Transient Execution Attacks

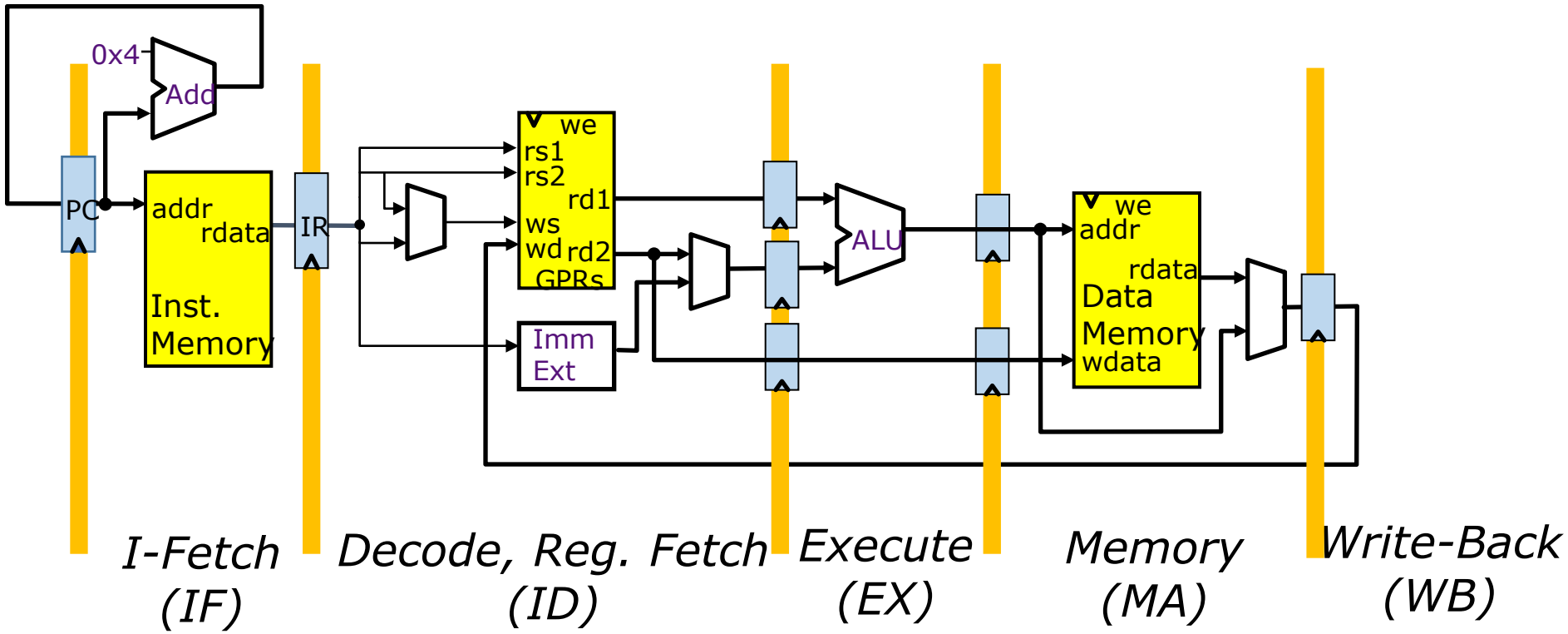
Mengjia Yan

Spring 2024

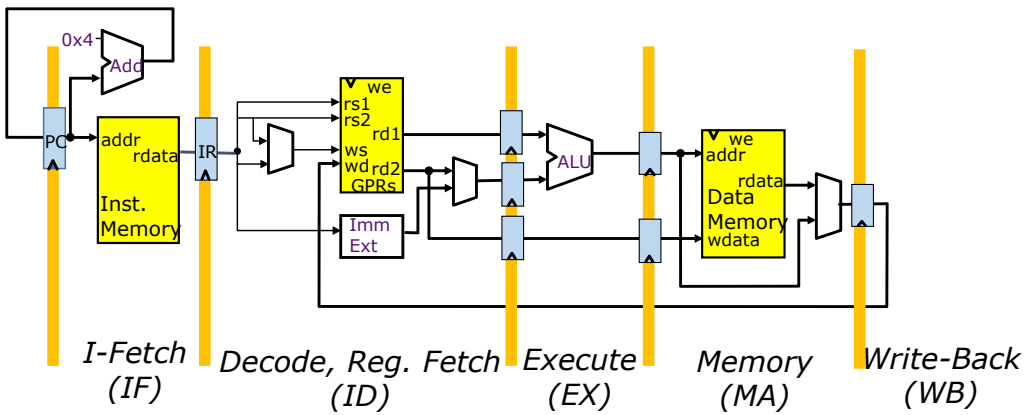
Outline

- What is speculative execution?
- How does Meltdown work?
  - We will connect the dots between a hardware optimization and a software optimization.
- How does Spectre and its variations work?
  - Let’s try to see through these variations and understand the fundamental problem.

Recap: 5-stage Pipeline



Recap: 5-stage Pipeline



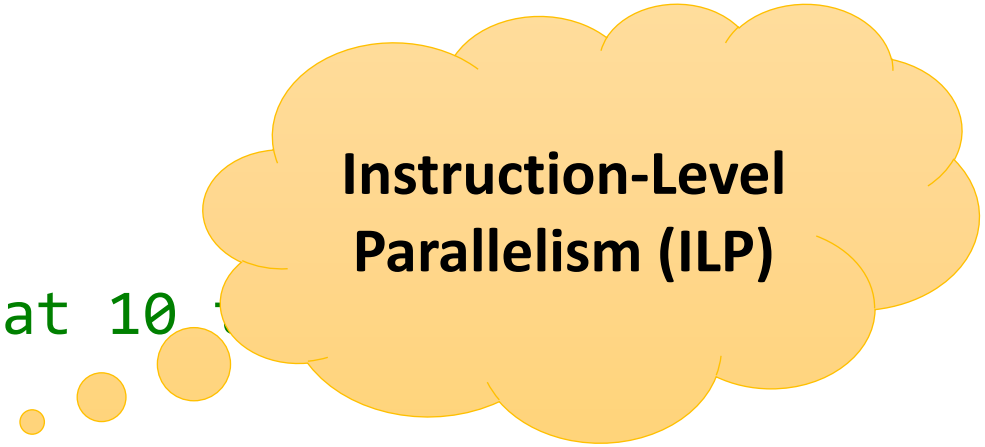
- In-order execution:
  - Execute instructions according to the program order
  - One instruction max per pipeline stage

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

Build High-Performance Processors

Example #1:

```
FMUL f1, f2, f3    ; 10 cycles
ADD  r4, r4, r1    ; 1 cycle  -> repeat 10
……
```

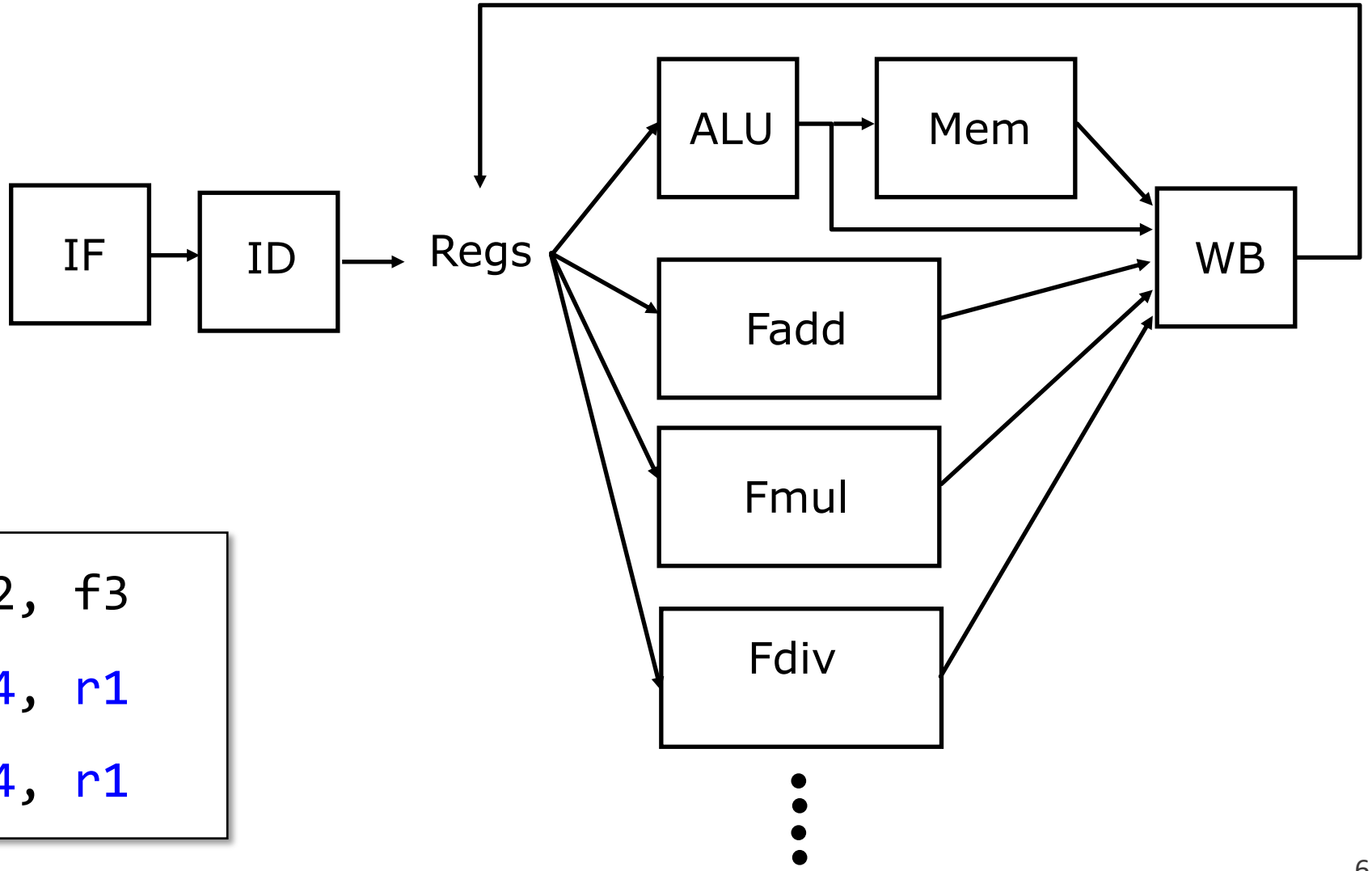


when there is **NO** data-dependency
or control-flow dependency

Example #2:

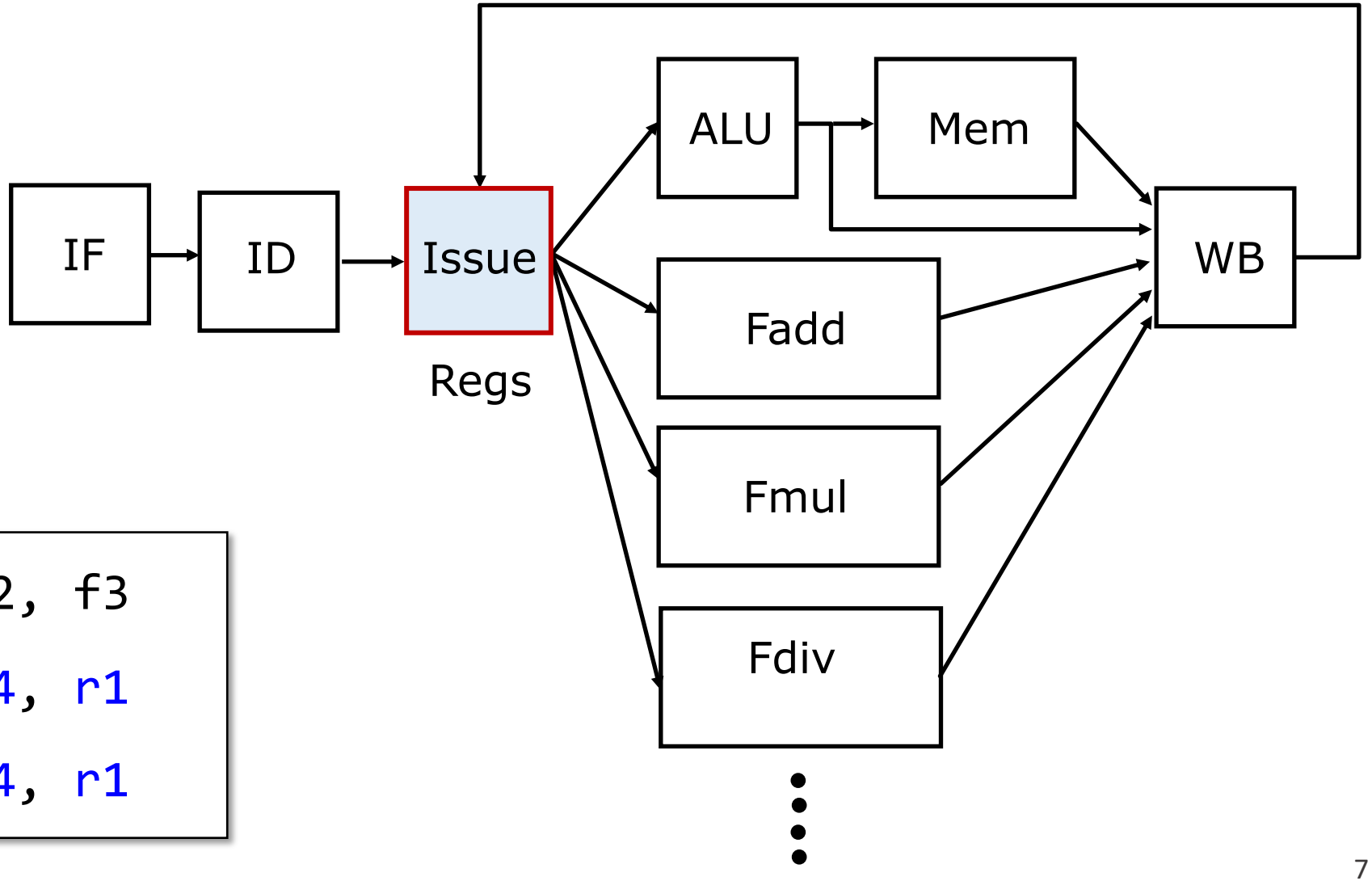
```
LD  r3, 0(r2)      ; 1-100 cycles
ADD r4, r4, r1     ; 1 cycle -> repeat 10 times
……
```

Technique #1: Add More Functional Units



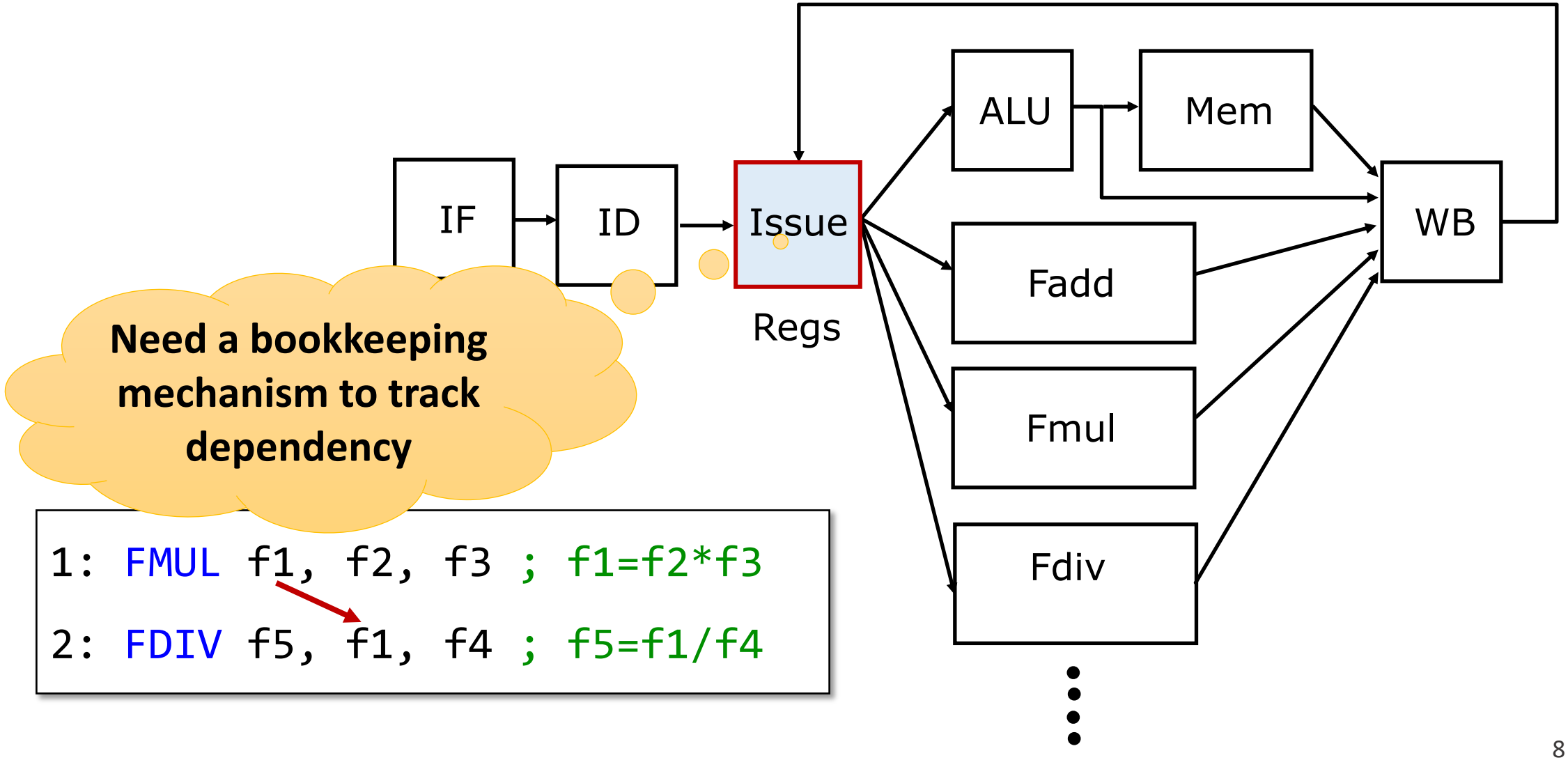

```
1: FMUL f1, f2, f3
2: ADD  r4, r4, r1
3: ADD  r4, r4, r1
```

Technique #1: Add More Functional Units




```
1: FMUL f1, f2, f3
2: ADD  r4, r4, r1
3: ADD  r4, r4, r1
```

Technique #1: Add More Functional Units



Technique #2: Scoreboard

Functional Unit	Busy?	Dest Reg	Src1 Reg	Src2 Reg
Int ALU				
Mem				
Fadd				
Fmul				
Fdiv				

Technique #2: Scoreboard

Functional Unit	Busy?	Dest Reg	Src1 Reg	Src2 Reg
Int ALU				
Mem				
Fadd				
Fmul	Y	f1	f2	f3
Fdiv				

```
1: FMUL f1, f2, f3
2: ADD  r4, r4, r1
```

No dependency, feel free to issue the ADD

Technique #2: Scoreboard

Functional Unit	Busy?	Dest Reg	Src1 Reg	Src2 Reg
Int ALU				
Mem				
Fadd				
Fmul	Y	f1	f2	f3
Fdiv				

Read-after-Write (RAW)

```
1: FMUL f1, f2, f3
2: FDIV f5, f1, f4
```

Write-after-Write (WAW)

```
1: FMUL f1, f2, f3 ; 10 cycles
2: FADD f1, f4, f5 ; 4 cycles
```

Technique #2: Scoreboard

- Upon issue an instruction, check:
  1. Whether any ongoing instructions will generate values for my source registers
  2. Whether any ongoing instructions will modify my destination register

We call such a processor: **in-order issue, out-of-order completion**.

A problem: how to handle interrupts/exceptions?

Exception in OoO Processors: Example #1

```
1: LD  r3, 0(r2)     ; Exception in 3 cycles
2: ADD r4, r4, r1    ; 1 cycle
```

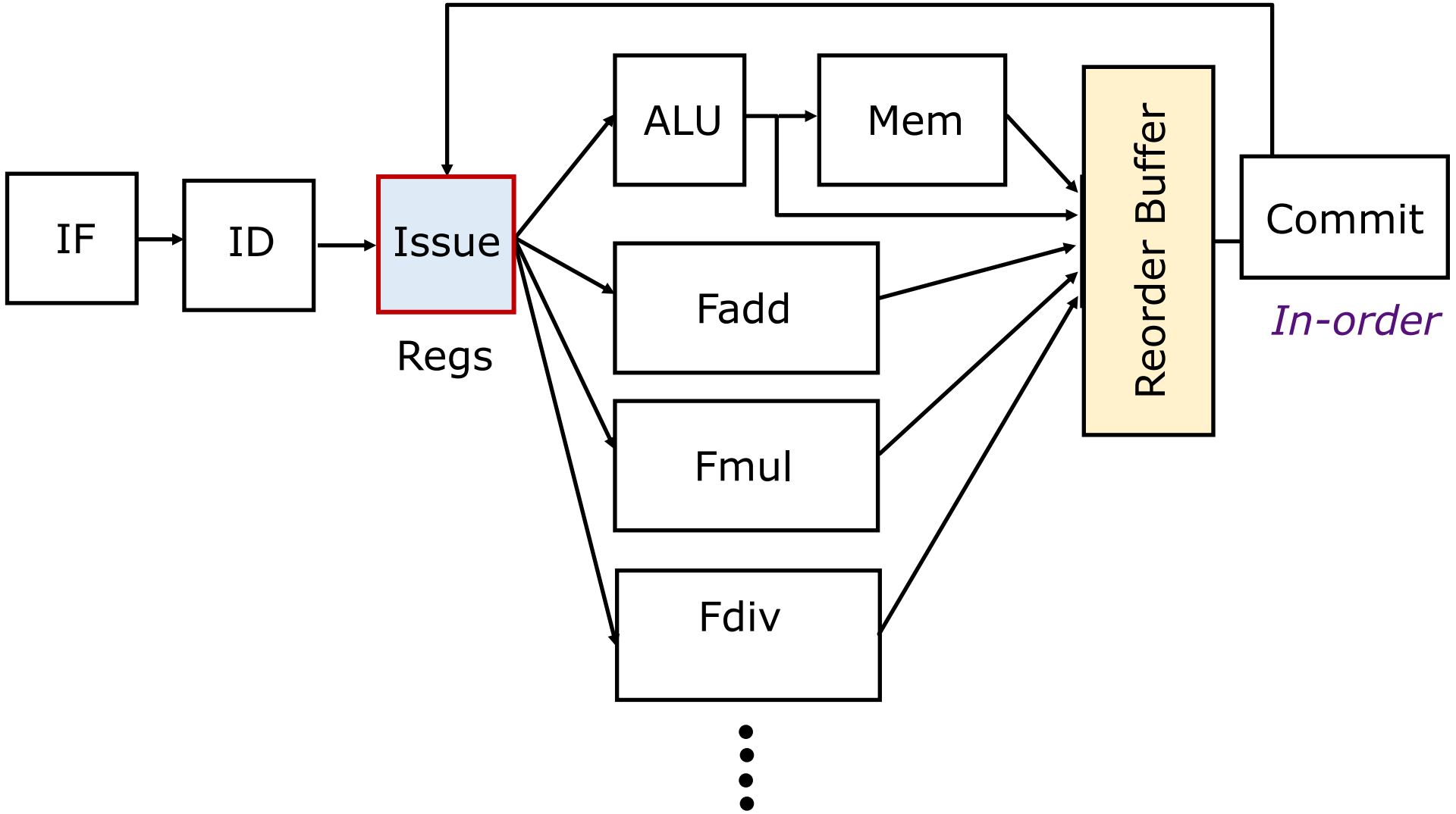
	1	2	3	4	5	6	7	8
1: LD	IF	ID	Issue	ALU	Mem	Mem	Mem	**Exception**
2: ADD		IF	ID	Issue	ALU	**WB**		

Exception in OoO Processors: Example #2

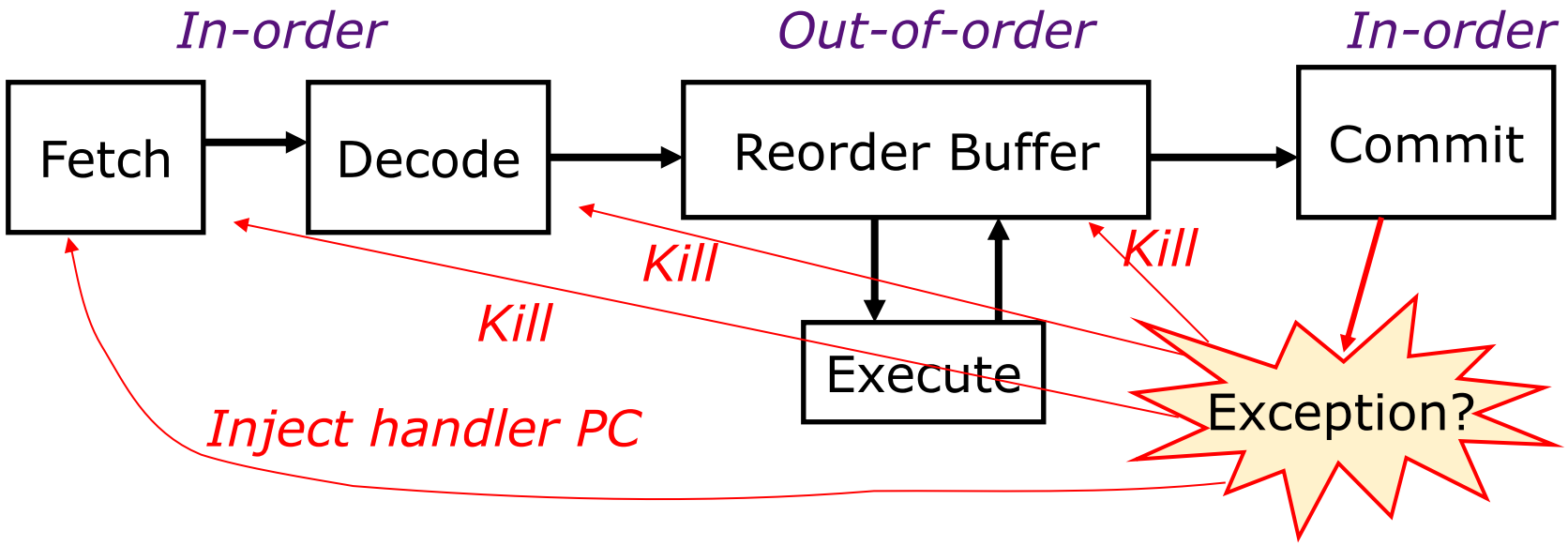
```
1: FMUL f1, f2, f3   ; 10 cycles
2: LD   r3, 0(r2)    ; Exception in 1 cycl
```

	1	2	3	4	5	6	7	8
1: FMUL	IF	ID	Issue	FMUL	FMUL	FMUL	FMUL	…
2: LD		IF	ID	Issue	ALU	Mem	**Exception**	

Technique #3: In-order Commit



Another Way to Draw It



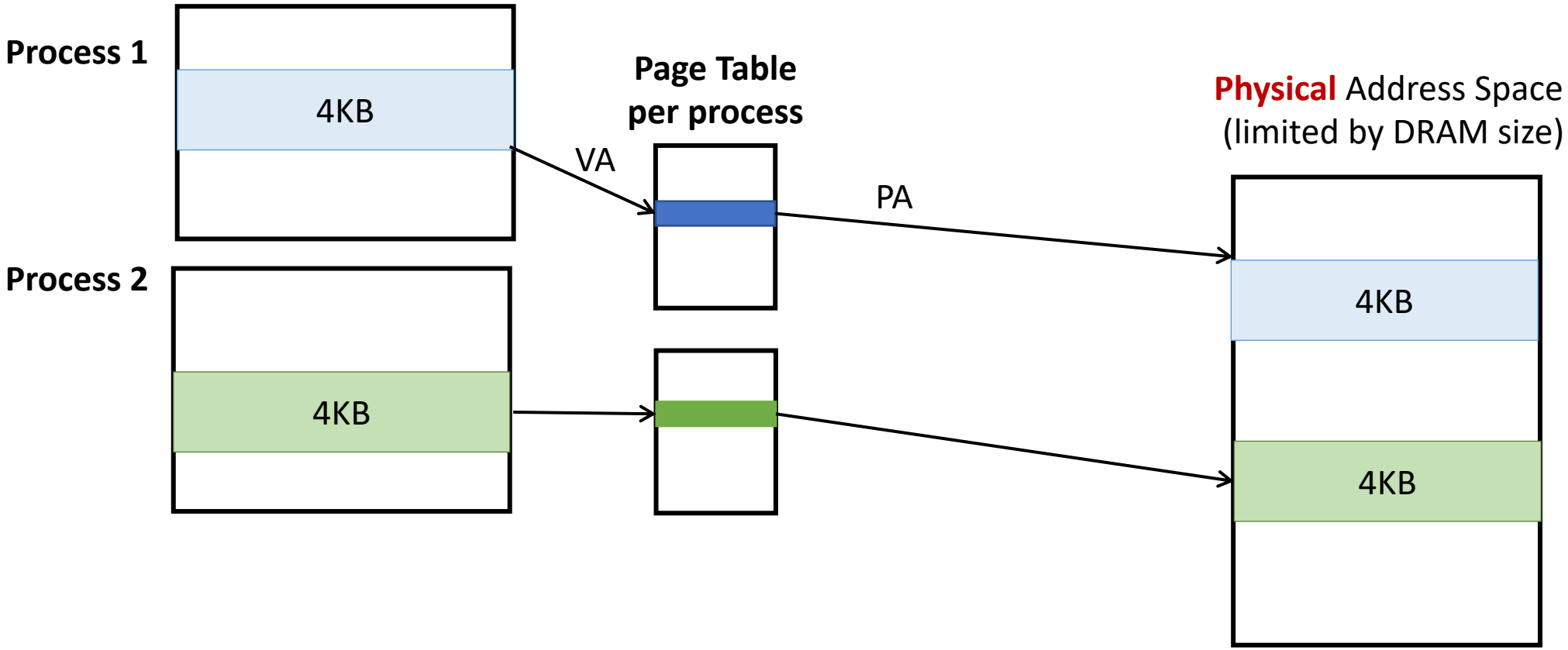
To know more advanced out-of-order (OoO) features, take 6.5900 [6.823]

Re-examine Examples With In-order Commit

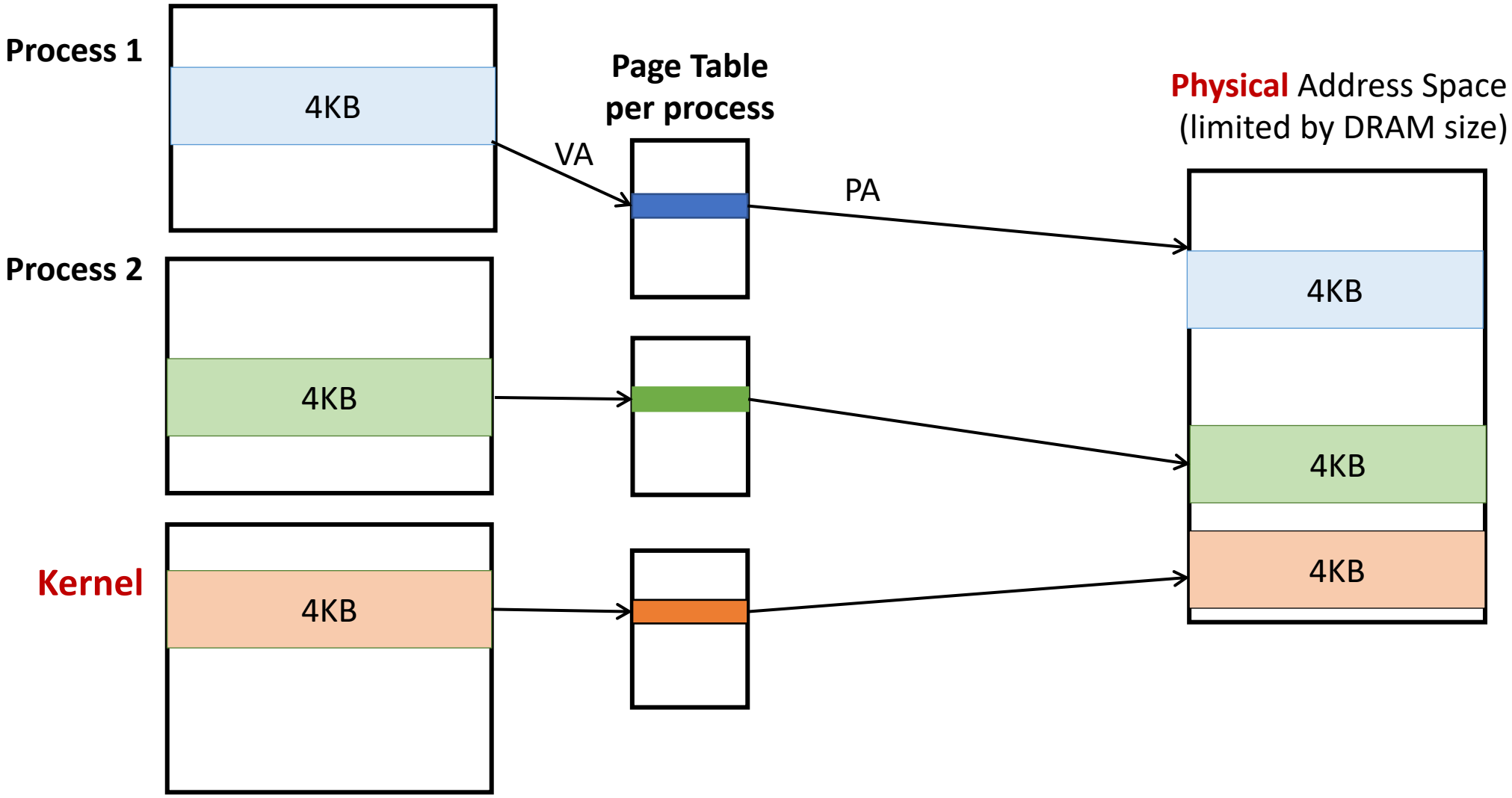
```
1: LD   r3, 0(r2)     ; Exception in 3 cycles
2: ADD r4, r4, r1     ; 1 cycle
```

```
1: FMUL f1, f2, f3    ; 10 cycles
2: LD   r3, 0(r2)     ; Exception in 1 cycle
```

Recap: Page Mapping

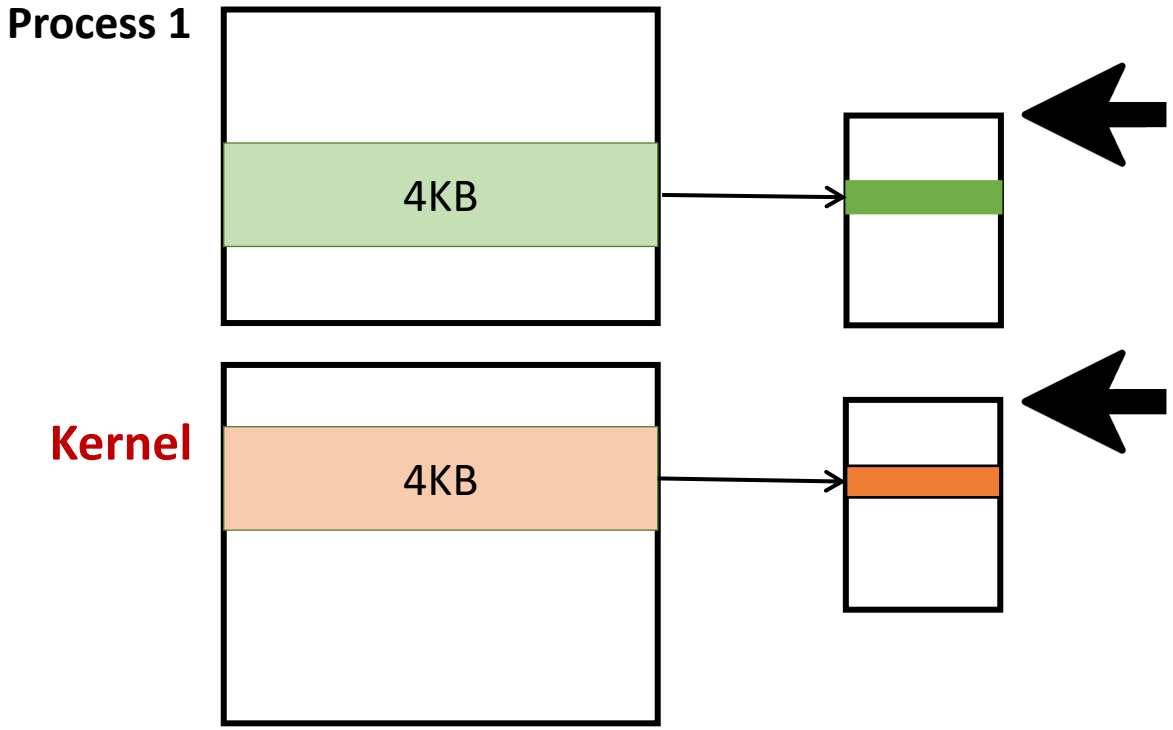


Mapping Kernel Pages



Jumping Between User and Kernel Space

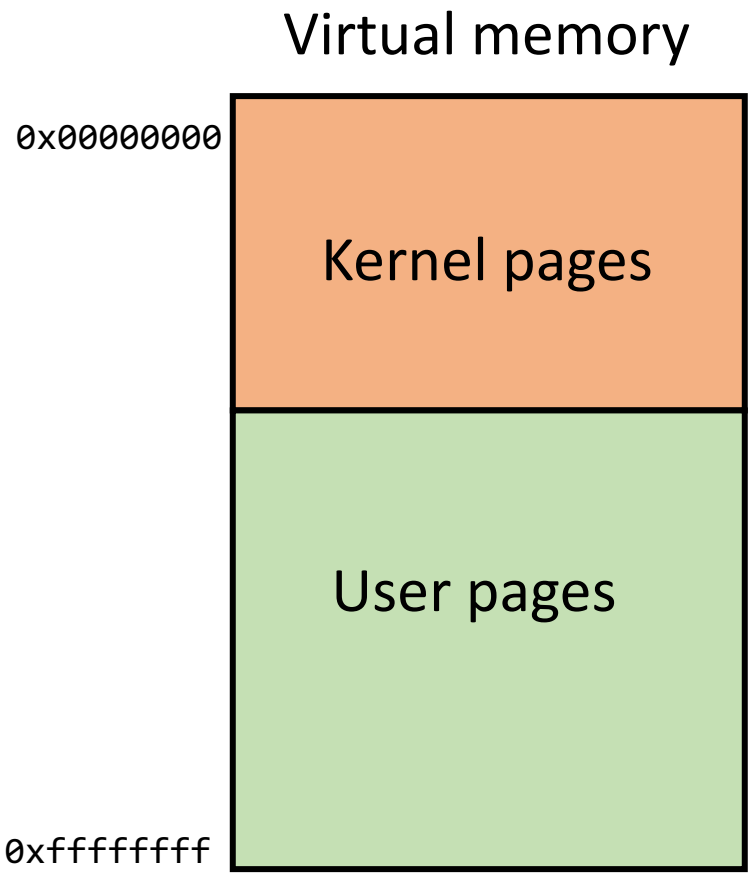
- Key challenge: need to make sure we use the correct page table
  - CR3 (in x86) or satp (in RISCV) stores the page table physical address

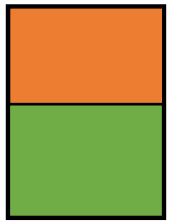


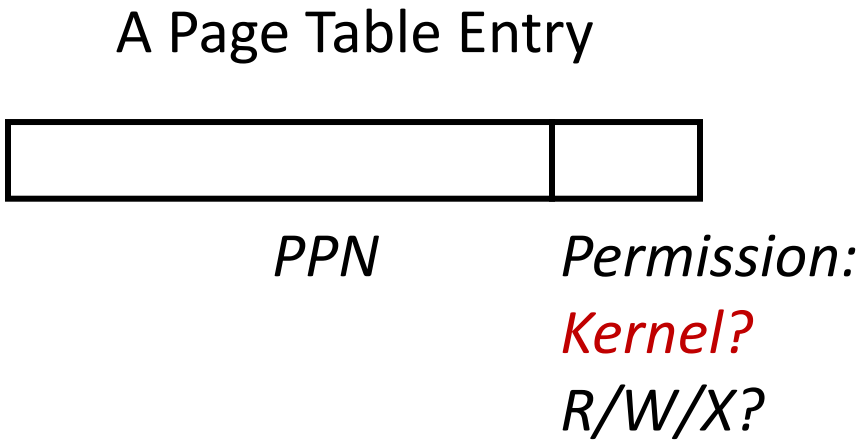
A Performance Optimization

- Context switch overhead:
  - Page table changes, so in many processors, we need to flush TLB
- But sometimes, we only go to kernel to do some simple things
  - E.g., getpid()
- The optimization: map kernel address into user space in a **secure** way

Map Kernel Pages Into User Space





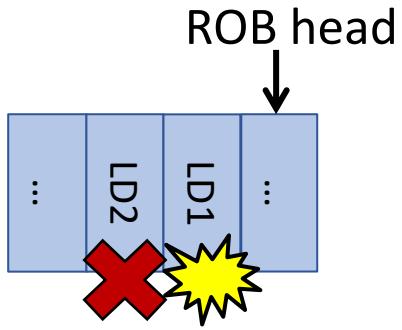


- What will happen if accessing kernel addresses in user mode?
  - Protection fault

Meltdown

- Meltdown explores the combined effects of two optimizations
  - Hardware optimization: out-of-order execution
  - Software optimization: mapping kernel addresses into user space
- Let's analyze the timing carefully
- Attack outcome: user space applications can read arbitrary kernel data

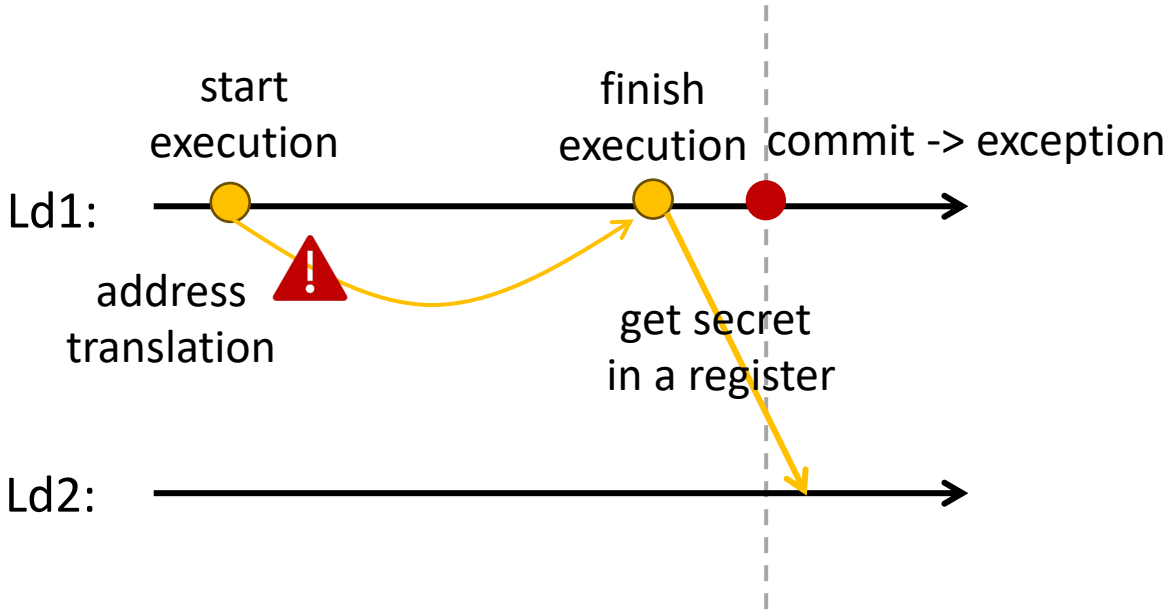
```
……
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
```



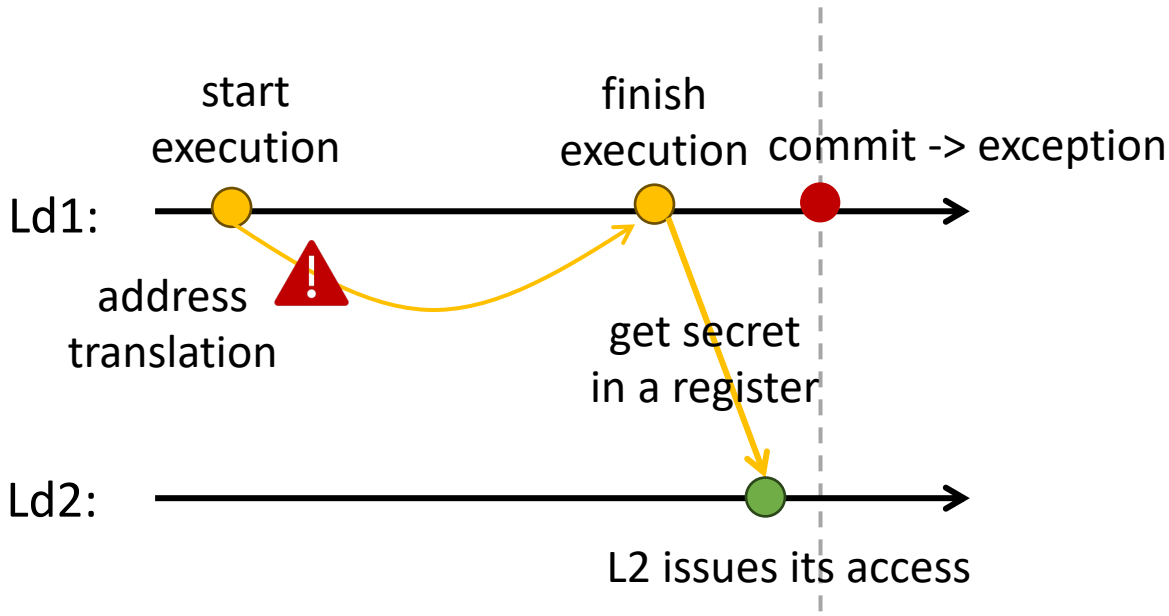
Meltdown Timing

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

Case 1: Fail. Ld2 is squashed before the corresponding memory access is issued.



Case 2: Attack works. Ld2's request is sent out before the instruction is squashed.



Meltdown w/ Flush+Reload

1. Setup: Attacker allocates `probe_array`, with 256 cache lines. Flushes 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 Mitigations

- Stop one of the optimizations should be sufficient
  - SW: Do not let user and kernel share address space (KPTI) -> broken by several groups (e.g., *EntryBleed*)
  - HW: Stall speculation; Register poisoning

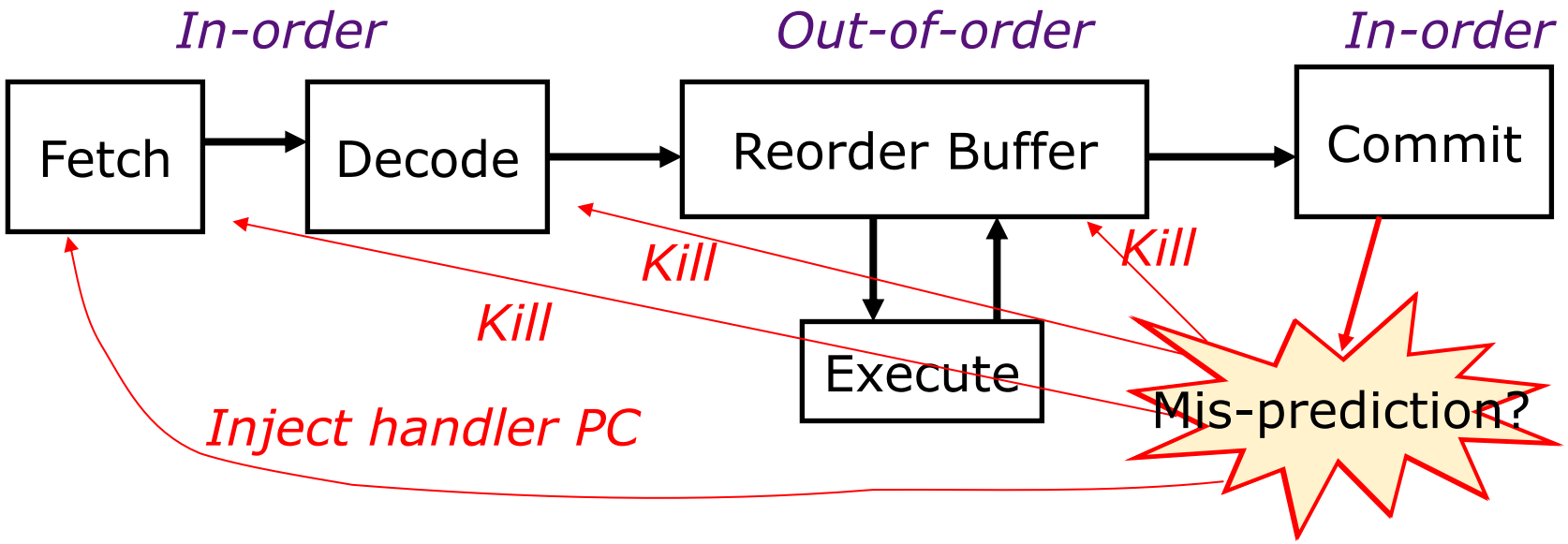
```
……
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
```

- We generally consider Meltdown as a design **bug**

Will Liu, EntryBleed, https://www.willsroot.io/2022/12/entrybleed.html?m=1

Branch Prediction

- Motivation: control-flow penalty
  - *Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution!*



Branch Prediction

- Naïve approach: PC+4

- More advanced, predict two things:
  - Direction of a branch (whether a branch is taken or not)
  - The target address of a branch

Branch Direction Predictor

- 1-bit predictor
  - If taken, set the bit to 1
  - If not-taken, set the bit to 0
  - Predict using this bit
- 2-bit predictor … N-bit predictor
- More advanced:
  - Use global and local information together
  - Use Neural networks…

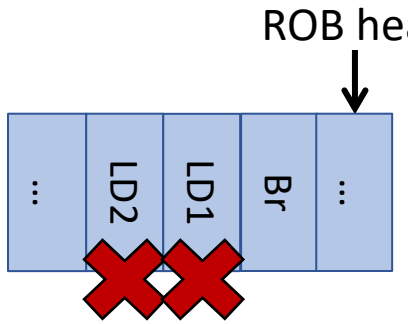
On not-taken ↓	↑ On taken			
		1	1	Strongly taken
		1	0	Weakly taken
		0	1	Weakly not-taken
		0	0	Strongly not-taken

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.



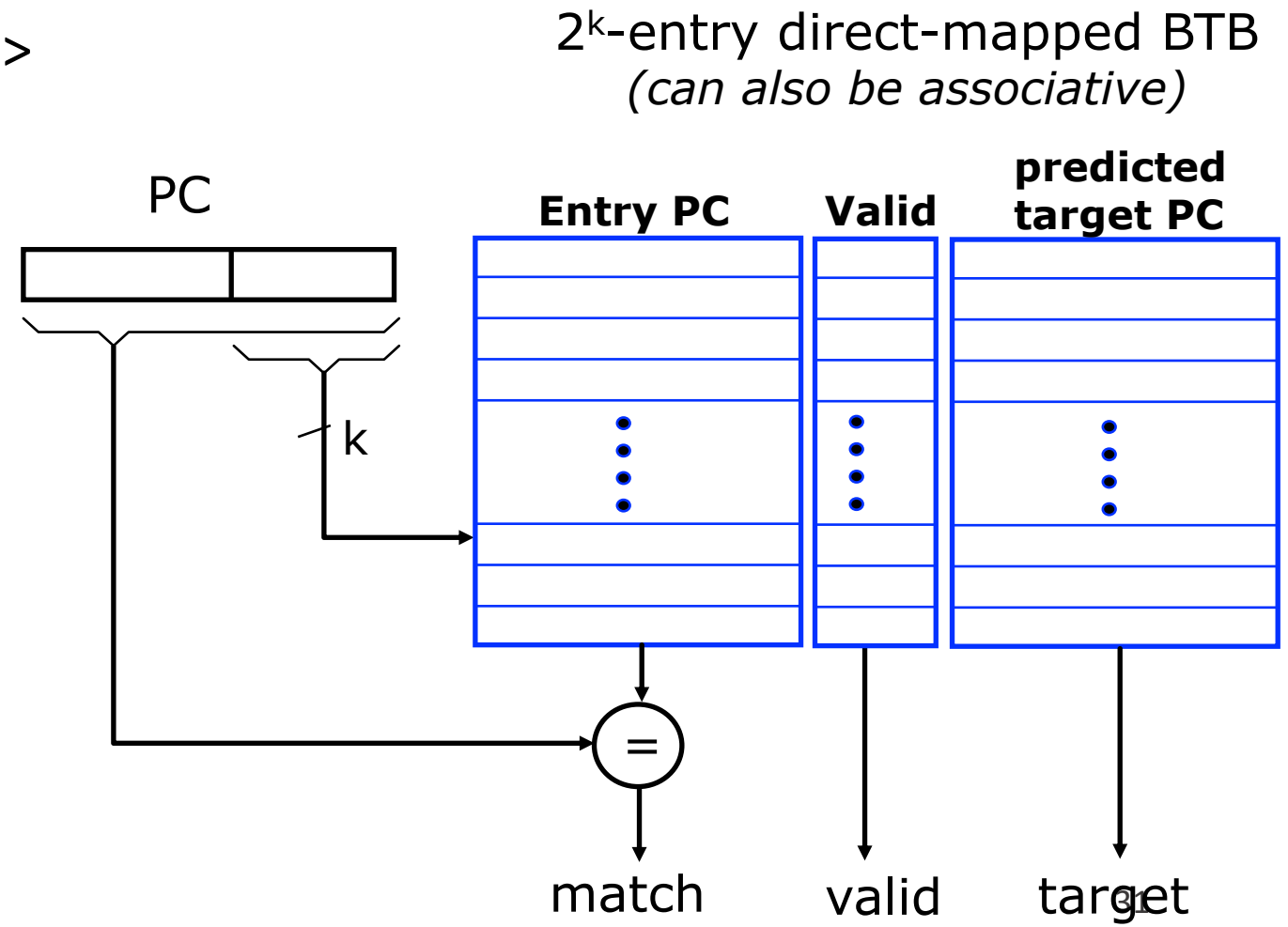
Attacker to read arbitrary memory:

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

More Branch Predictors

- How to predict the target address of a branch?
  - jal <label> and blt r1, r2, <label>
  - jalr <r1>
  - ret

- Two structures:
  - Branch Target Buffer (BTB)
  - RAS (Return Address Stack)

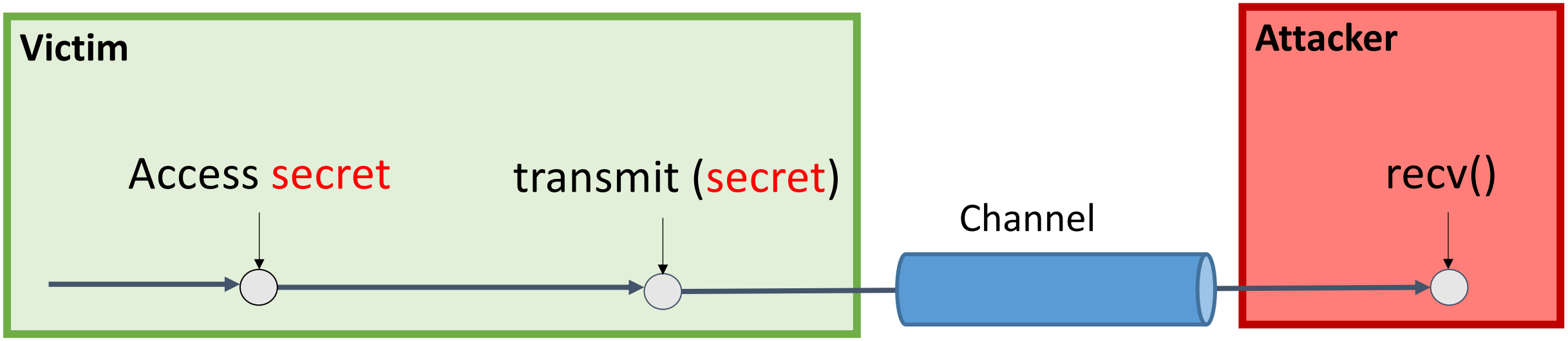


Spectre Variant 2 – Exploit Branch Target

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

Train BTB properly → Execute arbitrary gadgets speculatively

General Attack Schema



Apply the General Attack Scheme

The RSA Square-and-Multiply Exponentiation example.

Attackers aim to leak e

Which is **access** operation?
Which is **transmit** operation?

```
r = 1
for i = n-1 to 0 do
        r = sqr(r)
        r  = mod(r, m)
        if e_i == 1 then
                r = mul(r, b)
                r = mod(r, m)
        end
end
```

Apply the General Attack Scheme

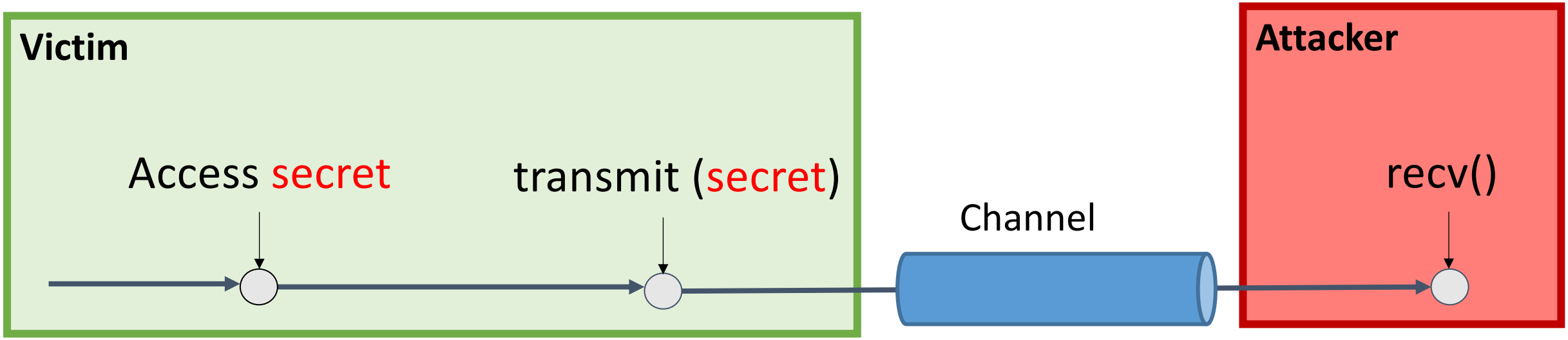
```
……
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
```

Which is **access** operation?
Which is **transmit** operation?

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

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

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

Next: Mitigations