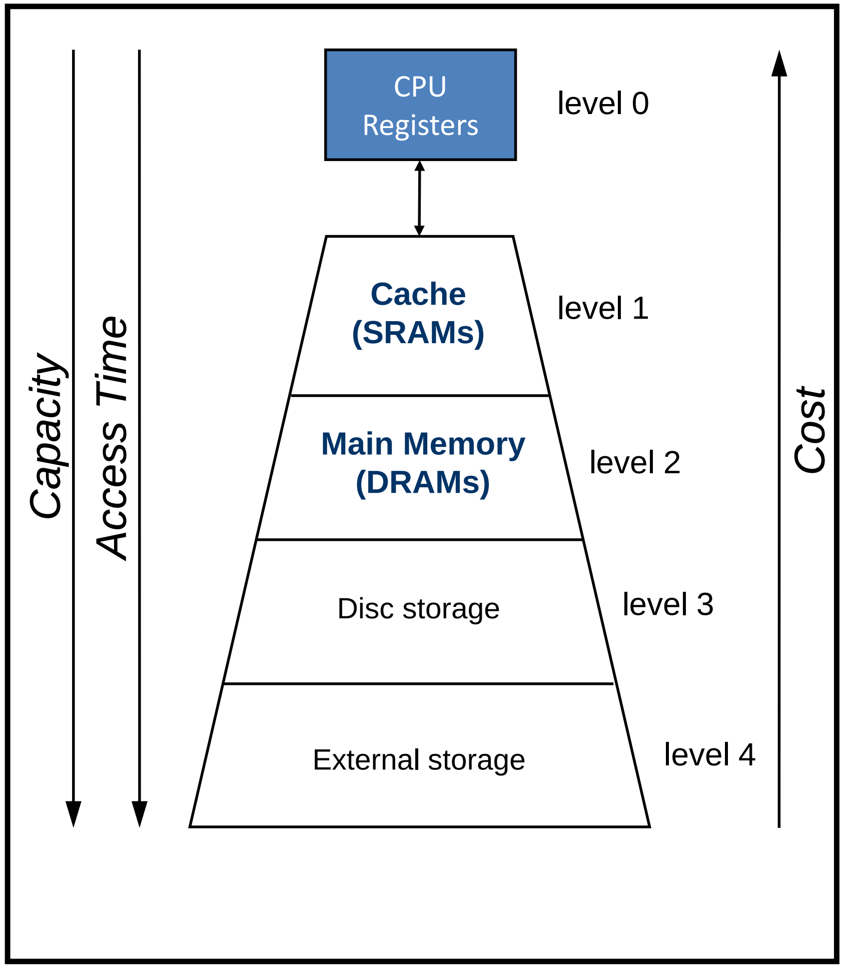
- For every memory access performed by the CPU, the cache controller copies the accessed data to a smaller and faster memory module (SRAM cache)
  - Penalty on the first access
  - Recovered if the CPU wants to access the same data in the future, likely by temporal locality principle.
- The memory controller copies not only the accessed data but a block of data
  - Penalty on the first access
  - Recovered in the future if the CPU wants to access nearby data, likely by spatial locality principle



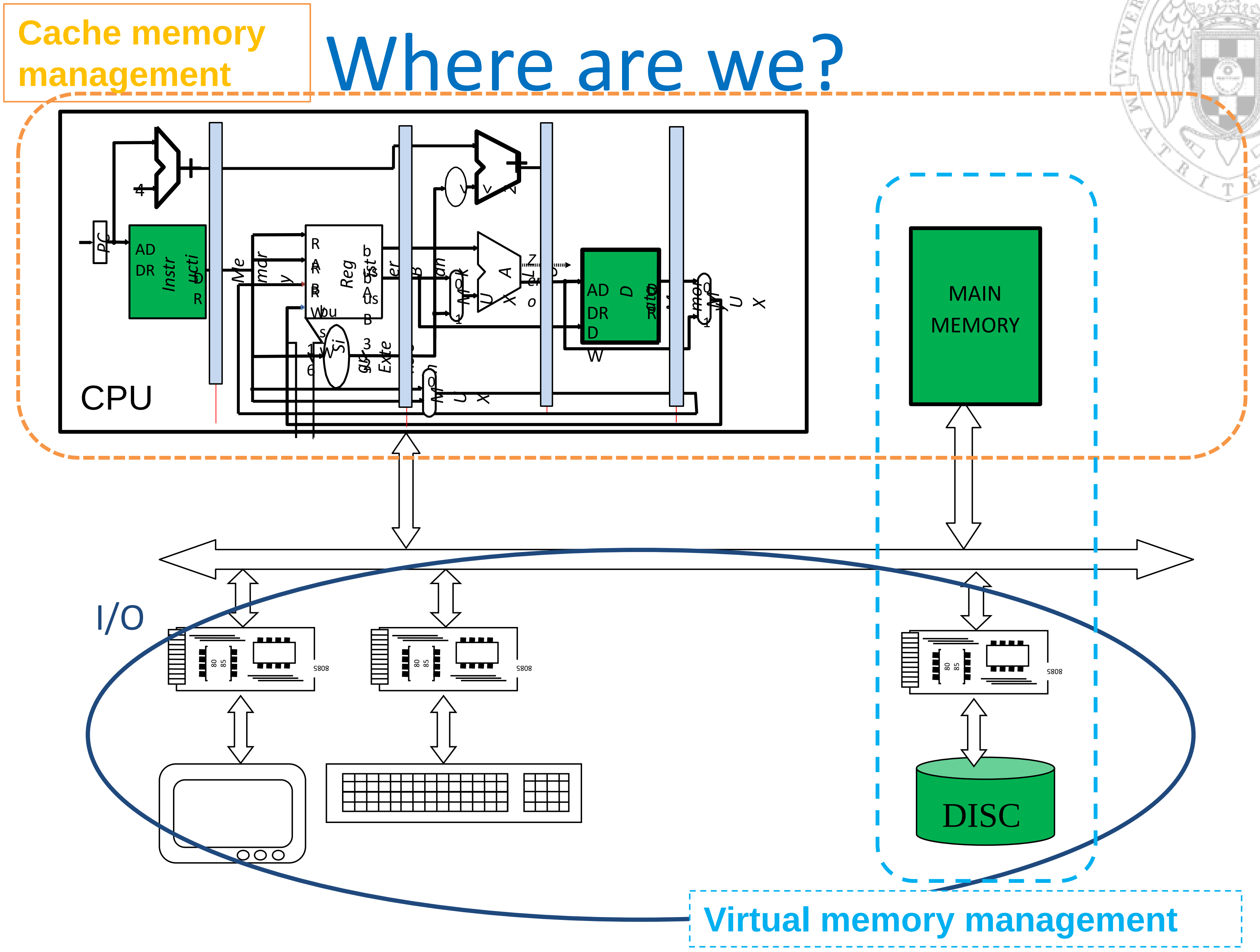
Memory hierarchy

Levels of the memory hierarchy

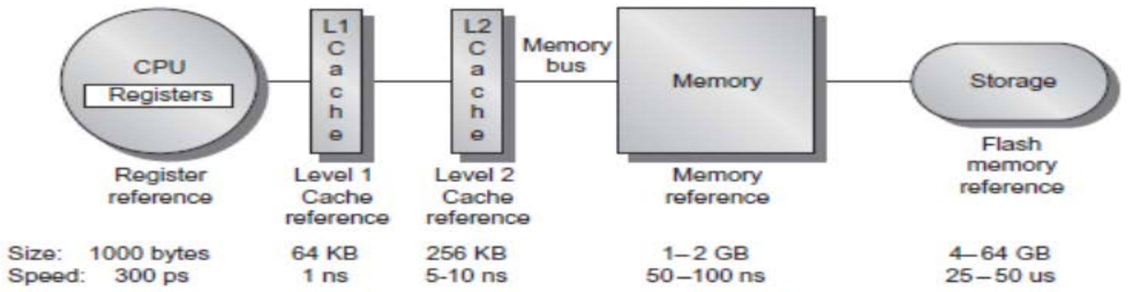
- The memory system of a computer is composed of several levels, with different sizes and technologies, hierarchically organized:
  - Register Bank an the CPU
  - Cache levels (SRAM)
  - Main Memory (DRAM)
  - Secondary memory (discs)
  - External storage: Pen-drives, DVDs
- The cost of the memory system exceeds the cost of the CPU
  - It is very important to optimize its usage



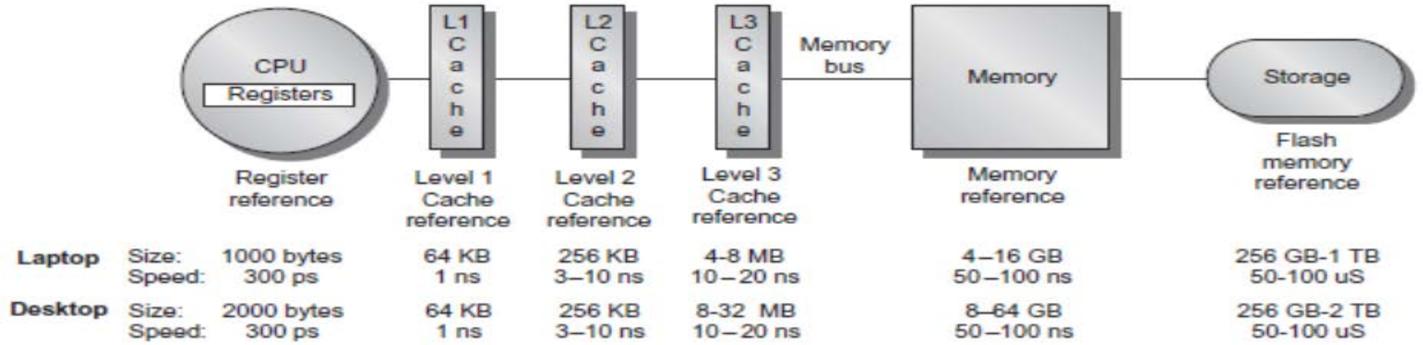
Where are we?


Cache memory management
CPU
MAIN MEMORY
I/O
DISC
Virtual memory management

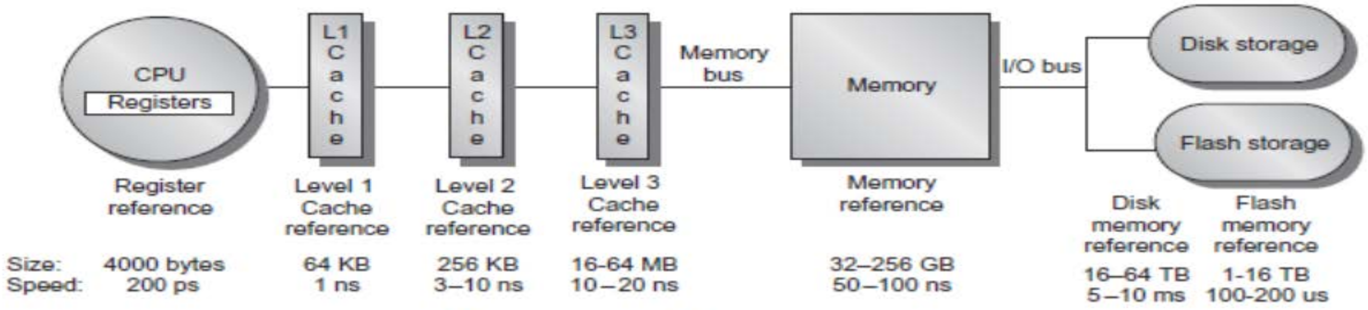
Memory hierarchy: some examples



(A) Memory hierarchy for a personal mobile device


CPU
Registers
L1 Cache
L2 Cache
L3 Cache
Memory bus
Memory
Storage
Register reference
Level 1 Cache reference
Level 2 Cache reference
Level 3 Cache reference
Memory reference
Flash memory reference
Laptop Size: 1000 bytes | 64 KB | 256 KB | 4-8 MB | 4–16 GB | 256 GB-1 TB
Speed: 300 ps | 1 ns | 3–10 ns | 10–20 ns | 50–100 ns | 50-100 uS
Desktop Size: 2000 bytes | 64 KB | 256 KB | 8-32 MB | 8–64 GB | 256 GB-2 TB
Speed: 300 ps | 1 ns | 3–10 ns | 10–20 ns | 50–100 ns | 50-100 uS

(B) Memory hierarchy for a laptop or a desktop



(C) Memory hierarchy for server

CO

Memory Hierarchy Properties

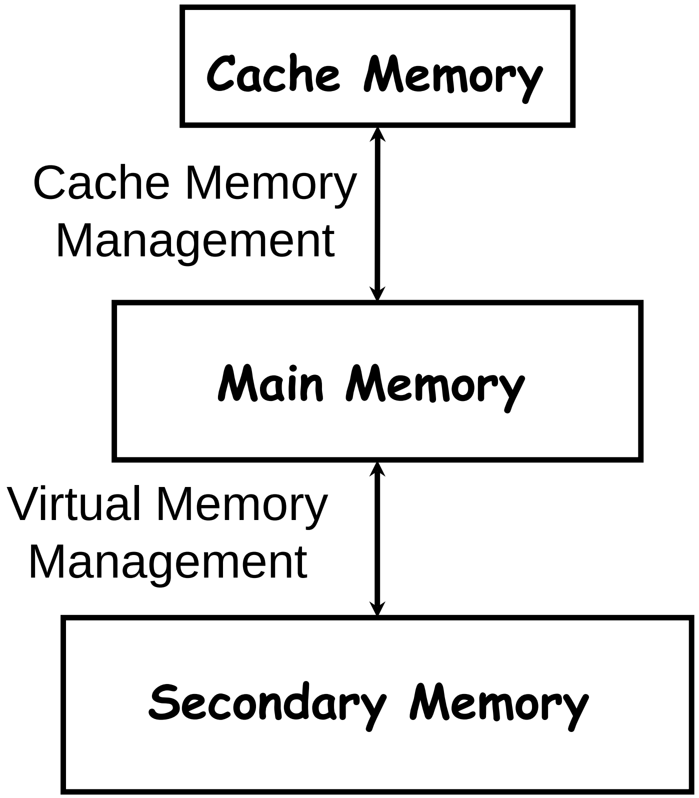
- Inclusion
  - Any information stored at a memory level Mi has to be also stored in levels Mi+1, Mi+2, …, Mn.

    i.e: $M1 \subset M2 \subset \ldots \subset Mn$

    It is not followed in all systems.
- Coherence:
  - The copies of the same information in different levels of the hierarchy must be coherent (identical)
  - If a block of data is modified in level Mi, al the copies in levels Mi+1,.., Mn must be also modified (updated)
- Locality
  - The memory references generated by the CPU, for data or instructions access, are gathered in time (repeated in short intervals) and space (proximity between address)

Memory Hierarchy

- **Cache Memory Management**
  - Controls the data transfers between the main memory and the Cache Memory
  - It involves specialized cache controllers
- **Virtual memory management**
  - Controls the data transfers between the secondary storage and the main memory
  - Accelerated by specialized hardware called Memory Management Unit (MMU). The OS is responsible of the virtual memory management.



Memory hierarchy management

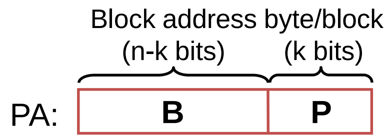
- **Block**: minimum transfer unit between two levels
- **Hit**: the requested data is in the cache
  - ***Hit ratio at level i***: the ration of hit access and total access to the level i cache
  - ***Hit time***: time to detect the hit + time to access the ith level data array. (i)
- **Miss**: the requested data is not in cache, it is necessary to get it from main memory
  - ***Miss ratio***: 1 – (Hit ratio)
  - ***Miss Penalty***: time required to copy one data block from level i+1 to level i, when the referenced block is not in level i.

Memory Hierarchy Management

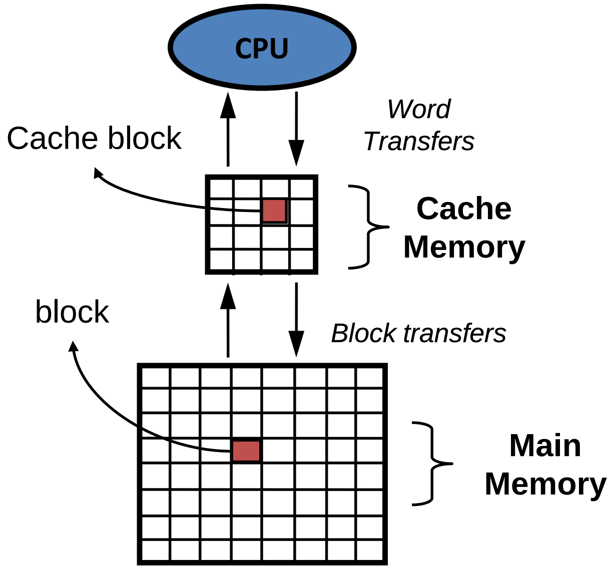
- When the CPU wants the data on a given address it first looks for it in the Cache
  - If the reference is not in cache: **MISS**
  - On a miss, a complete **BLOCK** of data is copied from the main memory to the cache
  - *By temporal locality principle*
    - The same address will likely be referenced in the near future
    - These subsequent accesses will **HIT** the cache
  - *By spatial locality principle*
    - Some subsequent access will likely be for the same data block
    - These accesses will **HIT** the cache

Cache Memory

- **Structure** of the cache memory:
  - *MM (main memory):*
    - Composed of 2^n addressable bytes
    - Organized en nB blocks of 2^K bytes:
  - Fields of the memory address

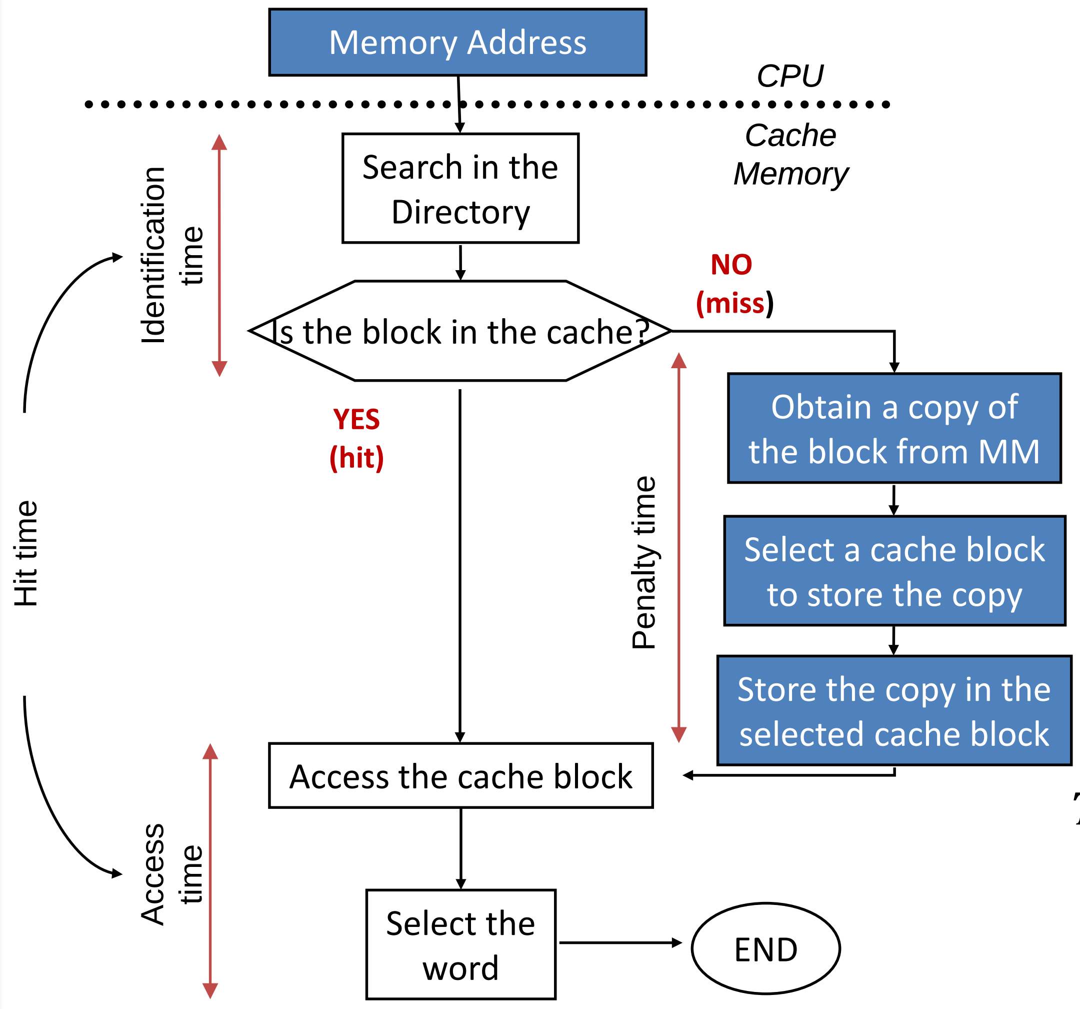


  - *CM (cache memory):*
    - Composed of nM blocks (or lines) of 2^K bytes (nM<<nB)
  - *Directory:*
    - Indicates which memory block is copied on each cache block.



nB: number of memory blocks
nM: number of cache blocks
B: block address
M: cache block address
P: position of the word In the block

Cache Memory Access



- Maximize the hit rate
- Minimize the penalty time
- Reduce the hardware cost

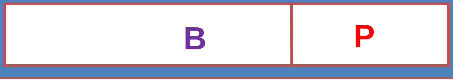
$$T_{total} = T_{hit} + (1 - \overset{\uparrow}{H})T_{penalty}$$

(hit rate)

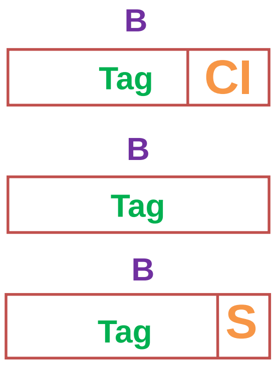
Cache Memory Policies

- How do we know if the desired word is in cache?
- And if it is, how do we find it?

PLACEMENT POLICIES: Memory address → Cache block



- Placement policy:
  - There are less cache blocks than blocks in main memory
  - It determines which cache block corresponds to a given memory block
- Three policies exist:
  - Direct mapping: the block with number CI (cache index)
  - Fully associative mapping: any block in the cache
  - N-way set associative mapping: any of the N cache blocks of set S



Placement Policies

- For all the examples:
  - Main memory of 4KB = 2^{12} B => n = 12
  - Block size: 128B = 2^7 B, k = 7

$$2^{12}/2^{7} = 2^{5} = 32$$

- Main memory with 32 blocks

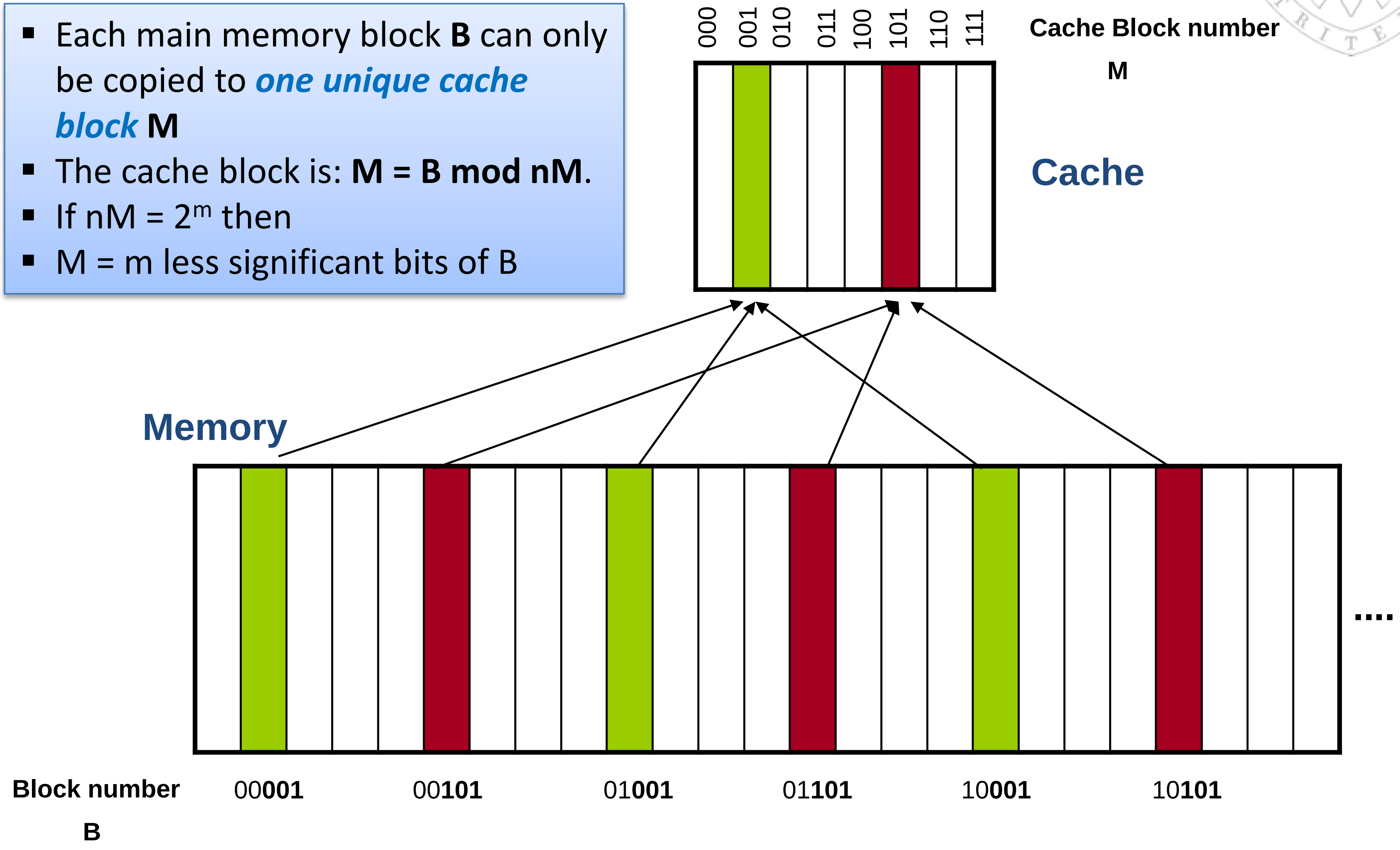
$$\text{PA: } \underbrace{\boxed{B}}_{5}\underbrace{\boxed{P}}_{7}$$

Cache memory with 8 blocks

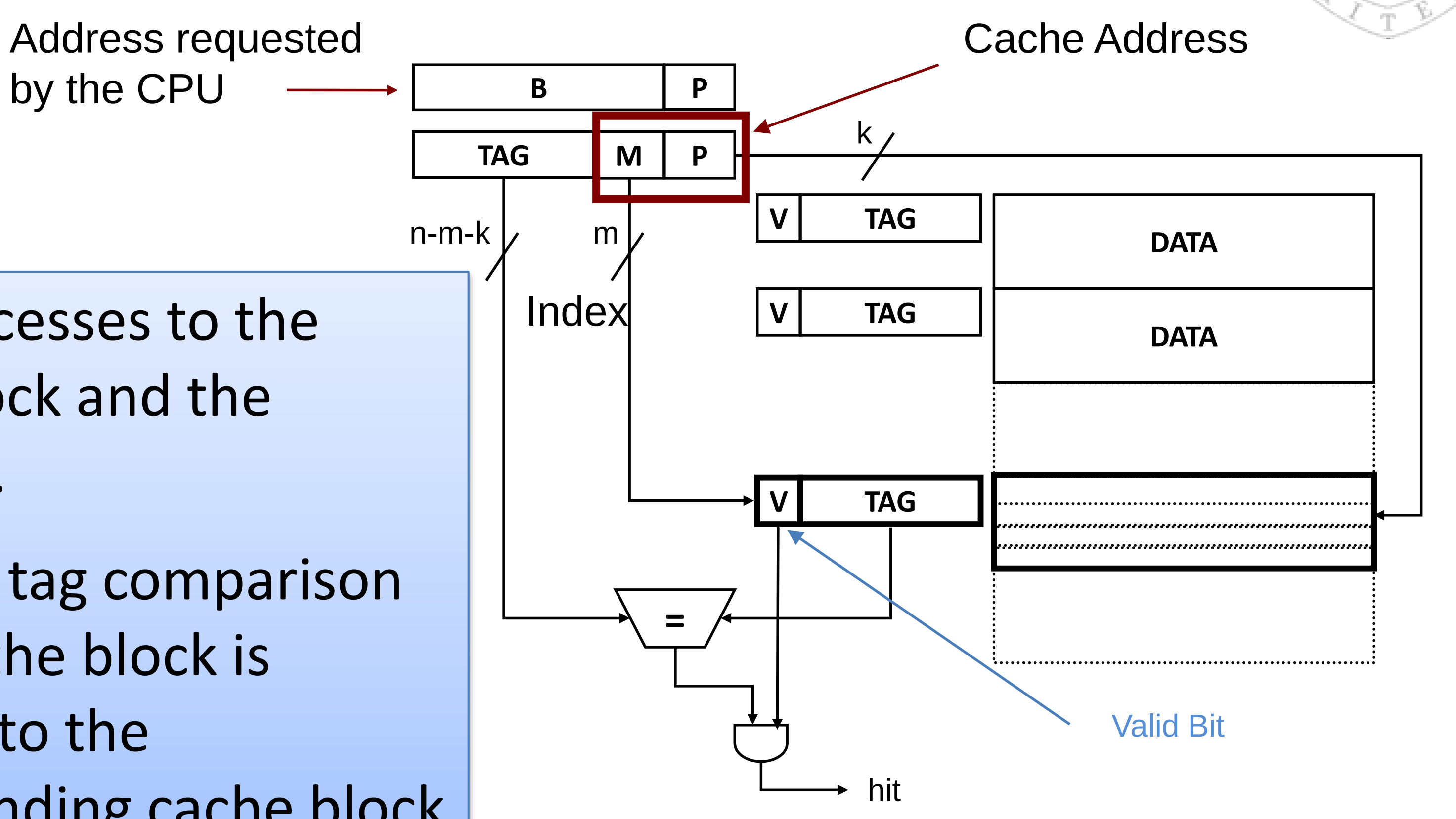
00000000…0 … **00000**111…1	Block 0
00001000…0 … **00001**111…1	Block 1
…	
11111000…0 … **11111**111…1	Block 31

Direct Mapping

- Each main memory block **B** can only be copied to ***one unique cache block*** **M**
- The cache block is: **M = B mod nM**.
- If nM = 2^m then
- M = m less significant bits of B



Direct Mapping

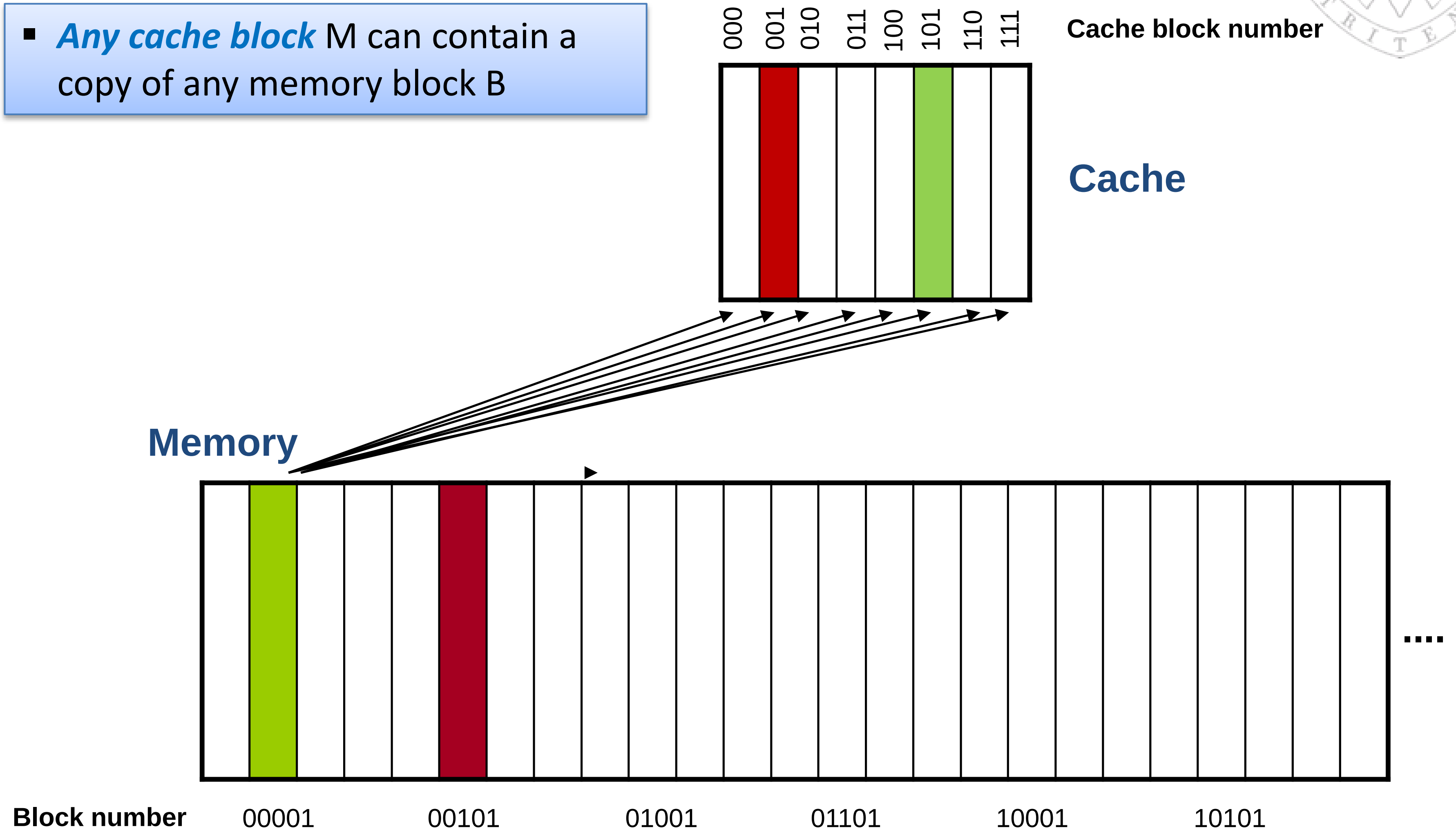


- Direct accesses to the cache block and the directory.
- Only one tag comparison to see if the block is copied into the corresponding cache block

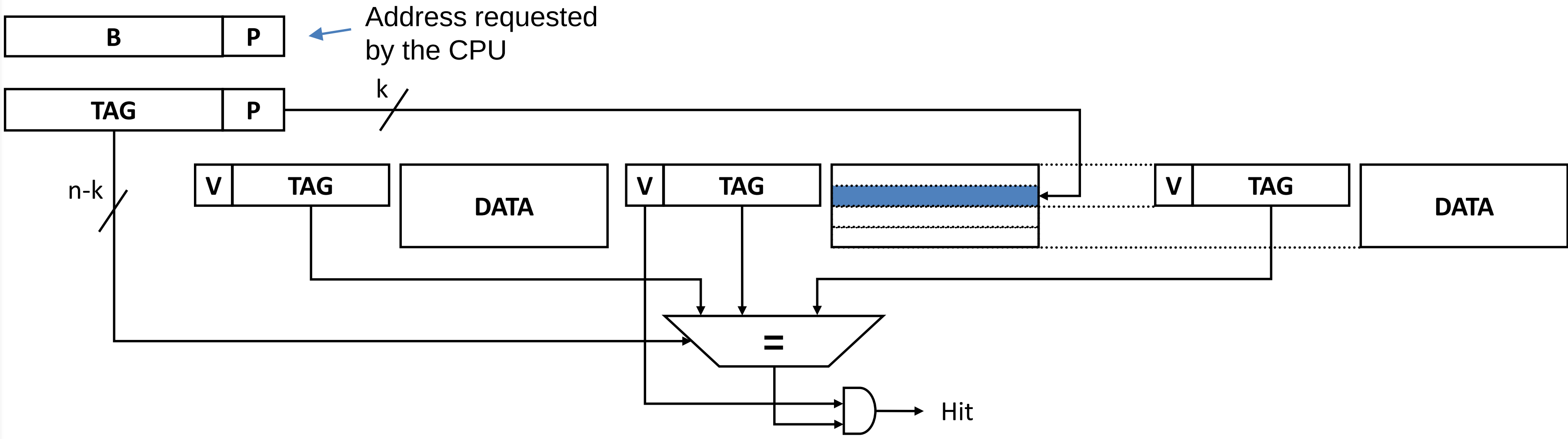
What is the value of m for our example?

Fully associative mapping

- ***Any cache block*** M can contain a copy of any memory block B



Fully Associative Mapping

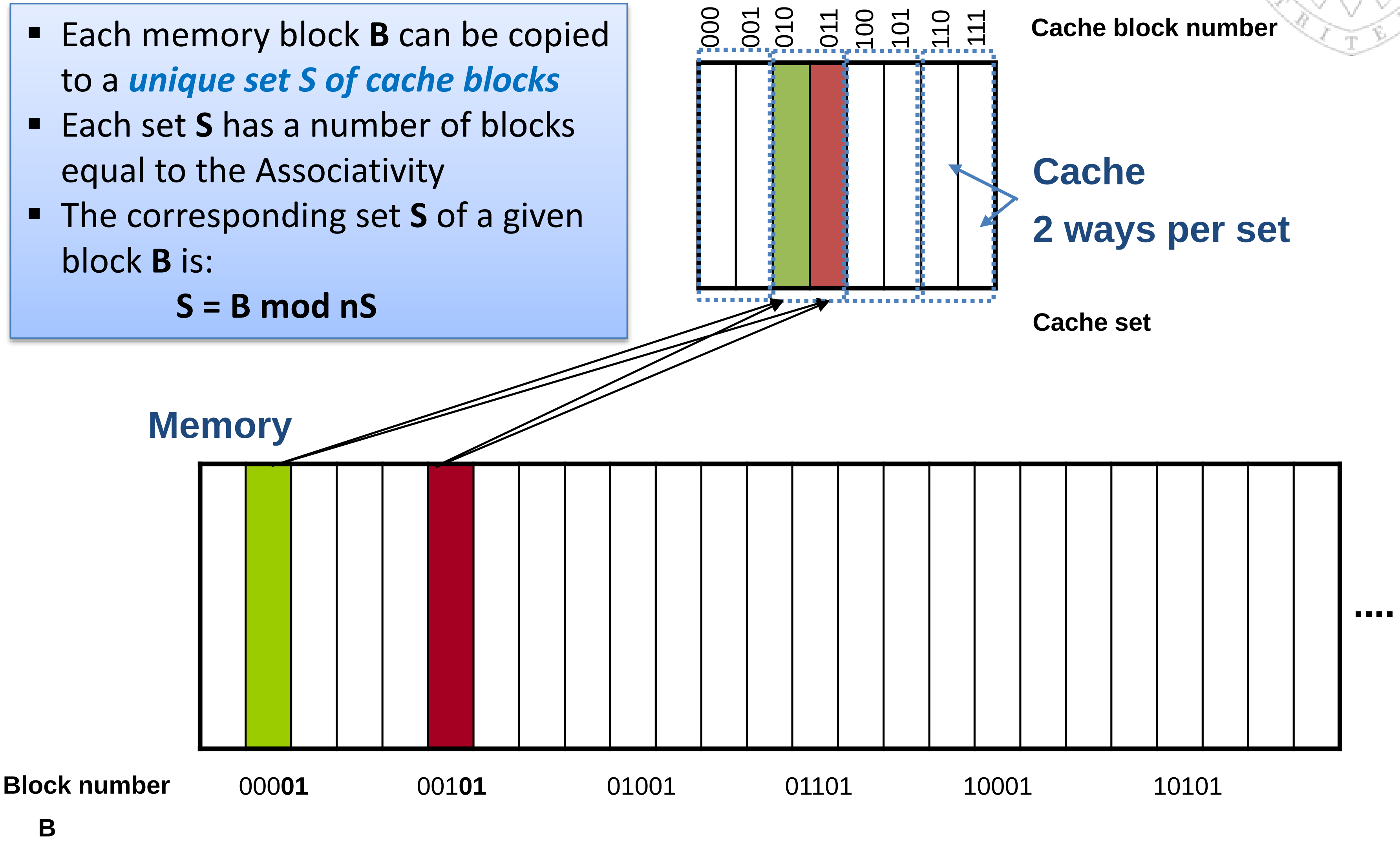


- The Tag of the requested address has to be compared with the tag of every cache block
- The whole block number is used as tag

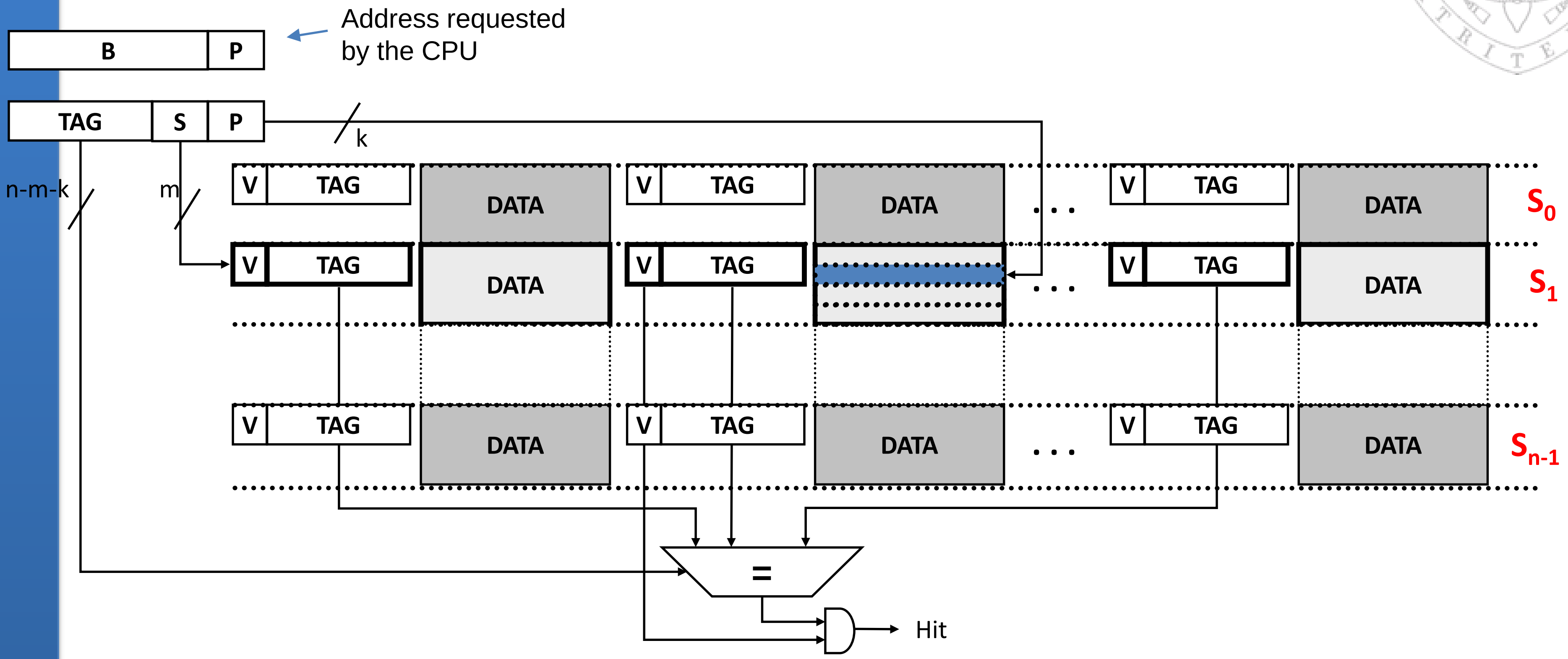
2-ways Set Associative Mapping

- Each memory block **B** can be copied to a ***unique set S of cache blocks***
- Each set **S** has a number of blocks equal to the Associativity
- The corresponding set **S** of a given block **B** is:

  S = B mod nS



N-ways Set Associative Mapping



- The Directory stores for each cache block a TAG, formed by the n-k-m most significant bits of the block address stored
- Direct access to the Set, associative access to the block

What is the value of m for our example?

2-ways Set Associative Mapping

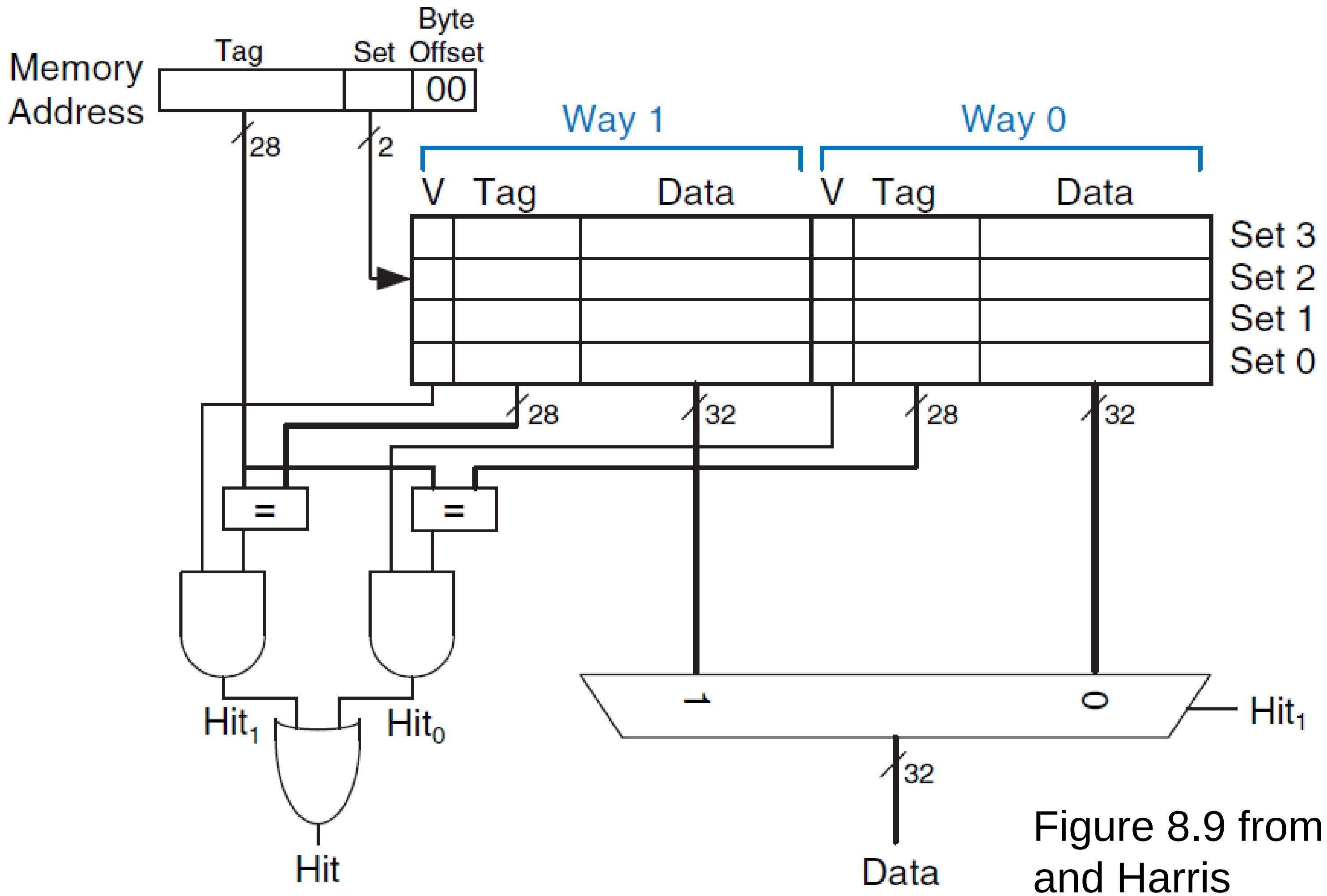


Figure 8.9 from Harris and Harris

Set Associative Mapping

- The ratio nM/nS is known as degree of associativity or number of ways
  - Degree of associativity = 1, equivalent to Direct Mapping
  - Degree of associativity = nM, equivalent to Fully Associative Mapping

Placement Policies

Placement Policy	Advantages	Disadvantages
Direct	Simple and fast access	High miss ratio when several memory blocks compete for the same cache block
Full Associative	Maximum cache usage	Increases the access time and energy consumption
Set Associtive	Intermediate between direct and full associative mappings. The associative degree affects performance, when it is increased the conflict misses decrease Optimal degree: between 2 y 16 (empirical) Most common degree: 4	Increasing the associativity degree increases the access time, the energy consumption and the hardware cost

Update policies

- **What happens when we write to a block?**
  - Do we update only the cache? Only main memory? Both?
- **Write-through:** all writes update both the cache and the main memory
  - On block replacement we can just invalidate the block: the data is already copied to memory
  - We only need one valid bit for each cache block
- **Write-back:** Writes update only the cache memory
  - On replacement we must first update the main memory
  - Two control bits, (V) valid and (D) dirty.

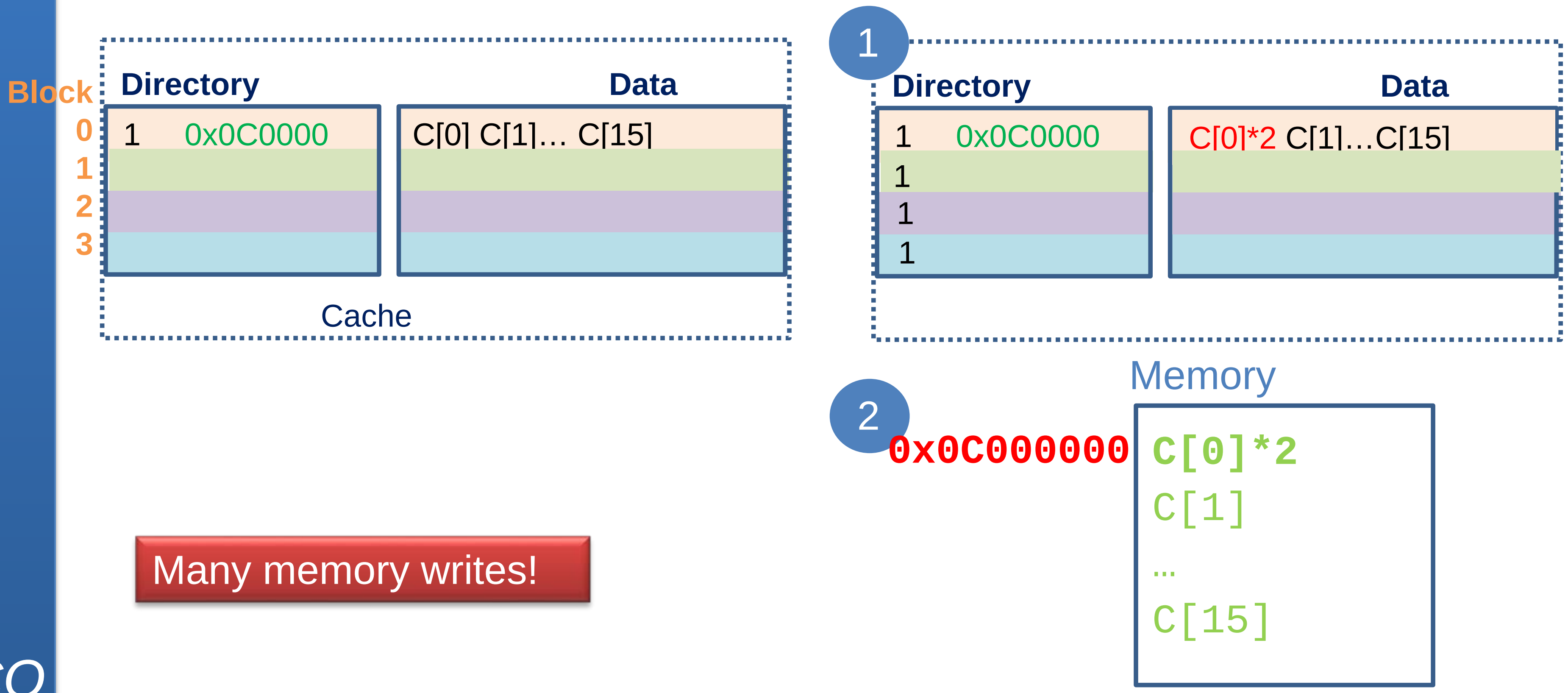
- **What happens on a write miss?**

- **Write allocate**: on a write miss the memory block is copied into a cache block
- **Write no-allocate**: on a write miss, the write goes directly to main memory (or next cache level) without affecting the cache.

Update policies: write-through

```
for (i=0;i<16;i++) {
      C[i]=C[i]*2; }
```

Write-through
1. Write in the cache block
2. Write in the memory

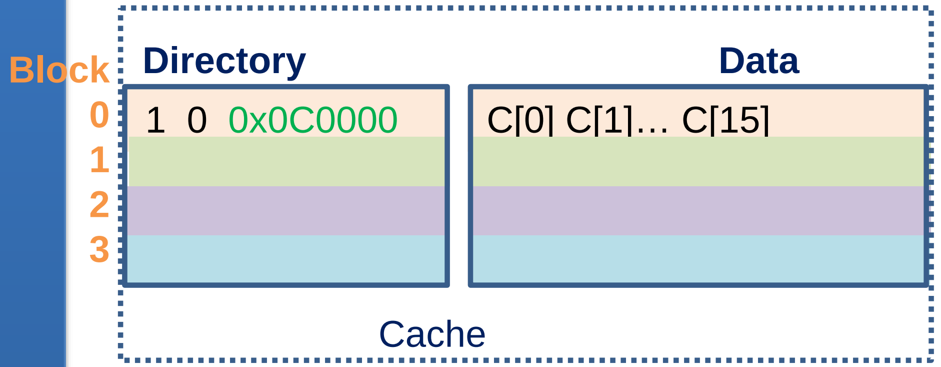


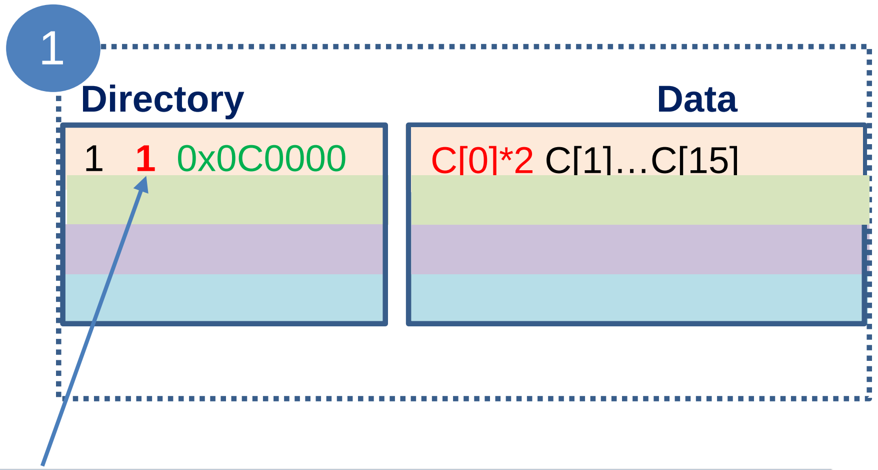
Many memory writes!

Update policies: write-back

```
for (i=0;i<16;i++) {
        C[i]=C[i]*2; }
```

Write-back
1. Write in the cache block





Dirty bit: cache block has been modified
When it is evicted from the cache, it has to be writen back to memory.

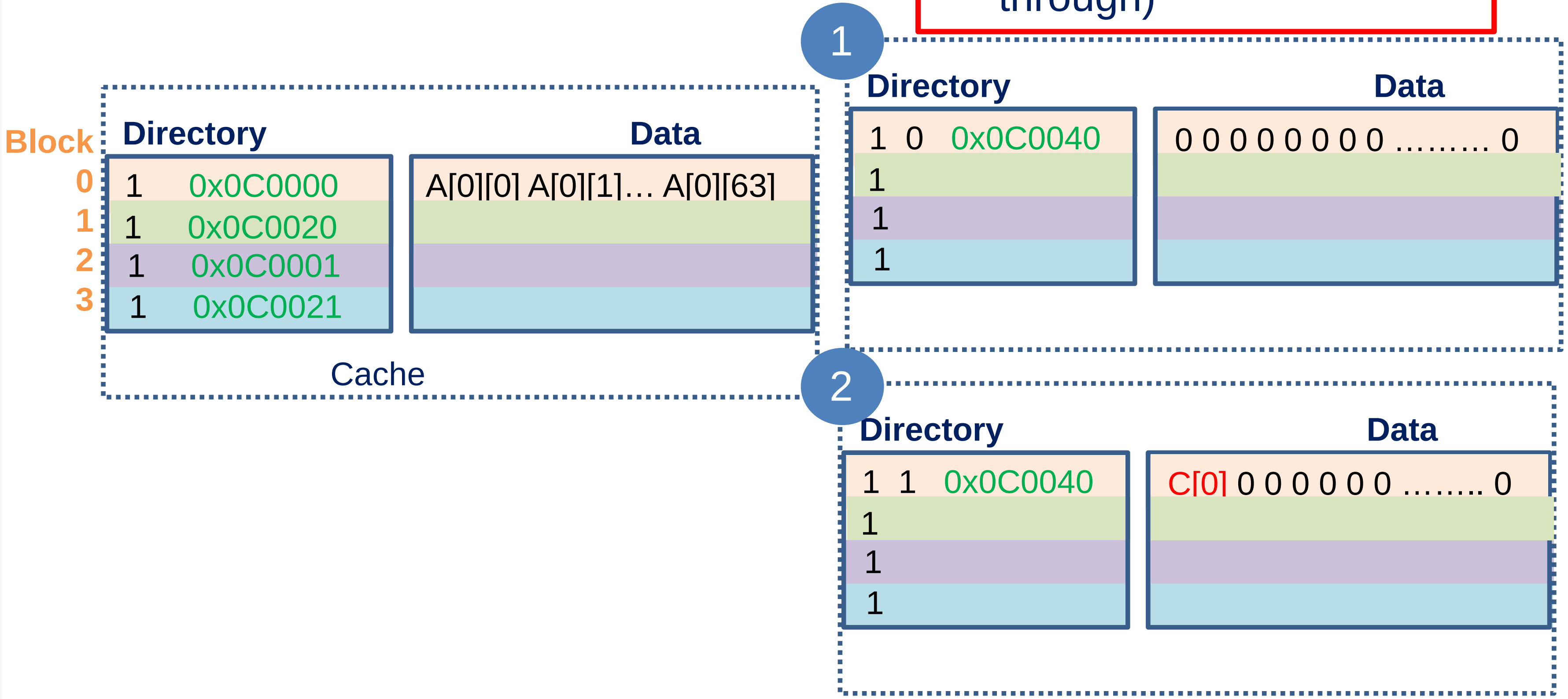
Cache and memory are inconsistent!

Update policies: write allocate

```
for (i=0;i<16;i++) {
   tmp=0;
   for (j=0;j<128;j++)
        tmp+=A[i][j] * B[i][j];
   C[i]=tmp; }
```

Write miss: Write allocate

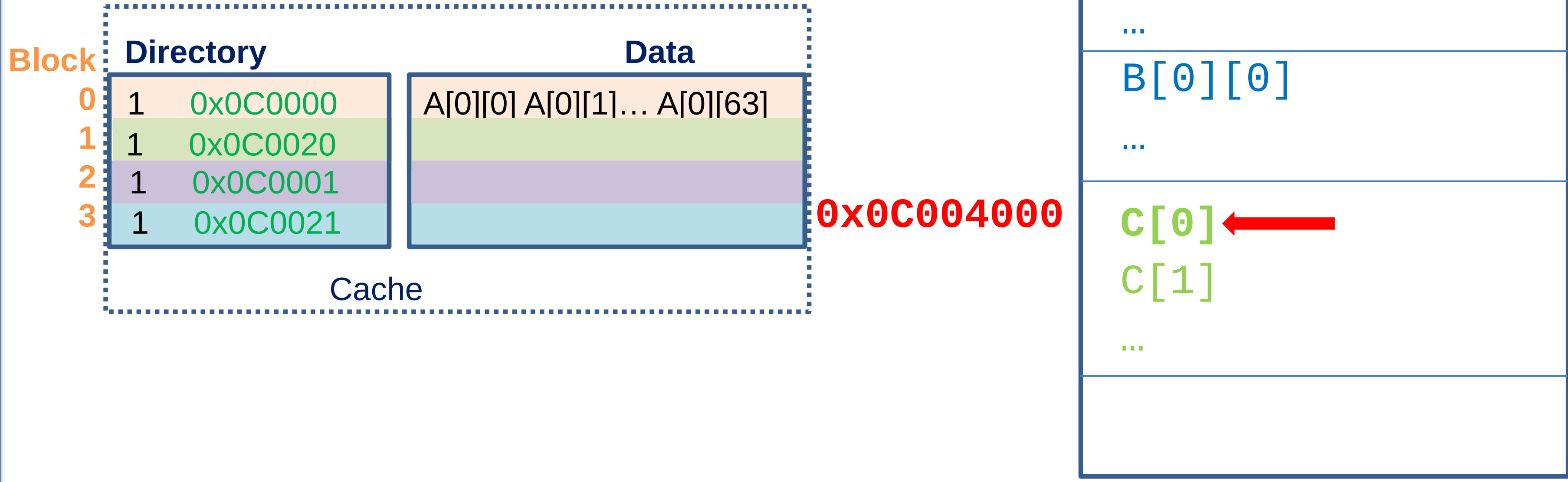
1. Bring block from memory
2. Write in the cache block (and memory, if write-through)



Update policies: no-write allocate

```
for (i=0;i<16;i++) {
   tmp=0;
   for (j=0;j<128;j++)
          tmp+=A[i][j] * B[i][j];
   C[i]=tmp; }
```

Write miss: no-write allocate
1. Write in the memory



Update policies

- Comparison:
  - Write-through:
    - The memory contains always the last value
    - Simple control
  - Write-back:
    - The memory can be outdated, containing a not valid data
    - Much less bandwidth required, can exploit burst accesses
    - Better tolerance to high latency in memory accesses

Replacement policies

- **What happens when the cache is full? Which cache block is replaced?**
- **Replacement space**: the set of cache blocks that can be replaced with the new block
  - **Direct Mapping**: only one cache block can contain the new block, there is no need for replacement policy
  - **Full Associative Mapping**: Any cache block
  - **Set Associative Mapping**: Any cache block in the set that corresponds to the new block
- **Algorithms** (hw implemented):
  - **Random**: a block is randomly selected from the replacement space
  - **FIFO**: the oldest block in the replacement space is selected
  - **LRU** (least recently used): the block from the replacement space accessed longer ago is selected
  - **LFU** (least frequently used): the block from the replacement space with less accesses is selected

"Cache Replacement Policies", A. Jain and C. Lin, Synthesis Lectures on Computer Architecture

Performance

- Perfect memory (ideal)
  - Tcpu = N x CPI x Tc
  - N x CPI = # cycles CPU --> Tcpu = # cycles CPU x Tc
- Real memory
  - Tcpu = (# cycles CPU + # wait cycles) x Tc
  - How many cycles due to memory access?
    - # wait cycles because of memory = # misses x Miss Penalty
    - # misses = # memory references x Miss Rate
    - # memory references = N x AMPI (Average # of Memory accesses Per Instruction)
  - Then:
    - # cycles memory is waiting = N x AMPI x Miss Rate x Miss Penalty
  - Finally:
    - Tcpu = [(N x CPI) + (N x AMPI x Miss Rate x Miss Penalty)] x Tc
    - Tcpu = N x [CPI + (AMPI x Miss Rate x Miss Penalty)] x Tc
- AMAT (Average Memory Access Time):

  AMAT = Hit time + Miss Rate x Miss Penalty

How to improve the performance of the cache?

Tcpu = N x [CPI + (AMPI x **Miss Rate** x **Miss Penalty**)] x **Tc**

AMAT = Hit time + **Miss Rate** x **Miss Penalty**

- Study techniques to:
  - Reduce the miss rate
  - Reduce the penalty of miss
  - Reduce time to hit (hit time)

Types of Cache Misses: The Three C's

1. Compulsory: On the first access to a block; the block must be brought into the cache; also called cold start misses, or first reference misses.
2. Capacity: A block is discarded from cache -because cache cannot contain all blocks needed- and later retrieved (program working set is larger than cache capacity).
3. Conflict: In the case of set associative or direct mapped block placement strategies, conflict misses occur when several blocks are mapped to the same set or block frame; also called collision misses or interference misses.

How to improve the performance of the cache?

Reduce miss rate	Reduce miss penalty	Reduce time hit	Increase bandwidth
Block size	Prioritize readings over writings	Small and simple cache	Non-blocking cache
Associativity	Prioritize critical word	Hide translation latency DV => DF	Multiple bank cache
Cache Memory size	Merge write buffers	Prediction of way	Pipelined cache
Replacement Algorithm	Multilevel Caches	Trace Cache	
Victim cache			
Code optimizations (compiler)			
Pre-fetching (sw and hw)			

2020 Version: contents in red are removed from the course.

Block size

- Increasing the block size decreases initial misses (compulsory) and takes better chance of the **spatial locality.**
  - **It also increases Miss penalty**
- But the miss rate due to capacity misses increases (less #blocks => takes worse chance of the **temporal locality**).
- The right block size depends on the size of the cache.

Most frequent (both Intel and ARM)

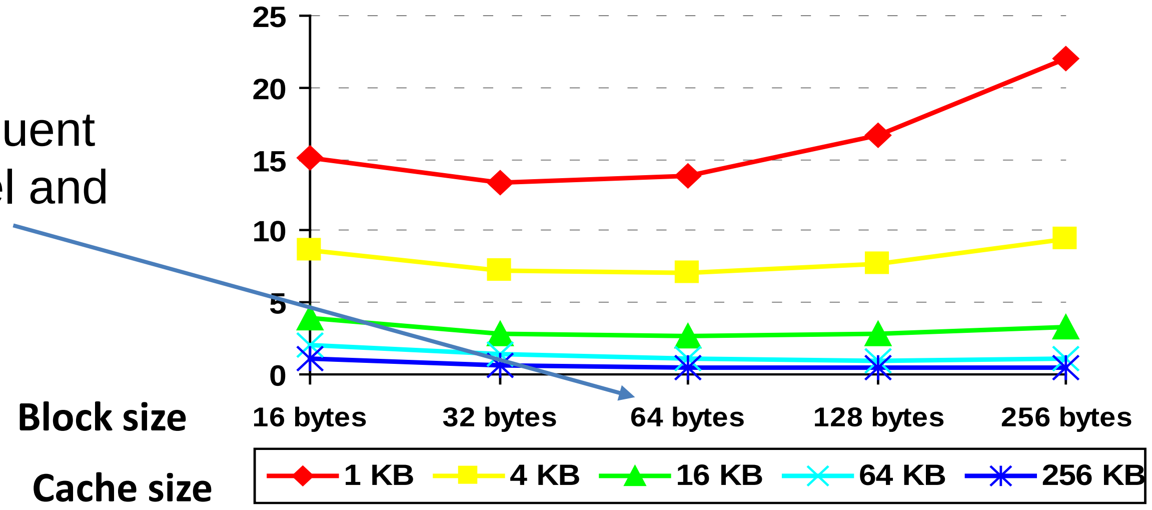


Figure from Hennessy, Patterson, Apendix B

Cache size

- Increasing the cache size decreases capacity misses
- But it also increases hit time (and power and cost)

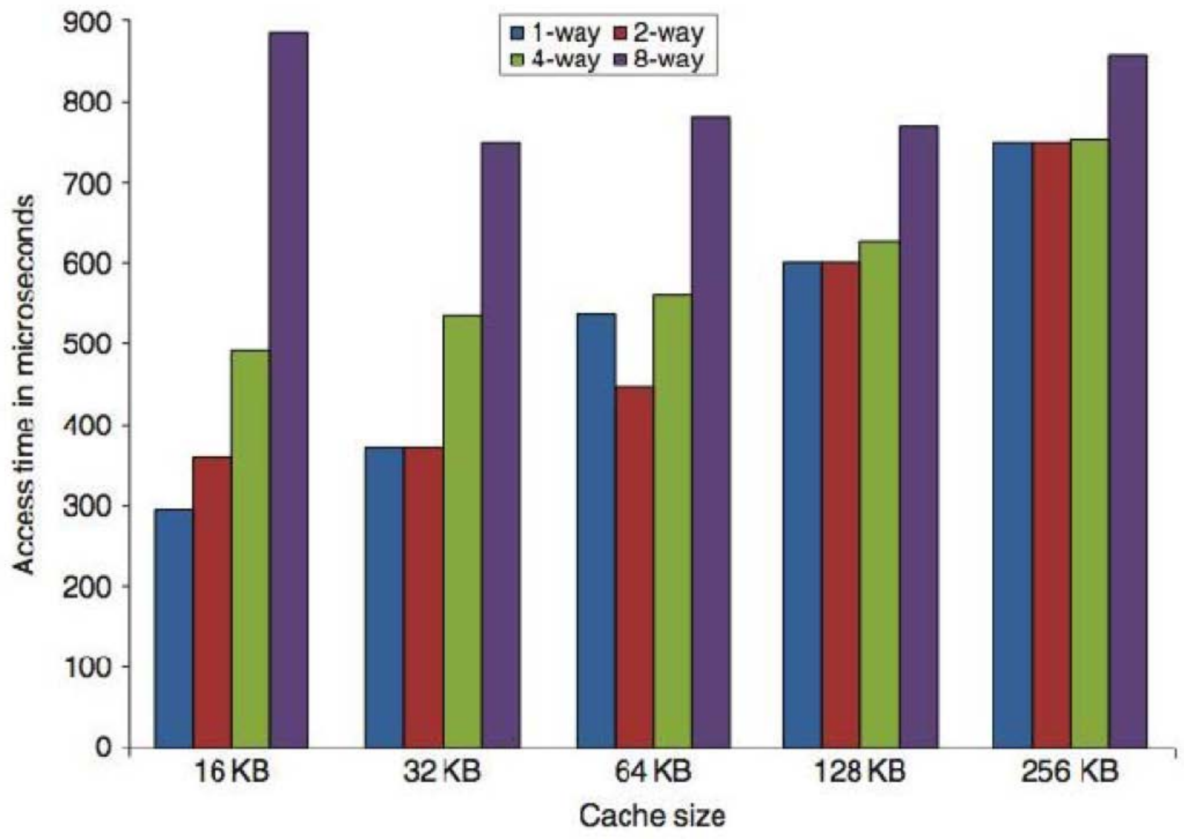
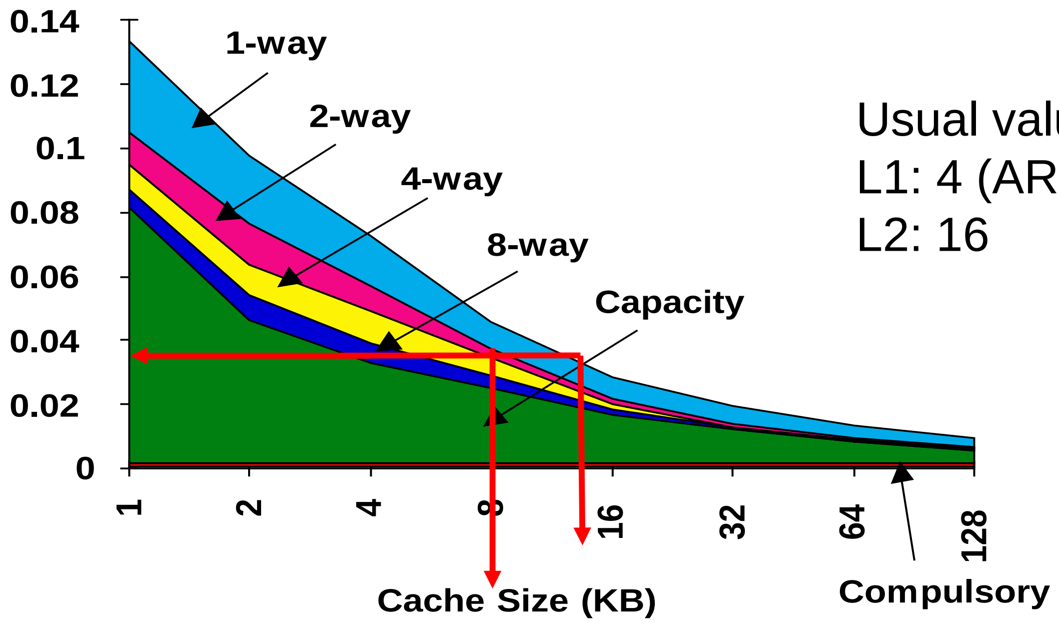


Figure from Hennessy, Patterson, Apendix B

Associativity

- Increasing associativity reduces conflict misses
- But, increasing associativity increases the hardware, the access time (hit time) and energy consumption.

Associativity



Usual values:
L1: 4 (ARM), 8 (Intel)
L2: 16

- The transition from direct-mapping to associative with 2 sets allows to decrease the cache size in one half
- For the same cache size, major improvements are not achieved with more than 8 ways

Figure from Hennessy, Patterson, Apendix B

Replacement algorithms

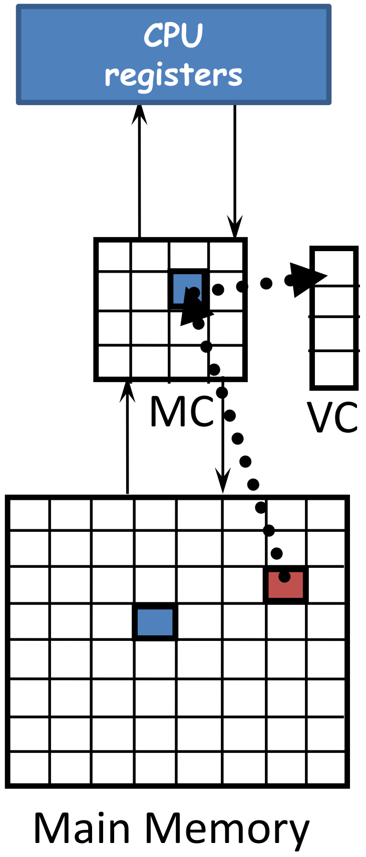
Nº ways	2		4		8	
CM Size	**LRU**	**Random**	**LRU**	**Random**	**LRU**	**Random**
16K	114.1	117.3	111.7	115.1	109.0	111.8
64K	103.4	104.3	102.4	102.3	99.7	100.5
256K	92.2	92.1	92.1	92.1	92.1	92.1

Data misses due to 1000 instructions

- Random replacement policy needs less HW and is faster than LRU, but produces a higher miss rate

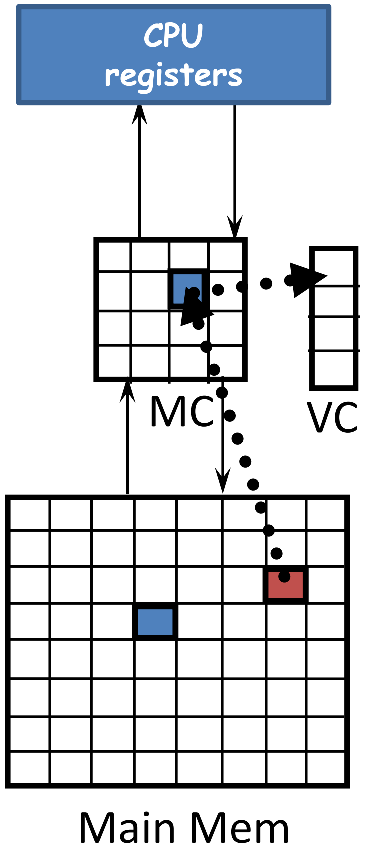
Victim Cache

- It was proposed in 1990.
- IDEA: Include a small associative cache, Victim Cache (VC), before the Main Cache (MC).
- The objective is to obtain a similar miss rate as if the MC was associative.



Victim Cache

- A block cannot be in MC and VC at the same time.
- The VC holds blocks that have been replaced from the MC
  - When a block is evicted from the MC, it is sent to the VC.
  - When a block is accessed at the VC, it is reinserted into the MC.



Code Optimization

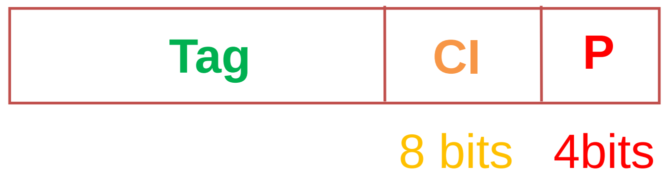
- **Goal: reducing the number of cache misses**
- **How?** Increasing locality of data.
- **Some techniques:**
  - Merge arrays
  - Arrays enlargement
  - Exchange loops
  - Merge Loops
  - Code Blocking

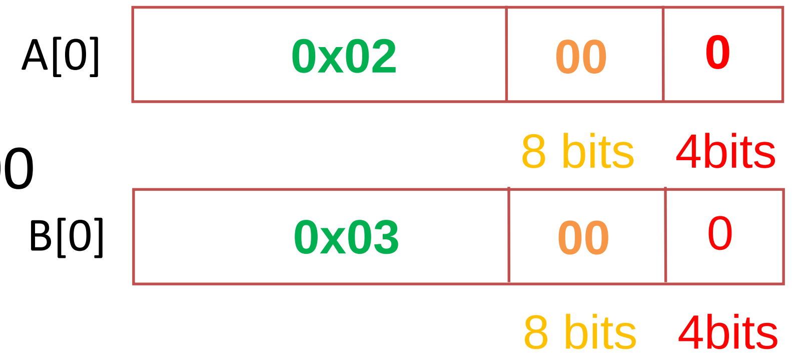
Code blocking is removed in 2020 version

Code Optimization Example

- For all examples:
  - Cache Memory: 4 KB = 2^{12} B
  - 1 block = 16 B = 2^4 B
  - Direct mapping
  - 256 blocks = 2^8 blocks
  - Data:
    - Integers are 32 bits long
    - Data stars at address 0x2000

```
int A [1024]; /* 2^12 B */
int B [1024];
```



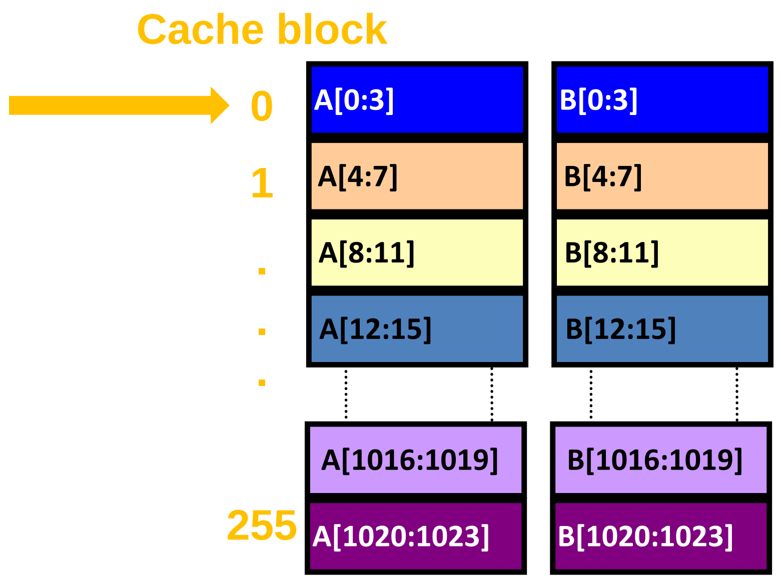


For the same index, A[i] and B[i] will always map to the same cache block.

Code optimization: cache mapping

- Original code:

```
for (i = 0; i < 1024; i = i + 1)
        C = C + (A[i] + B[i]);
```



2x1024 accesses
2x1024 misses
2x256 compulsory
Other misses 2x3x256

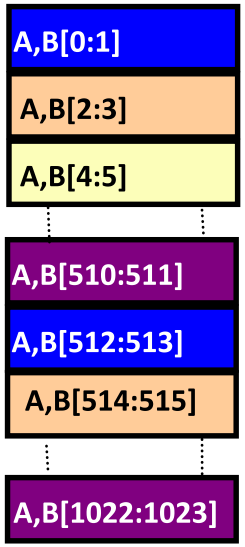
If the CM had 2 ways,
Would the # of misses be reduced?

Code Optimization

- **Merge arrays**: Improves the spatial locality to reduce misses (conflict and capacity)
  - Place the **same positions of different arrays in contiguous memory locations**

```
struct fusion{
        int A;
        int B;
} array[1024];
for (i = 0; i < 1024; i = i + 1)
        C = C + (array[i].A + array[i].B);
```

1024/2 misses
2x256 compulsory

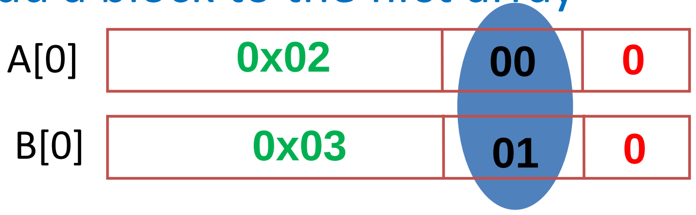


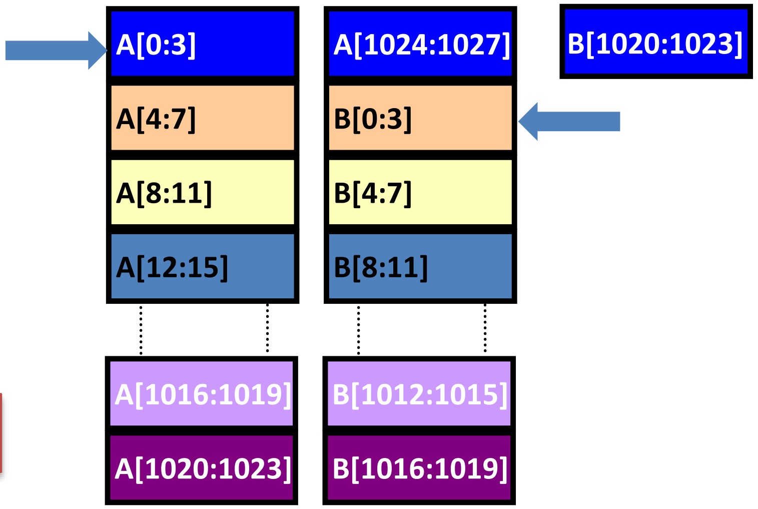
Improvement : x4

Code Optimization

- **Arrays enlargement**: Improves spatial locality to reduce conflict misses
  - To prevent competition for the same frame block on each loop iteration add a block to the first array

```
int A[1028];
int B[1024];

for (i=0; i < 1024; i=i+1)
        C = C + (A[i] + B[i]);
```





1024/2 misses
2x256 compulsory

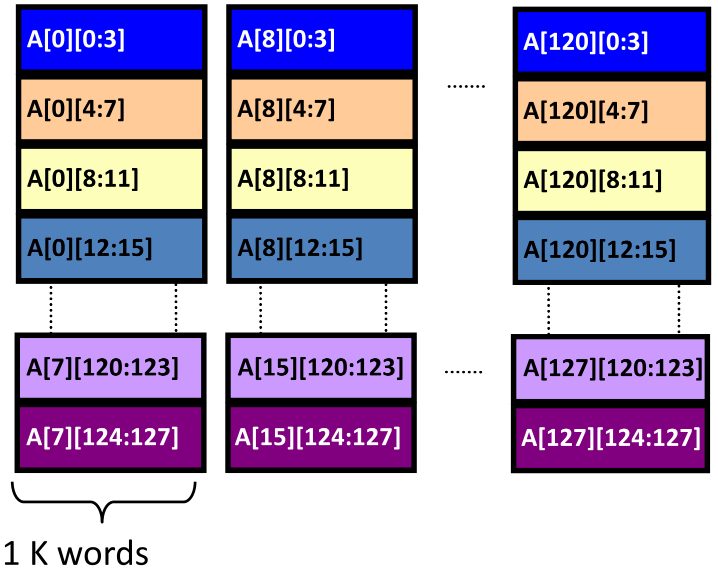
Improvement: x4

Code Optimization

- **Exchange loops**: Improves spatial locality to reduce misses
  - In C, the arrays are stored by rows, then the inner loop should run along the columns.

    int A[128][128];

A has 2^{14} words = 16 K-words => it is **16 times larger than the** Cache Memory



Code Optimization

- **Exchange loops :**
  - Row access

```
for (j=0; j < 128; j=j+1)
        for (i=0; i < 128;i=i+1)
                C = C * A[i][j];
```

128x128 acceses
128x128 misses
16x256 compulsory
Others (12288)

  - Column access

```
for (i=0; i < 128;i=i+1)
        for (j=0; j < 128; j=j+1)
                C = C * A[i][j];
```

128x128/4 misses
16x256 compulsory

Improvement: x4

Code Optimization

- **Merge Loops:** It improves temporal locality to reduce misses
  - Merge loops which use same arrays to re-use the same data found in cache before discarding it

```
for (i=0; i < 128; i=i+1))
        for (j=0; j < 128;j=j+1))
                C = C * A[i][j];

for (i=0; i < 128;i=i+1)
        for (j=0; j < 128;j=j+1)
                D = D + A[i][j];
```

128x128x2 acceses
(128x128/4)x2 misses

```
for (i=0; i < 128; i=i+1))
        for (j=0; j < 128;j=j+1))
                C = C * A[i][j];
                D = D + A[i][j];
```

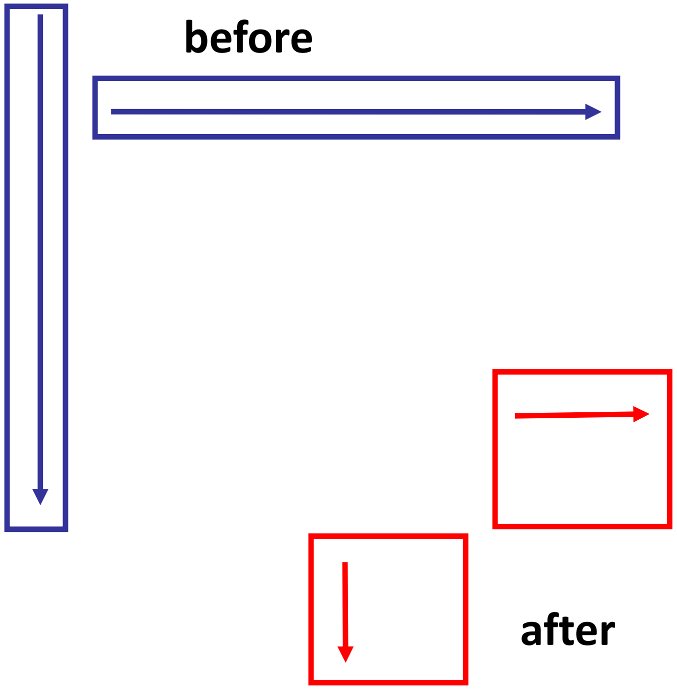
(128x128/4)x2 misses

Improvement: x4

Code Optimization. Blocking

- Improves **temporal locality** for reducing **capacity misses**

```
/* Before */
for (i=0; i < N; i=i+1)
   for (j=0; j < N; j=j+1)
          {r = 0;
           for (k=0; k < N; k=k+1)
                   r = r +
   y[i][k]*z[k][j];
           x[i][j] = r;
          };
```



- ✓ 2 internal loops. For each i:
  - Read all NxN elements in **z**
  - Read N elements of row i in **y**
  - Write N elements of row i in **x**
- ✓ Capacity misses are dependent on N and cache size
- ✓ IDEA: Use BxB matrices that fit in the cache

Code blocking is removed in 2020 version

Code Optimization. Blocking

```
/* After */
for (jj=0;jj < N; jj=jj+B)
for (kk=0;kk < N; kk=kk+B)
for (i=0; i < N; i=i+1)
  for (j=jj; j < min(jj+B-1,N); j=j+1)
      {r = 0;
       for (k=kk; k < min(kk+B-1,N); k=k+1)
             r = r + y[i][k]*z[k][j];
       x[i][j] = x[i][j]+r;
      };
```

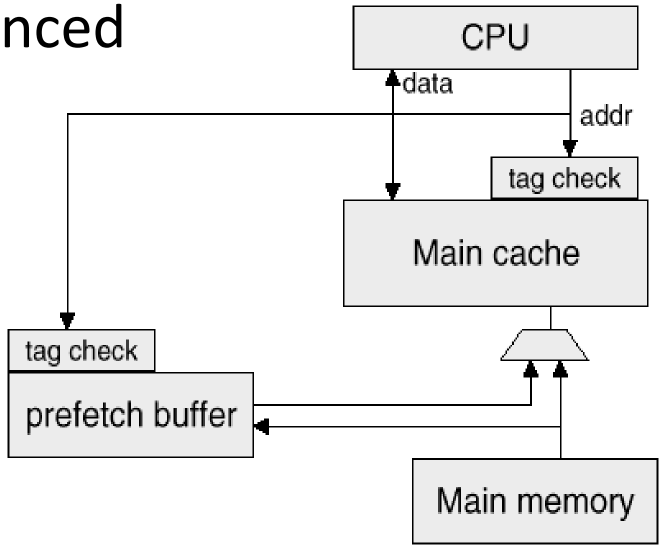
- ✓ B *(Blocking Factor)*

Pre-fetching

- Cache anticipates misses searching in advance before the processor demands a data or instruction that would cause a miss
  - This technique reduces the miss rate
  - This type of cache is only effective with a suitable bandwidth, since it will require more block transfers
  - This technique is most effective when a buffer for prefetched blocks is available, otherwise the prefetched blocks (which may not be used) may be discarding useful blocks from the MC

Hardware Pre-fetching

- “Next-line prefetching”: CPU fetches two blocks on misses (**the referenced block and the next one**).
  - The requested block is loaded in the cache
  - The pre-fetched block is loaded in a buffer ("prefetch buffer" or "buffer stream"). It is moved to cache as soon as it is referenced



Widely used as instruction prefetcher and also used as data prefetcher.

“A primer on hardware prefetching”, B. Falsafi and T.F. Wenisch, Synthesis Lectures on Computer Architecture

Sofware Pre-fetching

- Special instructions for Pre-fetching generated by the compiler
  - The efficiency depends on the compiler and the type of program.
    - It works well with loops and simple array access patterns. Computing applications
    - It does not provide good results on integer applications
  - Pre-fetching cache loading in the cache (MIPS IV, PowerPC, SPARC v. 9) or in a register (HP-PA)
  - Overhead for new instructions. More fetching. More memory occupation

Sofware Pre-fetching Example

✓If the miss penalty is equal to the execution time of 7 iterations.

✓Idea: Split the loop in two parts

```
/*  i=0 (prefetch a and b) */
for (j:=0; j<100; j:=j+1) {
        prefetch (b[j+7][0]);    /* b[j][0] for 7 iteracions later */
        prefetch (a[0][j+7]);    /* a[0][j] for 7 iteracions later*/
        a[0][j] := b[j][0] * b[j+1][0] ;  }


/* i=1,2 (prefetch a, since b is already in the cache) */
for (i:=1; i<3; i:=i+1)
        for (j:=0; j<100; j:=j+1) {
                prefetch (a[i][j+7]); /* a[i][j] for 7 iteracions later */
                a[i][j] := b[j][0] * b[j+1][0] ;  }
```

✓Total misses: $3 * \lceil 7/2 \rceil + 7 = 19$
✓Extra instructions (prefetch): 100*2 + 200*1 = 400
✓Misses avoided: 251 -19 = 232

How to improve the performance of the cache?

Reduce miss rate	Reduce miss penalty	Reduce time hit	Increase bandwidth
Block size	Prioritize readings over writings	Small and simple cache	Non-blocking cache
Associativity	Prioritize critical word	Hide translation latency DV => DF	Multiple bank cache
Cache Memory size	Merge write buffers	Prediction of way	Pipelined cache
Replacement Algorithm	Multilevel Caches	Trace Cache	
Victim cache			
Code optimizations (compiler)			
Pre-fetching (sw and hw)			

Multi level Caches

- Increasing the size of the CM:
  - Reduces the miss rate
  - But, increases the hit time
  - And also increases the cost of the processor
- A **quick** and inexpensive cache memory, has to be **small**
- Different cache levels can reduce the **miss penalty**
  - The cache closer to the CPU is small (L1) and the next level (L2) is bigger.
  - It is different to transfer from MM to cache, than from L2 to L1
  - L2 allows pre-fetching policies with fewer drawbacks

Multi level Caches

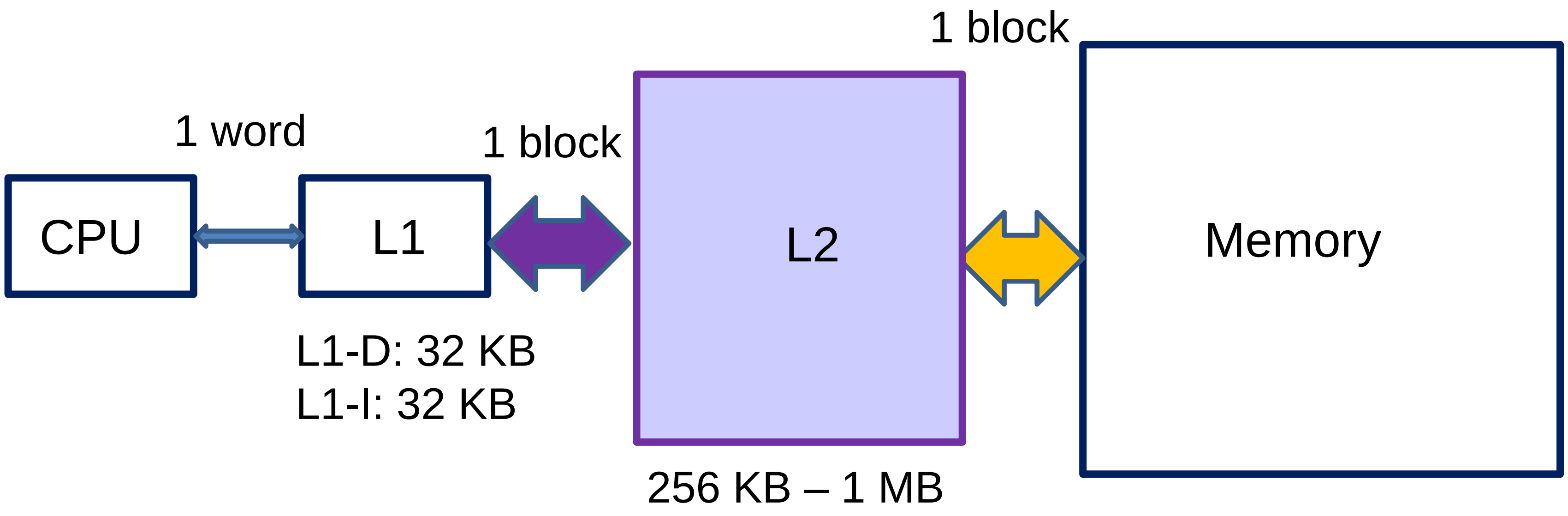
- **Multi level Cache (L1, L2, …)**
  - Average access time to memory (AMAT): One level

    AMAT = Hit time + Miss Rate x Miss Penalty
  - Average access time to memory: Two levels

    $\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$

    $\text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}$

    $\Rightarrow \text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times [\text{Hit Time}_{L2} + (\text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2})]$



Reduction of the miss penalty

- Prioritize reads over writes
  - A read miss can stop program execution; a write miss can be hidden
    - Write Buffer (fast). Store in the Buffer the words to be updated in memory and continue the execution.
    - Transfer from the buffer to memory is performed in parallel with the execution of the program
  - PROBLEM: read a word that is still in the buffer.

```
SW 512(R0), R3;        M[512] <- R3       (block 0)
LW R1, 1024(R0);       R1 <- M[1024]      (block 0)
LW R2, 512(R0);        R2 <- M[512]       (read miss block 0)
```

- On a read miss, check that the address is not the same as one of the pending writes

Reduction of the miss penalty

- Prioritize critical words
  - When the requested word is loaded into cache, it is also sent to the processor, without waiting for loading the entire block
  - Two different implementations:
    - **Early restart**: the requested word is written to the cache and then it is sent to the processor without waiting for the rest of the block.
    - **Critical word first**: the requested word is first sent to the cache and the processor and, then the rest of the block is written in the cache in the following clock cycles.
  - PROBLEM: high probability that the next access corresponds to the following sequential word.

How to improve the performance of the cache?

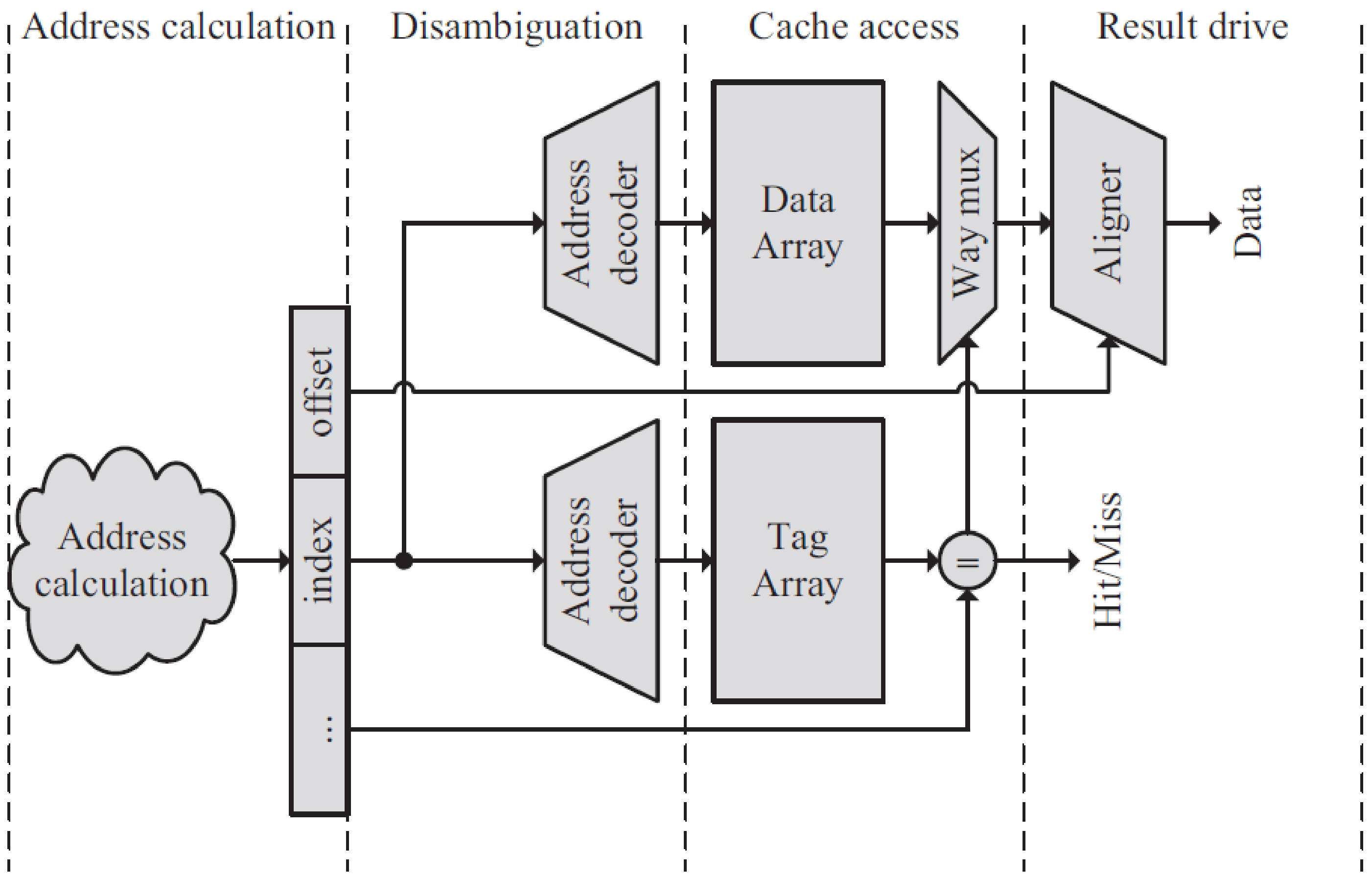
Reduce miss rate	Reduce miss penalty	Reduce time hit	Increase bandwidth
Block size	Prioritize readings over writings	Small and simple cache	Non-blocking cache
Associativity	Prioritize critical word	Hide translation latency DV => DF	Multiple bank cache
Cache Memory size	Merge write buffers	Prediction of way	Pipelined cache
Replacement Algorithm	Multilevel Caches	Trace Cache	
Victim cache			
Code optimizations (compiler)			
Pre-fetching (sw and hw)			

Pipelined Cache

- Pipelined Cache
  - Pipelining the accesses to the cache increases the bandwidth.
  - Problem: May increase latency
  - More problems: Larger penalty on wrong predicted jumps/branches
  - Examples: Number of stages of access to the cache on different processors

• Pentium	1 stage
• Pentium Pro to Pentium III	2 stages
• Pentium 4	4 stages

Pipelined Cache



Processor Microarchitecture. An implementation Perspective. A. González et al. Synthesis Lectures on Computer Architecture

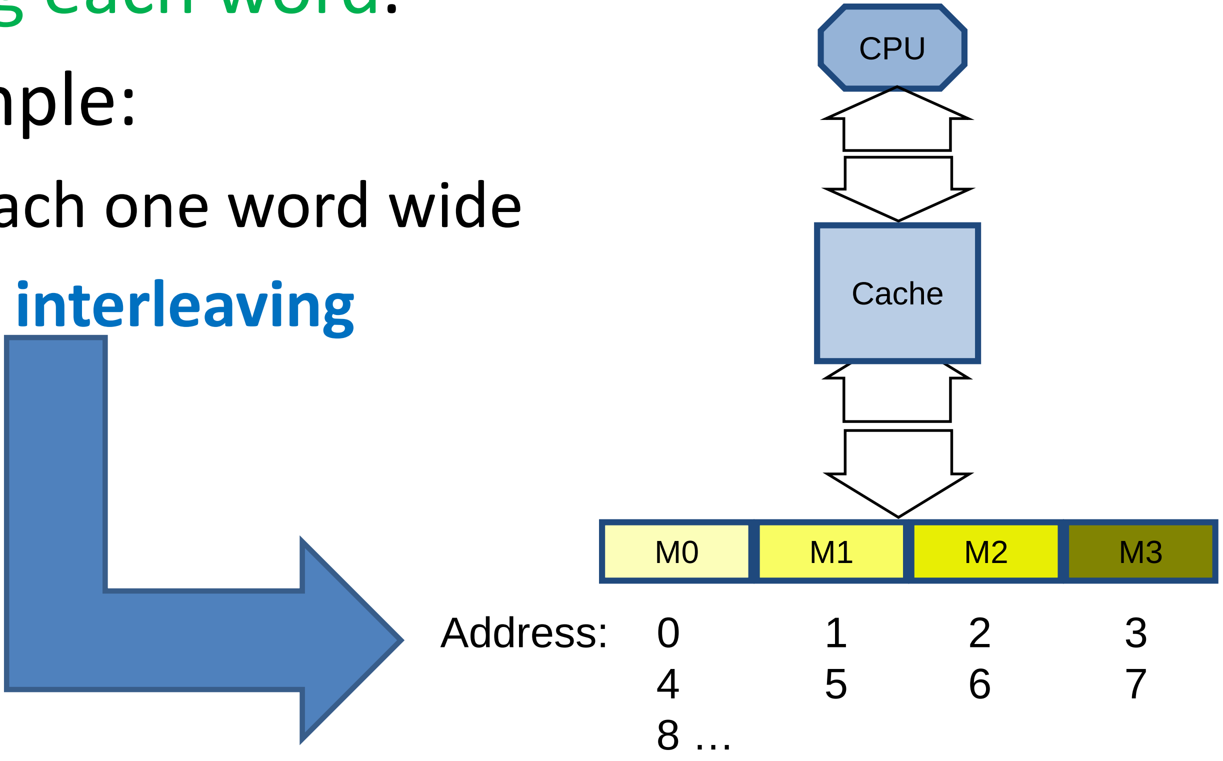
Interleaved memory (1)

- Goal: reducing time to transfer a memory block to cache
  - Buses from the CPU to the cache and from the cache to RAM are all one word wide
- Let’s assume the following three steps are taken when a cache needs to load data from memory:
  1. It takes 1 cycle to send an address to the RAM.
  2. There is a 15-cycle latency for each RAM access.
  3. It takes 1 cycle to return data from the RAM.

Time to transfer 1 word: 1 + 15 + 1 = 17 clock cycles
Time to transfer a 4-word block: 4 x (1 + 15 + 1) = 68 clock cycles

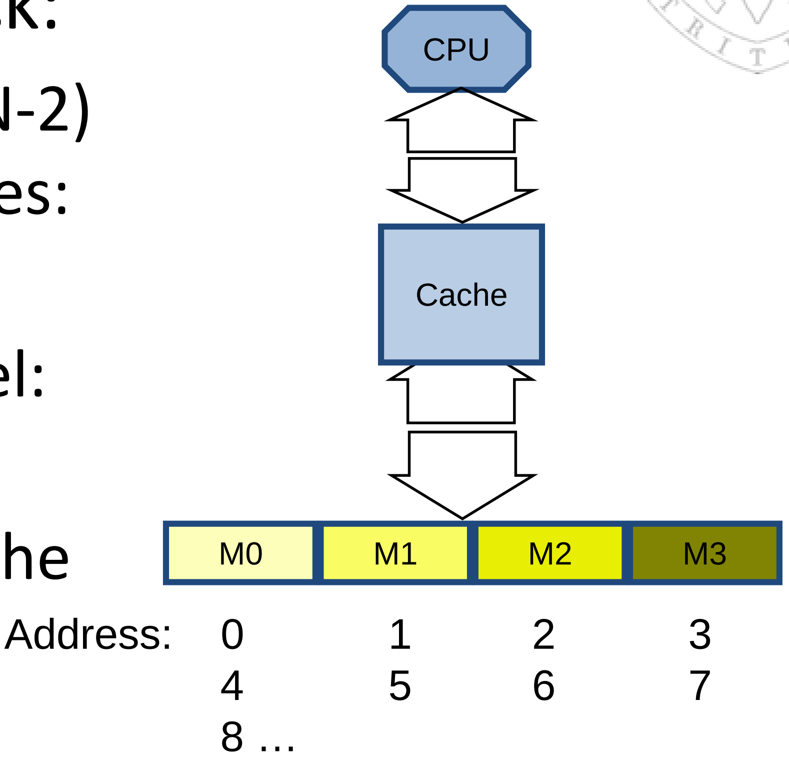
Interleaved memory (2)

- Another approach is to interleave the memory, or split it into "banks" that can be accessed individually.
  - The main benefit is overlapping the latencies of accessing each word.
  - For example:
  - – 4 banks, each one word wide
  - – **Low order interleaving**



Low order interleaved memory (3)

- Steps to read a memory block:
  - The high bits of the address (N-2) are sent to all memory modules:
    1 cycle
  - All modules are read in parallel:
    15 cycles
  - Each module sends a data to the bus (sequentially):
    4x1 cycles

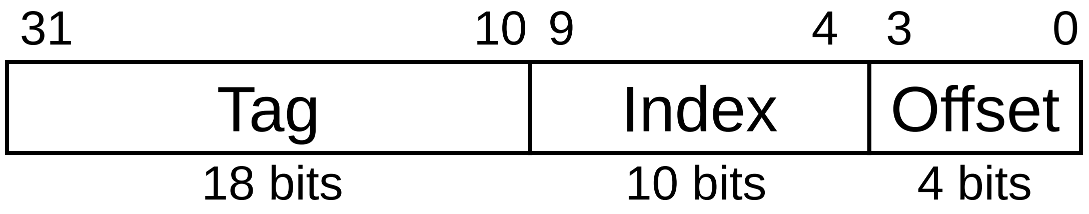


Time to transfer a word = 1 + 15 +1 cycles
Time to transfer a 4-word block = 1 + 15 + (4 x 1) = 20 cycles

It works for sequential accesses (block replacement)

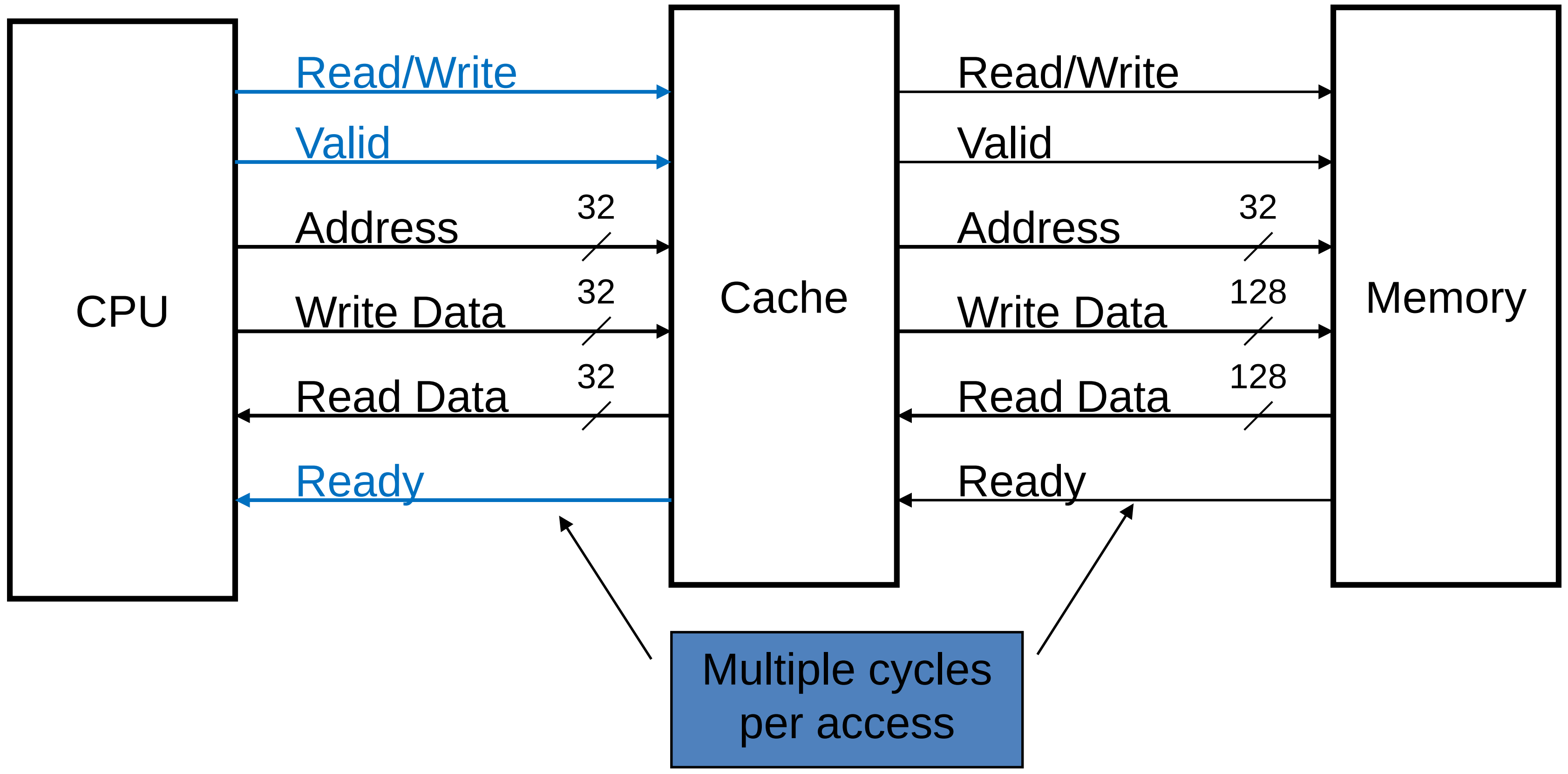
Design of a simple cache controller

- Example cache characteristics:
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache (CPU waits until access is complete)



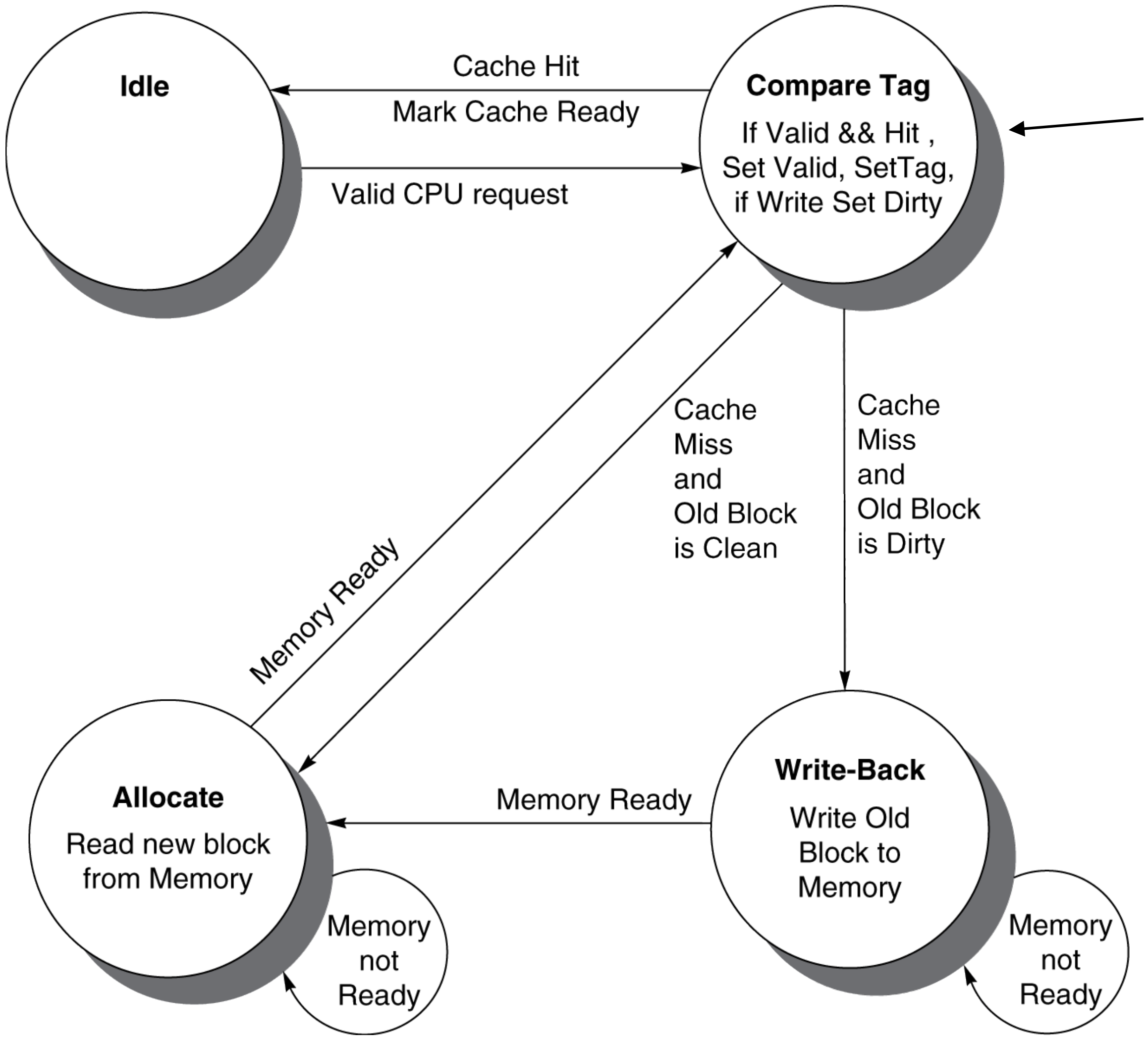
Patterson, D . Hennessy, J. L. “Computer Organization and Design. The hardware/software interface”. Section 5.9

Signals for the cache controller



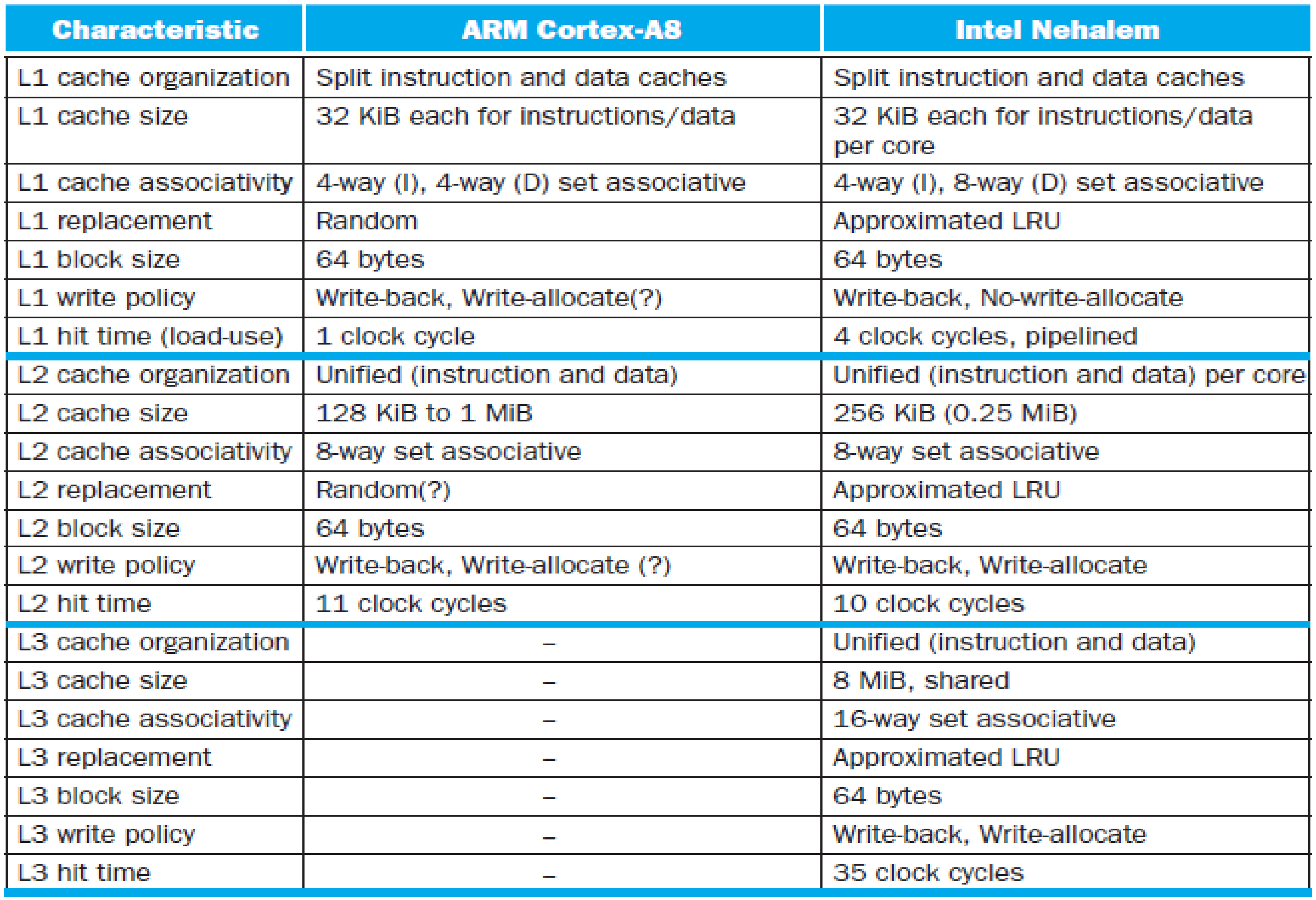
Valid = 1 means there is a R/W request

FSM for the cache controller



Could partition into separate states to reduce clock cycle time

Some real world Multilevel On-Chip Caches



Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

Patterson figure 5.44

IBM Power 5-6-7-8-9 Caches

	Power 5 130nm,	Power 6 65nm,	Power 7 45nm,	Power 8 22nm,	Power 9 14nm,
L1 I L1 D	64KB, direct, 128 bytes, read only 32KB, 2 ways, 128 bytes, write-through	64KB, 4 ways, 128 bytes, read only 64KB, 8 ways, 128 bytes, write-through	32KB, 4 ways, 128 bytes, read only 32KB, 8 ways, 128 bytes, write-through	32KB, 4 ways, 128 bytes, read only 64KB, 8 ways, 128 bytes, write-through	32KB, 4 ways, 128 bytes, read only 64KB, 8 ways, 128 bytes, write-through
L2	Shared, 1,92MB, 10 ways, 128 bytes, pseudoLRU, copy-back	Shared, 4MB, 8 ways, 128 bytes, pseudoLRU, copy-back	Core private, 256KB, 8 ways, 128 bytes, pseudoLRU, copy-back	Core private, 512KB, 8 ways, 128 bytes, pseudoLRU, copy-back	Core private, 512KB, 8 ways, 128 bytes, pseudoLRU, copy-back
L3	Off-chip Shared, Victims 36MB, 12 ways, 256 bytes, pseudoLRU	Off-Chip Shared, Victims 32MB, 16 ways, 128 bytes, pseudoLRU	On-Chip Shared, Victims 32MB, data migration, 4MB-banks	On-Chip Shared, Victims 96MB, data migration, 8MB-banks	On-Chip Shared Victims 120MB, data migration, 10MB-banks