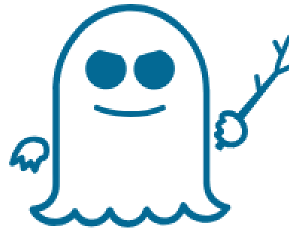
Transient Execution Attacks

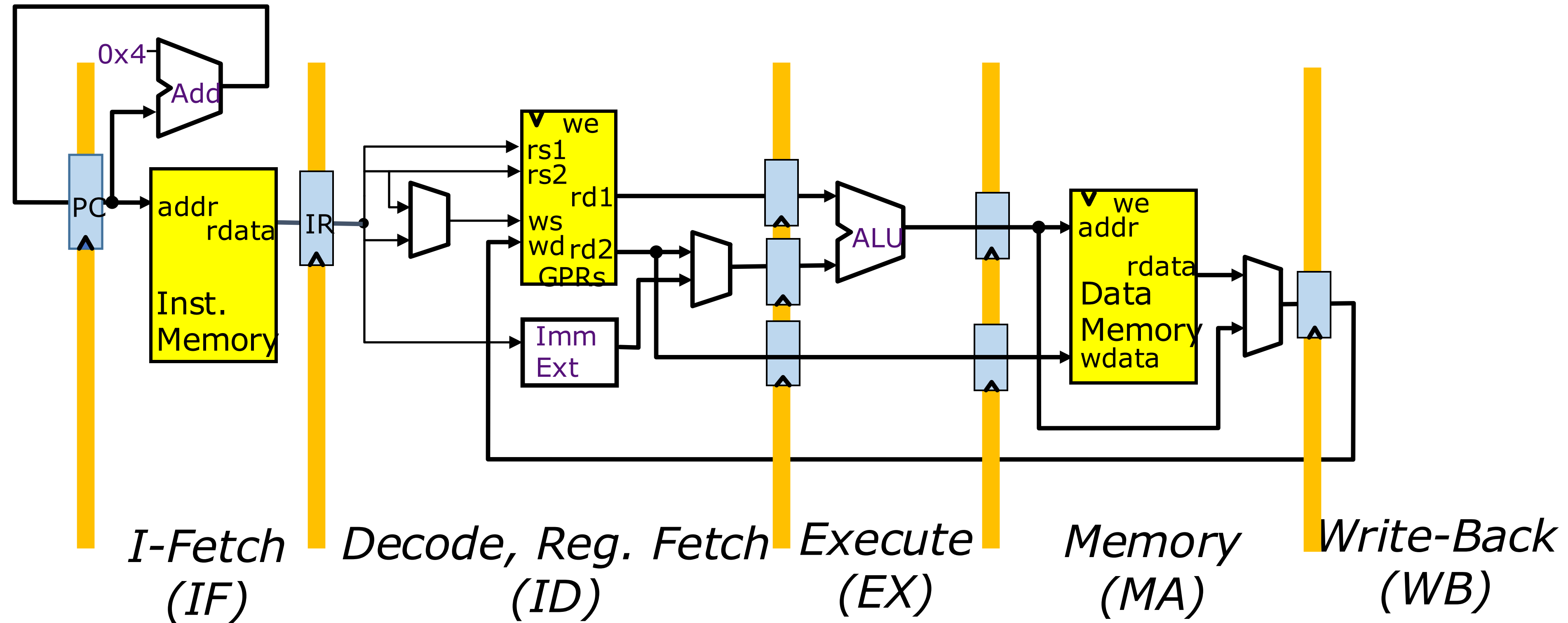
Mengjia Yan

Spring 2025

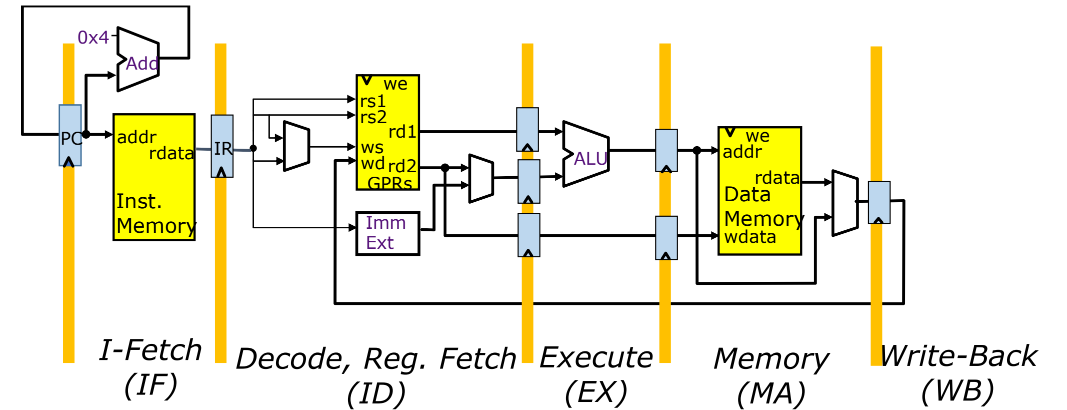
Outline

- Speculative execution
- Meltdown
- Spectre and its variations 

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:

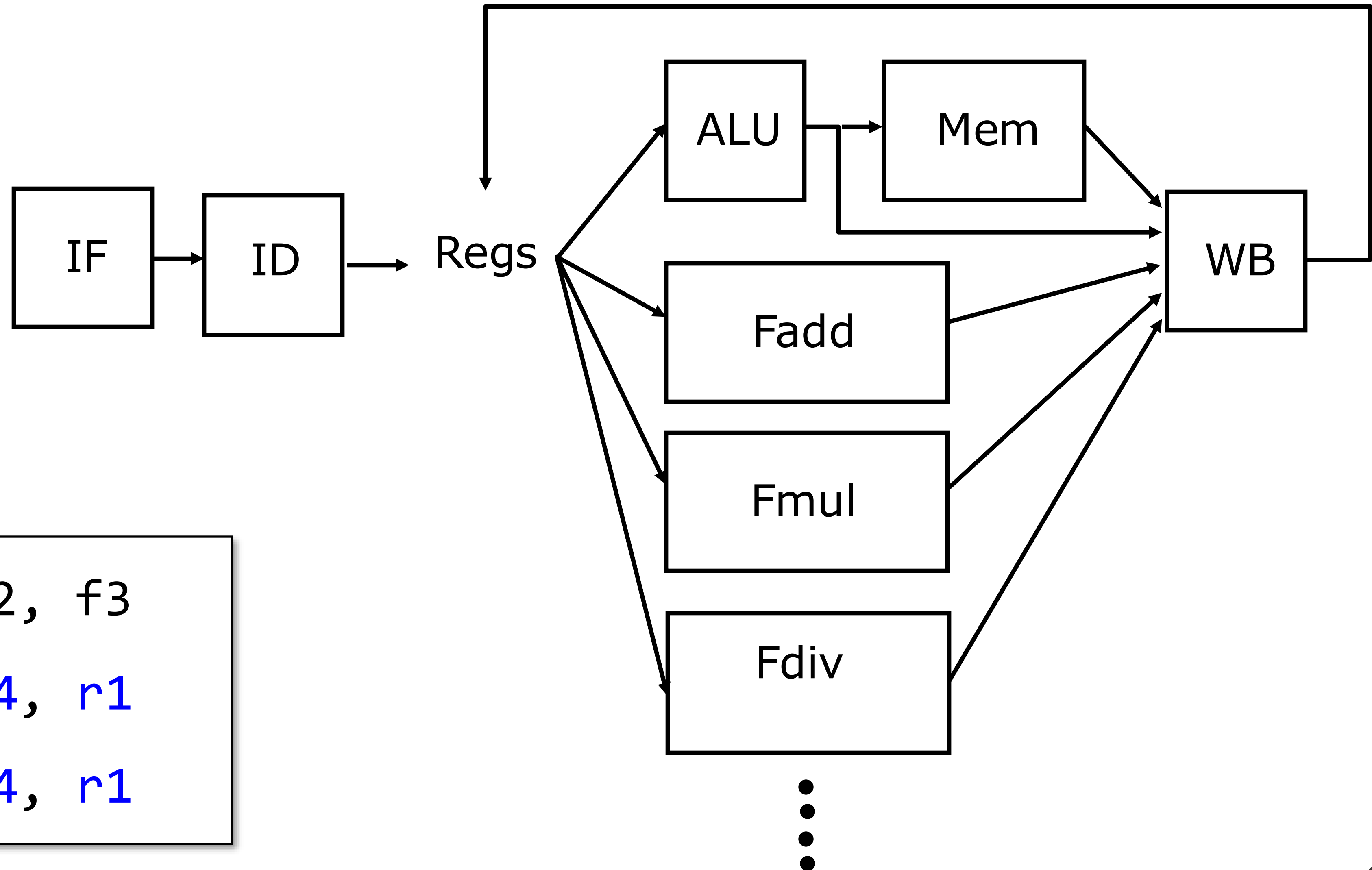
```
FMUL f1, f2, f3    ; 10 cycles
ADD  r4, r4, r1    ; 1 cycle   -> repeat 10 t
……
```

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

Example #2:

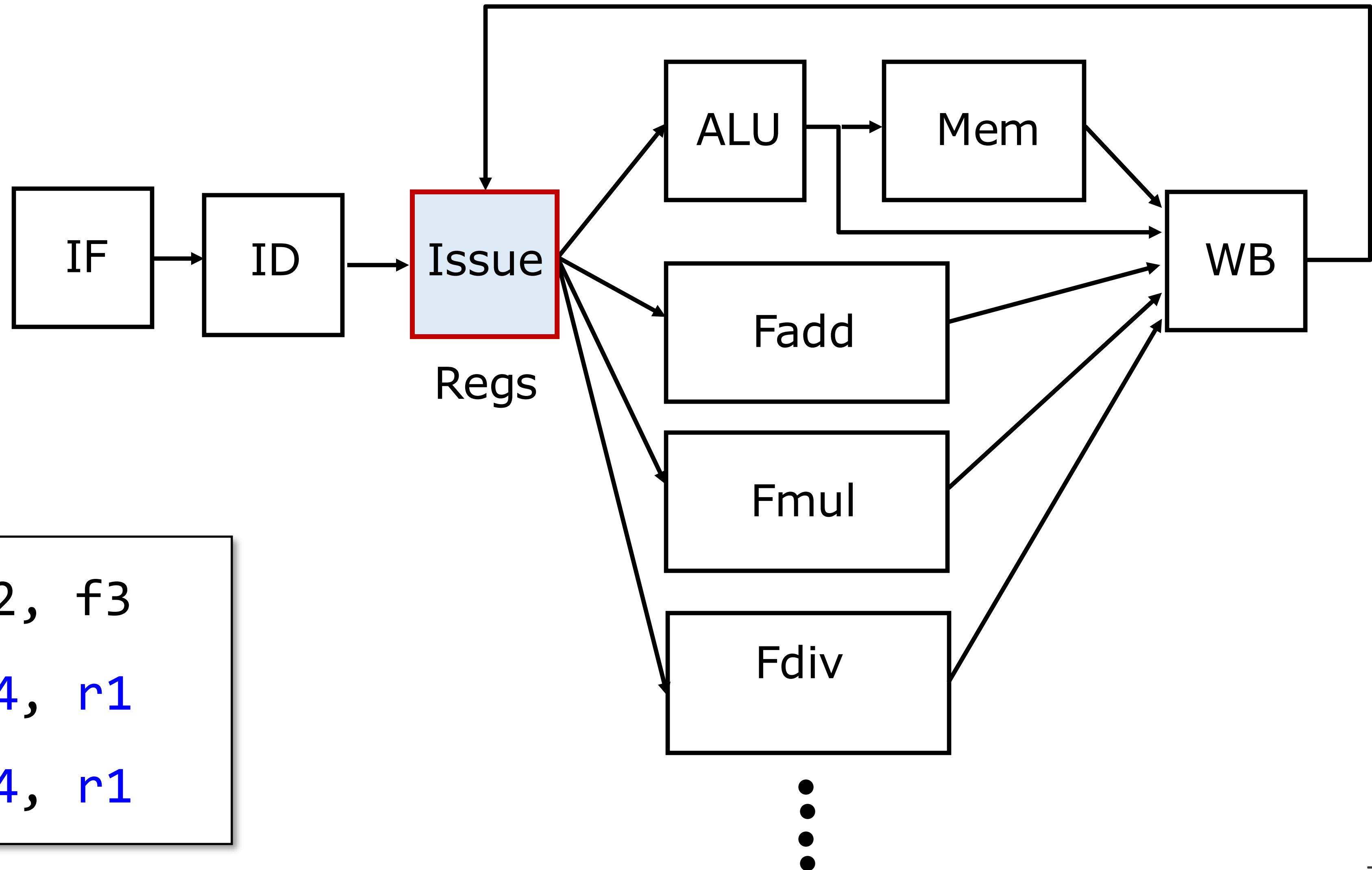
```
LD  r3, 0(r2)     ; 1-100 cycles
ADD r4, r4, r1    ; 1 cycle -> repeat 10 times
……
```

Technique #1: Add More Functional Units



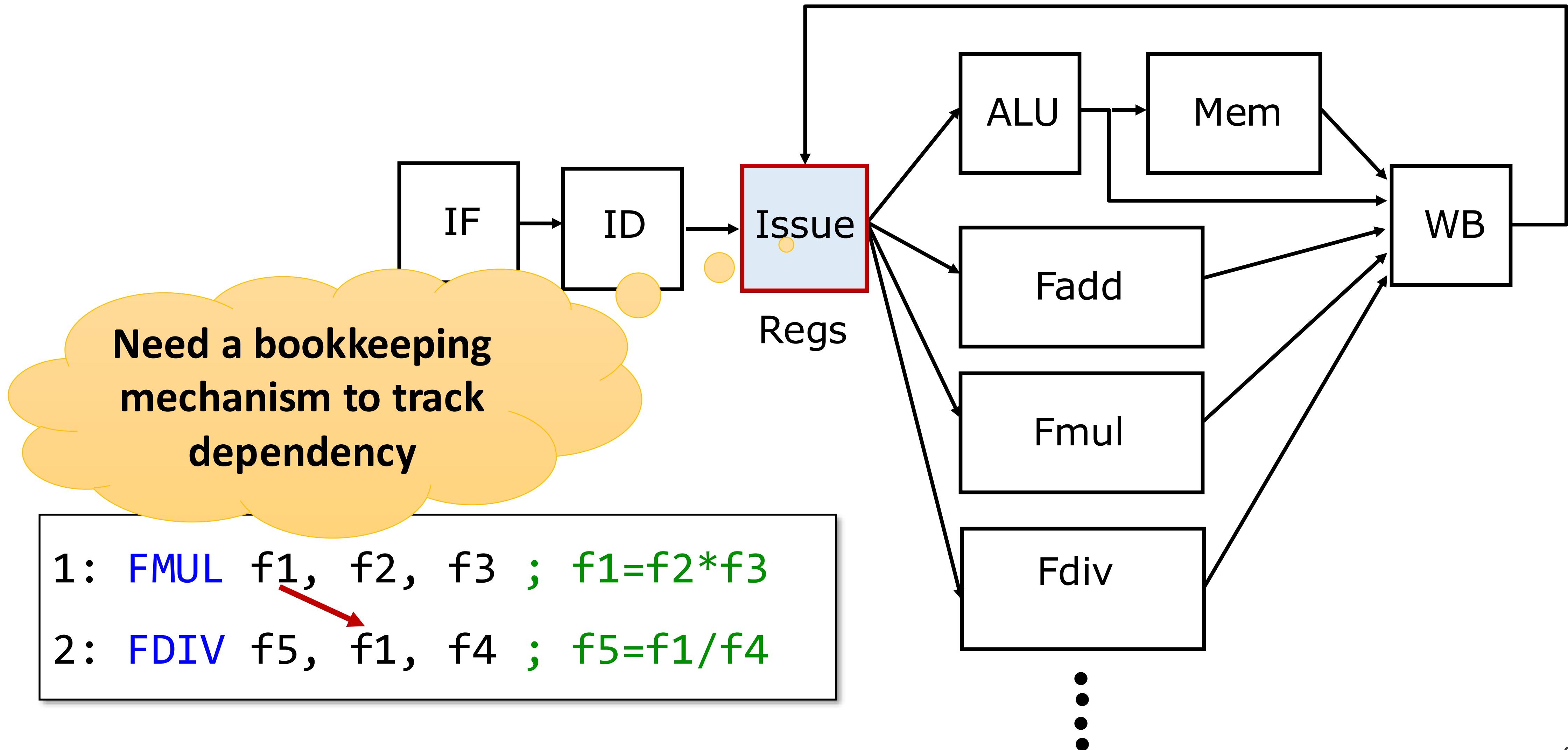

```
1: FMUL f1, f2, f3
2: ADD  r4, r4, r1
3: ADD  r4, r4, r1
```

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
```

Need to delay WB

	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**		

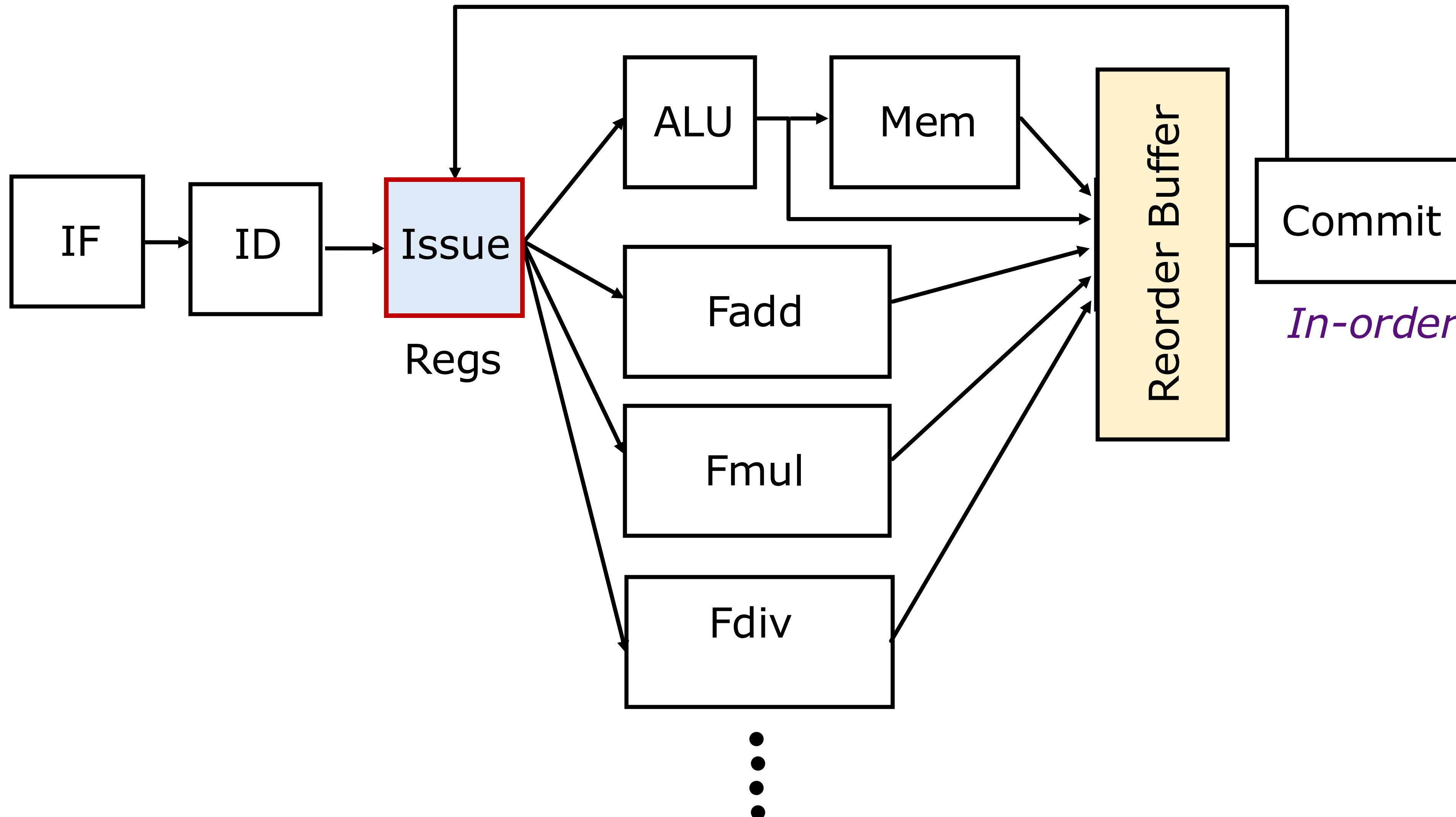
Exception in OoO Processors: Example #2

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

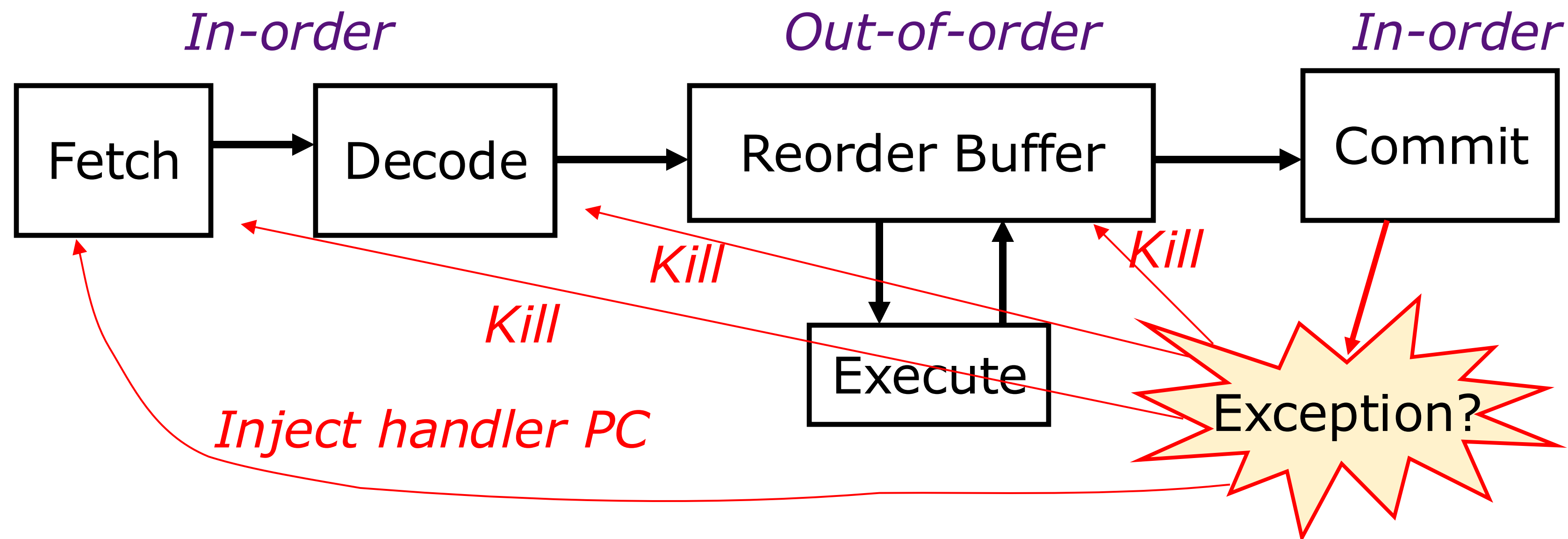
Need to delay Exception

	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

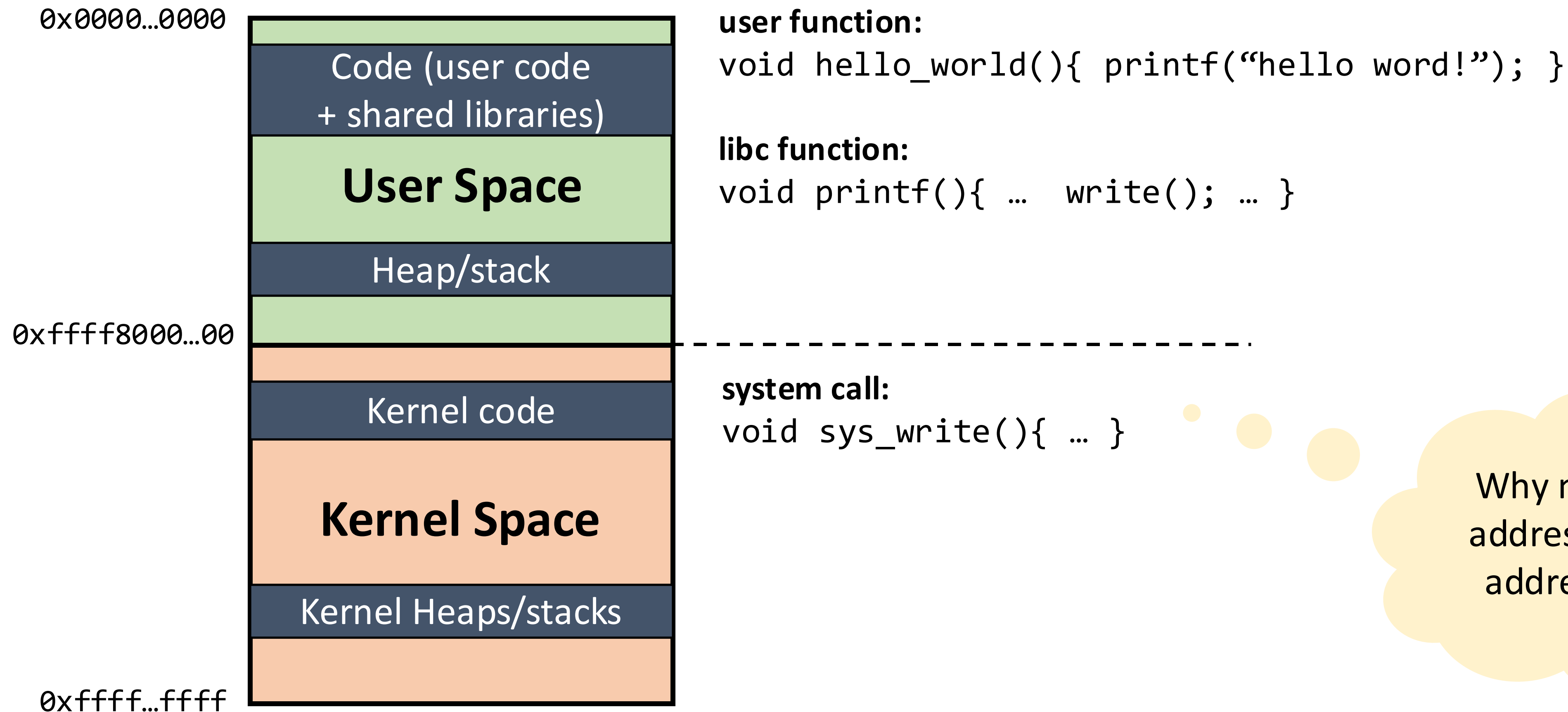


Another Way to Draw It

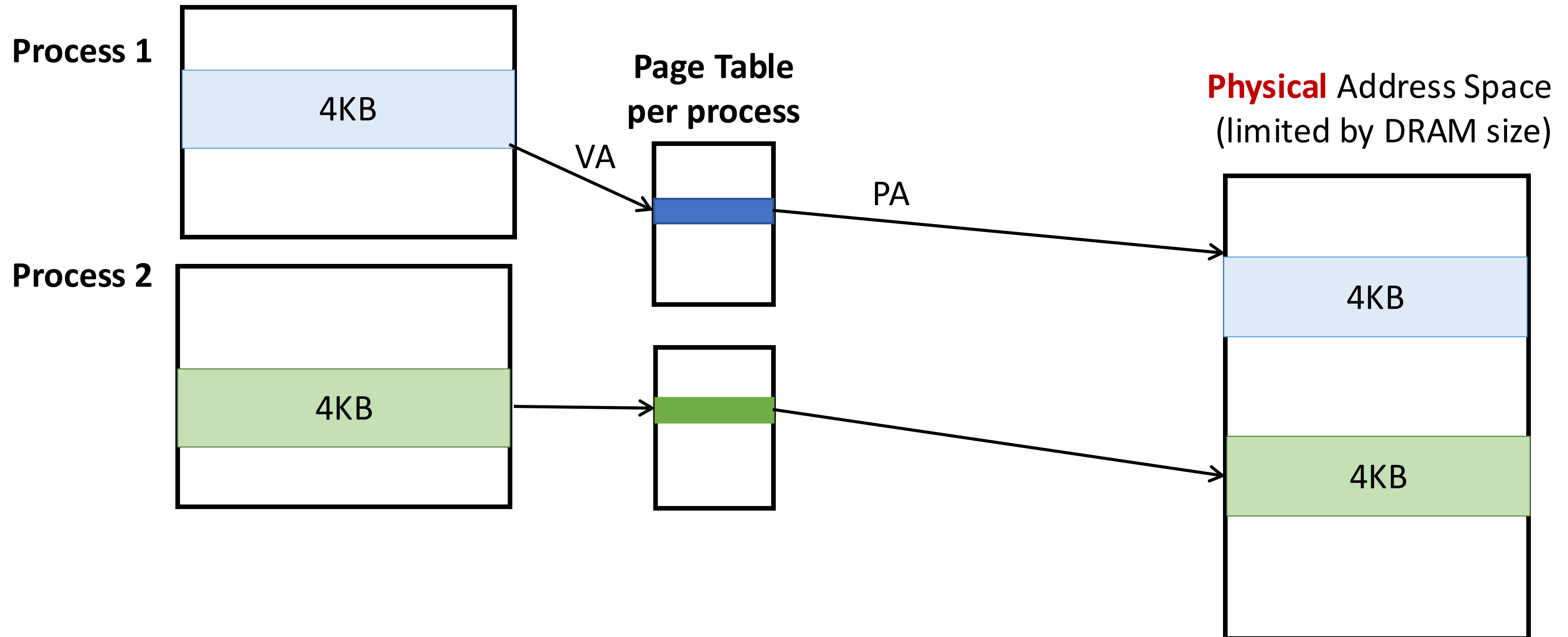


To know more advanced out-of-order (OoO) features, take 6.5900 [6.823]

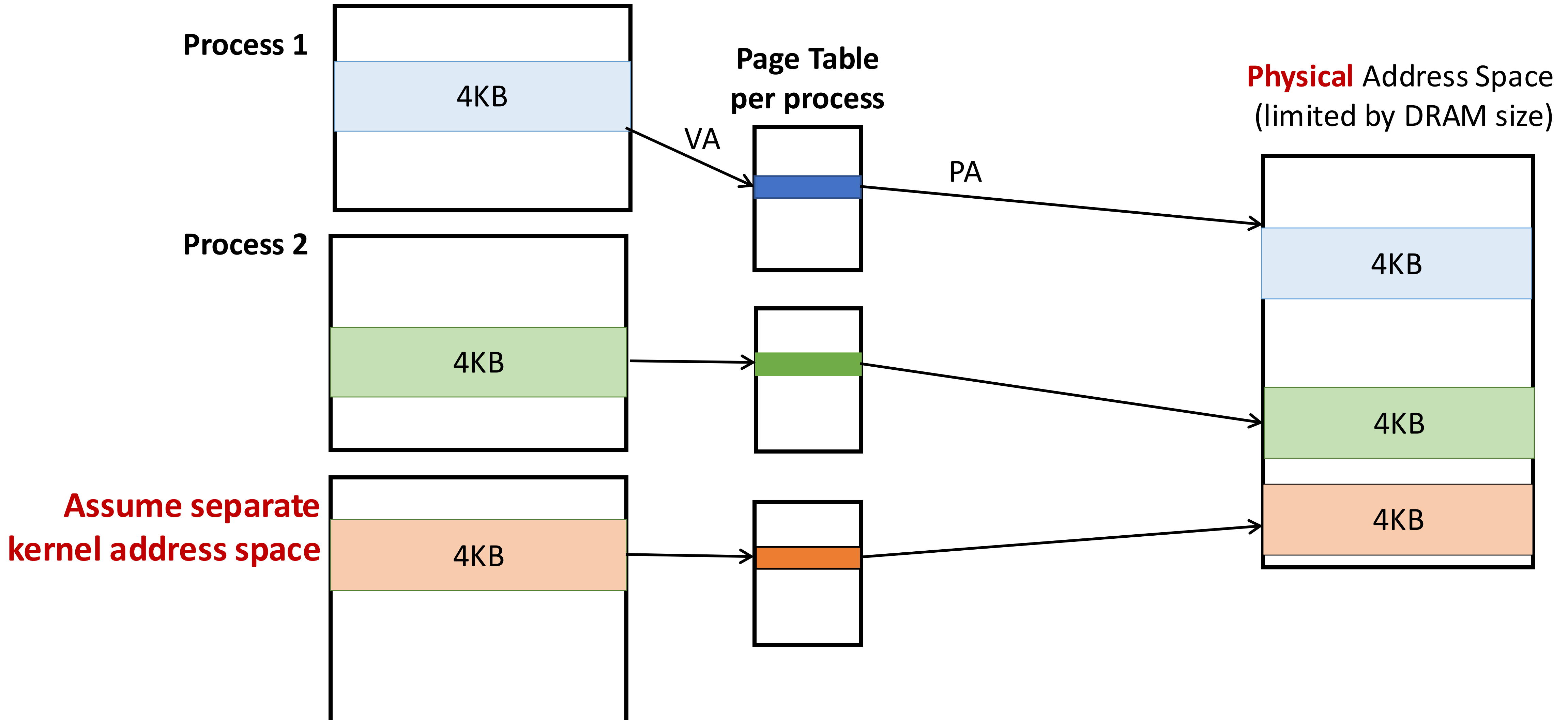
Virtual Memory



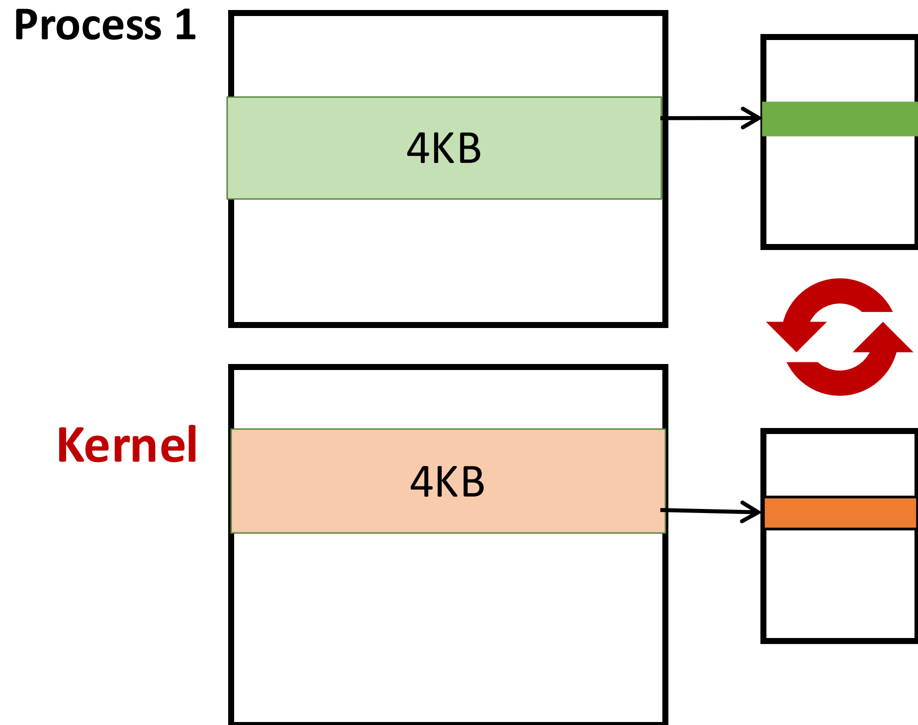
Recap: Page Mapping



Mapping Kernel Pages

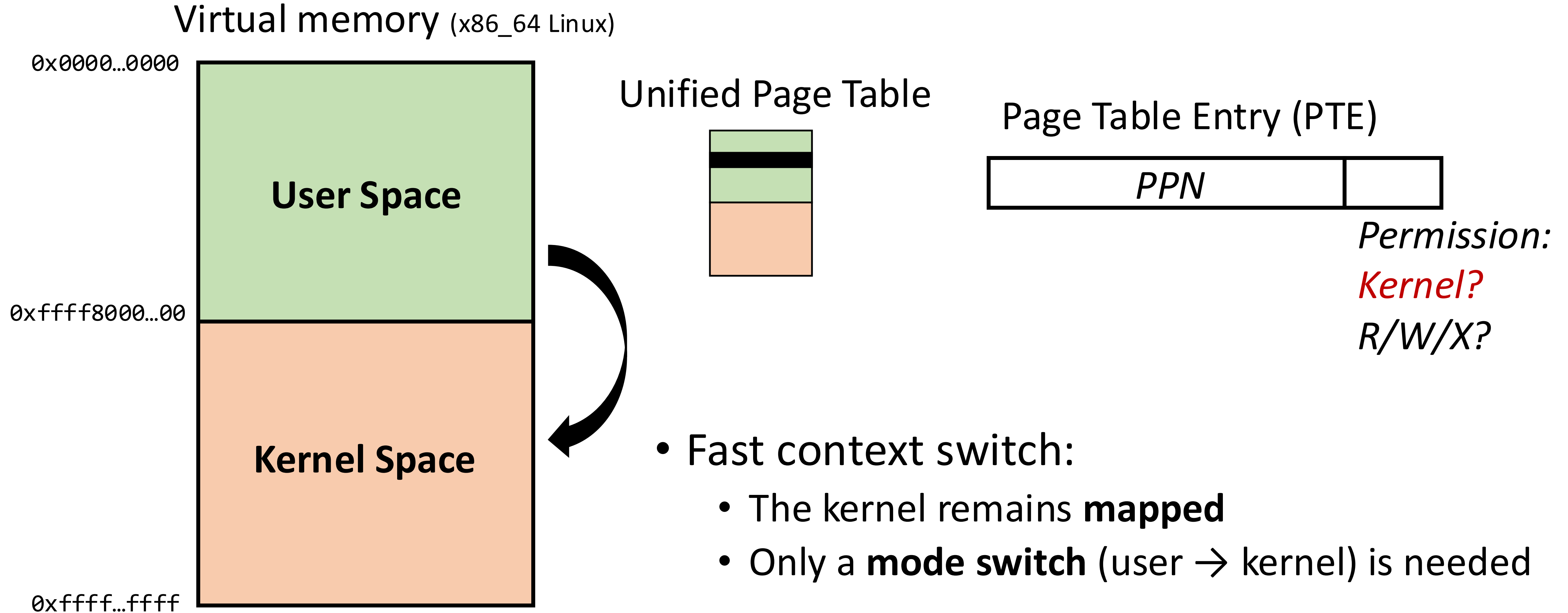


Jumping Between User and Kernel Space



- Context switch overhead:
  - Page table changes introduce perf overhead, e.g., flush TLB in some processors
- And sometimes, we only go to kernel to do some simple things, `getpid()`
- Performance optimization:
  - Map kernel address into user space in a **secure** way, so no need to swap page tables

Map Kernel Pages Into User Space Securely



- Fast context switch:
  - The kernel remains **mapped**
  - Only a **mode switch** (user → kernel) is needed

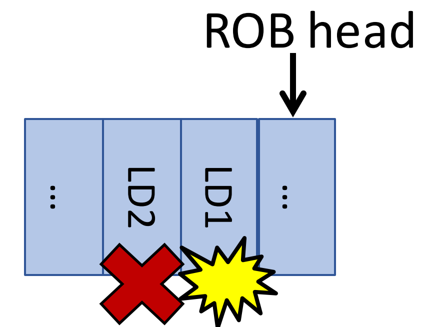
Meltdown

Meltdown

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

Goal: in user space, pick a kernel_address and leak its content

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

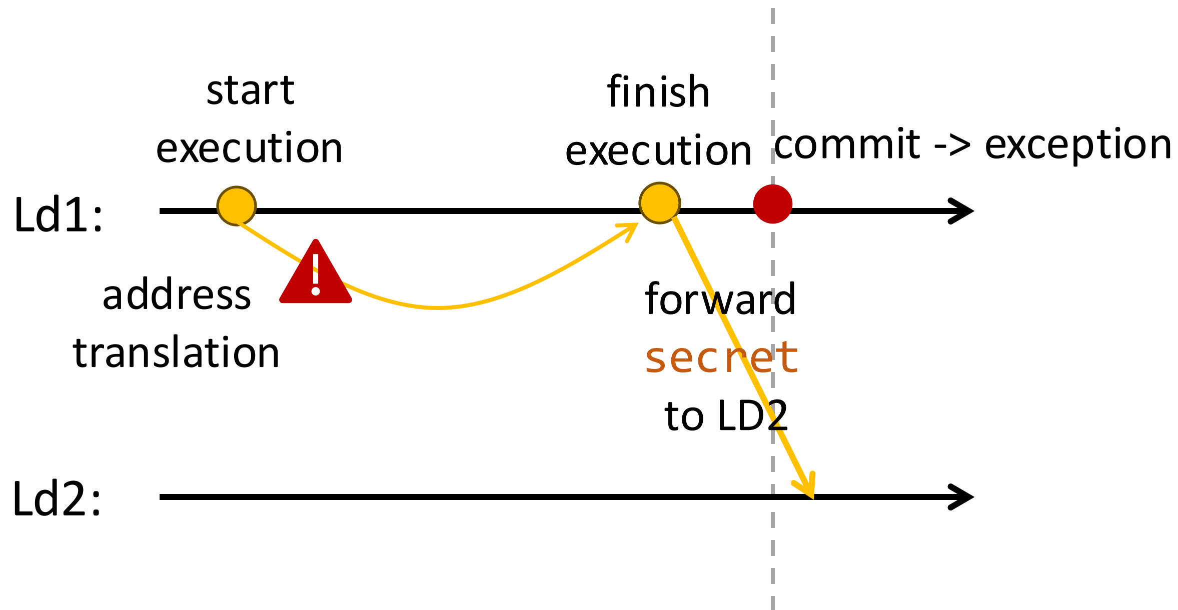


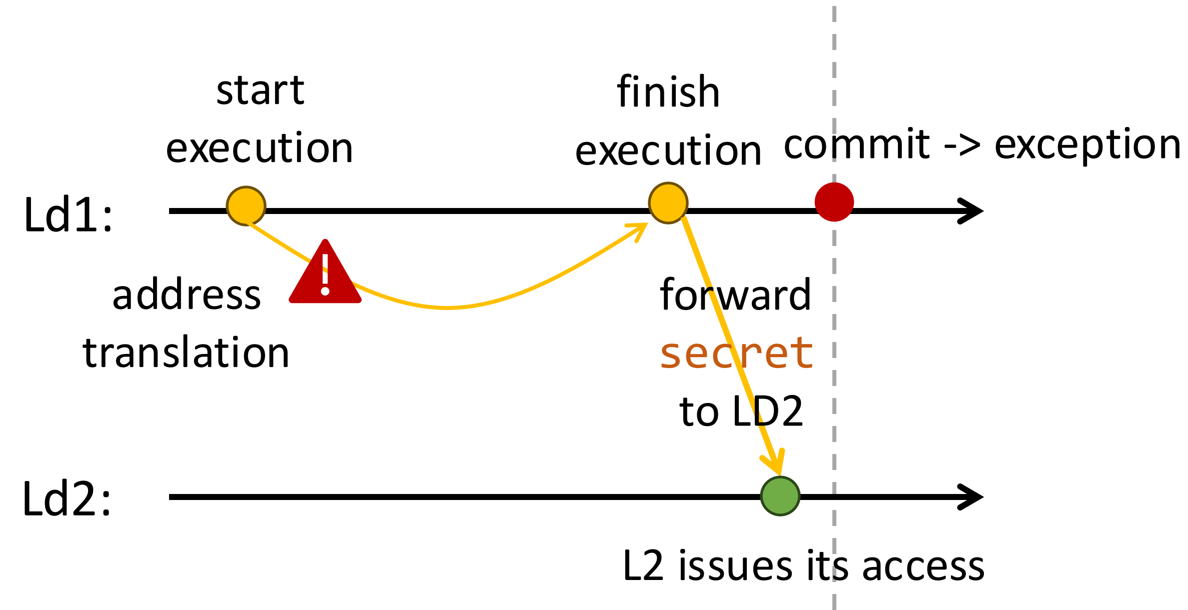
Meltdown Timing

```
……
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`

Why it takes so long for Meltdown to be discovered?

SW optimization:
Map kernel address
in user space

Contract: Memory access goes through page permission check,
and permission violation raises exceptions

HW optimization:
Speculation to delay
exception handling

Meltdown Mitigations

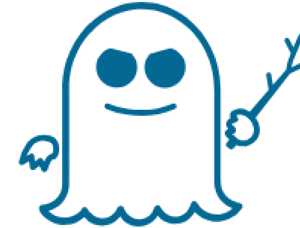
- Stop one of the optimizations should be sufficient
  - SW: Do not let user and kernel share address space (KPTI) -> broken by several groups (e.g., *EntryBleed*)
  - HW: Stall speculation; Register poisoning

```
……
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

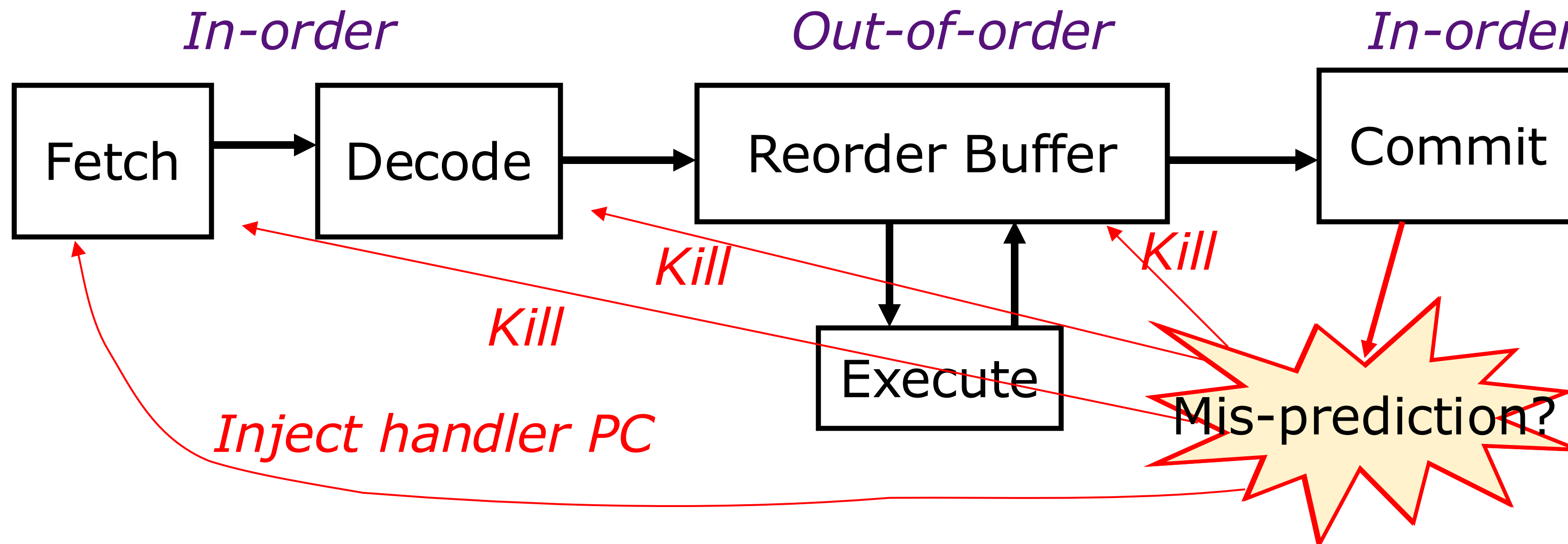
Spectre and its Variants



```
void func(int x){
    //prevent out-of-bound array access
    if (x < array_size) {
        val = array[x]
    }
    return val;
}
```

Branch Prediction

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



Branch Prediction

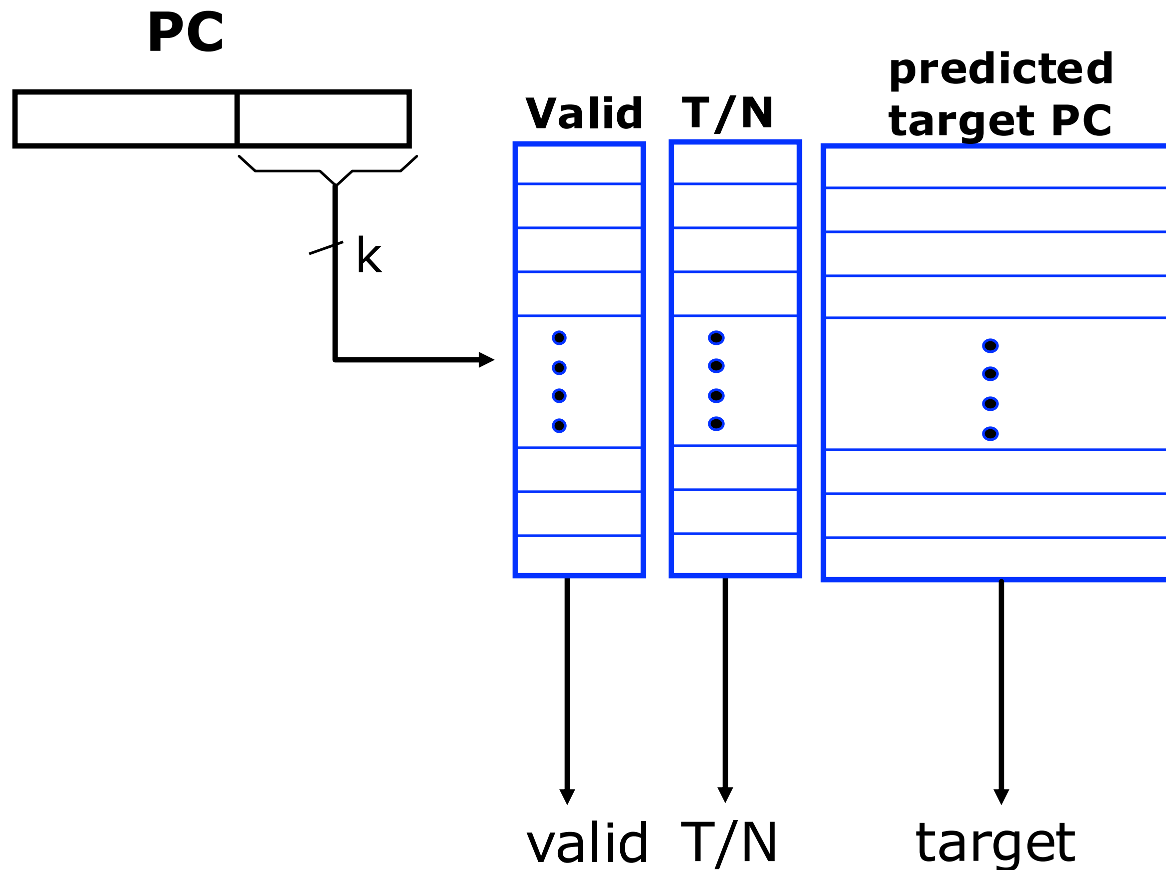
- Naïve approach: PC+4
- More advanced, predict two things:
  - Direction of a conditional branch (whether a branch is taken or not)
    - `blt r1, r2, <label>`

Idea: 1-bit predictor for loop

  - The target address of a branch
    - `jalr <reg>`
    - `ret`

Idea: memorizing branch source and destination pairs

A Simple Branch Predictor Unit (BPU)



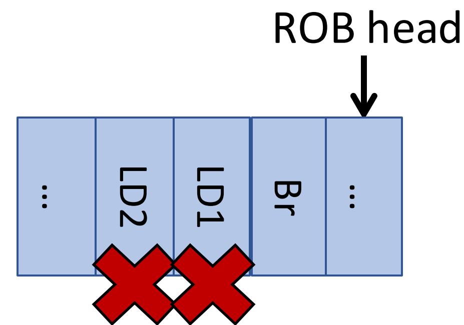
- When branch instruction commits
  - Update the predictor
- In the fetch stage
  - Use the predictor to decide what address to fetch next
- Limited space?
  - Use selected bits in PC to index into the predictor

Spectre V1 – Speculative Out-of-Bound

- Consider code running inside a sandbox

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

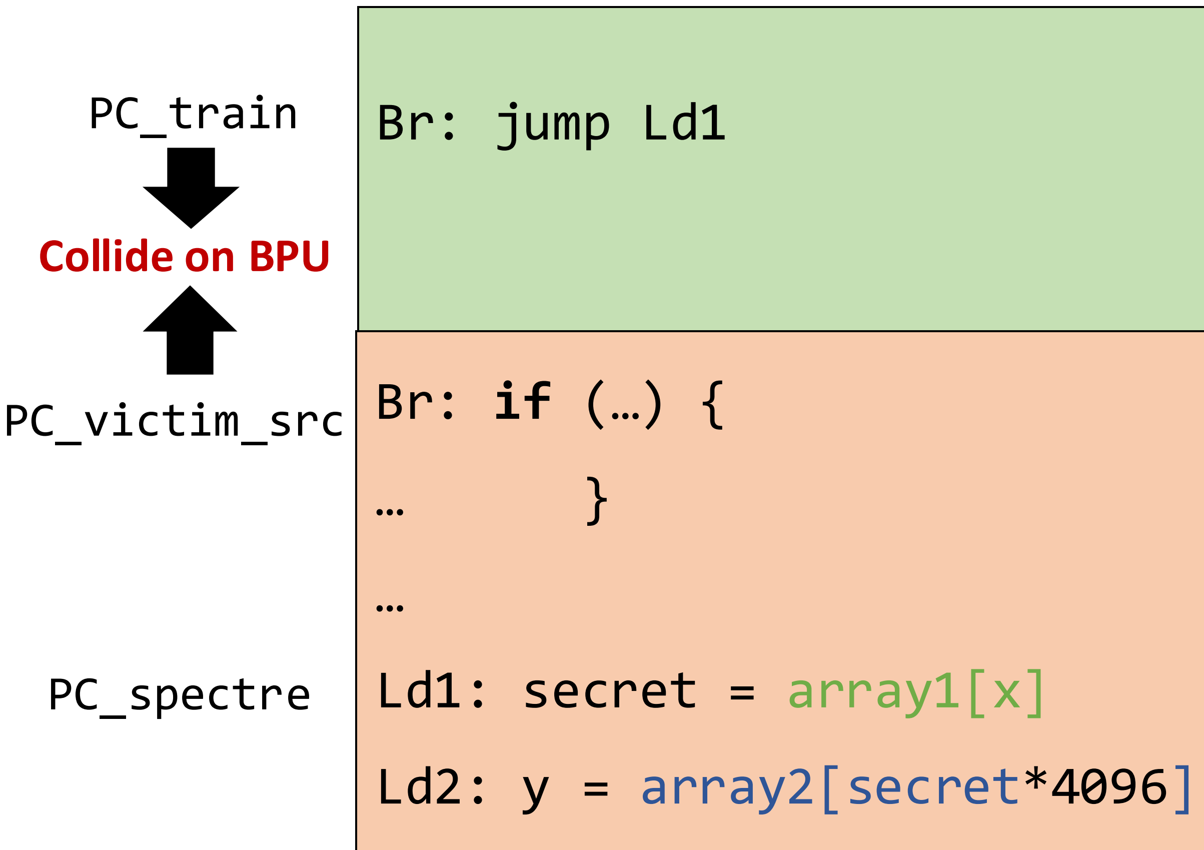
Always malicious?
No. It may be a benign misprediction.
We do not consider Spectre as a bug.

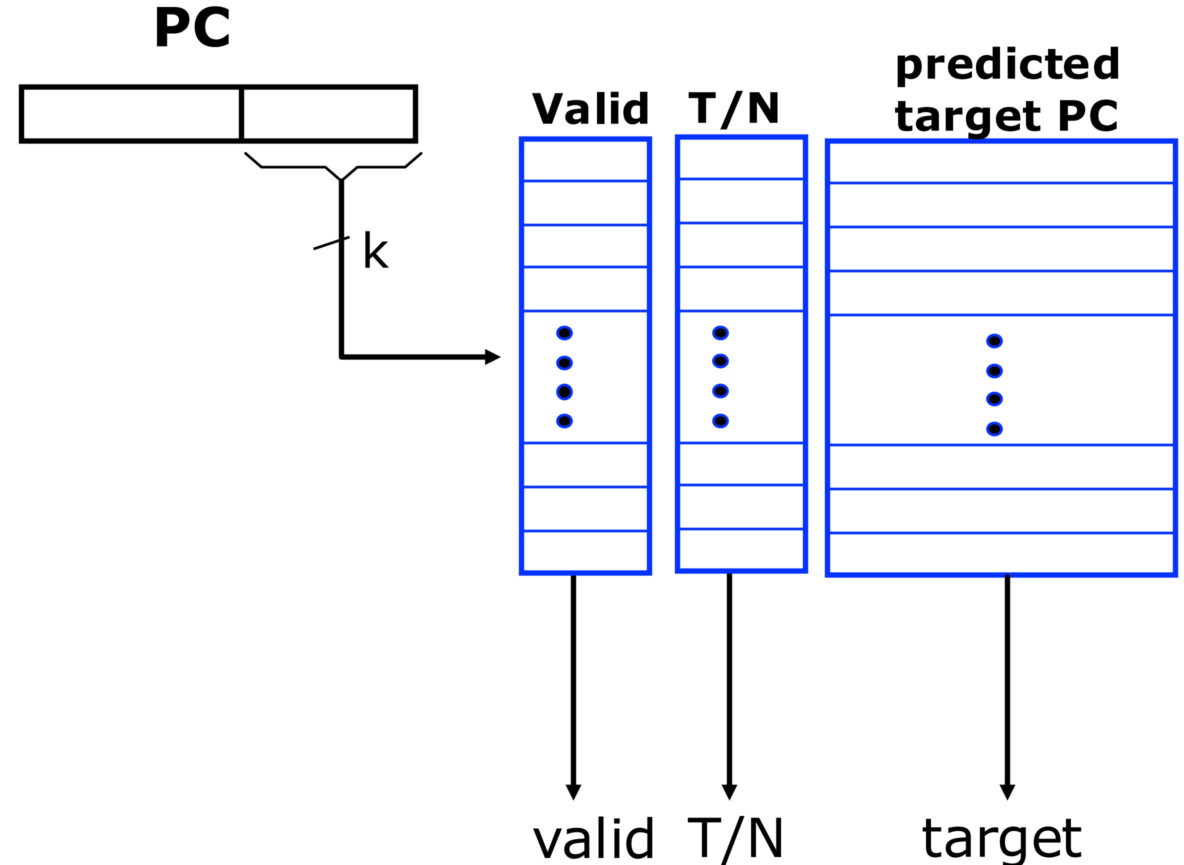


Attacker to read arbitrary memory:
1. Setup: Train branch predictor
2. Transmit: Trigger branch misprediction; *&array1[x]* maps to some desired kernel address
3. Receive: Attacker probes cache to infer which line of *array2* was fetched

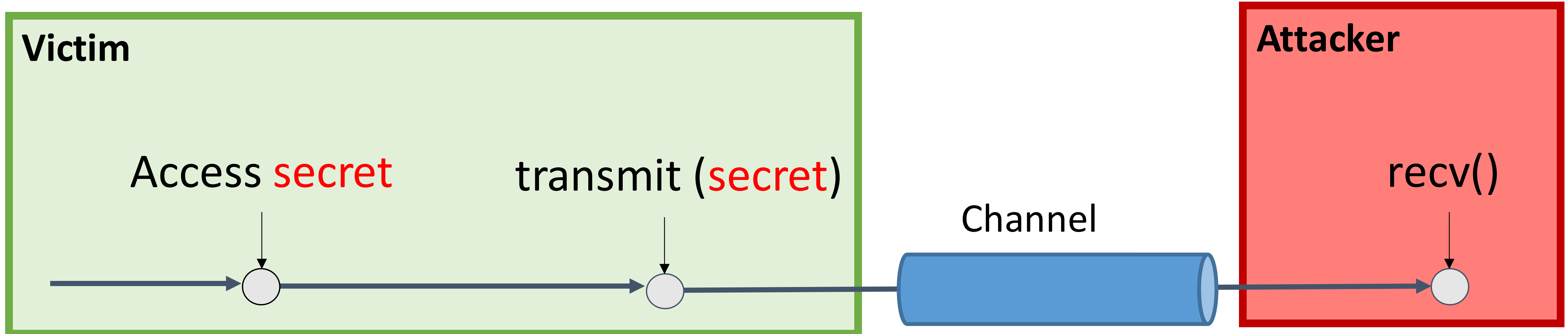
Spectre V2 – Speculative JOP

1. Insert <PC_train, PC_spectre>
2. Trigger PC_victim_src
3. Speculative execute PC_spectre





General Attack Schema



DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. Kiriansky et al. MICRO'18

Apply the General Attack Scheme

The RSA Square-and-Multiply Exponentiation example.

Attackers aim to leak e

```
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
```

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

Apply the General Attack Scheme

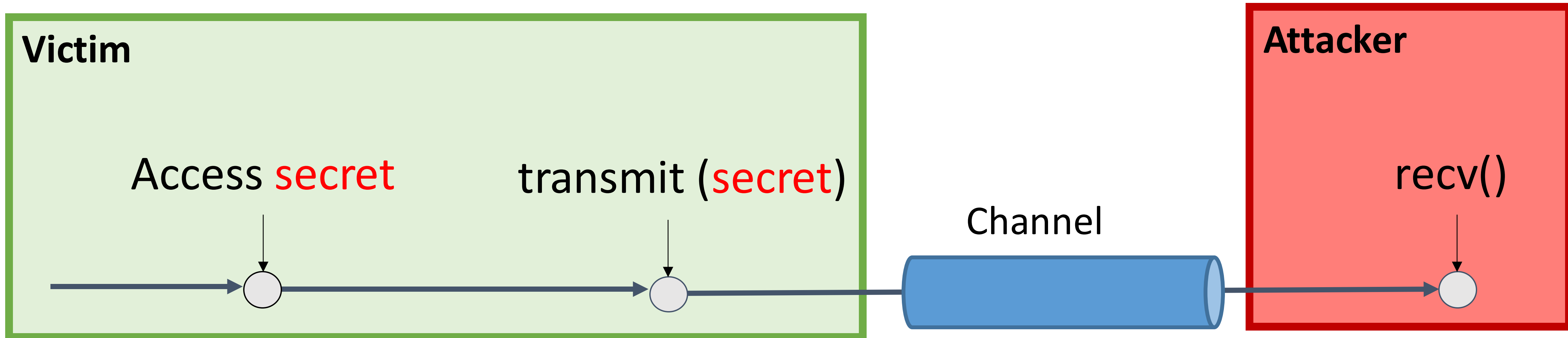
```
……
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
```

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

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

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

General Attack Schema



- Traditional (non-transient) attacks — Hard to fix
  - Gadget preexist in victim space. Leak data in-use
- Transient attacks: Gadget constructs via speculation. Leak data-at-rest
  - Meltdown = transient execution + deferred exception handling — "Easy" to fix
  - Spectre = transient execution on wrong paths — Hard to fix

Next: Cache Attack Recitation