ECE 2300
Digital Logic & Computer Organization

Fall 2016

More Caches

Cornell University

Associative Caches

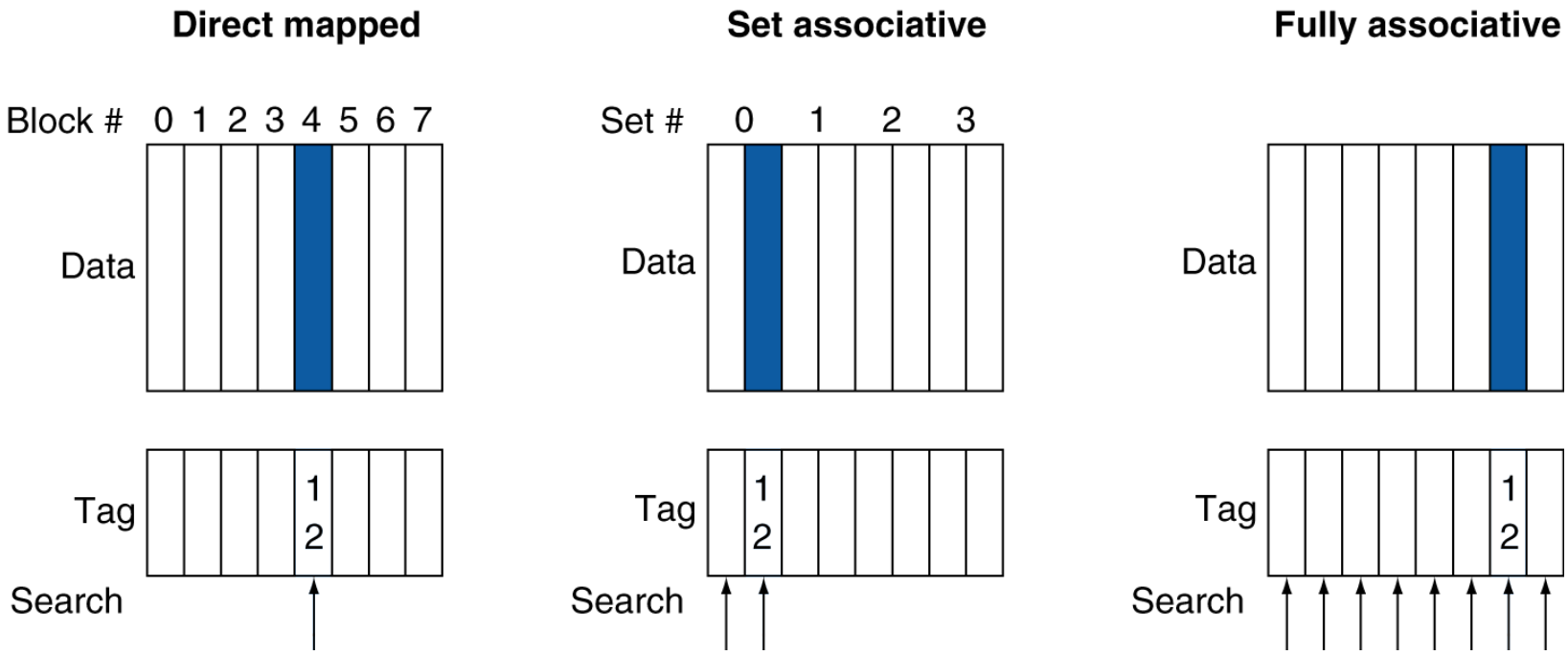
- Provide more flexible placement of blocks
- Each *set* in the cache contains >1 block
  - A block can be stored anywhere in the set
  - A set is searched in parallel to find a block
- Increasing the associatively...
  - Decreases the miss rate (fewer conflicts)
  - Increases the hit time (takes longer to search)

Associative Caches

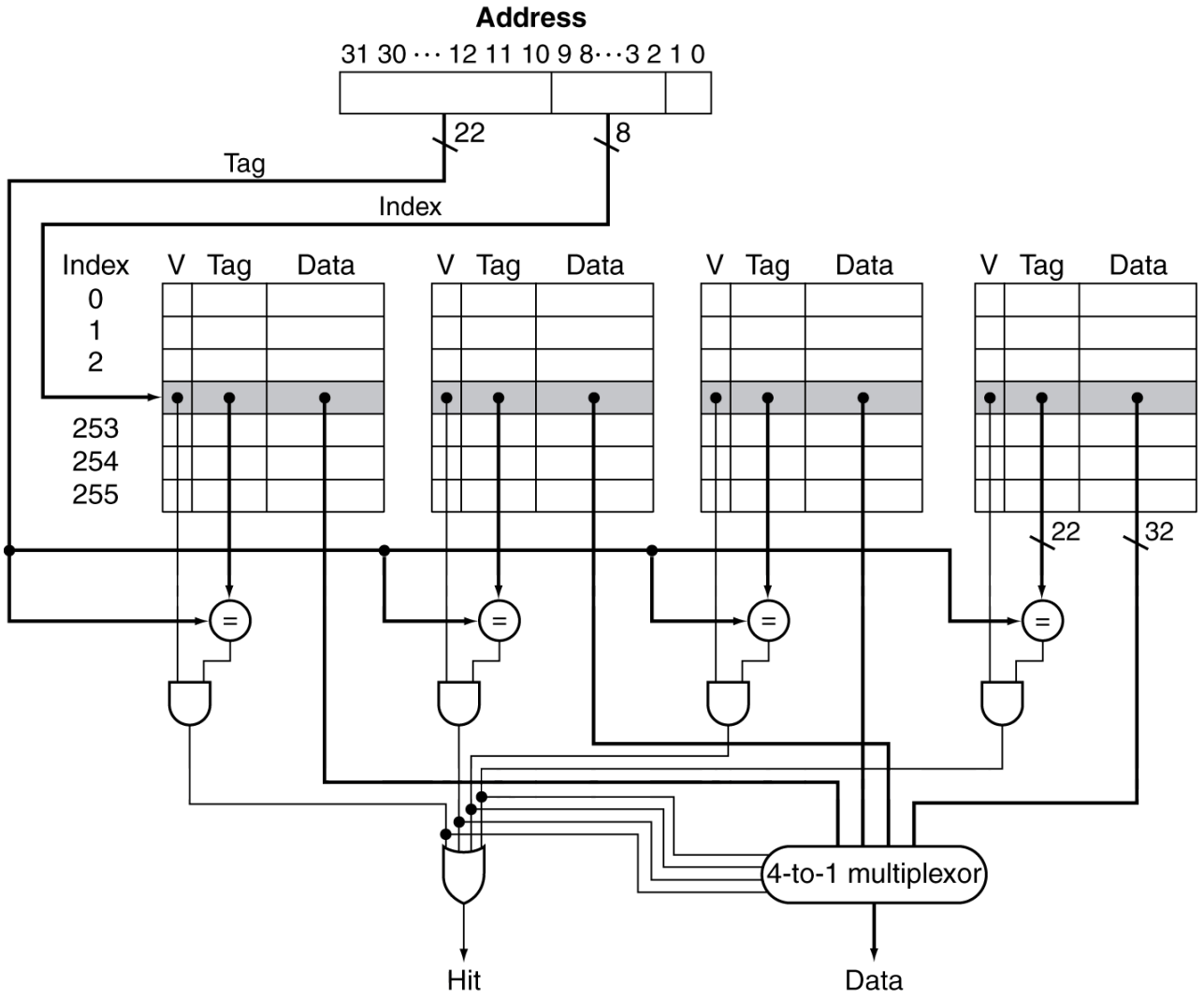
- **Fully associative**
  - Block can go in any cache location
  - All entries are searched at once
  - Comparator per entry (expensive)

- ***n*-way set associative**
  - Each set contains *n* entries (*ways*)
  - Block number determines which set
  - All ways in the selected set are searched at once
  - *n* comparators (less expensive)

Associative Cache Examples

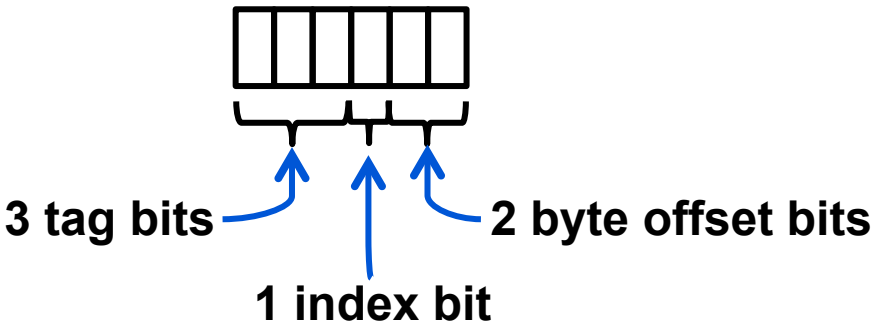


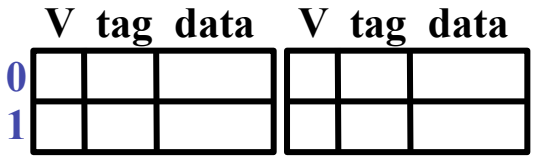
4-way Set Associative Cache



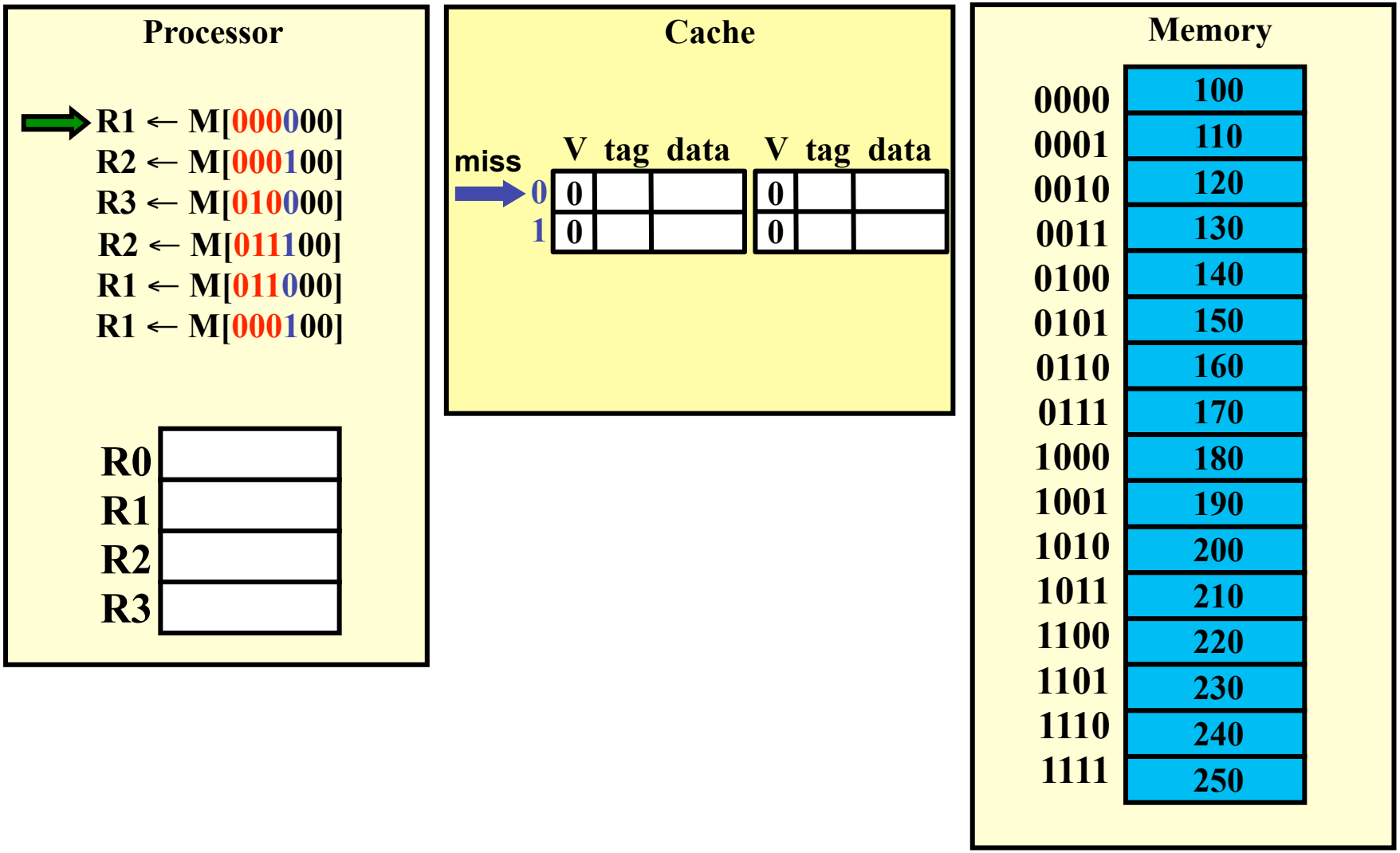
2-way Set Associative Example

- Size of each block is 4 bytes
- Cache holds 4 blocks, 2-way set associative
- Memory holds 16 blocks
- Memory address

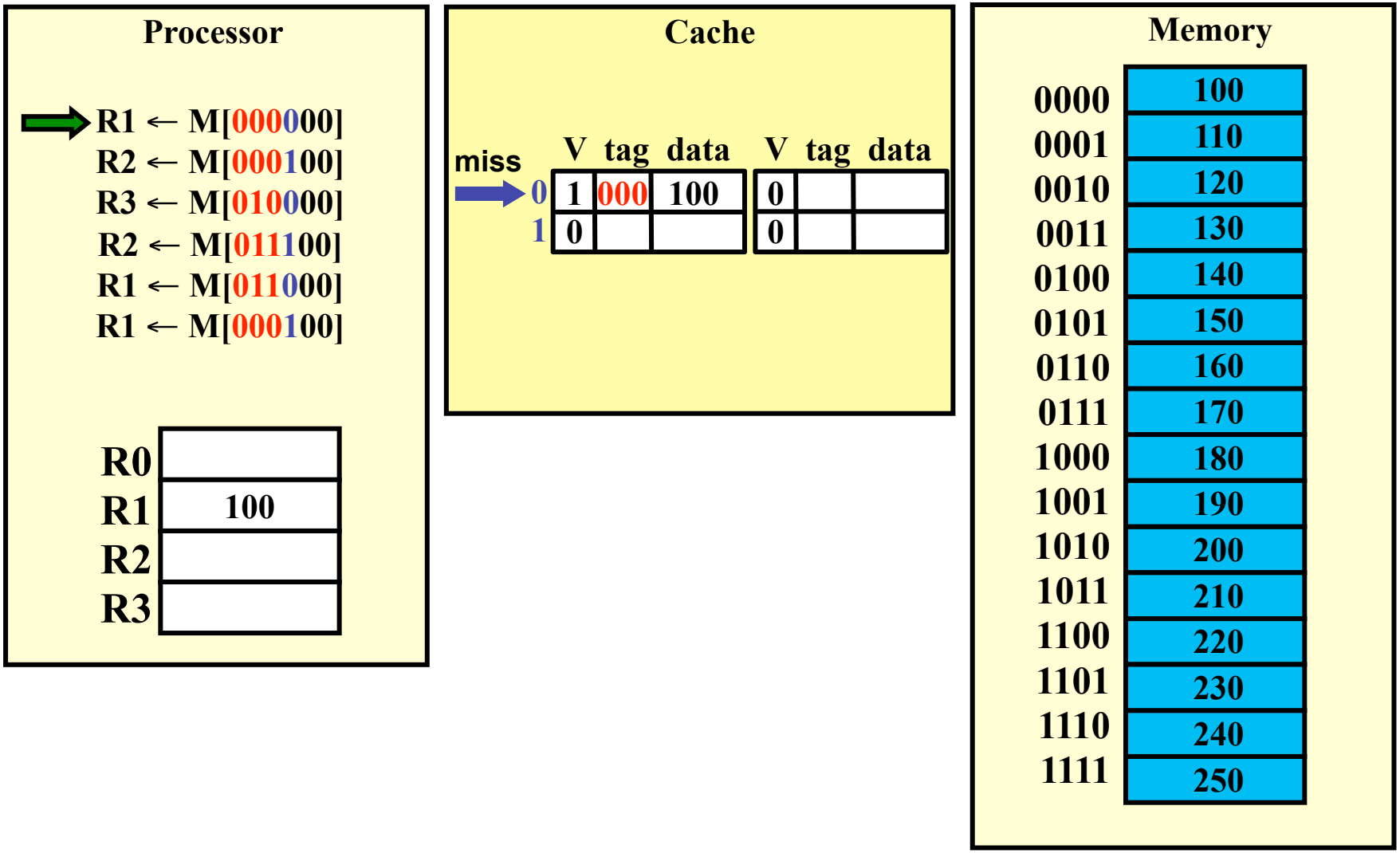




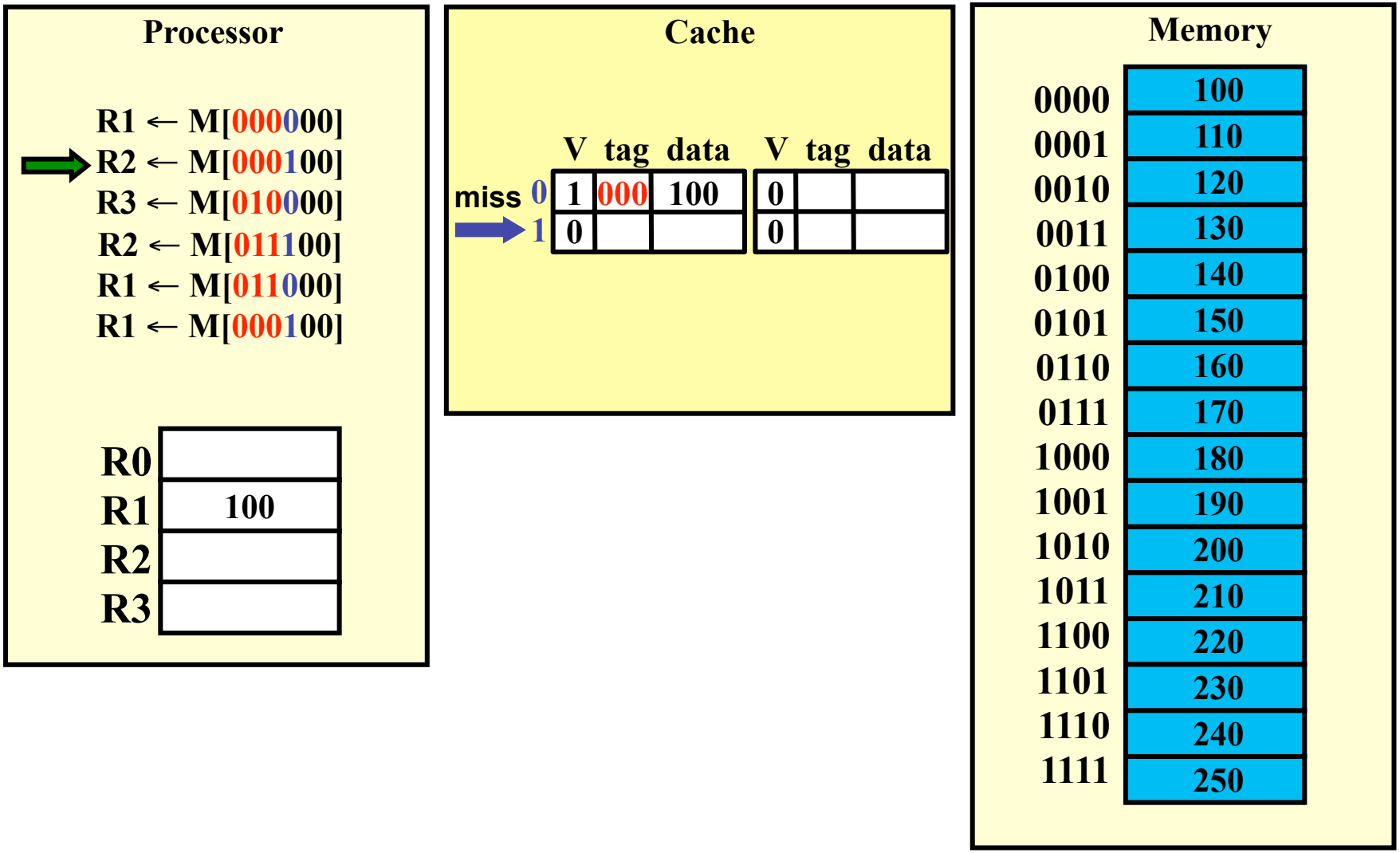
2-way Set Associative Example



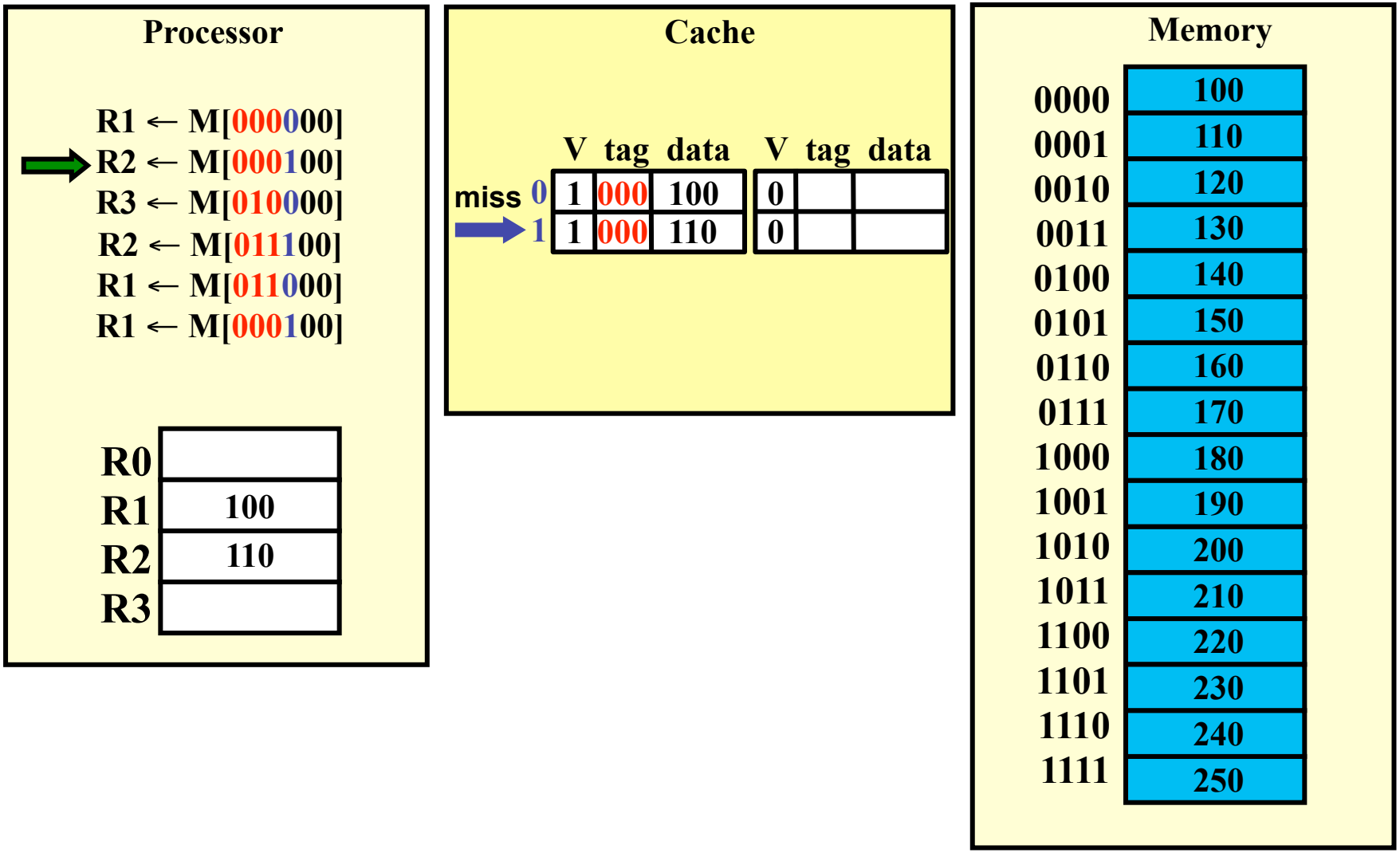
2-way Set Associative Example



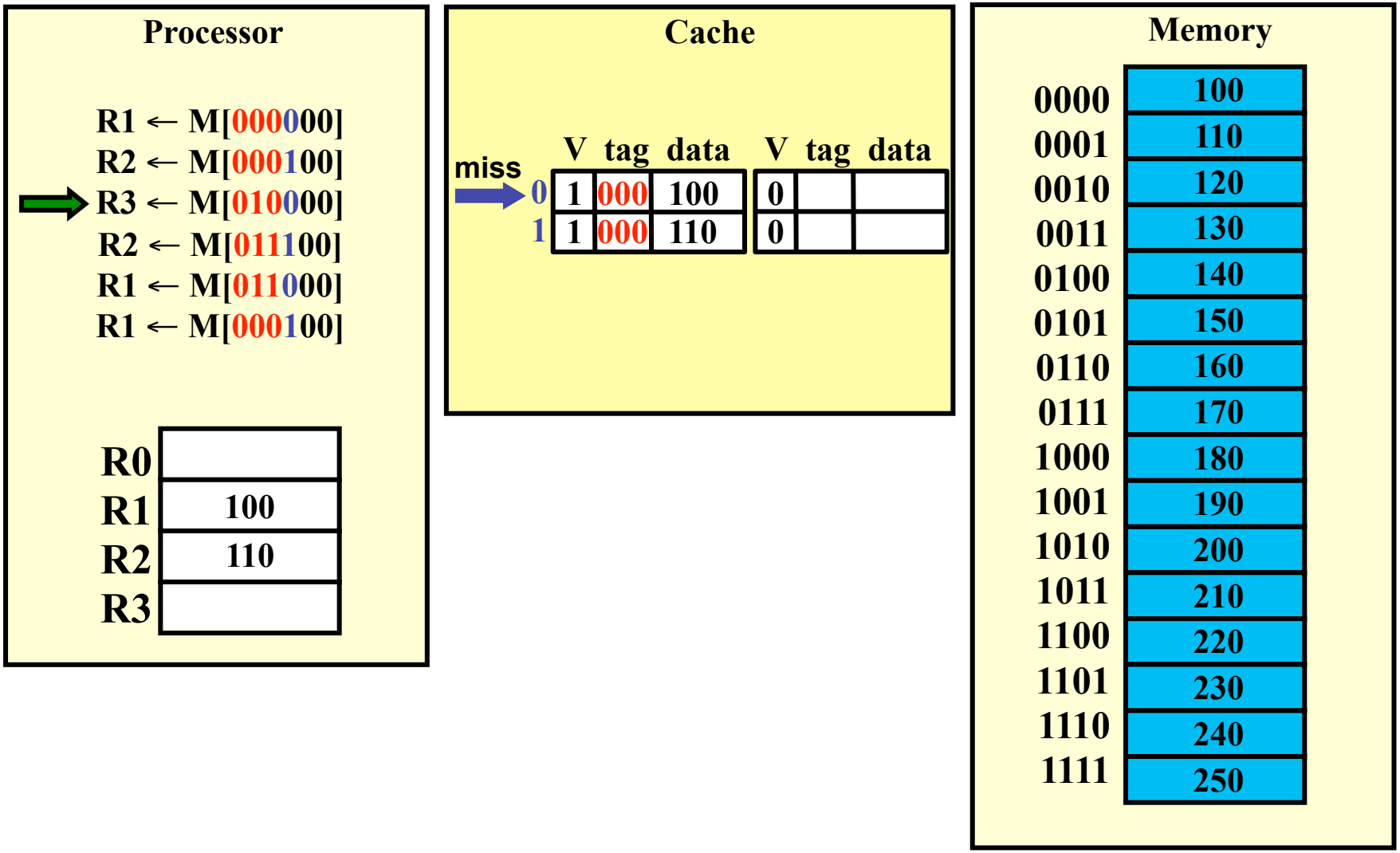
2-way Set Associative Example



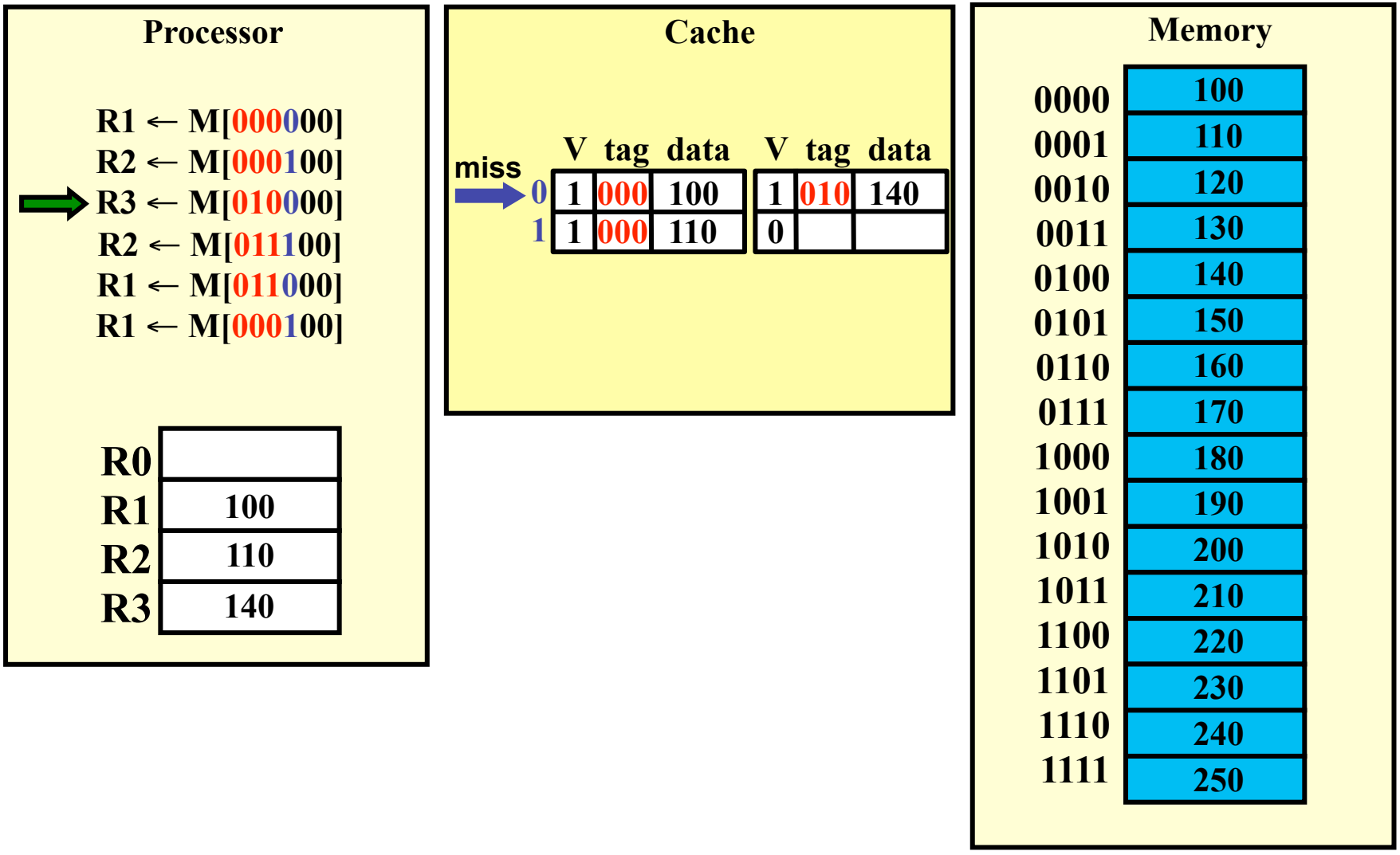
2-way Set Associative Example



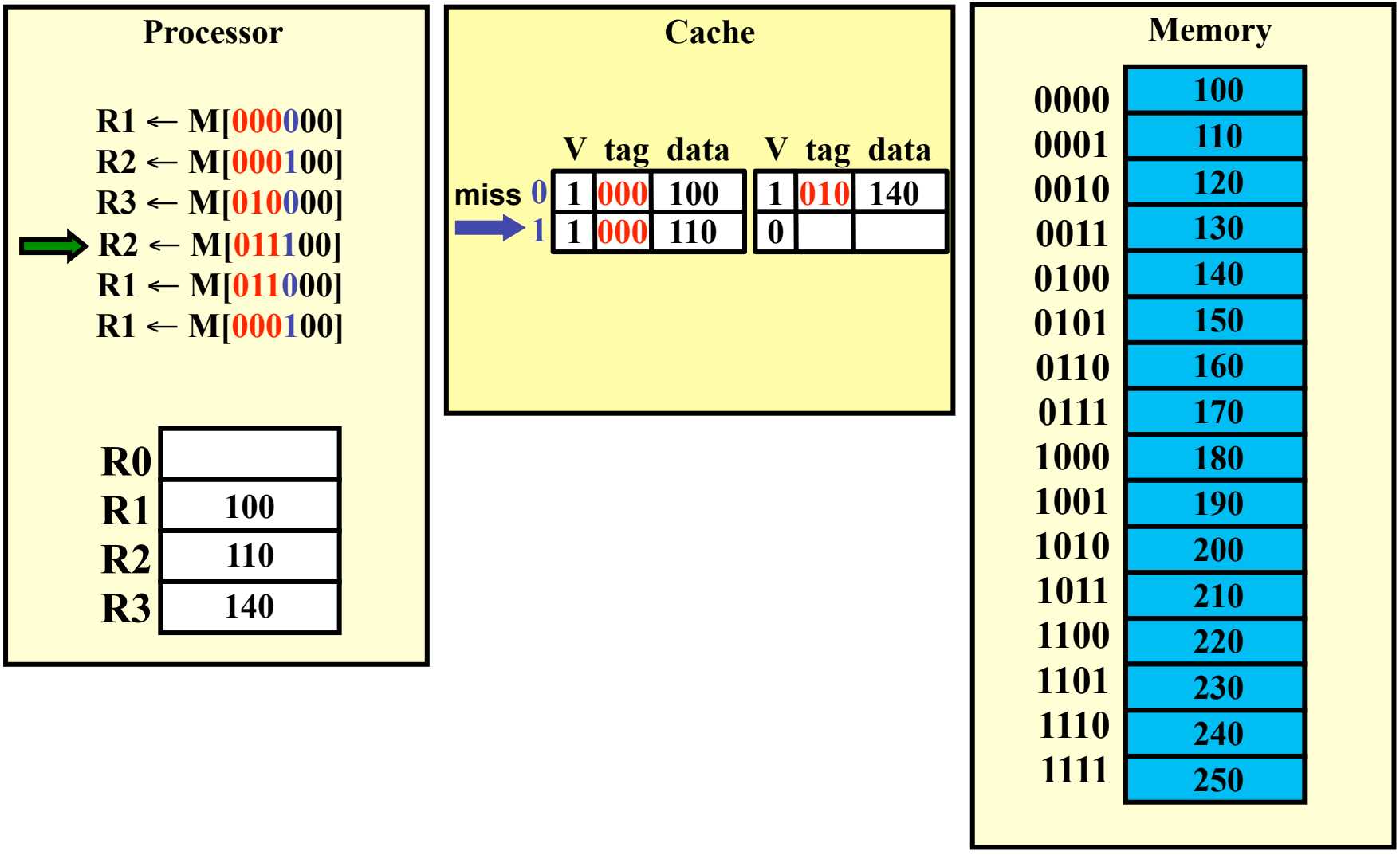
2-way Set Associative Example



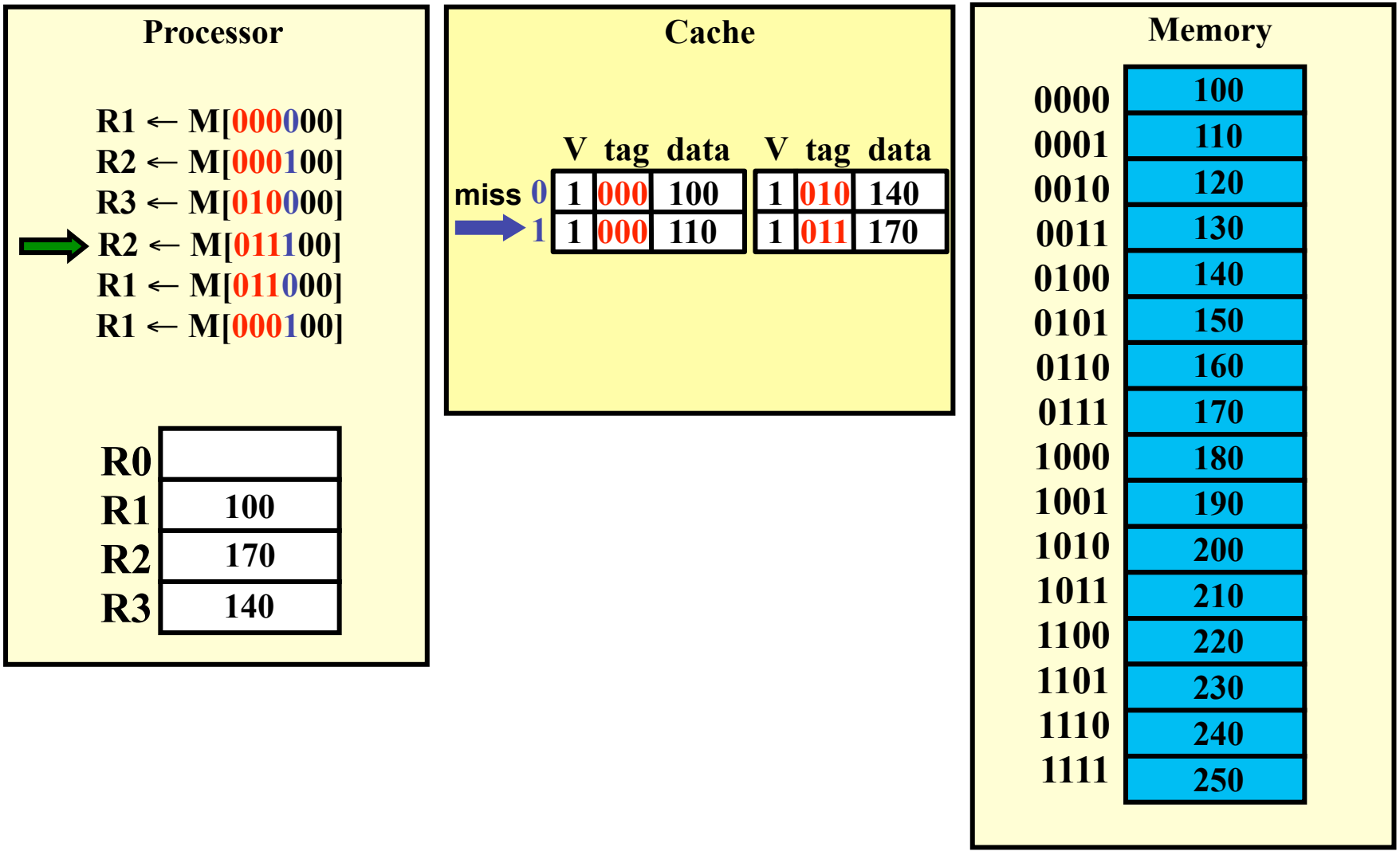
2-way Set Associative Example



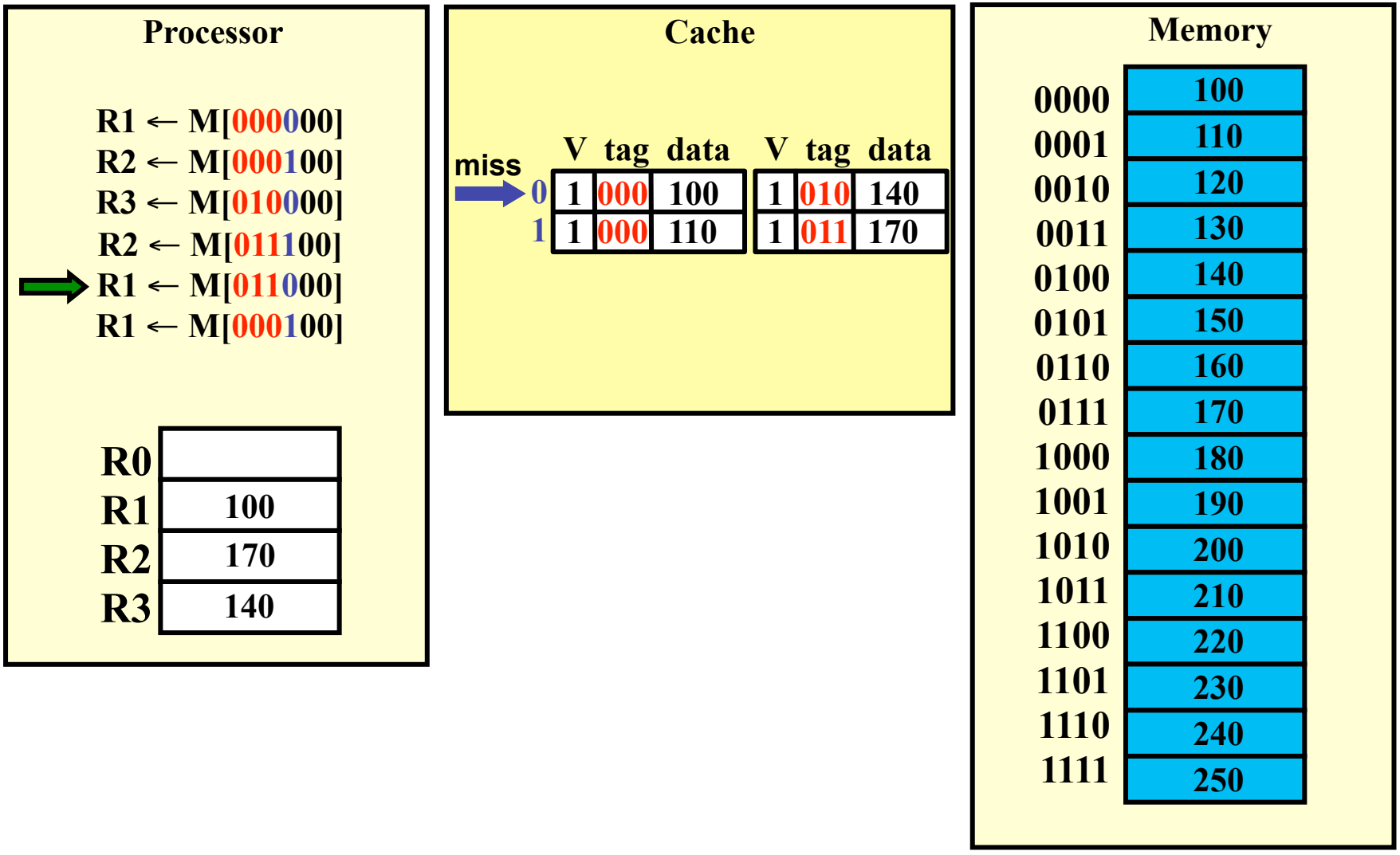
2-way Set Associative Example



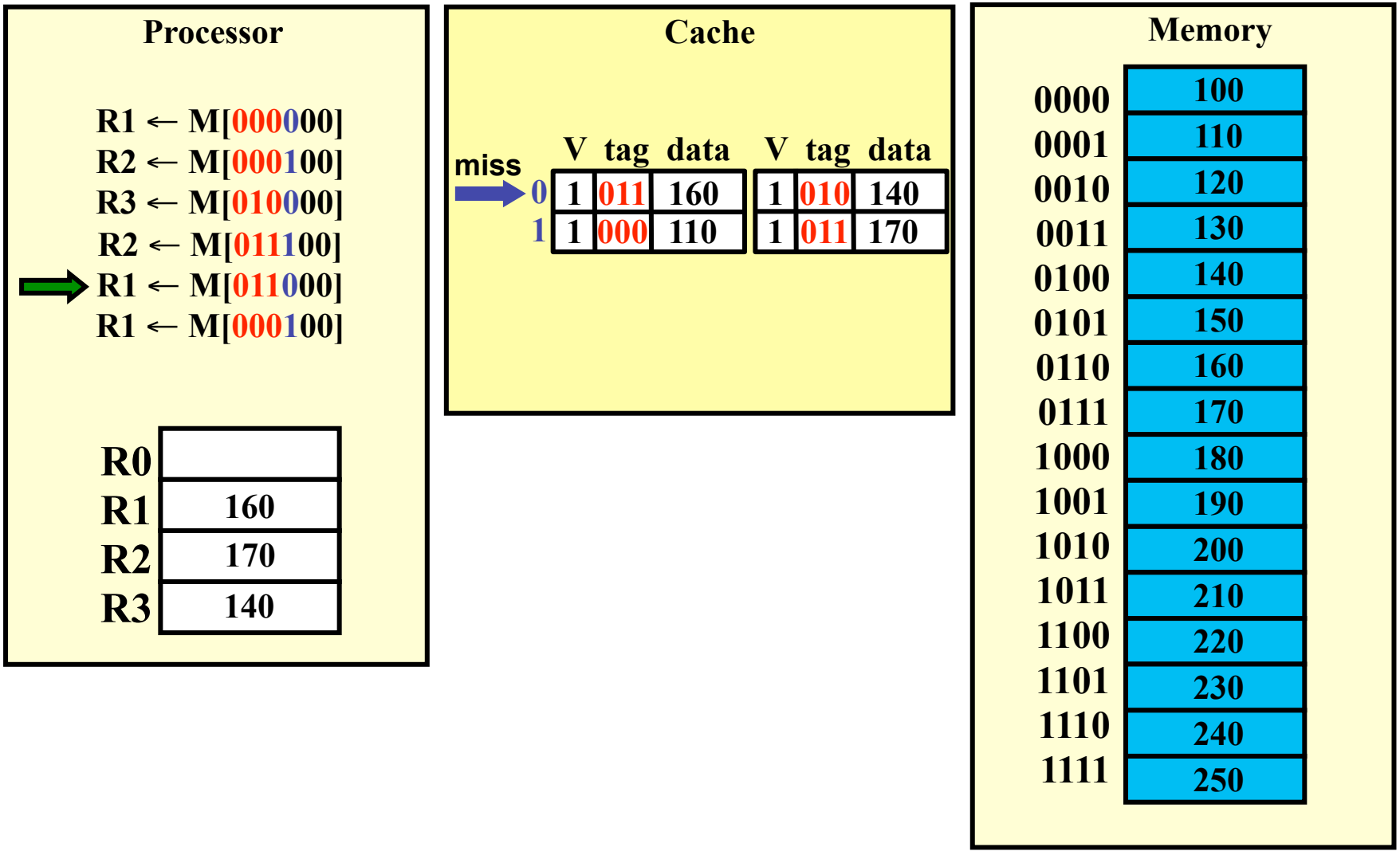
2-way Set Associative Example



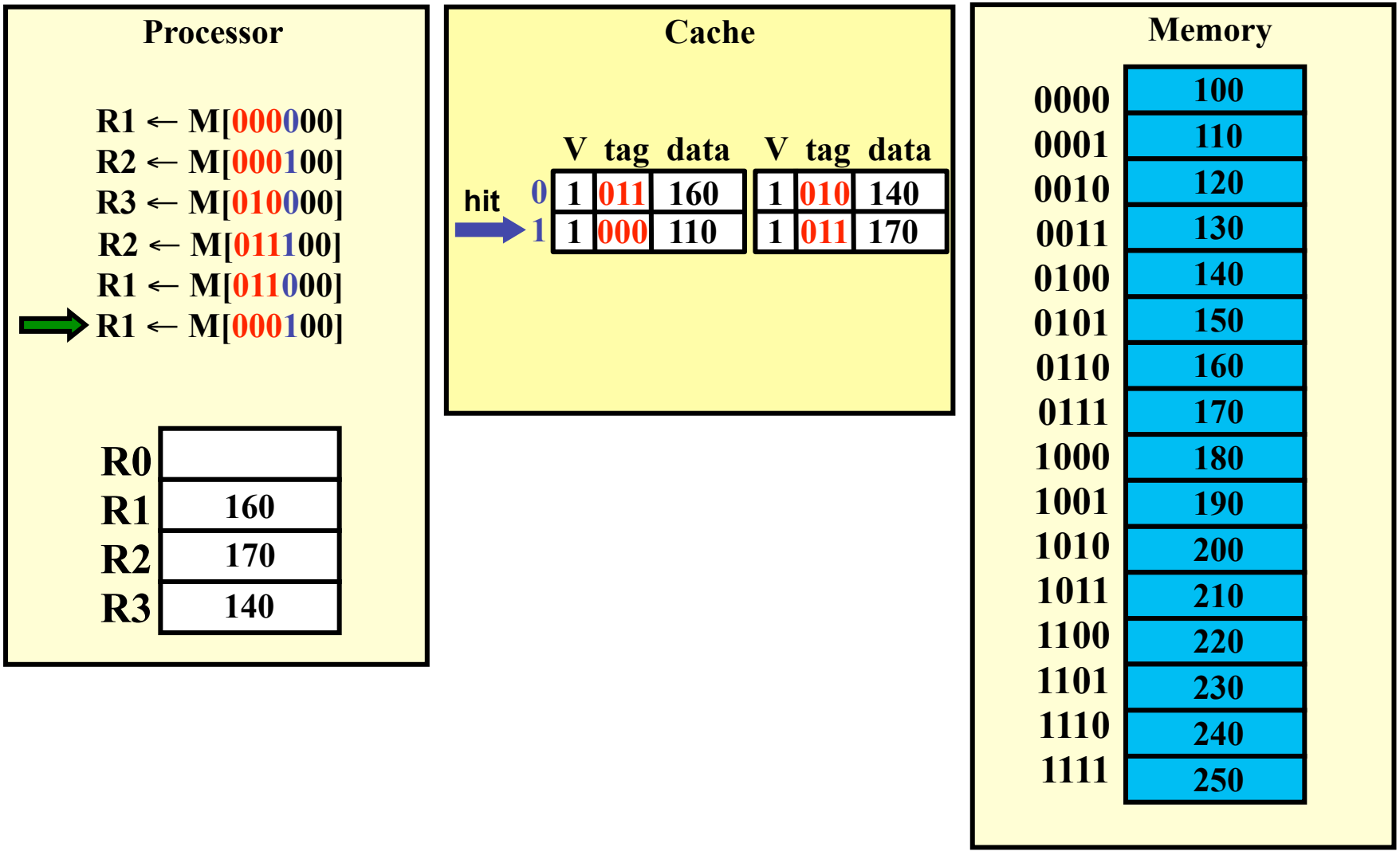
2-way Set Associative Example



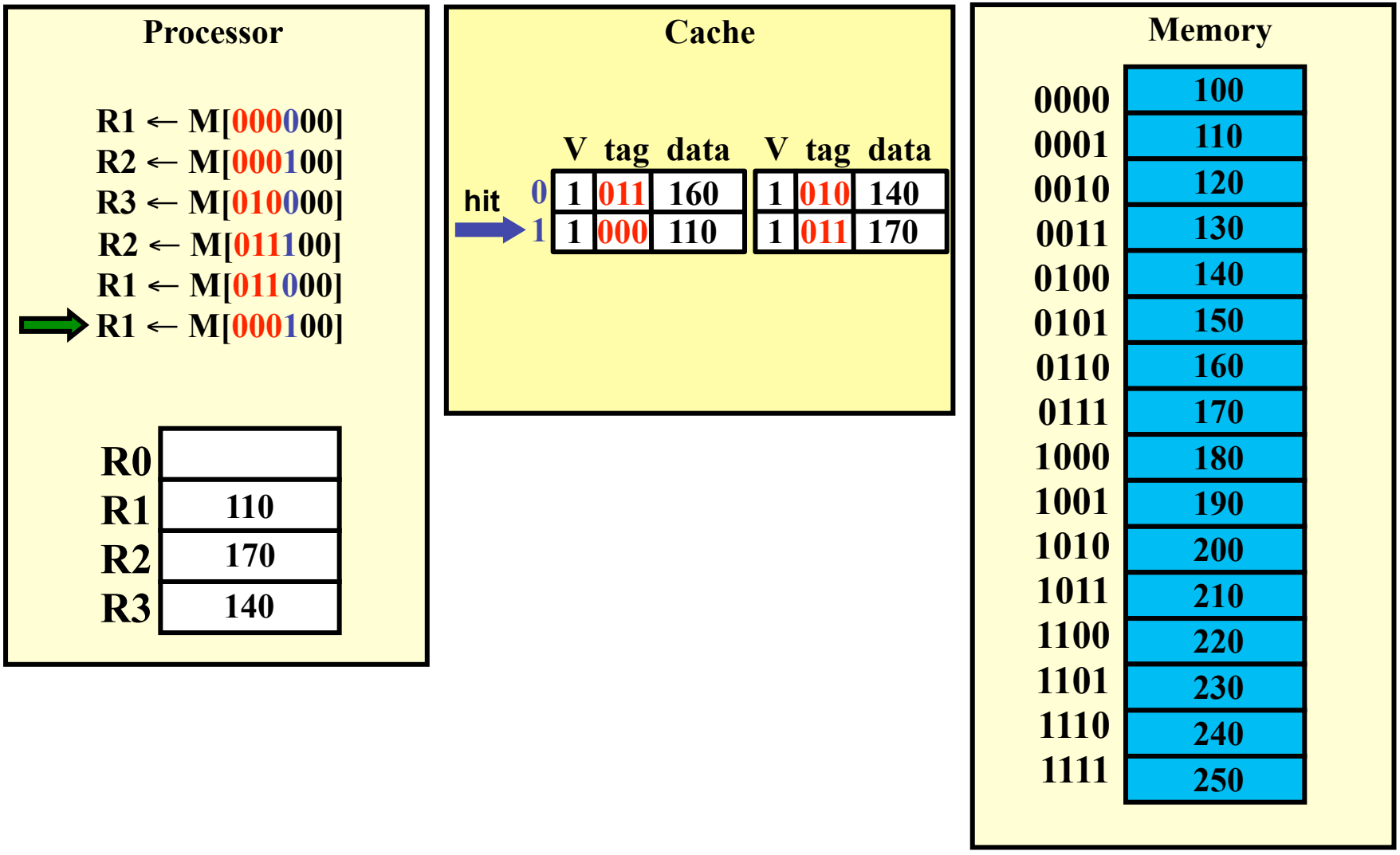
2-way Set Associative Example



2-way Set Associative Example



2-way Set Associative Example



Block Replacement Policy

- On a miss, which block to overwrite in a set associative or fully associative cache
- Look for a way within the selected set with V=0. If all ways are occupied, choose a block to replace.
- Common replacement policies
  - Least recently used (LRU)
    - Replace the block that has gone the longest between accesses
    - Requires storing extra bits to order the blocks
    - High overhead beyond 4-way set associative
  - Random
    - Similar performance as LRU for high associativity
  - Most recently used (MRU)
    - Randomly choose a block except for the most recently accessed

Miss Classification

- Compulsory (Cold) misses
  - Caused by the first access to a block
- Capacity misses
  - Occur because the cache might not big big enough to hold all the blocks needed during execution
- Conflict misses
  - Occur in set-associative or direct mapped cache when multiple blocks compete for the same set, but would not occur in a fully associative cache of the same size

Misses vs. Associativity Example

- **Compare caches with a capacity of 4 blocks**
  - **Direct mapped, 2-way set associative, fully associative**
  - **Block address sequence: 0, 8, 0, 6, 8**

- **Direct mapped**

Block address	Cache index	Hit/miss	Cache contents after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Misses vs. Associativity Example

- **2-way set associative**

Block address	Cache index	Hit/miss	Cache contents after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

- **Fully associative**

Block address		Hit/miss	Cache contents after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

How Much Associativity?

- Higher associativity decreases miss rate
  - But with diminishing returns

- Miss rates for 64KB data cache, 64 byte block size, SPEC2000 benchmarks
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

LRU Replacement Example

- **2-way set associative**

(*) = LRU block

Block address	Cache index	Hit/miss	Cache contents after access			
			Set 0		Set 1	
0	0	miss	Mem[0]			
4	0	miss	Mem[0] (*)	Mem[4]		
2	0	miss	Mem[2]	Mem[4] (*)		
6	0	miss	Mem[2] (*)	Mem[6]		
8	0	miss	Mem[8]	Mem[6] (*)		
0	0	miss	Mem[8] (*)	Mem[0]		
4	0	miss	Mem[4]	Mem[0] (*)		
2	0	miss	Mem[4] (*)	Mem[2]		
6	0	miss	Mem[6]	Mem[2] (*)		
8	0	miss	Mem[6] (*)	Mem[8]		
2	0	miss	Mem[2]	Mem[8] (*)		
6	0	miss	Mem[2] (*)	Mem[6]		
2	0	hit	Mem[2]	Mem[6] (*)		
0	0	miss	Mem[2] (*)	Mem[0]		

LRU Replacement Example

- **Fully associative**

(X) = access order

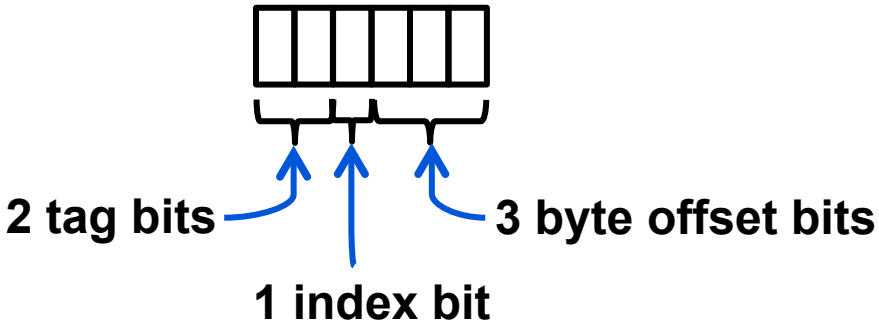
Block address		Hit/miss	Cache contents after access			
0		miss	Mem[0] (0)			
4		miss	Mem[0] (1)	Mem[4] (0)		
2		miss	Mem[0] (2)	Mem[4] (1)	Mem[2] (0)	
6		miss	Mem[0] (3)	Mem[4] (2)	Mem[2] (1)	Mem[6] (0)
8		miss	Mem[8] (0)	Mem[4] (3)	Mem[2] (2)	Mem[6] (1)
0		miss	Mem[8] (1)	Mem[0] (0)	Mem[2] (3)	Mem[6] (2)
4		miss	Mem[8] (2)	Mem[0] (1)	Mem[4] (0)	Mem[6] (3)
2		miss	Mem[8] (3)	Mem[0] (2)	Mem[4] (1)	Mem[2] (0)
6		miss	Mem[6] (0)	Mem[0] (3)	Mem[4] (2)	Mem[2] (1)
8		miss	Mem[6] (1)	Mem[8] (0)	Mem[4] (3)	Mem[2] (2)
2		hit	Mem[6] (2)	Mem[8] (1)	Mem[4] (3)	Mem[2] (0)
6		hit	Mem[6] (0)	Mem[8] (2)	Mem[4] (3)	Mem[2] (1)
2		hit	Mem[6] (1)	Mem[8] (2)	Mem[4] (3)	Mem[2] (0)
0		miss	Mem[6] (2)	Mem[8] (3)	Mem[0] (0)	Mem[2] (1)

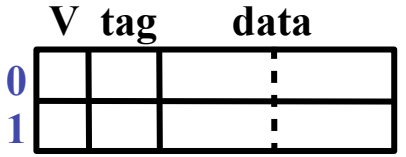
What About Writes?

- Where do we put the result of a store?
- Cache hit (block is in cache)
  - Write new value to the cache
  - Also write to memory (write through)
  - Don't write to memory (write back)
    - Requires an additional *dirty bit* for each block
- Cache miss (block is not in cache)
  - Allocate the line (bring it into the cache) (write allocate)
  - Write to memory without allocation (no write allocate or write around)

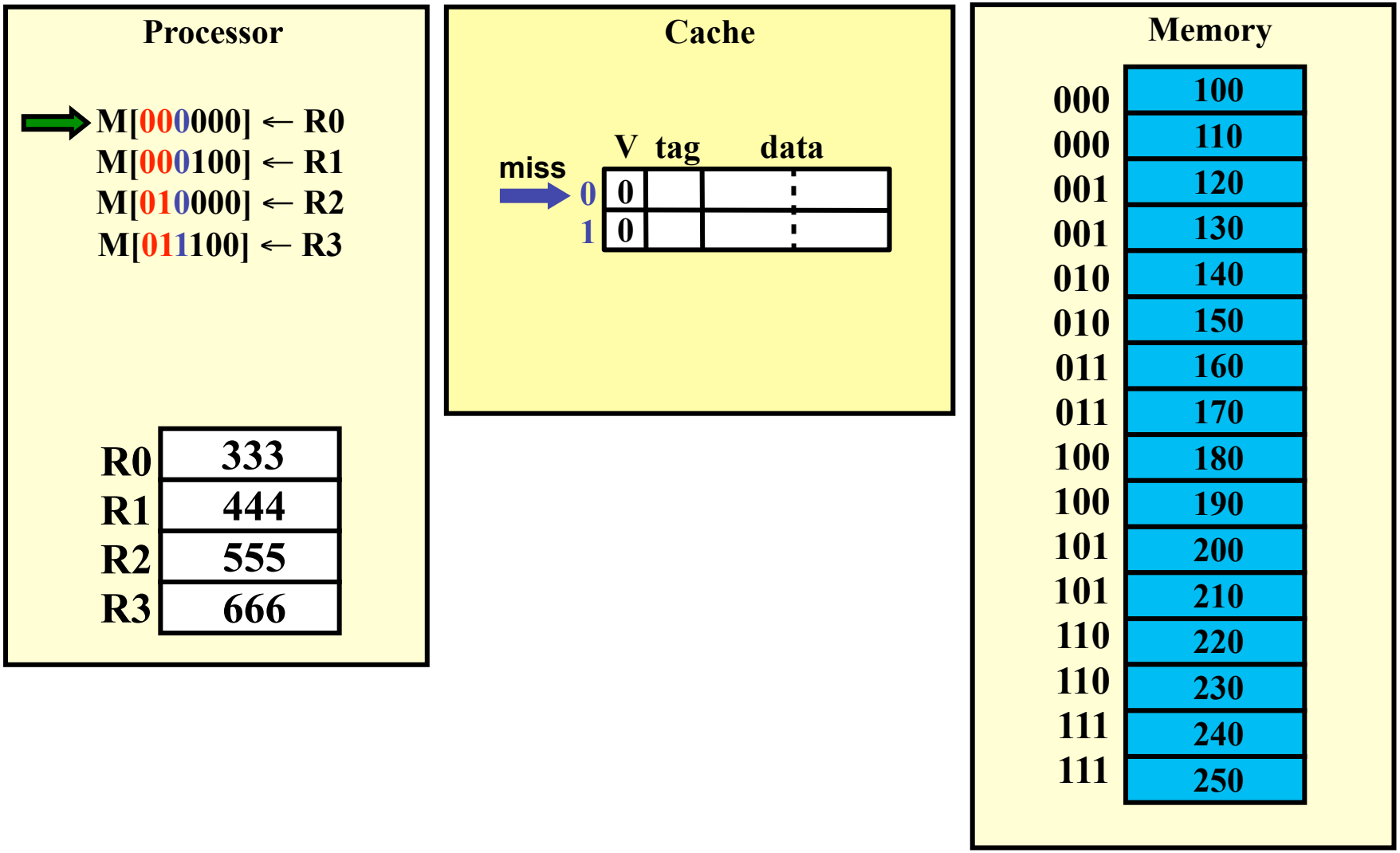
Write Through vs. Write Back Example

- Assume write allocate for both
- Size of each block is 8 bytes
- Cache holds 2 blocks
- Memory holds 8 blocks
- Memory address

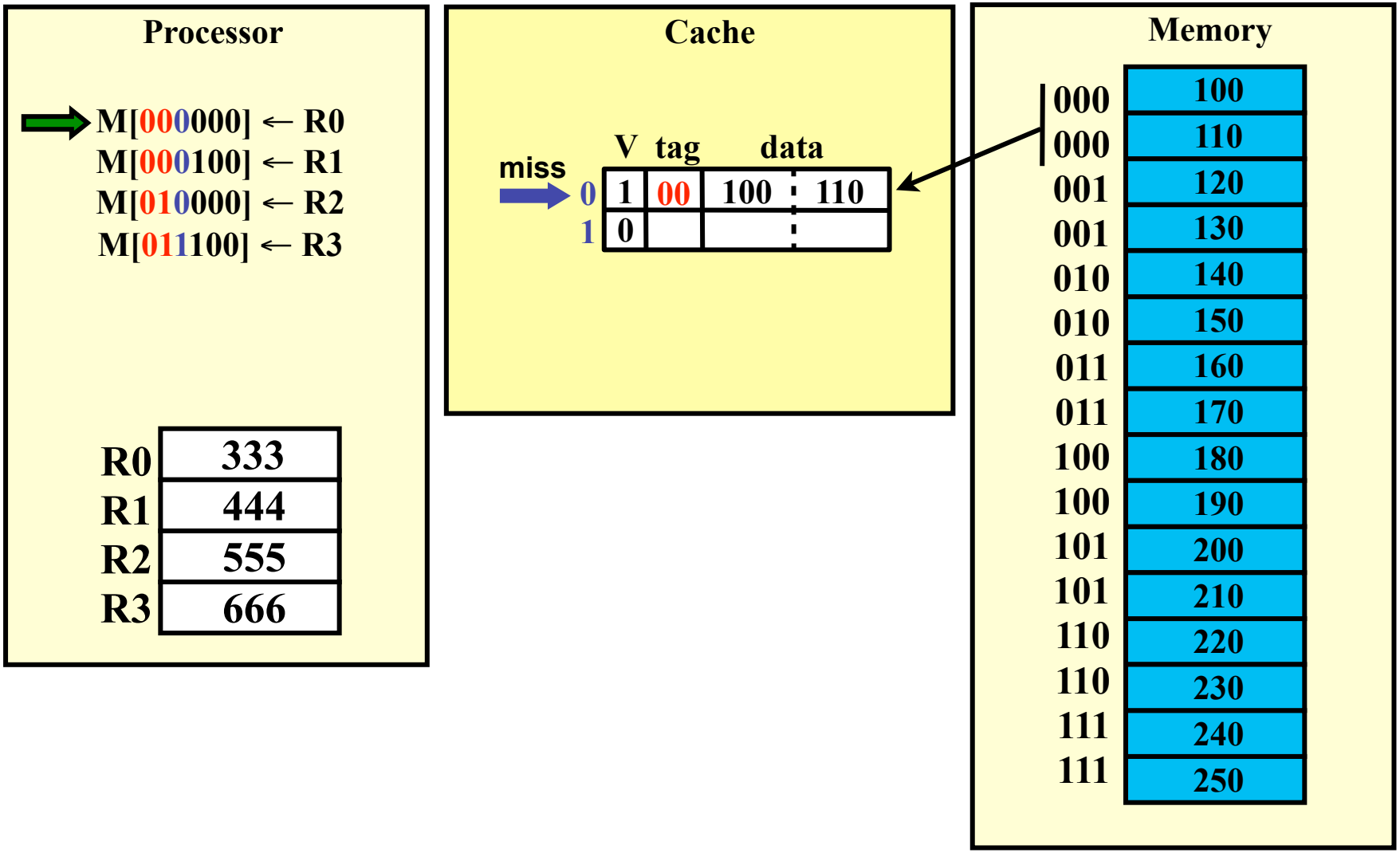




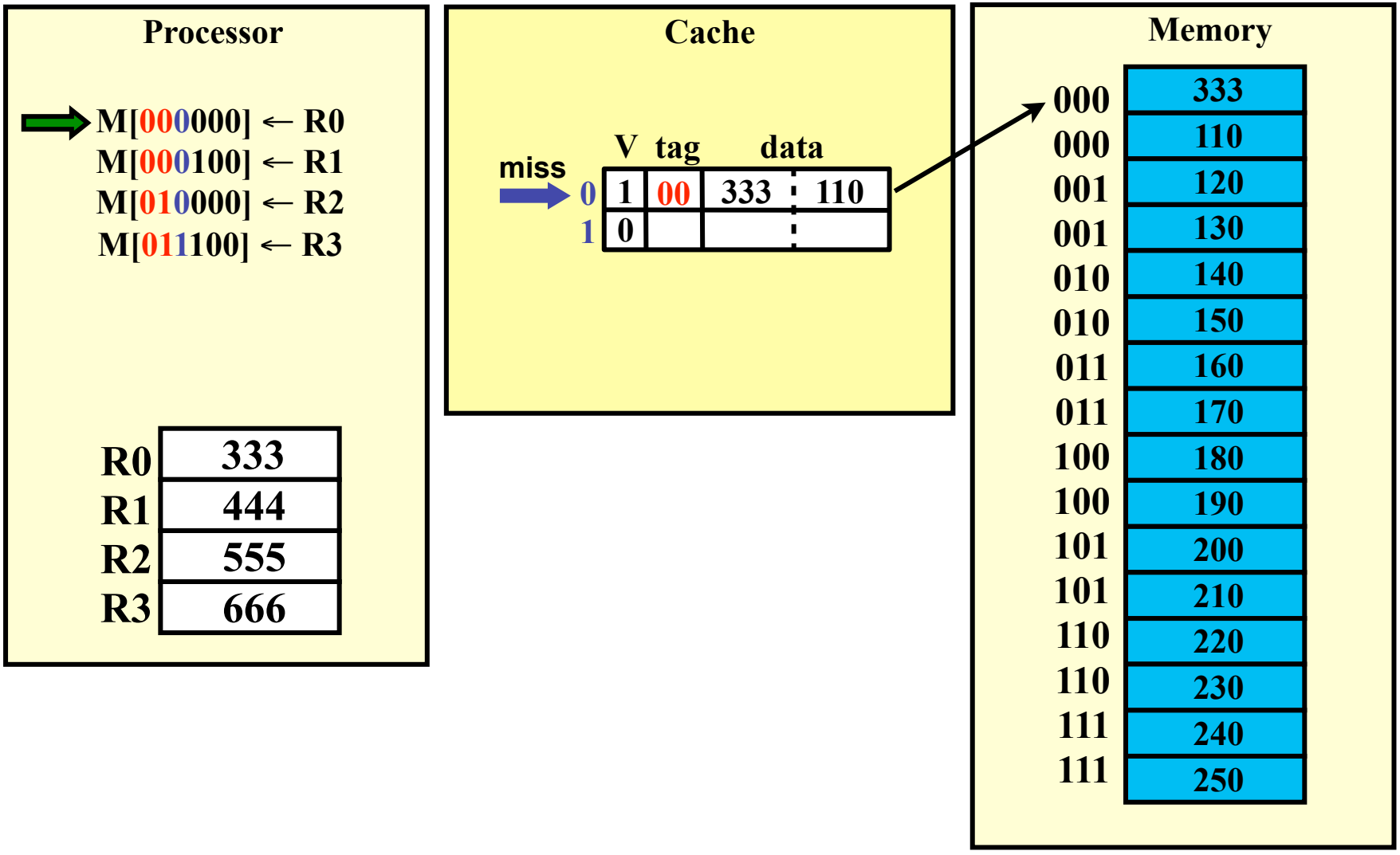
Write Through



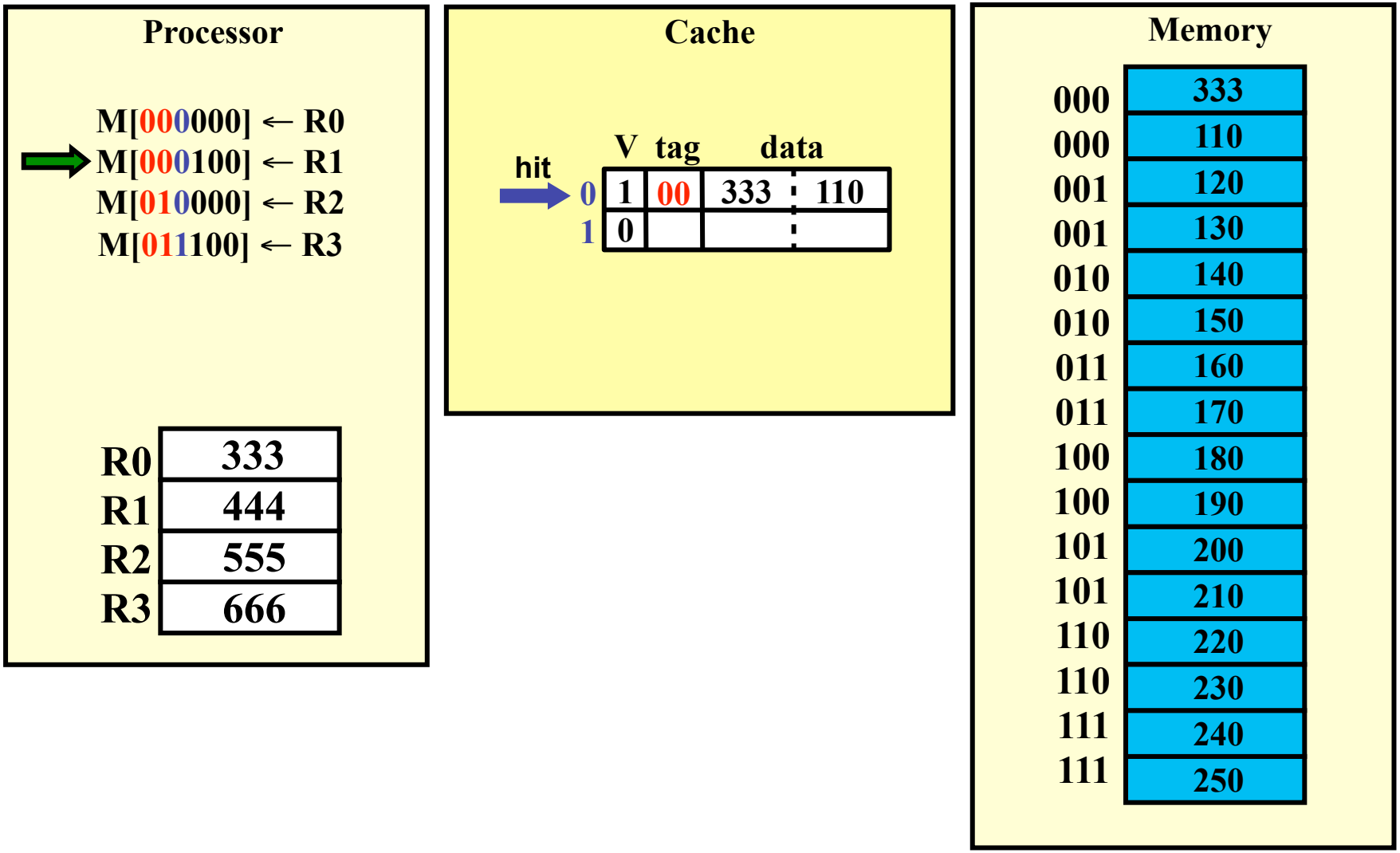
Write Through



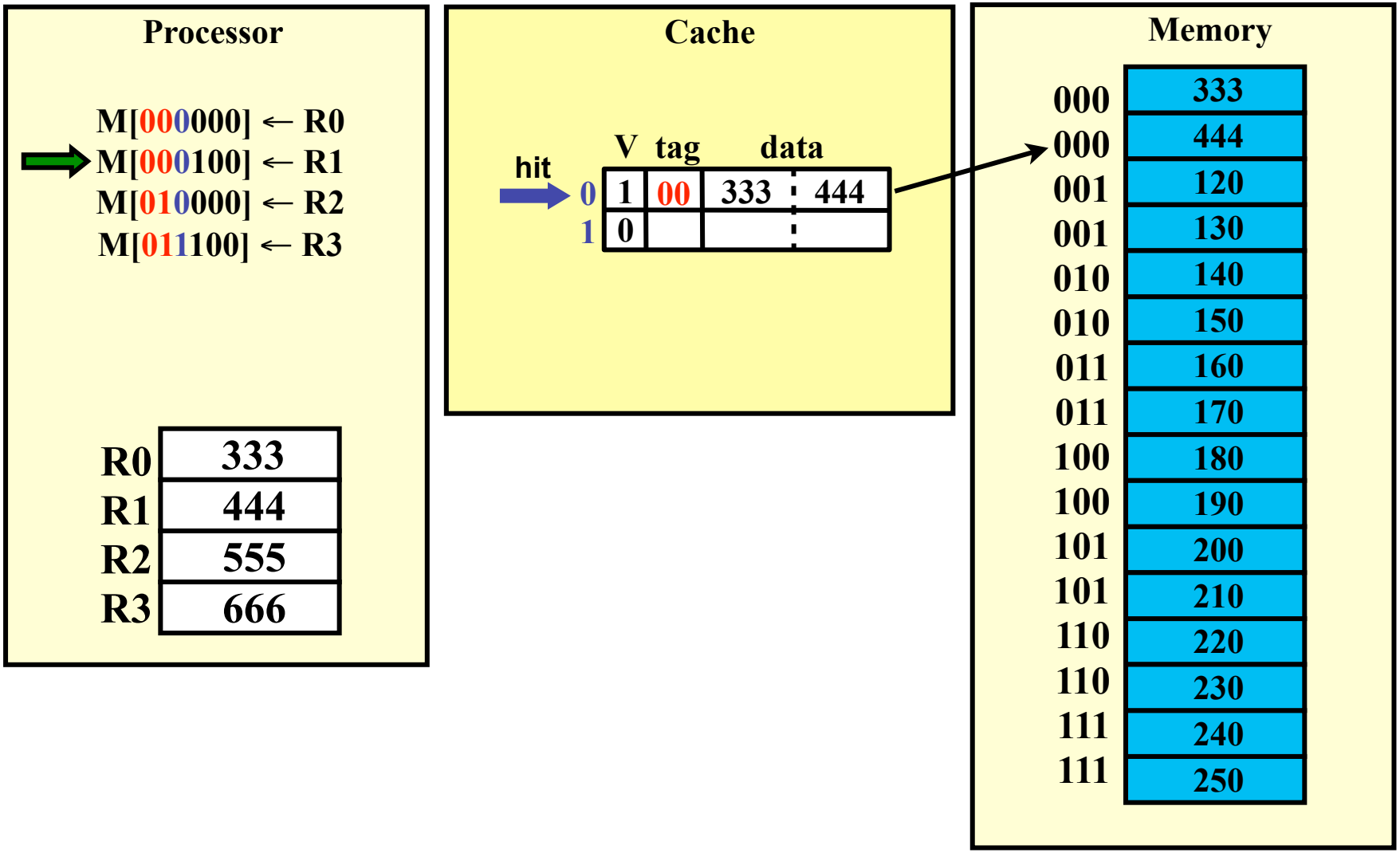
Write Through



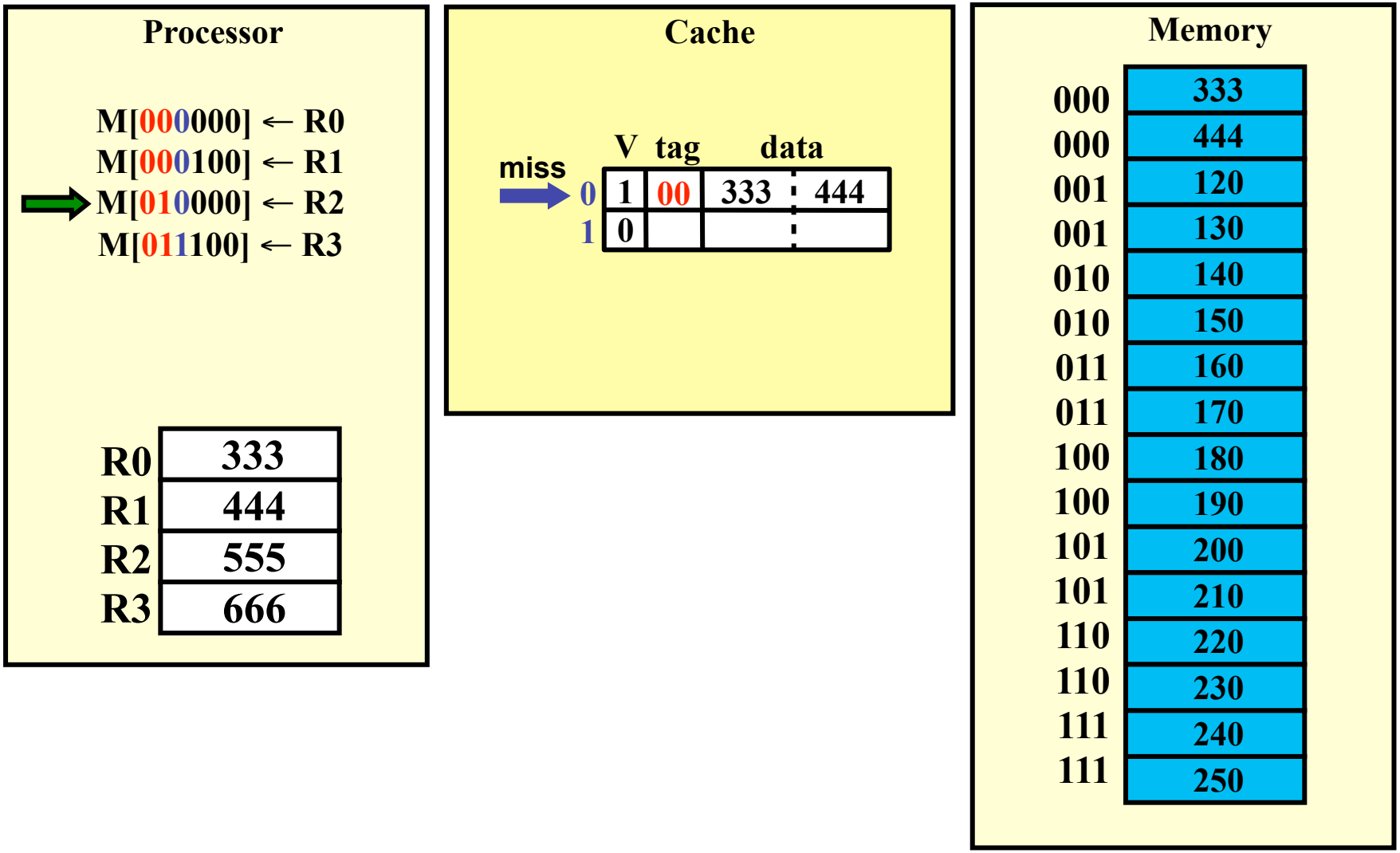
Write Through



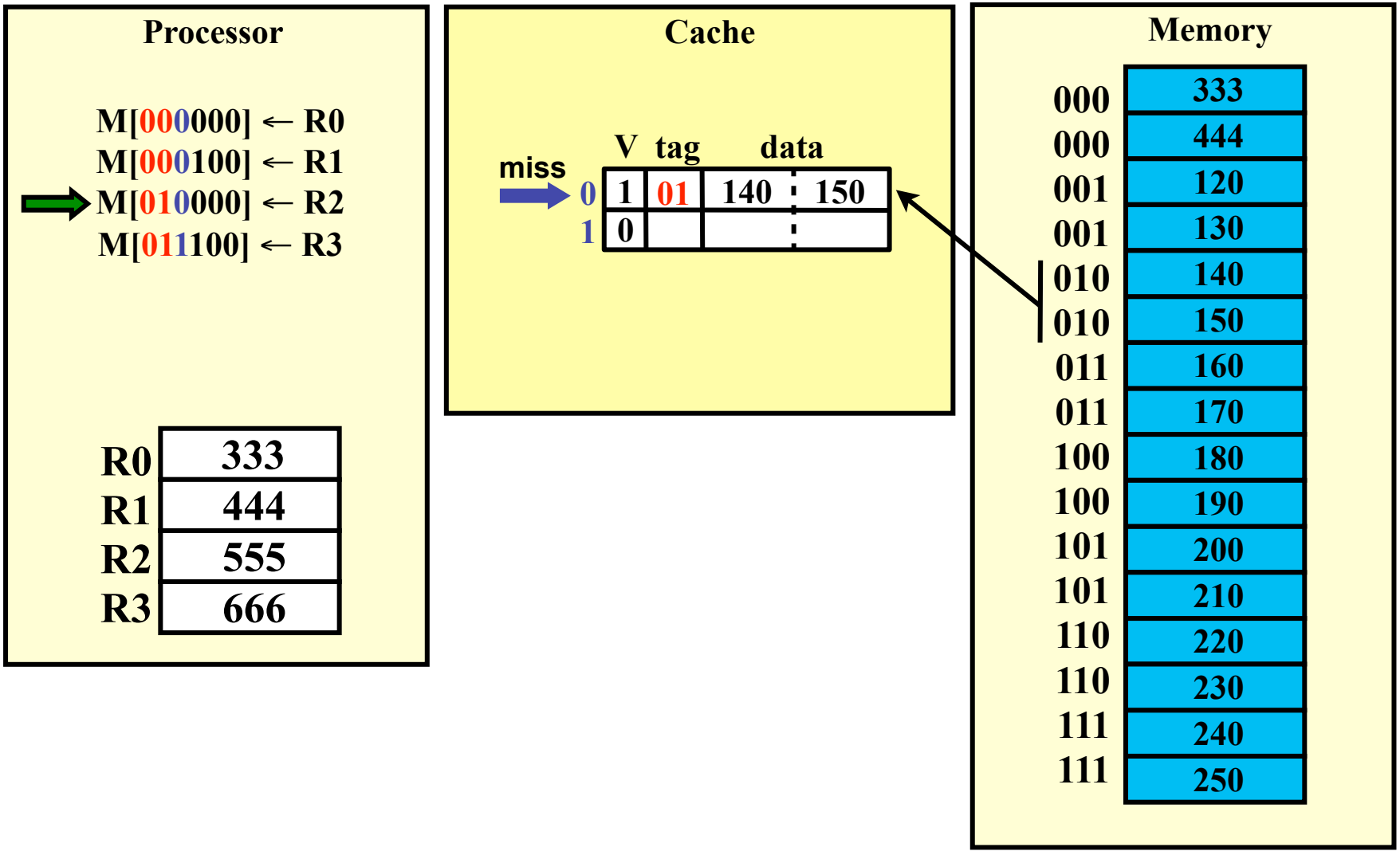
Write Through



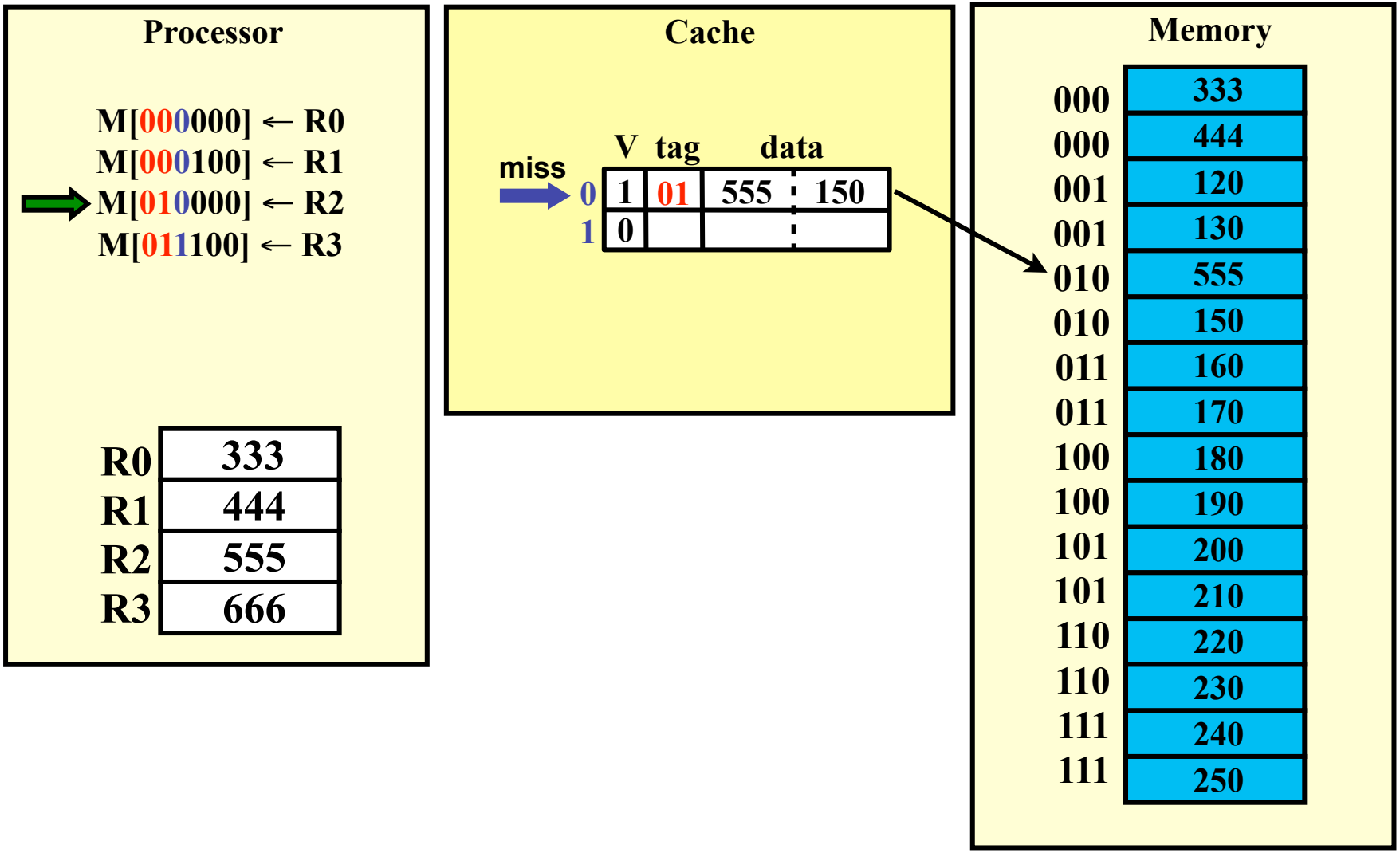
Write Through



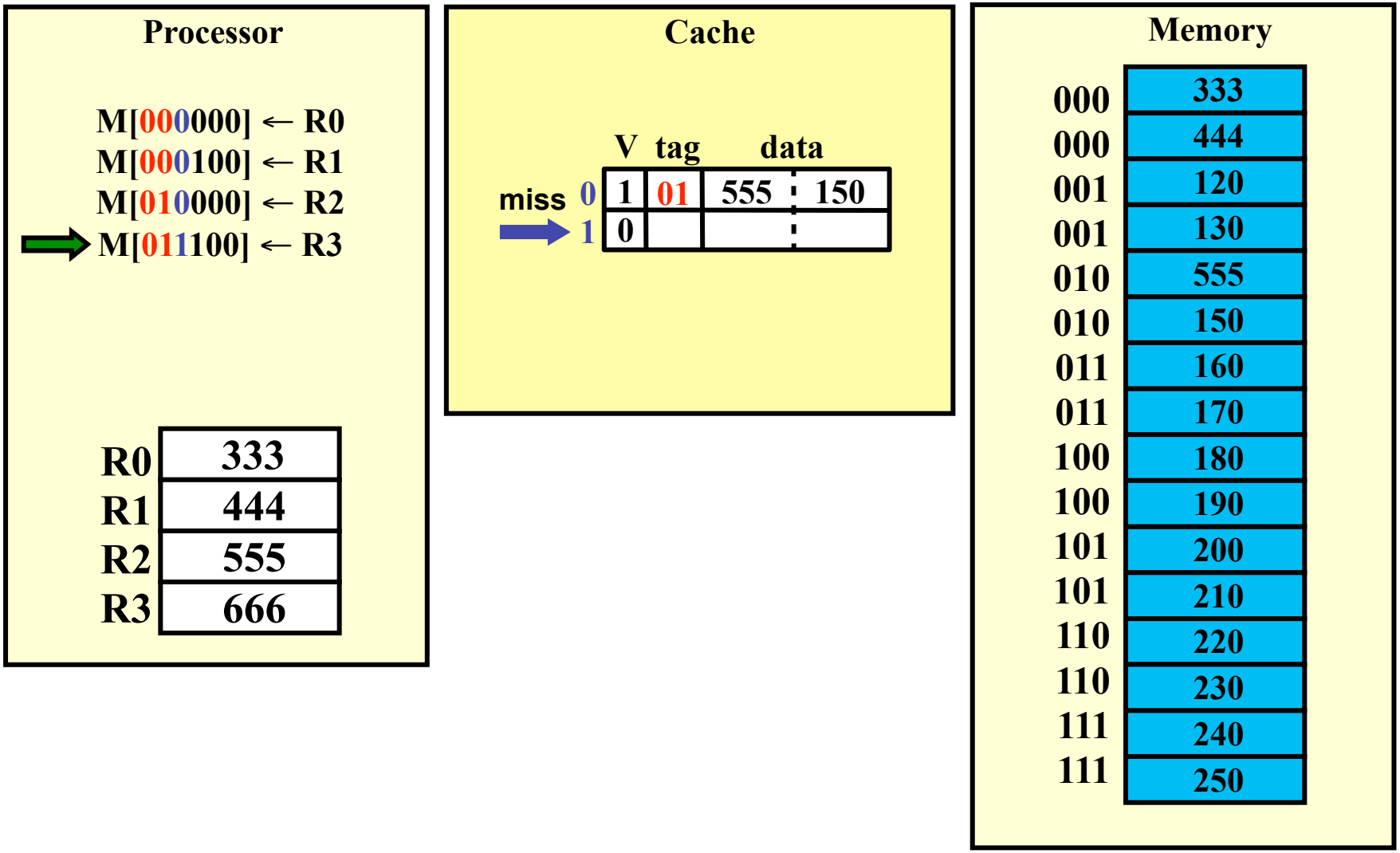
Write Through



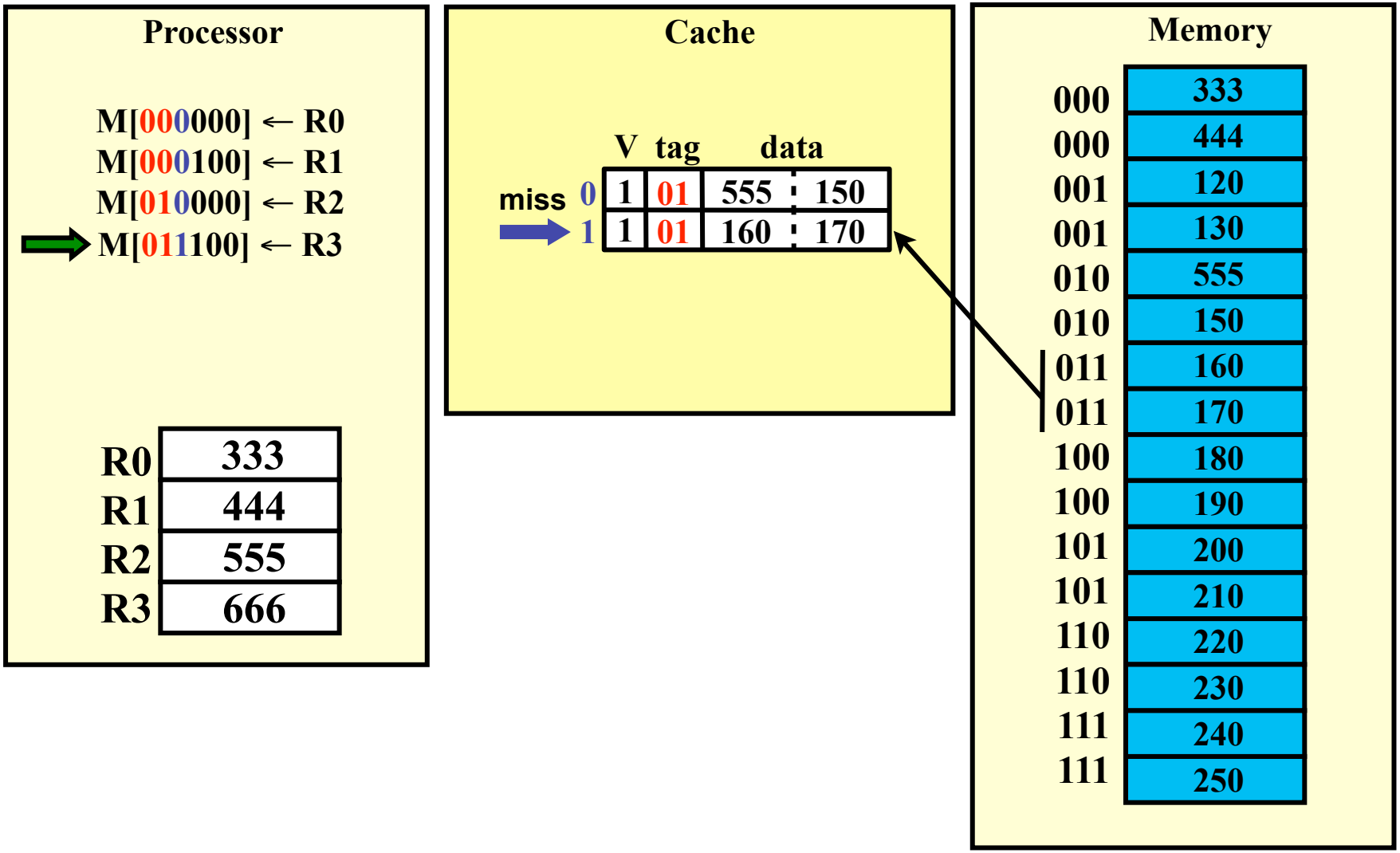
Write Through



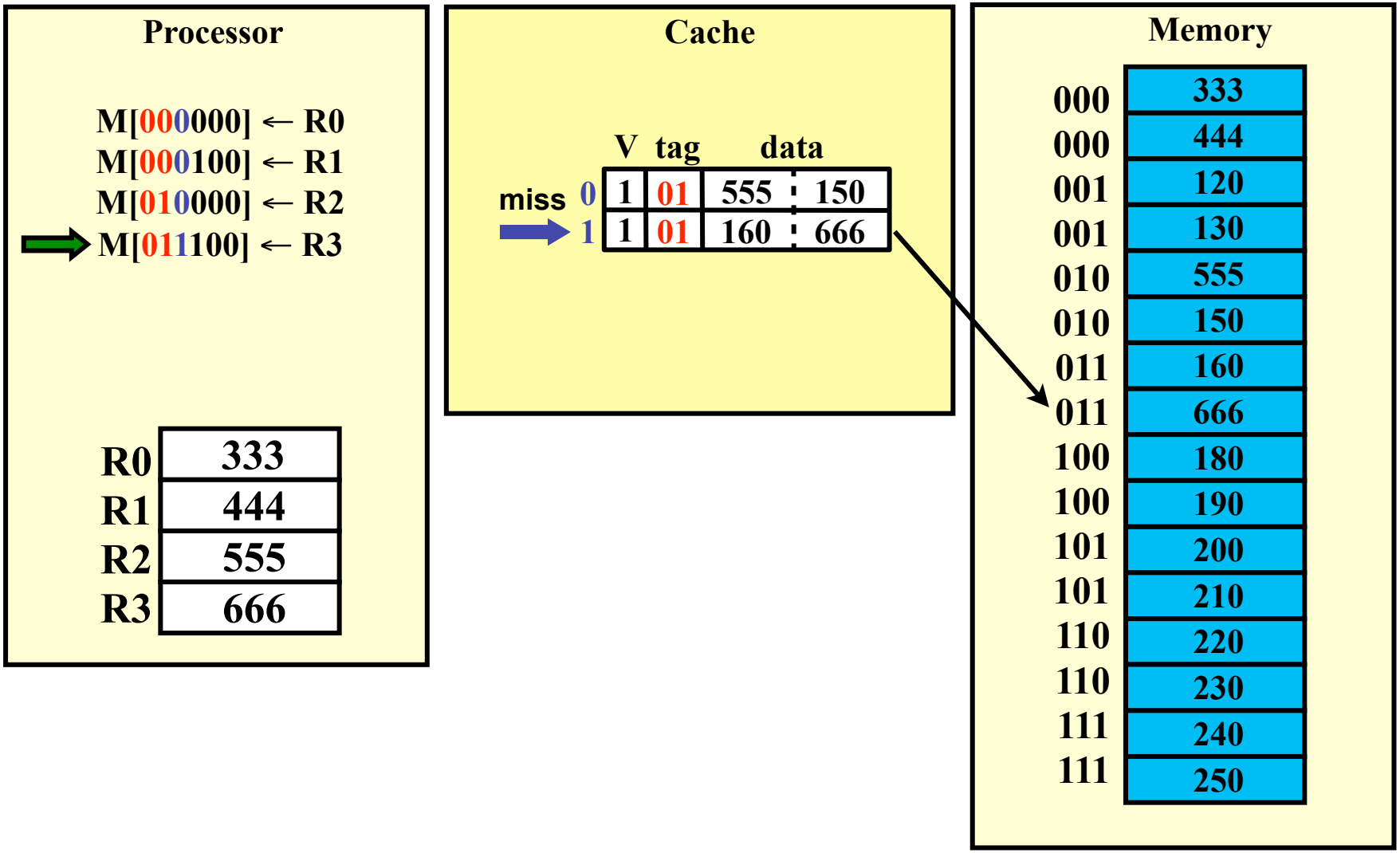
Write Through



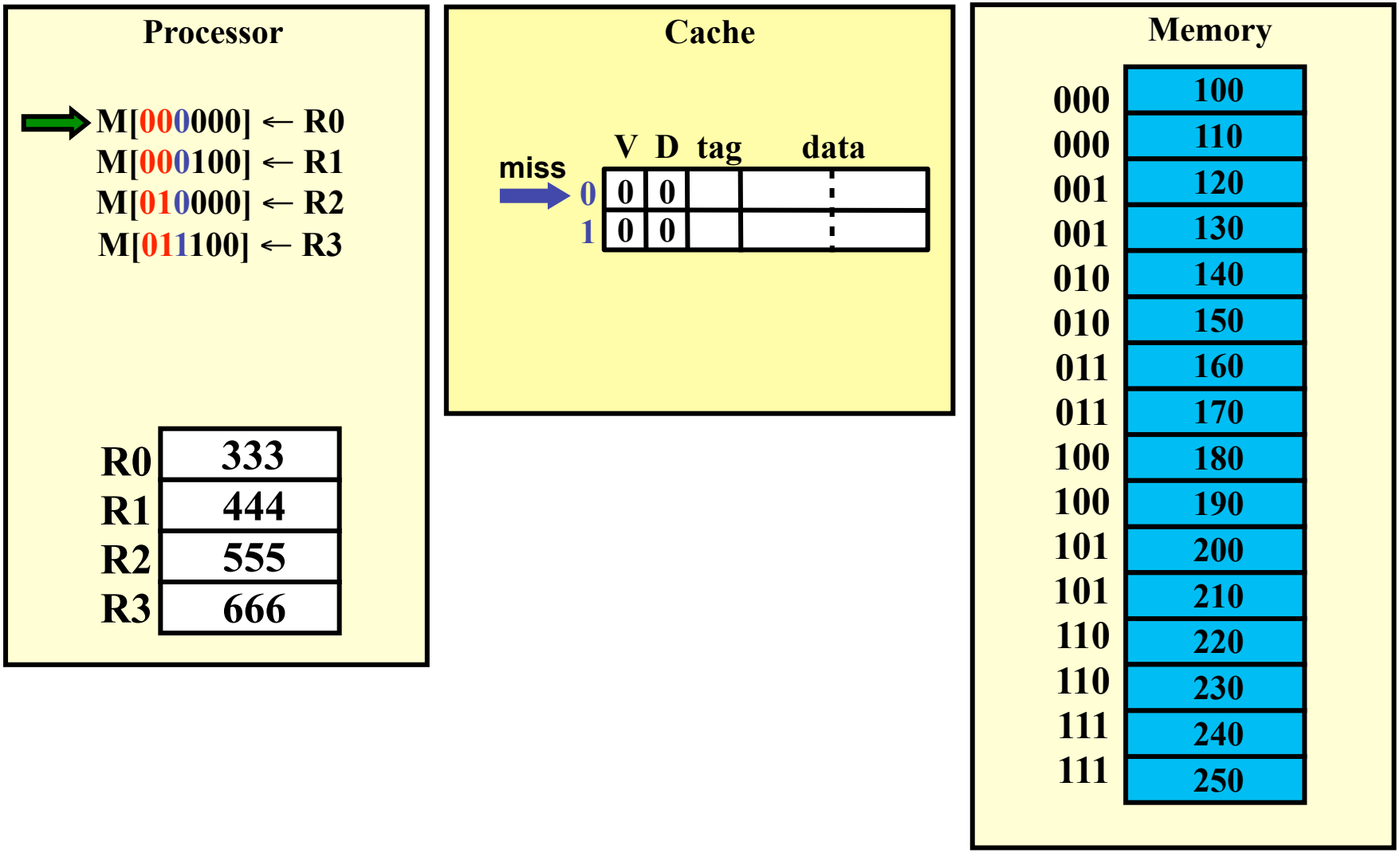
Write Through



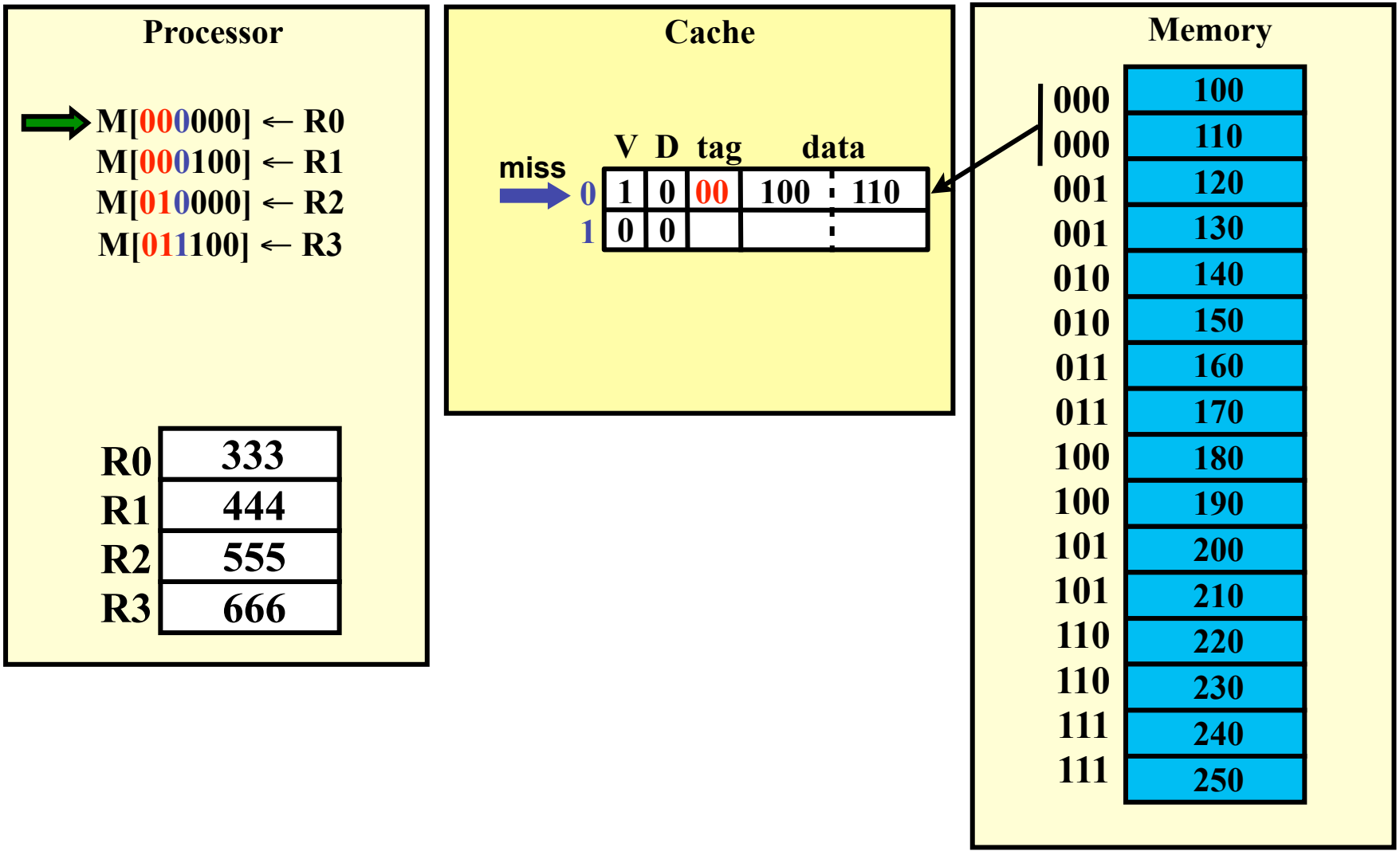
Write Through



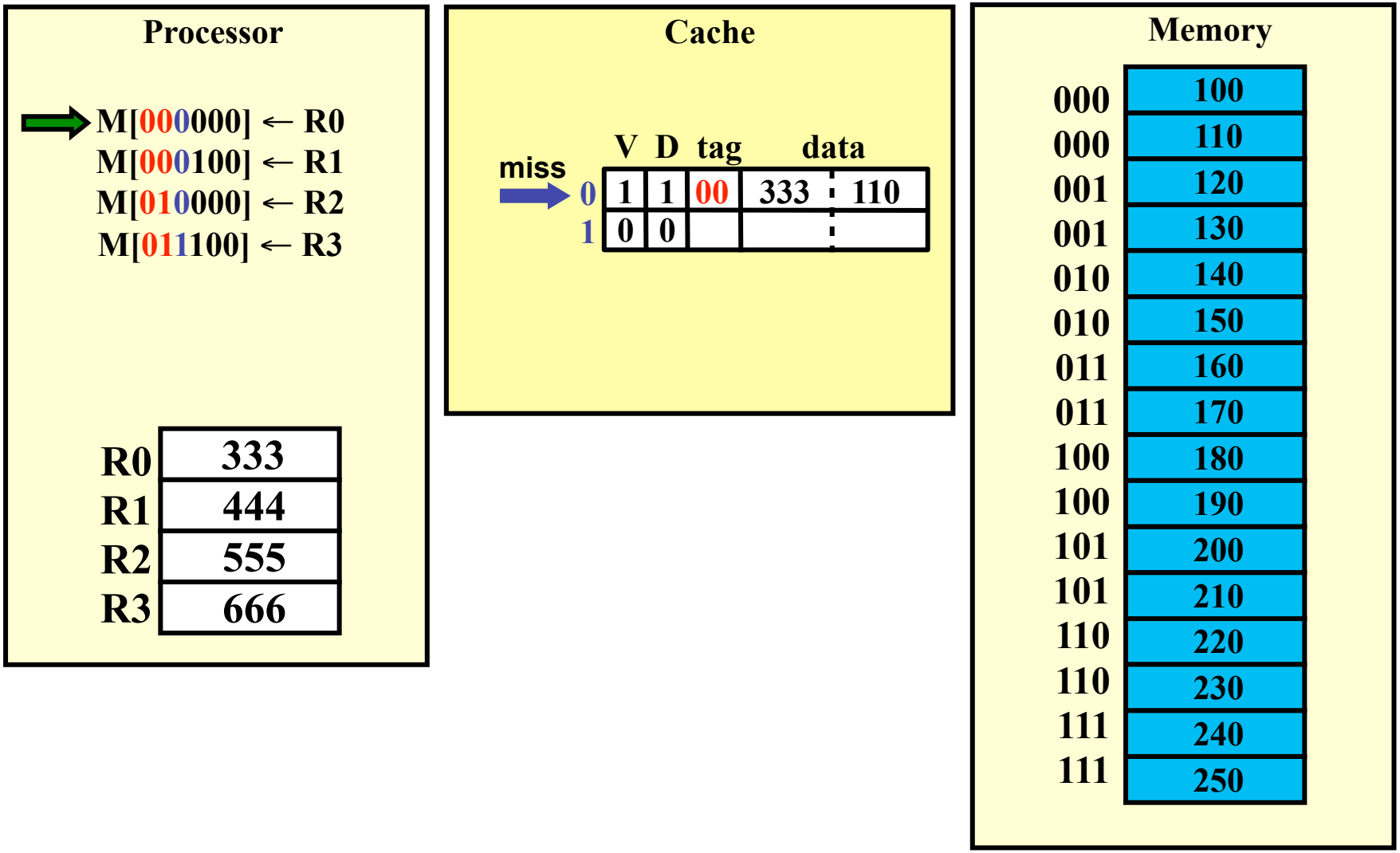
Write Back


Processor
M[000000] ← R0
M[000100] ← R1
M[010000] ← R2
M[011100] ← R3
R0 333
R1 444
R2 555
R3 666
Cache
miss
V D tag data
0 0 0
1 0 0
Memory
000 100
000 110
001 120
001 130
010 140
010 150
011 160
011 170
100 180
100 190
101 200
101 210
110 220
110 230
111 240
111 250

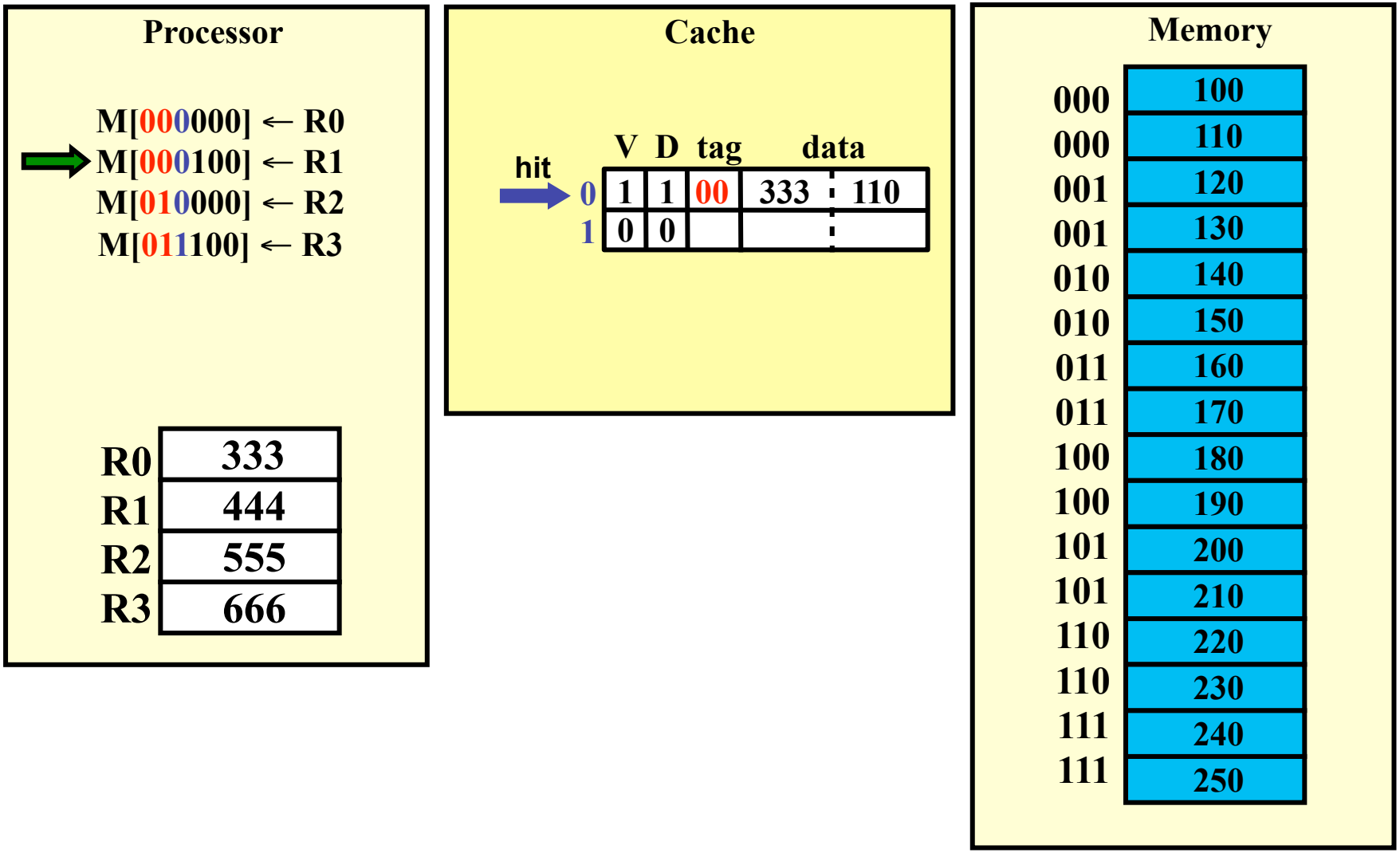
Write Back


Processor
M[000000] ← R0
M[000100] ← R1
M[010000] ← R2
M[011100] ← R3
R0 333
R1 444
R2 555
R3 666
Cache
miss
V D tag data
0 1 0 00 100 110
1 0 0
Memory
000 100
000 110
001 120
001 130
010 140
010 150
011 160
011 170
100 180
100 190
101 200
101 210
110 220
110 230
111 240
111 250

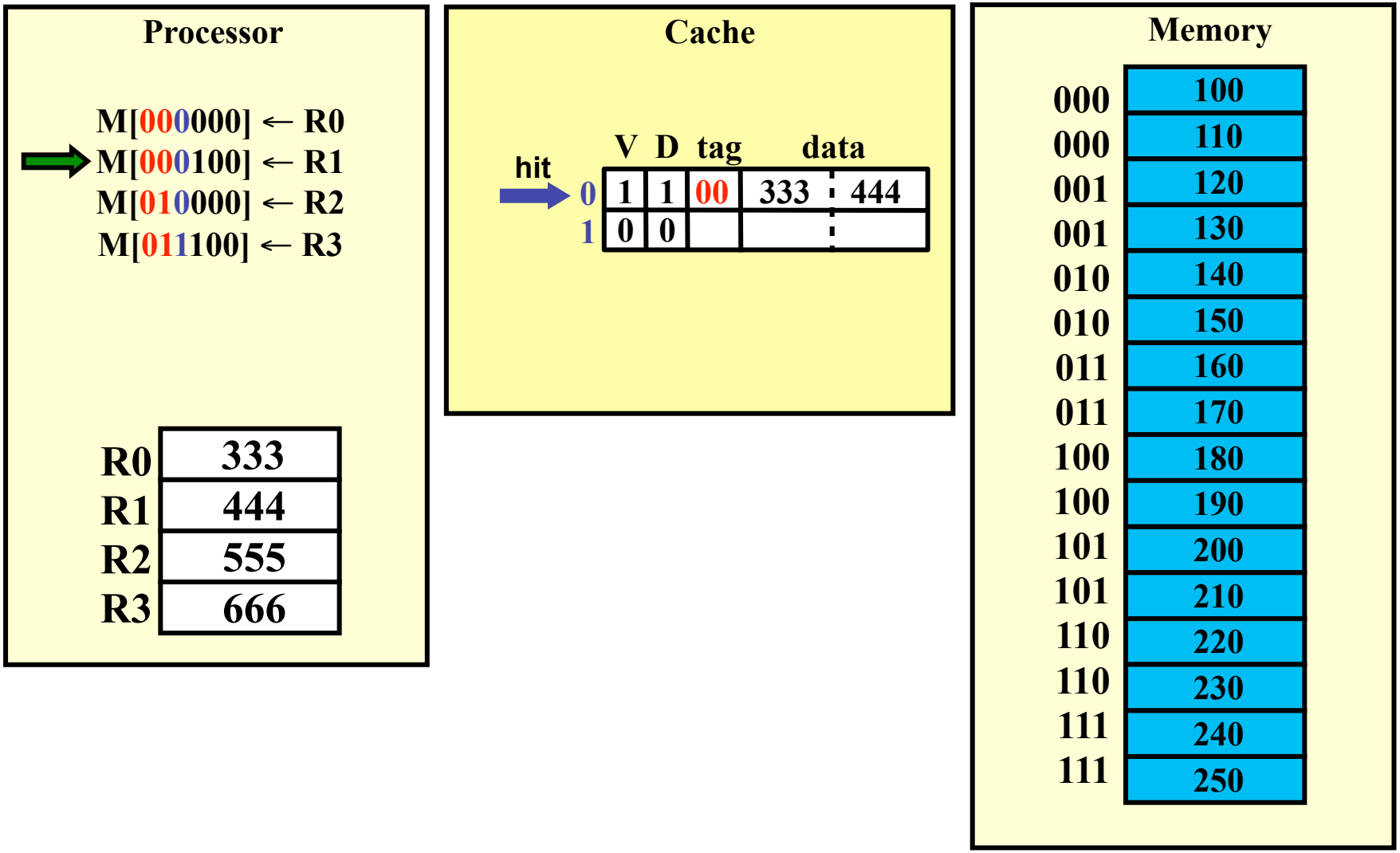
Write Back


Processor
M[000000] ← R0
M[000100] ← R1
M[010000] ← R2
M[011100] ← R3
R0 333
R1 444
R2 555
R3 666
Cache
miss
V D tag data
0 1 1 00 333 110
1 0 0
Memory
000 100
000 110
001 120
001 130
010 140
010 150
011 160
011 170
100 180
100 190
101 200
101 210
110 220
110 230
111 240
111 250

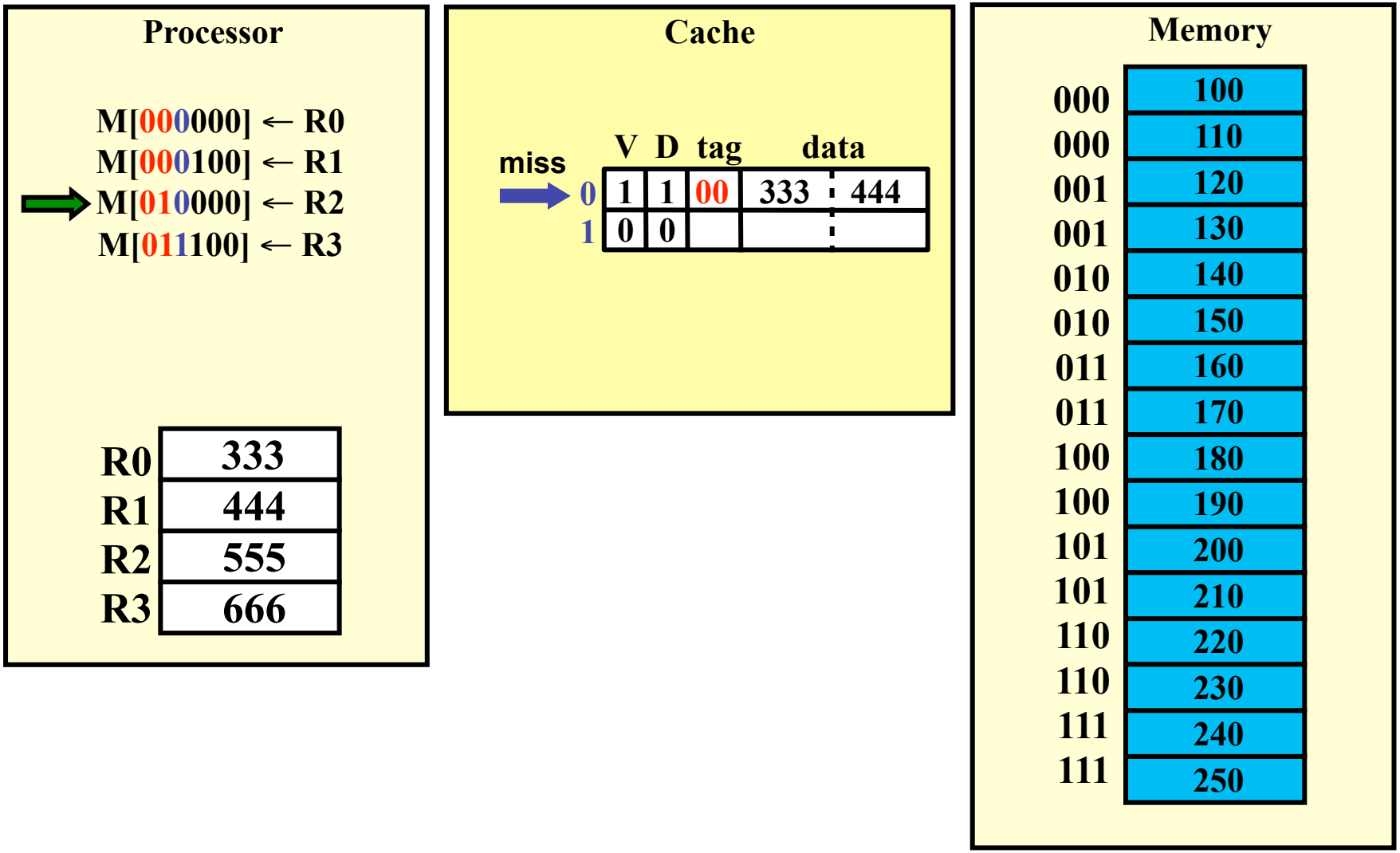
Write Back


Processor
M[000000] ← R0
M[000100] ← R1
M[010000] ← R2
M[011100] ← R3
R0 333
R1 444
R2 555
R3 666
Cache
hit
V D tag data
0 1 1 00 333 110
1 0 0
Memory
000 100
000 110
001 120
001 130
010 140
010 150
011 160
011 170
100 180
100 190
101 200
101 210
110 220
110 230
111 240
111 250

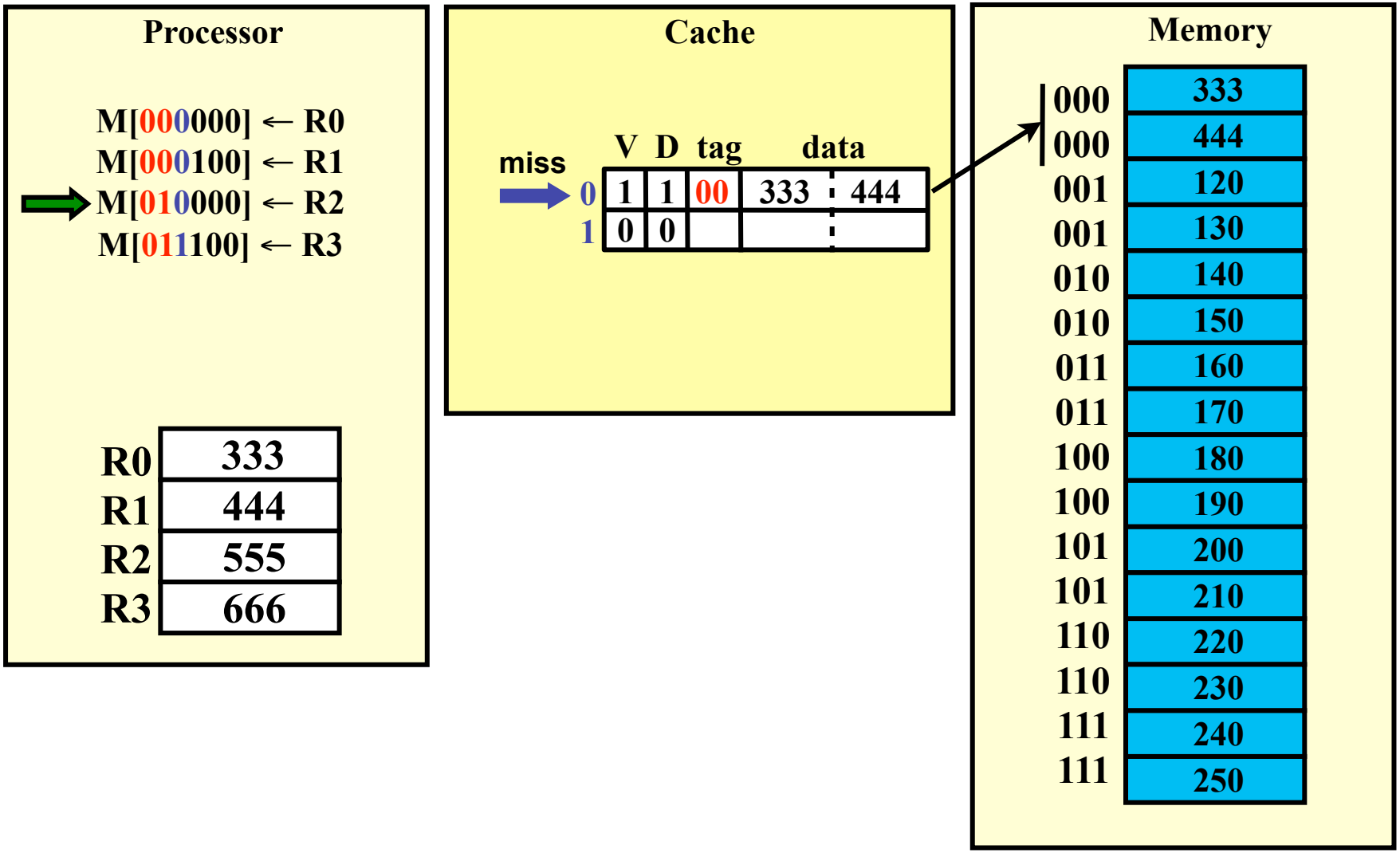
Write Back


Processor
M[000000] ← R0
M[000100] ← R1
M[010000] ← R2
M[011100] ← R3
R0 333
R1 444
R2 555
R3 666
Cache
hit
V D tag data
0 1 1 00 333 444
1 0 0
Memory
000 100
000 110
001 120
001 130
010 140
010 150
011 160
011 170
100 180
100 190
101 200
101 210
110 220
110 230
111 240
111 250

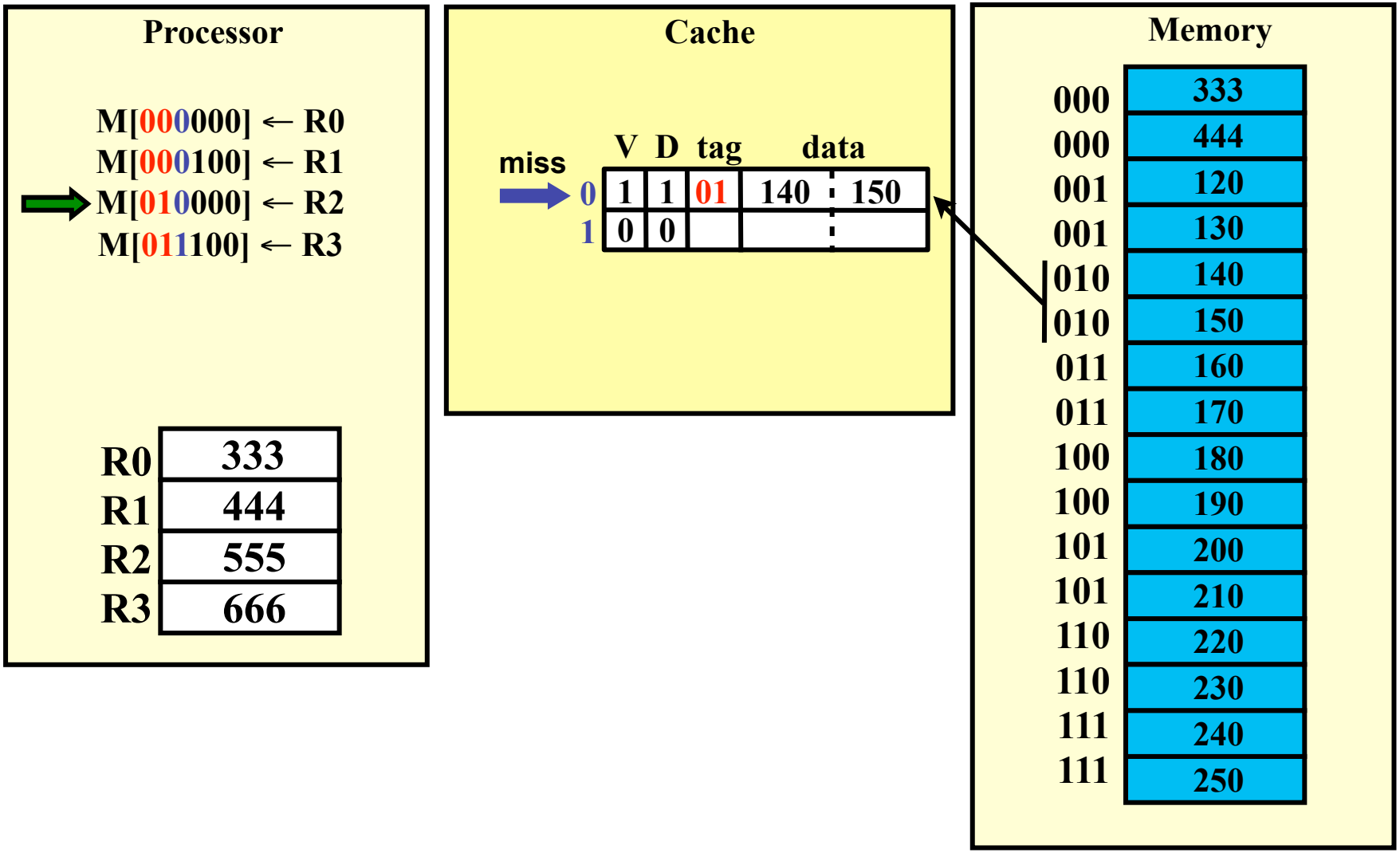
Write Back


Processor
M[000000] ← R0
M[000100] ← R1
M[010000] ← R2
M[011100] ← R3
R0 333
R1 444
R2 555
R3 666
Cache
miss
V D tag data
0 1 1 00 333 444
1 0 0
Memory
000 100
000 110
001 120
001 130
010 140
010 150
011 160
011 170
100 180
100 190
101 200
101 210
110 220
110 230
111 240
111 250

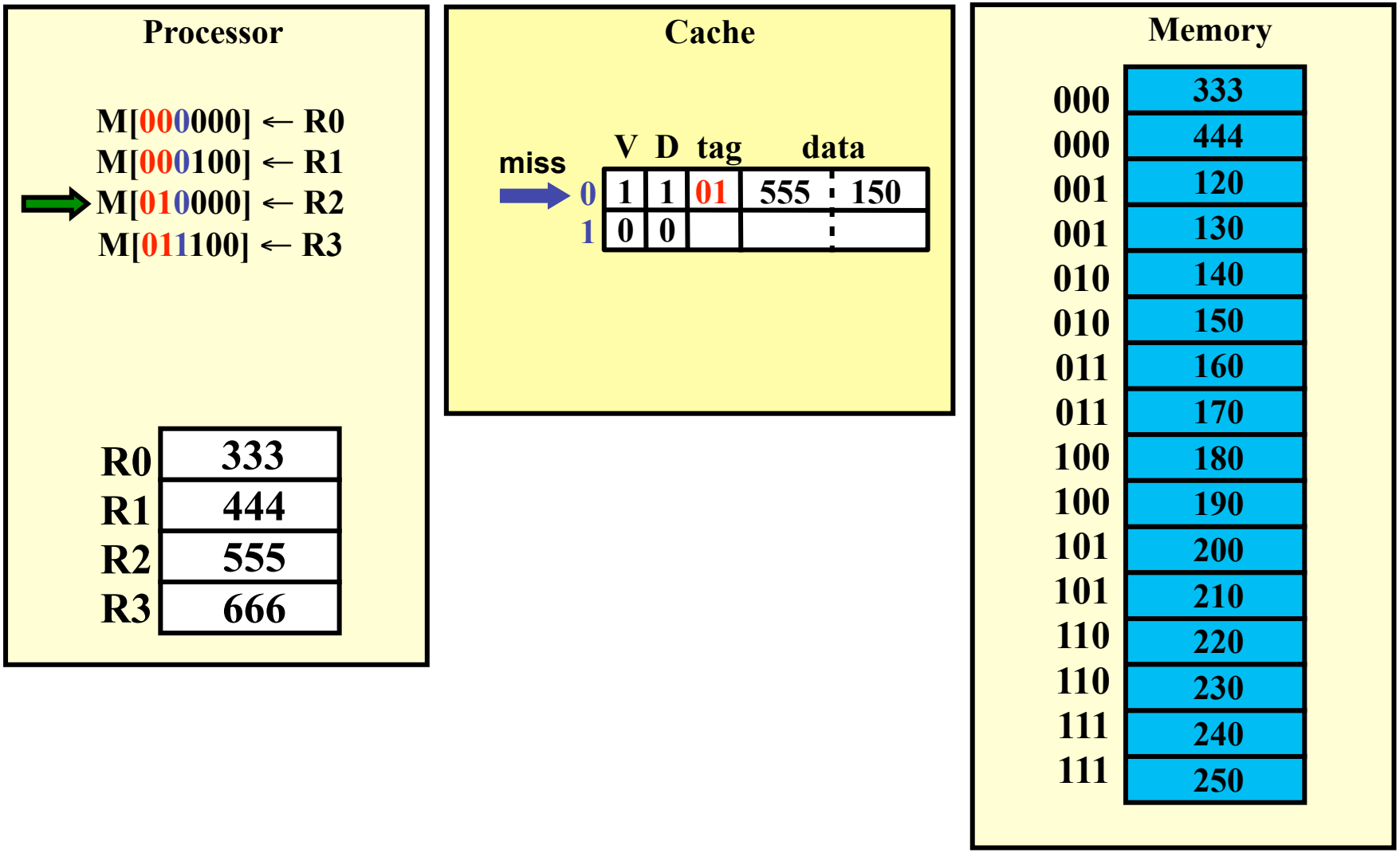
Write Back


Processor
M[000000] ← R0
M[000100] ← R1
M[010000] ← R2
M[011100] ← R3
R0 333
R1 444
R2 555
R3 666
Cache
miss
V D tag data
0 1 1 00 333 444
1 0 0
Memory
000 333
000 444
001 120
001 130
010 140
010 150
011 160
011 170
100 180
100 190
101 200
101 210
110 220
110 230
111 240
111 250

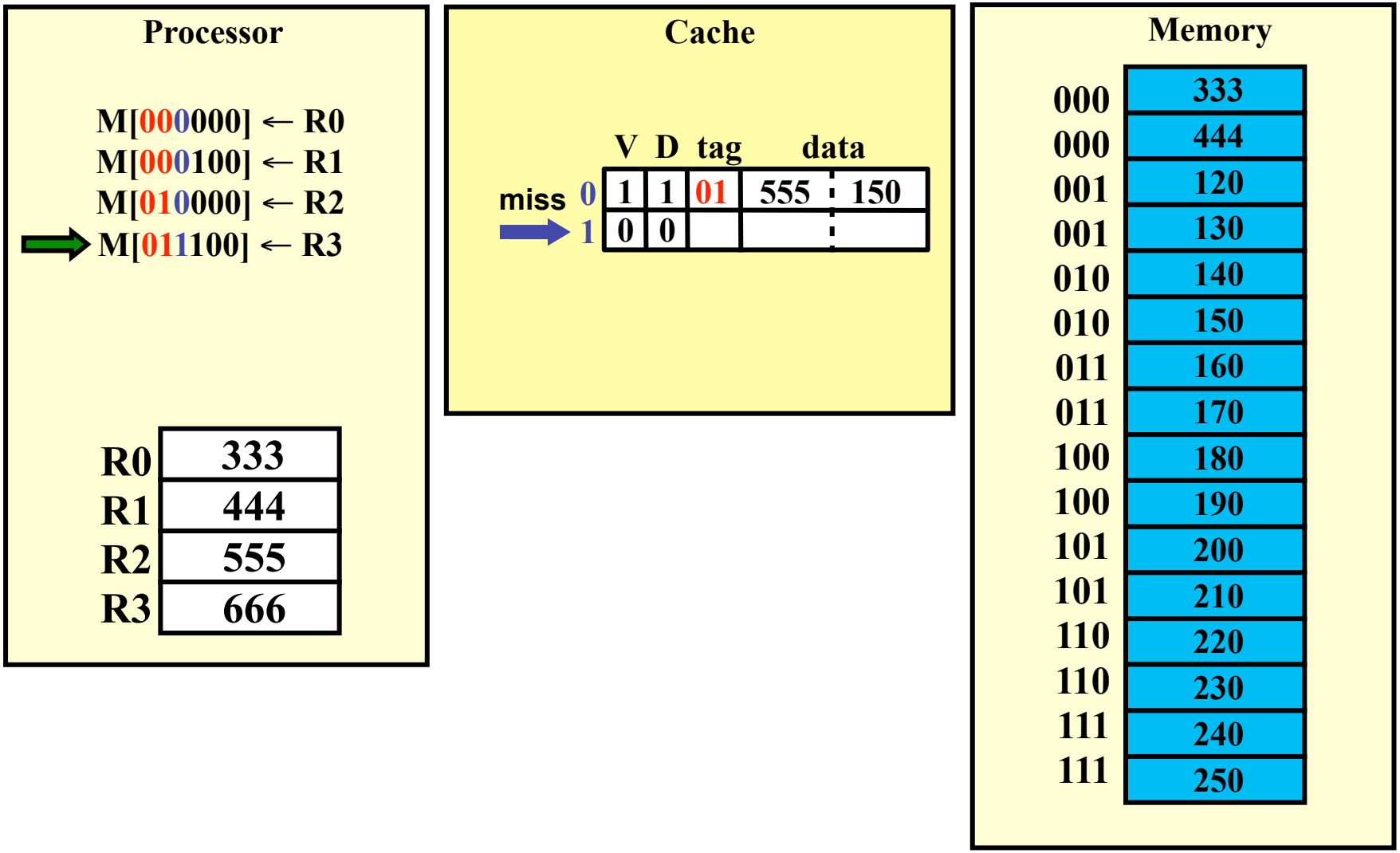
Write Back


Processor
M[000000] ← R0
M[000100] ← R1
M[010000] ← R2
M[011100] ← R3
R0 333
R1 444
R2 555
R3 666
Cache
miss
V D tag data
0 1 1 01 140 150
1 0 0
Memory
000 333
000 444
001 120
001 130
010 140
010 150
011 160
011 170
100 180
100 190
101 200
101 210
110 220
110 230
111 240
111 250

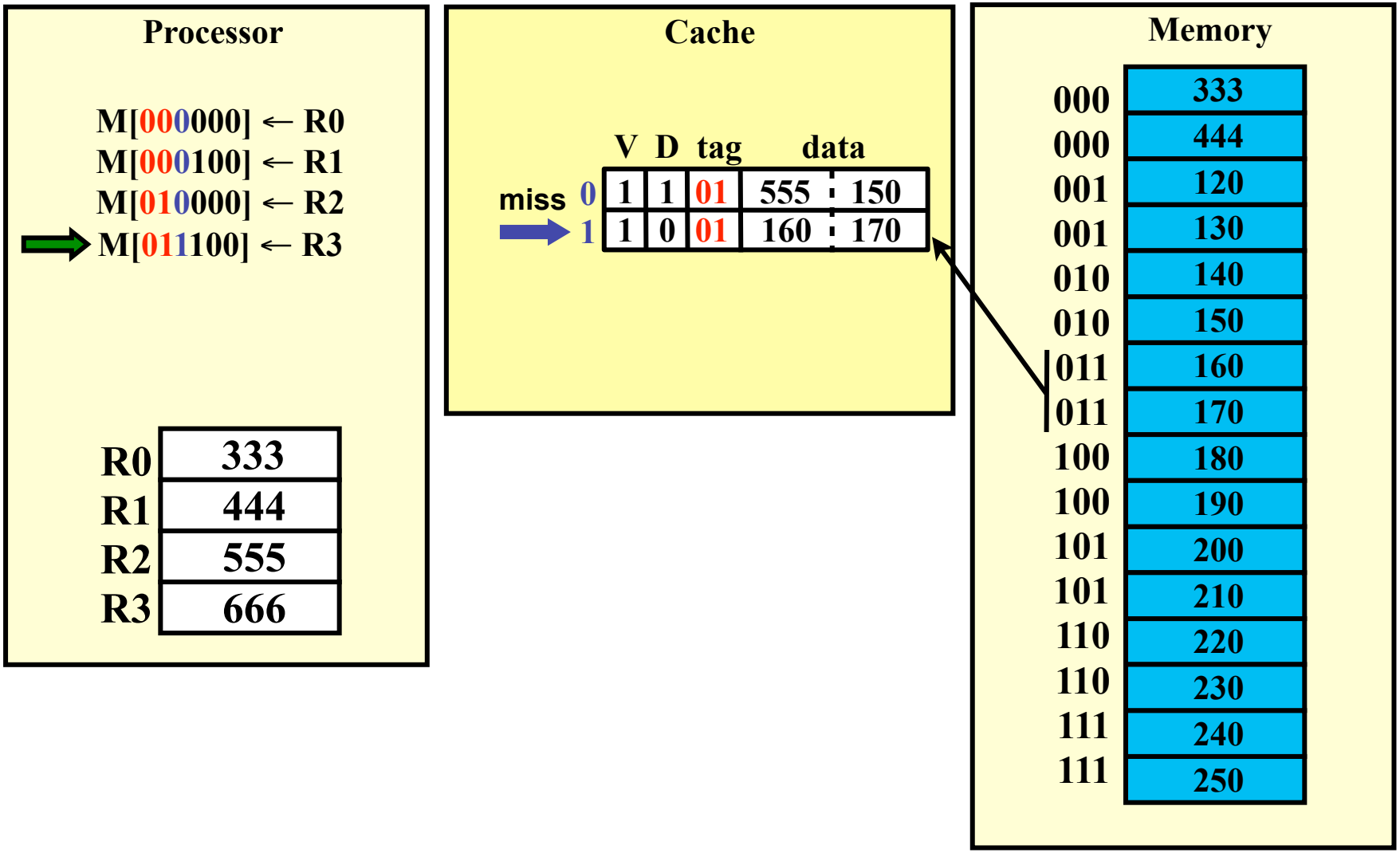
Write Back


Processor
M[000000] ← R0
M[000100] ← R1
M[010000] ← R2
M[011100] ← R3
R0 333
R1 444
R2 555
R3 666
Cache
miss
V D tag data
0 1 1 01 555 150
1 0 0
Memory
000 333
000 444
001 120
001 130
010 140
010 150
011 160
011 170
100 180
100 190
101 200
101 210
110 220
110 230
111 240
111 250

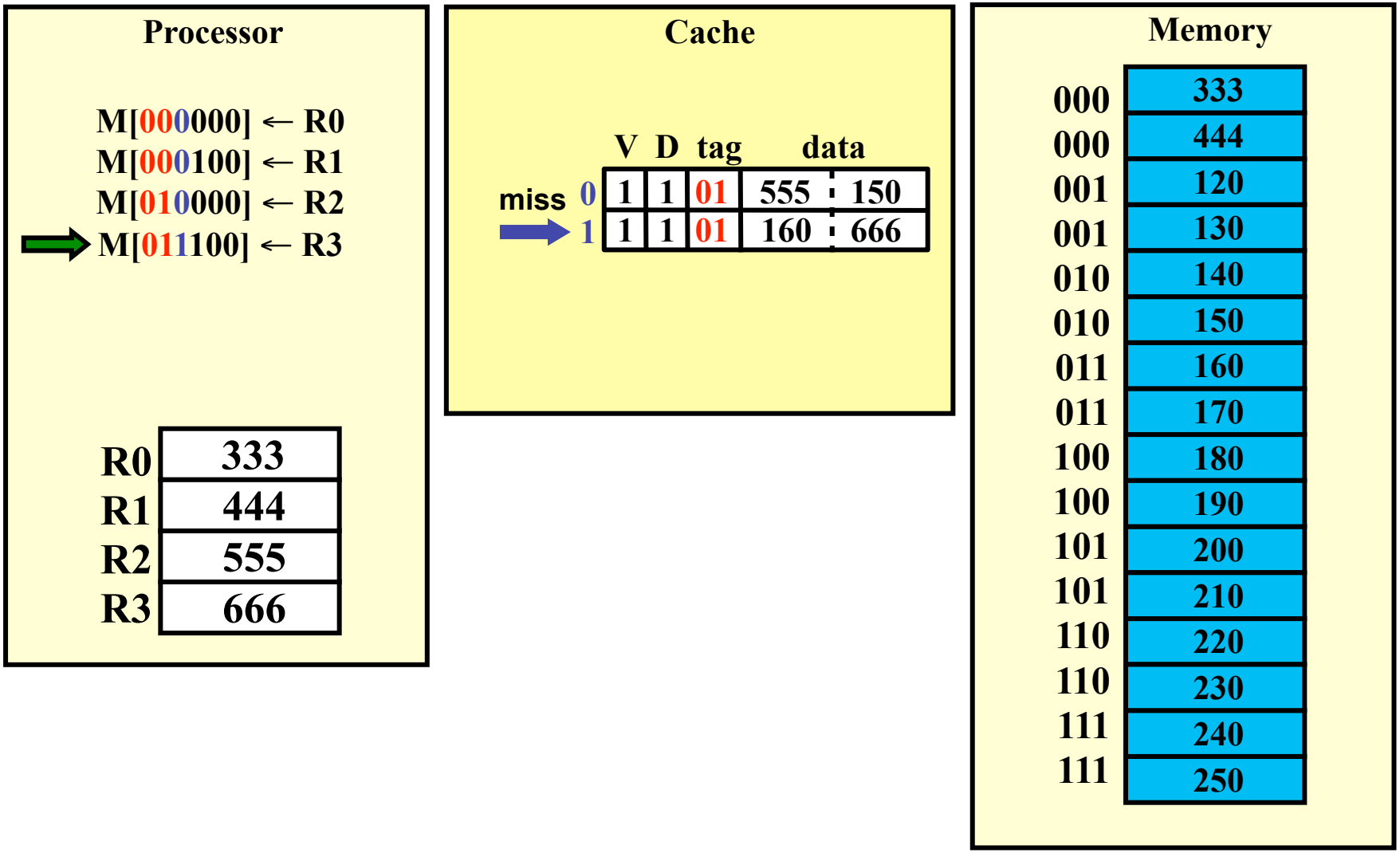
Write Back


Processor
M[000000] ← R0
M[000100] ← R1
M[010000] ← R2
M[011100] ← R3
R0 333
R1 444
R2 555
R3 666
Cache
V D tag data
miss 0 1 1 01 555 150
1 0 0
Memory
000 333
000 444
001 120
001 130
010 140
010 150
011 160
011 170
100 180
100 190
101 200
101 210
110 220
110 230
111 240
111 250

Write Back


Processor
M[000000] ← R0
M[000100] ← R1
M[010000] ← R2
M[011100] ← R3
R0 333
R1 444
R2 555
R3 666
Cache
V D tag data
miss 0 1 1 01 555 150
1 1 0 01 160 170
Memory
000 333
000 444
001 120
001 130
010 140
010 150
011 160
011 170
100 180
100 190
101 200
101 210
110 220
110 230
111 240
111 250

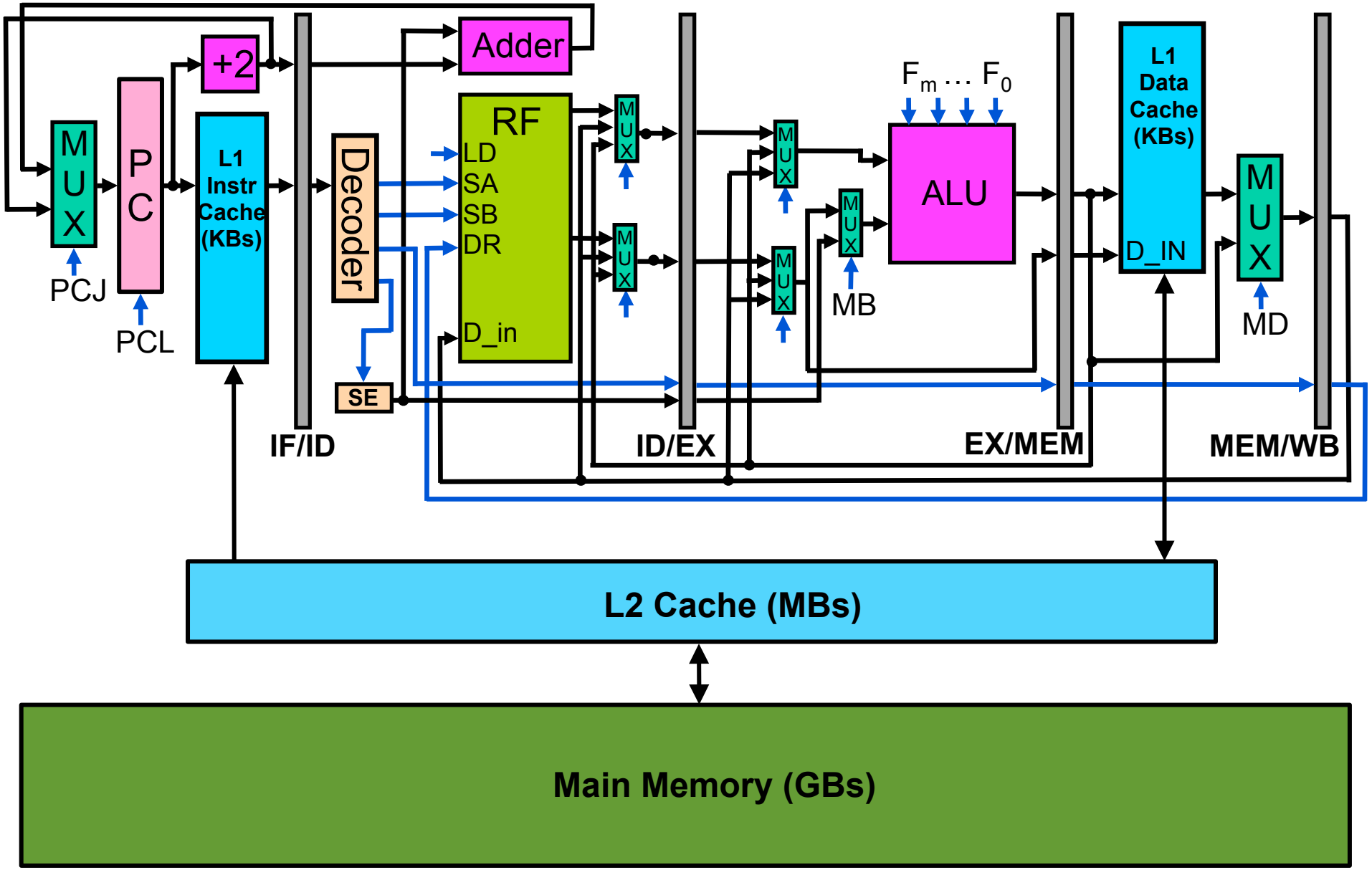
Write Back


Processor
M[000000] ← R0
M[000100] ← R1
M[010000] ← R2
M[011100] ← R3
R0 333
R1 444
R2 555
R3 666
Cache
V D tag data
miss 0 1 1 01 555 150
1 1 1 01 160 666
Memory
000 333
000 444
001 120
001 130
010 140
010 150
011 160
011 170
100 180
100 190
101 200
101 210
110 220
110 230
111 240
111 250

Cache Hierarchy

- Time to get a block from memory is so long that performance suffers even with a low miss rate
- Example: 3% miss rate, 100 cycles to memory
  - 0.03 × 100 = 3 extra cycles on average to access instructions and data
- Solution: Add another level of cache

Pipeline with a Cache Hierarchy



Cache Hierarchy

- **Level 1 (L1) instruction and data caches**
  - Small, but very fast
- **Level 2 (L2) cache handles L1 misses**
  - Larger and slower than L1, but much faster than main memory
  - Higher miss rate than L1 (less locality than L1)
- **Main memory handles L2 cache misses**
- **Assume 10 cycles to access L2, 20% L2 miss rate**
  - 0.03 × (10 + 0.2 × 100) = 0.9 extra cycles

Before Next Class

- H&H 7.5.5, 8.2

Next Time

Measuring Performance
Performance Tradeoffs