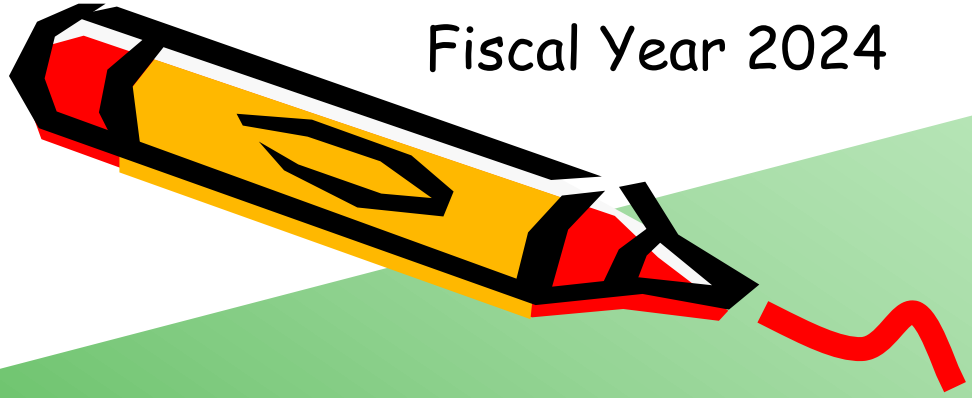


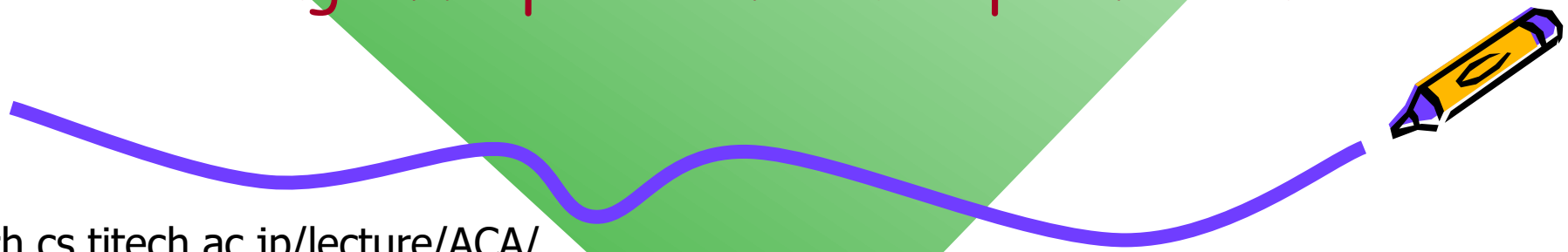
Fiscal Year 2024

Ver. 2025-01-16a

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

Advanced Computer Architecture

8. Instruction Level Parallelism: Exploiting ILP Using Multiple Issue and Speculation

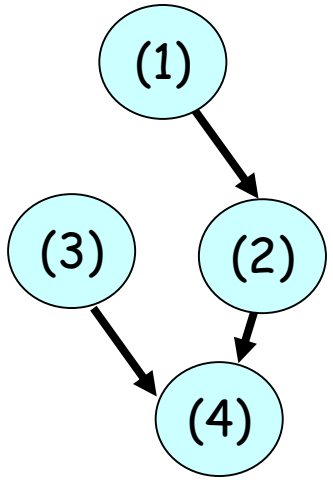


www.arch.cs.titech.ac.jp/lecture/ACA/
Room No. W8E-308, Lecture (Face-to-face)
Mon 13:30-15:10, Thr 13:30-15:10

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

Out-of-order execution (OoO execution)

- In in-order execution model, all instructions are executed in the order that they appear as (1), (2), (3), (4) ...
  This can lead to unnecessary stalls.
  - Instruction (3) stalls waiting for insn (2) to go first, even though it does not have a data dependence.
- With out-of-order execution,
  - Using register renaming to eliminate output dependence and antidependence, just having true data dependence
  - A processor executes instructions in an order governed by the availability of input data and execution units, and the processor can avoid being idle while waiting for the preceding instruction to complete.
    - insn (3) is allowed to be executed before the insn (2)
  - A key design philosophy behind OoO execution to extract ILP by executing instructions as quickly as possible.
    - Scoreboarding (CDC6600 in 1964)
    - Tomasulo algorithm (IBM System/360 Model 91 in 1967)




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

Data flow graph

Recommended Reading

- Focused Value Prediction
  - Sumeet Bandishte, Jayesh Gaur, Zeev Sperber, Lihu Rappoport, Adi Yoaz, and Sreenivas Subramoney, Intel
  - ACM/IEEE 47th International Symposium on Computer Architecture (ISCA), pp. 79-91, 2020
- A quote:
  "Value Prediction was proposed to speculatively break true data dependencies, thereby allowing Out of Order (OOO) processors to achieve higher instruction level parallelism (ILP) and gain performance. State-of-the-art value predictors try to maximize the number of instructions that can be value predicted, with the belief that a higher coverage will unlock more ILP and increase performance. Unfortunately, this comes at increased complexity with implementations that require multiple different types of value predictors working in tandem, incurring substantial area and power cost. In this paper we motivate towards lower coverage, but focused, value prediction. Instead of aggressively increasing the coverage of value prediction, at the cost of higher area and power, we motivate refocusing value prediction as a mechanism to achieve an early execution of instructions that frequently create performance bottlenecks in the OOO processor. Since we do not aim for high coverage, our implementation is light-weight, needing just 1.2 KB of storage. Simulation results on 60 diverse workloads show that we deliver 3.3% performance gain over a baseline similar to the Intel Skylake processor. This performance gain increases substantially to 8.6% when we simulate a futuristic up-scaled version of Skylake. In contrast, for the same storage, state-of-the-art value predictors deliver a much lower speedup of 1.7% and 4.7% respectively. Notably, our proposal is similar to these predictors in performance, even when they are given nearly eight times the storage and have 60% more prediction coverage than our solution.

Recommended Reading

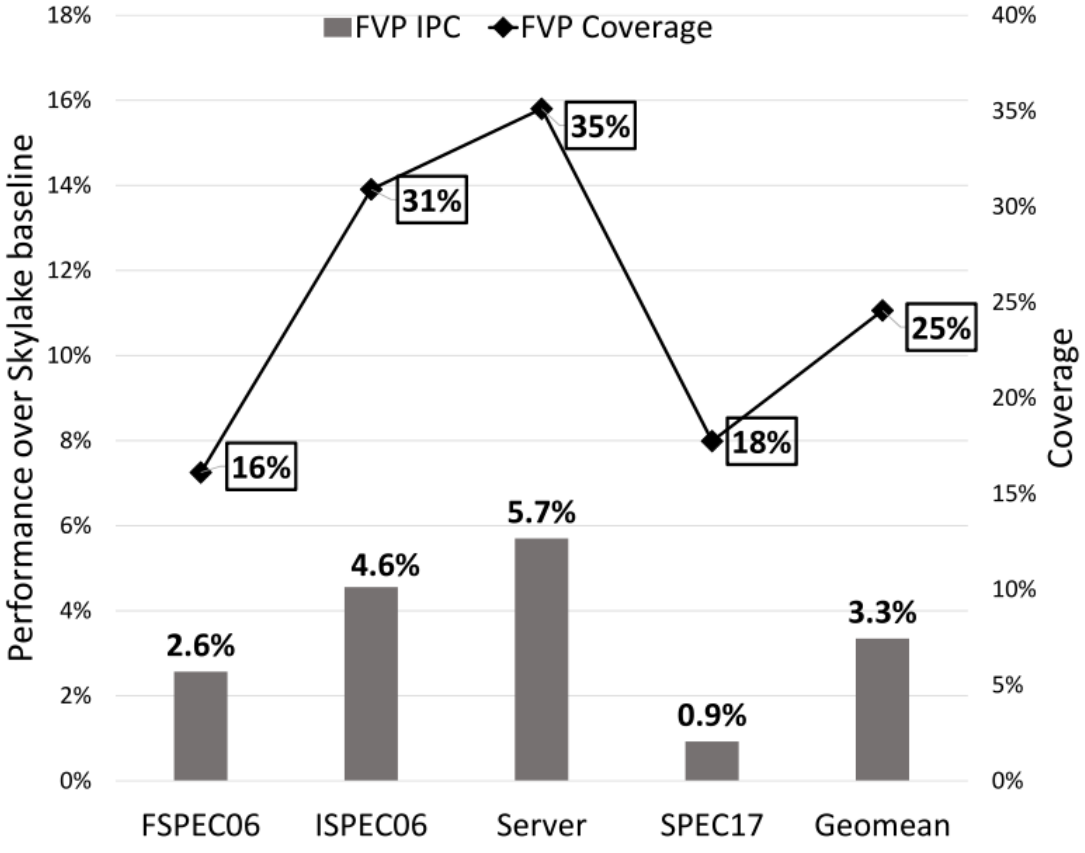


Fig. 6. Performance and Coverage of FVP on Skylake

FVP (Focused Value Prediction, proposal)

Front End	4 wide fetch and decode , TAGE/ITTAGE branch predictors [24], 20 cycles mis-prediction penalty, 64KB, 8-way L1 instruction cache, 4 wide rename into OOO with macro and micro fusion
Execution	224 ROB entries, 64 Load Queue entries, 60 Store Queue entries and 97 Issue Queue entries. 8 Execution units (ports) including 2 load ports, 3 store address ports (2 shared with load ports), 1 store-data port, 4 ALU ports, 3 FP/AVX ports, 2 branch ports. 8 wide retire and full support for bypass. Aggressive memory disambiguation predictor. Out of order load scheduling to L1
Caches	32 KB, 8-way L1 data caches with latency of 5 cycles, 256 KB 16-way L2 cache (private) with a round-trip latency of 15 cycles. 8 MB, 16 way shared LLC with data round-trip latency of 40 cycles. Aggressive multi-stream prefetching into the L2 and LLC. PC based stride prefetcher at L1
Memory	Two DDR4-2133 channels, two ranks per channel, eight banks per rank, and a data bus width per channel of 64 bits. 2 KB row buffer per bank with 15-15-15-39 (tCAS-tRCD-tRP-tRAS) timing parameters

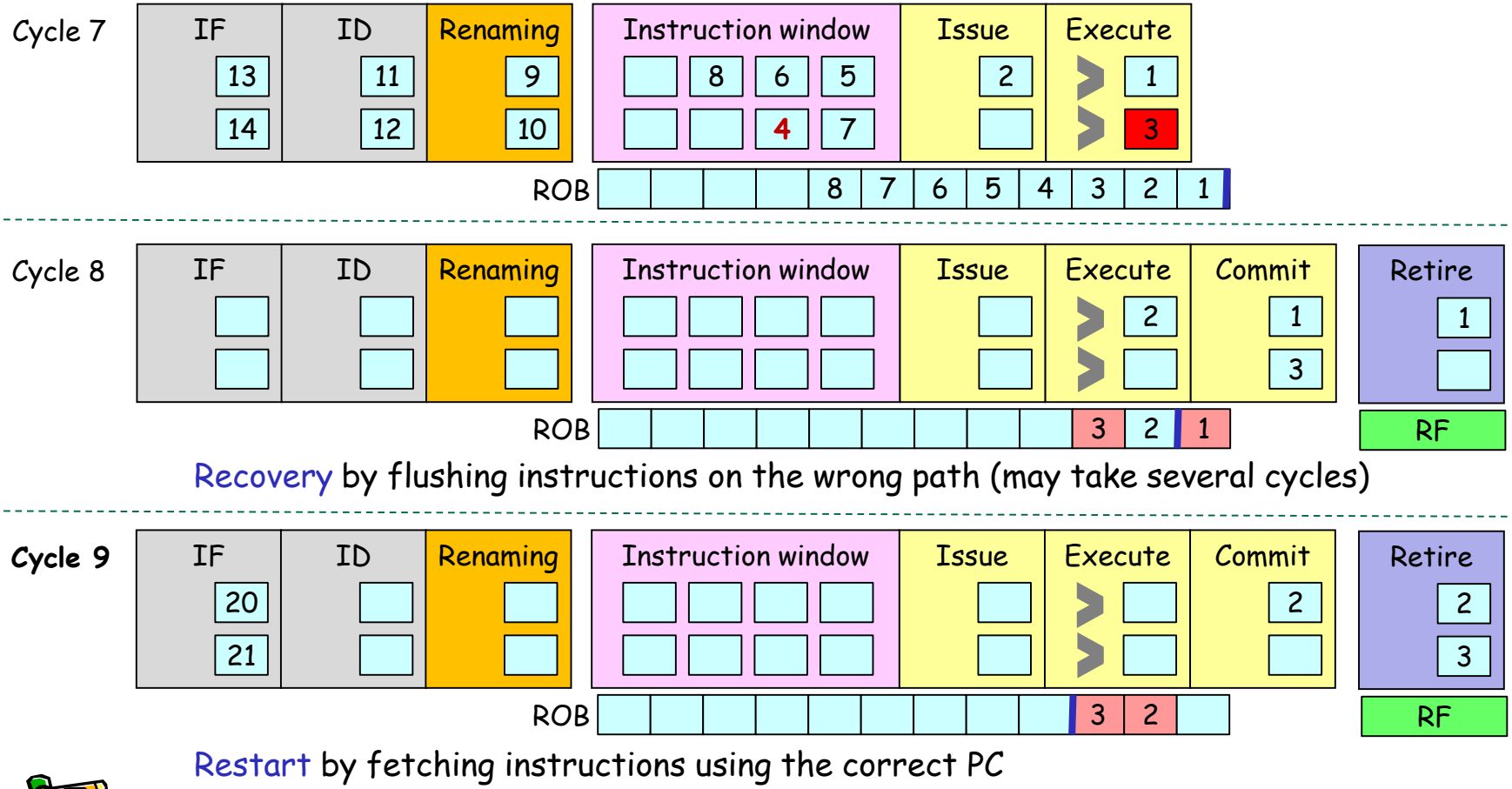
TABLE II
CORE PARAMETERS FOR SIMULATION

Benchmarks	Category
perlbench, bzip2, gcc, mcf, h264ref, gobmk,hmmer, sjeng, libquantum, omnetpp, astar, xalancbmk	SPEC INT 2006 (ISPEC06)
bwaves, gamess, milc, zeusmp, soplex, povray, calculix, gemsfdtd, tonto, wrf, sphinx3 gromacs, cactusADM, leslie3D, namd, deall	SPEC FP 2006 (FSPEC06)
nab, cam4, pop2, roms, leela, cactubssn, xz, gcc, mcf, xalanc, exchange2, omnetpp, perlbench, bwaves, lbm, fotonik3d	SPEC17
lammps [4], hplinpack [3], tpce, spark, cassandra [1], specjbb [5], specjenterprise, hadoop [2], specpower [6]	Server

TABLE III
APPLICATIONS USED IN THIS STUDY

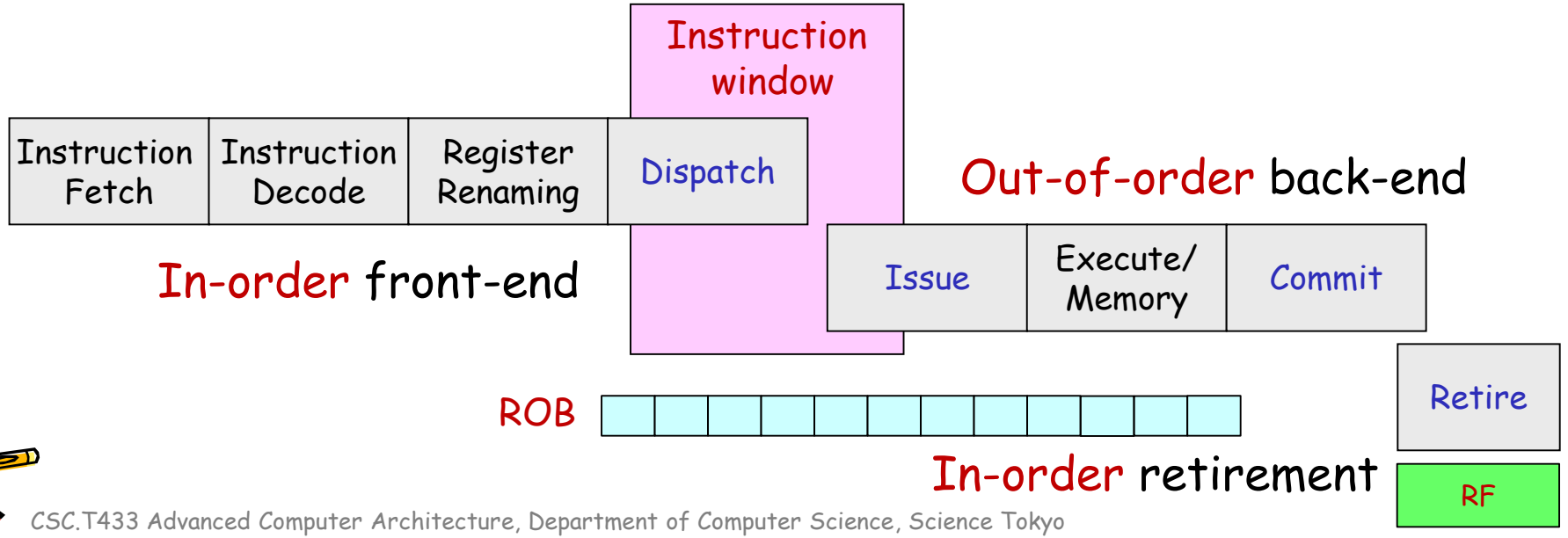
Branch prediction miss and aggressive recovery

- Instruction 3 is a miss predicted branch and its target insn is 20
- When insn 3 is executed, it recovers by flushing instructions after insn 3 and restarts



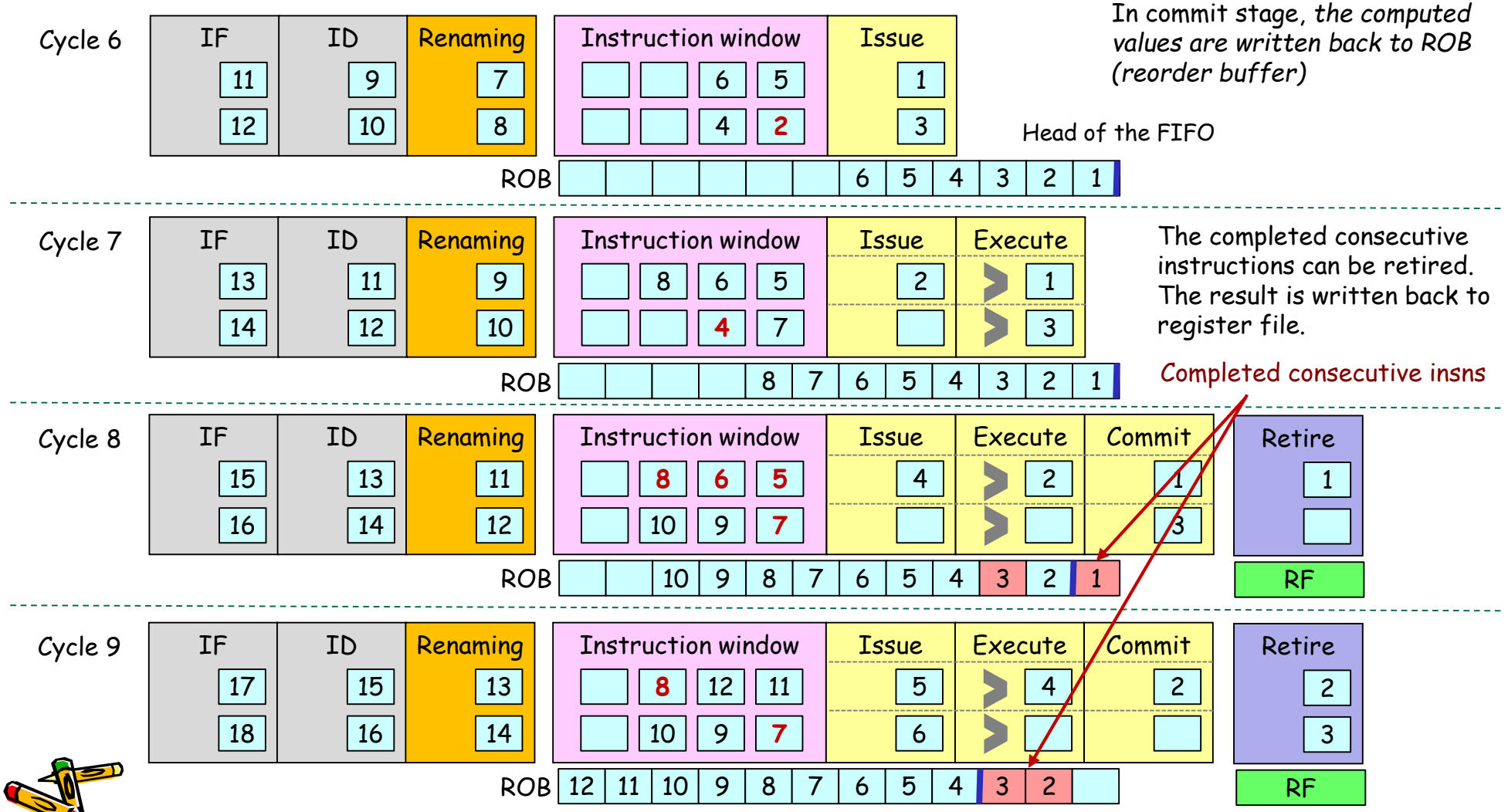
Instruction pipeline of OoO execution processor

- 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.



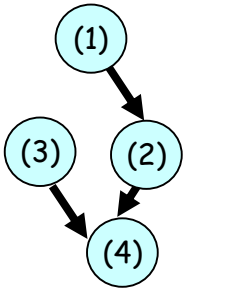
The key idea for OoO execution (last lecture)

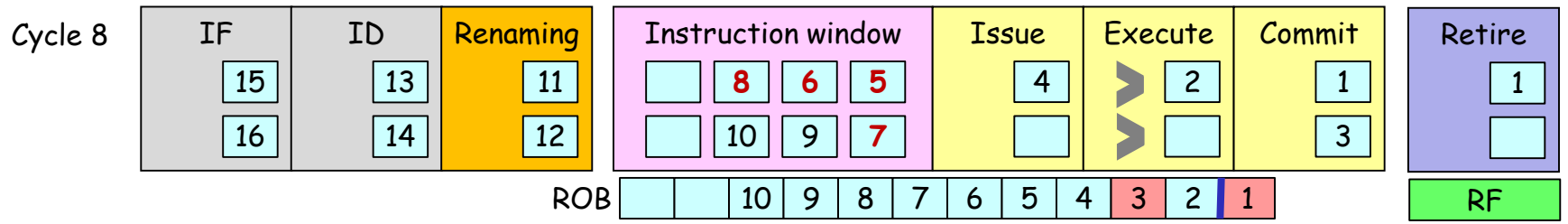
- In-order front-end, OoO execution core, in-order retirement using instruction window and reorder buffer (ROB)

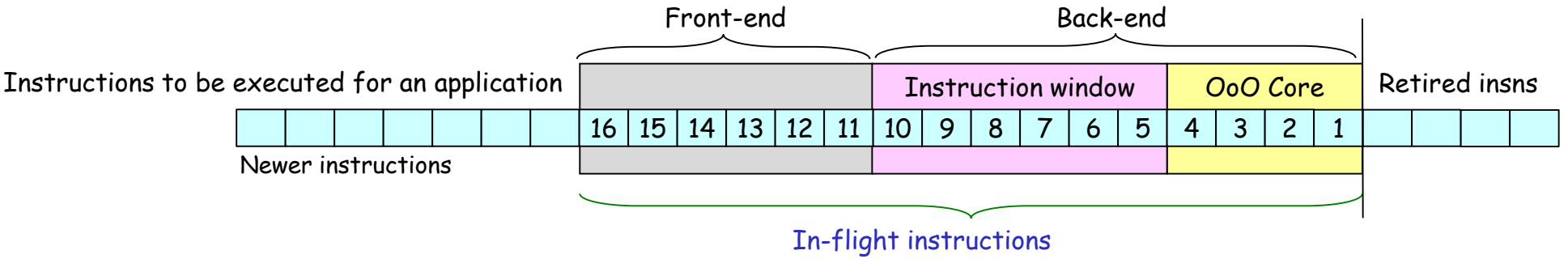


Register dataflow

- In-flight instructions are ones processing in a processor



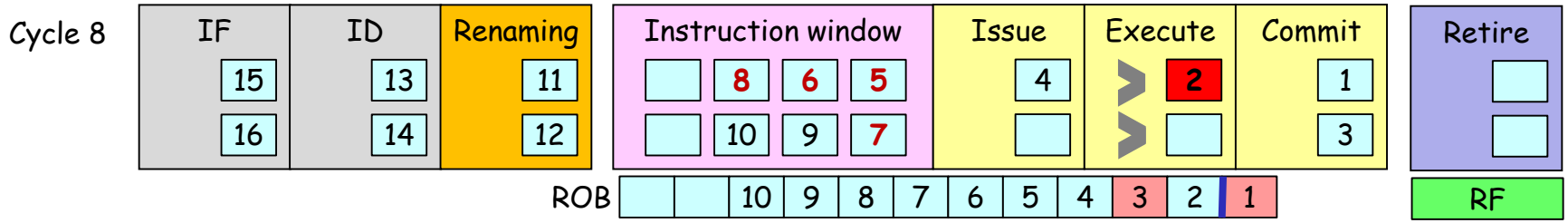


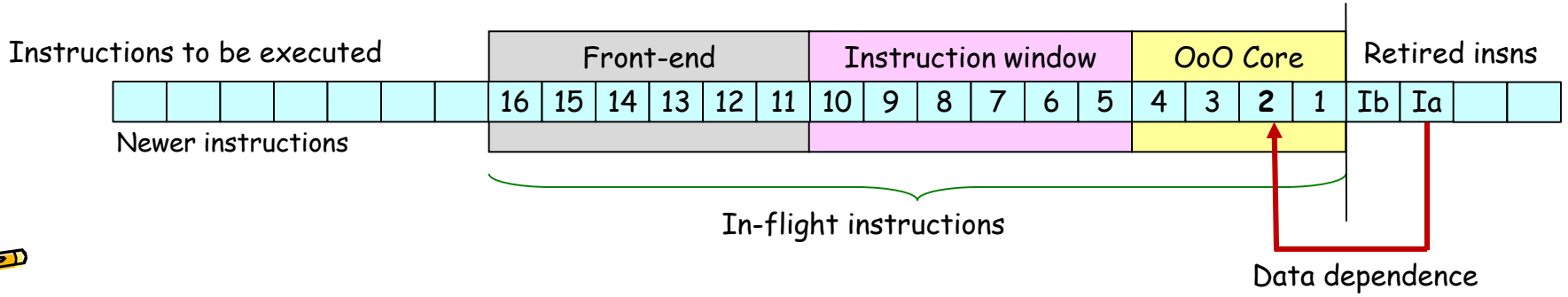


Case 1: Register dataflow from a far previous instn

- One source operand of insn I2 is from a retired instruction Ia.
- Because Ia is 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
```

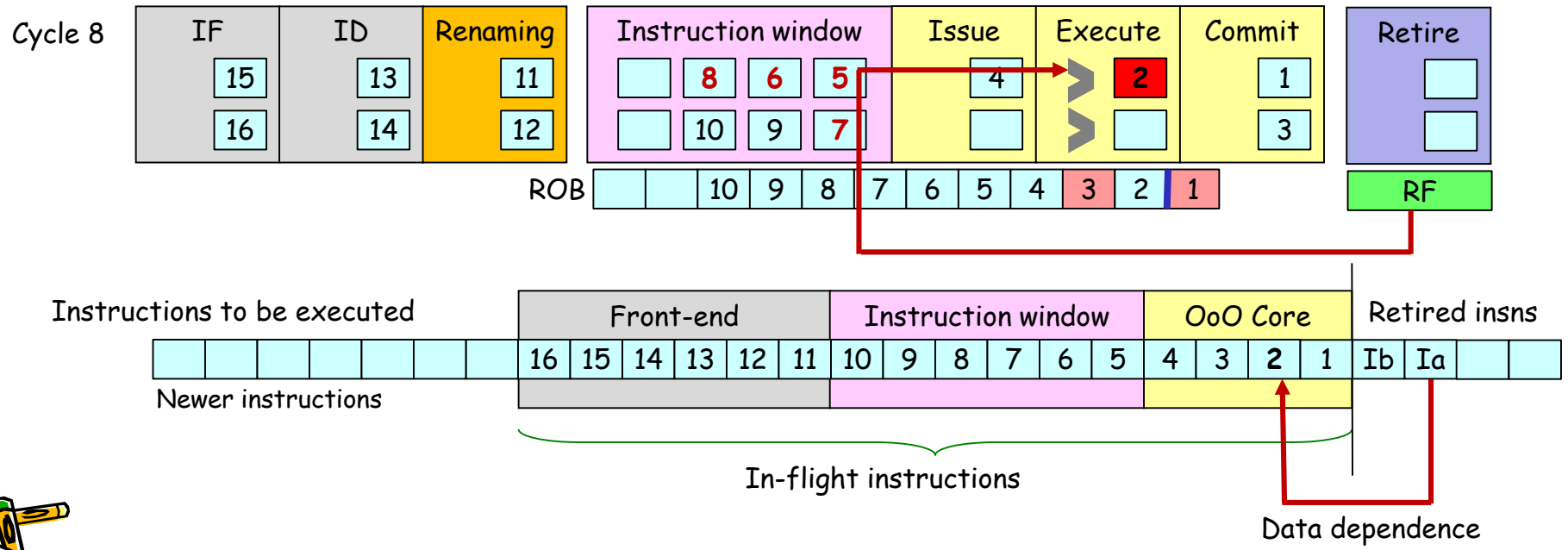




Case 1: Register dataflow from RF

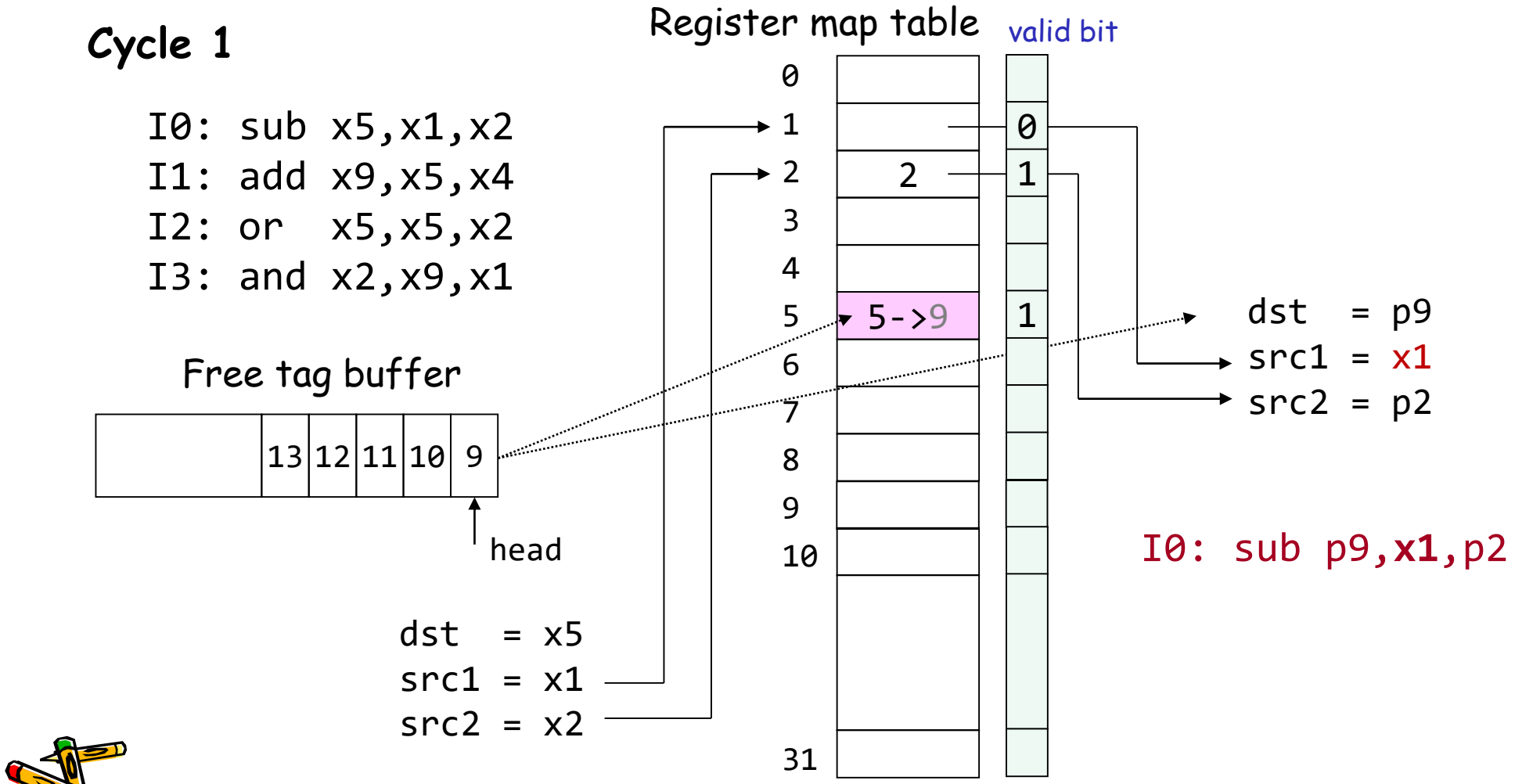
- One source operand of insn I2 is from a retired instruction Ia.
- Because Ia is retired long ago, the physical destination register has been freed. The tag of the source register x3 can not be renamed at the renaming stage for I2, still having the logical register tag x3.
- Where does the operand x3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

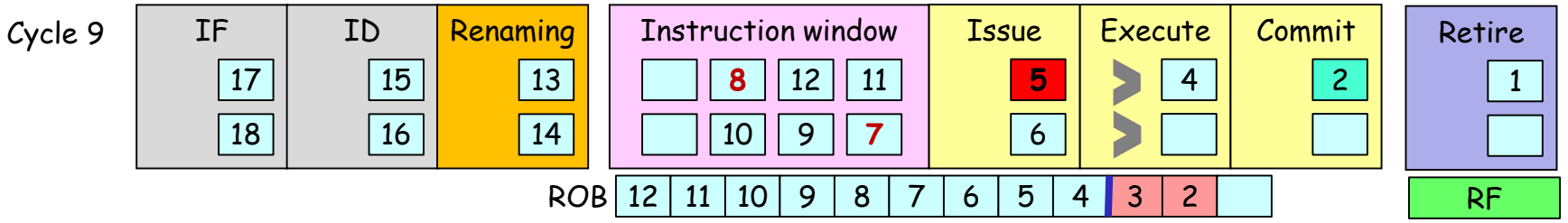
- A processor remembers a set of renamed logical registers.
- If x1 is not renamed for in-flight insn, it uses x1 instead of p1.

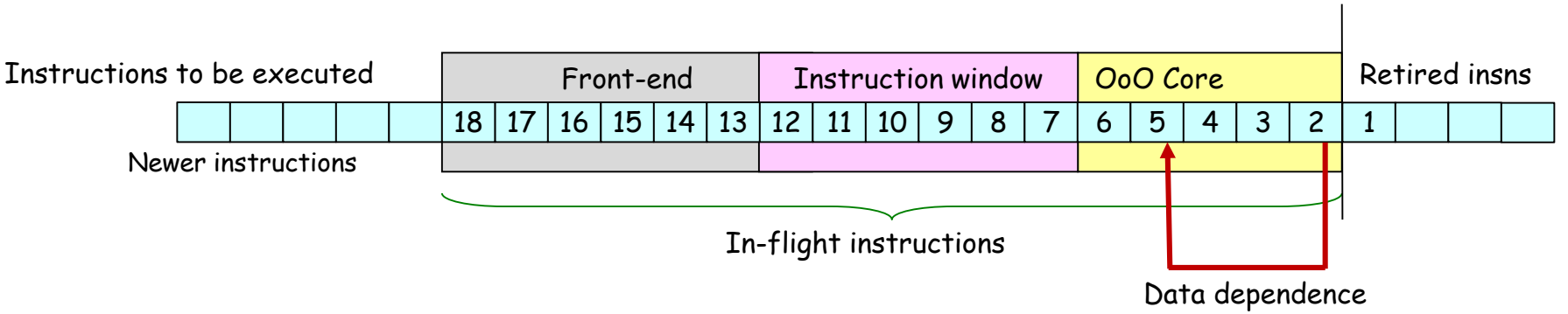


Case 2: Register dataflow

- 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
```

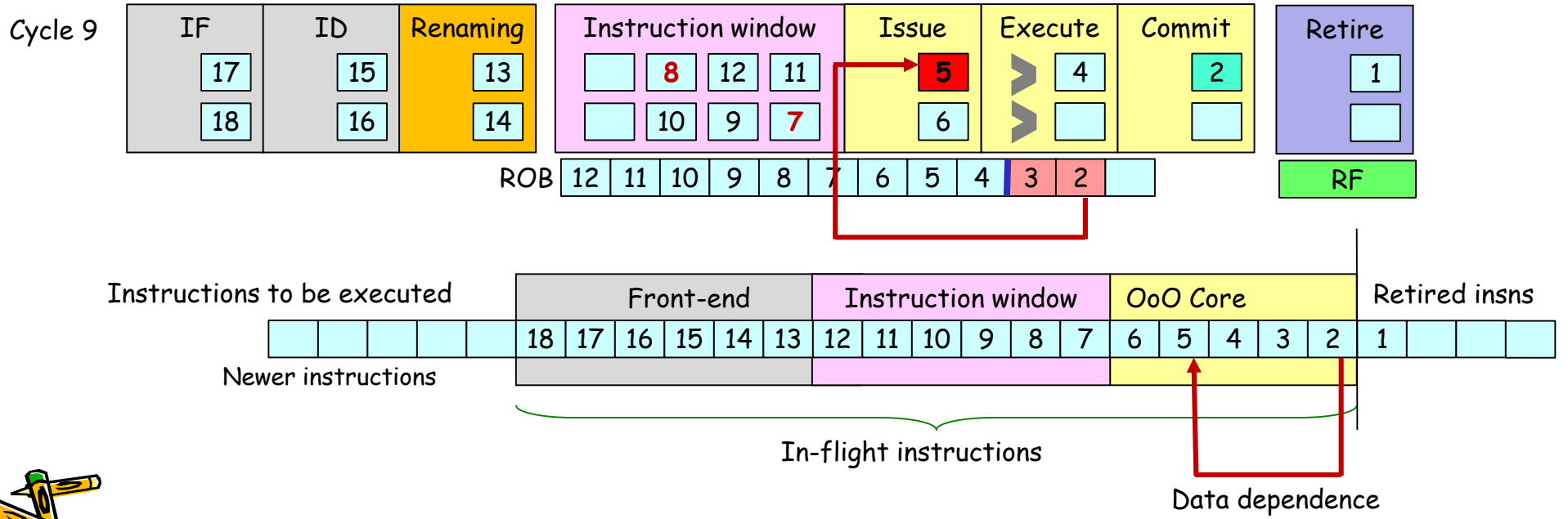




Case 2: Register dataflow from ROB

- 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?

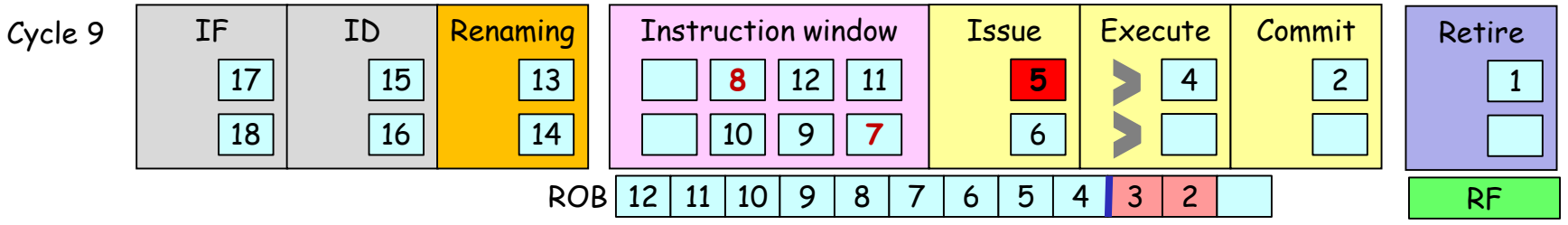
```
Ia: add x3,x0,x0
I1: sub p9,x1,x2
I2: add p10,p9,x3
I3: or  p11,x4,x5
I4: and p12,p10,p11
I5: nor p13,p10,p12
```

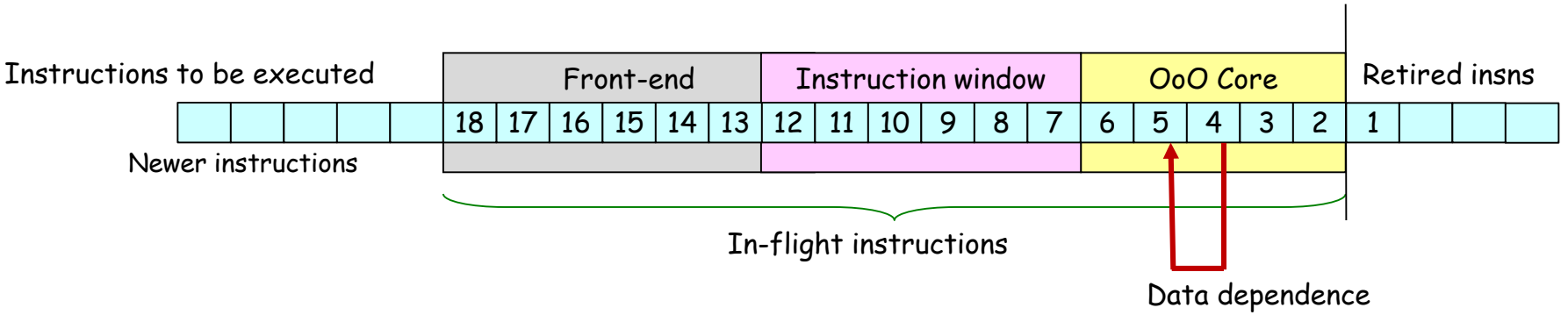


Case 3: Register dataflow

- 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
```

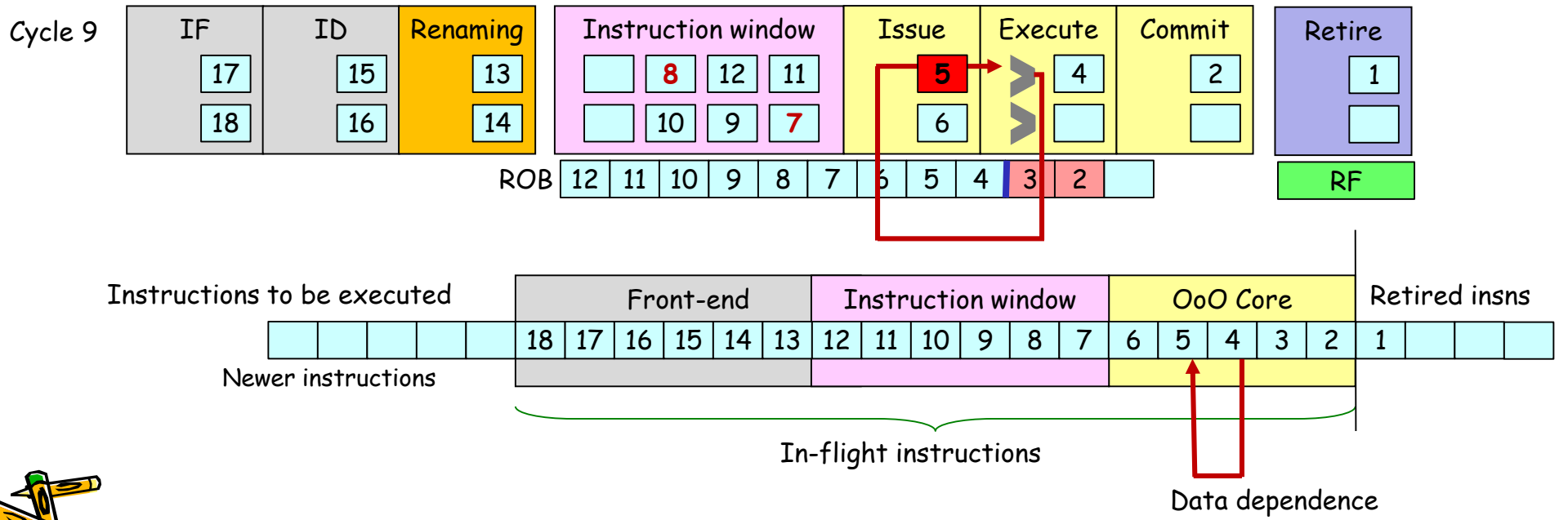




Case 3: Register dataflow from ALUs

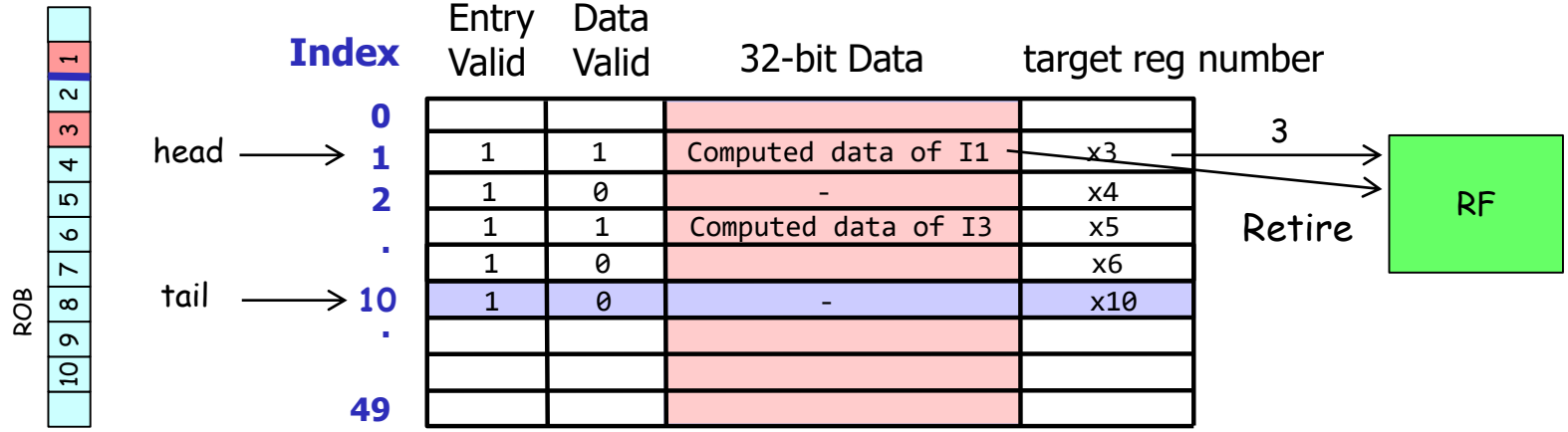
- Assume that the other source operand p12 of insn I5 is from I4 which is not committed. The operand is generated in the previous clock cycle.
- Because I4 is not retired, RF does not have the operand.
  Because I4 is not committed, ROB does not have the operand.
- Where does the operand of I5 come from?

```
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
```

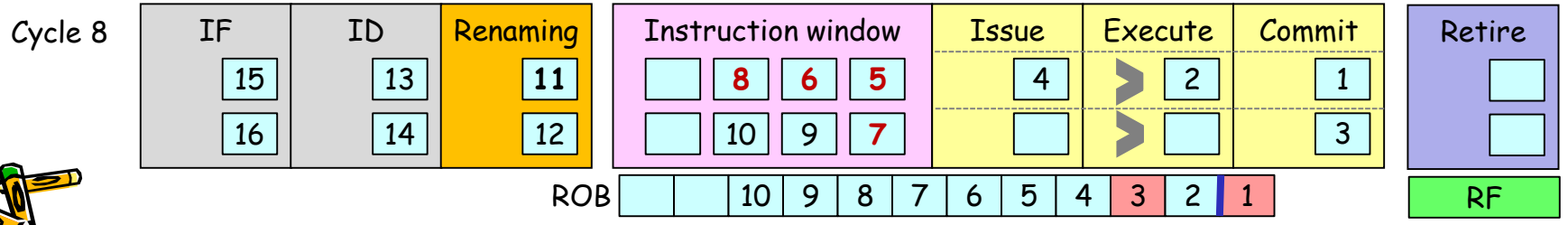


Reorder buffer (ROB)

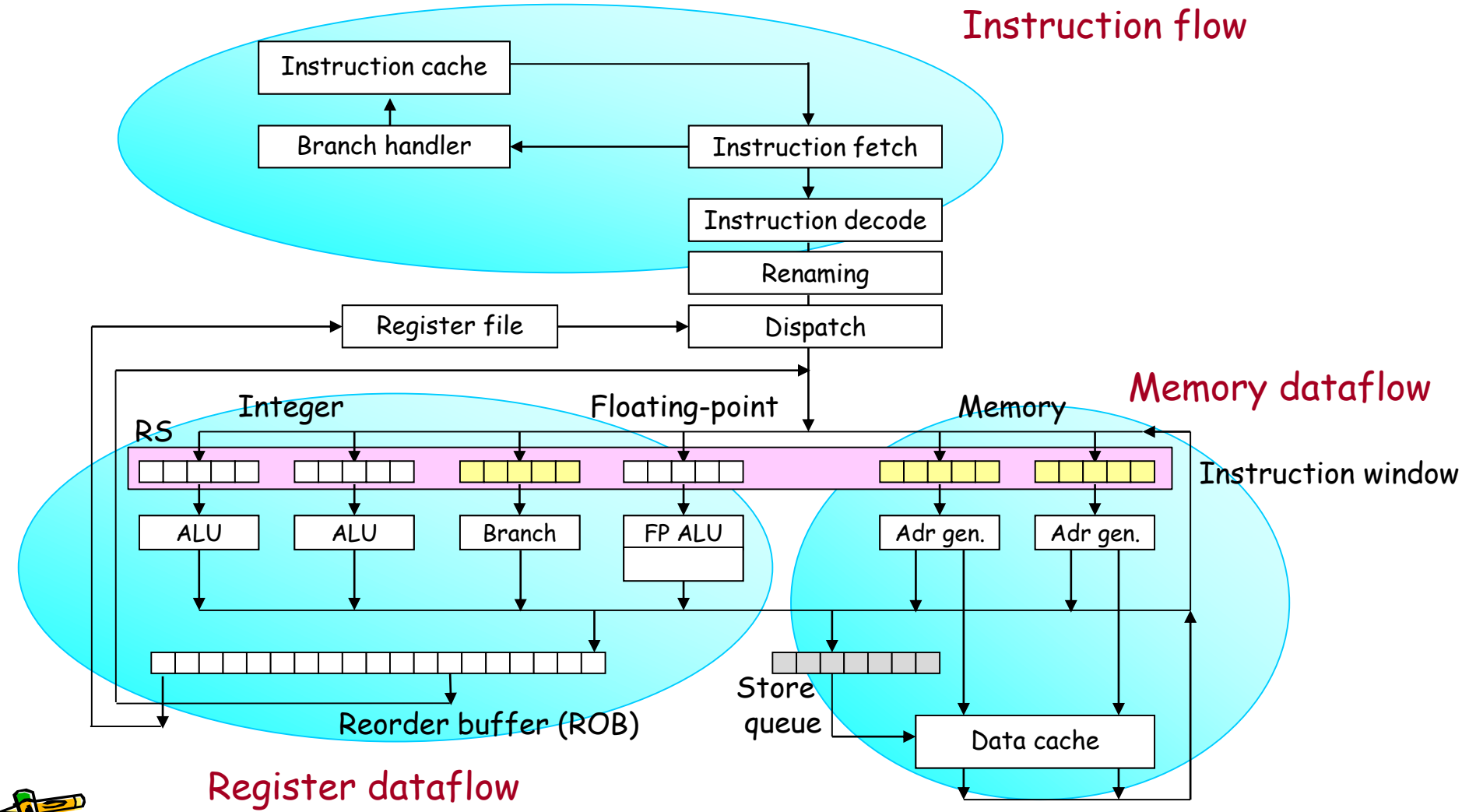
- Each ROB entry has following fields
  - entry valid bit, data valid bit, **data**, target register number, etc.
- ROB provides the large physical registers for renaming
  - in fact, physical register number is ROB entry number
- The value of a physical register may come from a matching ROB entry



I10: add p10,**p3**,p8 (add x10,x5,x6)

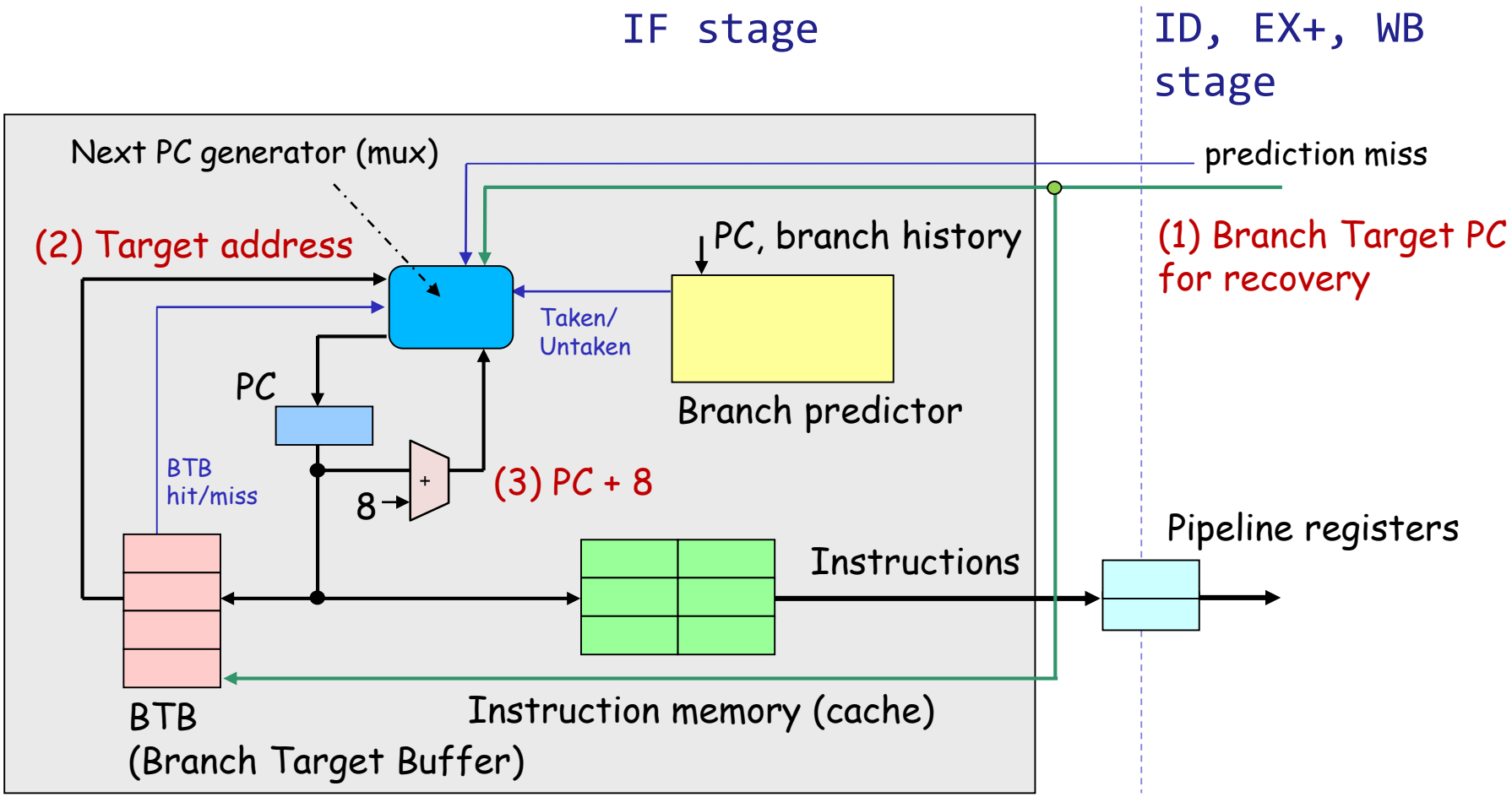


Datapath of OoO execution processor

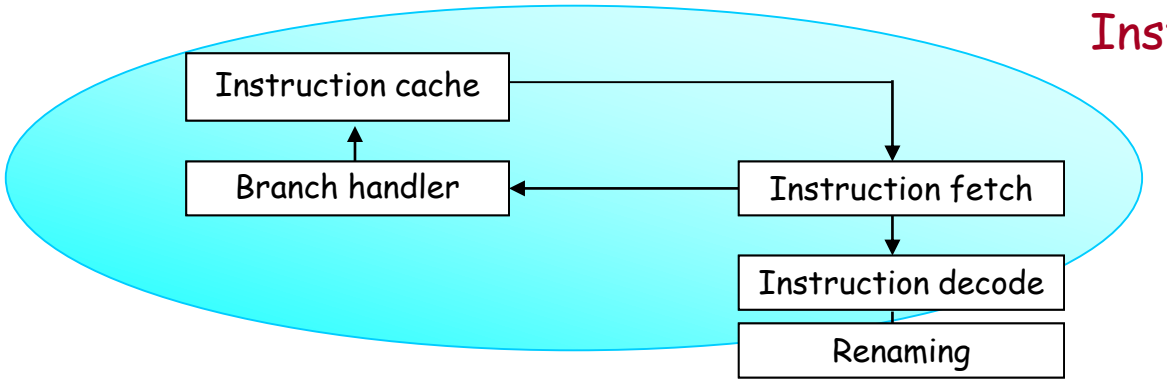


Instruction fetch unit of 2-way super-scalar

- High-bandwidth instruction delivery using prediction, and speculation

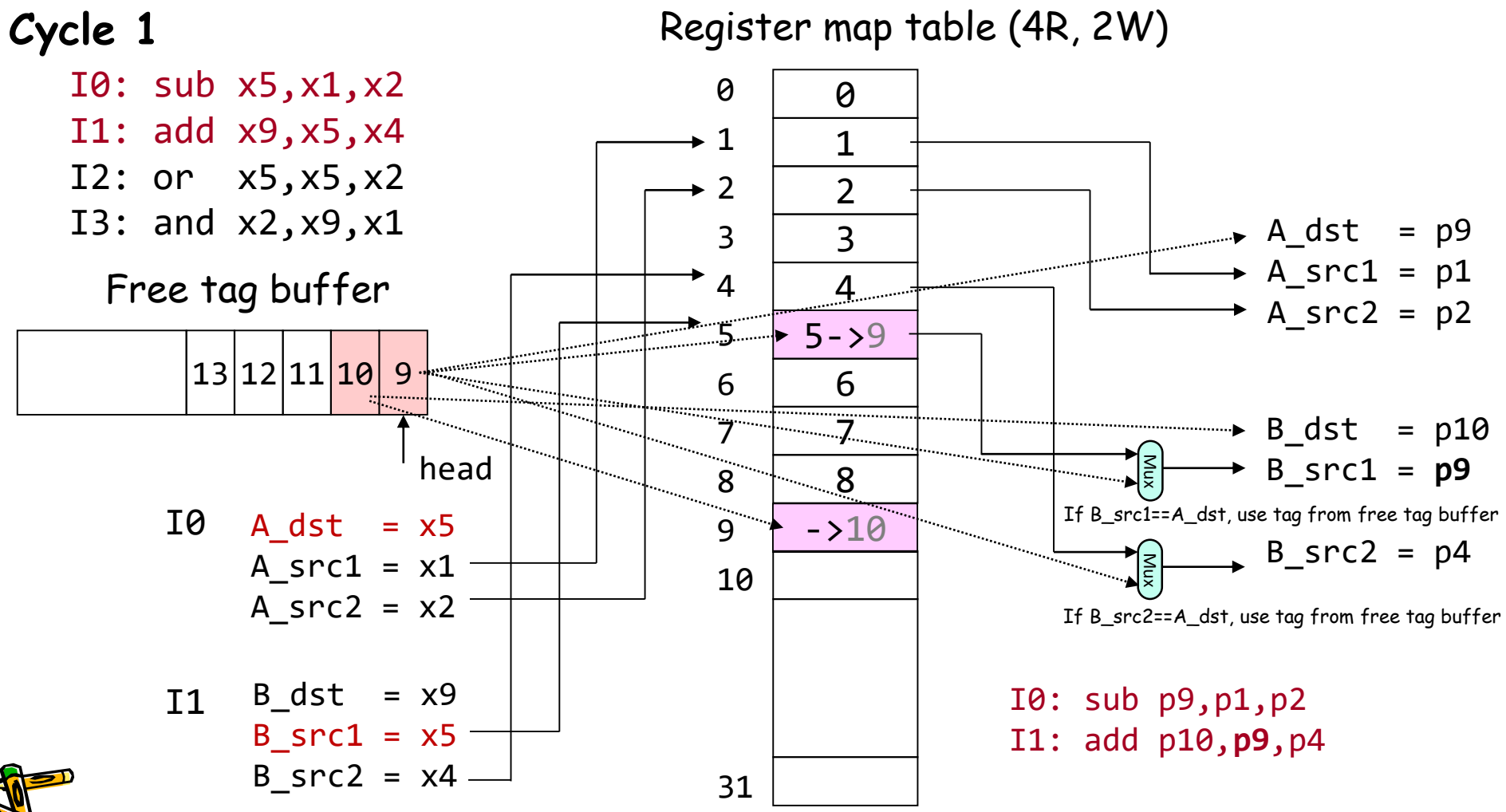


Datapath of OoO execution processor (partially)



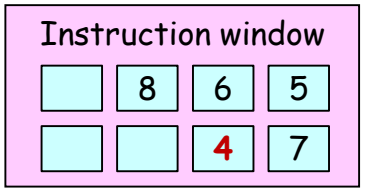
Renaming two instructions per cycle for n-way superscalar

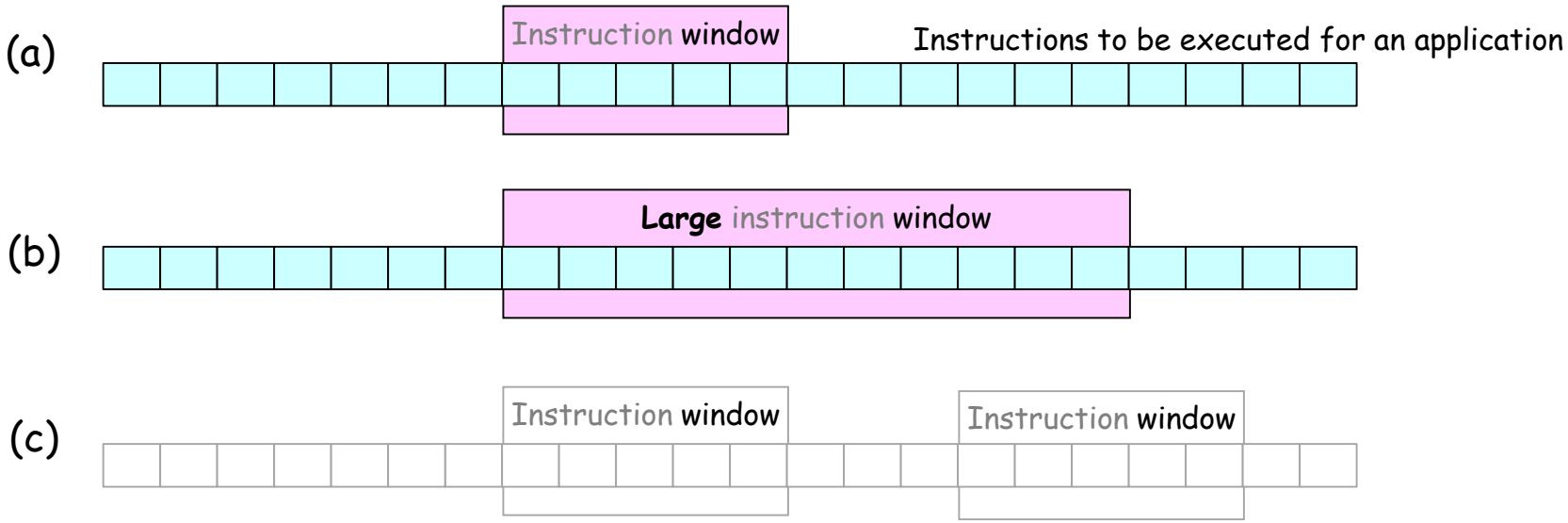
- Renaming instruction I0 and I1 (n = 2)



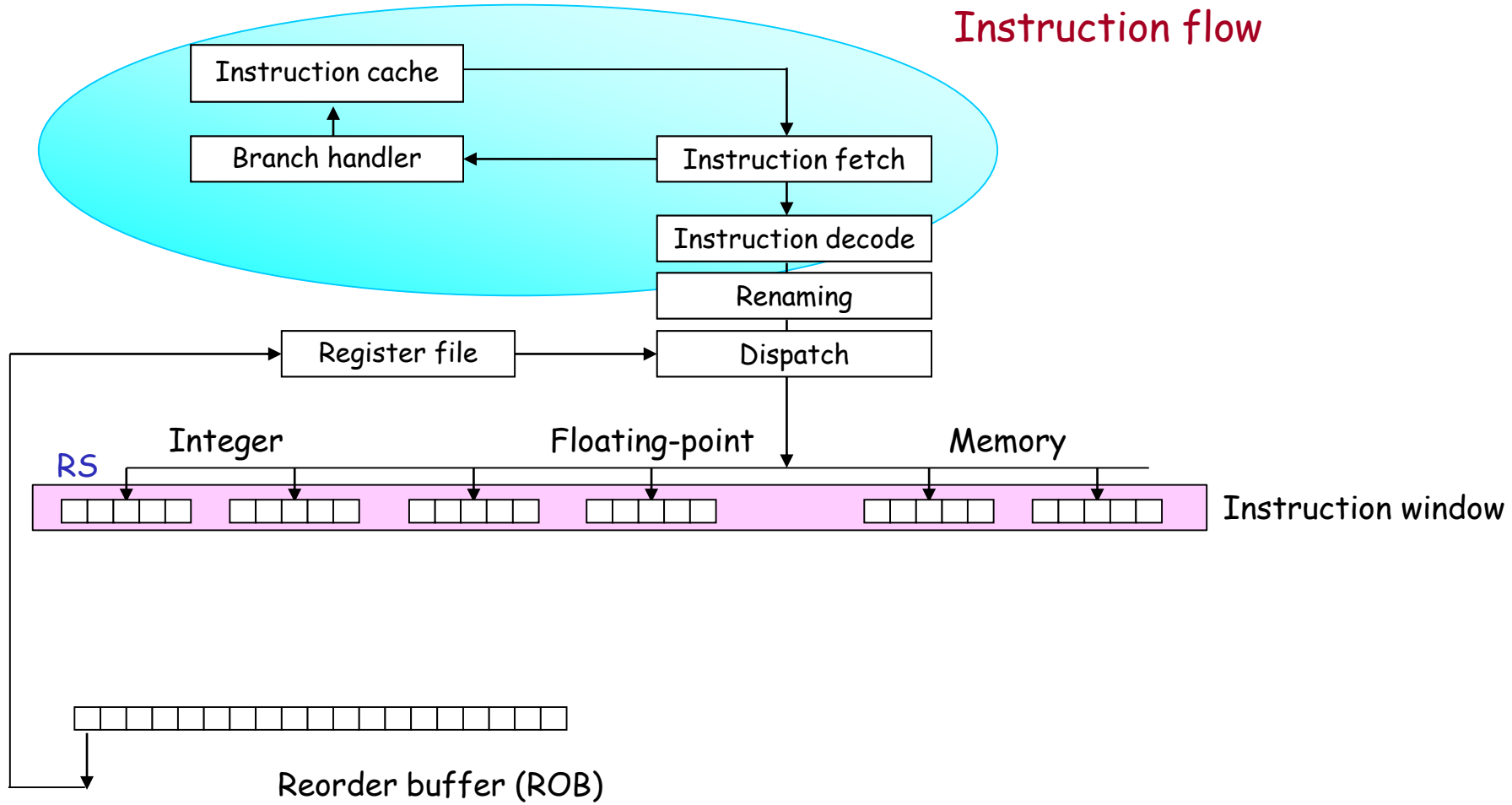
Aside: What is a window?

- 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)



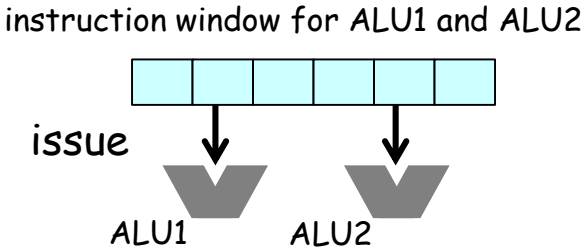


Datapath of OoO execution processor (partially)

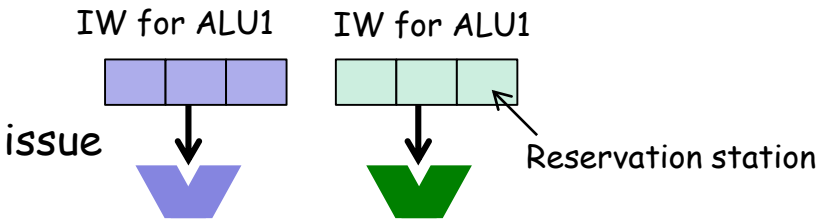


Reservation station (RS)

- To simplify the wakeup and select logic at issue stage, each functional unit (ALU) has own instruction window, an entry for such an an instruction window is called reservation station (RS).
- Each reservation station has
  - entry 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.

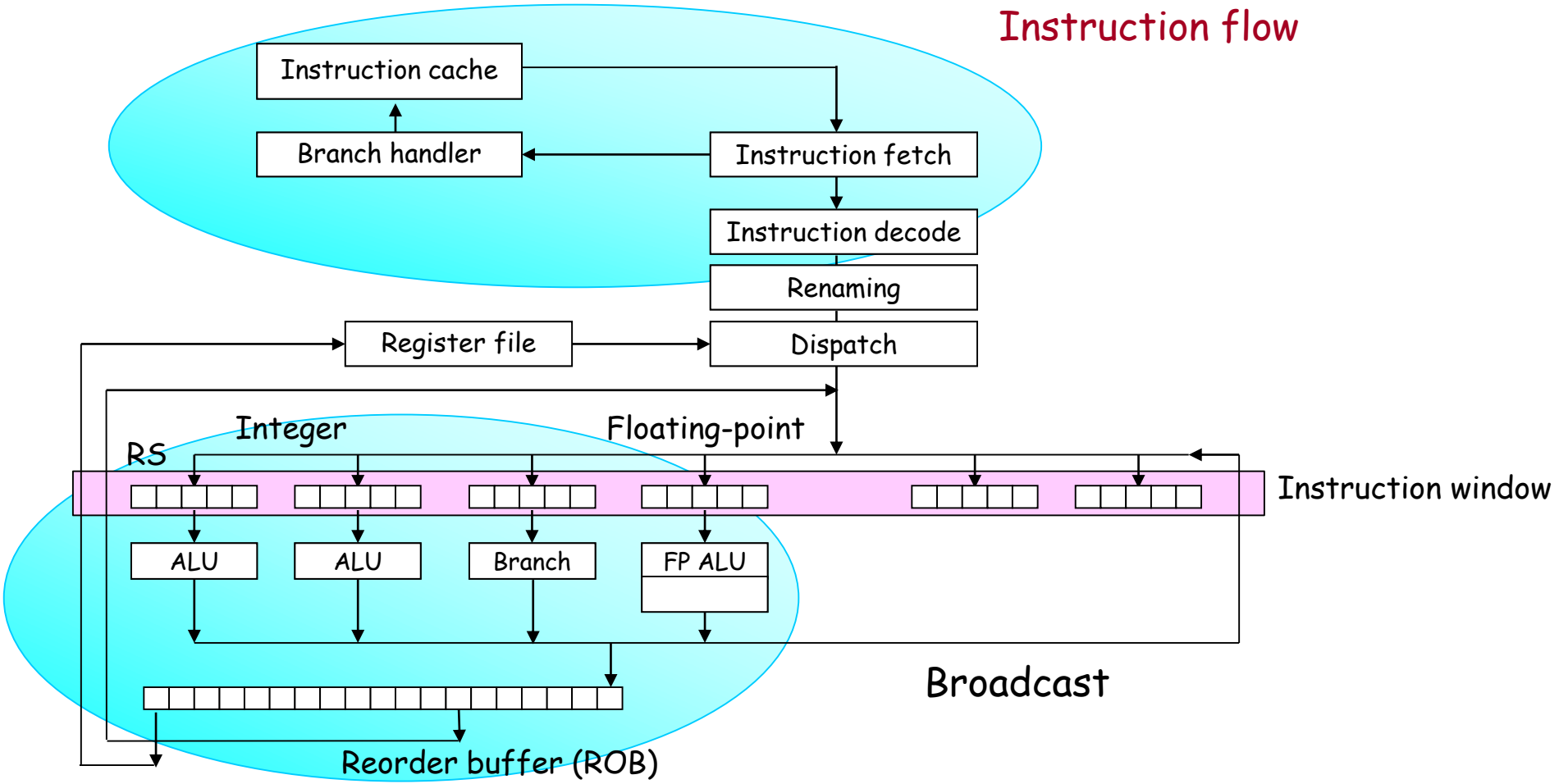


(a) Centralized instruction window



(b) Distributed instruction window using RS

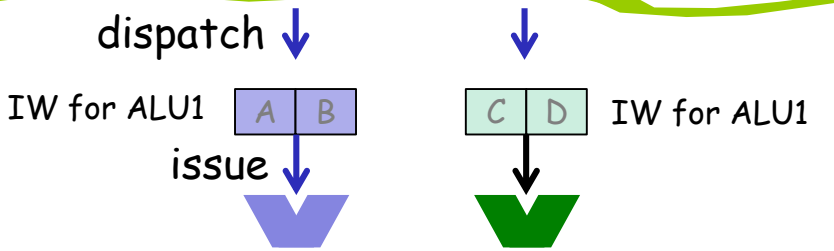
Datapath of OoO execution processor (partially)



Example behavior of reservation stations

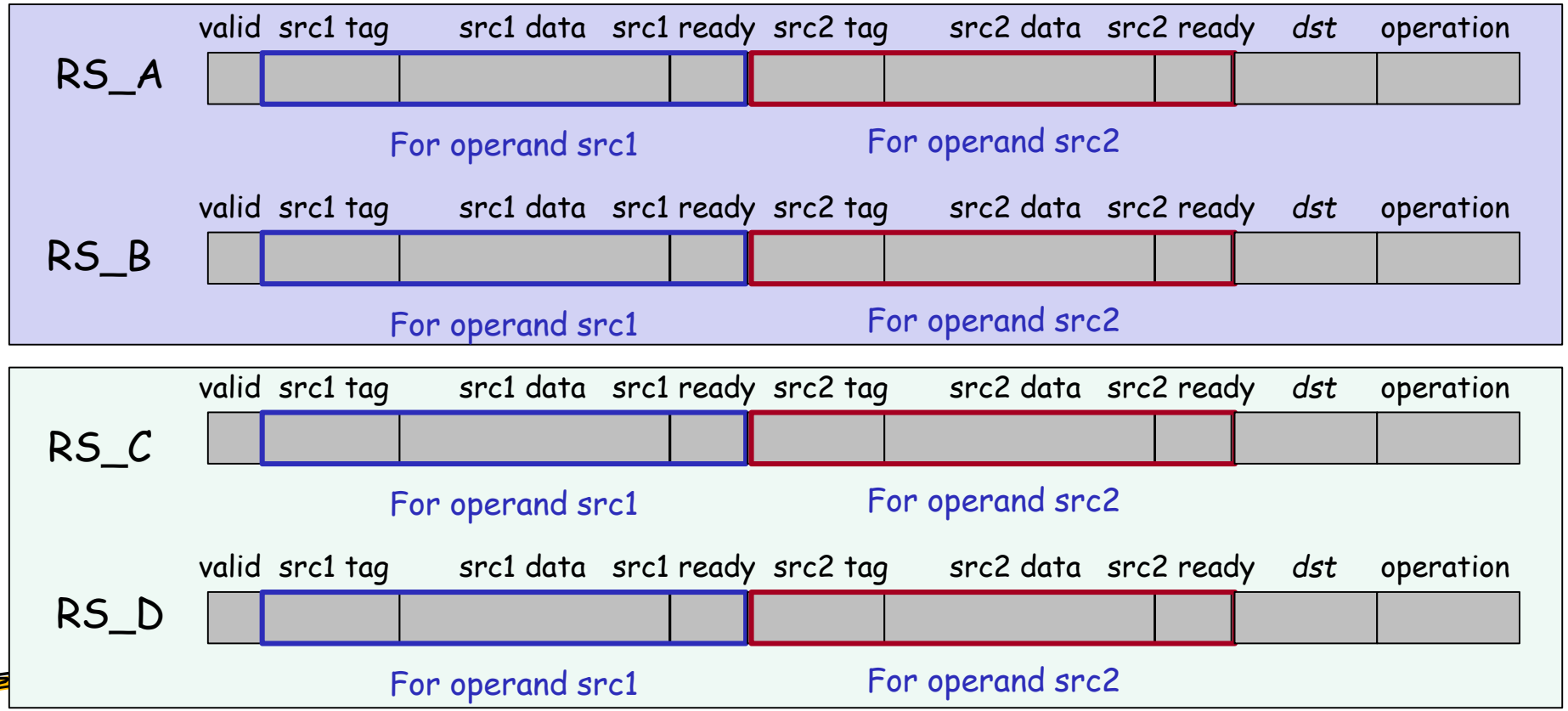
Cycle 0

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
```

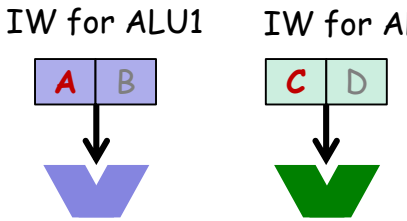
dispatch at most two instructions, one to A or B and the other to C or D



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
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

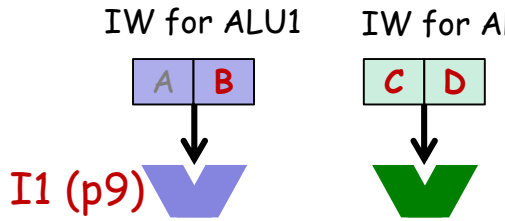
valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation

For operand src1 For operand src2

Example behavior of reservation stations

Cycle 2

dispatch I5, I6
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
1/0								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	value of p9	1	x3	value of x3	1	p10	I2: add

For operand src1 For operand src2

RS_D

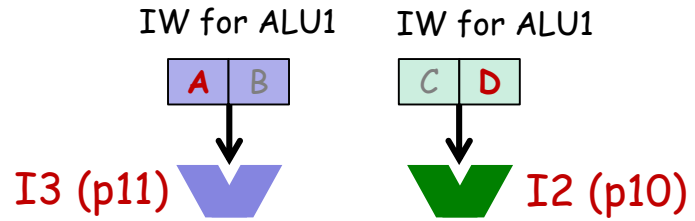
valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
1	p10		0	p11		0	p12	I4: and

For operand src1 For operand src2

Example behavior of reservation stations

Cycle 3

dispatch I7, I8
issue I4
execute I2, I3




```
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	value of p10	1	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
1/0								I3: or
	For operand src1			For operand src2				

RS_C

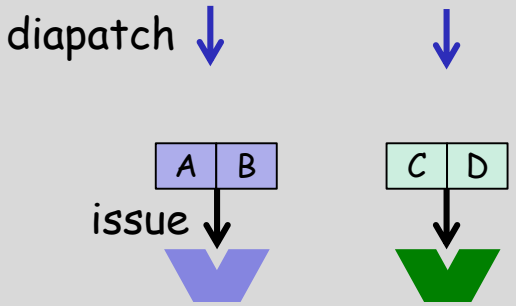
valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
1/0								I2: add
	For operand src1			For operand src2				

RS_D

valid	src1 tag	src1 data	src1 ready	src2 tag	src2 data	src2 ready	dst	operation
1	p10	value of p10	1	p11	value of p11	1	p12	I4: and
	For operand src1			For operand src2				

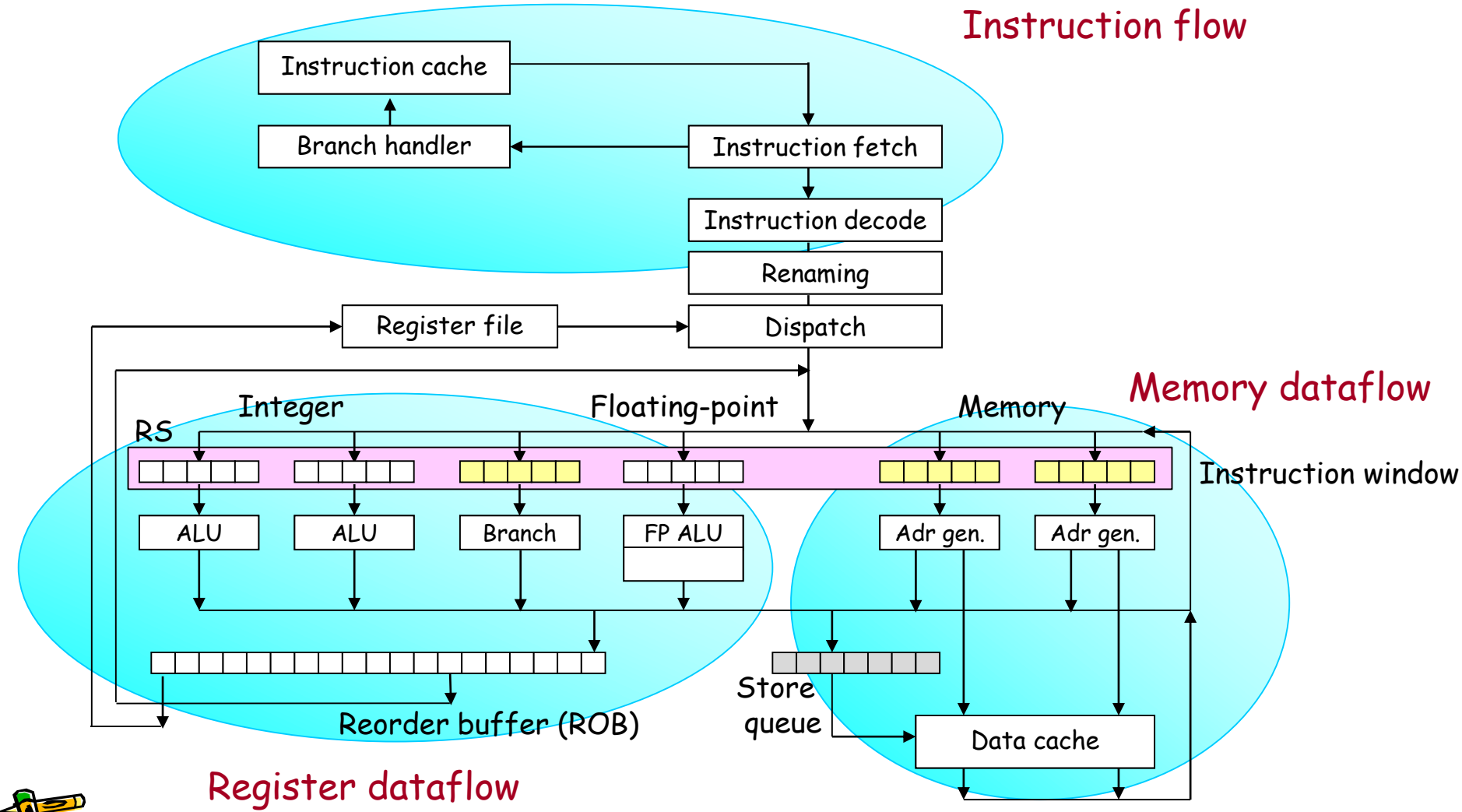
Exercise 1

- Example behavior of reservation stations



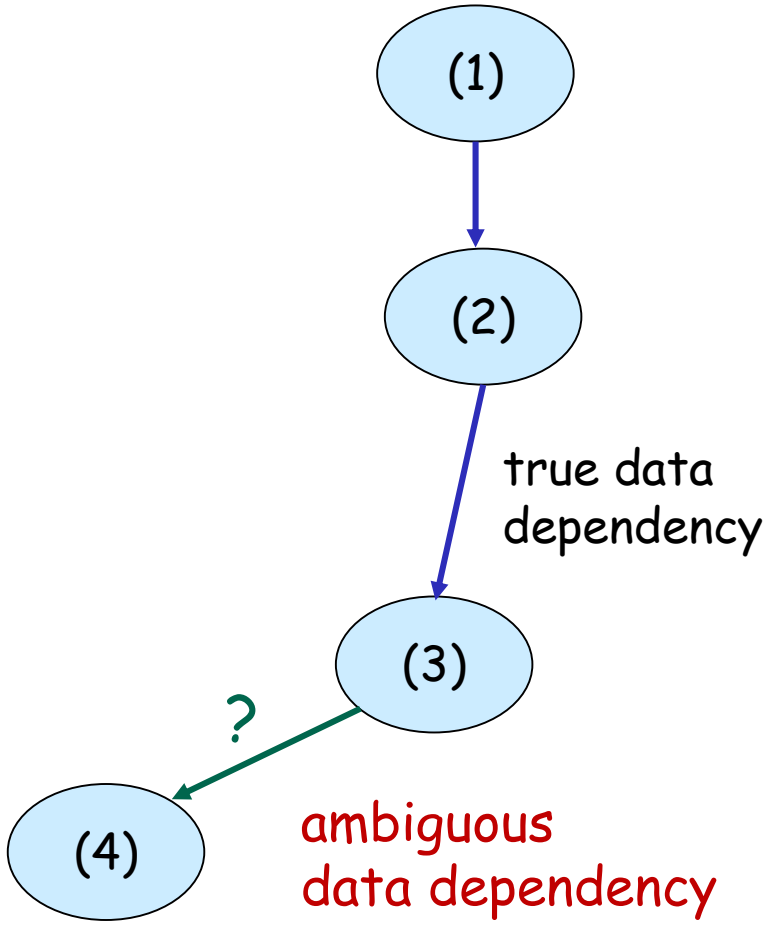

```
I1: sub p9,x1,x2
I2: add p10,p9,x3
I3: or  p11,p10,x4
I4: and p12,x5,x6
I5: nor p13,p11,p12
I6: add p14,p10,x7
```

Datapath of OoO execution processor



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).

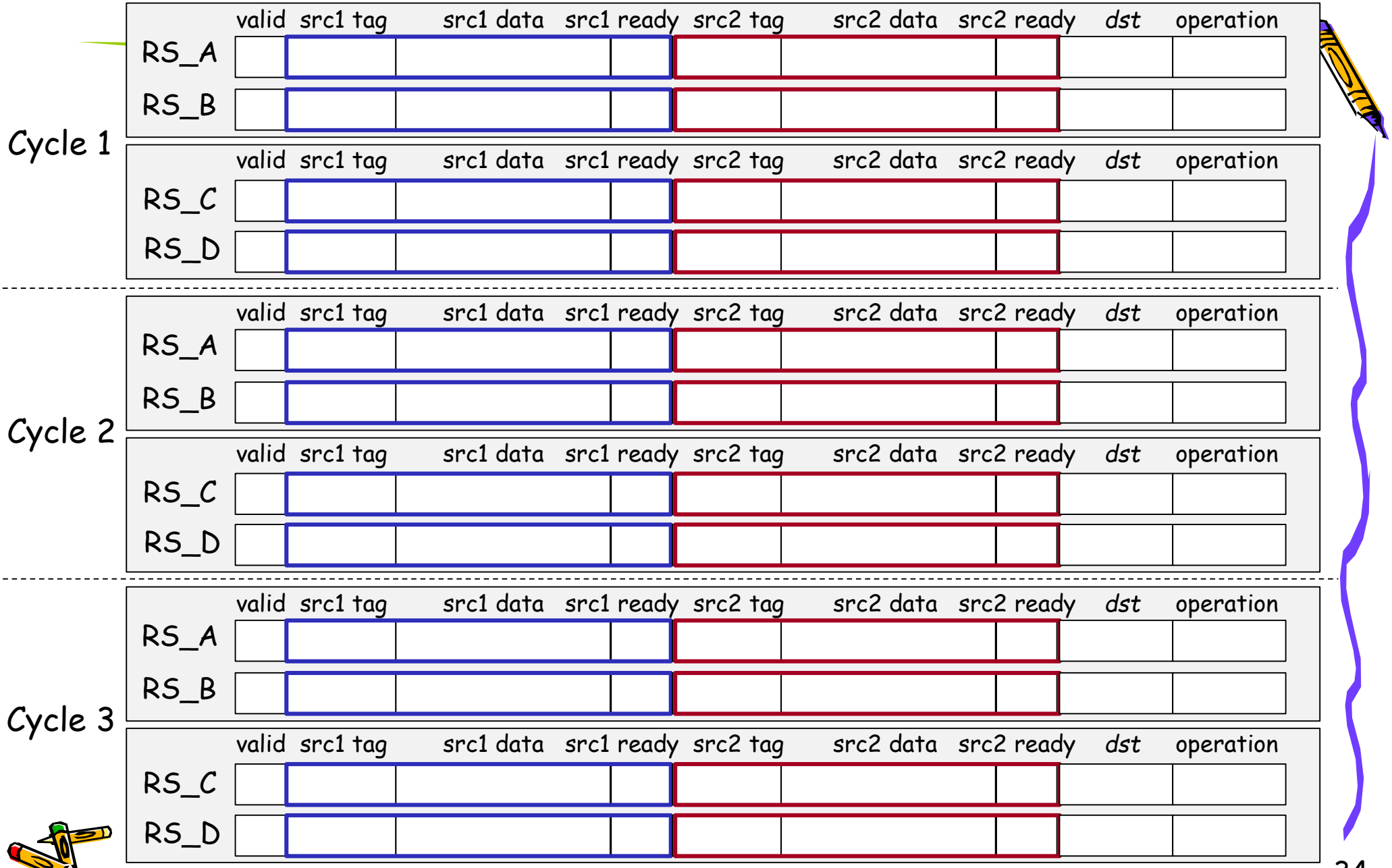
Pollack's Rule

- Pollack's Rule states that microprocessor "performance increase due to microarchitecture advances is roughly proportional to the square root of the increase in complexity". Complexity in this context means processor logic, i.e. its area.

WIKIPEDIA

Exercise 1

Cycle 0 dispatch I1, I2



Exercise 1

Cycle 0 dispatch I1, I2