Green Computing Platforms

Step 4 CA11: Multicores and Distributed Memory System

http://archlab.naist.jp/Lectures/ARCH/ca11/ca11e.pdf

Copyright © 2024 NAIST Y.Nakashima

Download the template and submit through UNIPA.

http://archlab.naist.jp/Lectures/ARCH/ca11/ca11e.docx

in http://archlab.naist.jp/Lectures

Parallel processing

If vectorizable, speedup will be possible by parallelization
Key points for parallelization are:

- **Reduce non parallelizable part**
  - Optimize the algorithm
- **Load balancing (equalize the execution time)**
  - Consider cache miss penalty
  - Minimize synchronization overhead
- **Reduce the communication overhead**
  - Reduce number of synchronization
  - Reduce communication traffic volume
  - Reduce number of communication
  - Overlap communication and computation
  - Reduce OS overhead

Multi-Threading and Multi-Processor

SMT (Simultaneous Multi-Threading) … e.g. HyperThreading

- Superscalar cannot make full use of all ALUs
- Execute multiple threads and achieve high utilization of ALUs
  - Equip multiple Program Counter
  - Share ALUs and Cache memory
  - If execute N threads, total throughput will be less than N

CMP (Chip Multi-Processor) … e.g. Multi-Core

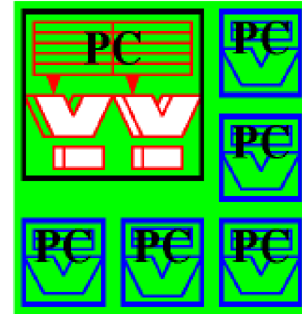
- Implement multiple processors on one LSI
  - Equip multiple Program Counter
  - Dedicated ALU, L1 Cache, and Shared L2 Cache
  - Total throughput will be less than N, but closer to N than SMT
  - Simpler and better energy efficiency than SMT

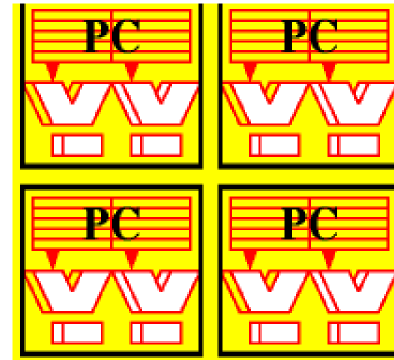
Proposed CMP (Chip Multi Processors)

Heterogeneous configuration

- One (or more) CPU with multiple VLIW/SIMD/DSP
- Offload some processing onto dedicated unit and optimize
- Good for real-time specific processing

IBM/SONY/TOSHIBA	Cell
MATSUSHITA	UniPhier
TOSHIBA	MeP
RENESAS	SH-Mobile
TI	OMAP
NEC	MP211
NECEL	EMMA





Homogeneous configuration

- Connect multiple same processors (can be with VLIW/SIMD/DSP)
- Parallel processing on the multiple CPUs
- Good for software development but bad for real-time processing

ARM	MPCore
FUJITSU	FR-V
INTEL	Core-Extreme

NAIST.

Multi-Threading and Execution model

Focus on thread-level parallelism in addition to instruction-level

Followings are traditional (non speculative) execution model

Execute multiple processes on a processor

- Traditional model
- Each process runs same speed or slower
- Just execute multiple processes

  Typical use of common SMT/CMP

Execute one process on multiple processors

- Traditional model
- Use inter-process communication for programming
- Parallel programing framework (such as OpenMP) is available

  Multi-Threaded application is popular

NAIST.

New execution model (Speculative)

Helper Threads

- Helper threads execute some instructions
- Accelerate main thread but do not update register/memory
- Execution mechanism with no update is required

  Applicable to SMT/CMP

  Prefetching to shared cache and better branch prediction

Speculative Architectural Threads

- Speculative threads execute some region of program
- Speculative threads update register/memory
- Save/Restore mechanism of register/memory is required

  Main thread=> Speculative thread management is required

  Acceleration by automatic parallelization

Traditional parallelization with Multi-Processing

One of the kernel processes in procimage

```
Depth_retrieval_L();
Depth_retrieval_R();
```

Minimum modification for parallelization

Each process has dedicated memory space

Send/Recv to spread/gather data

```
{ int  pfd[2]; int c = 0;
  pipe(pfd);                        /* Create pipe for inter-process communication */
  if (fork()) {                     /* Create child process */
    Depth_retrieval_L();            /* Process L image */
    while (c < sizeof(Dr))          /* Receive result from child process */
      c += read(pfd[0], (char*)&Dr+c, sizeof(Dr)-c);
    close(pfd[0]);
    wait((union wait *)0);          /* Wait termination of child process */
  }
  else {
    Depth_retrieval_R();            /* Process R image */
    while (c < sizeof(Dr))          /* Send result via pipe */
      c += write(pfd[1], (char*)&Dr+c, sizeof(Dr)-c);
    exit(0);                        /* Terminate child process */
  }
}
```

Shared Memory Space

NAIST

Parallel processing by multi-threading

One of the kernel processes in stereo image processing

```
Depth_retrieval_L();
Depth_retrieval_R();
```

Multi-threading by pthread_create()

```
pthread_create(&th_0, NULL, Depth_retrieval_L, NULL);
pthread_create(&th_1, NULL, Depth_retrieval_R, NULL);
pthread_create(&th_2, NULL, …, NULL);
pthread_create(&th_3, NULL, …, NULL);
```

All threads share the memory space. Send/Recv is not required

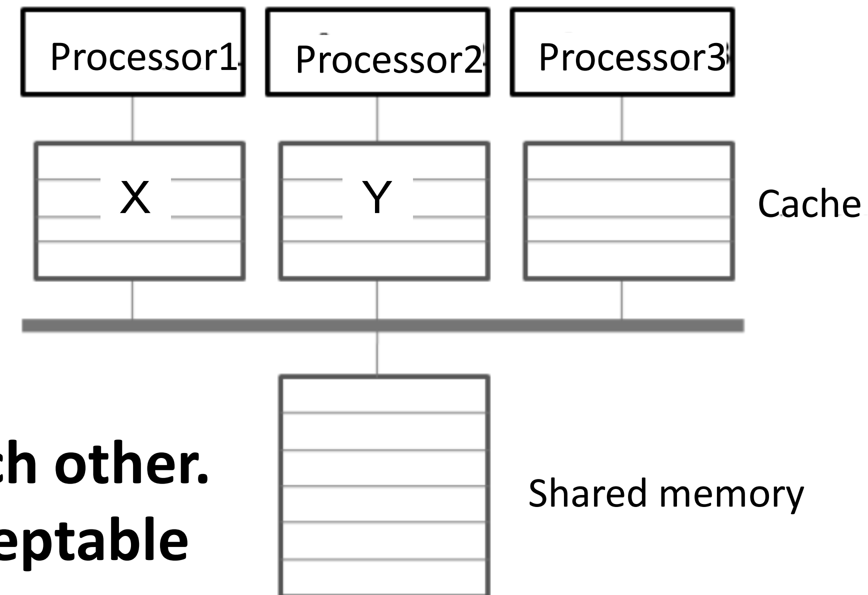
Local variables are not shared. Global variables are shared

```
pthread_join(th_0, &tr_0);
pthread_join(th_1, &tr_1);
pthread_join(th_2, &tr_2);
pthread_join(th_3, &tr_3);
```

Necessity of cache coherent management

- Original data is located in Shared memory
- Cache is important for performance
- How to manage multiple caches without inconsistency

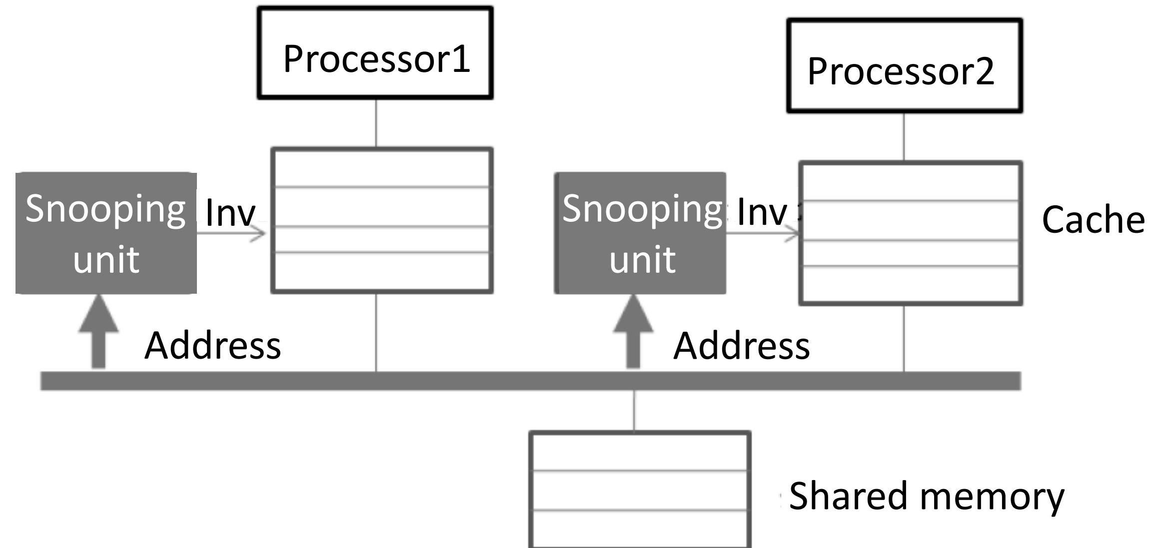
- CPU1 stores X in addr A
- And CPU2 stores Y in addr A



CPU1 and CPU2 should inspect each other.
However, inspection causes unacceptable
Overhead.
As the number of CPUs increases, the overhead
exponentially increases.

What is cache coherent management

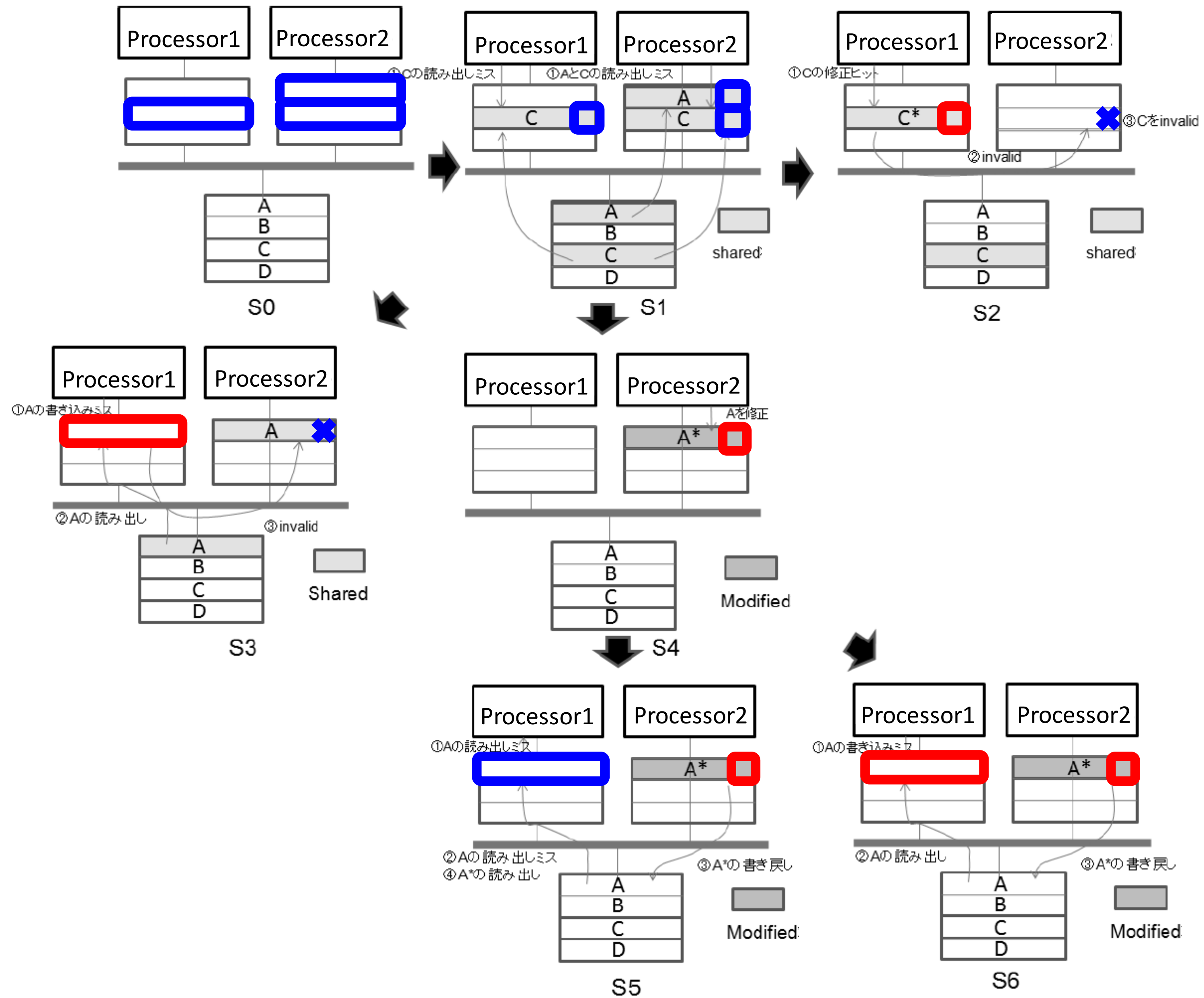
When data is updated, other caches should be invalidated.



Write invalidate protocol for write back cache (MSI protocol)

1)No valid data = Invalid
2)Potentially shared data = Shared
3)Potentially updated data = Modified

MSI Protocol



Cache coherent management for shared memory

MESI Protocol (Illinois protocol) --- Pentium

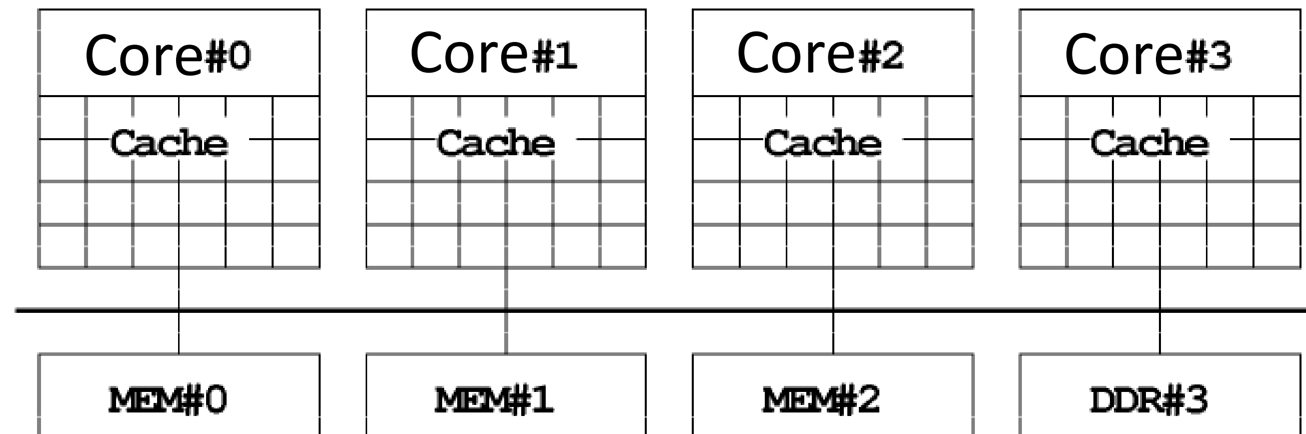
Modify: One cache has valid data (Memory data is obsoleted)
Exclusive:One cache has valid data (Memory data is also valid)
Shared: Some caches have valid data(Memory data is also valid)
Invalid: No cache has valid data
Writable only when M/E
Write for S -> Invalidate other cache
Other's Write for S -> Snoop and Invalidate
Other's Write for M -> Snoop and Writeback+Invalidate

MSI Protocol

E is represented by S, Write notification is always required (Slow)

MOESI Protocol --- AMD64

Owned: One cache has Modified data and some caches have Shared data
Owned provide valid data for other core's cache miss

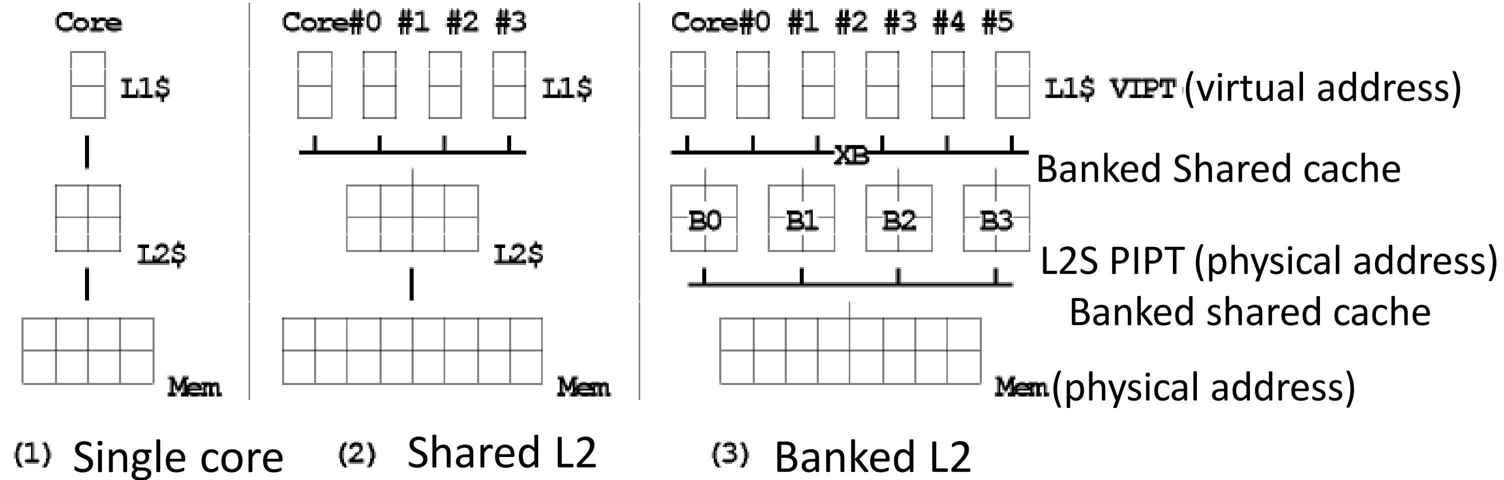


NAIST

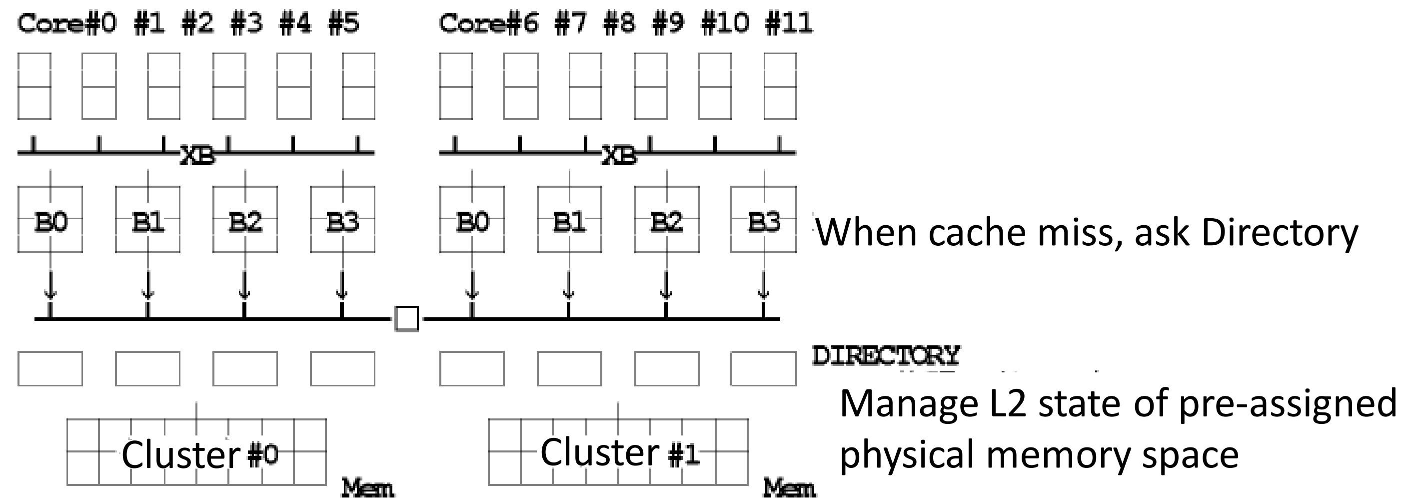
Cache coherent management for shared memory

Coherent mechanism becomes complicated for many cores.

Banked memory + Directory management



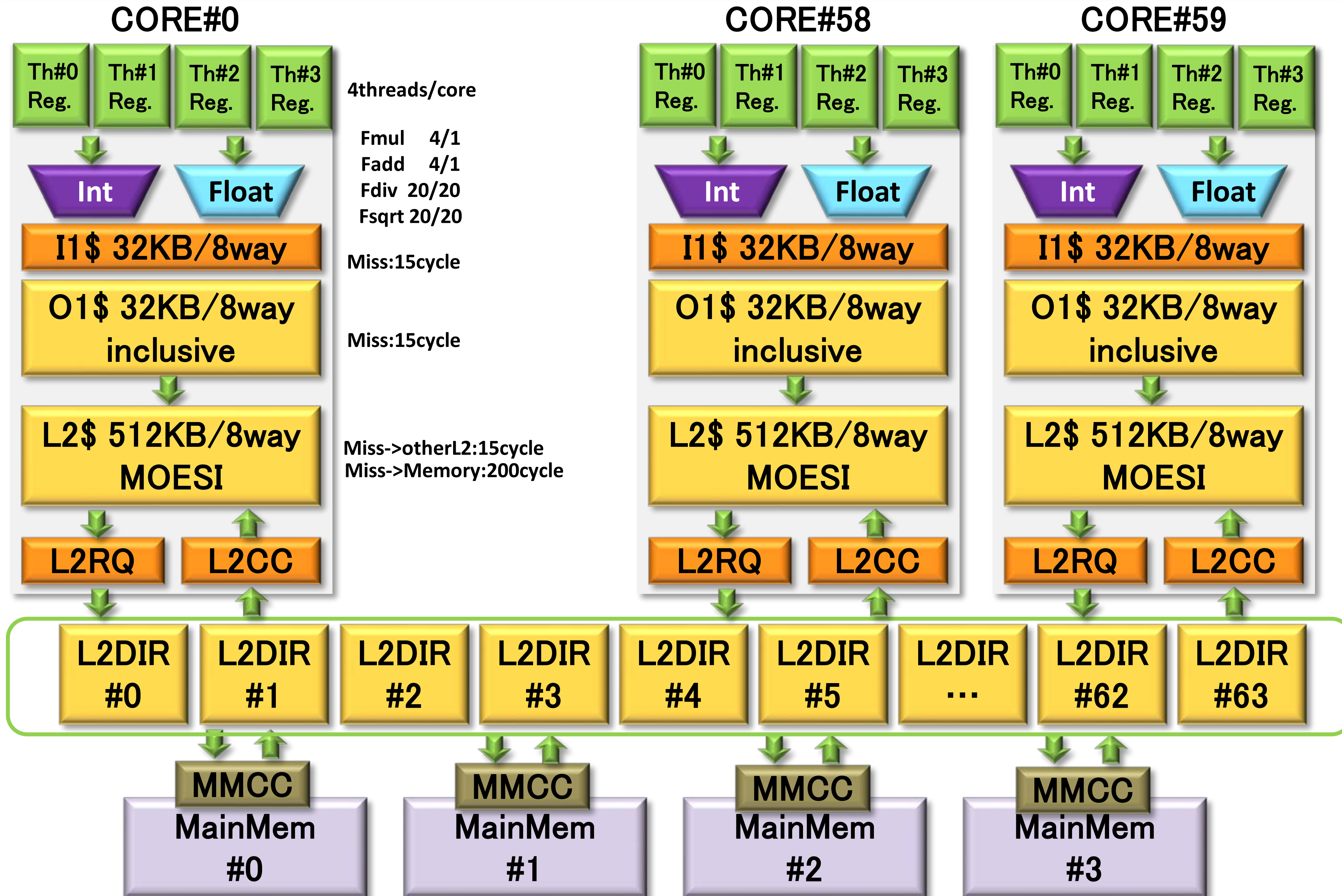
(1) Single core (2) Shared L2 (3) Banked L2



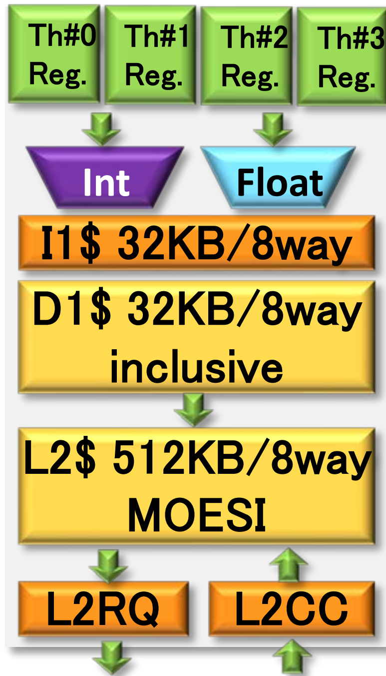
(4) Banked L2+Directory

NAIST

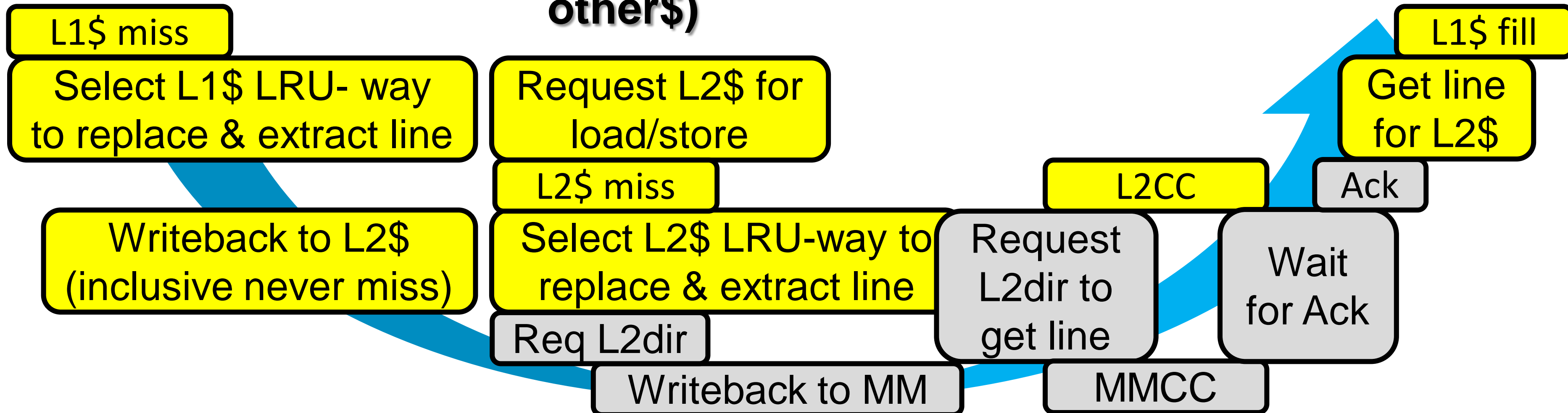
An Example (looks like Intel MIC)



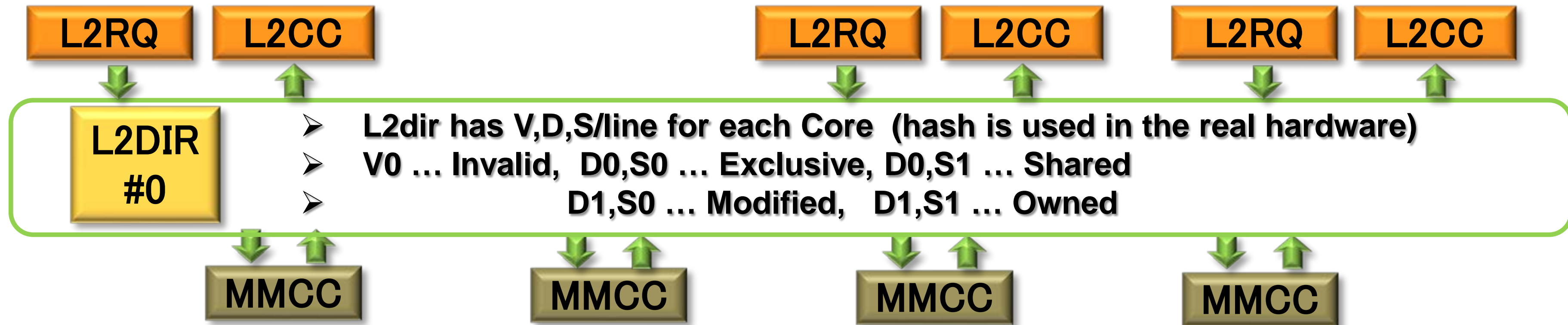
Sim_core.c



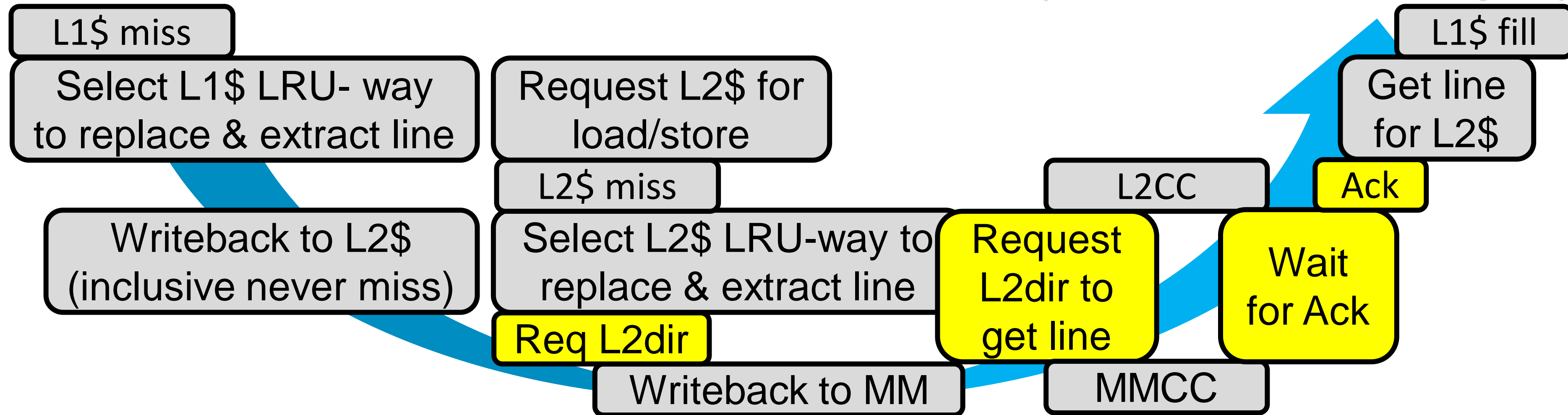
- I1$ is shared among thread and provides 1 instr./cycle (not 4 instr./cycle)
- D1$ and L2$ have Valid, Dirty, Shared bits/line
  - V0 … invalid -> miss
  - D0,S0 … load hit, store miss(update L2dir)
  - D1,S0 … load hit, store hit
  - D1,S1 … load hit, store miss(invalidate other$)
  - D0,S1 … load hit, store miss(invalidate other$)



Sim_mreq.c

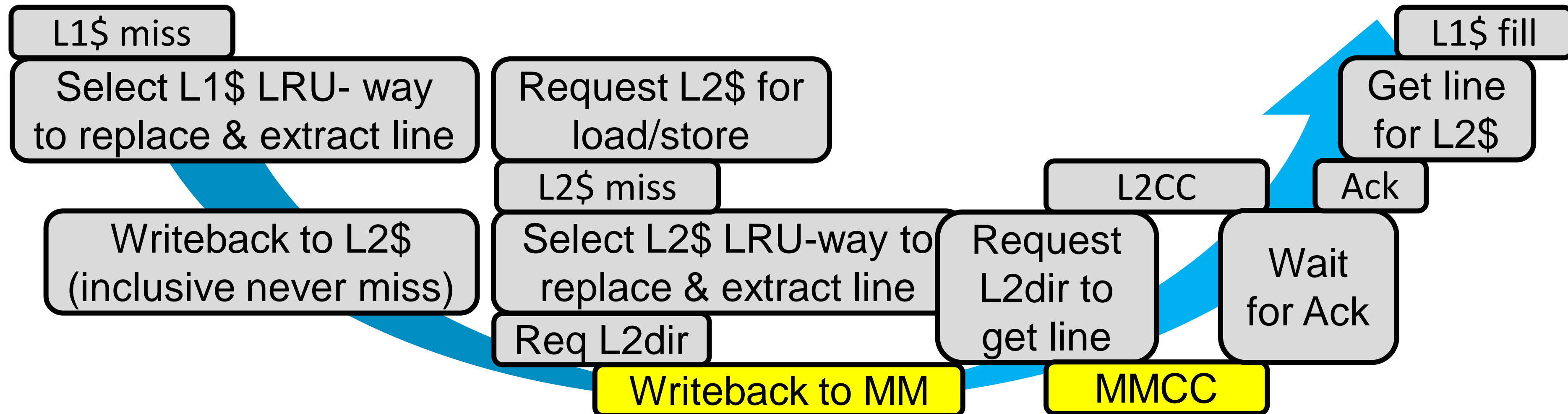


L2$ Invalidate Req:	Others: Exclusive/Shared > Invalid		Local: > Modified
MM Writeback Req:	Others: Exclusive/Shared > Invalid		Local: > Invalid
Read Req for load:	Home: > Owned	Others: no-change	Local: > Shared
	If no HOME, data should be loaded from MM (Shared can be invalidated anytime)		
Read Req for store:	Home: Copyback > Invalid	Others: > Invalid	Local: > Modified
	If no HOME, data should be loaded from MM (Shared can be invalidated anytime)		



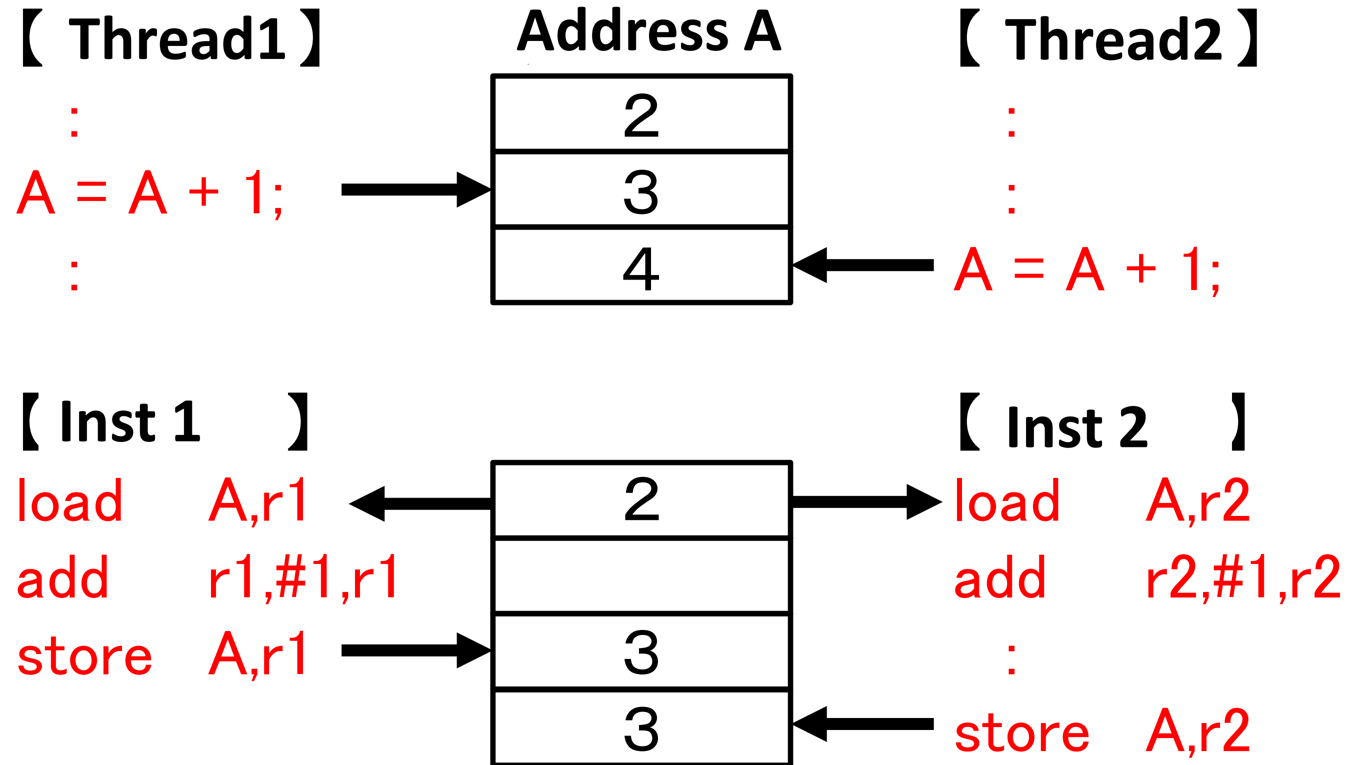
Sim_cluster.c

Simply gets the request from L2dir and sends ack to L2dir



Exclusive control

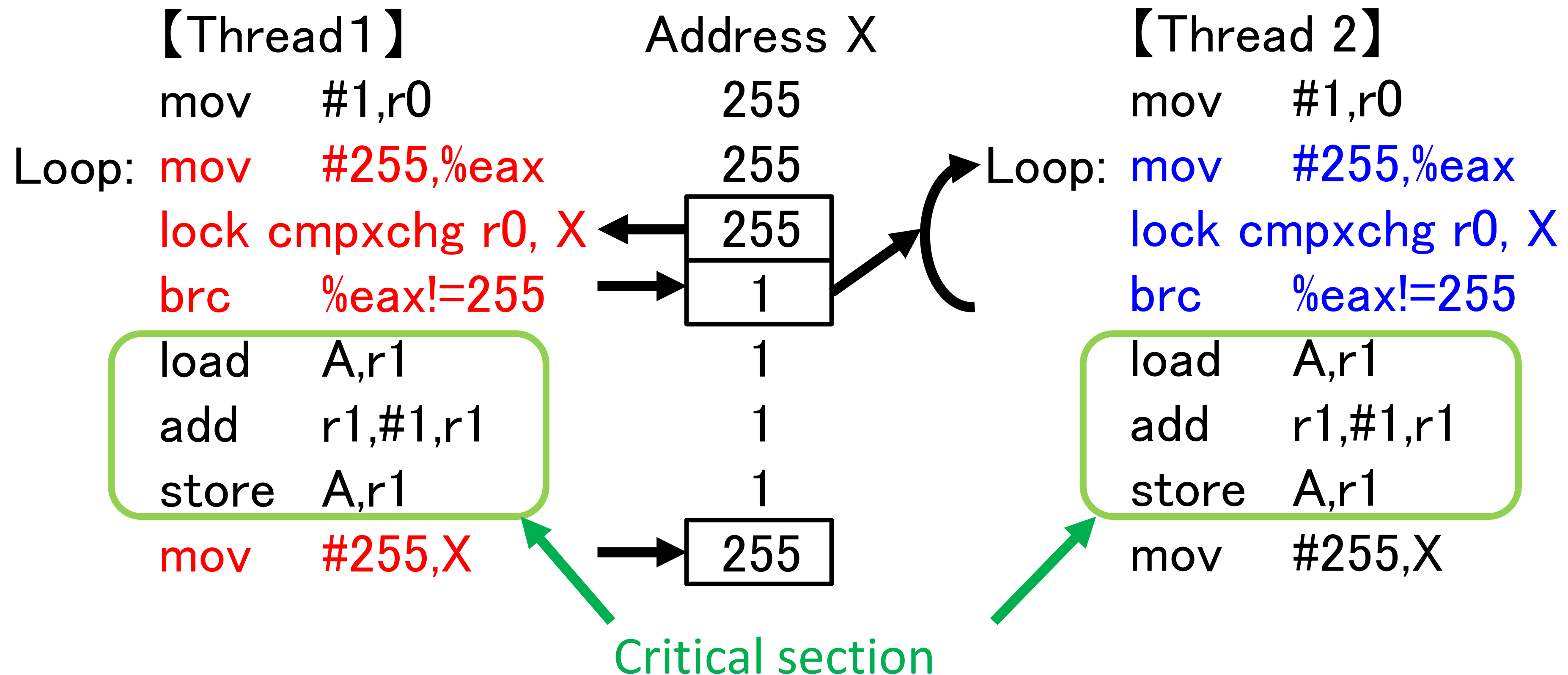
- Some threads update shared data
- How to manage it?



Even cache coherency is correct, result is wrong.
“Exclusive control” is required

What is Exclusive control

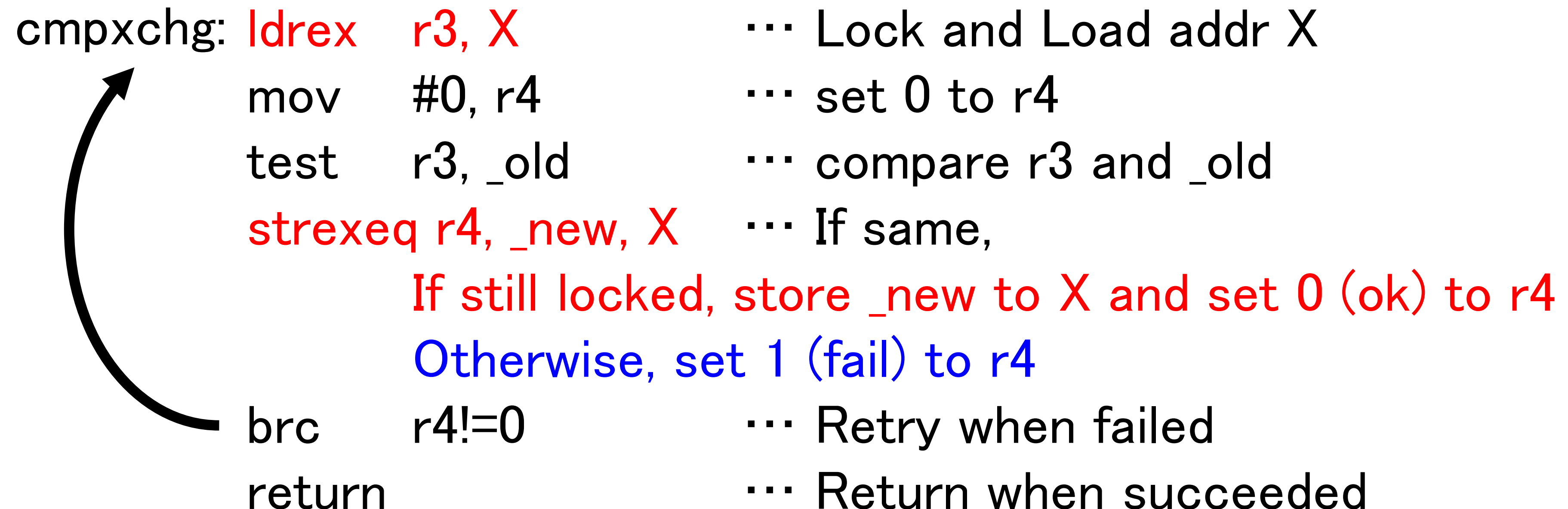
- **Intel uses** `cmpxchg reg, mem` **--- atomic instruction**
  When %eax = mem, store reg to mem
  When %eax != mem, load mem to %eax



What is Exclusive control

- **ARM uses** `ldrex` **and** `strex`
  `ldrex r0,X` **Lock addr X and load it to r0**
  -- Other thread may unlock addr X
  `strex r0,X` **If addr X is still locked, store r0 to X**
- **Bus lock is not required**

- cmpxchg (X, _old, _new) if X=_old, store _new to X



Exercise

Execute following 32 threads (tid=0,1,…,31)
Where does the difference of the execution speed come from?

```
#define SIZE 32768
double A[SIZE];
void *parallel(tid) int tid;
{
 int j, k;
 for (k=0; k<SIZE; k++) {
  for (j=0; j<SIZE; j++)
   A[tid*8] += k;
 }
}
```

4.3 Sec.

```
#define SIZE 32768
double A[SIZE];
void *parallel(tid) int tid;
{
 int j, k;
 for (k=0; k<SIZE; k++) {
  for (j=0; j<SIZE; j++)
   A[tid] += k;
 }
}
```

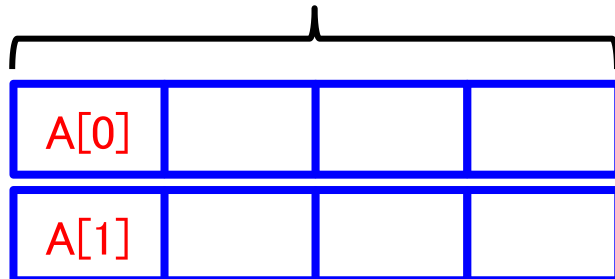
108 Sec.

Exercise

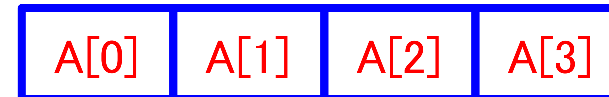

```
#define SIZE 32768
double A[SIZE];
void *parallel(tid) int tid;
{
 int j, k;
 for (k=0; k<SIZE; k++) {
  for (j=0; j<SIZE; j++)
   A[i*8] += k;
 }
}
```

```
#define SIZE 32768
double A[SIZE];
void *parallel(tid) int tid;
{
 int j, k;
 for (k=0; k<SIZE; k++) {
  for (j=0; j<SIZE; j++)
   A[i] += k;
 }
}
```

8byte×8=64byte
Located in different cache lines



Located in same cache line



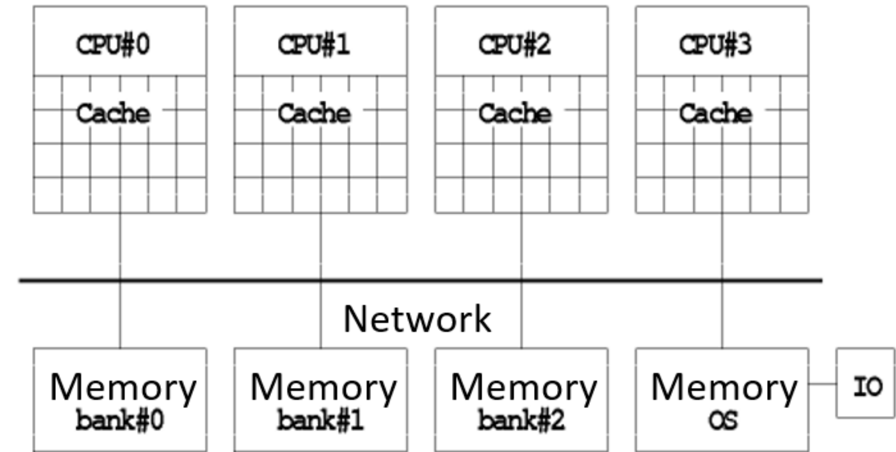
Distributed Memory System

Shared memory space or Distributed memory space

Single memory space (Shared)

- Multiple CPUs in one cabinet
- Multicore CPUs are also Shared

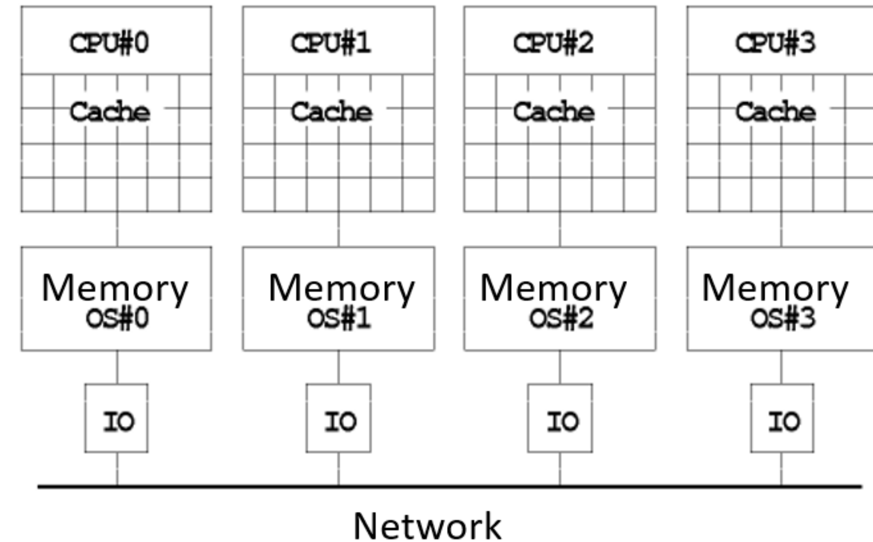
- Cache coherent management is required. Bad scalability but Good programmability



Multiple memory space (Distributed)

- Connect multiple PC/Server via network connection
- Super computer has also distributed memory space

- No cache coherent management Good scalability



Data transmission between different systems (OS, Memory)

Data transmission requires OS support

- **In shared memory systems, data access is always possible**
  No OS support is required
  - **Applications know address of shared data (available on shared memory)**
- **In distributed memory systems, data transmission is required to access other memories**
  - **Once data transmission start, hard to interrupt**
  - **Operating system dynamically manage virtual-physical memory mapping**
  - **Data transmission buffer should be kept while transmitting**

Data transmission between different systems (OS, Memory)

OS support

- Transmission buffer should be fixed physical address area
- Application send data via SEND() system call
- OS copies application data to the transmission buffer

  Then application data can be removed
- Communication driver transfers data from send buffer to receive buffer in receiver
- Communication driver in receiver node notifies OS of data arrival
- OS copies the transmission buffer to application data space

  Then application data can be removed
- Continue application (Complete RECV() or WAIT())

NAIST

Reduce OS overhead

Which operation can be reduced

- Application -> (System call) -> Send buffer
- Send buffer -> (Communication) -> Receive buffer
- Receive buffer -> (System call) -> Application

Optimal data transmission

- Application -> (Communication) -> Application

Fixed physical memory buffer in application is required

- When execution is started, application allocates fixed address space
- Can communicate directly

Summary

Processor always repeats specialization and generalization

Also repeats improving node performance and parallel computing (increasing number of nodes)

Improvement in technology repeats like spiral

Basic concepts are the same, but practical implementations may be different

It is important to recall traditional techniques!

That’s all for today