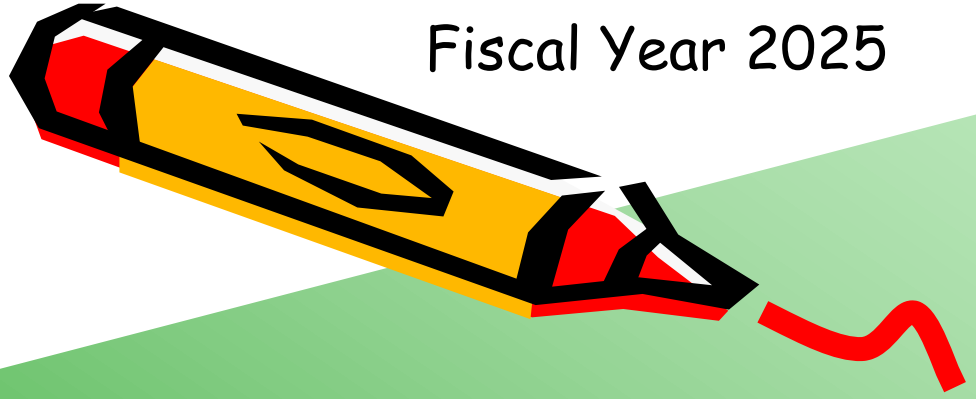


Fiscal Year 2025

Ver. 2025-12-25a

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

Computer Organization and Architecture

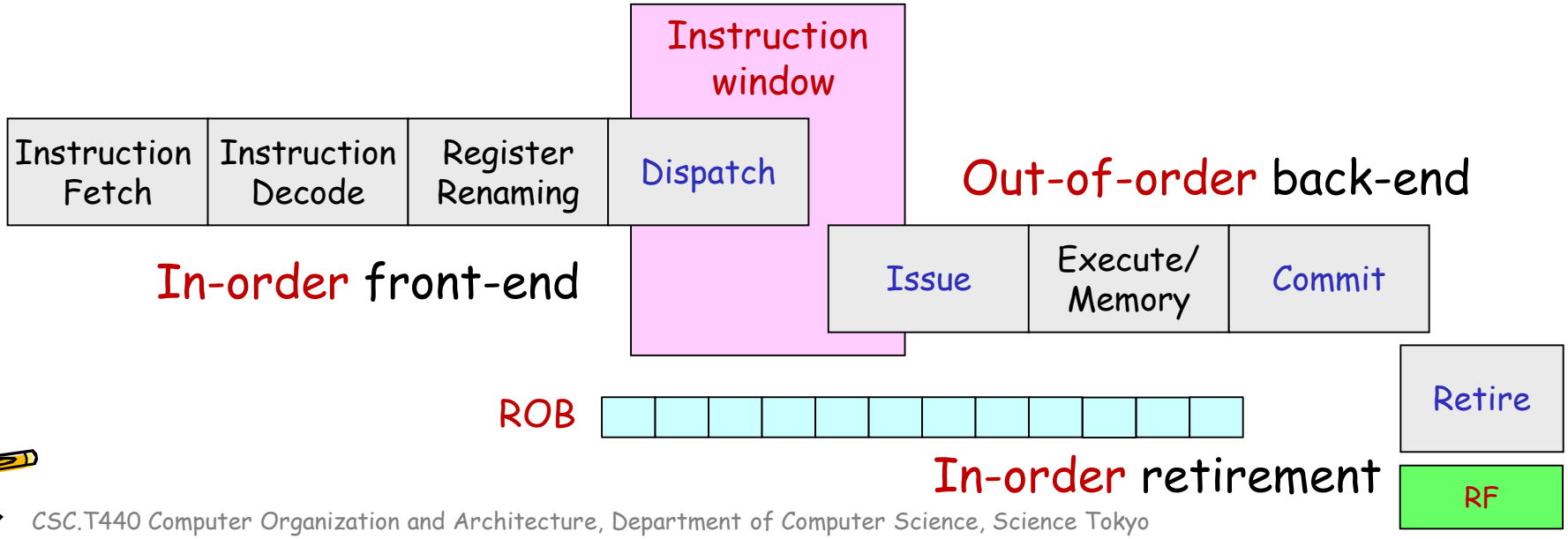
4. Instruction Level Parallelism: Multiple Issue, Speculation, and Out-of-order Execution

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

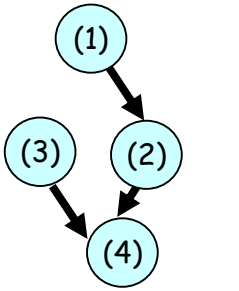
Instruction pipeline of OoO execution processor

- Allocating instructions to instruction window is called dispatch
- Issue or fire wakes up instructions and their executions begin
- In commit stage, *the computed values are written back to ROB (reorder buffer)*
- The last stage is called retire or graduate.
  The completed consecutive instructions can be retired.
  The result is written back to register file (*architectural register file of 32 registers*) using a logical register number from x0 to x31.

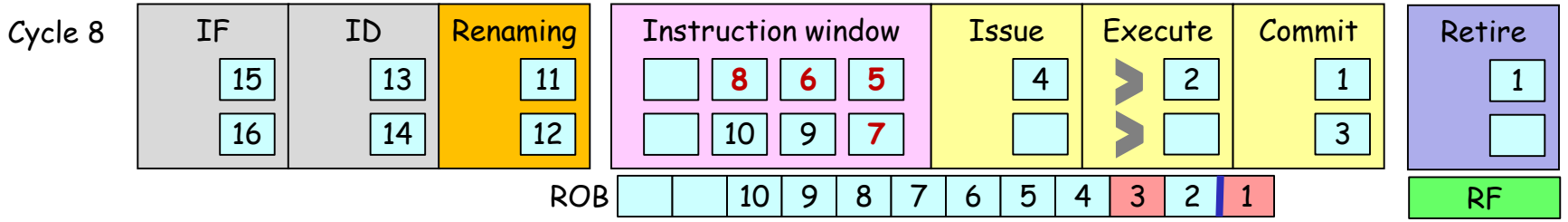


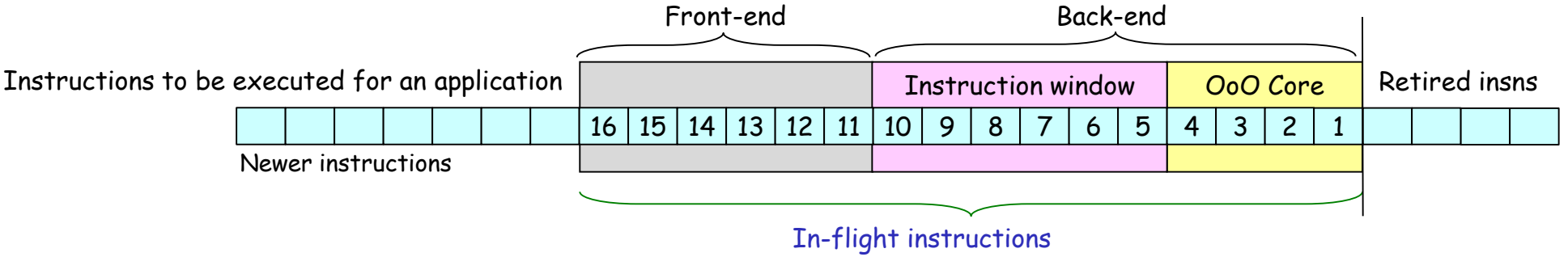
Register dataflow

- In-flight instructions are ones processing in a processor



Data flow graph

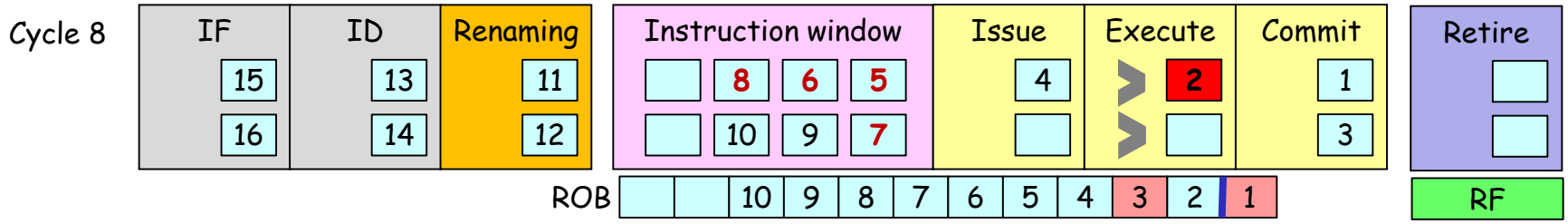


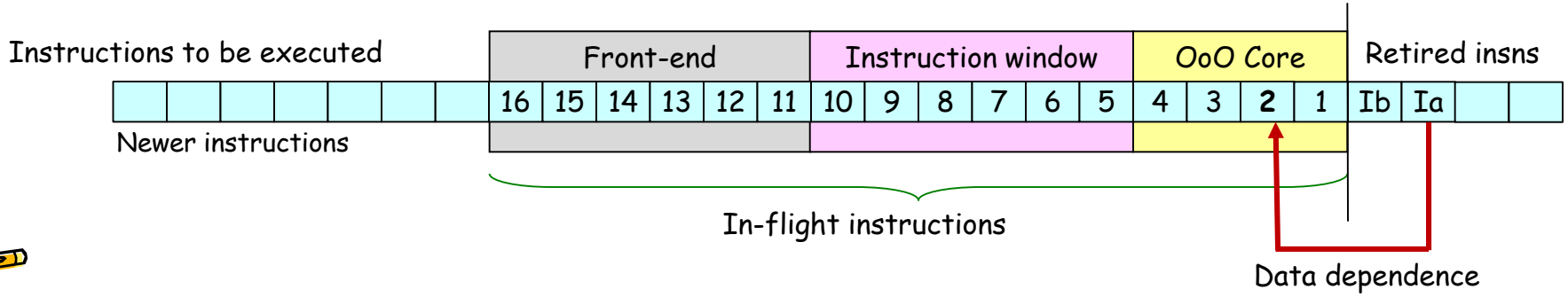


Case 1: Register dataflow from a far previous instn

- One source operand of insn I2 is from a retired instruction Ia.
- Because Ia was retired long ago, the physical destination register has been freed. The tag of the source register x3 can not be renamed at the renaming stage for I2, still having the logical register tag x3.
- Where does the operand x3 of I2 come from?

```
Ia: add x3,x0,x0
I1: sub p9,x1,x2
I2: add p10,p9,x3
I3: or  p11,x4,x5
I4: and p12,p10,p11
```

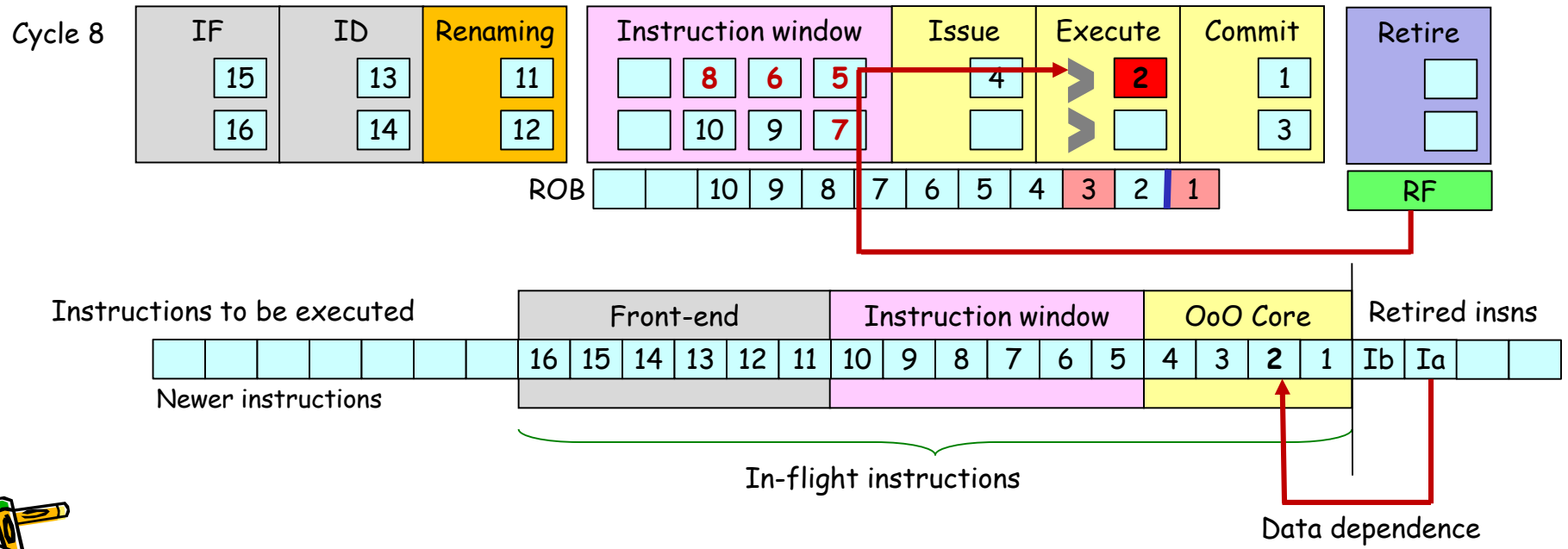




Case 1: Register dataflow from RF

- One source operand of insn I2 is from a retired instruction Ia.
- Because Ia was retired long ago, the physical destination register has been freed. The tag of the source register x3 can not be renamed at the renaming stage for I2, still having the logical register tag x3.
- Where does the operand x3 of I2 come from?

```
Ia: add x3,x0,x0
I1: sub p9,x1,x2
I2: add p10,p9,x3
I3: or  p11,x4,x5
I4: and p12,p10,p11
```

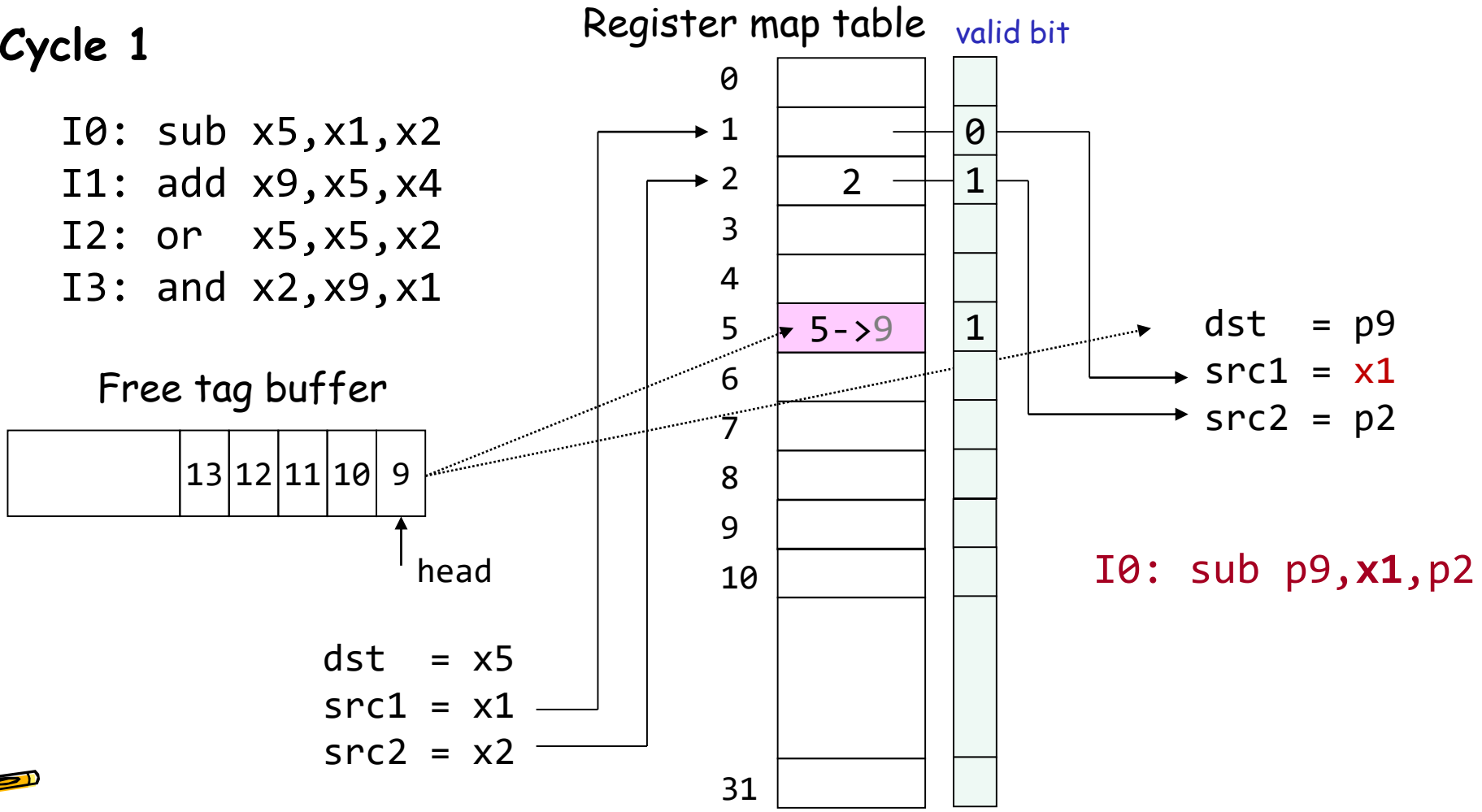


Example behavior of register renaming and valid bit

- A processor remembers a set of renamed logical registers.
- If x1 is not renamed for in-flight insn, it uses x1 instead of p1.

Cycle 1

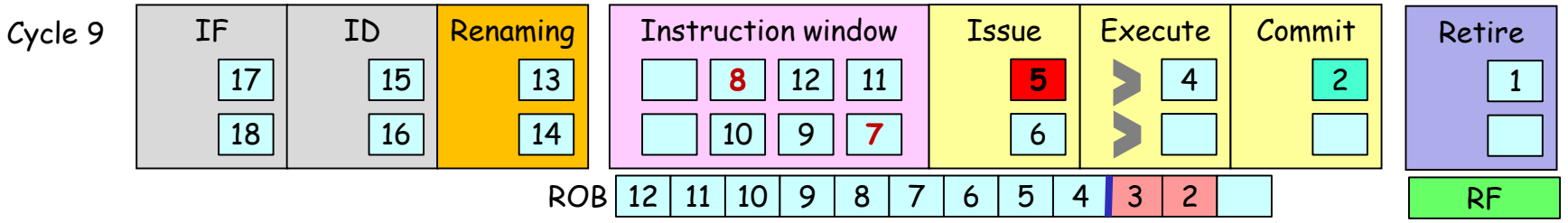
```
I0: sub x5,x1,x2
I1: add x9,x5,x4
I2: or  x5,x5,x2
I3: and x2,x9,x1
```

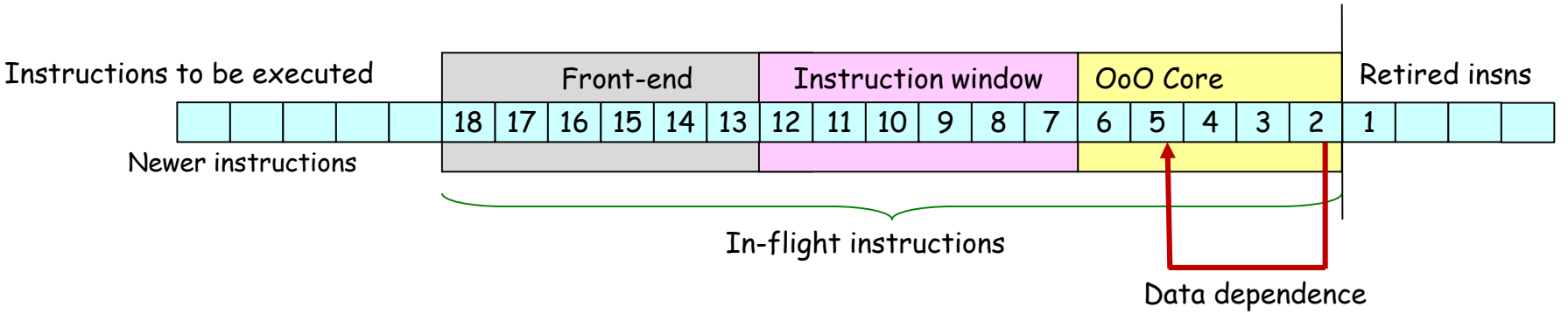


Case 2: Register dataflow

- Assume that one source operand p10 of insn I5 is from I2 which is not retired. The operand is generated a few clock cycles (tens of cycles sometimes) earlier.
- Because I2 is not retired, RF does not have the operand. Because I2 is committed, the operand is stored in ROB.
- Where does the operand of I5 come from?

```
Ia: add x3,x0,x0
I1: sub p9,x1,x2
I2: add p10,p9,x3
I3: or  p11,x4,x5
I4: and p12,p10,p11
I5: nor p13,p10,p12
```

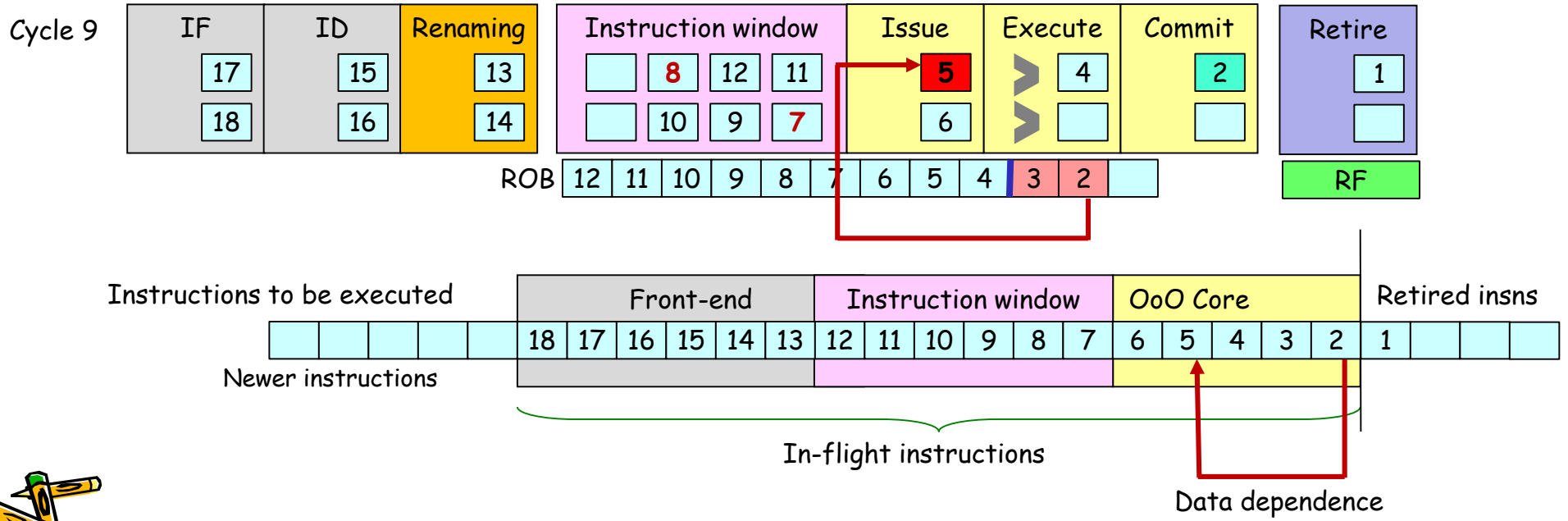




Case 2: Register dataflow from ROB

- Assume that one source operand p10 of insn I5 is from I2 which is not retired. The operand is generated a few clock cycles (tens of cycles sometimes) earlier.
- Because I2 is not retired, RF does not have the operand. Because I2 is committed, the operand is stored in ROB.
- Where does the operand of I5 come from?

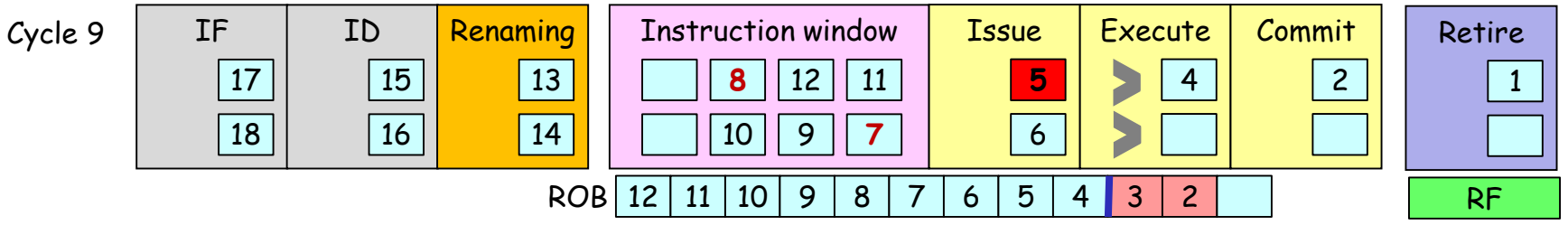
```
Ia: add x3,x0,x0
I1: sub p9,x1,x2
I2: add p10,p9,x3
I3: or  p11,x4,x5
I4: and p12,p10,p11
I5: nor p13,p10,p12
```

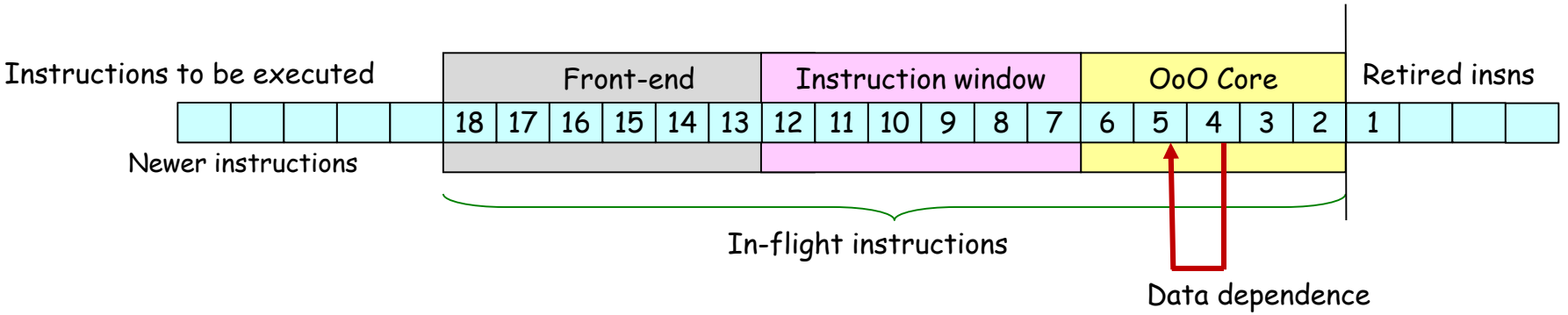


Case 3: Register dataflow

- Assume that the other source operand p12 of insn I5 is from I4 which is not committed. The operand is generated in the previous clock cycle.
- Because I4 is not retired, RF does not have the operand.
  Because I4 is not committed, ROB does not have the operand.
- Where does the operand of I5 come from?

```
Ia: add x3,x0,x0
I1: sub p9,x1,x2
I2: add p10,p9,x3
I3: or  p11,x4,x5
I4: and p12,p10,p11
I5: nor p13,p10,p12
```

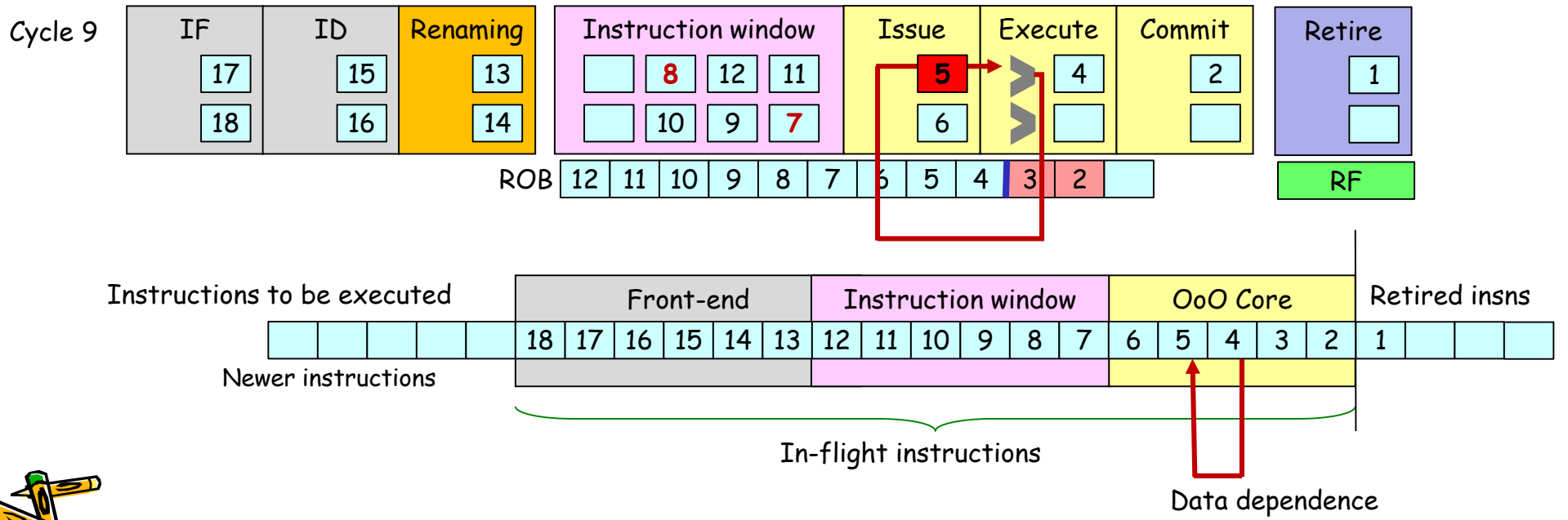




Case 3: Register dataflow from ALUs

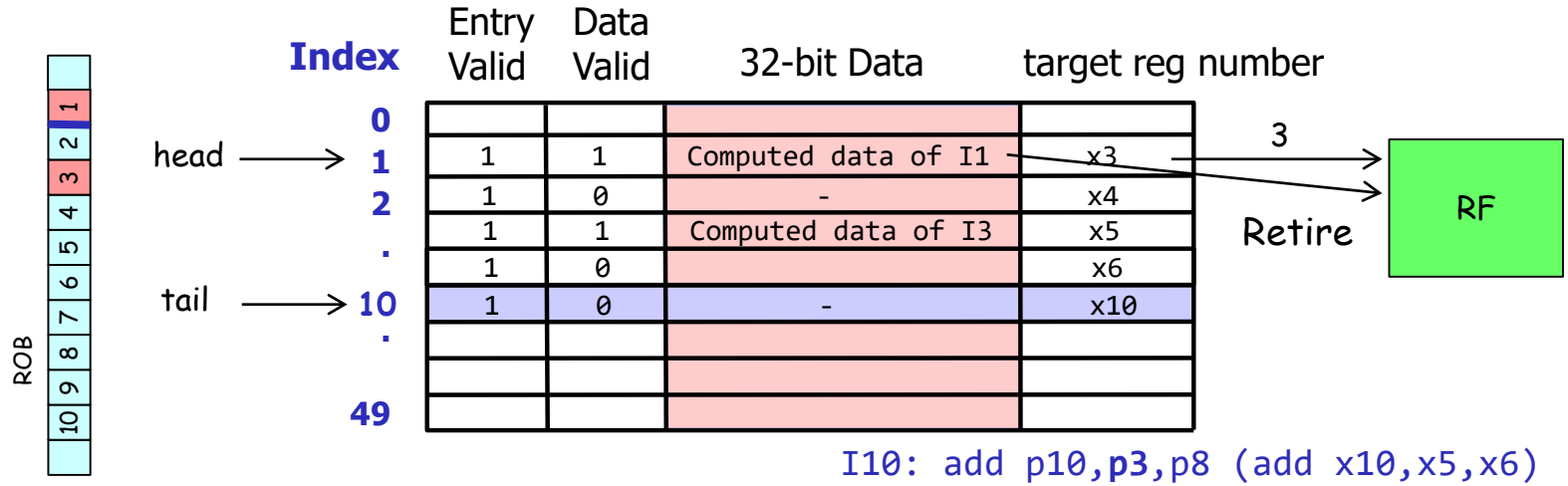
- Assume that the other source operand p12 of insn I5 is from I4 which is not committed. The operand is generated in the previous clock cycle.
- Because I4 is not retired, RF does not have the operand.
  Because I4 is not committed, ROB does not have the operand.
- Where does the operand of I5 come from?

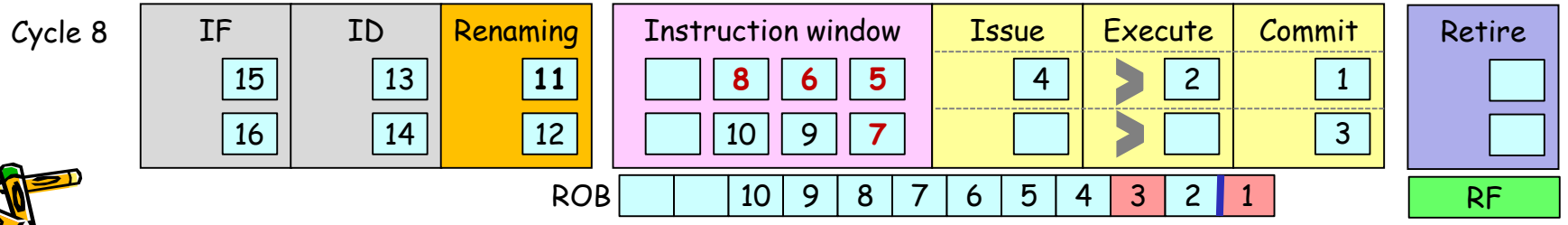
```
Ia: add x3,x0,x0
I1: sub p9,x1,x2
I2: add p10,p9,x3
I3: or  p11,x4,x5
I4: and p12,p10,p11
I5: nor p13,p10,p12
```



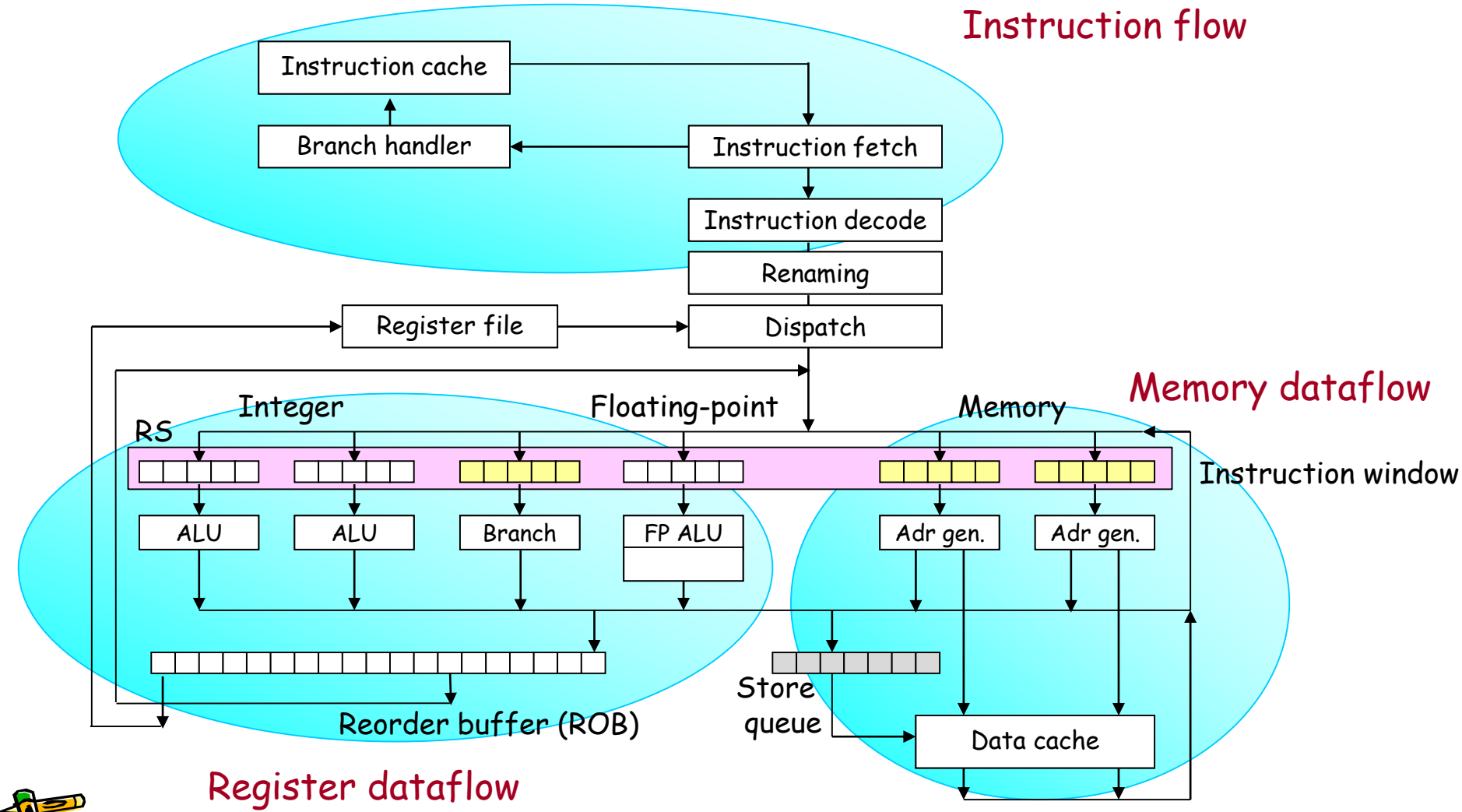
Reorder buffer (ROB)

- Each ROB entry has following fields
  - entry valid bit, data valid bit, **data**, target register number, etc.
- ROB provides the large physical registers for renaming
  - in fact, physical register number is ROB entry number
- The value of a physical register may come from a matching ROB entry



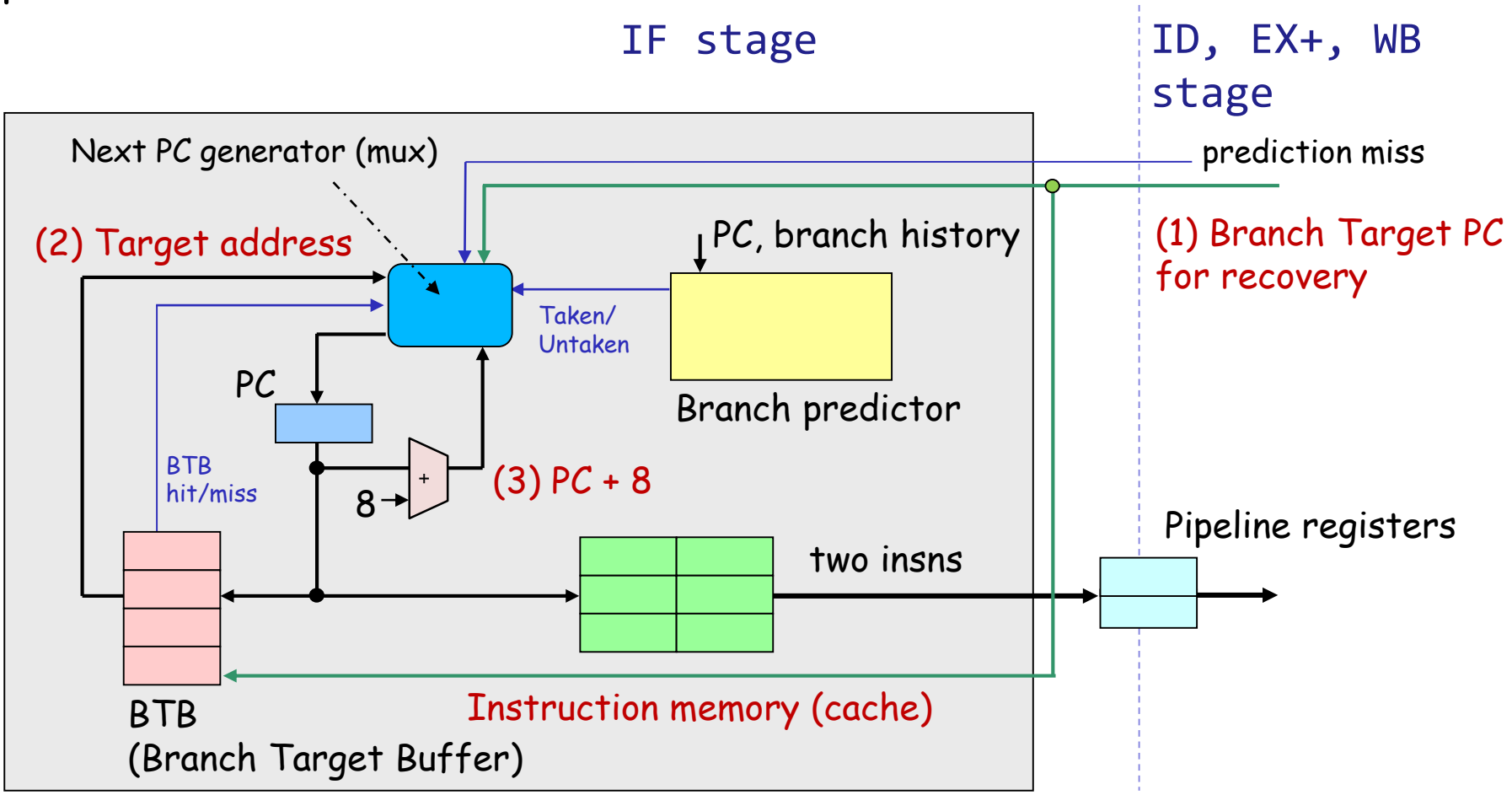


Datapath of OoO execution processor

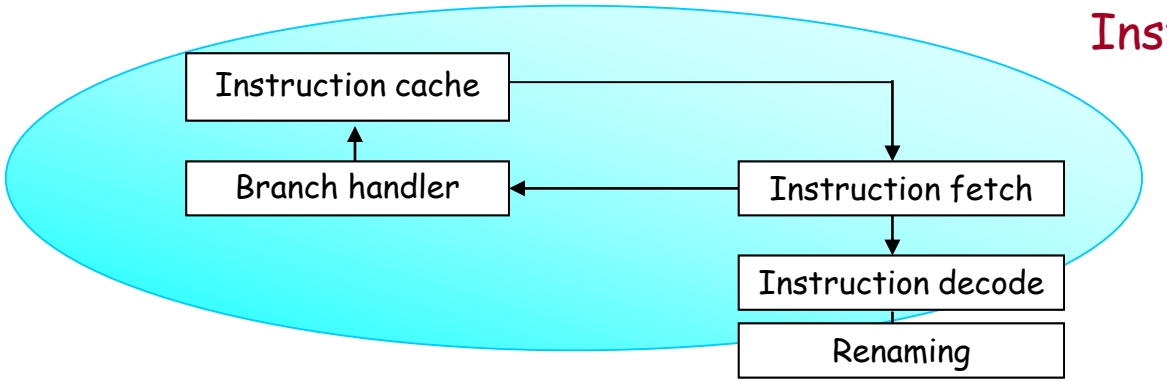


Instruction fetch unit of 2-way super-scalar

- High-bandwidth instruction delivery using branch prediction, and speculation

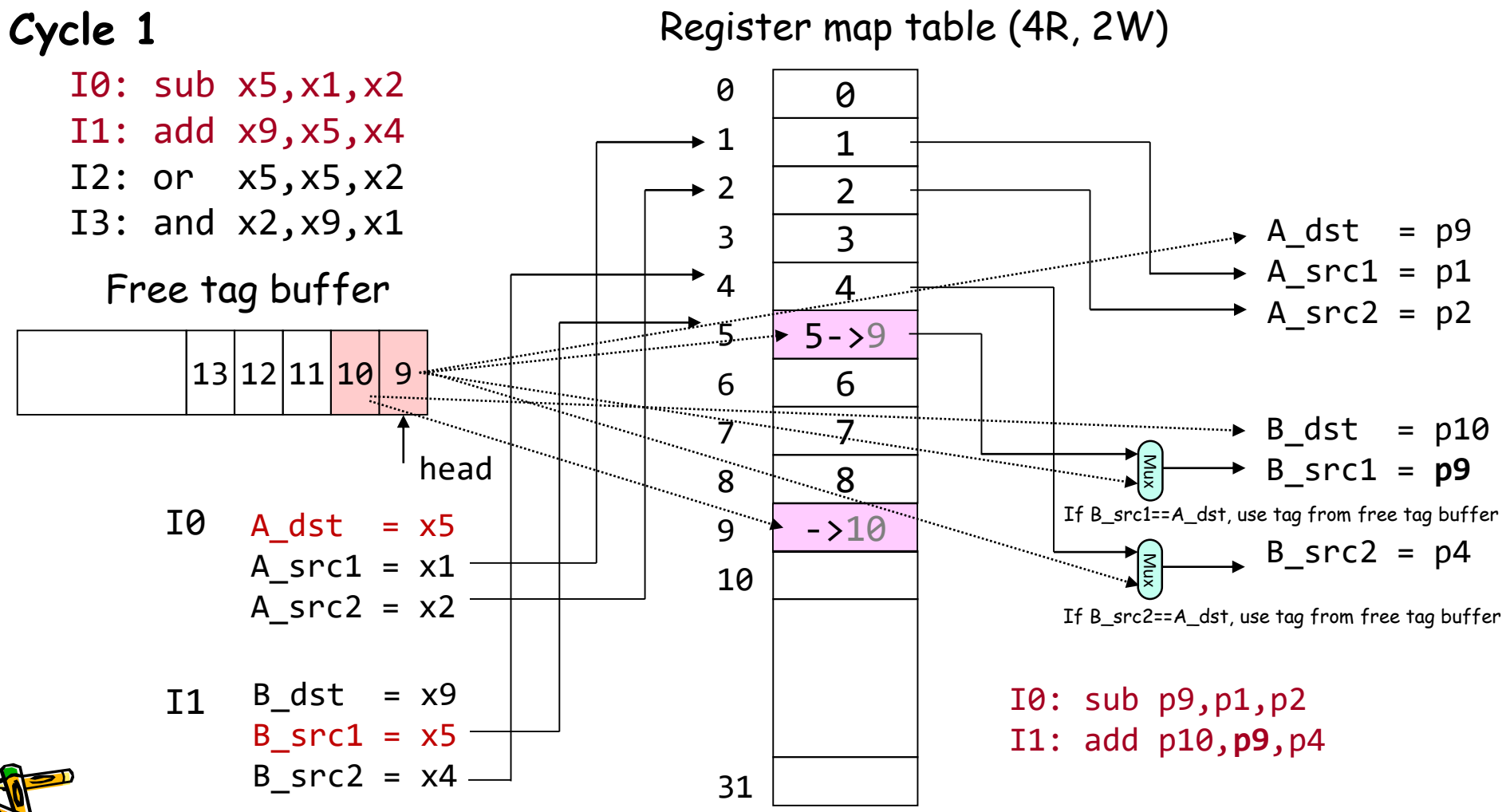


Datapath of OoO execution processor (partially)



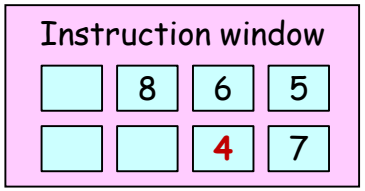
Renaming two instructions per cycle for n-way superscalar

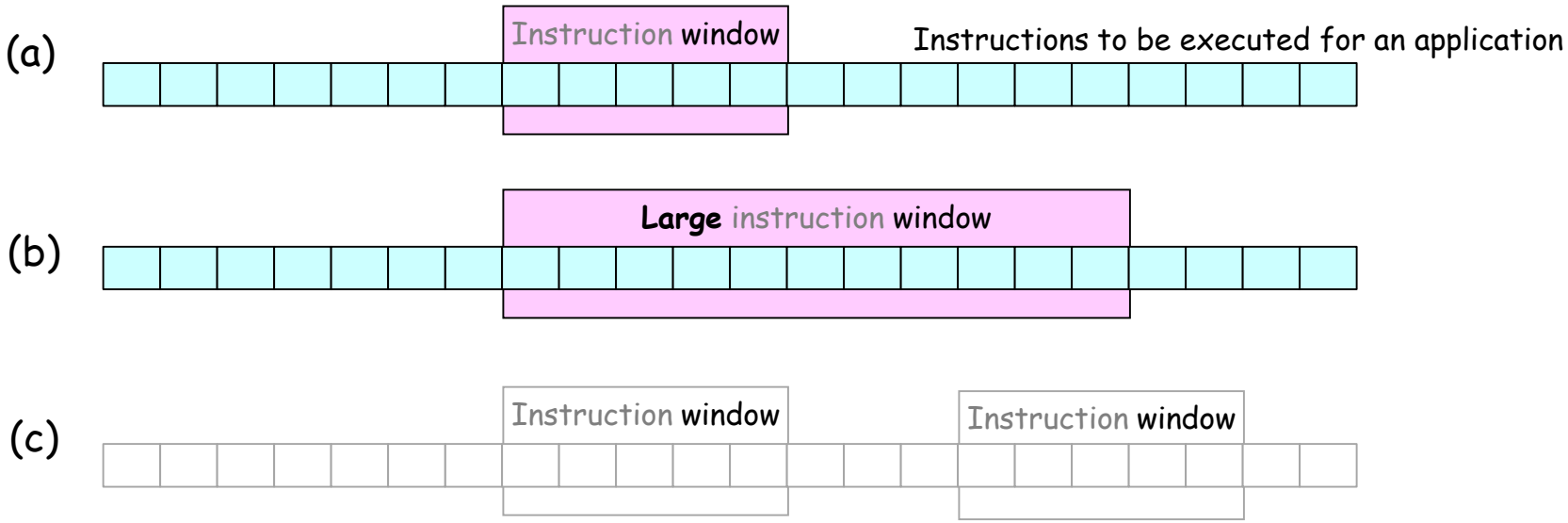
- Renaming instruction I0 and I1 (n = 2)



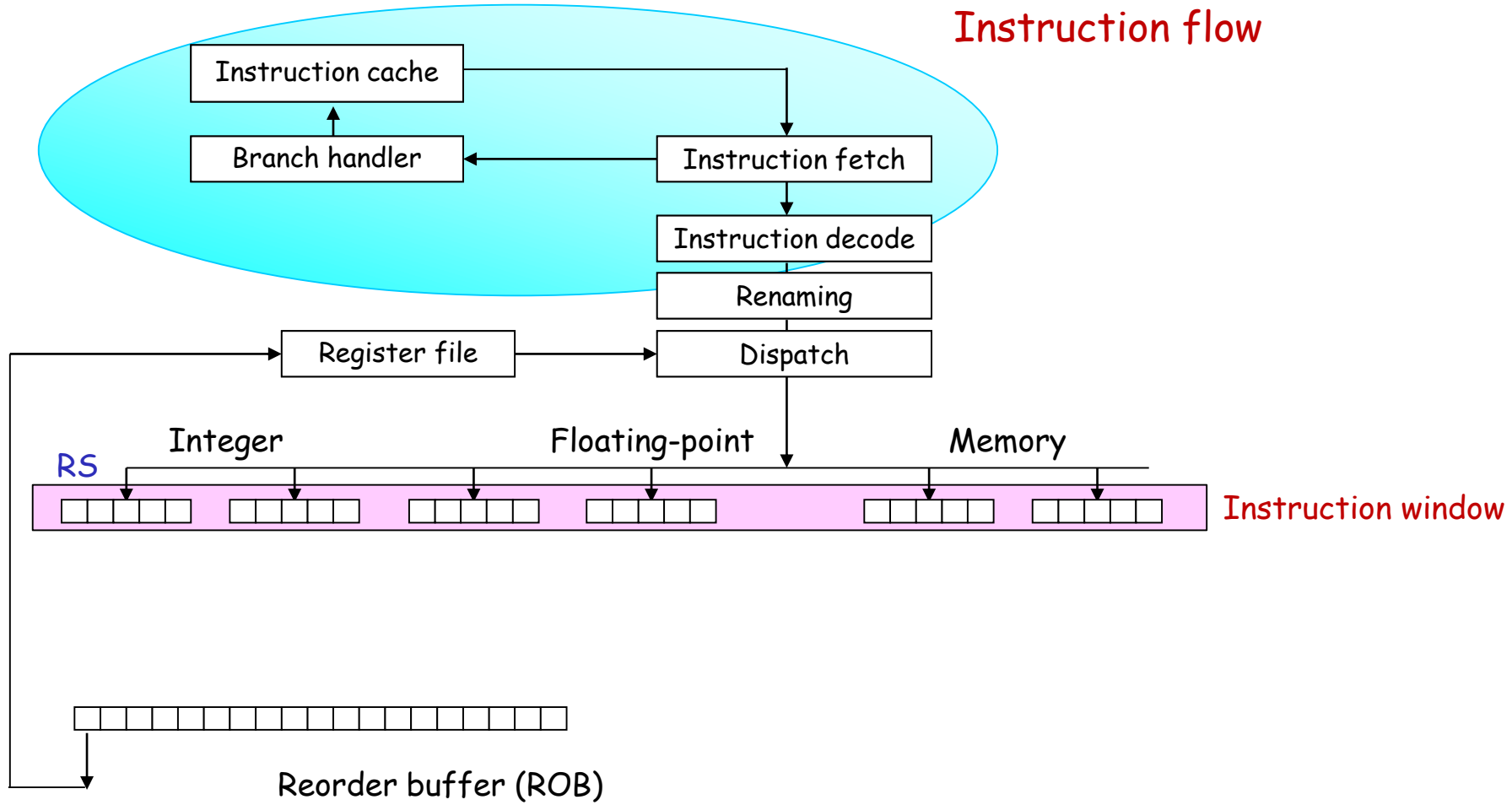
Aside: What is a window?

- A window is a space in the wall of a building or in the side of a vehicle, which has glass in it so that light can come in and you can see out. (Collins)



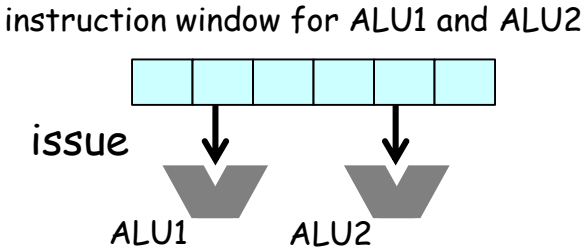


Datapath of OoO execution processor (partially)

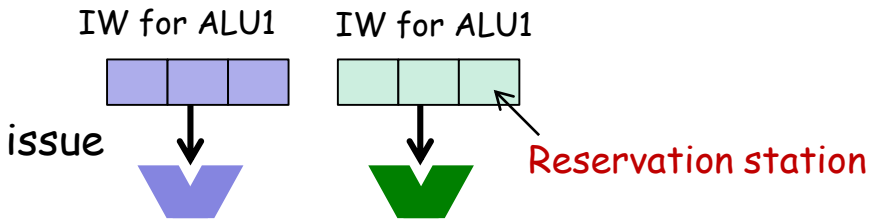


Reservation station (RS)

- To simplify the wakeup and select logic at issue stage, each functional unit (ALU) has own instruction window, an entry for such an instruction window is called reservation station (RS).
- Each reservation station has
  - valid bit, src1 tag, src1 data, src1 ready, src2 tag, src2 data, src2 ready, destination physical register number (*dst*), operation, …
  - The computed data (outcome) with its *dst* as tag is broadcasted to all RSs.

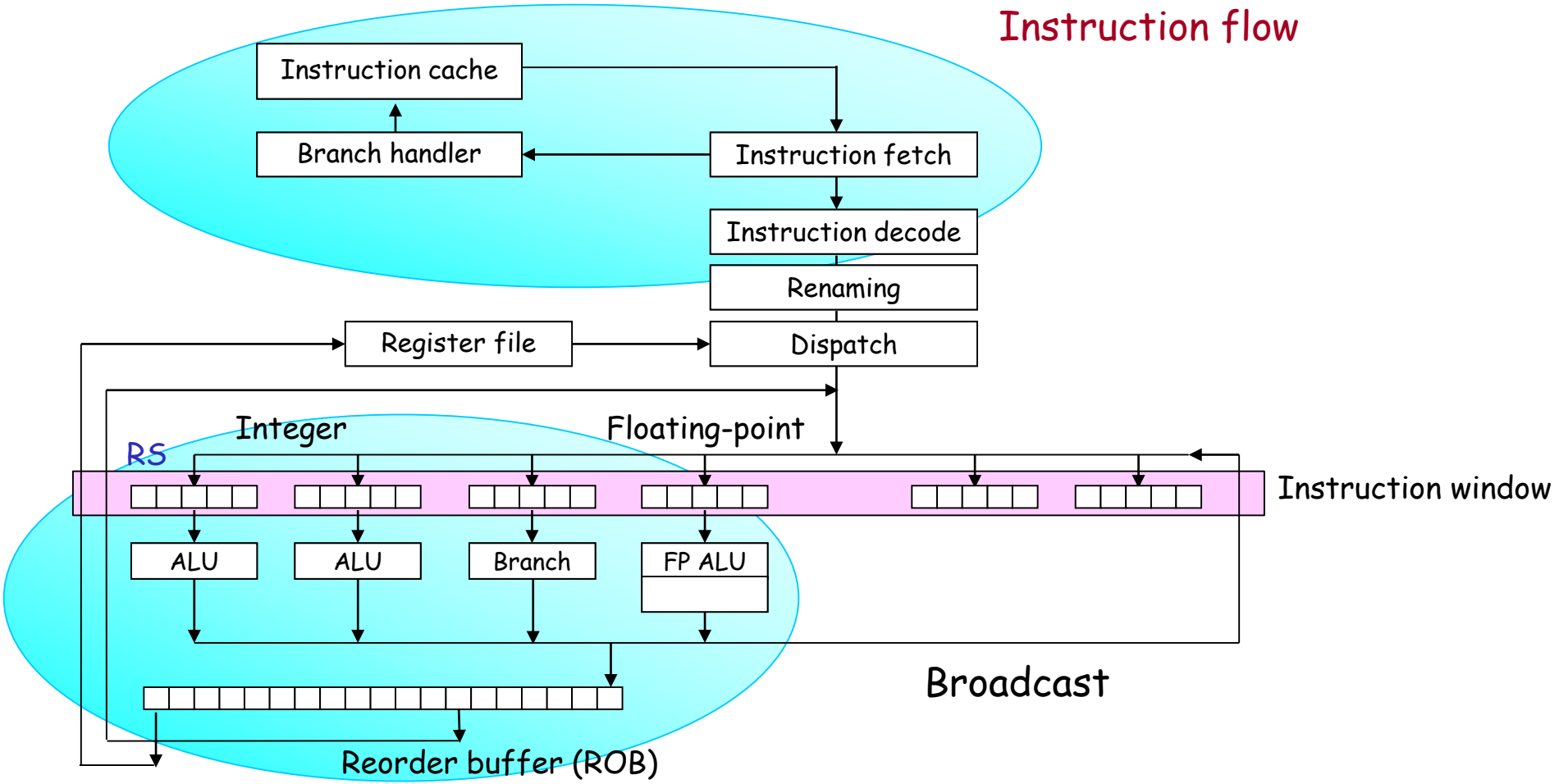


(a) Centralized instruction window



(b) Distributed instruction window using RS

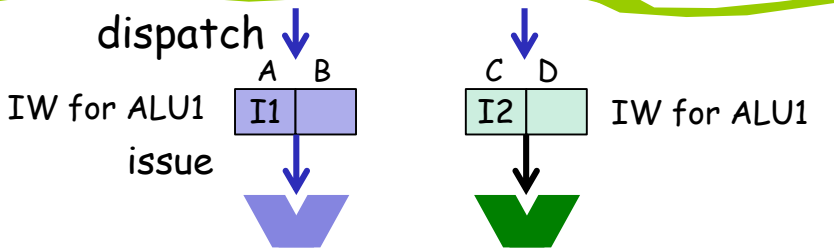
Datapath of OoO execution processor (partially)



Example behavior of reservation stations

Cycle 0

dispatch I1, I2




```
I1: sub p9,x1,x2
I2: add p10,p9,x3
I3: or  p11,x4,x5
I4: and p12,p10,p11
I5: nor p13,p10,p12
```

dispatch at most two instructions, one to A or B and the other to C or D

RS_A

valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
1	x1	value of x1	1	x2	value of x2	1	p9	I1: sub

For operand src1 For operand src2

RS_B

valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation

For operand src1 For operand src2

RS_C

valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
1	p9		0	x3	value of x3	1	p10	I2: add

For operand src1 For operand src2

RS_D

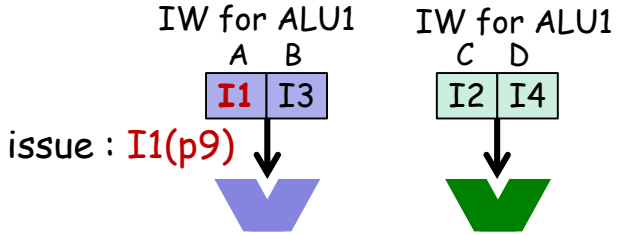
valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation

For operand src1 For operand src2

Example behavior of reservation stations

Cycle 1

dispatch I3, I4
issue I1




```
I1: sub p9,x1,x2
I2: add p10,p9,x3
I3: or  p11,x4,x5
I4: and p12,p10,p11
I5: nor p13,p10,p12
```

dispatch at most two instructions, one to A or B and the other to C or D

RS_A

valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
~~0~~	x1	value of x1	1	x2	value of x2	1	p9	I1: sub

For operand src1 For operand src2

RS_B

valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
1	x4	value of x4	1	x5	value of x5	1	p11	I3: or

For operand src1 For operand src2

RS_C

valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
1	p9		0	x3	value of x3	1	p10	I2: add

For operand src1 For operand src2

RS_D

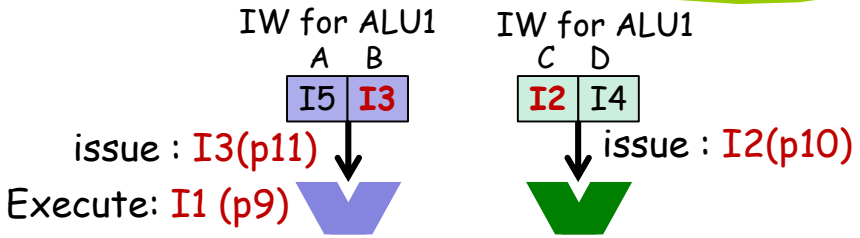
valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
1	p10		0	p11		0	p12	I4: and

For operand src1 For operand src2

Example behavior of reservation stations

Cycle 2

dispatch I5
issue I2, I3
execute I1




```
I1: sub p9,x1,x2
I2: add p10,p9,x3
I3: or  p11,x4,x5
I4: and p12,p10,p11
I5: nor p13,p10,p12
```

dispatch at most two instructions, one to A or B and the other to C or D

RS_A

valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
1	p10		0	p12		0	p13	I5: nor

For operand src1 For operand src2

RS_B

valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
~~1~~	~~x4~~	~~value of x4~~	~~1~~	~~x5~~	~~value of x5~~	~~1~~	p11	~~I3: or~~

For operand src1 For operand src2

RS_C

valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
~~1~~	~~p9~~	~~value of p9~~	~~1~~	~~x3~~	~~value of x3~~	~~1~~	p10	~~I2: add~~

For operand src1 For operand src2

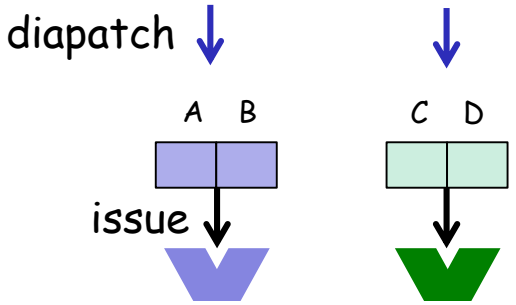
RS_D

valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
1	p10		0	p11		0	p12	I4: and

For operand src1 For operand src2

Exercise 1

- Example behavior of reservation stations

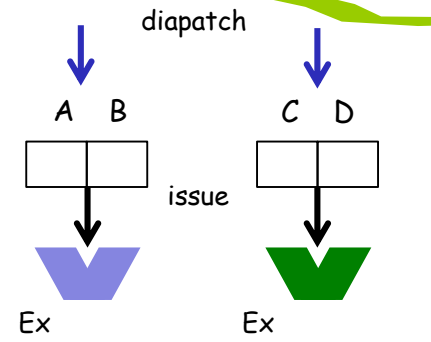



```
I1: sub p9,x1,x2
I2: add p10,p9,x3
----------------------------------------
I3: or  p11,p9,x4
I4: and p12,x5,x6
----------------------------------------
I5: nor p13,p11,p12
I6: add p14,p10,x7
----------------------------------------
I7: sub p15,p14,x1
I8: or  p16,p15,x1
```

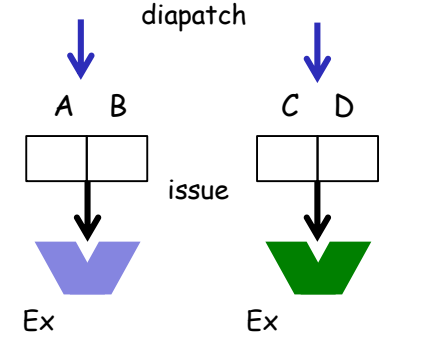
Exercise 1

Cycle 0 dispatch I1, I2

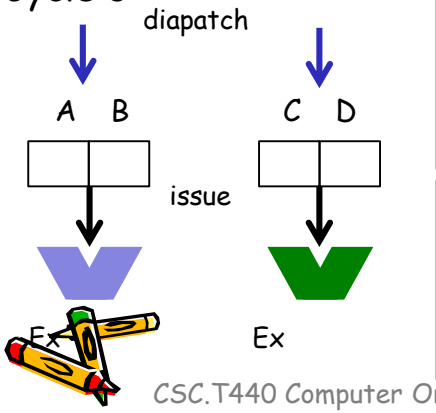
Cycle 1



Cycle 2



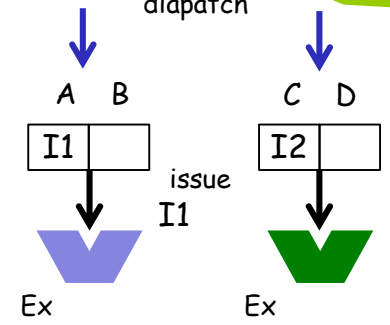
Cycle 3



Exercise 1

Cycle 0 dispatch I1, I2

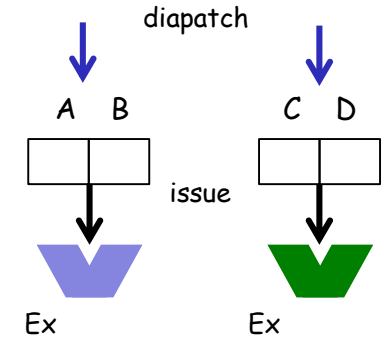
Cycle 1



	valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
RS_A	1	x1	value of x1	1	x2	value of x2	1	p9	I1:sub
RS_B									

	valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
RS_C	1	p9		0	x3	value of x3	1	p10	I2:add
RS_D									

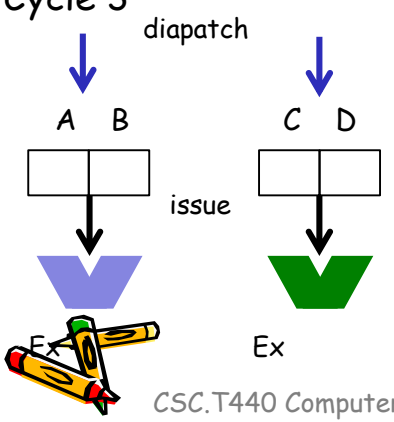
Cycle 2



	valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
RS_A									
RS_B									

	valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
RS_C									
RS_D									

Cycle 3



	valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
RS_A									
RS_B									

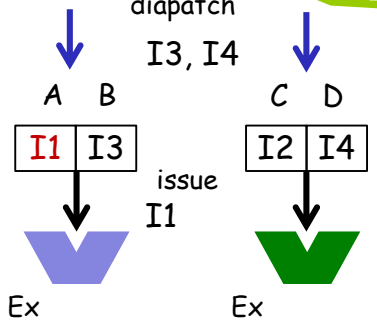
	valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
RS_C									
RS_D									

Exercise 1

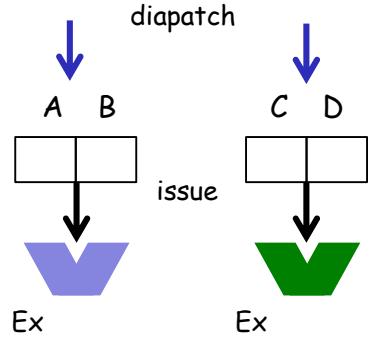
Cycle 0 dispatch I1, I2

	valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
RS_A	1	x1	value of x1	1	x2	value of x2	1	p9	I1:sub
RS_B	1	p9		0	x4	value of x4	1	p11	I3:or
RS_C	1	p9		0	x3	value of x3	1	p10	I2:add
RS_D	1	x5	value of x5	1	x6	value of x6	1	p12	I4:and

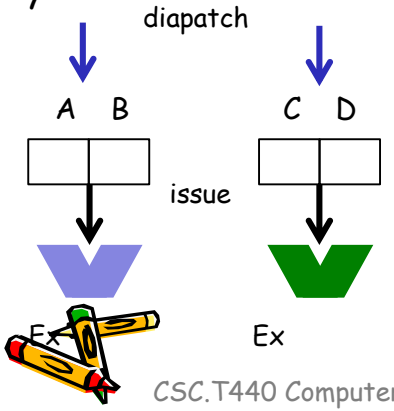
Cycle 1



Cycle 2



Cycle 3

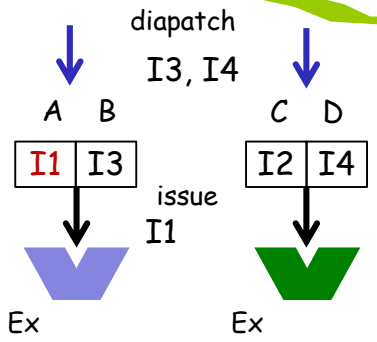


Exercise 1

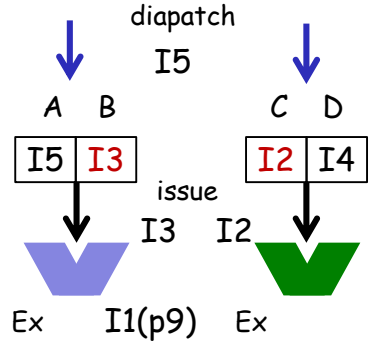
Cycle 0 dispatch I1, I2

	valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
RS_A	~~1~~	~~x1~~	~~value of x1~~	~~1~~	~~x2~~	~~value of x2~~	~~1~~	p9	I1:sub
RS_B	1	p9		0	x4	value of x4	1	p11	I3:or
RS_C	1	p9		0	x3	value of x3	1	p10	I2:add
RS_D	1	x5	value of x5	1	x6	value of x6	1	p12	I4:and

Cycle 1

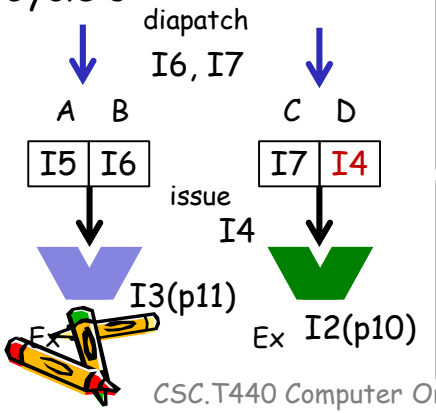


Cycle 2



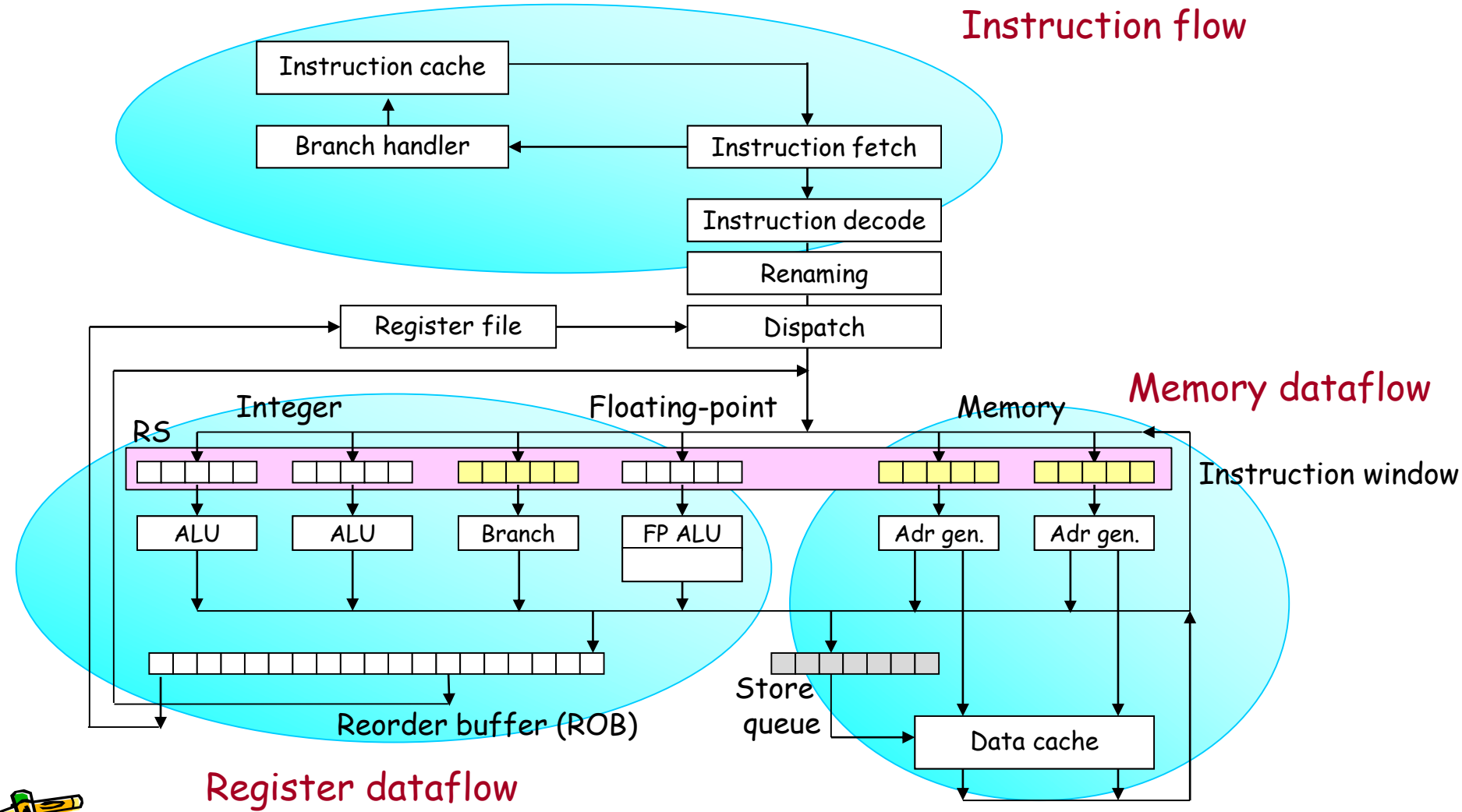
	valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
RS_A	1	p11		0	p12		0	p13	I5:nor
RS_B	~~1~~	~~x5~~	~~value of x5~~	~~1~~	~~x6~~	~~value of x6~~	~~1~~	~~p11~~	~~I3:or~~
RS_C	~~1~~	~~p9~~	~~value of p9~~	~~1~~	~~x3~~	~~value of x3~~	~~1~~	~~p10~~	~~I2:add~~
RS_D	1	x5	value of x5	1	x6	value of x6	1	p12	I4:and

Cycle 3



	valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
RS_A									
RS_B									
RS_C									
RS_D									

Datapath of OoO execution processor

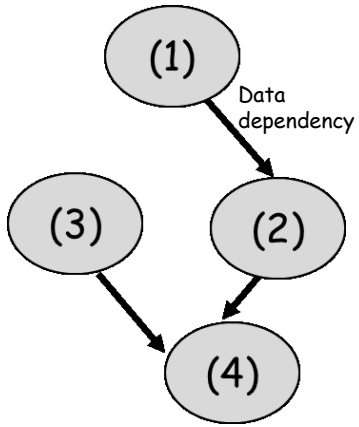


Exploiting Instruction Level parallelism (ILP)

- A superscalar has to handle some flows efficiently to exploit ILP
  - Control flow (control dependence)
    - To execute ***n*** instructions per clock cycle, the processor has to fetch at least ***n*** instructions per cycle.
    - The main obstacles are branch instruction (BNE, BEQ, ...)
    - Branch prediction
    - Another obstacle is instruction cache
  - Register data flow (data dependence)
    - Out-of-order execution
      - Register renaming
      - Dynamic scheduling
  - Memory data flow
    - Out-of-order execution
    - Another obstacle is data cache

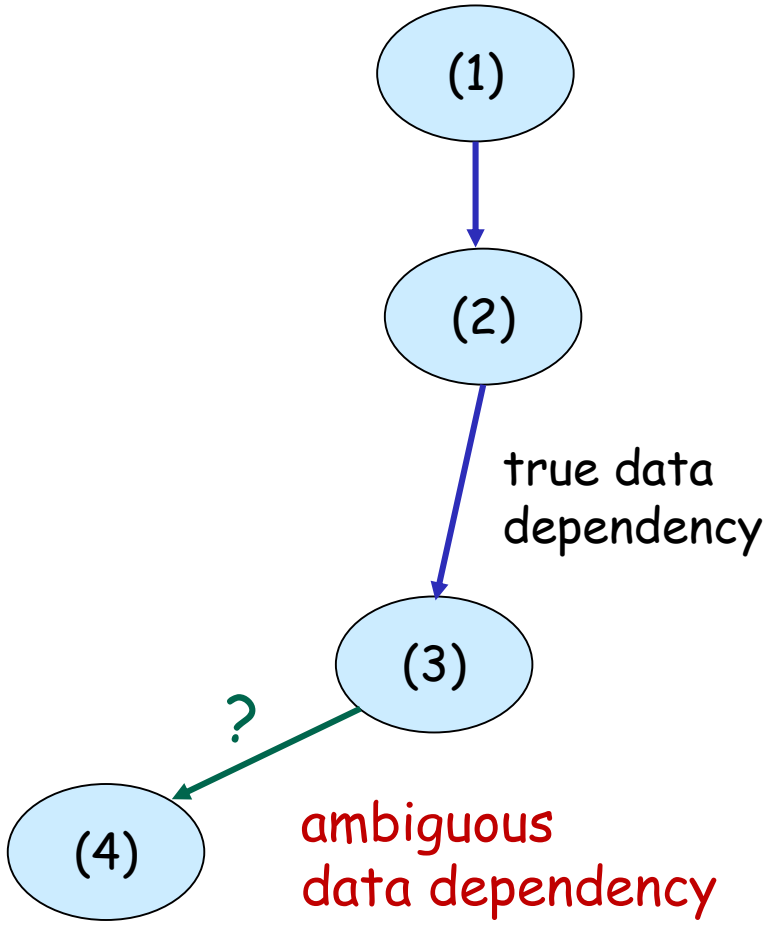
```
(1) add x5,x1,x2
(2) add x9,x5,x3
(3) lw  x4, 4(x7)
(4) add x8,x9,x4
```

```
(3) lw  x4, 4(x7)
(1) add x5,x1,x2
(2) add x9,x5,x3
(4) add x8,x9,x4
```



Instruction Level Parallelism (ILP)

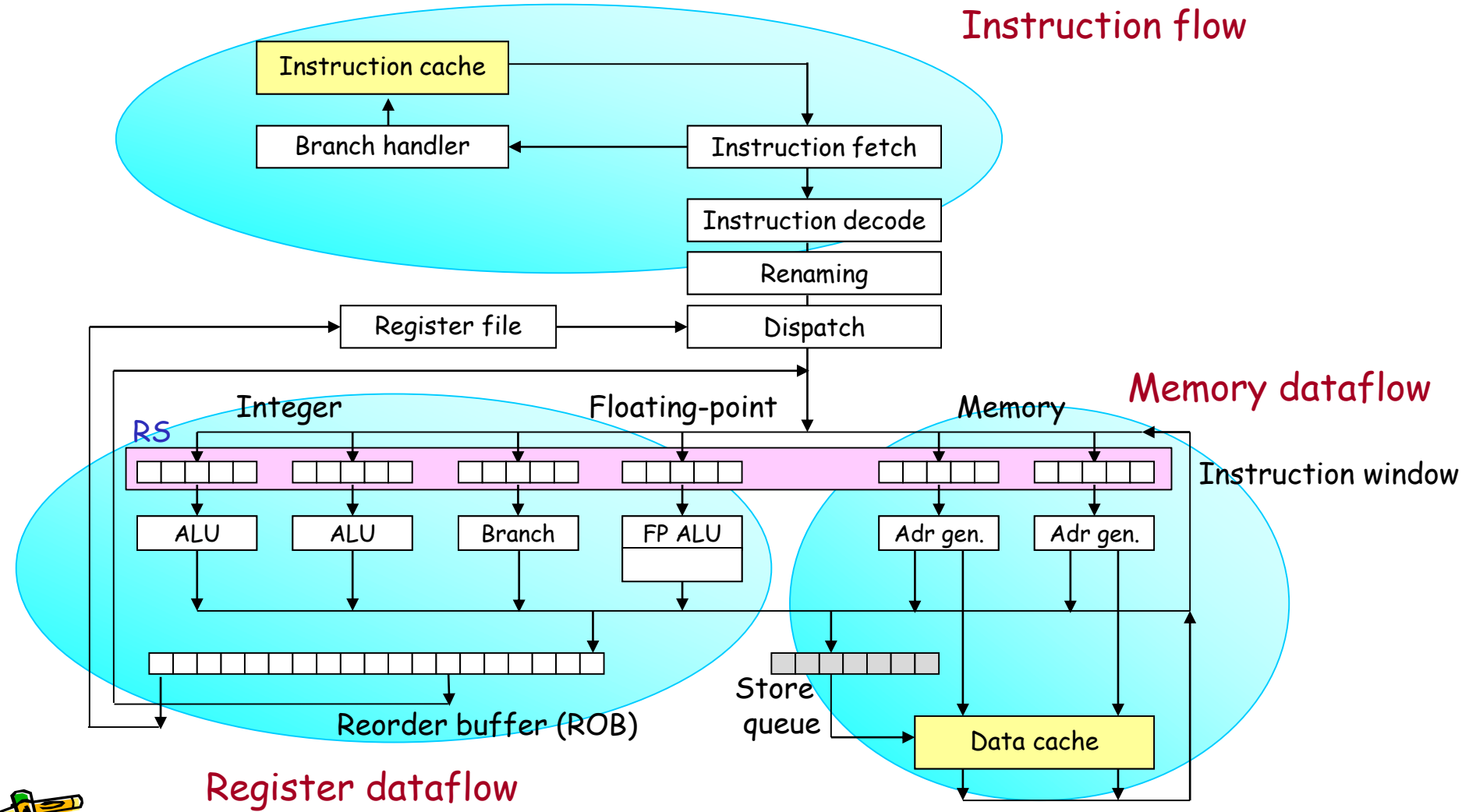
```
lw     x5, 0(x2)      (1)
addi   x6, x5, 4      (2)
sw     x6, 0(x3)      (3)
          ?
lw     x7, 0(x4)      (4)
```



Memory dataflow and branches

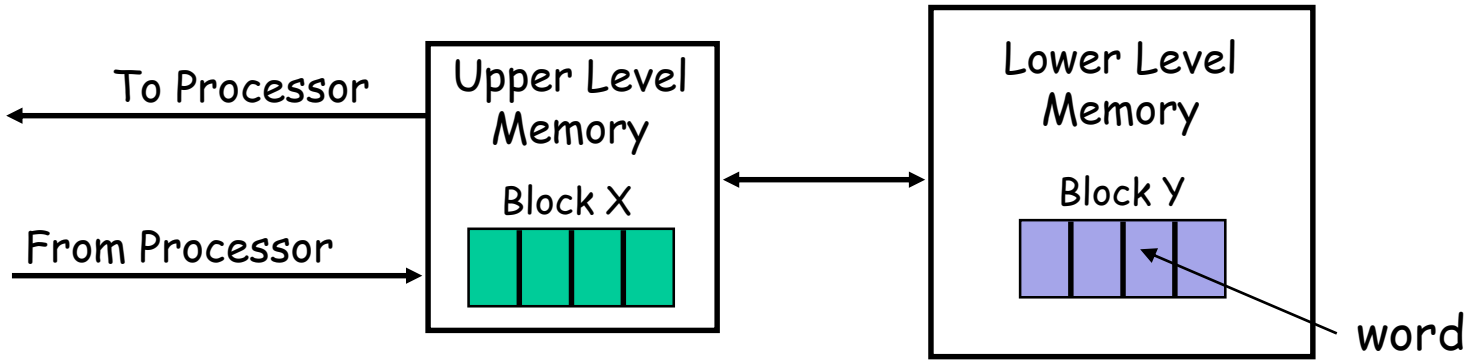
- The update of a data cache cannot be recovered easily.
  So, cache update is done at the retire stage in-order manner by using store queue.
  Because of the ambiguous memory dependency, load and store instructions can be executed in-order manner.
  - About 30% (or less) of executed instructions are load and stores.
  - Even if they are executed in-order, IPC of 3 can be achieved.
- Branch instructions can be executed in-order manner.
  - About 20% (or less) of executed instructions are jump and branch instructions.
  - Out-or-order branch execution and aggressive miss recovery may cause false recovery (recovery by a branch on the false control path).

Datapath of OoO execution processor



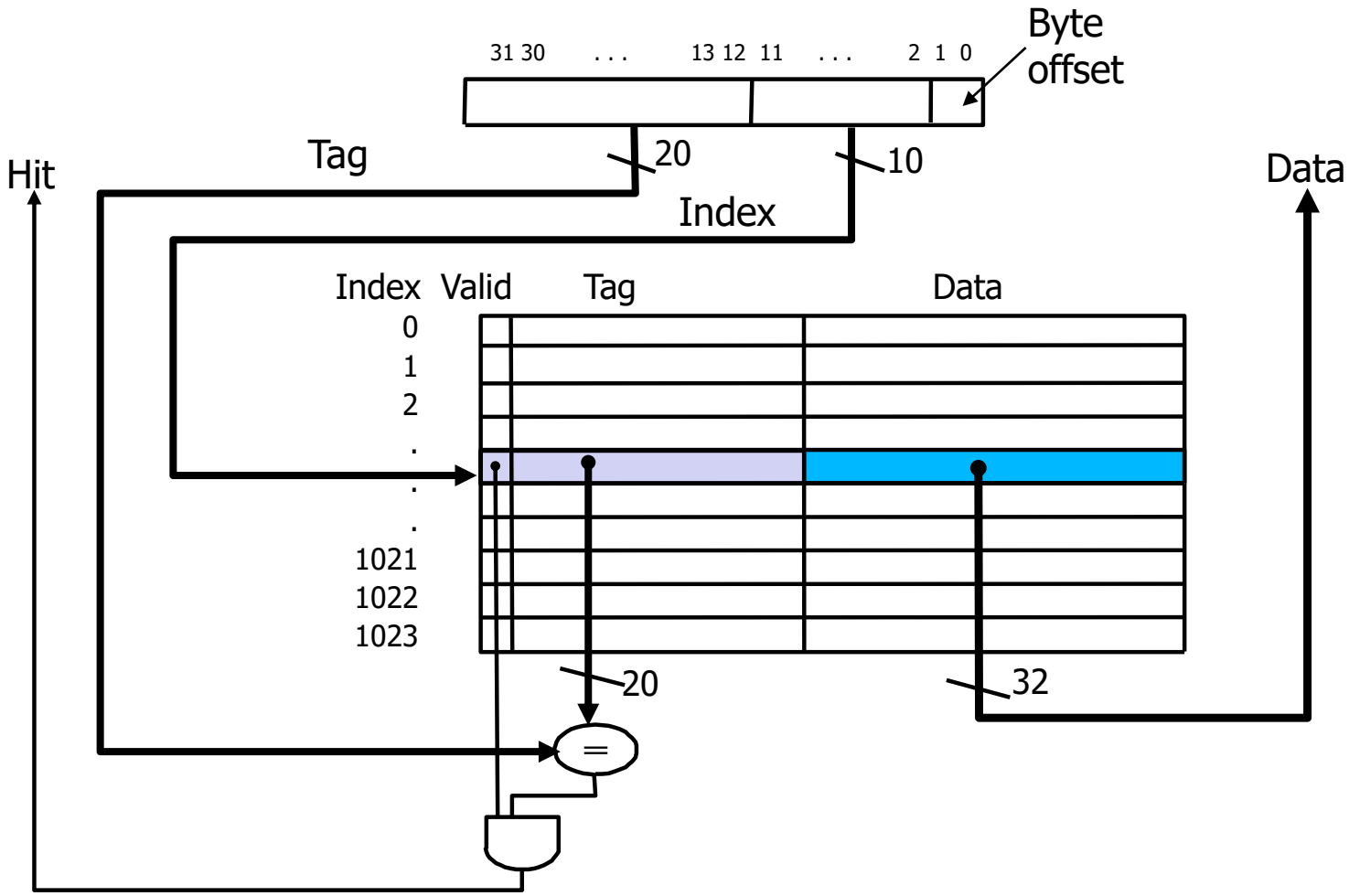
The Memory System's Fact and Goal

- Fact:
  Large memories are slow, and fast memories are small
- How do we create a memory that gives the illusion of being large, fast, and cheap?
- Cache memories
  - Temporal locality (Locality in Time):
    - Keep most recently accessed data items closer to the processor
  - Spatial locality (Locality in Space)
    - Move blocks consisting of contiguous words to the upper levels



Direct Mapped Cache Example

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



Example Behavior of 4-entry Direct Mapped Cache

- Consider the main memory word reference string (word addresses) 0 1 2 3 4 3 4 15

Start with an empty cache - all blocks initially marked as not valid

Tag

0 miss

00	Mem(0)

1 miss

00	Mem(0)
00	Mem(1)

2 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)

3 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 miss

01 ~~00~~ 4

~~00~~	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

3 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

15 miss

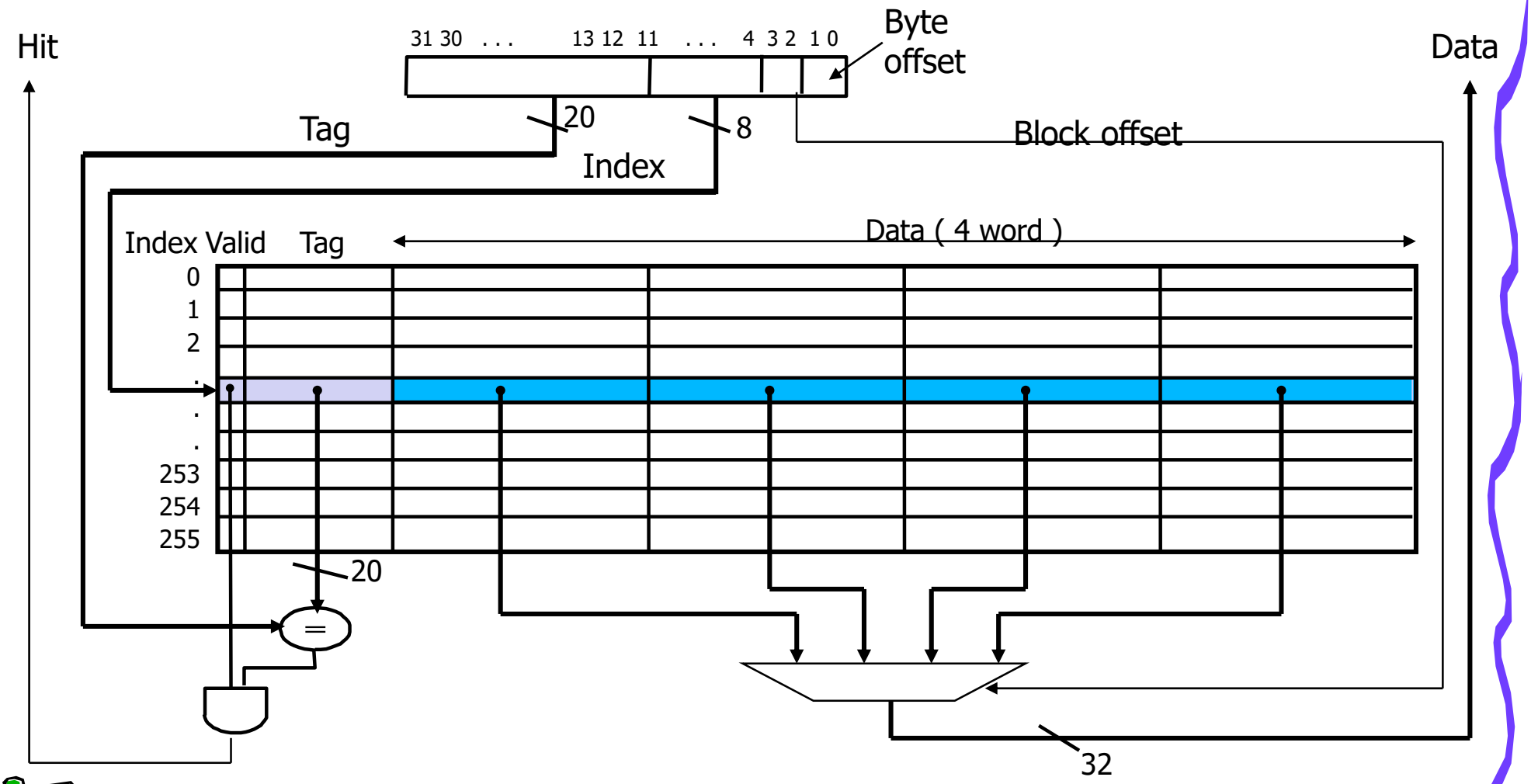
11 ~~00~~ Mem(3) 15

01	Mem(4)
00	Mem(1)
00	Mem(2)
~~00~~	Mem(3)

- 8 requests, 6 misses

Multiword Block Direct Mapped Cache

- Four words/block, cache size = 1K words



Taking Advantage of Spatial Locality

- Let cache block hold more than one word (two words/block)
  - 0 1 2 3 4 3 4 15

0 miss

00	Mem(1)	Mem(0)

1 hit

00	Mem(1)	Mem(0)

2 miss

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

3 hit

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

4 miss

~~00~~ 01	~~Mem(1)~~ 5	~~Mem(0)~~ 4
00	Mem(3)	Mem(2)

3 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

4 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

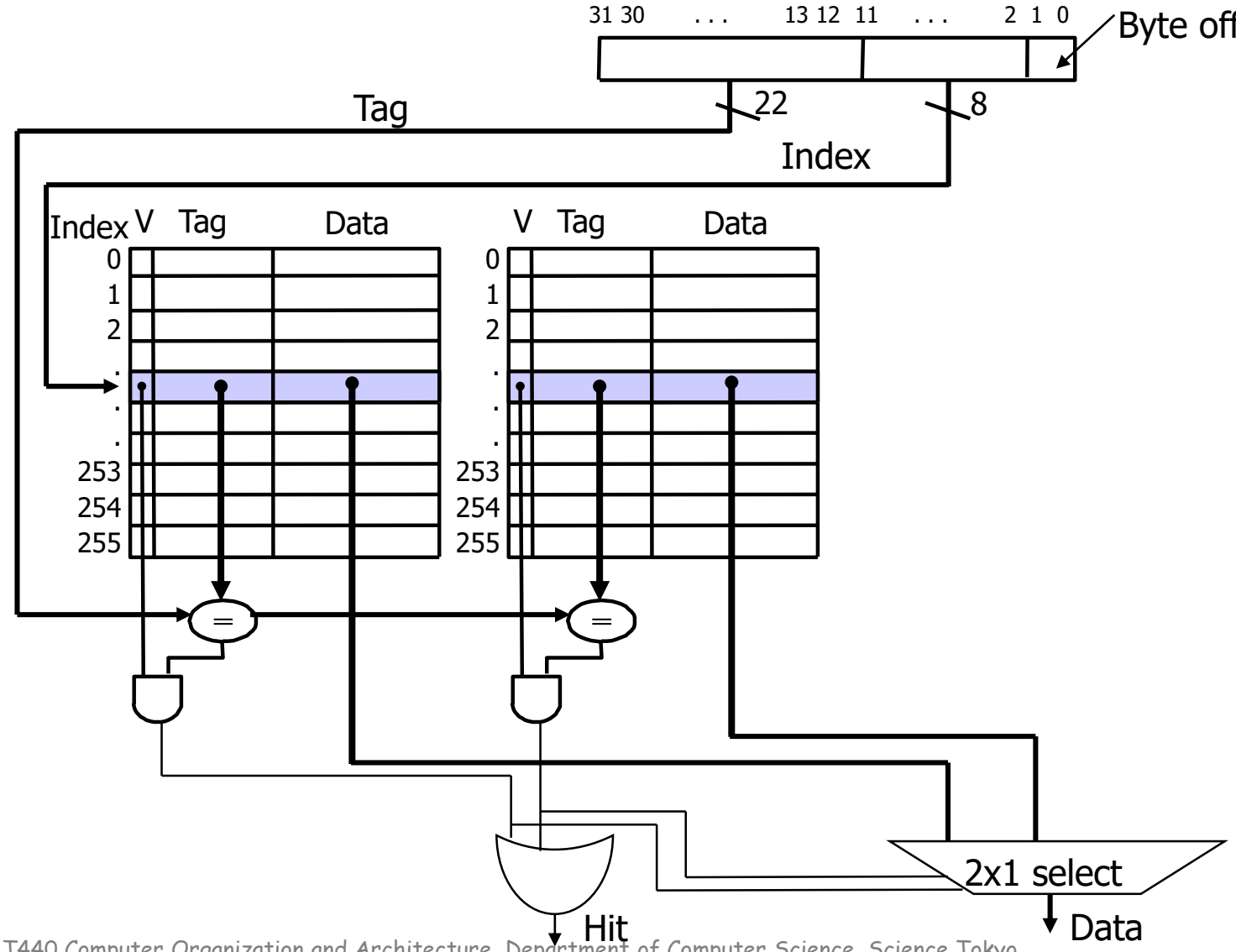
15 miss

~~00~~ 11	~~Mem(3)~~ 15	~~Mem(2)~~ 14
01	Mem(5)	Mem(4)

- 8 requests, 4 misses

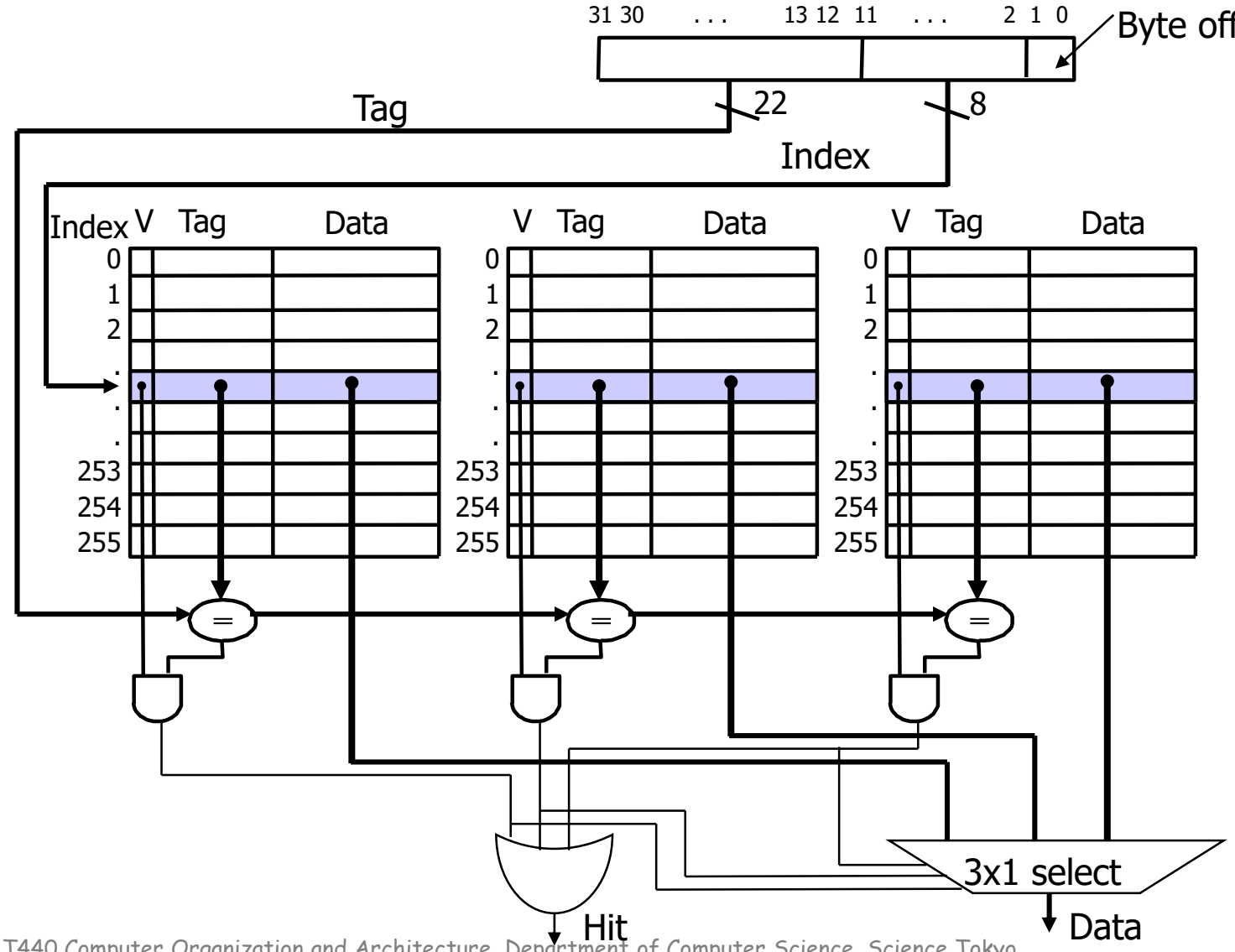
Two-Way Set Associative Cache

- One word/block, 2^8 = **256 sets** where each with three ways (each with one block)



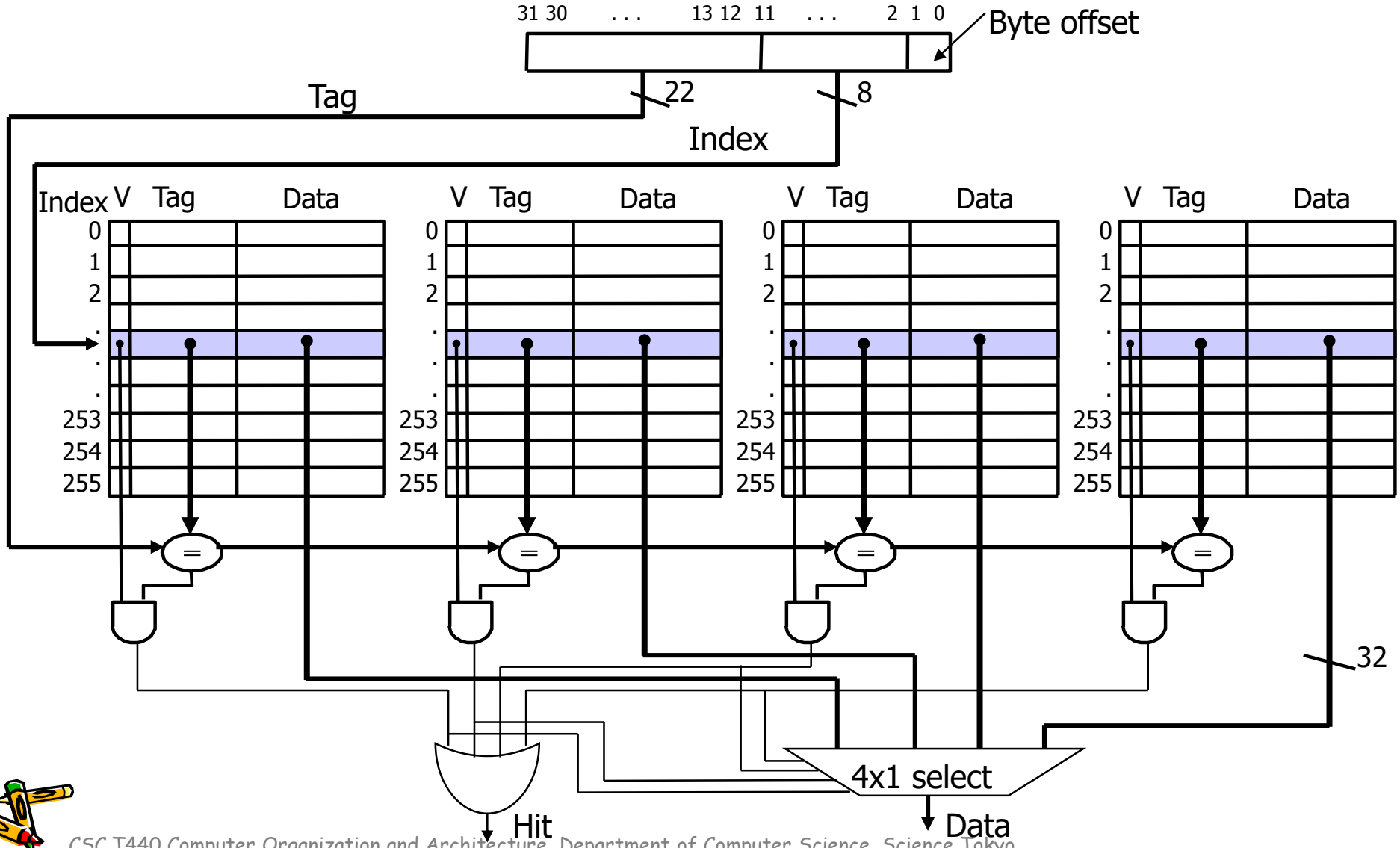
Three-Way Set Associative Cache

- One word/block, 2^8 = **256 sets** where each with three ways (each with one block)



Four-Way Set Associative Cache

- One word/block, 2^8 = **256 sets** where each with four ways (each with one block)



Set Associative Caches

- When a miss occurs, which way's block do we pick for replacement ?
  - Least Recently Used (LRU):
    the block replaced is the one that has been unused for the longest time
    - Must have hardware to keep track of when each way's block was used
    - For 2-way set associative, takes one bit per set → set the bit when a block is referenced
      (and reset the other way's bit)
  - Random
- N-way set associative cache costs
  - N comparators (delay and area)
  - MUX delay (set selection) before data is available
  - Data available after set selection and Hit/Miss decision.

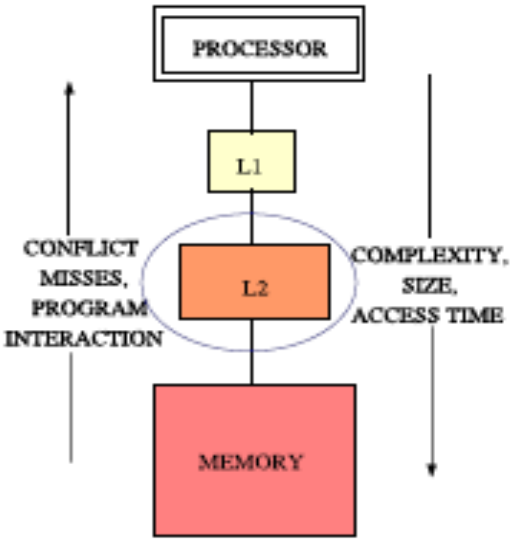
Recommended Reading

- Emulating Optimal Replacement with a Shepherd Cache
  - Kaushik Rajan, Govindarajan Ramaswamy, Indian Institute of Science
  - MICRO-40, pp. 445-454, 2007
  - Session 8: Cache Replacement Policies

- *A quote:*
  "The inherent temporal locality in memory accesses is filtered out by the L1 cache. As a consequence, an L2 cache with LRU replacement incurs significantly higher misses than the optimal replacement policy (OPT). We propose to narrow this gap through a novel replacement strategy that mimics the replacement decisions of OPT."

Memory Hierarchy Design

Memory Hierarchy



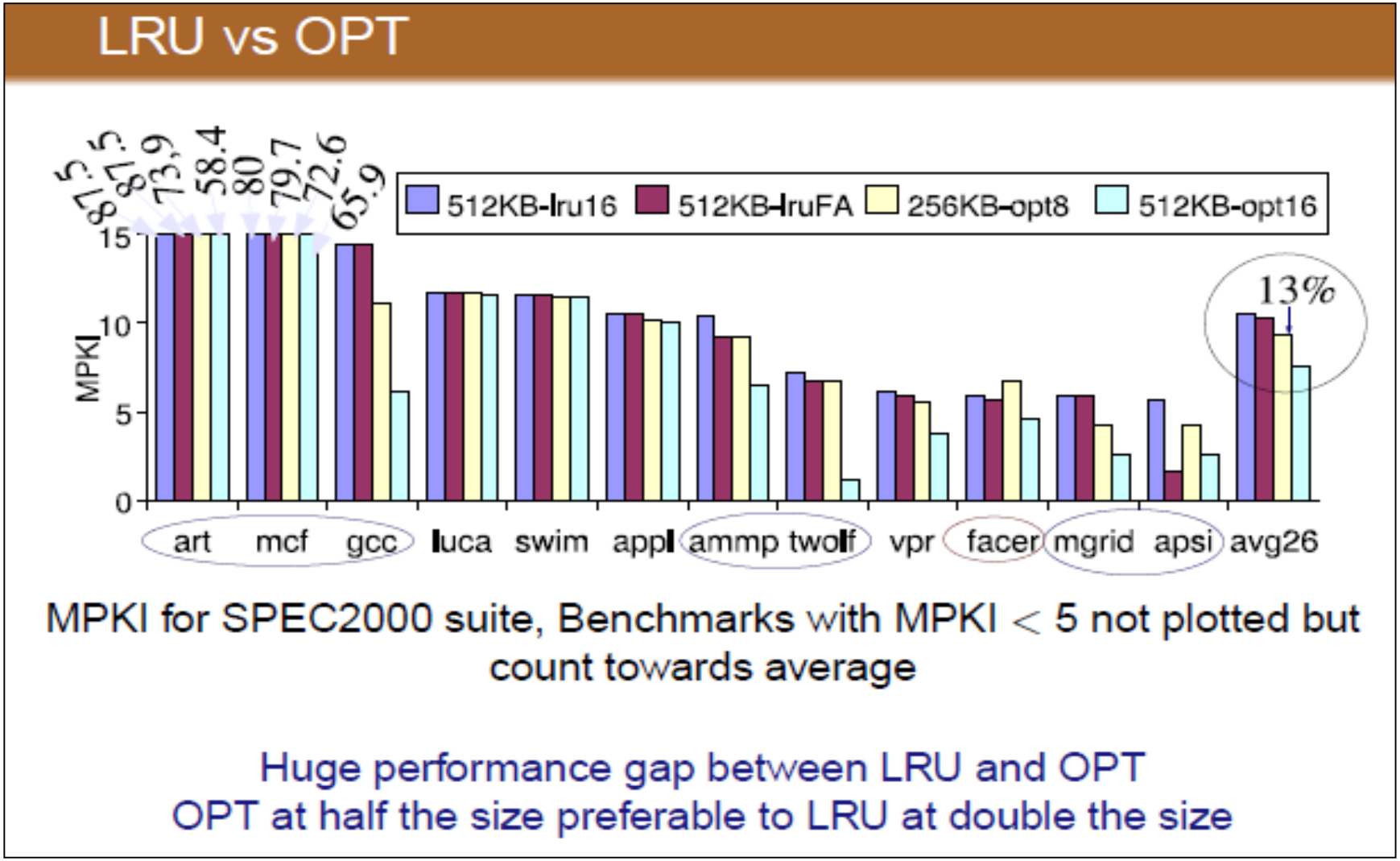
L2 and lower caches

- Objective : Need to reduce expensive memory accesses
- Design : Large size, Higher associativity, Complex design
- Problem : Do not interact with program directly and observe filtered temporal locality

- High Associativity $\Longrightarrow$ replacement policy crucial to performance
- L1 cache services temporal accesses $\Longrightarrow$ Lack of temporal accesses at L2 $\Longrightarrow$ LRU replacement inefficient
- Replacement decisions are taken off the processor critical path

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

LRU has room for improvement



MPKI: Miss Per Kilo Instructions

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

OPT: Optimal Replacement Policy

The Optimal Replacement Policy

1. Replacement Candidates : On a miss any replacement policy could either choose to replace any of the lines in the cache or choose not to place the miss causing line in the cache at all.
2. Self Replacement : The latter choice is referred to as a self-replacement or a cache bypass

Optimal Replacement Policy

On a miss replace the candidate to which an access is least imminent [Belady1966,Mattson1970,McFarling-thesis]

3. Lookahead Window : Window of accesses between miss causing access and the access to the least imminent replacement candidate. Single pass simulation of OPT make use of lookahead windows to identify replacement candidates and modify current cache state [Sugumar-SIGMETRICS1993]

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

Example of Optimal Replacement Policy

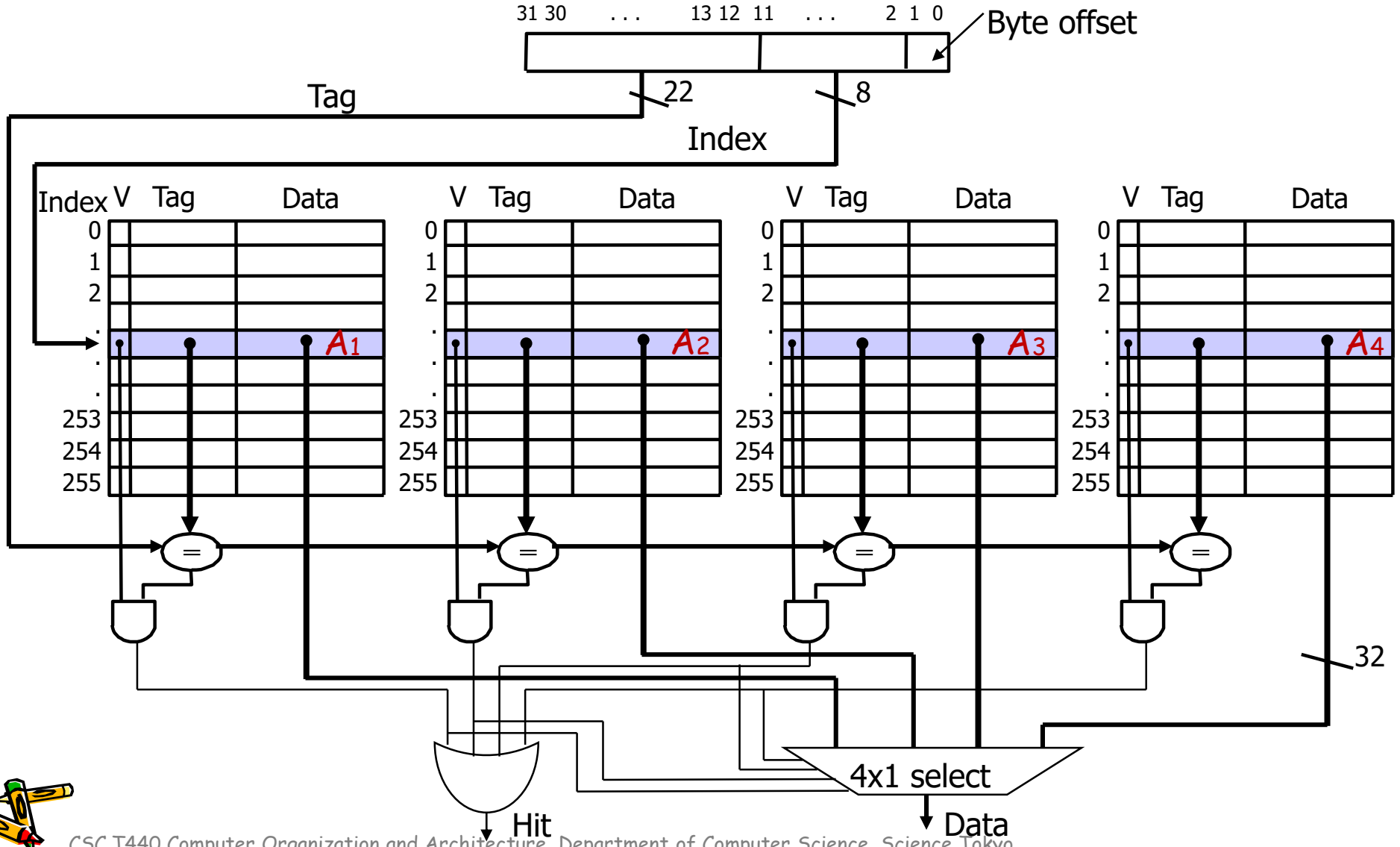
Understanding OPT

Access Sequence	A_5	A_1	A_6	A_3	A_1	A_4	A_5	A_2	A_5	A_7	A_6	A_8
OPT order for A_5		0		1		2	3	4				
OPT order for A_6				0	1	2	3				4	

- Consider 4 way associative cache with one set initially containing lines (A_1, A_2, A_3, A_4), consider the access stream shown in table
- Access A_5 misses, replacement decision proceeds as follows
  1. Identify replacement candidates : $(A_1, A_2, A_3, A_4, A_5)$
  2. Lookahead and gather imminence order : shown in table, lookahead window circled
  3. Make replacement decision : A_5 replaces A_2
- A_6 self-replaces, lookahead window and imminence order in table

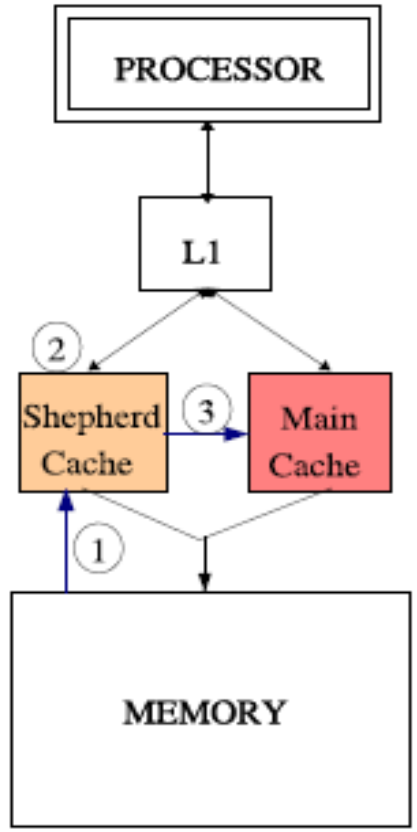
Four-Way Set Associative Cache

- One word/block, 2^8 = **256 sets** where each with four ways (each with one block)



Shepherd Cache emulation OPT

Emulating OPT with a Shepherd Cache

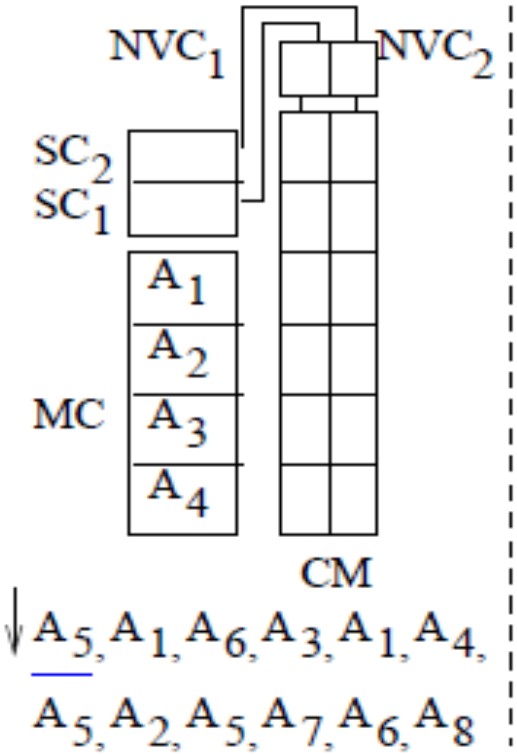


- Split the cache into two logical parts
  - Main Cache (MC) for which optimal replacement is emulated
  - Shepherd Cache (SC) used to provide a lookahead and guide replacements from MC towards OPT
- Operation
  1. Buffer lines temporarily in SC before moving them to MC, SC acts as a FIFO buffer
  2. While in SC, gather imminence information and emulate lookahead
  3. When forced out of SC, make an MC replacement based on the gathered imminence order

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

Shepherd Cache Overview

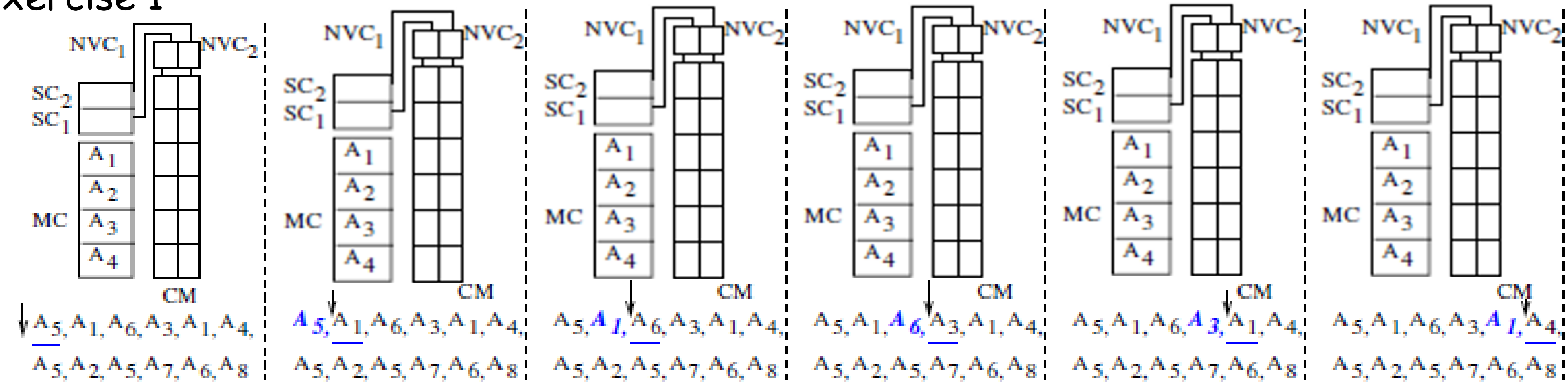
Overview of Shepherd Caching



- To emulate MC with 4 ways per set and 2 SC ways per set
- To gather imminence order add a counter matrix (CM)
- CM has one column per SC way to track imminence order w.r.t to it
- CM has one row per SC and MC line as any of them can be a replacement candidate
- Each column has one Next Value Counter (NVC) to track the next value to assign along column

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

Exercise 1



(a) Initial State

(b) A_5 inserted at SC_1

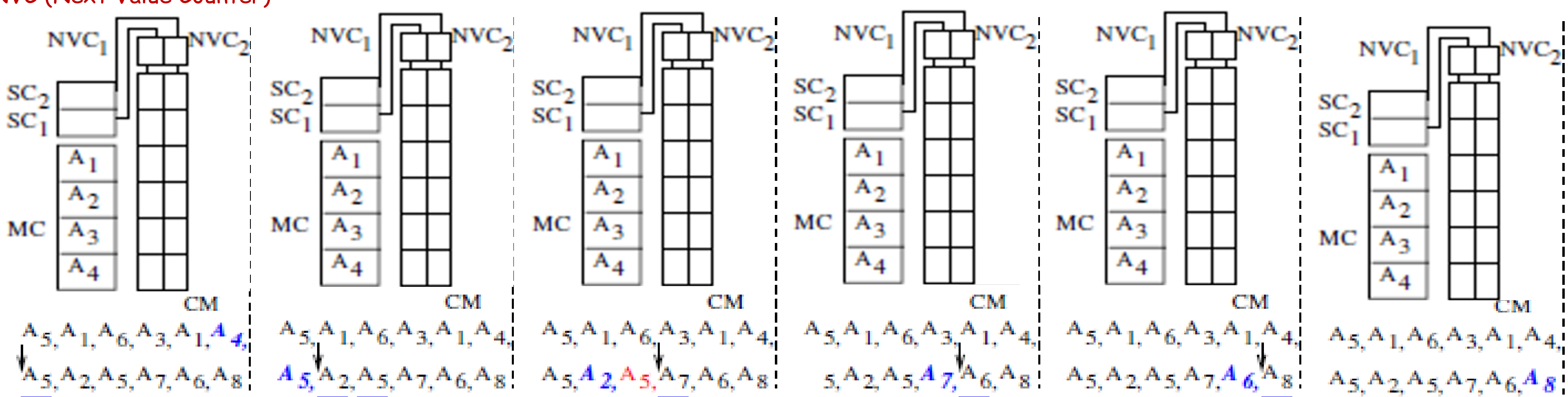
(c) A_1 added to the optimal order of SC_1

(d) A_6 inserted at SC_2

(e) A_3 added to the optimal order of SC_1,SC_2

(f) A_1 added to optimal order of SC_2

MC (Main Cache)
SC (Schepherd Cache)
CM (Counter Matrix)
NVC (Next Value Counter)



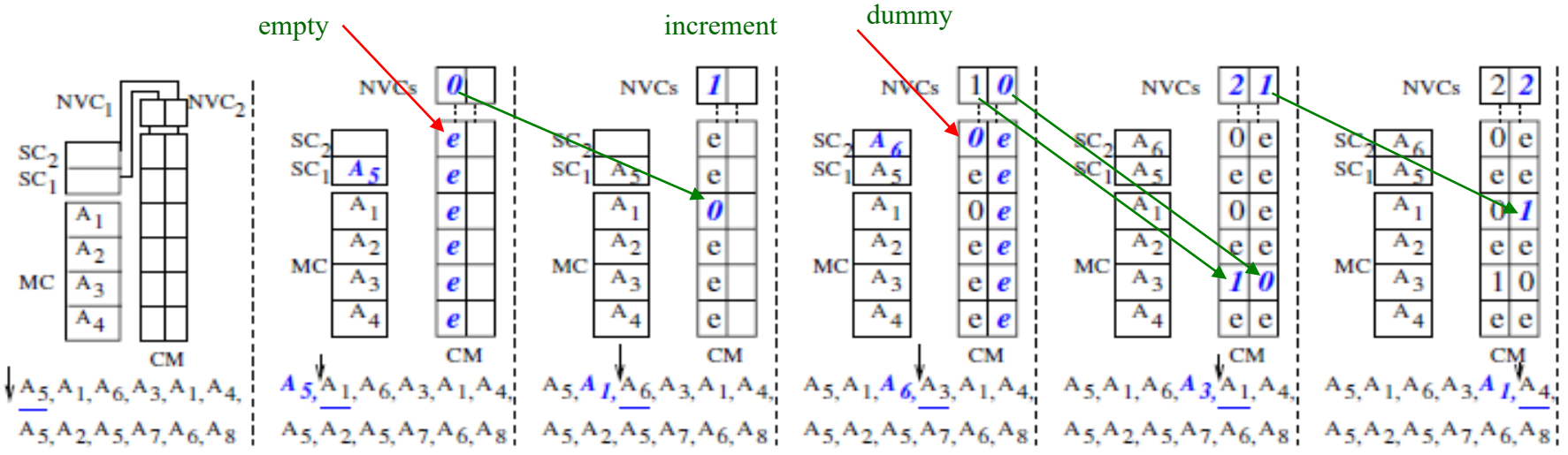
Access Sequence	A_5	A_1	A_6	A_3	A_1	A_4	A_5	A_2	A_5	A_7	A_6	A_8
OPT order for A_5		0		1		2	3	4				
OPT order for A_6				0	1	2	3				4	

A_2 added to
imal order of
$_1$,SC_2

(j) A_5 moves from SC to MC replacing A_2

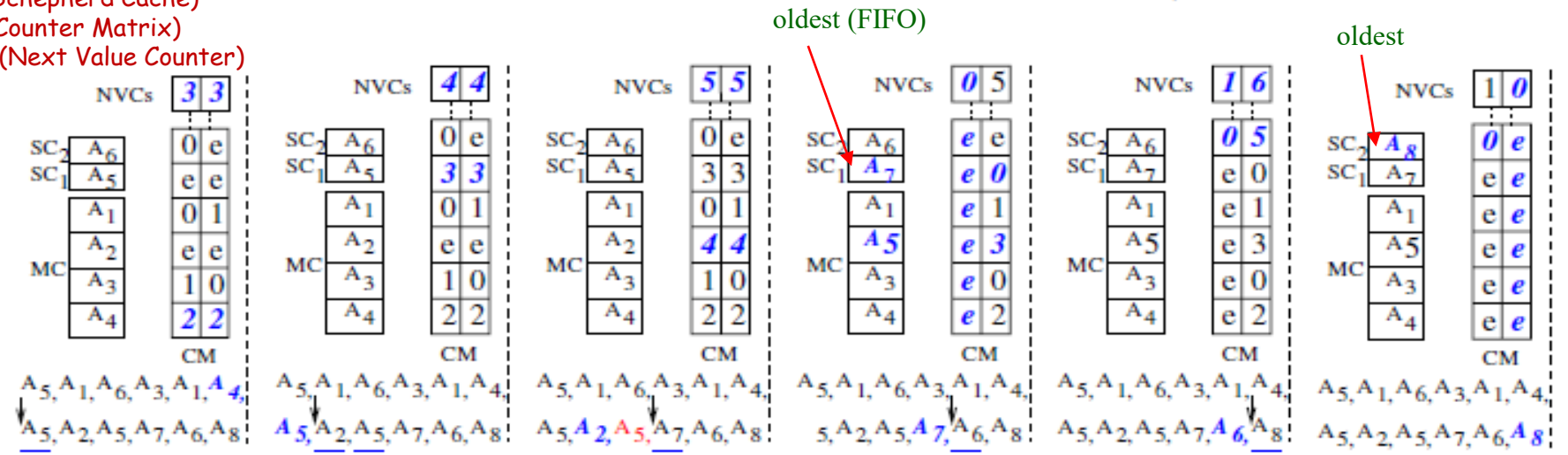
(k) A_6 added to optimal order

(l) Self Replacement (A_6 evicts itself)



(a) Initial State (b) A_5 inserted at SC_1 (c) A_1 added to the optimal order of SC_1 (d) A_6 inserted at SC_2 (e) A_3 added to the optimal order of SC_1,SC_2 (f) A_1 added to optimal order of SC_2

MC (Main Cache)
SC (Schepherd Cache)
CM (Counter Matrix)
NVC (Next Value Counter)

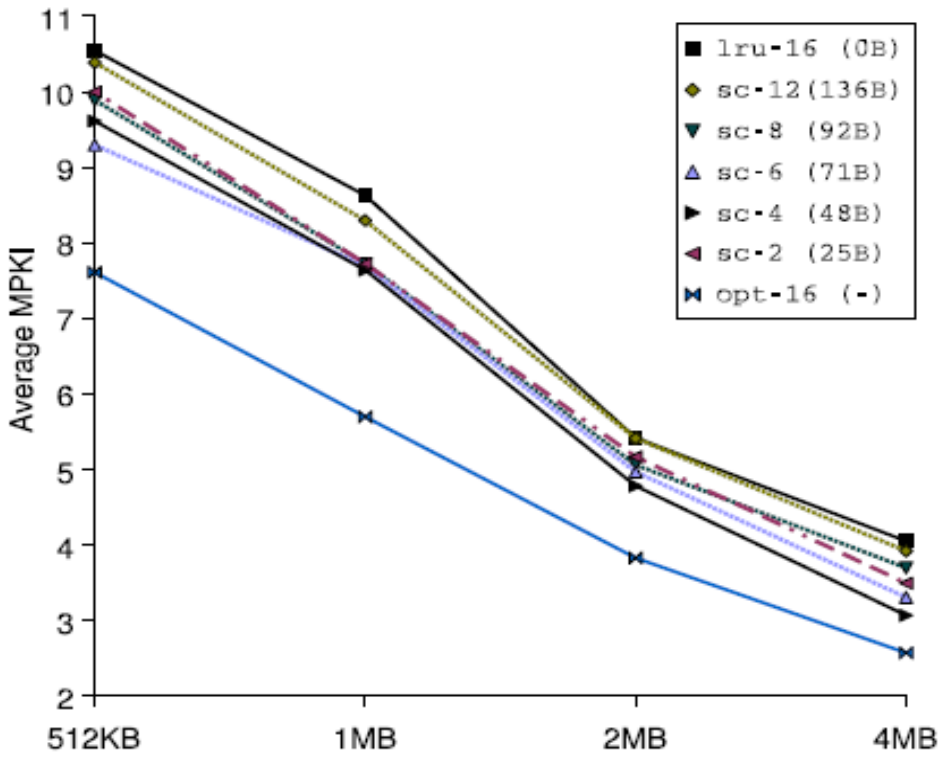


(g) A_4 added to optimal order of SC_1,SC_2 (h) A_5 added to optimal order of SC_1,SC_2 (i) A_2 added to optimal order of SC_1,SC_2 (j) from repla

Access Sequence	A_5	A_1	A_6	A_3	A_1	A_4	A_5	A_2	A_5	A_7	A_6	A_8
OPT order for A_5		0		1		2	3	4				
OPT order for A_6				0	1	2	3				4	

Shepherd cache bridges 32 – 52% of the gap

Bridging the performance gap



Avg MPKI over SPEC2000 suite

Bridging the LRU-OPT gap

- *SC-4 bridges 32-52% of gap*
- SC moves closer to OPT as cache size increases

MPKI: Miss Per Kilo Instructions

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007