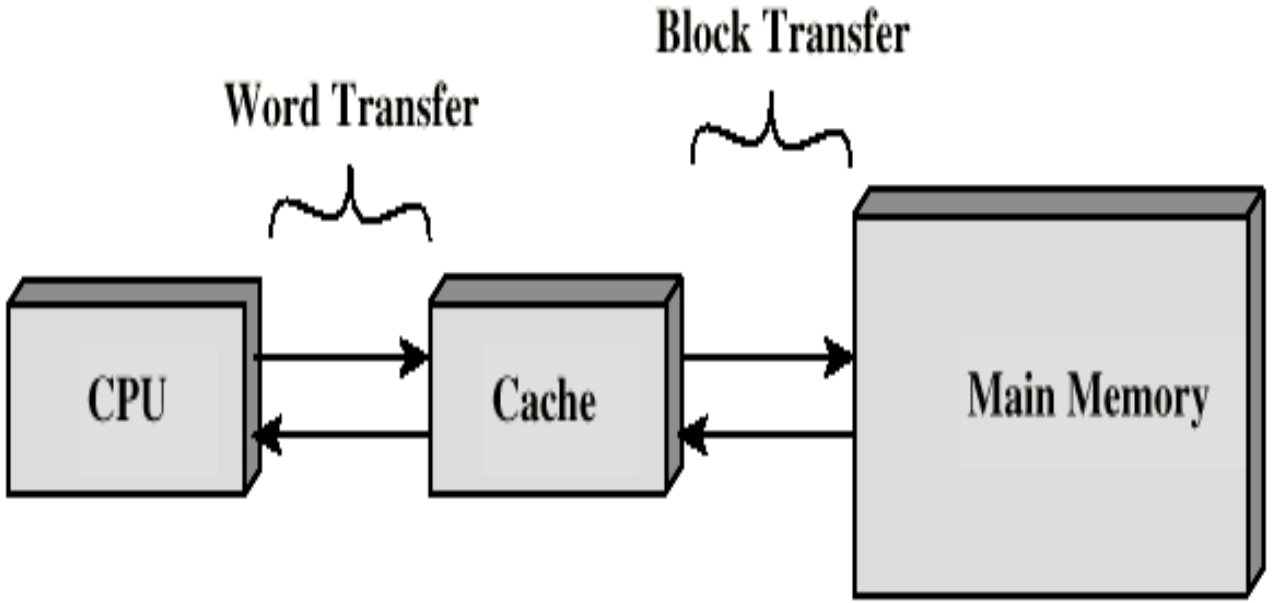
CACHE MEMORY

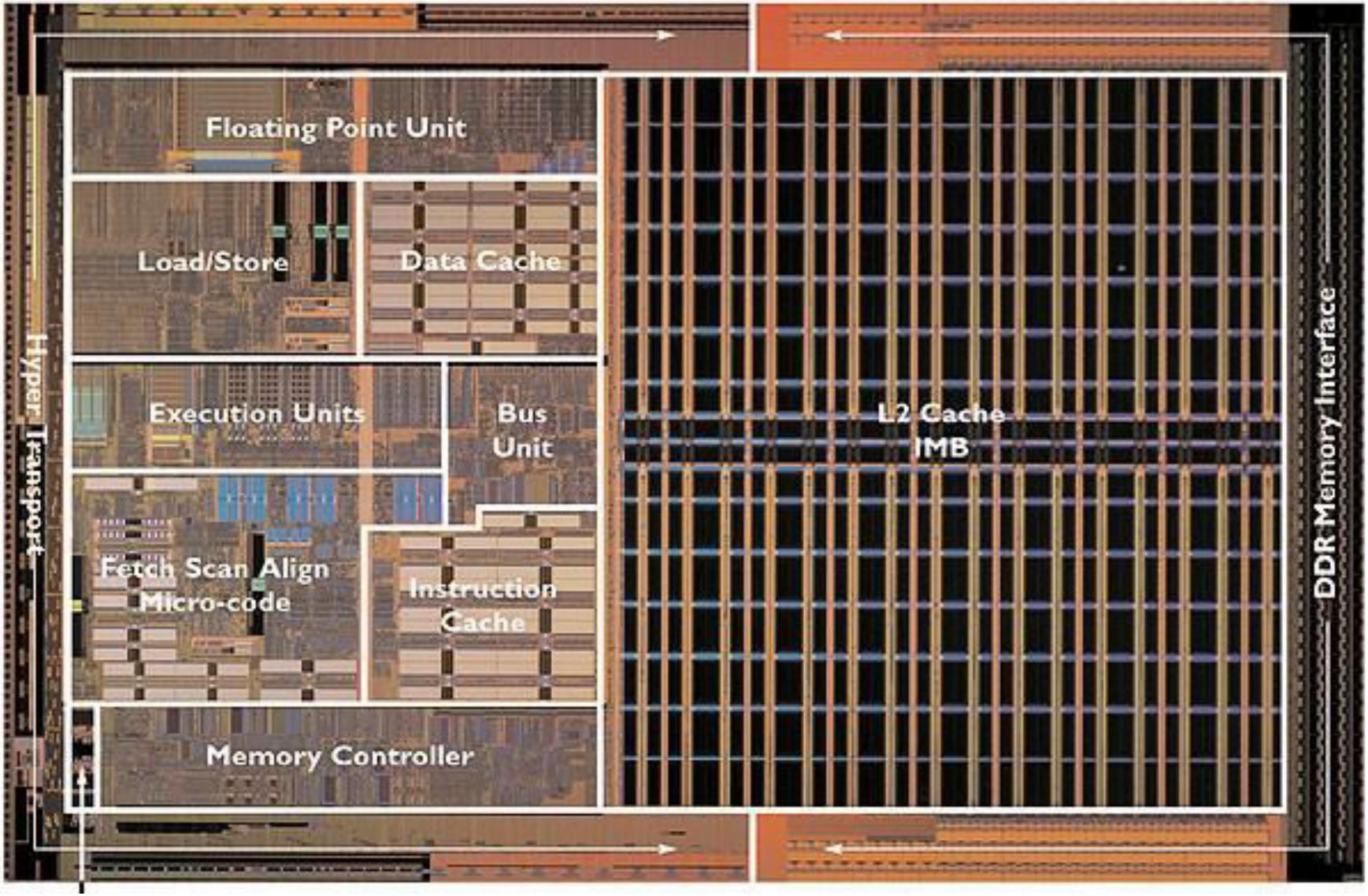
JASMIN T JOSE
ASST. PROFESSOR, SCSE
VIT UNIVERSITY, VELLORE

Cache Memory

A small amount of high speed memory between the main memory and the processor.
May be located on processor chip or separate module

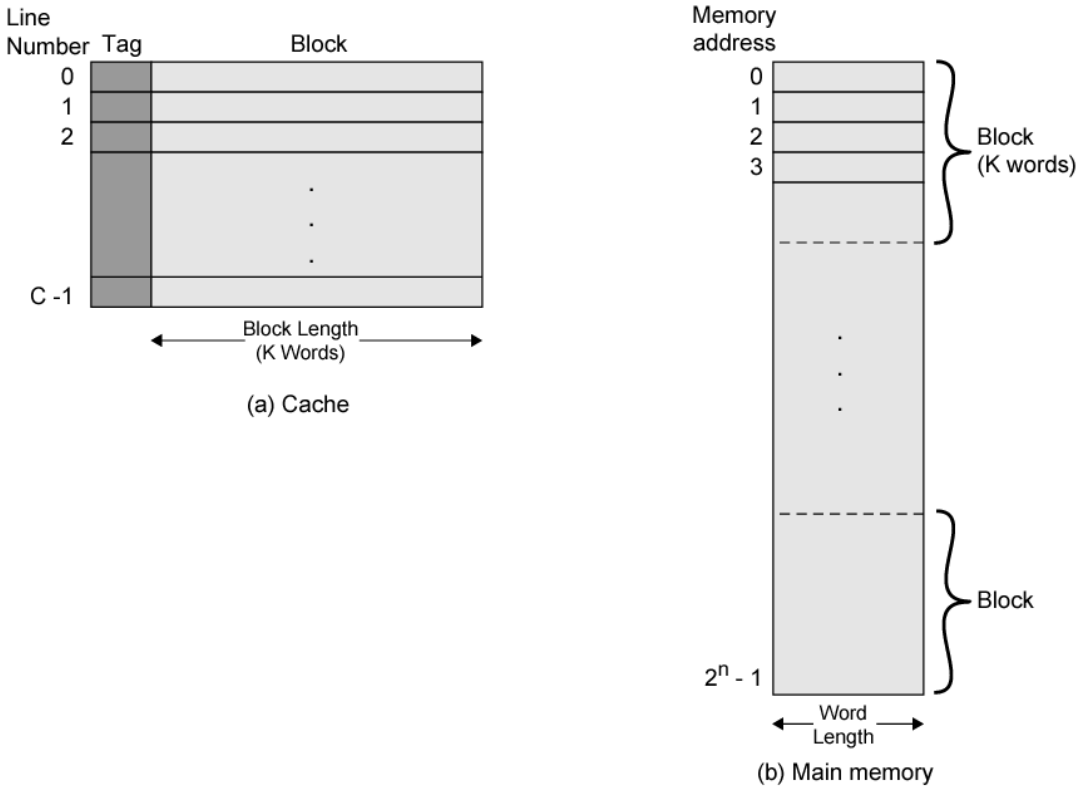


Cache Memory



Cache memory Vs Main memory structure

- Cache is partitioned into **lines** (also called **blocks**).
- Each line has 4-64 bytes in it. During data transfer, a whole line is read or written.
- Cache includes **tags** to identify which block of main memory is in each cache line.



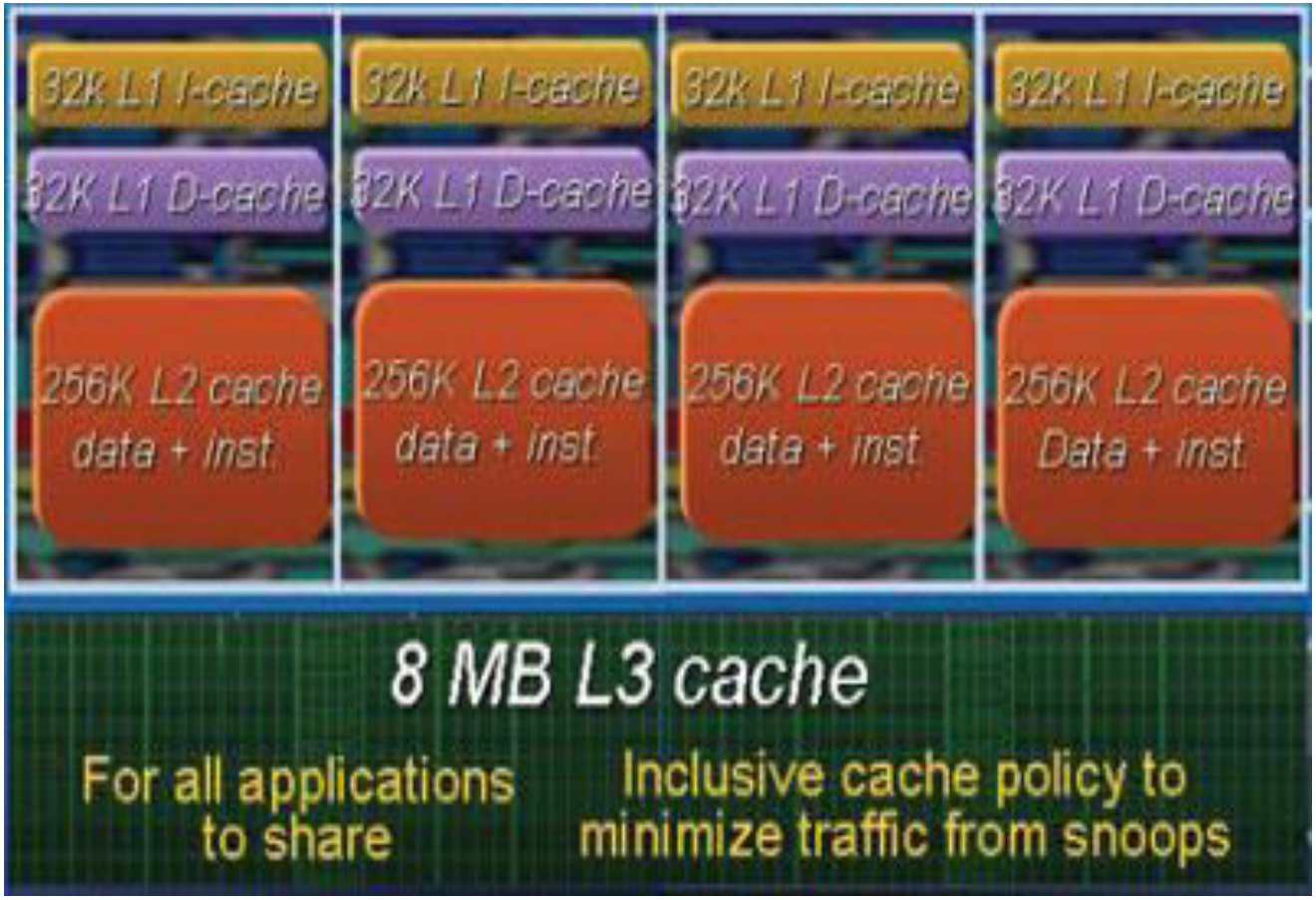
Cache organization

- Split cache
  - Separate caches for instructions and data
  - I-cache (Instruction) – mostly accessed sequentially
  - D-cache (data) – mostly random access
- Unified cache
  - Same cache for instruction and data
- Higher hit rate for unified cache as it balances between instruction and data
- Split caches eliminate contention for cache between the instruction processor and the execution unit – used for pipelining processes

Multilevel caches

- The penalty for a cache miss is the extra time that it takes to obtain the requested item from central memory.
- One way in which this penalty can be reduced is to provide another cache, the secondary cache, which is accessed in response to a miss in the primary cache.
- The primary cache is referred to as the L1 (level 1) cache and the secondary cache is called the L2 (level 2) cache.
- Most high-performance microprocessors include an L2 and L3 cache which is often located off-chip, whereas the L1 cache is located on the same chip as the CPU.
- With a two-level cache, central memory has to be accessed only if a miss occurs in both caches.

Nehalem's micro-architecture showing different cache memory levels



Small or slow

- Unfortunately there is a tradeoff between speed, cost and capacity.

Storage	Speed	Cost	Capacity
Static RAM	Fastest	Expensive	Smallest
Dynamic RAM	Slow	Cheap	Large
Hard disks	Slowest	Cheapest	Largest

- Fast memory is too expensive for most people to buy a lot of.
- Here are rough estimates of some current storage parameters.

Storage	Delay	Cost/MB	Capacity
Static RAM	1-10 cycles	~$5	128KB-2MB
Dynamic RAM	100-200 cycles	~$0.10	128MB-4GB
Hard disks	10,000,000 cycles	~$0.0005	20GB-400GB

The principle of locality

- It's usually difficult or impossible to figure out what data will be "most frequently accessed" before a program actually runs, which makes it hard to know what to store into the small, precious cache memory.
- But in practice, 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, there is a good chance that it will 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.

Temporal locality in programs

- The principle of temporal locality says that if a program accesses one memory address, there is a good chance that it will access the same address again.
- Loops are excellent examples of temporal locality in programs.
  - The loop body will be executed many times.
  - The computer will need to access those same few locations of the instruction memory repeatedly.
- For example:

```
Loop:   lw     $t0, 0($s1)
        add    $t0, $t0, $s2
        sw     $t0, 0($s1)
        addi   $s1, $s1, -4
        bne    $s1, $0, Loop
```

  - Each instruction will be fetched over and over again, once on every loop iteration.

Spatial locality in programs

- 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.

```
sub   $sp, $sp, 16
sw    $ra, 0($sp)
sw    $s0, 4($sp)
sw    $a0, 8($sp)
sw    $a1, 12($sp)
```

- Nearly every program exhibits spatial locality, because instructions are usually executed in sequence—if we execute an instruction at memory location *i*, then we will probably also execute the next instruction, at memory location *i+1*.
- Code fragments such as loops exhibit *both* temporal and spatial locality.

Parameters of cache memory

i) Cache hit

Data found in cache.

Results in data transfer at maximum speed.

ii) Cache miss

Data not found in cache. Processor loads data from memory and copies into cache (miss penalty).

iii) Hit ratio

Ratio of number of hits to total number of references =>
number of hits/(number of hits + number of Miss)

iv) Miss penalty

Additional cycles required to serve the miss

Time required for the cache miss depends on both the latency and bandwidth

v) Latency – time to retrieve the first word of the block

vi) Bandwidth – time to retrieve the rest of this block

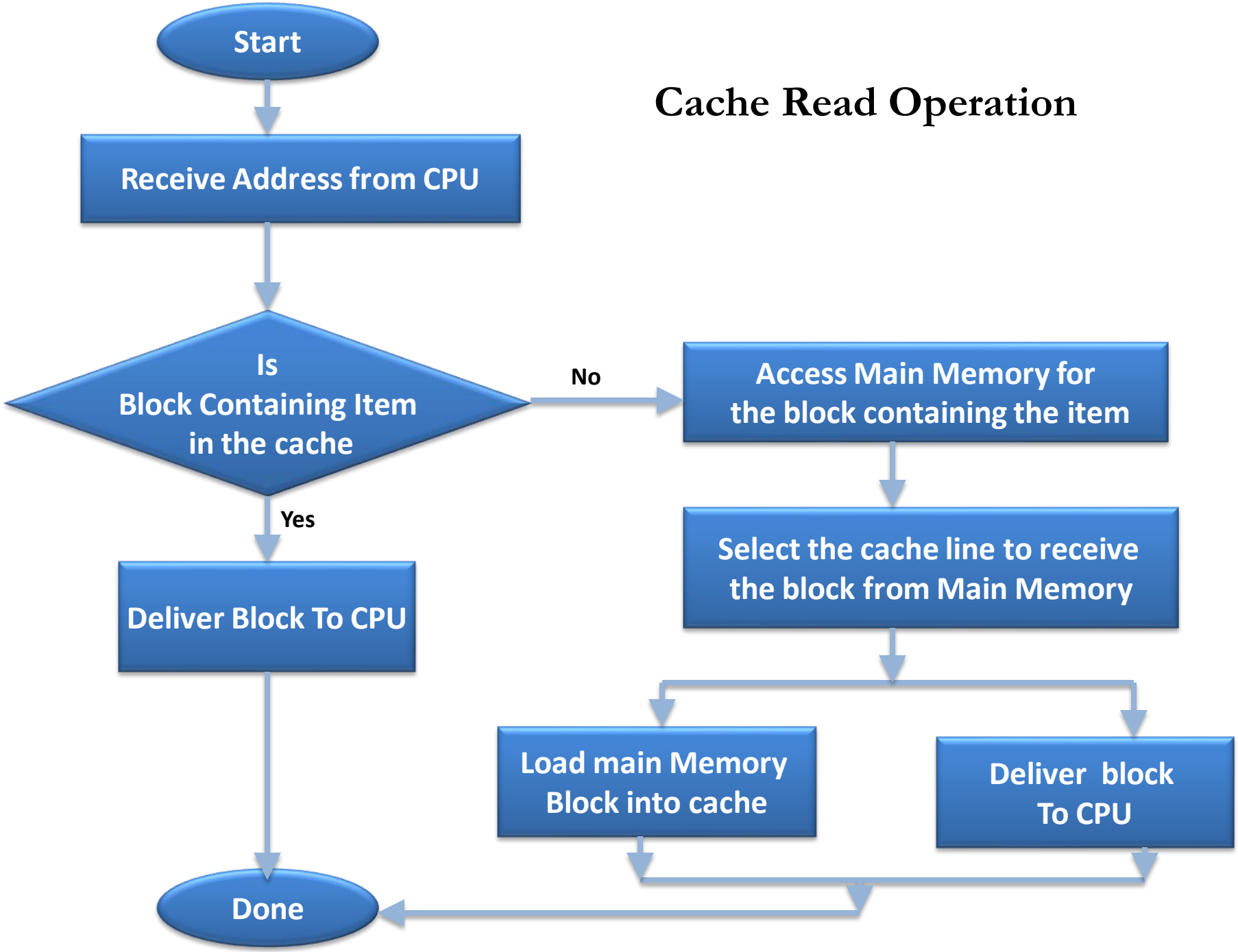
Sources of Cache Misses (Three C's)

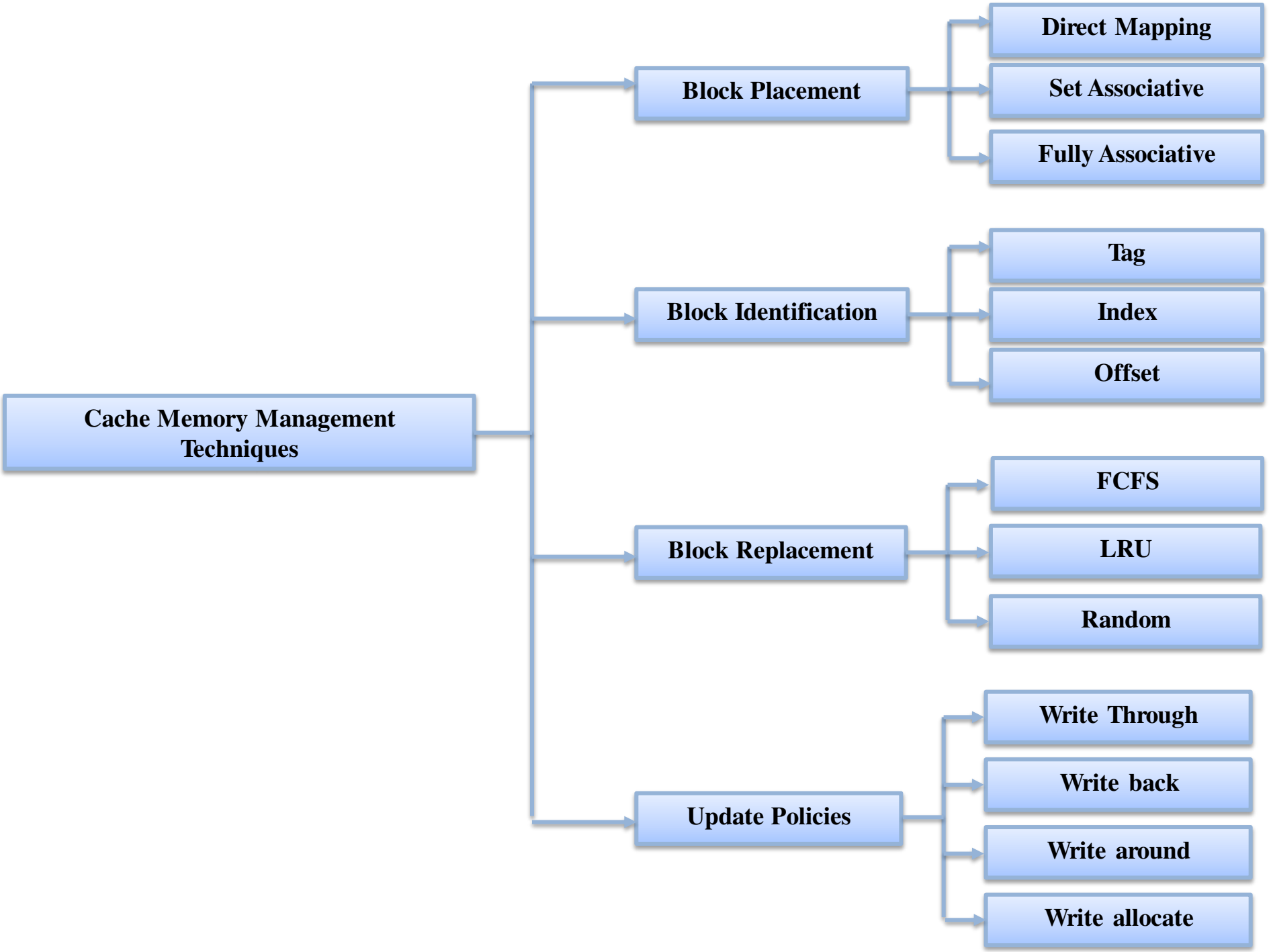
Compulsory Misses: These are misses that are caused by the cache being empty initially or by the first reference to a location in memory . Sometimes referred to as **Cold misses**.

Capacity Misses : If the cache cannot contain all the blocks needed during the execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.

Conflict Misses: If the cache mapping is such that multiple blocks are mapped to the same cache entry. Common in set associative or direct mapped block placement, where a block can be discarded and later retrieved if too many blocks map to its set. Also called collision or interference misses.

Cache Read Operation




Cache Memory Management Techniques
Block Placement
Direct Mapping
Set Associative
Fully Associative
Block Identification
Tag
Index
Offset
Block Replacement
FCFS
LRU
Random
Update Policies
Write Through
Write back
Write around
Write allocate

Example

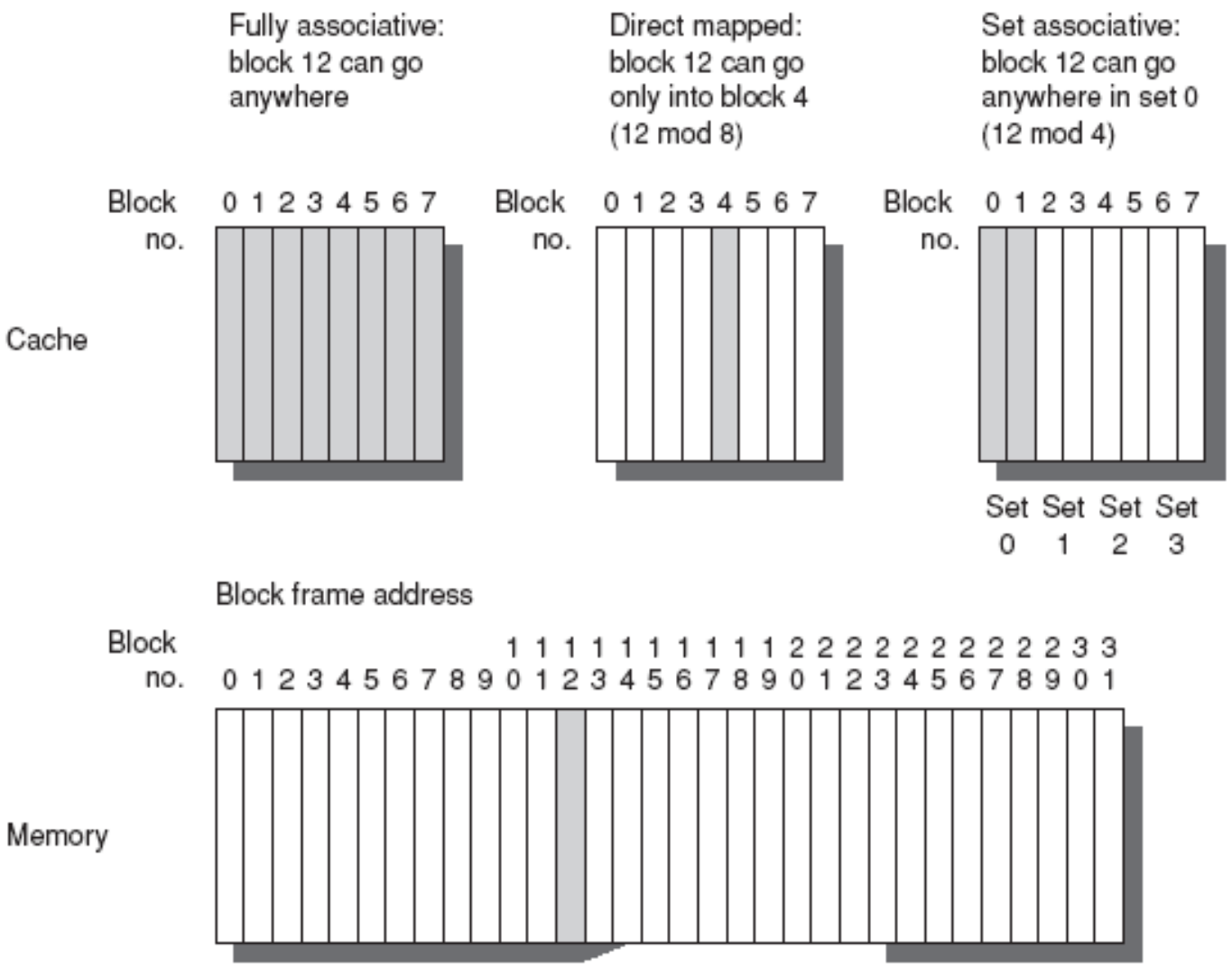
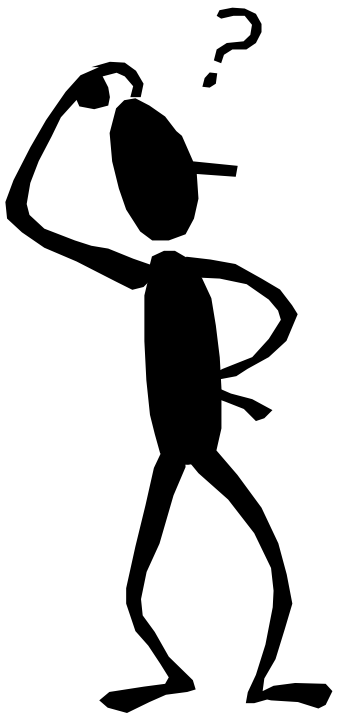


Figure C.2 This example cache has eight block frames and memory has 32 blocks.

Four important questions

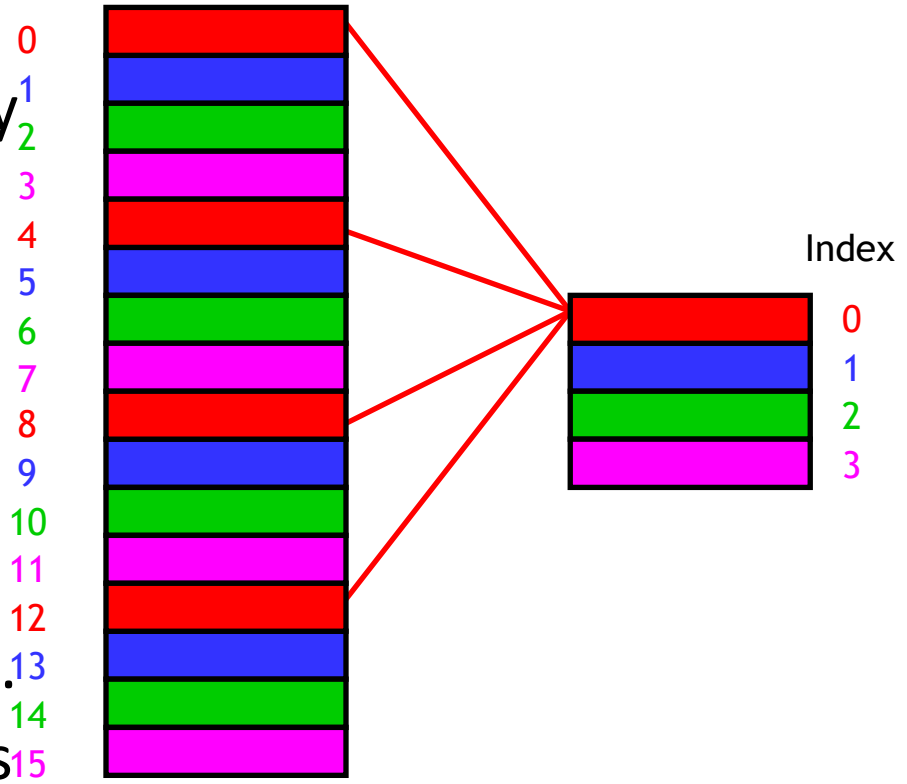


1. When we copy a block of data from main memory to the cache, where exactly should we put it?
2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?
3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we'd have to replace one of the existing blocks in the cache... which one?
4. How can *write* operations be handled by the memory system?

- Questions 1 and 2 are related—we have to know where the data is placed if we ever hope to find it again later!

Where should we put data in the cache?

- A direct-mapped cache is the simplest approach: each main memory address maps to exactly one cache block.
- For example, on the right is a 16-byte main memory and a 4-byte cache (four 1-byte blocks).
- Memory locations 0, 4, 8 and 12 all map to cache block 0.
- Addresses 1, 5, 9 and 13 map to cache block 1, etc.
- How can we compute this mapping?



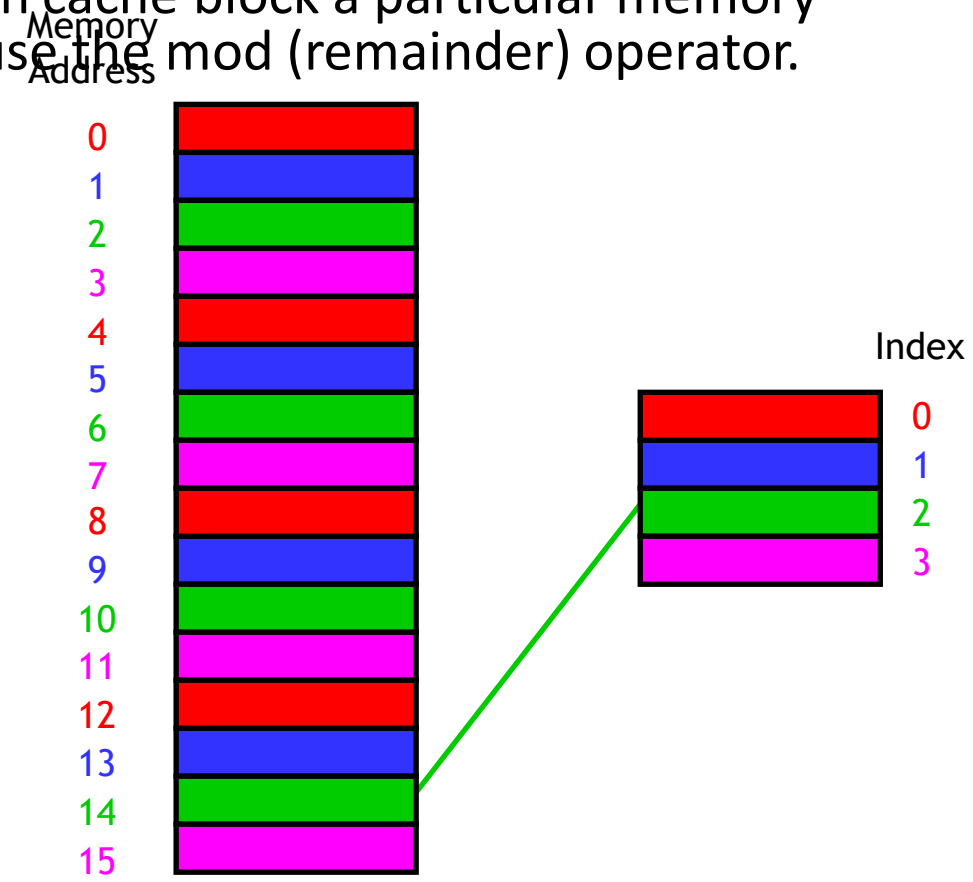
It's all divisions…

- One way to figure out which cache block a particular memory address should go to is to use the mod (remainder) operator.
- If the cache contains 2^k blocks, then the data at memory address *i* would go to cache block index

  $i \bmod 2^k$

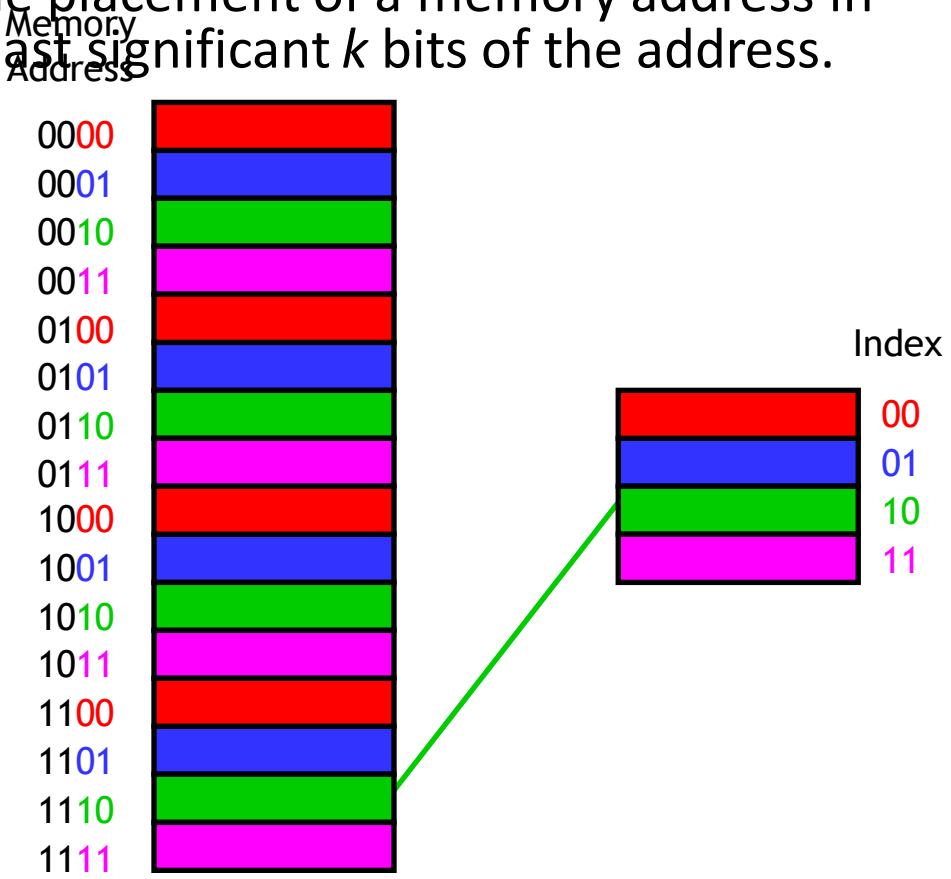
- For instance, with the four-block cache here, address 14 would map to cache block 2.

  $14 \bmod 4 = 2$



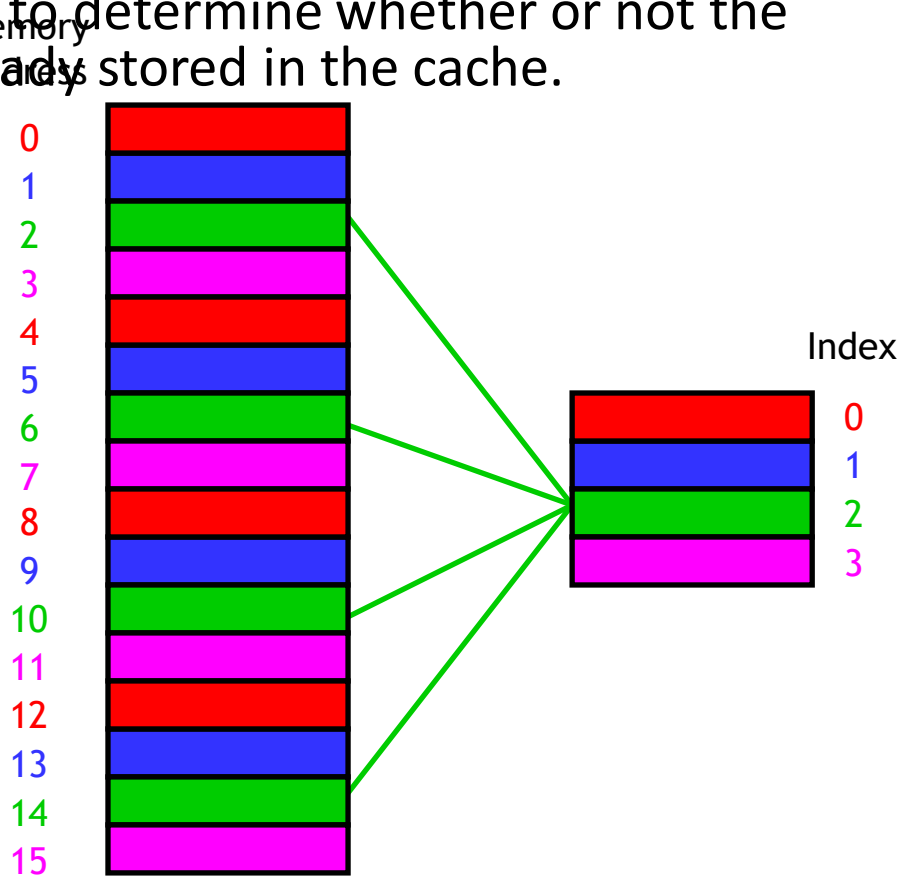
…or least-significant bits

- An equivalent way to find the placement of a memory address in the cache is to look at the least significant *k* bits of the address.
- With our four-byte cache we would inspect the two least significant bits of our memory addresses.
- Again, you can see that address 14 (1110 in binary) maps to cache block 2 (10 in binary).
- Taking the least *k* bits of a binary value is the same as computing that value mod 2^k.



How can we find data in the cache?

- The second question was how to determine whether or not the data we're interested in is already stored in the cache.
- If we want to read memory address *i*, we can use the mod trick to determine which cache block would contain *i*.
- But other addresses might *also* map to the same cache block. How can we distinguish between them?
- For instance, cache block 2 could contain data from addresses 2, 6, 10 *or* 14.



Adding tags

- We need to add tags to the cache, which supply the rest of the address bits to let us distinguish between different memory locations that map to the same cache block.

Assume,

Main Memory with,

→ 8 blocks, B_0 — B_7

→ 16 words.

→ 2^4 → 4 bit address.

→ 0000 to 1111

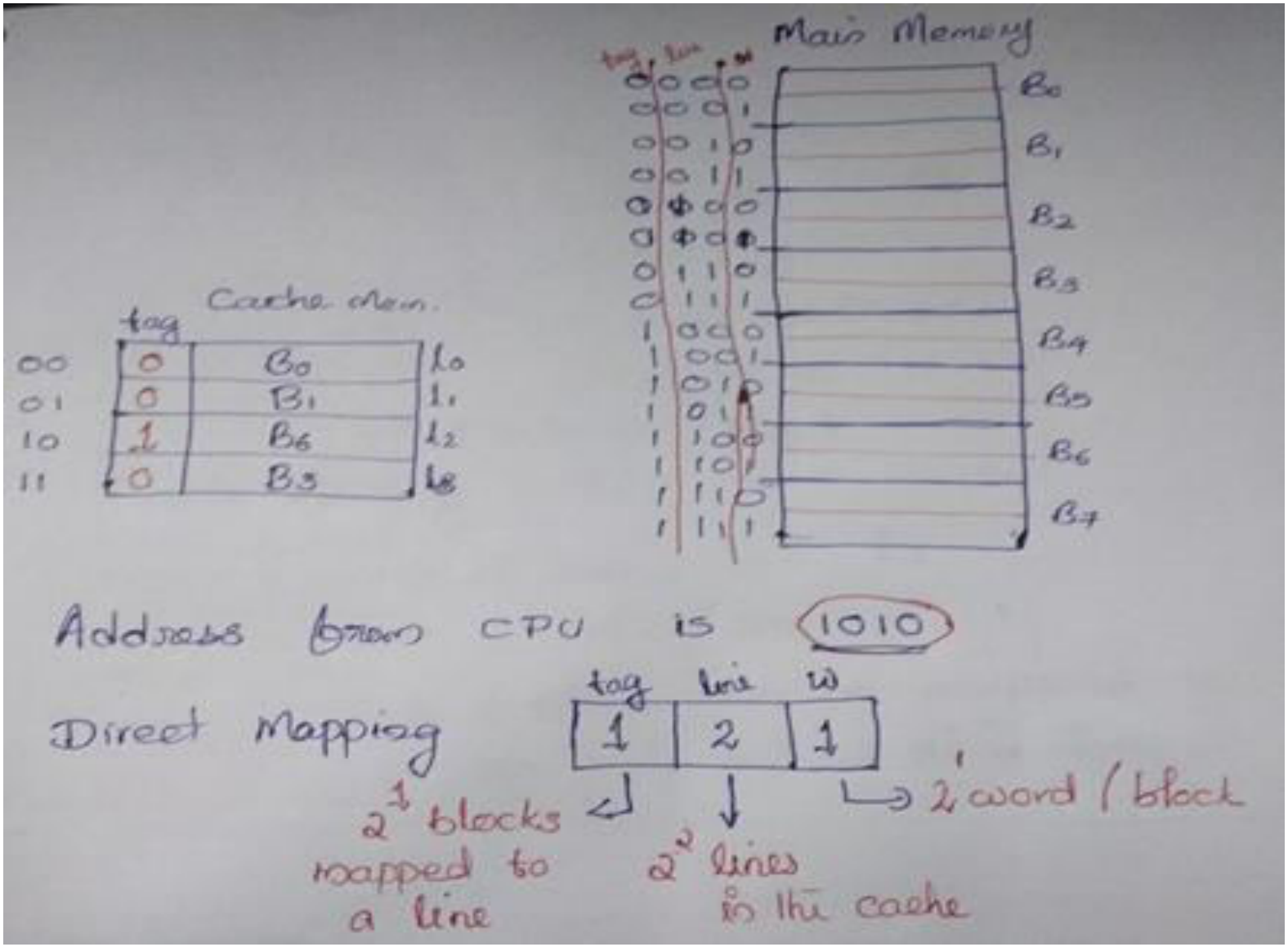
→ 2 words / block.

Cache Memory with,

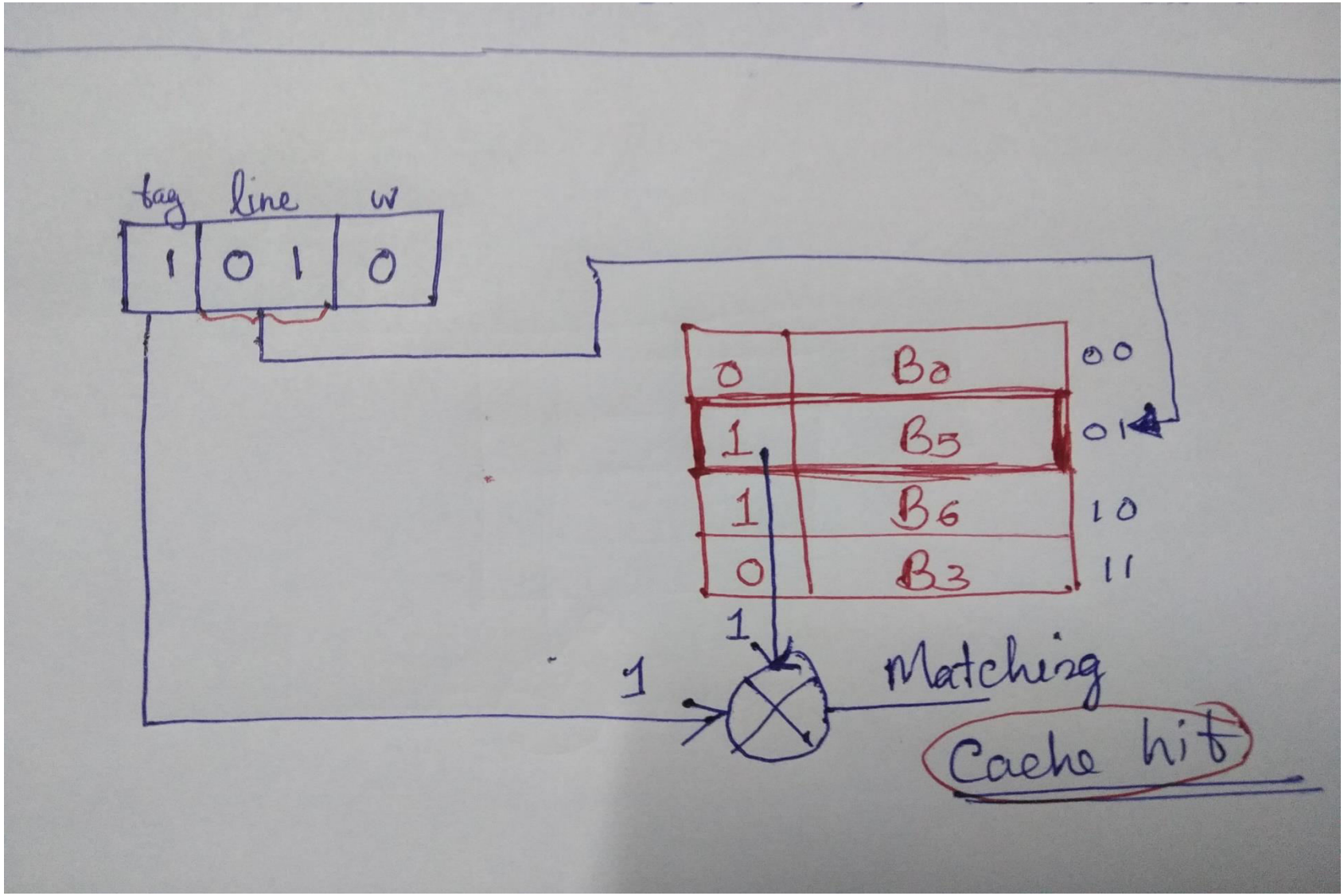
→ 4 lines, l_0 — l_3

→ 1 block / line.

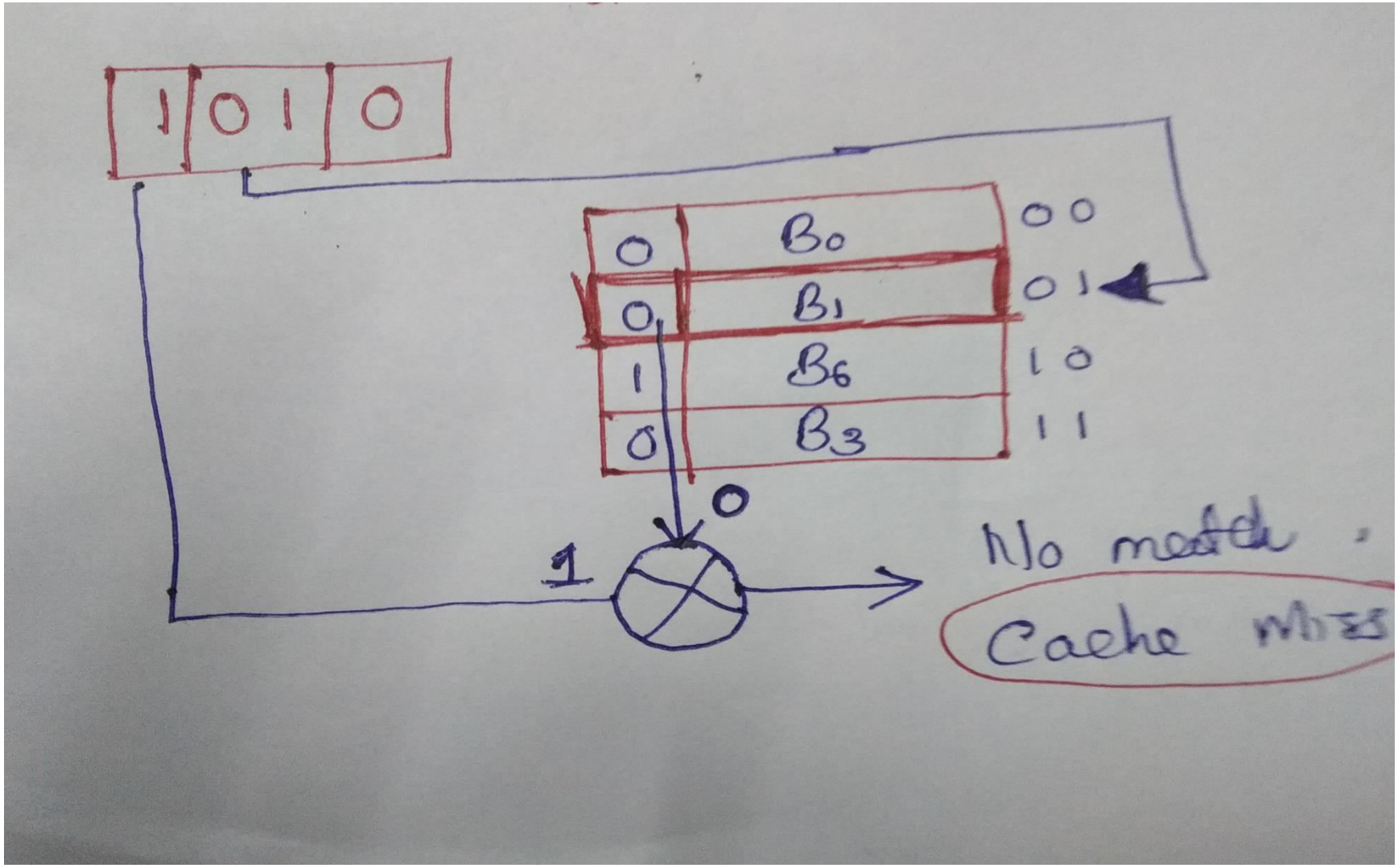
→ 2 block can be mapped to a line, but 1 at a time.


Main Memory
B0
B1
B2
B3
B4
B5
B6
B7
Cache Mem.
tag
00
01
10
11
0
0
1
0
B0
B1
B6
B3
l0
l1
l2
l3
Address from CPU is 1010
Direct Mapping
tag
line
w
1
2
1
2^1 blocks mapped to a line
2^2 lines in the cache
2^1 word / block

Cache hit

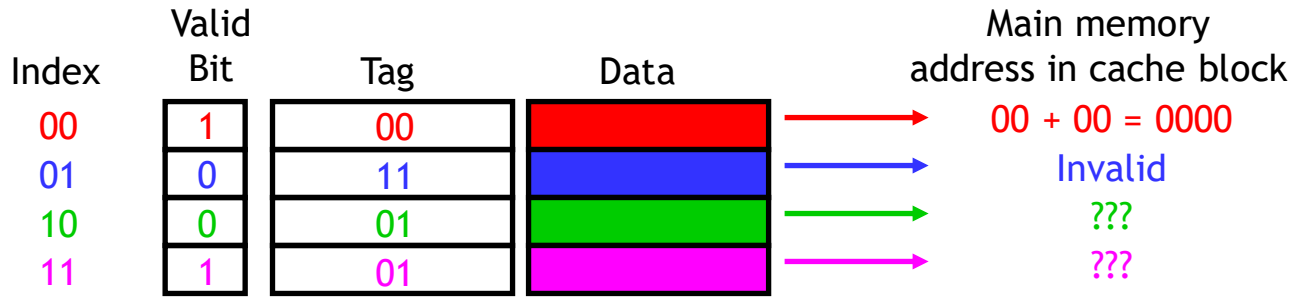

tag
line
w
1
0
1
0
0
B0
00
1
B5
01
1
B6
10
0
B3
11
1
1
Matching
Cache hit

Cache miss



One more detail: the valid bit

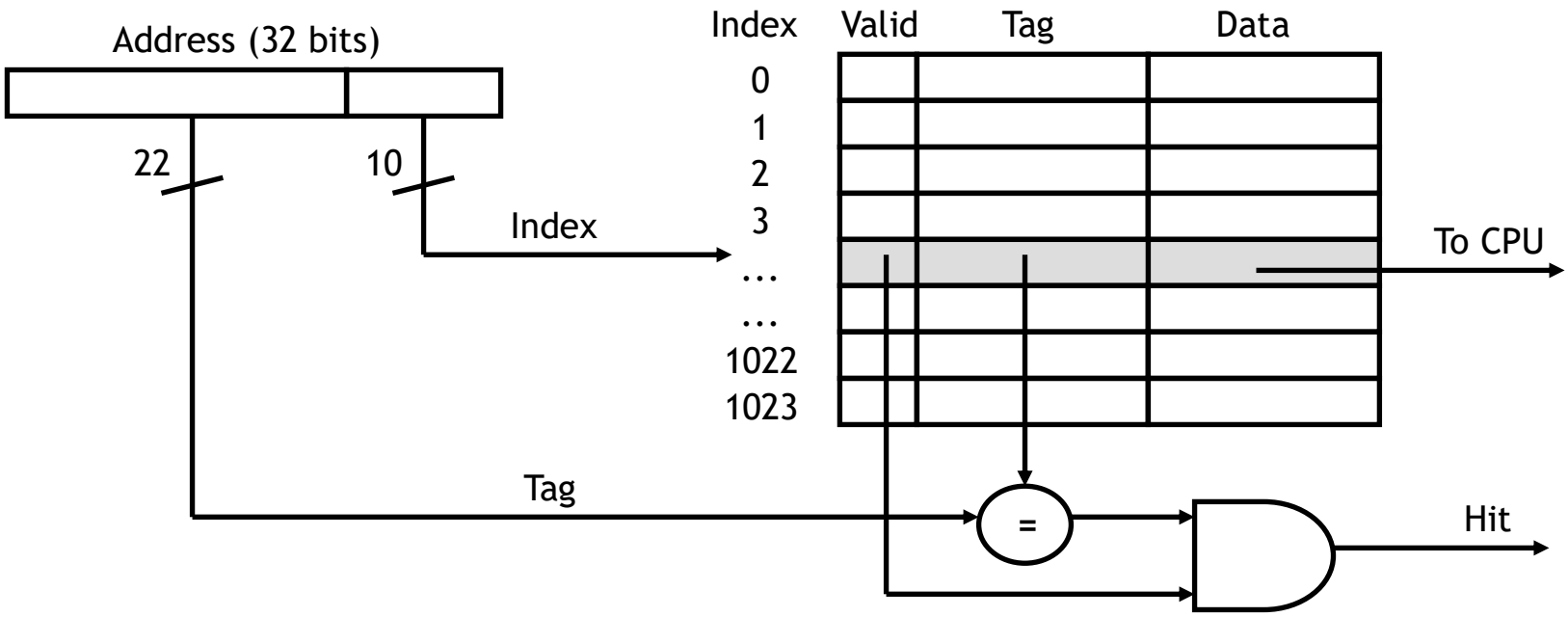
- When started, the cache is empty and does not contain valid data.
- We should account for this by adding a valid bit for each cache block.
  - When the system is initialized, all the valid bits are set to 0.
  - When data is loaded into a particular cache block, the corresponding valid bit is set to 1.



- So the cache contains more than just copies of the data in memory; it also has bits to help us find data within the cache and verify its validity.

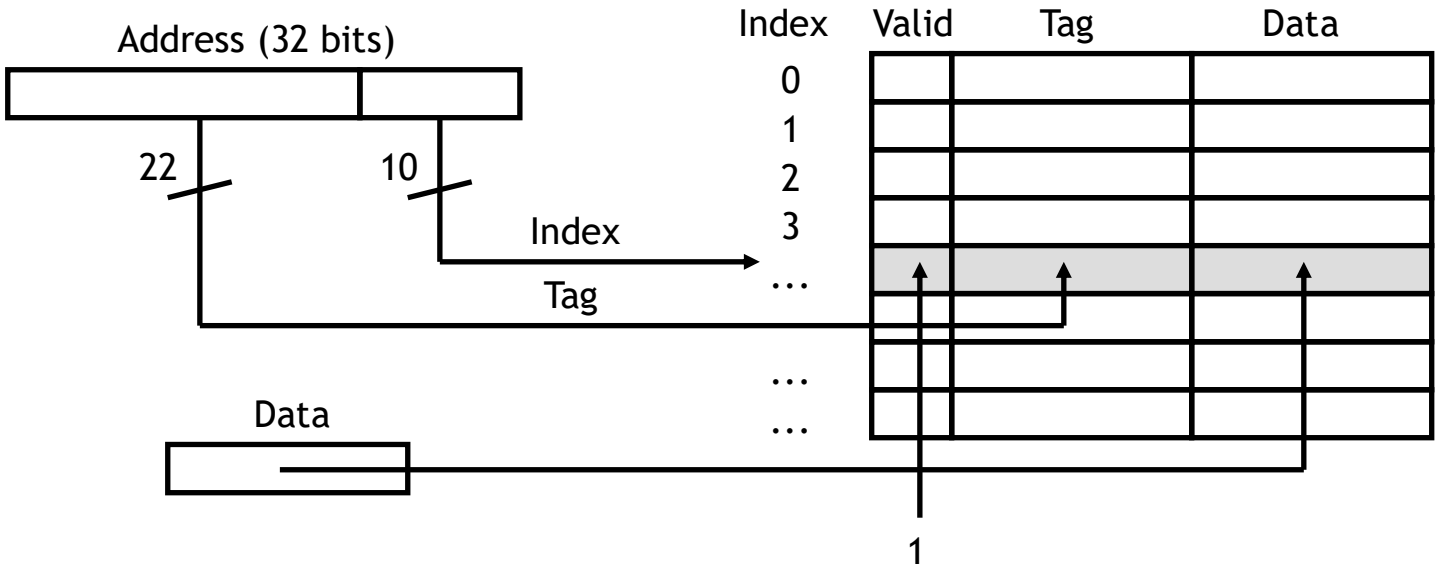
What happens on a cache hit

- When the CPU tries to read from memory, the address will be sent to a cache controller.
  - The lowest *k* bits of the address will index a block in the cache.
  - If the block is valid and the tag matches the upper (*m* − *k*) bits of the *m*-bit address, then that data will be sent to the CPU.
- Here is a diagram of a 32-bit memory address and a 2^{10}-byte cache.



Loading a block into the cache

- After data is read from main memory, putting a copy of that data into the cache is straightforward.
  - The lowest *k* bits of the address specify a cache block.
  - The upper (*m* − *k*) address bits are stored in the block's tag field.
  - The data from main memory is stored in the block's data field.
  - The valid bit is set to 1.



Mapping Function

Example 1

Assumption:

No. of lines(blocks) in a cache: 128

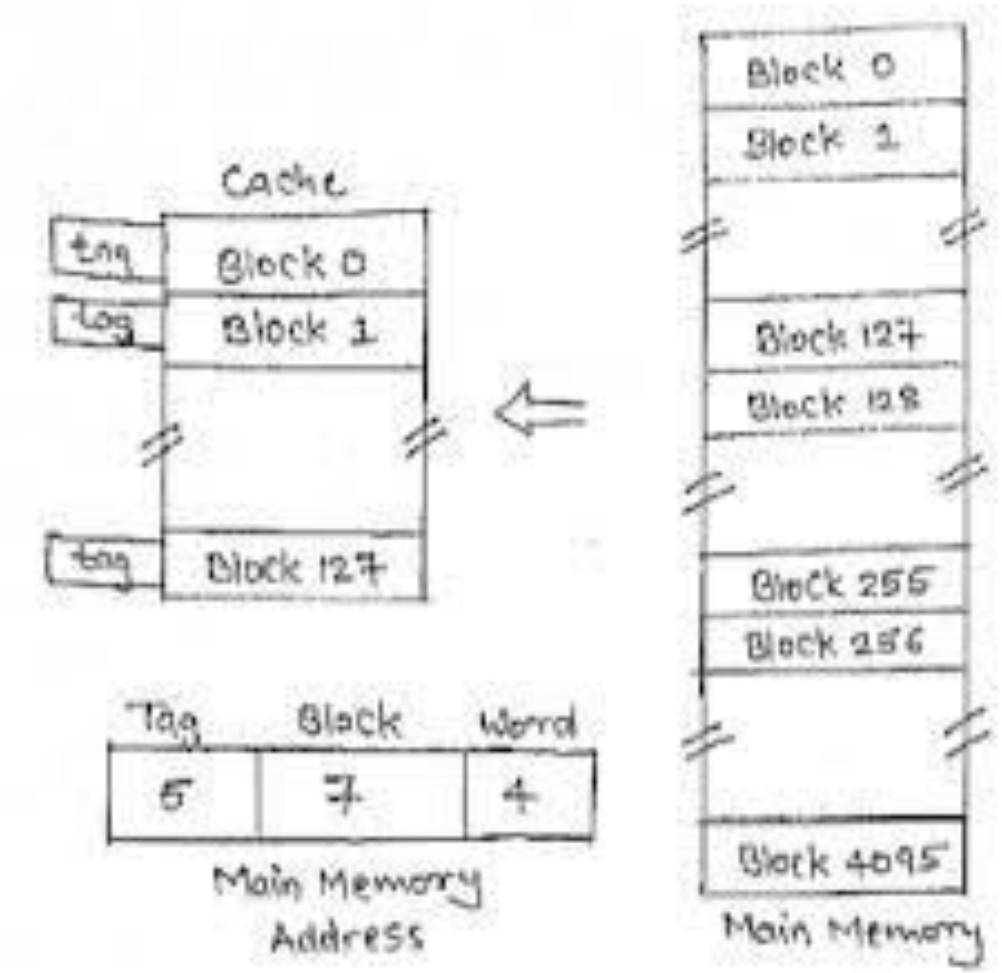
No. of blocks in main memory: 4096 blocks

No. of words in main memory: 64k words

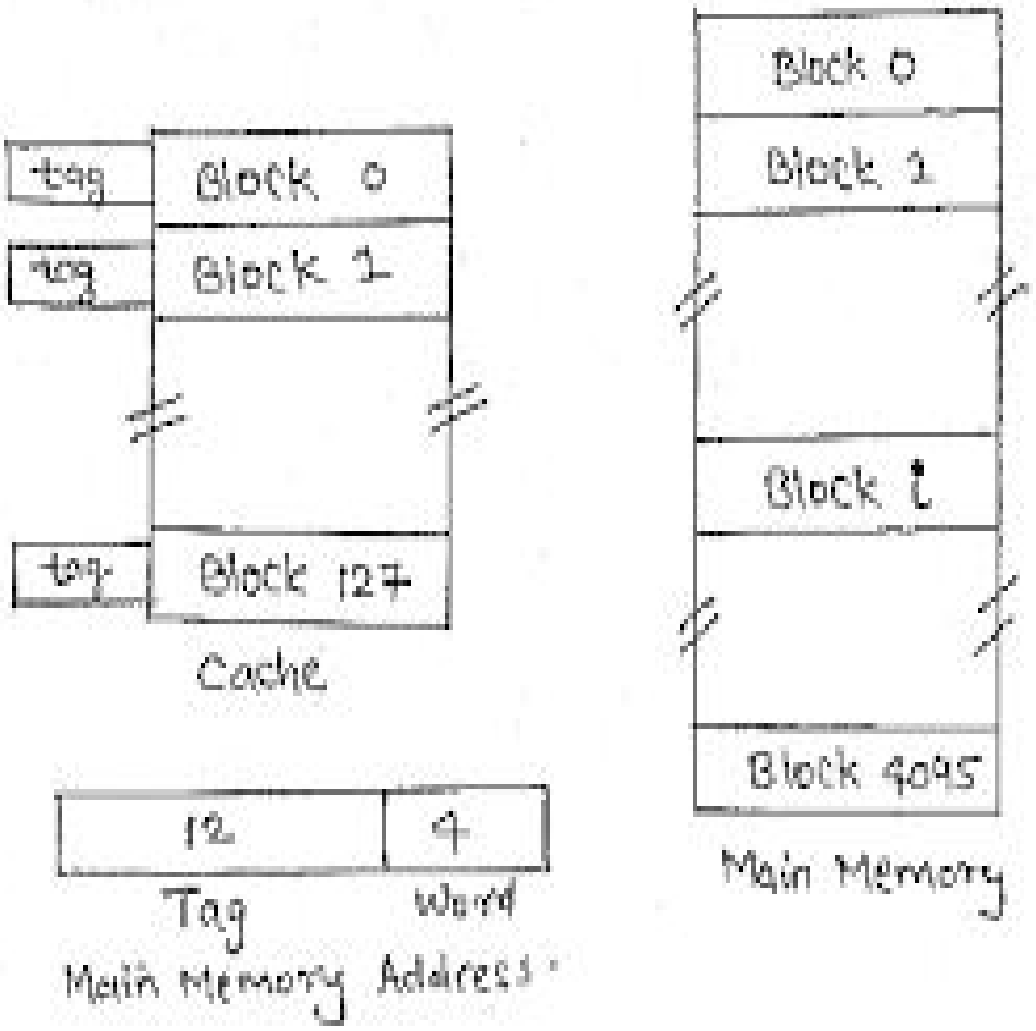
No. of Words in a block: 16

No. address bits: 16 ($64k=2^{10}*2^{6}$)

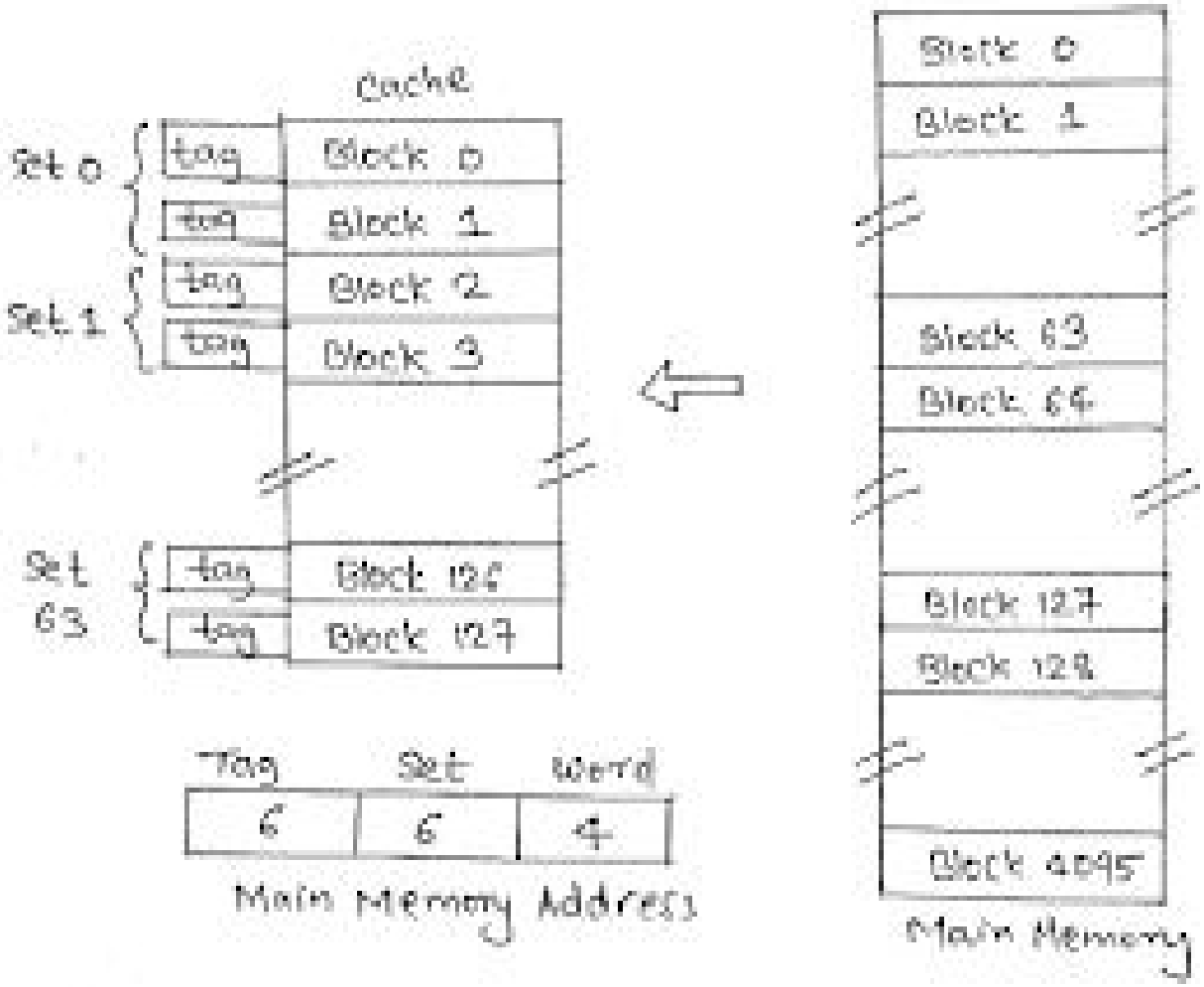
Direct Mapping



Fully Associative Mapping



2-Way Set Associative Mapping



Mapping Function

Example 2

Assumption:

No. of lines(blocks) in a cache: 128

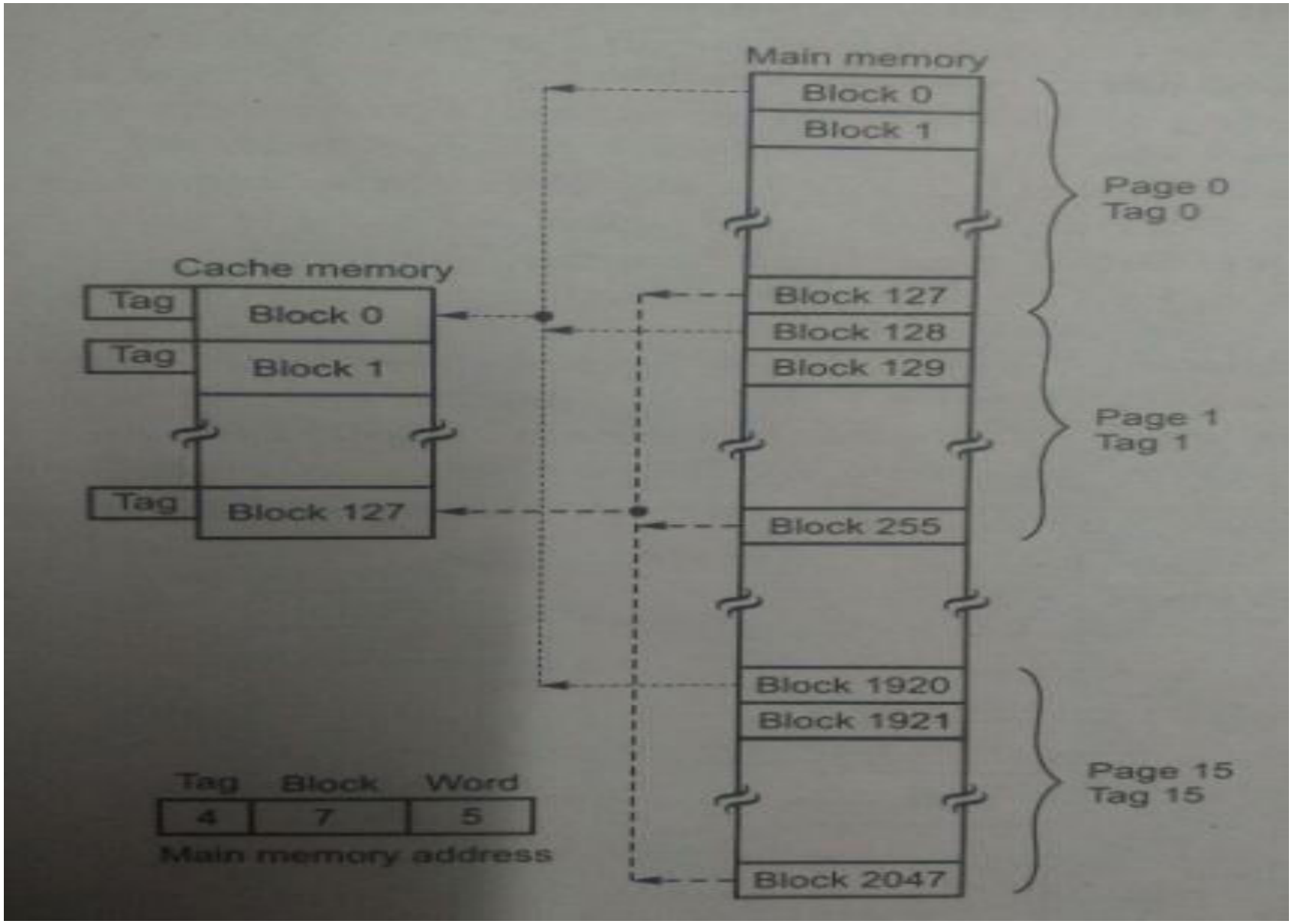
No. of blocks in main memory: 2046 blocks

No. of words in main memory: 64k words

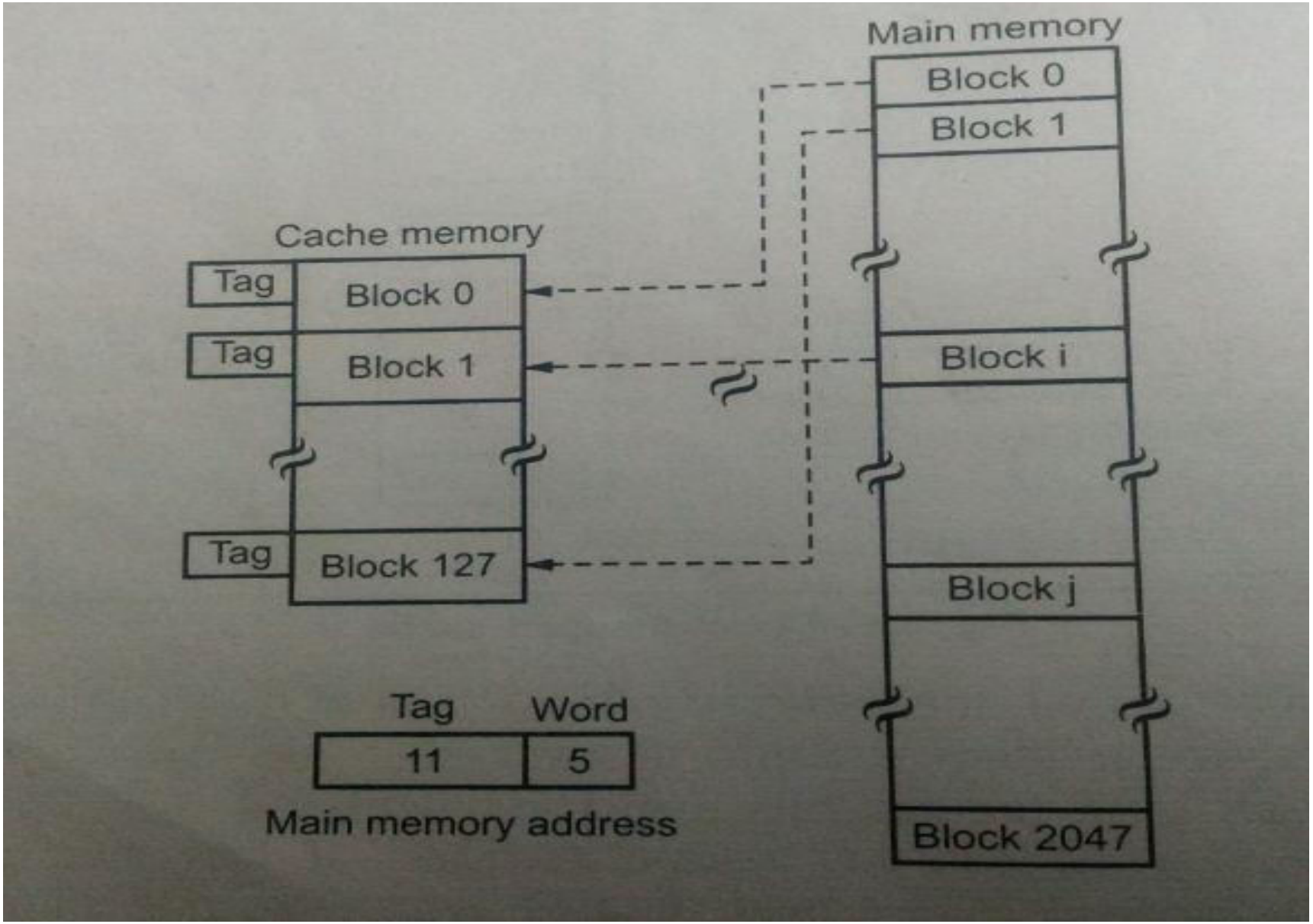
No. of Words in a block: 32

No. address bits: 16 (64k=2^{10}*2^{6})

Direct Mapping



Fully Associative Mapping



2-Way Set Associative Mapping

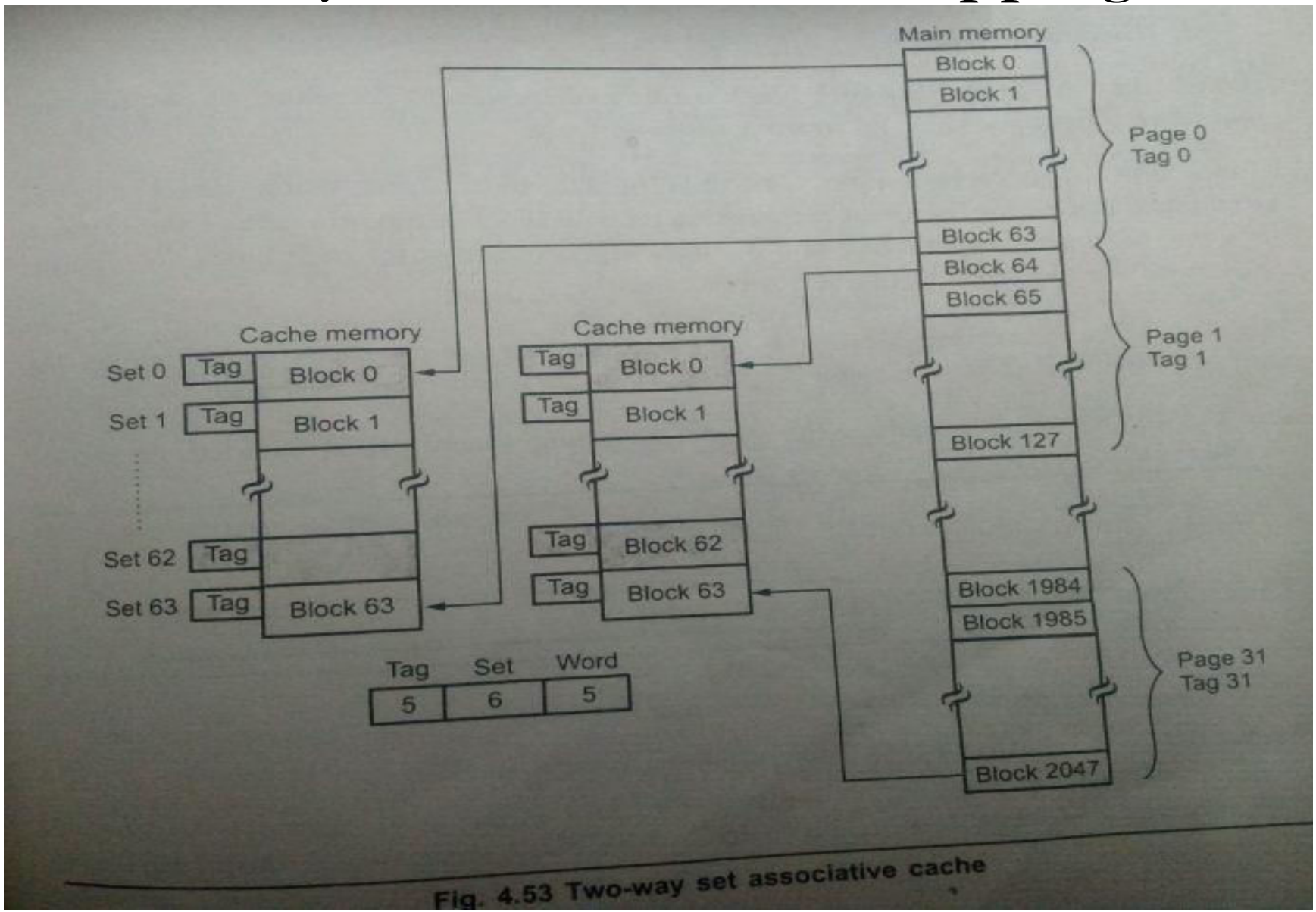


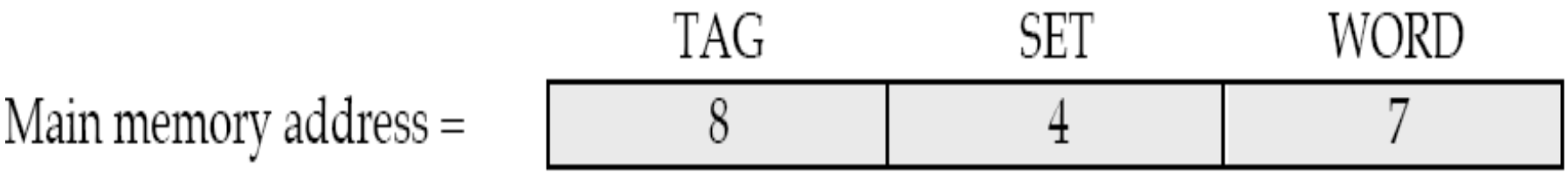
Fig. 4.53 Two-way set associative cache

Problem 1

- A set associative cache consists of 64 lines or slots, divided into four line sets. Main memory consists 4k blocks of 128 words each. Show the format of main memory addresses.

- A set associative cache consists of 64 lines or slots, divided into four line sets. Main memory consists 4k blocks of 128 words each. Show the format of main memory addresses.

The cache is divided into 16 sets of 4 lines each. Therefore, 4 bits are needed to identify the set number. Main memory consists of $4K = 2^{12}$ blocks. Therefore, the set plus tag lengths must be 12 bits and therefore the tag length is 8 bits. Each block contains 128 words. Therefore, 7 bits are needed to specify the word.



- Tag = $2^{12}/2^{4}$

Problem 2

- A two-way set associative cache has lines of 16 bytes and a total size of 8k bytes. The 64-Mbyte main memory is byte addressable. Show the format of main memory addresses.

Problem 2

- A two-way set associative cache has lines of 16 bytes and a total size of 8k bytes. The 64-Mbyte main memory is byte addressable. Show the format of main memory addresses.

There are a total of 8 kbytes / 16 bytes = 512 lines in the cache. Thus the cache consists of 256 sets of 2 lines each. Therefore 8 bits are needed to identify the set number. For the 64-Mbyte main memory, a 26-bit address is needed. Main memory consists of 64-Mbyte / 16 bytes = 2^{22} blocks. Therefore, the set plus tag lengths must be 22 bits, so the tag length is 14 bits and the word field length is 4 bits.

Main memory address =

TAG	SET	WORD
14	8	4

Problem 3

- Consider a cache consisting of 128 blocks of 16 words each, for a total of 2k words, and assume that the main memory is addressable by a 16 bit address and it consists of 4k blocks. Show the format of main memory address in all three types of mapping.
- Assume 2way set associative for set associative mapping.

Direct:

Tag → 5bits → $2^{12}/2^{7} = 2^{5}$
Block → 7bits → $2^{7} = 128$
Word → 4bits → 2^{4} =16 words in a block

Associative:

Set associative:

Tag → 6bits → $2^{12}/2^{6} = 2^{6}$
Set → 6bits → 2^{6} = 64 sets
Word → 4bits → 2^{4} =16 words in a block

problem4

- A block set associative cache consists of a total of 64-lines divided into four-line sets. The main memory contains 4096 blocks, each consisting of 128 words.
- What is the main memory address size?
- Format of main memory address?
- What is the size of cache memory?

- What is the main memory address size?
  - Total word=4096*128=2^{19}
  - Address size=19
- Format of main memory address?
  - Tag→8bits, set→4bits, word→7bits
- What is the size of cache memory? 64*128

Block Replacement

- **Least Recently Used: (LRU)**

  Replace that block in the set that has been in the cache longest with no reference to it.

- **First Come First Out: (FIFO)**

  Replace that block in the set that has been in the cache longest.

- **Least Frequently Used: (LFU)**

  Replace that block in the set that has experienced the fewest references

Update Policies - Write Through

- Update main memory with every memory write operation
- Cache memory is updated in parallel if it contains the word at specified address.
- Advantage:
- main memory always contains the same data as the cache
- easy to implement

  Disadvantage: -
- write is slower
- every write needs a main memory access

Write Back

- Only cache is updated during write operation and marked by flag. When the word is removed from the cache (at the time of replacement), it is copied into main memory
- *Advantage:*
  - writes occur at the speed of the cache memory
  - multiple writes within a block require only one write to main memory

  Disadvantage:
  - harder to implement
  - main memory is not always consistent with cache

Update policies – Contd..

- Write-Allocate
  - update the item in main memory and bring the block containing the updated item into the cache.
- Write-Around or Write-no-allocate
  - correspond to items not currently in the cache (i.e. write misses) the item is updated in main memory only without affecting the cache.

- ***Write Through with Write Allocate:***
  on hits it writes to cache and main memory
  on misses it updates the block in main memory and brings the block to the cache

- ***Write Through with No Write Allocate:***
  on hits it writes to cache and main memory;
  on misses it updates the block in main memory not bringing that block to the cache;

- ***Write Back with Write Allocate:***
  on hits it writes to cache setting dirty bit for the block, main memory is not updated;
  on misses it updates the block in main memory and brings the block to the cache;

- ***Write Back with No Write Allocate:***
  on hits it writes to cache setting dirty bit for the block, main memory is not updated;
  on misses it updates the block in main memory not bringing that block to the cache;

Update policies – Contd..

- Write back:-Write only in cache, updating main memory only at the time of replacement.
- Write through:-Both are updating for each write operation.
- Write-Allocate:- first in main memory, then copy the block into cache.
- Write-Around or Write-no-allocate:- when write miss occurred, updated in main memory without affecting the cache.

Performance analysis

- Look through: The cache is checked first for a hit, and if a miss occurs then the access to main memory is started.
- Look aside: access to main memory in parallel with the cache lookup.

- **Look through**
  $T_A = T_C + (1-h)*T_M$
  T_A – Average read access time
  T_C is the average cache access time
  T_M is the average main memory access time

- **Look aside**
  $T_A = h*T_C + (1-h)*T_M$

- hit ratio $h = \dfrac{\text{number of references found in the cache}}{\text{total number of memory references}}$

- **Miss Ratio m=(1-h)**

Example: assume that a computer system employs a cache with an access time of 20ns and a main memory with a cycle time of 200ns. Suppose that the hit ratio for reads is 90%,

a) what would be the average access time for reads if the cache is a look through-cache?

The average read access time ($\mathbf{T_A}$) = $\mathbf{T_C + (1-h)*T_M}$

$$20ns + 0.10*200ns = 40ns$$

b) what would be the average access time for reads if the cache is a "look-Aside" cache?

The average read access time in this case ($\mathbf{T_A}$)

$\mathbf{= h*T_C + (1-h)*T_M}$ = $0.9*20ns + 0.10*200ns = 38ns$

Problem 1

- Consider a memory system with T_c = 100ns and T_m = 1200ns. If the effective access time is 10% greater than the cache access time, what is the hit ratio H in look-through cache?

$\Rightarrow T_A = T_C + (1-h)*T_M$

$\Rightarrow 1.1\ T_c = T_c + (1-h)*T_M$

$\Rightarrow 0.1 T_C = (1-h)*\ T_M$

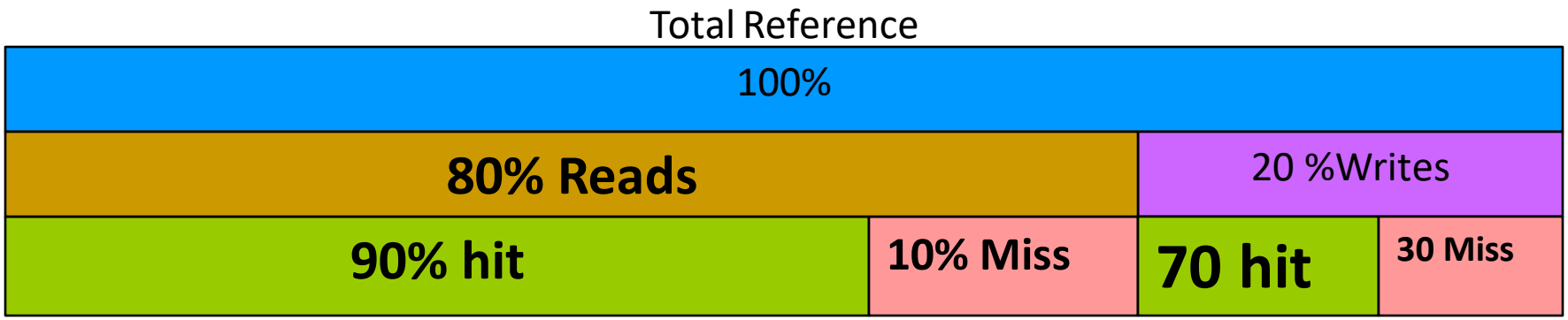
$\Rightarrow 0.1 * 100 = (1-h) *1200$

$\Rightarrow 1-h = 10/1200$

$\Rightarrow h = 1190/1200$

Problem 2

- A computer system employs a write-back cache with a 70% hit ratio for writes. The cache operates in look-aside mode and has a 90% read hit ratio. Reads account for 80% of all memory references and writes account for 20%. If the main memory cycle time is 200ns and the cache access time is 20ns, what would be the average access time for all references (reads as well as writes)?



The average access time for reads

$$= 0.9*20ns + 0.1*200ns = 38ns.$$

The average write time

$$= 0.7*20ns + 0.3*200ns = 74ns$$

Hence the overall average access time for combined reads and writes is

$$=0.8*38ns + 0.2*74ns = 45.2ns$$

References

- J. L. Hennessy & D.A. Patterson, Computer architecture: A quantitative approach, Fourth Edition, Morgan Kaufman, 2004.