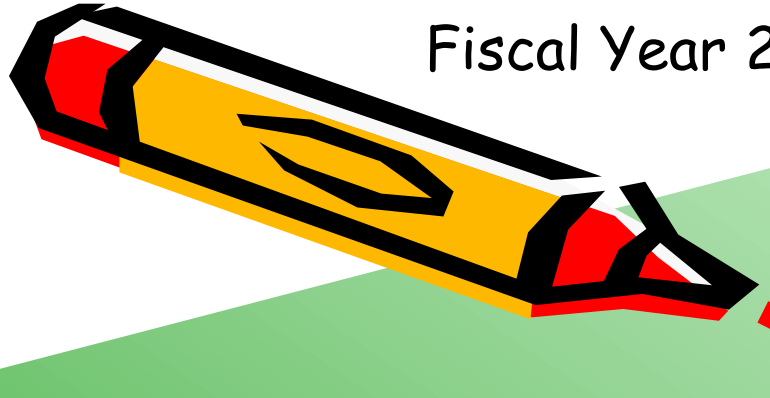
Fiscal Year 2023

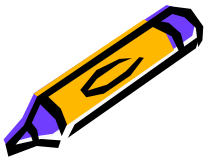
Ver. 2023-01-11a



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

Advanced Computer Architecture

7. Instruction Level Parallelism: Dynamic Scheduling

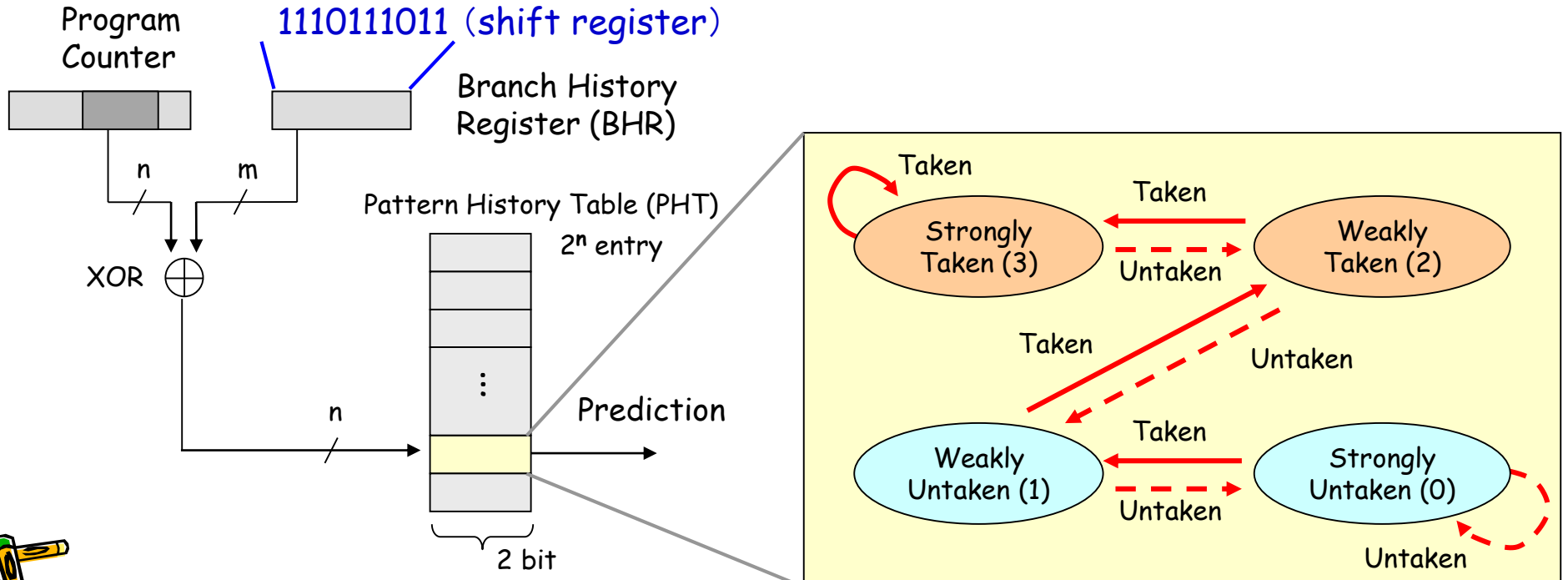


www.arch.cs.titech.ac.jp/lecture/ACA/
Room No.W834, Lecture (Face-to-face)
Mon 13:30-15:10, Thr 13:30-15:10

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

Gshare (TR-DEC 1993)

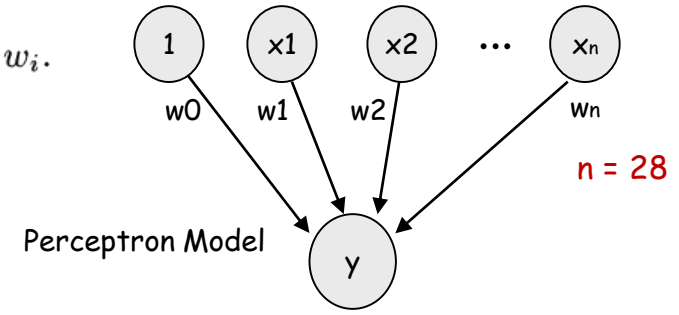
- How to predict
  - Using the exclusive OR of the global branch history and PC to access PHT, then MSB of the selected counter is the prediction.
- How to update
  - Shifting BHR one bit left and update LSB by branch outcome in IF stage.
  - Update the used counter in the same way as 2BC in WB stage.



Perceptron (HPCA 2001)

- How to predict
  - Select one perceptron by PC
  - Compute y using the equation. It predicts 1 if y>=0, predicts 0 if y<0
  - x is branch history. xi is either -1, meaning not taken or 1, meaning taken
- How to update
  - Train the weights of used perceptron when the prediction miss or |y| < T (Threshold)

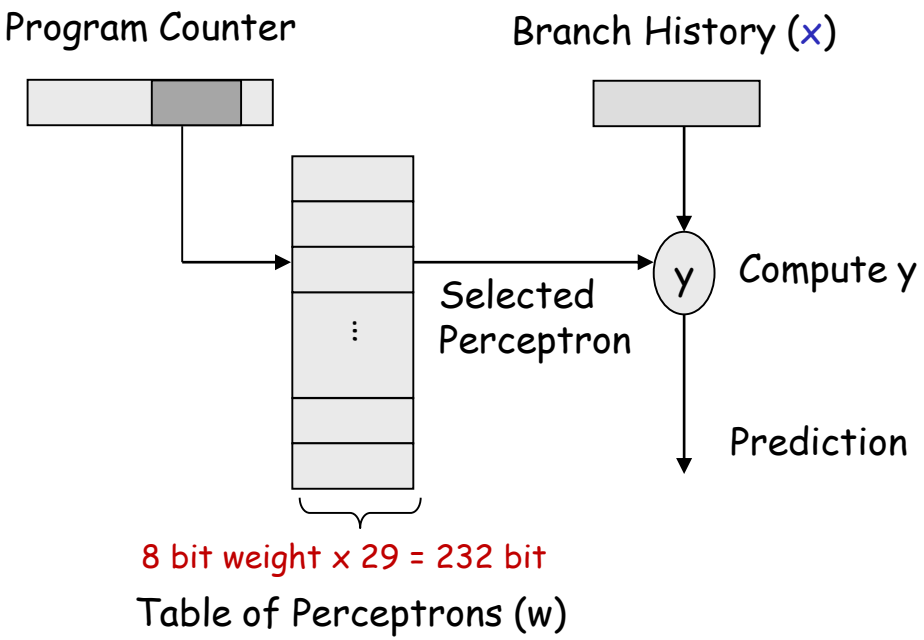
$$y = w_0 + \sum_{i=1}^{n} x_i w_i.$$



branch outcome

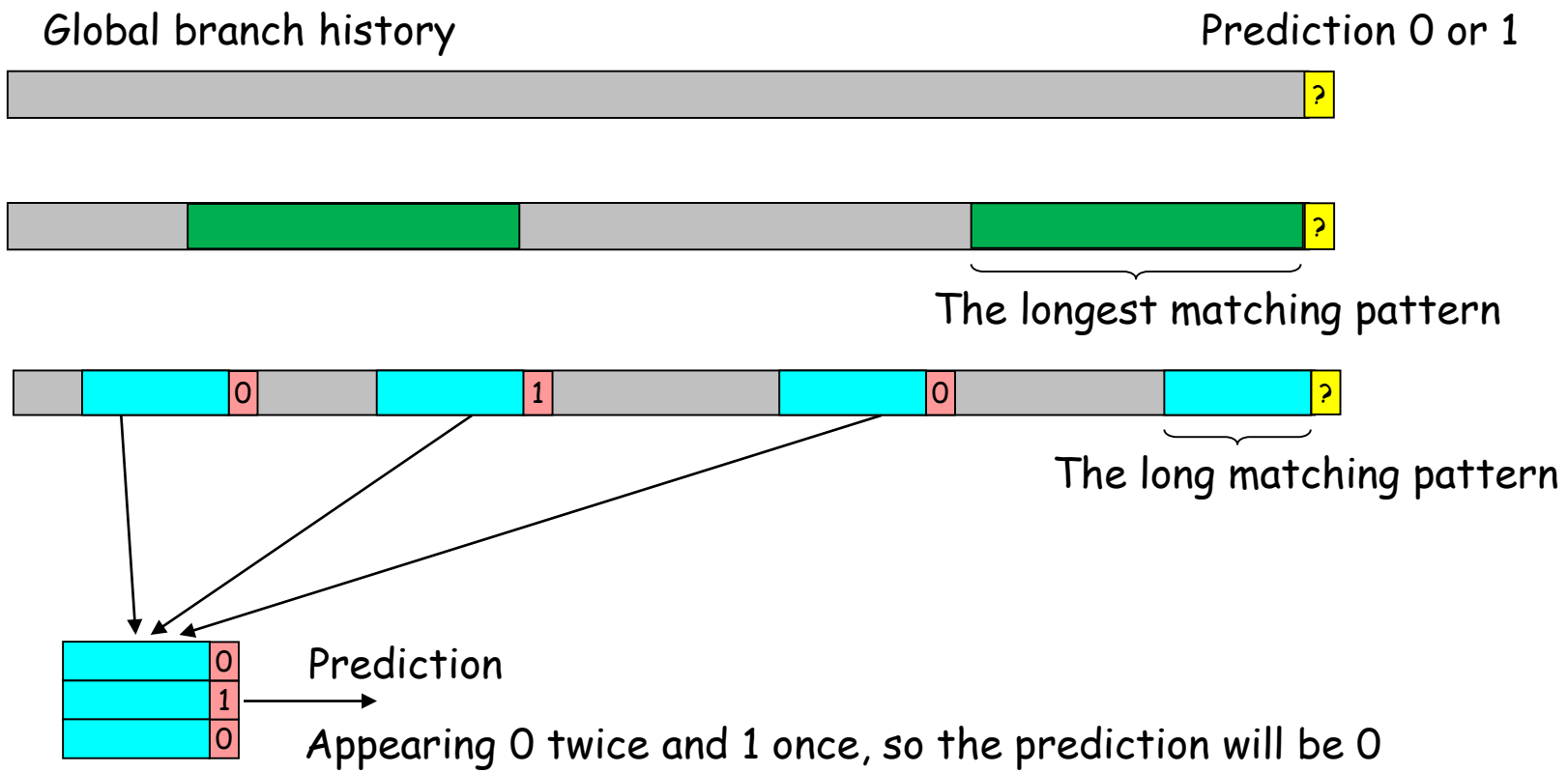
$$\begin{array}{l} \texttt{if } \mathrm{sign}(y_{out}) \neq t \texttt{ or } |y_{out}| \leq \theta \texttt{ then} \\ \quad \texttt{for } i := 0 \texttt{ to } n \texttt{ do} \\ \quad\quad w_i := w_i + t x_i \\ \quad \texttt{end for} \\ \texttt{end if} \end{array}$$

T = 1.93n + 14



Branch predictors based on pattern matching

- Find the longest matching pattern (green rectangle)
- Select the proper matching length or long matching pattern (blue rectangle)
- Count the number of 0 and the number of 1 after the long matting patterns (red rectangle), then predict by majority vote.



Recommended Reading

- Prophet-Critic Hybrid Branch Prediction
  - Ayose Falcon, UPC, Jared Stark, Intel, Alex Ramirez, UPC, Konrad Lai, Intel, Mateo Valero
  - ISCA-31 pp. 250-261 (2004)

A quote from Introduction (1/2)

Conventional predictors are analogous to a taxi with just one driver.

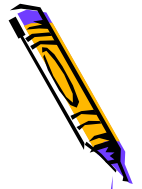
He gets the passenger to the destination using knowledge of the roads acquired from previous trips; i. e., using history information stored in the predictor's memory structures.

When he reaches an intersection, he uses this knowledge to decide which way to turn.

The driver accesses this knowledge in the context of his current location.

Modern branch predictors access it in the context of the current location (the program counter) plus a history of the most recent decisions that led to the current location.

A quote from Introduction (2/2)

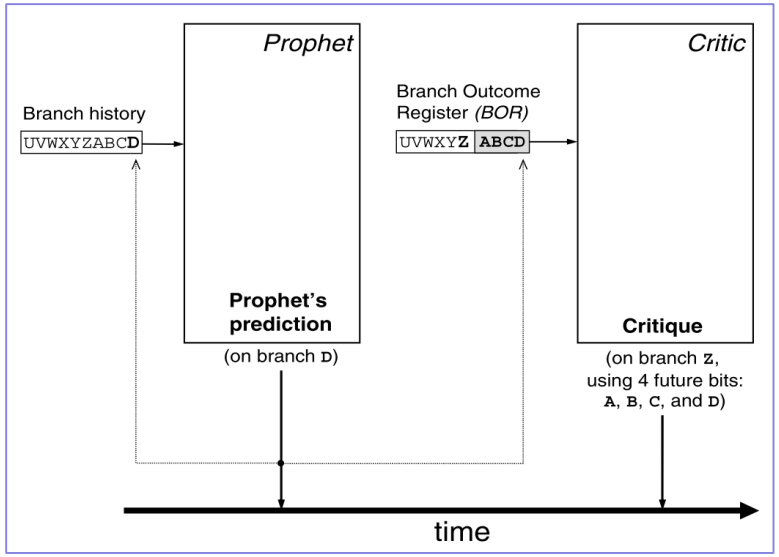


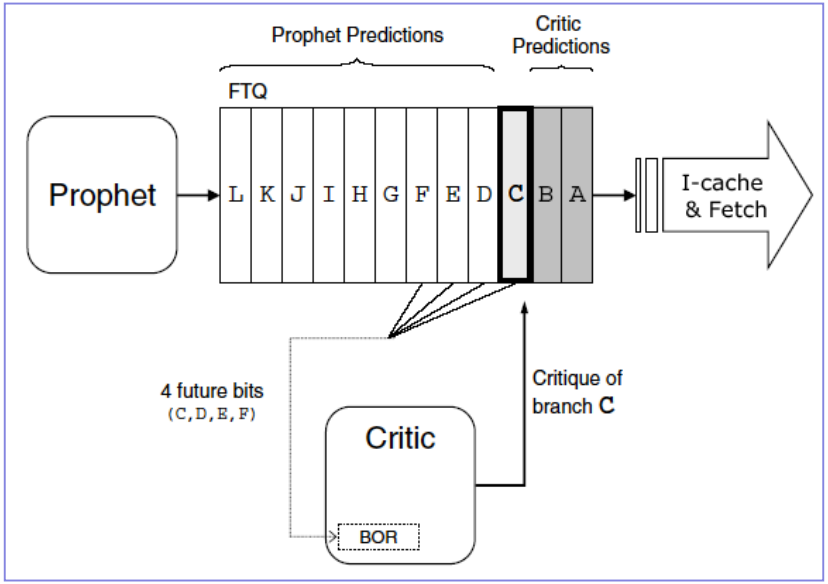
Prophet/critic hybrids are analogous to a taxi with two drivers: the front-seat and the back-seat. The front-seat driver has the same role as the driver in the single-driver taxi. This role is called the prophet.

The back-seat driver has the role of critic. She watches the turns the prophet makes at intersections. She doesn't say anything unless she thinks he's made a wrong turn. When she thinks he's made a wrong turn, she waits until he's made a few more turns to be certain they are lost. (Sometimes the prophet makes turns that initially look questionable, but, after he makes a few more turns, in hindsight appear to be correct.) Only when she's certain does she point out the mistake.

To recover, they backtrack to the intersection where she believes the wrong-turn was made and try a different direction.

Prophet-Critic Hybrid Branch Prediction





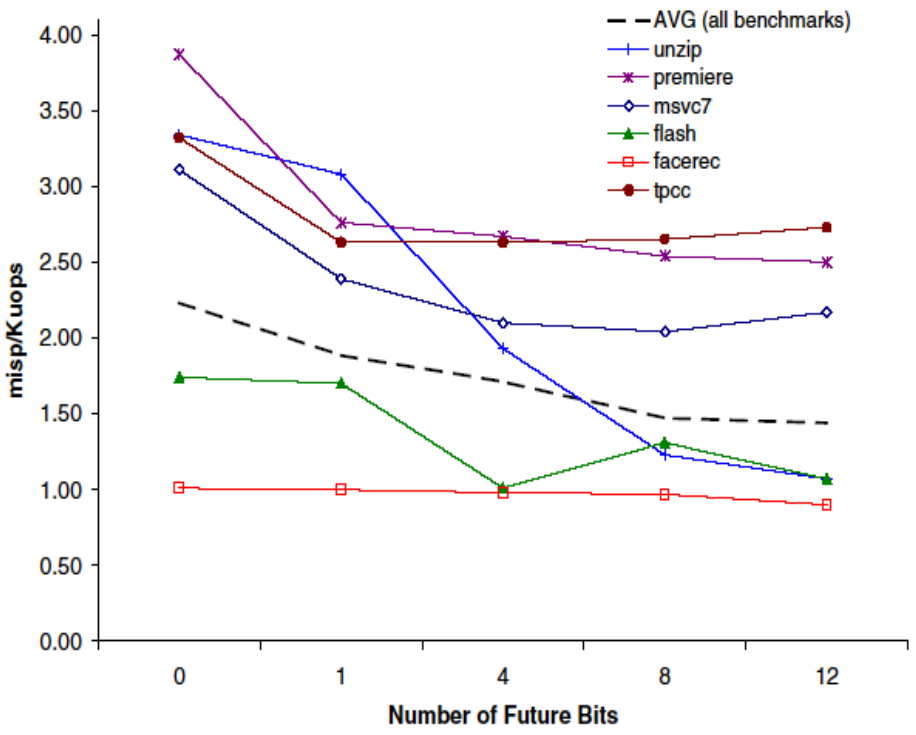
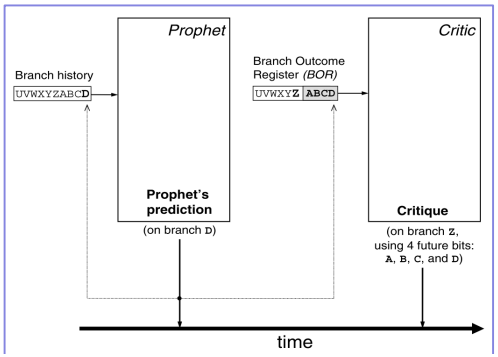


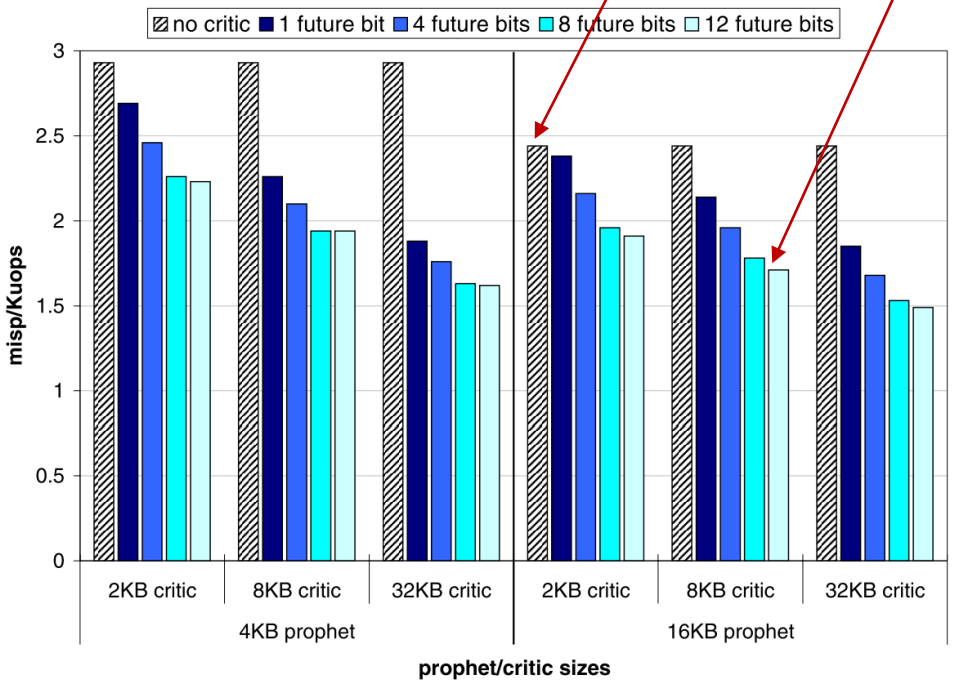
Figure 5. Effect of varying the number of future bits used by the critic on prediction accuracy for selected benchmarks. (prophet: 8KB perceptron; critic: 8KB tagged gshare)

Prophet-Critic Hybrid Branch Prediction



16KB gshare

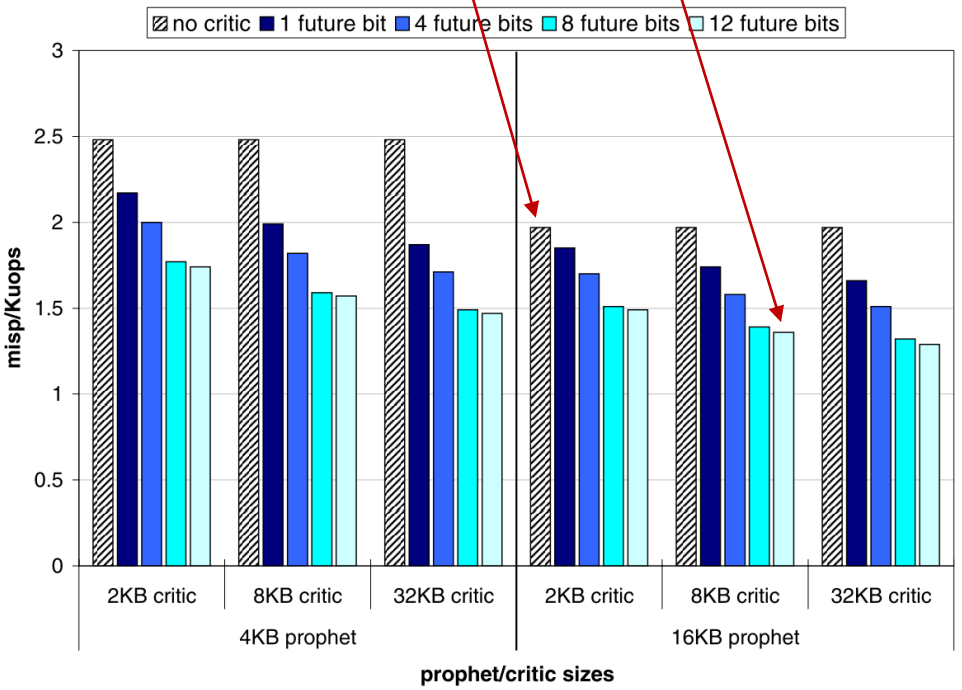
16KB gshare (prophet) + 8KB perceptron (critic)



(b) Prophet: gshare; Critic: filtered perceptron

16KB perceptron

16KB perceptron (prophet) + 8KB gshare (critic)



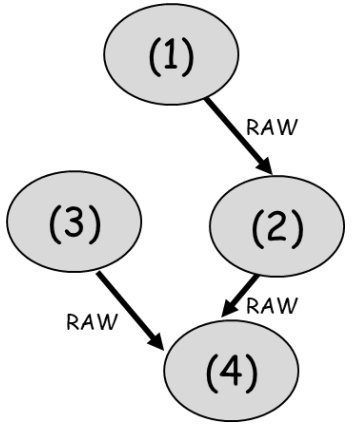
(c) Prophet: perceptron; Critic: tagged gshare

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)
    - 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
```



True data dependence

- Insn i writes a register that insn j reads, RAW (read after write)
- Program order must be preserved to ensure insn j receives the value of insn i.

```
R3 = 10
R5 = 2
R3 = R3 x R5      (1)
R4 = R3 + 1       (2)
R3 = R5 + 3       (3)
R7 = R3 + R4      (4)
```

```
20 = 10 x 2       (1)
21 = 20 + 1       (2)
5  = 2  + 3       (3)
26 = 5  + 21      (4)
```

wrong sequence

```
R3 = 10
R5 = 2
R3 = R3 x R5      (1)
R4 = R3 + 1       (2)
R7 = R3 + R4      (4)
R3 = R5 + 2       (3)
```

```
20 = 10 x 2       (1)
21 = 20 + 1       (2)
41 = 20 + 21      (4)
 5 = 2  + 3       (3)
```

Output dependence

- Insn i and j write the same register, WAW (write after write)
- Program order must be preserved to ensure that the value finally written corresponds to instruction j.

```
R3 = 10
R5 = 2
R3 = R3 x R5      (1)
R4 = R3 + 1       (2)
R3 = R5 + 3       (3)
R7 = R3 + R4      (4)
```

```
20 = 10 x 2       (1)
21 = 20 + 1       (2)
5  = 2  + 3       (3)
26 = 5  + 21      (4)
```

wrong sequence

```
R3 = 10
R5 = 2
R3 = R5 + 3       (3)
R3 = R3 x R5      (1)
R4 = R3 + 1       (2)
R7 = R3 + R4      (4)
```

```
5  = 2  + 3       (3)
20 = 10 x 2       (1)
21 = 20 + 1       (2)
41 = 20 + 21      (4)
```

Antidependence

- Insn i reads a register that insn j writes, WAR (write after read)
- Program order must be preserved to ensure that i reads the correct value.

```
R3 = 10
R5 = 2
R3 = R3 x R5      (1)
R4 = R3 + 1       (2)
R3 = R5 + 3       (3)
R7 = R3 + R4      (4)
```

```
20 = 10 x 2       (1)
21 = 20 + 1       (2)
5  = 2  + 3       (3)
26 = 5  + 21      (4)
```

wrong sequence

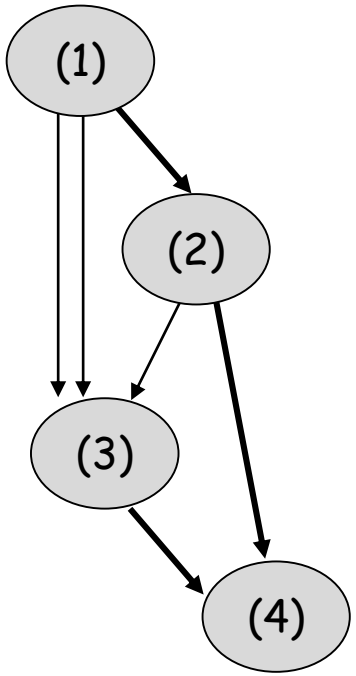
```
R3 = 10
R5 = 2
R3 = R3 x R5      (1)
R3 = R5 + 3       (3)
R4 = R3 + 1       (2)
R7 = R3 + R4      (4)
```

```
20 = 10 x 2       (1)
5  = 2  + 3       (3)
6  = 5  + 1       (2)
11 = 5  + 6       (4)
```

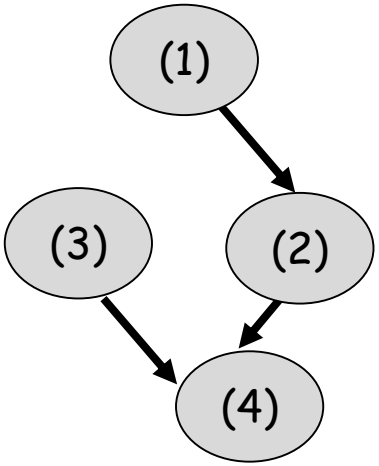
Data dependence and renaming

- True data dependence (RAW)
- Name (false) dependences
  - Output dependence (WAW)
  - Antidependence (WAR)

```
R3 = R3 x R5      (1)
R4 = R3 + 1       (2)
R3 = R5 + 3       (3)
R7 = R3 + R4      (4)
```




```
R3 = R3 x R5      (1)
R4 = R3 + 1       (2)
R8 = R5 + 3       (3)
R7 = R8 + R4      (4)
```



Hardware register renaming

- Logical registers (architectural registers) which are ones defined by ISA
  - x0, x1, … x31
- Physical registers
  - Assuming plenty of registers are available, p0, p1, p2, …
- A processor renames (converts) each logical register to a unique physical register dynamically in the renaming stage

Typical instruction pipeline of scalar processor

Typical instruction pipeline of high-performance superscalar processor

dequeue & allocate

collect & enqueue

Exercise 1

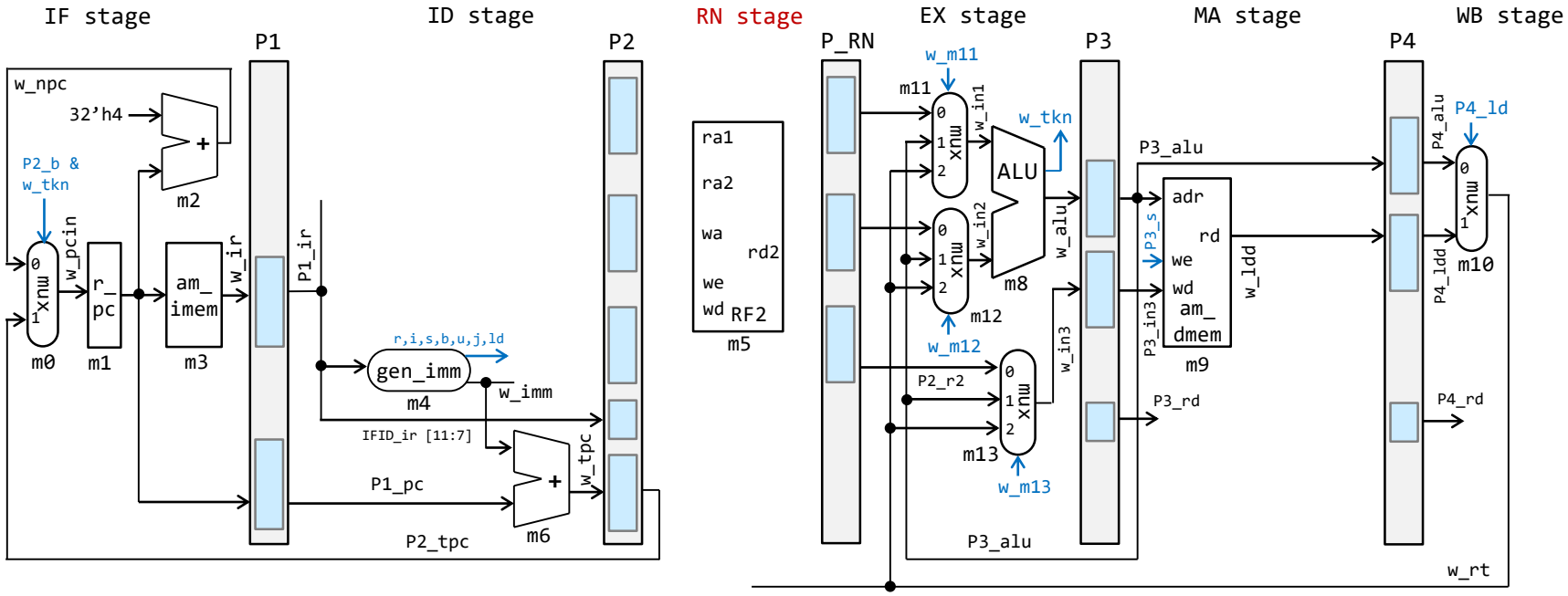
- Register renaming
- Rename the following instruction stream using physical registers of p9, p10, p11, and p12
  - assuming that x1 and x2 are renamed to p1 and p2, respectileby in advance

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

6-stage pipelining RISC-V processor and register renaming

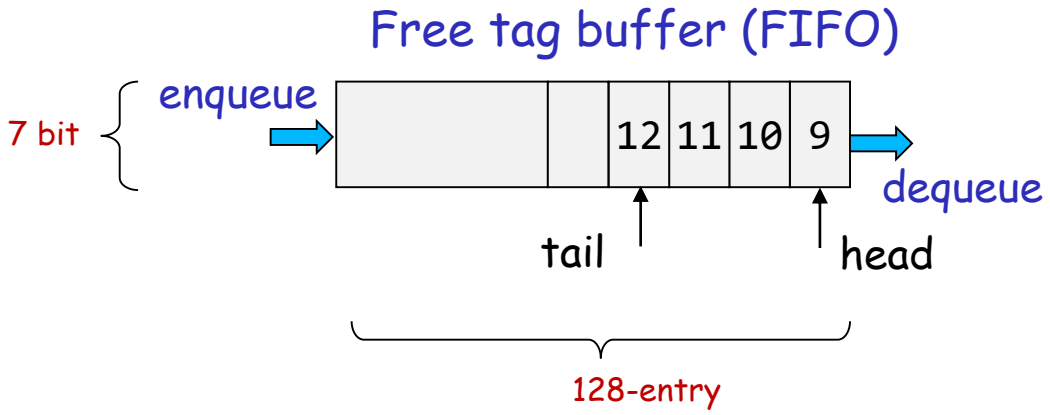
- The strategy is to separate instruction fetch step (IF), instruction decode step (ID), register renaming (RN), execution step (EX), memory access step (MA), and write back step (WB).

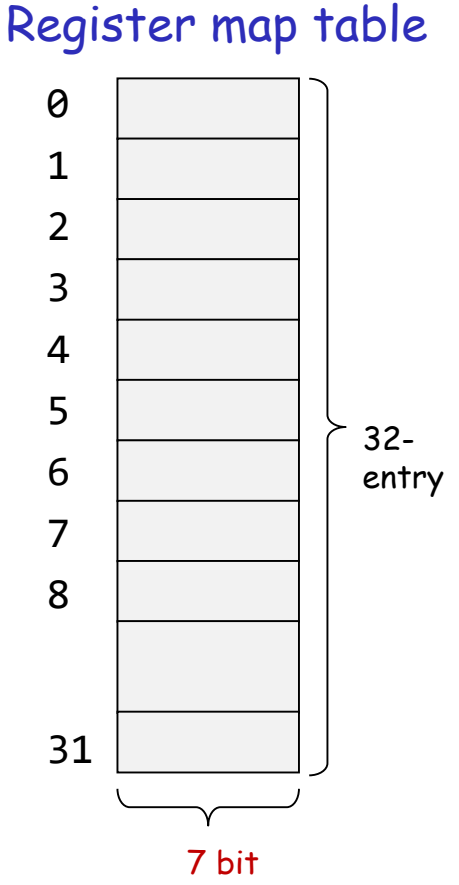
sub x5,x1,x2	sub x5,x1,x2	renaming	sub p9,p1,p2	sub p9,p1,p2	sub x5,x1,x2
in cycle1	in cycle2	in cycle3	in cycle4	in cycle5	in cycle6



The main hardware for register renaming

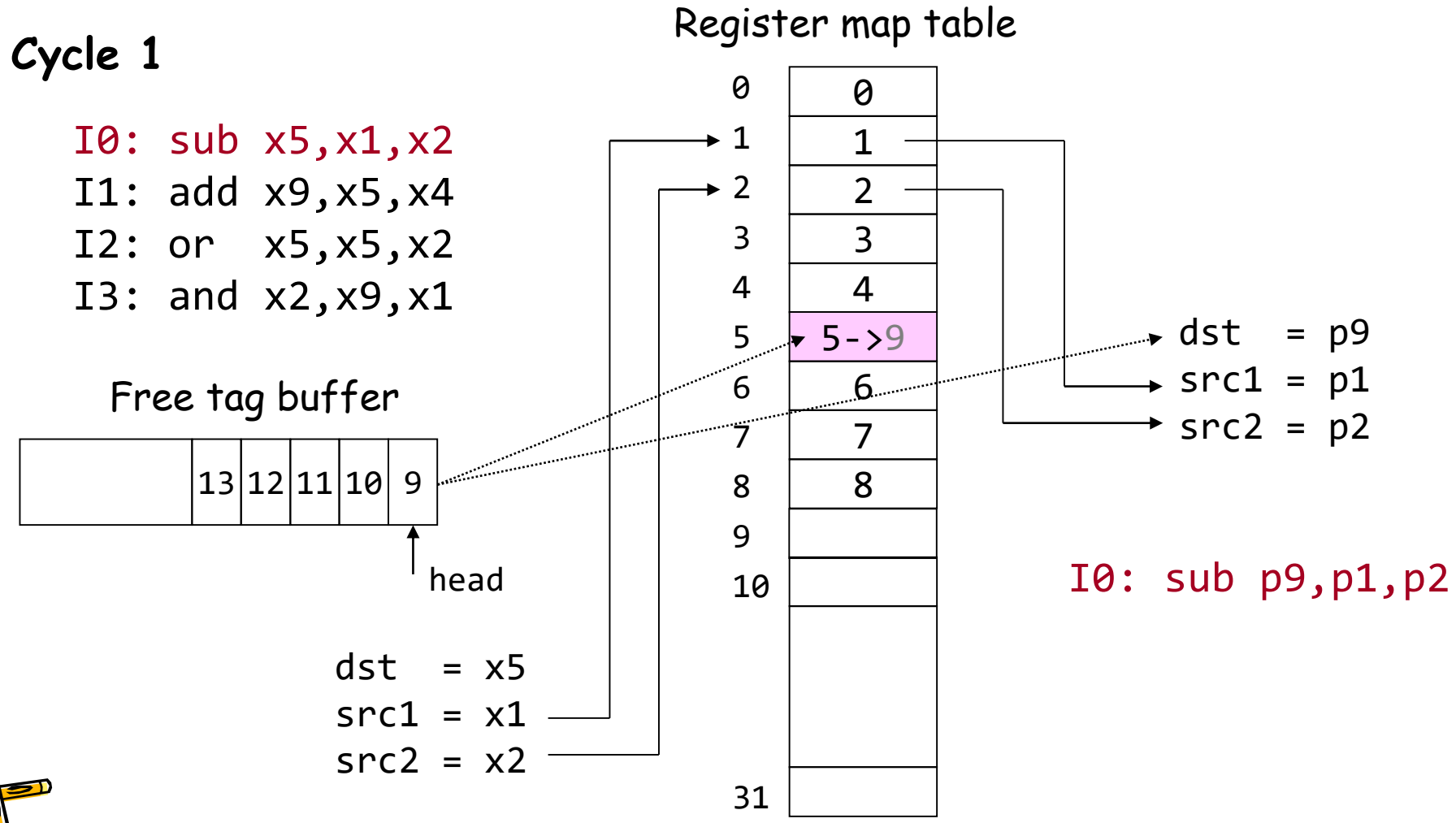
- Assume that we have 128 physical registers from p0 to p127
  - a physical register is identified with a 7-bit register number (physical reg ID)
- Free tag buffer
  - 7-bit width and 128-entry FIFO memory
  - having reg IDs of free (not allocated) physical registers
- Register map table
  - 7-bit width and 32-entry RAM
  - each logical register has its renamed physical reg ID





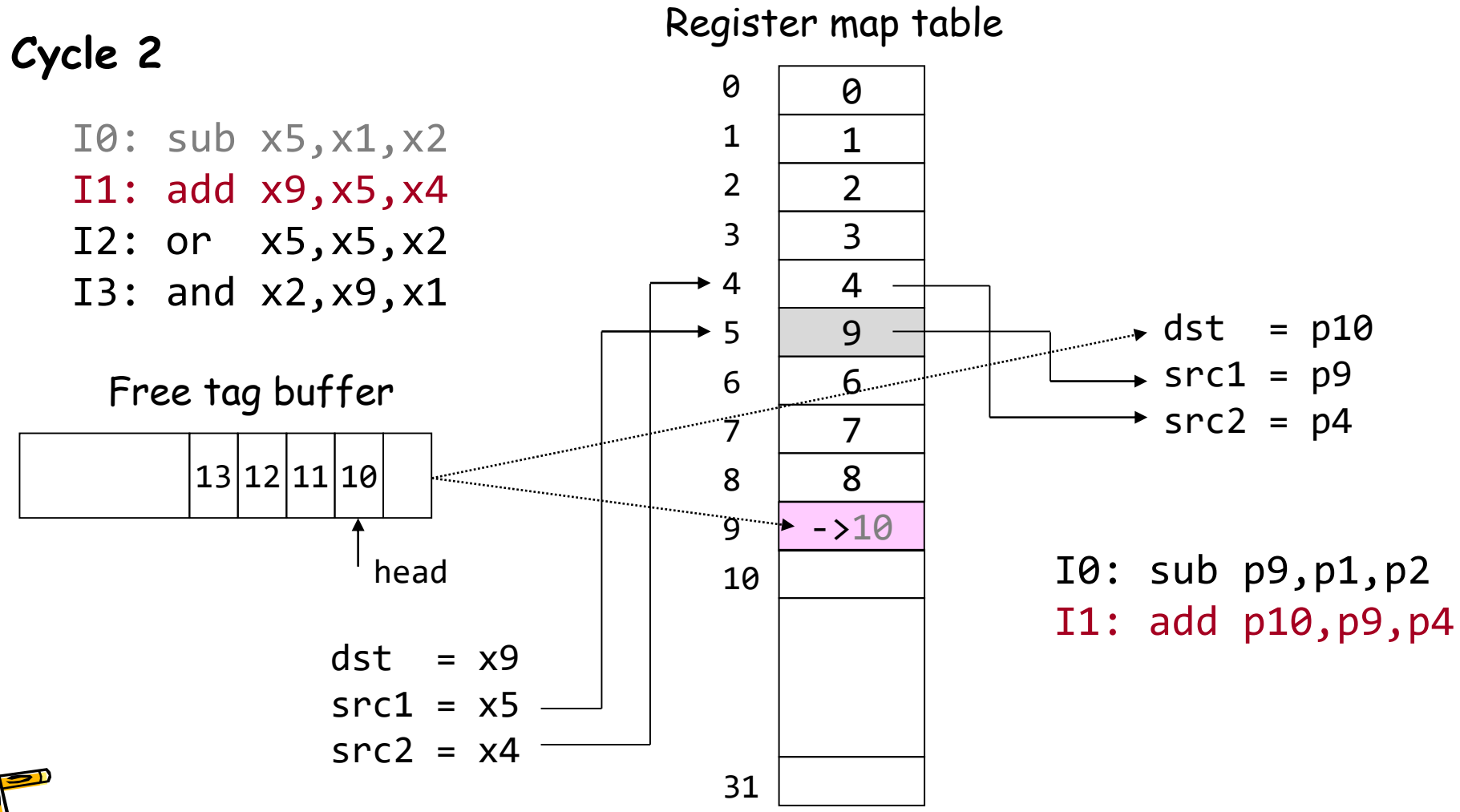
Example behavior of register renaming (1/4)

- Renaming the first instruction I0



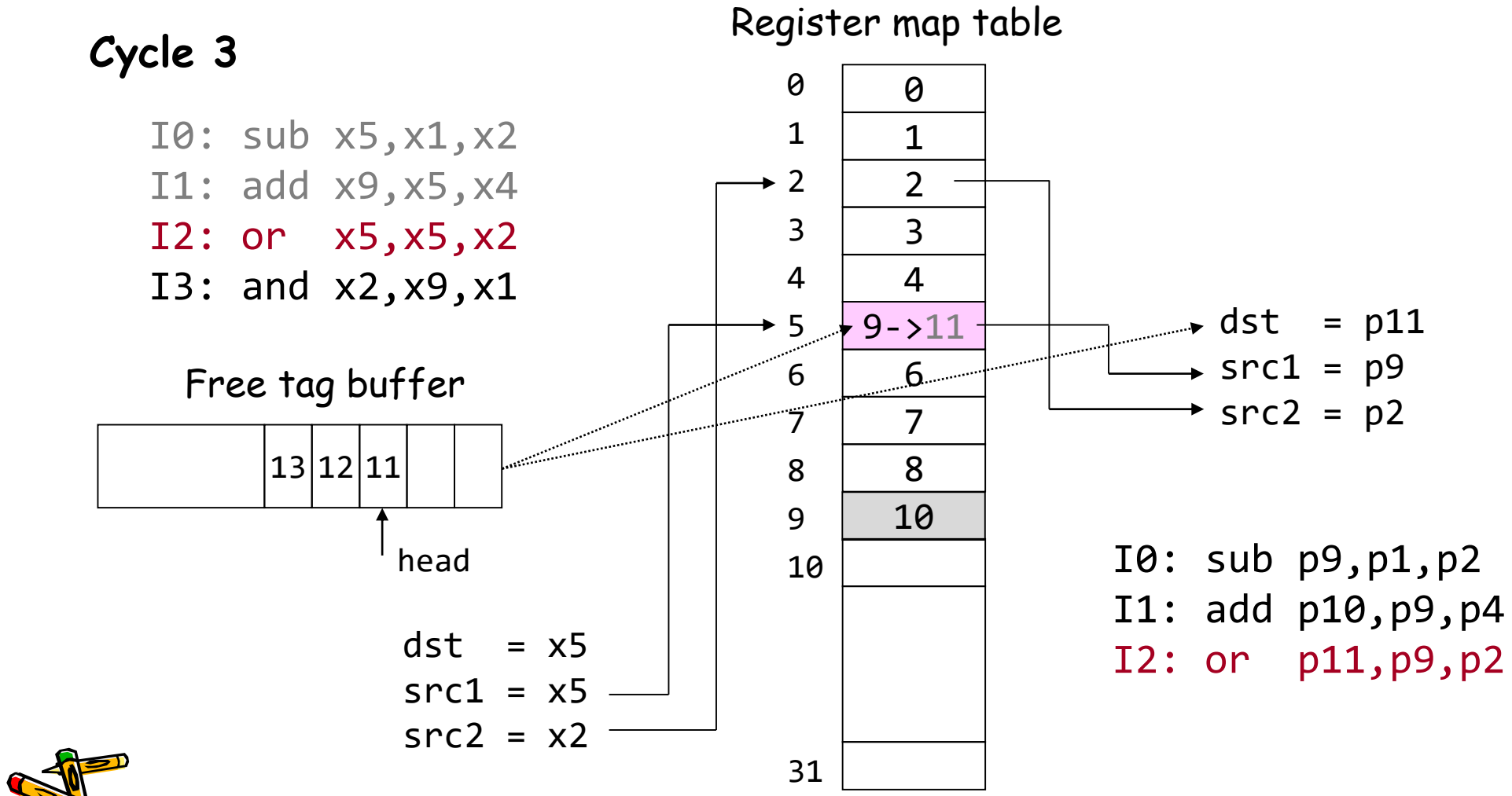
Example behavior of register renaming (2/4)

- Renaming the second instruction I1



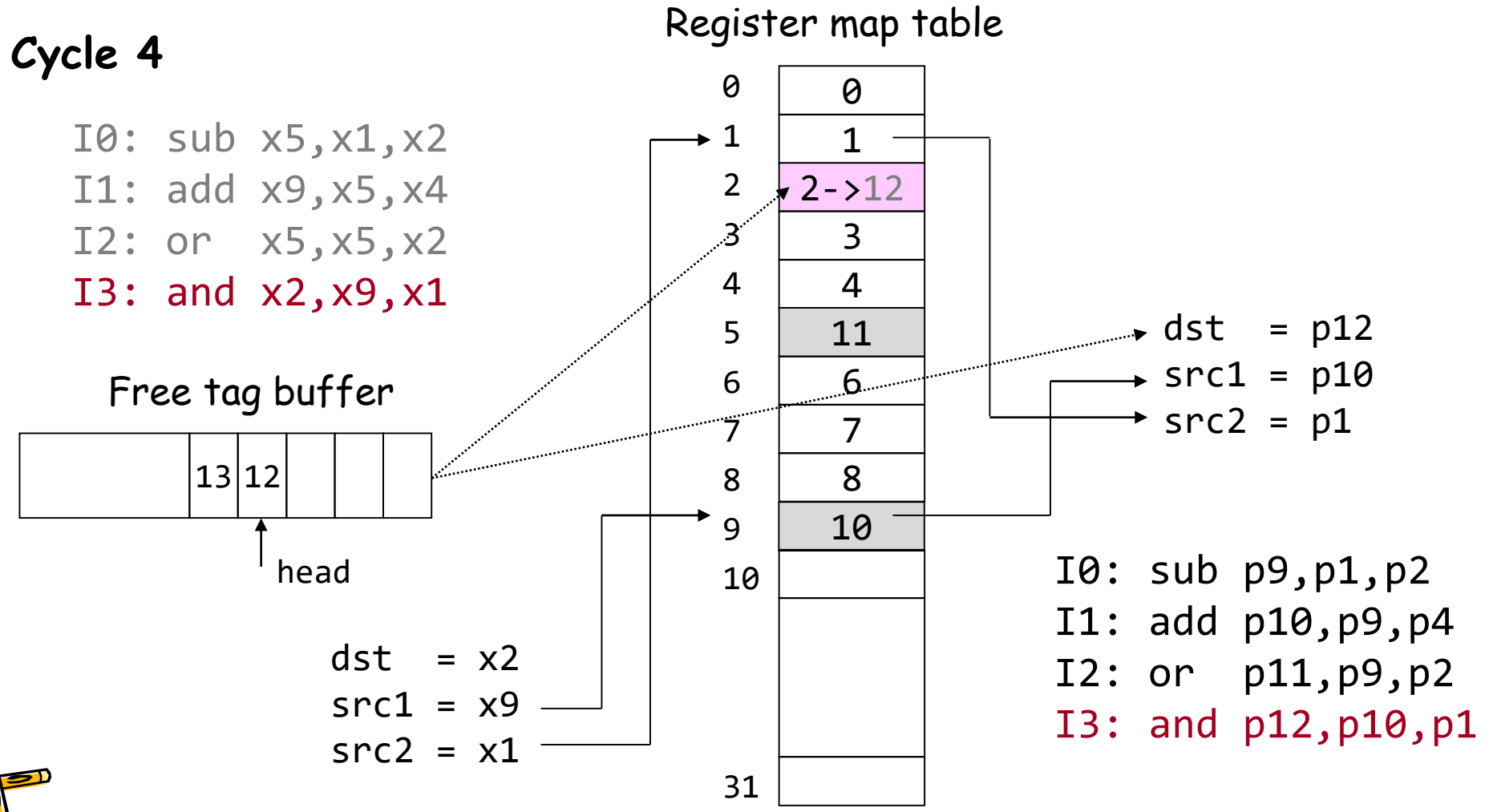
Example behavior of register renaming (3/4)

- Renaming instruction I2



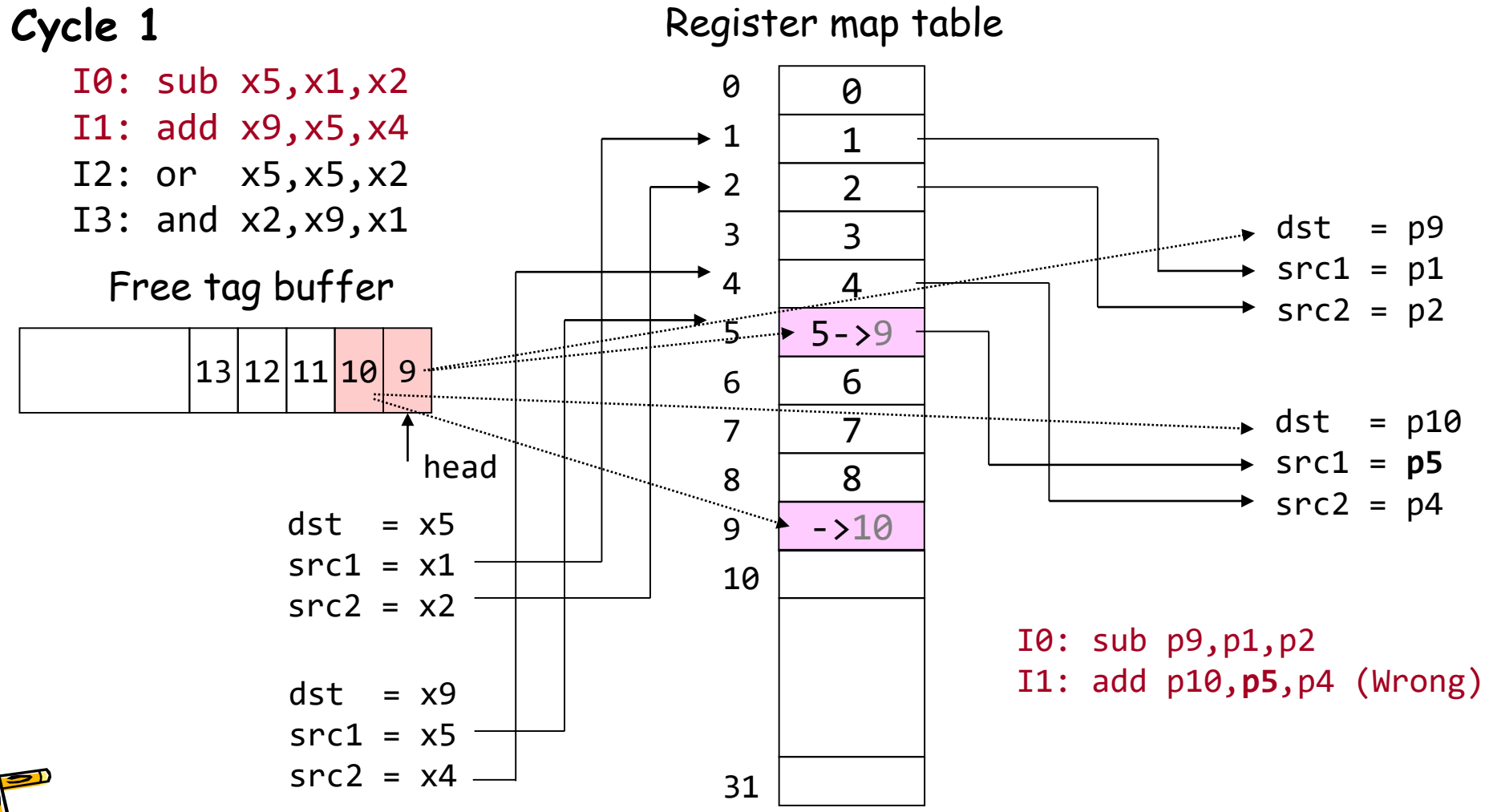
Example behavior of register renaming (4/4)

- Renaming instruction I3



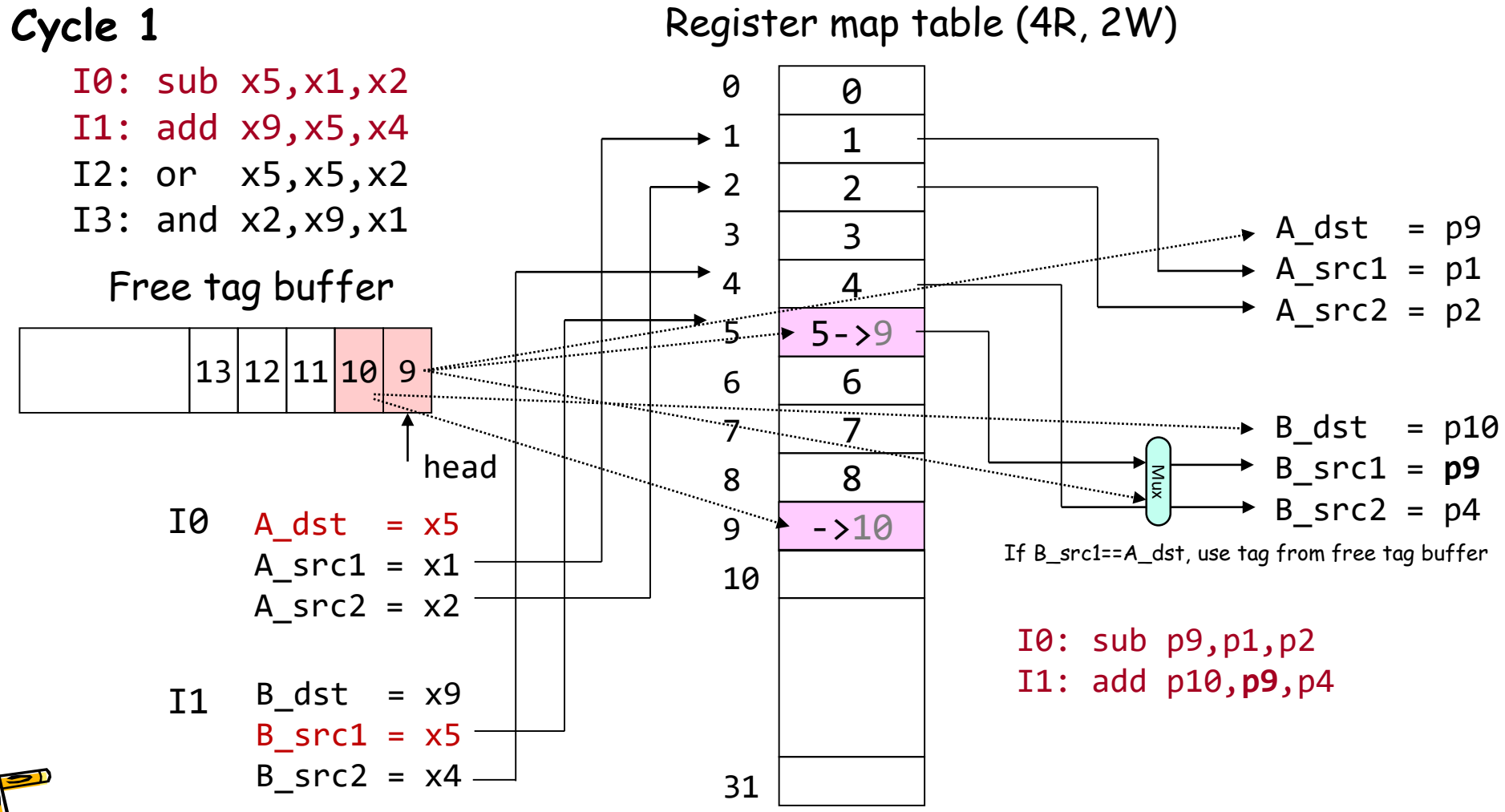
Renaming two instructions per cycle for superscalar

- Renaming instruction I0 and I1



Renaming two instructions per cycle for n-way superscalar

- Renaming instruction I0 and I1 (n = 2)



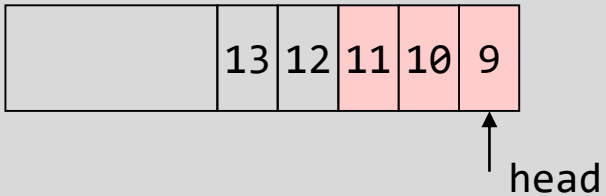
Exercise 2

- Renaming instruction I0, I1, and I2 (n = 3)

Cycle 1

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

Free tag buffer



draw the hardware organization and the example behabior of cycle 1 renaming three instructions.

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

Hardware register renaming

- Logical registers (architectural registers) which are ones defined by ISA
  - x0, x1, … x31
- Physical registers
  - Assuming plenty of registers are available, p0, p1, p2, …
- A processor renames (converts) each logical register to a unique physical register dynamically in the renaming stage

Typical instruction pipeline of scalar processor

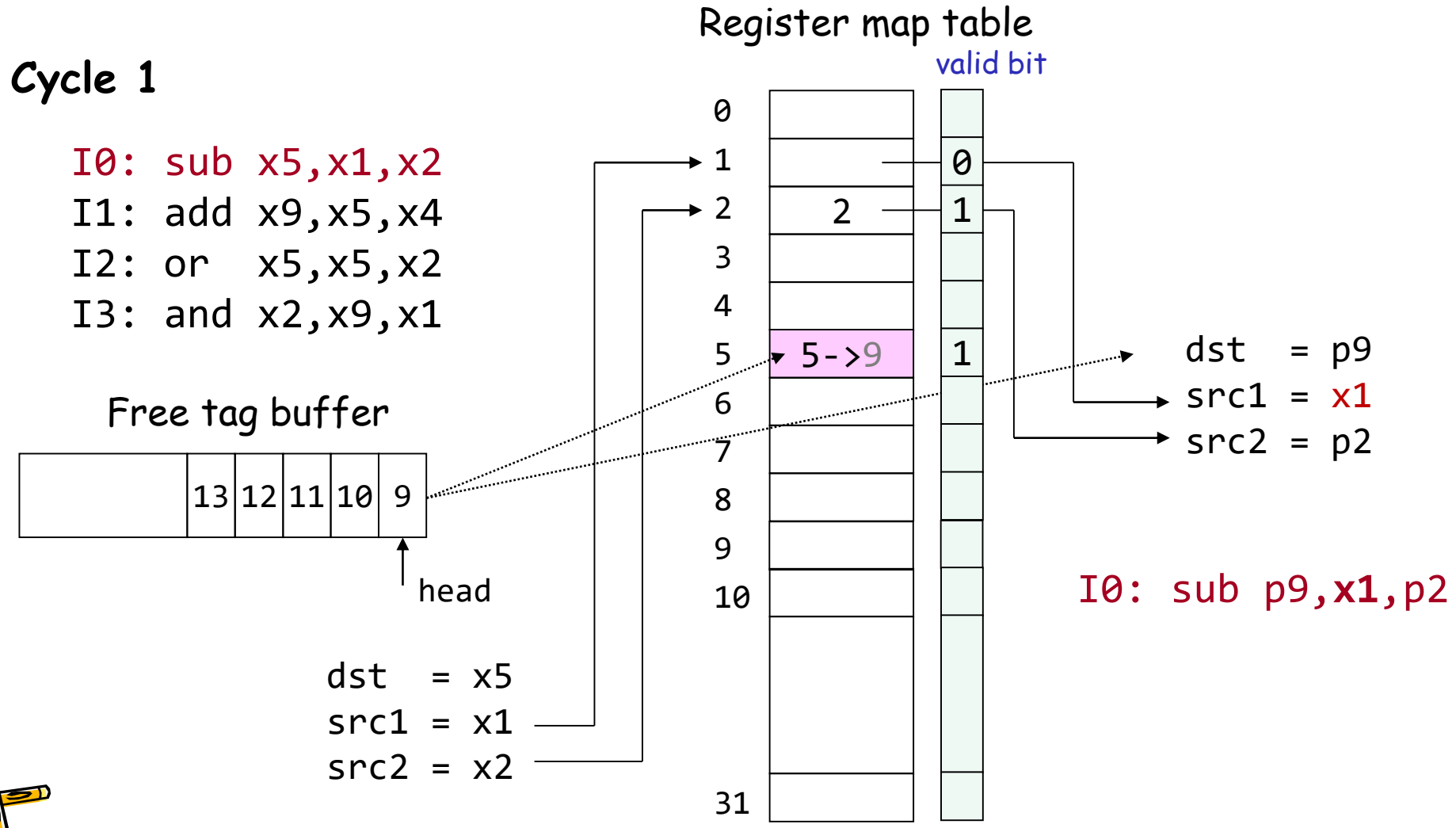
Typical instruction pipeline of high-performance superscalar processor

dequeue & allocate

collect & enqueue

Example behavior of register renaming and valid bit

- Renaming the first instruction I0



True data dependence

- Insn i writes a register that insn j reads, RAW (read after write)
- Program order must be preserved to ensure insn j receives the value of insn i.

```
R3 = 10
R5 = 2
R3 = R3 x R5      (1)
R4 = R3 + 1       (2)
R3 = R5 + 3       (3)
R7 = R3 + R4      (4)
```

```
20 = 10 x 2       (1)
21 = 20 + 1       (2)
5  = 2  + 3       (3)
26 = 5  + 21      (4)
```

wrong sequence

```
R3 = 10
R5 = 2
R3 = R3 x R5      (1)
R4 = R3 + 1       (2)
R7 = R3 + R4      (4)
R3 = R5 + 2       (3)
```

```
20 = 10 x 2       (1)
21 = 20 + 1       (2)
41 = 20 + 21      (4)
 5 = 2  + 3       (3)
```

Recommended Reading

- Focused Value Prediction
  - Sumeet Bandishte, Jayesh Gaur, Zeev Sperber, Lihu Rappoport, Adi Yoaz, and Sreenivas Subramoney, Intel
  - ACM/IEEE 47th Annual 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.