Chapter 21 Cache

Speaker: Lung-Sheng Chien

Reference: [1] David A. Patterson and John L. Hennessy, Computer Organization & Design

[2] Bruce Jacob, Spencer W. Ng, David T. Wang, Memory Systems Cache, DRAM, Disk

[3] Multiprocessor: Snooping Protocol, *www.**cs**.ucr.edu/~bhuyan/**CS213**/2004/LECTURE8.ppt*

[4] EEL 5708 High Performance Computer Architecture, Cache Coherency and Snooping Protocol, *classes.cecs.ucf.edu/**eel5708**/ejnioui/mem_hierarchy.ppt*

[5] Willian Stallings, Computer Organization and Architecture, 7th edition

[6] Intel 64 and IA-32 Architectures Software Developer's Manual, volume 1: Basic Architecture

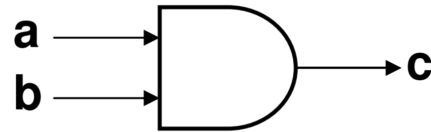
[7] Intel 64 and IA-32 Architectures Optimization Reference Manual

OutLine

- Basic of cache
  - locality
  - direct-mapped, fully associative, set associative
- Cache coherence
- False sharing
- Summary

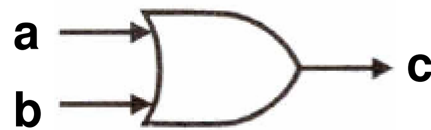
Basic logic gate

1. AND gate



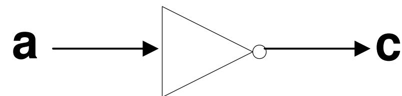
a	b	$c = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

2. OR gate



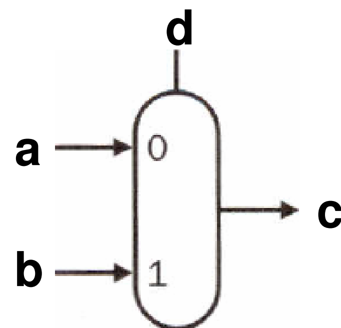
a	b	$c = a + b$
0	0	0
0	1	1
1	0	1
1	1	1

3. inverter



a	$c = \bar{a}$
0	1
1	0

4. multiplexer



d	c
0	a
1	b

Principle of locality

- Temporal locality (locality in time): if an item is referenced, it will tend to be referenced again soon.
- Spatial locality (locality in space): if an item is referenced, items whose addresses are close by will tend to be referenced soon.
- Algorithmic locality: traverse linked-list (may not be spatial locality)

```
// Multiply the two matrices together
  for ( ty = 0 ; ty < BLOCK_SIZE ; ty++ ){
    for ( tx = 0 ; tx < BLOCK_SIZE ; tx++ ){
      Csub = 0.0 ;
      for (k = 0; k < BLOCK_SIZE; ++k ){
         Asub = As[ty][k ] ;
         Bsub = Bs[k ][tx] ;
         Csub += Asub * Bsub ;
      }
      c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
      C[c + wB * ty + tx] += Csub;
    }// for tx ;
  }// for ty
```

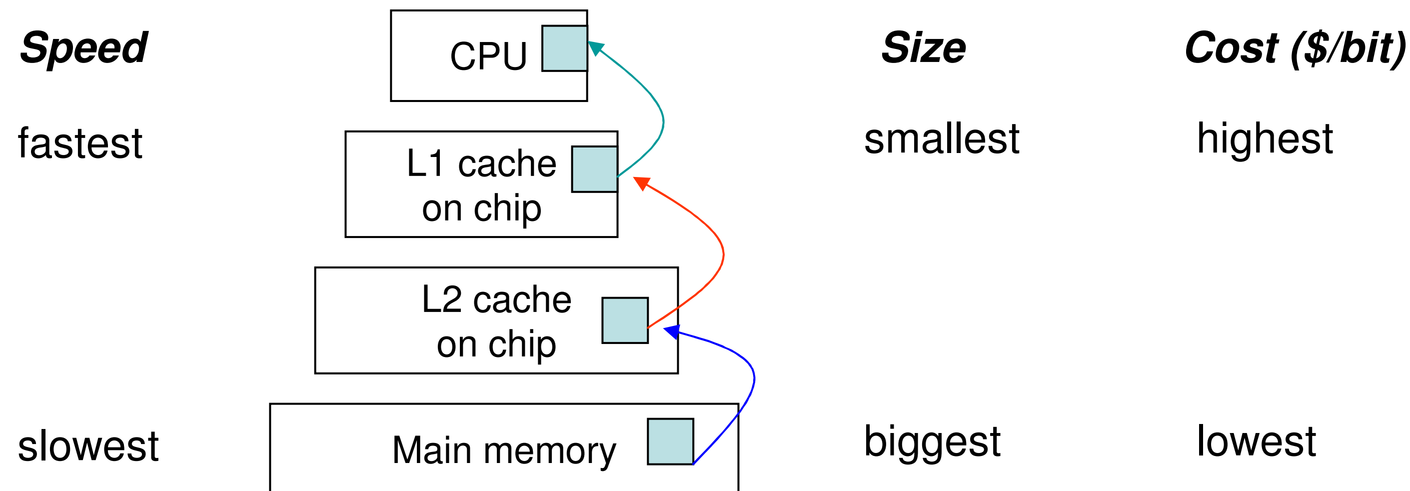
for-loop is temporal locality

array is spatial locality

Observation: temporal locality means that we don't put all program into memory whereas spatial locality means that we don't put all data into memory, hence we have "Memory Hierarchy"

Memory Hierarchy

Memory technology	Typical access time	$ per MByte in 1997
SRAM (cache)	5-25 ns	$100 - $250
DRAM (main memory)	60-120 ns	$5 - $10
Magnetic disk	10-20 ms	$0.1 - $0.2



Definition: If the data requested by processor appears in upper level, then this is called a "***hit***" , otherwise, we call "***miss***". Conventionally speaking, cache hit or cache miss

Definition: "***Hit time***" is the time to access upper level memory, including time needed to determine whether the access is a hit or a miss.

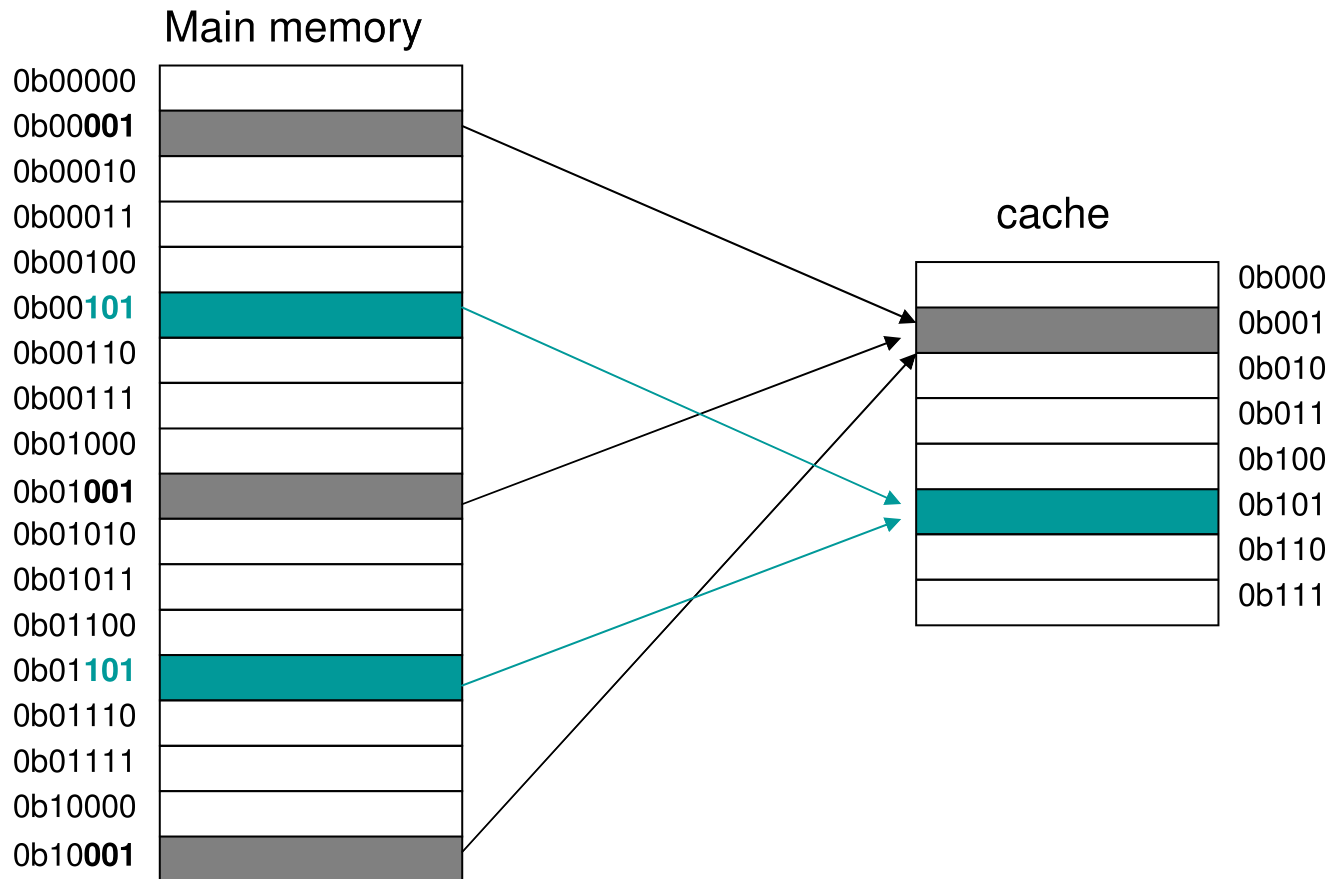
Definition: "***miss penalty***" is the time to replace a block in upper level with corresponding block from lower level, plus the time to deliver this block to processor.

Basic of cache [1]

- *Cache*: a safe place for hiding or storing things

Direct-mapped cache: each memory location is mapped to exactly one location in cache

Mapping rule: (block address) modulo (number of cache block in the cache)



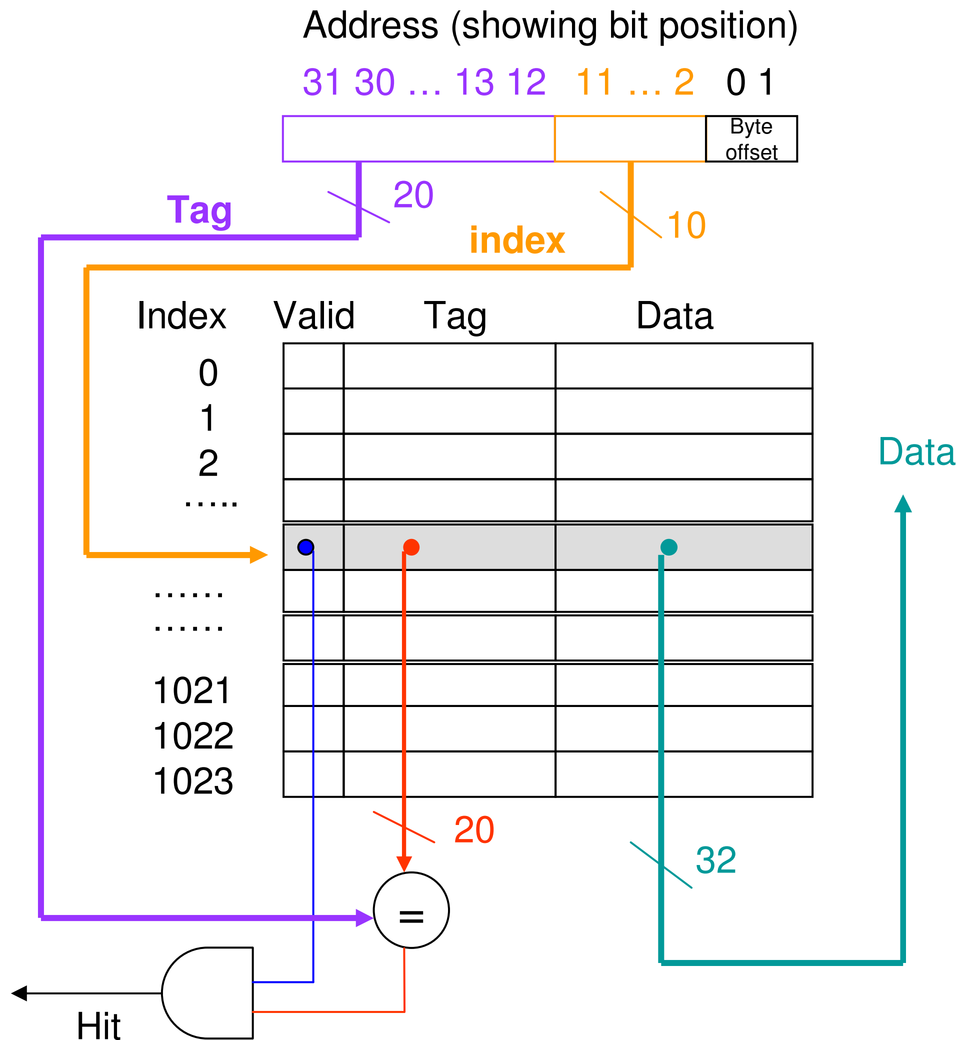
Observation: we only use 3 least significant bits to determine address.

Basic of cache [2]

Question 1: size of basic block of cache (also called ***cache line size***)

Question 2: if data is in cache, how to know whether a requested word is in the cache or not?

Question 3: if data is not in cache, how do we know?



Basic block is a word (4 byte), since each memory address binds a byte, so 4-byte require 2 bits.

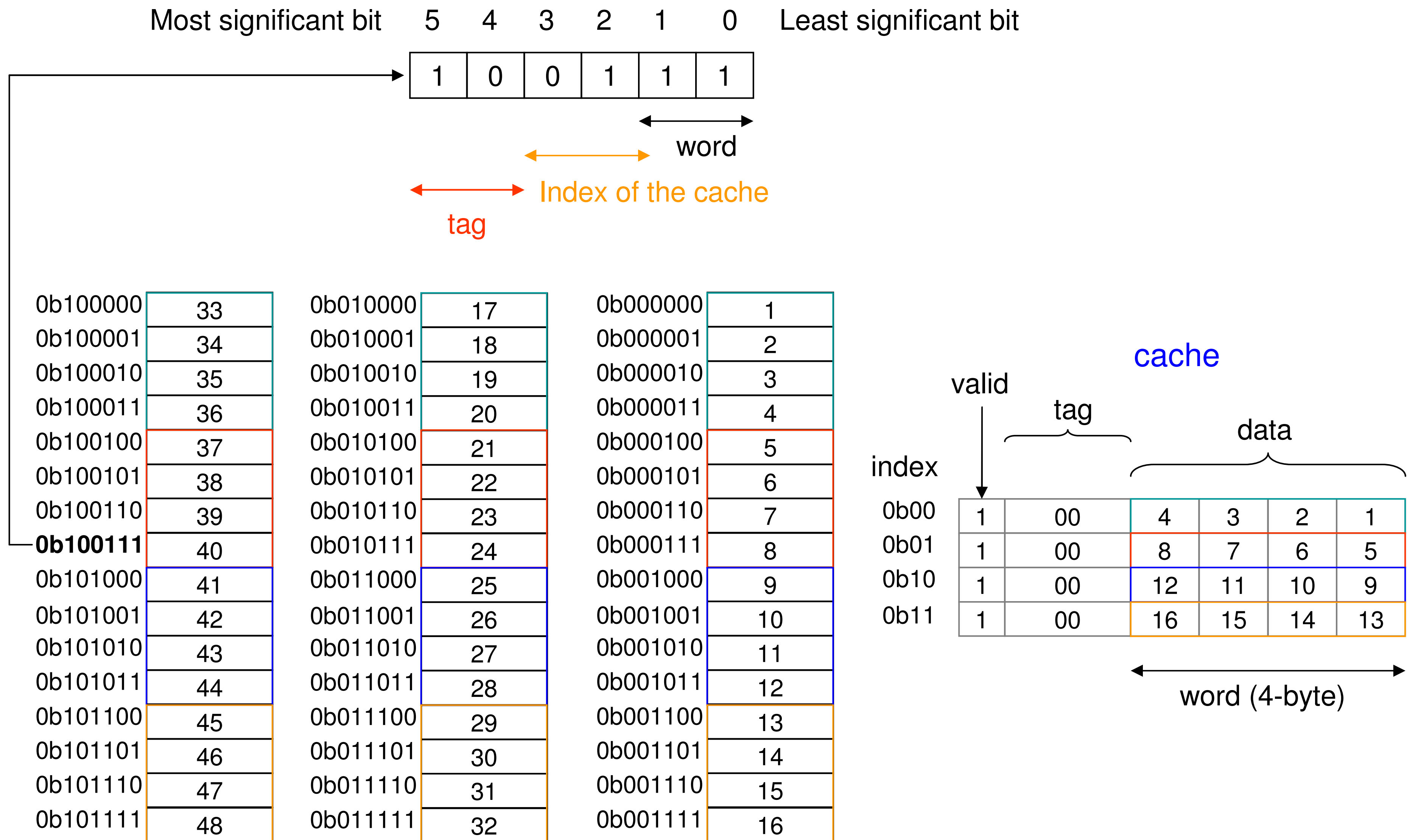
Use 10 bits to index address in cache, total number of block in the cache is 1024

Tag contains the address information required to identify whether a word in the cache corresponding to the requested word.

Valid bit: indicates whether an entry contains a valid address.

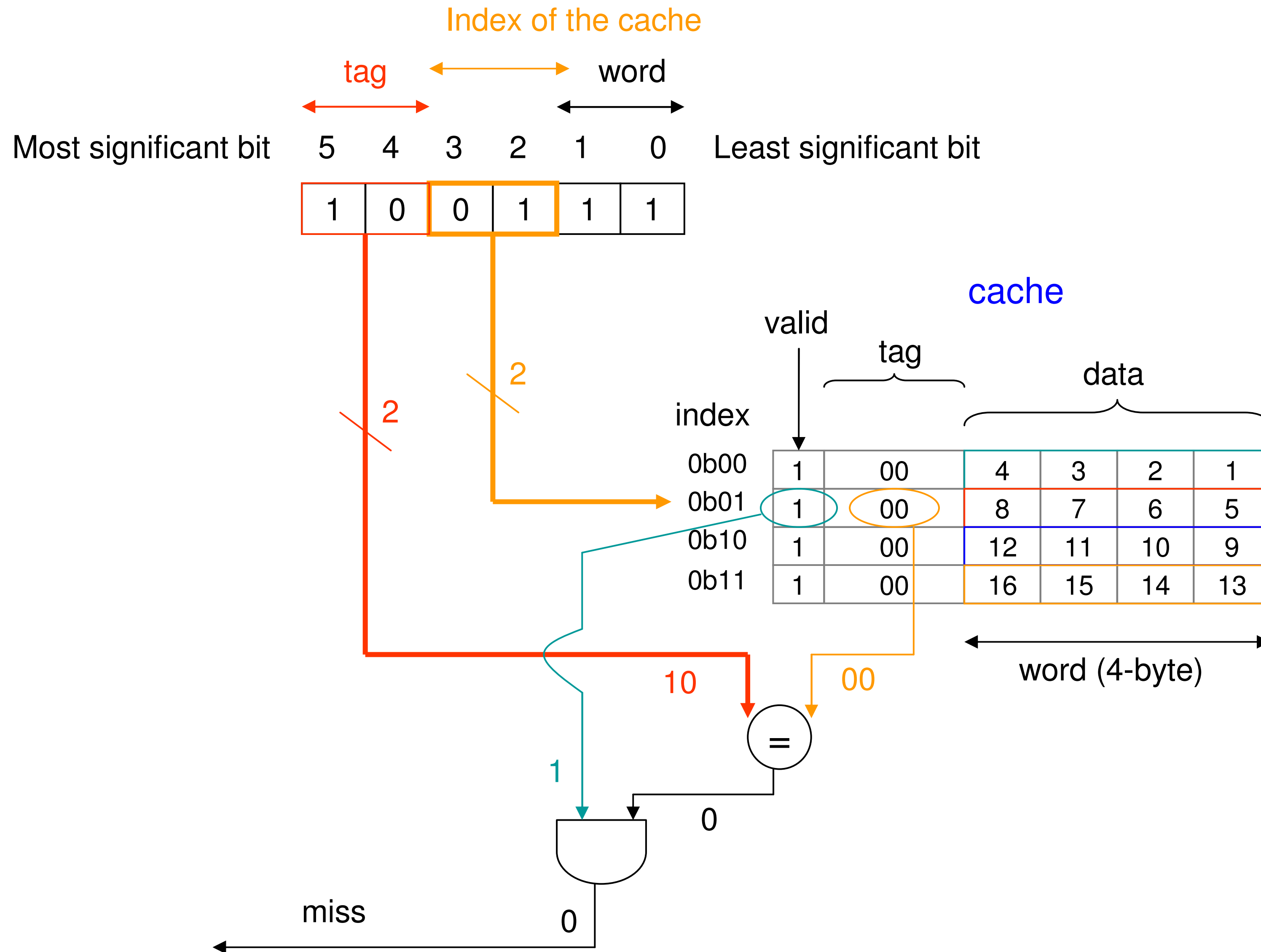
Basic of cache [3]

Configuration: Basic block of cache is ***word (4-byte)***, and cache has ***4*** blocks



Basic of cache [4]

Question 4: is data with address 0b100111 in the cache?



Example of direct-mapped cache [1]

Initial state of cache

index	V	Tag	Data
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

1. Access 0b10110

index	V	Tag	Data
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	1	10	Memory(0b10110)
111	0		

2. Access 0b11010

index	V	Tag	Data
000	0		
001	0		
010	1	11	Memory(0b11010)
011	0		
100	0		
101	0		
110	1	10	Memory(0b10110)
111	0		

3. Access 0b10110

index	V	Tag	Data
000	0		
001	0		
010	1	11	Memory(0b11010)
011	0		
100	0		
101	0		
110	1	10	Memory(0b10110)
111	0		

Example of direct-mapped cache [2]

index	V	Tag	Data
000	0		
001	0		
010	1	11	Memory(0b11010)
011	0		
100	0		
101	0		
110	1	10	Memory(0b10110)
111	0		

4. Access 0b10000

index	V	Tag	Data
000	1	10	Memory(0b10000)
001	0		
010	1	11	Memory(0b11010)
011	0		
100	0		
101	0		
110	1	10	Memory(0b10110)
111	0		

5. Access 0b00011

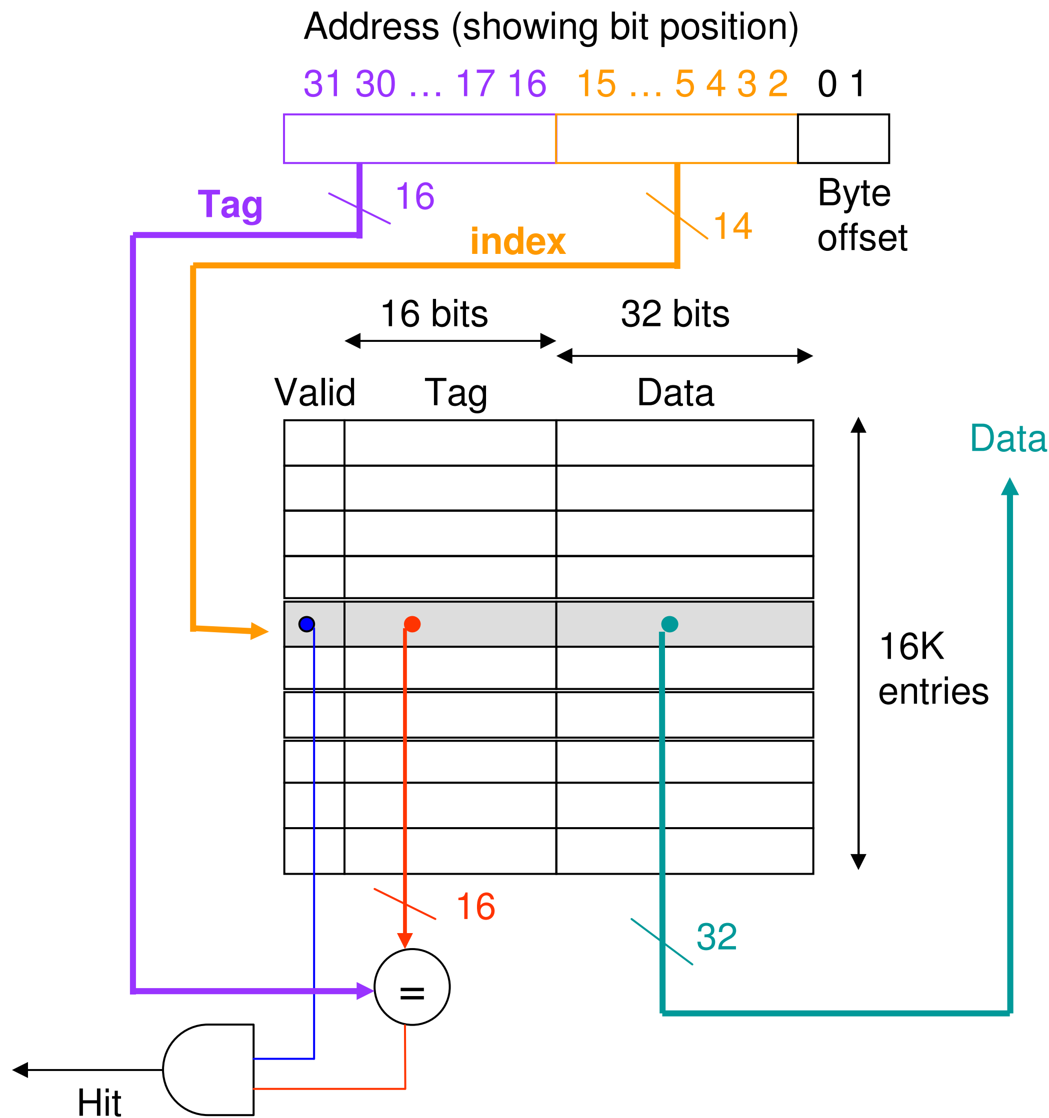
index	V	Tag	Data
000	1	10	Memory(0b10000)
001	0		
010	1	11	Memory(0b11010)
011	0	00	Memory(0b00011)
100	0		
101	0		
110	1	10	Memory(0b10110)
111	0		

6. Access 0b10010

index	V	Tag	Data
000	1	10	Memory(0b10000)
001	0		
010	1	10	Memory(0b10010)
011	0	00	Memory(0b00011)
100	0		
101	0		
110	1	10	Memory(0b10110)
111	0		

Advantage of spatial locality [1]

64kB cache with a word (4 byte) as block size



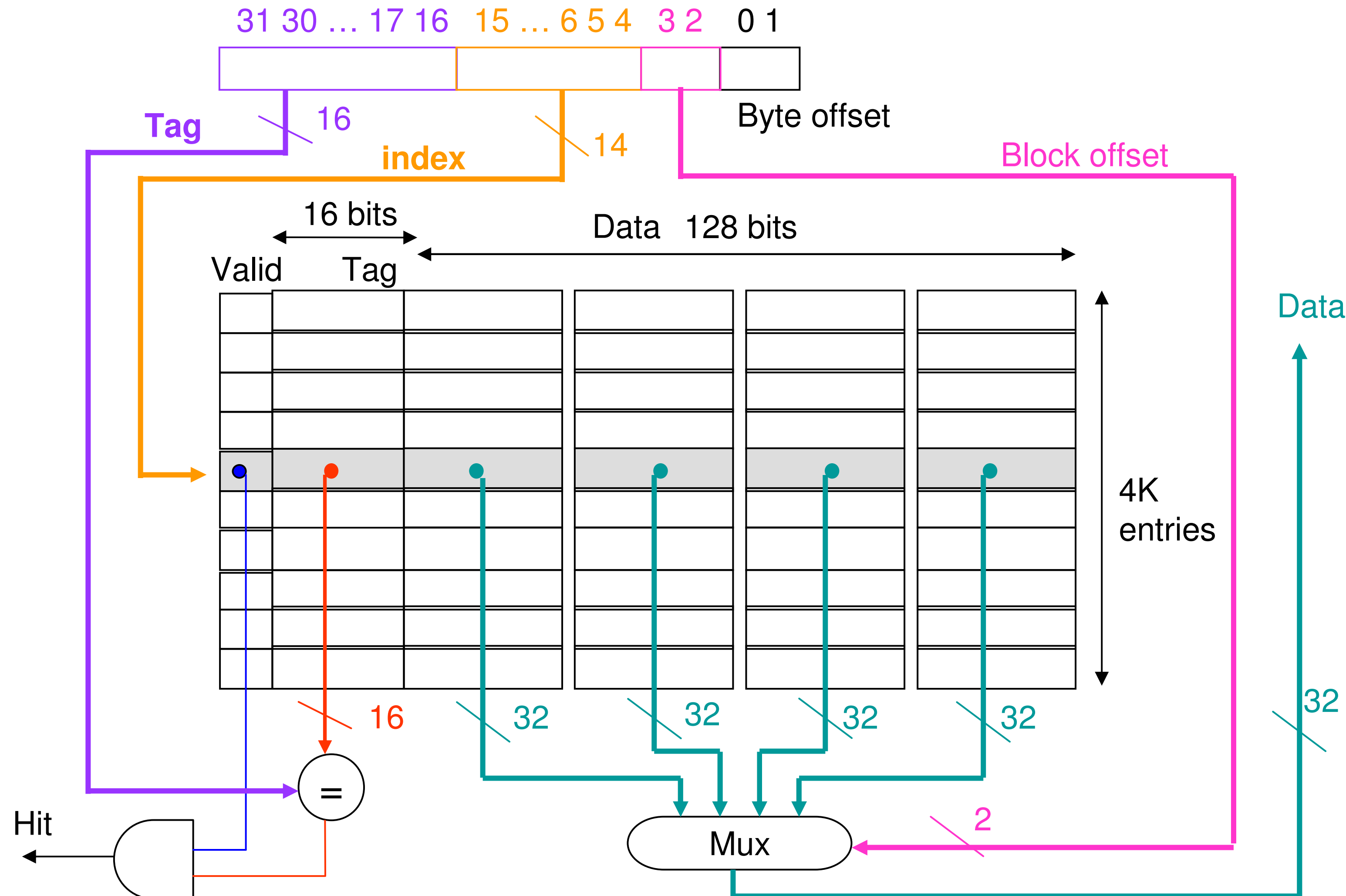

```
for ( i = 0; i < hA; ++i){
    for (j = 0; j < wB; ++j) {
        sum = 0.0 ;
        for (k = 0; k < wA; ++k) {
            a = A[i * wA + k];
            b = B[k * wB + j];
            sum += a * b;
        }// for k
        C[i * wB + j] = sum;
    }// for j
}// for i
```

To take advantage of spatial locality, we want to have a cache block that is larger than one word in length, why?

When a miss occurs, we will fetch multiple words that are adjacent and carry a high probability of being needed shortly.

Advantage of spatial locality [2]

64kB cache using 4 words (16 byte) blocks



1. Total number of blocks in cache is 4K, not 16K
2. We need signal block offset (2 bits) to determine which word we need
3. Mapping rule: (block address) modulo (number of cache block in the cache)

Advantage of spatial locality [3]

Exercise 1: consider a cache with 64 blocks and a block size of 16 bytes. What block number does byte address 1203 map to (assume 12-bit address)?

$$1203 = 4 \times 16^2 + 11 \times 16 + 3 = 0x4B3 = 0b0100\ 1011\ 0011$$

1. mask to 16 bytes (a block)

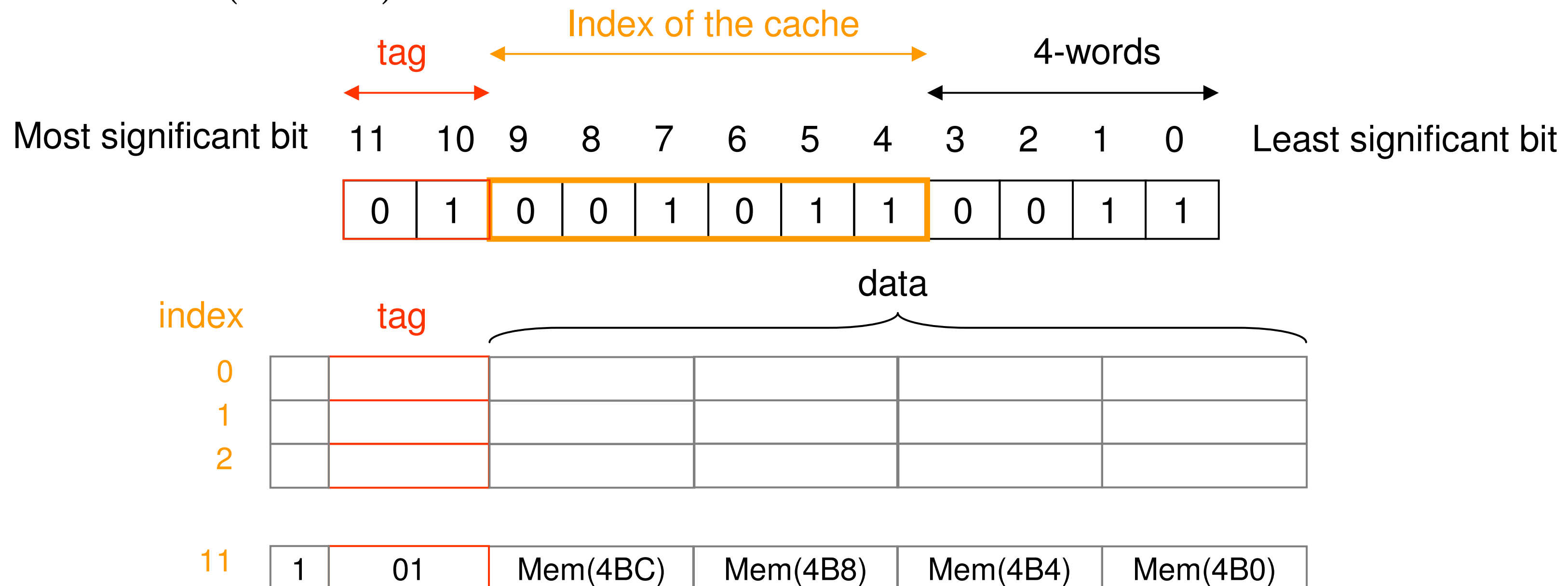
$$1203 \text{ and } FF0 = 0x4B3 \cdot FF0 = 0x4B0$$

2. Find block address

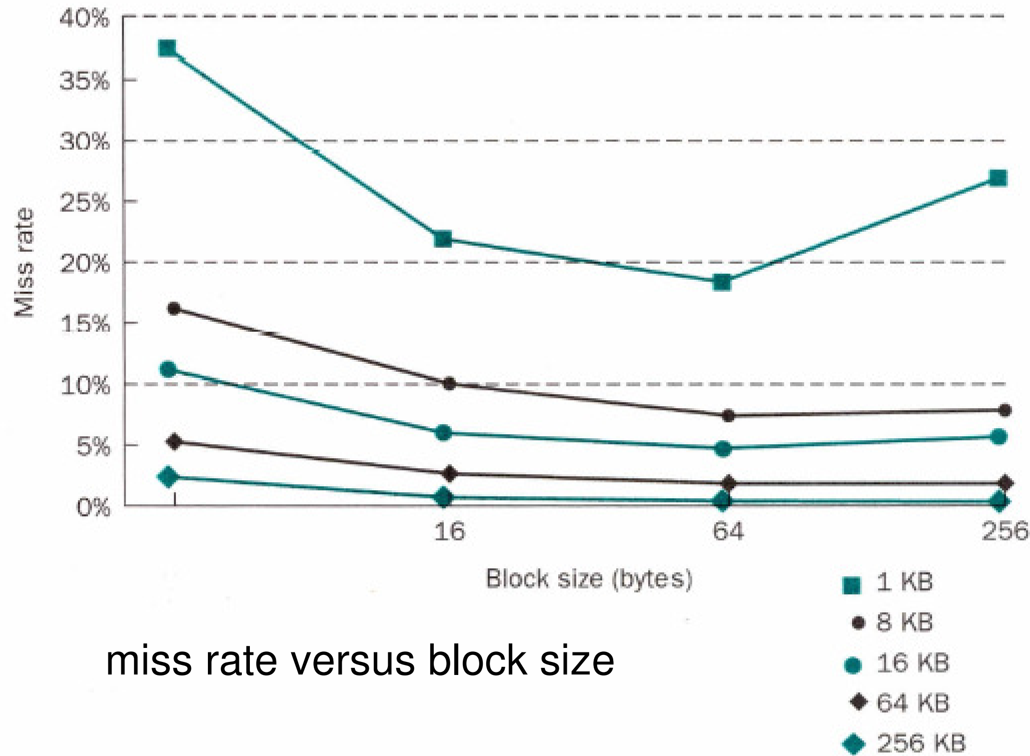
$$0x4B0 >> 4 = 0x4B = 75$$

3. mapping rule: (block address) modulo (number of cache block in the cache)

$$75 \equiv 11 (\text{mod } 64)$$



Advantage of spatial locality [4]



miss rate versus block size

Exercise 2: take a simple for-loop, discuss lager block size can reduce miss rate

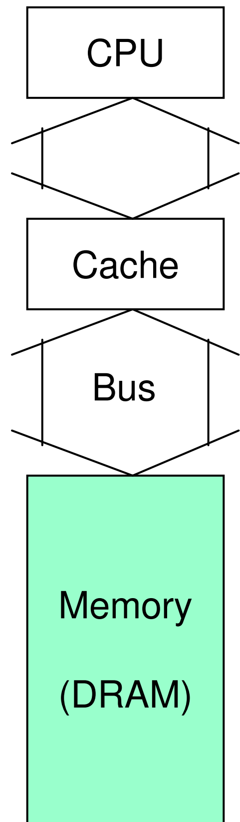
Question 5: why does miss rate increase when block size is more than 64 bytes?

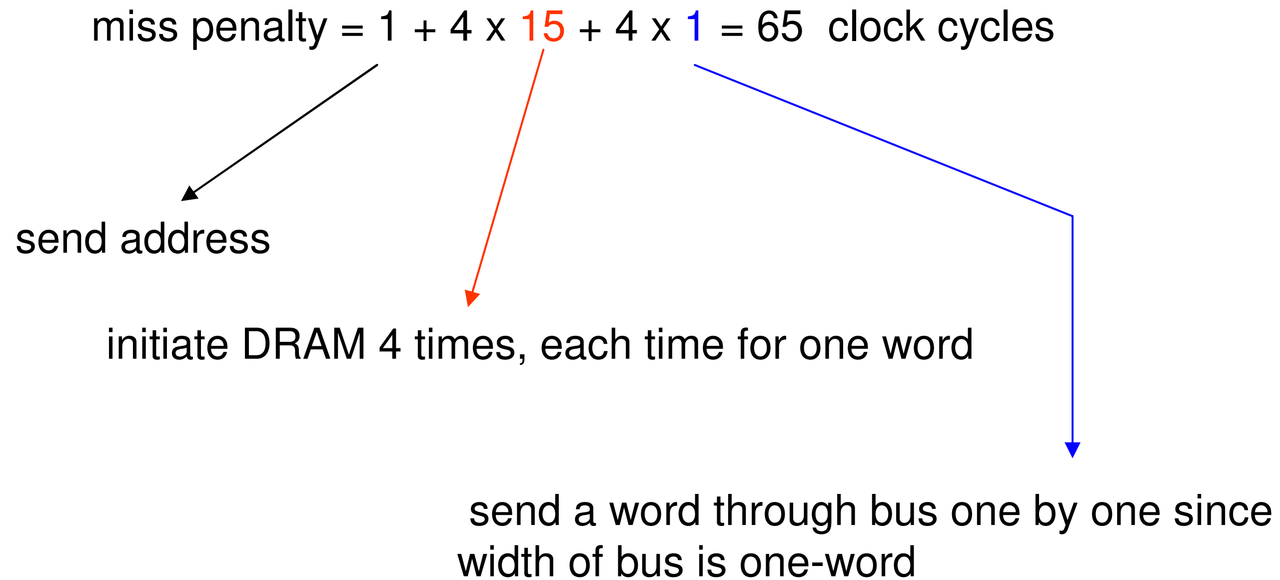
Question 6: what is trend of mss penalty when block size is getting larger?
(miss penalty is determined by the time required to fetch the block from the next lower level of memory hierarchy and load it into the cache. The time to fetch the block includes 1. latency to first word and 2. transfer time for the rest of the block)

Memory system versus cache [1]

Assumption :

- 1 clock cycle to send the address (Cache → DRAM)
- 15 clock cycles for each DRAM access initiated
- 1 clock cycle to send a word of data (depend on width of the bus)
- Cache block is 4-words

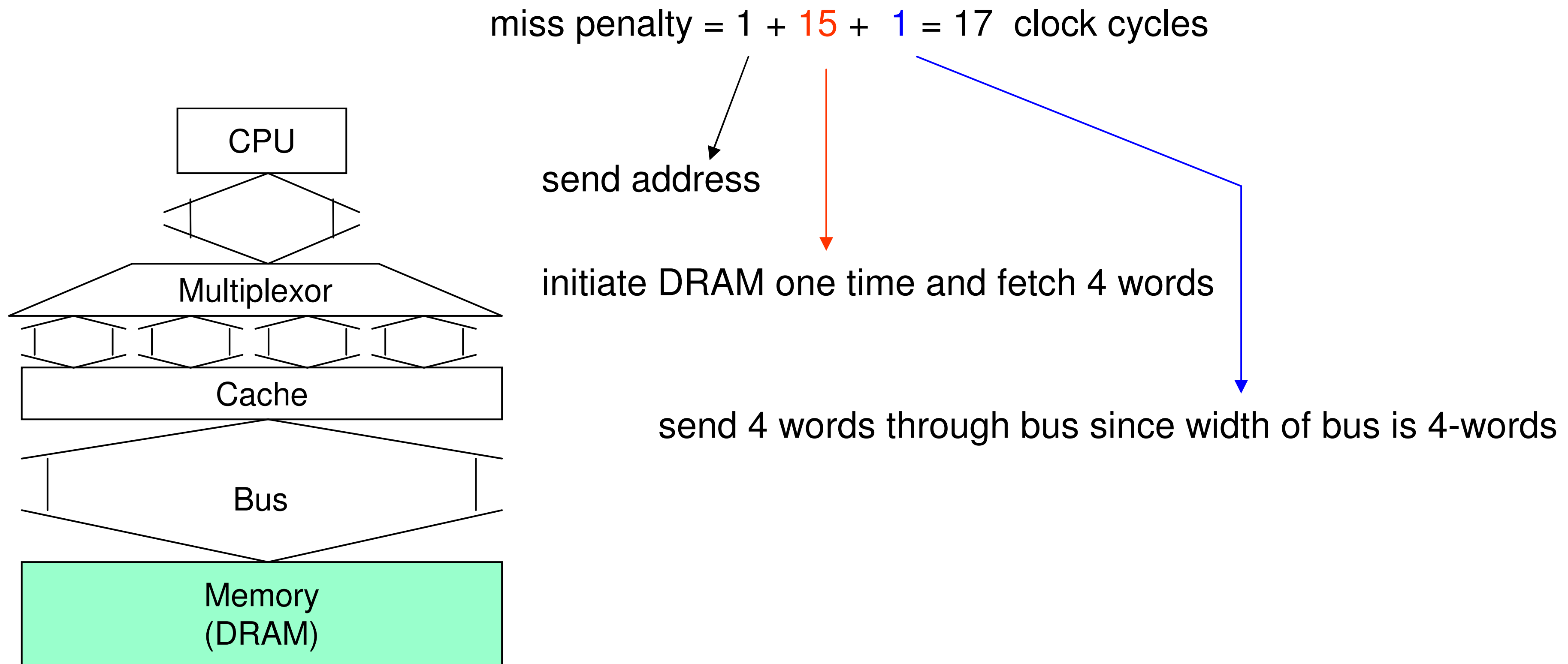




Number of bytes transferred per clock cycle for a single miss = $\frac{4 \times 4}{65} = 0.25$

one-word-wide memory organization

Memory system versus cache [2]

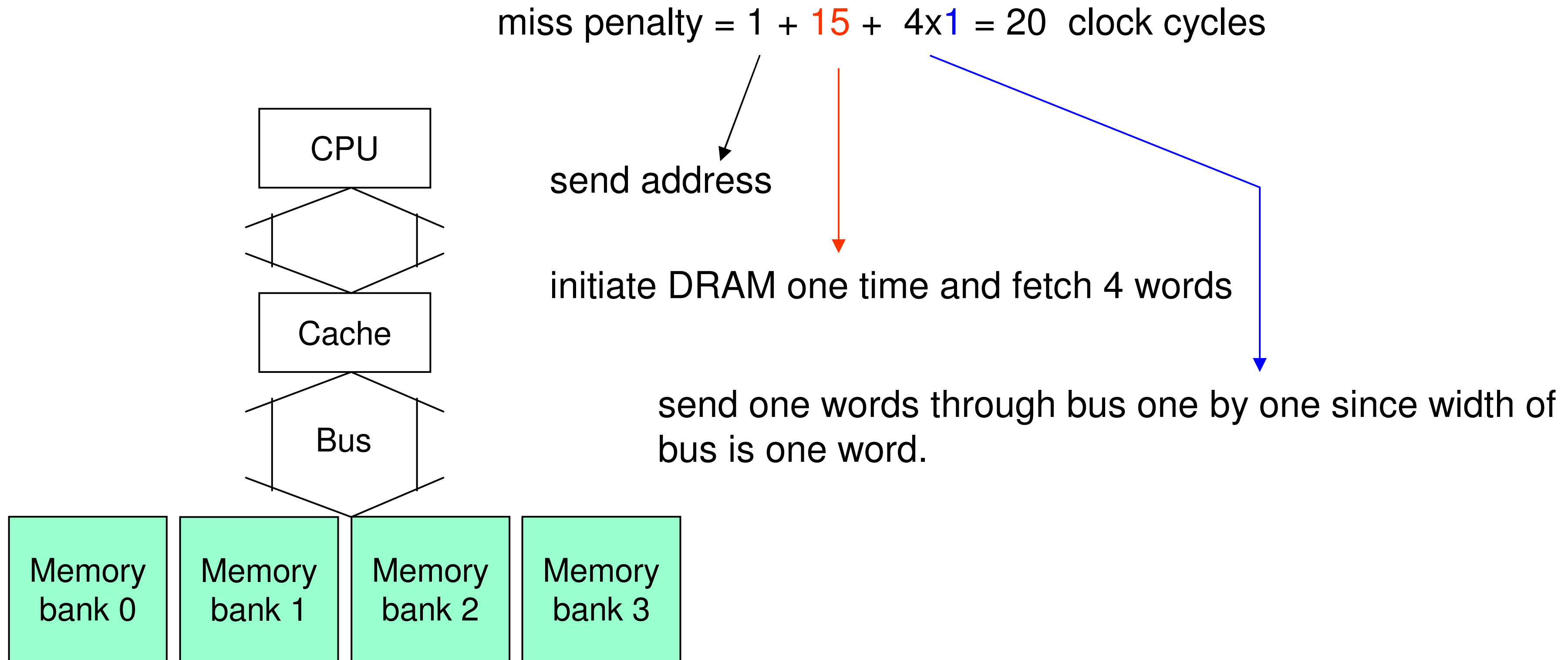


4-word-wide memory organization

Number of bytes transferred per clock cycle for a single miss = $\frac{4 \times 4}{17} = 0.94$

Question 7: what is drawback of wide memory organization ?

Memory system versus cache [3]

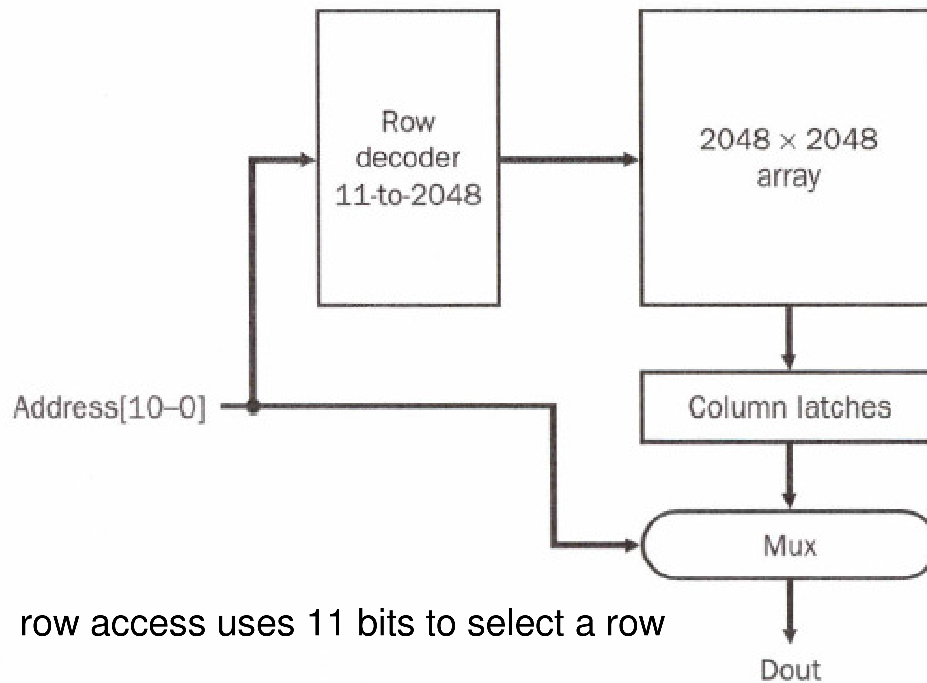


interleaved memory organization

Number of bytes transferred per clock cycle for a single miss = $\frac{4 \times 4}{20} = 0.8$

Question 8: what is difference between wide memory organization and interleaved memory organization ?

Two level decoder of DRAM [3]



row access uses 11 bits to select a row

1. row access chooses one row and activates corresponding word line
2. contents of all the columns in the active row are stored in a set of latches (page mode)
3. column access selects data from the column latches

Year introduced	Chip size	$ per MB	Total access time to a new row/column	Column access time to existing row
1980	64 Kbit	1500	250 ns	150 ns
1983	256 Kbit	500	185 ns	100 ns
1985	1 Mbit	200	135 ns	40 ns
1989	4 Mbit	50	110 ns	40 ns
1992	16 Mbit	15	90 ns	30 ns
1996	64 Mbit	10	60 ns	20 ns

Improve cache performance

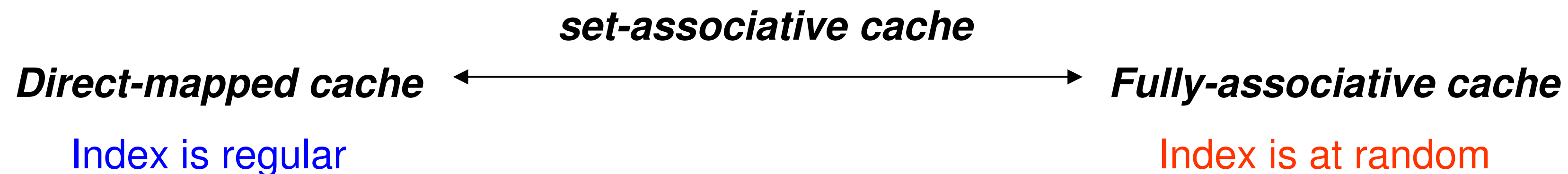
- Reduce miss rate:
  reduce probability that two different memory blocks will contend for the same cache location.
- Reduce miss penalty:
  add an additional level of hierarchy, say L1 cache, L2 cache and L3 cache.

Direct-mapped cache: each memory location is mapped to exactly one location in cache

Mapping rule: (block address) modulo (number of cache block in the cache)

Fully-associative cache: each block in memory may be associated with any entry in the cache.

Mapping rule: exhaust search each entry in cache to find an empty entry



Example of fully-associative cache [1]

Initial state of cache

index	V	Tag	Data
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

1. Access 0b10110

index	V	Tag	Data
000	1	10	Memory(0b10110)
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

2. Access 0b11010

index	V	Tag	Data
000	1	10	Memory(0b10110)
001	1	11	Memory(0b11010)
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

3. Access 0b10110

index	V	Tag	Data
000	1	10	Memory(0b10110)
001	1	11	Memory(0b11010)
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

Example of fully-associative cache [2]

index	V	Tag	Data
000	1	10	Memory(0b10110)
001	1	11	Memory(0b11010)
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

4. Access 0b10000

index	V	Tag	Data
000	1	10	Memory(0b10110)
001	1	11	Memory(0b11010)
010	1	10	Memory(0b10000)
011	0		
100	0		
101	0		
110	0		
111	0		

5. Access 0b00011

index	V	Tag	Data
000	1	10	Memory(0b10110)
001	1	11	Memory(0b11010)
010	1	10	Memory(0b10000)
011	1	00	Memory(0b00011)
100	0		
101	0		
110	0		
111	0		

6. Access 0b10010

index	V	Tag	Data
000	1	10	Memory(0b10110)
001	1	11	Memory(0b11010)
010	1	10	Memory(0b10000)
011	1	00	Memory(0b00011)
100	1	10	Memory(0b10010)
101	0		
110	0		
111	0		

set-associative cache

One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

eight-way set associative (fully-associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

A set-associative cache with ***n*** locations for a block is called an ***n-way*** set-associative cache

Mapping rule: (block address) modulo (number of sets in the cache)

Associativity in cache [1]

Example: there are three small caches, each consisting of 4 one-word blocks. One cache is fully-associative, a second is two-way set associative and the third is direct mapped. Find the number of misses for each cache organization given the following sequence of block addresses: 0, 8, 0, 6, 8.

One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				

fully-associative

Tag	Data	Tag	Data	Tag	Data	Tag	Data

1 ***Direct-mapped cache*** : (block address) modulo (number of block in the cache)

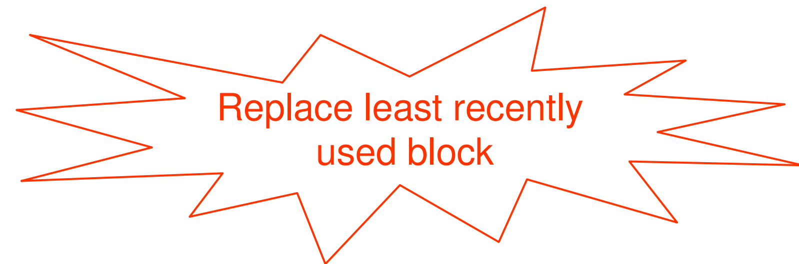
Block address	Cache block
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

Associativity in cache [2]

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	Miss	Memory[0]			
8	Miss	Memory[8]			
0	Miss	Memory[0]			
6	Miss	Memory[0]		Memory[6]	
8	Miss	Memory[8]		Memory[6]	

2 ***two-way associative cache*** : (block address) modulo (number of sets in the cache)

Block address	Cache set
0	(0 modulo 2) = 0
6	(6 modulo 2) = 0
8	(8 modulo 2) = 0



Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	Miss	Memory[0]			
8	Miss	Memory[0]	Memory[8]		
0	Hit	Memory[0]	Memory[8]		
6	Miss	Memory[0]	Memory[6]		
8	Miss	Memory[8]	Memory[6]		

Associativity in cache [3]

3 ***Fully associative cache*** : exhaust search for empty entry

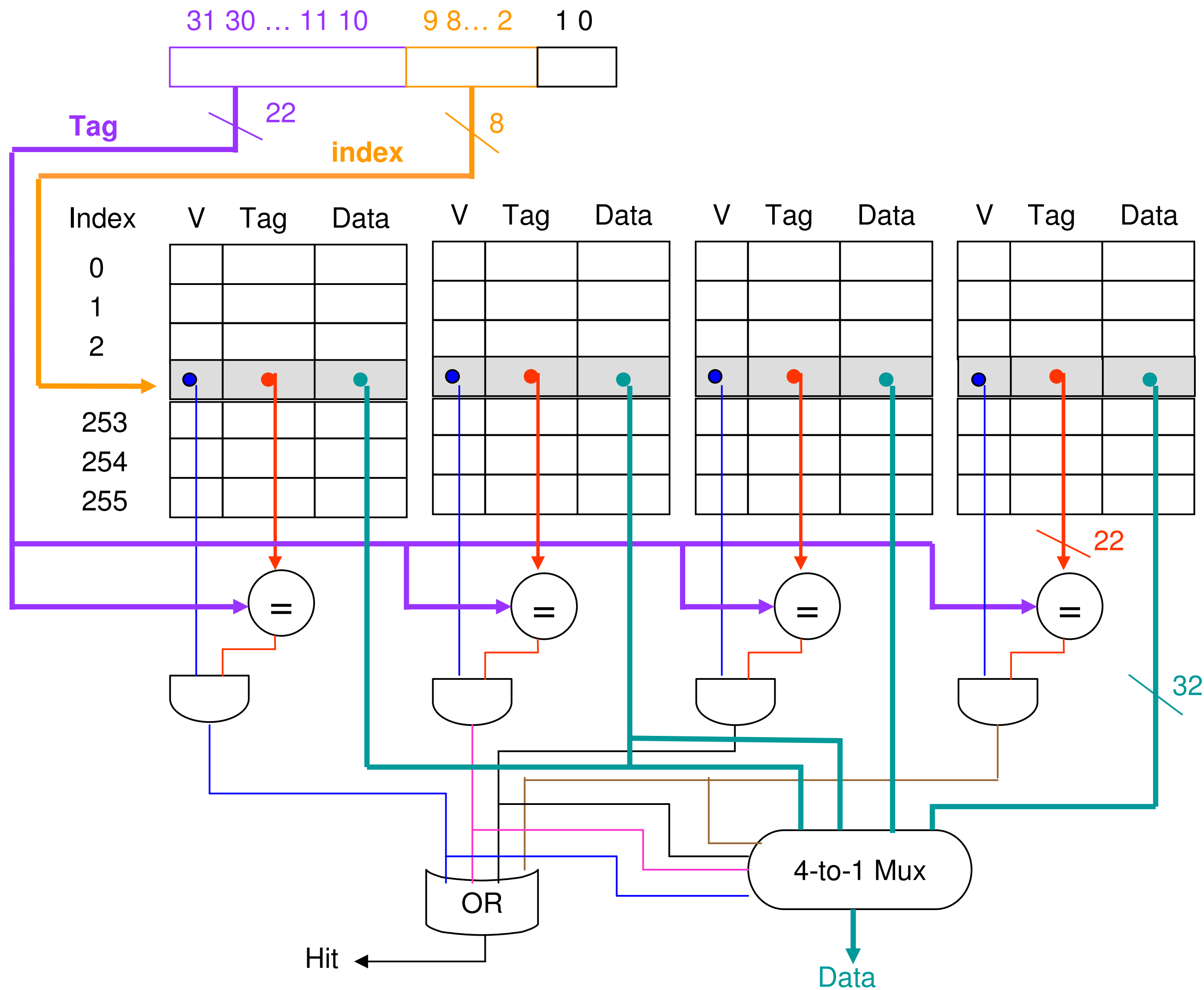
Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	Miss	Memory[0]			
8	Miss	Memory[0]	Memory[8]		
0	Hit	Memory[0]	Memory[8]		
6	Miss	Memory[0]	Memory[8]	Memory[6]	
8	Hit	Memory[0]	Memory[8]	Memory[6]	

Number of Miss : Direct-mapped (5) > two-way associative (4) > fully associative (3)

Question 9: what is optimal number of miss in this example?

Question 10: How about if we have 8 blocks in the cache? How about 16 blocks in the cache?

Implementation of set-associative cache



The tag of every cache block with appropriate set is checked to see if it matches the block address. In order to speedup comparison, we use 4 comparators to do in parallel

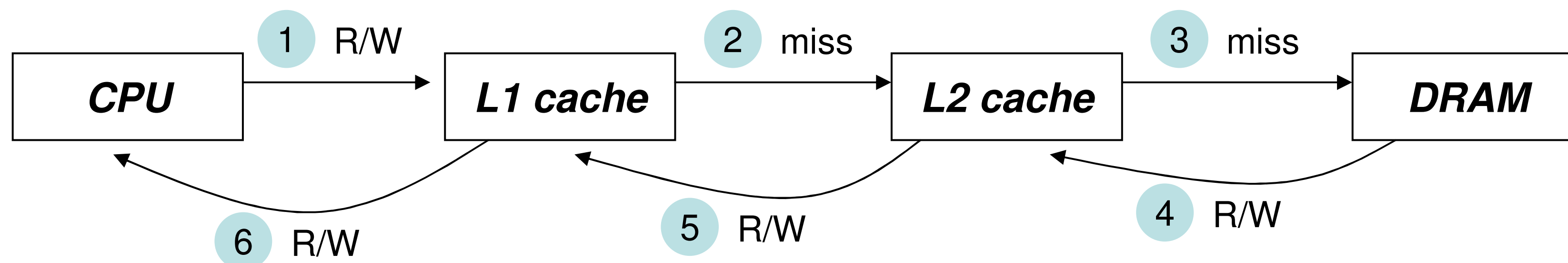
Reduce miss penalty using multi-level caches [1]

- Reduce miss rate:
  reduce probability that two different memory blocks will contend for the same cache location.
- Reduce miss penalty:
  add an additional level o hierarchy, say L1 cache, L2 cache and L3 cache.

CPI : average clock cycles per instruction

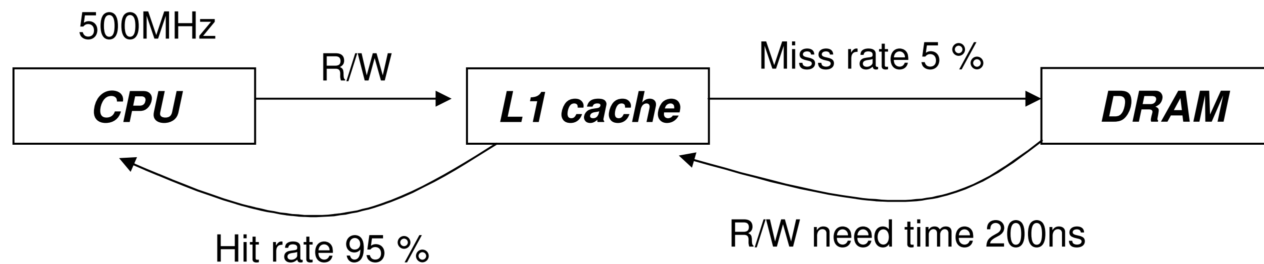
CPU time = instruction count x CPI x clock cycle time

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle



Reduce miss penalty using multi-level caches [2]

Example: suppose a processor (clock rate 500MHz) with a base CPI of 1.0, assuming all references hit in the L1 cache. Assume a main memory access time of 200ns, including all the miss handling. Suppose miss rate per instruction at L1 cache is 5%. How much faster will the machine be if we add a L2 cache that has 20 ns access time for either a hit or a miss and is large enough to reduce miss rate to main memory to 2%?

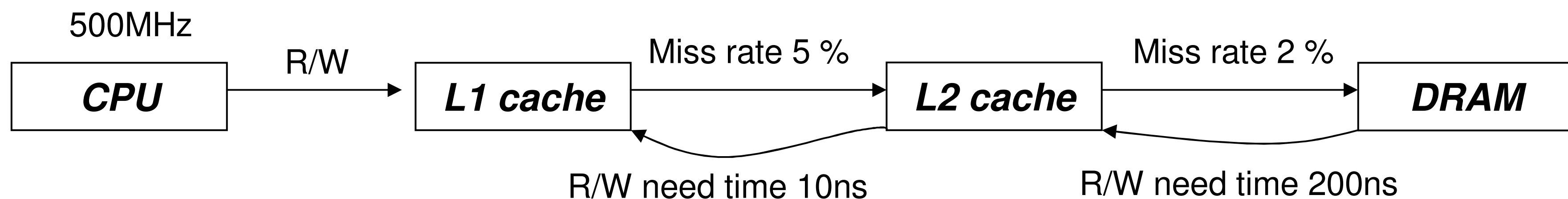


1. 500MHz → 2ns / clock cycle

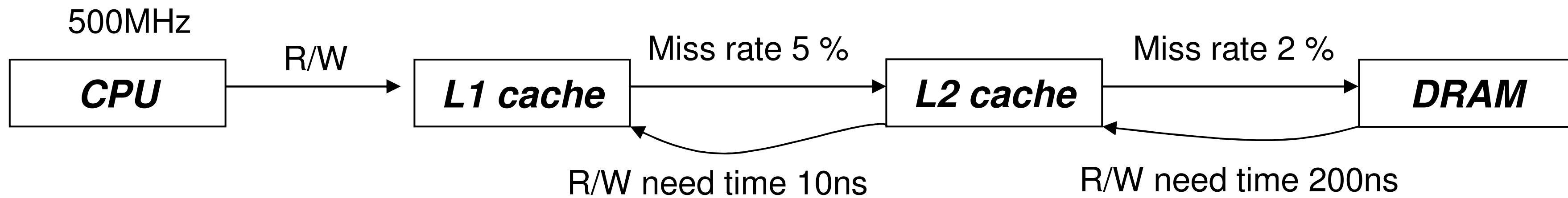
2. Miss penalty to main memory = 200ns / 2ns = 100 clock cycles (CPU clock cycle)

3. The effective CPI with L1 cache is given by

   Total CPI = base CPI + memory-stall cycles per instruction = 1.0 + 5% x 100 = 6.0

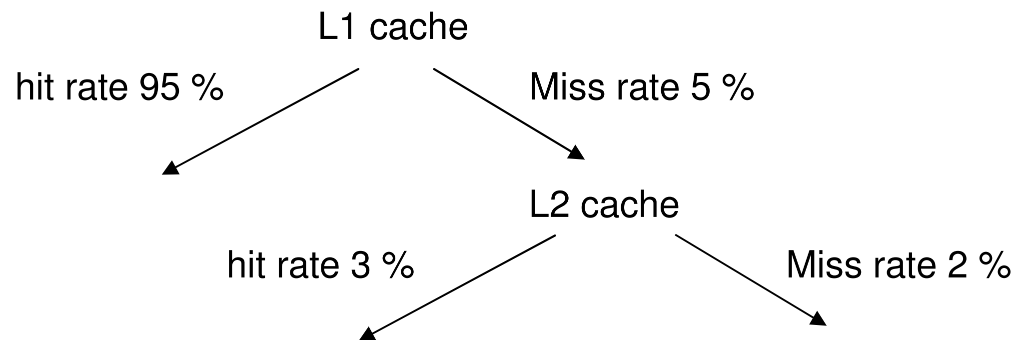


Reduce miss penalty using multi-level caches [3]



4 Miss penalty of L1 cache for an access to L2 cache = 20ns / 2ns = 10 clock cycles

5



Total CPI = 1.0 + stalls per instruction due to L1 cache miss and L2 cache hit stalls + stalls per instruction due to L1 cache miss and L2 cache miss

= 1+ (%5 - %2) x 10 + 2% x (10 + 100) = 1 + 0.3 + 2.2 = 3.5

6 The machine with L2 cache is faster by 6.0 / 3.5 = 1.7

Remark: L1 cache focus on "hit time" to yield short clock cycle whereas L2 cache focus on "miss rate" to reduce penal of long memory access time.

OutLine

- Basic of cache
- Cache coherence
  - simple snooping protocol
  - MESI
- False sharing
- Summary

Write policy in the cache

- Write-though: the information is written to both the block in cache and block in main memory.
- Write-back: information is only written to the block in cache. The modified block is written to main memory only when it is replaced.

Advantage of write-back

- Individual words can be written by the processor in the cache level, fast!
- Multiple writes within a block requires only one write to main memory
- When blocks are written back, the system can make effective use of a high bandwidth transfer.

disadvantage of write-back

- Interaction with other processors when RAW (Read after Write) hazard occurs, say other processor will read the incorrect data in its own cache.

Advantage of write-through

- Misses are simpler and cheaper because they never require a block in cache to be written to main memory.
- Easy to implement than write-back, a write-through cache only needs a write buffer.

disadvantage of write-through

- Cost since write to main memory is very slow

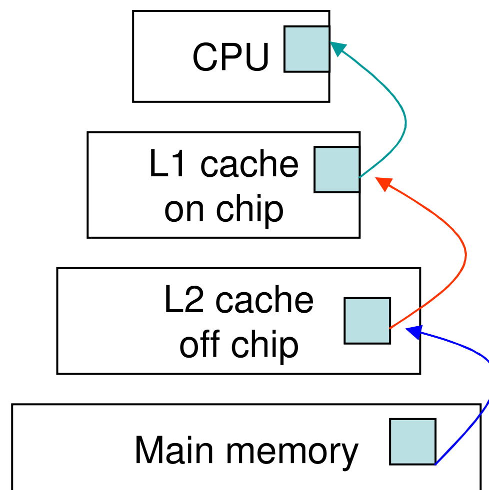
Consistency management in cache

- Keep the cache consistent with itself: avoid two copies of a single item in different places of the cache.

- Keep the cache consistent with the backing store (main memory): solve RAW (Read after Write) hazard
  - write-through policy
  - write-back policy

- Keep the cache consistent with other caches
  - L1 cache versus L2 cache in the same processor
  - L1 cache versus L1 cache in different processors
  - L1 cache versus L2 cache in different processors
  - L2 cache versus L2 cache in different processors

  two policies: inclusion or exclusion



Inclusion: $L1\ cache \subset L2\ cache \subset DRAM\ (main\ memory)$

What Does Coherency Mean?

- Informally:
  - "Any read must return the most recent write"
  - Too strict and too difficult to implement
- Better:
  - "Any write must eventually be seen by a read"
  - All writes are seen in proper order ("serialization")
- Two rules to ensure this:
  - "If P writes x and P1 reads it, P's write will be seen by P1 if the read and write are sufficiently far apart"
  - Writes to a single location are serialized: seen in one order
    - Latest write will be seen
    - Otherwise could see writes in illogical order (could see older value after a newer value)

Potential Hardware Coherency Solutions

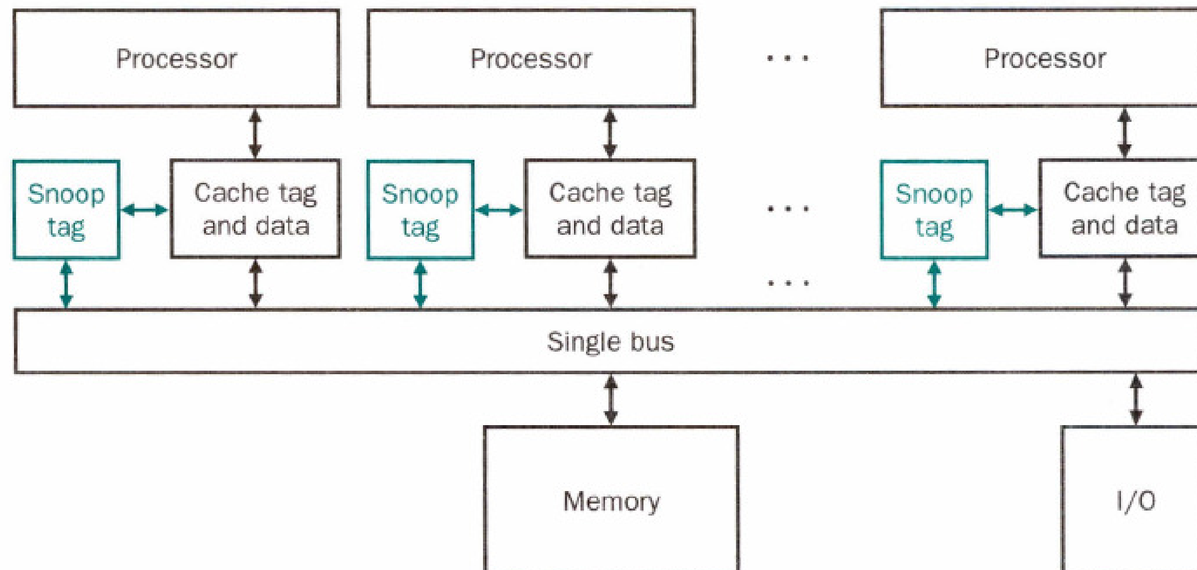
- ***Snooping Solution*** (Snoopy Bus):
  - Send all requests for data to all processors
  - Processors snoop to see if they have a copy and respond accordingly
  - Requires broadcast, since caching information is at processors
  - Works well with bus (natural broadcast medium)
  - Dominates for small scale machines (most of the market)

- ***Directory-Based Schemes***
  - Keep track of what is being shared in one centralized place
  - Distributed memory => distributed directory for scalability (avoids bottlenecks)
  - Send point-to-point requests to processors via network
  - Scales better than Snooping
  - Actually existed BEFORE Snooping-based schemes

Cache coherency in multi-processor: snooping protocol [1]

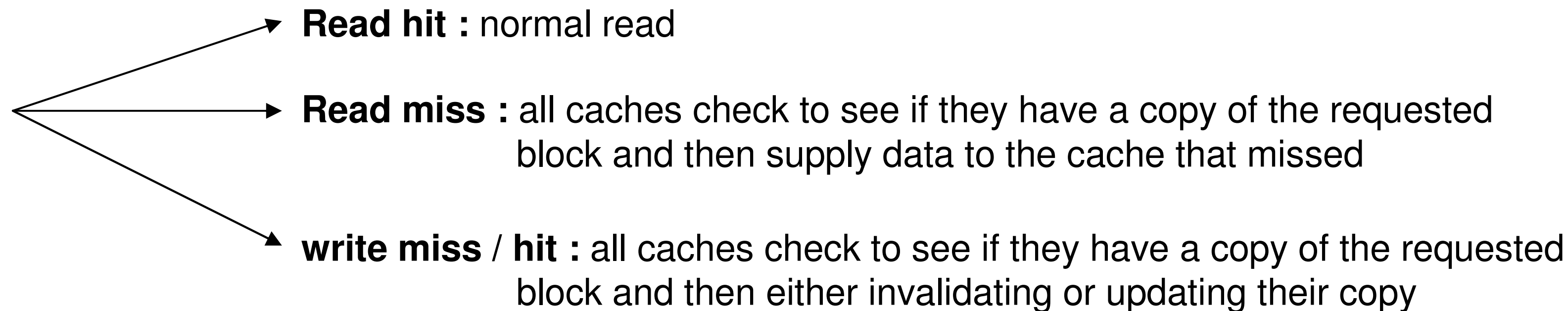
All cache controllers monitor (snoop) on the bus to determine whether or not they have a copy of the shared block

- Maintaining coherency has two components: read and write
  - read: not a problem with multiple copies
  - write: a processor must have exclusive access to write a word, so all processors must get new values after a write, say we must avoid RAW hazard
- The consequence of a write to shared data is either
  - to invalidate all other copies or
  - to update the shared copies with the value being written



Snoop tag is used to handle snoop requests

Cache coherency in multi-processor: snooping protocol [2]



Snooping protocols are of two types

- **Write-invalidate**: similar to write-back policy (commercial used)
  - multiple readers, single writer
  - write to shared data: an invalidate signal is sent to all caches which snoop and invalidate any copies.
  - Read miss:
    (1) write-through: memory is always up-to-date
    (2) write-back: snoop in caches to find most recent copy
- Write-update: similar to write-through
  - writing processor broadcasts the new data over the bus, then all copies are updated with the new value. This would consume much bus bandwidth.

Write-invalidation protocol based on write-back policy. Each cache block has three states

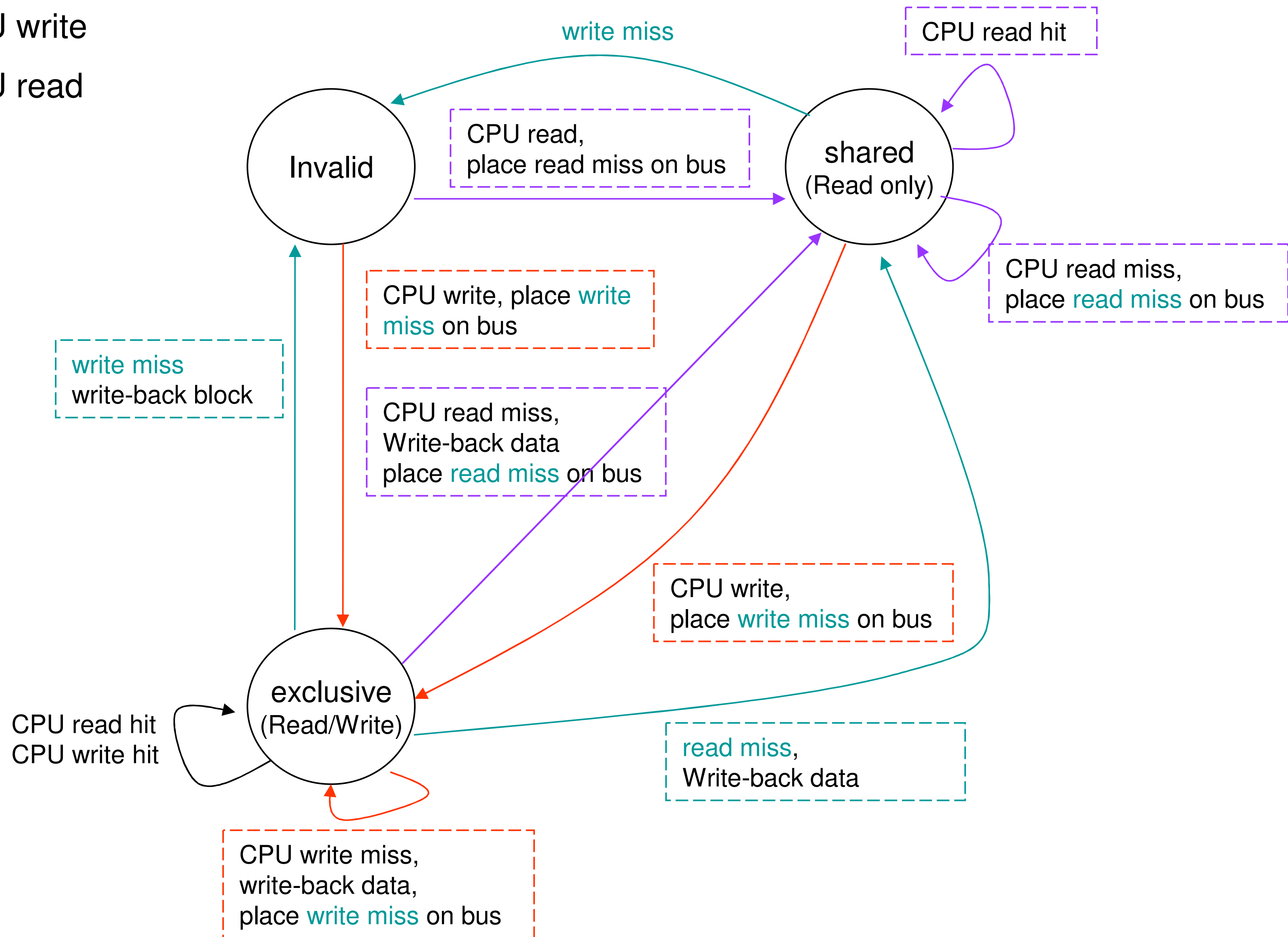
- ***Shared*** (Read only): this cache block is clean in all caches and up-to-date in memory, multiple read is allowed.
- ***Exclusive*** (Read/Write): this cache block is dirty in exactly one cache.
- ***Invalid***: this cache block does not have valid data

Simple snooping protocol [1]

color: bus signal, including read miss, write miss

color: CPU write

color: CPU read



Simple snooping protocol [2]

Cache coherence mechanism receives requests from both the processor and the bus and responds to these based on the type of request, state of cache block and hit/miss

State of cache block	source	Request	Function and explanation
invalid	processor	Read miss	place read miss on bus
	processor	Write miss	place write miss on bus
shared	processor	Read hit	Read data in cache
	processor	Read miss	Address conflict miss: place read miss on bus
	processor	Write hit	Place write miss on bus since data is modified and then other copies are not valid
	processor	Write miss	Address conflict miss: place write miss on bus
	bus	Read miss	No action: allow memory to service read miss
	bus	Write miss	Attempt to rite shared block: invalidate the block (go to invalid state)
exclusive	processor	Read hit	Read data in cache
	processor	Read miss	Address conflict miss: write back block, then place read miss on bus
	processor	Write hit	Write data in cache
	processor	Write miss	Address conflict miss: write back block, then place write miss on bus
	bus	Read miss	Attempt to shared data: place cache block on bus and change state to shared
	bus	Write miss	Attempt to write block that is exclusive elsewhere: write back the cache block and make its state invalid

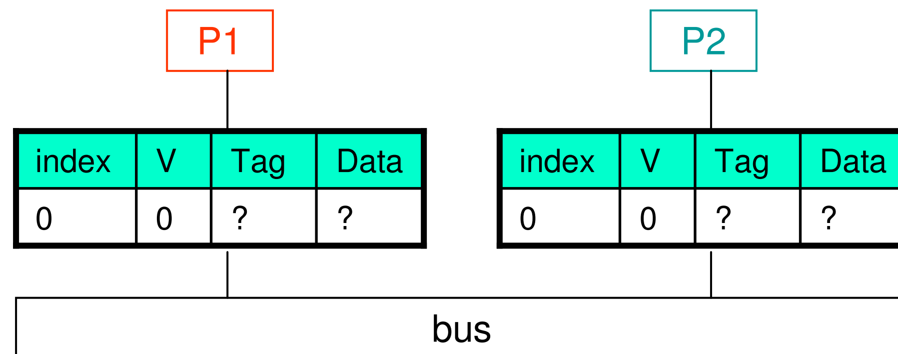
Example of state sequence [1]

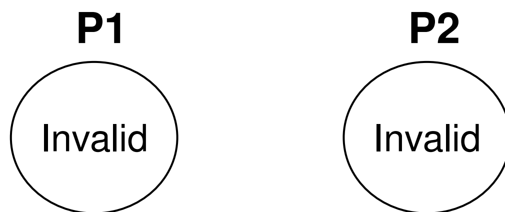
Processor 1 **Processor 2** **Bus** **Memory**

	P1			***P2***			***Bus***				***Memory***	
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*Value*
	Invalid			Invalid								
P1: write 10 to A1												
P1: read A1												
P2: read A1												
P2: write 20 to A1												
P1: write 40 to A2												

Assumption

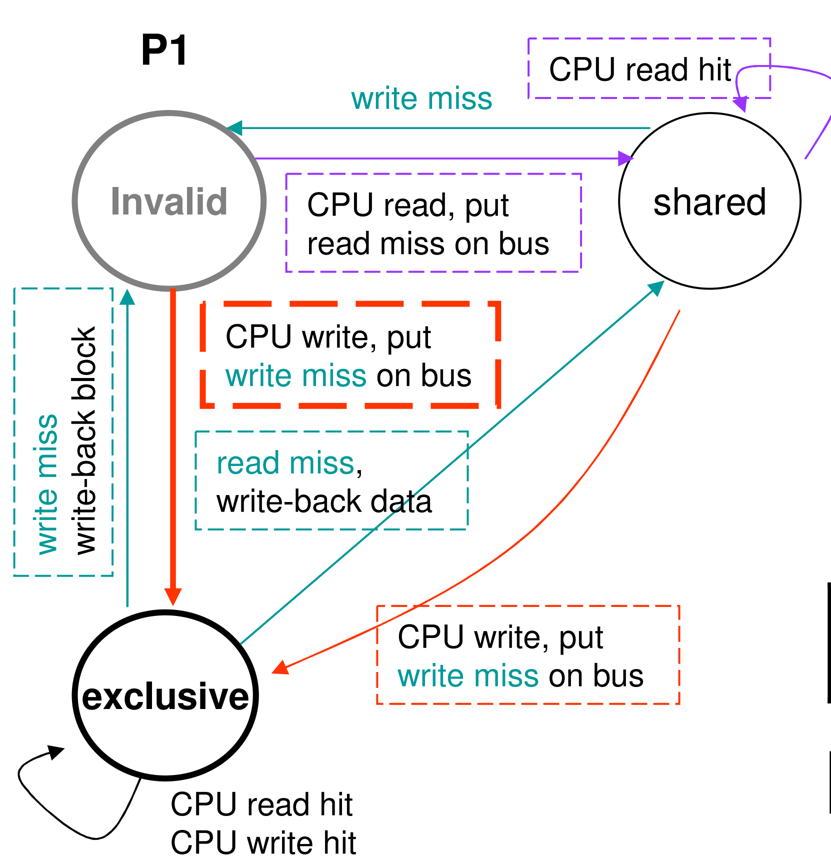
1. Initial cache state is invalid
2. A1 and A2 map to same cache block but A1 is not A2



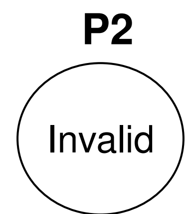


Example of state sequence [2]

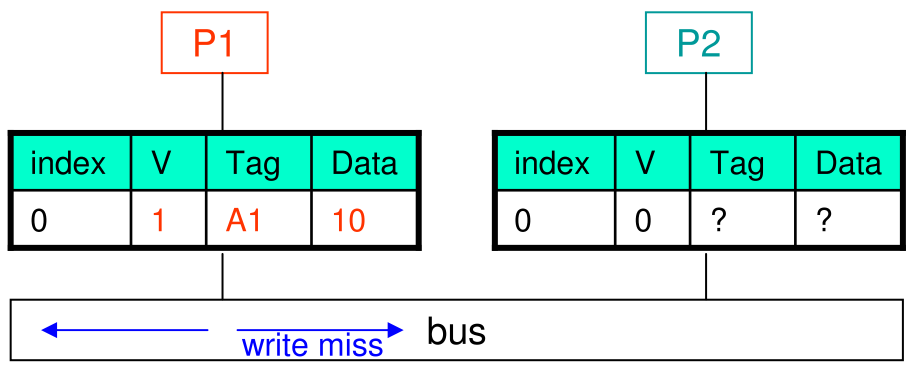
	P1			*P2*			*Bus*				*Memory*	
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*Value*
P1: write 10 to A1	*Excl*	*A1*	*10*	invalid			*WrMs*	*P1*	*A1*			
P1: read A1												
P2: read A1												
P2: write 20 to A1												
P1: write 40 to A2												



P2

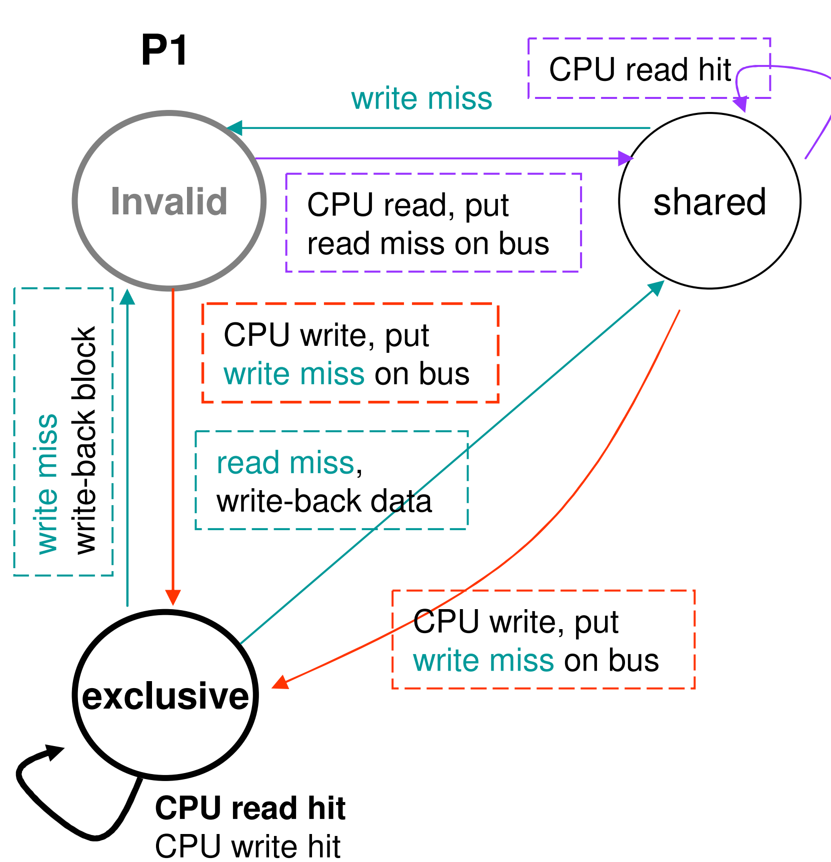


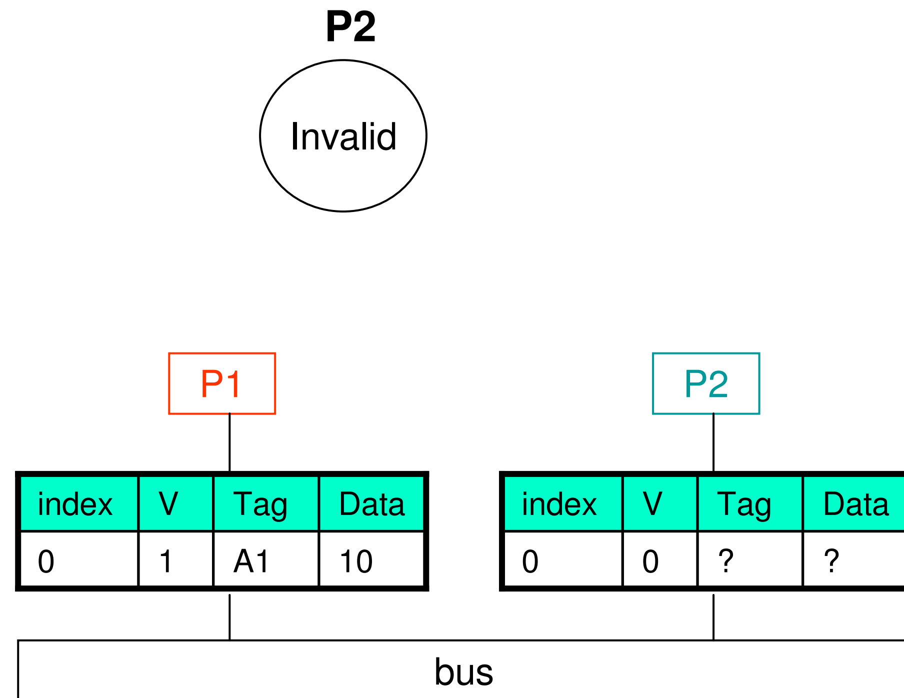
Write miss signal on bus does not affect P2



Example of state sequence [3]

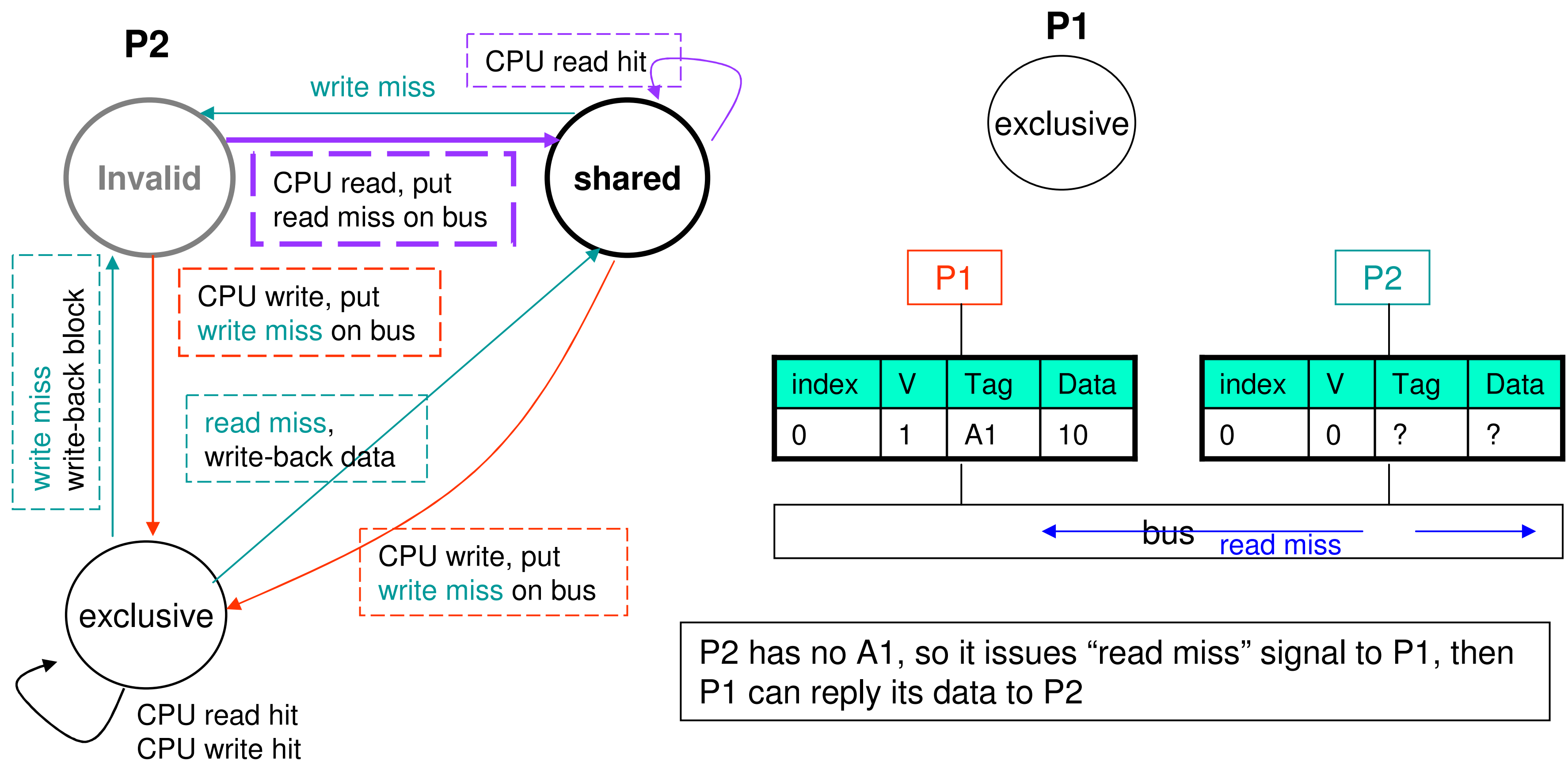
	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: write 10 to A1	Excl	A1	10	invalid			WrMs	P1	A1			
P1: read A1	Excl	A1	10	invalid								
P2: read A1												
P2: write 20 to A1												
P1: write 40 to A2												





Example of state sequence [3]

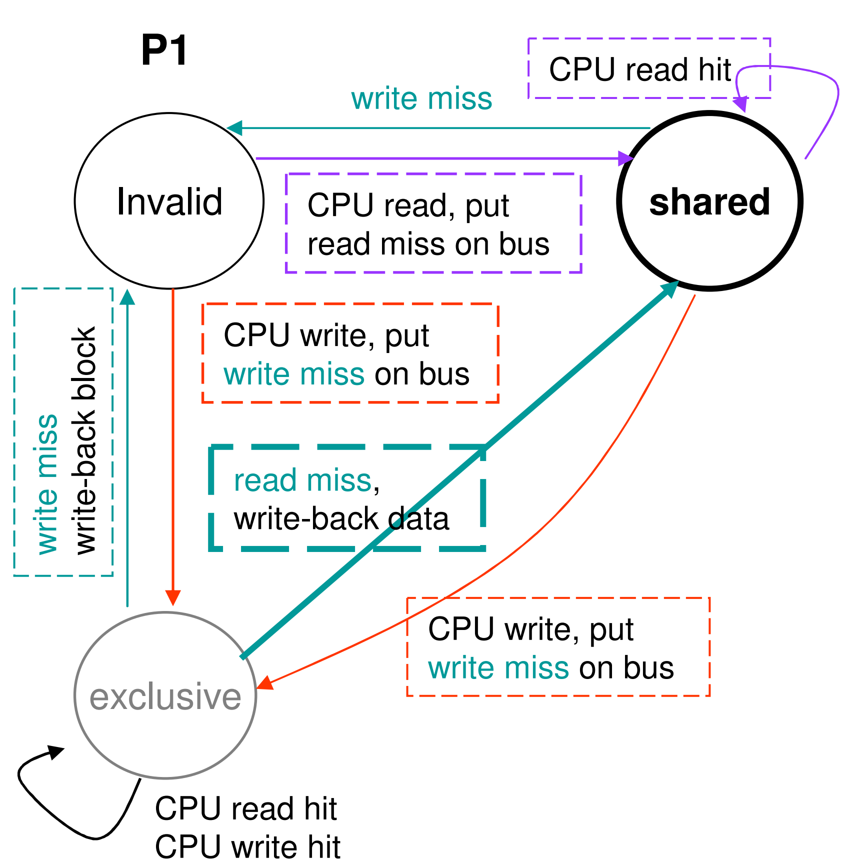
	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: write 10 to A1	Excl	A1	10	invalid			WrMs	P1	A1			
P1: read A1	Excl	A1	10	invalid								
P2: read A1				share	A1		RdMs	P2	A1			
P2: write 20 to A1												
P1: write 40 to A2												

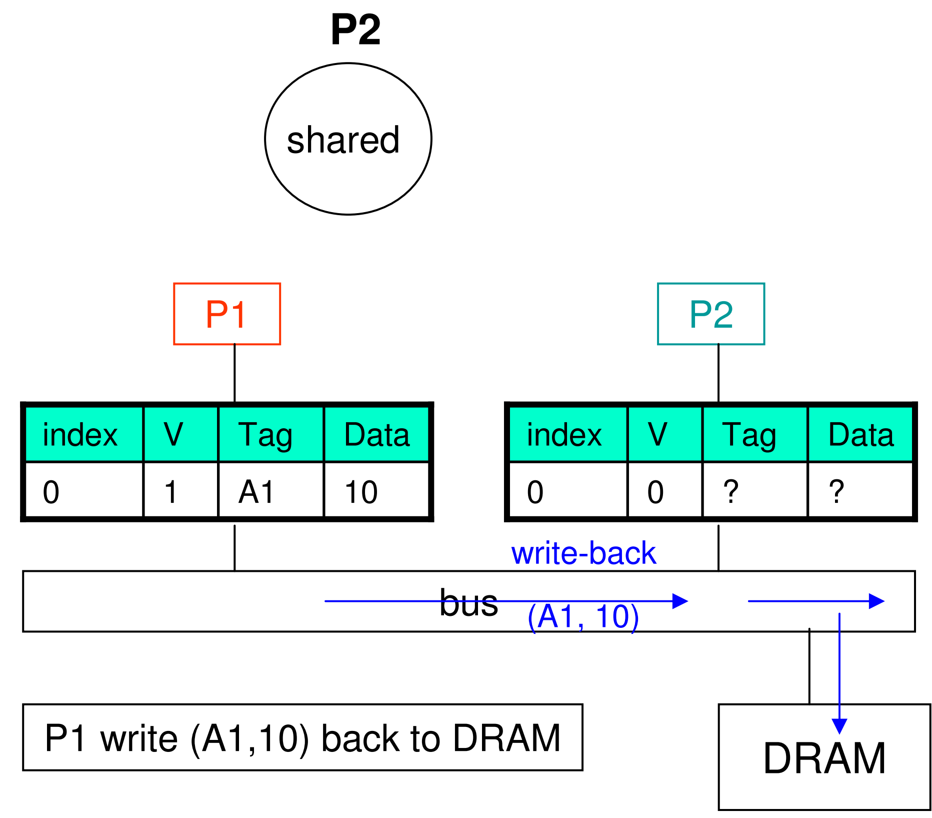


P2 has no A1, so it issues “read miss” signal to P1, then P1 can reply its data to P2

Example of state sequence [4]

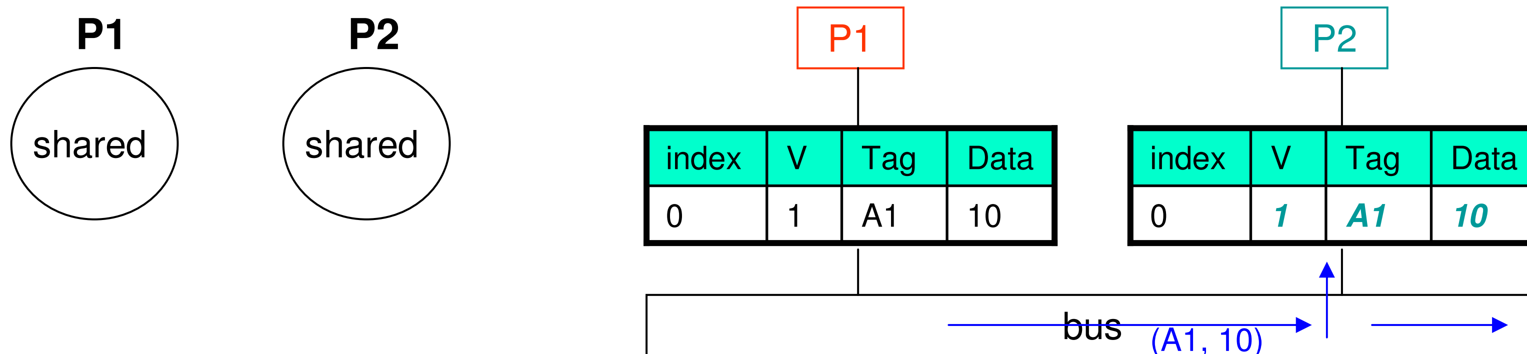
	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: read A1	Excl	A1	10	invalid								
P2: read A1				share	A1		RdMs	P2	A1			
	shared	A1	10				WrBk	P1	A1	10	A1	10
P2: write 20 to A1												
P1: write 40 to A2												





Example of state sequence [5]

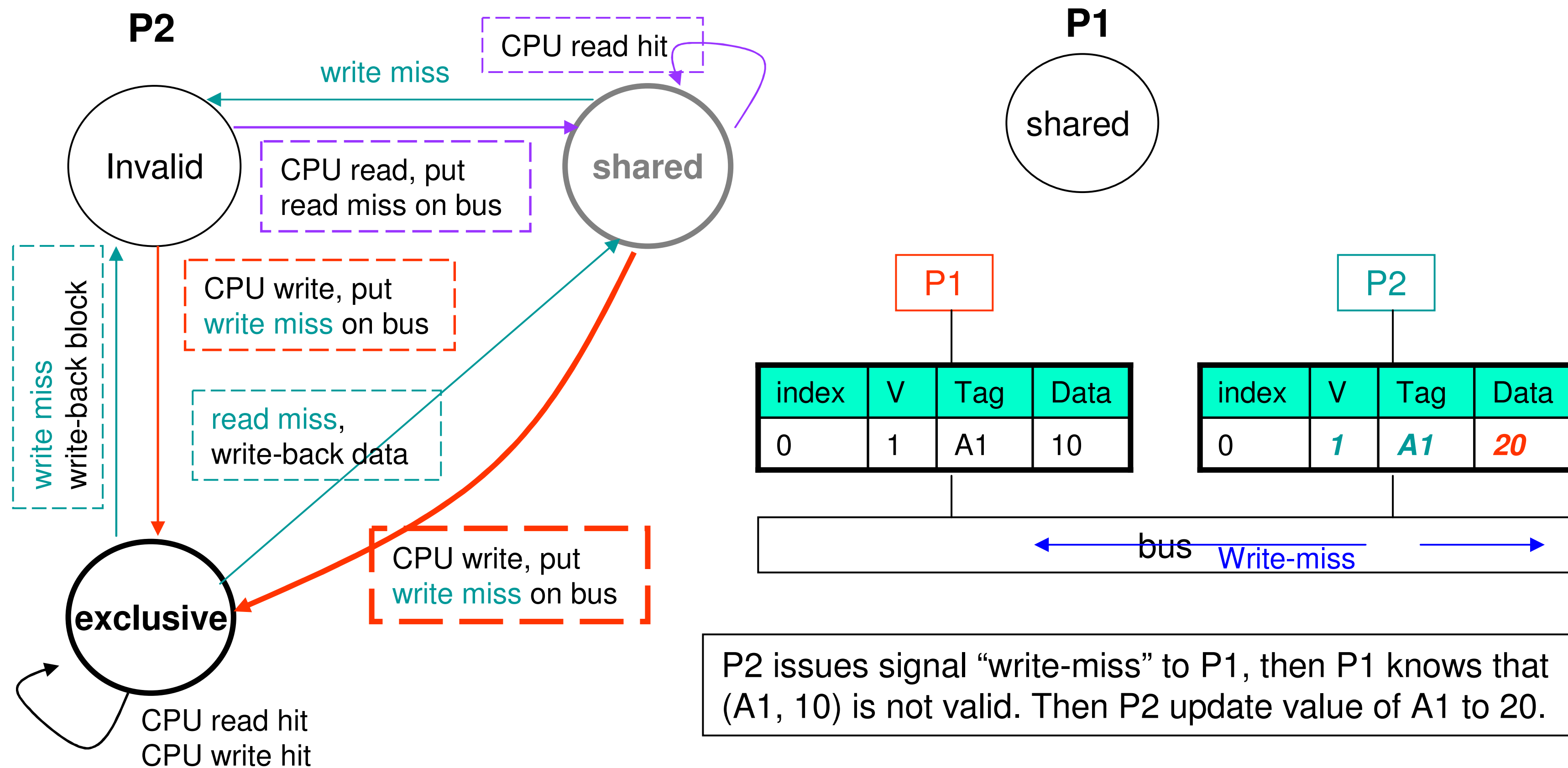
	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: read A1	Excl	A1	10	invalid								
P2: read A1				share	A1		RdMs	P2	A1			
	shared	A1	10				WrBk	P1	A1	10	A1	10
				share	A1	10	RdDa	P2	A1	10		10
P2: write 20 to A1												
P1: write 40 to A2												



P1 and P2 are all in state ***shared***, this means that (A1, 10) is shared by two processors and both processors can read (A1,10) at the same time from their own cache without any communication.

Example of state sequence [6]

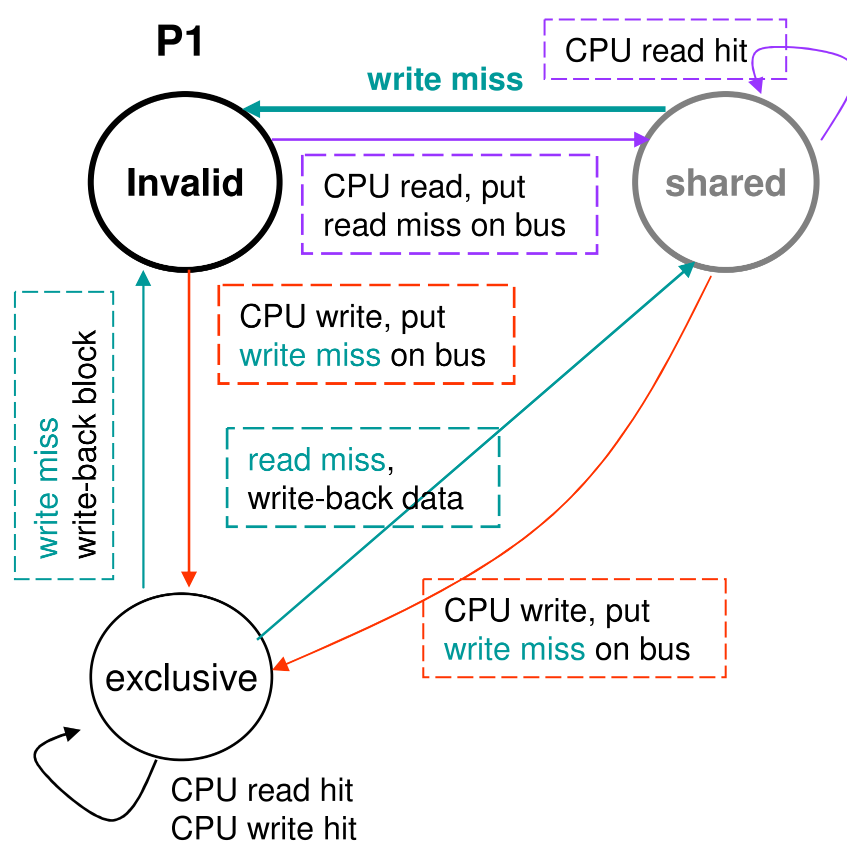
	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P2: read A1				share	A1		RdMs	P2	A1			
	shared	A1	10				WrBk	P1	A1	10	A1	10
				share	A1	10	RdDa	P2	A1	10		10
P2: write 20 to A1				excl	A1	20	WrMs	P2	A1			10
P1: write 40 to A2												

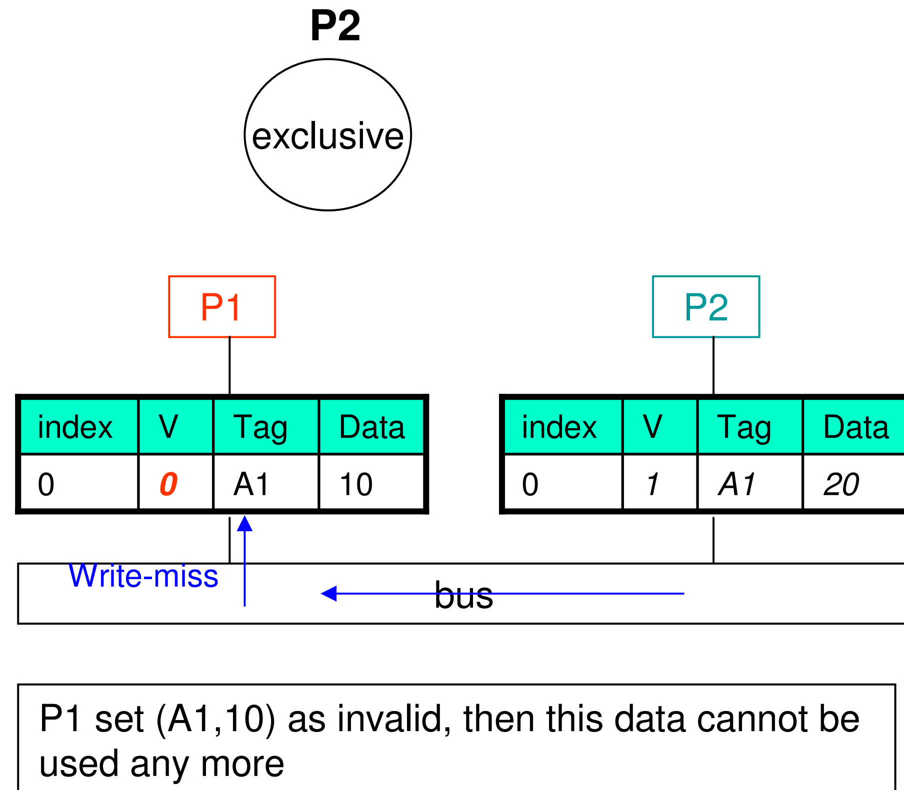


P2 issues signal “write-miss” to P1, then P1 knows that (A1, 10) is not valid. Then P2 update value of A1 to 20.

Example of state sequence [7]

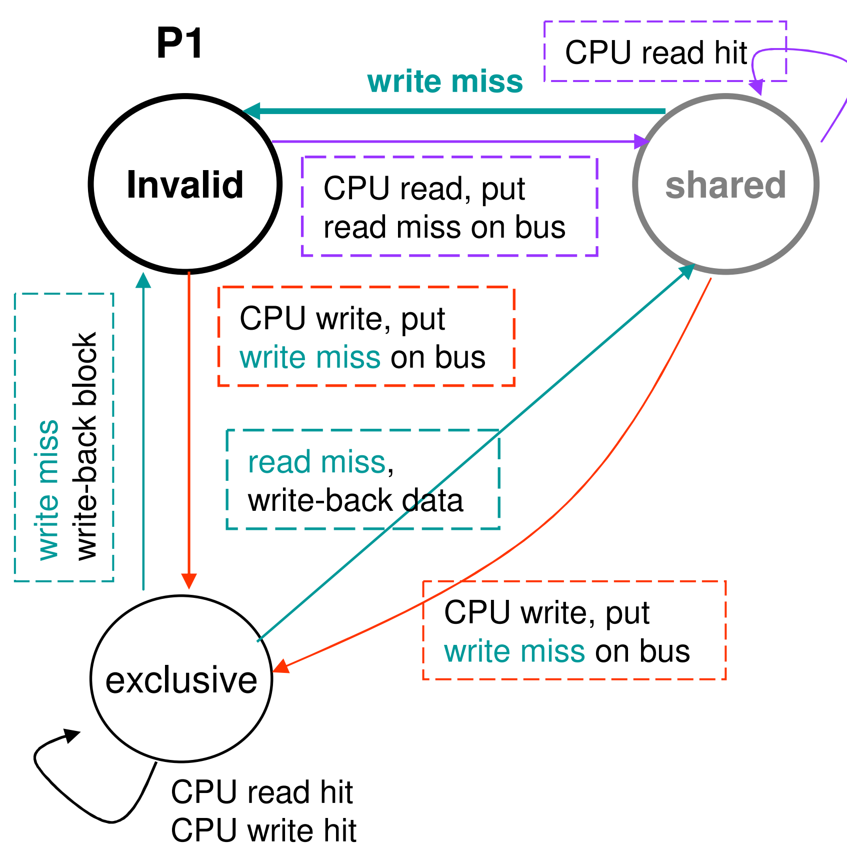
	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
	shared	A1	10				WrBk	P1	A1	10	A1	10
				share	A1	10	RdDa	P2	A1	10		10
P2: write 20 to A1				excl	A1	20	WrMs	P2	A1			10
	Invalid			excl								10
P1: write 40 to A2												

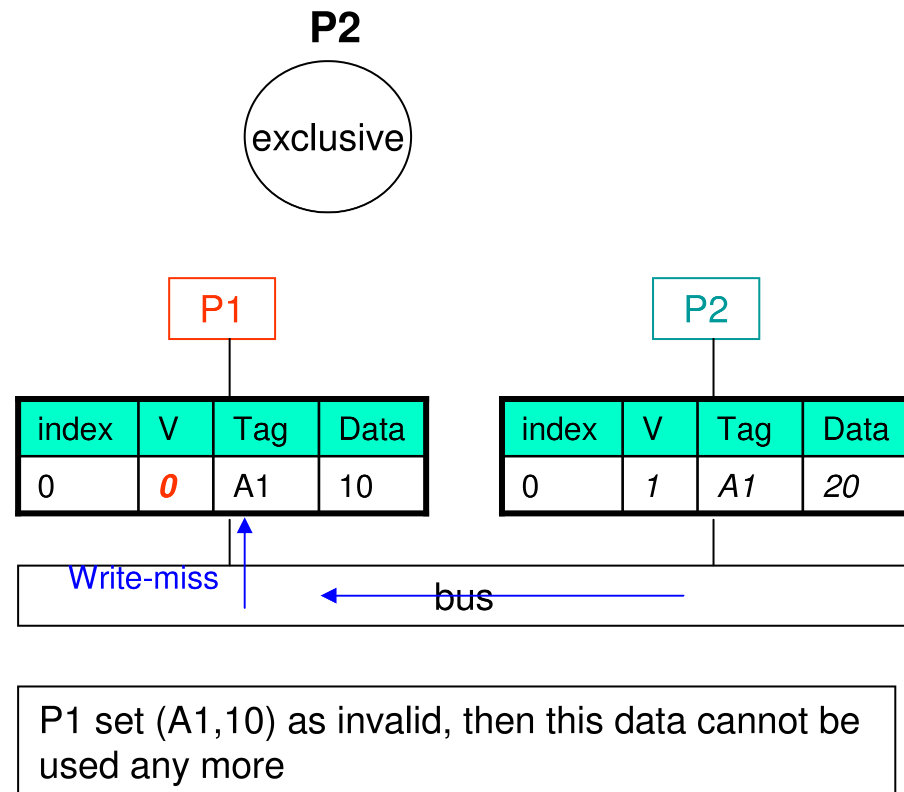




Example of state sequence [8]

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
	shared	A1	10				WrBk	P1	A1	10	A1	10
				share	A1	10	RdDa	P2	A1	10		10
P2: write 20 to A1				excl	A1	20	WrMs	P2	A1			10
	Invalid			excl								10
P1: write 40 to A2												

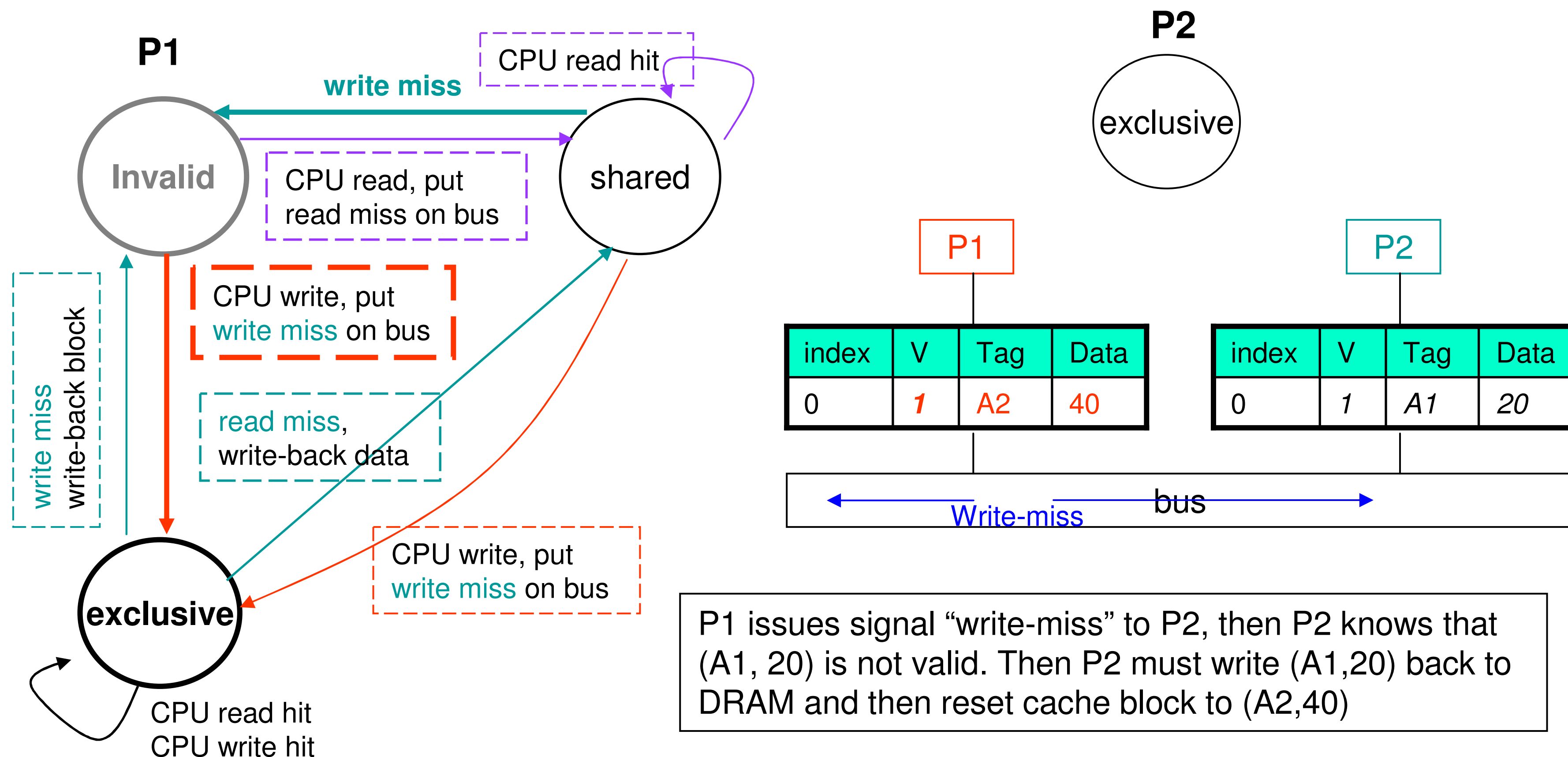




P1 set (A1,10) as invalid, then this data cannot be used any more

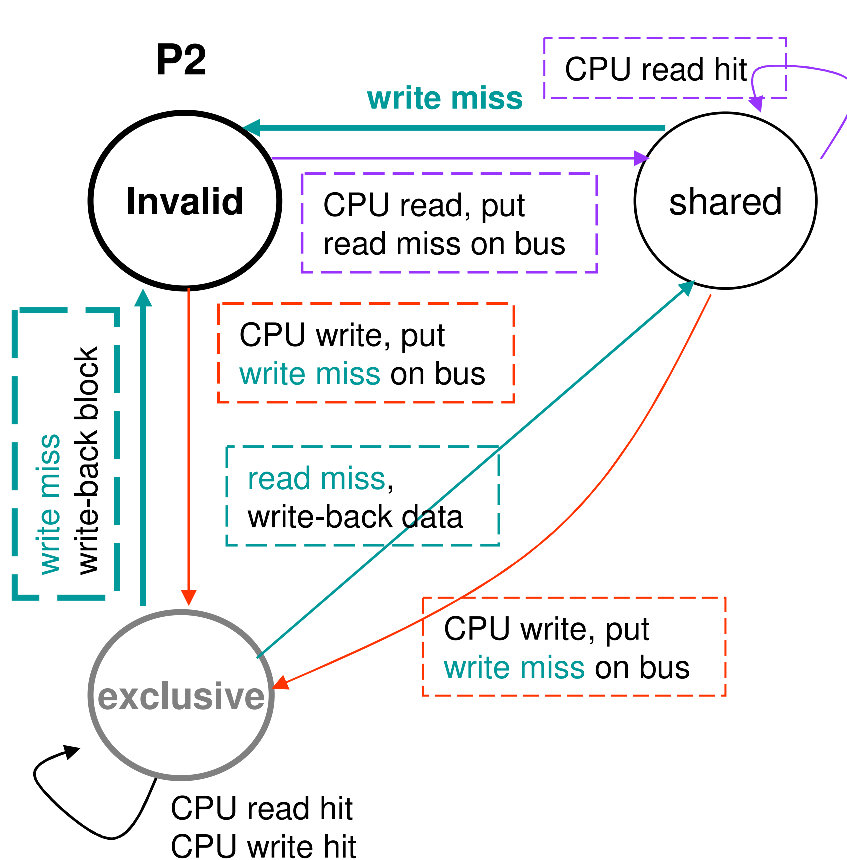
Example of state sequence [9]

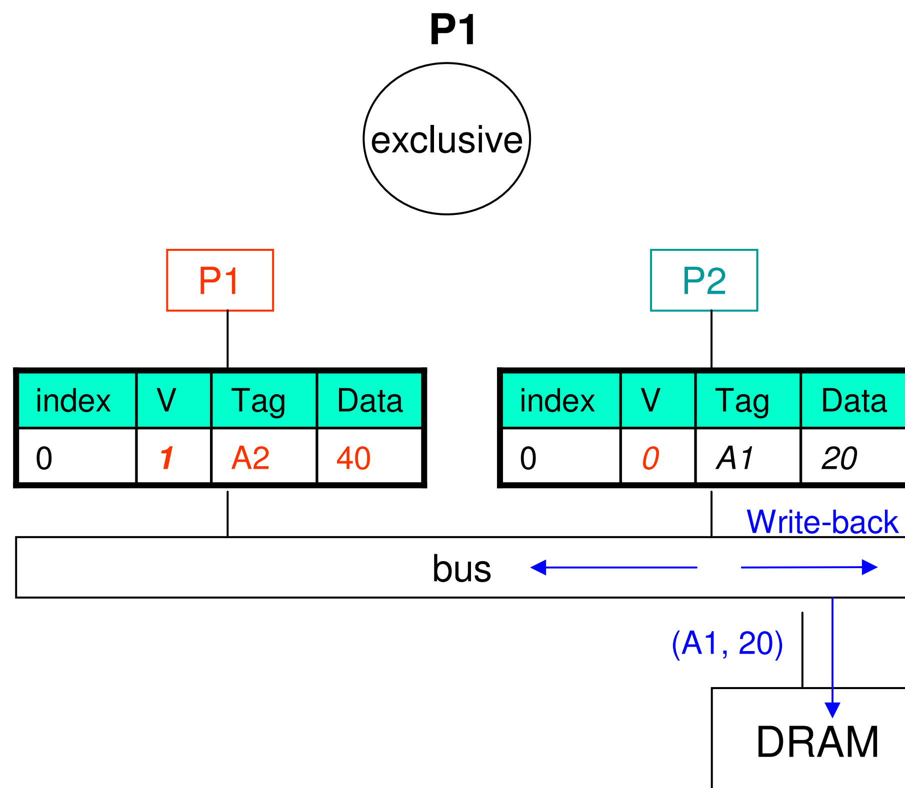
	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
	shared	A1	10				WrBk	P1	A1	10	A1	10
				share	A1	10	RdDa	P2	A1	10		10
P2: write 20 to A1				excl	A1	20	WrMs	P2	A1			10
	Invalid			excl								10
P1: write 40 to A2	excl	A2	40				WrMs	P1	A2			



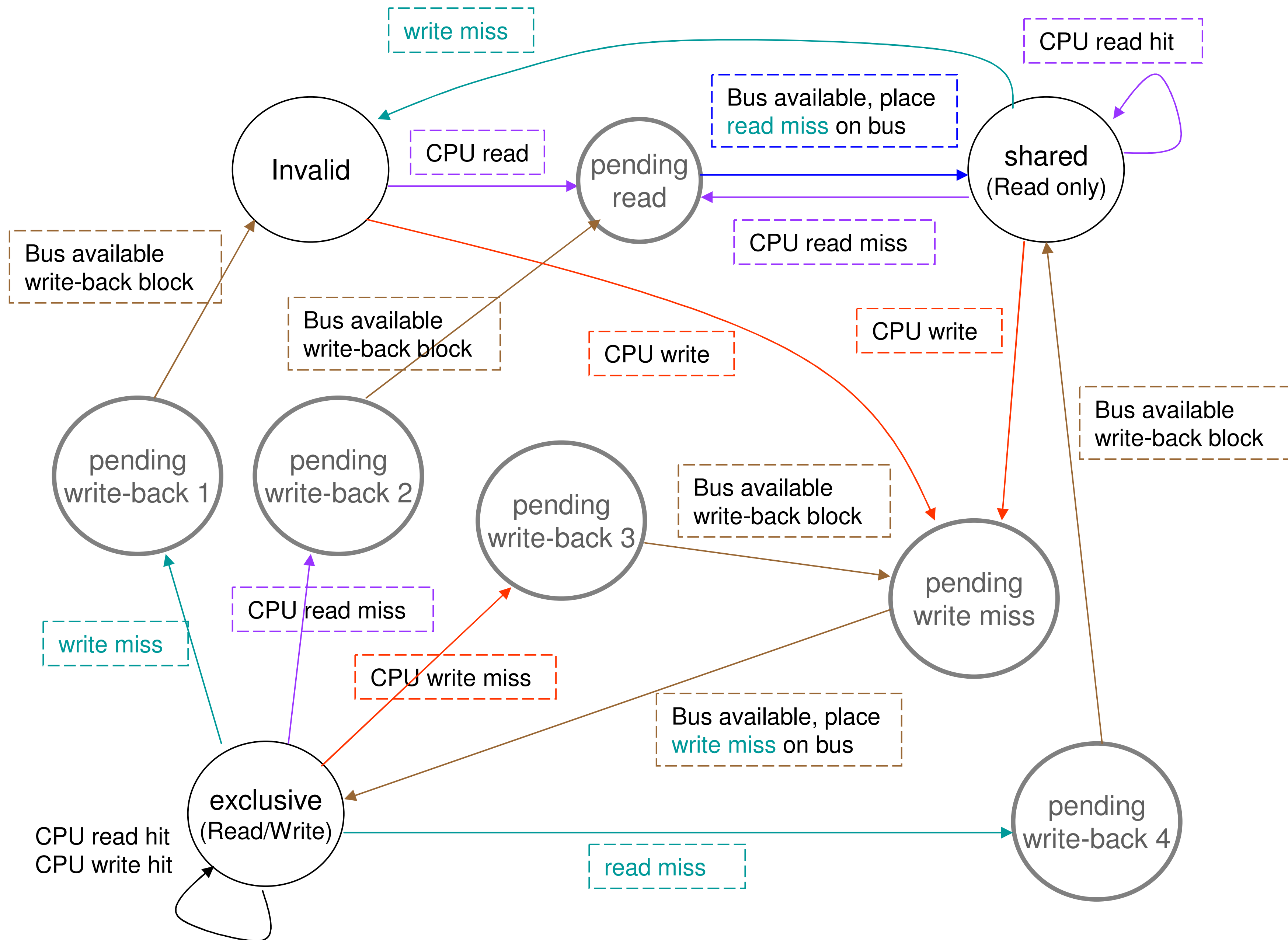
Example of state sequence [10]

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
				share	A1	10	RdDa	P2	A1	10		10
P2: write 20 to A1				excl	A1	20	WrMs	P2	A1			10
	Invalid			excl								10
P1: write 40 to A2	excl	A2	40				WrMs	P1	A2			
	excl			invalid			WrBk	P2	A1	20	A1	20



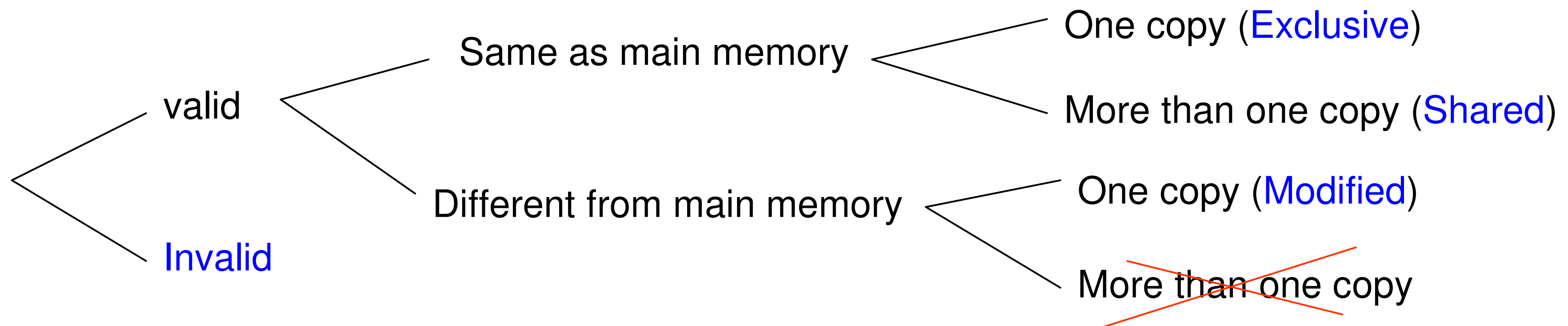


Finite state controller for a simple snooping cache



MESI Protocol (Intel 64 and IA-32 architecture)

- ***Modified:*** The line in the cache has been modified (different from main memory) and is available only in this cache since we only accept multiple read, single write
- ***Exclusive*** : the line in the cache is the same as that in main memory and is not present in any other cache.
- ***Shared*** : the line in the cache is the same as that in main memory and may be present in another cache. This supports multiple read.
- ***Invalid***: the line in the cache does not contains valid data.

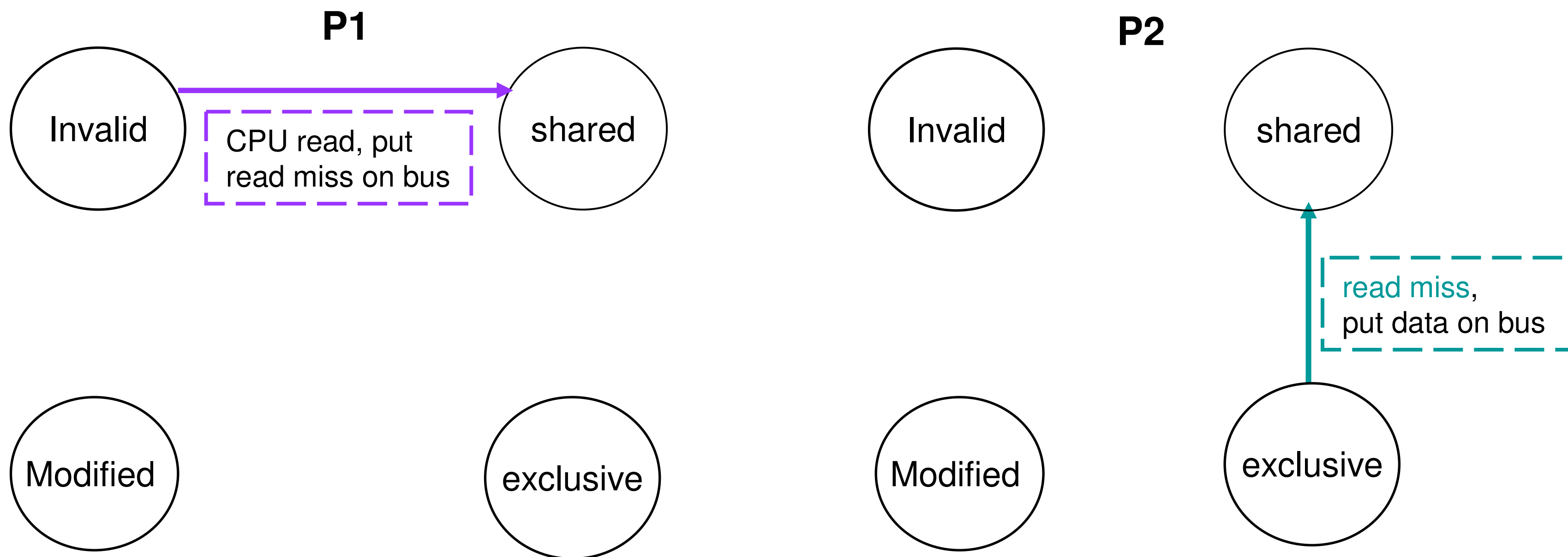


	Modified	***Exclusive***	***Shared***	***Invalid***
This cache line valid?	Yes	Yes	Yes	No
The memory copy is	out of date	valid	valid	----
Copies exist in other caches?	No	No	Maybe	Maybe
A write to this line	Does not go to bus	Does not go to bus	Goes to bus and updates cache	Goes directly to bus

Read miss [1]

When ***P1*** has a read miss, then it initiates a memory read to read the line in main memory (or other cache). So ***P1*** inserts a read miss signal on bus that alerts other processors.

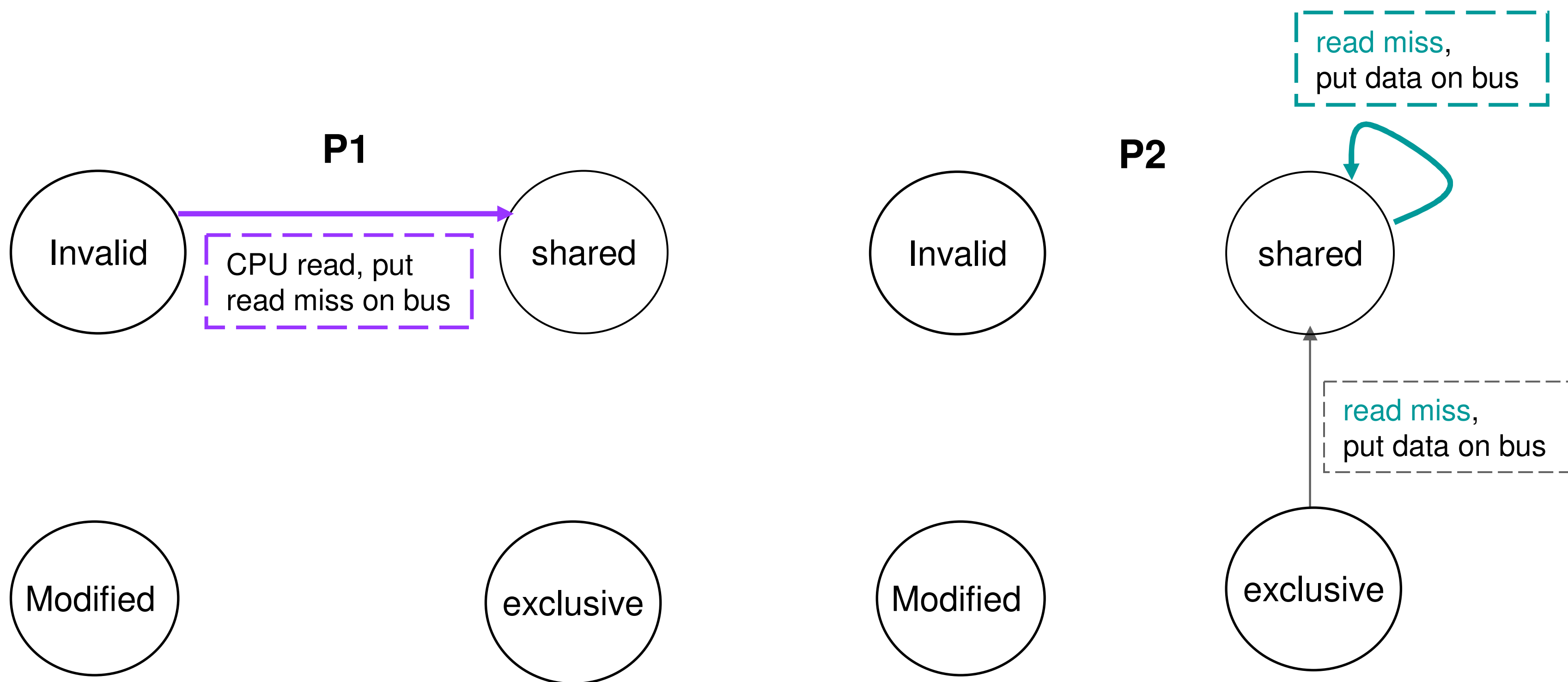
- Case 1: If ***P2*** has a clean copy of the line in the ***exclusive*** state, it returns a signal indicating that it shares this line. And then P2 transitions state from ***exclusive*** to ***shared*** since data is shared by ***P1*** and ***P2***. ***P1*** reads the line from bus and transitions state from ***invalid*** to ***shared***.



Read miss [2]

When ***P1*** has a read miss, then it initiates a memory read to read the line in main memory (or other cache). So ***P1*** inserts a read miss signal on bus that alerts other processors.

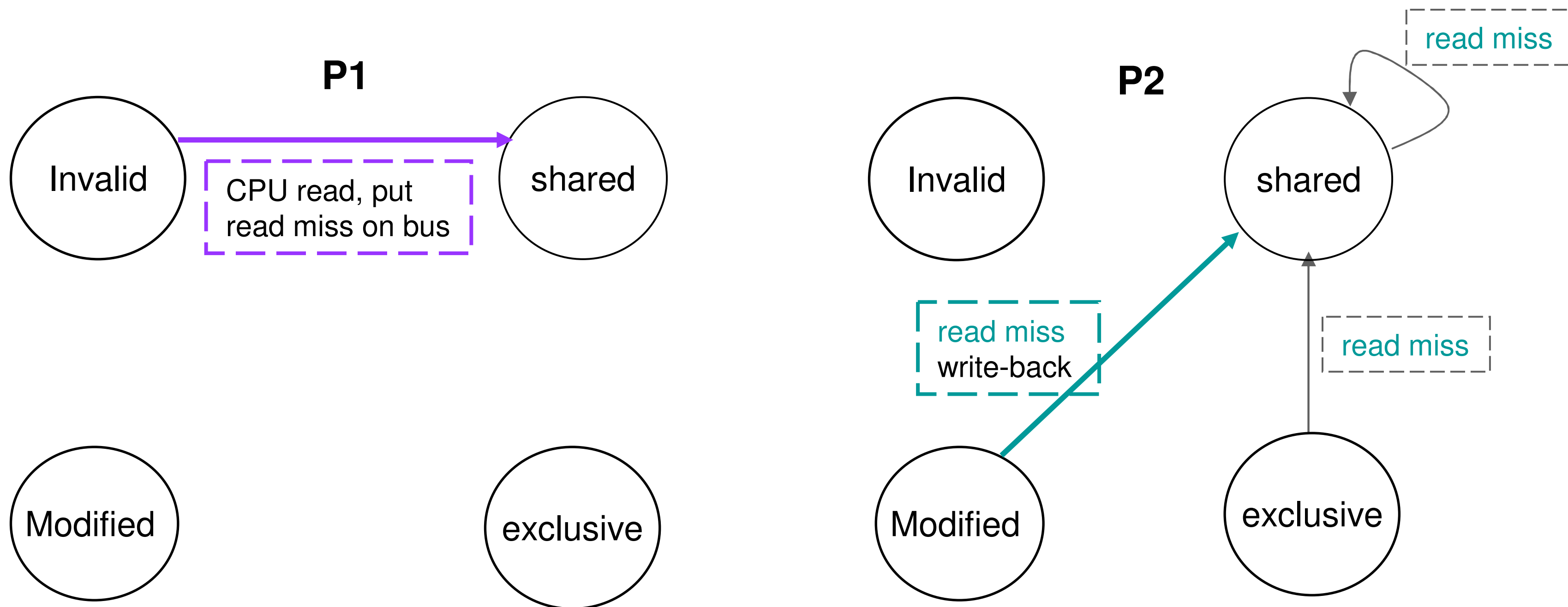
- Case 2: If ***P2*** has a clean copy of the line in the ***shared*** state, it returns a signal indicating that it shares this line. And then P2 keep state as ***shared***. ***P1*** reads the line from bus and transitions state from ***invalid*** to ***shared***.



Read miss [3]

When ***P1*** has a read miss, then it initiates a memory read to read the line in main memory (or other cache). So ***P1*** inserts a read miss signal on bus that alerts other processors.

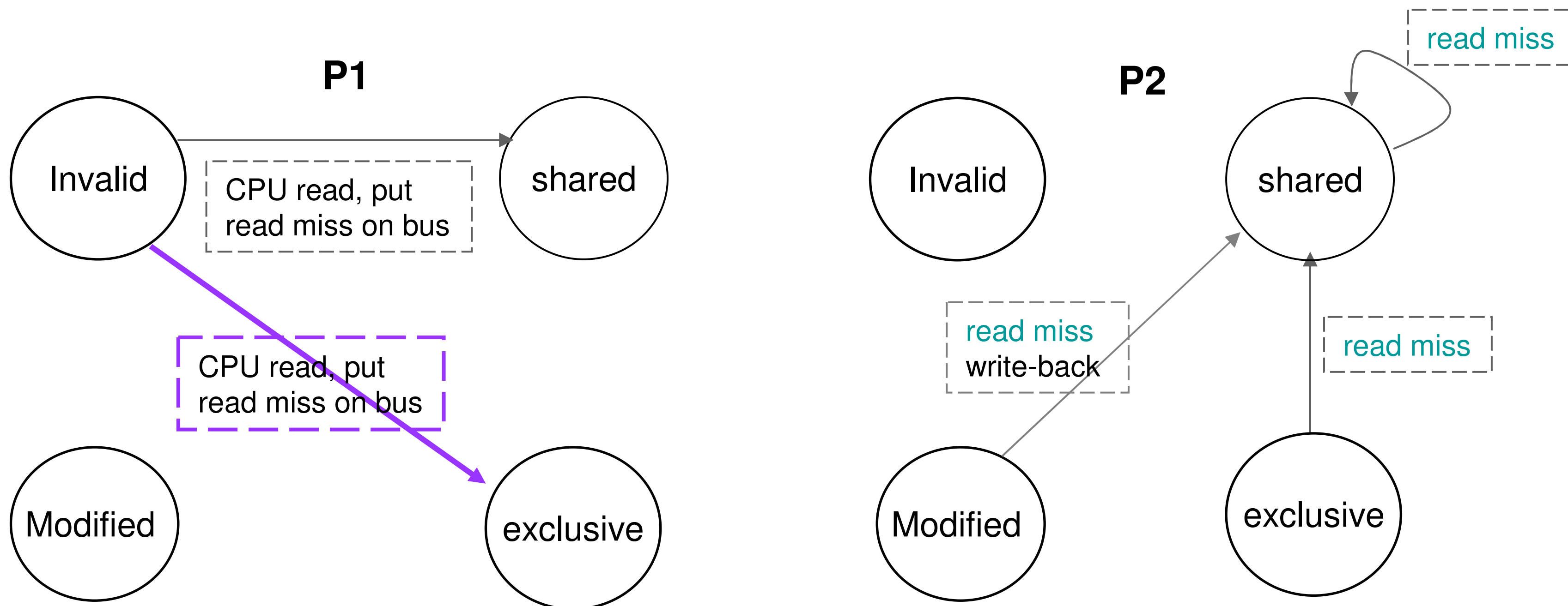
- Case 3: If ***P2*** has a "modified" copy of the line in the ***modified*** state, it blocks signal "memory read" and put data on bus. And then P2 transitions state from ***modified*** to ***shared*** (since ***P2*** goes to sate ***shared***, it must update line in main memory). ***P1*** reads the line from bus and transitions state from ***invalid*** to ***shared***.



Read miss [4]

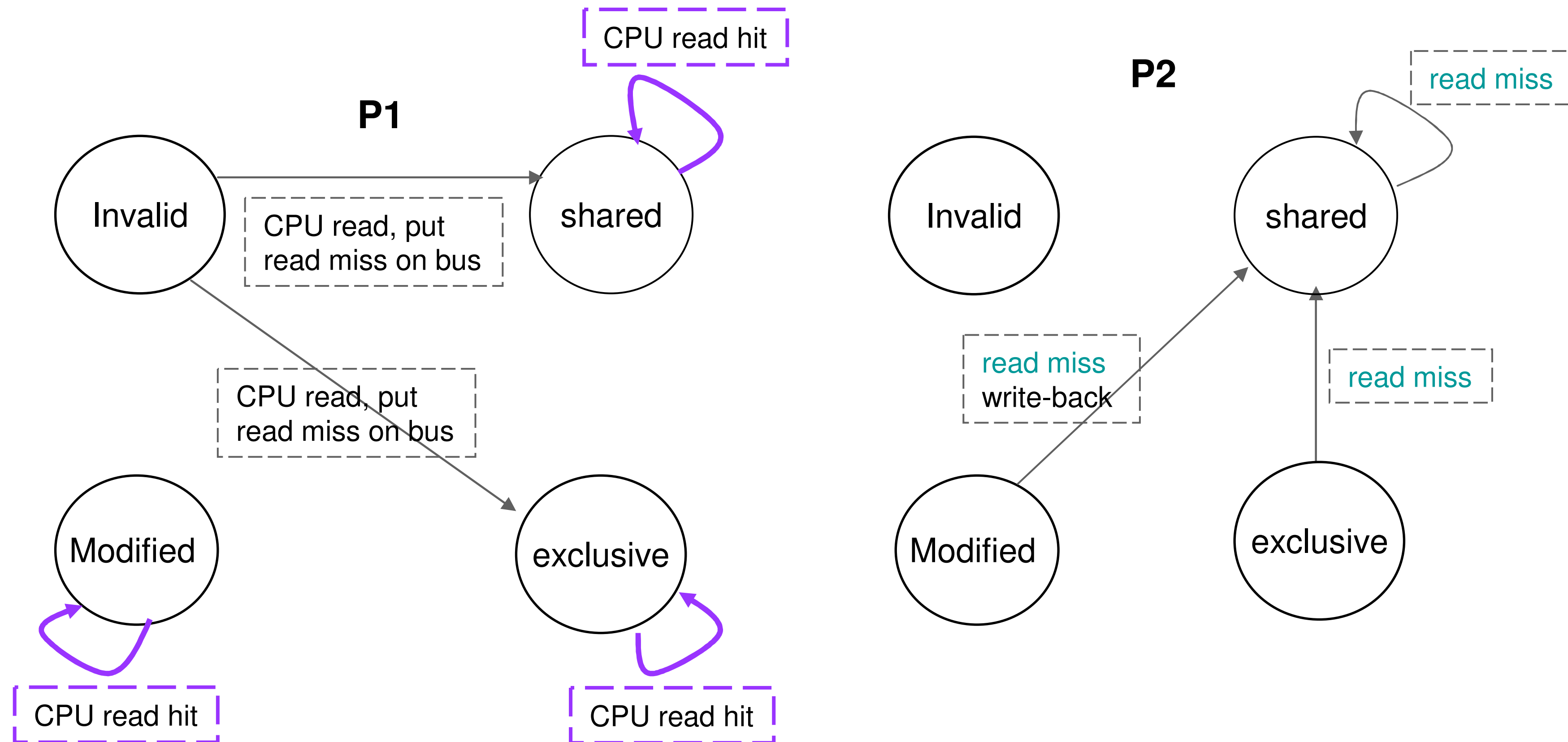
When ***P1*** has a read miss, then it initiates a memory read to read the line in main memory (or other cache). So ***P1*** inserts a read miss signal on bus that alerts other processors.

- Case 4: If no other cache has a copy of the line (clean or modified), then no signals are returned. ***P1*** is the only processor having the data so ***P1*** read data from main memory and transitions state from ***invalid*** to ***exclusive***.



Read hit

When ***P1*** has a read hit, then it read the line from cache directly. There is no state change, so state remains ***modified***, ***shared***, or ***exclusive.***



Write hit [1]

When ***P1*** has a write hit, then it update the line from cache directly.

- Case 1: If ***P1*** is in ***shared*** state, then it issues "write miss" signal to bus such that all processor sharing the line will change their state from ***shared*** to ***invalid*** (only ***P1*** has the data).
  P1 update the cache and transitions state from ***shared*** to ***modified***.

P1

Invalid
shared
Modified
exclusive
CPU read hit
CPU read, put read miss on bus
CPU write hit
CPU read, put read miss on bus
CPU read hit
CPU read hit

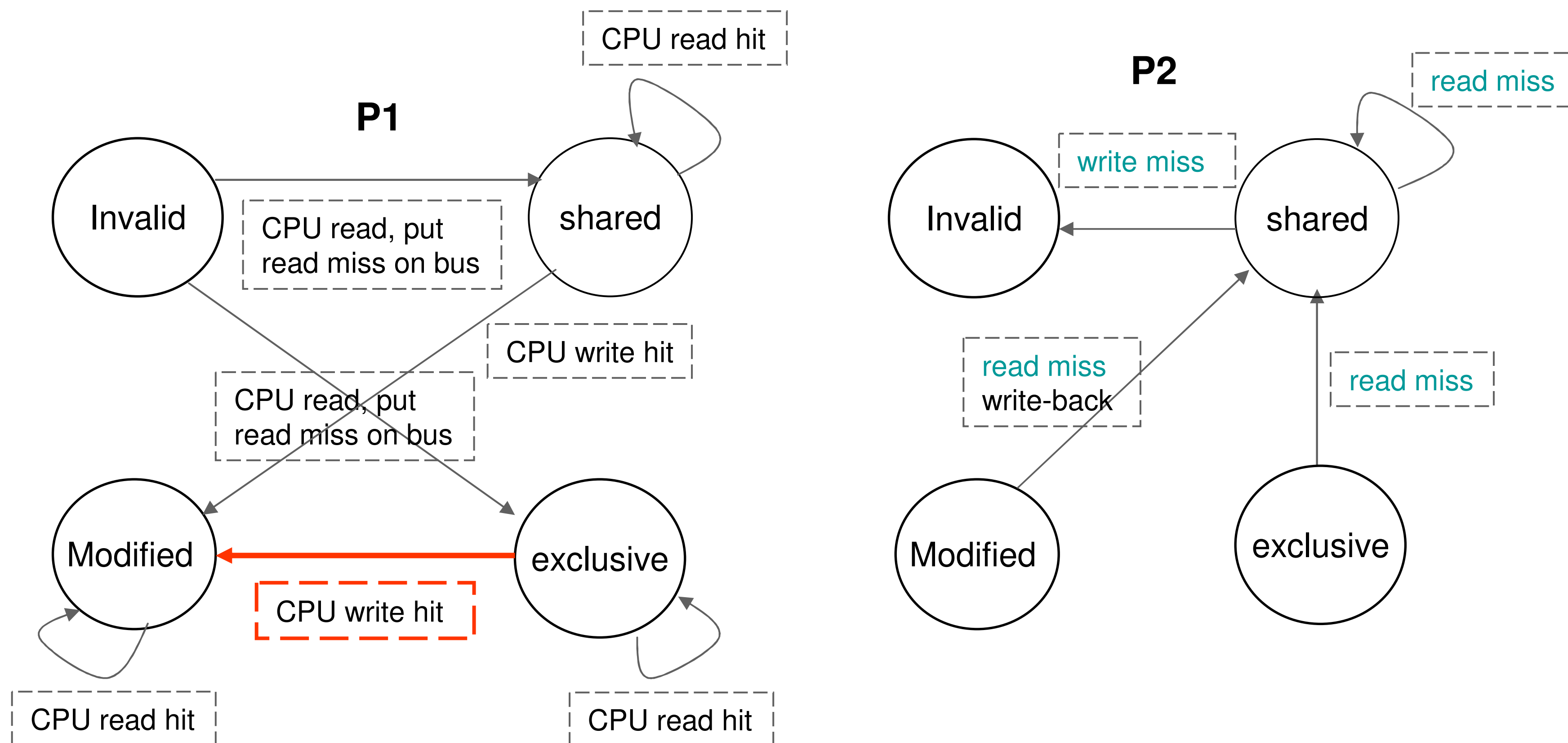
P2

Invalid
shared
Modified
exclusive
read miss
write miss
read miss write-back
read miss

Write hit [2]

When ***P1*** has a write hit, then it update the line from cache directly.

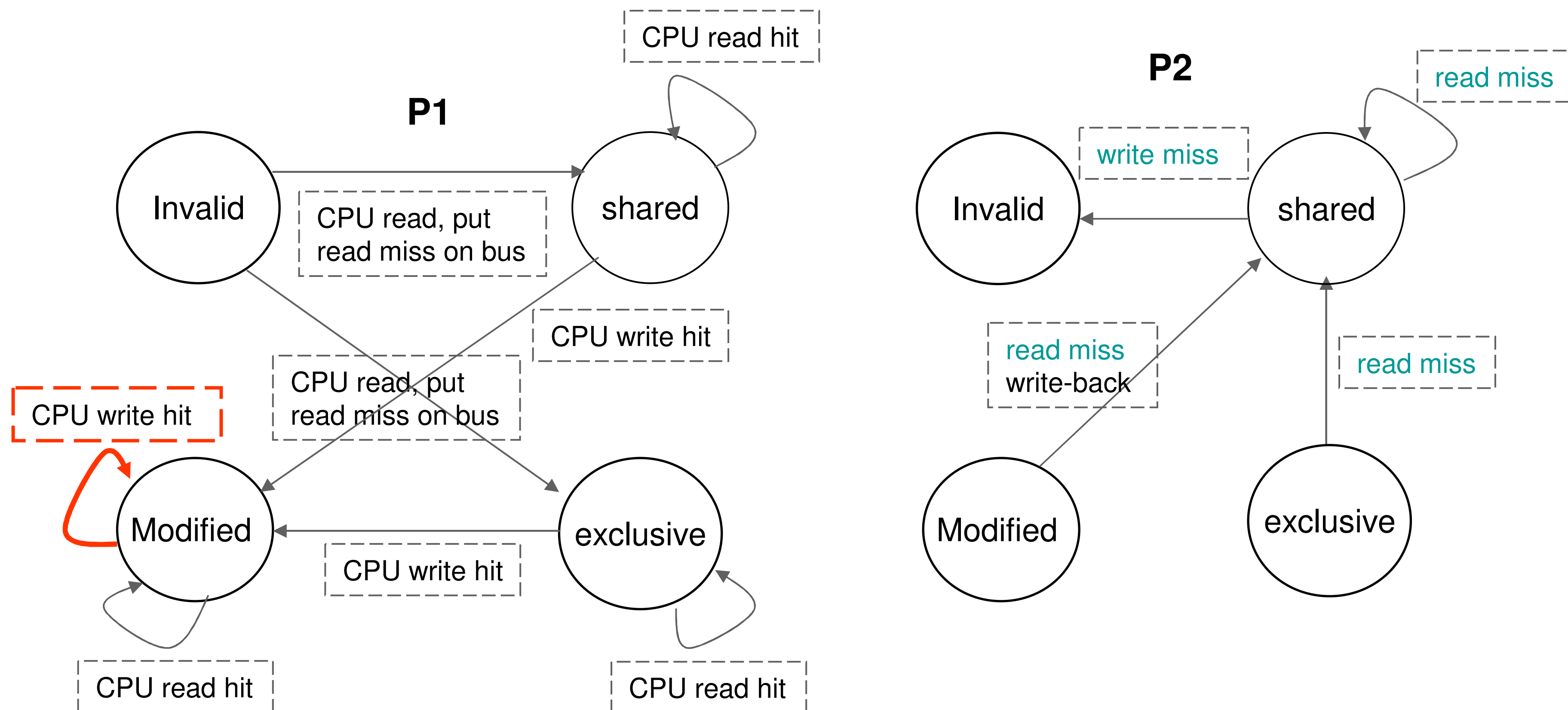
- Case 2: If ***P1*** is in ***exclusive*** state, then it updates the cache and transitions state from ***exclusive*** to ***modified*** since only ***P1*** has the data but this data is different from data in main memory, that is why ***P1*** must go to state ***modified***.



Write hit [3]

When ***P1*** has a write hit, then it update the line from cache directly.

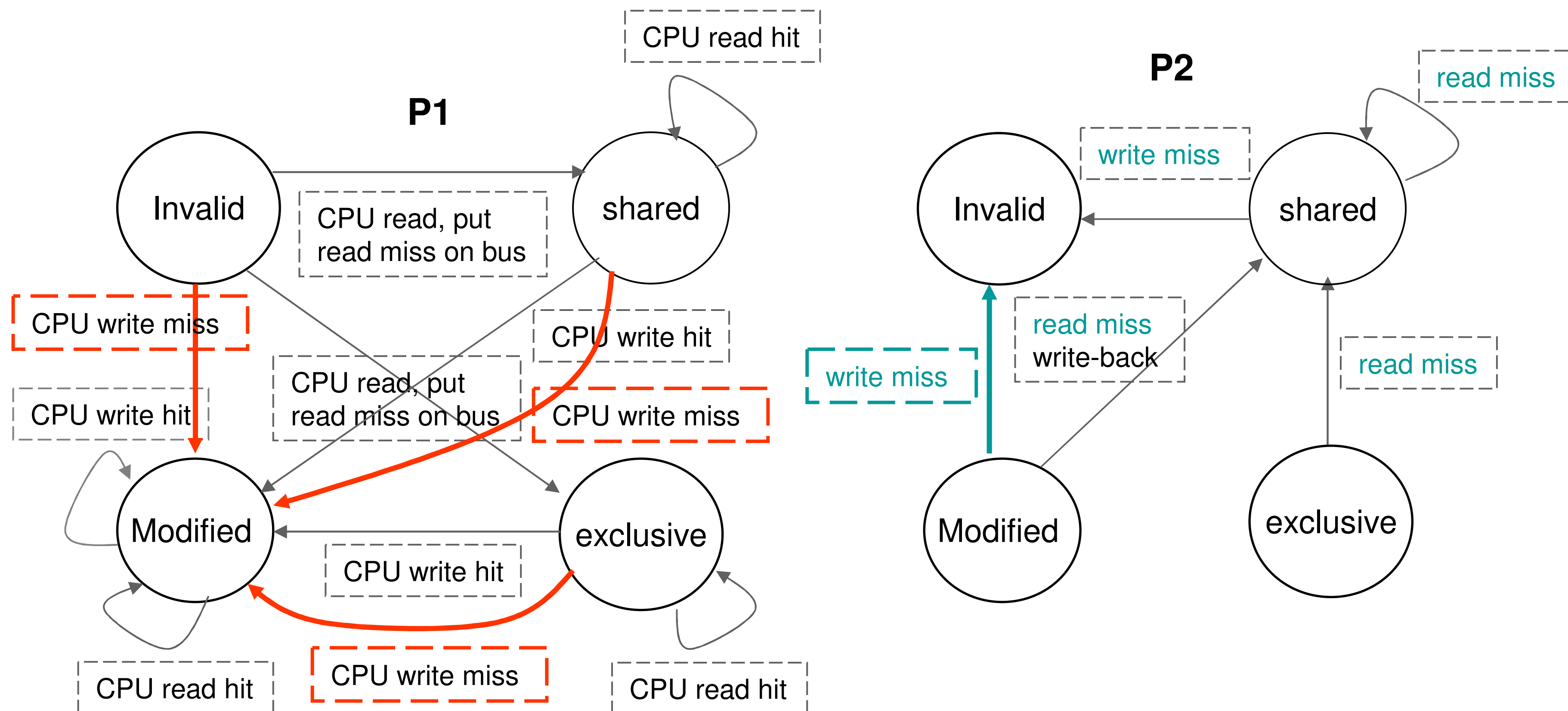
- Case 3: If ***P1*** is in ***modified*** state, then it updates the cache without state transition



Write miss [1]

When ***P1*** has a write miss (data is invalid or address conflict), then it issues a signal read-with-intent-to-modify (RWITM) to bus. After P1 update data in cache, then it transitions state to ***modified*** no matter which state (invalid, shared, exclusive) it locates.

- Case 1: If ***P2*** is in ***modified*** state, then ***P2*** must write-back data to main memory since ***P2*** will give its current data and P1 will have latest data. After write-back, P2 transitions state from ***modified*** to ***invalid***.

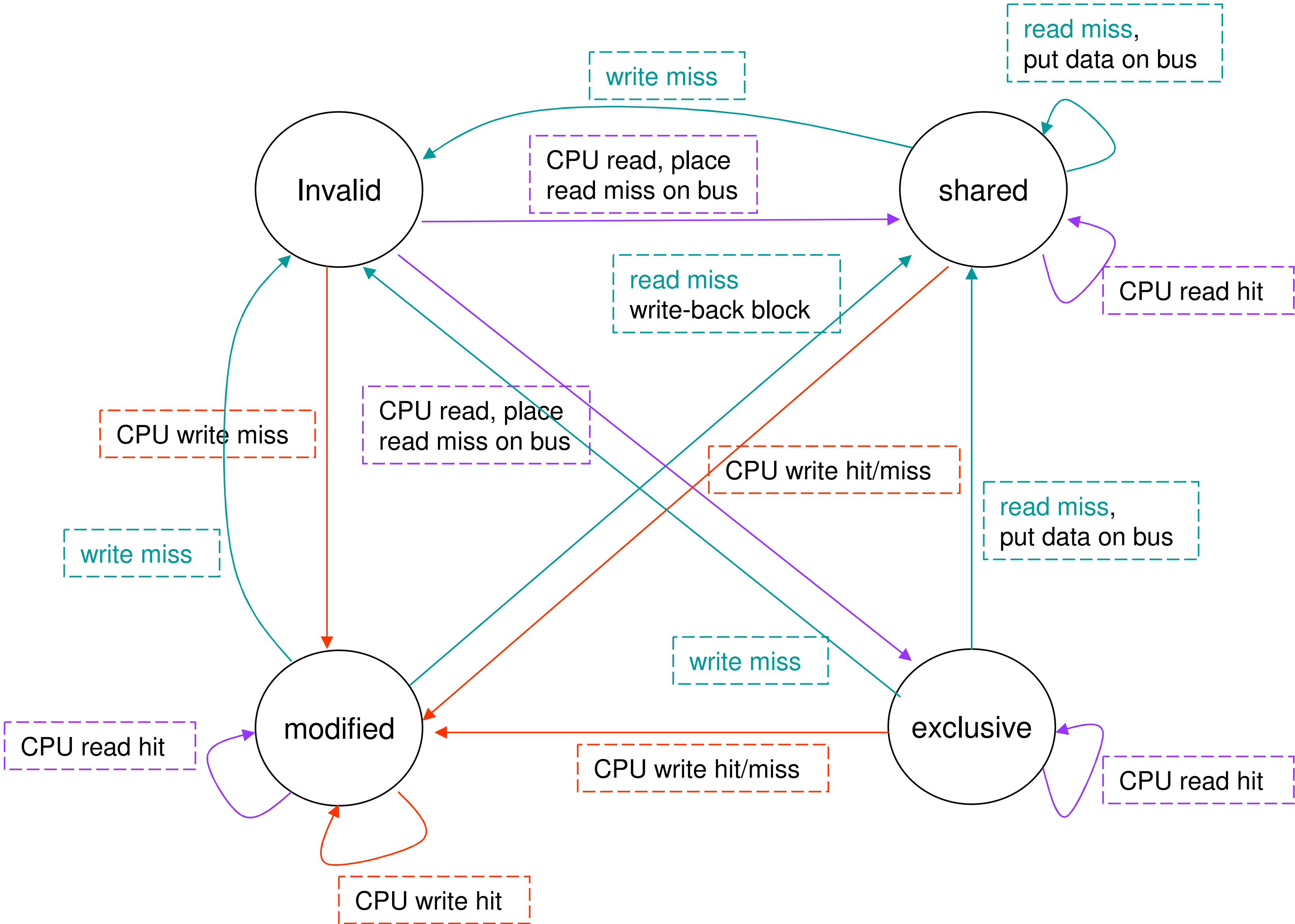


Write miss [2]

When ***P1*** has a write miss (data is invalid or address conflict), then it issues a signal read-with-intent-to-modify (RWITM) to bus. After P1 update data in cache, then it transitions state to ***modified*** no matter which state (invalid, shared, exclusive) it locates.

- Case 2: If ***P2*** is NOT in ***modified*** state, then P2 transitions state to ***invalid***.

Finite state controller of MESI protocol



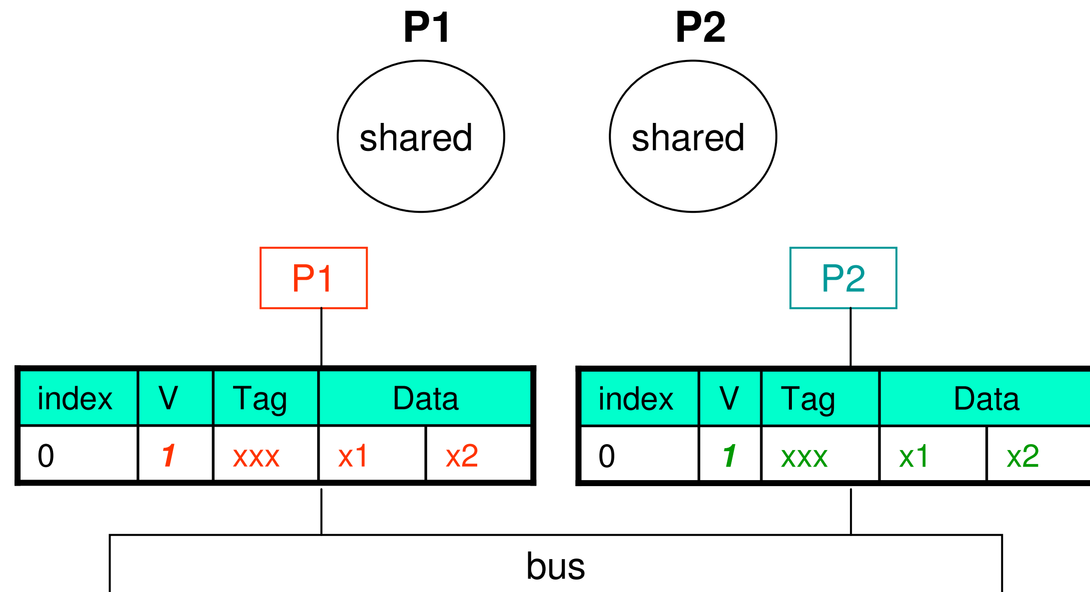
OutLine

- Basic of cache
- Cache coherence
- False sharing
- Summary

false sharing

"***False sharing***" occurs when two processors share two different part (words) that reside in the same block. The full block is exchanged between processors even though processors access different variables.

Time	P1	P2
1	Write *x1*	
2		Read *x2*
3	Write *x1*	
4		Write *x2*
5	Read *x2*	



Exercise 3 : Assume that words *x1* and *x2* are in the same cache block in a clean state of P1 and P2 which have previously read *x1* and *x2*.

Identify each miss as a true sharing miss or a false sharing miss by simple snooping protocol.

Example of false sharing [1]

```
3   #include <omp.h>
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <assert.h>
7   #include <math.h>
8   #include <qdatetime.h>
9
10  #define   NUM_THREADS    4
11
12  int main(int argc, char *argv[] )
13  {
14    long int N = 200*1024*1024 ;
15    long int i ;
16    float *a ;
17    int  th_id ;
18    float  max_A_partial[NUM_THREADS] ;
19    float  abs_A, max_A ;
20    QTime t; // QT timer
21
22    a = (float*) malloc( sizeof(float)*N ) ; assert(a) ;
23
24    t.start() ;
25    for ( i=0 ; i < N ; i++){ // a is increasing
26      a[i] = (float)i ;
27    }
28    printf("Time to initial a = %d (ms)\n", t.elapsed());
29
30    for ( i=0 ; i < NUM_THREADS ; i++){
31      max_A_partial[i] = 0 ;
32    }
34    t.start() ;
35  #pragma omp parallel default(none) num_threads(NUM_THREADS)  \
36       shared(a,N,max_A_partial) private(th_id,abs_A,i)
37    {
38      th_id = omp_get_thread_num();
39      #pragma omp for schedule( static ) nowait
40      for (i=0; i < N; i++){
41        abs_A = fabs( a[i] ) ;
42        max_A_partial[th_id] = QMAX( max_A_partial[th_id], abs_A) ;
43      }
44    } /* end of parallel section */
```

Objective: $\|a\|_\infty = \max_{0 \le j \le N-1} |a_j|$

a is increasing such that (line 26)

$$\|a\|_\infty = a[N-1]$$

Execute every time since ***a*** is increasing (line 42)

Example of false sharing [2]

Platform: octet1, with compiler icpc 10.0, -O0, size(a) = 800 MB

	1	2	4	8
max_A_partial	0x7fffaaca9568	0x7fff7fbaa048	0x7fffdba50ee8	0x7fffeff0b388
Time	1799 ms	6206 ms	4952 ms	19293 ms

Platform: octet1, with compiler icpc 10.0, -O2, size(a) = 800 MB

	1	2	4	8
max_A_partial	0x7fff4faab260	0x7fff3097cee0	0x7fffb13c8910	0x7fff4021f380
Time	475 ms	427 ms	238 ms	205 ms

Platform: octet1, with compiler icpc 10.0, -O0, size(a) = 6400 MB

	1	2	4	8
max_A_partial	0x7fff8443b9b8	0x7fff6a381828	0x7fffc4a01e98	0x7fff6fff7478
Time	14291 ms	46765 ms	90090 ms	113054 ms

Platform: octet1, with compiler icpc 10.0, -O2, size(a) = 6400 MB

	1	2	4	8
max_A_partial	0x7fffa51306c0	0x7fff53985ee0	0x7fff9f5edb40	0x7ffff9bfd130
Time	3848 ms	3501 ms	1814 ms	1427 ms

Example of false sharing [3]

```
[macrold@octet1 false_sharing]$
[macrold@octet1 false_sharing]$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Xeon(R) CPU           X5365  @ 3.00GHz
stepping        : 11
cpu MHz         : 2000.000
cache size      : 4096 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clfl
ush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx lm constant_tsc arch_perfmon pebs bts r
ep_good pni monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr dca lahf_lm
bogomips        : 5987.36
clflush size    : 64
cache_alignment : 64
address sizes   : 38 bits physical, 48 bits virtual
power management:
```

1: cache size — size of L2 cache

2: cache_alignment

3: address sizes

2 Cache line size (block size) = 64 byte

Exercise 4 : show all ***max_A_partial[NUM_THREADS]*** fall into the same cache block, then *false sharing* occurs.

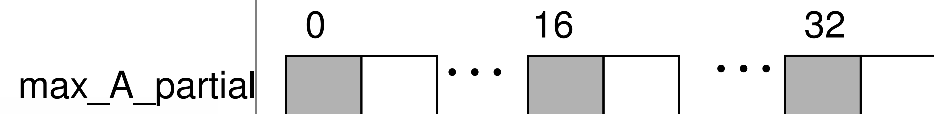
3 Address line is 48 bits, check it

Example of false sharing [4]

```
 9  #define   STRIDE        16
10  #define   NUM_THREADS    4
11
12  int main(int argc, char *argv[] )
13  {
14    long int N = 200*1024*1024 ;
15    long int i ;
16    float *a ;
17    int  th_id, index ;
18    float  max_A_partial[STRIDE * NUM_THREADS] ;
19    float  abs_A, max_A ;
20    QTime t; // QT timer

30    for ( i=0 ; i < STRIDE * NUM_THREADS ; i++){
31      max_A_partial[i] = 0 ;
32    }
33
34    t.start() ;
35  #pragma omp parallel default(none) num_threads(NUM_THREADS)  \
36       shared(a,N,max_A_partial) private(th_id,abs_A,i,index)
37    {
38      th_id = omp_get_thread_num();
39
40      index = th_id * STRIDE ;
41
42      #pragma omp for schedule( static ) nowait
43      for (i=0; i < N; i++){
44        abs_A = fabs( a[i] ) ;
45
46        max_A_partial[index] = QMAX( max_A_partial[index], abs_A) ;
47
48      }
49    } /* end of parallel section */
```

use non-adjacent location in array ***max_A_partial*** to avoid false sharing



Question 11: why do we choose STRIDE = 16?
Can we choose smaller value or larger value? Write program to test it

Example of false sharing [5]

Platform: octet1, with compiler icpc 10.0, -O0, size(a) = 800 MB

	1	2	4	8
max_A_partial	0x7fffa3abaf18	0x7fff75cc70e8	0x7ffff3486828	0x7fffad4e3788
Time	1782 ms	891 ms	454 ms	231 ms

Platform: octet1, with compiler icpc 10.0, -O2, size(a) = 800 MB

	1	2	4	8
max_A_partial	0x7fff39c50170	0x7fff17e2a300	0x7fff428b0890	0x7fff9a1ad550
Time	739 ms	400 ms	191 ms	184 ms

Platform: octet1, with compiler icpc 10.0, -O0, size(a) = 6400 MB

	1	2	4	8
max_A_partial	0x7fff9f906d68	0x7fff23808c28	0x7fff6afc0368	0x7fffa9c36ed8
Time	13609 ms	7196 ms	3416 ms	1708 ms

Platform: octet1, with compiler icpc 10.0, -O2, size(a) = 6400 MB

	1	2	4	8
max_A_partial	0x7fff190aa5c0	0x7fff470064e0	0x7fffeb4f0990	0x7fffdaa33dd0
Time	5882 ms	3077 ms	1490 ms	1097 ms

Question 11: performance is significant when number of threads increases, this proves "false sharing", right?

Exercise 5: chunk size versus false sharing

```
2  #include <omp.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <assert.h>
6  #include <qdatetime.h>
7
8  void randomInit( float* data, int size) ;
9
10 int main(int argc, char *argv[] )
11 {
12   long int N = 200*1024*1024 ;
13   int  thread_num = 8 ;
14   long int i;
15   float *a, *b, *c ;
16   QTime t; // QT timer
17
18   a = (float*) malloc( sizeof(float)*N ) ; assert(a) ;
19   b = (float*) malloc( sizeof(float)*N ) ; assert(b) ;
20   c = (float*) malloc( sizeof(float)*N ) ; assert(c) ;
21
22     t.start() ;
23   randomInit(a, N);
24   randomInit(b, N);
25     printf("To randomize a, b needs %d (ms)\n", t.elapsed());

27      t.start() ;
28 #pragma omp parallel default(none) num_threads(thread_num)  \
29      shared(a,b,c,N) private(i)
30   {
31     #pragma omp for schedule( static, 1 ) nowait
32     for (i=0; i < N; i++){
33       c[i] = a[i] + b[i];
34     }
35   } /* end of parallel section */
```

Chunk size = 1

Do experiment for chunk size = 1 with threads 1, 2, 4, 8. you can use optimization flag –O0 or –O2, what happens on timing? Explain.

What is “good” choice of chunk size?

Makefile

```
CC        = icc
CXX       = icpc
LEX       = flex
YACC      = yacc
CFLAGS    = -w -O2 -openmp -mp -g -pipe -Wall -
ector --param=ssp-buffer-size=4 -m64  -DQT_NO_
D_SUPPORT
CXXFLAGS = -w -O2 -openmp -mp -g -pipe -Wall -
ector --param=ssp-buffer-size=4 -m64  -DQT_NO_
D_SUPPORT
```

OutLine

- Basic of cache
- Cache coherence
- False sharing

- Summary

Summary [1]

pitfall 1: where can a block be placed

Scheme name	Number of sets	Blocks per sete
Direct mapped	Number of blocks in cache	Seconds for the program
Set associative	$\frac{\text{Number of blocks in cache}}{\text{associativity}}$	Associativity (typically 2-8)
Fully associative	1	Number of blocks in cache

pitfall 2: how is a block found

Associativity	Location method	Comparison required
Direct mapped	index	1
Set associative	Index the set, search among elements	Degree of associativity
Fully associative	Search all cache elements	Size of the cache

pitfall 3: which block should be replaced

- random: use hardware assistance, fast
- Least recently used (LRU): we need to keep track which one is used for longest time, it is slower.
- FIFO: first-in, first-out (least recently replaced)
- Cycle: the choice is made in a round-robin fashion

Summary [2]

pitfall 4: behavior of memory hierarchy, 3 C's

- ***Compulsory*** misses: first access to a block not in the cache.
- ***Capacity*** misses: cache cannot contain all the blocks needed during execution.
- ***Conflict*** misses: when multiple blocks are mapped into the same set in set-associative.

4-th C: cache Coherence : occurs in parallel architectures

Design change	Effect on miss rate	Possible negative performance effect
Increase size	Decrease capacity misses	May increase access time
Increase associativity	Decreases miss rate due to conflict misses	May increase access time
Increase block size	Decrease miss rate for a wide range of block sizes	May increase miss penalty

pitfall 5: how to utilize cache

- Loop-oriented optimization
- Software prefetching; a block of data is brought into the cache (L2 cache) before it is actually referenced, this will decrease miss rate.
  Example: search linked-list
- Hardware prefetching

Summary [3]

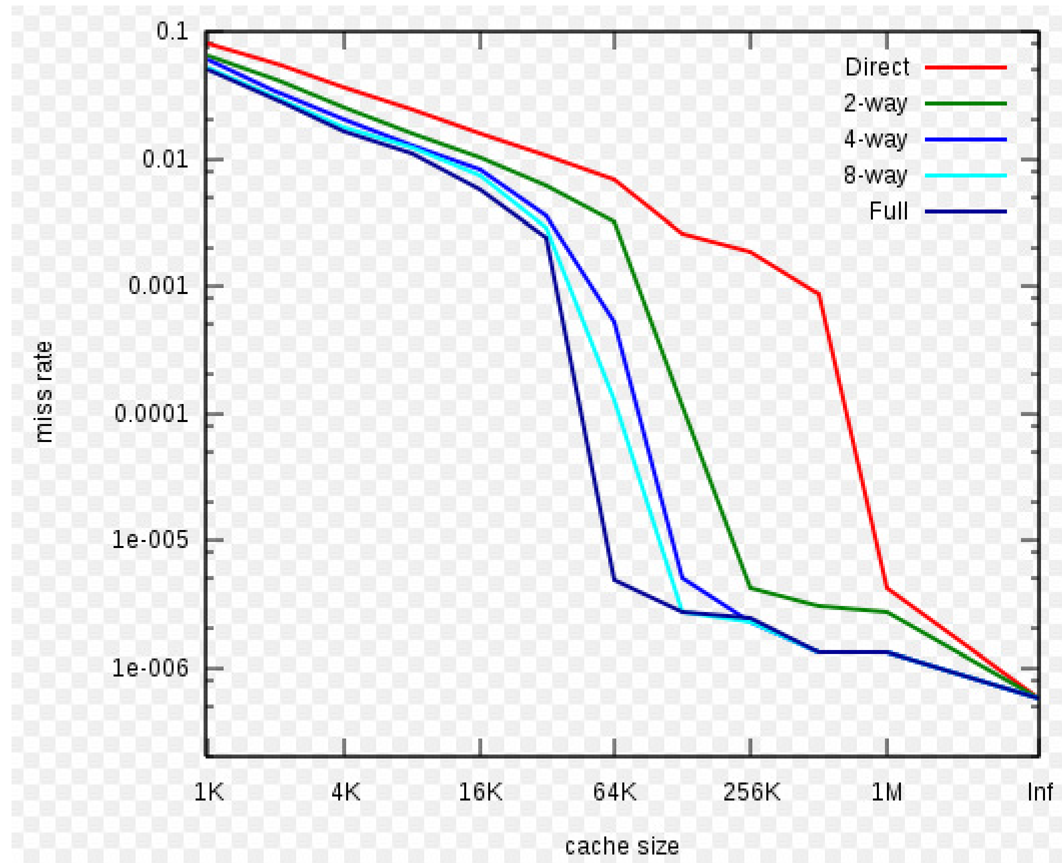
pitfall 6: cache information of commercial chip

CPU	L1 cache	L2 cache
Intel Pentium E2140	Data cache (per processor) 2 x 32 KBytes, 8-way set associative, 64-byte line size Instruction cache (per processor) 2 x 32 KBytes, 8-way set associative, 64-byte line size	(per processor) 2048 KBytes, 8-way set associative, 64-byte line size
Intel Core 2 Duo E4500	Data cache (per processor) 2 x 32 KBytes, 8-way set associative, 64-byte line size Instruction cache (per processor) 2 x 32 KBytes, 8-way set associative, 64-byte line size	(per processor) 2048 KBytes, 8-way set associative, 64-byte line size
Intel Core 2 Duo E6550	Data cache 2 x 32 KBytes, 8-way set associative, 64-byte line size Instruction cache 2 x 32 KBytes, 8-way set associative, 64-byte line size	4096 KBytes, 16-way set associative, 64-byte line size

Reference: http://www.pcdvd.com.tw/printthread.php?t=773280

Summary [4]

pitfall 7: larger associative?



Miss rate versus cache size on the integer portion of SPEC CPU2000

Reference: http://en.wikipedia.org/wiki/CPU_cache

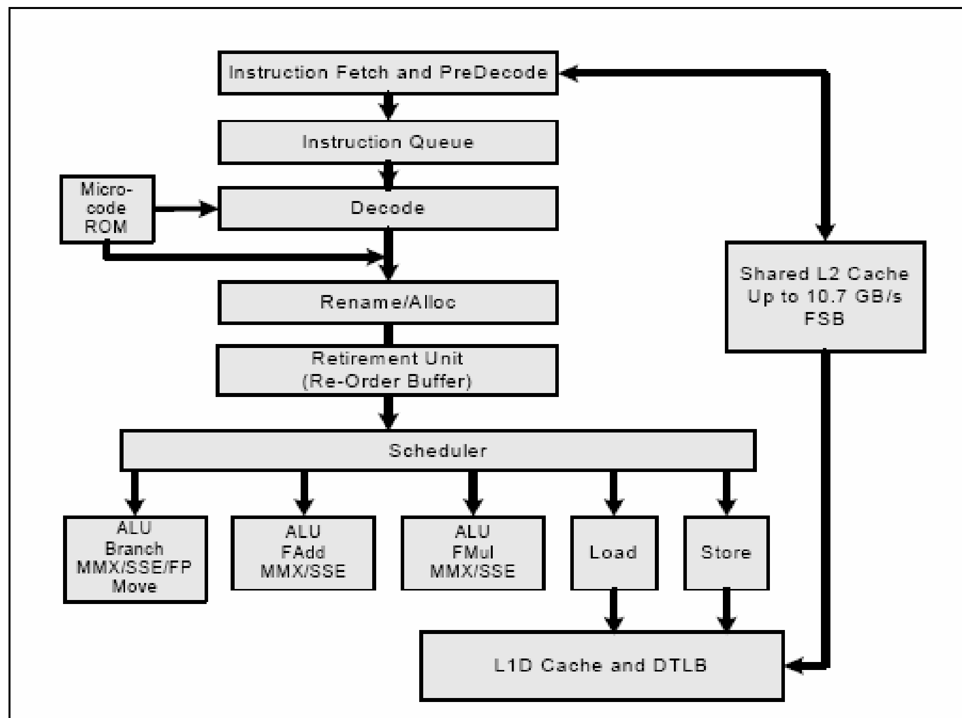
Further read: http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/

Summary [5]

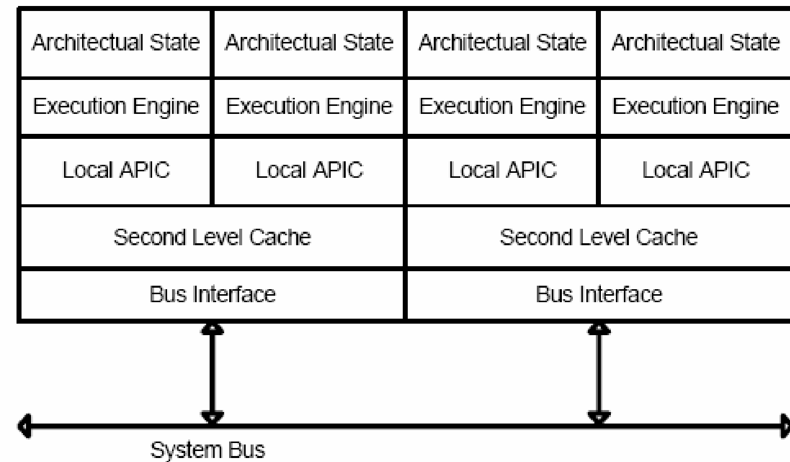
pitfall 8: L1-L2 Cache Consistency

- L1 cache does not connect to the bus directly but L2 cache does
- L1 data cache and L2 cache have two MESI status flags per cache line.
  - L1 adopts write-through policy: L1 write-though to L2, not to main memory, L1 cache is included in L2 cache (any data in L1 cache must be found in L2 cache)
  - L1 adopts write-back policy: more complicated, Intel Processor adopts this.

Question 11: what is cost of read/write miss?



The Intel Core Microarchitecture



Intel 64 processor, quad-core

Summary [6]

Cache parameters of processors based on Intel Core Microarchitecture

level	capacity	Associativity (ways)	Line size (bytes)	Access Latency (clocks)	Access Throughput (clocks)	Write update policy
Instruction cache	32 KB	8	N/A	N/A	N/A	N/A
L1 data cache	32 KB	8	64	3	1	Write-back
L2 cache	2, 4 MB	8 or 16	64	14	2	Write-back
L3 cache	3, 6 MB	12 or 24	64	15	2	Write-back

Characteristics of fetching first 4 bytes of different localities

	Load		**Store**	
Data locality	**Latency**	**Throughput**	**Latency**	**Throughput**
DCU (L1 data cache)	3	1	2	1
DCU of other core in modified state	14 + 5.5 bus cycles	14 + 5.5 bus cycles	14 + 5.5 bus cycles	
L2 cache	14	3	14	3
memory	14 + 5.5 bus cycles + memory	Depends on bus read protocol	14 + 5.5 bus cycles + memory	Depends on bus write protocol

Throughput is number of cycles to wait before the same operation can start again

Summary [7]

pitfall 9: cache statistics

- ***Address:*** how the address is decomposed into the tag and cache index.
  - cache index selects which line of the cache is checked;
  - tag field of the address is matched against the tag entry for the cache line to determine if there was a hit. The data select field selects the appropriate word/byte of the cache data.

- **Cost:** breaks out the cost of each of the cache components.
  - Register/SRAM is the storage required to hold cache data, tag information, valid bit and, if necessary, the dirty bit.
  - Comparators are used to perform the tag comparisons, and
  - 2-to-1 muxes (multiplexer) are needed when the words/line > 1 to select the appropriate data word to return to the CPU.

- **Performance:** enumerates how the cache has performed on all the memory accesses since the simulation was last reset.
  Example : a cache access takes 1 cycle and misses take an additional 4 cycles to access the first word from main memory plus 1 additional cycle to fetch subsequent words when words/line > 1. When the cache is off, each memory access takes 4 cycles.

Exercise 6: Can you write a cache simulator to report (1) number of cache hit (2) number of cache miss in a single processor?