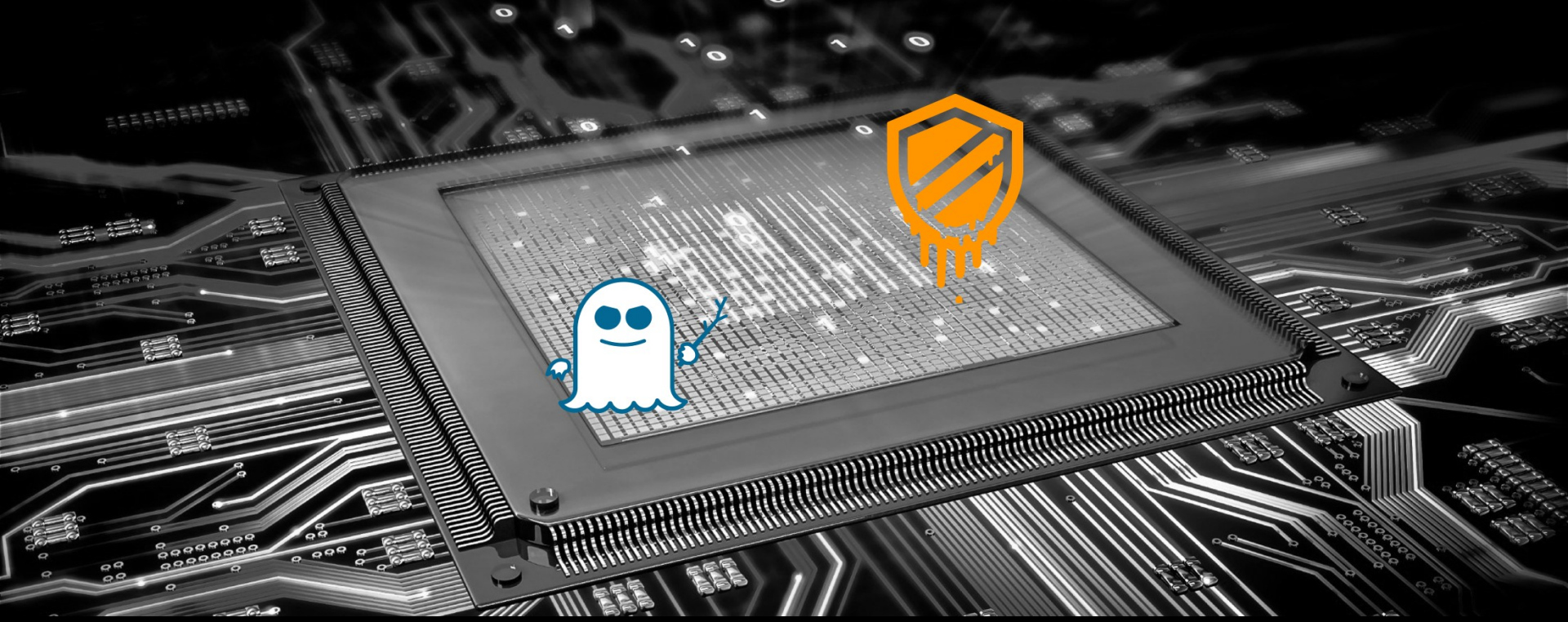


It’s not over with Meltdown

L1TF, POP SS, TLBleed and more ...

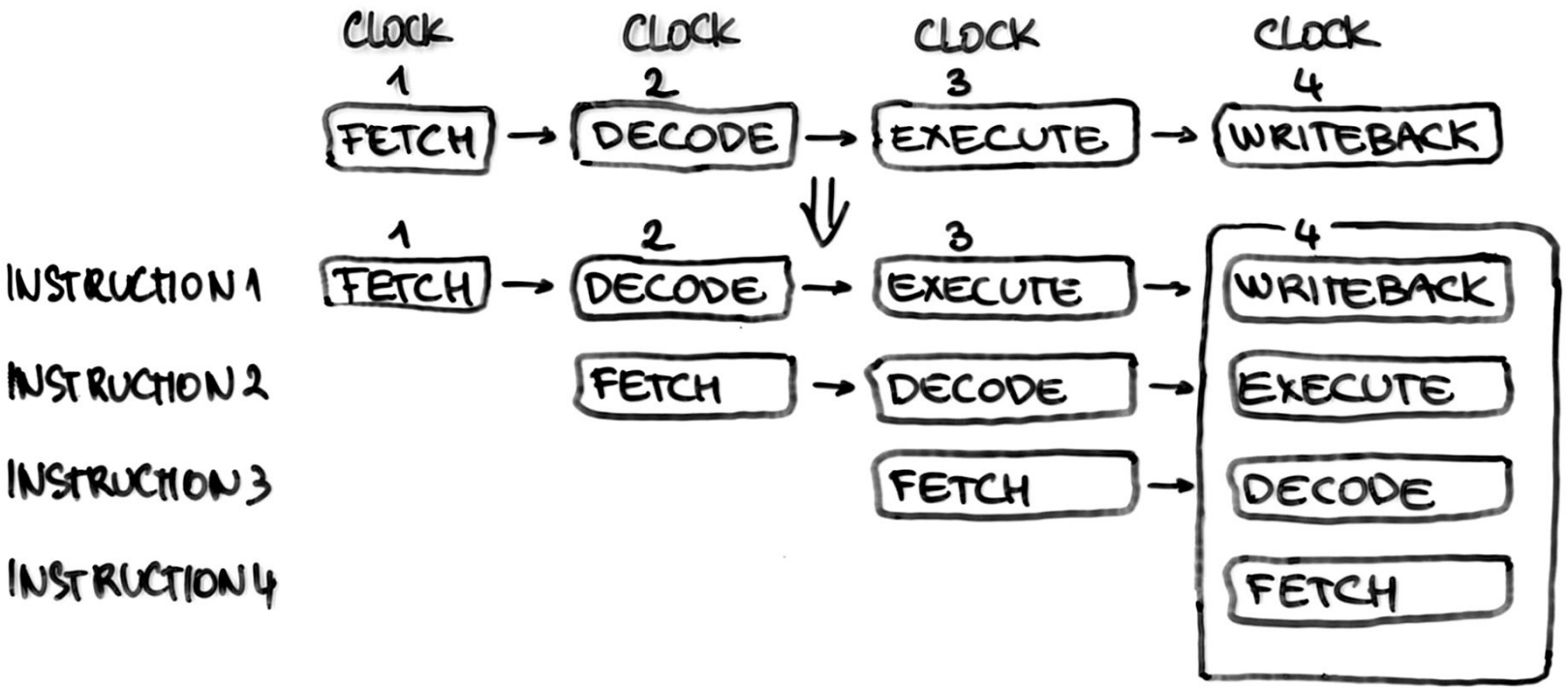
Vojtěch Pavlík
Director SUSE Labs

SUSE
We adapt. You succeed.

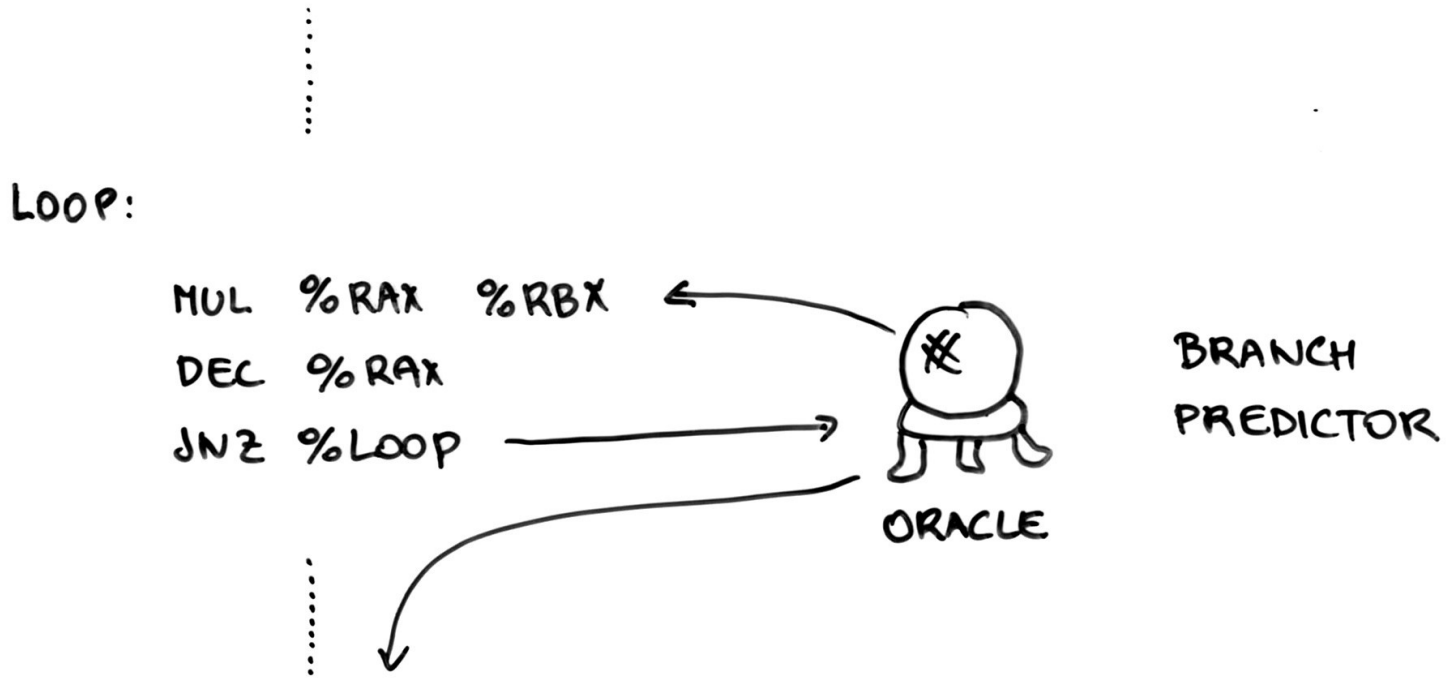
1967: Robert Marco Tomasulo

- Implemented a revolutionary algorithm
  - in IBM System/360 FPU
  - significantly improved execution efficiency
  - today known as Tomasulo's algorithm
- Received Eckert-Mauchly prize
  - like a Nobel prize in computer architecture
  - named after the creators of ENIAC
- But the consequences ...

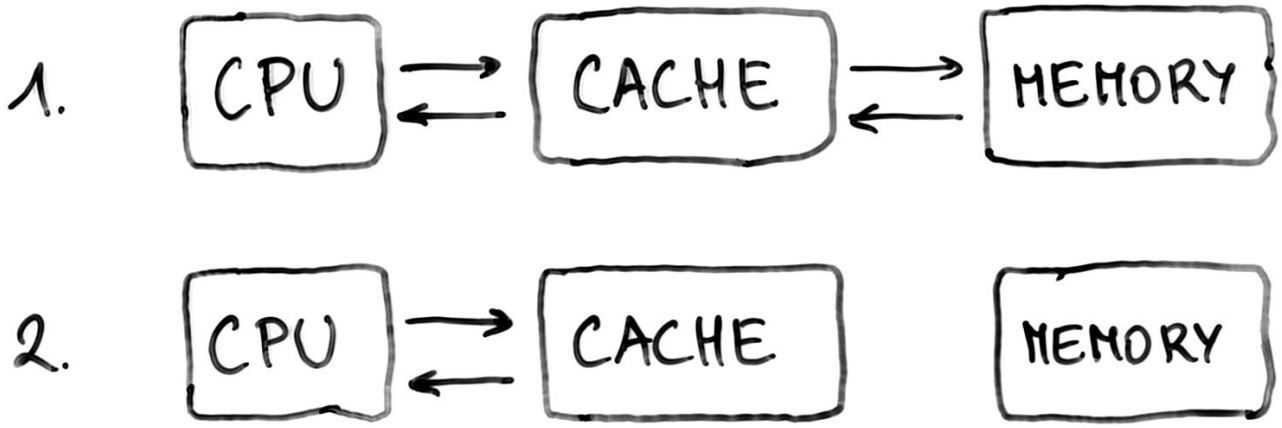
Pipelining


CLOCK 1
CLOCK 2
CLOCK 3
CLOCK 4
FETCH
DECODE
EXECUTE
WRITEBACK
INSTRUCTION 1
INSTRUCTION 2
INSTRUCTION 3
INSTRUCTION 4
FETCH
DECODE
EXECUTE
WRITEBACK
FETCH
DECODE
EXECUTE
FETCH
DECODE
FETCH

Branch prediction

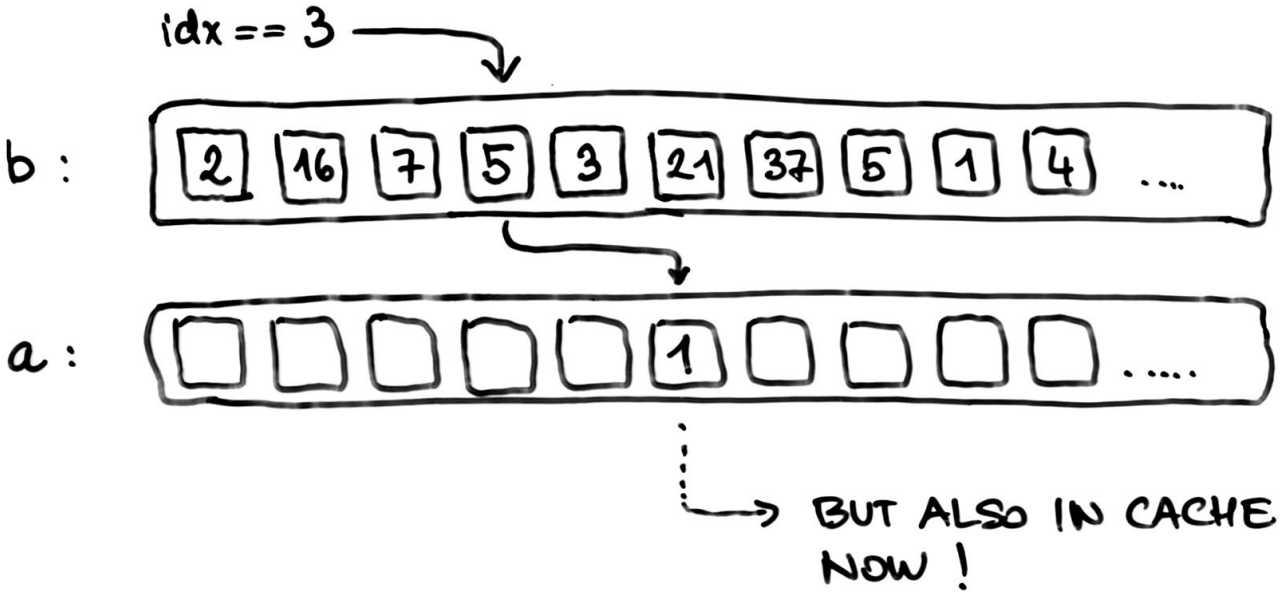


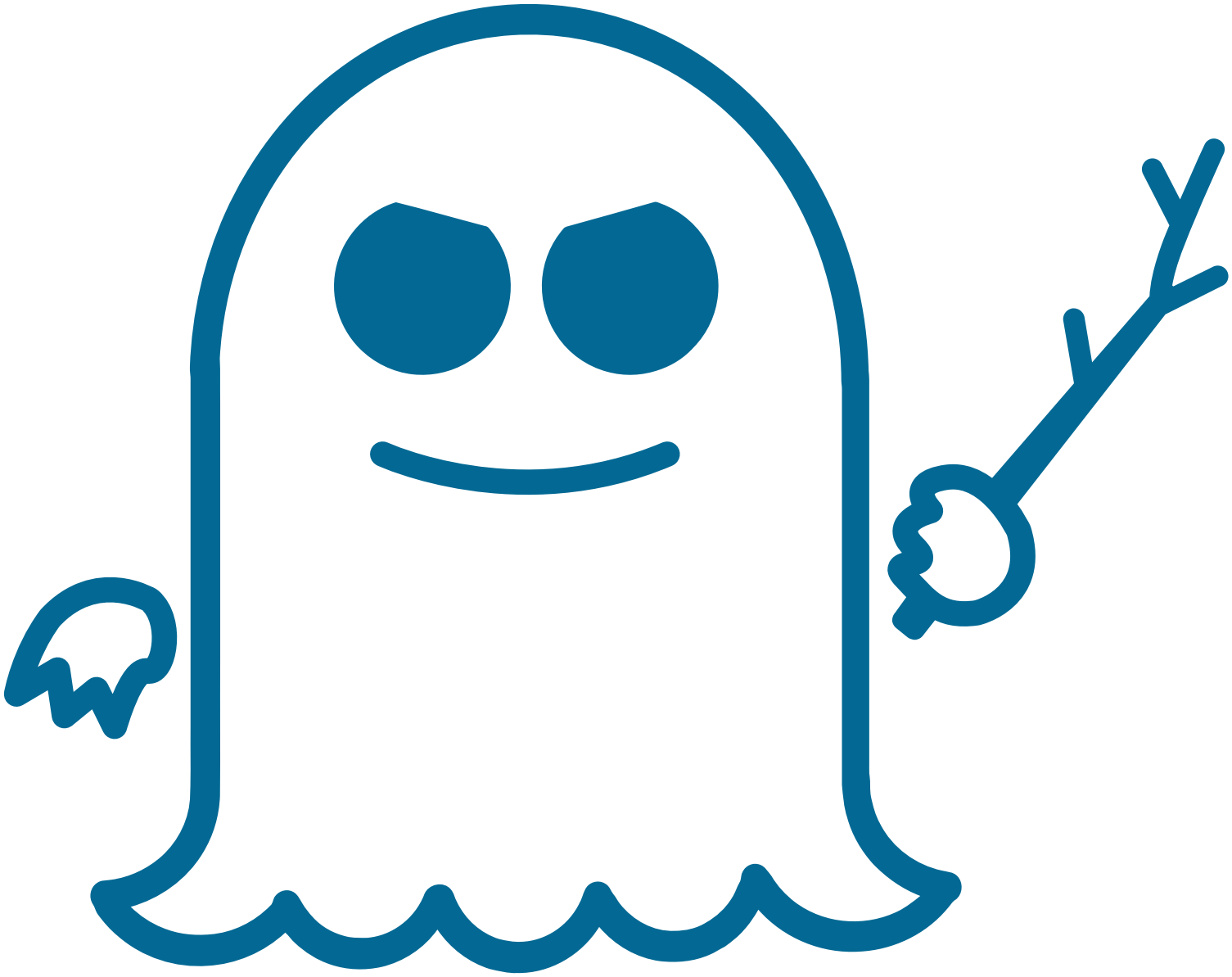
Cache



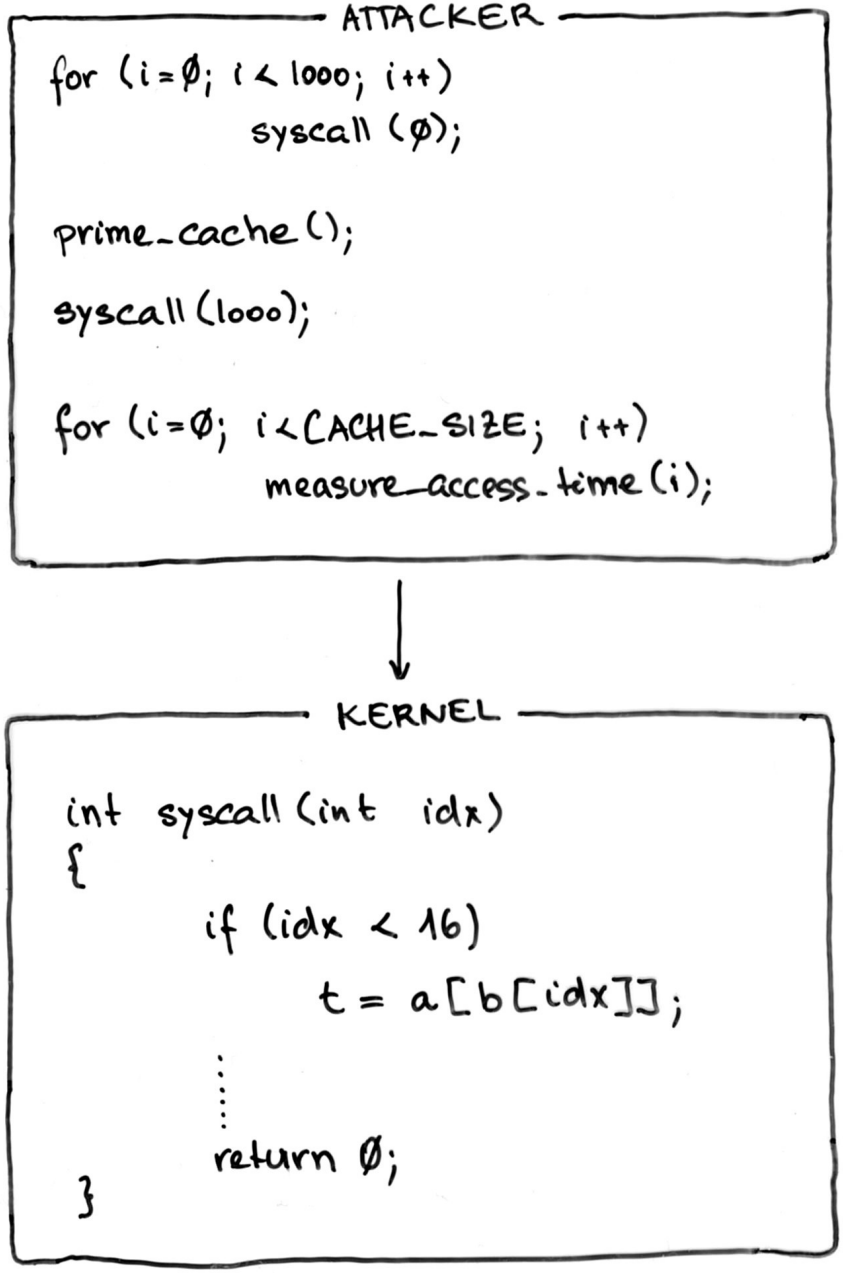
Double indirection

```
a[b[idx]] = 1;
```





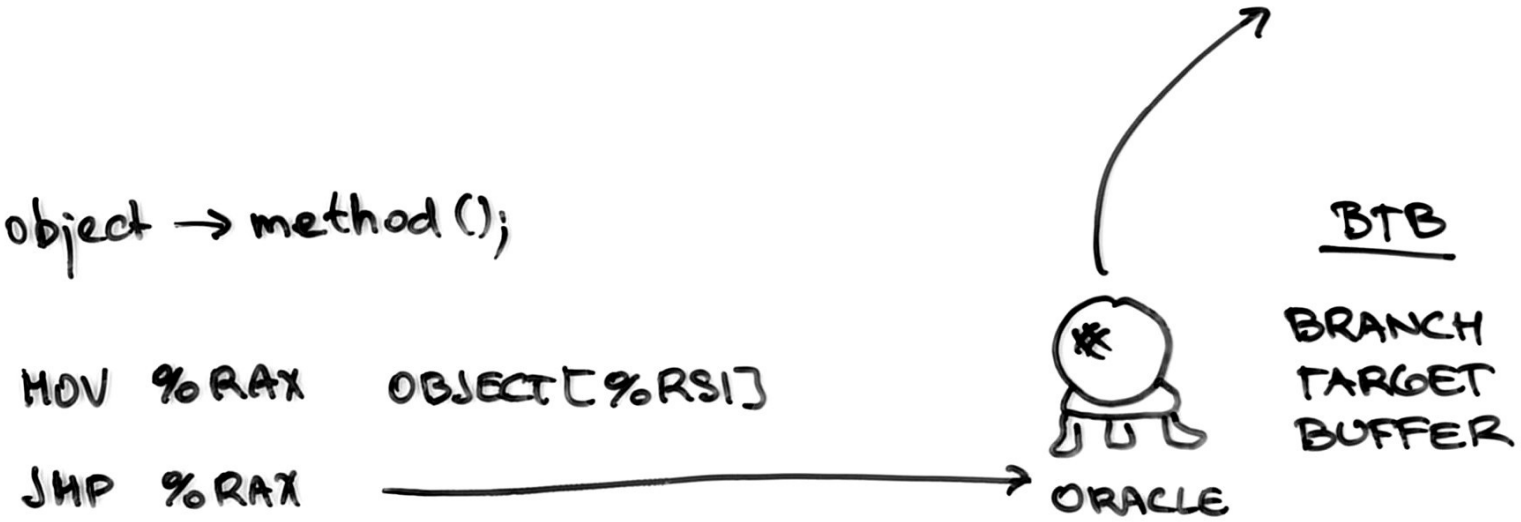
Spectre v1 Attack!



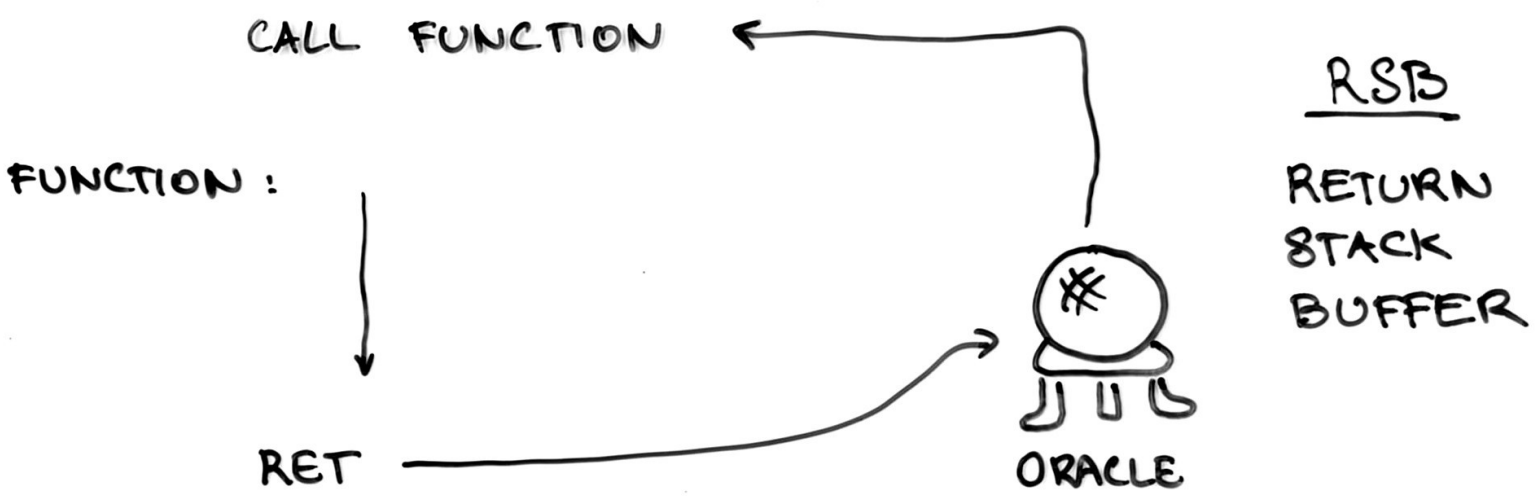
Arithmetics

```
int syscall (int idx)
{
    t = a[b[idx & 0xf]];
    if (idx < 16) {
        ⋮
```

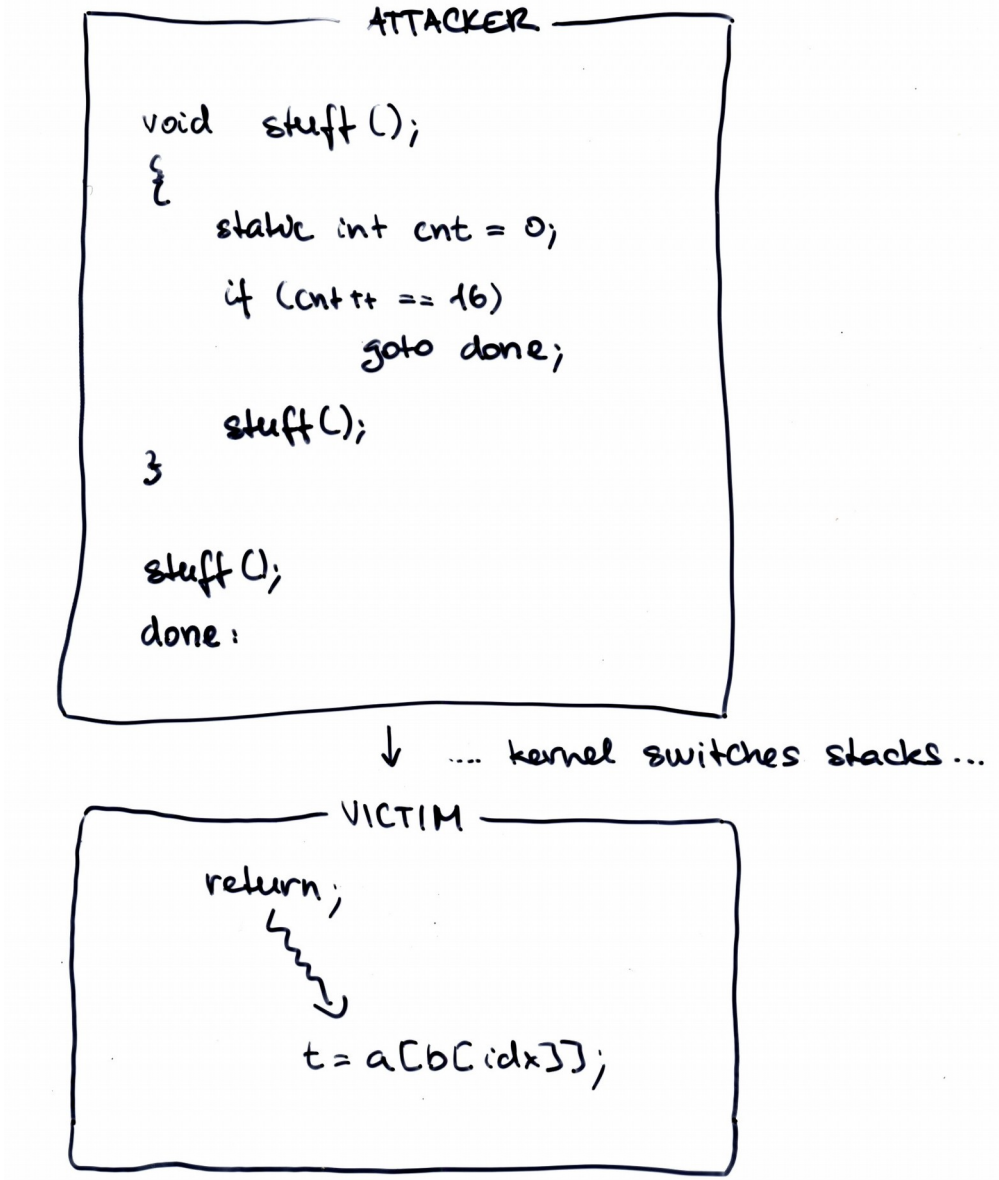
Indirect branch prediction



Return target prediction



SpectreRSB Attack!



RSB Stuffing

- To overcome SpectreRSB the kernel has to flush the RSB contents on every context switch
- Same what the attacker did – do a sequence of calls and remove them from stack
- Costs performance
  (not much of a surprise, is it?)

Lazy FPU switching

- FPU register save/restore used to be slow on 80387
- But *disabling* the FPU was fast
- So the kernel would on a context switch
  - Save the FPU state if it was used
  - Disable the FPU
- The CPU would raise an exception on FPU access
  - The kernel the restores saved FPU state
  - And remembers the FPU was used
- This avoids many costly saves and restores since most tasks don't use FPU – gaining performance!

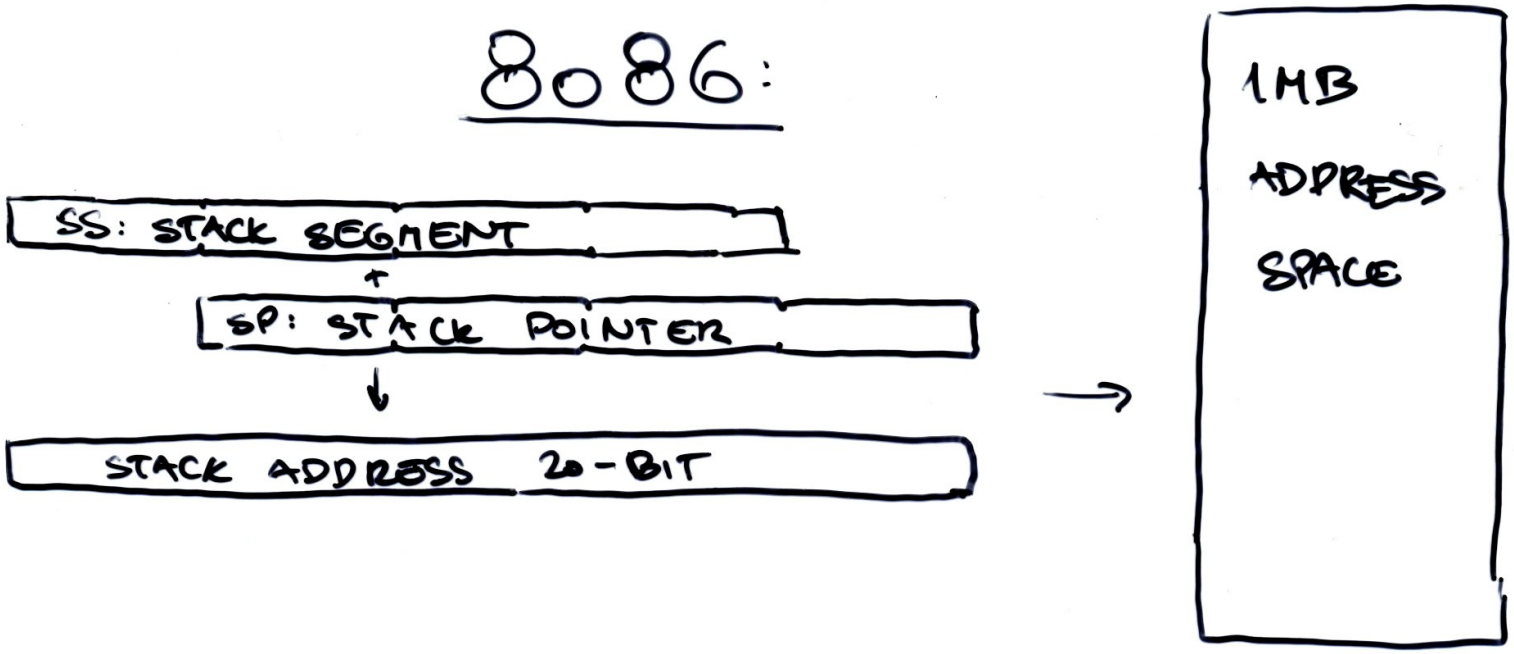
Lazy FPU Attack!

- When speculating, the CPU doesn’t check if FPU is disabled
- Combined with double-indirection, attacker can snoop FPU register contents

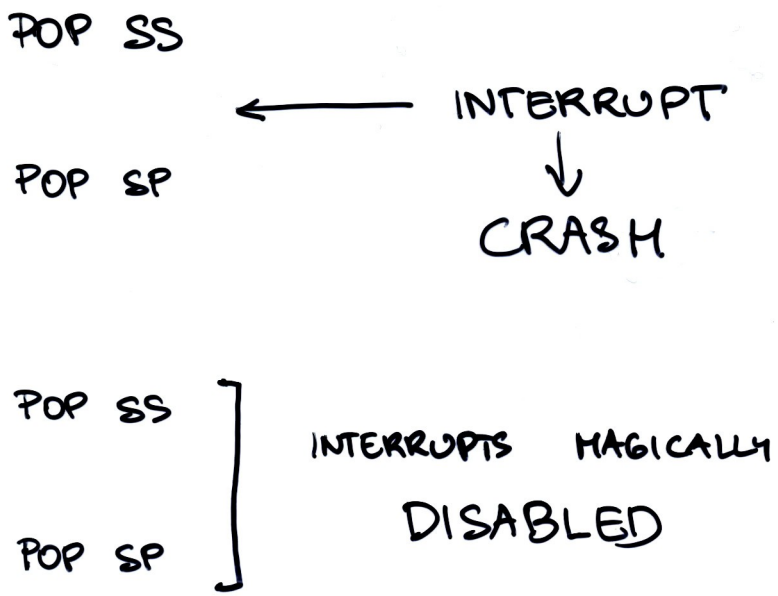
Eager FPU save/restore

- To mitigate, the kernel must not disable the FPU
- And save/restore FPU state on every context switch
- This costs (you knew this was coming) performance
- But fortunately there's FXSAVE and FXRSTOR
  Instructions for fast FPU state saving on modern CPUs
- So it's not that bad after all

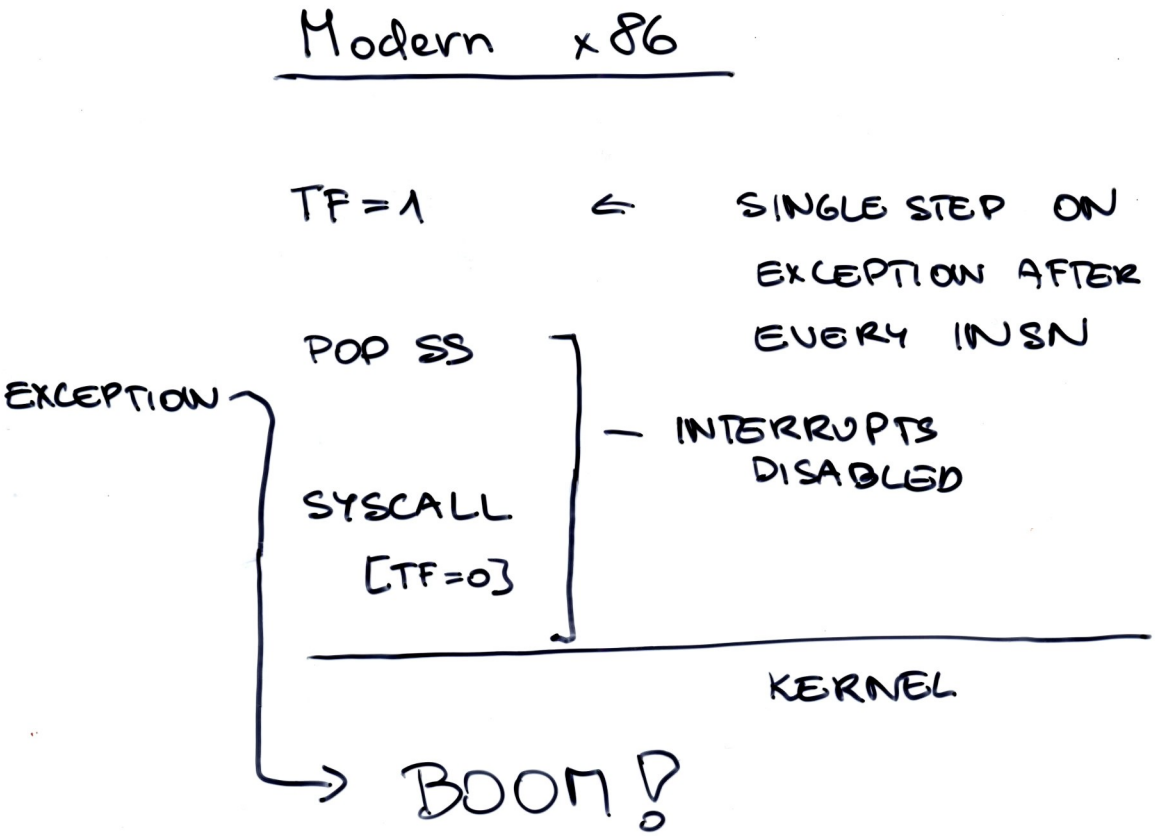
x86 Stack


8086:
SS: STACK SEGMENT
SP: STACK POINTER
STACK ADDRESS 20-BIT
1MB
ADDRESS
SPACE

POP SS Magic


POP SS
INTERRUPT
POP SP
CRASH
POP SS
INTERRUPTS MAGICALLY
DISABLED
POP SP

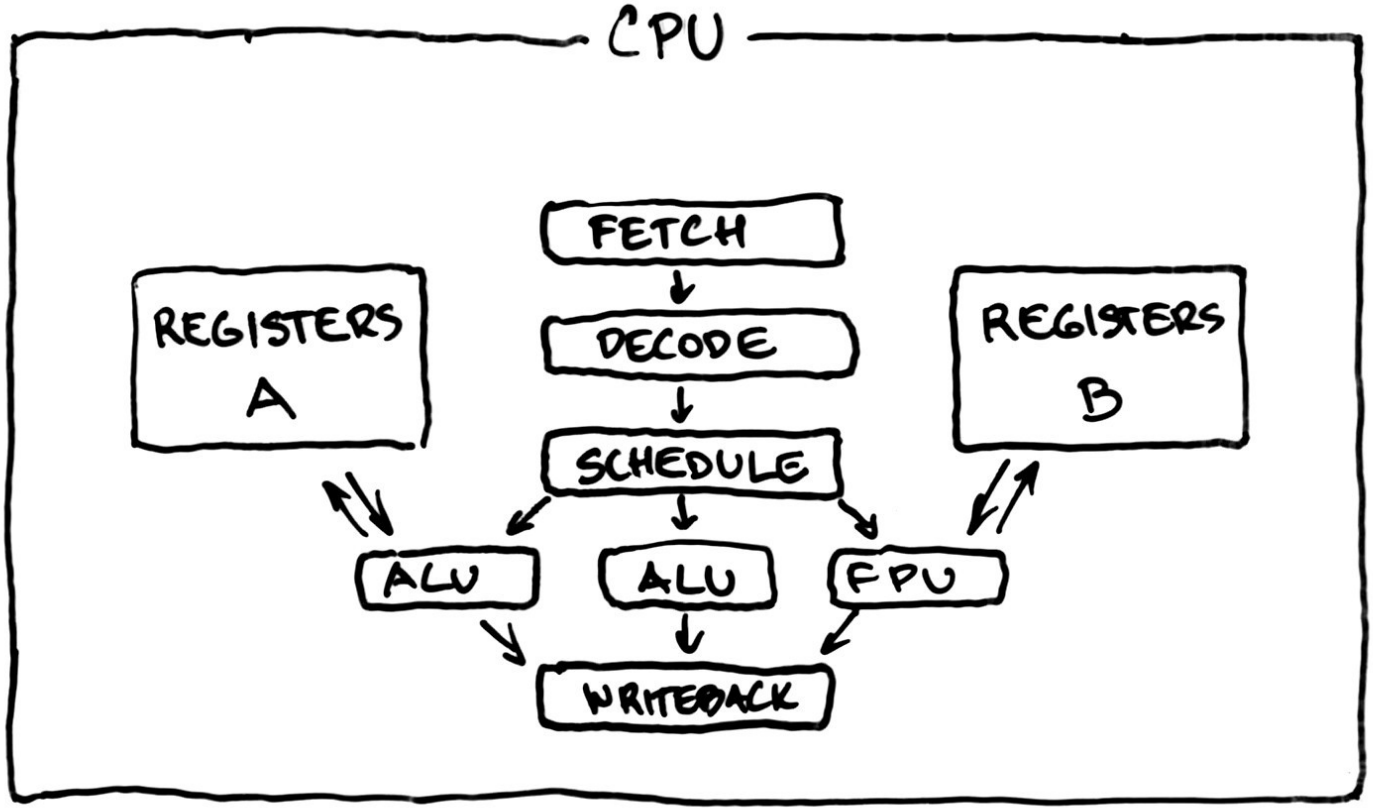
POP SS Attack!



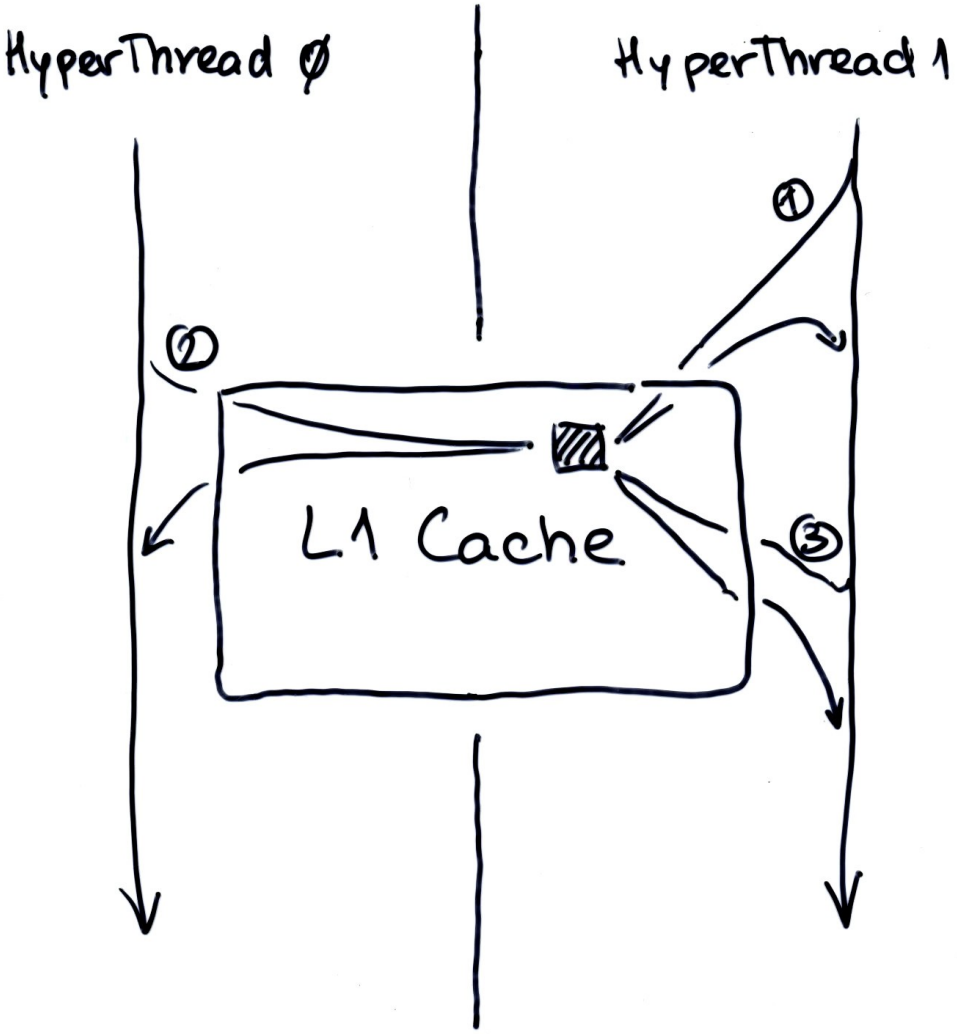
Interrupt stacks

- To overcome POP SS the kernel must use interrupt stack switching for ALL interrupts
- And when it sees an exception it needs to verify whether it may be coming from userspace
- Costs performance
  (actually not that much, only in specific exception handlers)

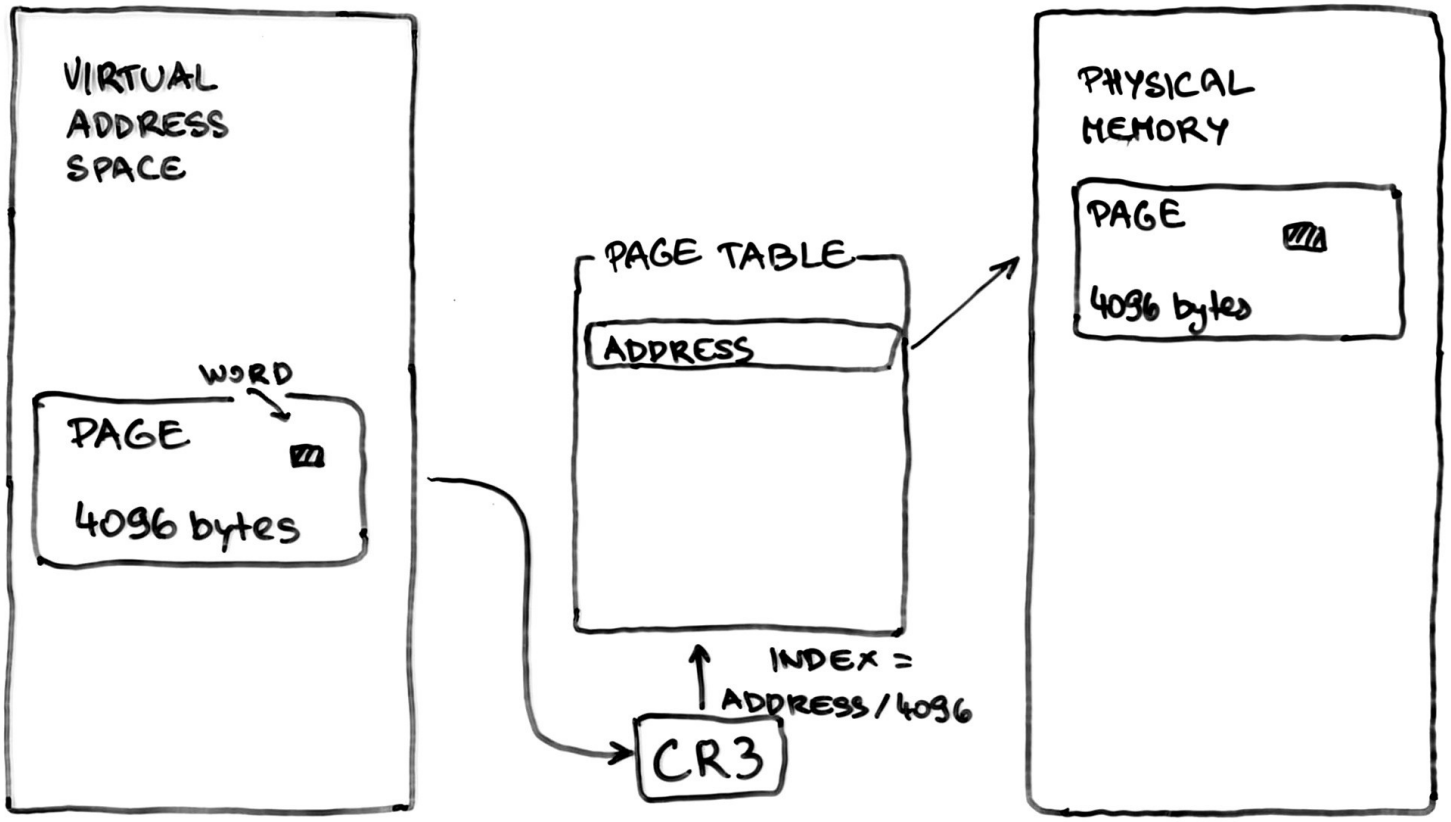
SMT / HT


CPU
FETCH
DECODE
SCHEDULE
REGISTERS
A
REGISTERS
B
ALU
ALU
FPU
WRITEBACK

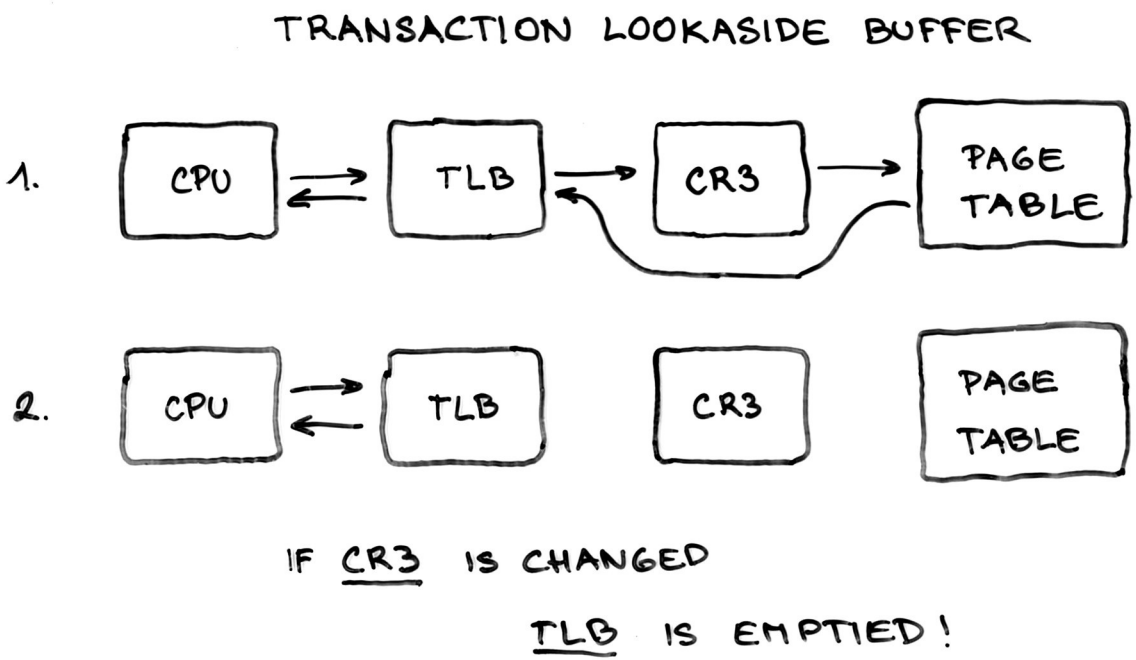
Percival 2005 Attack!



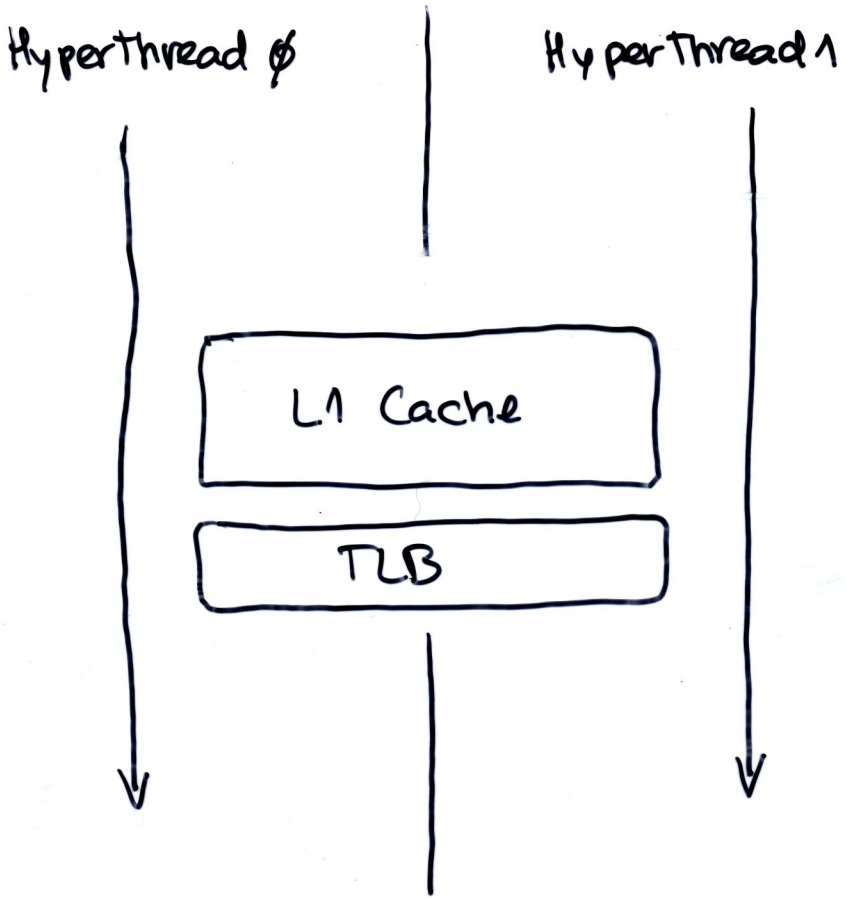
Page table and CR3



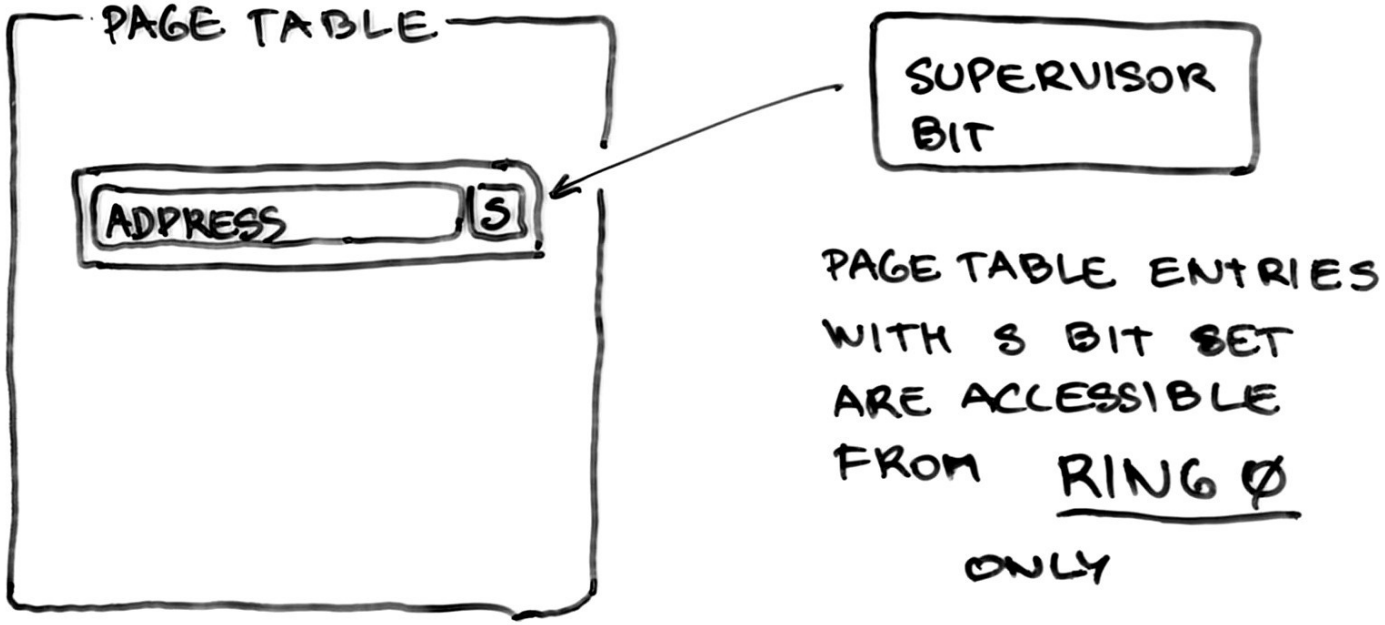
TLB


TRANSACTION LOOKASIDE BUFFER
1.
CPU
TLB
CR3
PAGE TABLE
2.
CPU
TLB
CR3
PAGE TABLE
IF CR3 IS CHANGED
TLB IS EMPTIED!

TLBleed Attack!

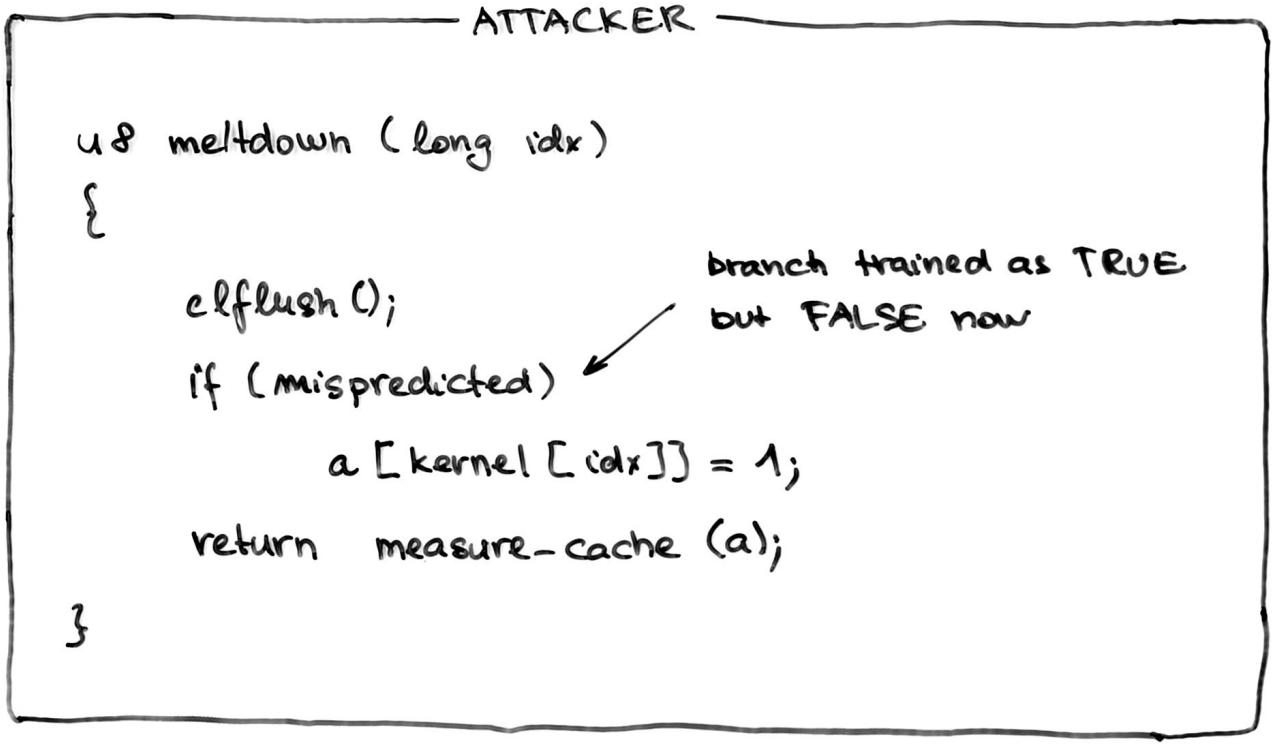


Supervisor bit



Meltdown (v3)

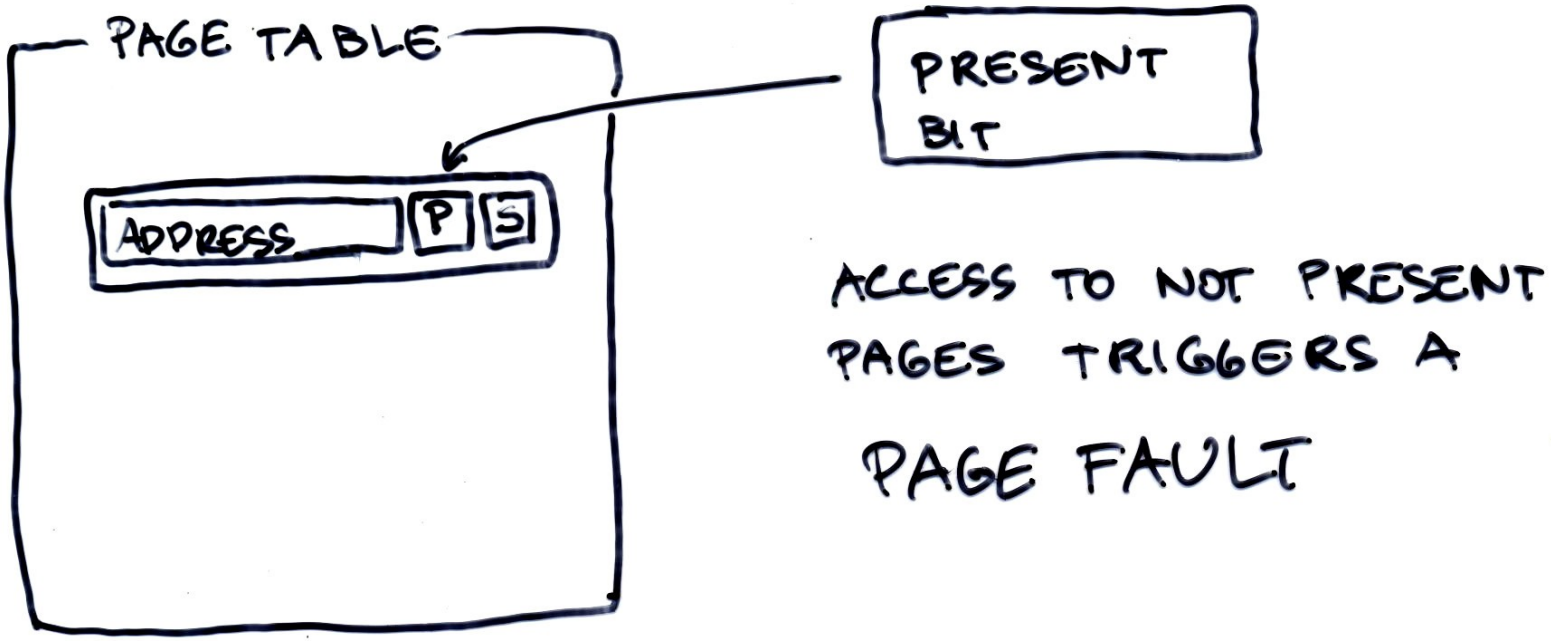
- Intel CPUs only look at the Supervisor bit once the speculation path is confirmed valid (at retirement)



KPTI

- Kernel Page Table Isolation
- Removes the "Supervisor bit" optimization
- Changes CR3 at every syscall
- Modifies stack address at every syscall
- Userspace speculation thus can’t see kernel data at all and can’t access them
- TLB flush at every syscall → syscall entry 4x slowdown

Present bit


PAGE TABLE
ADDRESS
P
S
PRESENT BIT
ACCESS TO NOT PRESENT PAGES TRIGGERS A
PAGE FAULT

L1TF Attack!

- Also known as Foreshadow
- Technique similar to Meltdown

- Can read L1D contents only

- The kernel puts stuff like swap space block numbers into non-present PTEs
- And these are small numbers
- Userspace cannot control them
- And the kernel lives on negative addresses
- So the attack is not all that useful

L1TF Mitigation

- To avoid L1TF from userspace the kernel has to sanitize all non-present PTEs to point to non-existent memory instead just low addresses
- This is done easily by putting 1’s in some top PTE address bits

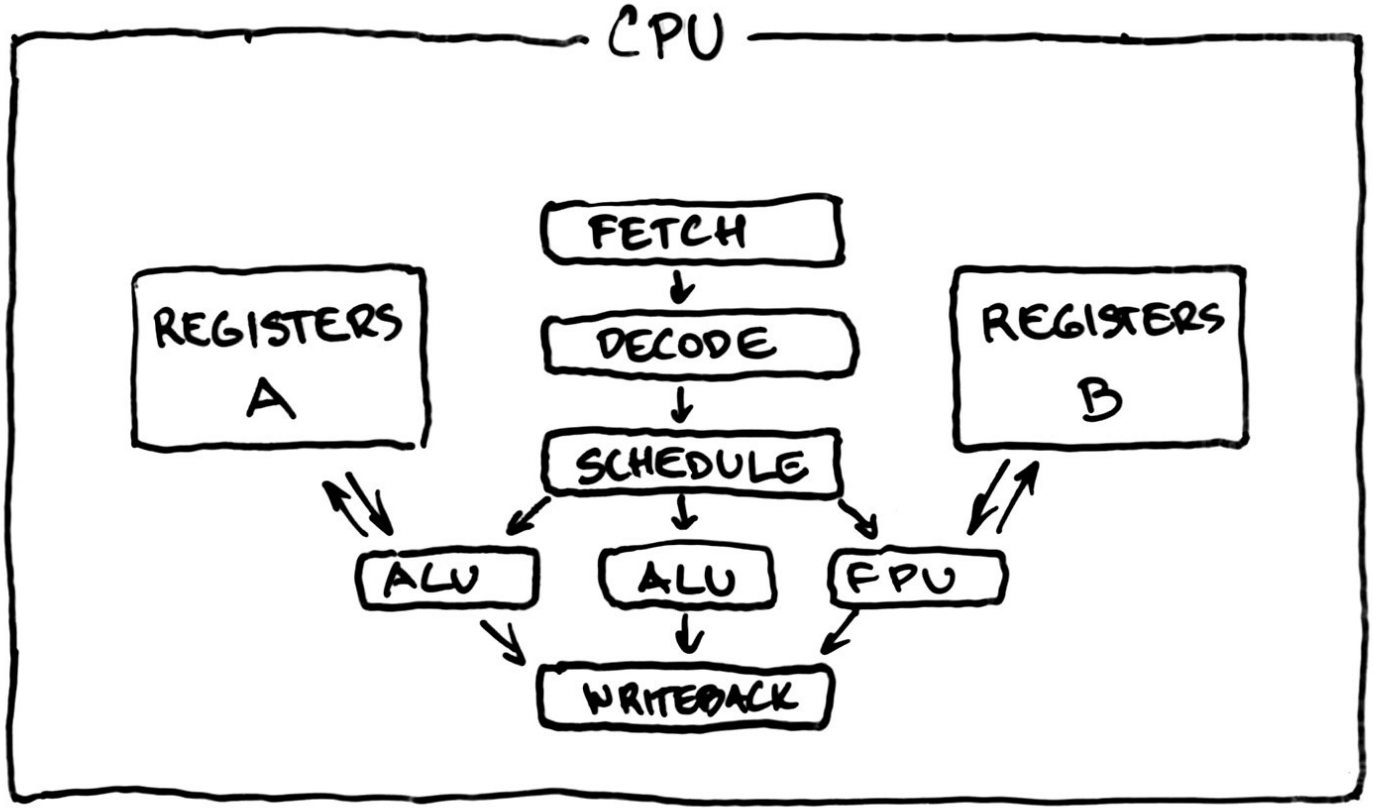
L1TF and Virtualization

- A Virtual machine is in control of page tables
- And can put ANYTHING in them
- And so it can read any address via L1D Cache
  - from other guests
  - and from the host
- And EPTs don’t help (they’re just ignored, too)

L1D Flushing

- To prevent L1TF from a VM, we must flush L1D before every VMENTER
- This hurts performance quite a bit

SMT / HT


CPU
FETCH
DECODE
SCHEDULE
REGISTERS
A
REGISTERS
B
ALU
ALU
FPU
WRITEBACK

VM L1TF and HyperThreading

- L1D is shared between hyperthreads
- If two VMs run on two hyperthreads, one can snoop the other
- If one hyperthread runs a VM and the other the host, the VM can snoop the host
- And there is nothing the host can do about that
  → NO MITIGATION
- The only way is to disable HT
  → 20-50% performance impact

Really nothing?

- We could disable EPT and do it in software
  - That's actually worse than disabling HT
- We could do ganged scheduling
  - That's hard and only solves the VM-VM scenario
- But it could be enough if the Host doesn't ever handle any valuable data
  - In a thin-hypervisor scenario, if no data pass through the hypervisor, the service domain (Dom0 in Xen, etc) is bound to a separate core and passes data via ring buffers to VMs, VMs never share cores, yes, then perhaps you can leave HT on.
  - Hyper-V seems to be able to do that, Xen might in the future, KVM probably never will.


SUSE®
We adapt. You succeed.