

Embedded Intelligent System and Novel Computer Architecture

Lecture 03(a) – Understanding Modern Processor: ILP and Optimization Code

Pengju Ren

**Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University**

<http://gr.xjtu.edu.cn/web/pengjuren>

Outline

- Instruction level parallel
- Pipeline
 - Data hazards
 - Control hazards
 - Structure hazards
- Out-of-Order Execution
 - Dataflow
- Optimization based on ILP
- Case study
 - Throughput bound
 - Latency bound
 - Performance Optimization

Many kinds of processors



CPU



GPU



FPGA



ASIC Etc.

Why so many? What differentiates these processors?

Why so many kinds of processors?

Each processor is designed for different kinds of programs

- CPUs
 - “Sequential” code – i.e., single / few threads
- GPUs
 - Programs with lots of independent work → “Embarrassingly parallel”
- Many others: Deep neural networks, Digital signal processing, Etc.

Parallelism pervades architecture

- **Speeding up programs is all about parallelism**
 - Find independent work
 - Execute it in parallel
 - Profit
- **Key questions:**
 - Where is the parallelism?
 - Whose job is it to find parallelism?

Where is the parallelism?

Different processors take radically different approaches

- CPUs: Instruction-level parallelism (ILP)
 - Implicit
 - Fine-grain
- GPUs: Thread- & data-level parallelism (TLP, DLP)
 - Explicit
 - Coarse-grain

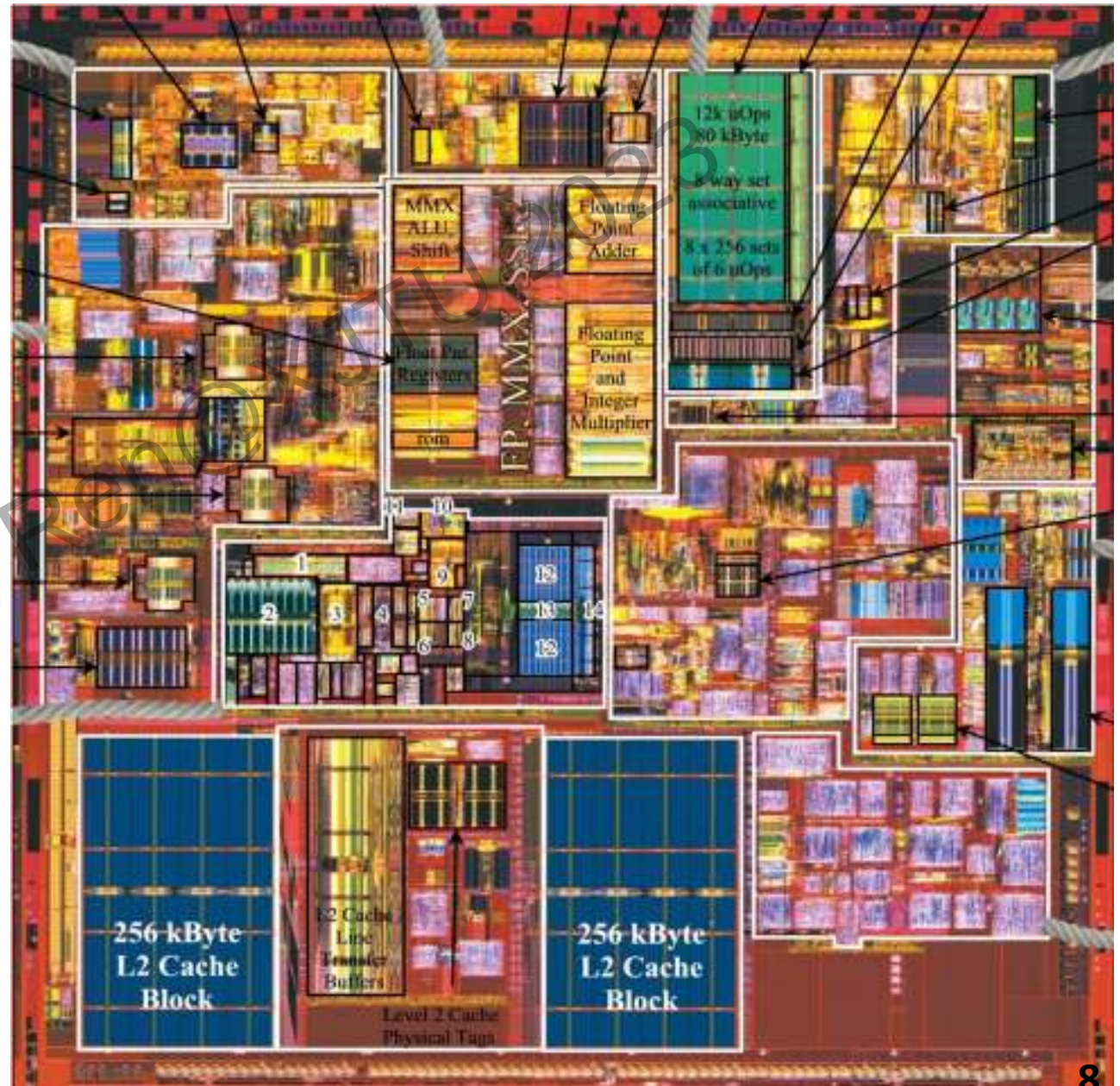
Whose job to find parallelism?

Different processors take radically different approaches

- CPUs: Hardware dynamically schedules instructions
 - Expensive, complex hardware → Few cores (tens)
 - (Relatively) *Easy to write fast software*
- GPUs: Software makes parallelism explicit
 - Simple, cheap hardware → Many cores (thousands)
 - (Often) *Hard to write fast software*

Visualizing these differences

- Pentium 4 “Northwood” (2002)



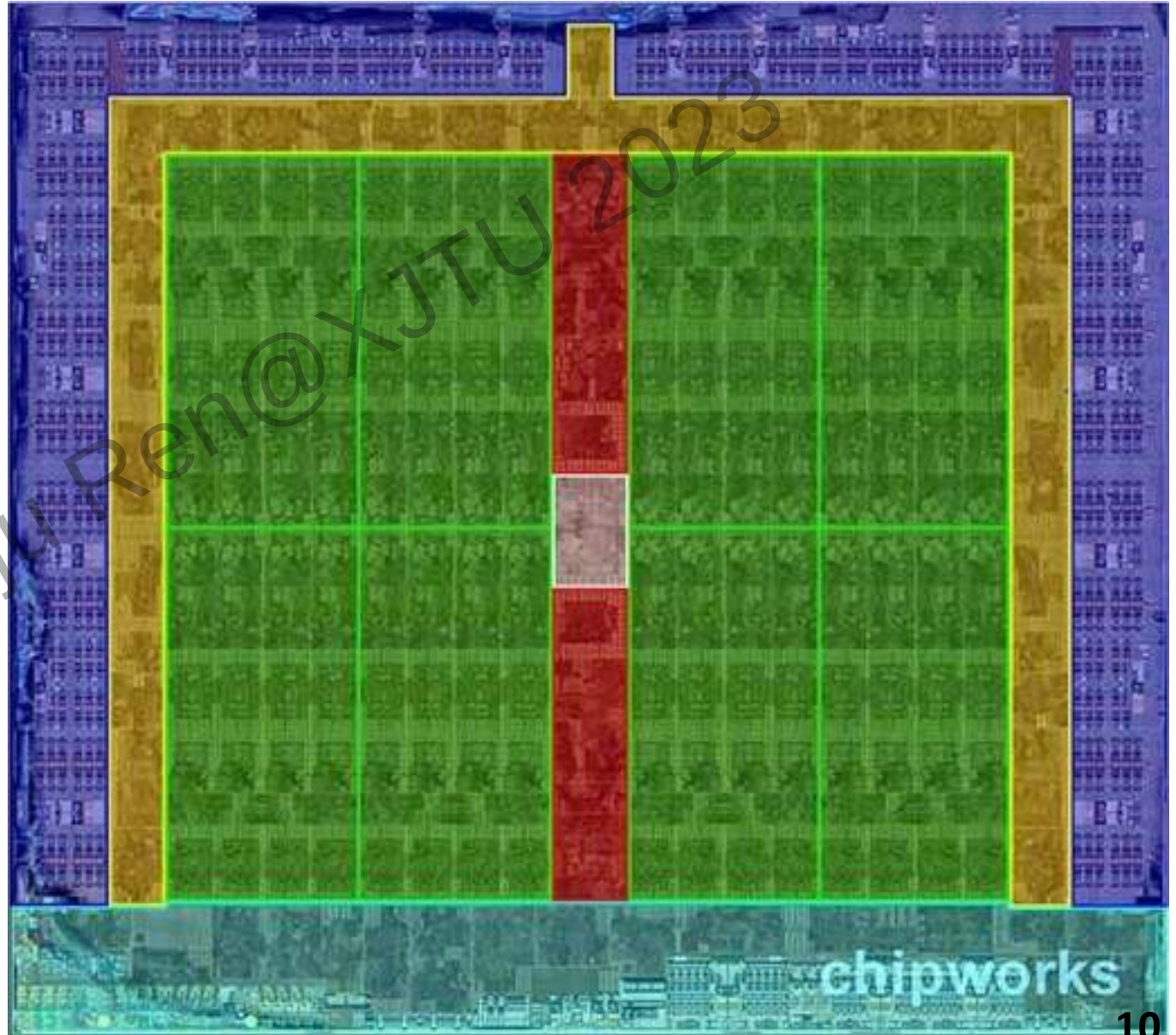
Visualizing these differences

- Pentium 4 “Northwood” (2002)
- Highlighted areas actually execute instructions
 - ➔ Most area spent on Caches and Scheduling (not on executing the program)



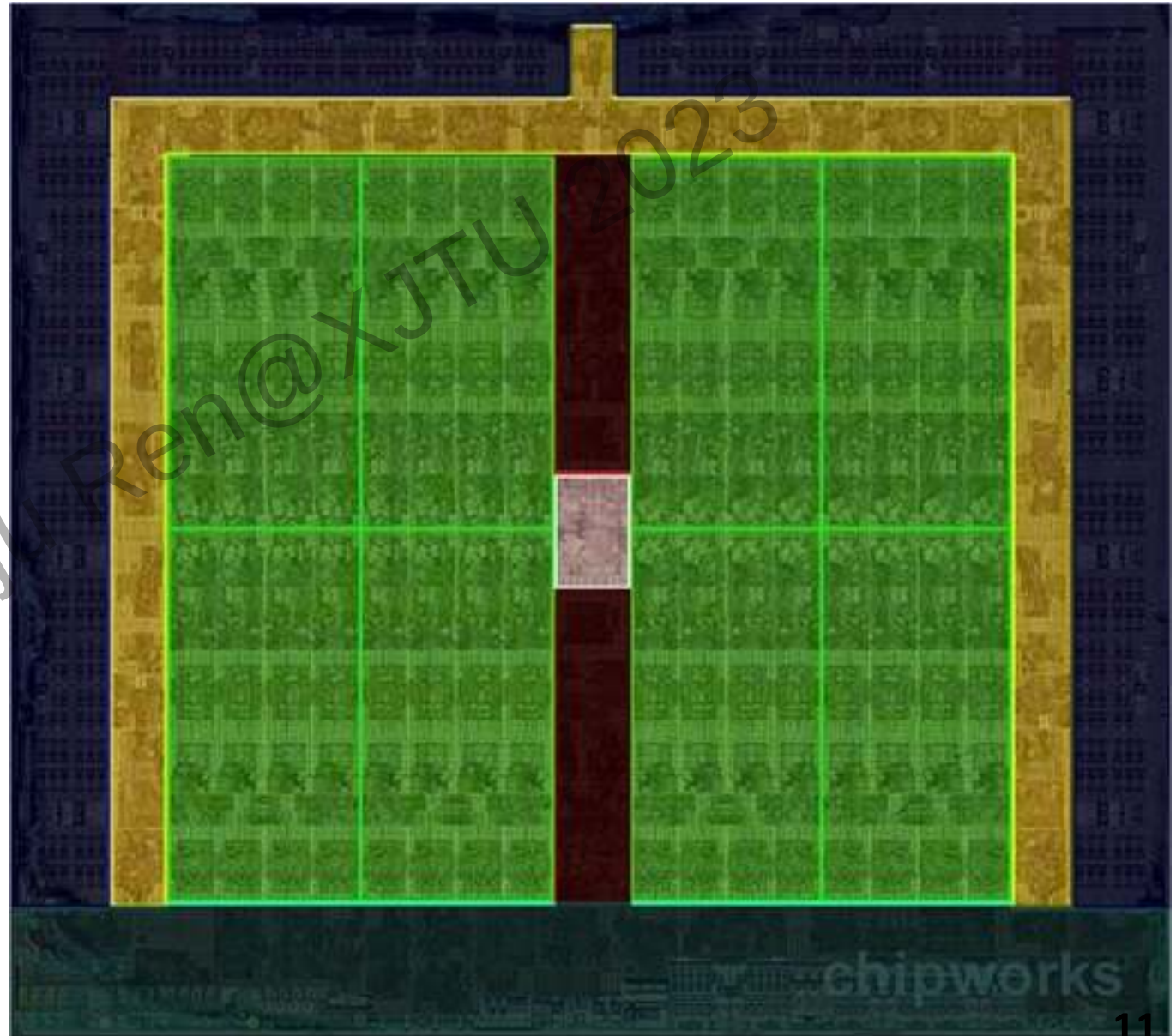
Visualizing these differences

- AMD Fiji
(GPU@2015)



Visualizing these differences

- AMD Fiji (GPU@2015)
- Highlighted areas actually execute instructions
 - ➔ Most area spent executing the program
 - (Rest is mostly I/O & memory, not scheduling)



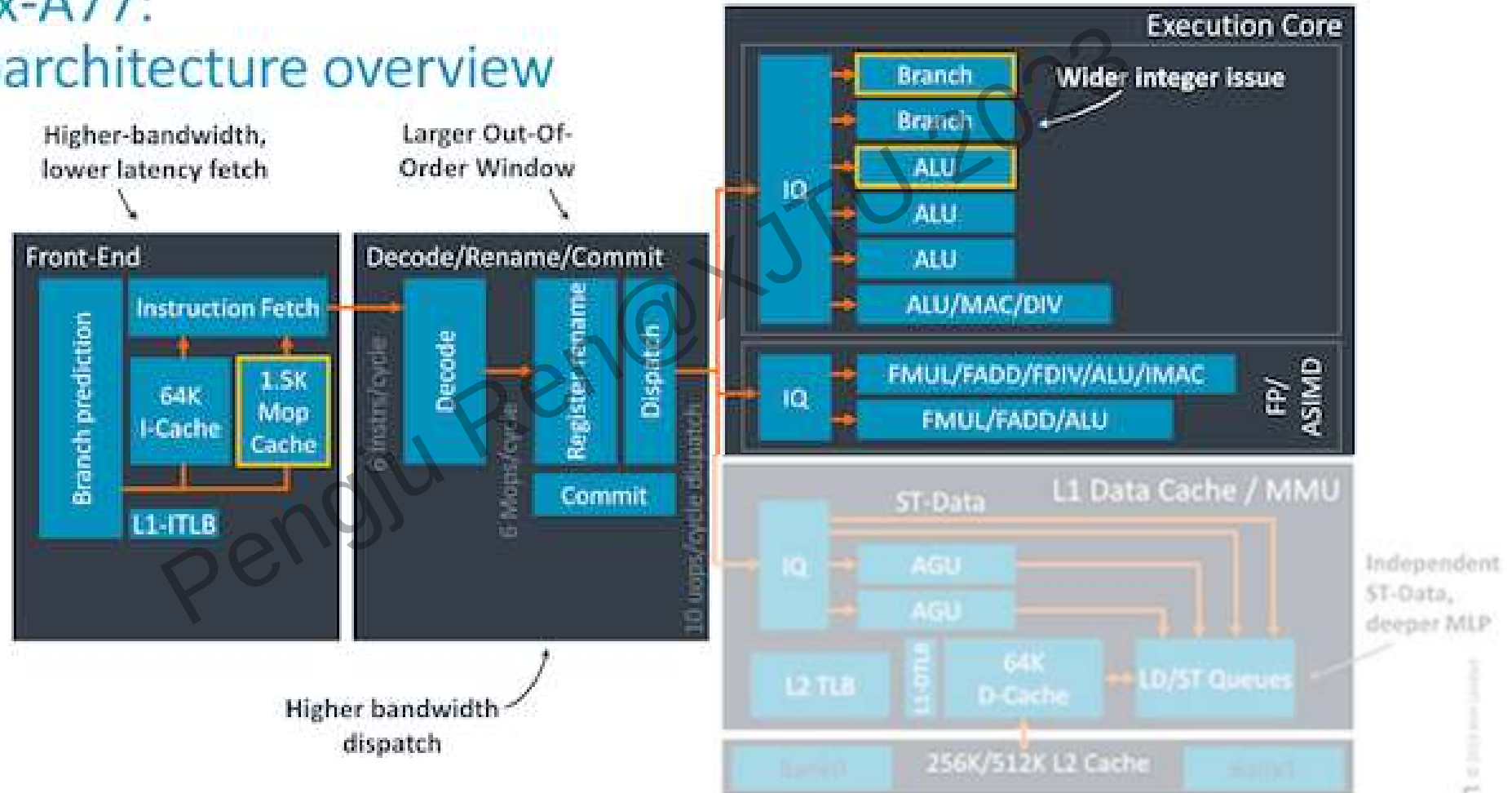
Today you will review (or learn) ...

How CPUs exploit ILP to speed up sequential code

- Key ideas:
 - Pipelining & Superscalar: Work on multiple instructions at once
 - Out-of-order execution: Dynamically schedule instructions whenever they are “ready”
 - Speculation: Guess what the program will do next to discover more independent work, “rolling back” incorrect guesses
- CPUs must do all of this while preserving the illusion that instructions execute **in-order, one-at-a-time**

In other words... Today is about:

Cortex-A77: Microarchitecture overview



The embargo for this content presented at Arm Tech Day will lift on Sunday, May 26 at 9:00 p.m. PT. Corresponding UK and Taiwan times are: Monday, May 27 at 5:00 a.m. BST / Monday, May 27 at 12:00 p.m. China Standard Time

Example: Polynomial evaluation

$$value = \sum_{j=0}^{terms} coef[j]x^j$$

```
int poly(int *coef,  
        int terms, int x) {  
    int power = 1;  
    int value = 0;  
    for (int j = 0; j < terms; j++) {  
        value += coef[j] * power;  
        power *= x;  
    }  
    return value;  
}
```

Example: Polynomial evaluation

$$value = \sum_{j=0}^{terms} coef[j]x^j$$

r0: value
r1: &coef[terms]
r2: x
r3: &coef[0]
r4: power
r5: coef[j]

■ Compiling on ARM

```
int poly(int *coef,  
        int terms, int x) {  
    int power = 1;  
    int value = 0;  
    for (int j = 0; j < terms; j++) {  
        value += coef[j] * power;  
        power *= x;  
    }  
    return value;  
}
```

```
poly:  
    cmp     r1, #0  
    ble     .L4  
    push    {r4, r5}  
    mov     r3, r0  
    add     r1, r0, r1, lsl #2  
    movs    r4, #1  
    movs    r0, #0  
    .L3:  
    ldr     r5, [r3], #4  
    cmp     r1, r3  
    mla     r0, r4, r5, r0  
    mul     r4, r2, r4  
    bne     .L3  
    pop     {r4, r5}  
    bx      lr  
    .L4:  
    movs    r0, #0  
    bx      lr
```

Compilers Manage Memory and Registers

Compilers for languages like C/C++:

- **Check that program is legal**
- **Translate into assembly code**
- **Optimizes the generated code**

Compiler performs “register allocation” to decide when to load/store and when to reuse

Example: Polynomial evaluation

r0: value
r1: &coef[terms]
r2: x
r3: &coef[j]
r4: power
r5: coef[j]

■ Compiling on ARM

```
int poly(int *coef,  
        int terms, int x) {  
    int power = 1;  
    int value = 0;  
    for (int j = 0; j < terms; j++) {  
        value += coef[j] * power;  
        power *= x;  
    }  
    return value;  
}
```

poly: cmp r1, #0 ble .L4 push {r4, r5} mov r3, r0 add r1, r0, r1, lsl #2 movs r4, #1 movs r0, #0	Preamble
.L3: ldr r5, [r3], #4 cmp r1, r3 mla r0, r4, r5, r0 mul r4, r2, r4 bne .L3	Iteration
pop {r4, r5} bx lr .L4: movs r0, #0 bx lr	Finish

Example: Polynomial evaluation

r0: value
r1: &coef[terms]
r2: x
r3: &coef[j]
r4: power
r5: coef[j]

■ Compiling on ARM

Iteration

```
for (int j = 0; j < terms; j++) {  
    value += coef[j] * power;  
    power *= x;  
}
```

↓

```
.L3:  
ldr    r5, [r3], #4  
cmp    r1, r3  
mla    r0, r4, r5, r0  
mul    r4, r2, r4  
bne    .L3
```

// r5 <- coef[j]; j++ (two operations)
// compare: j < terms?
// value += r5 * power (mul + add)
// power *= x
// repeat?

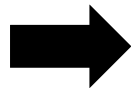
Example: Polynomial evaluation

▪ Executing `poly(A, 3, x)`

```
cmp      r1, #0
ble      .L4
push     {r4, r5}
mov      r3, r0
add      r1, r0, r1, lsl #2
movs     r4, #1
movs     r0, #0
ldr      r5, [r3], #4
cmp      r1, r3
mla      r0, r4, r5, r0
mul      r4, r2, r4
bne      .L3
...
```

Example: Polynomial evaluation

▪ Executing poly(A, 3, x)

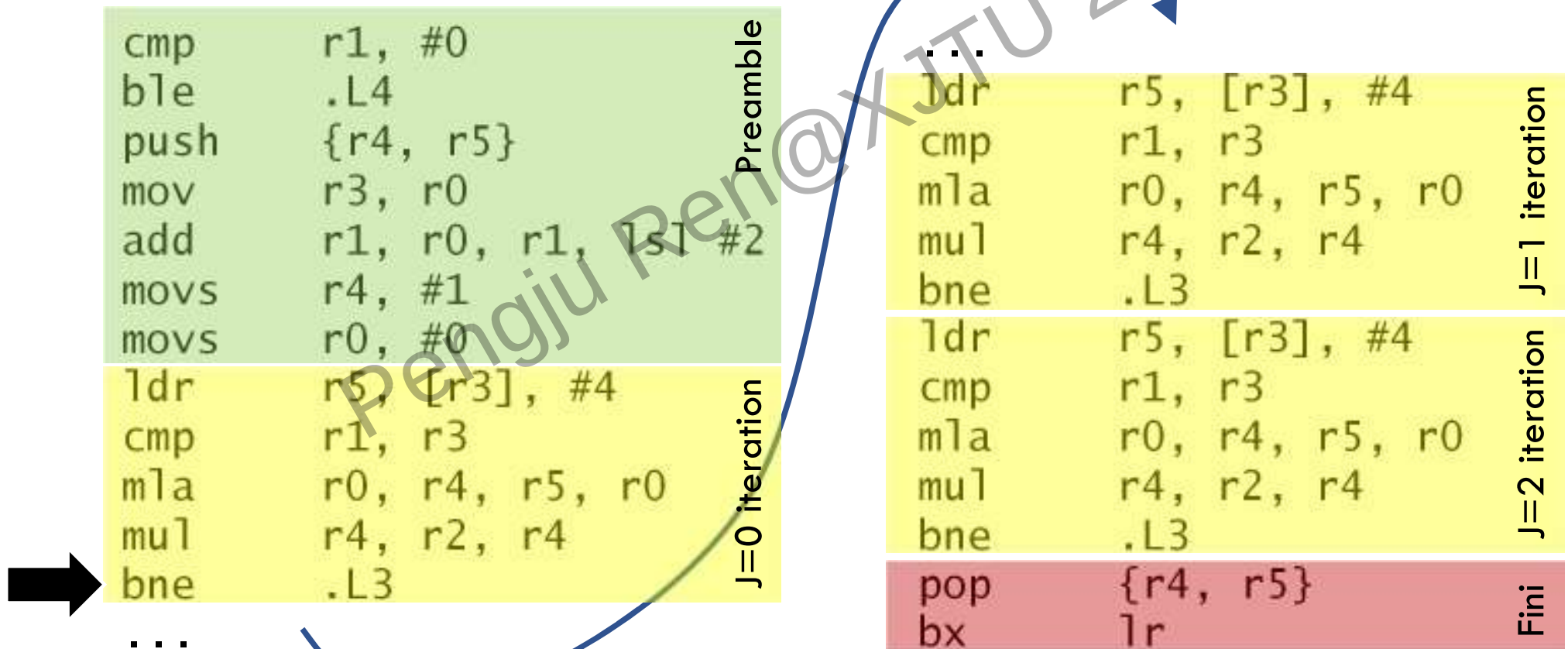


cmp	r1, #0	Preamble
ble	.L4	
push	{r4, r5}	
mov	r3, r0	
add	r1, r0, r1, lsl #2	
movs	r4, #1	
movs	r0, #0	
ldr	r5, [r3], #4	J=0 iteration
cmp	r1, r3	
mla	r0, r4, r5, r0	
mul	r4, r2, r4	
bne	.L3	

...

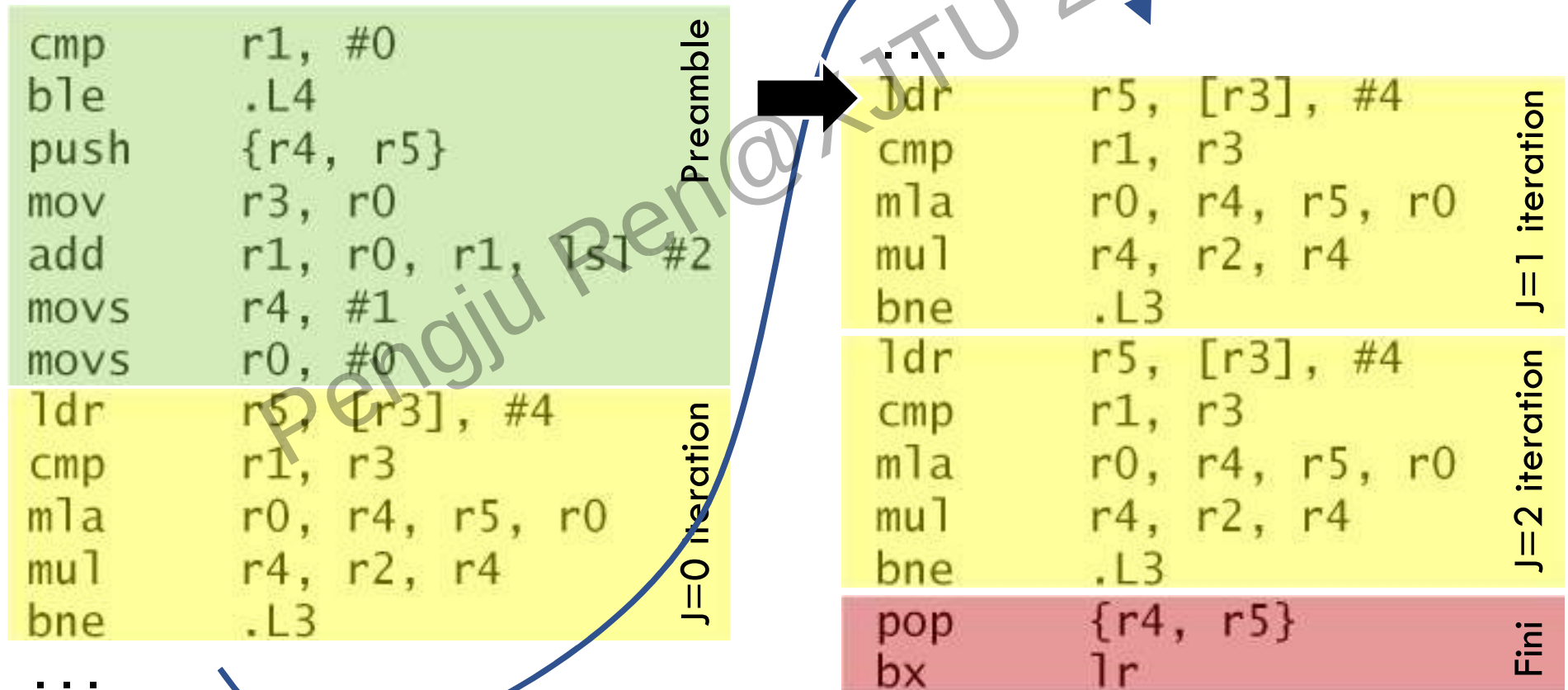
Example: Polynomial evaluation

▪ Executing poly(A, 3, x)



Example: Polynomial evaluation

▪ Executing poly(A, 3, x)



The software-hardware boundary

- The *instruction set architecture (ISA)* is a functional contract between hardware and software
 - It says what each instruction does, but not how
 - Example: Ordered sequence of x86 instructions
- A processor's *microarchitecture* is how the ISA is implemented

Arch : μ Arch :: Interface : Implementation

Simple CPU model

- **Execute instructions in program order**
- **Divide instruction execution into stages, e.g.:**
 - **1. Fetch – get the next instruction from memory**
 - **2. Decode – figure out what to do & read inputs**
 - **3. Execute – perform the necessary operations**
 - **4. Commit – write the results back to registers / memory**
- **(Real processors have many more stages)**

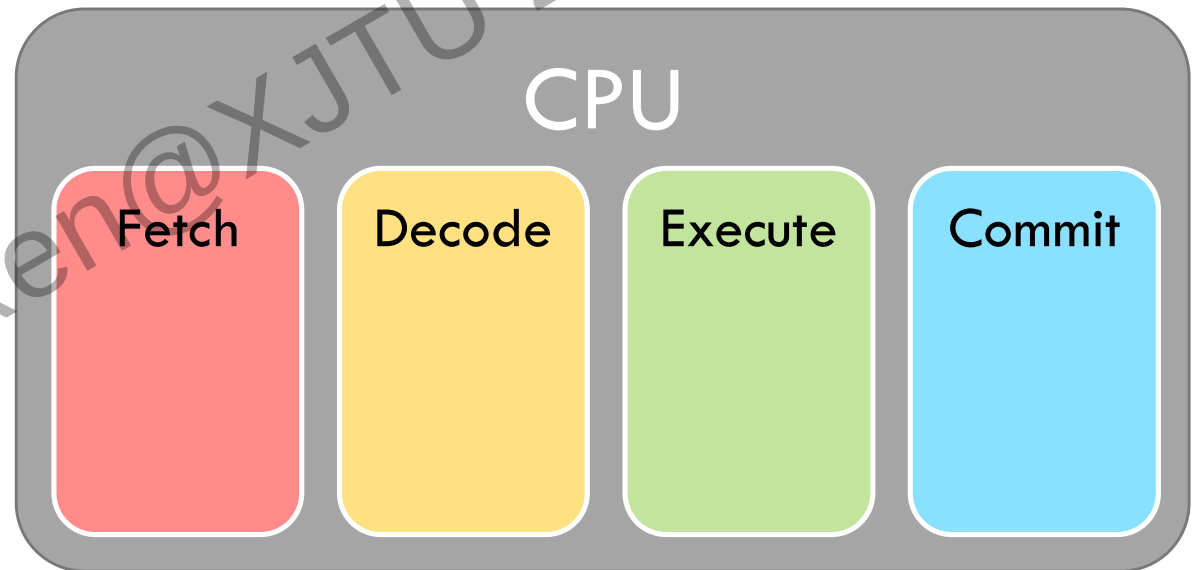
Evaluating polynomial on the simple CPU model

➔

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

...
```

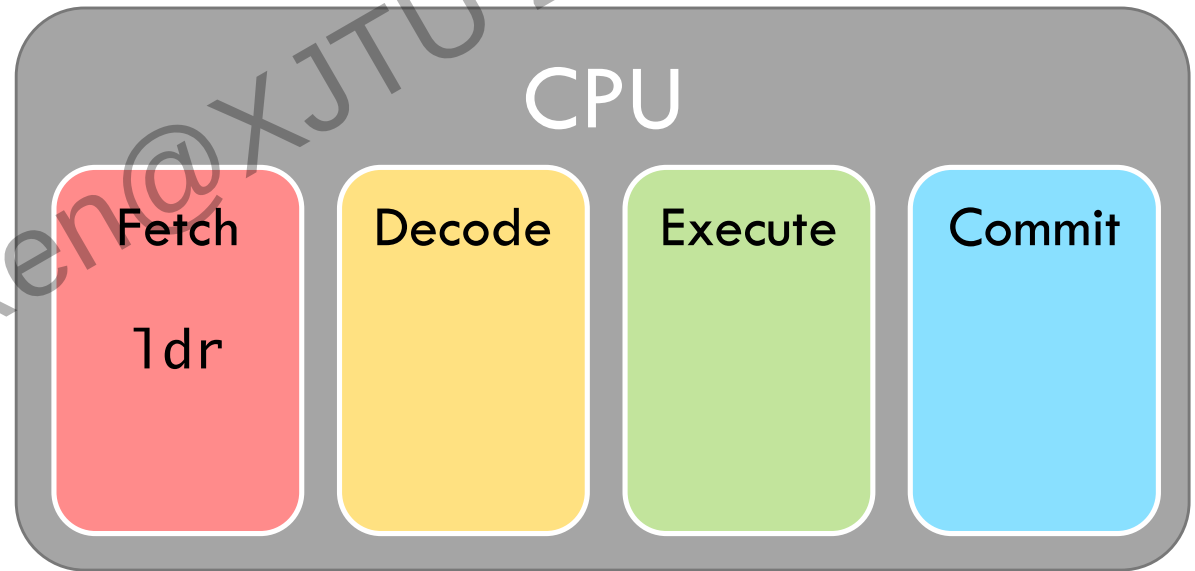


Evaluating polynomial on the simple CPU model

➔
ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3

ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3

...



1. Read "ldr r5, [r3] #4"
from memory

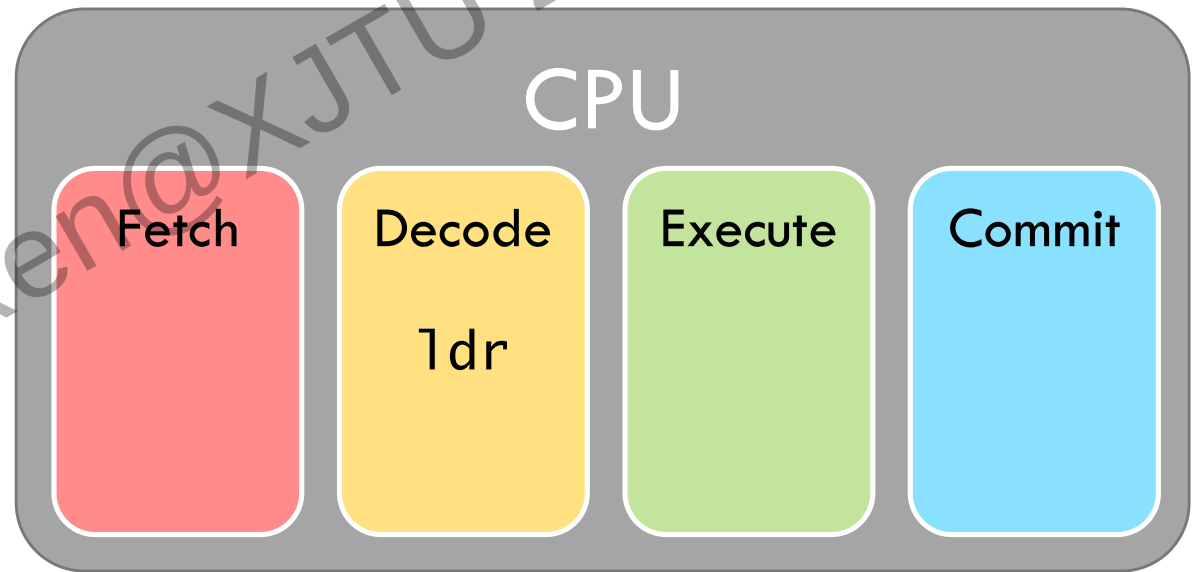
Evaluating polynomial on the simple CPU model

➔

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

...
```



**2. Decode “ldr r5, [r3] #4”
and read input regs**

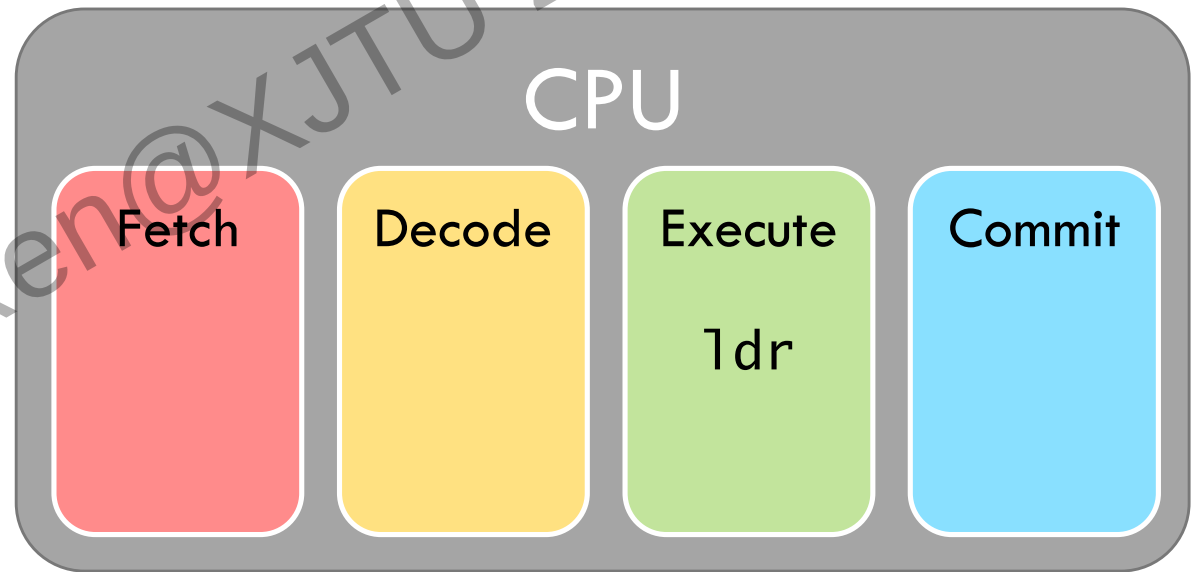
Evaluating polynomial on the simple CPU model

➔

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

...
```



**3. Load memory at r3 and
compute $r3 + 4$**

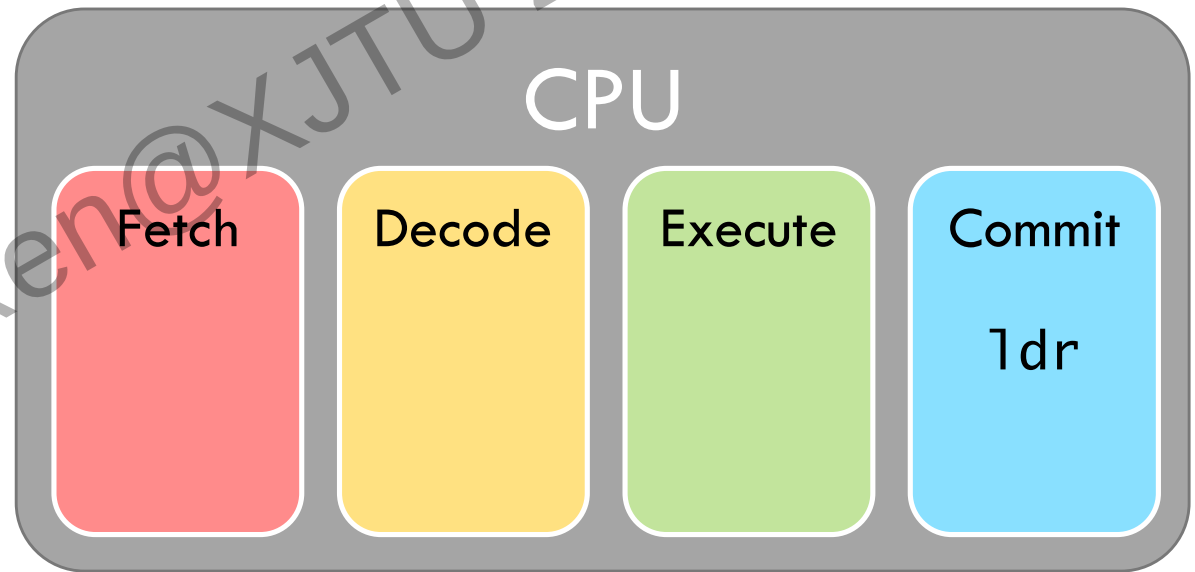
Evaluating polynomial on the simple CPU model

➔

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

...
```



**4. Write values
into regs r5 and r3**

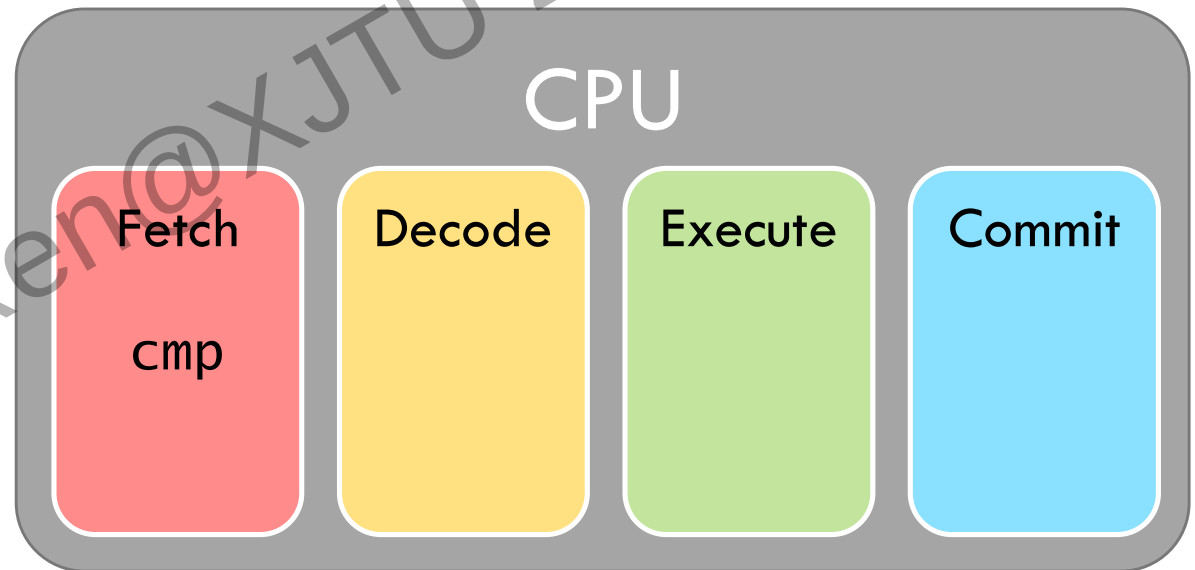
Evaluating polynomial on the simple CPU model

➔

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

...
```



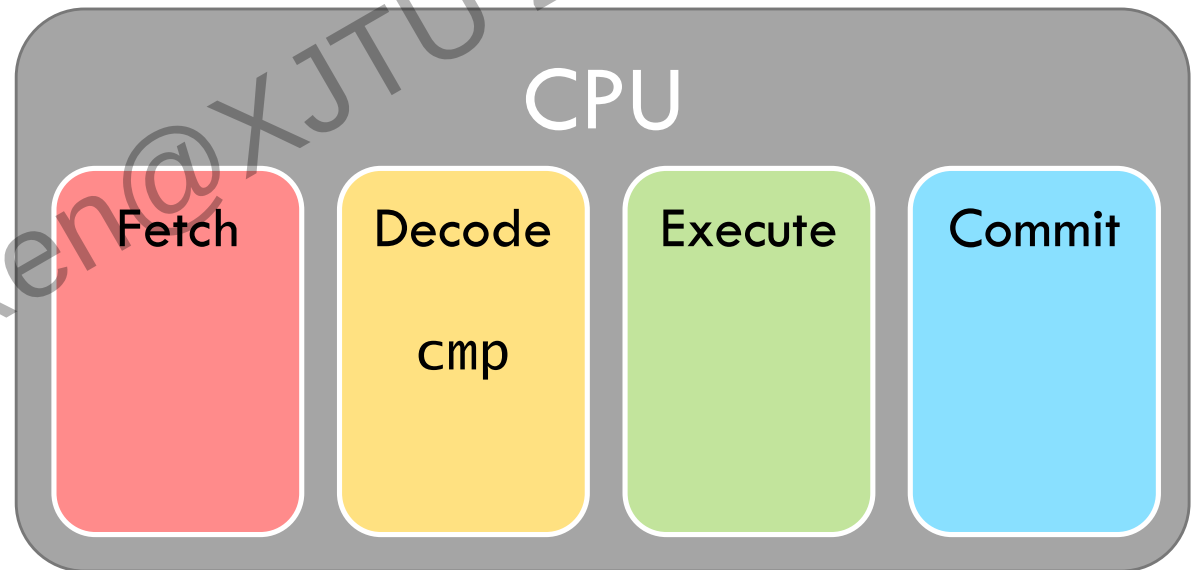
Evaluating polynomial on the simple CPU model

➔

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

...
```



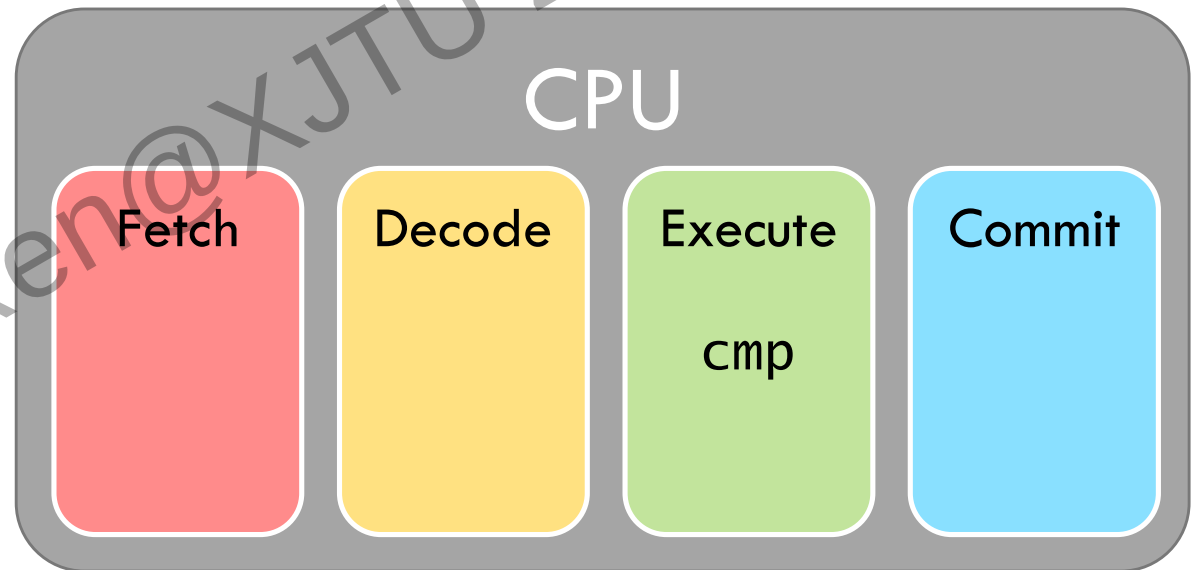
Evaluating polynomial on the simple CPU model

➔

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

...
```



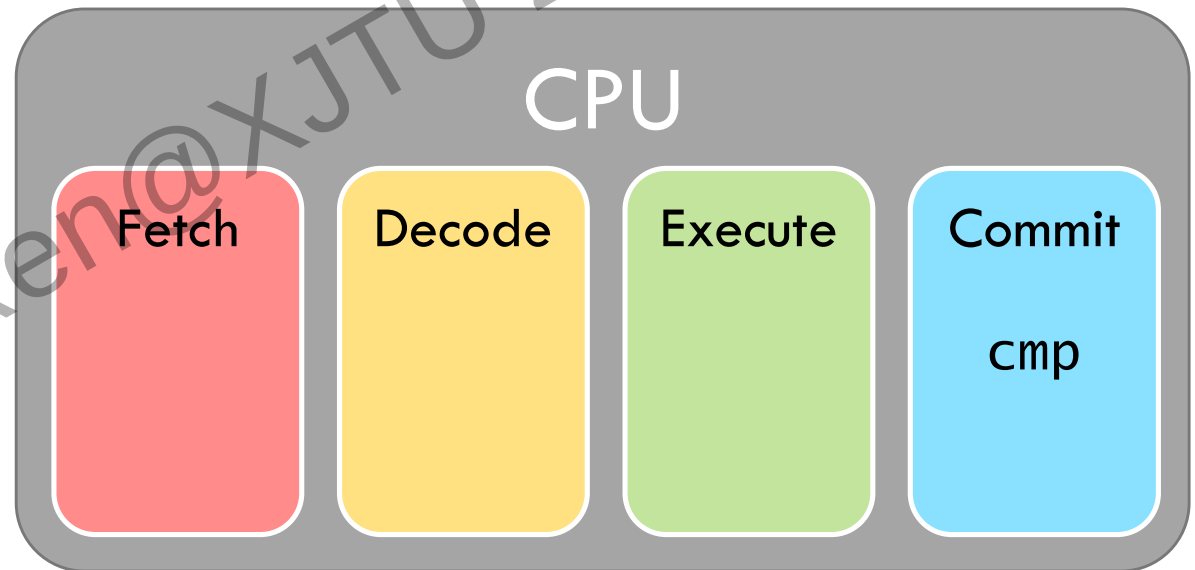
Evaluating polynomial on the simple CPU model

➔

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

...
```



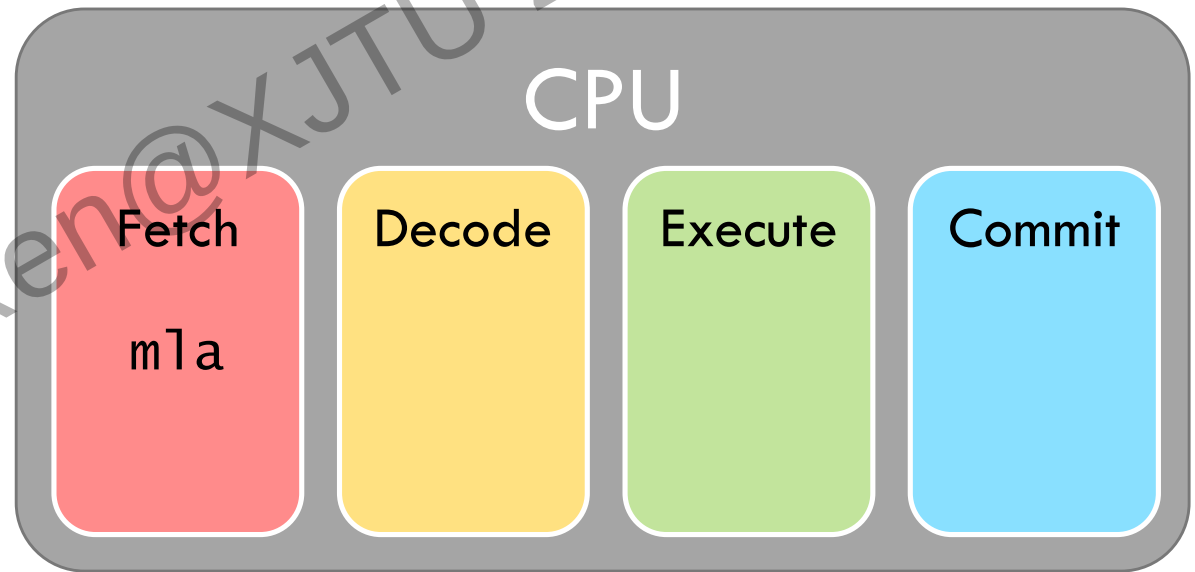
Evaluating polynomial on the simple CPU model

➔

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

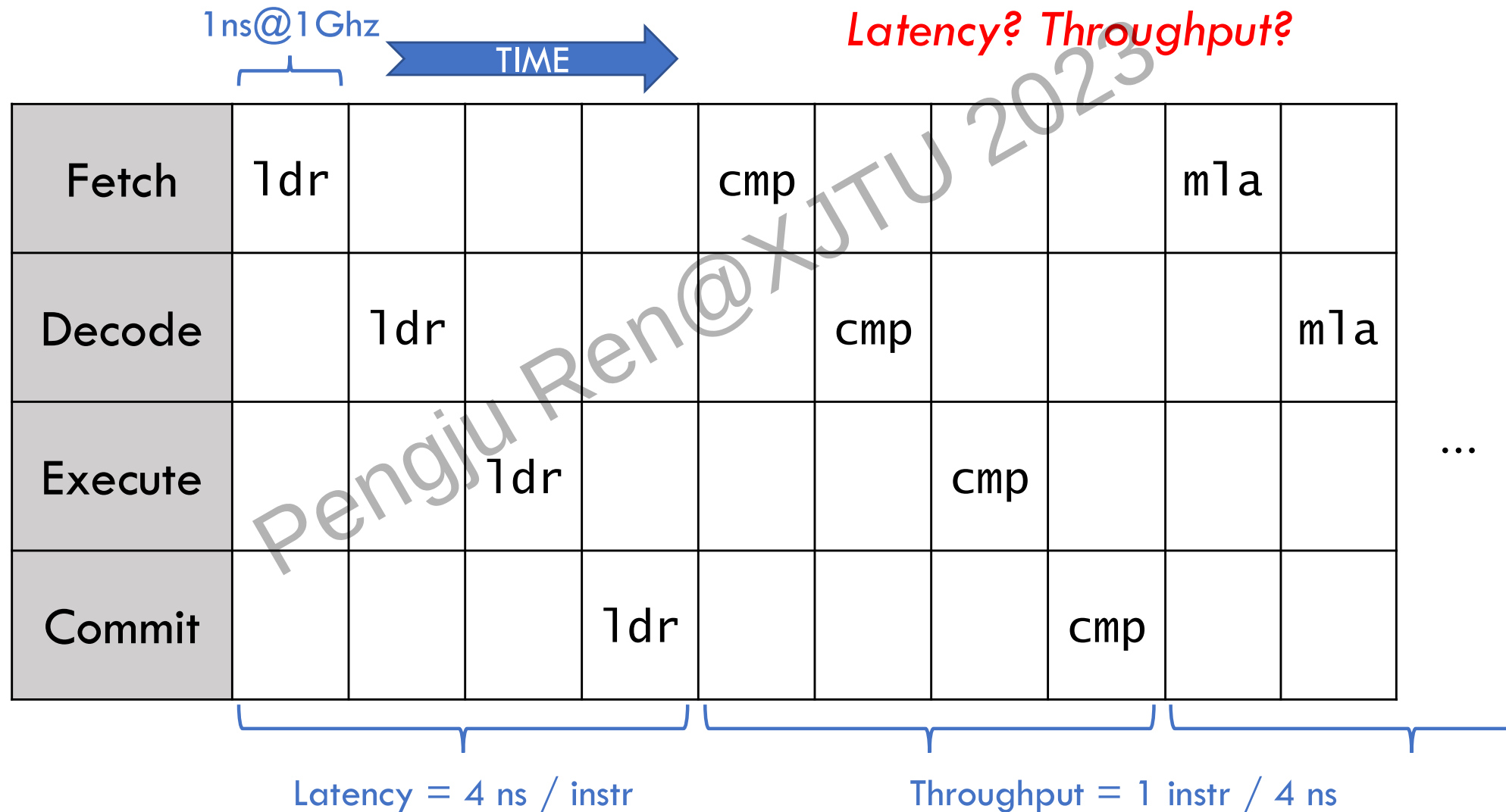
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

...
```

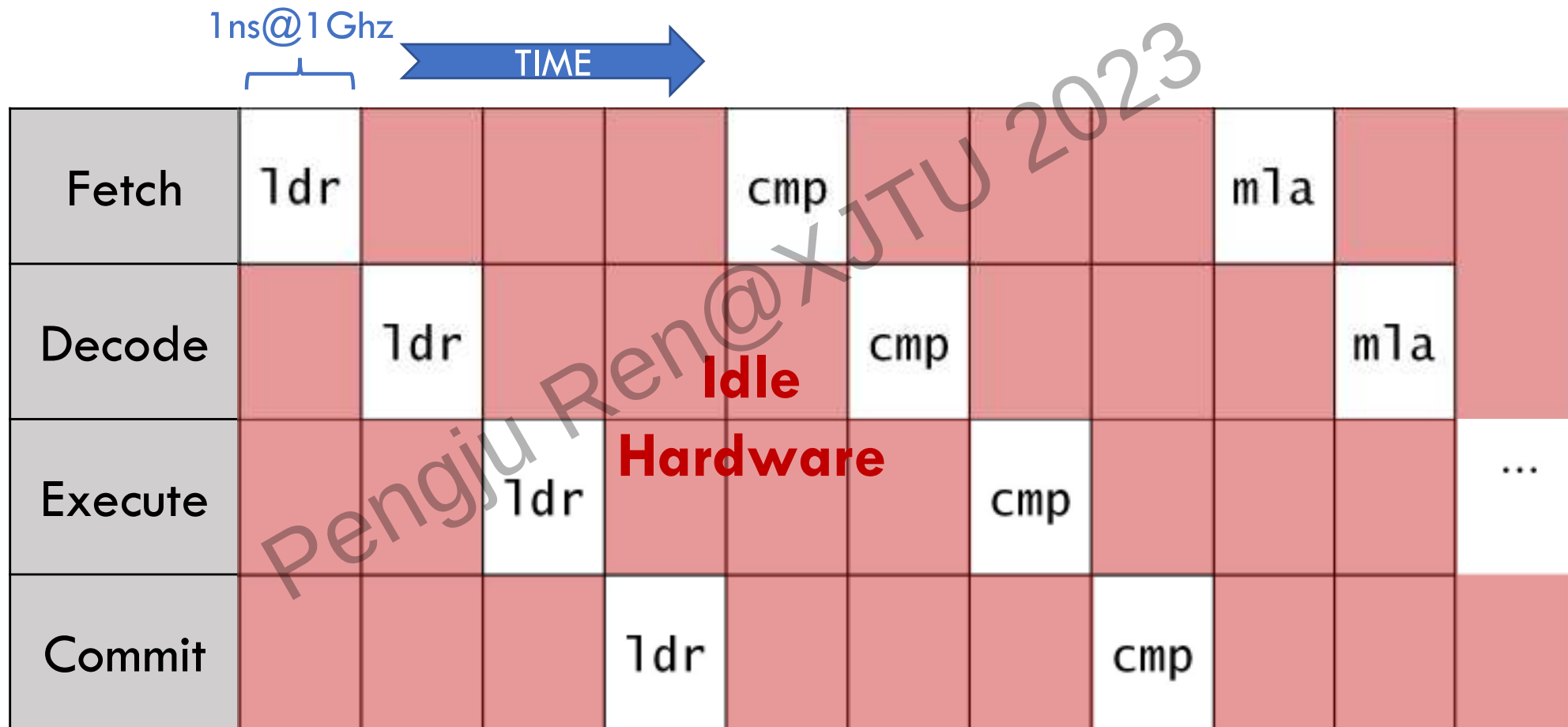


Evaluating polynomial on the simple CPU model

*How fast is this processor?
Latency? Throughput?*



Simple CPU is very wasteful

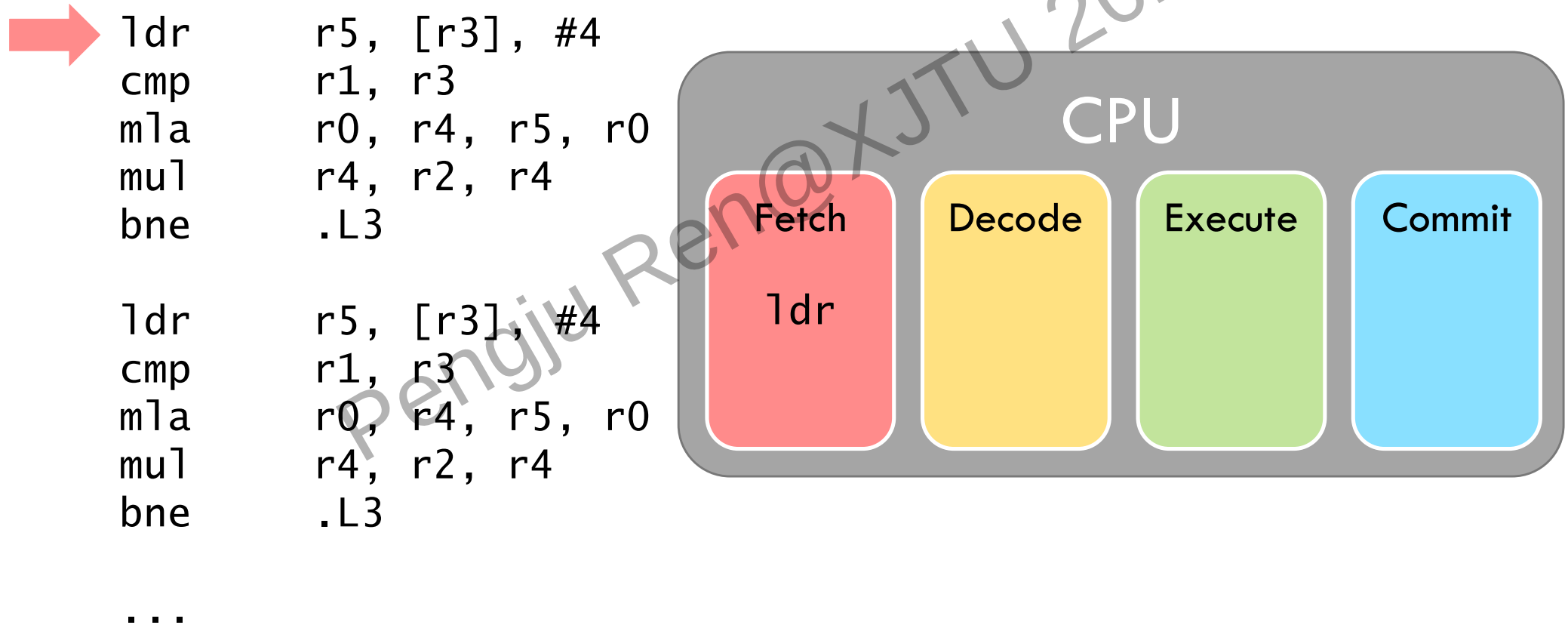


Review: Pipelining

Pengju Ren@NITU 2023

Pipelining keeps CPU busy through instruction-level parallelism

- Idea: Start on the next instr'n immediately



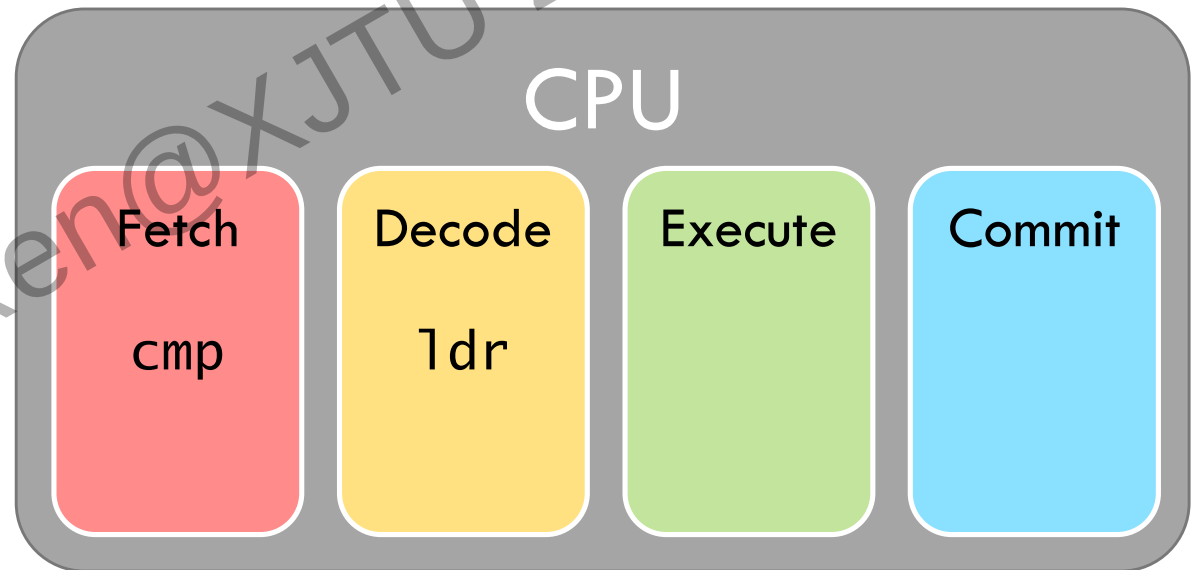
Pipelining keeps CPU busy through instruction-level parallelism

- Idea: Start on the next instr'n immediately

➡
➡
ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3

ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3

...



Pipelining keeps CPU busy through instruction-level parallelism

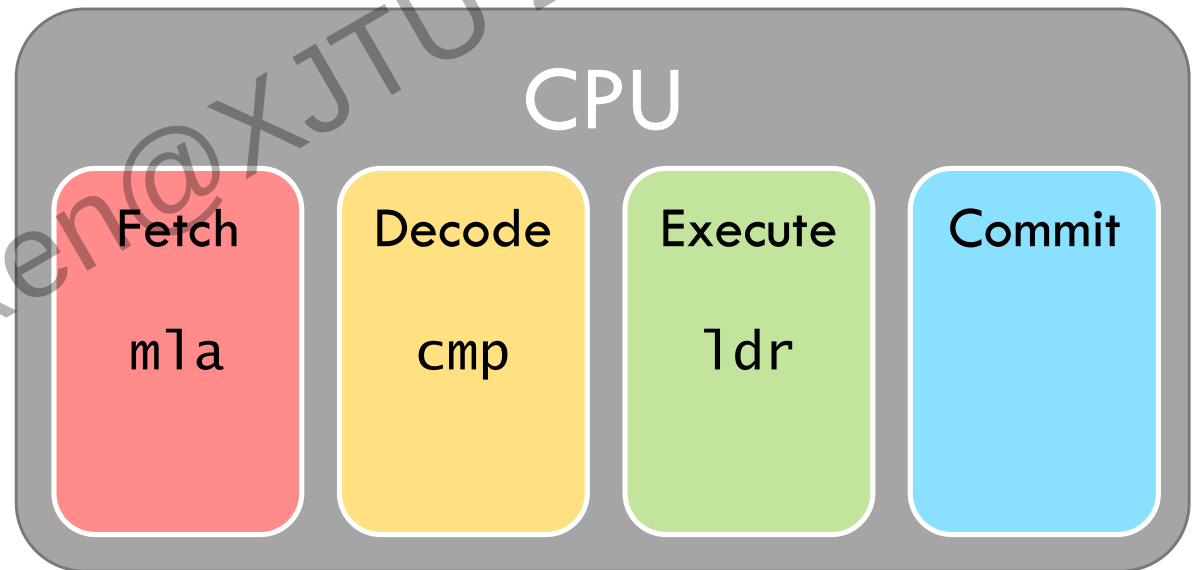
- Idea: Start on the next instr'n immediately

➡
➡
➡

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3

...
```



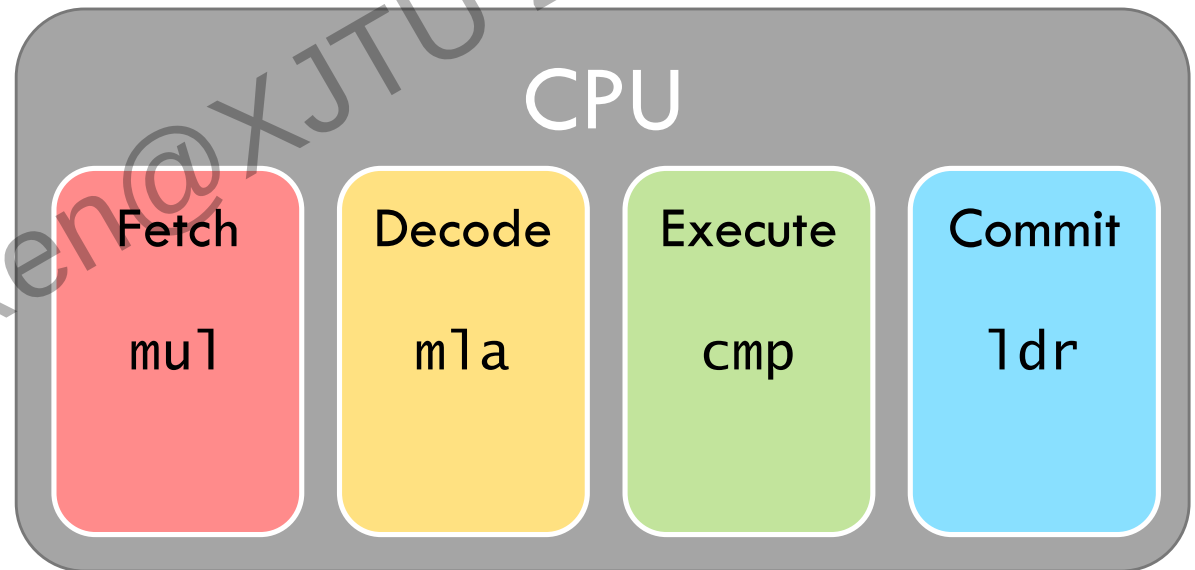
Pipelining keeps CPU busy through instruction-level parallelism

- Idea: Start on the next instr'n immediately

➡ ldr r5, [r3], #4
➡ cmp r1, r3
➡ mla r0, r4, r5, r0
➡ mul r4, r2, r4
bne .L3

ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3

...



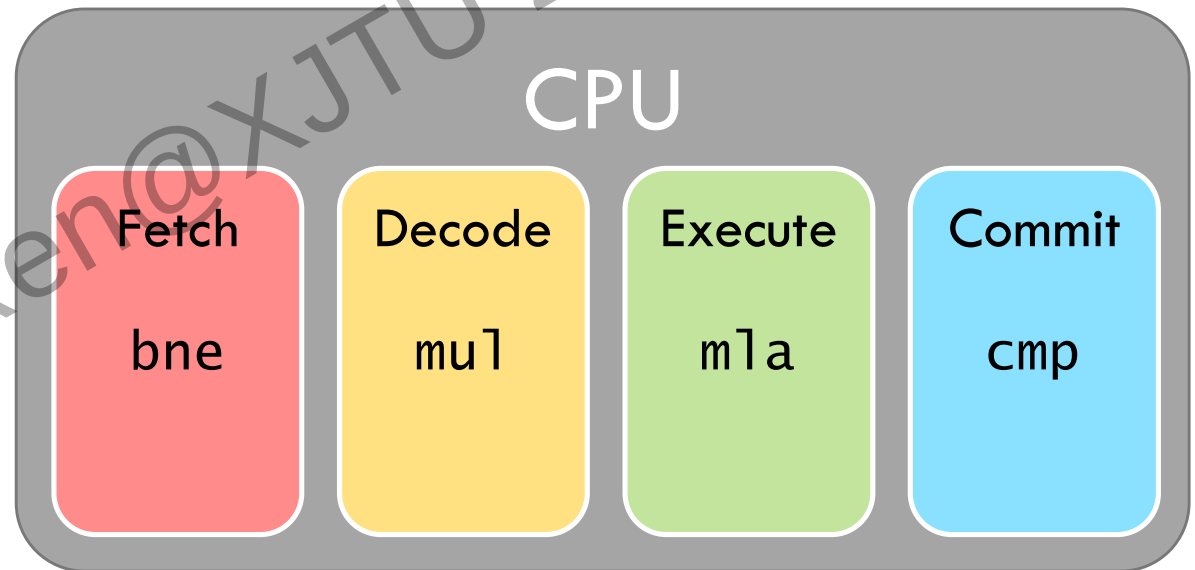
Pipelining keeps CPU busy through instruction-level parallelism

- Idea: Start on the next instr'n immediately

→ ldr r5, [r3], #4
→ cmp r1, r3
→ mla r0, r4, r5, r0
→ mul r4, r2, r4
→ bne .L3

ldr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3

...



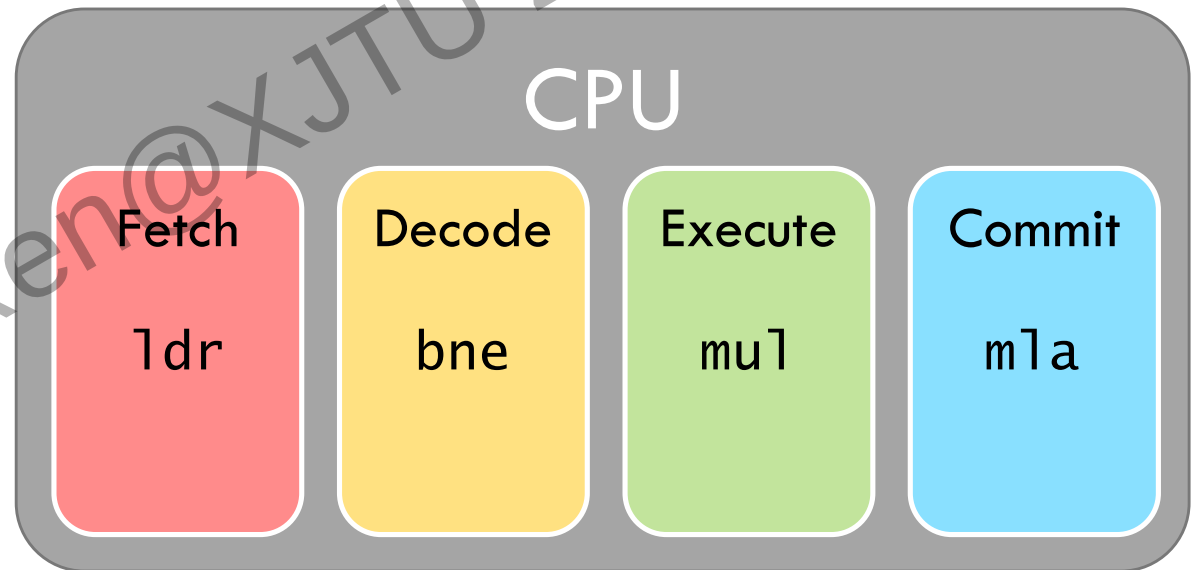
Pipelining keeps CPU busy through instruction-level parallelism

- Idea: Start on the next instr'n immediately

1dr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3

1dr r5, [r3], #4
cmp r1, r3
mla r0, r4, r5, r0
mul r4, r2, r4
bne .L3

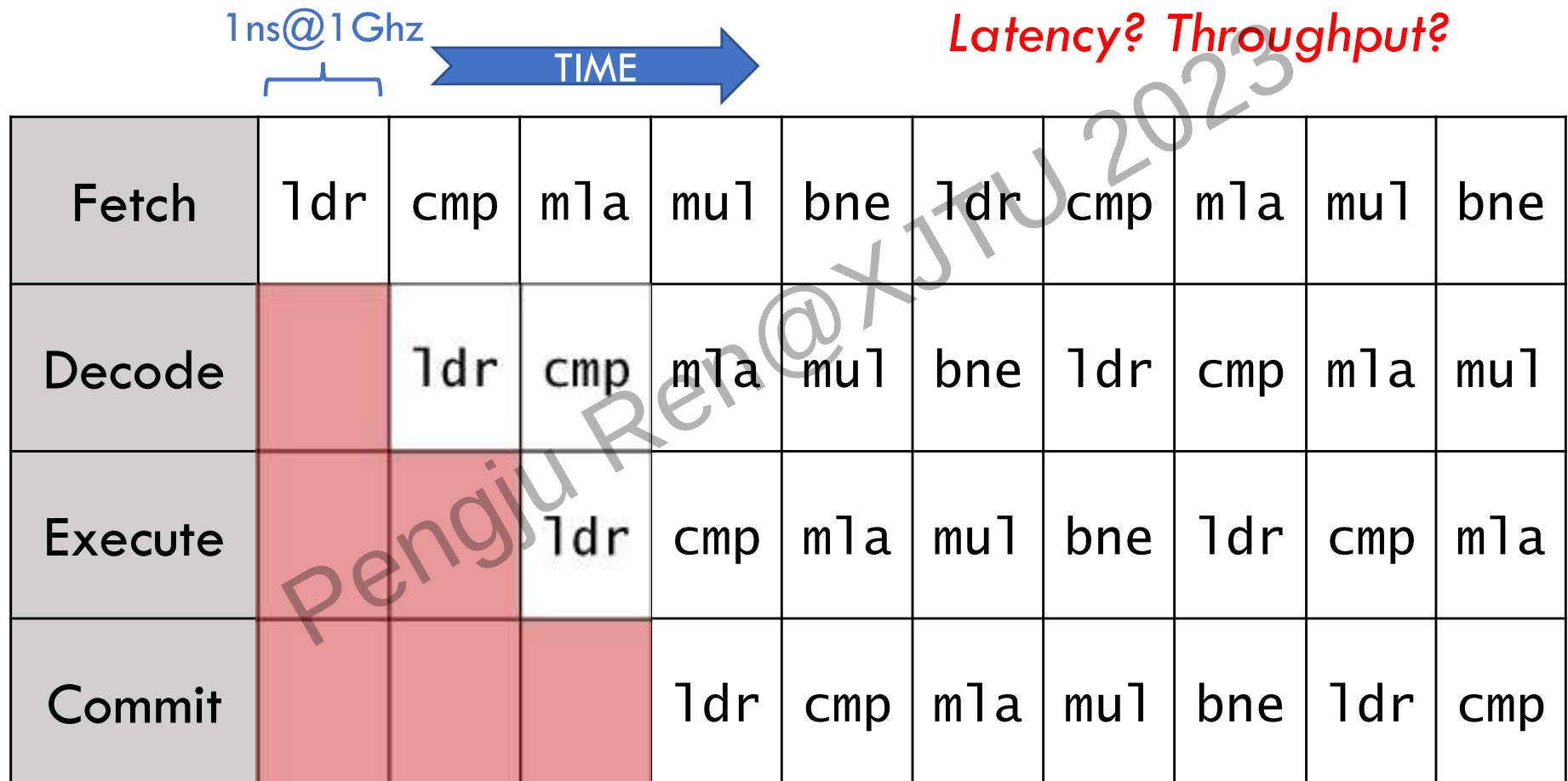
...



Evaluating polynomial on the pipelined CPU

How fast is this processor?

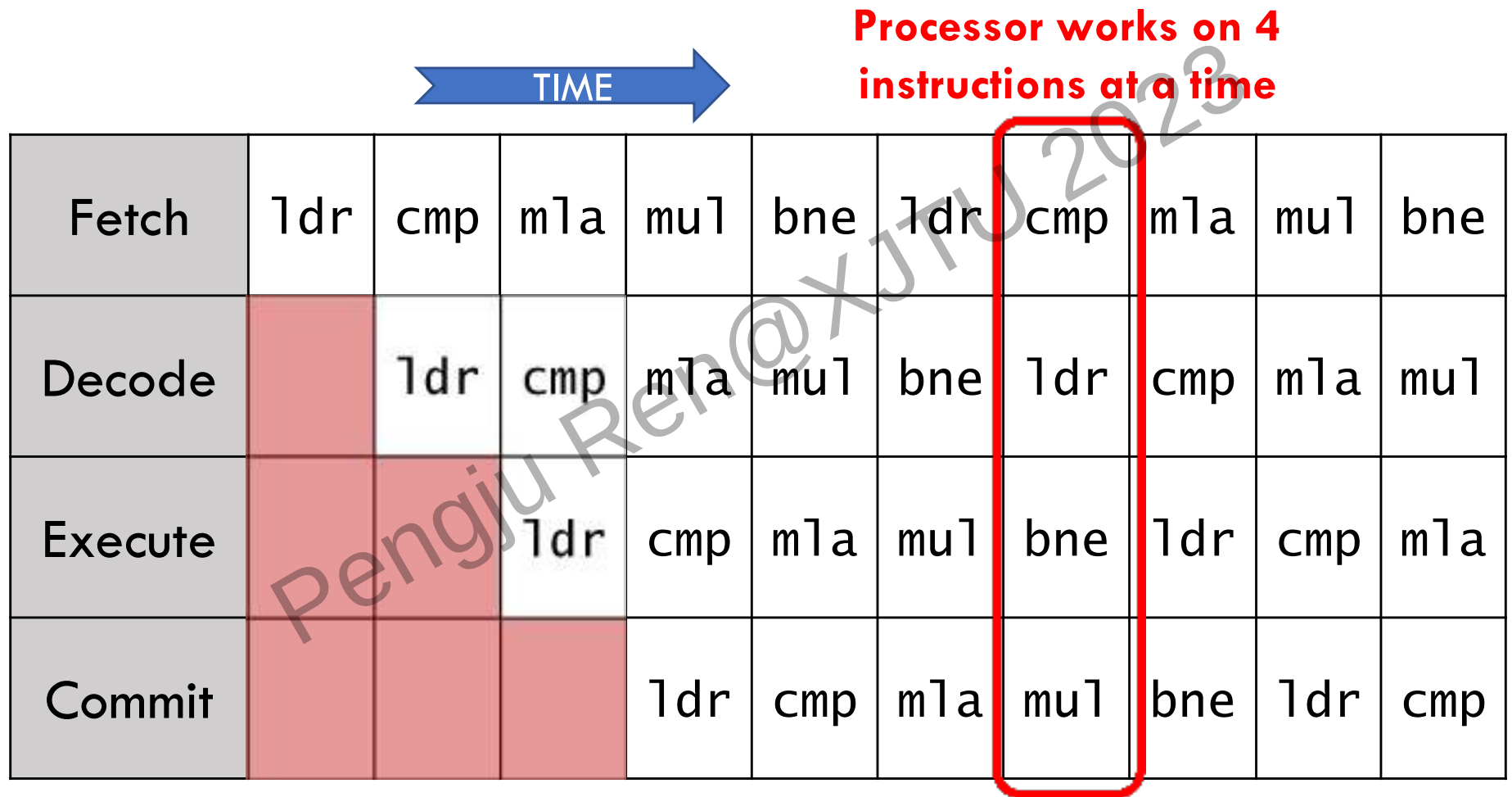
Latency? Throughput?



Latency = 4 ns / instr

Throughput = 1 instr / ns
4X speedup!

Speedup achieved through pipeline parallelism



Limitations of pipelining

- Parallelism requires independent work
- Q: Are instructions independent ?
- A: No! Many possible *hazards* limit parallelism...
 - ❑ Data hazards
 - ❑ Structure hazards
 - ❑ Control hazard

Data hazards

```
ldr rx, [rm], #4 // rx ← Memory[rm]; rm ← rm + 4
cmp ry, rn       // compare ry and rn
```

Q: When can the CPU pipeline the `cmp` behind `ldr`?

Fetch	ldr	cmp	...	...	...	...
Decode		ldr	cmp	...	...	...
Execute			ldr	cmp	...	...
Commit				ldr	cmp	...

■ **A: When they use different registers**

- Specifically, when `cmp` does not read any data written by `ldr`
- E.g.,
 - `rx != ry`
 - `rx != rn`
 - `rm != rn`
 - `rm != ry`

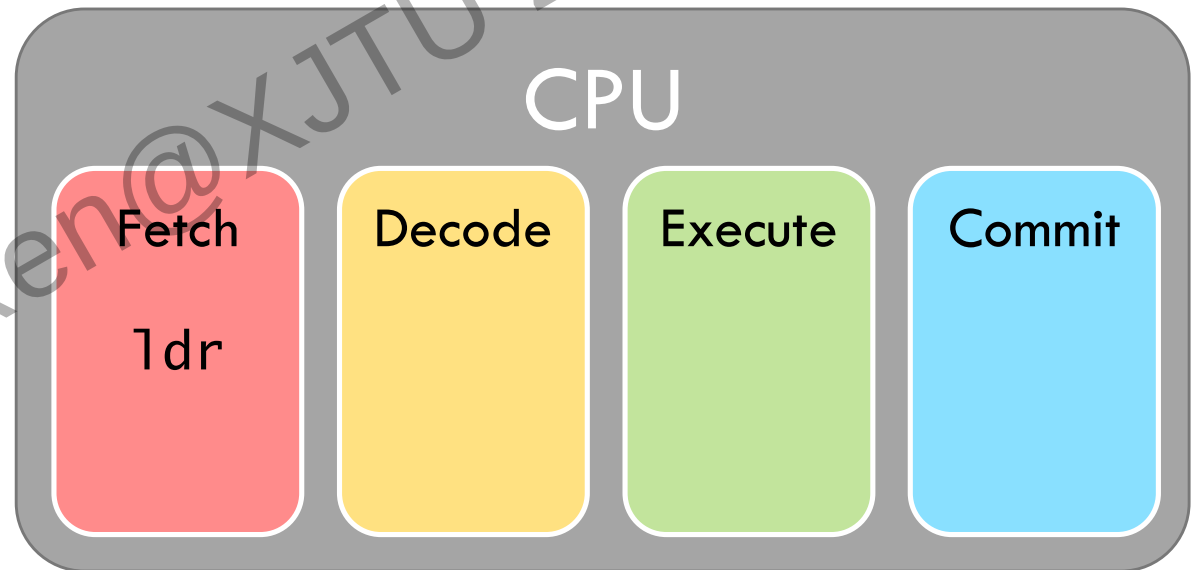
Dealing with data hazards: Stalling the pipeline

▪ Cannot pipeline `cmp` (`ldr` writes `r3`)

➔
`ldr r5, [r3], #4`
`cmp r1, r3`
`mla r0, r4, r5, r0`
`mul r4, r2, r4`
`bne .L3`

`ldr r5, [r3], #4`
`cmp r1, r3`
`mla r0, r4, r5, r0`
`mul r4, r2, r4`
`bne .L3`

...



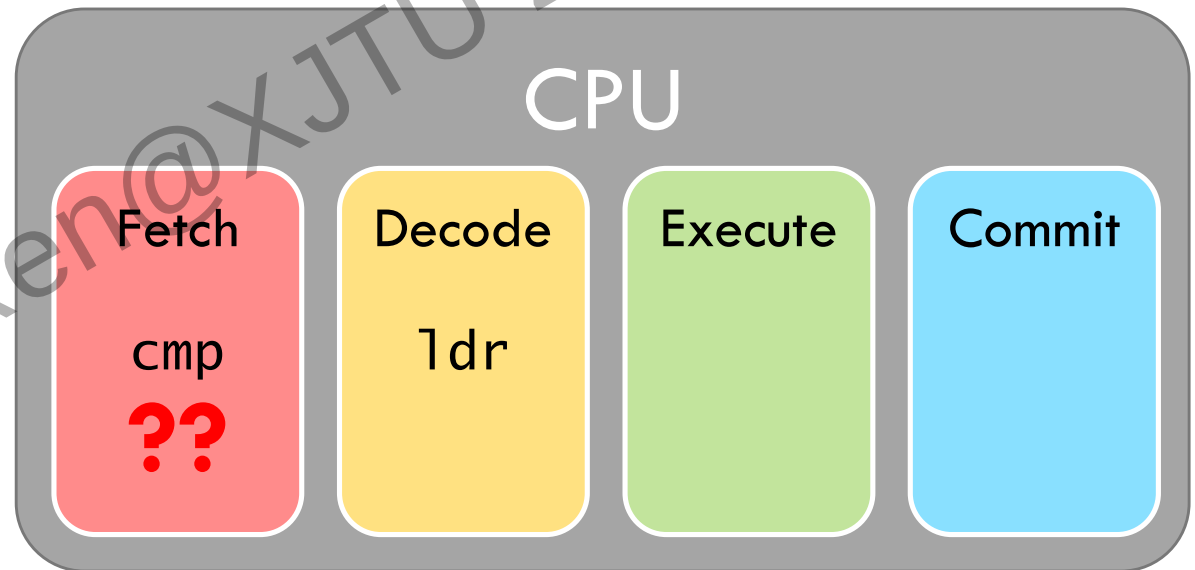
Dealing with data hazards: Stalling the pipeline

▪ Cannot pipeline `cmp` (`ldr` writes `r3`)

→ `ldr` `r5, [r3], #4`
→ `cmp` `r1, r3`
`m1a` `r0, r4, r5, r0`
`mul` `r4, r2, r4`
`bne` `.L3`

`ldr` `r5, [r3], #4`
`cmp` `r1, r3`
`m1a` `r0, r4, r5, r0`
`mul` `r4, r2, r4`
`bne` `.L3`

...



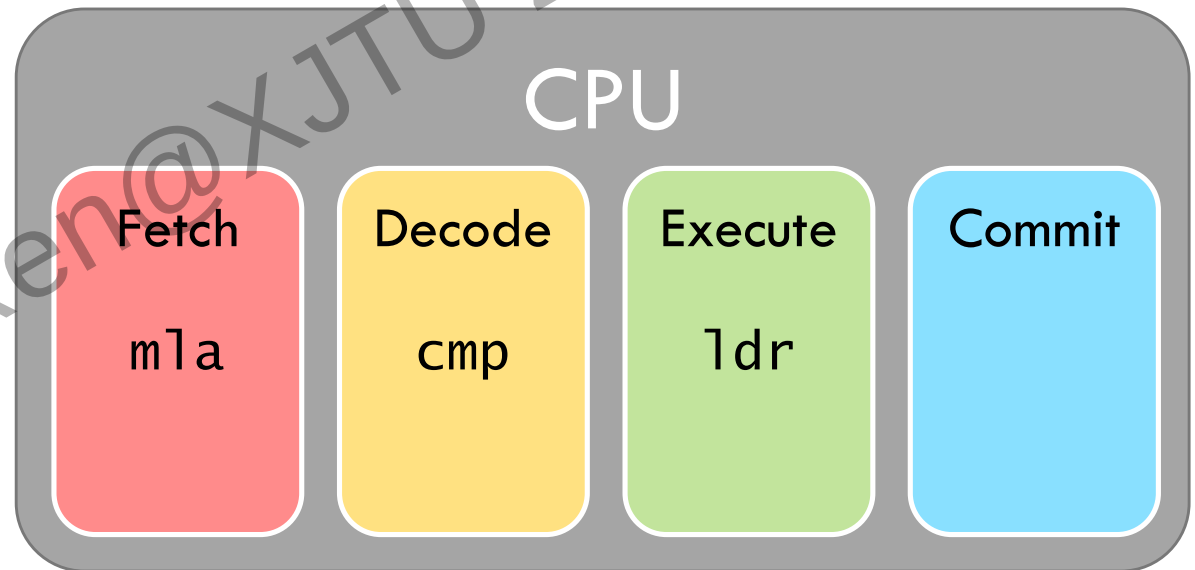
Dealing with data hazards: Stalling the pipeline

▪ Cannot pipeline `cmp` (`ldr` writes `r3`)

➡ `ldr` `r5, [r3], #4`
➡ `cmp` `r1, r3`
➡ `m1a` `r0, r4, r5, r0`
`mul` `r4, r2, r4`
`bne` `.L3`

`ldr` `r5, [r3], #4`
`cmp` `r1, r3`
`m1a` `r0, r4, r5, r0`
`mul` `r4, r2, r4`
`bne` `.L3`

...



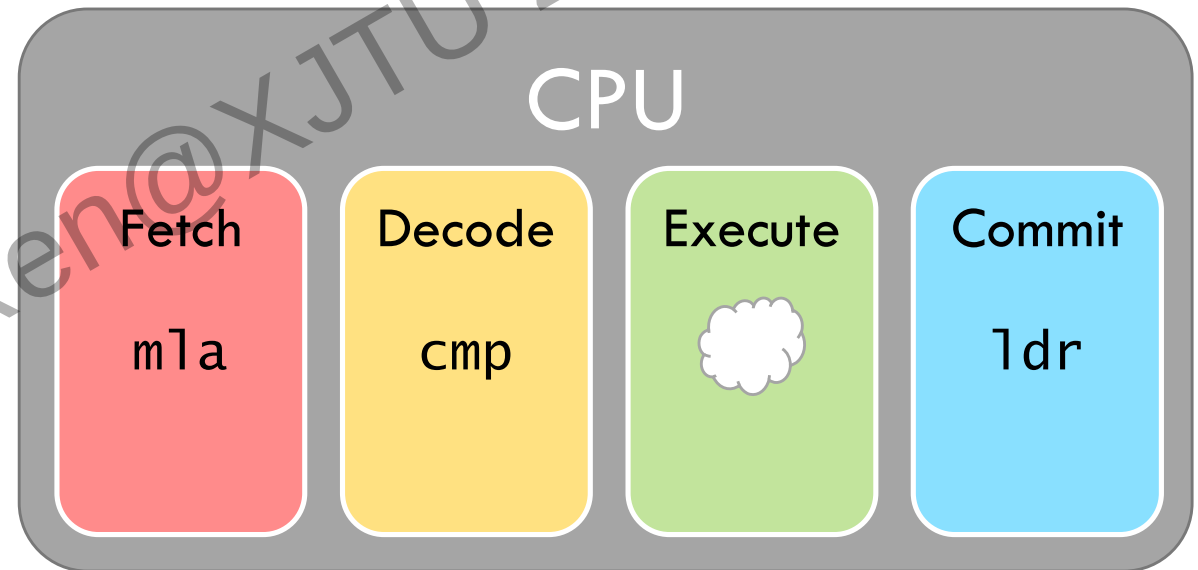
Dealing with data hazards: Stalling the pipeline

■ Cannot pipeline `cmp` (`ldr` writes `r3`)

→ `ldr` `r5, [r3], #4`
→ `cmp` `r1, r3`
→ `m1a` `r0, r4, r5, r0`
`mul` `r4, r2, r4`
`bne` `.L3`

`ldr` `r5, [r3], #4`
`cmp` `r1, r3`
`m1a` `r0, r4, r5, r0`
`mul` `r4, r2, r4`
`bne` `.L3`

...



Inject a “bubble” (NOP)
into the pipeline

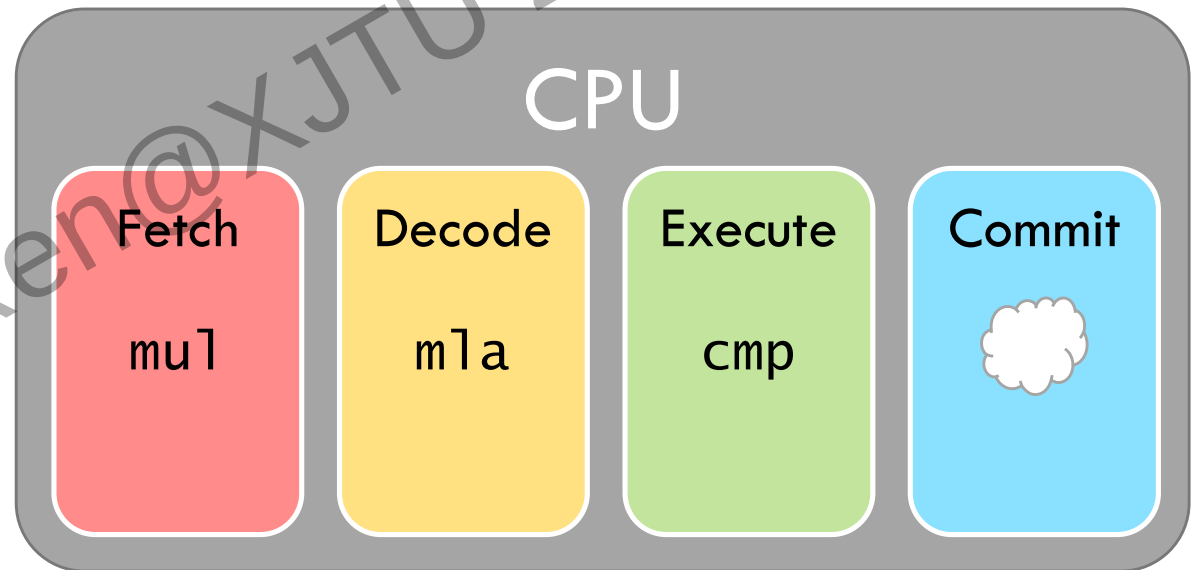
Dealing with data hazards: Stalling the pipeline

■ Cannot pipeline `cmp` (`ldr` writes `r3`)

→ `ldr` `r5, [r3], #4`
→ `cmp` `r1, r3`
→ `m1a` `r0, r4, r5, r0`
→ `mul` `r4, r2, r4`
 `bne` `.L3`

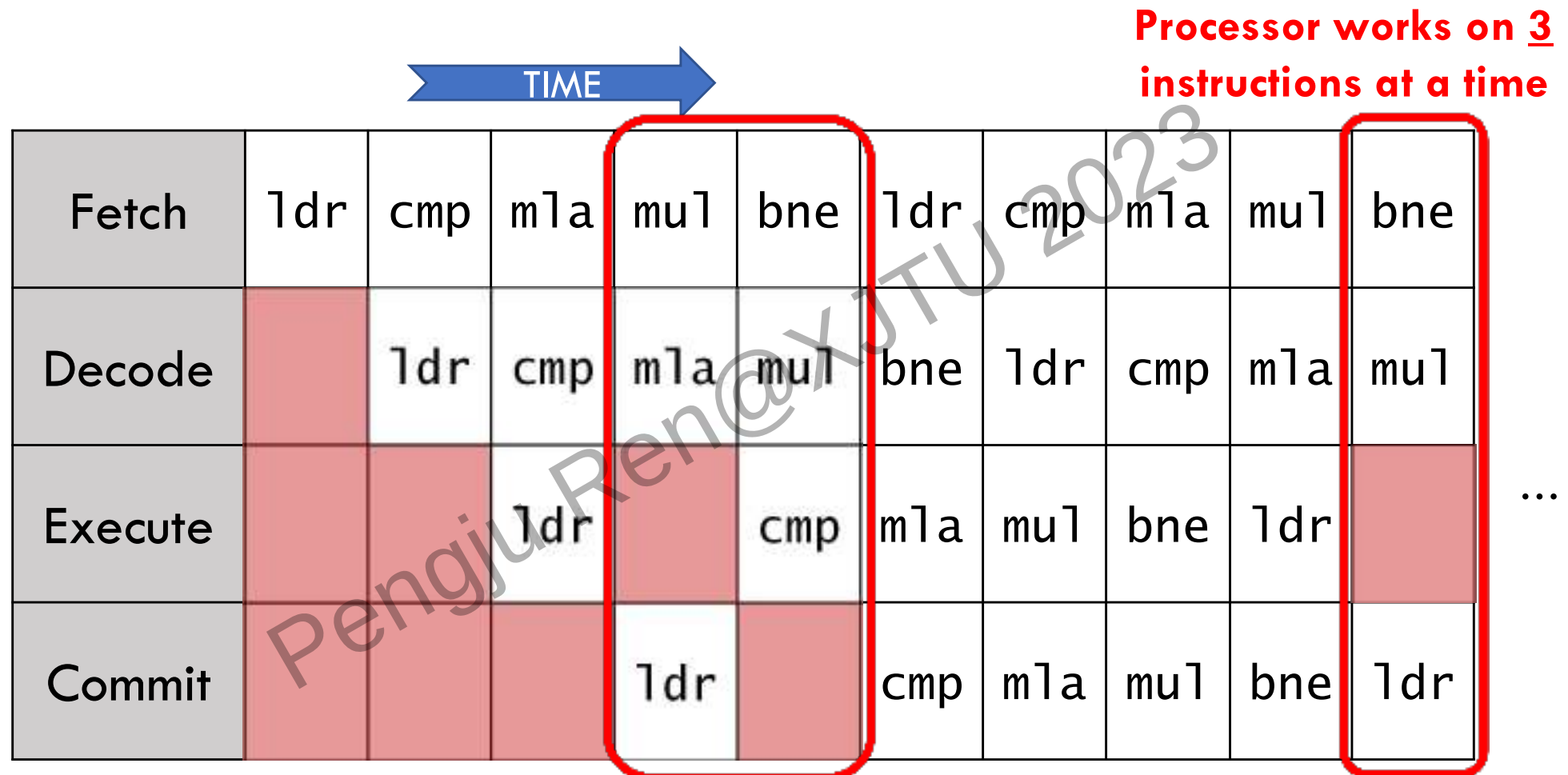
`ldr` `r5, [r3], #4`
`cmp` `r1, r3`
`m1a` `r0, r4, r5, r0`
`mul` `r4, r2, r4`
`bne` `.L3`

...



**cmp proceeds once `ldr`
has committed**

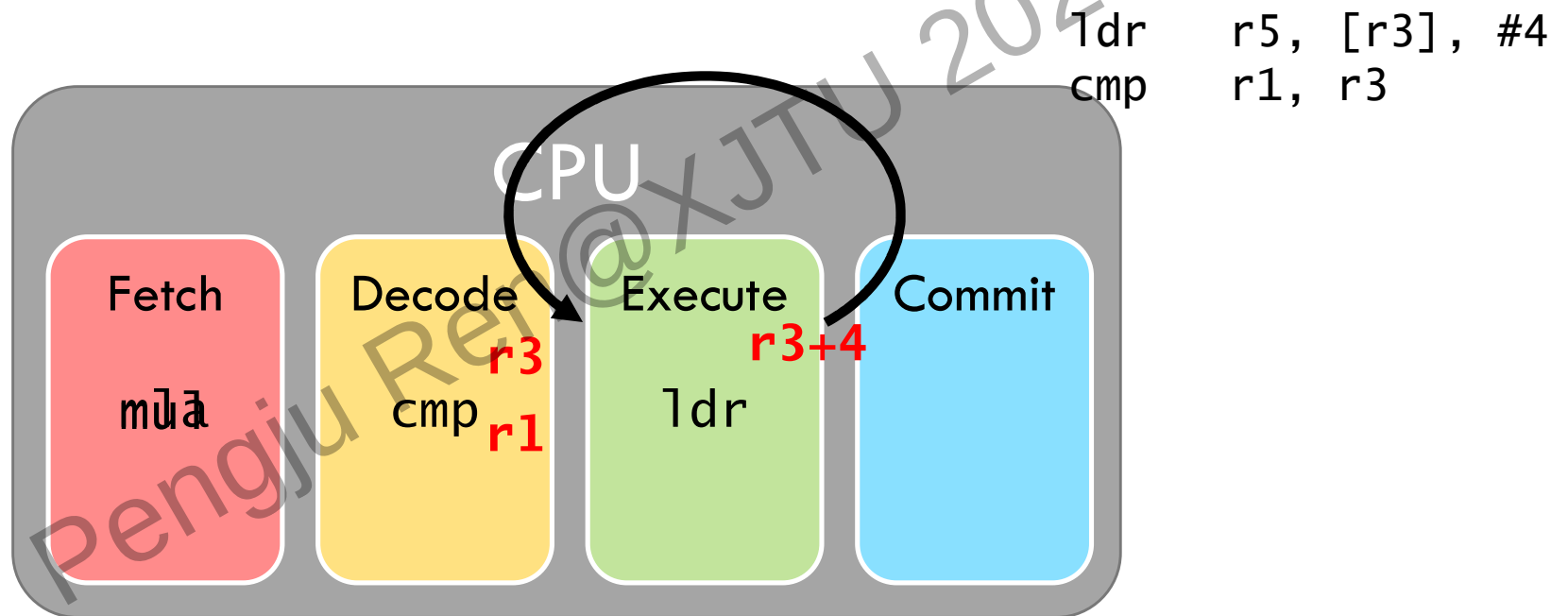
Stalling degrades performance



- But stalling is sometimes unavoidable
 - E.g., long-latency instructions (divide, cache miss)

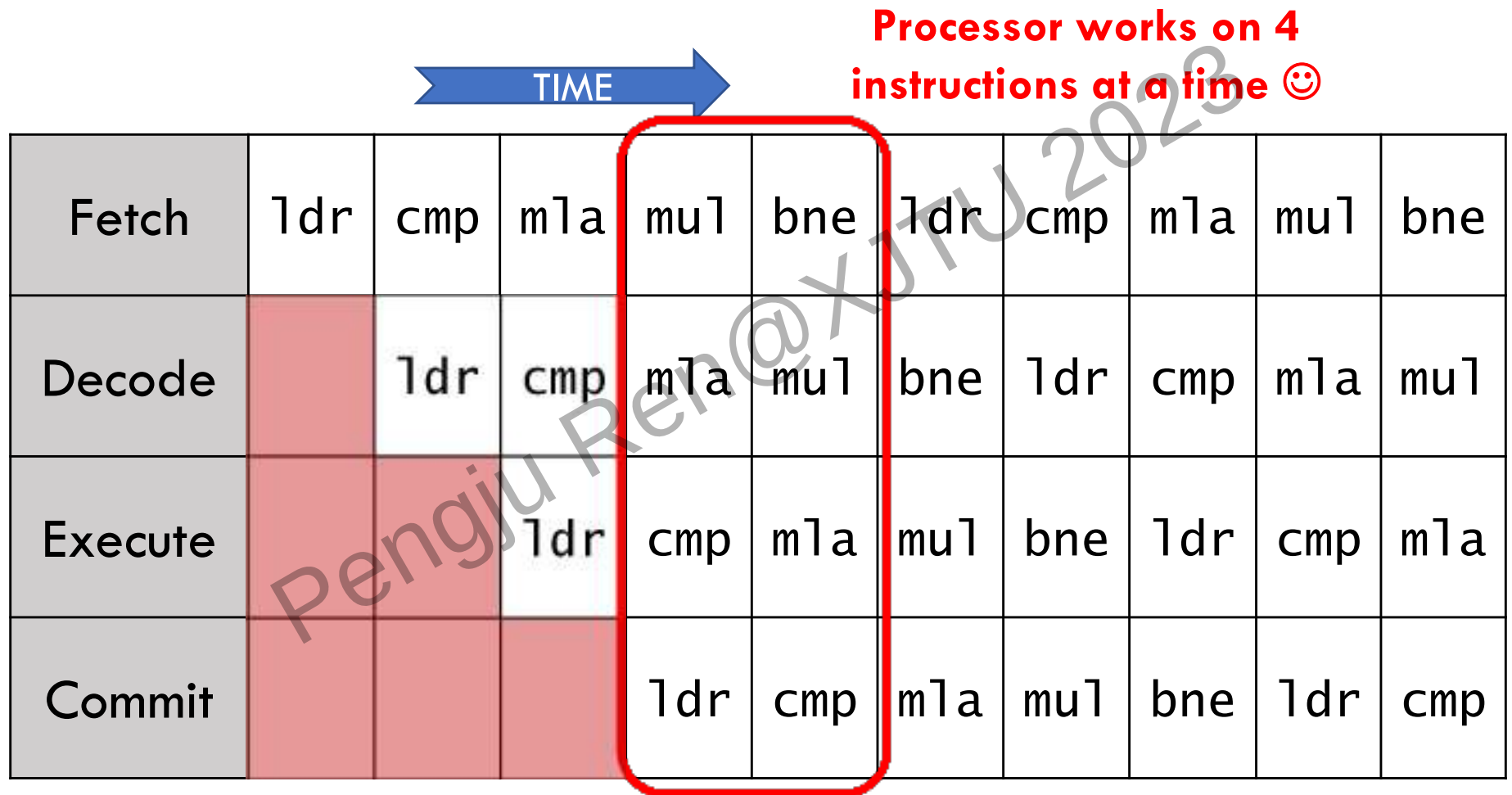
Dealing with data hazards: Forwarding data

- Wait a second... data is available after Execute!



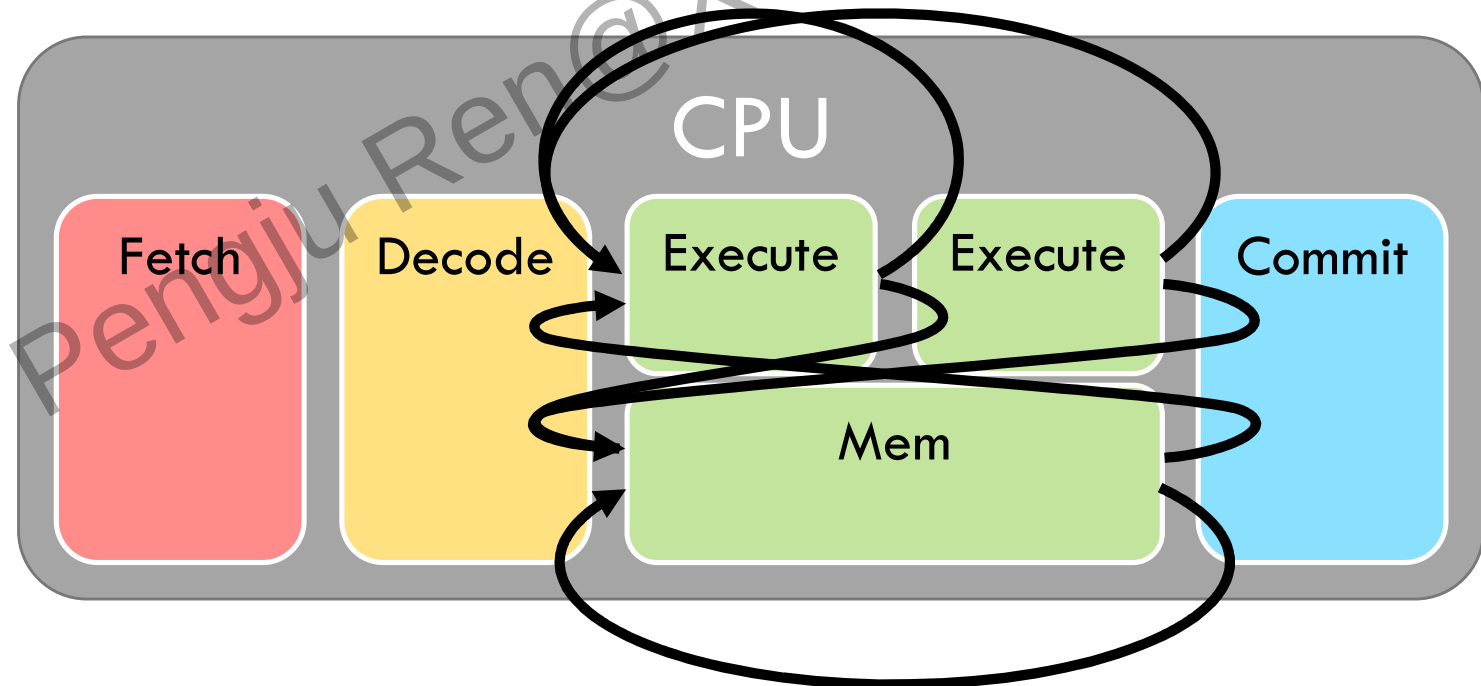
- Forwarding eliminates many (not all) pipeline stalls

Speedup achieved through pipeline parallelism



Pipelining is not free!

- Q: How well does forwarding scale?
- A: Not well... many forwarding paths in deep & complex pipelines

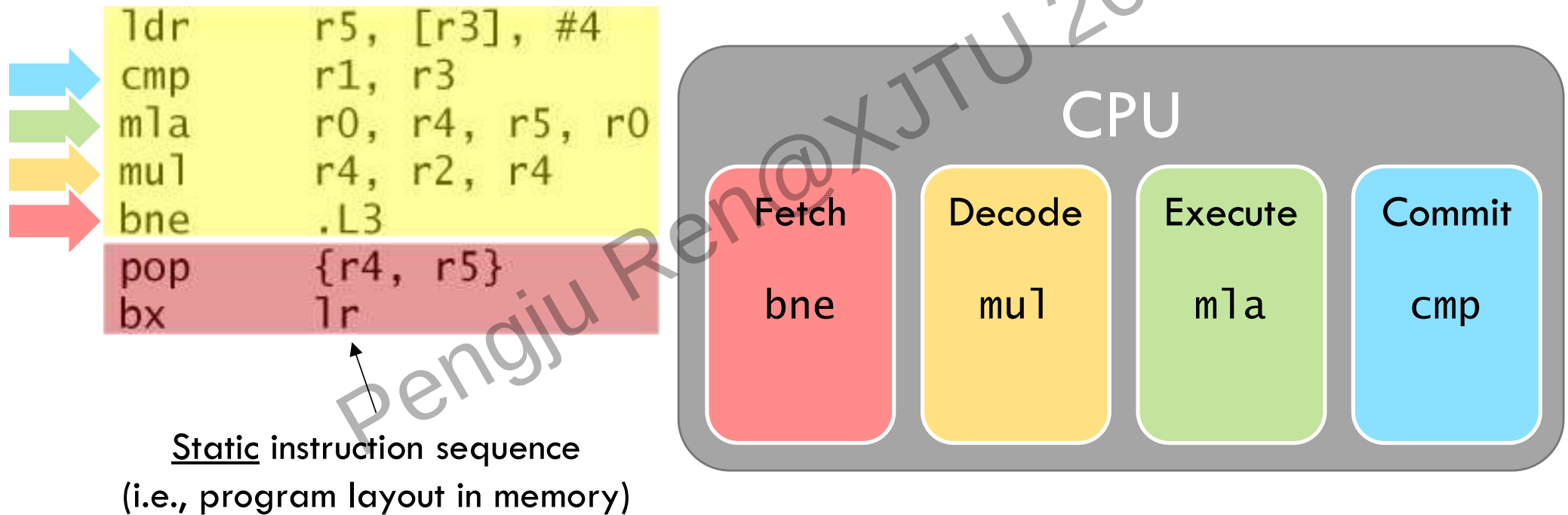


Control hazards + Speculation

- Programs must appear to execute *in program order*
→ All instructions depend on earlier ones
- Most instructions implicitly continue at the next...
- But **branches** redirect execution to new location

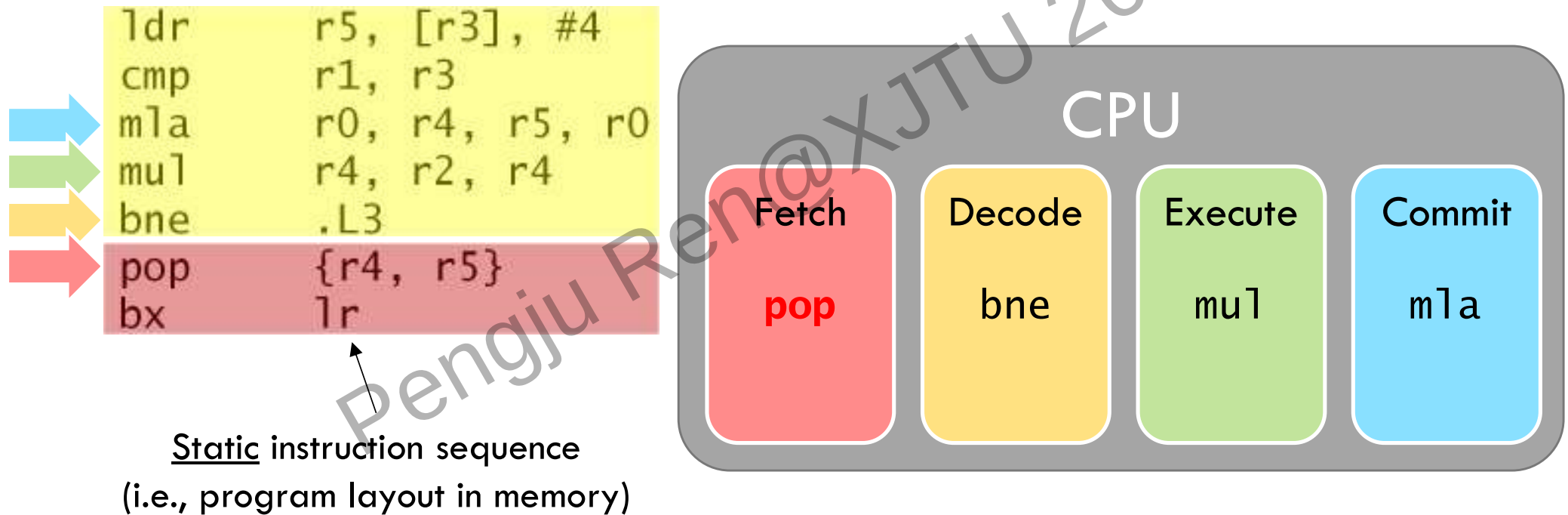
Dealing with control hazards: Flushing the pipeline

- What if we always fetch the next instruction?



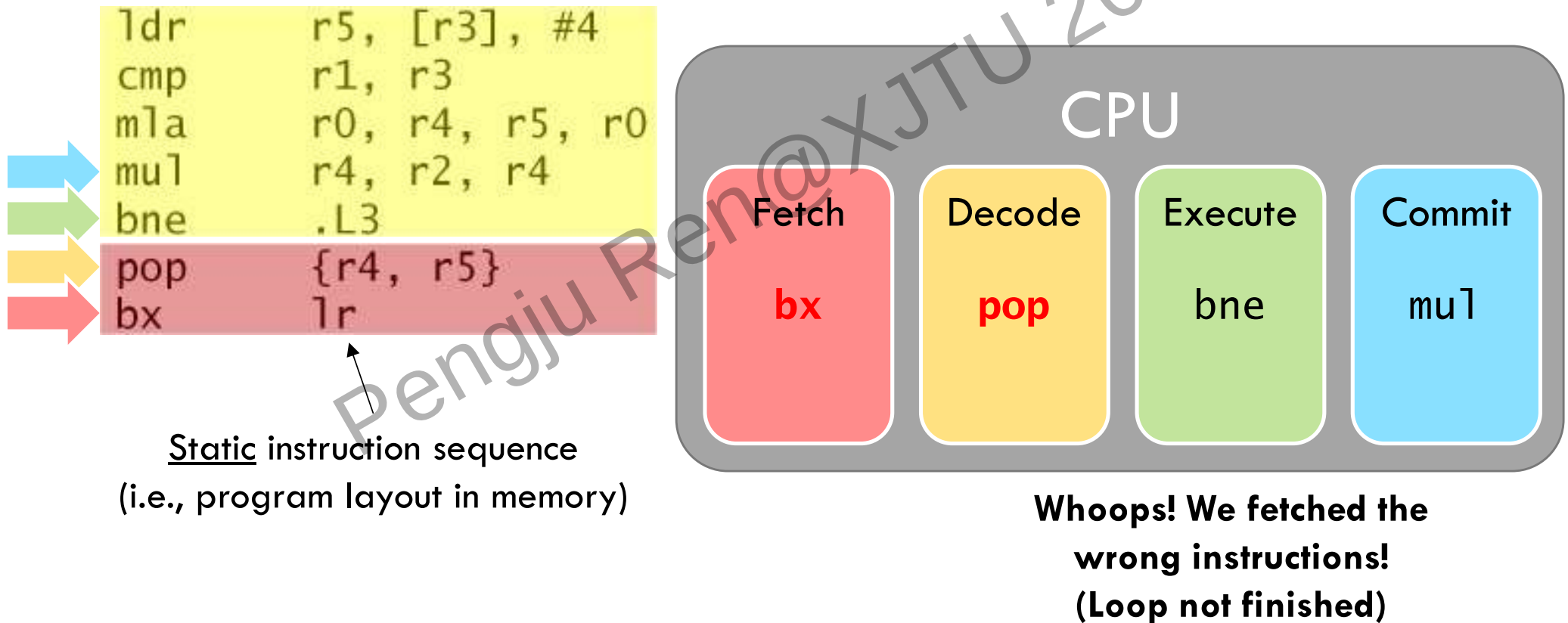
Dealing with control hazards: Flushing the pipeline

- What if we always fetch the next instruction?



Dealing with control hazards: Flushing the pipeline

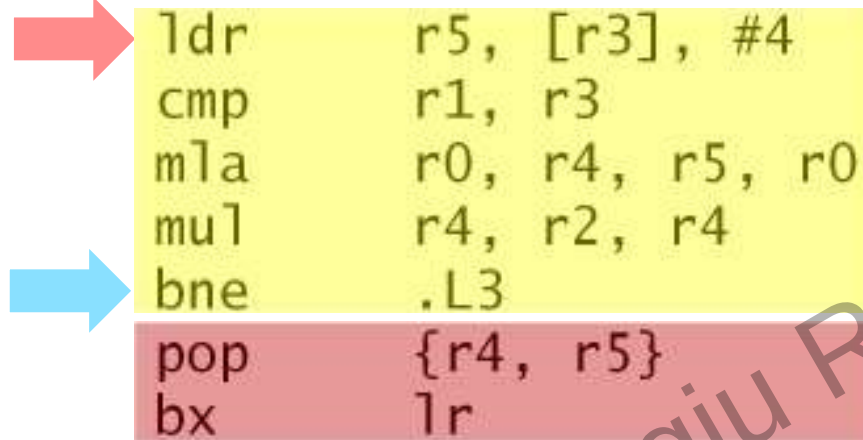
- What if we always fetch the next instruction?



Dealing with control hazards: Flushing the pipeline

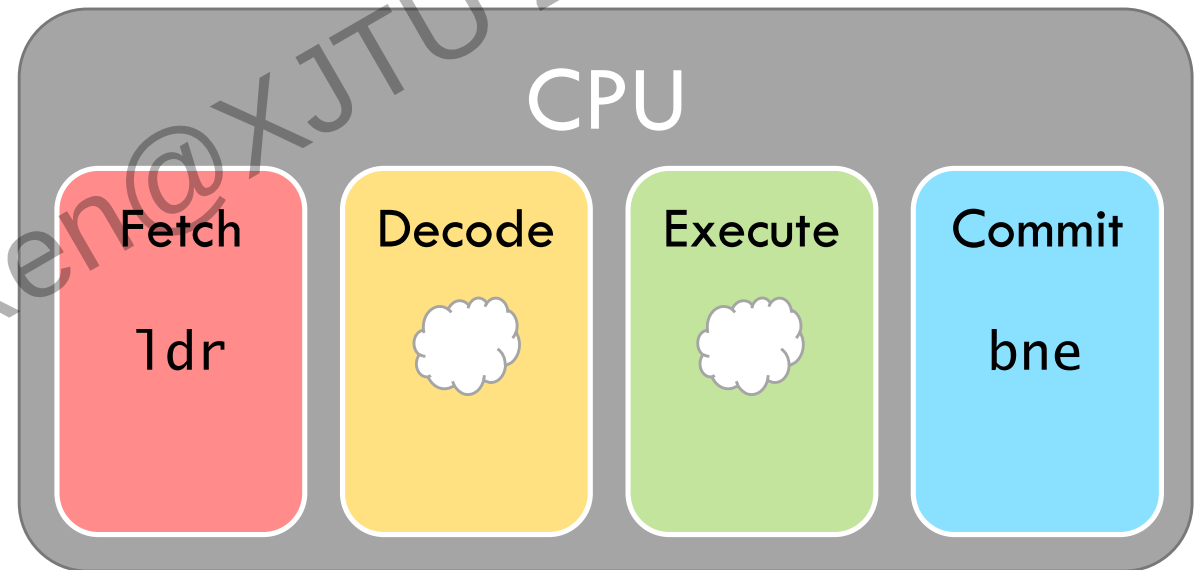
(Next loop iteration)

- What if we always fetch the next instruction?



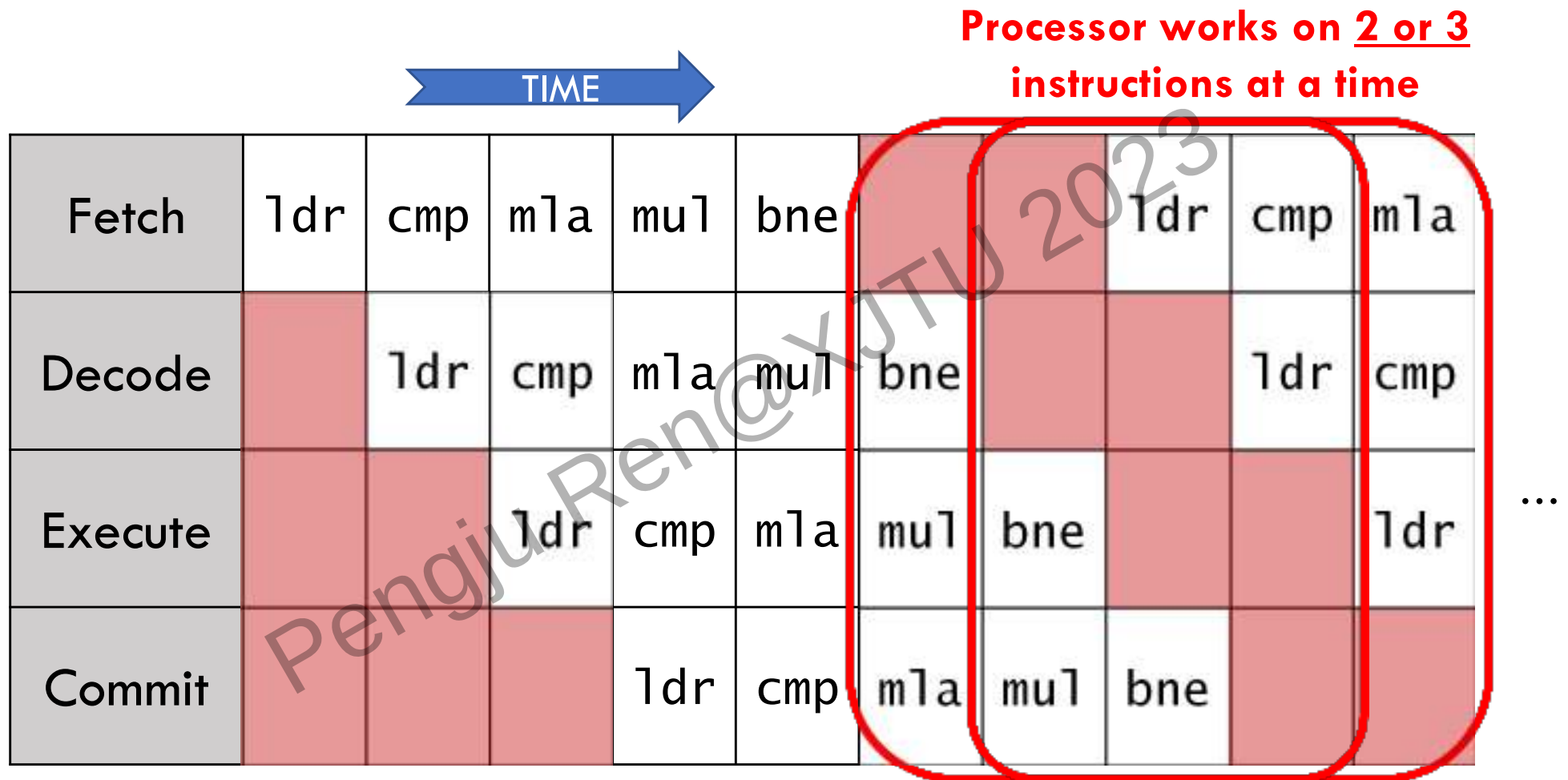
```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
pop    {r4, r5}
bx     lr
```

Static instruction sequence
(i.e., program layout in memory)



Whoops! We fetched the
wrong instructions!
(Loop not finished)

Pipeline flushes destroy performance



- Penalty increases with deeper pipelines

Dealing with control hazards: *Speculation!*

- Processors do not wait for branches to execute
- Instead, they **speculate** (i.e., guess) where to go next + start fetching
- Modern processors use very sophisticated mechanisms
 - E.g., speculate in Fetch stage—before processor even knows instrs is a branch! (*Branch Instrs can be detected by PC*)
 - >95% prediction accuracy
 - Still, branch mis-speculation is major problem (*The wider and deeper the pipeline, the more serious the problem*)

Pipelining Summary

- Pipelining is a simple, effective way to improve throughput
 - N -stage pipeline gives up to $N \times$ speedup
 - Pipelining has limits
 - Hard to keep pipeline busy because of hazards
 - Forwarding is expensive in deep pipelines(critical path)
 - Pipeline flushes are expensive in deep pipelines
- ➔ Pipelining is ubiquitous, but tops out at $N \approx 15$

Software Takeaways

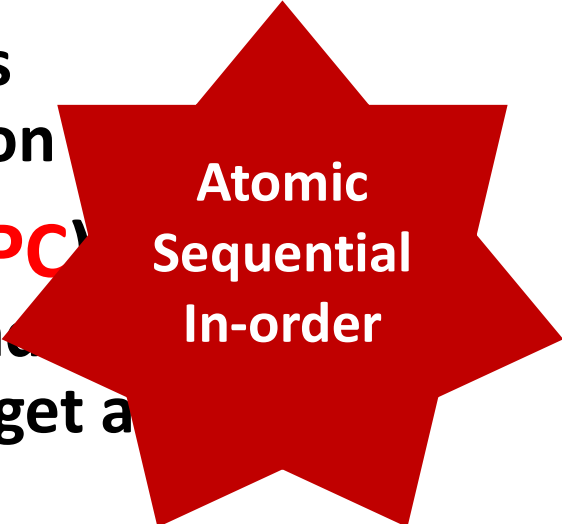
- **Processors with a simple “in-order” pipeline are very sensitive to running “good code”**
 - **Compiler should target a specific model of CPU**
 - **Low-level assembly hacking**
- **...But very few CPUs are in-order these days**
 - **E.g., embedded, ultra-low-power applications**
- **Instead, $\approx$ all modern CPUs are “out-of-order”**
 - **Even in classic “low-power domains” (like mobile)**

Out-of-Order Execution

Pengju Ren@XJTU 2023

Instruction Classes (as convention)

- **Arithmetic and logical operations**
 - compute a result as a function of the operands
 - update PC to the next sequential instruction
- **Data “movement” operations (no compute)**
 - *fetch* operands from specified locations
 - *store* operand values to specified locations
 - update PC to the next sequential instruction
- **Control flow operations (affects only PC)**
 - compute a branch condition and a target address
 - if “branch condition is true” then PC $\leftarrow$ target address
else PC $\leftarrow$ next seq. instruction



Atomic
Sequential
In-order

Superpipelined and SuperScalar Execution

Code1: $r1 \leftarrow r2 + 1$
 $r3 \leftarrow r1 * 2$
 $r4 \leftarrow r0 - r3$

Code2: $r1 \leftarrow r2 + 1$
 $r3 \leftarrow r9 * 2$
 $r4 \leftarrow r0 - r10$

Code1 : **ILP=1** i.e., must execute serially

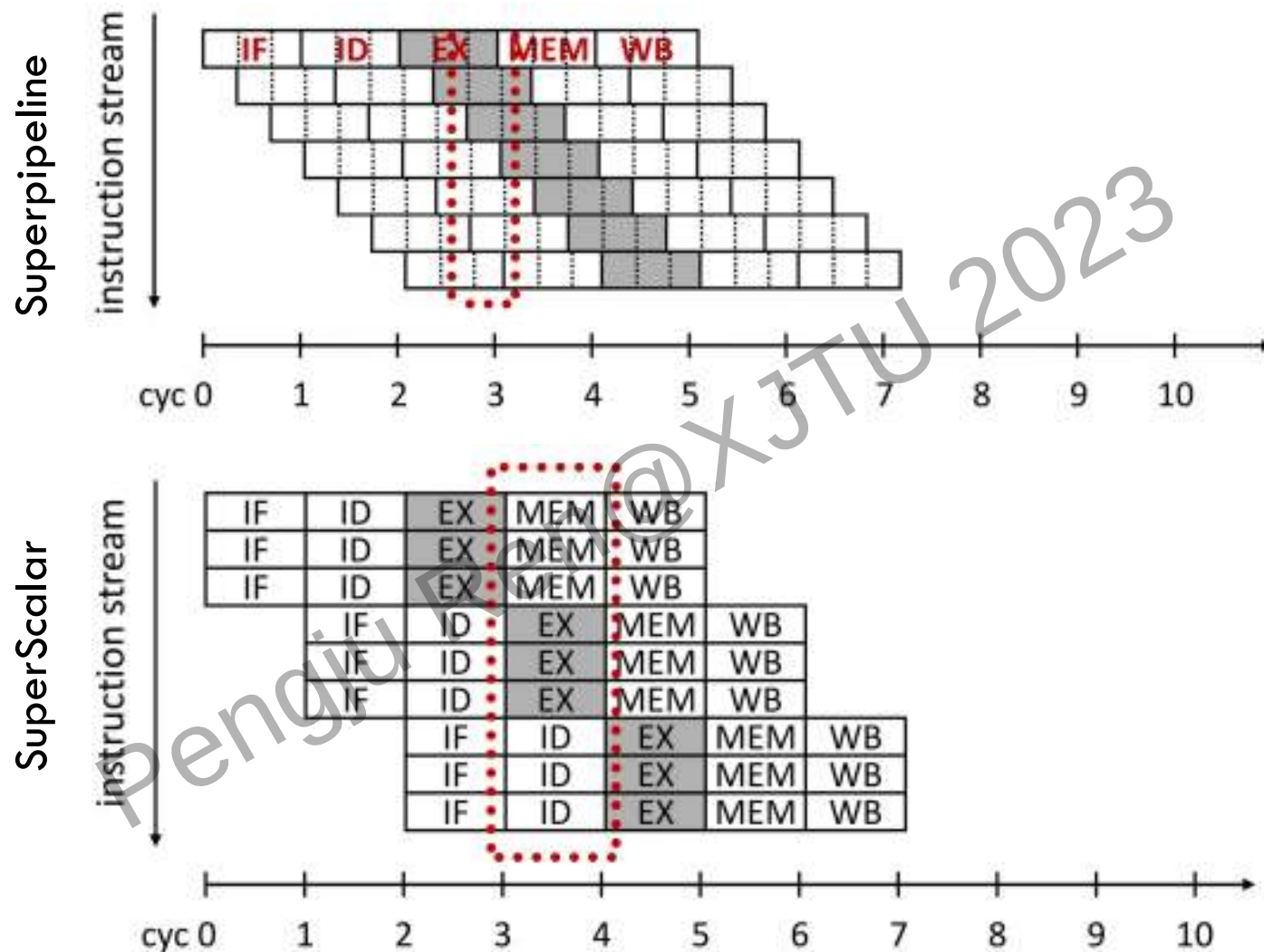
Code2 : **ILP=3** i.e., can execute at the same time

ILP=2 [Code3: $r1 \leftarrow r2 + 1$
 $r3 \leftarrow r1 * 2$
 $r4 \leftarrow r0 - r3$] **ILP=1**
 $r11 \leftarrow r12 + 1$
 $r13 \leftarrow r19 * 2$
 $r14 \leftarrow r0 - r20$

Accessing ILP=2 requires:

(1) larger scheduling window and (2) out-of-order execution

Superpipelined and SuperScalar Execution



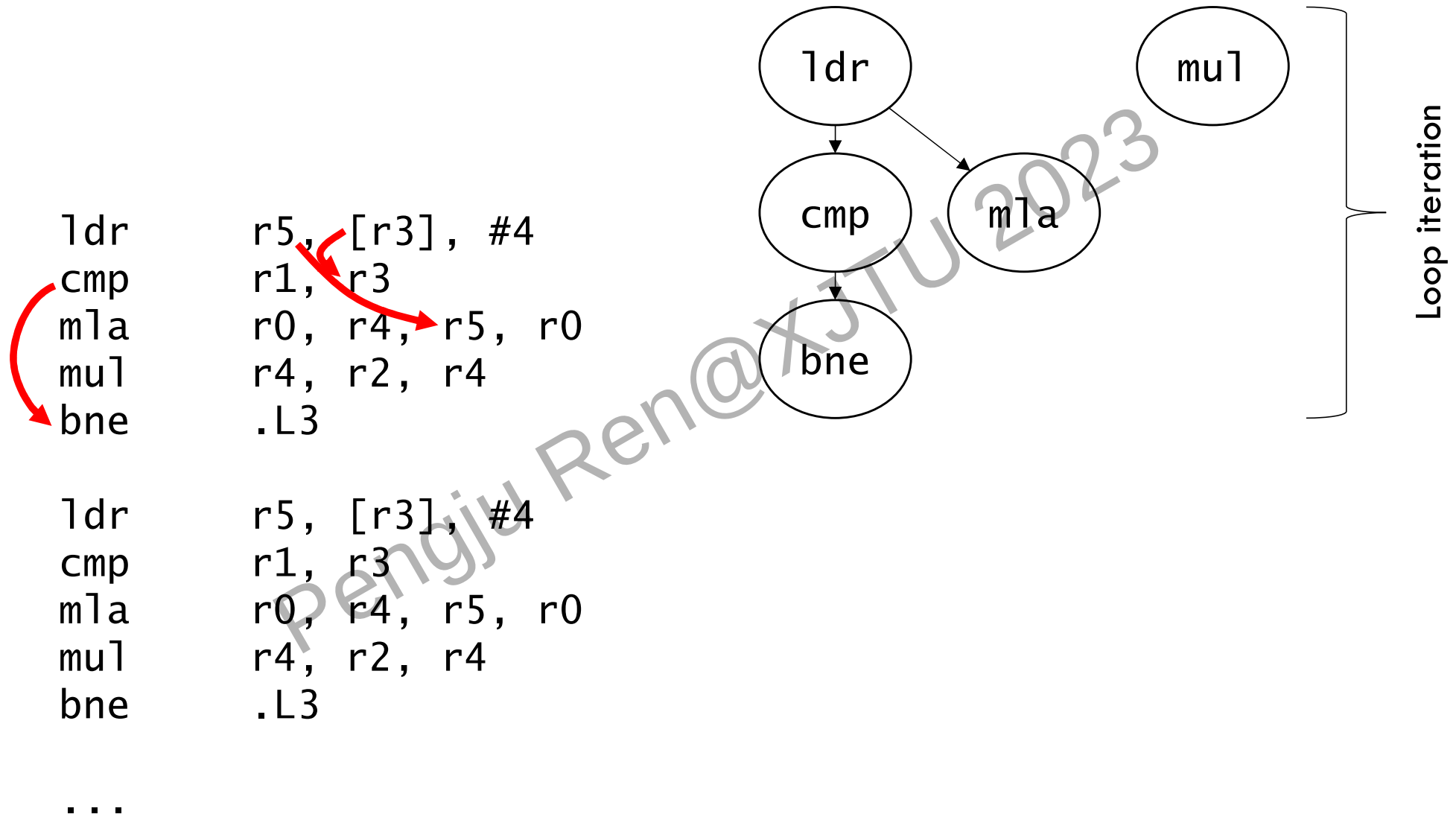
Achieving full performance requires finding **N** “independent” instructions on every cycle

Increasing parallelism via dataflow

- Parallelism limited by many *false dependencies*, particularly *sequential program order*
- Dataflow tracks how instructions actually depend on each other
 - *True dependence*: **read-after-write**
 - *False dependence*: **write-after-write, write-after-read**

Dataflow increases parallelism by eliminating unnecessary dependences

Example: Dataflow in polynomial evaluation

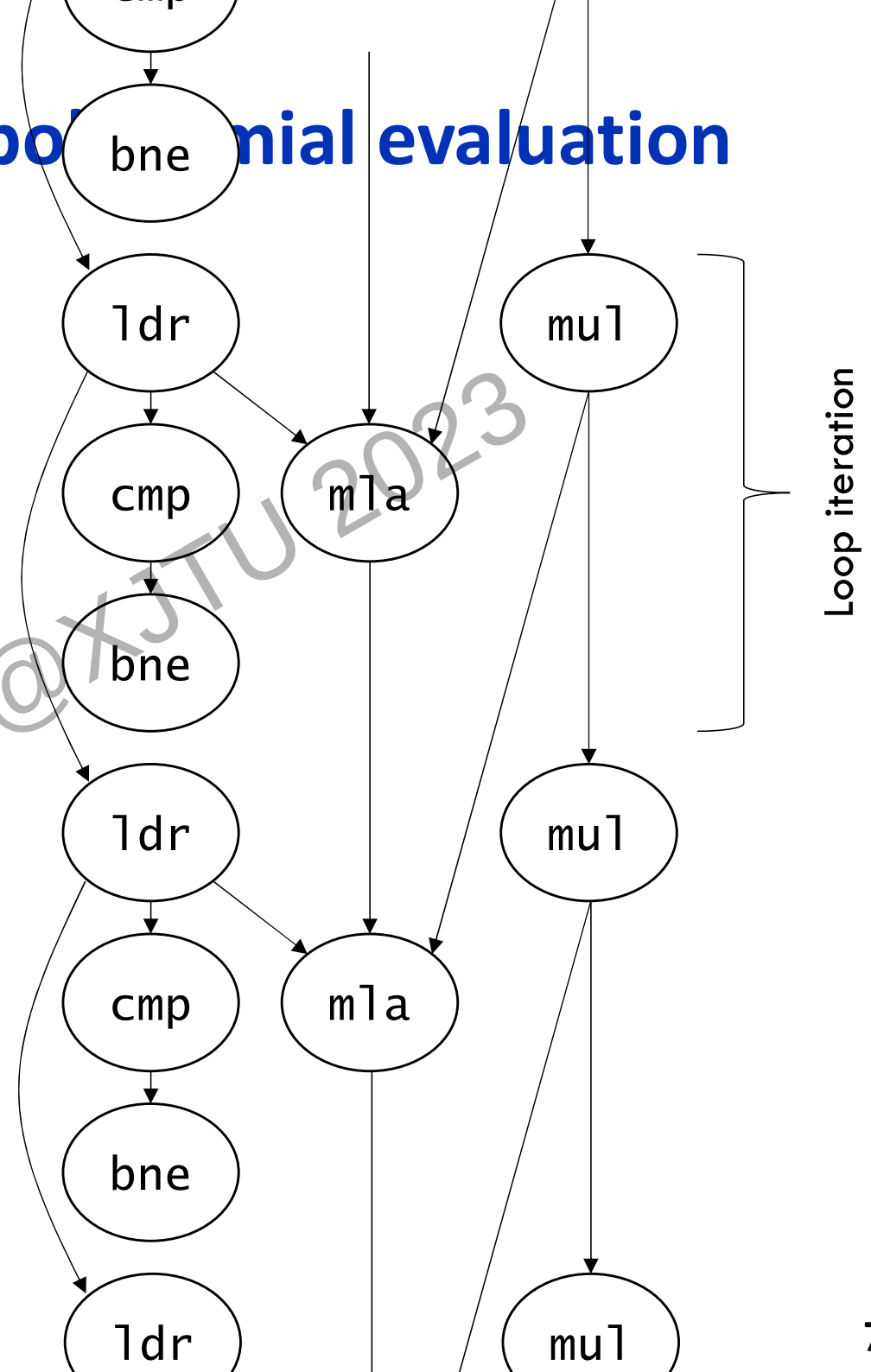


Example: Dataflow in polynomial evaluation

```
ldr    r5, [r3], #4  
cmp    r1, r3  
mla    r0, r4, r5, r0  
mul    r4, r2, r4  
bne    .L3
```

```
ldr    r5, [r3], #4  
cmp    r1, r3  
mla    r0, r4, r5, r0  
mul    r4, r2, r4  
bne    .L3
```

...



Example: Dataflow polynomial execution

- Execution only, with perfect scheduling & unlimited execution units
 - ldr, mul execute in 2 cycles
 - cmp, bne execute in 1 cycle
 - mla executes in 3 cycles
- Q: Does dataflow speedup execution? By how much?
- Q: What is the performance bottleneck?



1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

ldr

Pengju Ren@XJTU 2023

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
```



1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

ldr



cmp

Pengju Ren@XJTU 2023

```
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
```



1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

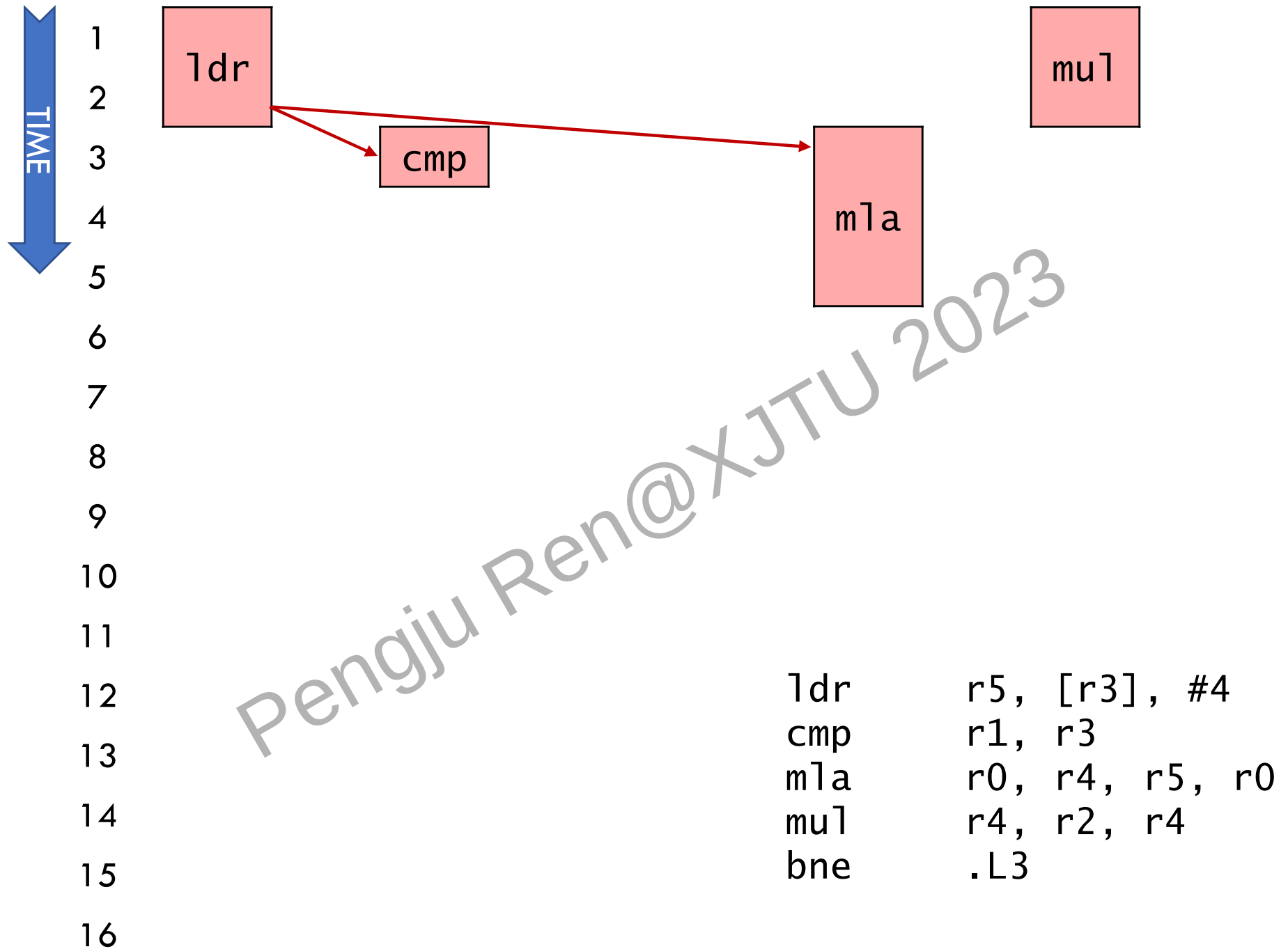
ldr

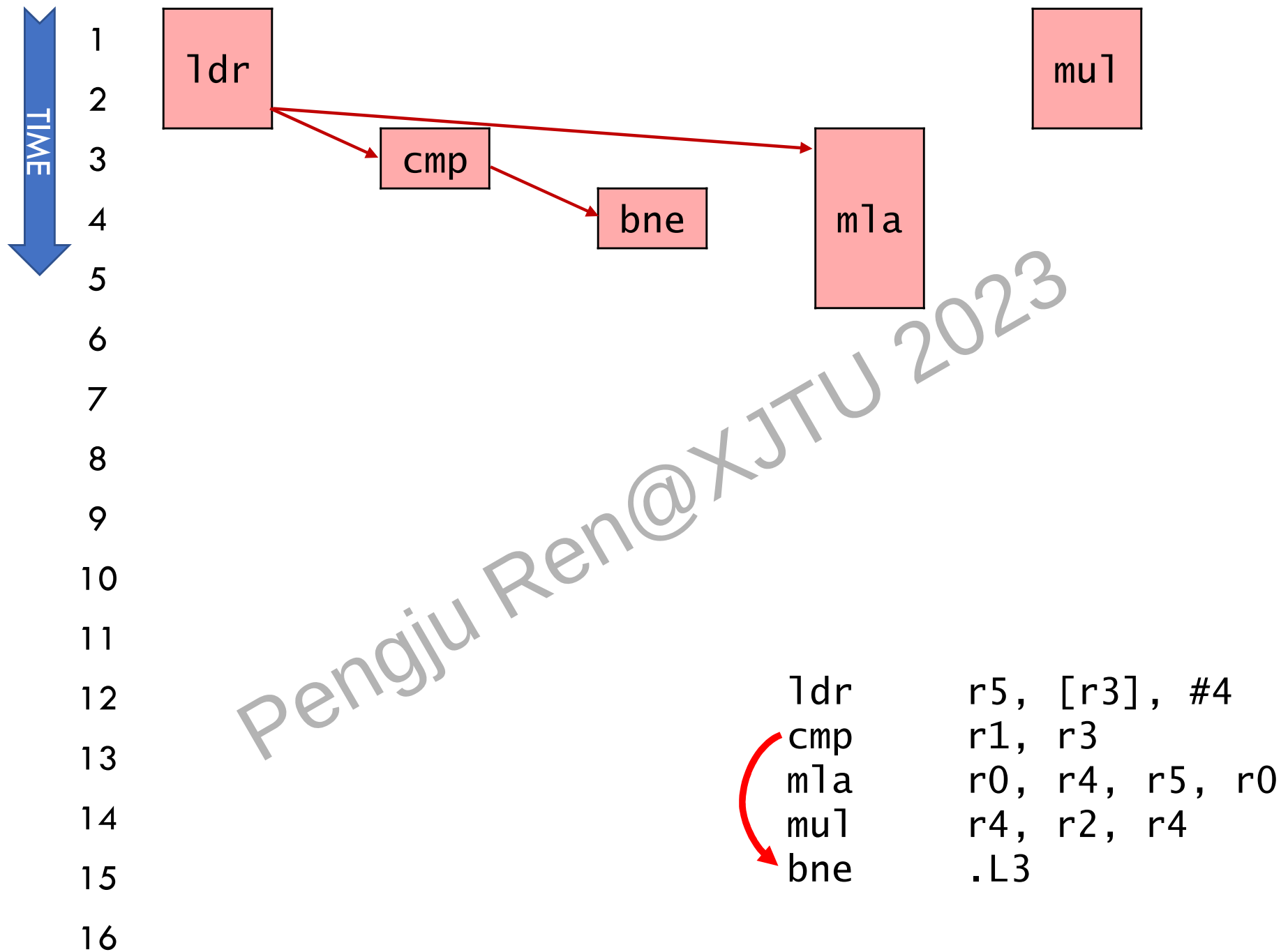
cmp

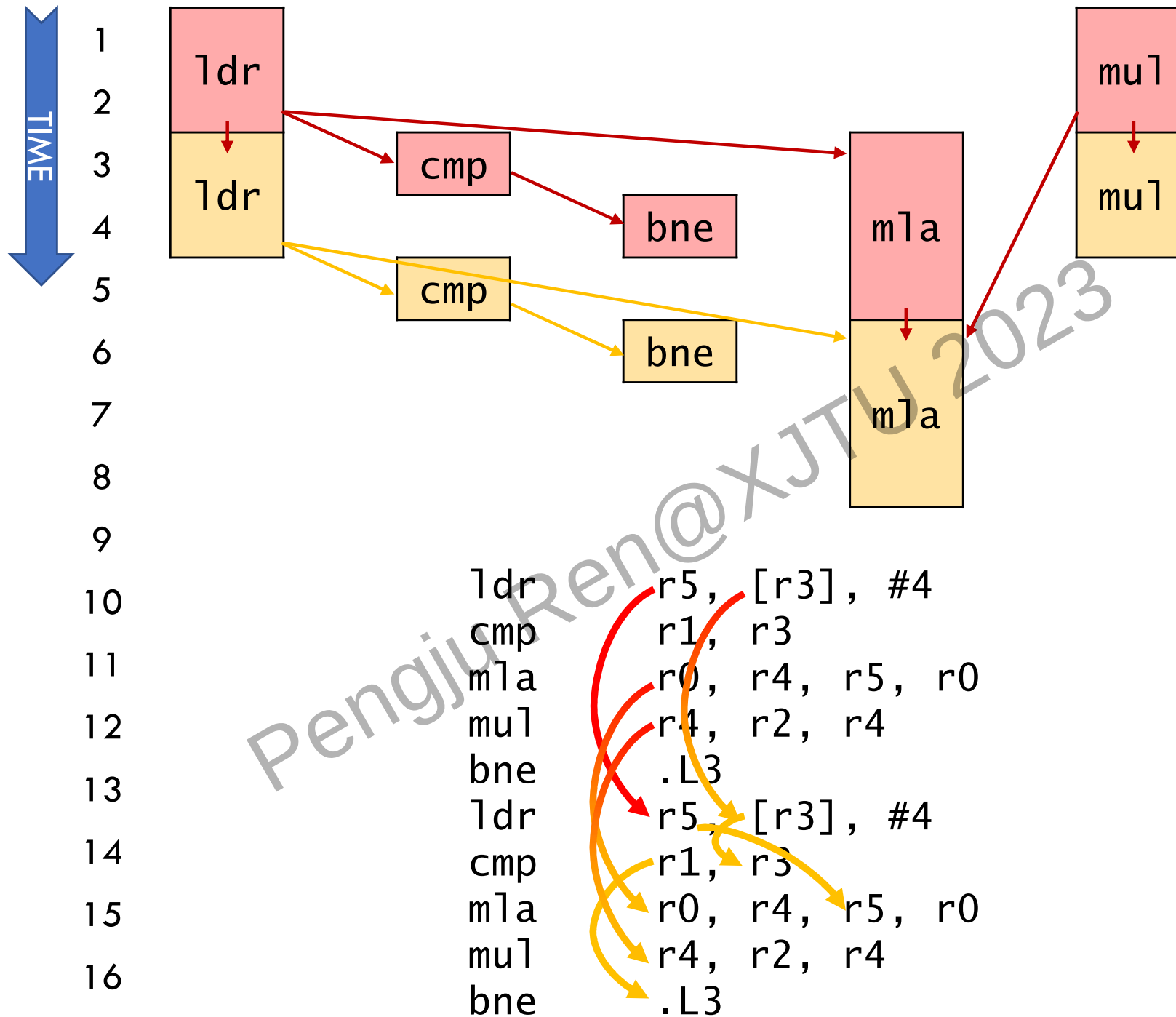
m1a

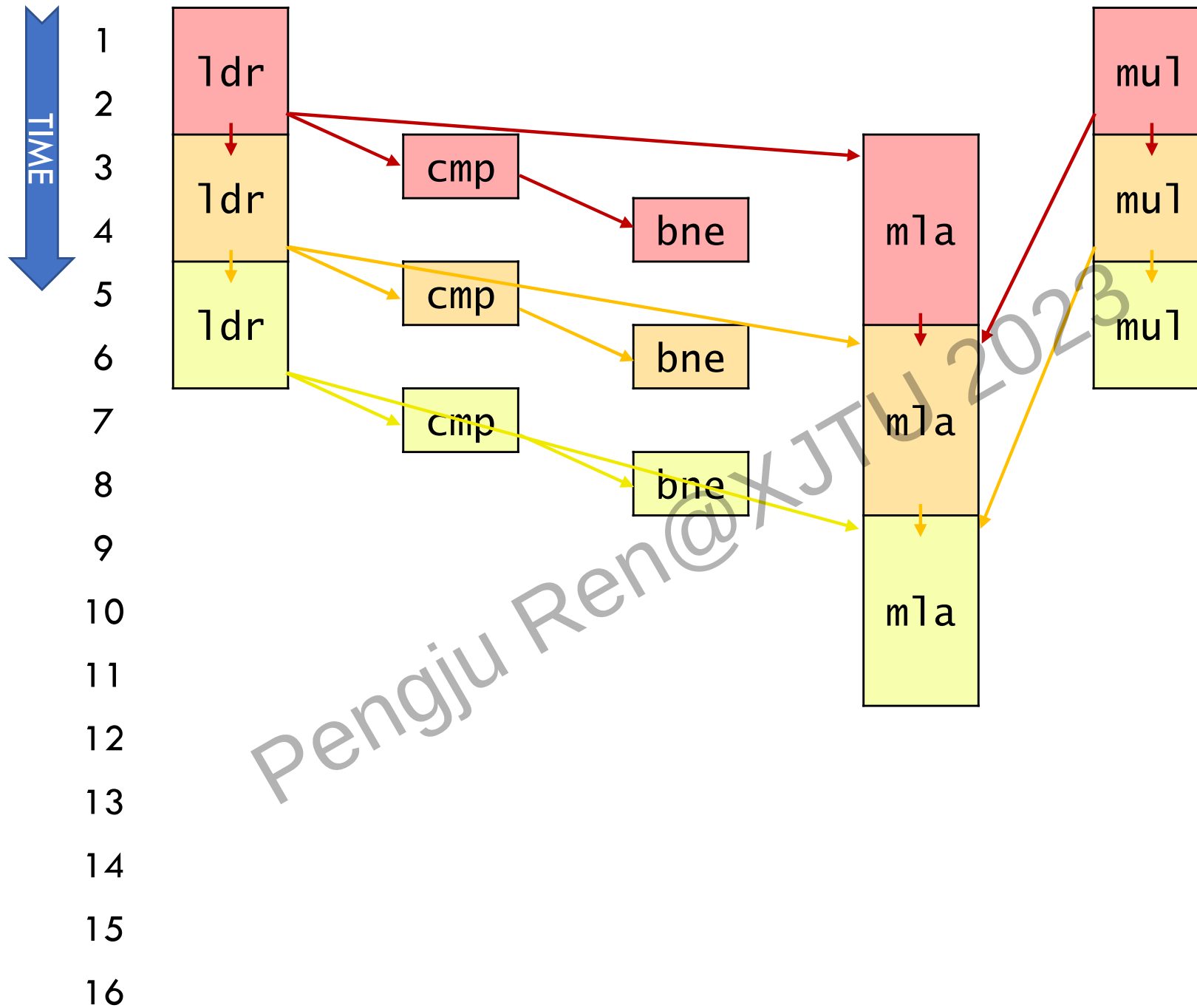
Pengju Ren@XJTU 2023

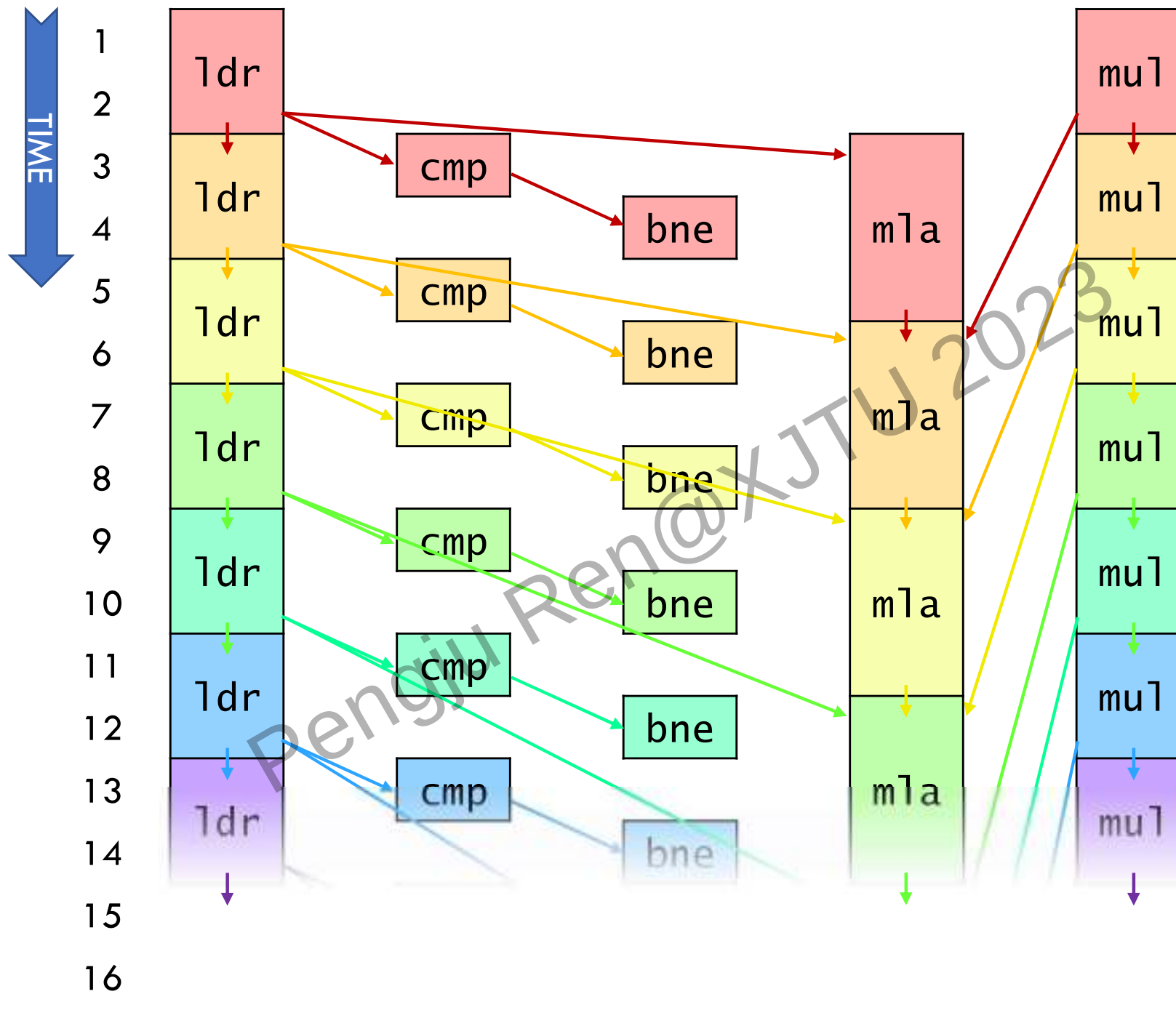
ldr	r5, [r3], #4
cmp	r1, r3
m1a	r0, r4, r5, r0
mul	r4, r2, r4
bne	.L3











Example: Dataflow polynomial execution

- **Q: Does dataflow speedup execution? By how much?**
 - Yes! 3 cycles / loop iteration
 - Instructions per cycle (IPC) = $5/3 \approx 1.67$
(vs. 1 for perfect pipelining)
- **Q: What is the performance bottleneck?**
 - m1a: Each m1a depends on previous m1a & takes 3 cycles
 - ➔ This program is **latency-bound**

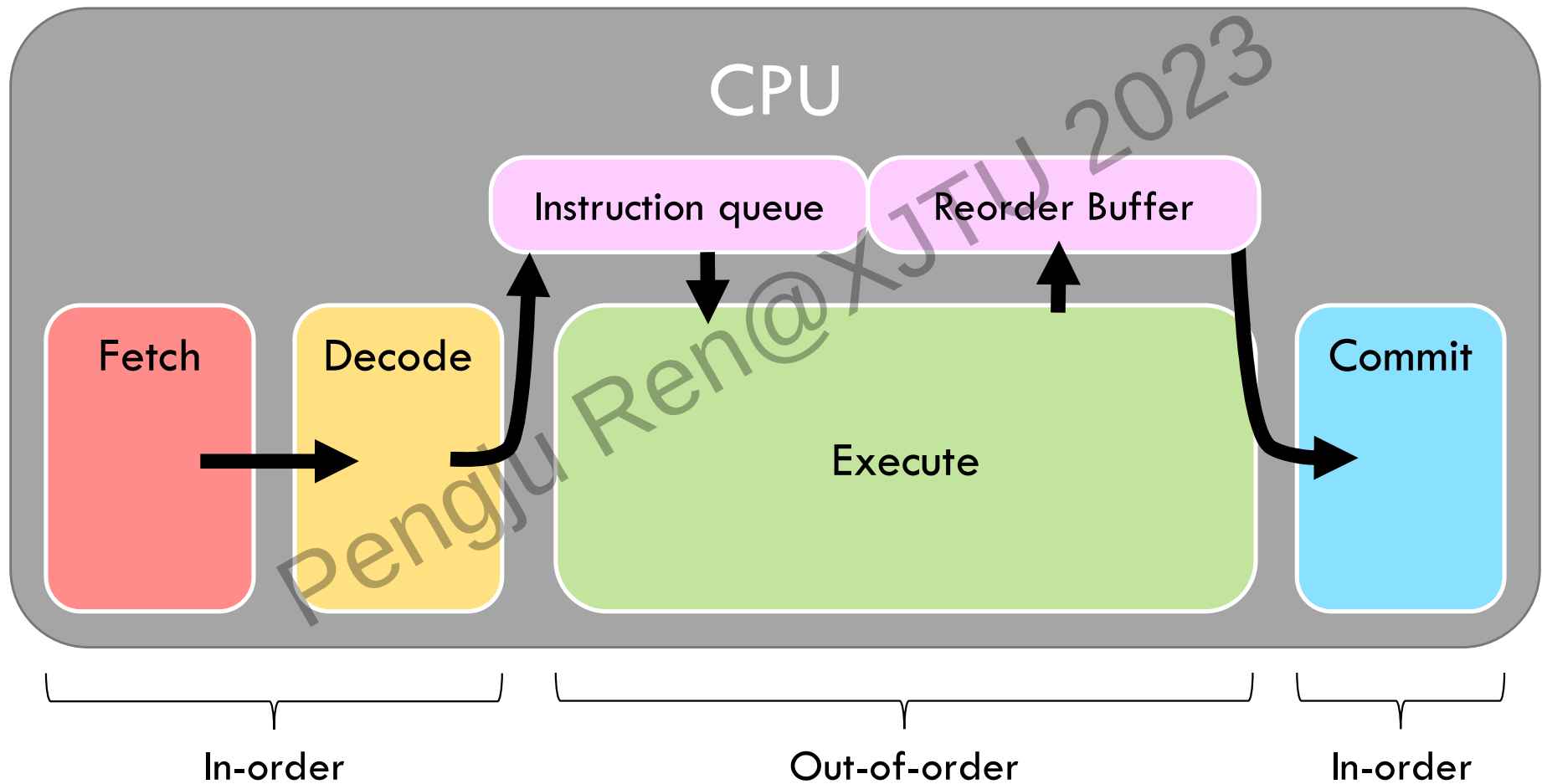
Latency Bound

- What is the “**critical path**” of the computation?
 - Longest path across iterations in dataflow graph
 - E.g., `m1a` in last slide (but could be multiple ops)
- Critical path limits maximum performance
- Real CPUs may not achieve latency bound, but useful mental model + tool for program analysis

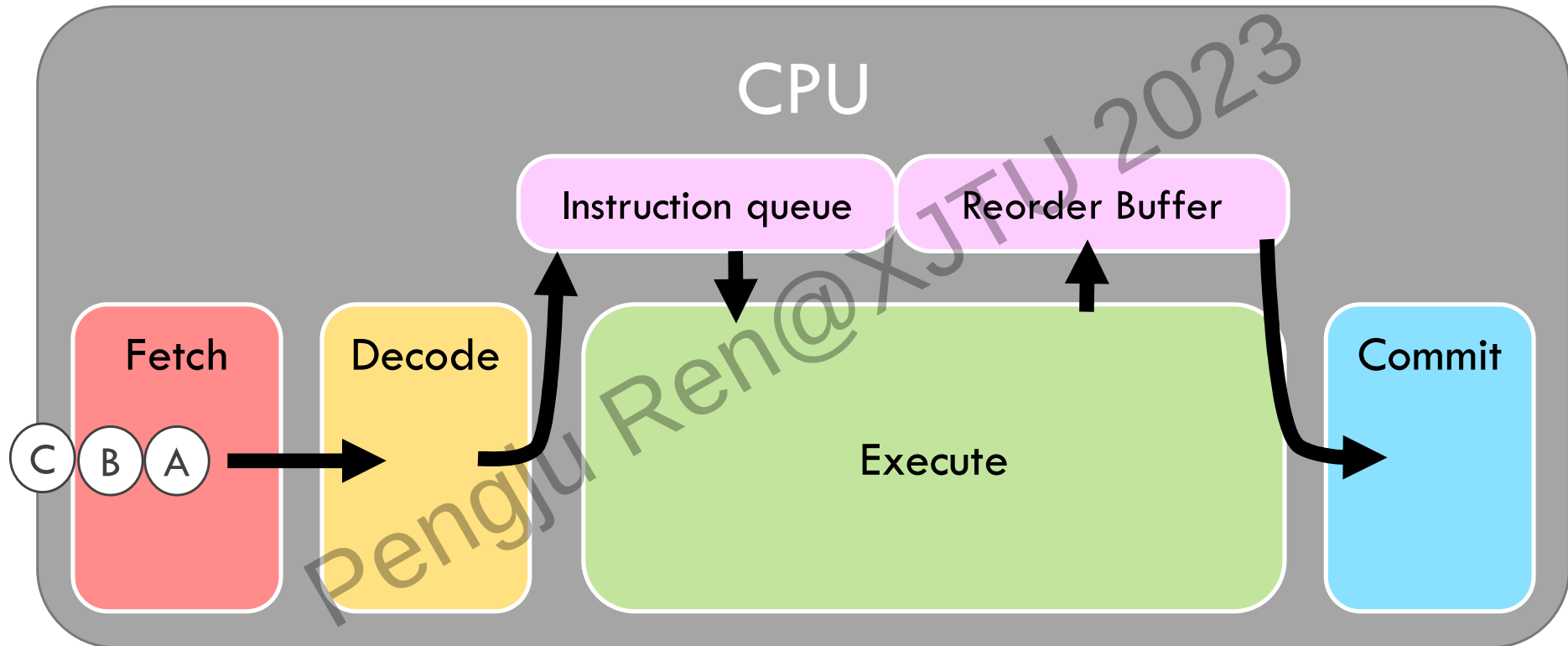
Out-of-order (OoO) execution uses dataflow to increase parallelism

- Idea: Execute programs in dataflow order, but give the *illusion* of sequential execution
- This is a “restricted dataflow” model
 - *Restricted* to instructions near those currently committing
 - (Pure dataflow processors also exist that expose dataflow to software)

High-level OoO microarchitecture

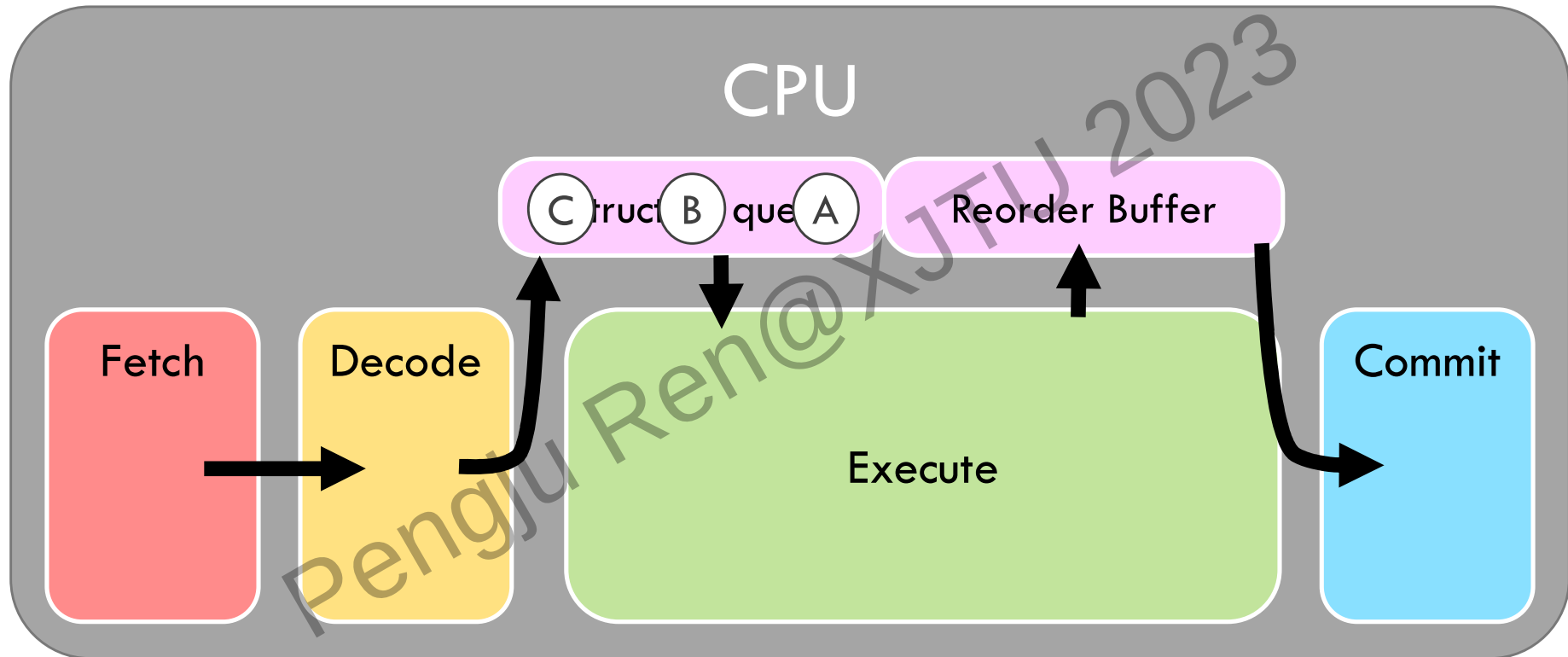


OoO is hidden behind in-order frontend & commit



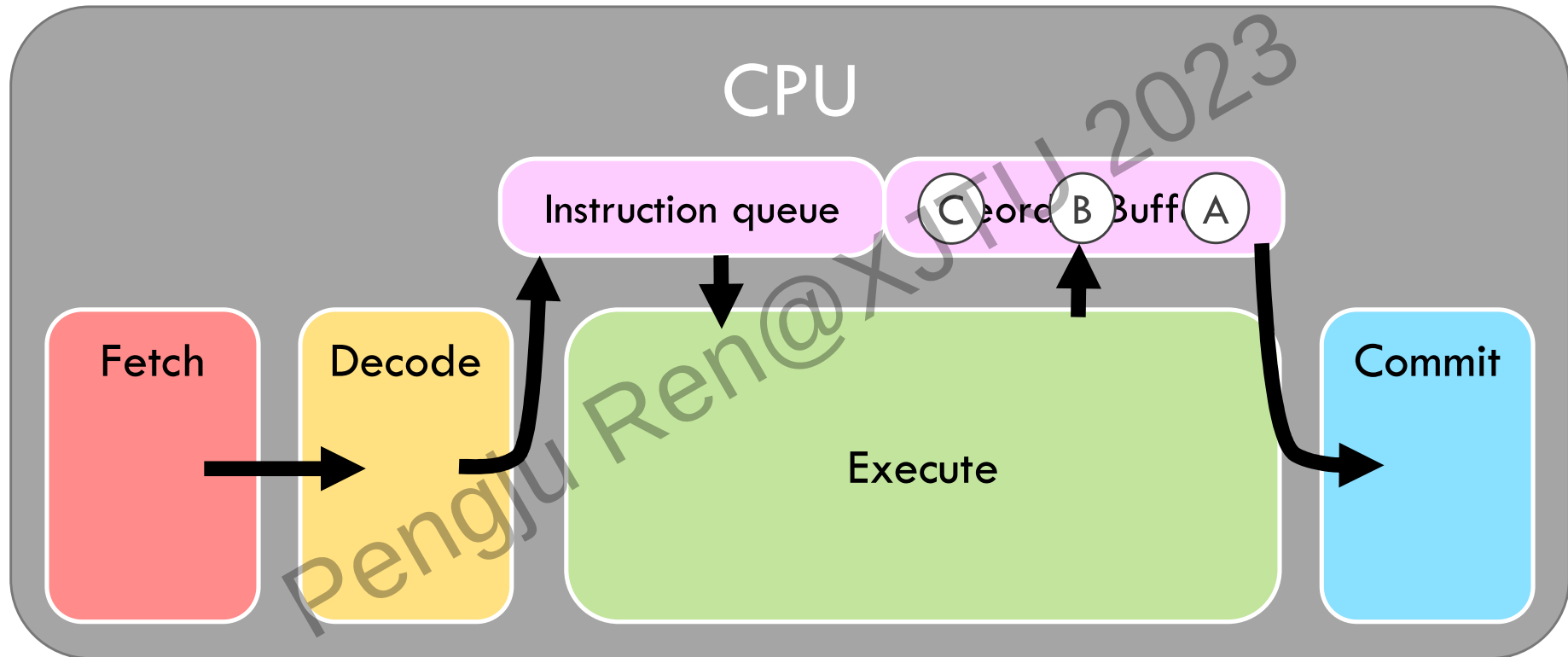
- Instructions only **enter instruction queue(IQ) and leave reorder buffer(ROB) in program order**; all bets are off in between!

OoO is hidden behind in-order frontend & commit



- Instructions only **enter instruction queue(IQ) and leave reorder buffer(ROB) in program order**; all bets are off in between!

OoO is hidden behind in-order frontend & commit



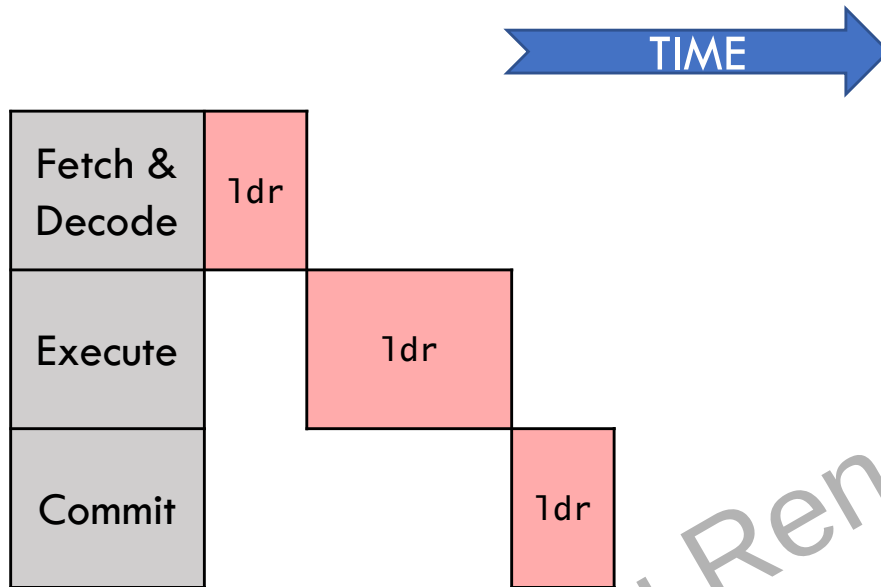
- Instructions only **enter instruction queue(IQ)** and **leave reorder buffer(ROB) in program order**; all bets are off in between!

Example: OoO polynomial evaluation

- Q: Does OoO speedup execution? By how much?
- Q: What is the performance bottleneck?
- Assume perfect forwarding & branch prediction

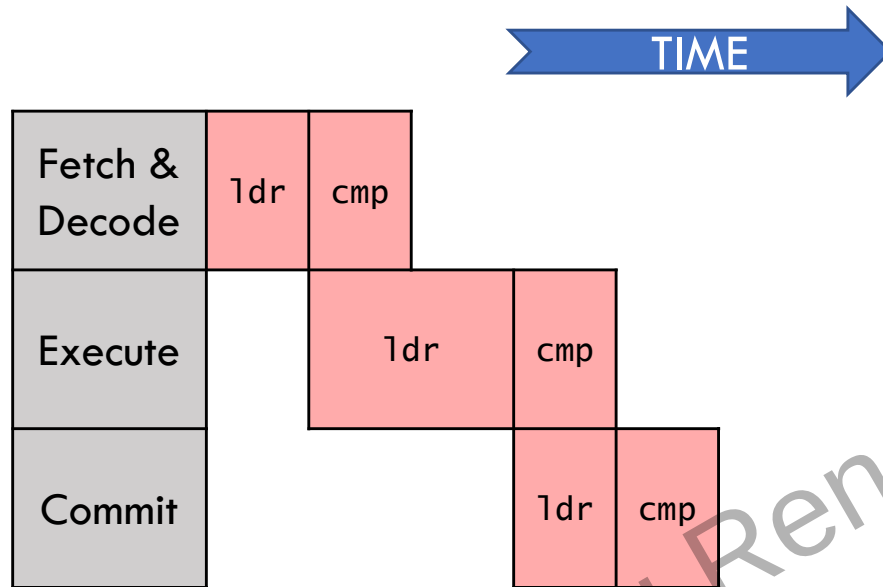
Example: OoO polynomial evaluation pipeline diagram

ldr, mul execute in 2 cycles
cmp, bne execute in 1 cycle
mla executes in 3 cycles



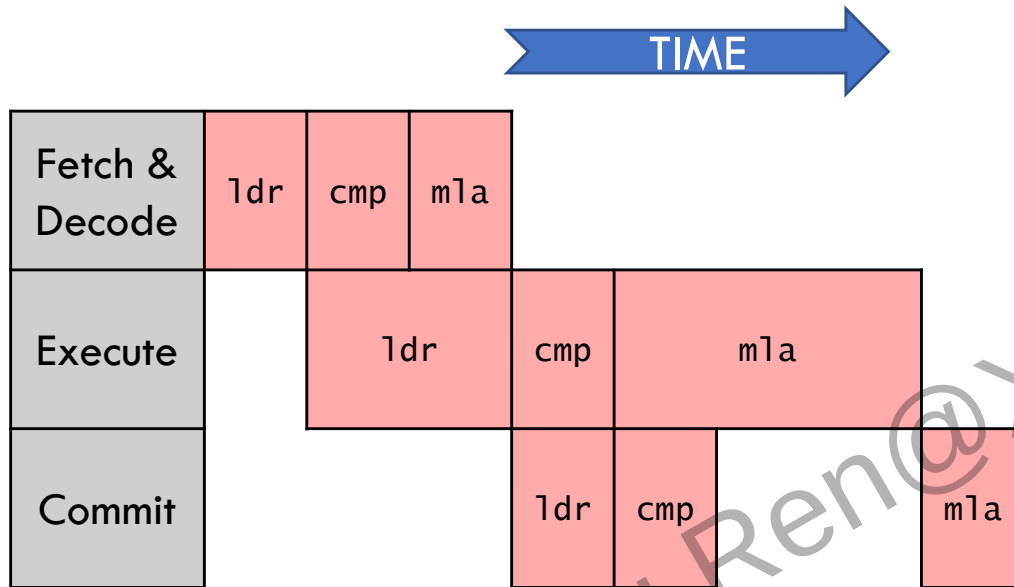
Example: OoO polynomial evaluation pipeline diagram

ldr, mul execute in 2 cycles
cmp, bne execute in 1 cycle
mla executes in 3 cycles



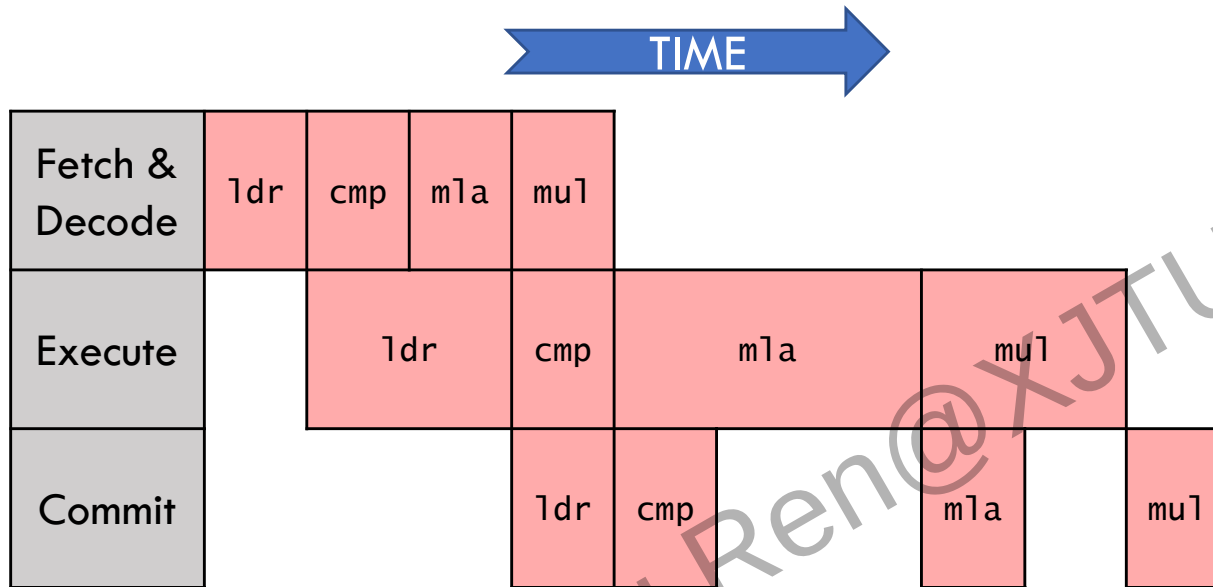
Example: OoO polynomial evaluation pipeline diagram

ldr, mul execute in 2 cycles
cmp, bne execute in 1 cycle
mla executes in 3 cycles



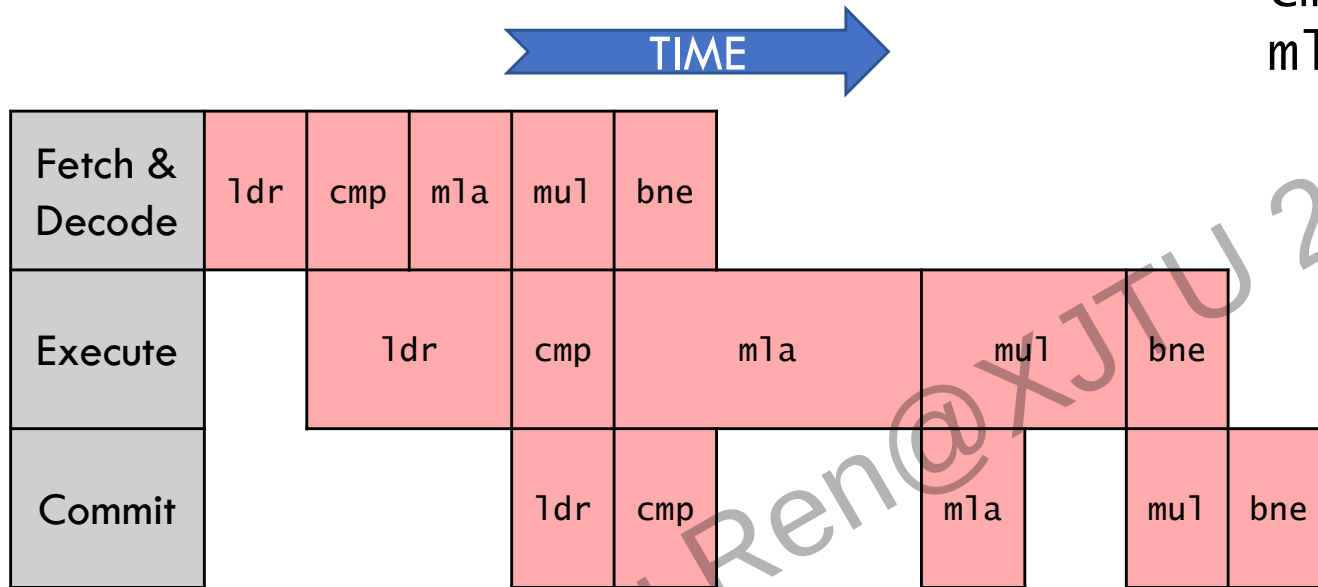
Example: OoO polynomial evaluation pipeline diagram

ldr, mul execute in 2 cycles
cmp, bne execute in 1 cycle
mla executes in 3 cycles



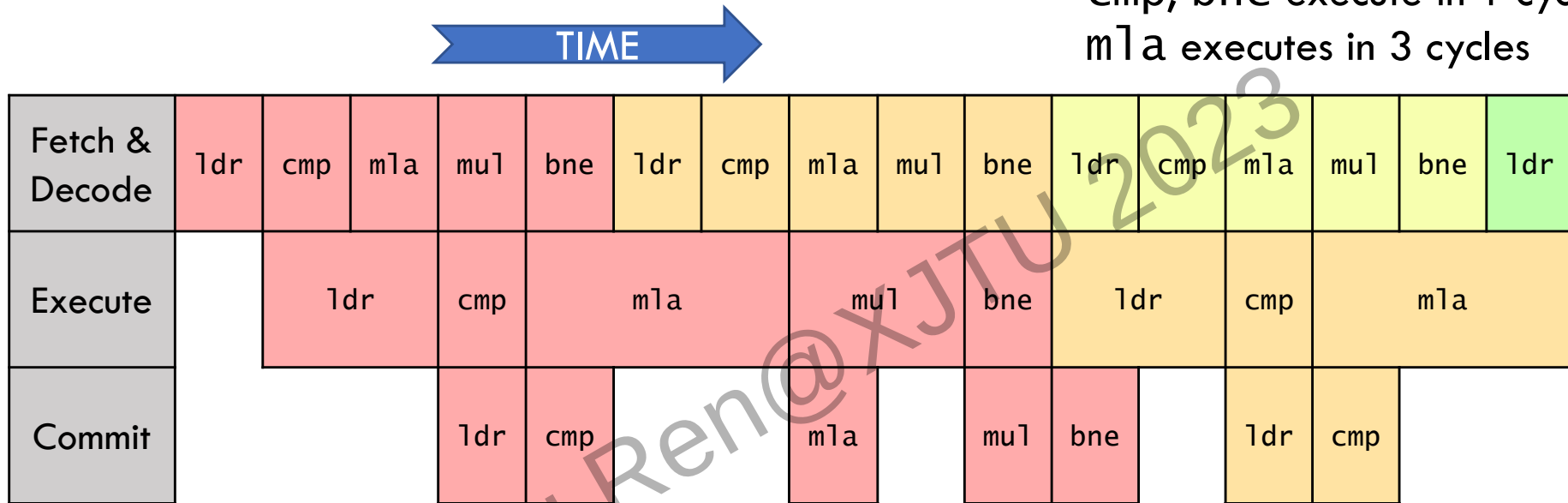
Example: OoO polynomial evaluation pipeline diagram

ldr, mul execute in 2 cycles
cmp, bne execute in 1 cycle
mla executes in 3 cycles



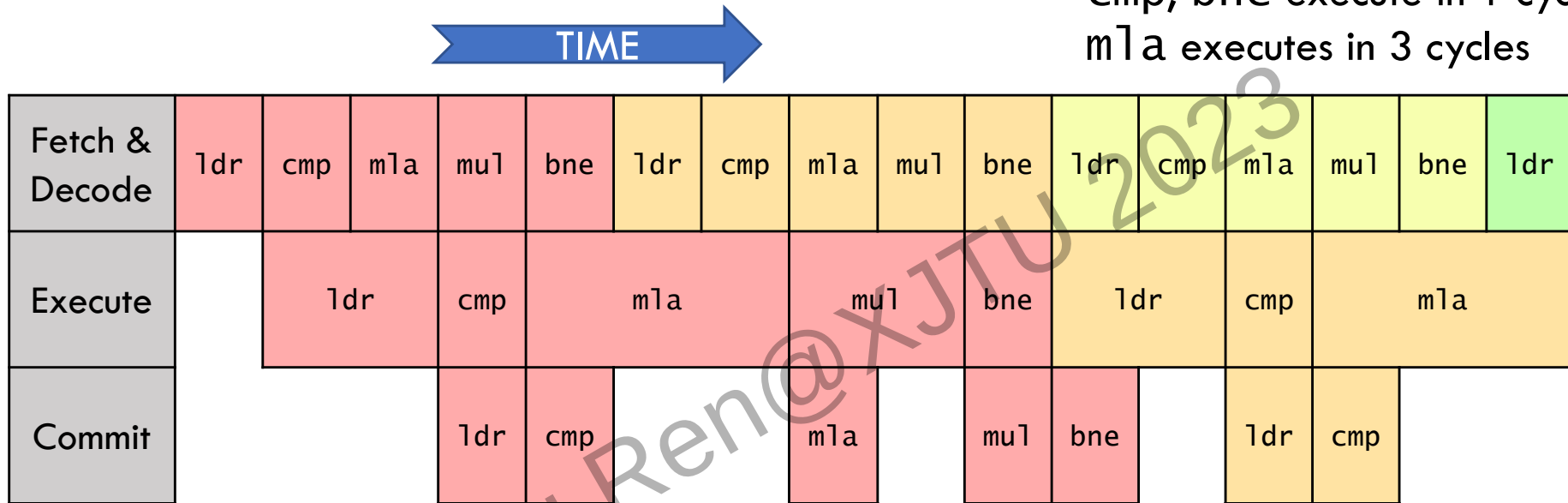
Example: OoO polynomial evaluation pipeline diagram

ldr, mul execute in 2 cycles
 cmp, bne execute in 1 cycle
 mla executes in 3 cycles



Example: OoO polynomial evaluation pipeline diagram

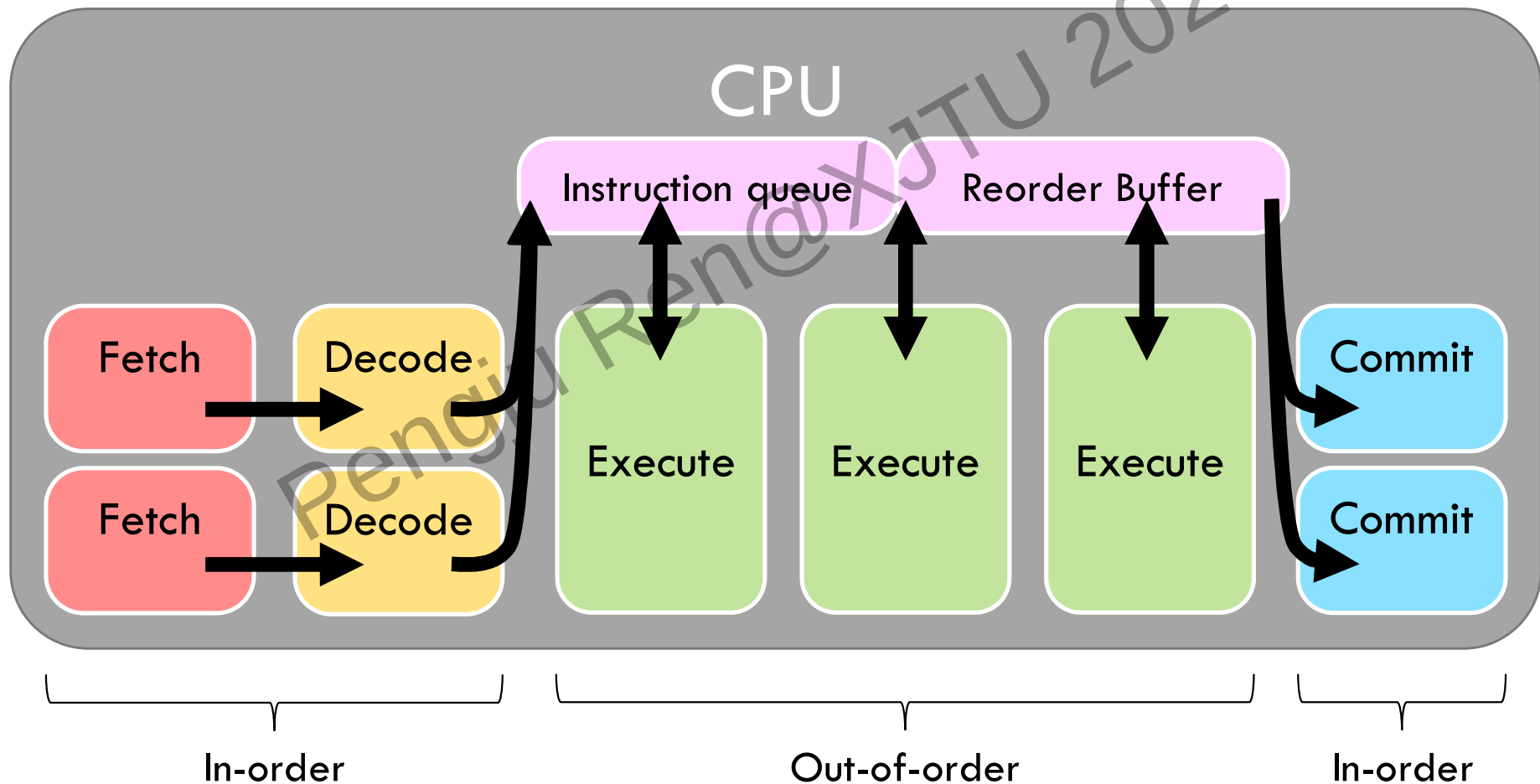
ldr, mul execute in 2 cycles
cmp, bne execute in 1 cycle
mla executes in 3 cycles



- This isn't OoO... or even faster than a simple pipeline!
- Q: What went wrong?
- A: We're throughput-limited: can only exec 1 instrn

High-level Superscalar OoO microarchitecture

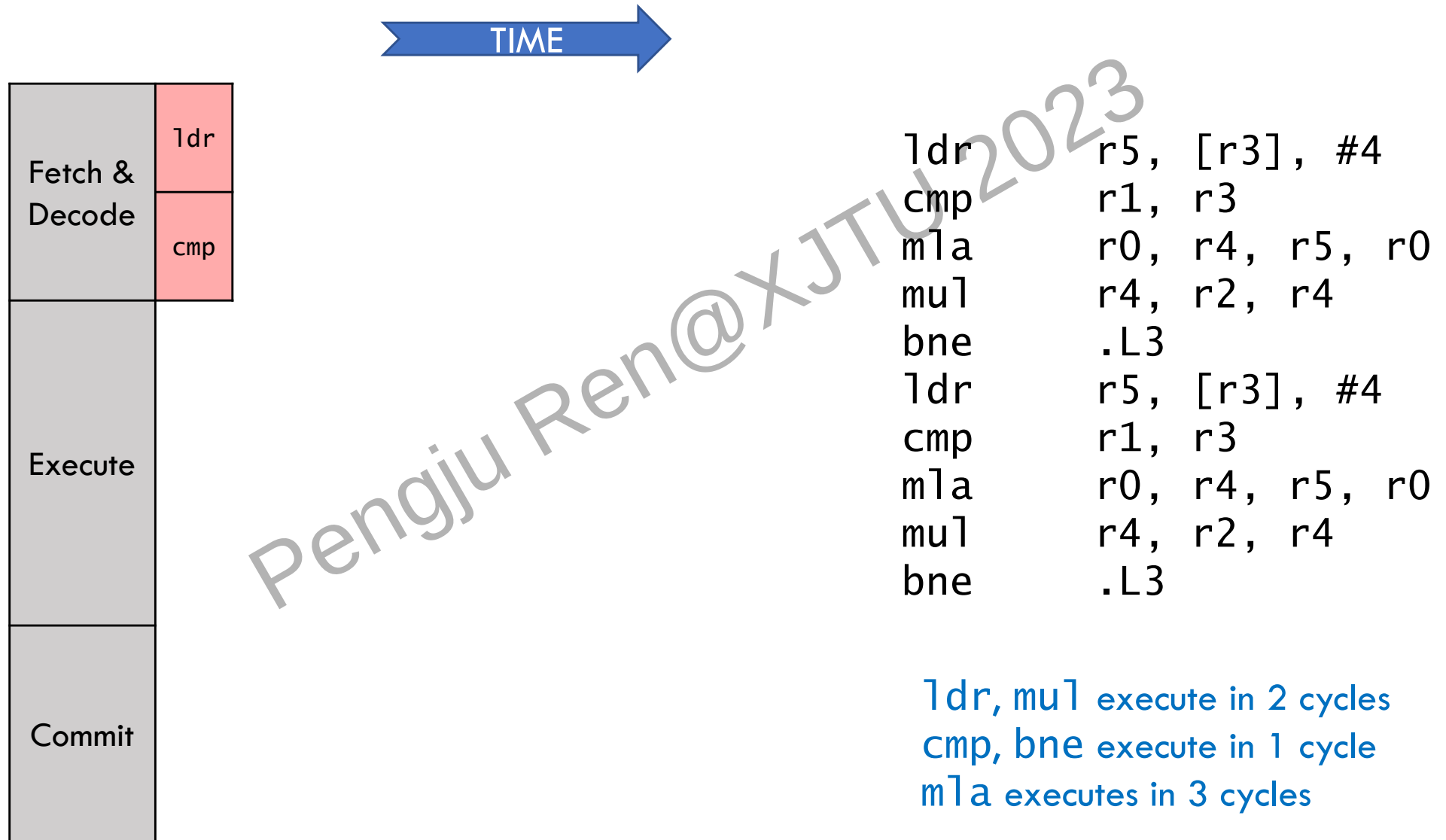
- Must increase *pipeline width* to increase $ILP > 1$ (2-way 3-issue)



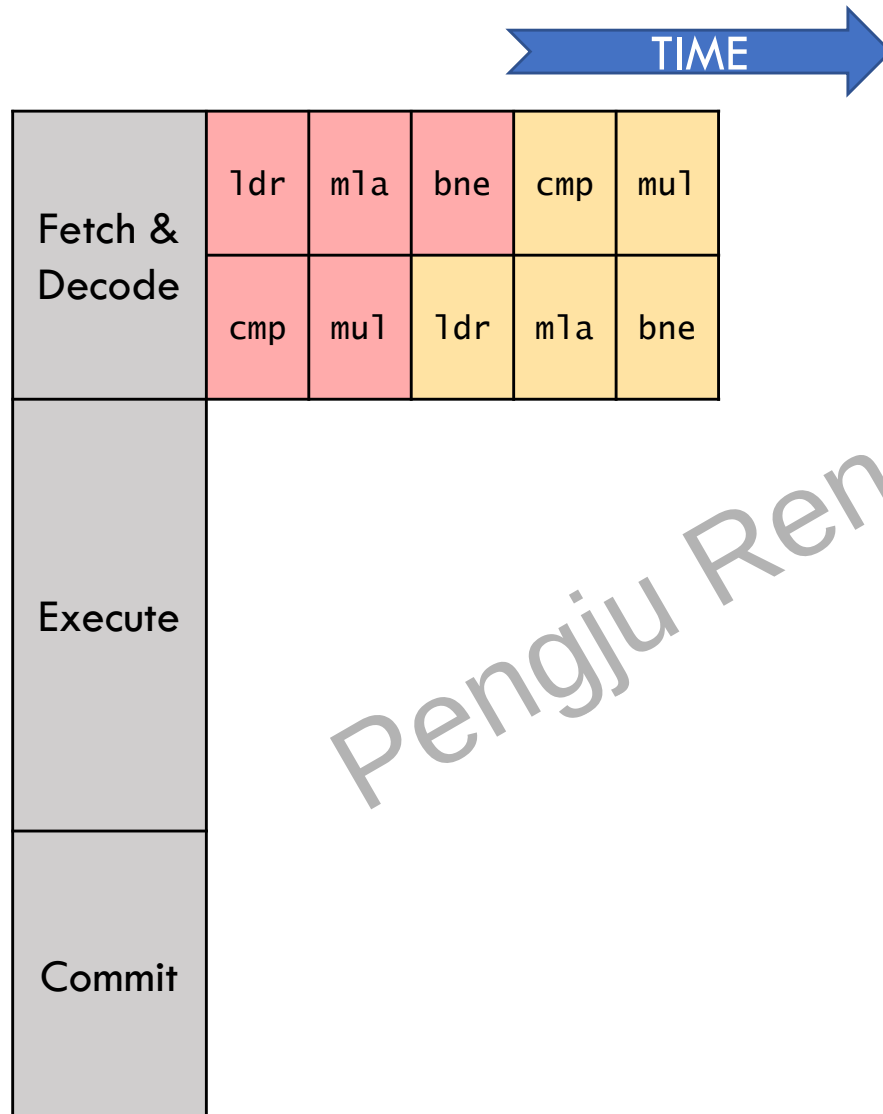
Focus on Execution, not Fetch & Commit

- Goal of OoO design is to only be limited by dataflow execution
- Fetch and commit are *over-provisioned* so that they (usually) do not limit performance
 - ➔ Programmers can (usually) ignore fetch/commit
- NOTES: Programs with *inherently unpredictable* control flow will often be limited by fetch stalls (*branch misprediction*)
 - E.g., branching based on random data

Example: Superscalar OoO polynomial evaluation



Example: Superscalar OoO polynomial evaluation

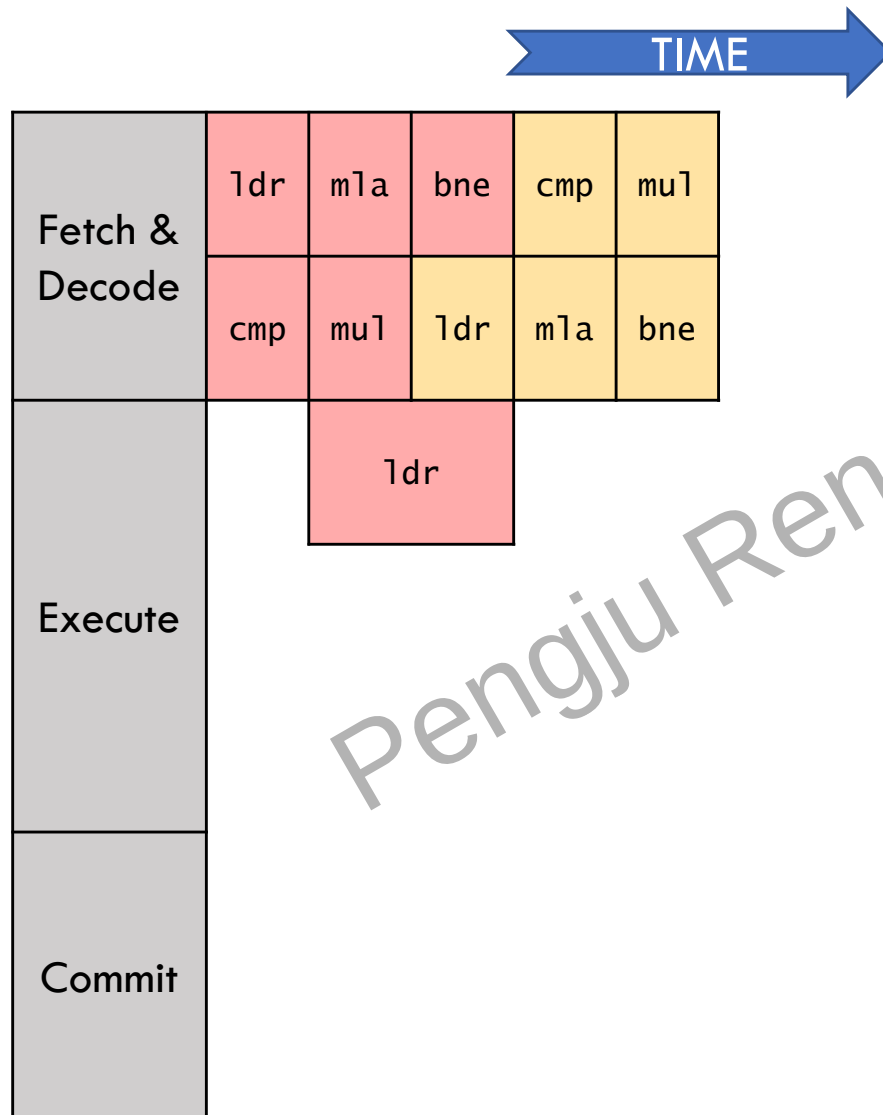


```

ldr    r5, [r3], #4
cmp    r1, r3
m1a    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
ldr    r5, [r3], #4
cmp    r1, r3
m1a    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
    
```

ldr, mul execute in 2 cycles
 cmp, bne execute in 1 cycle
 m1a executes in 3 cycles

Example: Superscalar OoO polynomial evaluation



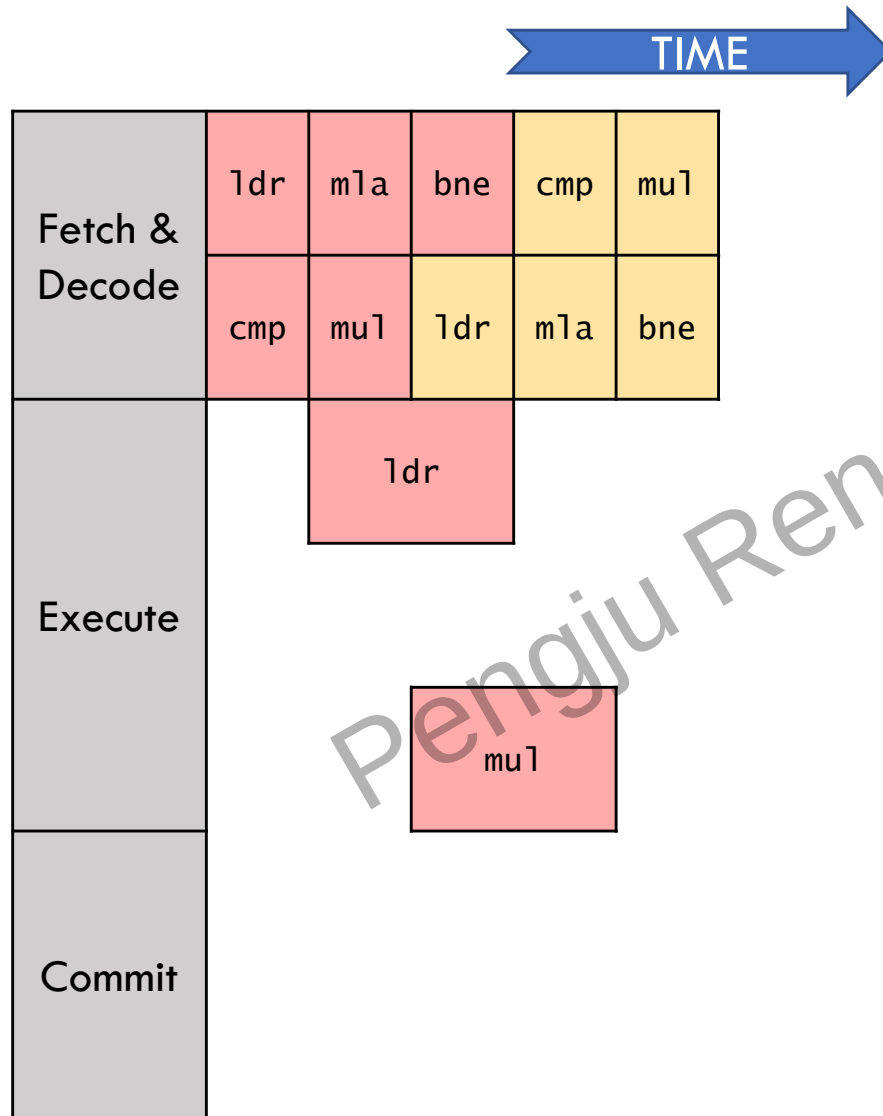
```

ldr    r5, [r3], #4
cmp    r1, r3
m1a    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
ldr    r5, [r3], #4
cmp    r1, r3
m1a    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
    
```

Red arrows indicate data dependencies between instructions in the second block, showing that the second block's instructions are dependent on the first block's instructions.

ldr, mul execute in 2 cycles
 cmp, bne execute in 1 cycle
 m1a executes in 3 cycles

Example: Superscalar OoO polynomial evaluation

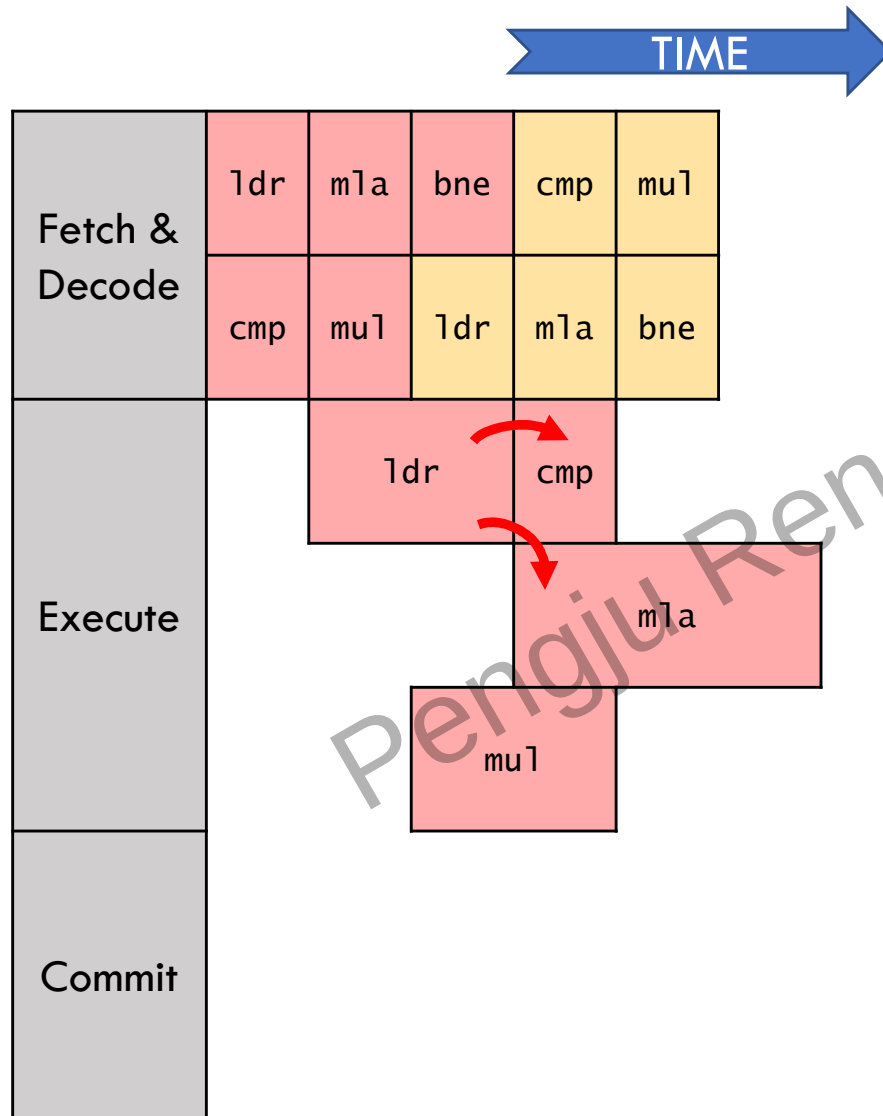


```

ldr    r5, [r3], #4
cmp    r1, r3
m1a    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
ldr    r5, [r3], #4
cmp    r1, r3
m1a    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
    
```

ldr, mul execute in 2 cycles
 cmp, bne execute in 1 cycle
 m1a executes in 3 cycles

Example: Superscalar OoO polynomial evaluation

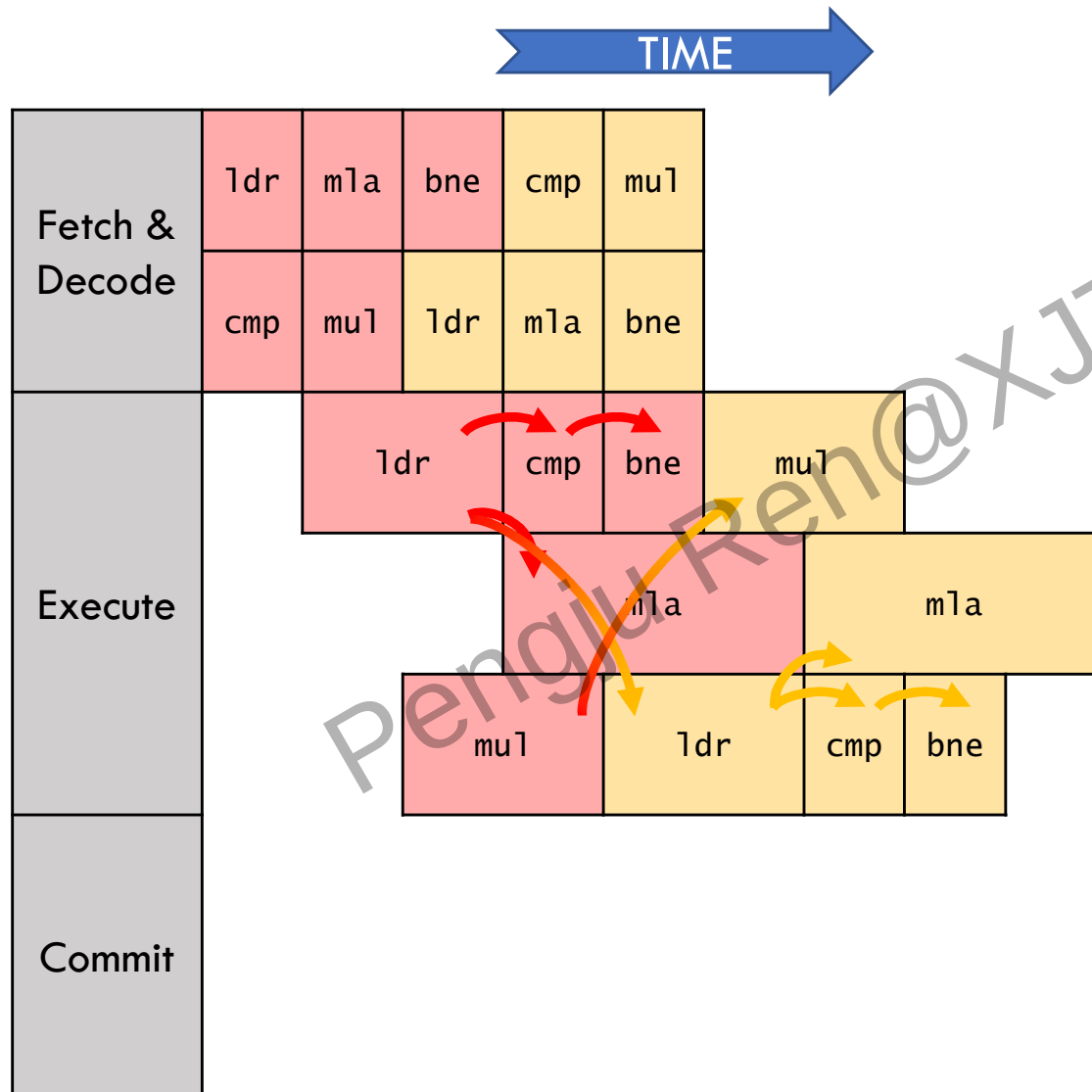


```

ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
ldr    r5, [r3], #4
cmp    r1, r3
mla    r0, r4, r5, r0
mul    r4, r2, r4
bne    .L3
    
```

ldr, mul execute in 2 cycles
 cmp, bne execute in 1 cycle
 mla executes in 3 cycles

Example: Superscalar OoO polynomial evaluation

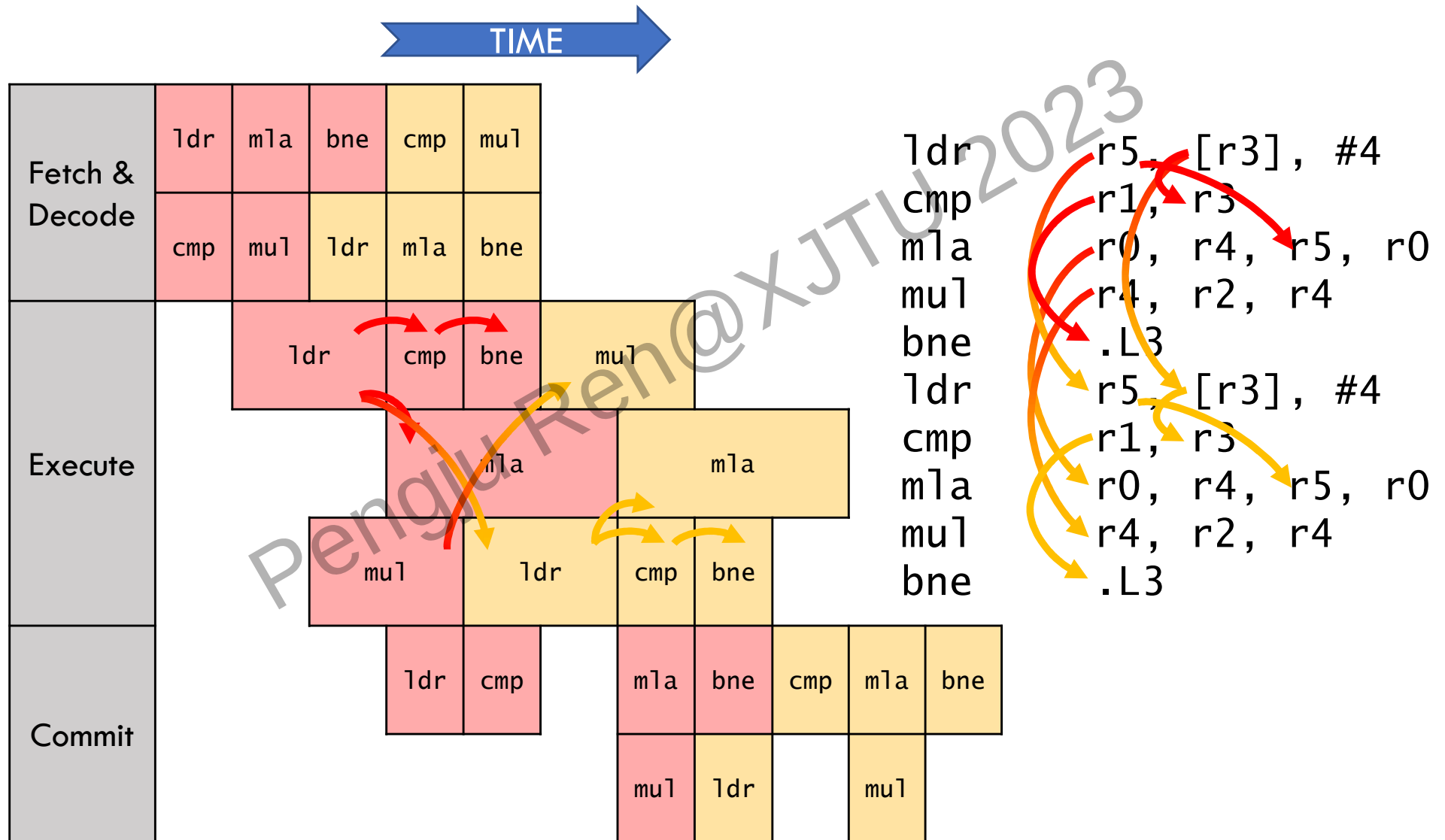


```

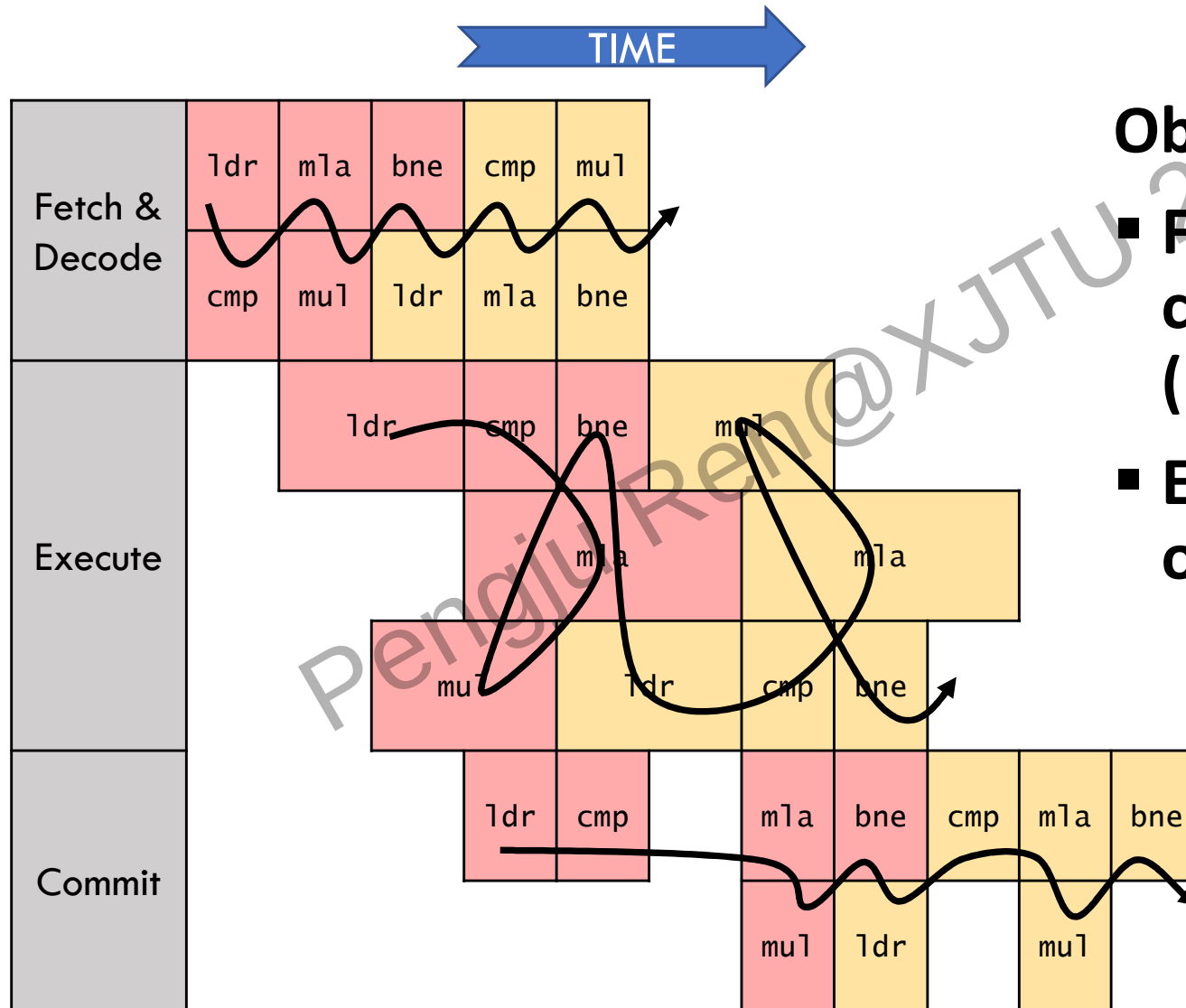
ldr r5, [r3], #4
cmp r1, r3
m1a r0, r4, r5, r0
mul r4, r2, r4
bne .L3
ldr r5, [r3], #4
cmp r1, r3
m1a r0, r4, r5, r0
mul r4, r2, r4
bne .L3
    
```

ldr, mul execute in 2 cycles
 cmp, bne execute in 1 cycle
 m1a executes in 3 cycles

Example: Superscalar OoO polynomial evaluation



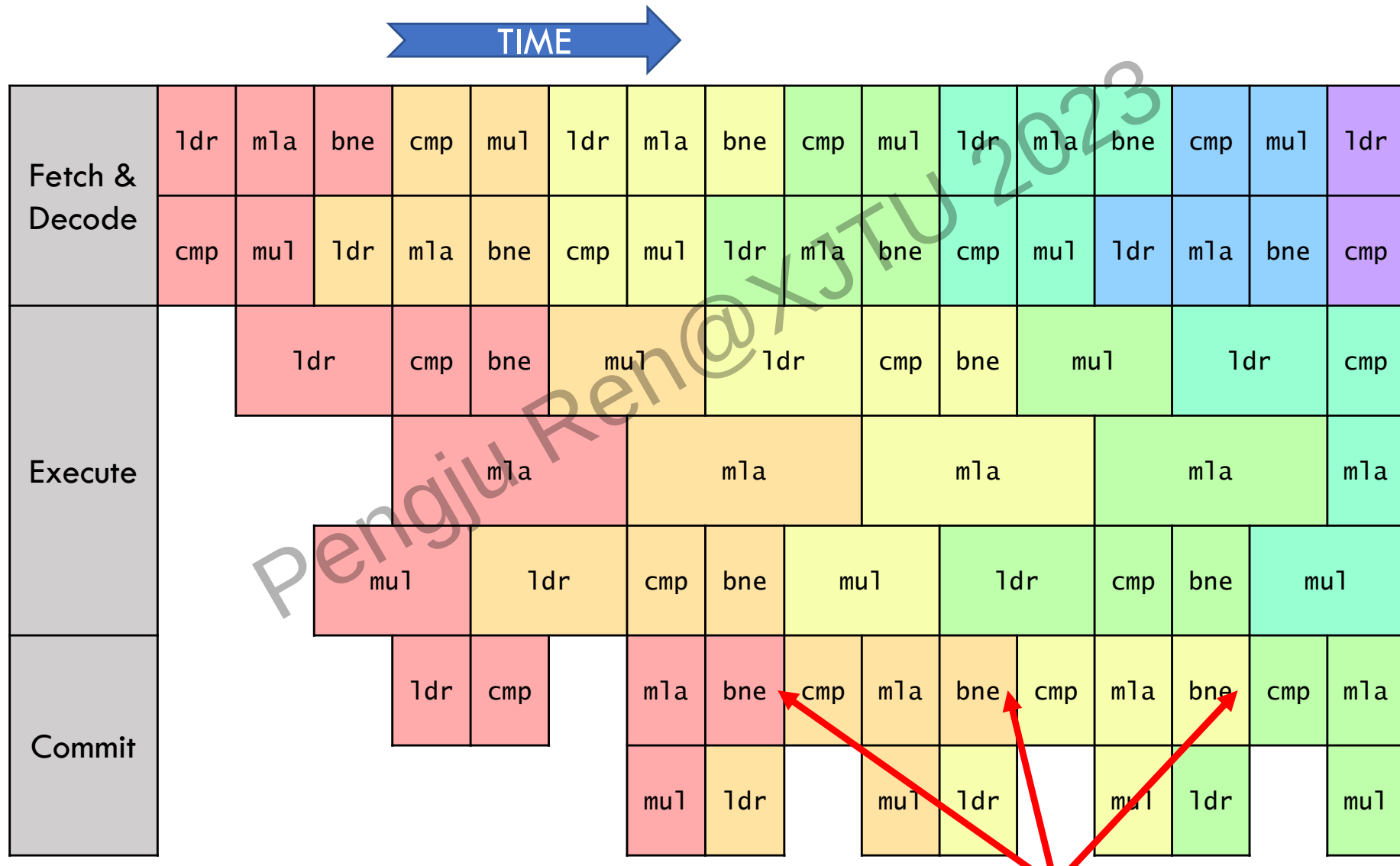
Example: Superscalar OoO polynomial evaluation



Observe:

- Front-end & commit in-order (i.e., left-to-right)
- Execute out-of-order

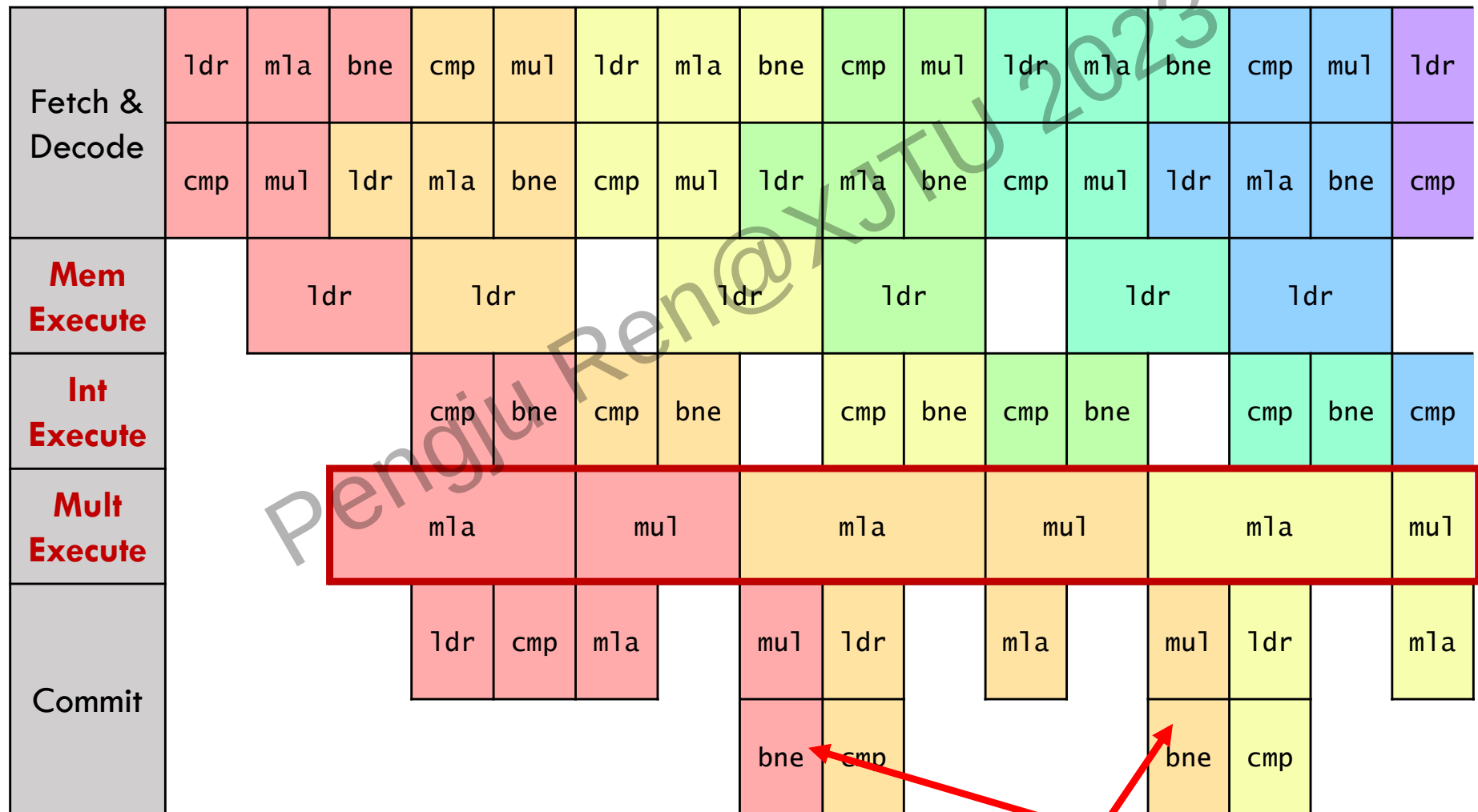
Example: Superscalar OoO polynomial evaluation



Structural hazards: Other throughput limitations

- However, **execution units are specialized**
 - Floating-point (add/multiply)
 - Integer (add/multiply/compare)
 - Memory (load/store)
- Processor designers must choose **which** execution units to include and **how many**
- ***Structural hazard***: Data is ready, but instr'n cannot issue because no hardware is available

Example: Structural hazards can severely limit performance



One loop iteration / 5 cycles 110

Throughput Bound

- **Ingredients:**
 - Number of operations to perform (of each type)
 - Number & issue rate of “execution ports”/“functional units” (of each type)
- **Throughput bound = ops / issue rate**
 - E.g., $(1 \text{ mla} + 1 \text{ mul}) / (2 + 3 \text{ cycles})$
- **Again, a real CPU might not exactly meet this bound**

Software Takeaway

- OoO is much less sensitive to “good code”
 - Better performance portability
 - Of course, compiler still matters
- OoO makes performance analysis much simpler
 - **Throughput bound:** Availability of *execution ports*
 - **Latency bound:** *“Critical path” latency*
 - Slowest gives good approximation of program perf

Out-of-Order Execution: Under the Hood

Pengju Ren@XJTLU 2023

Register Renaming

- “False dependences” can severely limit parallelism
 - Write-after-read (**WAR**)
 - Write-after-write(**WAW**)
 - Read-after-read (**RAR**)
- OoO processors eliminate false dependences by transparently *renaming registers*
 - CPU has many more “**physical**” than “**architectural**” registers
 - Each time register is written, it is allocated to a new physical register
 - Physical registers freed when instructions commit

Register Renaming

Original: $r1 \leftarrow r2/r3$

$r4 \leftarrow r1 * r5$

$r1 \leftarrow r3 + r6$

$r3 \leftarrow r1 - r5$

Renamed: $p1 \leftarrow p2/p3$

$p4 \leftarrow p1 * p5$

p8 $\leftarrow p3 + p6$

p9 $\leftarrow$ **p8** - $p5$

Architectural Registers
(ISA defined: $r0 \dots r31$)

Rename
table

rename p8

Physical
Registers
($p0 \dots p79$)

- Maintain mapping from ISA reg. names to physical registers
- When decoding an instruction that updates '**rx**':
 - allocate unused physical register '**py**' to hold inst result
 - set new mapping from '**rx**' to '**py**'
 - younger instructions using '**rx**' as input finds '**py**'
- De-allocate a physical register for reuse
- Need a place to hold *free physical registers* (Free list)

Memory Disambiguation

- CPU must respect store → load ordering
 - E.g., a later instruction reads a value from memory written by an earlier instruction, but the address might be implicit.

```
st X3 #4  
ld X2 #16
```

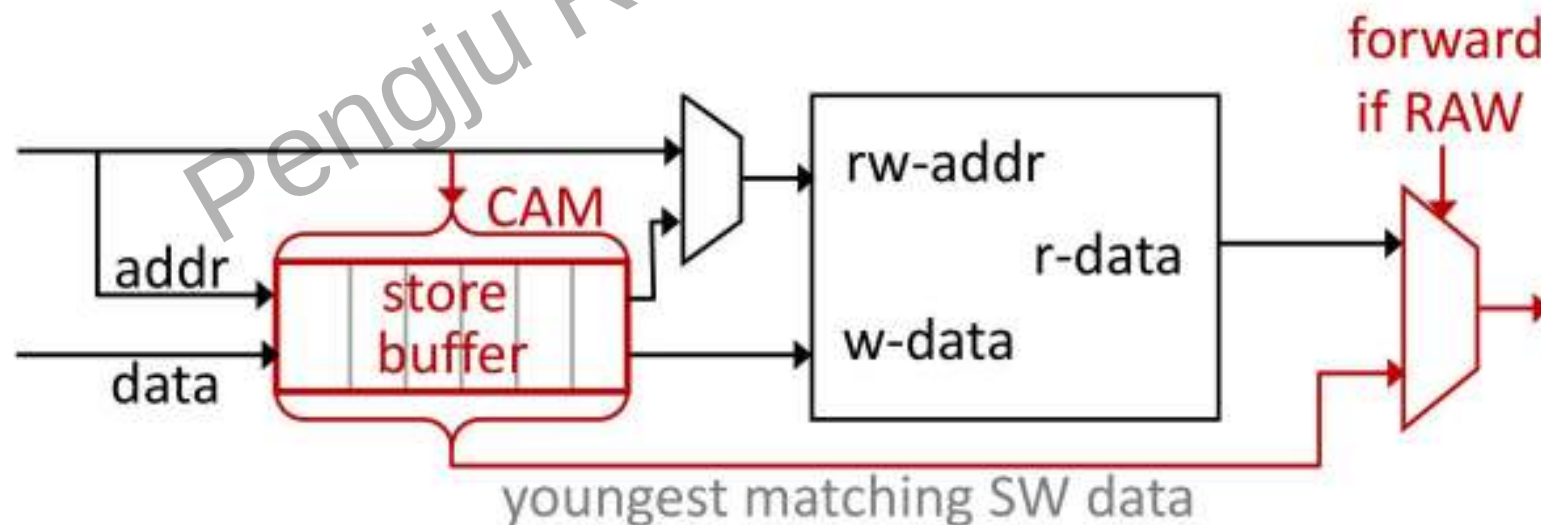
- But what if the OoO CPU executes the load first?
 - Must “rollback” + execute the load again (next slide)
- Corollary: OoO CPU must track the order of all loads & stores, and only write memory when a store commits

Store Buffer

allow younger **LD** to execute (out-of-order), must ensure **ST** target block not evicted

Memory dependence and forwarding

- younger **LD** must check against pending **ST** addresses in store buffer (CAM) for RAW dependence

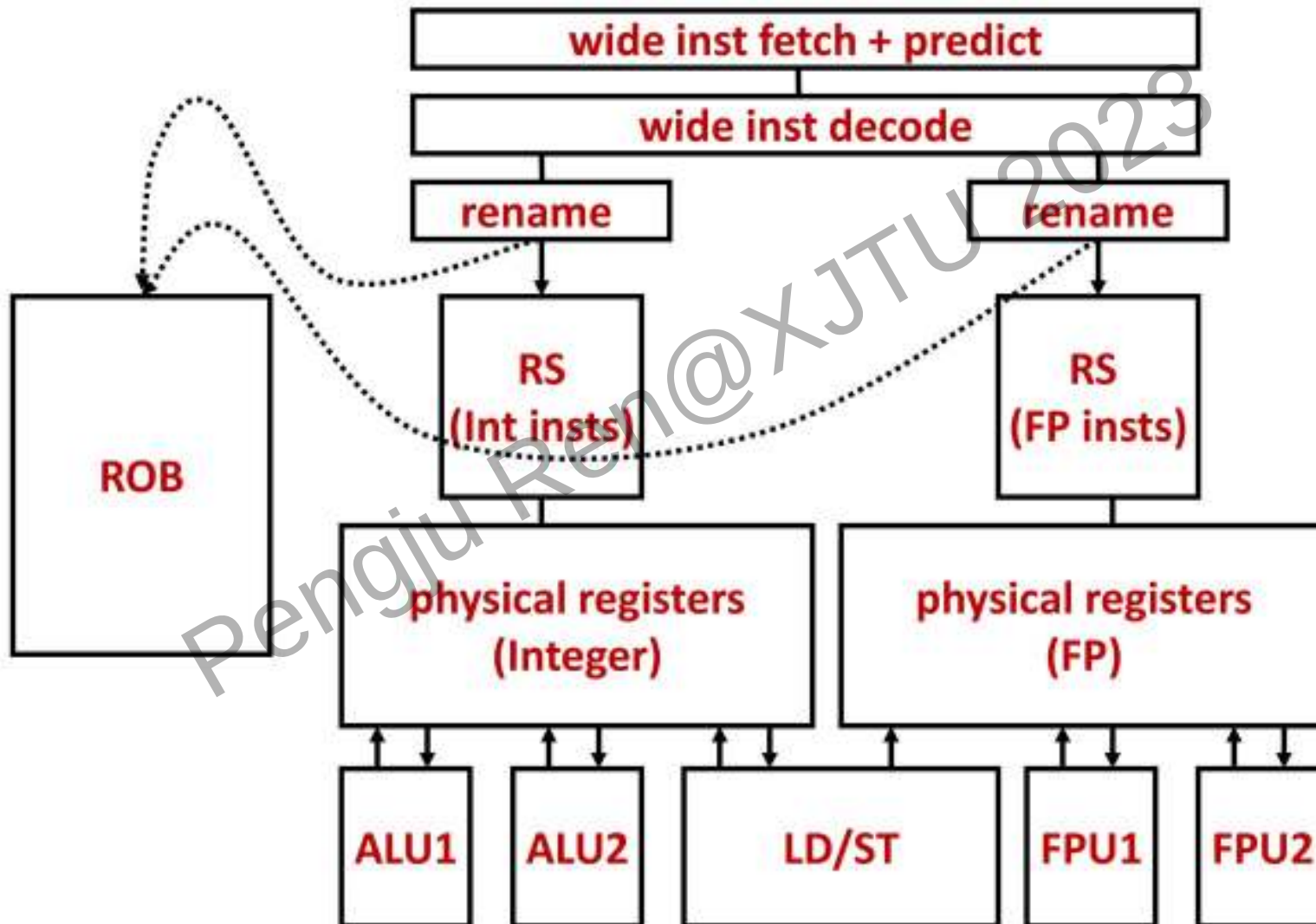


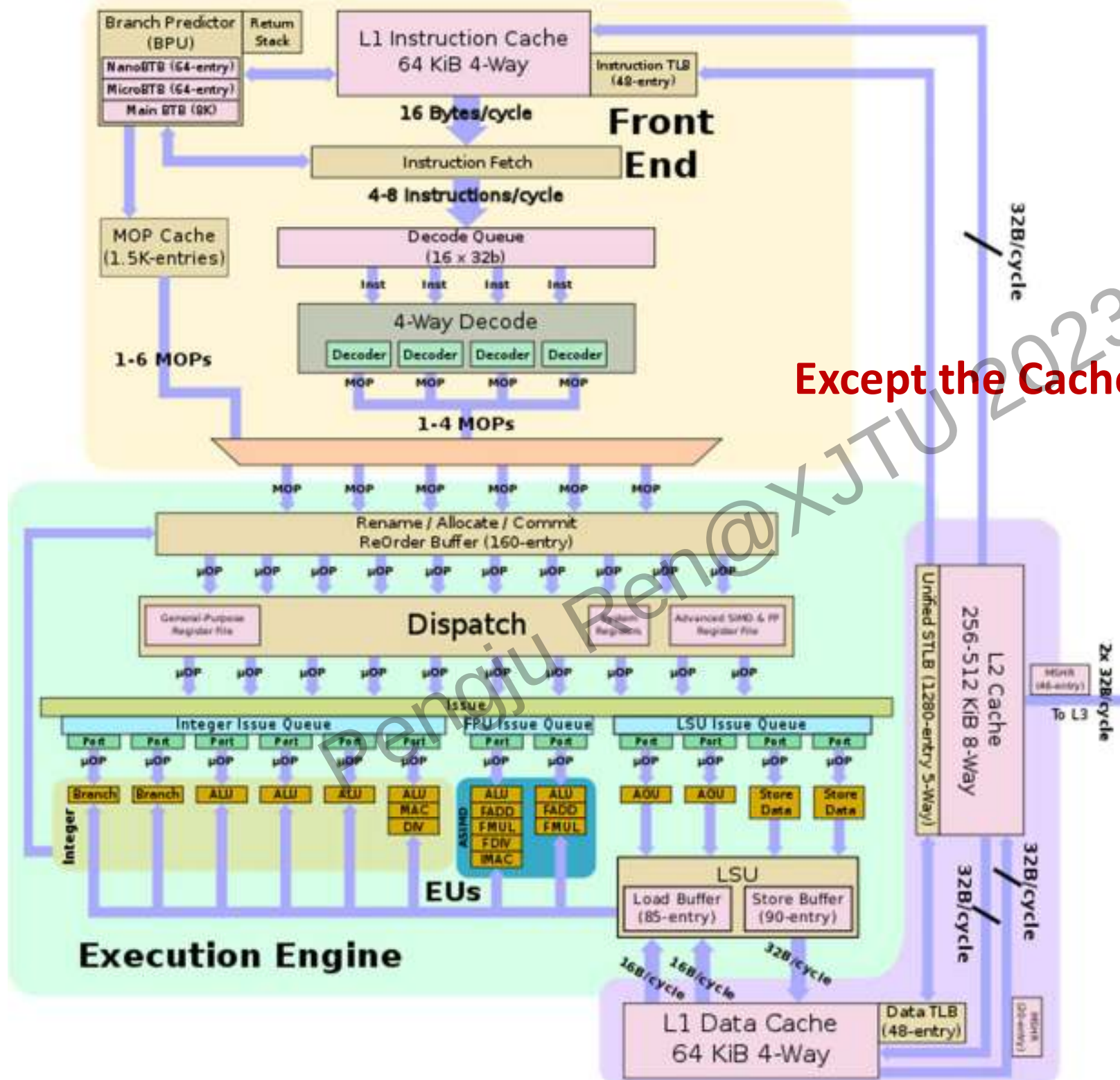
Rollback & Recovery

- OoO CPUs speculate constantly to improve performance
 - E.g., even guessing the results of a computation (“value prediction”)
- Need mechanisms to “**rollback**” to an earlier point in execution when speculation goes wrong
 - Complex: Need to recover old register names, flush pending memory operations, etc (Using **Checkpoint**, support fewer branch instructions on-the-fly, the # of **Checkpoint** is limited)
- Very expensive: Up to hundreds of instrns of work lost!
(width*depth + size_of_ROB)

SuperScalar Speculative OOO All Together

For an example:





Except the Caches and TLBs

Outline

- Instruction level parallel
- Pipeline
 - Data hazards
 - Control hazards
 - Structure hazards
- Out-of-Order Execution
 - Dataflow
- Optimization based on ILP
- Case study
 - Throughput bound
 - Latency bound
 - Performance Optimization

Optimization Code

Why optimize code is this programmers' problem?

- **In theory, compilers and hardware “understand” all this and can optimize your program; in practice they don’t.**
- **Understanding the capabilities and limitations of optimizing compilers**
- **They won’t know about a different algorithm that might be a much better “match” to the processor**

Example : Limitation of Optimizing Compiler(1)

Compilers must be careful to apply only **SAFE** optimization to a program. Instead, the compiler assumes the *worst case* and programmers must put more effort into writing programs to assist compiler to generate efficient code.

```
void twiddle1(long *xp, long *yp)
{
    *xp += *yp;
    *xp += *yp;
}

void twiddle2 (long *xp, long *yp)
{
    *xp += 2* *yp;
}
```

The compiler knows nothing about how twiddle1 will be called, it must assume that arguments *xp* and *yp* can be equal (*memory aliasing*).

Example : Limitation of Optimizing Compiler(2)

Compilers must be careful to apply only **SAFE** optimization to a program. Instead, the compiler assumes the *worst case* and programmers must put more effort into writing programs to assist compiler to generate efficient code.

```
long f();
```

```
long func1()
{
    return f() + f() + f() + f();
}
```

```
long func2()
{
    return 4*f();
}
```

```
long counter = 0;
long f()
{
    return counter++;
}
```

`f()` modifies some part of the global program state (`counter`). Changing the number of times it gets called changes the program behavior.

Example Program

Compute $\sin(x)$ using Taylor Expansion: $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$

For each element of an array of N floating-point numbers

```
void sinx(int N, int terms, float * x,
         float *result) {
    for (int i=0; i<N; i++) {
        float value = x[i];
        float number = x[i]*x[i]*x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++) {
            value += sign * number / denom;
            number *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

X[1]	X[2]	X[3]		X[n-1]	X[n]
------	------	------	------	--------	------

Taylor expansion of $\sin(x)$

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

```
void sinx(int N, int terms, float * x,
         float *result) {
    for (int i=0; i<N; i++) {
        float value = x[i];
        float number = x[i]*x[i]*x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++) {
            value += sign * number / denom;
            number *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

X[1]	X[2]	X[3]		X[n-1]	X[n]
------	------	------	------	--------	------

- How fast is this code?
- Where should we focus optimization efforts?
- What is the bottleneck?

Taylor expansion of $\sin(x)$

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

```
void sinx(int N, int terms, float * x,  
         float *result) {  
    for (int i=0; i<N; i++) {  
        float value = x[i];  
        float numer = x[i]*x[i]*x[i];  
        int denom = 6; // 3!  
        int sign = -1;  
  
        for (int j=1; j<=terms; j++) {  
            value += sign * numer / denom;  
            numer *= x[i] * x[i];  
            denom *= (2*j+2) * (2*j+3);  
            sign *= -1;  
        }  
  
        result[i] = value;  
    }  
}
```

- Where should we focus optimization efforts?
- A: Where most of the time is spent

Taylor expansion of $\sin(x)$

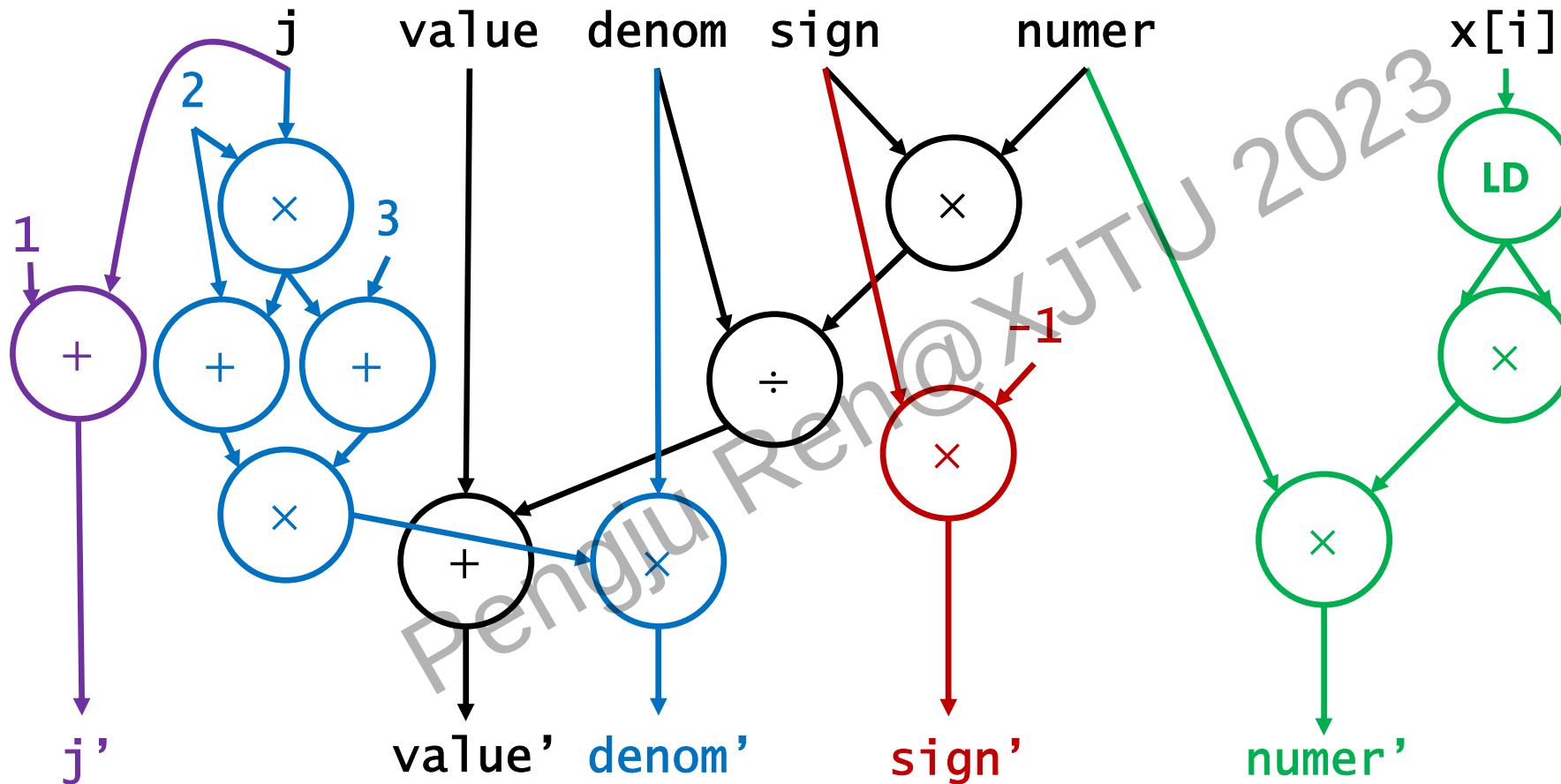
$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

```
void sinx(int N, int terms, float * x,  
         float *result) {  
    for (int i=0; i<N; i++) {  
        float value = x[i];  
        float numer = x[i]*x[i]*x[i];  
        int denom = 6; // 3!  
        int sign = -1;  
  
        for (int j=1; j<=terms; j++) {  
            value += sign * numer / denom;  
            numer *= x[i] * x[i];  
            denom *= (2*j+2) * (2*j+3);  
            sign *= -1;  
        }  
  
        result[i] = value;  
    }  
}
```

■ What is the bottleneck?

Dataflow for a single iteration

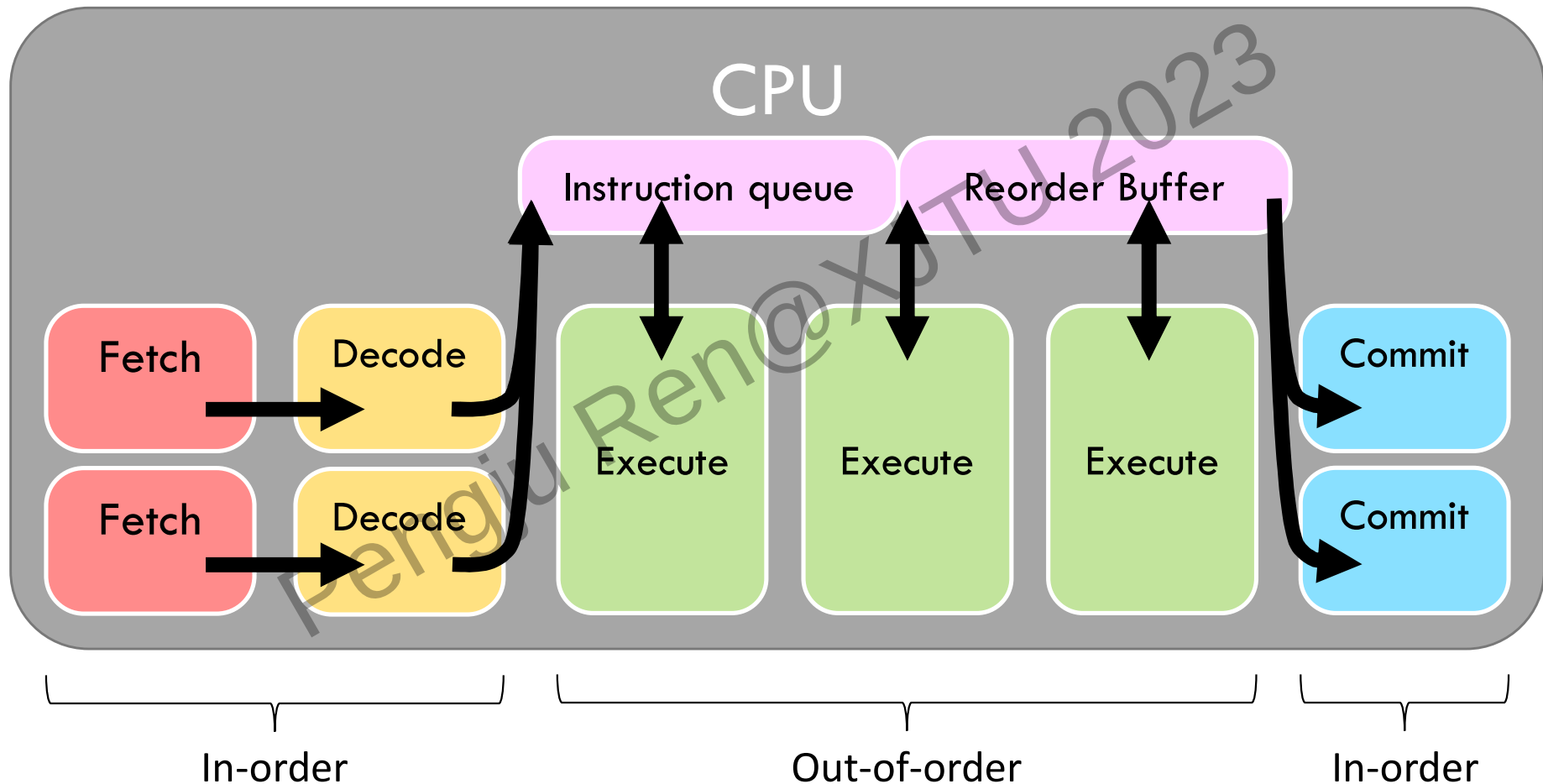
```
for (int j=1; j<=terms; j++) {  
    value += sign * number / denom;  
    number *= x[i] * x[i];  
    denom *= (2*j+2) * (2*j+3);  
    sign *= -1;  
}
```



OK, but how does this perform on a real machine?

Superscalar OOO Processor

- What in microarchitecture should we worry



OOO Processor Microarchitecture

■ What in microarchitecture should we worry about?

■ Fetch & Decode?

NO. Any reasonable machine will have sufficient frontend throughput to keep execution busy + all branches in this code are easy to predict (not always the case!).

■ Execution?

YES. This is where dataflow + most structural hazards will limit our performance.

■ Commit?

NO. Again, any reasonable machine will have sufficient commit throughput to keep execution busy.

Intel Skylake Execution Microarchitecture

	Integer			Floating Point		
	Latency	Pipelined?	Number	Latency	Pipelined?	Number
Add	1	✓	4	4*	✓	2
Multiply	3	✓	1	4	✓	2
Divide	21-83	✗	1	3-15	✗**	1
Load	2	✓	2			

* 3 cycles if using x87 instructions

** Can issue another operation after 4 cycles

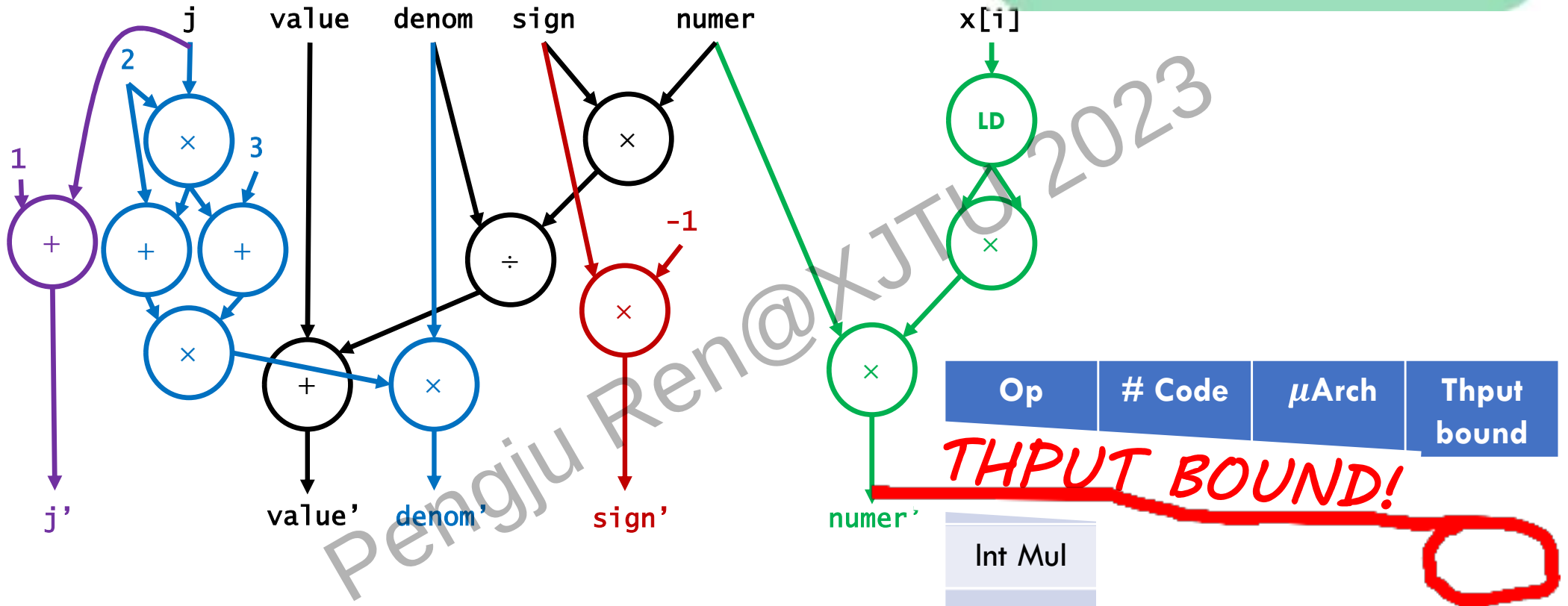
Source: Search for “Skylake” in

<https://www.agner.org/optimize/microarchitecture.pdf>

https://www.agner.org/optimize/instruction_tables.pdf

What is our throughput bound?

```
for (int j=1; j<=terms; j++) {
    value += sign * number / denom;
    number *= x[i] * x[i];
    denom *= (2*j+2) * (2*j+3);
    sign *= -1;
}
```

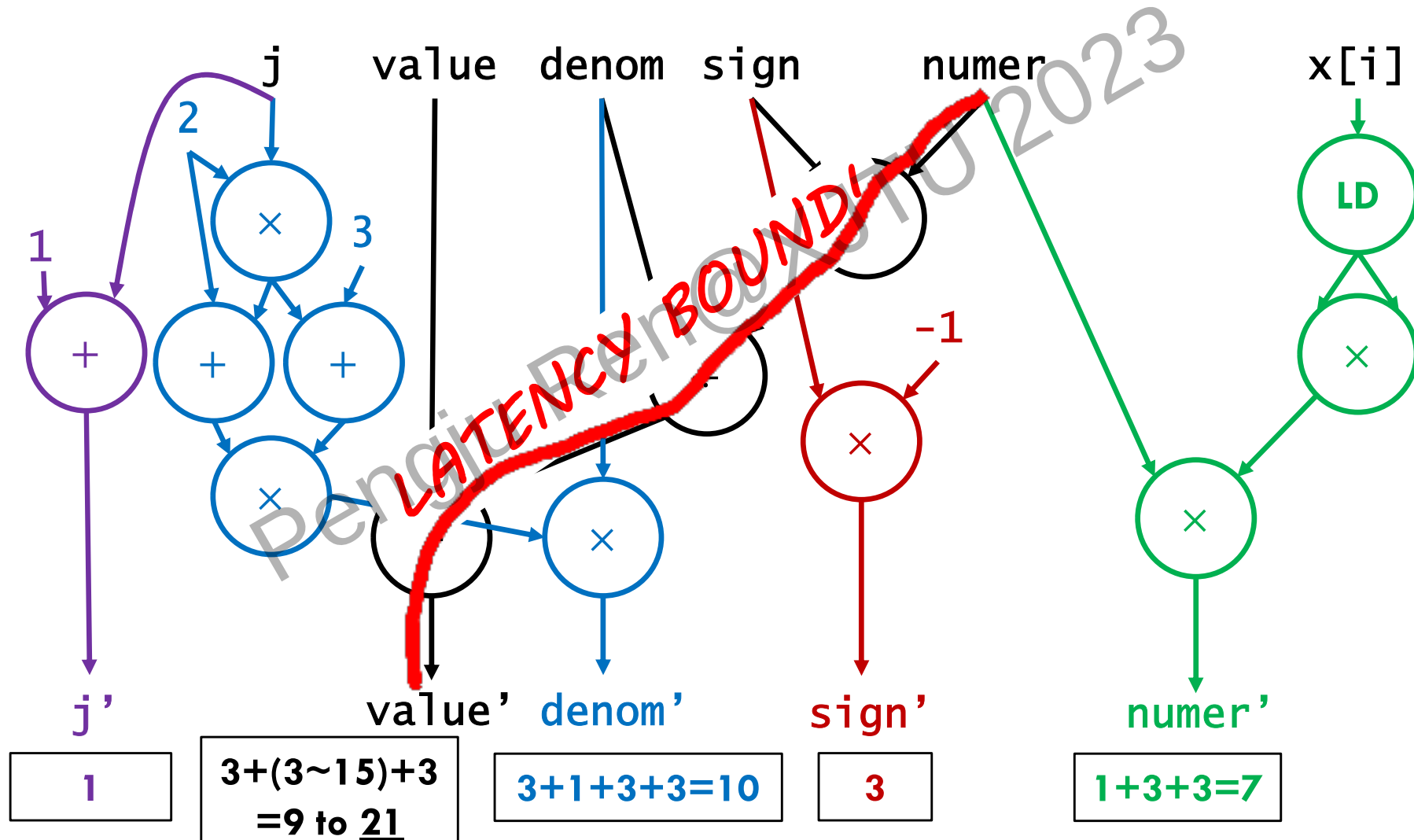


Throughput bound: Ignore data hazards, think *only* about max issue rate due to structural hazards

Op	# Code	μ Arch	Thput bound
Int Mul			
Int Div			
FP Add			
FP Mul			
FP Div			
Load			

What is our latency bound?

- **Latency bound:** Ignore structural hazards, think *only* about the critical path through data hazards



Takeaways

- **Observe performance of 23 cycles / element**
- **Latency bound dominates throughput bound**
→ We are latency bound!
- **Notes**
 - This analysis can often be “eyeballed” w/out full dataflow
 - Actual execution is more complicated, but latency/throughput bounds are good approximation
 - (Also, avoid division!!!)

Speeding up $\sin(x)$: Attempt #1

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

```
for (int j=1; j<=terms; j++) {  
    value += sign * number / denom;  
    number *= x[i] * x[i];  
    denom *= (2*j+2) * (2*j+3);  
    sign *= -1;  
}
```

■ What if we eliminate unnecessary work?

```
void sinx_better(int N, int terms, float * x,  
                float *result) {  
    for (int i=0; i<N; i++) {  
        float value = x[i];  
        float x2 = x[i]*x[i];  
        float number = x2*x[i];  
        int denom = 6; // 3!  
        int sign = -1;  
  
        for (int j=1; j<=terms; j++) {  
            value += sign * number / denom;  
            number *= x2;  
            denom *= (2*j+2) * (2*j+3);  
            sign = -sign;  
        }  
  
        result[i] = value;  
    }  
}
```

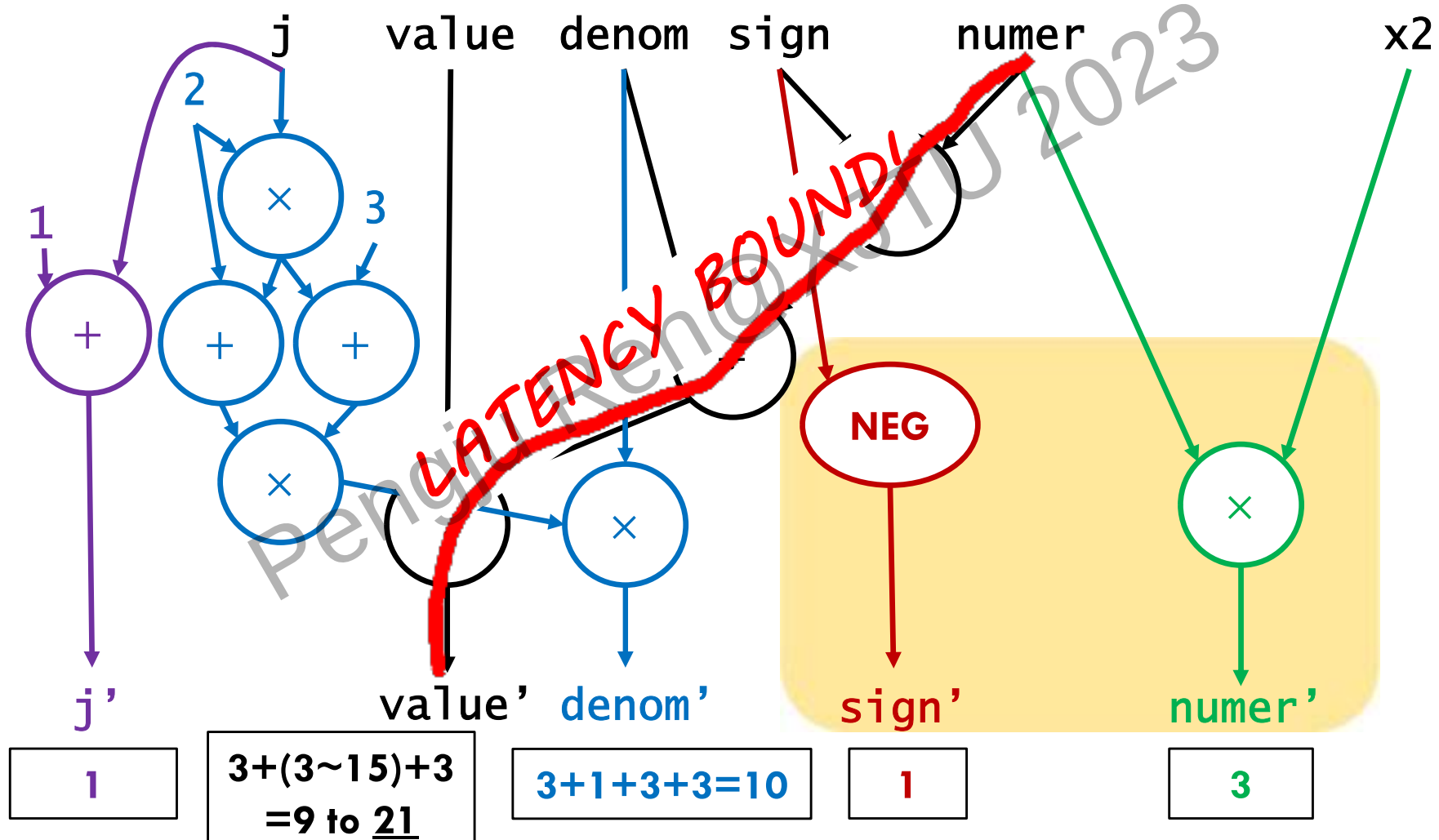
A: Small improvement.

6ns / element $\approx$
18 cycles / element

Why not better?

What is our latency bound?

- Find the critical path in the dataflow graph



Attempt #1 Takeaways

- First attempt didn't change latency bound
- To get real speedup, we need to focus on the performance *bottleneck*
- Q: Why did we get any speedup at all?
- A: Actual dynamic scheduling is complicated; would need to simulate execution in more detail (minus the usage of multiplier, therefore reduce the % of *structure hazard*)

Speeding up $\sin(x)$: Attempt #2

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

```
for (int j=1; j<=terms; j++) {  
    value += sign * number / denom;  
    number *= x2;  
    denom *= (2*j+2) * (2*j+3);  
    sign = -sign;  
}
```

■ Let's focus on that pesky division...

```
void sinx_predenom(int N, int terms, float * x, float *result) {  
    float rdenom[MAXTERMS];  
    int denom = 6;  
    for (int j = 1; j <= terms; j++) {  
        rdenom[j] = 1.0/denom;  
        denom *= (2*j+2) * (2*j+3);  
    }  
    for (int i=0; i<N; i++) {  
        float value = x[i];  
        float x2 = value * value;  
        float number = x2 * value;  
        int sign = -1;  
        for (int j=1; j<=terms; j++) {  
            value += sign * number * rdenom[j];  
            number *= x2;  
            sign = -sign;  
        }  
        result[i] = value;  
    }  
}
```

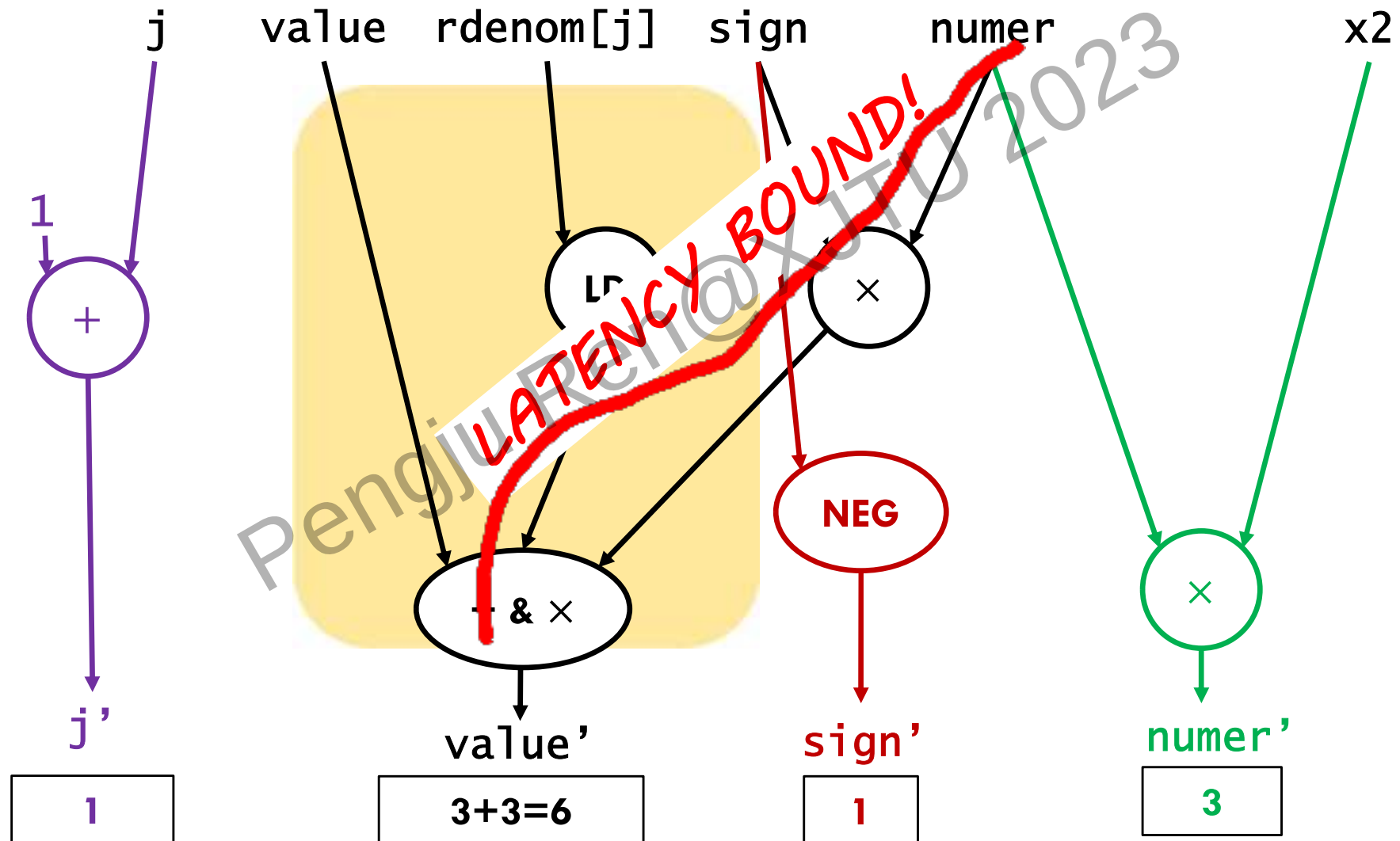
A: Big improvement!

2.4ns / element $\approx$

7.7 cycles / element

What is our latency bound?

- Find the critical path in the dataflow graph



Attempt #2 Takeaways

- **Attacking the bottleneck got nearly 3×!**
- **...But performance is still near the latency bound, can we do better?**

Pengju Ren@XJTU 2023

Speeding up $\sin(x)$: Attempt #3

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

```
for (int j=1; j<=terms; j++) {  
    value += sign * numer * rdenom[j];  
    numer *= x2;  
    sign = -sign;  
}
```

■ Don't need sign in inner-loop either

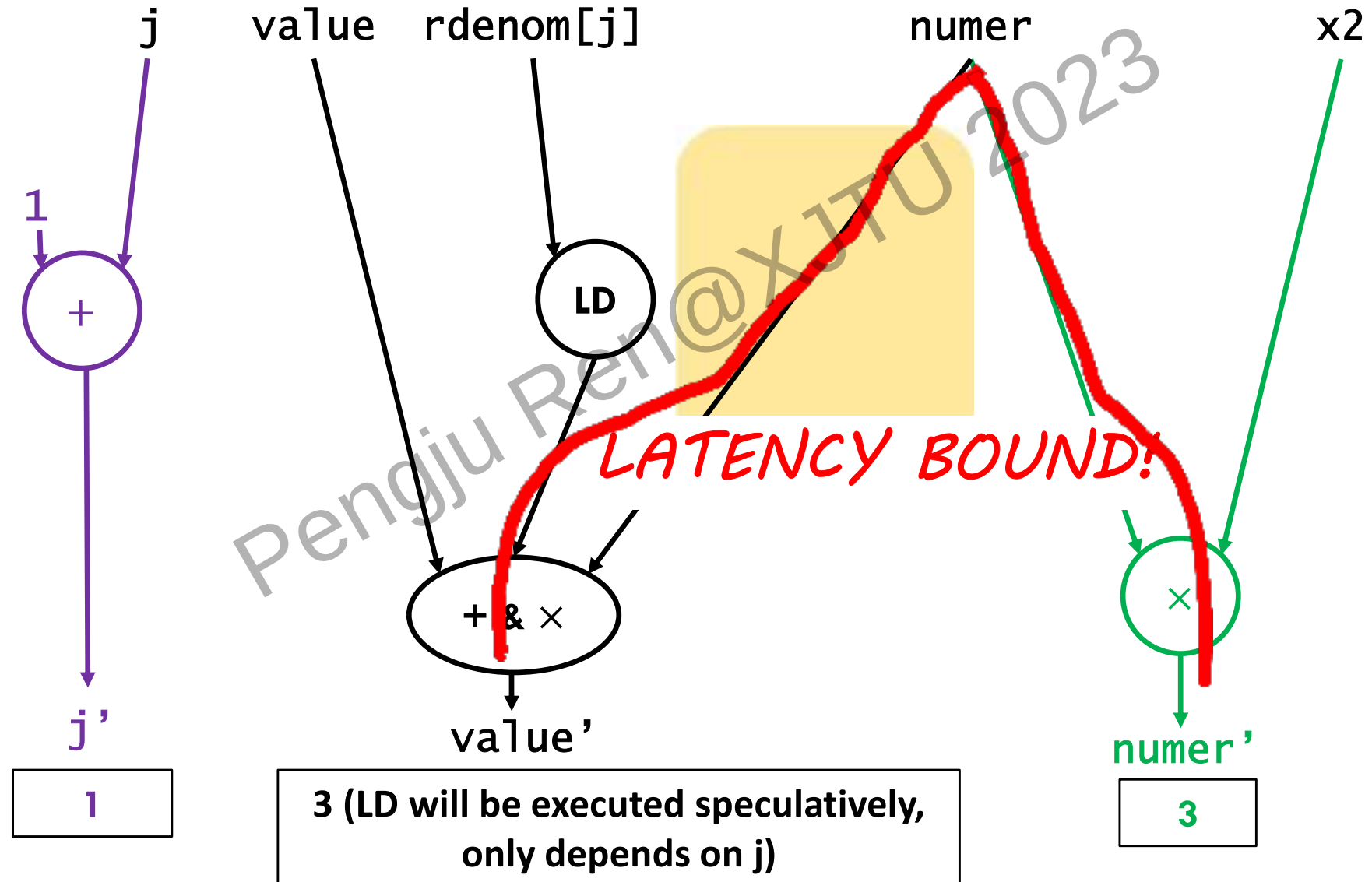
```
void sinx_predenoms(int N, int terms, float * x, float *result) {  
    float rdenom[MAXTERMS];  
    int denom = 6;  
    float sign = -1.0;  
    for (int j = 1; j <= terms; j++) {  
        rdenom[j] = sign/denom;  
        denom *= (2*j+2) * (2*j+3);  
        sign = -sign;  
    }  
    for (int i=0; i<N; i++) {  
        float value = x[i];  
        float x2 = value * value;  
        float numer = x2 * value;  
        for (int j=1; j<=terms; j++) {  
            value += numer * rdenom[j];  
            numer *= x2;  
        }  
        result[i] = value;  
    }  
}
```

1.1ns / element $\approx$

3.5 cycles / element

What is our latency bound?

- Find the critical path in the dataflow graph



Attempt #3 Takeaways

- We're down to the latency of a single, fast operation per iteration
- + Observed performance is very close to this latency bound, so throughput isn't limiting
- → We're done optimizing individual iterations
- How to optimize multiple iterations?
 - Eliminate dependence chains across iterations
 - A) Loop unrolling (ILP)
 - B) Explicit parallelism (SIMD, threading)

Speeding up $\sin(x)$: Loop unrolling

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

- Compute multiple elements per iteration

```
for (int i=0; i<N; i++) {  
    float value = x[i];  
    float x2 = value * value;  
    float numer = x2 * value;  
    for (int j=1; j<=terms; j++) {  
        value += numer * rdenom[j];  
        numer *= x2;  
    }  
    result[i] = value;  
}
```

```
void sinx_unrollx2(int N, int terms, float *x, float *result) {  
    // same predom stuff as before...  
    for (int i=0; i<N; i++) {  
        float value = x[i];  
        float x2 = value * value;  
        float x4 = x2 * x2;  
        float numer = x2 * value;  
        for (int j=1; j<=terms; j+=2) {  
            value += numer * rdenom[j];  
            value += numer * x2 * rdenom[j+1];  
            numer *= x4;  
        }  
        result[i] = value;  
    }  
}
```

Correct?

Speeding up $\sin(x)$: Loop unrolling

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

■ Compute multiple elements per iteration

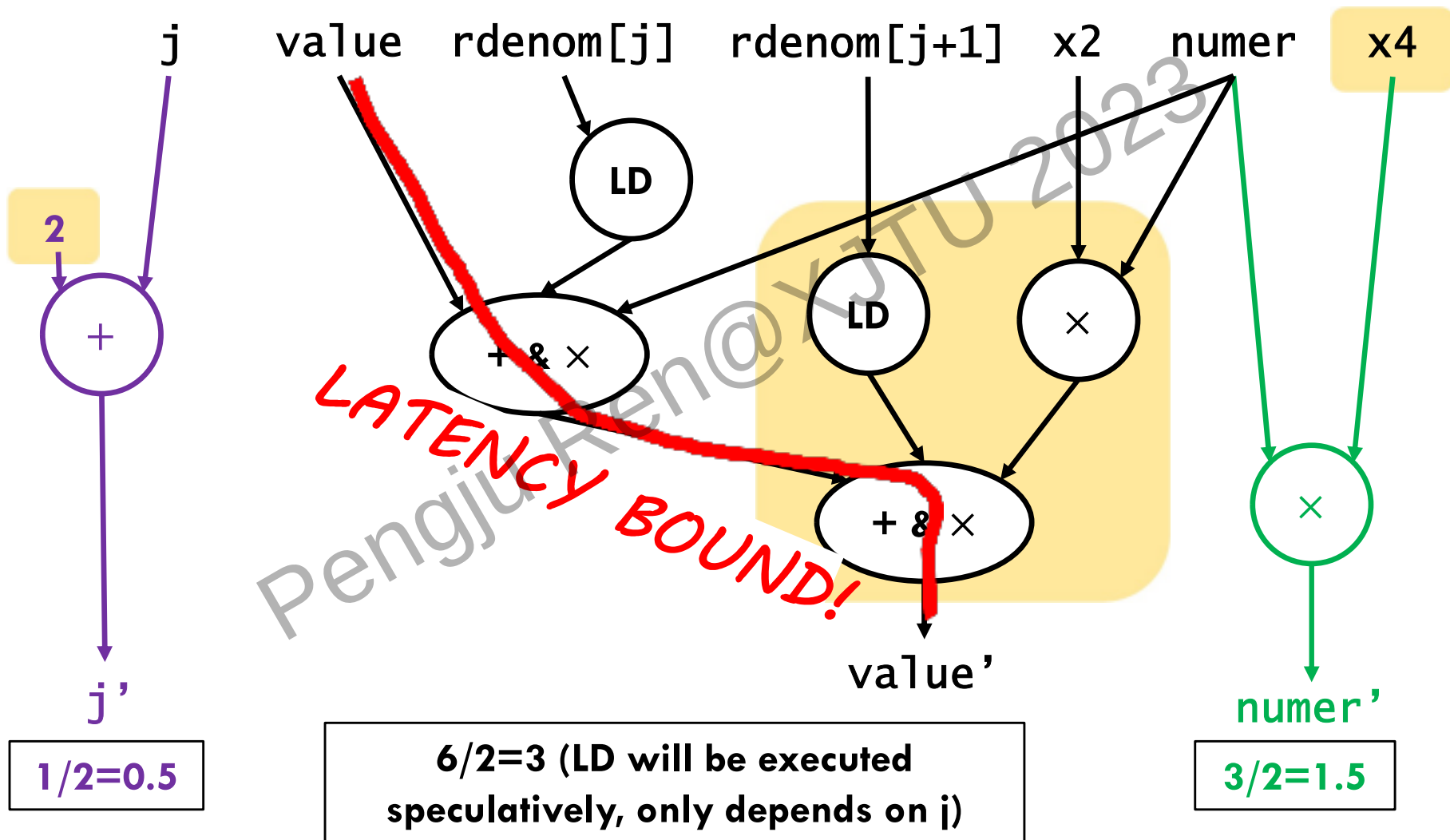
```
void sinx_unrollx2(int N, int terms, float * x, float *result) {  
    // same predom stuff as before...  
    for (int i=0; i<N; i++) {  
        float value = x[i];  
        float x2 = value * value;  
        float x4 = x2 * x2;  
        float numer = x2 * value;  
        int j;  
        for (j=1; j<=terms-1; j+=2) {  
            value += numer * rdenom[j];  
            value += numer * x2 * rdenom[j+1];  
            numer *= x4;  
        }  
        for (; j<=terms; j++) {  
            value += numer * rdenom[j];  
            numer *= x2;  
        }  
        result[i] = value;  
    }  
}
```

0.99 ns / element $\approx$
3.2 cycles / element

Didn't change ☹

What is our latency bound?

- Find the critical path in the dataflow graph



Speeding up $\sin(x)$: Loop unrolling #2

■ What if floating point associated + distributed?

```
void sinx_unrollx2(int N, int terms, float * x, float *result){
```

```
    // same predom stuff as before...
```

```
    for (int i=0; i<N; i++) {
```

```
        float value = x[i];
```

```
        float x2 = value * value;
```

```
        float x4 = x2 * x2;
```

```
        float numer = x2 * value;
```

```
        int j;
```

```
        for (j=1; j<=terms-1; j++) {
```

```
            value += numer * (rdenom[j] + x2 * redom[j+1]);
```

```
            numer *= x4;
```

```
        }
```

```
        for (; j<=terms; j++) {
```

```
            value += numer * rdenom[j];
```

```
            numer *= x2;
```

```
        }
```

```
        result[i] = value;
```

```
    }
```

```
}
```

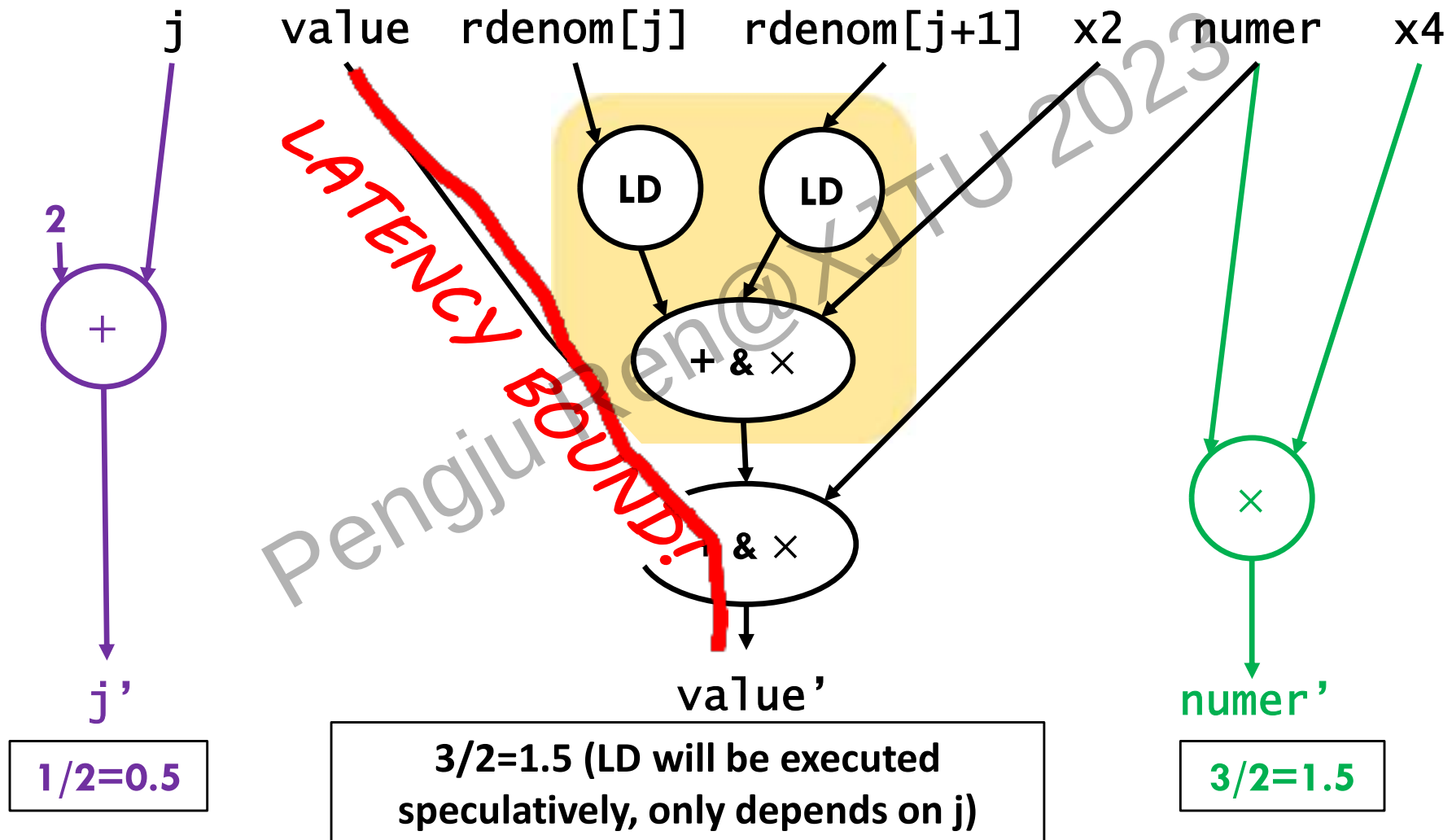
```
for (j=1; j<=terms-1; j+=2) {  
    value += numer * rdenom[j];  
    value += numer * x2 * rdenom[j+1];  
    numer *= x4;  
}
```

0.69 ns / element $\approx$

2.2 cycles / element

What is our latency bound?

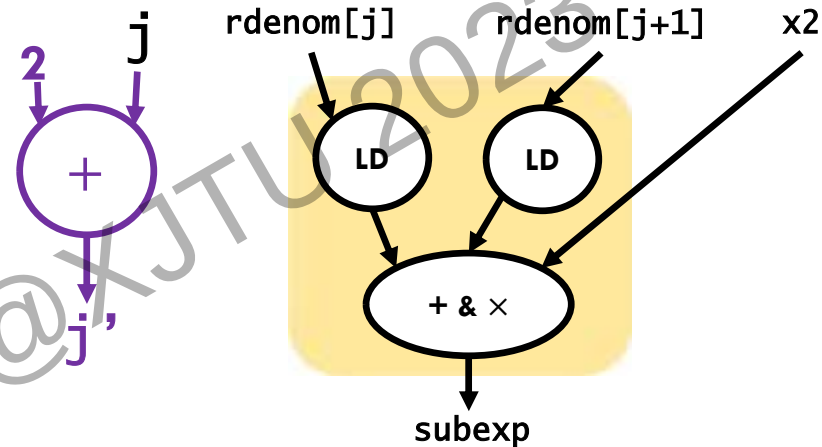
- Find the critical path in the dataflow graph



Loads do not limit $\sin(x)$

- Consider just the slice of the program that generates the subexpression: $(\text{rdenom}[j] + x2 \times \text{rednom}[j + 1])$

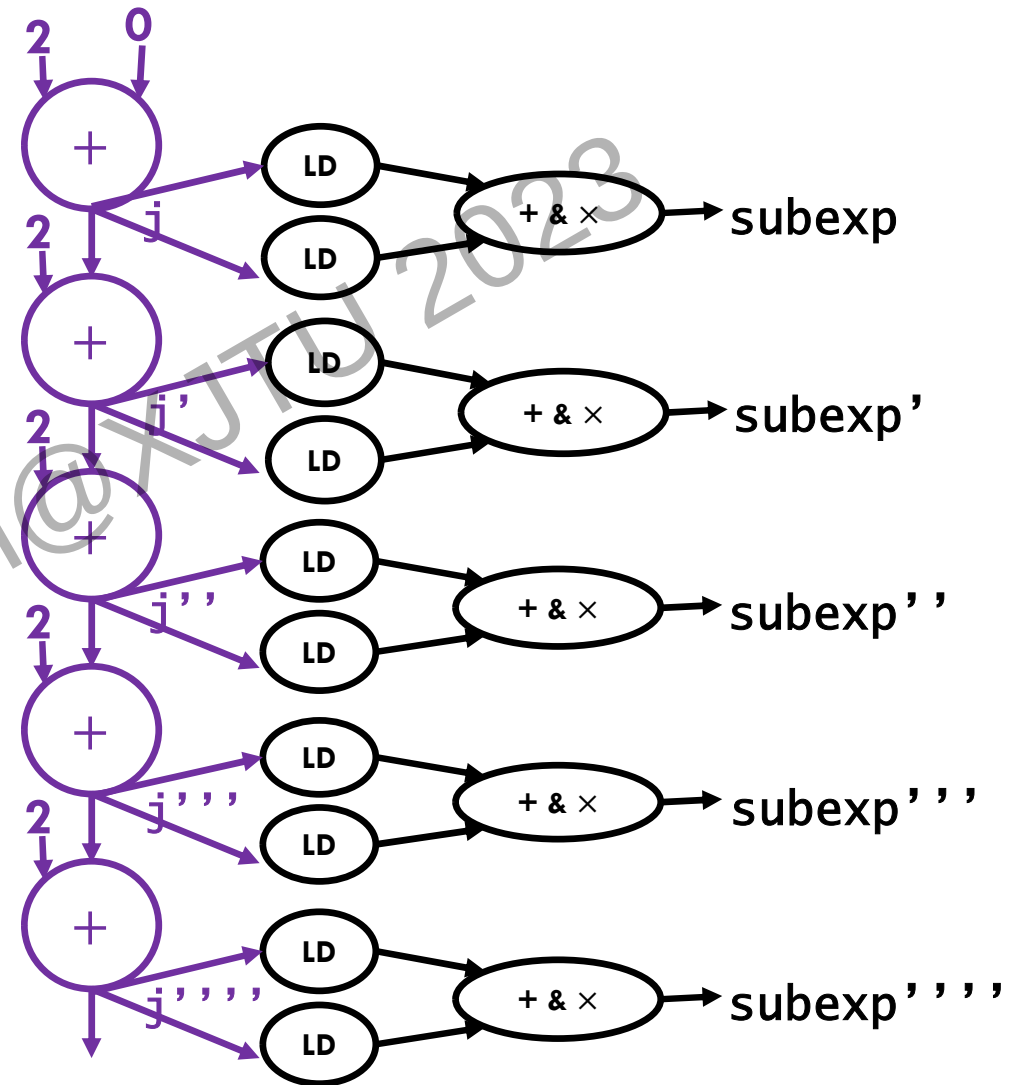
- What is this program's latency + throughput bound?



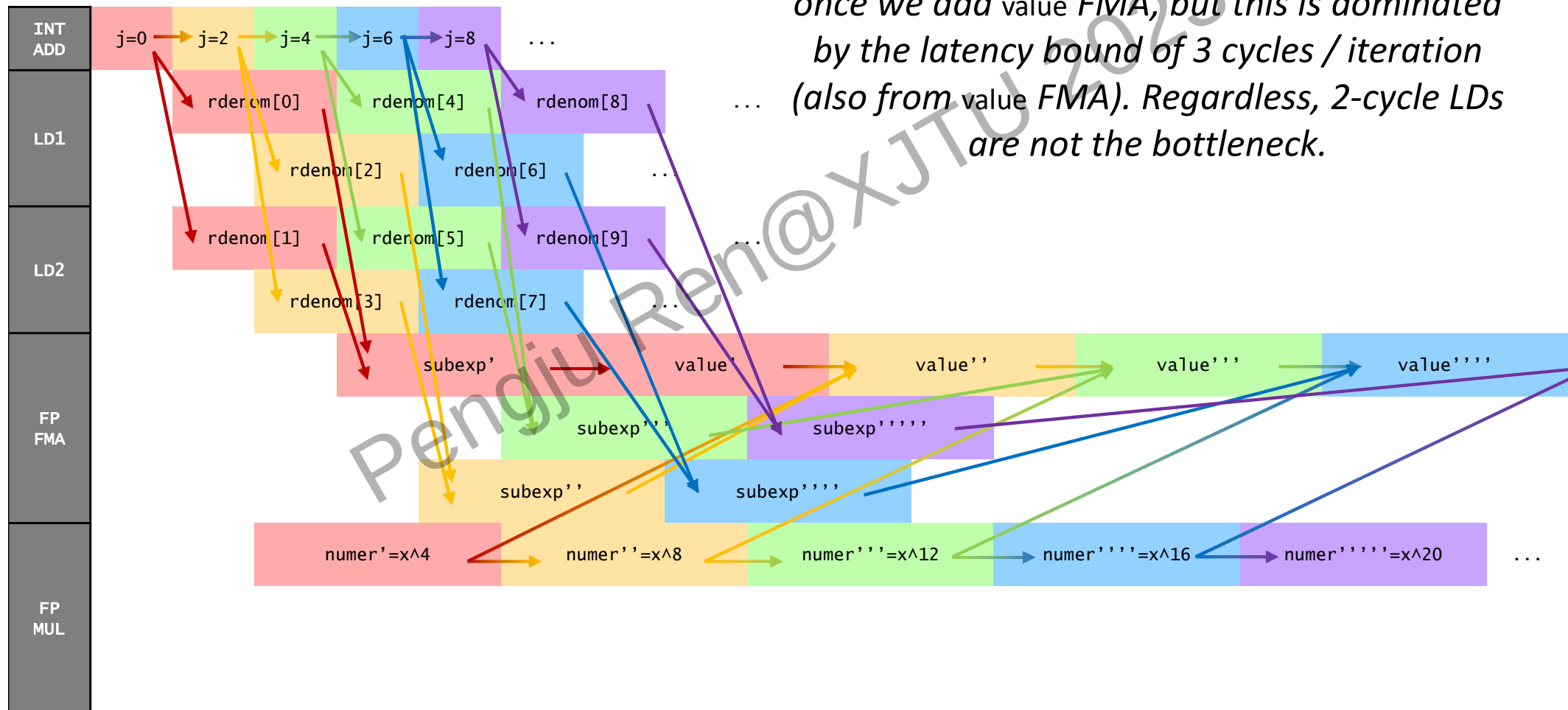
- Latency bound: 1 cycle / iteration!
 - Through j' computation, not the subexpression computation – *there is no cross-iteration dependence in the subexpression!*
- Throughput bound: also 1 cycle / iteration
 - 1 add / 4 adders; 2 LDs / 2 LD units; 1 FP FMA / 1 FP unit
 - (This will change to 2 cycles if we add the value FMA)

Loads do not limit $\sin(x)$: Visualization

- Consider just the slice of the program that generates the **subexpression**: $(\text{rdenom}[j] + x2 \times \text{rednom}[j + 1])$
- Subexpressions are *off the critical path* + we have enough throughput to produce next subexpression each cycle (excluding value FMA)



Loads do not limit $\sin(x)$: Example execution



Note: Throughput limit is 2 cycles / iteration once we add value FMA, but this is dominated by the latency bound of 3 cycles / iteration ... (also from value FMA). Regardless, 2-cycle LDs are not the bottleneck.

Loop unrolling takeaways

- **Need to break dependencies across iterations to get speedup**
 - Unrolling by itself doesn't help
- **We are now seeing throughput effects**
 - Latency bound = 1.5 vs. observed = 2.2
- **Can unroll loop 3x, 4x to improve further, but...**
- **...Diminishing returns (1.65 cycles / element at 4x)**

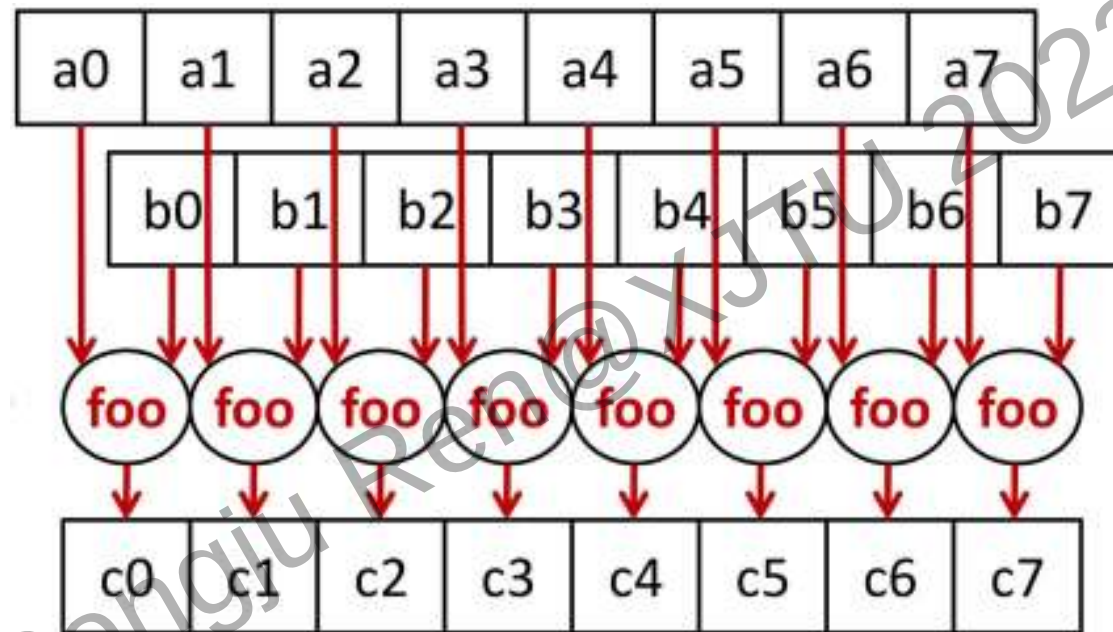
What if? #1 Impact of structural hazards

- **Q: What would happen to $\sin(x)$ if we only had a single, unpipelined floating-point multiplier?**
- **A1: Performance will be much worse**
- **A2: We will hit throughput bound much earlier**
- **A3: Loop unrolling will help by reducing multiplies**

What if? #2 Impact of structural hazards

- **Q: What would happen to $\sin(x)$ if LDs (cache hits) took 2 cycles instead of 1 cycle?**
- **A: Nothing. This program is latency bound, and LDs are not on the critical path.**

SIMD (Single Instruction Multiple Data)



Instantiate **k** copies of the hardware unit **foo** to process **k** iterations of the loop in parallel

Speeding up $\sin(x)$: Going parallel (explicitly)

■ Use ISPC to vectorize the code

```
export void sinx_reference
    (uniform int N,
     uniform int terms,
     uniform float x[],
     uniform float result[]) {
    foreach (i=0 ... N) {
        float value = x[i];
        float number = x[i]*x[i]*x[i];
        uniform int denom = 6; // 3!
        uniform int sign = -1;
        for (uniform int j=1; j<=terms; j++) {
            value += sign * number / denom;
            number *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[i] = value;
    }
}
```

```
void sinx
    (int N,
     int terms,
     float * x,
     float *result) {
    for (int i=0; i<N; i++) {
        float value = x[i];
        float number = x[i]*x[i]*x[i];
        int denom = 6; // 3!
        int sign = -1;
        for (int j=1; j<=terms; j++) {
            value += sign * number / denom;
            number *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[i] = value;
    }
}
```

1.0 ns / element $\approx$ 3.2 cycles / element

Speeding up $\sin(x)$: Going parallel (explicitly) + optimize

```
export void sinx_unrollx2a(uniform int N, uniform int terms,
                          uniform float x[],
                          uniform float result[]) {
    uniform float rdenom[MAXTERMS];
    uniform int denom = 6;
    uniform float sign = -1;
    for (uniform int j = 1; j <= terms; j++) {
        rdenom[j] = sign/denom;
        denom *= (2*j+2) * (2*j+3);
        sign = -sign;
    }
    foreach (i=0 ... N) {
        float value = x[i];
        float x2 = value * value;
        float x4 = x2 * x2;
        float numer = x2 * value;
        uniform int j;
        for (j=1; j<=terms-1; j+=2) {
            value += numer * (rdenom[j] + x2 * rdenom[j+1]);
            numer *= x4;
        }
        for (; j <= terms; j++) {
            value += numer * rdenom[j];
            numer *= x2;
        }
        result[i] = value;
    }
}
```

0.14 ns / element $\approx$
0.45 cycles / element

SIMD takeaways

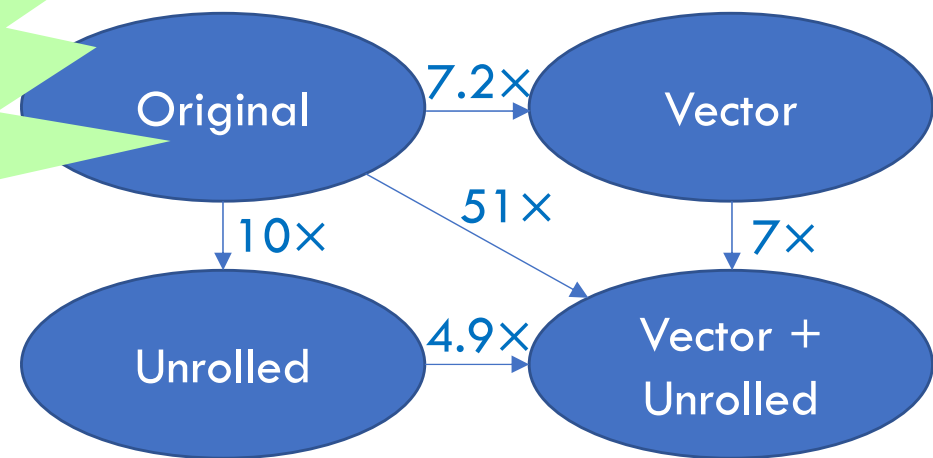
- Well, that was easy!

- Cycles per element:

	Scalar	Vector
Unoptimized	23	3.2
Unrolled	2.2	0.45

- Speedup

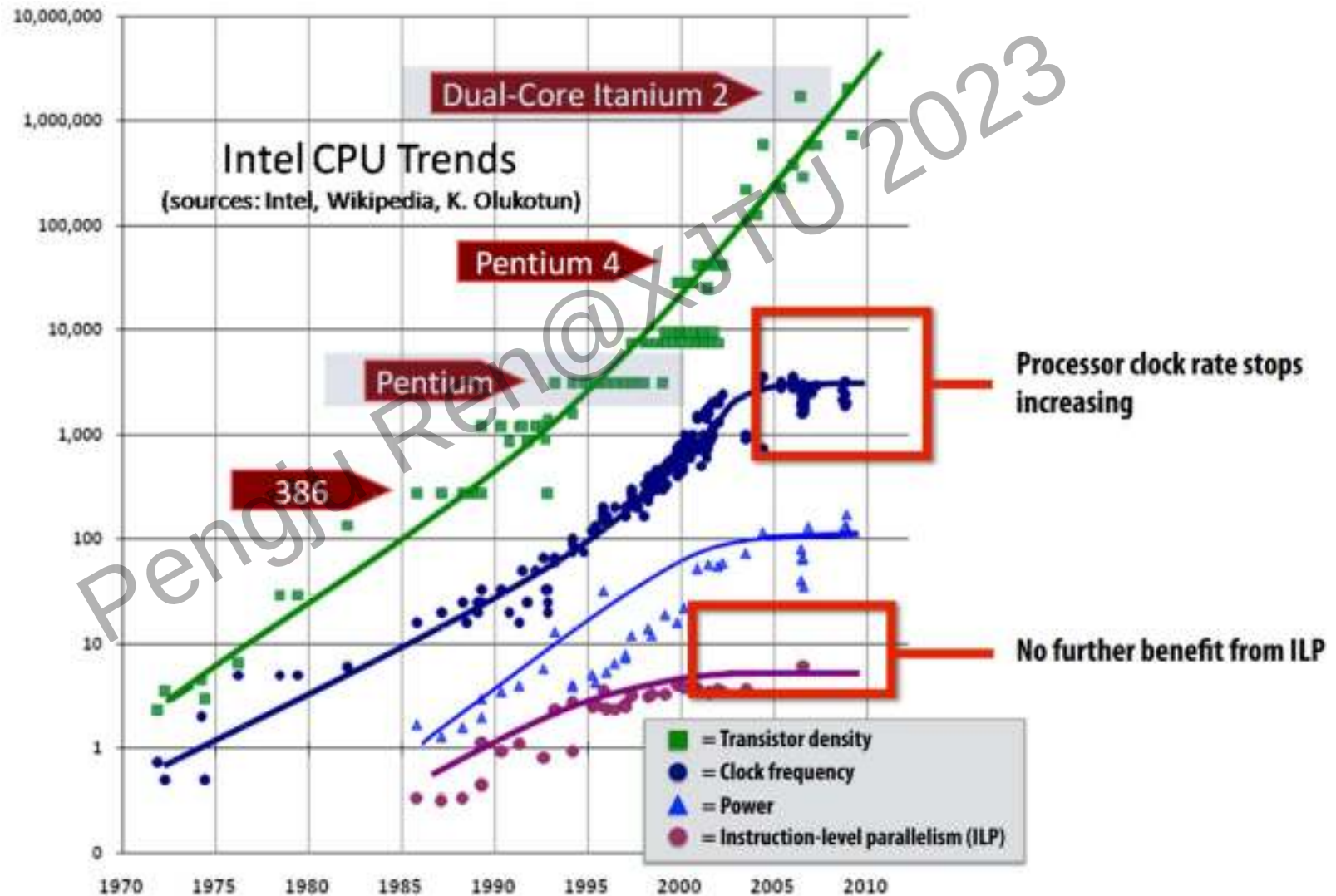
Maximum speedup
requires hand tuning
+ explicit parallelism!



Scaling Instruction-Level Parallelism

Pengju Ren@XJTU 2023

Recall from last time: ILP & pipelining tapped out... why?



Superscalar scheduling is complex & hard to scale

- **Q: When is it safe to issue two instructions?**
- **A: When they are independent**
 - Must compare all pairs of input and output registers
- **Scalability: $O(W^2)$ comparisons where W is “issue width” of processor**
 - Not great!

Limitations of ILP

- 4-wide superscalar $\times$ 20-stage pipeline = 80 instrns in flight
- High-performance OoO buffers *hundreds* of instructions
- **Programs have limited ILP**
 - Even with perfect scheduling, >8-wide issue doesn't help
- Pipelines can only go so deep
 - Branch misprediction penalty grows
 - Frequency (GHz) limited by power
- Dynamic scheduling overheads are significant
- Out-of-order scheduling is expensive

Limitations of ILP → SIMD\Multithread\Multicore

- ILP works great! ...But is complex + hard to scale
- From hardware perspective, multicore is much more efficient, but needs programmer's effort based on the knowledge about underlying architecture.
- Parallel software is hard!
 - Industry resisted multicore for as long as possible
 - When multicore finally happened, CPU μ arch simplified → more cores
 - Many program(mer)s still struggle to use multicore effectively

*Next Lecture : Understanding Modern
Processor : DLP and TLP*