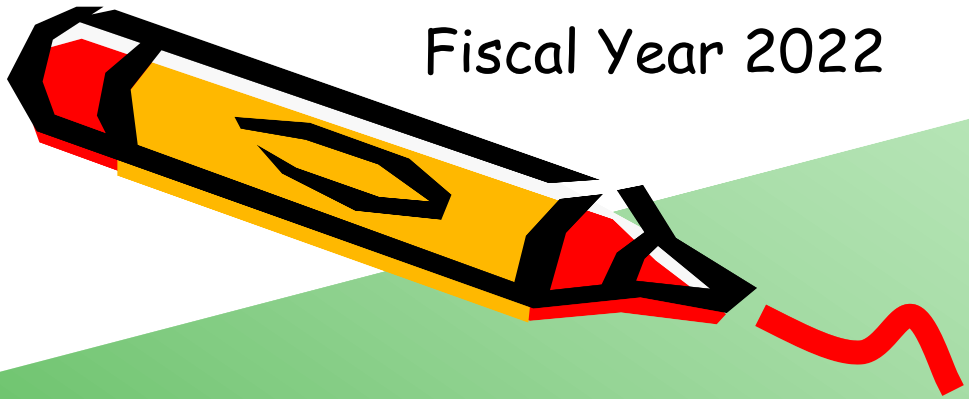
Fiscal Year 2022

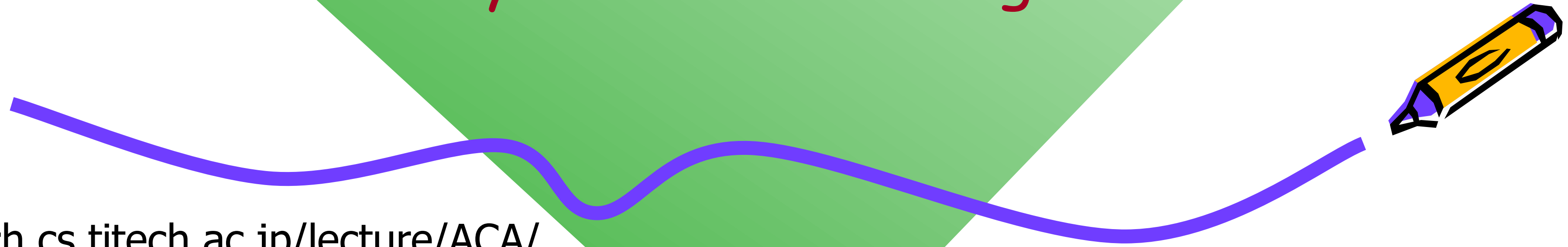
Ver. 2022-01-06a



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.W831, HyFlex
Mon 13:45-15:25, Thr 13:45-15:25

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

Assignment 5

1. Design a four stage pipelined scalar processor supporting MIPS add and bne instruction in Verilog HDL. Please download proc06.v and proc07.v from the support page and refer it.
2. Verify the behavior of designed processor using following assembly code.

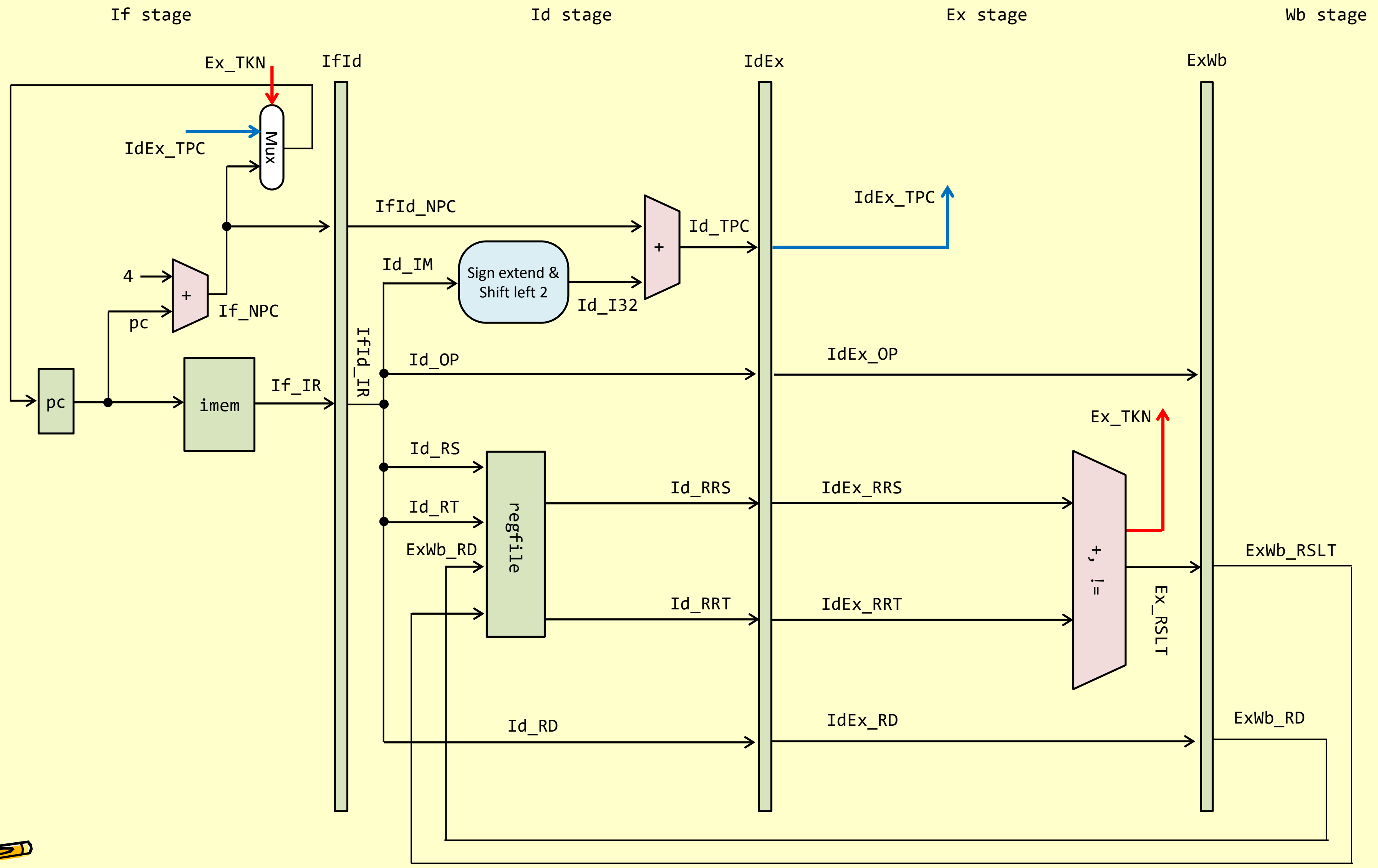
```
p.imem.mem[0] = {6'h0, 5'd0, 5'd0, 5'd0, 5'd0, 6'h20};  //     NOP
p.imem.mem[1] = {6'h0, 5'd5, 5'd1, 5'd5, 5'd0, 6'h20};  // L1: add  $5, $5, $1
p.imem.mem[2] = {6'h0, 5'd0, 5'd0, 5'd0, 5'd0, 6'h20};  //     NOP
p.imem.mem[3] = {6'h0, 5'd0, 5'd0, 5'd0, 5'd0, 6'h20};  //     NOP
p.imem.mem[4] = {6'h0, 5'd0, 5'd0, 5'd0, 5'd0, 6'h20};  //     NOP
p.imem.mem[5] = {6'h5, 5'd4, 5'd5, 16'hfffb};           //     bne  $4, $5, L1
p.imem.mem[6] = {6'h0, 5'd0, 5'd0, 5'd0, 5'd0, 6'h20};  //     NOP
p.imem.mem[7] = {6'h0, 5'd0, 5'd0, 5'd0, 5'd0, 6'h20};  //     NOP
p.imem.mem[8] = {6'h0, 5'd0, 5'd0, 5'd5, 5'd0, 6'h20};  //     add  $5, $0, $0
p.imem.mem[9] = {6'h0, 5'd0, 5'd0, 5'd0, 5'd0, 6'h20};  //     NOP
p.imem.mem[10]= {6'h0, 5'd0, 5'd0, 5'd0, 5'd0, 6'h20};  //     NOP
p.imem.mem[11]= {6'h5, 5'd2, 5'd0, 16'hfff5};           //     bne  $2, $0, L1
p.imem.mem[12]= {6'h0, 5'd0, 5'd0, 5'd0, 5'd0, 6'h20};  //     NOP
p.imem.mem[13]= {6'h0, 5'd0, 5'd0, 5'd0, 5'd0, 6'h20};  //     NOP
p.regfile.r[1] = 1; p.regfile.r[2] = 22; p.regfile.r[3] = 0; p.regfile.r[4] = 4; p.regfile.r[5] = 0;
```

```
while(1){
  for(int i=1; i!=4; i++){

  }
}
```

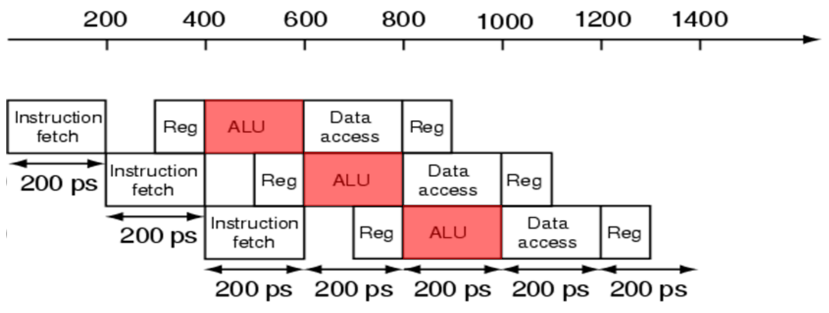
3. Submit your report in a PDF file via E-mail **by the next Thursday**.
   - The report should include a block diagram, a source code in Verilog HDL, and obtained waveforms of your design.

Four stage pipelined processor supporting ADD and BNE, which does not adopt data forwarding (proc08.v, Assignment 5)

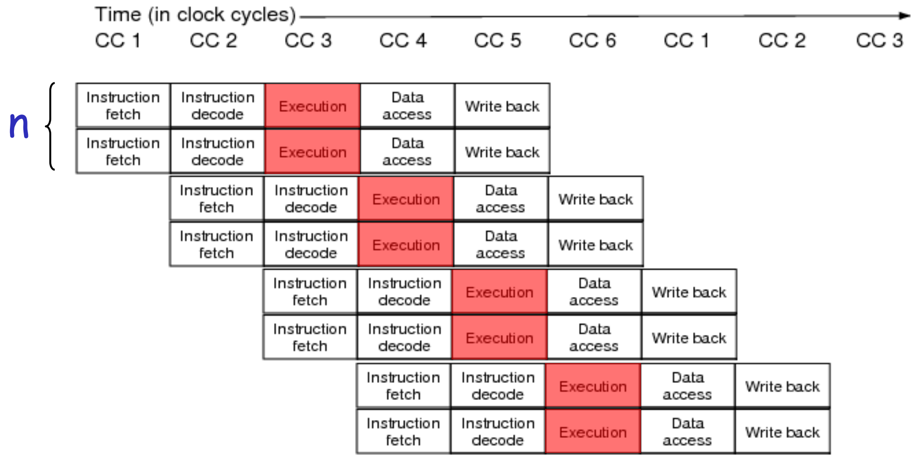


Scalar and Superscalar processors

- Scalar processor can execute at most one instruction per clock cycle by using one ALU.
    - IPC (Executed Instructions Per Cycle) is less than 1.
- Superscalar processor can execute more than one instruction per clock cycle by executing multiple instructions by using multiple pipelines.
    - IPC (Executed Instructions Per Cycle) can be more than 1.
    - using n pipelines is called n-way superscalar



(a) pipeline diagram of scalar processor



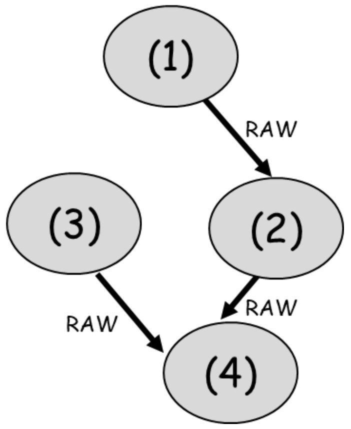
(b) pipeline diagram of 2-way superscalar processor

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)
    - 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 $5,$1,$2
(2) add $9,$5,$3
(3) lw  $4, 4($7)
(4) add $8,$9,$4
```

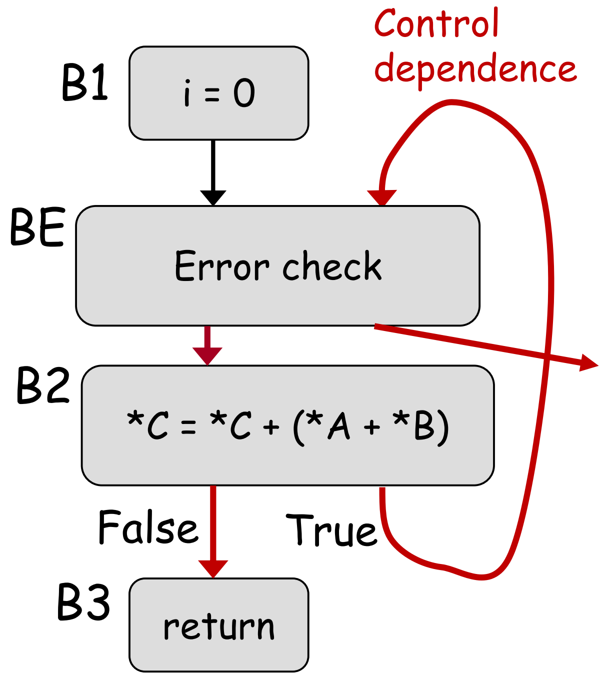
```
(3) lw  $4, 4($7)
(1) add $5,$1,$2
(2) add $9,$5,$3
(4) add $8,$9,$4
```



Exploiting Instruction Level Parallelism (ILP)

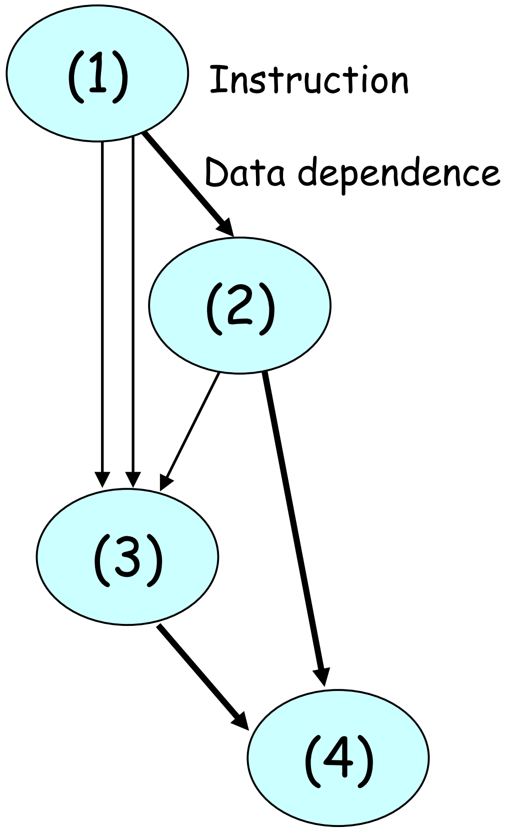
What is the solution?

Prediction & speculation



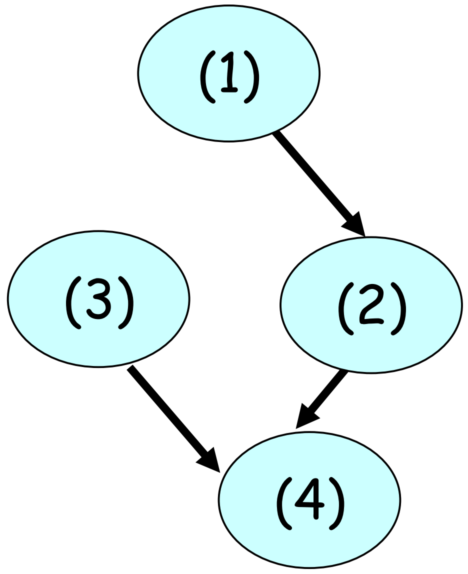
Control flow graph

4 cycles for 4 insns
ILP = 1.0



Data flow graph

3 cycles for 4 insns
ILP = 1.33



Data flow graph

this slide is to be used as a whiteboard

Exercise: what is data dependence

- Draw a data flow graph for each instruction stream

```
R3 = R2 + 1 (1)
R5 = R4 + 2 (2)
R7 = R6 + 3 (3)
```

Instruction stream 1

```
R3 = R2 + 1 (1)
R5 = R4 + 2 (2)
R7 = R3 + 3 (3)
```

Instruction stream 2

```
R3 = R2 + 1 (1)
R3 = R4 + 2 (2)
R7 = R6 + 3 (3)
```

Instruction stream 3

```
R3 = R2 + 1 (1)
R5 = R4 + 2 (2)
R4 = R6 + 3 (3)
```

Instruction stream 4

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

```
Assume R3=10, R5=3
20 = 10 x 2       (1)
21 = 20 + 1       (2)
5  = 3  + 2       (3)
26 = 5  + 21      (4)
```

```
Assume R3=10, R5=3
20 = 10 x 2       (1)
21 = 20 + 1       (2)
41 = 20 + 21      (4)
5  = 3  + 2       (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 = R3 x R5       (1)
R4 = R3 + 1        (2)
R3 = R5 + 2        (3)
R7 = R3 + R4       (4)
```

```
Assume R3=10, R5=3           Assume R3=10, R5=3
20 = 10 x 2        (1)       5  = 3  + 2        (3)
21 = 20 + 1        (2)       20 = 10 x 2        (1)
5  = 3  + 2        (3)       21 = 20 + 1        (2)
26 = 5  + 21       (4)       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 = R3 x R5        (1)
R4 = R3 + 1         (2)
R3 = R5 + 2         (3)
R7 = R3 + R4        (4)
```

```
Assume R3=10, R5=3
20 = 10 x 2         (1)
21 = 20 + 1         (2)
5  = 3  + 2         (3)
26 = 5  + 21        (4)
```

```
Assume R3=10, R5=3
20 = 10 x 2         (1)
5  = 3  + 2         (3)
6  = 5  + 1         (2)
11 = 5  + 6         (4)
```

Exercise: what is data dependence

- Draw a data flow graph for each instruction stream

```
R3 = R2 + 1 (1)
R5 = R4 + 2 (2)
R7 = R6 + 3 (3)
```

Instruction stream 1

```
R3 = R2 + 1 (1)
R5 = R4 + 2 (2)
R7 = R3 + 3 (3)
```

Instruction stream 2

```
R3 = R2 + 1 (1)
R3 = R4 + 2 (2)
R7 = R6 + 3 (3)
```

Instruction stream 3

```
R3 = R2 + 1 (1)
R5 = R4 + 2 (2)
R4 = R6 + 3 (3)
```

Instruction stream 4

Data dependence and renaming

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

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

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

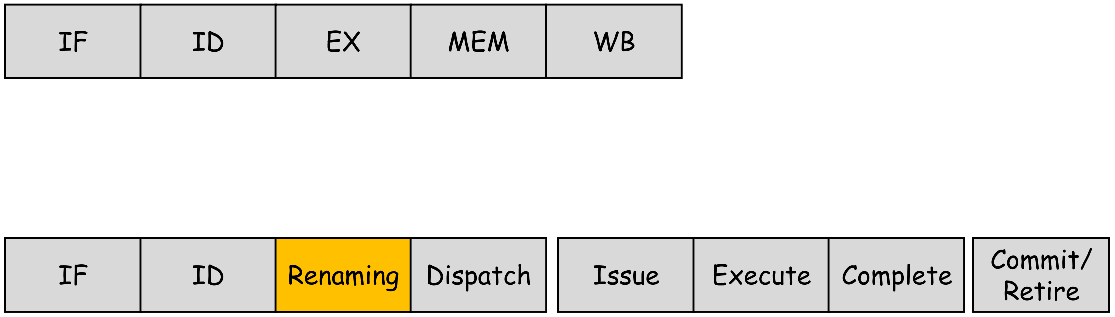
$$\begin{tikzcd}
(1) \arrow[dr, "RAW"] \arrow[dd, bend right, "WAW"'] \arrow[dd, "WAR"] & \\
 & (2) \arrow[dl] \arrow[dd, "RAW"] \\
(3) \arrow[dr, "RAW"'] & \\
 & (4)
\end{tikzcd}$$

$$\begin{tikzcd}
 & (1) \arrow[d, "RAW"] \\
(3) \arrow[dr, "RAW"'] & (2) \arrow[d, "RAW"] \\
 & (4)
\end{tikzcd}$$

this slide is to be used as a whiteboard

Hardware register renaming

- Logical registers (architectural registers) which are ones defined by ISA
  - $0, $1, … $31
- 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



Hardware register renaming

- Logical registers (architectural registers) which are ones defined by ISA
  - $0, $1, … $31
- 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

IF	ID	EX	MEM	WB

Typical instruction pipeline of high-performance superscalar processor

IF	ID	Renaming	Dispatch	Issue	Execute	Commit	Retire

Exercise: register renaming

- Rename the following instruction stream using physical registers of p9, p10, p11, and p12

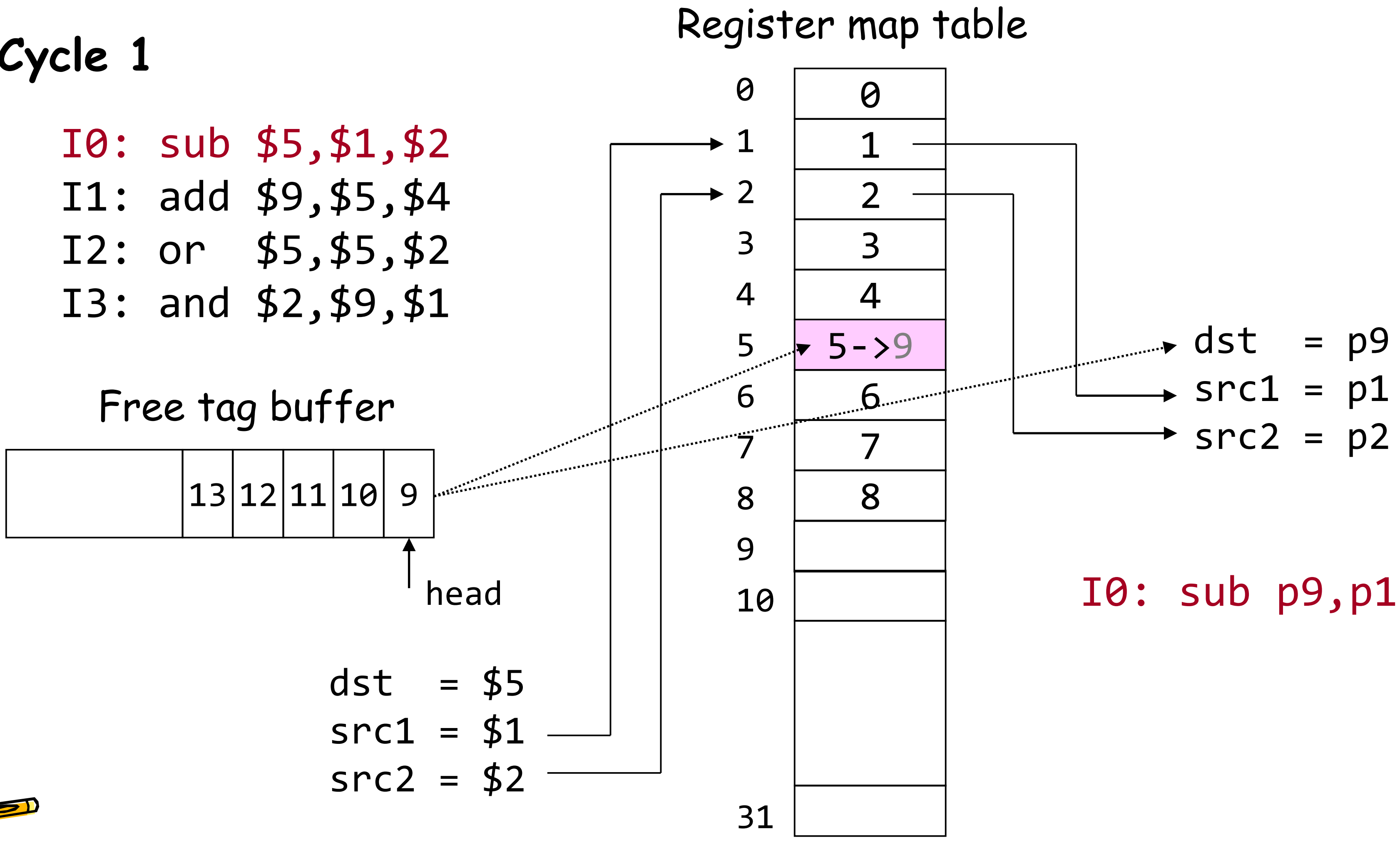
```
I0: sub $5,$1,$2
I1: add $9,$5,$4
I2: or  $5,$5,$2
I3: and $2,$9,$1
```

Example behavior of register renaming (1/4)

- Renaming the first instruction I0

Cycle 1

```
I0: sub $5,$1,$2
I1: add $9,$5,$4
I2: or  $5,$5,$2
I3: and $2,$9,$1
```



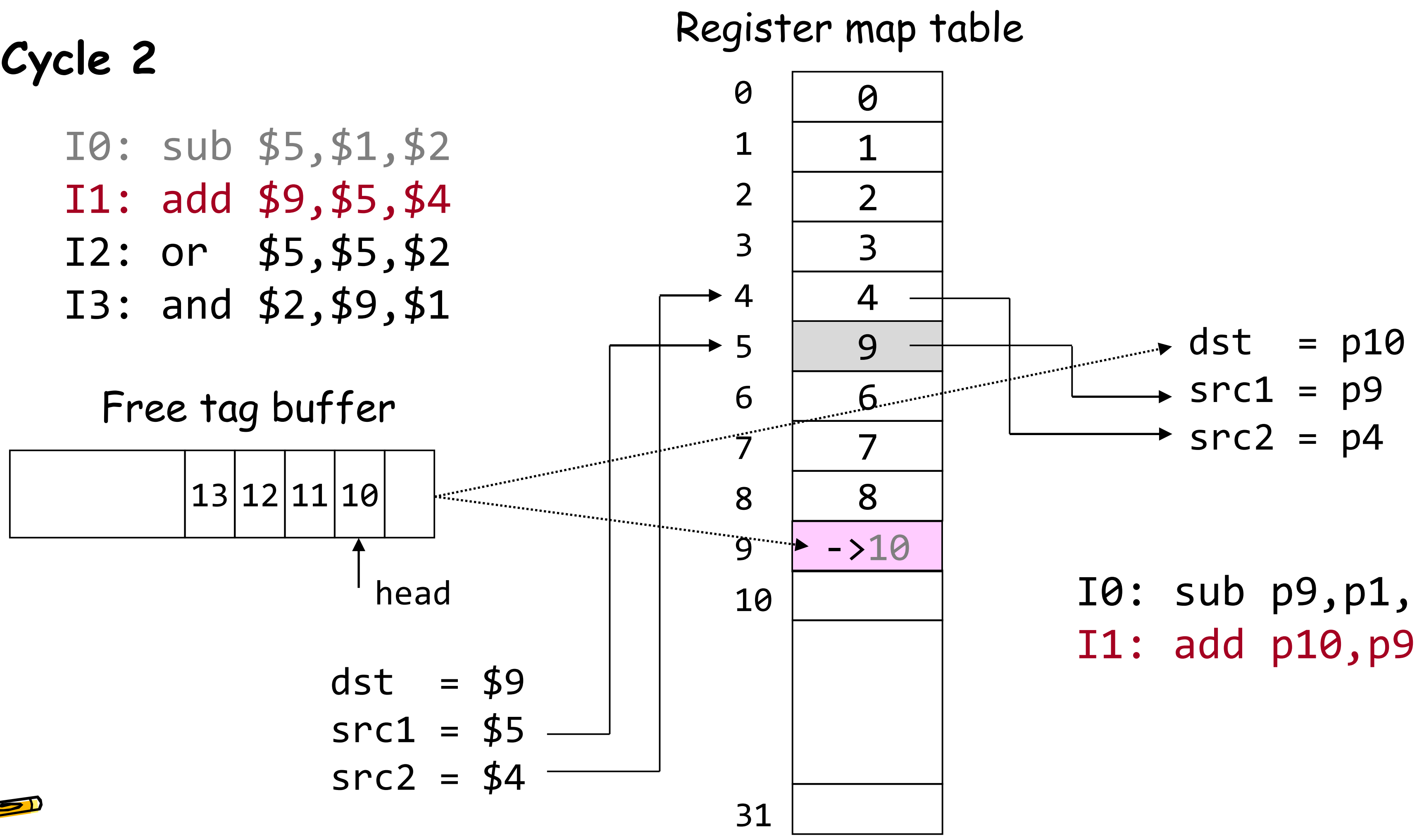
```
I0: sub p9,p1,p2
```

Example behavior of register renaming (2/4)

- Renaming the second instruction I1

Cycle 2

```
I0: sub $5,$1,$2
I1: add $9,$5,$4
I2: or  $5,$5,$2
I3: and $2,$9,$1
```



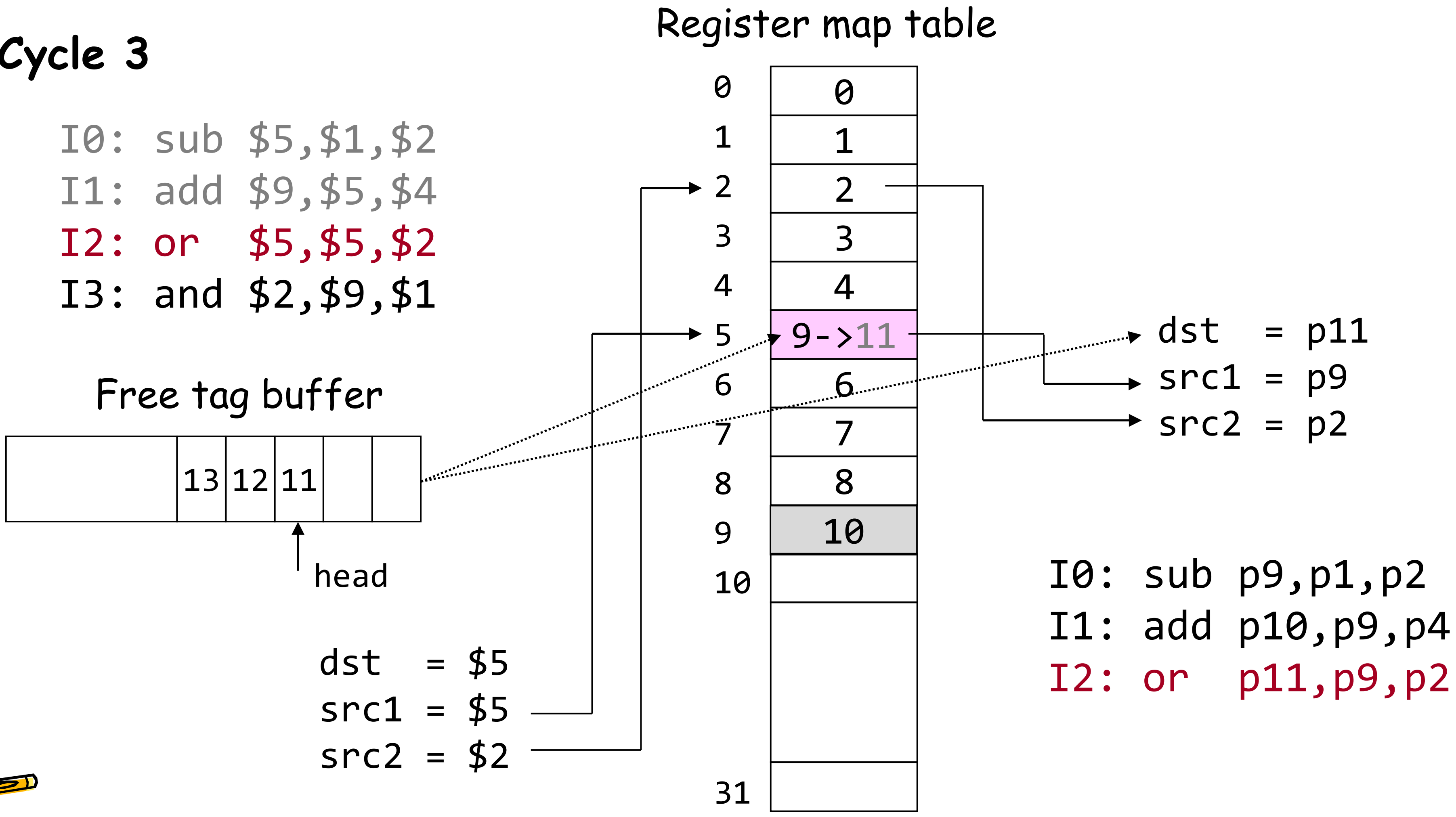
```
I0: sub p9,p1,p2
I1: add p10,p9,p4
```

Example behavior of register renaming (3/4)

- Renaming instruction I2

Cycle 3

```
I0: sub $5,$1,$2
I1: add $9,$5,$4
I2: or  $5,$5,$2
I3: and $2,$9,$1
```



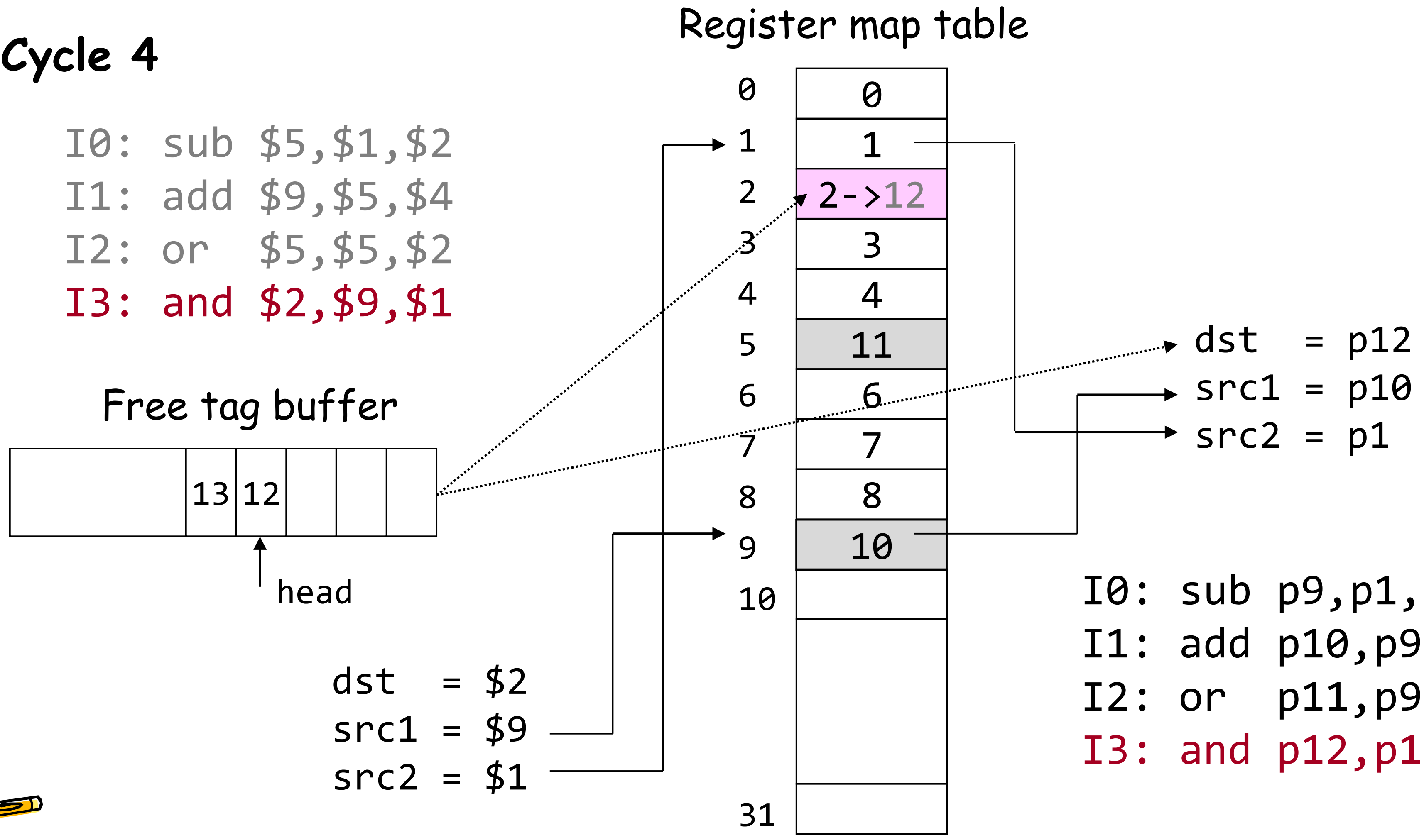
```
I0: sub p9,p1,p2
I1: add p10,p9,p4
I2: or  p11,p9,p2
```

Example behavior of register renaming (4/4)

- Renaming instruction I3

Cycle 4

```
I0: sub $5,$1,$2
I1: add $9,$5,$4
I2: or  $5,$5,$2
I3: and $2,$9,$1
```



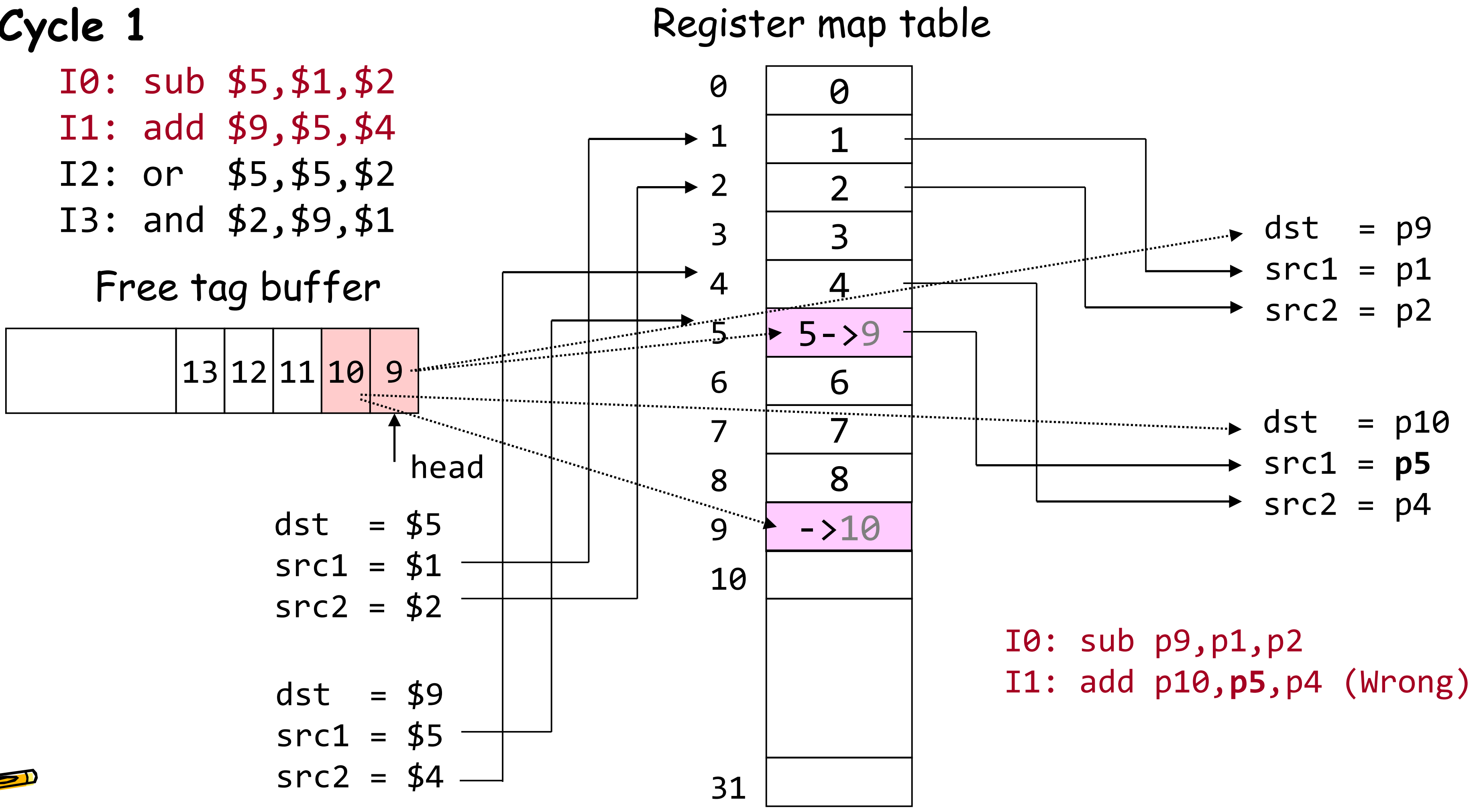
```
I0: sub p9,p1,p2
I1: add p10,p9,p4
I2: or  p11,p9,p2
I3: and p12,p10,p1
```

Renaming two instructions per cycle for superscalar

- Renaming instruction I0 and I1

Cycle 1

```
I0: sub $5,$1,$2
I1: add $9,$5,$4
I2: or  $5,$5,$2
I3: and $2,$9,$1
```



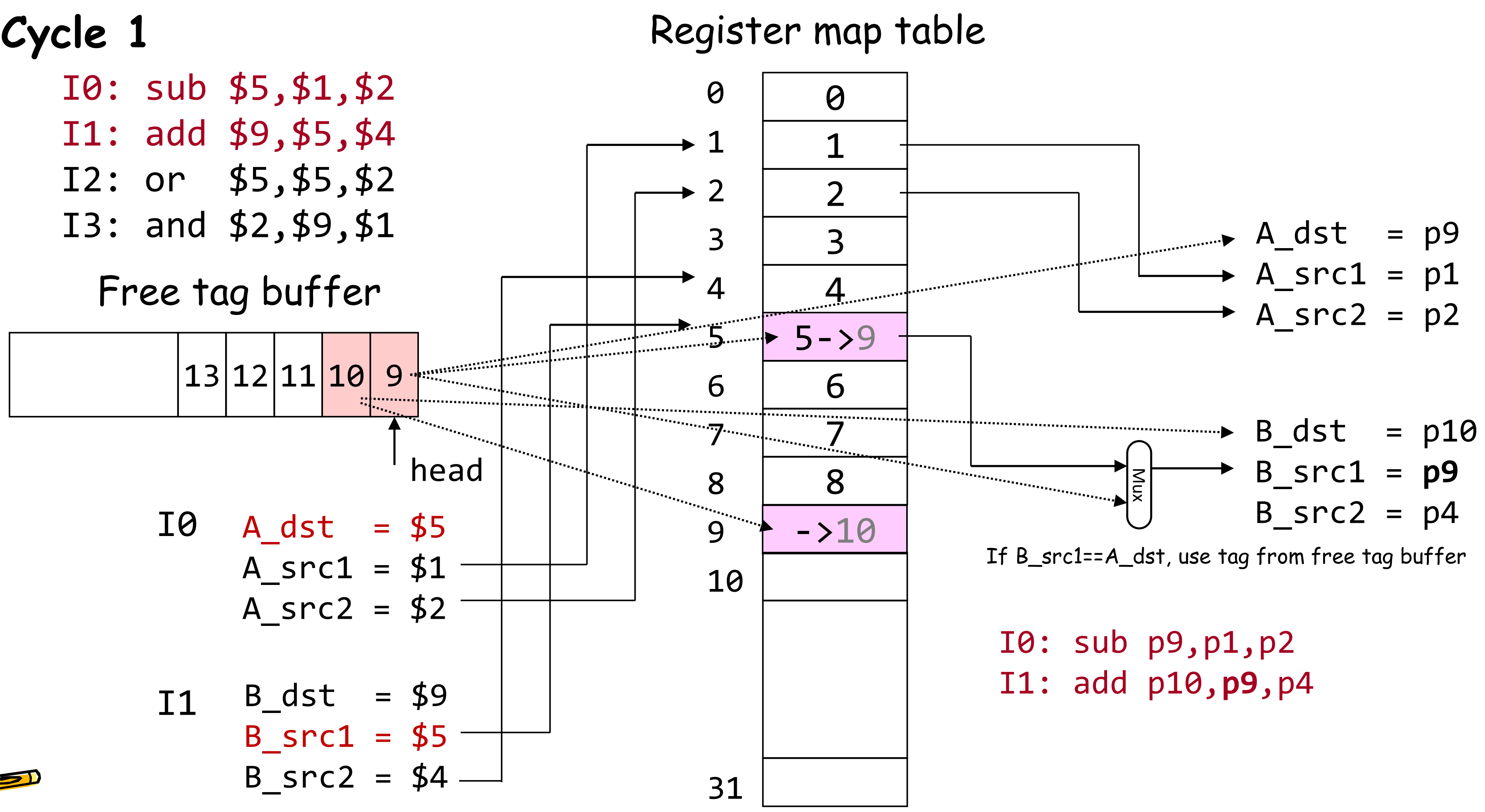
```
I0: sub p9,p1,p2
I1: add p10,p5,p4 (Wrong)
```

Renaming two instructions per cycle for superscalar

- Renaming instruction I0 and I1

Cycle 1

```
I0: sub $5,$1,$2
I1: add $9,$5,$4
I2: or  $5,$5,$2
I3: and $2,$9,$1
```



If B_src1==A_dst, use tag from free tag buffer

```
I0: sub p9,p1,p2
I1: add p10,p9,p4
```

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

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

```
Assume R3=10, R5=3
20 = 10 x 2       (1)
21 = 20 + 1       (2)
5  = 3  + 2       (3)
26 = 5  + 21      (4)
```

```
Assume R3=10, R5=3
20 = 10 x 2       (1)
21 = 20 + 1       (2)
41 = 20 + 21      (4)
5  = 3  + 2       (3)
```

this slide is to be used as a whiteboard

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.

this slide is to be used as a whiteboard

Final report of Advanced Computer Architecture

1. This is a tentative version. The contents may change slightly.
2. Submit your final report describing your answers to questions 1 - 7 in a PDF file
   via E-mail (kise [at] c.titech.ac.jp) by February 13, 2023
   - E-mail title should be "Report of Advanced Computer Architecture"
3. Enjoy!

1. Academic paper reading

- Select an academic paper from the list below and
  - In your own word, describe the problem that the authors try to solve,
  - Describe the key idea of the proposal,
  - Describe your opinion why the authors could solve the problem although there may be many researchers try to solve similar problems.
- List
  - Prophet/critic hybrid branch prediction, ISCA'04, 2004
  - The V-Way Cache: Demand Based Associativity via Global Replacement, ISCA'05, 2005
  - Emulating Optimal Replacement with a Shepherd Cache, MICRO-40, 2008
  - A new case for the TAGE branch predictor, MICRO-44, 2011
  - Skewed Compressed Caches, MICRO-47, 2014
  - Focused Value Prediction, ISCA, 2020

2. MIPS assembly programming

- Write MIPS assembly code asm1.s for code1.c in C.

```
int sum = 0;
int i, j;
for (i=0; i=<100; i++)
  for (j=0; j=<100; j++) sum += (j+i);
```

code1.c

- Write MIPS assembly code asm2.s for code2.c in C.

```
int A[200];
int sum = 0;
int i;
for (i=0; i<200; i++) A[i] = i;              /* initialize the array */
for (i=1; i<200; i++) A[i] = A[i-1] + A[i]; /* compute              */
for (i=0; i<200; i++) sum += A[i];          /* obtain the sum       */
```

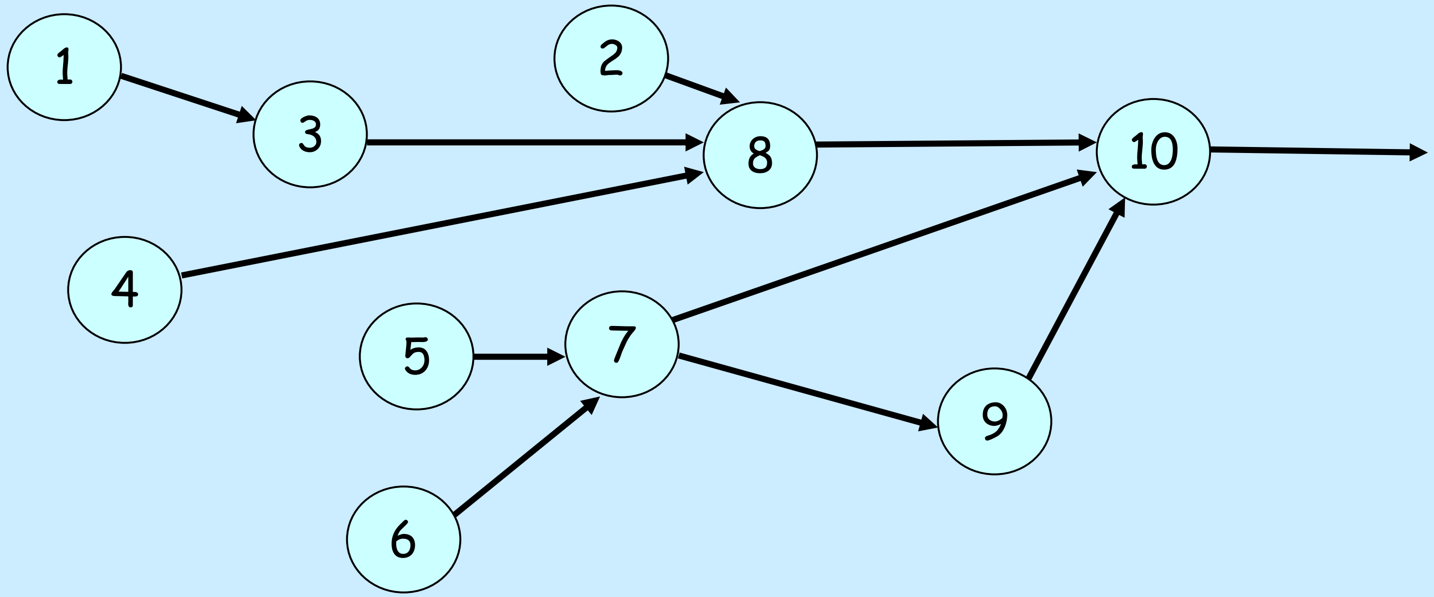
code2.c

3. Pipelined processor

- Design a three stage pipelined scalar processor supporting MIPS add, addi, lw, sw, and bne instructions in Verilog HDL.
  Configure the critical paths of the three stages to have similar delays.
  Please download proc08.v from the support page and refer it.
  Note that you do not need to implement data forwarding.
- Verify the behavior of designed processor using asm1.s and asm2.s.
  You may insert NOP instructions if necessary.
- The report should include a block diagram, a source code in Verilog HDL, the description of the changes of the code, and obtained waveforms of your design.

4. OoO execution and dynamic scheduling

- Draw the cycle by cycle processing behavior of these 10 instructions
- Modify this dataflow graph and draw another cycle by cycle processing behavior of the graph having 10 instructions



5. Parallel programming

- Describe an efficient parallel program for the following sequential program using LOCK(), UNLOCK() and BARRIER() assuming a shared memory architecture
- Explain why your cord runs correctly and why your code is efficient.

main02.c

```
#define N 8       /* the number of grids */
#define TOL 15.0 /* tolerance parameter */
float A[N+2], B[N+2];

void solve () {
    int i, done = 0;
    while (!done) {
        float diff = 0;
        for (i=1; i<=N; i++) { /* use A as input */
            B[i] = 0.333 * (A[i-1] + A[i] + A[i+1]);
        }
        for (i=1; i<=N; i++) { /* use B as input */
            A[i] = 0.333 * (B[i-1] + B[i] + B[i+1]);
            diff = diff + fabsf(B[i] - A[i]);
        }
        if (diff <TOL) done = 1;
        for (i=0; i<=N+1; i++) printf("%6.2f ", B[i]);
        printf("| diff=%6.2f¥n", diff); /* for debug */
    }
}

int main() {
    int i;
    for (i=1; i<N-1; i++) A[i] = B[i] = 100+i*i;
    solve();
}
```

6. Building blocks for synchronization

- Fetch-and-increment reads an original value from memory and increments (adds one to) it in memory atomically
- Implement fetch-and-increment (FAI) using the load-linked/store-conditional instruction pair
  - Refer the discussion of implementing an atomic exchange EXCH
- Implement BARRIER() using FAI

7. Cache coherence protocols

- Select your favorite commercial multi-core processor
    - Describe the memory organization including caches and main memory
        - cache line size, write policy, write allocate/no-allocate, direct-mapped/set-associative, the number of caches (L1, L2, and L3?)
    - Describe the cache coherence protocol used there