The Roman Xpl0it

TRANSIENT EXECUTION ATTACKS

Episode II

ATTACK OF THE ZOMBIES

by pietroborrello

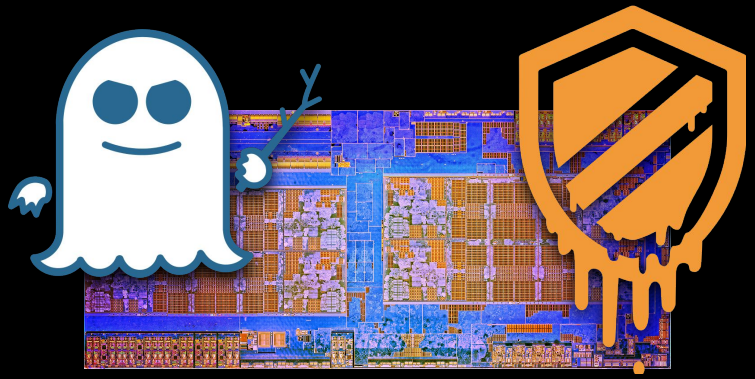
A long time ago, in a galaxy far, far away...

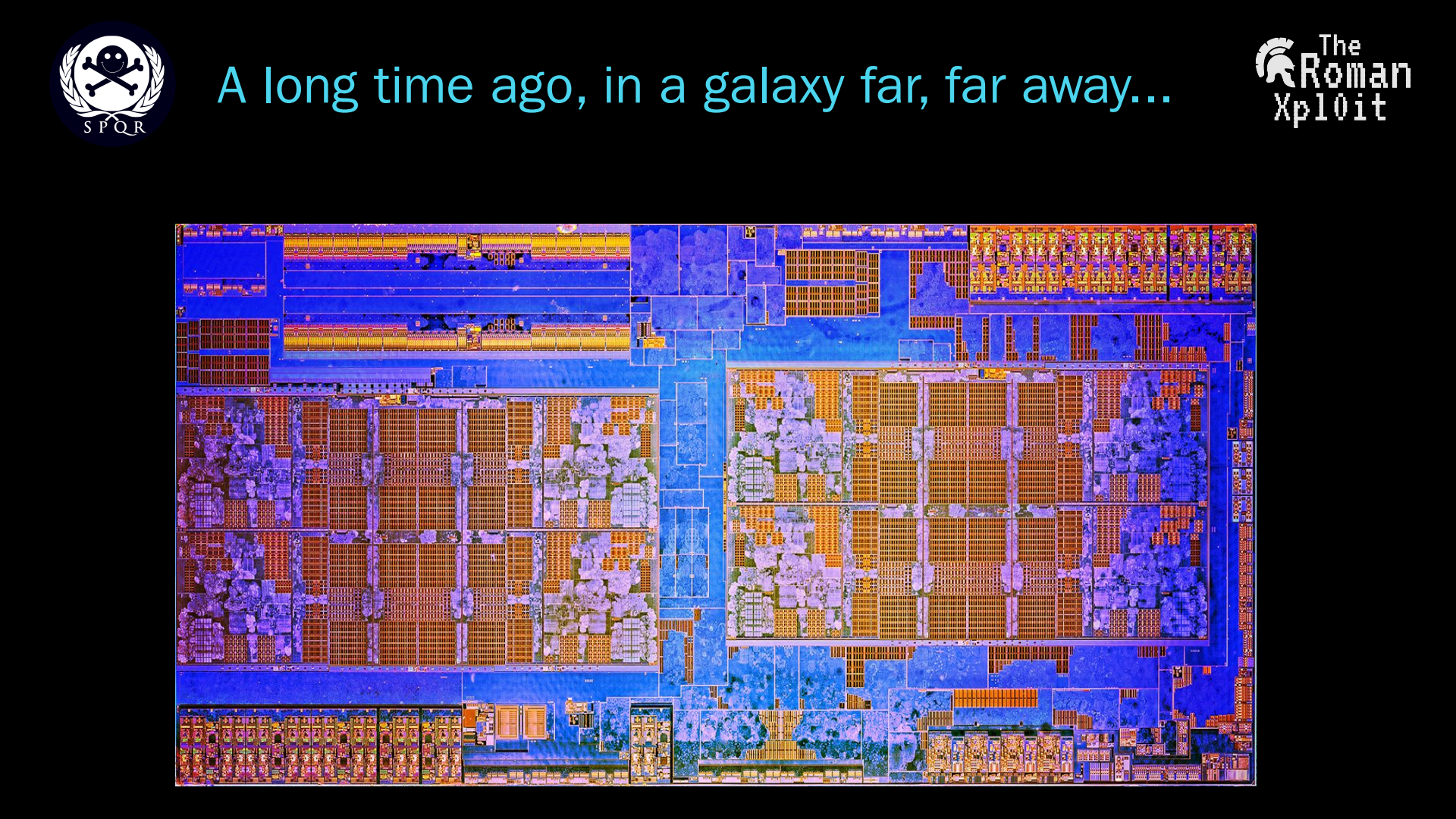
The Roman Xp10it

We have seen how modern CPU leave traces of what they did during transient execution due to microarchitectural optimizations:

- **Meltdown**: Exceptions during execution are enforced lazily
  ⇒ There is a small window where the result of faulty instructions is accessible (e.g. kernel memory content!)

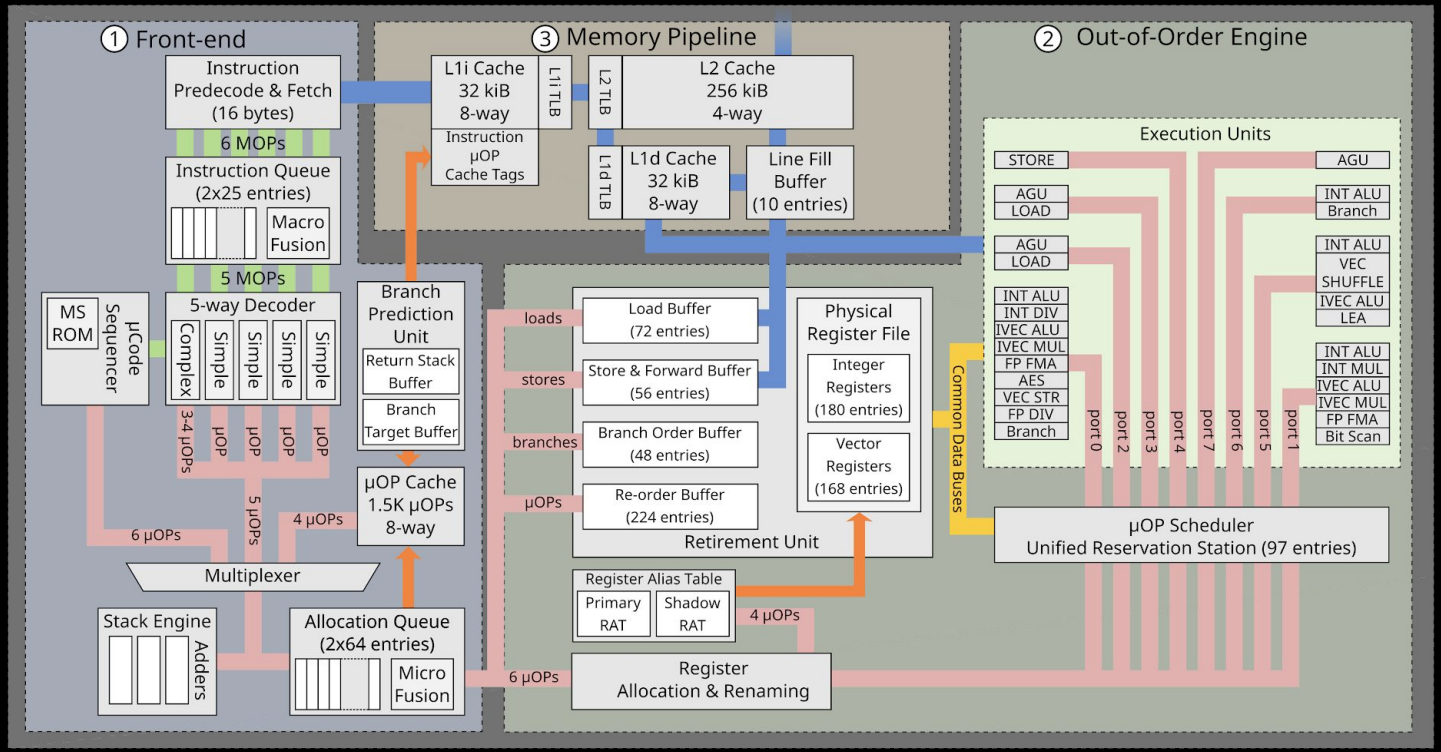
- **Spectre**: Predicted instructions are executed transiently
  ⇒ There is a small window of instructions that shouldn't be executed, due to misprediction





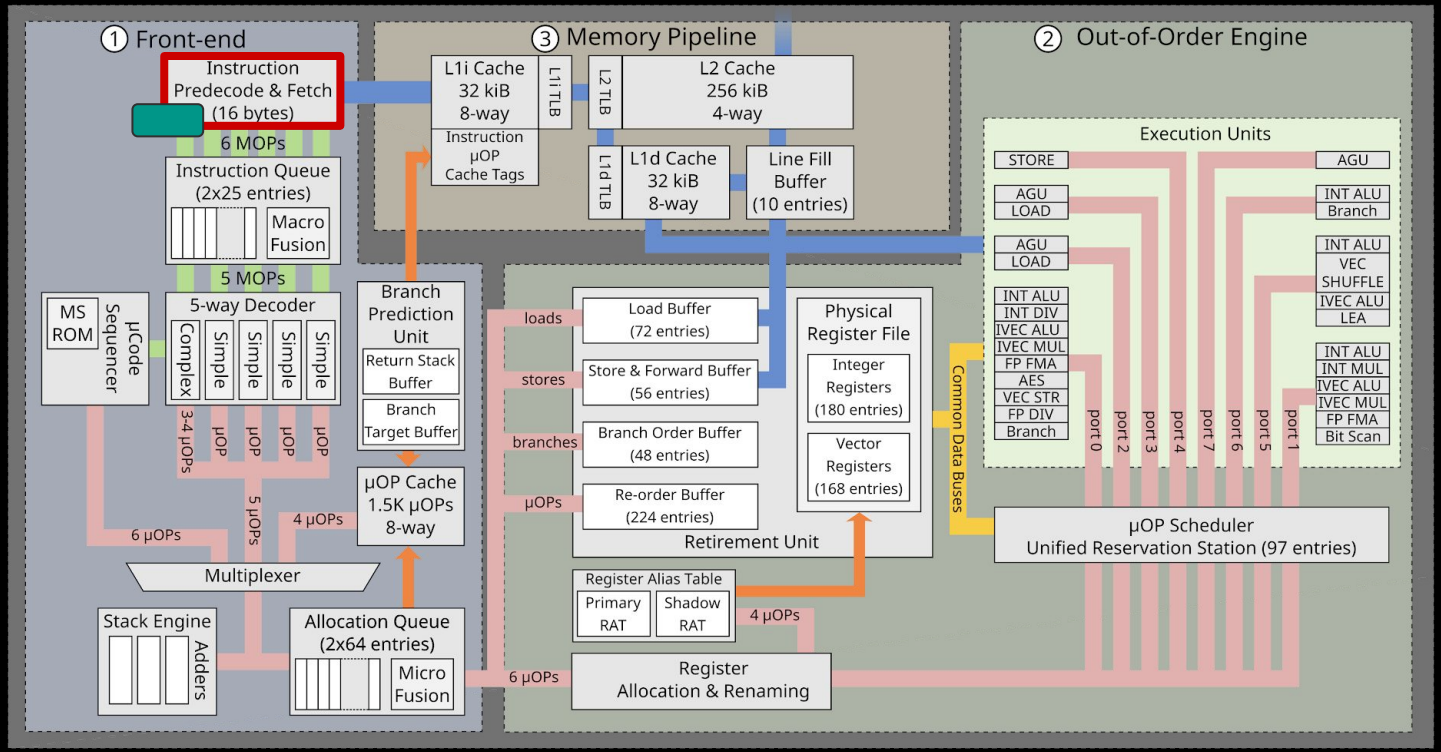
A long time ago, in a galaxy far, far away...

A long time ago, in a galaxy far, far away...



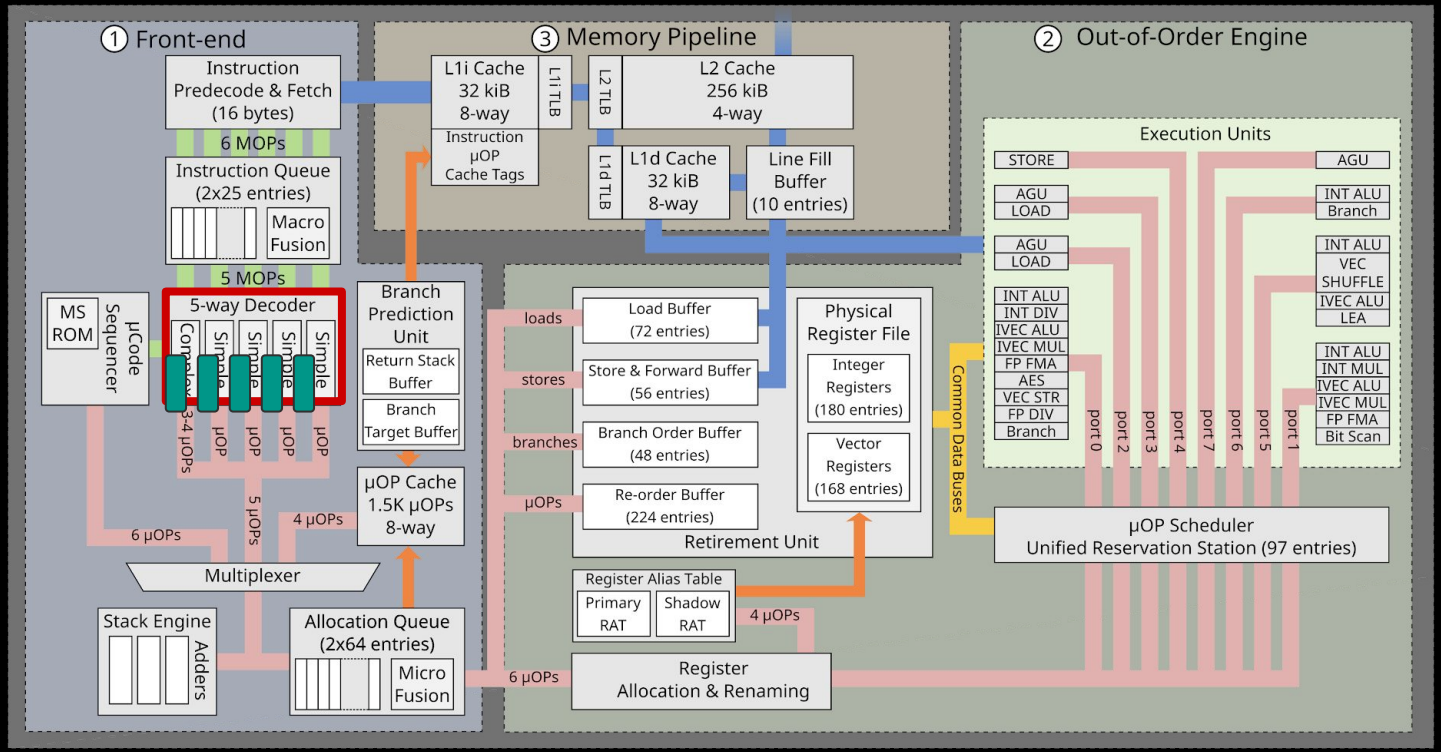
A long time ago, in a galaxy far, far away...

```
add qword ptr [rax], rbx
```



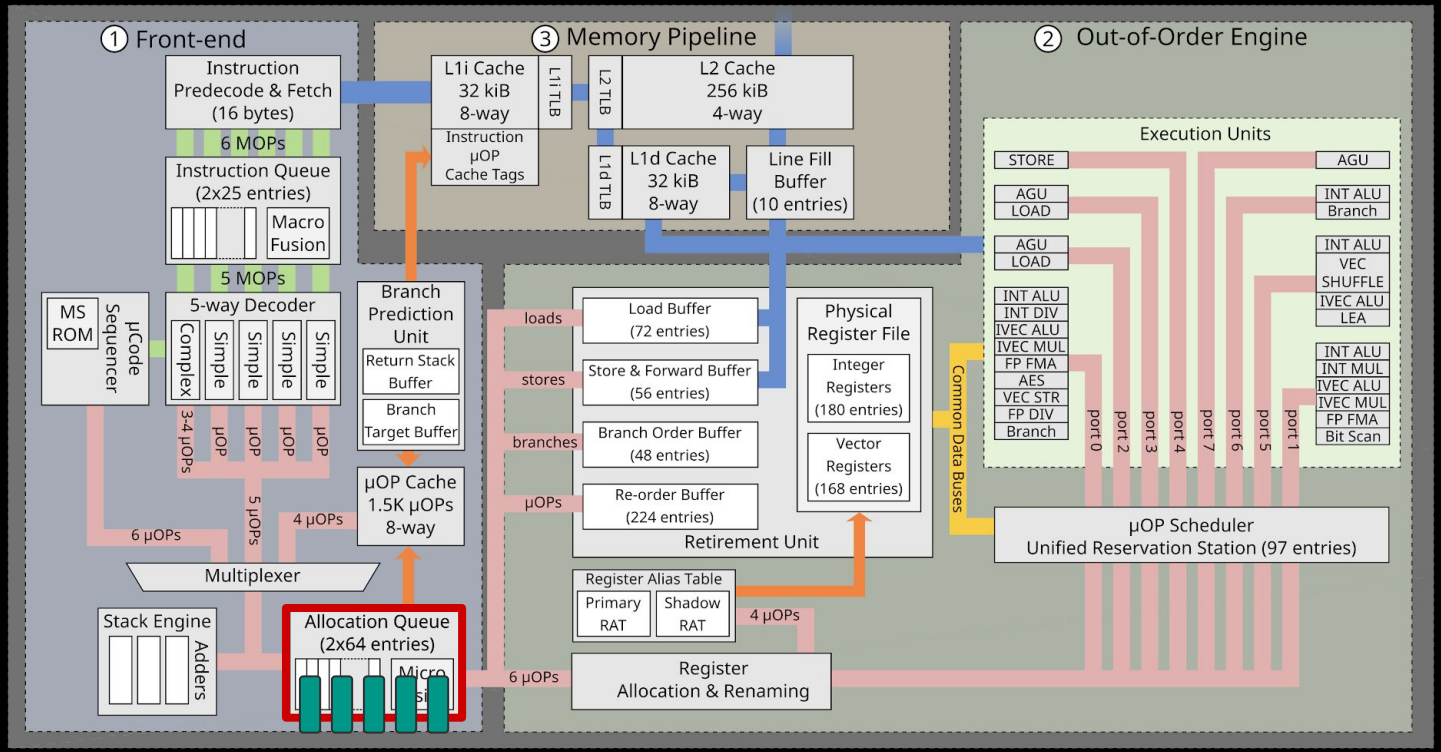
A long time ago, in a galaxy far, far away...

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



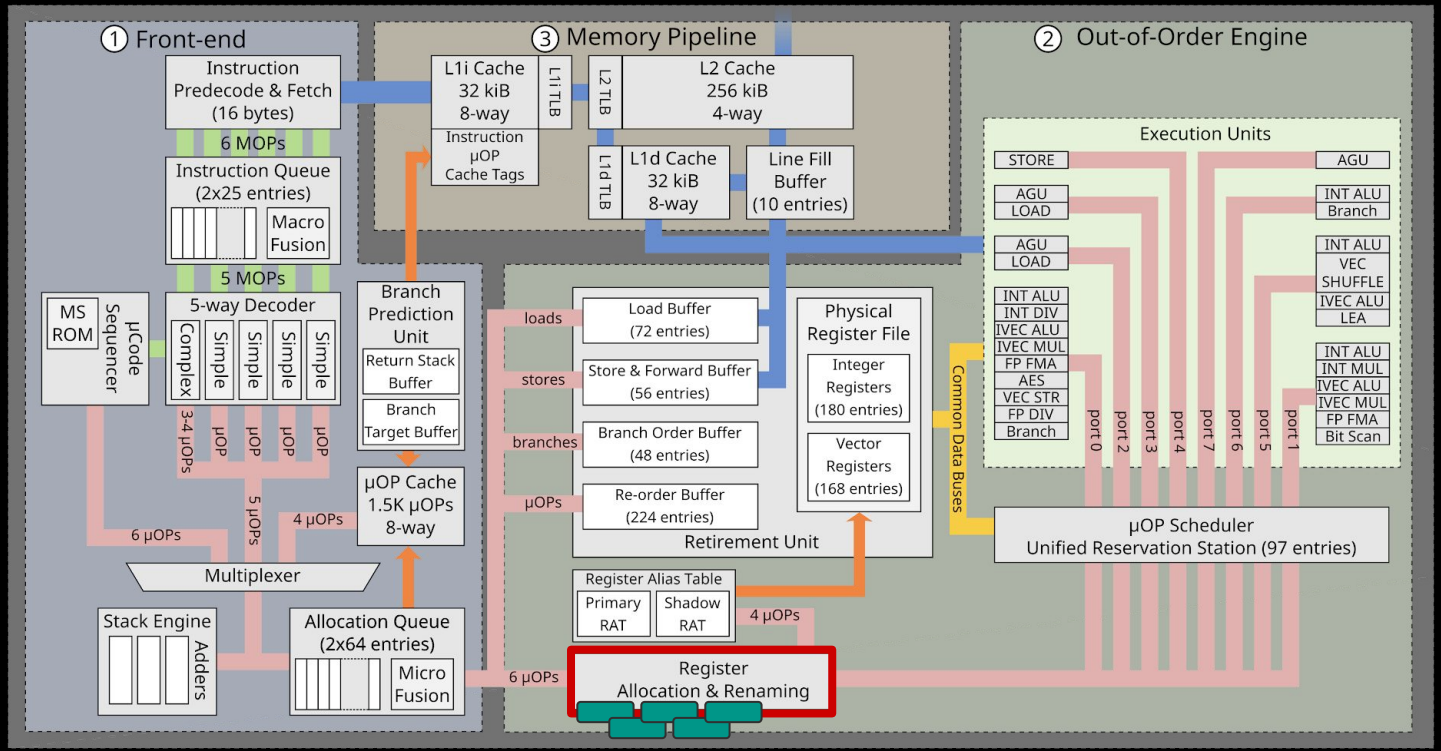
A long time ago, in a galaxy far, far away...

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



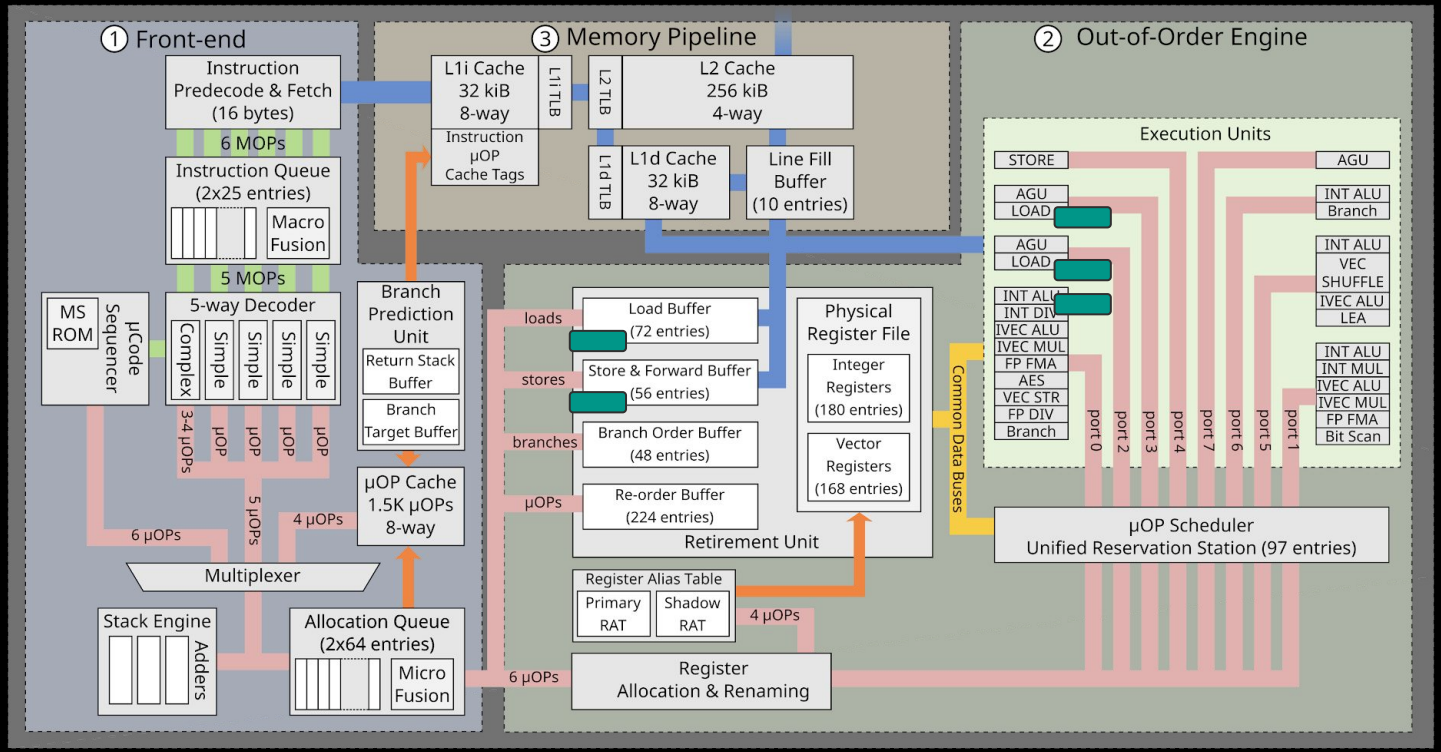
A long time ago, in a galaxy far, far away...

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



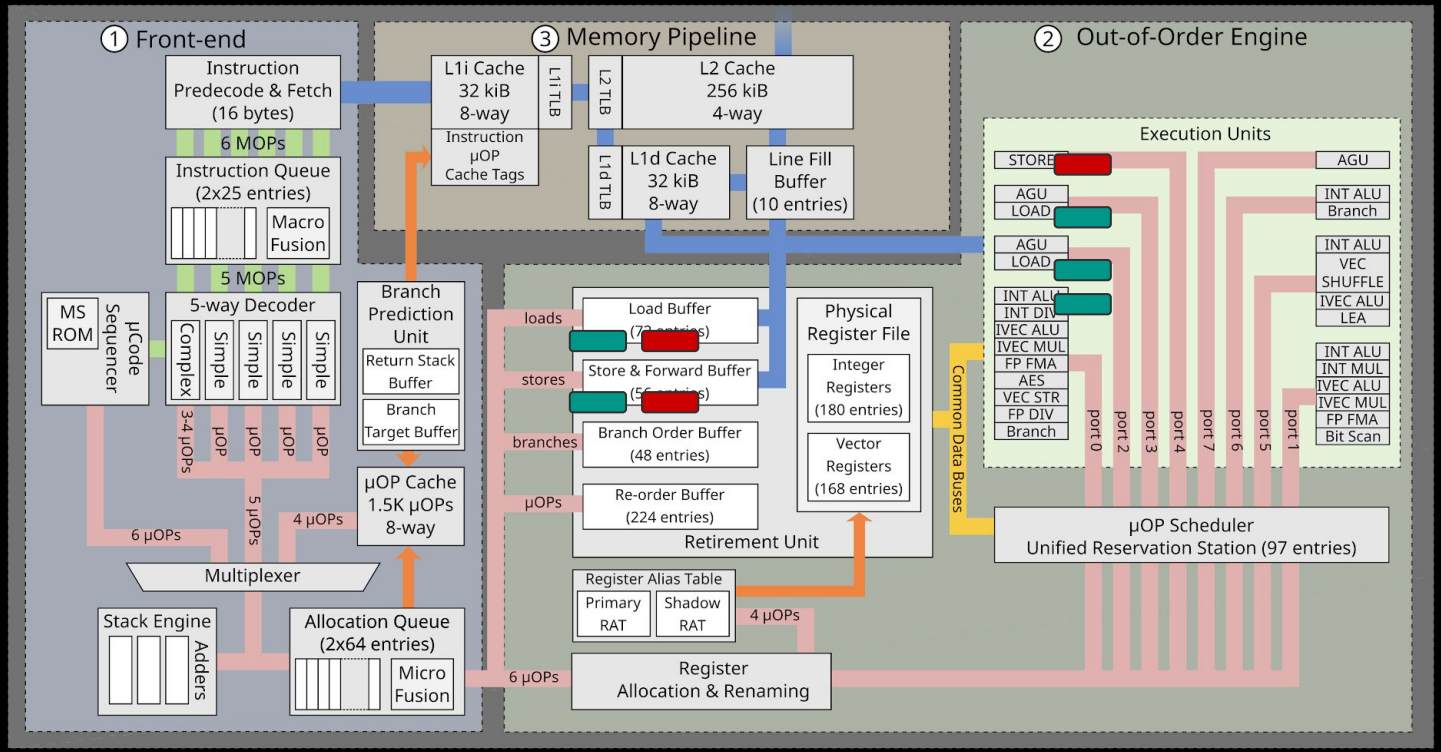
A long time ago, in a galaxy far, far away...

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



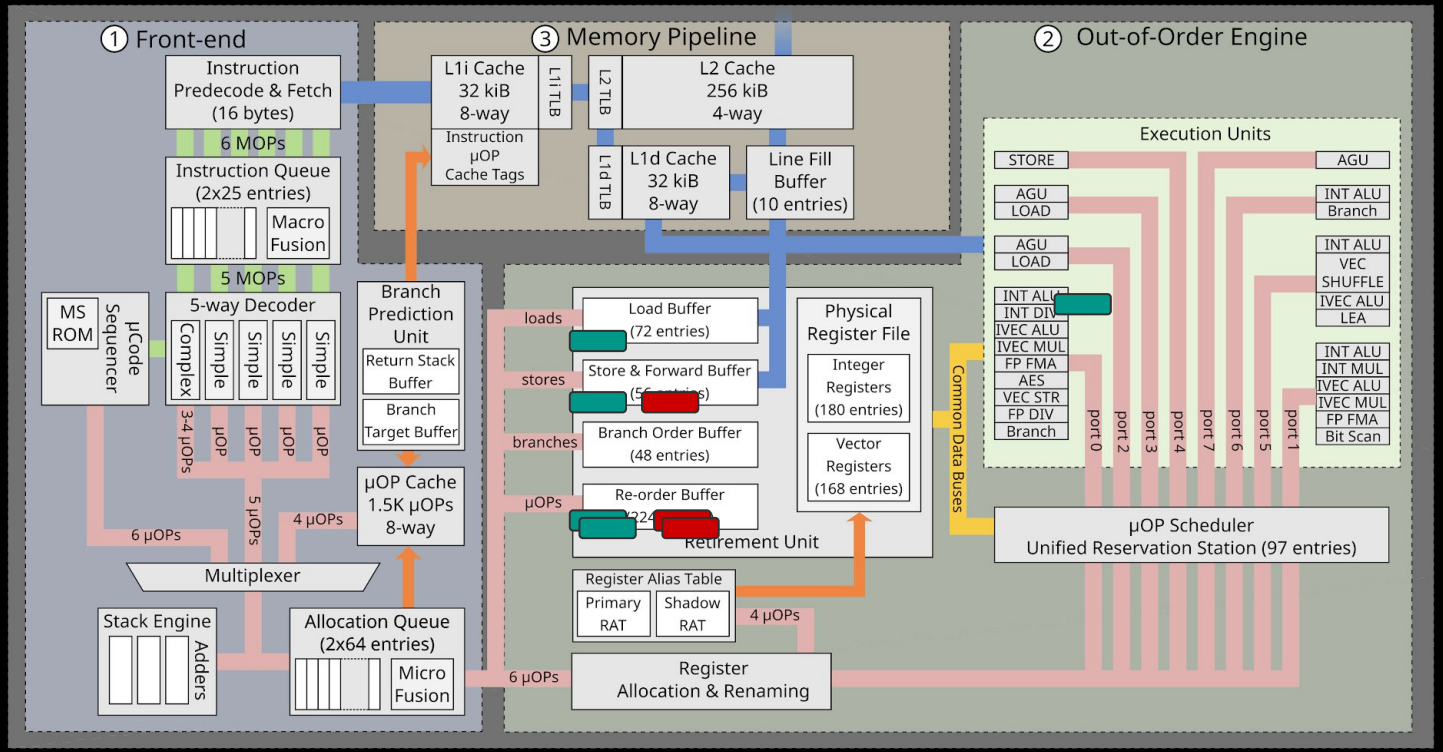
A long time ago, in a galaxy far, far away...

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



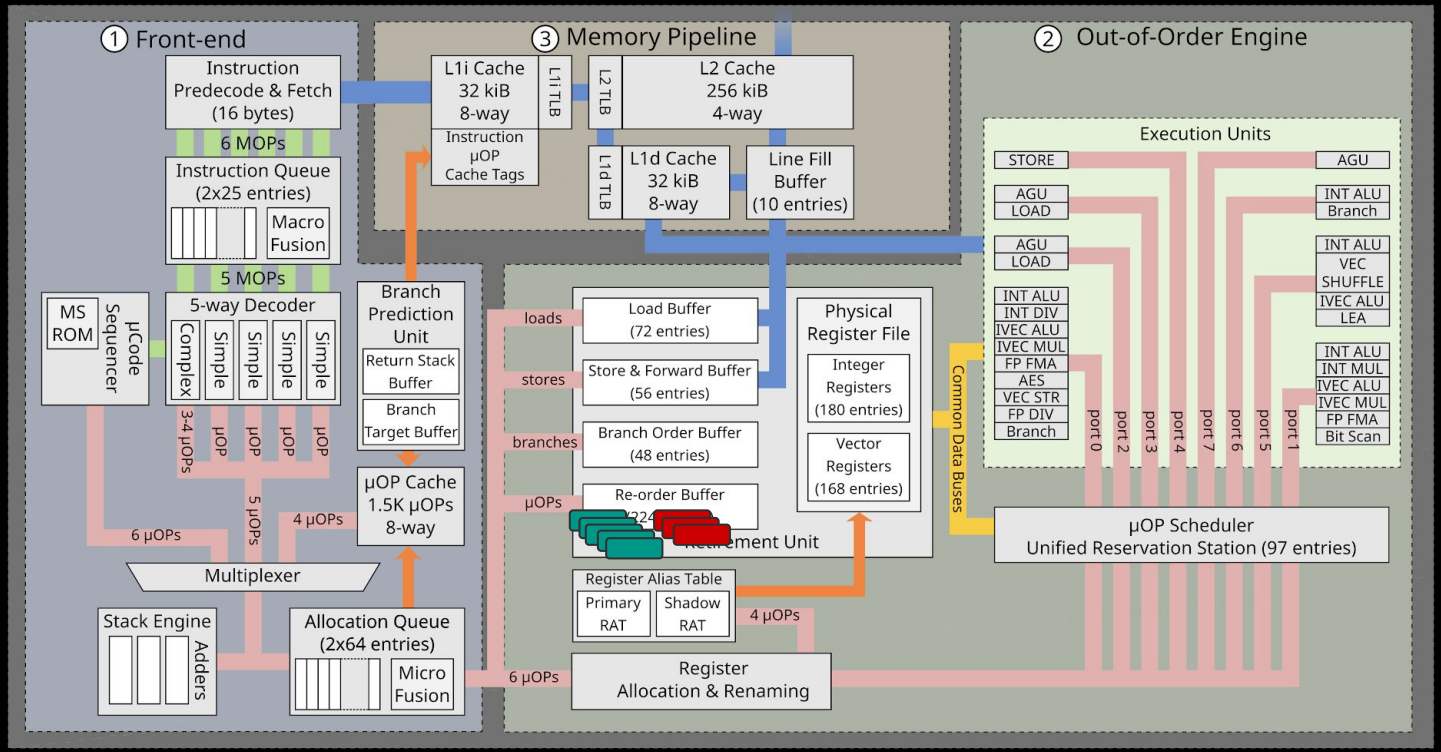
A long time ago, in a galaxy far, far away...

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



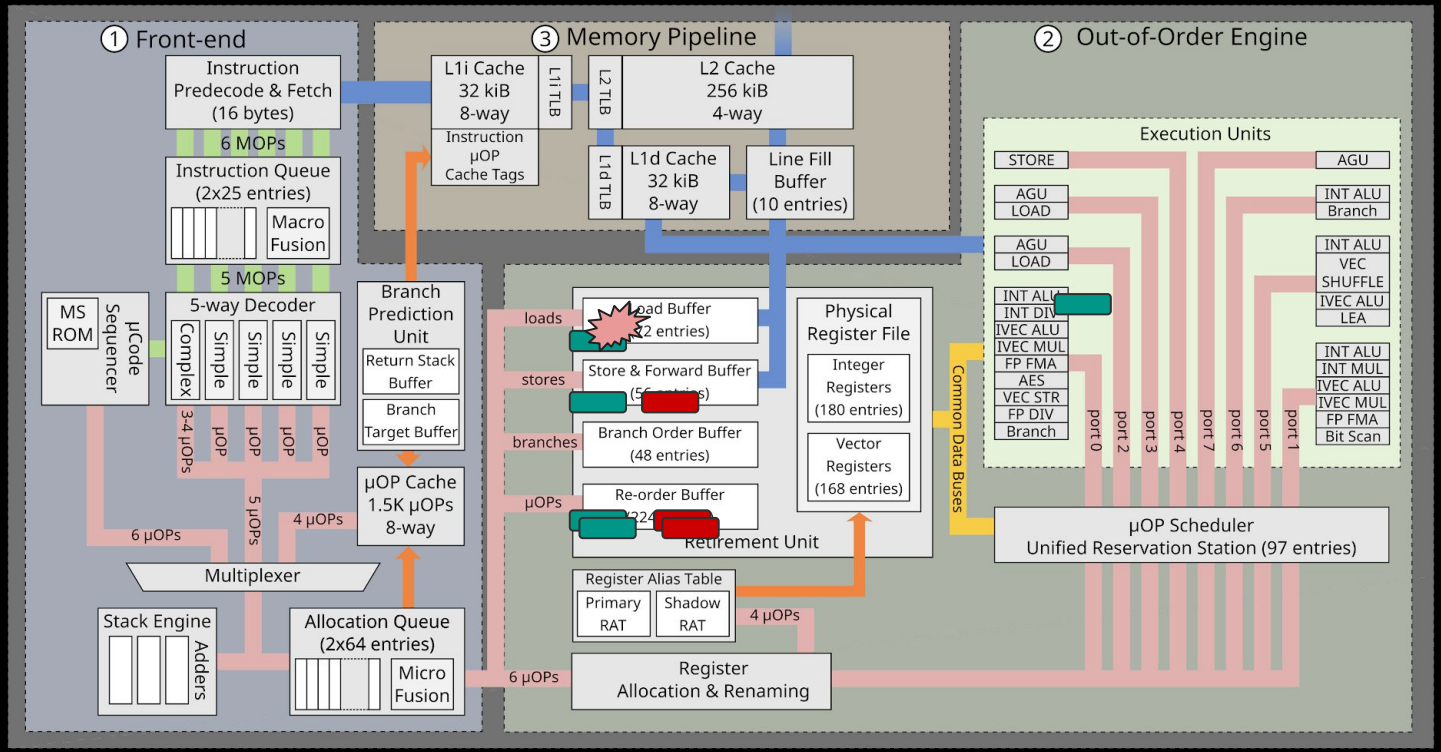
A long time ago, in a galaxy far, far away...

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



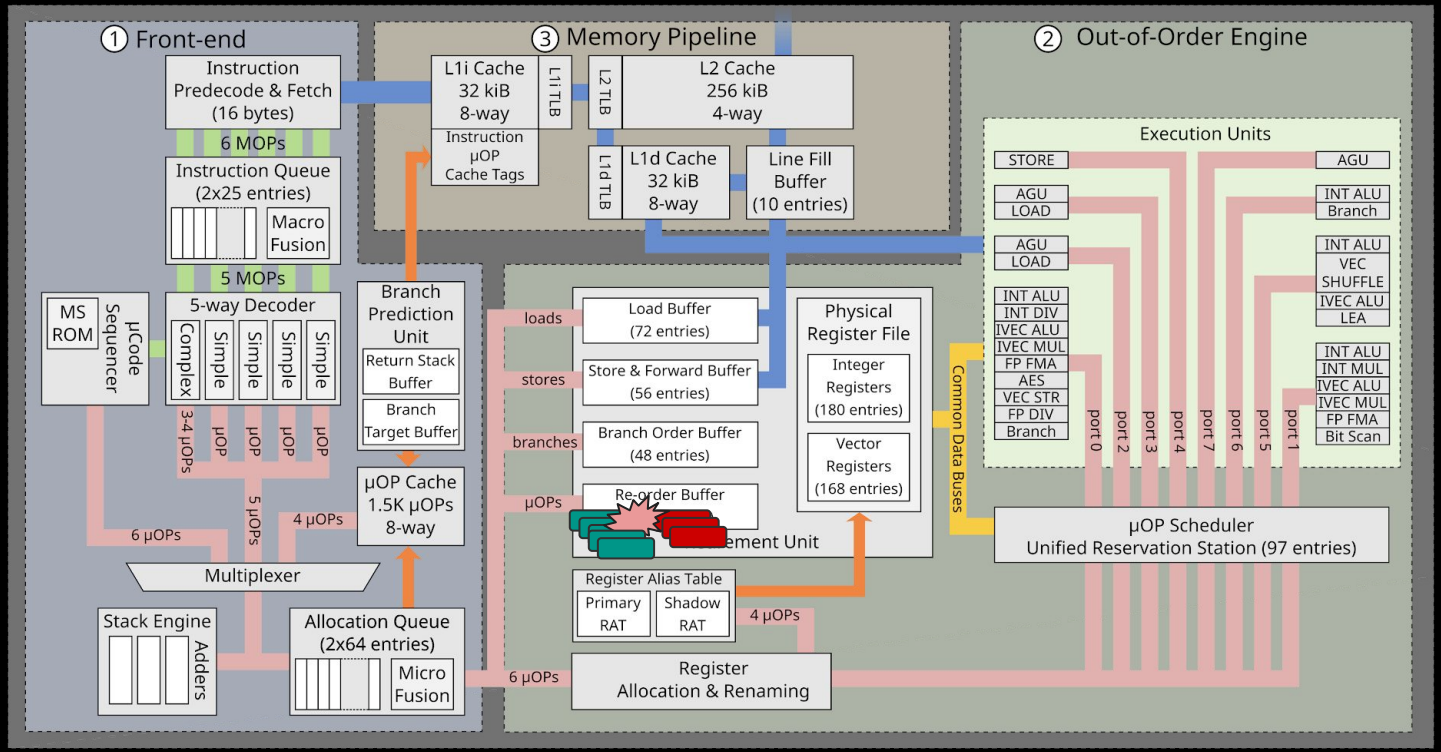
A long time ago, in a galaxy far, far away...

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



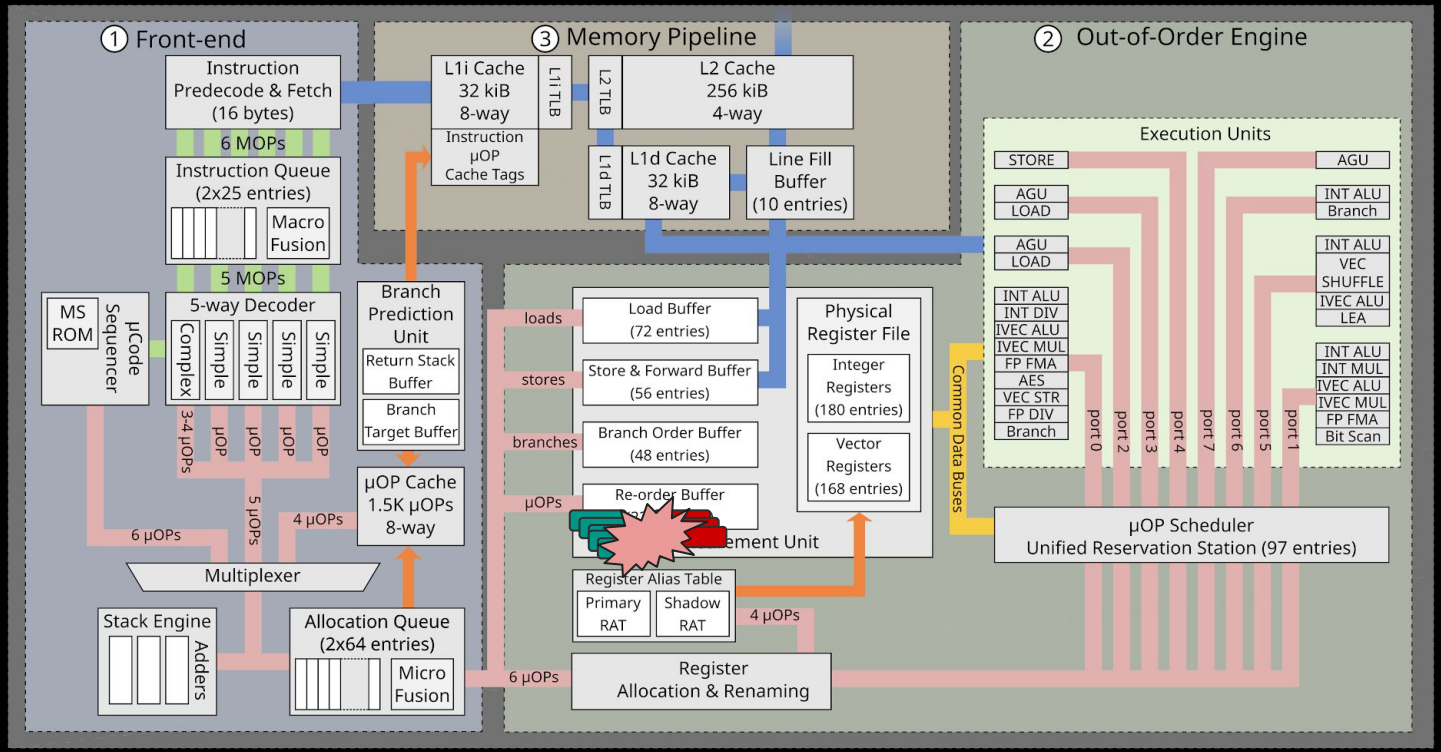
A long time ago, in a galaxy far, far away...

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



A long time ago, in a galaxy far, far away...

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



A long time ago, in a galaxy far, far away...

The Roman Xploit

Meltdown idea:

- Reading Kernel Memory rises a General Protection Fault
- But we can access the value during transient execution!

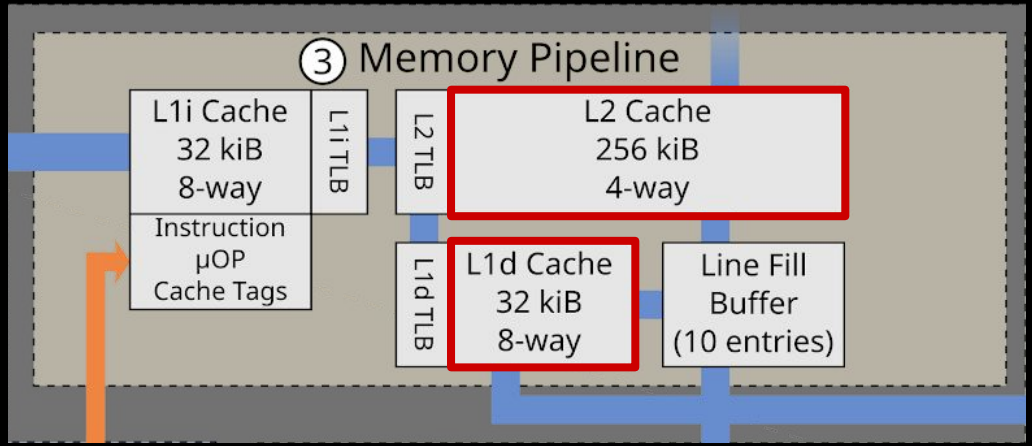
1. `char array[256 * 4096]`
2. `flush all array cache lines`
3. `read kernel byte into X`
4. `tmp = array[X * 4096]`

1. `handle SIGSEGV`
2. `for(i = 0; i < 256; i++)`
   `measureTime(array[i*4096])`
3. `The index with fastest access corresponds to X`

Episode II

The Roman Xploit

- Meltdown attacks leak secrets loading them into the L1 and L2 caches
- The secrets values are brought into the caches and then accessed by faulty instructions

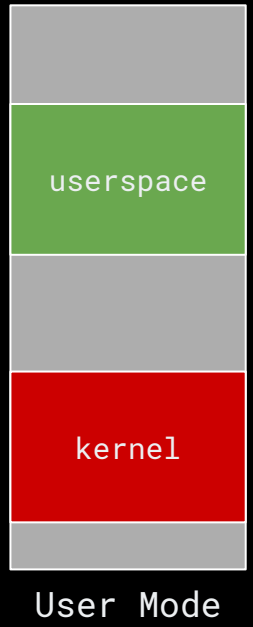


Episode II

The Roman Xploit

- Meltdown attacks leak secrets loading them into the L1 and L2 caches
- The secrets values are brought into the caches and then accessed by faulty instructions

- MITIGATION:
  isolate or mask valuable addresses
  - e.g. unmap kernel addresses from userspace



Episode II

The Roman Xploit

- Meltdown attacks leak secrets ~~loading them into the L1 and L2 caches~~
- The secrets values are ~~brought into the caches~~ and then accessed by faulty instructions

- MITIGATION:
  isolate or mask valuable addresses
  - e.g. unmap kernel addresses from userspace

Episode II

The Roman Xpl0it

- Meltdown attacks leak secrets ~~loading them into the L1 and L2 caches~~
- The secrets values are ~~brought into the caches~~ and then accessed by faulty instructions

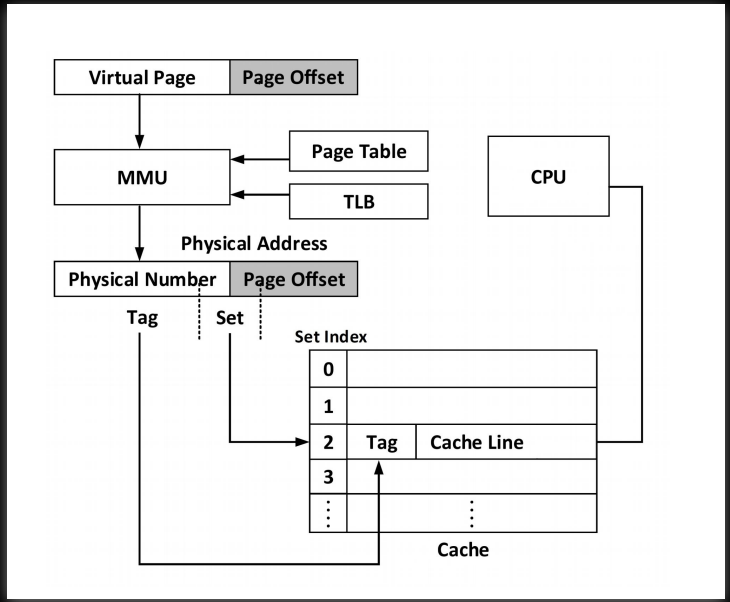
- MITIGATION:
  isolate or mask valuable addresses
  - e.g. unmap kernel addresses from userspace

- Maybe… we should understand how caches really work

Caches 101

- This is how we always looked at caches

1. Virtual address ⇒ Physical address
2. Get Tag and Cache Set
3. Take Page Offset
4. Search for Tag in the Cache Set:
   a. HIT: get data
   b. MISS: load data from cache hierarchy

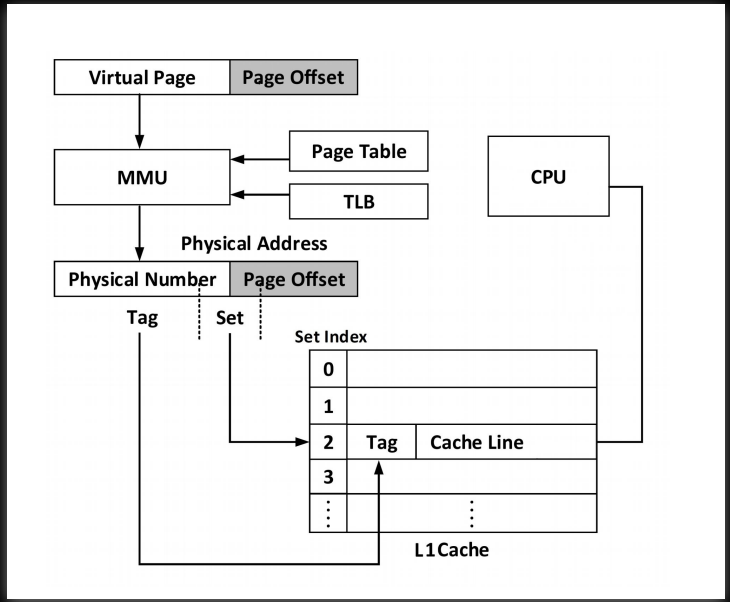


Caches 101

- This is how we always looked at caches

1. Virtual address ⇒ Physical address
2. Get Tag and Cache Set
3. Take Page Offset
4. Search for Tag in the Cache Set:
   a. HIT: get data
   b. MISS: load data from cache hierarchy

Wait, wait, what does it mean to load data from cache hierarchy?!



L1 Cache Misses 101

The Roman Xpl0it

- Let’s focus just on L1 cache!
- L1 cache will send memory request to the L2 cache to load the requested address
- Reserve cache line to host future data

  Then what?

  a. Wait for completion & Block other loads/stores
  b. Try to optimize it

L1 Cache Misses 101

The Roman Xp10it

- Let’s focus just on L1 cache!
- L1 cache will send memory request to the L2 cache to load the requested address
- Reserve cache line to host future data

  Then what?
  a. Wait for completion & Block other loads/stores
  b. Try to optimize it

Super Fast L1 Cache Misses

- Send memory request to the L2 cache
- Reserve a Line Fill Buffer entry to host future data
  Now the cache is free!
- Continue serving other requests

- Eventually the data will be in the Line Fill Buffer entry
  Now move data into the right cache line and complete the memory request

Super Fast L1 Cache Misses

- Send memory request to the L2 cache
- Reserve a LINE FILL BUFFER entry to host future data
  Now the cache is free!
- Continue serving other requests

- Eventually the data will be in the LINE FILL BUFFER entry
  Now move data into the right cache line and complete the memory request

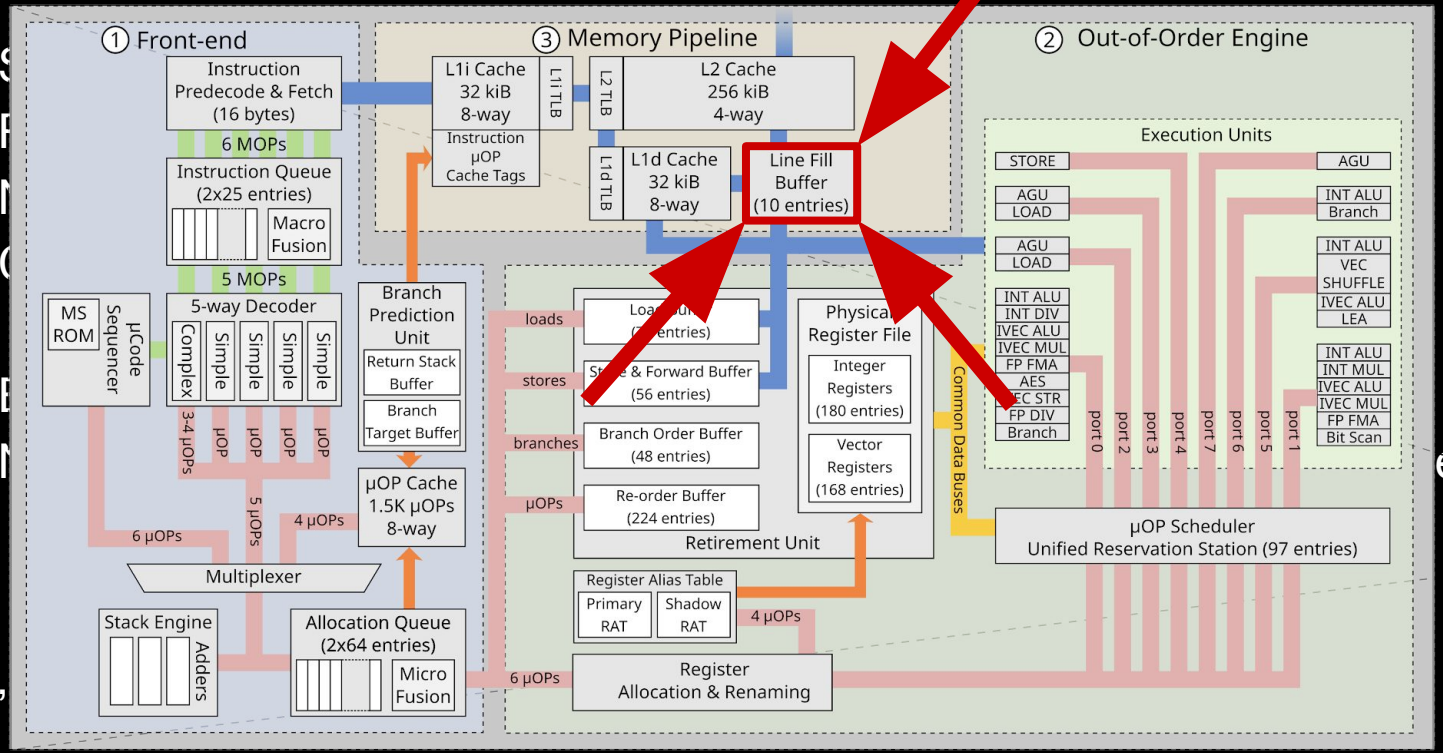
Wait, WTF is LINE FILL BUFFER ?!

No one ever mentioned them!

Super Fast L1 Cache Misses

- S
- F
- M
- C
- E
- M

equest



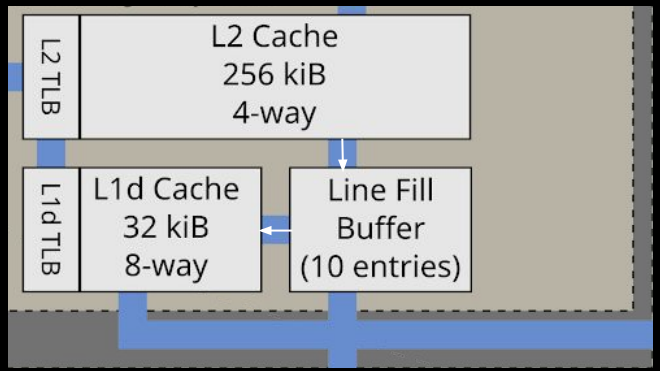
Wait,

No one ever mentioned them!

Let's meet the Line Fill Buffer

The Roman Xploit

- It's an internal CPU buffer, between L1d and L2 caches
- Temporary place to store data while its loading from memory
- Each L1 cache miss will have a corresponding LFB entry where the L2 cache will load the data



Let’s meet the Line Fill Buffer

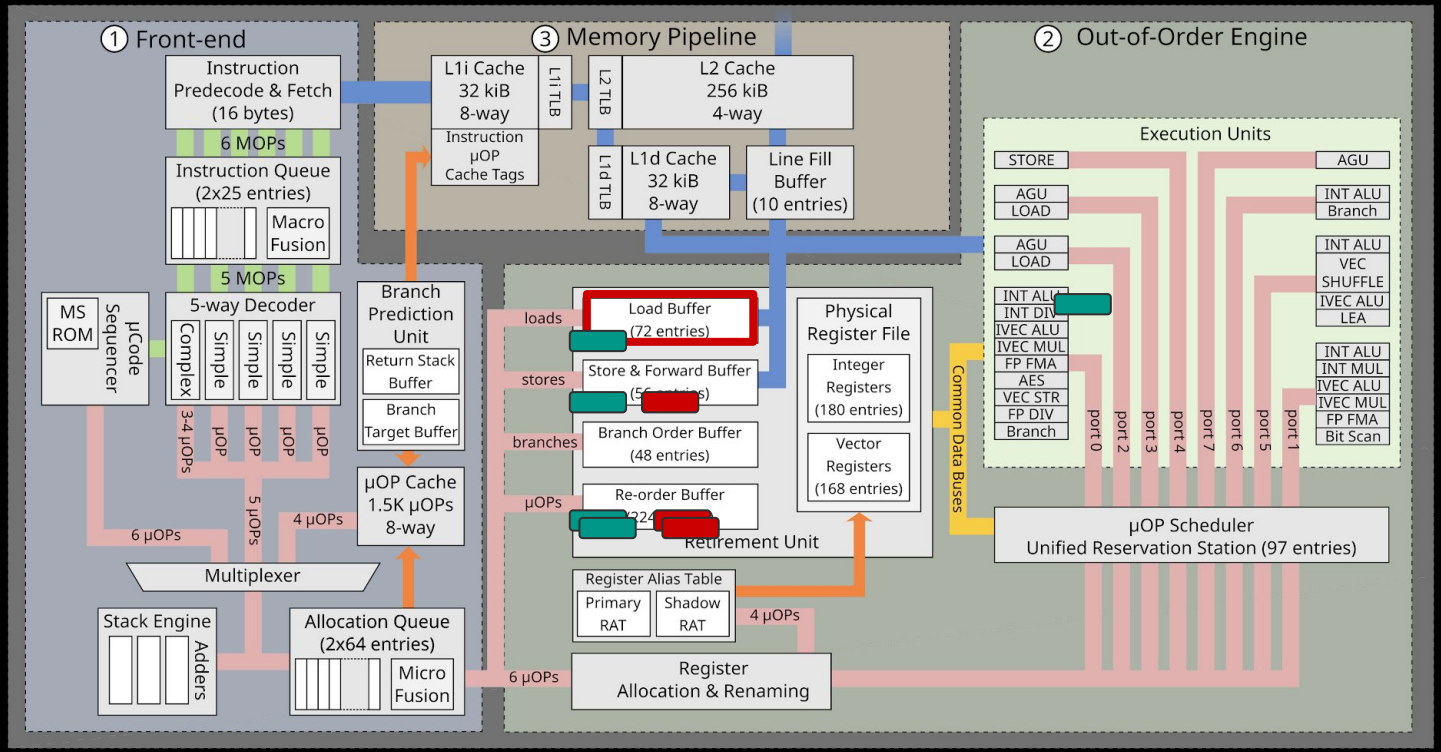
The Roman Xpl0it

- It’s an internal CPU buffer, between L1d and L2 caches
- Temporary place to store data while its loading from memory
- Each L1 cache miss will have a corresponding LFB entry where the L2 cache will load the data

- Just an optimization to increase memory throughput

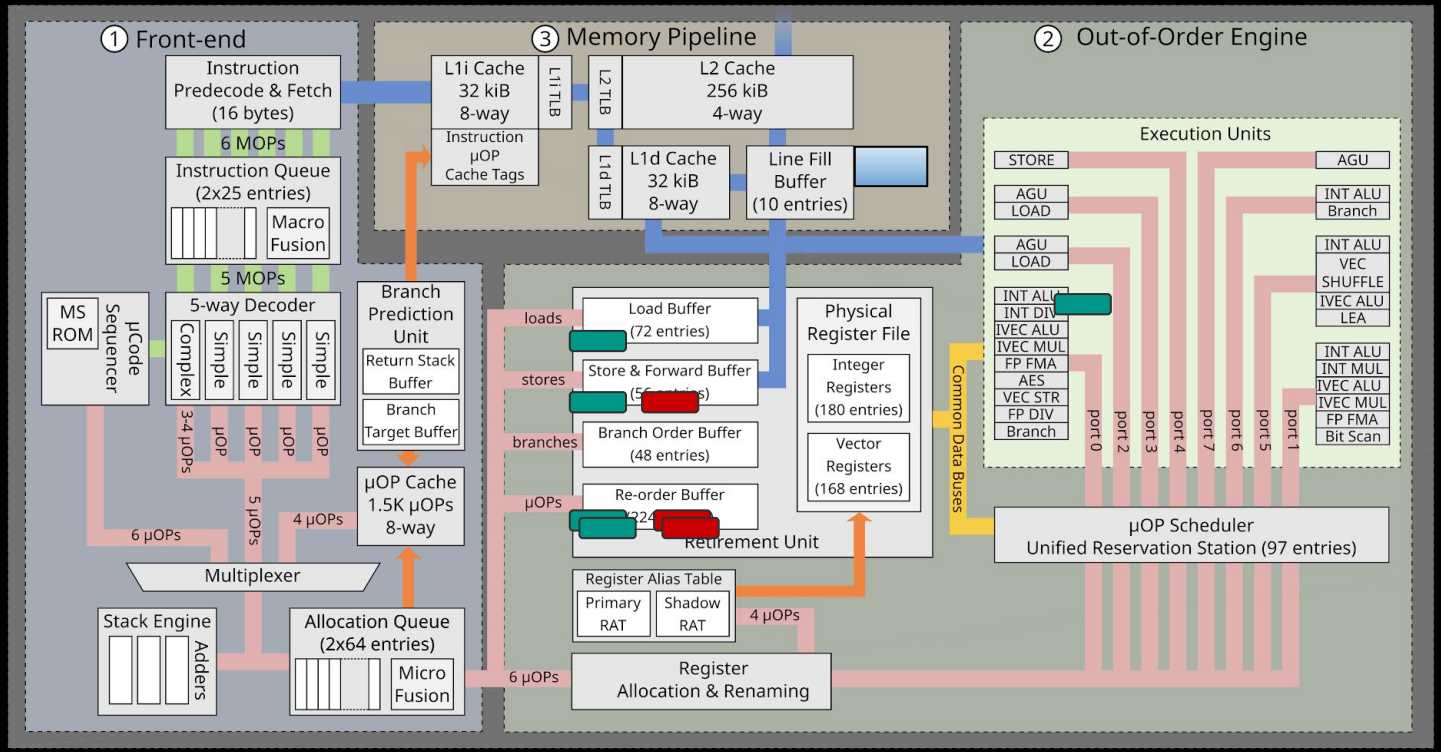
Line Fill Buffers in action

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



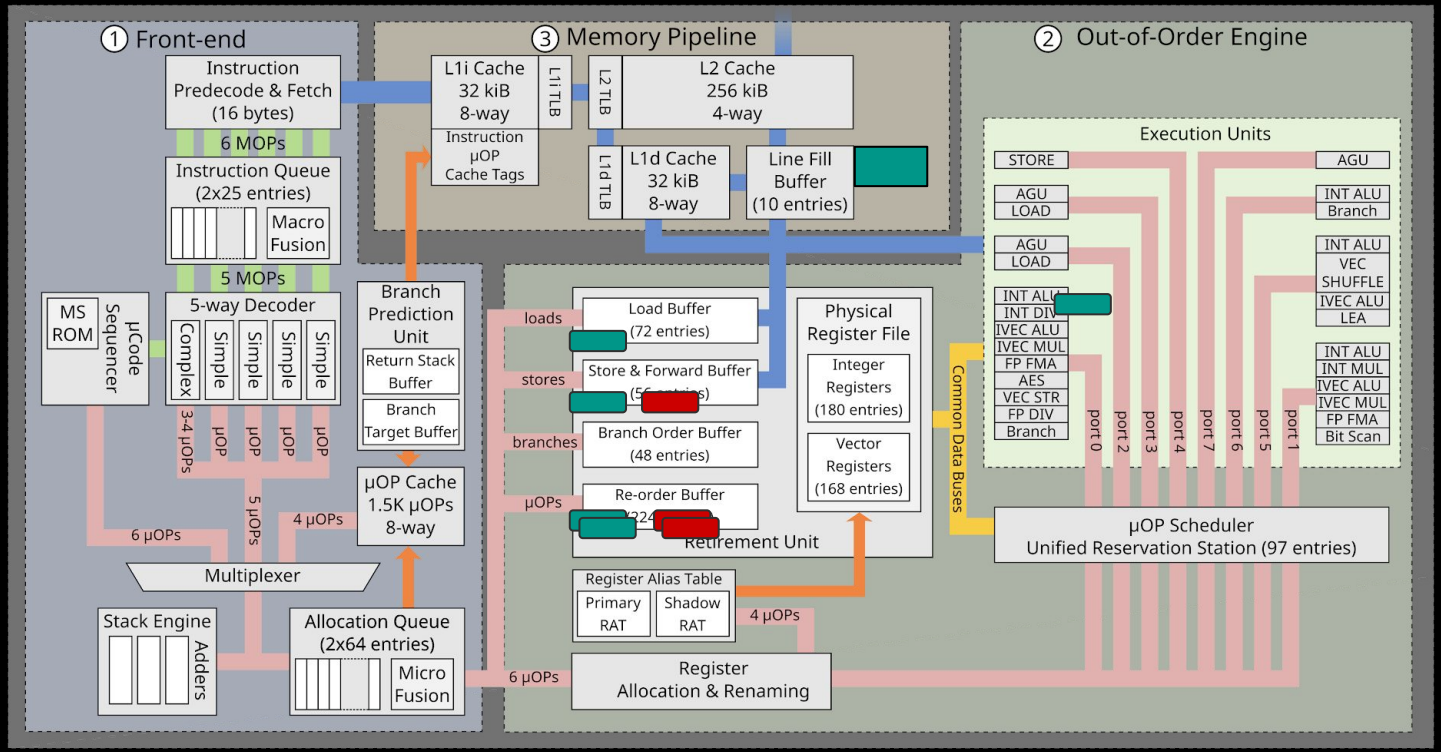
Line Fill Buffers in action

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



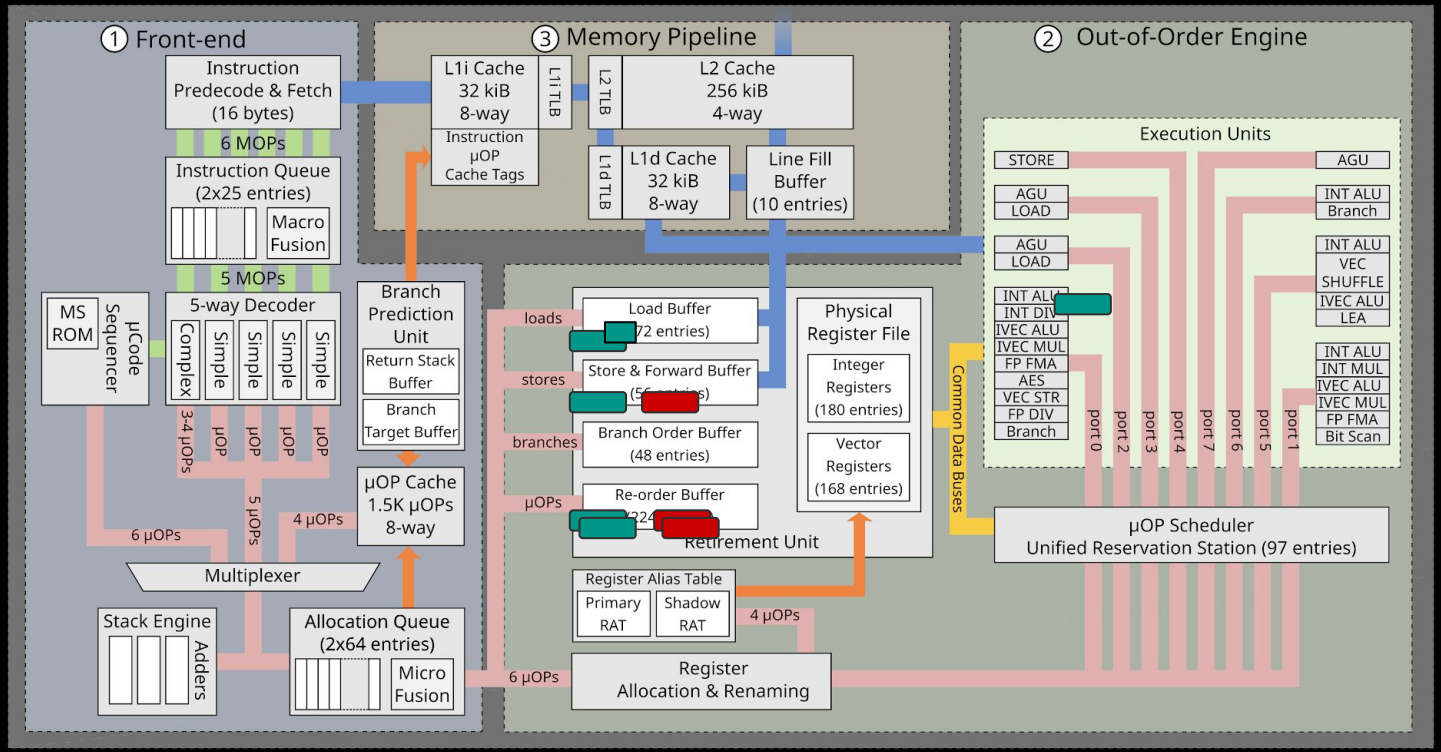
Line Fill Buffers in action

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



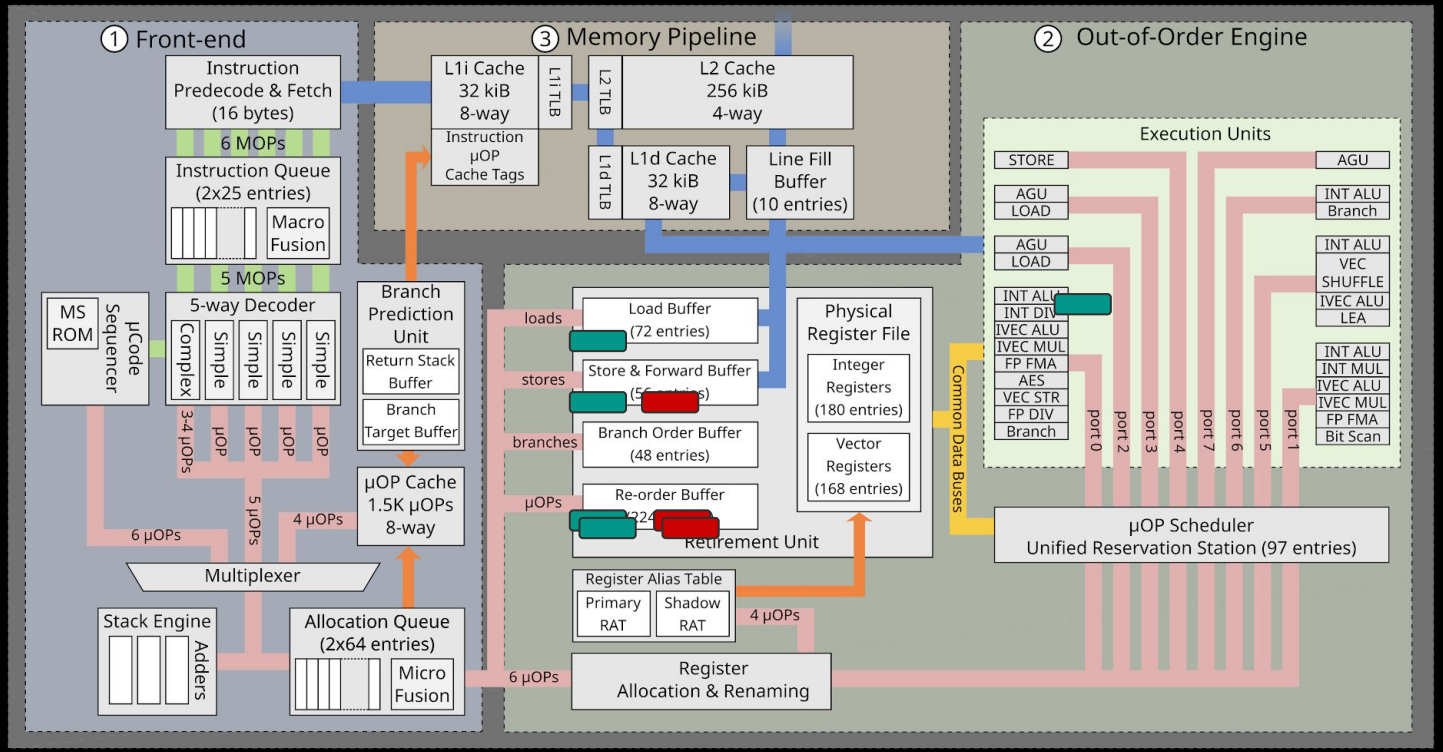
Line Fill Buffers in action

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



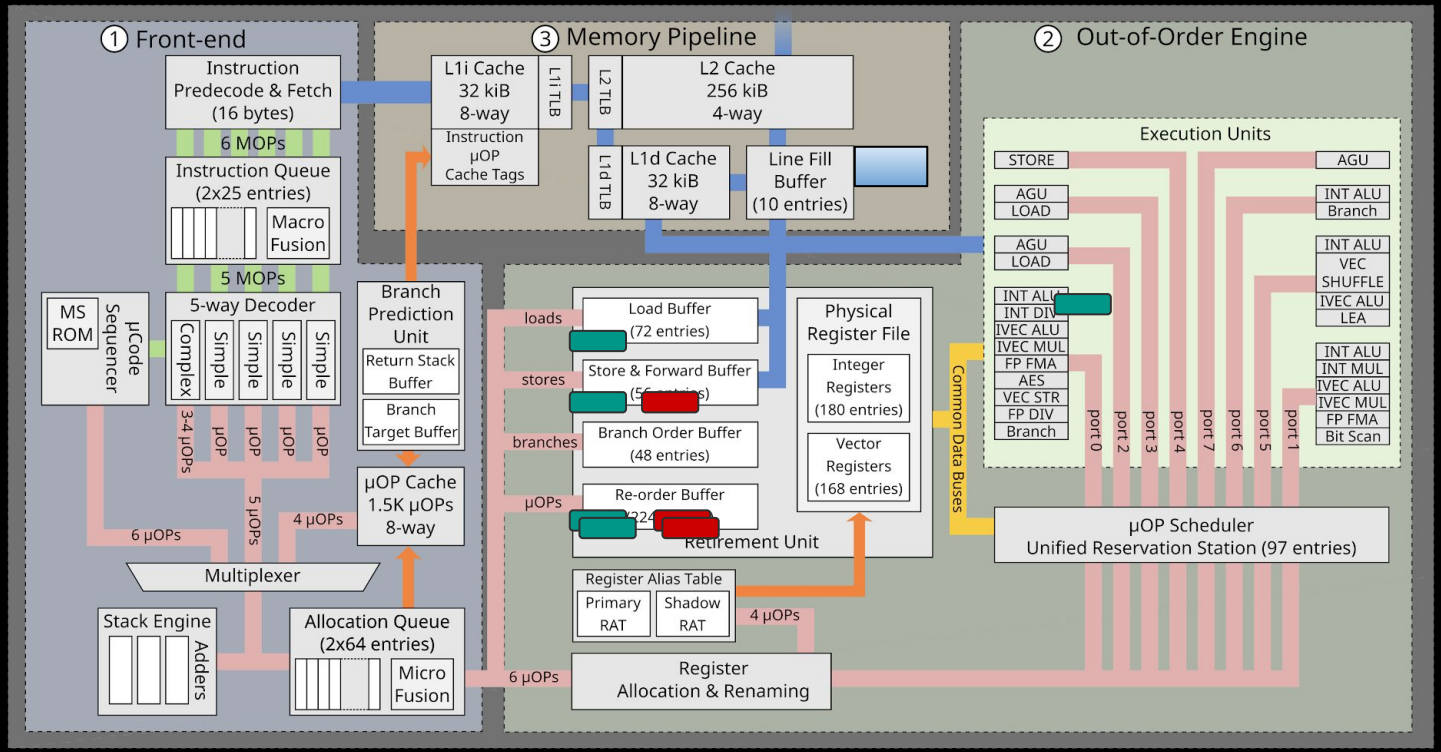
Faults with Line Fill Buffers

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



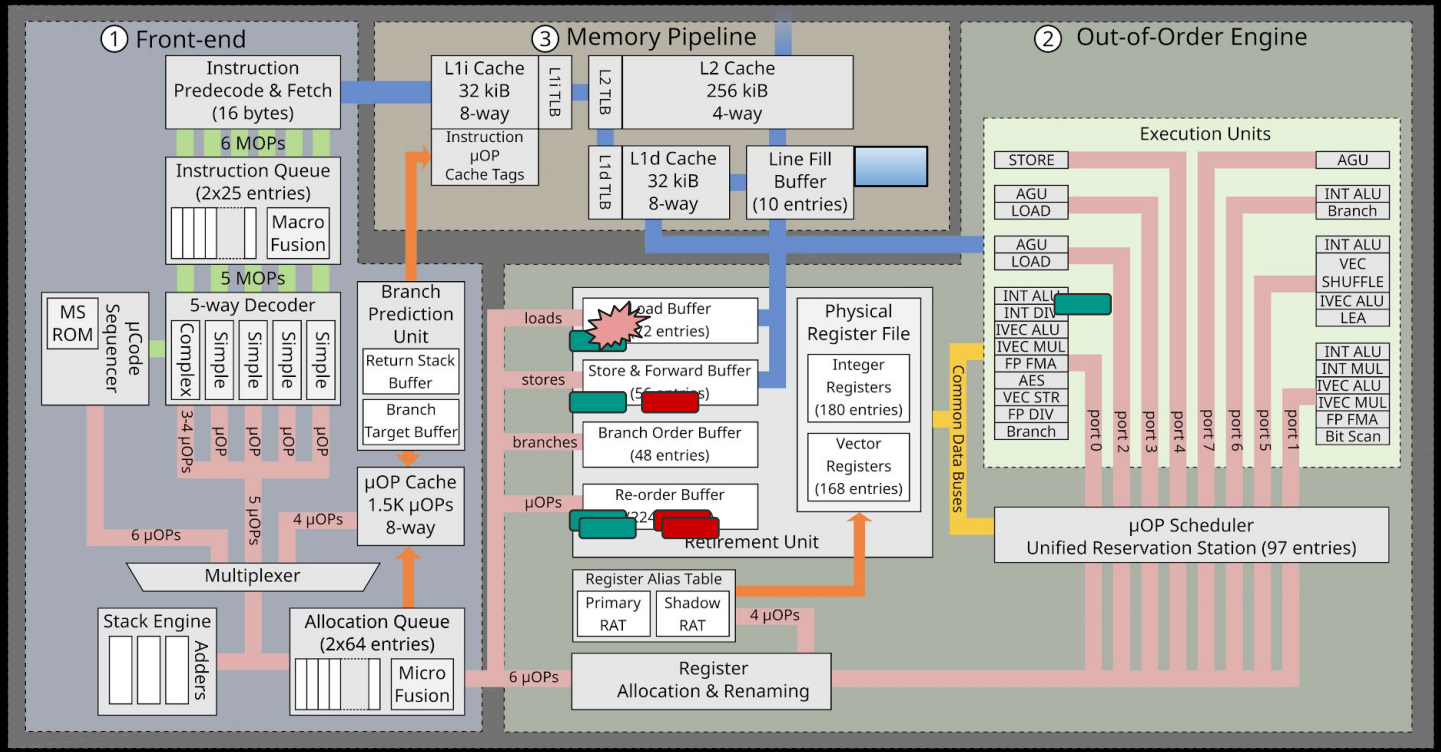
Faults with Line Fill Buffers

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



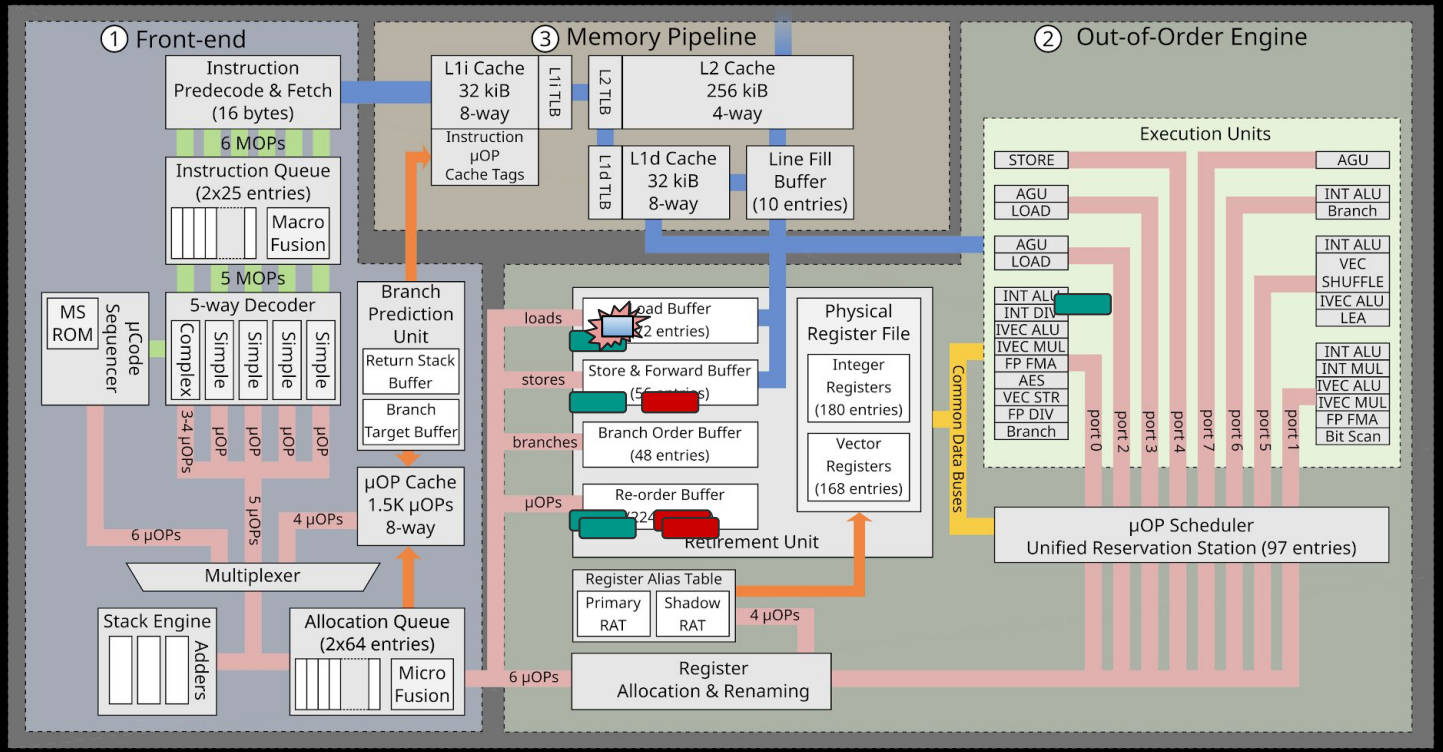
Faults with Line Fill Buffers

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



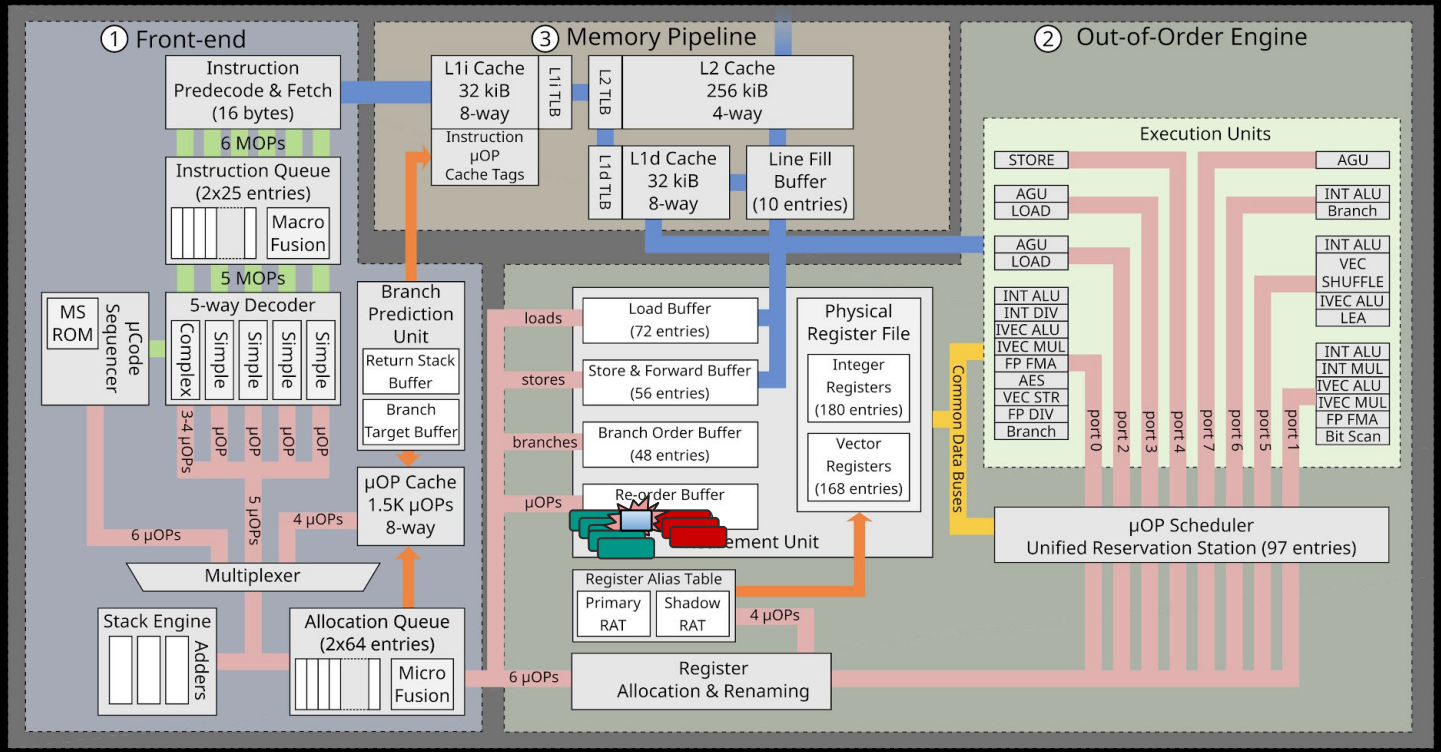
Faults with Line Fill Buffers

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



Faults with Line Fill Buffers

`add qword ptr [rax], rbx` ⇒ REG_A, REG_B, LOAD_A, ADD, STORE_A



Faults with Line Fill Buffers

- When there is a fault the pipeline is flushed
- Instructions that are in-flight finish executing
- If a load allocated a LFB entry the memory transaction that would fill the entry is aborted
  ⇒ the load finish execution with a stale LFB entry

Faults with Line Fill Buffers

- When there is a fault the pipeline is flushed
- Instructions that are in-flight finish executing
- If a load allocated a LFB entry the memory transaction that would fill the entry is aborted
  ⇒ the load finish execution with a stale LFB entry
  ⇒ yes…
  stale means that the data previously in the LFB entry is not overridden

Faults with Line Fill Buffers

- When there is a fault the pipeline is flushed
- Instructions that are in-flight finish executing
- If a load allocated a LFB entry the memory transaction that would fill the entry is aborted
  ⇒ the load finish execution with a stale LFB entry
  ⇒ yes…
  stale means that the data previously in the LFB entry is not overridden

Yes…

This is a Hardware use after free!

Faults with Line Fill Buffers

Yes…

This is a Hardware use after free!

What can possibly go wrong?

We have potentially access to data that was previously loaded through the caches, and it’s not cleaned

- What data?
  - Whatever recently passed through Line Fill Buffers!

What can possibly go wrong?

The Roman Xploit

We have potentially access to data that was previously loaded through the caches, and it’s not cleaned

- What data?
  - Whatever recently passed through Line Fill Buffers!

- Let’s meet:

RIDL & ZOMBIE LOAD

RIDL & ZombieLoad

1. Cause a fault during a cache miss which requires microcode assist
2. New data will not be loaded, so the previous value that was filled in the LFB entry remains unchanged
3. Access the data transiently and leak it through a side channel

RIDL & ZombieLoad

1. Cause a fault during a cache miss which requires microcode assist
2. New data will not be loaded, so the previous value that was filled in the LFB entry remains unchanged
3. Access the data transiently and leak it through a side channel

```
3.  X = *(char *)(ptr)
```

RIDL & ZombieLoad

1. Cause a fault during a cache miss which requires microcode assist
2. New data will not be loaded, so the previous value that was filled in the LFB entry remains unchanged
3. Access the data transiently and leak it through a side channel

```
1. char array[256 * 4096]
2. flush all array cache lines
3. X = *(char *)(ptr)
4. tmp = array[X * 4096]
```

```
1. handle SIGSEGV
2. for(i = 0; i < 256; i++)
       measureTime(array[i*4096])
3. The index with fastest access
   corresponds to X
```

ptr?: RIDL vs ZombieLoad


```
X = *(char *)(ptr)
```

- They were developed almost concurrently
- Really similar attacks, change in how to cause fault (in crafting `ptr`)

ZombieLoad:

- Really fast
- A bit more complicated to setup

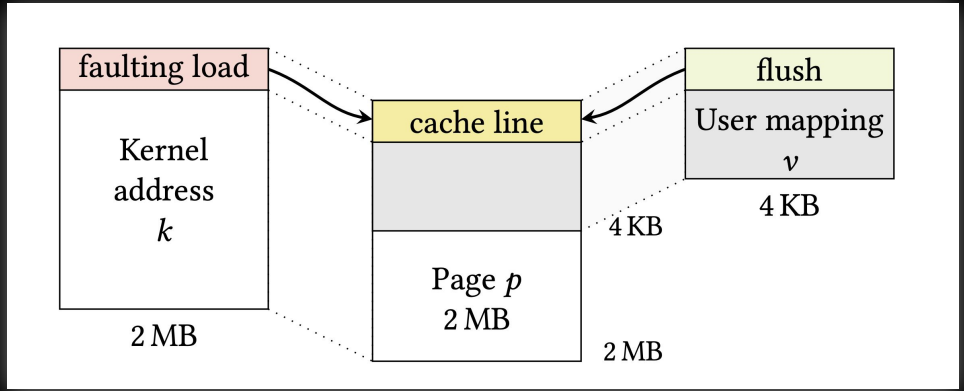
RIDL:

- Slower
- Really easy to setup

ZombieLoad


```
X = *(char *)(ptr)
```

- `ptr` is crafted such that a microcode assist is needed to handle the fault
- variant 1:
  - *k* and *v* point to the same page *p*
  - the page IS accessible by user
  - the page is not in the cache
  - `ptr` = *k*



ZombieLoad


```
X = *(char *)(ptr)
```

- a fault doesn’t need to be necessarily destructive (e.g. SEGFAULT)
- simply faults generated by microcode assist requests are fine
  - e.g: a load that requires a page table walk will fault!!
- variant 2: (for Windows mainly)
  - the windows kernel periodically clears ACCESSED BIT in the page table
  - access a page with ACCESSED BIT not set

RIDL


```
X = *(char *)(ptr)
```

- There are lots of ways to induce leaks!
- We don’t need necessarily invalid instructions!
  - valid page faults

```
ptr = mmap (... , PAGE_SIZE , ...) ;
```

  - invalid page faults

```
ptr = NULL;
```

  - misaligned reads

```
ptr = <PAGE_SIZE aligned> buf + CACHE_LINE_SIZE - 1
```

RIDL


```
X = *(char *)(ptr)
```

- There are lots of ways to induce leaks!
- We don’t need necessarily invalid instructions!
- The leaks do not depend in any way on the address!

RIDL

The Roman Xploit

The simplest attack you have ever seen!

1. `char array[256 * 4096]`
2. `flush all array cache lines`
3. `X = *(volatile char *)(NULL)`
4. `tmp = array[X * 4096]`

1. `handle SIGSEGV`
2. `for(i = 0; i < 256; i++)`
   `measureTime(array[i*4096])`
3. `The index with fastest access corresponds to X`

RIDL + ZombieLoad

So we are able to intercept data in flight in the CPU

⇒ MICROARCHITECTURAL DATA SAMPLING ATTACKS!

RIDL + ZombieLoad

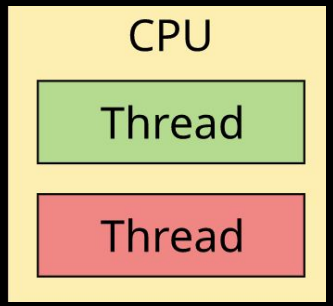
The Roman Xploit

So we are able to intercept data in flight in the CPU

Which data?

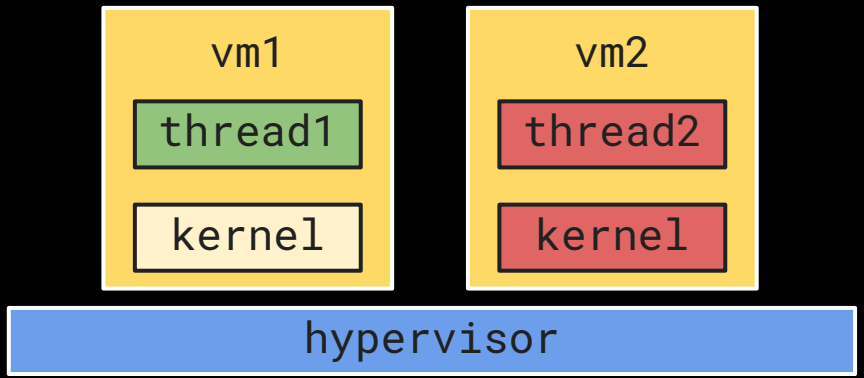
- Any data that passes through the caches!
- Thanks to Intel Hyper Threading the L1 cache is shared by processes on the same physical core

⇒ can leak secrets used by different threads!



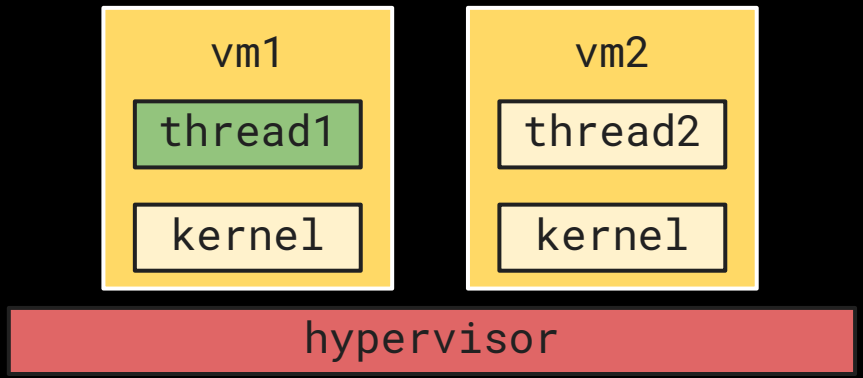
RIDL + ZombieLoad

- Caches know no security domains ⇒ all use the cache!
- Anything that pass is leakable, from everywhere
- Even from different virtual machines



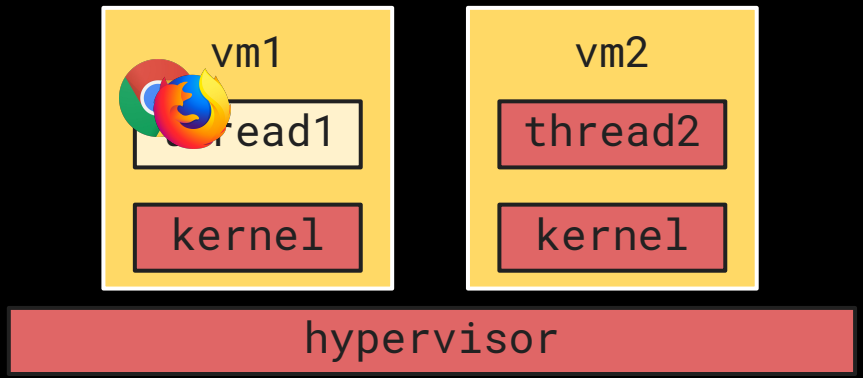
RIDL + ZombieLoad

- Caches know no security domains ⇒ all use the cache!
- Anything that pass is leakable, from everywhere
- Even from different virtual machines
- Or even values used by the hypervisor



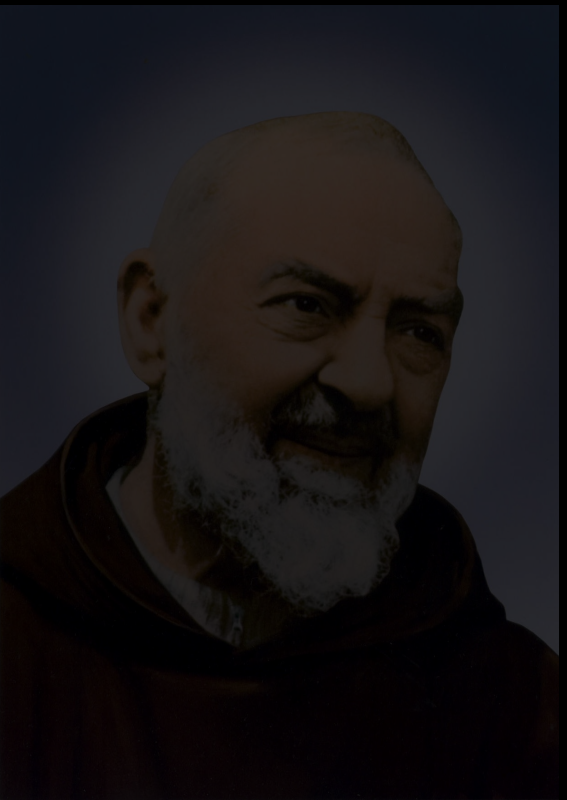
RIDL + ZombieLoad

- Caches know no security domains ⇒ all use the cache!
- Anything that pass is leakable, from everywhere
- Even from different virtual machines
- Or even values used by the hypervisor
- Since RIDL doesn't require illegal faults can mount the attack even from javascript!

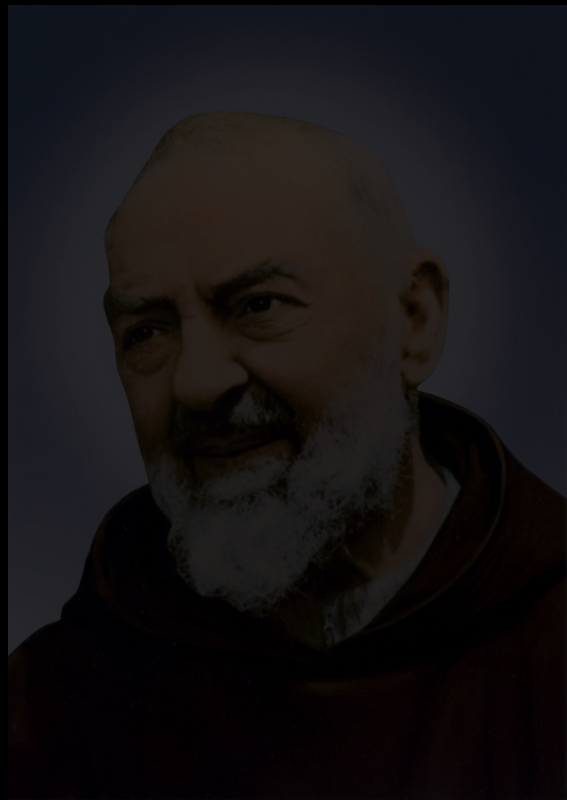


RIDL + ZombieLoad

The Roman Xploit



DEMO TIME!



Microarchitectural Data Sampling

- A new type of Transient execution attack
- Be able to sample for values passing in the CPU caches (or in-flight)
- You will hear also of FALLOUT among MDS attacks:
  - A variant that leaks value stored into the cache
  - But the whole process is the same

Questions?

The Roman Xploit

Thank you!

References

MUST READ:

[+] RIDL Paper: https://mdsattacks.com/files/ridl.pdf
Slides: https://mdsattacks.com/slides/slides.html (thanks for some images :))

[+] ZombieLoad Paper: https://zombieloadattack.com/zombieload.pdf

[+] Fallout Paper: https://mdsattacks.com/files/fallout.pdf

[+] Interactive guide to Speculative execution attacks:
https://mdsattacks.com/diagram.html