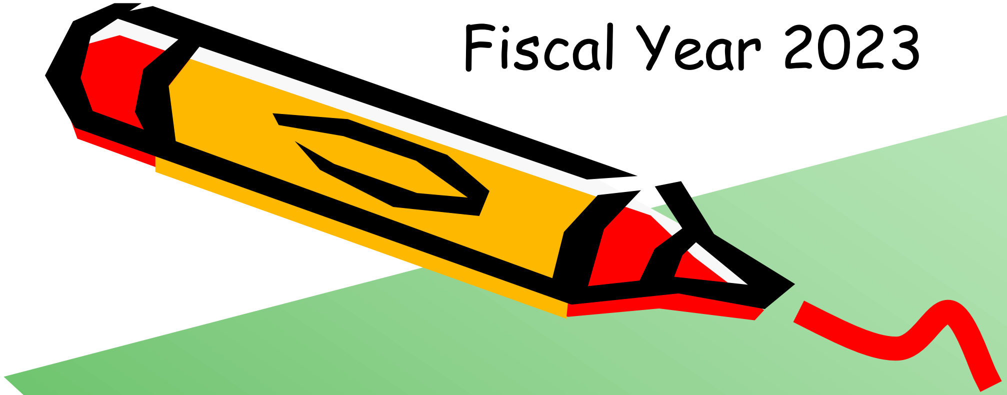


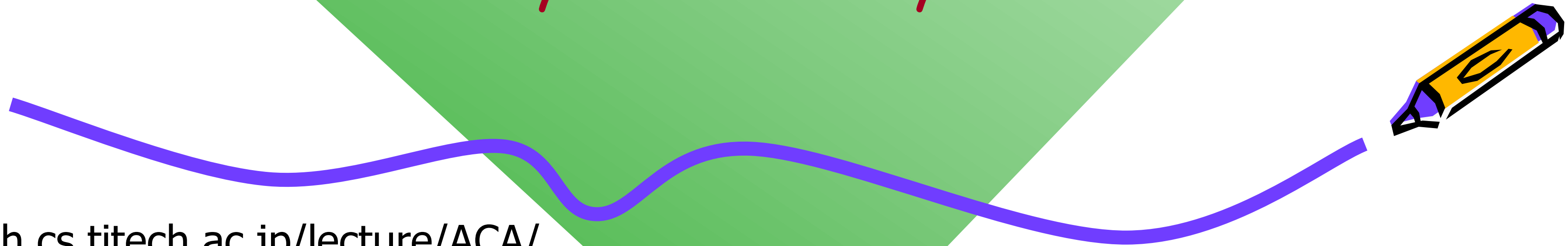
Fiscal Year 2023

Ver. 2024-02-01a

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

Advanced Computer Architecture

13. Thread Level Parallelism: Memory Consistency Model

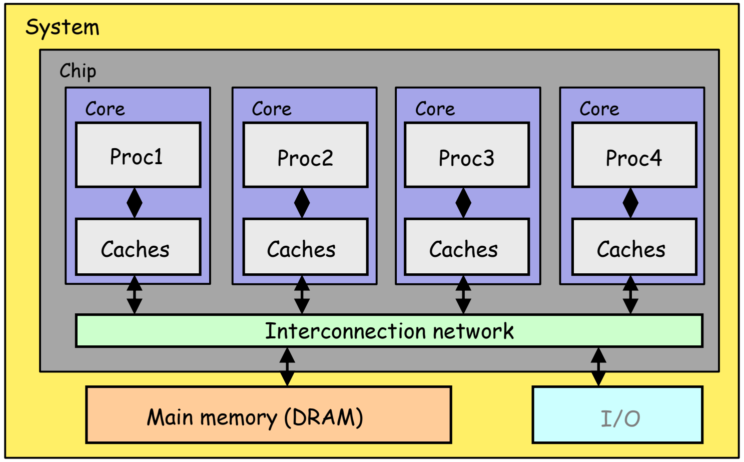


www.arch.cs.titech.ac.jp/lecture/ACA/
Room No.W834, Lecture (Face-to-face)
Mon 13:30-15:10, Thr 13:30-15:10

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

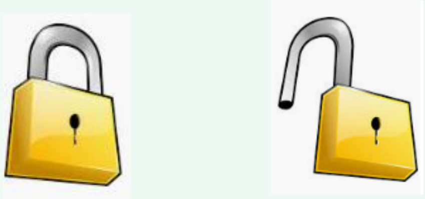
Key components of many-core processors

- 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 */
int ncores = 2;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_barrier_t barrier;
void solve_pp (int pid) {
    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]);
        }
        pthread_mutex_lock(&m);
        diff = diff + mydiff;
        pthread_mutex_unlock(&m);

        pthread_barrier_wait(&barrier);
        if (diff <TOL) done = 1;
        pthread_barrier_wait(&barrier);
        if (pid==1) diff = 0.0;
        for (i=mymin; i<=mymax; i++) A[i] = B[i];
        pthread_barrier_wait(&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.

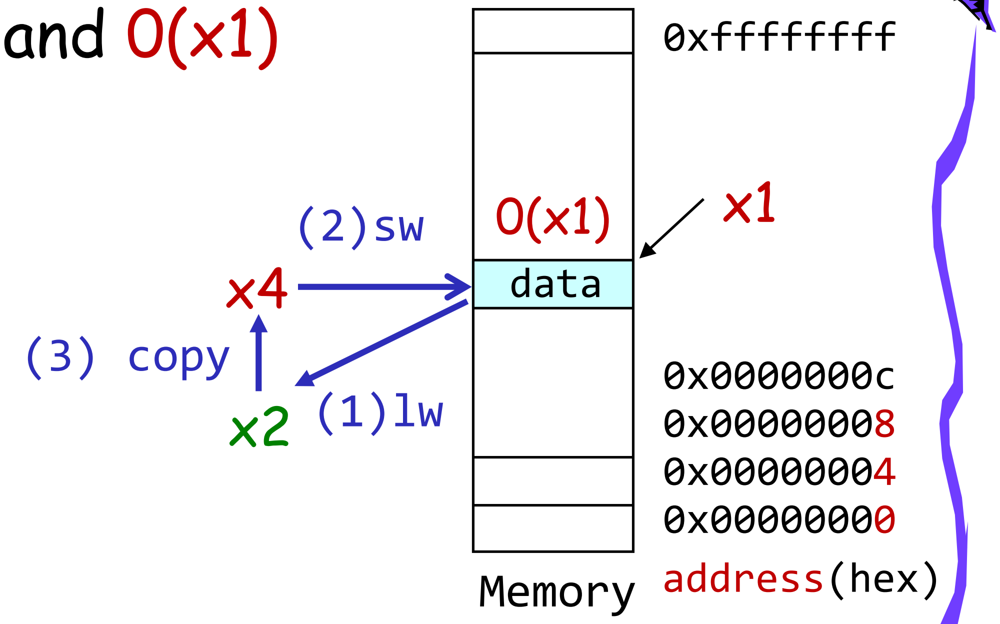
```
if (diff <TOL) done = 1;
```

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 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?



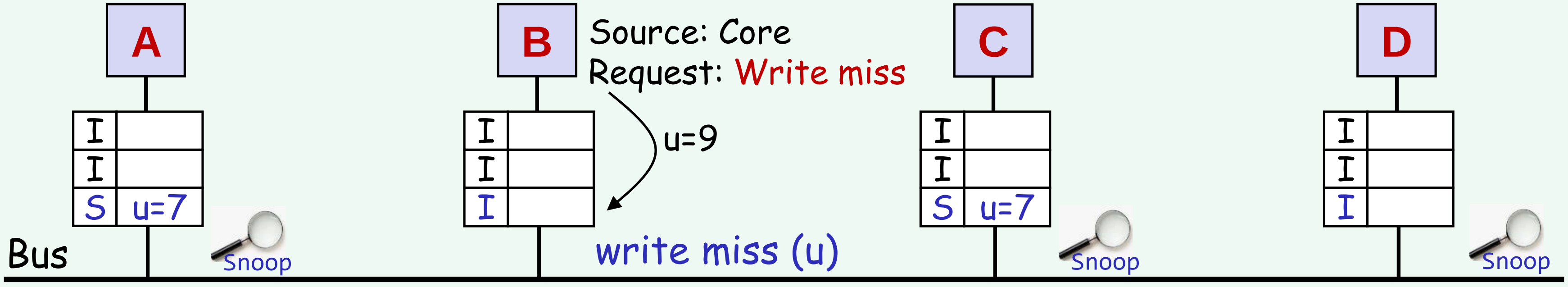
```
lw  x2,0(x1)    # load  word,  Tmp <- shared data
sw  x4,0(x1)    # store word,  x4  -> shared data
add x4,x2,x0    # copy,        x4  <- Tmp
```

Timer interrupt, cache coherence protocol

Coherence 4 (Coh4)

- Core B
  - Source: Core
  - State: Invalid
  - Request: Write miss (u)
  - Function: Place write miss on bus

- Coh4 (Core A, C)
  - Source: Bus
  - State: Shared
  - Request: Write miss (u)
  - Function: attempt to write shared block; invalidate the cache block



A: I / I / S u=7 — Snoop

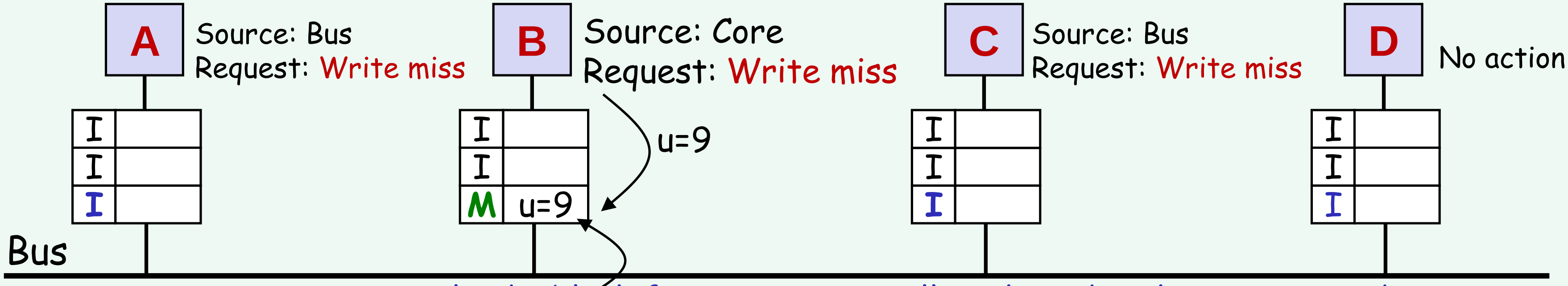
B: Source: Core
Request: Write miss
u=9
B: I / I / I

C: I / I / S u=7 — Snoop

D: I / I / I — Snoop

Bus

write miss (u)



A: Source: Bus
Request: Write miss
A: I / I / I

B: Source: Core
Request: Write miss
u=9
B: I / I / M u=9

C: Source: Bus
Request: Write miss
C: I / I / I

D: No action
D: I / I / I

Bus

load a block from memory or allow shared cache to service data

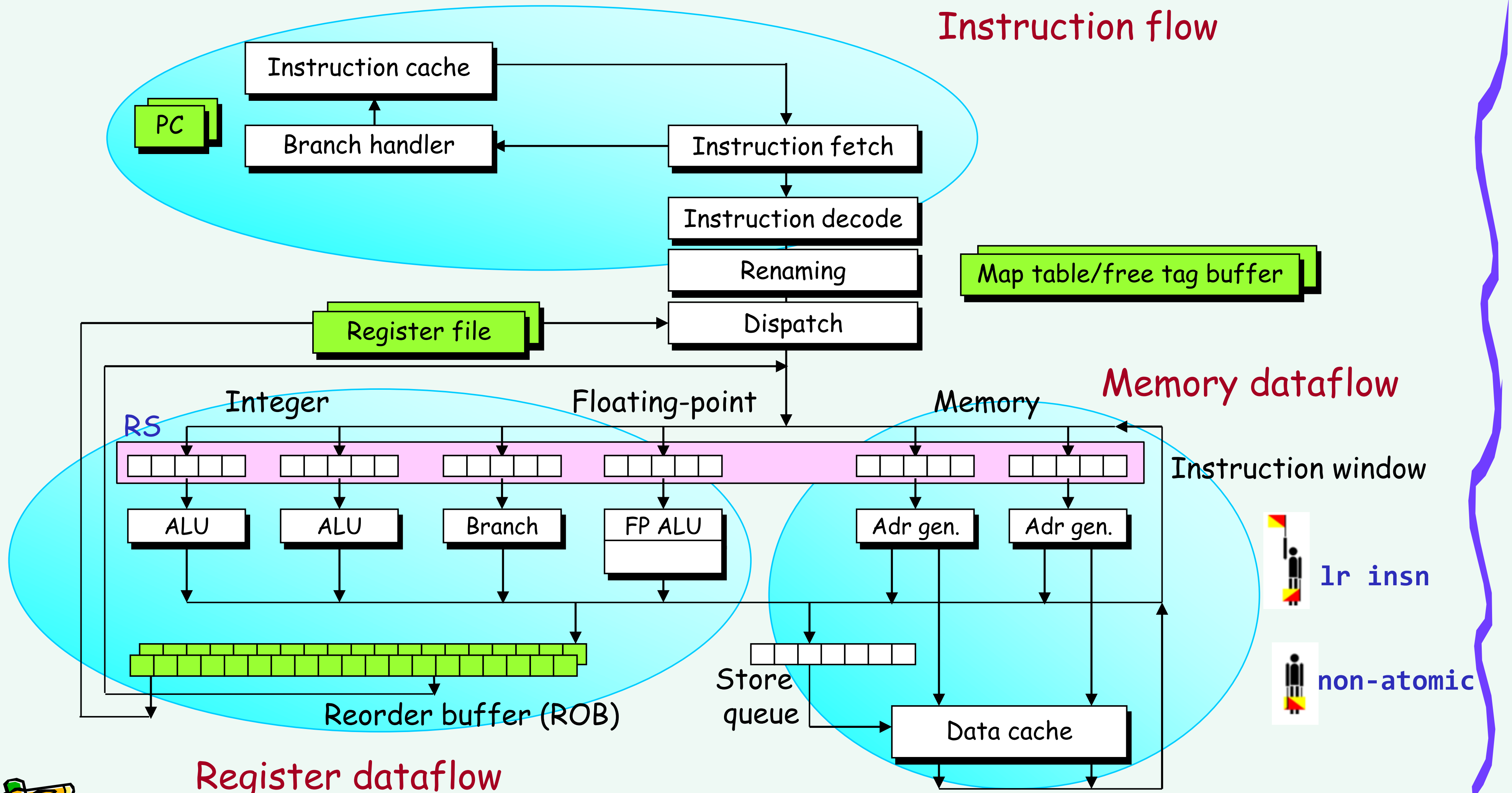
Implementing an atomic exchange EXCH

- 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 SMT OoO execution processor

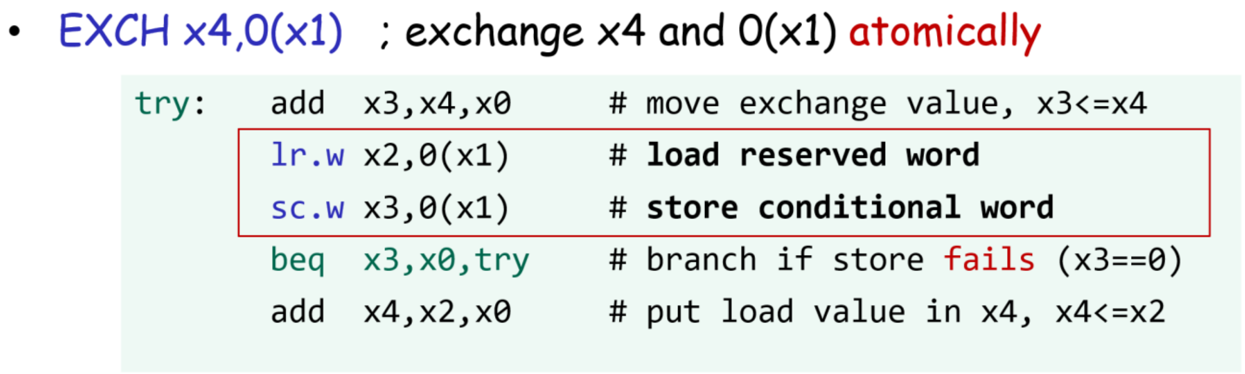


Reservation station (RS)

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 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         # load 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
```

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 */
}
```

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 processor, Code(1) and 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.

Code(1)

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

Code(2)

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

Core 1

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

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)

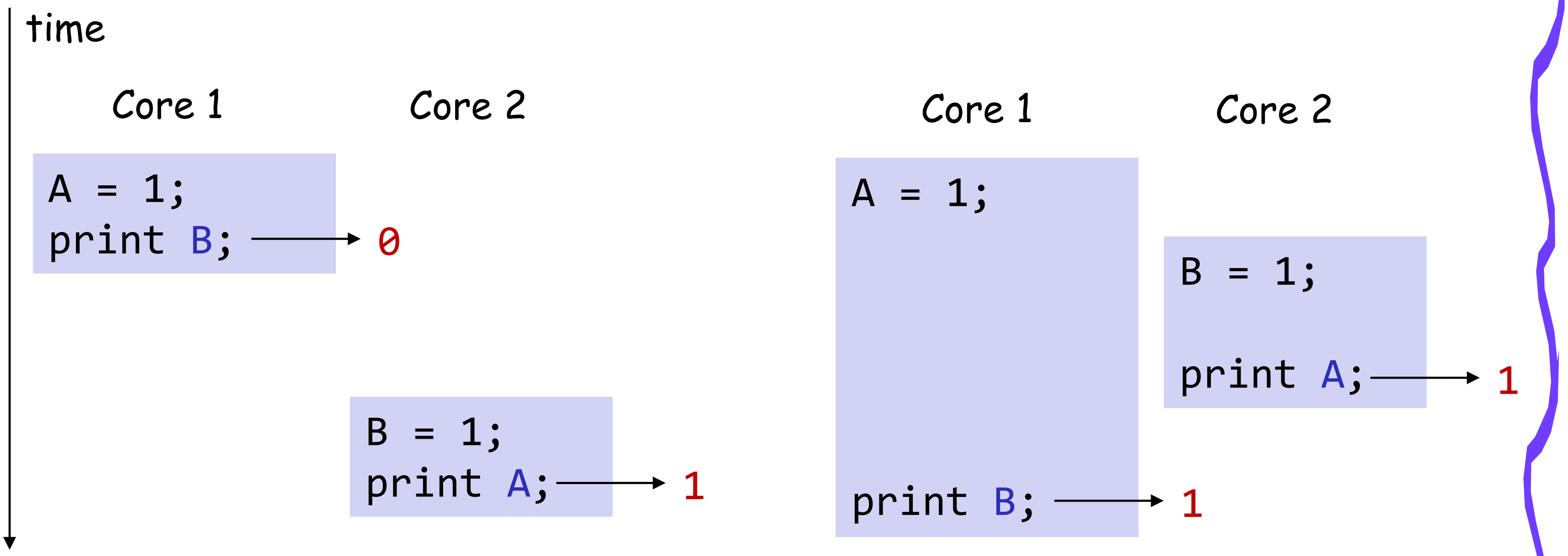
```
A = 1;
print B;
```

C2 (Core 2)

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

Example behaviours

- 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.

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

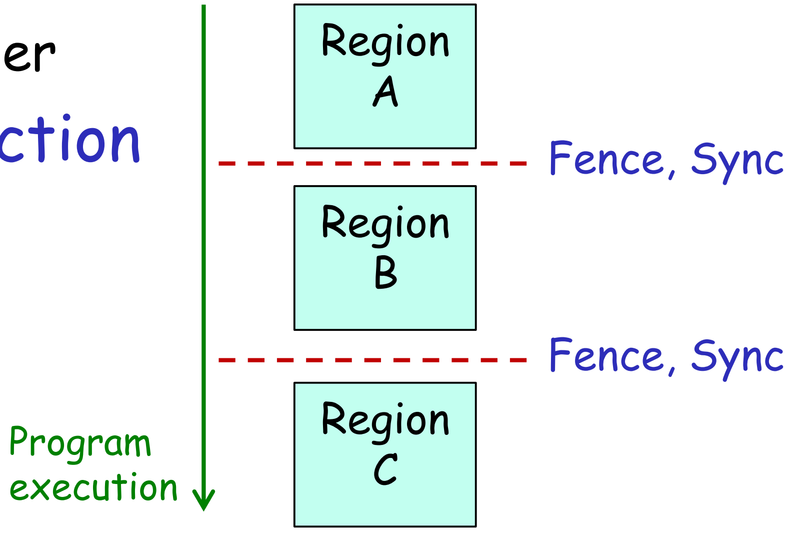
- Question: what constraints must we put on single-core instruction reordering so that
  - shared-memory programming is intuitive
  - but we do not lose single-core performance?
- The answers are called memory consistency models supported by the processor
  - Memory consistency models are all about ordering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

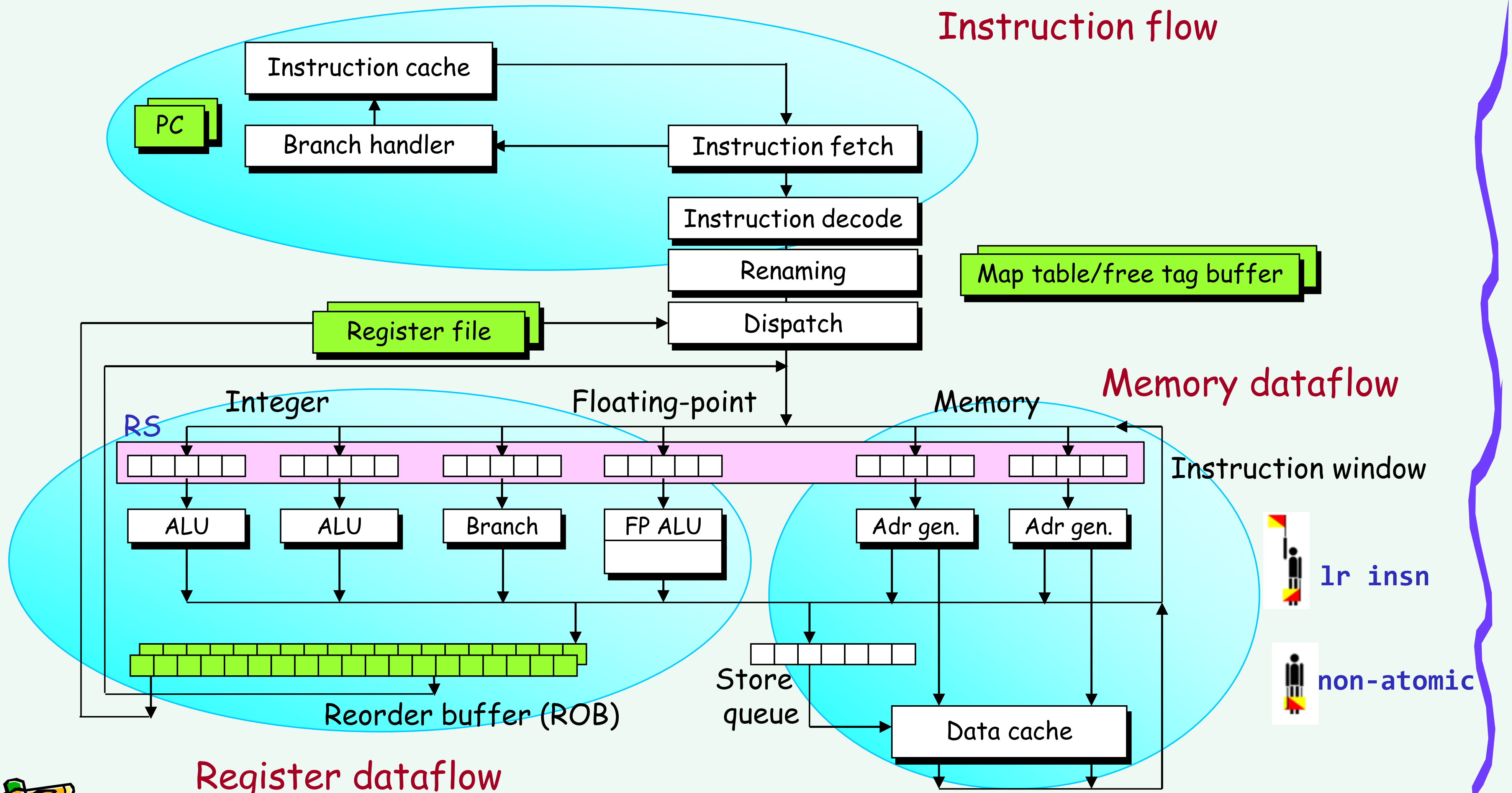
Relaxed consistency model: weak consistency

- 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



Memory operations within a region can be reordered

Datapath of SMT OoO execution processor

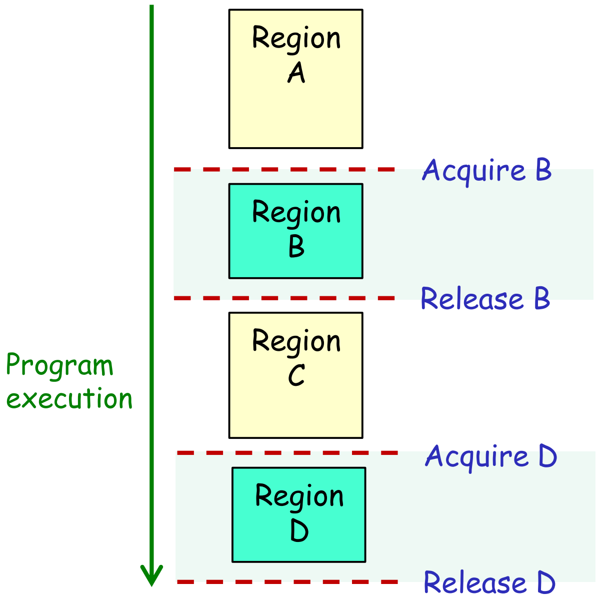


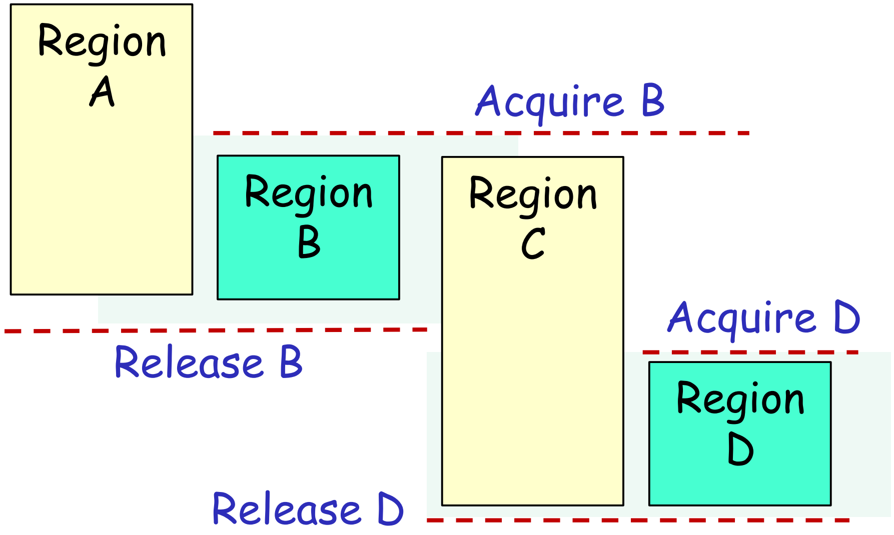
Register dataflow

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



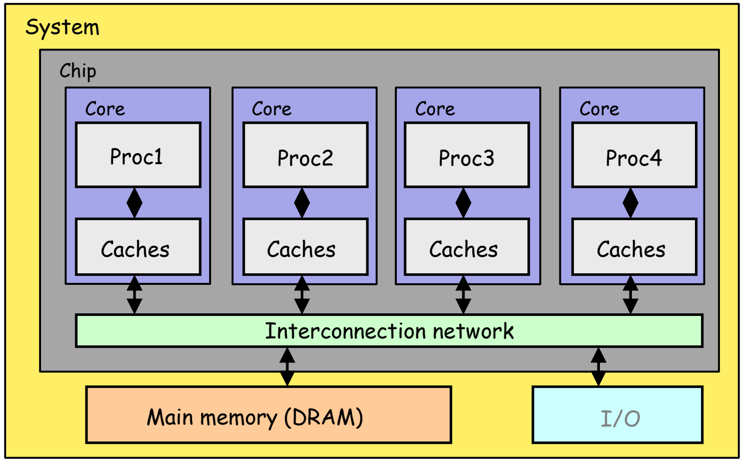


Memory Consistency Model

- 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

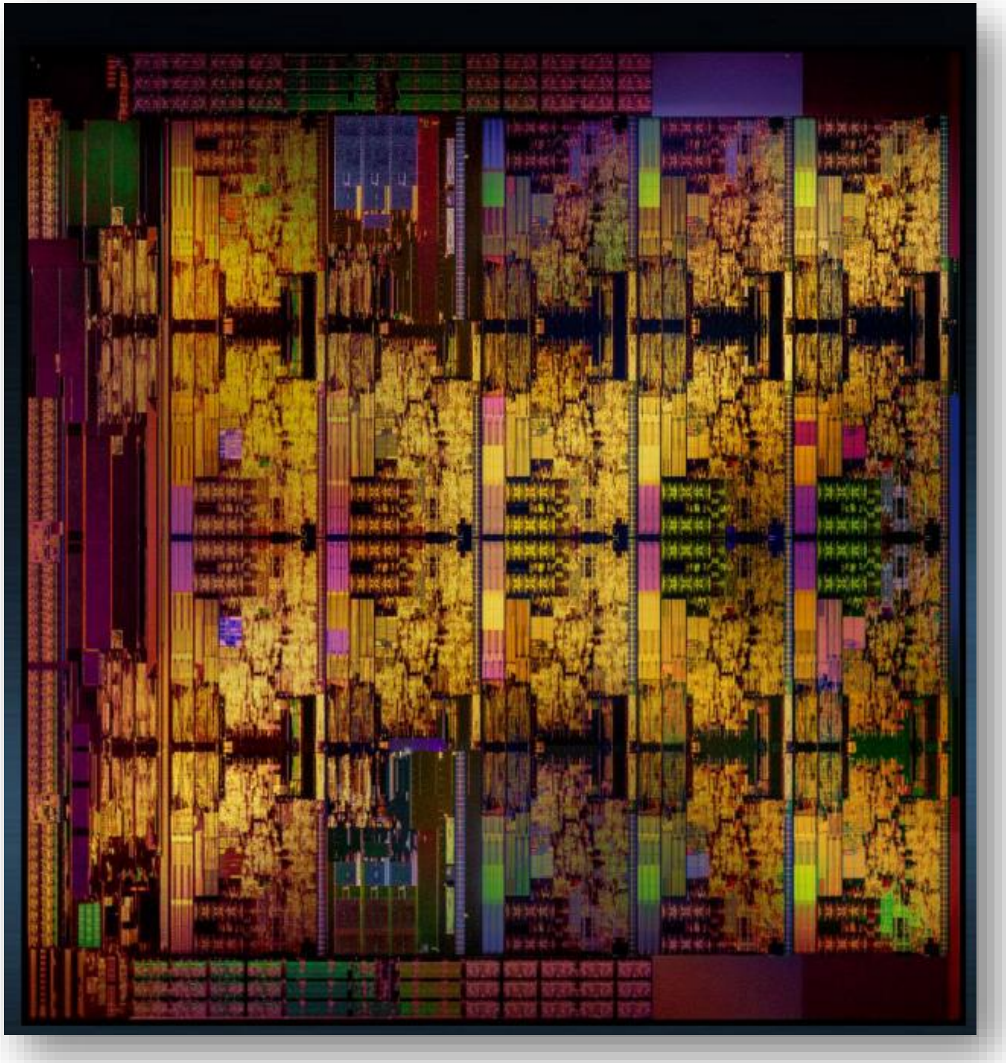
Key components of many-core processors

- 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

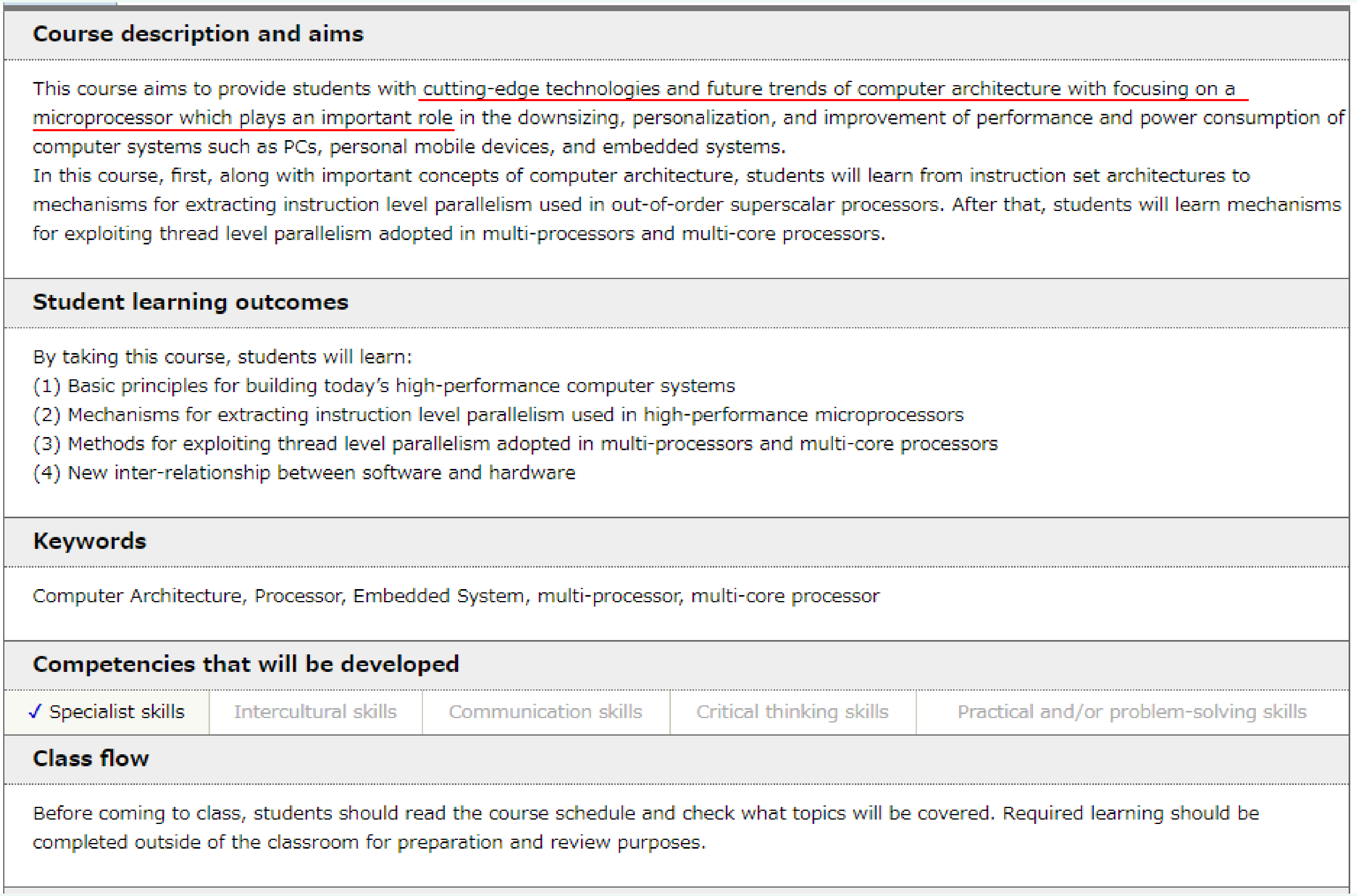


Putting It All Together

- 18 core
- 2D mesh topology



Syllabus (1/3)



Course description and aims

This course aims to provide students with cutting-edge technologies and future trends of computer architecture with focusing on a microprocessor which plays an important role in the downsizing, personalization, and improvement of performance and power consumption of computer systems such as PCs, personal mobile devices, and embedded systems.
In this course, first, along with important concepts of computer architecture, students will learn from instruction set architectures to mechanisms for extracting instruction level parallelism used in out-of-order superscalar processors. After that, students will learn mechanisms for exploiting thread level parallelism adopted in multi-processors and multi-core processors.

Student learning outcomes

By taking this course, students will learn:
(1) Basic principles for building today's high-performance computer systems
(2) Mechanisms for extracting instruction level parallelism used in high-performance microprocessors
(3) Methods for exploiting thread level parallelism adopted in multi-processors and multi-core processors
(4) New inter-relationship between software and hardware

Keywords

Computer Architecture, Processor, Embedded System, multi-processor, multi-core processor

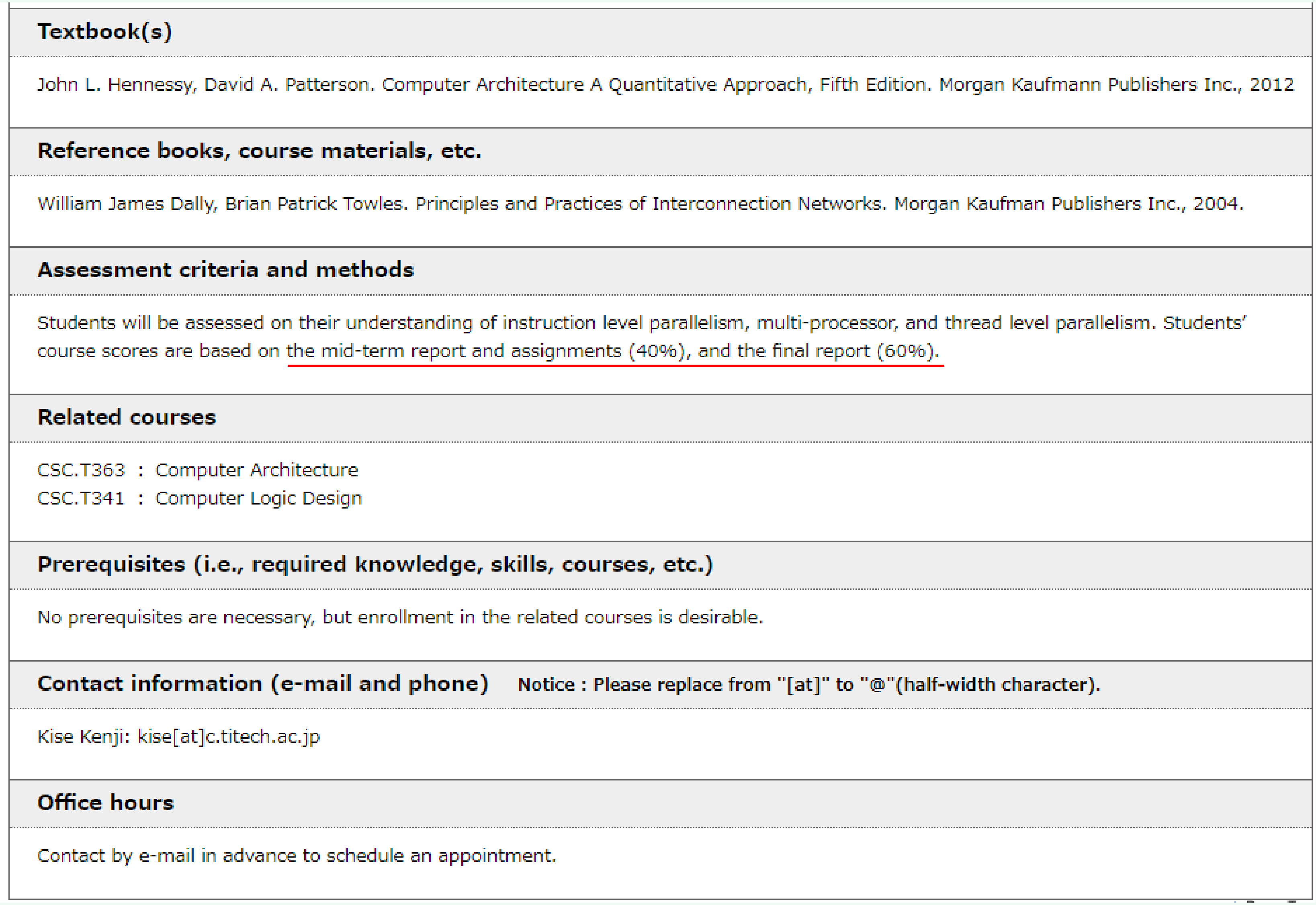
Competencies that will be developed

✓ Specialist skills	Intercultural skills	Communication skills	Critical thinking skills	Practical and/or problem-solving skills

Class flow

Before coming to class, students should read the course schedule and check what topics will be covered. Required learning should be completed outside of the classroom for preparation and review purposes.

Syllabus (2/3)



Textbook(s)

John L. Hennessy, David A. Patterson. Computer Architecture A Quantitative Approach, Fifth Edition. Morgan Kaufmann Publishers Inc., 2012

Reference books, course materials, etc.

William James Dally, Brian Patrick Towles. Principles and Practices of Interconnection Networks. Morgan Kaufman Publishers Inc., 2004.

Assessment criteria and methods

Students will be assessed on their understanding of instruction level parallelism, multi-processor, and thread level parallelism. Students' course scores are based on the mid-term report and assignments (40%), and the final report (60%).

Related courses

CSC.T363 ： Computer Architecture
CSC.T341 ： Computer Logic Design

Prerequisites (i.e., required knowledge, skills, courses, etc.)

No prerequisites are necessary, but enrollment in the related courses is desirable.

Contact information (e-mail and phone)

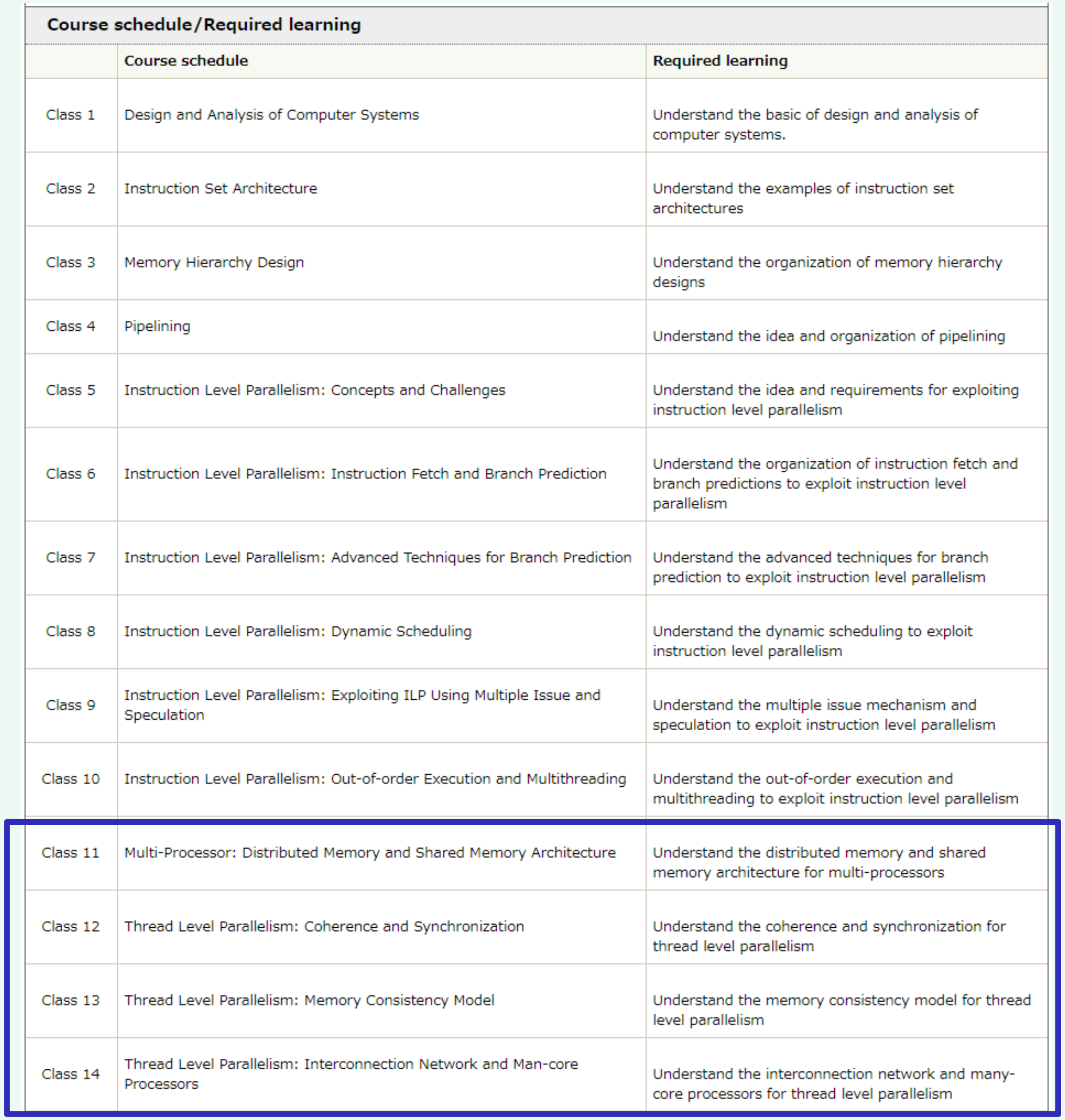
Notice : Please replace from "[at]" to "@"(half-width character).

Kise Kenji: kise[at]c.titech.ac.jp

Office hours

Contact by e-mail in advance to schedule an appointment.

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, 2024
   - 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!