ECE 2300
Digital Logic & Computer Organization

Spring 2025

Caches

Cornell University

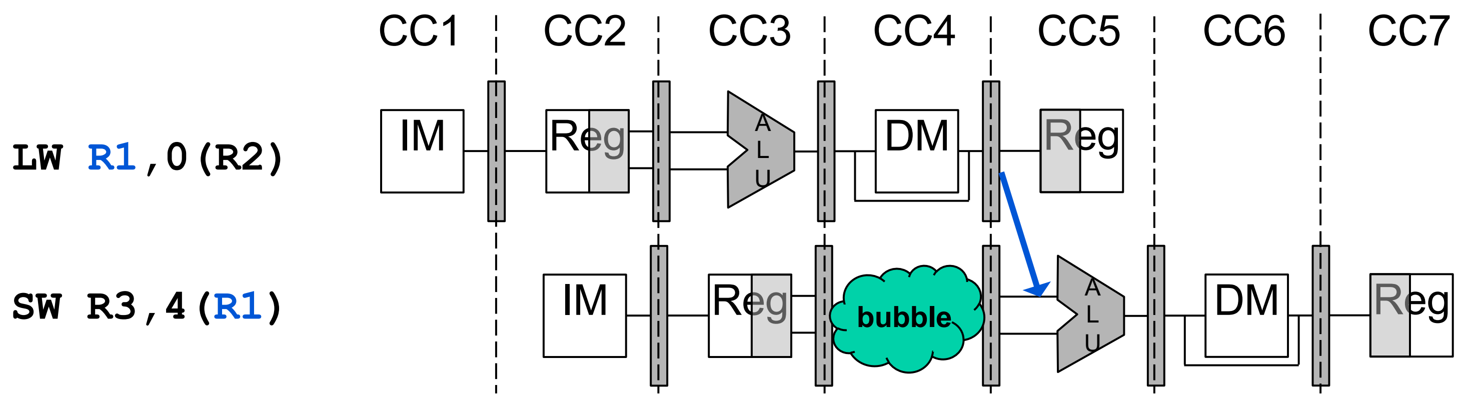
Announcements

- Lab 4A due tomorrow

Review: Data Hazards Requiring Bubbles

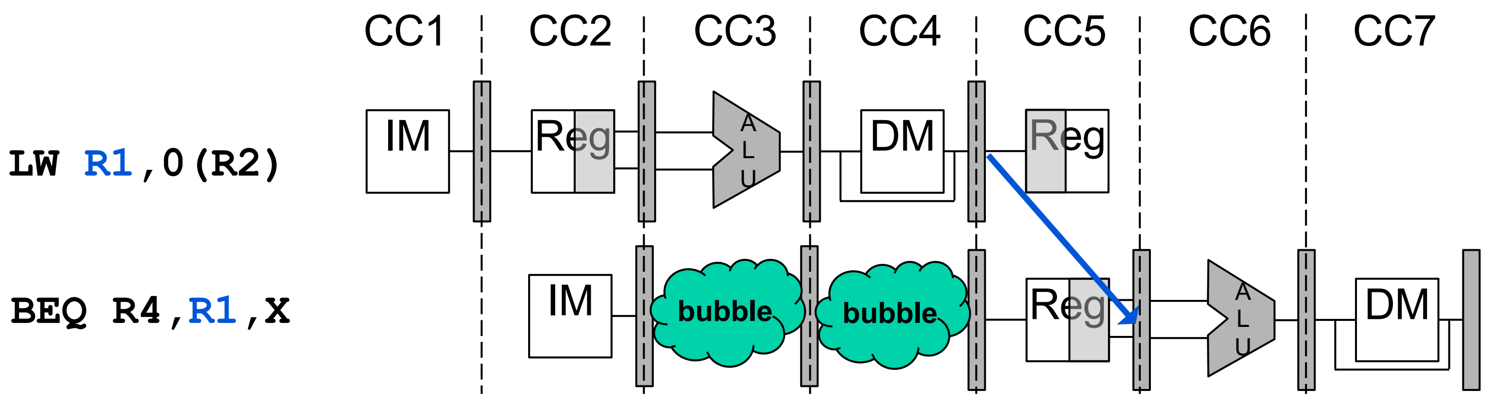
- Occur when instructions are too close together for forwarding to be effective
- Requires adding bubbles in the pipeline
- Data hazard conditions to detect and handle
  - Load followed by R-type
  - Load followed by I-type ALU instruction
  - Load followed by Load
  - Load followed by Store (two cases)
    - Forwarding for R[SA] and R[SB]
  - Load followed by Branch
  - ALU instruction followed by Branch

Load Followed by Store Instruction



- WB→EX forwarding enabled after one bubble inserted for R[SA] of the store instruction

Load Followed by Branch Instruction



Two bubbles needed

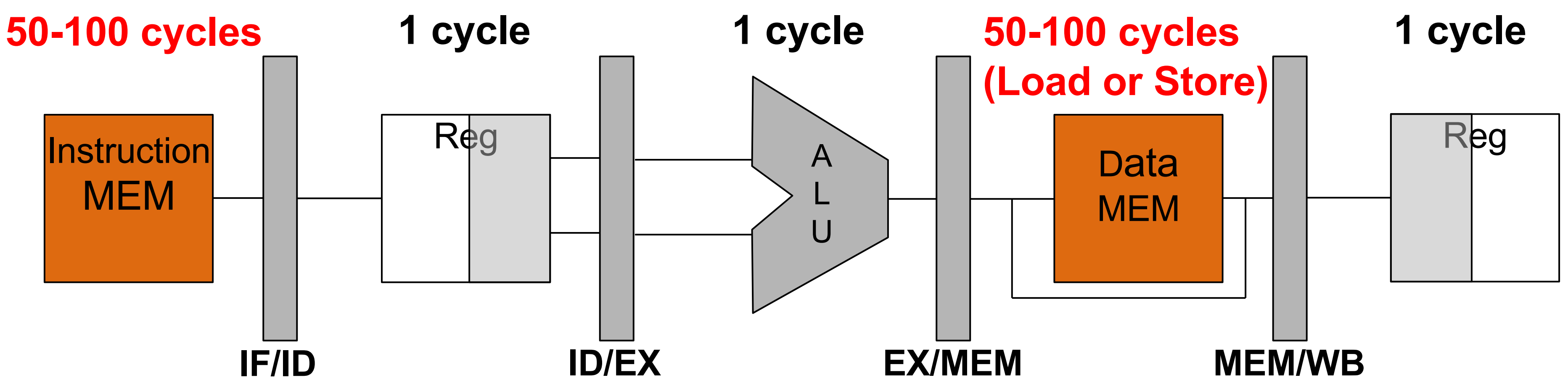
- WB→ID forwarding enabled after two bubbles for the branch instruction

Course Content

- Binary numbers and logic gates
- Boolean algebra and combinational logic
- Sequential logic and state machines
- Binary arithmetic
- Memories
- Instruction set architecture
- Processor organization

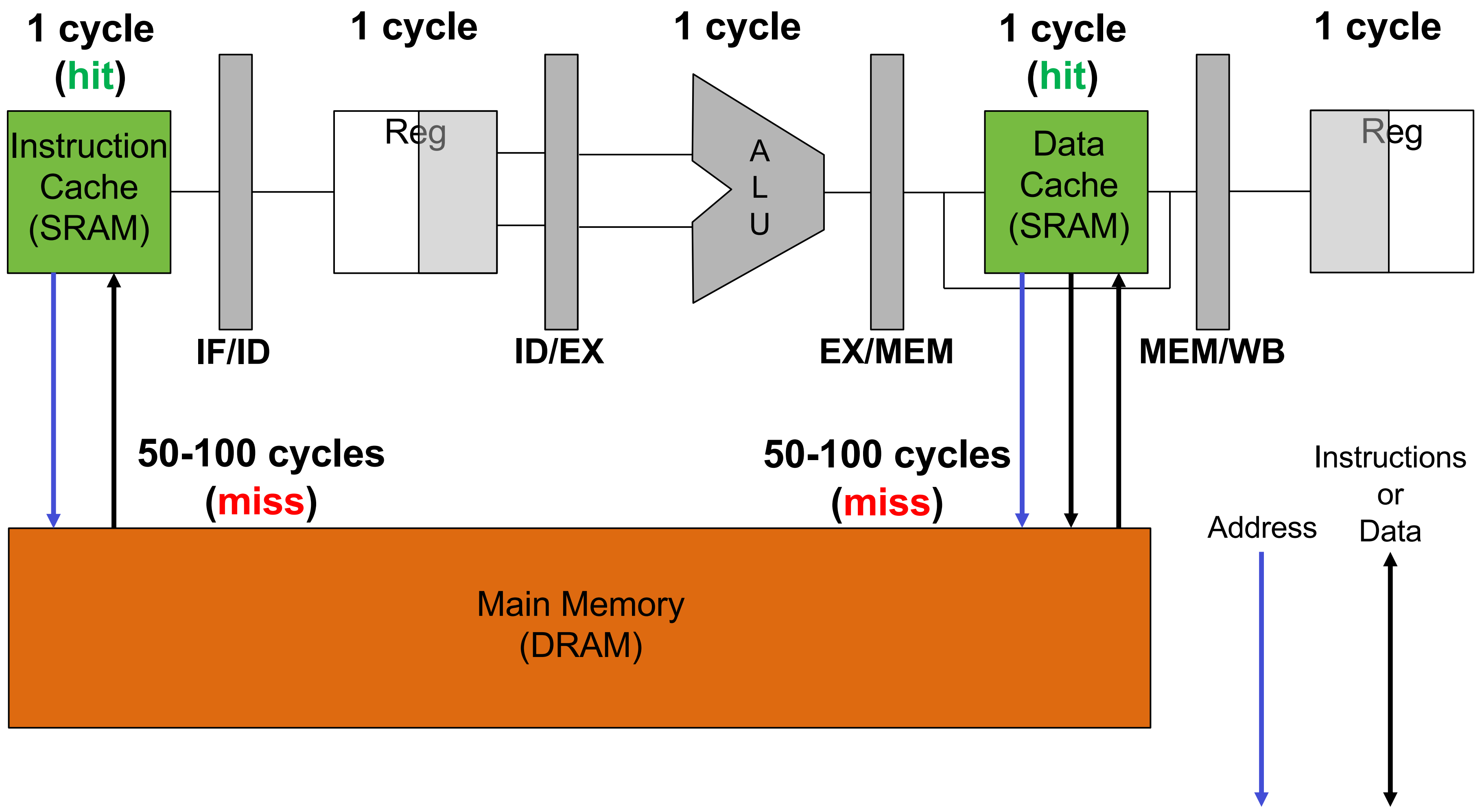
- Caches and virtual memory
- Input/output
- Advanced topics

DRAM is Too Slow for Our Pipeline



- **Typical processor cycle time: 300ps to 2ns (3GHz-500MHz)**
- **DRAM**
  - **Slow to access (10-50 ns for a read or write)**
  - **Cheap (1 transistor per bit cell); High capacity**
- **SRAM**
  - **Fast to access (100's of ps to few ns for a read/write)**
  - **Expensive (6 transistors per bit cell); Low capacity**

Using Caches in the Pipeline



Cache

- Small SRAM memory that permits rapid access to a subset of instructions or data
  - If the data is in the cache (cache hit), we retrieve it without slowing down the pipeline
  - If the data is not in the cache (cache miss), we retrieve it from the main memory (penalty incurred in accessing DRAM)

- The hit rate is the fraction of memory accesses found in the cache
- The miss rate = 1 – hit rate

Memory Access with Cache

- Average memory access time with cache:
  Hit time + Miss rate * Miss penalty

- An example
  - Main memory access time = 50ns
  - Cache hit time = 2ns
  - Miss rate = 10%

Average mem access time w/o cache = 50ns

Average mem access time w/ cache = 2 + 0.1*50 = 7ns

Why Caches Work: Principle of Locality

- Temporal locality
  - If memory location X is accessed, then it is likely to be accessed again in the near future
    - Caches exploit temporal locality by keeping a referenced instruction or data in the cache

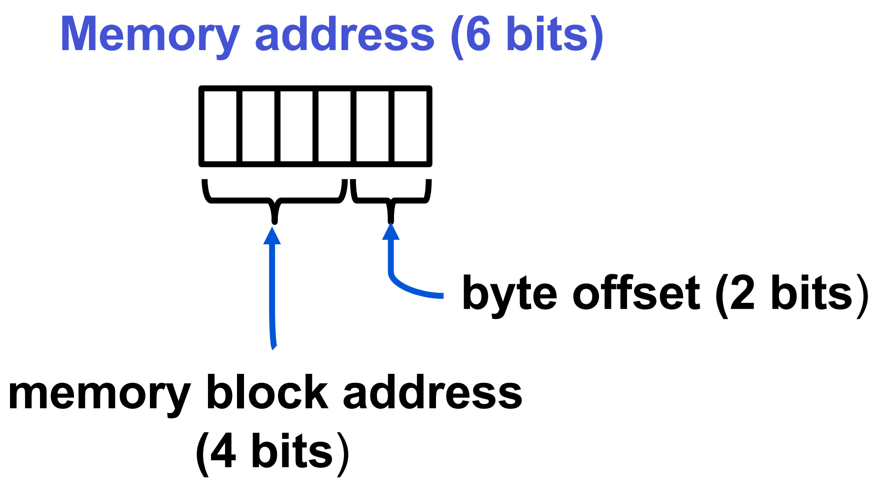
- Spatial locality
  - If memory location X is accessed, then locations near X are likely to be accessed in the near future
    - Caches exploit spatial locality by bringing in a *block* of instructions or data into the cache on a miss

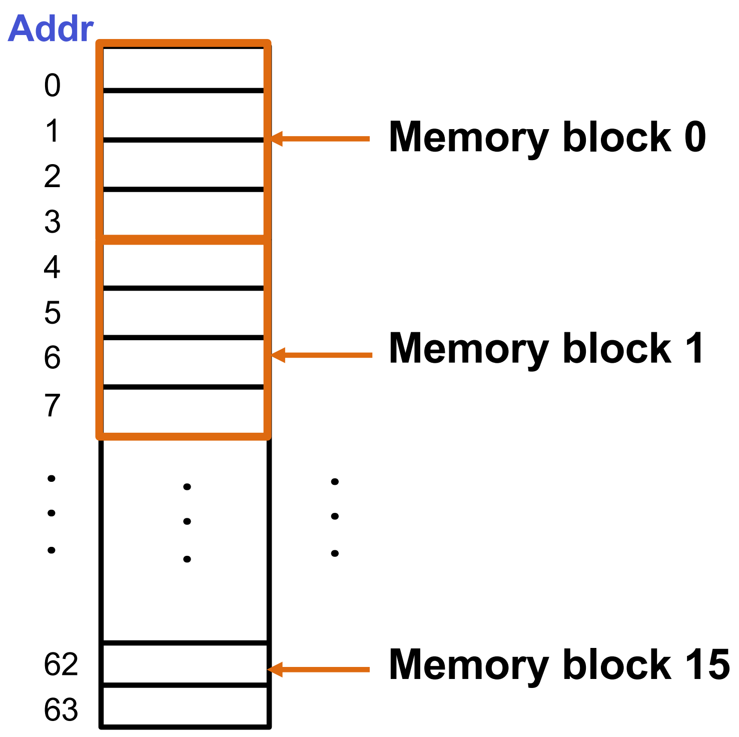
Memory Blocks

- Main memory is partitioned into blocks
  - Each block typically contains *multiple bytes* of data (block size is a power of 2)
  - A whole block is read or written during data transfer between main memory and cache

Memory Block Example

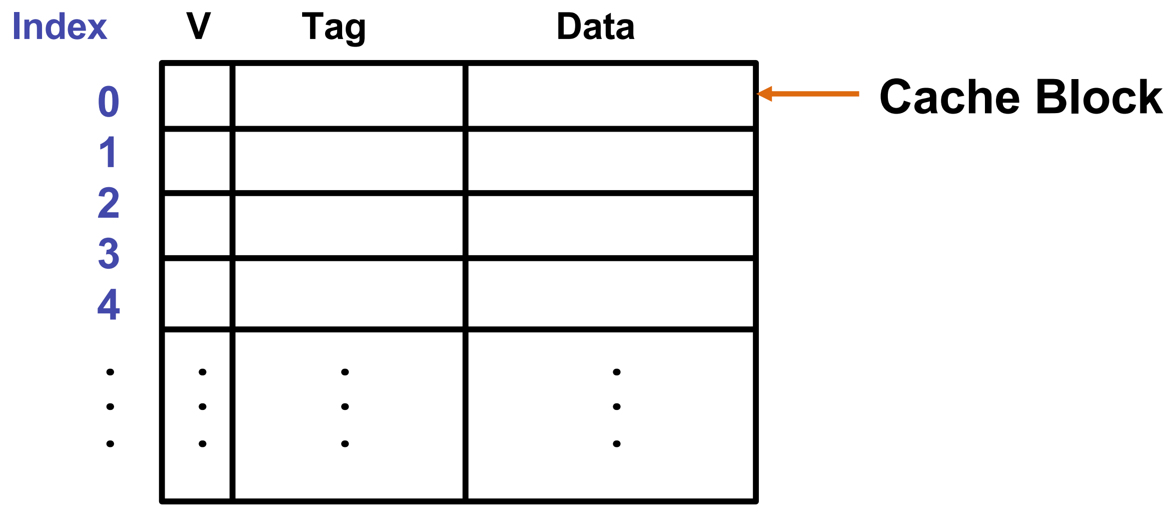
- **Memory address has 6 bits => Memory holds 64 bytes**
- **Size of each block is 4 bytes => Memory holds 16 blocks**
  - Each byte within a block is indexed by a **byte offset**, which is the lowest 2 bits of the memory address (00, 01, 10, 11)





Cache Blocks

- The cache is also divided into blocks, each of which holds data of the same size as a memory block
  - The cache is accessed at the block level using an index for addressing
- Each cache block is associated with a valid bit and a tag
  - *Valid bit (V)*: indicates a cache block is occupied (=1) or not (=0)
  - *Tag*: A unique ID (a portion of memory address) used to identify which memory block occupies the cache block



Cache Intuition

Direct Mapped (DM) Cache Concepts

- Each memory block is mapped to one and only one cache block (many-to-one mapping)

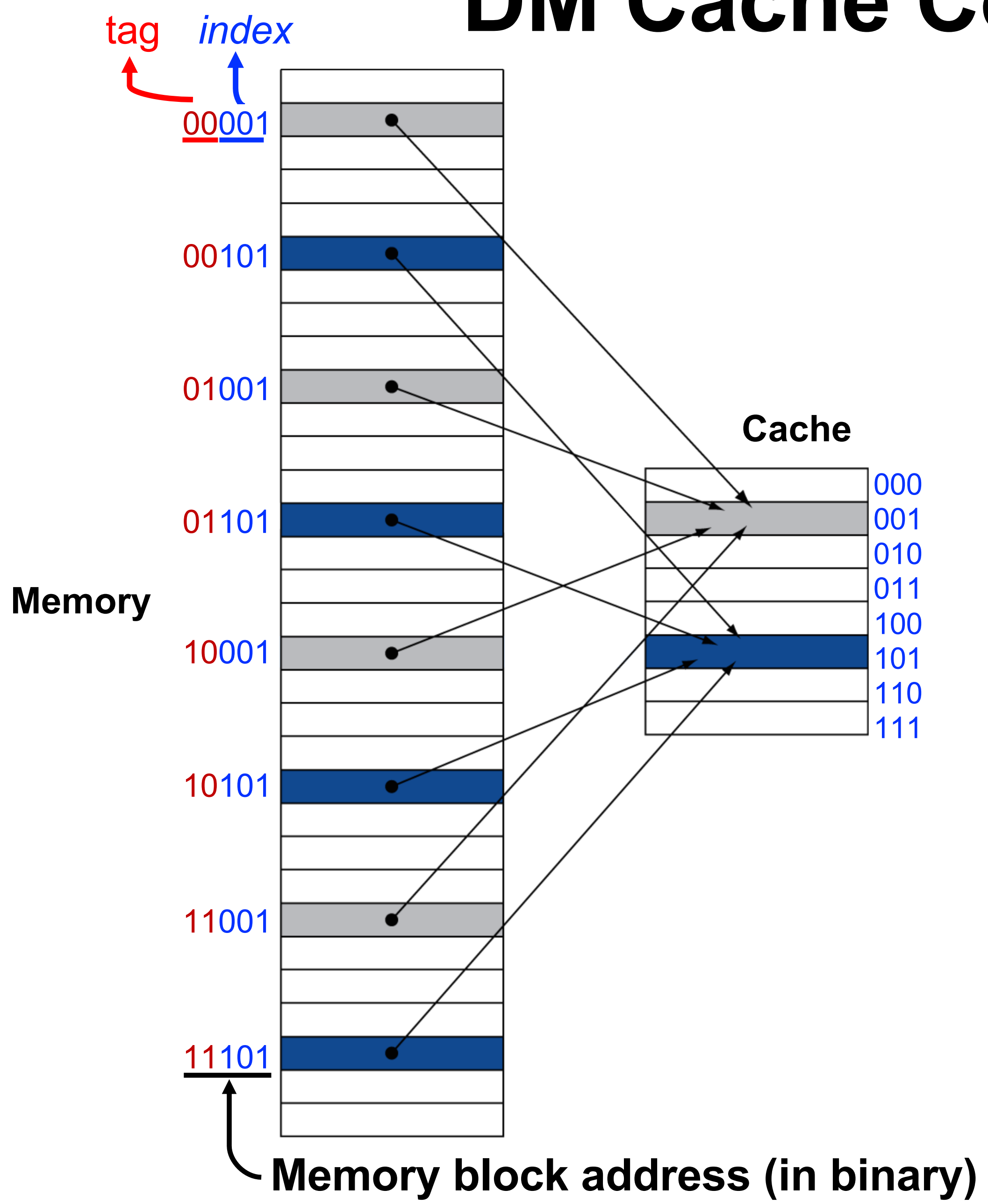
Example:

- A cache with 8 blocks
  – Each cache block has an index (in decimal here)
- Assume the main memory has 32 blocks (4 times larger than cache)
  – Block addresses in decimal

Memory Block (block address)	Cache Block (index)
0, 8, 16, 24	0
1, 9, 17, 25	1
2, 10, 18, 26	2
3, 11, 19, 27	3
4, 12, 20, 28	4
5, 13, 21, 29	5
6, 14, 22, 30	6
7, 15, 23, 31	7

4 different memory blocks are mapped to same cache block in this example

DM Cache Concepts

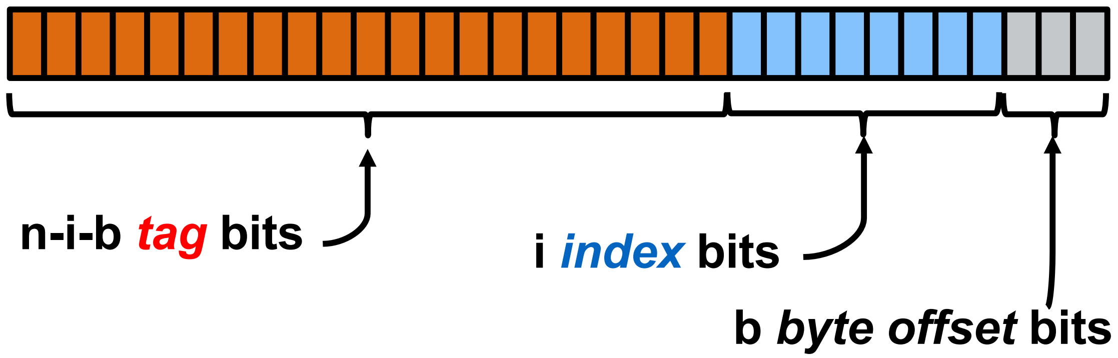


Same Example:

- 8 blocks in cache
  - Cache block indices in binary
- 32 blocks in main memory
  - Memory block addresses in binary
- 4 different memory blocks mapped to the same cache location
  - Last 3 bits of the memory block address used for *indexing* the cache block
  - Remaining 2 bits are tags

Address Translation for DM Cache

- Breakdown of n-bit memory address for cache use

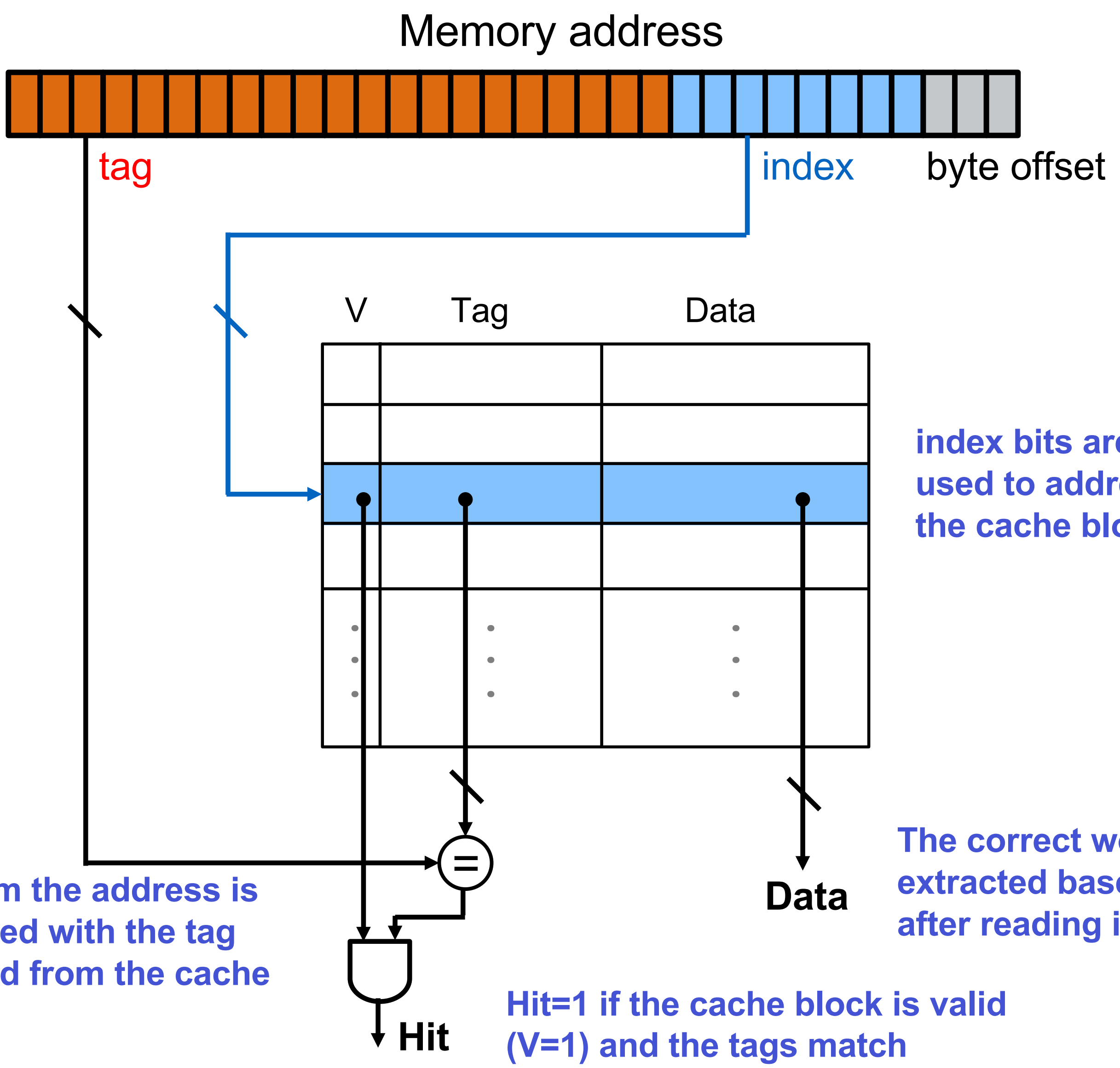


Main memory is byte addressable

- ***tag*** **together with** ***index*** **form the** ***memory block address*** **with (n-b) bits**
- ***memory block address*** **and** ***byte offset*** **make up the complete address**

- DM cache parameters
  - Number of cache blocks is 2^i
    - index bits are used to address the cache blocks
  - Size of each cache block is 2^b bytes
    - "cache block" and "cache line" are synonymous
  - Total cache size is $2^i \times 2^b = 2^{i+b}$ bytes

DM Cache Organization



index bits are used to address the cache blocks

The correct word or byte will be extracted based on the byte offset after reading it

Tag from the address is compared with the tag retrieved from the cache

Hit=1 if the cache block is valid (V=1) and the tags match

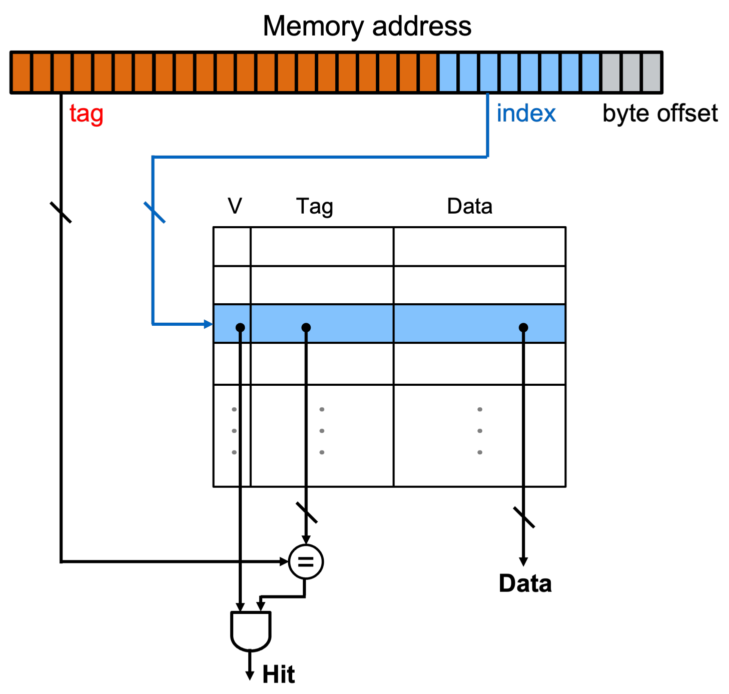
Reading DM Cache

- Use the index bits to address the cache and retrieve the tag, data, and valid bit

- Compare the tag from the address with the retrieved tag

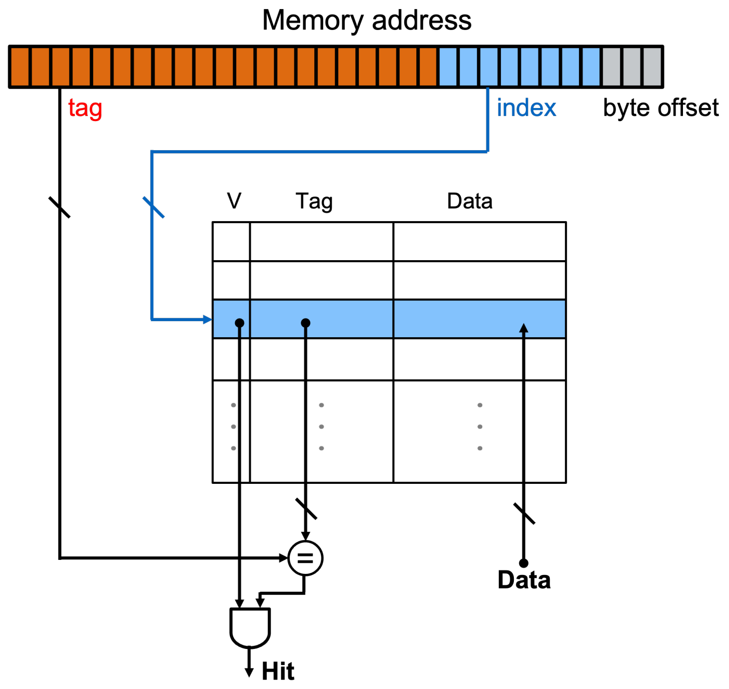
- If valid & a match in tag (hit), select the desired data using the byte offset

- Otherwise (miss)
  - Bring the memory block into the cache (also set valid=1)
  - Store the tag from the address associated with the memory block
  - Select the desired data using the byte offset



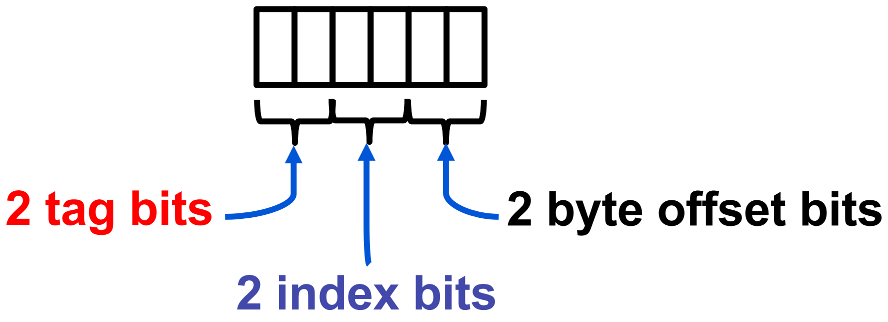
Writing DM Cache

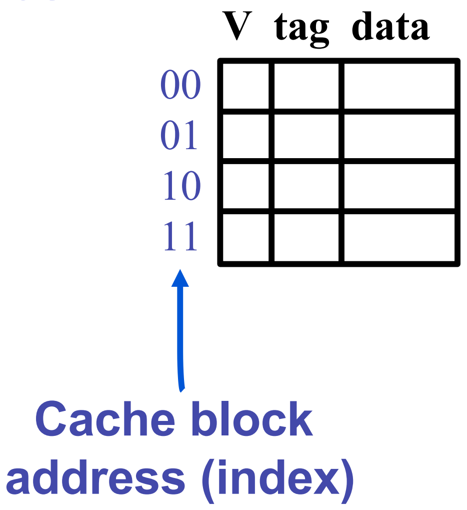
- Use the index bits to address the cache and retrieve the tag and valid bit
- Compare the tag from the address with the retrieved tag
- If valid & a match in tag (hit), write the data into the cache location
- Otherwise (miss), one option
  - Bring the memory block into the cache (also set valid)
  - Store the tag from the address associated with the memory block
  - Write the data into the cache location



Direct Mapped Cache Example

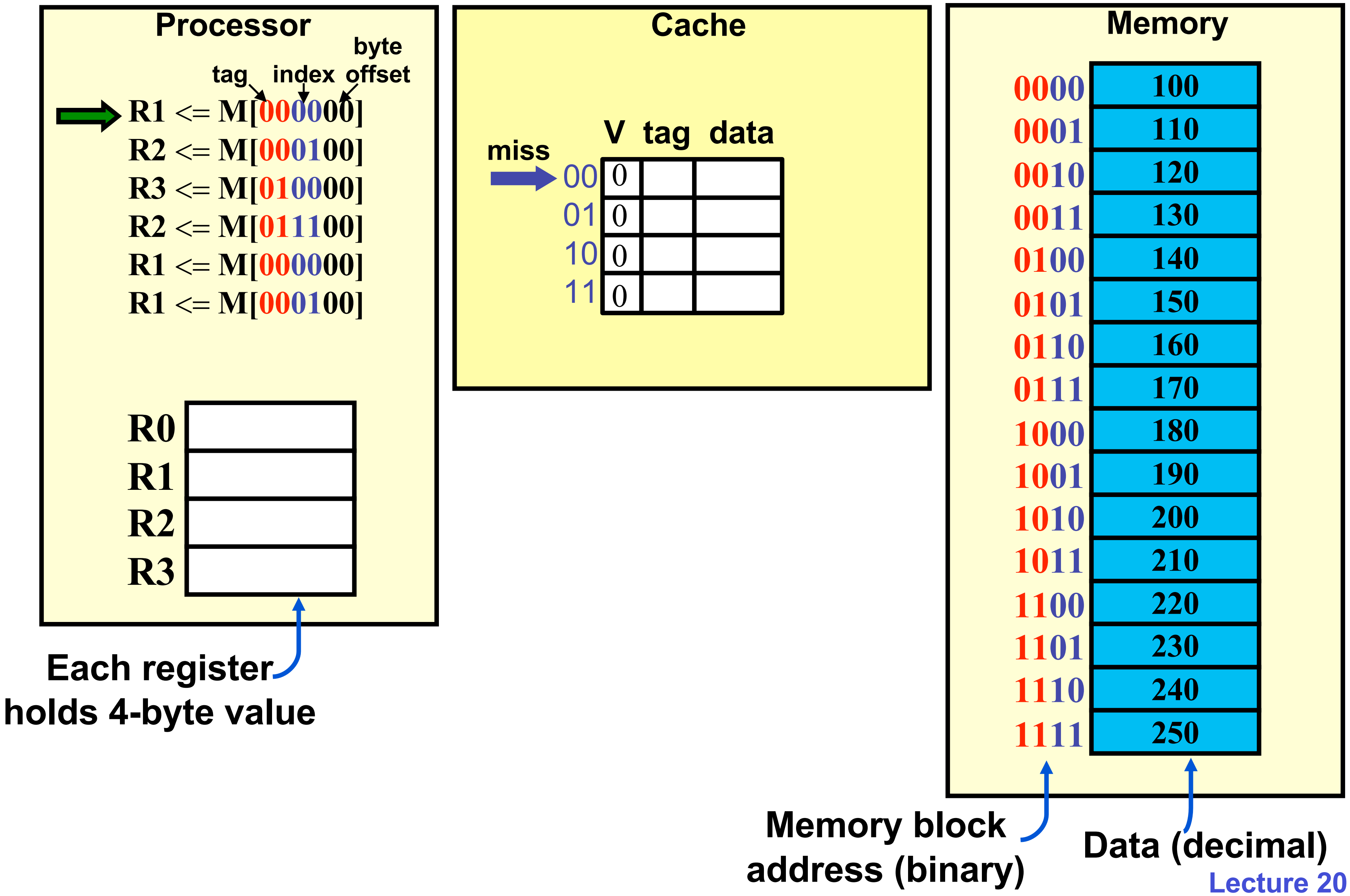
- Size of each block is 4 bytes
- Cache holds 4 blocks
- Memory holds 16 blocks
- Memory address has 6 bits



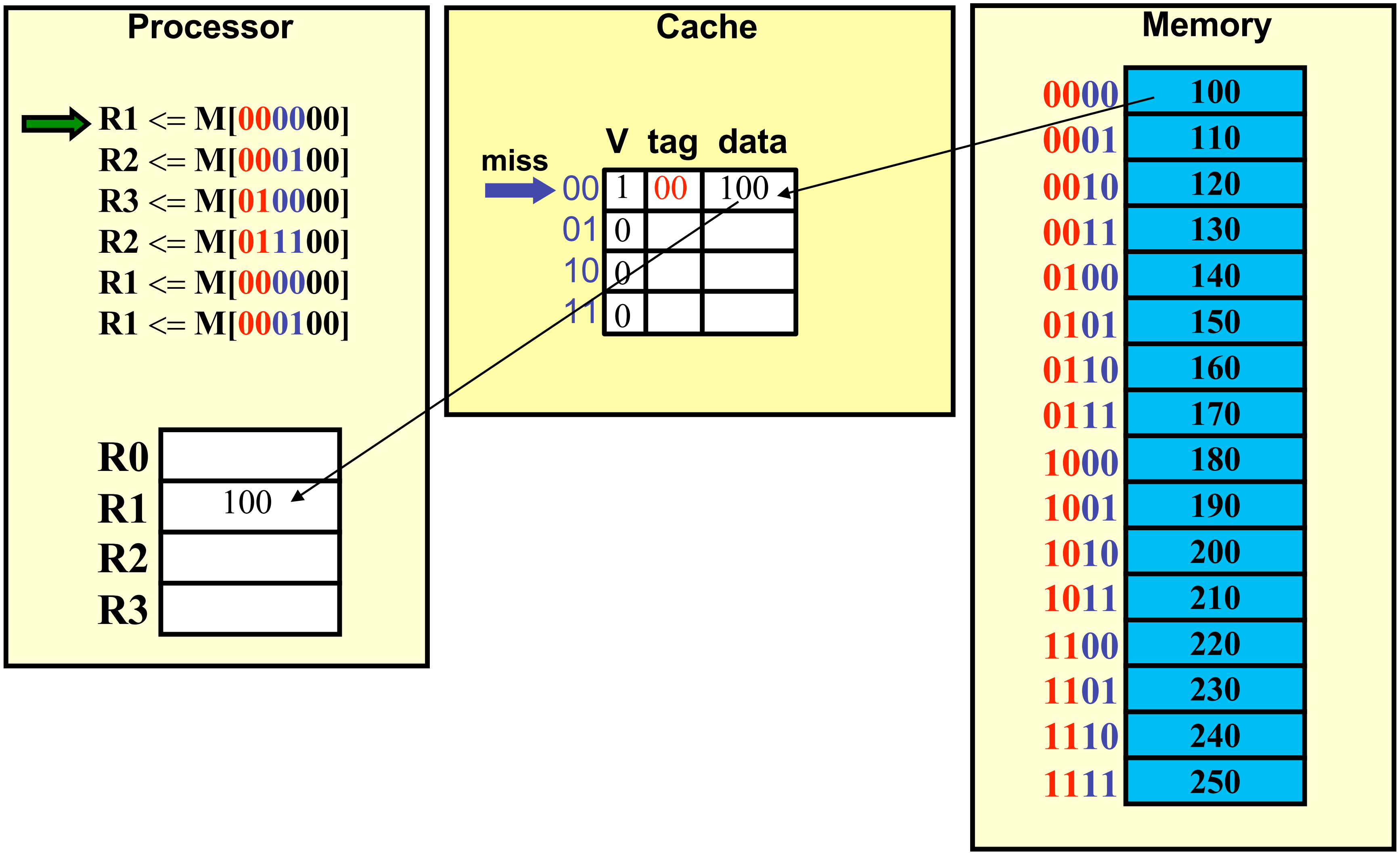


{ tag, index } = memory block address

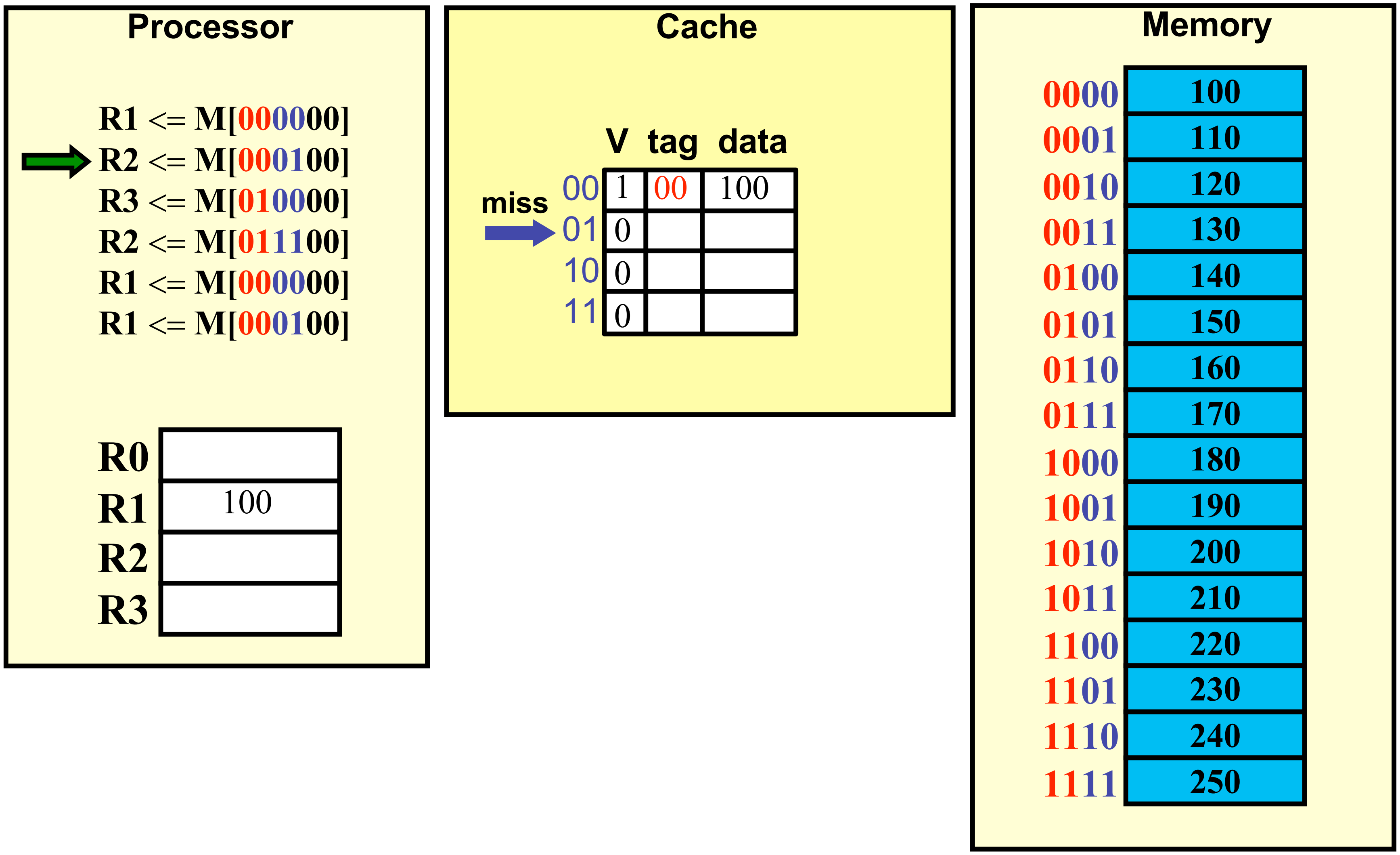
Direct Mapped Cache Example



Direct Mapped Cache Example



Direct Mapped Cache Example



Processor

R1 <= M[000000]
➡ R2 <= M[000100]
R3 <= M[010000]
R2 <= M[011100]
R1 <= M[000000]
R1 <= M[000100]

R0	
R1	100
R2	
R3	

Cache

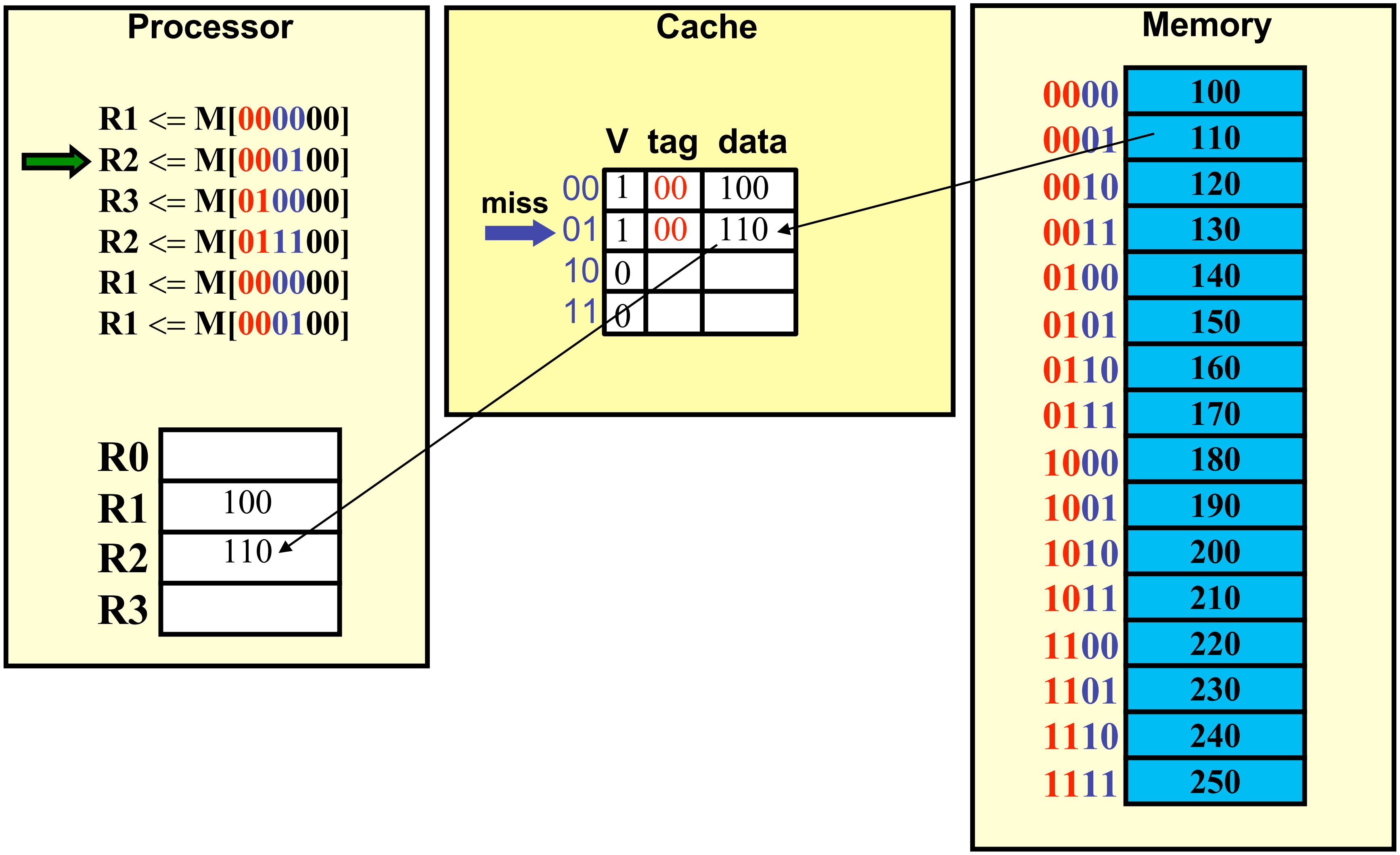
miss ➡ 01

	V	tag	data
00	1	00	100
01	0		
10	0		
11	0		

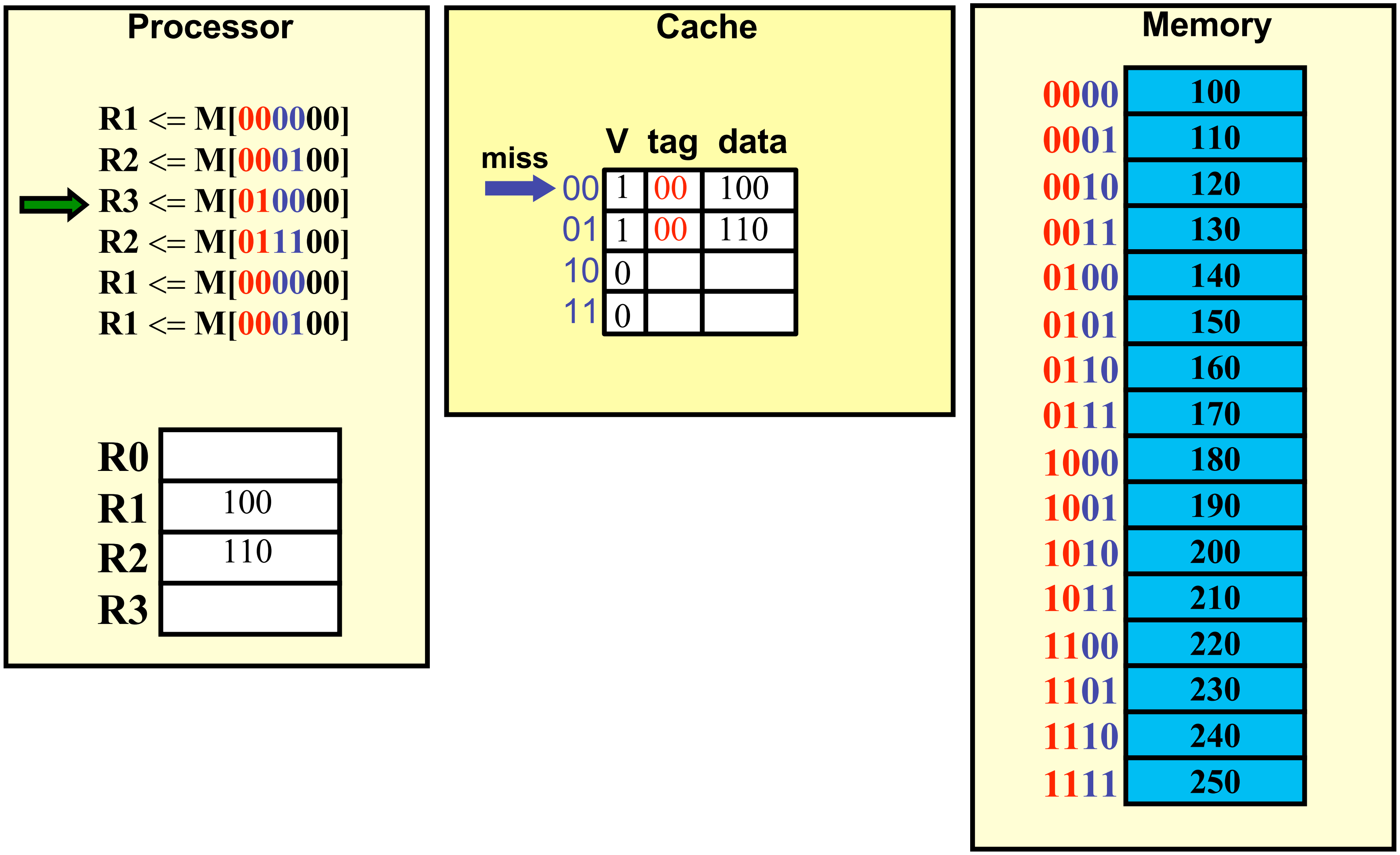
Memory

0000	100
0001	110
0010	120
0011	130
0100	140
0101	150
0110	160
0111	170
1000	180
1001	190
1010	200
1011	210
1100	220
1101	230
1110	240
1111	250

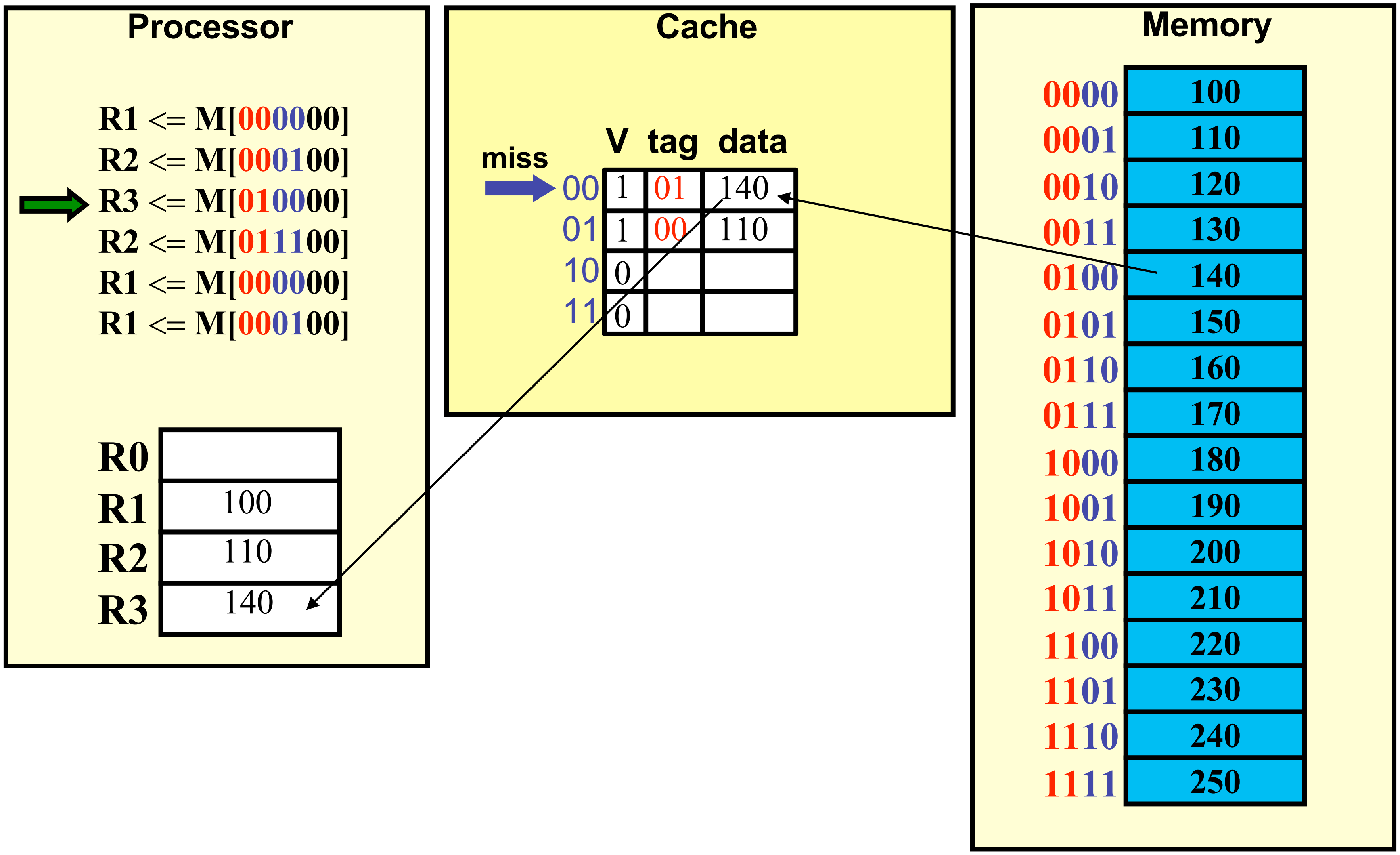
Direct Mapped Cache Example



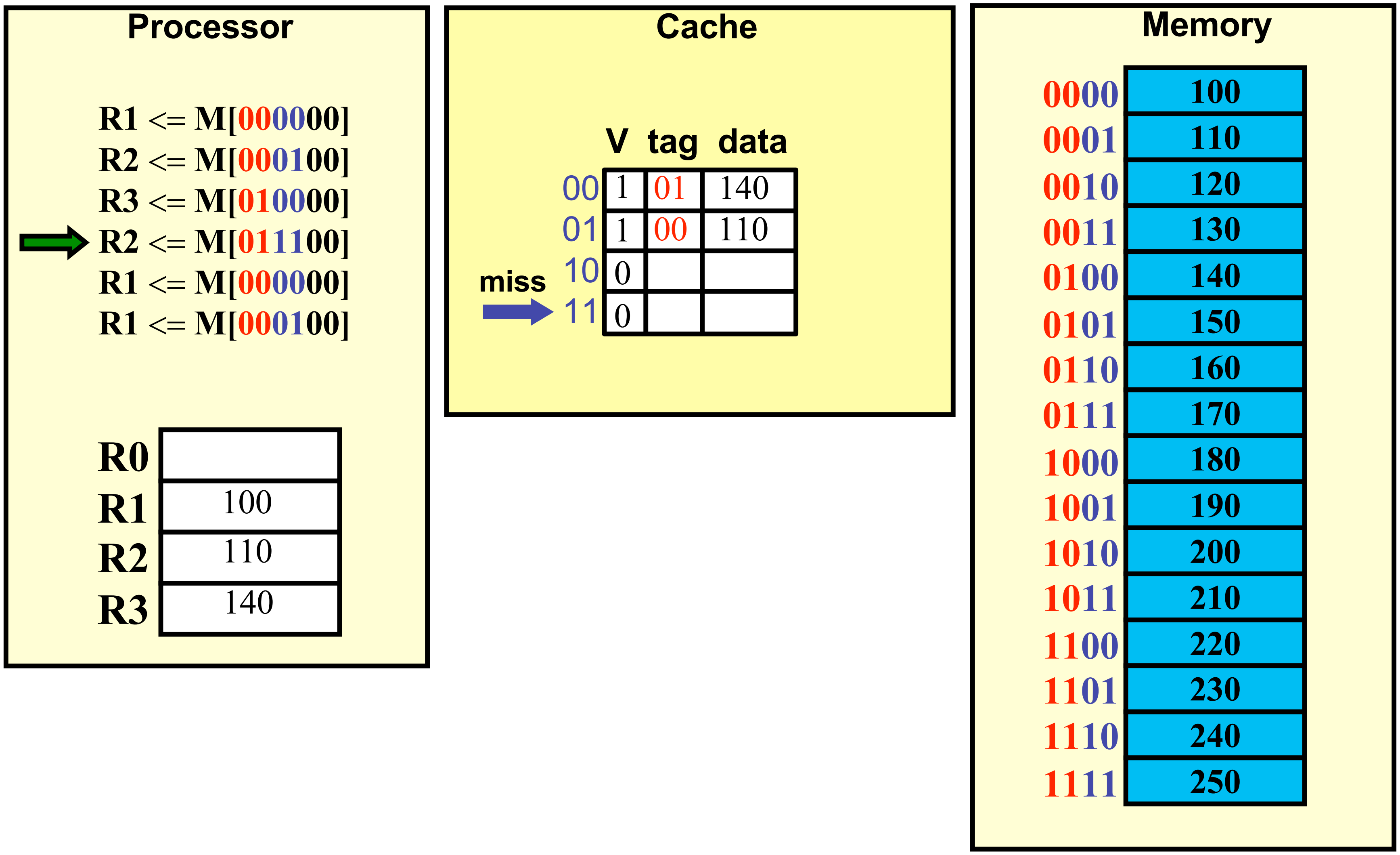
Direct Mapped Cache Example



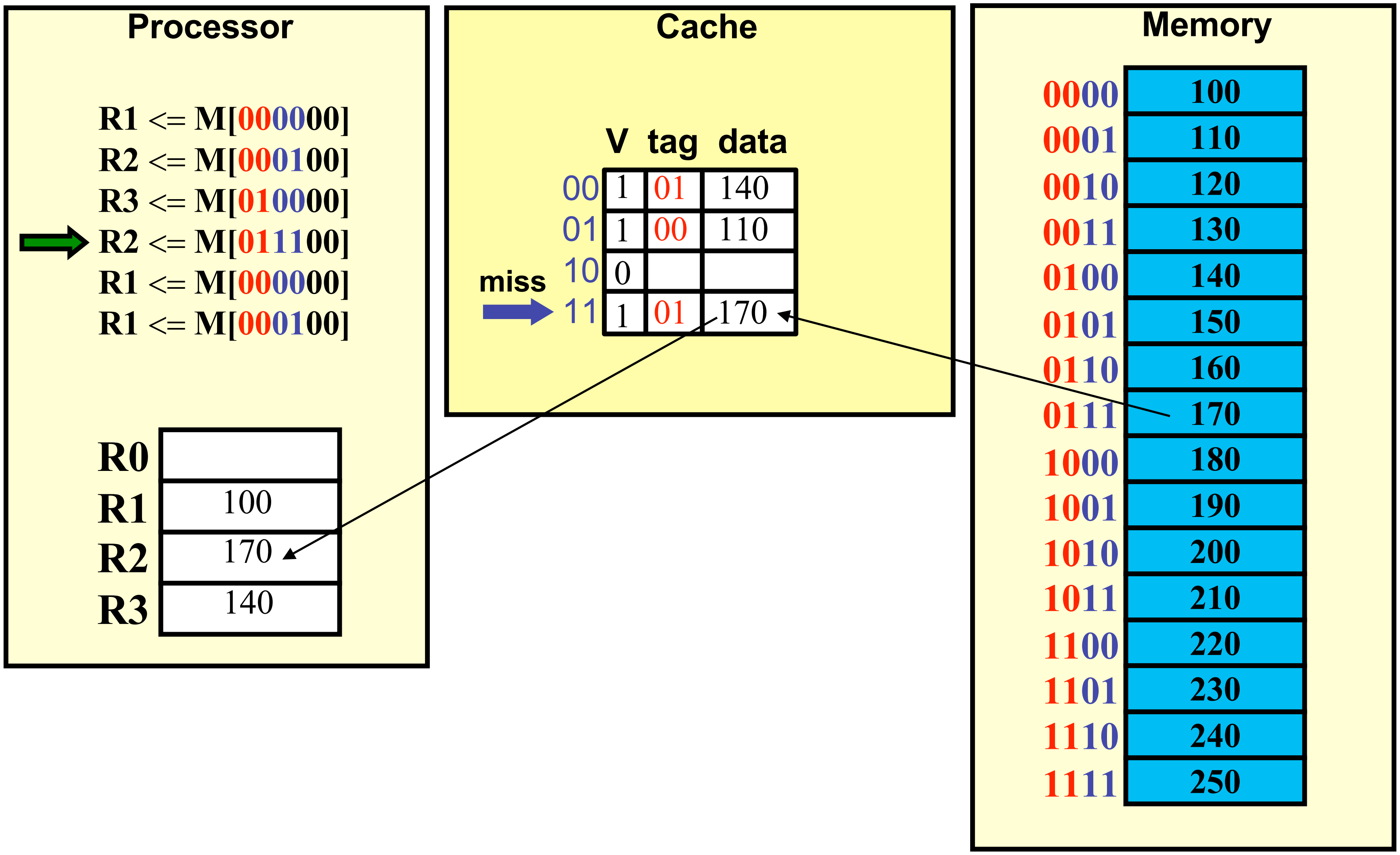
Direct Mapped Cache Example



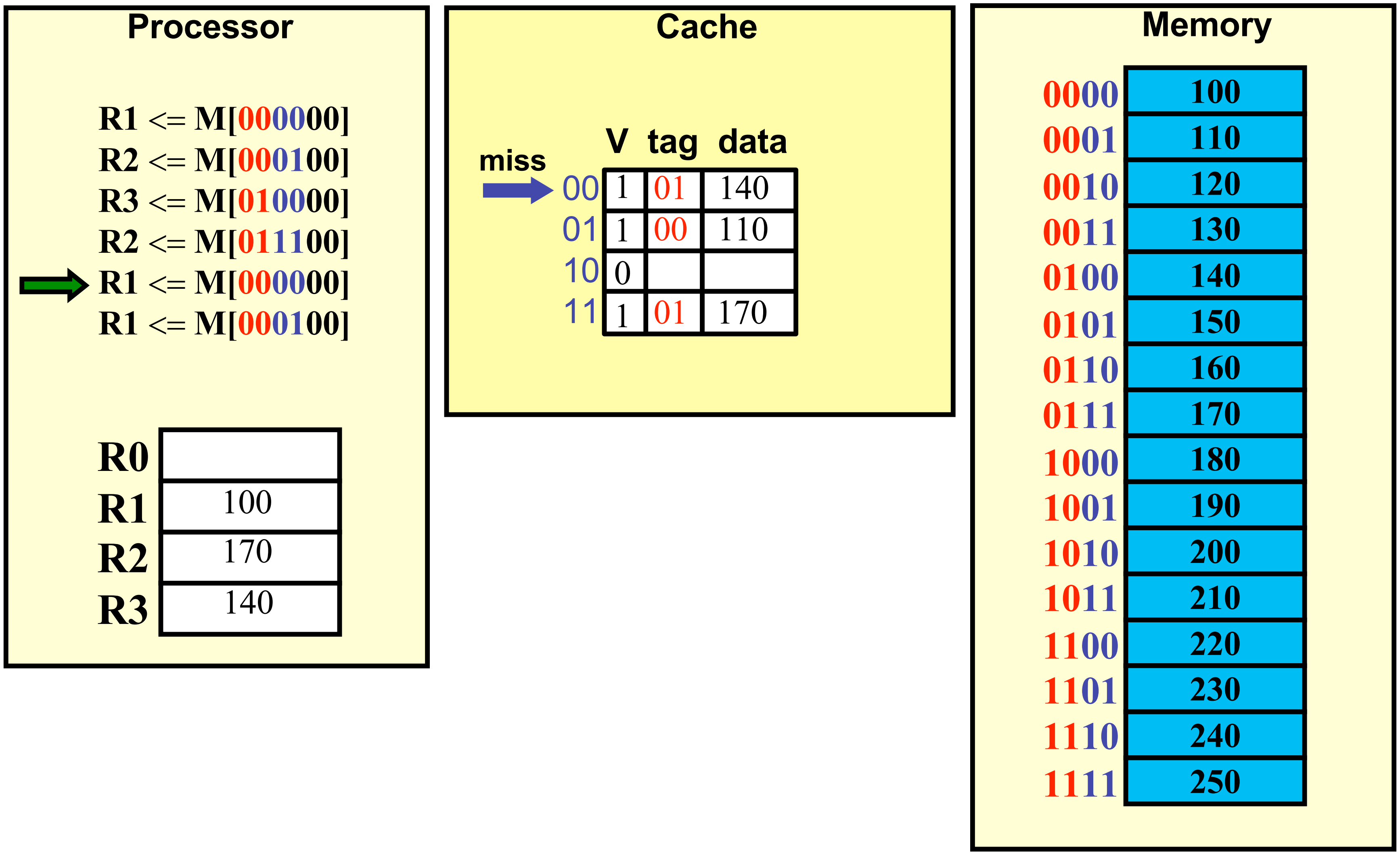
Direct Mapped Cache Example



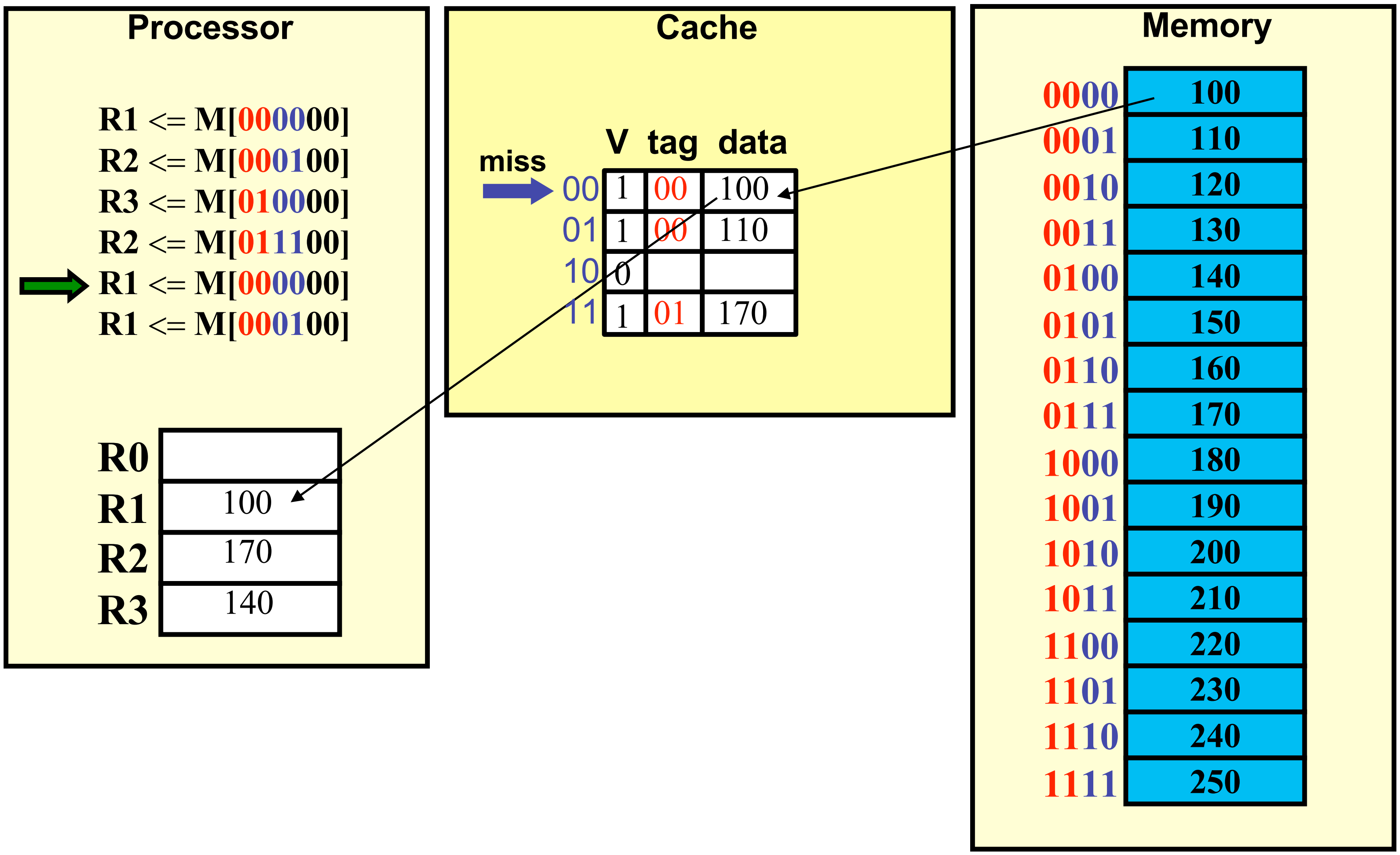
Direct Mapped Cache Example



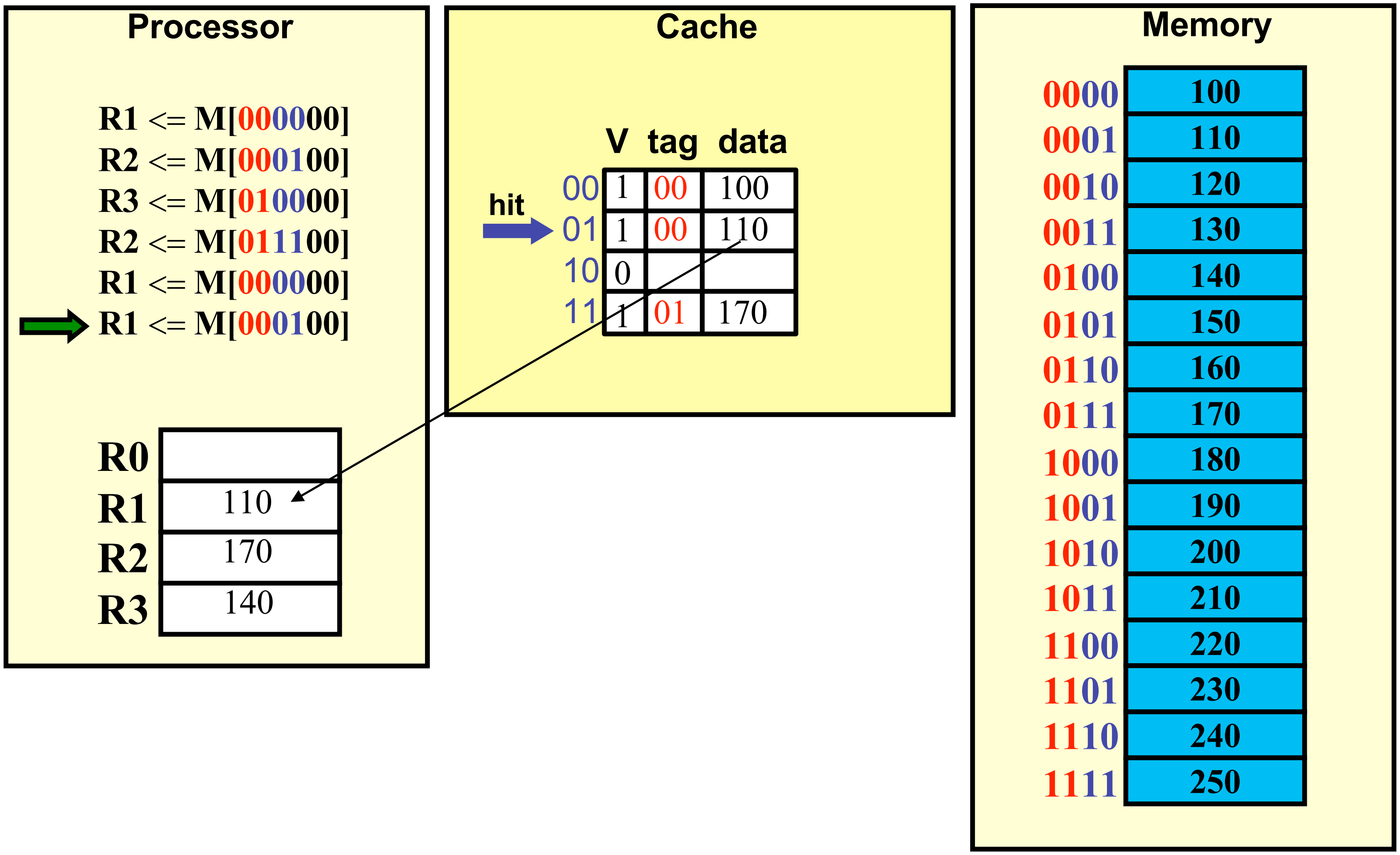
Direct Mapped Cache Example



Direct Mapped Cache Example



Direct Mapped Cache Example



Next Class

More Caches