Fiscal Year 2018

Ver. 2019-01-27a

Course number: CSC.T433
School of Computing,
Graduate major in Computer Science

Advanced Computer Architecture

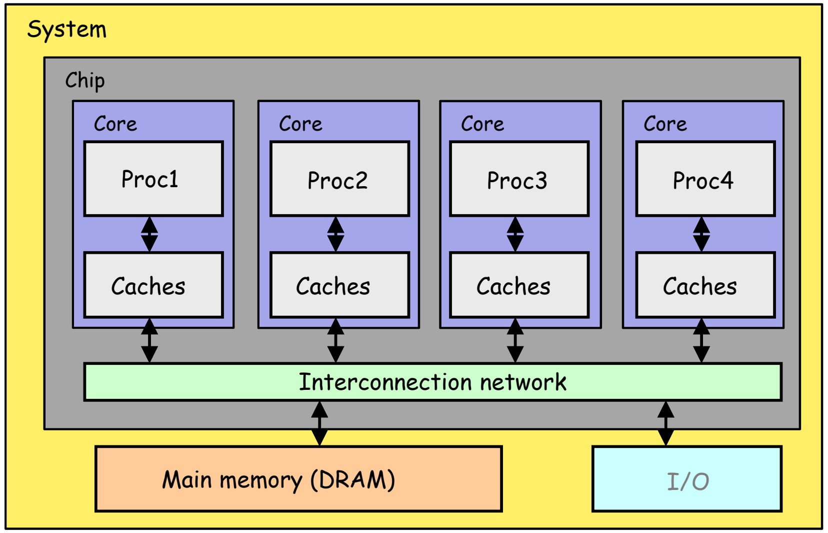
13. Thread Level Parallelism: Coherence and Synchronization

www.arch.cs.titech.ac.jp/lecture/ACA/
Room No.W936
Mon 13:20-14:50, Thr 13:20-14:50

Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp

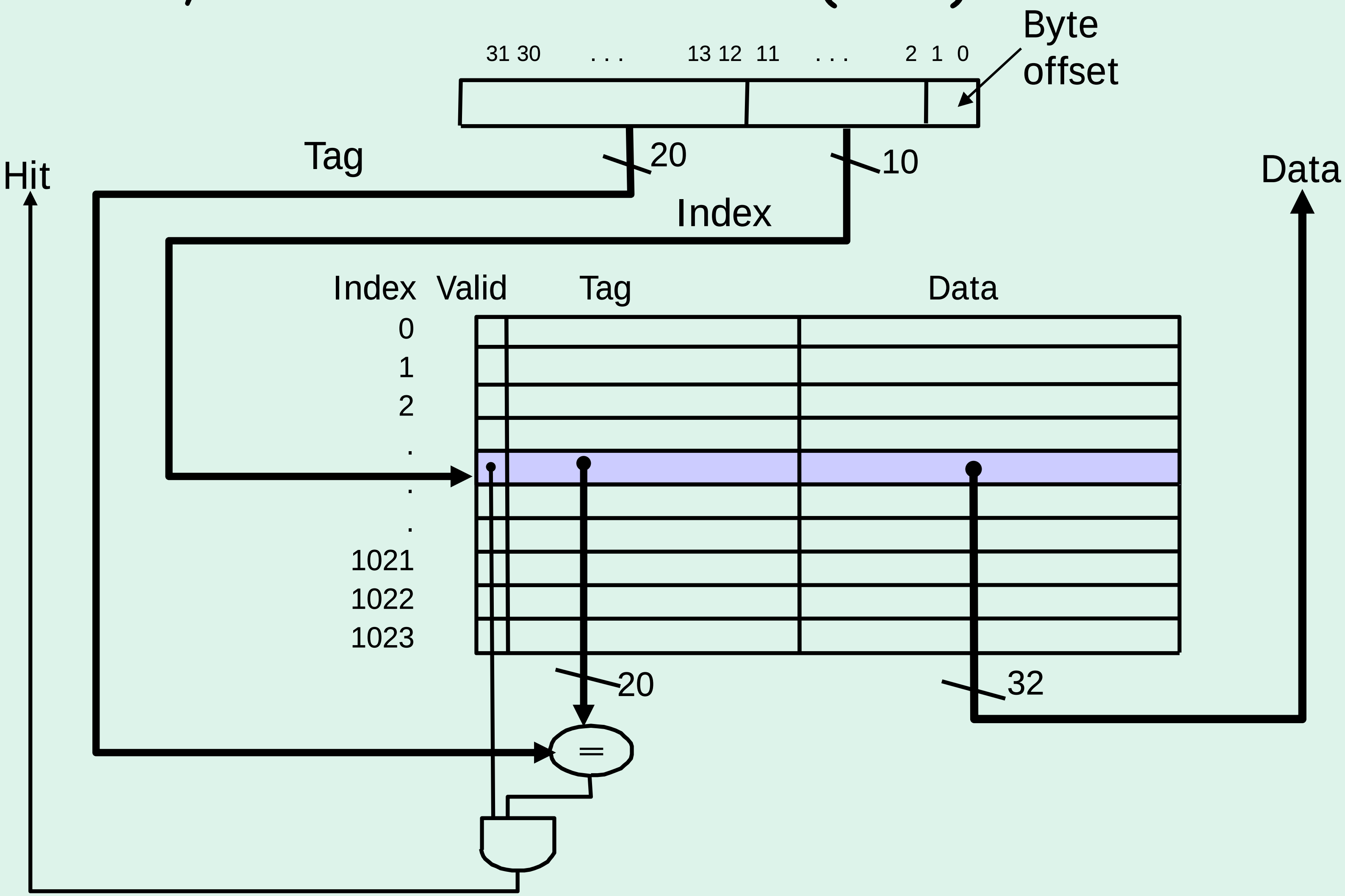
Key components of many-core processors

- Main memory and caches
  - Caches are used to reduce latency and to lower network traffic
  - A parallel program has private data and shared data
  - New issues are cache coherence and memory consistency
- Interconnection network
  - connecting many modules on a chip achieving high throughput and low latency
- Core
  - High-performance superscalar processor providing a hardware mechanism to support thread synchronization



MIPS Direct Mapped Cache Example

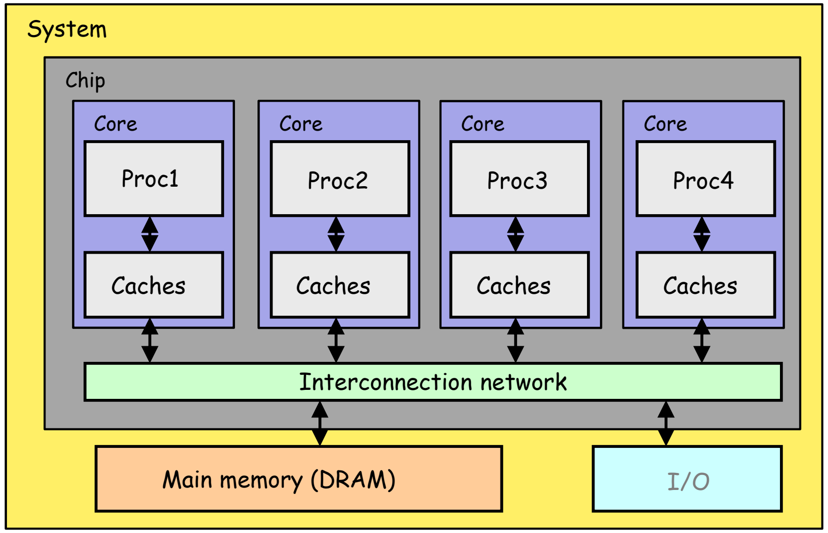
- One word/block, cache size = 1K words (4KB)



What kind of locality are we taking advantage of?

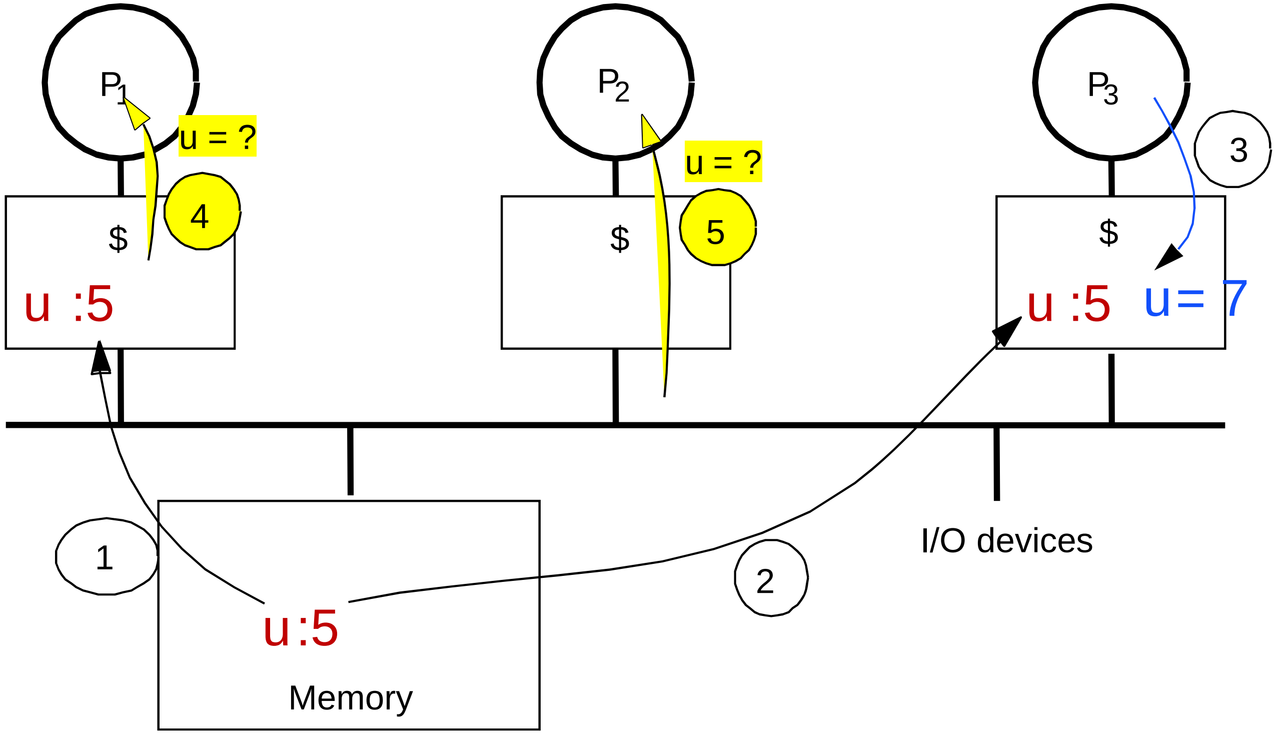
Cache writing policy

- Write-through
  - writing is done synchronously both to the cache and to the main memory. All stores update the main memory.
- Write-back
  - initially, writing is done only to the cache. The write to the main memory is postponed until the modified content is about to be replaced by another cache block.
  - reduces the required network and memory bandwidth.
- Which policy is better for many-core?



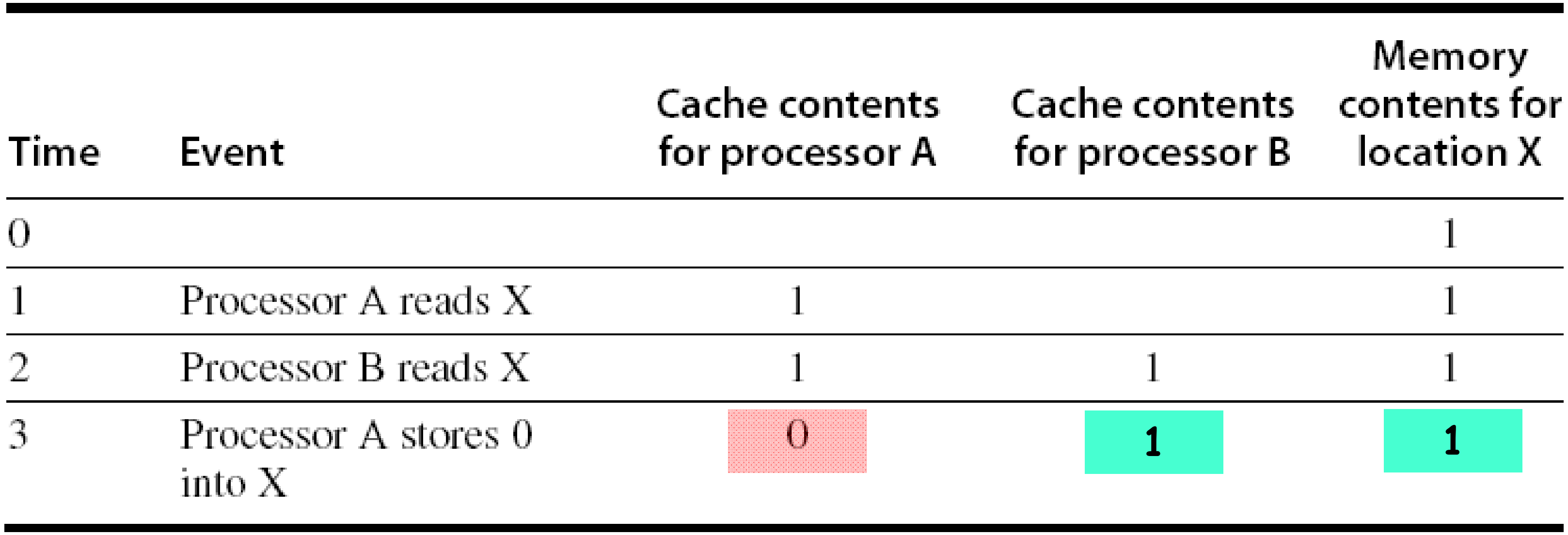
Cache Coherence Problem

- Processors see different values for shared data u after event 3
- With write back caches, value written back to memory depends on which cache flushes or writes back value when
  - Processes accessing main memory may see stale (out-of-date) value
- Unacceptable for programming, and its frequent!



Cache Coherence Problem

- Processors may see different values through their caches
  - assuming a write-back cache
  - after the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not



Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	**1**	**1**

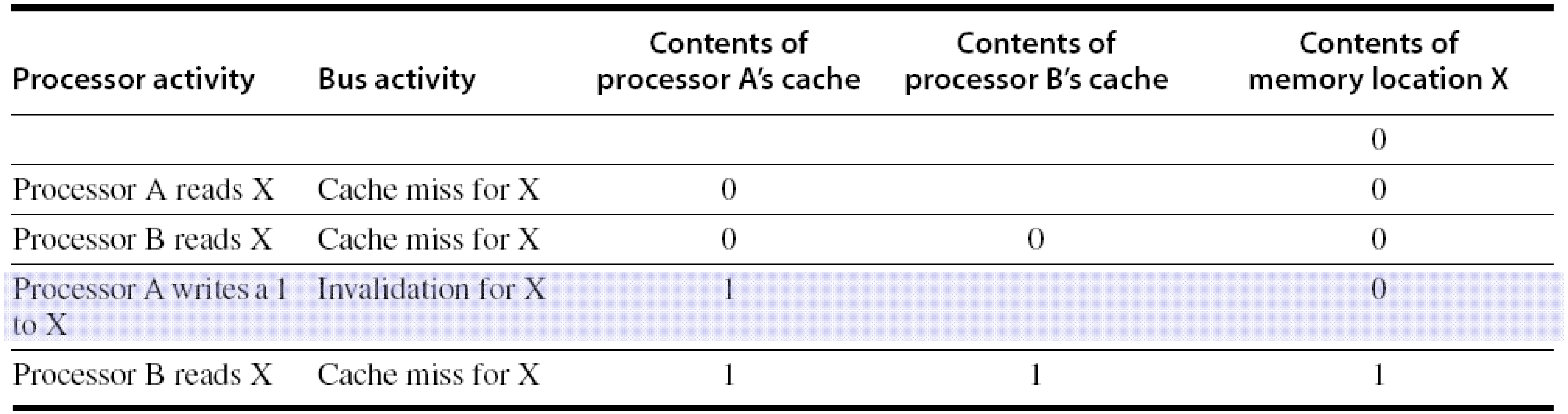
Cache Coherence and enforcing coherence

- Cache Coherence
  - All reads by any processor must return the most recently written value
  - Writes to the same location by any two processors are seen in the same order by all processors

- Cache coherence protocols
  - Snooping (write invalidate / write update)
    - Each core tracks sharing status of each block
  - Directory based
    - Sharing status of each block kept in one location

Snooping coherence protocols using bus network

- Write invalidate
  - On write, invalidate all other copies by an invalidate broadcast
  - Use bus itself to serialize
    - Write cannot complete until bus access is obtained



Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

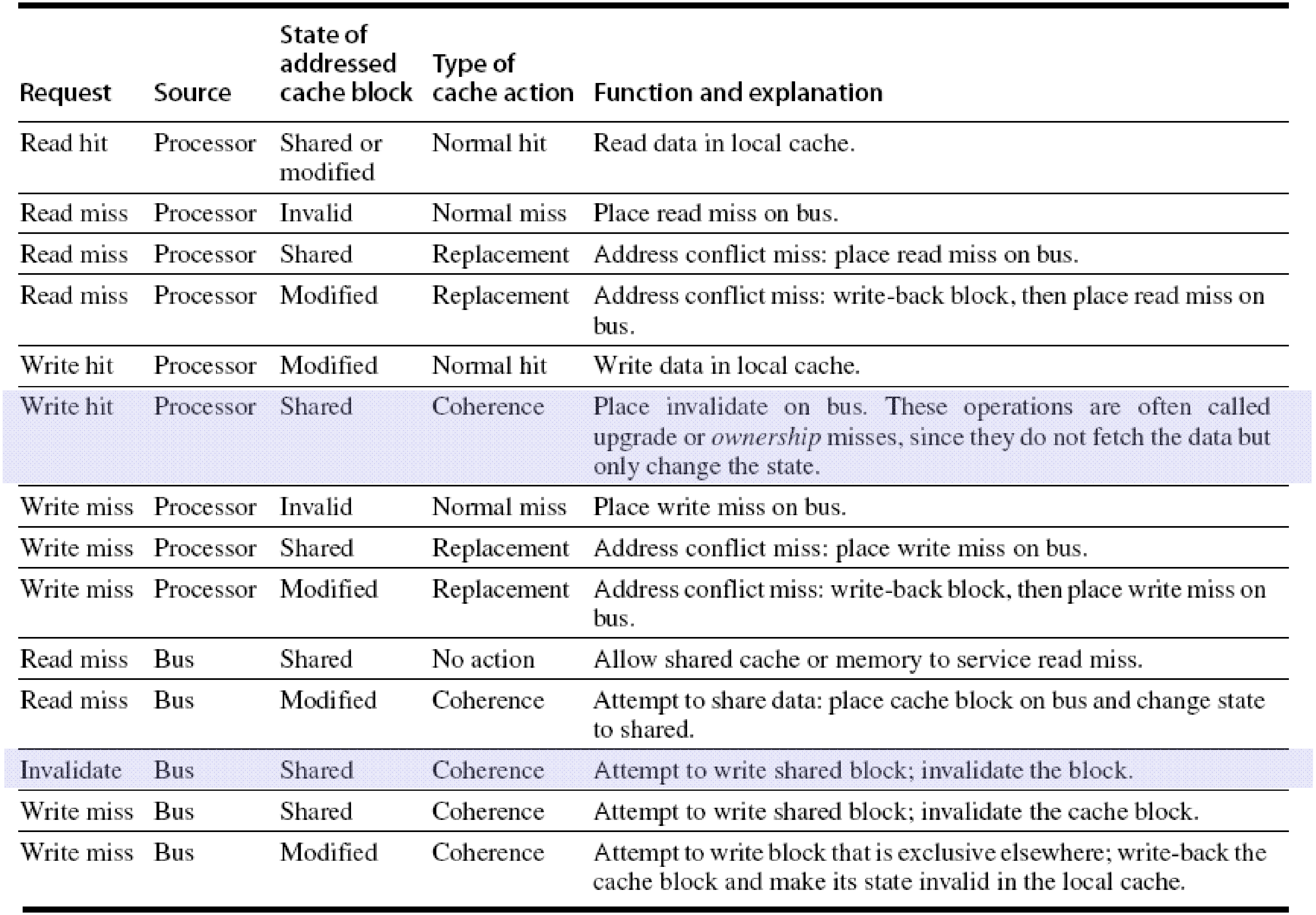
- Write update
  - On write, update all copies

Snooping coherence protocols using bus network

- Cache lines marked as invalid, shared or modified (exclusive)
    - The shared state indicates that the block in the private cache is potentially shared.
    - The modified state indicates that the block has been updated in the private cache; note that the modified state implies that the block is exclusive.
    - Only writes to shared lines need an invalidate broadcast
    - After this, the line is marked as exclusive

Snooping coherence protocols using bus network

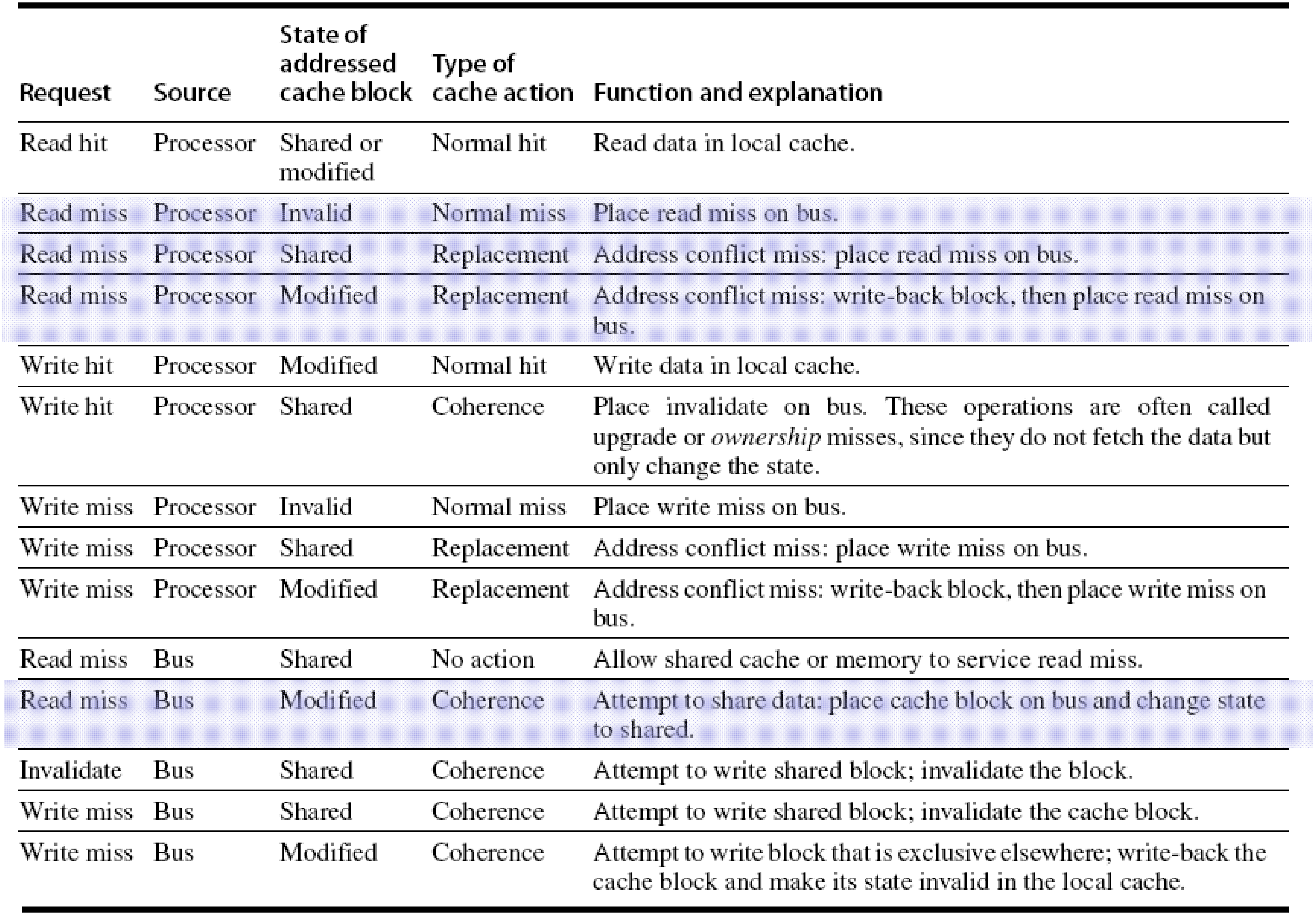
- The coherence mechanism of a private cache



	Request	Source	State of addressed cache block	Type of cache action	Function and explanation
	Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
	Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
	Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
	Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
	Write hit	Processor	Modified	Normal hit	Write data in local cache.
C1	Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
	Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
	Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
	Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
	Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
C2	Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
C3	Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
C4	Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
C5	Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Snooping coherence protocols using bus network

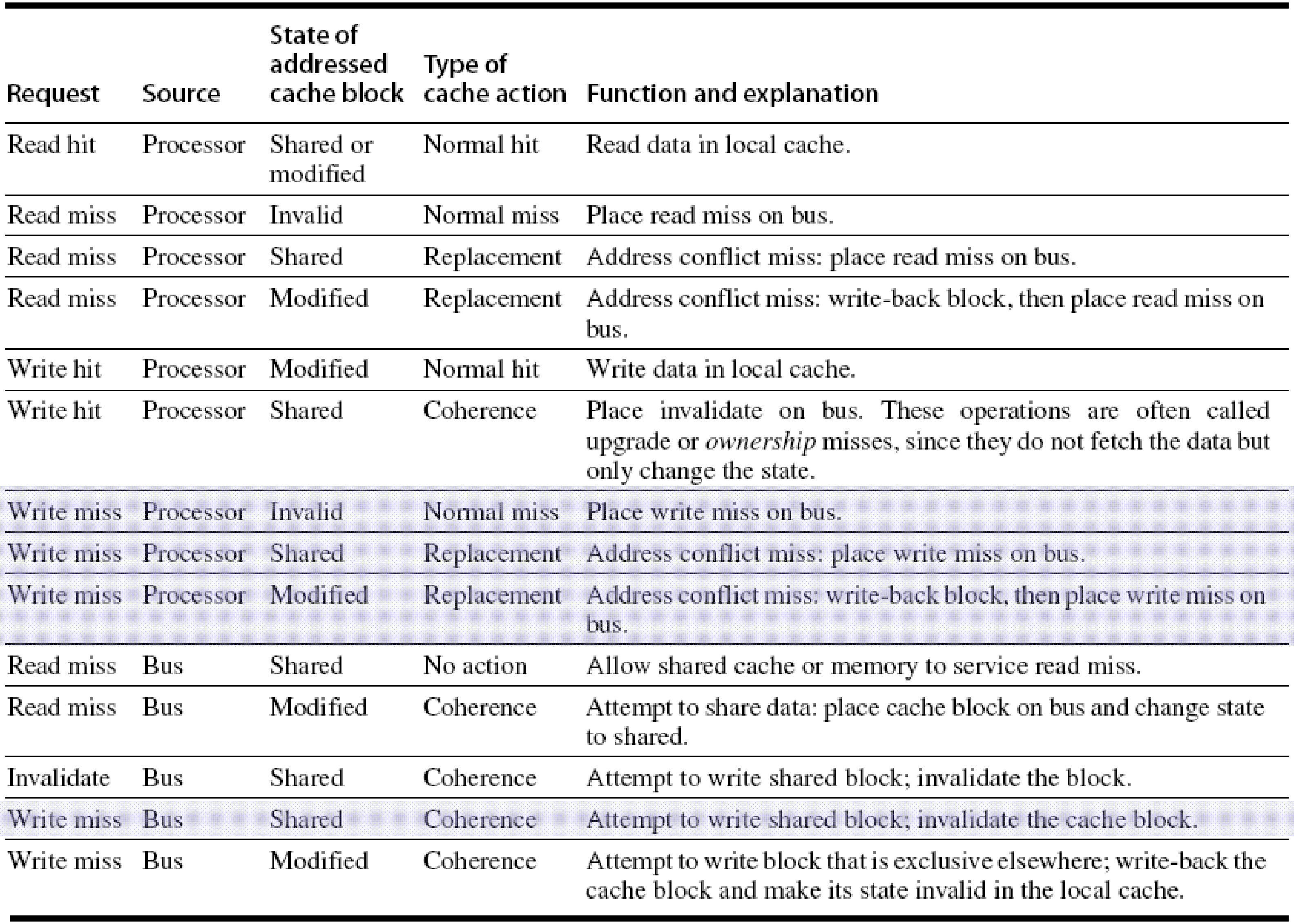
- The coherence mechanism of a private cache



	Request	Source	State of addressed cache block	Type of cache action	Function and explanation
	Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
	Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
	Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
	Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
	Write hit	Processor	Modified	Normal hit	Write data in local cache.
C1	Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
	Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
	Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
	Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
	Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
C2	Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
C3	Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
C4	Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
C5	Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Snooping coherence protocols using bus network

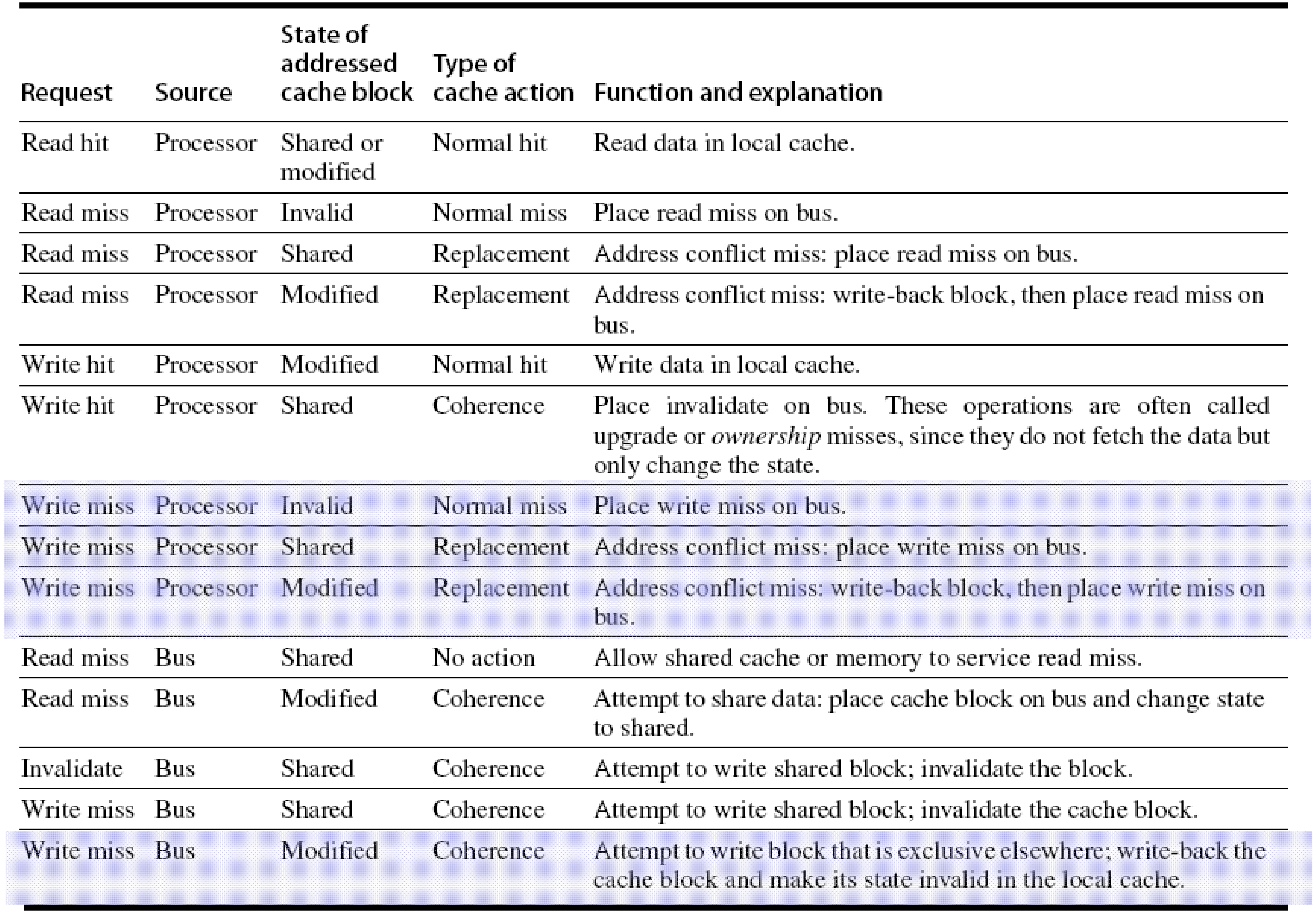
- The coherence mechanism of a private cache



	Request	Source	State of addressed cache block	Type of cache action	Function and explanation
	Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
	Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
	Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
	Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
	Write hit	Processor	Modified	Normal hit	Write data in local cache.
C1	Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
	Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
	Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
	Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
	Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
C2	Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
C3	Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
C4	Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
C5	Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Snooping coherence protocols using bus network

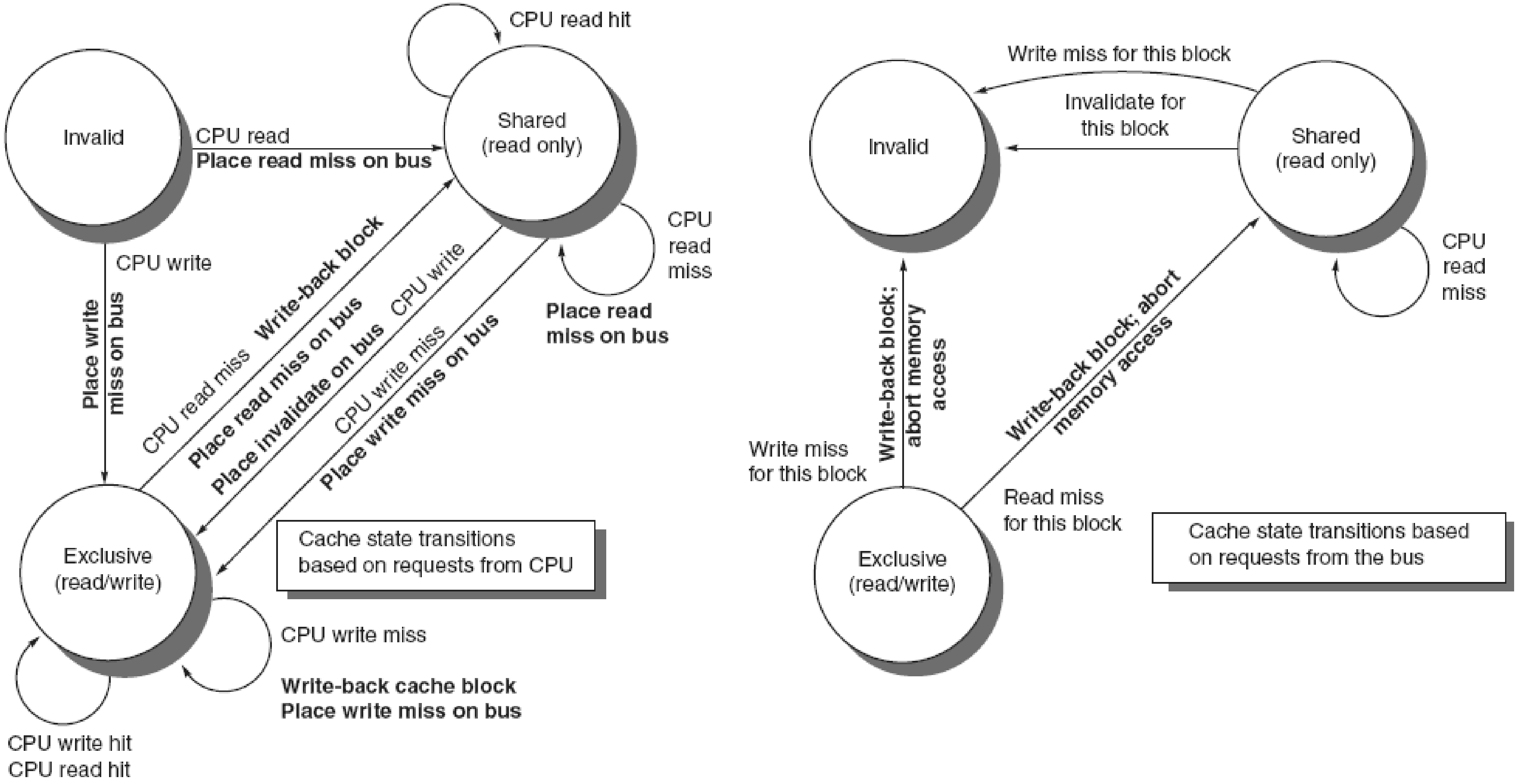
- The coherence mechanism of a private cache



	Request	Source	State of addressed cache block	Type of cache action	Function and explanation
	Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
	Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
	Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
	Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
	Write hit	Processor	Modified	Normal hit	Write data in local cache.
C1	Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
	Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
	Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
	Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
	Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
C2	Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
C3	Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
C4	Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
C5	Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Snooping coherence protocols using bus network

- A write invalidate, cache coherence protocol for a private write-back cache showing the states and state transitions for each block in the cache



Snooping coherence protocols using bus network

- The basic coherence protocol
    - MSI (Modified, Shared, Invalid) protocol
- Extensions
    - MESI (Modified, Exclusive, Shared, Invalid) protocol
    - MOESI (MESI + Owned) protocol

Directory Protocols

- Snooping coherence protocols are based on the use of bus network.
  What are the protocols for mesh topology NoC?

- Directory protocols
  - A logically-central directory keeps track of where the copies of each cache block reside. Caches consult this directory to ensure coherence.

Coherence influences cache miss rate

- Coherence misses
  - True sharing misses
    - Write to shared block (transmission of invalidation)
    - Read an invalidated block
  - False sharing misses
    - Read an unmodified word in an invalidated block

Decomposition and assignment

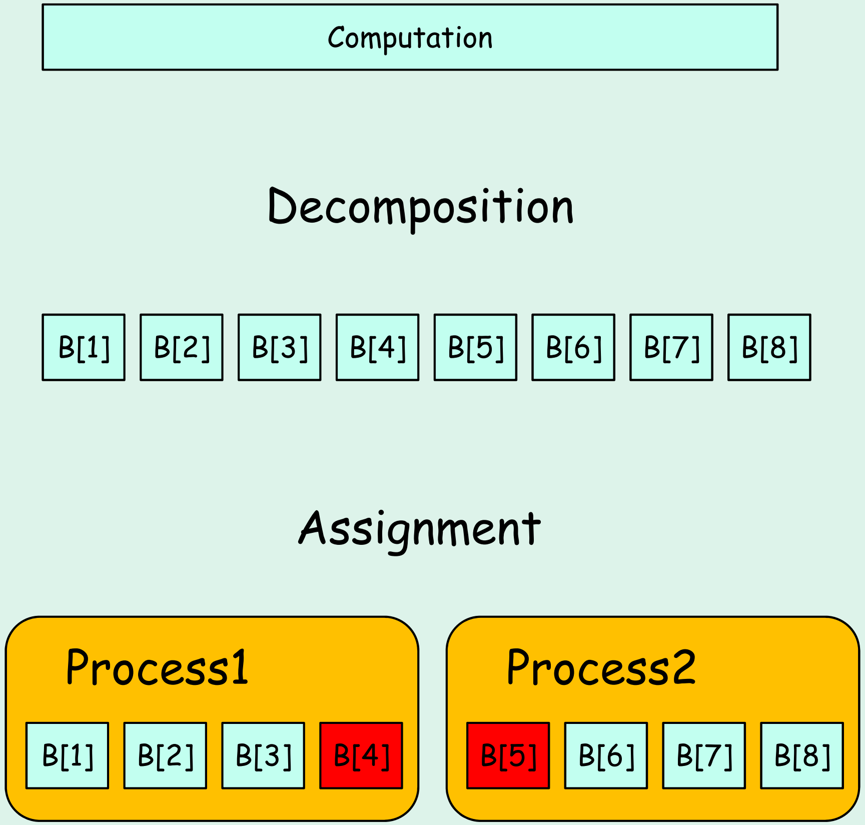
- Single Program Multiple Data (SPMD)
  - Decomposition: there are eight tasks to compute B[i]
  - Assignment: the first four tasks for process1, the last four for process2

```
float A[N+2], B[N+2]; /* these are in shared memory */
float diff=0;         /* variable  in shared memory */

void solve_pp (int pid, int ncores) {
    int i, done = 0;                    /* private variables */
    int mymin = 1 + (pid * N/ncores);   /* private variable  */
    int mymax = mymin + N/ncores – 1;   /* private variable  */
    while (!done) {
        float mydiff = 0;
        for (i=mymin; i<=mymax; i++) {
            B[i] = 0.333 * (A[i-1] + A[i] + A[i+1]);
            mydiff = mydiff + fabsf(B[i] - A[i]);
        }
        diff = diff + mydiff;

        if (diff <TOL) done = 1;
        if (pid==1) diff = 0;
        for (i=mymin; i<=mymax; i++) A[i] = B[i];
    }
}

int main() { /* solve this using two cores */
    initialize shared data A and B;
    create thread1 and call solve_pp(1, 2);
    create thread2 and call solve_pp(2, 2);
}
```



Computation

Decomposition

B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]

Assignment

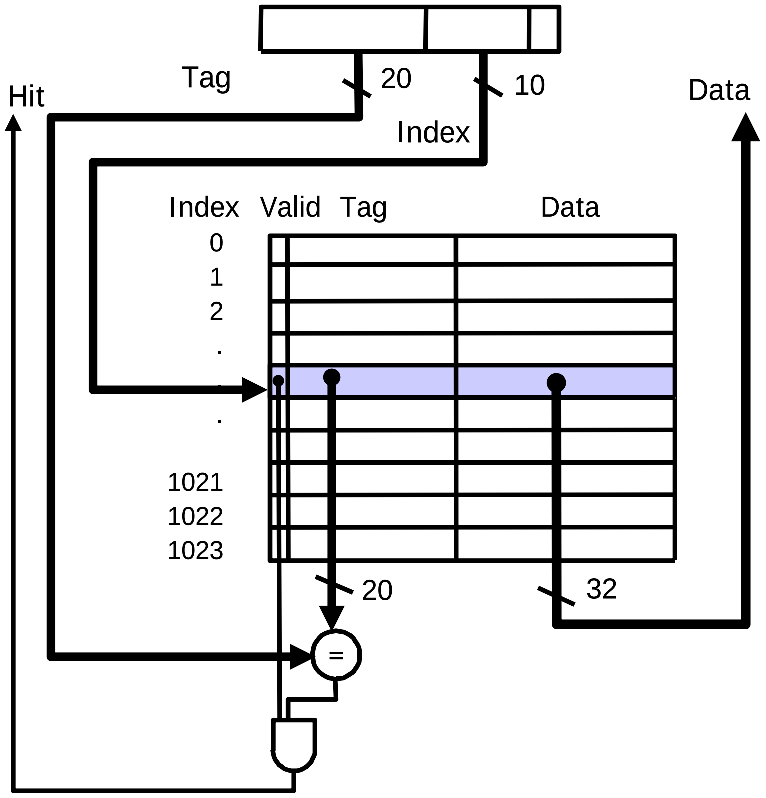
Process1

B[1]	B[2]	B[3]	B[4]

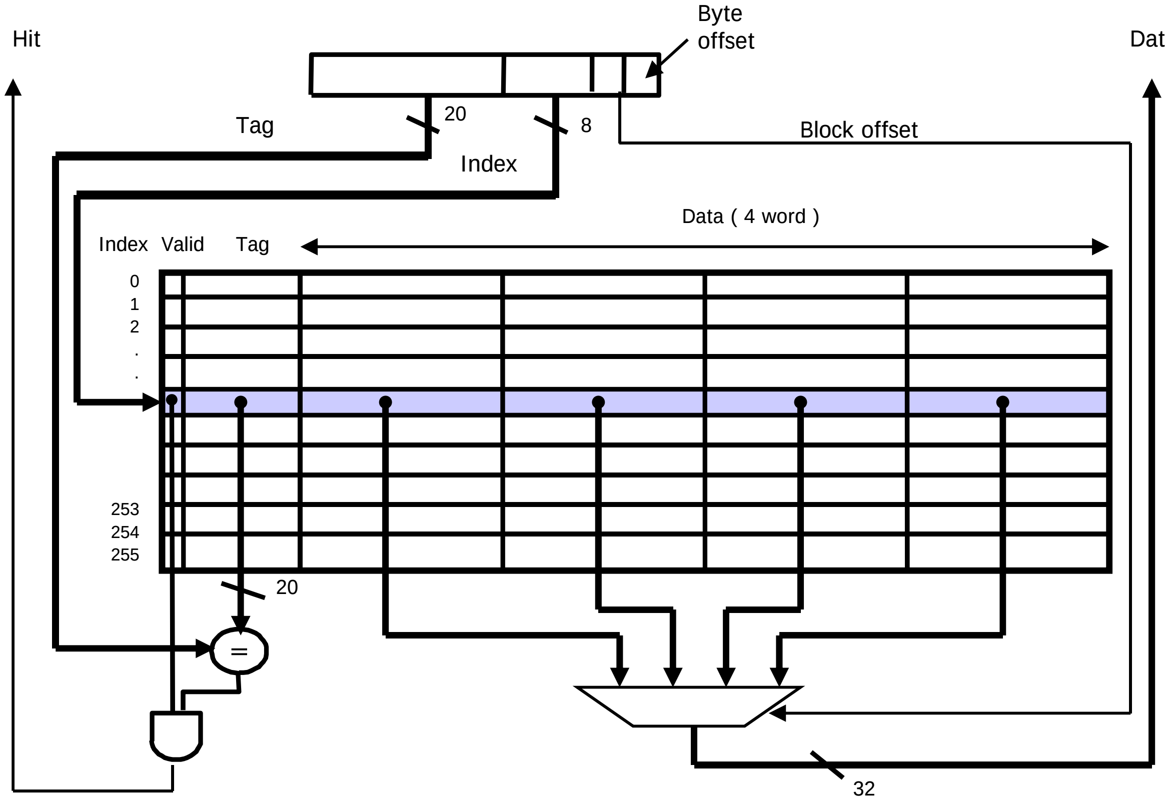
Process2

B[5]	B[6]	B[7]	B[8]

Two caches of different block sizes



One word/block



Four words/block

Synchronization

- Basic building blocks:
  - Atomic exchange
    - Swaps register with memory location
  - Test-and-set
    - Sets under condition
  - Fetch-and-increment
    - Reads original value from memory and increments it in memory
  - These requires memory read and write in uninterruptable instruction

  - load linked/store conditional
    - If the contents of the memory location specified by the load linked are changed before the store conditional to the same address, the store conditional fails

Orchestration

- LOCK and UNLOCK around critical section
    - Set of operations we want to execute atomically
- BARRIER ensures all reach here

```
float A[N+2], B[N+2]; /* these are in shared memory */
float diff=0.0;       /* variable  in shared memory */

void solve_pp (int pid, int ncores) {
    int i, done = 0;                   /* private variables */
    int mymin = 1 + (pid * N/ncores);  /* private variable  */
    int mymax = mymin + N/ncores – 1;  /* private variable  */
    while (!done) {
        float mydiff = 0;
        for (i=mymin; i<=mymax; i++) {
            B[i] = 0.333 * (A[i-1] + A[i] + A[i+1]);
            mydiff = mydiff + fabsf(B[i] - A[i]);
        }
        LOCK();
        diff = diff + mydiff;
        UNLOCK();

        BARRIER();
        if (diff <TOL) done = 1;
        BARRIER();
        if (pid==1) diff = 0;
        for (i=mymin; i<=mymax; i++) A[i] = B[i];
        BARRIER();
    }
}
```

These operations must be executed atomically

```
(1) load diff
(2) add
(3) store diff
```

After all cores update the diff,
if statement must be executed.

```
if (diff <TOL) done = 1;
```

Implementing an atomic exchange EXCH

- Load linked/store conditional instructions
  - If the contents of the memory location specified by the load linked are changed before the store conditional to the same address, the store conditional fails
- Store conditional instruction
  - it returns 1 if it was successful and a 0 otherwise

- EXCH R4,0(R1)   ; exchange R4 and 0(R1) atomically

```
try:    ADD R3,R4,R0       ; move exchange value, R3<=R4
        LL  R2,0(R1)       ; load linked
        SC  R3,0(R1)       ; store conditional
        BEQ R3,R0,try      ; branch if store fails (R3==3)
        ADD R4,R2,R0       ; put load value in R4, R4<=R2
```

Implementing Locks using coherence

- Spin lock
  - R1 is the address of the lock variable and its initial value is 0.
  - We can cache the lock using the coherence mechanism to maintain the lock value coherently.
  - This code spins by doing read on a local copy of the lock until it successfully sees that the lock is available (lock variable is 0).

```
lockit:   LD      R2,0(R1)        ; load of lock
          BNE     R2,R0,lockit    ; not available-spin if R2==1
          ADDI    R2,R0,1         ; load locked value, R2<=1
          EXCH    R2,0(R1)        ; swap
          BNE     R2,R0,lockit    ; branch if lock wasn’t 0
```

Implementing Barriers using coherence

- This code counts up the arrived threads using a shared variable counter.
- If all threads counts up the variable, the last thread set the shared variable flag to exit the barrier.

```
BARRIER(){
    LOCK();
        if (counter == 0) flag = 0; /* counter and flag are shared data */
        mycount = counter++;        /* mycount is a private variable    */
    UNLOCK();
    if (mycount == p) {
        counter = 0;
        flag = 1;
    }
    else while (flag == 0) { };     /* wait until all threads reach BARRIER */
}
```