Revised: 7/28/2024

Memory Architecture II

Michael C. Hackett

Assistant Professor, Computer Science

COMMUNITY COLLEGE OF PHILADELPHIA

Lecture Topics

- Cache Memory
  - Direct Mapping
  - N-way Associative Set Mapping
  - Fully Associative Set Mapping

- Dependability
  - Parity Bits
  - Error Correction Code

Cache Memory

- **Cache memory** is SRAM in the processor that holds:
  - The most frequently used data
  - The most recently accessed data

- Processors will often have several levels of cache memory.
  - **Level 1 (L1) Cache** – smallest, fastest cache
  - **Level 2 (L2) Cache** – larger, slower than the L1 cache
  - **Level 3 (L3) Cache** – larger, slower than the L2 cache
  - And so on…

Cache Memory

- Cache memory holds copies of data from main memory in units called blocks
  - Each block has a fixed size of bytes

- For example, a cache memory could have 512 blocks that each store 128 bytes
  - $2^9\ blocks\ \times 2^7\ \frac{bytes}{block} = 2^{16}\ bytes = 64KiB$

Cache Memory – Direct Mapping

- In a **direct mapped** cache, each block of main memory is mapped to a block in cache memory.

- Main memory is much larger than the cache, so multiple blocks of main memory will map to the same block in the cache

Cache Memory – Direct Mapping

- To calculate the cache block that corresponds to a main memory block:

$$M_b \ mod \ B_c$$

  - Where M_b is the block number of main memory block
  - Where B_c is the total number of blocks in in the cache

Cache Memory – Direct Mapping

- When a word is to be *read/accessed* from main memory, the CPU first checks the cache to see if the block that contains the word is already in the cache.
  - If so, it simply obtains that data (**Cache Hit**)
  - If not, then the block is copied from main memory to its appropriate cache block (**Cache Miss**)

Cache Memory – Direct Mapping

- When a word is to be *stored* to main memory, the CPU first checks the cache to see if that main memory address is already mapped in the cache.
  - If so, it simply stores that value in that block of the cache
  - If not, then the main memory block is copied to its appropriate cache block. The word is then written to that cache block.

- Either way, the data in the updated cache block must be copied back to main memory
  - Can be accomplished in one of two ways

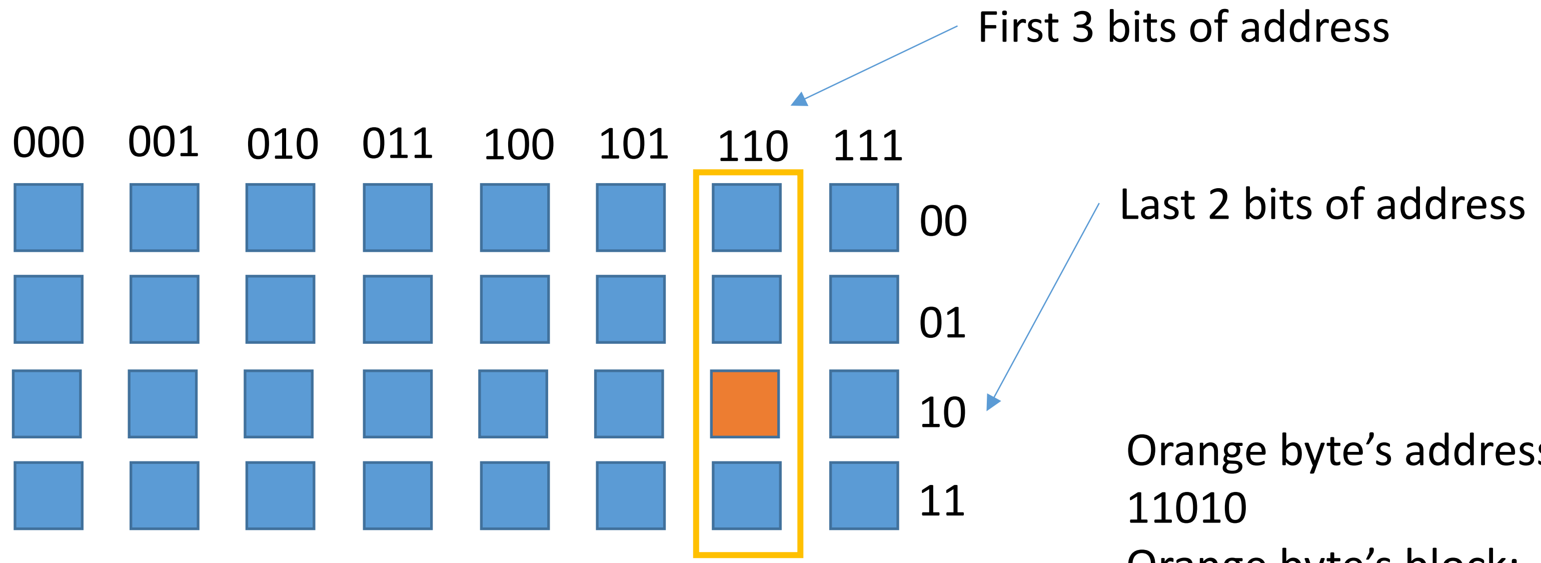
Cache Memory – Direct Mapping

- One way is that whenever a write operation occurs, the new data in the cache is written back to its corresponding address in main memory

- Another way is the cache stores a **dirty bit** (or **valid bit**) for each cache block
    - When the content of the block is changed, the valid bit is set to 1
    - When a new main memory block is to be copied to cache, the valid bit of the corresponding cache block is checked.
        - If 1, it writes the cache block to its corresponding main memory block, then loads the new main memory block (and resets the valid bit to 0)

Cache Memory – Direct Mapping

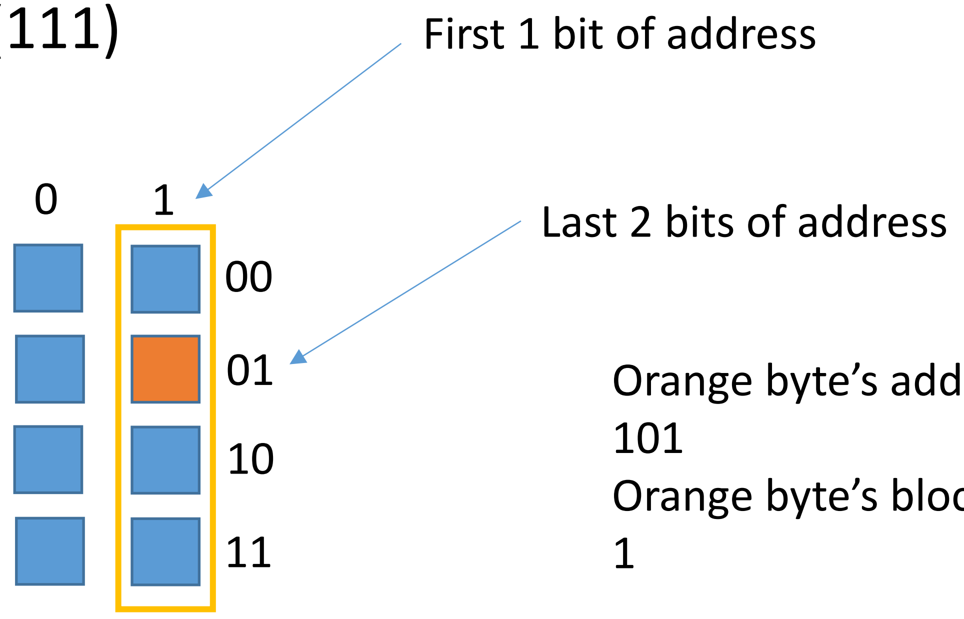
- For illustration, here is a system with 32 bytes of <u>main memory</u>
  - Each square represents one byte
  - Each column is a block
  - Addresses 0 (00000) through 31 (111111)



Orange byte's address:
11010
Orange byte's block:
110 (6)

Cache Memory – Direct Mapping

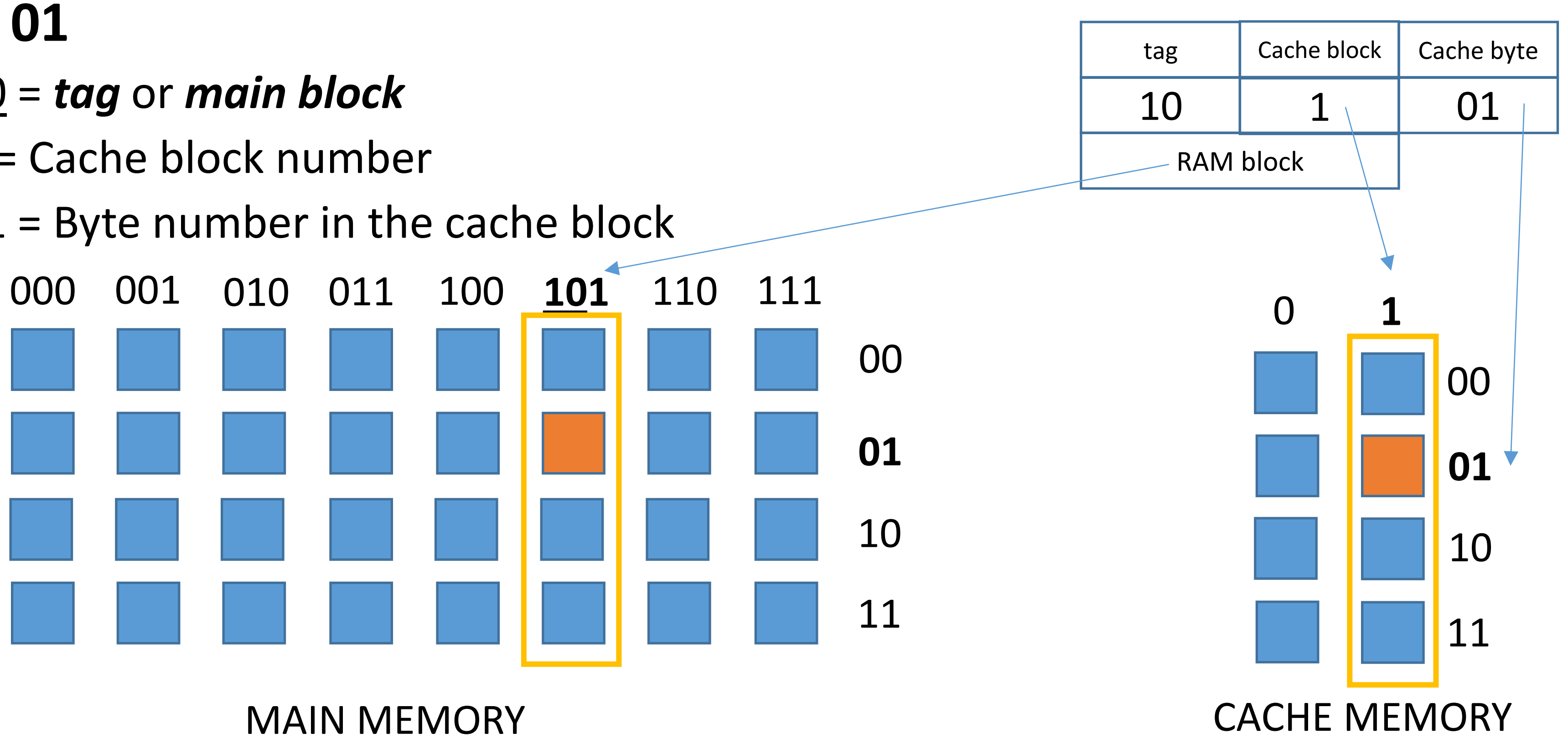
- We'll consider a simple CPU that has 16 bytes of cache memory
    - Each square represents one byte
    - Each column is a block
    - Addresses 0 (000) through 7 (111)



Orange byte's address:
101
Orange byte's block:
1

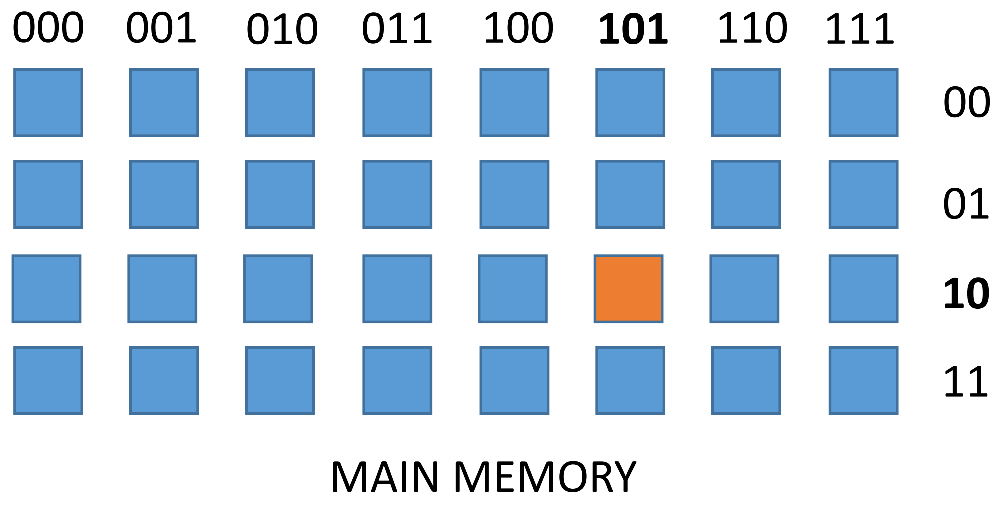
Cache Memory – Direct Mapping

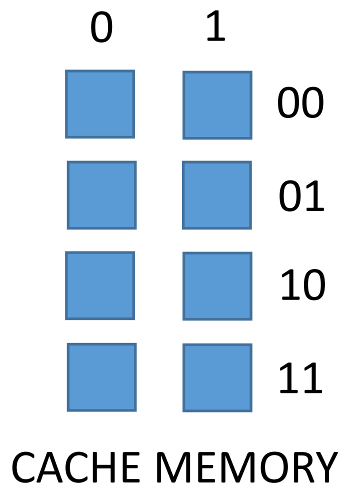
- Let’s see how the main memory address 10101 maps to the cache
  - **<u>10</u> 1 01**
    - <u>10</u> = ***tag*** or ***main block***
    - 1 = Cache block number
    - 01 = Byte number in the cache block



Cache Memory – Direct Mapping

- The CPU is instructed to read the byte at address **10110** from main memory
  - The orange byte below

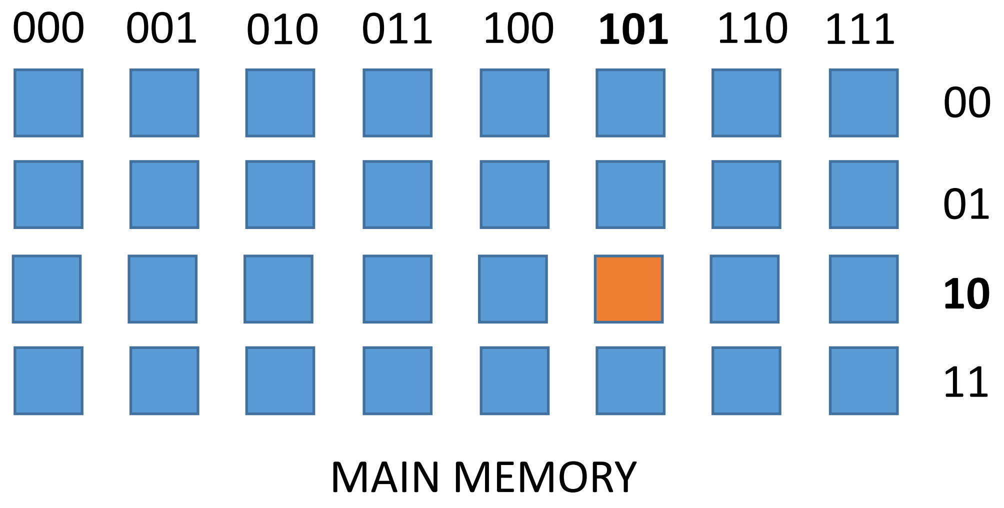


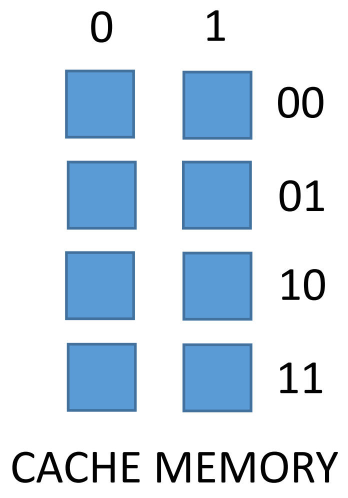


Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
	1 00	
	1 01	
	1 10	
	1 11	

Cache Memory – Direct Mapping

- First the CPU checks if the value is in the cache
    - **10 1 10**
    - Not present (*cache miss*)

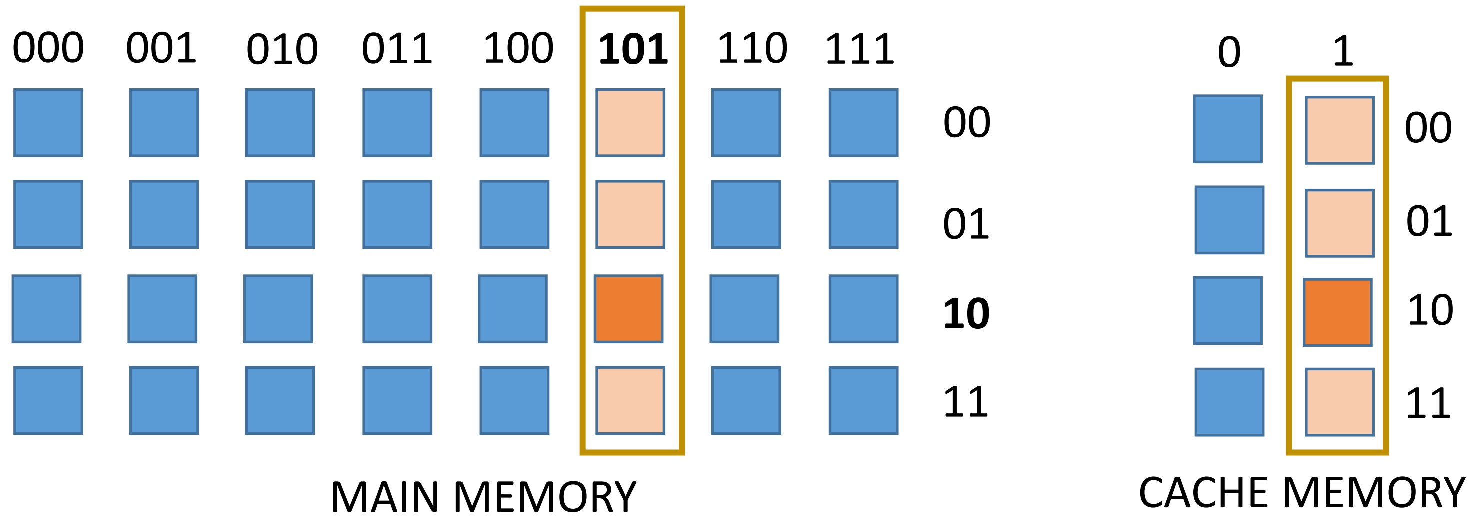




Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
	1 00	
	1 01	
	1 10	
	1 11	

Cache Memory – Direct Mapping

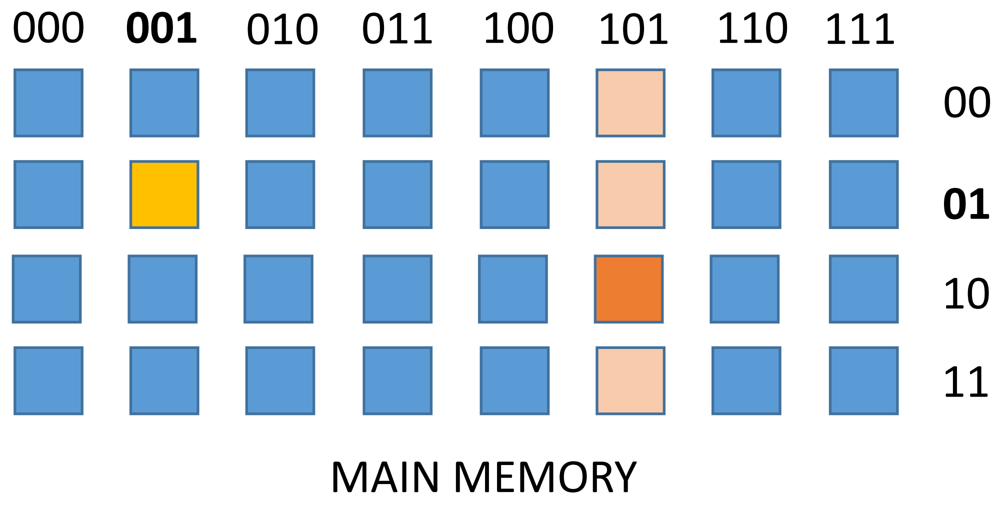
- The block (block 5) is loaded into cache
  - 10 1 00
  - 10 1 01
  - 10 1 10
  - 10 1 11

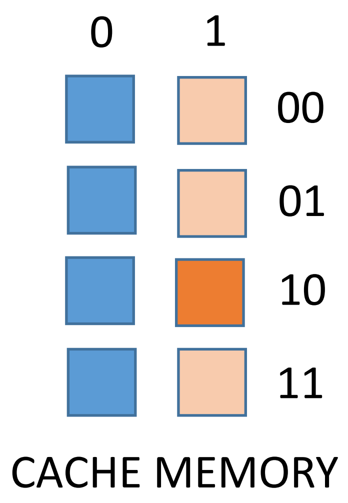


Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
10	1 00	▢ 10100
10	1 01	▢ 10101
10	1 10	■ 10110
10	1 11	▢ 10111

Cache Memory – Direct Mapping

- The CPU is instructed to read the byte at address 00101 from main memory
    - The yellow byte below

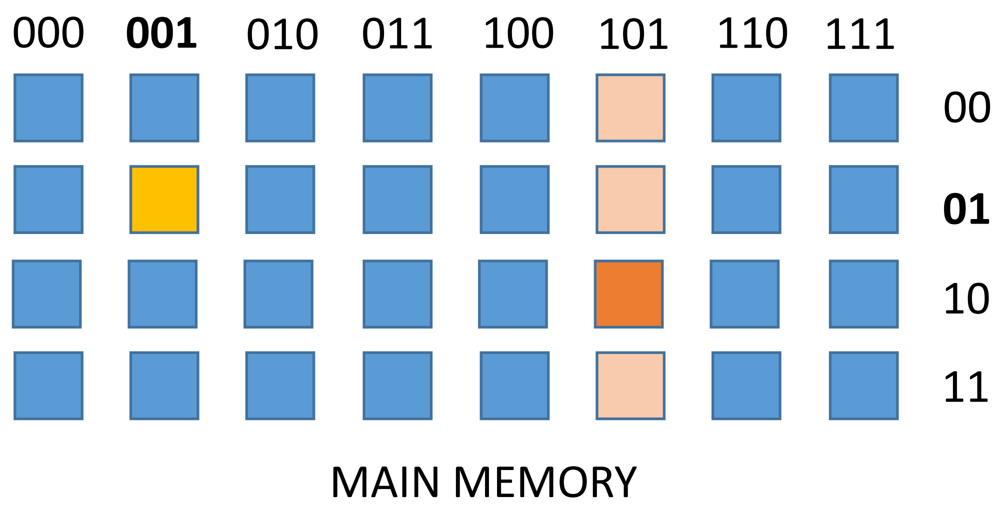


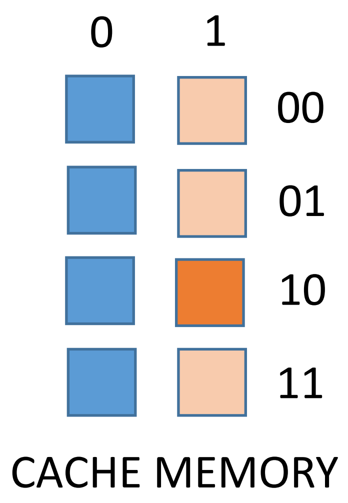


Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
10	1 00	▢ 10100
10	1 01	▢ 10101
10	1 10	🟧 10110
10	1 11	▢ 10111

Cache Memory – Direct Mapping

- First the CPU checks if the value is in the cache
  - **00 1 01**
  - Data in cache address 101, but wrong tag (*cache miss*)

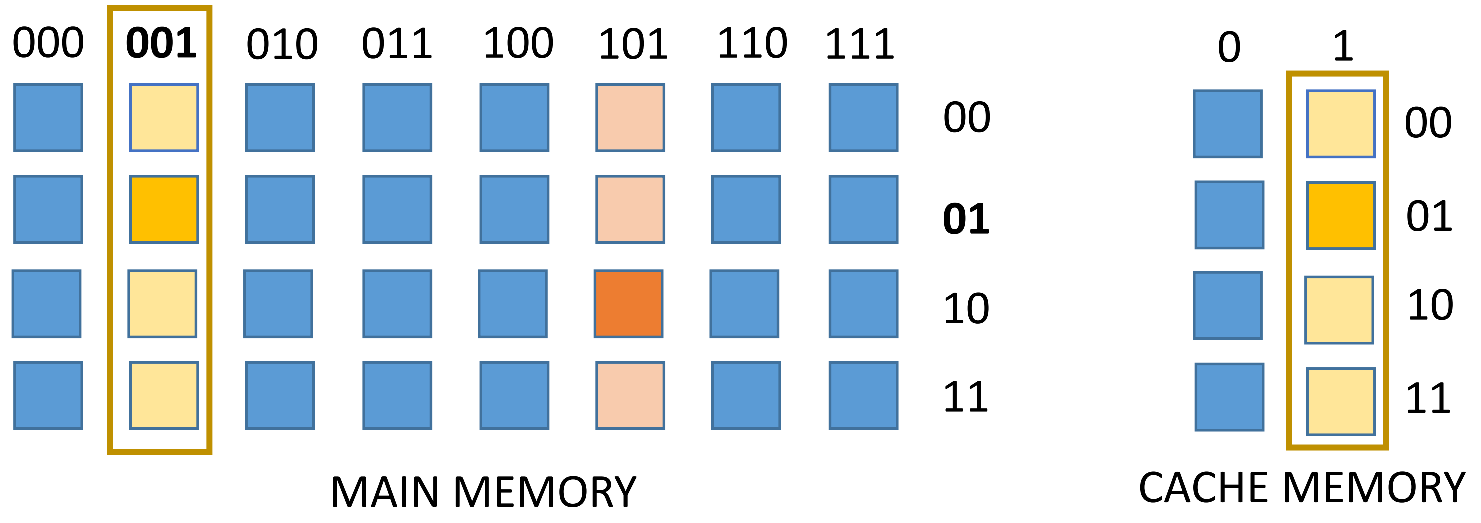




Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
10	1 00	▢ 10100
10	**1 01**	▢ 10101
10	1 10	■ 10110
10	1 11	▢ 10111

Cache Memory – Direct Mapping

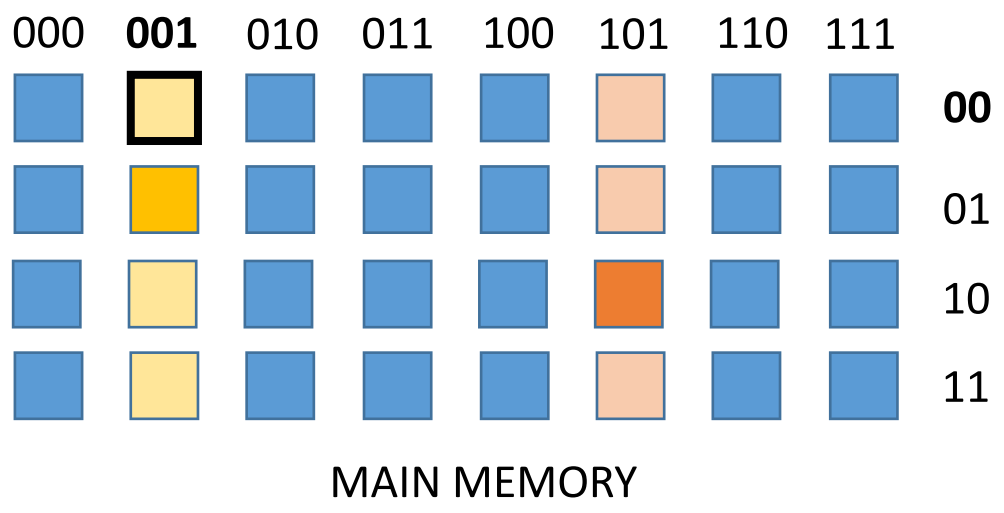
- The block (block 1) is loaded into cache
  - 00 1 00
  - 00 1 01
  - 00 1 10
  - 00 1 11

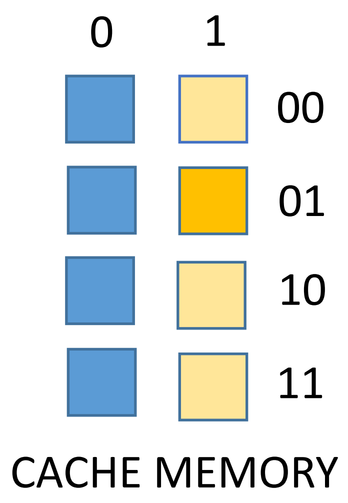


Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
00	1 00	🟨 00100
00	1 01	🟧 00101
00	1 10	🟨 00110
00	1 11	🟨 00111

Cache Memory – Direct Mapping

- The CPU is instructed to read the byte at address **00100** from main memory

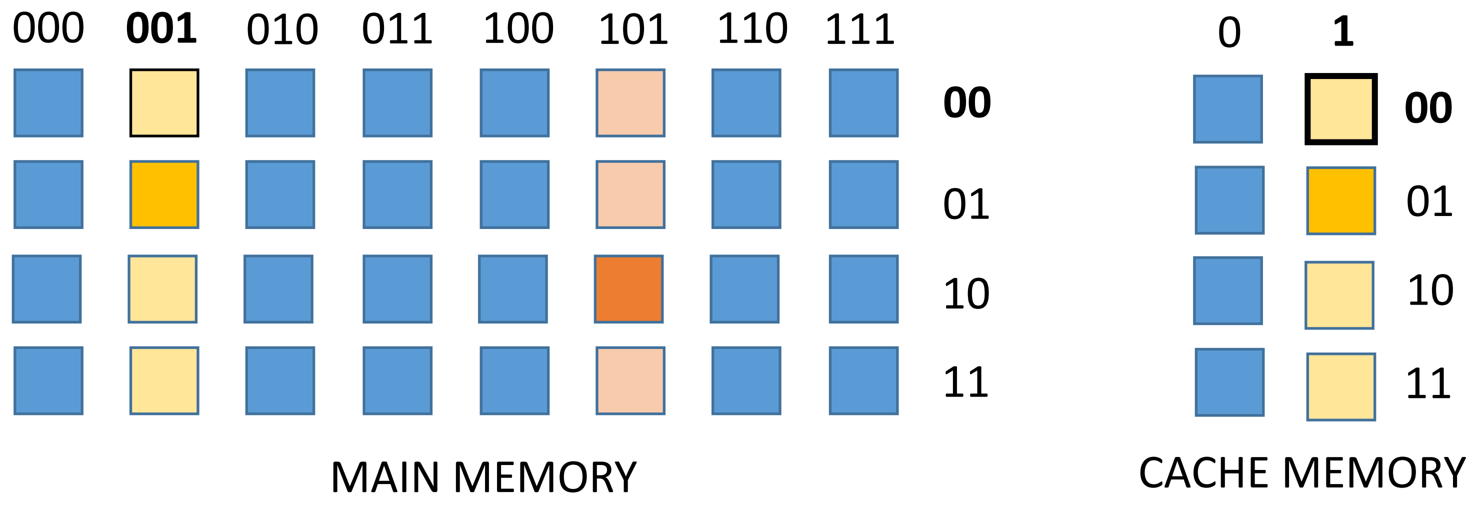




Tag	Cache Address	Data/Main Memory Address
	Set Byte	
	0 00	
	0 01	
	0 10	
	0 11	
00	1 00	🟨 00100
00	1 01	🟧 00101
00	1 10	🟨 00110
00	1 11	🟨 00111

Cache Memory – Direct Mapping

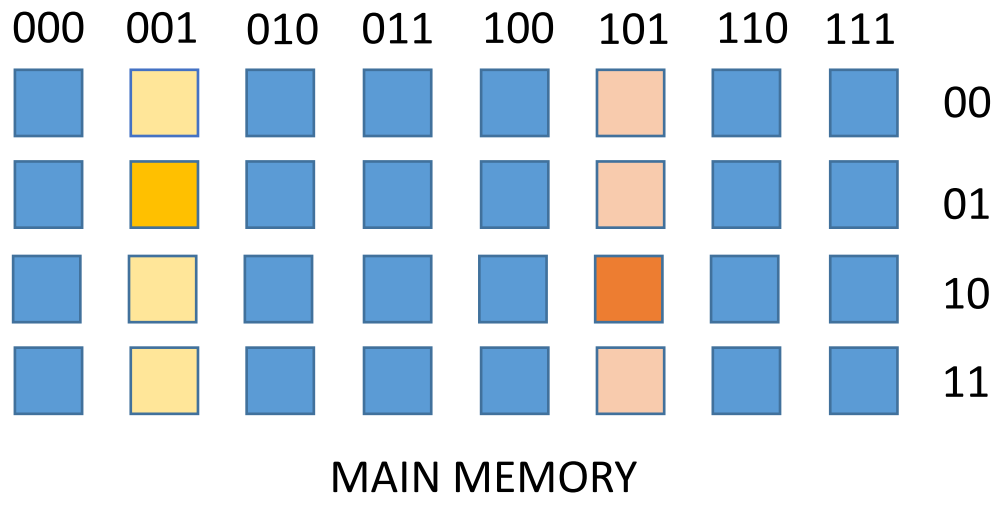
- First the CPU checks if the value is in the cache
  - **00 1 00**
  - Data in cache address 100 with correct tag (*cache hit*)
  - Does not retrieve from main memory

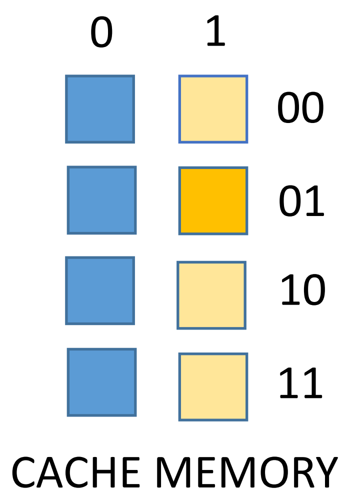


Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
00	**1 00**	🟨 00100
00	1 01	🟧 00101
00	1 10	🟨 00110
00	1 11	🟨 00111

Cache Memory – Direct Mapping

- The CPU is instructed to store a byte to address 00111 in main memory
  - Will be represented as 🟩

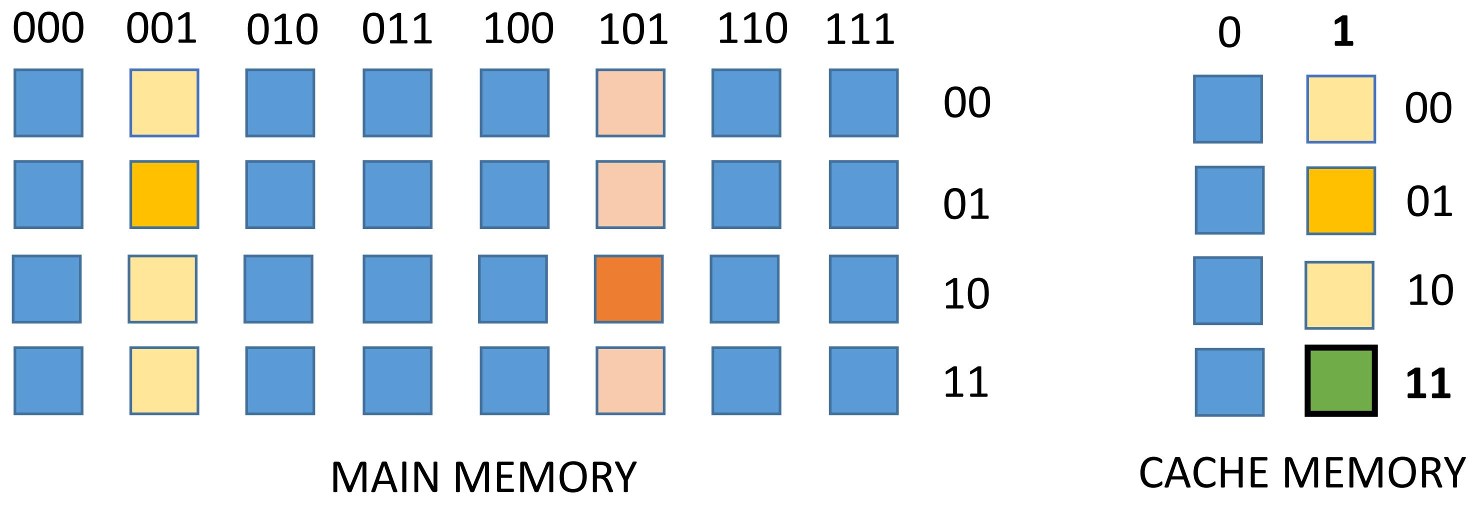




Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
00	1 00	🟨 00100
00	1 01	🟧 00101
00	1 10	🟨 00110
00	1 11	🟨 00111

Cache Memory – Direct Mapping

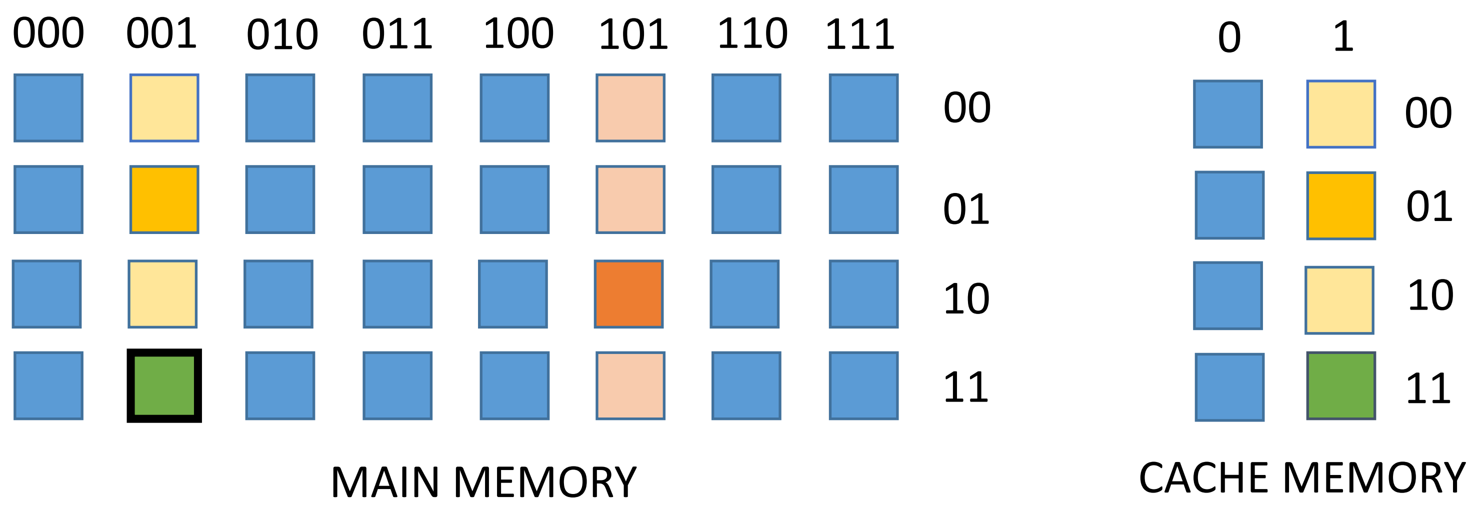
- The CPU writes the value to the cache
  - 00 **1 11**
  - Data in cache address 111 with correct tag (*cache hit*)
    - Data in cache is updated



Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
00	1 00	🟨 00100
00	1 01	🟧 00101
00	1 10	🟨 00110
00	1 11	🟩 00111

Cache Memory – Direct Mapping

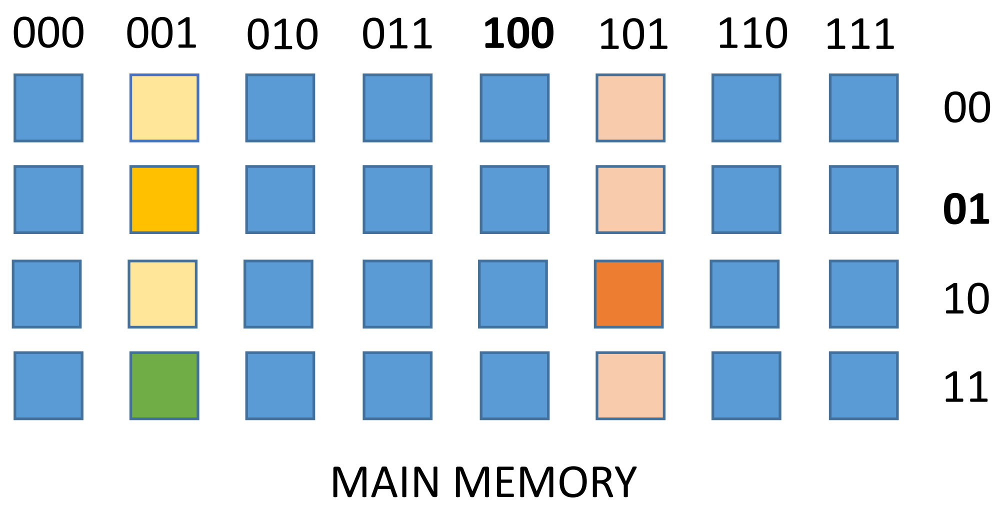
- Data is stored to main memory (or dirty bit is set)

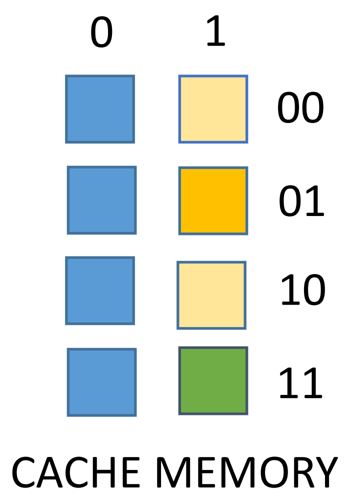


Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
00	1 00	🟨 00100
00	1 01	🟧 00101
00	1 10	🟨 00110
00	1 11	🟩 00111

Cache Memory – Direct Mapping

- The CPU is instructed to store a byte to address 10001 in main memory
  - Will be represented as 🟪

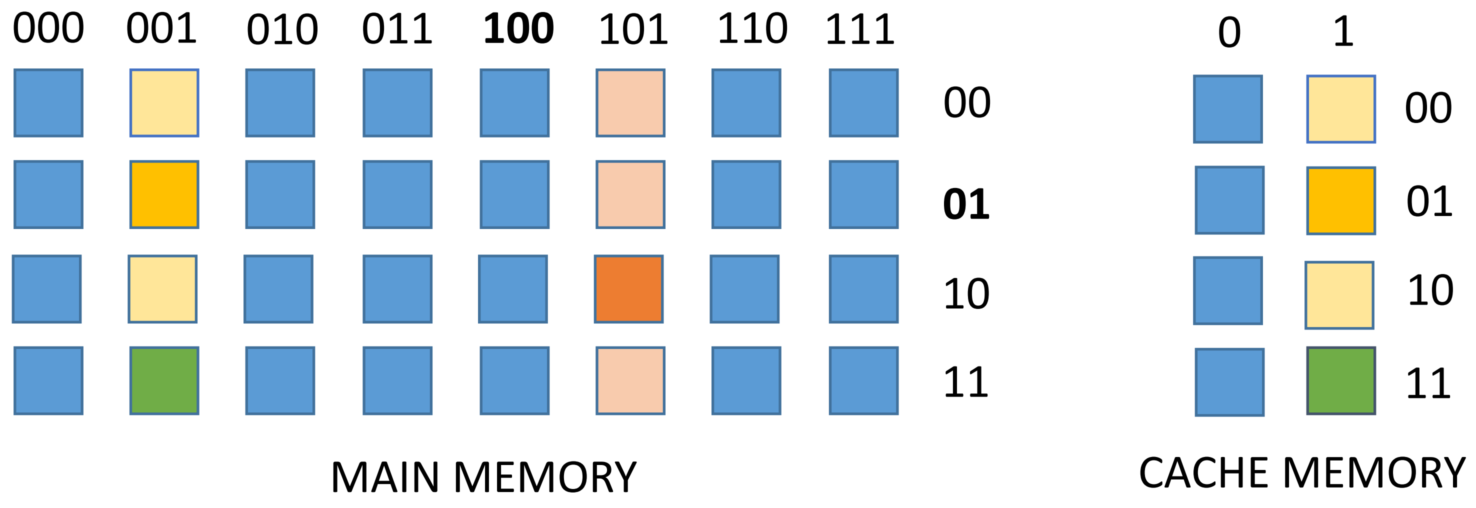




Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
00	1 00	🟨 00100
00	1 01	🟧 00101
00	1 10	🟨 00110
00	1 11	🟩 00111

Cache Memory – Direct Mapping

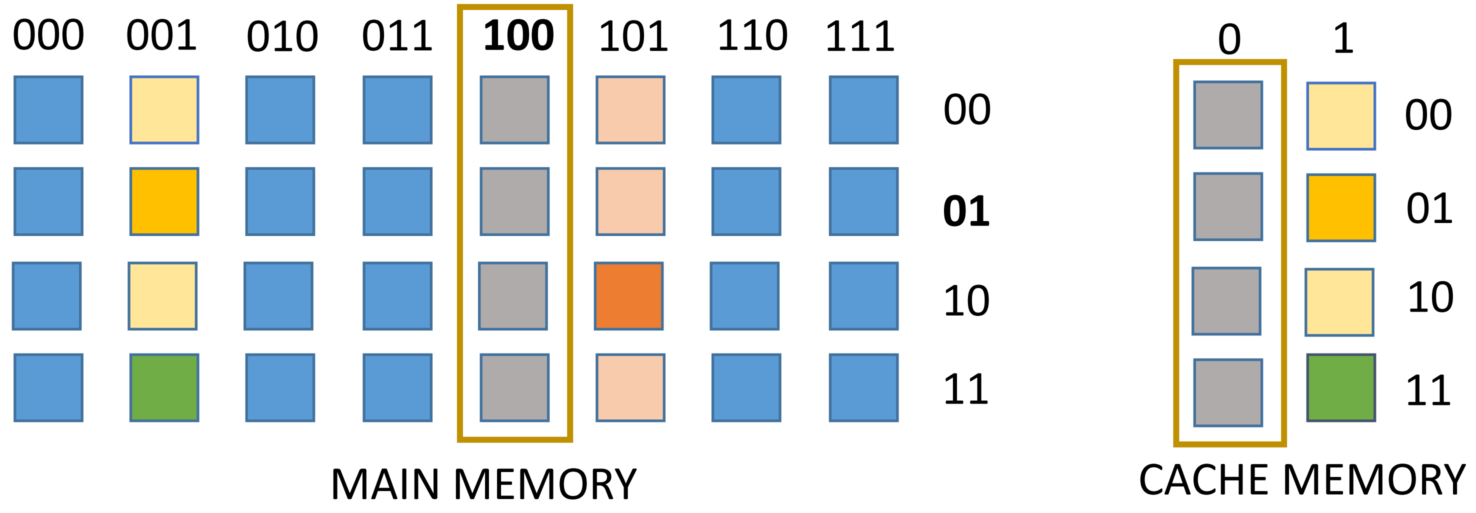
- First the CPU checks if the value is in the cache
  - **10 0 01**
  - No data in cache address 001 (cache miss)



Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
00	1 00	🟨 00100
00	1 01	🟧 00101
00	1 10	🟨 00110
00	1 11	🟩 00111

Cache Memory – Direct Mapping

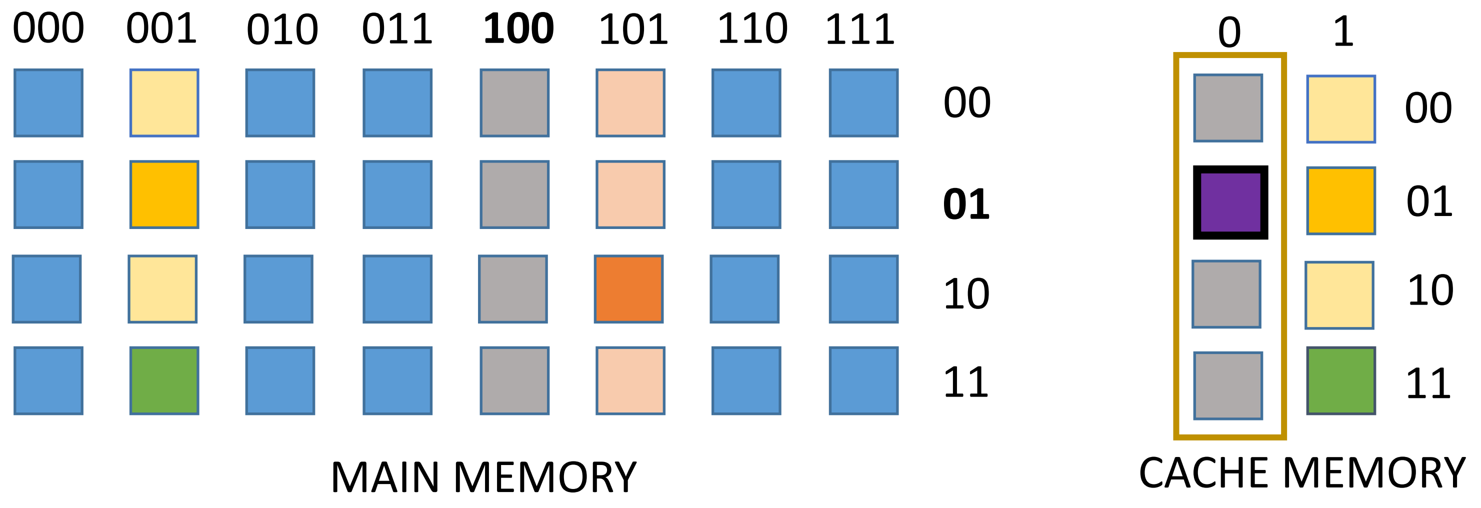
- The block (block 4) is loaded into cache
  - 10 0 00
  - 10 0 01
  - 10 0 10
  - 10 0 11



Tag	Cache Address *Set Byte*	Data/Main Memory Address
10	0 00	◼ 10000
10	0 01	◼ 10001
10	0 10	◼ 10010
10	0 11	◼ 10011
00	1 00	🟨 00100
00	1 01	🟧 00101
00	1 10	🟨 00110
00	1 11	🟩 00111

Cache Memory – Direct Mapping

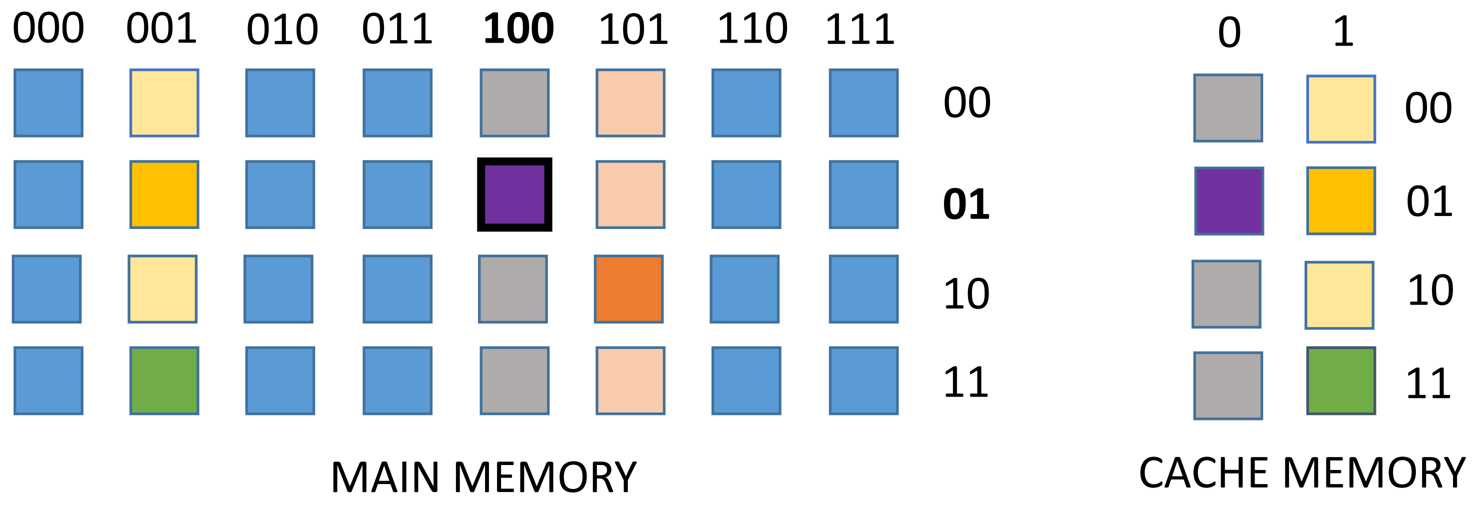
- The CPU writes the value to the cache
  - 10 **0 01**
  - Data in cache is updated



Tag	Cache Address *Set Byte*	Data/Main Memory Address
10	0 00	⬜ 10000
10	**0 01**	🟪 10001
10	0 10	⬜ 10010
10	0 11	⬜ 10011
00	1 00	🟨 00100
00	1 01	🟧 00101
00	1 10	🟨 00110
00	1 11	🟩 00111

Cache Memory – Direct Mapping

- Data is stored to main memory (or dirty bit is set)



Tag	Cache Address *Set Byte*	Data/Main Memory Address
10	0 00	⬜ 10000
10	**0 01**	🟪 10001
10	0 10	⬜ 10010
10	0 11	⬜ 10011
00	1 00	🟨 00100
00	1 01	🟧 00101
00	1 10	🟨 00110
00	1 11	🟩 00111

Cache Memory – Direct Mapping

- Each row in the table below represents the read/write operations illustrated on the previous slides

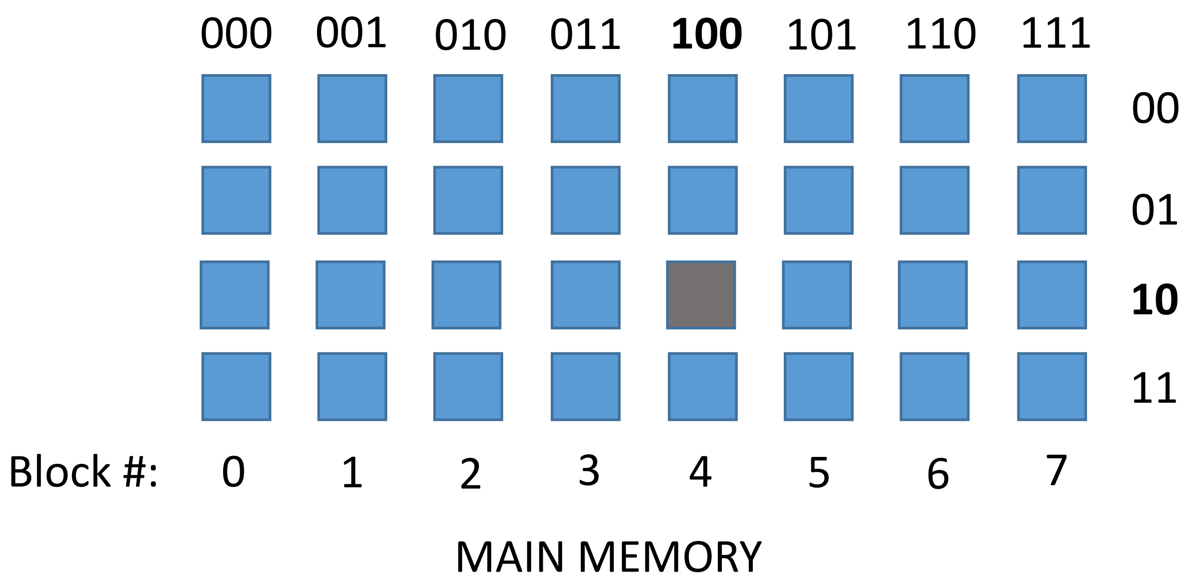
Memory Address	Hit or Miss	Cache Contents							
		0	1	2	3	4	5	6	7
10110	M					🔸	🔸	🟧	🔸
00101	M					🟡	🟨	🟡	🟡
00100	H					🟡	🟨	🟡	🟡
00111	H					🟡	🟨	🟡	🟩
10001	M	◼	🟪	◼	◼	🟡	🟨	🟡	🟩

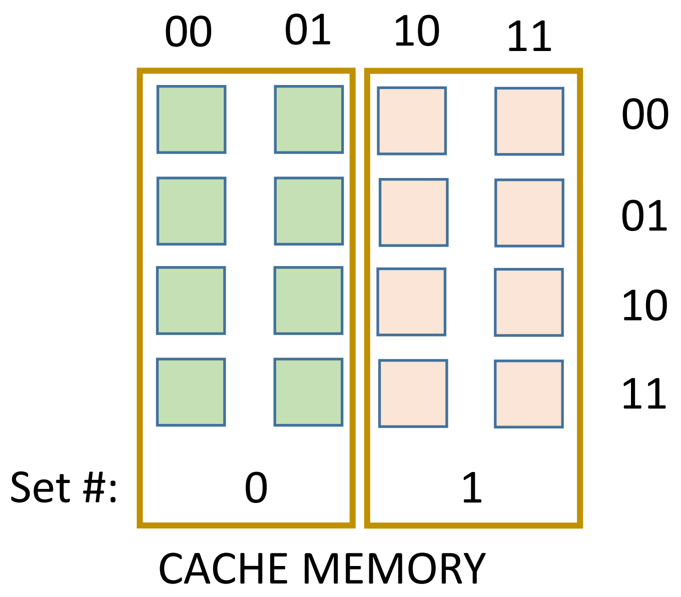
Cache Memory – Associative Set Mapping

- In an **associative mapped** cache, each block of main memory is mapped to a *set of blocks* in cache memory.
- When a block is copied from RAM to the cache, the block can be stored in any one of the blocks that belong to the set.
- An ***n-way set associative cache*** indicates that each cache set contains *n* blocks per set.
  - A 3-way set associative cache has three blocks per set
  - A 1-way set associative cache has one block per set
    - A 1-way set associative cache *is* direct mapping

Cache Memory – Associative Set Mapping

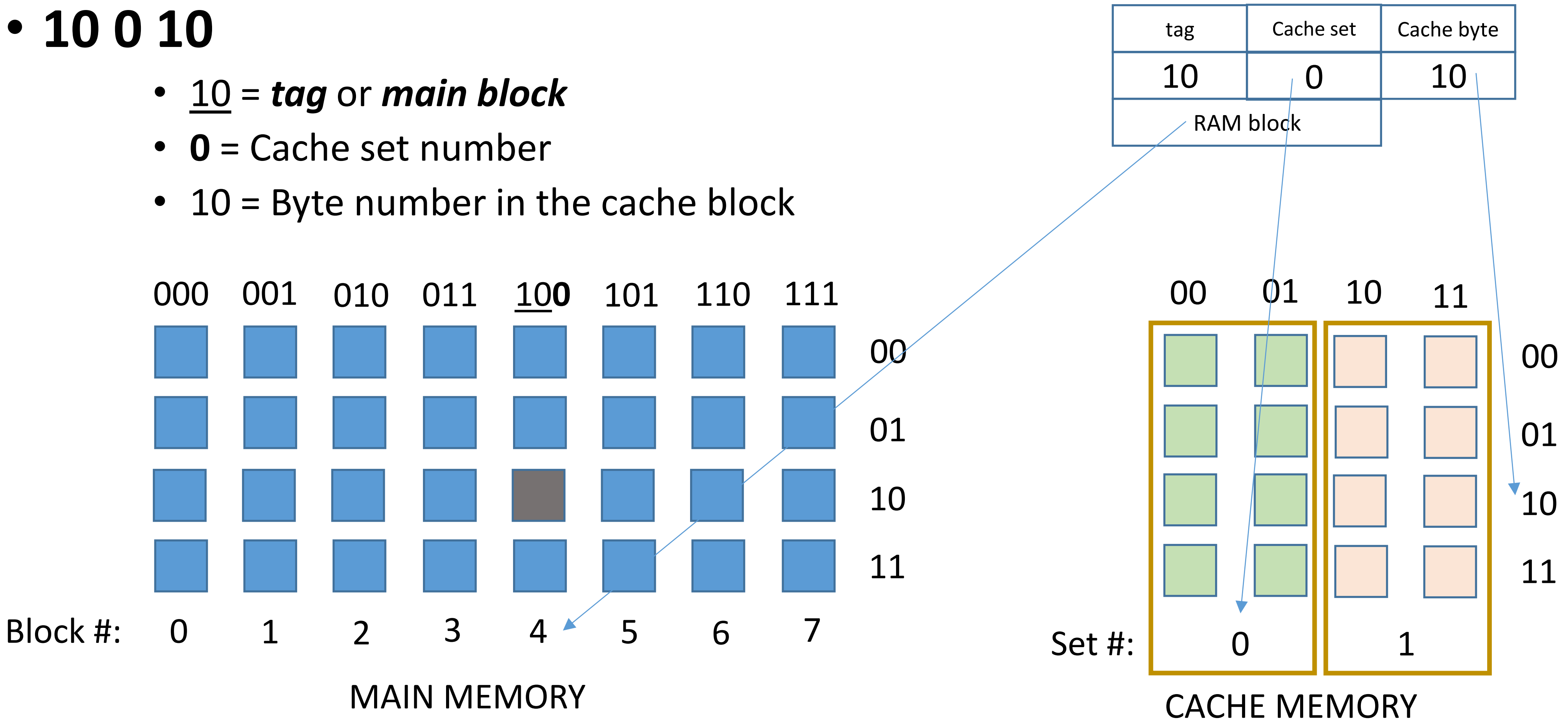
- 2-way associative cache
- Consider the gray square/byte with the address **10010**





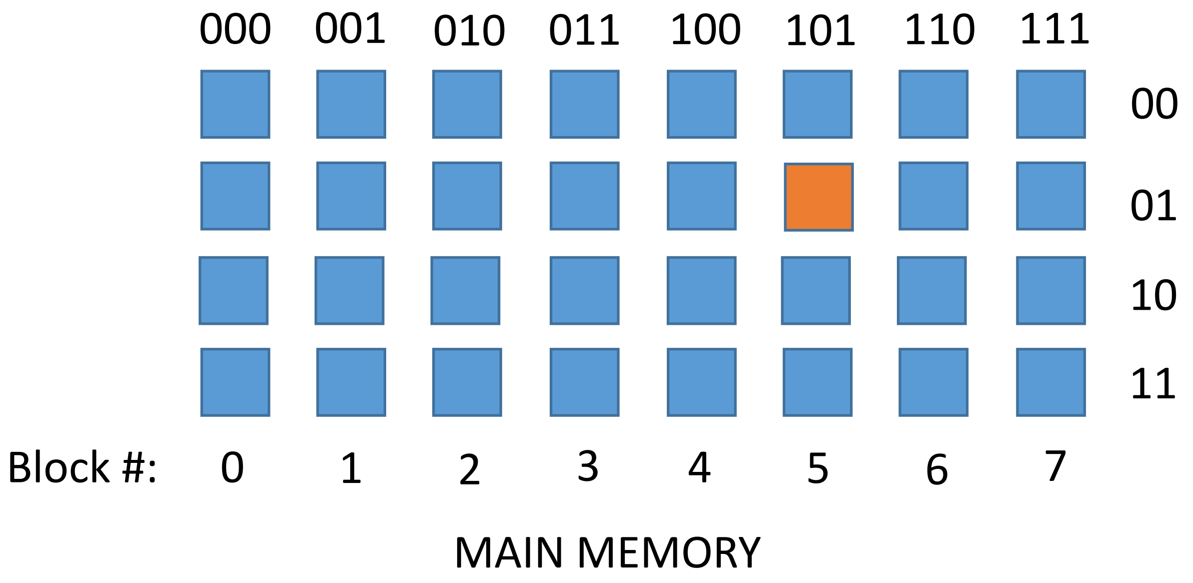
Cache Memory – Associative Set Mapping

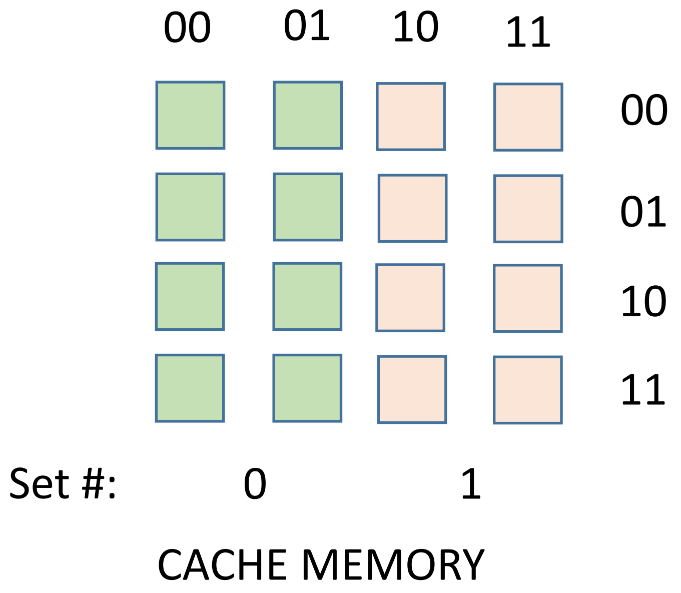
- **10 0 10**
  - 10 = ***tag*** or ***main block***
  - **0** = Cache set number
  - 10 = Byte number in the cache block



Cache Memory – Associative Set Mapping

- The CPU is instructed to read the byte at address **10101** from main memory
  - The orange byte below

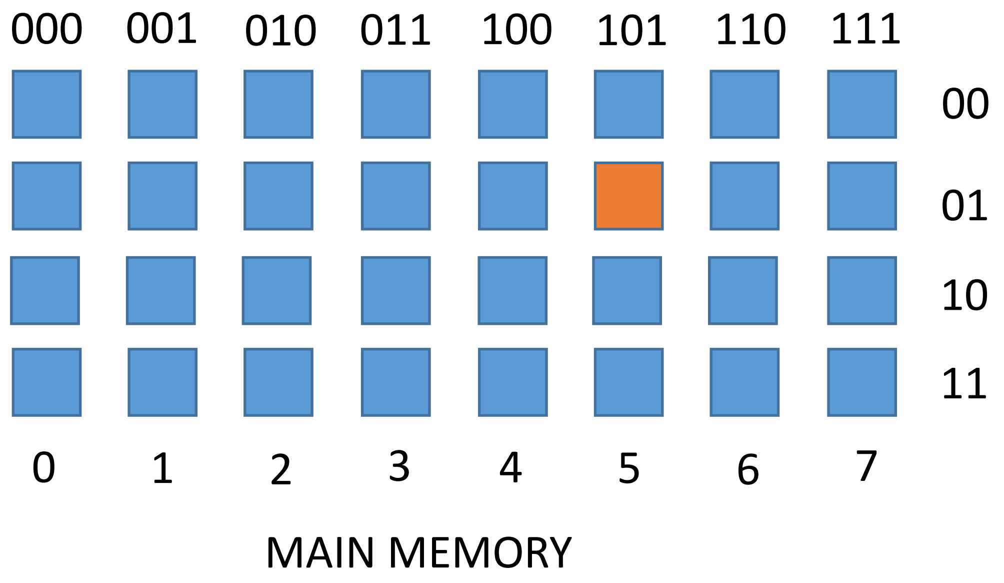


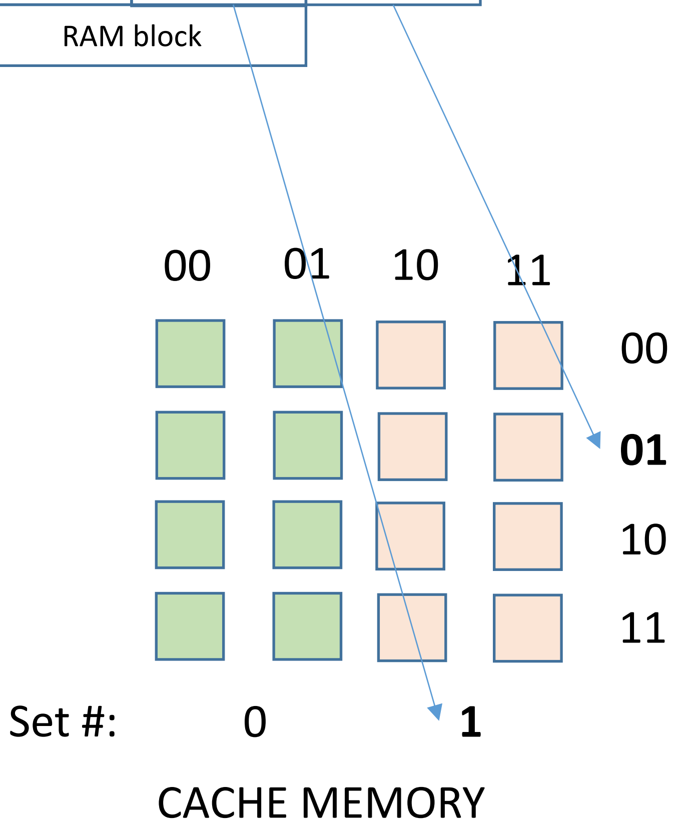


Cache Memory – Associative Set Mapping

- **10 1 01**

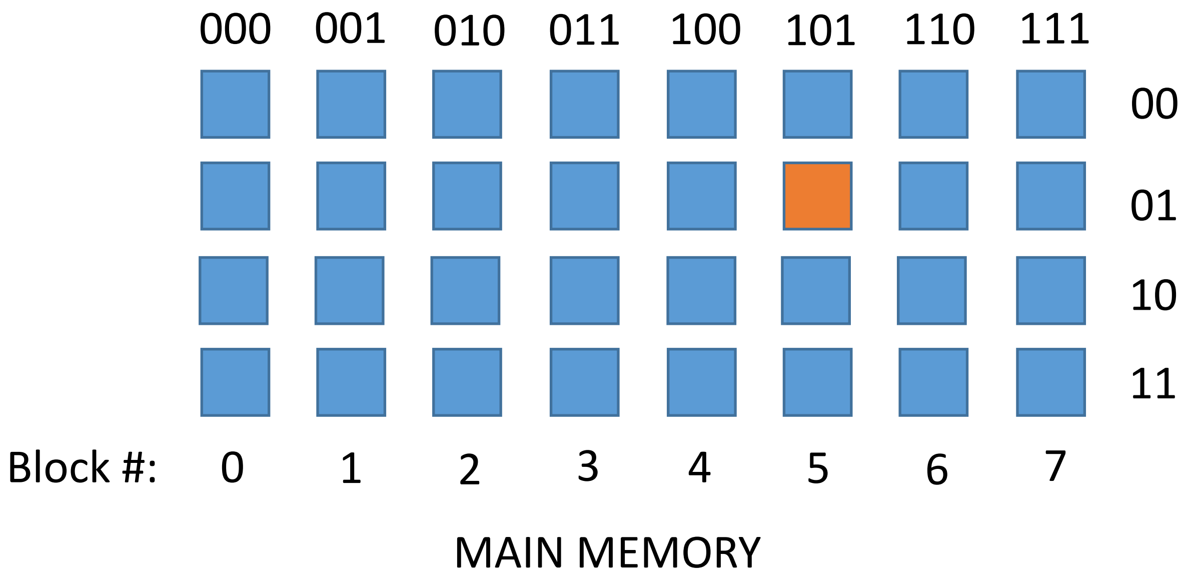
tag	Cache set	Cache byte
10	1	01
RAM block		

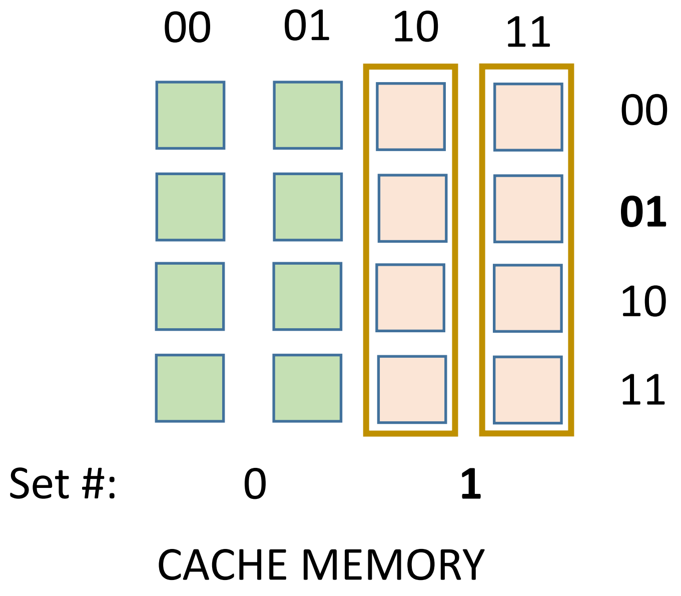




Cache Memory – Associative Set Mapping

- Within this set, there are two blocks to chose from (blocks 2 and 3).
  - Which block is chosen will depend on the *block replacement strategy* implemented





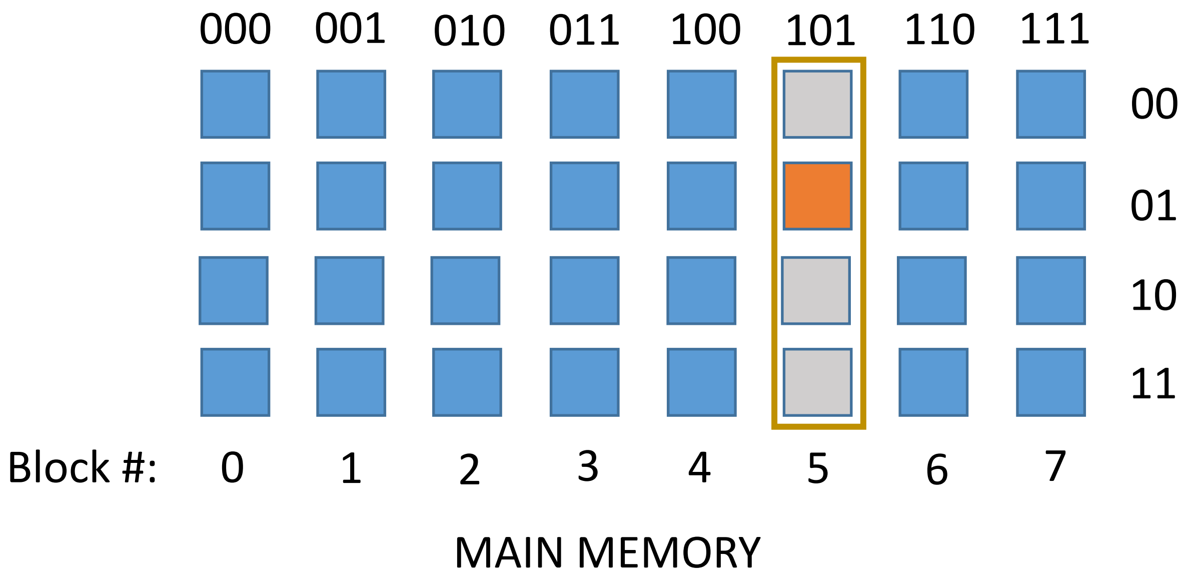
Cache Memory – Associative Set Mapping

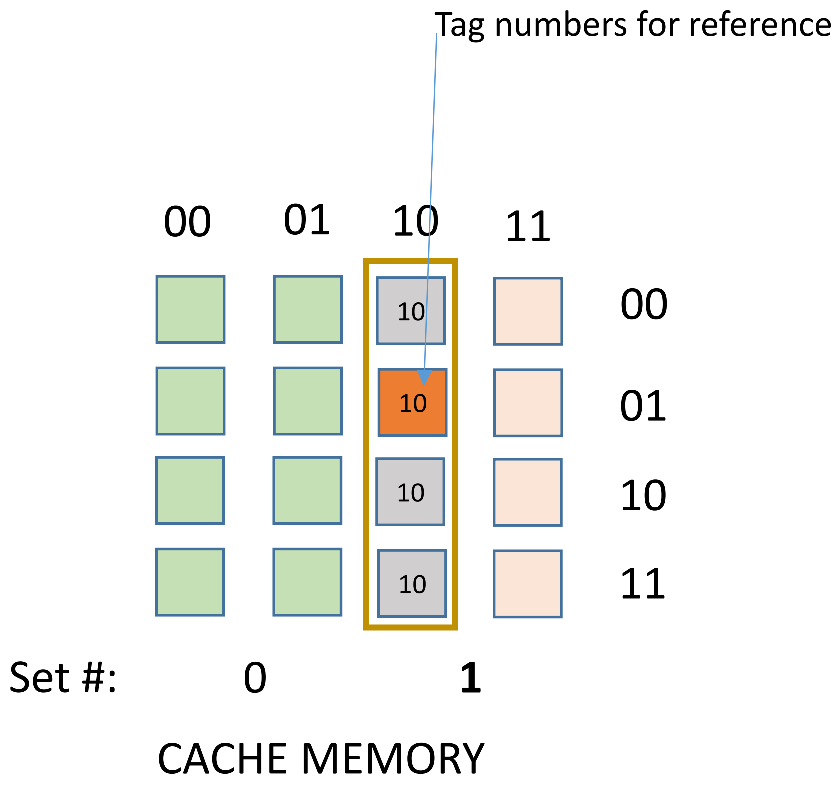
- One such strategy is called **Least Recently Used (LRU)**
  - The block to be replaced in the set is the block that has been sitting, unused, for the longest time.
- Another block replacement strategy is called **First In, First Out (FIFO)**
  - The block to be replaced in the set is the block that has been in the cache for the longest time.
- A third strategy is to randomly choose a block for replacement

- Each strategy has their pros and cons
  - No perfect/optimal block replacement strategy

Cache Memory – Associative Set Mapping

- We'll assume LRU.
  - Block 2 is chosen

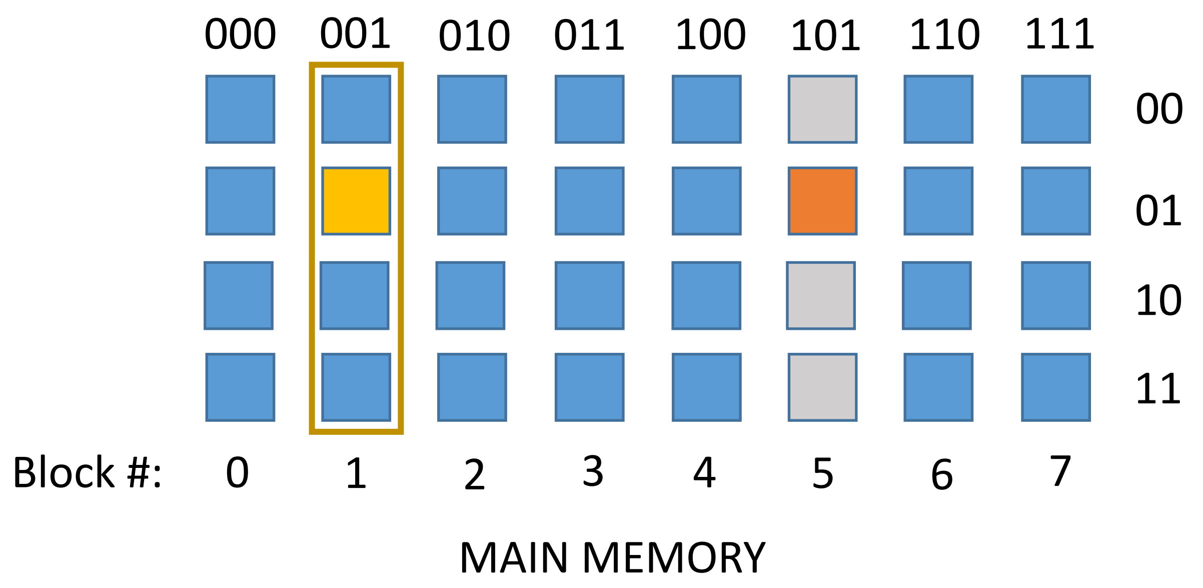




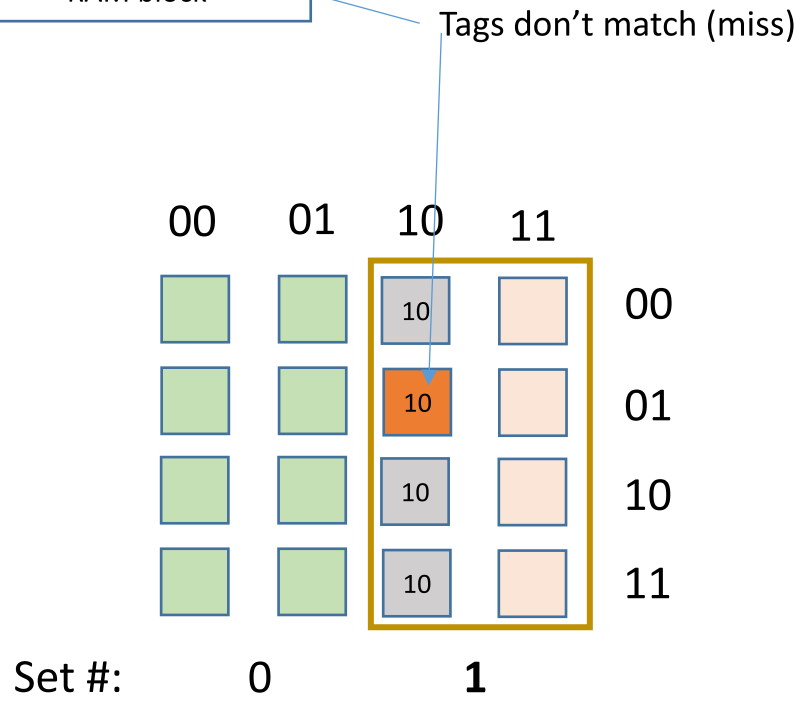
Cache Memory – Associative Set Mapping

- **00101**

tag	Cache set	Cache byte
00	1	01
RAM block		



MAIN MEMORY

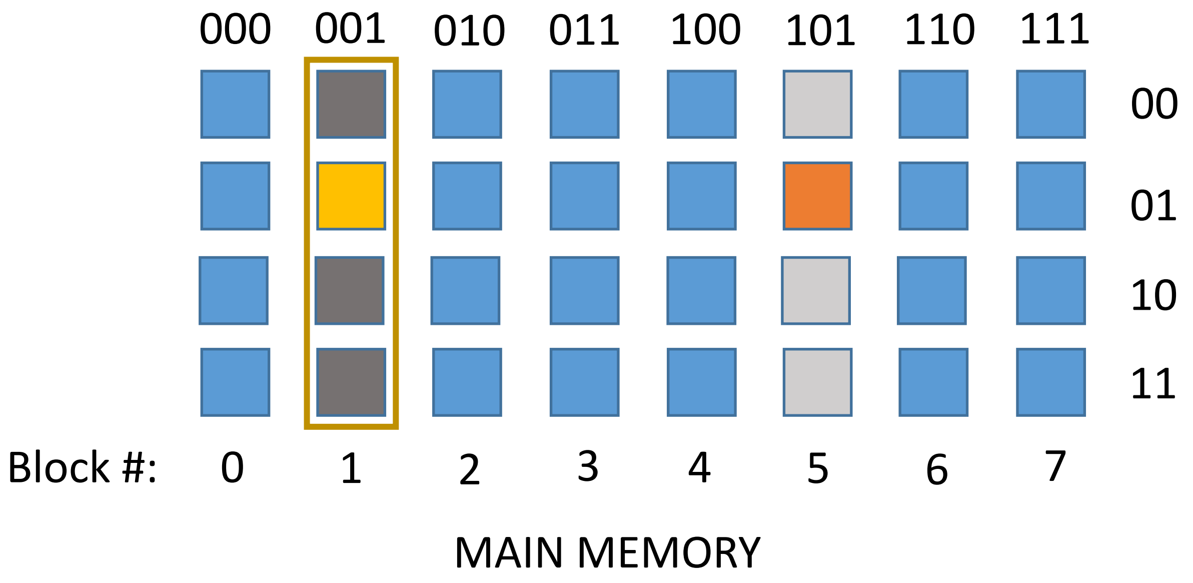


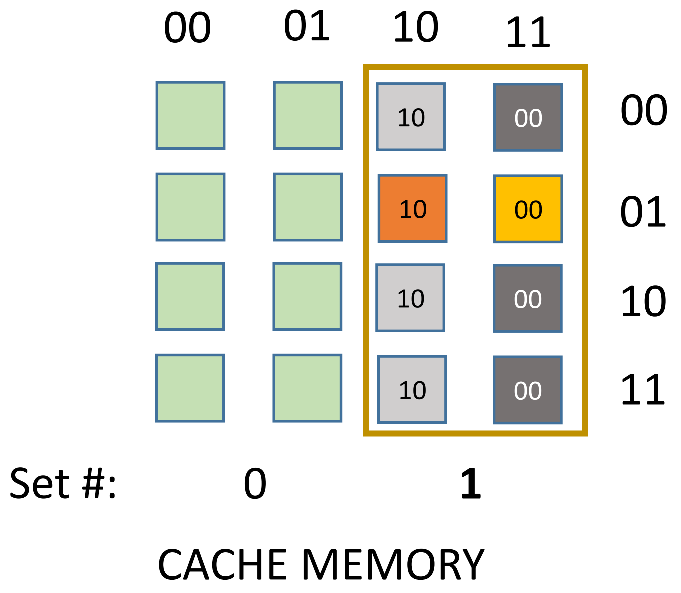
CACHE MEMORY

Cache Memory – Associative Set Mapping

- **00101**

tag	Cache set	Cache byte
00	1	01
RAM block		

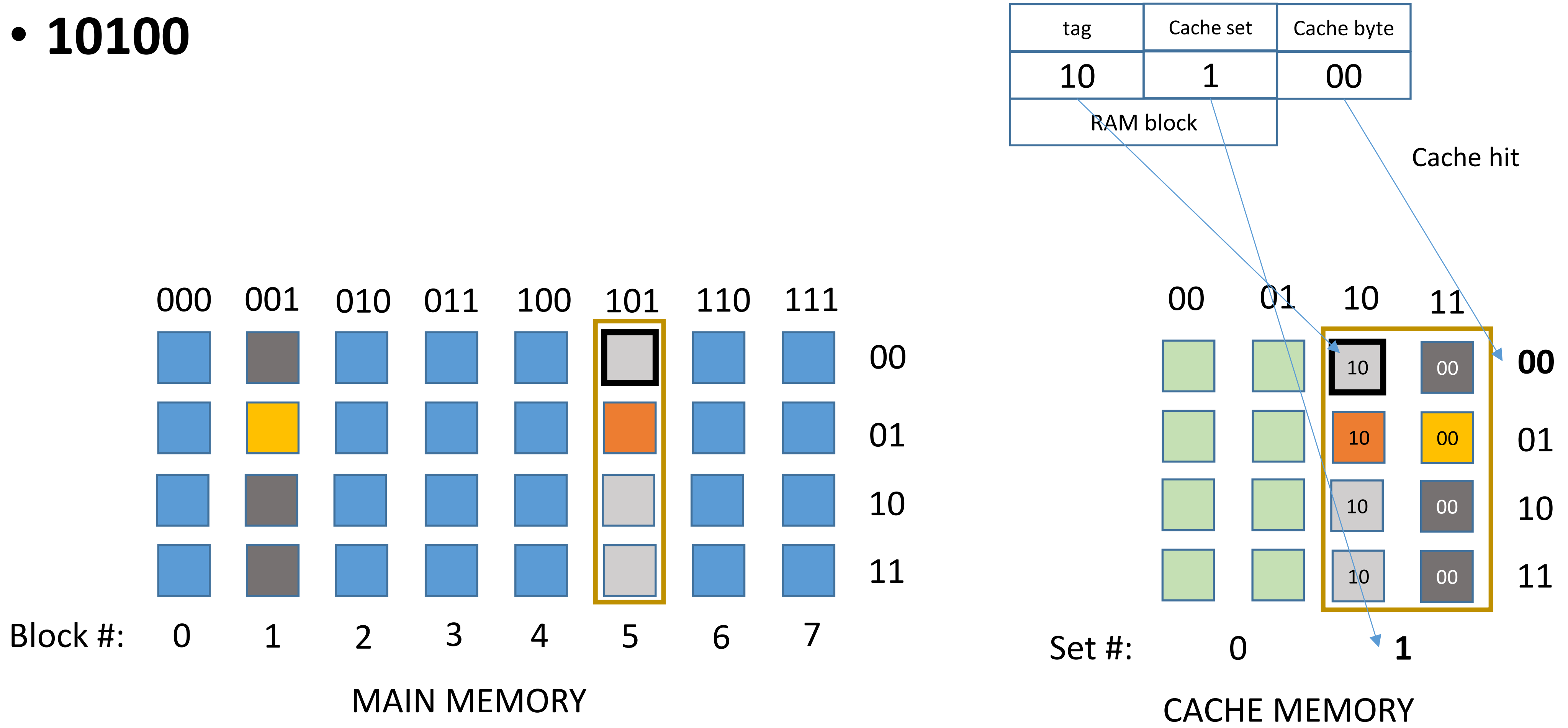




Cache Memory – Associative Set Mapping

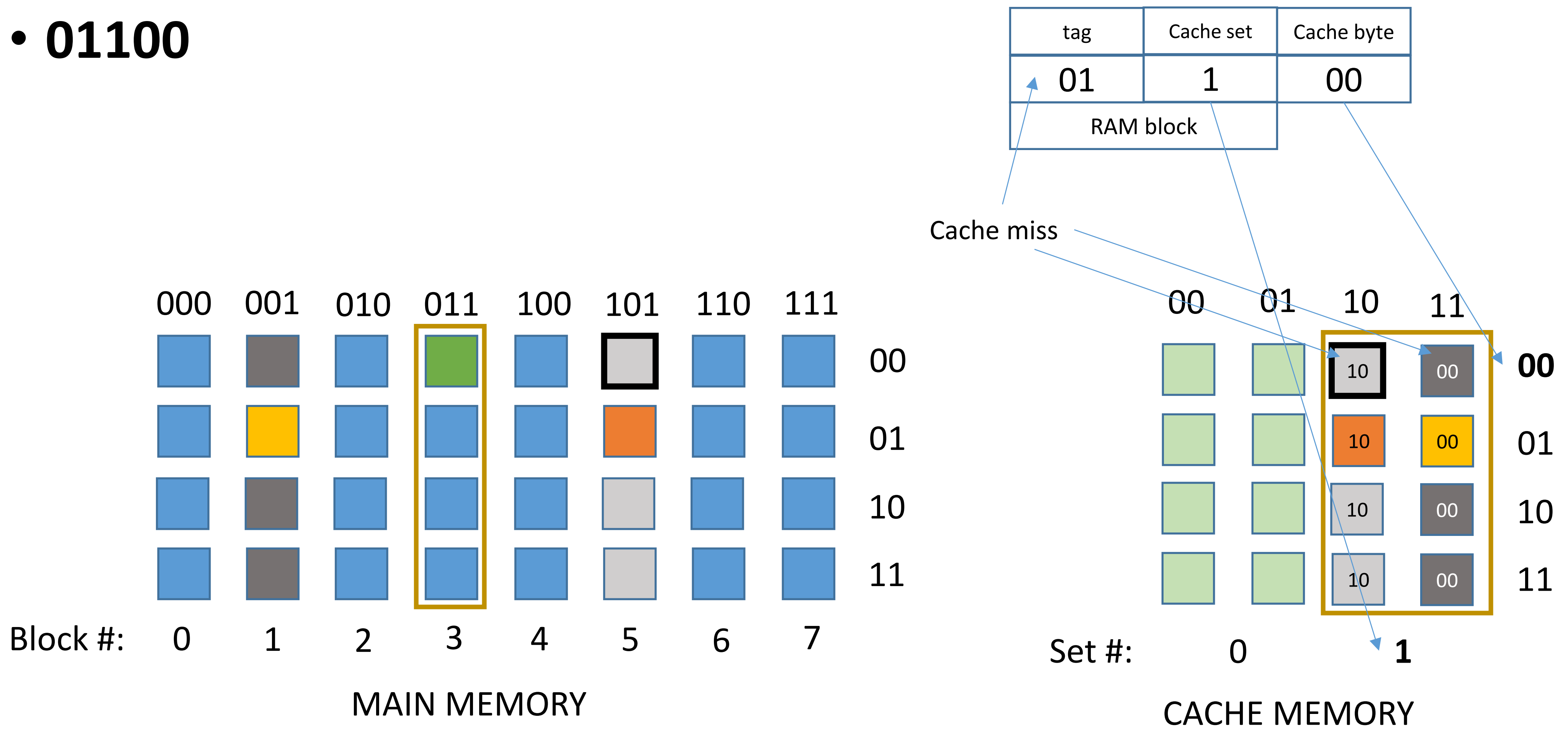
- **10100**

tag	Cache set	Cache byte
10	1	00



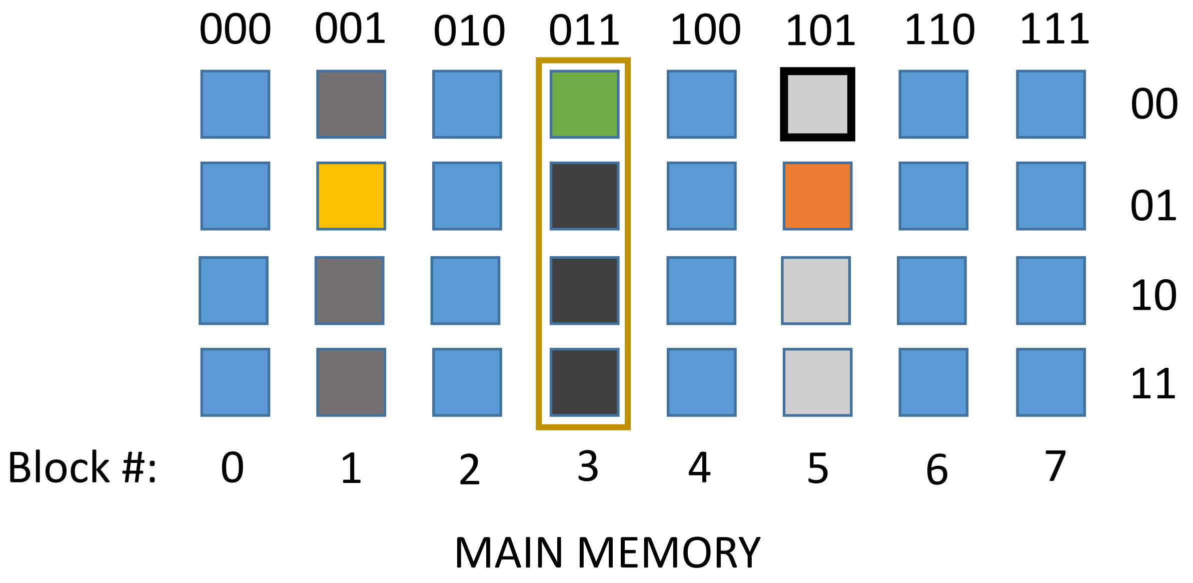
Cache Memory – Associative Set Mapping

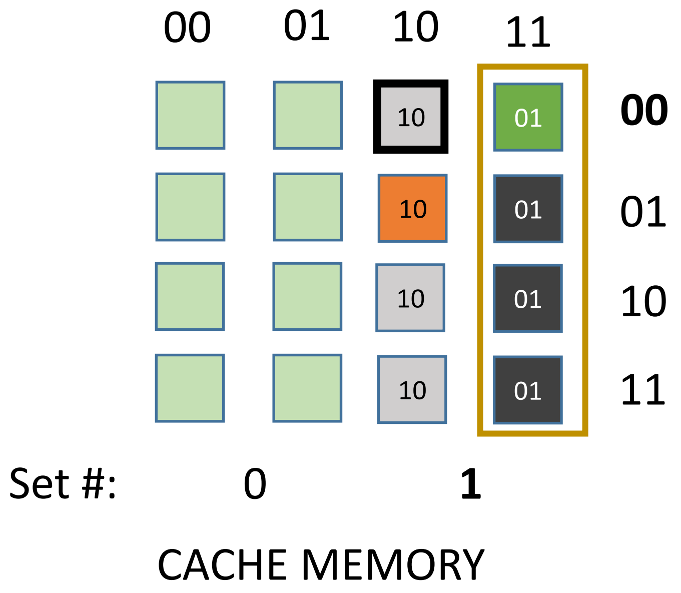
- **01100**



Cache Memory – Associative Set Mapping

- Since we are using LRU, we will replace block 3 (since we just used block 2)
    - Had we been using FIFO, then block 2 would have been replaced because that has been in the cache longer than the data in block 3





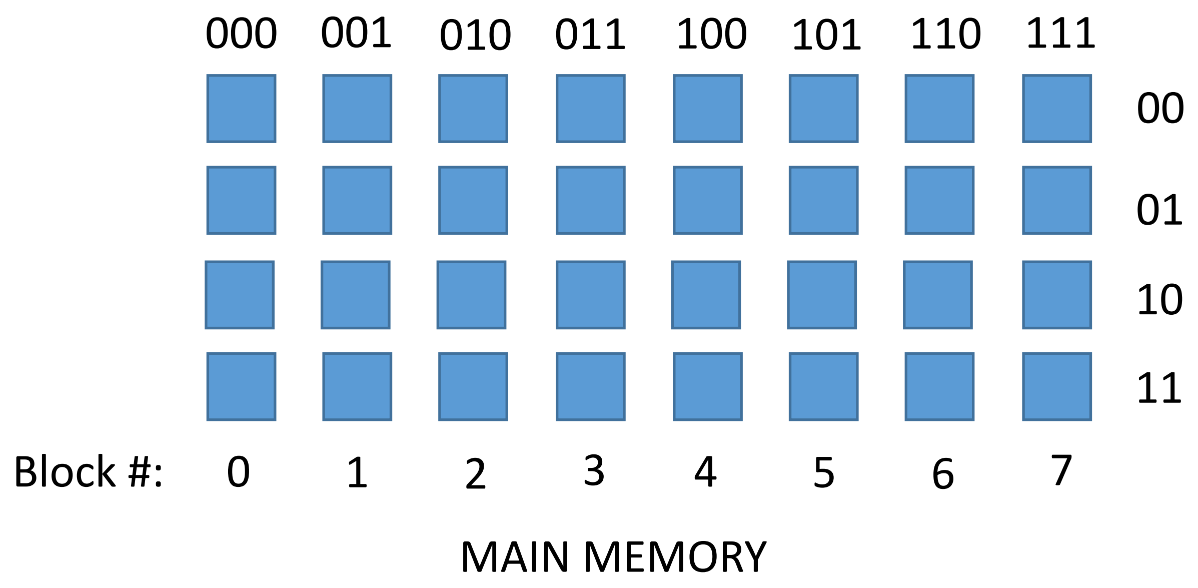
Cache Memory – Associative Set Mapping

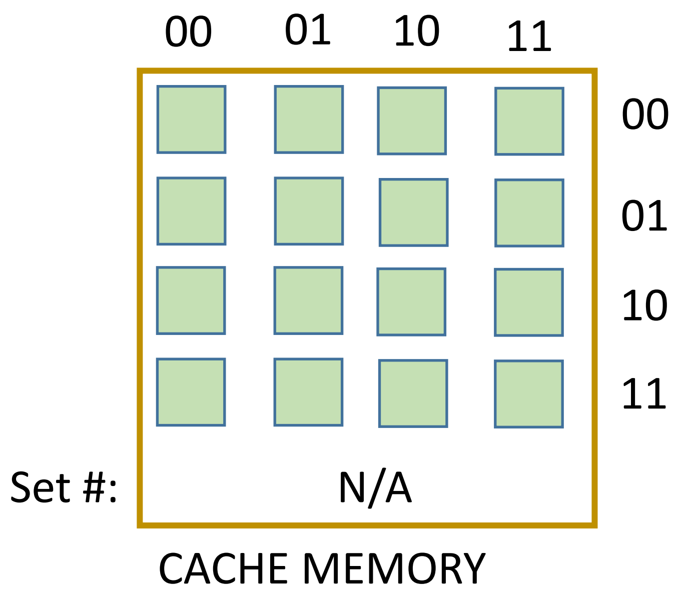
- Each row in the table below represents the read/write operations illustrated on the previous (associative mapping) slides

Memory Address	Hit or Miss	Cache Contents			
		Set 0 (Block 0)	Set 0 (Block 1)	Set 1 (Block 2)	Set 1 (Block 3)
10101	M			10 10 10 10	
00101	M			10 10 10 10	00 00 00 00
10100	H			10 10 10 10	00 00 00 00
01100	M			10 10 10 10	01 01 01 01

Cache Memory – Fully Associative Set Mapping

- A **fully associative mapped** cache consists of one set that contains every block of cache memory.

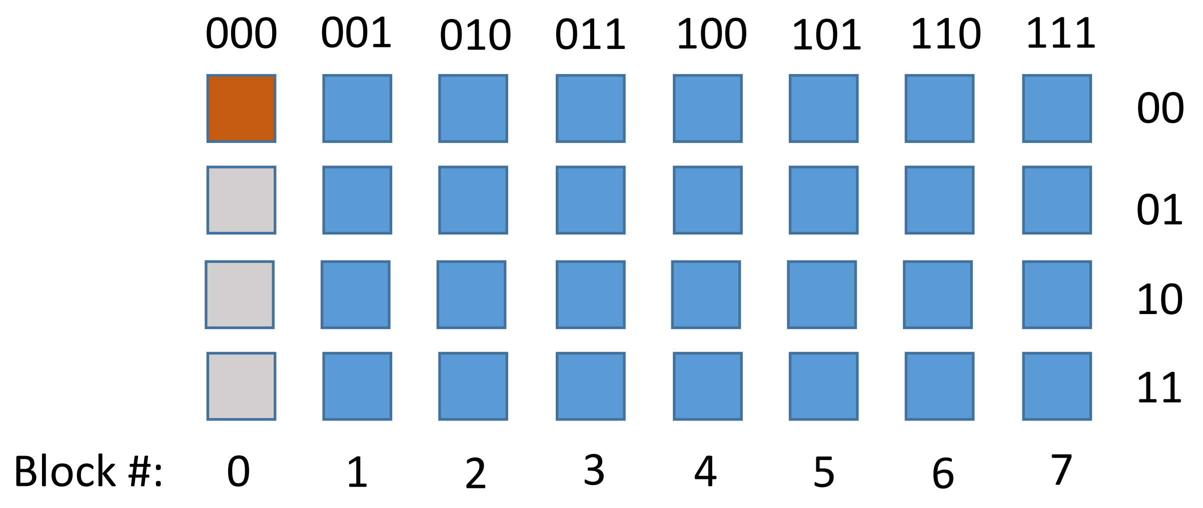




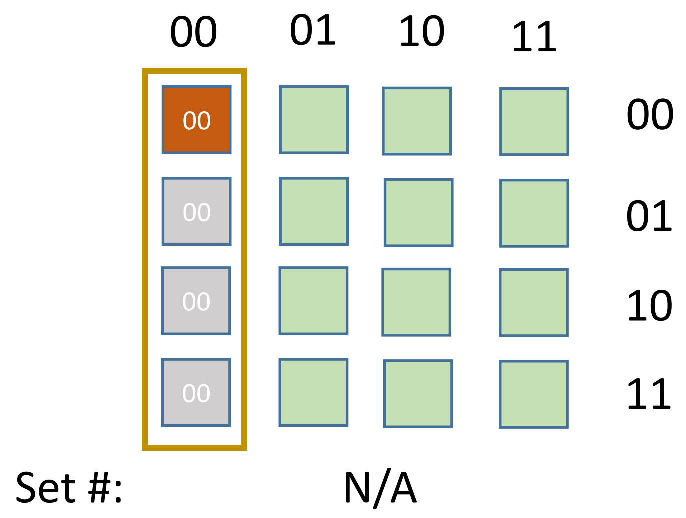
Cache Memory – Fully Associative Set Mapping

- **00000**

tag	Cache set	Cache byte
00	~~0~~	00
RAM block		



MAIN MEMORY

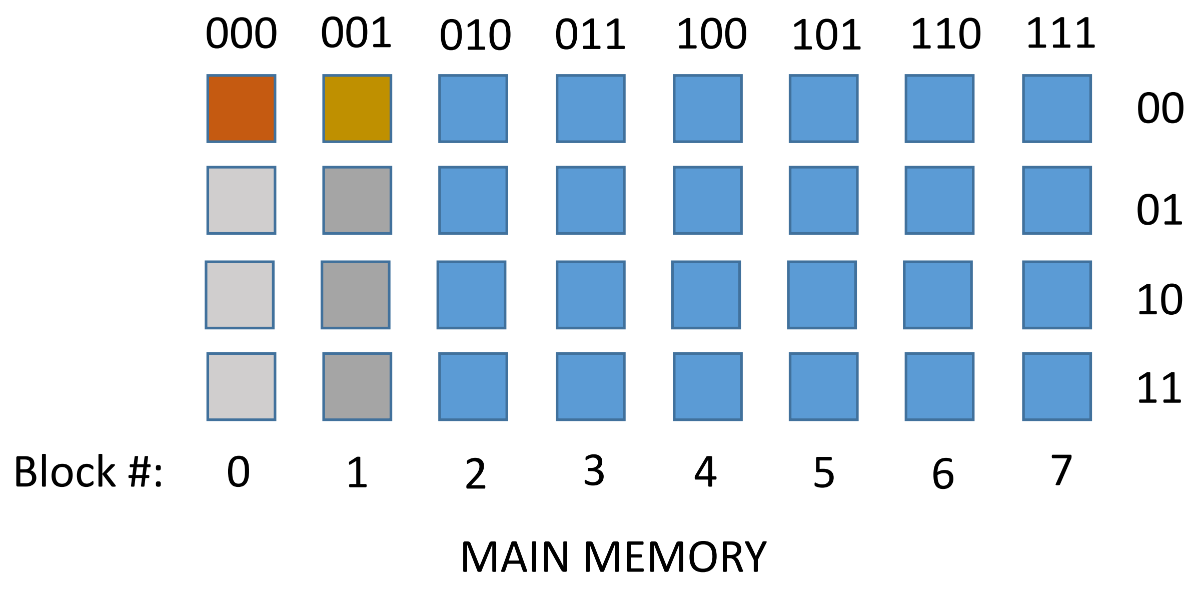


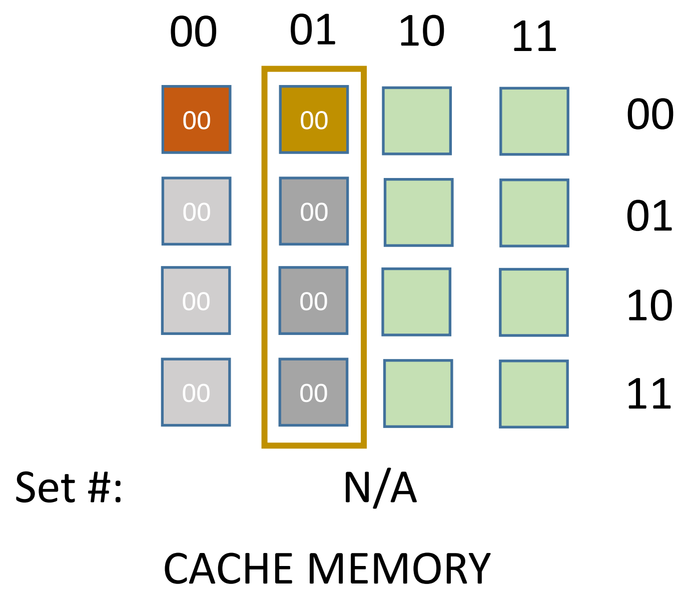
CACHE MEMORY

Cache Memory – Fully Associative Set Mapping

- **00100**

tag	Cache set	Cache byte
00	~~1~~	00
RAM block		

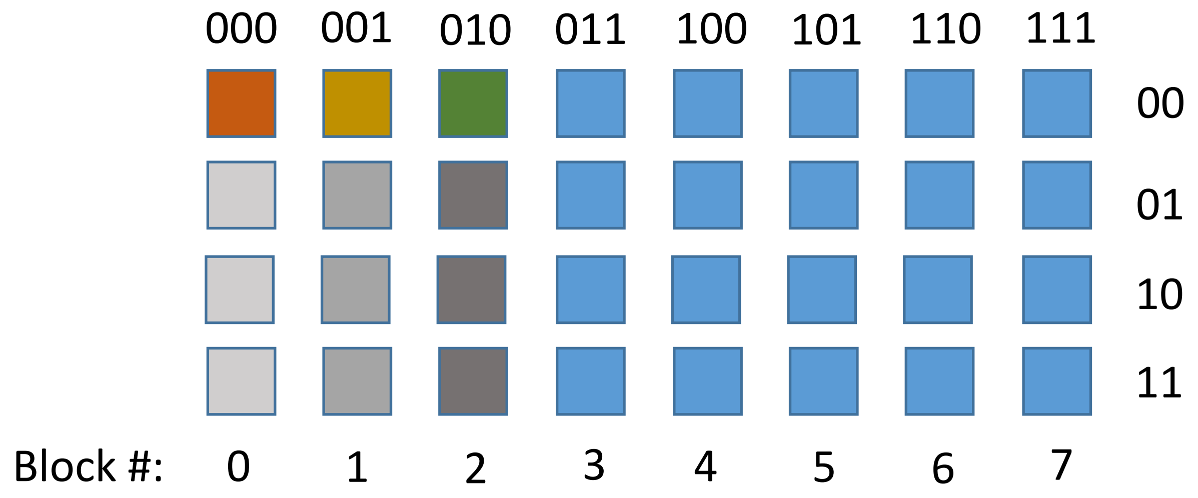




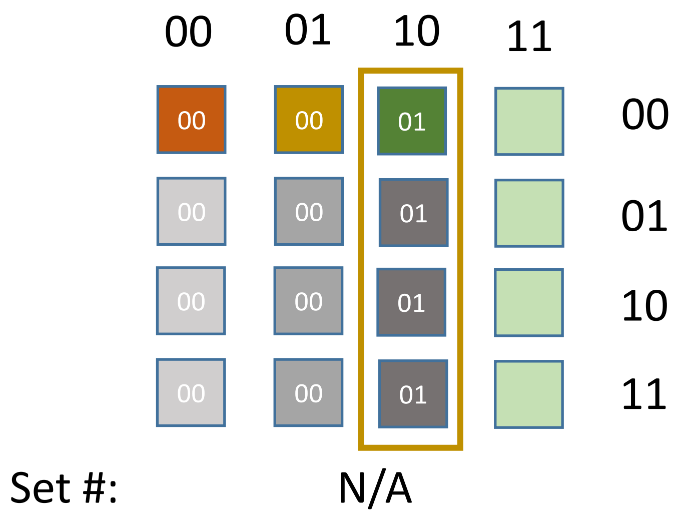
Cache Memory – Fully Associative Set Mapping

- **01000**

tag	Cache set	Cache byte
01	~~0~~	00
RAM block		



MAIN MEMORY

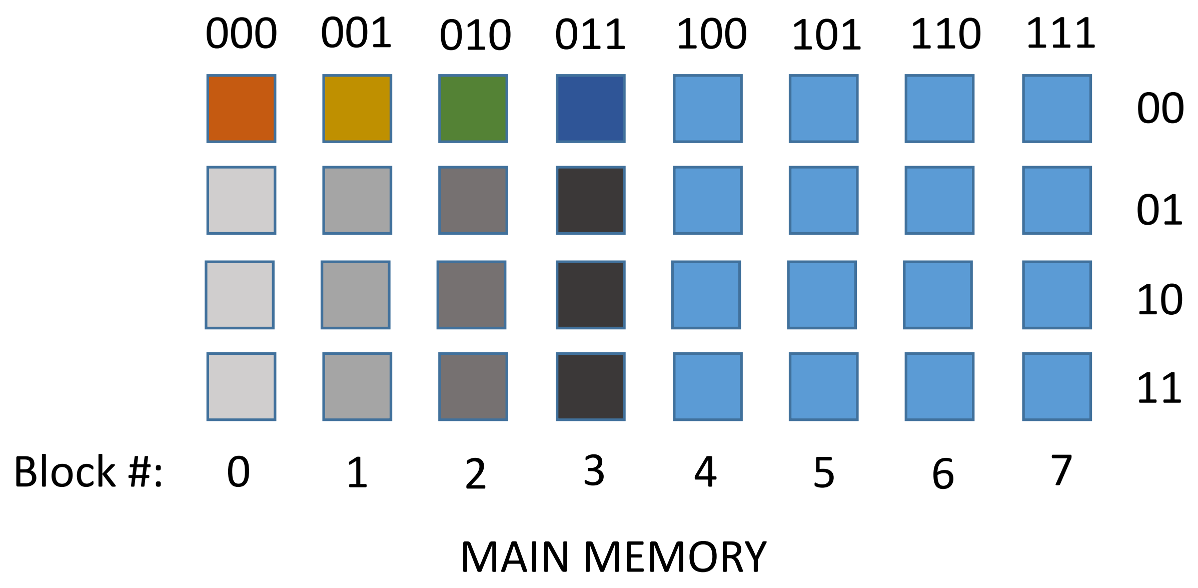


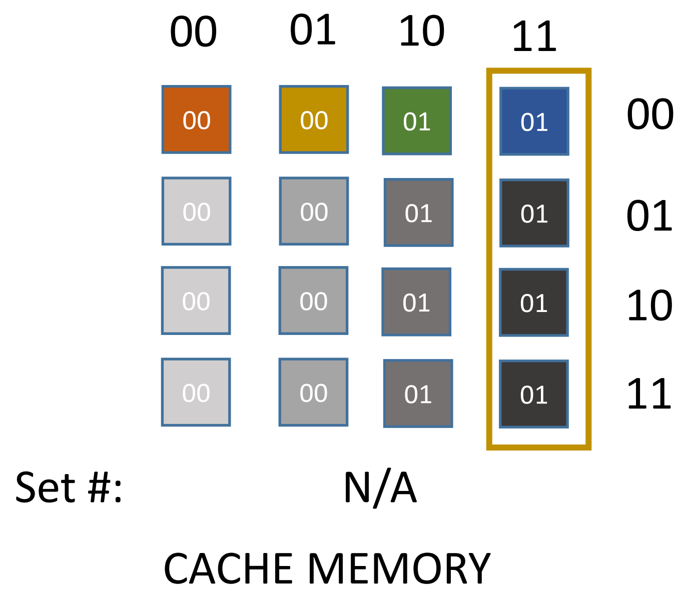
CACHE MEMORY

Cache Memory – Fully Associative Set Mapping

- **01100**

tag	Cache set	Cache byte
01	~~1~~	00
RAM block		

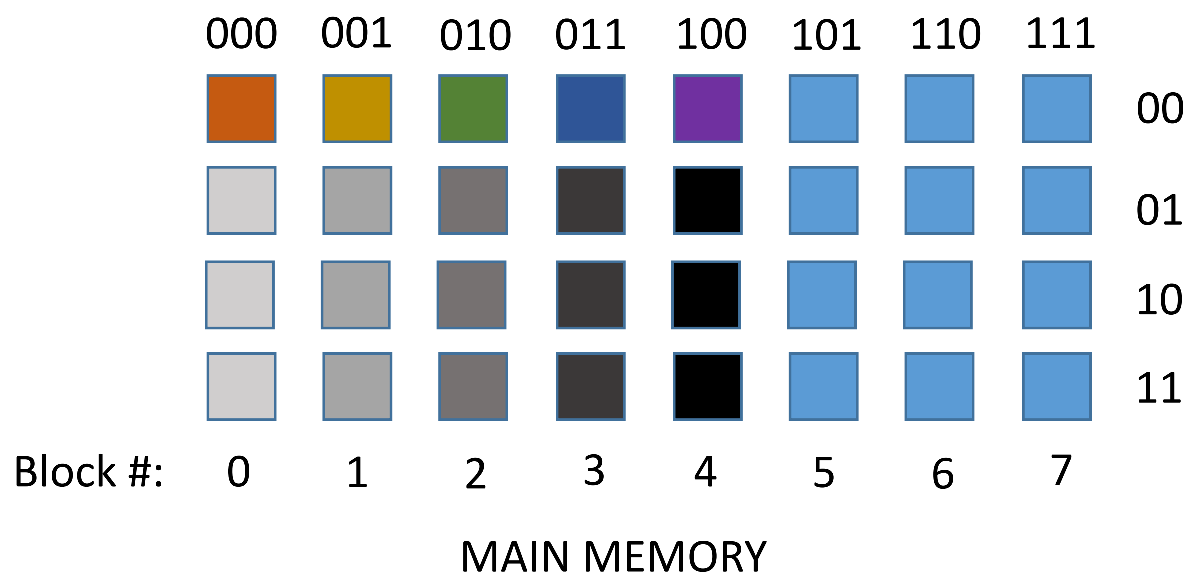


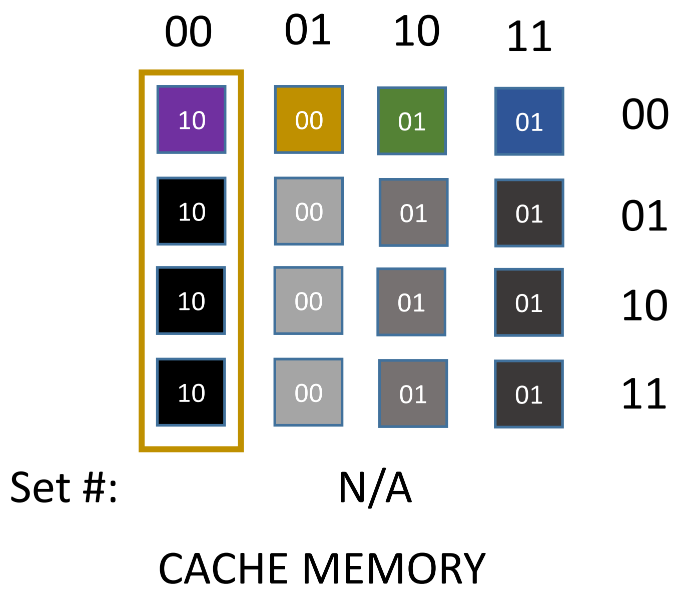


Cache Memory – Fully Associative Set Mapping

- **10000**
  - Out of space
  - The block to replace (LRU or FIFO in this case) is block 0

tag	Cache set	Cache byte
10	~~0~~	00
RAM block		





Cache Memory – Associative Set Mapping

- Each row in the table below represents the read/write operations illustrated on the previous (associative mapping) slides

Memory Address	Hit or Miss	Cache Contents			
		Block 0	Block 1	Block 2	Block 3
00000	M	00 00 00 00			
00100	M	00 00 00 00	00 00 00 00		
01000	M	00 00 00 00	00 00 00 00	01 01 01 01	
01100	M	00 00 00 00	00 00 00 00	01 01 01 01	01 01 01 01
10000	M	10 10 10 10	00 00 00 00	01 01 01 01	01 01 01 01

Dependability

- A device (such as a hard drive) alternates between two states of service
- **Service Accomplishment**
  - The device is working (delivering service) as designed
- **Service Interruption**
  - The device is not working (not delivering service) as designed

- A **failure** causes the device to go from Service Accomplishment to Service Interruption
- A **restoration** causes the device to go from Service Interruption to Service Accomplishment

Dependability

- The measure of continuous, uninterrupted Service Accomplishment until the device experiences a failure its **reliability**
  - Two such measures of reliability are:
  - **Mean Time to Failure (MTTF)**
    - The average time it takes until a device fails
  - **Annual Failure Rate (AFR)**
    - The percentage of devices expected to fail in a year within a given MTTF

Dependability

- As an example, we'll say a server farm has 25,000 servers.
  - Each server has 3 hard disks (all the same model)
    - $3 \times 25000 = 75000\ disks$
  - The MTTF of these disks are said to be 1,000,000 hours

$$One\ year = 365\ days\ \times 24 \frac{hours}{day} = 8760\ hours$$

$$AFR = \frac{8760\ hours}{1000000\ hours} = .00876 = .876\%$$

- The AFR of each disk is 0.876%

$$75000\ disks\ \times .00876 = 657$$

- 657 disks are expected to fail annually
  - $\frac{657\ disks}{365\ days} = 1.8 \sim 2\ disks\ per\ day$

Dependability

- Service Interruption is measured by **Mean Time to Repair (MTTR)**
- The **Mean Time Between Failures (MTBF)** is the sum of the MTTF and MTTR
- Availability is the measure of Service Accomplishment (MTTF) with respect to the alternation between Service Accomplishment and Service Interruption (MTBF)

$$Availability = \frac{MTTF}{(MTTF + MTTR)}$$

Dependability

- Using the previous example, we'll say MTTR for each disk in the server farm is 2 hours.
  - Each server has 3 hard disks (all the same model)
    - $3 \times 25000 = 75000\ disks$
  - The MTTF of these disks are said to be 1,000,000 hours
  - 657 disks are expected to fail annually (calculated on a previous slide)

$$Availability = \frac{1000000\ hours}{(1000000\ hours + (2\ hours\ \times 657\ disks))} \sim .9986 \sim 99.86\%$$

Dependability

- MTTF can be increased through improving the quality of components or designing them to continue operating in the event of failures
  - **Fault avoidance**
    - Preventing faults in the design of the device/components
  - **Fault tolerance**
    - Using redundancy to continue functioning in the event of faults
  - **Fault forecasting**
    - Predicting faults so that devices/components may be replaced before a fault occurs

Parity Bits

- **Parity bits** are used as a way of validating that binary data makes it from source to destination intact.
  - A type of *error detection code*

- Before a binary word is sent through the system's data path (we'll use 8-bit words for this example), the total number of 1's in the word are counted.
  - If there are an even number of 1's, a 0 is appended to the number
  - If there are an odd number of 1's, a 1 is appended to the number

  - Either way, parity bits ensure an even number of 1's in the *n*+1 bit word.

  - If the received binary word (plus the parity bit) contain an odd number of 1's, it indicates there was a problem with the integrity of the data, and the word should be re-sent.

Parity Bits

- 01101100
  - Even number of 1’s
  - **0**01101100

Parity bit

- 01001100
  - Odd number of 1’s
  - **1**01001100

Parity bit

Bit gets corrupted in transit

0011**1**1100

Destination

001111100

- Not an even number of 1’s
- Data must have been corrupted.

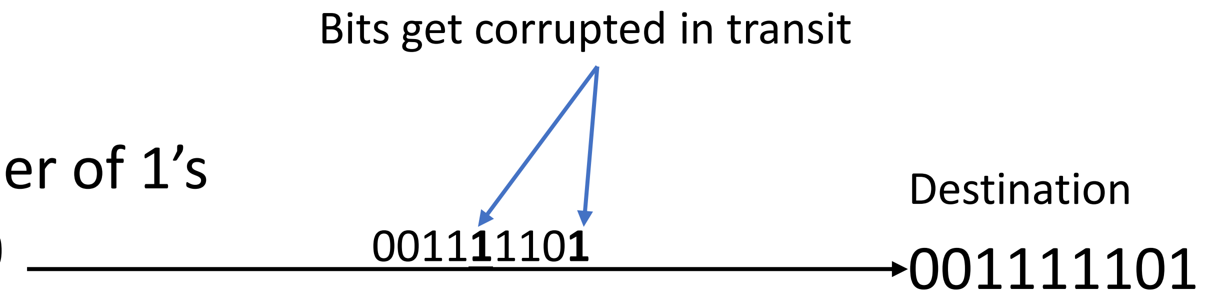
Parity Bits

- Parity bits alone are only able to *detect* errors.

- It is possible that an error will not be detected:

- 01101100
  - Even number of 1’s
  - **0**01101100

Parity bit



- Even number of 1’s
- No error detected

Error Correction Code

- **Error Correction Code** (ECC or Hamming Code, named after its creator Richard Hamming) uses multiple parity bits that can detect and *correct* errors in a binary word.

- A simple algorithm is followed to insert parity bits into the binary data.

Error Correction Code

1. Number the bits starting at 1 from the left side:

1	2	3	4	5	6	7	8
0	1	1	1	1	1	0	1

2. Mark the positions that are powers of 2 (1, 2, 4, 8, 16, and so on) as the parity bits

1	**2**	3	**4**	5	6	7	**8**
0	1	1	1	1	1	0	1

Error Correction Code

3. All bit positions that are not powers of two will be used for the actual data:

$$\begin{array}{cccccccc} \mathbf{1} & \mathbf{2} & 3 & \mathbf{4} & 5 & 6 & 7 & \mathbf{8} \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{array} \longrightarrow \begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ \_ & \_ & 0 & \_ & 1 & 1 & 1 & \_ & 1 & 1 & 0 & 1 \end{array}$$

Error Correction Code

4. The position of a parity bit determines which bits it checks:
   - Bit 1 checks positions 1, 3, 5, 7, 9, and so on
   - Bit 2 checks positions 2, 3, 6, 7, 10, 11, 14, 15, and so on
   - Bit 4 checks positions 4-7, 12-15, 20-23, and so on
   - Bit 8 checks positions 8-15, 24-31, 40-47, and so on

   - Each bit of the actual data is checked by two or more parity bits

Error Correction Code

- Bit 1 checks positions 1, 3, 5, 7, 9, and so on:

1	2	3	4	5	6	7	8	9	10	11	12
_	_	**0**	_	**1**	1	**1**	_	**1**	1	**0**	1

- Odd number of 1's

1	2	3	4	5	6	7	8	9	10	11	12
1	_	0	_	1	1	1	_	1	1	0	1

Error Correction Code

- Bit 2 checks positions 2, 3, 6, 7, 10, 11, 14, 15, and so on :

1	2	3	4	5	6	7	8	9	10	11	12
1	_	**0**	_	1	**1**	**1**	_	1	**1**	**0**	1

- Odd number of 1’s

1	2	3	4	5	6	7	8	9	10	11	12
1	**1**	0	_	1	1	1	_	1	1	0	1

Error Correction Code

- Bit 4 checks positions 4-7, 12-15, 20-23, and so on :

$$\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
1 & 1 & 0 & \_ & \mathbf{1} & \mathbf{1} & \mathbf{1} & \_ & 1 & 1 & 0 & \mathbf{1}
\end{array}$$

- Even number of 1’s

$$\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
1 & 1 & 0 & \mathbf{0} & 1 & 1 & 1 & \_ & 1 & 1 & 0 & 1
\end{array}$$

Error Correction Code

- Bit 8 checks positions 8-15, 24-31, 40-47, and so on :

1	2	3	4	5	6	7	8	9	10	11	12
1	1	0	0	1	1	1	_	**1**	**1**	**0**	**1**

- Odd number of 1’s

1	2	3	4	5	6	7	8	9	10	11	12
1	1	0	0	1	1	1	**1**	1	1	0	1

Error Correction Code

- We'll say these bits are received with an error:

1 1 0 0 1 1 1 1 1 1 0 1

$\downarrow$

1 1 0 0 $\underline{0}$ 1 1 1 1 1 0 1

Error Correction Code

- Bit 1 gets checked (positions 1, 3, 5, 7, 9, and so on):
  - Odd parity; **Error detected**

$$\mathbf{1}\ 1\ \mathbf{0}\ 0\ \underline{\mathbf{0}}\ 1\ \mathbf{1}\ 1\ \mathbf{1}\ 1\ \mathbf{0}\ 1$$

- Bit 2 gets checked (positions 2, 3, 6, 7, 10, 11, 14, 15, and so on):
  - Even parity; **No error detected**

$$1\ \mathbf{1}\ \mathbf{0}\ 0\ \underline{0}\ \mathbf{1}\ \mathbf{1}\ 1\ 1\ \mathbf{1}\ \mathbf{0}\ 1$$

Error Correction Code

- Bit 4 gets checked (positions 4-7, 12-15, 20-23, and so on):
  - Odd parity; **Error detected**

$$1\ 1\ 0\ \mathbf{0}\ \underline{\mathbf{0}}\ \mathbf{1}\ \mathbf{1}\ 1\ 1\ 1\ 0\ \mathbf{1}$$

- Bit 8 gets checked (positions 8-15, 24-31, 40-47, and so on):
  - Even parity; **No error detected**

$$1\ 1\ 0\ 0\ \underline{0}\ 1\ 1\ \mathbf{1}\ \mathbf{1}\ \mathbf{1}\ \mathbf{0}\ \mathbf{1}$$

Error Correction Code

- **Bit 1; Error detected** **1** 1 **0** 0 $\underline{\mathbf{0}}$ 1 **1** 1 **1** 1 **1** 1
- Bit 2; No error detected 1 **1** **0** 0 $\underline{0}$ **1** **1** 1 1 **1** **1** 1
- **Bit 4; Error detected** 1 1 0 **0** $\underline{\mathbf{0}}$ **1** **1** 1 1 1 1 **1**
- Bit 8; No error detected 1 1 0 0 $\underline{0}$ 1 1 **1** **1** **1** **1** **1**

- **Bit 1 + Bit 4 = Bit 5**
  - **Bit 5 is wrong**

1 1 0 0 $\underline{\mathbf{1}}$ **1 1 1 1 1 1 1**