Module 5b:
Introduction To Memory System (CACHE MEMORY)

REFERENCES:

STALLINGS, COMPUTER ORGANIZATION AND ARCHITECTURE

MORRIS MANO, COMPUTER ORGANIZATION AND ARCHITECTURE

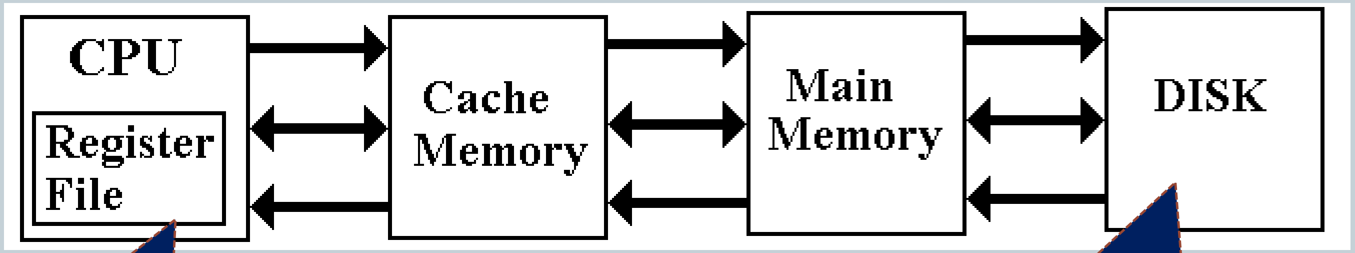
PATTERSON AND HENNESSY, COMPUTER ORGANIZATION AND DESIGN

NULL AND LOBUR, THE ESSENTIALS OF COMPUTER ORGANIZATION AND ARCHITECTURE

DOUGLAS E. COMER, ESSENTIALS OF COMPUTER ARCHITECTURE

Logical Multi–Level View of Memory

- A three–level view with cache memory, main memory, and virtual memory

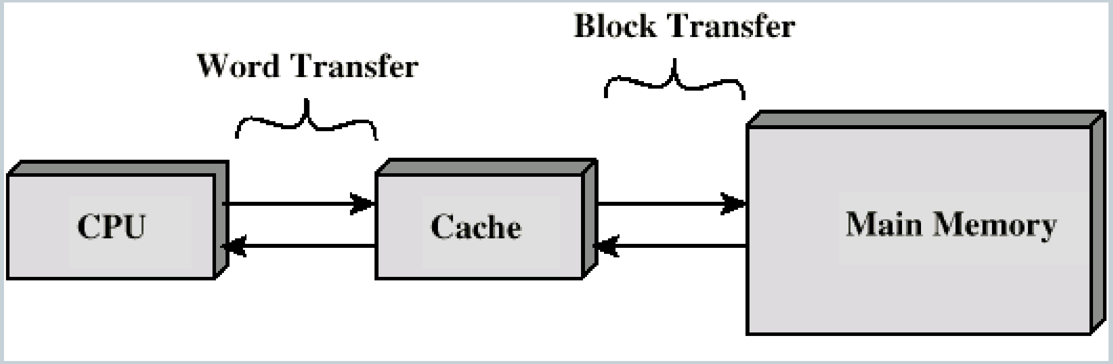


Smaller, faster, more expensive memory

Larger, not as fast, not as expensive memory

Cache Memory

- Small amount of fast, expensive memory.
- Sits between normal main memory (slower) and CPU.
- May be located on CPU chip or module.
- It keeps a copy of the most frequently used data from the main memory.
- Reads and writes to the most frequently used addresses will be serviced by the cache.
- We only need to access the slower main memory for less frequently used data.



Principle of Locality

- 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 and Data

- **Programs**
  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.

- **Data**
  Programs often access the same variables over and over, especially within loops.

Spatial Locality in Programs and Data

- **Programs**: 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*.
- **Data**: Programs often access data that is stored contiguously.
  - Arrays, like a in the code on the top, are stored in memory contiguously.
  - The individual fields of a record or object like employee are also kept contiguously in memory.

How caches take advantage of temporal locality?

- The first time the processor reads from an address in main memory, a copy of that data is also stored in the cache.
  - The next time that same address is read, we can use the copy of the data in the cache *instead of* accessing it from the slower dynamic memory.
  - Therefore the first read is a little slower than before since it goes through both main memory and the cache, but subsequent reads are much faster.
  - 1st – go memory, go cache
  - 2nd – go cache

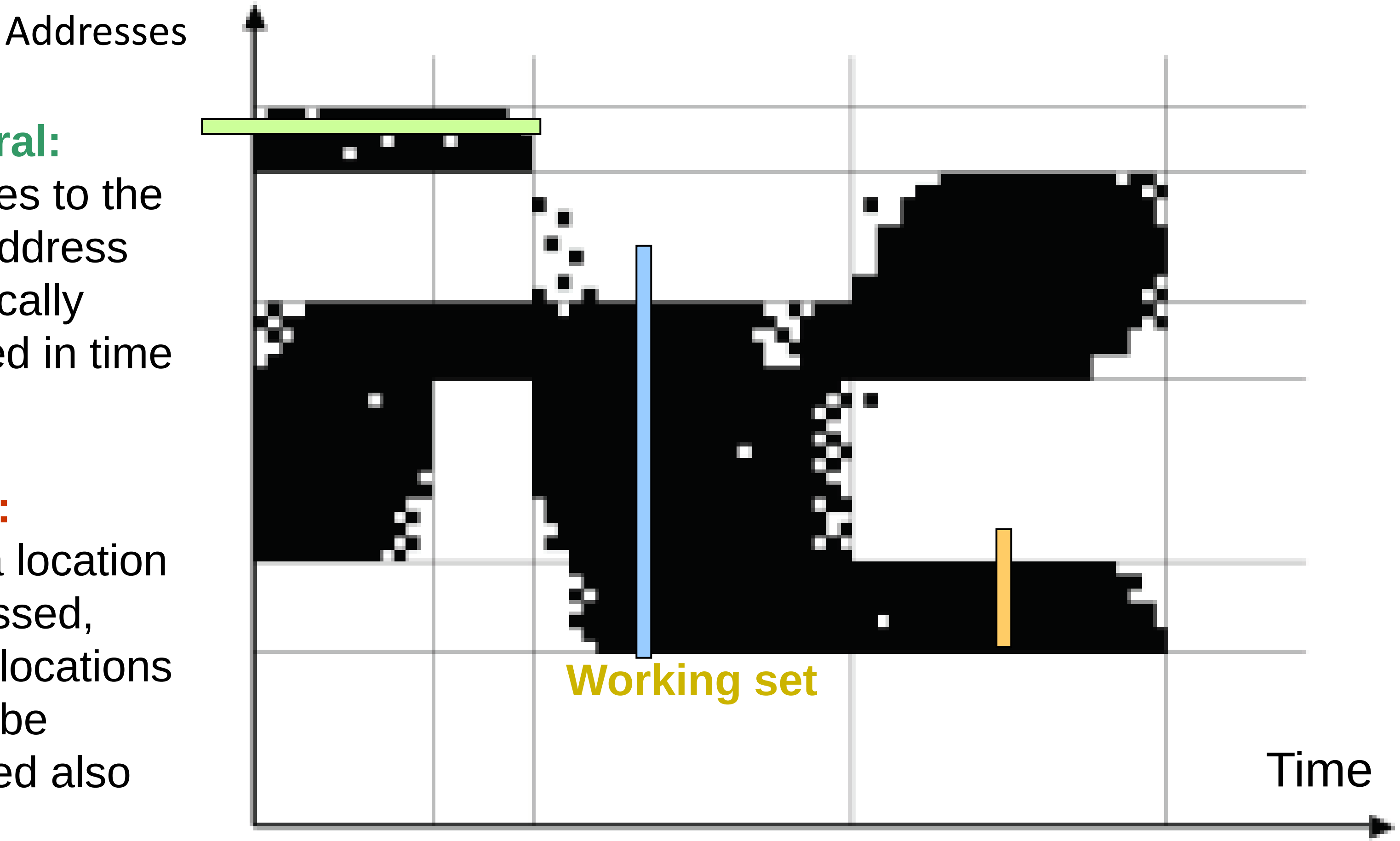
How caches take advantage of spatial locality?

- When the CPU reads location *i from main memory, a* copy of that data is placed in the cache.
- But instead of just copying the contents of location *i*, we can copy ***several values into the cache at once***, such as the four bytes from locations *i through i* + 3.
  - If the CPU later does need to read from locations *i + 1, i + 2 or i + 3, it can access that data from* the cache and not the slower main memory.
  - For example, instead of reading just one array element at a time, the cache might actually be loading four array elements at once.
- Again, the initial load incurs a performance penalty, but we're gambling on spatial locality and the chance that the CPU will need the extra data.

Temporal and Spatial Localities

Temporal:
Accesses to the same address are typically clustered in time

Spatial:
When a location is accessed, nearby locations tend to be accessed also



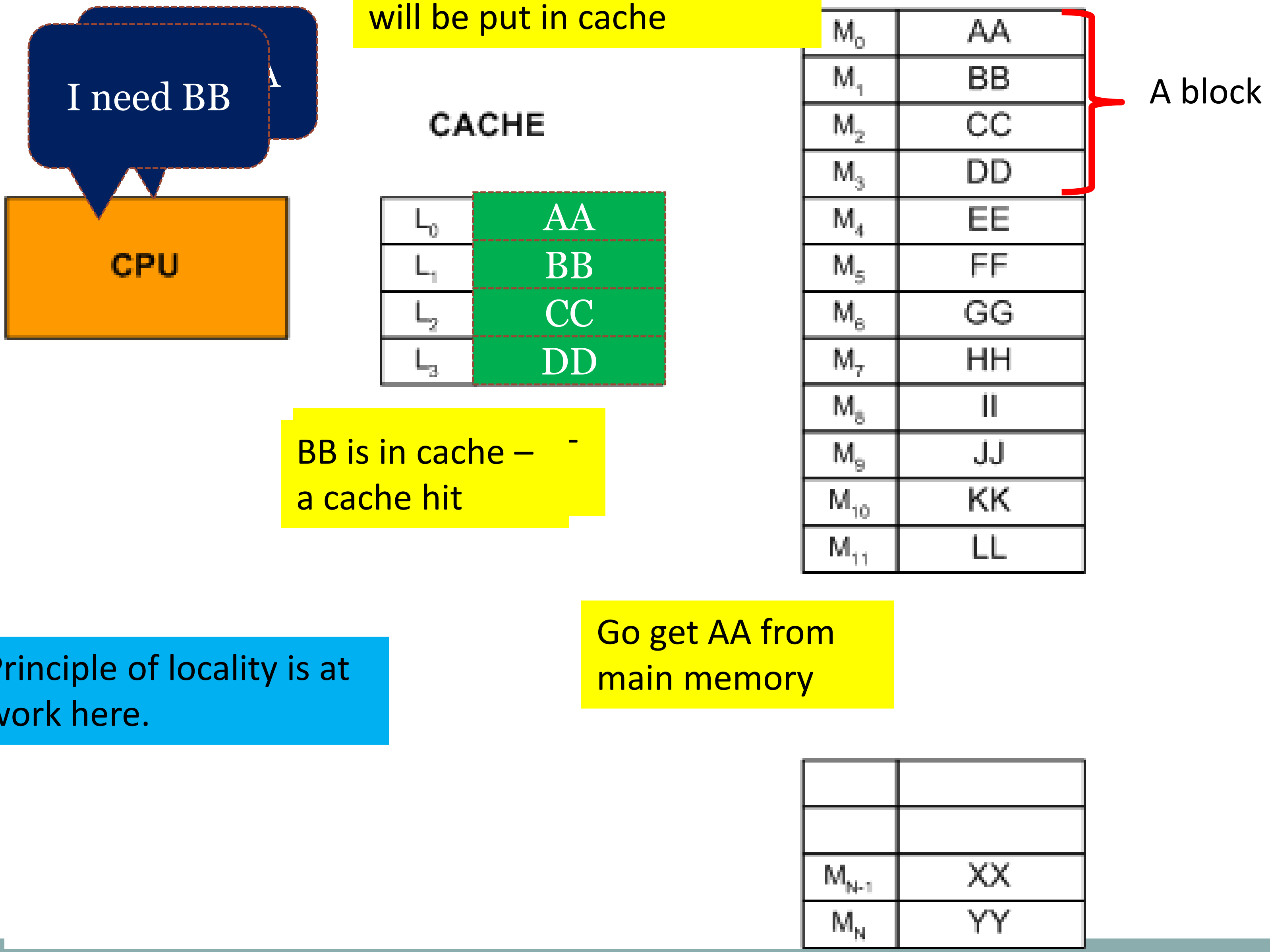
Hits and Misses

- A cache hit occurs if the cache contains the data that we're looking for.
  - cache can return the data much faster than main memory.
- A cache miss occurs if the cache does not contain the requested data.
  - CPU must then wait for the slower main memory.

How data is copied into cache?

- Main memory and cache are both divided into the same size blocks (the size of these blocks varies).
- When a memory address is generated, cache is searched first to see if the required word exists there.
- When the requested word is not found in cache, the entire main memory block in which the word resides is loaded into cache.
- As previously mentioned, this scheme is successful because of the principle of locality—if a word was just referenced, there is a good chance words in the same general vicinity/location will soon be referenced as well.

A block from main memory will be put in cache



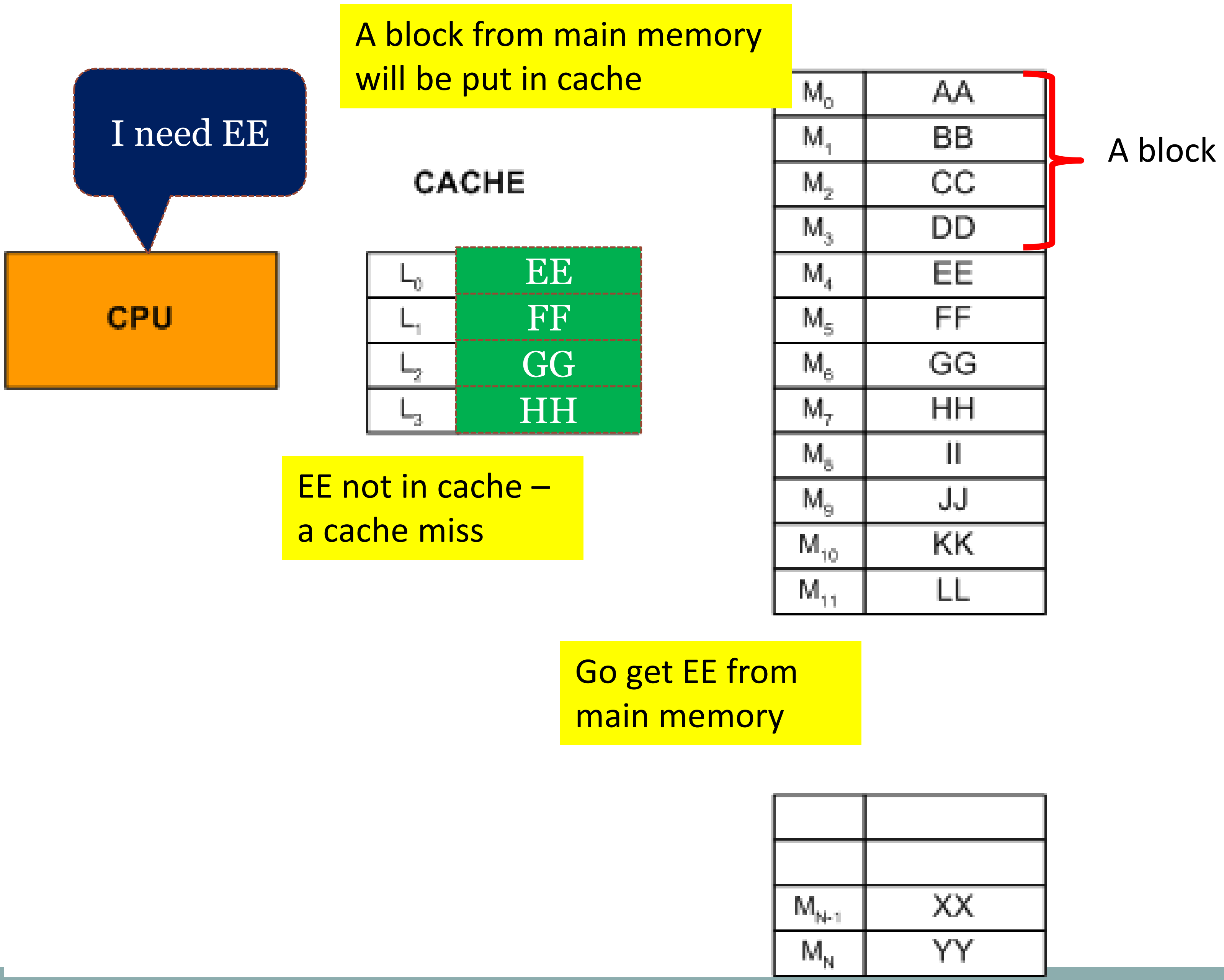
L_0	AA
L_1	BB
L_2	CC
L_3	DD

M_0	AA
M_1	BB
M_2	CC
M_3	DD
M_4	EE
M_5	FF
M_6	GG
M_7	HH
M_8	II
M_9	JJ
M_{10}	KK
M_{11}	LL
M_{N-1}	XX
M_N	YY

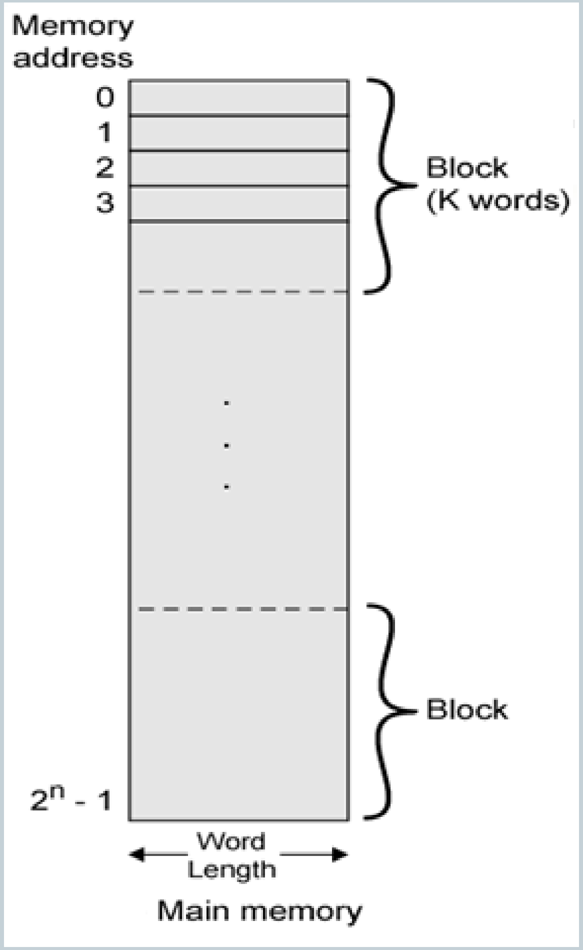
BB is in cache – a cache hit

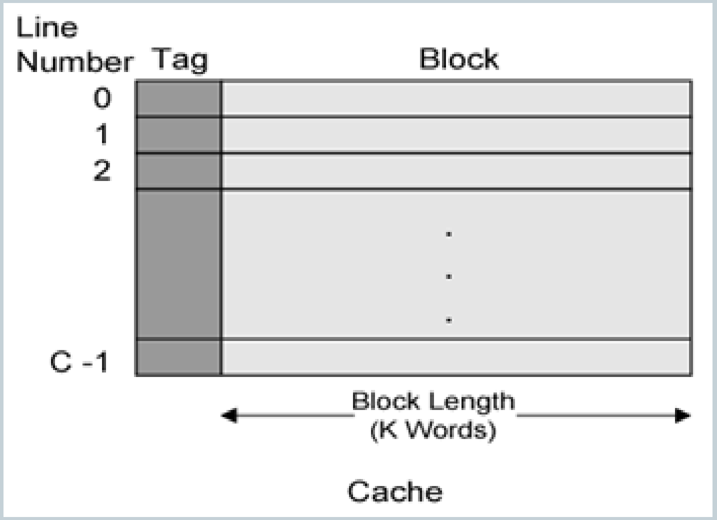
Go get AA from main memory

Principle of locality is at work here.


A block from main memory will be put in cache
I need EE
CPU
CACHE
L_0 EE
L_1 FF
L_2 GG
L_3 HH
EE not in cache – a cache miss
M_0 AA
M_1 BB
M_2 CC
M_3 DD
M_4 EE
M_5 FF
M_6 GG
M_7 HH
M_8 II
M_9 JJ
M_{10} KK
M_{11} LL
A block
Go get EE from main memory
M_{N-1} XX
M_N YY

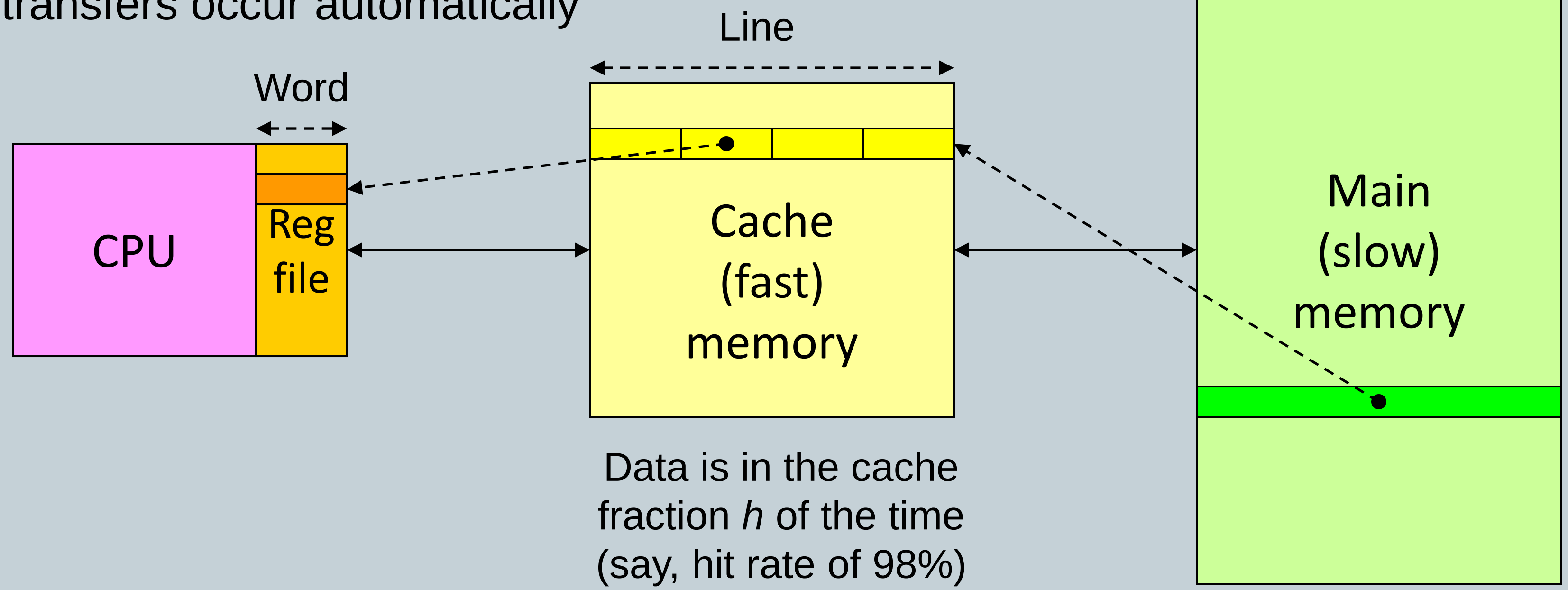
Cache-Main Memory Structure





Cache, Hit/Miss Rate, and Effective Access Time

Cache is transparent to user;
transfers occur automatically



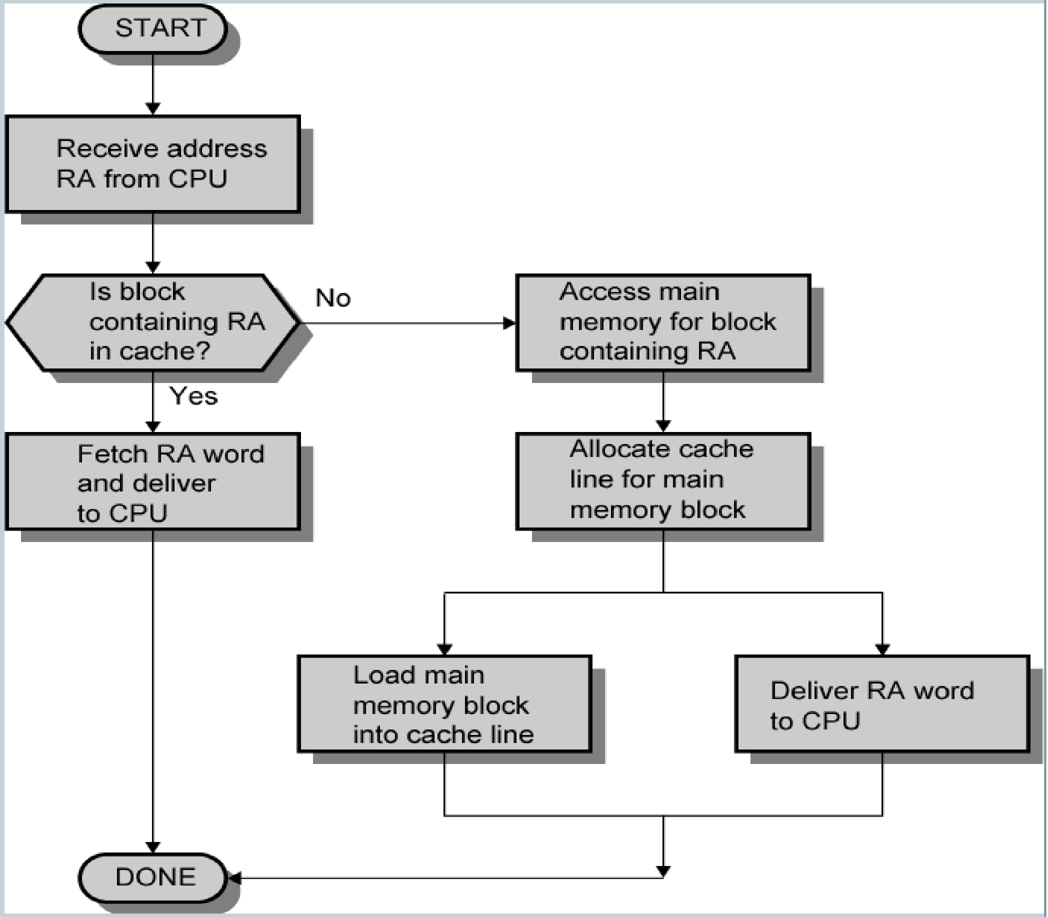
Data is in the cache fraction *h* of the time (say, hit rate of 98%)

One level of cache with hit rate *h*

$$C_{\text{eff}} = hC_{\text{fast}} + (1 - h)(C_{\text{slow}} + C_{\text{fast}}) = C_{\text{fast}} + (1 - h)C_{\text{slow}}$$

Cache Read Operation

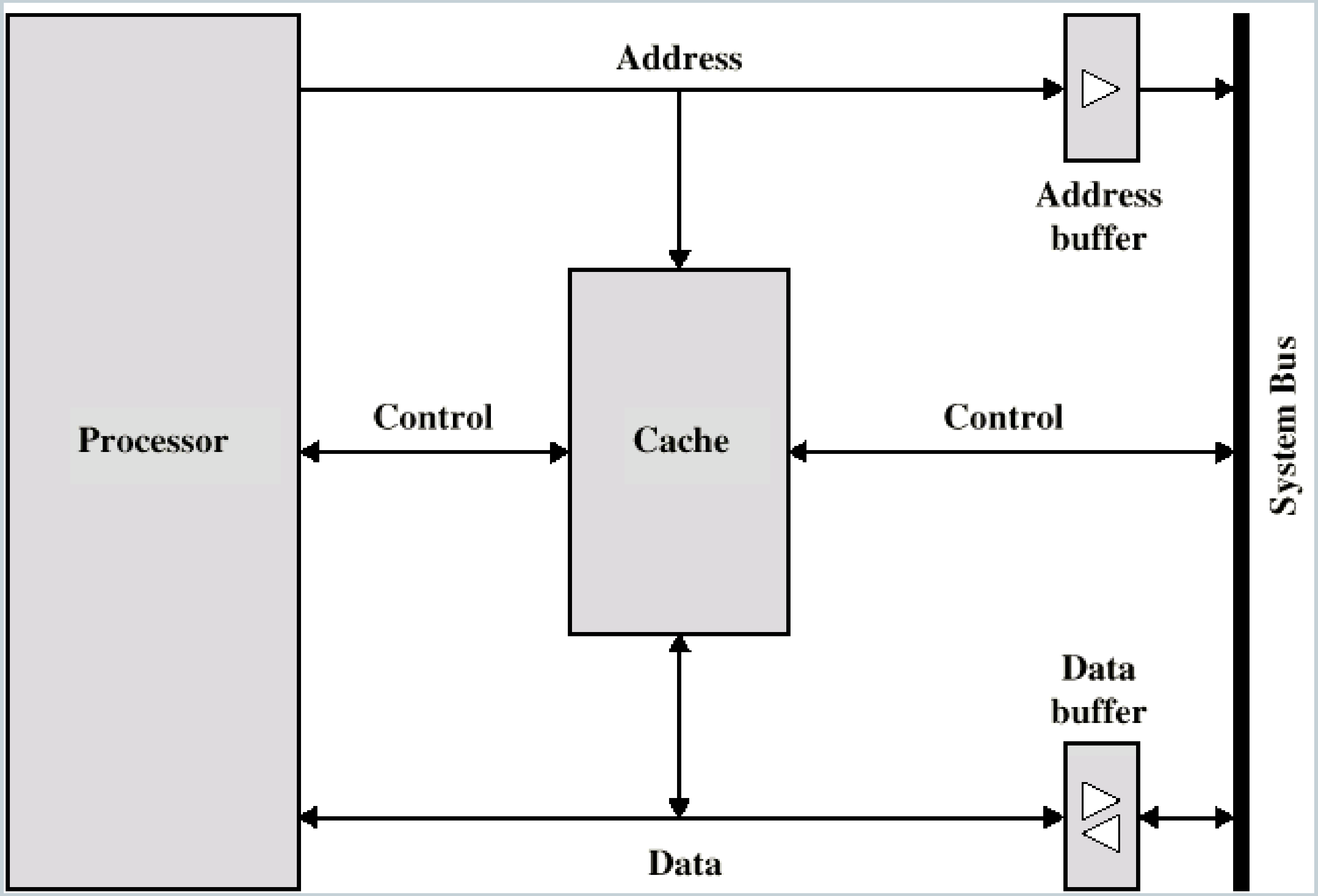
- CPU requests contents of memory location.
- Check cache for this data.
- If present, get from cache (fast).
- If not present, read required block from main memory to cache.
- Then deliver from cache to CPU.
- Cache includes tags to identify which block of main memory is in each cache slot.


START
Receive address RA from CPU
Is block containing RA in cache?
No
Yes
Access main memory for block containing RA
Allocate cache line for main memory block
Fetch RA word and deliver to CPU
Load main memory block into cache line
Deliver RA word to CPU
DONE

Size of Cache Design

- We would like the size of the cache to be
  - small enough so that the overall average cost per bit is close to that of main memory alone
  - large enough so that the overall average access time is close to that of the cache alone.
- There are several other motivations for minimizing cache size.
  - The larger the cache, the larger the number of gates involved in addressing the cache.
  - The result is that large caches tend to be slightly slower than small ones

Typical Cache Organization



Mapping functions

- Because there are fewer cache lines than main memory blocks, an algorithm is needed for mapping main memory blocks into cache lines.
- There is also a need for a way to determine which main memory block currently occupies a cache line.
- The choice of the mapping function dictates how the cache is organized.
- Three techniques can be used:
  - Direct (single or block mapping),
  - associative,
  - set associative.

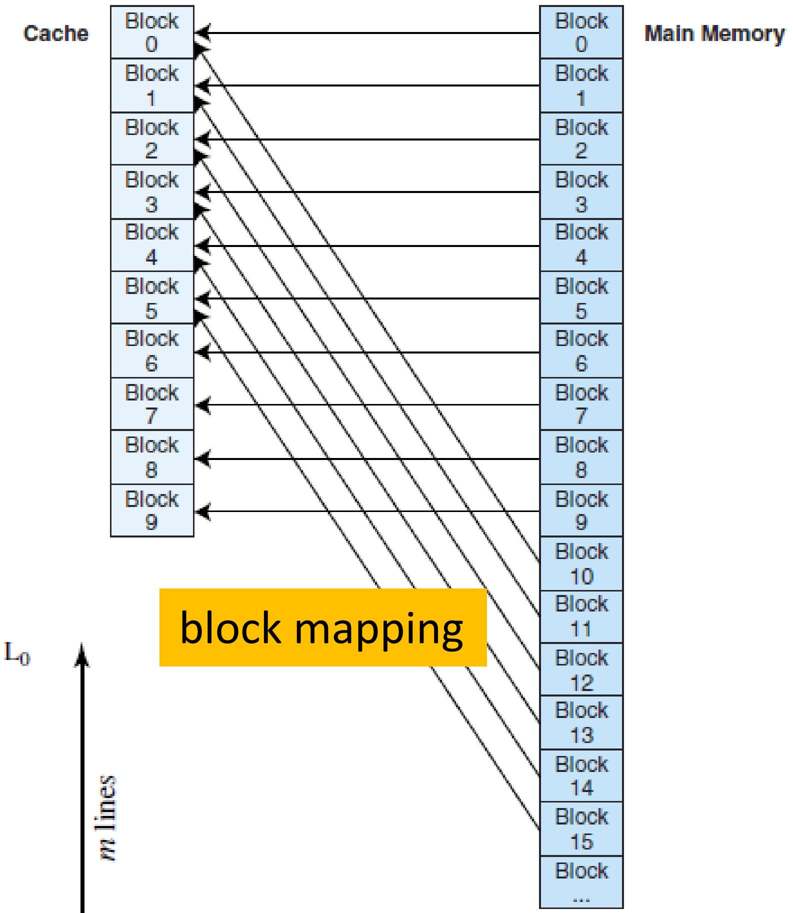
Direct Mapped cache

- Direct mapped cache assigns cache mappings using a **modular approach.**
- Because there are more main memory blocks than there are cache blocks, it should be clear that main memory blocks compete for cache locations.

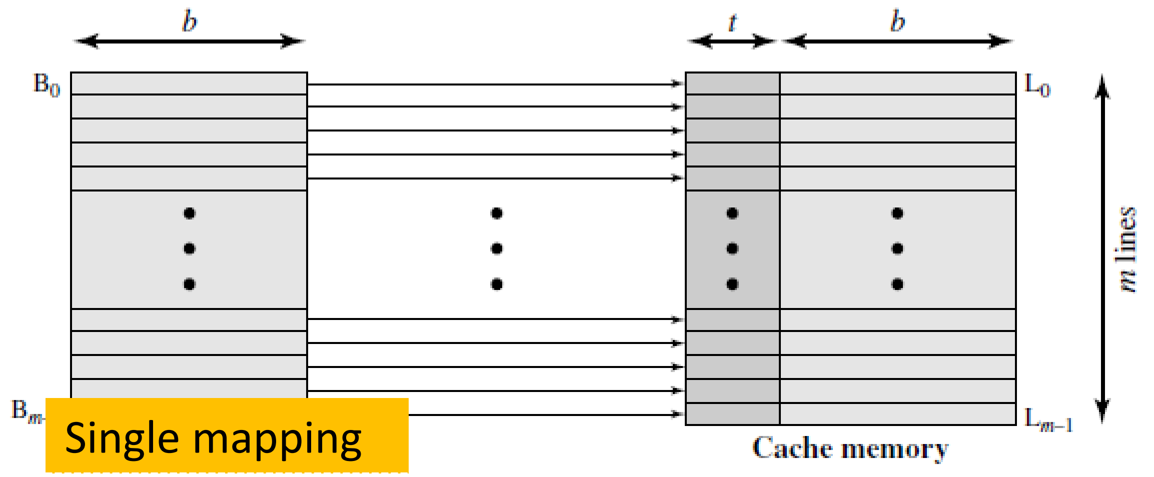
Y = X MOD N

- Direct mapping maps block *X of main memory to block Y of cache, mod N, where N is the total* number of blocks in cache.
- For example, if cache contains 10 blocks (N=10), then
  - Main memory block 0 maps to cache block 0, — Y = 0 MOD 10 = 0
  - main memory block 1 maps to cache block 1, — Y = 1 MOD 10 = 1
  - main memory block 9 maps to cache block 9,
  - main memory block 10 maps to cache block 0. — Y = 10 MOD 10 = 0

Direct Mappe



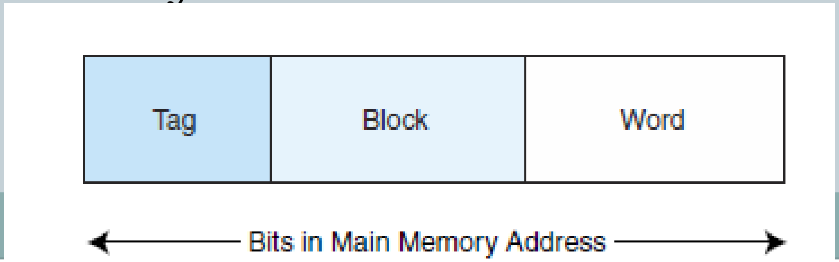
block mapping



Single mapping

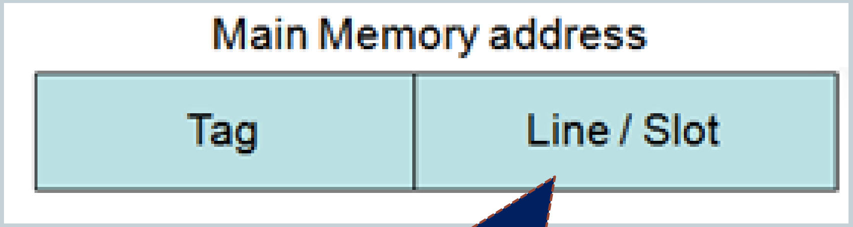
Direct Mapped cache

- If main memory blocks 0 and 10 both map to cache block 0, how does the CPU know which block actually resides in cache block 0 at any given time?
- The answer is that each block is copied to cache and identified by a tag.
- To perform direct mapping, the binary main memory address is partitioned into the fields.
- The size of each field depends on the physical characteristics of main memory and cache.

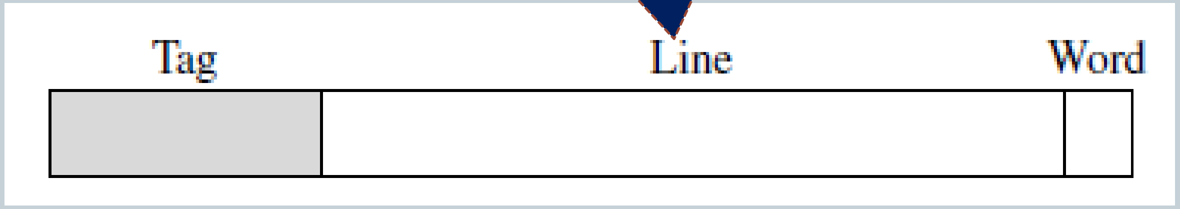


Direct Mapped cache

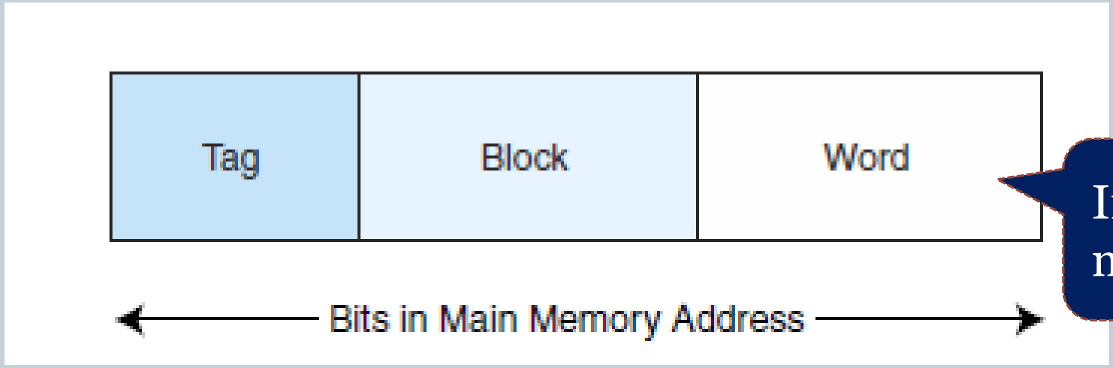
- In some examples the division is



Some authors use line instead of block



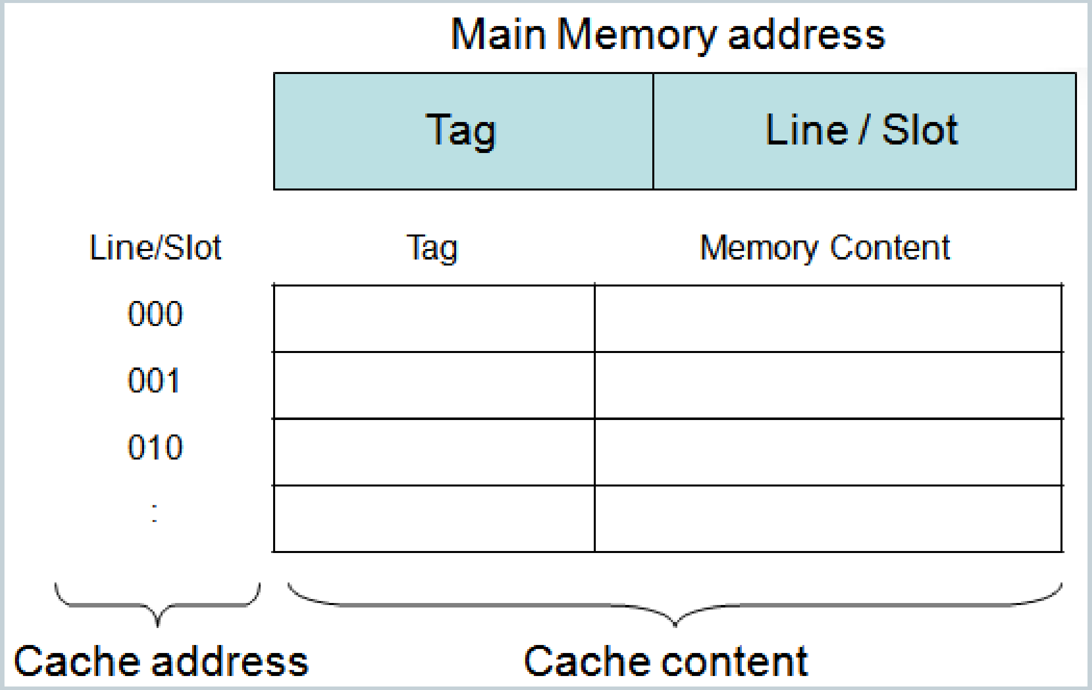
In this case, single word in each line/slot



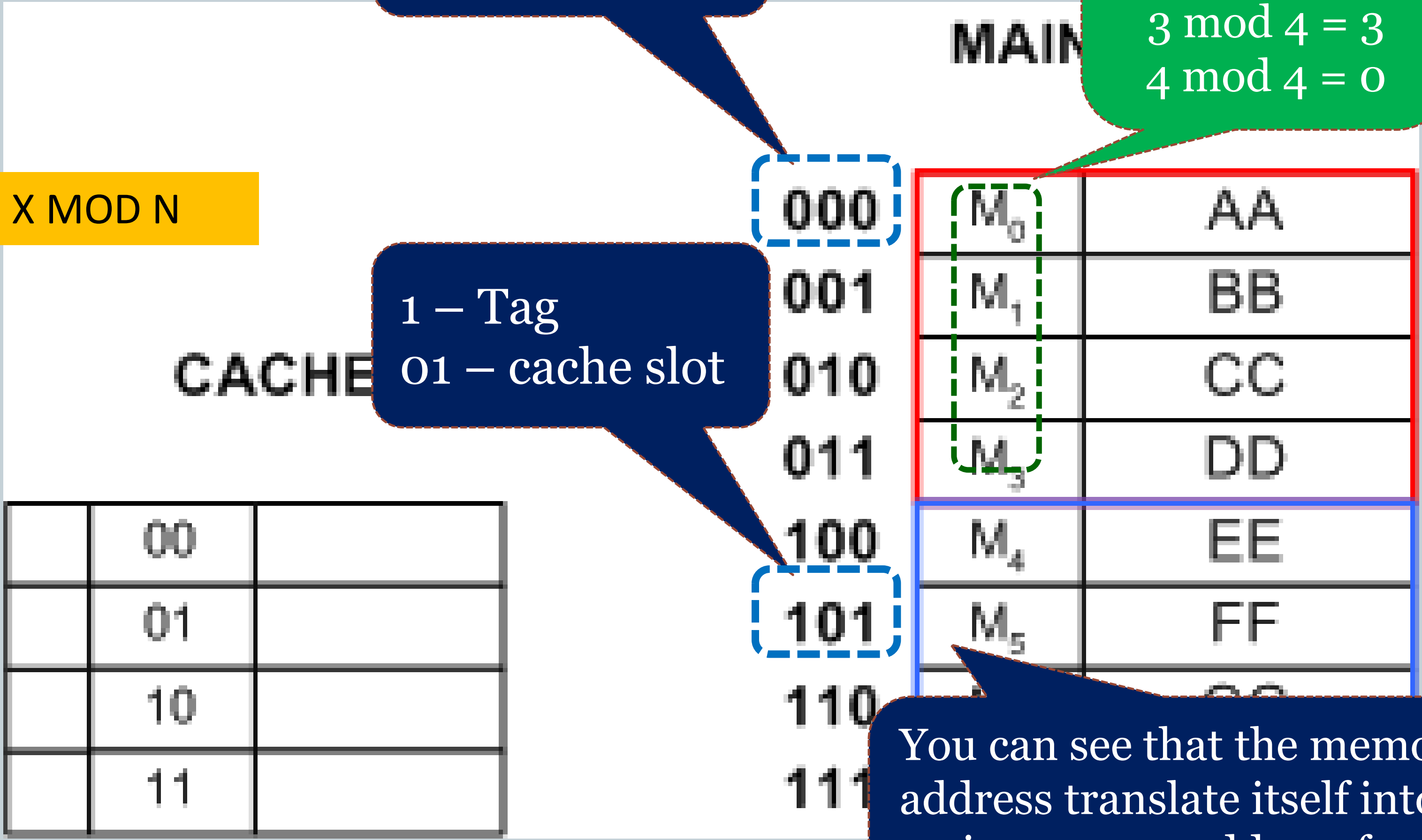
In this case, block mapping

Direct Mapping

- Each block of main memory maps to only one cache line
  - i.e. if a block is in cache, it must be in one specific place
- Address is in two parts



Y = X MOD N



000	M_0	AA
001	M_1	BB
010	M_2	CC
011	M_3	DD
100	M_4	EE
101	M_5	FF
110	[illegible]	[illegible]
111		

	00	
	01	
	10	
	11	

0 – Tag
00 – cache slot

1 – Tag
01 – cache slot

0 mod 4 = 0
1 mod 4 = 1
2 mod 4 = 2
3 mod 4 = 3
4 mod 4 = 0

You can see that the memory address translate itself into the main memory address format
➔ M5 into 101, M4 into 100

N = 4 SLOTS

Example 1: Direct Mapping

A main memory contains 8 words while the cache has only 4 words. Using direct address mapping, identify the fields of the main memory address for the cache mapping.

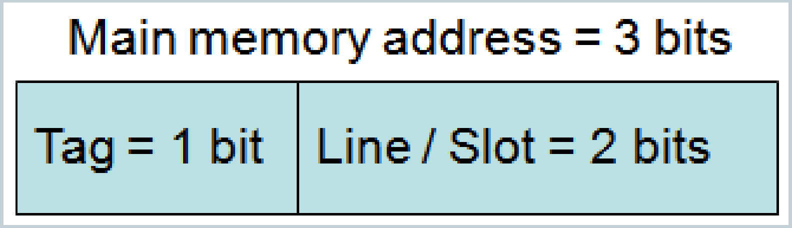
Solution:

Total memory words = 8 = 2^3

→ Require 3 bit for main memory address.

Total cache words = 4 = 2^2

→ Require 2 bit for cache address → line / slot.



Main memory address = 3 bits	
Tag = 1 bit	Line / Slot = 2 bits

Example 2: Direct Mapping

A main memory system has the following specification:

- Main memory contain 16 words, each word is 16 bit long
- Cache memory contain 4 words, each word is 16 bit

i. Identify the size (bits) of tag ar

ii. What is the size of the cache w

iii. Draw the memory system and

Solution:

Total memory words = 16 = 2^4

→Require 4 bit for main memory address.

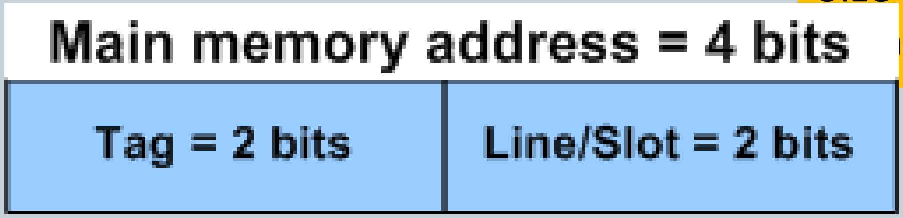
Total cache words = 4 = 2^2

→Require 2 bit for cache address → line / slot

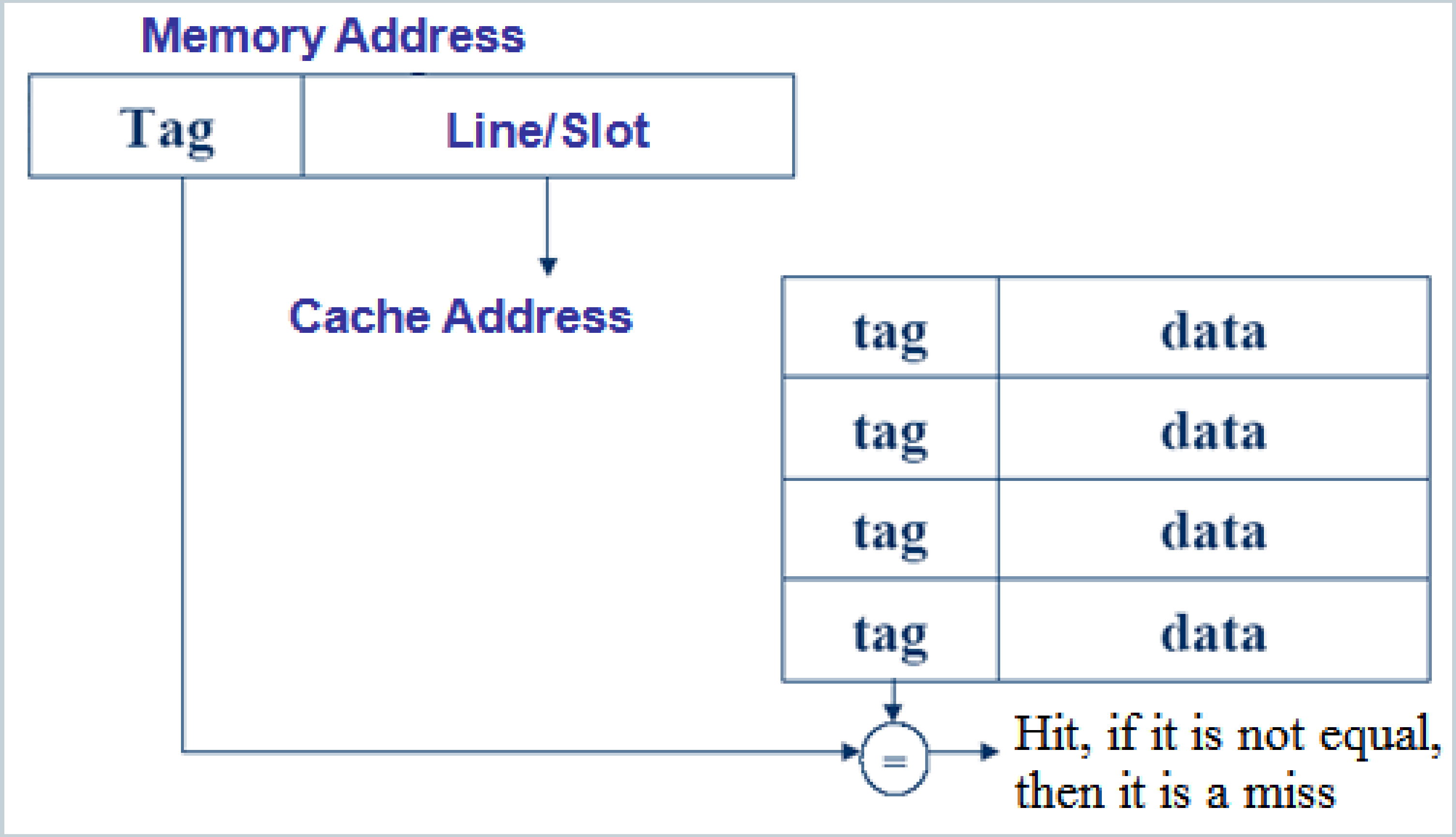
The tag size = 4-2 = 2 bits

Size of cache word = Tag + (No words in cache ×

) = 2 + (1 x 16) = 18 bits



Direct Mapping: Hit or Miss



Example 3A: Hit and Miss

Assume cache is empty

Consider an eight-word direct mapped cache. Show the
che as it responds to a series of
addresses):

8 word → need 3 bits for slots

22, 26, 22, 26, 16, 3, 16, 18

10110
→ 10 = tag, 110 = slot

2				16	3	16	18
10110	11010	10110	11010	10000	00011	10000	10010
miss	miss	hit	hit	miss	miss	hit	miss
110	010	110	010	000	011	000	010

cache
10110
11010

No tag 10 in cache

miss

No tag 11 in cache

miss

Have tag 10 in cache and slot is correct

hit

Example 3B: Hit and Miss

8 word ➔ need 3 bits for slots

0010 0101 ➔
101 = slots
00100 = tag

- Consider an 8-slot direct mapped cache. You are g mory content and the current content of the cache. Show he cache as it responds to the series of requests. ***Note: the re hexadecimal.*

Part of main memory

22	AA
23	BB
24	CC
25	DD
26	EE
27	FF
28	GG

Current cache content

Slot	Tag	Content
0		
1		
2	04	33
3	05	23
4	05	33
5		
6	04	45
7		

Request (Hexadecimal address)	n inary	Hit/Miss ?
25	0010 0101	Miss
26	0010 0110	
22	0010 0010	Hit
24	0010 0100	
27		
23	0010 0011	
24	0010 0100	
23		
28		

Example 3B: Hit and Miss

First request is 25 → tag 00100 (4)	Yes there is → check slot 101 (5)	Slot don't match → miss → fetch
Then request is 26 → tag 00100 (4)	Yes there is → check slot 110 (6)	Slot match → HIT

Part of main memory

22	AA
23	BB
24	CC
25	DD
26	EE
27	FF
28	GG
33	HH
34	JJ

Current cache content

Slot	Tag	Content
0		
1		
2	04	AA
3	06	JJ
4	06	HH
5	04	DD
6	04	EE
7		

Request (Hexadecimal address)	In Binary	Hit/Miss ?
25	0010 0101	Miss
26	0010 0110	HIT
22	0010 0010	Hit
24	0010 0100	
34		
23	0010 0011	
24	0010 0100	
23		
28		

Example 3B: Hit and Miss

Then request is 22 → tag 00100 (4)

Yes there is → check slot 010 (2)

Slot match → HIT

Then request is 24 → tag 00100 (4)

Yes there is → check slot 100 (4)

Slot don't match → miss → fetch

Part of main memory

22	AA
23	BB
24	CC
25	DD
26	EE
27	FF
28	GG
33	HH
34	JJ

Current cache content

Slot	Tag	Content
0		
1		
2	04	AA
3	06	JJ
4	04	CC
5	04	DD
6	04	EE
7		

Request (Hexadecimal address)	In Binary	Hit/Miss ?
25	0010 0101	Miss
26	0010 0110	HIT
22	0010 0010	Hit
24	0010 0100	MISS
34		
23	0010 0011	
24	0010 0100	
23		
28		

Example 3B: Hit and Miss

Then request is 34 → tag 00110 (6)

Yes there is → check slot 100 (4)

Slot don't match → miss → fetch

Then request is 23 → tag 00100 (4)

Yes there is → check slot 011 (3)

Slot don't match → miss → fetch

Part of main memory

22	AA
23	BB
24	CC
25	DD
26	EE
27	FF
28	GG
33	HH
34	JJ

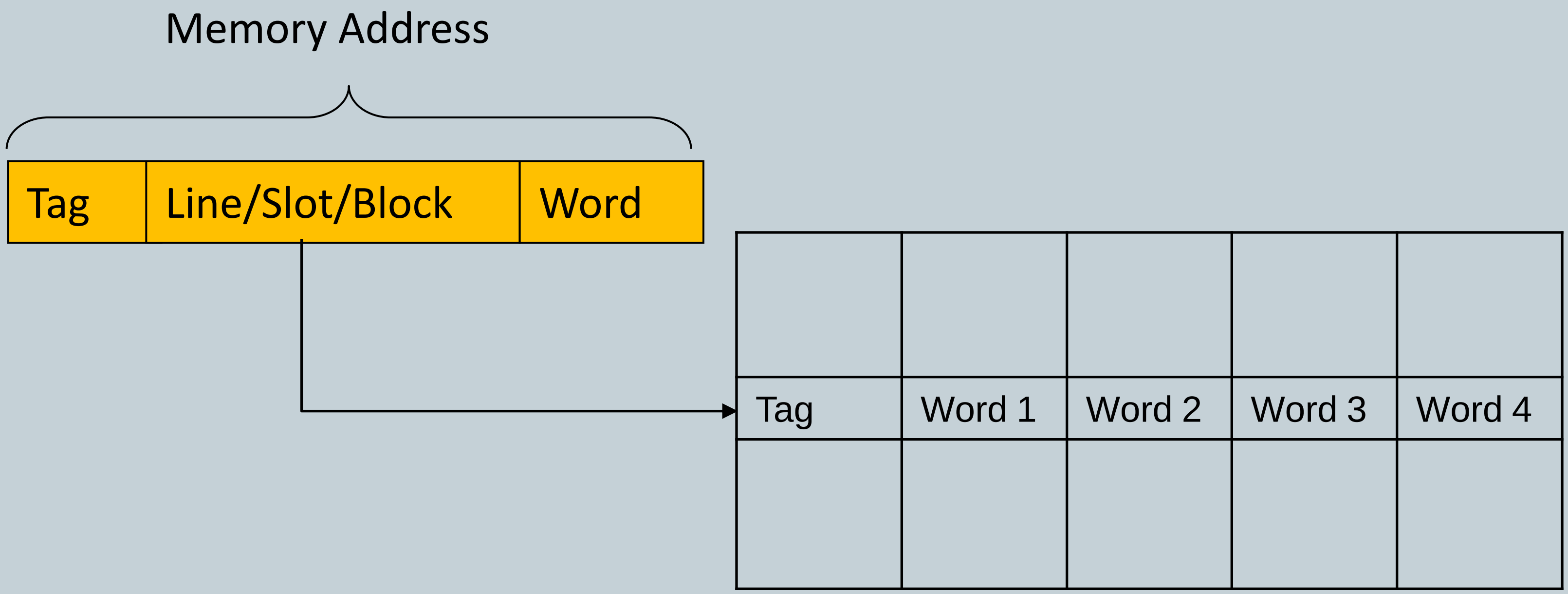
Current cache content

Slot	Tag	Content
0		
1		
2	04	AA
3	04	BB
4	06	JJ
5	04	DD
6	04	EE
7		

Request (Hexadecimal address)	In Binary	Hit/Miss ?
25	0010 0101	Miss
26	0010 0110	HIT
22	0010 0010	Hit
24	0010 0100	MISS
34	00110100	MISS
23	0010 0011	MISS
24	0010 0100	
23		
28		

P/S :please finish the rest

Block Direct Mapping



Direct Mapped cache

When a block of main memory is copied to cache, this tag is stored with the block and uniquely identifies this block.

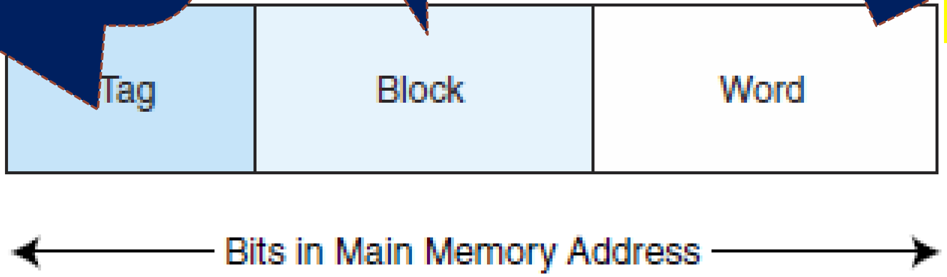
Lets call this T

The *block field selects a* unique block of cache

Lets call this B

The *word field uniquely identifies* a word from a specific block

Lets call this W



Direct Mapped cache : example

- Example: Assume main memory consists of 2^{14} words, cache has 16 blocks, and each block [illegible] rds.

8 words per block

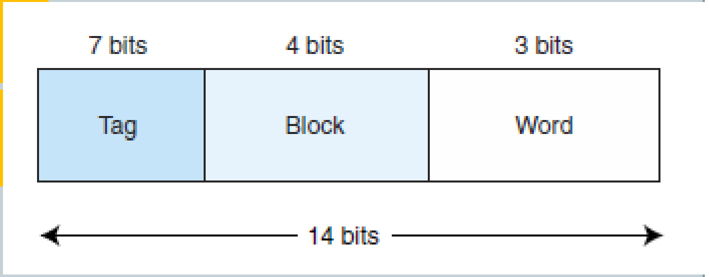
Main memory uses 14 bits to address ea ory.

Main memory has 2^{11} blocks ➔ $2^{14}/2^{3} = 2^{11}$

To identify each of the 8 words in a block, we need 3 bits ($\rightarrow 2^{3} = 8$) ➔ W = 3bits

To identify each of the 16 blocks in cache, we need 4 bits ($\rightarrow 2^{4} = 16$) ➔ B = 4bits

So, T = 14 – W - B = 7 bits



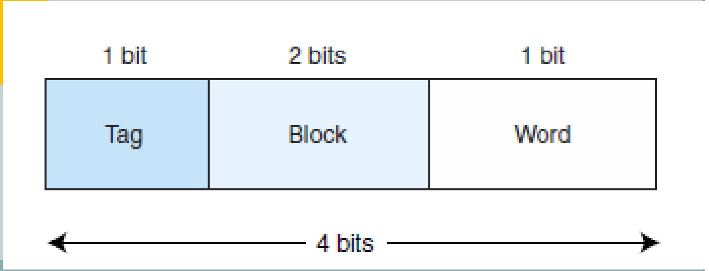
- Suppose we have a system using direct mapping with 16 words of main memory divided into 8 blocks (so each block has 2 words). Lets assume the cache is 4 blocks in size (for a total of 8 words).

Main memory uses 4 bits ($\rightarrow 2^4 = 16$)to address each memory.

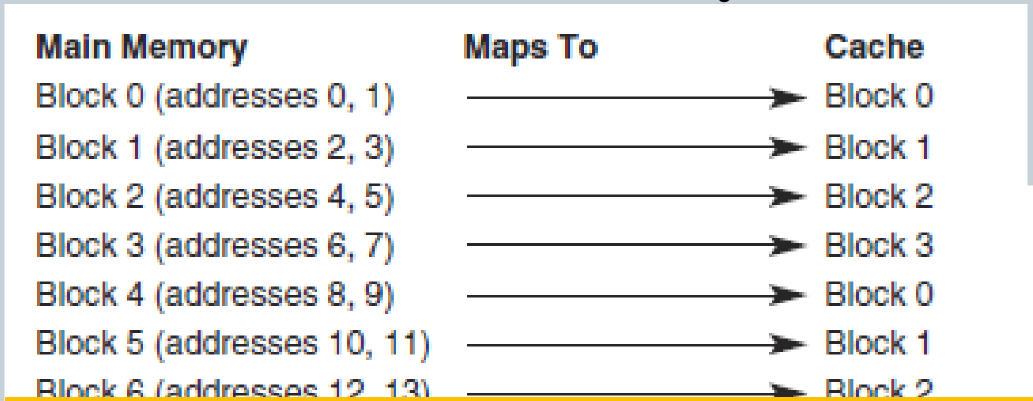
To identify each of the 2 words in a block, we need 1 bit ($\rightarrow 2^1 = 2$) ➔ W = 1 bit

To identify each of the 4 blocks in cache, we need 2 bits ($\rightarrow 2^2 = 4$) ➔ B = 2 bits

So, T = 4 – W - B = 1 bit



- So the main memory address is divided into

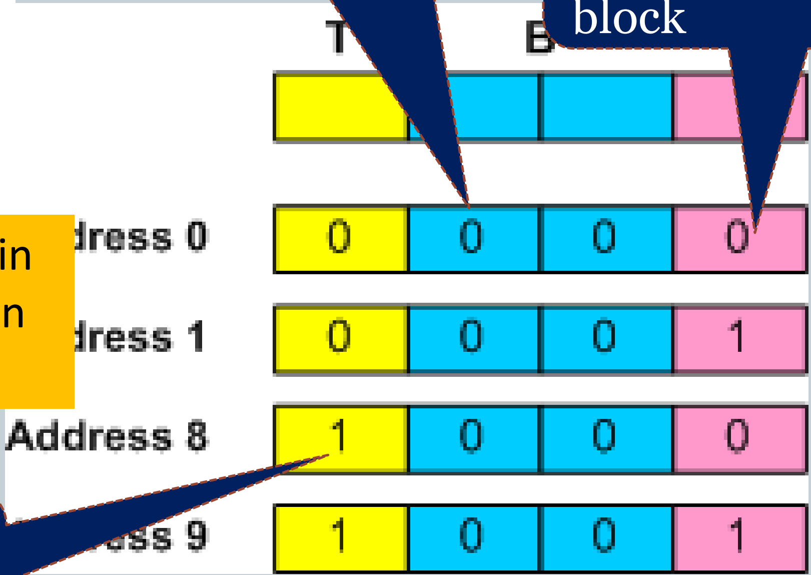


Main Memory	Maps To	Cache
Block 0 (addresses 0, 1)	⟶	Block 0
Block 1 (addresses 2, 3)	⟶	Block 1
Block 2 (addresses 4, 5)	⟶	Block 2
Block 3 (addresses 6, 7)	⟶	Block 3
Block 4 (addresses 8, 9)	⟶	Block 0
Block 5 (addresses 10, 11)	⟶	Block 1
Block 6 (addresses 12, 13)	⟶	Block 2

If the cache tag is 1, then block 4 currently resides in cache block 0. If the tag is 0, then block 0 from main memory is located in block 0 of cache.

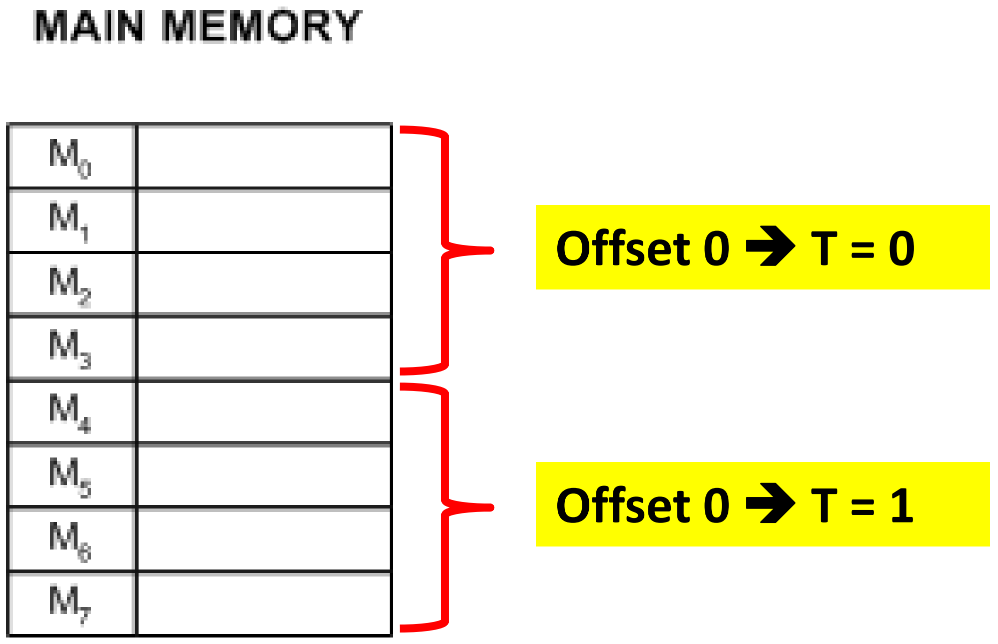
Resides in block 0 in cache

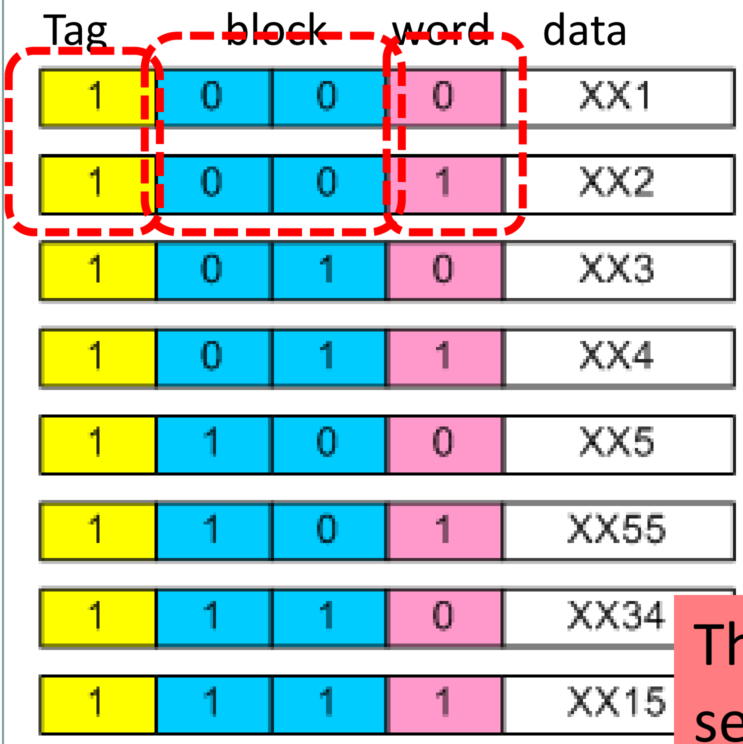
The first word in the block



Tag identifies that this is a different block in main memory ➔ like an offset

Analogy





Tag	block		word	data
1	0	0	0	XX1
1	0	0	1	XX2
1	0	1	0	XX3
1	0	1	1	XX4
1	1	0	0	XX5
1	1	0	1	XX55
1	1	1	0	XX34
1	1	1	1	XX15

CPU generates main memory address 9 (1001b).

First – looks at the block field bits 00. This indicates that cache block 0

If the cache block is valid, it then compares the tag field value of 1 (in the main memory address) to the tag

The tag matches , which means

Then the word field value of 1 is used to select one of the two words residing in the block.

➔ Because the bit is 1, we select the word with offset 1, which results in retrieving the data copied from main memory address 9.

Example 5: Block Direct Mapping

Tag (4 bits)	Line/Slot/Block (8 bits)	Word (3 bits)

Given the main memory address format as above.
If each main memory word is 8 bit, calculate:

i. The main memory capacity.
ii. Total cache words
iii. The size of cache words

… Example 5:

Solution:

Total main memory address bit = 4 + 8 + 3 = 15 bits

Total main memory words = 2^{15} = 32K words

Main memory capacity = 32K × 8 bits = 256 Kbits

Line/Slot/Block = 8 bits

Total cache words = 2^{8} = 256 words.

Total word = 2^{3} = 8 words

Cache word size = Tag + (No. of words in cache × size)

= 4 + (8 × 8) = 72bits

Example 6: Interpreting Memory Address

Tag (5 bits)	Line/Slot/Block (7 bits)	Word (4 bits)

Suppose we want to read or write a word at the address 357A, whose 16 bits are 0011010101111010_2.

This translates to Tag = 00110_2 = 6, Line = 1010111_2 = 87, and Word = 1010_2 = 10.

If line 87 in the cache has the same tag (6), then memory address 357A is in the cache. Otherwise, a miss has occurred and the contents of cache line 87 must be replaced by the memory line 001101010111 = 855 before the read or write is executed.

Example 7

- A block direct mapping cache has lines/slot that contains 4 bytes of data. The cache size is 16K lines and the main memory capacity is 16Mbytes.
- How many bits are used for main memory address?

  Main memory uses 24 bits ($\rightarrow 2^4 \times 2^{20} = 24$)to address each memory.
- How many lines/slots (cache address) in cache?

  16K $\rightarrow 2^4 \times 2^{10} = 2^{14} = 16384$ lines/slots
- Draw the format for main memory address by specifying the size of tag, line/slot and word.

  To identify each of the 4 bytes of data, we need 2 bits $\rightarrow$ W = 2 bits
  To identify each of the 16K of cache, we need 14 bits $\rightarrow$ B = 14 bits
  So, Tag = 24 – W - B = 8 bit

Tag	Line/slots	Word
8 bits	14 bits	2 bits

Example 7

- Draw the format for main memory address by specifying the size of tag, line/slot and word.
- A data is located at the 2nd word in the block of the main memory. If the data is at line/slot 3A5B in the cache, what is the main memory address for that data? Assume the tag is AA.

AA ➔ 1010 1010

The 2nd word
➔ 00, **01**, 10, 11

3	A	5	B
11	1010	0101	1011

Tag (8 bits)	Line/slots (14 bits)	Word (2 bits)
1010 1010	11 1010 0101 1011	01

THE MAIN MEMORY ADDRESS IS : 101010101110100101101101

Direct Mapping: Pros & Cons

- The direct mapping technique is simple and inexpensive to implement.
- Its main disadvantage is that there is a fixed cache location for any given block.
  - Thus, if a program happens to reference words repeatedly from two different blocks that map into the same line, then the blocks will be continually swapped in the cache, and the hit ratio will be low (a phenomenon known as *thrashing*).
- Least effective in its cache utilization
  - it may leave some cache lines unused.

Replacement Algorithm : Direct Mapping

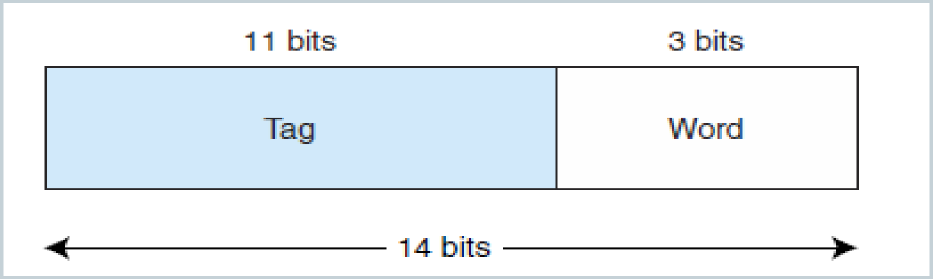
- With direct mapping, if a block already occupies the cache location where a new block must be placed, the block currently in cache is removed
  - it is written back to main memory if it has been modified
  - or simply overwritten if it has not been changed
- There is no choice as each line/slot or block only maps to one line/slot or block

Associative Mapping

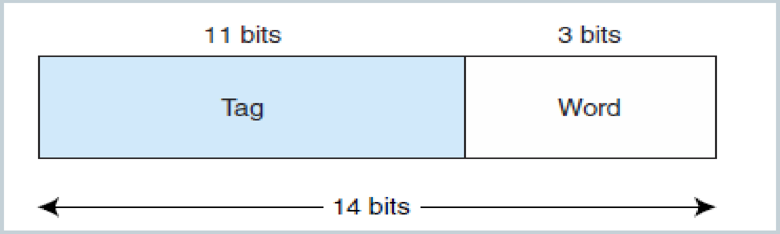
- This scheme allows a main memory block to be placed anywhere in cache.
- The only way to find a block mapped this way is to search all of cache.
- *A single* search must compare the requested tag to *all tags in cache to determine whether* the desired data block is present in cache.
- Associative memory requires special hardware to allow associative searching, and is, thus, quite expensive.

Associative Mapping

- Using associative mapping, the main memory address is partitioned into two pieces, the tag and the word.
- For example, main memory configuration with 2^{14} words, a cache with 16 blocks, and blocks of 8 words.
  - The word field is still 3 bits, but now the tag field is 11 bits.



- This tag must be stored with each block in cache.
- When the cache is searched for a specific main memory block, the tag field of the main memory address is compared to all the valid tag fields in cache;
  - if a match is found, the block is found. (Remember, the tag uniquely identifies a main memory block.)
  - If there is no match, we have a cache miss and the block must be transferred from main memory.



Example 7: Associative Mapping

Main memory address

Tag (12 bits)	Word (4 bits)

- Tag field identifies one of the $2^{12} = 4096$ memory lines; all the cache tags are searched to find out whether or not the Tag field matches one of the cache tags.
- If so, we have a hit, and if not there's a miss and we need to replace one of the cache lines by this line before reading or writing into the cache.

Associative Mapping

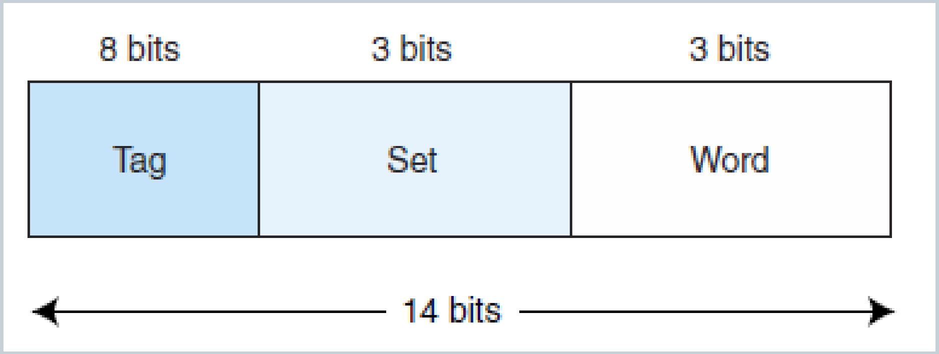
- With associative mapping, there is flexibility as to which block to replace when a new block is read into the cache.
- Replacement algorithms (discussed later), are designed to maximize the hit ratio.
- The principal disadvantage of associative mapping is the complex circuitry required to examine the tags of all cache lines in parallel.

Set Associative Mapping

- The third mapping scheme is *N-way set associative cache mapping*, a combination of these two approaches.
- This scheme is similar to direct mapped cache, in that we use the address to map the block to a certain cache location.
  - The important difference is that instead of mapping to a single cache block, an address maps to a *set of several cache blocks*.
- *All sets in cache must be the* same size.
- If there are 2 sets, its called *2-way set associative cache mapping*

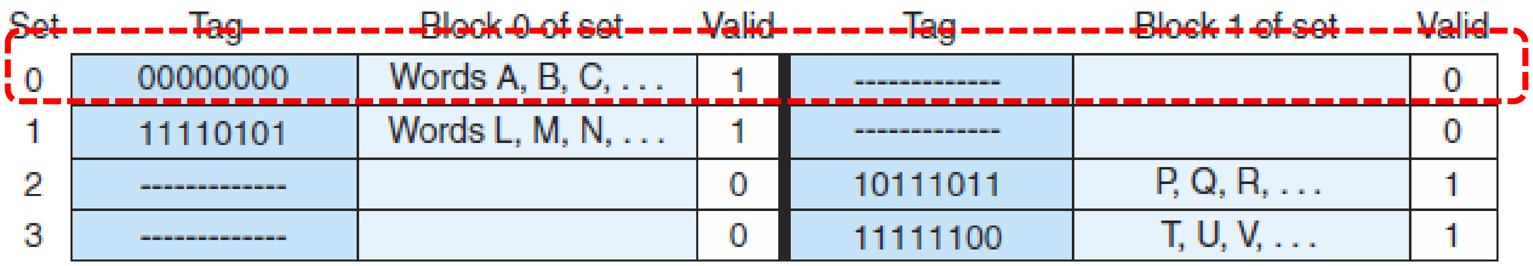
Set Associative Mapping

- For example: a main memory of 2^{14} words, a cache with 16 blocks, where each block contains 8 words.
- If cache consists of a total of 16 blocks, and each set has 2 blocks, then there are 8 sets in cache.
- Therefore, the set field is 3 bits, the word field is 3 bits, and the tag field is 8 bits.



a 2-way set associative cache

A set can be in 2 different blocks



Set	Tag	Block 0 of set	Valid	Tag	Block 1 of set	Valid
0	00000000	Words A, B, C, . . .	1	-------------		0
1	11110101	Words L, M, N, . . .	1	-------------		0
2	-------------		0	10111011	P, Q, R, . . .	1
3	-------------		0	11111100	T, U, V, . . .	1

Set 0 contains two blocks, one that is valid and holds the data A, B, C, . . . , and another that is not valid.

Given a tag, only search in the set :
If one of the 2 matches ➔ it's a hit.
Else ➔ it's a miss ➔ go fetch from main memory

In an 8-way set associative cache, there are 8 cache blocks per set

Direct mapped cache is a special case of *N-way set associative cache mapping* where the set size is one

Replacement Algorithms

- For the associative and set associative techniques, a replacement algorithm is needed.
- To achieve high speed, such an algorithm must be implemented in hardware.
- The most common four :
- Least Recently used (LRU)
  - Replace that block that has been in the cache longest with no reference to it.
- First in first out (FIFO)
  - replace block that has been in cache longest.
- Least frequently used (LFU)
  - replace block which has had fewest hits/references
- Random
  - pick a block at random from among the candidate blocks

Write Policy

- Must not overwrite a cache block unless main memory is up to date
  - Any changes made in cache MUST be updated in main memory before block is replaced.
- There are 2 problems
  - Multiple CPUs may have individual caches
    - So a change in one cache, may invalidate the others
  - I/O may address main memory directly
    - If I/O changes memory, cache may be invalid
    - If cache changes, a haven't update memory, then I/O may use invalid data

Write Through

- The simplest technique is called **write through.**
- Using this technique, all write operations are made to main memory **as well as** to the cache, ensuring that main memory is always valid.
- Any other processor–cache module can monitor traffic to main memory to maintain consistency within its own cache.
- The main disadvantage is that it generates substantial memory traffic and may create a bottleneck.

Write Back

- Updates initially made in cache only
- Update bit (or dirty bit) for cache slot is set when update occurs
- If block is to be replaced, write to main memory only if update bit is set
- Problem:
  - Other caches get out of sync
  - I/O must access main memory through cache