Revised: 2/25/2021

Memory Architecture I

Michael C. Hackett

Assistant Professor, Computer Science

Community College of Philadelphia

Lecture Topics

- Memory Hierarchy
- Memory Technologies
    - RAM
    - ROM
- Cache Memory
    - Direct Mapping
    - N-way Associative Set Mapping
    - Fully Associative Set Mapping

Memory Hierarchy

- The **Memory Hierarchy** is the structured levels of a computer system’s memory.
  - As the distance from the processor increases, the size and the access time increases

1. Registers (Fastest/Smallest)
2. Cache Memory
3. Main Memory
4. Secondary Storage (Slowest/Largest)

Memory Hierarchy

- In the first tier are *registers*, which is memory used for storing data currently in use by the CPU.

- Registers have the fastest access time, but they are limited in number.

Memory Hierarchy

- In the second tier is *cache memory*, which is memory used for storing data the CPU has recently used.

- The cache memory is built into the CPU and has a much larger capacity than the limited number of registers.
  - Cache memory is still limited in space, as only so much can fit in the CPU.

Memory Hierarchy

- In the third tier is *main memory*, which is memory used for storing data the CPU doesn’t immediately need.
- Main Memory is often referred to as the system’s *RAM*
  - In this course, RAM will indicate a specific type of memory technology.
- Much larger capacity than the CPU’s cache memory.

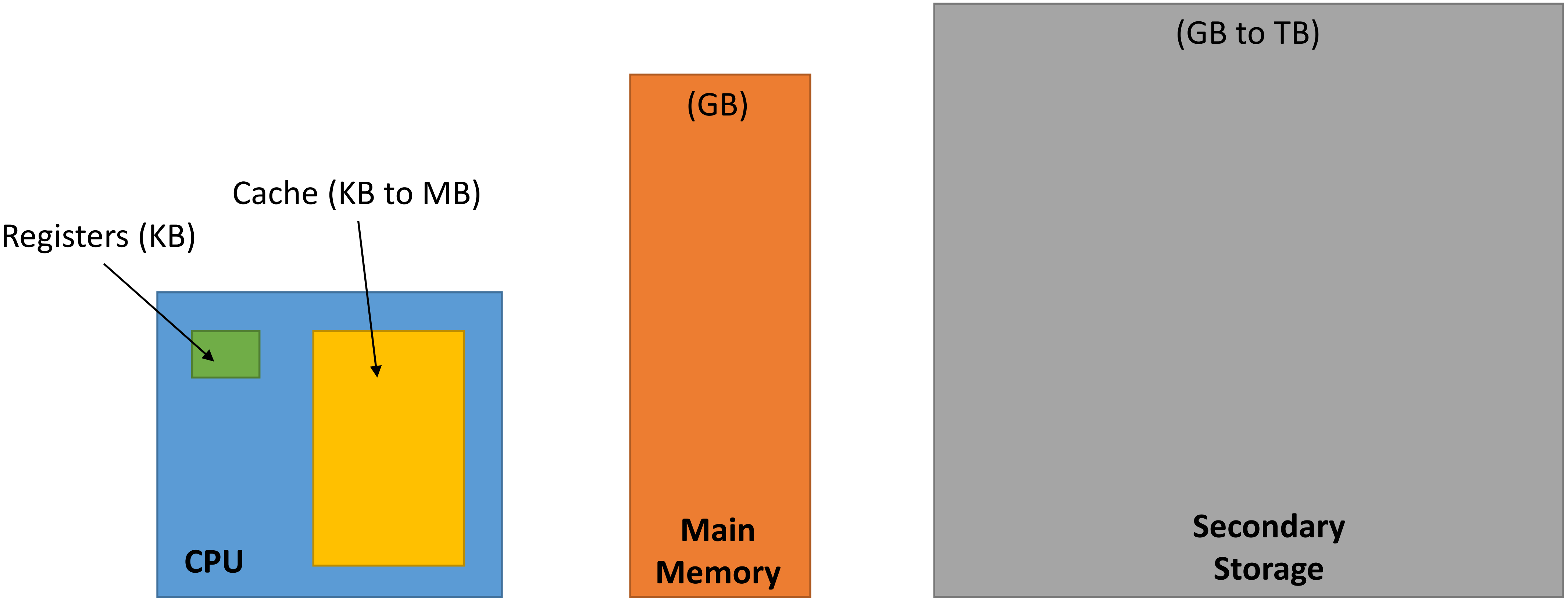
Memory Hierarchy

- In the fourth tier is *secondary storage*, which is memory used for storing data long term.
  - Magnetic Disks and Tape, Flash Memory, Optical Disks, etc.

- It is the memory with the slowest access time but has the greatest capacities.

- Secondary storage use **non-volatile** memory technologies.
  - The data stored in these technologies remains stored even when the system's power is turned off.
- Registers, cache and main memory use **volatile** memory technologies.
  - The data stored in these technologies are lost when the system's power is turned off.

Memory Hierarchy



Cache (KB to MB)

Registers (KB)

CPU

(GB)

Main Memory

(GB to TB)

Secondary Storage

Memory Technologies

- There are a variety of memory technologies used for secondary storage.
  - We'll discuss them in the next lecture

- This lecture focuses on the more fundamental types of memory technologies: RAM and ROM

Memory Technologies

- In older computer systems, reels of magnetic tape were used for main memory and secondary storage
- To access data, the reels were spun forward and reverse
  - *Sequential Access*
- And yes, tape is still a thing for secondary storage
  - Cost efficient means of backing up data

RAM

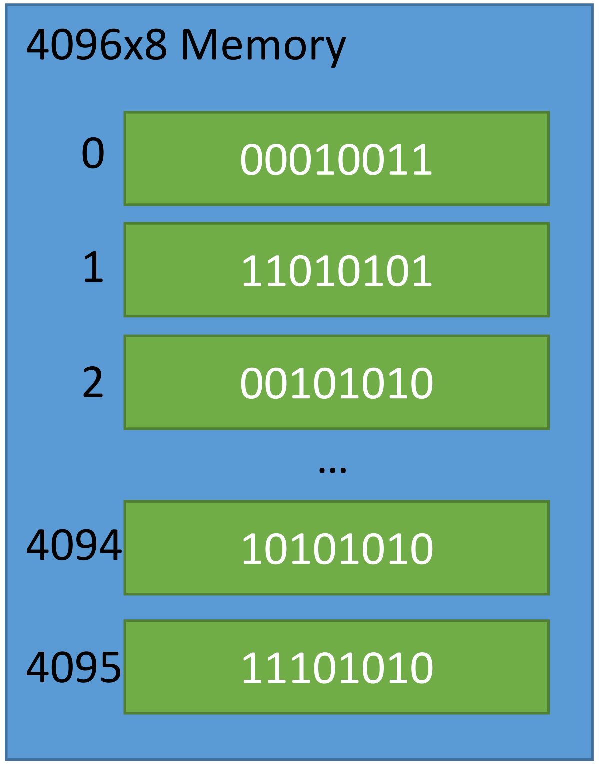
- **RAM** (**R**andom **A**ccess **M**emory) is a type of volatile memory that does not retain its stored bits when the system is powered off.

- Data stored in RAM can be accessed at random using an address

- Consists of N words of M bits each (**$\boldsymbol{N \times M}$ memory**)
  - Each word has a unique address

- For example, a 4096×8 memory:
  - 4096 8-bit words (32768 bits total)
  - 4096 unique addresses (0 through 4095)

RAM



4096x8 Memory

0	00010011
1	11010101
2	00101010
	…
4094	10101010
4095	11101010

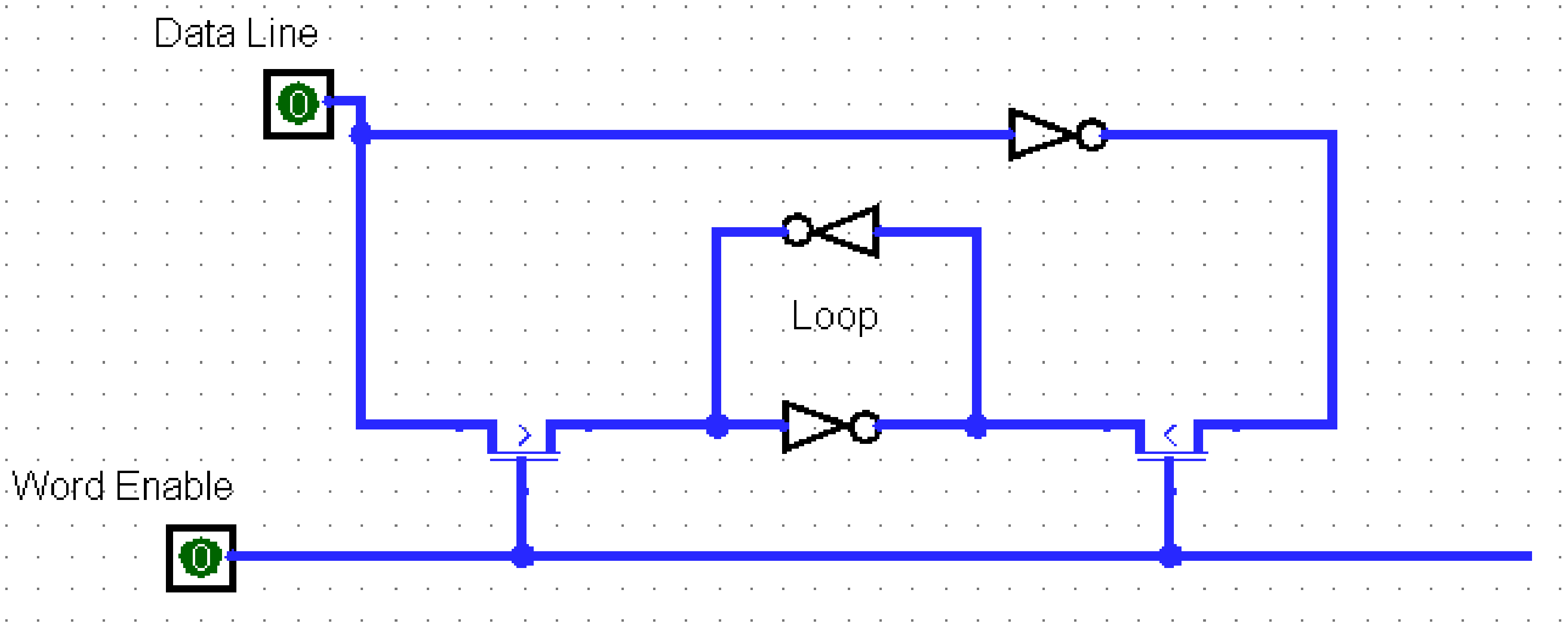
RAM

- **S**tatic **R**andom **A**ccess **M**emory (**SRAM**) and **D**ynamic **R**andom **A**ccess **M**emory (**DRAM**) are types of RAM built from an array of memory cells

- Each cell can store one bit of information
  - SRAM memory cell: Uses transistors and a loop of not gates
  - DRAM memory cell: Uses transistors and a capacitor

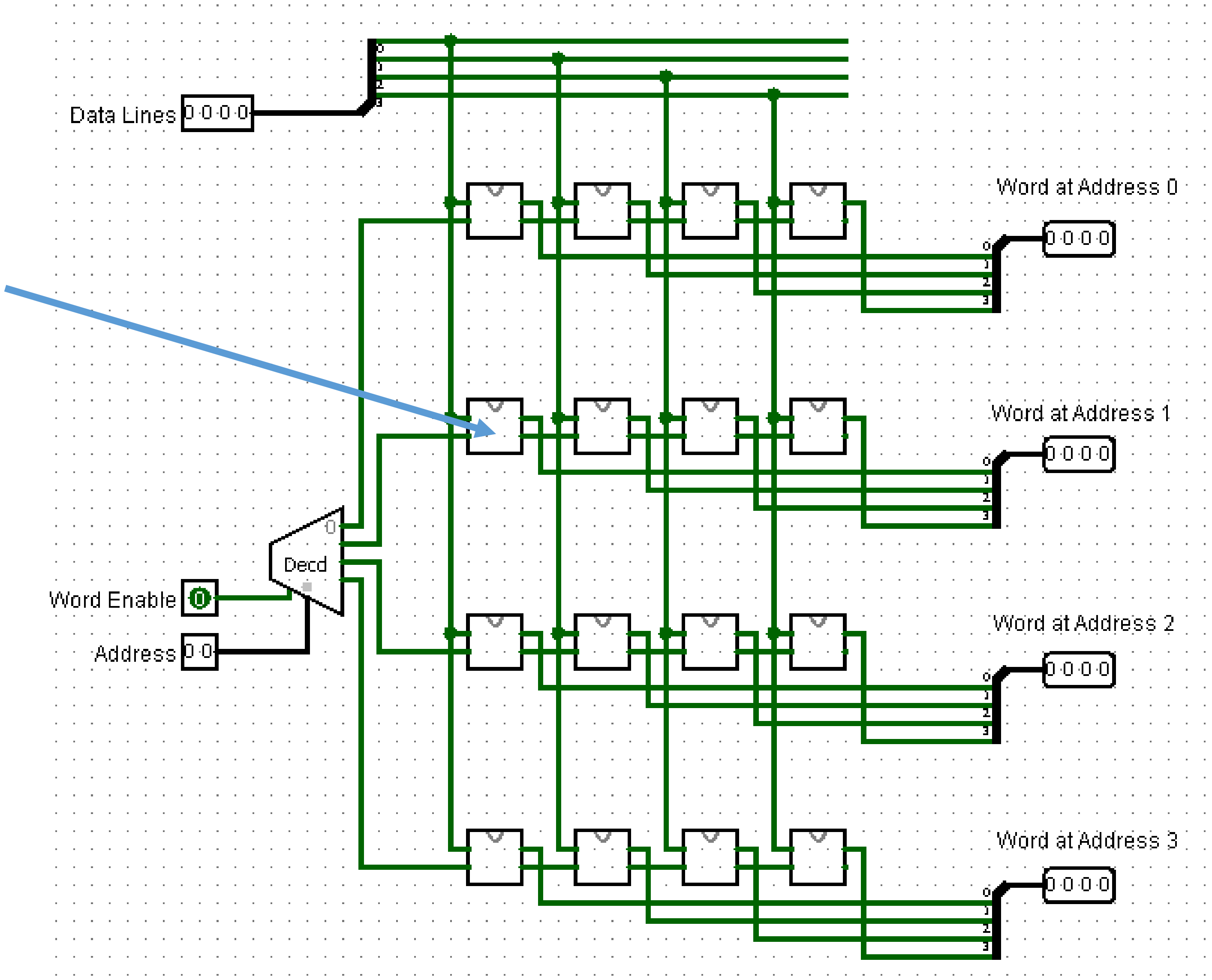
- Each cell is connected to a word enable line and data line
  - The **word enable line** allows the cell's value to be changed
  - The **data line** is the data to be stored in the cell

RAM – SRAM Cell



SRAM Cell

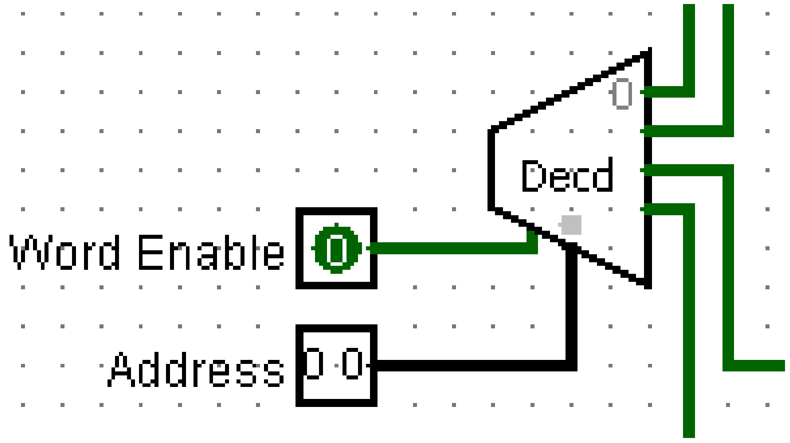
RAM



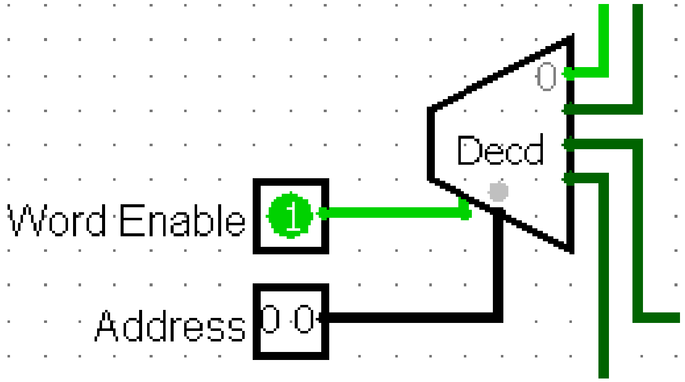
Memory Cells

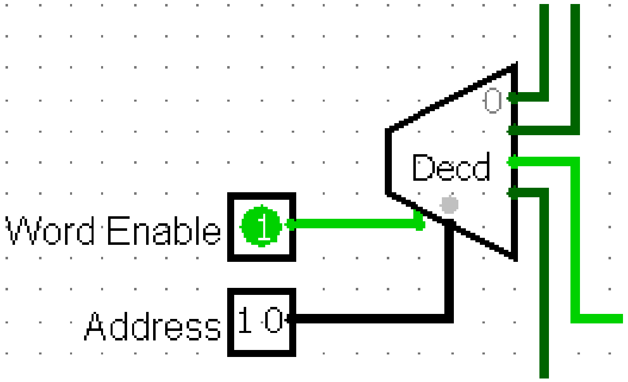
RAM

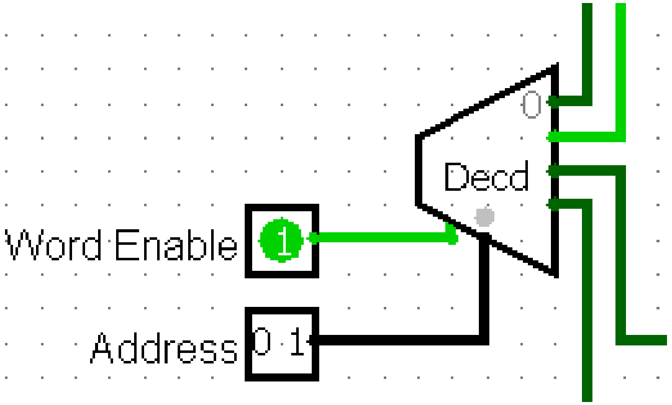
- To select a word, the word’s address is decoded to enable that row of cells
  - The “Word Enable” here is controlling if the decoder is enabled
    - Each output is a word enable to a row of cells
  - The “Address” is the input to the decoder

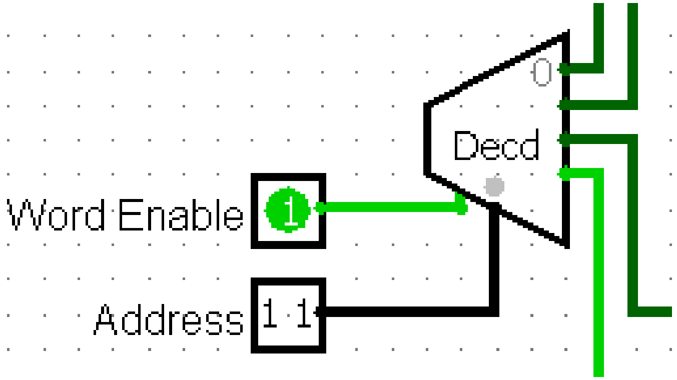


RAM

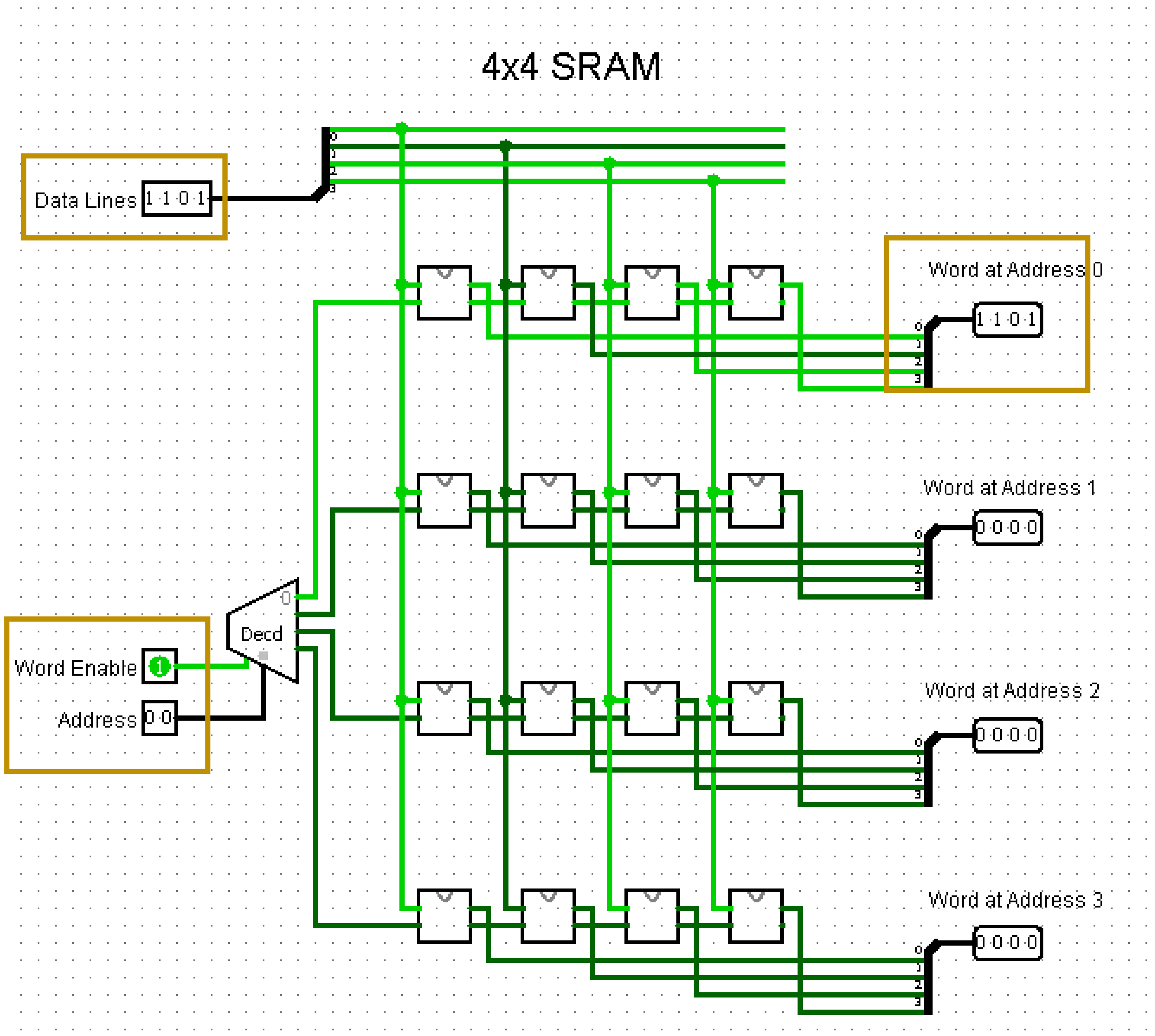

0
Decd
Word Enable
1
Address
0 0


0
Decd
Word Enable
1
Address
1 0

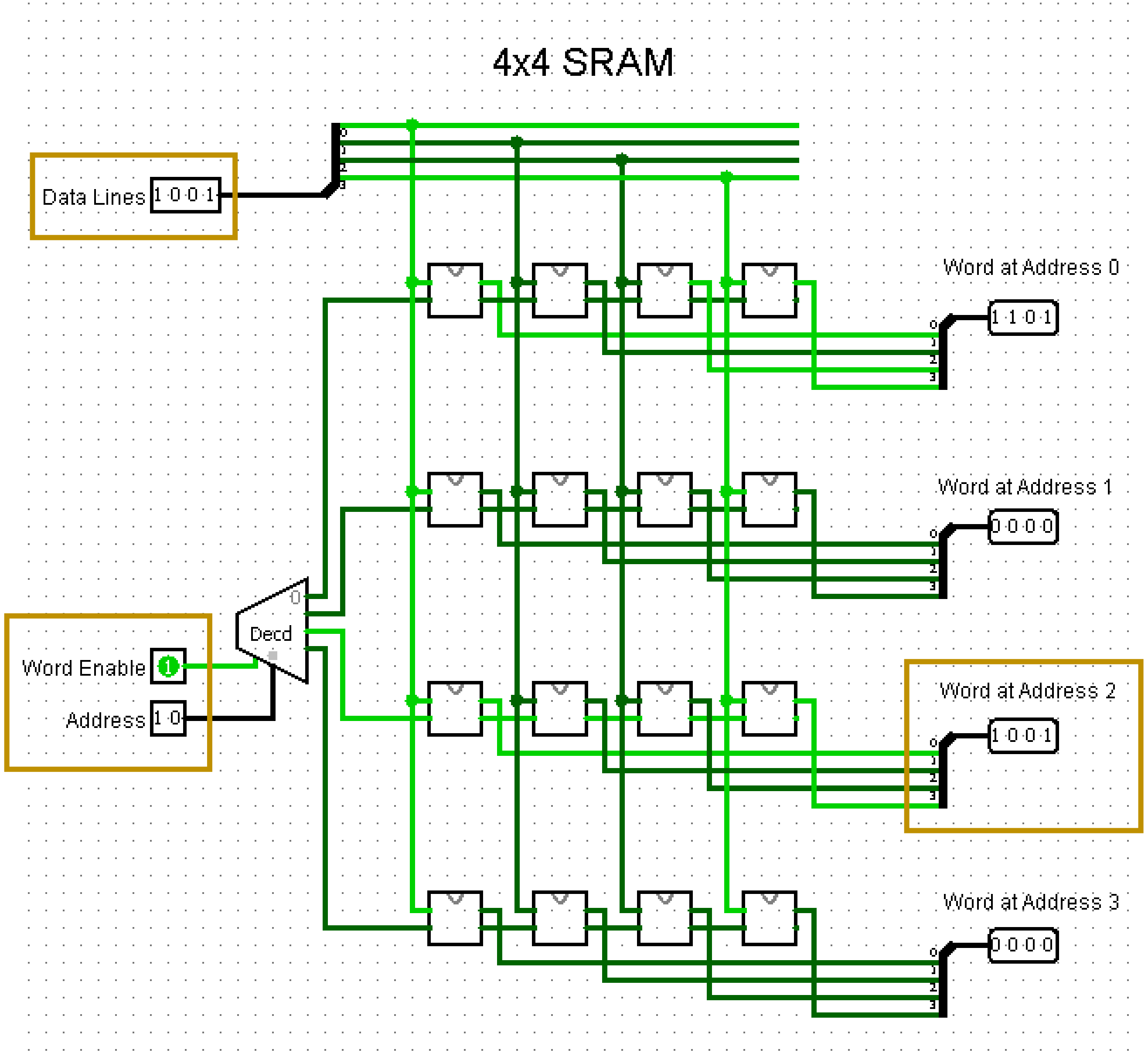

0
Decd
Word Enable
1
Address
0 1


0
Decd
Word Enable
1
Address
1 1

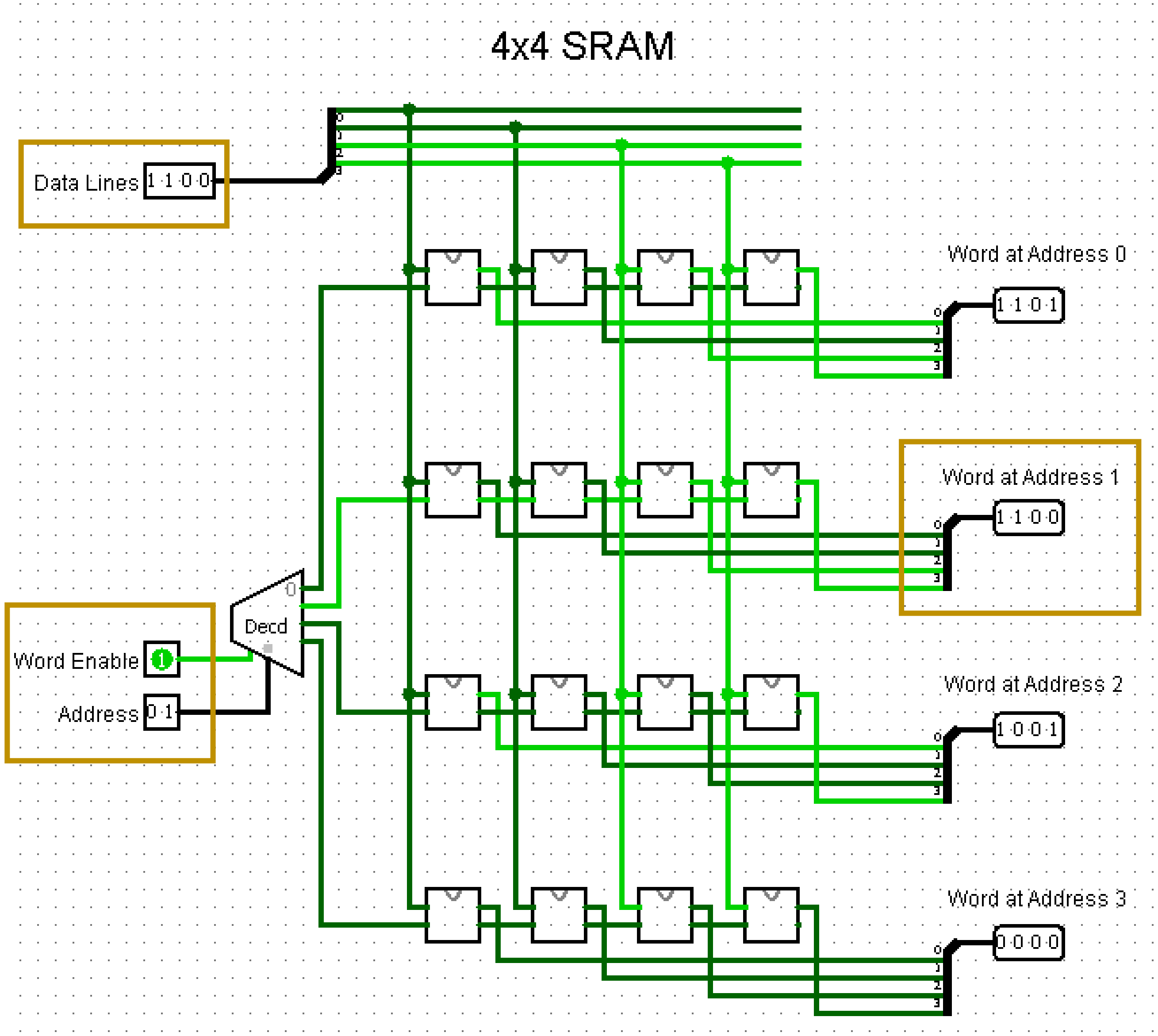
RAM


4x4 SRAM
Data Lines
1·1·0·1
Word at Address 0
1·1·0·1
Word at Address 1
0·0·0·0
Word at Address 2
0·0·0·0
Word at Address 3
0·0·0·0
Decd
Word Enable
1
Address
0·0

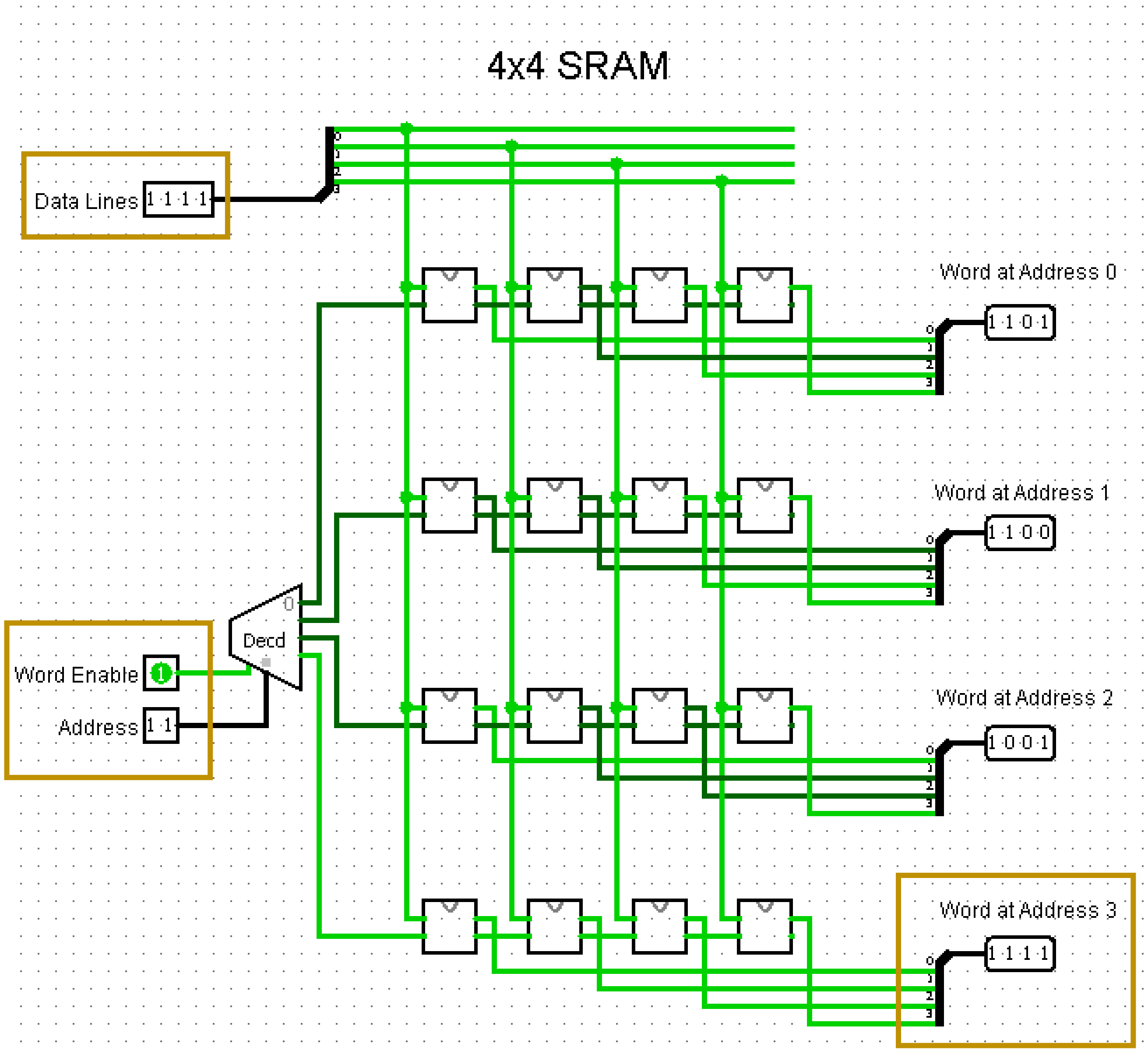
RAM


4x4 SRAM
Data Lines
1·0·0·1
Word Enable
1
Address
1·0
Decd
0
Word at Address 0
1·1·0·1
Word at Address 1
0·0·0·0
Word at Address 2
1·0·0·1
Word at Address 3
0·0·0·0

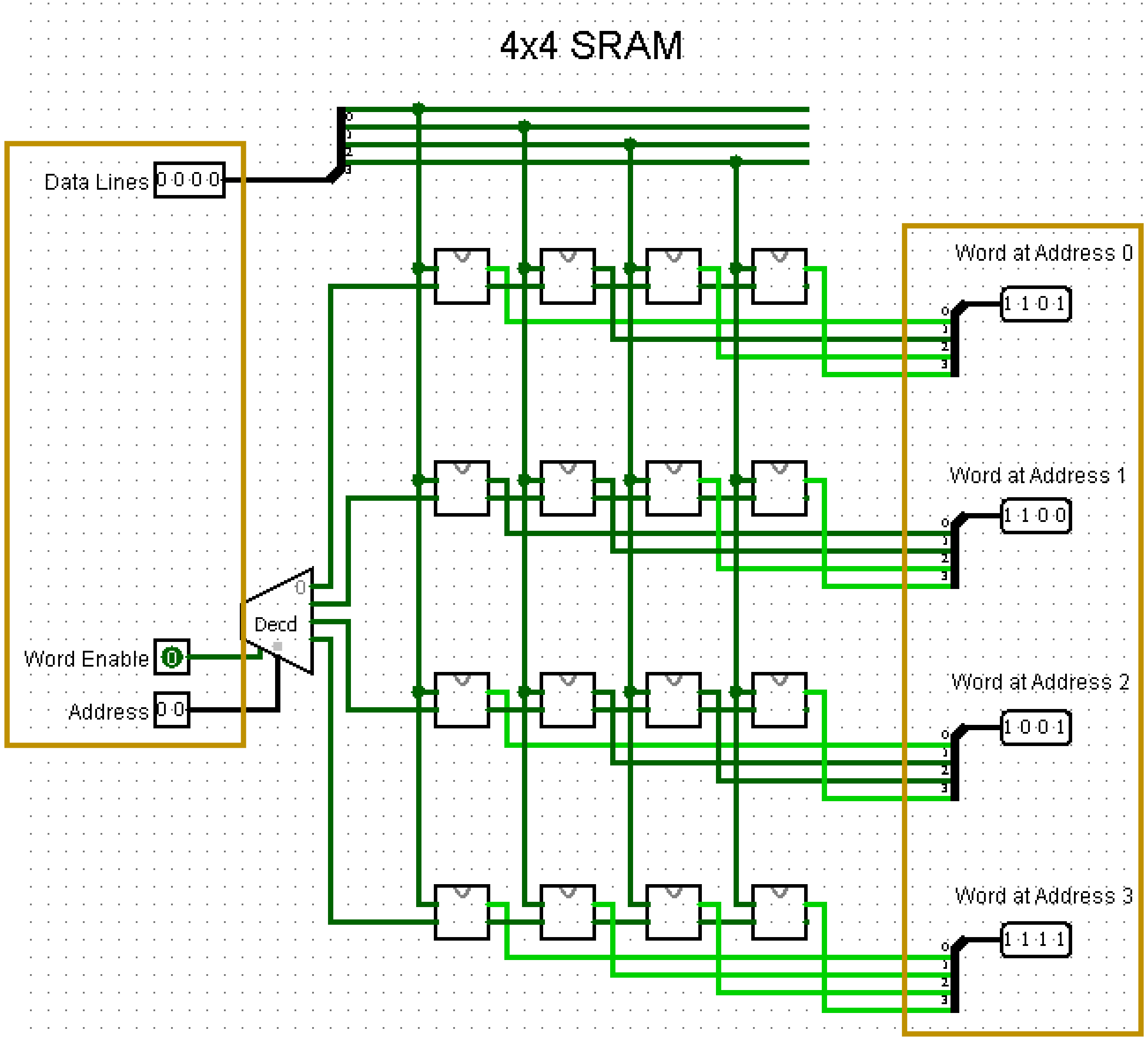
RAM


4x4 SRAM
Data Lines
1·1·0·0
Word Enable
1
Address
0·1
Decd
0
Word at Address 0
1·1·0·1
Word at Address 1
1·1·0·0
Word at Address 2
1·0·0·1
Word at Address 3
0·0·0·0

RAM


4x4 SRAM
Data Lines
1 1 1 1
0
1
2
3
Word at Address 0
1 1 0 1
Word at Address 1
1 1 0 0
Word at Address 2
1 0 0 1
Word at Address 3
1 1 1 1
Decd
0
Word Enable
1
Address
1 1

RAM


4x4 SRAM
Data Lines
0·0·0·0
Word Enable
0
Address
0·0
Decd
0
Word at Address 0
1·1·0·1
Word at Address 1
1·1·0·0
Word at Address 2
1·0·0·1
Word at Address 3
1·1·1·1

RAM

- Repeated reads and writes are required to refresh the capacitor in a DRAM cell
    - As opposed to the static storage of SRAM
    - This gives SRAM faster access than DRAM

- DRAM cells have fewer components and thus much smaller than SRAM cells
    - DRAM cells can be packed more densely than SRAM cells
    - DRAM is cheaper per bit than SRAM

RAM

- An improvement of DRAM is **S**ynchronous **DRAM** (**SDRAM**)
  - The clock keeps the processor and main memory synchronized.

- The use of a clock gives SDRAM the benefit of implementing pipelining
  - Pipelining allows a device to perform multiple operations at once
  - For example, SDRAM might output data to the processor while receiving the next address- simultaneously
  - We'll discuss pipelining in a later lecture

RAM

- An improvement of SDRAM allows two words to be written or read during a single clock cycle
  - One word when the clock is 1, and another word when the clock is 0
- This is referred to as **D**ouble **D**ata **R**ate (**DDR**) **SDRAM**
  - DDR2 doubles the data rate by allowing four words to be read/written during a single cycle
  - DDR3 quadruples the data rate by allowing eight words to be read/written during a single cycle
  - DDR4 (current technology)
    - Does not increase the words read/written during a single cycle
    - Uses a more advanced architecture for a higher transfer rate with a faster clock

ROM

- **ROM** (**R**ead-**O**nly **M**emory) is a type of non-volatile memory that retains its stored bits.

- ROM is used by technologies that write data slowly, meaning writing data to the ROM is less frequent than reading data from it.
  - Contrary to what its name suggests, writing to ROM (*"programming the ROM"*) is possible.

- ROM commonly uses a floating-gate transistor (FGT), where electrons remain trapped even when no longer powered
  - A large positive voltage traps the electrons
  - A large negative voltage releases the electrons

ROM

- There are several different types of ROM
  - **Masked-Programmed ROM**: The word line to bit line connections are hardwired and can never be changed

  - **O**ne-**T**ime **P**rogrammable **ROM** (**OTP ROM**): The word line to bit line connections have a fuse that, when blown, can break the connection; Can only be programmed once.

  - **E**rasable **P**rogrammable **ROM** (**EPROM**): Electrons are trapped in FGTs using a large positive voltage; Electrons are released when the chip is exposed to ultraviolet light

ROM

- **E**lectrically **E**rasable **P**rogrammable **ROM** (**EEPROM**): Electrons are trapped in FGTs using a large positive voltage; Electrons are released from FGTs using a large negative voltage.
- **Flash**: A type of EEPROM; Electrons are trapped in FGTs using a large positive voltage; Using a large negative voltage, electrons in entire blocks of FGTs are released at once.

EPROM

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

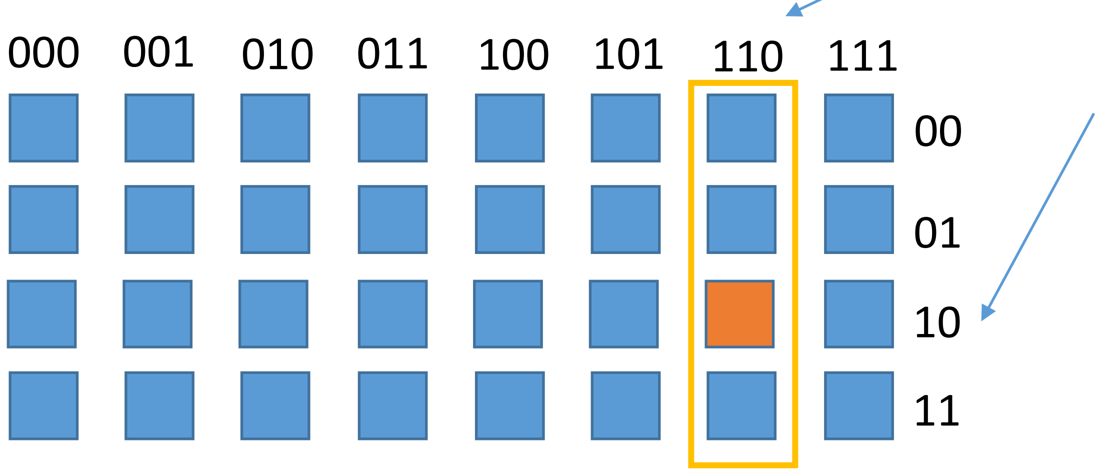
- One way is that whenever a write operation occurs, the new data in the cache is written back to its corresponding address in main memory

- 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 through 31

First 3 bits of address

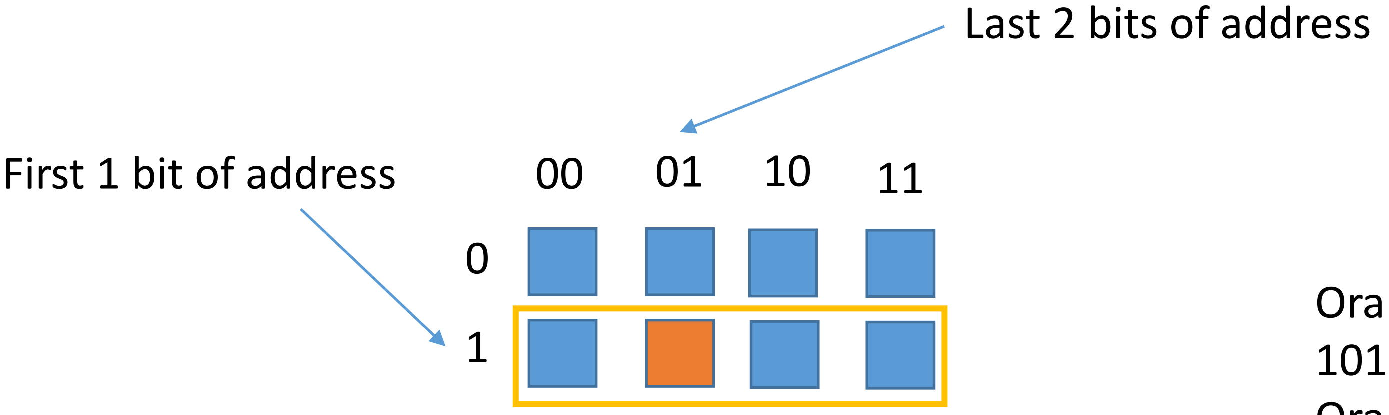


Last 2 bits of address

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

Cache Memory – Direct Mapping

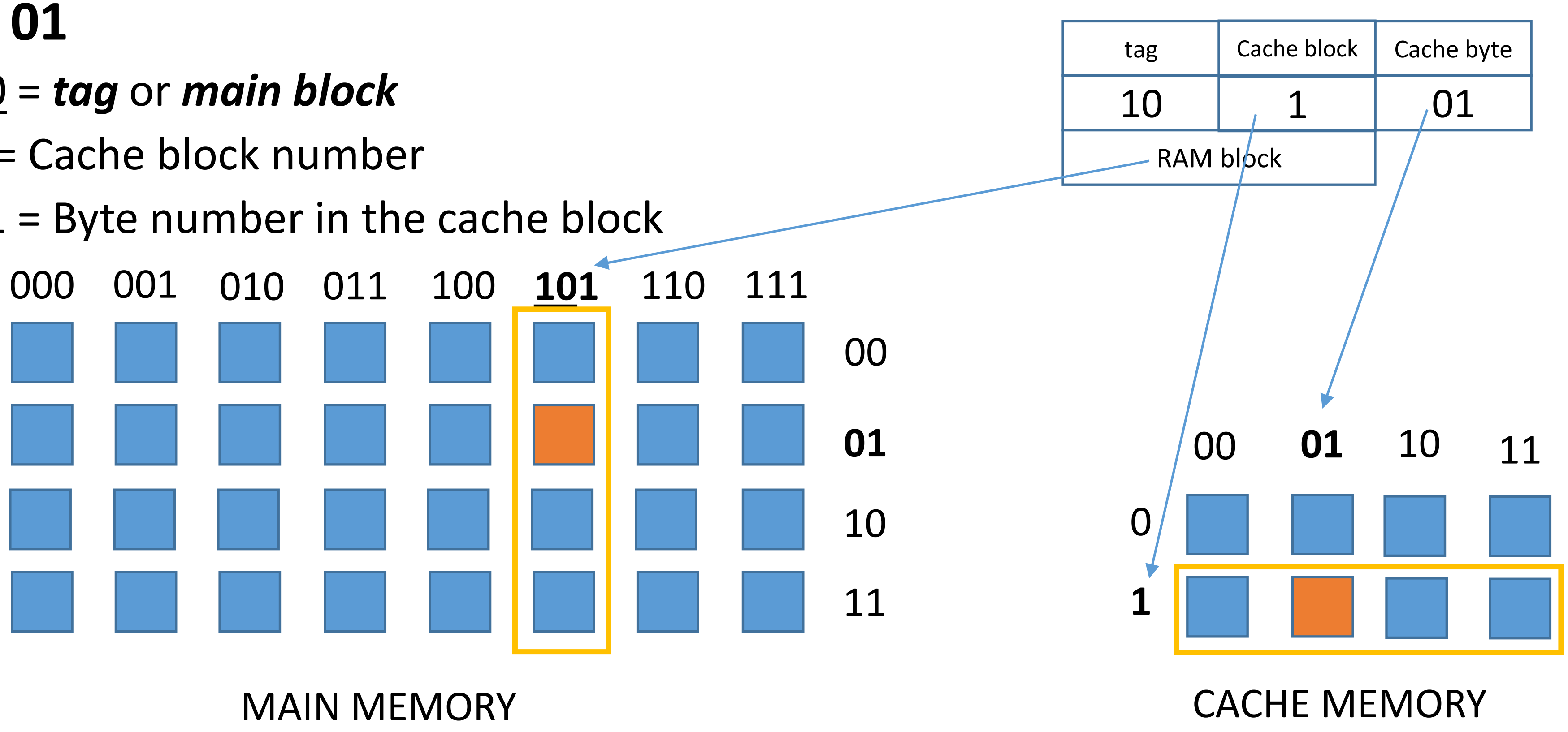
- We'll consider a simple CPU that has 16 bytes of cache memory
  - Each square represents one byte
  - Each **row** is a block
  - Addresses 0 through 7



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

Cache Memory – Direct Mapping

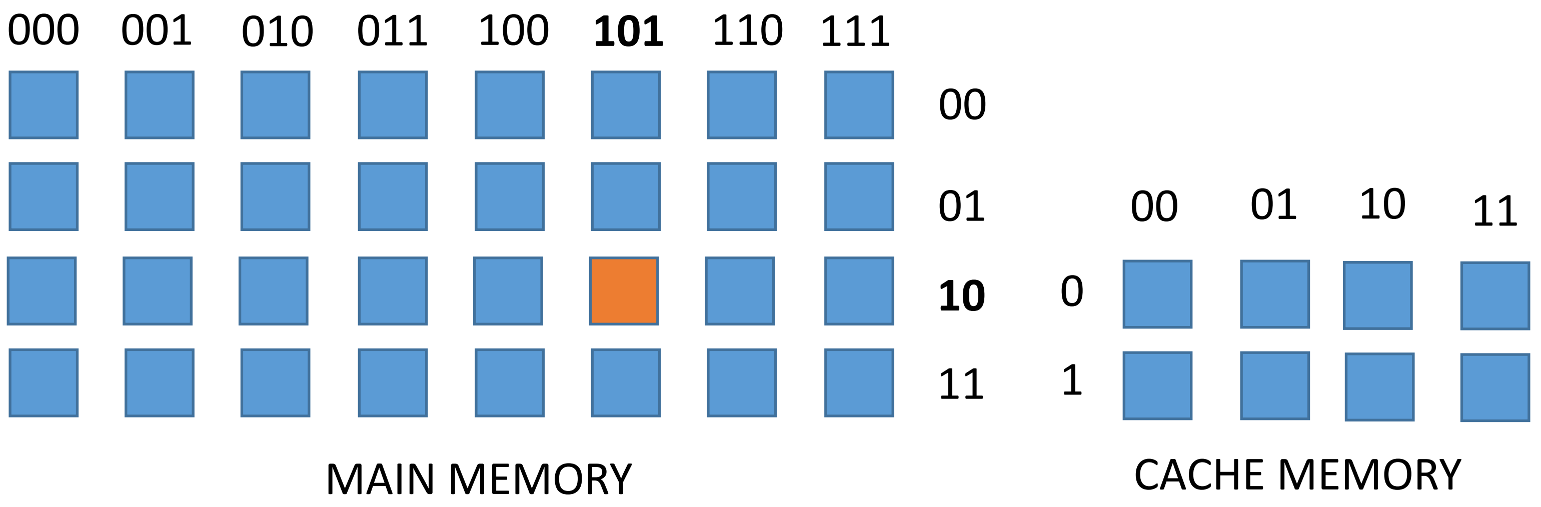
- Let’s see how the main memory address 10101 maps to the cache
  - **10 1 01**
    - 10 = ***tag*** or ***main block***
    - 1 = Cache block number
    - 01 = Byte number in the cache block



tag	Cache block	Cache byte
10	1	01
RAM block		

Cache Memory – Direct Mapping

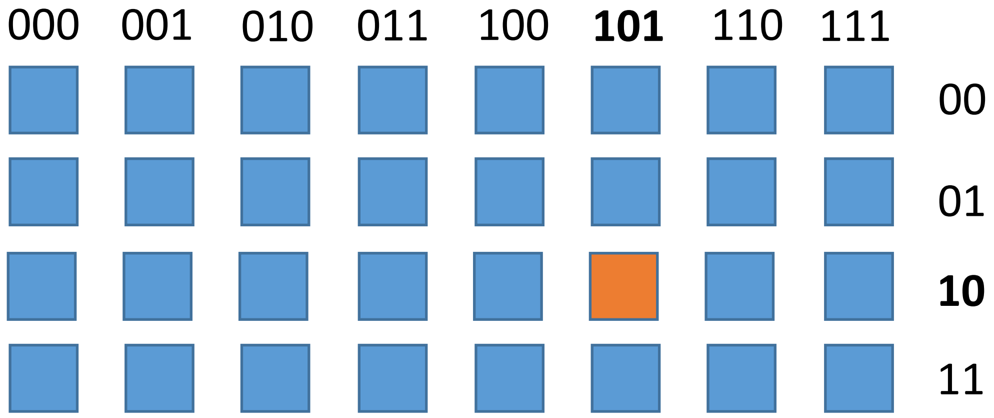
- The CPU is instructed to read the byte at address **10110** from main memory
  - The orange byte below



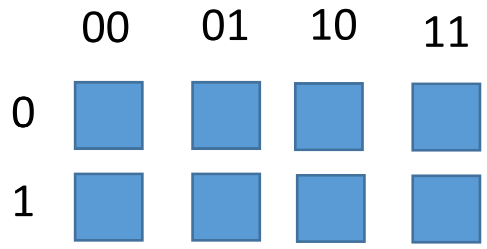
Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
	1 00	
	1 01	
	1 10	
	1 11	

Cache Memory – Direct Mapping

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



MAIN MEMORY

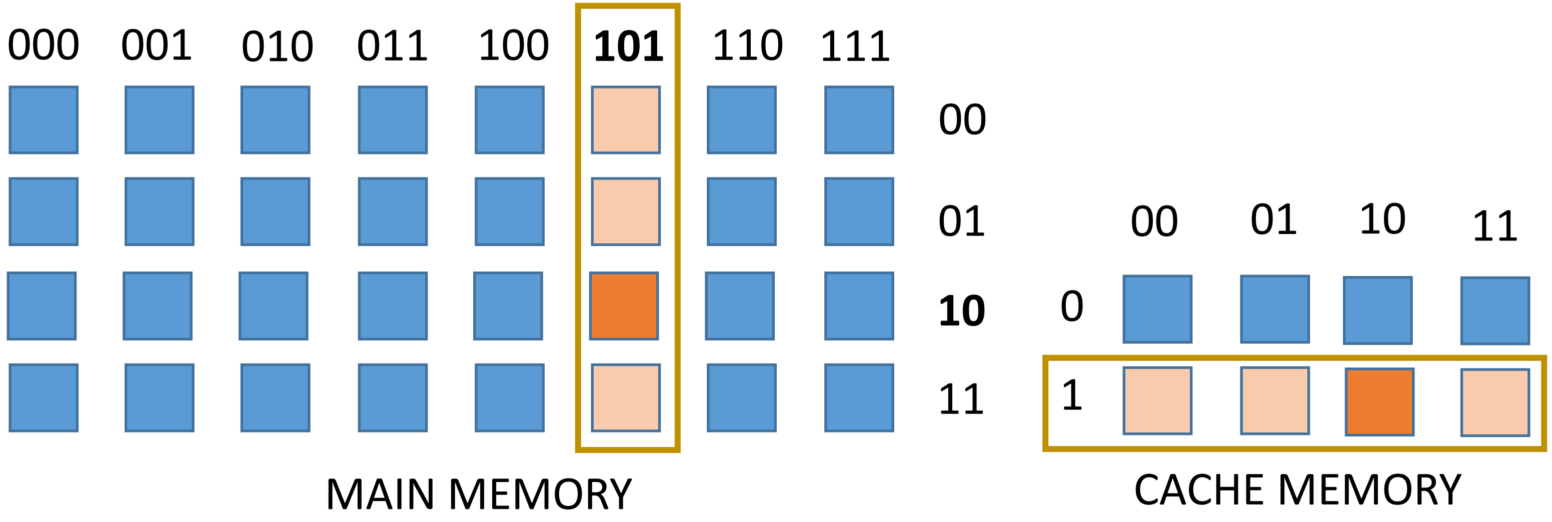


CACHE MEMORY

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

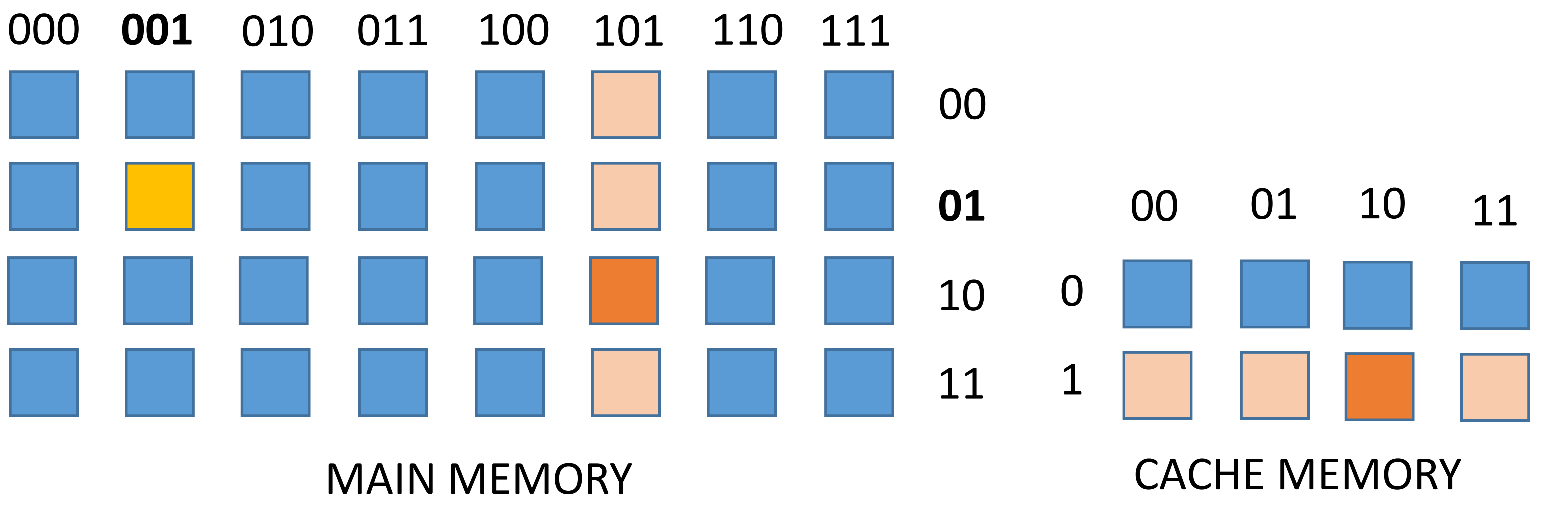
- The block (block 5) is loaded into cache
  - 10 1 00
  - 10 1 01
  - 10 1 10
  - 10 1 11



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

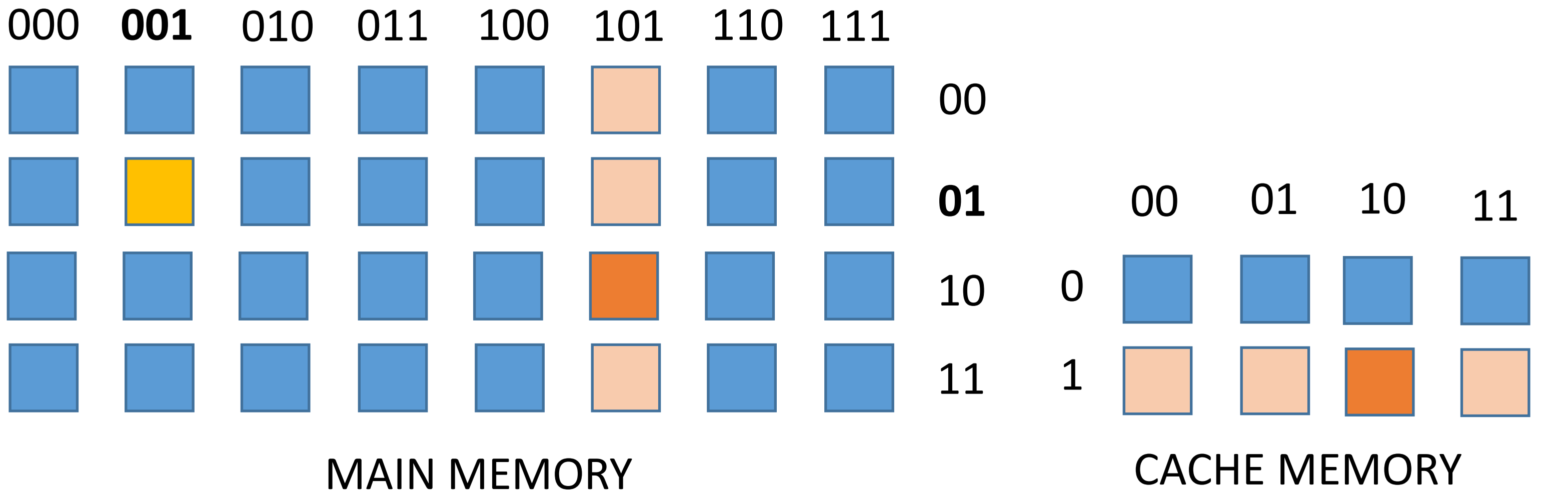
- The CPU is instructed to read the byte at address 00101 from main memory
  - The yellow byte below



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

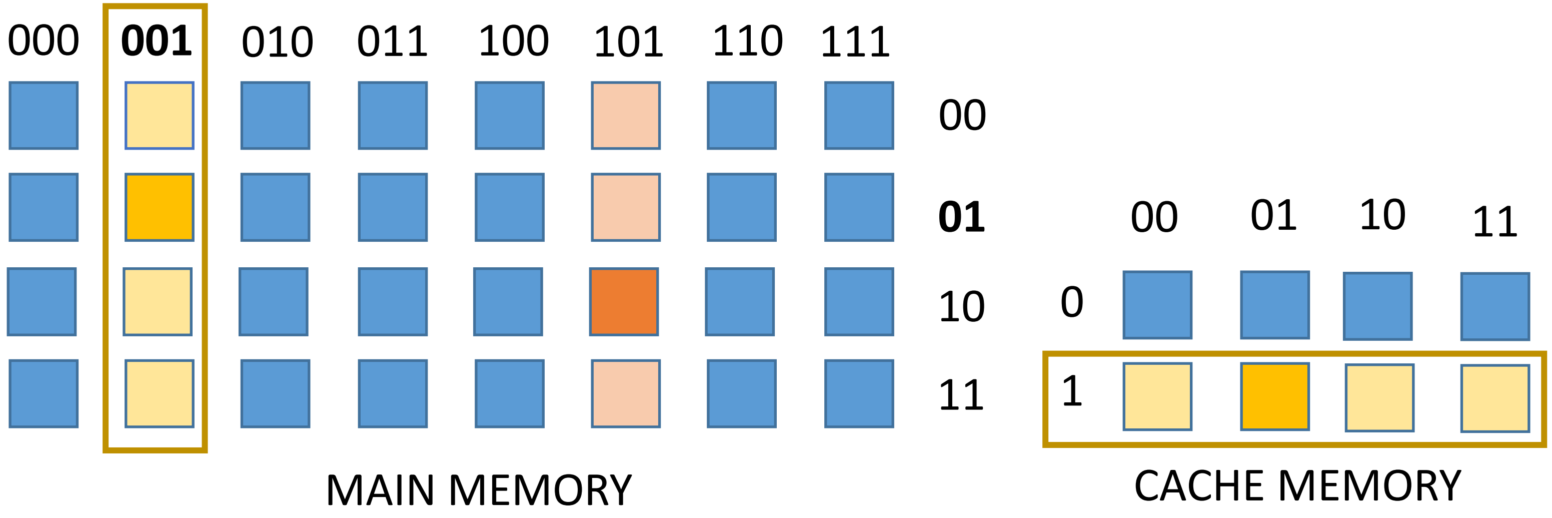
- First the CPU checks if the value is in the cache
  - **00 1 01**
  - Data in cache address 101, but wrong tag (*cache miss*)



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

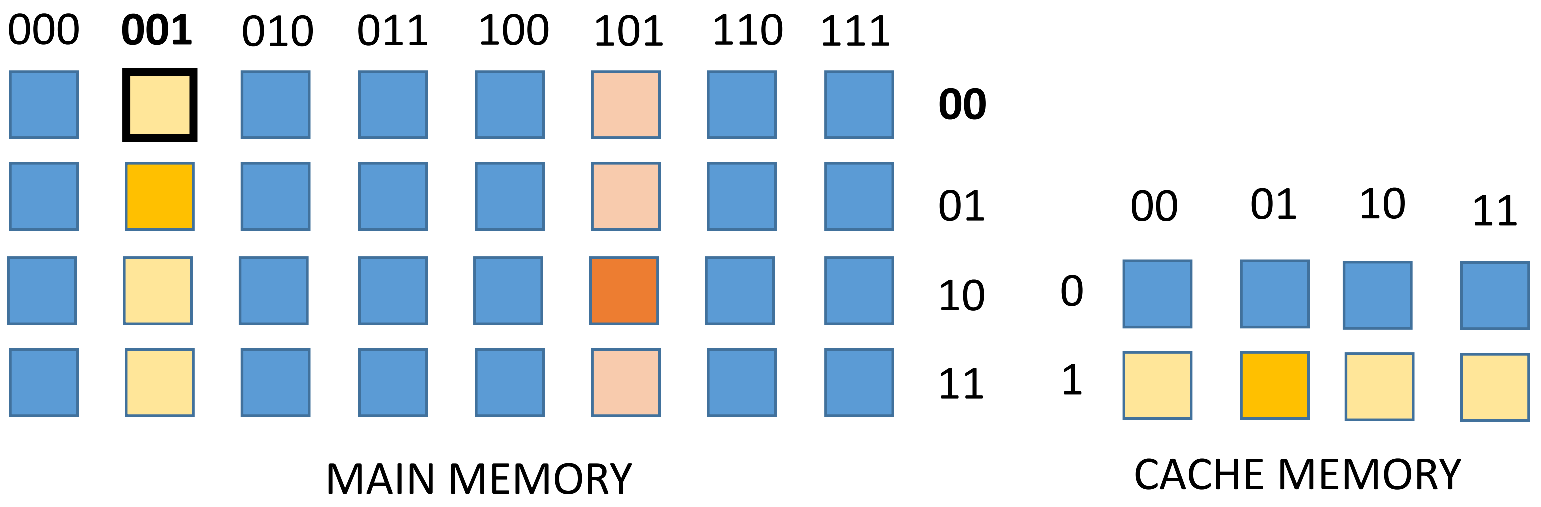
- The block (block 1) is loaded into cache
  - 00 1 00
  - 00 1 01
  - 00 1 10
  - 00 1 11



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

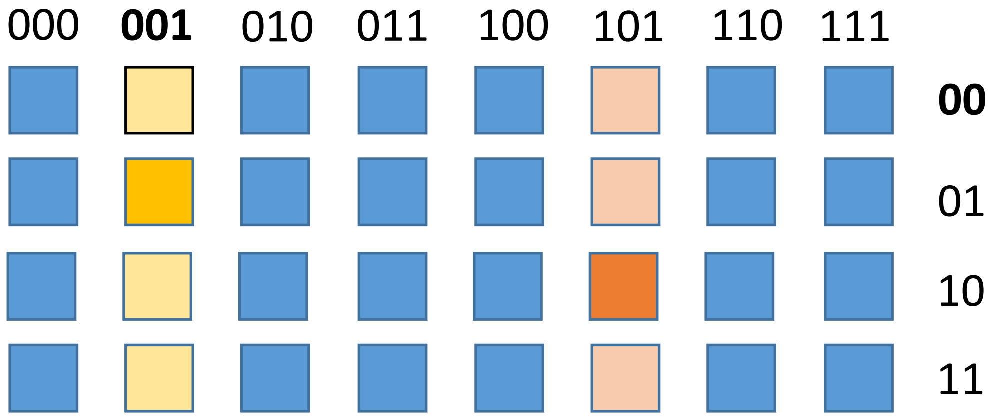
- The CPU is instructed to read the byte at address **00100** from main memory



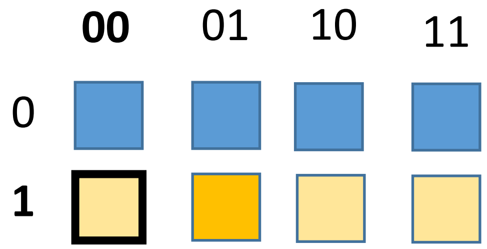
Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
00	1 00	🟨 00100
00	1 01	🟧 00101
00	1 10	🟨 00110
00	1 11	🟨 00111

Cache Memory – Direct Mapping

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



MAIN MEMORY

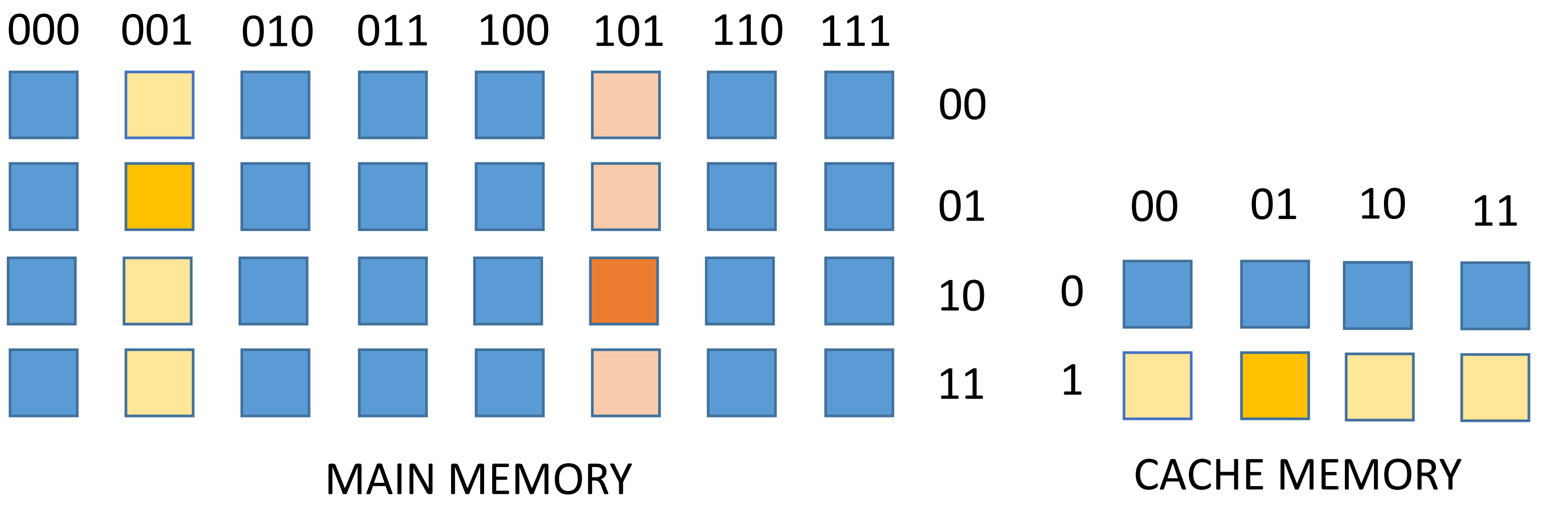


CACHE MEMORY

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

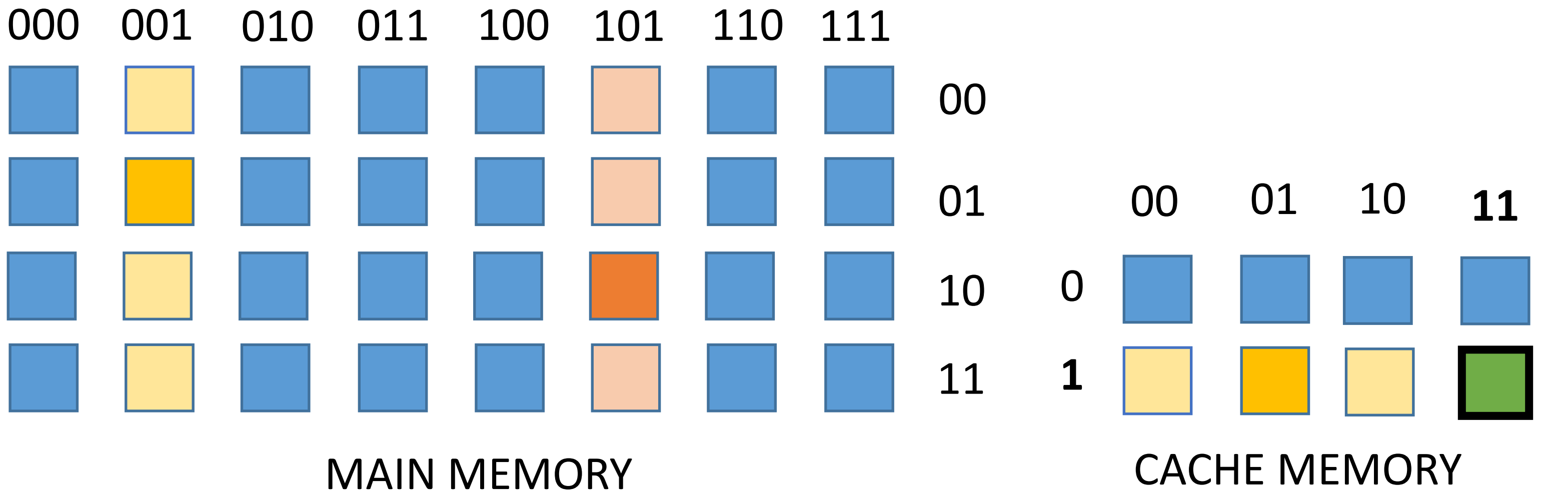
- The CPU is instructed to store a byte to address 00111 in main memory
  - Will be represented as 🟩



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

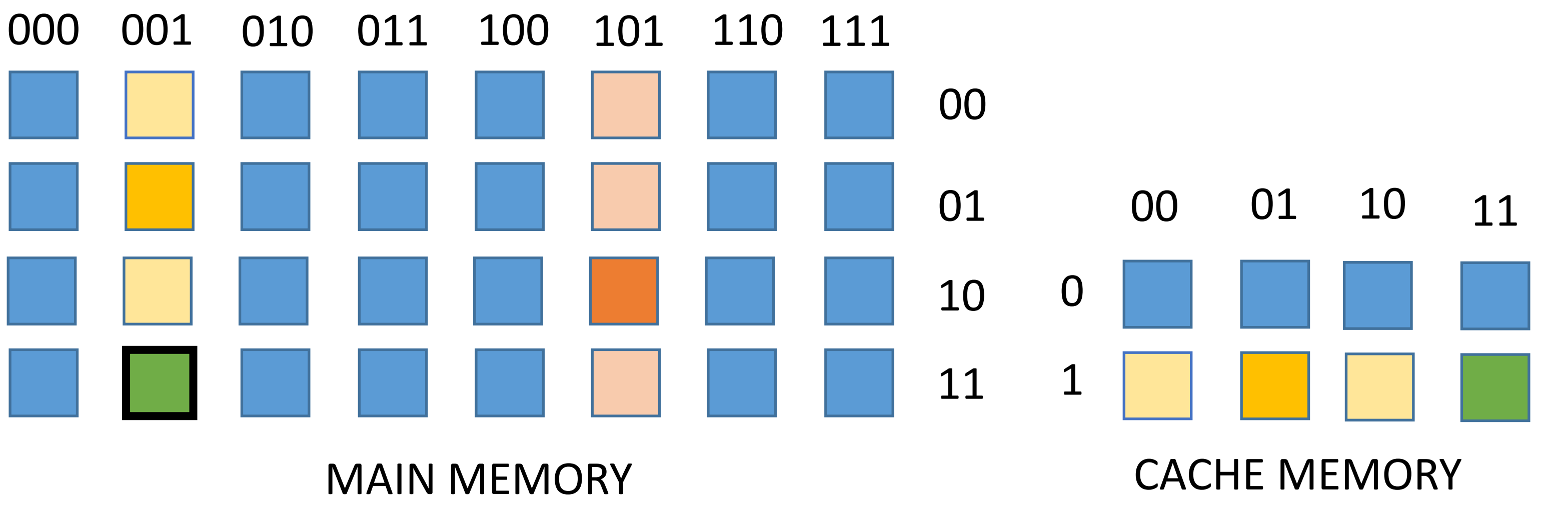
- The CPU writes the value to the cache
  - 00 **1 11**
  - Data in cache address 111 with correct tag (*cache hit*)
    - Data in cache is updated



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

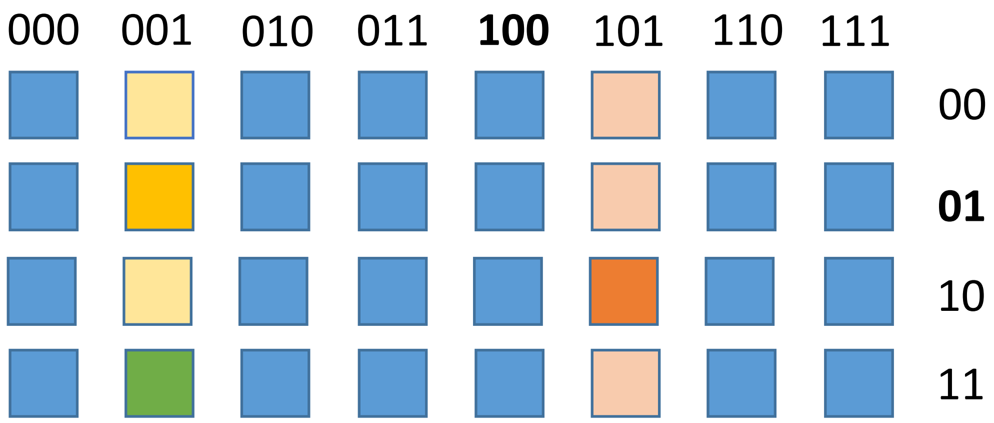
- Data is stored to main memory (or dirty bit is set)



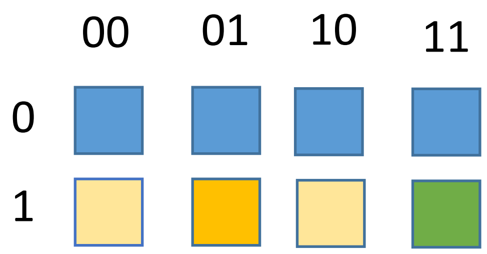
Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
00	1 00	🟨 00100
00	1 01	🟧 00101
00	1 10	🟨 00110
00	1 11	🟩 00111

Cache Memory – Direct Mapping

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



MAIN MEMORY

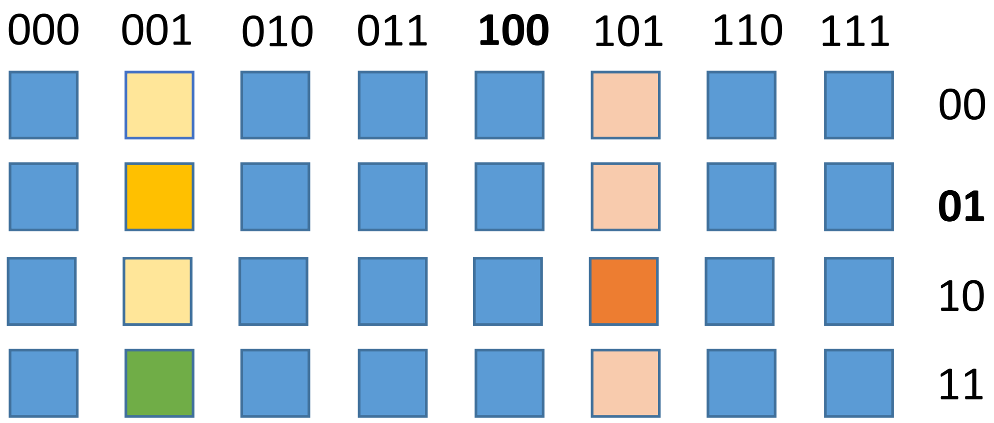


CACHE MEMORY

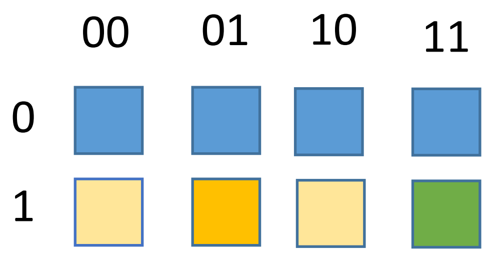
Tag	Cache Address *Set Byte*	Data/Main Memory Address
	0 00	
	0 01	
	0 10	
	0 11	
00	1 00	🟨 00100
00	1 01	🟧 00101
00	1 10	🟨 00110
00	1 11	🟩 00111

Cache Memory – Direct Mapping

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



MAIN MEMORY

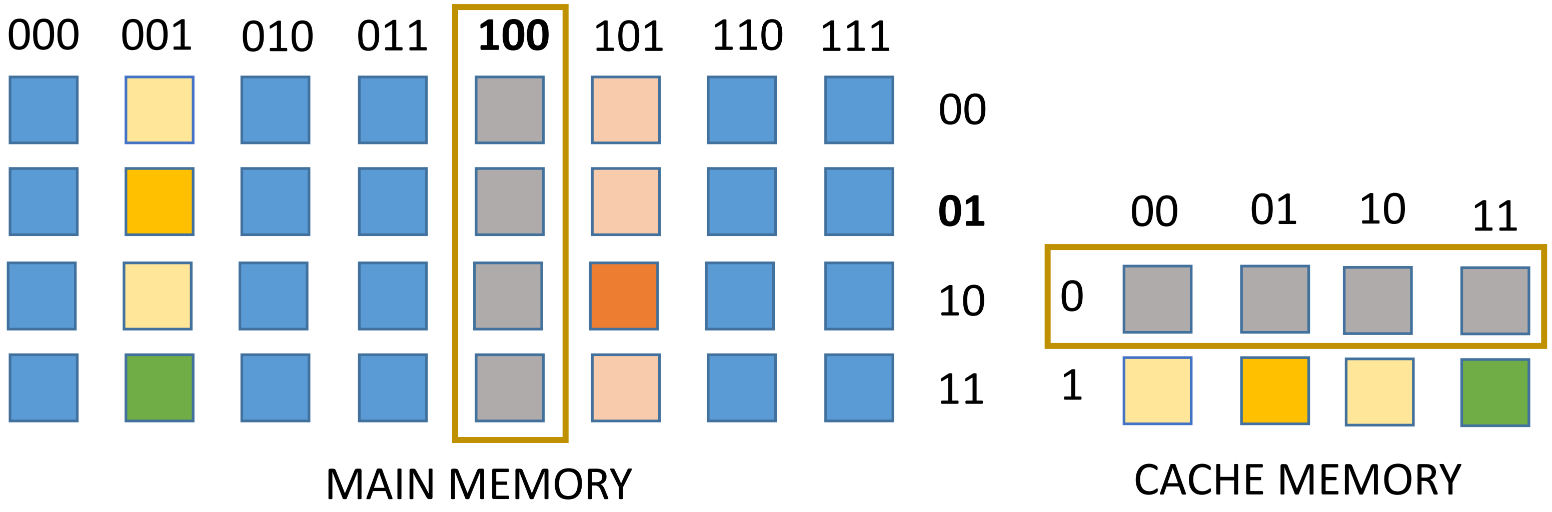


CACHE MEMORY

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

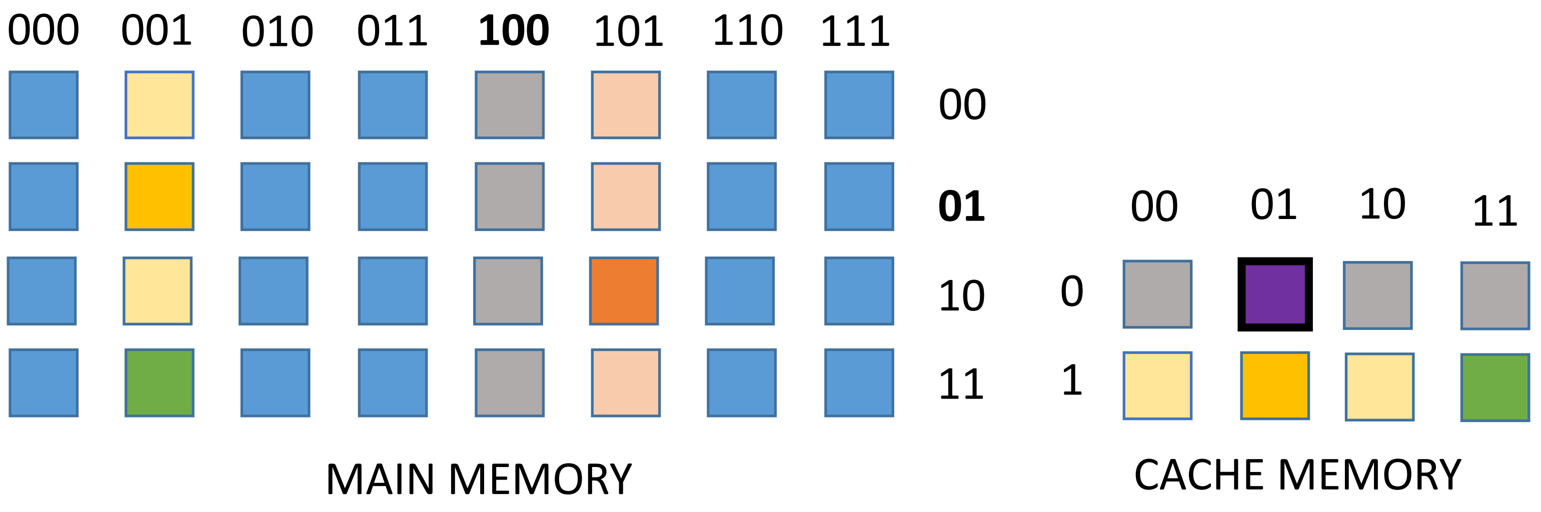
- The block (block 4) is loaded into cache
  - 10 0 00
  - 10 0 01
  - 10 0 10
  - 10 0 11



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

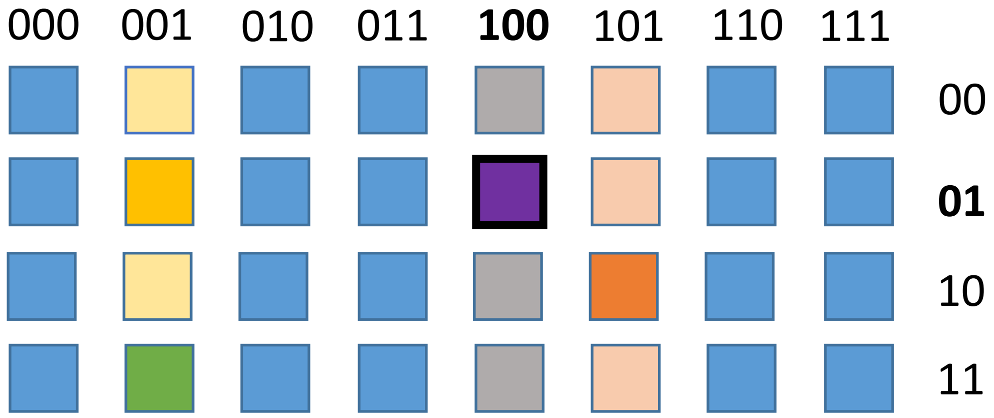
- The CPU writes the value to the cache
  - 10 **0 01**
  - Data in cache is updated



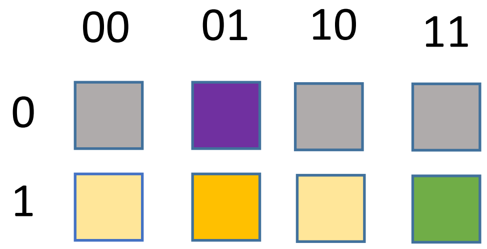
Tag	Cache Address *Set Byte*	Data/Main Memory Address
10	0 00	▨ 10000
10	**0 01**	🟪 10001
10	0 10	▨ 10010
10	0 11	▨ 10011
00	1 00	🟨 00100
00	1 01	🟧 00101
00	1 10	🟨 00110
00	1 11	🟩 00111

Cache Memory – Direct Mapping

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



MAIN MEMORY

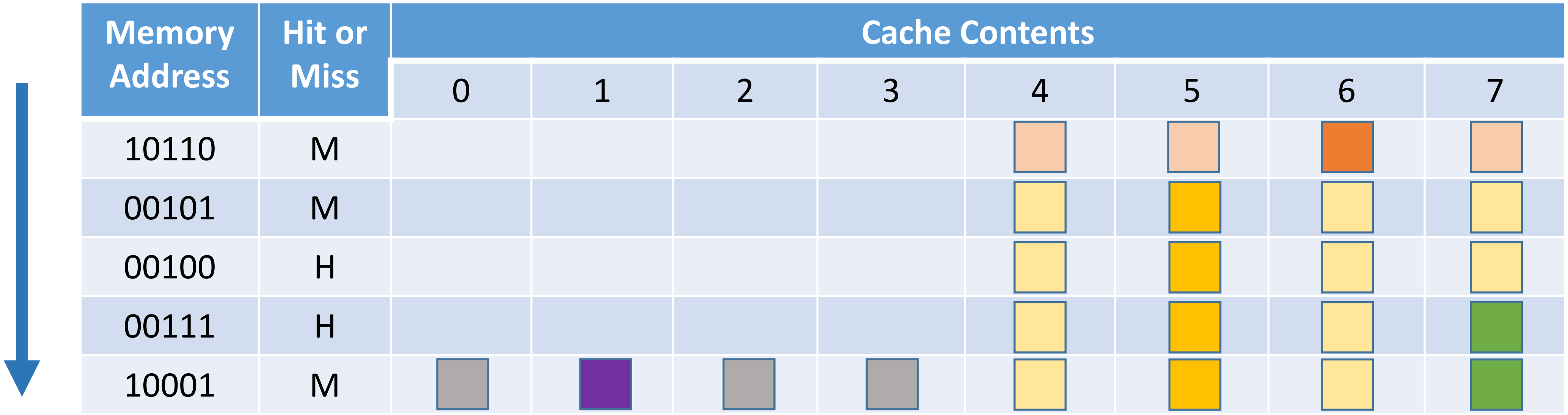


CACHE MEMORY

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

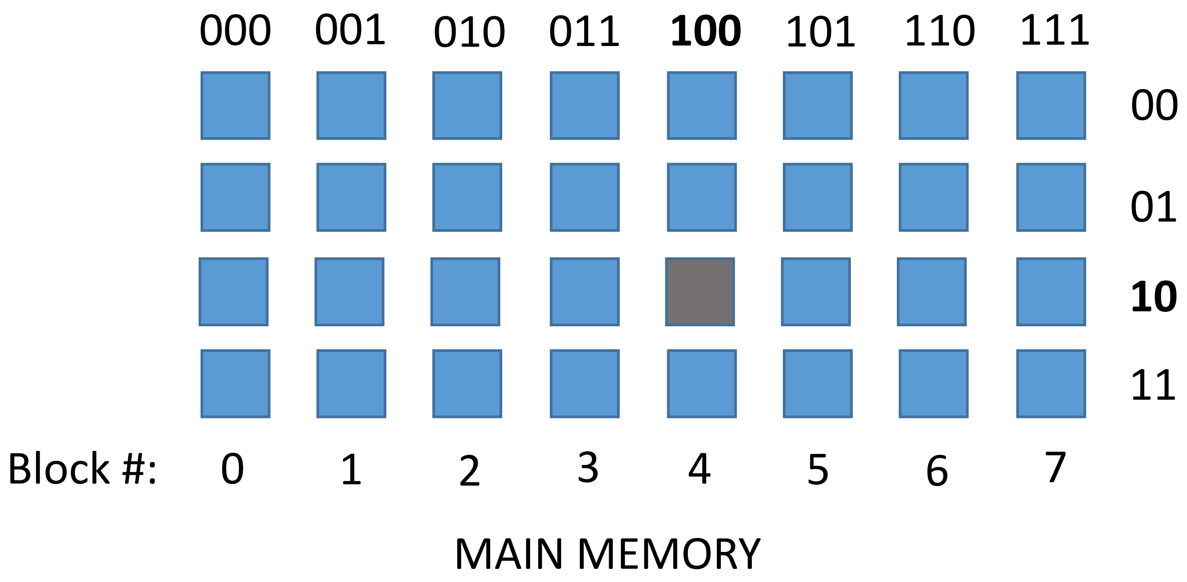
- In an **associative mapped** cache, each block of main memory is mapped to a *set of blocks* in cache memory.

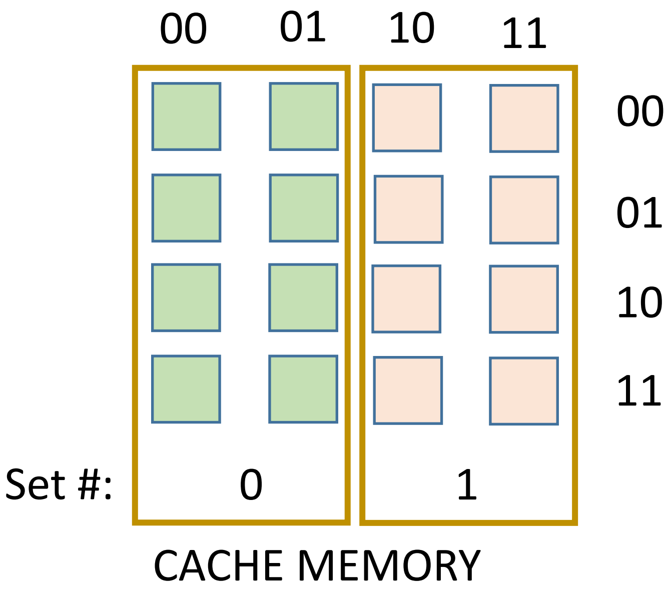
- When a block is copied from RAM to the cache, the block can be stored in any one of the blocks that belong to the set.

- 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**

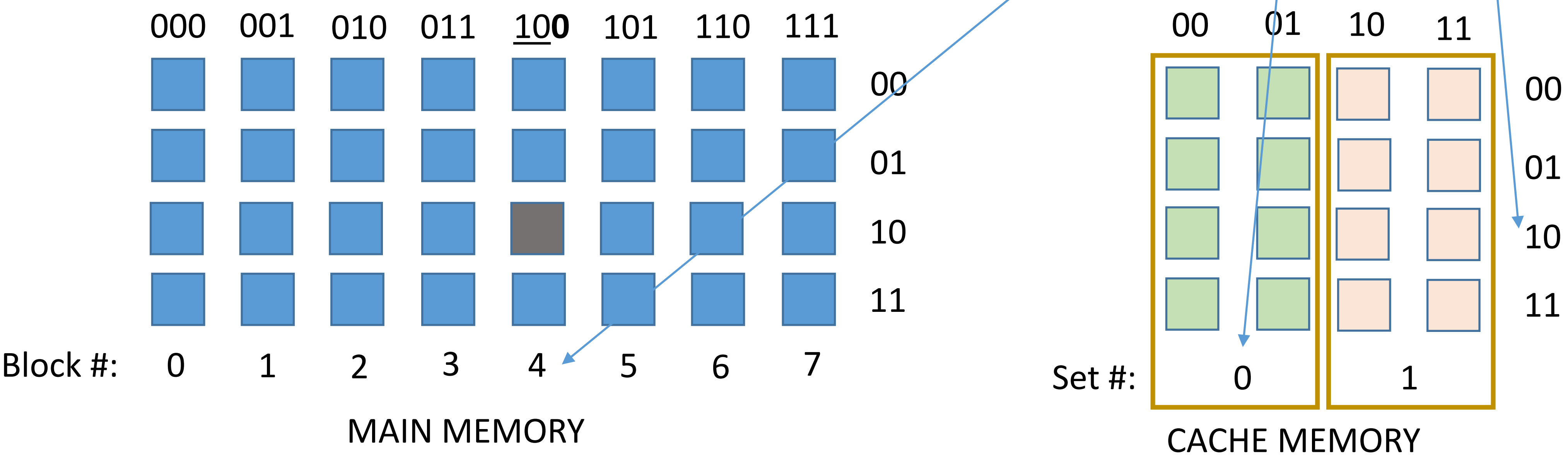




Cache Memory – Associative Set Mapping

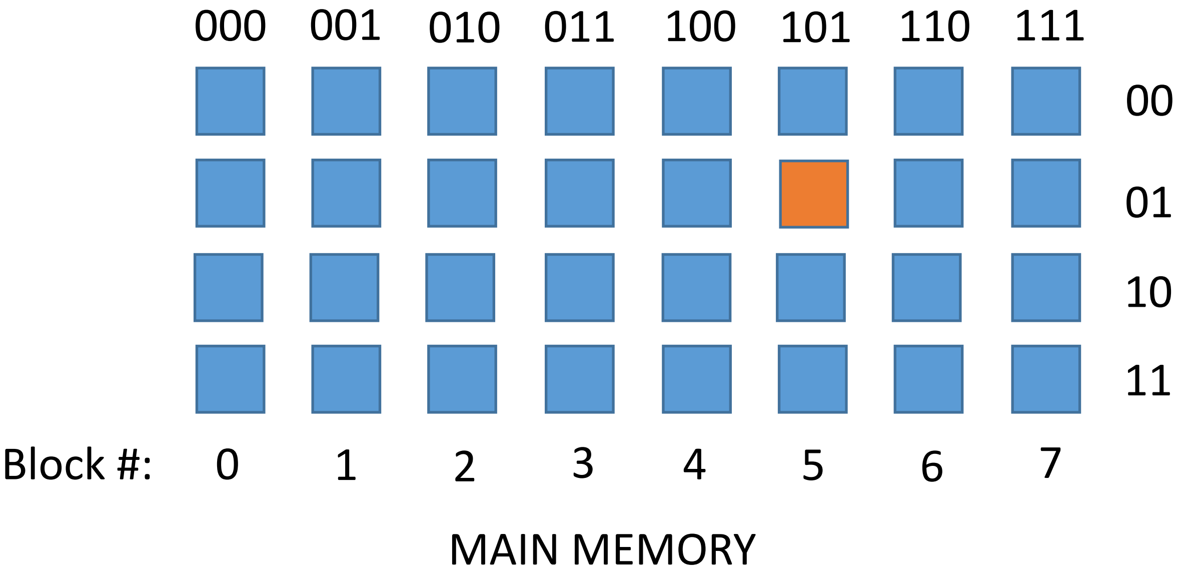
- **10 0 10**
  - 10 = ***tag*** or ***main block***
  - **0** = Cache set number
  - 10 = Byte number in the cache block

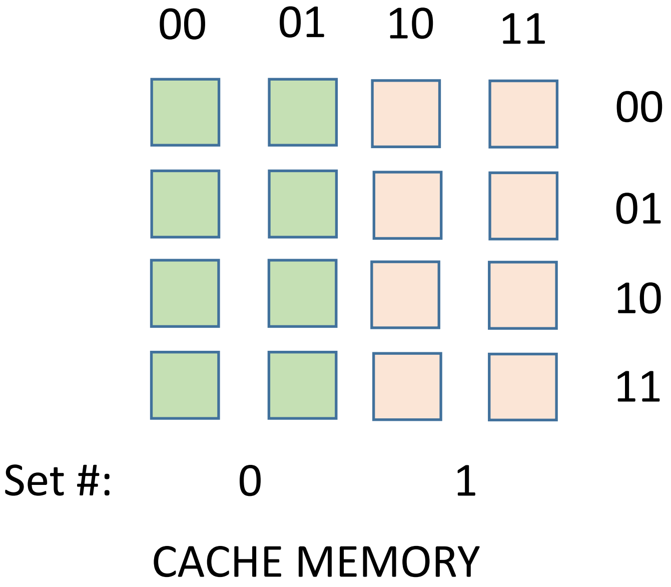
tag	Cache set	Cache byte
10	0	10
RAM block		



Cache Memory – Associative Set Mapping

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

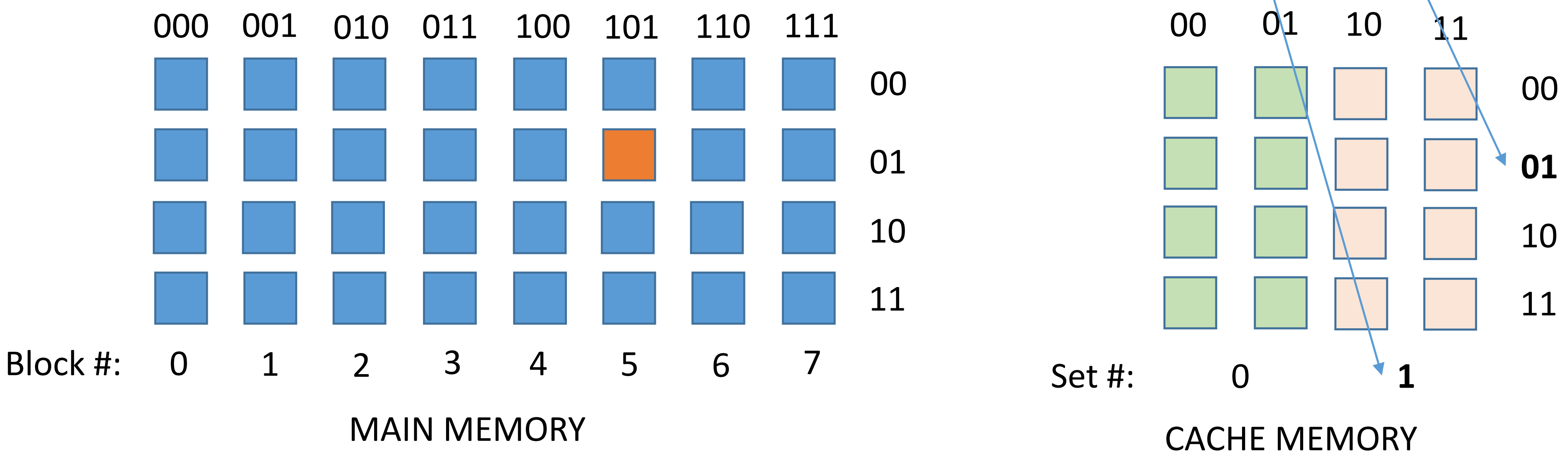




Cache Memory – Associative Set Mapping

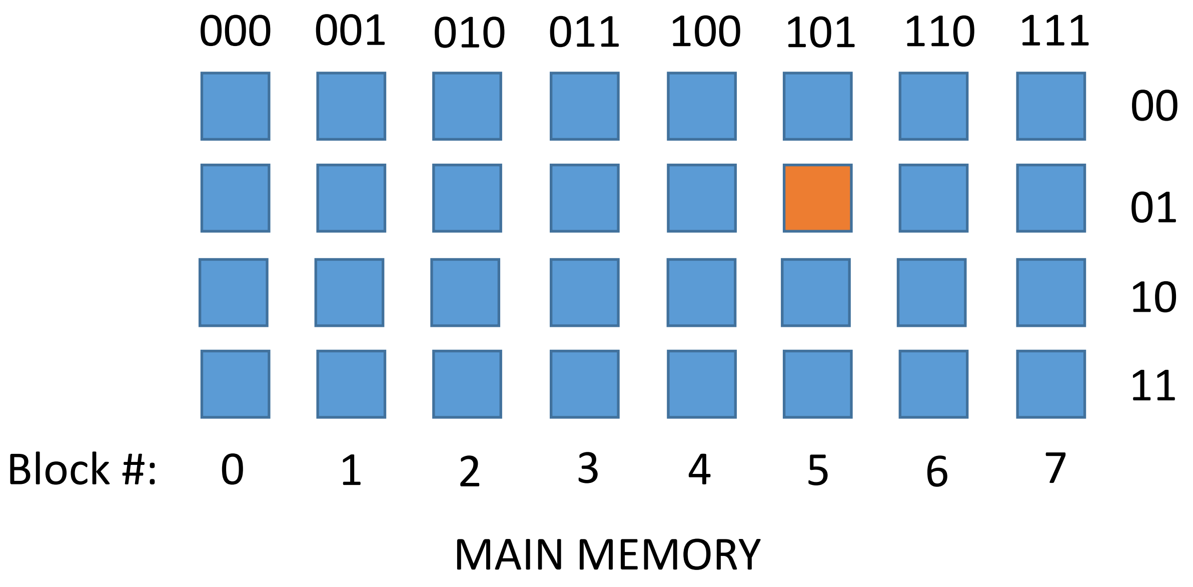
- **10 1 01**

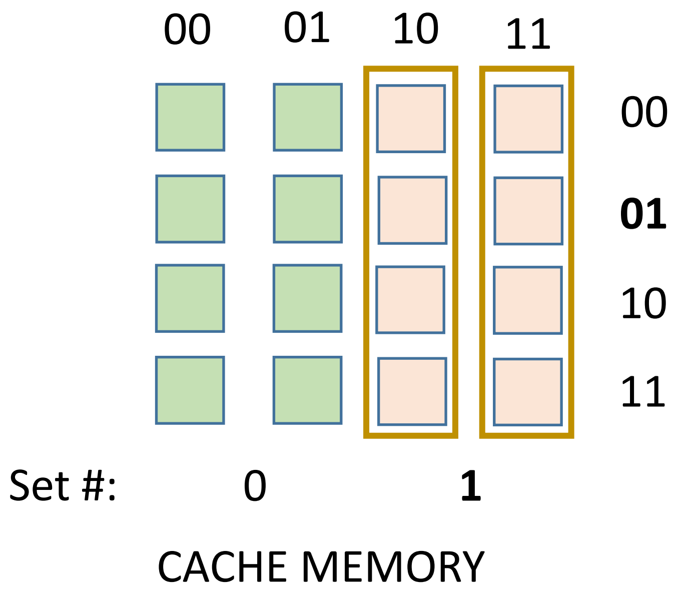
tag	Cache set	Cache byte
10	1	01
RAM block		



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





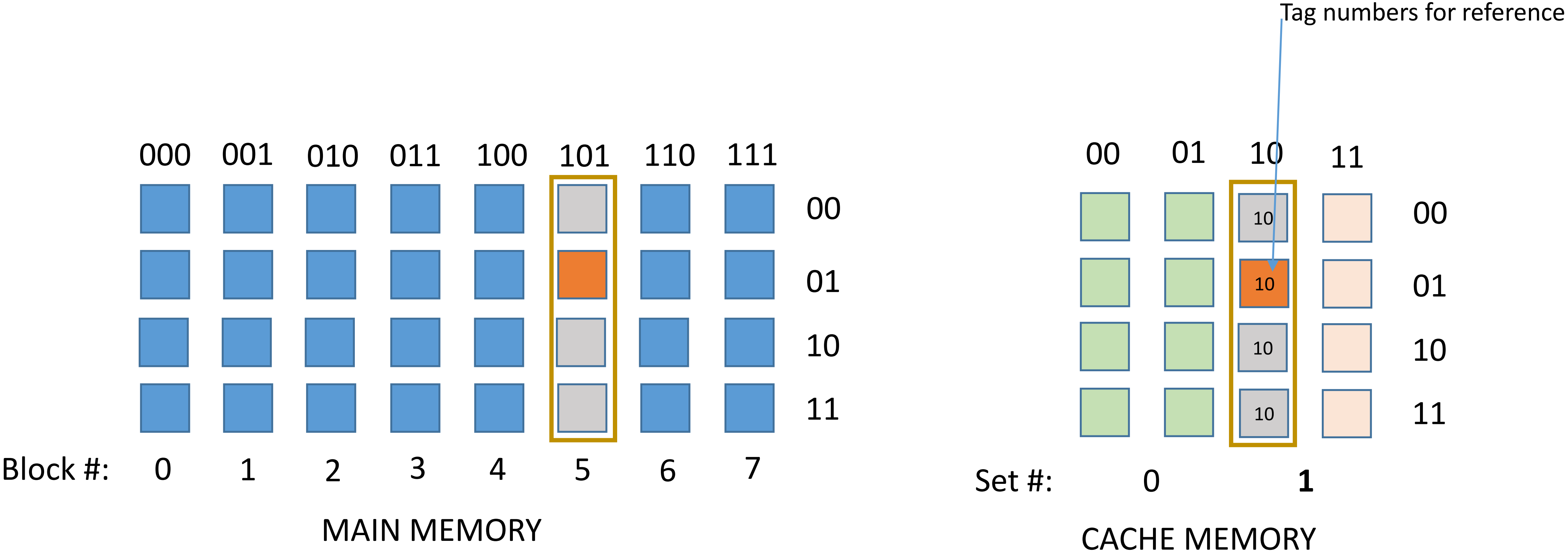
Cache Memory – Associative Set Mapping

- 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

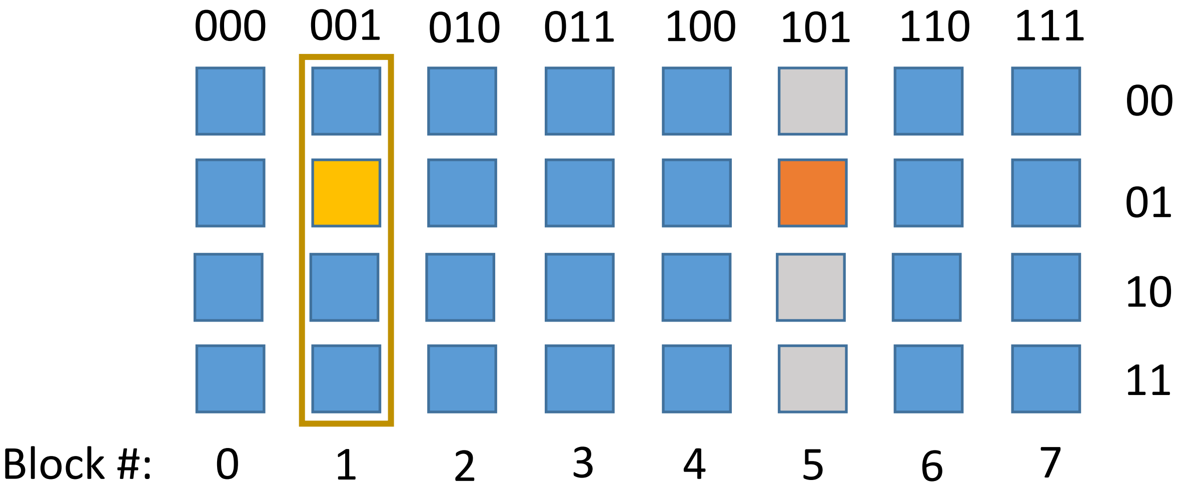


Cache Memory – Associative Set Mapping

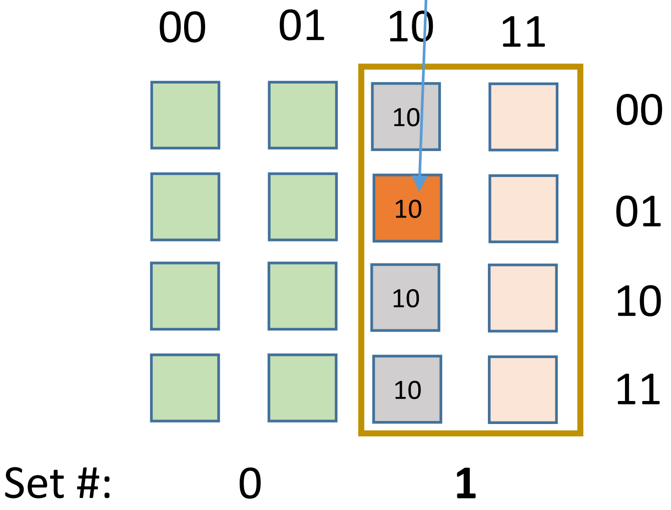
- **00101**

tag	Cache set	Cache byte
00	1	01
RAM block		

Tags don’t match (miss)



MAIN MEMORY

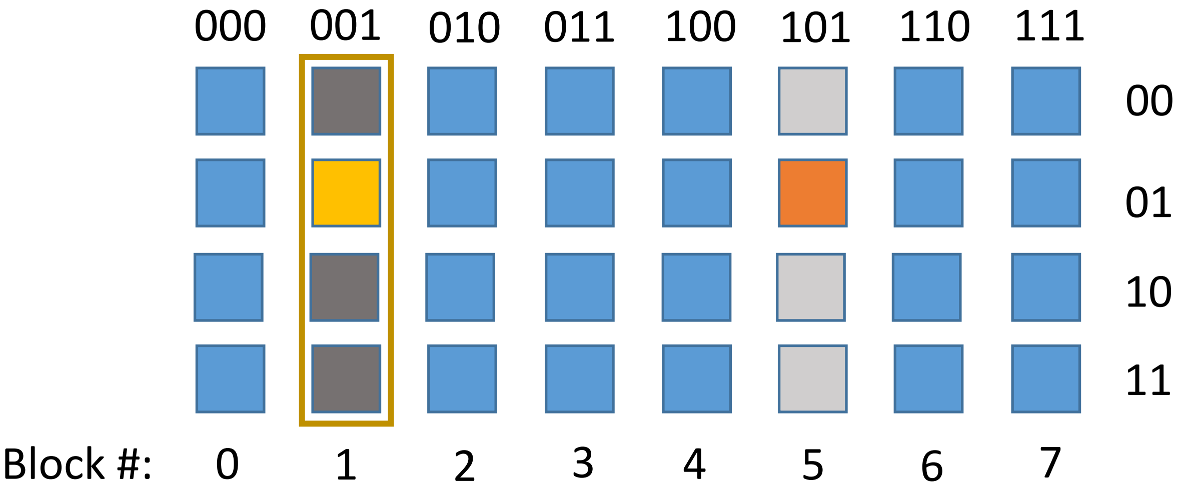


CACHE MEMORY

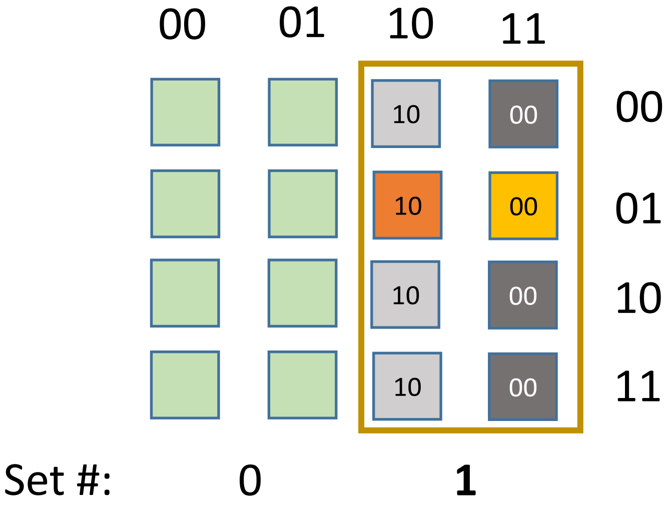
Cache Memory – Associative Set Mapping

- **00101**

tag	Cache set	Cache byte
00	1	01
RAM block		



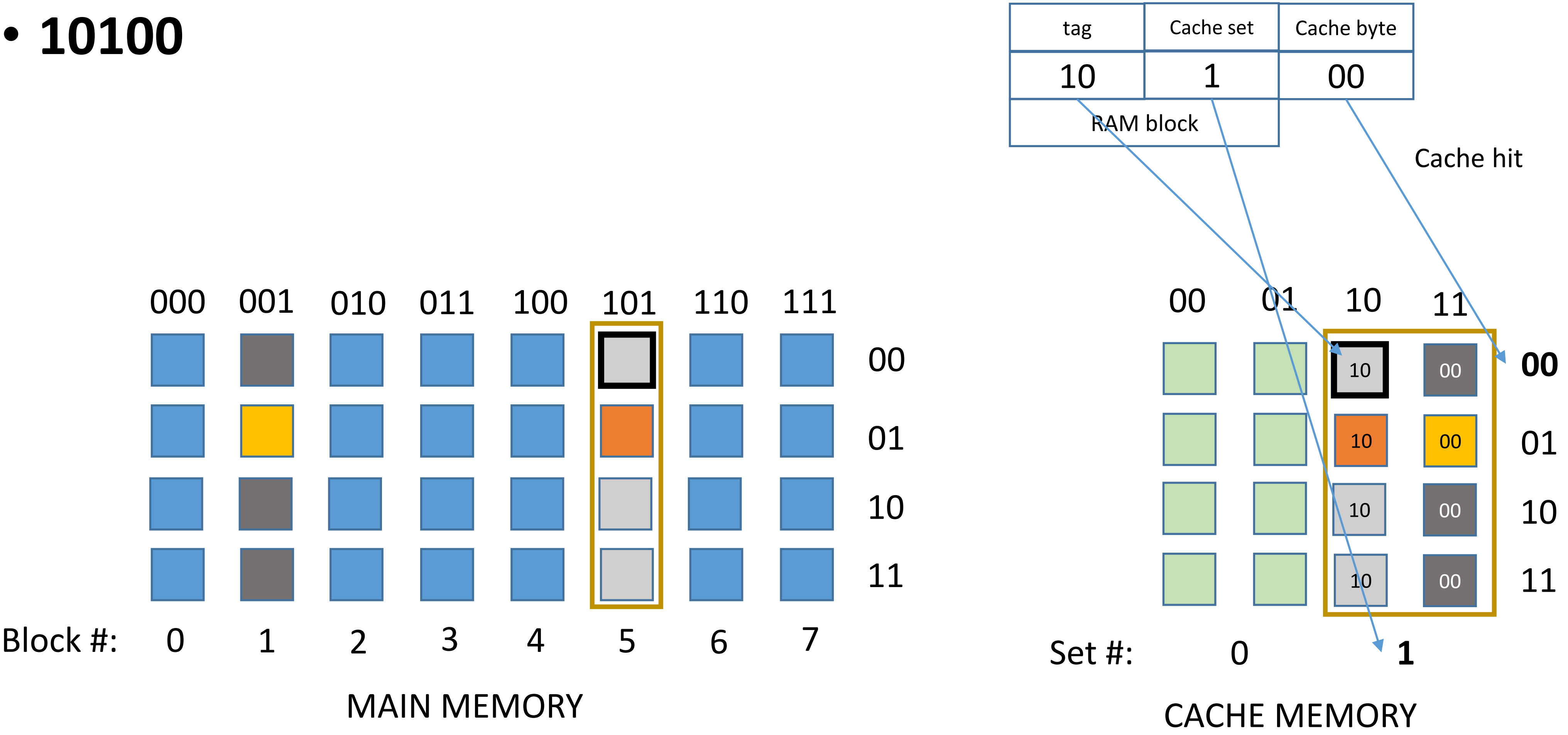
MAIN MEMORY



CACHE MEMORY

Cache Memory – Associative Set Mapping

- **10100**

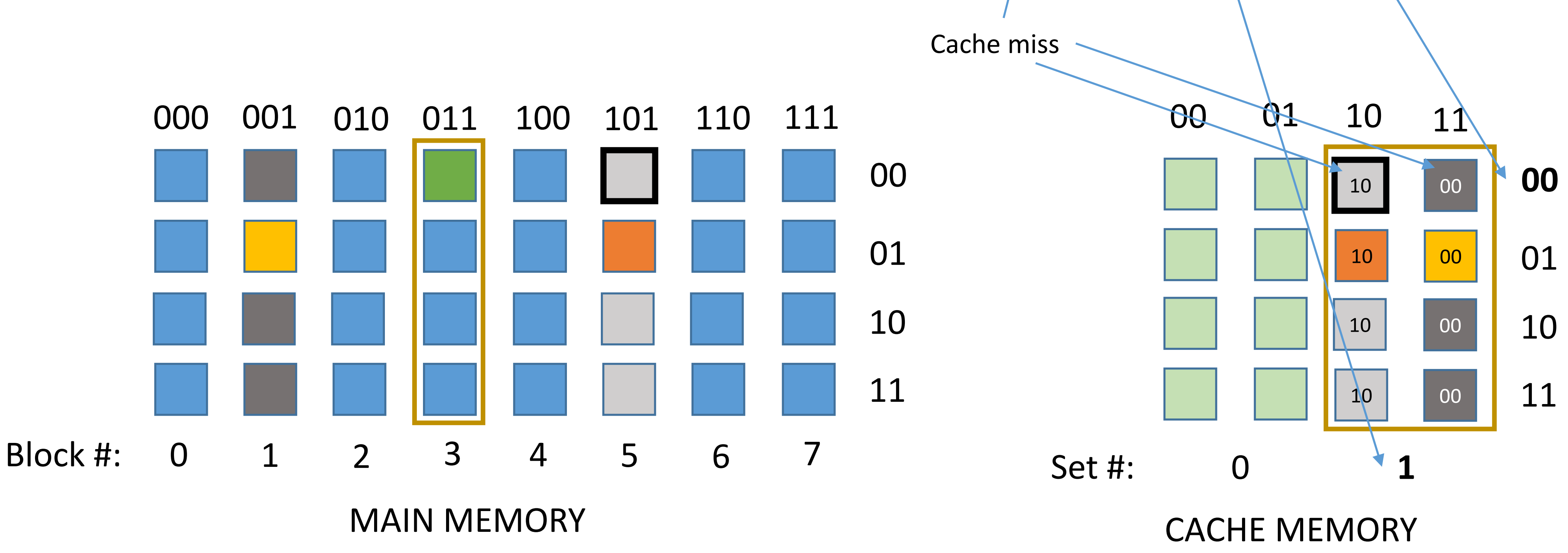


tag	Cache set	Cache byte
10	1	00
RAM block		

Cache Memory – Associative Set Mapping

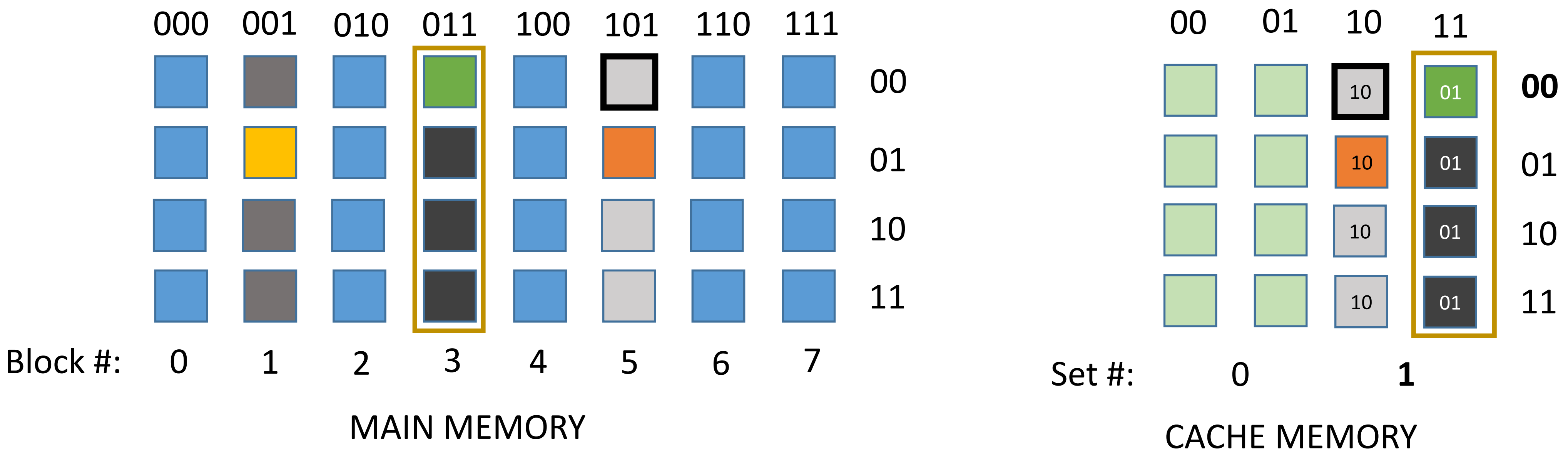
- **01100**

tag	Cache set	Cache byte
01	1	00
RAM block		



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



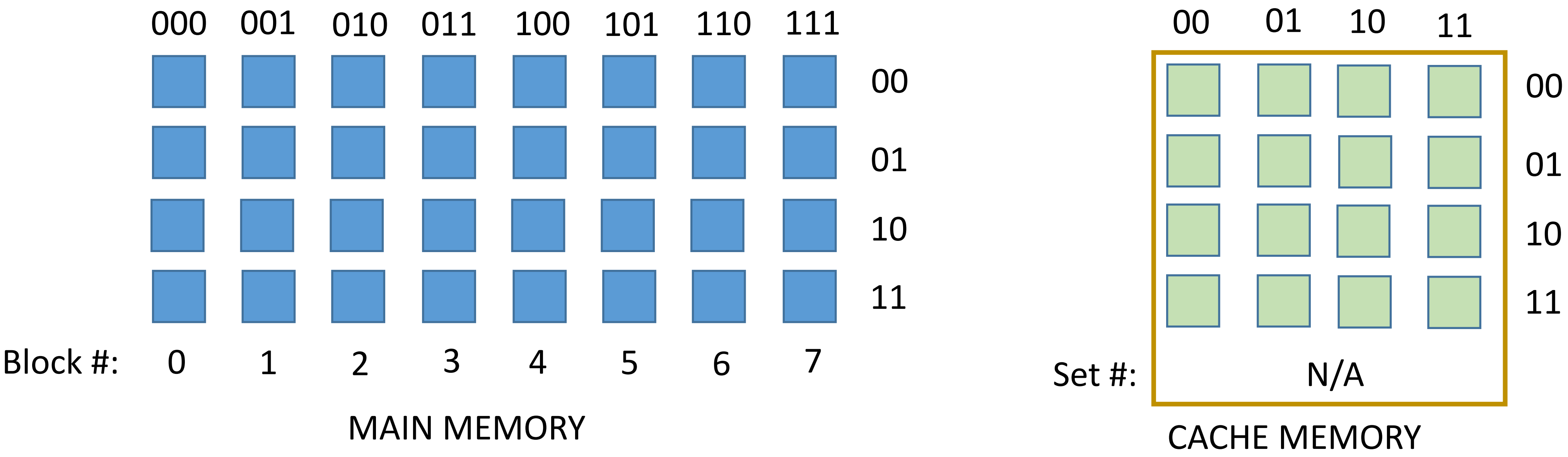
Cache Memory – Associative Set Mapping

- 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

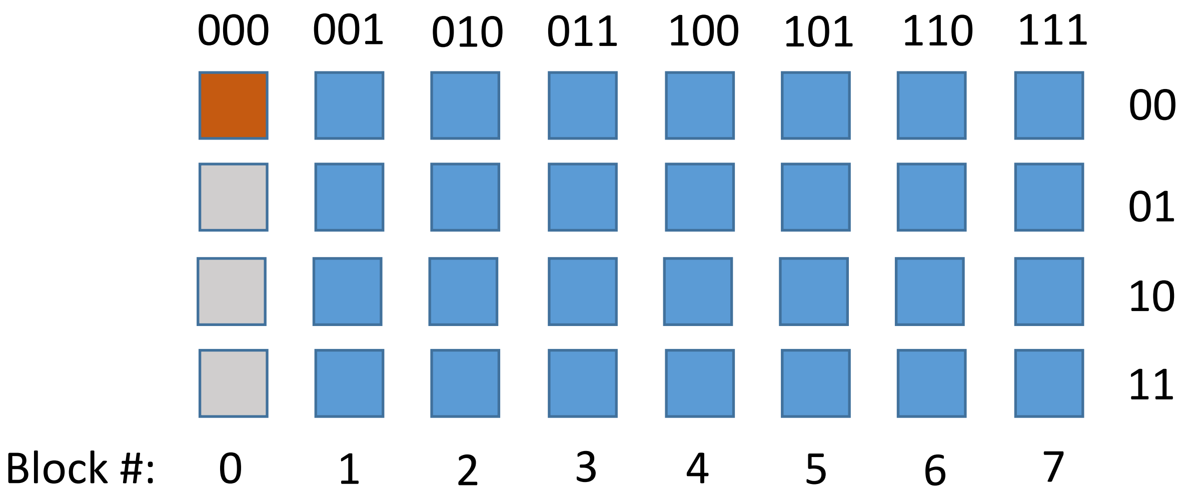
- A **fully associative mapped** cache consists of one set that contains every block of cache memory.



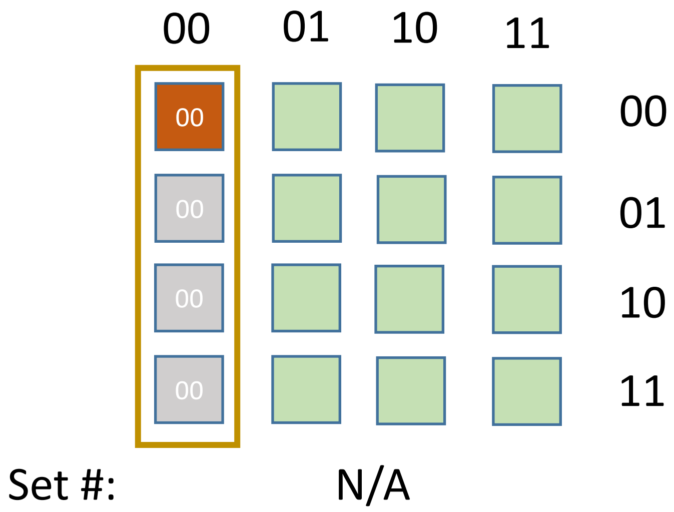
Cache Memory – Fully Associative Set Mapping

- **00000**

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



MAIN MEMORY

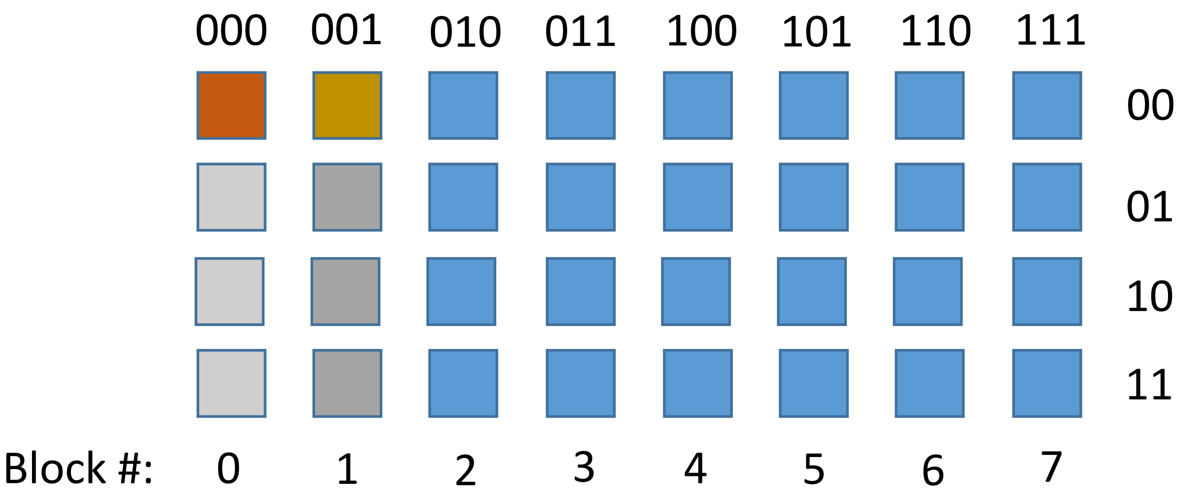


CACHE MEMORY

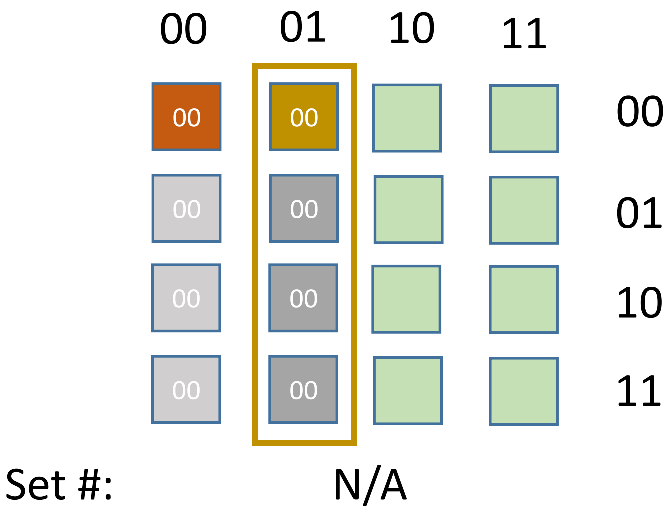
Cache Memory – Fully Associative Set Mapping

- **00100**

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



MAIN MEMORY

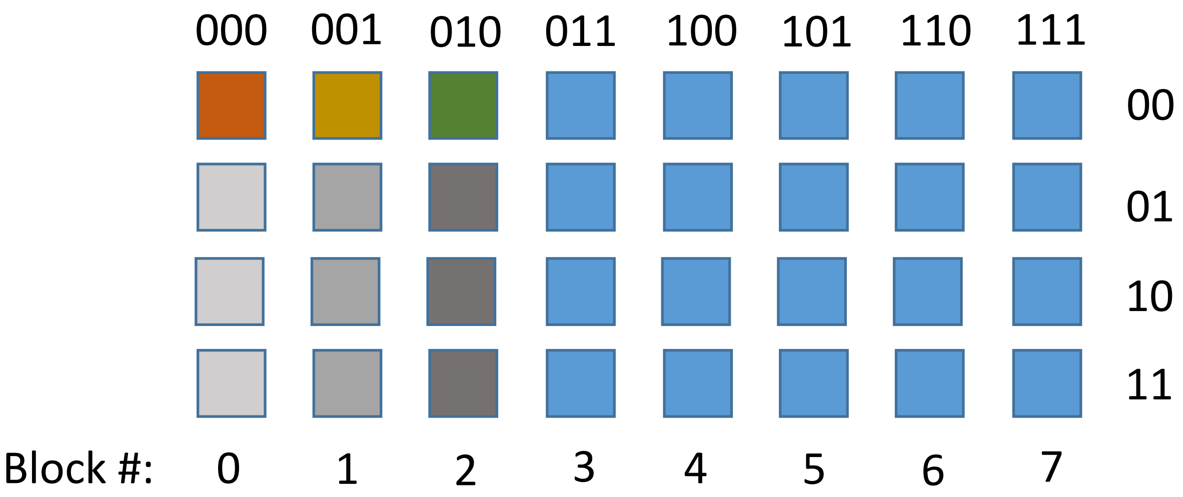


CACHE MEMORY

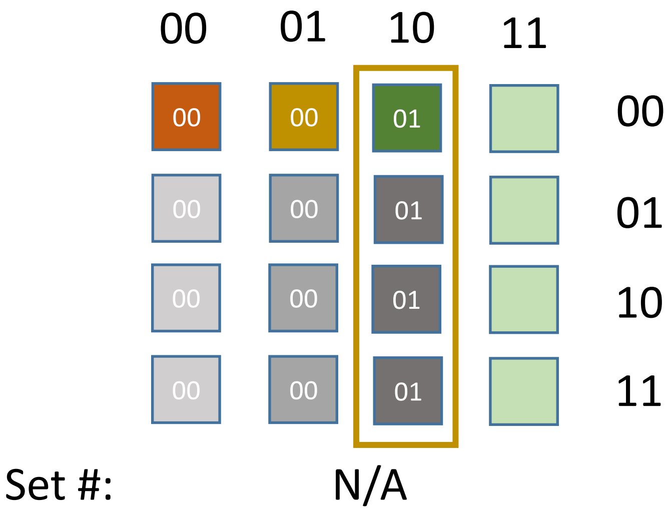
Cache Memory – Fully Associative Set Mapping

- **01000**

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



MAIN MEMORY

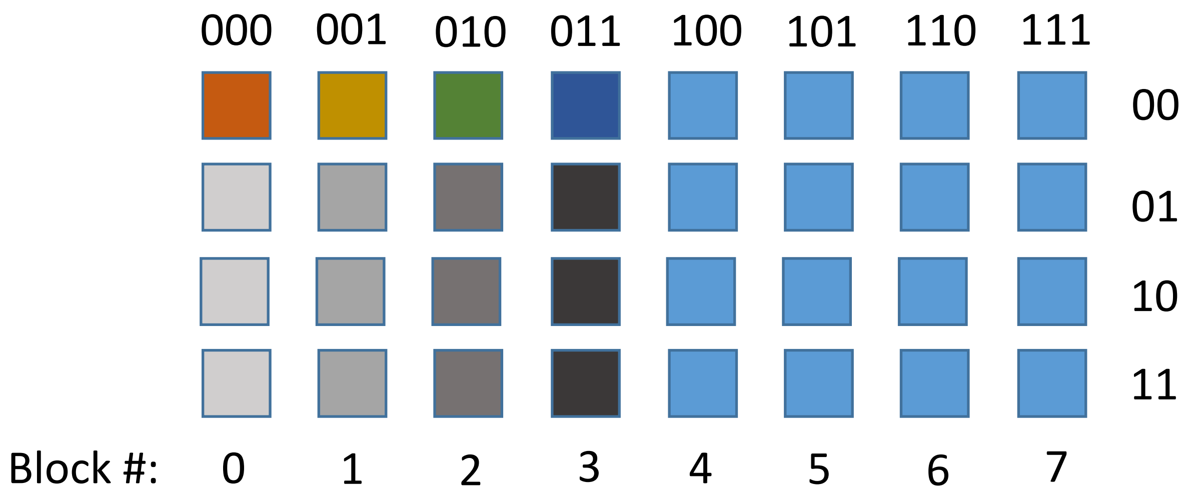


CACHE MEMORY

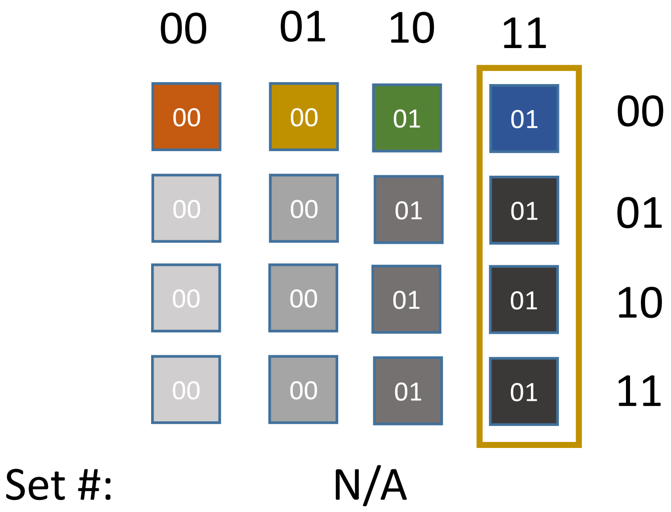
Cache Memory – Fully Associative Set Mapping

- **01100**

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



MAIN MEMORY

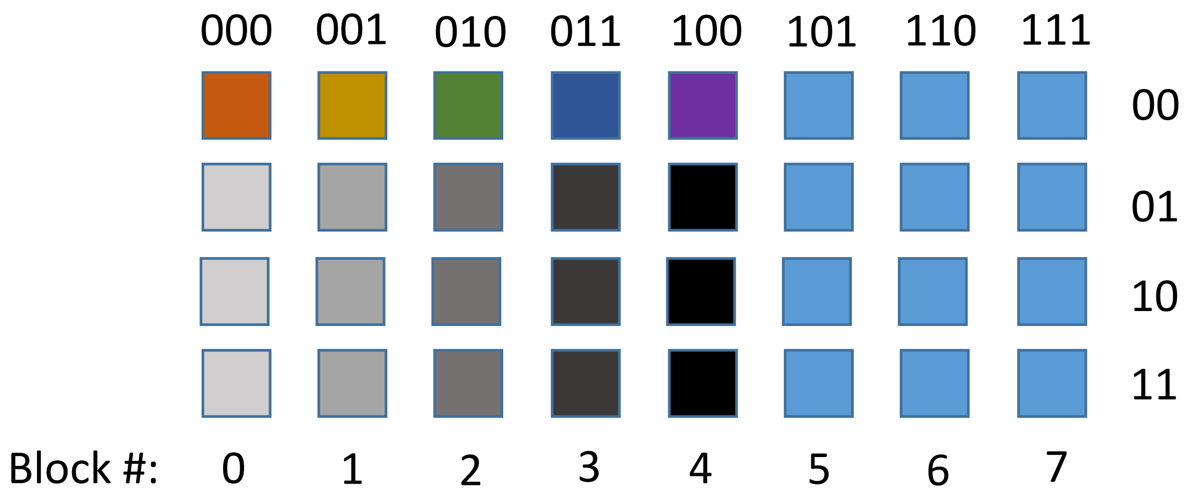


CACHE MEMORY

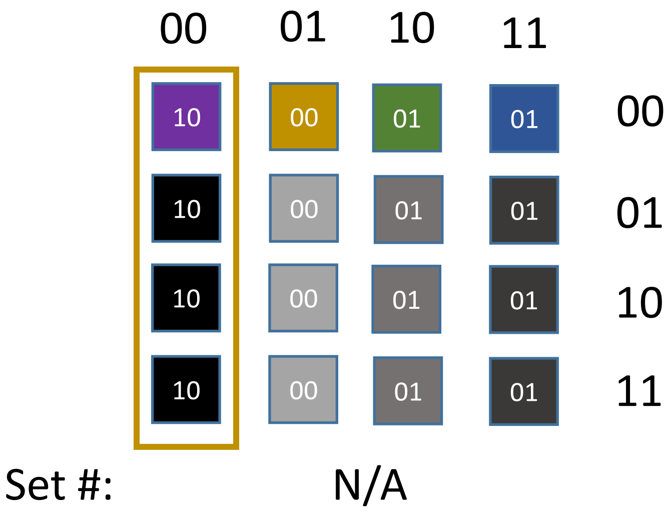
Cache Memory – Fully Associative Set Mapping

- **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		



MAIN MEMORY



00	01	10	11	
10	00	01	01	00
10	00	01	01	01
10	00	01	01	10
10	00	01	01	11

CACHE MEMORY

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