Computer Architecture

Lecture 03 – Pipeline and hazard (Instruction level Parallelism)

Pengju Ren
Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University

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

Agenda

Pipeline and hazards:

— Pipeline Basics

— Structural Hazards

— Data Hazards

— Control Hazards

"Iron Law" of Processor Performance

Algorithm\Compiler\ISA Extension

Workload of a Program (how many line of codes)

Microarchitecture

CPI

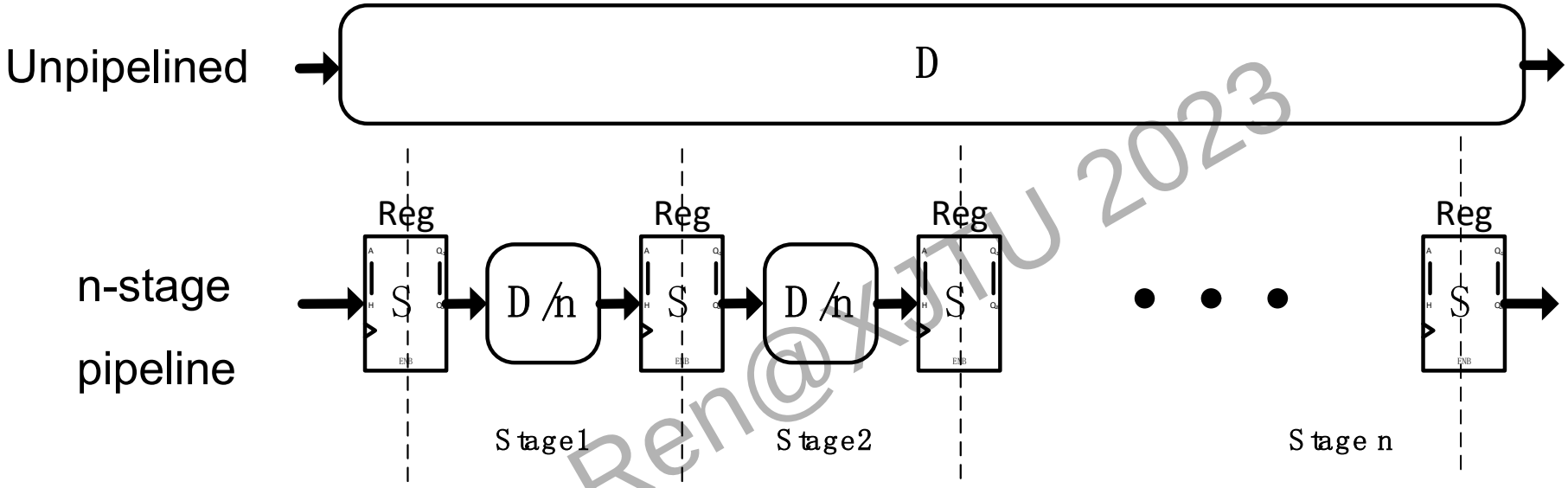
manufacture Process (e.g. 7nm, 28nm)

Clock Freq

$$\textit{Total Instructions} \times \frac{\textit{Cycles}}{\textit{Instruction}} \times \frac{\textit{Seconds}}{\textit{Cycle}}$$

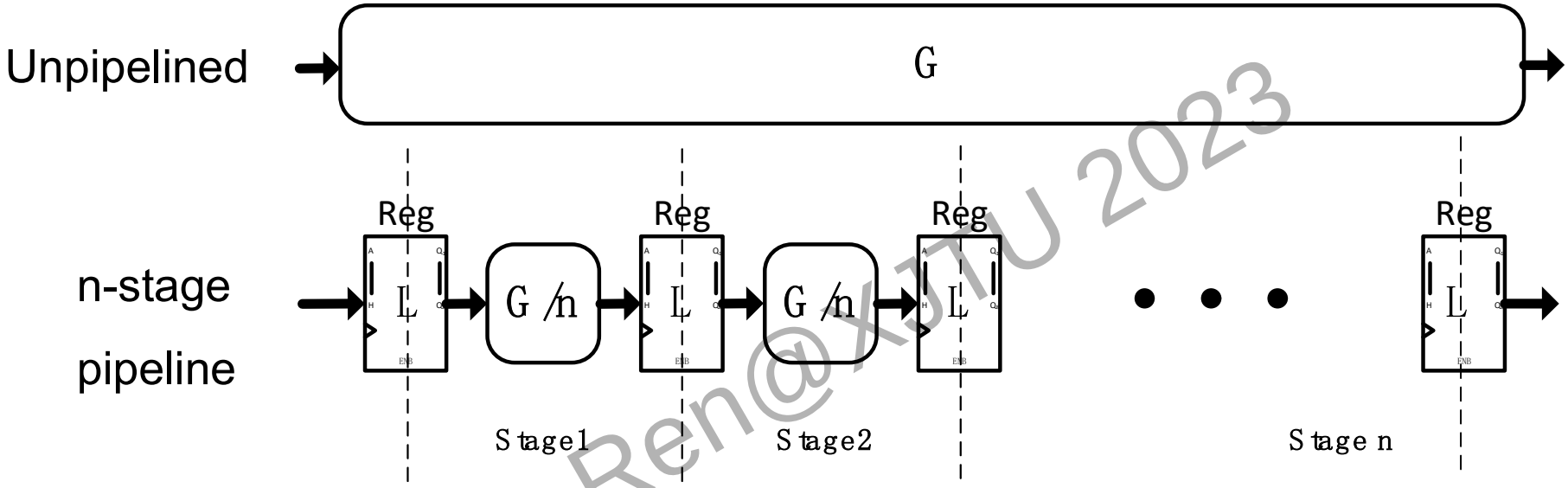
- Instructions per program depends on source code, compiler technology, and ISA
- Cycles per instructions (CPI) depends on ISA and microarchitecture
- Time per cycle depends upon the microarchitecture and base technology

Pipeline v.s Unpipeline (Timing analysis)



	Execution Time	Freq
Unpipelined	**D**	**1/D**
n-stage Pipeline	**D/N+S**	**1/(D/n+S)**

Pipeline v.s Unpipeline (Area analysis)



	Area Cost	Cost/Performance
Unpipelined	**G**	**G*D**
n-stage Pipeline	**G+n*L**	**(G+n*L)*(D/n+S)**

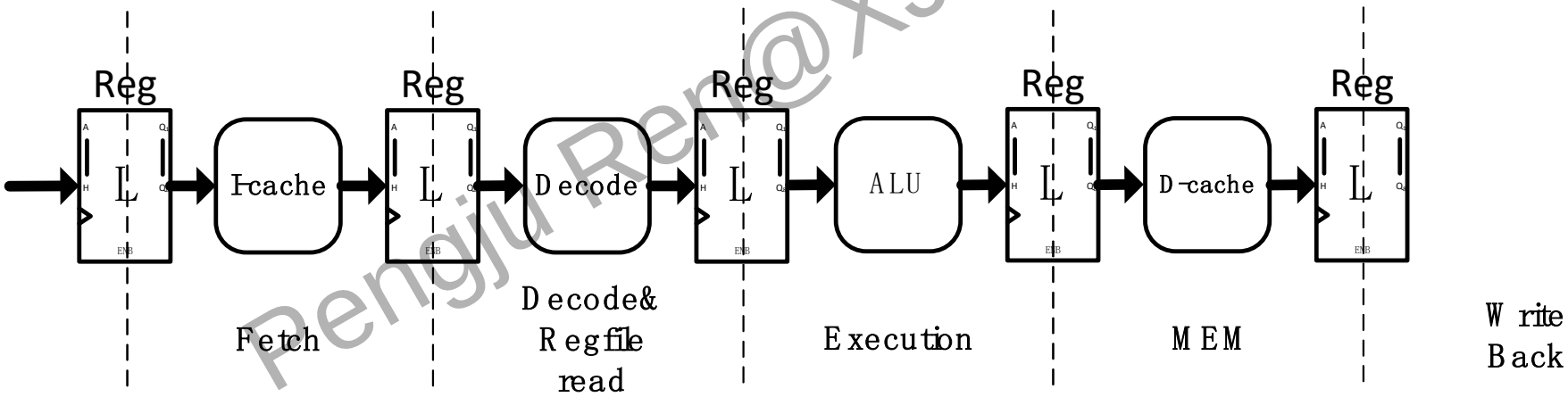
$$f(x) = x + a/x$$

GD/n+SL*n+LD+GS

$$\frac{d(\frac{GD}{n} + SL * n + LD + GS)}{dn} = SL - \frac{GD}{n^2} \quad \Rightarrow \quad n = \sqrt{\frac{GD}{LS}}$$

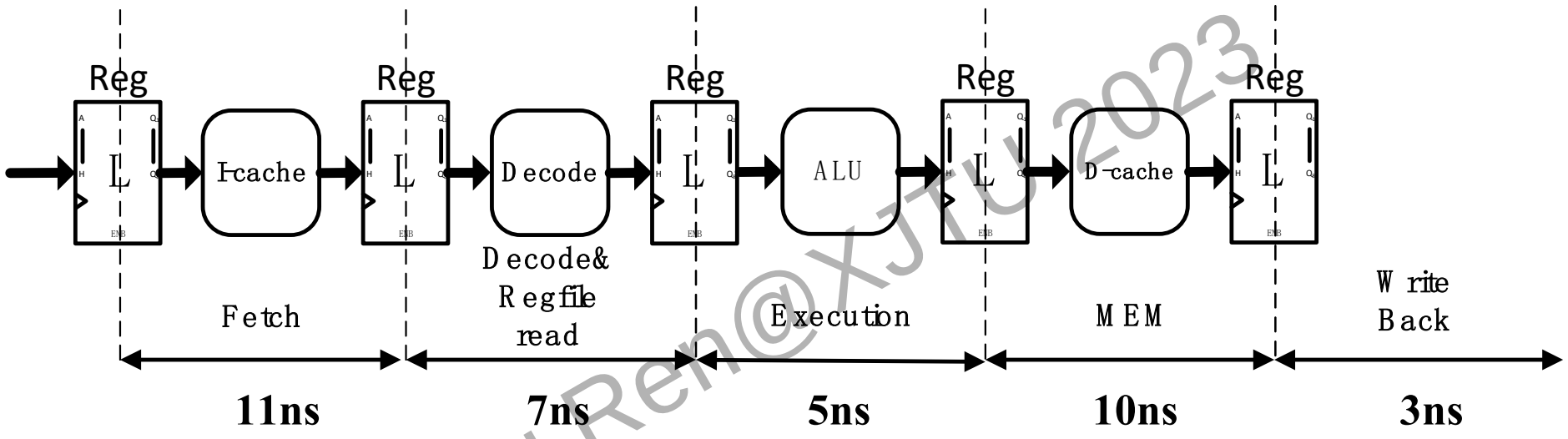
More about Pipeline

- **The Clock Period is depended on the longest stage of the pipeline**
- **Dependence among different stages raise challenges for high efficient pipeline (e.g., RAW, WAW, WAR)**



The classical 5-stages Pipeline of RISC-V

Clock Frequency of Pipeline

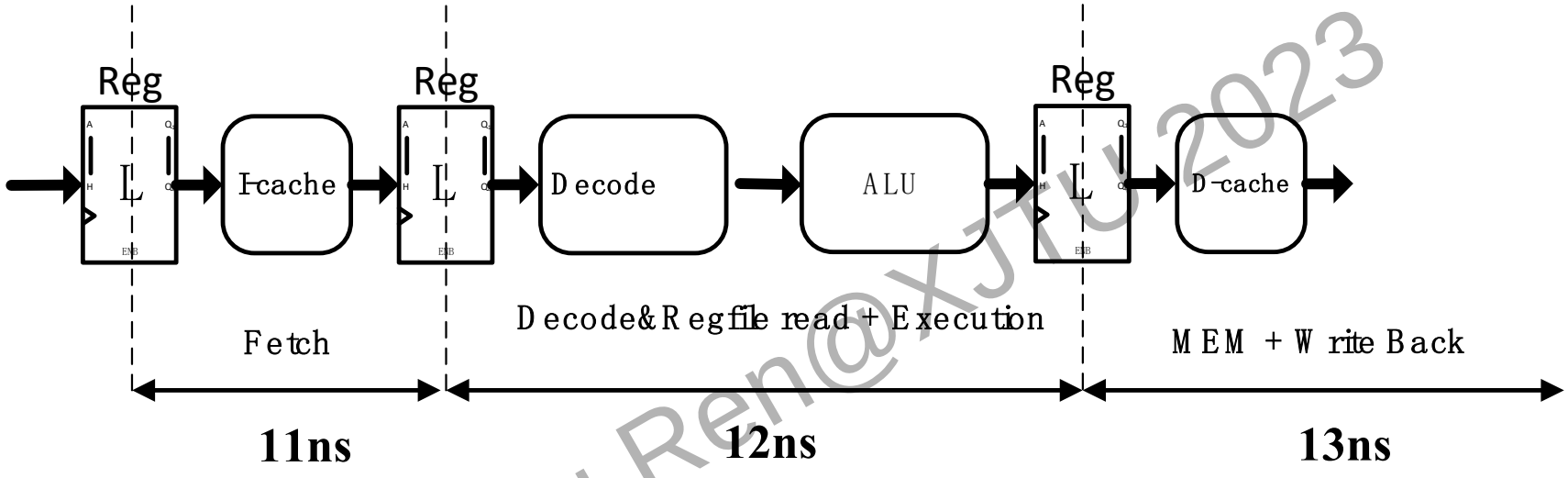


Unpipeline: T = 11+7+5+10+3-0.5*5 = 33.5ns

Pipeline: T = max(11, 7, 5, 10, 3) = 11ns

NOTEs: Assuming the time of the Reg operations of each stage is 0.5ns

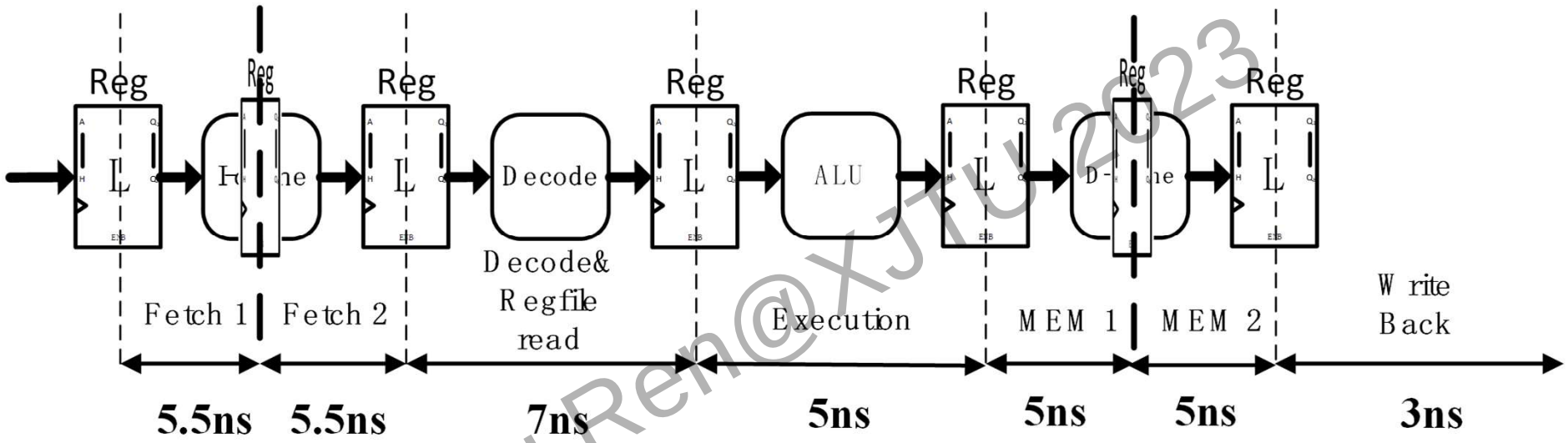
Merge multiple stages into one (Shallow pipeline)



Pipeline: T = max(11,12,13) = 13ns

	Area Cost	Freq
5-stages	G+5*L	1/11ns=90.9Mhz
3-stages	G+3*L	1/13ns=76.9Mhz

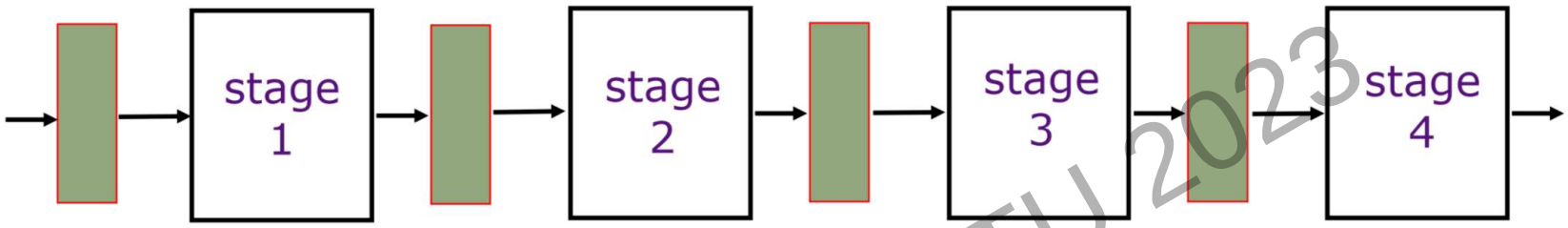
Divide one stage into multiple stages (Deeper pipeline)



Pipeline: T = max(5.5,5.5,7,5,5,3) = 7ns

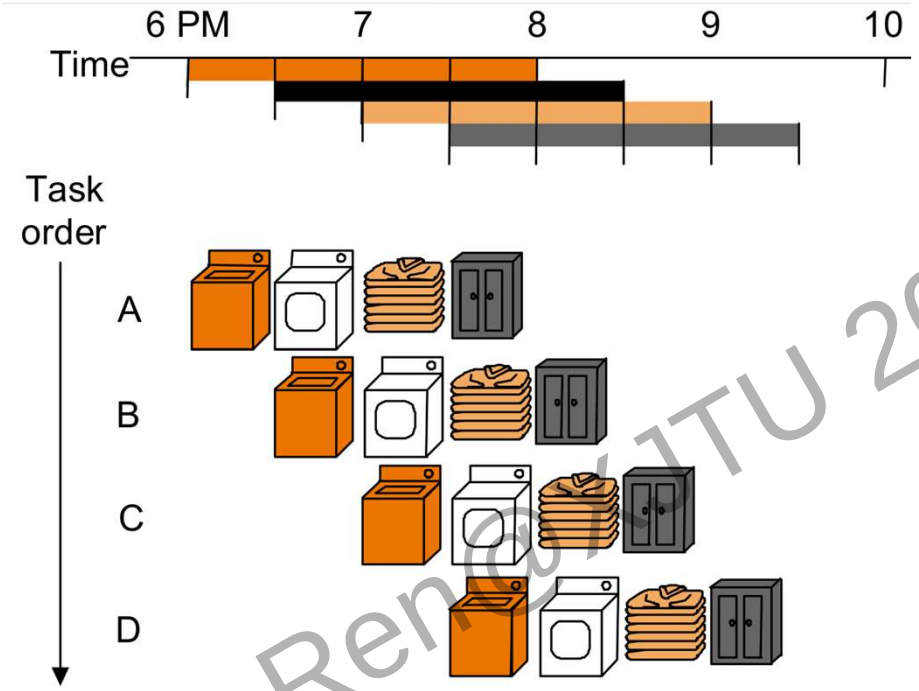
	Area Cost	Freq
5-stages	G+5*L	1/11ns=90.9Mhz
7-stages	G+7*L	1/7ns=142.8Mhz

An Ideal Pipeline



- All instructions go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- Scheduling of a transaction entering the pipeline is not affected by the transactions in other stages
- These conditions generally hold for industry assembly lines, but instructions depend on each other causing various hazards

An Ideal Pipeline



- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- Scheduling of a transaction entering the pipeline is not affected by the transactions in other stages
- These conditions generally hold for industry assembly lines, but instructions depend on each other causing various hazards

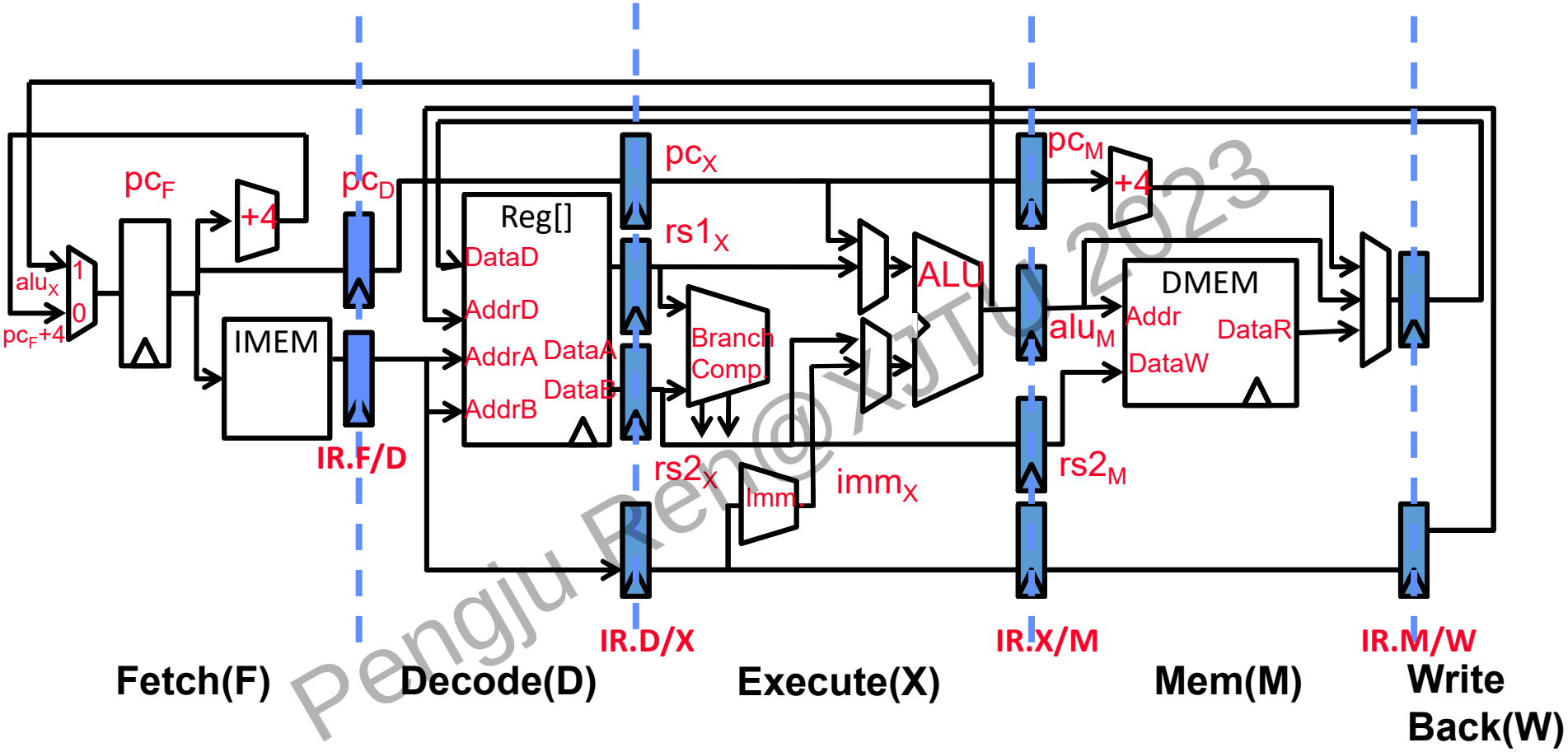
Instructions Interact With Each Other in Pipeline

- **Structural Hazard:** An instruction in the pipeline needs a resource being used by another instruction in the pipeline
- **Data Hazard:** An instruction depends on a data value produced by an earlier instruction
- **Control Hazard:** Whether or not an instruction should be executed depends on a control decision made by an earlier instruction (branches, interrupts)

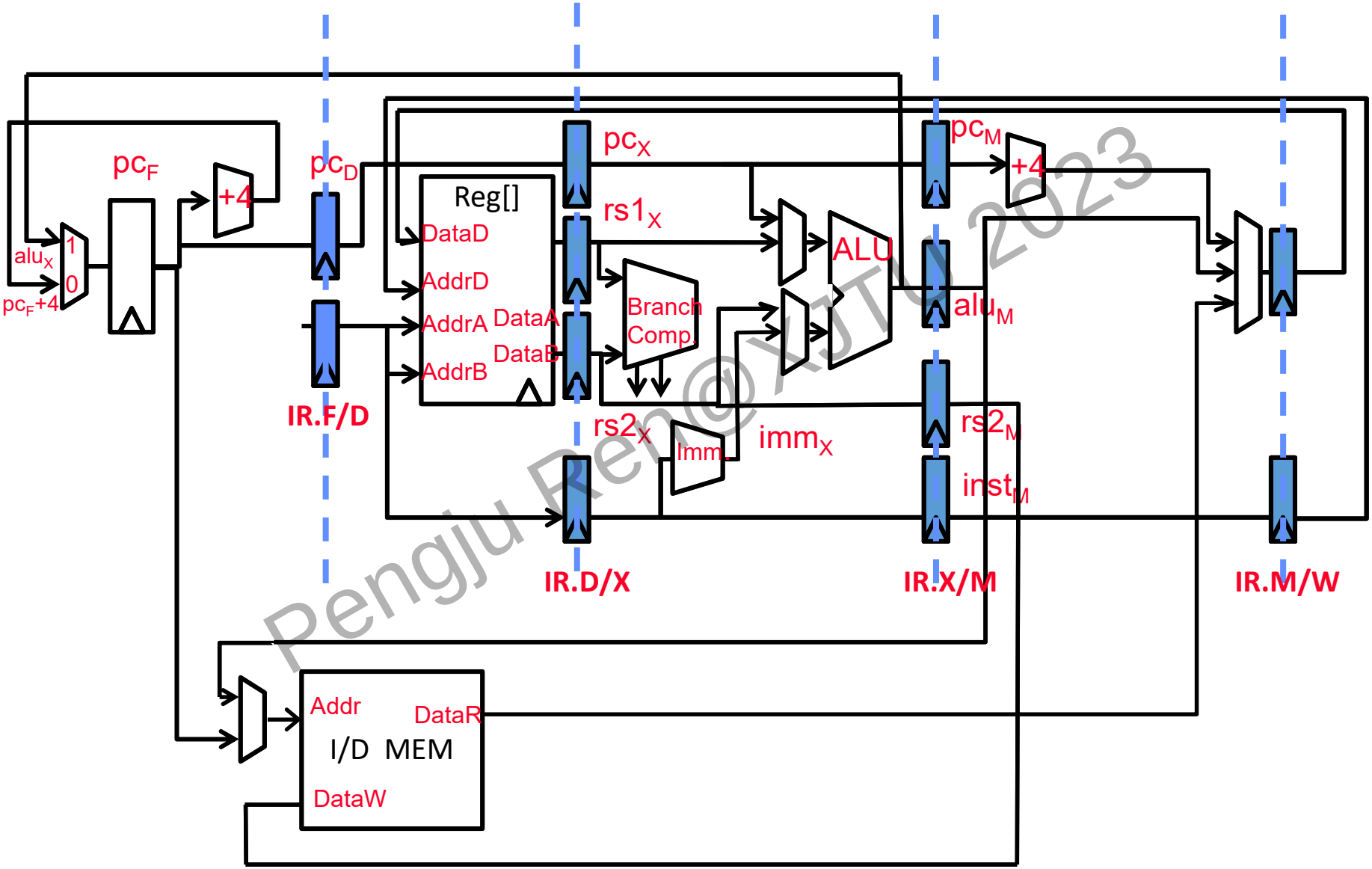
Overview of Structural Hazard

- Structural hazards occur when two instructions need the same hardware resource at the same time
- Approaches to resolving structural hazards
  – **Schedule**: Programmer explicitly avoids scheduling instructions that would create structural hazards
  – **Stall:** Hardware includes control logic that stalls until earlier instruction is no longer using contended resource
  – **Duplicate:** Add more hardware to design so that each instruction can access independent resources at the same time

Example of Structural Hazard: Unified Memory



Example of Structural Hazard: Unified Memory

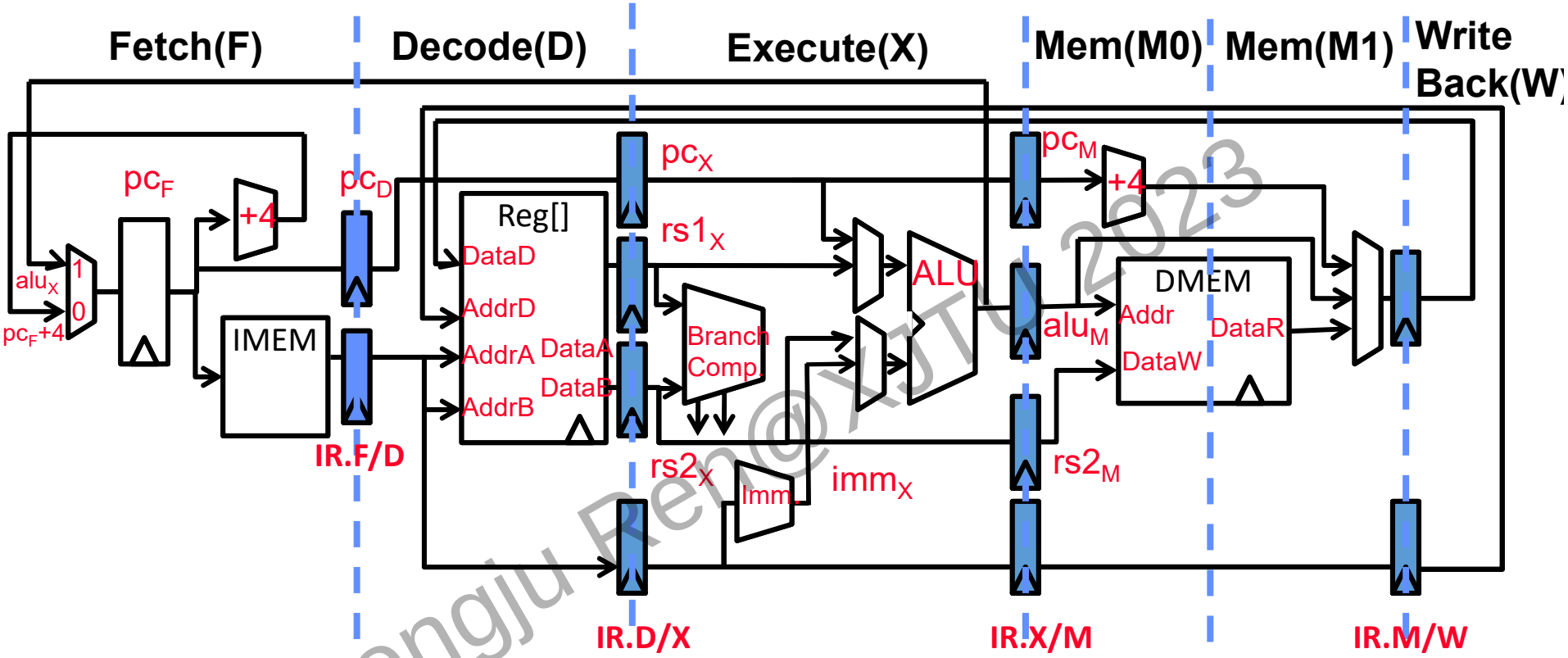


Example of Structural Hazard: Unified Memory

time	t0	t1	t2	t3	t4	t5	t6	t7	t8
LW	F	D	X	M	W				
ADD		F	D	E	M	W			
ADD			F	D	E	M	W		
SUB				F	D	E	M	W	
ST					F	D	E	M	W

I/D Mem can not be R/W at the same time, is component conflict.

Example of Structural Hazard: Two Cycle Mem



time	t0	t1	t2	t3	t4	t5	t6	t7	t8
LW	F	D	X	M0	M1	W			
ADD X1, X2, X3		F	D	E	M0	M1	W		
LW			F	D	E	M0	M1	W	
ST				F	D	E	M0	M1	W

A scoreboard records the resource occupation can assist to detect structural hazard

Agenda

Pipeline and hazards:

- Pipeline Basics
- Structural Hazards
- Data Hazards
- Control Hazards

Overview of Data Hazards

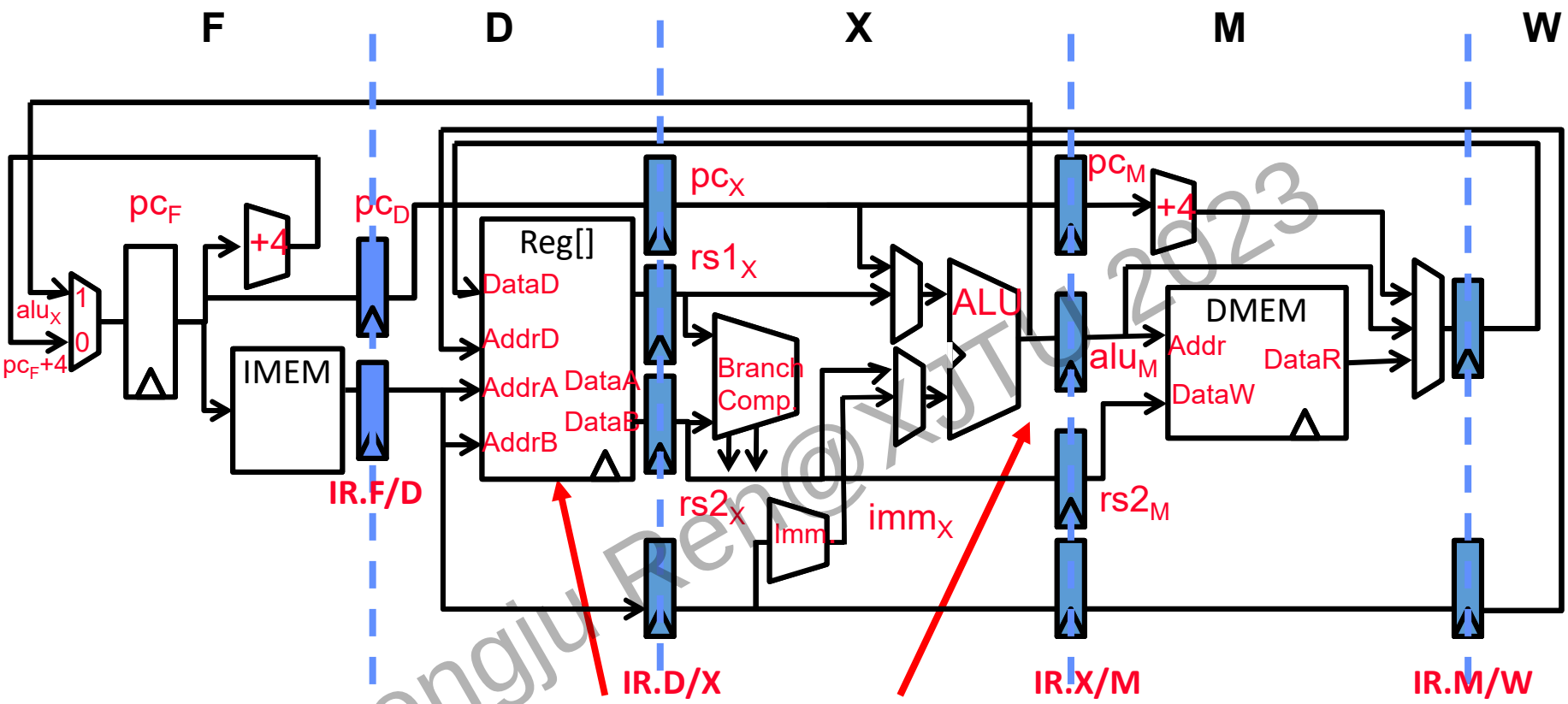
- Data hazards occur when one instruction depends on a data value produced by a preceding instruction still in the pipeline
- Approaches to resolving data hazards
  - **Stall:** Wait for the result to be available by freezing earlier pipeline stages
  - **Bypass:** Route data as soon as possible after it is calculated to the earlier pipeline stage
  - **Speculate:**

    Two cases:

    Guessed correctly -> do nothing

    Guessed incorrectly -> kill and restart

Example of Data Hazards



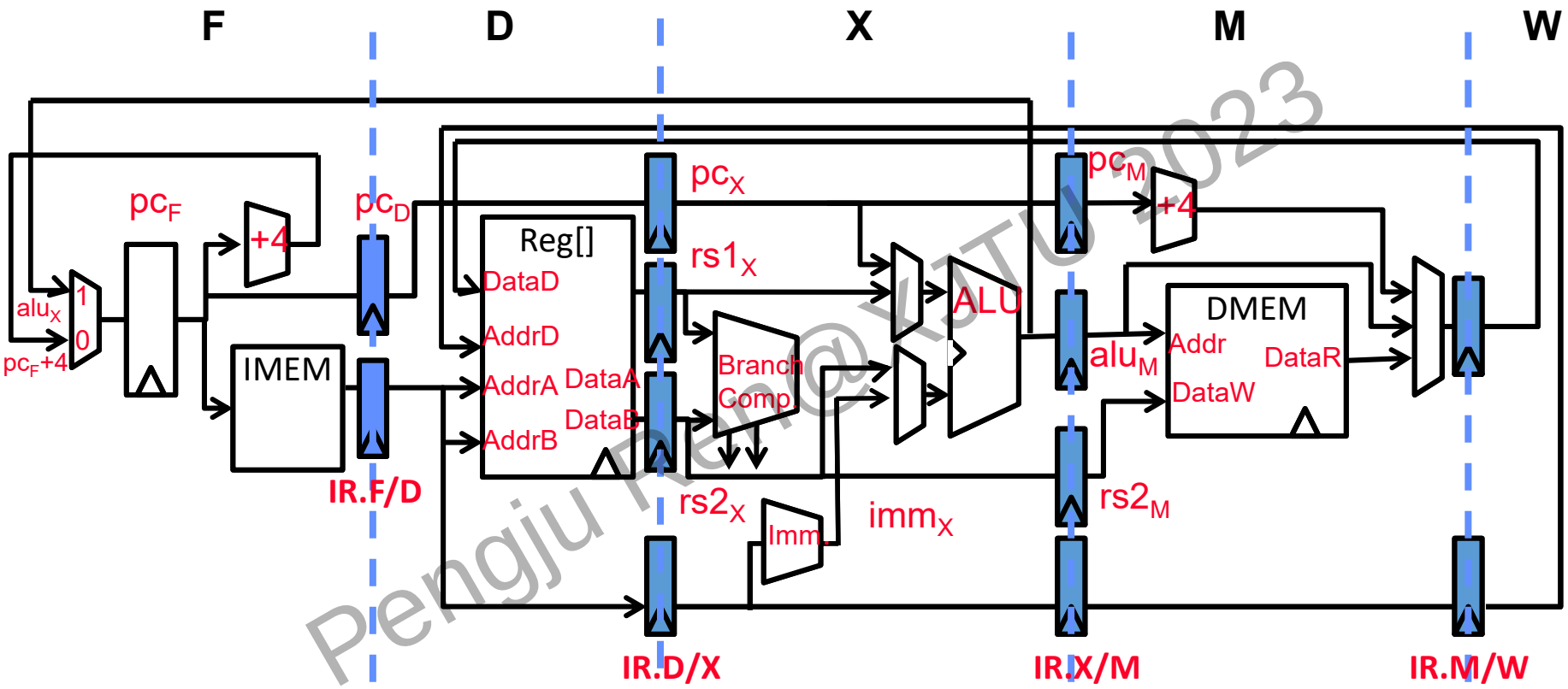
X2: X1 is stale X1=X2+10

X1 ← X2 + 10 (ADDI X1, X2, #10)

X4 ← X1 + 17 (ADDI X4, X1, #17)

time	t0	t1	t2	t3	t4
ADDI X1, X2, #10	F	D	X	M	W
ADDI X4, X1, #17		F	D	E	M

Resolving Data Hazards by Stalling



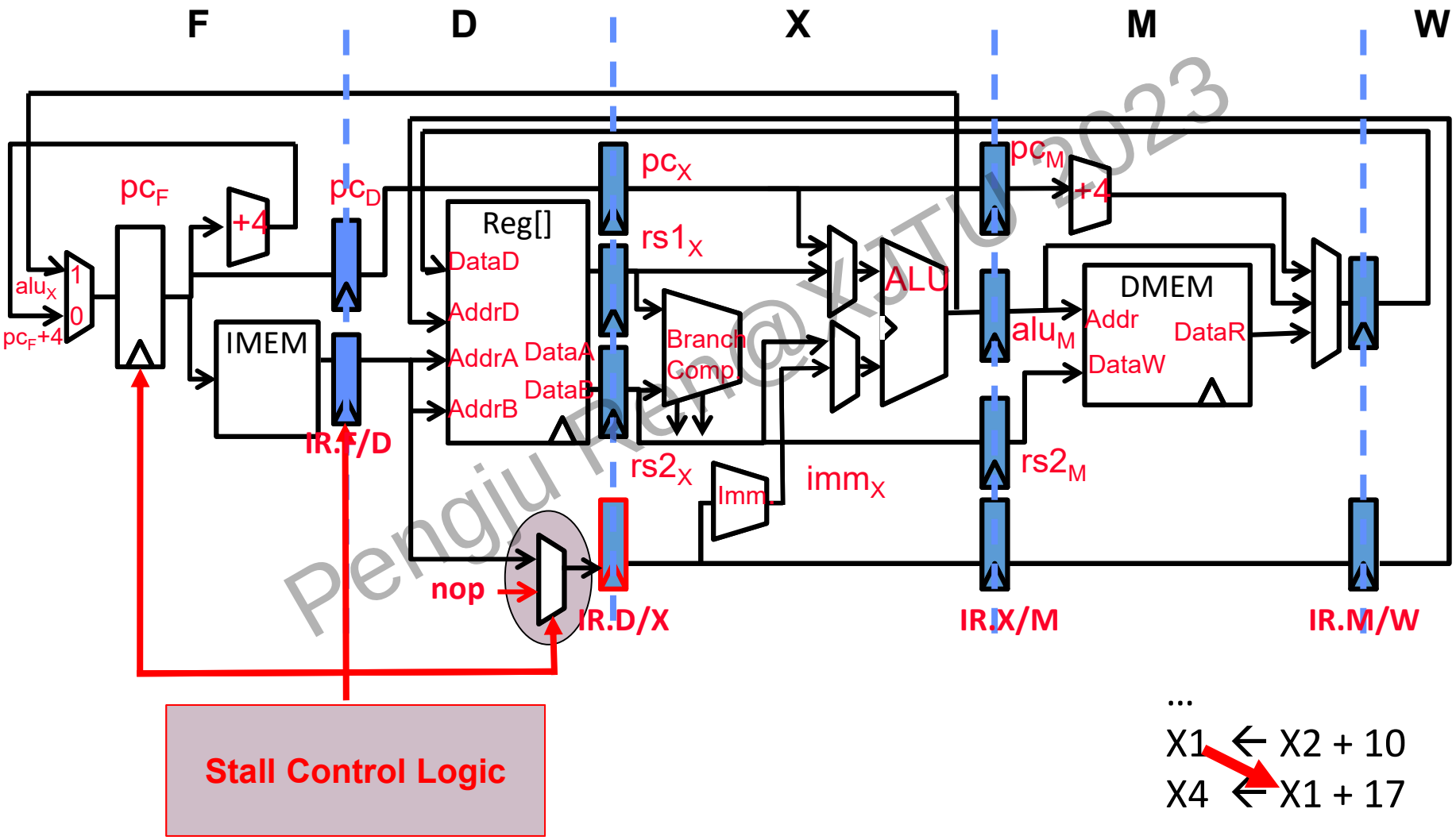
...

X1 ← X2 + 10

X4 ← X1 + 17

...

Resolving Data Hazards by Stalling



…

X1 ← X2 + 10

X4 ← X1 + 17

…

Stalled Stages and Pipeline Bubbles

	t0	t1	t2	t3	t4	t5	t6	t7	…	
(I_1) **X1<-(X2)+10**	IF_1	ID_1	EX_1	MA_1	WB_1					
(I_2) **X4<-(X1)+17**		IF_2	ID_2	ID_2	ID_2	EX_2	MA_2	WB_2		
(I_3)			IF_3	IF_3	IF_3	ID_3	EX_3	MA_3	WB_3	
(I_4)										

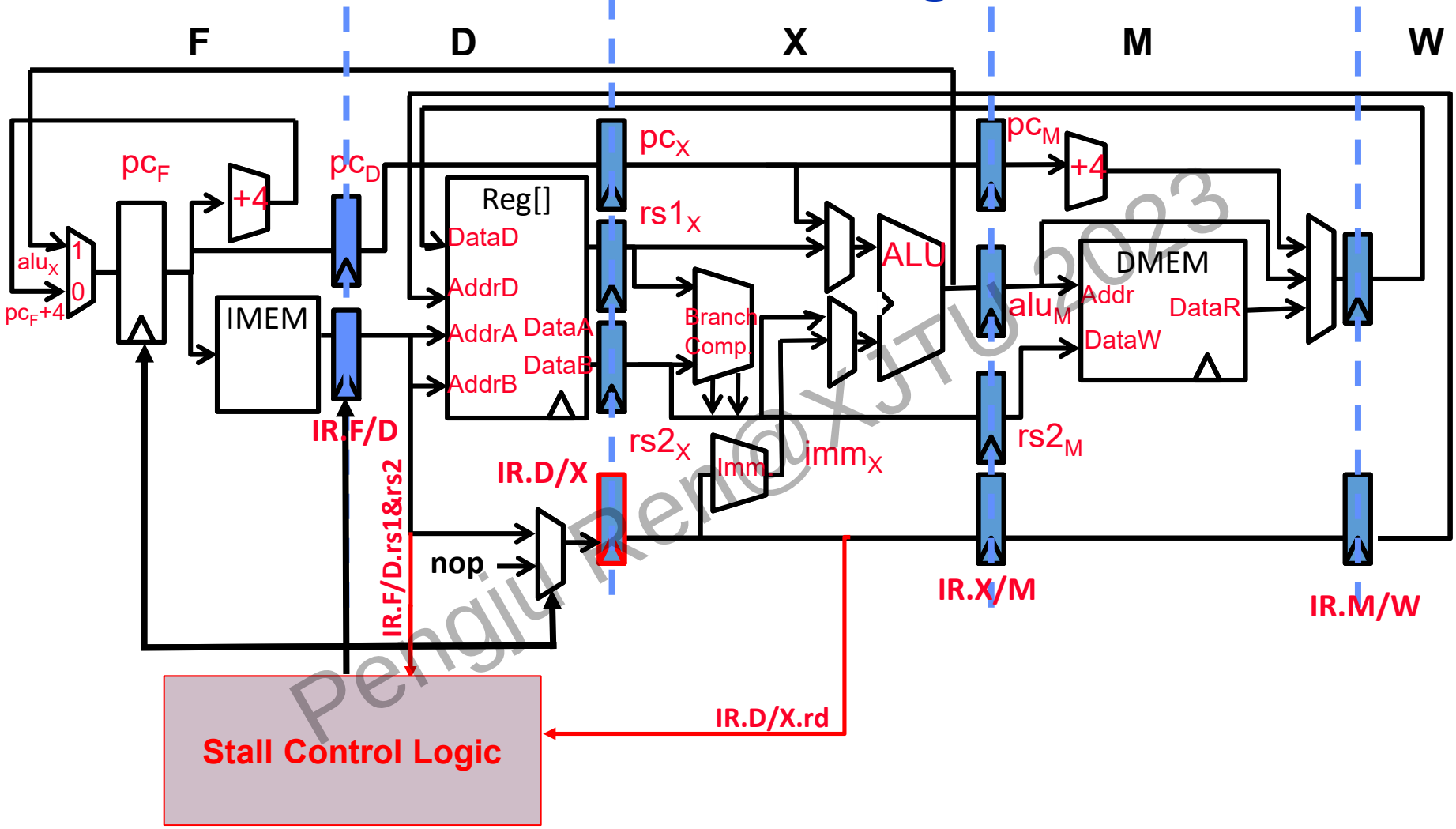
Stall stages

	t0	t1	t2	t3	t4	t5	t6	t7	…	
IF	I_1	I_2	I_3	I_3	I_3	I_4	I_5			
ID		I_1	I_2	I_2	I_2	I_3	I_4	I_5		
EX			I_1	nop	nop	I_2	I_3	I_4	I_5	
MA				I_1	nop	nop	I_2	I_3	I_4	
WB					I_1	nop	nop	I_2	I_3	I_4

nop => pipeline bubble

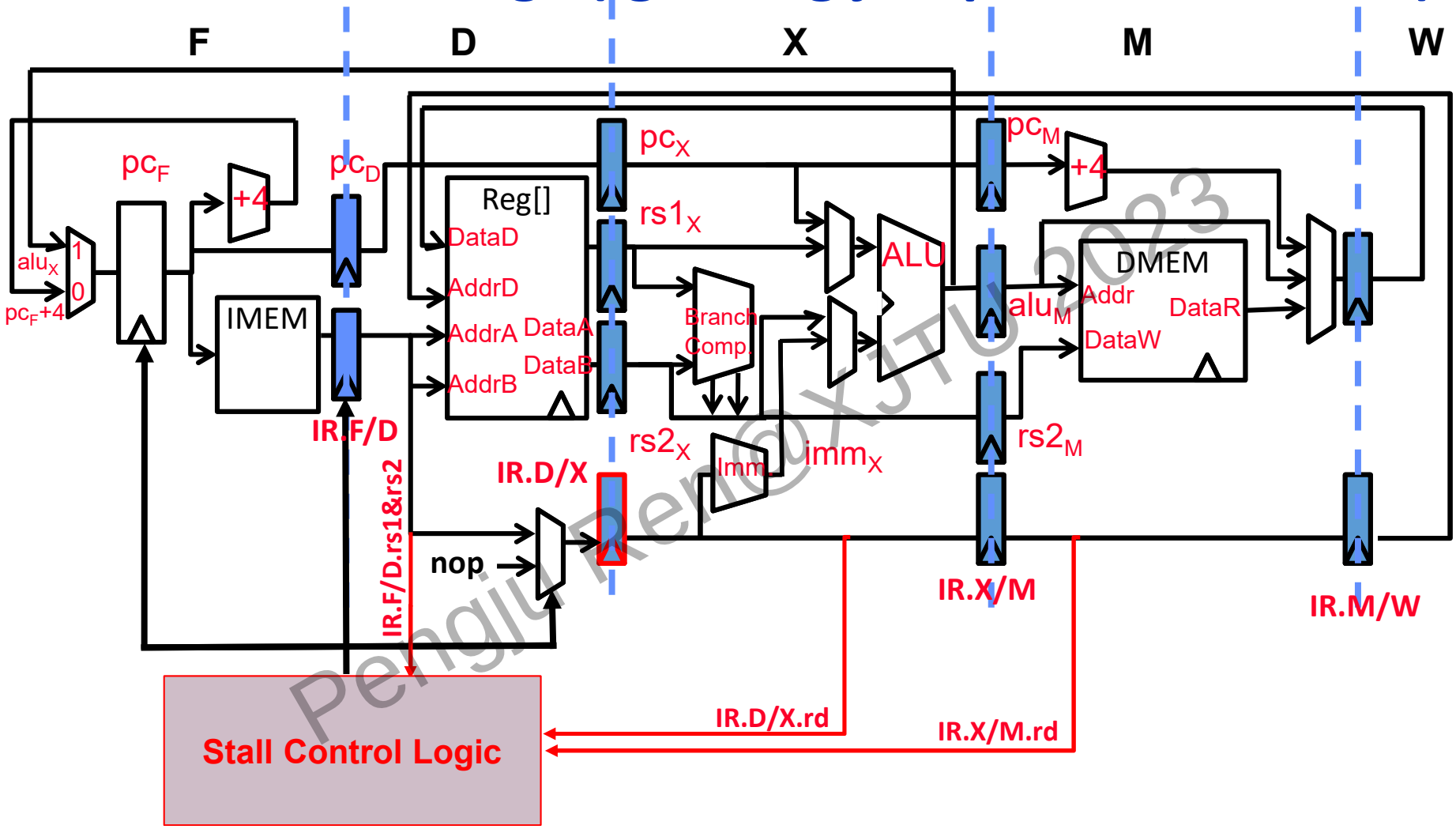
…
X1 ← X2 + 10
X4 ← X1 + 17
…

Stall Control Logic



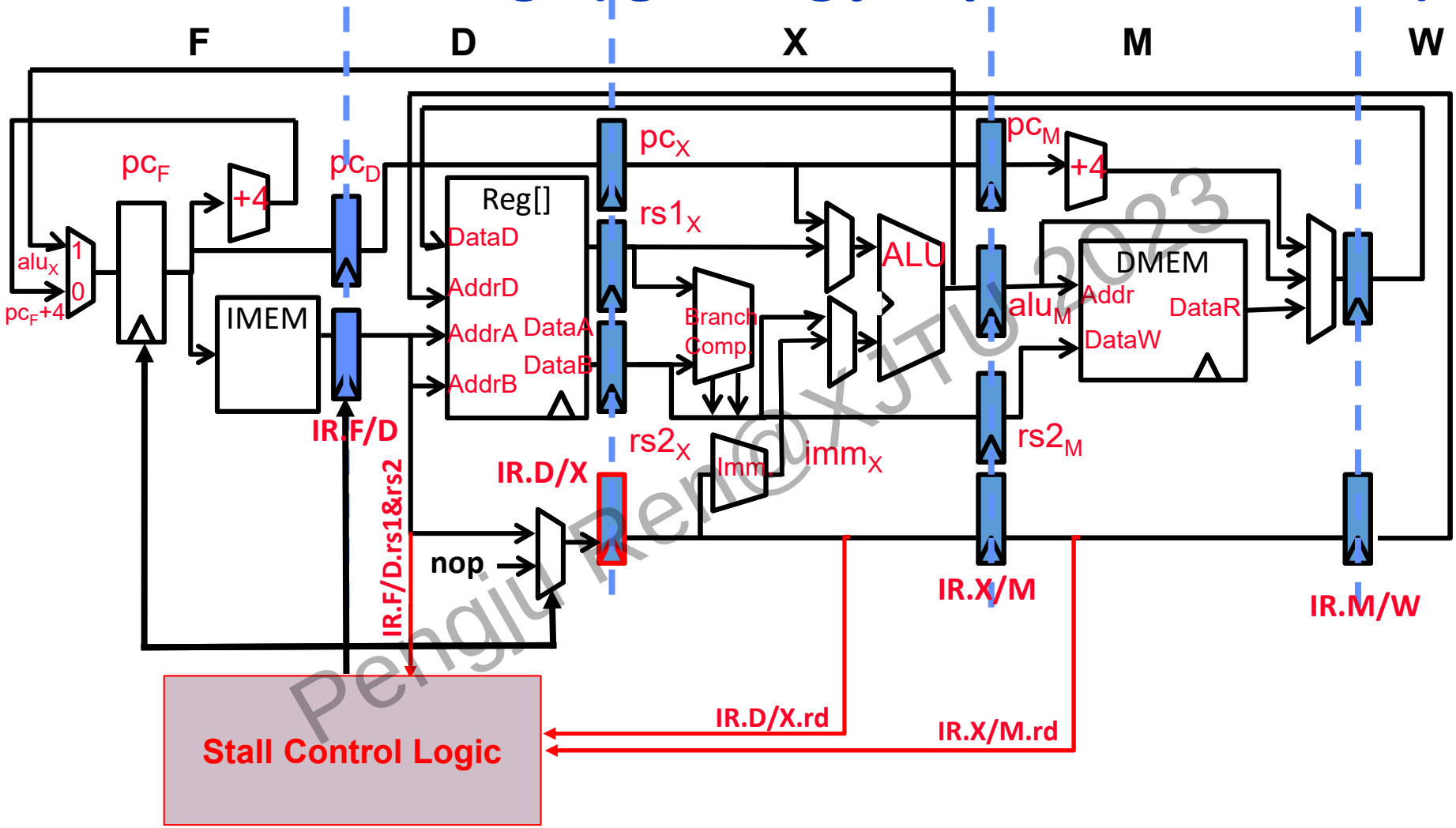
Compare the **source registers** of the instruction at IR.F/D with the **destination register** of the instruction at IR.D/X (uncommitted instructions).

Stall Control Logic(Ignoring jumps and branches)



Why do not compare IR.F/D.rs1 and rs2 with IR.M/W.rd?

Stall Control Logic(Ignoring jumps and branches)



Should we always stall if the *rs* field matches some *rd*?

not every instruction writes a register => *we*

not every instruction reads a register => *re*

Source & Destination Registers (RISC-V)

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

Inst-Type	Funct	Source(s)	Destination	Result ready
R	rd <- (rs1) func (rs2)	rs1,rs2	rd	E(X)-stage
I	rd <- (rs1) op immediate	rs1	rd	E(X)-stage
	lw、lb、load		rd	**M-stage**
	jalr		rd	**M-stage**
S	M[(rs1)+immediate] <- (rs2)	rs1,rs2	-	-
SB	cond (rs1, rs2): ■ True: PC <- PC + immediate ■ False: PC <- PC + 4	rs1,rs2	-	-
U	LUI rd <- Imm.gen(immediate)		rd	E(X)-stage
UJ	rd <- PC + 4, PC <- PC +immediate		rd	**M-stage**

Deriving the Stall Signal

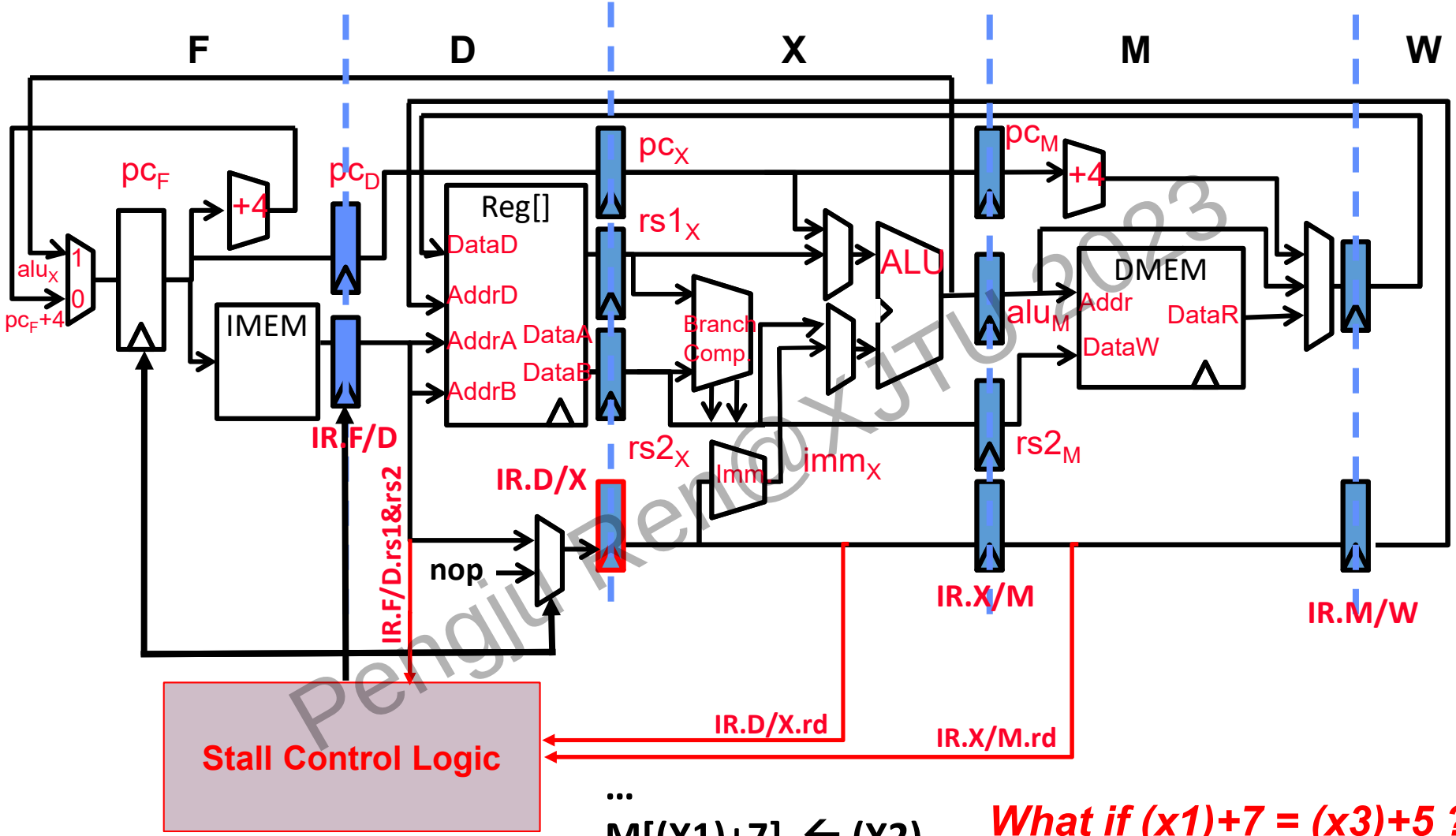
rd depends on opcode
R,I,U,J rd≠0
Others rd=0
we depends on opcode
R,I,U,J we=on
Others we=off

re1 depends on opcode
R,I,S,B re1=on
Others re1=off
re2 depends on opcode
R,S,B re2=on
Others re2=off

*Stall =((IR.F/D.rs1==IR.D/X.rd)**IR.D/X.we***
*+ (IR.F/D.rs1==IR.X/M.rd)**IR.X/M.we**) **IR.F/D.re1***
or
*((IR.F/D.rs2==IR.D/X.rd)**IR.D/X.we***
*+ (IR.F/D.rs2==IR.X/M.rd)**IR.X/M.we**) **IR.F/D.re2***

This is not the whole story!

Data Hazards due to Loads & Stores



…

M[(X1)+7] ← (X2)

X4 ← M[(X3) + 5]

…

What if (x1)+7 = (x3)+5 ?

Is there any possible data hazard in this instruction sequence?

Data Hazards Due to Loads and Stores

- **Example instruction sequence：**

  M[(X1)+7] ← (X2)
  X4 ← M[(X3) + 5]

- **What if Regs[X1]+7 == Regs[X3]+5 ?**

  — Writing and reading to/from the same address
  — Hazard is avoided because our memory system completes writes in a single cycle *(Actually it is not)*
  — More realistic memory system will require more careful handling of data hazards due to loads and stores *(More on this later in the course)*

Overview of Data Hazards

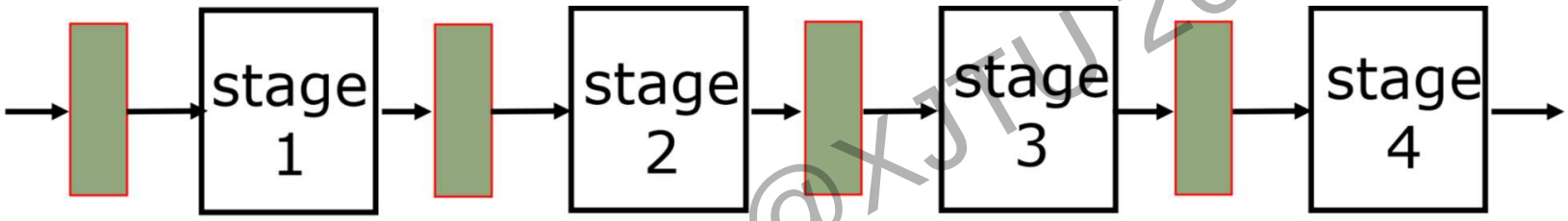
- Data hazards occur when one instruction depends on a data value produced by a preceding instruction still in the pipeline
- Approaches to resolving data hazards
  - **Stall:** Wait for the result to be available by freezing earlier pipeline stages
  - **Bypass**: Route data as soon as possible after it is calculated to the earlier pipeline stage
  - **Speculate:**

    Two cases:

    Guessed correctly -> do nothing

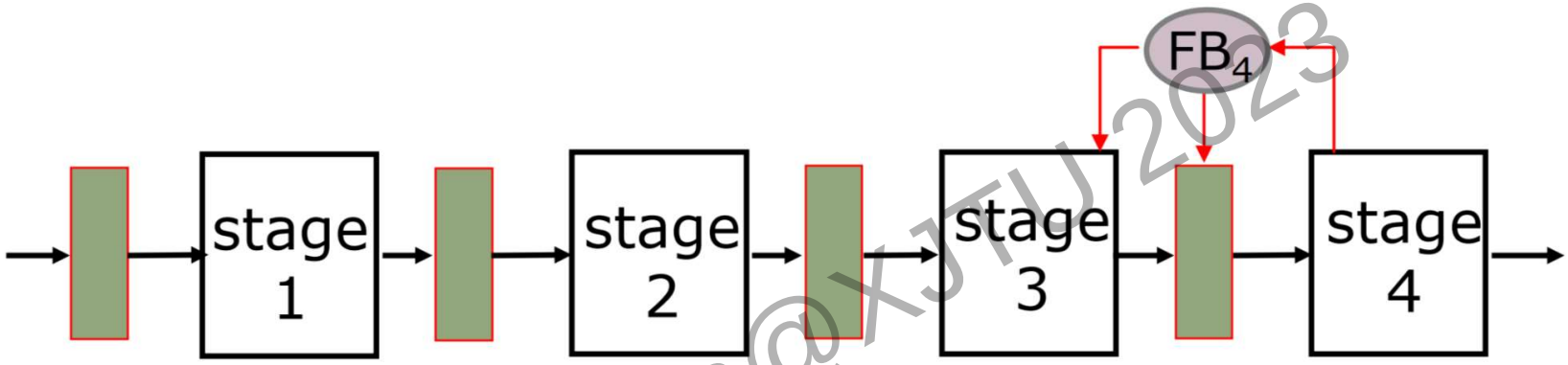
    Guessed incorrectly -> kill and restart

Feedback to Resolve Hazards



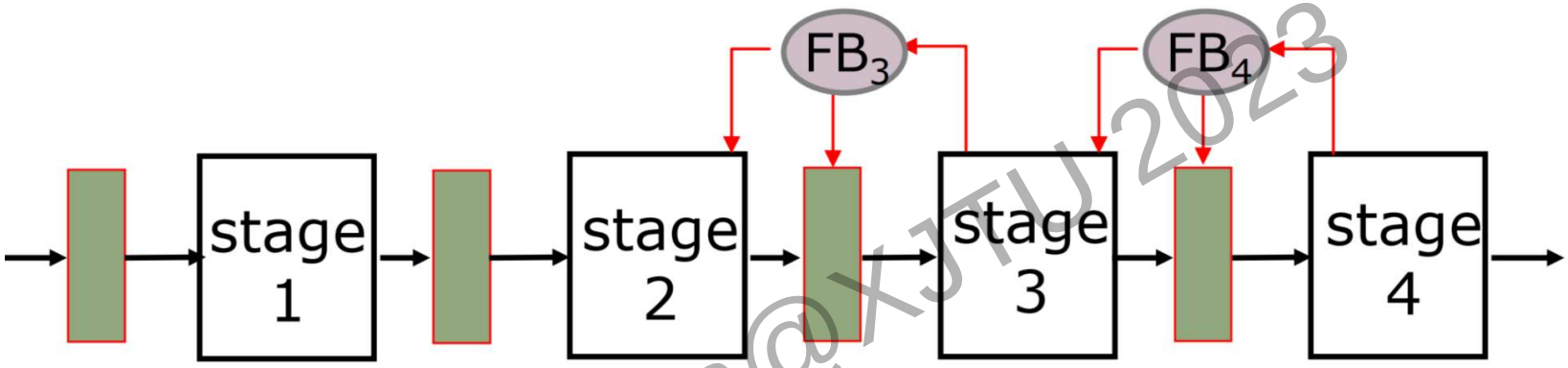
- Later stages provide dependence information to earlier stages which can stall (or kill) instructions

Feedback to Resolve Hazards



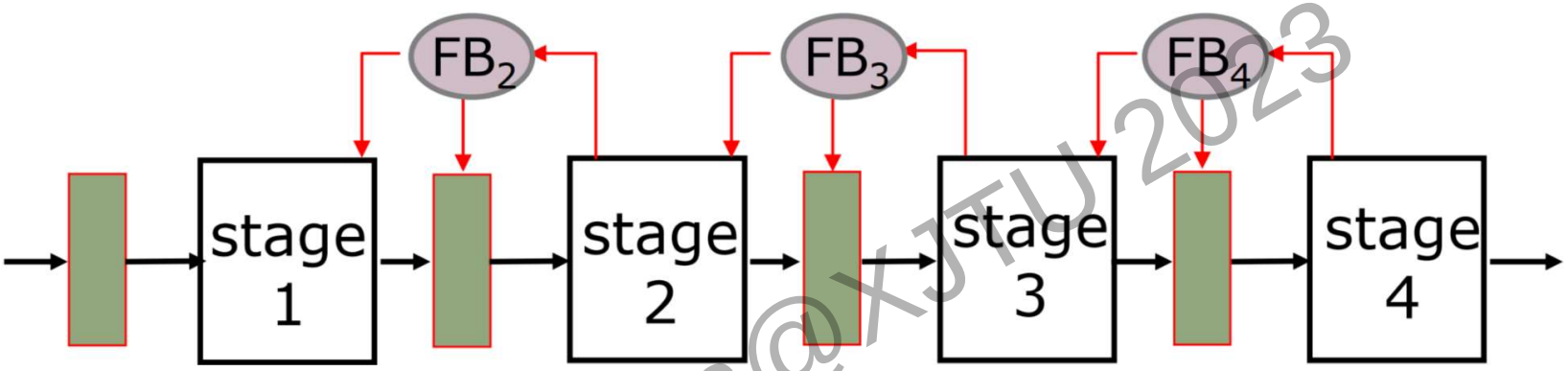
- Later stages provide dependence information to earlier stages which can stall (or kill) instructions

Feedback to Resolve Hazards



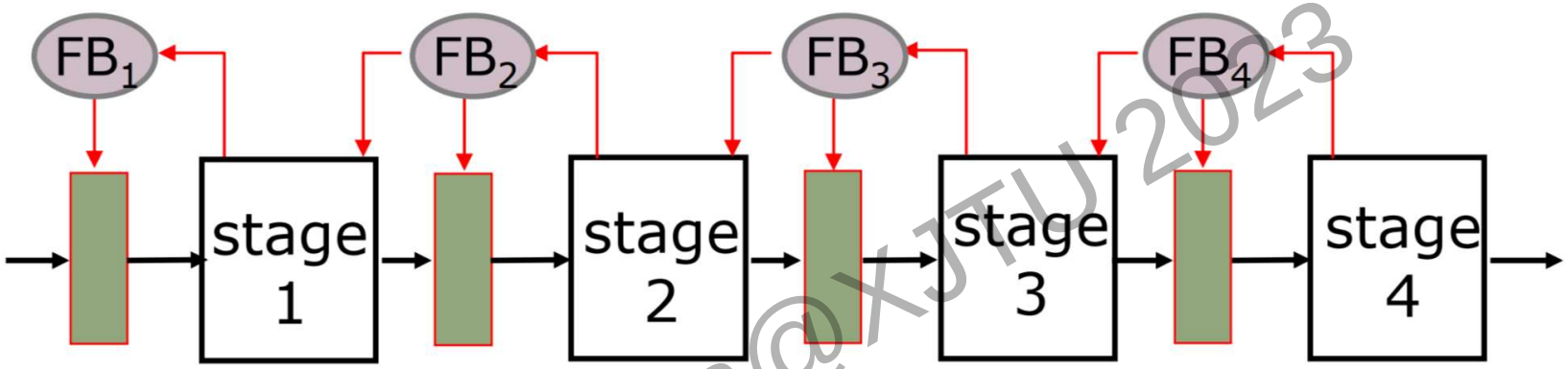
- Later stages provide dependence information to earlier stages which can stall (or kill) instructions

Feedback to Resolve Hazards



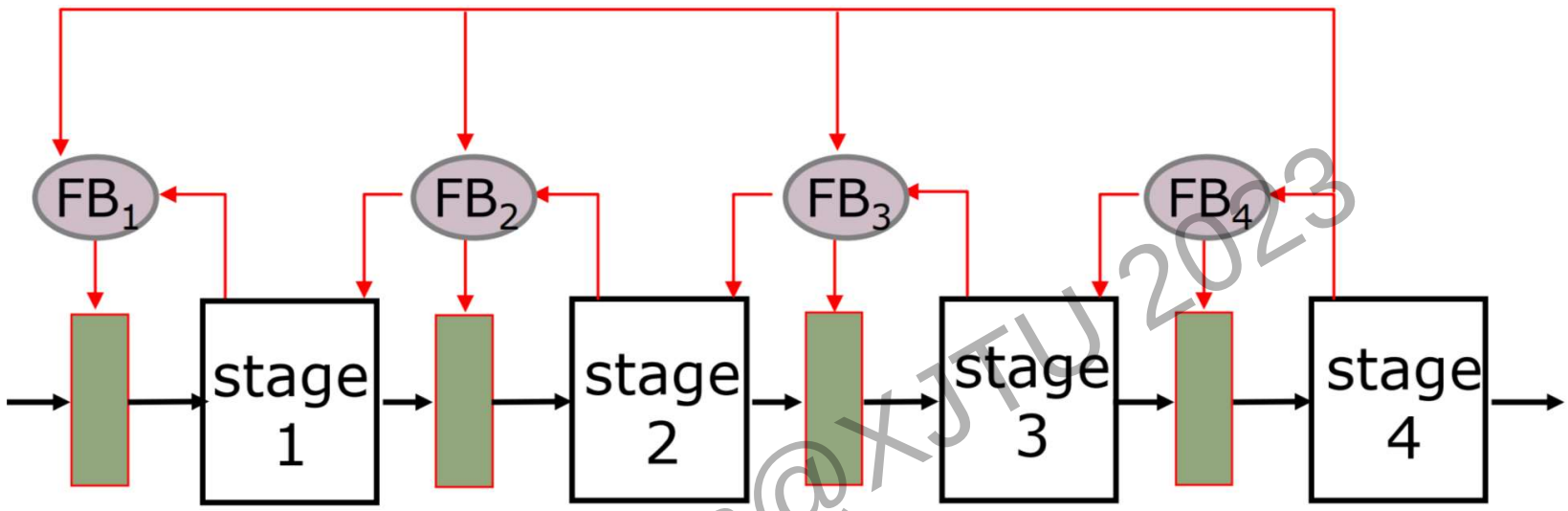
- Later stages provide dependence information to earlier stages which can stall (or kill) instructions

Feedback to Resolve Hazards



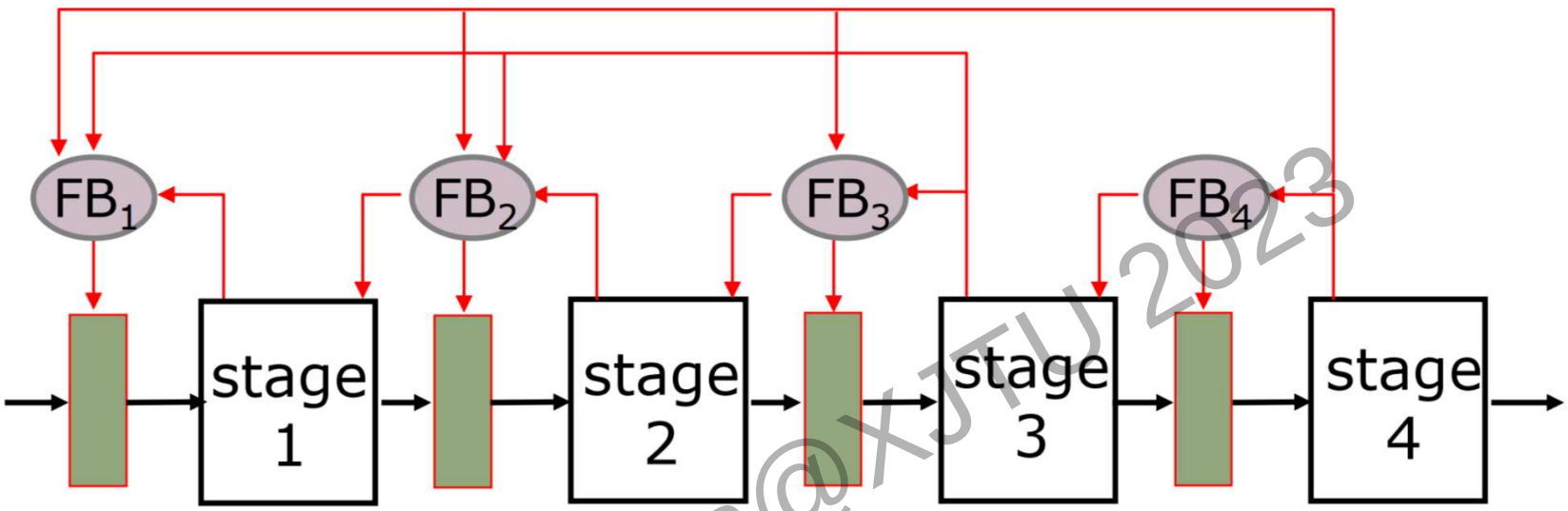
- Later stages provide dependence information to earlier stages which can stall (or kill) instructions

Feedback to Resolve Hazards



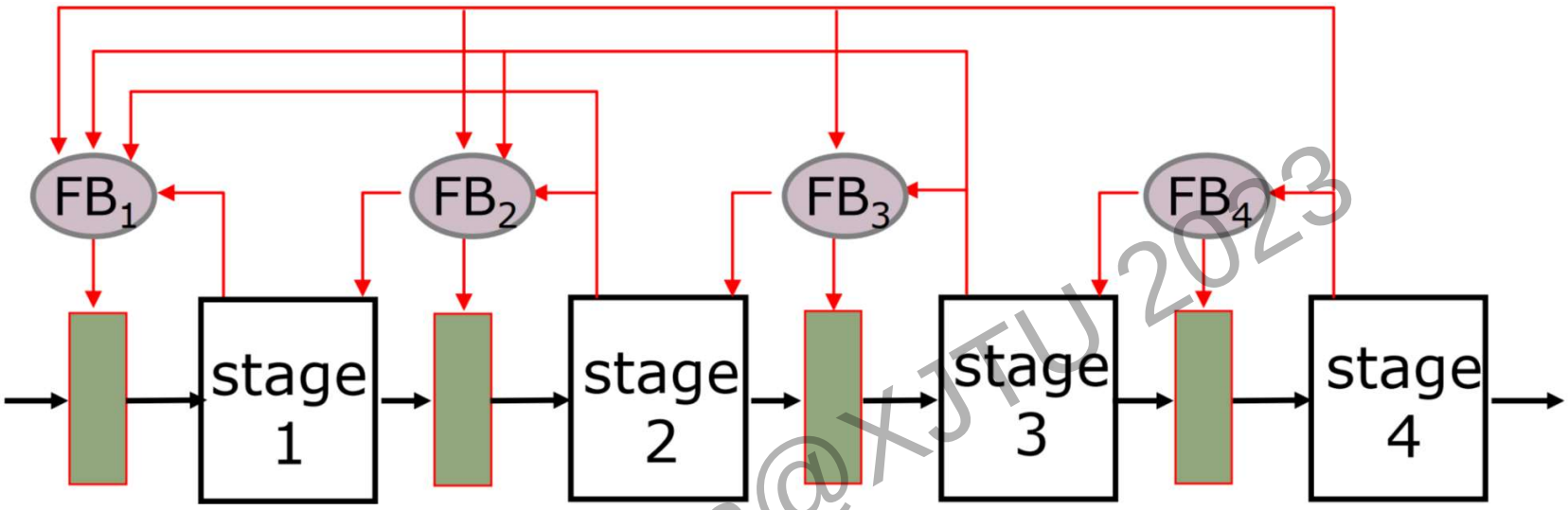
- Later stages provide dependence information to earlier stages which can stall (or kill) instructions

Feedback to Resolve Hazards



- Later stages provide dependence information to earlier stages which can stall (or kill) instructions

Feedback to Resolve Hazards



- Later stages provide dependence information to earlier stages which can stall (or kill) instructions
- Controlling a pipeline in this manner works provided the instruction at stage i+1 can complete without any interference from instructions in stages 1 to i

Bypassing

	t0	t1	t2	t3	t4	t5	t6	t7	…	
(I_1) **X1<-(X2)+10**	IF_1	ID_1	EX_1	MA_1	WB_1					
(I_2) **X4<-(X1)+17**		IF_2	ID_2	ID_2	ID_2	EX_2	MA_2	WB_2		
(I_3)			IF_3	IF_3	IF_3	ID_3	EX_3	MA_3	WB_3	
(I_4)				*Stall stages*						

Each stall or kill introduces a bubble => CPI > 1

When is data actually available? At Execute Stage

Bypassing

	t0	t1	t2	t3	t4	t5	t6	t7	…	
(I_1) **X1<-(X2)+10**	IF_1	ID_1	EX_1	MA_1	WB_1					
(I_2) **X4<-(X1)+17**		IF_2	ID_2	ID_2	ID_2	EX_2	MA_2	WB_2		
(I_3)			IF_3	IF_3	IF_3	ID_3	EX_3	MA_3	WB_3	
(I_4)				*Stall stages*						

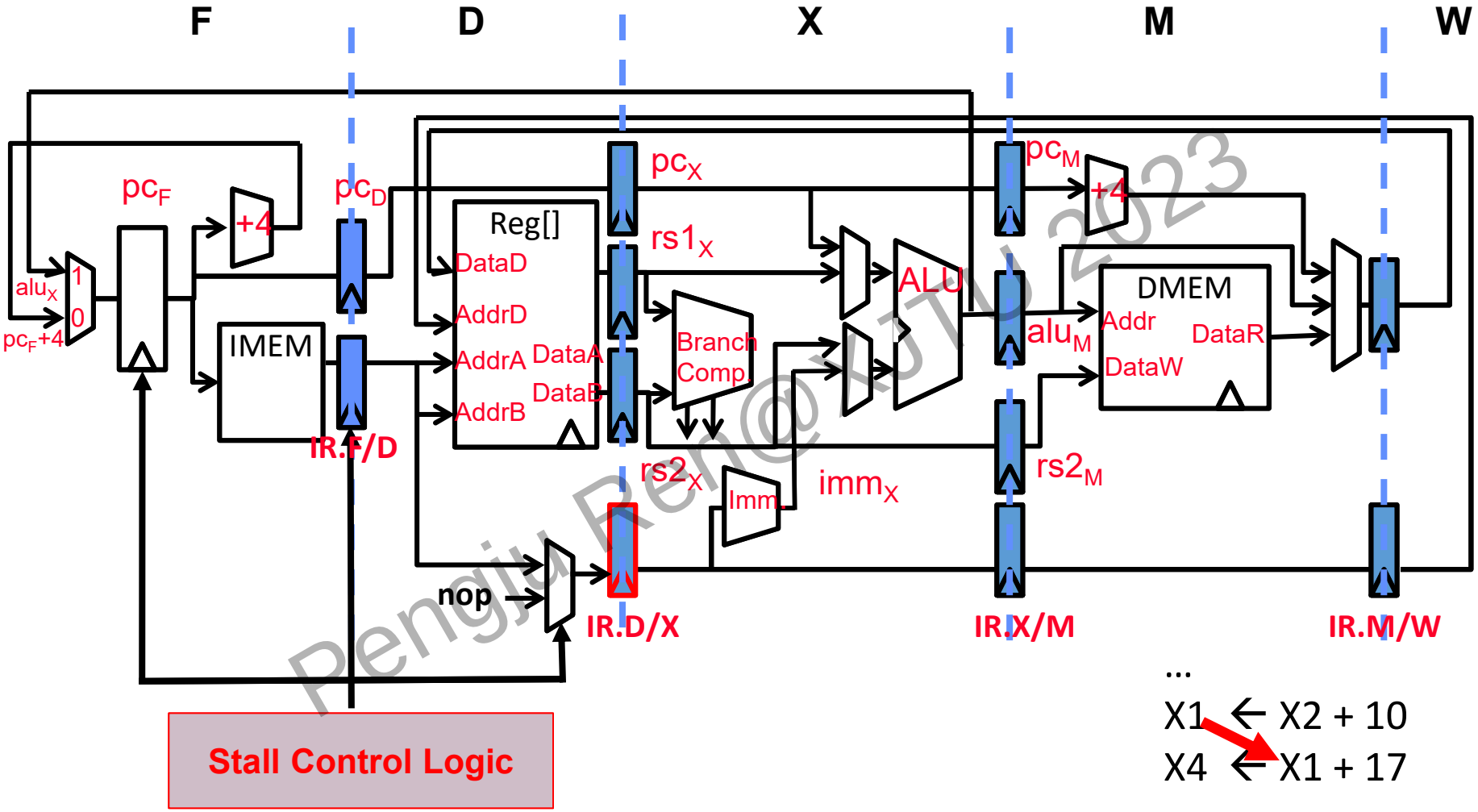
Each stall or kill introduces a bubble => CPI > 1

When is data actually available? At Execute Stage

	t0	t1	t2	t3	t4	t5	t6	t7	…	
(I_1) **X1<-(X2)+10**	IF_1	ID_1	EX_1	MA_1	WB_1					
(I_2) **X4<-(X1)+17**		IF_2	ID_2	EX_2	MA_2	WB_2				
(I_3)			IF_3	ID_3	EX_3	MA_3	WB_3			
(I_4)				IF_4	ID_4	EX_4	MA_4	WB_4		

A new datapath, i.e., a bypass (or feedback), can get the data from the output of the ALU to its input

Adding a Bypass



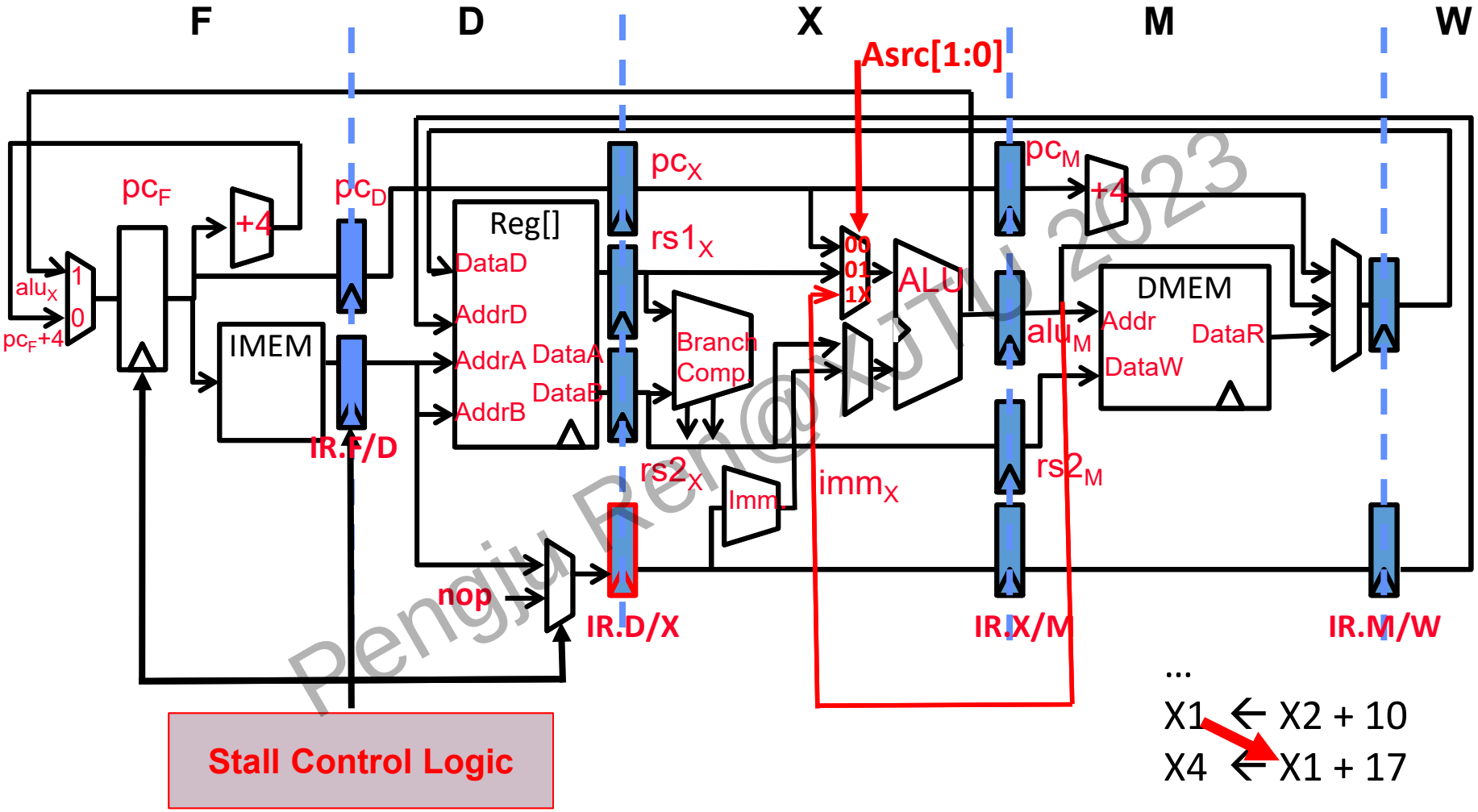
…

X1 ← X2 + 10

X4 ← X1 + 17

…

Adding a Bypass

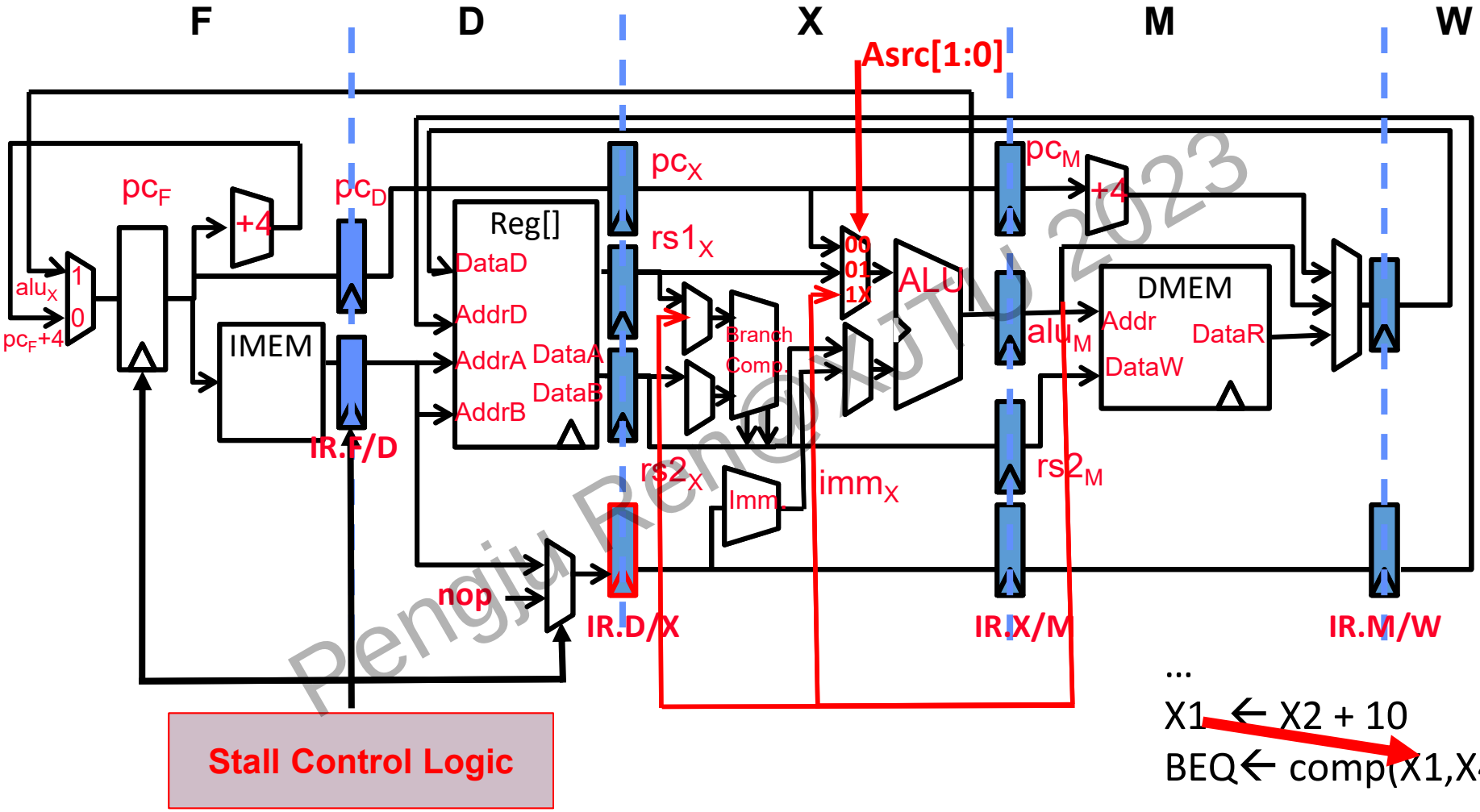


Stall Control Logic

...
X1 ← X2 + 10
X4 ← X1 + 17
...

Is this correct ?

Adding a Bypass



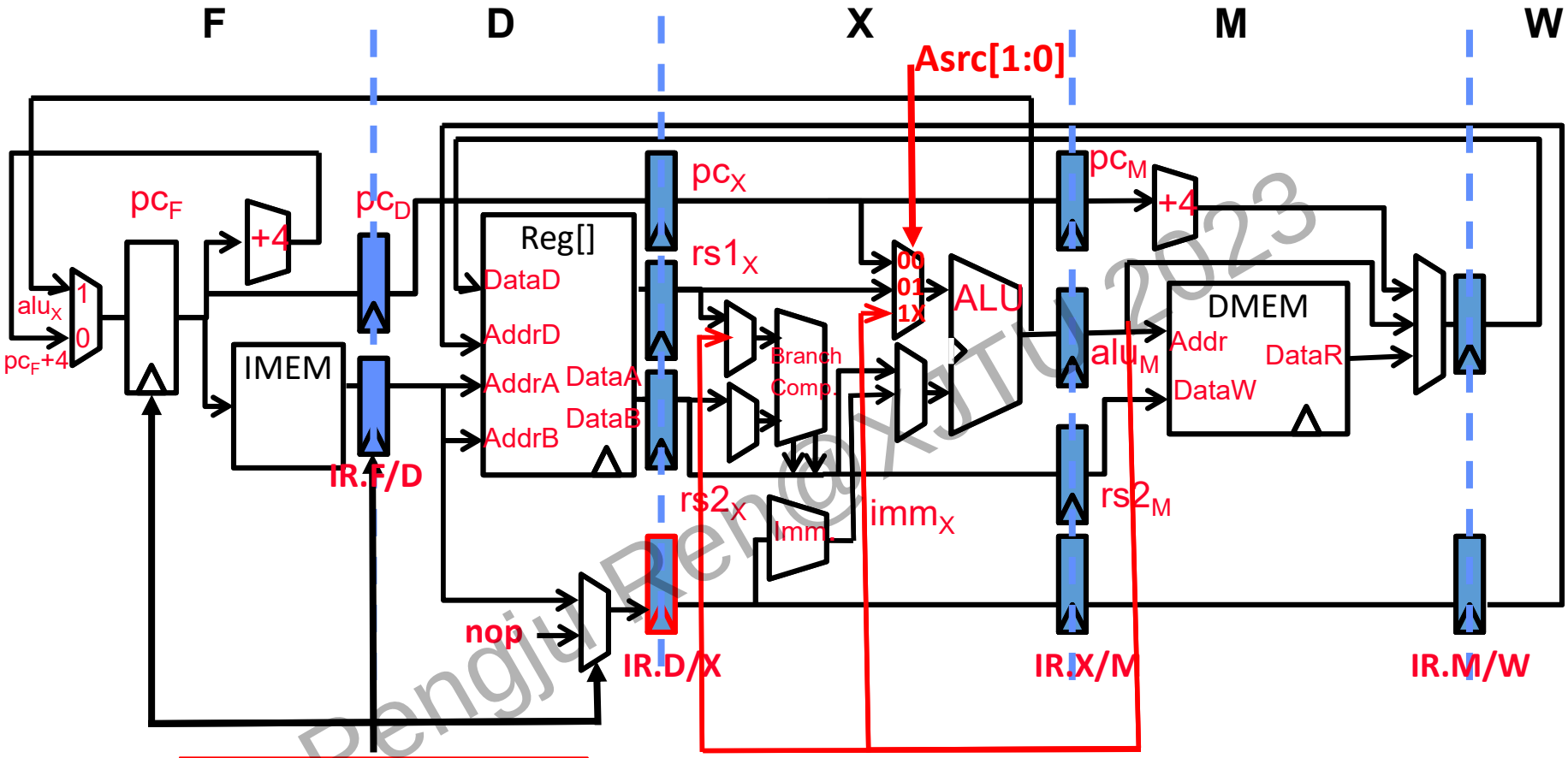
…

X1 ← X2 + 10

BEQ← comp(X1,X4)

…

Adding a Bypass



Stall Control Logic

When does this bypass help ?

yes

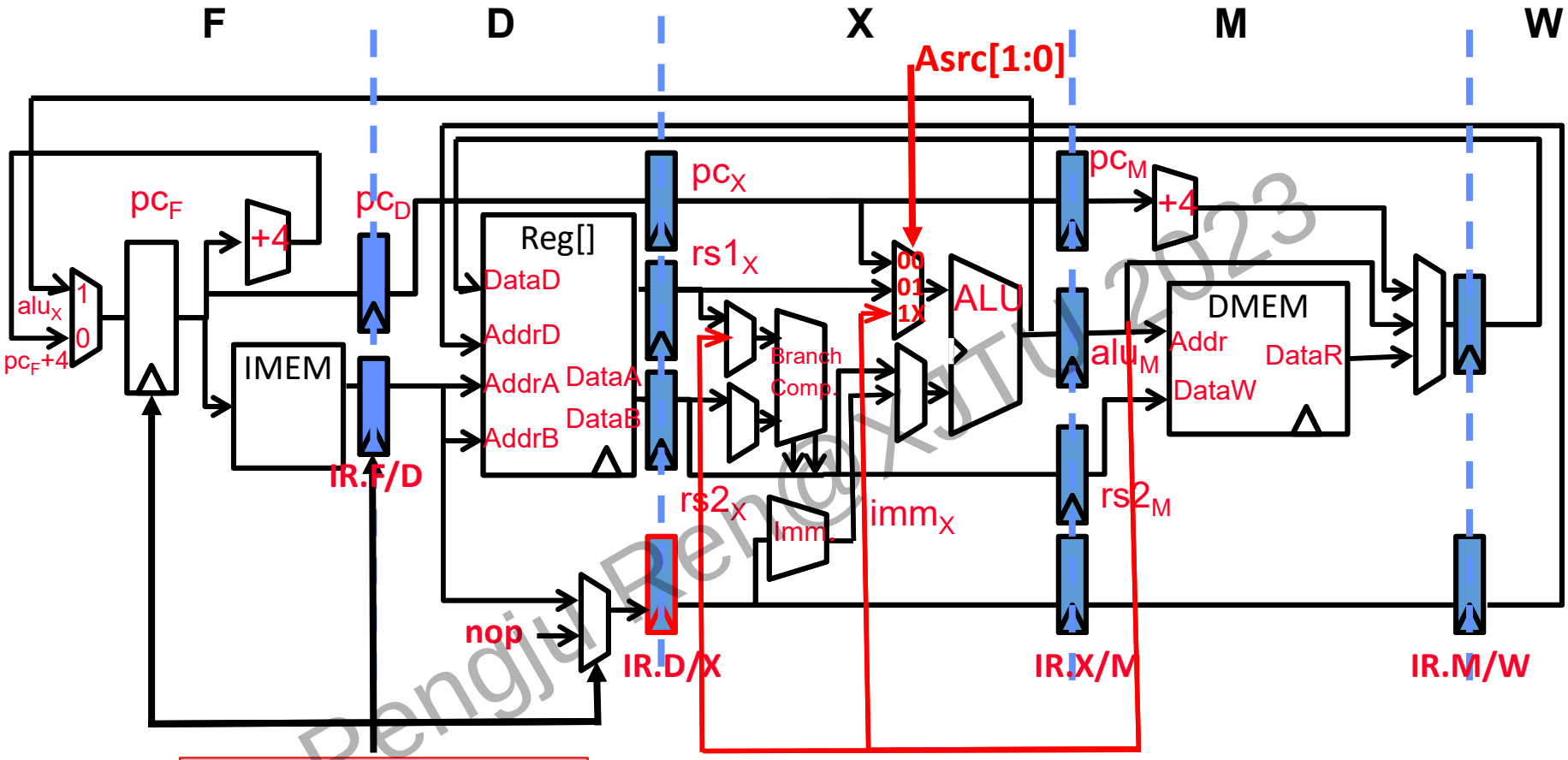
…

X1 ← X2 + 10

X4 ← X1 + 17

…

Adding a Bypass



Stall Control Logic

When does this bypass help ?

…

X1 ← X2 + 10

X4 ← X1 + 17

…

yes

…

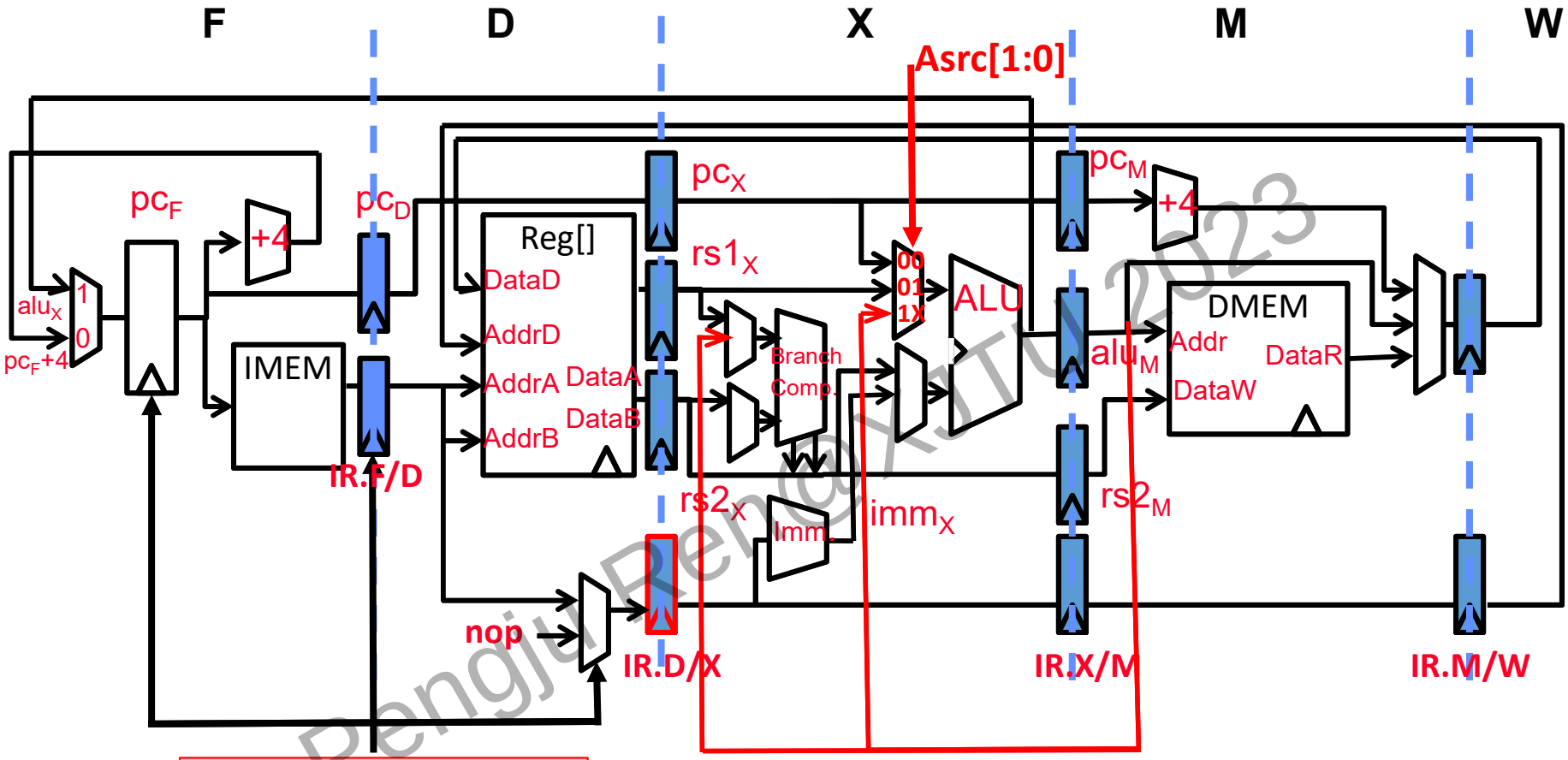
X1 ← M[X2 + 10]

X4 ← X1 + 17

…

No

Adding a Bypass



Stall Control Logic

When does this bypass help ?

yes	No	No
…	…	…
X1 ← X2 + 10	X1 ← M[X2 + 10]	JAL 500
X4 ← X1 + 17	X4 ← X1 + 17	X4 ← X1 + 17
…	…	…

Bypassing

	t0	t1	t2	t3	t4	t5	t6	t7	…	
(I_1) **X1<-(X2)+10**	IF_1	ID_1	EX_1	MA_1	WB_1					
(I_2) **X4<-(X1)+17**		IF_2	ID_2	ID_2	ID_2	EX_2	MA_2	WB_2		
(I_3)			IF_3	IF_3	IF_3	ID_3	EX_3	MA_3	WB_3	
(I_4)										

Stall stages

Each stall or kill introduces a bubble => CPI > 1

When is data actually available? At Execute Stage

The Bypass Signal

Asrc = (IR.D/X.rs1==IR.X/M.rd)IR.X/M.we

Stall=(~~(IR.D/X.rs1==IR.X/M.rd)IR.X/M.we~~

*+ (IR.D/X.rs1==IR.M/W.rd)**IR.M/W.we**) **IR.D/X.re1***

or

Bsrc = (IR.D/X.rs2==IR.X/M.rd)IR.X/M.we

~~((IR.D/X.rs2==IR.X/M.rd)IR.X/M.we~~

*+ (IR.D/X.rs2==IR.M/W.rd)**IR.M/W.we**) **IR.D/X.re2***

rd depends on opcode
R,I,U,UJ rd exist
Others rd not-exist
we depends on opcode
R,I,U,UJ we=on
Others we=off

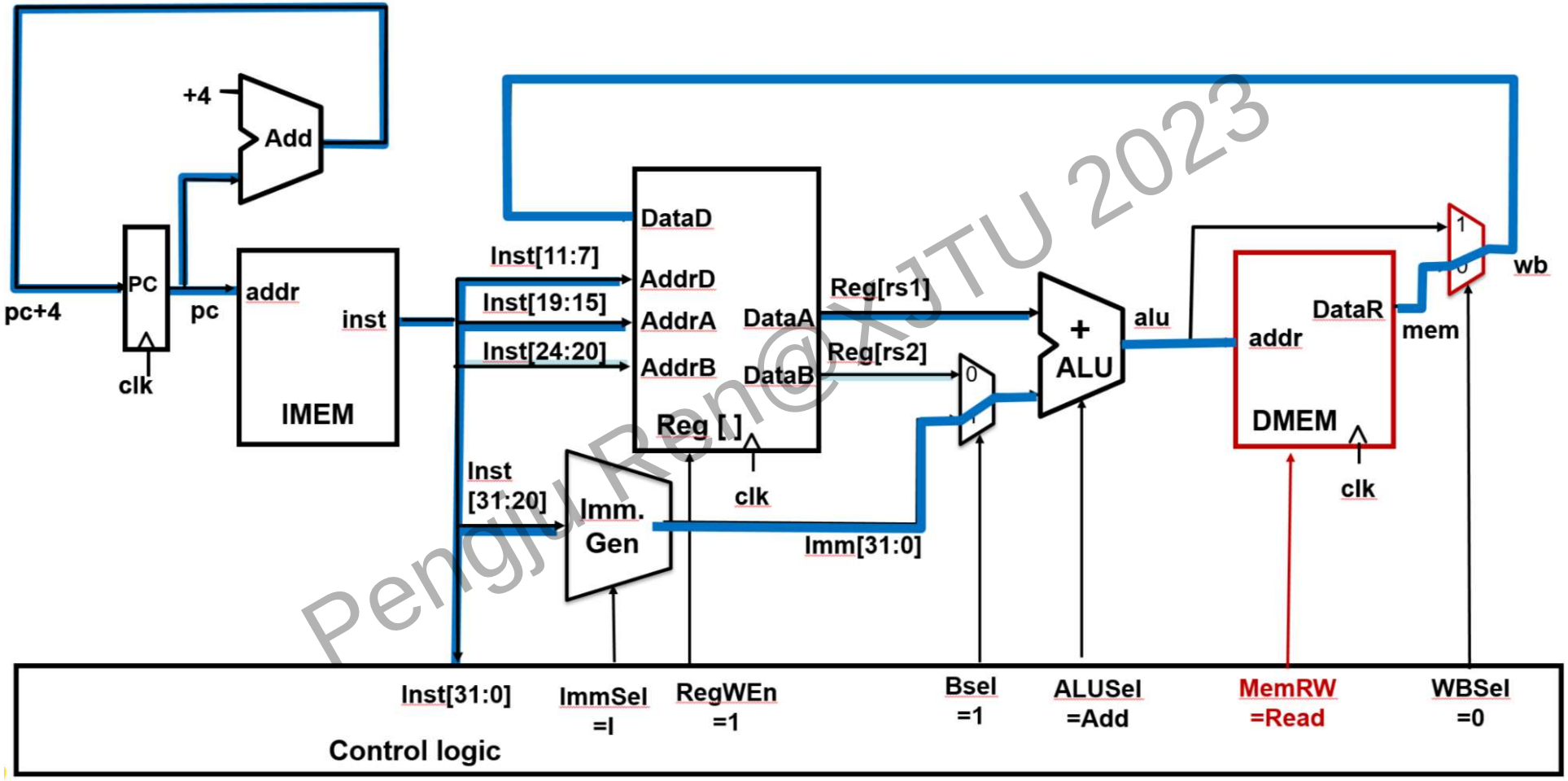
Is this correct ?

No, because only R, U and partial of I instructions can benefit from this bypass

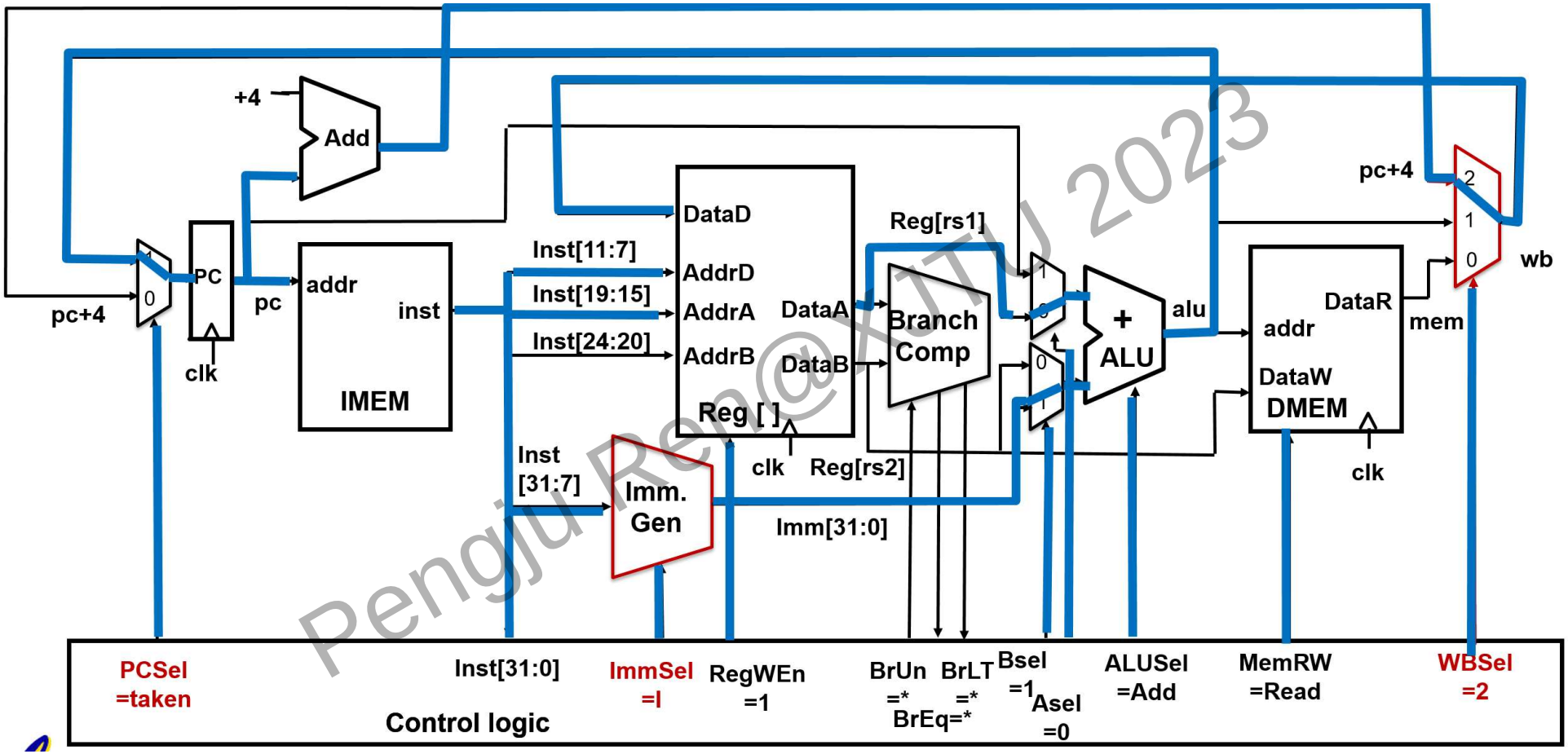
How might we address this?

Split we into two components: **we-bypass** and **we-stall**

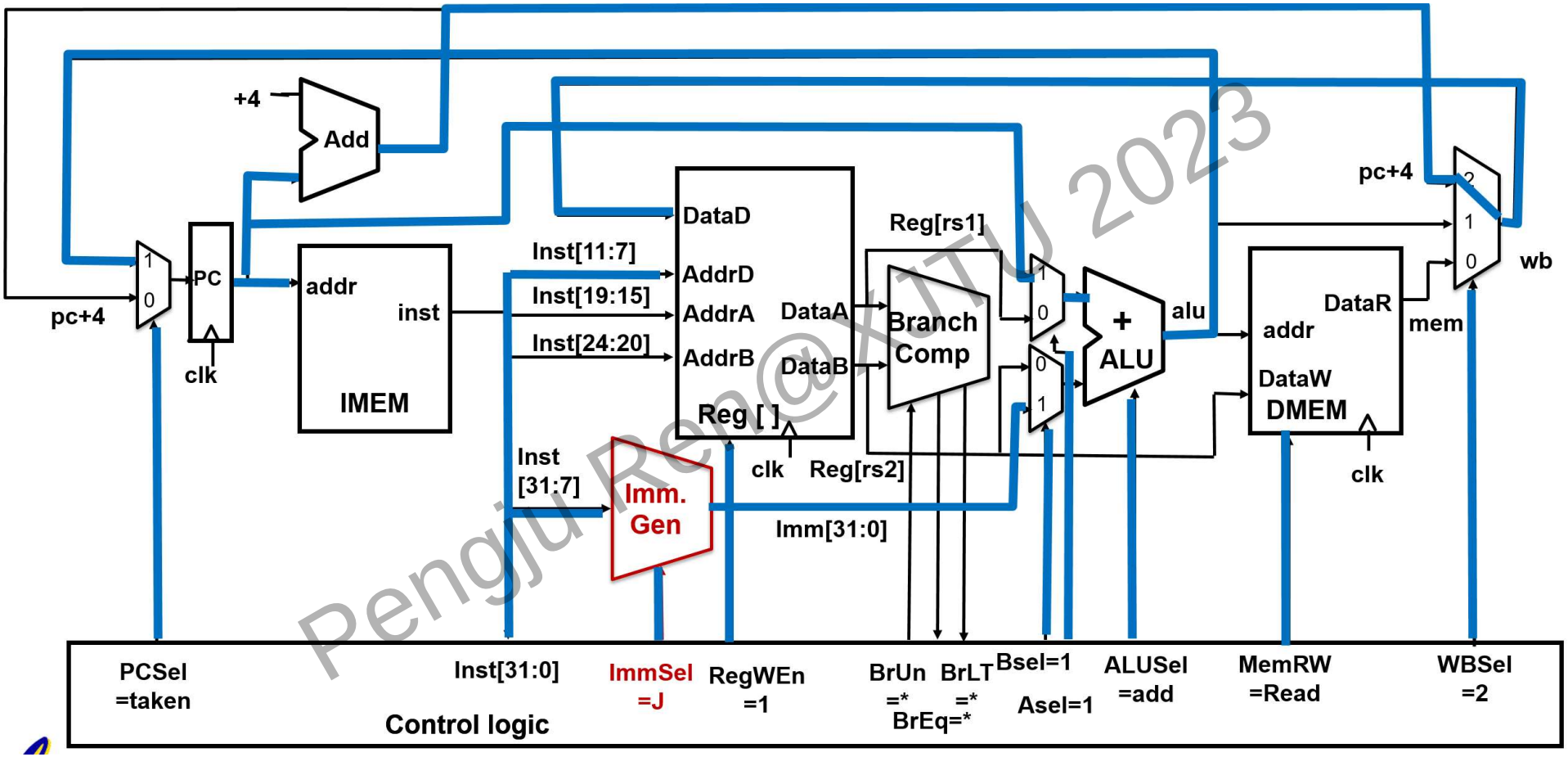
Recap: “load instructions” of I-type



Recap: JALR (R[rd] = PC+4; PC = R[rs1] + imm) of I-type



Adding JAL (R[rd] = PC+4; PC = PC + {imm,1b'0}) of UJ-type



Bypass and Stall Signals

Split X/M.we_E into two components: **X/M.$we - bypass$, X/M.$we - stall$**

we depends on opcode
R,I,U,UJ we=on
Others we=off

$we - bypass_E$ depends on opcode
R，U，I*（opcode=0000011）=> on
Others => off

$we - stall_E$ depends on opcode
I**，UJ => on
Others => off

Asrc = (IR.D/X.rs1==IR.X/M.rd)IR.X/M.we

Stall =((~~IR.D/X.rs1==IR.X/M.rd)IR.X/M.we~~

*+ (IR.D/X.rs1==IR.M/W.rd)**IR.M/W.we**) **IR.D/X.re1***

or

Bsrc = (IR.D/X.rs2==IR.X/M.rd)IR.X/M.we

((~~IR.D/X.rs2==IR.X/M.rd)IR.D/X.we~~

*+ (IR.D/X.rs2==IR.M/W.rd)**IR.M/W.we**) **IR.D/X.re2***

I*：I指令中的立即数操作；I**：I指令中的其它指令（如：Load和JALR）

Bypass and Stall Signals

Split X/M.we_E into two components: **X/M.$we - bypass$, X/M.$we - stall$**

we depends on opcode
R,I,U,UJ we=on
Others we=off

$we - bypass_E$ depends on opcode
R，U，I*（opcode=0000011）=> on
Others => off

$we - stall_E$ depends on opcode
I**，UJ => on
Others => off

Asrc = (IR.D/X.rs1==IR.X/M.rd)IR.X/M.we-bypass

*Stall =((IR.D/X.rs1==IR.X/M.rd)**IR.X/M.we-stall***

*+ (IR.D/X.rs1==IR.M/W.rd)**IR.M/W.we**) **IR.D/X.re1***

or *Bsrc = (IR.D/X.rs2==IR.X/M.rd)IR.X/M.we-bypass*

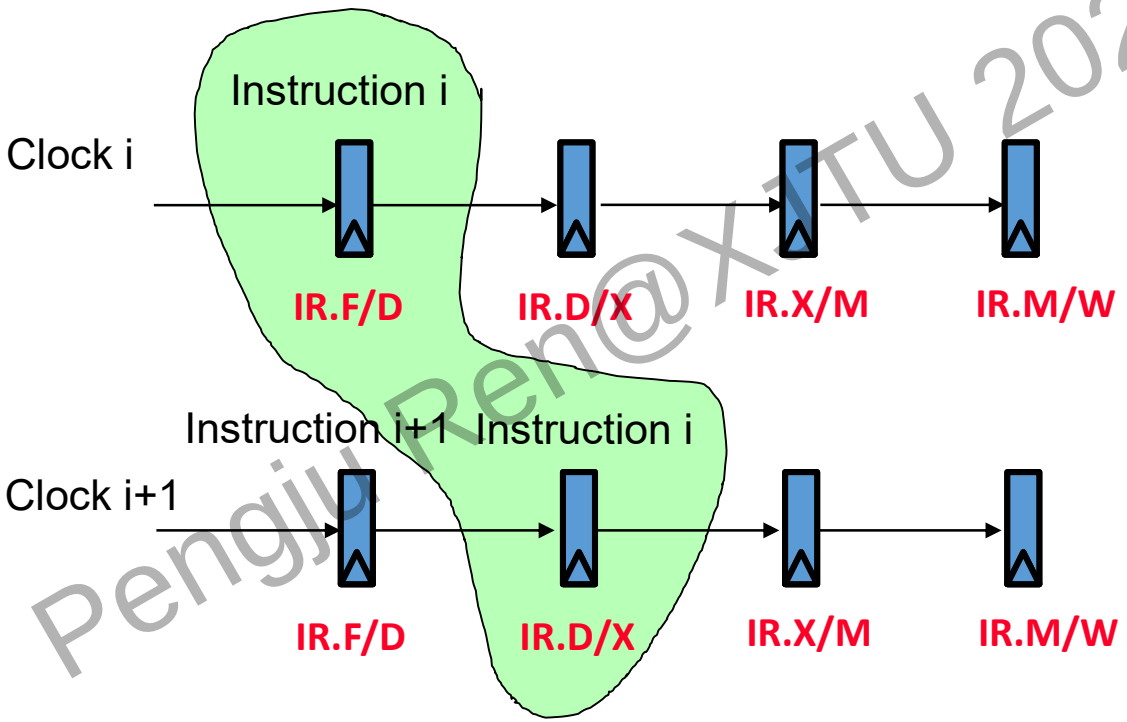
*((IR.D/X.rs2==IR.X/M.rd)**IR.X/M.we-stall***

*+ (IR.D/X.rs2==IR.M/W.rd)**IR.M/W.we**) **IR.D/X.re2***

I*：I指令中的立即数操作；I**：I指令中的其它指令（如：Load和JALR）

Bypass and Stall Signal

Deriving Bypass from the Stall Signal



Bypass and Stall Signal

Deriving Bypass from the Stall Signal

Asrc = (IR.D/X.rs1==IR.X/M.rd)IR.X/M.we-bypass

*Stall =((IR.D/X.rs1==IR.X/M.rd)**IR.X/M.we-stall***

*+ (IR.D/X.rs1==IR.M/W.rd)**IR.M/W.we**) **IR.D/X.re1***

or *Bsrc = (IR.D/X.rs2==IR.X/M.rd)IR.X/M.we-bypass*

*((IR.D/X.rs2==IR.X/M.rd)**IR.X/M.we-stall***

*+ (IR.D/X.rs2==IR.M/W.rd)**IR.M/W.we**) **IR.D/X.re2***

Asrc' = (IR.F/D.rs1==IR.D/X.rd)IR.D/X.we-bypass

*Stall =((IR.F/D.rs1==IR.D/X.rd)**IR.D/X.we-stall***

*+ (IR.F/D.rs1==IR.X/M.rd)**IR.X/M.we**) **IR.F/D.re1***

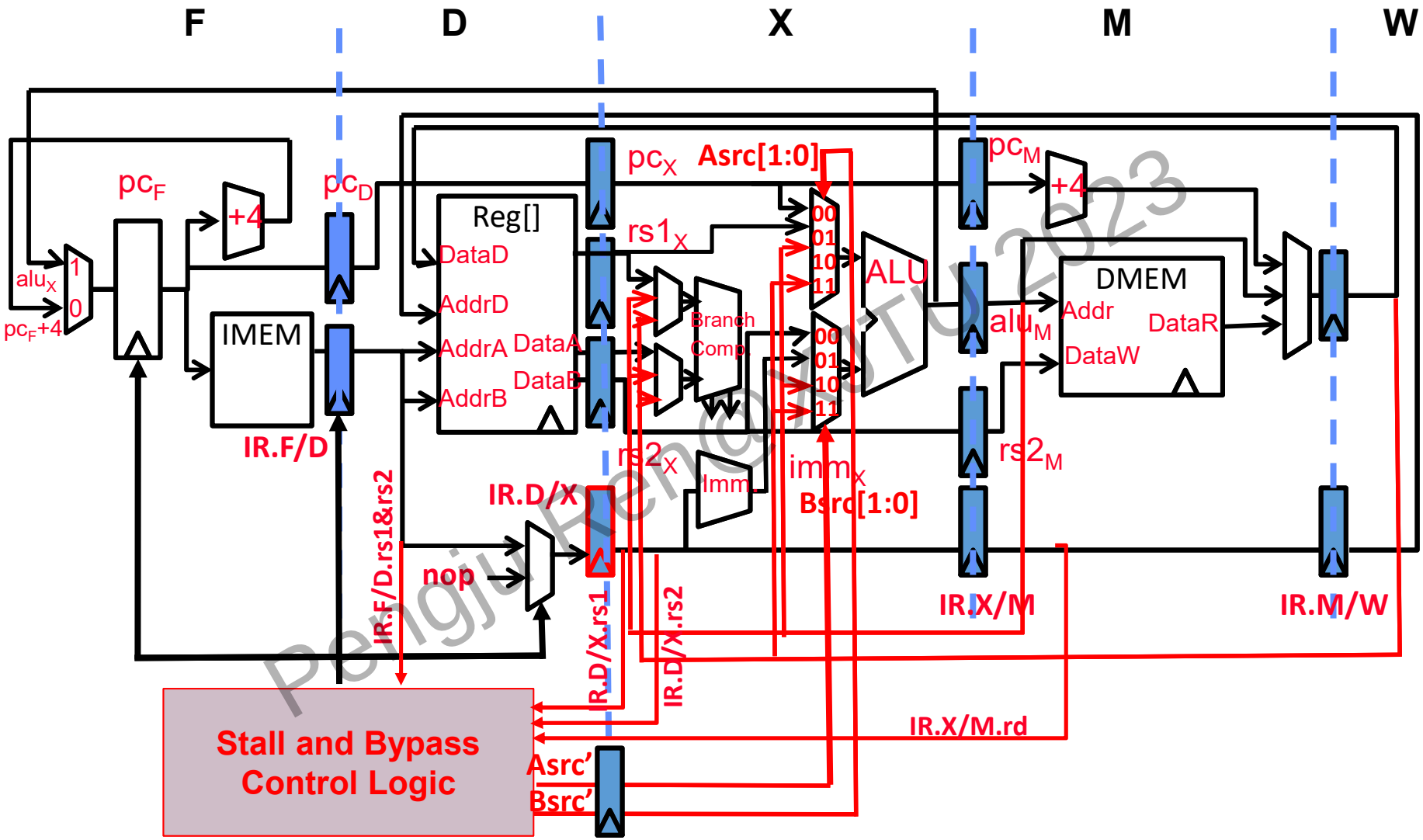
or *Bsrc' = (IR.F/D.rs2==IR.D/X.rd)IR.D/X.we-bypass*

*((IR.F/D.rs2==IR.D/X.rd)**IR.D/X.we-stall***

*+ (IR.F/D.rs2==IR.X/M.rd)**IR.X/M.we**) **IR.F/D.re2***

Asrc' and Bsrc' are generated one clock earlier than Asrc and Bsrc

Fully Bypassed Datapath



Note: Assumes data written to registers in a W-stage is readable in parallel D-stage. Extra write data register and bypass paths required if this is not possible.

Overview of Data Hazards

- Data hazards occur when one instruction depends on a data value produced by a preceding instruction still in the pipeline
- Approaches to resolving data hazards
  - **Stall:** Wait for the result to be available by freezing earlier pipeline stages
  - **Bypass:** Route data as soon as possible after it is calculated to the earlier pipeline stage
  - **Speculate:** (later in course)

    Two cases:

    Guessed correctly -> do nothing

    Guessed incorrectly -> kill and restart

Agenda

Pipeline and hazards:

— Pipeline Basics

— Structural Hazards

— Data Hazards

— Control Hazards

Instruction to Instruction Dependence

- **What do we need to calculate next PC?**
  - For Jumps
    - Opcode, offset, and PC
  - For Jump Register
    - Opcode and register value
  - For Conditional Branches
    - Opcode, offset, PC, and register (for condition)
  - For all others
    - Opcode and PC
- **In what stage do we know these?**
  - PC → Fetch
  - Opcode, offset → Decode (or Fetch?)
  - Register value → Decode
  - Branch condition (rs1==rs2) → Execute (or Decode?)

NextPC Calculation Bubbles

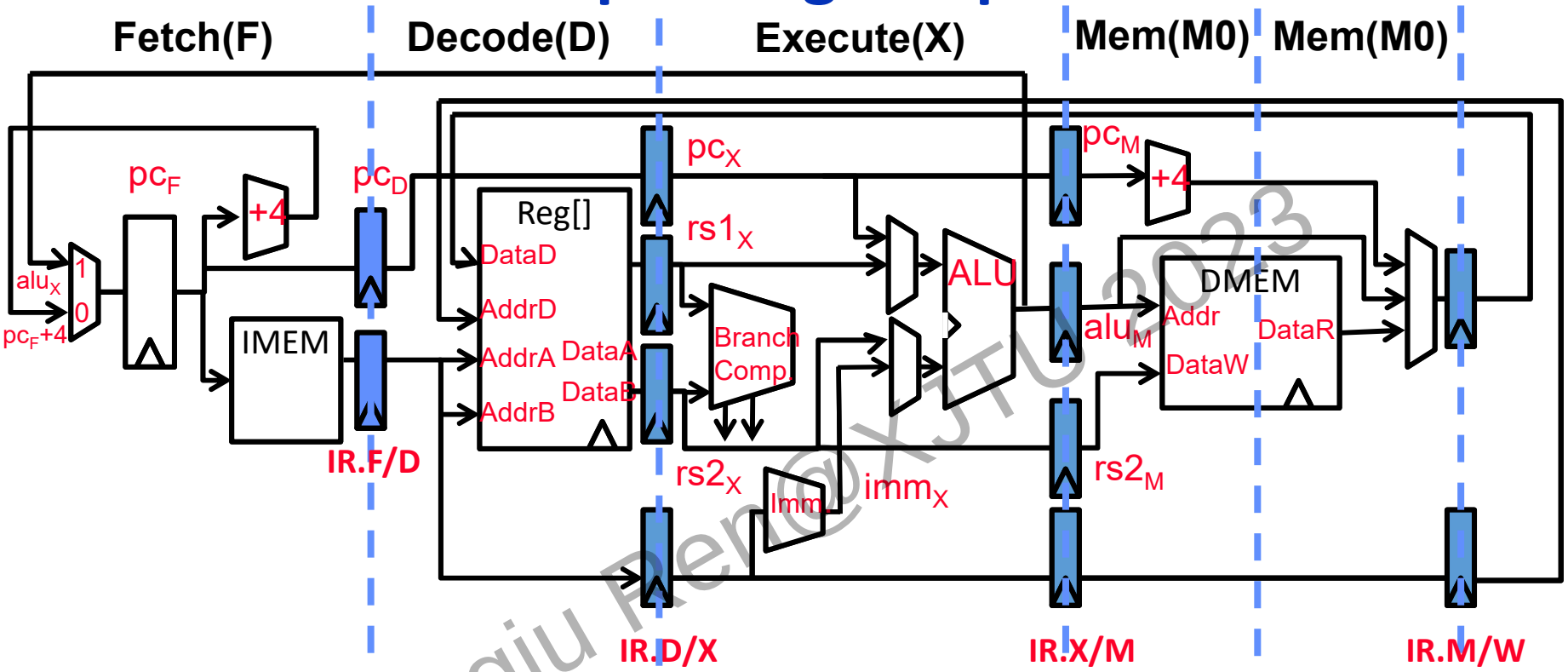
	t0	t1	t2	t3	t4	t5	t6	t7	…	
(I_1) **X1<-(X2)+10**	IF_1	ID_1	EX_1	MA_1	WB_1					
(I_2) **X4<-(X3)+17**		IF_2	$\boxed{IF_2}$	ID_2	EX_2	MA_2	WB_2			
(I_3)				IF_3	$\boxed{IF_3}$	ID_3	EX_3	MA_3	WB_3	
(I_4)						IF_4	$\boxed{IF_4}$	ID_4	EX_4	MA_4

	t0	t1	t2	t3	t4	t5	t6	t7	…	
IF	I_1	nop	I_2	nop	I_3	nop	I_4			
ID		I_1	nop	I_2	nop	I_3	nop	I_4		
EX			I_1	nop	I_2	nop	I_3	nop	I_4	
MA				I_1	nop	I_2	nop	I_3	nop	I_4
WB					I_1	nop	I_2	nop	I_3	nop

nop => pipeline bubble

What's a good guess for next PC ? PC + 4

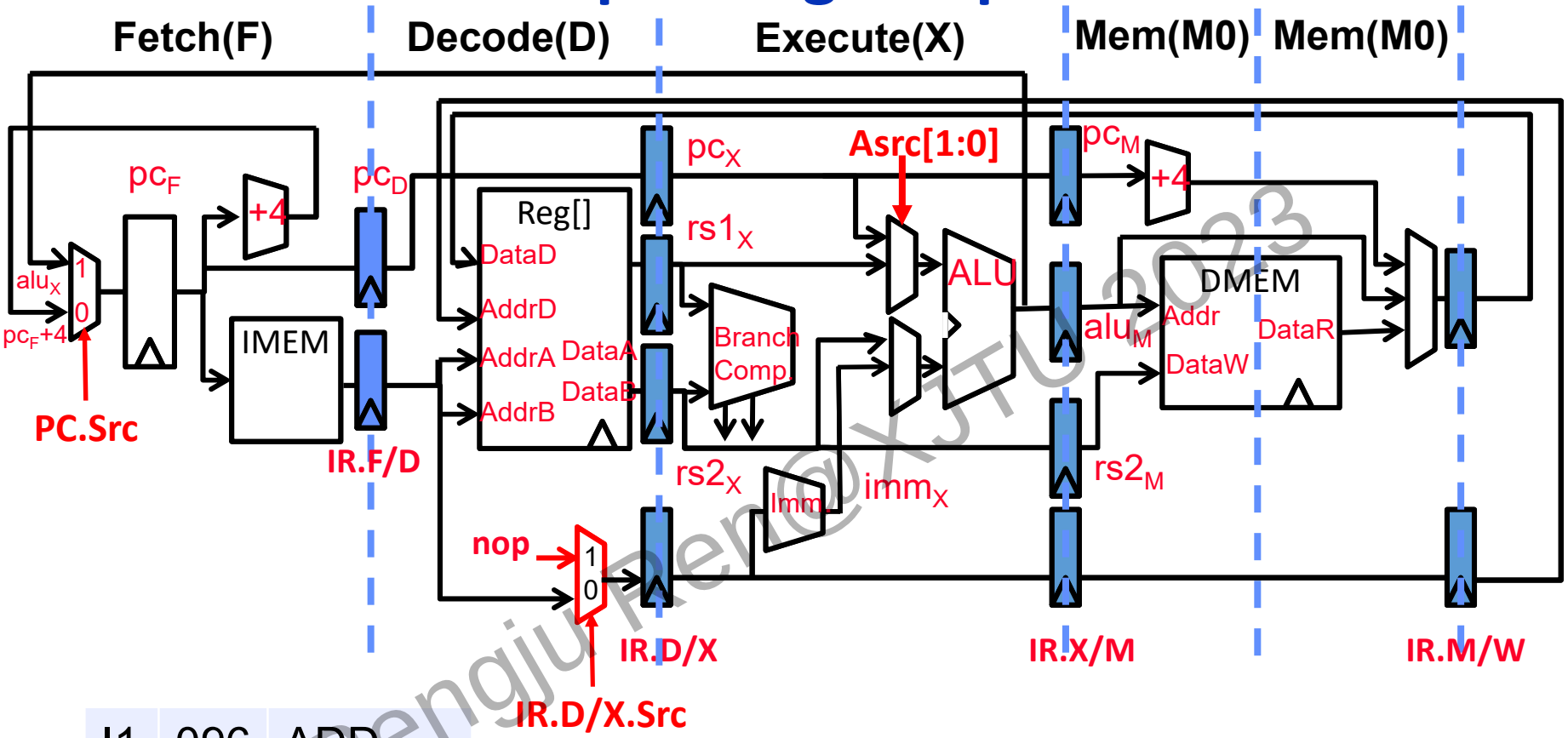
Pipelining Jumps



I1	096	ADD
I2	100	JAL 200
I3	104	ADD
I4	--	--
I5	300	ADD

What happens on mis-speculation,
i.e., when next instruction is not PC+4 ?

Pipelining Jumps

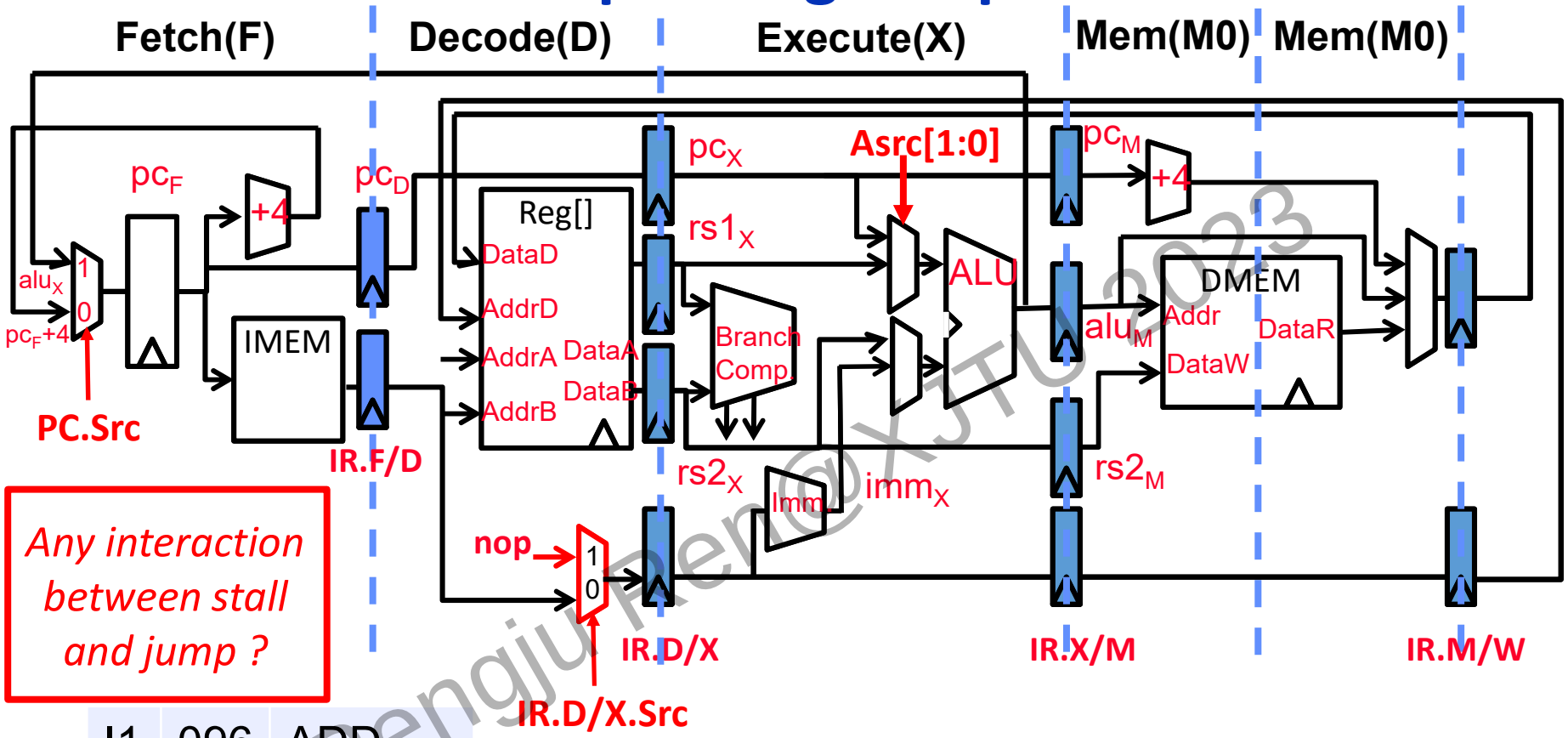


I1	096	ADD
I2	100	JAL 200
~~I3~~	~~104~~	~~ADD~~
I4	--	--
I5	300	ADD

Kill

1. To kill a fetched instruction – Insert a nop
2. Selecting the right PC for next instruction

Pipelining Jumps



Any interaction between stall and jump ?

I1	096	ADD
I2	100	JAL 200
~~I3~~	~~104~~	~~ADD~~
I4	--	--
I5	300	ADD

Kill

PC_{Src}, A_{src} and $IR.D/X.Src$ depends on $opcode_D$
if UJ: Asrc = 00; else Asrc=others
If UJ: PC.src=1; else PC.src=0
If UJ: IR.D/X.src=1; else IR.D/X.src=0

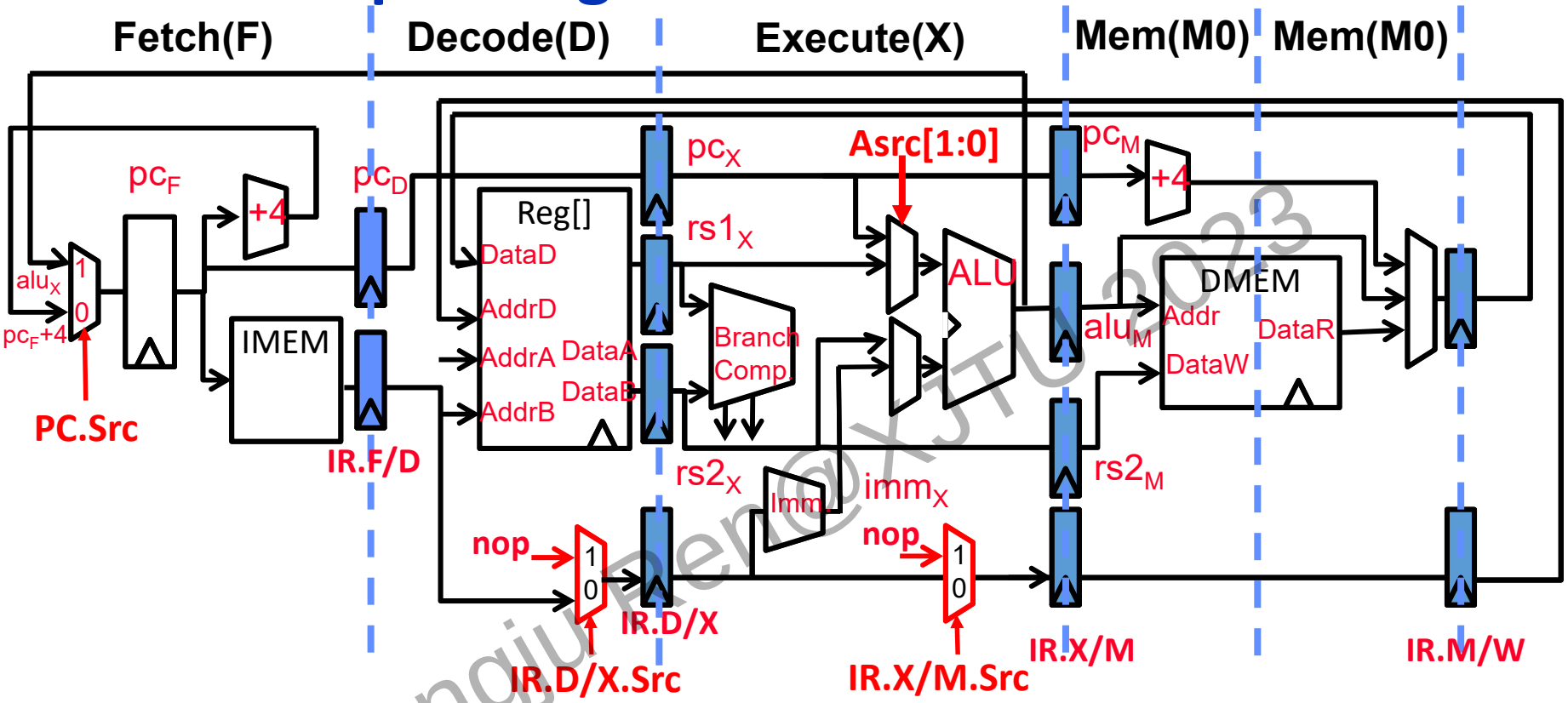
NextPC Calculation Bubbles

	t0	t1	t2	t3	t4	t5	t6	t7	…	
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	WB_1					
(I_2) 100: JAL 200		IF_2	ID_2	EX_2	MA_2	WB_2				
(I_3) 104: ADD			IF_3	nop	nop	nop	nop			
(I_4) –-				--	--	--	--	--		
(I_5) 300: ADD					IF_5	ID_5	EX_5	MA_5	WB_5	

	t0	t1	t2	t3	t4	t5	t6	t7	…	
IF	I_1	I_2	I_3	--	I_5					
ID		I_1	I_2	nop	--	I_5				
EX			I_1	I_2	nop	--	I_5			
MA				I_1	I_2	nop	--	I_5		
WB					I_1	I_2	nop	–-	I_5	

nop => pipeline bubble

Pipelining Conditional Branches



I1	096	ADD
I2	100	BEQ X1,X2, 200
~~I3~~	~~104~~	~~ADD~~
~~I4~~	~~108~~	~~ADD~~
I5	300	ADD

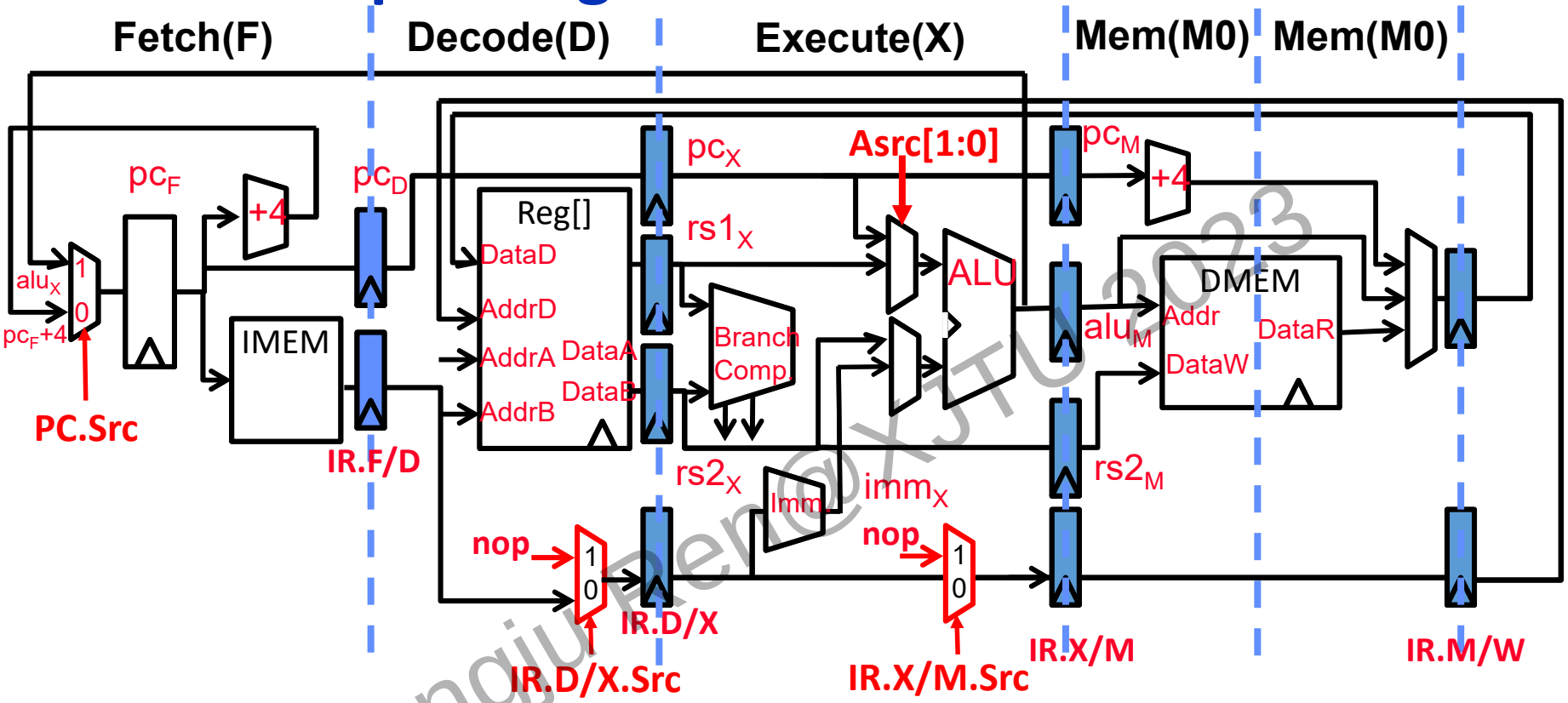
Kill

Branch condition is not known until the execute stage
what action should be taken in the decode stage?

If the branch is taken:

1. Kill the two following instructions
2. The instruction at the decode stage is not valid (Stall signal is not valid)

Pipelining Conditional Branches



I1	096	ADD
I2	100	BEQ X1,X2, 200
~~I3~~	~~104~~	~~ADD~~
~~I4~~	~~108~~	~~ADD~~
I5	300	ADD

Kill

PC.src, Asrc, IR.D/X.Src and IR.X/M.Src are depends on opcodeD and when Result of Branch Comp is True(BC.r).
if SB&BranchComp.r==T: Asrc = 00; else Asrc=others
If SB&BranchComp.r==T: PC.src=1; else PC.src=0
If SB&BranchComp.r==T: IR.D/X.src=1 and IR.D/X.src=1; else IR.D/X.src=0 and IR.D/X.src=0

Branch Pipeline Diagrams

	t0	t1	t2	t3	t4	t5	t6	t7	…
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) 100: BEQ X1,X2,200		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) 104: ADD			IF_3	ID_3	nop	nop	nop		
(I_4) 108:				IF_4	nop	nop	nop		
(I_5) 300: ADD					IF_5	ID_5	EX_5	MA_5	WB_5

	t0	t1	t2	t3	t4	t5	t6	t7	…	
IF	I_1	I_2	I_3	I_4	I_5					
ID		I_1	I_2	I_3	Nop	I_5				
EX			I_1	I_2	nop	nop	I_5			
MA				I_1	I_2	nop	nop	I_5		
WB					I_1	I_2	nop	nop	I_5	

nop => pipeline bubble

New Stall Signal

Stall =((IR.F/D.rs1==IR.D/X.rd)IR.D/X.we
+ (IR.F/D.rs1==IR.X/M.rd)IR.X/M.we) IR.F/D.re1
or
((IR.F/D.rs2==IR.D/X.rd)IR.D/X.we
+ (IR.F/D.rs2==IR.X/M.rd)IR.X/M.we) IR.F/D.re2
and
$!((opcode_E == BX)BX_{true}?)$

Don't stall if the branch is taken. Why?
Instruction at the decode stage is invalid

Control Equations for PC and IR Muxes

IR.D/X depends on $opcode_E$
$SB\ \&BranchComp == True \Rightarrow nop$
IR.D/X depends on $opcode_D$
$JAL, JALR \Rightarrow nop$
Others ⇒IR.F/D

IR.X/M depends on $opcode_E$
$SB\ \&BranchComp == True \Rightarrow nop$
Others ⇒ IR.D/X

$PC.Src$ denpends on $opcode_E$
$SB\ \&BranchComp == True \Rightarrow$ (PC=PC+{immediate,1b'0)
$PC.Src$ denpends on $opcode_D$
JAL ⇒ (PC=PC+{immediate,1'b0)
$JALR$ ⇒PC= (R[rs1]+immediate)
Ebreak or ecall ⇒ PC jump to **Debug** or **OS control**
Others ⇒ PC=PC+4

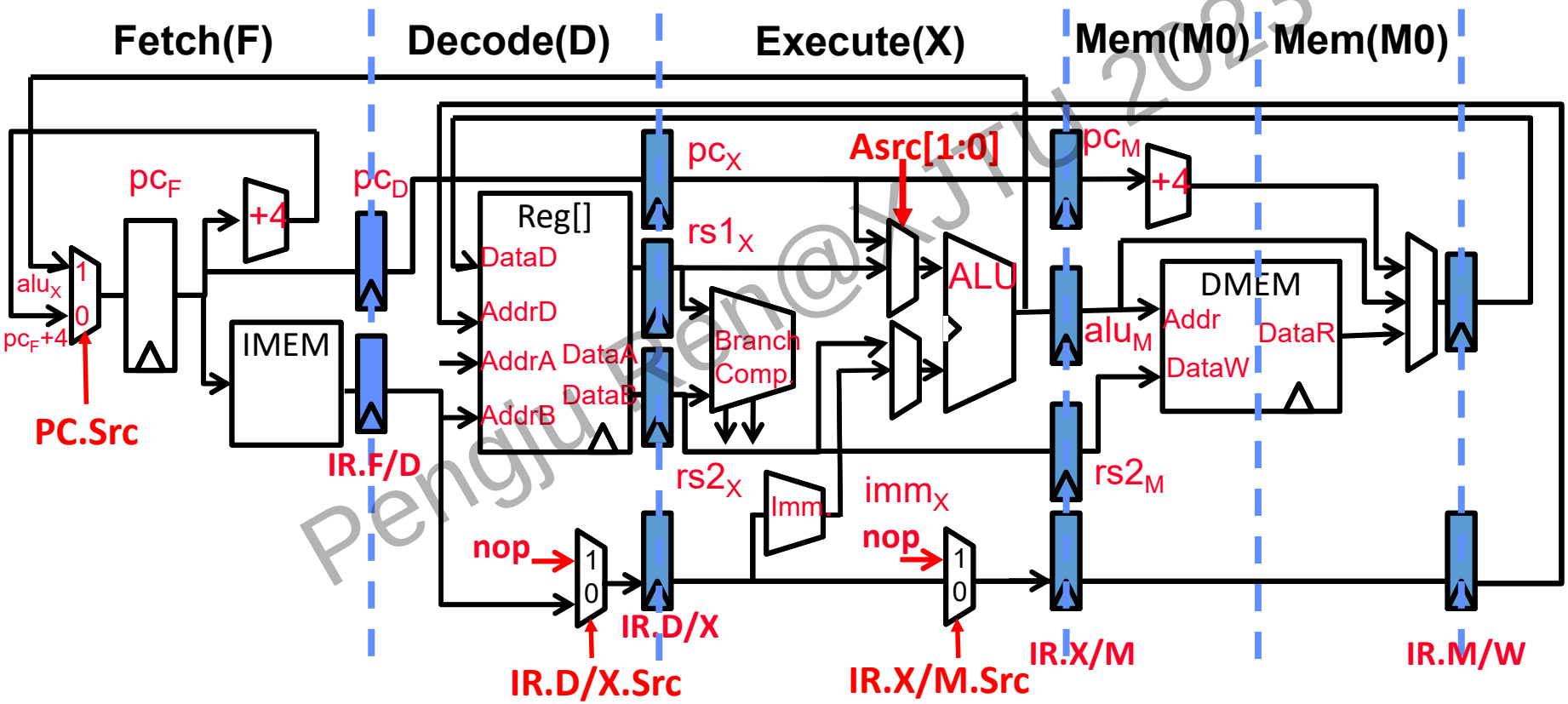
Give priority to the older instruction, i.e., execute stage instruction over decode stage instruction

Why?

pc+4 is a speculative guess

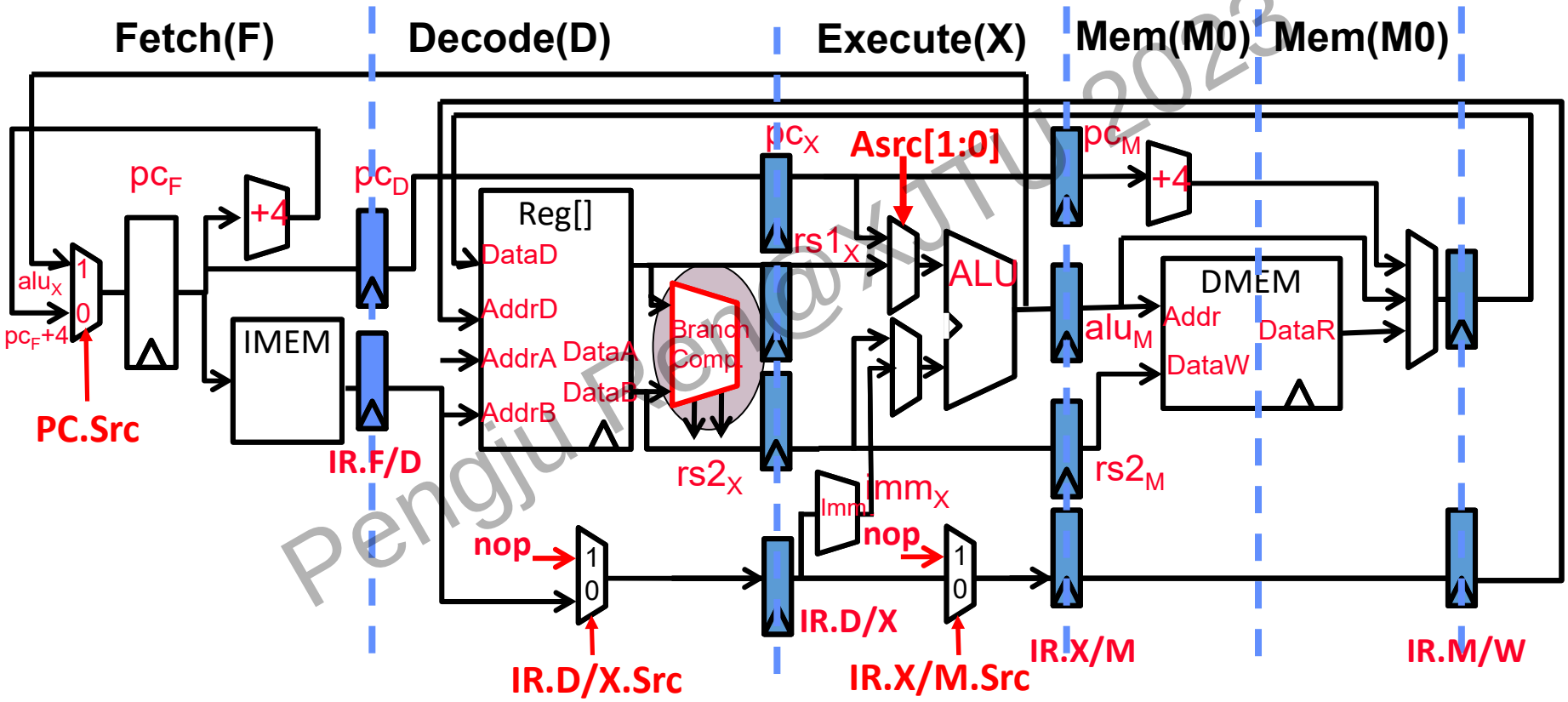
Reducing Branch Penalty

One pipeline bubble can be removed if an extra comparator(or adder) is used in the Decode stage



Reducing Branch Penalty

One pipeline bubble can be removed if an extra comparator is used in the Decode stage



Pipeline diagram now same as for jumps

Branch Delay Slots

Change the ISA semantics so that the instruction that follows a jump or branch is always executed

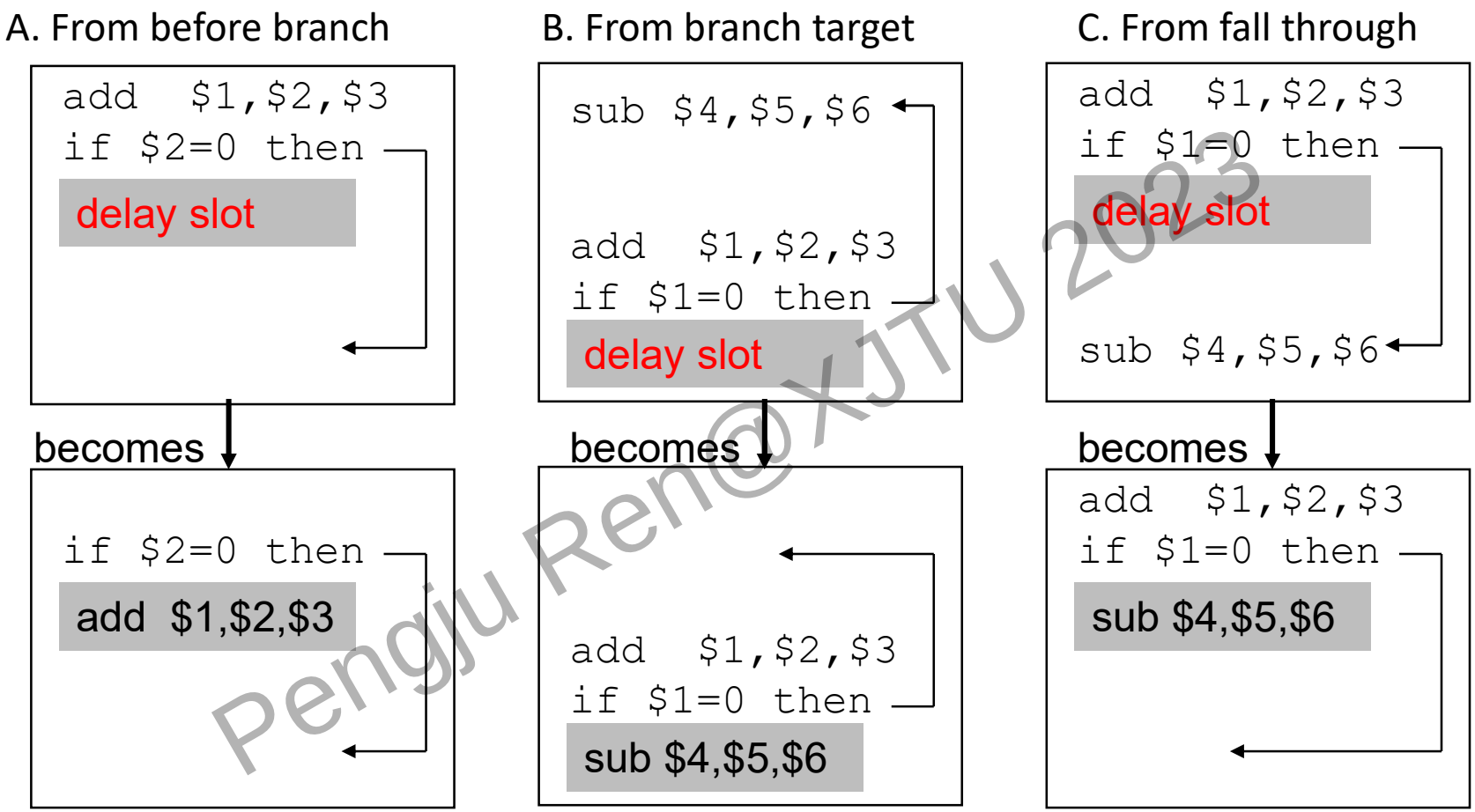
– gives compiler the flexibility to put in a useful instruction where normally a pipeline bubble would have resulted.

I1	096	ADD
I2	100	BEQ X1,X2, 200
I3	104	ADD
I4	108	ADD
I5	300	ADD

Delay slot instruction executed regardless of branch outcome

Other techniques include branch prediction, which can dramatically reduce the branch penalty... *to come later*

Scheduling Branch Delay Slots



- A is the best choice, fills delay slot & reduces instruction count (IC) **(#1)**
- In B, the sub instruction may need to be copied, increasing IC
- In B and C, must be okay to execute sub when branch fails

Why an instruction may not be dispatched every cycle (CPI > 1)

- **Full bypassing may be too expensive to implement**
  - Typically all frequently used paths are provided
  - Some infrequently used bypass paths may increase cycle time and counteract the benefit of reducing CPI
- **Loads have two cycle latency**
  - Instruction after load cannot use load result
  - MIPS-I ISA defined load delay slots, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard). Removed in MIPS-II.
- **Conditional branches may cause bubbles**
  - Kill following instruction(s) if no delay slots

Machines with software-visible delay slots may execute significant number of NOP instructions inserted by the compiler

Traps and Interrupts (other Control hazards)

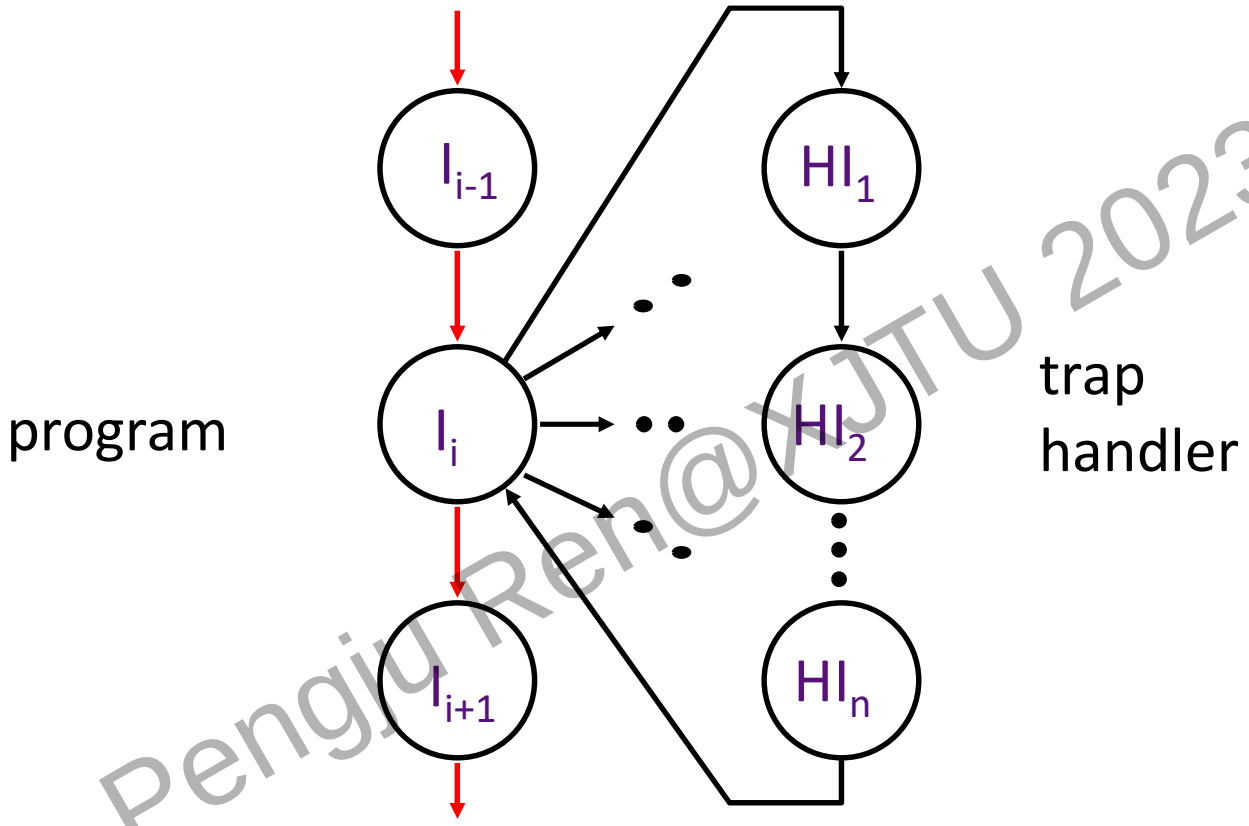
In class, we'll use following terminology

- ***Exception***: An unusual internal event caused by program during execution
  - E.g., page fault, arithmetic underflow
- ***Interrupt***: An external event outside of running program
- ***Trap***: Forced transfer of control to supervisor caused by exception or interrupt
  - Not all exceptions cause traps (c.f. IEEE 754 floating-point standard)

Asynchronous Interrupts

- An I/O device requests attention by asserting one of the *prioritized interrupt request lines*
- When the processor decides to process the interrupt
  - It stops the current program at instruction I_i , completing all the instructions up to I_{i-1} *(precise interrupt)*
  - It saves the PC of instruction I_i in a special register (EPC)
  - It disables interrupts and transfers control to a designated interrupt handler running in supervisor mode

Trap:
altering the normal flow of control



An *external or internal event* that needs to be processed by another (system) program. The event is usually unexpected or rare from program's point of view.

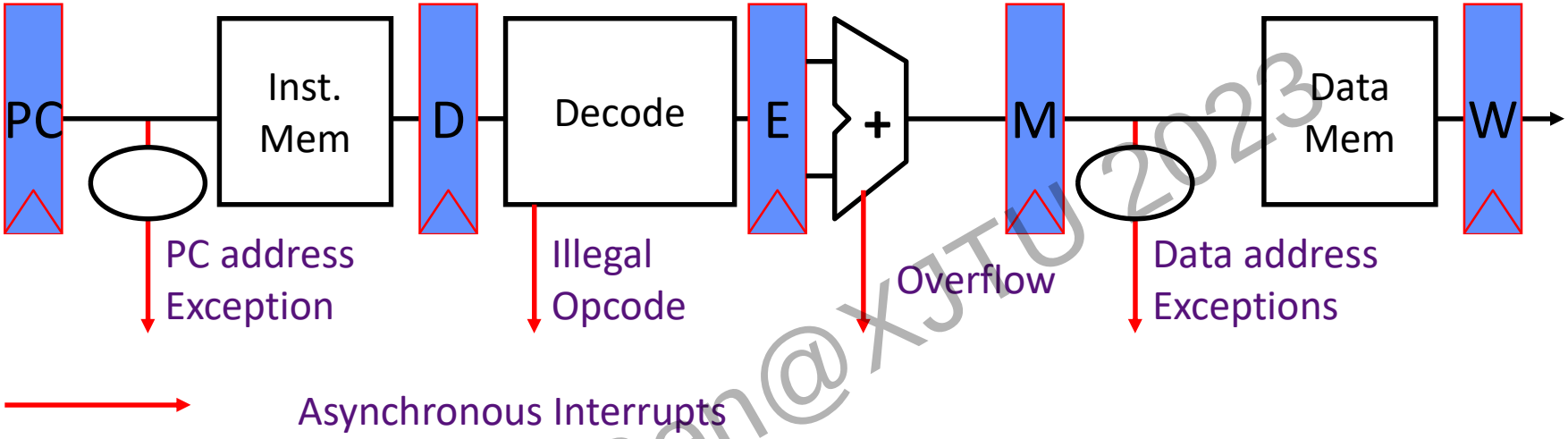
Trap Handler

- Saves ***EPC*** before enabling interrupts to allow nested interrupts ⇒
  - need an instruction to move EPC into GPRs
  - need a way to mask further interrupts at least until EPC can be saved
- Needs to read a *status register* that indicates the ***cause*** of the trap
- Uses a special indirect jump instruction ERET (*return-from-environment*) which
  - enables interrupts
  - restores the processor to the user mode
  - restores hardware status and control state

Synchronous Trap

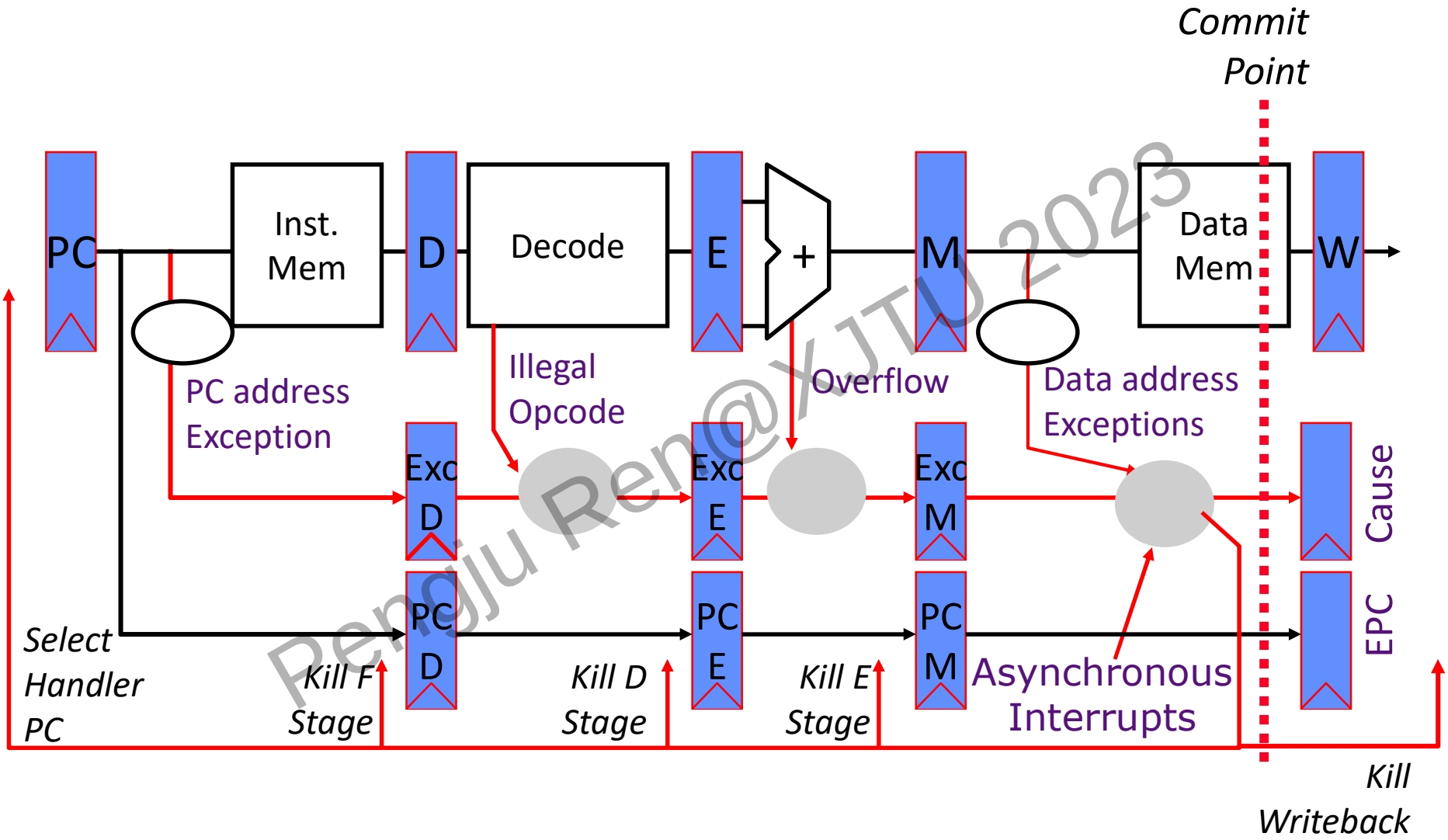
- A synchronous trap is caused by an exception on a *particular instruction*
- In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled
  - requires undoing the effect of one or more partially executed instructions
- In the case of a system call trap, the instruction is considered to have been completed
  - a special jump instruction involving a change to a privileged mode

Exception Handling 5-Stage Pipeline



- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?

Exception Handling 5-Stage Pipeline



Exception Handling 5-Stage Pipeline

- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier pipe stages override later exceptions *for a given instruction*
- Inject external interrupts at commit point (override others)
- If trap at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

Speculating on Exceptions

- Prediction mechanism
  - Exceptions are rare, so simply predicting no exceptions is very accurate!
- Check prediction mechanism
  - Exceptions detected at end of instruction execution pipeline, special hardware for various exception types
- Recovery mechanism
  - Only write architectural state at commit point, so can throw away partially executed instructions after exception
  - Launch exception handler after flushing pipeline
- Bypassing allows use of uncommitted instruction results by following instructions

Exception Pipeline Diagram

			t0	t1	t2	t3	t4	t5	t6	t7	t8
(I1)	096	ADD	F	D	E	M	Nop				
(I2)	100	XOR		F	D	E	Nop	Nop			
(I3)	104	SUB			F	D	Nop	Nop	Nop		
(I4)	108	ADD				F	Nop	Nop	Nop	Nop	
(I5)	Exc	Handler code					F	D	E	M	W

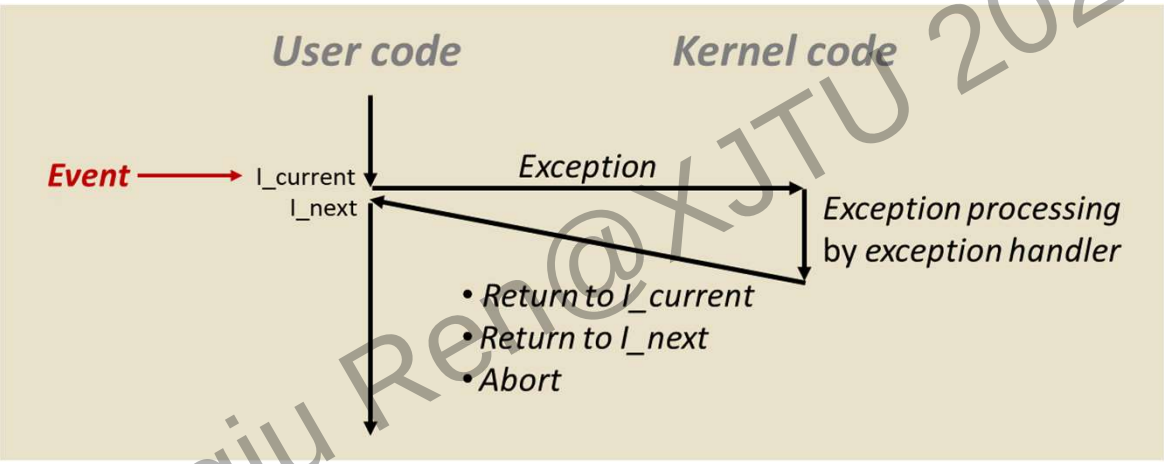
	t0	t1	t2	t3	t4	t5	t6	t7	t8
F	I1	I2	I3	I4	I5				
D		I1	I2	I3	Nop	I5			
E			I1	I2	Nop	Nop	I5		
M				I1	Nop	Nop	Nop	I5	
W					Nop	Nop	Nop	Nop	I5

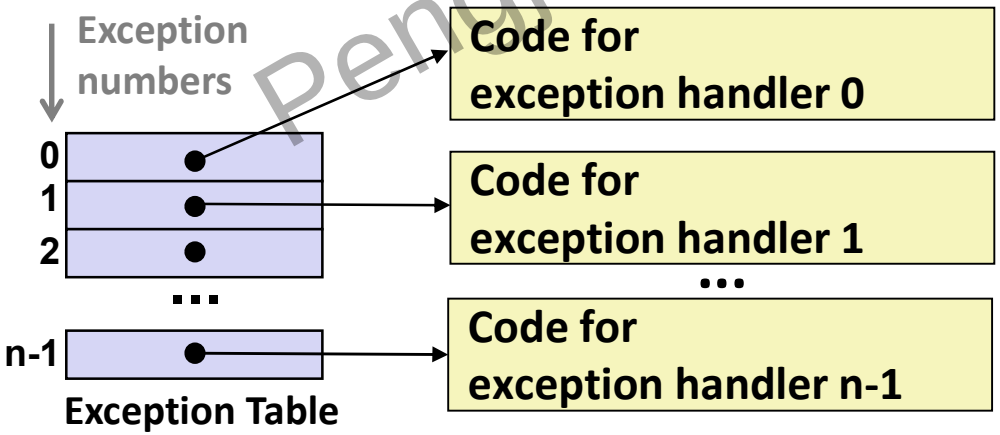
Resource Usage

Exceptions handled by OS Kernel

An *exception* is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)

- Kernel is the memory-resident part of the OS

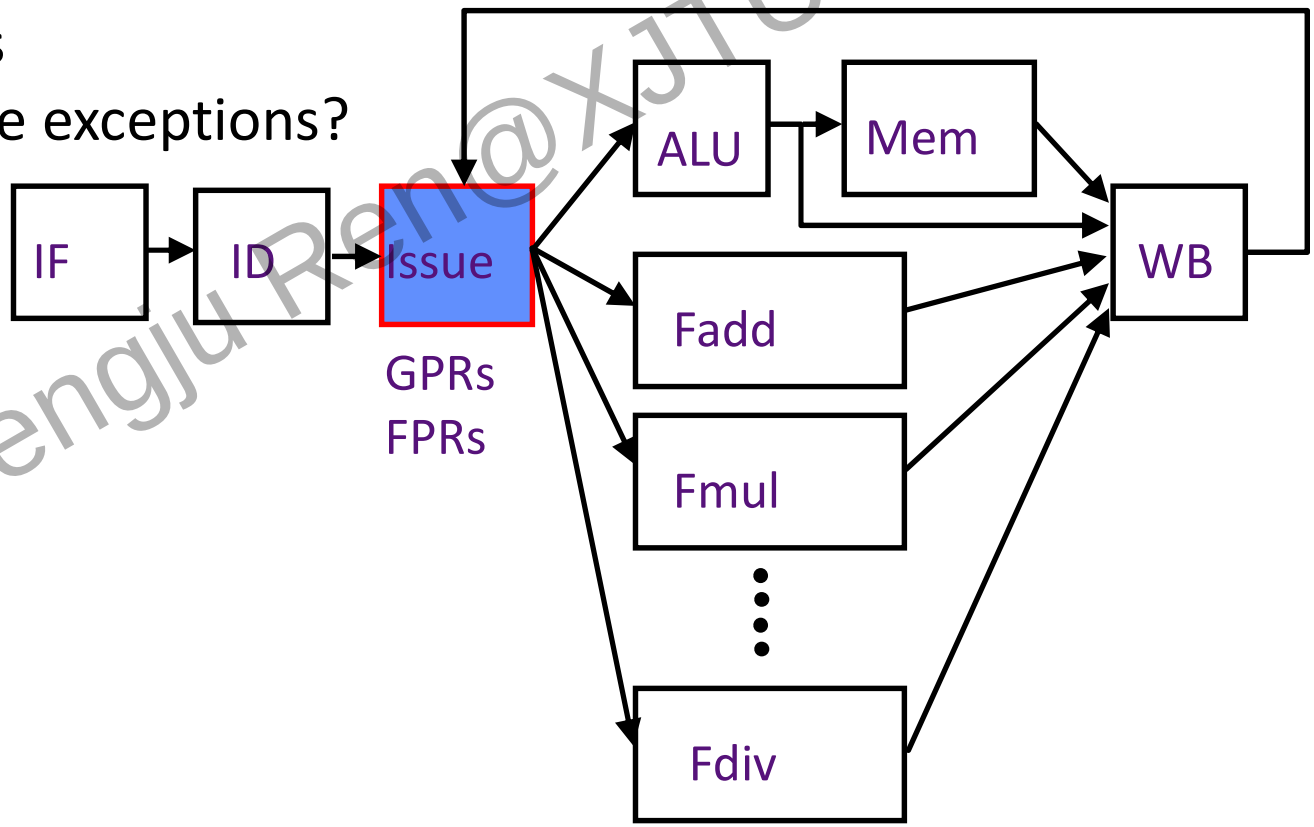




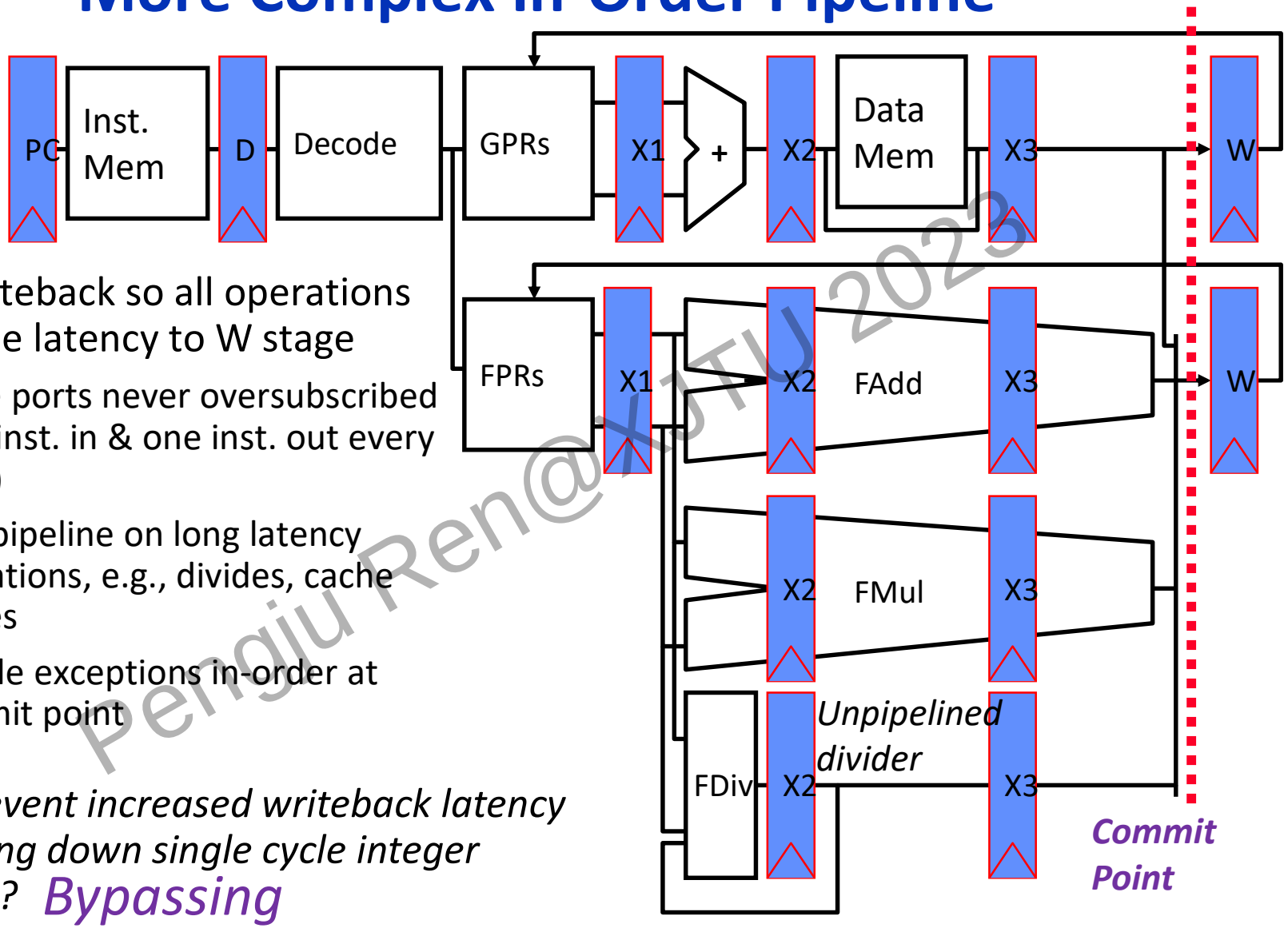
- **Each type of event has a unique exception number k**
- **k = index into exception table (a.k.a. interrupt vector)**
- **Handler k is called each time exception k occurs**

Issues in Complex Pipeline Control

- Structural conflicts at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle
- Structural conflicts at the write-back stage due to variable latencies of different functional units
- Out-of-order write hazards due to variable latencies of different functional units
- How to handle exceptions?



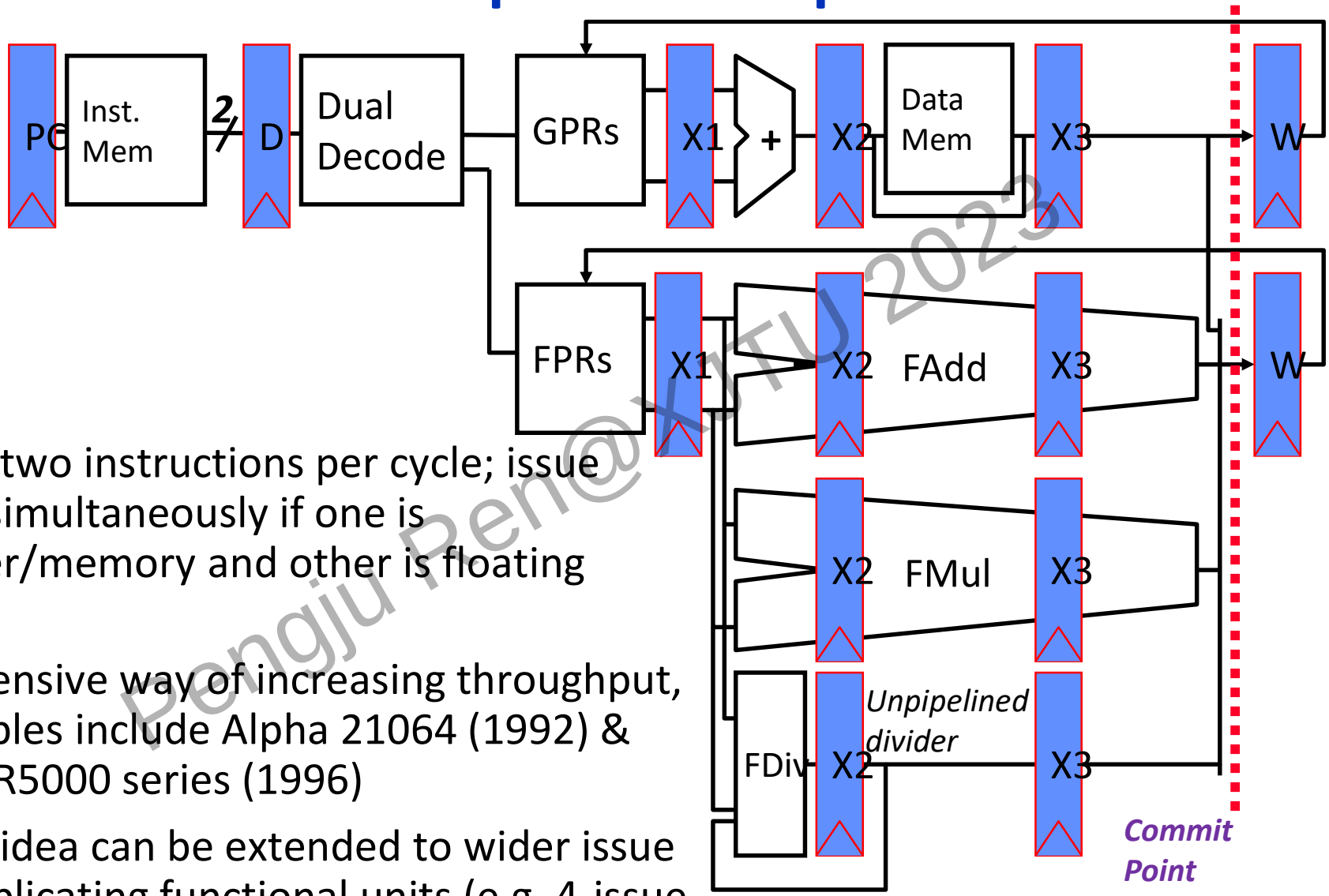
More Complex In-Order Pipeline



- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)
  - Stall pipeline on long latency operations, e.g., divides, cache misses
  - Handle exceptions in-order at commit point

How to prevent increased writeback latency from slowing down single cycle integer operations? *Bypassing*

In-Order Superscalar Pipeline



- Fetch two instructions per cycle; issue both simultaneously if one is integer/memory and other is floating point
- Inexpensive way of increasing throughput, examples include Alpha 21064 (1992) & MIPS R5000 series (1996)
- Same idea can be extended to wider issue by duplicating functional units (e.g. 4-issue UltraSPARC & Alpha 21164) but regfile ports and bypassing costs grow quickly

Next Lecture： SuperScalar Processor

(Instruction level parallel)