Comp 590-184: Hardware Security and Side-Channels

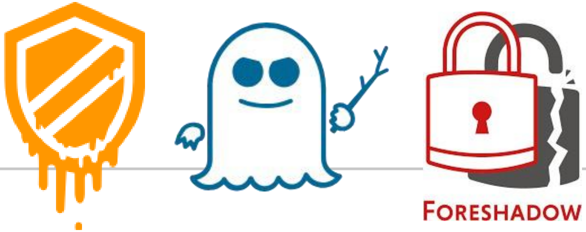
Lecture 9: Transient Execution Attacks

February 12, 2026
Andrew Kwong

THE UNIVERSITY
of NORTH CAROLINA
at CHAPEL HILL

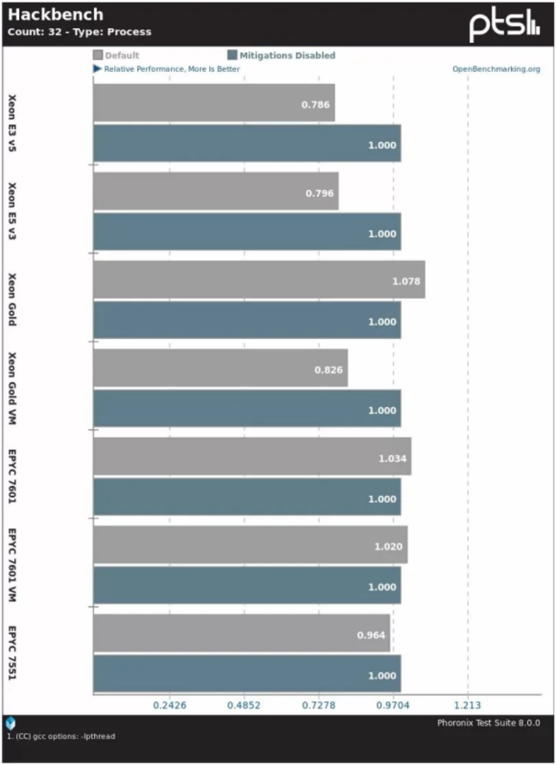
Slides adapted from
Mengjia Yan
(shd.mit.edu)

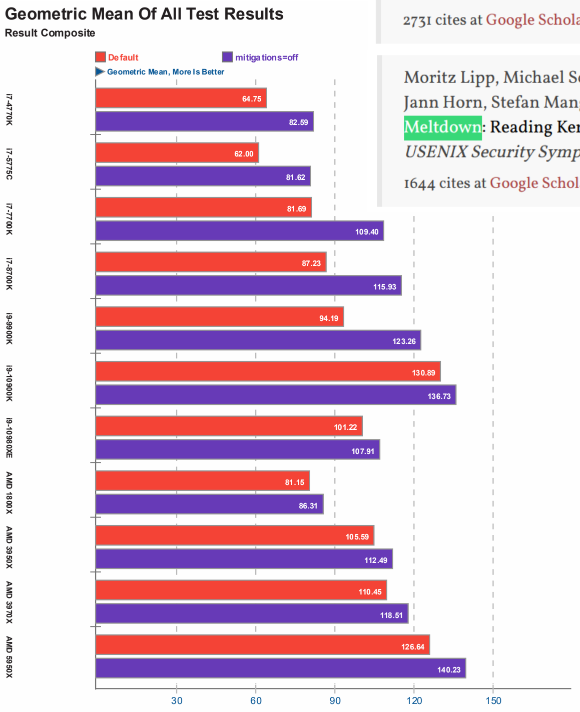
Outline

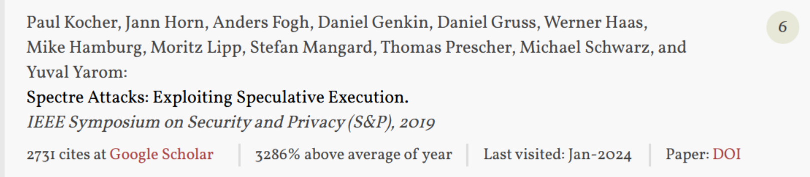


- What are transient execution attacks?
- How does Meltdown work?
  - Background on out-of-order processors
  - We will connect the dots between a hardware optimization and a software optimization.
- How do Spectre and its variations work?
  - Background on speculative execution
  - Let’s try to see through these variations and understand the fundamental problem.

Impact







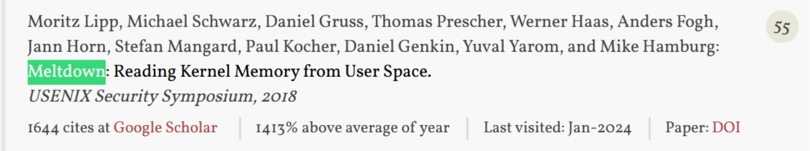
Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom:

Spectre Attacks: Exploiting Speculative Execution.

IEEE Symposium on Security and Privacy (S&P), 2019

2731 cites at Google Scholar | 3286% above average of year | Last visited: Jan-2024 | Paper: DOI

6



Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg:

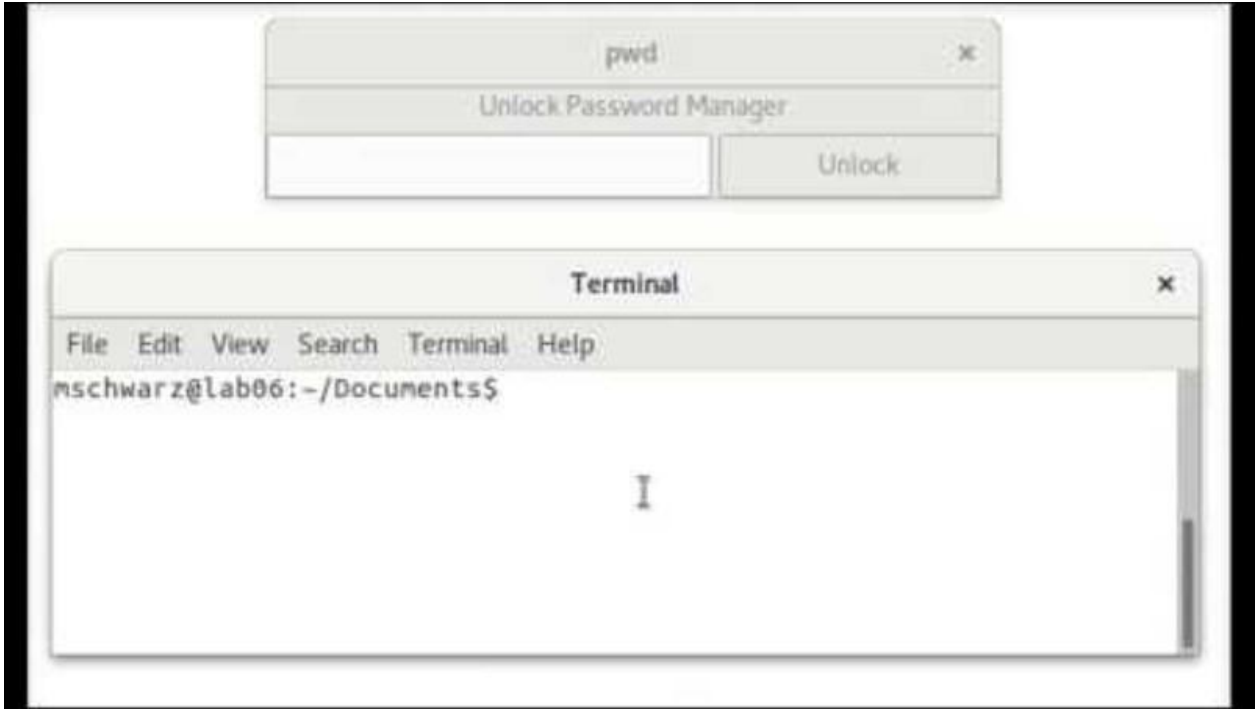
Meltdown: Reading Kernel Memory from User Space.

USENIX Security Symposium, 2018

1644 cites at Google Scholar | 1413% above average of year | Last visited: Jan-2024 | Paper: DOI

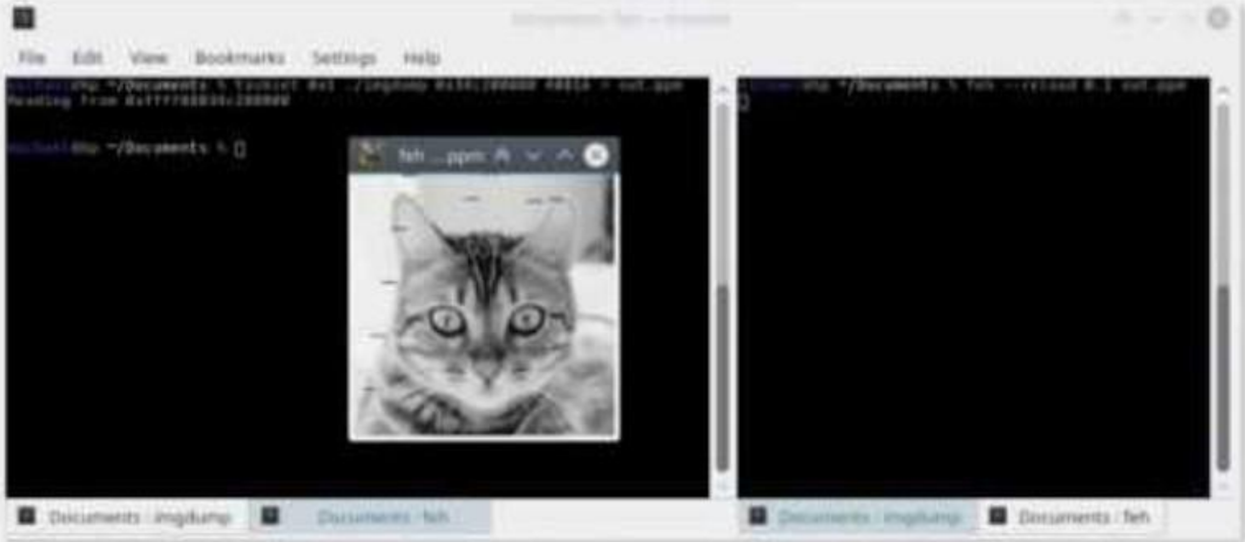
55

Meltdown



```
mschwarz@lab06:~/Documents$
```

Meltdown



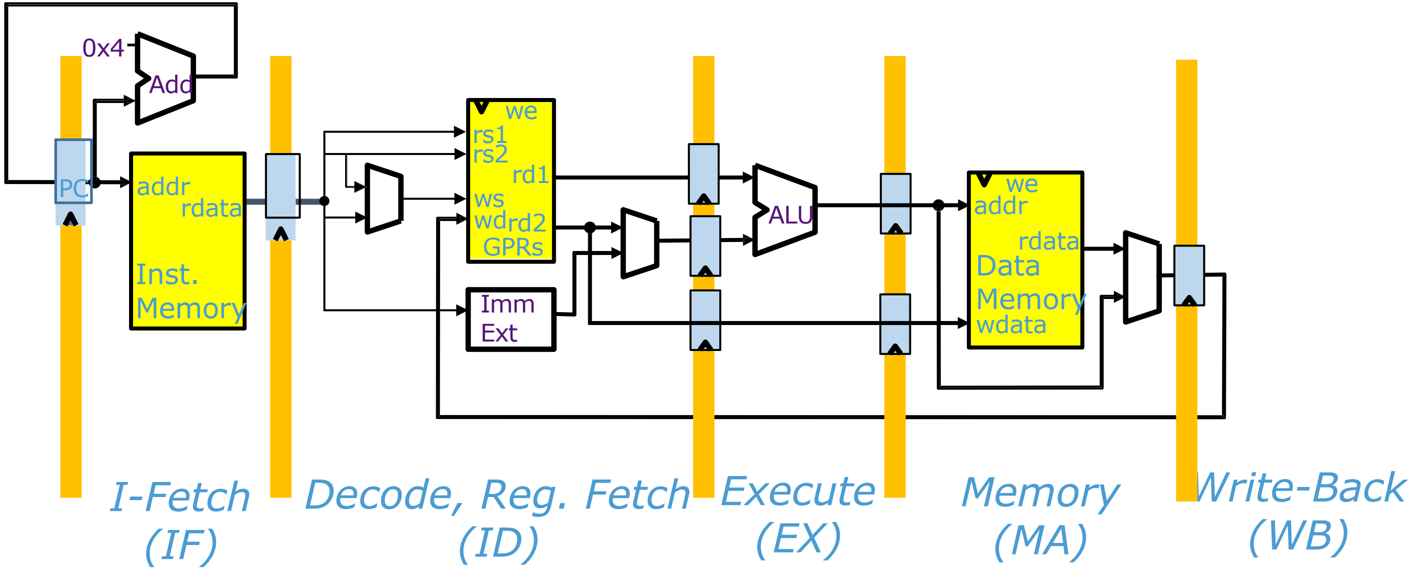
```
[illegible] ~/Documents % taskset 0x1 ./imgdump [illegible] [illegible] > out.ppm
Reading from [illegible]

[illegible] ~/Documents % []
```

```
[illegible] ~/Documents % feh --reload 0.1 out.ppm
[]
```

Meltdown Root Causes

- Due to the combination of both a hardware and software optimization
  - Out of order execution
  - Mapping kernel memory into user space

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$_5$

Build High-Performance Processors

Example #1:

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

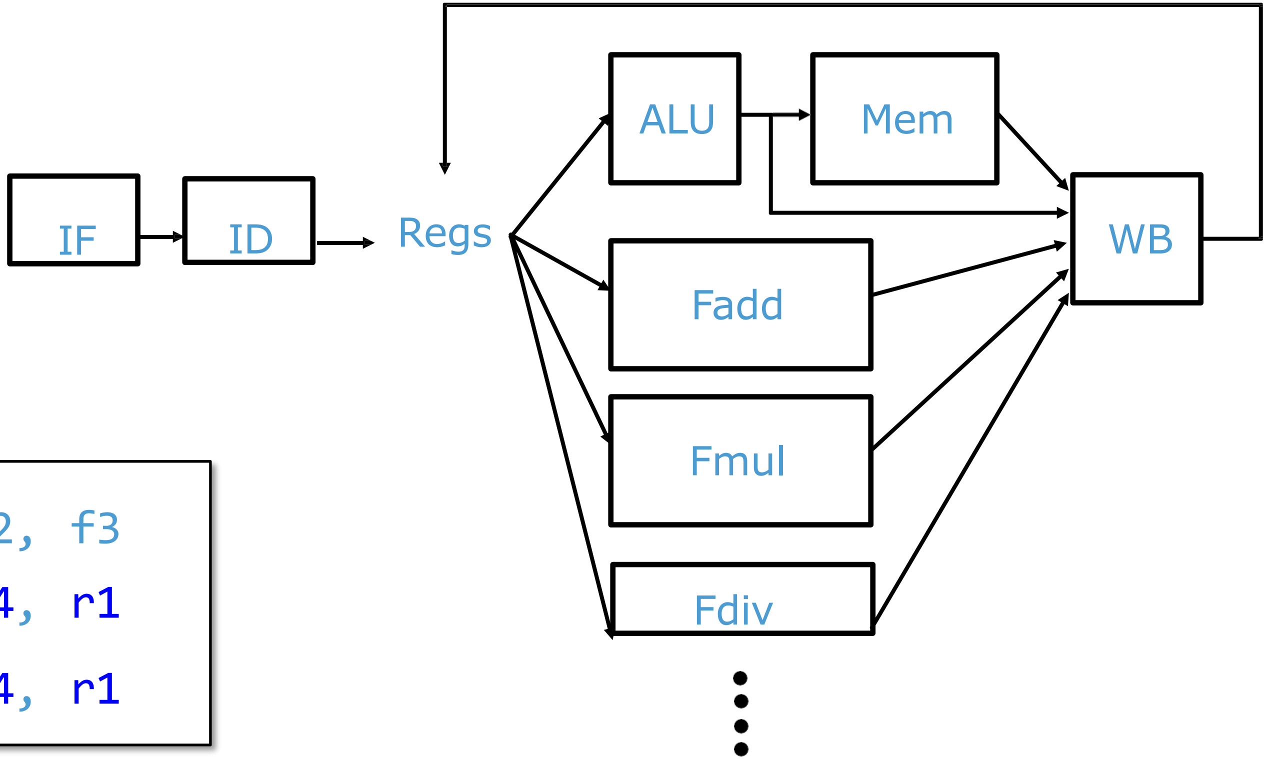
Instruction-Level Parallelism (ILP)

when there is no data-dependency or control-flow dependency between instructions

Example #2:

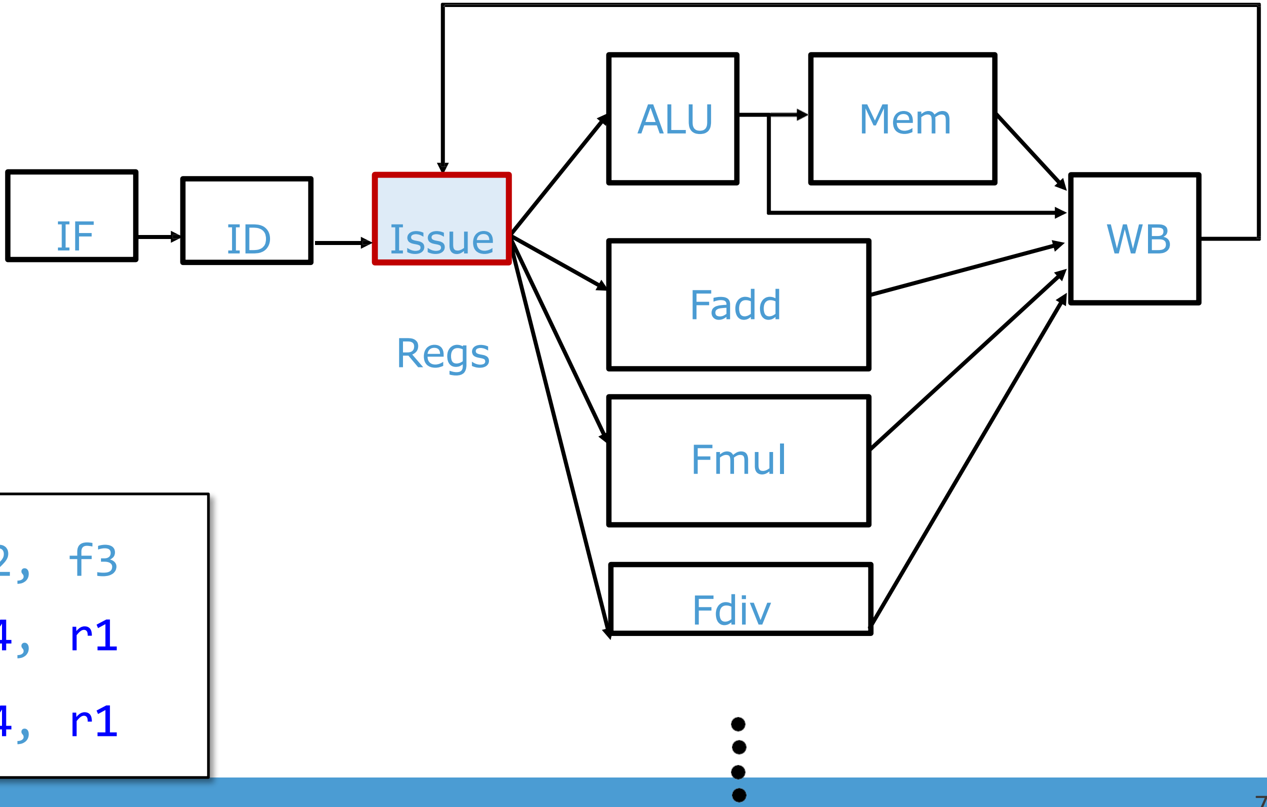
```
LD  r3, 0(r2)     ; 1-100 cycles
ADD r4, r4, r1    ; 1 cycle -> repeat 10 times
……
```

Technique #1: Add More Functional Units



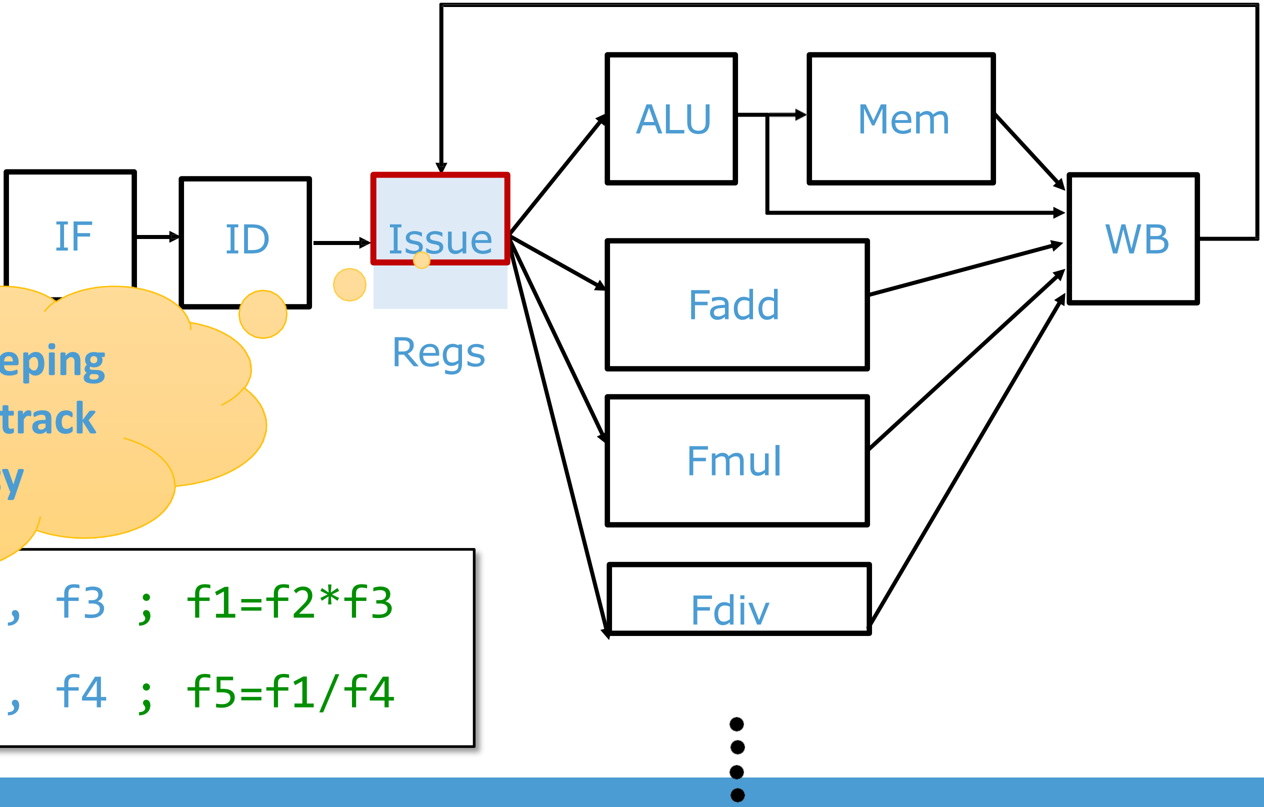
```
1:  FMUL f1,  f2,  f3
2:  ADD  r4,  r4,  r1
3:  ADD  r4,  r4,  r1
```

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				

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

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
```

Technique #2: Scoreboard

Functional Unit	Busy?	Dest Reg	Src1 Reg	Src2 Reg
Int ALU	Y	r4	r4	r1
Mem				
Fadd				
Fmul	Y	f1	f2	f3
Fdiv				

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

Technique #2: Scoreboard

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

```
1: FMUL f1, f2, f3

2: FDIV f5, f1, f4
```

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: FDIV f5, f1, f4
```

Technique #2: Scoreboard

Functional Unit	Busy?	Dest Reg	Src1 Reg	Src2 Reg
Int ALU				
Mem				
Fadd				
Fmul	Y	f1	f2	f3
Fdiv	Y	f5	f1	f4

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

Read-after-write (RAW)

Technique #2: Scoreboard

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

write-after-write (WAW)

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

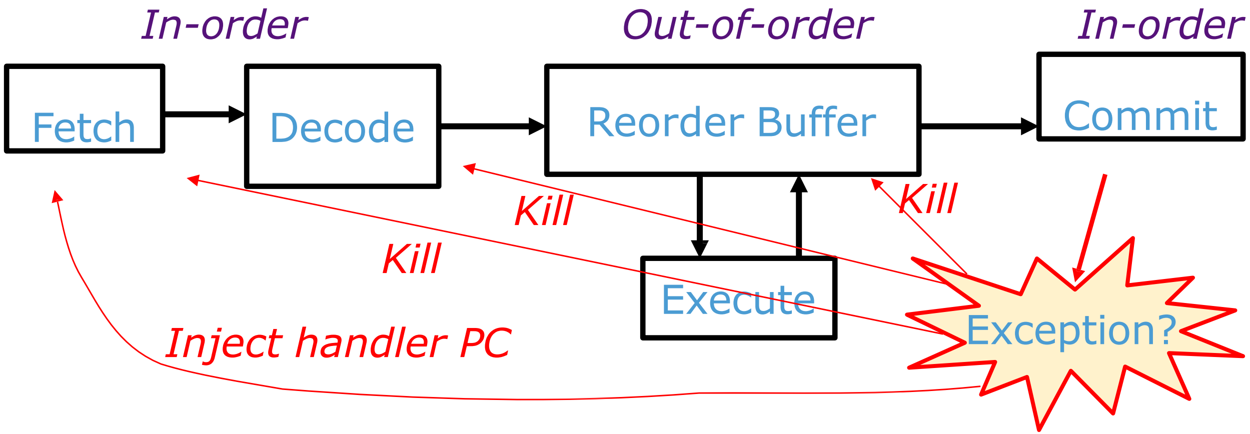
Technique #2: Scoreboard

- Upon issue of 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?

Another Way to Draw It



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		

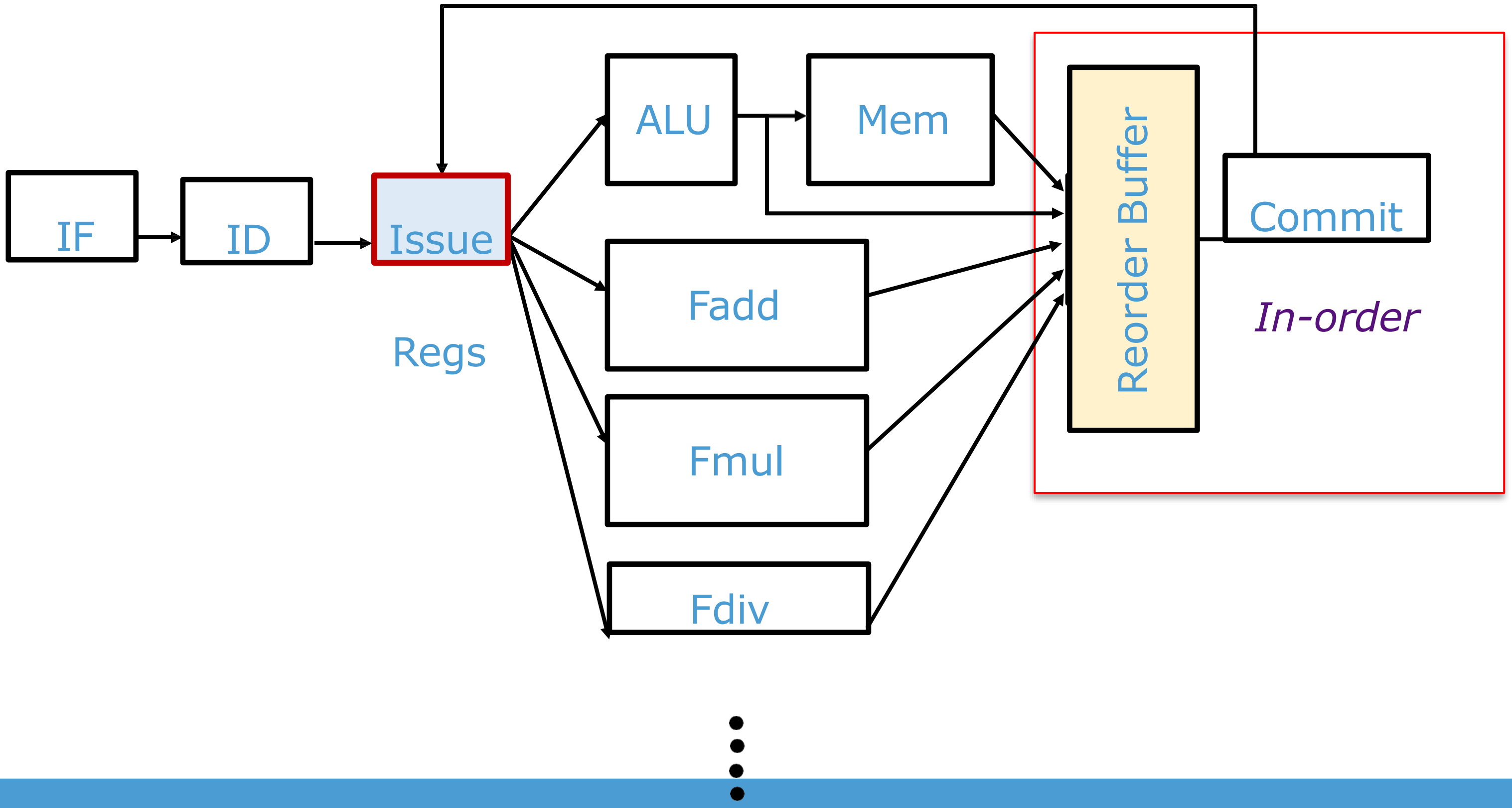
Exception in OoO Processors: Example #2

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

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

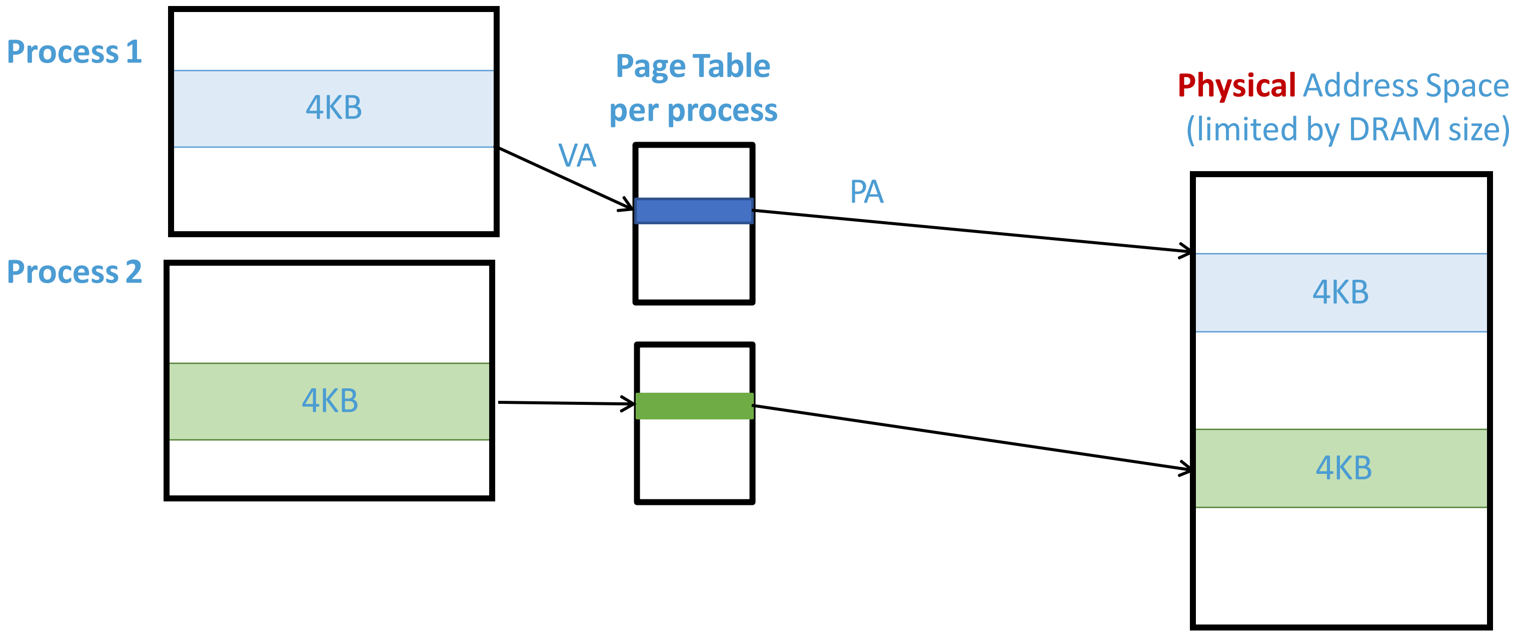


Re-examine Examples With In-order Commit

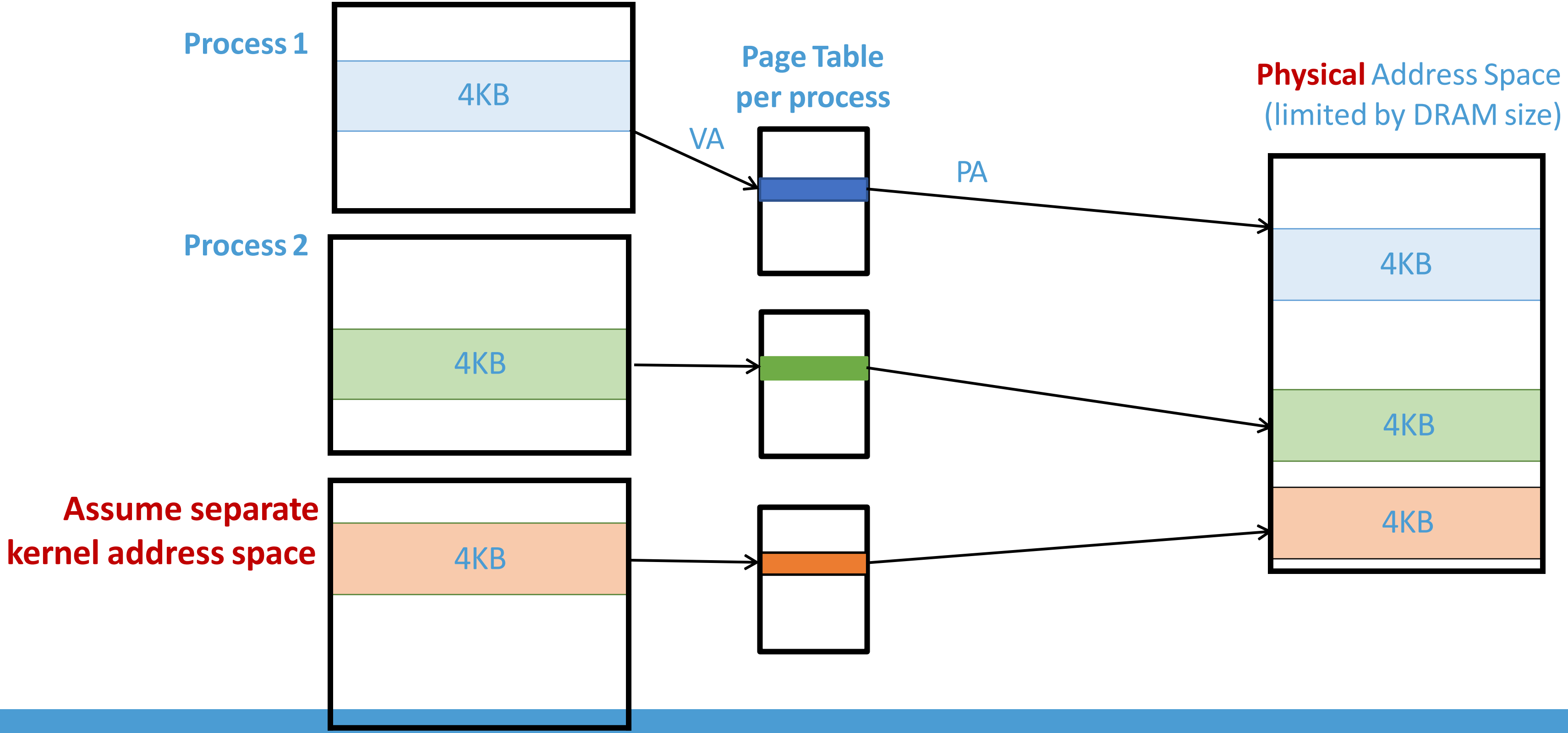
```
1: LD   r3, 0(r2)     ; Exception in 3 cycles
2: ADD r4, r4, r1     ; 1 cycle
```

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

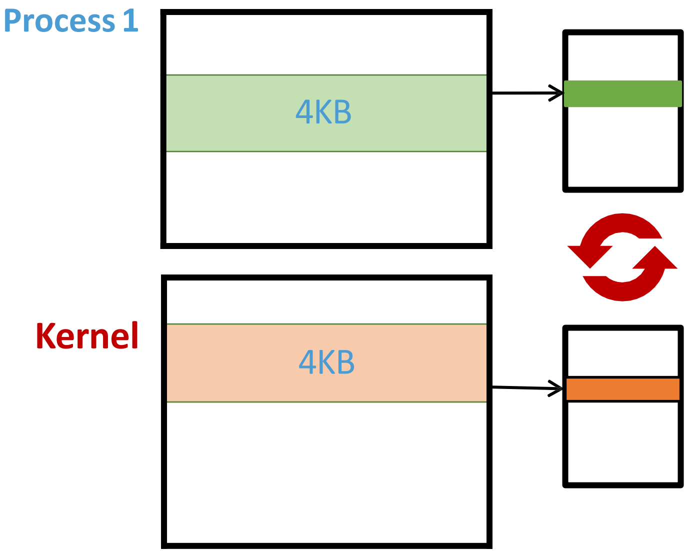
Recap: Page Mapping



Mapping Kernel Pages



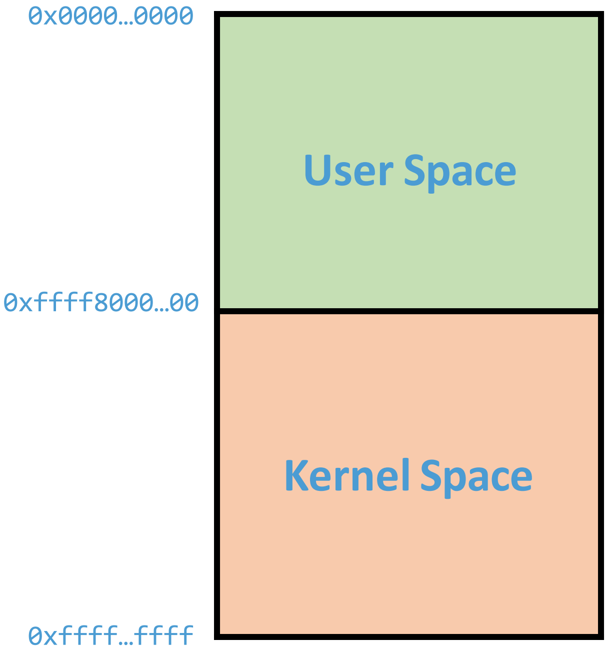
Jumping Between User and Kernel Space



- 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

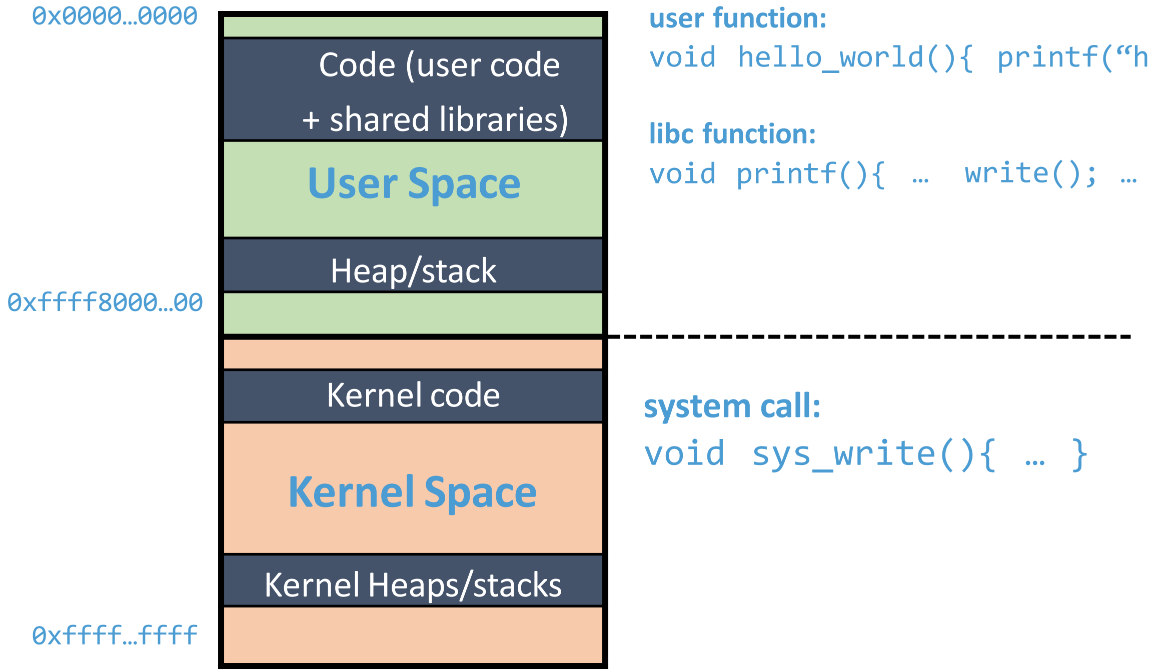
Map Kernel Pages Into User Space Securely

Virtual memory (x86_64 Linux)



Virtual Memory

Virtual memory (x86_64 Linux)



user function:

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

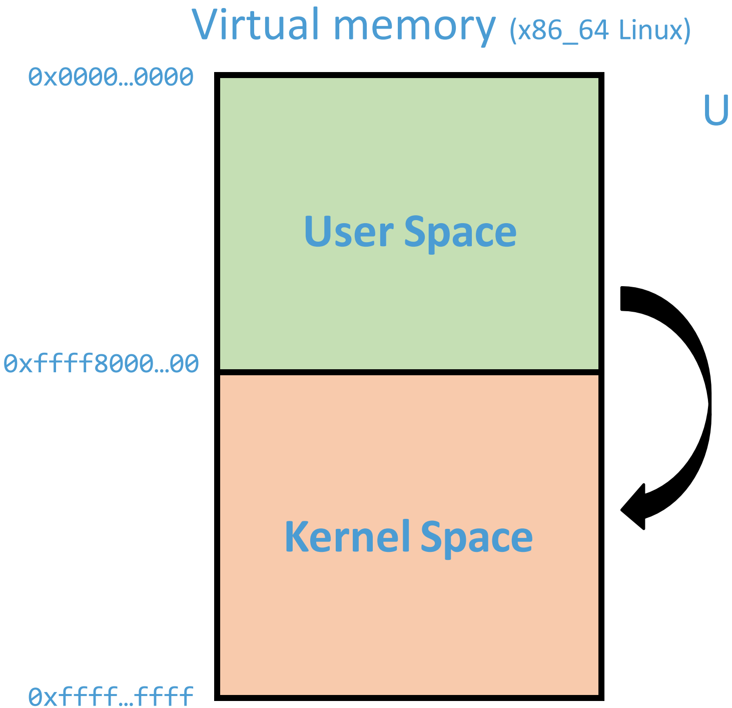
libc function:

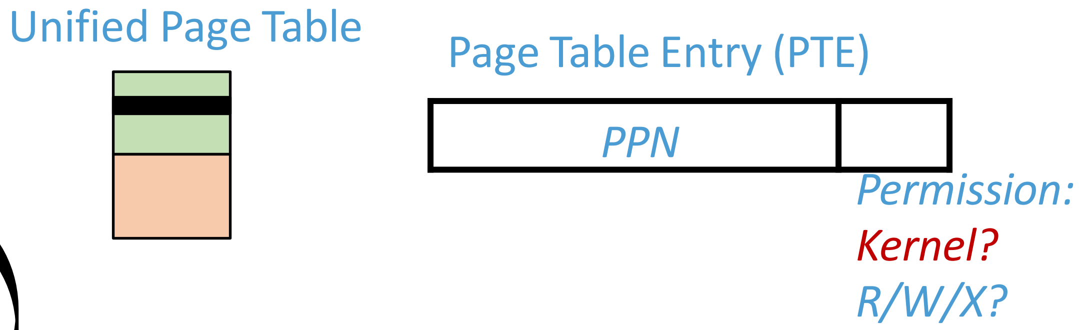
```
void printf(){ …  write(); … }
```

system call:

```
void sys_write(){ … }
```

Map Kernel Pages Into User Space **Securely**





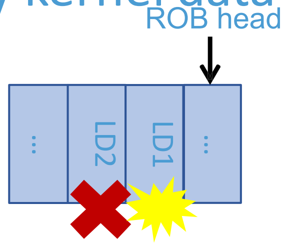
- Fast context switch:
  - The kernel remains **mapped**
  - Only a **mode switch** (user → kernel) is needed

Meltdown

- Put two optimizations together, we have Meltdown
  - Hardware optimization: out-of-order execution
    - Deferred exception handling
  - Software optimization: mapping kernel addresses into user space

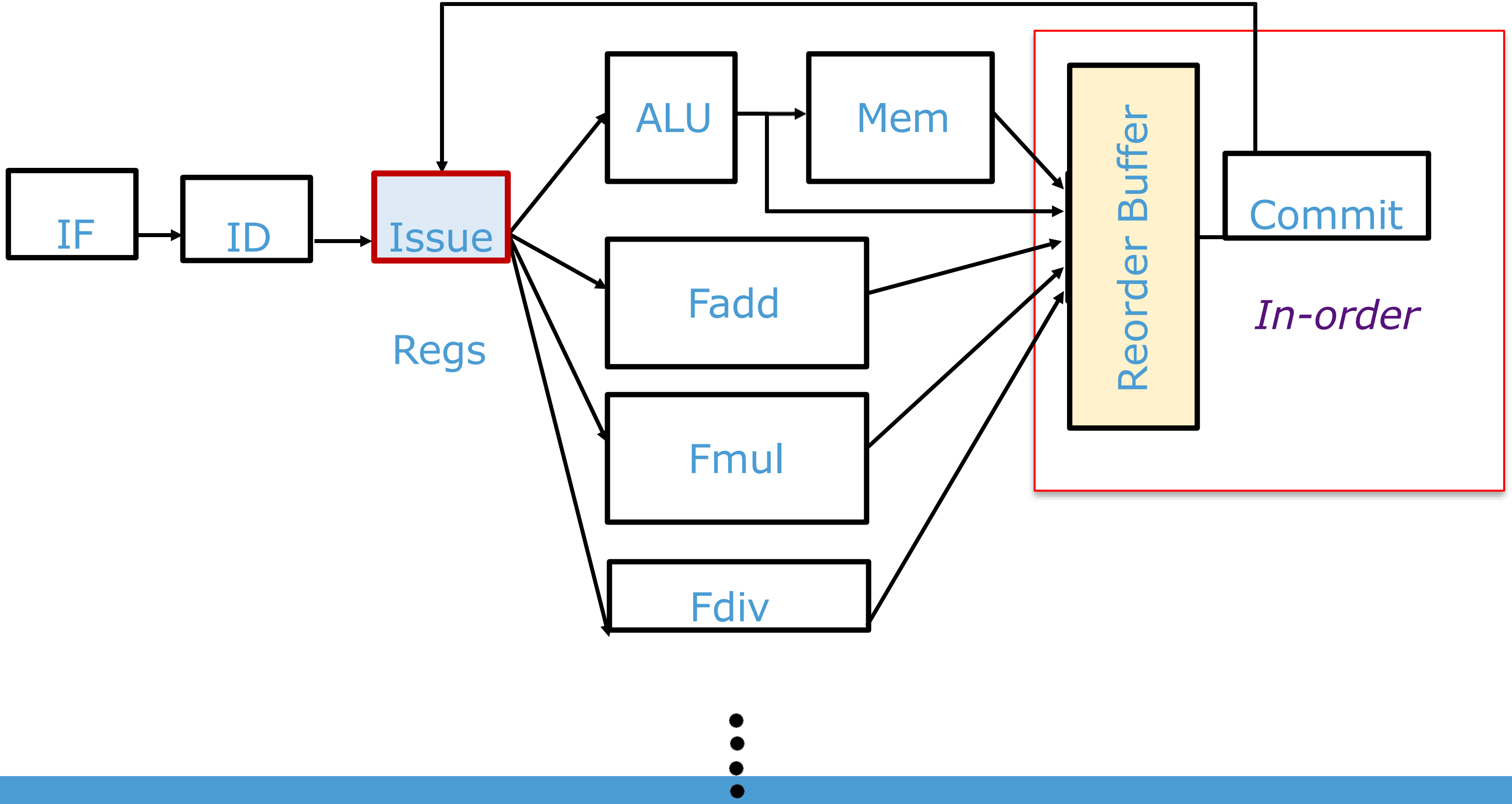
- Attack outcome: user space applications can read arbitrary kernel data

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



2nd line of code can transiently execute before the exception occurs!

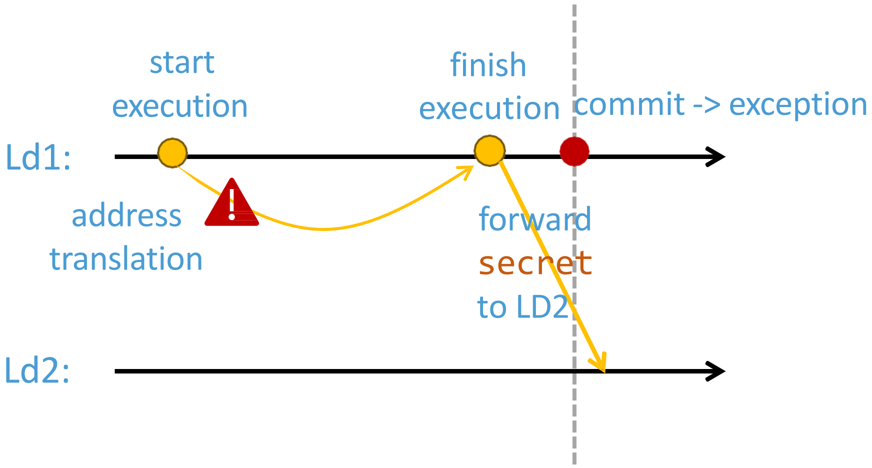
Technique #3: In-order Commit



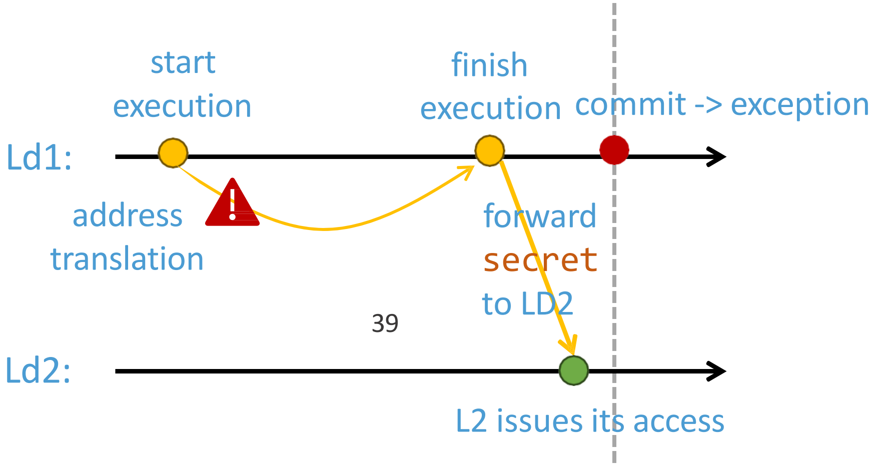
Meltdown Timing

```
……
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.


start execution
finish execution
commit -> exception
Ld1:
address translation
forward secret to LD2
Ld2:
L2 issues its access