Computer Architecture

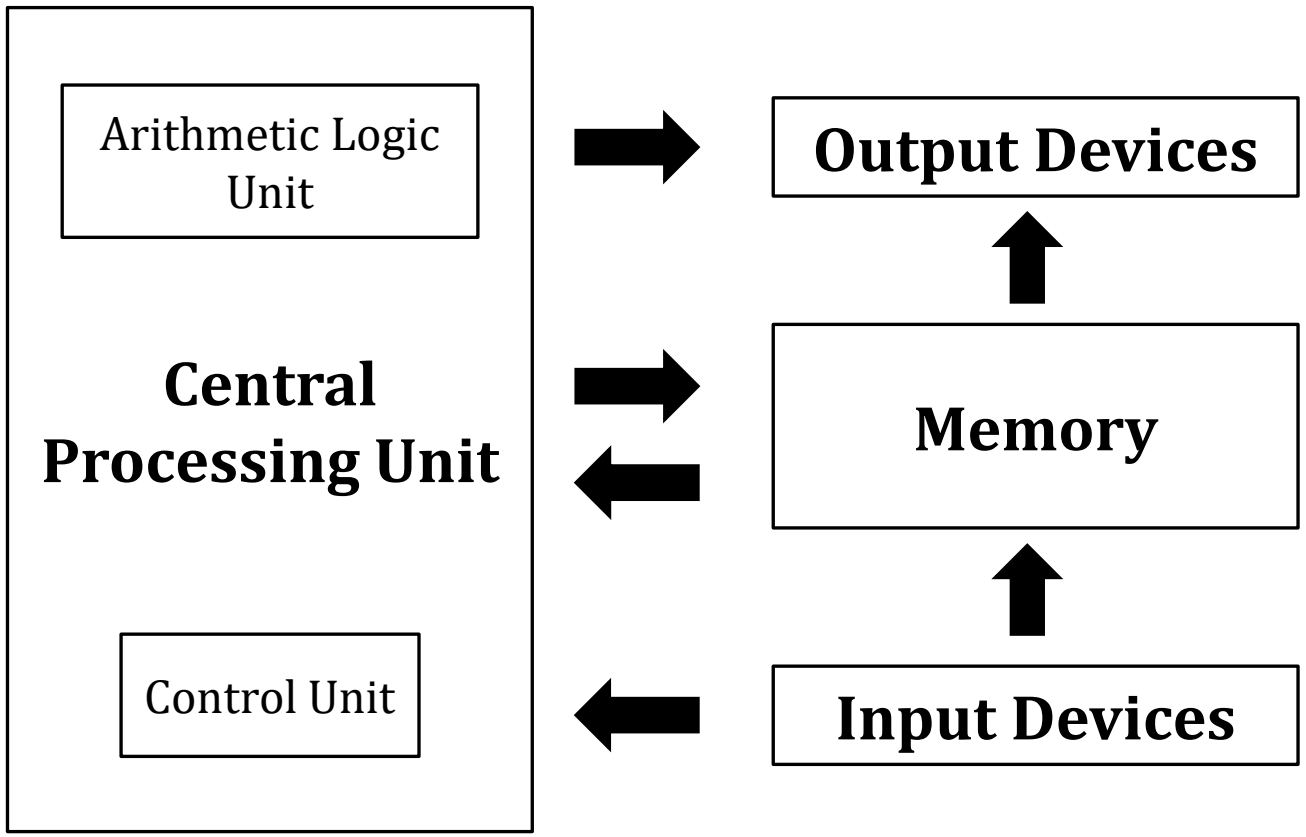
Angshuman Paul

Assistant Professor

Department of Computer Science & Engineering

The Memory

The Components of a Computer



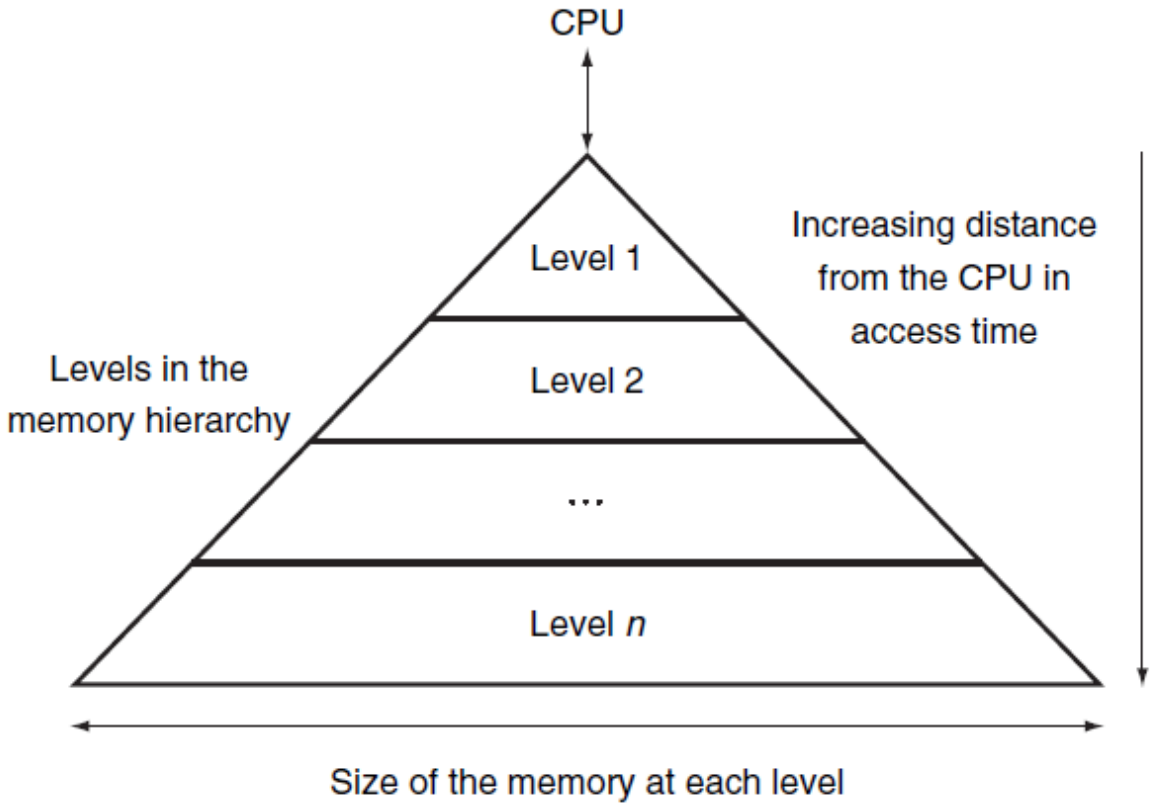
Memory Organization

- Memory: stores data
- Cost of memory
  - Speed
  - Size
- Multiple levels of memory
  - Different speed
  - Different size
  - Different cost

Memory Hierarchy

- Faster memory
  - More expensive
  - Smaller in size
  - Close to processor
- Slower memory
  - Less expensive
  - Larger in size
  - Farther away from processor

Memory Hierarchy



Memory Hierarchy

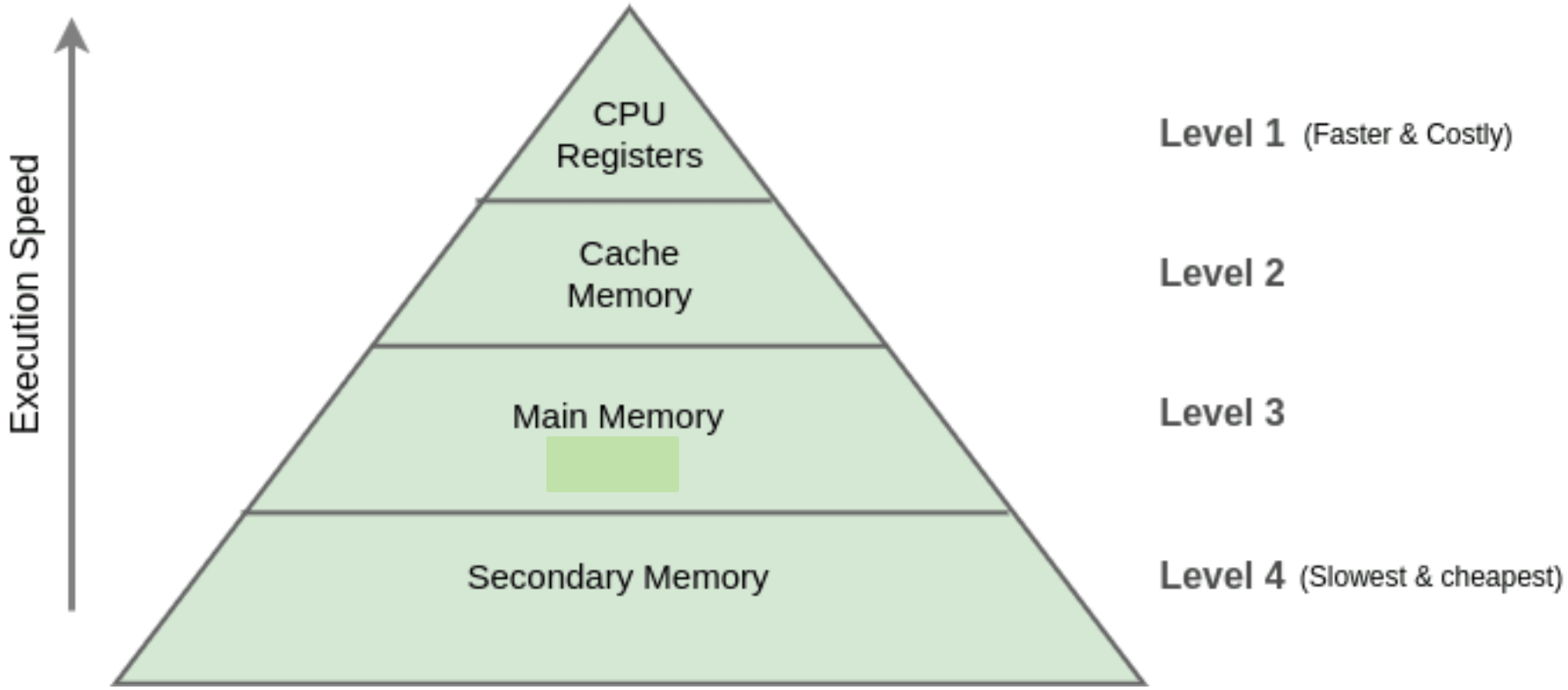
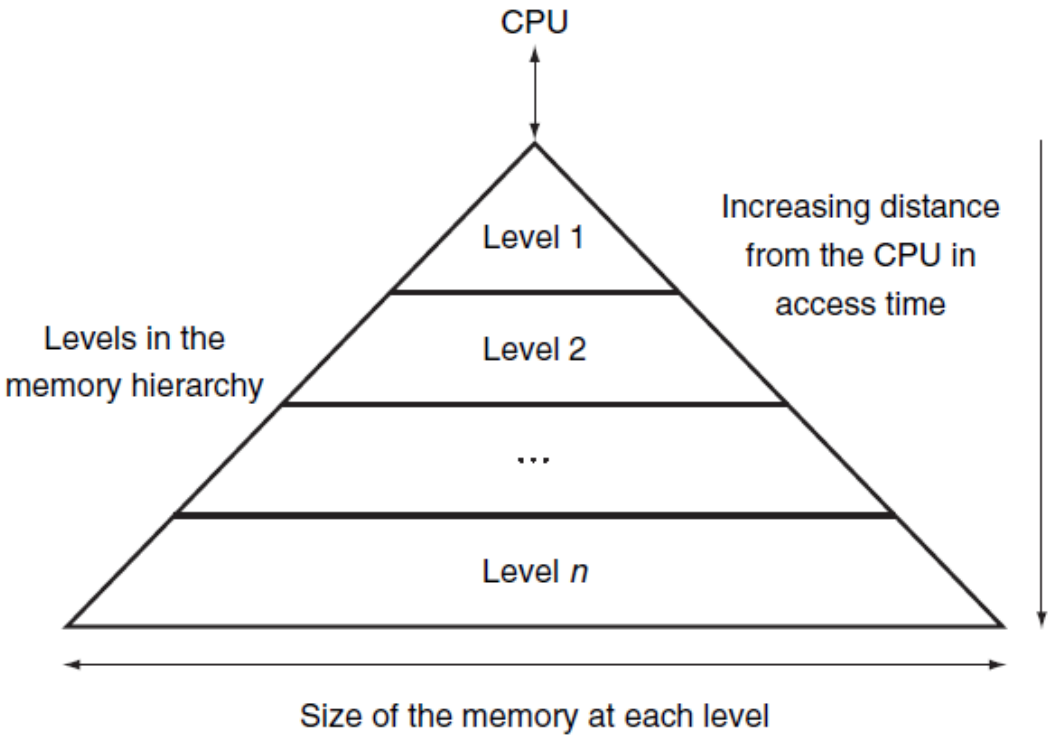


Image Source: https://nlogn.in/what-is-cpu-cache-memory-in-computer-architecture/

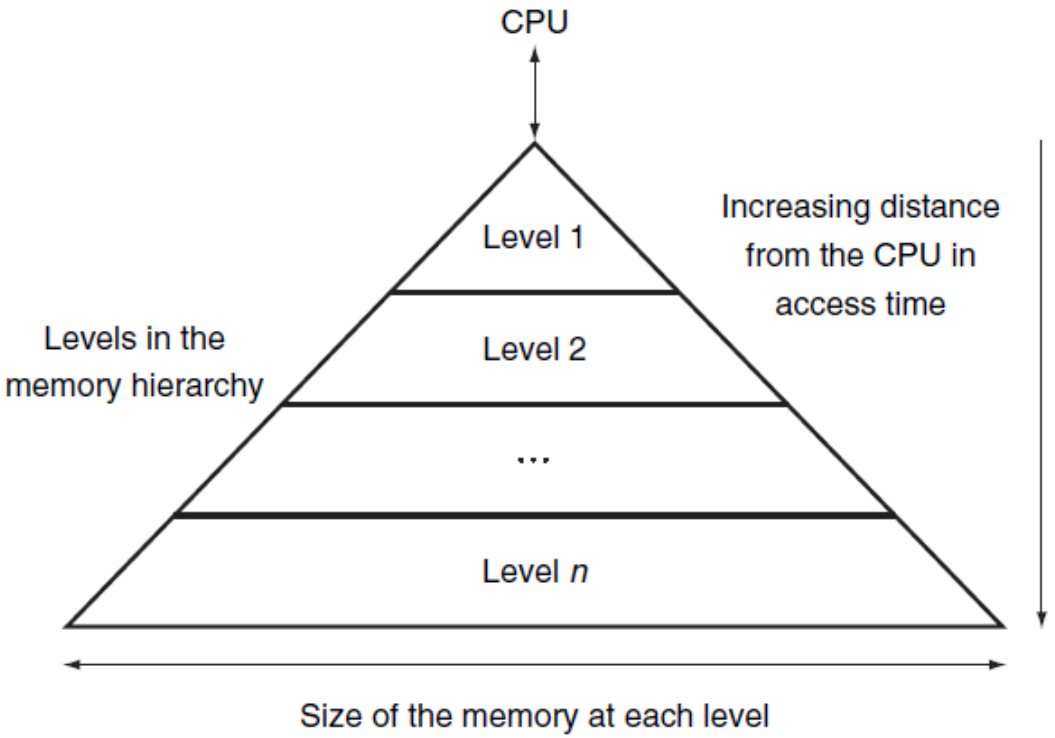
Memory Hierarchy

- Processor access data from the highest level
- Higher level keeps a subset of data from the lower level



Memory Technology

- Primary memory: DRAM
- Cache: SRAM



The Concept of Cache

- To resolve the problem of mismatch between processor and memory speeds
- Cache stores a copy of a subset of main memory
- Modern processors have separate caches for instructions and data
- Several levels of caches may be implemented in different sizes and technologies (e.g., processor vs. SRAM)

How Does Cache Memory Work?

- Cache memory
  - Intended for storing a frequently used subset of data from main memory
  - Faster than main memory
  - CPU looks in the cache before accessing the main memory
  - If found in cache
    - Cache hit
  - Else
    - Cache miss
  - If cache hit: significantly faster access to data
  - Goal: increase cache hit

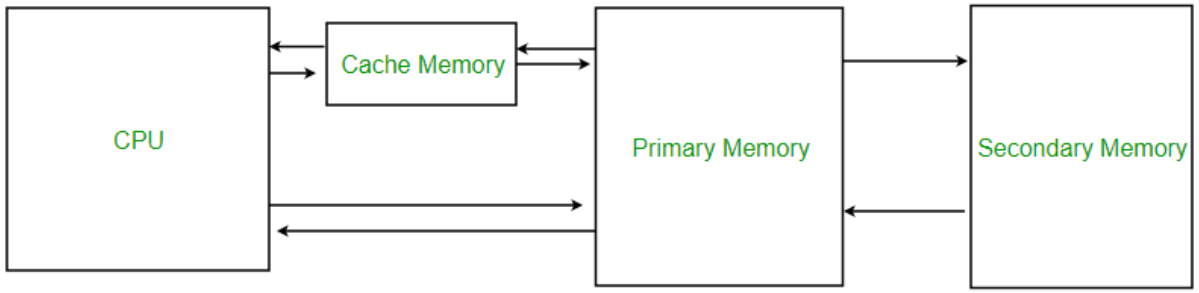


Image source: https://www.geeksforgeeks.org/cache-memory-in-computer-organization/

Why Does Cache Memory Work?

- **Principle of locality**

  Programs generally access a small portion of address space at any instant of time

Why Does Cache Memory Work?

- Temporal locality (locality in time): if an item is referenced, it will tend to be referenced again soon

- Spatial locality (locality in space): if an item is referenced, items whose addresses are close by will tend to be referenced soon.

Why Does Cache Memory Work?

- Temporal locality (locality in time): if an item is referenced, it will tend to be referenced again soon

- Spatial locality (locality in space): if an item is referenced, items whose addresses are close by will tend to be referenced soon.

How do these facts help in the context of cache memory?

Why Does Cache Memory Work?

- Temporal locality (locality in time): if an item is referenced, it will tend to be referenced again soon
  - Keep most recently accessed data items closer to the processor

- Spatial locality (locality in space): if an item is referenced, items whose addresses are close by will tend to be referenced soon
  - Move blocks consisting of contiguous words closer to the processor

Cache Terminology

- When reading cache memory, following things can happen:
- Cache hit
  - cache block is valid and contains proper address, so read desired word
- Cache miss (Case 1 – cache block is empty/invalid)
  - Nothing in the appropriate block of the cache, so fetch from memory
- Cache miss: (Case 2- block replacement)
  - Wrong data is in cache at appropriate block, so discard it and fetch desired data from memory (cache always copy)

Cache Terminology

- Hit - The data was found in the cache
- Miss - The data was not found in the cache
- Hit rate = hits/total accesses
- Miss rate = 1- Hit rate
- Cache line/ Cache block: the basic unit of data storage in a cache, generally several words.
- Tag: the high order address bits stored along with the data to identify the actual address of the cache line
- Hit time -- time to service a hit
- Miss time -- time to service a miss (this is a function of the lower level caches.)

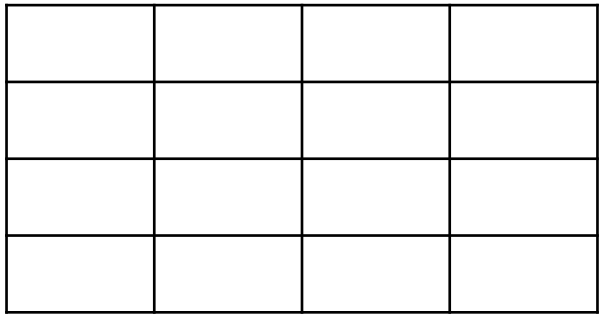
Main Memory

- Word: Largest addressable unit in memory
- Main memory is logically divided into blocks of several words

				Block Number	Binary Representation
w0	w1	w2	w3	0	0000
w4	w5	w6	w7	1	0001
w8	w9	w10	w11	2	0010
w12	w13	w14	w15	3	0011
w16	w17	w18	w19	4	0100
w20	w21	w22	w23	5	0101
w24	w25	w26	w27	6	0110
w28	w29	w30	w31	7	0111

Cache Memory

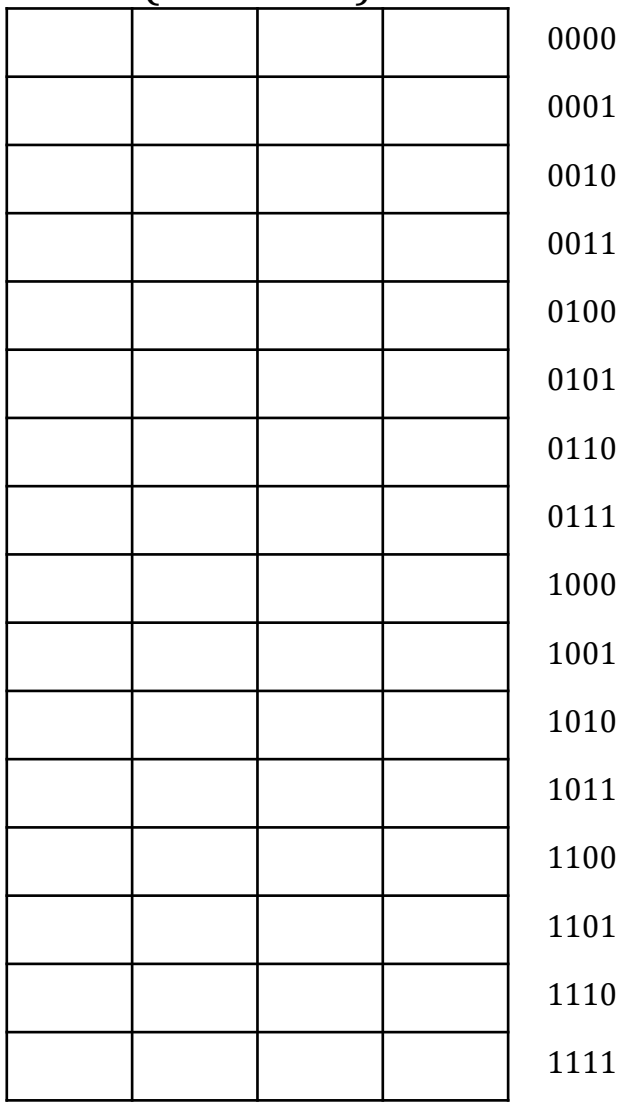
- Cache memory is also logically divided into blocks/ lines of several words
  - Cache line



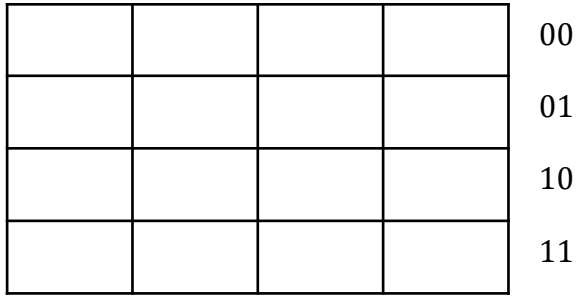
Line Number	Binary Representation
0	0000
1	0001
2	0010
3	0011

Main Memory and Cache

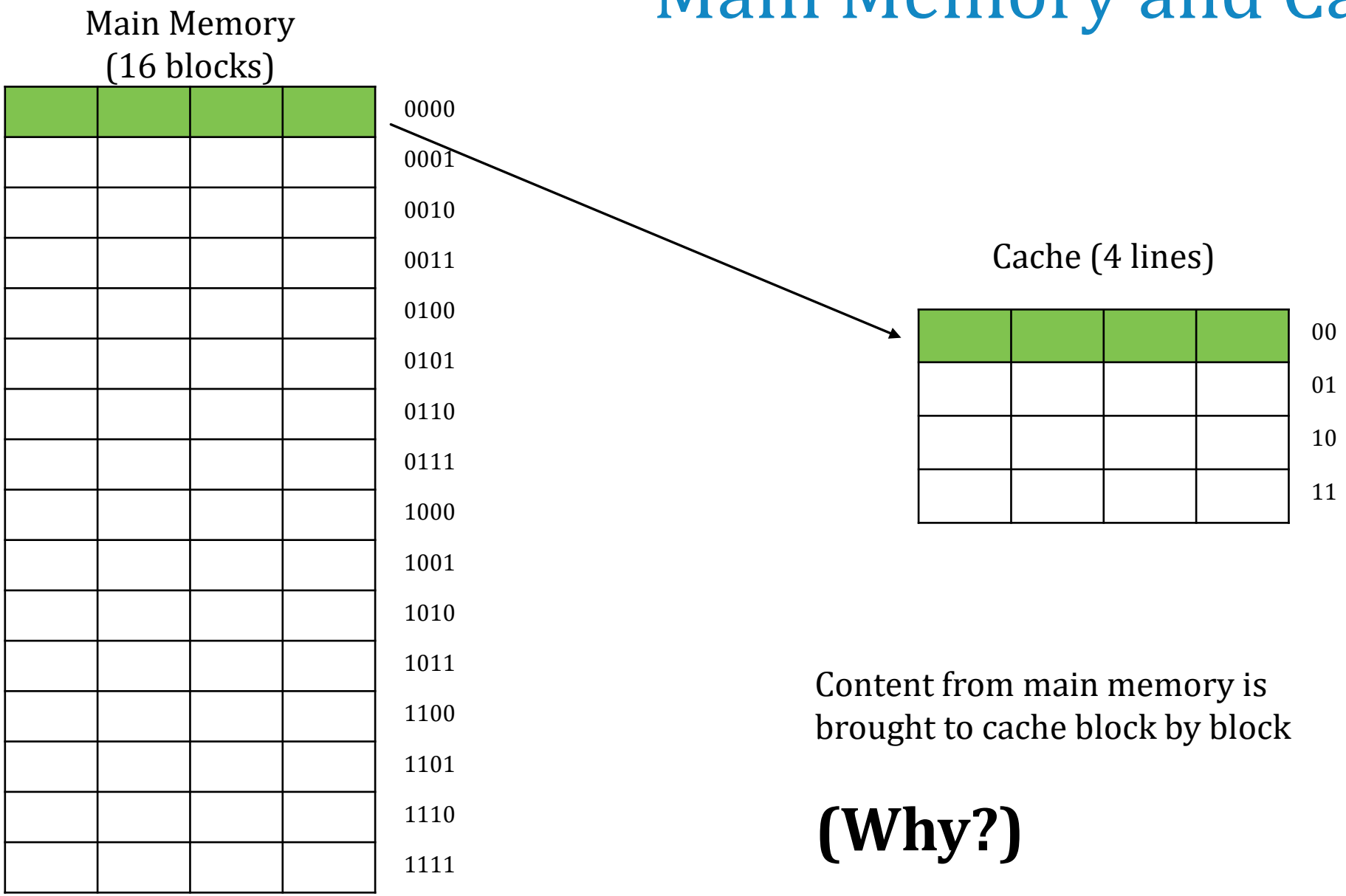
Main Memory
(16 blocks)



Cache (4 lines)



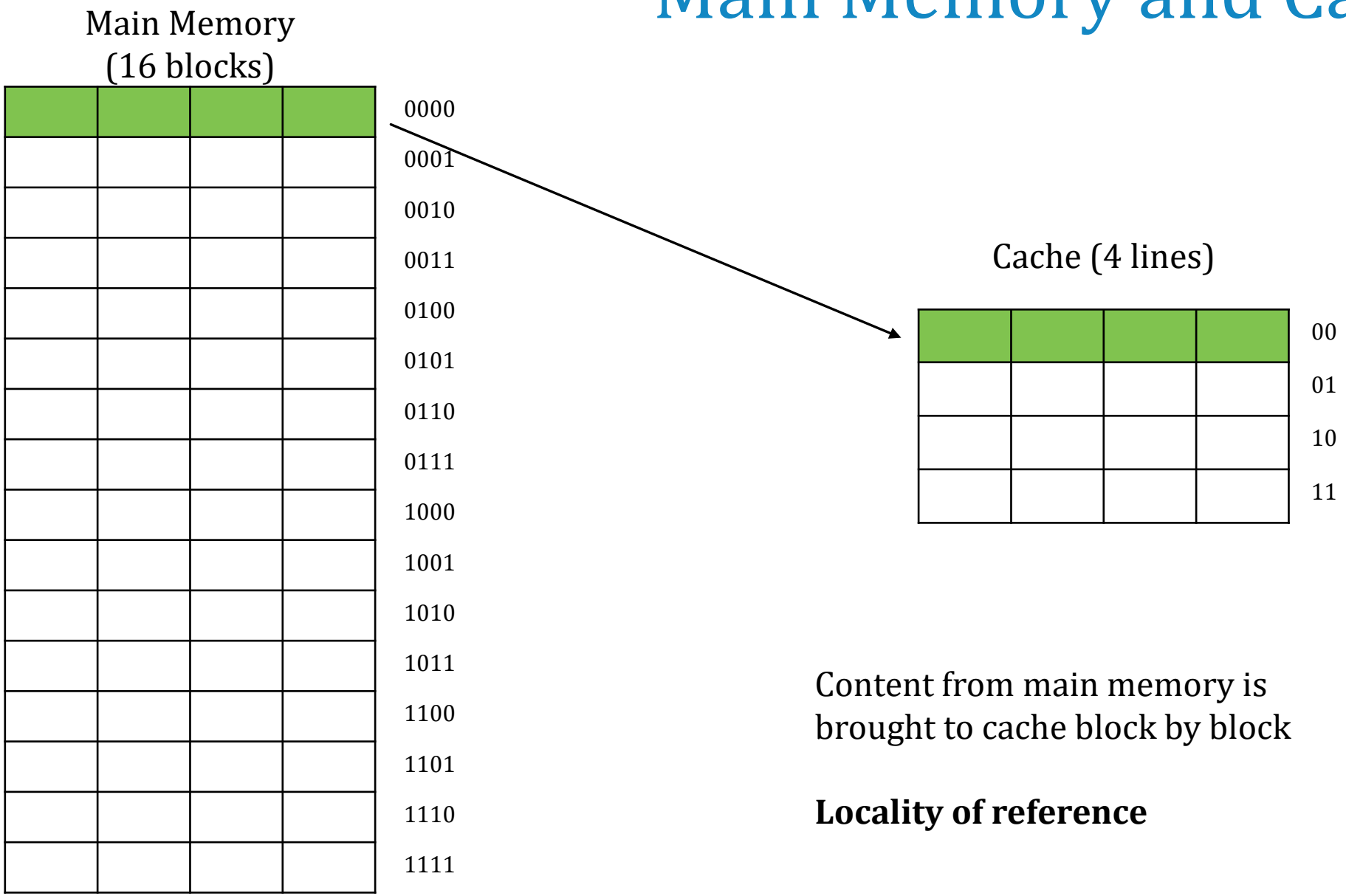
Main Memory and Cache



Content from main memory is brought to cache block by block

(Why?)

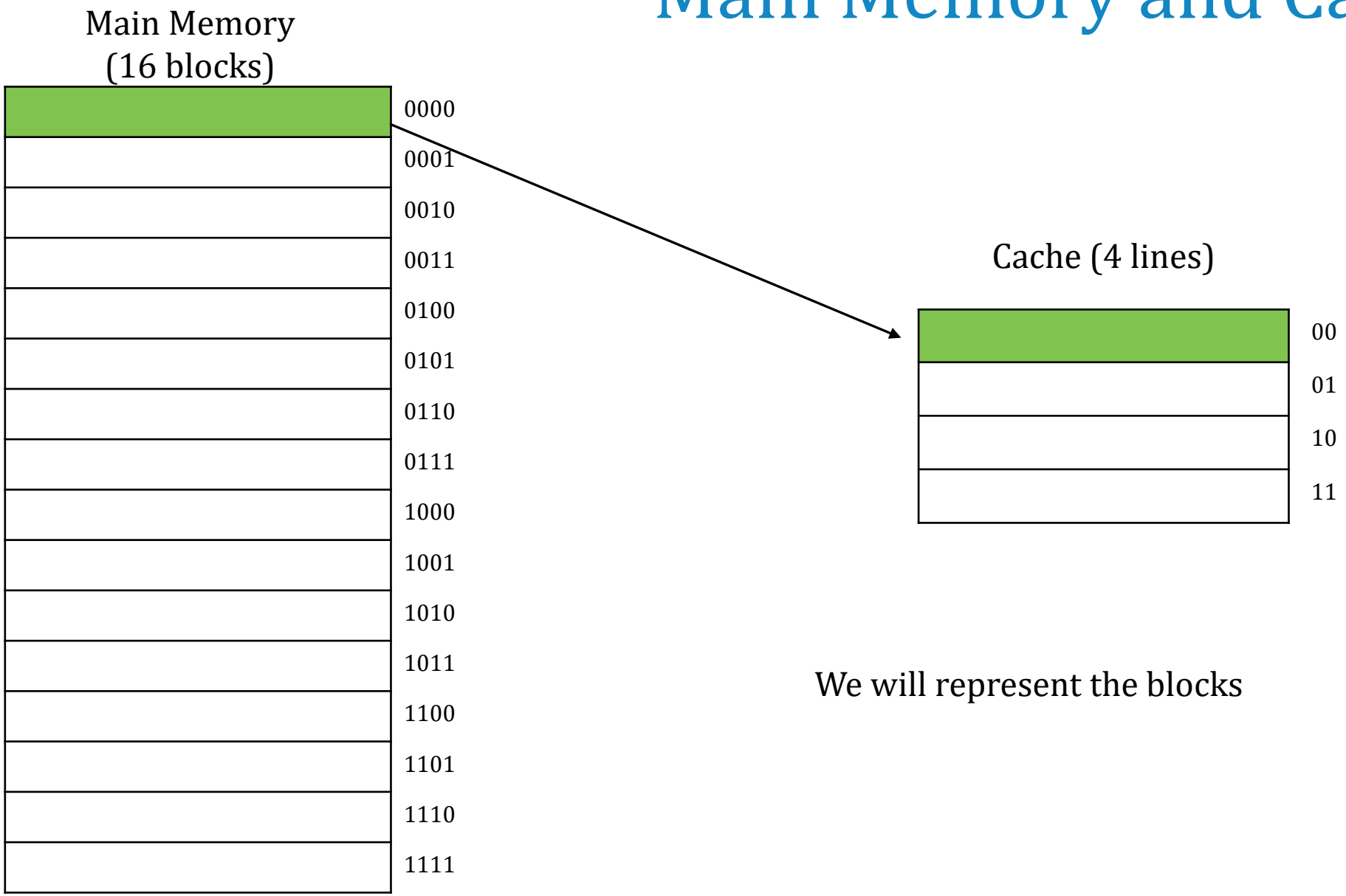
Main Memory and Cache



Content from main memory is brought to cache block by block

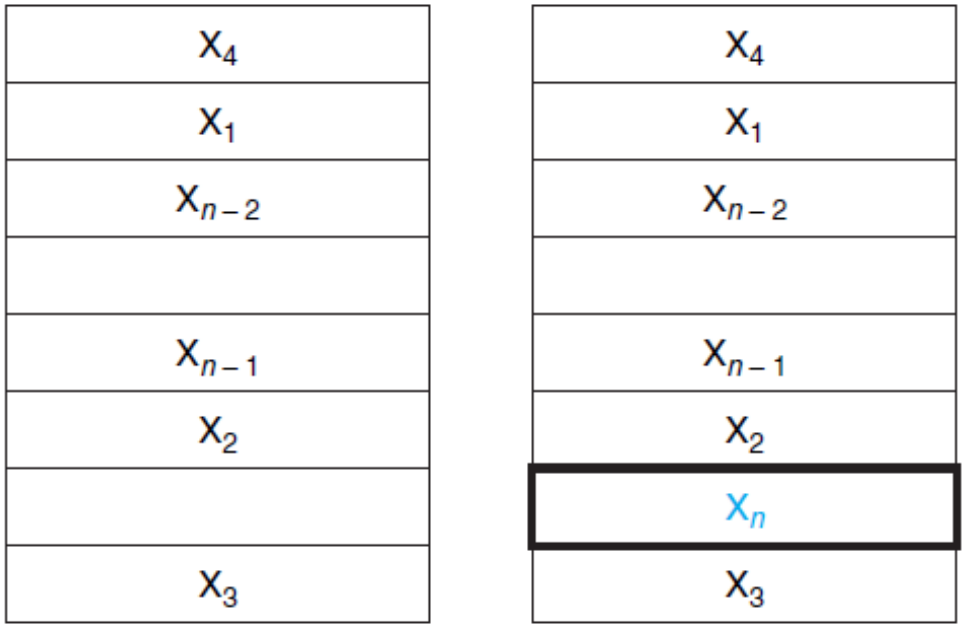
Locality of reference

Main Memory and Cache



We will represent the blocks

Implementation of a Simple Cache

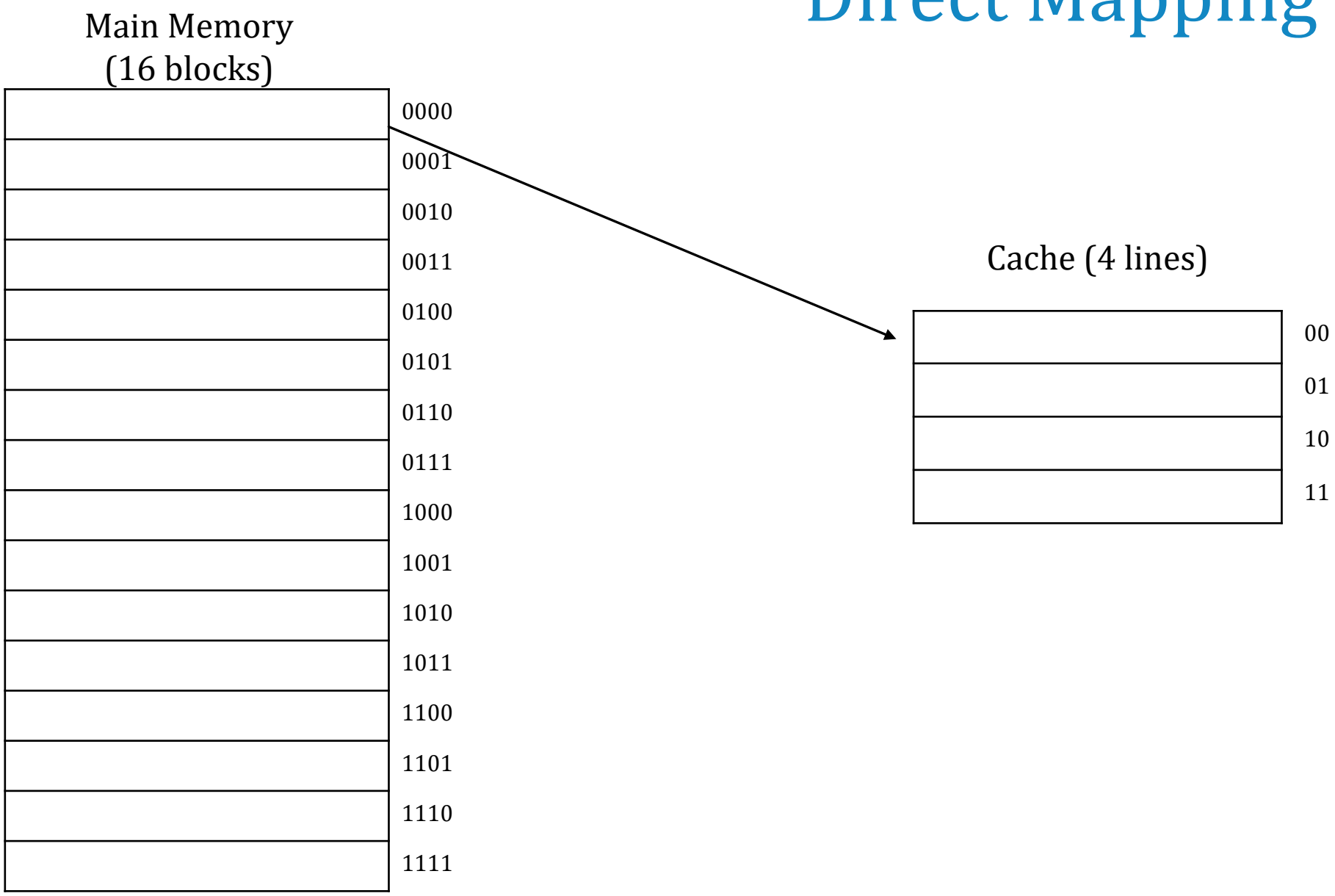


a. Before the reference to X_n b. After the reference to X_n

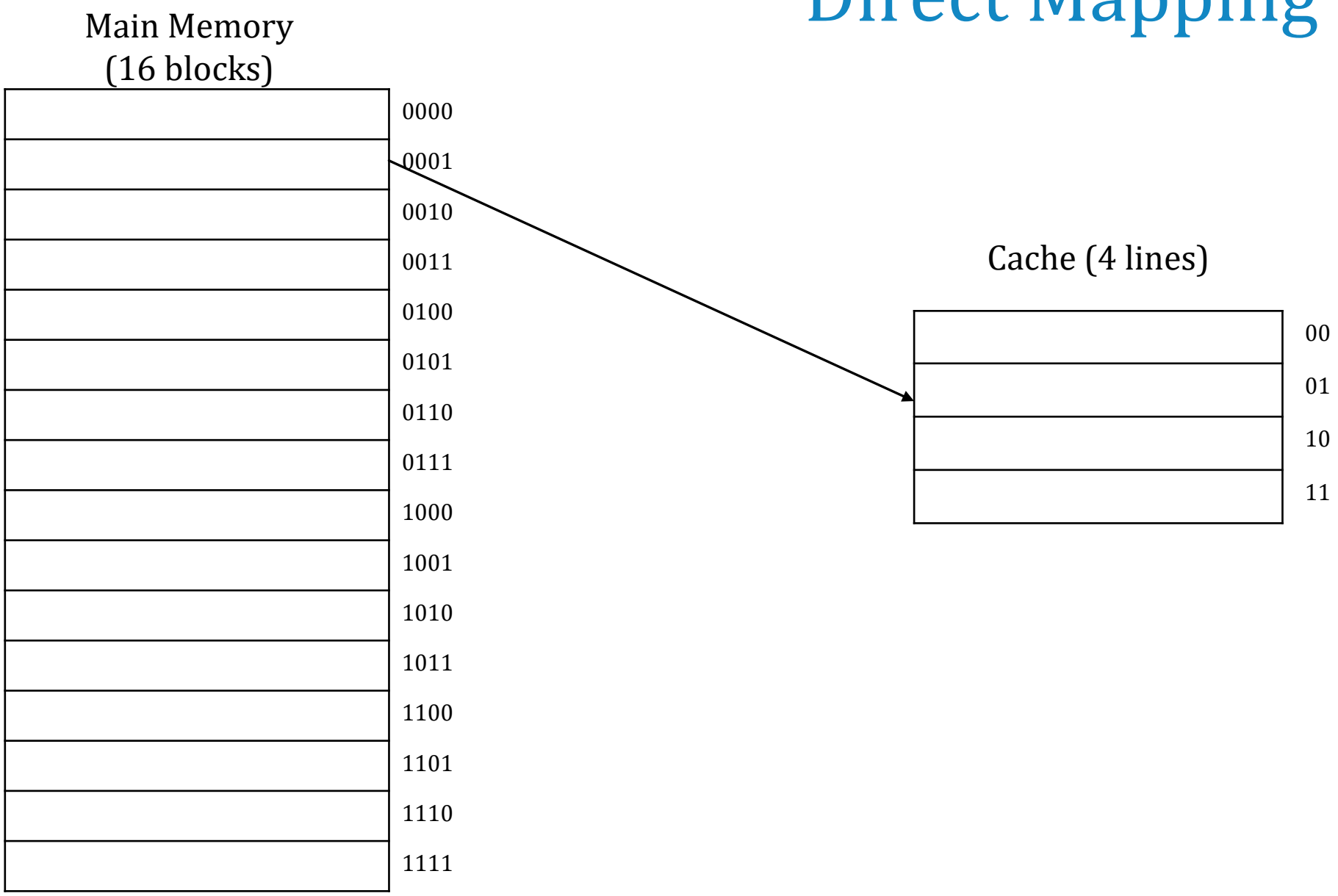
Direct Mapped Cache

- Each block in memory maps to exactly one block in cache

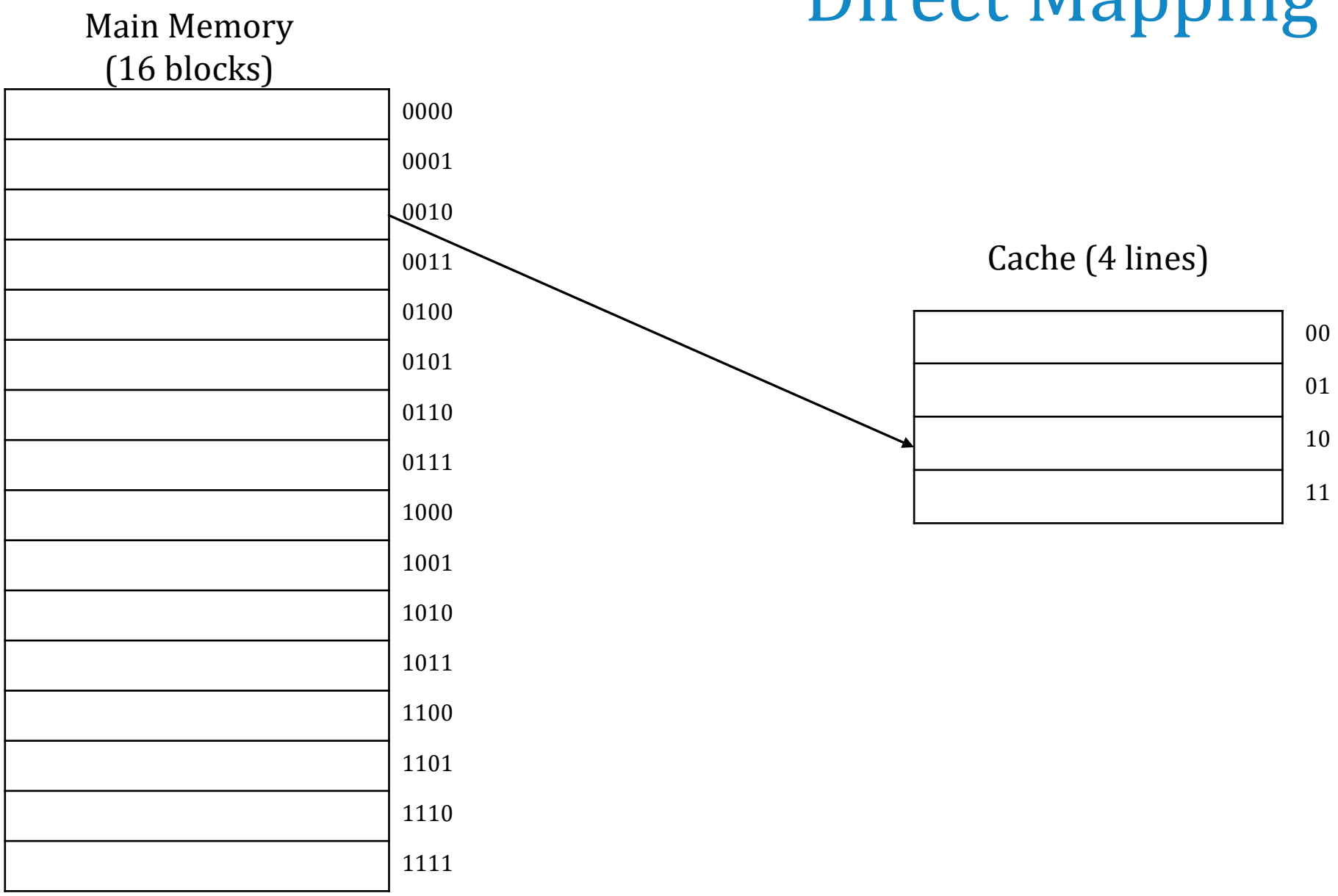
Direct Mapping



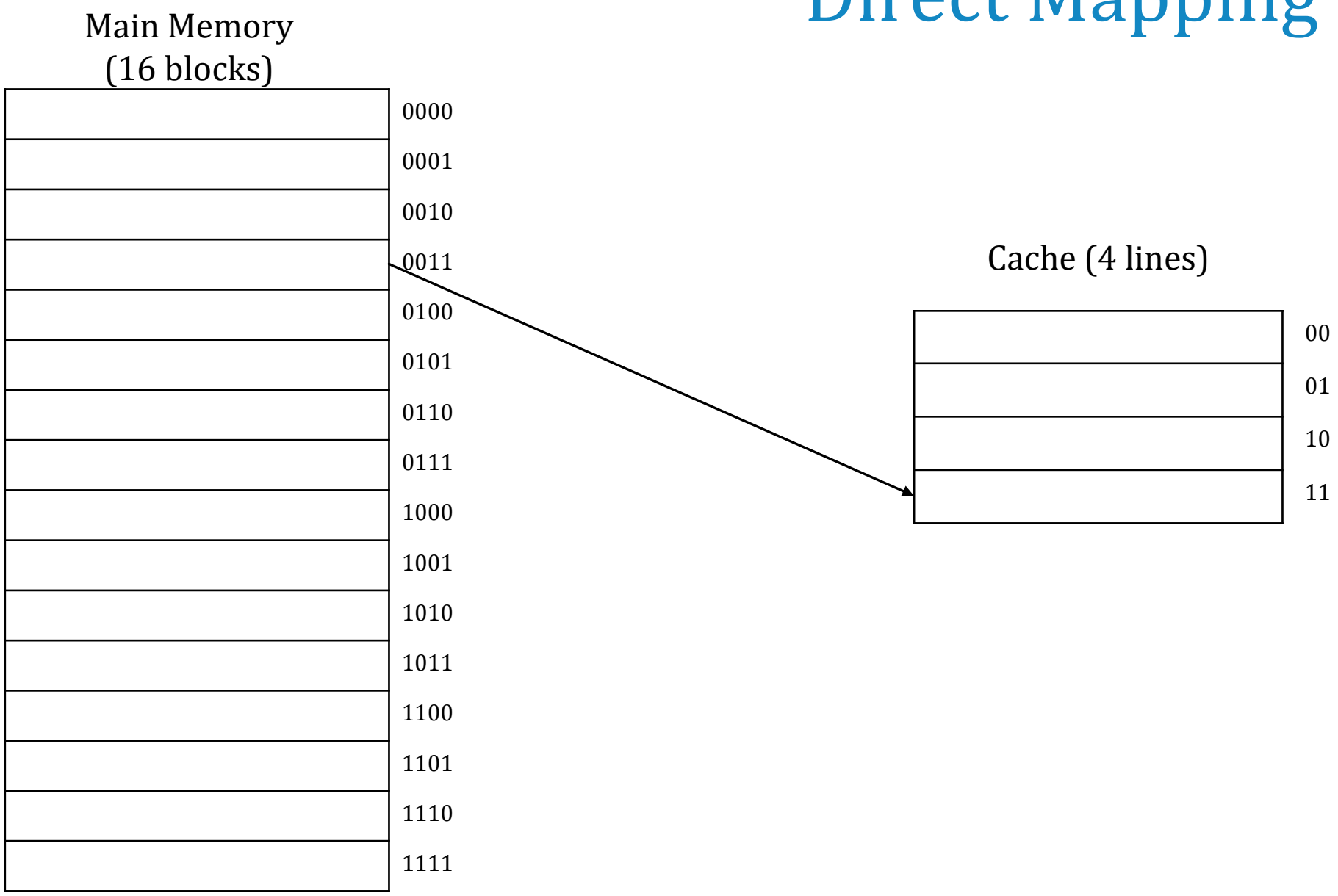
Direct Mapping



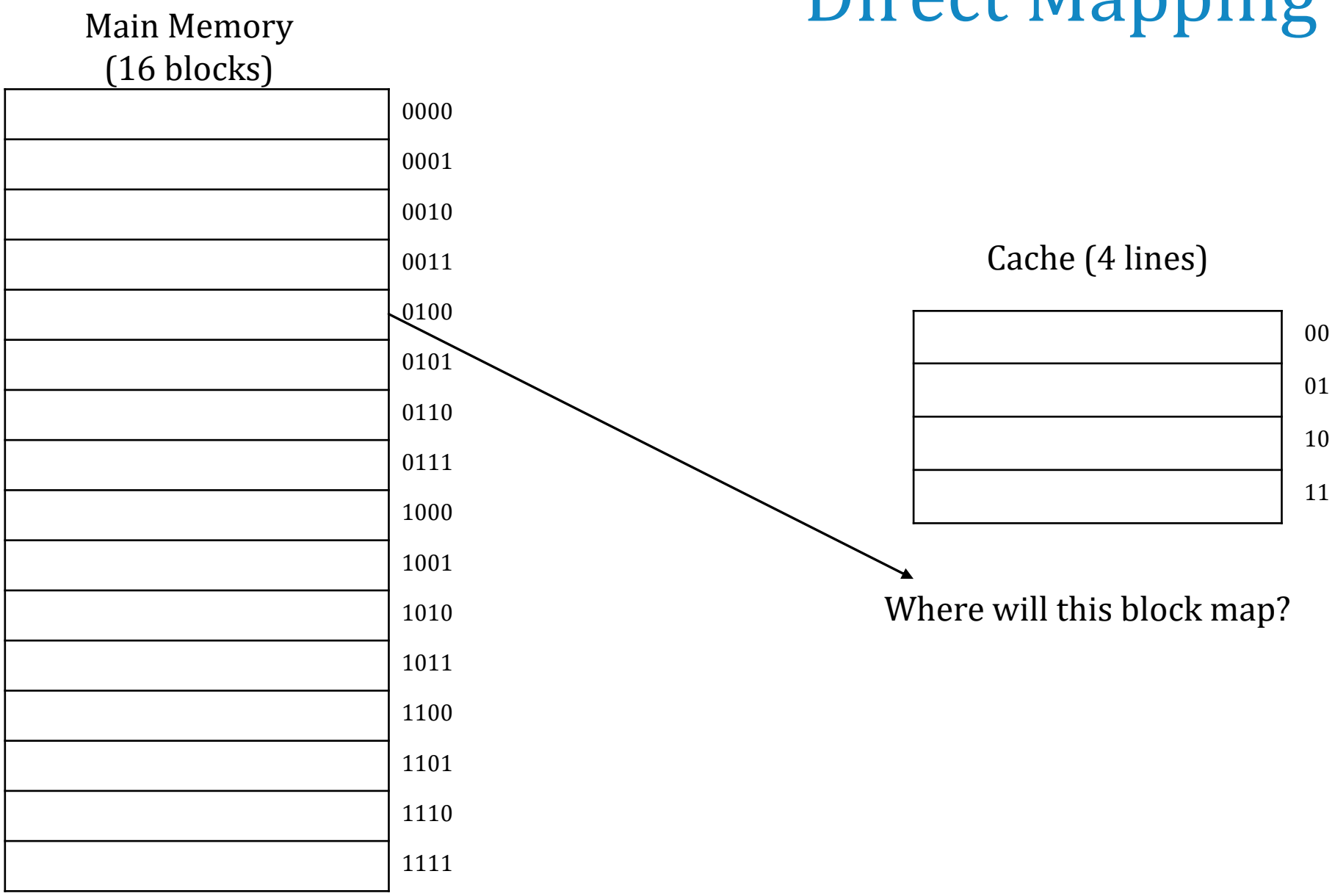
Direct Mapping



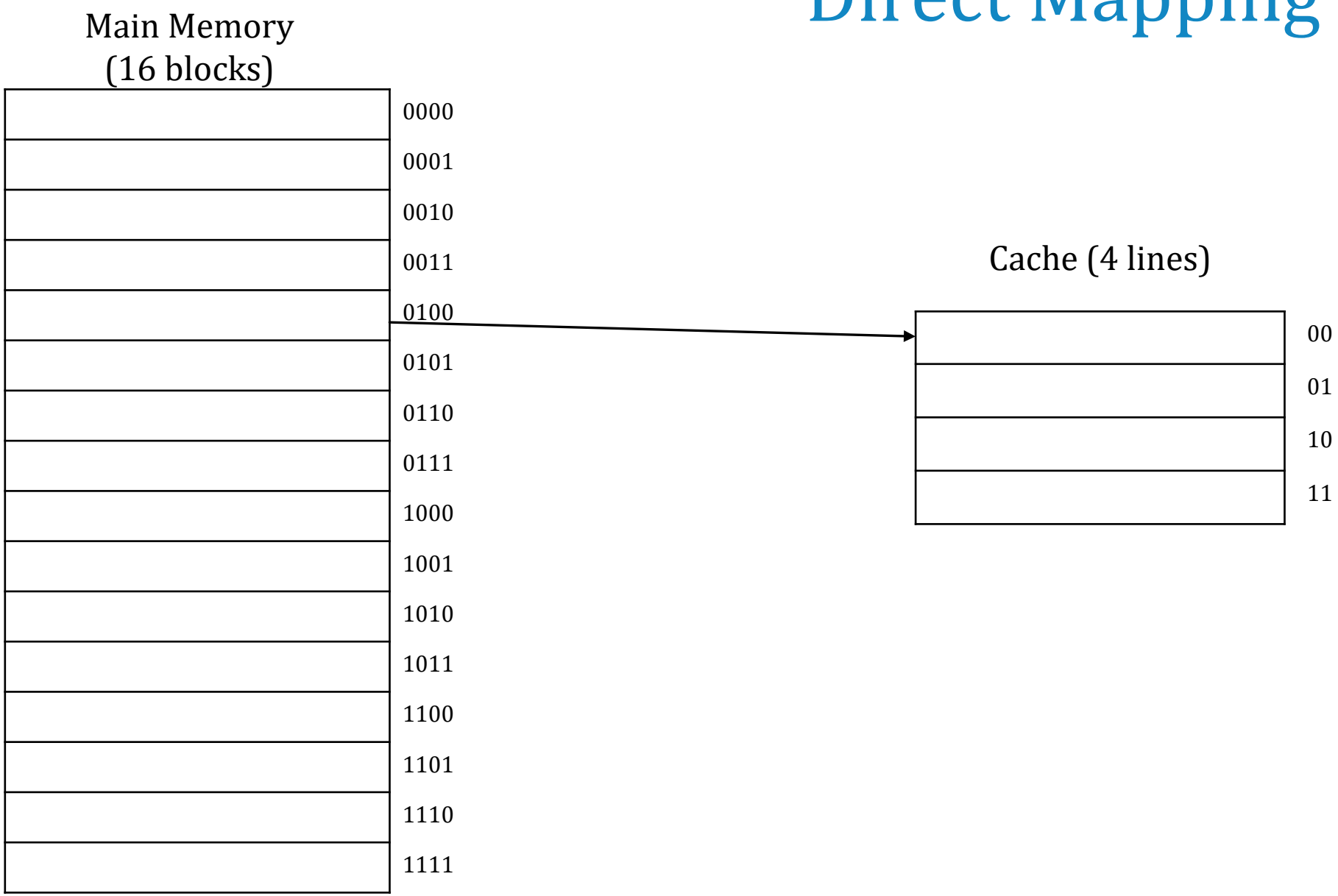
Direct Mapping



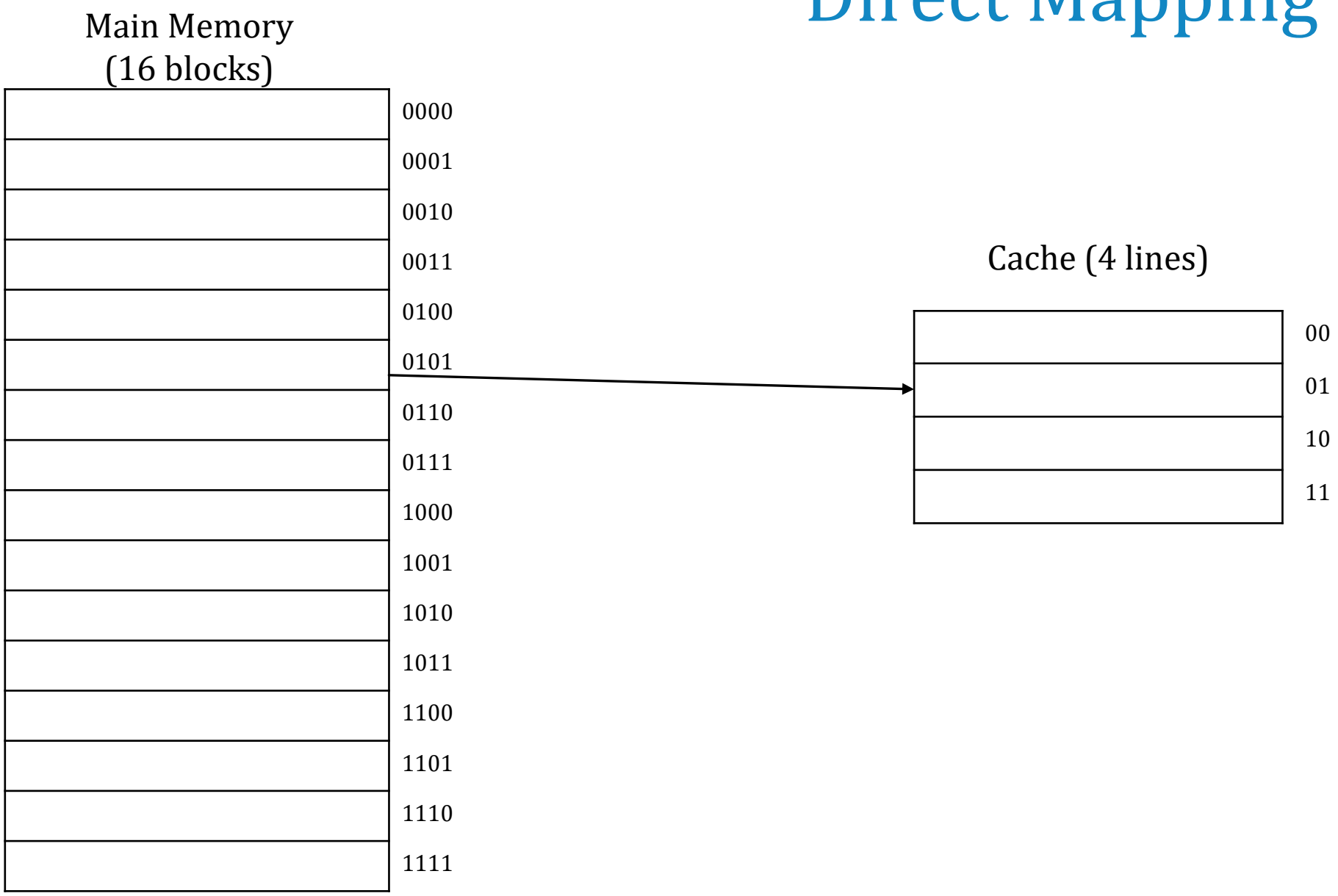
Direct Mapping



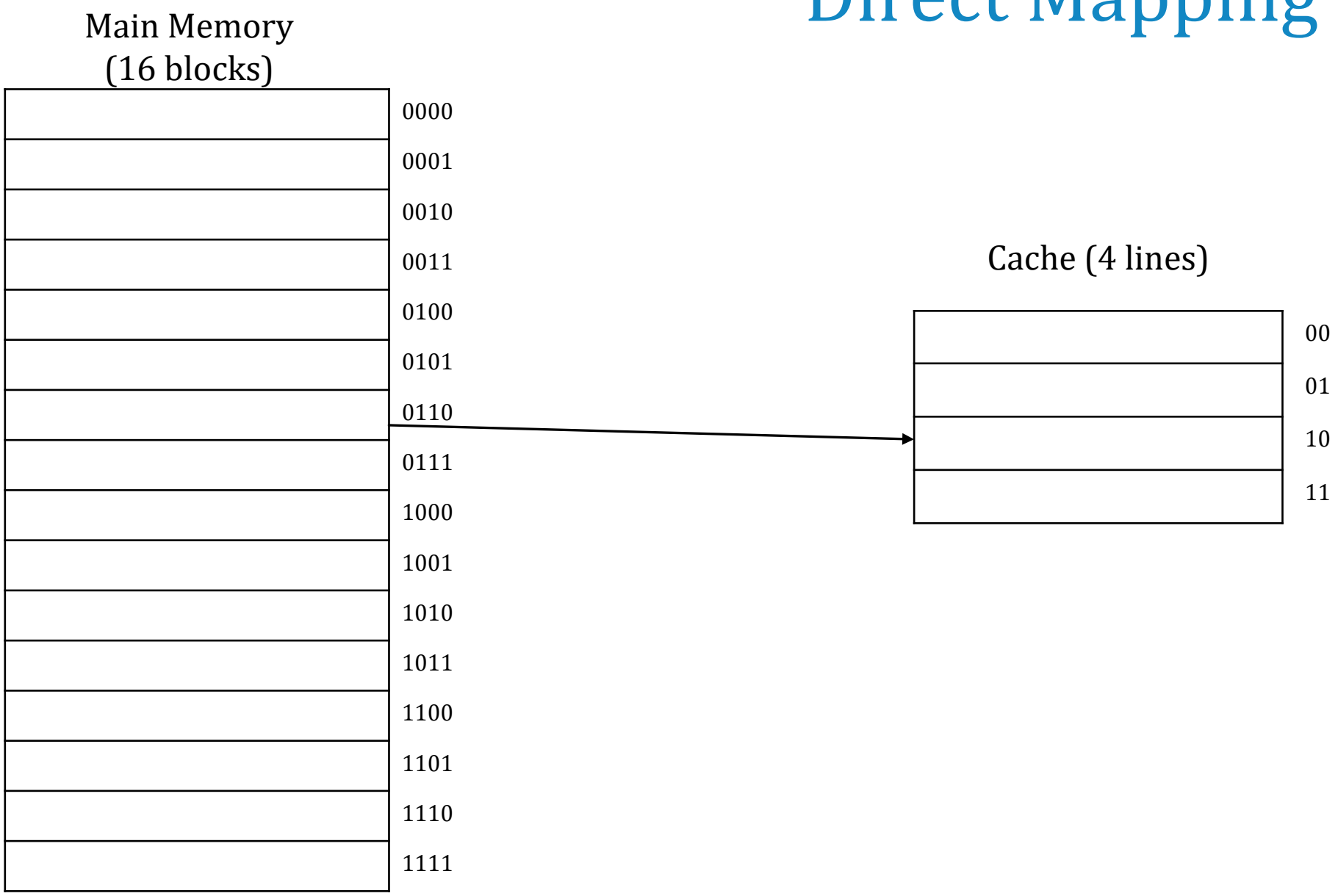
Direct Mapping



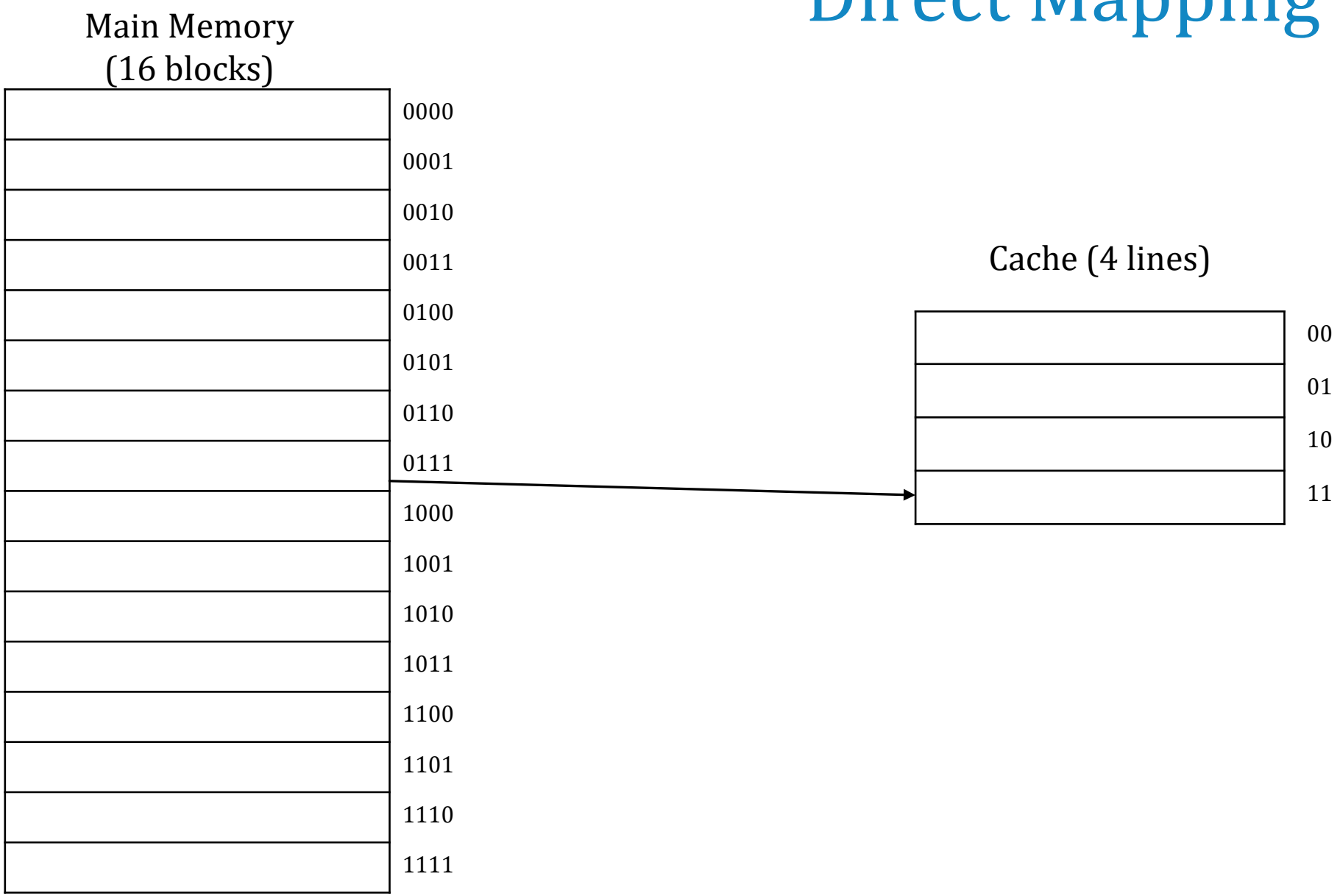
Direct Mapping



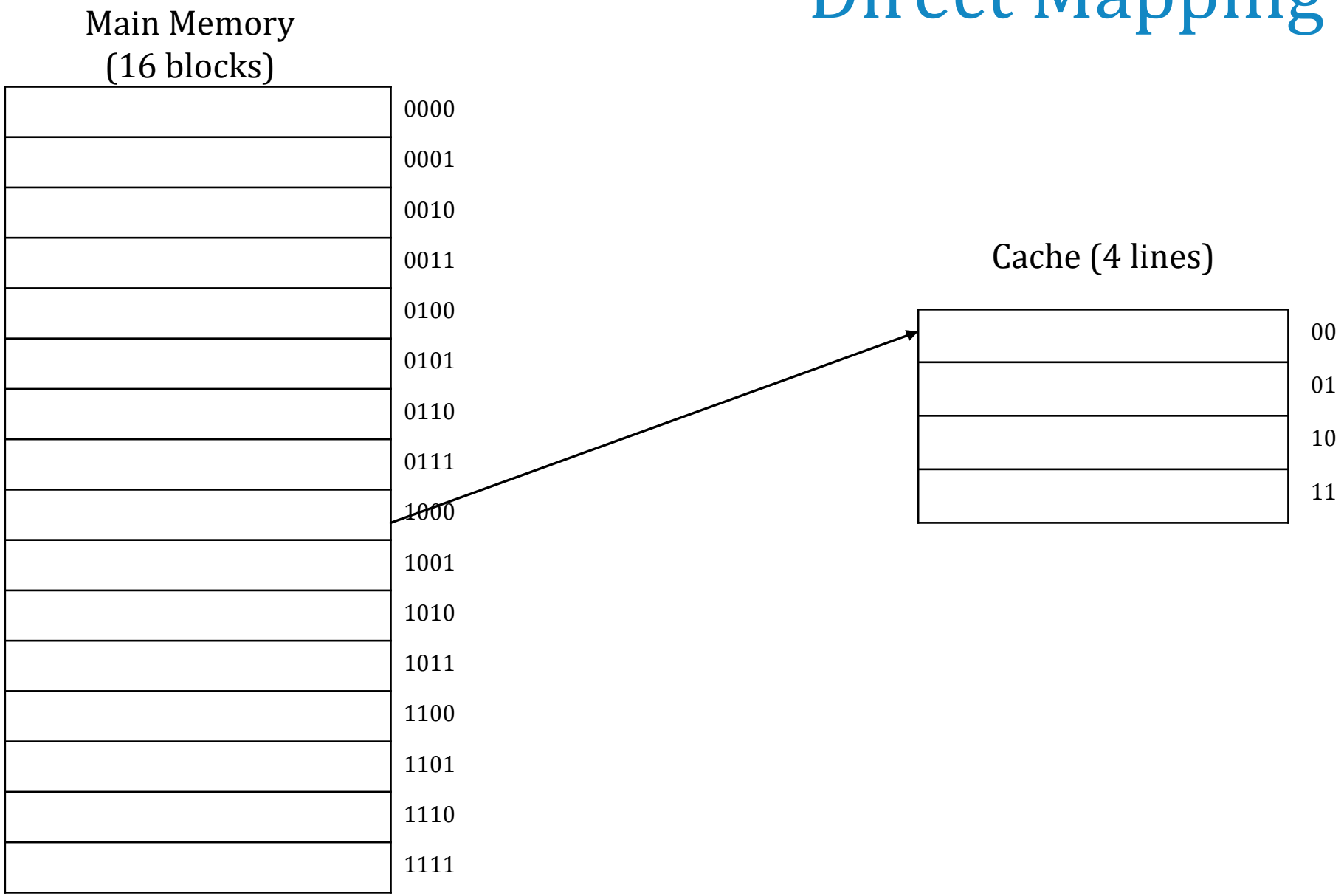
Direct Mapping



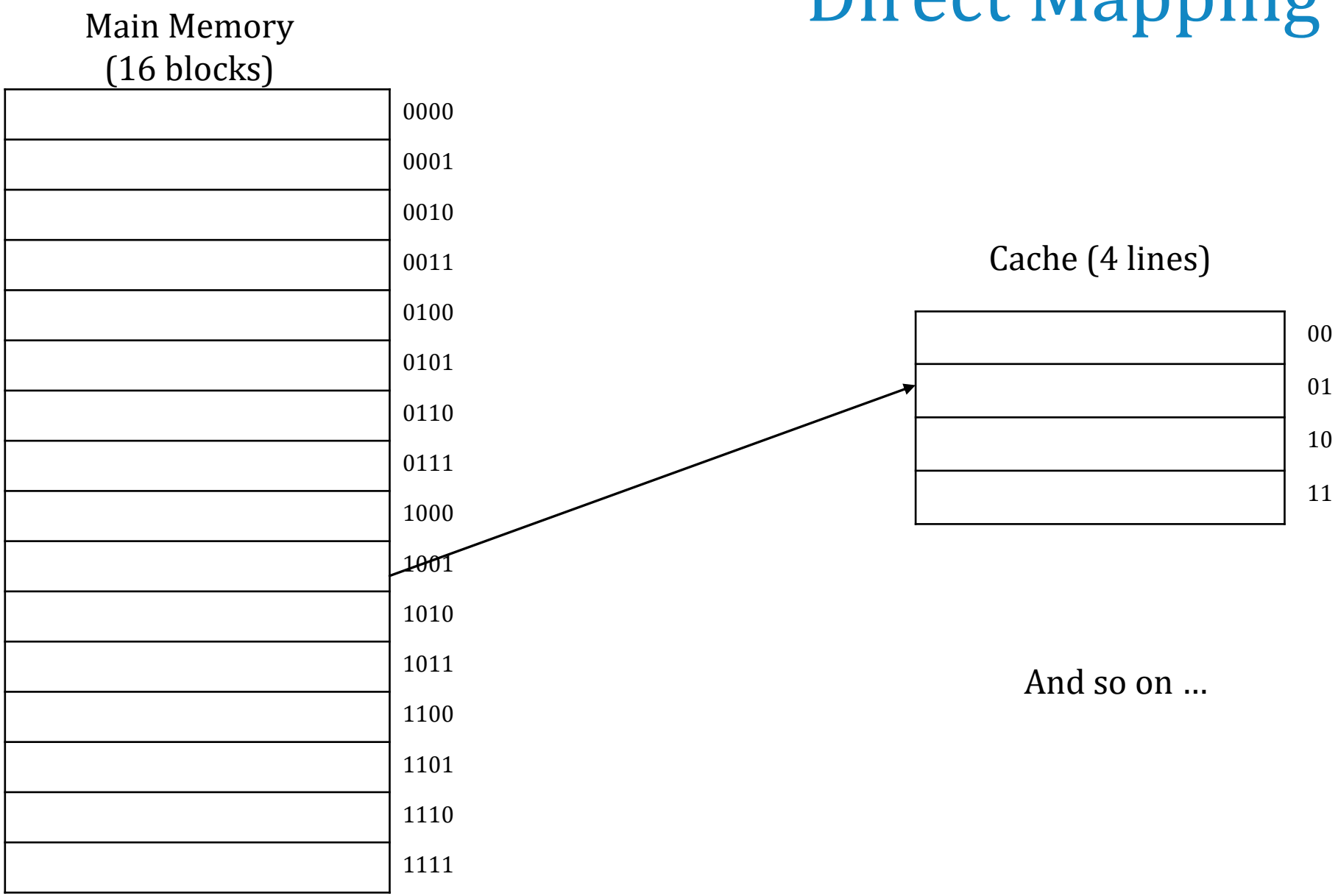
Direct Mapping



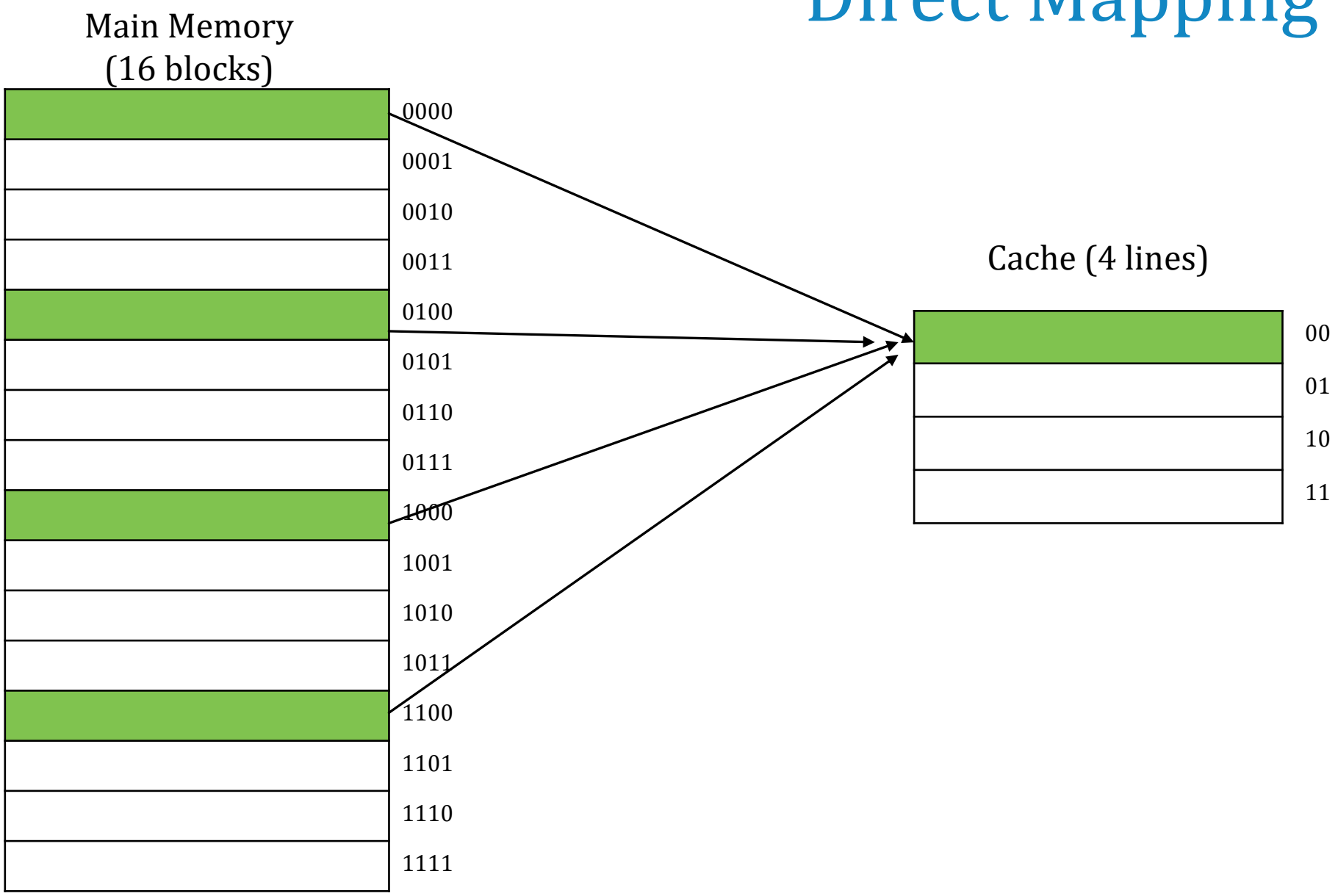
Direct Mapping



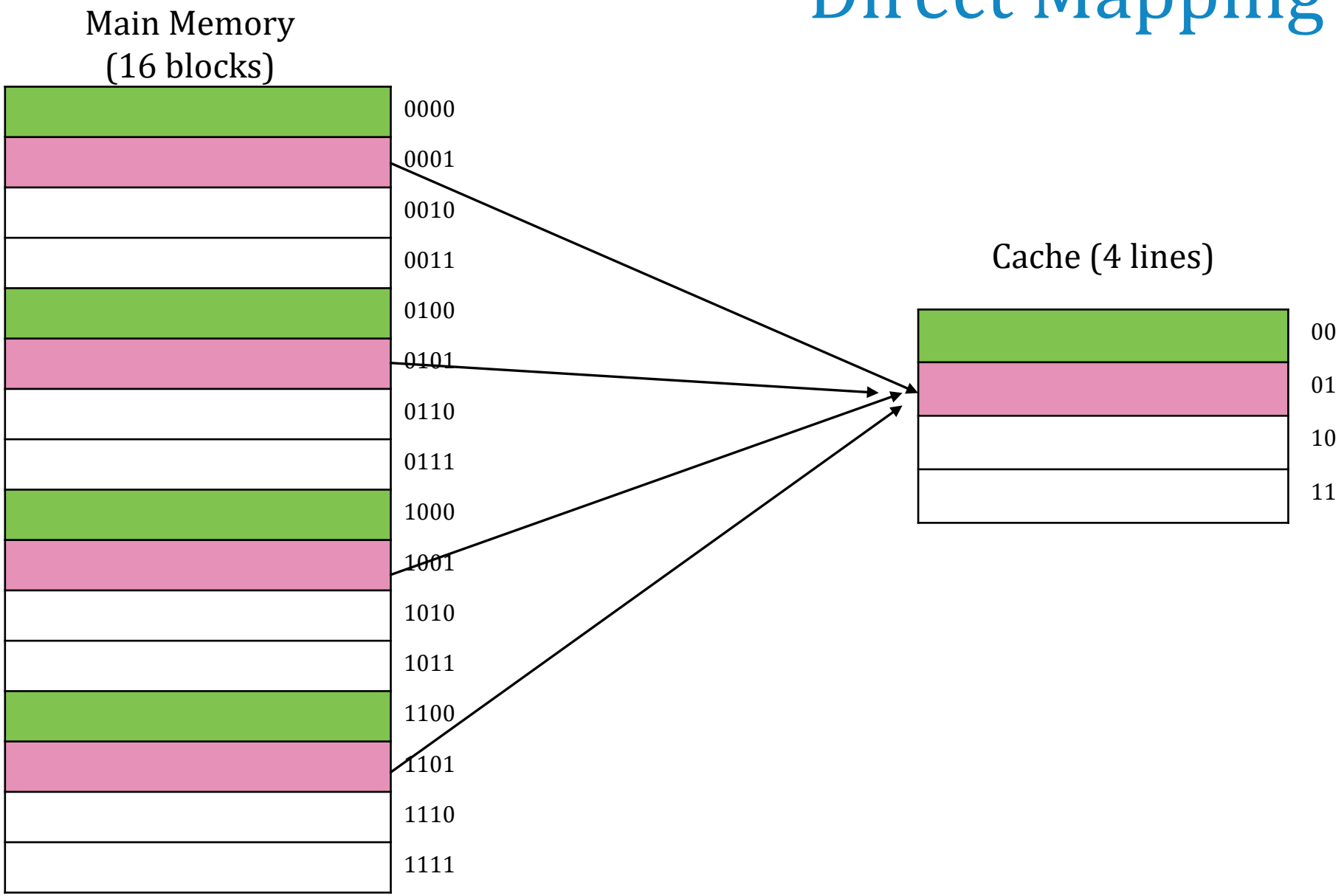
Direct Mapping



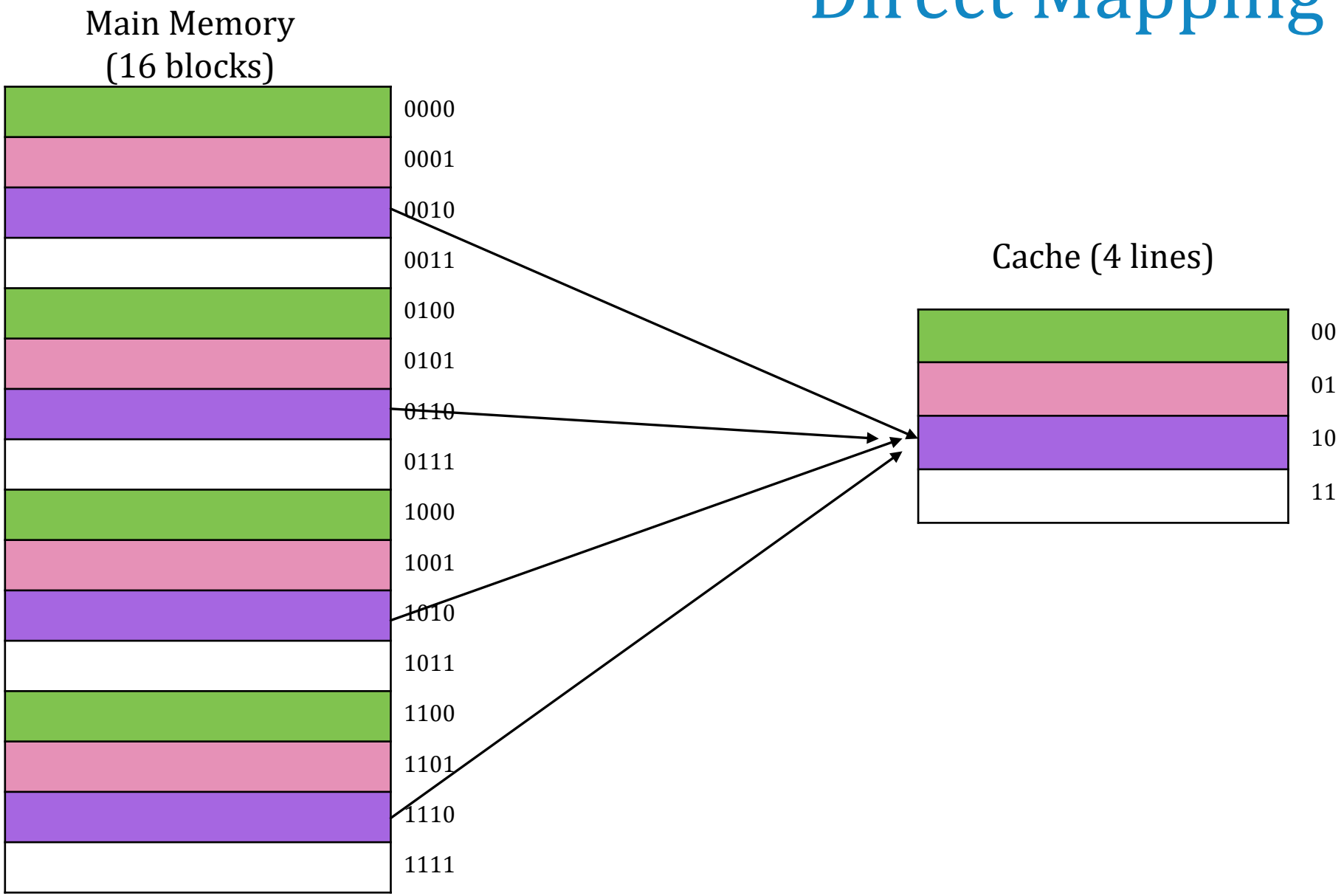
Direct Mapping



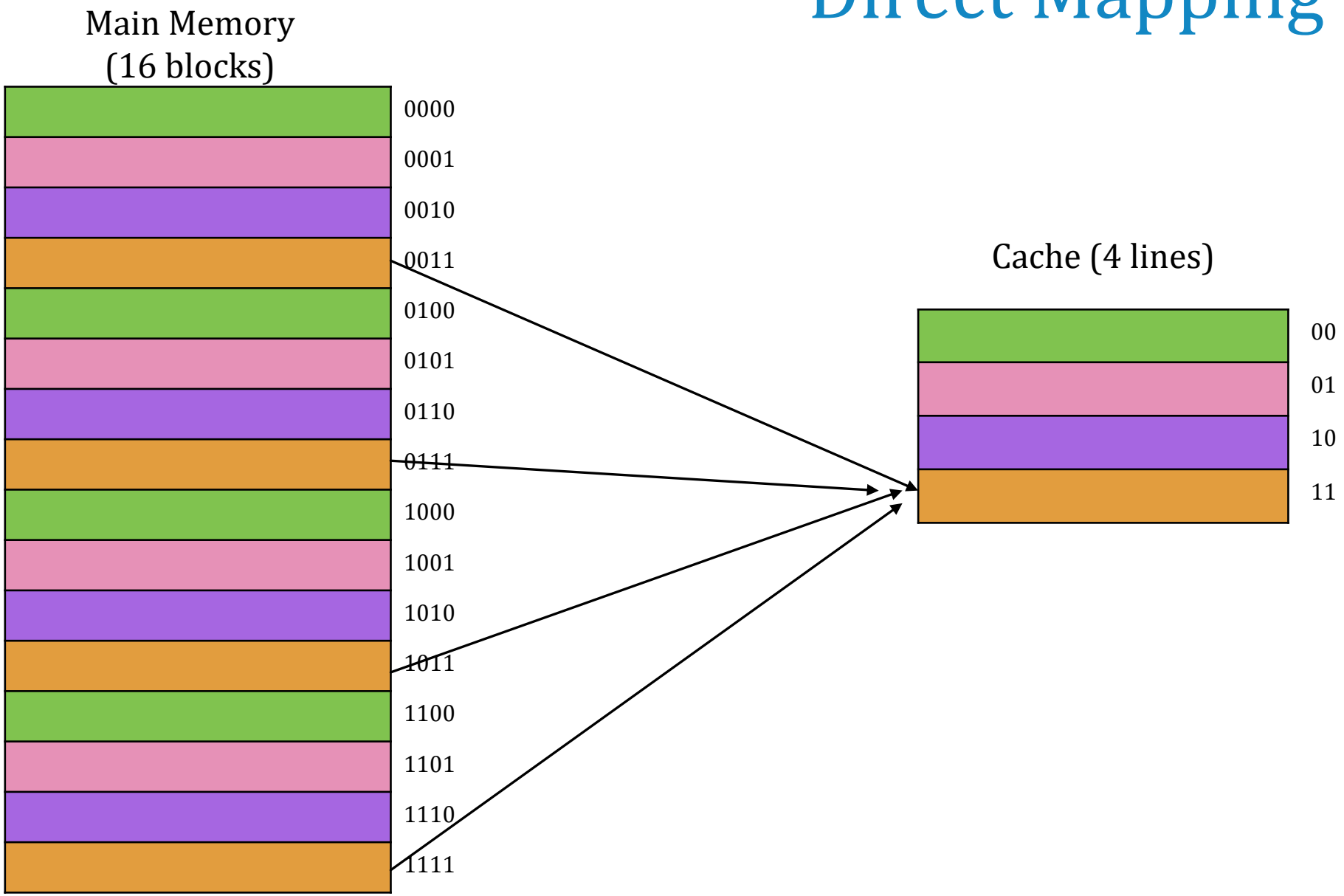
Direct Mapping



Direct Mapping



Direct Mapping



Direct Mapping

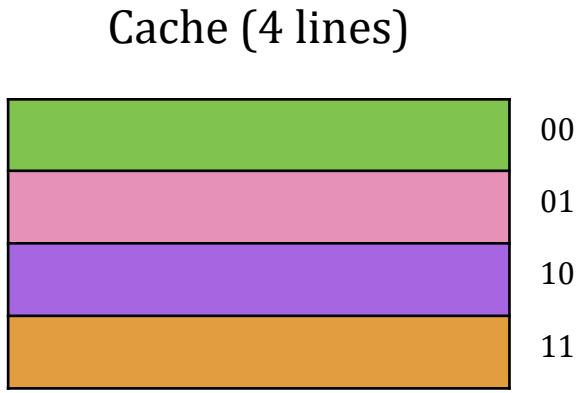
Cache (4 lines)

How many addresses do I need
to distinctly identify cache lines?

Direct Mapping

How many addresses do I need
to distinctly identify cache lines?

4

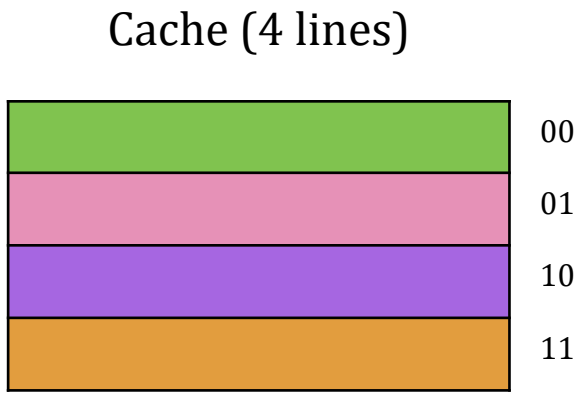


Direct Mapping

How many addresses do I need to distinctly identify cache lines?

4

How many bits do I need to generate 4 distinct addresses?



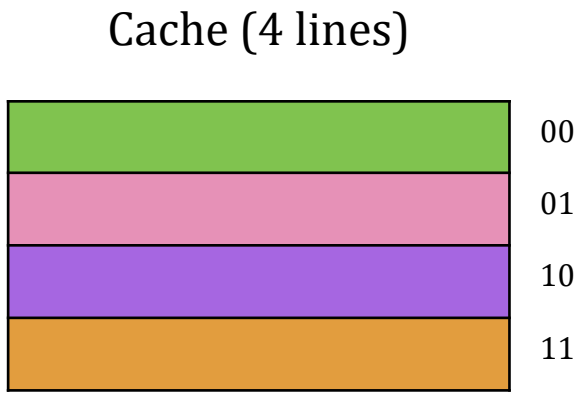
Direct Mapping

How many addresses do I need to distinctly identify cache lines?

4

How many bits do I need to generate 4 distinct addresses?

2



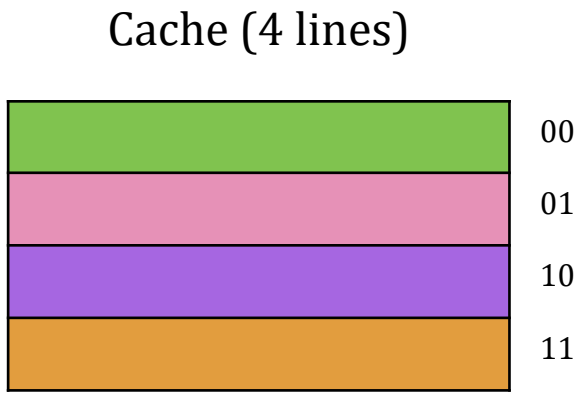
Direct Mapping

How many addresses do I need to distinctly identify L cache lines?

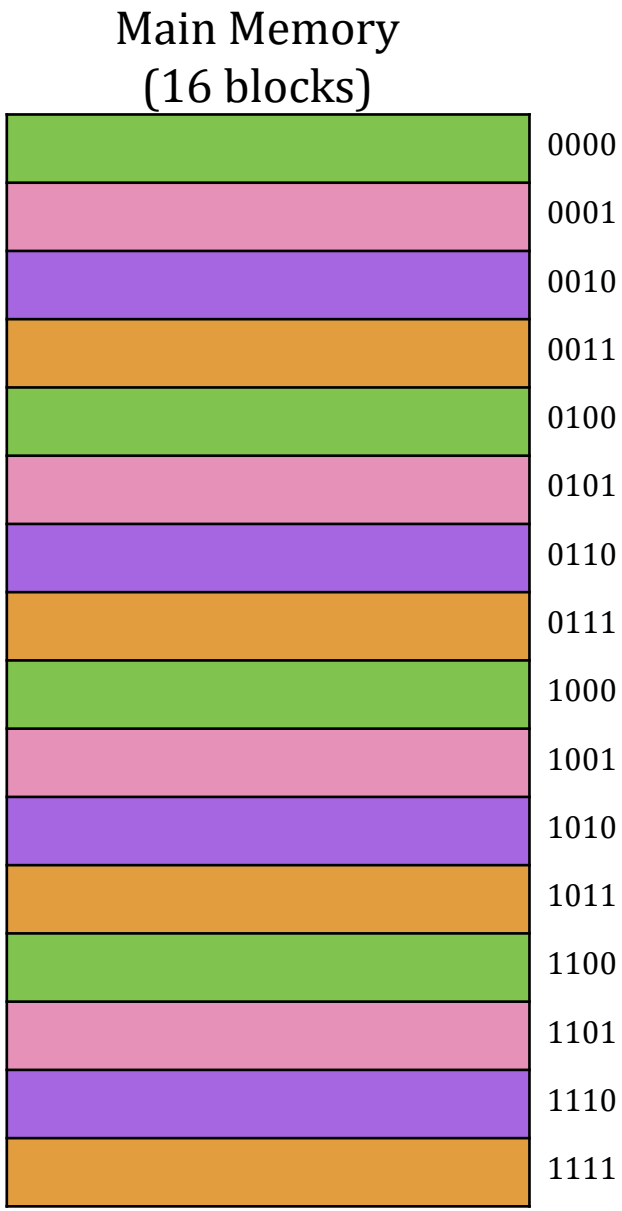
L

How many bits do I need to generate **L** distinct addresses (**L cache lines**)?

$$\log_2 L$$



Direct Mapping



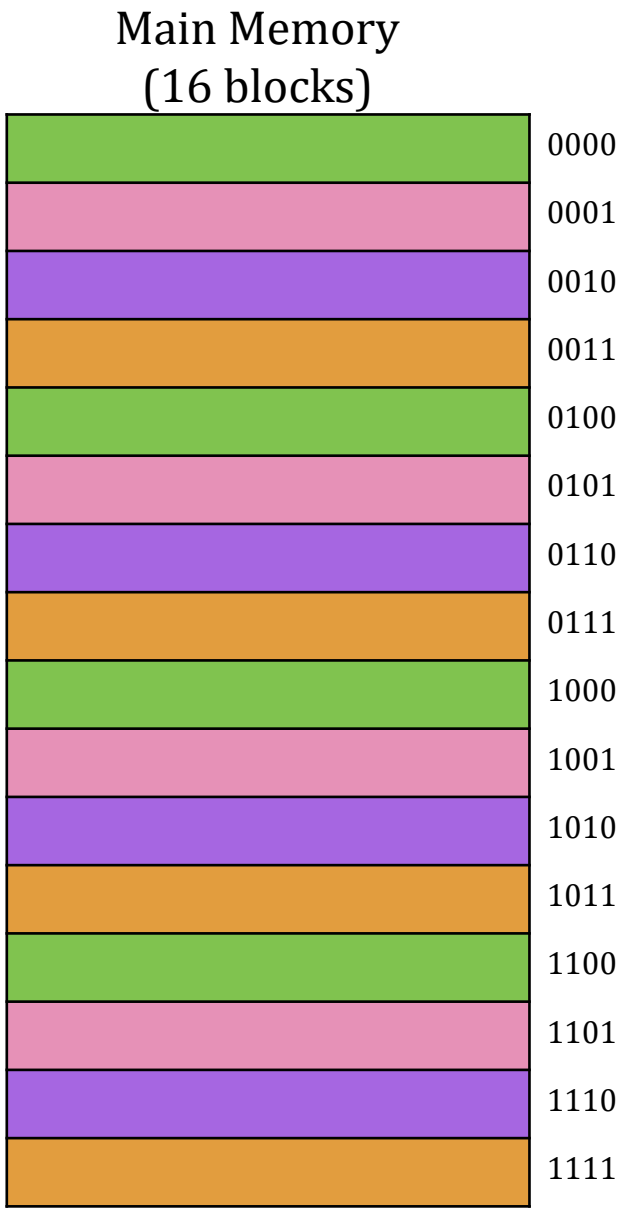
How many addresses do I need to distinctly identify memory blocks?

16

How many bits do I need to generate 16 distinct addresses?

4

Direct Mapping



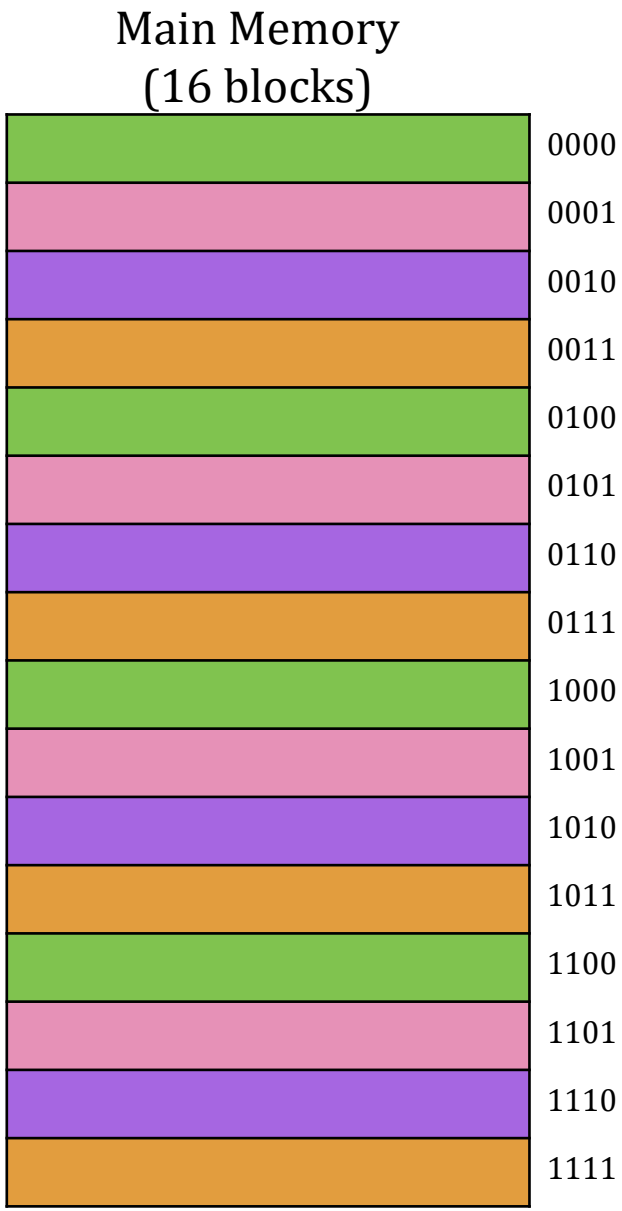
How many addresses do I need to distinctly identify M memory blocks?

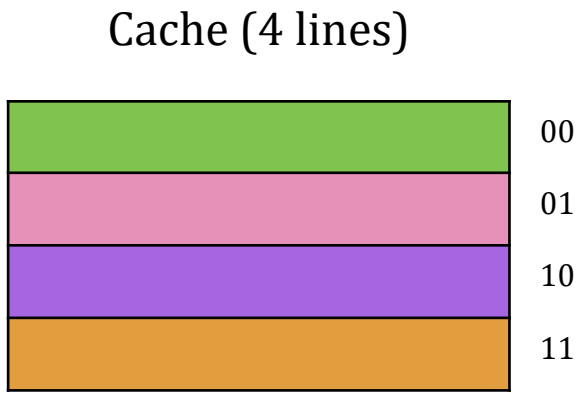
M

How many bits do I need to generate distinct addresses for **M memory blocks** ?

$$\log_2 \boldsymbol{M}$$

Direct Mapping



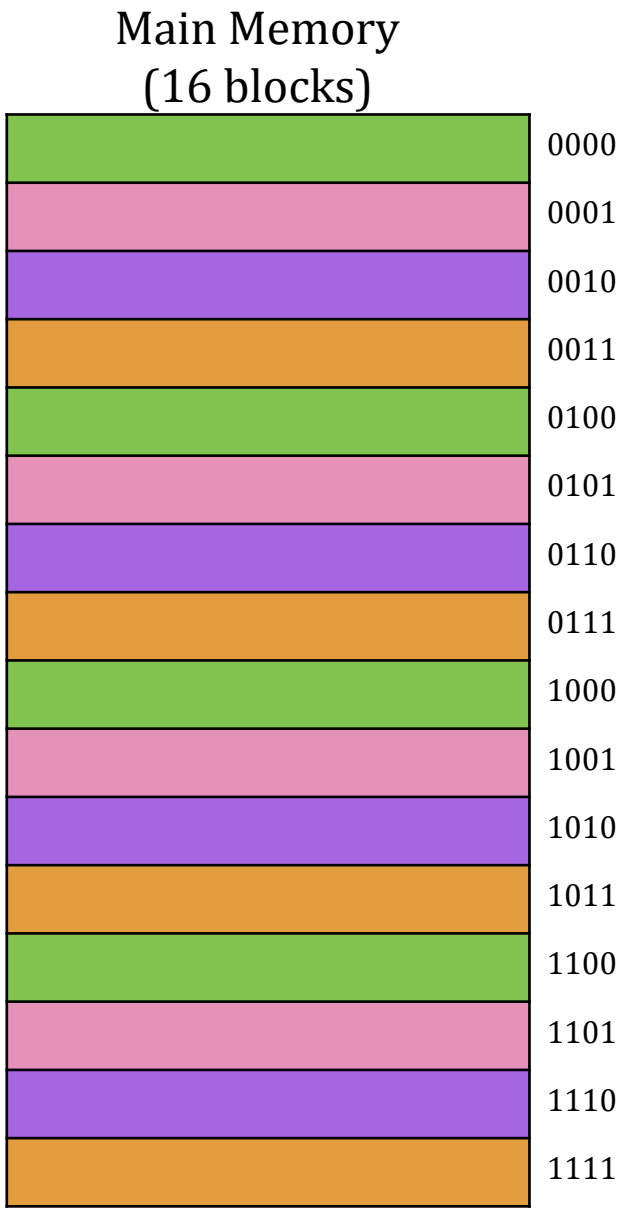


In our problem, **M=16, L=4**

4 bits are required to identify memory block

2 bits are required to identify cache lines

Direct Mapping



In our problem, **M=16, L=4**

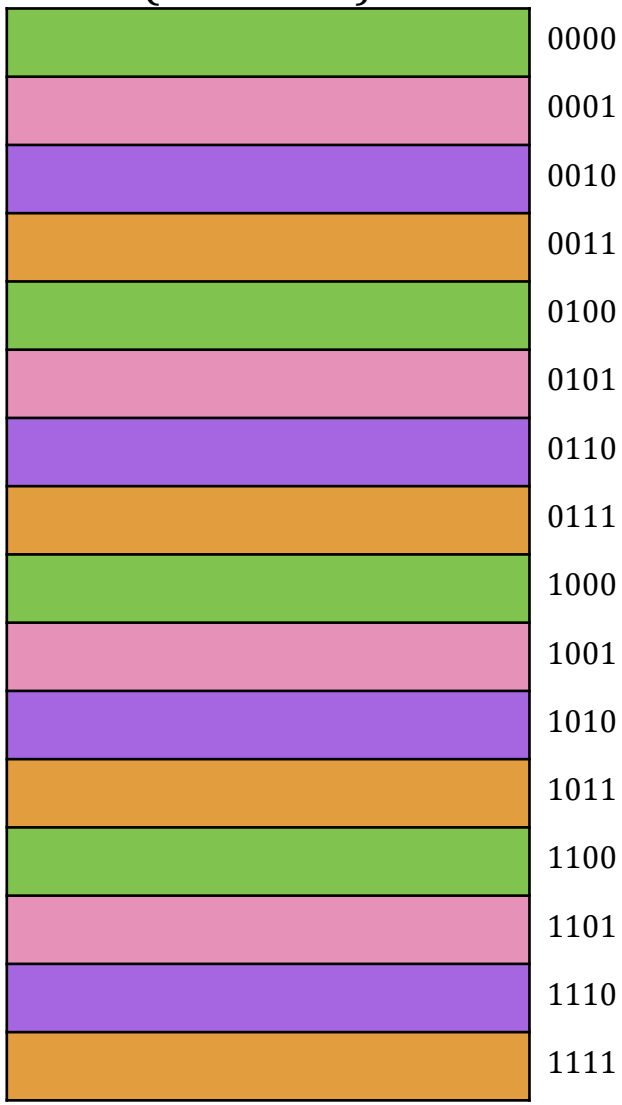
4 bits are required to identify memory block

2 bits are required to identify cache lines

If I know the address of the memory block, can I find the address of the corresponding cache line?

Direct Mapping

Main Memory
(16 blocks)



In our problem, **M=16, L=4**

4 bits are required to identify memory block

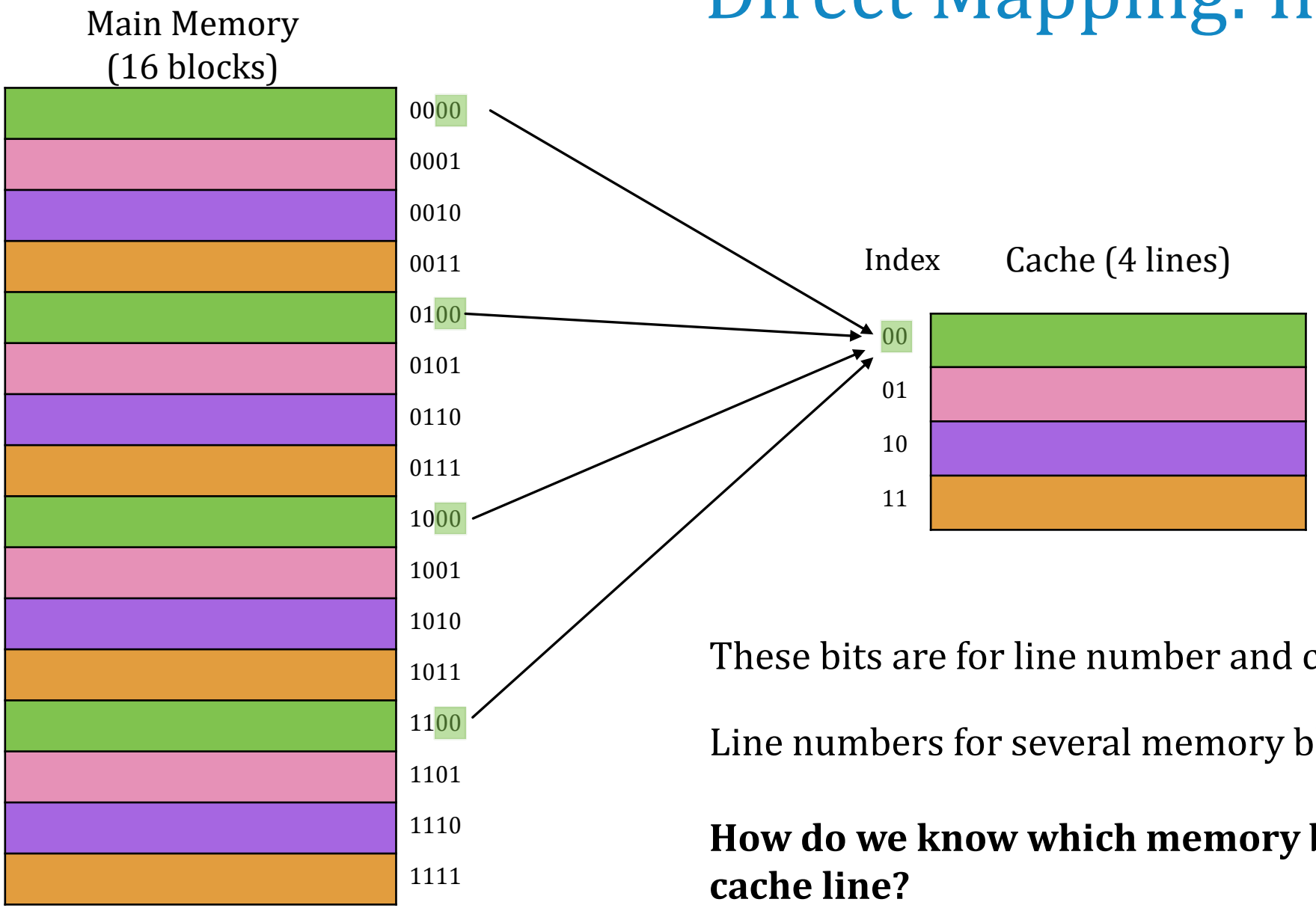
2 bits are required to identify cache lines

Cache (4 lines)

If I know the address of the memory block, can I find the address of the corresponding cache line?

Last (LSB) 2 bits ($\log_2 L$ bits) of the memory block address is the address of the cache line

Direct Mapping: Index

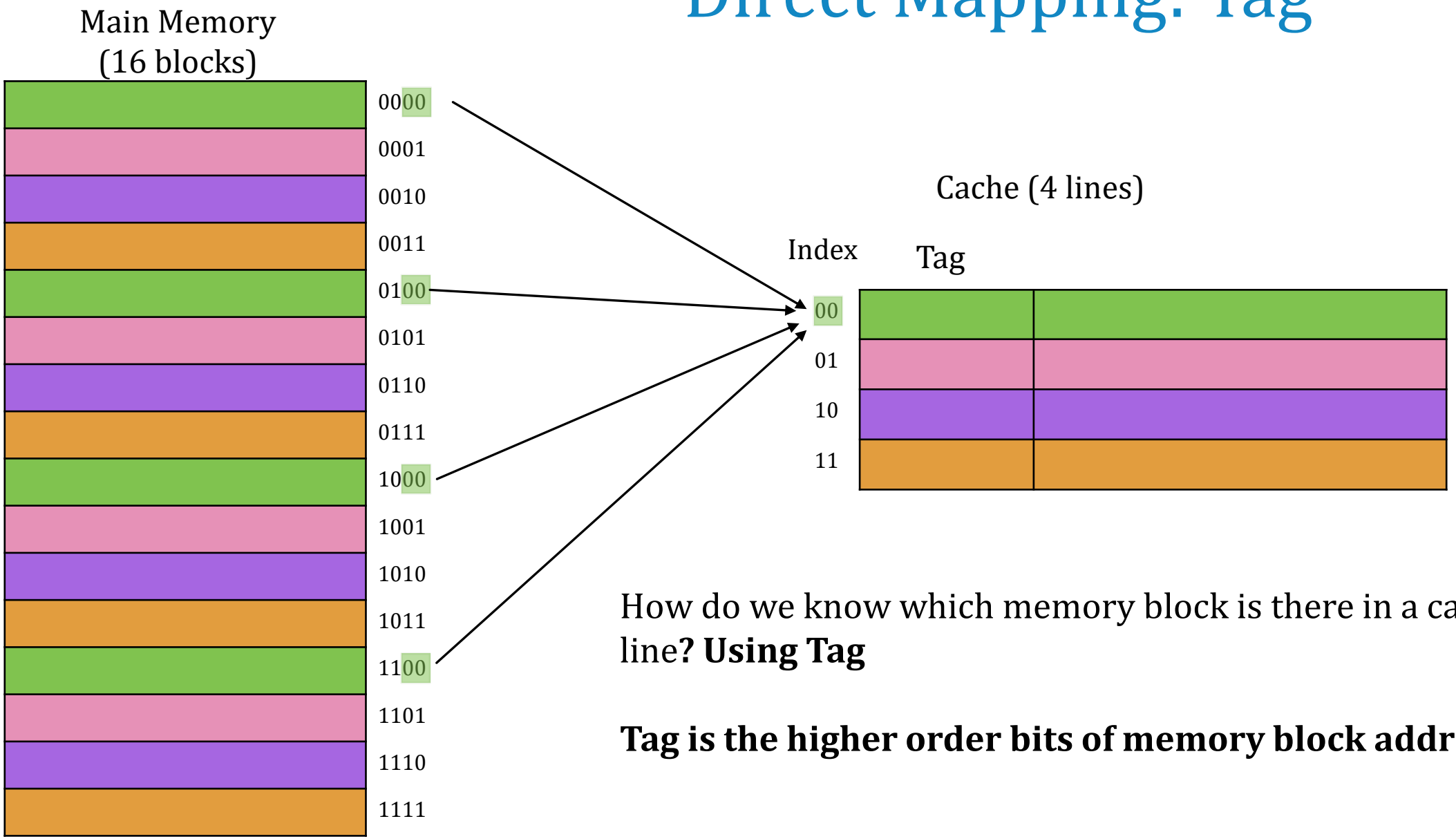


These bits are for line number and called **index**

Line numbers for several memory blocks are the same

How do we know which memory block is there in a cache line?

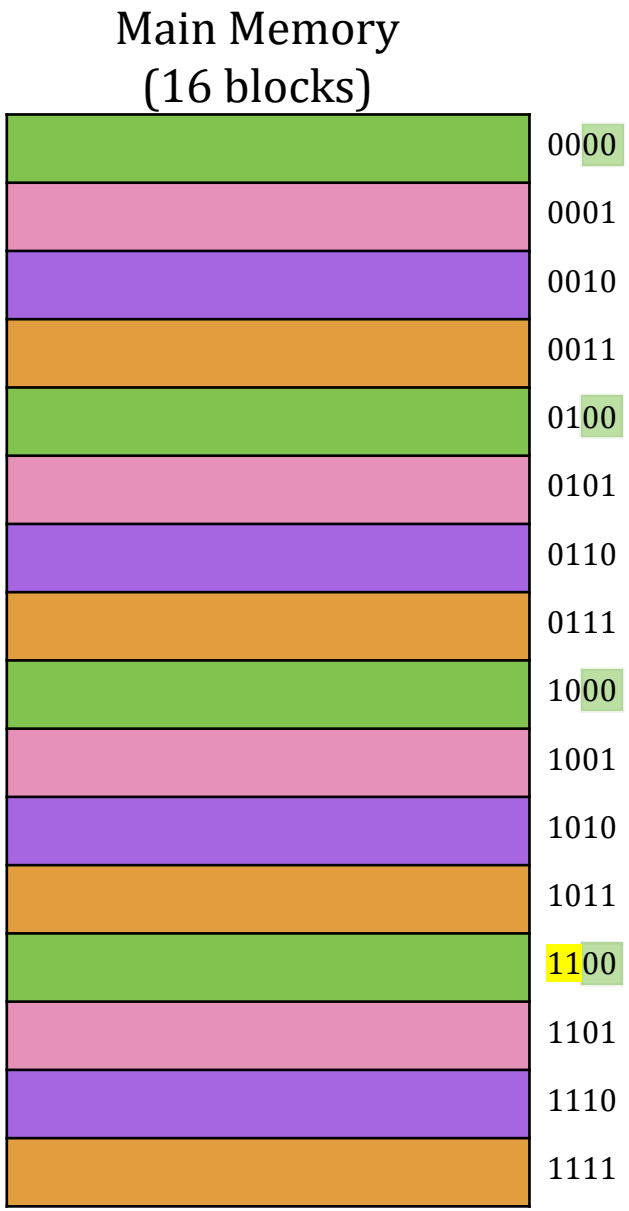
Direct Mapping: Tag



How do we know which memory block is there in a cache line**? Using Tag**

Tag is the higher order bits of memory block address

Direct Mapping: Tag

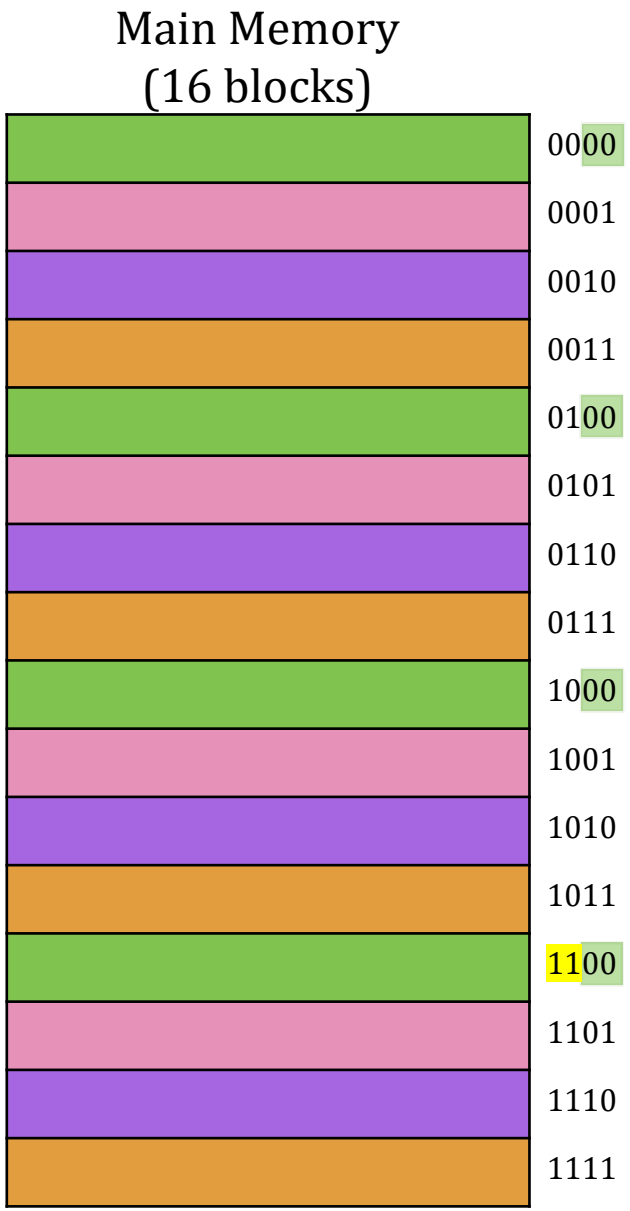


Cache (4 lines)

Index	Tag	
00	11	
01	01	
10	11	
11	10	

Tag is the higher order bits of memory block address

Direct Mapping: Tag



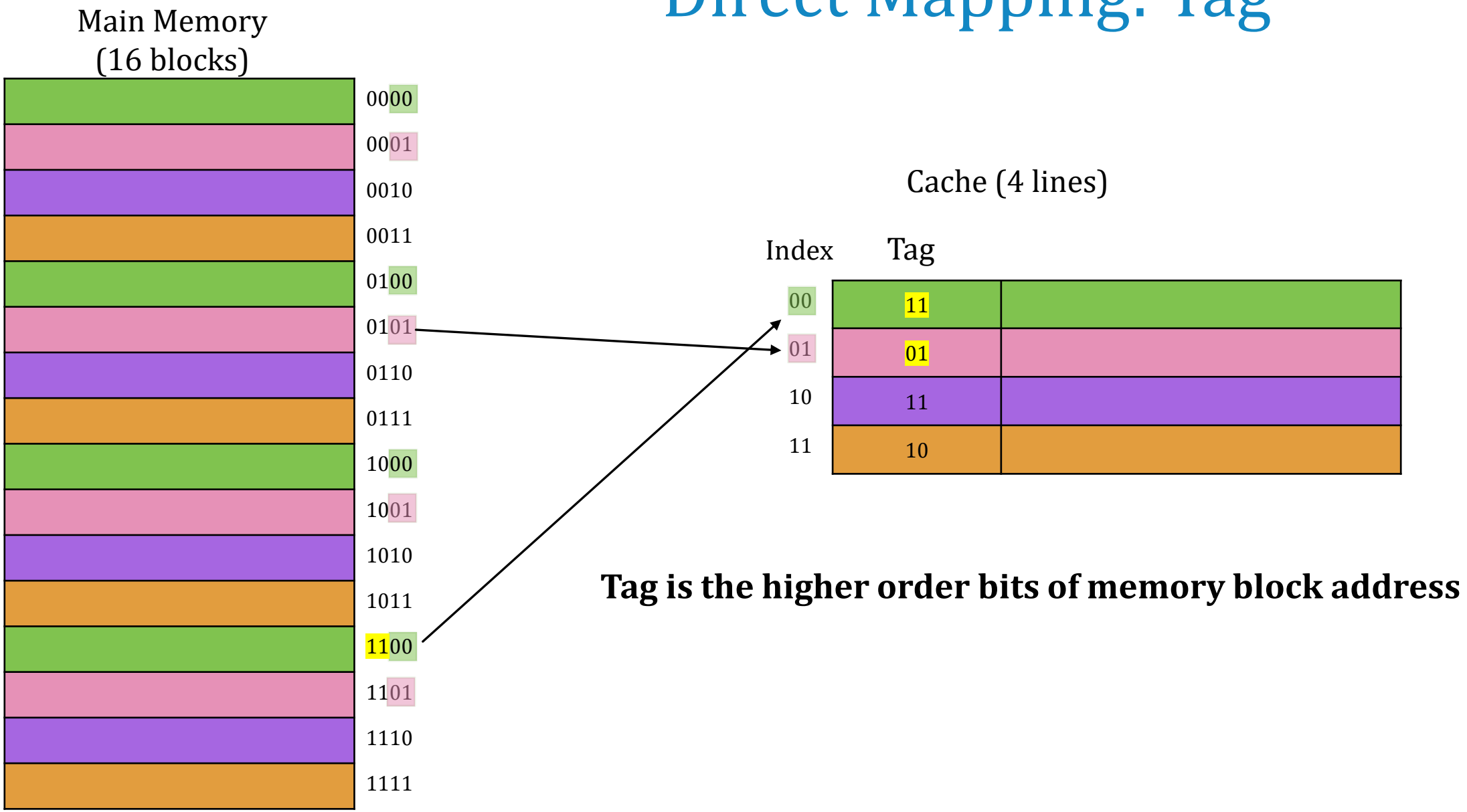
This block is in cache

Cache (4 lines)

Index	Tag	
00	11	
01	01	
10	11	
11	10	

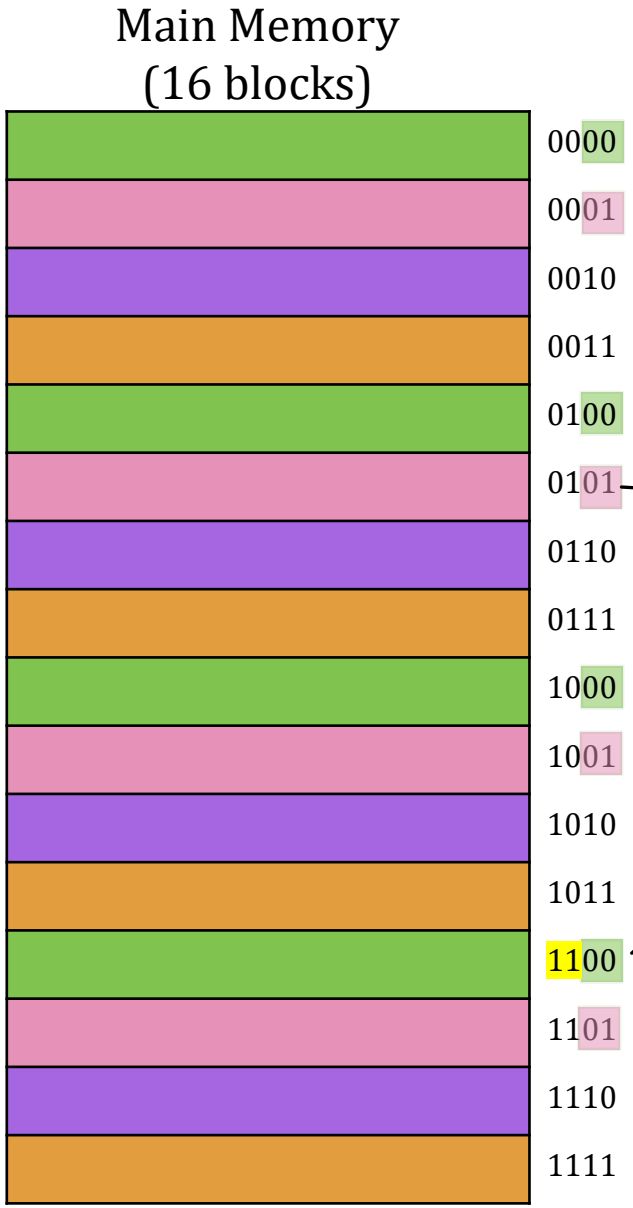
Tag is the higher order bits of memory block address

Direct Mapping: Tag



Tag is the higher order bits of memory block address

Valid bit

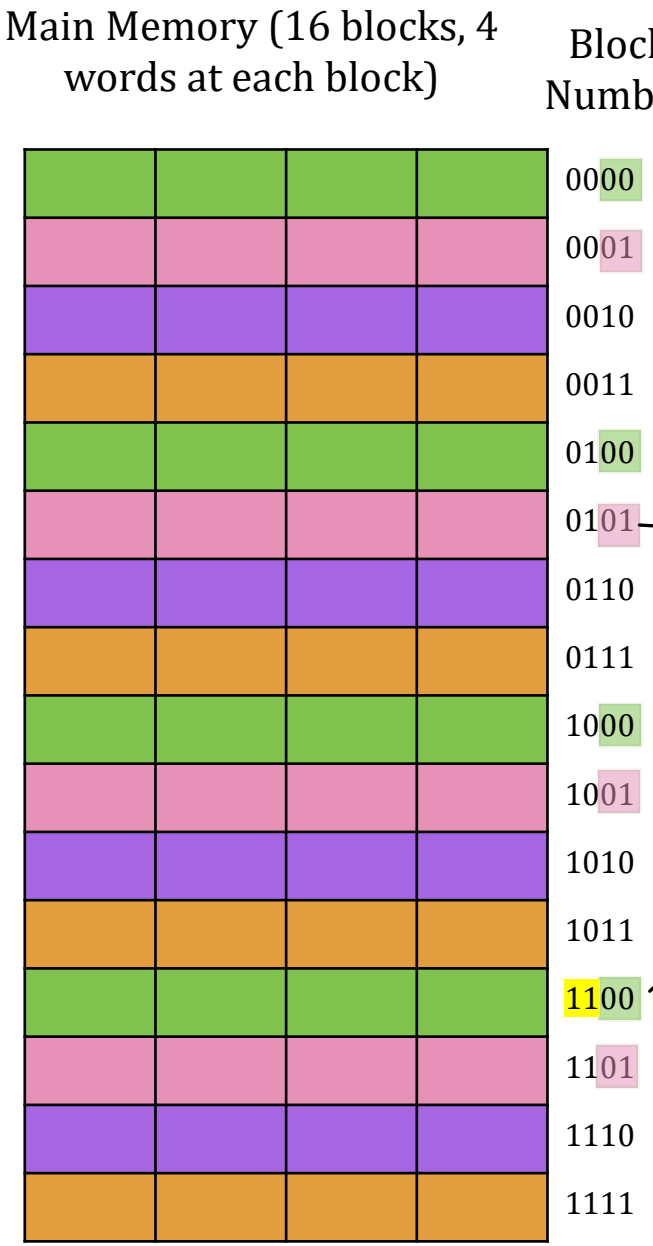


Cache (4 lines)

Index	Tag	Valid bit
00	11	1
01	01	0
10	11	0
11	10	1

- All cache blocks may not contain valid information
- Valid bit: Indicates whether a cache line has valid information

Access by the Processor



Cache (4 lines, 4 words at each line)

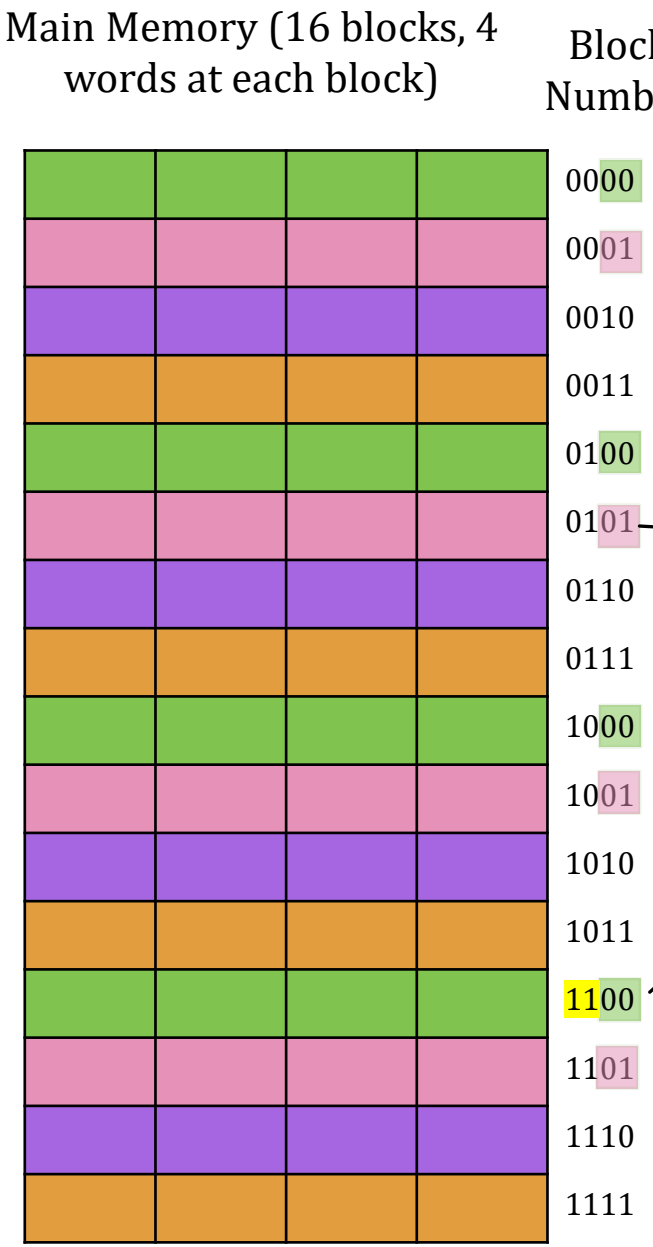
Index	Tag	Valid bit	Words			
00	11	1				
01	01	0				
10	11	0				
11	10	1				

CPU has to access words (not blocks)

In our example, we have **64 words**

How many bits do I need to access each word distinctly?

Access by the Processor



Cache (4 lines, 4 words at each line)

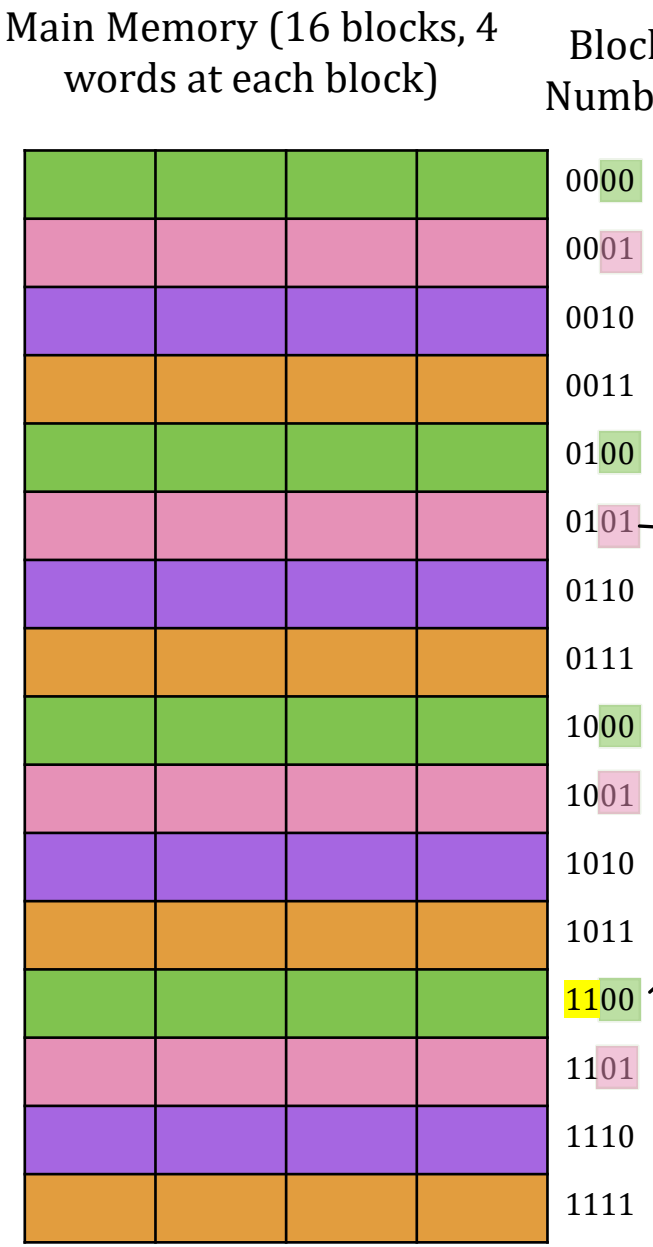
Index	Tag	Valid bit	Words
00	11	1	
01	01	0	
10	11	0	
11	10	1	

In our example, we have **64 words**

How many bits do I need to access each word distinctly? **6**

Therefore, the CPU has to generate a 6 bit address

Access by the Processor



Cache (4 lines, 4 words at each line)

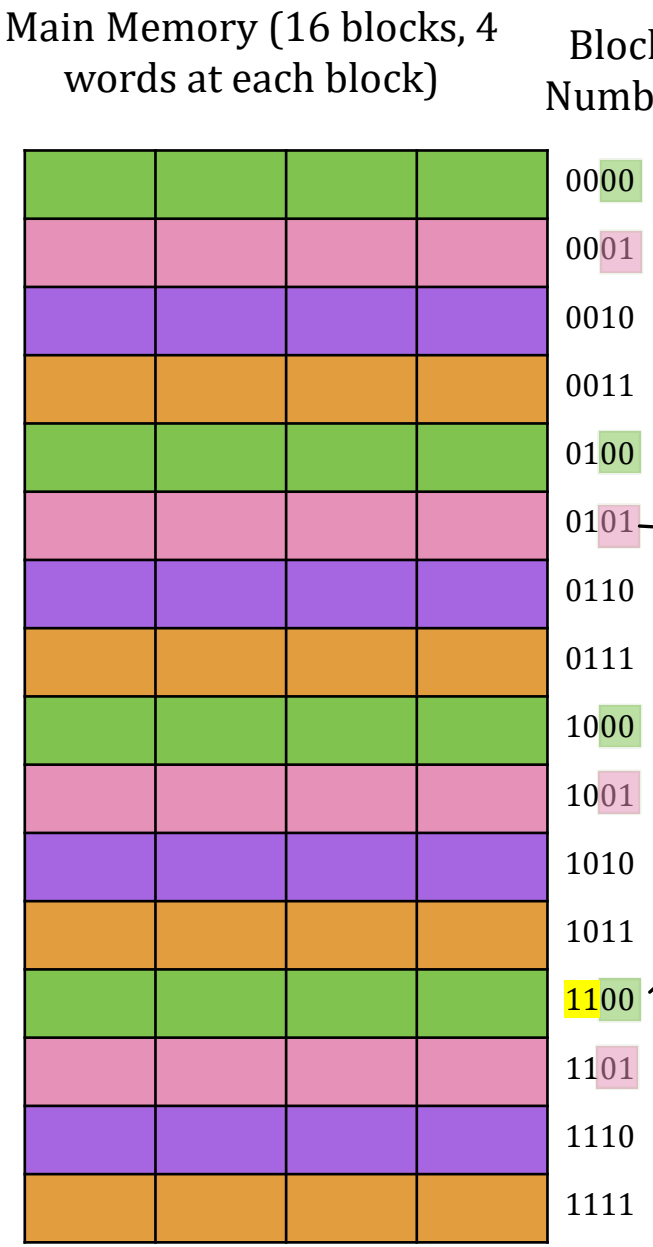
Index	Tag	Valid bit	Words			
00	11	1				
01	01	0				
10	11	0				
11	10	1				

Suppose the CPU is looking for a word W_k

CPU will first enquire whether cache has that word or not

How will the CPU search in the cache?

Access by the Processor



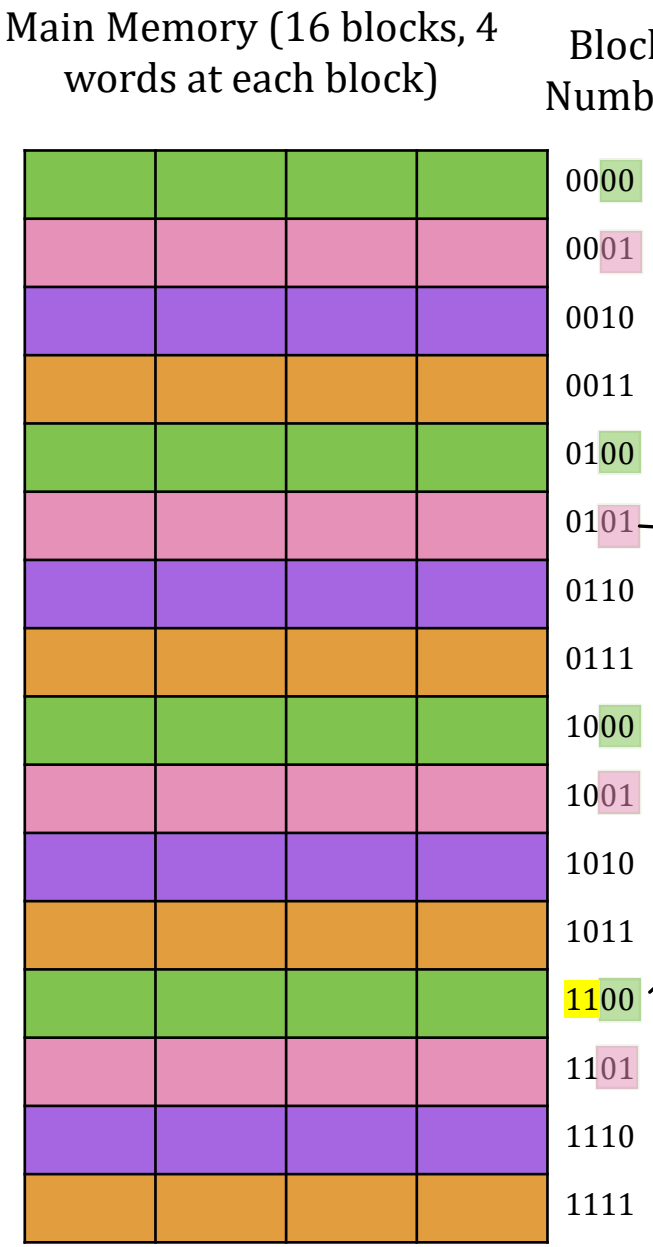
Cache (4 lines, 4 words at each line)

Index	Tag	Valid bit	Words			
00	11	1				
01	01	0				
10	11	0				
11	10	1				

How will the CPU search for this word W_k in the cache?

Corresponding to the word, the CPU has to identify the cache line number first

Access by the Processor



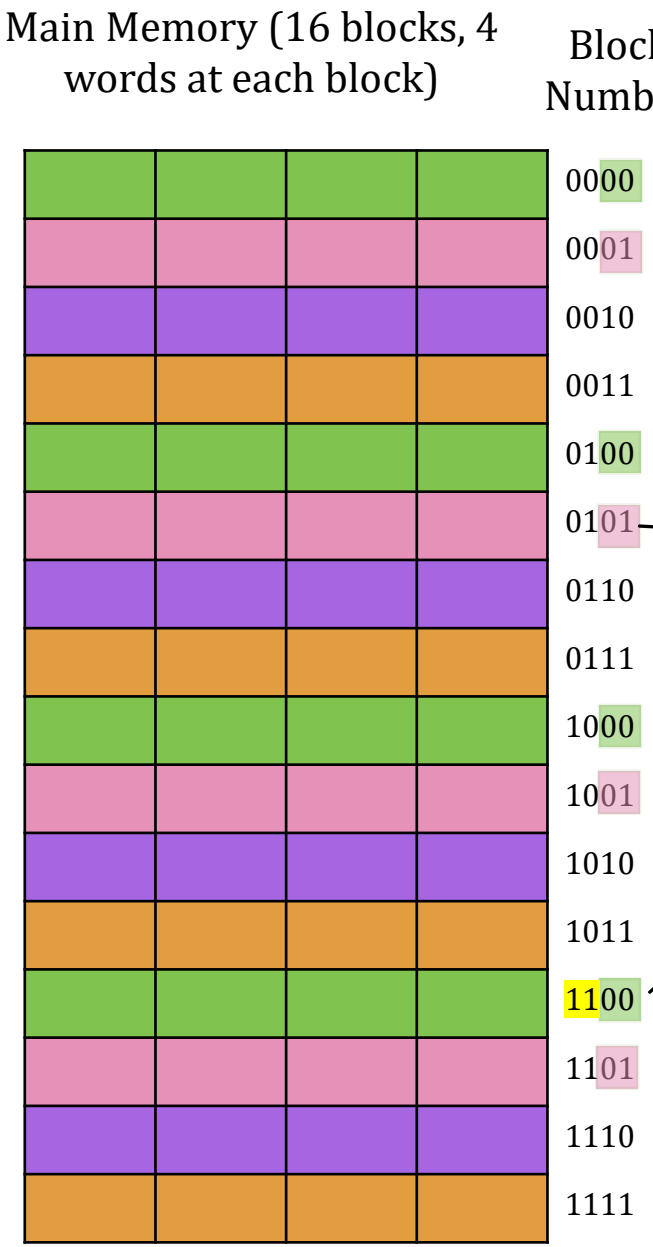
Cache (4 lines, 4 words at each line)

Index	Tag	Valid bit	Words			
00	11	1				
01	01	0				
10	11	0				
11	10	1				

Corresponding to the word, the CPU has to identify the cache line number first

If CPU knows the memory block number, it can find out cache line number

Access by the Processor



Cache (4 lines, 4 words at each line)

Index	Tag	Valid bit	Words			
00	11	1				
01	01	0				
10	11	0				
11	10	1				

If CPU knows the memory block number, it can find out cache line number

How do the CPU know the memory block number corresponding to word W_k?

How do the CPU Know the Memory Block Number?

Main Memory

w0	w1	w2	w3
w4	w5	w6	w7
w8	w9	w10	w11
w12	w13	w14	w15
w16	w17	w18	w19
w20	w21	w22	w23
w24	w25	w26	w27
w28	w29	w30	w31
w32	w33	w34	w35
w36	w37	w38	w39
w40	w41	w42	w43
w44	w45	w46	w47
w48	w49	w50	w51
w52	w53	w54	w55
w56	w57	w58	w59
w60	w61	w62	w63

Block Number	Binary Representation of block number
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

How do the CPU Know the Memory Block Number?

Main Memory

w0	w1	w2	w3
w4	w5	w6	w7
w8	w9	w10	w11
w12	w13	w14	w15
w16	w17	w18	w19
w20	w21	w22	w23
w24	w25	w26	w27
w28	w29	w30	w31
w32	w33	w34	w35
w36	w37	w38	w39
w40	w41	w42	w43
w44	w45	w46	w47
w48	w49	w50	w51
w52	w53	w54	w55
w56	w57	w58	w59
w60	w61	w62	w63

Block Number	Binary Representation of block number
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Suppose CPU want to access w25

Hence, CPU generates a 6 bit number for 25

What is the number?

How do the CPU Know the Memory Block Number?

Main Memory

w0	w1	w2	w3
w4	w5	w6	w7
w8	w9	w10	w11
w12	w13	w14	w15
w16	w17	w18	w19
w20	w21	w22	w23
w24	w25	w26	w27
w28	w29	w30	w31
w32	w33	w34	w35
w36	w37	w38	w39
w40	w41	w42	w43
w44	w45	w46	w47
w48	w49	w50	w51
w52	w53	w54	w55
w56	w57	w58	w59
w60	w61	w62	w63

Block Number	Binary Representation of block number
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Suppose CPU want to access w25

Hence, CPU generates a 6 bit number for 25

What is the number?

011001

How do the CPU Know the Memory Block Number?

Memory size: 64 words

Each block: 4 words

No. of blocks M=64/4=16

Main Memory

w0	w1	w2	w3
w4	w5	w6	w7
w8	w9	w10	w11
w12	w13	w14	w15
w16	w17	w18	w19
w20	w21	w22	w23
w24	w25	w26	w27
w28	w29	w30	w31
w32	w33	w34	w35
w36	w37	w38	w39
w40	w41	w42	w43
w44	w45	w46	w47
w48	w49	w50	w51
w52	w53	w54	w55
w56	w57	w58	w59
w60	w61	w62	w63

Block Number	Binary Representation of block number
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Suppose CPU want to access w25

Hence, CPU generates a 6 bit number for 25

What is the number?

011001

The top 4 ($\log_2 M$ **bits**) indicate the memory block number

How do the CPU Know the Memory Block Number?

Memory size: 64 words

Each block: 4 words

No. of blocks M=64/4=16

Main Memory

w0	w1	w2	w3
w4	w5	w6	w7
w8	w9	w10	w11
w12	w13	w14	w15
w16	w17	w18	w19
w20	w21	w22	w23
w24	w25	w26	w27
w28	w29	w30	w31
w32	w33	w34	w35
w36	w37	w38	w39
w40	w41	w42	w43
w44	w45	w46	w47
w48	w49	w50	w51
w52	w53	w54	w55
w56	w57	w58	w59
w60	w61	w62	w63

Block Number	Binary Representation of block number
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

011001

The bottom (LSB) 2 bits ($6 - \log_2 M$ **bits**) indicate the position of the word

Offset

How do the CPU Know the Memory Block Number?

Memory size: 64 words

Each block: 4 words

No. of blocks M=64/4=16

Main Memory

w0	w1	w2	w3
w4	w5	w6	w7
w8	w9	w10	w11
w12	w13	w14	w15
w16	w17	w18	w19
w20	w21	w22	w23
w24	w25	w26	w27
w28	w29	w30	w31
w32	w33	w34	w35
w36	w37	w38	w39
w40	w41	w42	w43
w44	w45	w46	w47
w48	w49	w50	w51
w52	w53	w54	w55
w56	w57	w58	w59
w60	w61	w62	w63

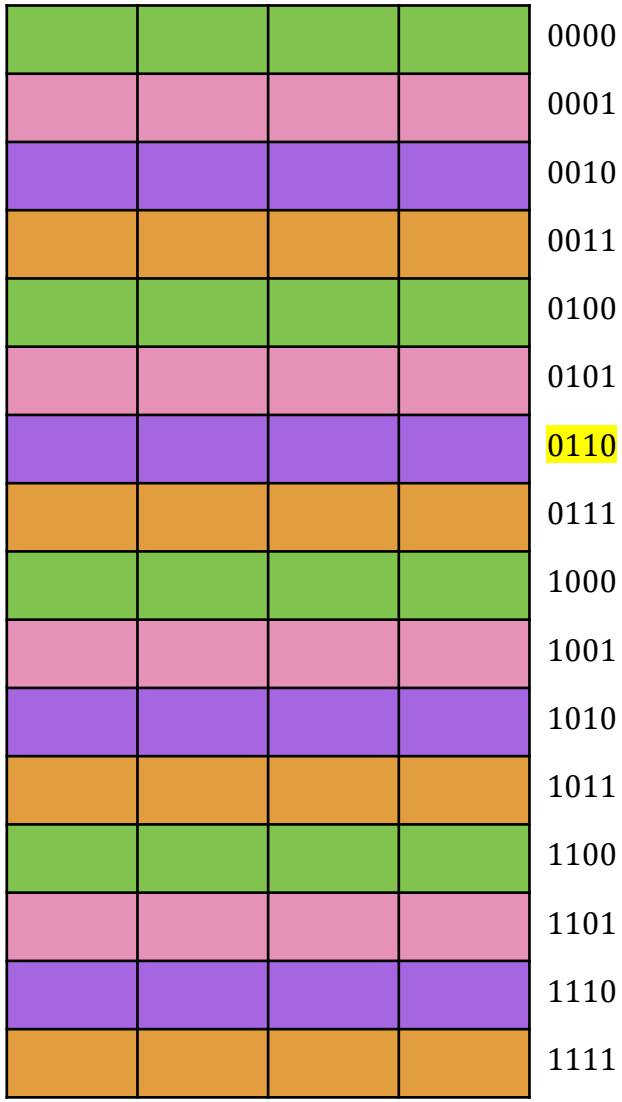
Block Number	Binary Representation of block number
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

011001

Memory Block number is 0110

Access by the Processor

Main Memory (16 blocks, 4 words at each block)



Cache (4 lines, 4 words at each line)

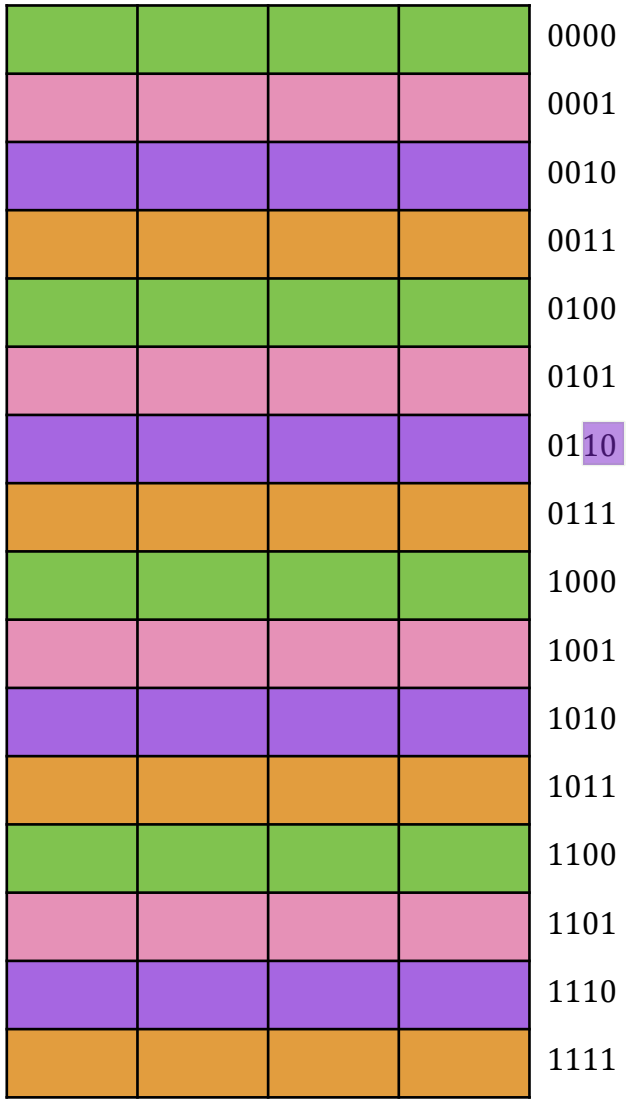
Index	Tag	Valid bit	Words			
00	11	1				
01	01	0				
10	11	0				
11	10	1				

Memory Block number is 0110

So cache line number is

Access by the Processor

Main Memory (16 blocks, 4 words at each block)



Cache (4 lines, 4 words at each line)

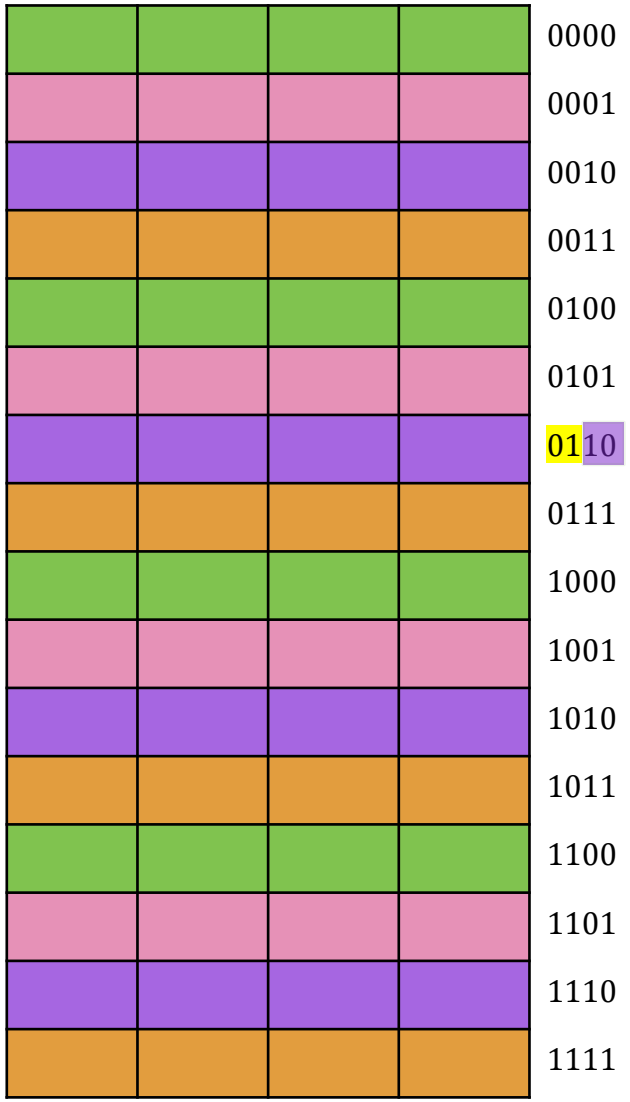
Index	Tag	Valid bit	Words			
00	11	1				
01	01	0				
10	01	1				
11	10	1				

Memory Block number is 0110

So cache line number is 10

Access by the Processor

Main Memory (16 blocks, 4 words at each block)



Cache (4 lines, 4 words at each line)

Index	Tag	Valid bit	Words			
00	11	1				
01	01	0				
10	01	1				
11	10	1				

Memory Block number is 0110

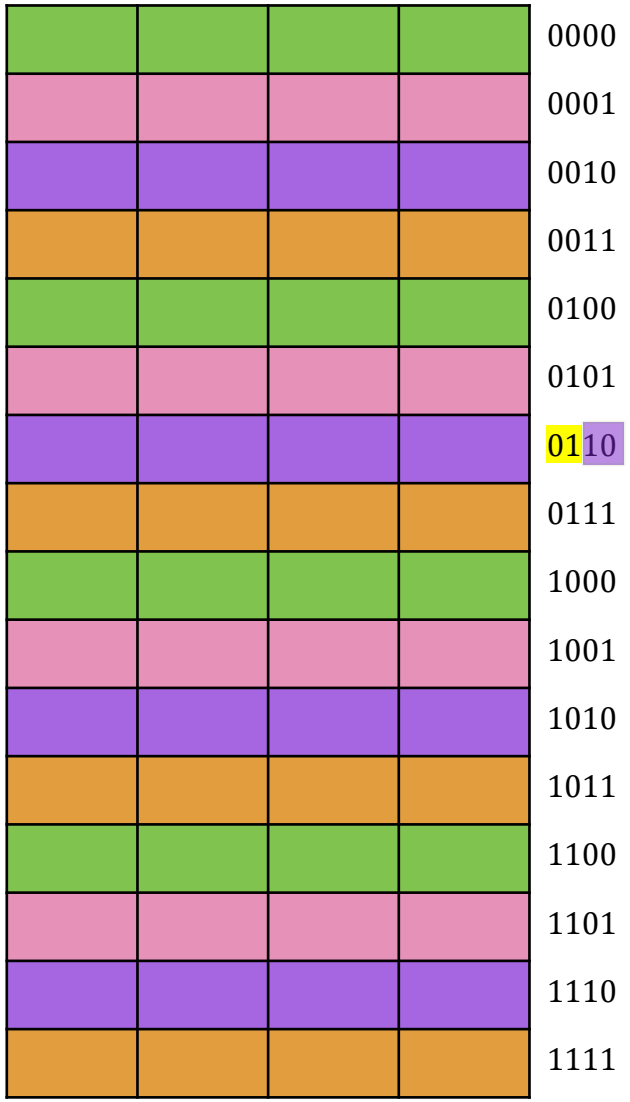
So cache line number is 10

Now CPU checks the tag and that matches

Valid bit is 1

Access by the Processor

Main Memory (16 blocks, 4 words at each block)



Cache (4 lines, 4 words at each line)

Index	Tag	Valid bit	Words			
00	11	1				
01	01	0				
10	01	1				
11	10	1				

Now CPU checks the tag and that matches

Valid bit is 1

So it's a cache hit

How do the CPU Know the Memory Block Number?

Memory size: 64 words

Each block: 4 words

No. of blocks M=64/4=16

Main Memory

w0	w1	w2	w3
w4	w5	w6	w7
w8	w9	w10	w11
w12	w13	w14	w15
w16	w17	w18	w19
w20	w21	w22	w23
w24	w25	w26	w27
w28	w29	w30	w31
w32	w33	w34	w35
w36	w37	w38	w39
w40	w41	w42	w43
w44	w45	w46	w47
w48	w49	w50	w51
w52	w53	w54	w55
w56	w57	w58	w59
w60	w61	w62	w63

Block Number	Binary Representation of block number
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

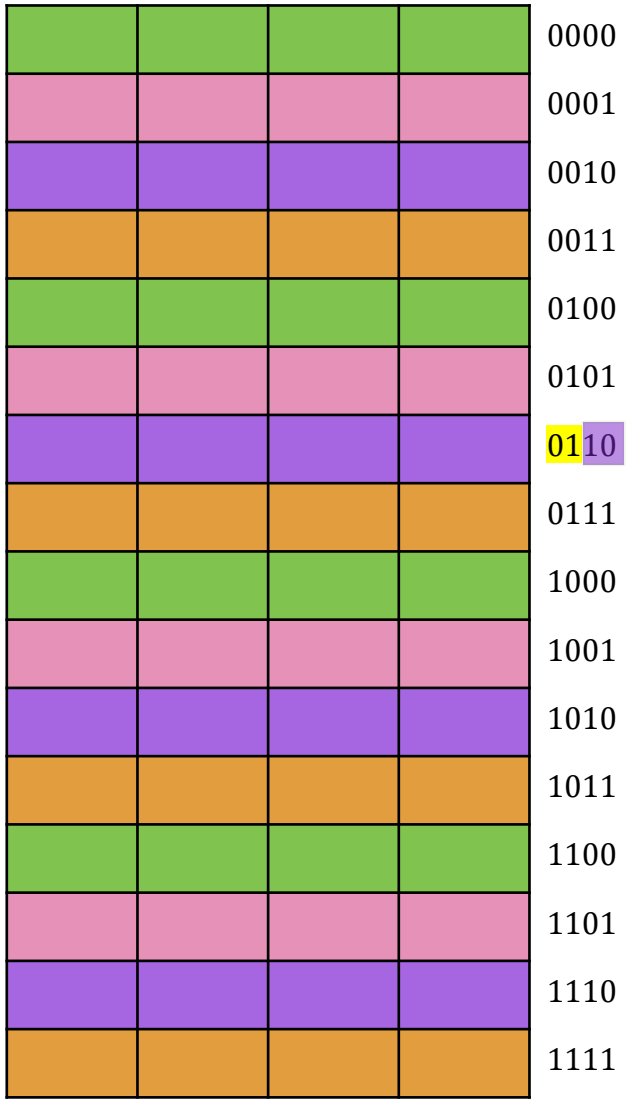
011001

The bottom (LSB) 2 bits ($6 - \log_2 M$ **bits**) indicate the position of the word

Offset

Access by the Processor

Main Memory (16 blocks, 4 words at each block)



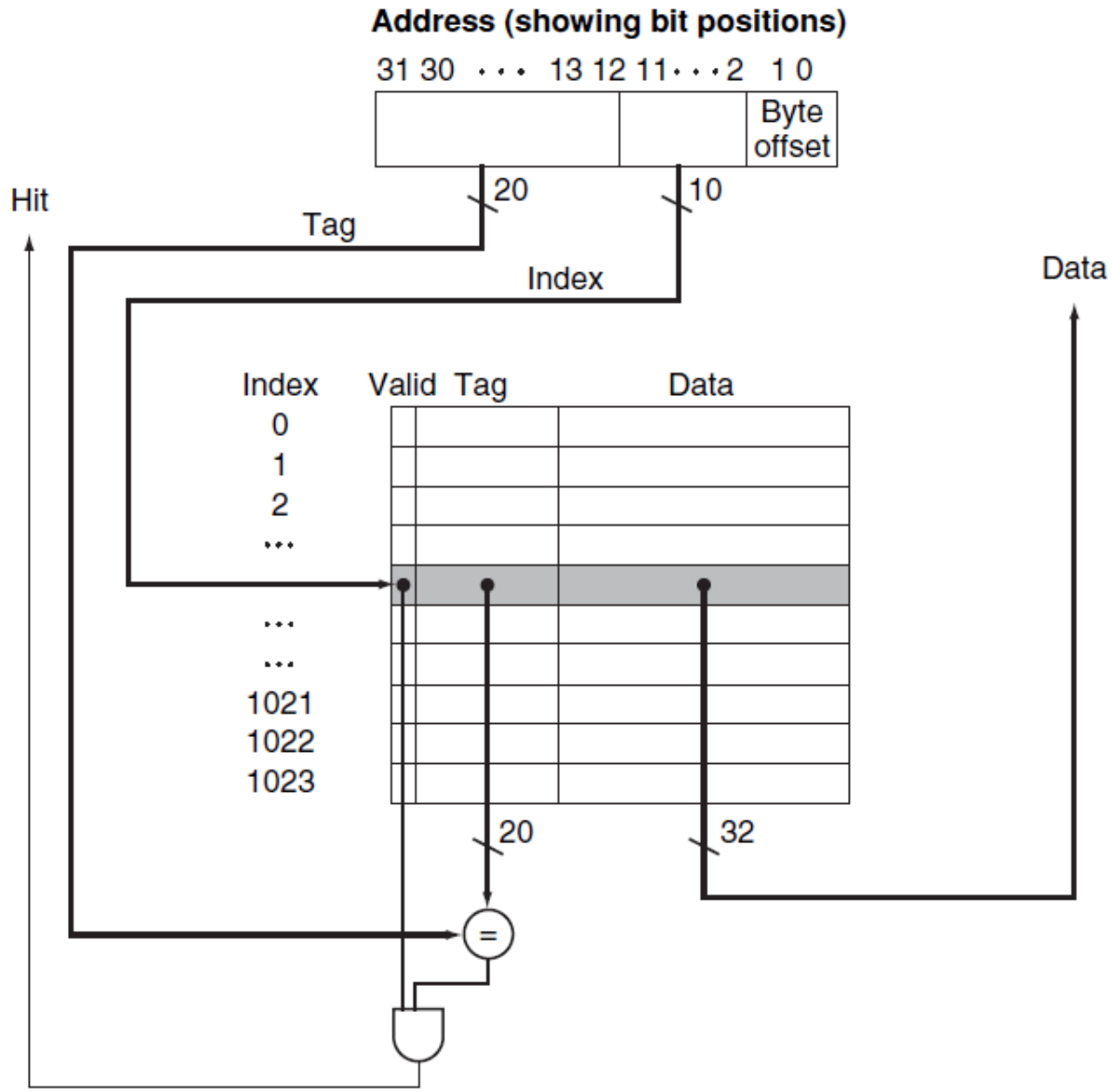
Cache (4 lines, 4 words at each line)

Index	Tag	Valid bit	Words			
00	11	1				
01	01	0				
10	01	1				
11	10	1				

So it's a cache hit

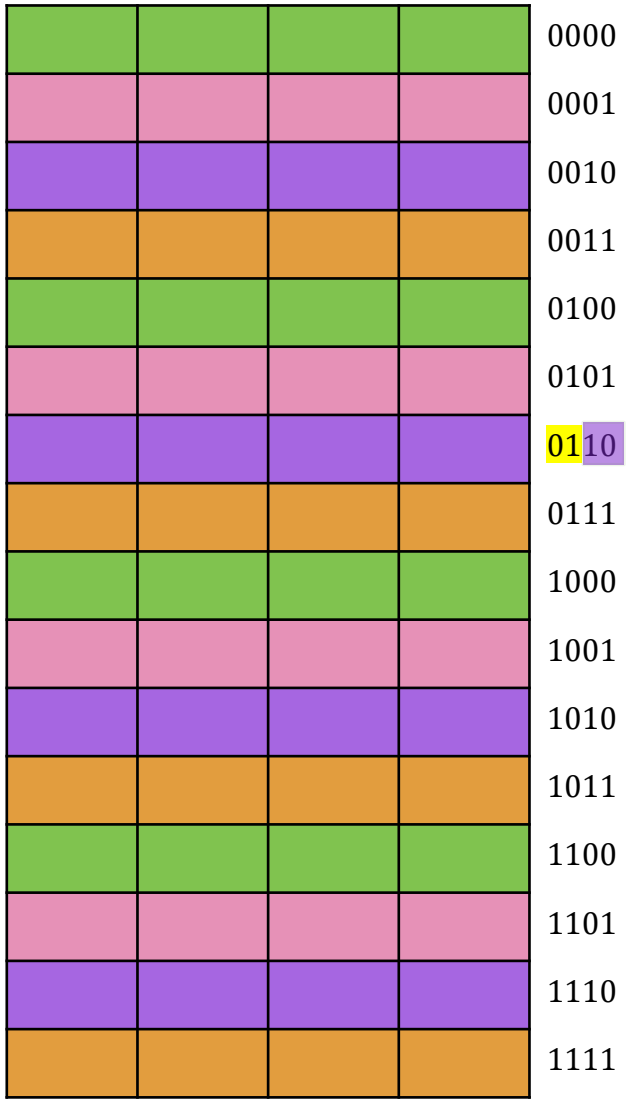
Now CPU use the offset 01 to find the exact word (w25) from cache

Accessing the Cache



Cache Miss

Main Memory (16 blocks, 4 words at each block)

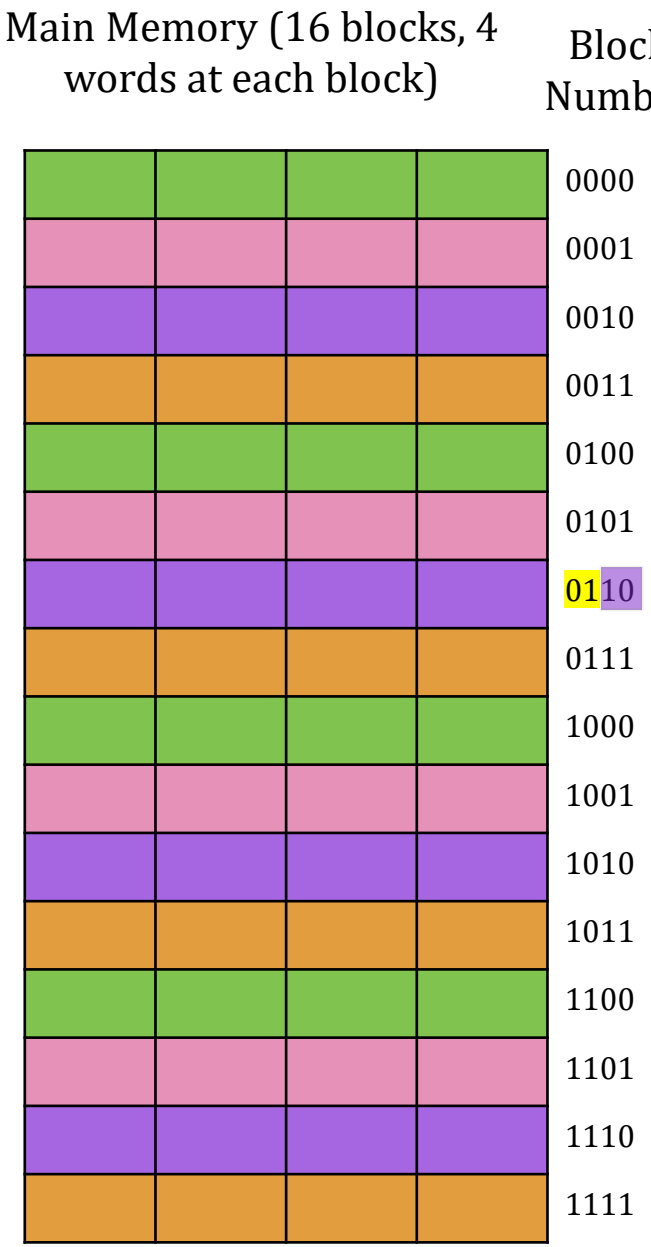


Cache (4 lines, 4 words at each line)

Index	Tag	Valid bit	Words			
00	11	1				
01	01	0				
10	11	1				
11	10	1				

If CPU checks the tag and does not match
It's a cache miss

Cache Miss



Cache (4 lines, 4 words at each line)

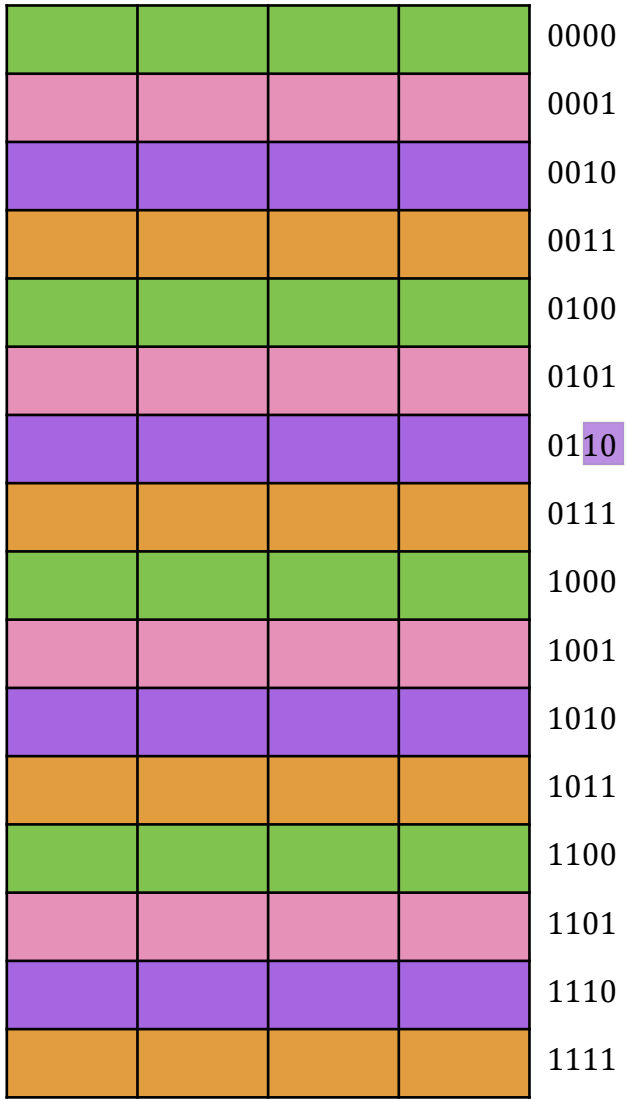
Index	Tag	Valid bit	Words
00	11	1	
01	01	0	
10	01	1	
11	10	1	

If CPU checks the tag and does not match
It's a cache miss

The cache line will be replaced by the appropriate memory block

Cache Miss: Invalid Block

Main Memory (16 blocks, 4 words at each block)

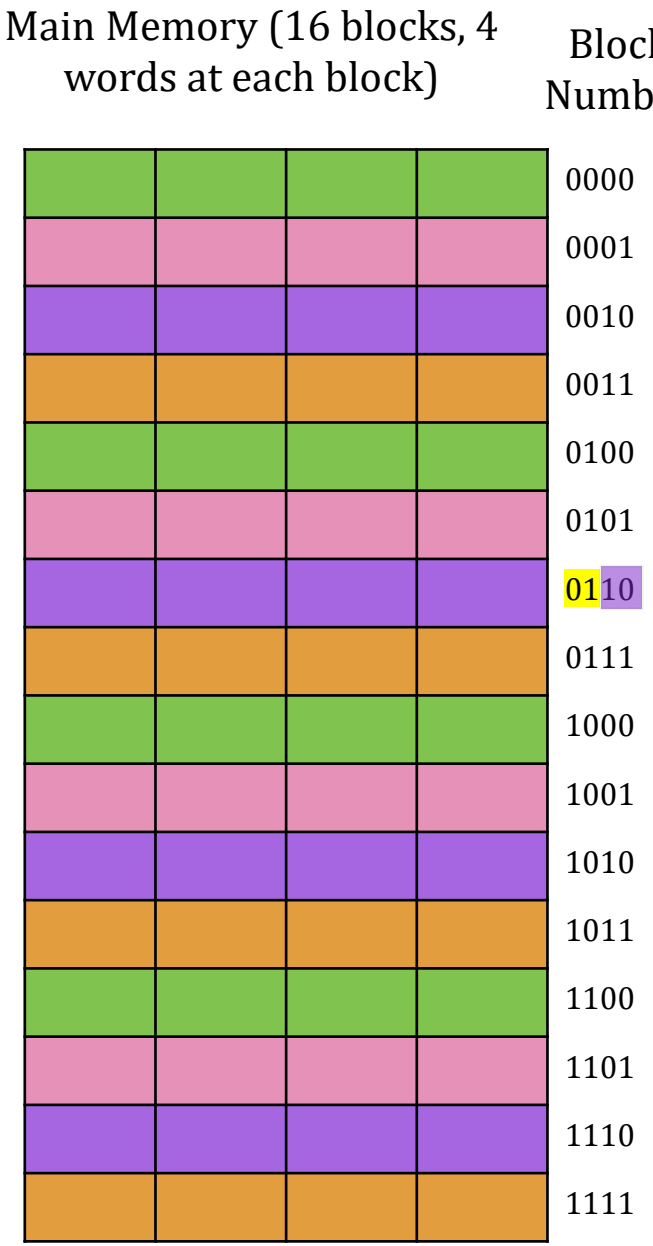


Cache (4 lines, 4 words at each line)

Index	Tag	Valid bit	Words			
00	11	1				
01	01	0				
10	...	0				
11	10	1				

If valid bit is 0, the data is invalid
So, it's a cache miss

Cache Miss: Invalid Block



Cache (4 lines, 4 words at each line)

Index	Tag	Valid bit	Words
00	11	1	
01	01	0	
10	01	1	
11	10	1	

If valid bit is 0, the data is invalid
So, it's a cache miss

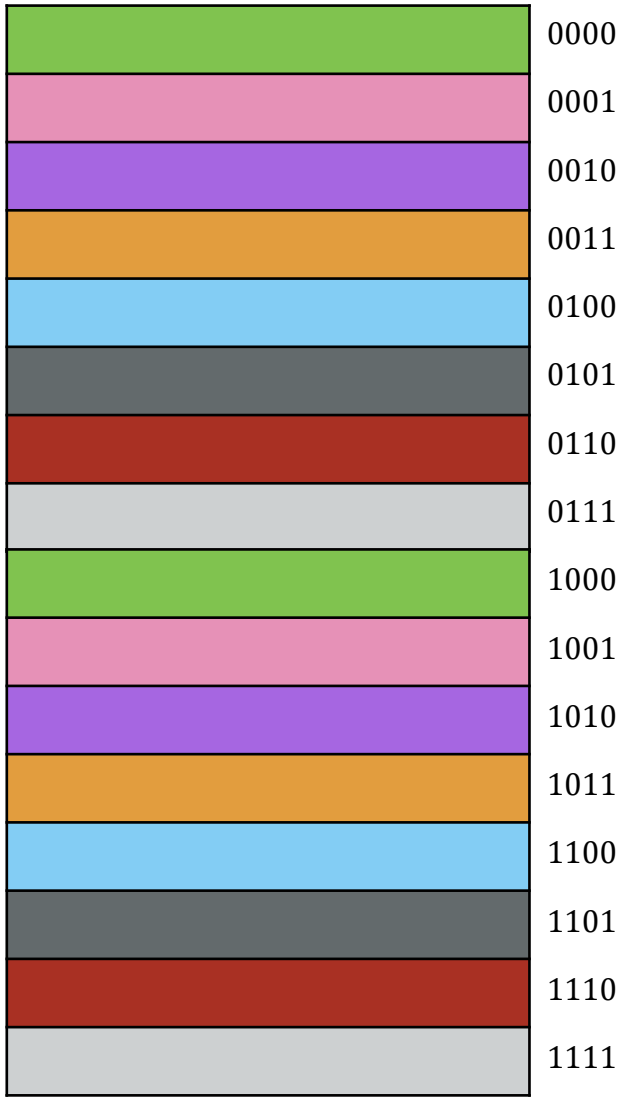
The cache line will be replaced by the appropriate memory block

Associativity

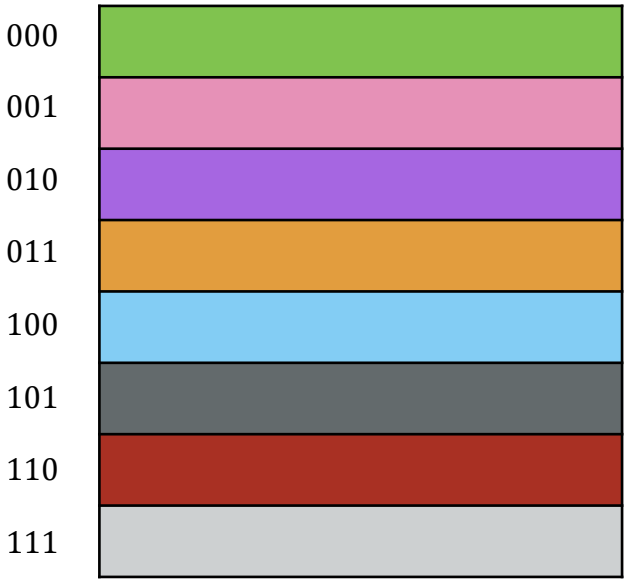
- Direct mapping is rigid
- (set) Associativity means providing more than one place for a cache line to live.
- The level of associativity is the number of possible locations
  - 2-way set associative
  - 4-way set associative
- One group of lines corresponds to each index: a set
- Each line in a set is called a “way”

Direct Mapping

Main Memory
(16 blocks)



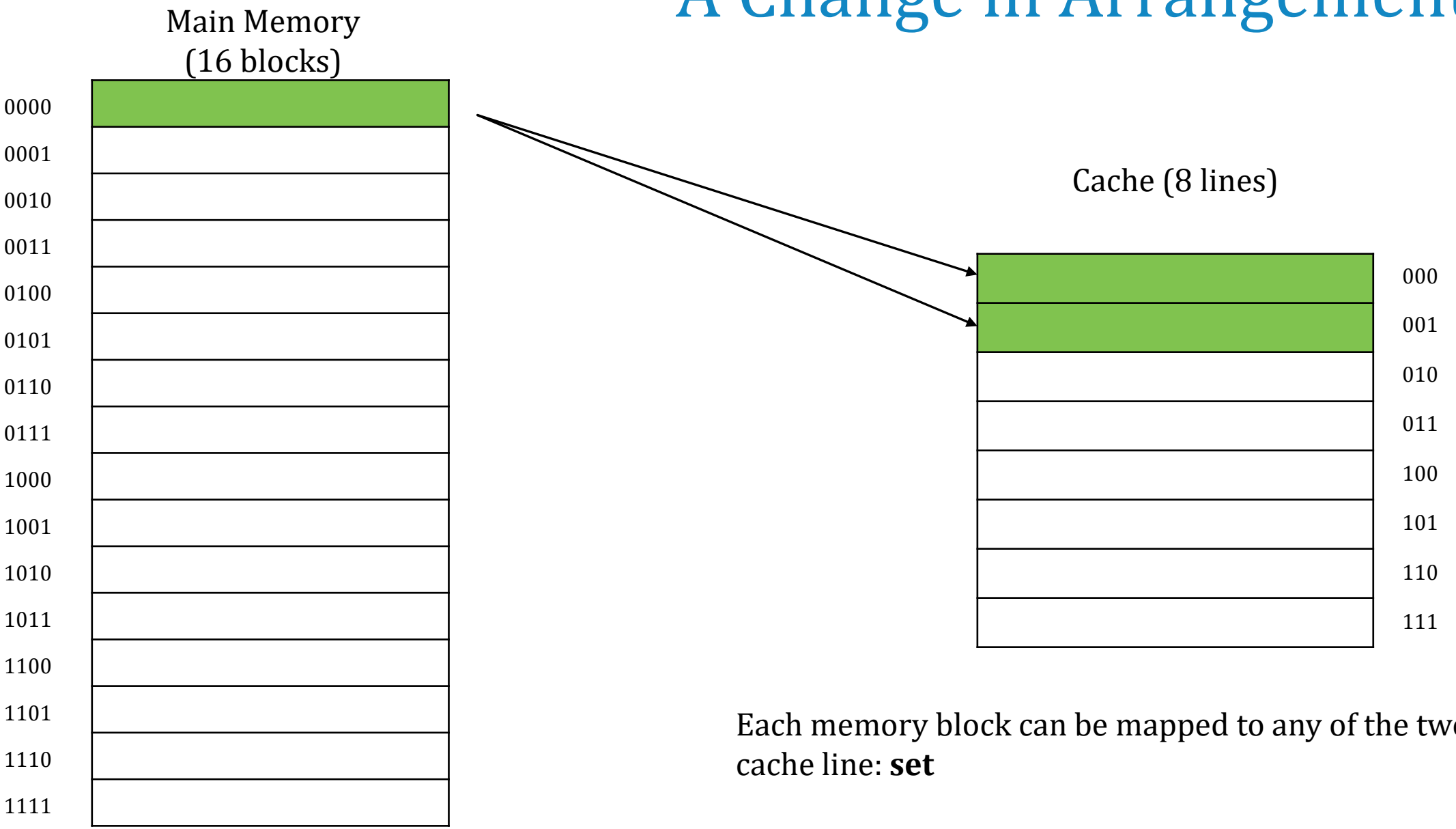
Cache (8 lines)



One cache line for several memory block

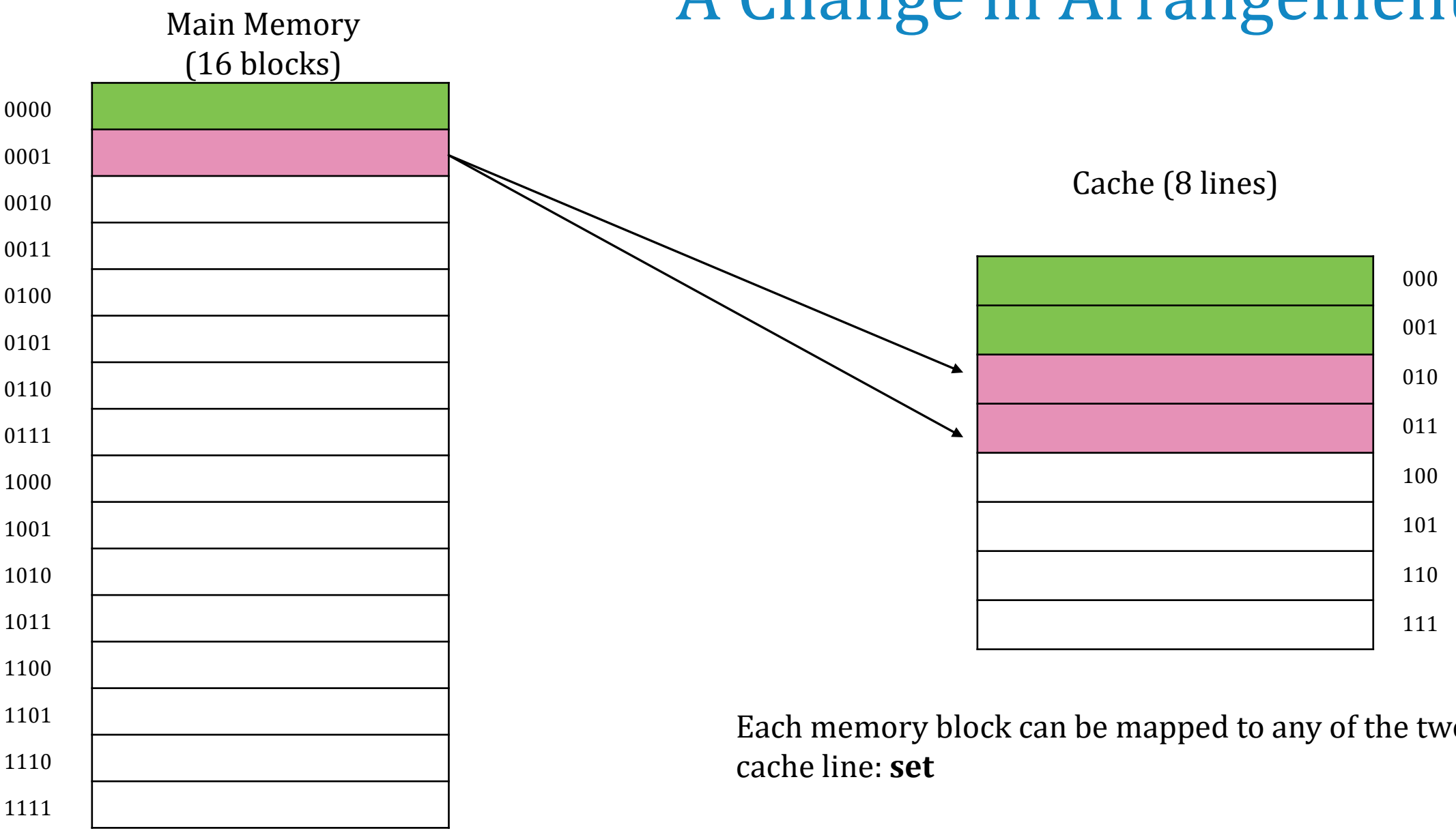
Frequent replacement

A Change in Arrangement



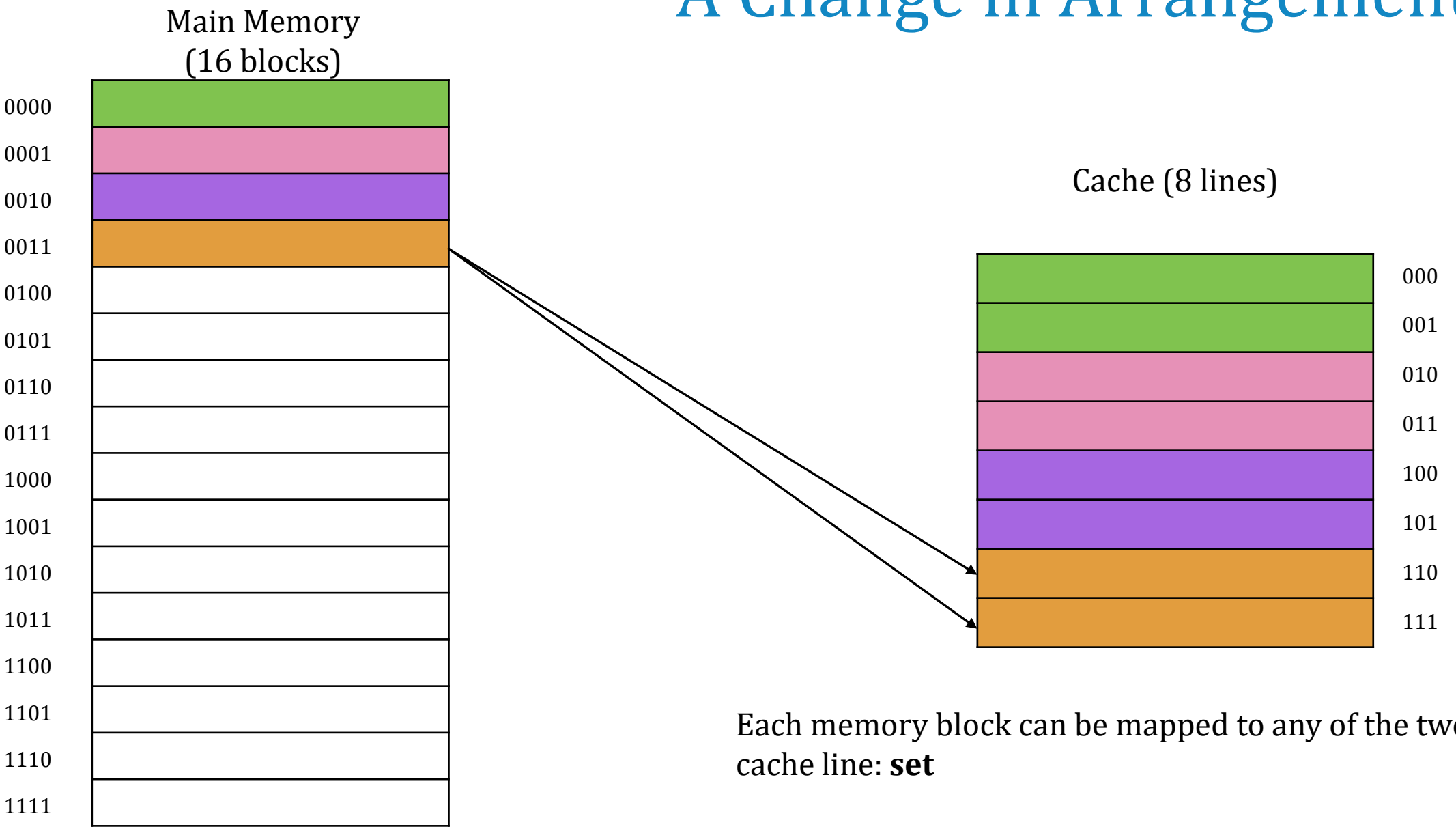
Each memory block can be mapped to any of the two possible cache line: **set**

A Change in Arrangement



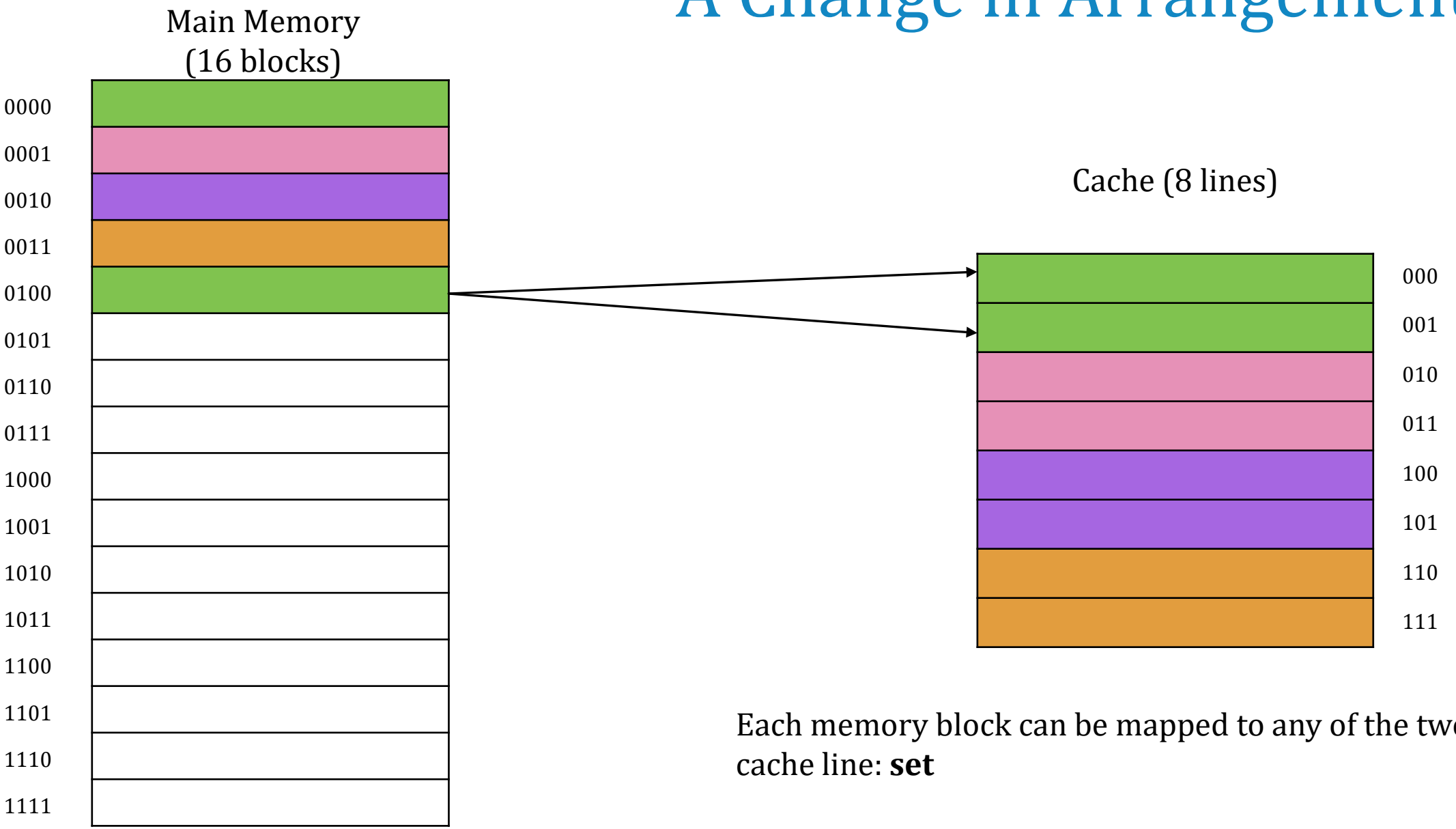
Each memory block can be mapped to any of the two possible cache line: **set**

A Change in Arrangement



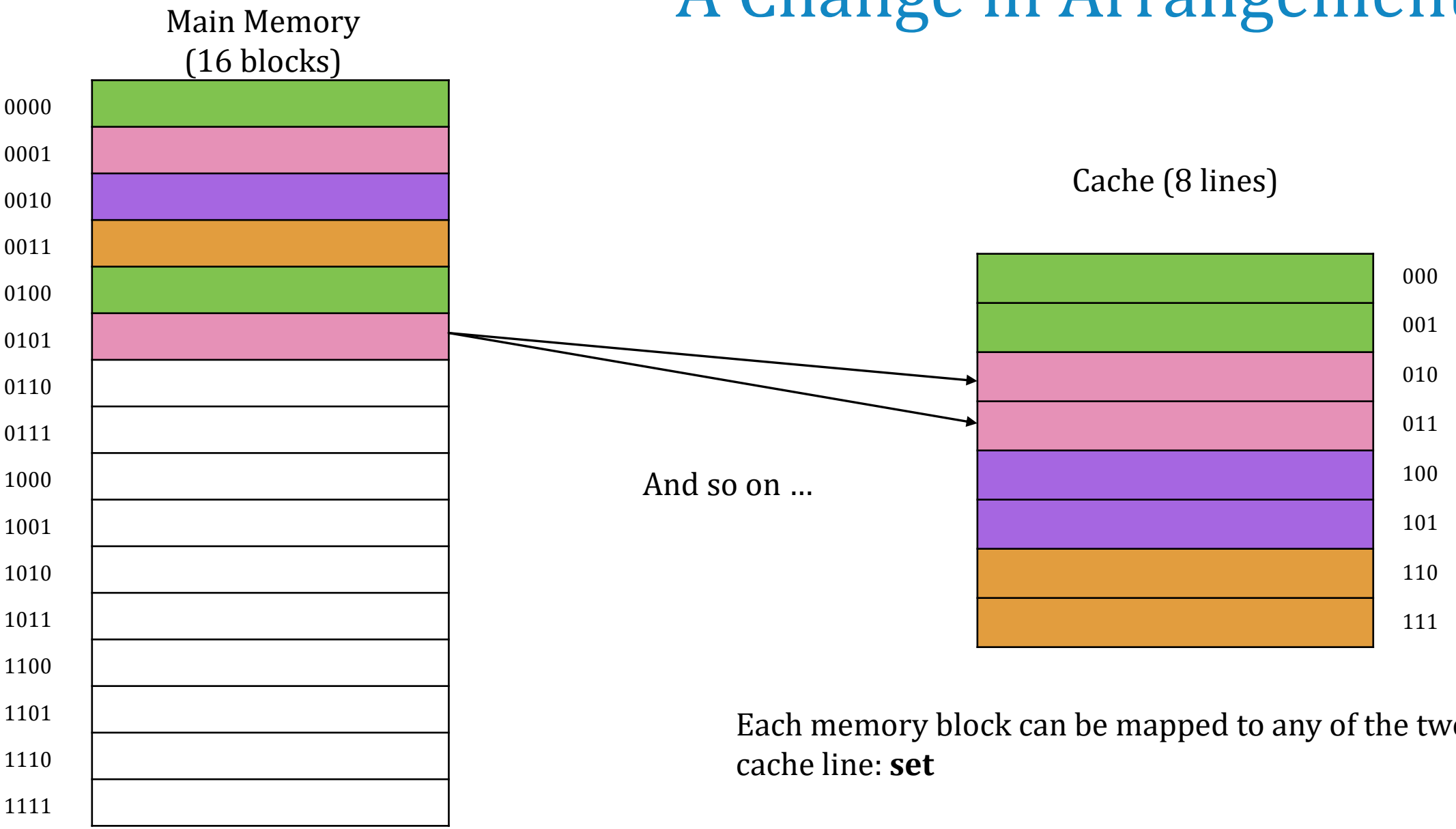
Each memory block can be mapped to any of the two possible cache line: **set**

A Change in Arrangement



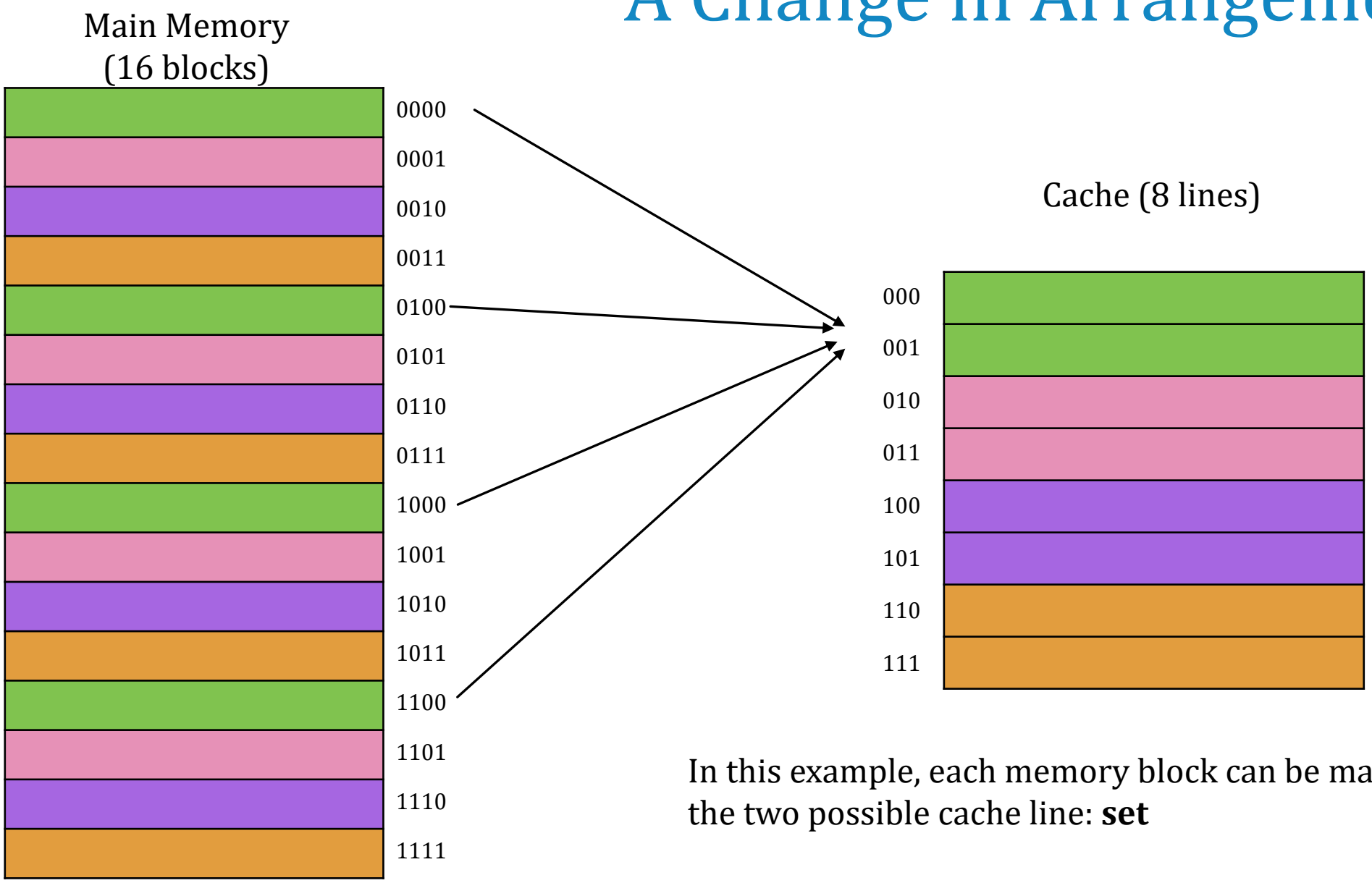
Each memory block can be mapped to any of the two possible cache line: **set**

A Change in Arrangement



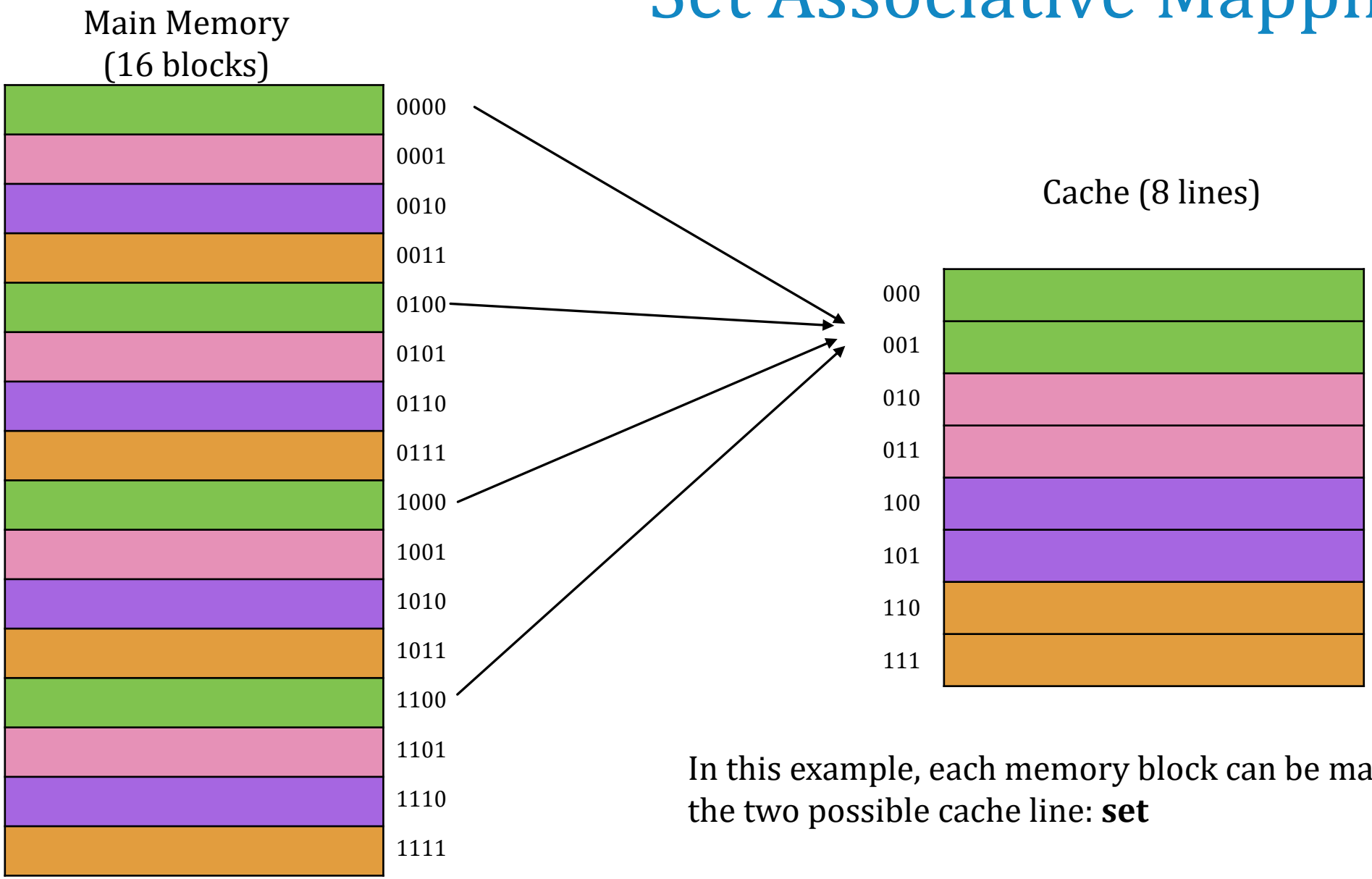
Each memory block can be mapped to any of the two possible cache line: **set**

A Change in Arrangement



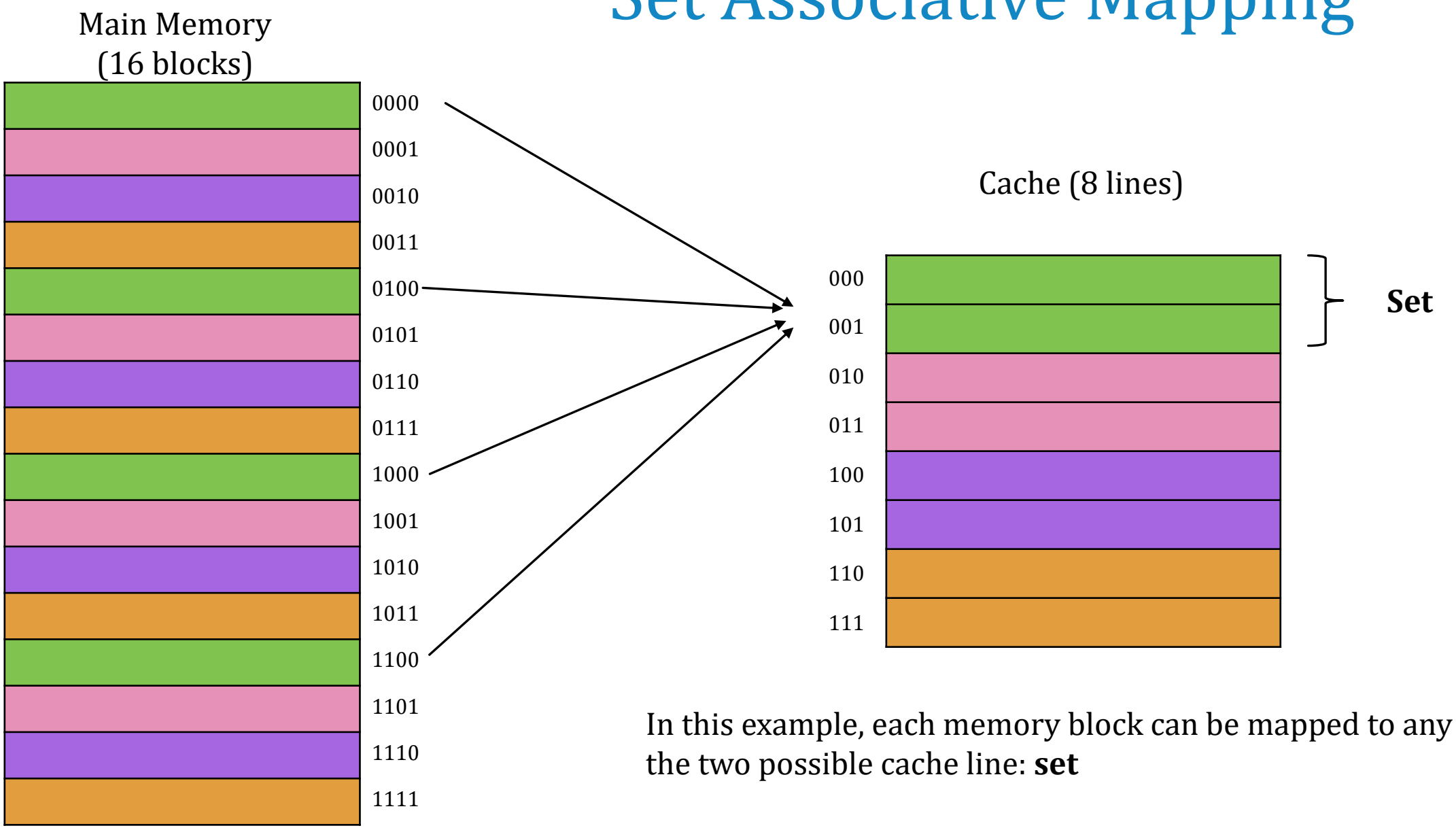
In this example, each memory block can be mapped to any of the two possible cache line: **set**

Set Associative Mapping



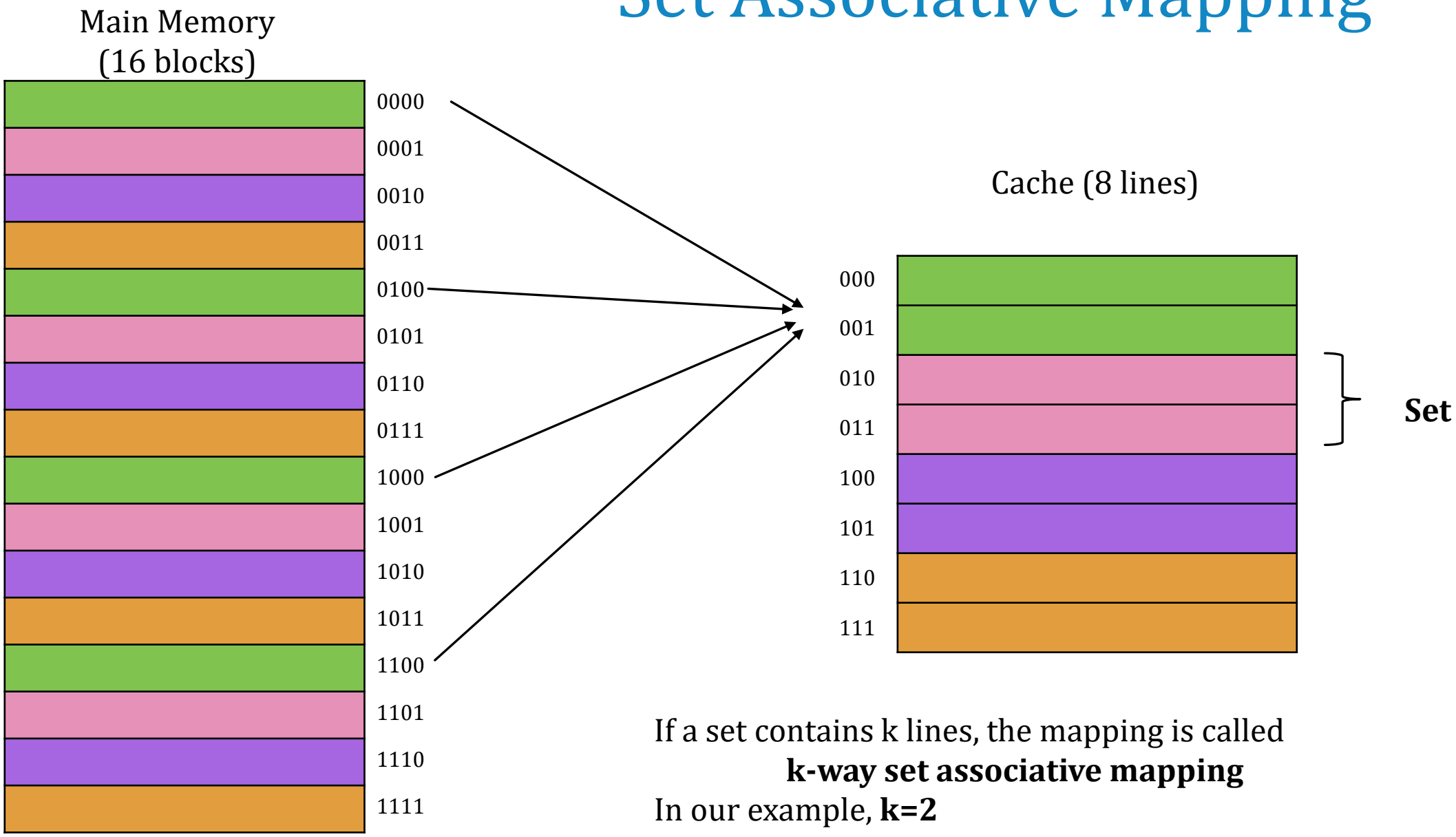
In this example, each memory block can be mapped to any of the two possible cache line: **set**

Set Associative Mapping



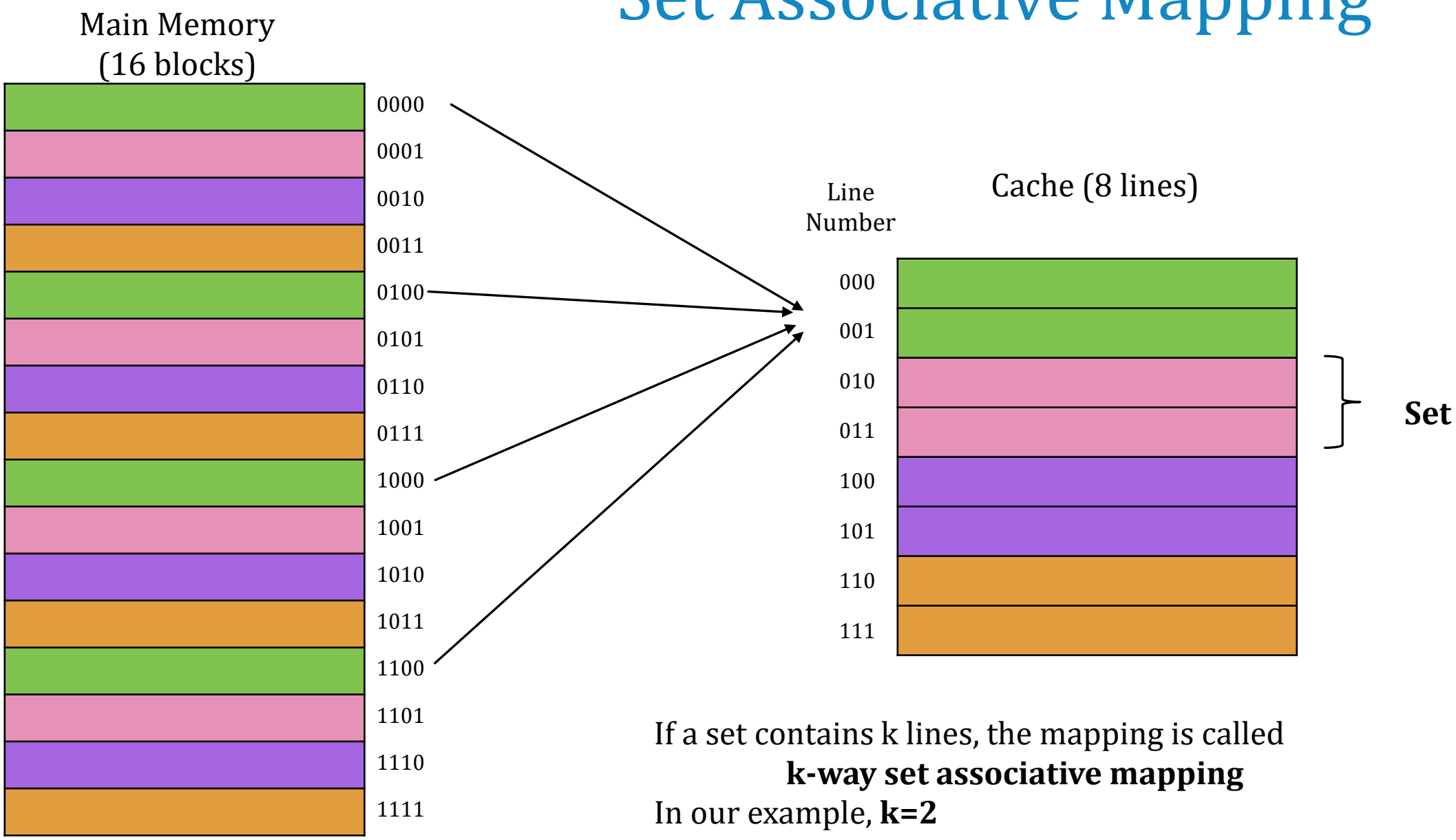
In this example, each memory block can be mapped to any of the two possible cache line: **set**

Set Associative Mapping



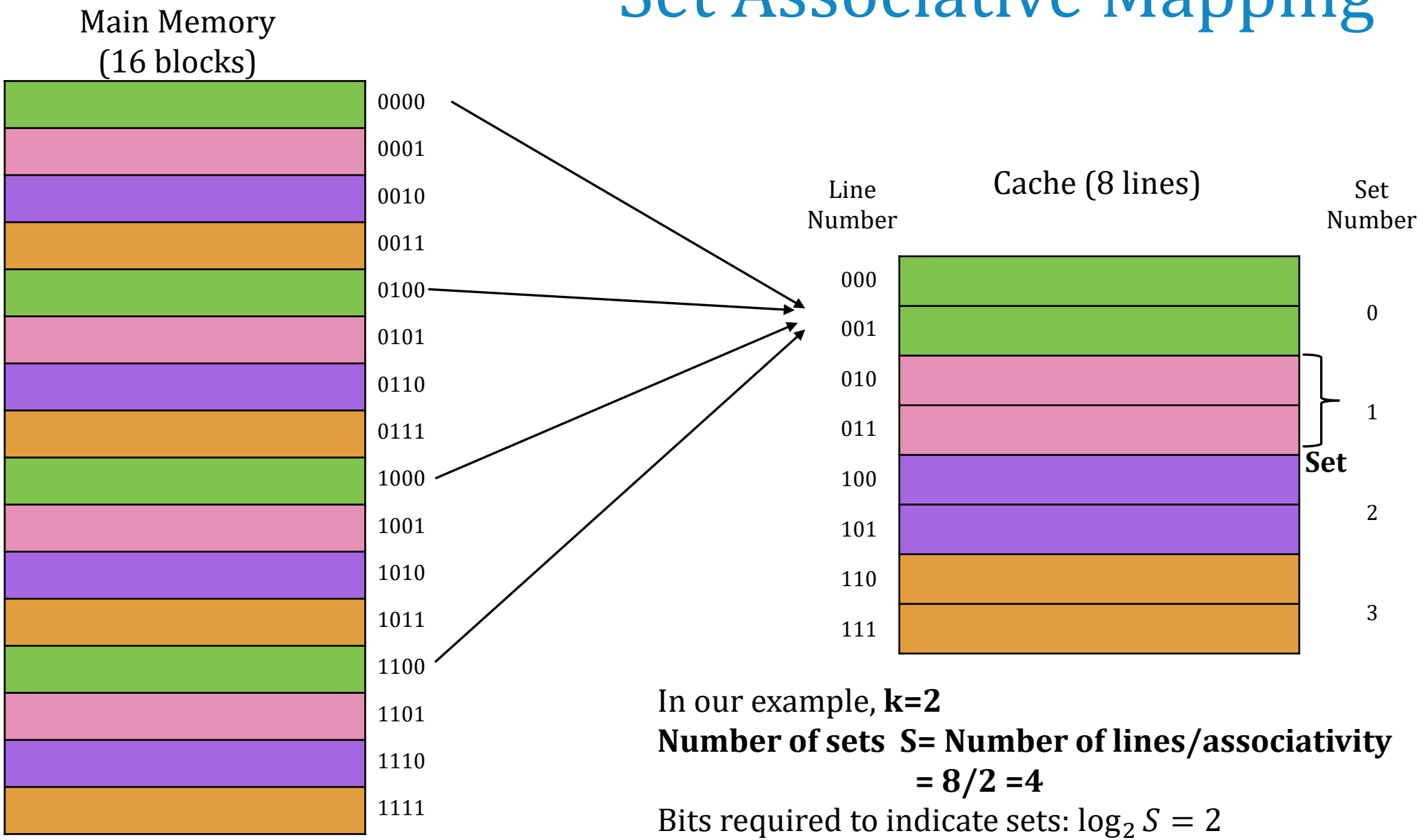
If a set contains k lines, the mapping is called
k-way set associative mapping
In our example, **k=2**

Set Associative Mapping



If a set contains k lines, the mapping is called
k-way set associative mapping
In our example, **k=2**

Set Associative Mapping



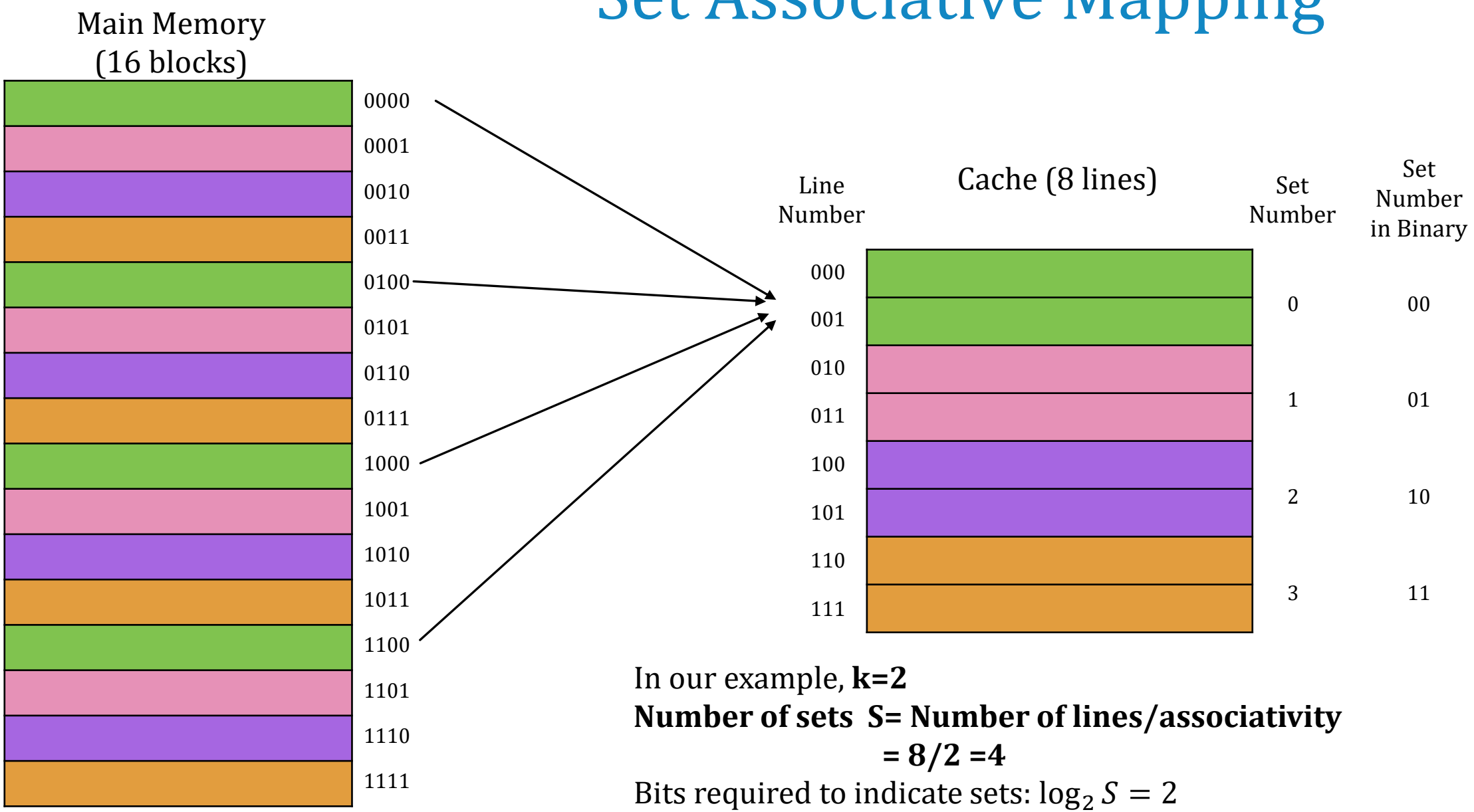
In our example, **k=2**

Number of sets S= Number of lines/associativity

= 8/2 =4

Bits required to indicate sets: $\log_2 S = 2$

Set Associative Mapping



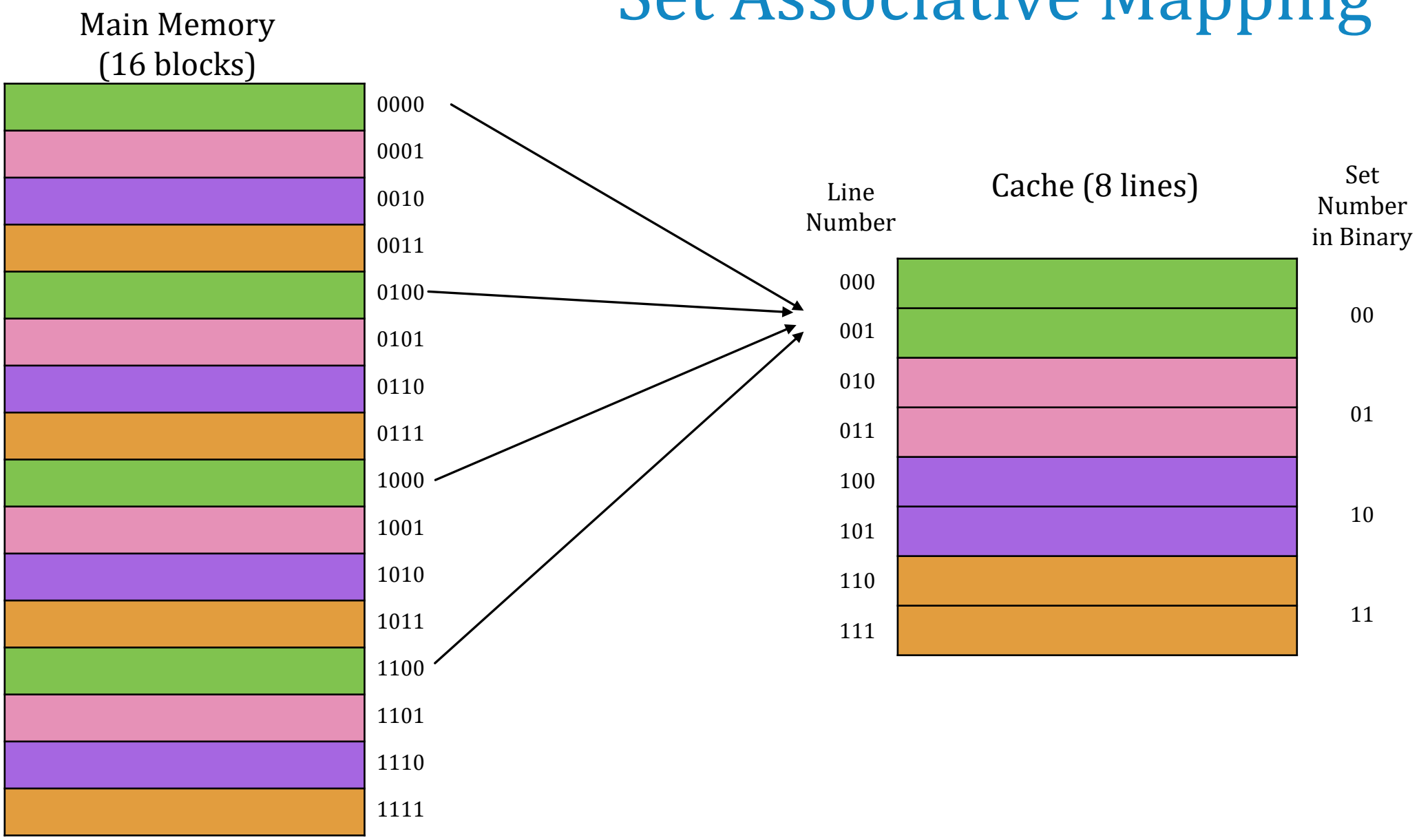
In our example, **k=2**

Number of sets S= Number of lines/associativity

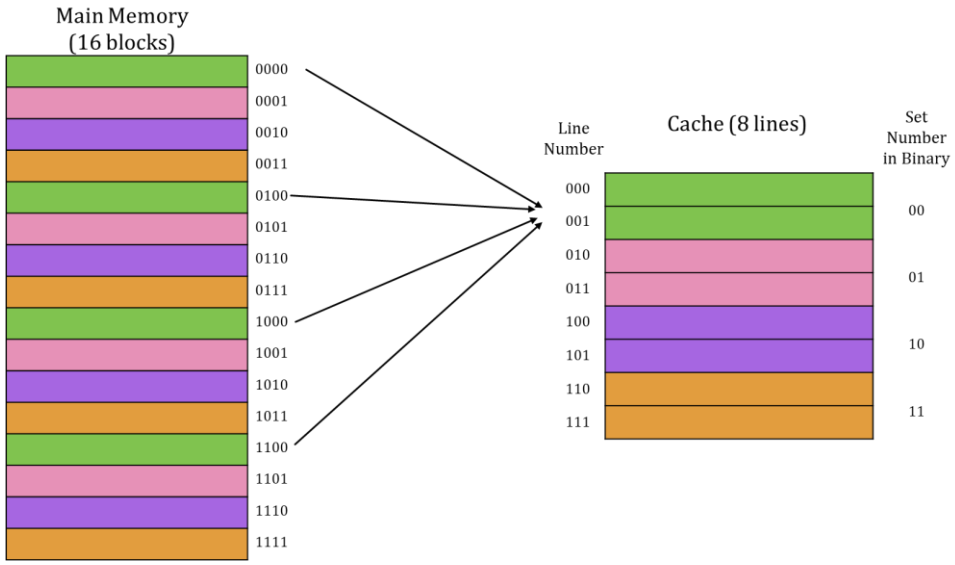
= 8/2 =4

Bits required to indicate sets: $\log_2 S = 2$

Set Associative Mapping



Set Associative Mapping



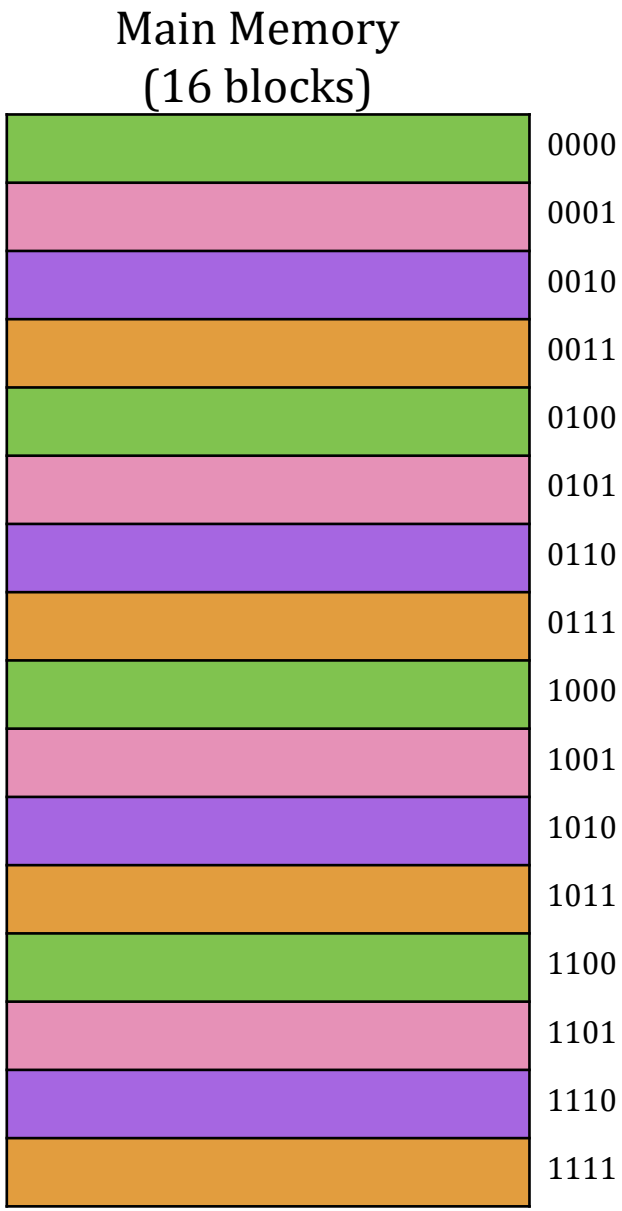
- Main memory: 16 block
- Each block: 4 words
- Total 64 words
- Address size: 6 bit (MSB 4 bits for block address, and LSB 2 bits for offset)

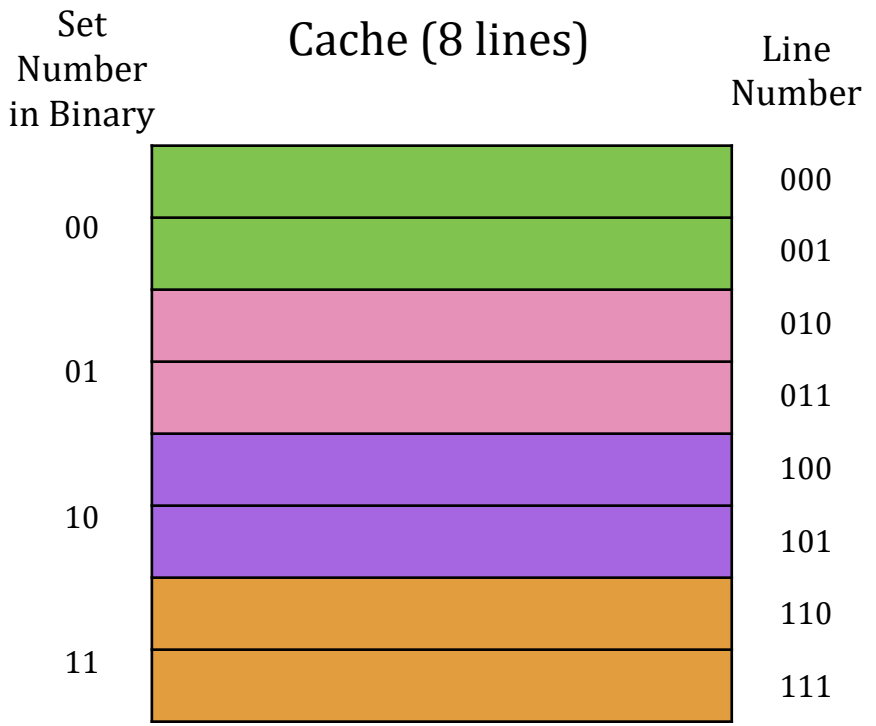
- Cache line L=8
- Bits required for line number (b_L): 3
- Number of sets S=4
- Bits required to indicate any set (b_S) =2

- Number of bits to indicate a line in a set is

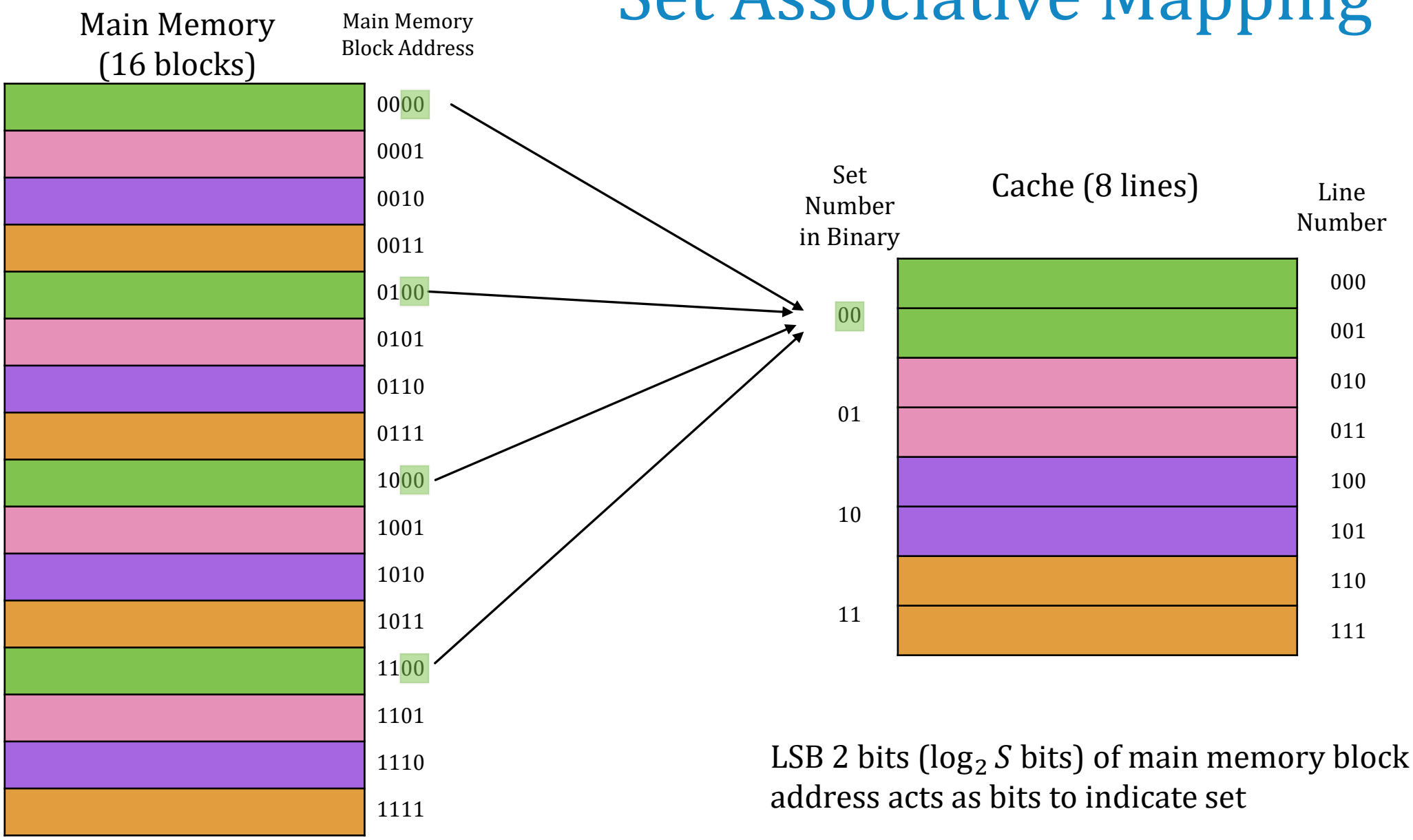
$$b_L - b_S = 1$$

Set Associative Mapping



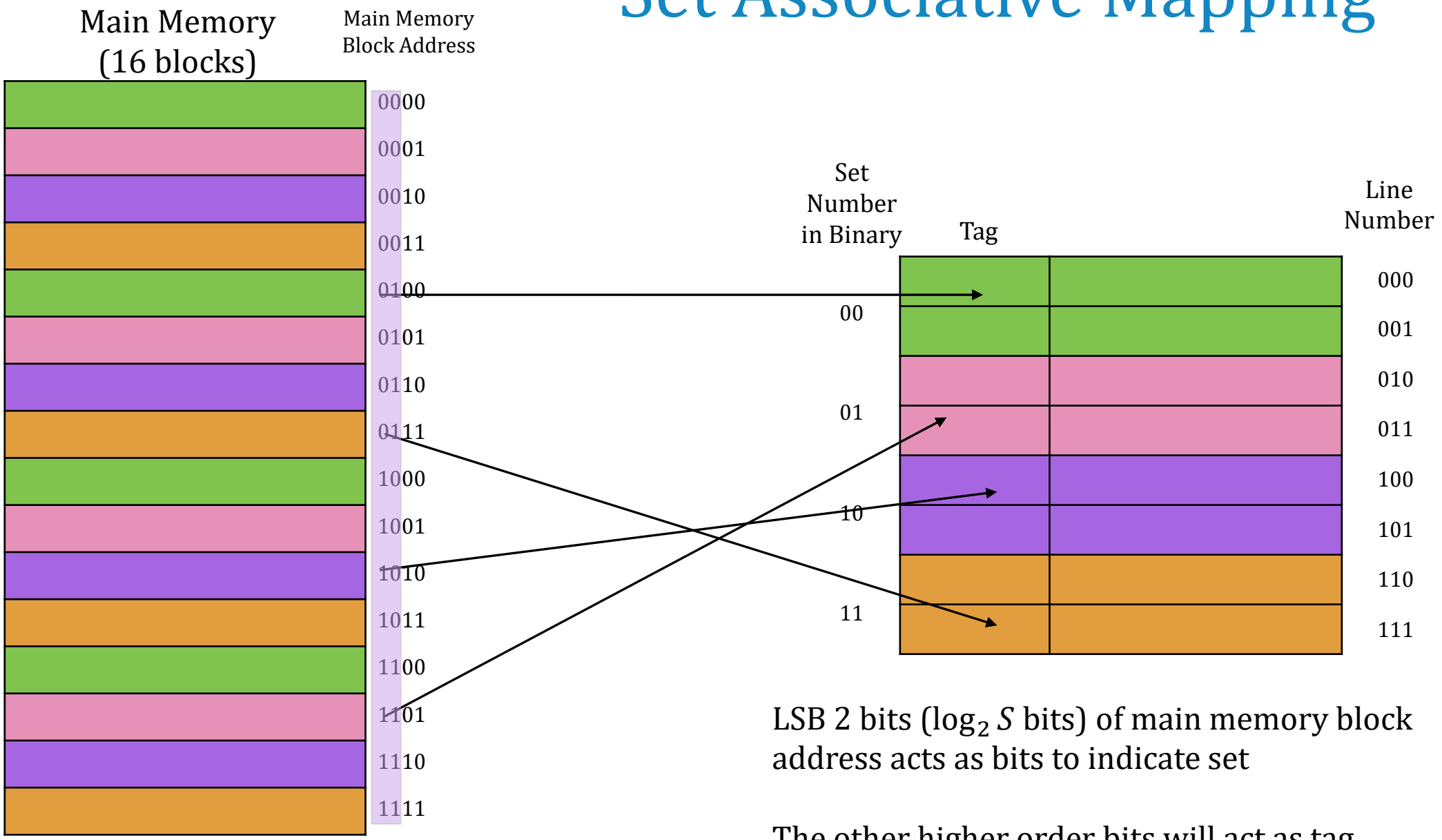


Set Associative Mapping



LSB 2 bits ($\log_2 S$ bits) of main memory block address acts as bits to indicate set

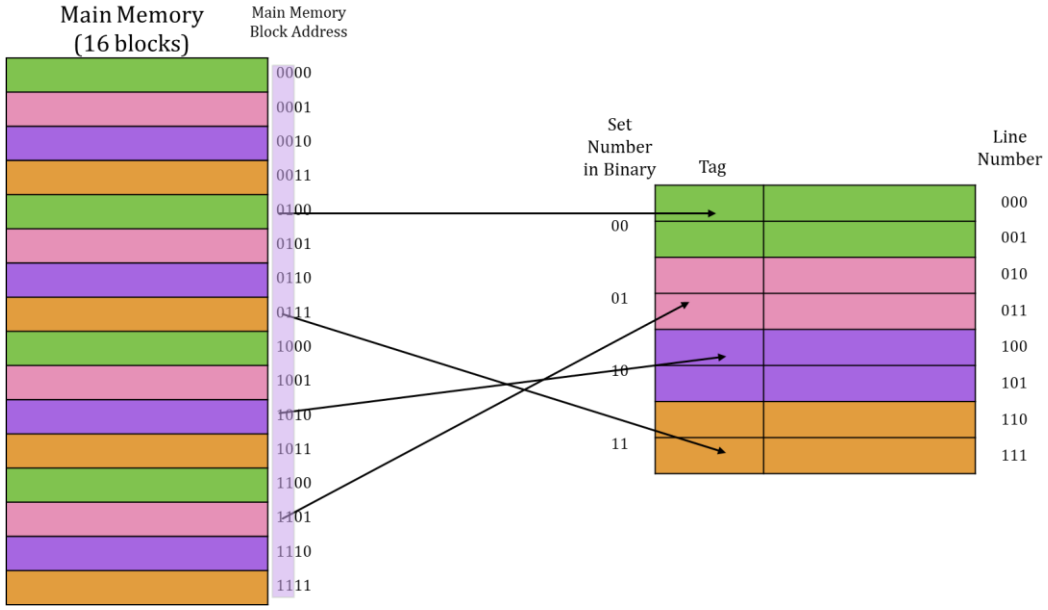
Set Associative Mapping



LSB 2 bits ($\log_2 S$ bits) of main memory block address acts as bits to indicate set

The other higher order bits will act as tag

Set Associative Mapping

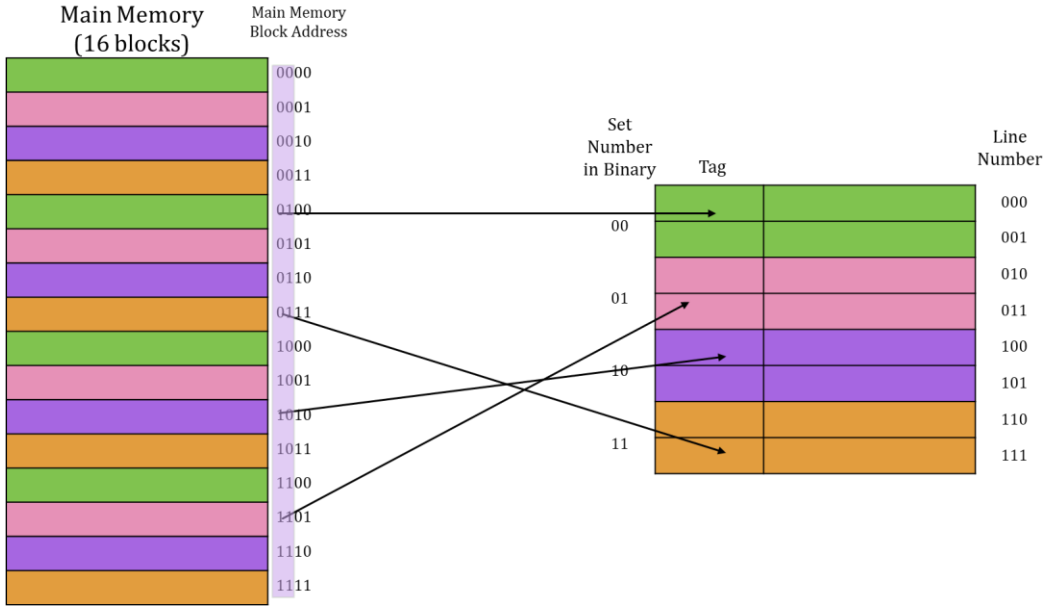


- Address size generated by CPU: 6 bit

(MSB 4 bits for block address, and LSB 2 bits for offset)

- Cache line L=8
- Associativity k=2
- Number of sets S = L/k =4
- Bits required to indicate any set $b_S = \log_2 S = \log_2(\frac{L}{k}) = 2$
  - set bit/ index bit
  - LSB b_S (2 in our case) bits of memory block address
- Tag:
  memory block address length (4 in this example) $-\log_2 S$
  LSB bits of memory block address

Set Associative Mapping



- Address size generated by CPU: 6 bit

(MSB 4 bits for block address, and LSB 2 bits for offset)

- Cache line L=8
- Associativity k=2
- Number of sets S = L/k =4
- Bits required to indicate any set $b_S = \log_2 S = \log_2(\frac{L}{k}) = 2$
  - set bit/ index bit
  - LSB b_S (2 in our case) bits of memory block address
- Tag:
  memory block address length (4 in this example) $-\log_2 S$
  LSB bits of memory block address

=Total address bits (6 in this example) - LSBs for offset (2 in this example) $-\log_2 S$ LSBs after offset bits

Accessing the Set Associative Cache: Example

- Cache lines = L
- Cache line size = N (Number of smallest addressable units in a block)
- Address length = A = 32 bit
- Associativity = k
- Index bits = $\log_2$ (L/k)
- Offset bits = $\log_2$ (N)
- Tag bits = A - (index bits + offset bits)

32KiB, 2048 Lines/ Blocks, 4-way associative cache

Line/ block size:?

Sets:
Index bits:

Offset bits:
Tag bits:

Accessing the Set Associative Cache: Example

- Cache lines = L
- Cache line size = N (Number of smallest addressable units in a block)
- Address length = A = 32 bit
- Associativity = k
- Index bits = $\log_2$ (L/k)
- Offset bits = $\log_2$ (N)
- Tag bits = A - (index bits + offset bits)

32KiB, 2048 Lines/ Blocks, 4-way associative cache

Line/ block size =(32×1024)B/2048 = 16B

Sets: ?
Index bits: ?

Offset bits:
Tag bits:

Accessing the Set Associative Cache: Example

- Cache lines = L
- Cache line size = N (Number of smallest addressable units in a block)
- Address length = A = 32 bit
- Associativity = k
- Index bits = $\log_2$ (L/k)
- Offset bits = $\log_2$ (N)
- Tag bits = A - (index bits + offset bits)

32KiB, 2048 Lines/ Blocks, 4-way associative cache

Line/ block size =$(32 \times 1024)B/2048 = 16B$

Sets: 2048/4=512
Index bits: ?

Offset bits:
Tag bits:

Accessing the Set Associative Cache: Example

- Cache lines = L
- Cache line size = N (Number of smallest addressable units in a block)
- Address length = A = 32 bit
- Associativity = k
- Index bits = $\log_2$ (L/k)
- Offset bits = $\log_2$ (N)
- Tag bits = A - (index bits + offset bits)

32KiB, 2048 Lines/ Blocks, 4-way associative cache

Line/ block size =$(32 \times 1024)B/2048 = 16B$

Sets: 512
Index bits: 9

Offset bits:
Tag bits:

Accessing the Set Associative Cache: Example

- Cache lines = L
- Cache line size = N (Number of smallest addressable units in a block)
- Address length = A = 32 bit
- Associativity = k
- Index bits = $\log_2$ (L/k)
- Offset bits = $\log_2$ (N)
- Tag bits = A - (index bits + offset bits)

32KiB, 2048 Lines/ Blocks, 4-way associative cache

Line/ block size =$(32 \times 1024)B/2048 = 16B$

Sets: 512
Index bits: 9

Offset bits: depends on addressability. Assume that the memory and the cache is byte addressable

Tag bits:

Accessing the Set Associative Cache: Example

- Cache lines = L
- Cache line size = N (Number of smallest addressable units in a block)
- Address length = A = 32 bit
- Associativity = k
- Index bits = $\log_2$ (L/k)
- Offset bits = $\log_2$ (N)
- Tag bits = A - (index bits + offset bits)

32KiB, 2048 Lines/ Blocks, 4-way associative cache

Line/ block size =$(32 \times 1024)B/2048 = 16B$

Sets: 512
Index bits: 9

Offset bits: 4

Tag bits:

Accessing the Set Associative Cache: Example

- Cache lines = L
- Cache line size = N (Number of smallest addressable units in a block)
- Address length = A = 32 bit
- Associativity = k
- Index bits = $\log_2$ (L/k)
- Offset bits = $\log_2$ (N)
- Tag bits = A - (index bits + offset bits)

32KiB, 2048 Lines/ Blocks, 4-way associative cache

Line/ block size =$(32 \times 1024)B/2048 = 16B$

Sets: 512
Index bits: 9

Offset bits: 4

Tag bits: Total address length – no. of index bits – no. of offset bits

Accessing the Set Associative Cache: Example

- Cache lines = L
- Cache line size = N (Number of smallest addressable units in a block)
- Address length = A = 32 bit
- Associativity = k
- Index bits = $\log_2$ (L/k)
- Offset bits = $\log_2$ (N)
- Tag bits = A - (index bits + offset bits)

32KiB, 2048 Lines/ Blocks, 4-way associative cache

Line/ block size =$(32 \times 1024)B/2048 = 16B$

Sets: 512
Index bits: 9

Offset bits: 4
Tag bits: 19

Accessing the Set Associative Cache: Example

- Cache lines = L
- Cache line size = N (Number of smallest addressable units in a block)
- Address length = A = 32 bit
- Associativity = k
- Index bits = $\log_2$ (L/k)
- Offset bits = $\log_2$ (N)
- Tag bits = A - (index bits + offset bits)

32KiB, 2048 Lines/ Blocks, 4-way associative cache

Line/ block size =$(32 \times 1024)B/2048 = 16B$

Sets: 512
Index bits: 9

Offset bits: depends on addressability. Assume that the memory and the cache is word addressable

Tag bits:

Accessing the Set Associative Cache: Example

- Cache lines = L
- Cache line size = N (Number of smallest addressable units in a block)
- Address length = A = 32 bit
- Associativity = k
- Index bits = $\log_2$ (L/k)
- Offset bits = $\log_2$ (N)
- Tag bits = A - (index bits + offset bits)

32KiB, 2048 Lines/ Blocks, 4-way associative cache

Line/ block size =$(32 \times 1024)B/2048 = 16B$

Sets: 512
Index bits: 9

Offset bits: 2 (since number of words in a line = line size/word size =16B/ 4B=4)

Tag bits:

Accessing the Set Associative Cache: Example

- Cache lines = L
- Cache line size = N (Number of smallest addressable units in a block)
- Address length = A = 32 bit
- Associativity = k
- Index bits = $\log_2$ (L/k)
- Offset bits = $\log_2$ (N)
- Tag bits = A - (index bits + offset bits)

32KiB, 2048 Lines/ Blocks, 4-way associative cache

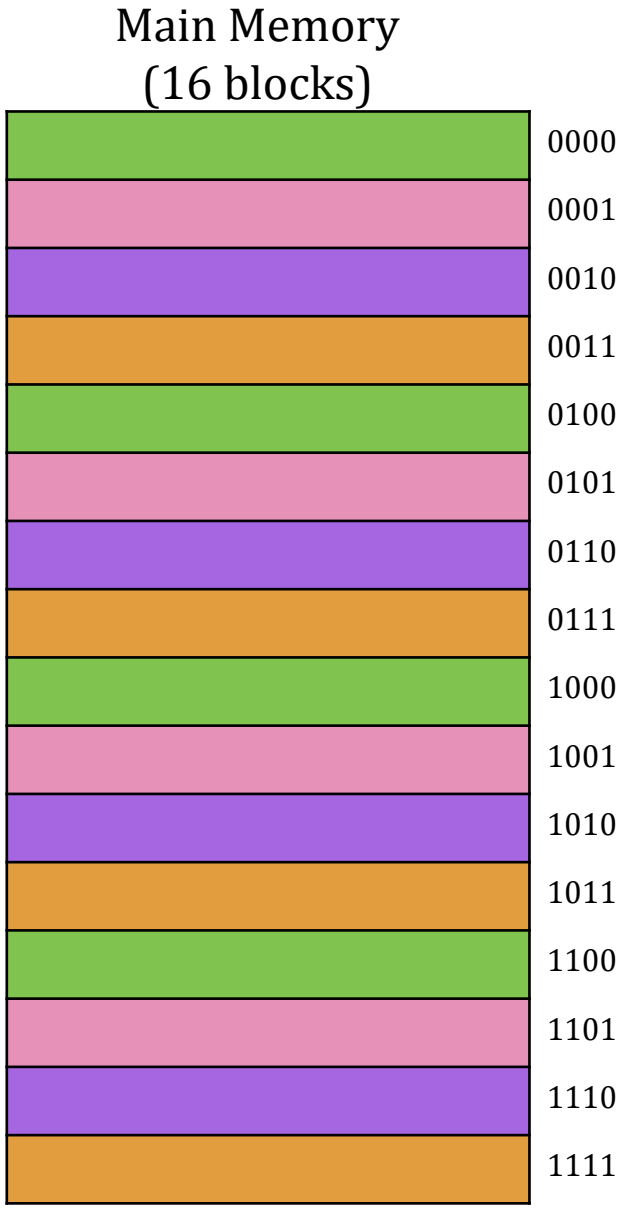
Line/ block size =$(32 \times 1024)B/2048 = 16B$

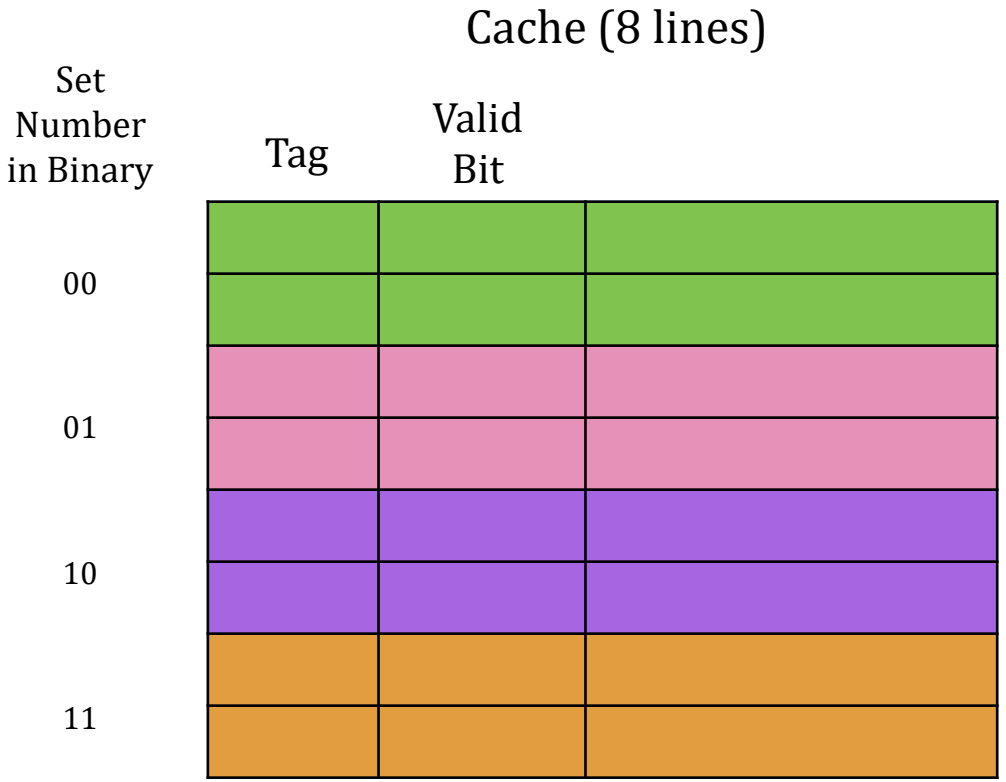
Sets: 512
Index bits: 9

Offset bits: 2 (since number of words in a line = line size/word size =16B/ 4B=4)

Tag bits: 21

Accessing the Set Associative Cache

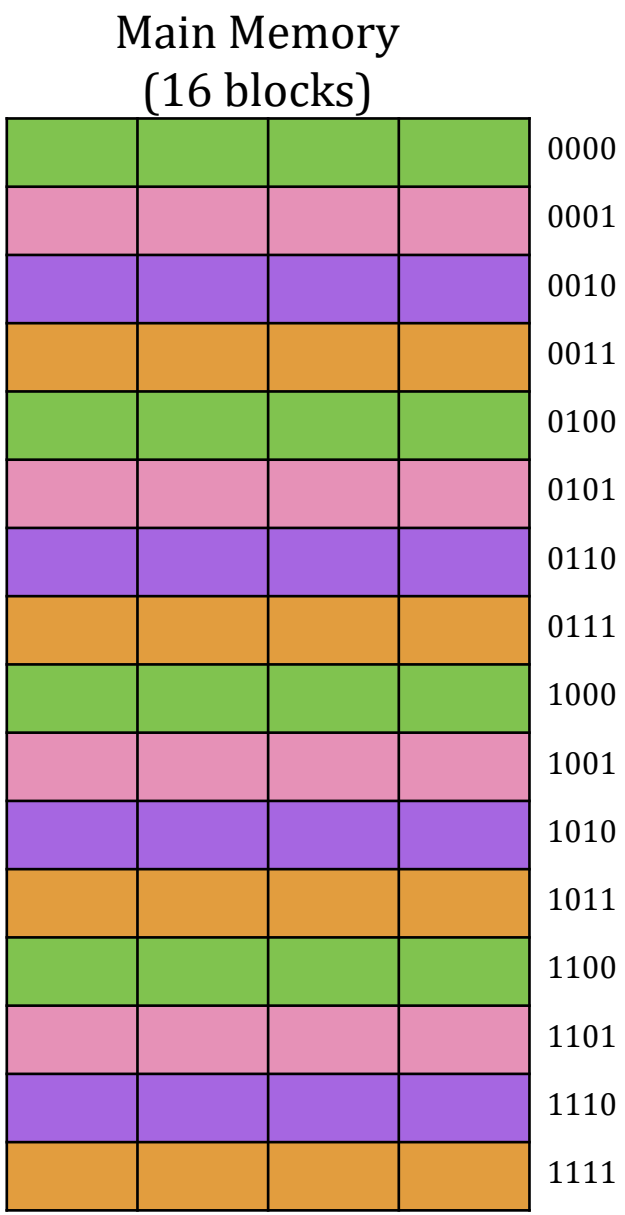


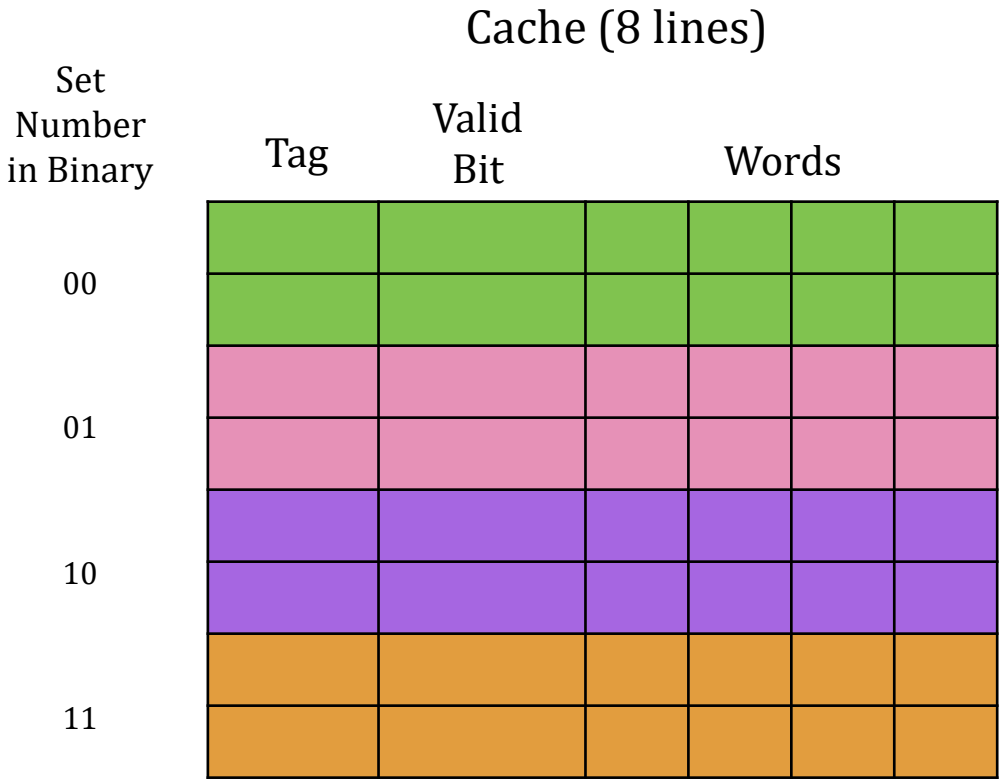


For k-way set associative, the CPU will find out the set number

Then the CPU will have to search k blocks in that set for finding out the right block by tag matching (and checking valid bit)

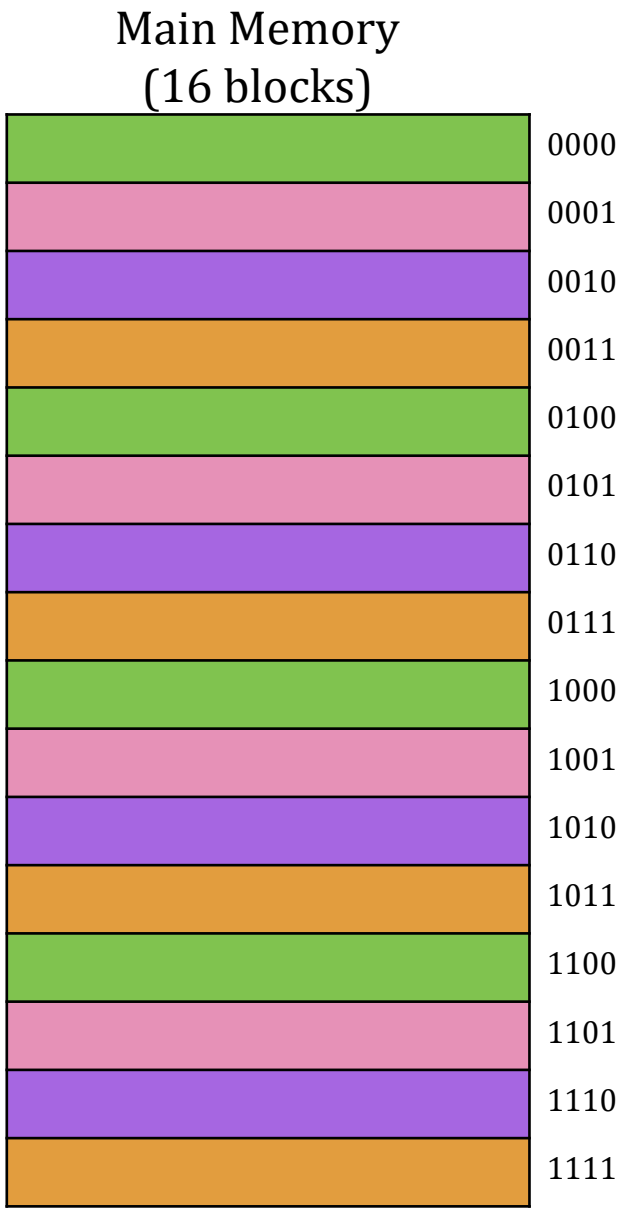
Accessing the Set Associative Cache

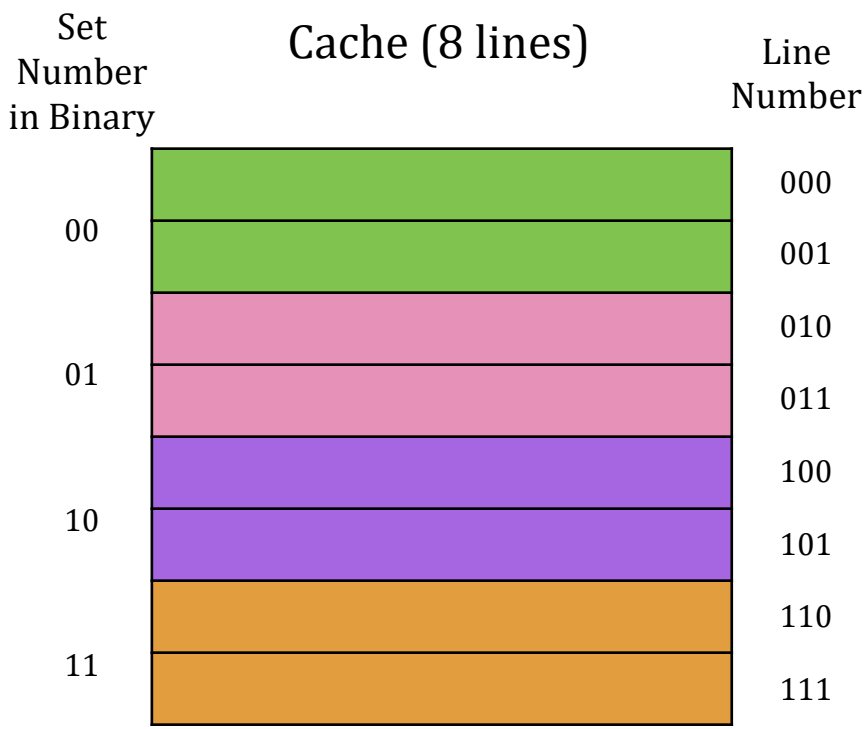




Then, the right word is found using the offset

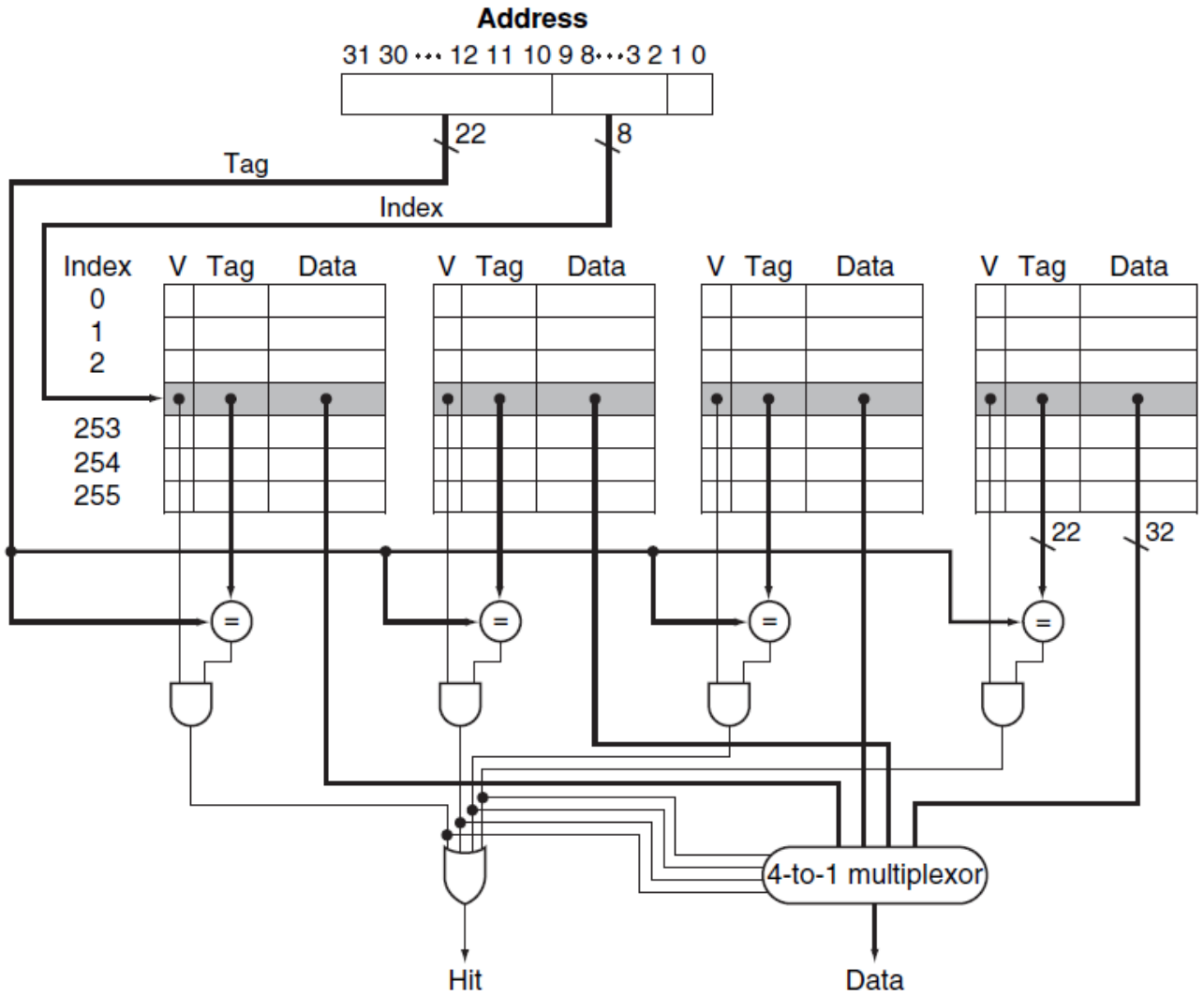
Accessing the Set Associative Cache



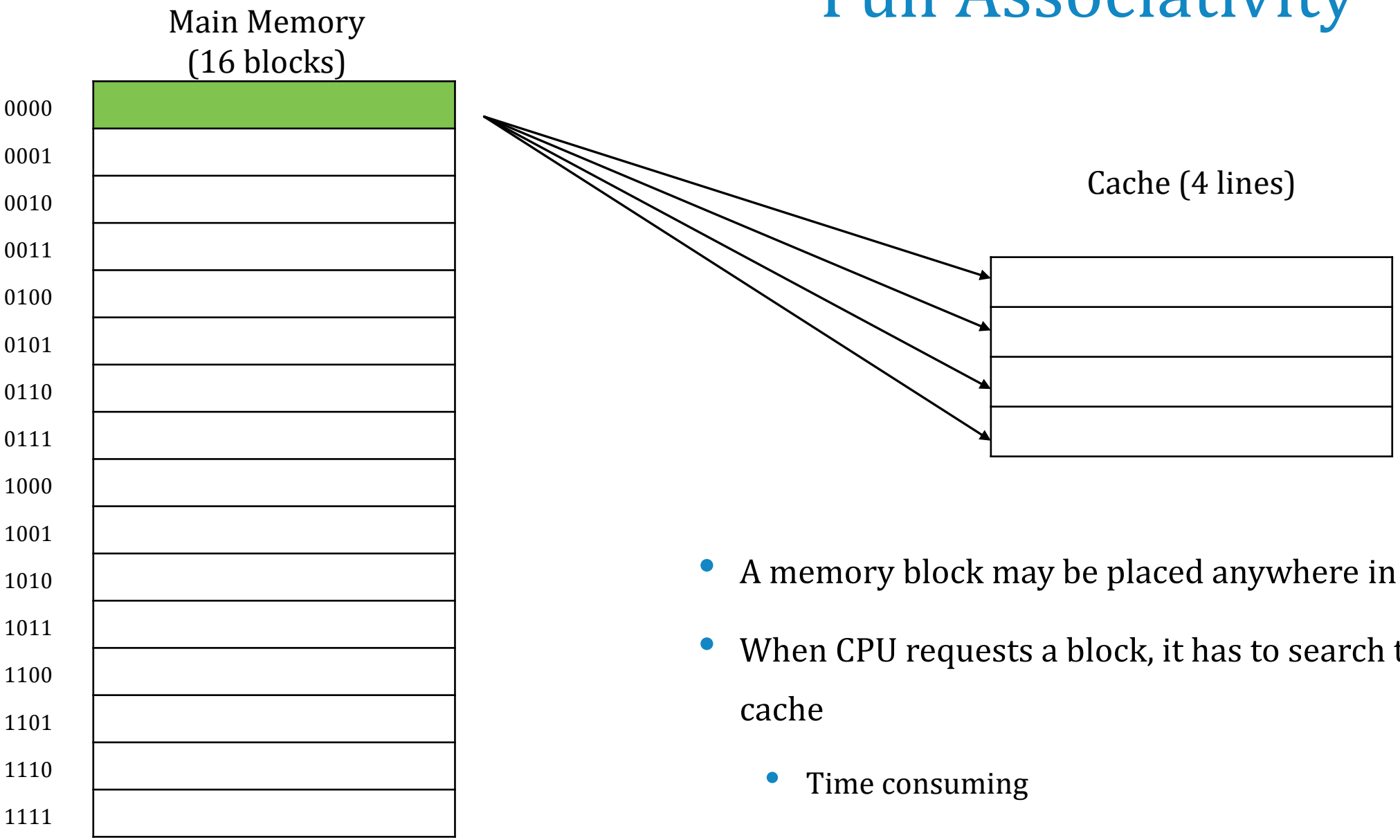


A one-way associative cache is called ____

Locating a Block in Set Associative Cache

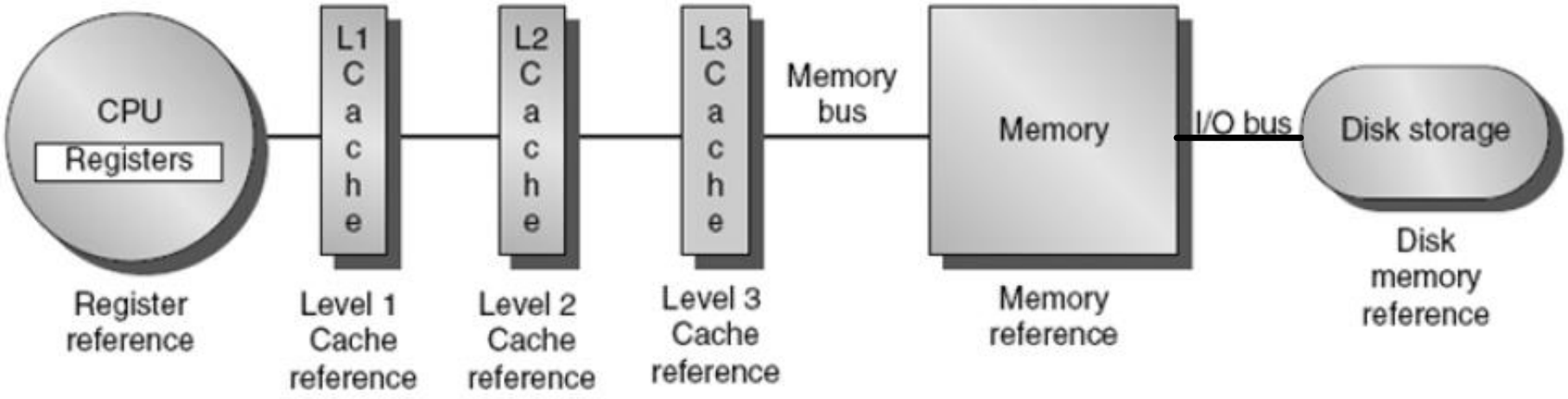


Full Associativity

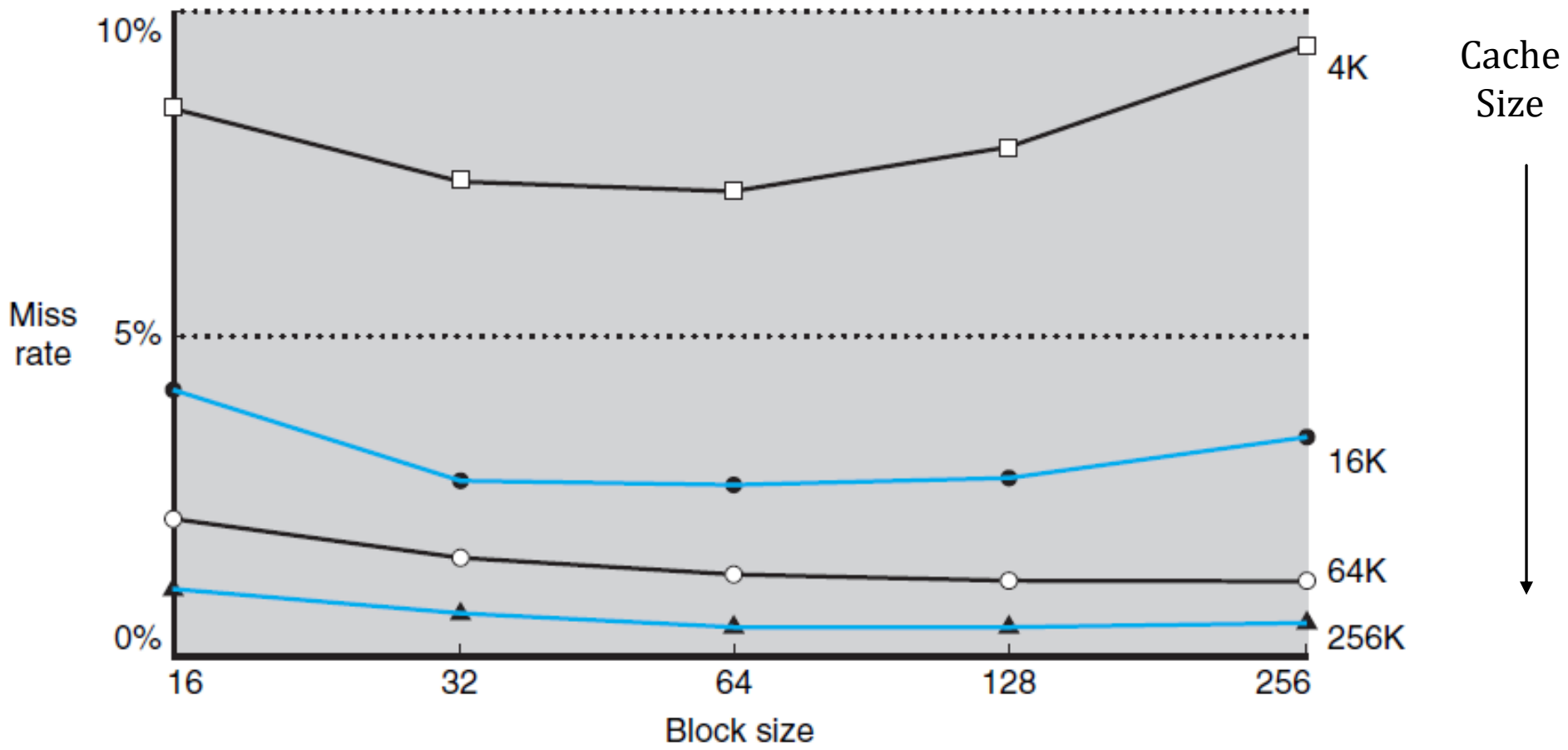


- A memory block may be placed anywhere in the cache
- When CPU requests a block, it has to search the entire cache
  - Time consuming

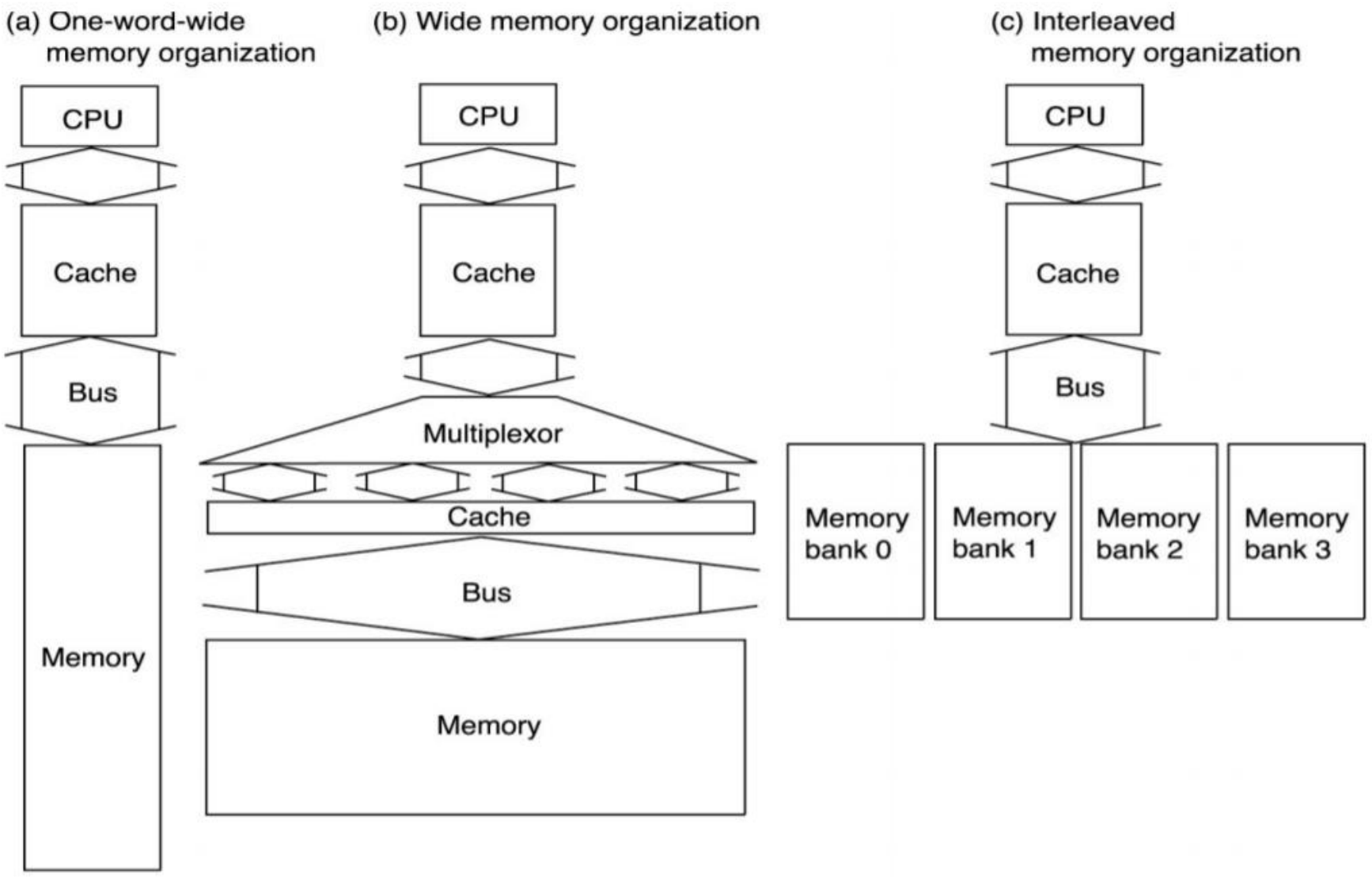
Memory Hierarchy



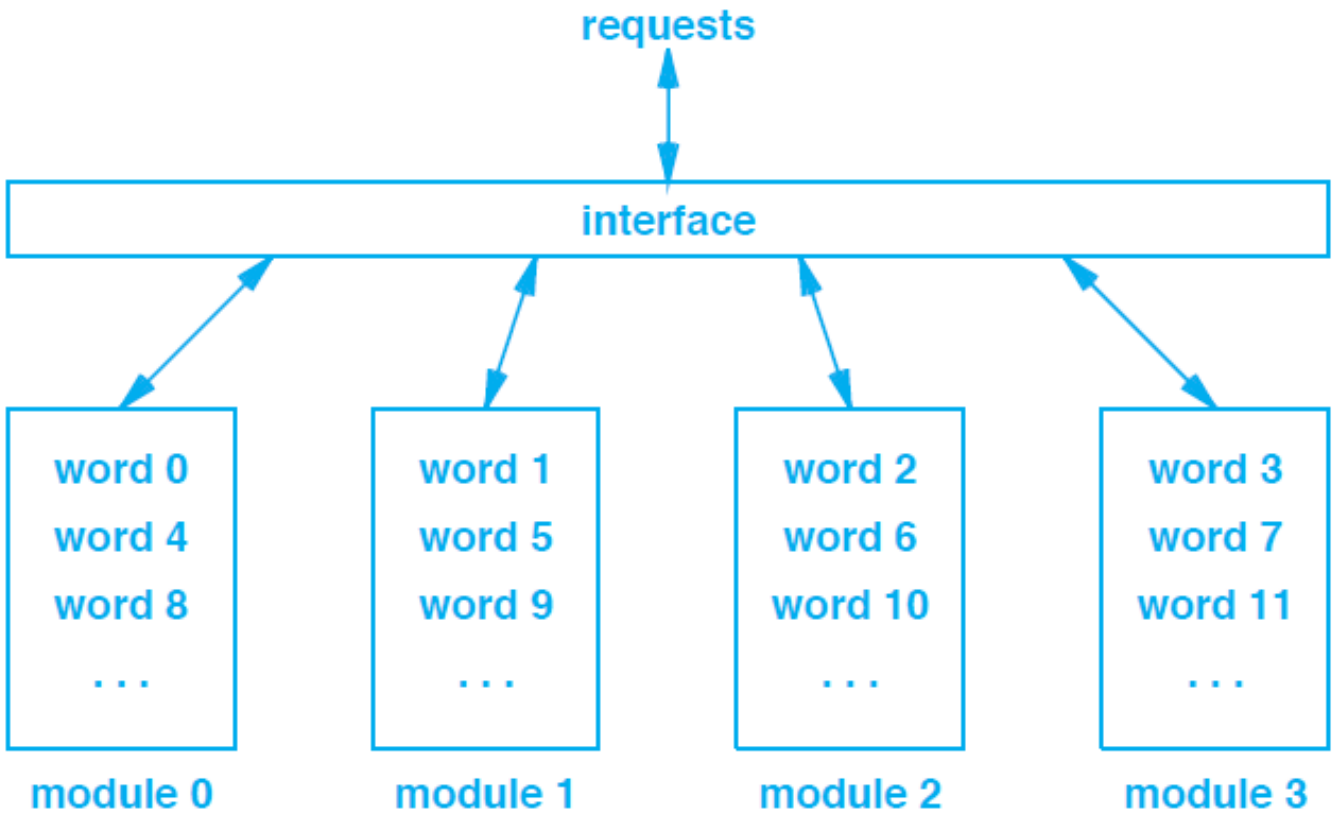
Miss Rate


10%
5%
0%
Miss rate
4K
16K
64K
256K
16
32
64
128
256
Block size
Cache Size

Achieving Higher Memory Bandwidth

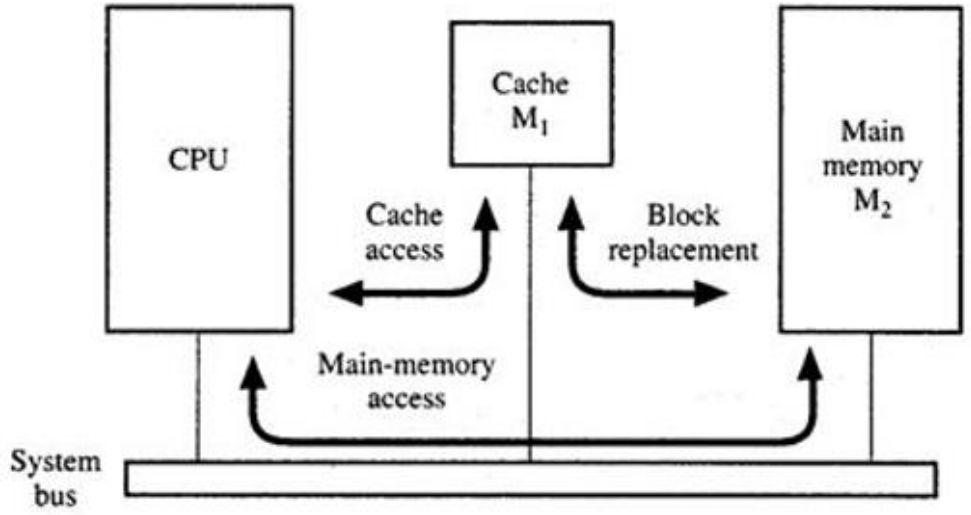


Memory Interleaving



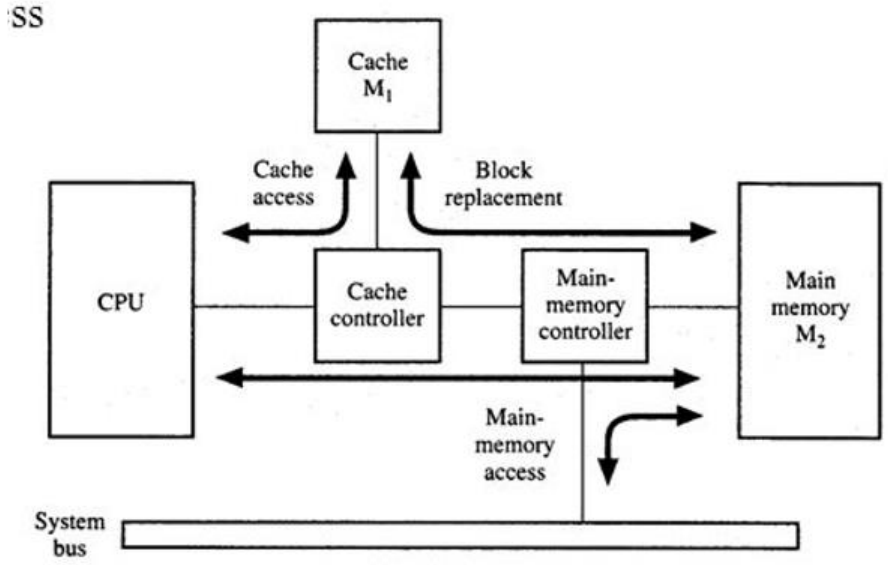
Cache Reading: Look Aside Cache

- Request from processor goes to cache and main memory in parallel
- On cache hit
  - Processor loaded from cache, bus cycle terminates
- On cache miss
  - Processor & cache loaded from memory in parallel
- Less expensive, better response to cache miss
- Processor cannot access cache while another controller is working with main memory



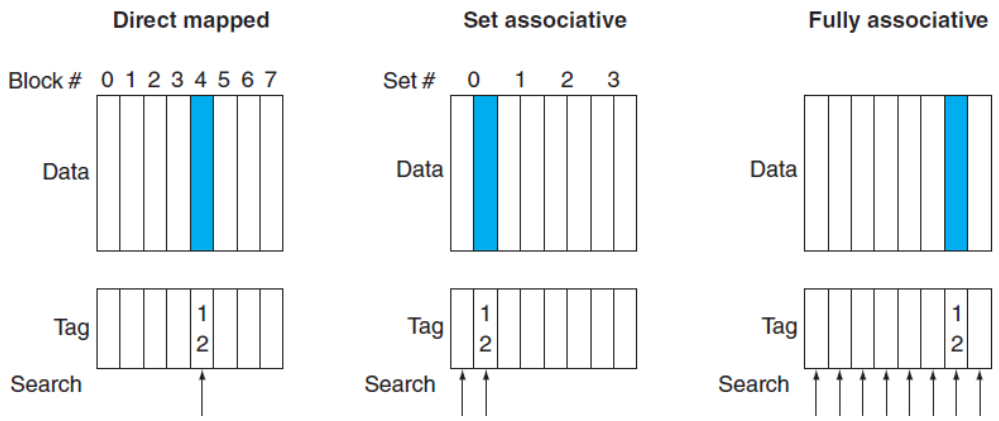
Cache Reading: Look Through Cache

- Cache checked first when processor requests data from memory
- On cache hit
  - Data loaded from cache
- On cache miss
  - Cache loaded from memory, then processor loaded from cache
- Processor can access cache while another controller is working with main memory
- An access to the main memory is slower because first of all data are searched in the cache and after that in main memory



Block Replacement

- What to do when cache is full and processor is requesting a new block?
  - Replace some cache block
- Which block is to be replaced?
- Useful for set associative and fully associative cache
- For set associative
  - Replace a block from the relevant set
- For fully associative
  - Replace a block from the entire cache



Block Replacement Algorithms

- Random (Pseudo-random)
- First In First Out (FIFO)
- Optimal
- Least Recently Used

Random (Pseudo-random) Replacement Algorithm

- Requires pseudo-random number generator
- Constant overhead
- Does not use any prior information
- Not efficient in general

FIFO Replacement Algorithm

- Replace the block that has been in the cache for the longest duration
- Requires a queue to store references
- Potential problems?

Optimal Replacement Algorithm

- Replace the block that will not be used for longest time in the future
- Requires knowledge of future
- Can not be implemented in real-world scenarios

Least Recently Used Replacement Algorithm

- Replace the block that has been used least recently (i.e. at the most distant past)
- For set associative cache
- Implementation using a queue
  - Whenever a block is referenced, it is brought to the front of the queue
  - Replace: block at the tail of the queue
  - New block: placed at the front of the queue

Writing in Cache

- Consider a store instruction such as sw
- Where should we write the data
  - In cache only?
  - In main memory only?
  - In both?
- In the first two cases, content of cache would be different from that of main memory (inconsistent)

Writing in Cache: Write Through

- The information is written to both the block in the cache and to the main memory
- In case of write miss
  - Data at the missed write location is not loaded to cache
  - Overwrite the data in main memory
  - No-write allocate or write around
- Simple implementation
- Not efficient: Every write causes data to be written in main memory

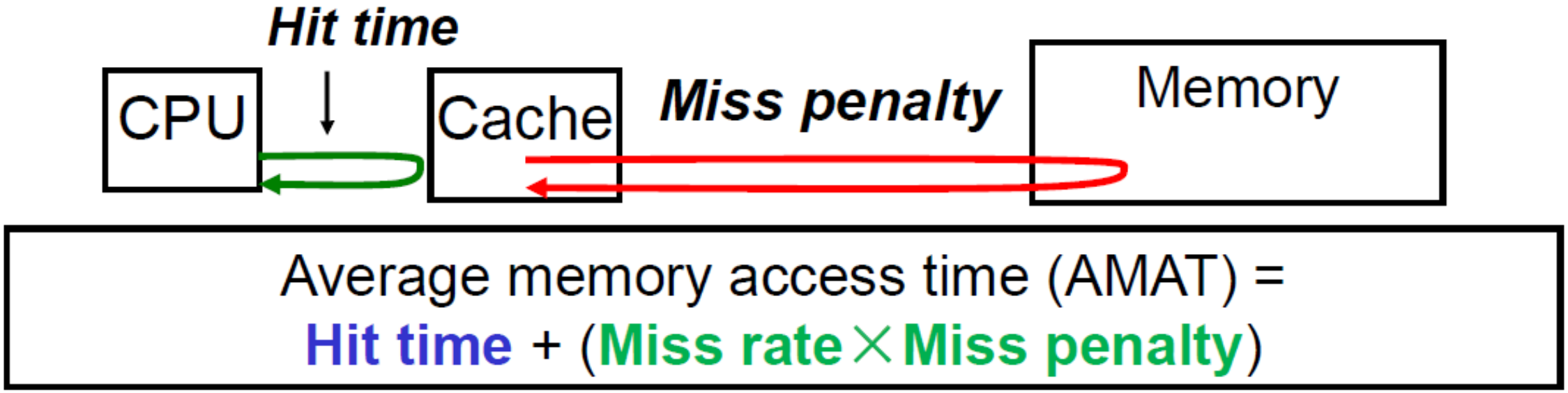
Writing in Cache: Write Back

- The information is written to the block in the cache
- The updated cache block is written to main memory only when it is replaced
- Dirty bit in cache block
  - Indicates whether the block was modified (dirty) or not modified (clean)
  - If clean, no action required at the time of replacement
  - If dirty, write the cache block to the main memory before replacement
- In case of write miss
  - Missed block fetched to cache from main memory (similar to read miss)
  - Data in cache is overwritten

Types of Cache Misses

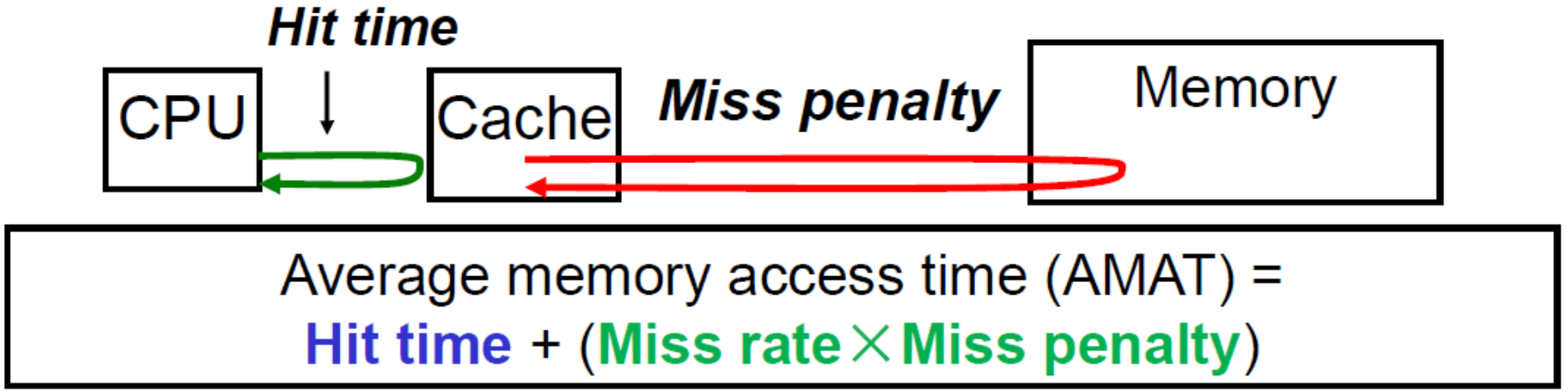
- Compulsory
  - Very first access to a block
  - Will occur even for an infinite cache
- Capacity
  - Occurs if cache cannot contain all the blocks needed
  - Misses in fully associative cache (due to the capacity)
- Conflict
  - Occurs if too many blocks map to the same set
  - Occurs in associative or direct mapped cache

Cache Performance



Miss Penalty: The extra time required to bring the data into cache (upper level of memory) from the Main memory (lower level of memory) whenever there is a miss in the cache

Cache Performance

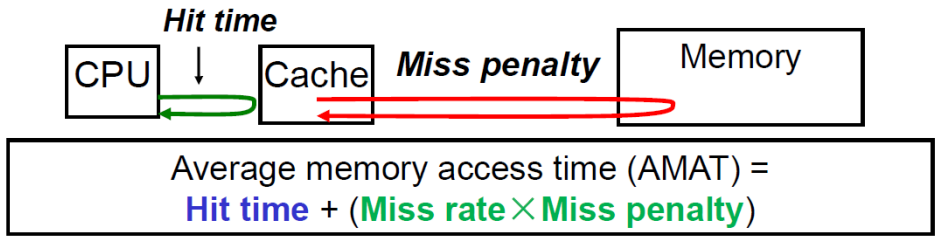


CPU time =(CPU execution clock cycles +Memory-stall clock cycles) × Clock cycle time

Memory-stall clock cycles = No. of Memory accesses × Miss rate × Cycles required for each miss penalty

(Assuming equal penalties for read miss and write miss)

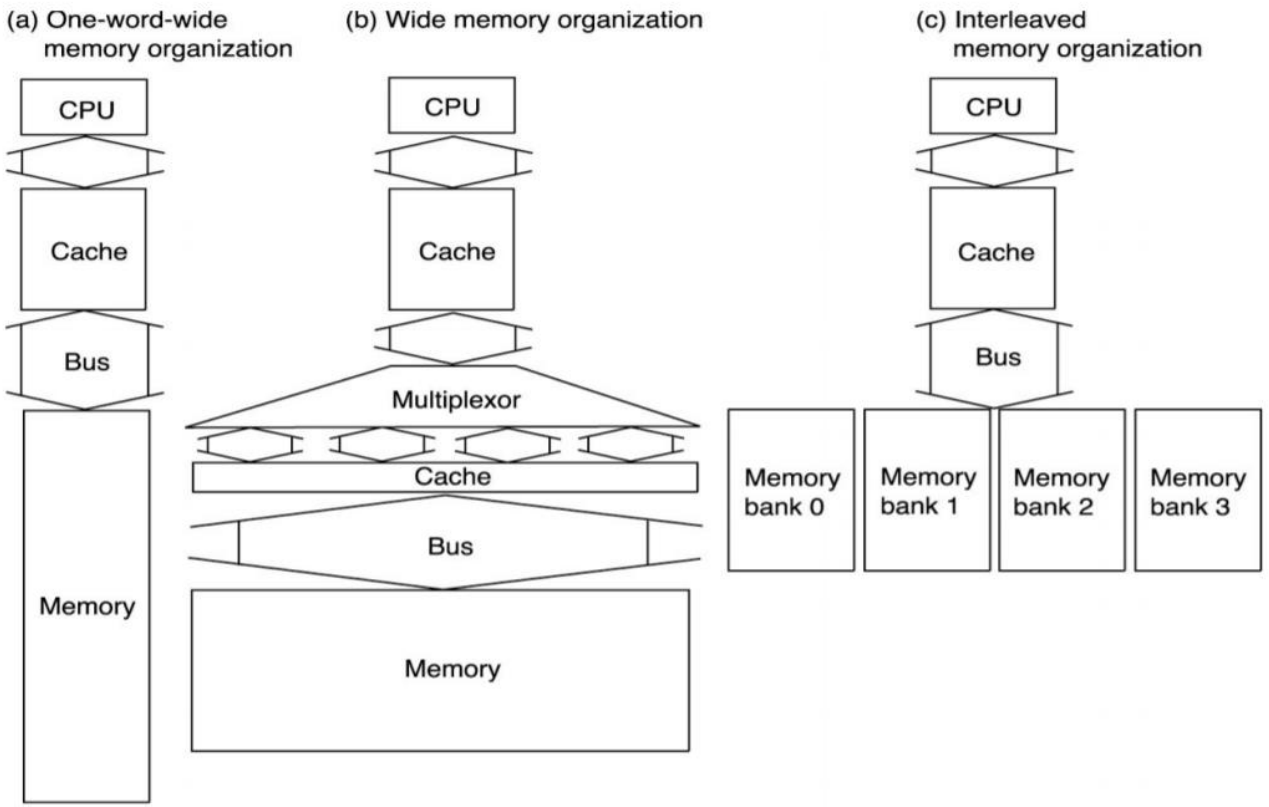
Improving Performance



- Reducing
  - Miss rate
    - Flexible block placement (set associative cache)
    - Best possible block replacement algorithm
  - Miss penalty
  - Hit time

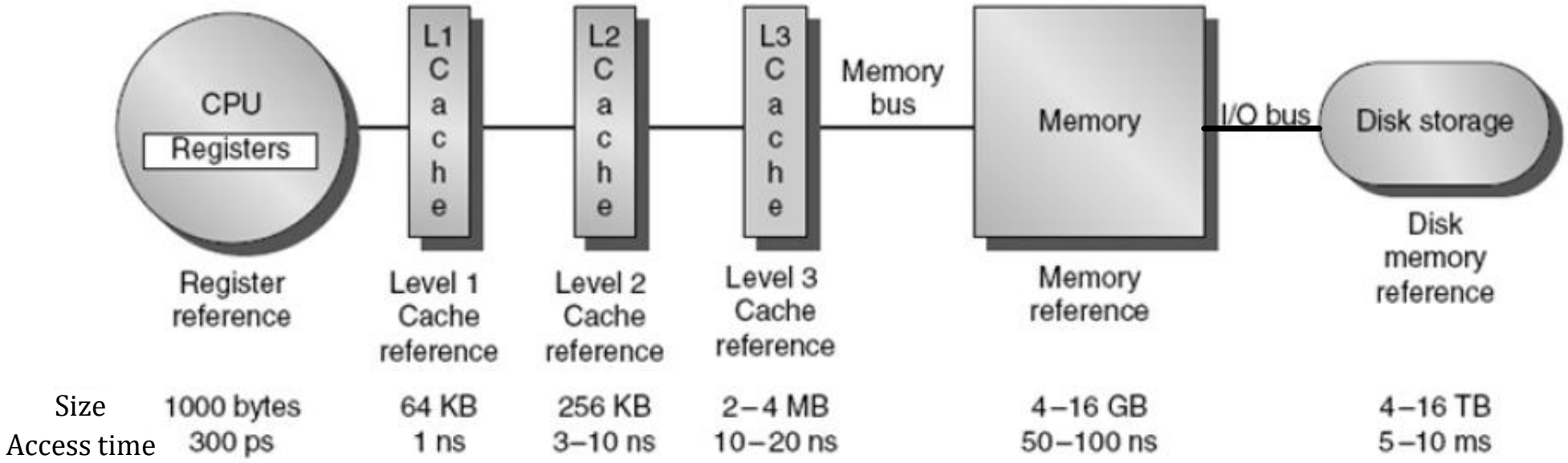
Reducing Miss Penalty

- Miss penalty depends on the speed of data transfer between main memory and cache



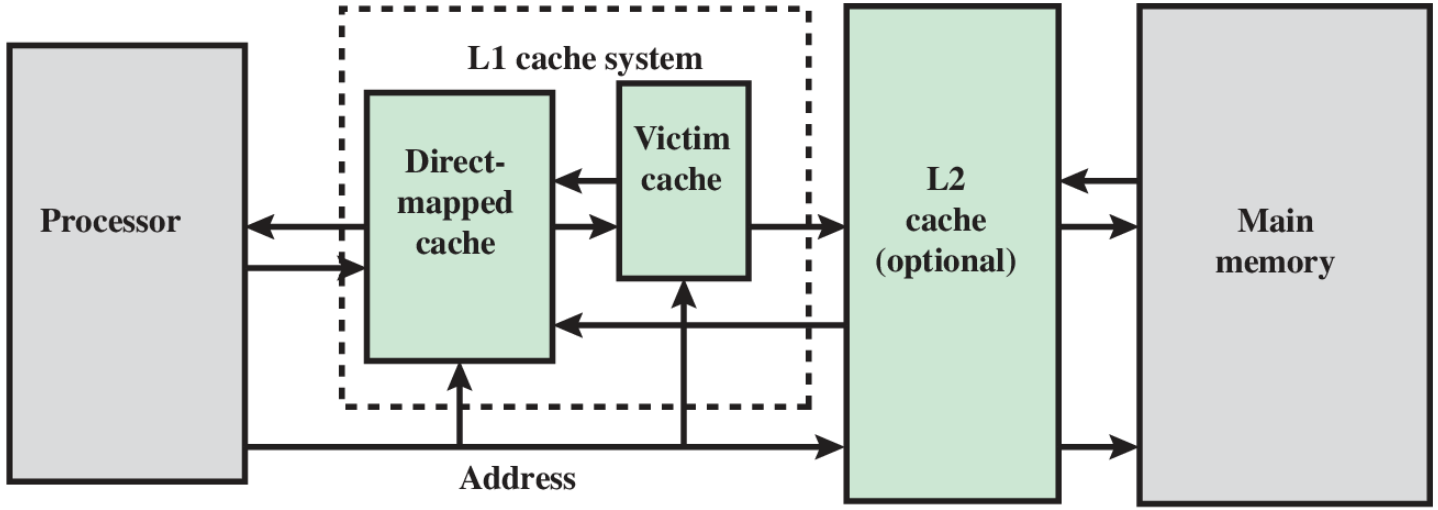
Reducing Miss Penalty: Multilevel Caches

- Accessing another level of cache: much faster than accessing the main memory
- On-chip L1 cache: reduction of processors external bus activity
- External L2 cache



Reducing Miss Penalty: Victim Caches

- Additional storage near L1 for MR evicted blocks
- Look up Victim Caches before L2



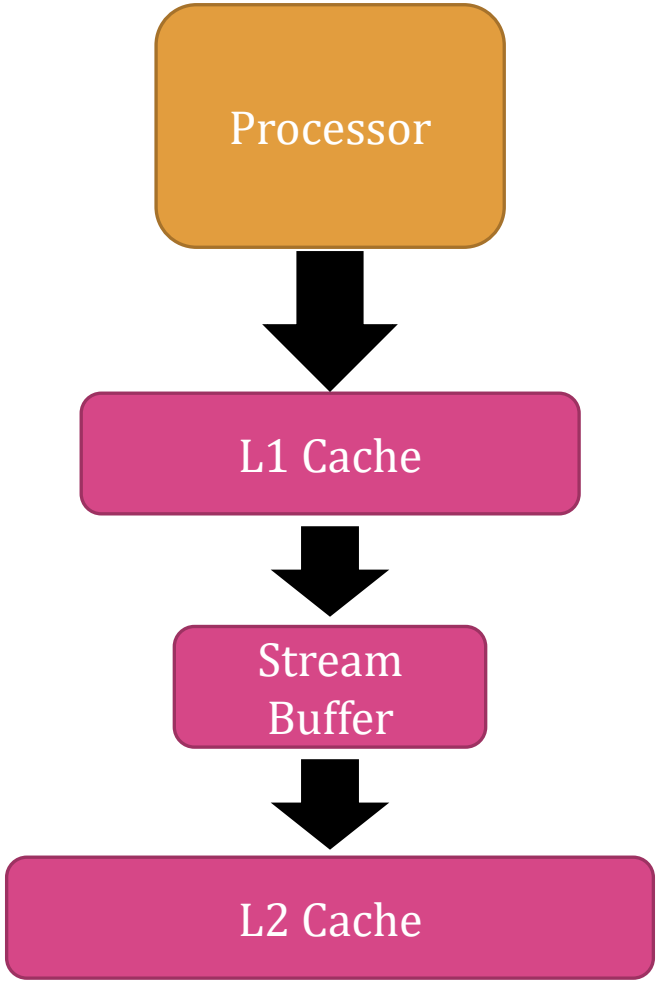
https://www.semanticscholar.org/paper/APPENDIX-D-VICTIM-CACHE-STRATEGIES-Stallings/fe9a9a554eced252eb1a1bb47a4128cb91c88623

Reducing Miss Penalty: Pre-fetching

- Pre-fetch blocks before processor request them
  - Data pre-fetching
  - Instruction pre-fetching
- Which blocks to pre-fetch?
  - Hardware pre-fetching
  - Software pre-fetching

Hardware Pre-fetching

- Stream buffers: A FIFO queue placed in between L1 and L2 cache
- When a miss occurs in L1, say at address A, the Stream Buffer immediately starts to prefetch elements from A+1
- Subsequent accesses: check the head of the Stream Buffer before going to L2
  - If block found in buffer: no cache miss



Software Pre-fetching

- Usually compiler-directed
- Compiler predicts future cache misses
- Inserts a prefetch instruction based on the miss penalty and execution time of the instructions
- Prefetches are non-blocking memory operations
  - These memory accesses do not interfere with actual memory accesses

Software Pre-fetching

No Pre-fetching

```
for (int i=0; i<1000; i++) {
   var[i] = 10+var[i];
}
```

Pre-fetching

```
for (int i=0; i<1000; i++) {
   prefetch (var [i + n]);
   var[i] = 10+var[i];
}
```

Data and Instruction Cache

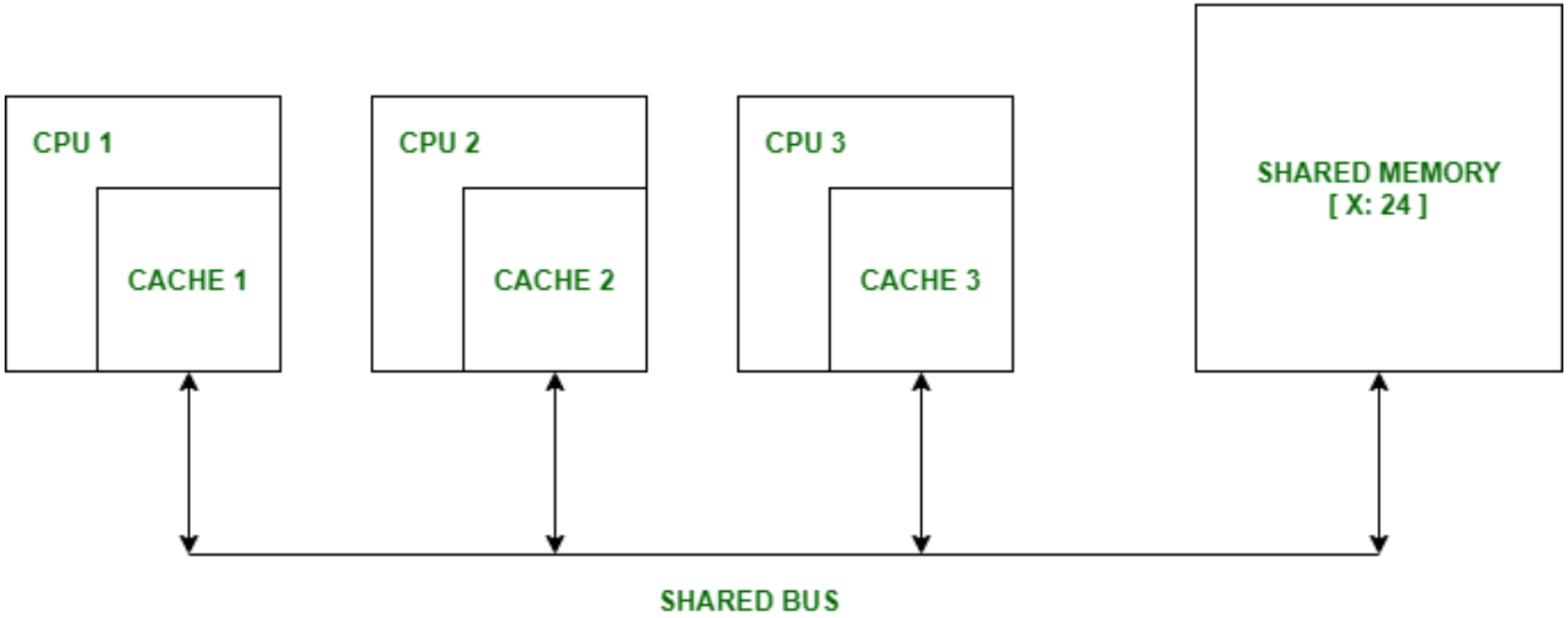
- Why have different I and D caches?
  - Different areas of memory
  - Different access patterns
- I-cache accesses
  - Lots of spatial locality
  - Mostly sequential accesses
  - Predictable to the extent that branches are predictable
- D-cache accesses are typically less predictable

Data and Instruction Cache

- Split cache: separate cache for data and instructions
  - Simultaneous access of data and instruction
- Unified cache: same cache for data and instructions
  - Best utilization of cache capacity
- Usually L1 is split cache

Cache Coherence

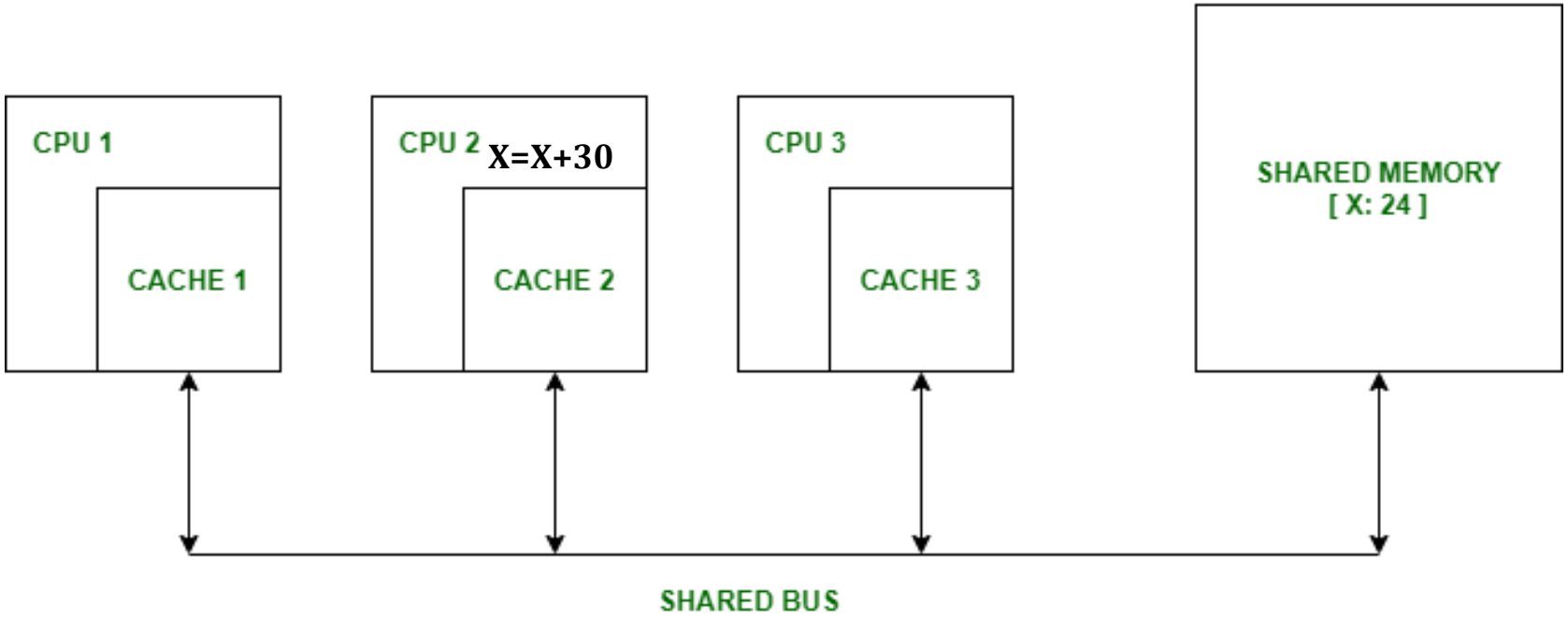
Assume write back cache in a multiprocessor system



https://www.geeksforgeeks.org/cache-coherence/

Cache Coherence

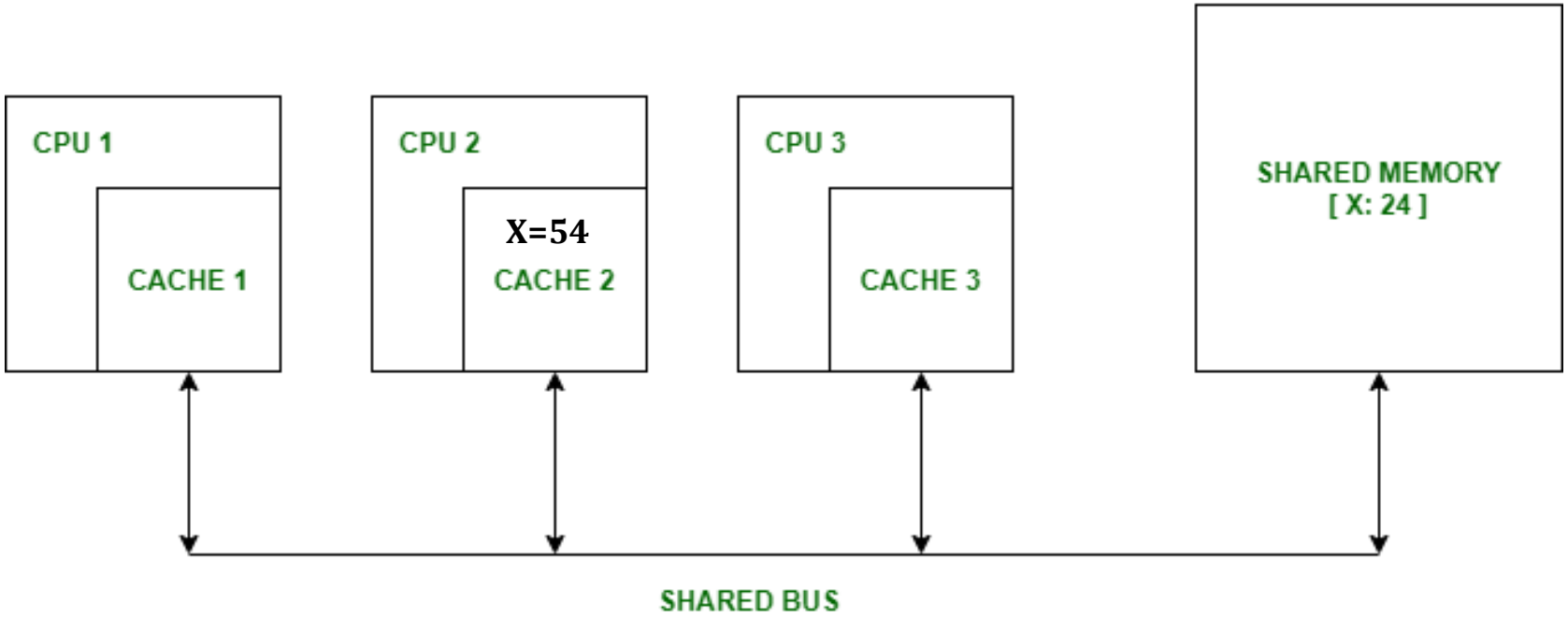
Assume write back cache in a multiprocessor system



https://www.geeksforgeeks.org/cache-coherence/

Cache Coherence

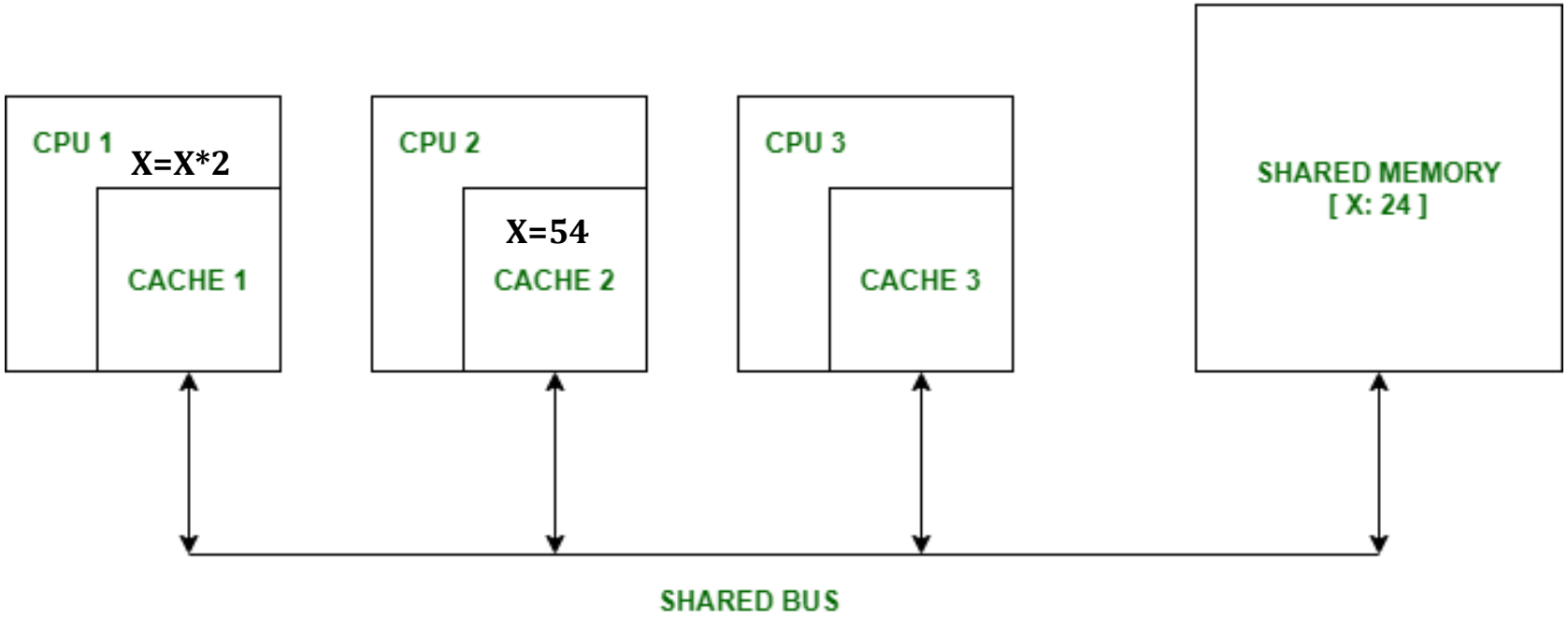
Assume write back cache in a multiprocessor system



https://www.geeksforgeeks.org/cache-coherence/

Cache Coherence

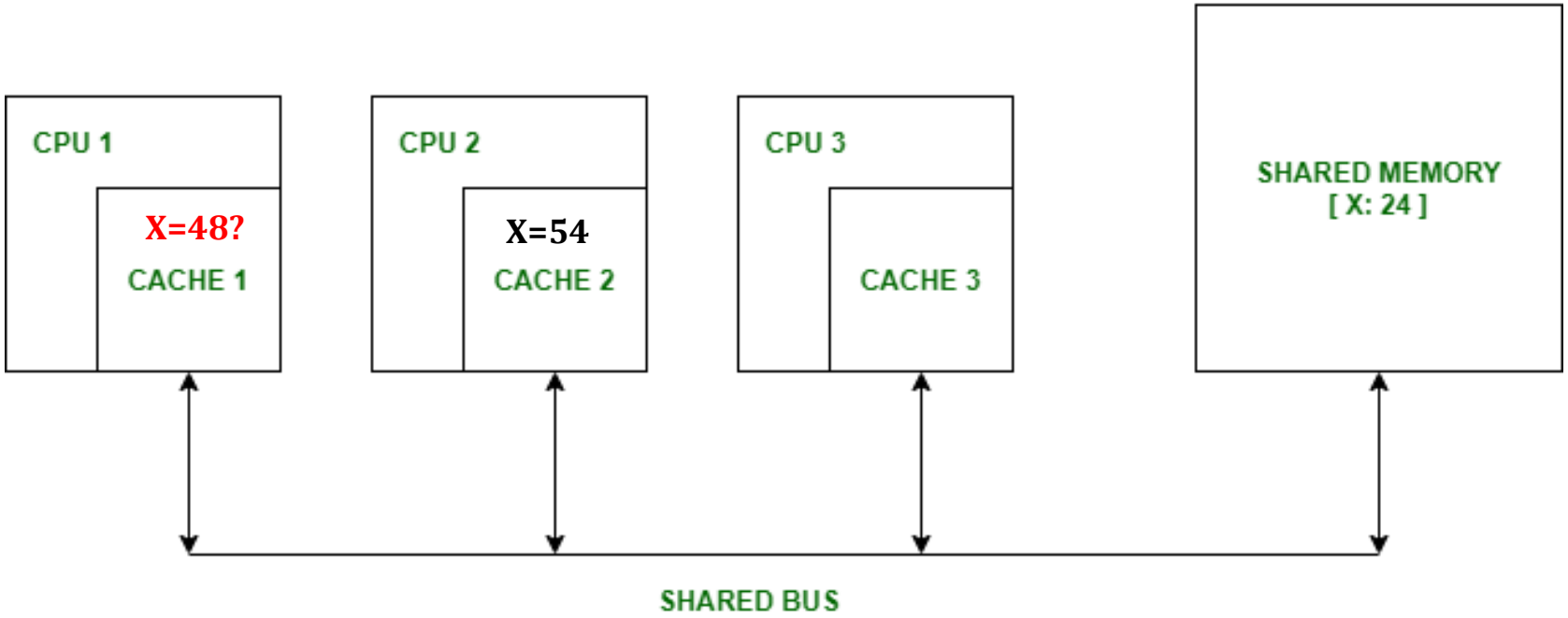
Assume write back cache in a multiprocessor system



https://www.geeksforgeeks.org/cache-coherence/

Cache Coherence

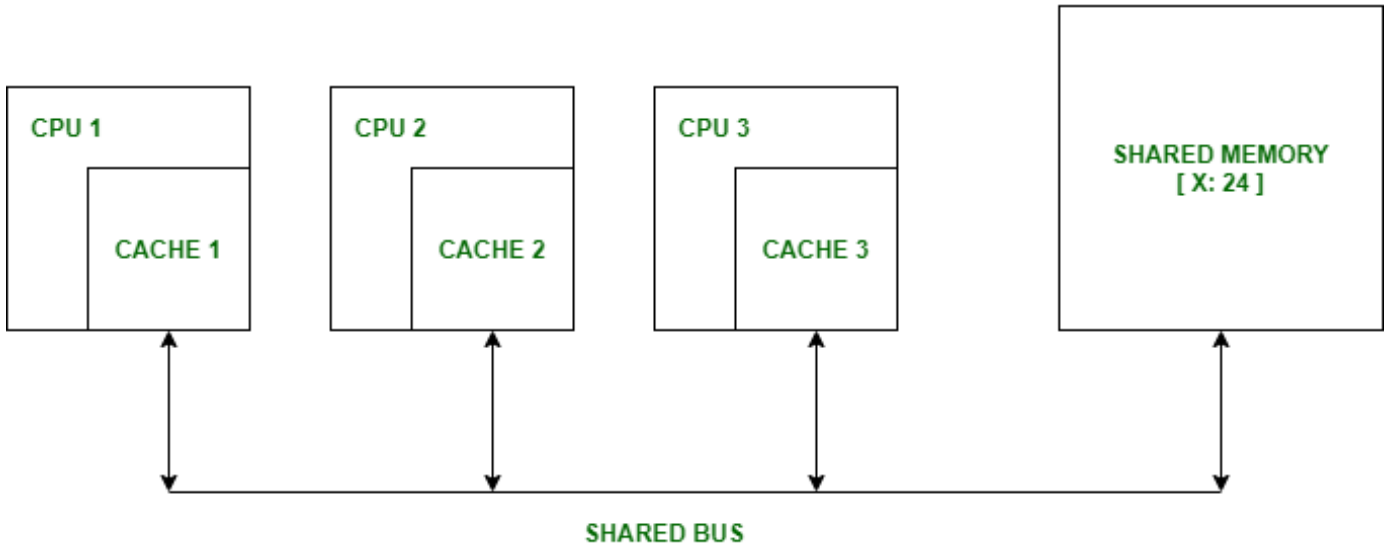
Assume write back cache in a multiprocessor system



https://www.geeksforgeeks.org/cache-coherence/

Cache Coherence

- Uniformity of shared data that gets stored in multiple local caches



https://www.geeksforgeeks.org/cache-coherence/

States of Cache Block

- A cache block may have the following states
- **Modified:** The value in the cache is dirty (the value in current cache is different from that of the main memory)
- **Shared:** The cache value holds the most recent data copy and that is what shared among all the cache and main memory as well
- **Invalid:** This states that the current cache block itself is invalid and is required to be fetched from other cache or main memory
- **Owned:** It means that the current cache holds the block and is now the owner of that block, that is having all rights on that particular blocks. It is responsible for satisfying the read requests from other processors
- **Exclusive:** The cache block is present only in the current cache and its content is same as that present in the main memory (the value is clean)

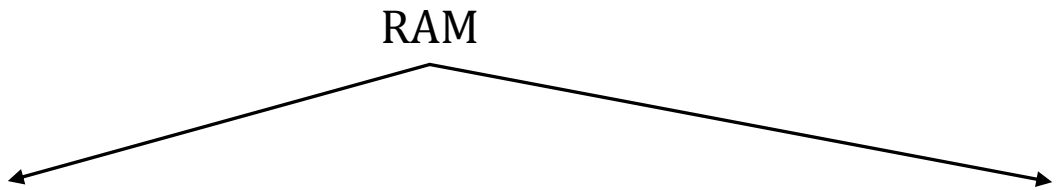
Cache Coherence Protocols

- **Snooping**
  - Works with bus-based systems
  - Snooping (monitoring) on the bus at all times by individual cache controllers
  - Processor generates broadcast requests
  - Two protocols are possible
    - Write update protocol: When a local cache block is updated, the new data block is broadcast to all caches containing a copy of the block for updating them
    - Write invalidate protocol: Invalidate all remote copies of cache and shared memory when a local cache block is updated

Cache Coherence Protocols

- **Directory-based**
  - The data being shared is placed in a common directory that maintains the coherence between caches
  - The directory acts as a filter through which the processor must ask permission to load an entry from the primary memory to its cache
  - When an entry is changed, the directory either updates or invalidates the other caches with that entry.

Random Access Memory (RAM)



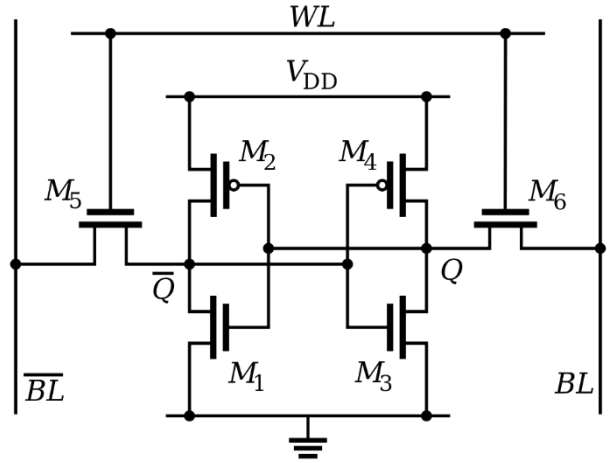
- **Used for implementing cache**
- **Designed using flip-flops**
- **Once the information is stored it stays there until power is on**

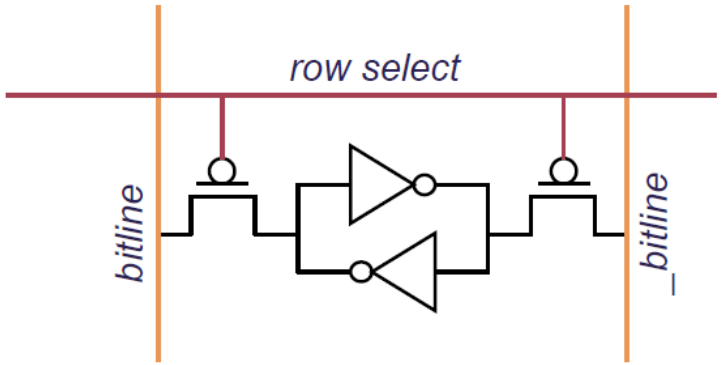
- **Used for implementing main memory**
- **Designed using capacitors**
- **Information need to be refreshed periodically**

RAM is volatile

Memory Technology: SRAM

- Static random access memory
- Two cross coupled inverters store a single bit
- Typically 6 transistors required for storing one bit
- Both the target bit and its inverse are made available

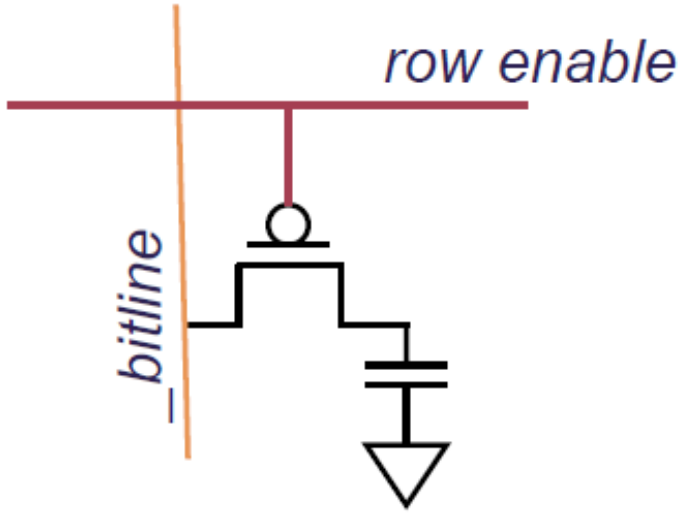




https://en.wikipedia.org/wiki/Static_random-access_memory#/media/File:SRAM_Cell_(6_Transistors).svg

Memory Technology: DRAM

- Dynamic random access memory
- Capacitor charge state indicates stored value
- Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor
  - 1 access transistor
- Capacitor leaks through the RC path
  - DRAM cell loses charge over time
  - DRAM cell needs to be refreshed
- Refresh: DRAM controller must periodically read all rows within refresh time such that charge is restored in cells



Memory Hierarchy

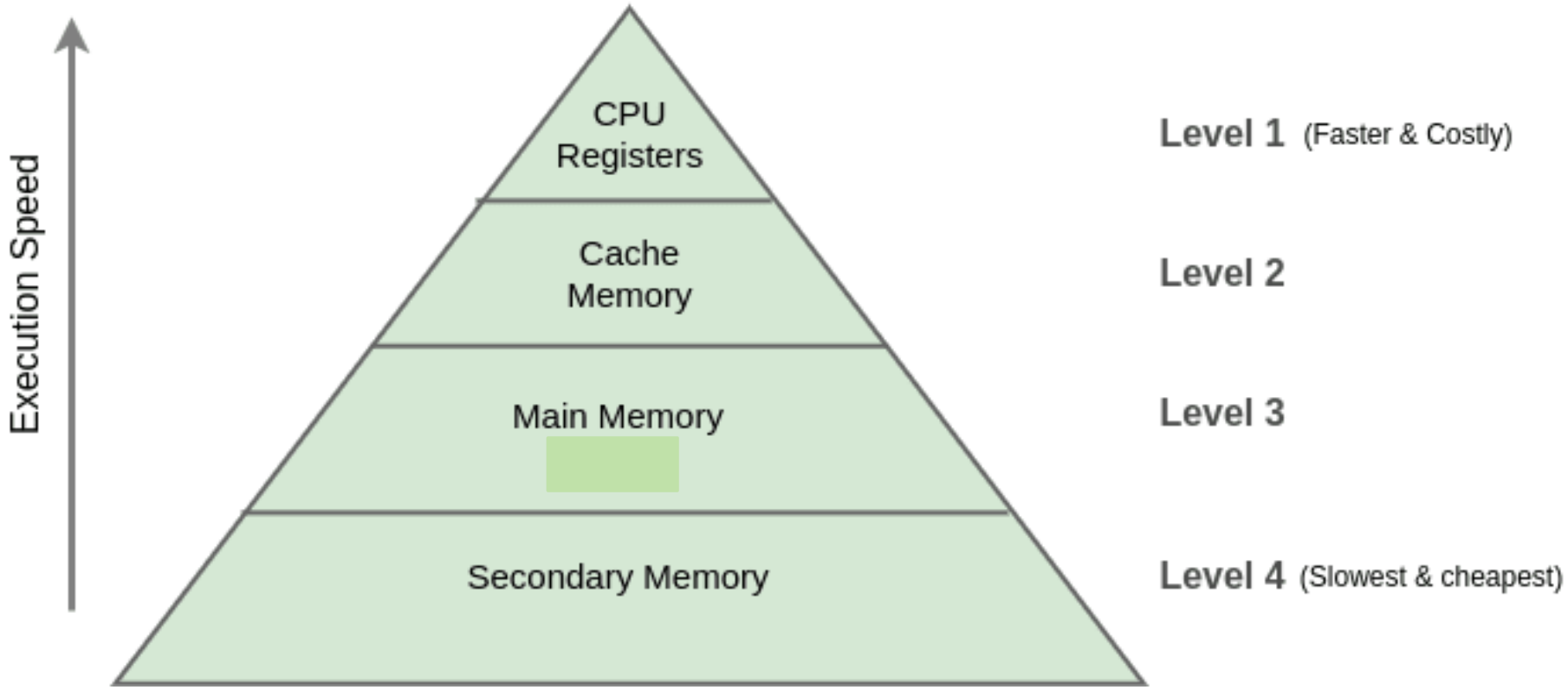
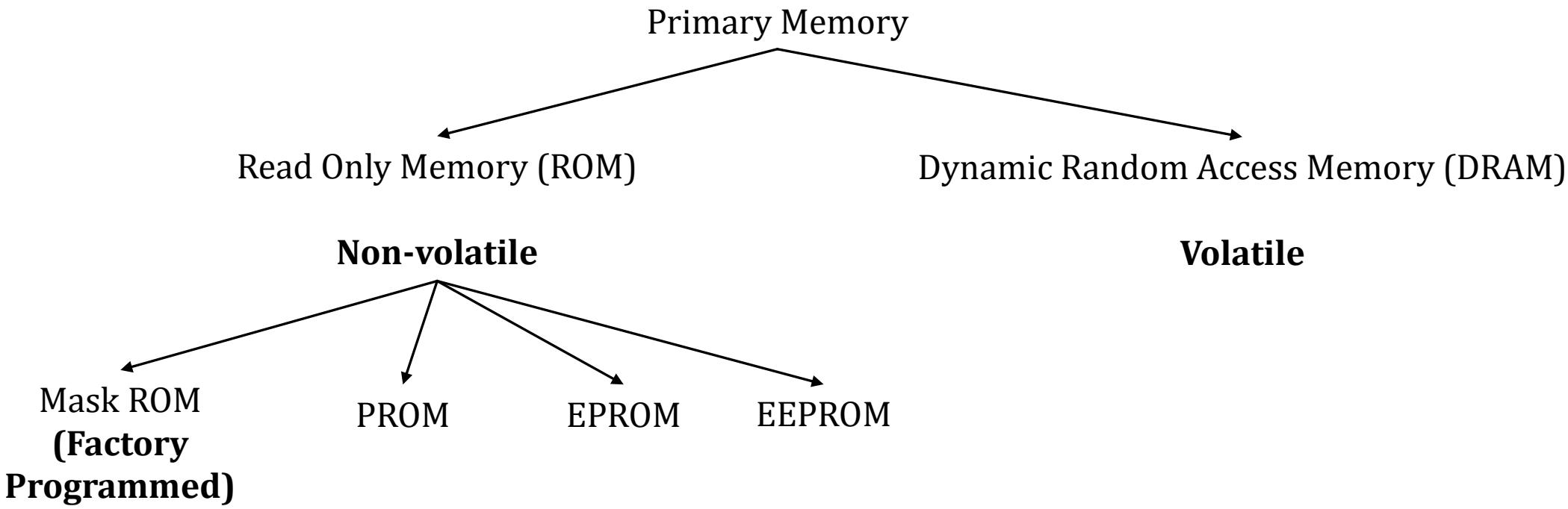


Image Source: https://nlogn.in/what-is-cpu-cache-memory-in-computer-architecture/

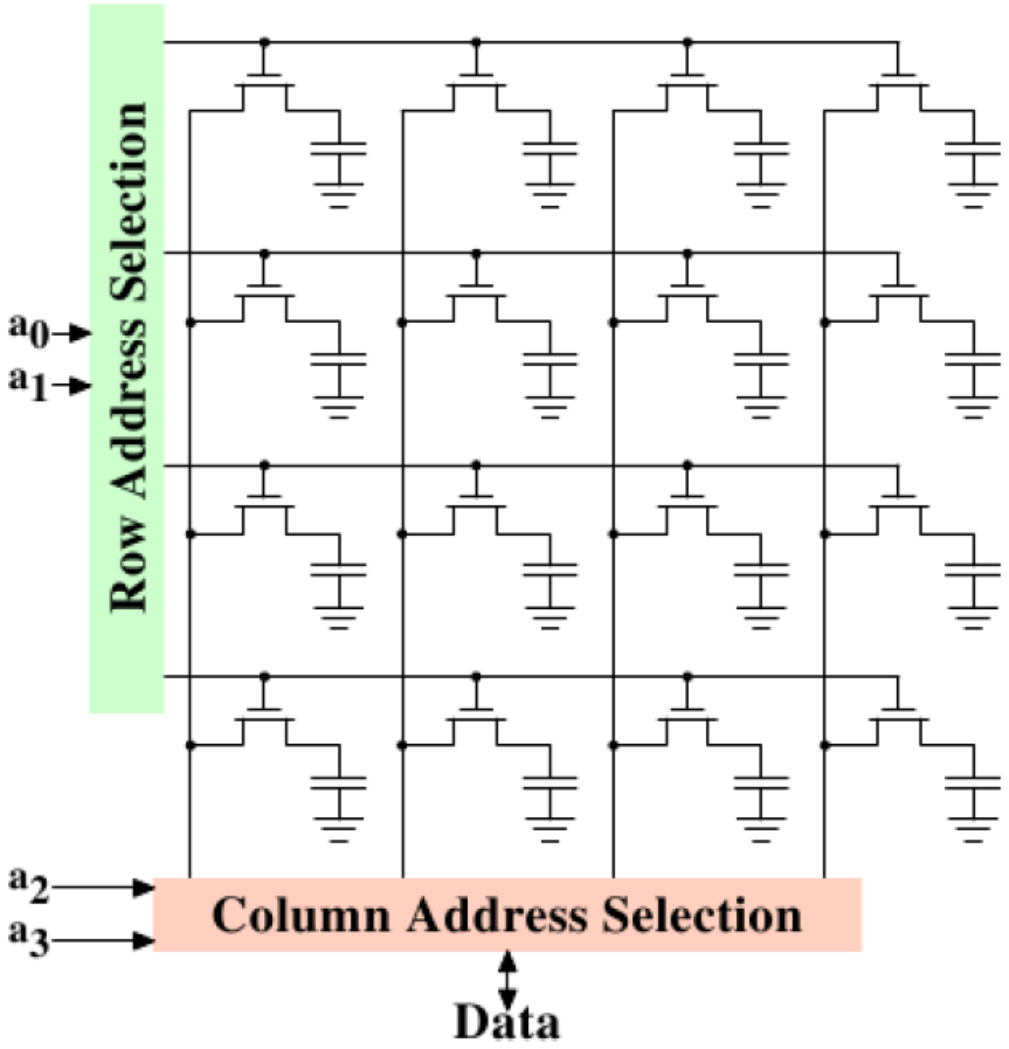
Primary/ Main Memory



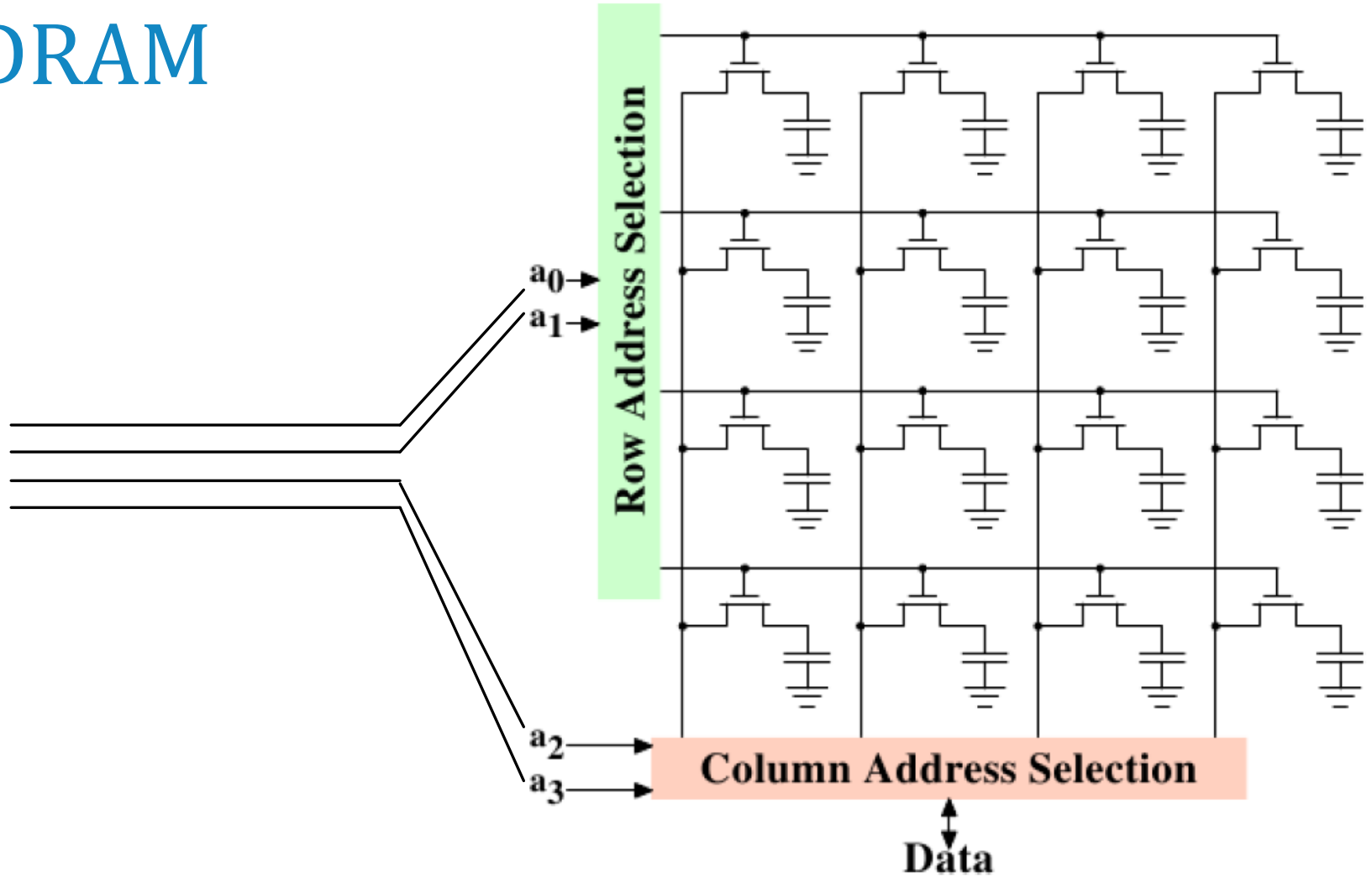
Main Memory

- Ideal Main memory
  - Zero access time (latency)
  - Infinite capacity, zero cost
  - Infinite bandwidth (to support multiple accesses in parallel)
- Memory Access time: The time it takes between a memory access request is issued to main memory and the time the requested information is available to cache/CPU.
- Memory Cycle time: The minimum time between requests to memory (greater than access time in DRAM to allow address lines to be stable)
- Peak Memory bandwidth: The maximum sustained data transfer rate between main memory and cache/CPU.

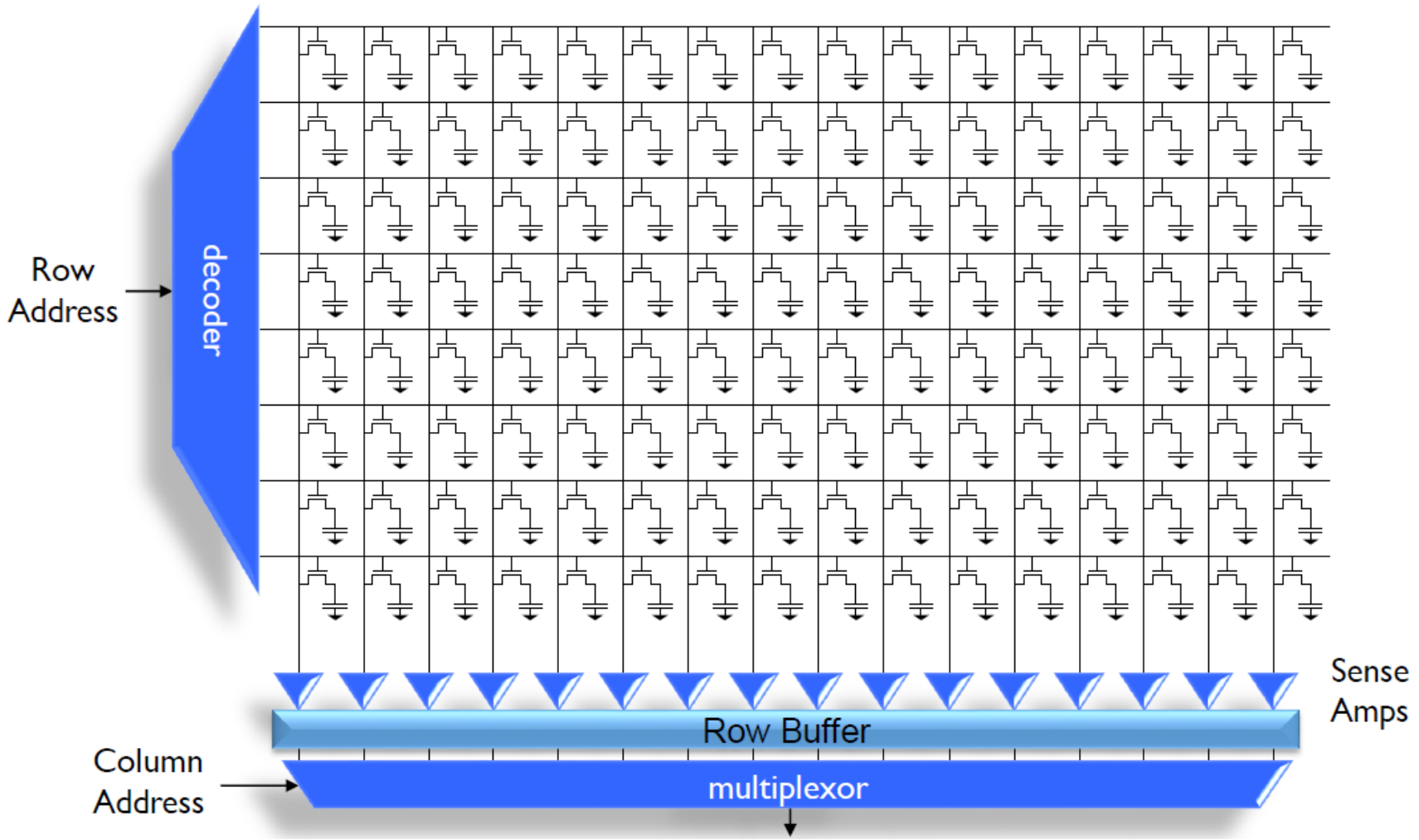
DRAM


Row Address Selection
a0
a1
a2
a3
Column Address Selection
Data

DRAM


Row Address Selection
a0
a1
a2
a3
Column Address Selection
Data

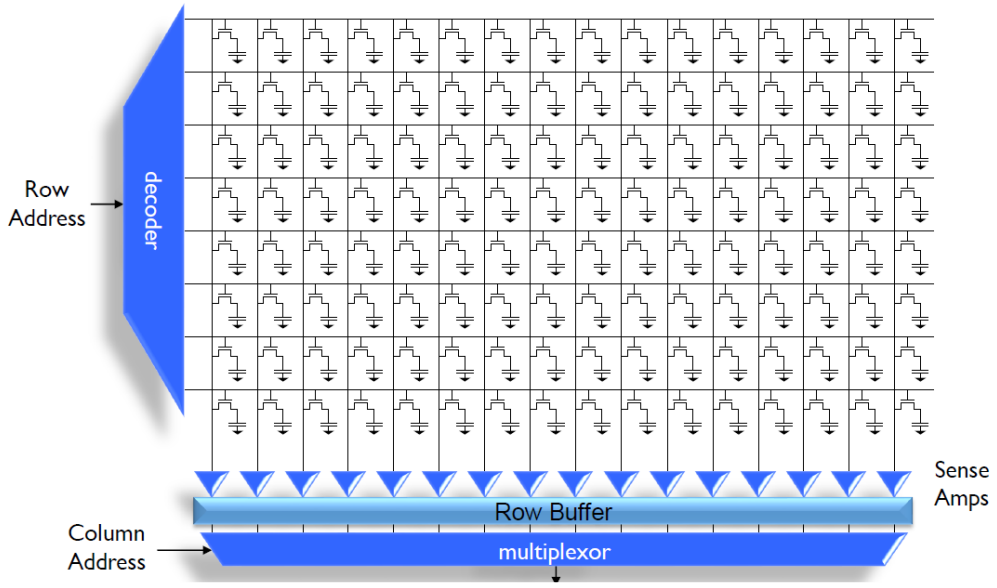
DRAM



https://compas.cs.stonybrook.edu/~nhonarmand/courses/sp15/cse502/slides/06-main_mem.pdf

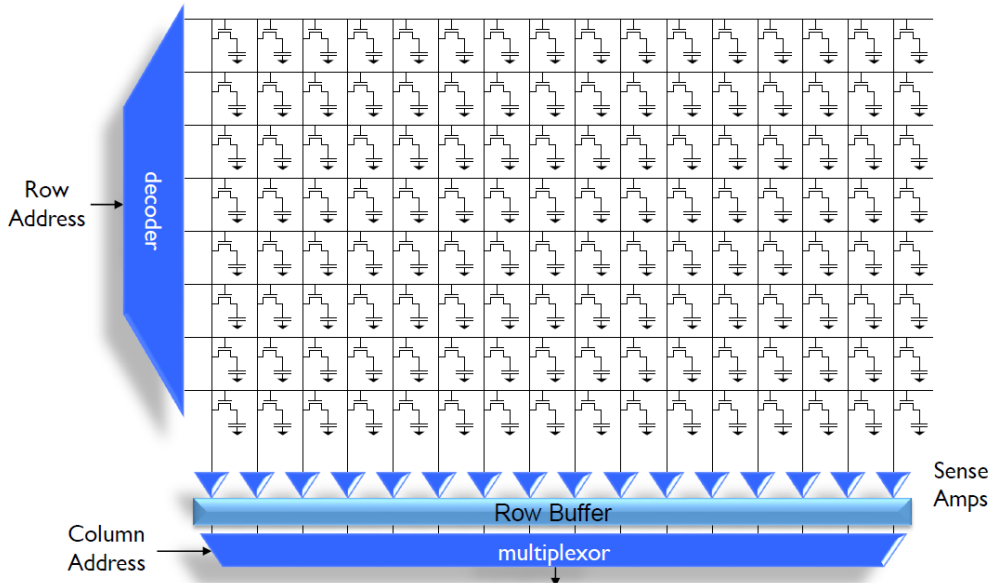
Fast Page Mode

- Sometimes a row is called a page (it is ***not the page that we use for virtual memory***)
- Accessing a row every time is time consuming
- Once a row is read, latch (row buffer may retain the entire row)
- If the next reference is from the same page (row)
  - Don't access the row
  - Just read the appropriate column from the latch (row buffer)



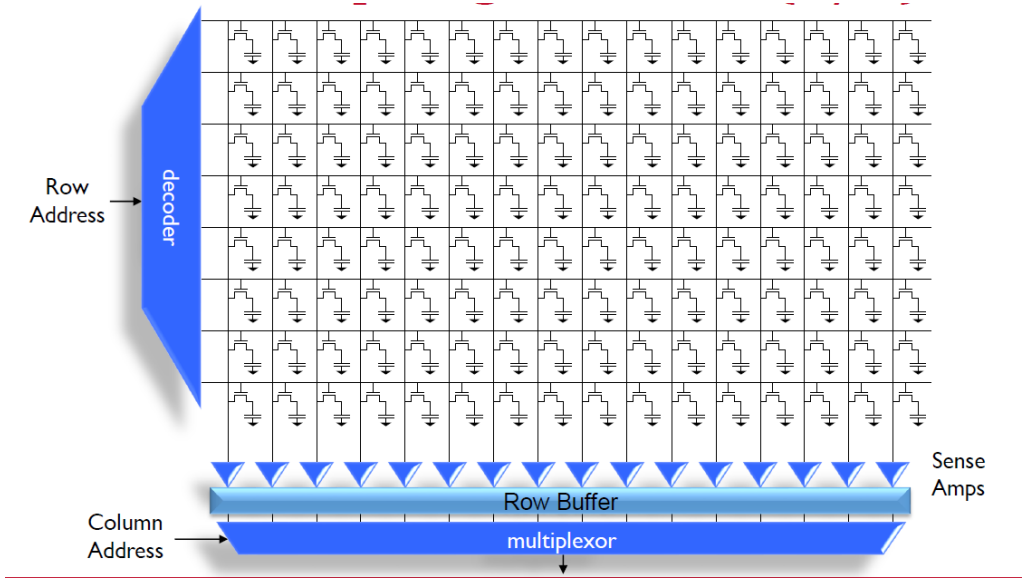
Fast Page Mode

- Opening a page
  - Provide row address
  - Select row
  - Sense amplification
  - Latch into row buffer
- Do read / write
- Close the page
  - Write data from row buffer to memory row

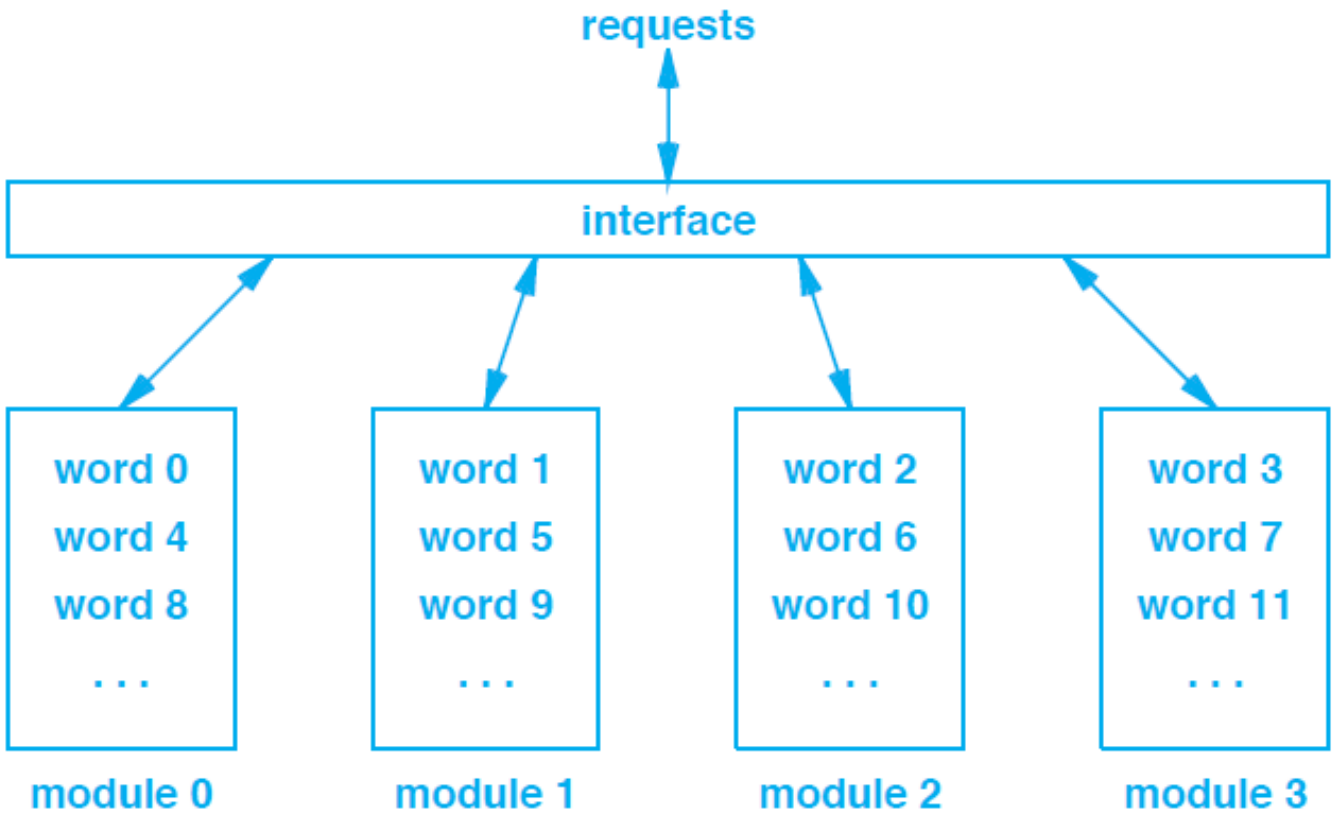


Interleaving (Banking)

- Problem: a single monolithic memory array takes long to access and does not enable multiple accesses in parallel
- Goal: Reduce the latency of memory array access and enable multiple accesses in parallel
- Idea: Divide the array into multiple banks that can be accessed independently (in the same cycle or in consecutive cycles)
- Each bank is smaller than the entire memory storage
- Accesses to different banks can be overlapped



Memory Interleaving

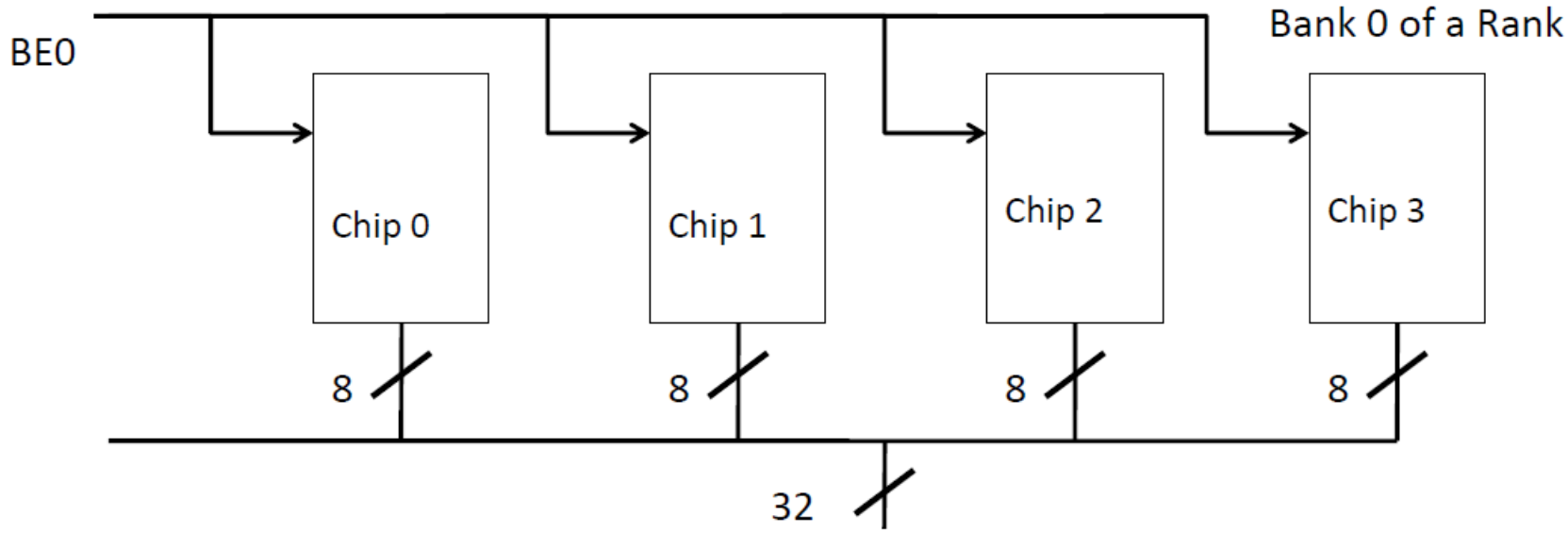


DRAM Rank

- Rank: Multiple chips operate together to form a wide interface

- All chips comprising a rank are controlled at the same time
  - Respond to a single command
  - Share address and command buses, but provide different data

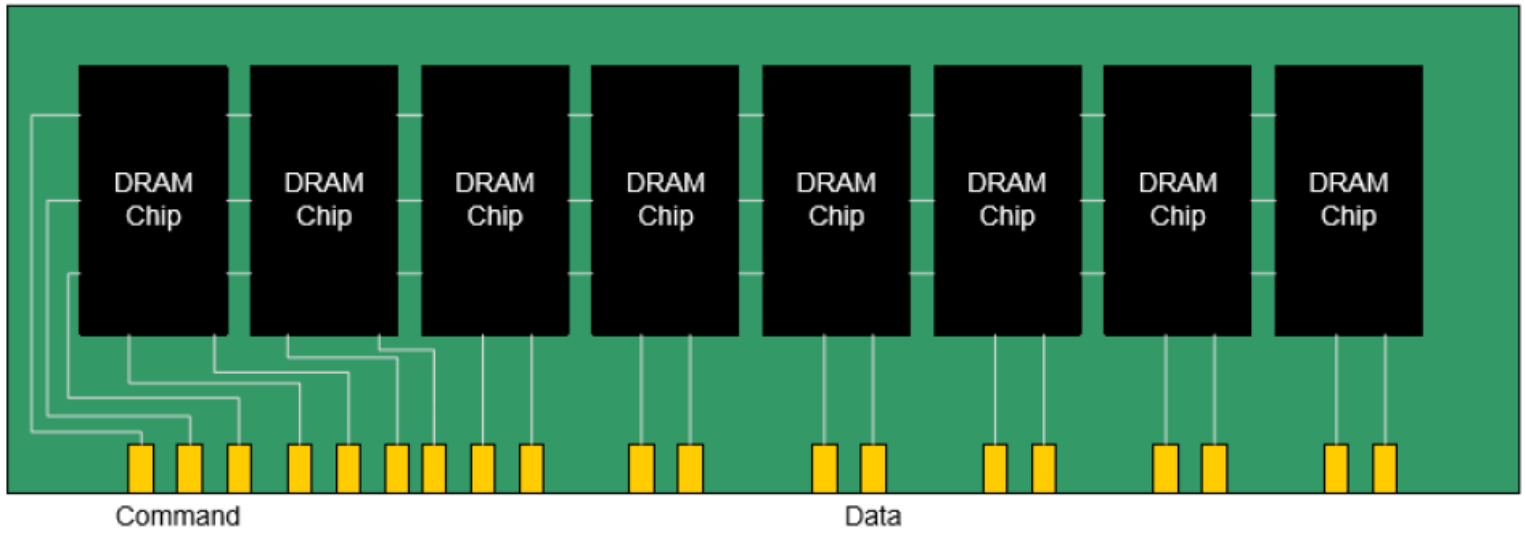
DRAM Rank



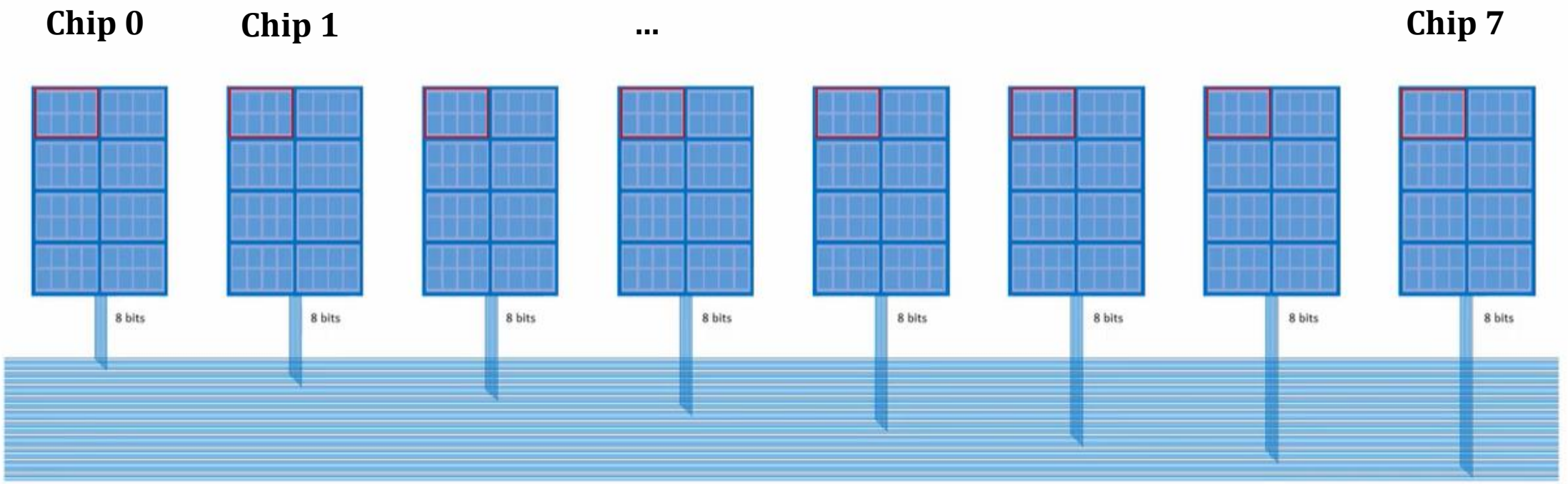
- Easy to produce 8 bit chip than 32 bit chip
- Produce an 8 bit chip but control and operate them as a rank to get a 32 bit data one shot

DRAM Module

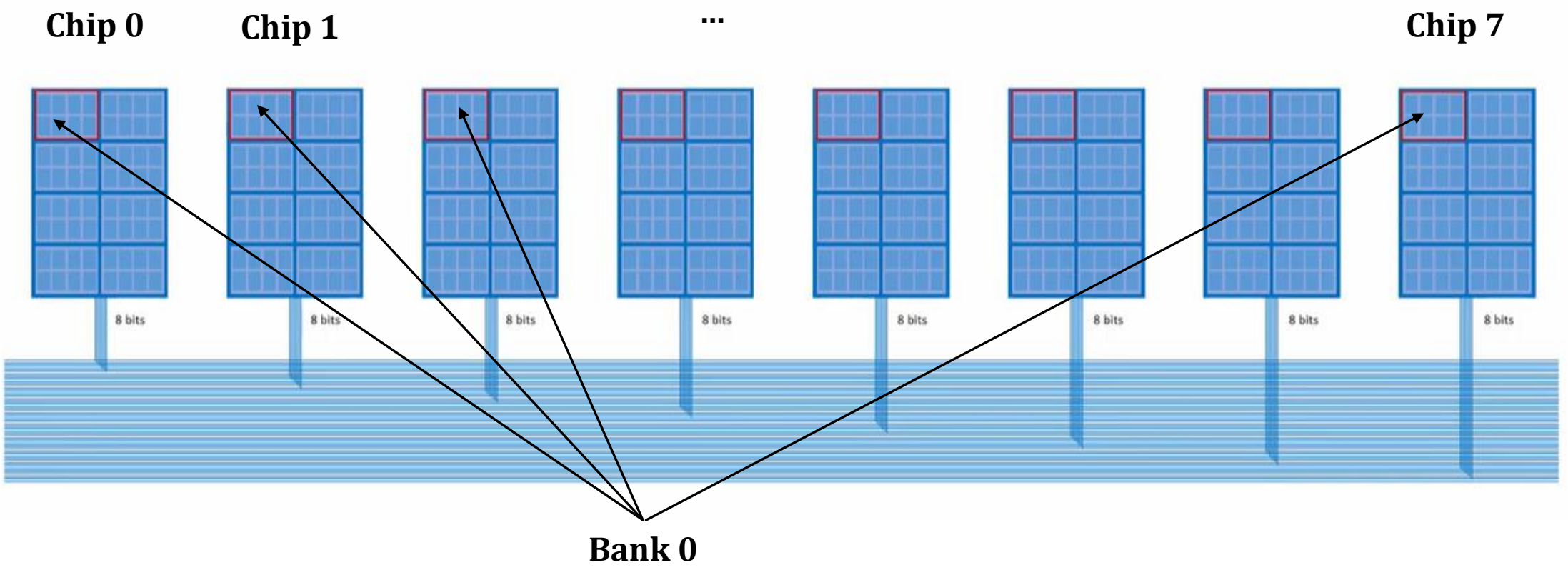
- A DRAM module consists of one or more ranks
  - e.g., DIMM (dual inline memory module)
  - This is what we plug into the motherboard
- If we have chips with 8-bit interface, to read 8 bytes in a single access, use 8 chips in a DIMM



DRAM Module

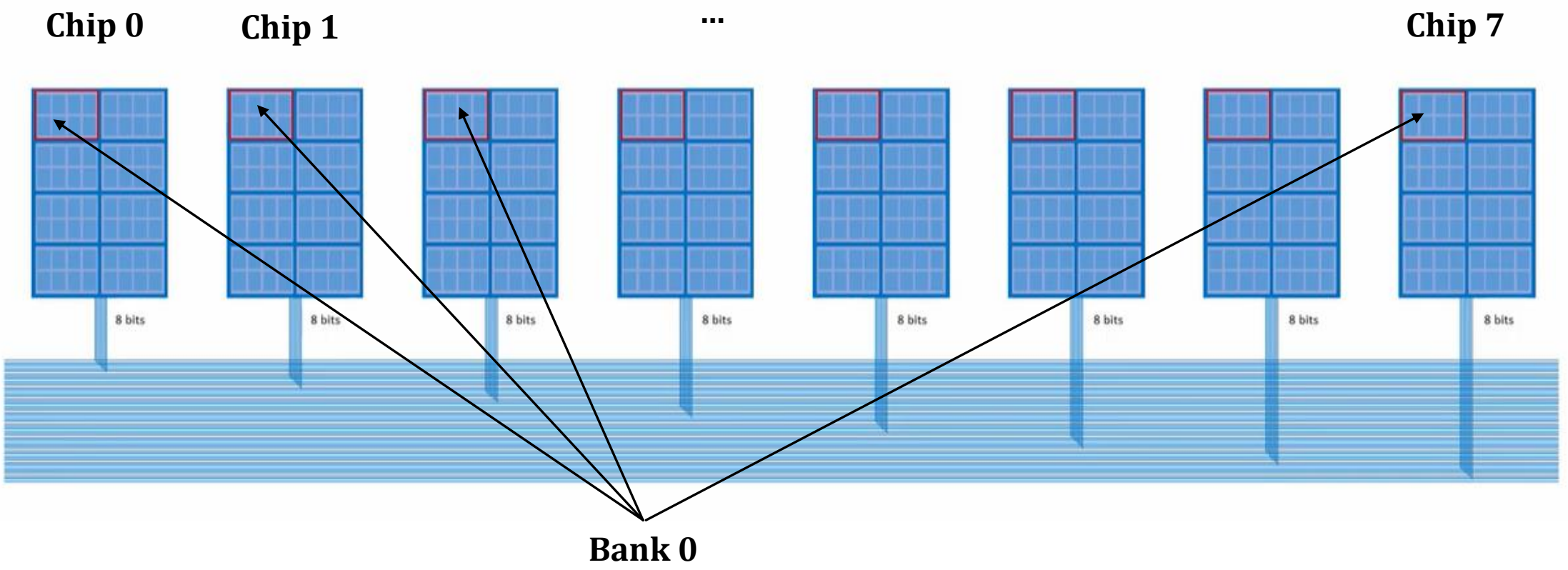

Chip 0
Chip 1
...
Chip 7
8 bits
8 bits
8 bits
8 bits
8 bits
8 bits
8 bits
8 bits

DRAM Module


Chip 0
Chip 1
...
Chip 7
8 bits
8 bits
8 bits
8 bits
8 bits
8 bits
8 bits
8 bits
Bank 0

DRAM Module

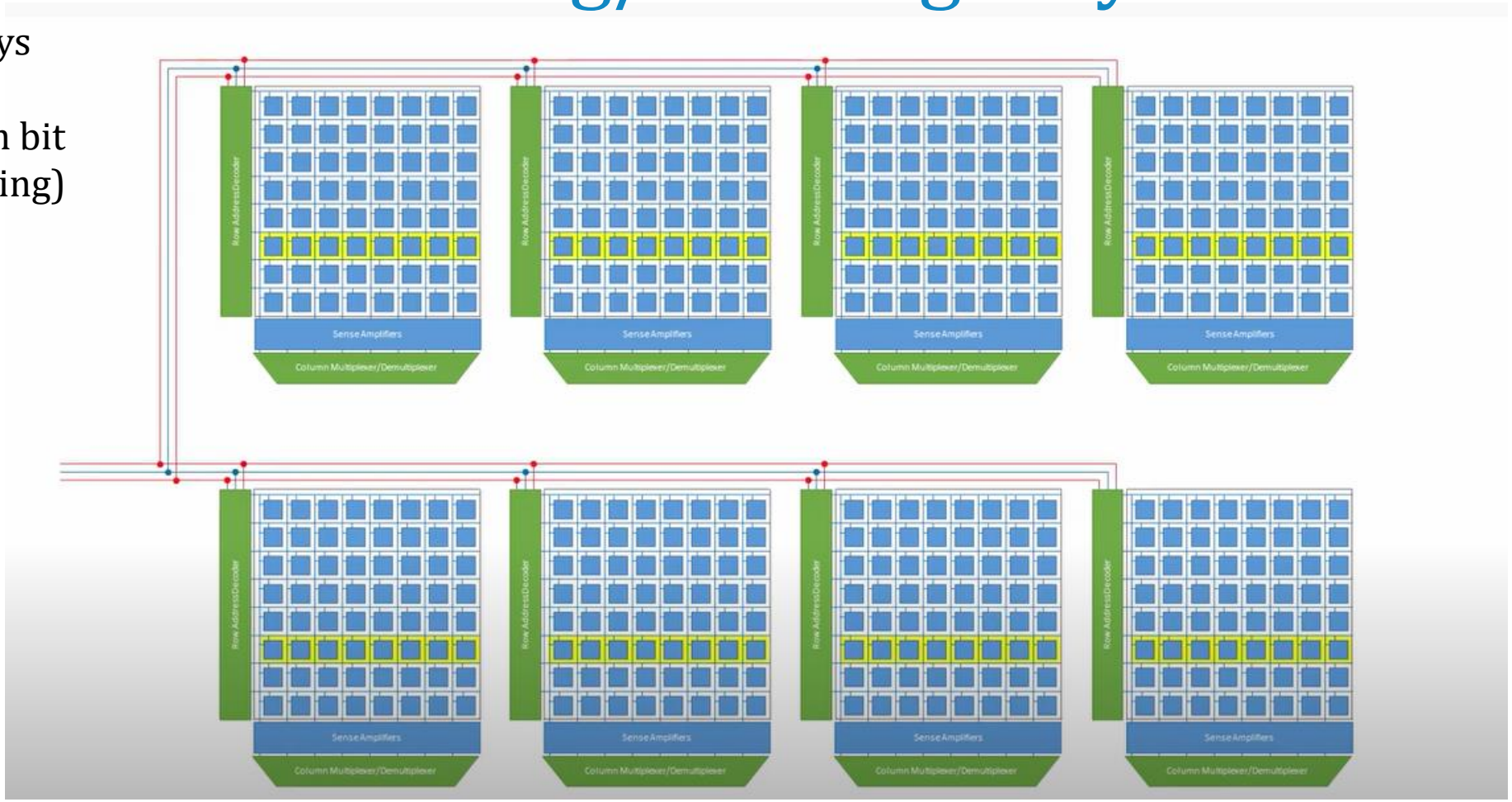
All these chips belong to a specific rank



Reading/ Writing a Byte

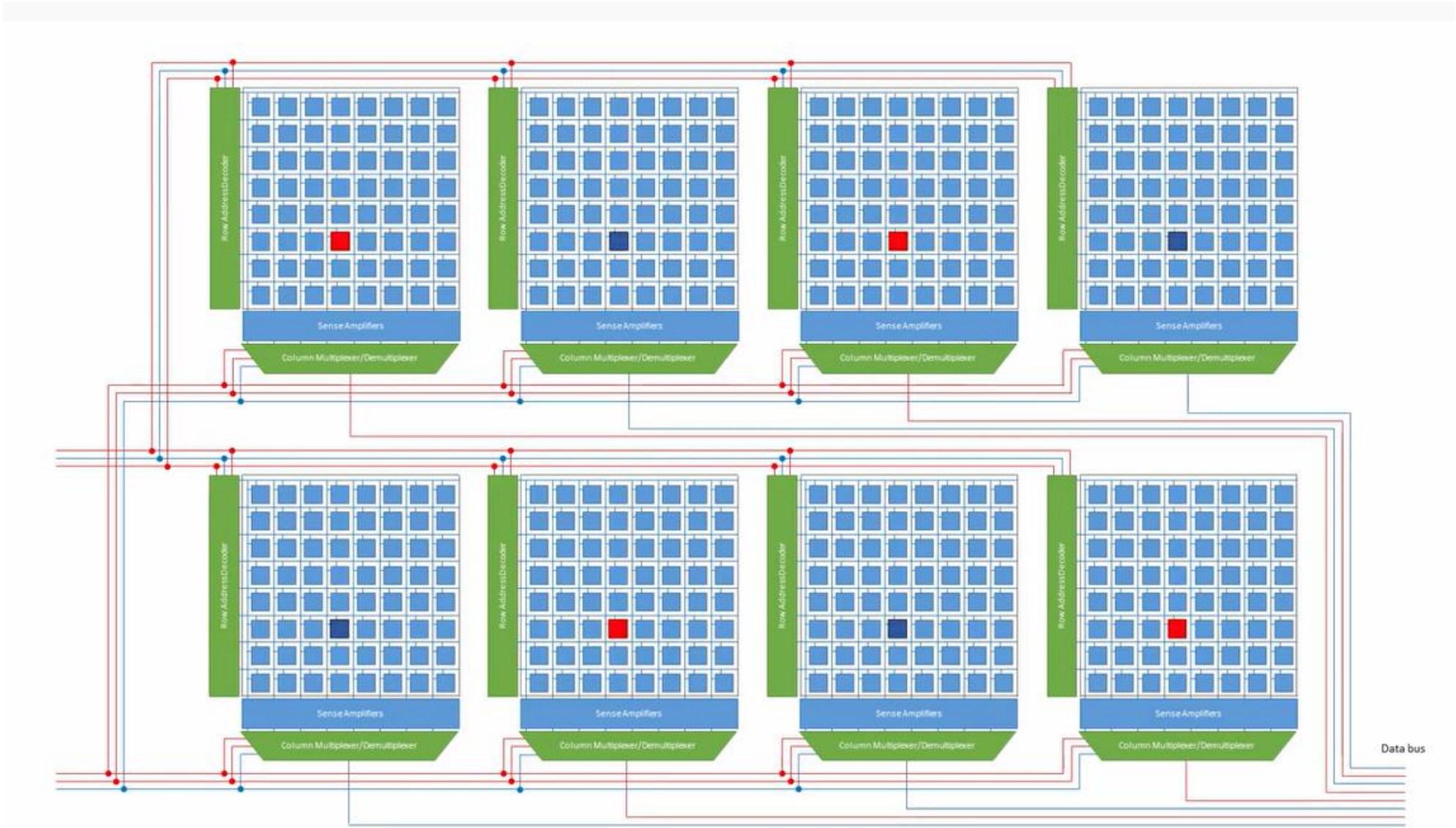
Multiple Arrays

(n arrays for n bit reading/ writing)



https://www.youtube.com/watch?v=Mhqi70OPW0o

Reading/ Writing a Byte



https://www.youtube.com/watch?v=Mhqi70OPW0o