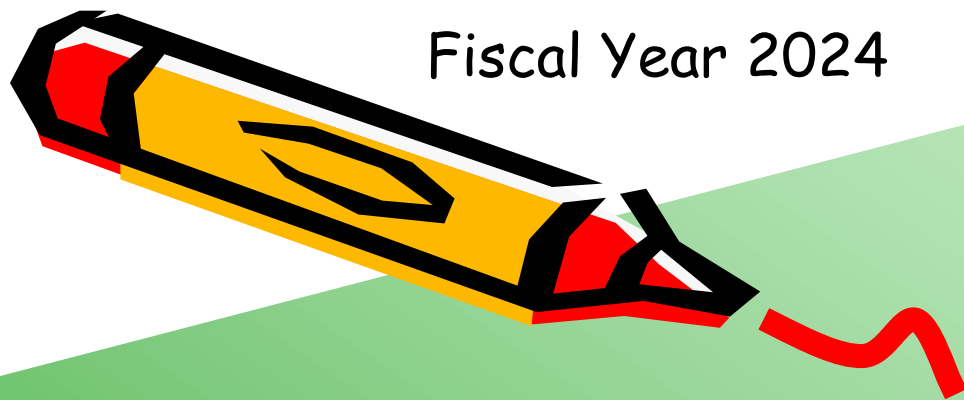


Fiscal Year 2024

Ver. 2025-01-08a



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. W8E-308, Lecture (Face-to-face)
Mon 13:30-15:10, Thr 13:30-15:10

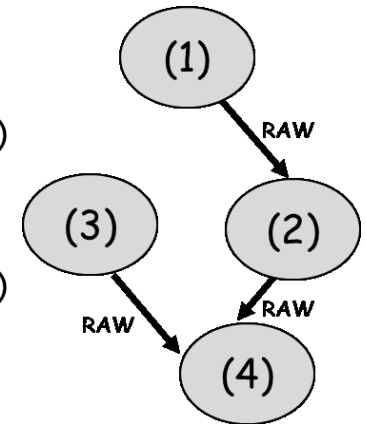
Kenji Kise, Department of Computer Science
kise_at_c.titech.ac.jp

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 instruction 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.

x3 = 10		x3 = 10
x5 = 2		x3 = 10
x3 = x3 x x5	(1)	x3 = 20
x4 = x3 + 1	(2)	x3 = 20
i x3 = x5 + 3	(3)	x3 = 5
j x7 = x3 + R4	(4)	x3 = 5

wrong sequence reordering (3) and (4)

x3 = 10		x3 = 10
x5 = 2		x3 = 10
x3 = x3 x x5	(1)	x3 = 20
x4 = x3 + 1	(2)	x3 = 20
j x7 = x3 + x4	(4)	x3 = 20
i x3 = x5 + 3	(3)	x3 = 5

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

	20 = 10 x 2	(1)
	21 = 20 + 1	(2)
j 41 = 20 + 21	(4)	
i 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.

wrong sequence reordering (1) and (3)

$x3 = 10$		$x3 = 10$
$x5 = 2$		$x3 = 10$
i $x3 = x3 \times x5$	(1)	$x3 = 20$
$x4 = x3 + 1$	(2)	$x3 = 20$
j $x3 = x5 + 3$	(3)	$x3 = 5$
$x7 = x3 + R4$	(4)	$x3 = 5$

$x3 = 10$		$x3 = 10$
$x5 = 2$		$x3 = 10$
j $x3 = x5 + 3$	(3)	$x3 = 5$
$x4 = x3 + 1$	(2)	$x3 = 5$
i $x3 = x3 \times x5$	(1)	$x3 = 20$
$x7 = x3 + x4$	(4)	$x3 = 20$

i $20 = 10 \times 2$	(1)
$21 = 20 + 1$	(2)
j $5 = 2 + 3$	(3)
$26 = 5 + 21$	(4)

j $5 = 2 + 3$	(3)
$6 = 5 + 1$	(2)
i $20 = 10 \times 2$	(1)
$41 = 20 + 21$	(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.

wrong sequence reordering (2) and (3)

$x3 = 10$		$x3 = 10$
$x5 = 2$		$x3 = 10$
$x3 = x3 \times x5$	(1)	$x3 = 20$
i $x4 = x3 + 1$	(2)	$x3 = 20$
j $x3 = x5 + 3$	(3)	$x3 = 5$
$x7 = x3 + x4$	(4)	$x3 = 5$

$x3 = 10$		$x3 = 10$
$x5 = 2$		$x3 = 10$
$x3 = x3 \times x5$	(1)	$x3 = 20$
j $x3 = x5 + 3$	(3)	$x3 = 5$
i $x4 = x3 + 1$	(2)	$x3 = 5$
$x7 = x3 + x4$	(4)	$x3 = 5$

	$20 = 10 \times 2$	(1)
i	$21 = 20 + 1$	(2)
j	$5 = 2 + 3$	(3)
	$26 = 5 + 21$	(4)

	$20 = 10 \times 2$	(1)
j	$5 = 2 + 3$	(3)
i	$6 = 5 + 1$	(2)
	$11 = 5 + 6$	(4)

Data dependence and **register renaming**

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

$$x3 = x3 \times x5 \quad (1)$$

$$x4 = x3 + 1 \quad (2)$$

$$\textcolor{red}{x8} = x5 + 3 \quad (3)$$

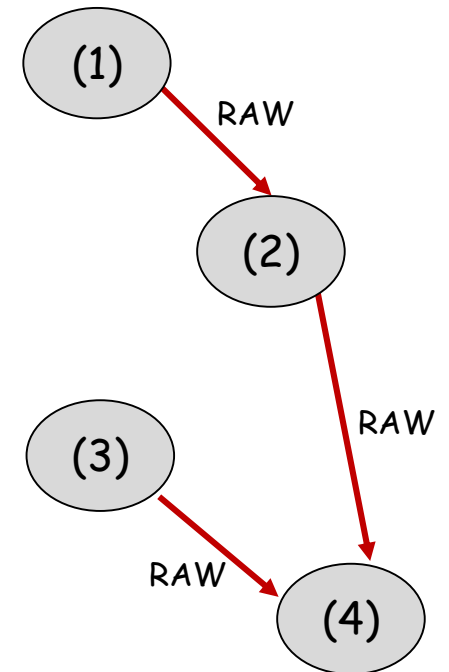
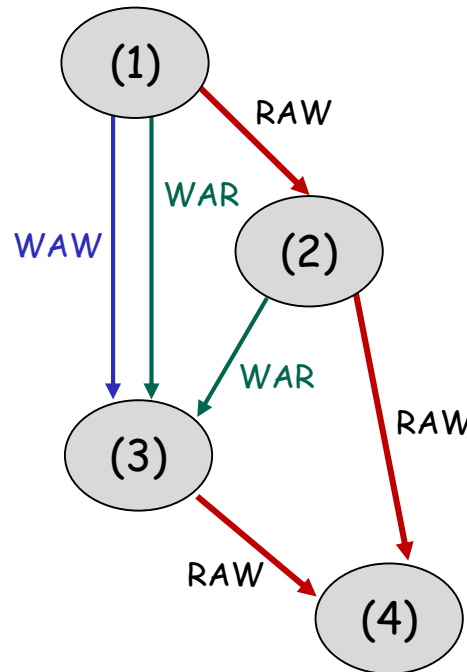
$$x7 = \textcolor{red}{x8} + x4 \quad (4)$$

$$x3 = x3 \times x5 \quad (1)$$

$$x4 = x3 + 1 \quad (2)$$

$$x3 = x5 + 3 \quad (3)$$

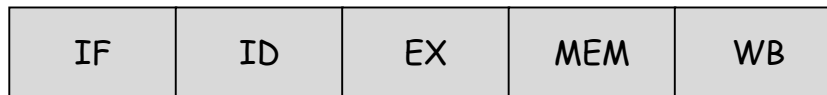
$$x7 = x3 + x4 \quad (4)$$



Hardware register renaming

- Logical registers (architectural registers) which are ones defined by ISA
 - x_0, x_1, \dots, x_{31}
- Physical registers
 - Assuming plenty of registers are available, p_0, p_1, p_2, \dots
- 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 **by hand**
- Rename the following instruction stream using physical registers of **p9**, **p10**, **p11**, and **p12** for I0, I1, I2, and I3, respectively
 - assuming that x1, x2, and x4 are renamed to p1, p2, and p4 respectively in advance

I0: sub x5,x1,x2

I1: add x9,x5,x4

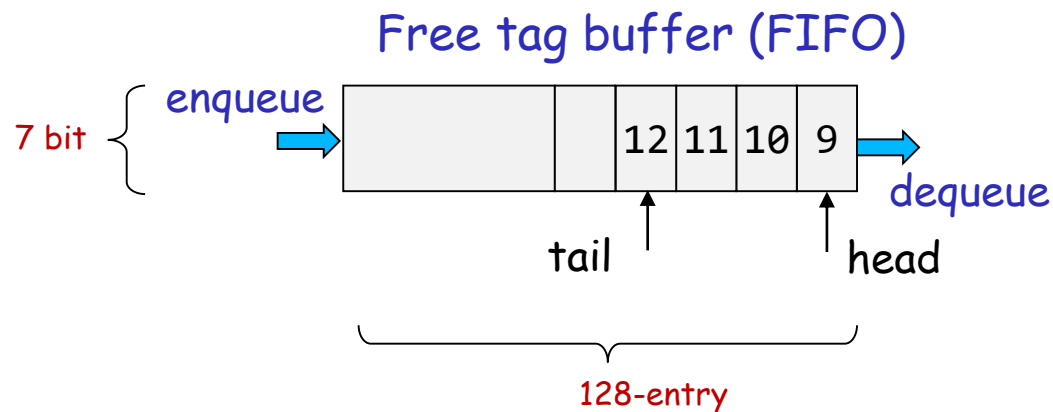
I2: or x5,x5,x2

I3: and x2,x9,x1

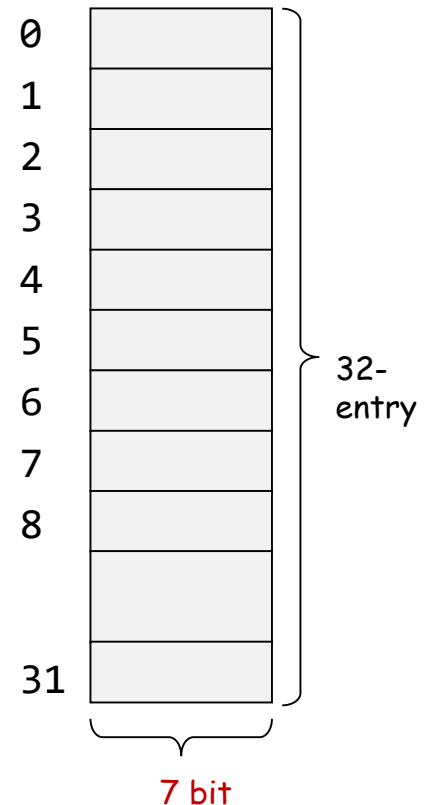


The main hardware for register renaming

- 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



Register map table



Example behavior of register renaming (1/4)

- Renaming the first instruction I0

Cycle 1

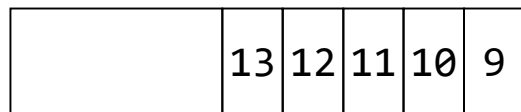
I0: sub x5,x1,x2

I1: add x9,x5,x4

I2: or x5,x5,x2

I3: and x2,x9,x1

Free tag buffer



↑ head

dst = x5

src1 = x1

src2 = x2

Register map table

0	0
1	1
2	2
3	3
4	4
5	5 -> 9
6	6
7	7
8	8
9	
10	
31	

dst = p9

src1 = p1

src2 = p2

I0: sub p9,p1,p2



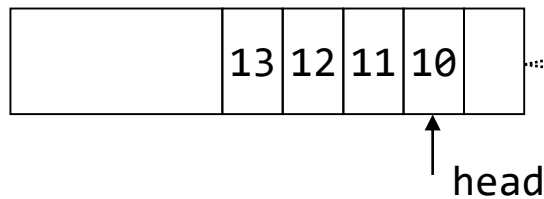
Example behavior of register renaming (2/4)

- Renaming the second instruction I1

Cycle 2

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

Free tag buffer



dst = x9
src1 = x5
src2 = x4

Register map table

0	0
1	1
2	2
3	3
4	4
5	9
6	6
7	7
8	8
9	->10
10	
31	

dst = p10
src1 = p9
src2 = p4

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



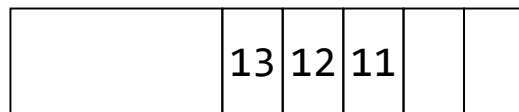
Example behavior of register renaming (3/4)

- Renaming instruction I2

Cycle 3

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

Free tag buffer



head

dst = x5
src1 = x5
src2 = x2

Register map table

0	0
1	1
2	2
3	3
4	4
5	9->11
6	6
7	7
8	8
9	10
10	
31	

dst = p11
src1 = p9
src2 = p2

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

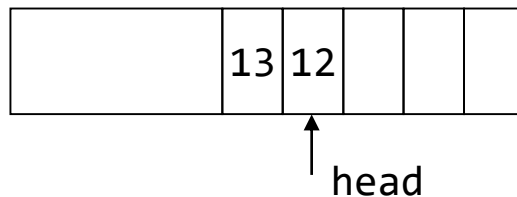
Example behavior of register renaming (4/4)

- Renaming instruction I3

Cycle 4

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

Free tag buffer



dst = x2
src1 = x9
src2 = x1

Register map table

0	0
1	1
2	2->12
3	3
4	4
5	11
6	6
7	7
8	8
9	10
10	
31	

dst = p12
src1 = p10
src2 = p1

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

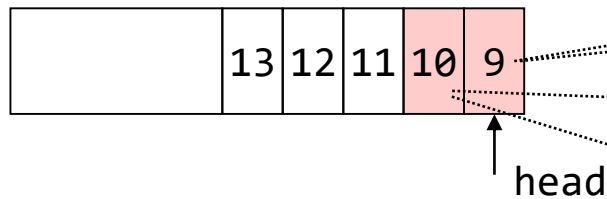
I0: sub x5,x1,x2

I1: add x9,x5,x4

I2: or x5,x5,x2

I3: and x2,x9,x1

Free tag buffer



dst = x5
src1 = x1
src2 = x2

dst = x9
src1 = x5
src2 = x4

Register map table

0	0	
1	1	
2	2	
3	3	
4	4	
5	5->9	dst = p9 src1 = p1 src2 = p2
6	6	
7	7	
8	8	
9	->10	dst = p10 src1 = p5 src2 = p4
10		
31		

I0: sub p9,p1,p2

I1: add p10,p5,p4 (Wrong)



Exercise 2

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

Cycle 1

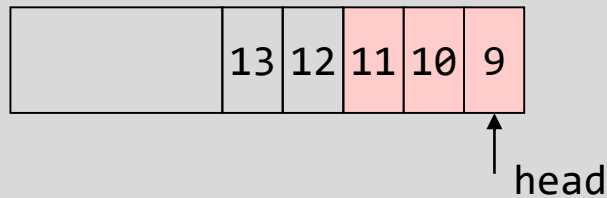
I0: sub x5,x1,x2

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 behavior of cycle 1
renaming three instructions.



Pollack's Rule

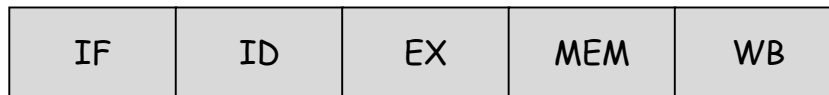
- Pollack's Rule states that microprocessor "performance increase due to microarchitecture advances **is roughly proportional to the square root of the increase in complexity**". Complexity in this context means processor logic, i.e. its area.

WIKIPEDIA

Hardware register renaming

- Logical registers (architectural registers) which are ones defined by ISA
 - $x0, x1, \dots, x31$
- Physical registers
 - Assuming plenty of registers are available, $p0, p1, p2, \dots$
- 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 by hand
- Rename the following instruction stream using physical registers of p9, p10, p11, and p12 for I0, I1, I2, and I3, respectively
 - assuming that x1, x2, and x4 are renamed to p1, p2, and p4 respectively in advance

I0: sub x5,x1,x2

I1: add x9,x5,x4

I2: or x5,x5,x2

I3: and x2,x9,x1



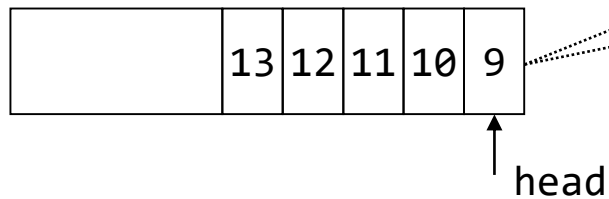
Example behavior of register renaming and valid bit

- Renaming the first instruction I0

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



dst = x5
src1 = x1
src2 = x2

Register map table

		valid bit
0		
1		0
2		0
3		
4		
5	5 -> 9	1
6		
7		
8		
9		
10		
31		

dst = p9
src1 = x1
src2 = x2

I0: sub p9,x1,x2

Recommended Reading



- [Clockhands: Rename-free Instruction Set Architecture for Out-of-order Processors](#)
 - Toru Koizumi (NITech), Ryota Shioya, Shu Sugita, Taichi Amano, Yuya Degawa, Junichiro Kadomoto, Hidetsugu Irie, Shuichi Sakai (U.Tokyo)
 - 56th IEEE/ACM International Symposium on Microarchitecture (MICRO'23)
- A quote:

"Out-of-order superscalar processors are currently the only architecture that speeds up irregular programs, but they suffer from poor power efficiency. To tackle this issue, we focused on how to specify register operands. Specifying operands by register names, as conventional RISC does, requires register renaming, resulting in poor power efficiency and preventing an increase in the front-end width. In contrast, a recently proposed architecture called STRAIGHT specifies operands by inter-instruction distance, thereby eliminating register renaming. However, STRAIGHT has strong constraints on instruction placement, which generally results in a large increase in the number of instructions.
- We propose Clockhands, a novel instruction set architecture that has multiple register groups and specifies a value as "the value written in this register group k times before." Clockhands does not require register renaming as in STRAIGHT. In contrast, Clockhands has much looser constraints on instruction placement than STRAIGHT, allowing programs to be written with almost the same number of instructions as Conventional RISC. We implemented a cycle-accurate simulator, FPGA implementation, and first-step compiler for Clockhands and evaluated benchmarks including SPEC CPU. On a machine with an eight-fetch width, the evaluation results showed that Clockhands consumes 7.4% less energy than RISC while having performance comparable to RISC. This energy reduction increases significantly to 24.4% when simulating a futuristic up-scaled processor with a 16-fetch width, which shows that Clockhands enables a wider front-end."



Recommended Reading

(a) A simple code

```
void iota( int arr[], int N ) {  
    int i;  
    for( i = 0; i < N; ++i) {  
        arr[i] = i;  
    }  
}
```

(c) STRAIGHT assembly

```
iota:  
    ble    [3], zero, .L1  
    spaddi -8  
    addi   zero, 0    # i  
    sd     [4], 0(sp) # _RetAddr  
    mv     [6]      # &arr[i]  
    mv     [8]      # N  
    j      .L3  
.L2:  
    addi   [6], 4     # &arr[i]  
    mv     [6]      # N relay  
    nop    # dist. adjust  
.L3:  
    sw     [5], 0([3])  
    addiw  [6], 1     # ++i  
    bne    [1], [4], .L2  
    ld     0(sp)  
    spaddi 8  
.L1:  
    ret    [2]
```

(b) A RISC (RISC-V) assembly

```
iota:  
    ble    a1, zero, .L1  
    addi   a5, zero, 0    # i  
.L3:  
    sw     a5, 0(a0)  
    addiw  a5, a5, 1     # ++i  
    addi   a0, a0, 4     # &arr[i]  
    bne    a1, a5, .L3  
.L1:  
    ret    ra
```

(d) Clockhands assembly

```
iota:  
    ble    s[2], zero, .L1  
    addi   t, zero, 0    # i  
    mv     t, s[1]      # &arr[i]  
.L3:  
    sw     t[1], 0(t[0])  
    addiw  t, t[1], 1     # ++i  
    addi   t, t[1], 4     # &arr[i]  
    bne    t[1], s[2], .L3  
.L1:  
    ret    s[0]
```

Figure 1: (a) Simple code written in C. (b) Assembly code compiled for RISC-V, a conventional RISC architecture. `a0`, `a1`, and `a5` are logical register names. (c) Assembly code compiled for STRAIGHT, an existing rename-free architecture. The shaded parts indicate instructions that have been added compared with the RISC-V code. In STRAIGHT, the destination register of an instruction is not specified and is implicitly assigned from a ring buffer. A source operand of an instruction is specified by an *inter-instruction* distance, such as `[1]`, `[3]`, and `[6]` (e.g., `[3]` represents a reference to the result of three previous instructions.). (d) Assembly code compiled for Clockhands, our proposed architecture. `t` and `s` represent the names of *hands* (i.e., register groups). In Clockhands, the destination register of an instruction is specified by a hand identifier. A source register of an instruction is specified by combining a hand identifier and an *inter-register* distance, denoted as `[2]` (e.g., `t[2]` represents a reference to the result of three previous registers in the hand `t`).

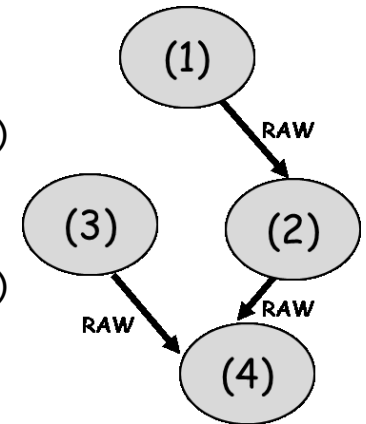


Exploiting Instruction Level parallelism (ILP)

- 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

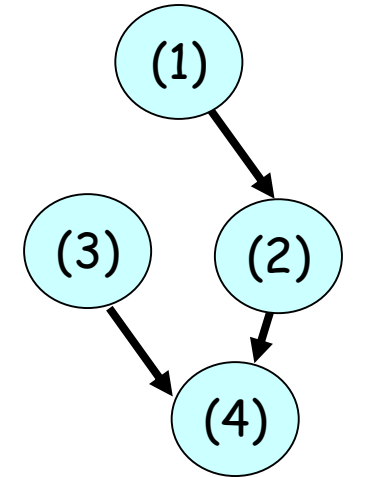
(1) add x5, x1, x2
(2) add x9, x5, x3
(3) lw x4, 4(x7)
(4) add x8, x9, x4

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



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)



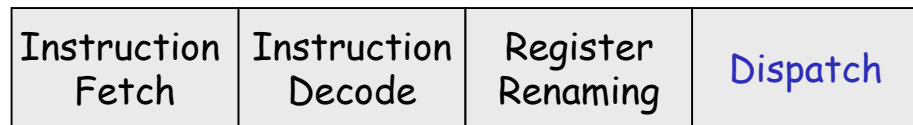
(1) add **x5**, x1, x2
(2) add **x9**, **x5**, x3
(3) lw **x4**, 4(x7)
(4) add x8, **x9**, x4

Data flow graph



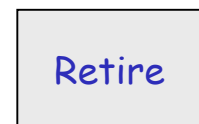
Instruction pipeline of OoO execution processor

- 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) using a logical register number.



In-order front-end

Out-of-order back-end



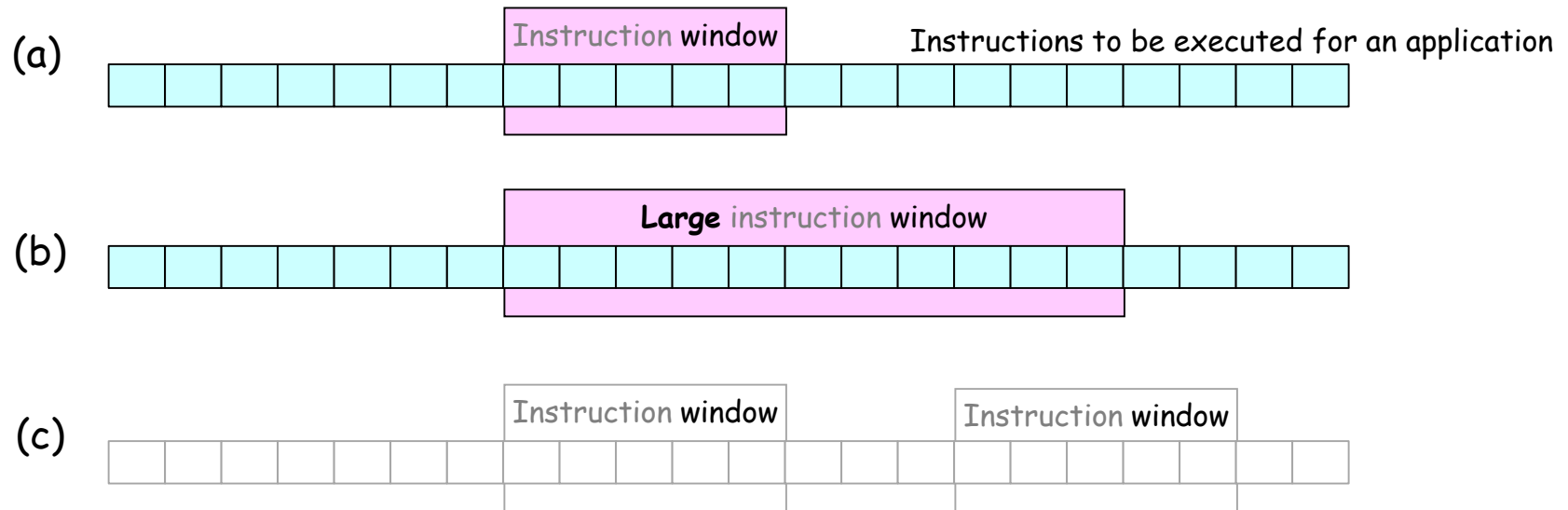
In-order retirement

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)

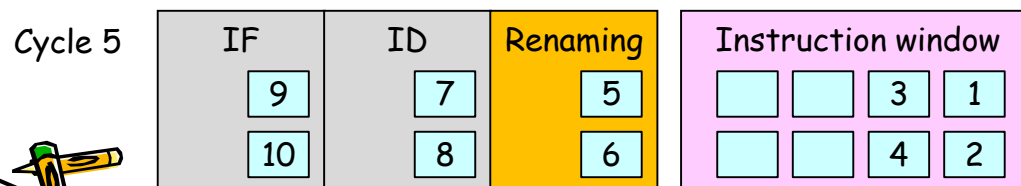
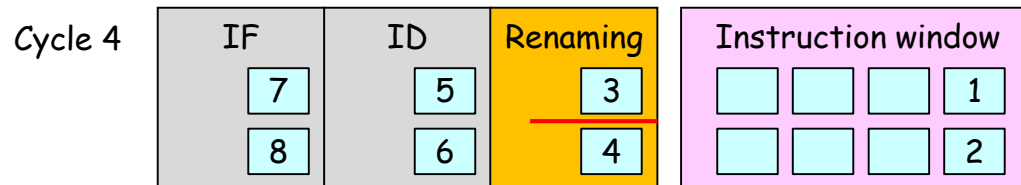
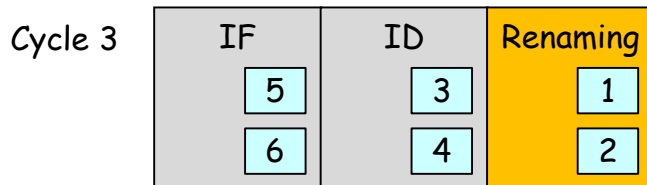
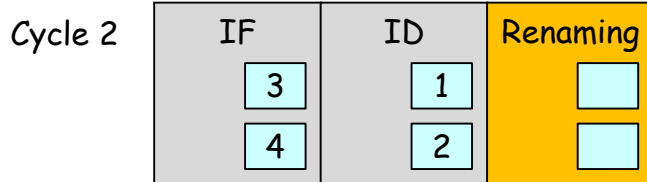
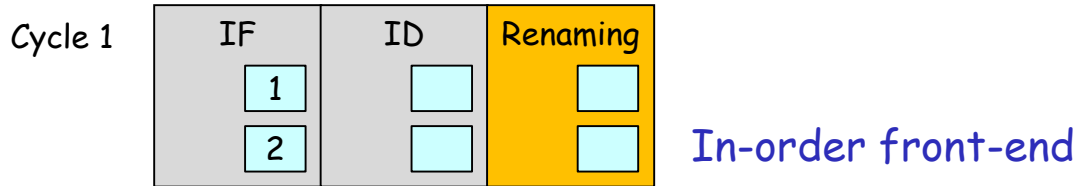


Instruction window			
	8	6	5
		4	7

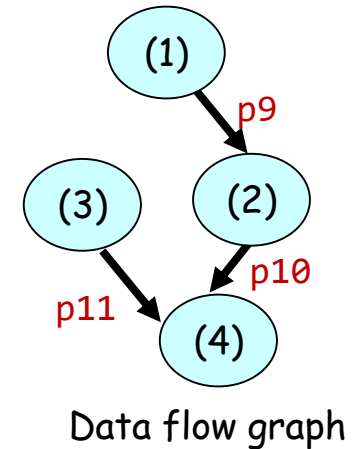


The key idea for OoO execution (1/3)

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



I1: sub p9, p1, p2
 I2: add p10, p9, p3
 I3: or p11, p4, p5
 I4: and p12, p10, p11



assume that instructions cannot exit the instruction window until cycle 5

The key idea for OoO execution (2/3)

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

Cycle 5

IF	ID	Renaming	Instruction window			
9	7	5			3	1
10	8	6			4	2

I1: sub **p9**, p1, p2
 I2: add **p10**, **p9**, p3
 I3: or **p11**, p4, p5
 I4: and **p12**, **p10**, **p11**

Cycle 6

IF	ID	Renaming	Instruction window				Issue
11	9	7			6	5	1
12	10	8			4	2	3

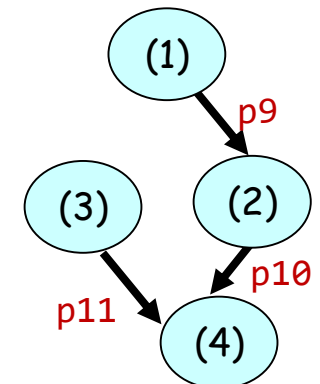
We assume that I1 and I3 can be issued at cycle 6 by dependence.

Cycle 7

IF	ID	Renaming	Instruction window				Issue	Execute
13	11	9		8	6	5	2	➤ 1
14	12	10			4	7		➤ 3

Cycle 8

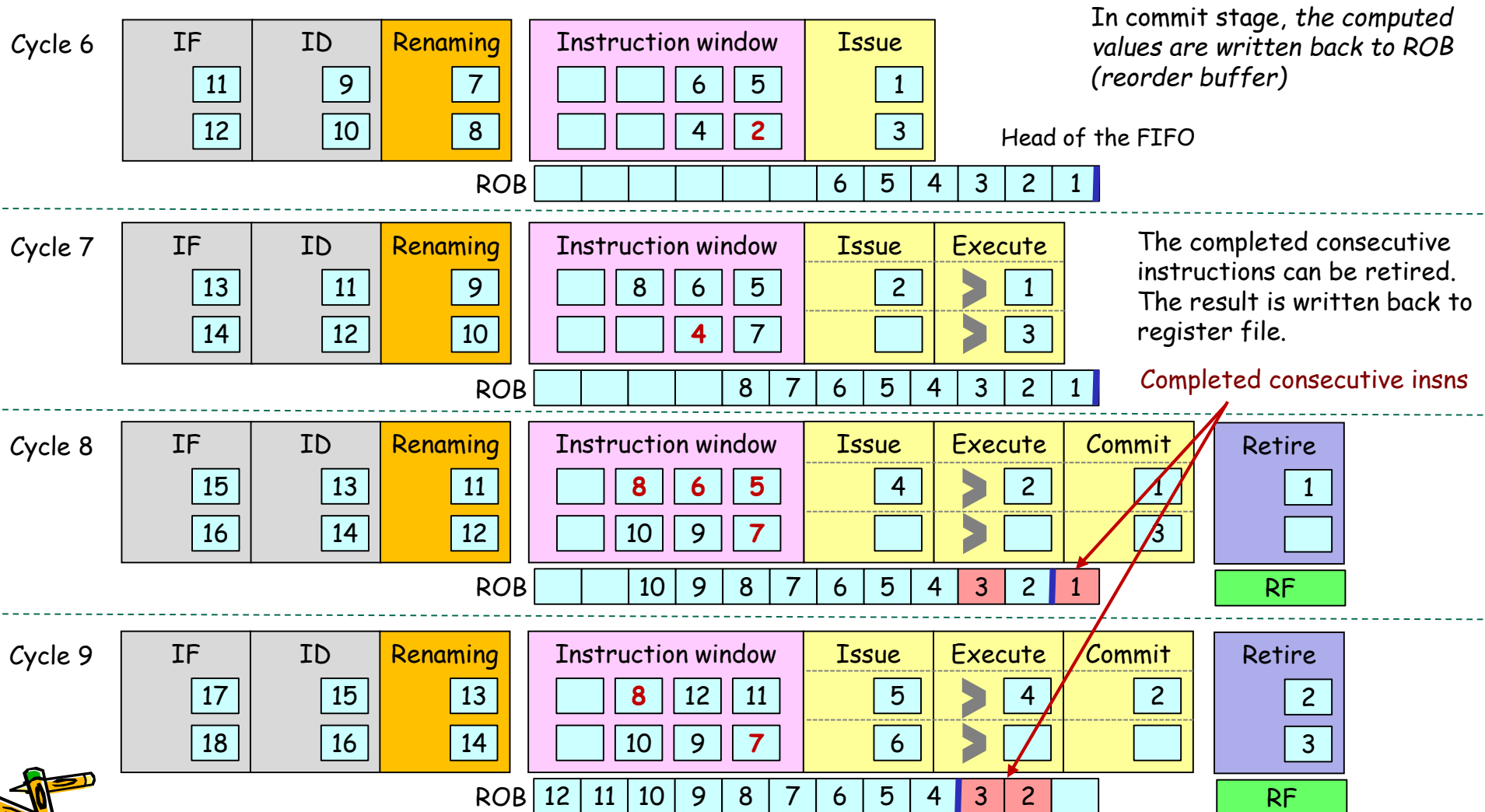
IF	ID	Renaming	Instruction window				Issue	Execute	Commit
15	13	11		8	6	5	4	➤ 2	1
16	14	12		10	9	7		➤	3



Data flow graph

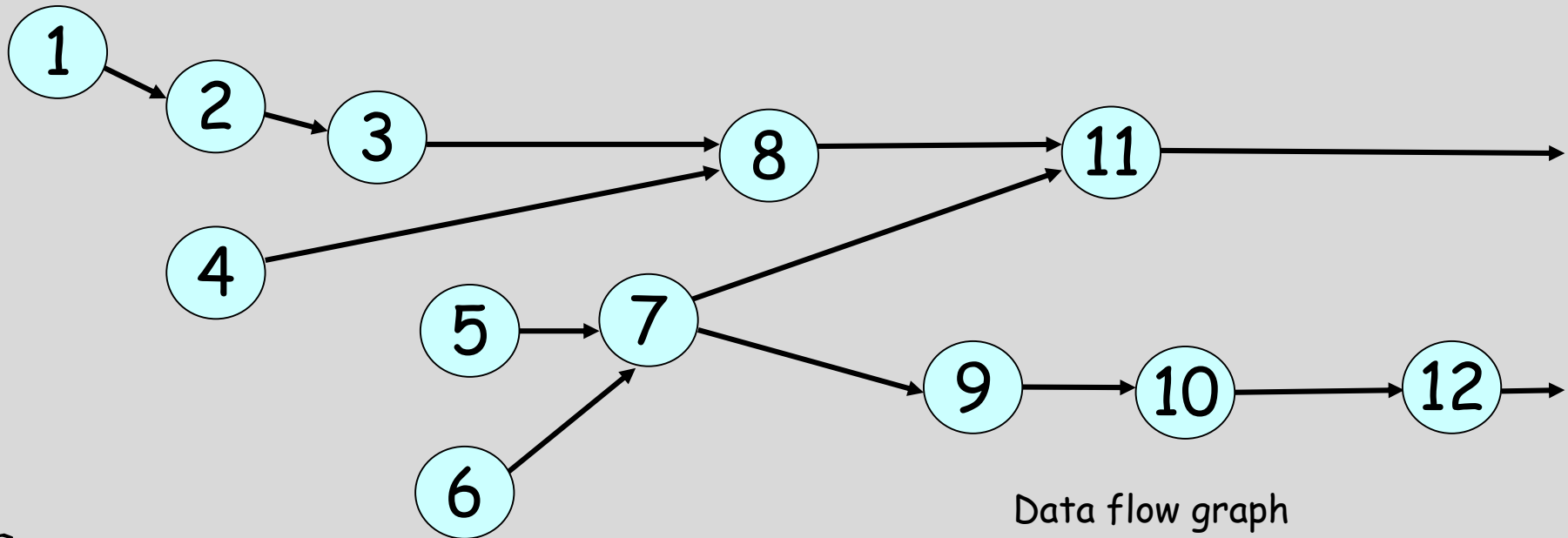
The key idea for OoO execution (3/3)

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



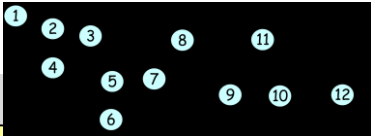
Exercise 3

- OoO execution
- Fill out the cycle by cycle processing behavior of these 12 instructions
 - wakeup
 - select

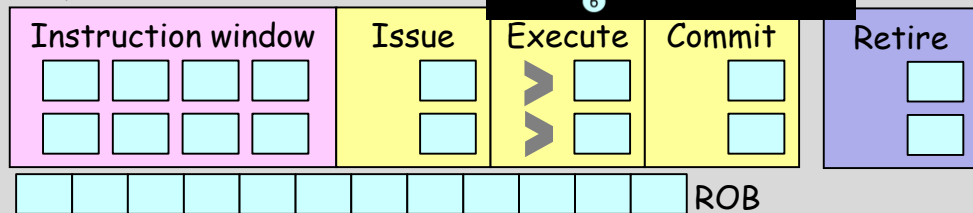


Data flow graph

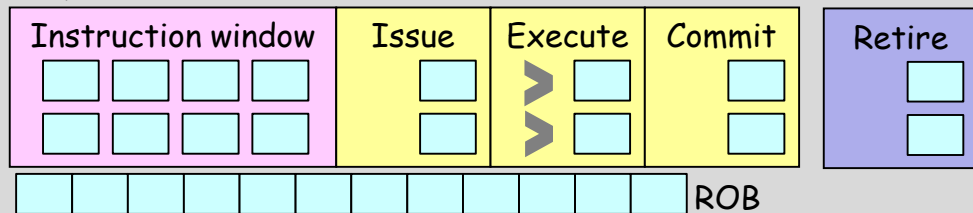




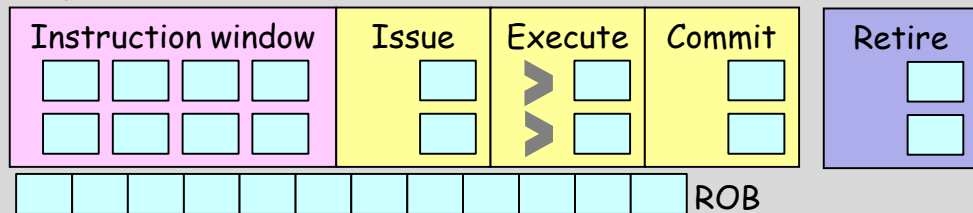
Cycle 1



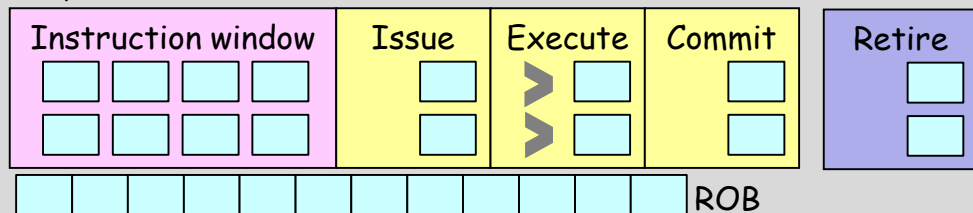
Cycle 2



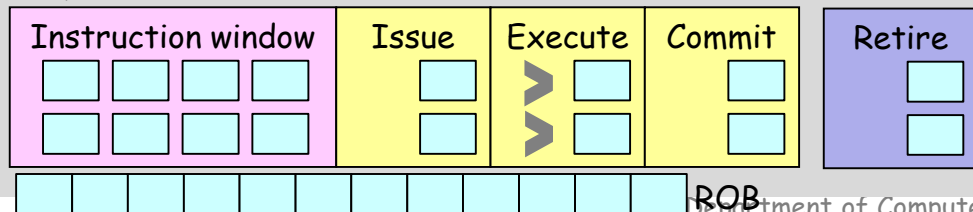
Cycle 3



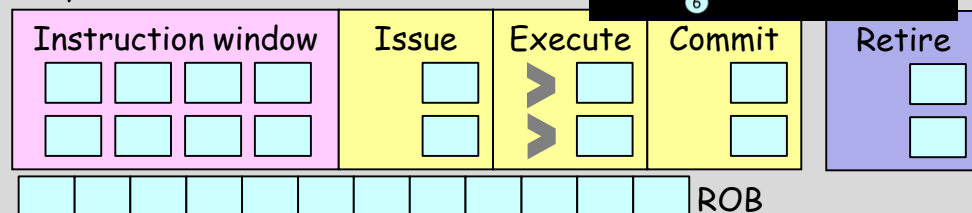
Cycle 4



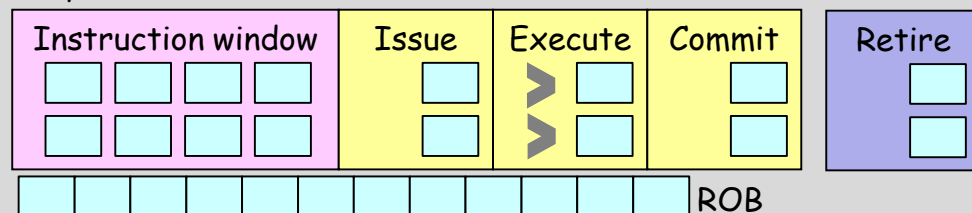
Cycle 5



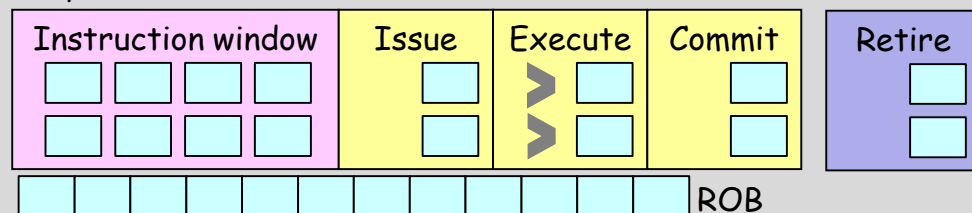
Cycle 6



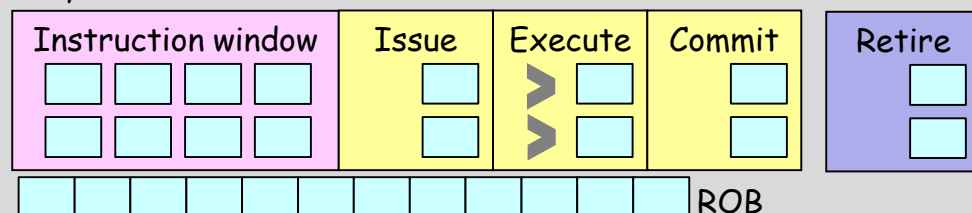
Cycle 7



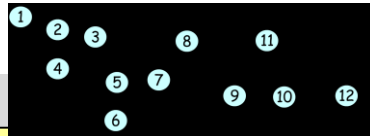
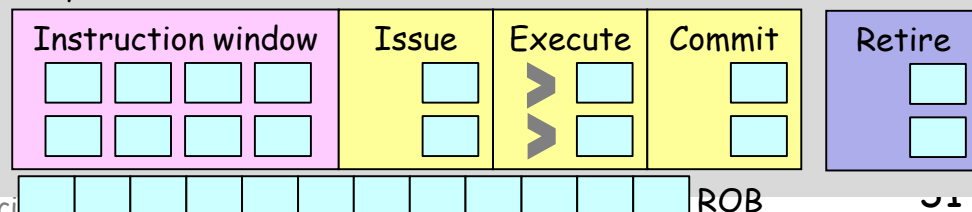
Cycle 8



Cycle 9

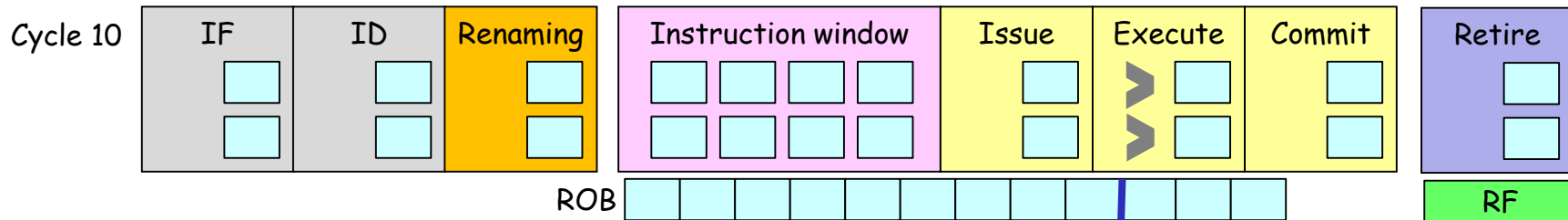
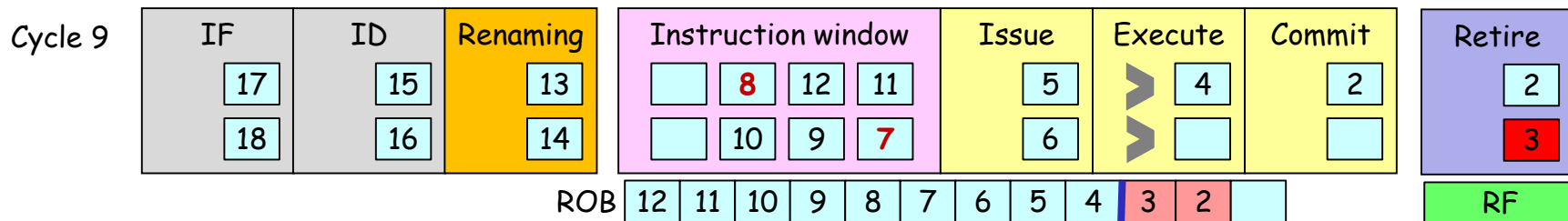


Cycle 10

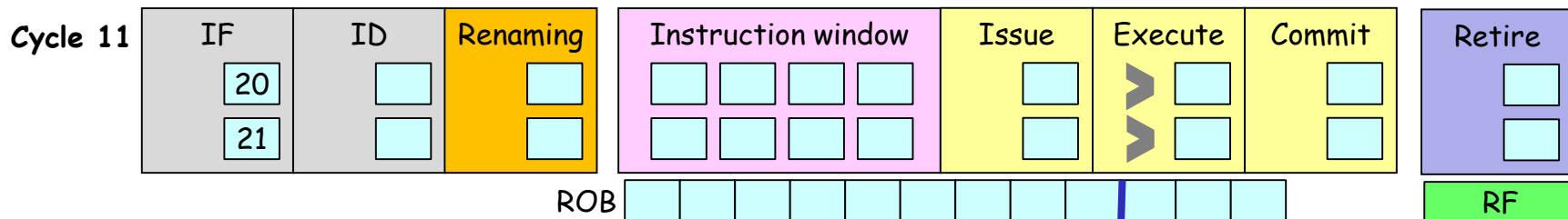


Prediction miss and recovery

- Assume that instruction 3 is a **miss predicted branch** and its target insn is 20
- When insn 3 is **retired**, it recovers by **flushing** all instructions and restart
- Register file (and PC) has the **architecture state** after insn 3 is executed



Recovery by flushing instructions on the wrong path (may take several cycles)

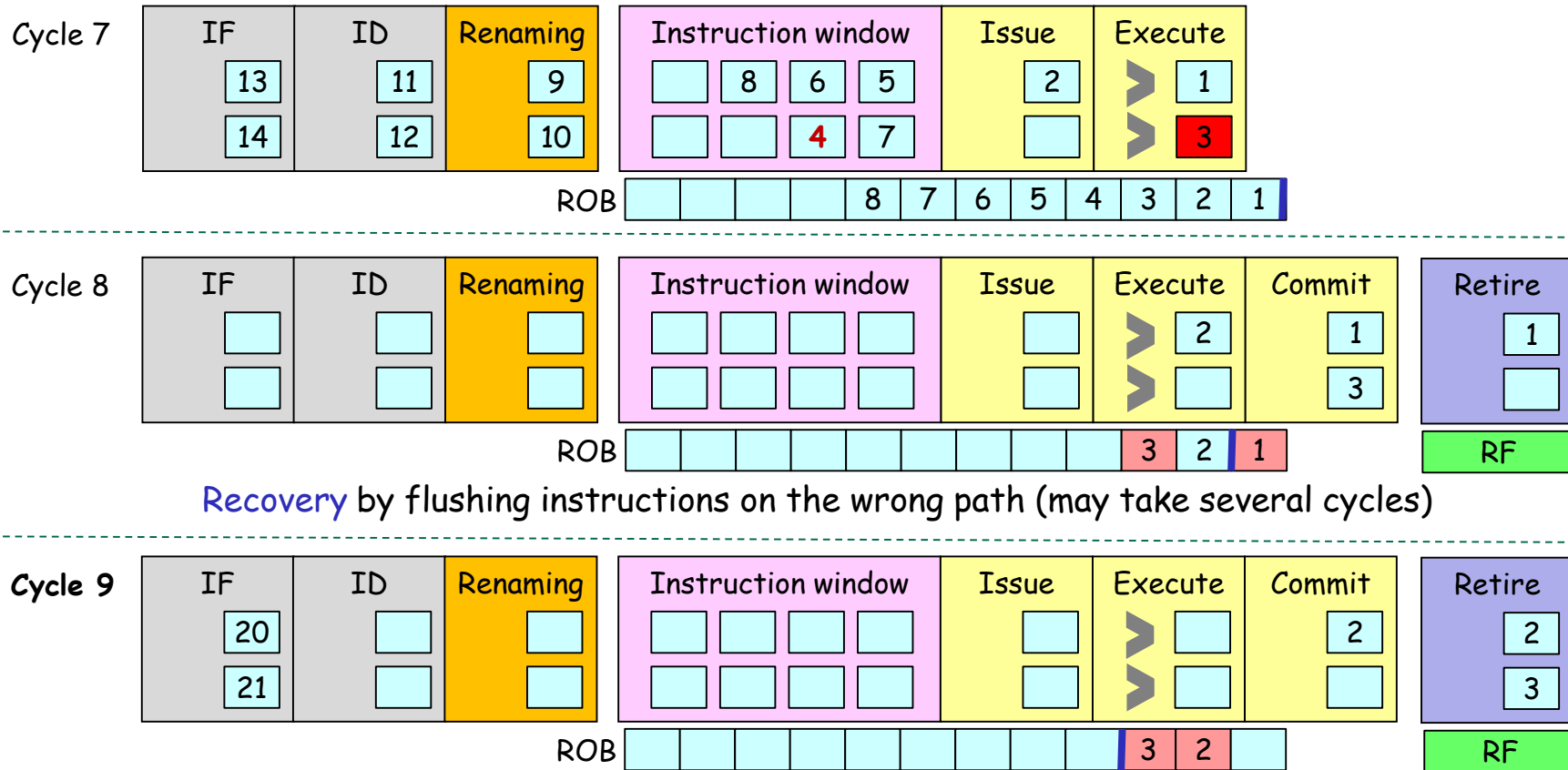


Restart by fetching instructions using the correct PC



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



Restart by fetching instructions using the correct PC

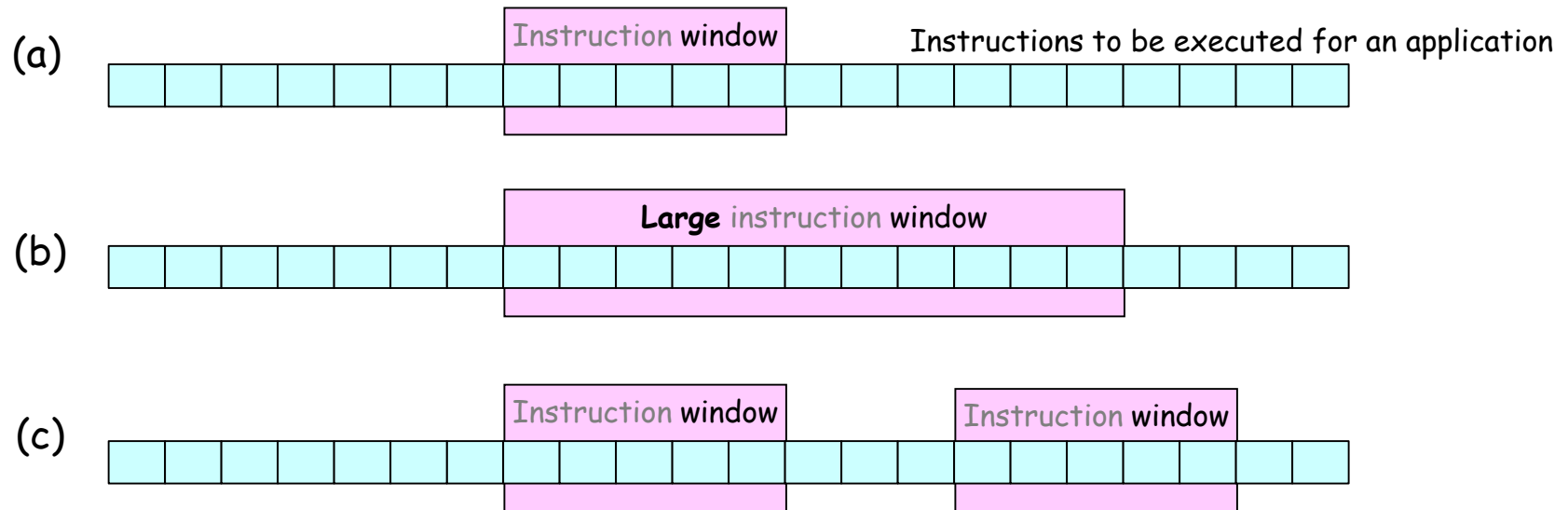


Aside: What is a window?

- 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)

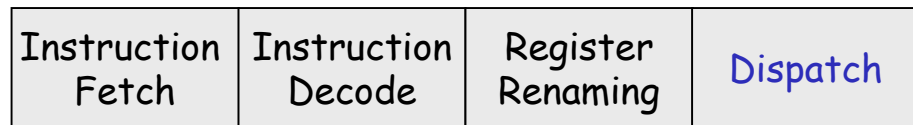


Instruction window			
	8	6	5
		4	7



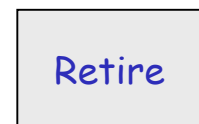
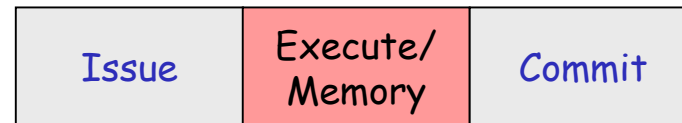
Instruction pipeline of OoO execution processor

- 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) using a logical register number.



In-order front-end

Out-of-order back-end



In-order retirement

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

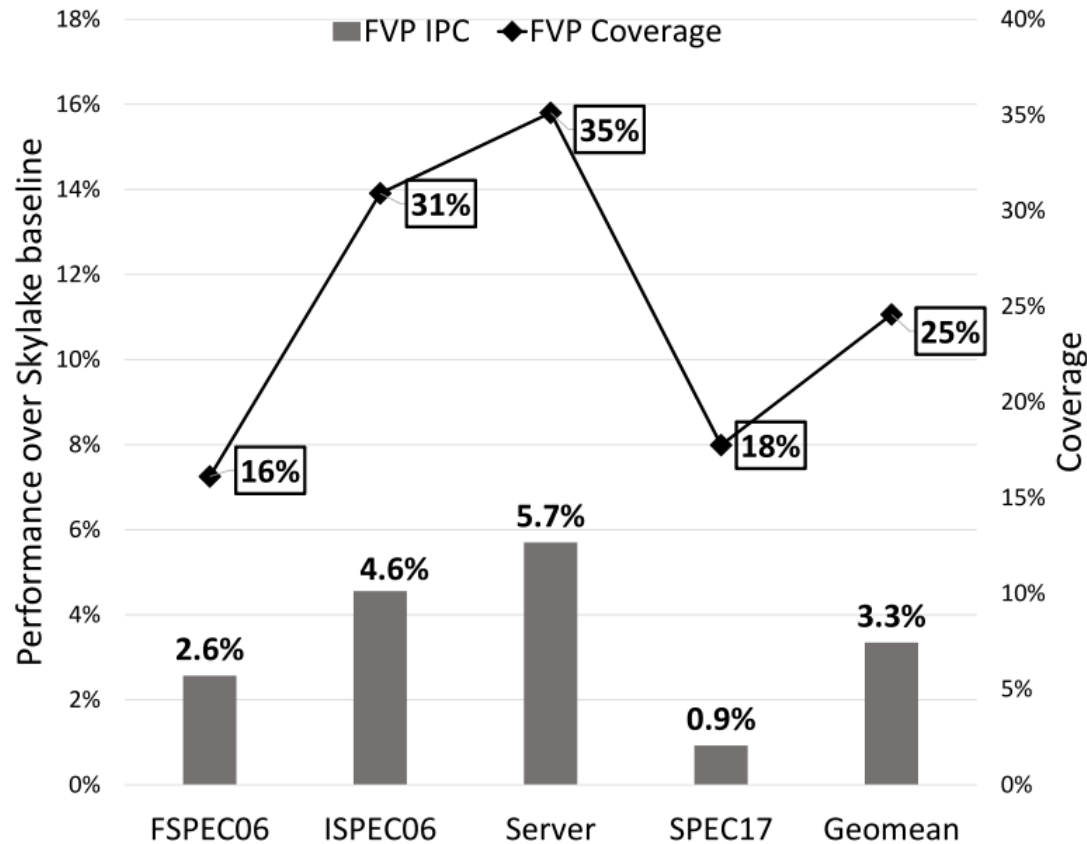


Fig. 6. Performance and Coverage of FVP on Skylake

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, gemsfddt, 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
lammmps [4], hplinpac [3], tpce, spark, cassandra [1], specjbb [5], specjenterprise, hadoop [2], specpower [6]	Server

TABLE III
APPLICATIONS USED IN THIS STUDY