Transient Execution Attacks

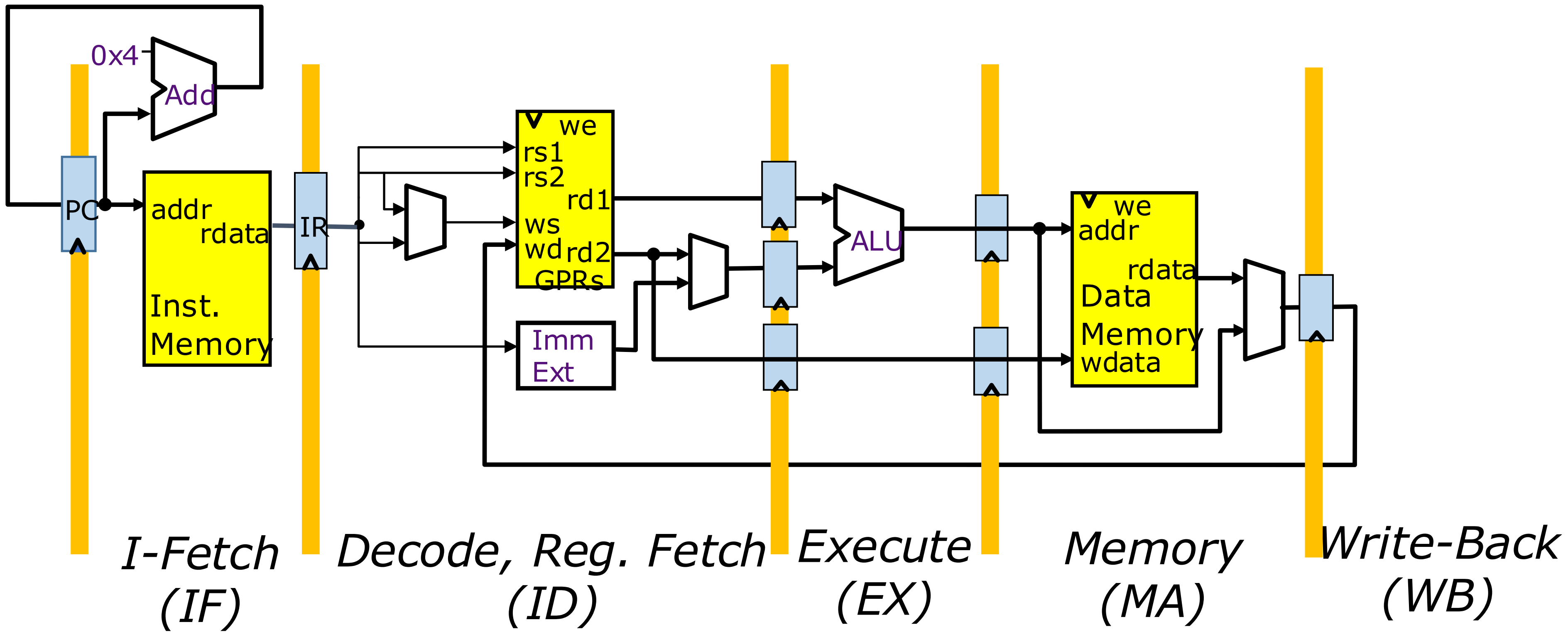
Mengjia Yan

Spring 2026

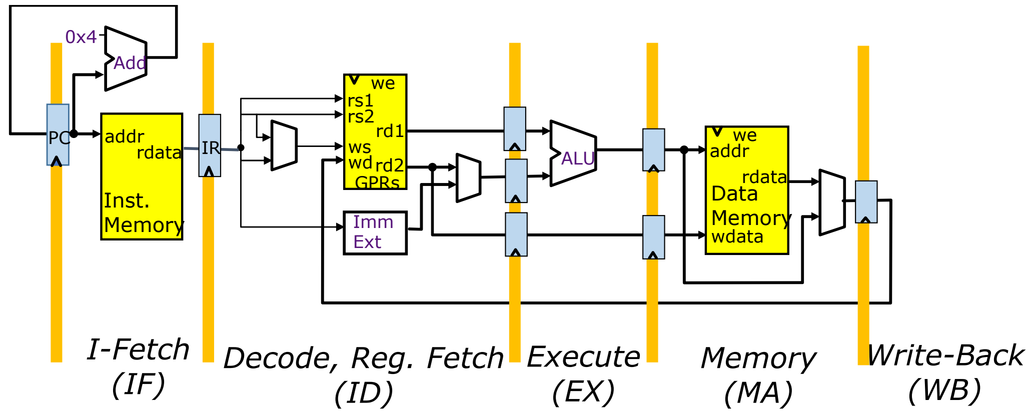
Outline

- 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
……
```

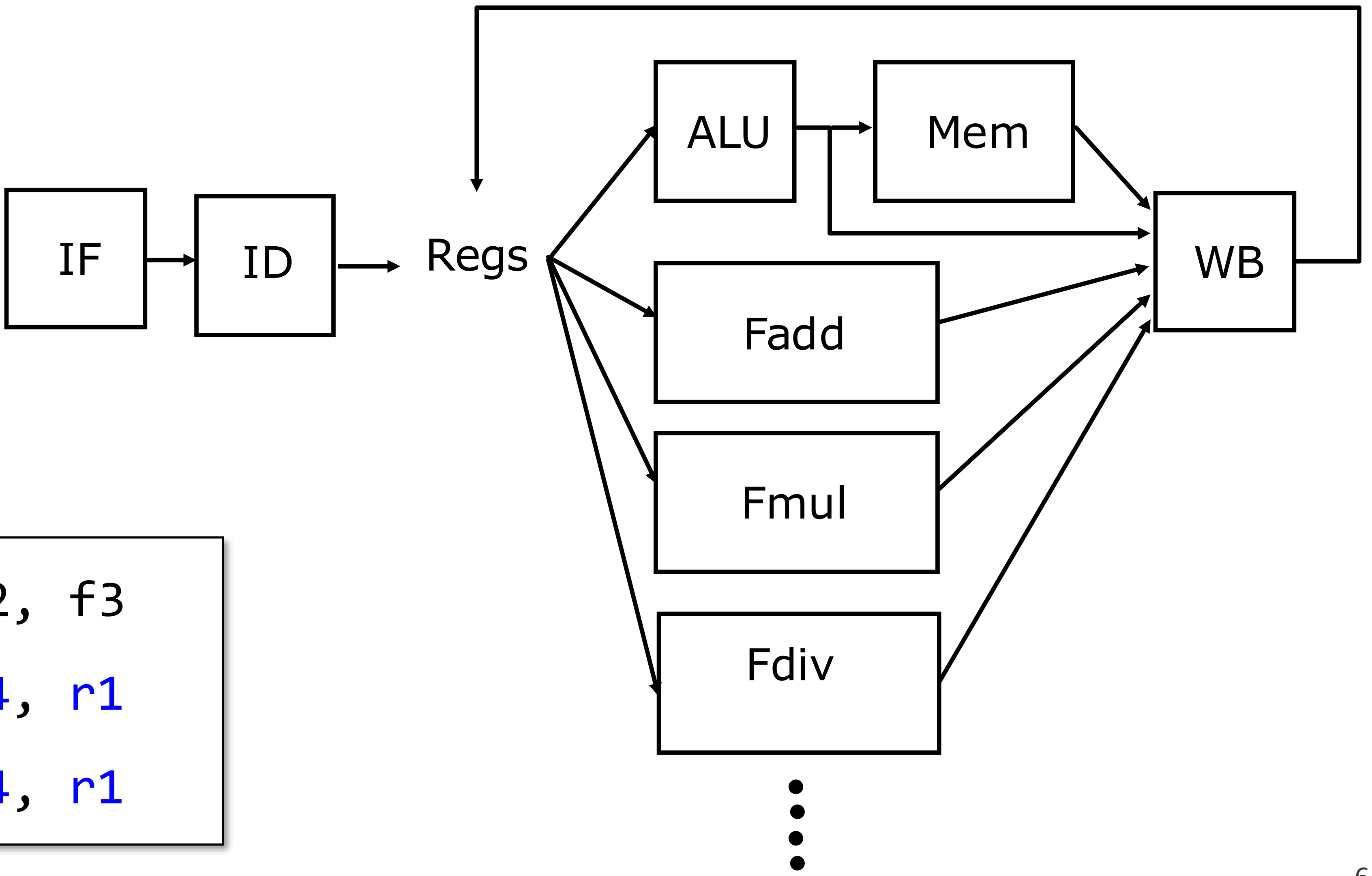
Instruction-Level Parallelism (ILP)

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

Example #2:

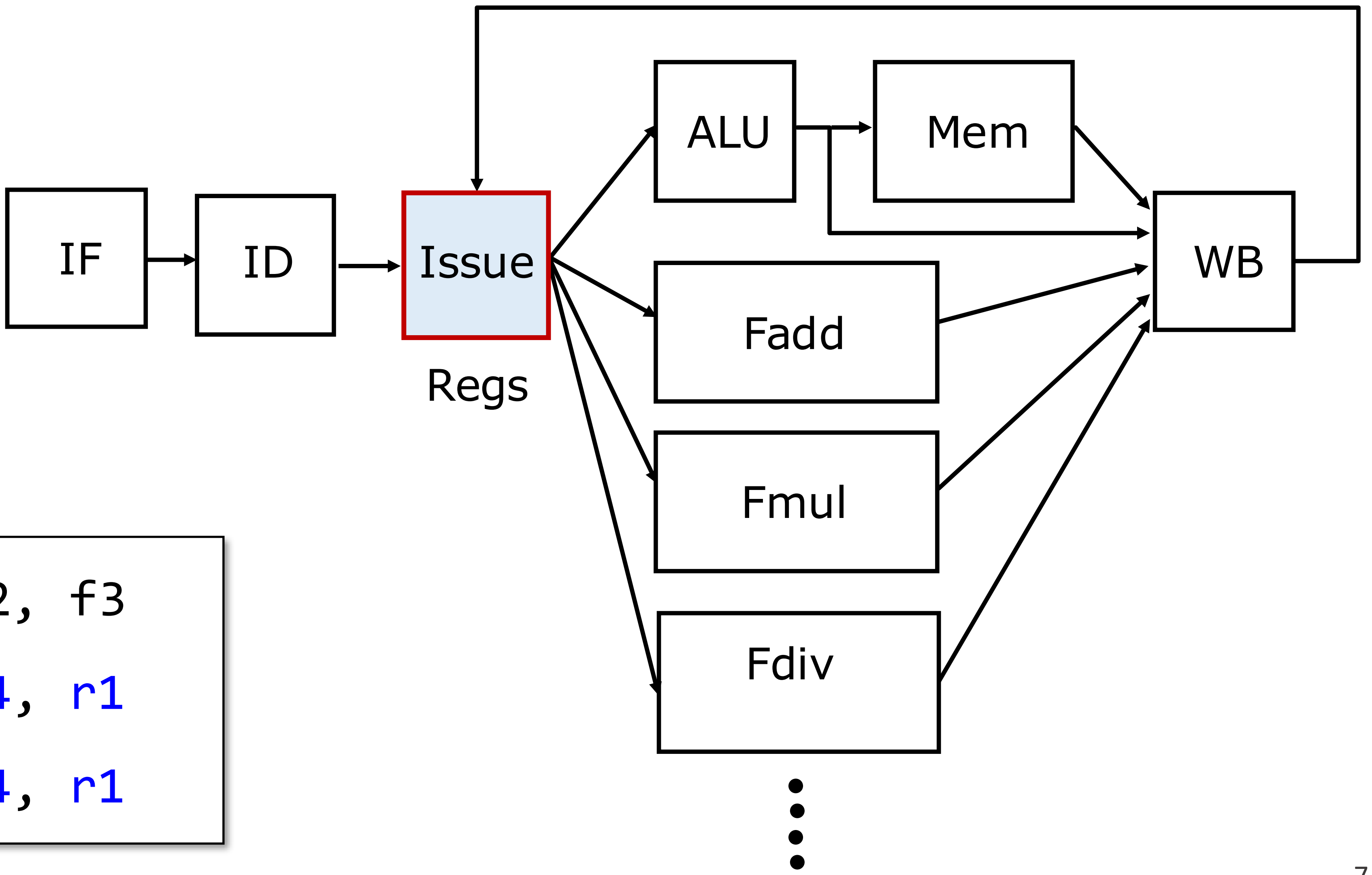
```
LD  r3, 0(r2)    ; 1-100 cycles
ADD r4, r4, r1   ; 1 cycle -> repeat 10 times
……
```

Technique #1: Add More Functional Units



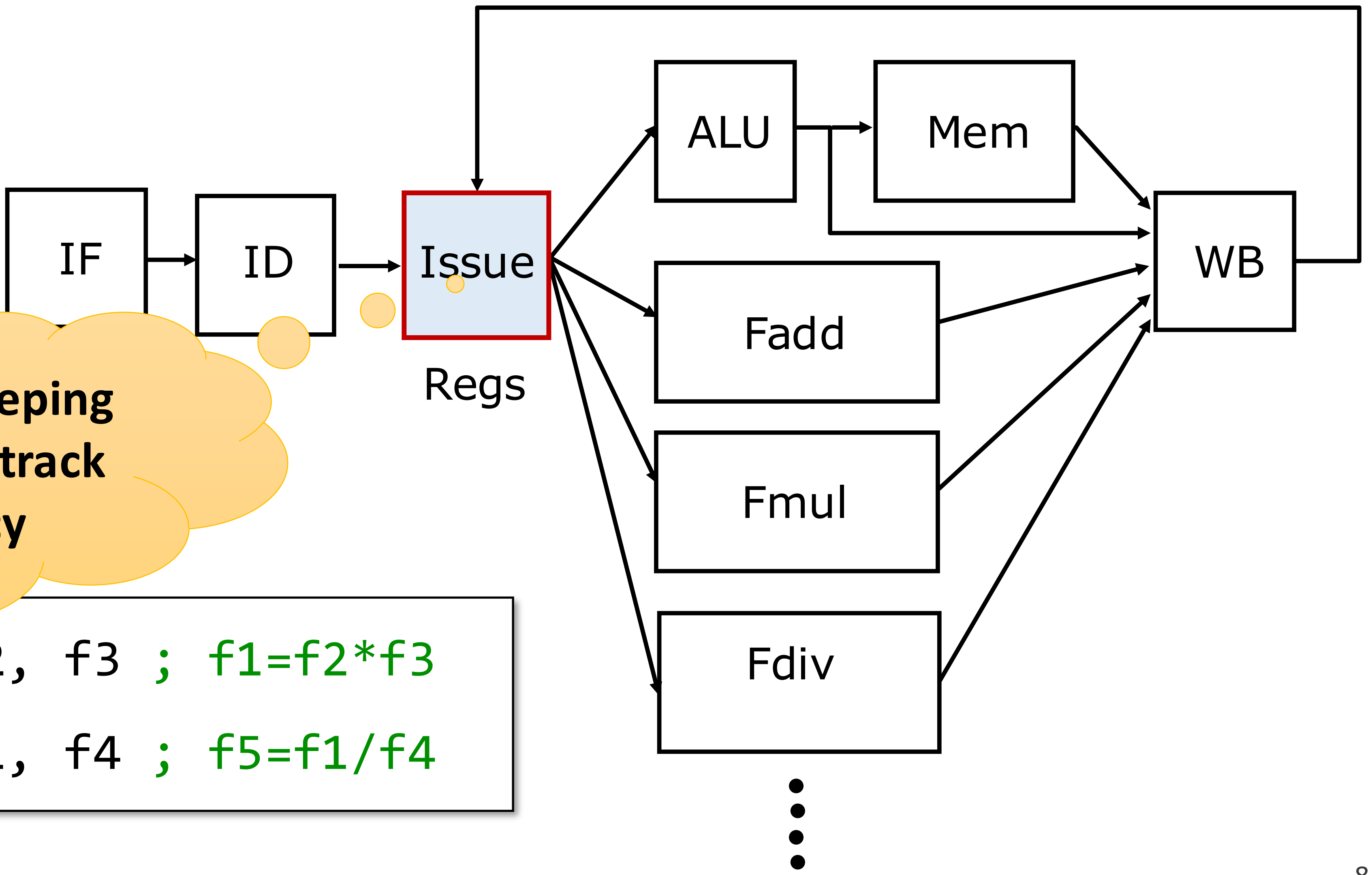
```
1: FMUL f1, f2, f3
2: ADD  r4, r4, r1
3: ADD  r4, r4, r1
```

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



Need a bookkeeping mechanism to track dependency

```
1: FMUL f1, f2, f3 ; f1=f2*f3
2: FDIV f5, f1, f4 ; f5=f1/f4
```

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
```

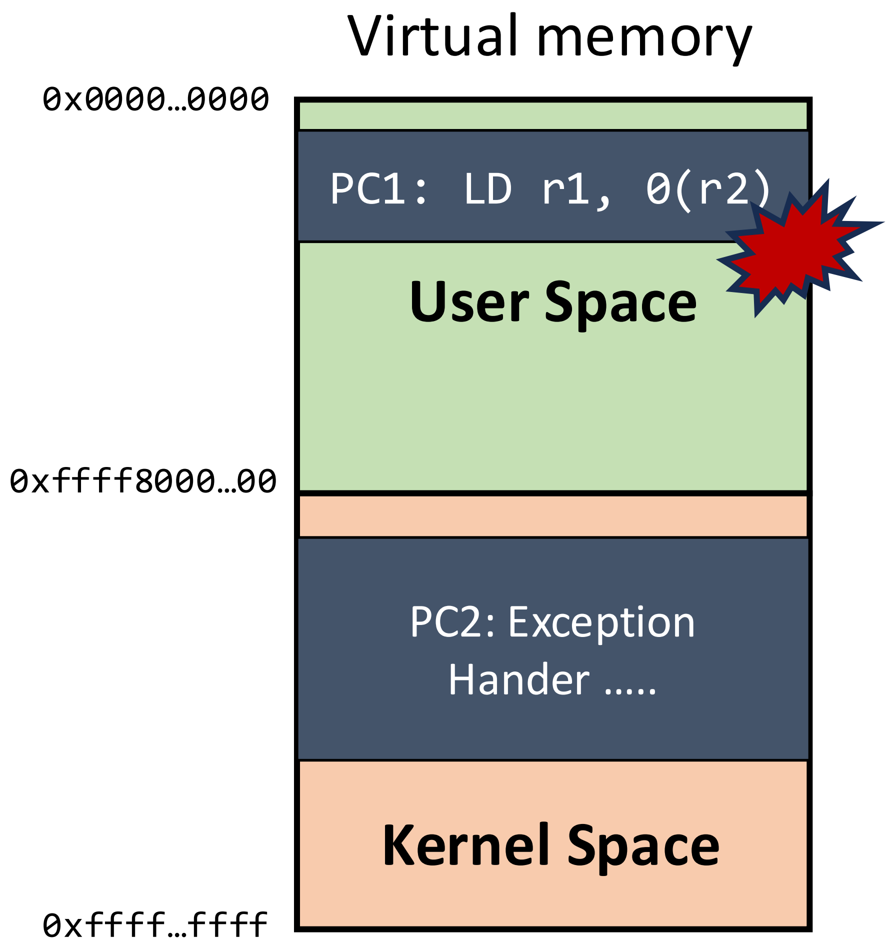
Technique #2: Scoreboard

- Algorithm summary: Upon issue an instruction, we 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?

Simplified Exception Handling (RISCV)



`mtvec`	Machine Trap-Vector Base-Address Register
`mepc`	Machine Exception Program Counter Register

Timeline:

1. Set mtvec to PC2 upon booting Linux
2. PC1 triggers an exception
3. Write PC1 to mepc, save current registers
4. Jump to mtvec (PC2) and execute exception hander
5. Finish handler code, restore registers, and jump back to mepc (PC1)

Exception in OoO Processors: Example #1

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

What will happen after we return from exception handling?

	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**		

Need to delay the WriteBack

Exception in OoO Processors: Example #2

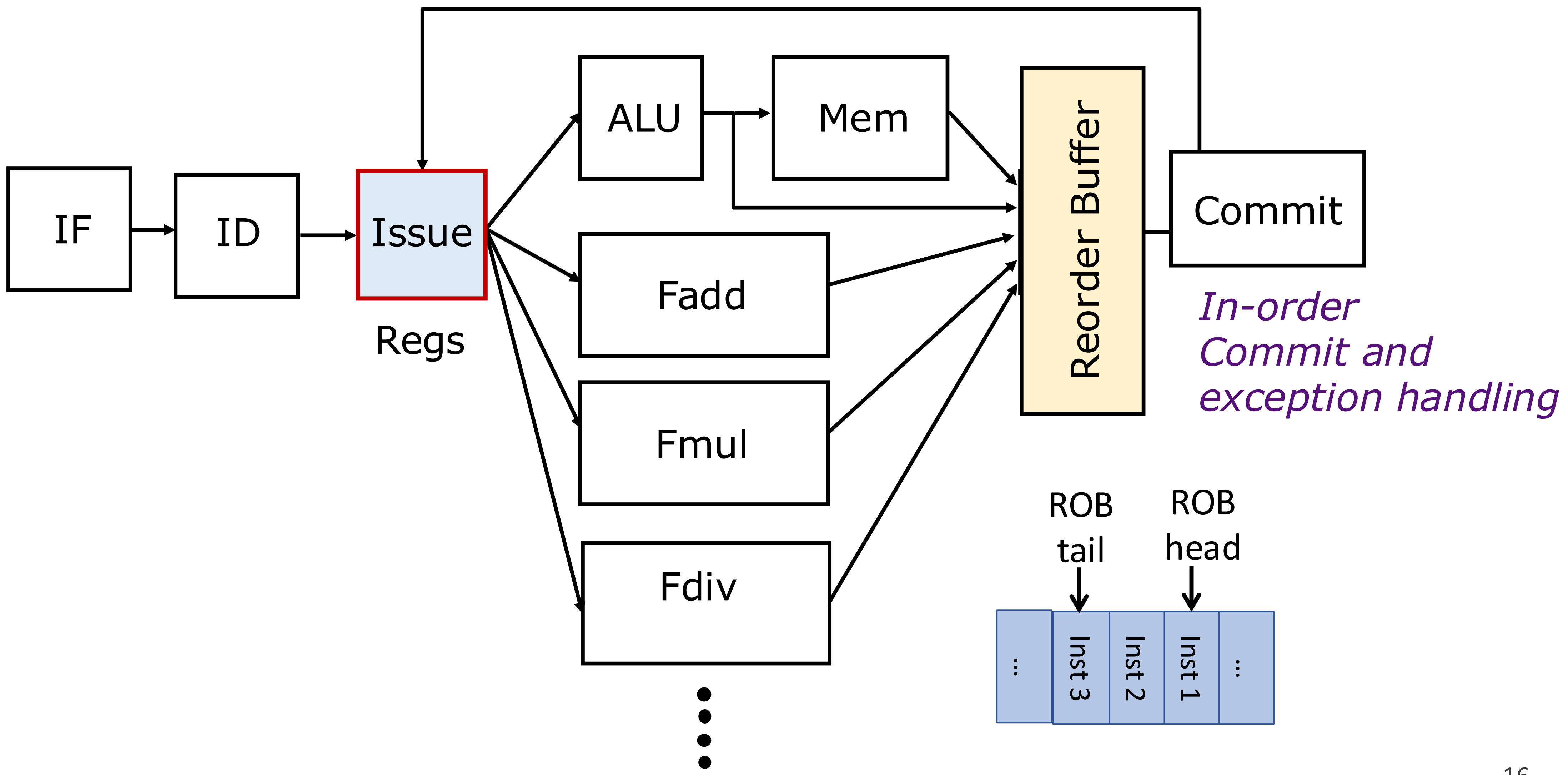
```
1: FMUL f1, f2, f3  ; 10 cycles
2: LD  r3, 0(r2)     ; Exception in 1 cyc
```

What will happen if we jump to exception handler at **`t7`**?

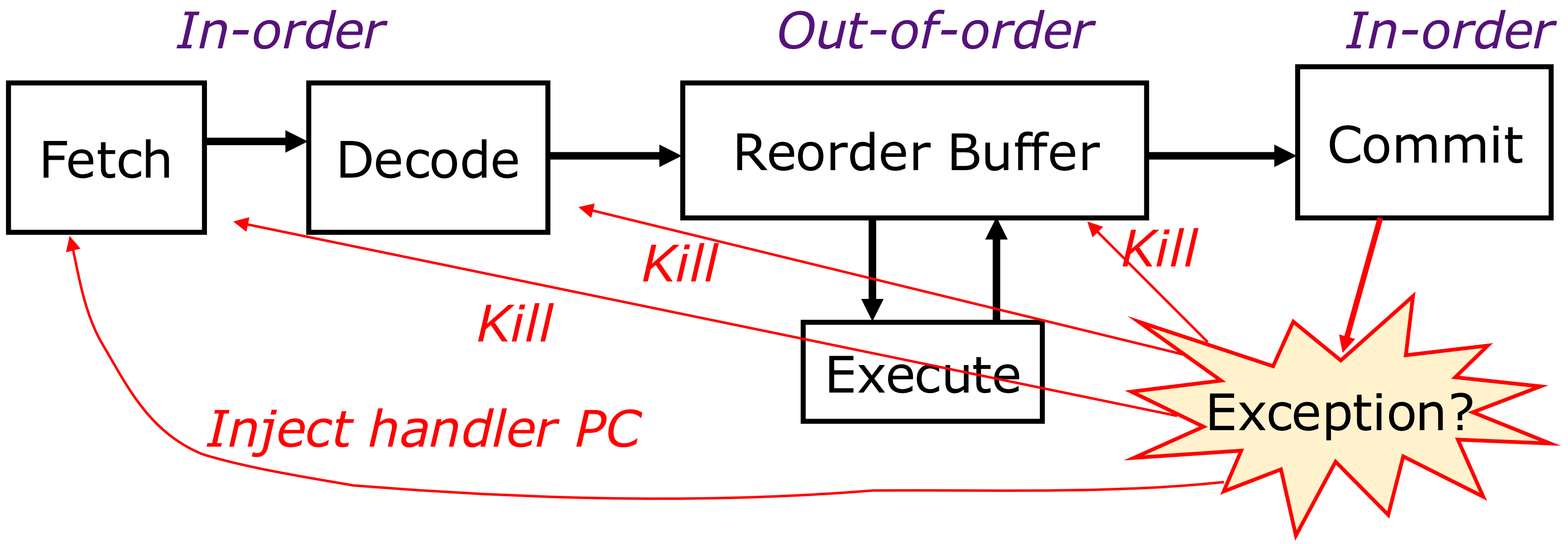
	1	2	3	4	5	6	7 ●	8
1: FMUL	IF	ID	Issue	FMUL	FMUL	FMUL	FMUL ●	…
2: LD		IF	ID	Issue	ALU	Mem	**Exception** ●	

Need to delay the Exception Handling

Technique #3: In-order Commit



Another Way to Draw It



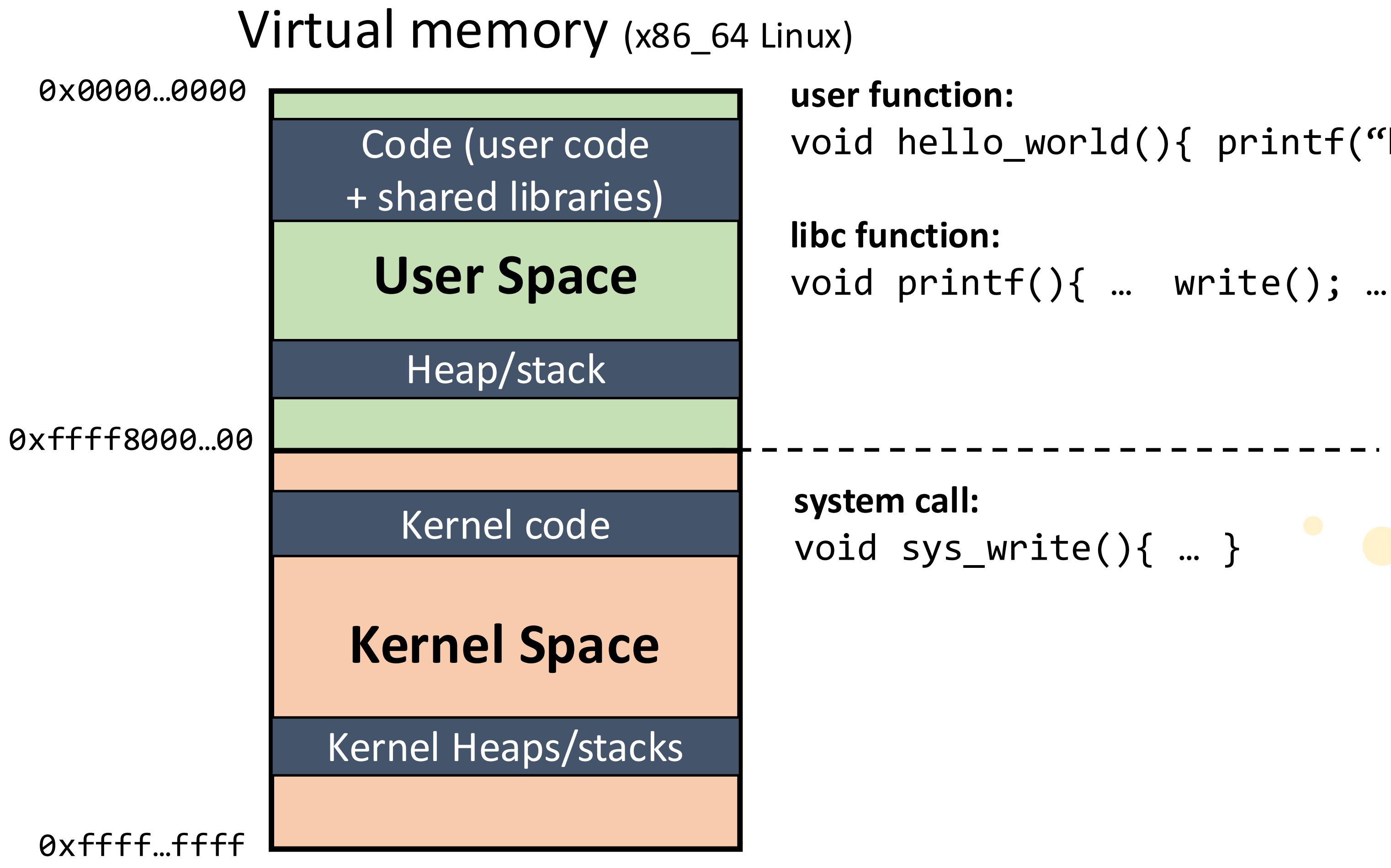
To know more advanced out-of-order (OoO) features, take 6.5900 [6.823]

Meltdown

Leak arbitrary kernel data from user space

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

Virtual Memory



user function:

```
void hello_world(){ printf(“hello word!”); }
```

libc function:

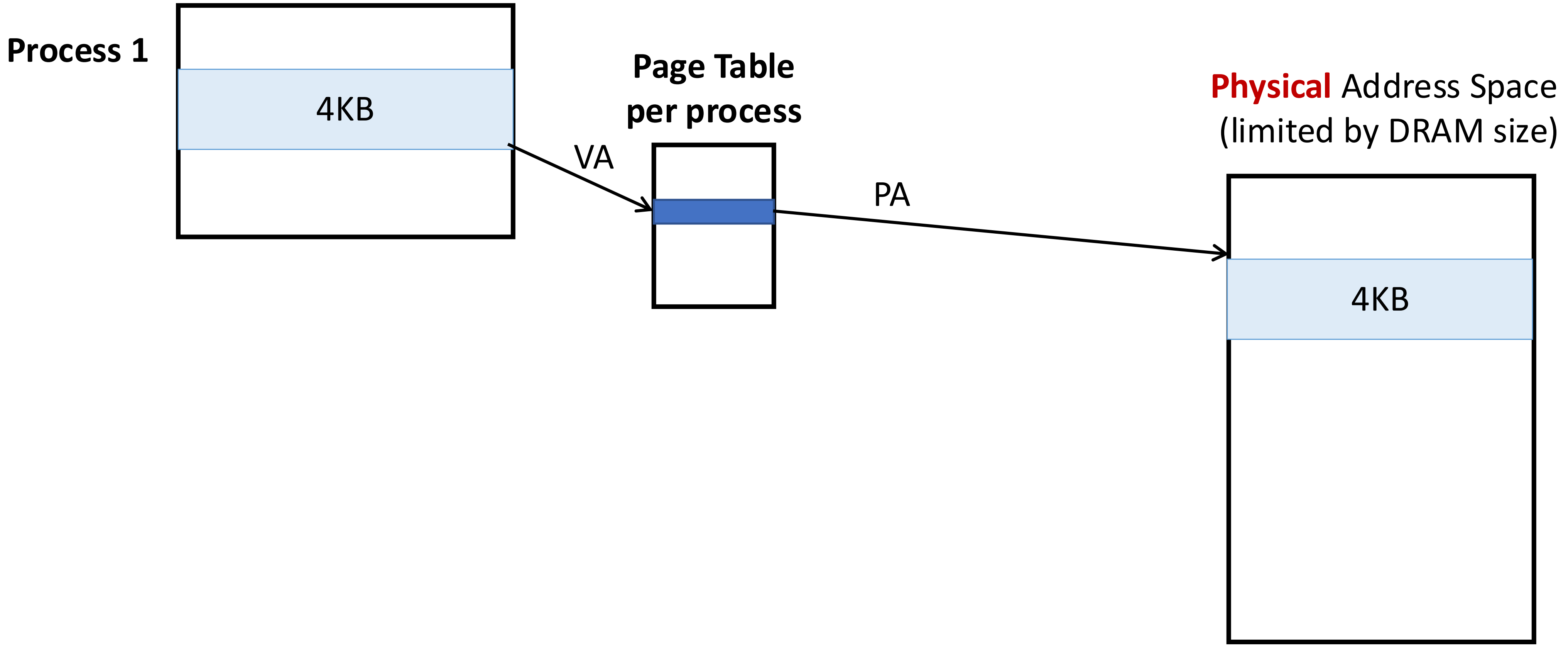
```
void printf(){ …  write(); … }
```

system call:

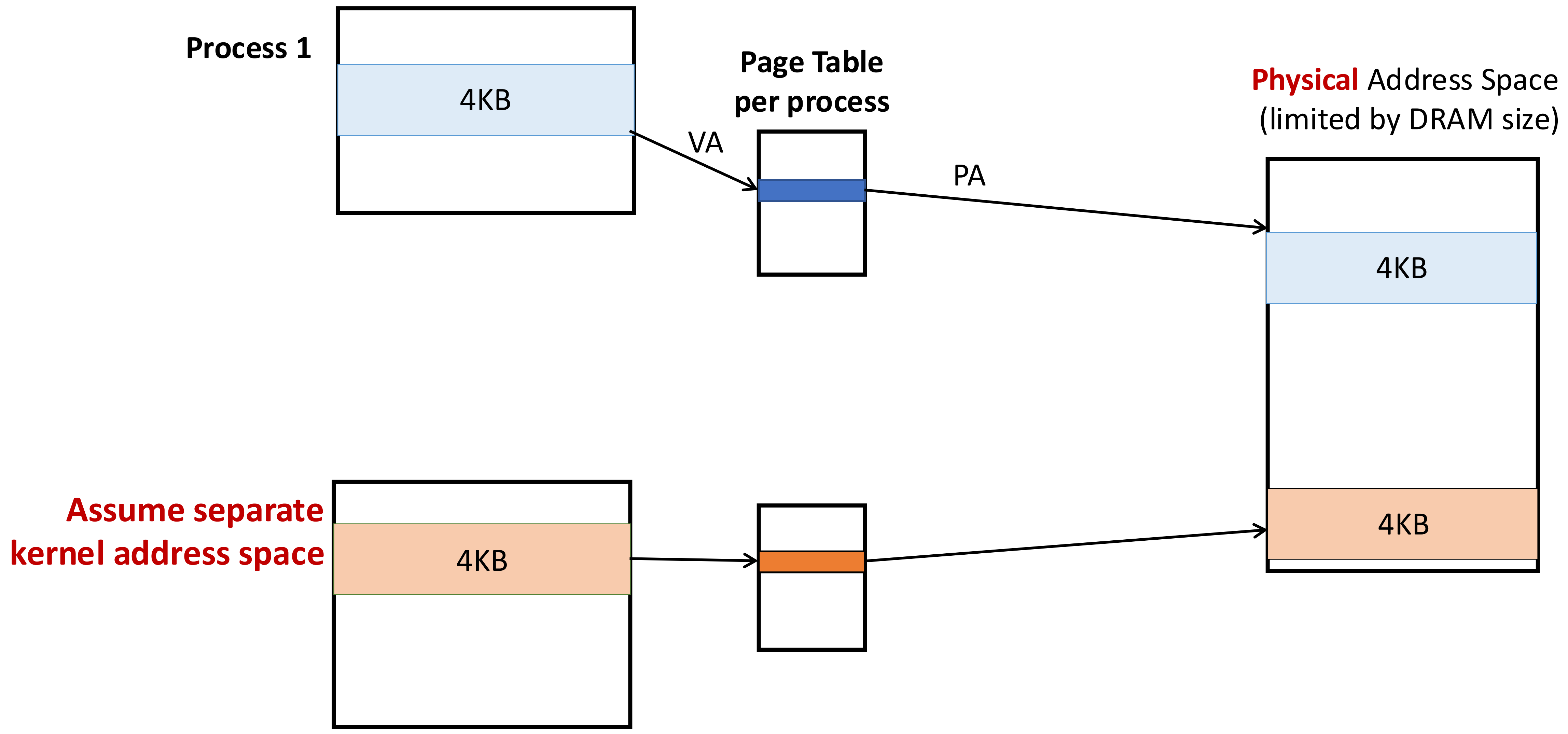
```
void sys_write(){ … }
```

Why map kernel address into user address space?

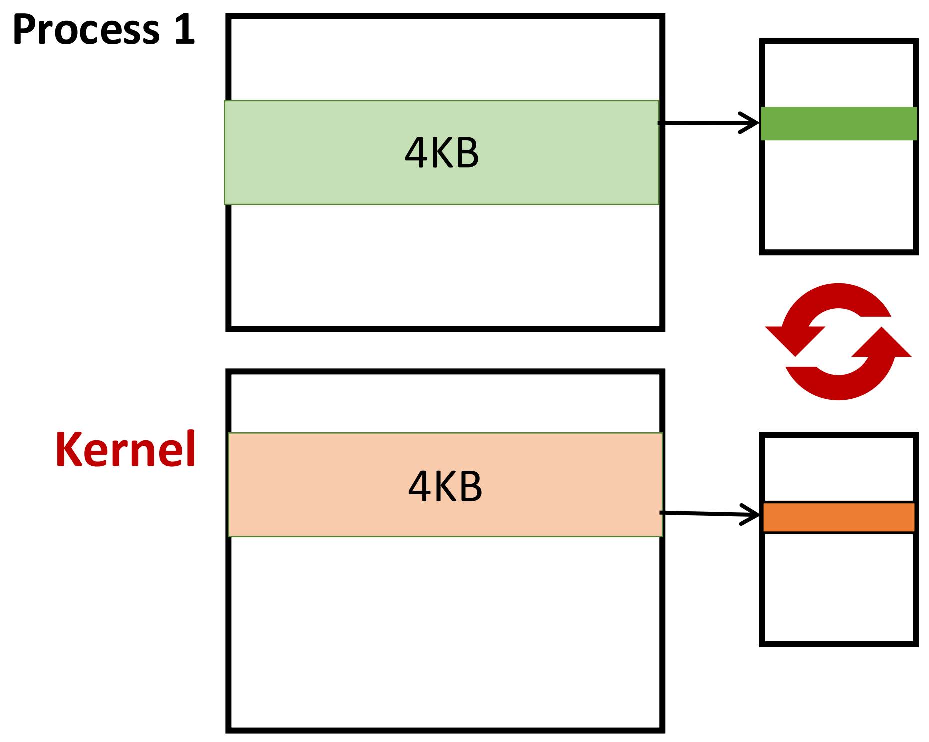
Recap: Page Mapping



Mapping Kernel Pages

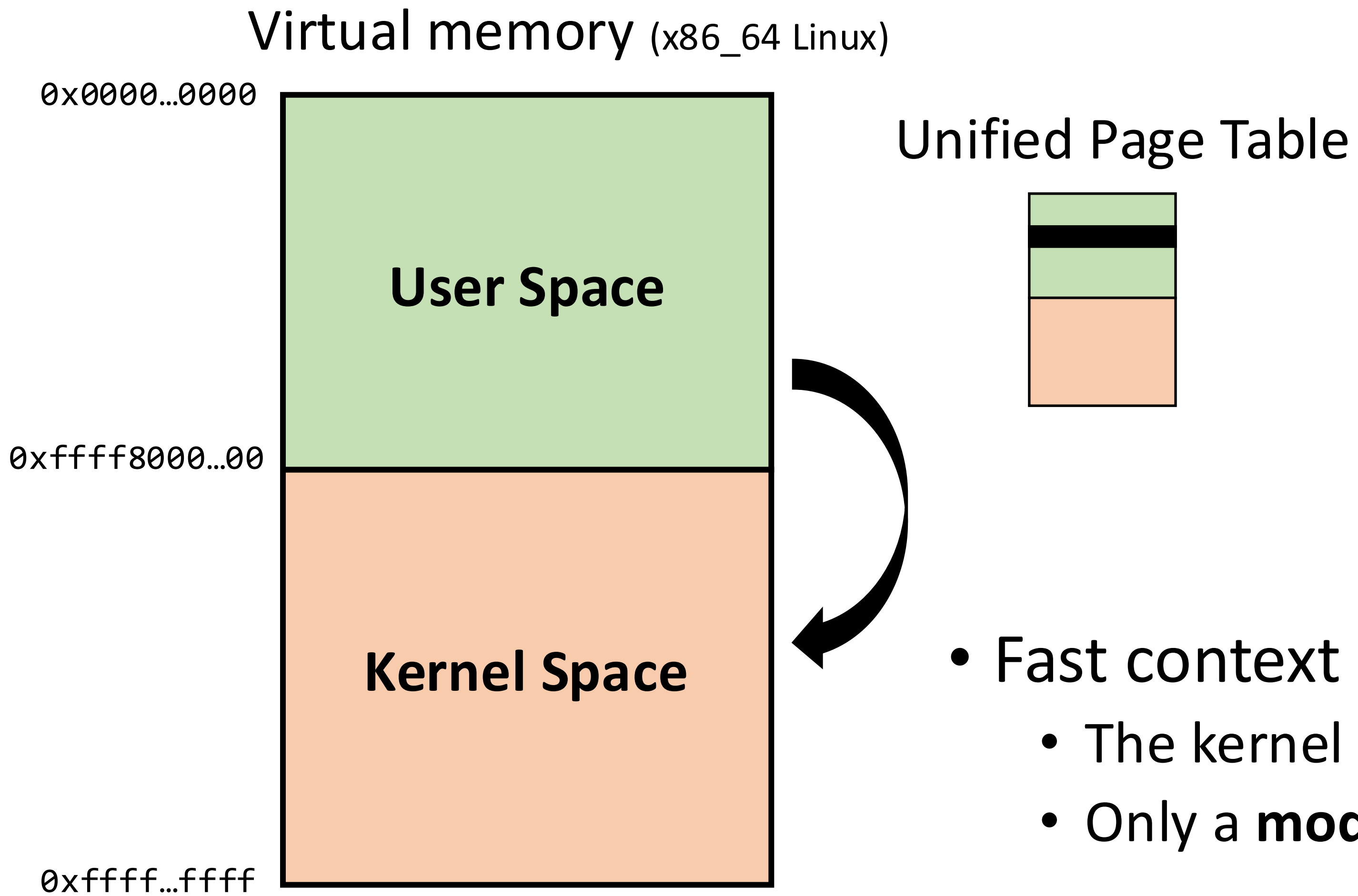


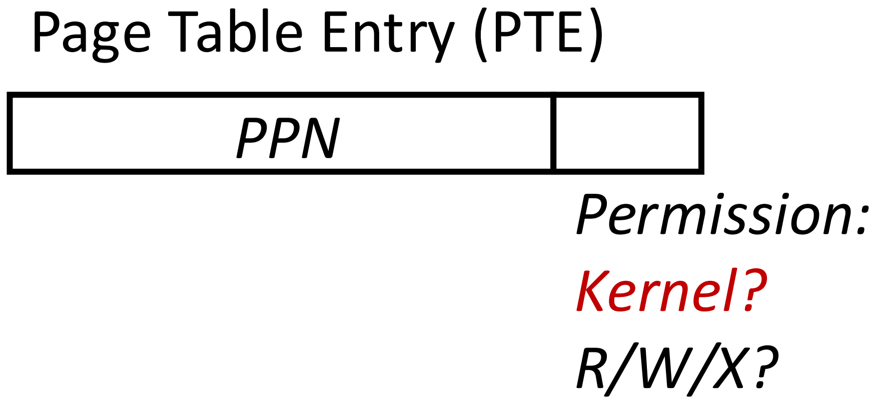
Jumping Between User and Kernel Space



- 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 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

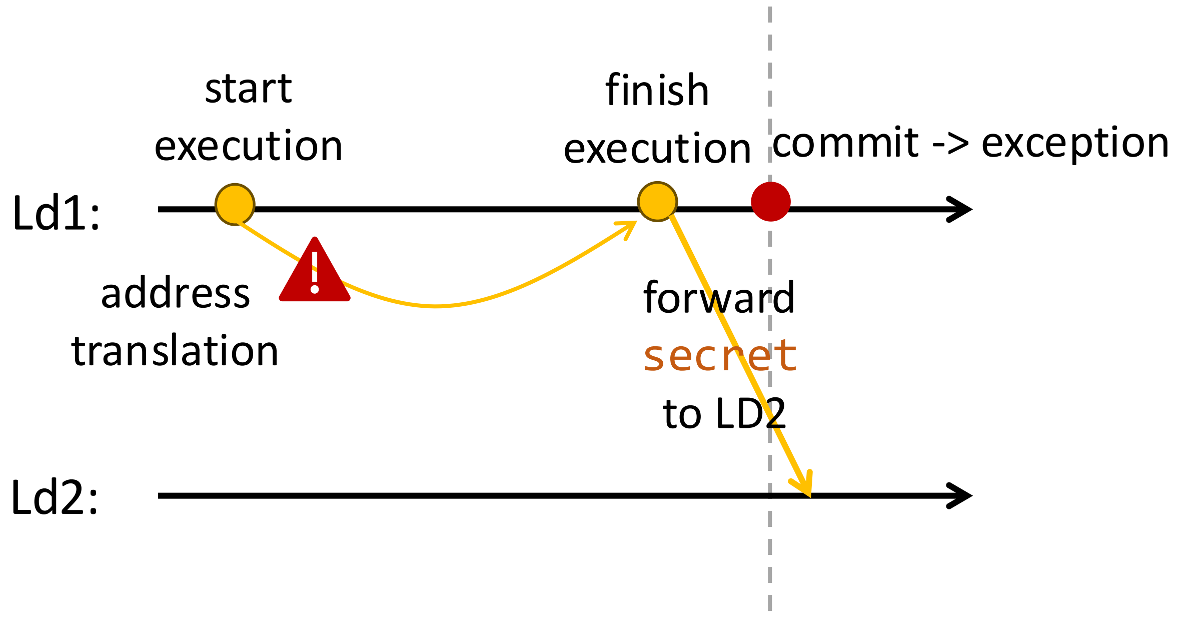
```
……
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
```

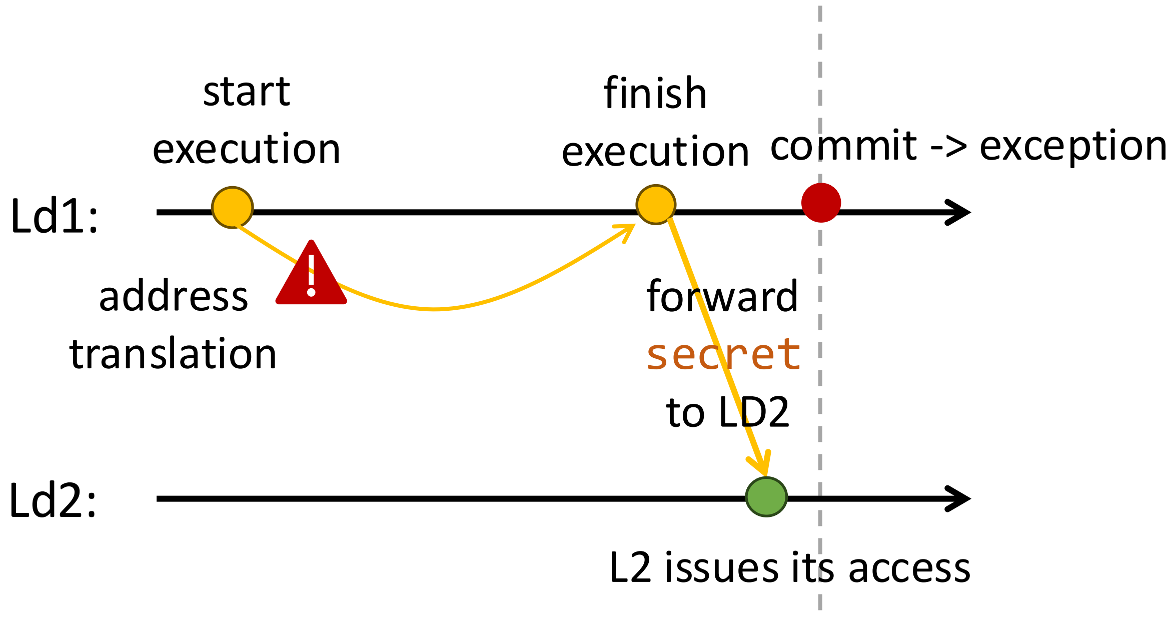
Meltdown Timing

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

Case 1: Attack fails. 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?

Software **Hardware**

SW optimization:
Map kernel address
into 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)
  - HW: Stall speculation; Register poisoning

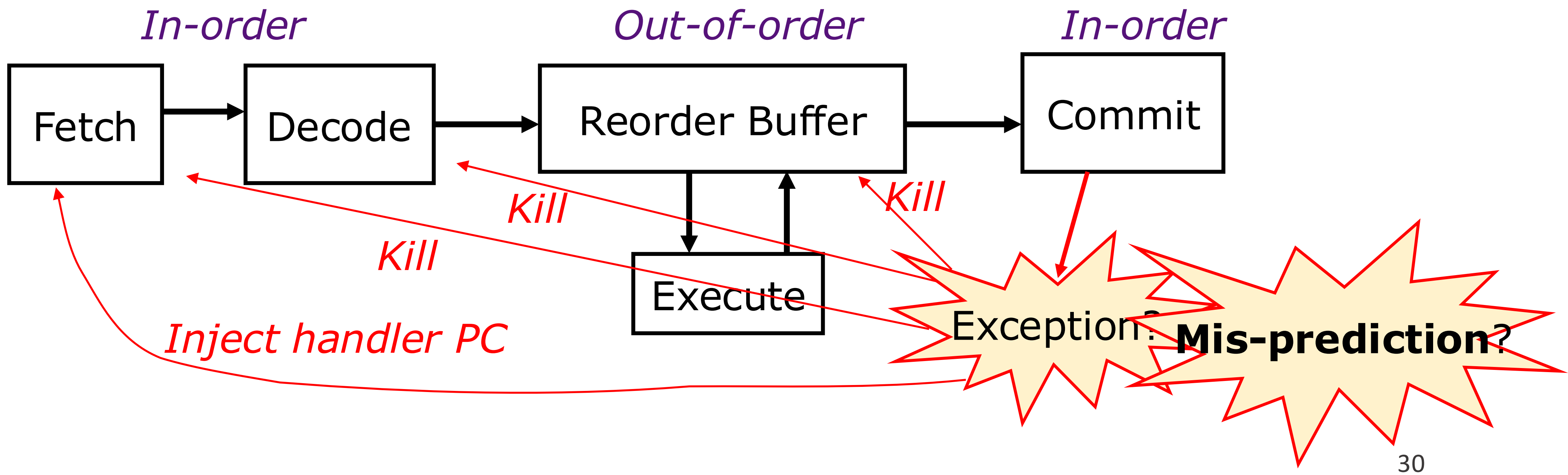
- We generally consider Meltdown as a design **bug**, relatively easy to fix in HW

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

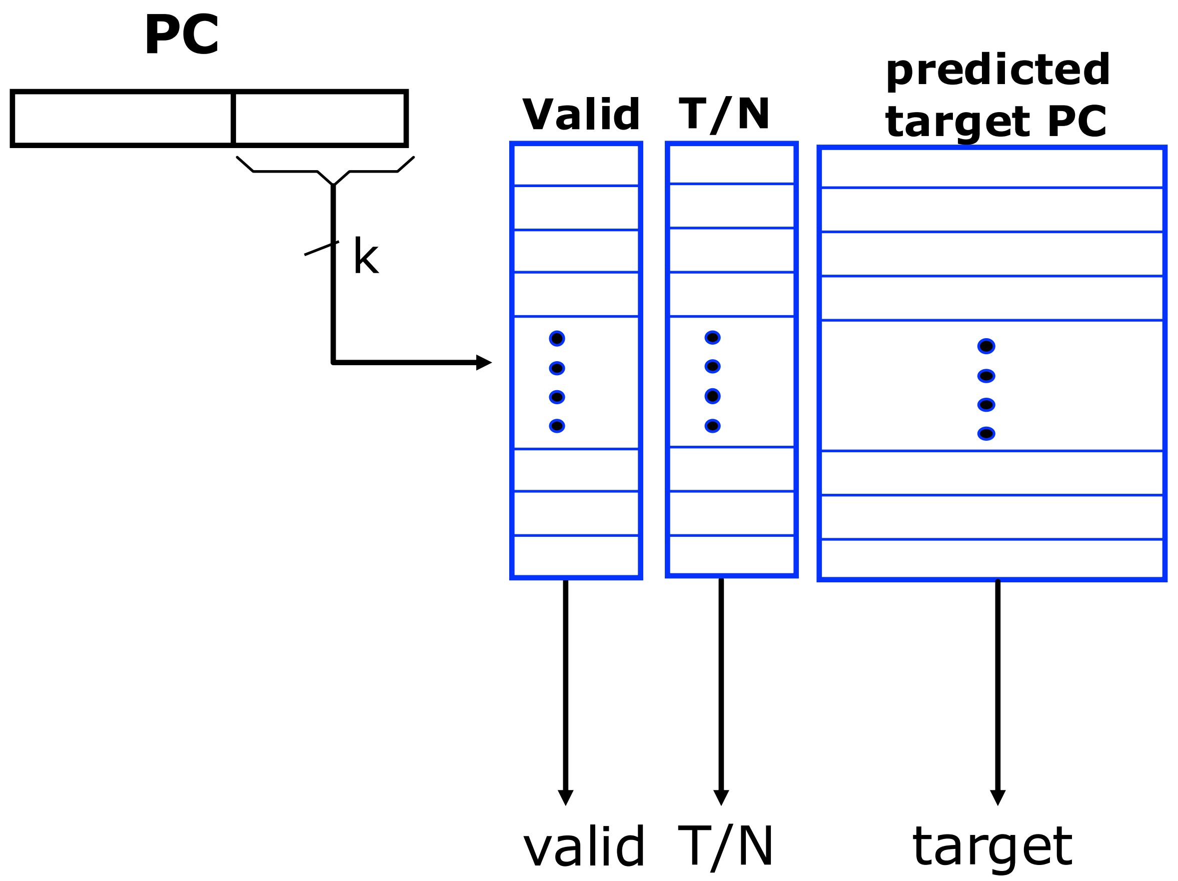
- More advanced, predict two things:
  - Direction of a conditional branch (whether a branch is taken or not)
    - `blt r1, r2, <label>`

Idea: 1-bit predictor for loop

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

Idea: memorizing branch source and destination pairs

Simplified 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
  - Store compressed bits for target PC

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]
     }
```

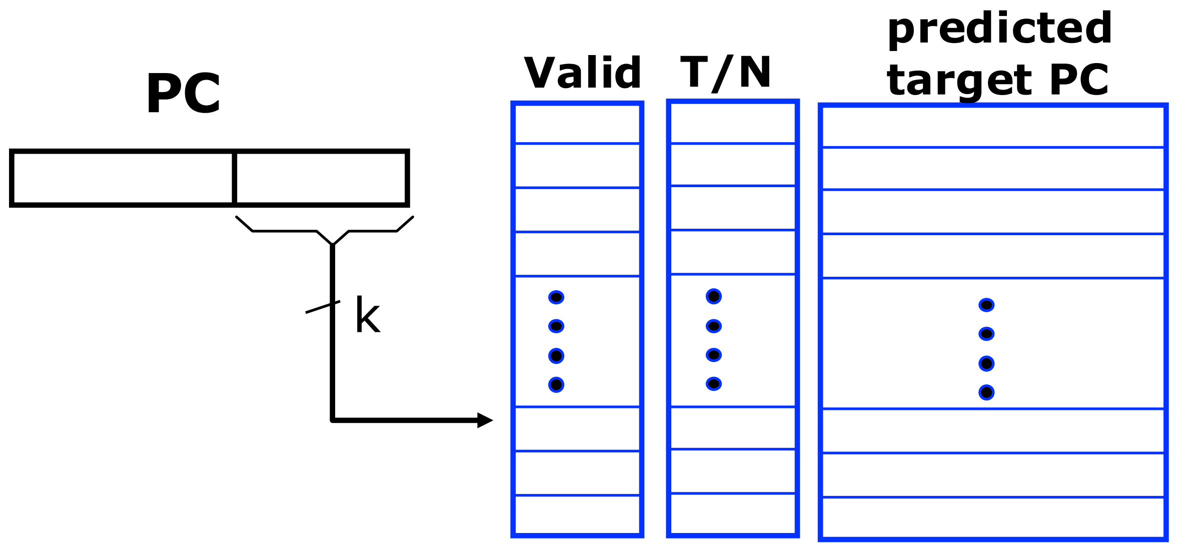
Always malicious?
No. It may be a benign misprediction.
Mitigating Spectre is more challenging.

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

```
Br_train: jump <reg>
// reg=Target_train
……
Target_train: nop
```

```
Br_victim: jump <reg>
// reg=other benign target
……
Target_victim:
    secret = array1[x]
    y = array2[secret*4096]
```



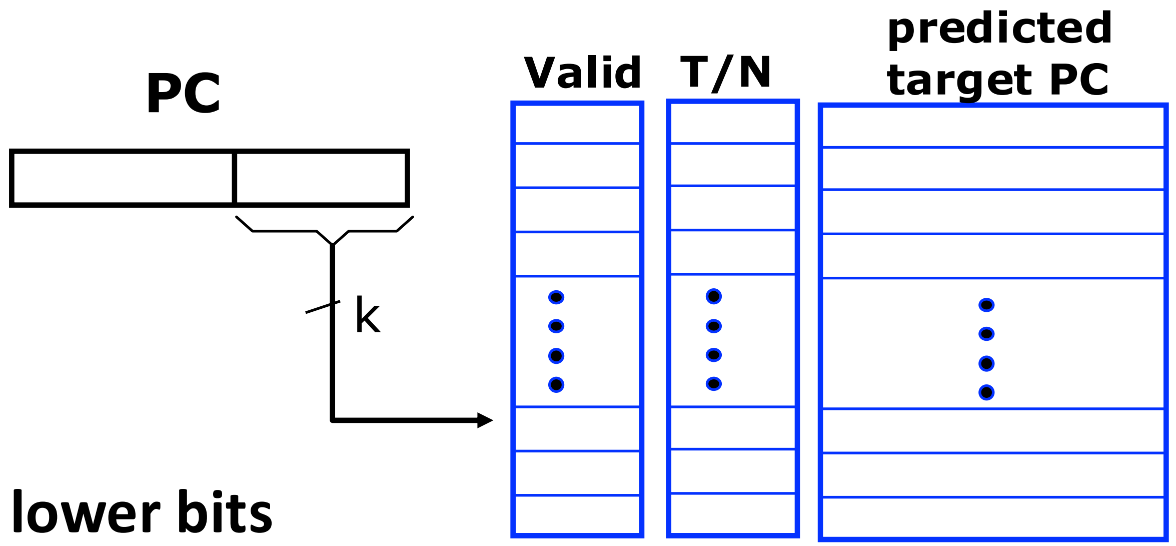
Spectre V2 – Speculative JOP

```
Br_train: jump <reg>
// reg=Target_train
……
Target_train: nop
```

```
Br_victim: jump <reg>
// reg=other benign target
……
Target_victim:
    secret = array1[x]
    y = array2[secret*4096]
```

Alias on BPU

Match lower bits



Timeline

1. Insert <Br_train, Target_train> into BPU
2. Trigger Br_victim
3. Speculatively execute Target_victim

Next:
Software-Hardware Contracts