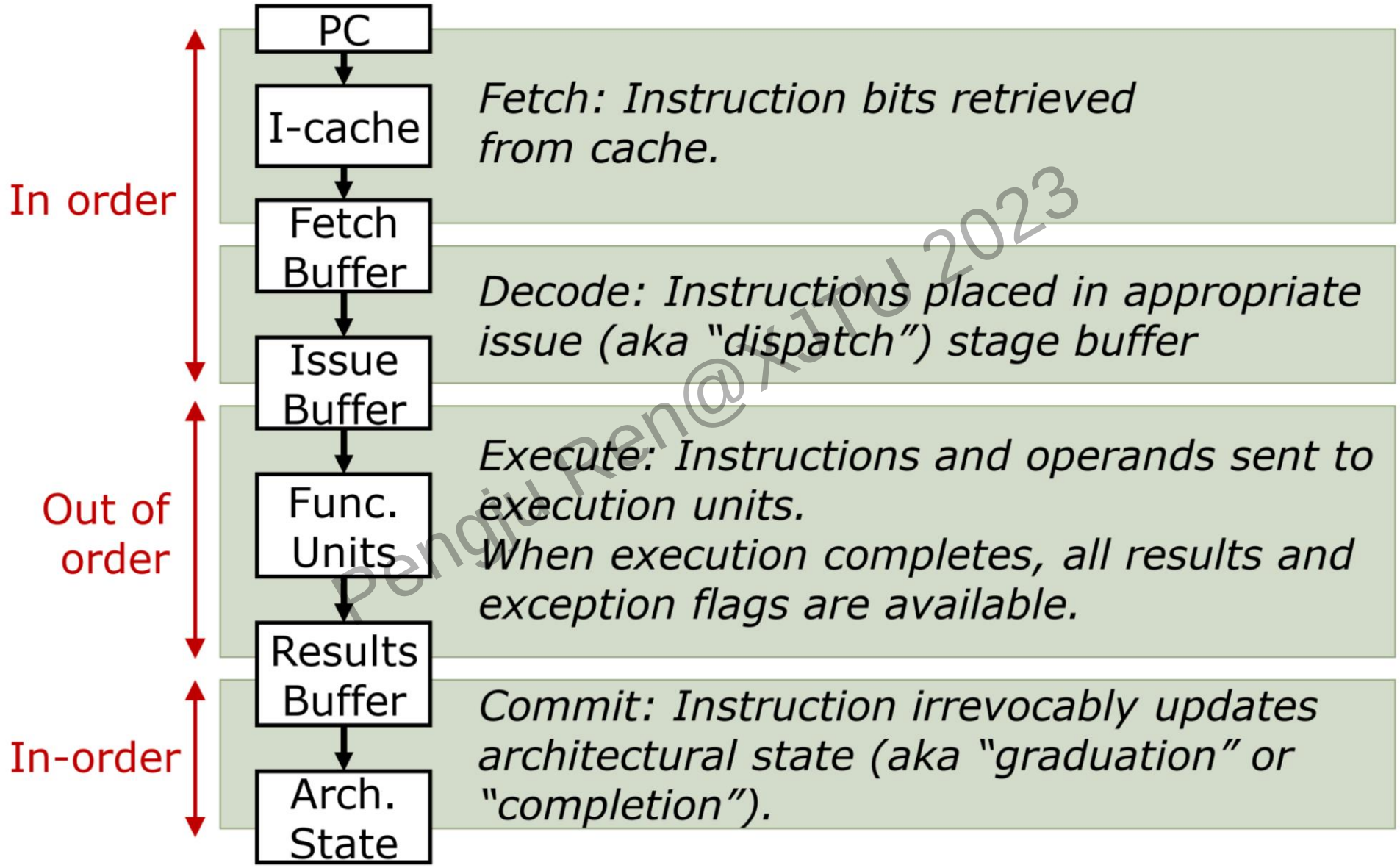
Computer Architecture

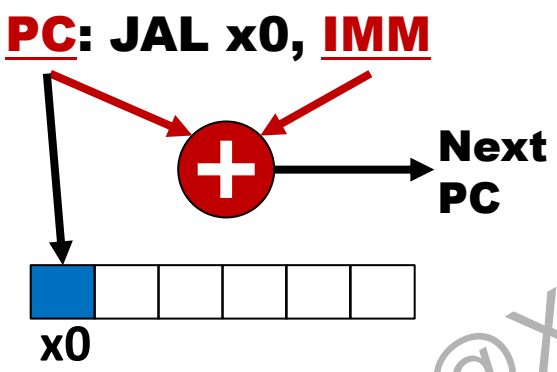
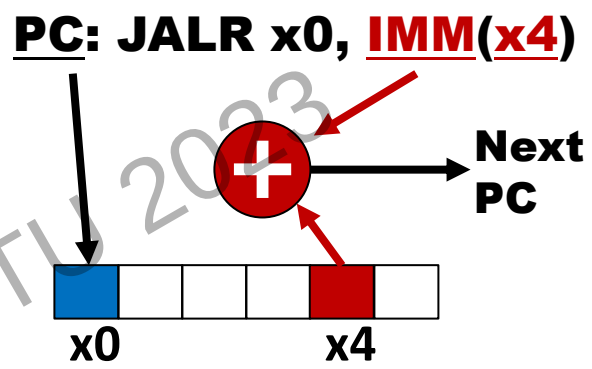
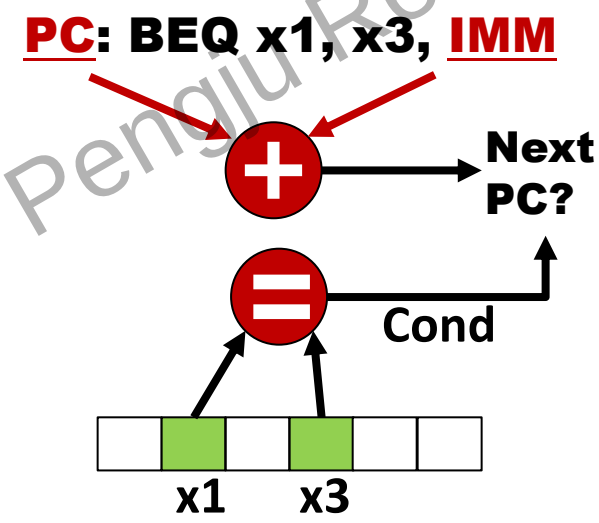
Lecture 08– Branch Prediction

Tian Xia
Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University
http://gr.xjtu.edu.cn/web/pengjuren

Recap: Phases of Instruction Execution of OoO



Jump/Branch Instructions (in RISC-V)

	PC-Relative Address	Absolute Address
Unconditional Jump		
Conditional Jump (Branch)	 (BEQ,BNE,BLT,BLTU,BGT,BGTU)	■ No conditional absolute address branch in RISC-V ■ Exists in other ISA -- e.g. conditional BEQ in ARM

Jump/Branch Instructions (in RISC-V)

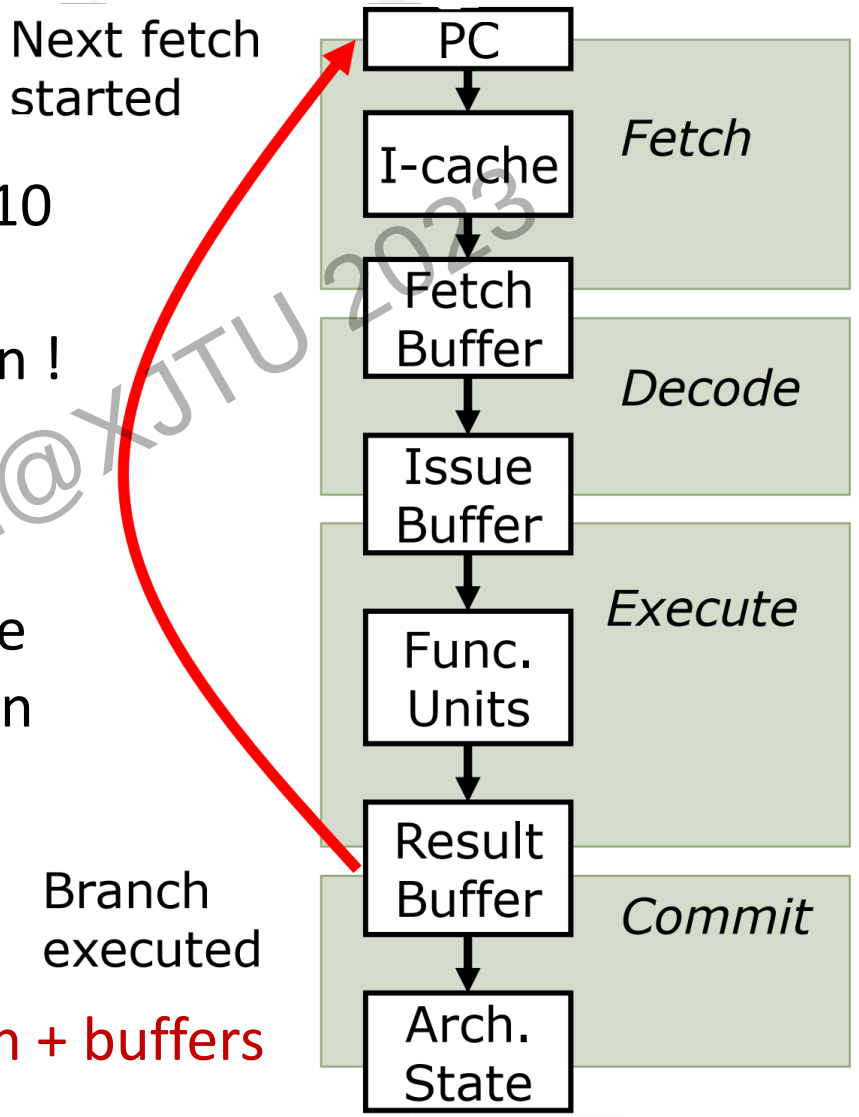
	PC-Relative Address	Absolute Address
Unconditional Jump	If Jump? YES Jump Where? Known Fast	If Jump? YES Jump Where? Known Slow (depend on regs)
Conditional Jump (Branch)	If Jump? Conditional (depend on regs) Jump Where? Known Fast	Jump Destination Prediction Branch Direction Prediction

Incorrect Control-Flow Penalty

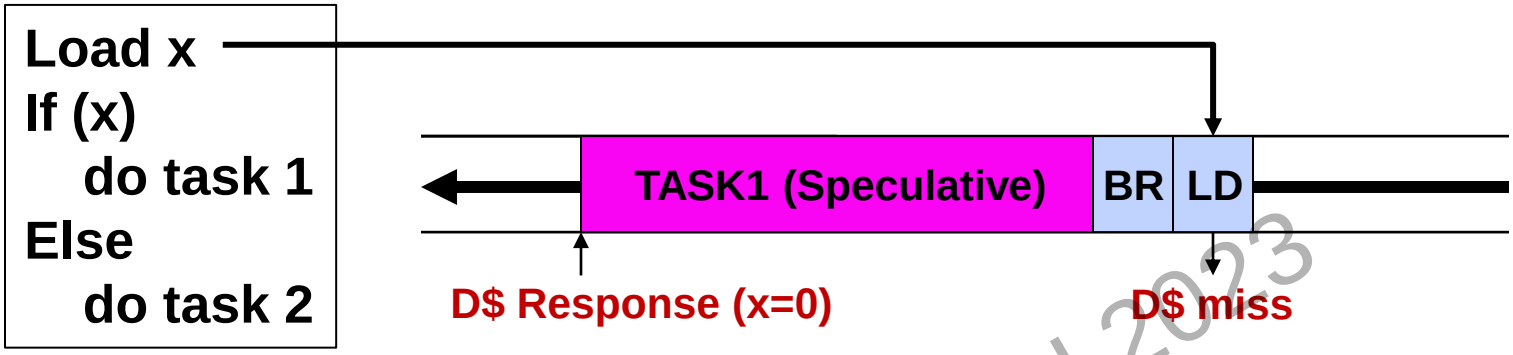
Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution !

How much work is lost if pipeline doesn't follow correct instruction flow?

- Flush pipeline
- Flush reorder buffer
- ~ Loop length * pipeline width + buffers

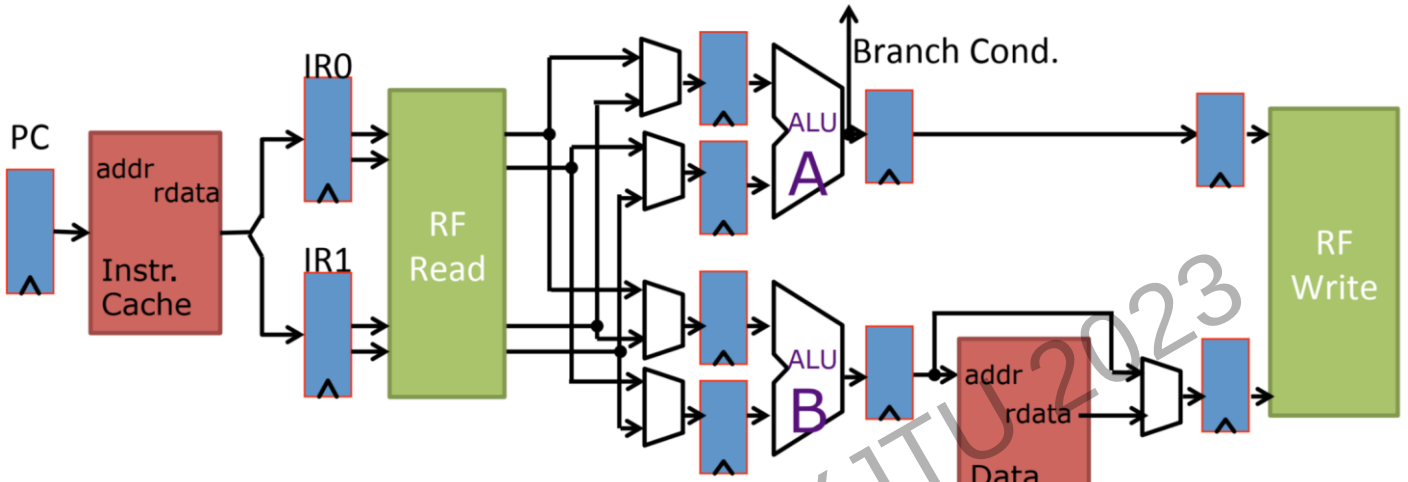


Importance of Branch Prediction



- Consider **4-way** superscalar with **8 pipeline stages** from fetch to dispatch, and **80-entry ROB**, and 3 cycles from issue to branch resolution
- On a misprediction, could throw away 8*4+(80-1)=111 instructions (in **worst case**)
- Improving from 90% to 95% prediction accuracy, removes 50% of branch mispredictions
  - If 1/6 instructions are branches, then move from 60 instructions between mispredictions, to 120 instructions between mispredictions

Misprediction Penalty in SuperScalars

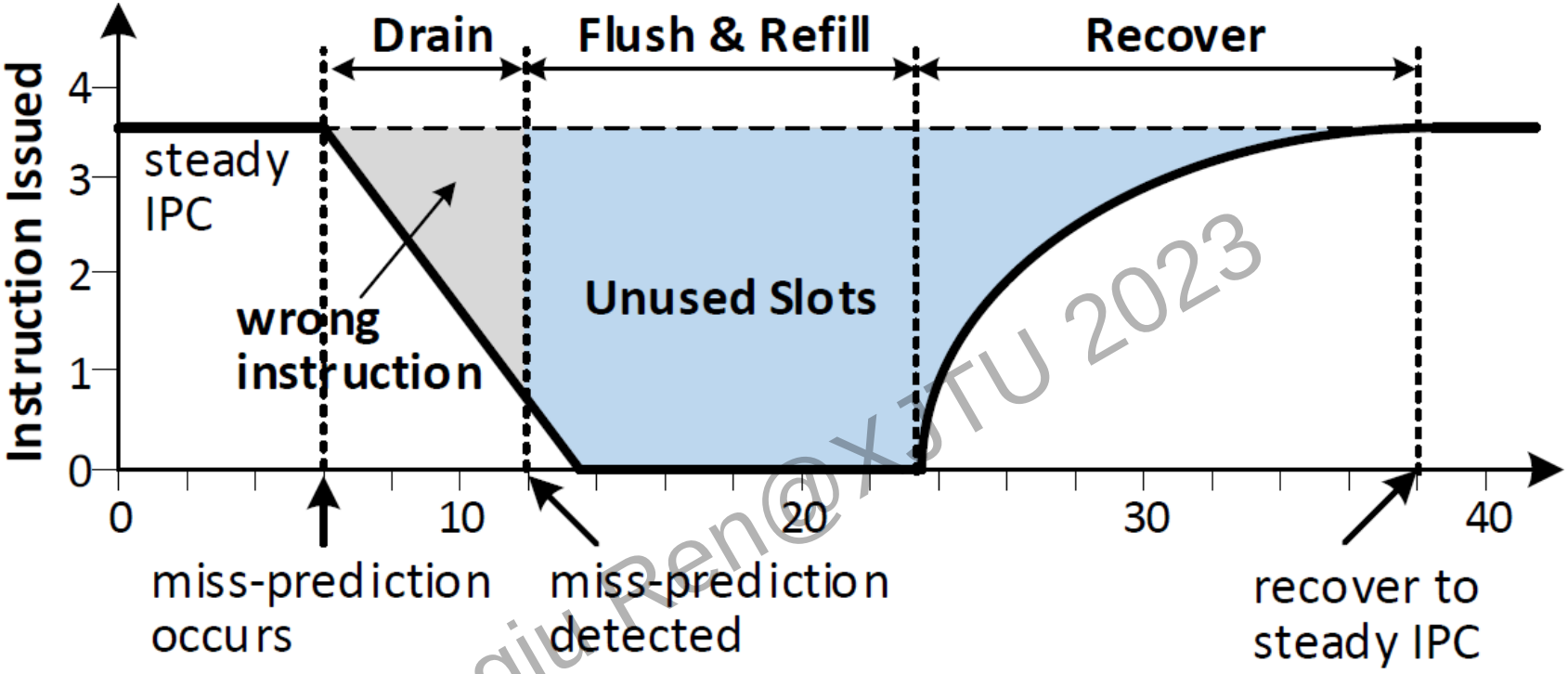


Misprediction Detect

BEQ	F	D	I	A0	A1	W				
OpA	F	D	I	B0	-	-				
OpB		F	D	I	-	-	-			
OpC			F	D	I	-	-	-		
OpD				F	D	-	-	-	-	
OpE				F	D	-	-	-	-	
OpF					F					
OpG					F					

Maximal Wasted Pipeline Slots

Misprediction Penalty in OoO Pipeline



- Steady instruction **issue throughput (Insn-Per-Cycle, IPC)** is broken
- **Drain**: linearly drop to 0
- **Refill**: wait for useful instructions reach issue port
- **Recover**: issue bandwidth grows in curve
- Wasted cycles = unused_slots / issue_throughput
  - Typically 10—20 cycles in Intel Xeon CPU (4 issue bandwidth)

Average Run-Length between Branches

What is the average run length between braches ?

	SPEC(int)	SPEC(fp)
Branches	**19%**	**11%**
Loads	24%	26%
Stores	10%	7%
Others	47%	56%

Roughly 1 jump for every 5-10 instructions

- Essential in **modern processors** to mitigate branch delay latency
- Two types of Prediction:
  - **Predict Branch Direction** (Taken or not ?)
  - **Predict Branch/Jump Address** (Target address ? PC related and Absolute)
- Typical Ideal Prediction: **>95%**

Average dynamic instruction mix of SPEC CPU 2017 [Limaye and Adegbiya, ISPASS'18]

Reducing Control-Flow Penalty

- Software solutions
  - Eliminate branches - **loop unrolling**
    - Increases the run length
  - Reduce resolution time - **instruction scheduling**
    - Compute the branch condition as early as possible (of limited value because branches often in critical path through code)
- Hardware solutions
  - Find something else to do (**delay slots**)
    - Replaces pipeline bubbles with useful work (requires software cooperation) – quickly see diminishing returns
  - **Speculate, i.e., branch prediction**
    - Speculative execution of instructions beyond the branch
    - Many advances in accuracy, widely used

Branch Prediction

Motivation:

Branch penalties limit performance of deeply pipelined processors

Modern branch predictors have high accuracy ***(>95%)*** and can reduce branch penalties significantly

Required hardware support:

Prediction structures:

- *Branch history tables*（**BHT**）, *branch target buffers*（**BTB**）, etc.

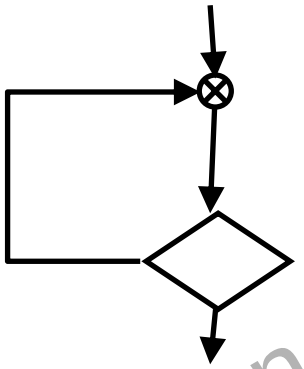
Misprediction recovery mechanisms:

- Keep result computation separate from commit
- Kill instructions following branch in pipeline（a.k.a. flush）
- Restore state to that following branch

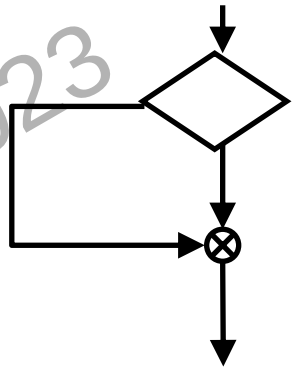
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

Loop:
Backward 90%



If-Else:
Forward 50%

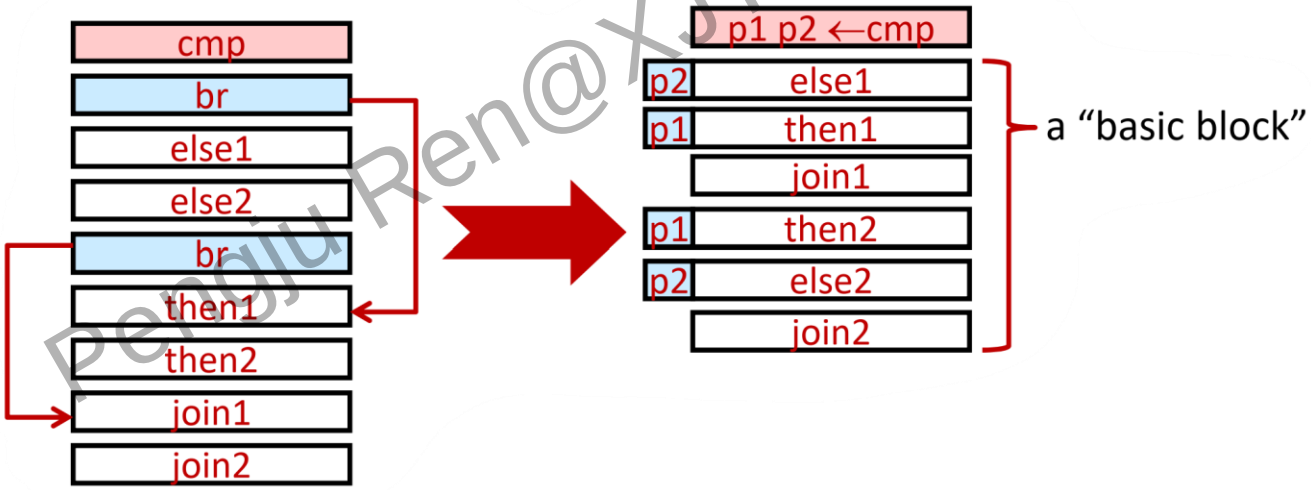


ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110

- bne0 *(preferred taken)* beq0 *(not taken)*
- typically reported as ~80% accurate

Speculating Both Directions?

- An alternative to branch prediction is to **execute both directions** of a branch speculatively (Predicated Execution)
  - **Resource requirement** is proportional to the number of concurrent speculative executions
  - Only **half the resources** engage in useful work when both directions of a branch are executed speculatively



- With accurate branch prediction, it is **more cost effective** to dedicate all resources to the predicted direction!

Dynamic Branch Prediction

Learning based on past behavior

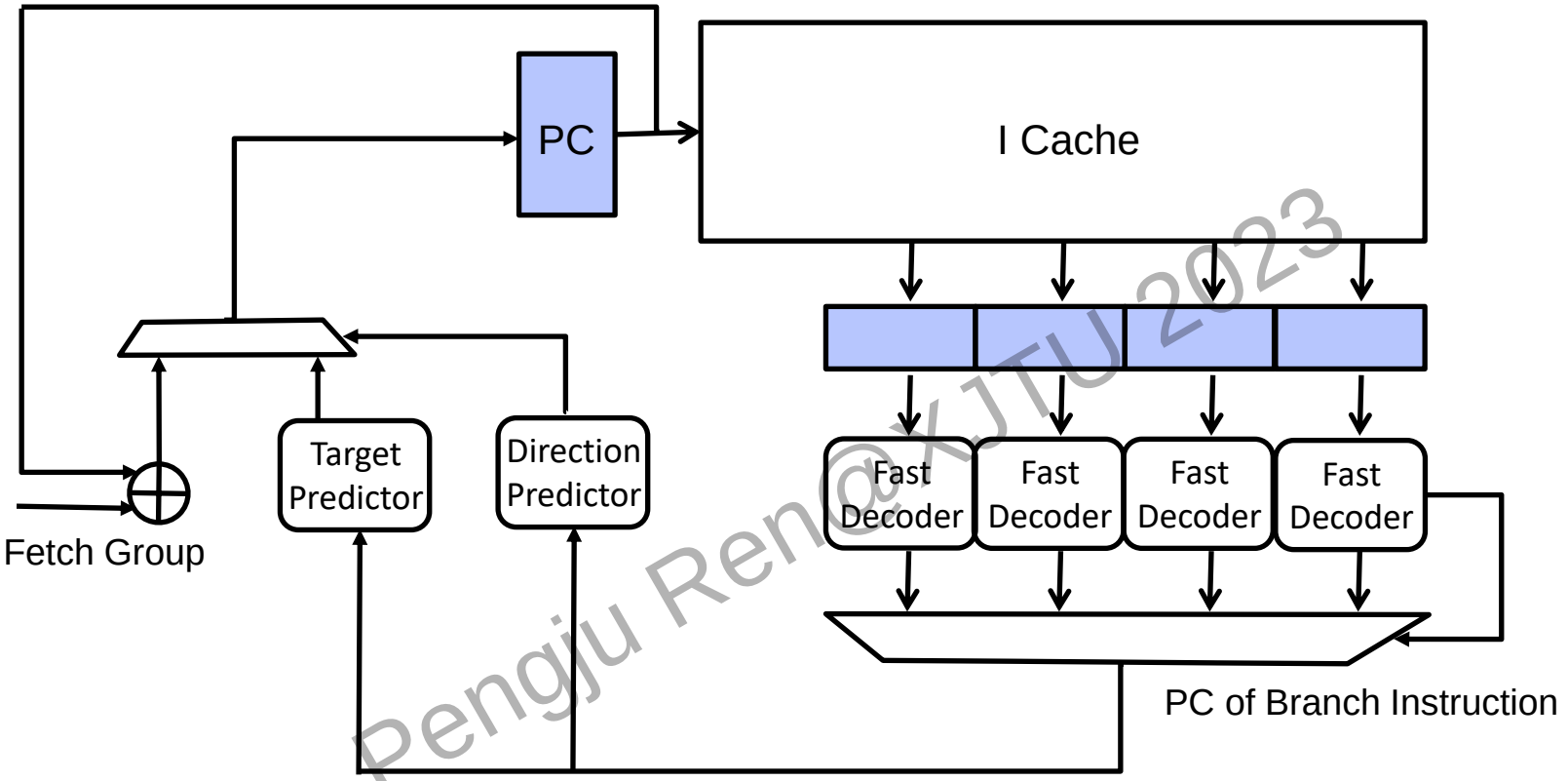
- Branch behavior is monitored during program execution
  - History data can influence prediction of future execution of the branch instruction
- Temporal correlation
  - The way a branch resolves may be a good predictor of the way it will resolve at **the next execution**

```
for (i=0; i<ITER; i++)
    y[i] += x[i];
```

- Spatial correlation
  - **Several branches** may resolve in a highly correlated manner (a preferred path of execution)

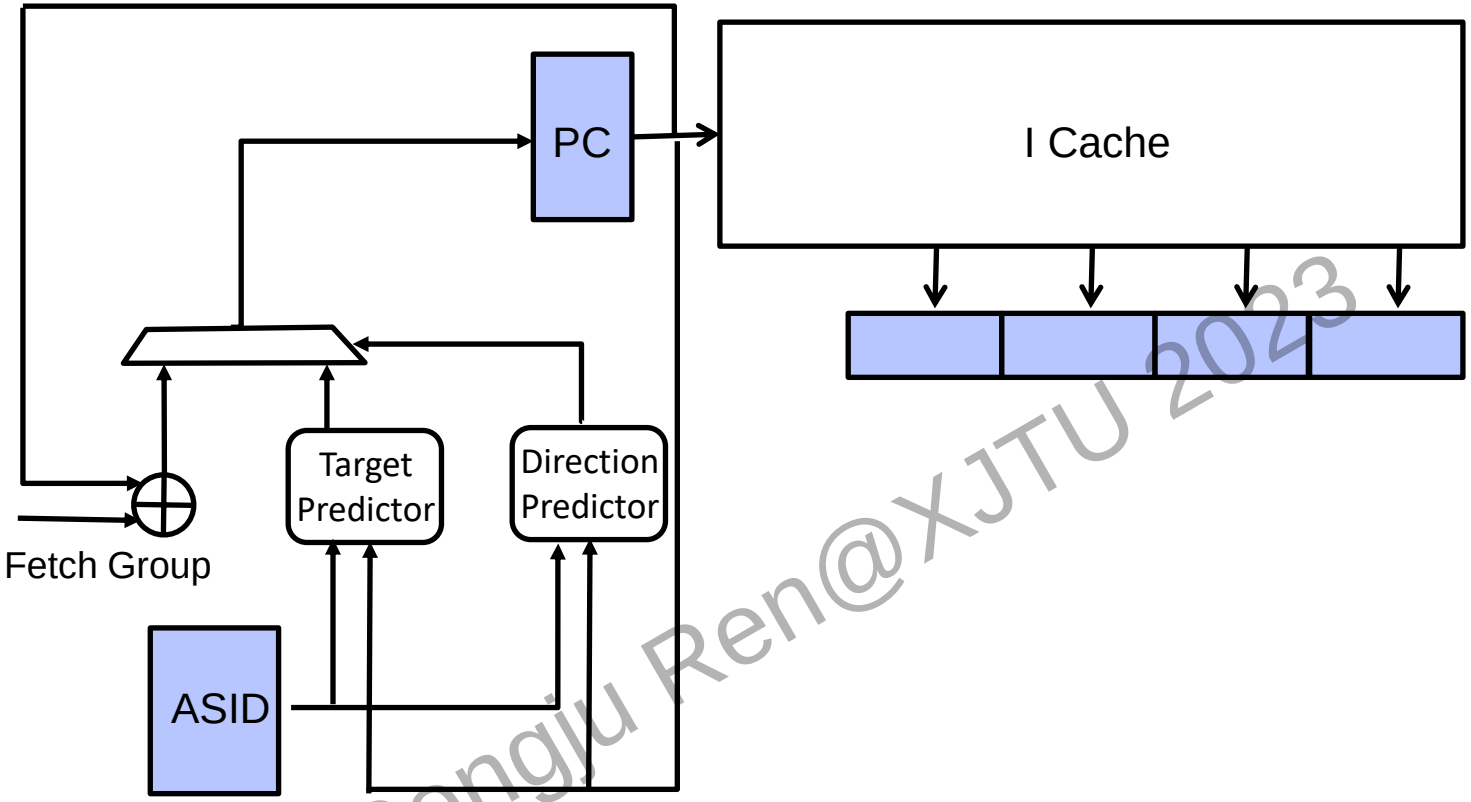
```
if (x[i] < 7) then
    y += 1;
if (x[i] < 5) then
    c -= 4;
```

Prediction based on Instruction



- **Fast decode** instructions belonging to one fetch group, only detect whether it is branch or not ?
- Some design doing **pre-decode** when moving Instructions from L2$ to L1I$ (with **Branch Flag**)

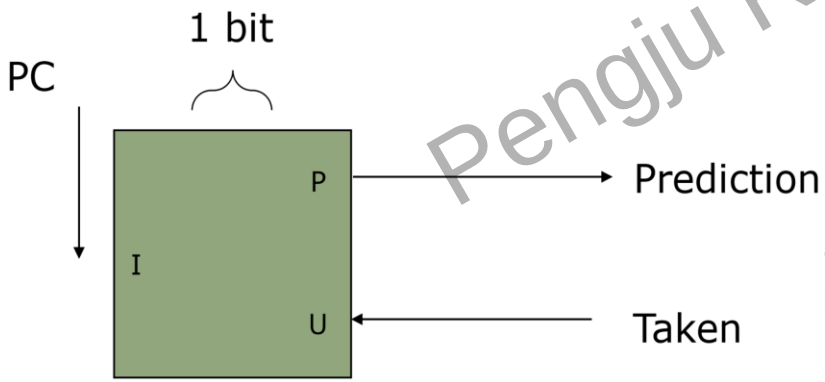
Prediction based on PC

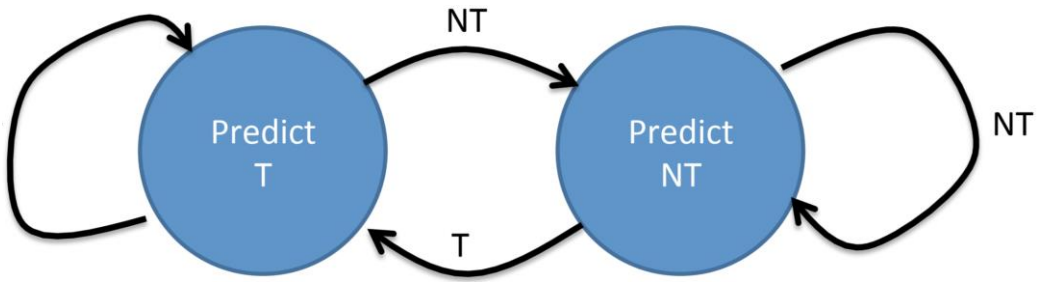


- The Physical Address of an instruction of a program is changeable (Depends on OS Page Scheduler)
- The **Virtual Address of an instruction of a program is fixed after compilation**
- ASID is used together with PC for prediction (Otherwise, Branch predictor need to be clear after context switch)

One-Bit Branch History Predictor

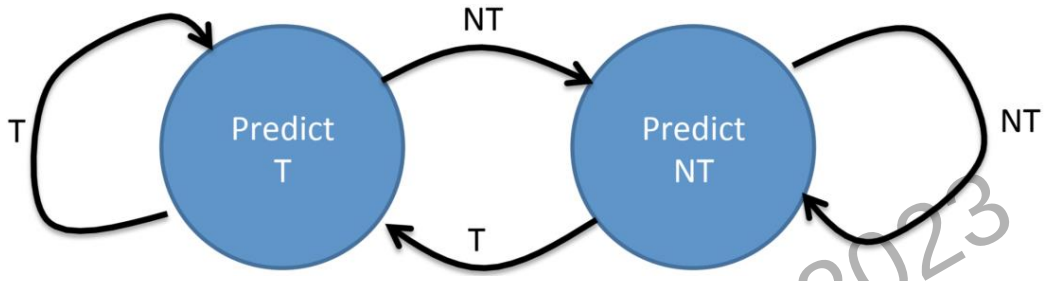
- For each branch, remember **last way** branch went
- Has problem with **loop-closing** backward branches, as two mispredictions occur on every loop execution
  1. **first iteration** predicts loop backwards branch not-taken (loop was exited last time)
  2. **last iteration** predicts loop backwards branch taken (loop continued last time)





1-bit Saturating Counter

1-Bit Saturating Counter



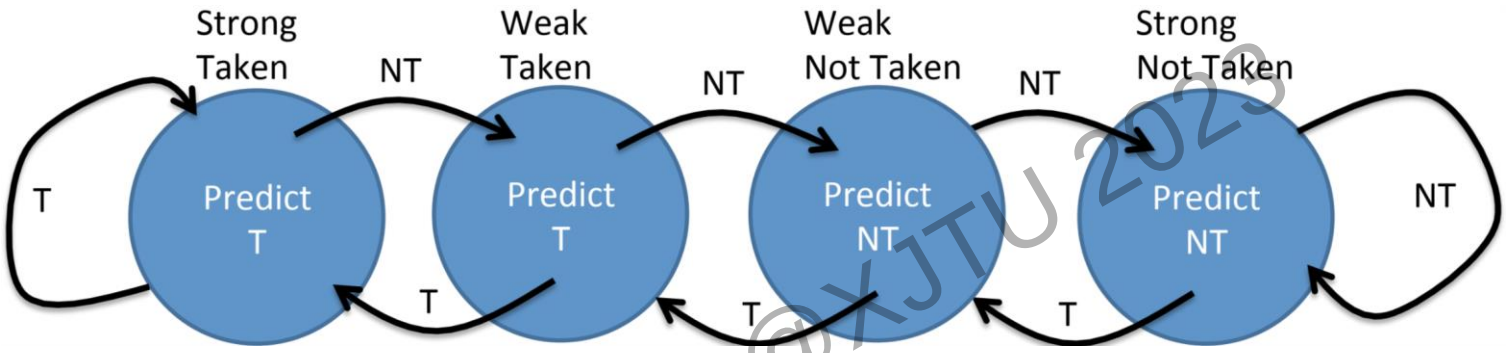
Iteration	Prediction	Actual	Mispredict?
1	**NT**	**T**	**Yes**
2	T	T	No
…	T	T	…
N-1	T	T	No
N	**T**	**NT**	**Yes**

What happens on loop branches ?

At best, two mispredictions for every use of loop

2-Bit Branch Prediction

- Assume 2 BP bits per instruction instead of one
- Change the prediction after ***two consecutive mistakes***!

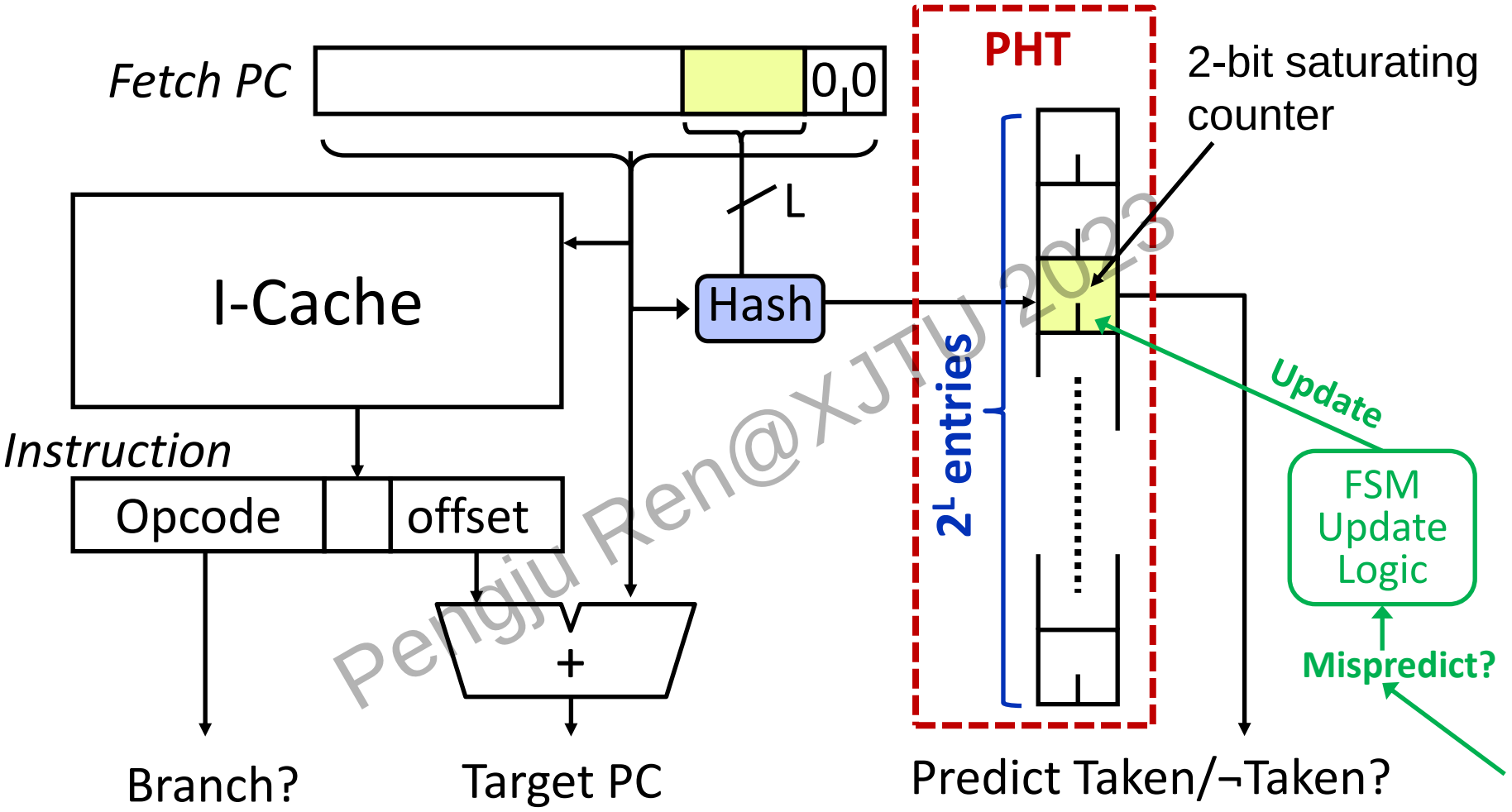


i	Prediction	Actual	Mispredict?	State
1	**NT**	**T**	**Yes**	**Strong NT**
2	**NT**	**T**	**Yes**	**Weak NT**
3	T	T	No	Weak T
4	T	T	No	Stong T
..	…	…	No	..
N	**T**	**N**	**Yes**	**Strong T**
1	T	T	No	Weak T

i	Prediction	Actual	Mispredict?	State
1	**NT**	**T**	**Yes**	**Weak NT**
2	T	T	No	Weak T
3	T	T	No	Stong T
4	T	T	No	..
..	…	…	No	Strong T
N	**T**	**N**	**Yes**	**Strong T**
1	T	T	No	Weak T

BP state:(*predict* take/¬take) x (*last prediction* right/wrong)

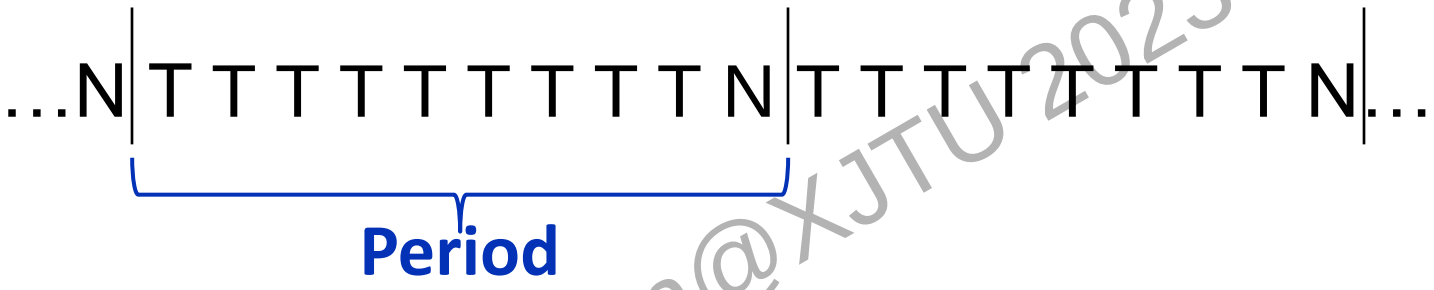
Pattern History Table (PHT)



- 4K-entry PHT（***Why the low bits of PC?***）, 2 bits/entry (~80-90% correct predictions)
- PHT is updated at the **commit stage**

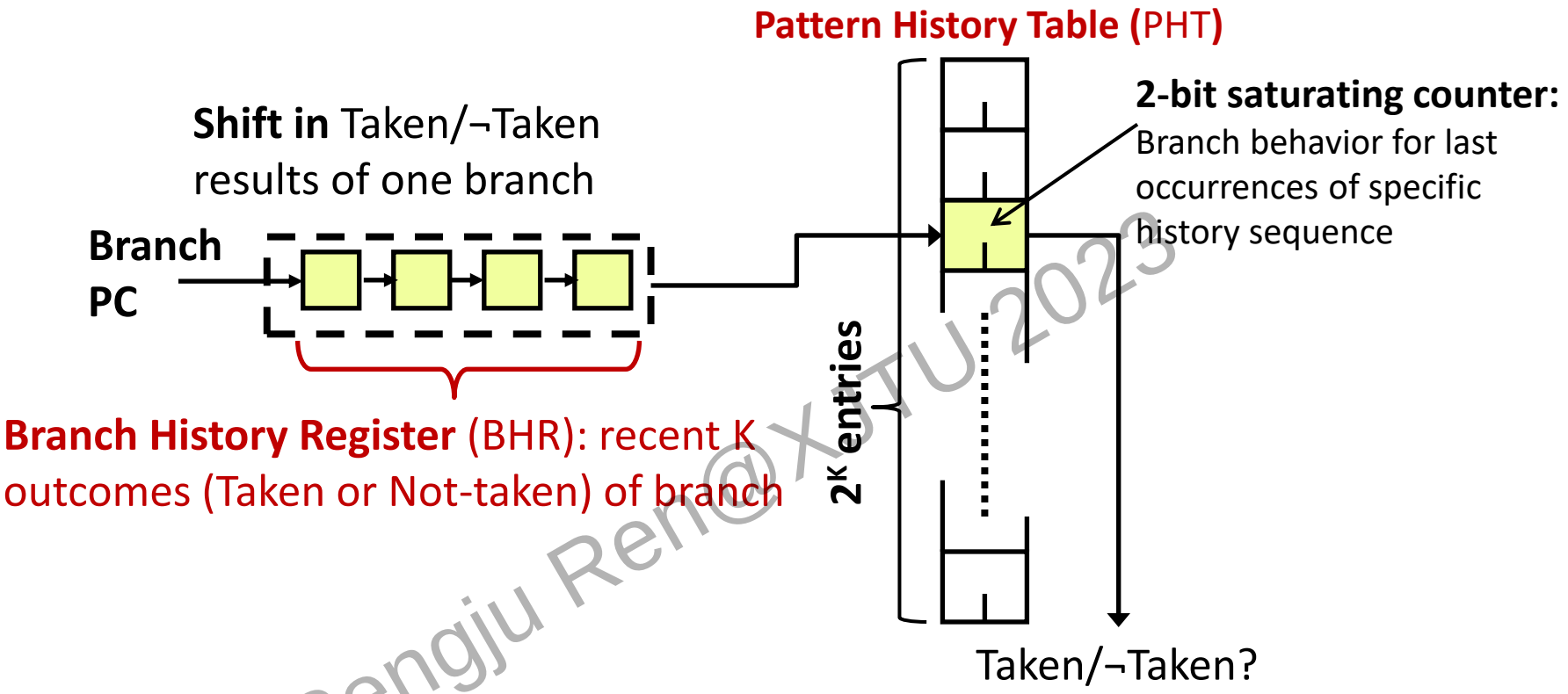
Prediction Based on Local Branch History

```
for (i=0; i<ITER; i++)
    y[i] += x[i];
```



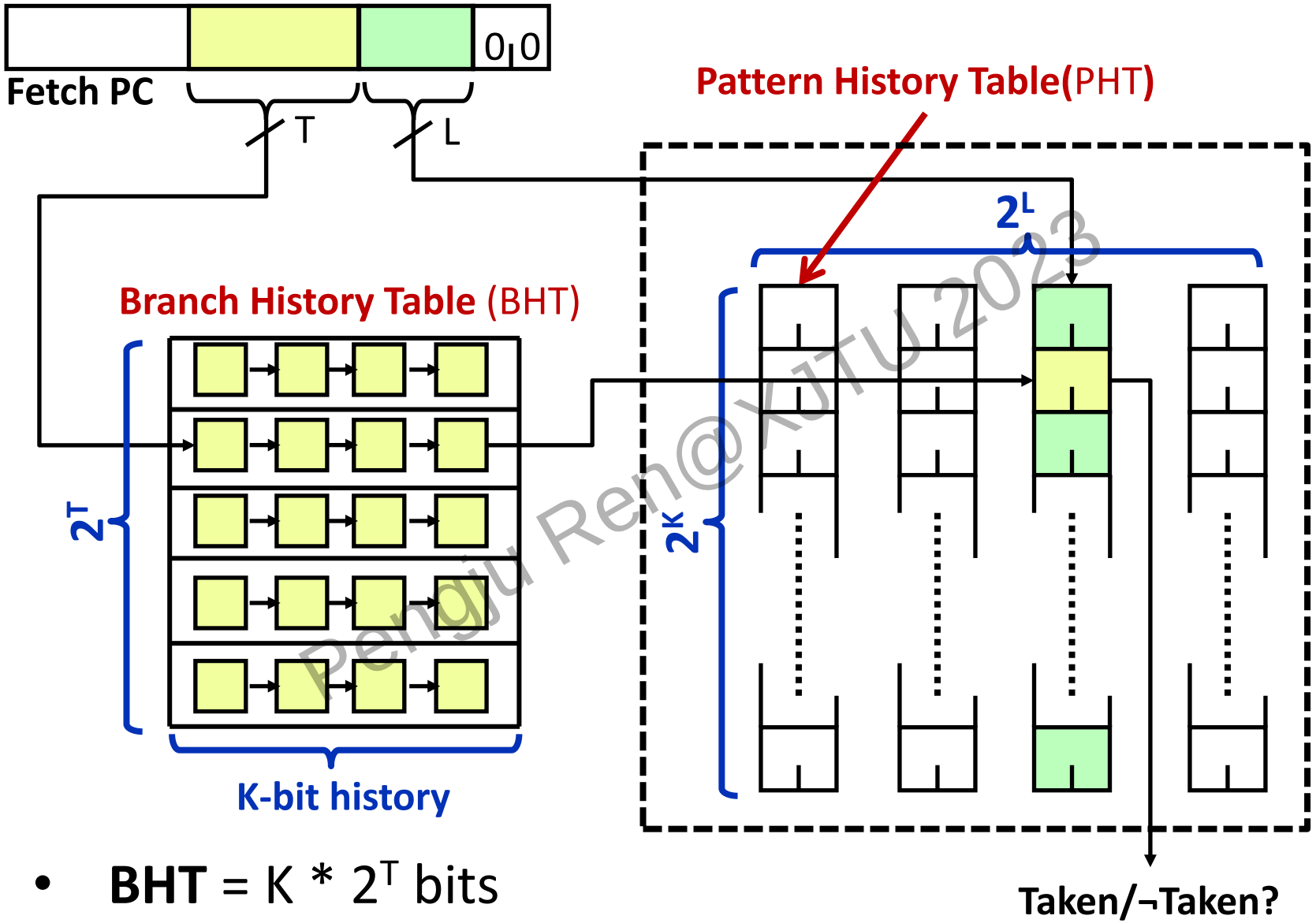
- **Local branch history**: latest branch results of one branch instruction
- In many codes, loops may have **fixed iteration period**
- Use **local history buffer** to capture loop patterns, need to be **larger** than iteration period

Prediction Based on Local Branch History



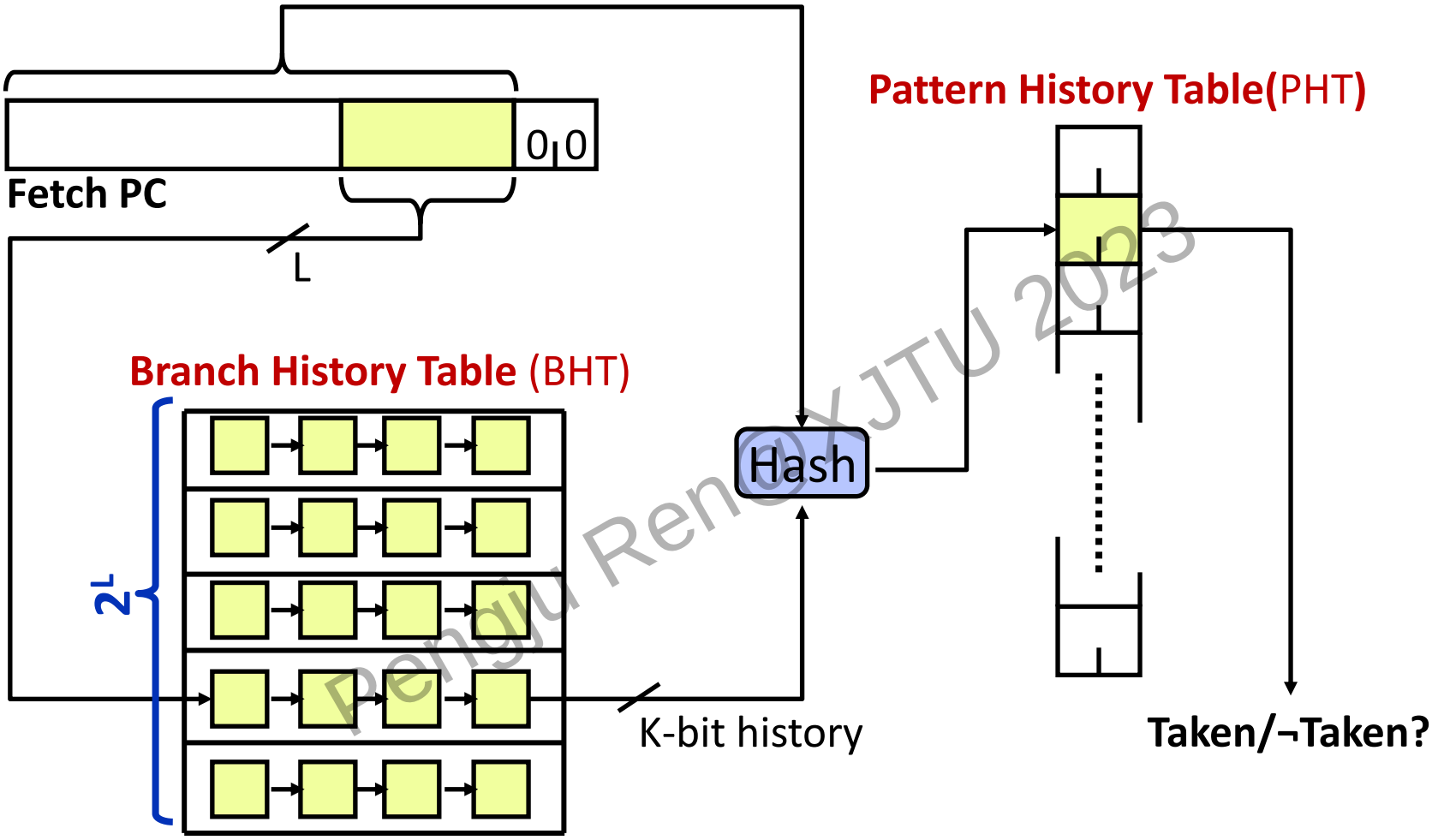
- For **each branch** instruction, record its **latest K** branch results
- For total **2^K possible** history record, put one **2-bit saturating counter** for each case (into a PHT)
- Prediction reach **100%** if K > Loop Period
- **Too expensive** to hold all branch instructions!

Two-Level Branch Predictor



- **BHT** = K * 2^T bits
- **PHTs** = 2^K * 2^L * 2 bits

Two-Level Branch Predictor (Hash)



- Avoid conflict of **different branches** with the **same history**
- Less **hardware resource** requirement

Exploiting Spatial Correlation

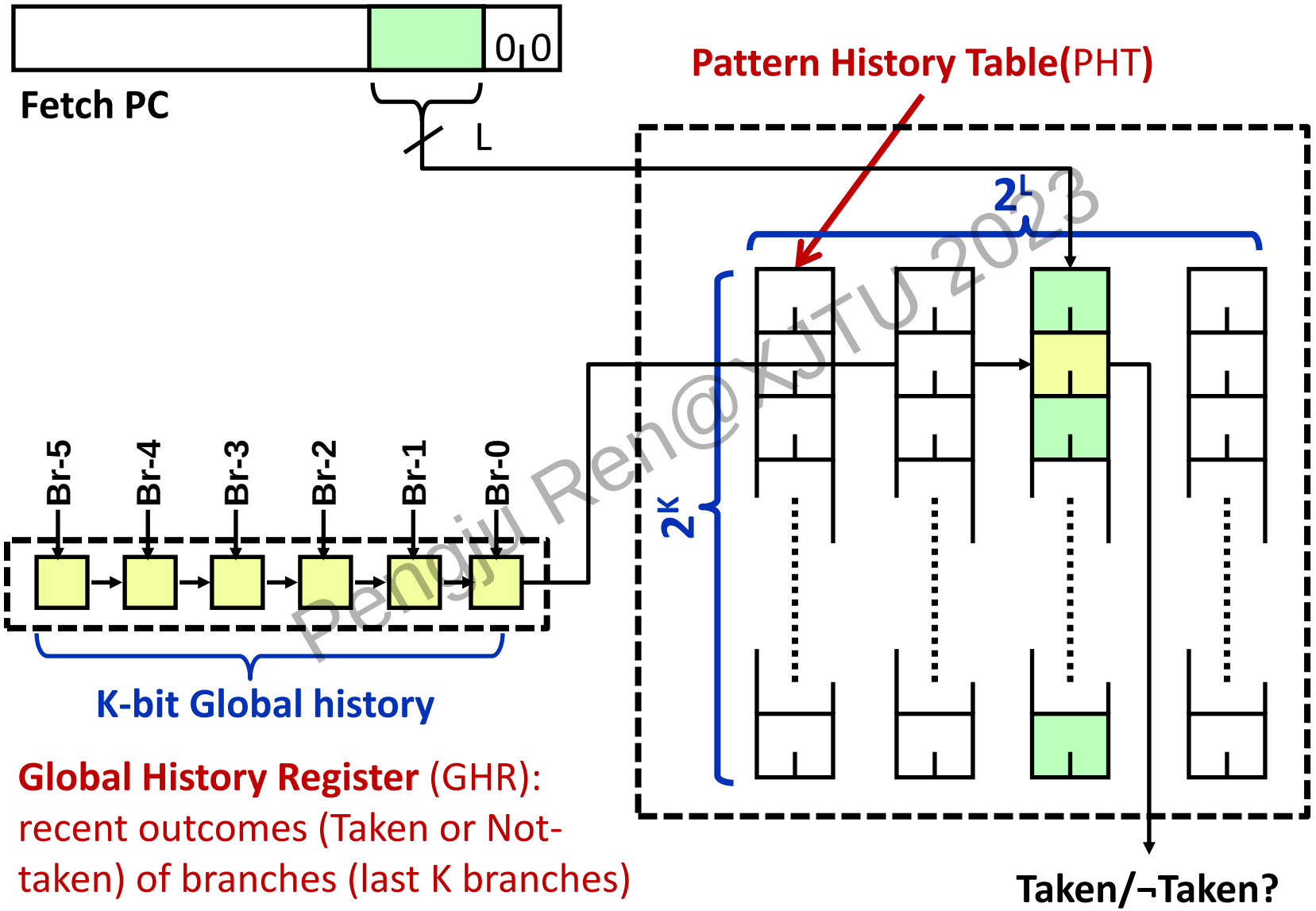
Yeh and Patt, 1992

```
if (x[i] < 7) then
     y += 1;
if (x[i] < 5) then
     c -= 4;
```


- If **first** condition false, **second** condition also false
- Prediction must be a function of own branch as well as recent outcomes of other branches.

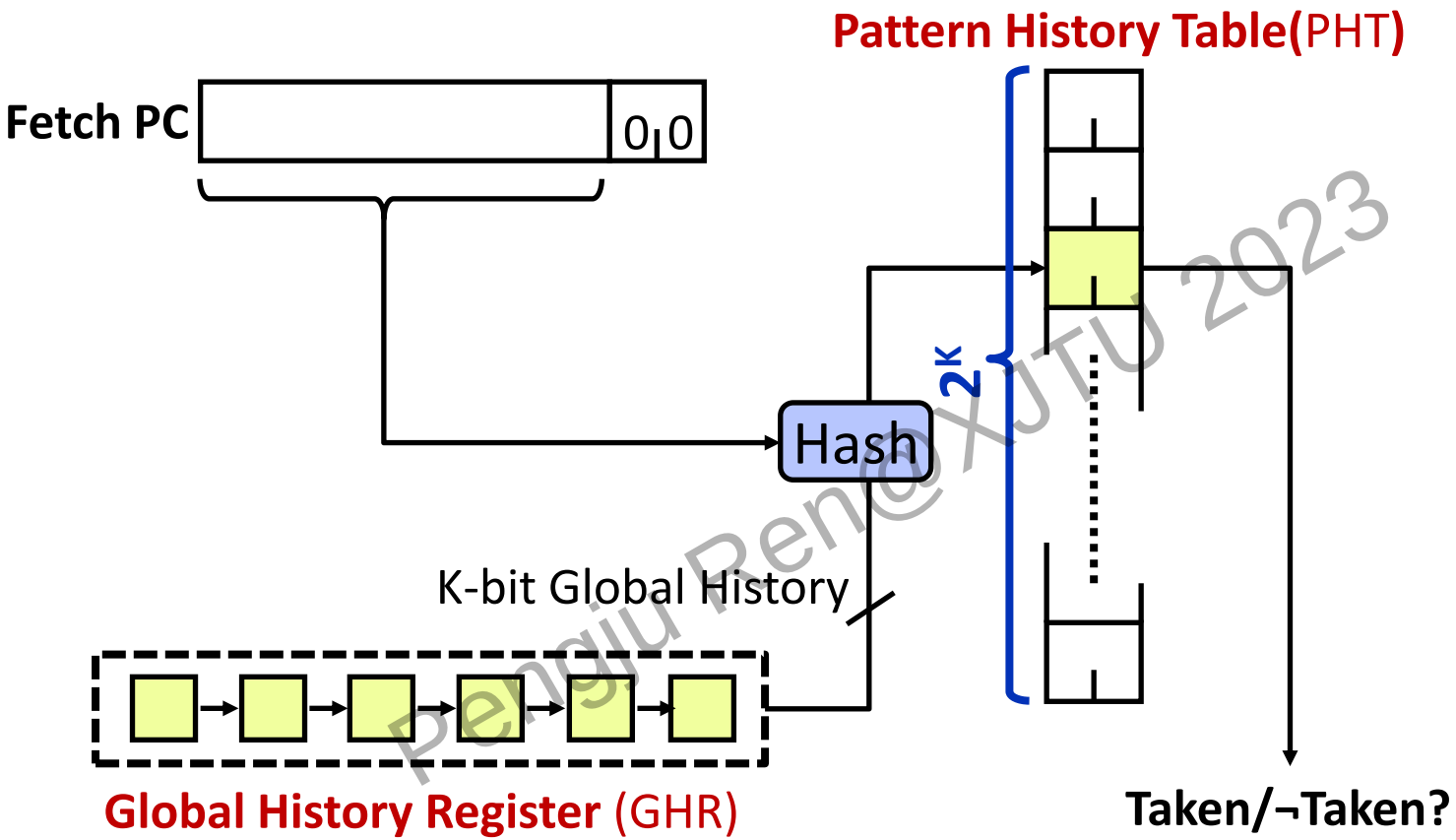
Global branch history: the direction of the last N branches executed by the processor

Two-Level Branch Predictor
(Global history based prediction)



Global History Register (GHR): recent outcomes (Taken or Not-taken) of branches (last K branches)

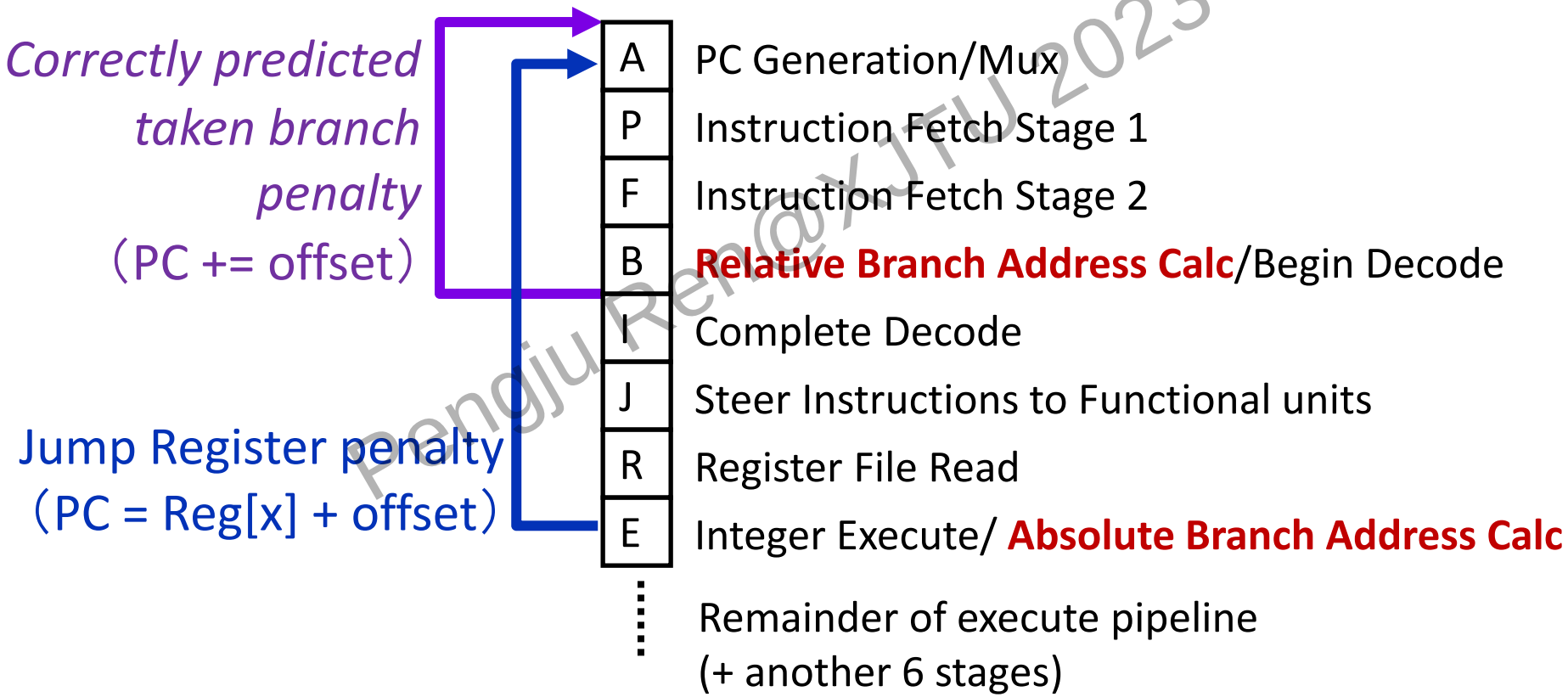
Two-Level Branch Predictor (Hash)
(Global history based prediction)



- All branch instructions share one PHT
- Can blend branch PC with GHR bits to avoid conflicts

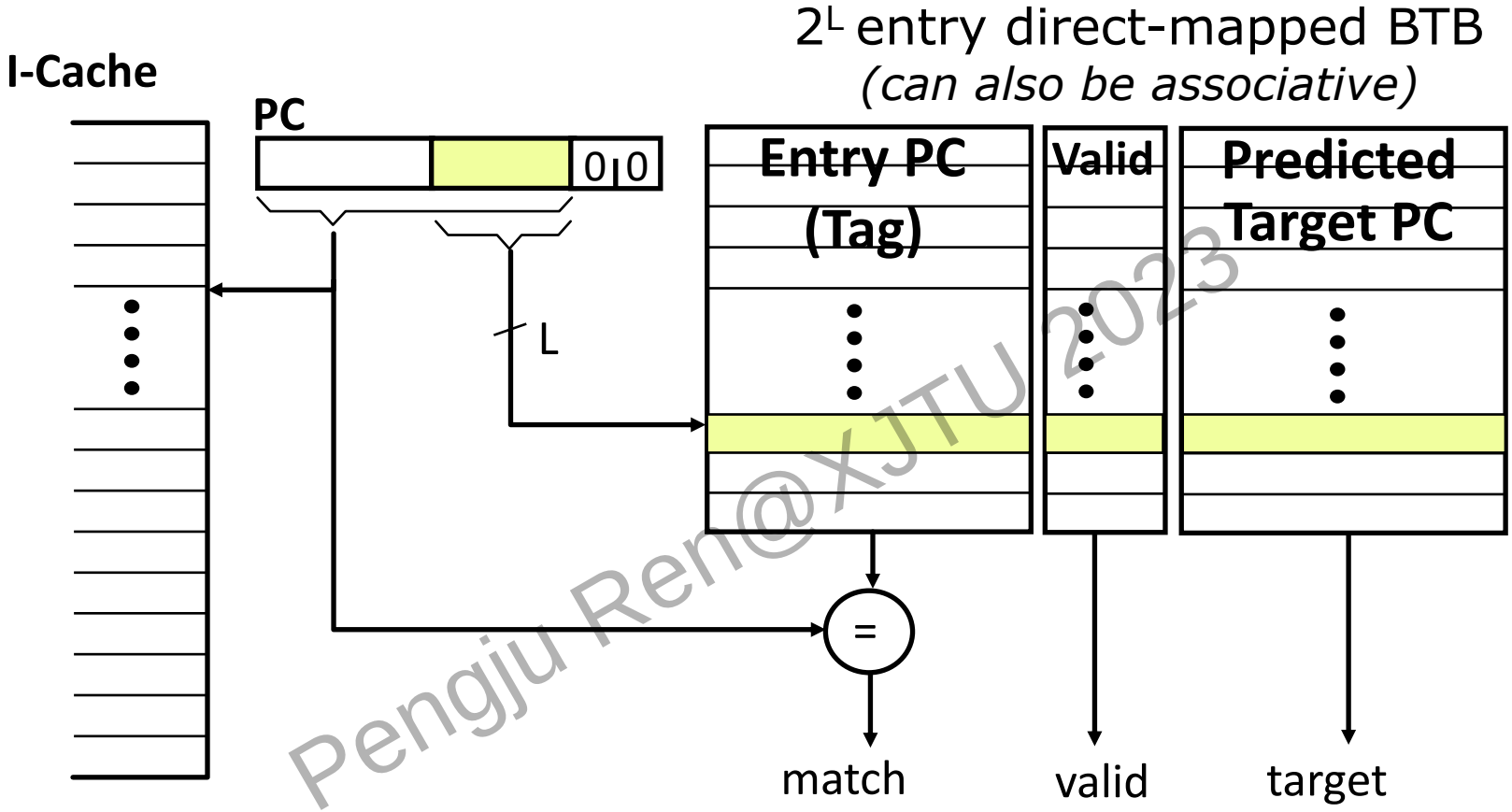
Limitations of BHT/PHTs

- If predict branch **taken**, need to know **WHERE TO JUMP**!
- BHT/PHT only predicts **branch direction**. Therefore, cannot redirect fetch stream until after **branch target** is determined.



UltraSPARC-III fetch pipeline

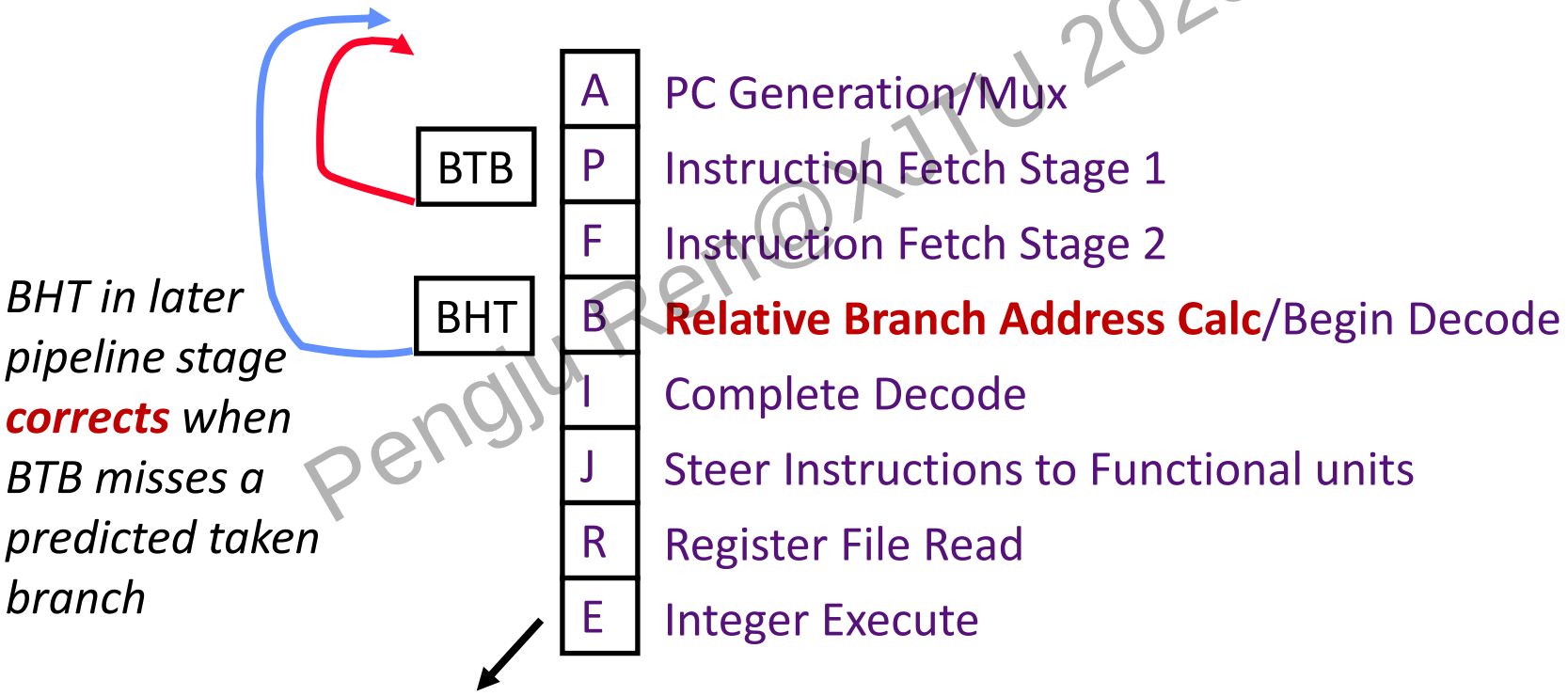
Branch Target Buffer (BTB)



- Keep both the branch PC and **target PC** in the BTB
- Only ***taken branches*** and ***jumps*** held in BTB (to save space)
- If PC doesn't exist in BTB: PC+4 is fetched (continue as **non-taken**)
- Next PC determined ***before* the branch instruction is fetched**
  -- Useful for both **absolute branch** and **PC-relative branch**

Combining BTB and BHT

- BTB entries are considerably **more expensive** than BHT, but can redirect fetches **at earlier stage** in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is **more accurate**



Absolute Address Jump/Branch
with uses of Jump Register (JALR)

- Switch statements (jump to address of matching case)

  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)

  BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

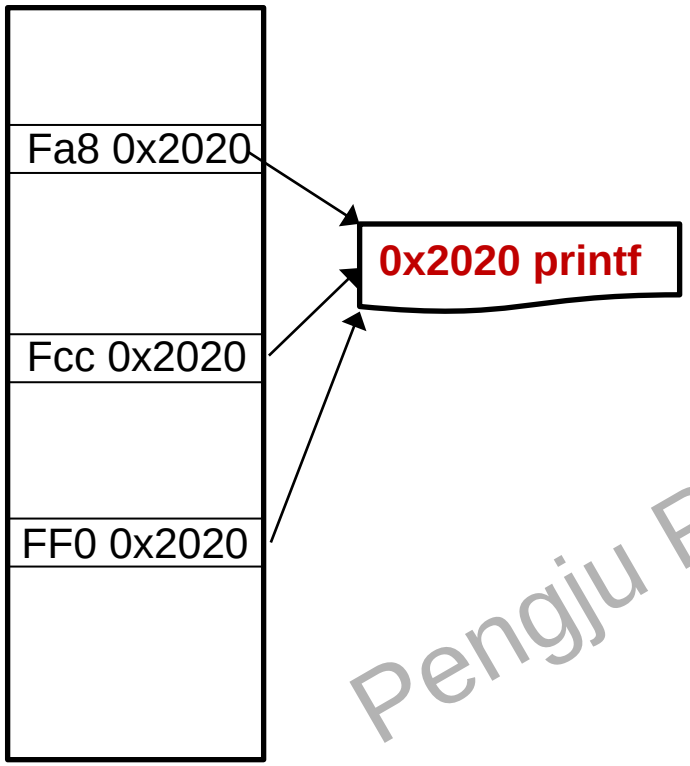
- Subroutine returns (jump to return address)

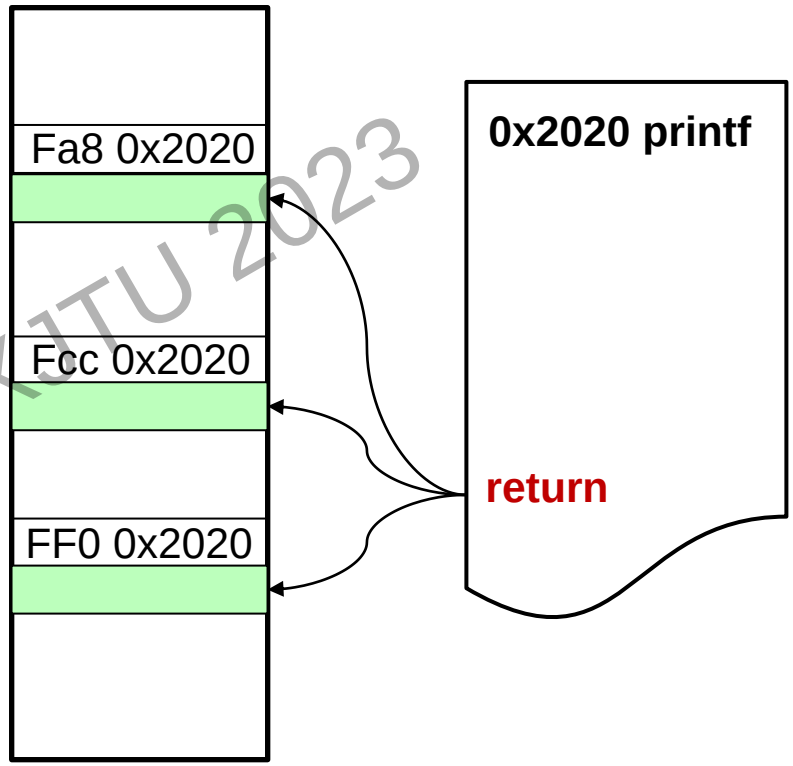
  BTB works well if usually return to the same place

  ⇒ *Often one function called from many distinct call sites!*

  How well does BTB work for each of these cases?

Absolute Address Jump/Branch
(Function calls and returns)





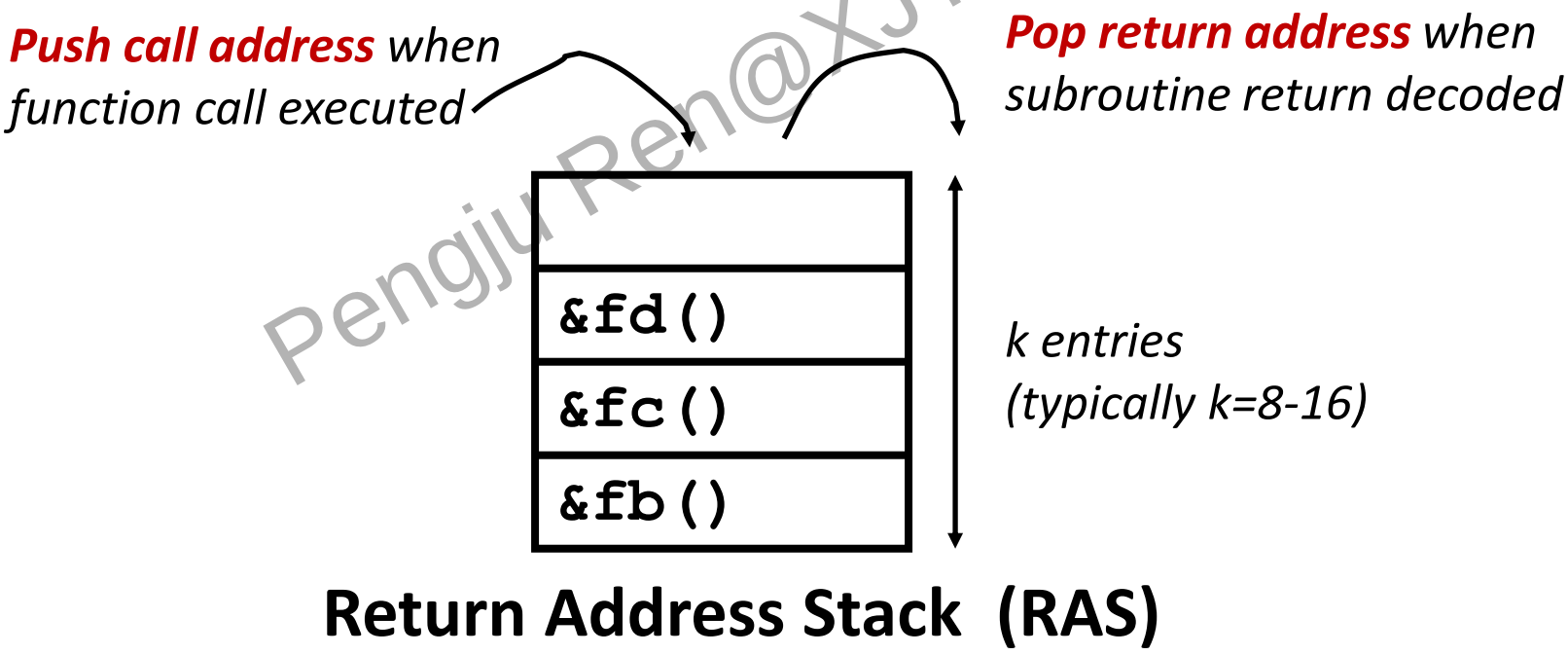
- CALL for some shared subroutine
- The Target address of call is **fixed**
- **Can be solved using BTB**

- Return from callee to **different callers**
- The target(return) address is **unfixed**
- **Can't be solved using BTB**

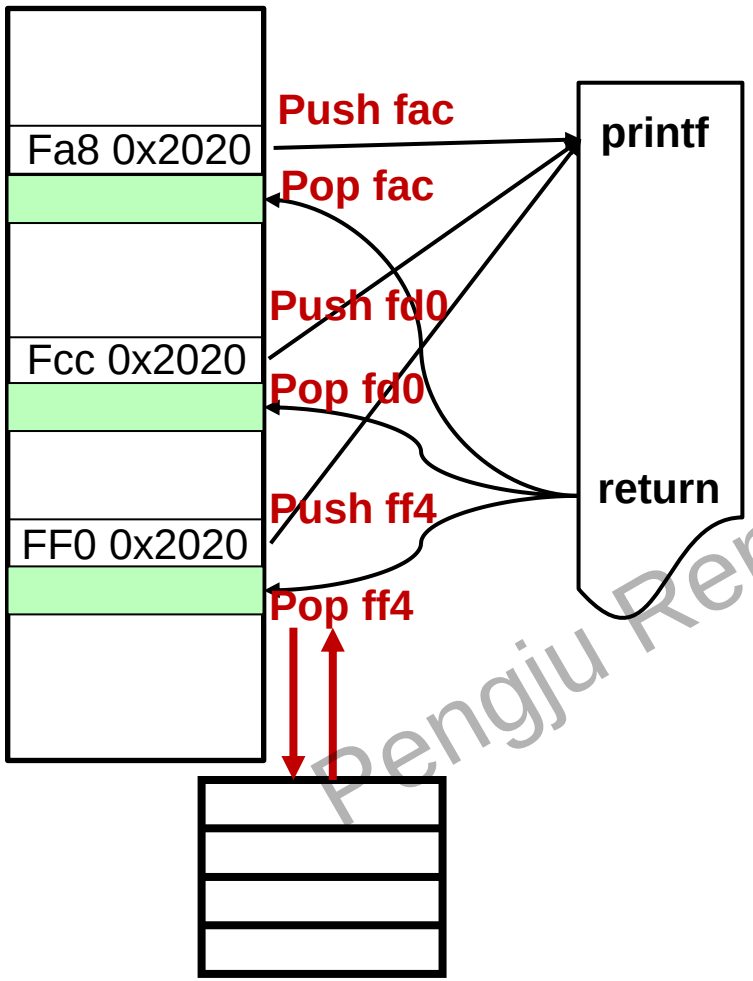
Subroutine Return Address Stack (RAS)

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

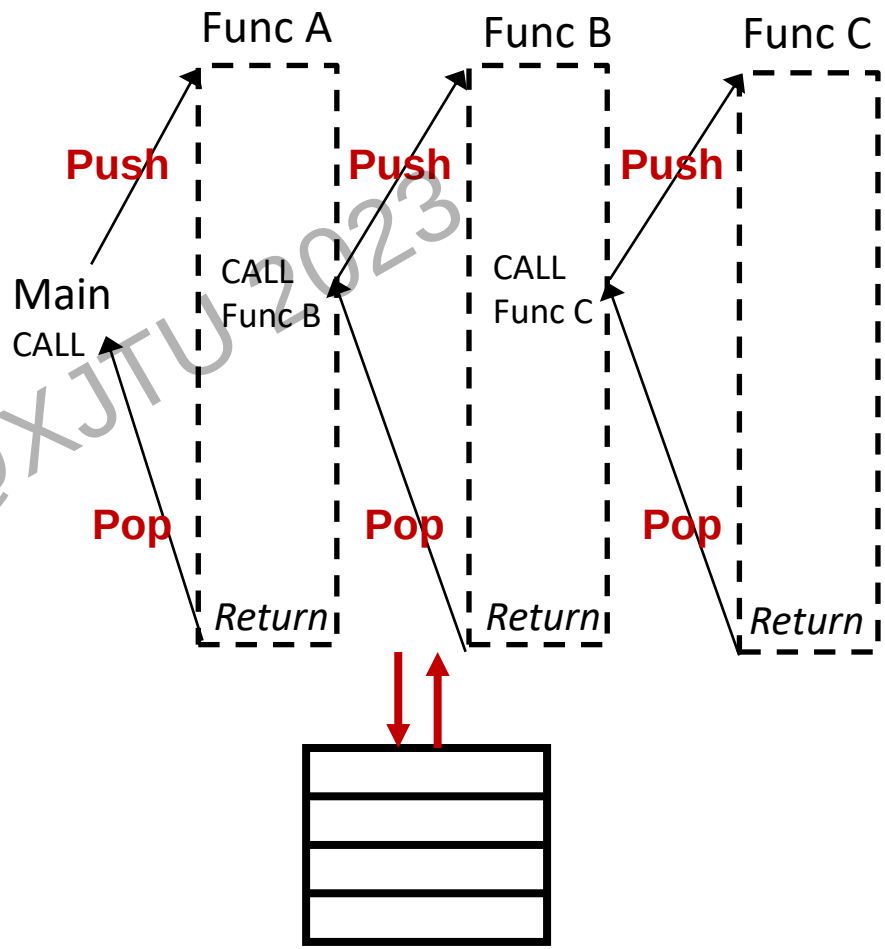
```
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```



Absolute Branch Prediction with RAS



Return Address Stack (RAS)



Return Address Stack (RAS)

Decode Call and Return Instructions

- Subroutine Call and Return instructions are usually **pseudo instructions**
- Automatically translated into **fixed-format real instructions** by **compiler**

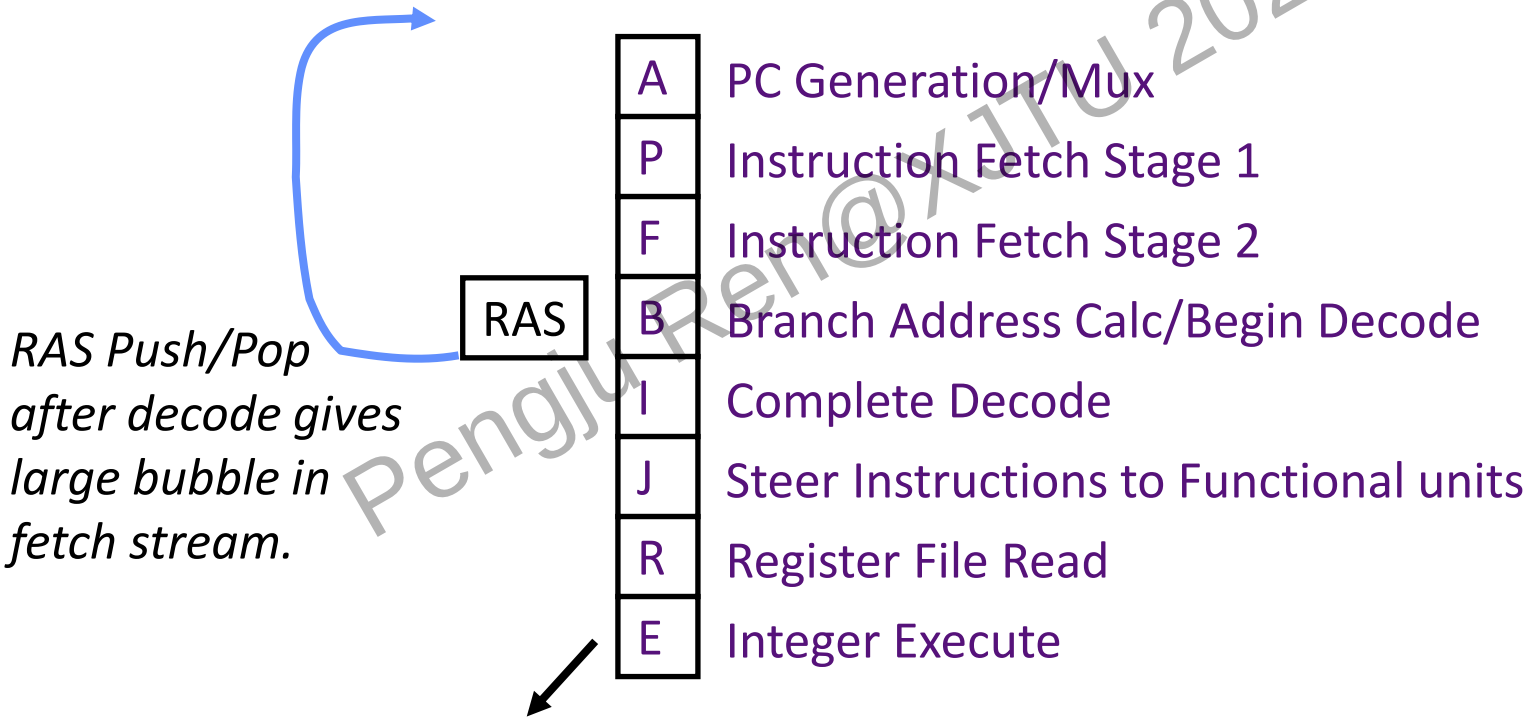
RISC-V Pseudo Call and Return

Pseudo Code	Instructions	Description
CALL	auipc x1,offset[31:12] jalr x0,x1,offset[11:0]	Call far-away subroutine
RET	jalr x0, x1, 0	Return from subroutine call

- **Decoder** recognizes the **fixed encoding** to find subroutine **Call and Return branch** instructions.

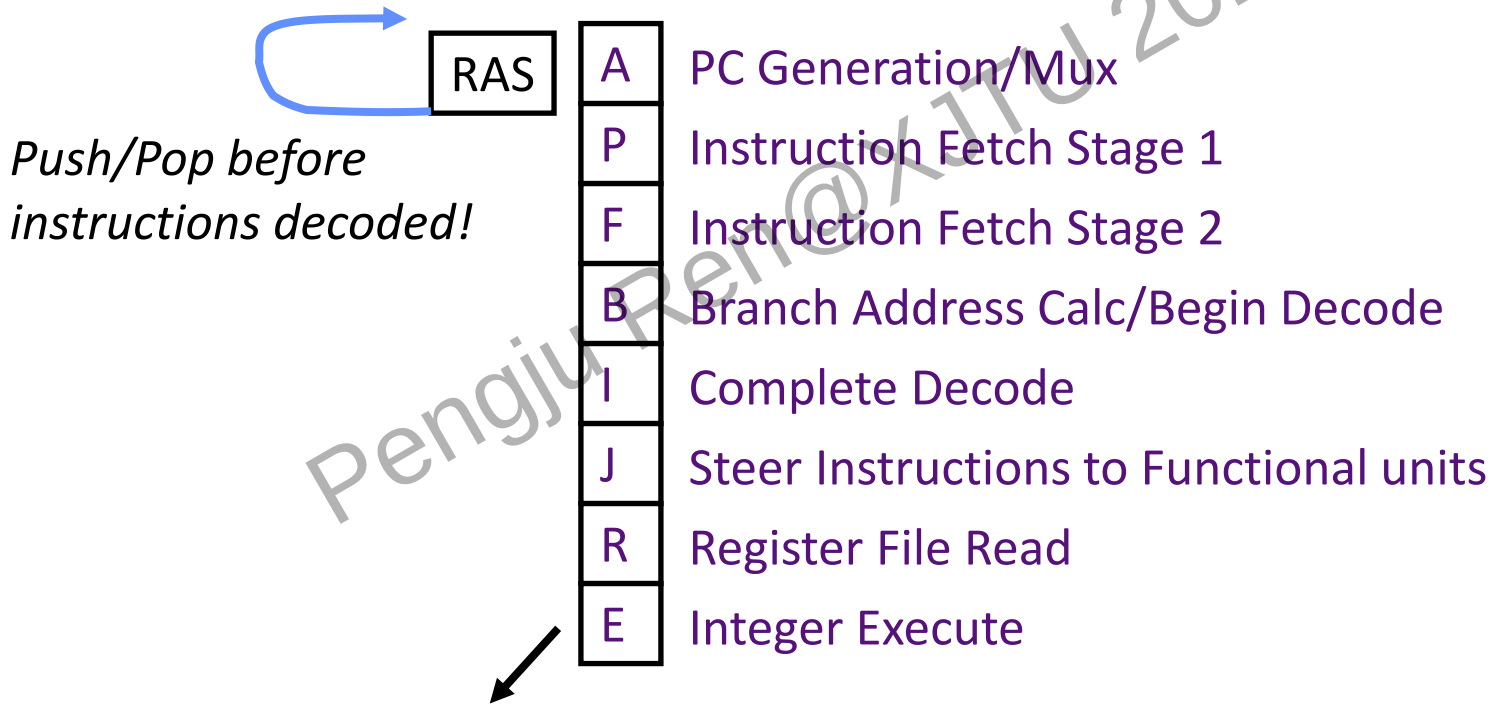
Return Address Stack in Pipeline

- How to use return address stack (RAS) in deep fetch pipeline?
- Only know if subroutine call/return at **decode stage**

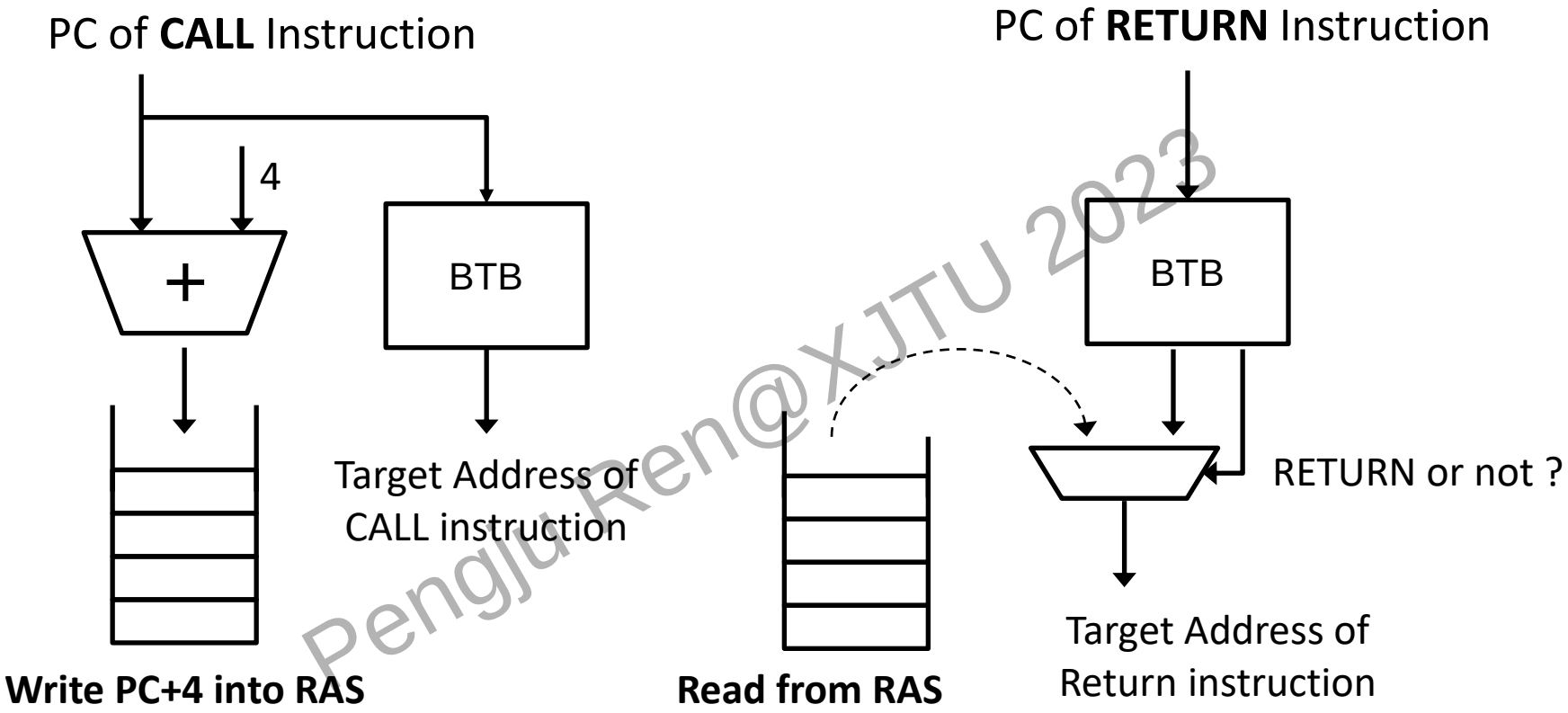


Return Address Stack in Pipeline

- Can remember **whether PC is call/return** in extended **BTB** structure
- Instead of target-PC, **just store push/pop bit**



Branch Prediction for CALL/Return



If RAS is **full**, drop the new one? Or the oldest?

- Drop the oldest in a **First-in-First-out** fashion (FIFO)
- Use it in a **Last-in-First-out** fashion (Stack)

Absolute Address Jump/Branch

(Switch case, function pointer array, …)

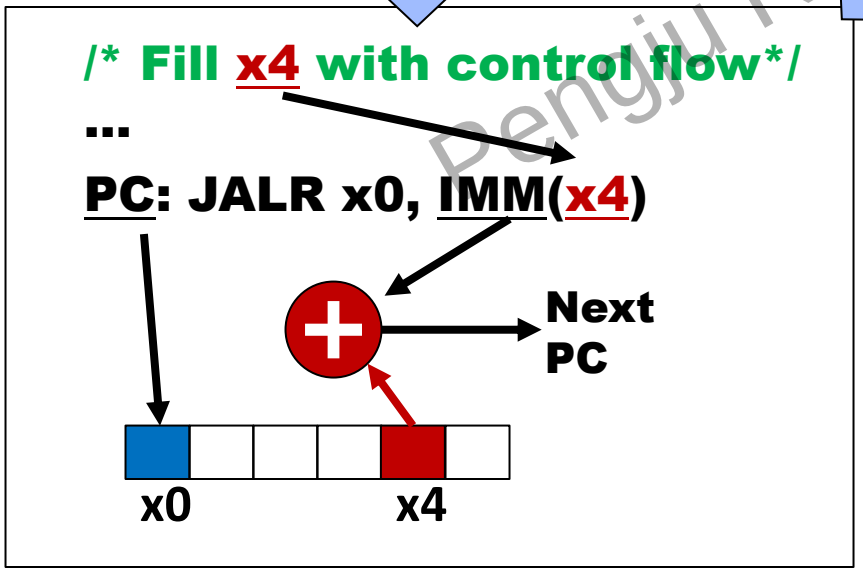
Switch Case

```
Case (cond)
  case 1: Branch Target 1
  case 2: Branch Target 2
  case 3: Branch Target 3
  case 4: Branch Target 4
  …
  Default:Branch Target T
```

Function Pointer Array

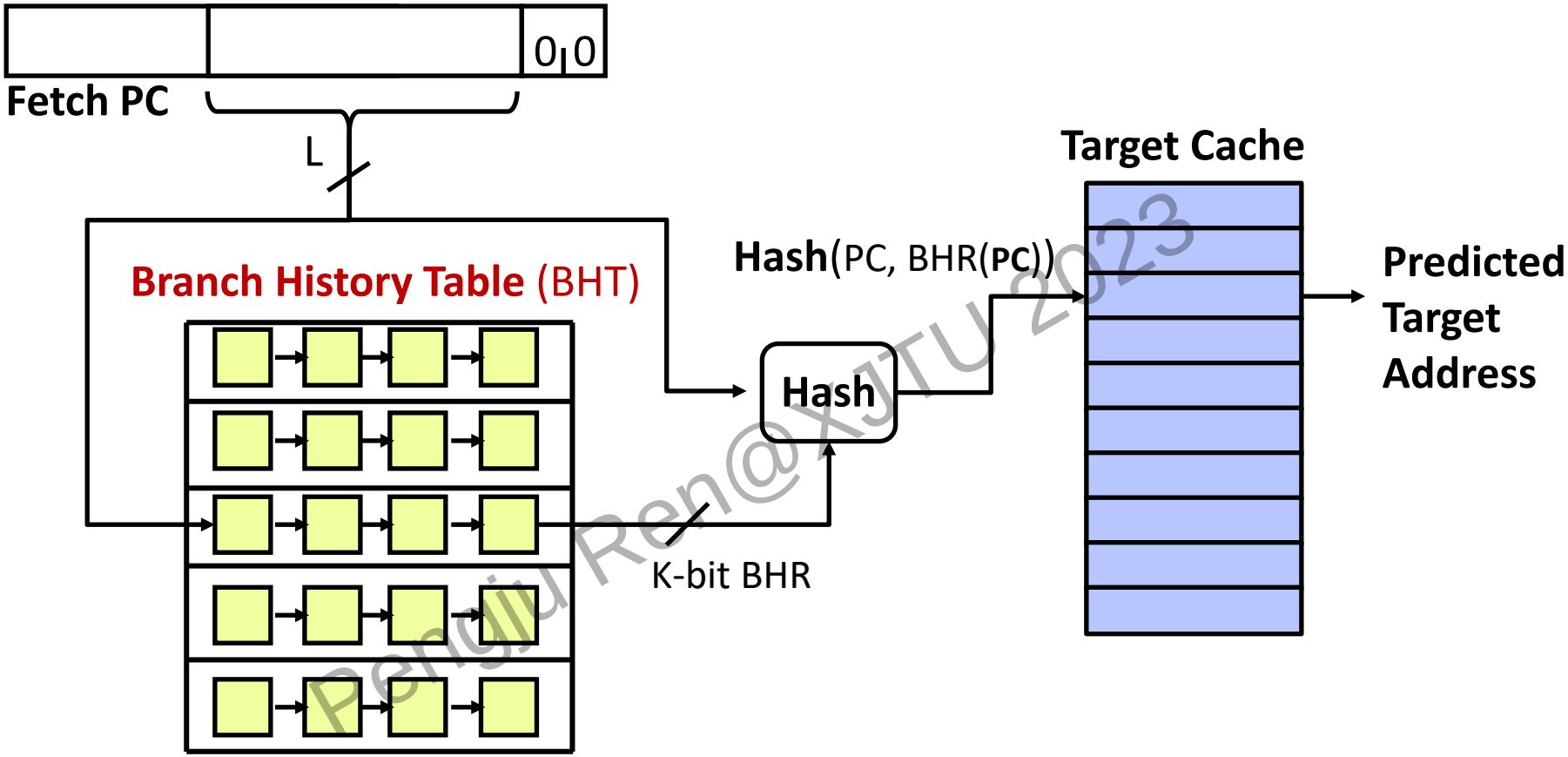
```
int x = 20
int y = 50

int (*arr[4])(int, int);
arr = {&add,&sub,&mul,&div};
for (i=0; i<4; i++)
  printf(arr[i](x, y));
```



- Jump address is dynamically generated
- The **condition of control flow** influences target address
- **Branch history** might help!

Absolute Branch Prediction with Target Cache



- Use **local branch history** to help predict target address
- Use **hash** to combine branch PC and branch history
- Like **BHT**, but stores **target address** instead of direction

Temporal and Spatial Correlation

```
if (x[i] < 7) then
    y += 1;
if (x[i] < 5) then
    c -= 4;
```

If *first condition* false, *second condition* also false.

How about the *first condition* is true, does this help make the prediction of the *second "if"* statement ?

No!

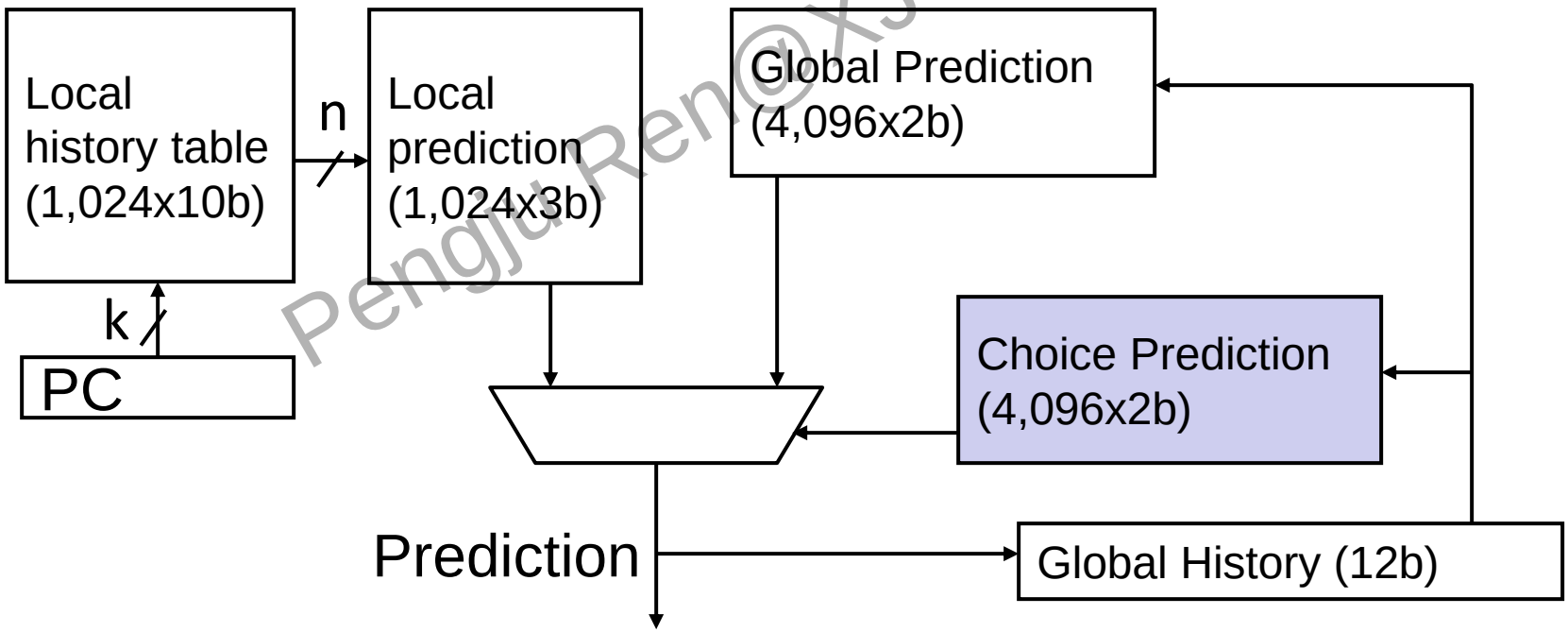
Prediction must be a function of **own branch** as well as recent outcomes of **other branches**.

History register records the direction of the last N branches executed by the processor

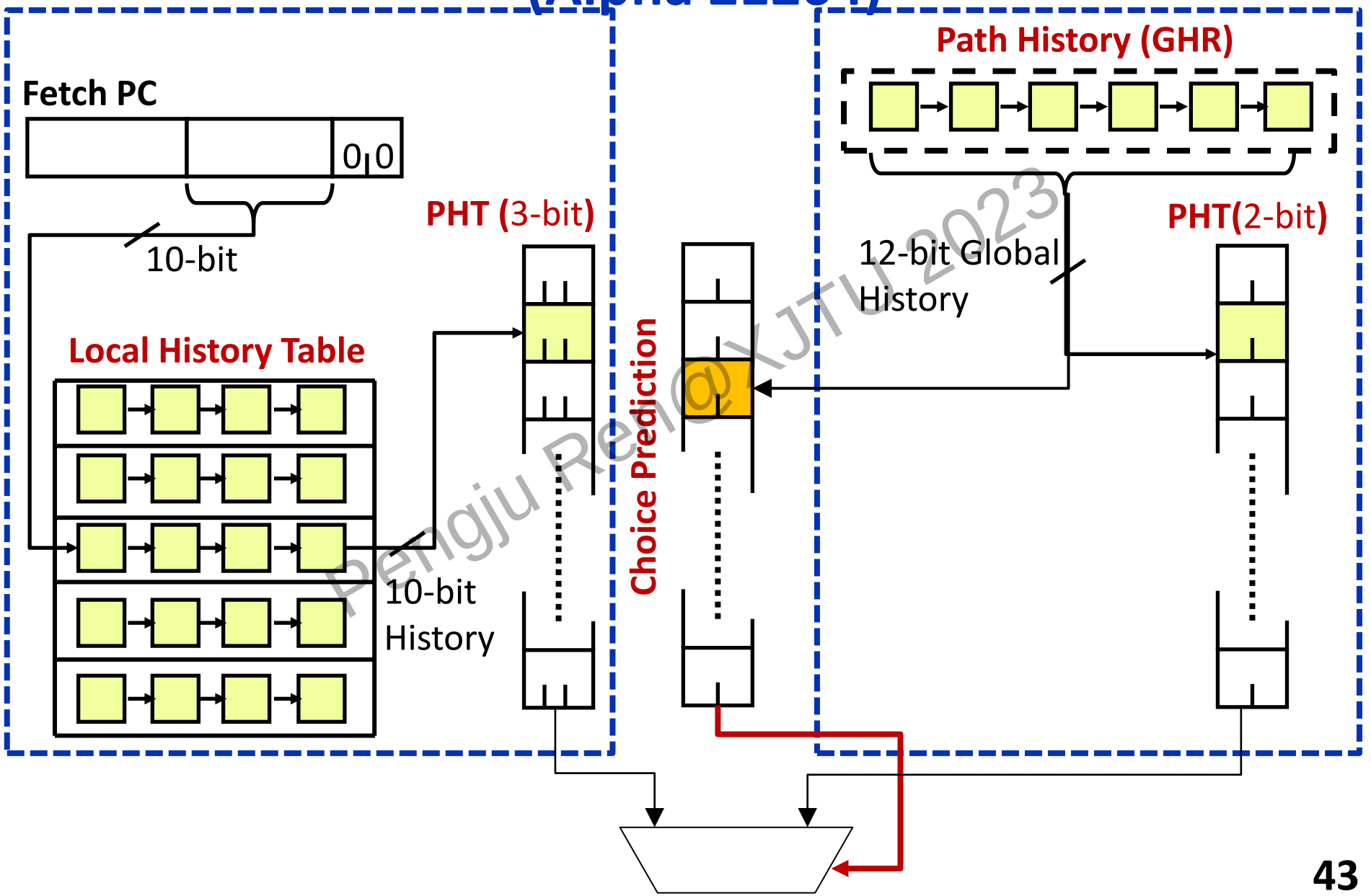
- *Local branch history*
- *Global branch history*

Tournament Branch Predictor (Alpha 21264)

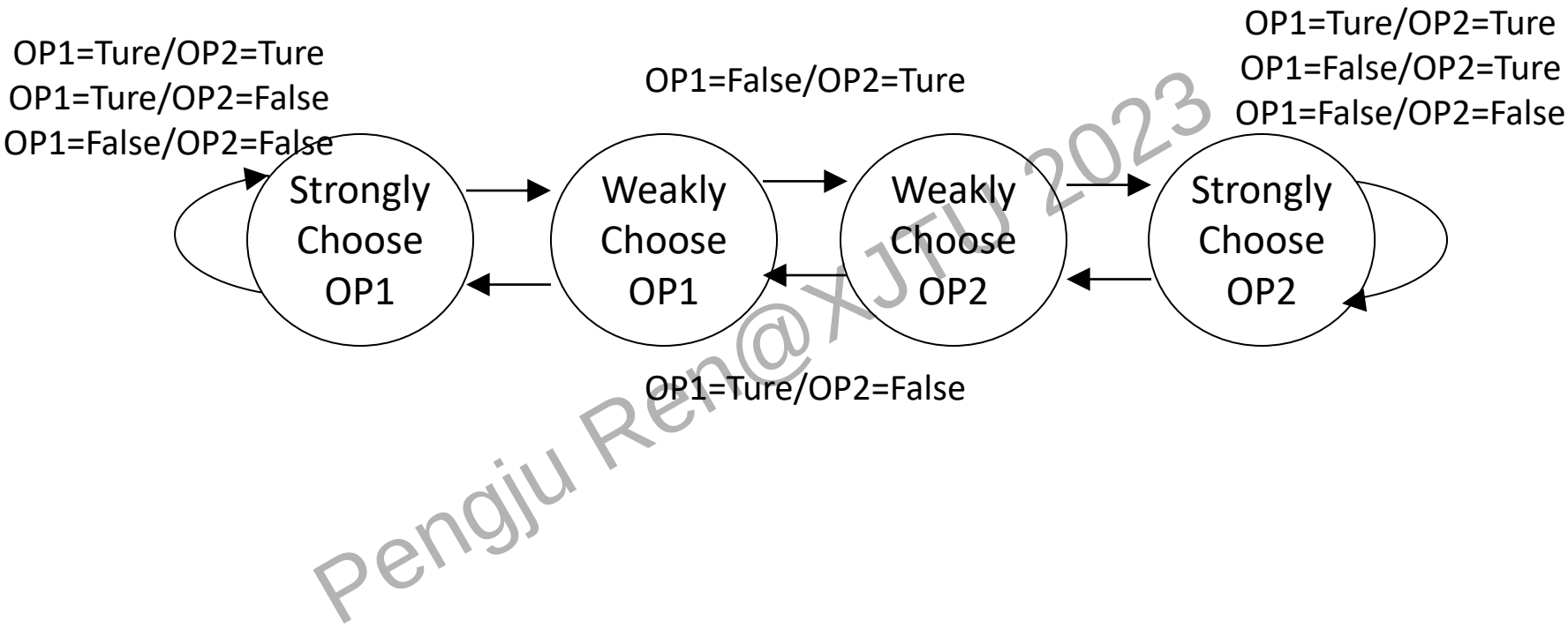
- **Choice predictor** learns whether best to use **local** or **global** branch history in predicting next branch (Different schemes work better for different branches)
- Global history is speculatively updated but restored on mispredict
- Claim **90-100%** success on range of applications



Tournament Branch Predictor (Alpha 21264)

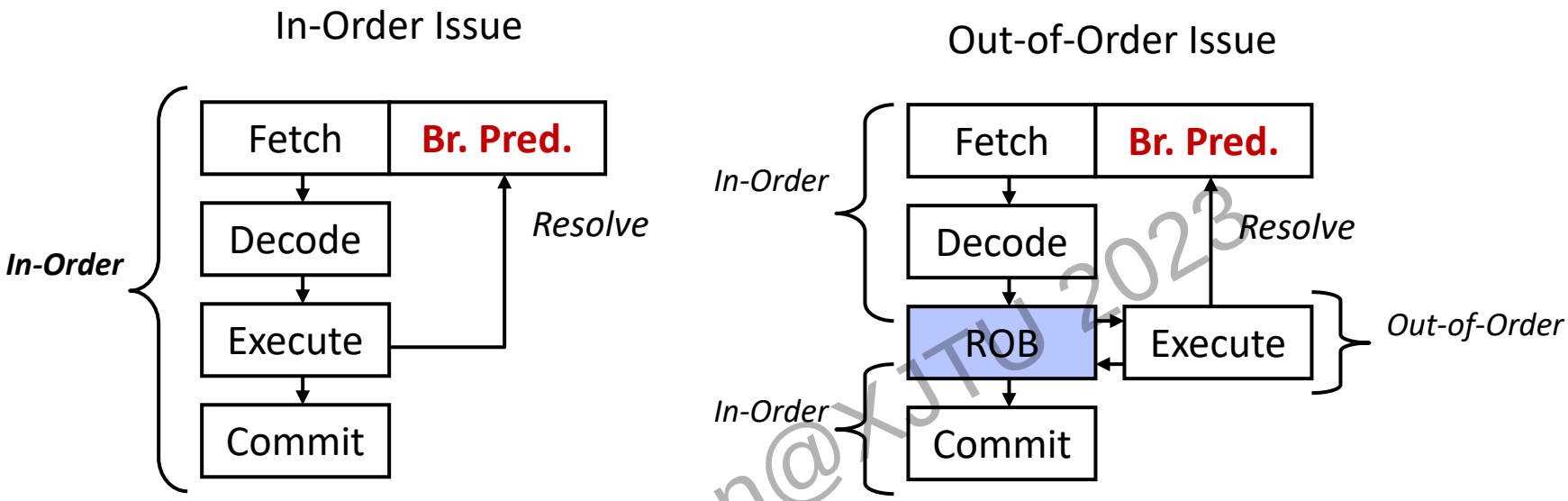


2-Bit Choice Prediction



Tournament Predictors – Combine approaches (local and global)

In-Order vs. Out-of-Order Branch Prediction



- **Speculative fetch but not speculative execution** - branch resolves before later instructions complete
- **Completed values** held in **bypass network** until commit

- **Speculative execution**, with branches resolved after later instructions complete
- **Completed values** held **in rename registers in ROB** or unified physical register file until commit

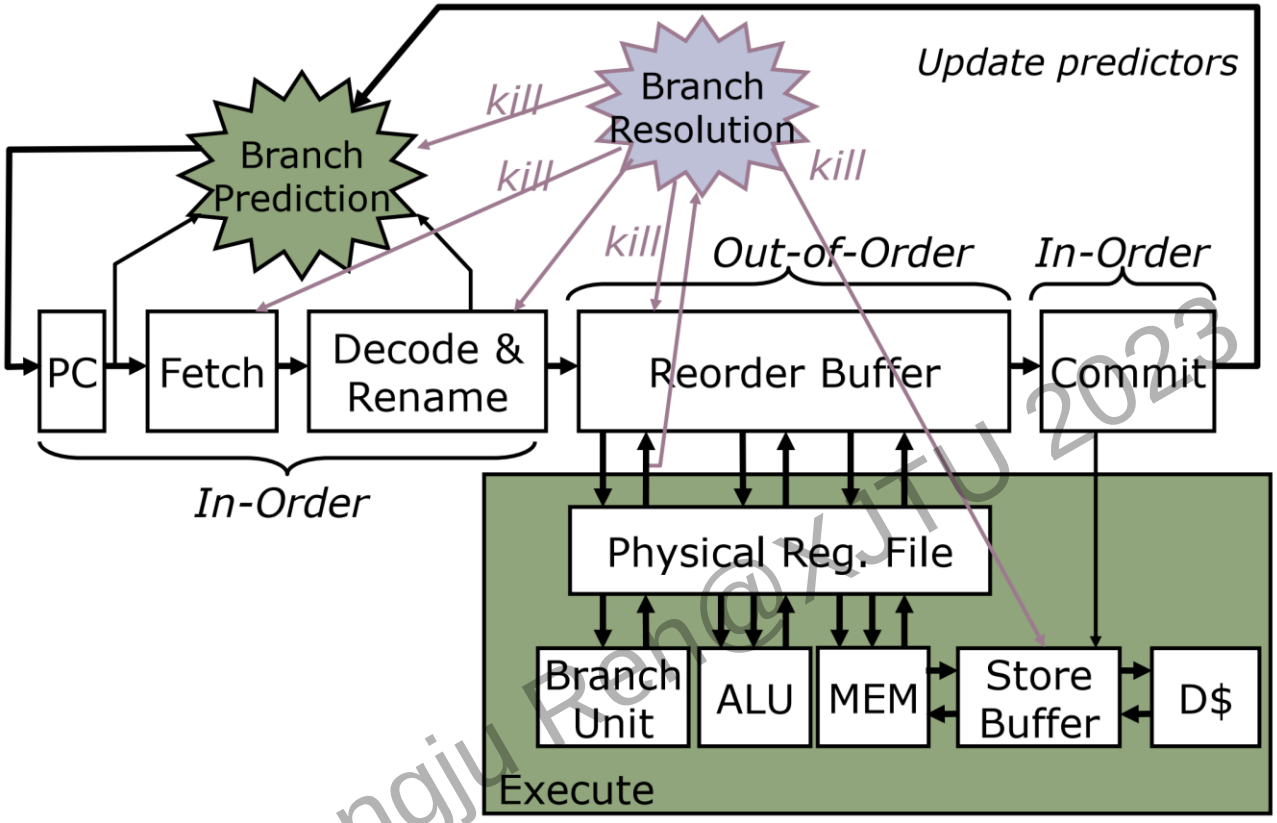
Both styles of machine can use **same branch predictors** in front-end fetch pipeline, and both can execute multiple instructions per cycle

Common to have **10-30 pipeline stages** in either style of design

InO vs. OoO Mispredict Recovery

- In-order execution?
  - Design so **no instruction issued after branch** can write-back before branch resolves
    - **Kill all instructions in pipeline** behind mispredicted branch
- Out-of-order execution?
  - Multiple instructions following branch in program order can **complete before branch** resolves
  - A simple solution would be to handle like **precise exception**
    - **Kill following instructions** in pipeline and ROB
    - **Restore rename register** status to the branch time point
  - How about **multiple branches** are encountered?

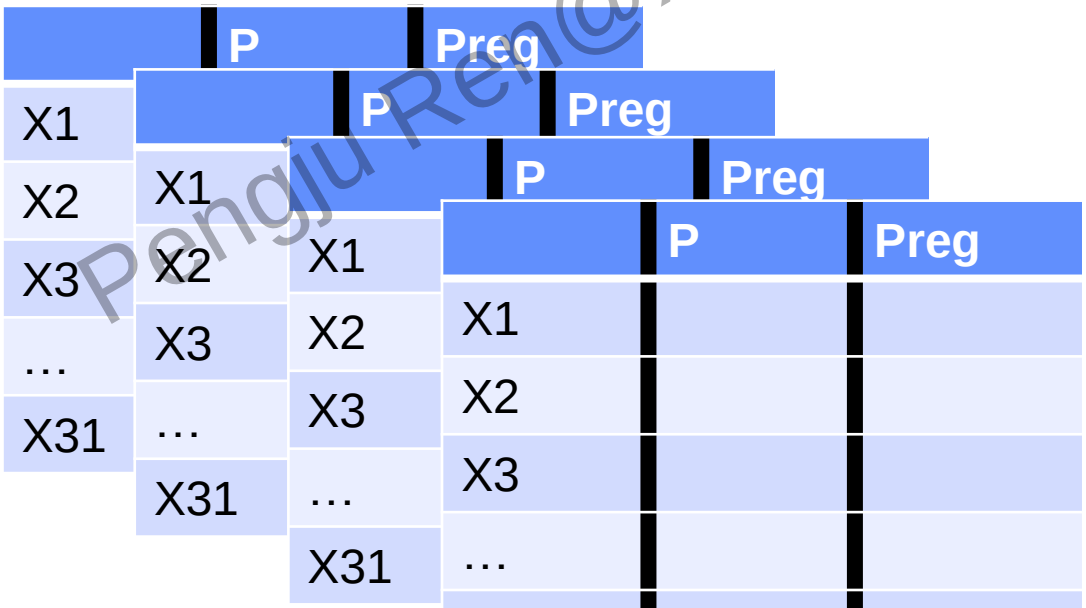
Branch Misprediction in OoO Pipeline



- Can have **multiple unresolved branches** in ROB
- Can resolve branches out-of-order by **killing all the instructions** in ROB that follow a mispredicted branch
- Use **mask bits** to tag instructions that are dependent on **different speculative branches**
- Mask bits cleared as branch resolves, and **reused** for next branch

Rename Table Recovery

- Have to quickly recover **rename table** on branch mispredictions
- MIPS R10K only has **four snapshots** for each of four outstanding speculative branches
- Alpha 21264 has **80 snapshots**, one per ROB instruction

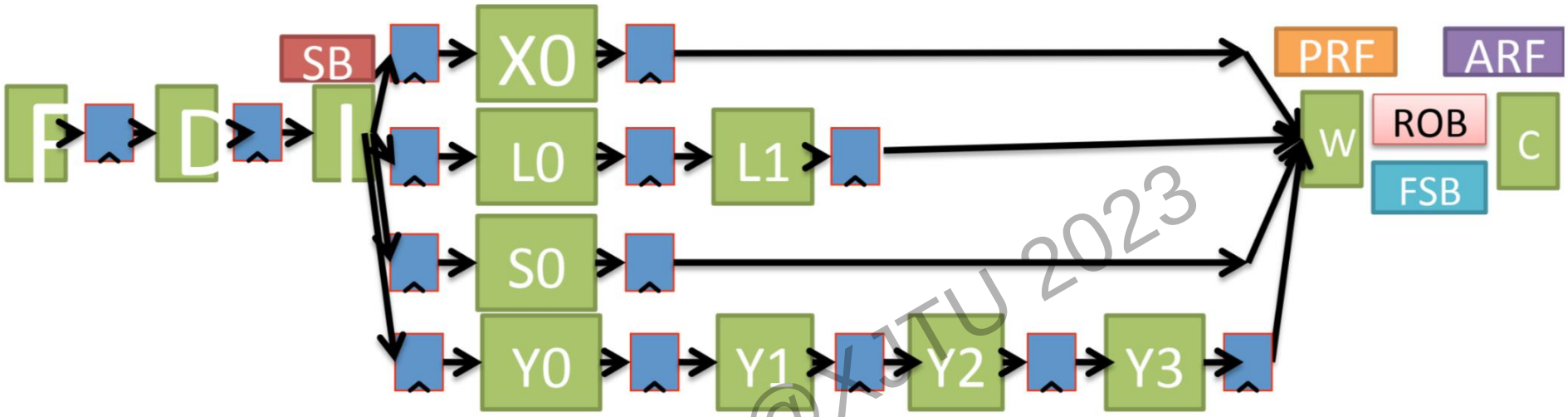


Snapshots of RT for mispredict recovery

Load-Store Queue Design

- After control hazards, **data hazards** through memory are probably next most important bottleneck to superscalar performance
- Modern superscalars use very sophisticated load-store reordering techniques to reduce effective memory latency by allowing **loads and stores to be speculatively issued**
  - Speculative load → ROB and physical register (rename)
  - Speculative store → Store Buffer

Recap: I2OI (OoO superscalar)

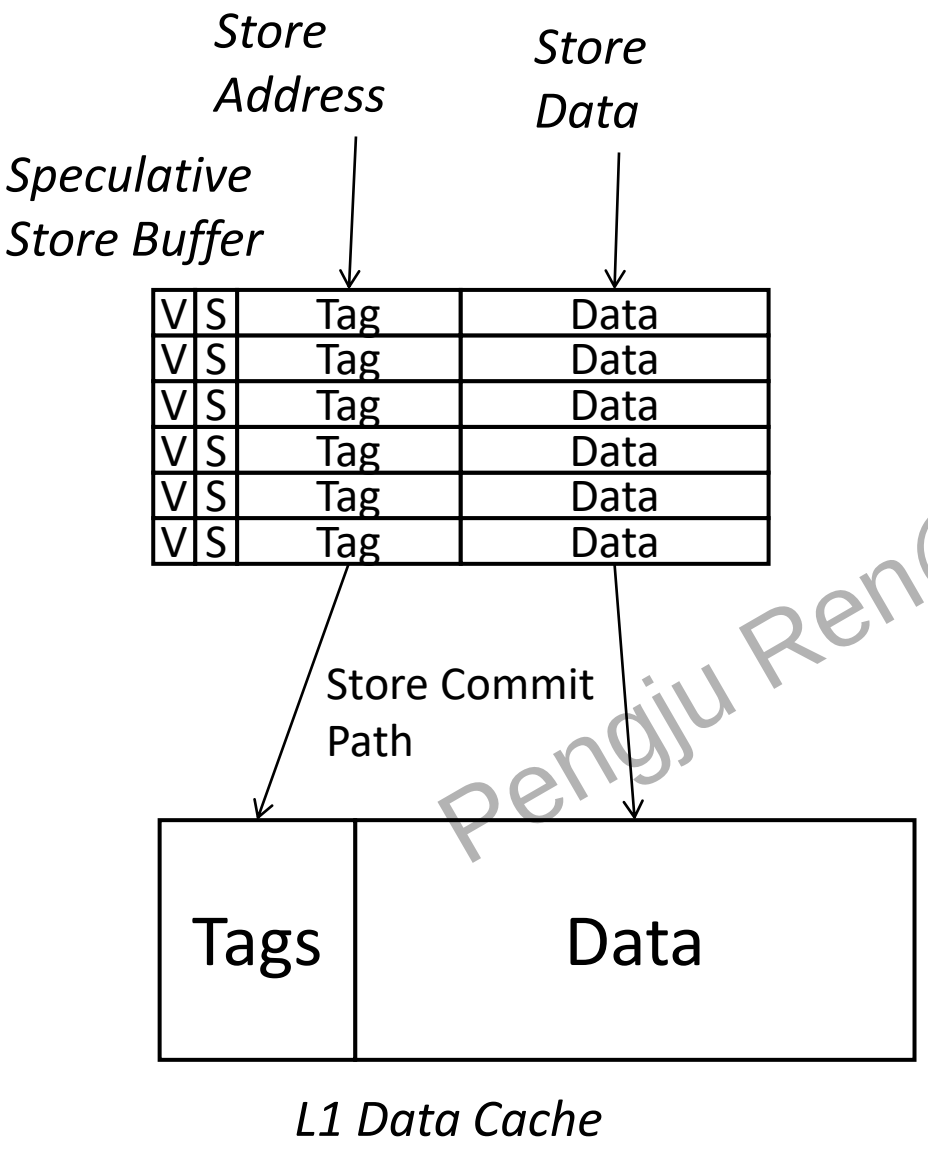


ARF				W
SB	R/W		W	
PRF	R		W	
ROB	R/W		W	R/W
FSB			W	R/W

PRF = Physical Register File (Future File)

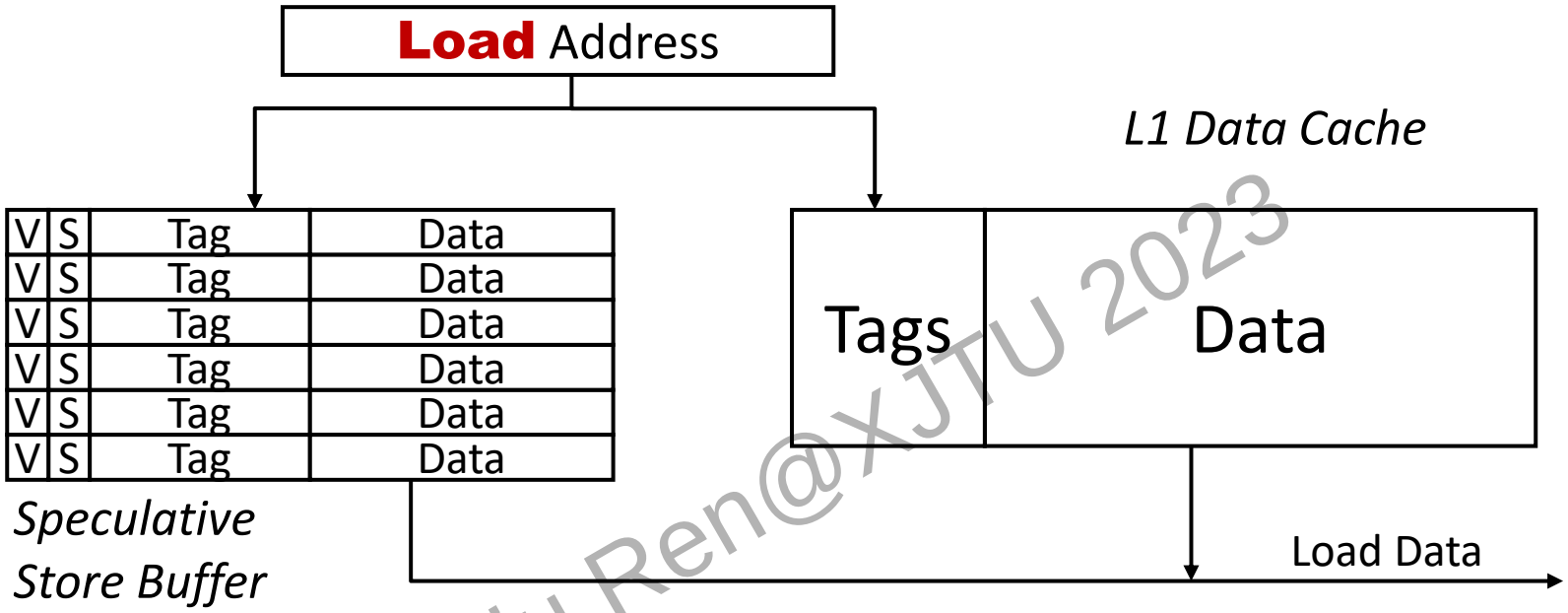
ROB=Reorder Buffer, FSB=Finished Store Buffer (1 Entry)

Speculative Store Buffer



- Just like register updates, stores should not modify the memory until after the instruction is committed. A **speculative store buffer** is a structure introduced to hold speculative store data.
- During decode, store buffer slot allocated in program order
- Stores split into "store address" and "store data" micro-operations
- "Store address" execution writes tag
- "Store data" execution writes data
- **Store commits** when oldest instruction and both address and data available:
  - clear speculative bit and eventually **move data to cache**
  - **On store abort:,** clear valid bit

Load bypass from speculative store buffer



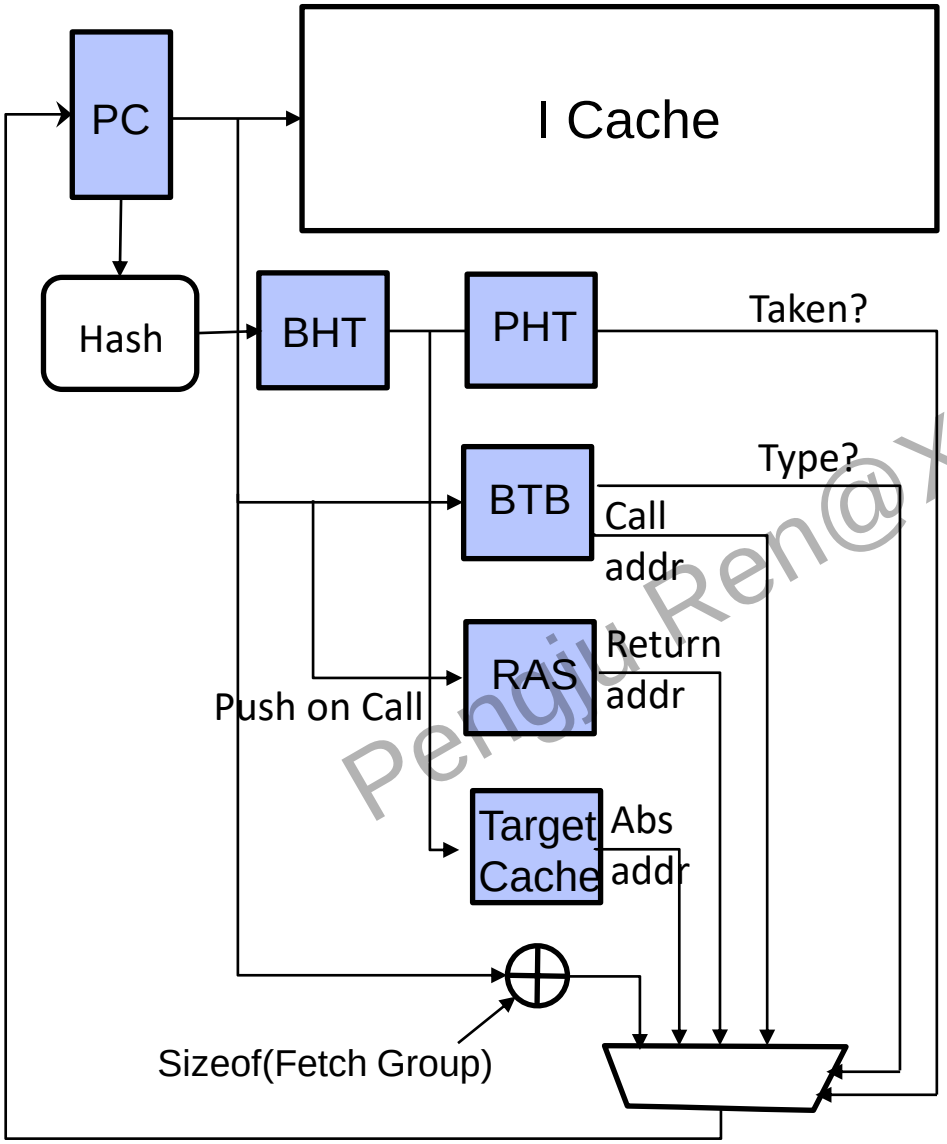
- If data in both store buffer and cache, which should we use?

  Speculative store buffer

- If same address in store buffer twice, which should we use?

  Youngest store older than load

Summary: a Full Branch Prediction method



- BHT+PHT:
  - Branch direction
  - Conditional branch
- BTB:
  - Branch Target Address
  - Absolute branch and PC-Relative branch
  - Call of functions
- RAS:
  - Branch Target Address
  - Absolute branch
  - Return of calls
- Target Cache:
  - Branch Target Address
  - Absolute branch
  - Dynamic long jumps

Next Lecture：Very Long Instruction Words

(Data Level Parallel)