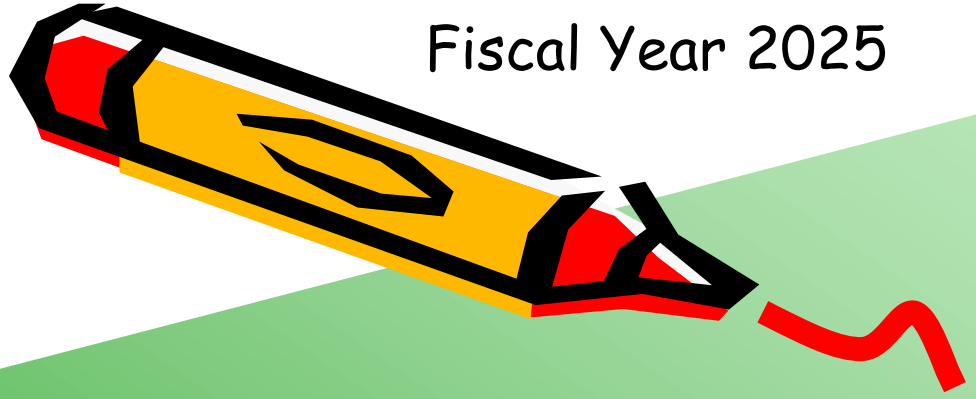


Fiscal Year 2025

Ver. 2026-01-29a

Course number: CSC.T440
School of Computing,
Graduate major in Computer Science

Computer Organization and Architecture

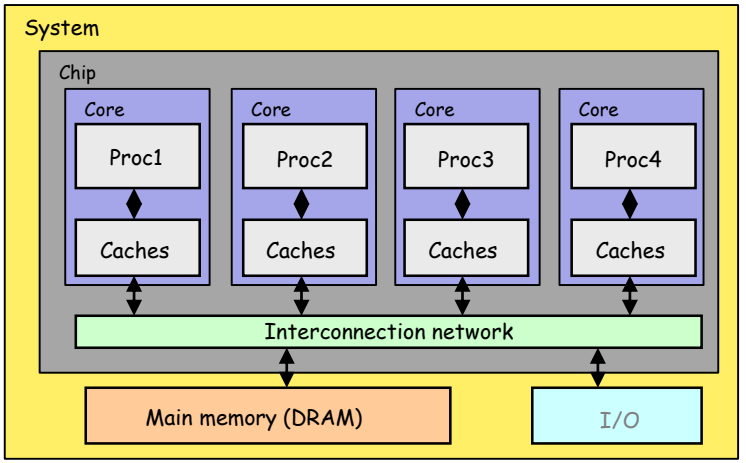
7. Thread Level Parallelism: Synchronization and Memory Consistency Model

www.arch.cs.titech.ac.jp/lecture/coa/
Room No. M-112(H117), Lecture (Face-to-face)
Thr 13:30-15:10

Kenji Kise, Department of Computer Science
kise[at]comp.isct.ac.jp

Key components of many-core processors

- Interconnection network
  - connecting many modules on a chip achieving high throughput and low latency
- 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
- Core
  - High-performance superscalar processor providing a hardware mechanism to support thread synchronization (lock, unlock, barrier)



Shared memory many-core architecture

Orchestration

- LOCK and UNLOCK around critical section
  - Lock provides exclusive access to the locked data.
  - 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 */

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_barrier_t barrier;
void solve_pp (int pid) {
    int i, done = 0;                    /* private variables */
    int mymin = (pid==0) ? 1 : 5;       /* private variable  */
    int mymax = (pid==0) ? 4 : 8;       /* 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]);
        }
        pthread_mutex_lock(&m);
        diff = diff + mydiff;
        pthread_mutex_unlock(&m);

        pthread_barrier_wait(&barrier);  // Barrier 1
        if (diff <TOL) done = 1;
        pthread_barrier_wait(&barrier);  // Barrier 2
        if (pid==1) diff = 0.0;
        for (i=mymin; i<=mymax; i++) A[i] = B[i];
        pthread_barrier_wait(&barrier);  // Barrier 3
    }
}
```

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.

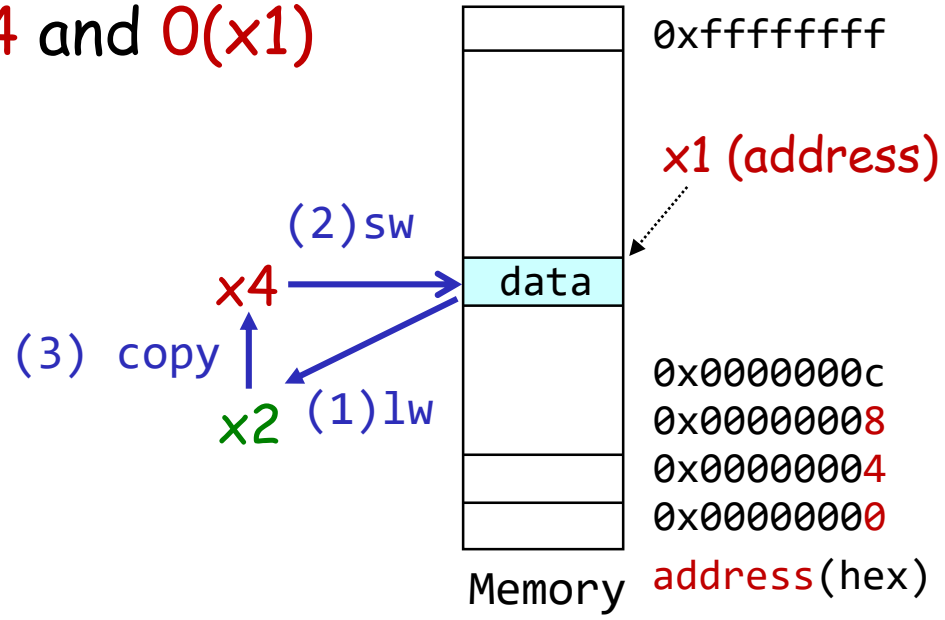
```
if (diff <TOL) done = 1;
```

Synchronization

- Basic building blocks (instructions) :
  - 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 reserved (load linked) / store conditional
    - If the contents of the memory location specified by the load reserved are changed before the store conditional to the same address, the store conditional fails

Implementing an exchange EXCH

- EXCH x4, 0(x1) ; exchange x4 and 0(x1)
  - Why isn't this code atomic?



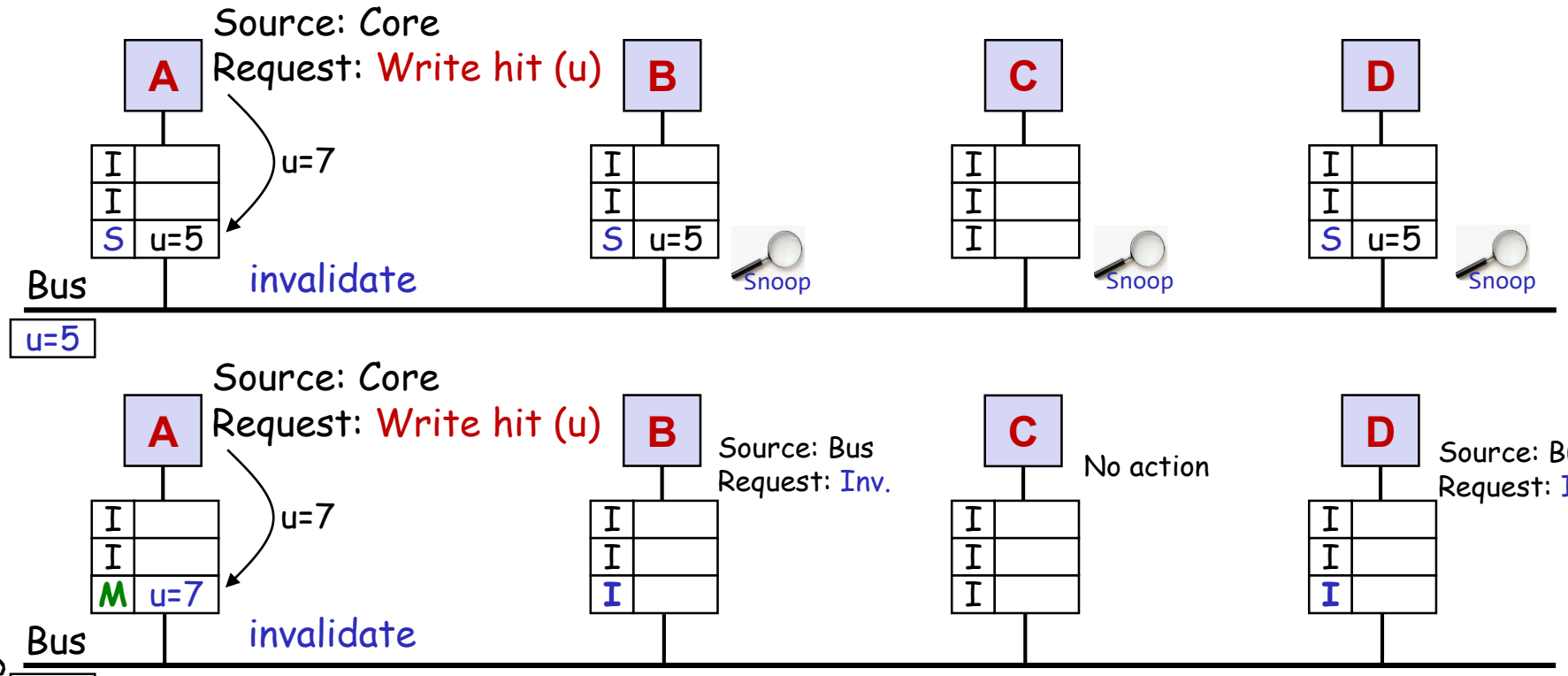

```
(1) lw  x2,0(x1)     # load  word,  Tmp <- shared data
(2) sw  x4,0(x1)     # store word,  x4  -> shared data
(3) add x4,x2,x0     # copy,        x4  <- Tmp
```

Timer interrupt, cache coherence protocol

Coherence 1 (Coh1) and Coherence3 (Coh3)

- Coh1 (Core A)
  - Source: Core
  - State: Shared
  - Request: Write hit (u)
  - Function: Place invalidate on bus
- Coh3 (Core B, D)
  - Source: Bus
  - State: Shared
  - Request: Invalidate
  - Function: attempt to write shared block; invalidate the block

Event:
Core A writes u



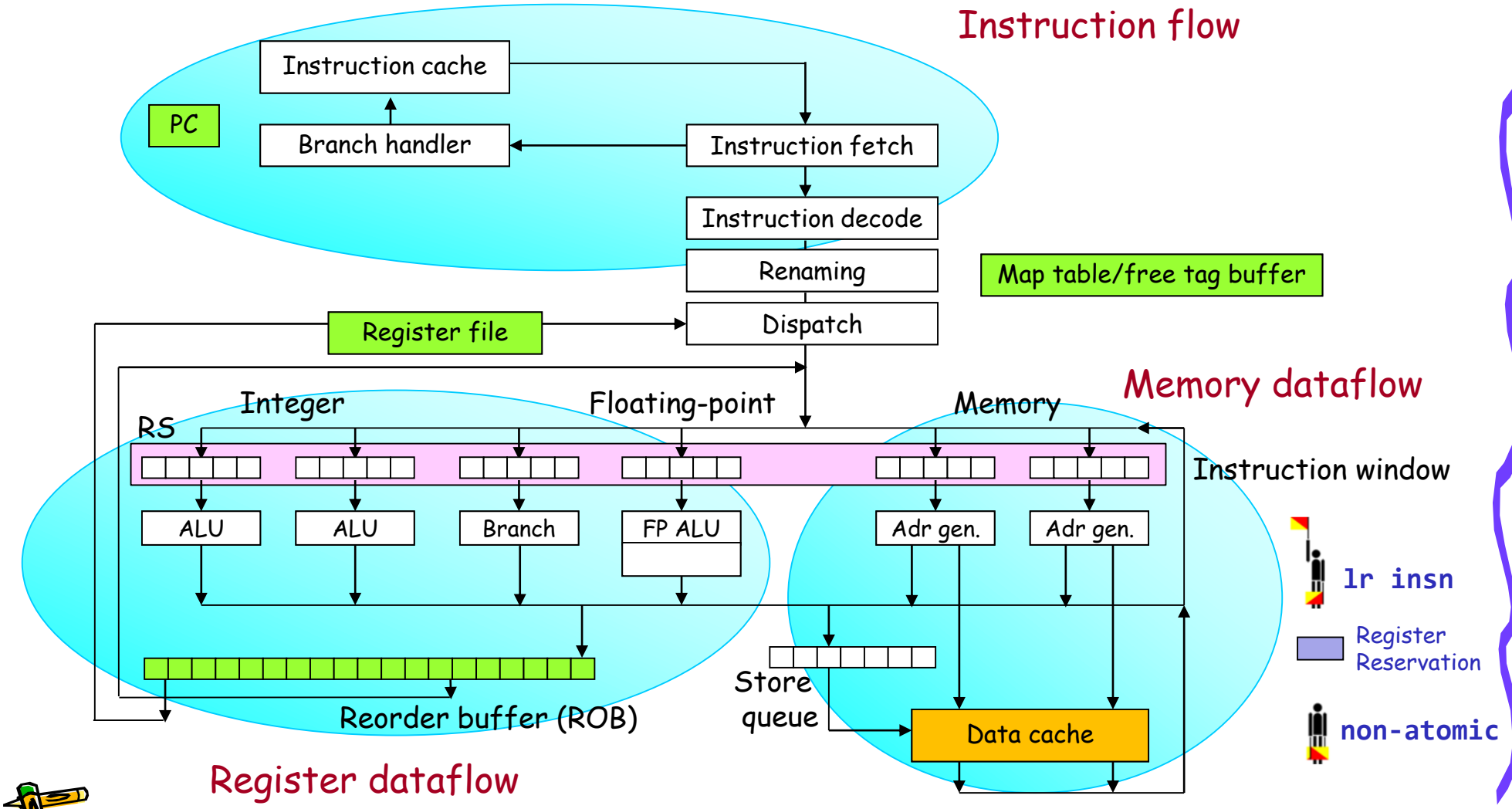
Implementing an atomic exchange EXCH

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

- EXCH x4,0(x1) ; exchange x4 and 0(x1) atomically

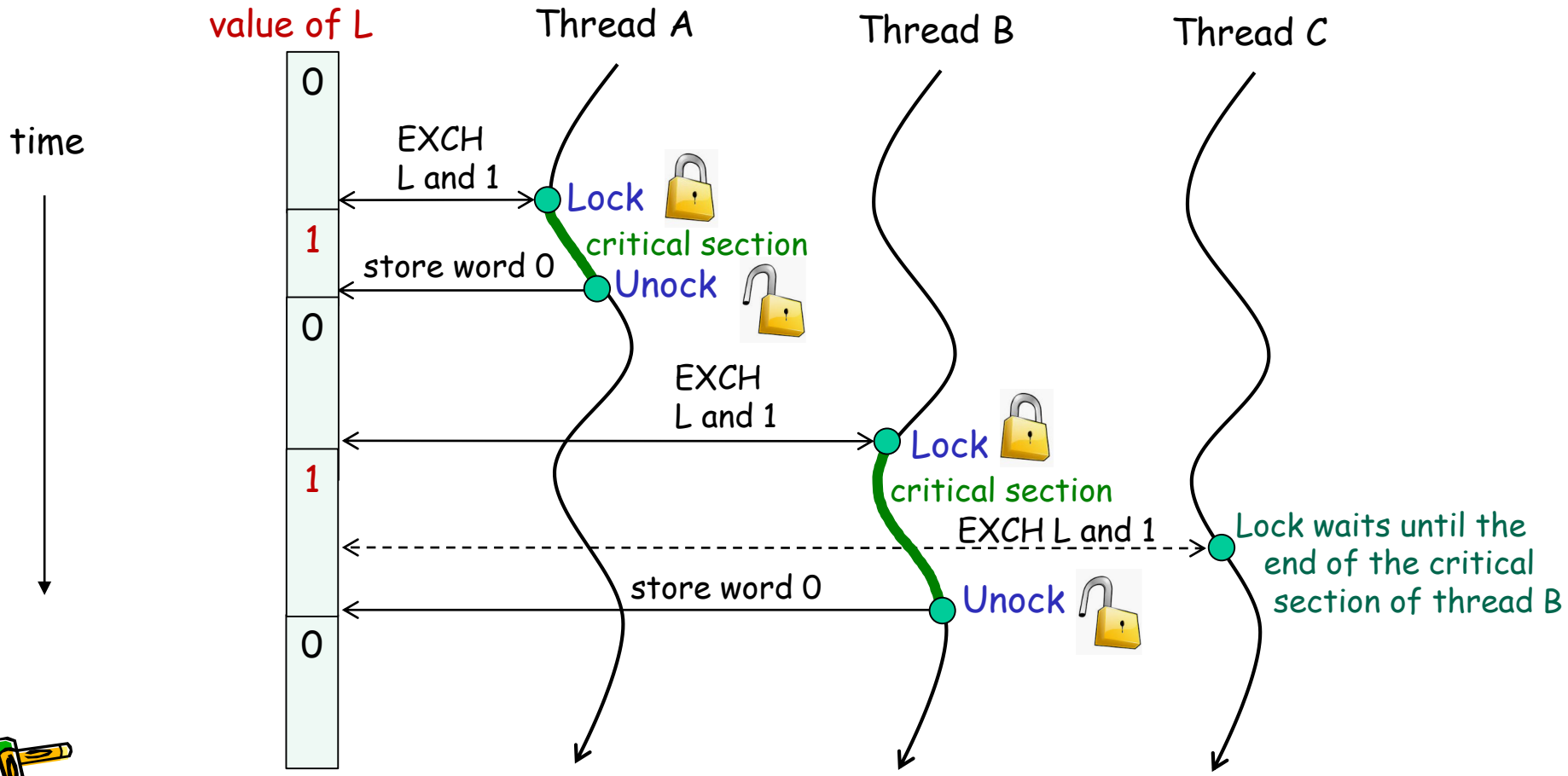
```
try:    add  x3,x4,x0      # move exchange value, x3<=x4
        lr.w x2,0(x1)      # load reserved word
        sc.w x3,0(x1)      # store conditional word
        beq  x3,x0,try     # branch if store fails (x3==0)
        add  x4,x2,x0      # put load value in x4, x4<=x2
```

Datapath of OoO execution processor (core)



Implementing Lock with a lock variable

- LOCK and UNLOCK around critical section
  - Lock provides exclusive access to the set of operations we want to execute atomically
- lock variable L whose initial value is 0



Implementing Lock (simple version)

- Spin lock version 1.0
  - x1 is the address of the lock variable (shared variable) and its initial value is 0 (not locked).

```
lock:     addi   x4,x0,1        # x4 <= 1
lockit:   EXCH   x4,0(x1)       # atomic exchange
          bne    x4,x0,lockit   # already locked?
```

- EXCH x4,0(x1) ; exchange x4 and 0(x1) atomically

```
try:    add  x3,x4,x0     # move exchange value, x3<=x4
        lr.w x2,0(x1)     # load reserved word
        sc.w x3,0(x1)     # store conditional word
        beq  x3,x0,try    # branch if store fails (x3==0)
        add  x4,x2,x0     # put load value in x4, x4<=x2
```

Implementing Lock using coherence

- Spin lock version 2.0
  - x1 is the address of the lock variable and its initial value is 0.
  - We can cache the lock using the cache 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).
  - This reduces the number of executions of expensive store instructions.

```
lock:   ld     x4,0(x1)      # load of lock
        bne    x4,x0,lock    # not available-spin if x4==1
        addi   x4,x0,1       # set locked value, x4<=1
        EXCH   x4,0(x1)      # swap
        bne    x4,x0,lock    # branch if lock wasn't 0
```

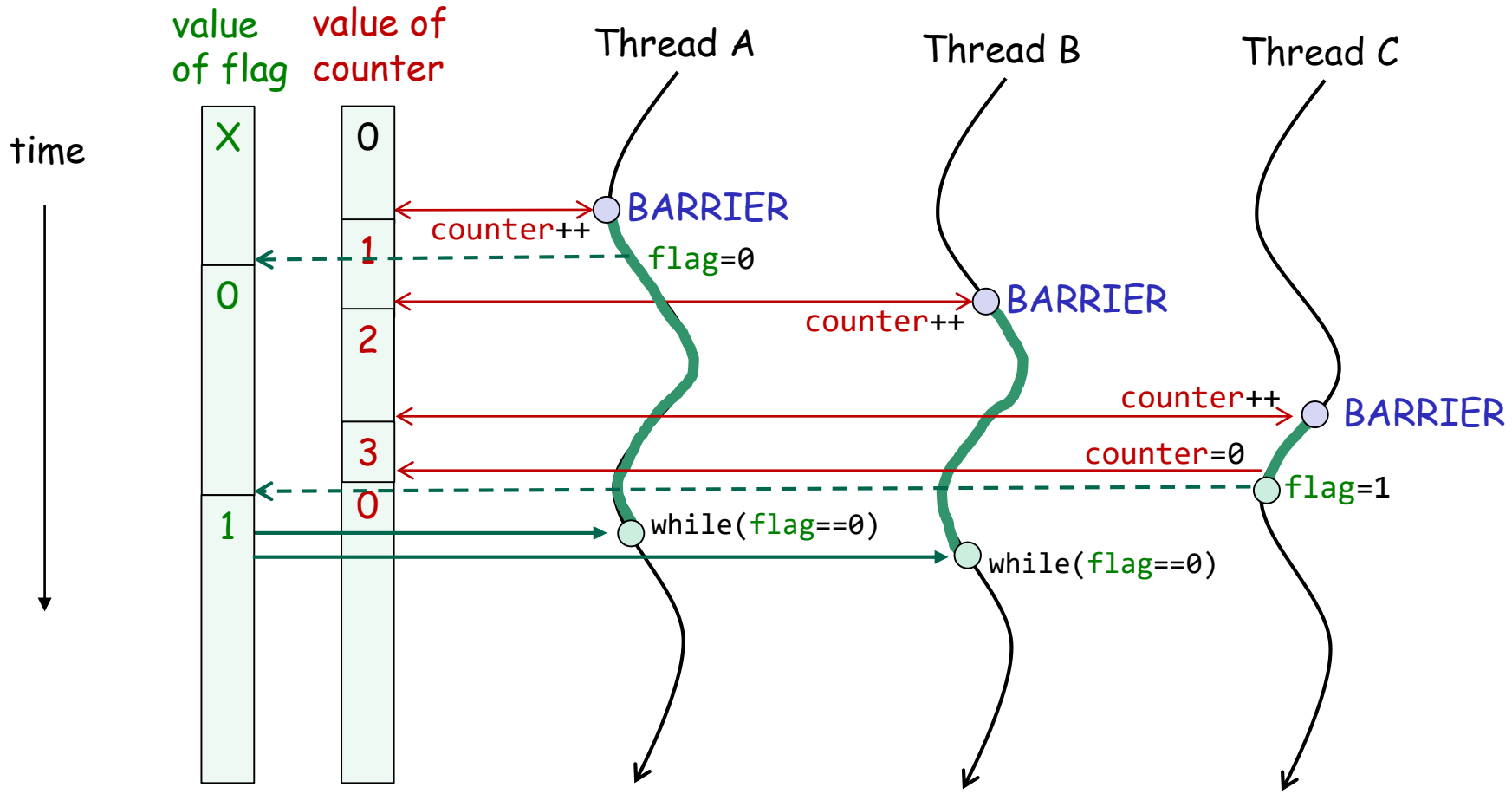
Implementing Unlock using coherence

- Unlock
  - x1 is the address of the lock variable and its initial value is 0 (not locked).
  - Just resetting the lock variable x1

```
unlock:    sw  x0, 0(x1)  # reset the lock, lock_variable <= 0
```

Implementing BARRIER with two shared variables

- shared variable counter and flag having initial values of 0
- counts up the arrived threads using a shared variable counter
- the last thread set the shared variable flag to exit the barrier



Exercise 1

- Implementing BARRIER using coherence
  - This code counts up the arrived threads using a shared variable counter.
  - All threads increments the variable, and the last thread set the shared variable flag to exit the barrier.
  - Lock() and Unlock() are the functions defined earlier.

```
int counter = 0;
int flag = 0;
int cores = 4; /* the number of cores */

BARRIER(){


   Lock();


   Unlock();


}
```

Implementing BARRIER using coherence

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

```
int counter = 0;
int flag = 0;
int cores = 4; /* the number of cores */

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

Implementing BARRIER using coherence

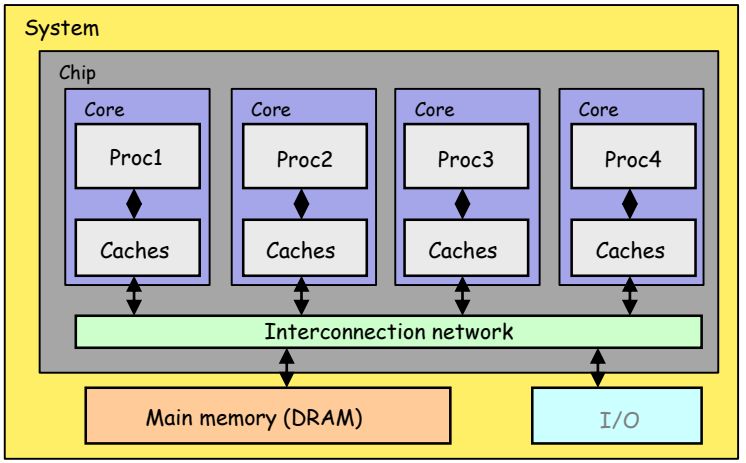
- This code counts up the arrived threads using a shared variable counter.
- All threads increments the variable, and the last thread set the shared variable flag to exit the barrier.

```
int counter = 0;
int flag = 0;
int cores = 4; /* the number of cores */

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

Key components of many-core processors

- Interconnection network
  - connecting many modules on a chip achieving high throughput and low latency
- 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
- Core
  - High-performance superscalar processor providing a hardware mechanism to support thread synchronization (lock, unlock, barrier)



Shared memory many-core architecture

Memory consistency: problem in multi-core context

- Assume that shared data A=0 and Flag=0 initially
- Core 1 writes data into A and sets Flag to tell Core 2 that data value can be read (loaded) from A.
- Core 2 waits till Flag is set and then reads (loads) data from A.
- What is the printed value by Core 2?

Core 1	Core 2
`A = 3;`	`while (Flag==0);`
`Flag = 1;`	`print A;`

Problem in multi-core context

- If the two writes (stores) of different addresses on Core 1 can be reordered, it is possible for Core 2 to read 0 from variable A.
- This can happen on most modern processors.
  - For single-core, Source code(1) and Source code(2) are equivalent. These writes may be reordered by compilers statically or by OoO execution units dynamically.
  - The printed value by Core 2 will be 0 or 3.

Source code(1)

```
A = 3;
Flag = 1;
```

Source code(2)

```
Flag = 1;
A = 3;
```

Core 1	Core 2
(1) Flag = 1;	(2) while (Flag==0);
(4) A = 3;	(3) print A;

Assume that A=0 and Flag=0 initially

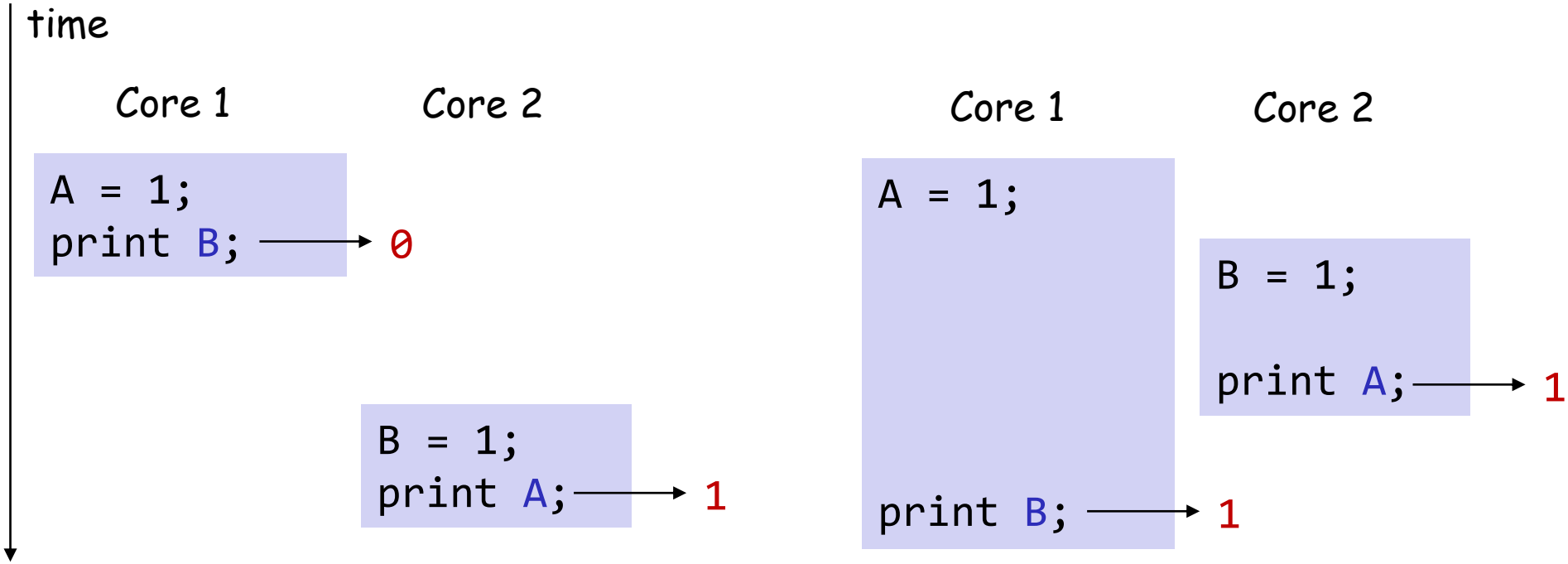
Problem in multi-core context

- Assume that A=0 and B=0 initially
- Should be impossible for both outputs to be zero.
  - Intuitively, the outputs may be 01, 10, and 11.

C1 (Core 1)	C2 (Core 2)
`A = 1;`	`B = 1;`
`print B;`	`print A;`

Example behaviours

- Assume that A=0 and B=0 initially



The outputs are 01.

The outputs are 11.

Problem in multi-core context

- Assume that A=0 and B=0 initially
- Should be impossible for both outputs to be zero.
  - Intuitively, the outputs may be 01, 10, and 11.
  - This is true only if reads and writes on the same core to different locations are not reordered by the compiler or the hardware.
  - The outputs may be 01, 10, 11, and 00.

Core 1	Core 2
`A = 1;`	`B = 1;`
`print B;`	`print A;`

Memory consistency models

- A single-core processor can reorder instructions subject only to control and data dependence constraints
- These constraints are not sufficient in shared-memory multi-cores
  - simple parallel programs may produce counter-intuitive results

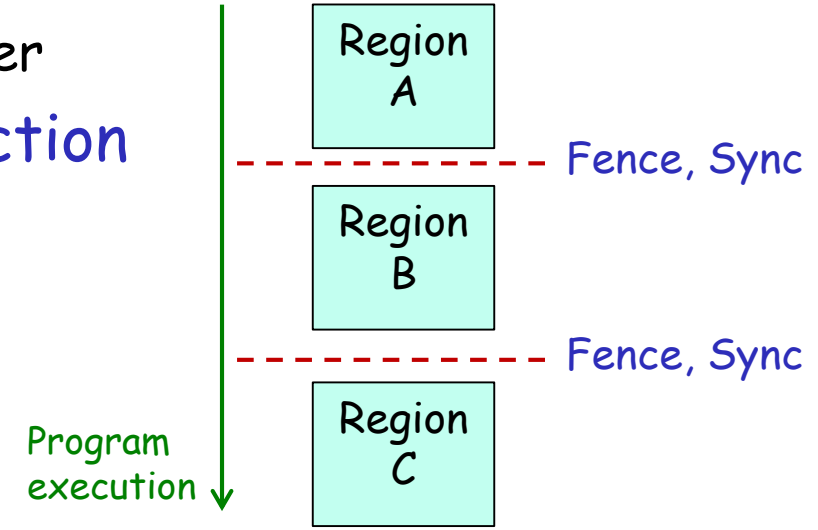
- Question: what constraints must we put on single-core instruction reordering so that
  - shared-memory programming is intuitive
  - but we do not lose single-core performance?
- The answers are called memory consistency models supported by the processor
  - Memory consistency models are all about ordering constraints on independent memory operations in a single-core's instruction stream

Simple and intuitive model: sequential consistency

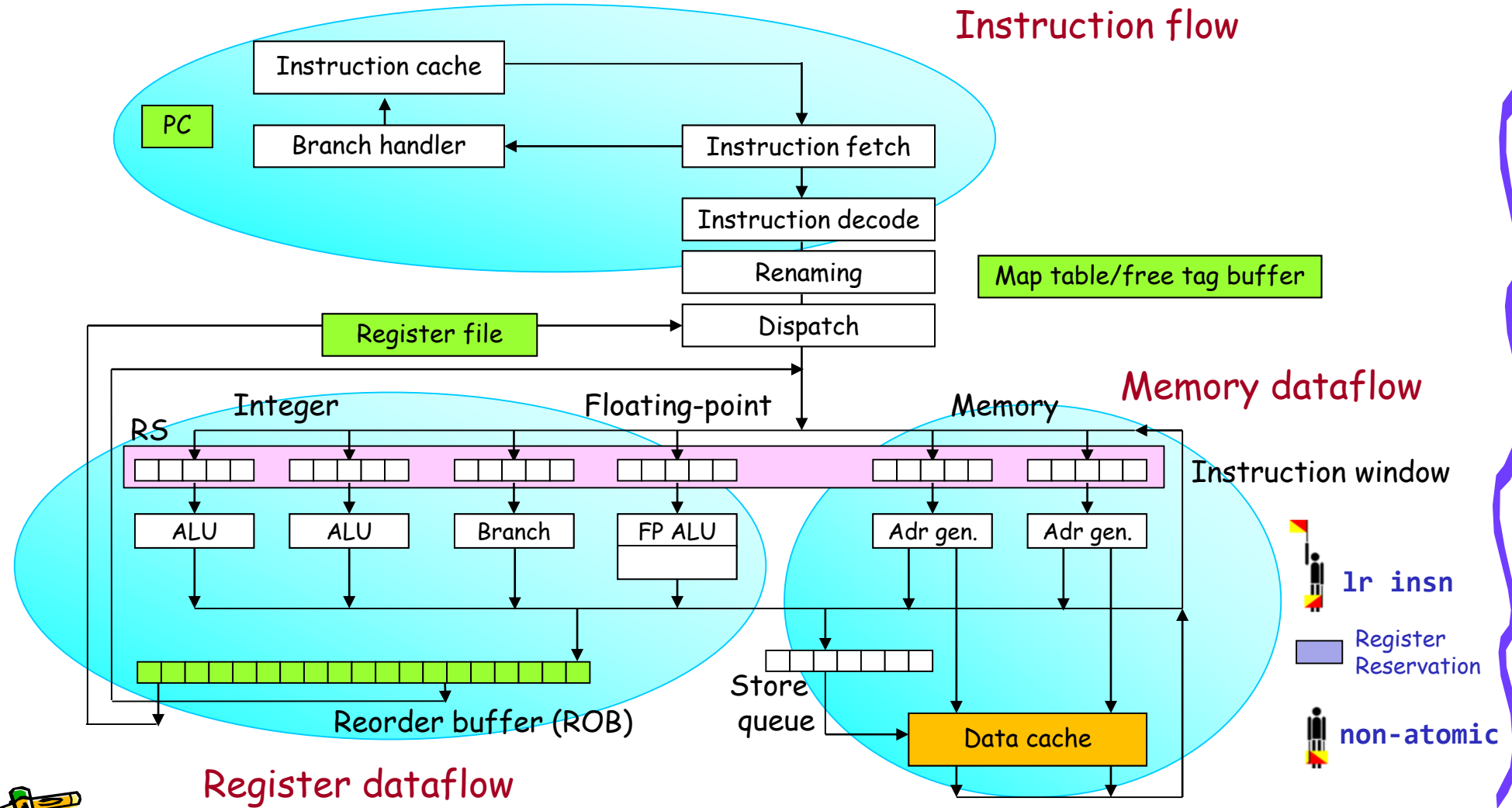
- Sequential consistency (SC) model
  - It constrains all memory operations:
    - Write -> Read
    - Write -> Write
    - Read -> Read
    - Read -> Write
  - Simple model for reasoning about parallel programs
    - You can verify that the examples considered earlier work correctly under sequential consistency.
  - This simplicity comes at the cost of single-core performance.
    - How to implement SC ?
  - How do we modify sequential consistency model with the demands of performance?

Relaxed consistency model: weak consistency

- Programmer specifies regions within which global memory operations can be reordered
- Processor has fence (or sync) instruction:
  - all data operations before fence in program order must complete before fence is executed
  - all data operations after fence in program order must wait for fence to complete
  - fences are performed in program order
- Example: RISC-V has fence instruction
- Implementation of fence
  - a processor may flush all instructions when a fence instruction is retired

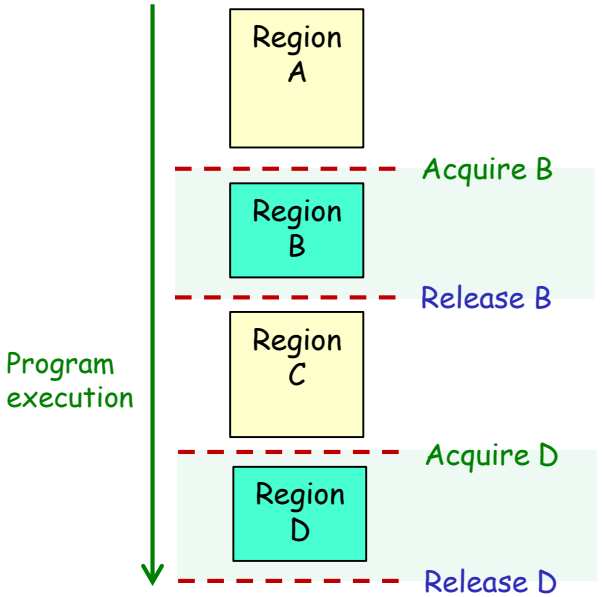


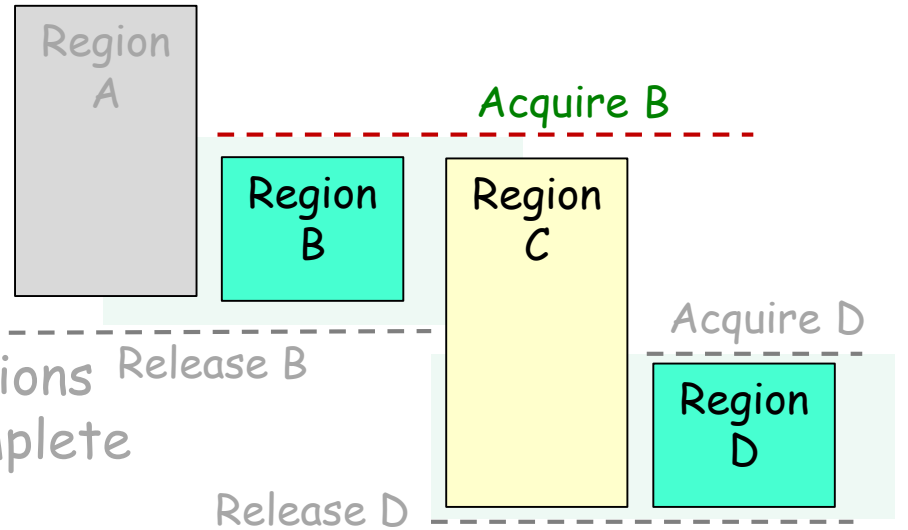
Datapath of OoO execution processor (core)



Release consistency model

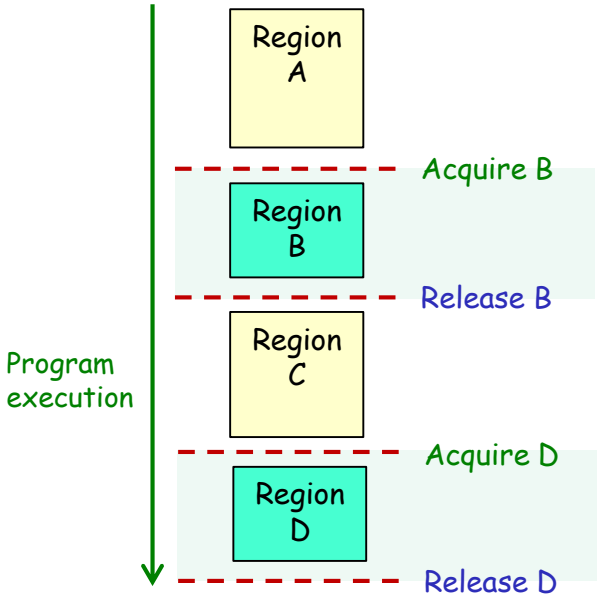
- Further relaxation of weak consistency
- A fence instruction is divided into
  - Acquire: operation like lock
  - Release: operation like unlock
- Semantics of Acquire:
  - Acquire must complete before all following memory accesses
  - After Acquire B, memory operations in region B, C and D must complete
- Semantics of Release:
  - all memory operations before Release must complete before the Release
  - Before Release B, memory operations in region A and region B must complete

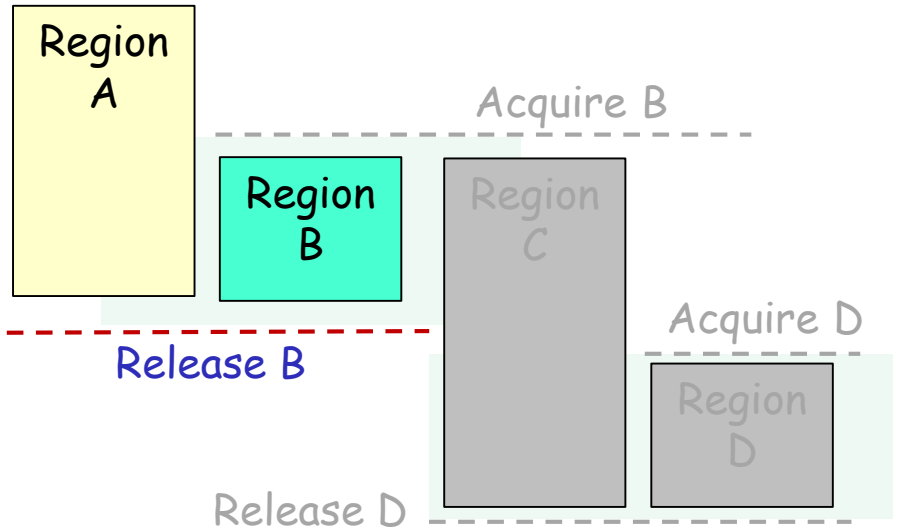




Release consistency model

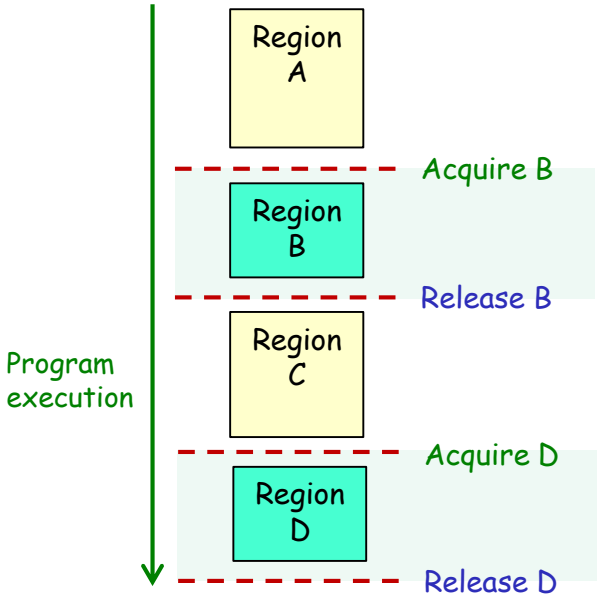
- Further relaxation of weak consistency
- A fence instruction is divided into
  - Acquire: operation like lock
  - Release: operation like unlock
- Semantics of Acquire:
  - Acquire must complete before all following memory accesses
  - After Acquire B, memory operations in region B, C and D must complete
- Semantics of Release:
  - all memory operations before Release must complete before the Release
  - Before Release B, memory operations in region A and region B must complete

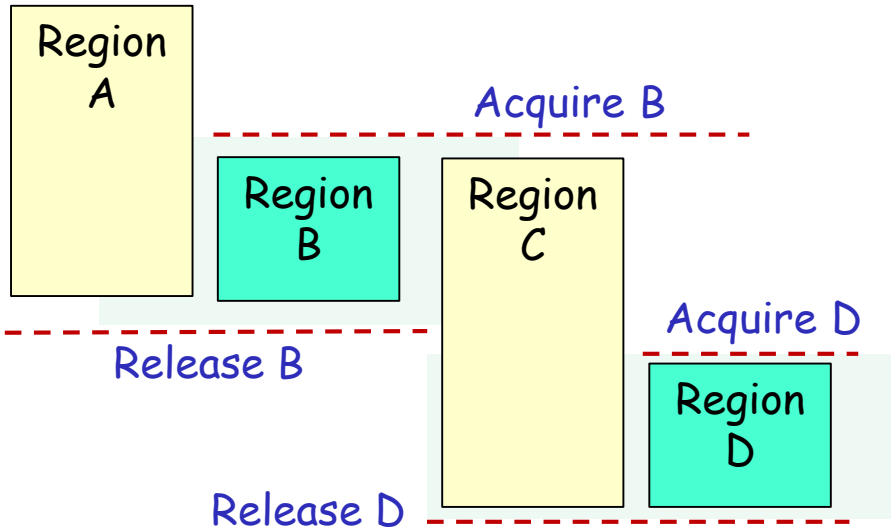




Release consistency model

- Further relaxation of weak consistency
- A fence instruction is divided into
  - Acquire: operation like lock
  - Release: operation like unlock
- Semantics of Acquire:
  - Acquire must complete before all following memory accesses
  - After Acquire B, memory operations in region B, C and D must complete
- Semantics of Release:
  - all memory operations before Release must complete before the Release
  - Before Release B, memory operations in region A and region B must complete



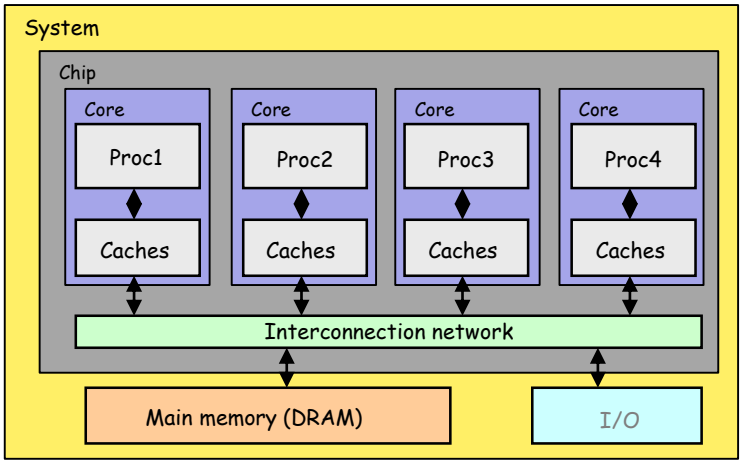


Memory Consistency Model

- In the literature, there are a large number of other consistency models
  - Sequential consistency
  - Causal consistency
  - Processor consistency
  - Weak consistency (weak ordering)
  - Release consistency
  - Entry consistency
  - …
- It is important to remember that these are concerned with reordering of independent memory operations within a single thread.
- Weak or Release Consistency Models are adequate

Key components of many-core processors

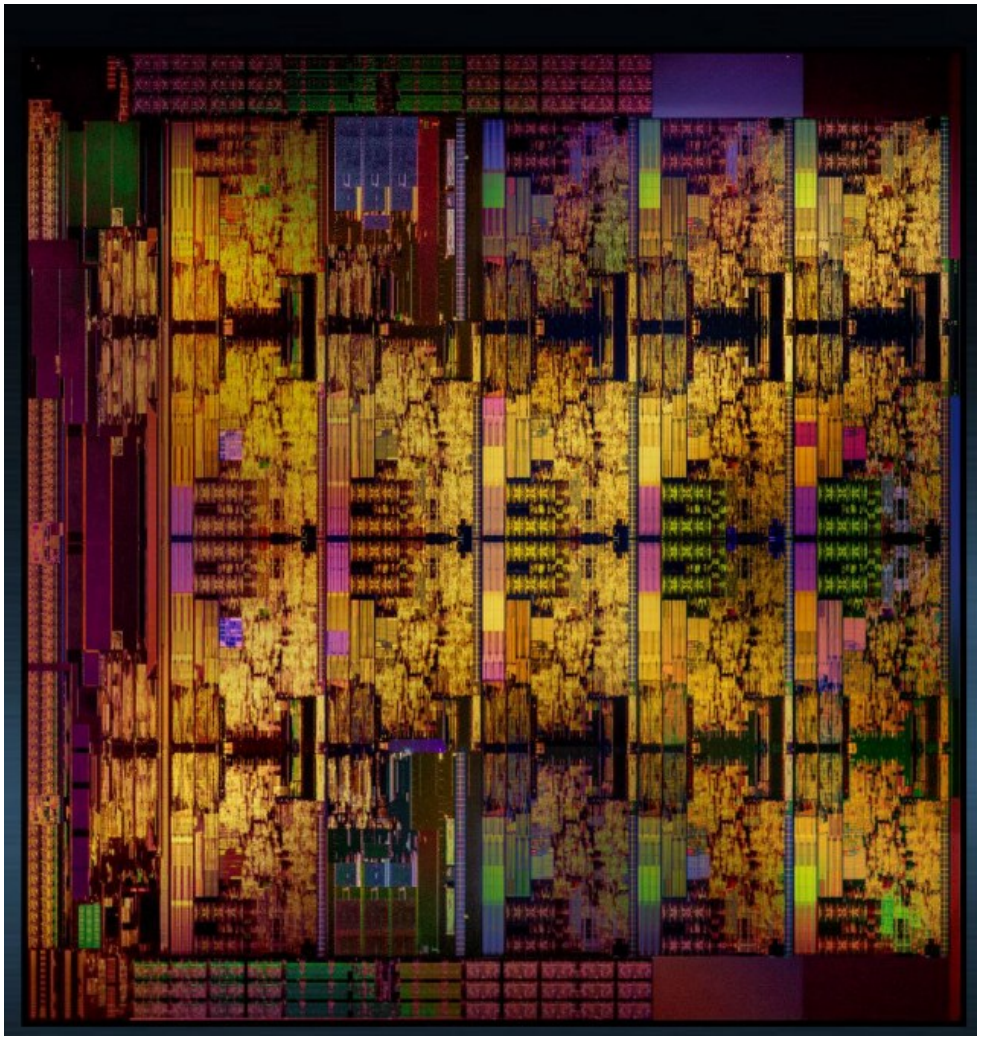
- Interconnection network
  - connecting many modules on a chip achieving high throughput and low latency
- Main memory and caches
  - Caches are used to reduce latency and to lower network traffic
  - A parallel program has private data and shared data
  - New issues are cache coherence and memory consistency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



Shared memory many-core architecture

Putting It All Together

- 18 core
- 2D mesh topology



Final report of Computer Organization and Architecture

1. Please submit your final report describing your answers to all questions in a PDF file
   via E-mail (kise[at]comp.isct.ac.jp) by February 9, 2026
   - E-mail title must be "Final Report of Computer Organization and Architecture"
2. Please submit the report in 10 pages or less on A4 size PDF file, including the cover page.

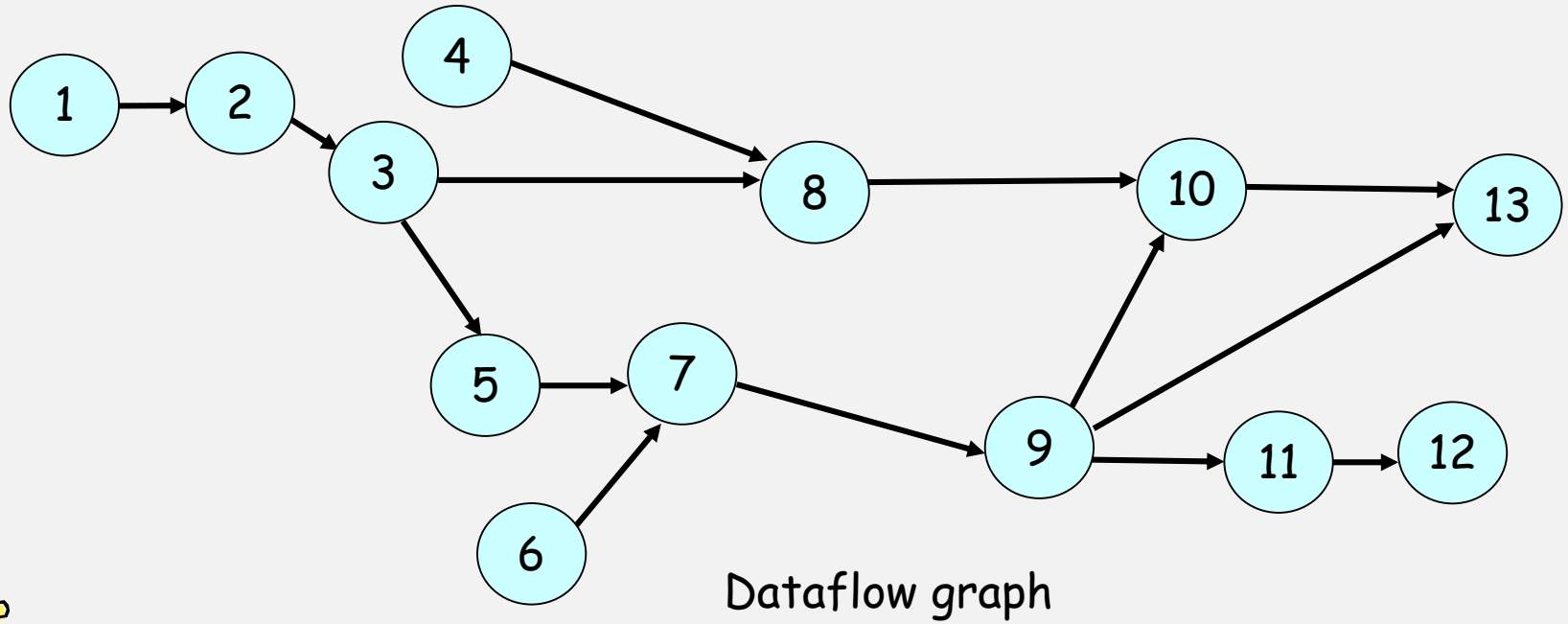
3. You can discuss it with your colleague, but try to solve the questions yourself. Enjoy!

1. Academic paper reading

- Select an academic paper from the list below and
  - In your own word, describe the problem that the authors try to solve,
  - Describe the key idea of the proposal,
  - Describe your opinion why the authors could solve the problem although there may be many researchers try to solve similar problems.
  - Choose the figure you consider to be the most important in the paper and explain why.
- List
  - Increasing Processor Performance by Implementing Deeper Pipelines, ISCA,2002
  - Combining Branch Predictors, WRL TN, 1993
  - Dynamic branch prediction with perceptrons, HPCA, 2001
  - Prophet/critic hybrid branch prediction, ISCA, 2004
  - Clockhands: Rename-free Instruction Set Architecture for Out-of-order Processors, MICRO, 2023
  - Focused Value Prediction, ISCA, 2020
  - Emulating Optimal Replacement with a Shepherd Cache, MICRO, 2008

2. OoO execution and dynamic scheduling

- Draw the cycle by cycle processing behavior of these 13 instructions
- Modify this dataflow graph by removing two edges of the graph so that the number of execution cycles is reduced. Draw another cycle by cycle processing behavior of the modified graph.



Dataflow graph


Cycle 1
Instruction window
Issue
Execute
Commit
Retire
ROB
Cycle 2
Instruction window
Issue
Execute
Commit
Retire
ROB
Cycle 3
Instruction window
Issue
Execute
Commit
Retire
ROB
Cycle 4
Instruction window
Issue
Execute
Commit
Retire
ROB
Cycle 5
Instruction window
Issue
Execute
Commit
Retire
ROB
Cycle 6
Instruction window
Issue
Execute
Commit
Retire
ROB
Cycle 7
Instruction window
Issue
Execute
Commit
Retire
ROB
Cycle 8
Instruction window
Issue
Execute
Commit
Retire
ROB
Cycle 9
Instruction window
Issue
Execute
Commit
Retire
ROB
Cycle 10
Instruction window
Issue
Execute
Commit
Retire
ROB

3. Parallel programming

- Adjust the number of elements N so that this sequential program (main8.c) takes about 5 second. Use this adjusted value for N. Use the `time` command to measure the execution time.
  Please use –O0 optimization flag while compiling.
- Describe an efficient parallel program for the sequential program of main8.c using LOCK, UNLOCK, and BARRIER of pthread assuming a shared memory architecture of 4 cores.
- Explain why your code runs correctly and why your code is efficient.
- Show your speedup over the sequential execution. To measure the execution time, use a computer with four or more cores.

main8.c

```
#include <stdio.h>
#include <math.h>
#define N   30   /* the number of grids */
#define TOL 15.0 /* tolerance parameter */
float A[N+2], B[N+2];
float diff;

void solve () {
    int i, done = 0;
    while (!done) {
        diff = 0;
        for (i=1; i<=N; i++) {
            B[i] = 0.333 * (A[i-1] + A[i] + A[i+1]);
            diff = diff + fabsf(B[i] - A[i]);
        }
        if (diff <TOL) done = 1;
//        for (i=0; i<=N+1; i++) printf(" %6.2f", A[i]);
//        printf("\n");
        for (i=1; i<=N; i++) A[i] = B[i];
  }
}

int main() {
    int i;
    for (i=1; i<=N; i++) A[i] = 100+i*2;
    A[0] = 90;
    A[N+1] = 50+N*2;

    solve();
    for (i=N/2-10; i<=N/2; i++) printf(" %6.3f", B[i]);
    printf("\n diff=%6.3f\n", diff);
}
```

4. Building blocks for synchronization

- Implement your BARRIER() using some global variables, pthred lock, and unlock.
- Show your code and explain why your code runs correctly and why your code is efficient.

- Replace the barrier in the program in Question 3 with your designed one and measure the speedup over the sequential program.

5. Cache coherence protocols

- Select your favorite commercial Intel or AMD multi-core processor with more than three cores shipped after 2020
  - Describe the memory organization, including caches and main memory
    - cache line size, write policy, write allocate/no-allocate, direct-mapped/set-associative, the number of caches (L1, L2, and L3?)
  - Describe the cache coherence protocol used there