



# Transient Execution Attacks

explained to your Grandma



by pietroborrello

inspired by: [[A Systematic Evaluation of Transient Execution Attacks and Defenses](#)]



# Outline

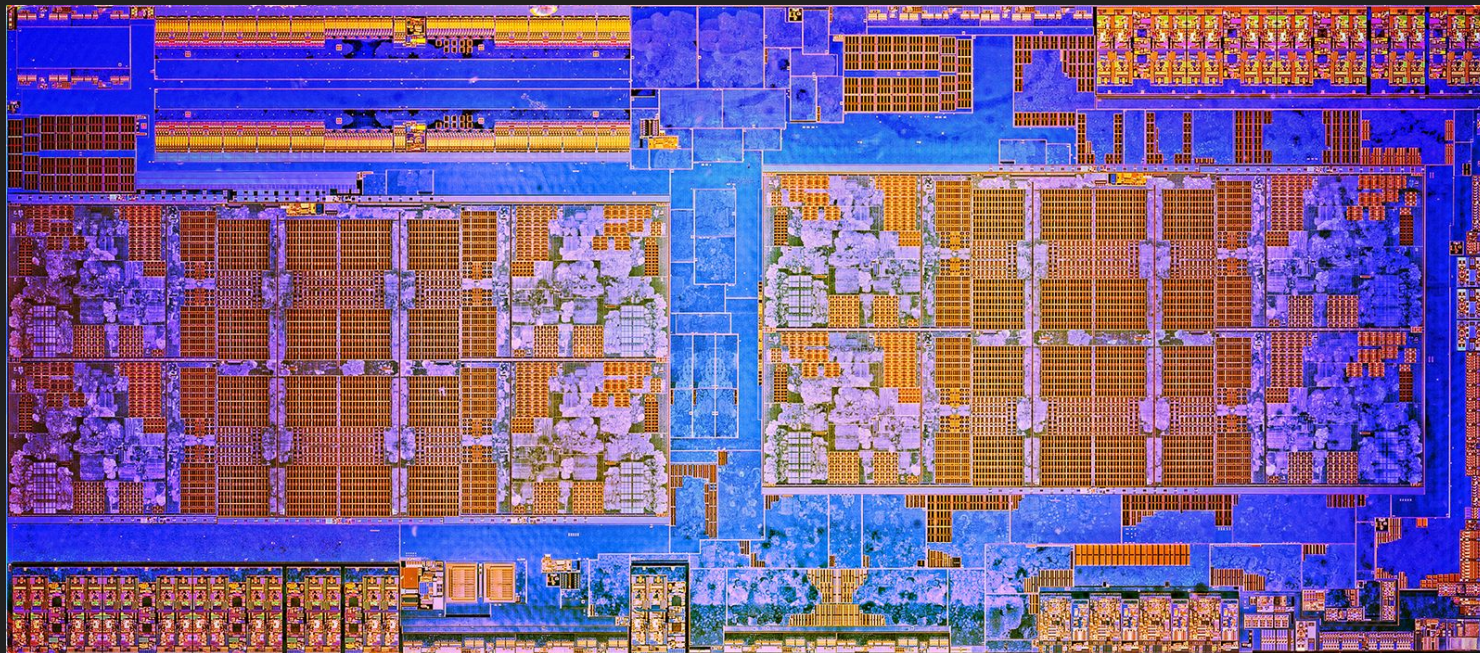


1. How do Modern Processors work?
2. Let's dive into micro-architectural attacks!



# AMD Ryzen

The  
Roman  
Xploit





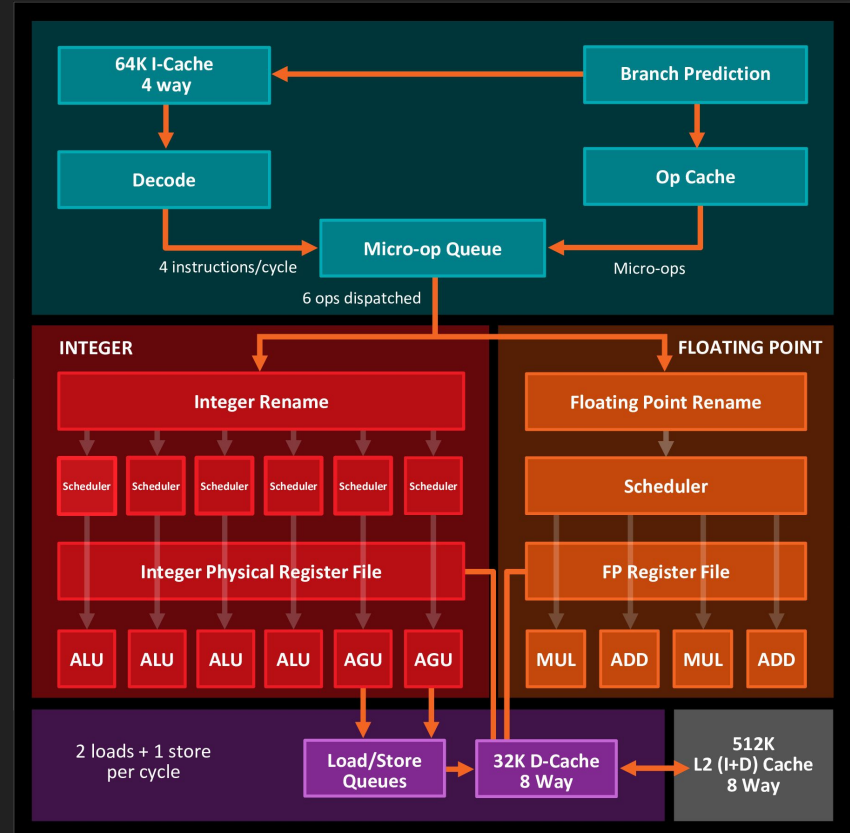


# AMD Ryzen





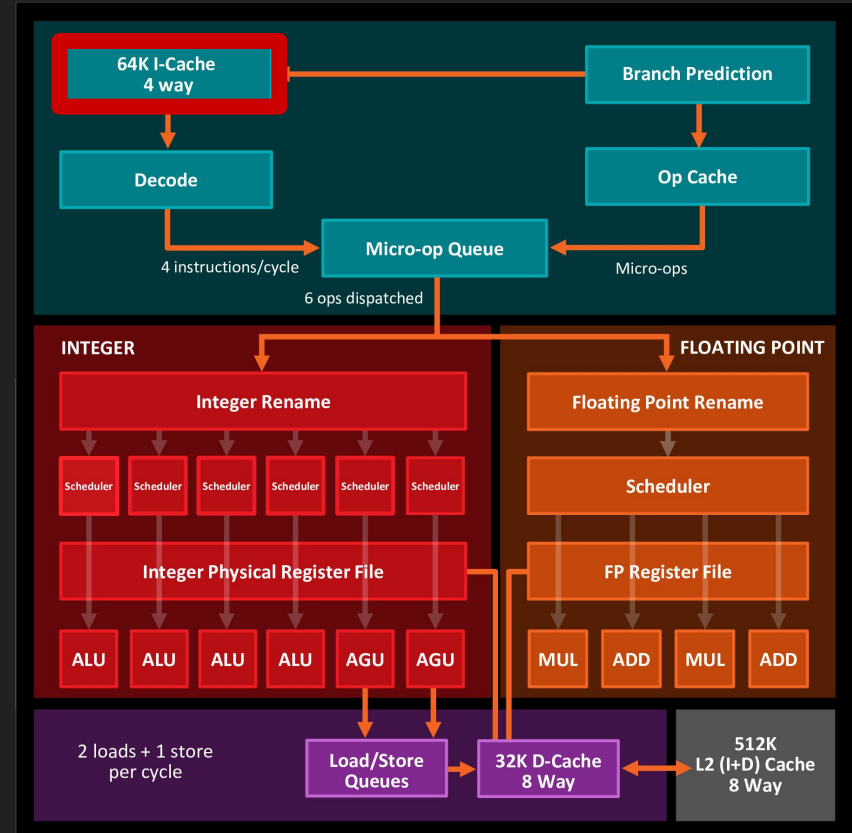
# Modern Processors





# Modern Processors

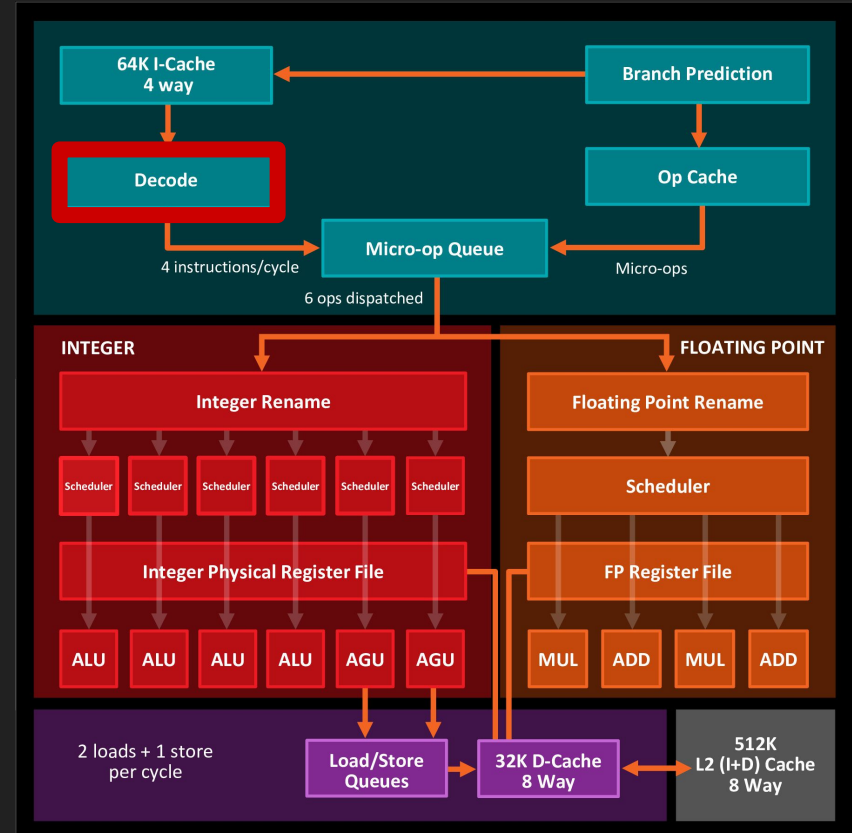
add qword ptr [rax], rbx





# Modern Processors

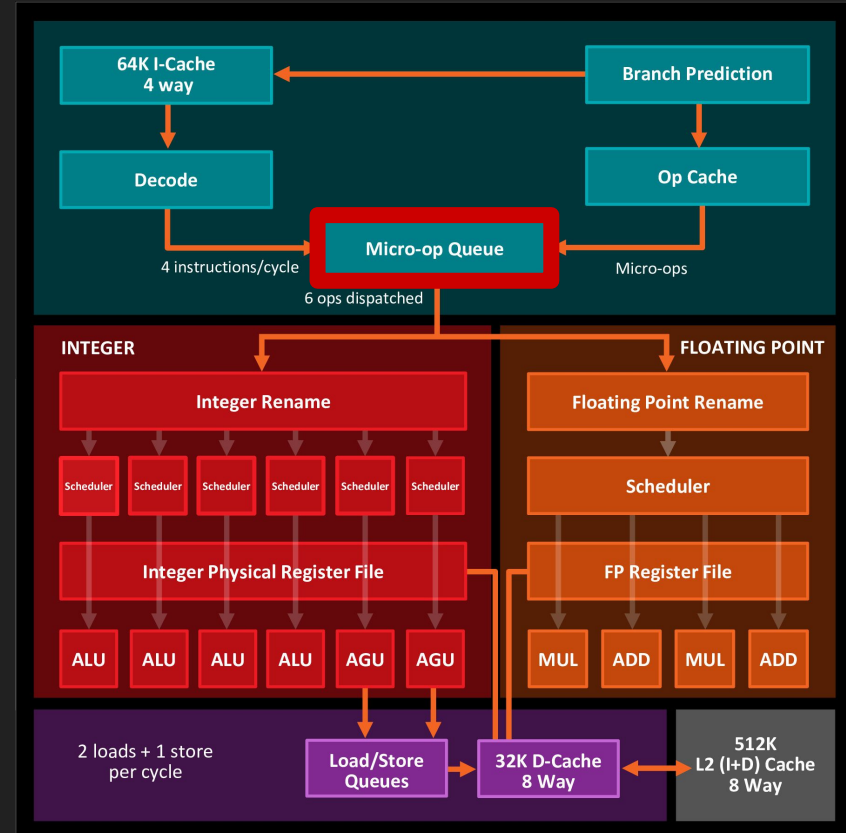
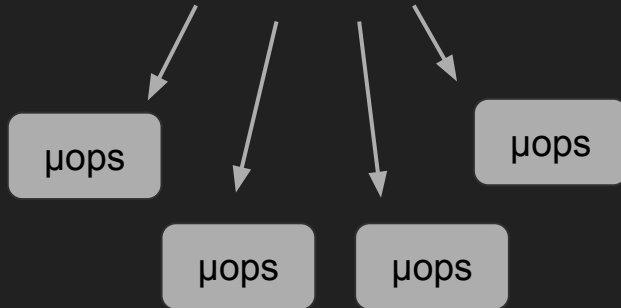
add qword ptr [rax], rbx





# Modern Processors

add qword ptr [rax], rbx

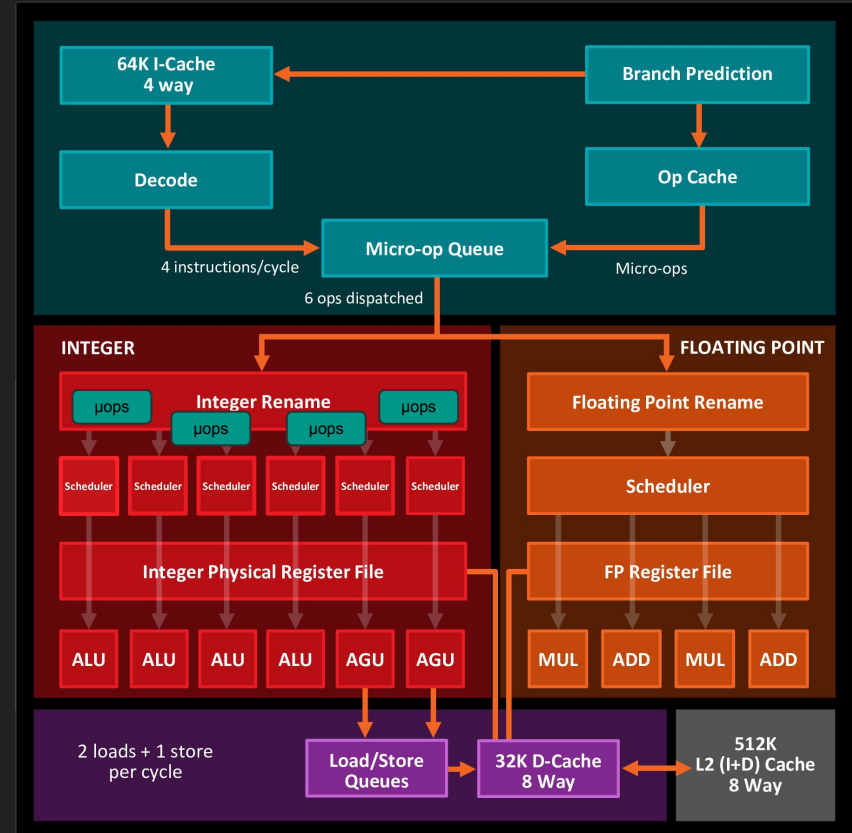
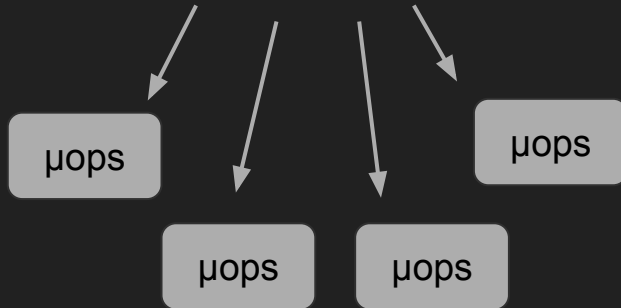






# Modern Processors

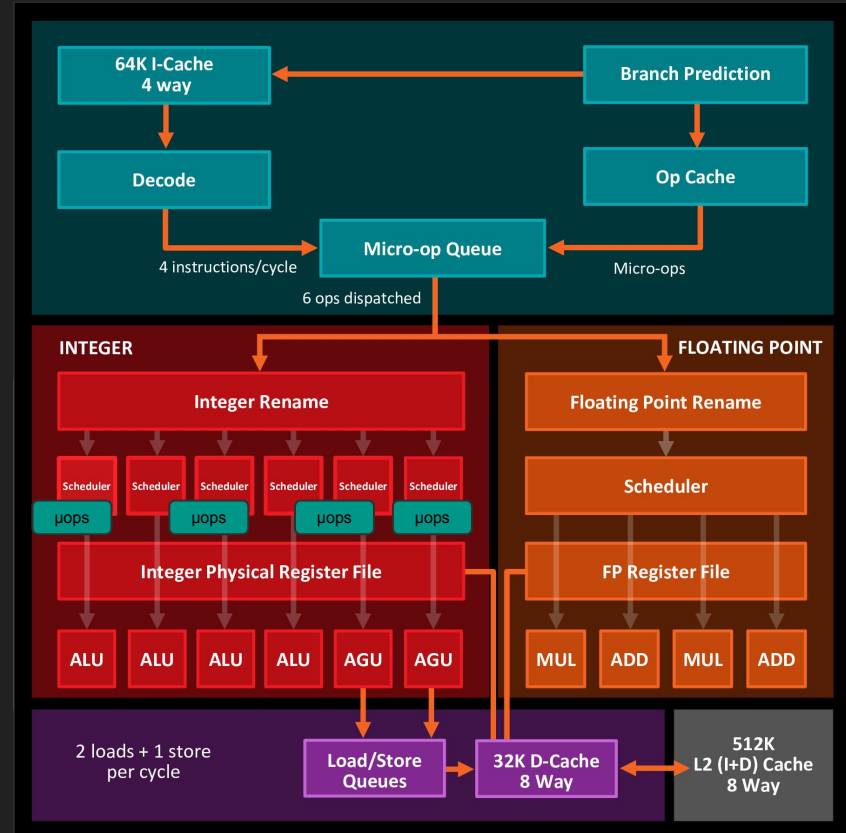
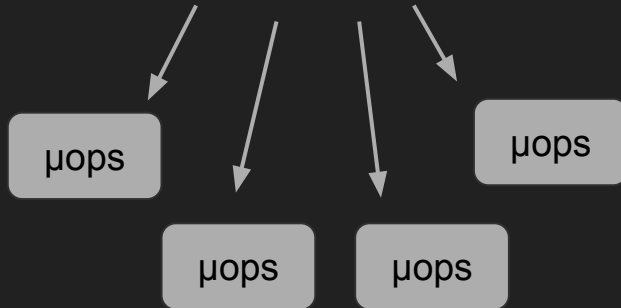
add qword ptr [rax], rbx





# Modern Processors

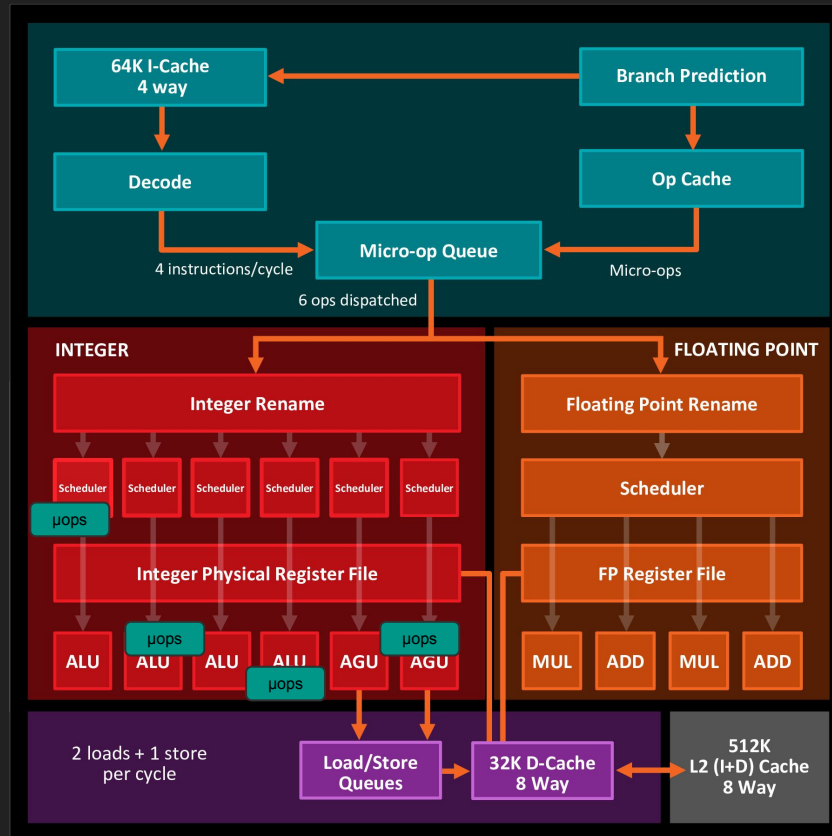
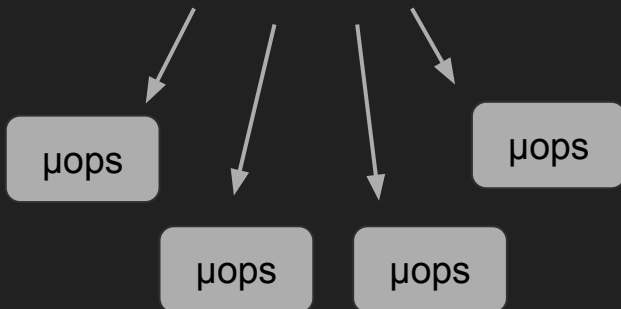
add qword ptr [rax], rbx





# Modern Processors

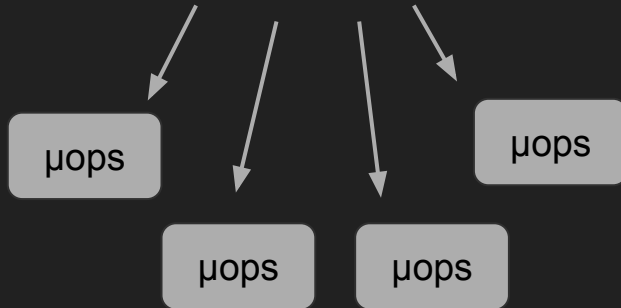
add qword ptr [rax], rbx



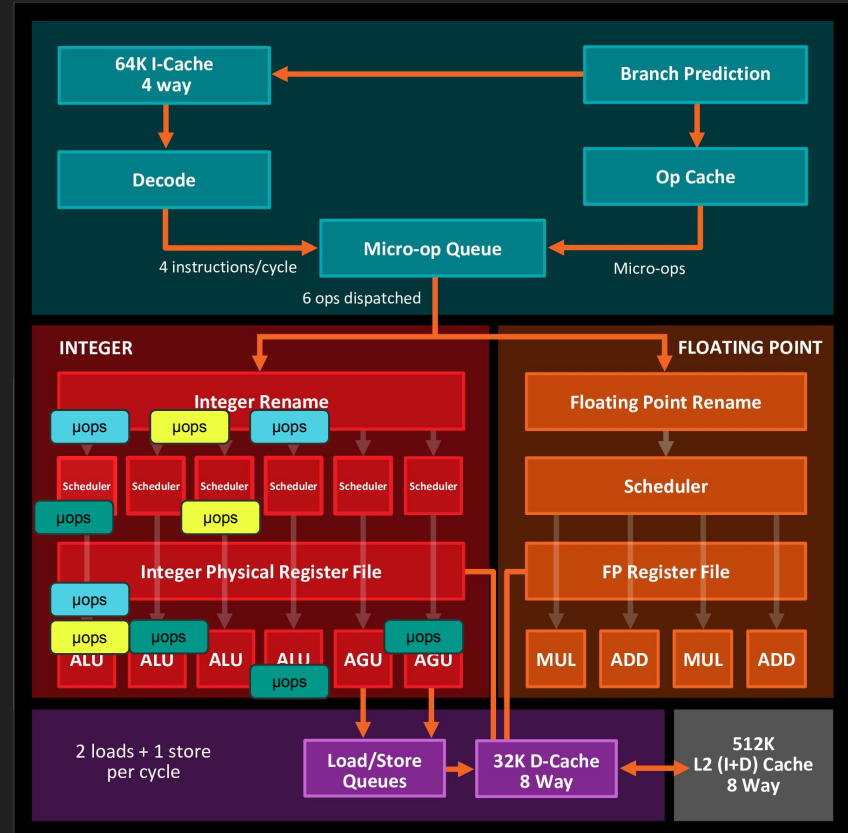


# Modern Processors

add qword ptr [rax], rbx



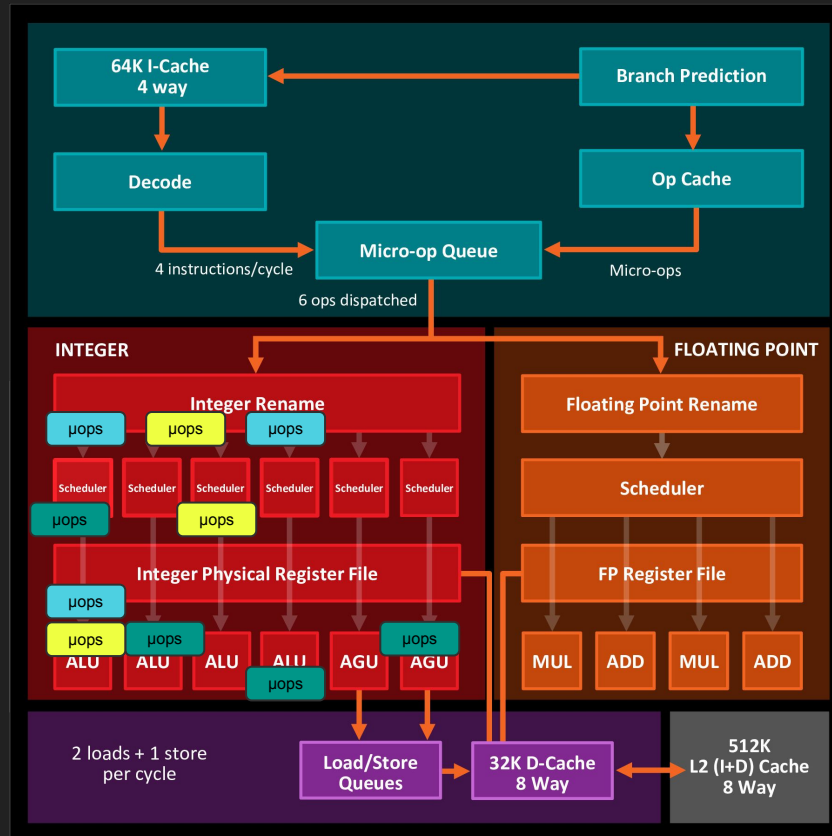
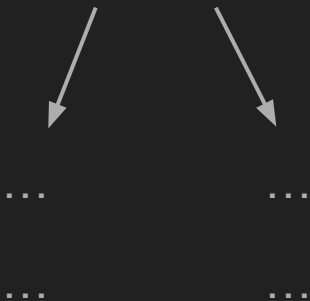
cmp rdx, qword ptr [rax]  
jne 0xdeadbeef





# Modern Processors

```
add qword ptr [rax], rbx
cmp rdx, qword ptr [rax]
jne 0xdeadbeef
```

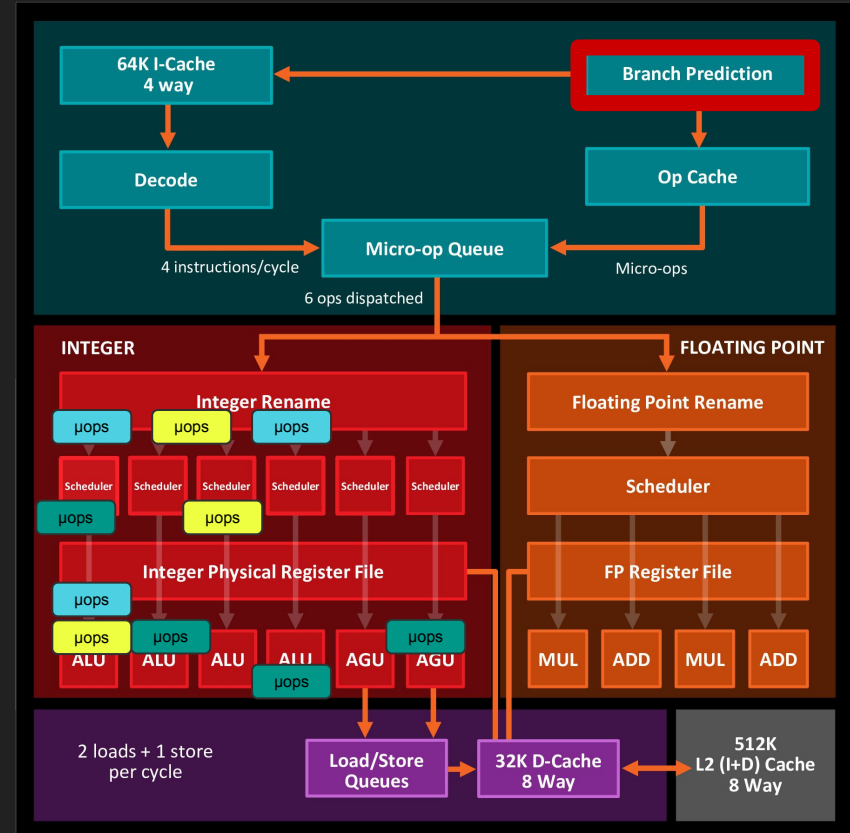
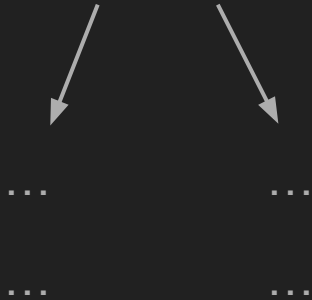






# Modern Processors

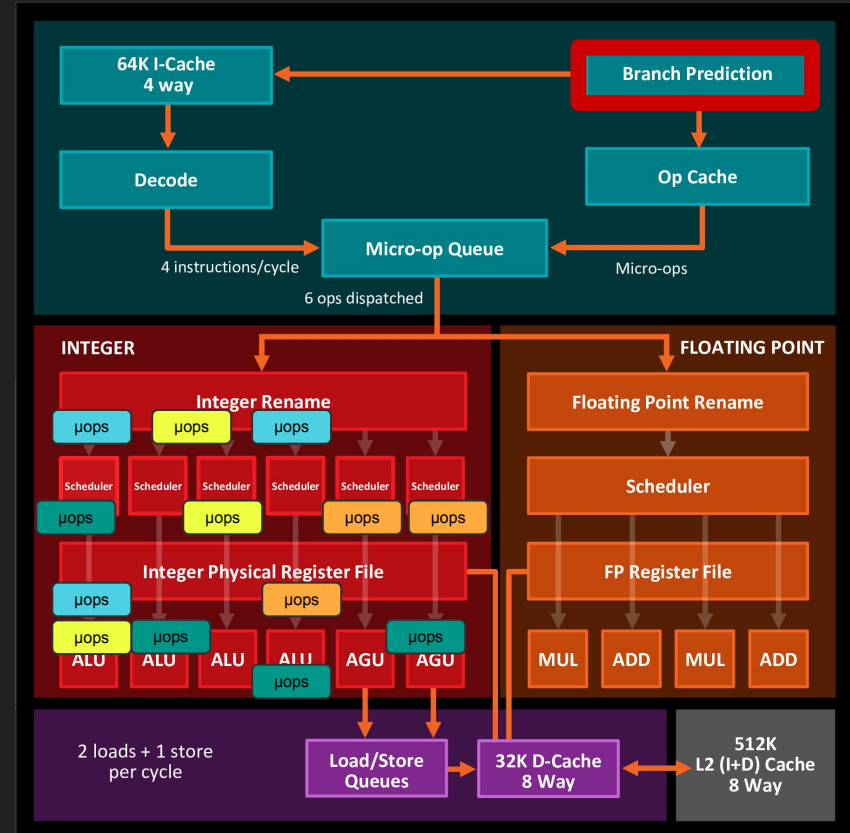
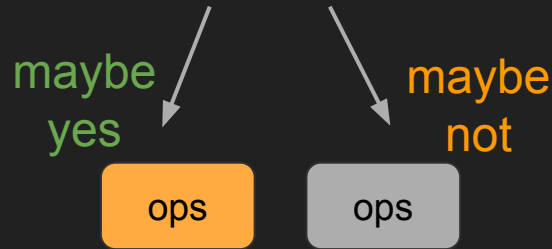
```
add qword ptr [rax], rbx  
cmp rdx, qword ptr [rax]  
jne 0xdeadbeef
```





# Modern Processors

```
add qword ptr [rax], rbx
cmp rdx, qword ptr [rax]
jne 0xdeadbeef
```

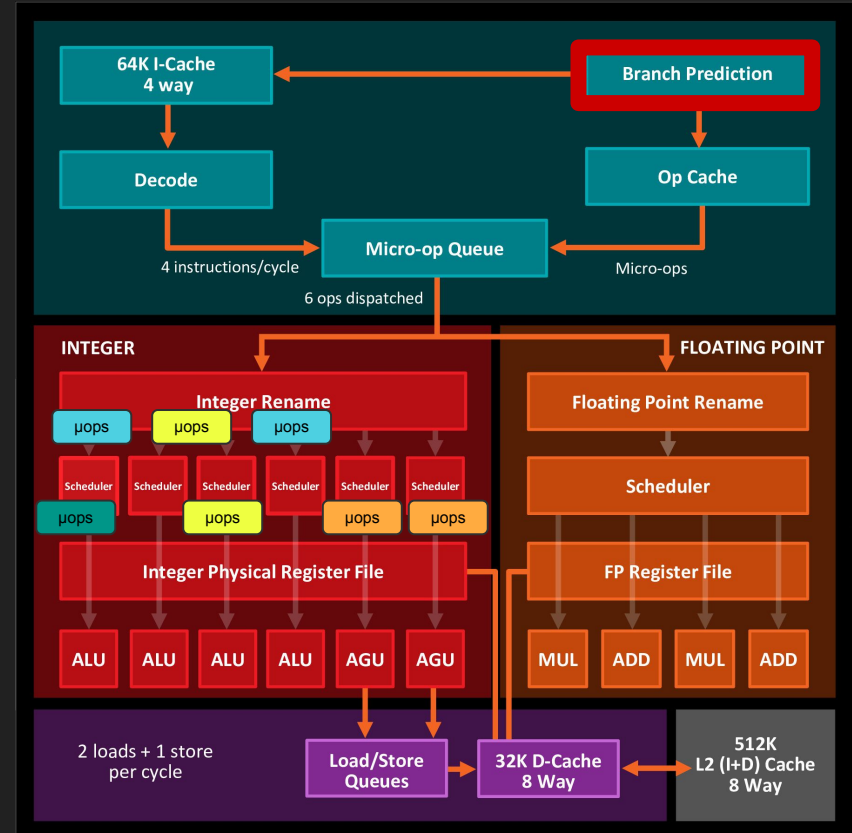




# Modern Processors

```
add qword ptr [rax], rbx
cmp rdx, qword ptr [rax]
jne 0xdeadbeef
```

REORDER  
BUFFER

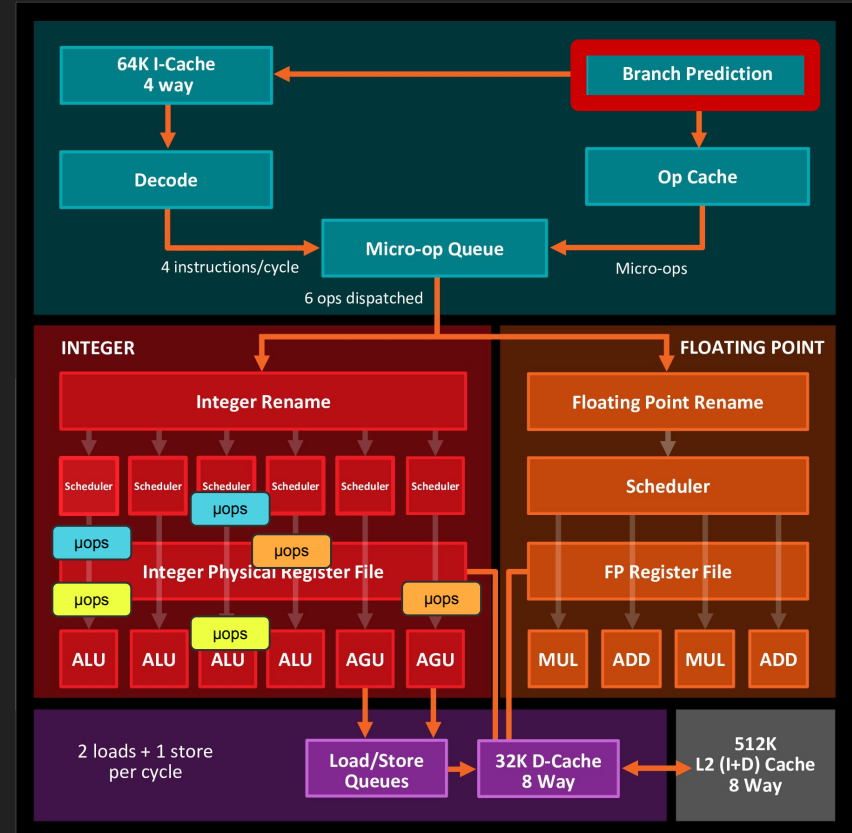
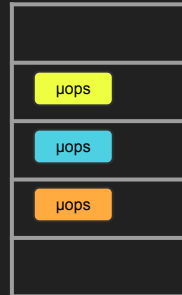




# Modern Processors

```
add qword ptr [rax], rbx  
cmp rdx, qword ptr [rax]  
jne 0xdeadbeef
```

REORDER  
BUFFER

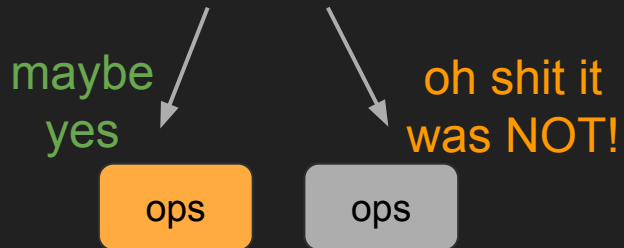




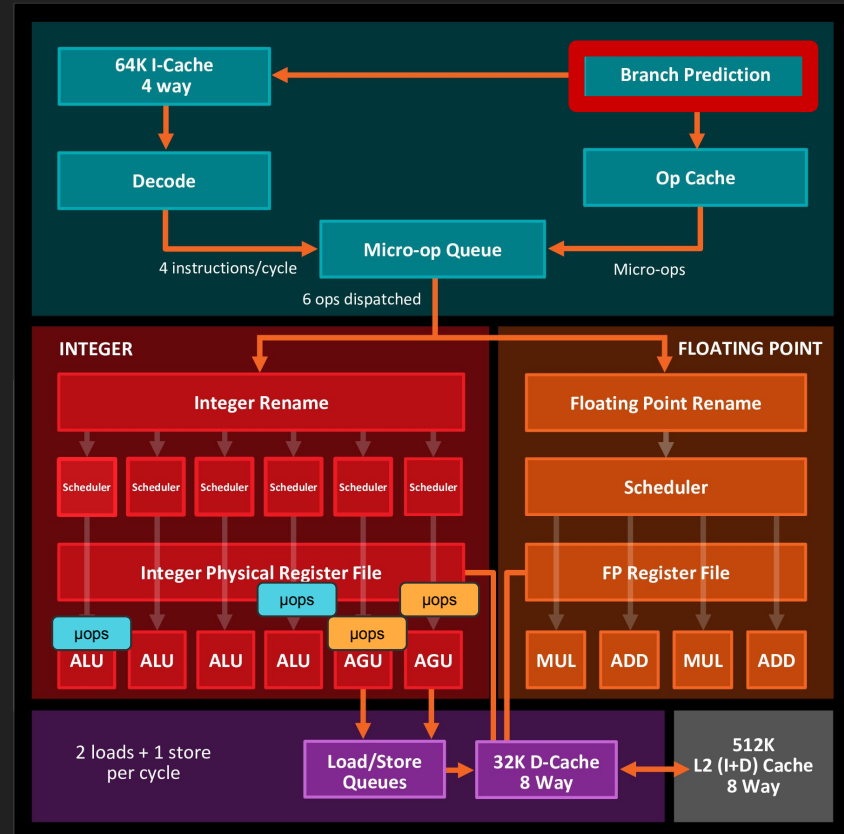
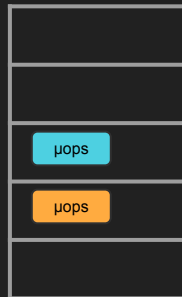
# Modern Processors



```
add qword ptr [rax], rbx
cmp rdx, qword ptr [rax]
jne 0xdeadbeef
```



REORDER  
BUFFER

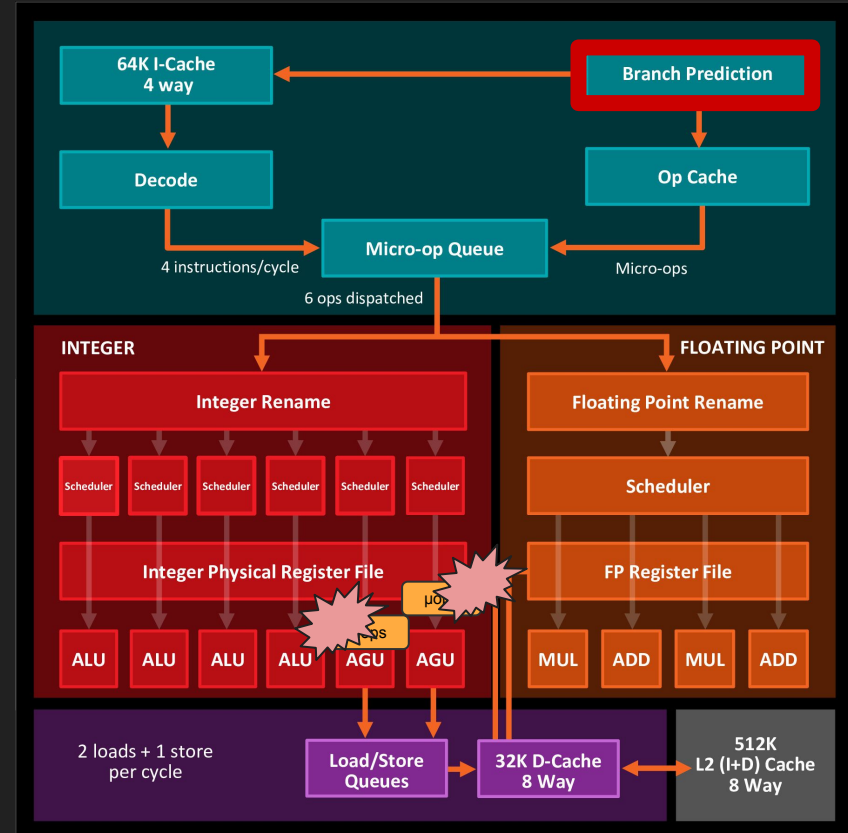
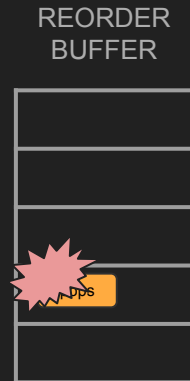
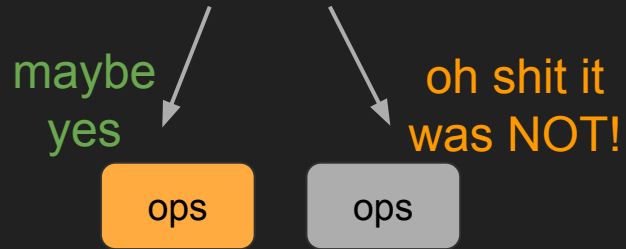






# Modern Processors

```
add qword ptr [rax], rbx
cmp rdx, qword ptr [rax]
jne 0xdeadbeef
```



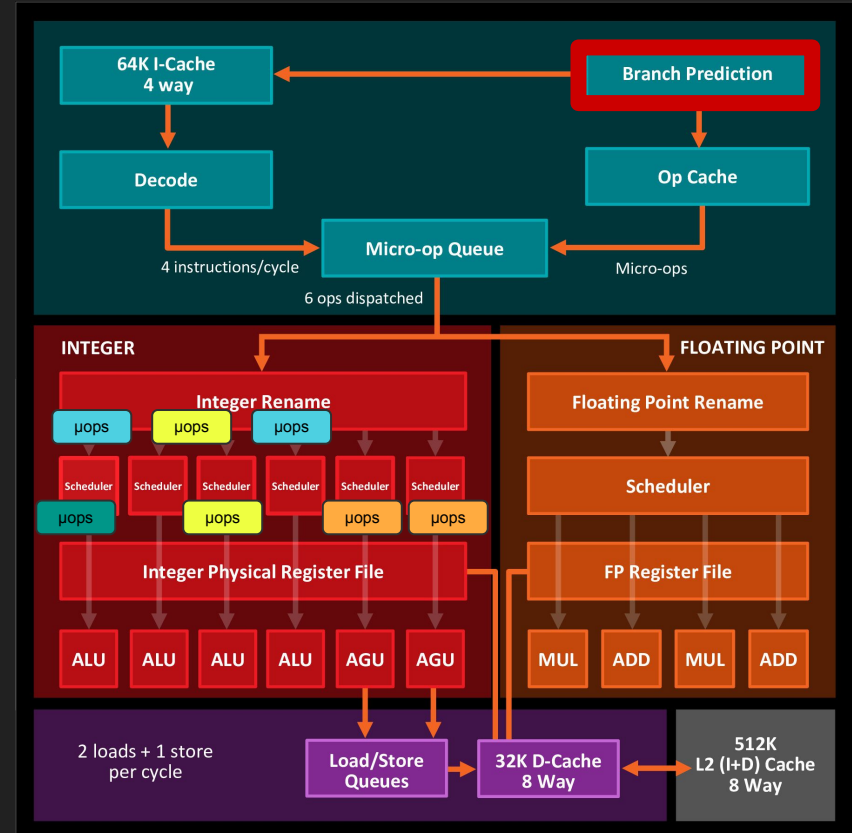


# Modern Processors

```
add qword ptr [rax], rbx  
cmp rdx, qword ptr [rax]  
jne 0xdeadbeef
```

oh shit RAX  
was pointing  
to kernel  
memory!!

REORDER  
BUFFER

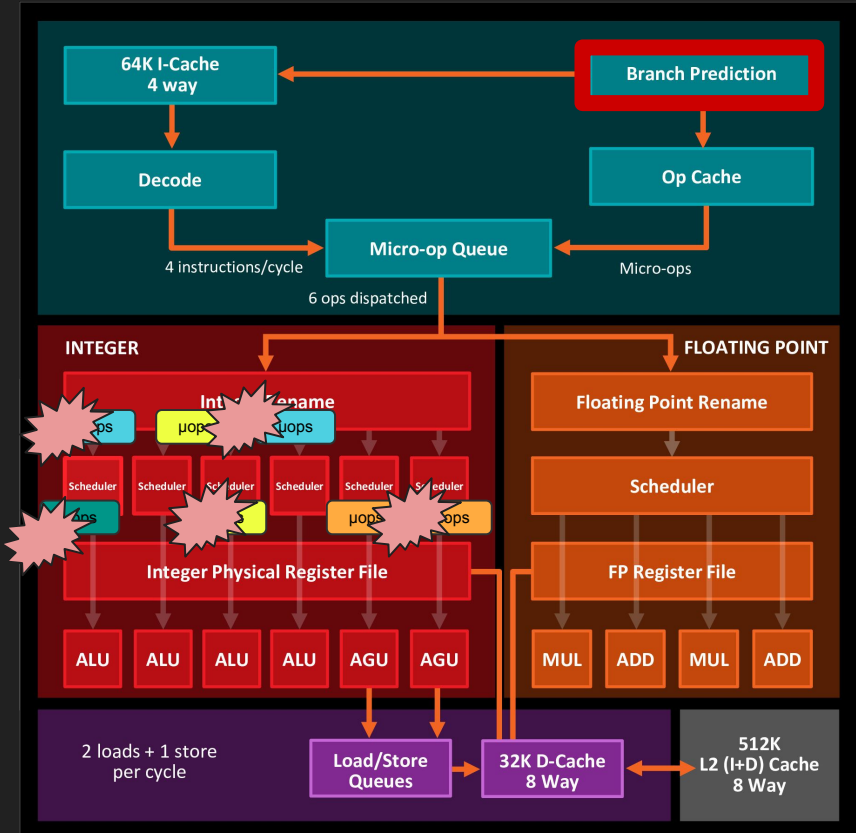
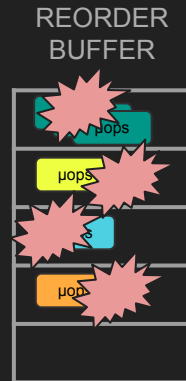




# Modern Processors

```
add qword ptr [rax], rbx  
cmp rdx, qword ptr [rax]  
jne 0xdeadbeef
```

oh shit RAX  
was pointing  
to kernel  
memory!!



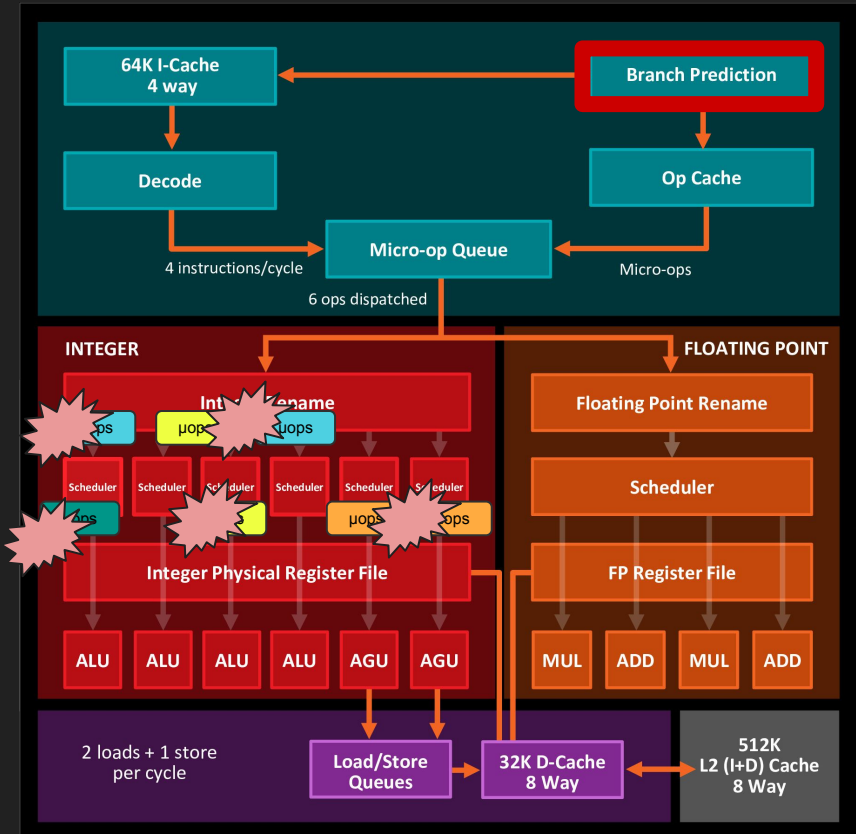
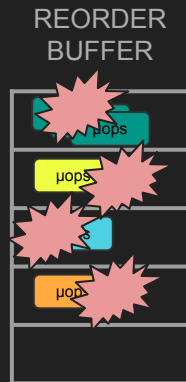


# Modern Processors

```
add qword ptr [rax], rbx  
cmp rdx, qword ptr [rax]  
jne 0xdeadbeef
```

General  
Protection  
Error!

oh shit RAX  
was pointing  
to kernel  
memory!!





# Roll Back?



- What does it mean to roll back (undo) an operation for a CPU?
- You cannot undo a Logical operation (it was an Electrical signal!)
- But you can hide what you did

⇒ Behave as nothing happened

- Do not save the operation into the architectural state





# What is the architectural state?

- General Purpose Registers (RAX, RSP, ...)
- Control Registers (RFLAGS, GDTR, IDTR, CR0, CR1, ...)
- Model Specific Registers
- Floating Point Registers
- Memory
- ...

But, this doesn't include:

- All Instruction and Data Caches (L1, L2, ...), TLB, ...
- Branch Predictors
- And all the microarchitecture that we just saw...



# LEAKY BOI?



- So we are using data or executing code we shouldn't and we are exposing it into the microarchitecture!
- But we cannot access directly the microarchitecture



# LEAKY BOI?

- So we are using data or executing code we shouldn't and we are exposing it into the microarchitecture!
- But we cannot access directly the microarchitecture
- Directly...

1. Read kernel dword into X

2. if(X == 0xdeadbeef)

flush\_entire\_cache

} Executed only transiently

When resuming from SIGSEGV,  
is the cache flushed?



# Some order

- Two ways to induce a roll back of a transient execution:

Out-Of-Order Exceptions



MELTDOWN Family

Misprediction



SPECTRE Family

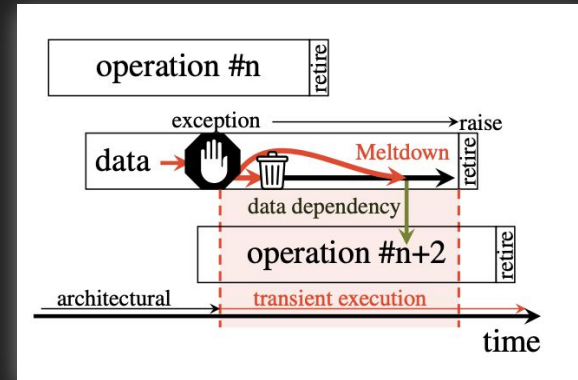


# Meltdown Family



- Exceptions are enforced lazily

⇒ There is a small window where we can use the result of faulty instructions, and access data that should be architecturally inaccessible (e.g. kernel memory!)







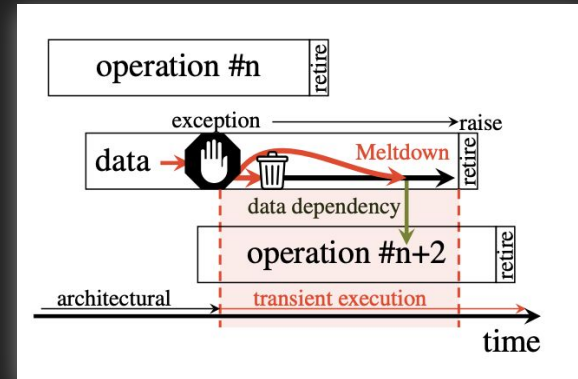
# Meltdown Family

- Exceptions are enforced lazily

⇒ There is a small window where we can use the result of faulty instructions, and access data that should be architecturally inaccessible (e.g. kernel memory!)

- What to do with the results of faulty instructions?  
How can we read them?

⇒ Use a micro-architectural covert channel!





# Flush+Reload



- Use cache as covert channel:

HIT: fast

MISS: slow

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



# Flush+Reload

- Use cache as covert channel:

HIT: fast

MISS: slow

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

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



# Flush+Reload

- Use cache as covert channel:

HIT: fast

MISS: slow

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

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



# Meltdown Attacks



- Different types of faults can be involved, depending on what I shouldn't read:
  - Kernel Memory
  - Secure Enclave Memory
  - Privileged System Registers
  - FPU Registers of other Processes
  - Unreadable pages, bypassing Protection Keys
  - Out-of-Bound access driven by exceptions (more with Spectre)



# Supervisor Bypass



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

- |   |  |
|---|--|
| 1. <code>char array[256 * 4096]</code>      | 1. <code>handle SIGSEGV</code>                       |
| 2. <code>flush all</code> array cache lines | 2. <code>for(i = 0; i &lt; 256; i++)</code>          |
| 3. <code>read kernel byte into X</code>     | <code>measureTime(array[i*4096])</code>              |
| 4. <code>tmp = array[X * 4096]</code>       | 3. The index with fastest access<br>corresponds to X |

- Dump entire kernel memory byte by byte



# Enclave Bypass (Foreshadow)



- Trusted execution environment, with integrity and confidentiality guarantees
- Isolated and HW encrypted compartment, even secret for the kernel
- The memory is silently replaced with 0xFF when try to read ⇒ No fault!



# Enclave Bypass (Foreshadow)



- Trusted execution environment, with integrity and confidentiality guarantees
- Isolated and HW encrypted compartment, even secret for the kernel
- The memory is silently replaced with 0xFF when try to read ⇒ No fault!

1. Execute the enclave to bring unencrypted data to L1 cache
2. Manually revoke access permission to enclave memory
3. Now when trying to access enclave memory we have a Page Fault!  
Before 0xFF substitution takes place

⇒ Then same attack!

(And can also be extended to break VM isolation)





# System Register Bypass



- Privileged system registers can be read and written by the kernel
- They contain private kernel informations  
(i.e. IA32\_LSTAR MSR contains fast syscall handler address)
- Accessing them from users space issues a General Protection Fault

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



# System Register Bypass



- Privileged system registers can be read and written by the kernel
- They contain private kernel informations  
(i.e. IA32\_LSTAR MSR contains fast syscall handler address)
- Accessing them from users space issues a General Protection Fault

1. `char array[256 * 4096]`
2. `flush all` array cache lines
3. `rdmsr` byte `into X`  $\Rightarrow$  Now you have broken KASLR!
4. `tmp = array[X * 4096]`



# FPU Register Bypass



- At context switches the kernel saves all the registers of the current process
- Floating Point Unit and SIMD registers are huge!  
So kernel doesn't save them, but marks them as NOT AVAILABLE
- If FPU or SIMD is used by next process, a NOT AVAILABLE exception is raised, and the kernel can save them, before next process can access them



# FPU Register Bypass



- At context switches the kernel saves all the registers of the current process
- Floating Point Unit and SIMD registers are huge!  
So kernel doesn't save them, but marks them as NOT AVAILABLE
- If FPU or SIMD is used by next process, a NOT AVAILABLE exception is raised, and the kernel can save them, before next process can access them

⇒ EXCEPTION??

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



# FPU Register Bypass



- At context switches the kernel saves all the registers of the current process
- Floating Point Unit and SIMD registers are huge!  
So kernel doesn't save them, but marks them as NOT AVAILABLE
- If FPU or SIMD is used by next process, a NOT AVAILABLE exception is raised, and the kernel can save them, before next process can access them

⇒ EXCEPTION??

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

Can leak SIMD cryptographic computations!



# Other Bypasses



- With the same approach we can bypass memory protection (i.e. Execute Only) even if enforced with Protection Keys
- Additionally can perform out of bound speculative reads, if enforced with bound instruction



# Spectre Family

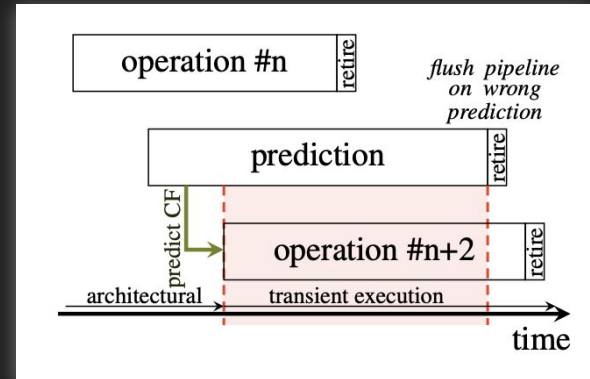


- The CPU executes predicted instructions transiently

⇒ There is a small window of instructions that shouldn't be executed, due to misprediction

- If we manage to control the mispredictions, we may be able to induce a program execute (transiently) arbitrary code

⇒ Predictors are shared between processes!





# Predictors

## Pattern History Table

```
jne 0xdeadbeef
```

Will it take the branch?

## Branch Target Buffer

```
call [rax]
```

Where will it jump?

## Return Stack Buffer

```
ret
```

Where will I return?

## Store to Load Forwarding

```
mov [rax+1], 1  
mov rdx, [rcx-1]
```

Is the same address?





# PHT - Bounds Check Bypass



```
if (x < len(array1)) {  
    y = array2[array1[x] * 4096]; }
```

- This is a dangerous loop to mispredict!
- If the loop is taken long enough, the Pattern History Table will predict it will be taken not depending on the value of x  
⇒ bypass the `if` check transiently

⇒ read `array1[x]` with arbitrary x, and then `array2` will act as covert channel!

- We have an arbitrary out of bound read in the context of a process (i.e. Javascript sandboxed program executed on your machine!)



# PHT - Bounds Check Bypass



```
if (x < len(array1)) {  
    y = array2[array1[x] * 4096]; }
```

- The predictor can be mistrained from the same process, making it repeatedly executing on safe inputs, and then attack
- But also from another process with an equivalent loop on the same address, since predictors are indexed by virtual addresses



# BTB - Branch Target Injection



## Attacker context

```
0x1000: *rdx = 0xdeadbeef
0x1001: while(true)
0x1002:     call [rdx]
```

## Victim context

```
0x1002: call [rdx]
                                spectre gadget
0xdeadbeef: A = rdi[*rsi];
```

- Attacker also controls `rdi` and `rsi` in the victim context
- Use `rsi` to read victim memory, and `rdi` as an oracle buffer for covert channel



# RSB - Return Stack Buffer



## Attacker context

```
0xdeadbeee: rdx = 0xdeadbeef  
0xdeadbeef: call rdx
```

## Victim context

```
0x1002: ret  
spectre gadget  
0xdeadbeef: A = rdi[*rsi];
```

- Attacker also controls `rdi` and `rsi` in the victim context
- Use `rsi` to read victim memory, and `rdi` as an oracle buffer for covert channel



# STL - Speculative Store Bypass



Victim context

```
mov byte [rax], 0xff  
movzx r8, byte [rcx]  
mov rcx, [rdx + r8*4096]
```

- The victim may inadvertently, leak the value that was in memory at `[rax]`
- Difficult to exploit



# Thank you!

The  
Roman  
Xploit



Questions?

