Fiscal Year 2022

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

Advanced Computer Architecture

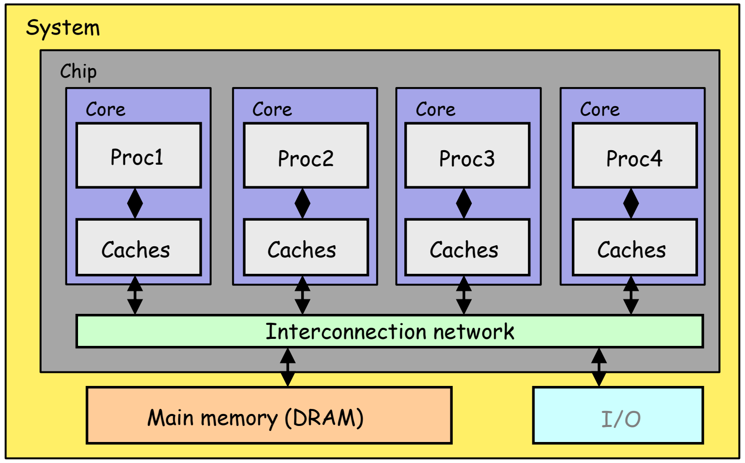
13. Thread Level Parallelism: Memory Consistency Model

www.arch.cs.titech.ac.jp/lecture/ACA/
Room No.W831, HyFlex
Mon 13:45-15:25, Thr 13:45-15:25

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



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 */

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;
```

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 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

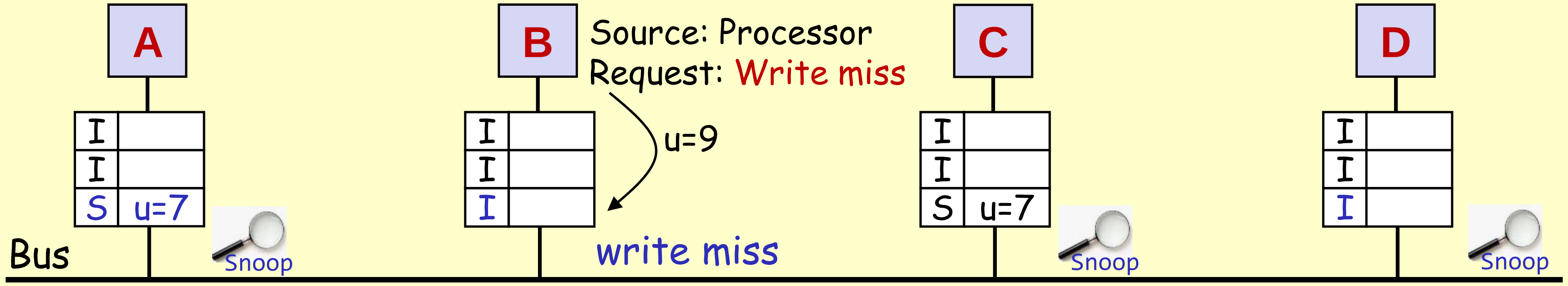
Implementing an exchange EXCH

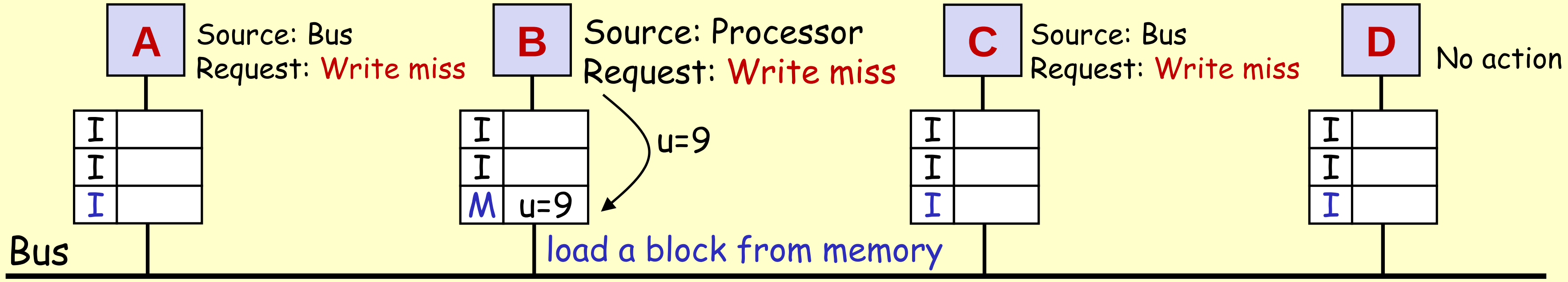
- EXCH R4,0(R1) ; exchange R4 and 0(R1)
  - Why isn't this code atomic?

```
LW  R2,0(R1)      ; load  word,  Tmp <- shared data
SW  R4,0(R1)      ; store word,  R4  -> shared data
ADD R4,R2,R0      ; copy,        R4  <- Tmp
```

Coherence 4 (C4)

- Core B
  - Source: Processor
  - State: Invalid
  - Request: Write miss
  - Function: Place write miss on bus
- C4 (Core A, C)
  - Source: Bus
  - State: Shared
  - Request: Write miss
  - Function: attempt to write shared block; invalidate the cache block





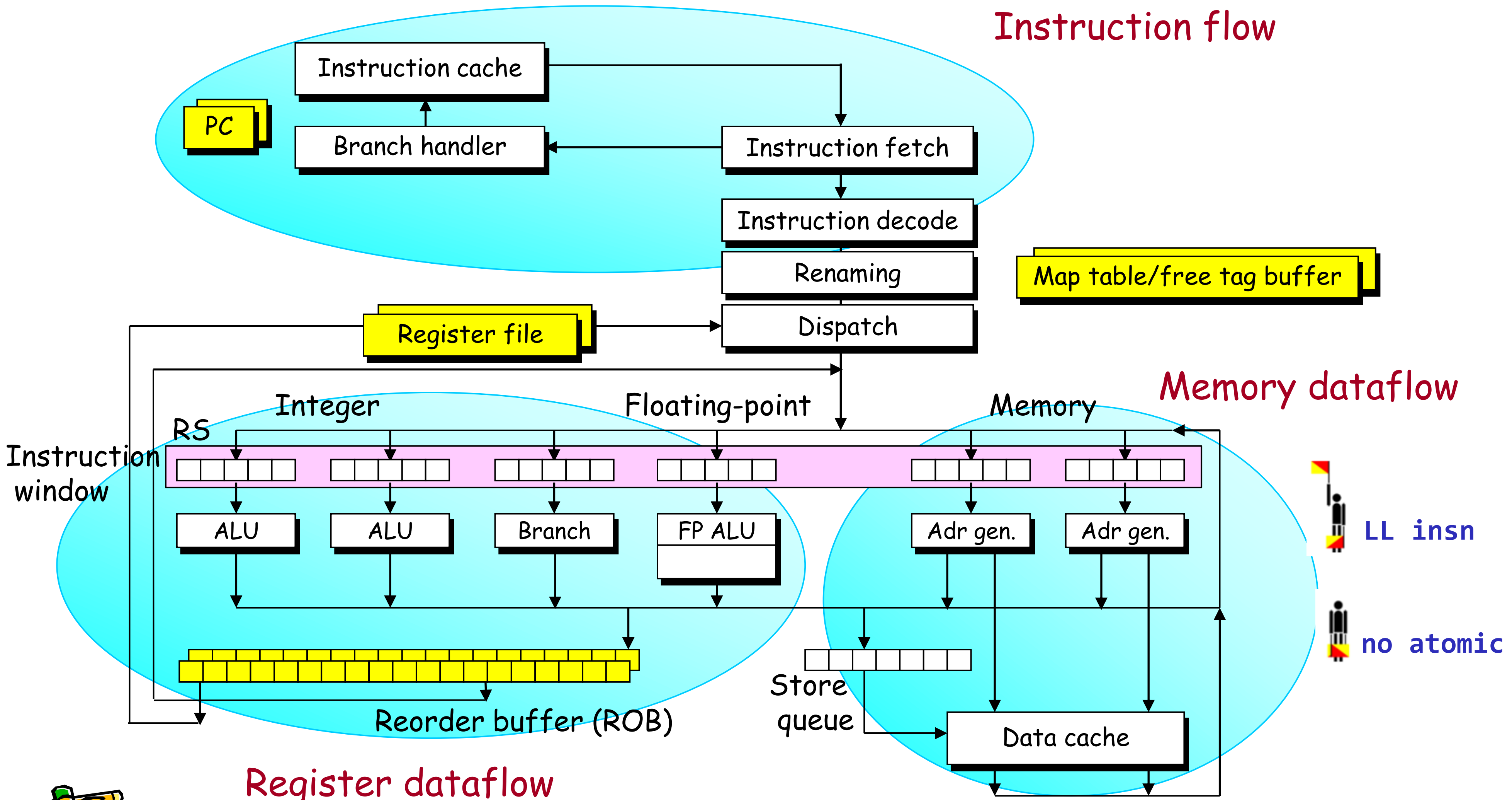
Implementing an atomic exchange EXCH

- 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 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==0)
       ADD R4,R2,R0        ; put load value in R4, R4=R2
```

Datapath of SMT OoO execution processor



Reservation station (RS)

this slide is to be used as a whiteboard

Implementing Locks (simple version)

- Spin lock
  - R1 is the address of the lock variable and its initial value is 0 (not locked).

```
            ADDI    R4, R0, 1       ; R2 = 1
lockit:     EXCH    R4, 0(R1)       ; atomic exchange
            BNE     R4,R0,lockit    ; already locked?
```

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==0)
        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      R4, 0(R1)        ; load of lock
           BNE     R4,R0,lockit     ; not available-spin if R4==1
           ADDI    R4,R0,1          ; load locked value, R4<=1
           EXCH    R4,0(R1)         ; swap
           BNE     R4,R0,lockit     ; branch if lock wasn't 0
```

Performance Factors

$$\text{CPU execution time for a program} = \frac{\text{\# CPU clock cycles for a program}}{\text{clock rate}}$$

$$\text{Performance for a program} = \text{clock rate} \times 1 / \text{\# CPU clock cycles for a program}$$

- Performance = f x IPC
  - f: frequency (clock rate)
  - IPC: retired instructions per cycle

```
int flag = 1;

int foo(){
  while(flag);
}
```

this slide is to be used as a whiteboard

Implementing Unlocks using coherence

- Unlock
  - Just resetting the lock variable

```
unlock:    SW  R0, 0(R1)  ; reset the lock, lock_variable = 0
```

Implementing Barriers using coherence

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

```
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 == p) {
        counter = 0;
        flag = 1;
    }
    else while (flag == 0);         /* wait until all threads reach BARRIER */
}
```

this slide is to be used as a whiteboard

Memory consistency: problem in multi-core context

- Assume that A=0 and Flag=0 initially
- Core 1 (C1) writes data into A and sets Flag to tell C2 that data value can be read (loaded) from A.
- C2 waits till Flag is set and then reads (loads) data from A.
- What is the printed value by C2?

C1 (Core 1)	C2 (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 C1 can be reordered, it is possible for C2 to read 0 from variable A.
- This can happen on most modern processors.
    - For single-core processor, Code1 and Code2 are equivalent. These writes may be reordered by compilers statically or by OoO execution units dynamically.
    - The printed value by C2 will be 0 or 3.

Code1

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

Code2

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

C1 (Core 1)

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

C2 (Core 2)

```
while (Flag==0);
print A;
```

Assume that A=0 and Flag=0 initially

Problem in multi-core context

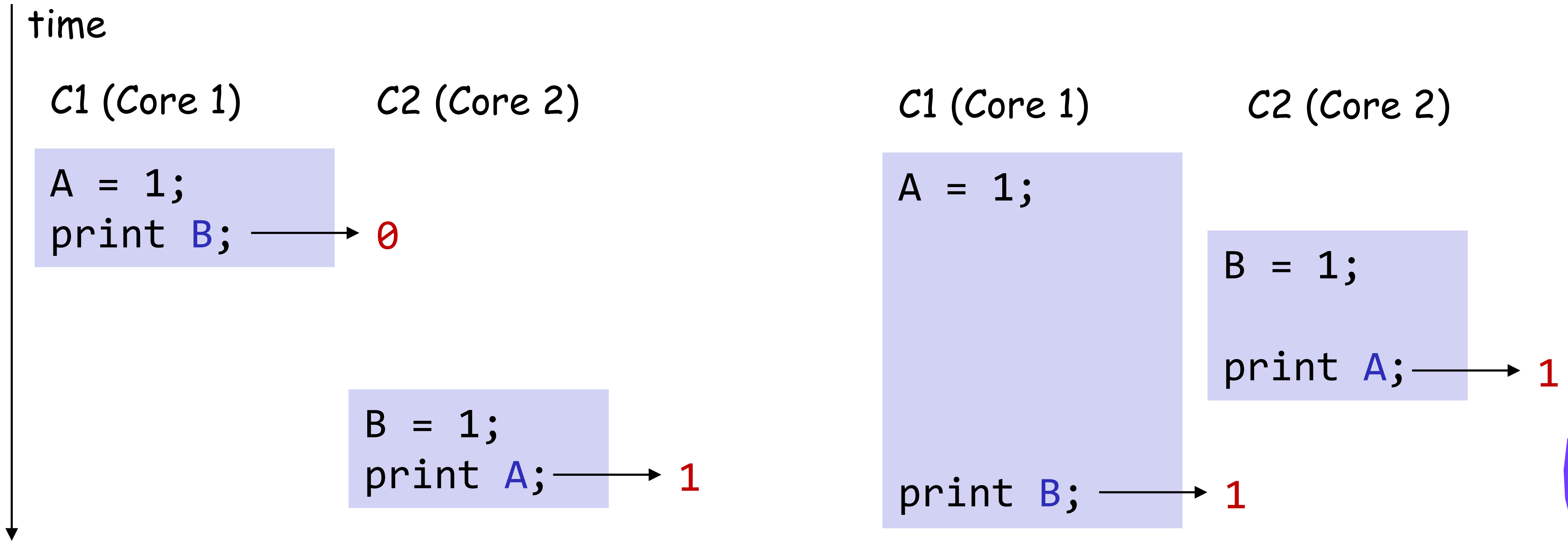
- Assume that A=0 and B=0 initially
- Should be impossible for both outputs to be zero.
  - Intuitively, the outputs may be 01, 10, and 11.

C1 (Core 1)	C2 (Core 2)

`A = 1;`
`print B;`
	`B = 1;`
	`print A;`

Examples

- Assume that A=0 and B=0 initially



```
A = 1;
print B;
```

```
B = 1;
print A;
```

```
A = 1;
print B;
```

```
B = 1;
print A;
```

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.

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

Memory Consistency Models

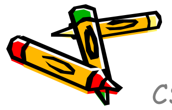
- A single-core processor can reorder instructions subject only to control and data dependence constraints
- These constraints are not sufficient in shared-memory multi-cores
  - simple parallel programs may produce counter-intuitive results

- 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

this slide is to be used as a whiteboard

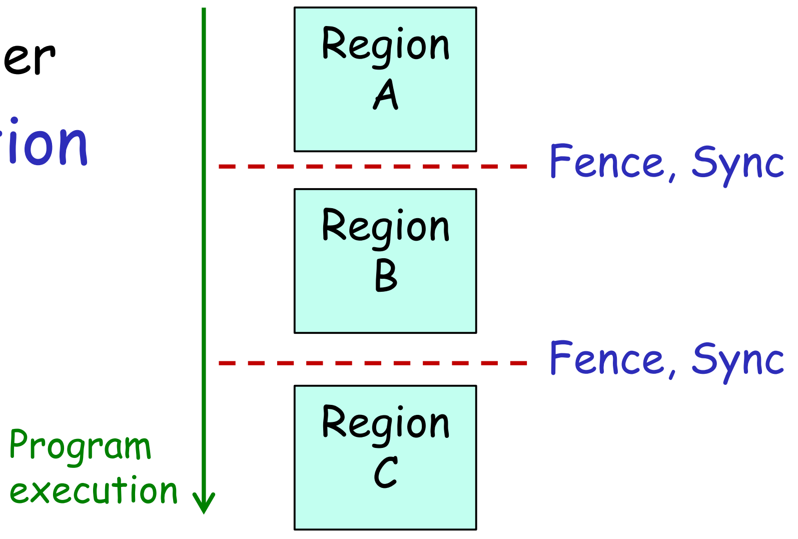
Simple and Intuitive Model: Sequential Consistency

- 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?



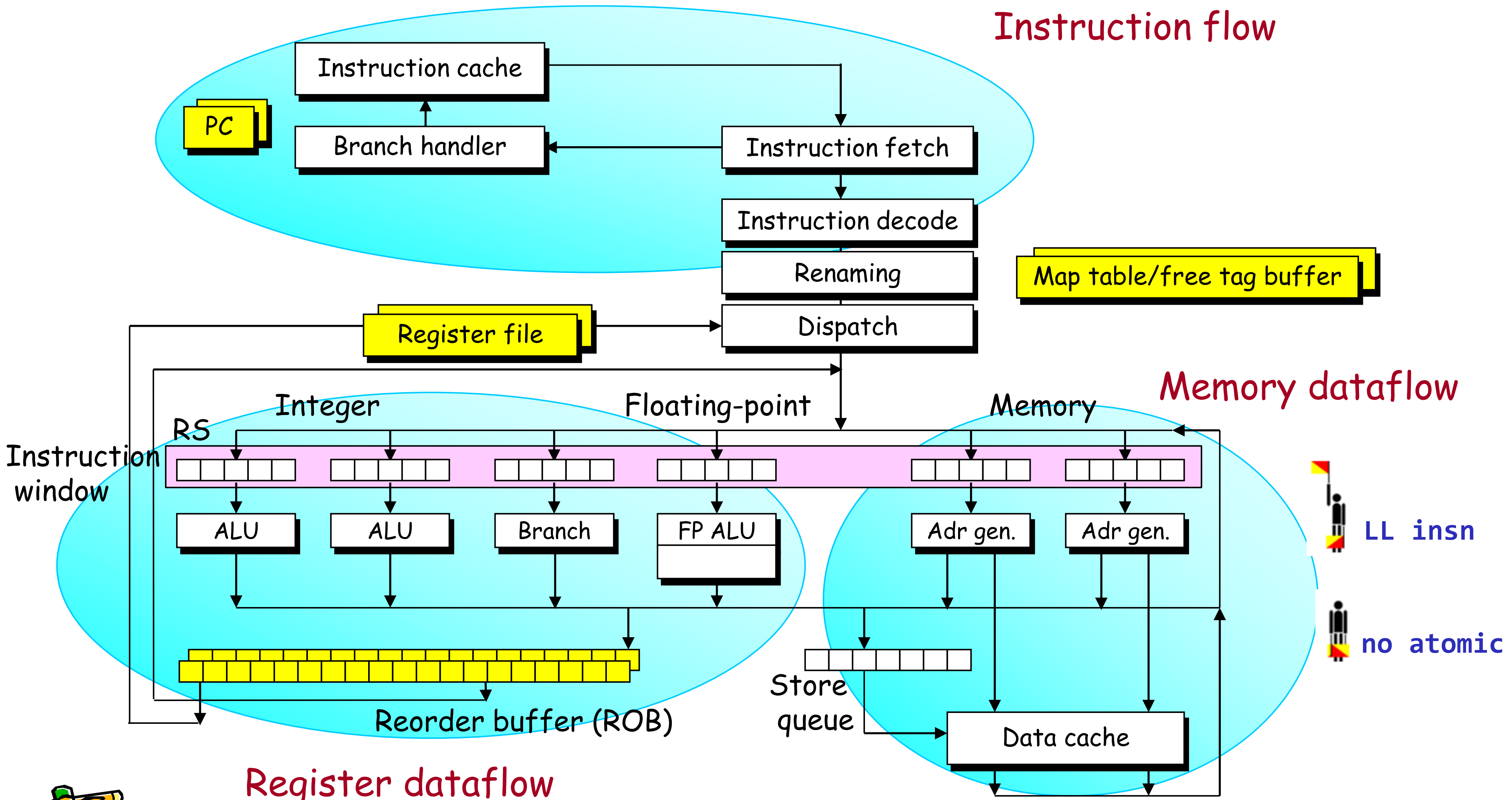
Relaxed consistency model: Weak Consistency

- 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: MIPS has SYNC instruction
- Implementation of SYNC
  - a processor may flush all instructions when a SYNC instruction is retired



Memory operations within a region can be reordered

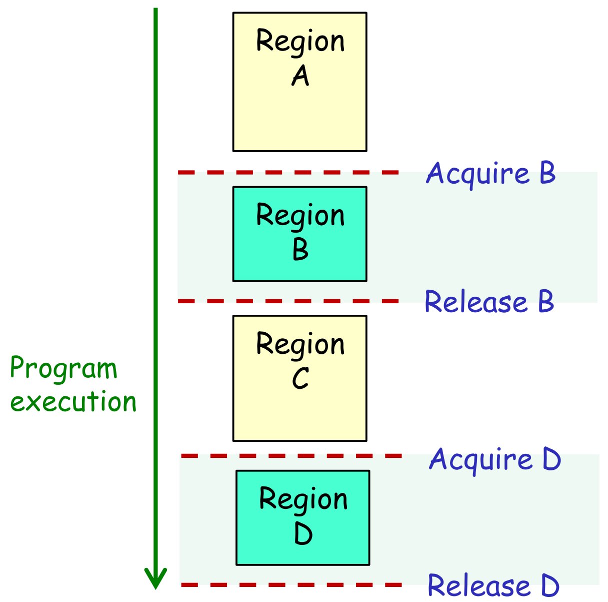
Datapath of SMT OoO execution processor

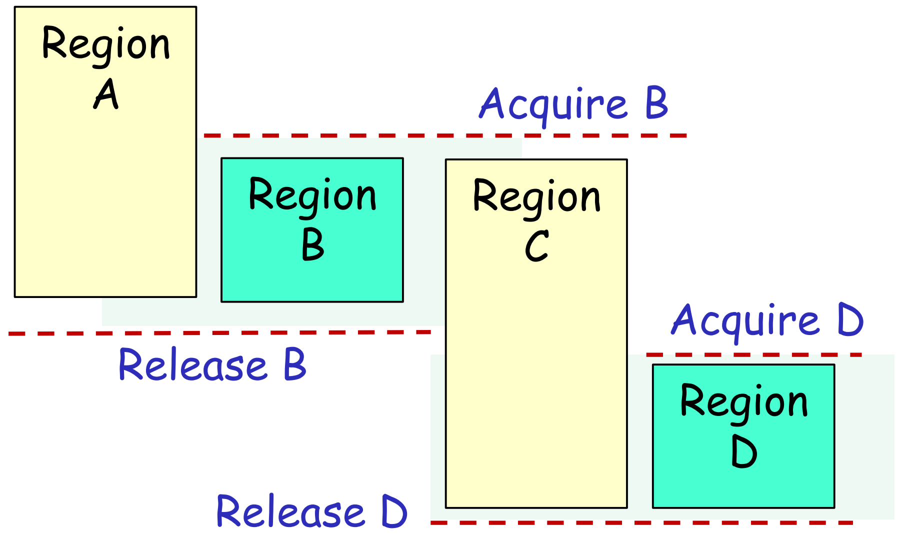


Reservation station (RS)

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
  - Memory operations in region B and C must complete after Acquire B
- Semantics of Release:
  - all memory operations before Release are complete
  - Memory operations in region A and B must complete before Release B



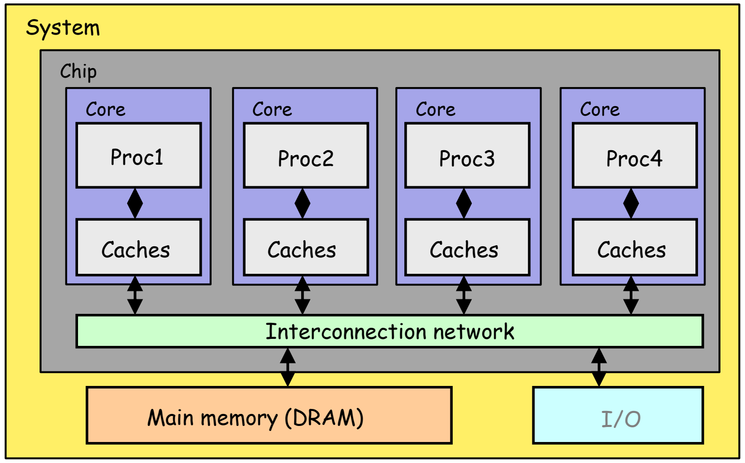


Memory Consistency Model

- 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

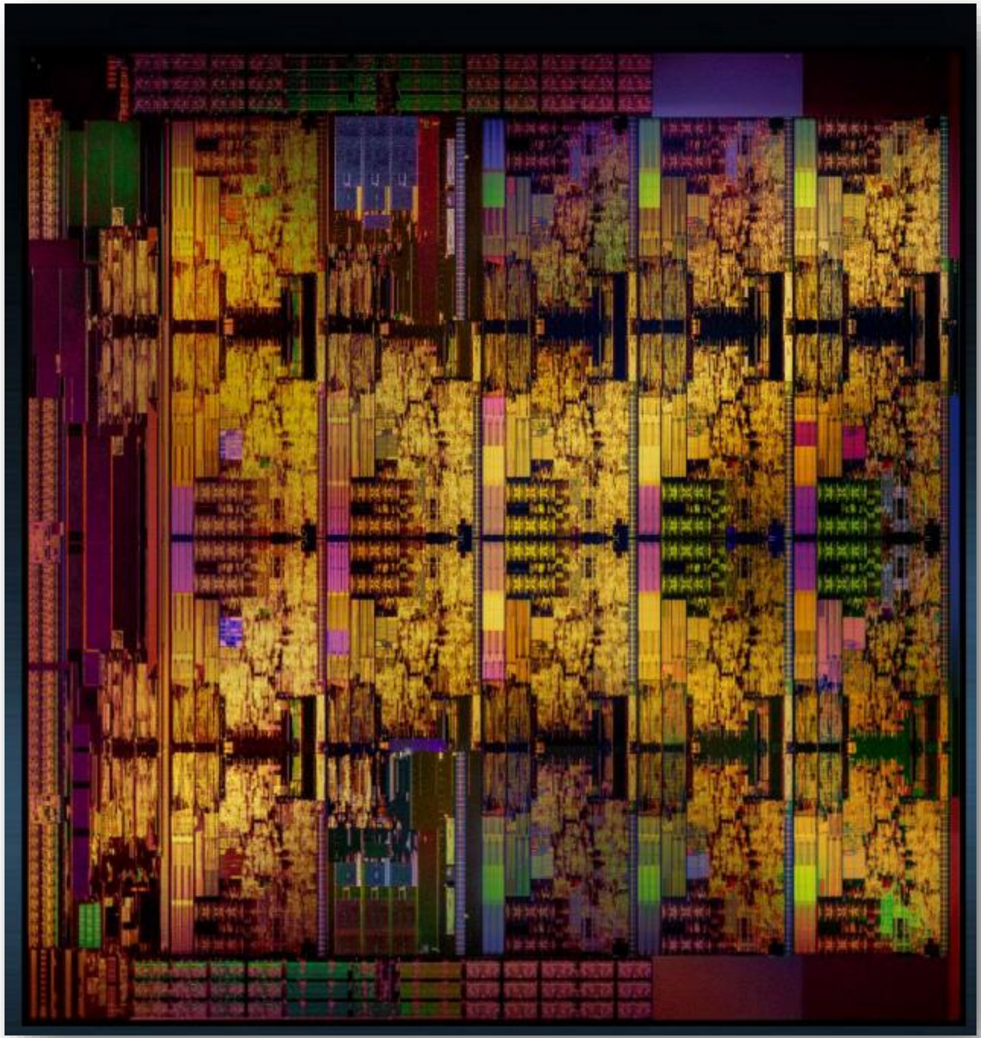
Key components of many-core processors

- 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

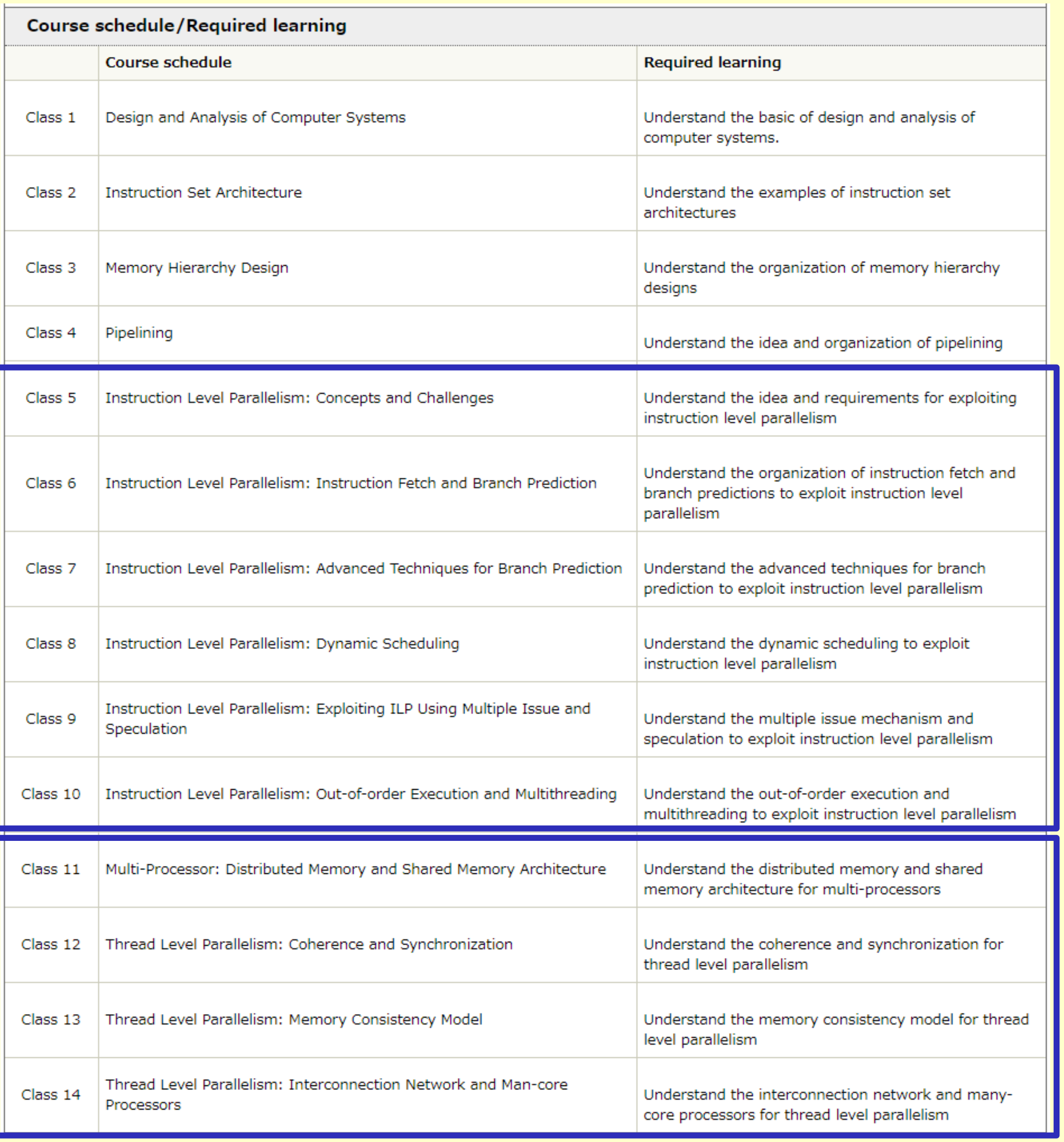


Putting It All Together

- 18 core
- 2D mesh topology



Syllabus (3/3)



Course schedule/Required learning

	Course schedule	Required learning
Class 1	Design and Analysis of Computer Systems	Understand the basic of design and analysis of computer systems.
Class 2	Instruction Set Architecture	Understand the examples of instruction set architectures
Class 3	Memory Hierarchy Design	Understand the organization of memory hierarchy designs
Class 4	Pipelining	Understand the idea and organization of pipelining
Class 5	Instruction Level Parallelism: Concepts and Challenges	Understand the idea and requirements for exploiting instruction level parallelism
Class 6	Instruction Level Parallelism: Instruction Fetch and Branch Prediction	Understand the organization of instruction fetch and branch predictions to exploit instruction level parallelism
Class 7	Instruction Level Parallelism: Advanced Techniques for Branch Prediction	Understand the advanced techniques for branch prediction to exploit instruction level parallelism
Class 8	Instruction Level Parallelism: Dynamic Scheduling	Understand the dynamic scheduling to exploit instruction level parallelism
Class 9	Instruction Level Parallelism: Exploiting ILP Using Multiple Issue and Speculation	Understand the multiple issue mechanism and speculation to exploit instruction level parallelism
Class 10	Instruction Level Parallelism: Out-of-order Execution and Multithreading	Understand the out-of-order execution and multithreading to exploit instruction level parallelism
Class 11	Multi-Processor: Distributed Memory and Shared Memory Architecture	Understand the distributed memory and shared memory architecture for multi-processors
Class 12	Thread Level Parallelism: Coherence and Synchronization	Understand the coherence and synchronization for thread level parallelism
Class 13	Thread Level Parallelism: Memory Consistency Model	Understand the memory consistency model for thread level parallelism
Class 14	Thread Level Parallelism: Interconnection Network and Man-core Processors	Understand the interconnection network and many-core processors for thread level parallelism

Final report of Advanced Computer Architecture

1. Please submit your final report describing your answers to questions 1 - 7 in a PDF file
   via E-mail (kise [at] c.titech.ac.jp) by February 13, 2023
   - E-mail title should be "Report of Advanced Computer Architecture"
2. Please submit the report in 16 pages or less on A4 size PDF file, including the cover page.
3. Enjoy!