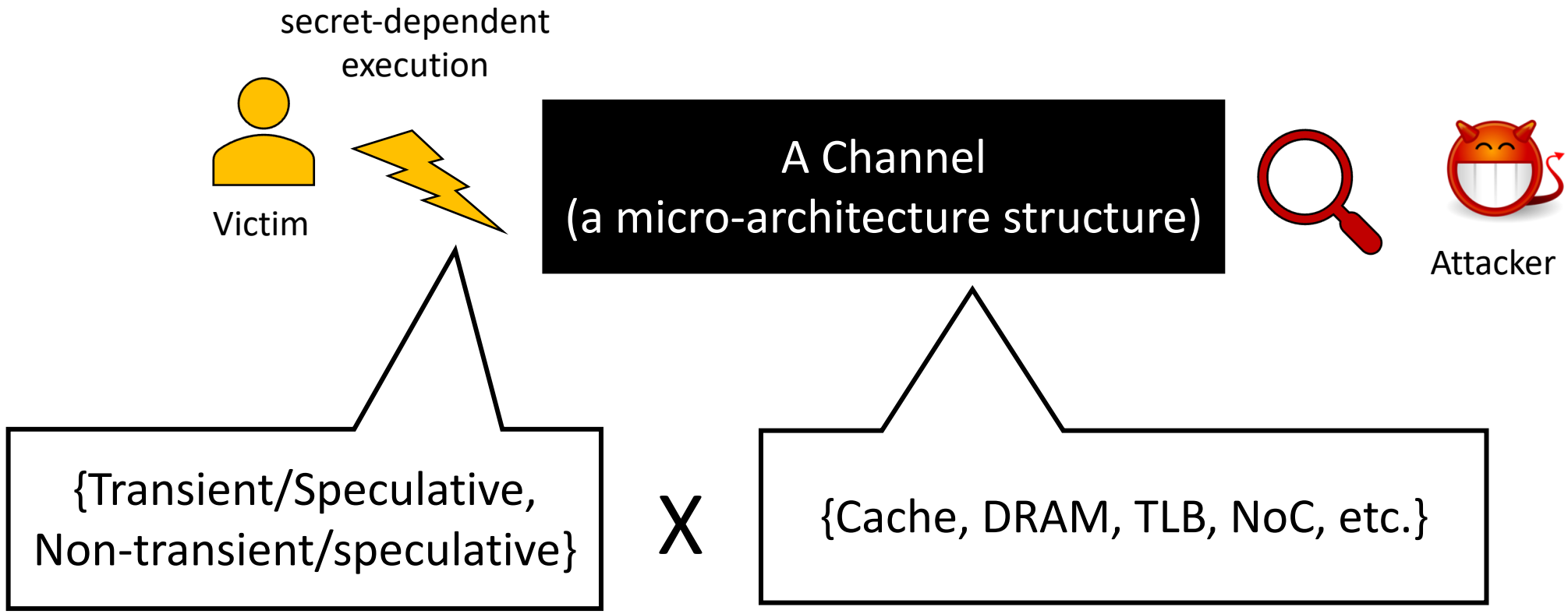
Transient Execution Attacks

Mengjia Yan

Spring 2022

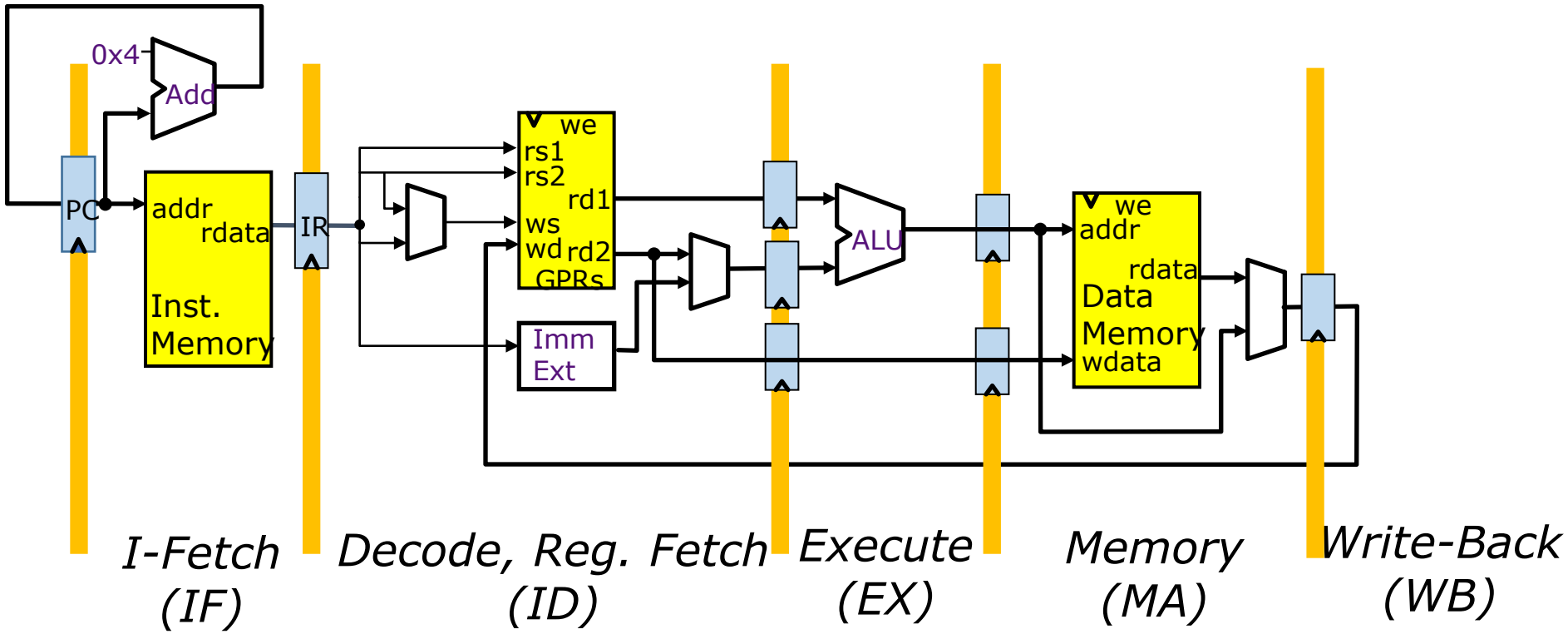
Based on slides from Christopher Fletcher

Micro-architecture Side Channels

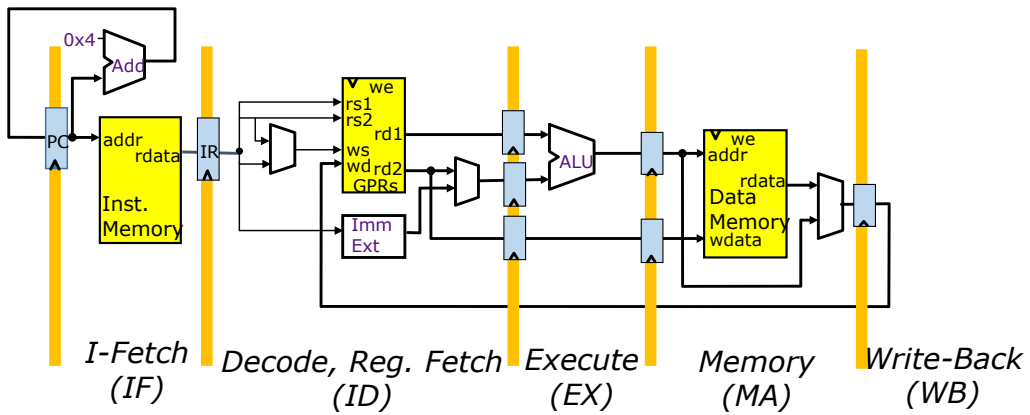


Kiriansky et al. DAWG: a defense against cache timing attacks in speculative execution processors. MICRO'18

Recap: 5-stage Pipeline



Recap: 5-stage Pipeline



- In-order execution:
  - Execute instructions according to the program order
  - What is the ideal instruction throughput? -- instruction per cycle (IPC)

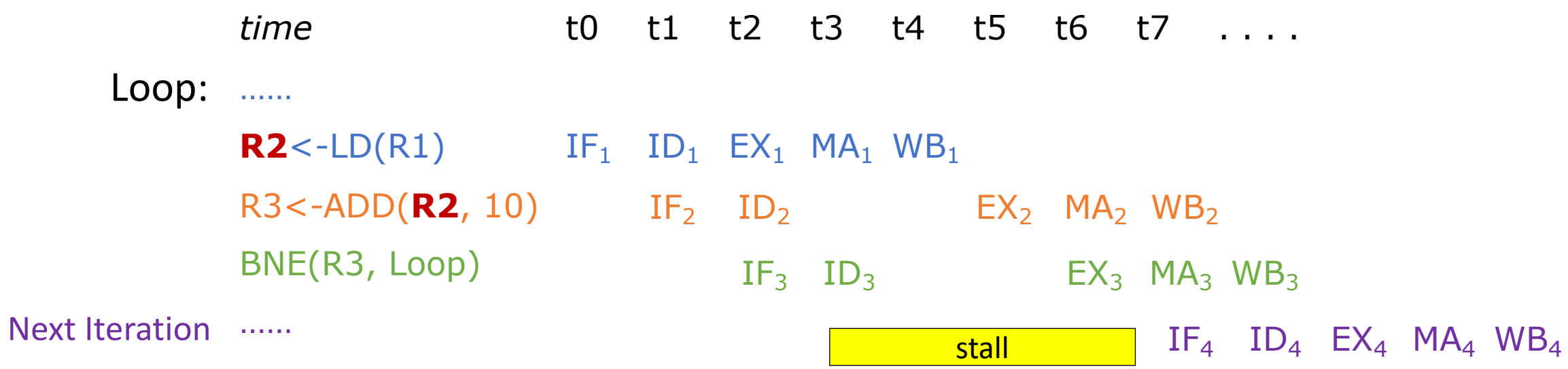
time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
instruction1	IF_1	ID_1	EX_1	MA_1	WB_1				
instruction2		IF_2	ID_2	EX_2	MA_2	WB_2			
instruction3			IF_3	ID_3	EX_3	MA_3	WB_3		
instruction4				IF_4	ID_4	EX_4	MA_4	WB_4	
instruction5					IF_5	ID_5	EX_5	MA_5	WB_5

Data Hazard and Control Hazard

- Approaches to resolve Hazards:
  1. Stall
  2. Bypass
  3. Speculation

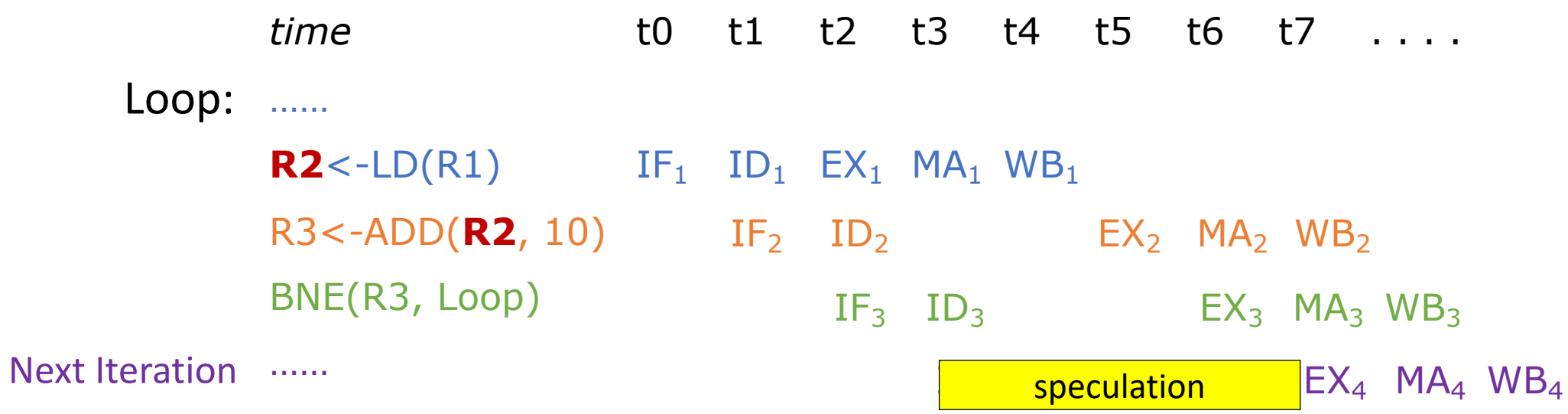
Data Hazard and Control Hazard

- Approaches to resolve Hazards:
  1. Stall
  2. Bypass
  3. Speculation



Data Hazard and Control Hazard

- Resolving Hazards:
  1. Stall
  2. Bypass
  3. Speculation



Question: What if speculation is correct? What if it is incorrect?

Branch Predictor

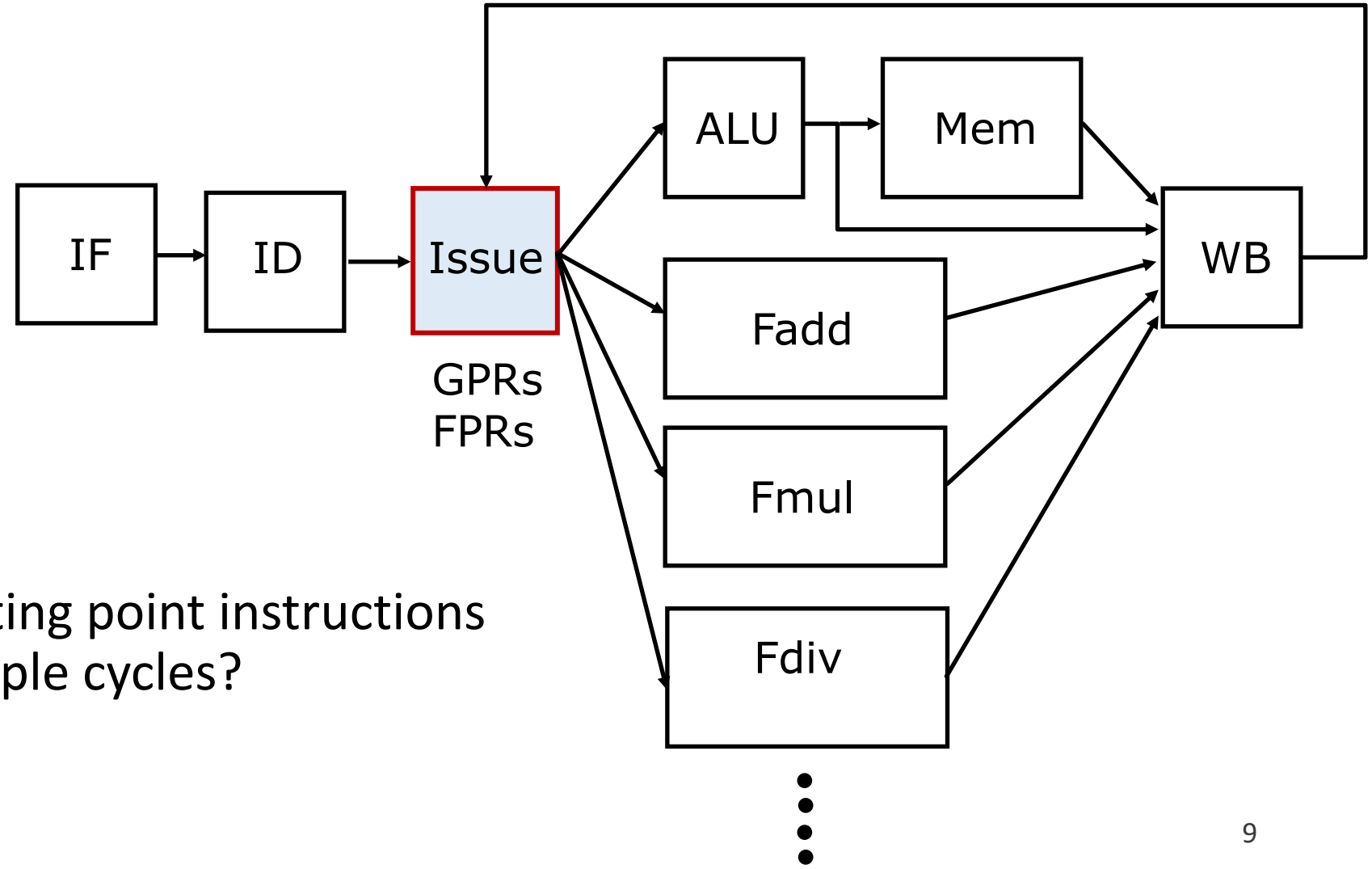
- Predict Taken/Not taken
  - Not taken: PC+4
  - Taken: need to know target address

How?

- **Predict** target address for indirect branch
  - Branch target buffer (BTB)
  - Map <current PC, target PC>

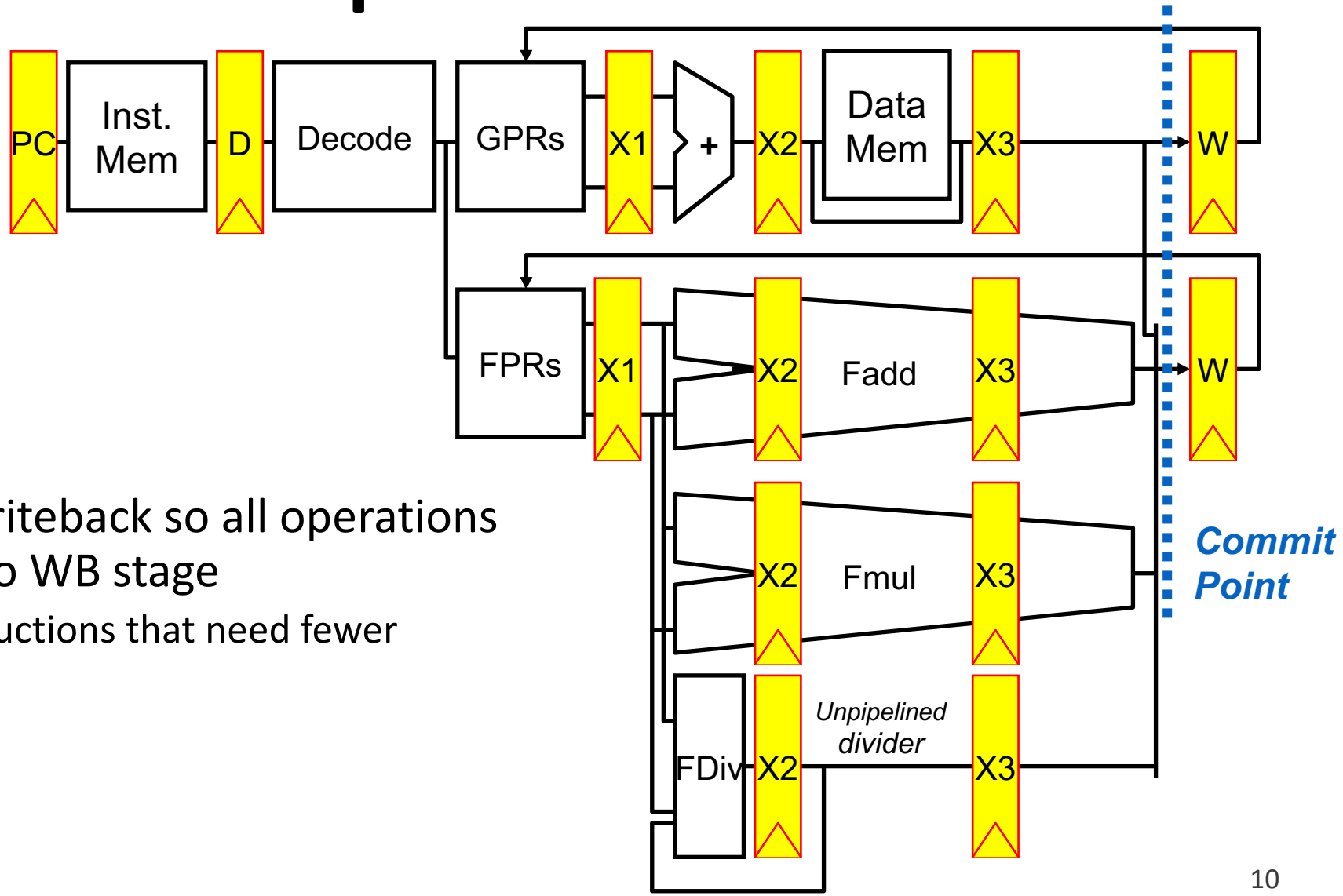
- Use history information to setup the predictor

Complex In-order Pipeline



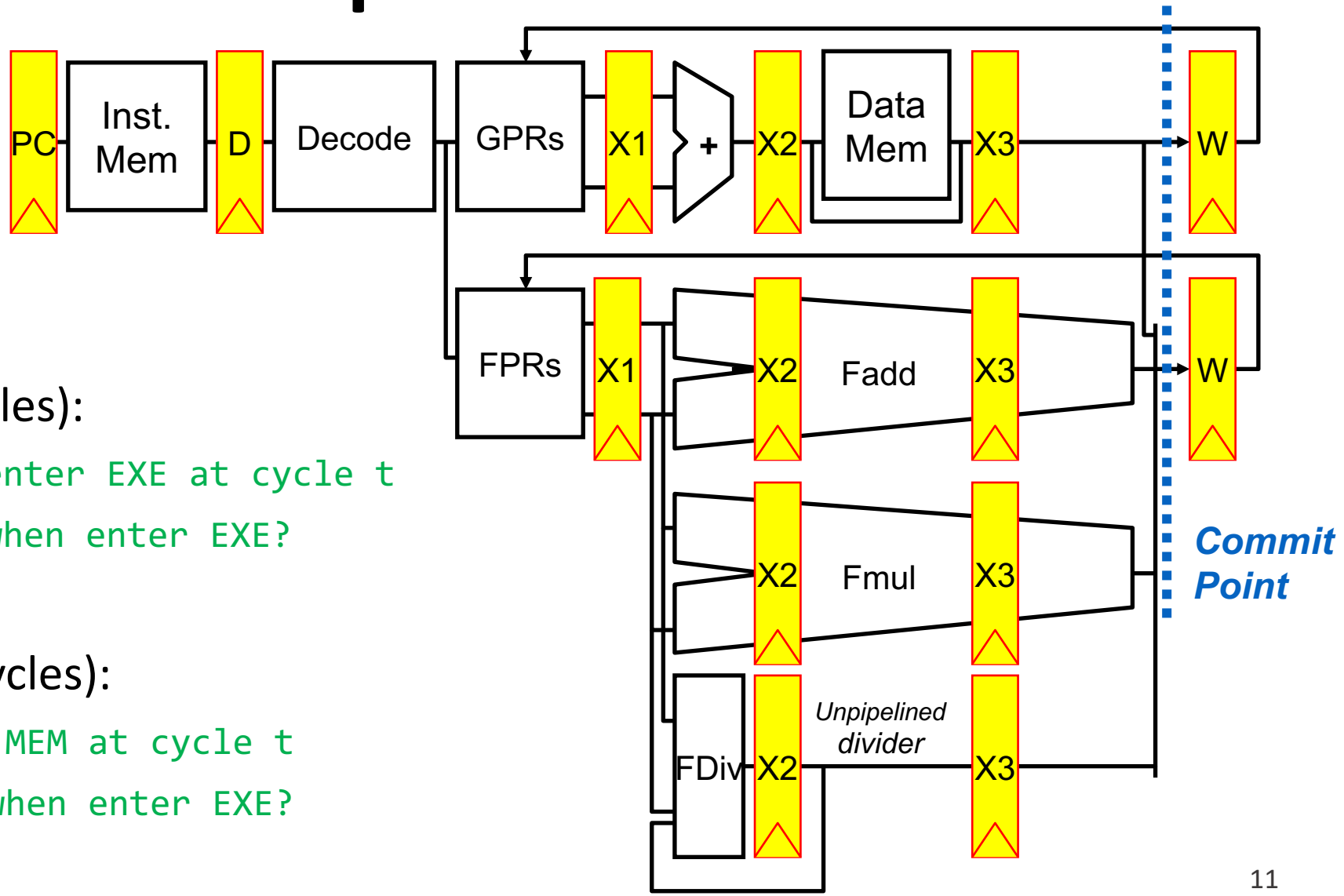
- How to support floating point instructions which can take multiple cycles?

Complex In-Order Pipeline



- Naïve idea: Delay writeback so all operations have same latency to WB stage
  - Slow: penalize instructions that need fewer pipeline stages

Complex In-Order Pipeline



Example 1 (Fadd 3 cycles):

```
F3 <- Add(F1, F2) // enter EXE at cycle t
R3 <- Add(R1, R2) // when enter EXE?
```

Example 2 (Ld 1~50 cycles):

```
R4 <- Ld(R5) // enter MEM at cycle t
F3 <- Add(F1, F2) // when enter EXE?
```

Out-of-order Completion

- The idea: Make use of idle functional units when pipeline is stalled

Any problems?

Example 1 (Fadd 3 cycles):

```
F3 <- Add(F1, F2) // enter EXE at cycle t
R3 <- Add(R1, R2) // when enter EXE?
```

Example 2 (Ld 1~50 cycles):

```
R4 <- Ld(R5) // enter MEM at cycle t
R3 <- Add(R1, R2) // when enter EXE?
```

Problem of Out-of-order Completion

Consider executing a sequence of

$$r_k \leftarrow (r_i) \ \text{op} \ (r_j)$$

type of instructions

Data-dependence

$r_3 \leftarrow (r_1)$ op (r_2) Read-after-Write

$r_5 \leftarrow (r_3)$ op (r_4) (RAW) hazard

Anti-dependence

$r_3 \leftarrow (r_1)$ op (r_2) Write-after-Read

$r_1 \leftarrow (r_4)$ op (r_5) (WAR) hazard

Output-dependence

$r_3 \leftarrow (r_1)$ op (r_2) Write-after-Write

$r_3 \leftarrow (r_6)$ op (r_7) (WAW) hazard

Scoreboard: Detect Hazards Dynamically

- Approach: Stall issue until sure that issuing will cause no dependence problems…

- What to check before the Issue stage can dispatch an instruction?
  - Is the required function unit available?
  - Is the input data available? $\Rightarrow$ RAW?
  - Is it safe to write the destination? $\Rightarrow$ WAR? WAW?
  - Is there a structural conflict at the WB stage?

A Data Structure for Correct In-order Issues

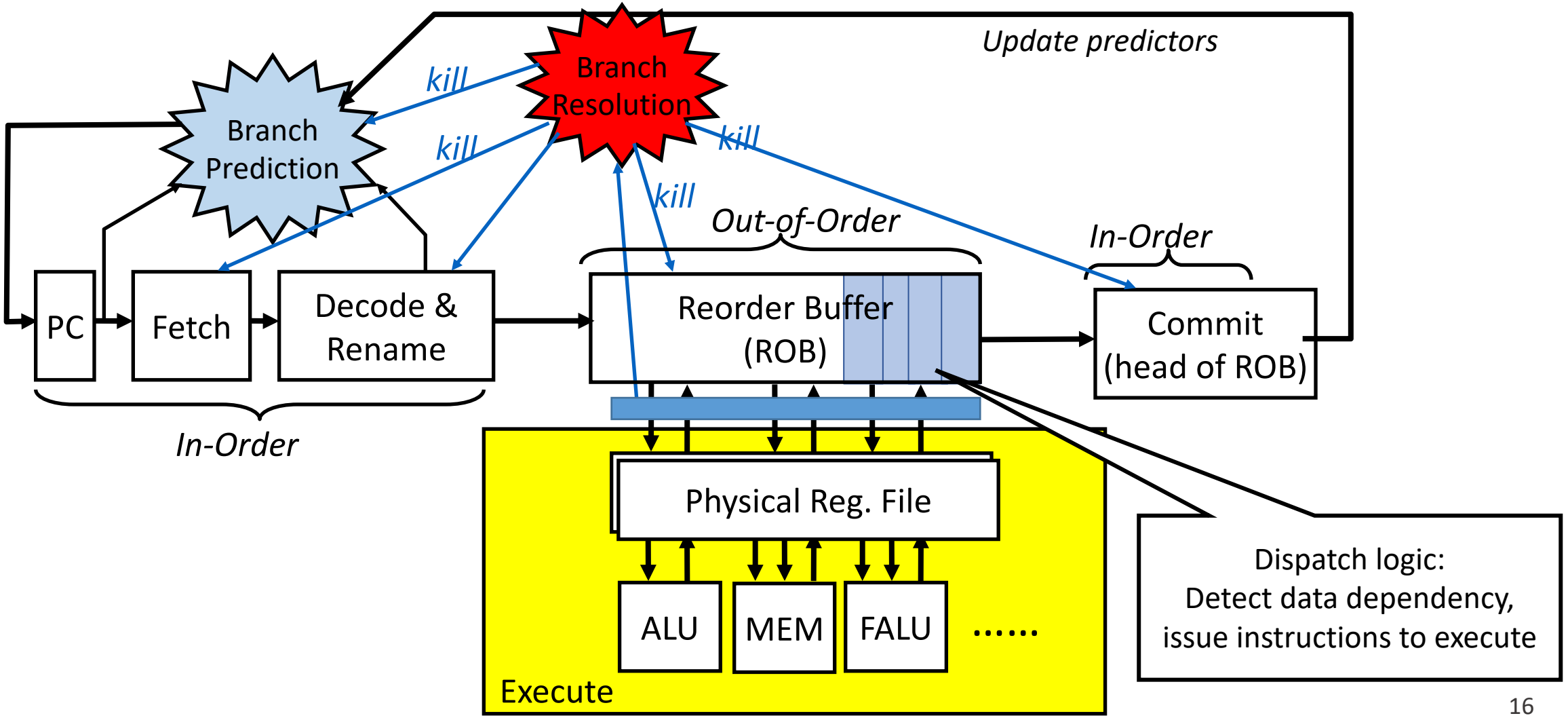
Name	*Busy*	*Op*	*Dest*	*Src1*	*Src2*
Int					
Mem					
Add1					
Add2					
Add3					
Mult1					
Mult2					
Div					

The instruction i at the Issue stage consults this table

FU available?	check the busy column
RAW?	search the dest column for i's sources
WAR?	search the source columns for i's destination
WAW?	search the dest column for i's destination

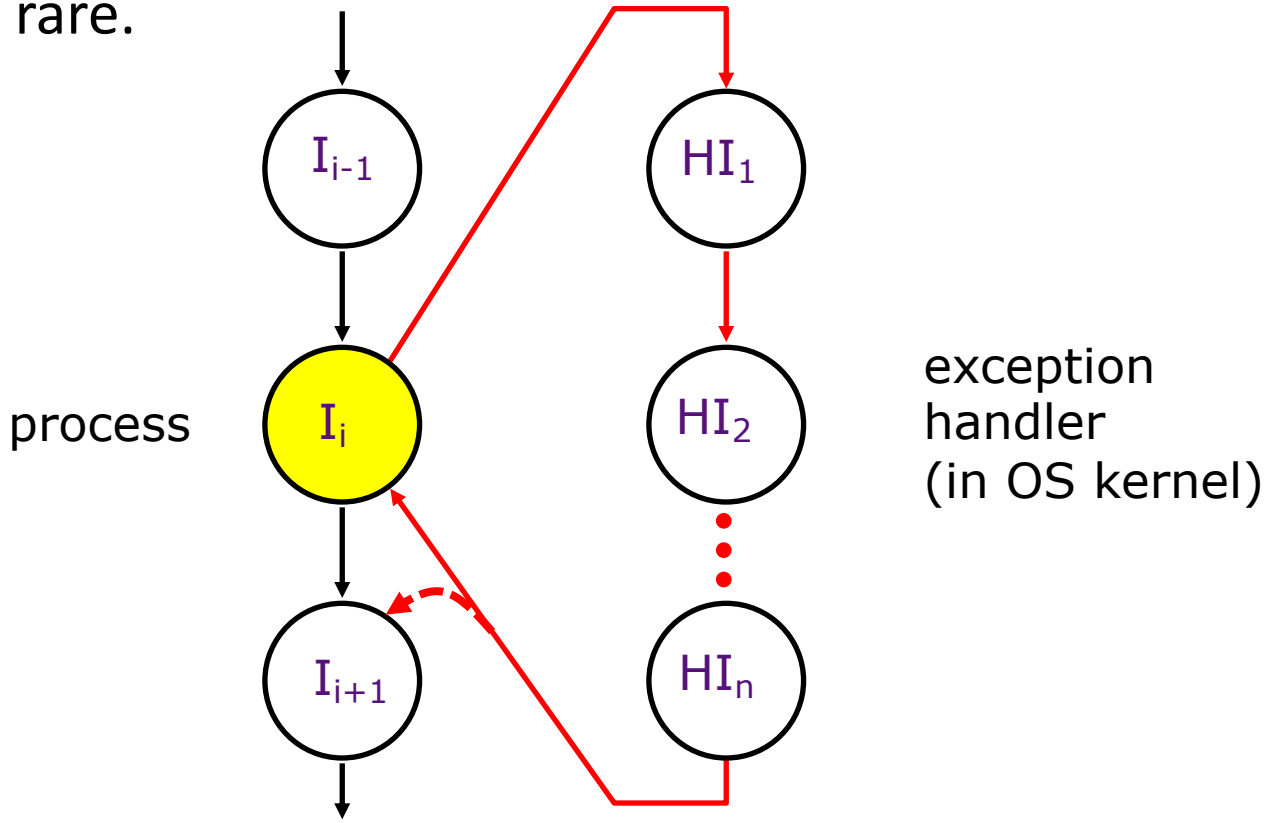
An entry is added to the table if no hazard is detected;
An entry is removed from the table after Write-Back

Superscalar Processors in 6.823



Precise Exception

- Exceptions: Event that needs to be processed by the OS kernel.
  - The event is usually unexpected or rare.
  - divide by zero, page fault, etc.



Handling Exceptions in OoO Processors

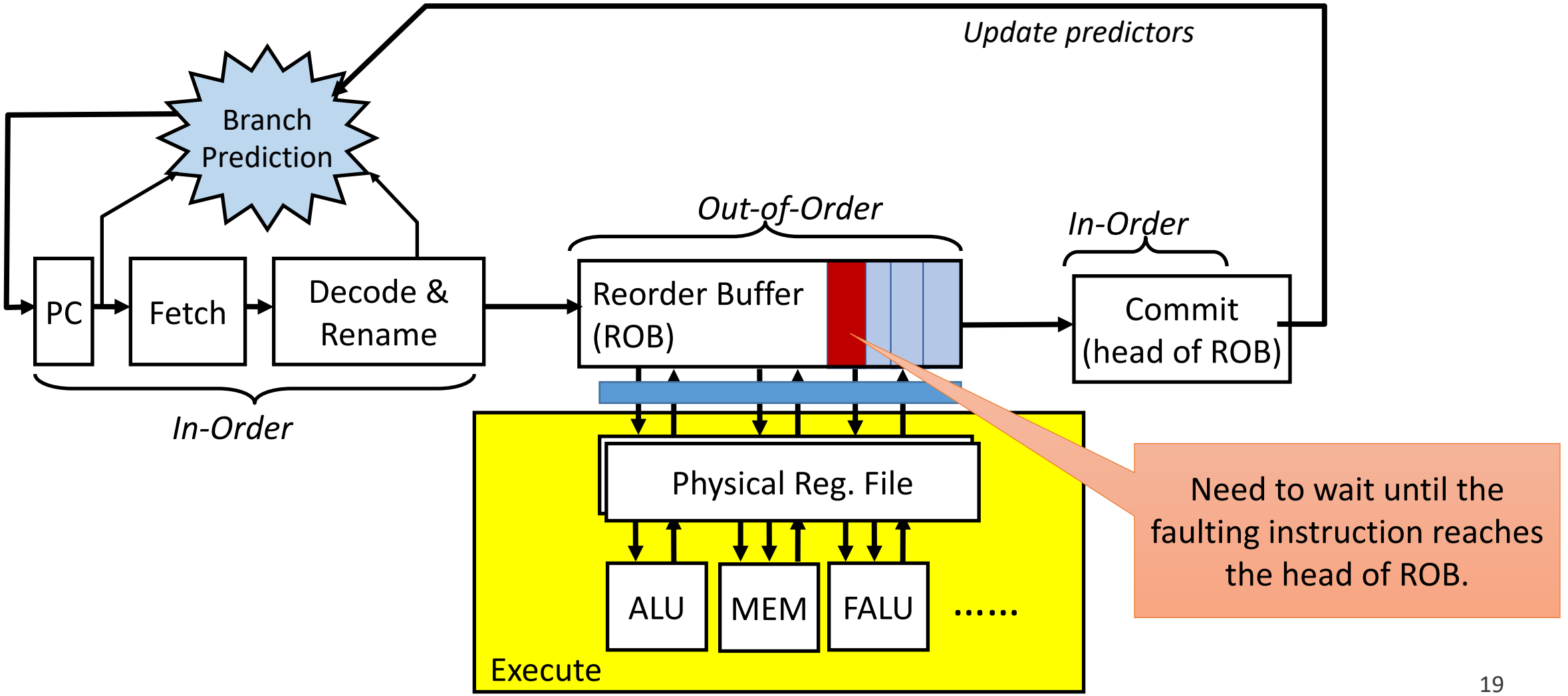
- Exceptions create a control-flow dependence
- Options for handling this dependence:

• Stall	No
• Bypass	No
• Find something else to do	No
• Speculate!	Most common approach!

- How can we handle rollback on mis-speculation?

Delay state update until commit on speculated instructions

Handling Exceptions in OoO Processors



Terminology

A **speculative** instruction may squash.

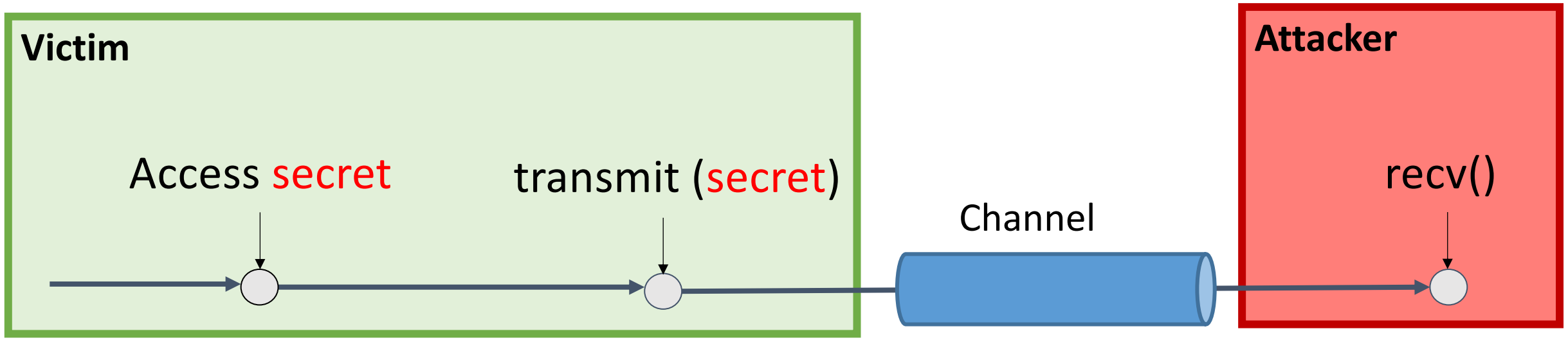
- When executed, can change uArch state

A **Transient** instruction *will* squash, i.e., will not commit.

A **Non-Transient** instruction will not squash, i.e., will eventually retire.

That is, **transient instructions** are unreachable on a non-speculative microarchitecture.

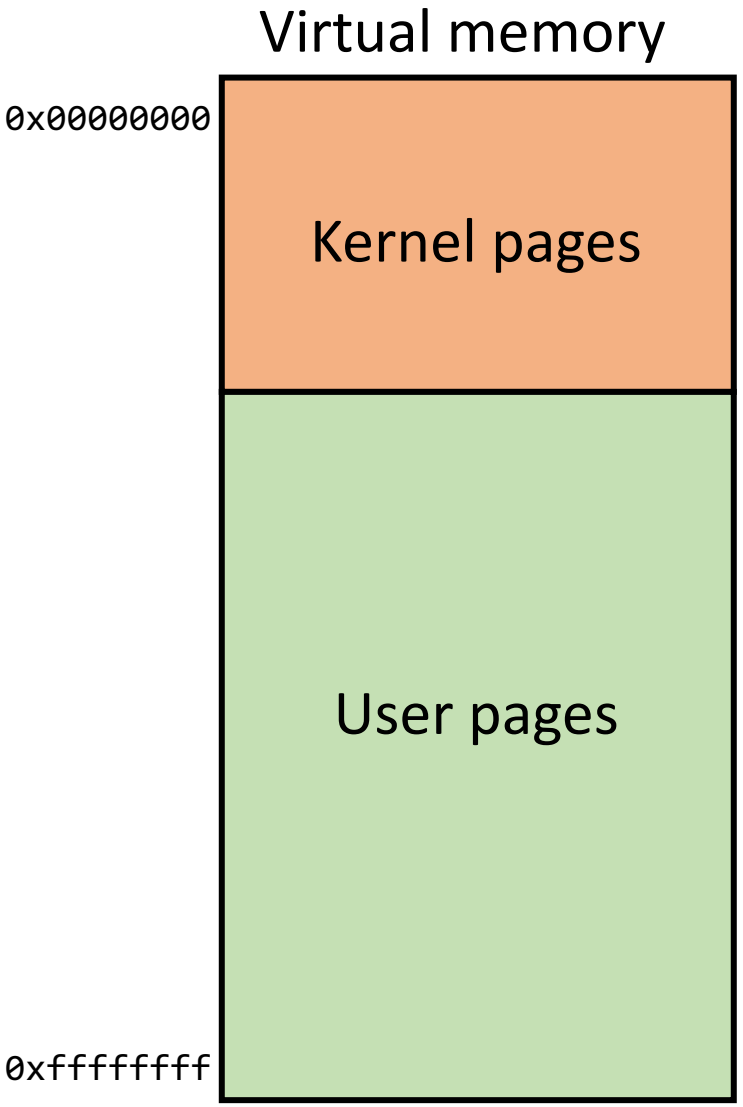
General Attack Schema



- The difference between transient and non-transient side channels
  - Whether the secret access or transmitter execution is transient

Meltdown & Spectre

Kernel/User Pages



- In x86, a process's virtual address space includes kernel pages, but kernel pages are only accessible in kernel mode
  - For performance purpose
  - Avoids switching page tables on context switches
- What will happen if accessing kernel addresses in user mode?
  - Protection fault

Meltdown

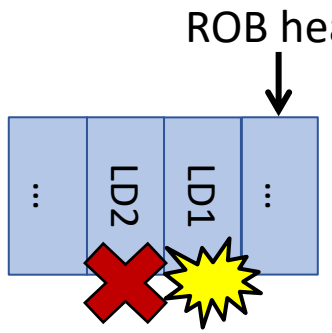
Exception handling is deferred when the instruction reaches the head of ROB.

- Problem: Speculative instructions can change uArch state, e.g., ca

- Attack procedure

1. Setup: Attacker allocates `probe_array`, with 256 cache lines. Flu es all its cache lines
2. Transmit: Attacker executes

```
……
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
```



3. Receive: After handling protection fault, attacker performs cache side channel attack to figure out which line of `probe_array` is accessed → recovers `byte`

Meltdown Type Attacks

- Can be used to read arbitrary memory
- Leaks across privilege levels
  - OS ←→ Application
  - SGX ←→ Application (e.g., Foreshadow)

- Mitigations:
  - HW: Stall speculation; Register p
  - SW: Do not let user and kernel share address space (KPTI) -> broken by several groups (talks at BlackHat)
- We generally consider it as a design **bug**

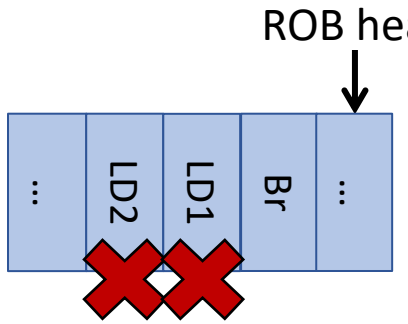
```
……
Ld1: uint8_t byte = *kernel_address;
Ld2: unit8_t dummy = probe_array[byte*64];
```

Spectre Variant 1 – Exploit Branch Condition

- Consider the following kernel code, e.g., in a syste

```
Br:  if (x < size_array1) {
Ld1:       secret = array1[x]
Ld2:       y = array2[secret*64]
     }
```

Always malicious?
No. It may be a benign misprediction.
We do not consider Spectre as a bug.



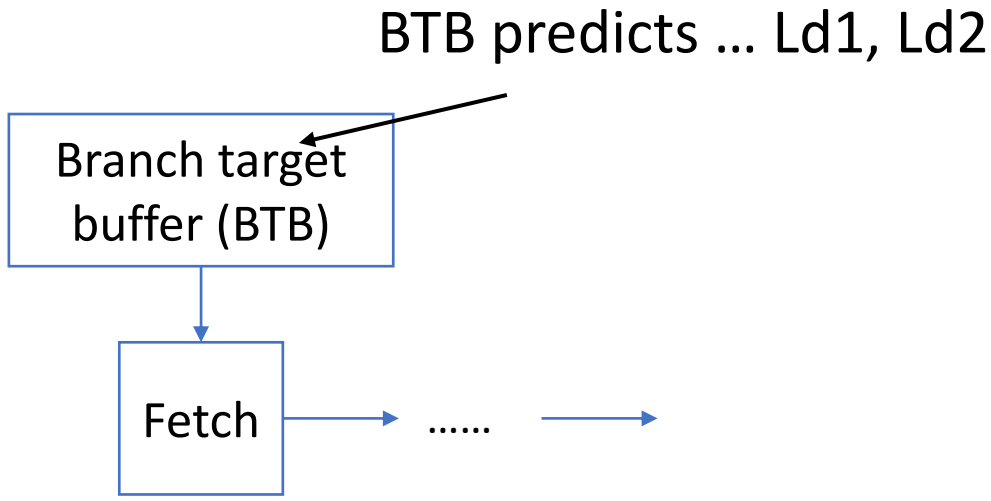
Attacker to read arbitrary memory:

1. Setup: Train branch predictor
2. Transmit: Trigger branch misprediction; *&array1[x]* maps to some desired kernel address
3. Receive: Attacker probes cache to infer which line of *array2* was fetched

Spectre Variant 2 – Exploit Branch Target

- Most BTBs store partial tags and targets…
  - <last n bits of current PC, target PC>

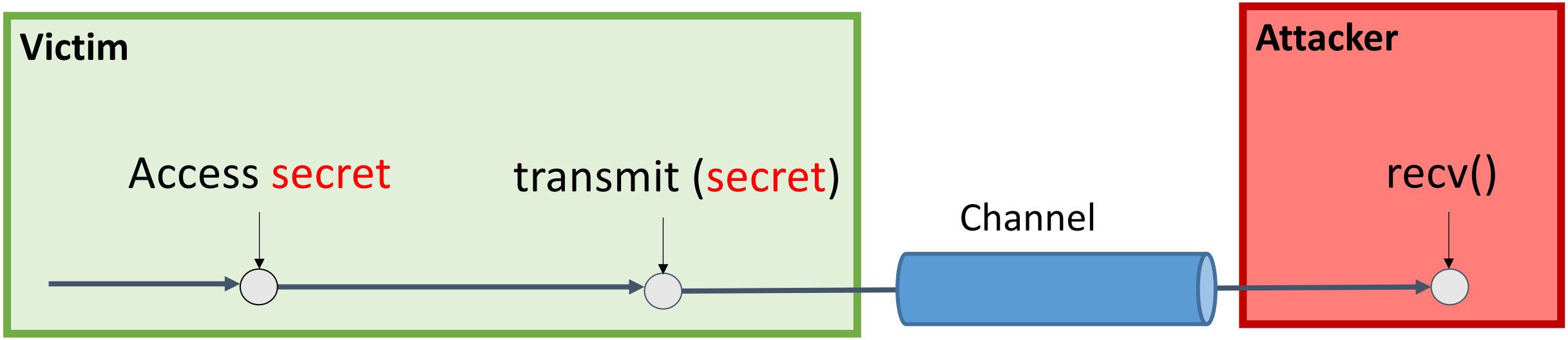
```
oxfff110  Br: if (…) {
          …       }
          …
oxfff234  Ld1: secret = array1[x]
          Ld2: y = array2[secret*4096]
```



Train BTB properly → Execute arbitrary gadgets speculatively

Mitigations?

General Attack Schema



- Traditional (non-transient) attacks
  - Data in-use

Hard to fix

- Transient attacks: can leak data-at-rest
  - Meltdown = transient execution + deferred exception handling
  - Spectre = transient execution on wrong paths

"Easy" to fix

Hard to fix

Takeaways

Transient execution attacks *use* (not "are") side/covert channels.

"Spectre" (wrong-path execution) is **fundamental**.
Speculation/prediction is not perfect.

"Meltdown" (deferred exceptions) is **not fundamental**.

Next Paper Discussion:

An Analysis of Speculative Type Confusion Vulnerabilities in the Wild