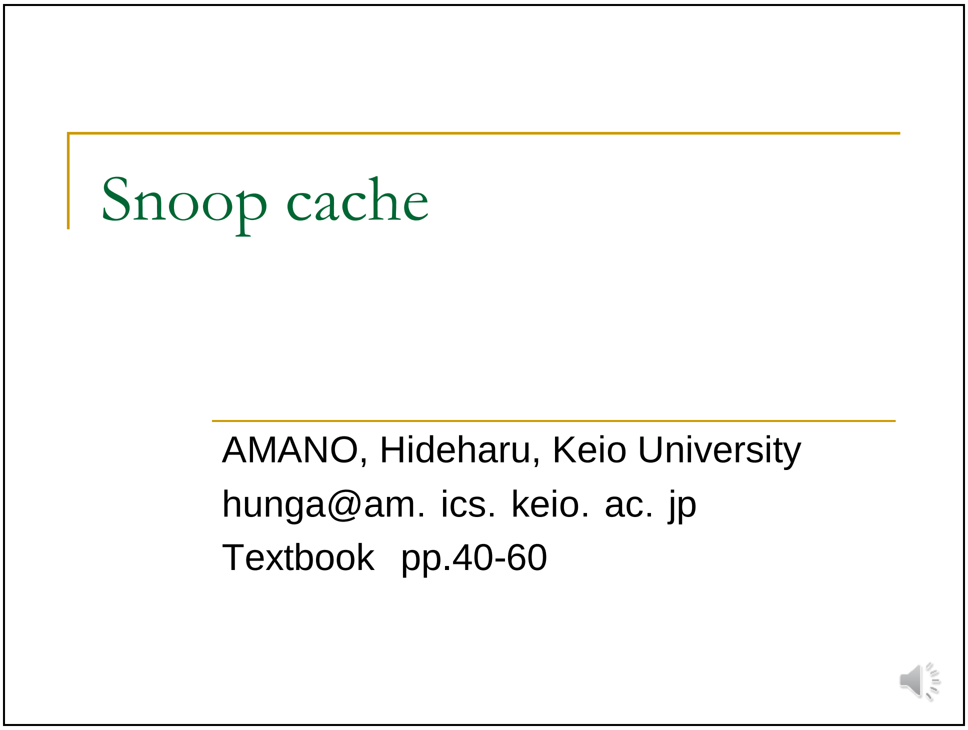
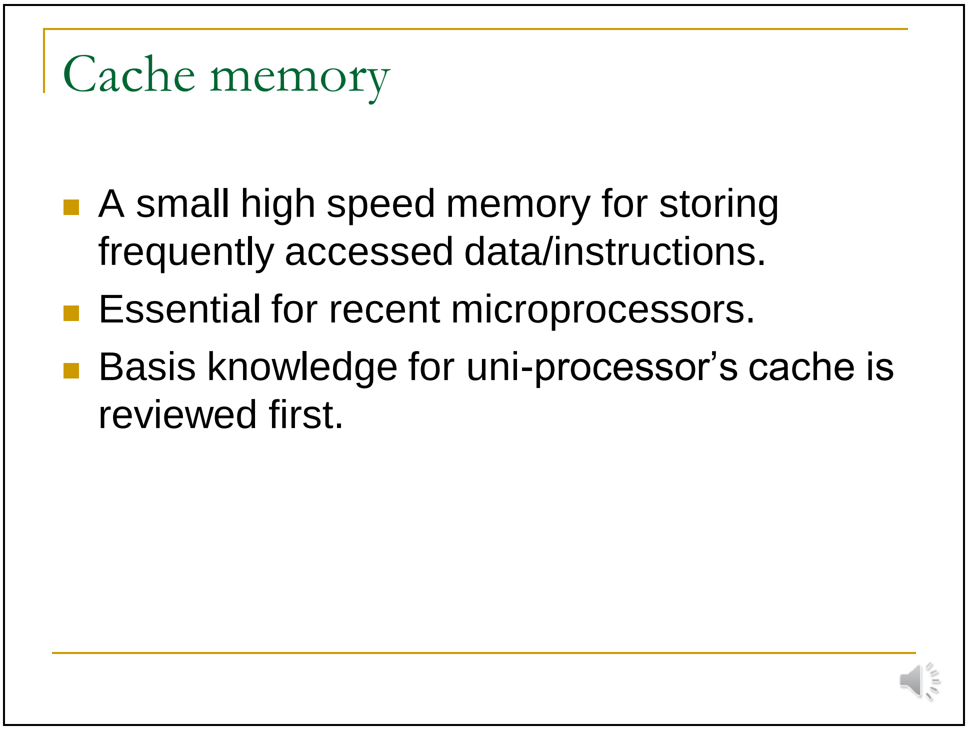


Snoop cache

AMANO, Hideharu, Keio University
hunga@am. ics. keio. ac. jp

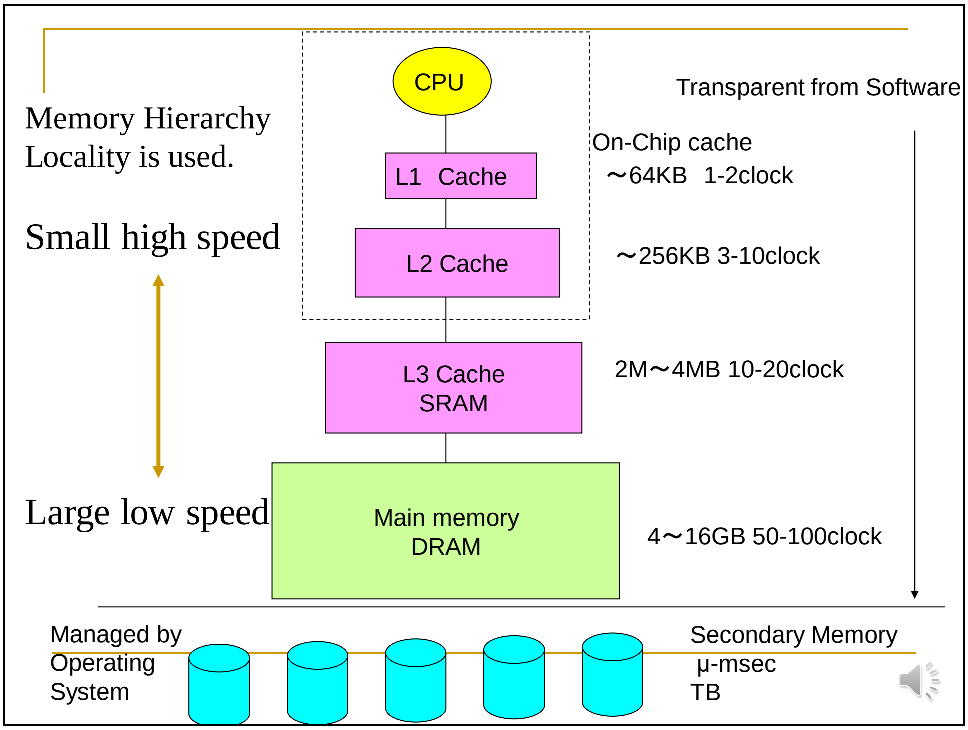
Textbook pp.40-60



Cache memory

- A small high speed memory for storing frequently accessed data/instructions.
- Essential for recent microprocessors.
- Basis knowledge for uni-processor’s cache is reviewed first.

First of all, let’s review the cache memory. It is a small high speed memory for storing frequently accessed data or instructions. It is essential for modern computers, so most of you know of it.



Memory Hierarchy
Locality is used.

Small high speed

Large low speed

CPU

L1 Cache

L2 Cache

L3 Cache
SRAM

Main memory
DRAM

Transparent from Software

On-Chip cache
～64KB 1-2clock

～256KB 3-10clock

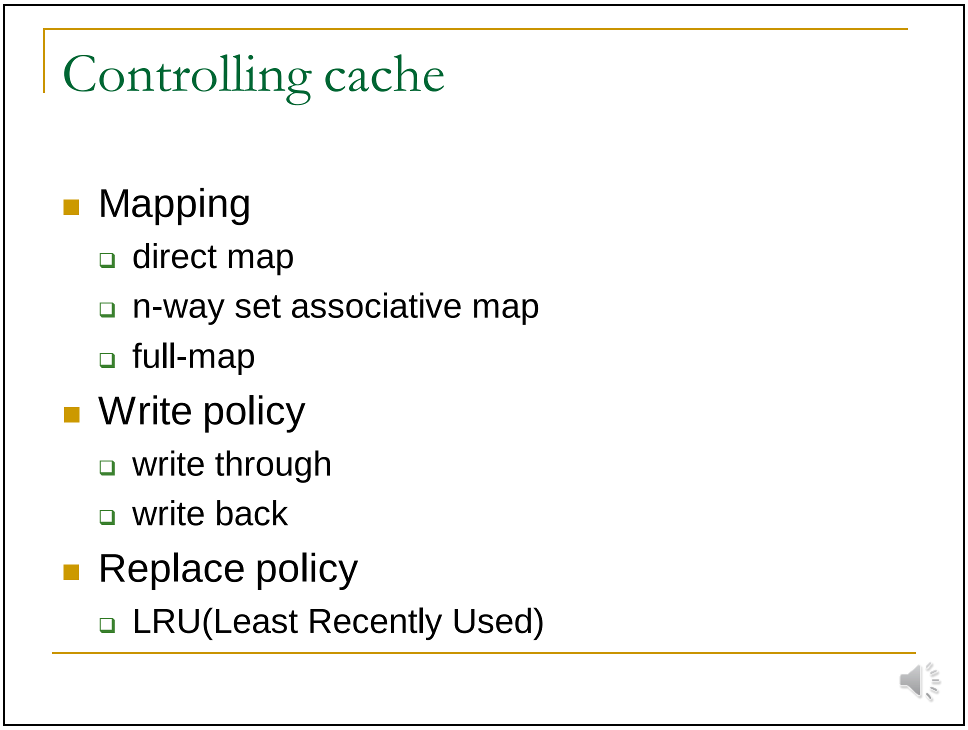
2M～4MB 10-20clock

4～16GB 50-100clock

Managed by
Operating
System

Secondary Memory
μ-msec
TB

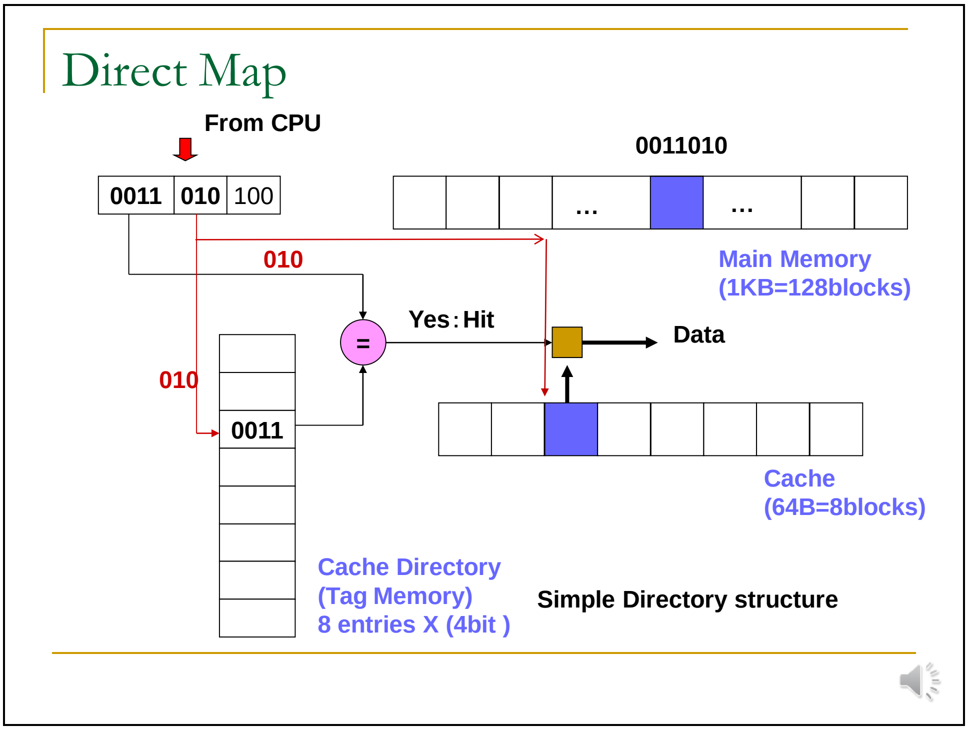
I showed this diagram of memory hierarchy in the previous lesson. In this diagram, there are three levels, but I simplize the structure to only a single one.



Controlling cache

- Mapping
  - direct map
  - n-way set associative map
  - full-map
- Write policy
  - write through
  - write back
- Replace policy
  - LRU(Least Recently Used)

Important design issues of cache are shown here.



Direct Map

From CPU

0011 010 100

0011010

…

…

010

Main Memory
(1KB=128blocks)

Yes : Hit

=

Data

010

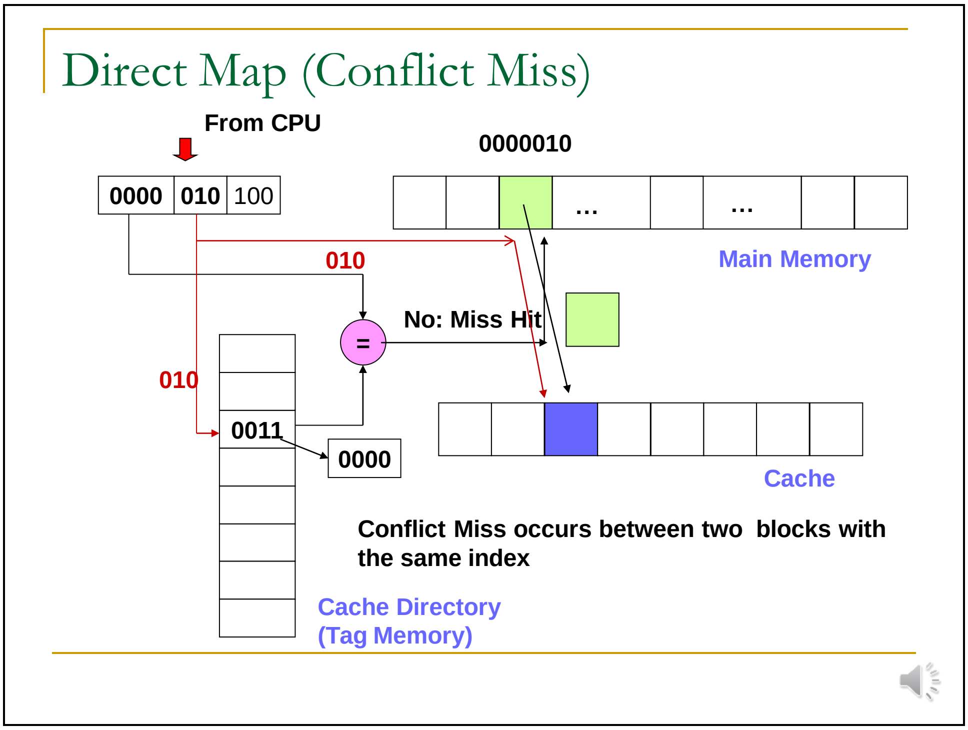
0011

Cache
(64B=8blocks)

Cache Directory
(Tag Memory)
8 entries X (4bit)

Simple Directory structure

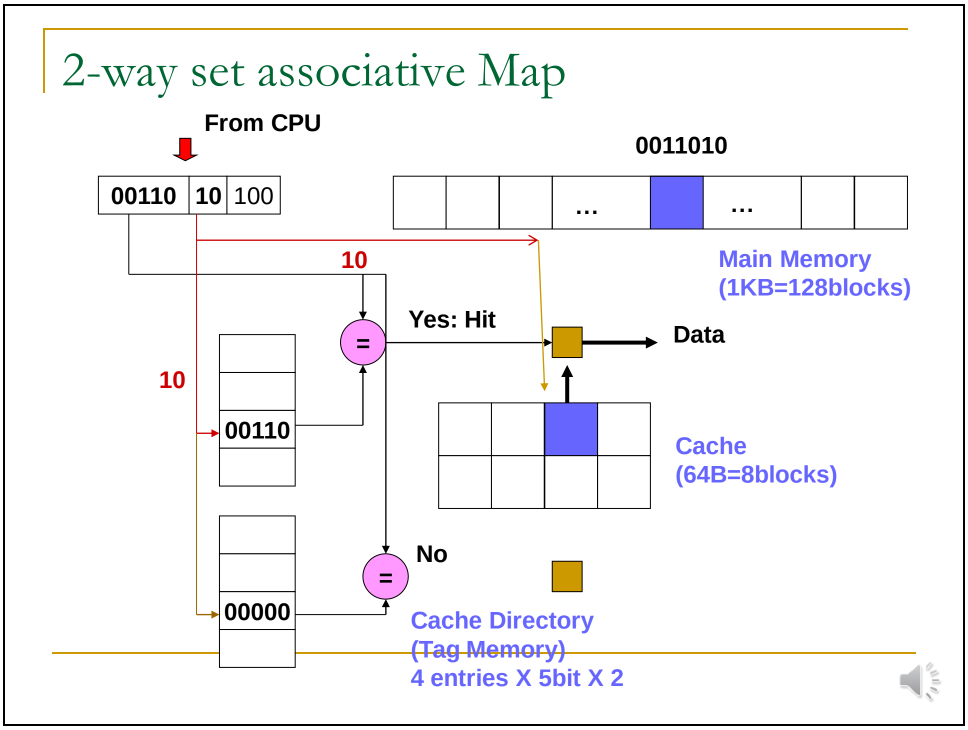
Let me explain the basic structure of cache with this simple figure for very small cache system. Here, the main memory is 1KB and cache is 64B. Cache system is managed with small data block. Here a block is 8 bytes. That is this cache can store 8 blocks. Of course, the size is too small but the structure itself is the same as the practical cache. And also note that the main memory is L2 cache for the multi-core CPU. So, I will use this figure in order to save the number of digits. There are 128 blocks in the main memory, and a block is stored into the cache block whose least three bits are the same. Here this 0011010 block is stored into 010 of the cache memory. These three bits are called the index. The other 4 bits are used as an identifier of the block, and called key or tag. The cache directory or tag memory is provided to keep the tag that is stored in the cache. This example, 0011 is stored here. When address from CPU is given, the cache directory and the cache are referred at the same time by the index, and if the upper 4 bits of the address matches to the key stored in the cache memory, it means that the target block is in the cache. It is called hit. In this case, the read out data is forwarded to the requesting CPU. This simplest mapping is called the direct map.



Direct Map (Conflict Miss)

Conflict Miss occurs between two blocks with the same index

If the upper 4bit address from the CPU does not match the key from the directory, it means that the target block is not stored in the cache. This is called cache miss. In this case, the target block must be fetched from the main memory and stored in the cache. At the same time, the directory is rewritten with the corresponding key. In this case, 0000. This operation is called replace. In the direct map cache, two blocks whose index is the same cannot stored in the cache memory. The cache miss by conflicting the index is called the conflict miss. The miss ratio of the direct map cache is not good because of this type miss.



2-way set associative Map

From CPU

00110 | 10 | 100

0011010

…

…

Main Memory
(1KB=128blocks)

10

Yes: Hit

Data

10

00110

Cache
(64B=8blocks)

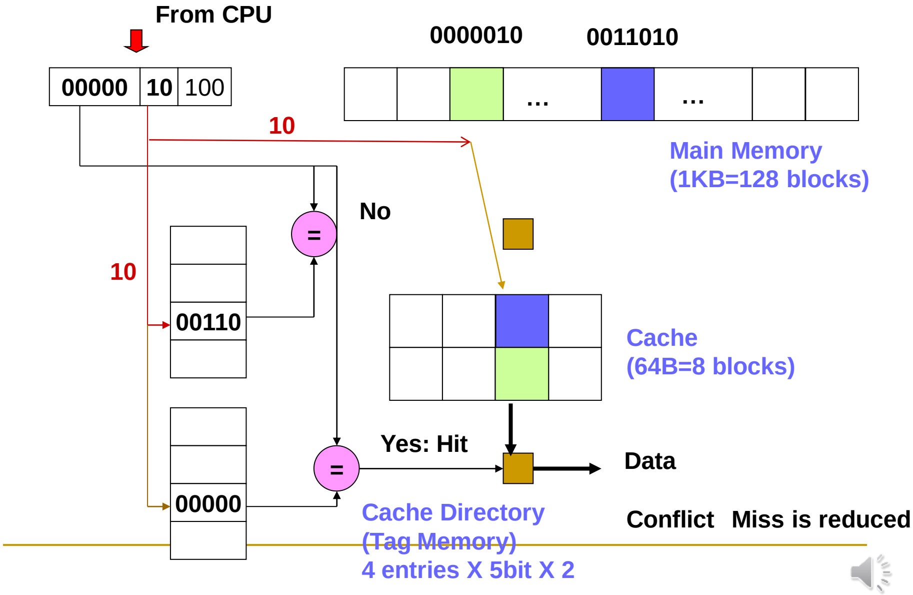
No

00000

Cache Directory
(Tag Memory)
4 entries X 5bit X 2

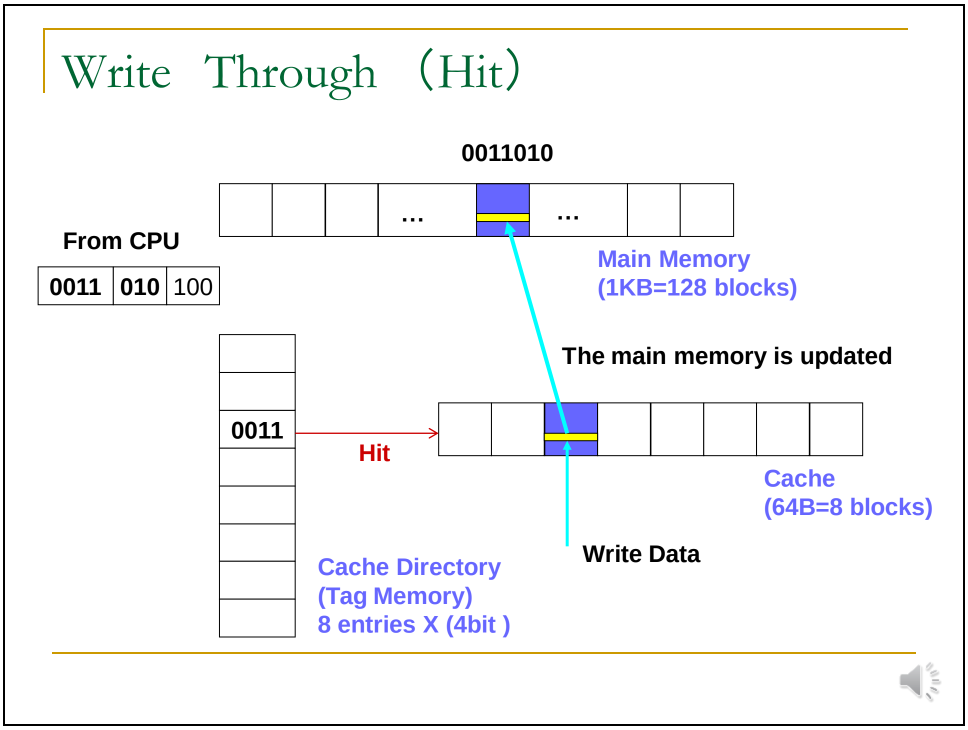
In order to improve it, two cache blocks form a set, and the index is assigned to each set. Here, there are 4 sets, and least two digits of the block number is used for the index. Here index is 10, and the block 0011010 is stored here. Two cache directories are provided for each way of the cache, and referred by the same index in parallel. The comparison is also done in parallel, and if either of them matches, the data is forwarded to the CPU. This structure is called 2-way set associative map. Apparently, the flexibility of storing the block is improved compared with the direct map method.

2-way set associative Map



Conflict Miss is reduced

The case that causes the conflict can be saved because they can be stored in the different ways like this. Similarly, 4-way, 8-way and more can be defined. But here I will omit the explanation.



Write Through (Hit)

0011010

From CPU

0011	010	100

0011

Hit

Main Memory
(1KB=128 blocks)

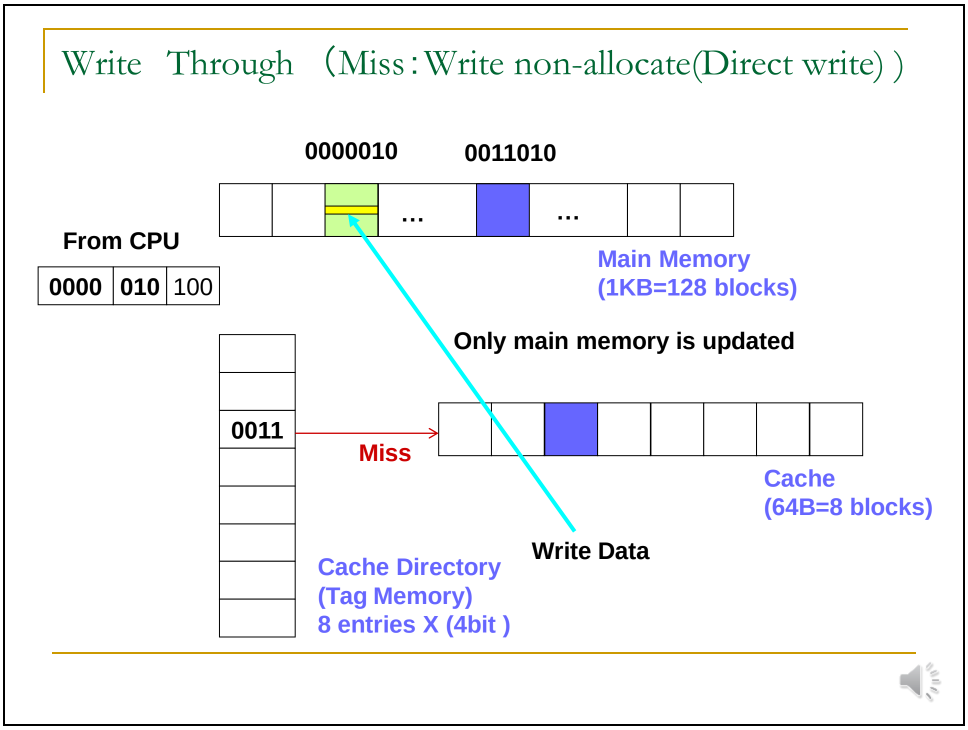
The main memory is updated

Cache
(64B=8 blocks)

Write Data

Cache Directory
(Tag Memory)
8 entries X (4bit)

For the read operation, when cache is hit, the data are forwarded to the CPU. If miss happens, the main memory is accessed and the fetched block is filled into the cache. However, for the write operation, there are two policies. One is write through. In this policy, when write request hits, the written data are forwarded to the main memory directly.



Write Through (Miss：Write non-allocate(Direct write))

0000010

0011010

…

…

From CPU

0000	010	100

Main Memory
(1KB=128 blocks)

Only main memory is updated

0011

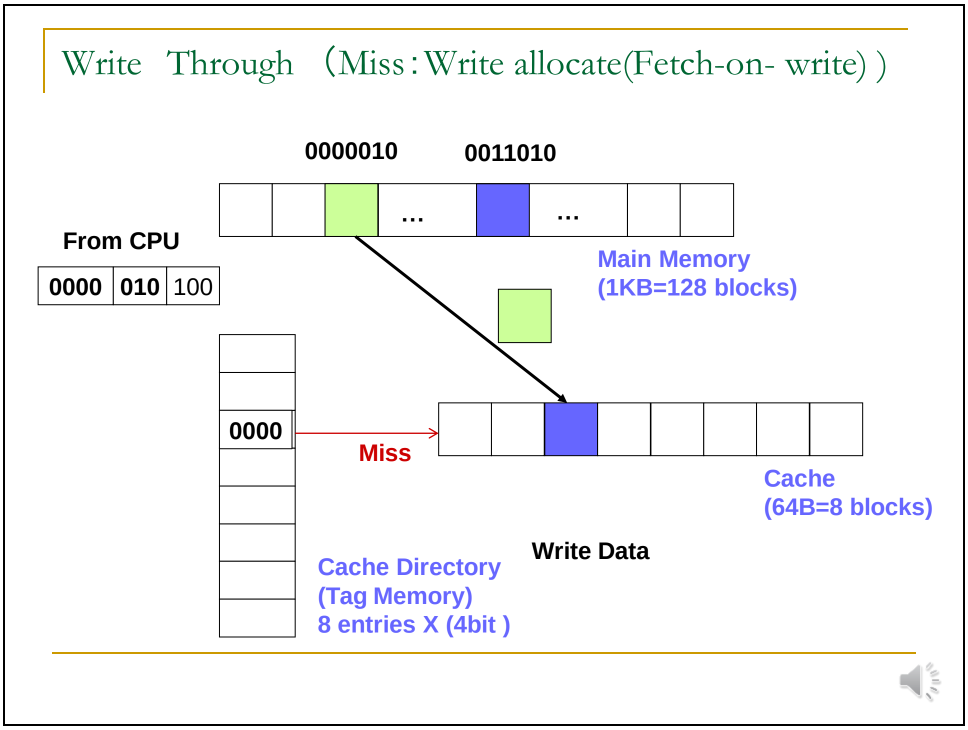
Miss

Cache
(64B=8 blocks)

Write Data

Cache Directory
(Tag Memory)
8 entries X (4bit)

There are two policies when write request miss-hits. Write non-allocate or direct write bypasses the cache and the data are directly written into the main memory. This policy is easy to be implemented, but the hit ratio is slightly degraded.



Write Through （Miss：Write allocate(Fetch-on- write))

0000010

0011010

…

…

From CPU

0000	010	100

Main Memory
(1KB=128 blocks)

0000

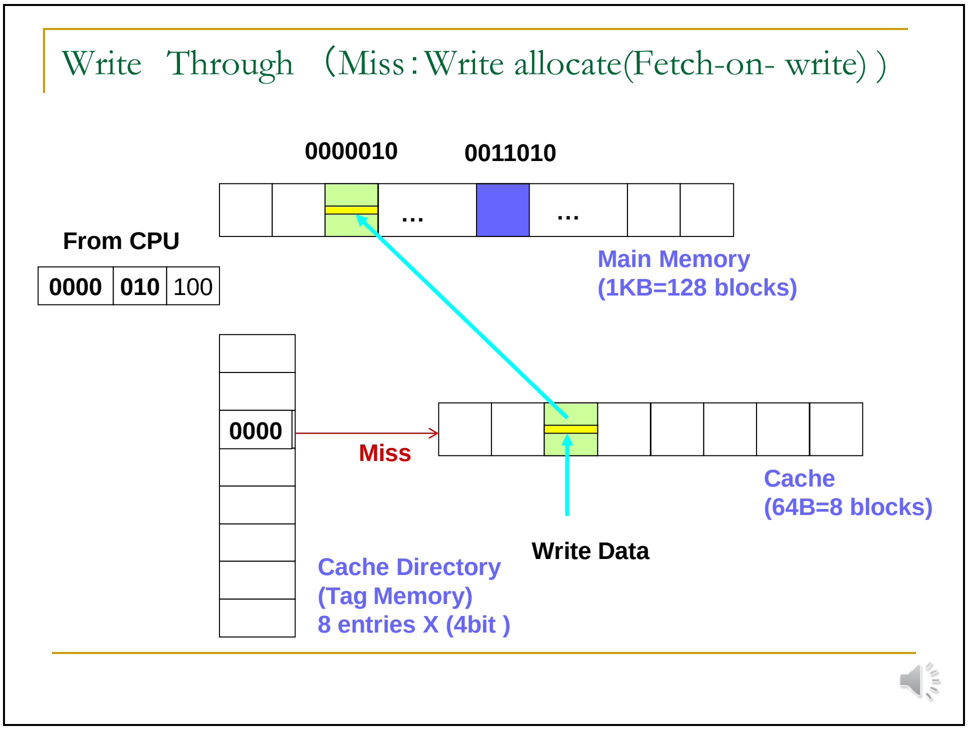
Miss

Cache
(64B=8 blocks)

Write Data

Cache Directory
(Tag Memory)
8 entries X (4bit)

Write allocated or fetch-on-write fetches the block like read-miss happens.



Write Through （Miss：Write allocate(Fetch-on- write))

0000010

0011010

…

…

From CPU

0000	**010**	100

Main Memory

(1KB=128 blocks)

0000

Miss

Cache

(64B=8 blocks)

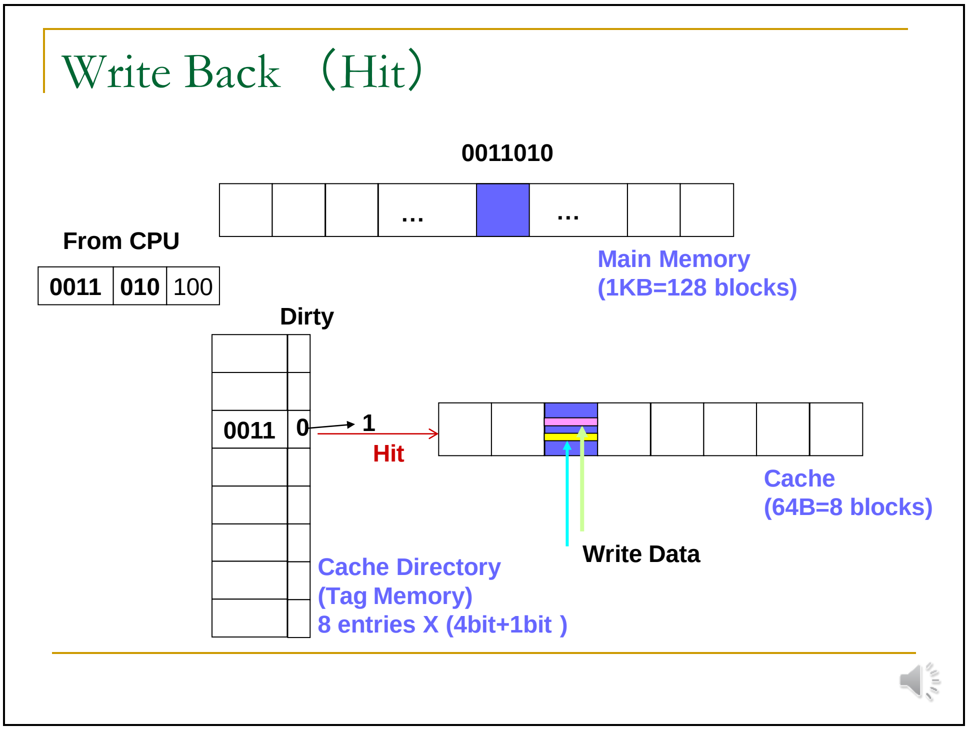
Write Data

Cache Directory

(Tag Memory)

8 entries X (4bit)

After the replacement, the data are written just like the case of write hits. Because of the locality of access, the replaced block may be accessed soon. So, the hit ratio is slightly better than that of write-non-allocate policy.



Write Back （Hit）

0011010

From CPU

0011 | 010 | 100

Main Memory
(1KB=128 blocks)

Dirty

0011 0 1

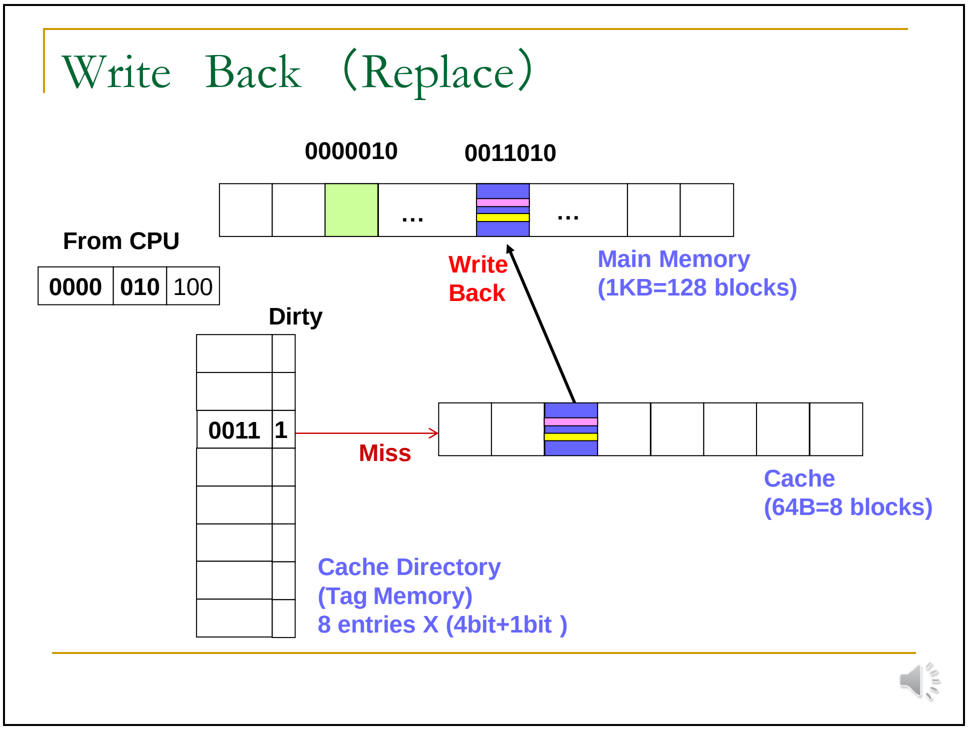
Hit

Cache
(64B=8 blocks)

Write Data

Cache Directory
(Tag Memory)
8 entries X (4bit+1bit)

Another policy is write back. In this policy, the write data are written into only cache, but the main memory are not updated. In this case, the content of cache is different from the main memory. In order to show it, the dirty bit is provided to every directory entry and when first write hits, the bit turns on.



Write Back (Replace)

0000010

0011010

…

…

From CPU

0000 010 100

Write Back

Main Memory
(1KB=128 blocks)

Dirty

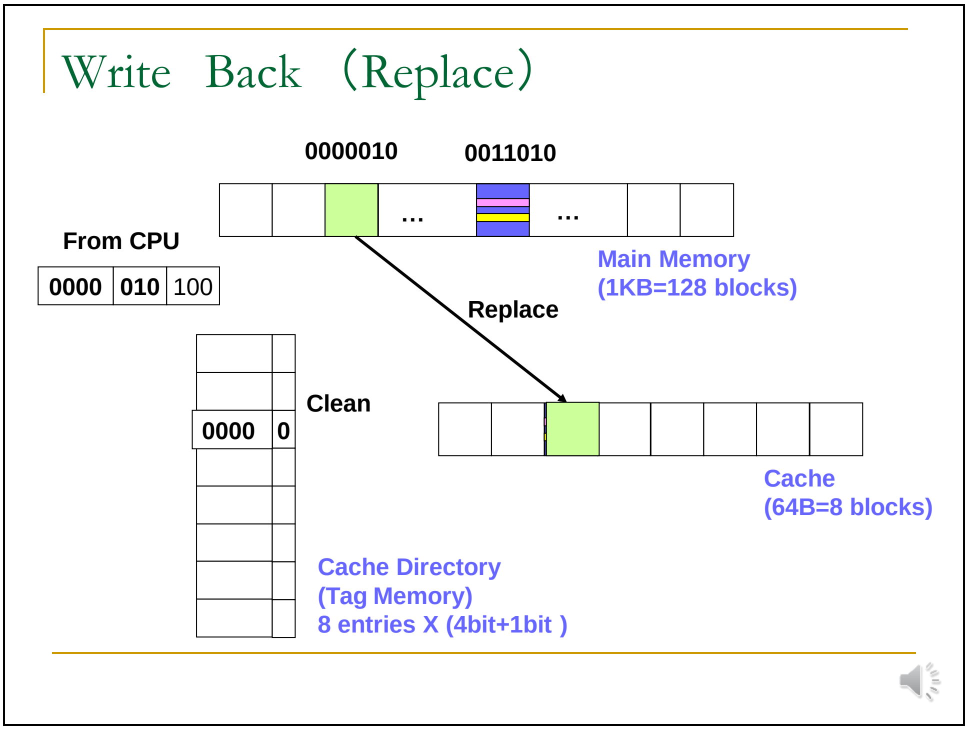
0011 1

Miss

Cache
(64B=8 blocks)

Cache Directory
(Tag Memory)
8 entries X (4bit+1bit)

When the dirty cache block becomes the target of replacement, it must be written back.



Write Back (Replace)

0000010

0011010

…

…

From CPU

0000	010	100

Main Memory
(1KB=128 blocks)

Replace

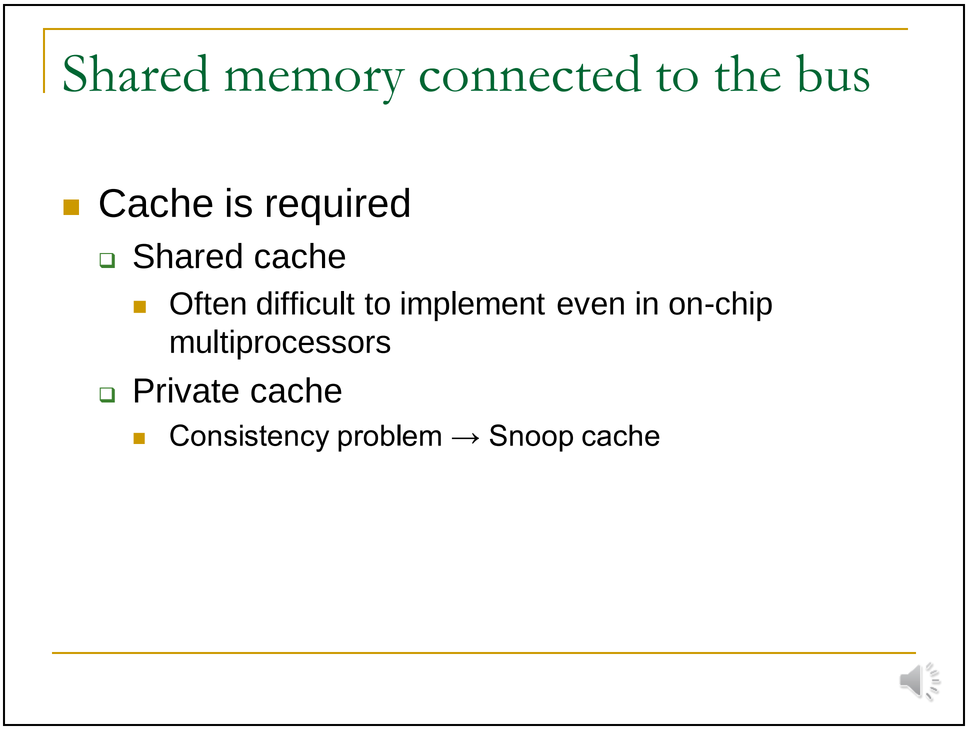
Clean

0000	0

Cache
(64B=8 blocks)

Cache Directory
(Tag Memory)
8 entries X (4bit+1bit)

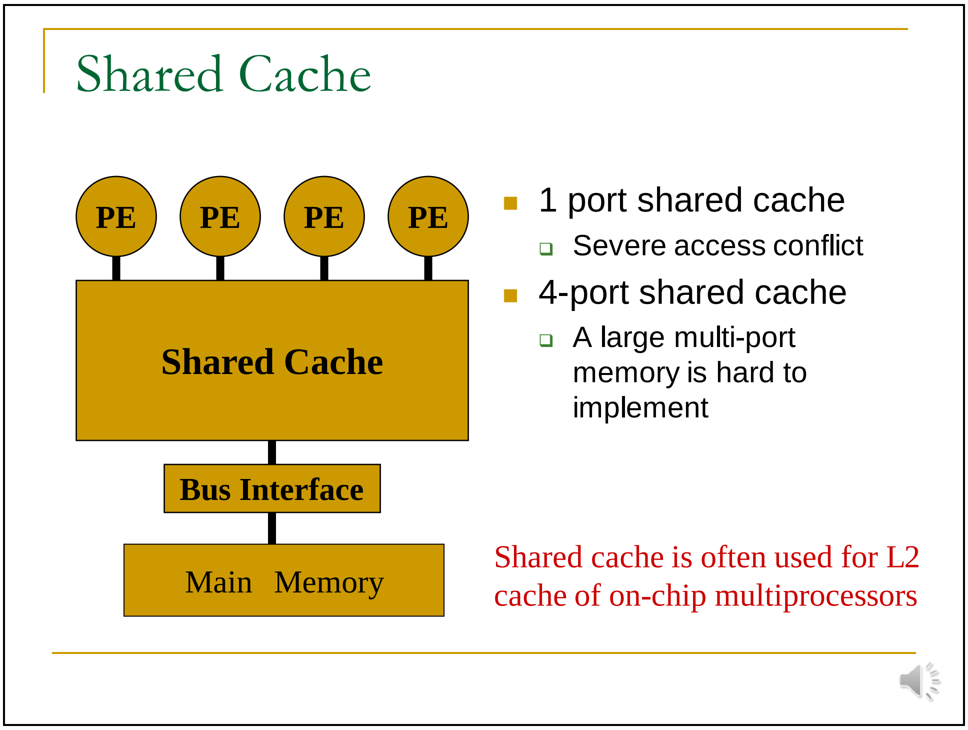
After the write-back, the new cache is filled, at that time the dirty bit of the cache entry is reset to show the cache is clean, that is the content of the cache block is the same as the main memory.



Shared memory connected to the bus

- Cache is required
  - Shared cache
    - Often difficult to implement even in on-chip multiprocessors
  - Private cache
    - Consistency problem → Snoop cache

OK. Now, let me explain about the cache for multi-core system.



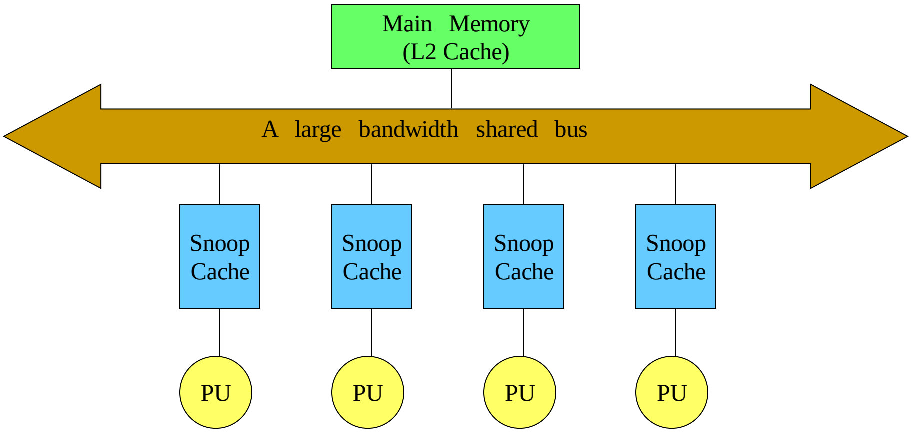
Shared Cache

- 1 port shared cache
  - Severe access conflict
- 4-port shared cache
  - A large multi-port memory is hard to implement

Shared cache is often used for L2 cache of on-chip multiprocessors

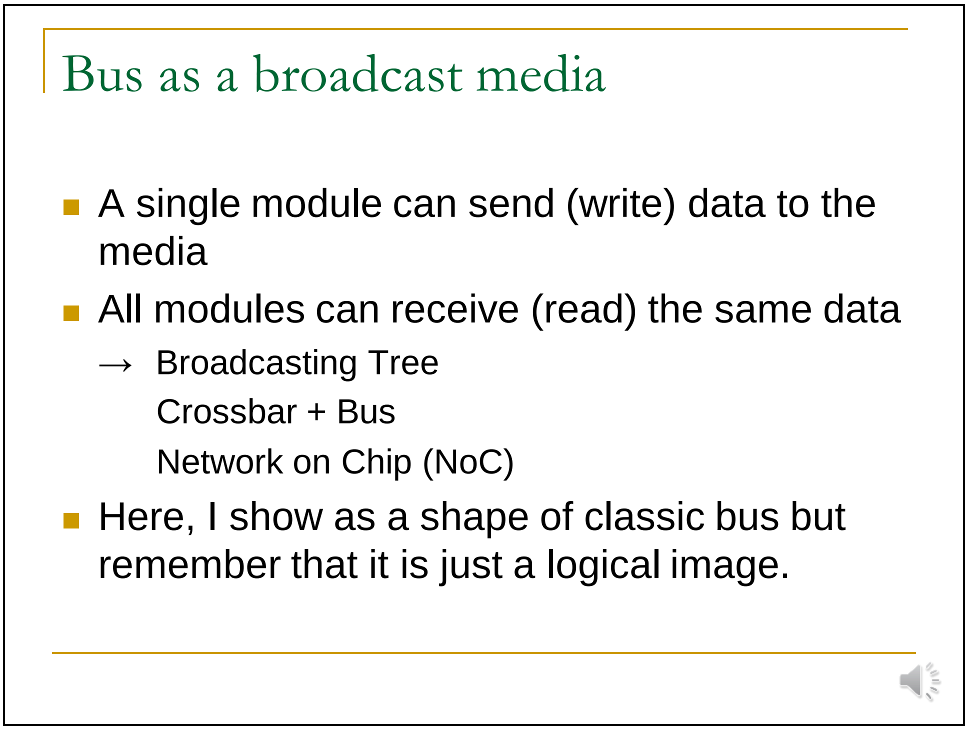
A simple idea is to share the cache with multi cores. Of course, it causes the severe access conflict at the cache. If multi-port memory is used, the conflict is reduced. However, a large multi-port memory is hard to be implemented, and the multi-port function is only available for read requests. Thus, shared cache is used for a L2 cache.

Private（Snoop） Cache



Each PU provides its own private cache

Another idea is private cache. Each processing unit or core has its own cache. It has a benefit of high speed data access and also reducing the shared bus. However, it causes the cache coherence problem or cache consistency problem.



Bus as a broadcast media

- A single module can send (write) data to the media
- All modules can receive (read) the same data

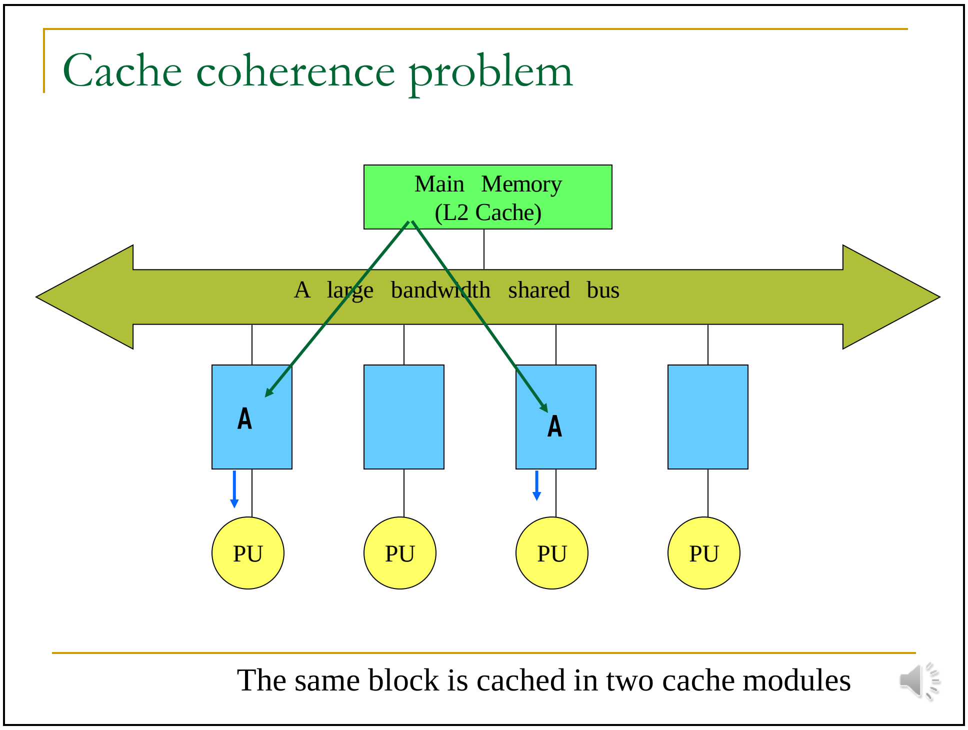
→ Broadcasting Tree

Crossbar + Bus

Network on Chip (NoC)

- Here, I show as a shape of classic bus but remember that it is just a logical image.

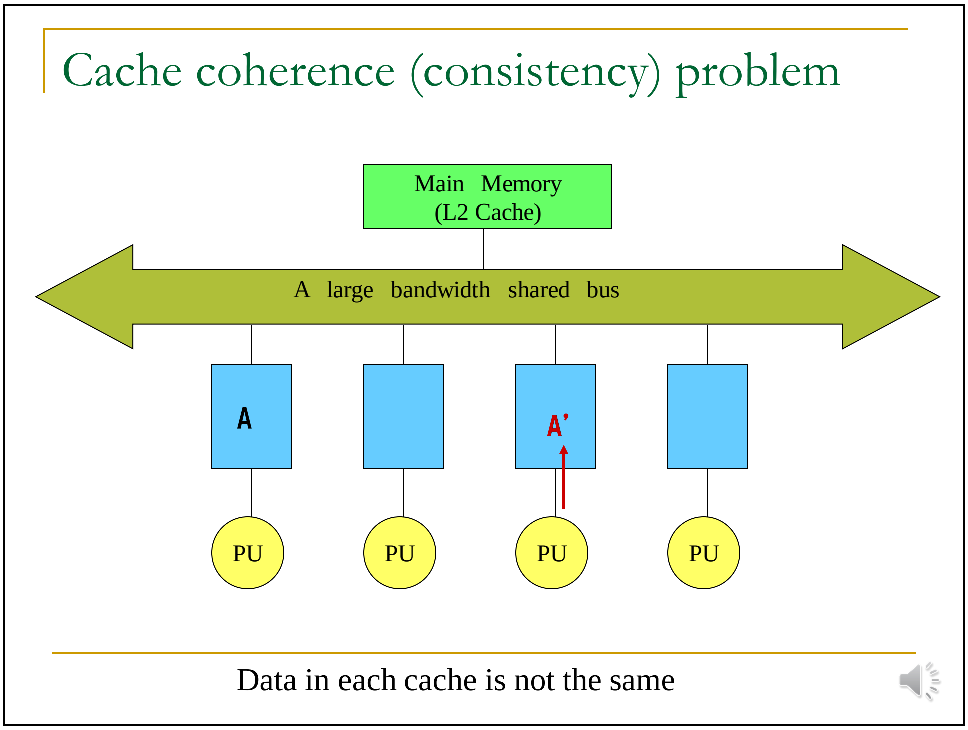
Here, I will show a bus as a simple wire images. However, note that it is just a logical image. It can be actually implemented as logic gates inside the chip.



Cache coherence problem

The same block is cached in two cache modules

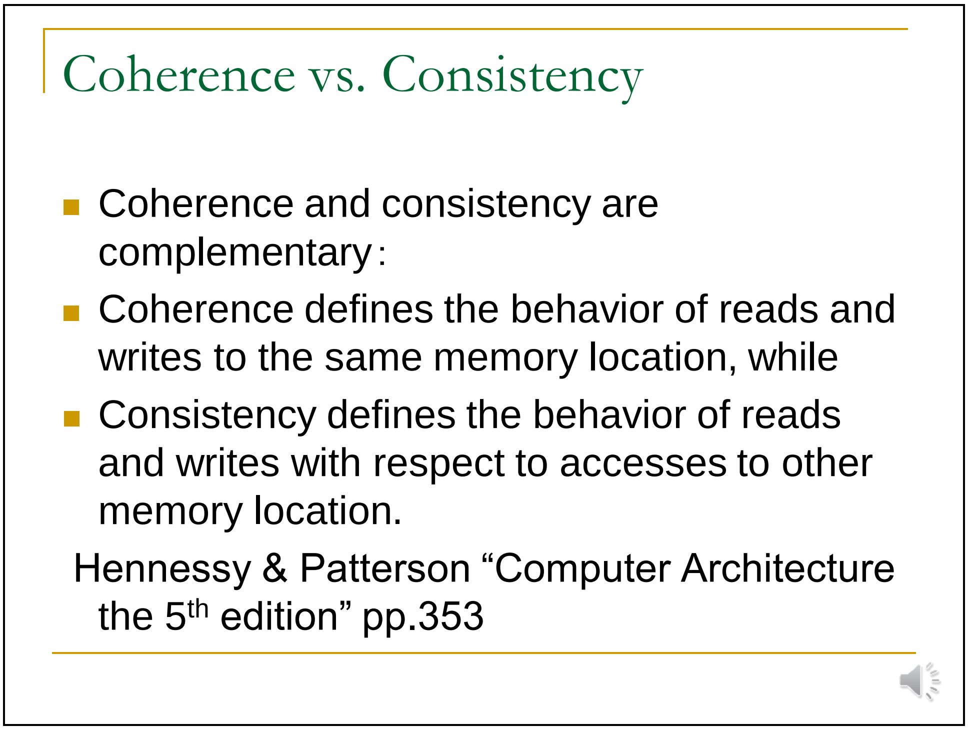
When a PU reads the block, the main memory or L2 cache is accessed and the block A is read out. Another PU can copy the same block A in its own cache. If only read operations are executed, there is no problem.



Cache coherence (consistency) problem

Data in each cache is not the same

However, what happens a PU writes data, the block is updated, but another PU who has the same copy cannot know of that. As a result, the content of two caches becomes different or inconsistent.

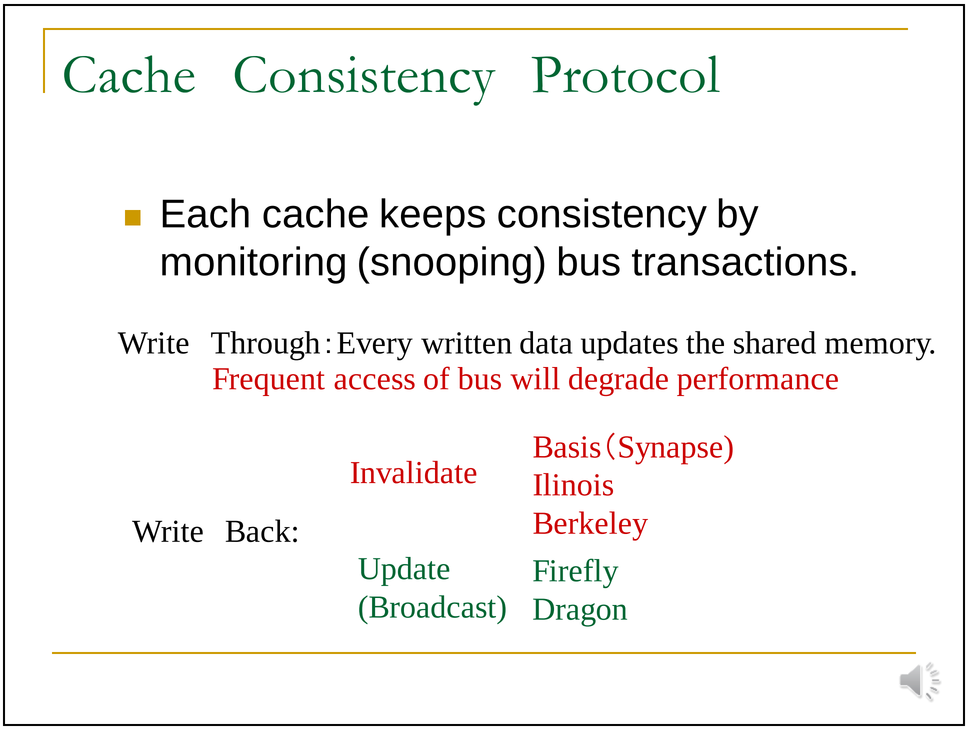


Coherence vs. Consistency

- Coherence and consistency are complementary:
- Coherence defines the behavior of reads and writes to the same memory location, while
- Consistency defines the behavior of reads and writes with respect to accesses to other memory location.

Hennessy & Patterson “Computer Architecture the 5th edition” pp.353

The words coherence and consistency are complement. That is coherence defines the behavior or reads and writes to the same memory location, while consistency is for other memory location. Since the cache block may include both, I think both are OK to use.



Cache Consistency Protocol

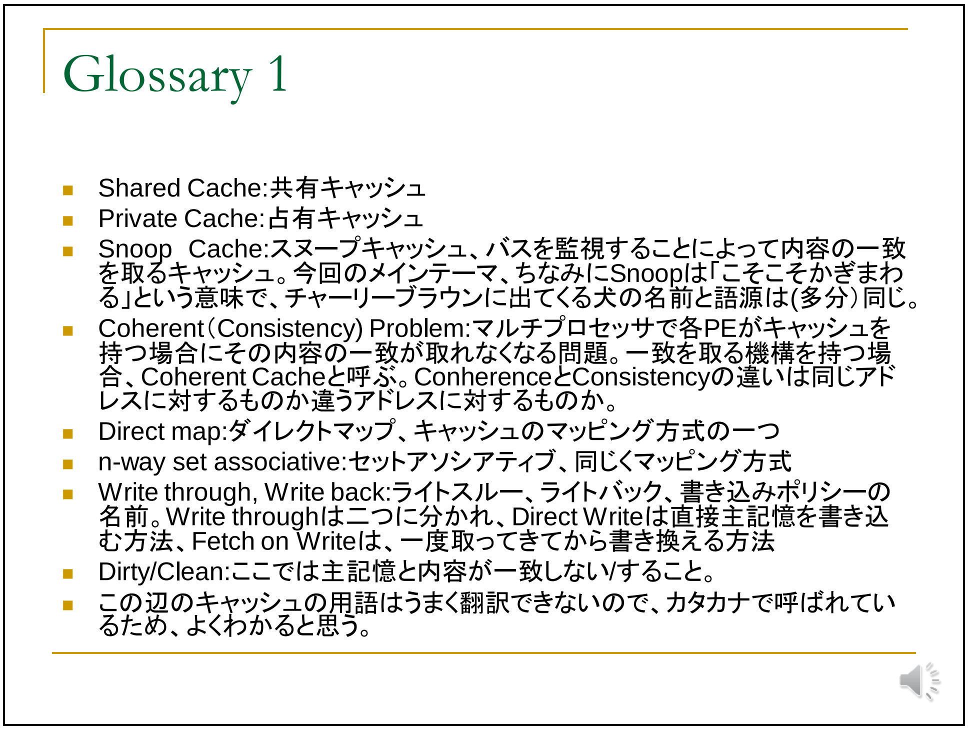
- Each cache keeps consistency by monitoring (snooping) bus transactions.

Write Through：Every written data updates the shared memory.
Frequent access of bus will degrade performance

Write Back:

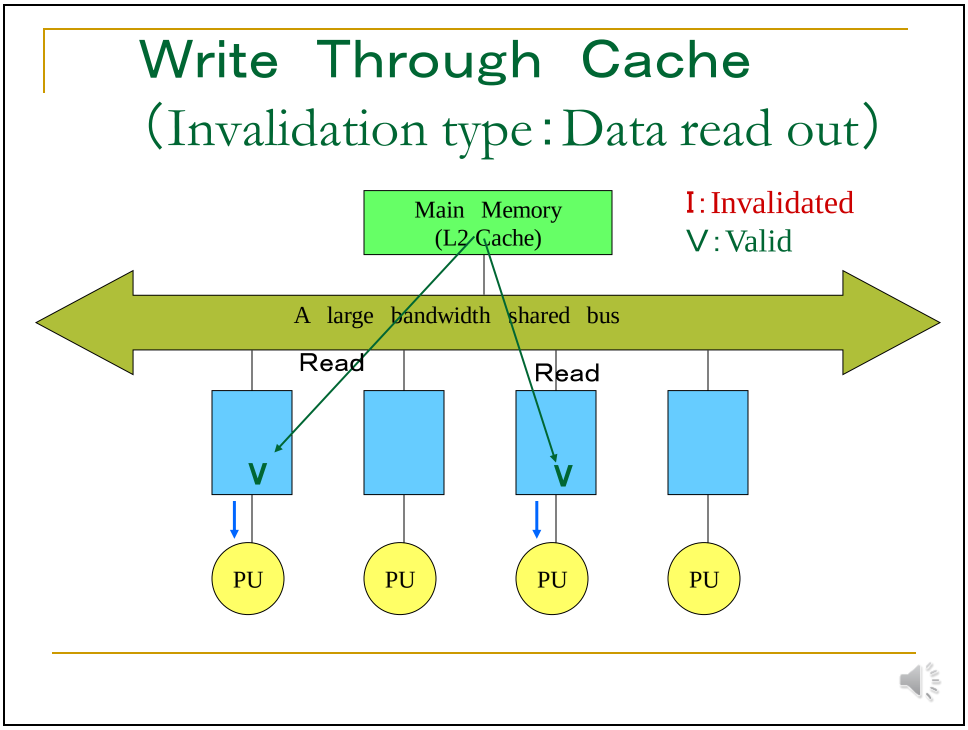
Write Back:	Invalidate	Basis（Synapse) Ilinois Berkeley
	Update (Broadcast)	Firefly Dragon

For keep consistency a certain protocol is needed. There are various types of protocols. I will introduce representative ones.



Glossary 1

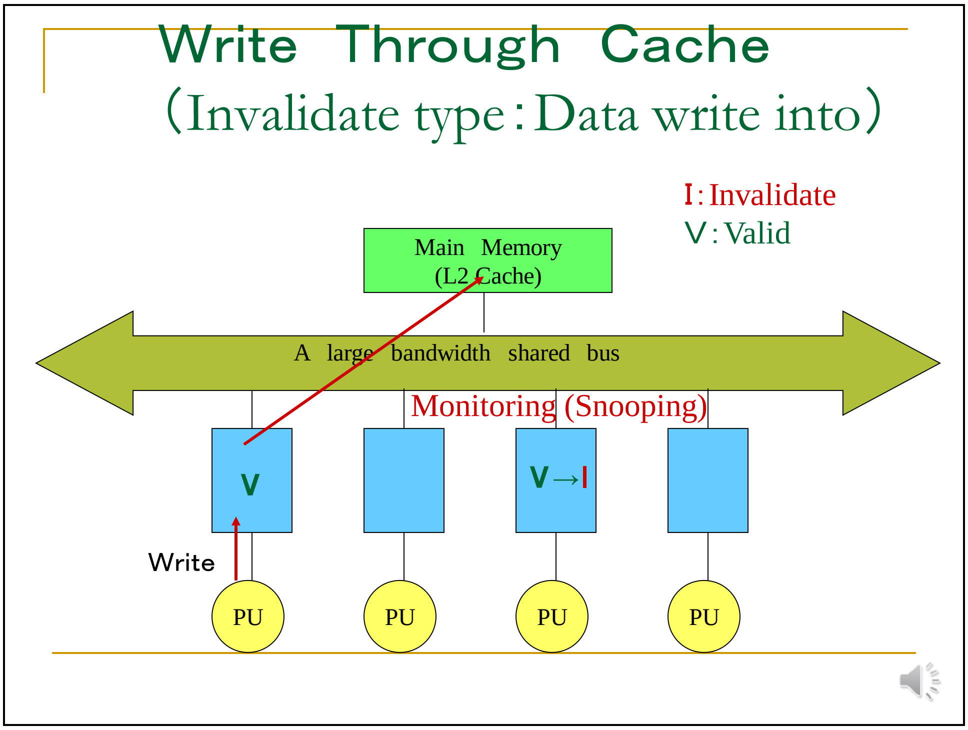
- Shared Cache:共有キャッシュ
- Private Cache:占有キャッシュ
- Snoop Cache:スヌープキャッシュ、バスを監視することによって内容の一致を取るキャッシュ。今回のメインテーマ、ちなみにSnoopは「こそこそかぎまわる」という意味で、チャーリーブラウンに出てくる犬の名前と語源は(多分)同じ。
- Coherent（Consistency) Problem:マルチプロセッサで各PEがキャッシュを持つ場合にその内容の一致が取れなくなる問題。一致を取る機構を持つ場合、Coherent Cacheと呼ぶ。ConherenceとConsistencyの違いは同じアドレスに対するものか違うアドレスに対するものか。
- Direct map:ダイレクトマップ、キャッシュのマッピング方式の一つ
- n-way set associative:セットアソシアティブ、同じくマッピング方式
- Write through, Write back:ライトスルー、ライトバック、書き込みポリシーの名前。Write throughは二つに分かれ、Direct Writeは直接主記憶を書き込む方法、Fetch on Writeは、一度取ってきてから書き換える方法
- Dirty/Clean:ここでは主記憶と内容が一致しない/すること。
- この辺のキャッシュの用語はうまく翻訳できないので、カタカナで呼ばれているため、よくわかると思う。



Write Through Cache
（Invalidation type：Data read out）

I：Invalidated
V：Valid

The idea of the snoop cache came from the write through cache. I am going to explain the invalidation type first. Assume that two PUs read the cache block, two copies are made.



Write Through Cache (Invalidate type：Data write into)

Ｉ: Invalidate
Ｖ: Valid

Main Memory
(L2 Cache)

A large bandwidth shared bus

Monitoring (Snooping)

V

V→I

Write

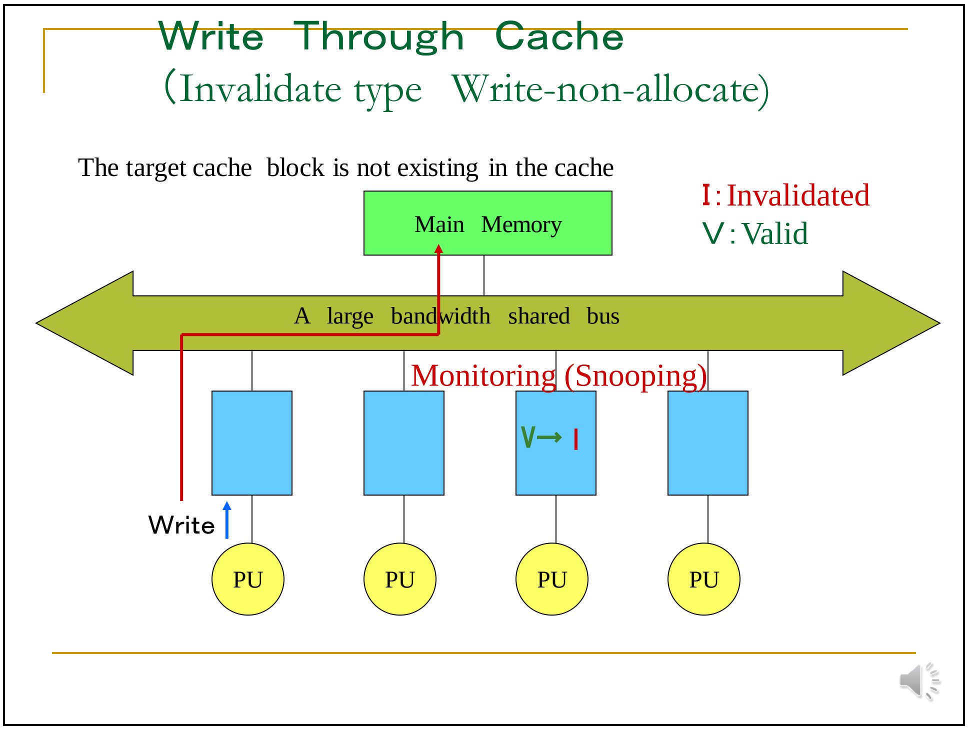
PU

PU

PU

PU

When a PU writes the data, it is transferred to the main memory since it is a write through cache. At that time, the address and the information which shows the request is write are transferred through the bus. Since the information on the bus can be monitored by other cache, all cache modules check its address and whether it is matched any block in the cache. If it matches, the cache turns its block to invalidate or I. For this purpose, like the dirty bit, all entries need the flag which shows the valid or not. This monitoring operation is called snoop, since the cache monitors everything secretly.



Write Through Cache
(Invalidate type Write-non-allocate)

The target cache block is not existing in the cache

I : Invalidated
V : Valid

Main Memory

A large bandwidth shared bus

Monitoring (Snooping)

V→ I

Write

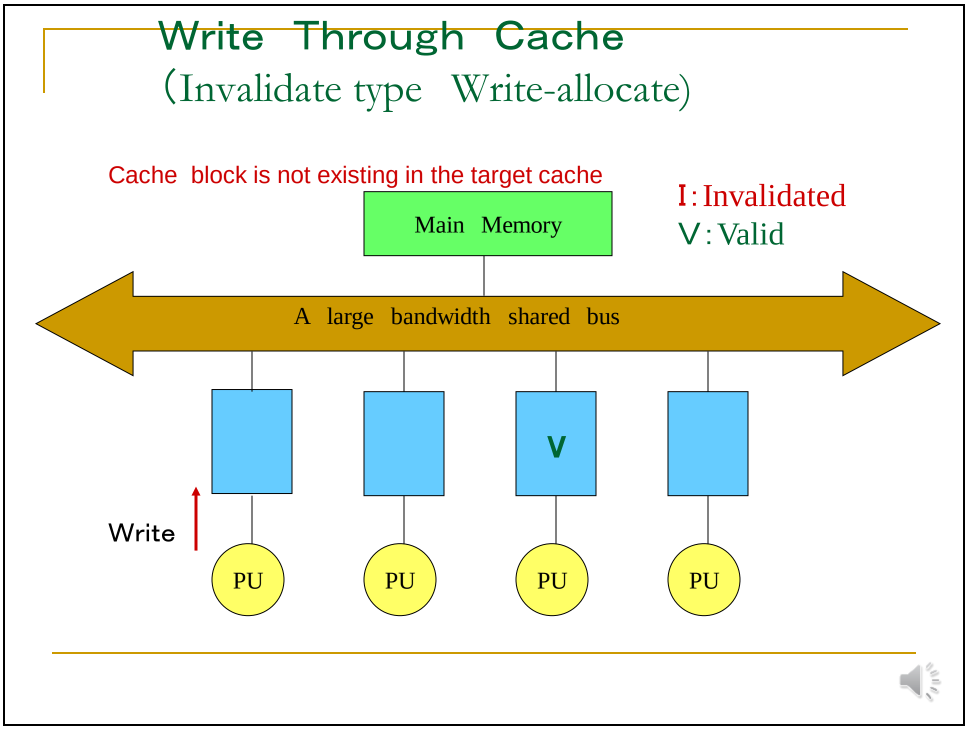
PU

PU

PU

PU

When write miss happens, the data are sent to the main memory directly in the write-non-allocate type cache. But each cache snoops it and invalidates the block similarly to the case of write hit.



Write Through Cache

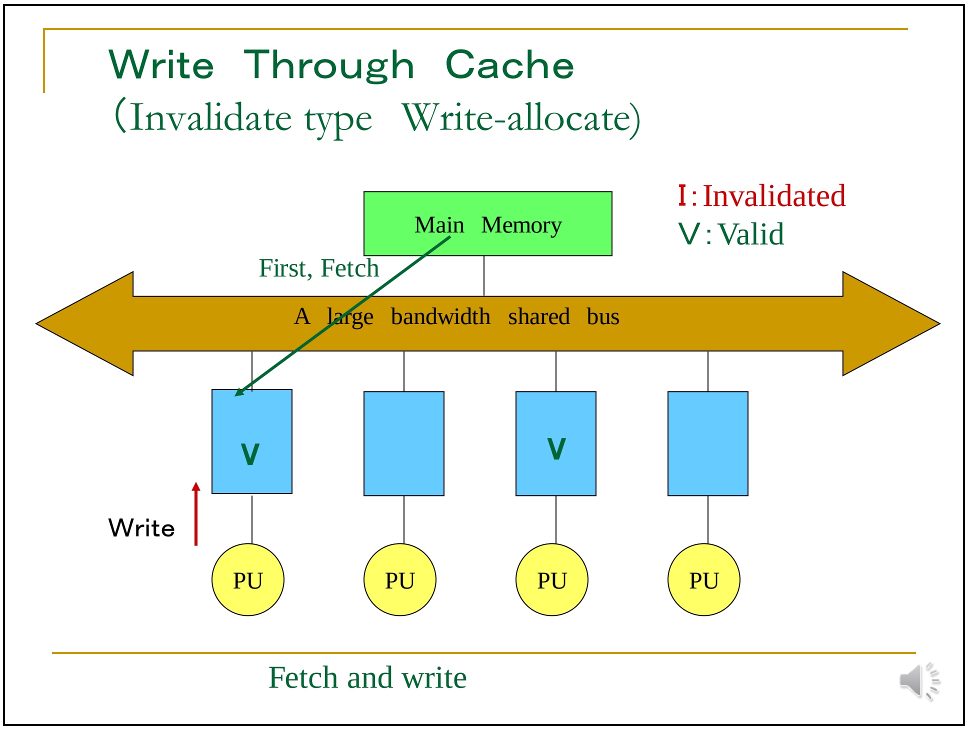
（Invalidate type Write-allocate)

Cache block is not existing in the target cache

I : Invalidated

V : Valid

Write allocate type cache can work in the same manner. If write miss happens.



Write Through Cache

(Invalidate type Write-allocate)

I : Invalidated

V : Valid

Main Memory

First, Fetch

A large bandwidth shared bus

V

V

Write

PU

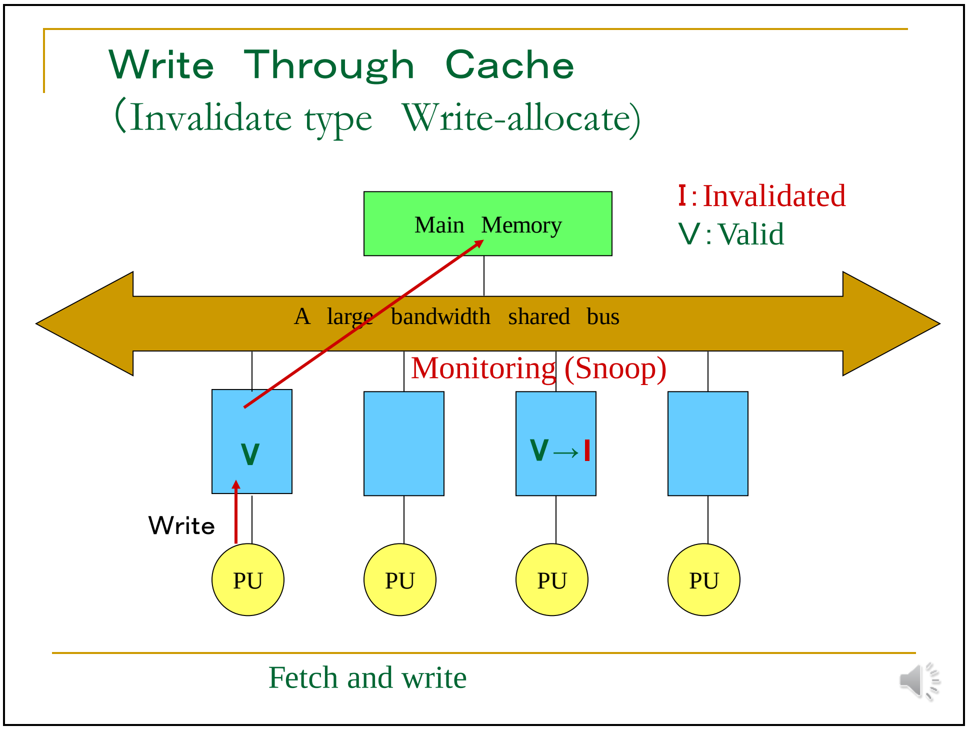
PU

PU

PU

Fetch and write

The block is fetched from the main memory, and the data are written on it.

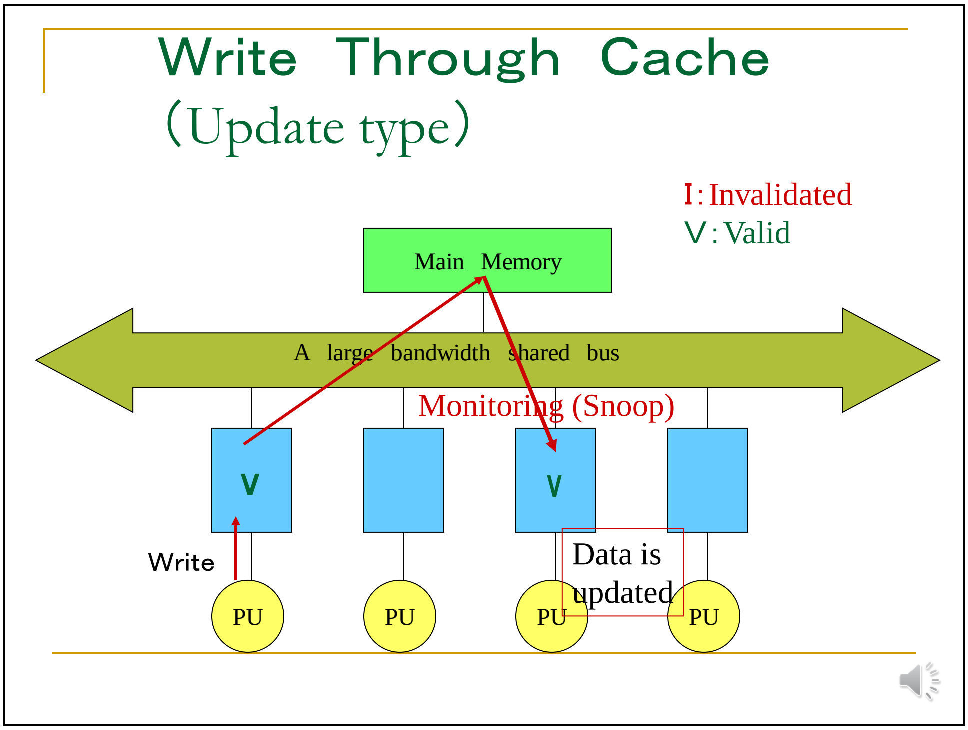


Write Through Cache

（Invalidate type Write-allocate)

Fetch and write

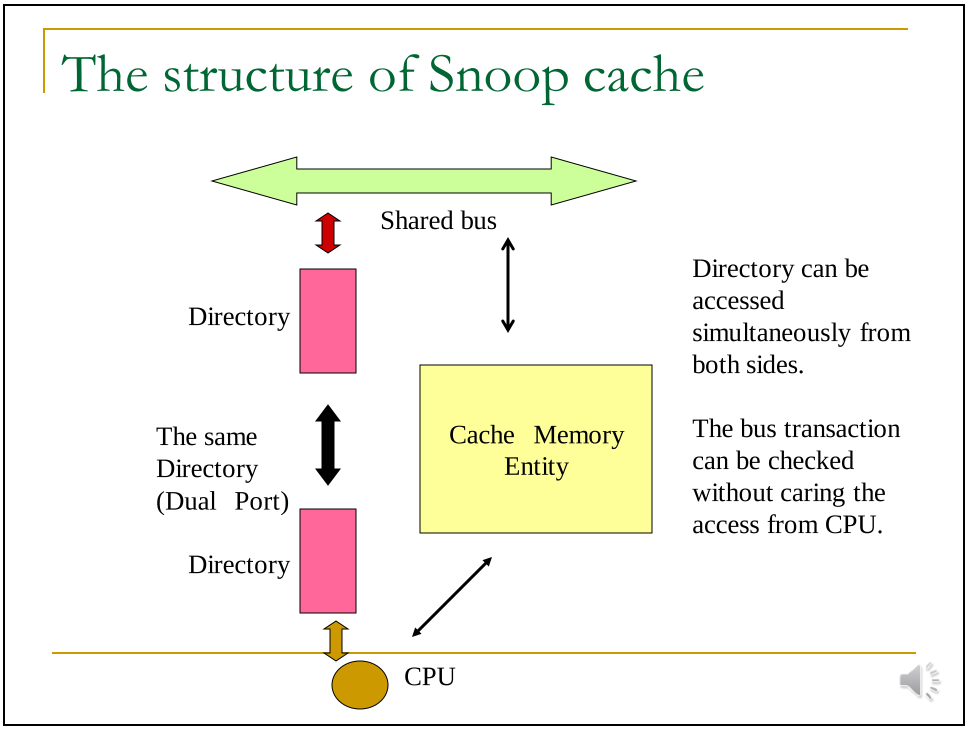
Other copies are invalidated by snooping the data.



Write Through Cache (Update type)

Ｉ:Invalidated
Ｖ:Valid

Instead of invalidating the block, the data can be written into the cache as well as main memory. The copies are updated, and the contents can be kept the same as the main memory. This case, the block was not invalidate. This concept is called the update style.



The structure of Snoop cache

Shared bus

Directory

The same Directory (Dual Port)

Directory

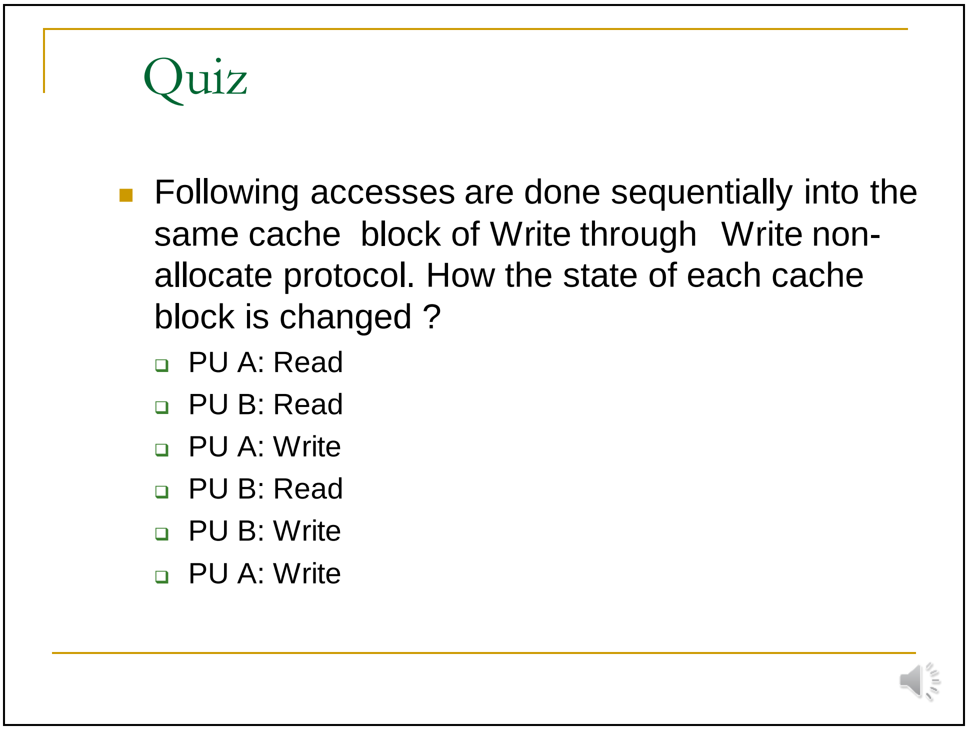
Cache Memory Entity

CPU

Directory can be accessed simultaneously from both sides.

The bus transaction can be checked without caring the access from CPU.

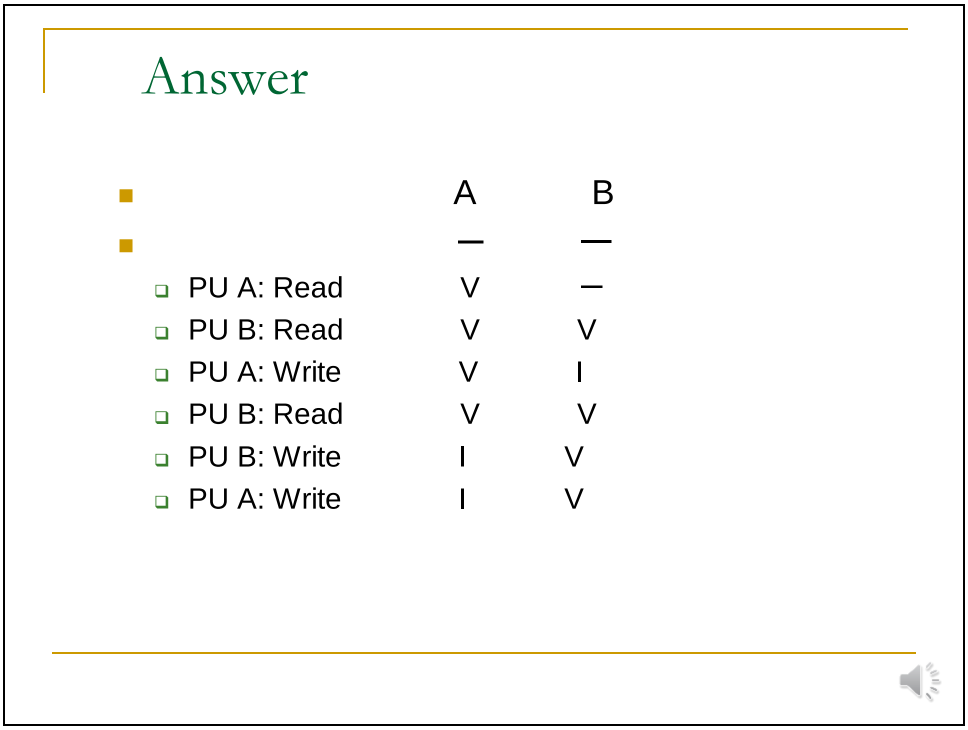
For snooping the shared bus, the directory is needed for shared bus as well as for the CPU. The contents must be the same. Dual port memory is sometimes used, but twin memory modules which synchronized only writing is easier.



Quiz

- Following accesses are done sequentially into the same cache block of Write through   Write non-allocate protocol. How the state of each cache block is changed ?
  - PU A: Read
  - PU B: Read
  - PU A: Write
  - PU B: Read
  - PU B: Write
  - PU A: Write

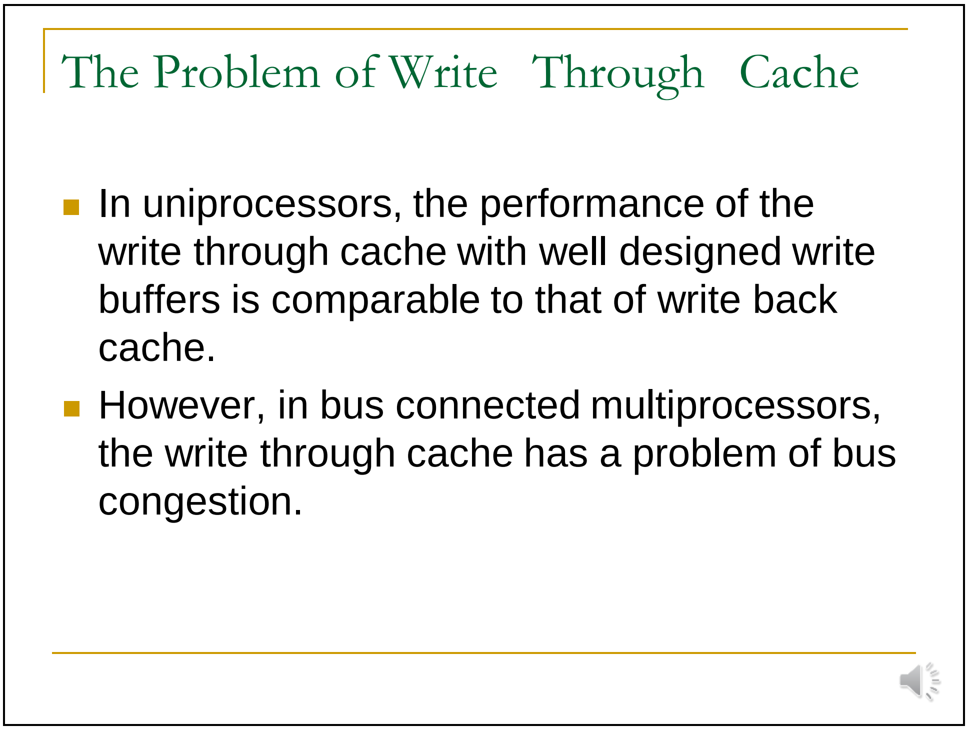
Let’s try a simple quiz.



Answer

	A	B
	—	—
PU A: Read	V	—
PU B: Read	V	V
PU A: Write	V	I
PU B: Read	V	V
PU B: Write	I	V
PU A: Write	I	V

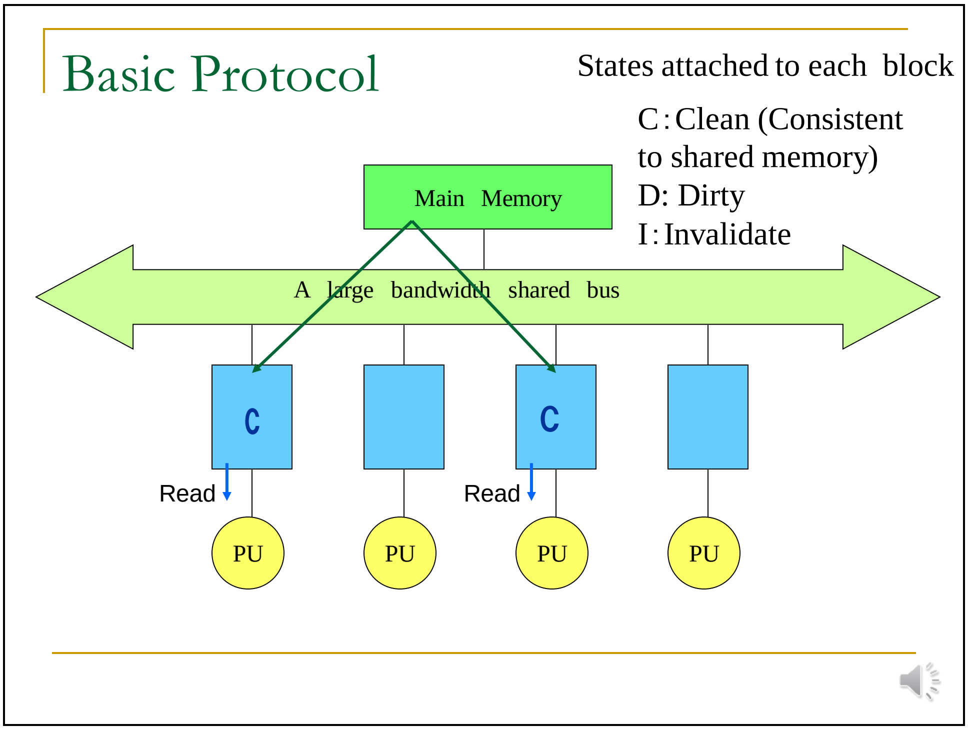
Let’s try a simple quiz.



The Problem of Write Through Cache

- In uniprocessors, the performance of the write through cache with well designed write buffers is comparable to that of write back cache.
- However, in bus connected multiprocessors, the write through cache has a problem of bus congestion.

The early snoop cache uses the write through cache and several commercial machines were successful. However, in bus connected multiprocessors, the write through cache has a problem of bus congestion, and the performance improvement of a CPU become great, the write through cache became unpractical.

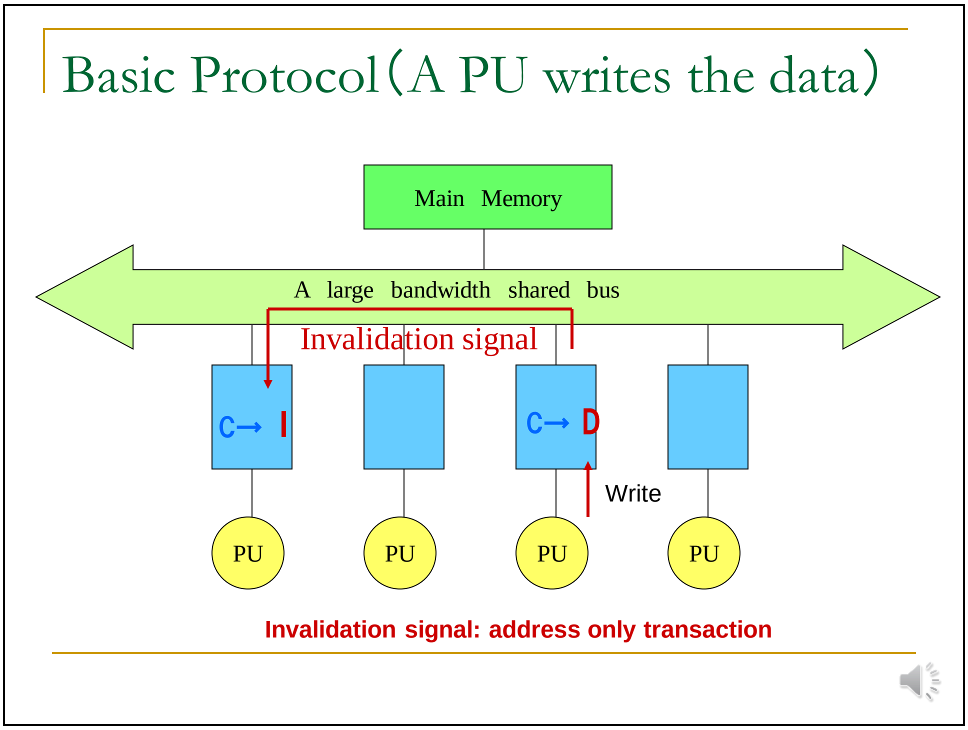


Basic Protocol

States attached to each block

C: Clean (Consistent to shared memory)
D: Dirty
I: Invalidate

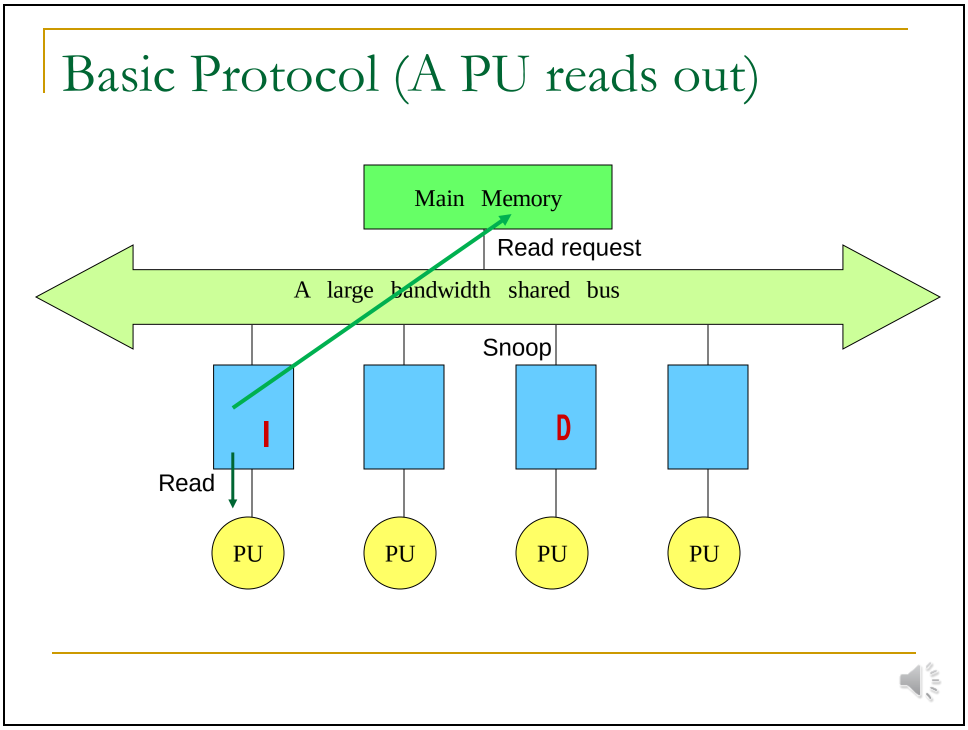
The implementation to the write back cache is relatively difficult. Here, I show the simplest protocol. Each directory entry has 2 bits; valid/invalid and clean/dirty. A block has three states Clean, Dirty or Invalidate. When two PUs read the data, these blocks become Clean.



Basic Protocol (A PU writes the data)

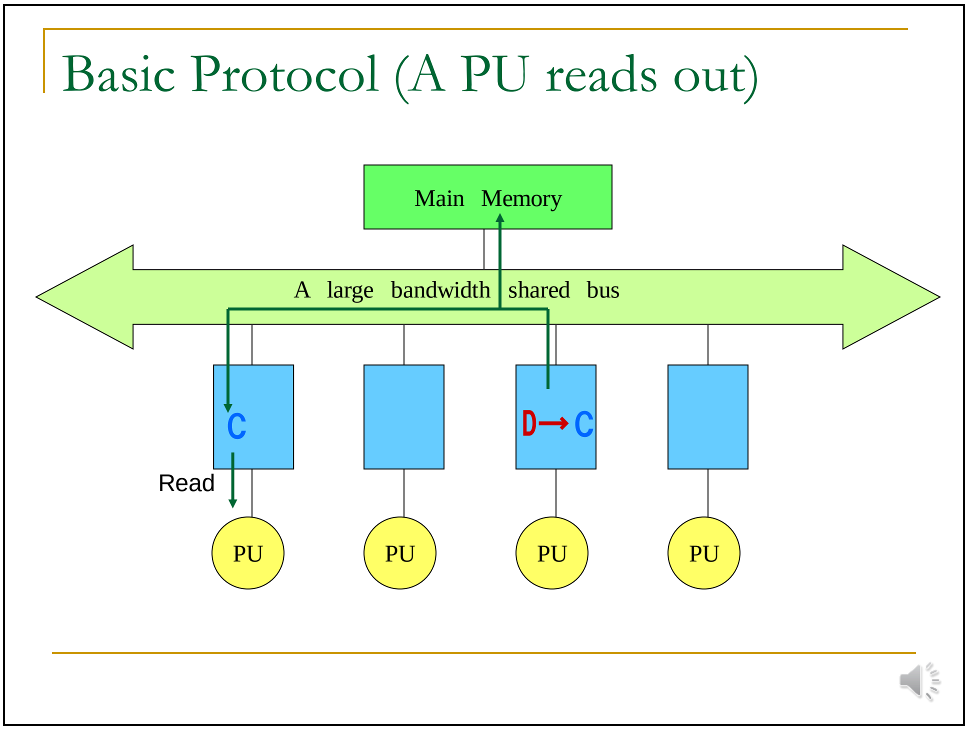
Invalidation signal: address only transaction

The problem is when a PU writes the data. As common write back cache, the state changes from Clean to Dirty. In order to notice other PUs, the invalidation signal is transferred on the shared bus. It includes only address and invalidation request. By snooping the address, each cache changes its state from Clean to Invalidated like this. Once the state becomes Dirty, of course, no transactions on the shared bus are needed. The block marked Dirty can be read and write freely as common write back cache does.



Basic Protocol (A PU reads out)

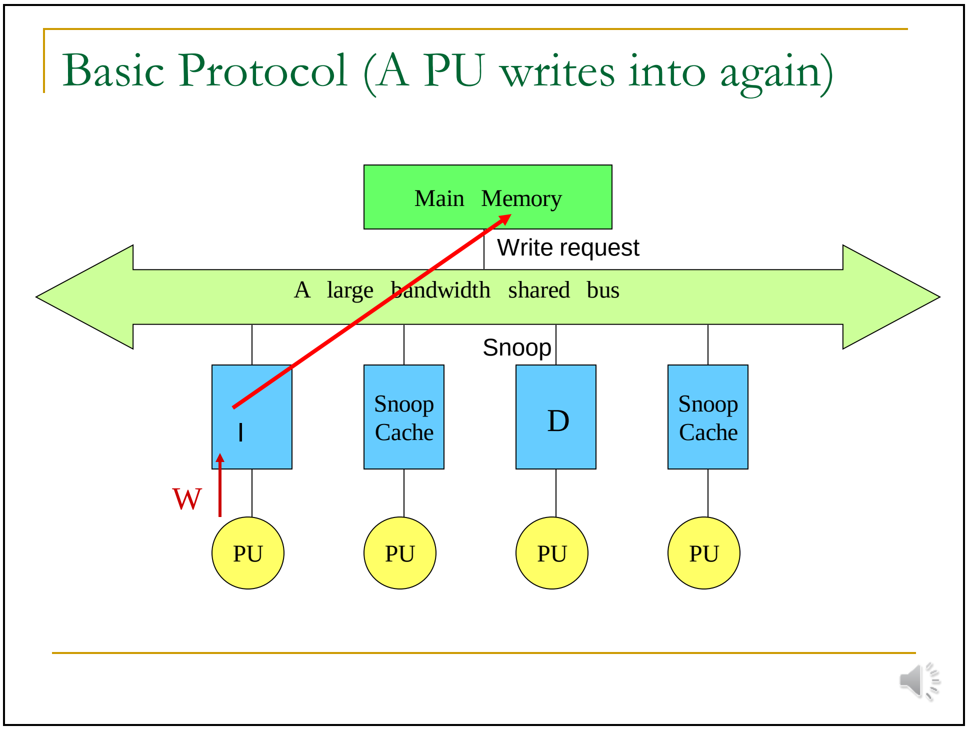
The problem happens when the cache marked dirty is accessed by the other PU. When this PU reads the data, since it is invalidated, a miss occurs. It sends the read request to the main memory, but there is a dirty cache.



Basic Protocol (A PU reads out)

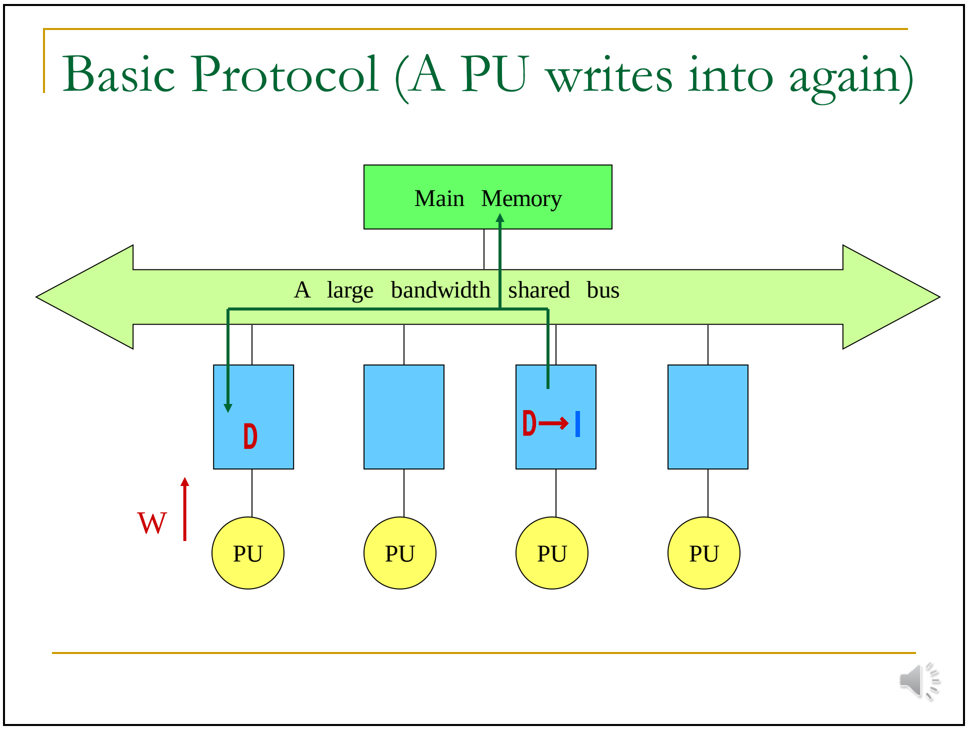
This cache snoops the shared bus and recognizes the read request. At that time, this cache stops main memory answering the request and

instead it, the cache sends the block on the bus and write back is done. After that, the block is transferred to the requesting cache. The write back and filling requesting cache can be done in the multi-casting manner if the bus protocol allows. After this operation, both cache blocks become Clean.



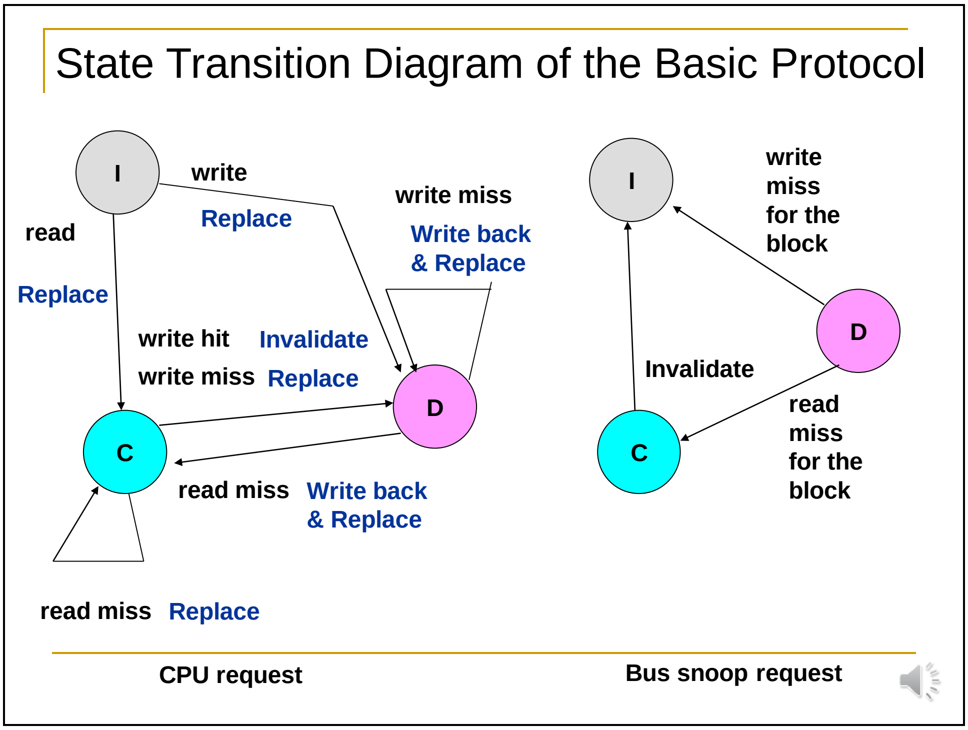
Basic Protocol (A PU writes into again)

What happens the PU causes the write miss. As the case of read miss, the request goes to the main memory same, since the write back cache uses the write allocate policy.



Basic Protocol (A PU writes into again)

The cache providing the Dirty block responds as well and writes the block back to the main memory, then it is forwarded to the requesting cache. After writing data into the block, it directly turns into Dirty state. On the other hand, after sending the block, the supplier's state becomes Invalidated directly.



State Transition Diagram of the Basic Protocol

I

write

Replace

read

Replace

write miss

Write back
& Replace

write hit Invalidate

write miss Replace

D

C

read miss Write back
& Replace

read miss Replace

CPU request

I

write
miss
for the
block

D

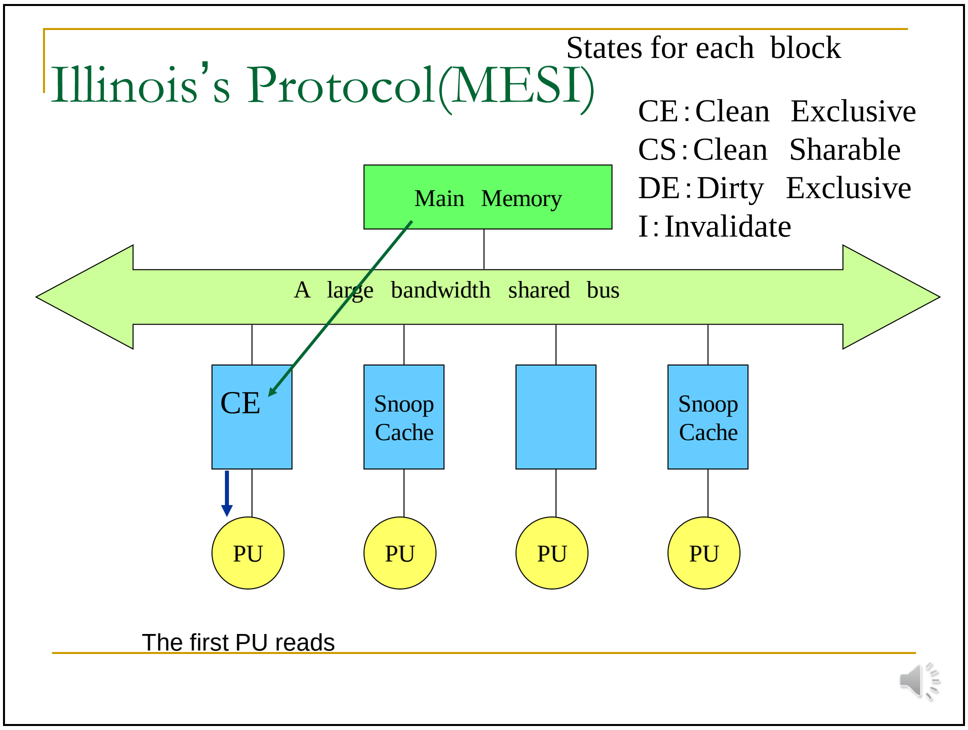
Invalidate

C

read
miss
for the
block

Bus snoop request

The cache protocol can be described by the state transition diagrams. Two diagrams must be provided. One is from CPU request and the other is from the shared bus.



Illinois's Protocol(MESI)

States for each block

CE : Clean Exclusive
CS : Clean Sharable
DE : Dirty Exclusive
I : Invalidate

Main Memory

A large bandwidth shared bus

CE

Snoop Cache

Snoop Cache

PU

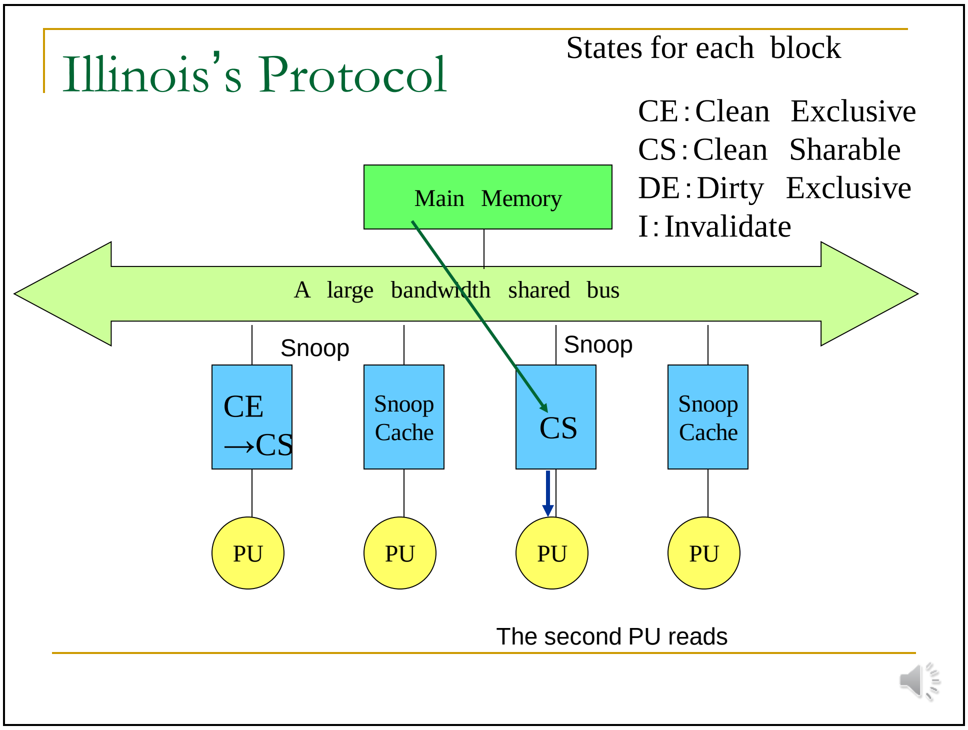
PU

PU

PU

The first PU reads

The basic protocol can be improved. One idea is providing exclusive state by adding an extra bit to each directory entry. It is set when there is no other copy in the system. Dirty cache is always exclusive. So, four states; clean exclusive, clean sharable, dirty exclusive and invalidated are used. When the first PU reads the block it becomes Clean Exclusive. It can be detected there is no notice from other caches snooping the shared bus.



Illinois's Protocol

States for each block

CE : Clean Exclusive
CS : Clean Sharable
DE : Dirty Exclusive
I : Invalidate

Main Memory

A large bandwidth shared bus

Snoop

Snoop

CE
→CS

Snoop
Cache

CS

Snoop
Cache

PU

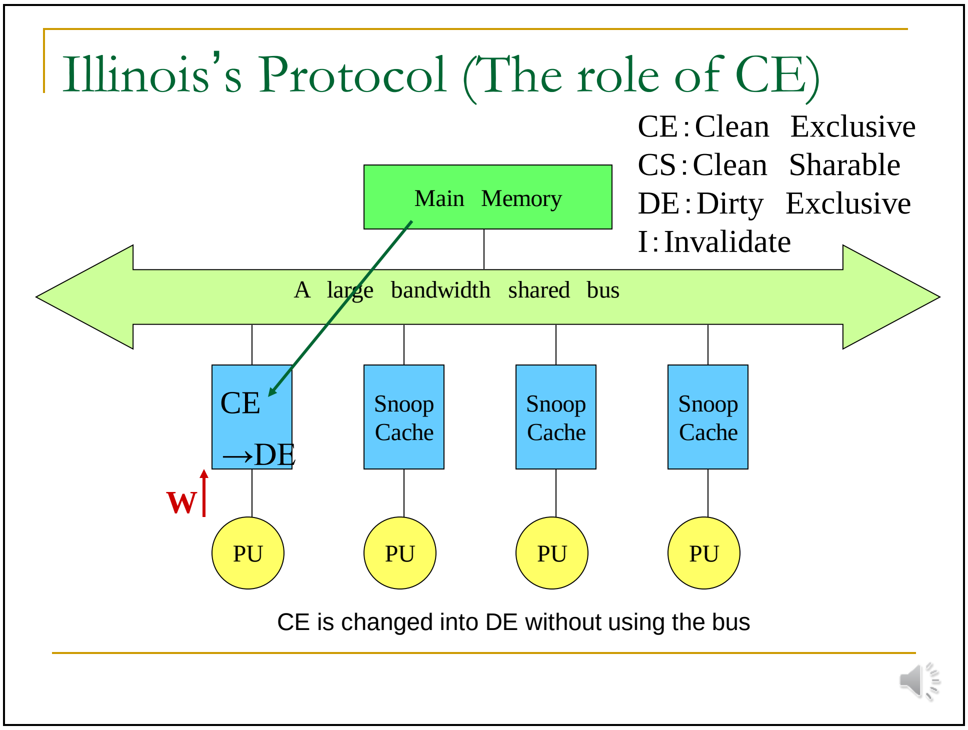
PU

PU

PU

The second PU reads

If other cache reads the same block, the cache with CE block responds, and both cache blocks turn to the Clean Sharable. In this case, the operation of this protocol is the same as the basic protocol.



Illinois's Protocol (The role of CE)

CE : Clean Exclusive
CS : Clean Sharable
DE : Dirty Exclusive
I : Invalidate

Main Memory

A large bandwidth shared bus

CE →DE

Snoop Cache

Snoop Cache

Snoop Cache

w

PU

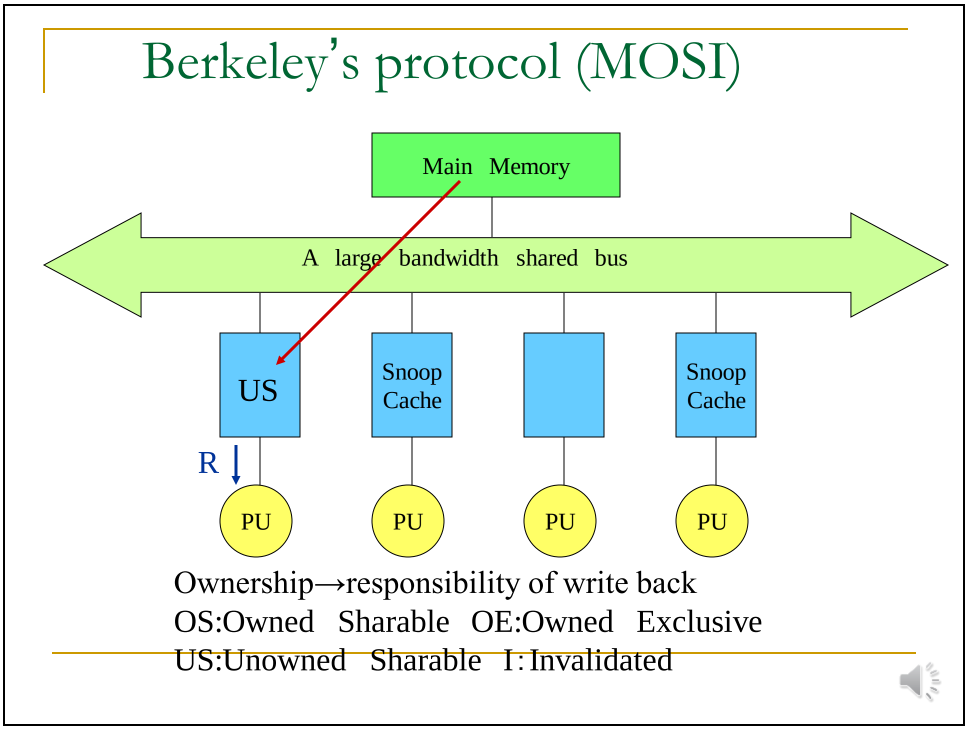
PU

PU

PU

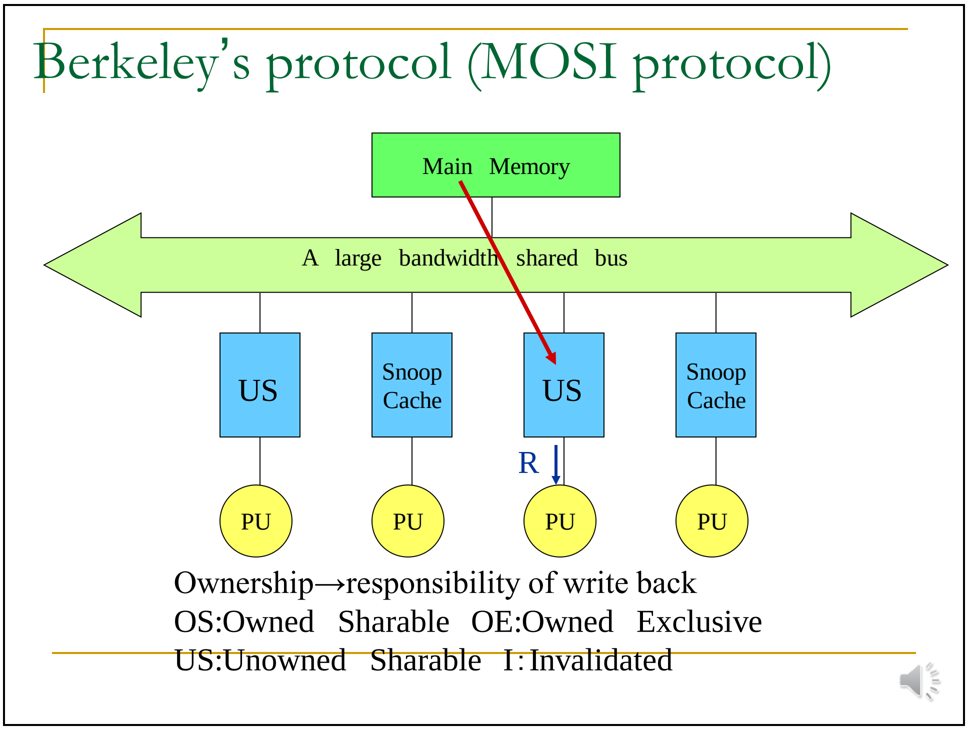
CE is changed into DE without using the bus

When the PU writes the data into the block with the CE, it changes its state into Dirty Exclusive without sending the invalidation signal on the bus. This is only the benefit of introducing the exclusive state. Some people think that the performance improvement by introducing the CE is not so large. However, because the shared data between PUs is actually not so large, most of invalidation signals in the basic protocol are in vain. So, this protocol called MESI or Illinois protocol is popularly used.



Berkeley's protocol (MOSI)

Ownership→responsibility of write back
OS:Owned Sharable OE:Owned Exclusive
US:Unowned Sharable I: Invalidated



Berkeley's protocol (MOSI protocol)

Main Memory

A large bandwidth shared bus

US

Snoop Cache

US

Snoop Cache

R

PU

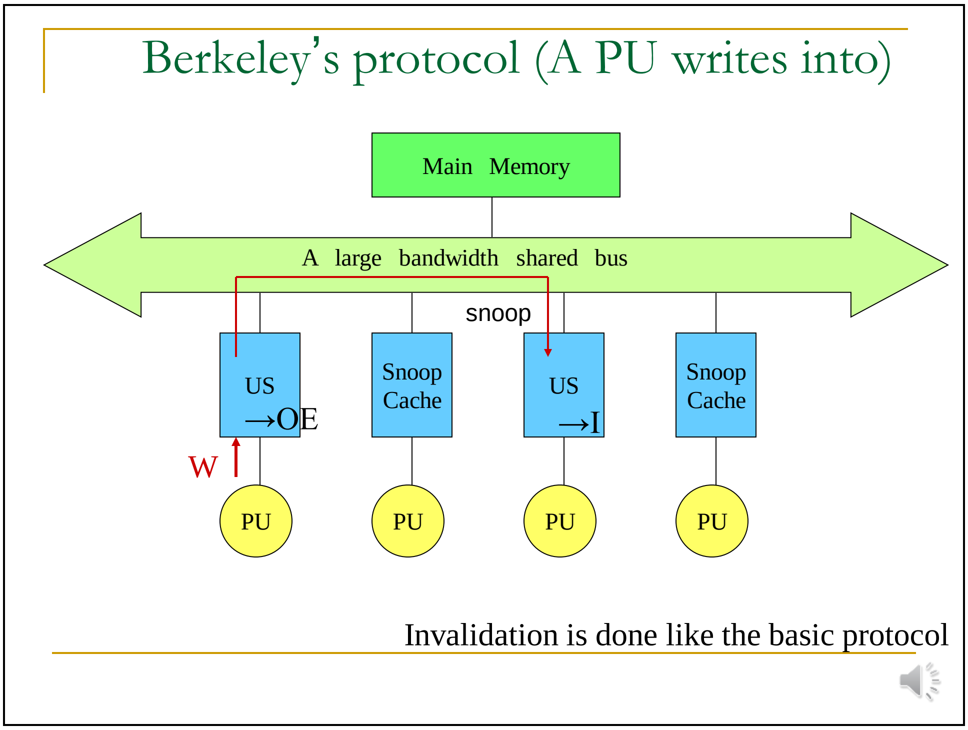
PU

PU

PU

Ownership→responsibility of write back
OS:Owned Sharable OE:Owned Exclusive
US:Unowned Sharable I: Invalidated

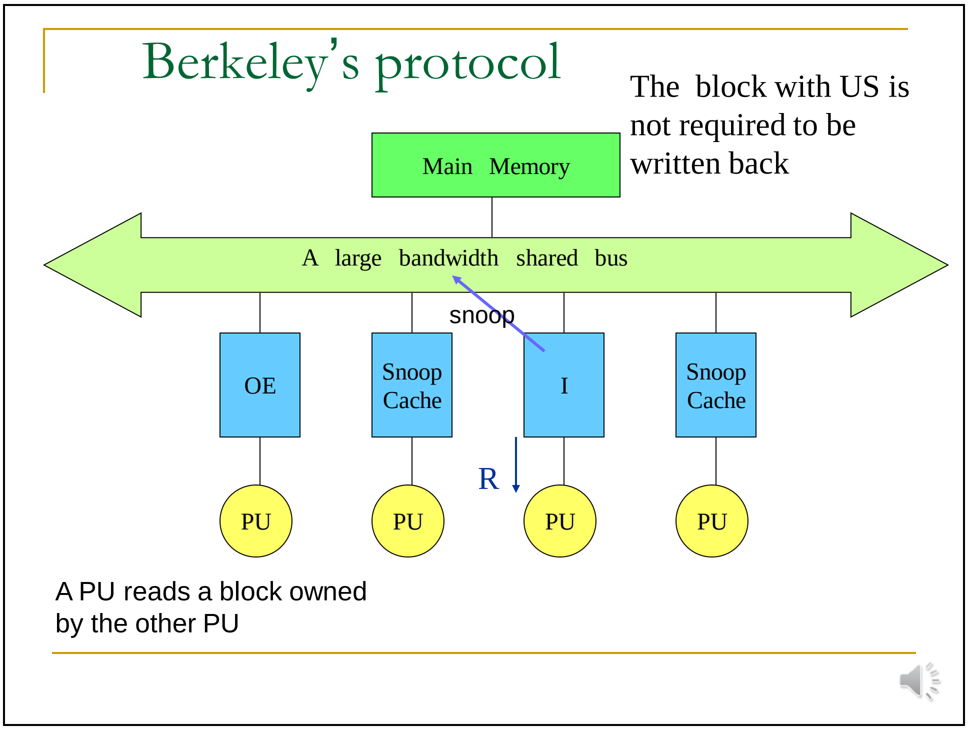
Another idea is introducing the concept of ownership. Dirty or clean is decided whether the content of block is the same as that of the main memory or not. But, by using the concept of ownership, the cache can behave instead of the main memory. Here, four states, owned sharable, owned exclusive, unowned sharable and invalidated. The default owner is the main memory, when a PU reads the data the copy becomes unowned sharable.



Berkeley's protocol (A PU writes into)

Invalidation is done like the basic protocol

When a PU writes the data, the block with unowned sharable sends the invalidation signals, and the all US blocks are invalidated. At that time, the block turns Owned Exclusive, that is, the PU becames the owner.

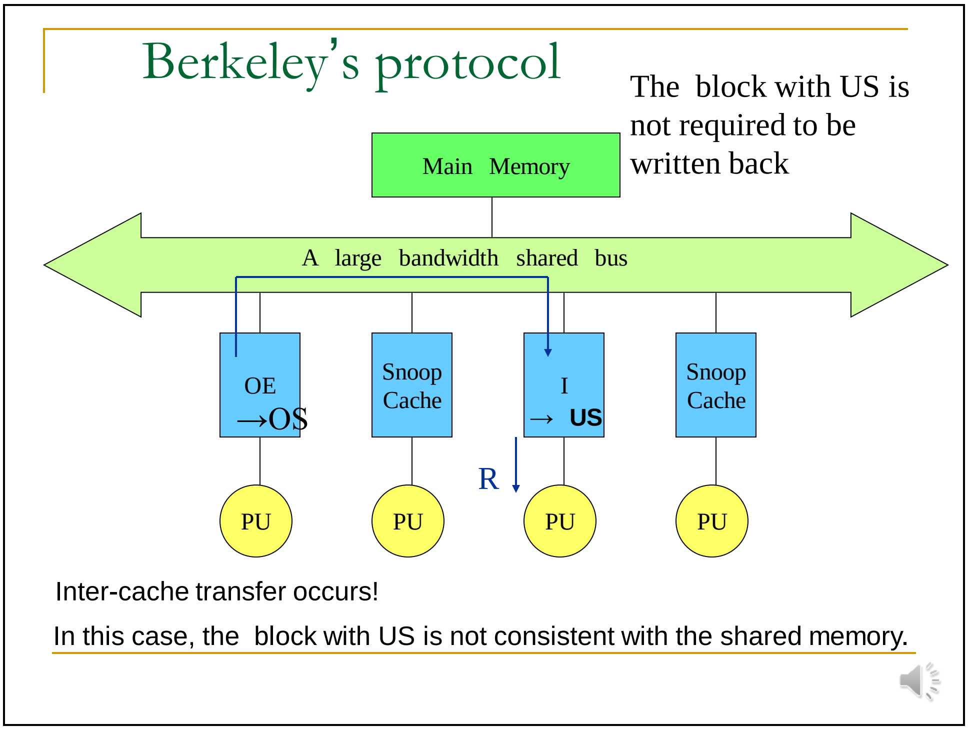


Berkeley's protocol

The block with US is not required to be written back

A PU reads a block owned by the other PU

When a PU occurs the read miss, the cache issues the request to the owner not for the main memory. This case, the owner, the cache with OE responds.



Berkeley's protocol

The block with US is not required to be written back

Main Memory

A large bandwidth shared bus

OE →OS

Snoop Cache

I → US

Snoop Cache

R

PU

PU

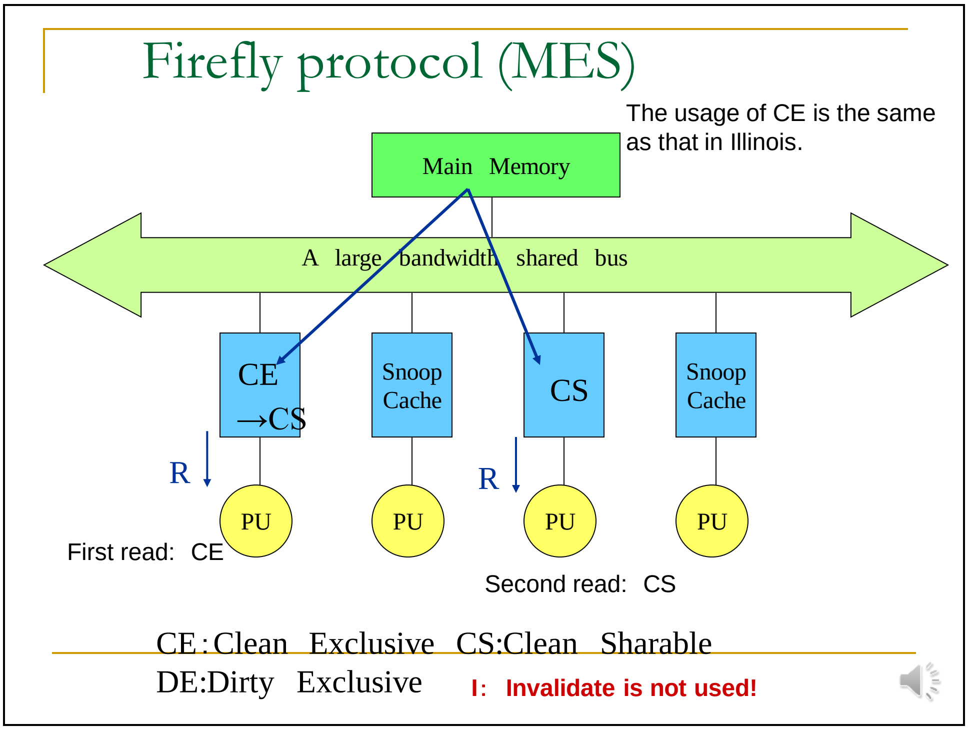
PU

PU

Inter-cache transfer occurs!

In this case, the block with US is not consistent with the shared memory.

In this case, the block is transferred to the requesting cache without write-back to the main memory. This method has two benefits: First, since the write back only occurs when the owner is replaced, redundant write back can be reduced. Second, the cache-to-cache data transfer can be done with much more speed than data transfer between the main memory or upper level cache. After the block transfer, the owner becomes owned sharable, and the requesting cache becomes US. Note that, this US block is consistent to the owner, not the main memory. This is called MOSI protocol or Berkeley protocol.



Firefly protocol (MES)

The usage of CE is the same as that in Illinois.

First read: CE

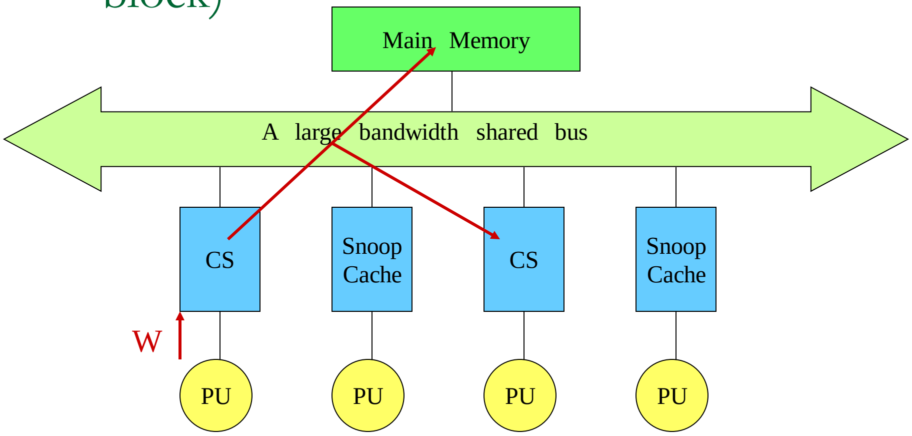
Second read: CS

CE: Clean Exclusive CS:Clean Sharable
DE:Dirty Exclusive

I: Invalidate is not used!

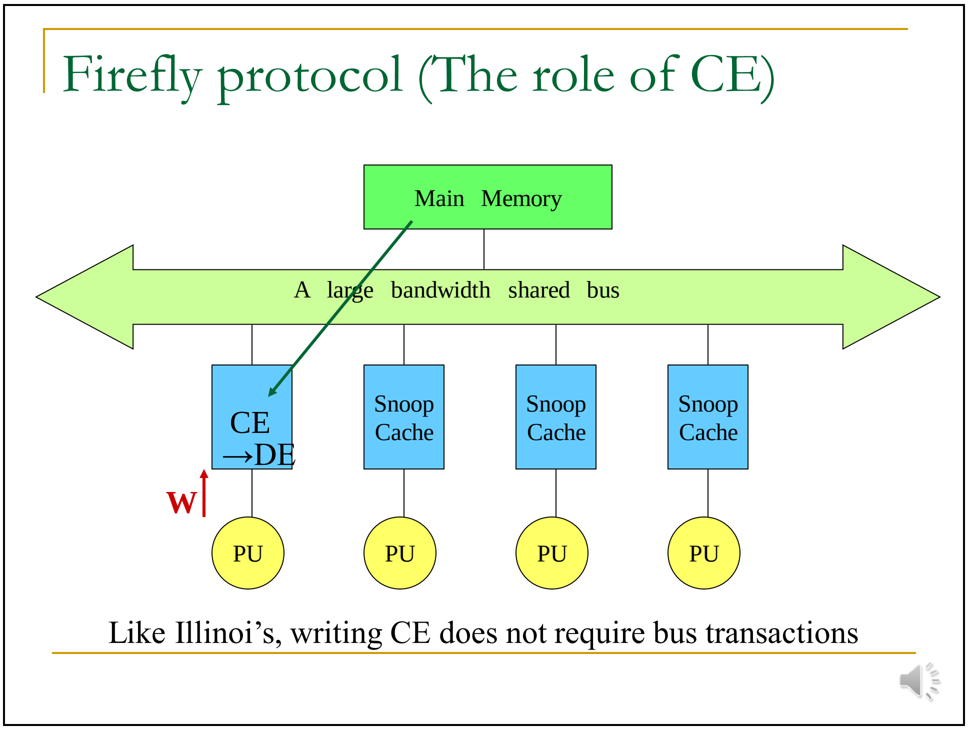
We can use update style protocol instead of the invalidation. This MES protocol only uses CE, CS, and DE. The protocol is almost the same as that of Illinois or MESI protocol. That is, the first reading cache block becomes CE but by the access from the next PUs, they all turn to CS.

Firefly protocol (Writes into the CS block)



All caches and shared memory are updated → Like update type Write Through Cache

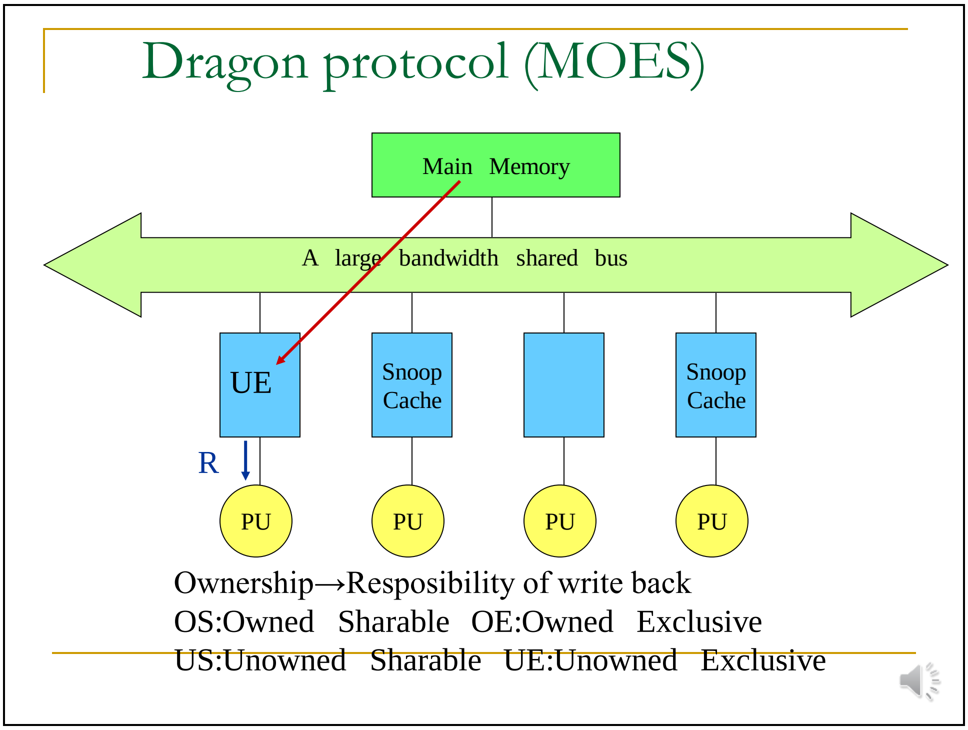
But the next step is quite different. When a PU writes the data, it is transferred both to main memory and other copies. Since the other cache block copy is updated, the state keeps CS. It means that once the state becomes CS, the writing data are always transferred through the shared bus.



Firefly protocol (The role of CE)

Like Illinoi's, writing CE does not require bus transactions

However, when a PU writes the data into CE block, it turns into DE without sending invalidation signal. And for DE blocks, the connecting CPU can read/write freely. It is called MES protocol or Firefly protocol named after DEC's workstation.

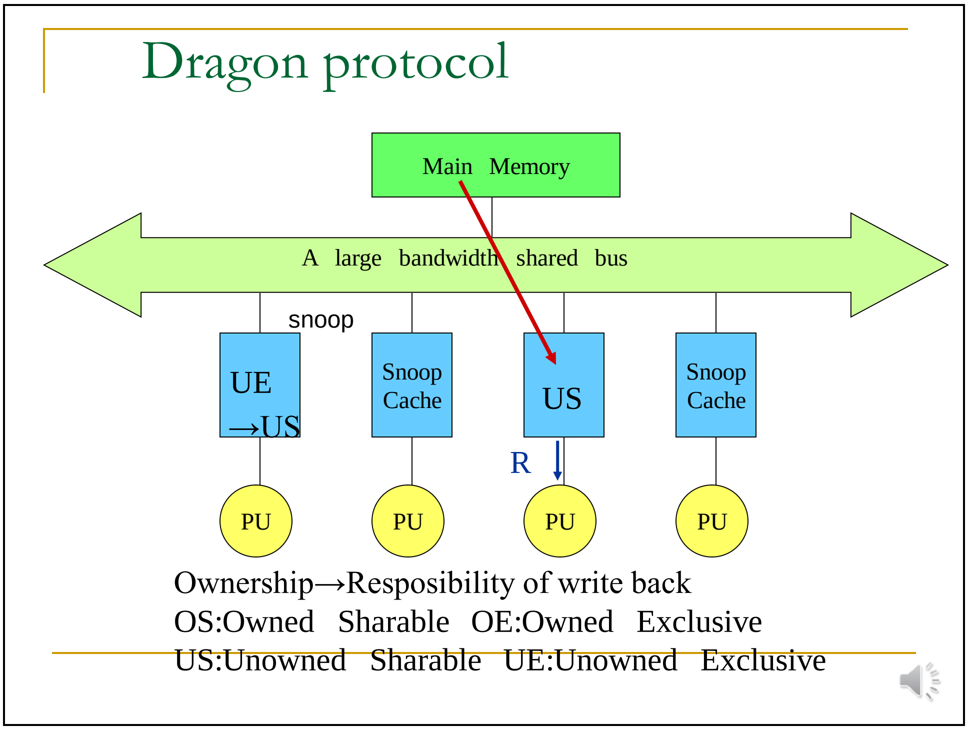


Dragon protocol (MOES)

Ownership→Resposibility of write back
OS:Owned Sharable OE:Owned Exclusive
US:Unowned Sharable UE:Unowned Exclusive

Another update style protocol uses both the concept of exclusive/shared and ownership. Thus, four states OS, OE, US, and UE are used.

When the first PU reads the data, the block state becomes UE.



Dragon protocol

Main Memory

A large bandwidth shared bus

snoop

UE →US

Snoop Cache

US

Snoop Cache

R

PU

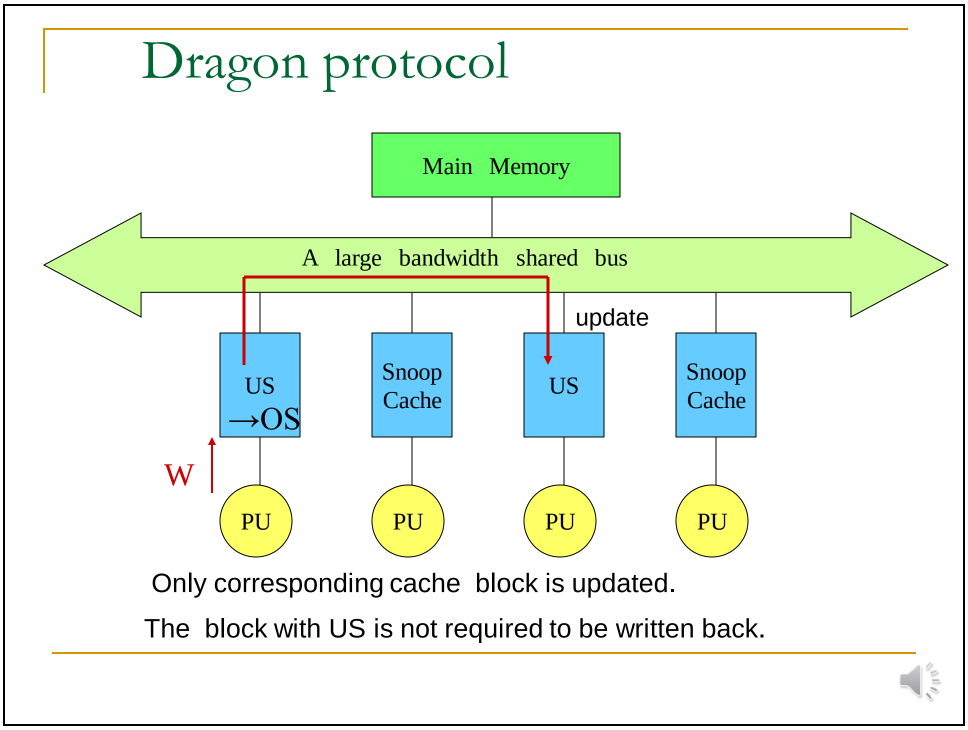
PU

PU

PU

Ownership→Resposibility of write back
OS:Owned Sharable OE:Owned Exclusive
US:Unowned Sharable UE:Unowned Exclusive

If the second PU reads the same cache block, both blocks become US.

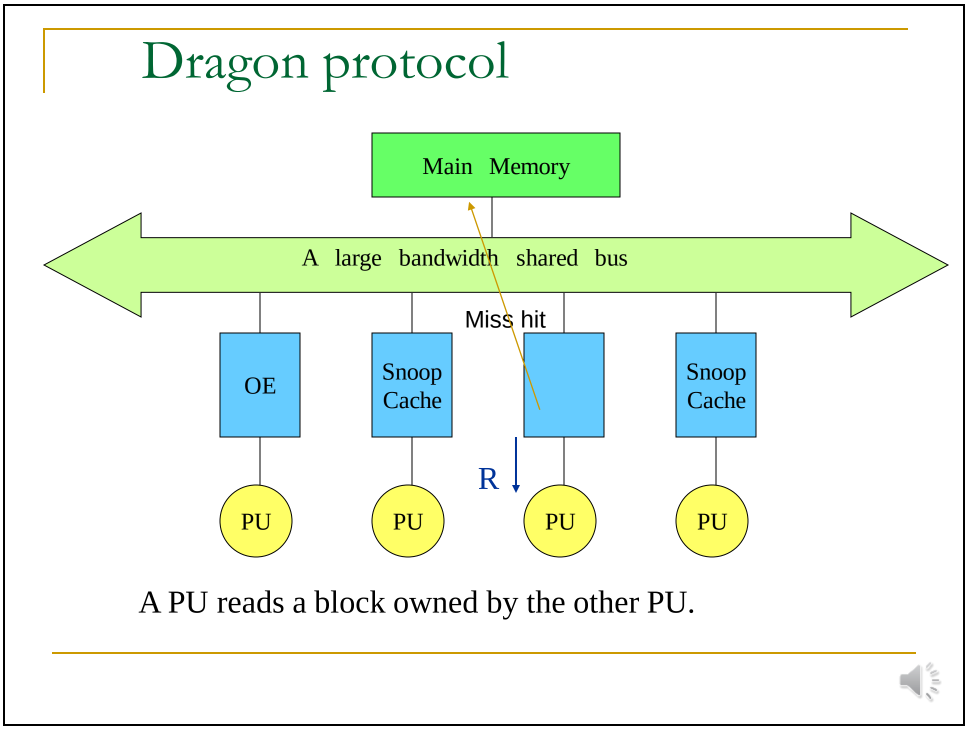


Dragon protocol

Only corresponding cache block is updated.

The block with US is not required to be written back.

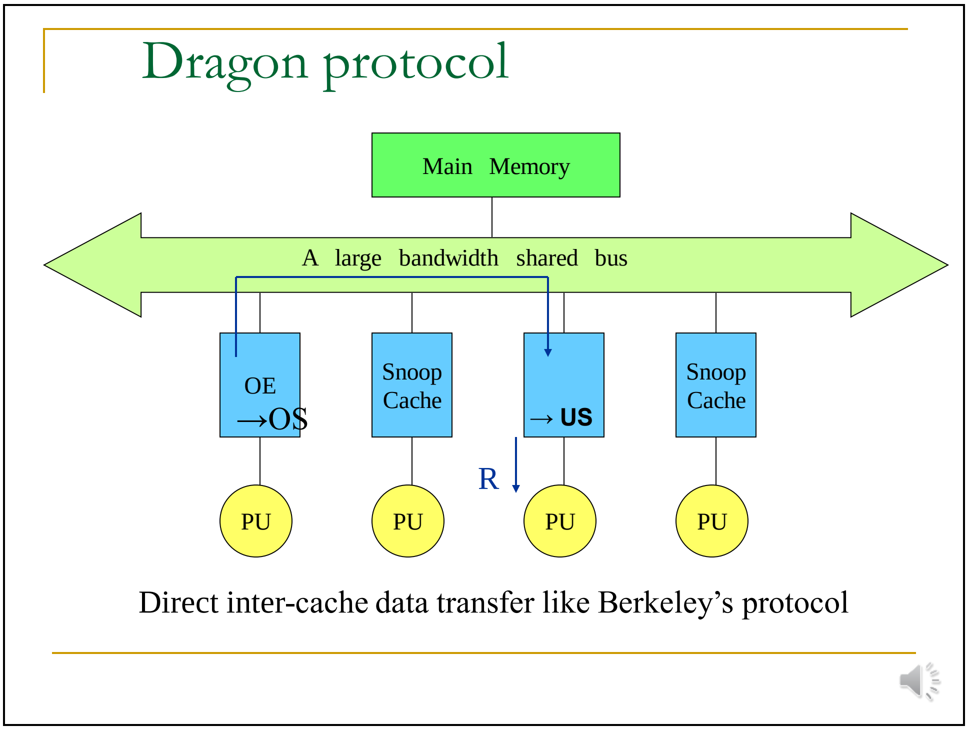
When a PU writes the block with US, the data are directly transferred through the shared bus and the cache copy is updated. Thus, although the owner state becomes OS, the state of copies stays in US. It is like the behavior of Firefly protocol.



Dragon protocol

A PU reads a block owned by the other PU.

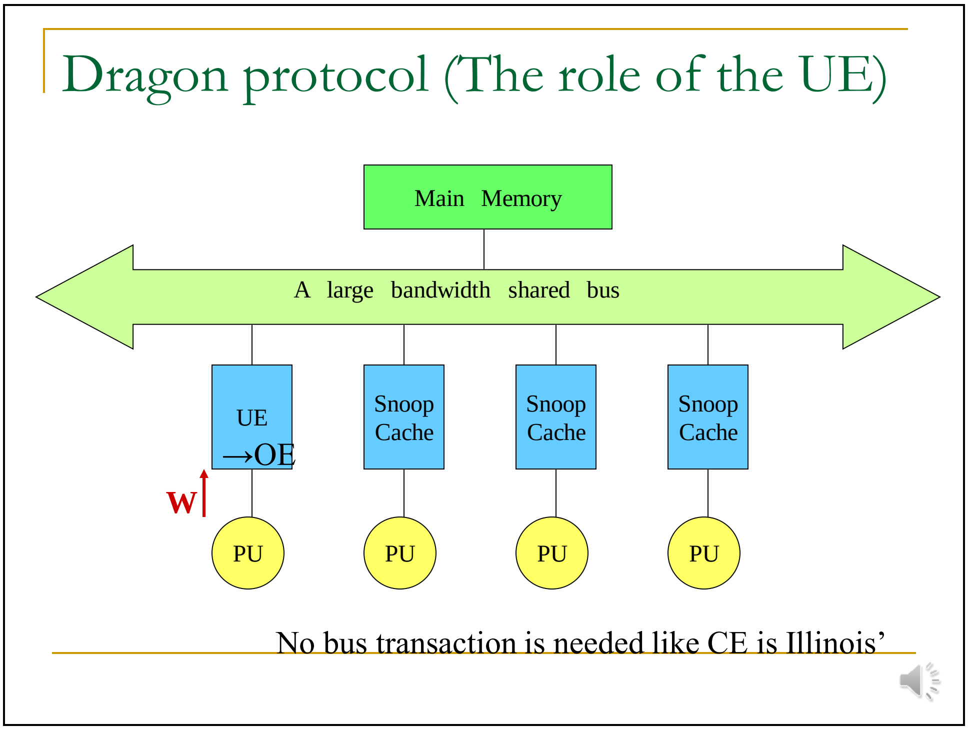
On the other hand, when the PU miss-hits the block, it requires the block to the owner like Berkeley protocol.



Dragon protocol

Direct inter-cache data transfer like Berkeley’s protocol

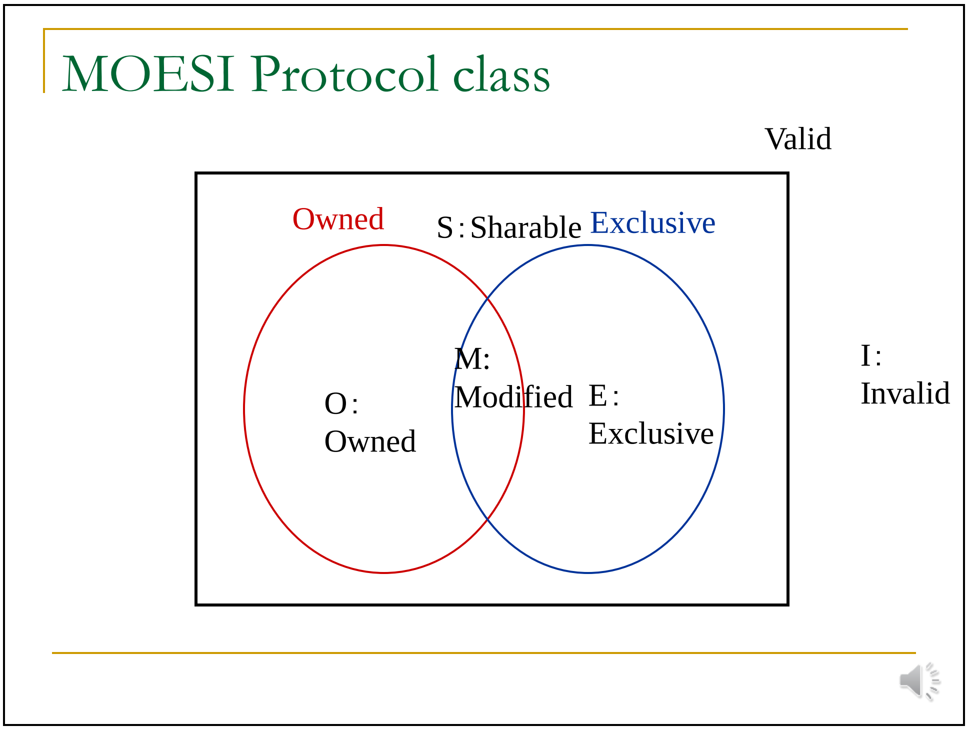
In this case, the cache block is transferred from the owner directly. In this case, the owner becomes OS while the requesting cache block becomes US.



Dragon protocol (The role of the UE)

No bus transaction is needed like CE is Illinois’

Like Illinois protocol, the writing request to US block changes its state into OE without using any bus transaction.



MOESI Protocol class

Valid

Owned

S : Sharable

Exclusive

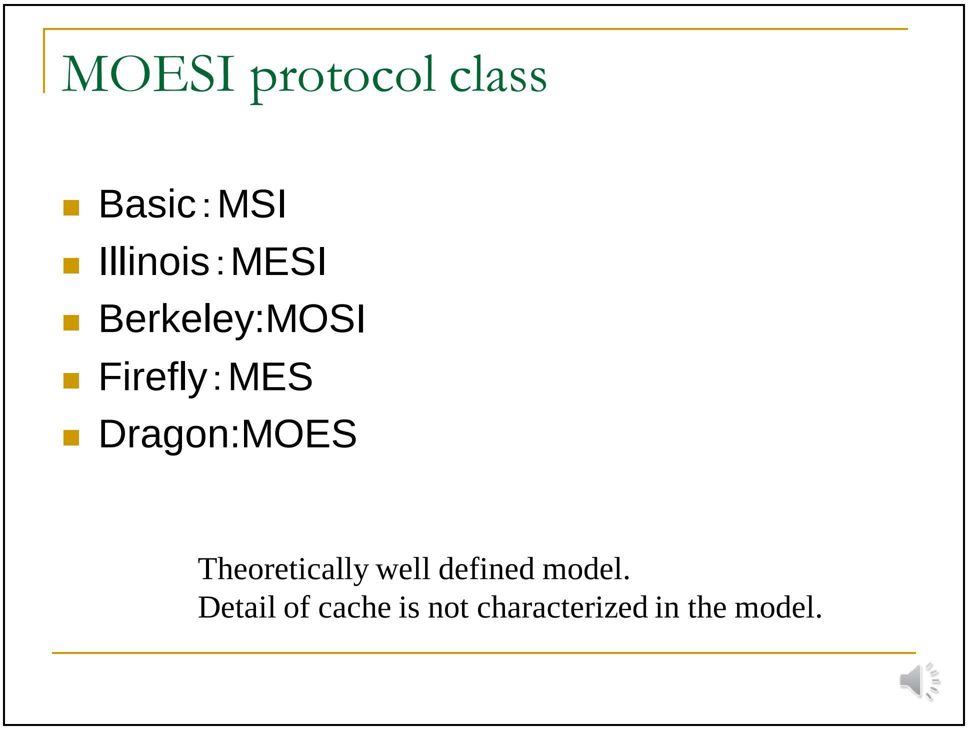
M: Modified

O : Owned

E : Exclusive

I : Invalid

The cache coherence protocol can be classified by the states attached to each cache block. First, the block is classified into valid or invalid. For the valid block, if there is no other copy, it is exclusive. Also, if it is an owner, the state is owned. Thus, a block is in one of five states OE, OS, UE, US, and I. In order to simple representation, OE is called M for modified, OS is O for owned, UE is E for exclusive, US is S for sharable, and I for invalidate. Since the protocol is represented with five letters MOESI, it is called the MOESI protocol class.

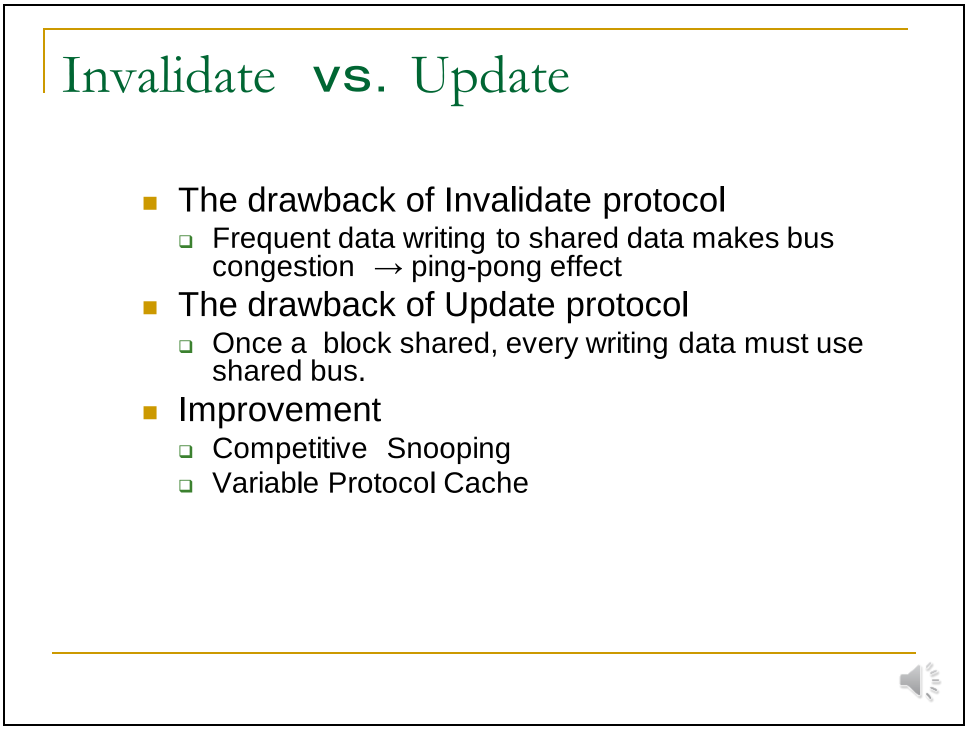


MOESI protocol class

- Basic：MSI
- Illinois：MESI
- Berkeley:MOSI
- Firefly：MES
- Dragon:MOES

Theoretically well defined model.
Detail of cache is not characterized in the model.

The protocols introduced before were summarized in this slide.

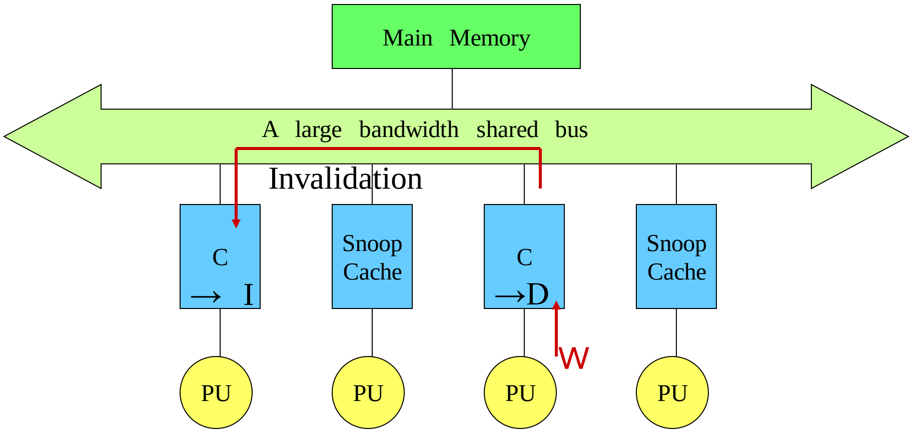


Invalidate vs. Update

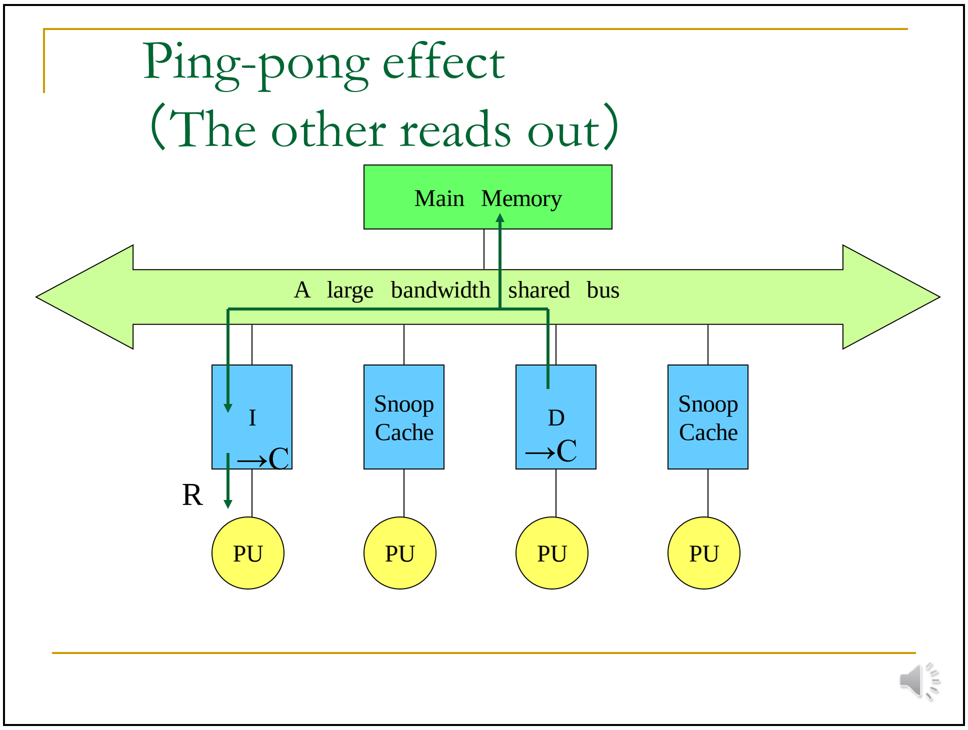
- The drawback of Invalidate protocol
  - Frequent data writing to shared data makes bus congestion → ping-pong effect
- The drawback of Update protocol
  - Once a block shared, every writing data must use shared bus.
- Improvement
  - Competitive Snooping
  - Variable Protocol Cache

Let’s compare the invalidate protocol versus update protocol. The drawback of invalidate protocol is the bus congestion caused by the frequent data writing to the shared data.

Ping-pong effect（A PU writes into）



Assume that these two PUs share the same cache block and frequently write and read data. When this PU writes the data, it invalidates the copy with invalidation signal.



Ping-pong effect (The other reads out)

Main Memory

A large bandwidth shared bus

I →C

R

Snoop Cache

D →C

Snoop Cache

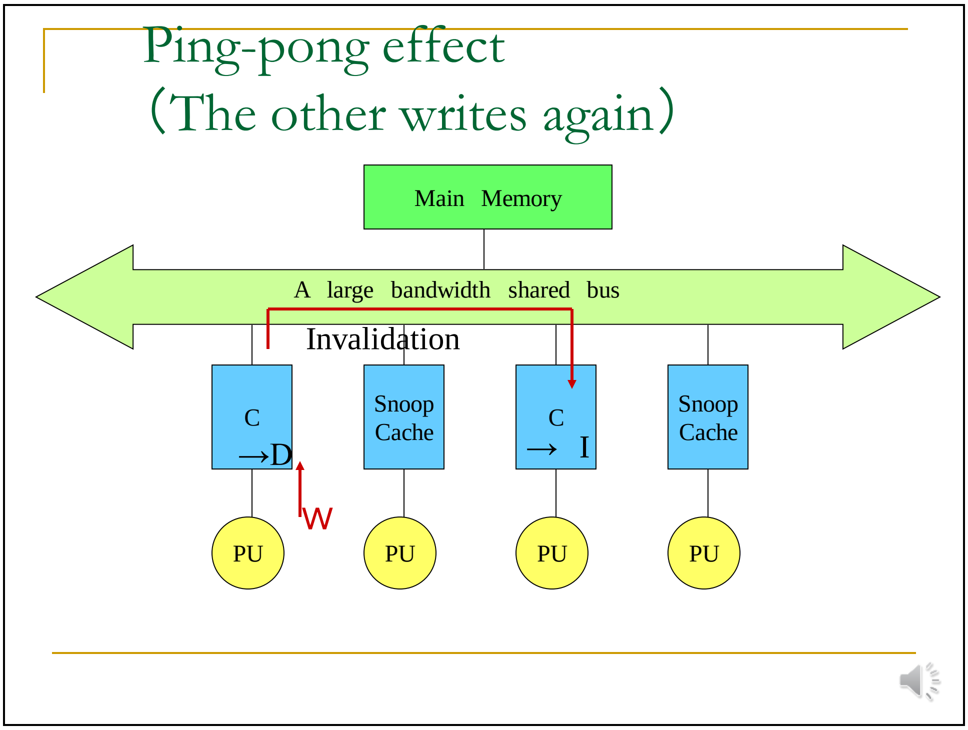
PU

PU

PU

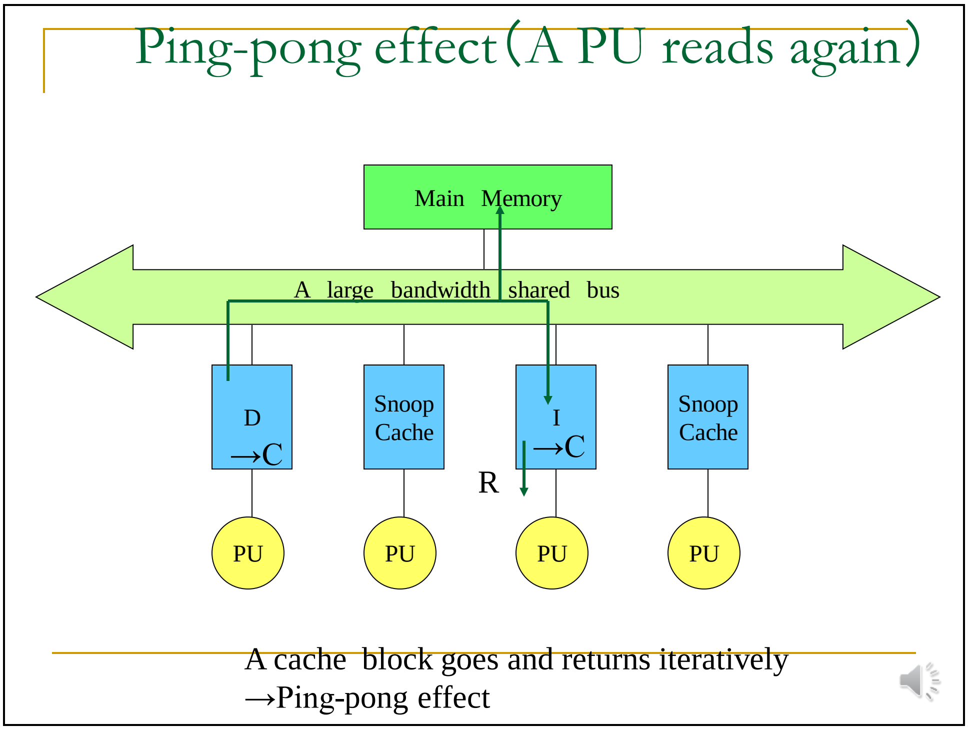
PU

When PU attached to the invalidated cache reads the data, the cache block is written back to the main memory and transferred to the requesting cache.



Ping-pong effect
（The other writes again）

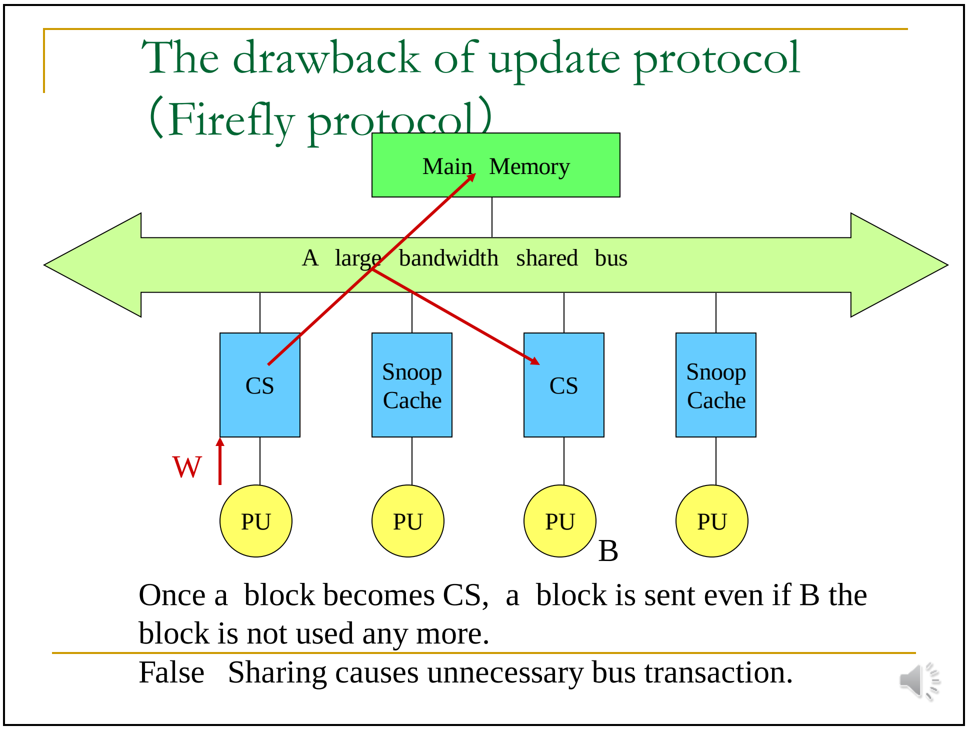
Then this time, assume that this PU writes the data, this block is invalidated.



Ping-pong effect（A PU reads again）

A cache block goes and returns iteratively
→Ping-pong effect

Then, if the PU reads the data, this time the data block is transferred the opposite direction. That is, by the frequent reads/writes by two PUs for the same block, a cache block goes and returns iteratively. This phenomenon is called the ping-pong effect.



The drawback of update protocol (Firefly protocol)

Once a block becomes CS, a block is sent even if B the block is not used any more.

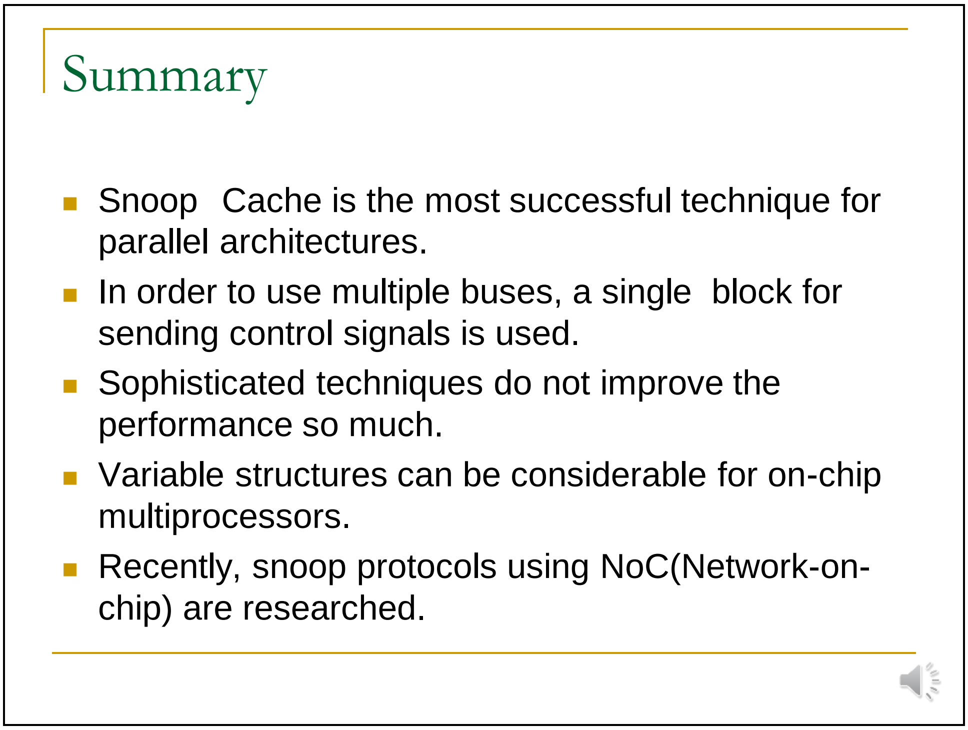
False Sharing causes unnecessary bus transaction.

On the other hand, in the update protocol, once the state becomes CS, all writing data are transferred through the bus, if the update target cache is replaced. That is, the performance is the same as that of write through cache.

There are several proposals to solve these problems, there is no definitive method. In the current multi-core, the MESI or MOESI protocols are used now.

Glossary 2

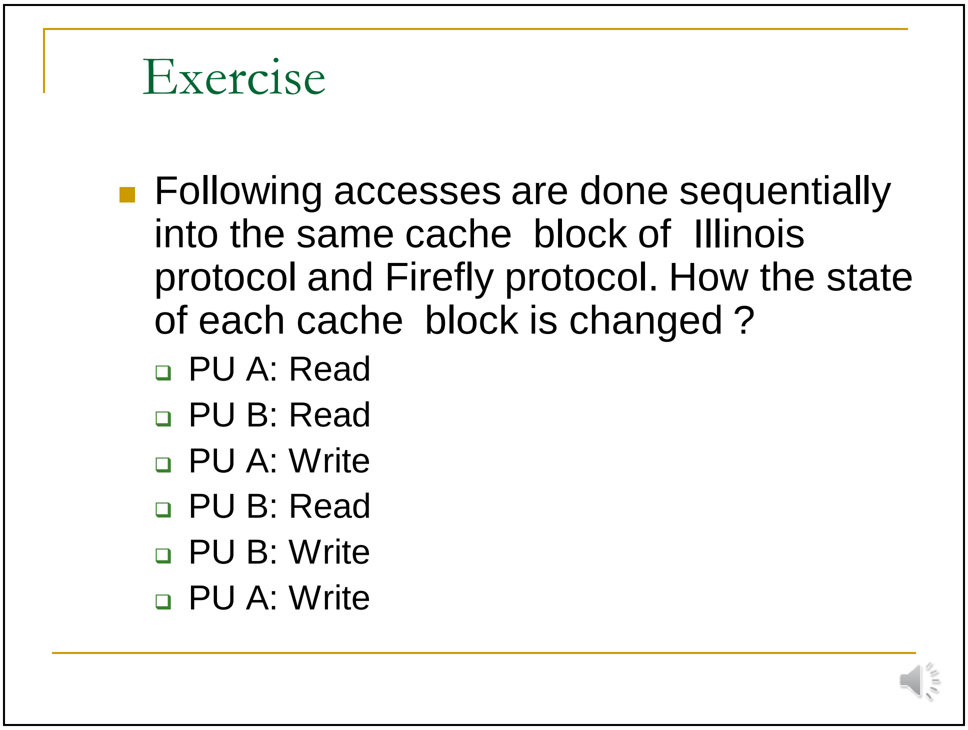
- Invalidation:無効化、Update:更新
- Consistency Protocol: キャッシュの一致性を維持するための取り決め
- Illinois, Berkeley, Dragon, Firefly:プロトコルの名前。IllinoisとBerkeleyは提案した大学名、Dragon,FireflyはそれぞれXeroxとDECのマシン名
- Exclusive:排他的、つまり他にコピーが存在しないこと
- Modify:変更したこと
- Owner:オーナ、所持者だが実は主記憶との一致に責任を持つ責任者である。Ownershipは所有権
- Competitive:競合的、この場合は二つの方法を切り替える意に使っている。
- Injection:注入、つまり取ってくるというよりは押し込んでしまう意



Summary

- Snoop Cache is the most successful technique for parallel architectures.
- In order to use multiple buses, a single block for sending control signals is used.
- Sophisticated techniques do not improve the performance so much.
- Variable structures can be considerable for on-chip multiprocessors.
- Recently, snoop protocols using NoC(Network-on-chip) are researched.

This slide shows the summary of today’s lesson.

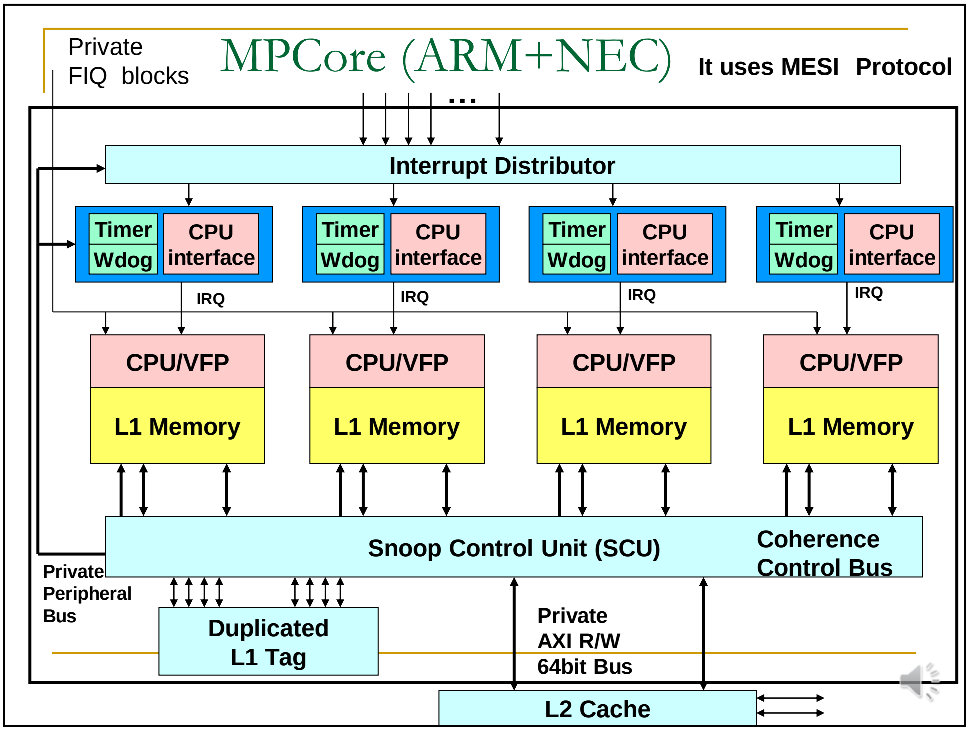


Exercise

- Following accesses are done sequentially into the same cache block of Illinois protocol and Firefly protocol. How the state of each cache block is changed ?
  - PU A: Read
  - PU B: Read
  - PU A: Write
  - PU B: Read
  - PU B: Write
  - PU A: Write

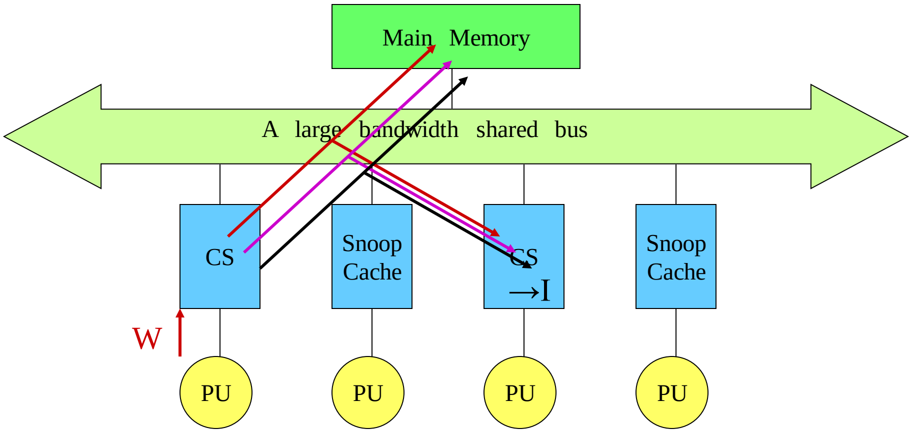
MPCore (ARM+NEC)

It uses MESI Protocol



４つのコアを搭載可能、共有データのキャッシングはスヌープで管理する

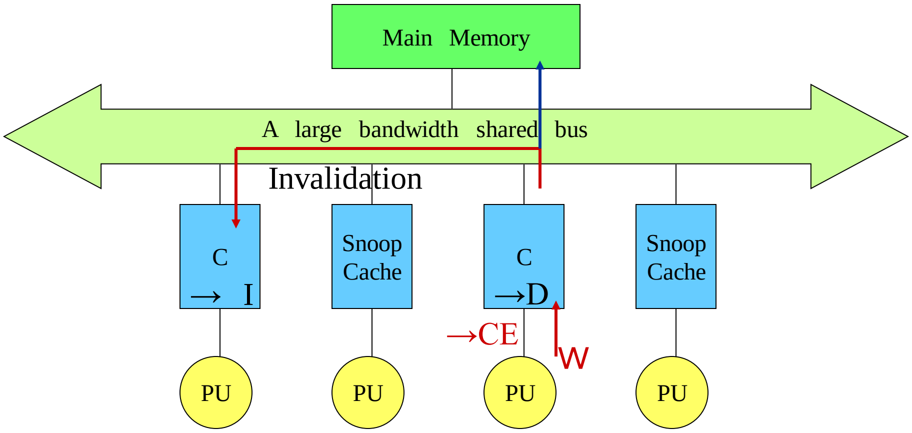
Competitive Snooping



Update *n* times, and then invalidates

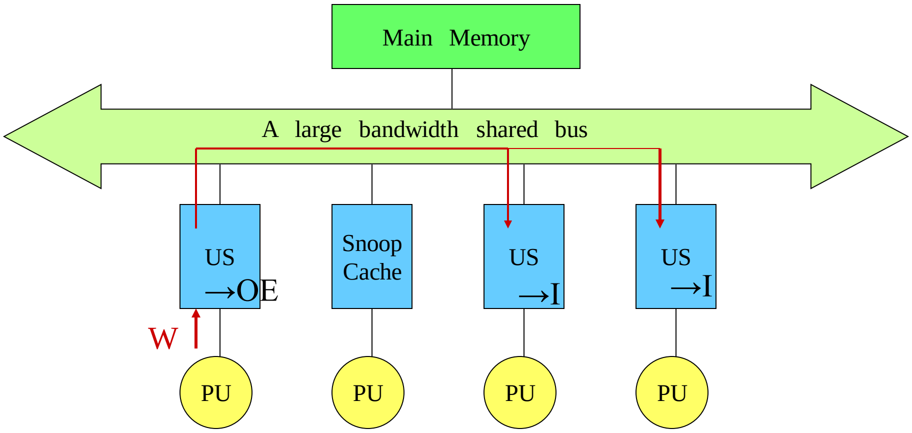
The performance is degraded in some cases.

Write Once (Goodman Protocol)



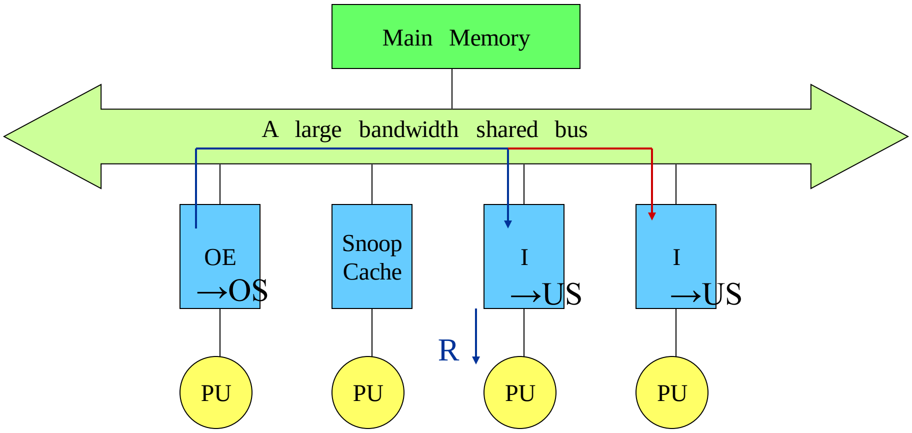
Main memory is updated with invalidation.
Only the first written data is transferred to the main memory.

Read Broadcast(Berkeley)



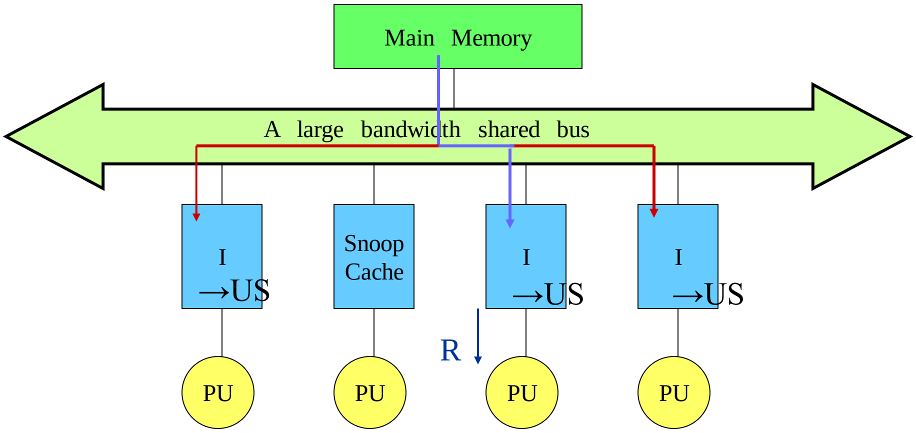
Invalidation is the same as the basic protocol.

Read Broadcast



Read data is broadcast to other invalidated cache.

Cache injection



The same block is injected.