grsecurity

The AMD Branch (Mis)predictor

New Types and Methods of Straight-Line Speculation (SLS) Vulnerabilities

April 2022
hardwear.io webinar

Pawel Wieczorkiewicz
Open Source Security, Inc.

whoami

- Pawel Wieczorkiewicz
  - Email: wipawel@grsecurity.net
  - Twitter: @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

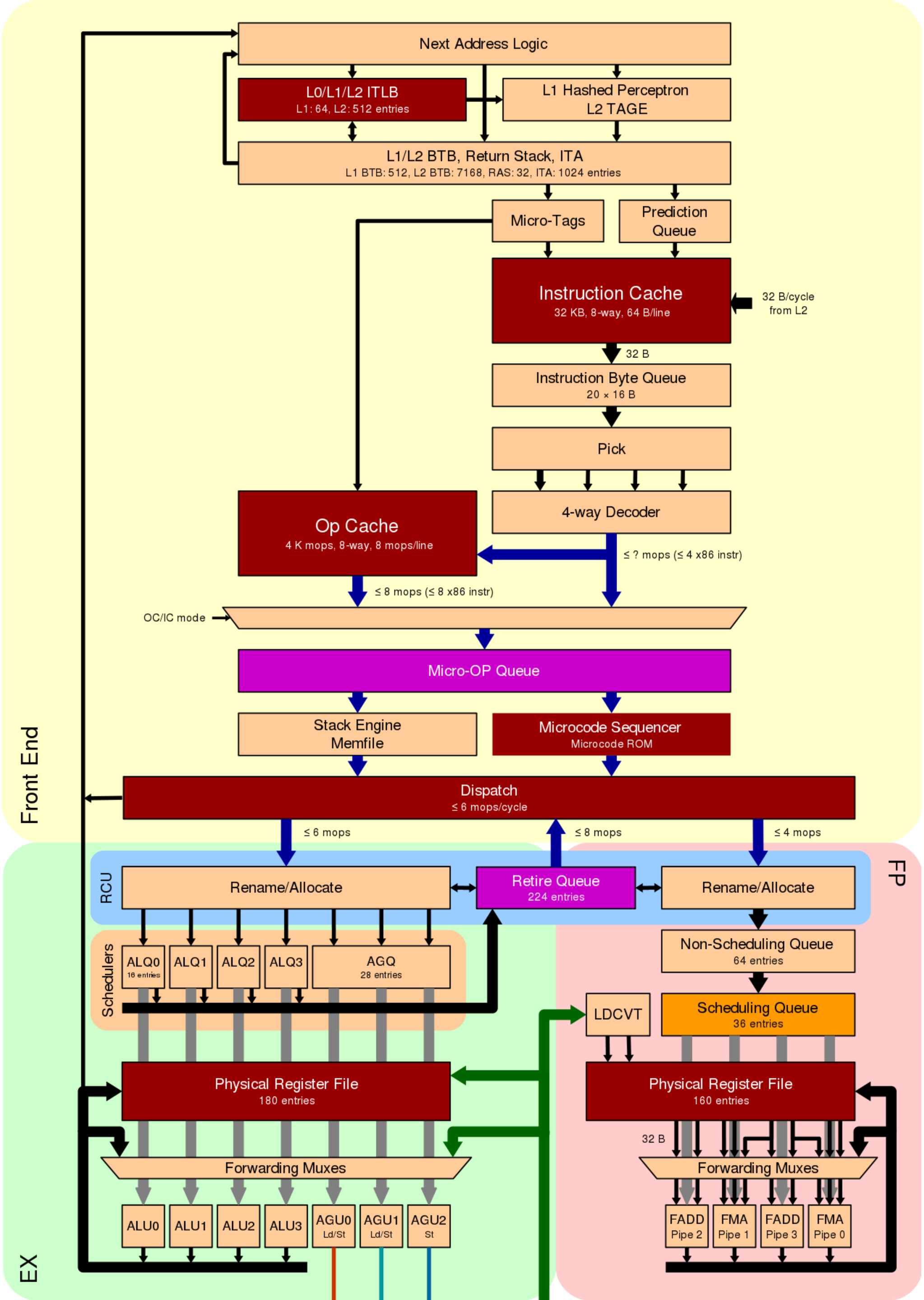
Outline

- **Theory**
    - Quick AMD microarchitecture overview
    - Branch predictors
        - Basic introduction
        - Purpose
        - Building blocks and functionality
        - Different types of branches
    - 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)
    - 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
    - SLS mitigations

Microarchitecture - overview

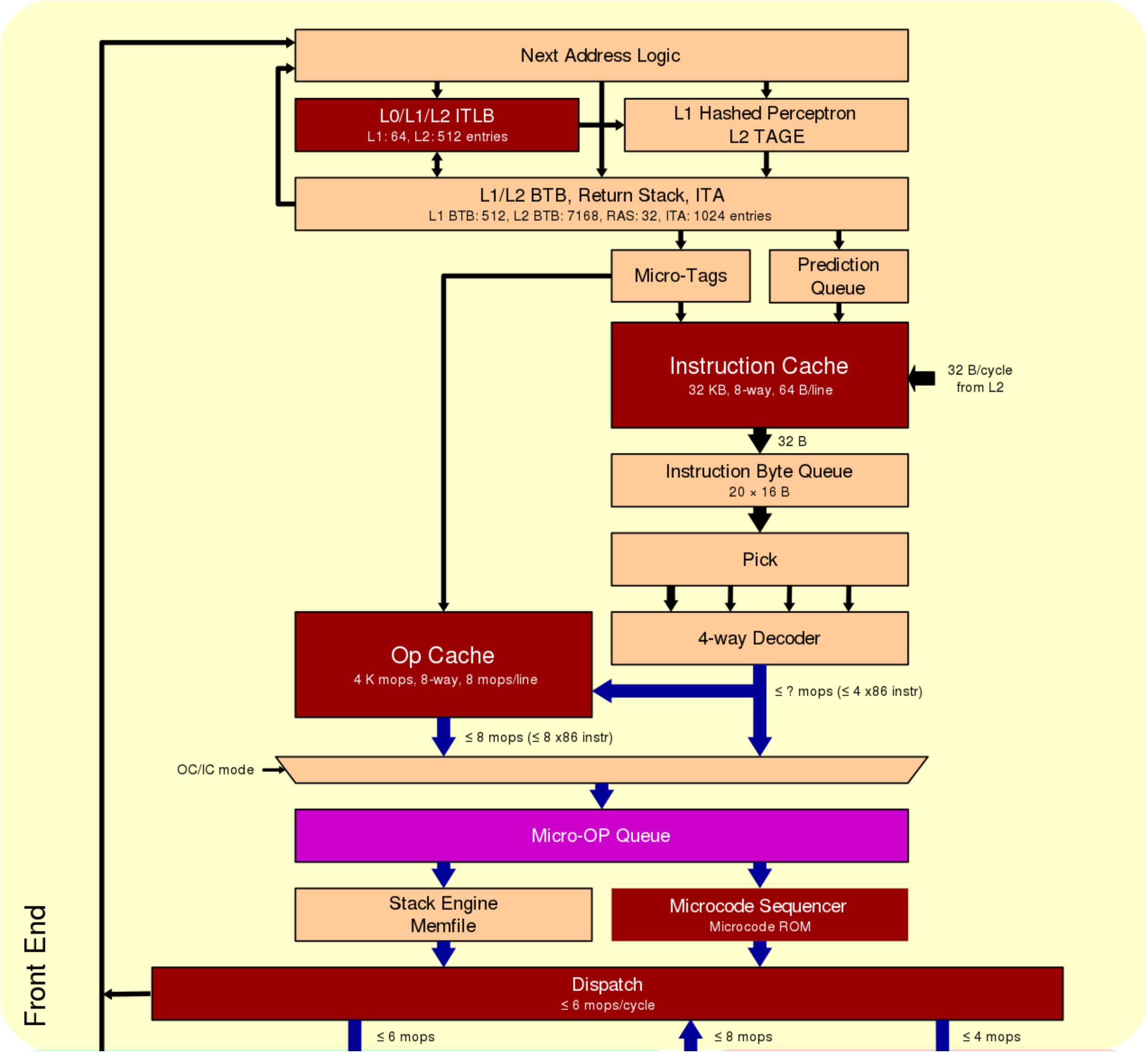
- AMD Zen2 microarchitecture



source: en.wikichip.org

Microarchitecture - overview

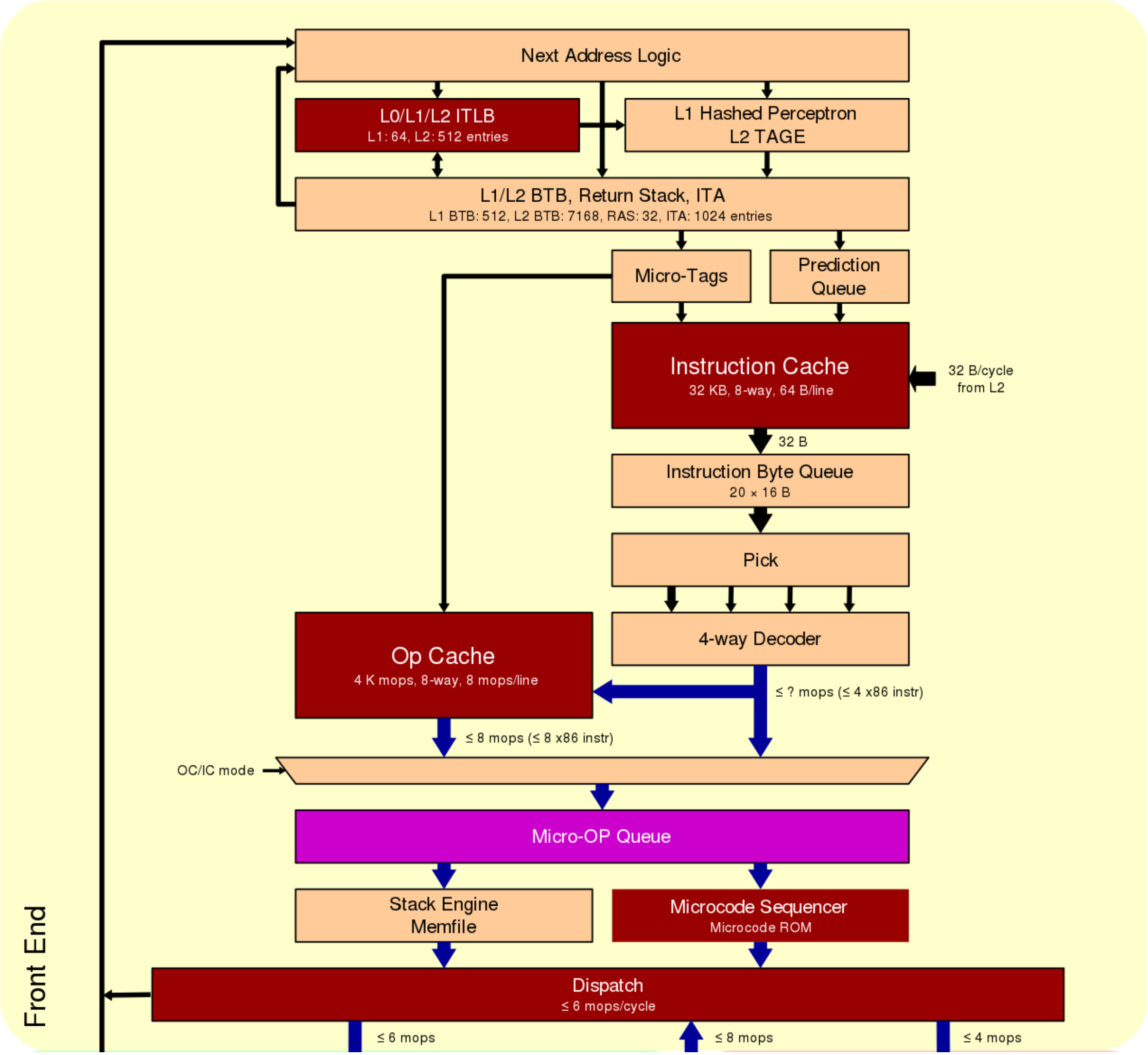
- AMD Zen2 microarchitecture
  - Frontend



source: en.wikichip.org

Microarchitecture - overview

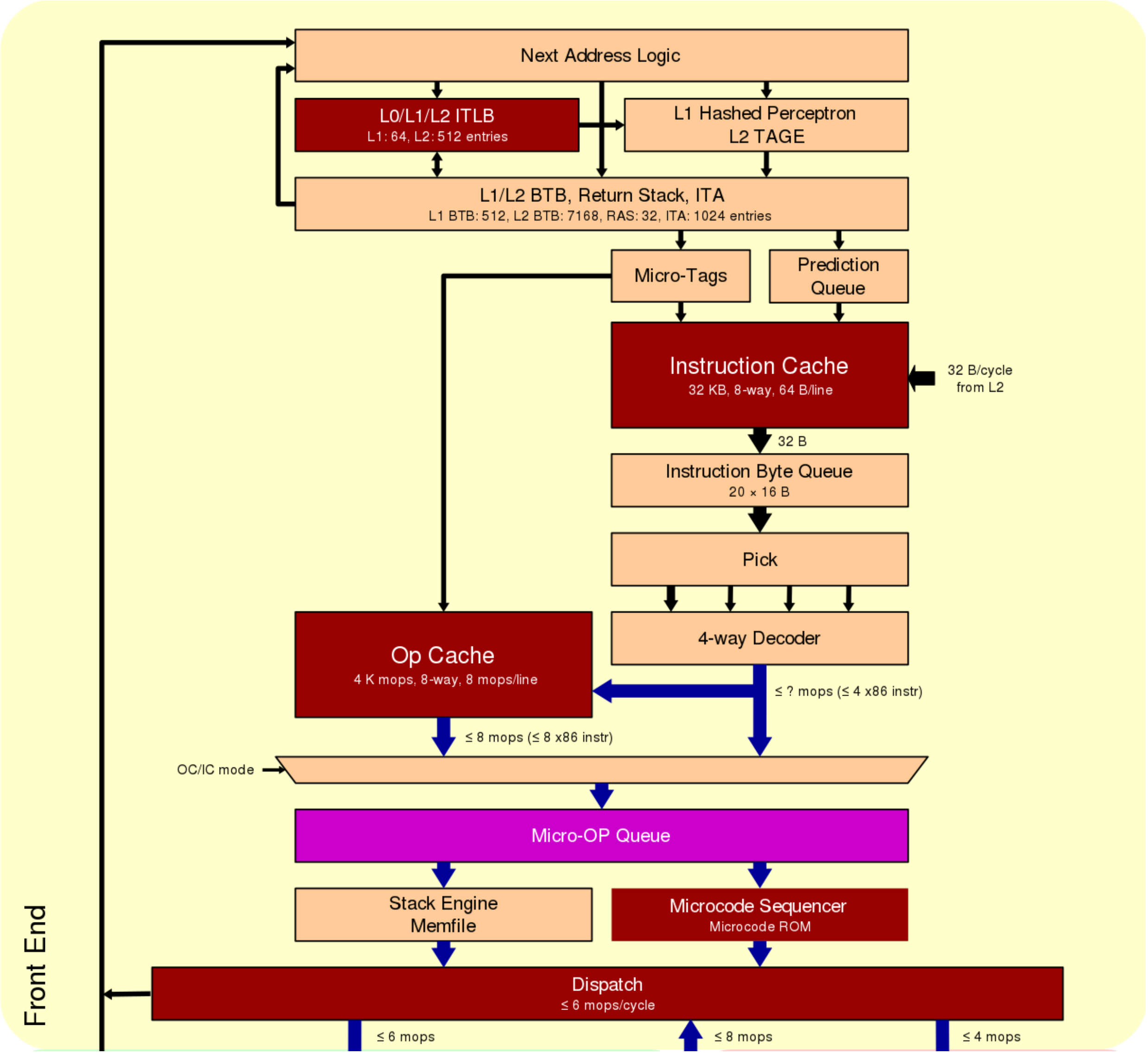
- AMD Zen2 microarchitecture
  - Frontend
    - Fetch



source: en.wikichip.org

Microarchitecture - overview

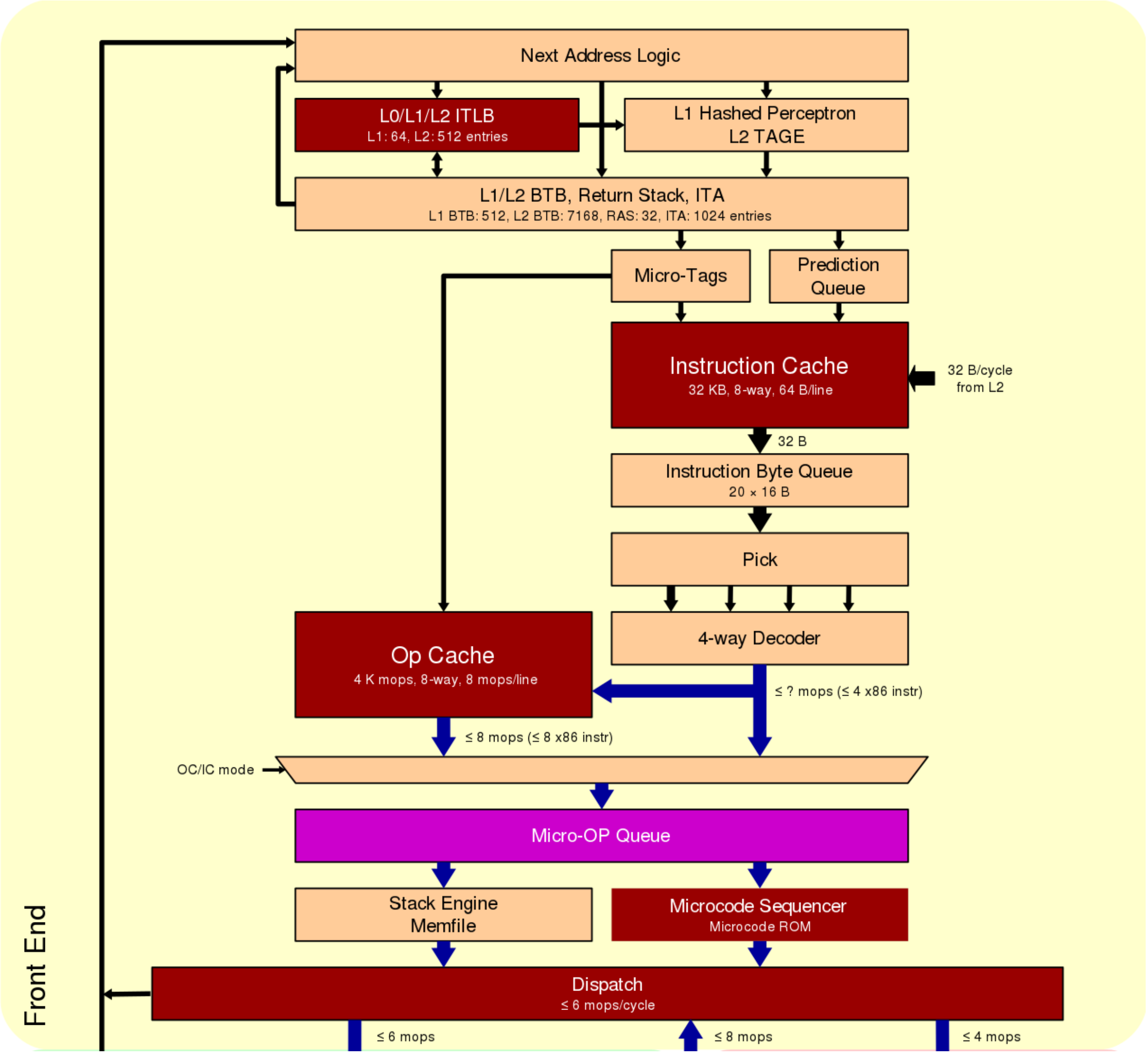
- AMD Zen2 microarchitecture
  - Frontend
    - Fetch
    - Decode



source: en.wikichip.org

Microarchitecture - overview

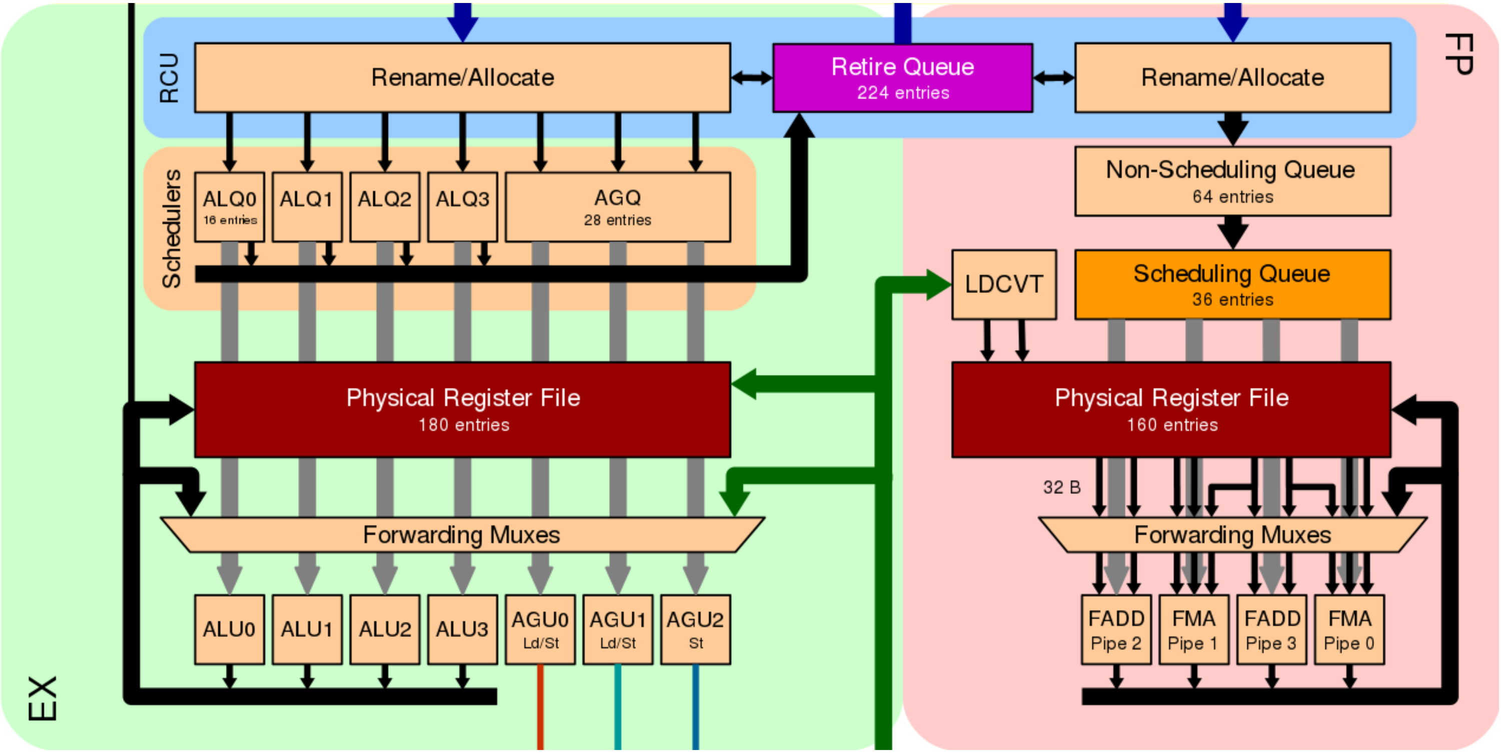
- AMD Zen2 microarchitecture
  - Frontend
    - Fetch
    - Decode
    - Dispatch



source: en.wikichip.org

Microarchitecture - overview

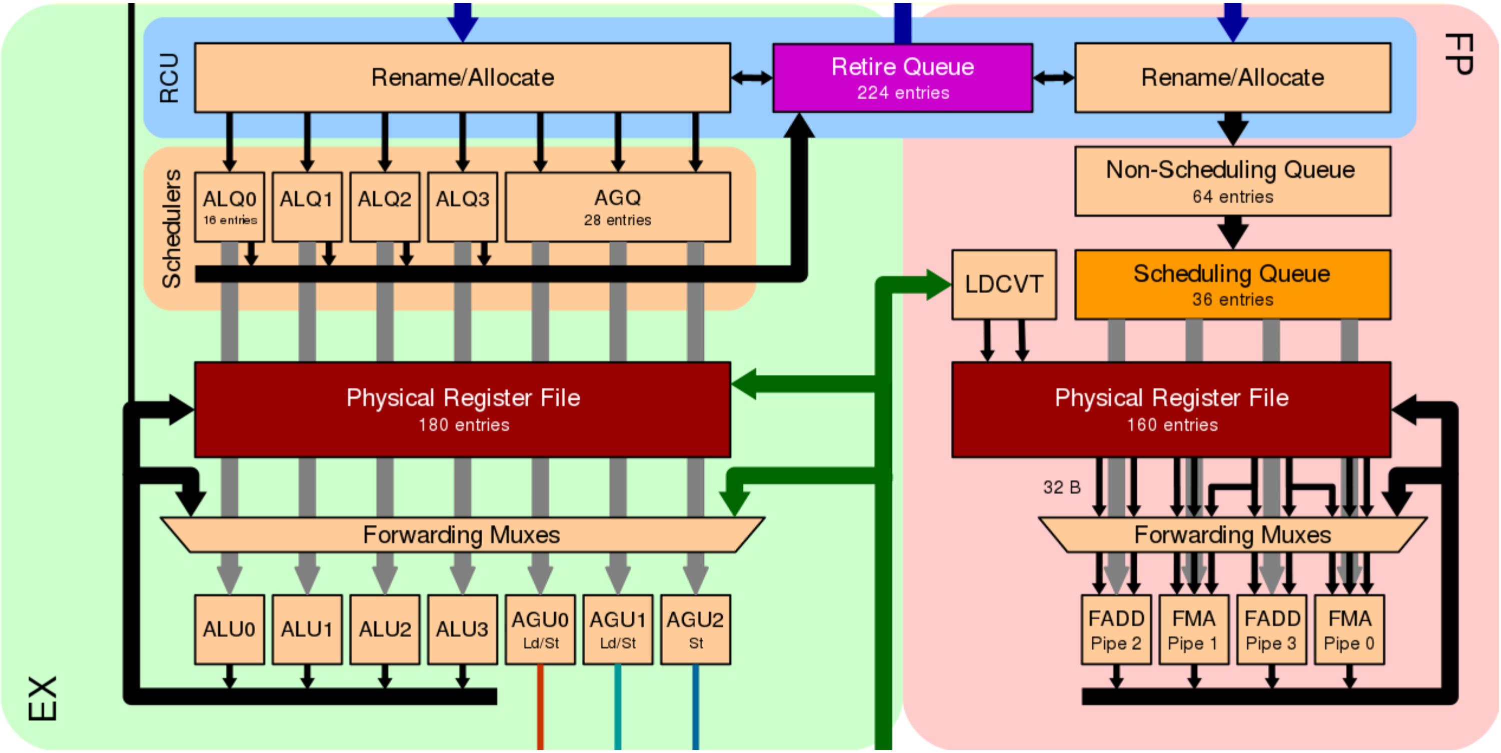
- AMD Zen2 microarchitecture
    - Backend



source: en.wikichip.org

Microarchitecture - overview

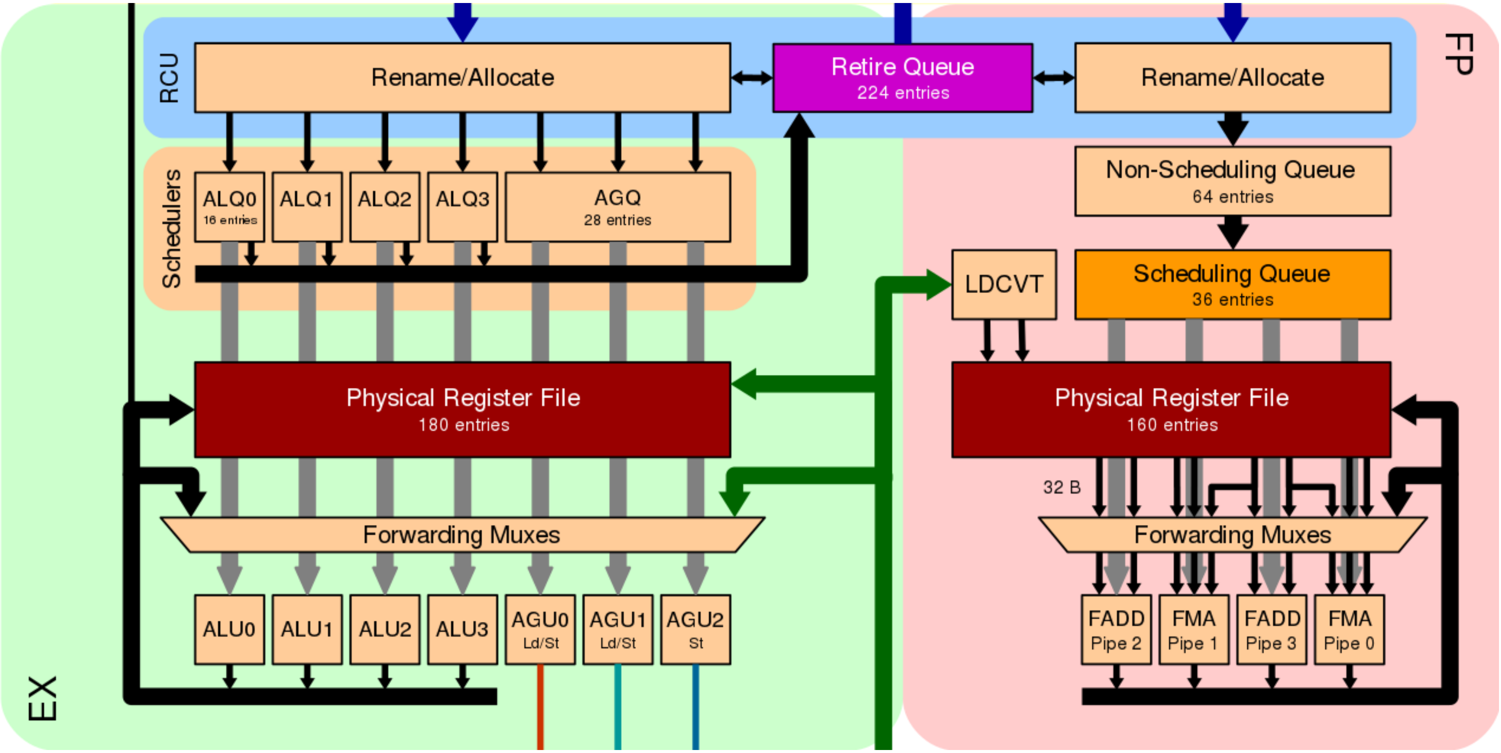
- AMD Zen2 microarchitecture
  - Backend
    - Superscalar



source: en.wikichip.org

Microarchitecture - overview

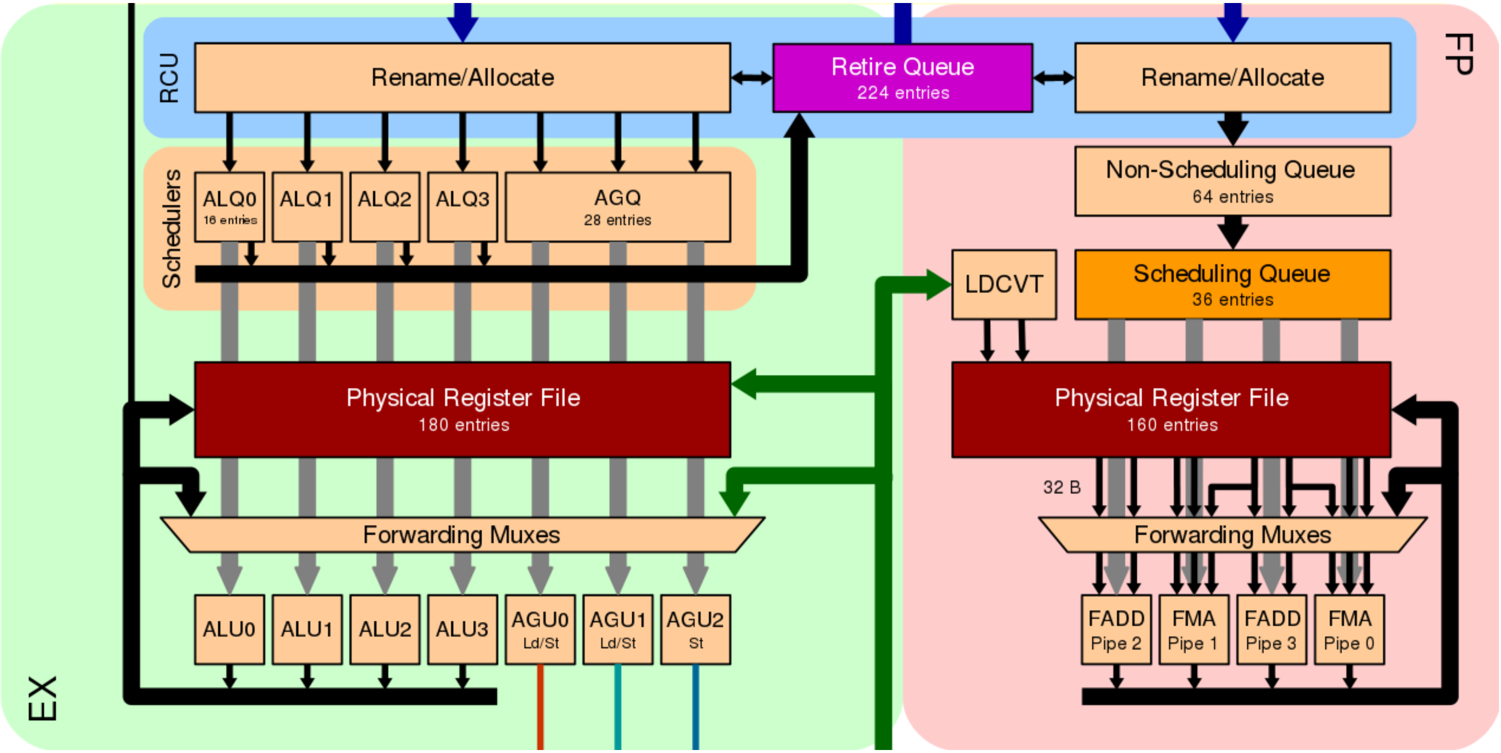
- AMD Zen2 microarchitecture
    - Backend
        - Superscalar
        - Out-of-order execution



source: en.wikichip.org

Microarchitecture - overview

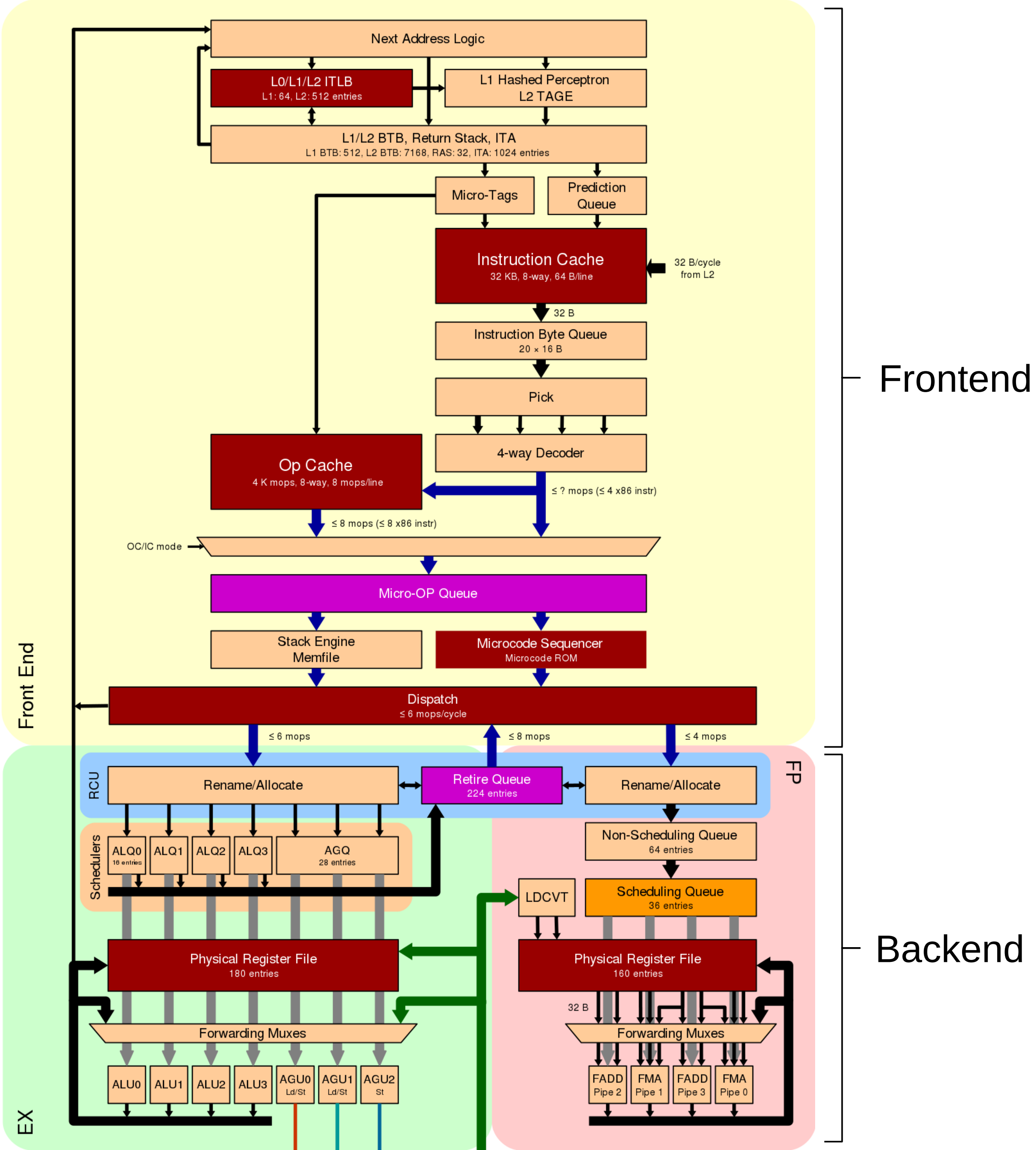
- AMD Zen2 microarchitecture
    - Backend
        - Superscalar
        - Out-of-order execution
        - In-order retire



source: en.wikichip.org

Microarchitecture - overview

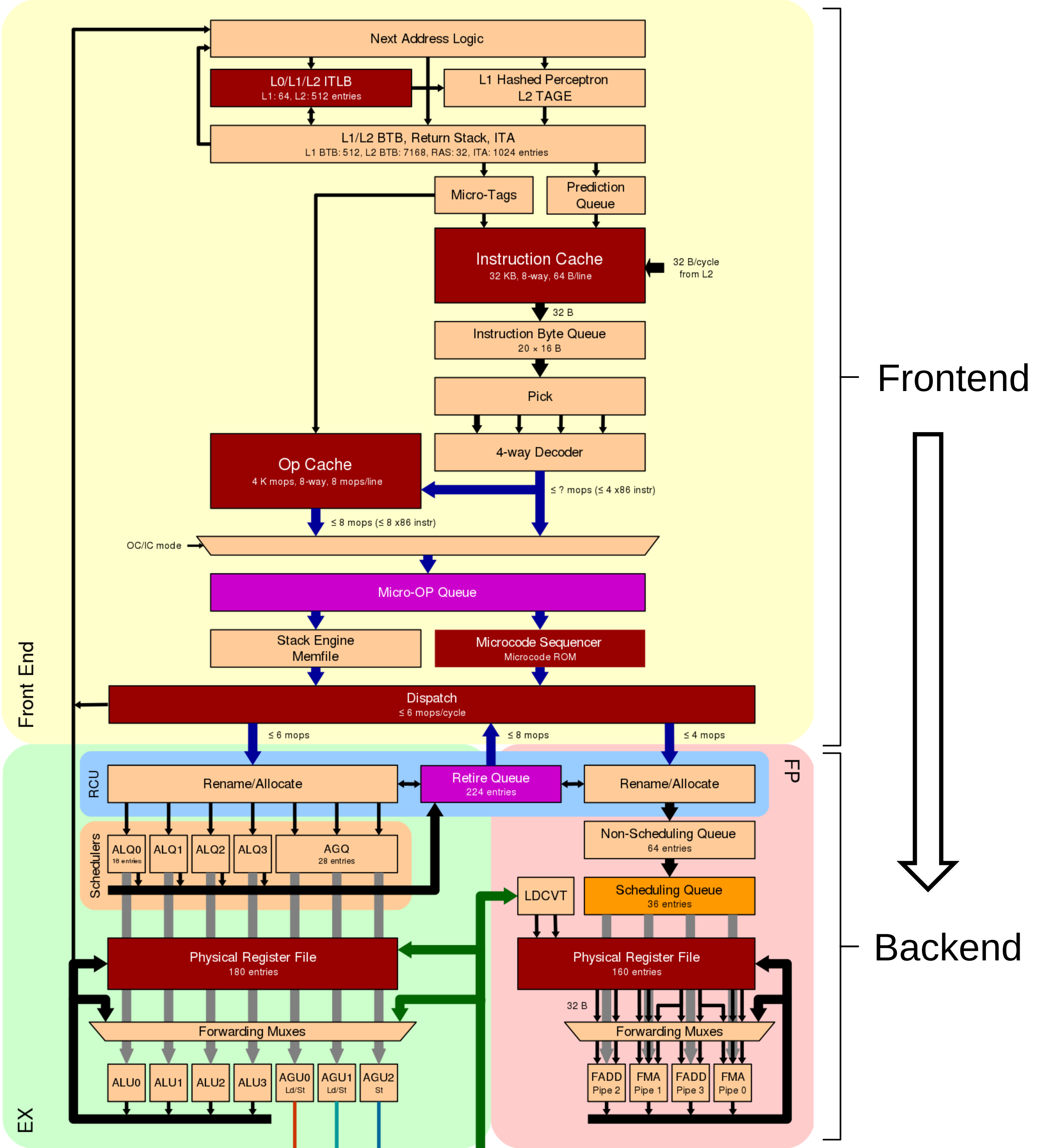
- AMD Zen2 microarchitecture
  - Frontend
    - Fetch
    - Decode
    - Dispatch
  - Backend
    - Superscalar
    - Out-of-order execution
    - In-order retire



source: en.wikichip.org

Microarchitecture - overview

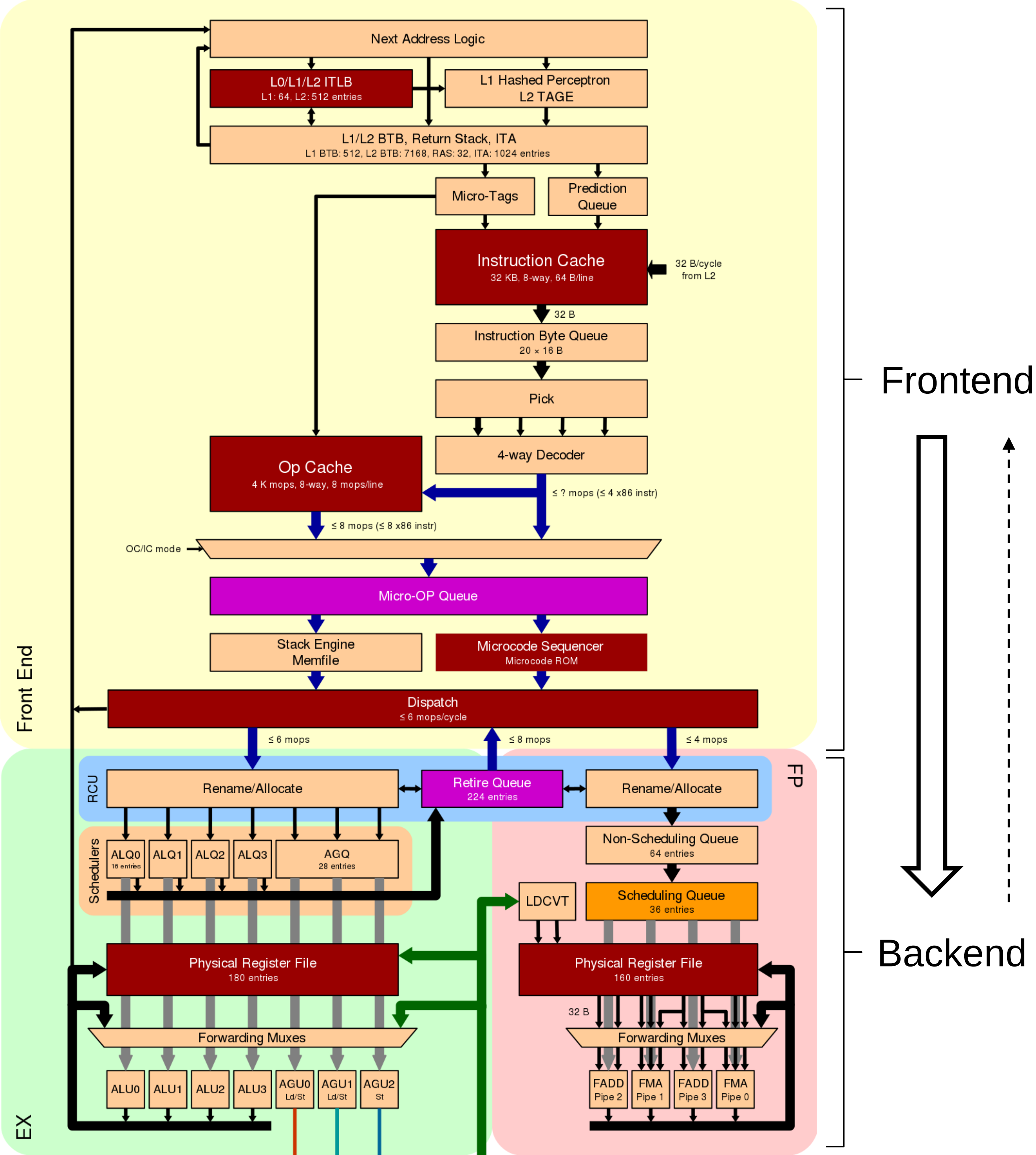
- AMD Zen2 microarchitecture
  - Frontend
    - Fetch
    - Decode
    - Dispatch
  - Backend
    - Superscalar
    - Out-of-order execution
    - In-order retire



source: en.wikichip.org

Microarchitecture - overview

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



source: en.wikichip.org

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 will stall the CPU
        - Must be avoided
        - Some definitive information available only in the Backend
            - Frontend must **predict** the likely outcome upfront
                - Correct prediction ➔ performance win
                - Misprediction ➔ penalty, Frontend re-steer when Backend detects
                - The better (more accurate) prediction rate, the better performance (fewer bubbles)
    - 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

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 branches taken (loops accuracy)
  - Forward 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

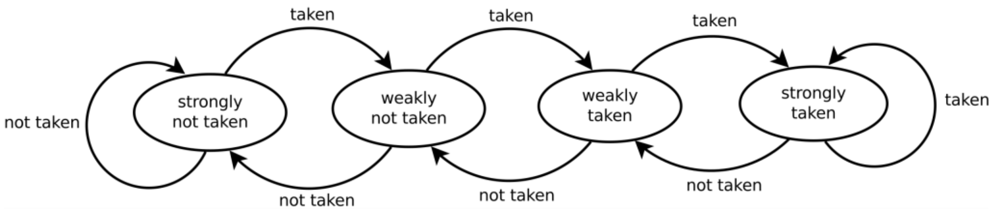
- Branch Prediction Unit (BPU)
    - Many different designs and categories
        - Static vs **Dynamic**
        - **One-Level** vs Two-level
        - Local vs Global
        - Adaptive
        - Agree
        - Hybrid
        - Neural (Machine Learning)
            - Perceptron-based (AMD Zen2)

- Prediction based on previous executions 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)

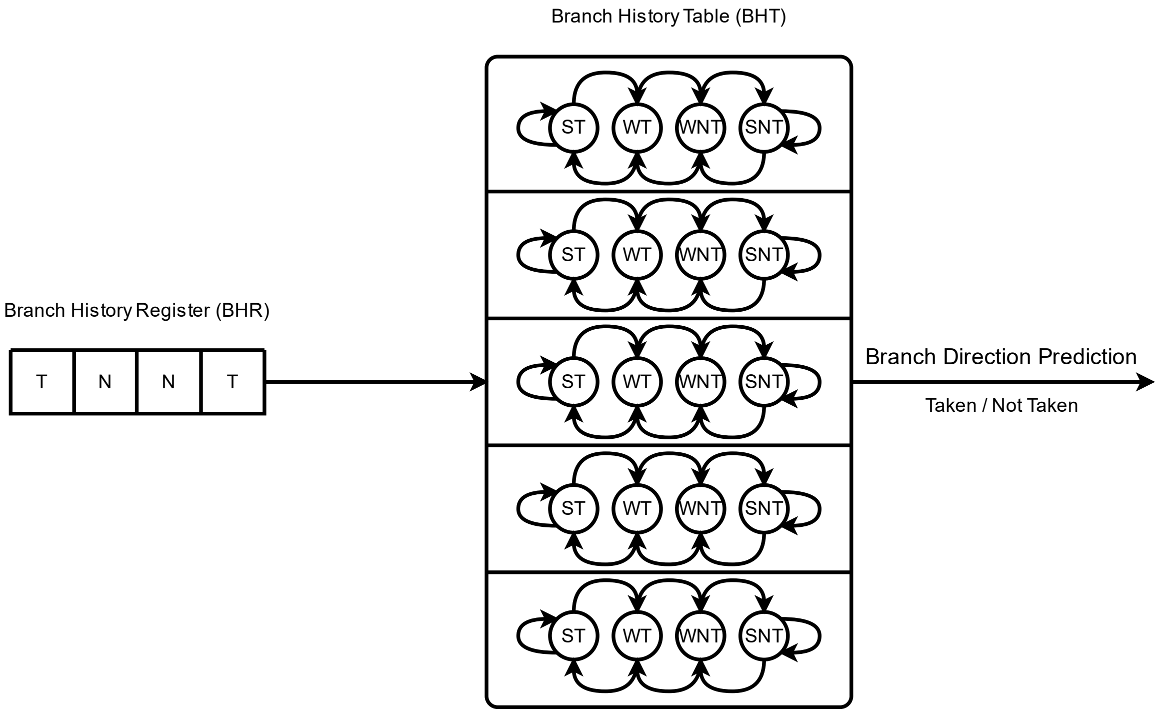
- Prediction based on previous executions results of a given branch
  - If taken before, likely to be taken again
    - 1-bit saturation counter
      - Previously taken or not taken
    - 2-bit saturation counter
      - Four states state machine



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



Branch History Table (BHT)

Branch History Register (BHR)

T N N T

ST WT WNT SNT

ST WT WNT SNT

ST WT WNT SNT

ST WT WNT SNT

ST WT WNT SNT

Branch Direction Prediction

Taken / 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)

- Branch History Table is indexed using a distinct branch history register for each encountered conditional branch

Branch predictors - design and building blocks

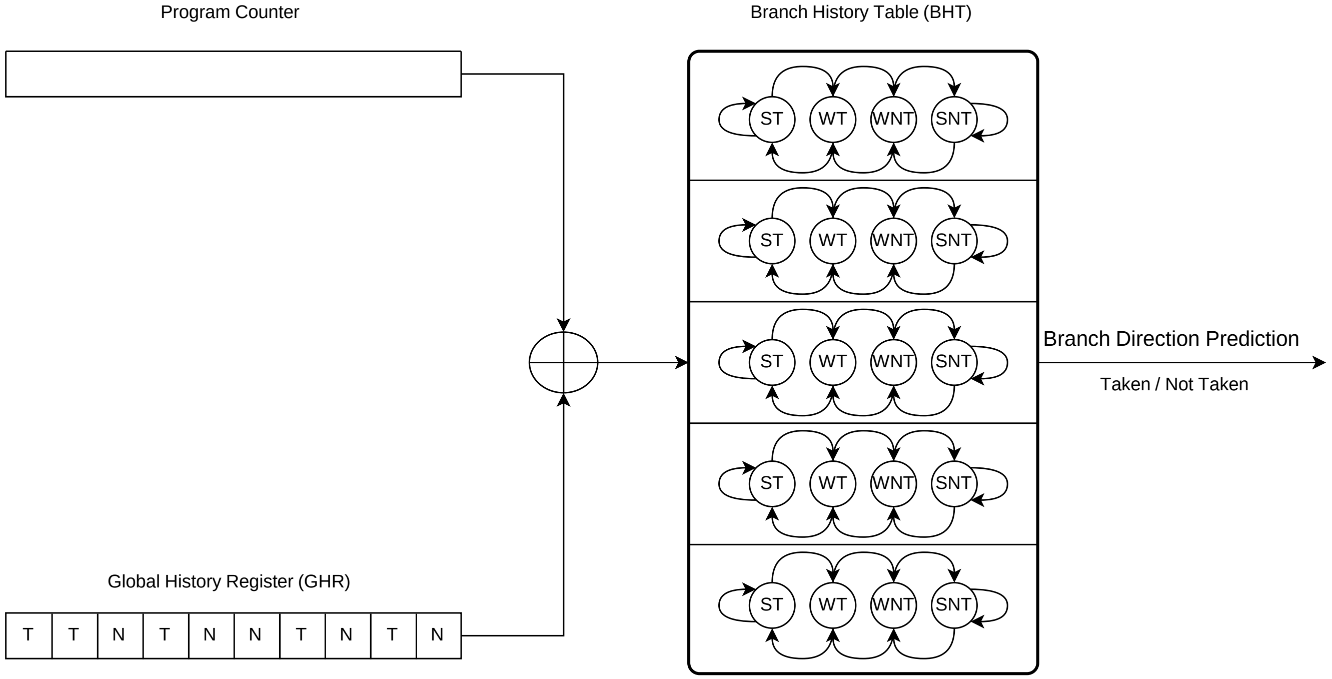
- Branch Prediction Unit (BPU)
  - Many different designs and categories
    - Static vs **Dynamic**
    - One-Level vs **Two-level**
    - Local vs **Global**
    - **Adaptive**
    - Agree
    - Hybrid
    - Neural (Machine Learning)
      - Perceptron-based (AMD Zen2)

- Branch History Table is indexed using a distinct branch history register for each encountered conditional branch
- Branch History Table is indexed using a shared (global) branch history register for all encountered conditional branches
  - Correlation between different branches is considered
  - 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


Program Counter
Branch History Table (BHT)
ST
WT
WNT
SNT
Branch Direction Prediction
Taken / Not Taken
Global History Register (GHR)
T T N T N N T N T N

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** at all

Branch predictors - design and building blocks

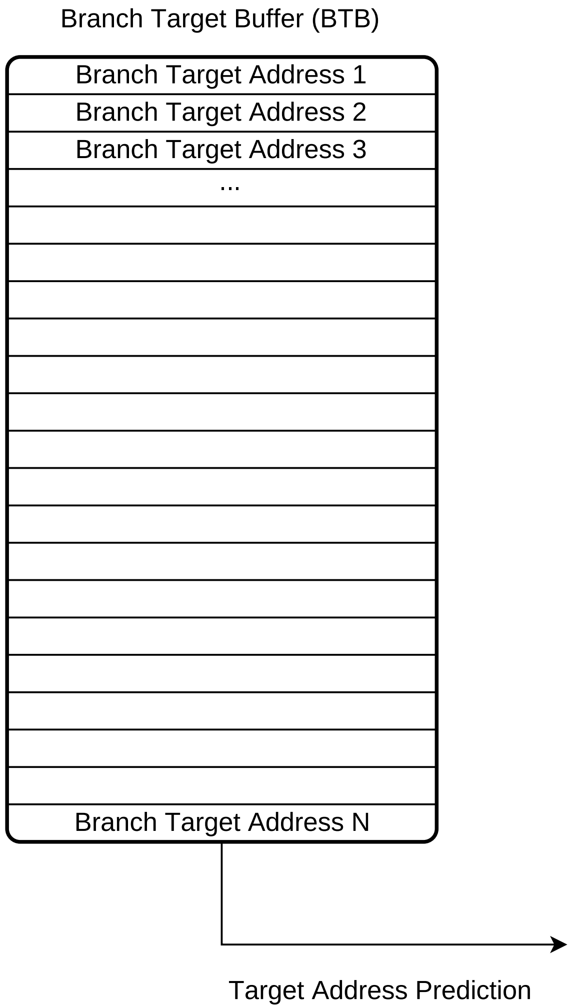
- So far, we have been implicitly focusing on direct conditional branch predictions
    - Taken / Not taken
    - Question: **IF** at all
- What about other branch types?
    - Do they need a branch predictor too?

Branch predictors - design and building blocks

- So far, we have been implicitly focusing on direct conditional branch predictions
  - Taken / Not taken
  - Question: **IF** at all
- What about other branch types?
  - Do they need a branch predictor too?
    - Yes, they do!
    - Question: **Where-to**

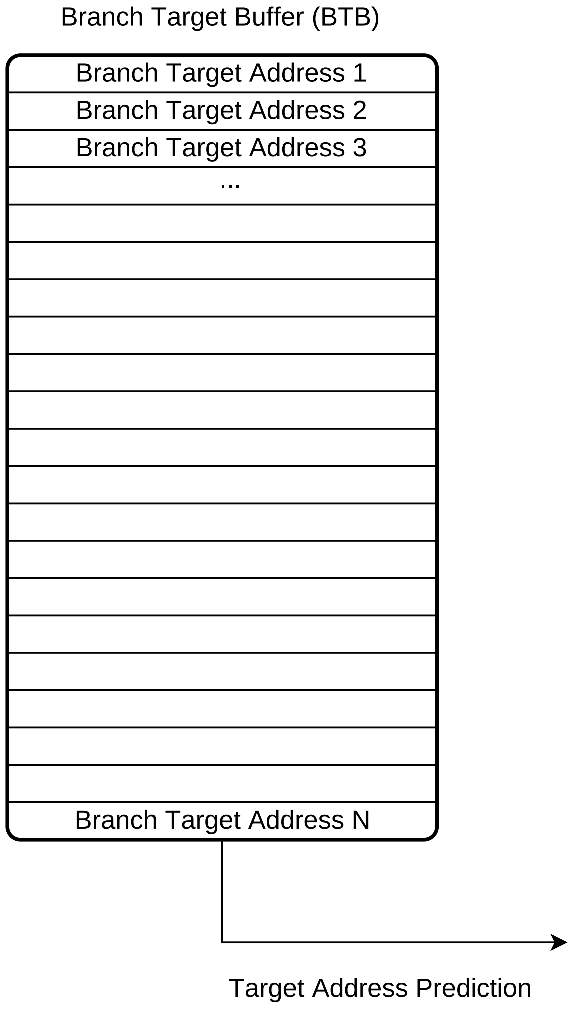
Branch predictors - design and building blocks

- So far, we have been implicitly focusing on direct conditional branch predictions
  - Taken / Not taken
  - Question: **IF** branch at all
- What about other branch types?
  - Do they need a branch predictor too?
    - Yes, they do!
    - Question: **Where-to**
  - Another important BPU component:
    - Branch Target Buffer (BTB)



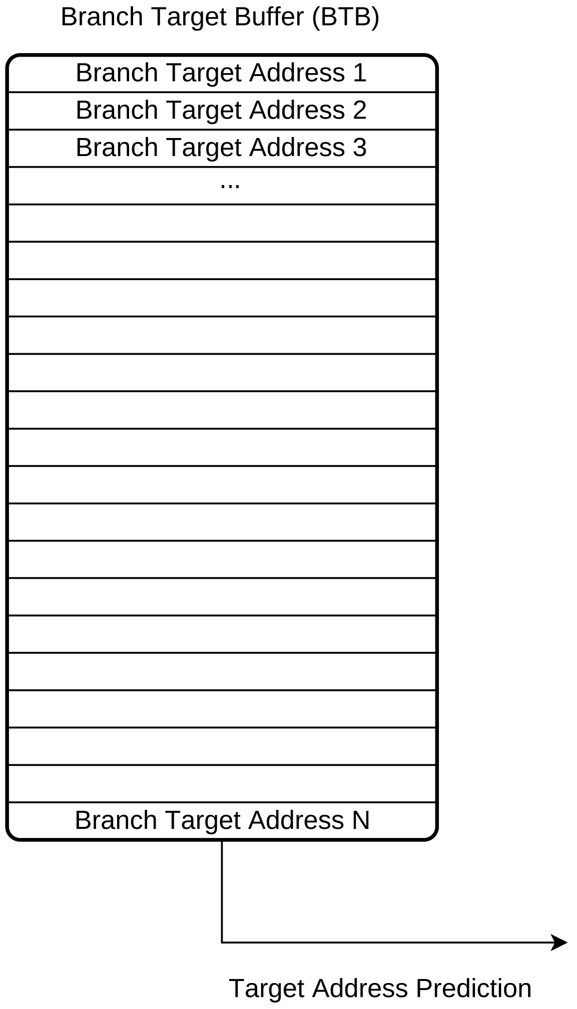
Branch predictors – branch target buffer

- Predicts address of next instructions after the control flow changes because of a branch
- Turns out: **ALL branch types need BTB!**
  - Frontend fetches and decodes, but does not execute instructions
  - Frontend needs to know where to fetch next instructions from upon a branch
    - It must not wait for Backend
      - Performance!
  - Hence, BPU is a Frontend’s component and leverages BTB to steer Frontend upon branches

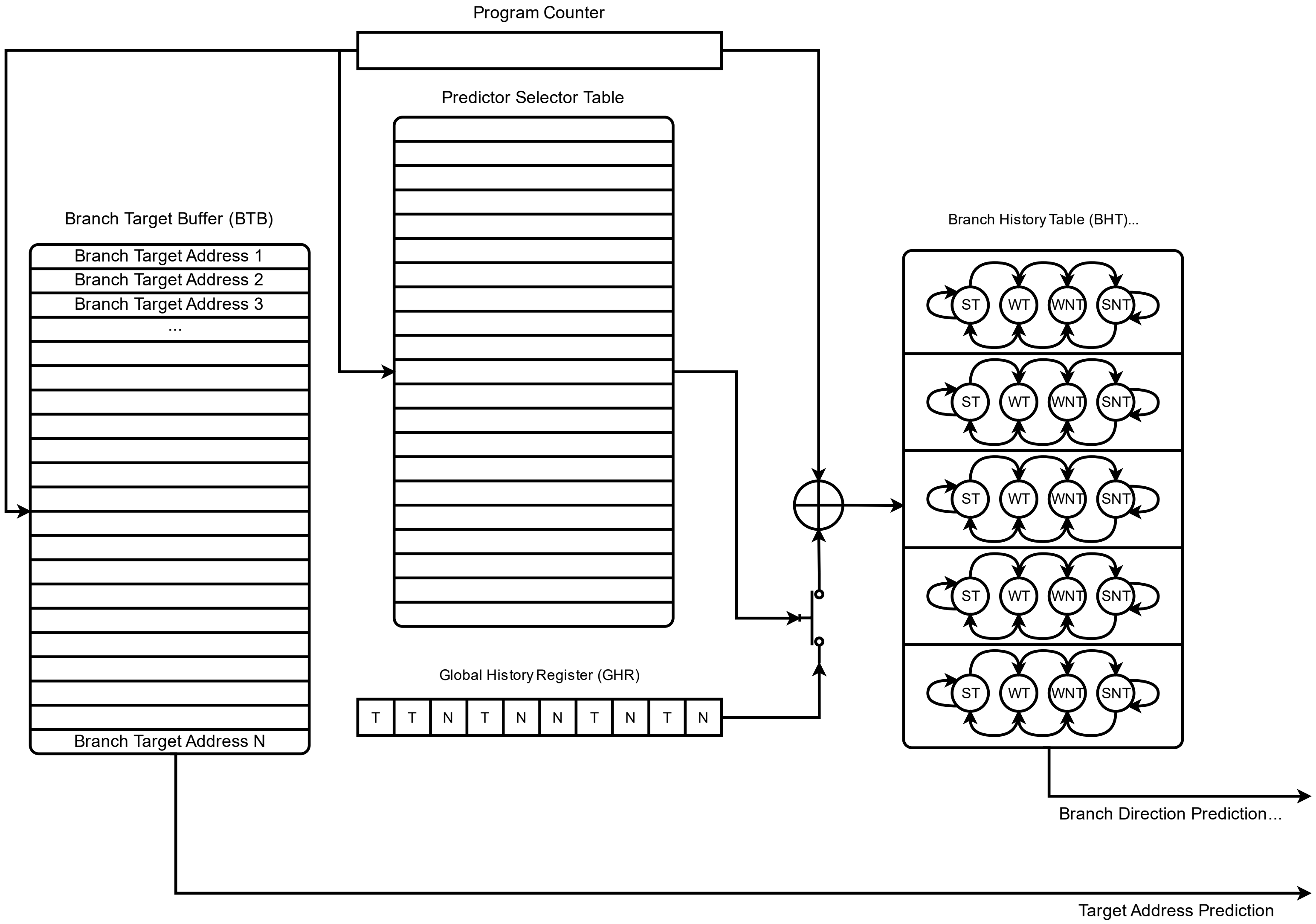


Branch predictors – branch target buffer

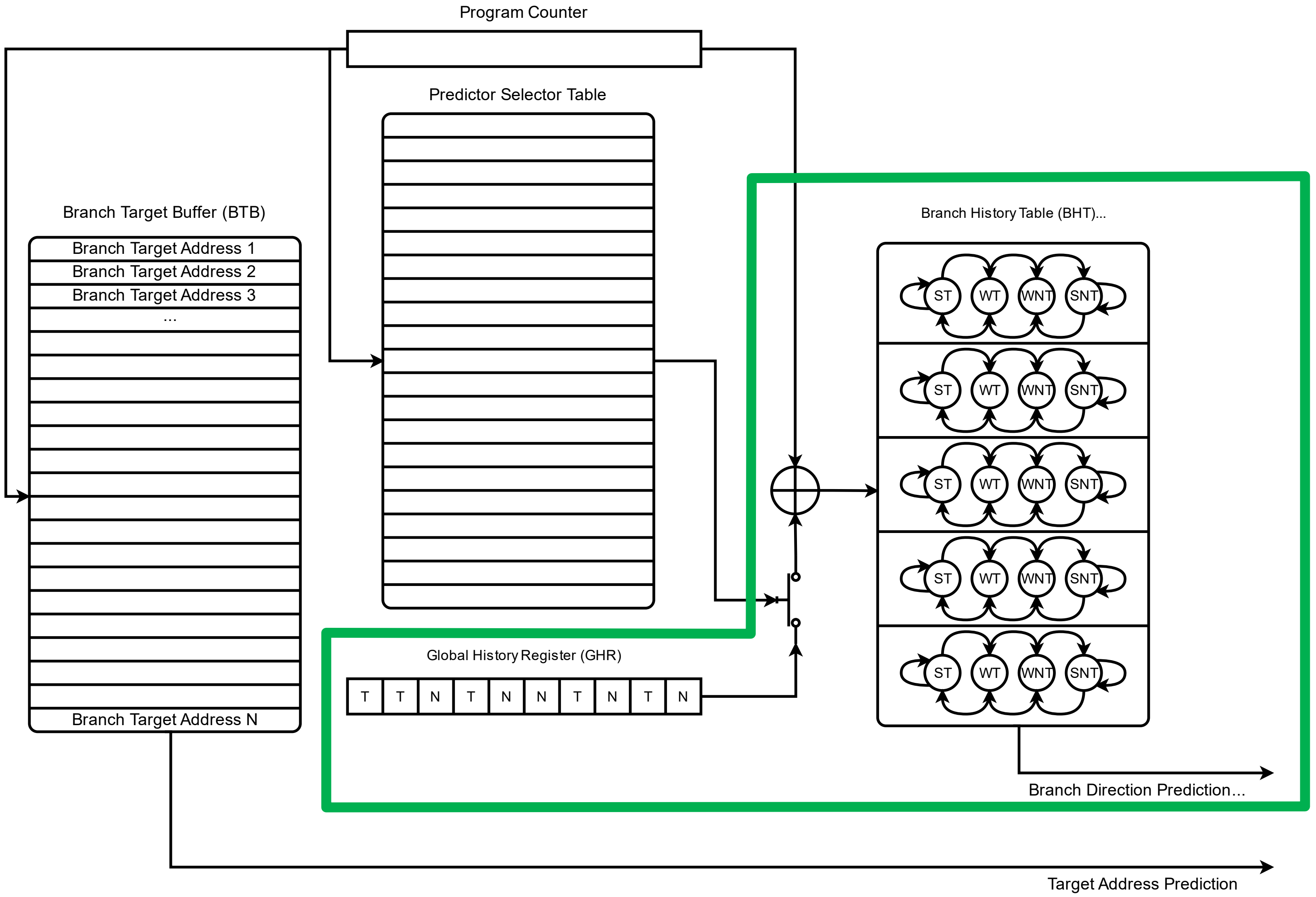
- Analyzing branch instructions addressing is backend’s job
- Where-To problem:
    - Direct conditional branches:
        - Not taken ➔ next instruction
            - easy
        - Taken ➔ where-to?
            - backward, forward, not easy
    - Direct unconditional branches:
        - Always taken ➔ where?
            - backward, forward, not easy
    - Indirect unconditional branches:
        - Always taken ➔ where?
            - backward, forward, not easy
        - Target address may change at runtime, not static
            - static prediction will not do
            - BTB is crucial for performance



Hybrid branch predictor – example



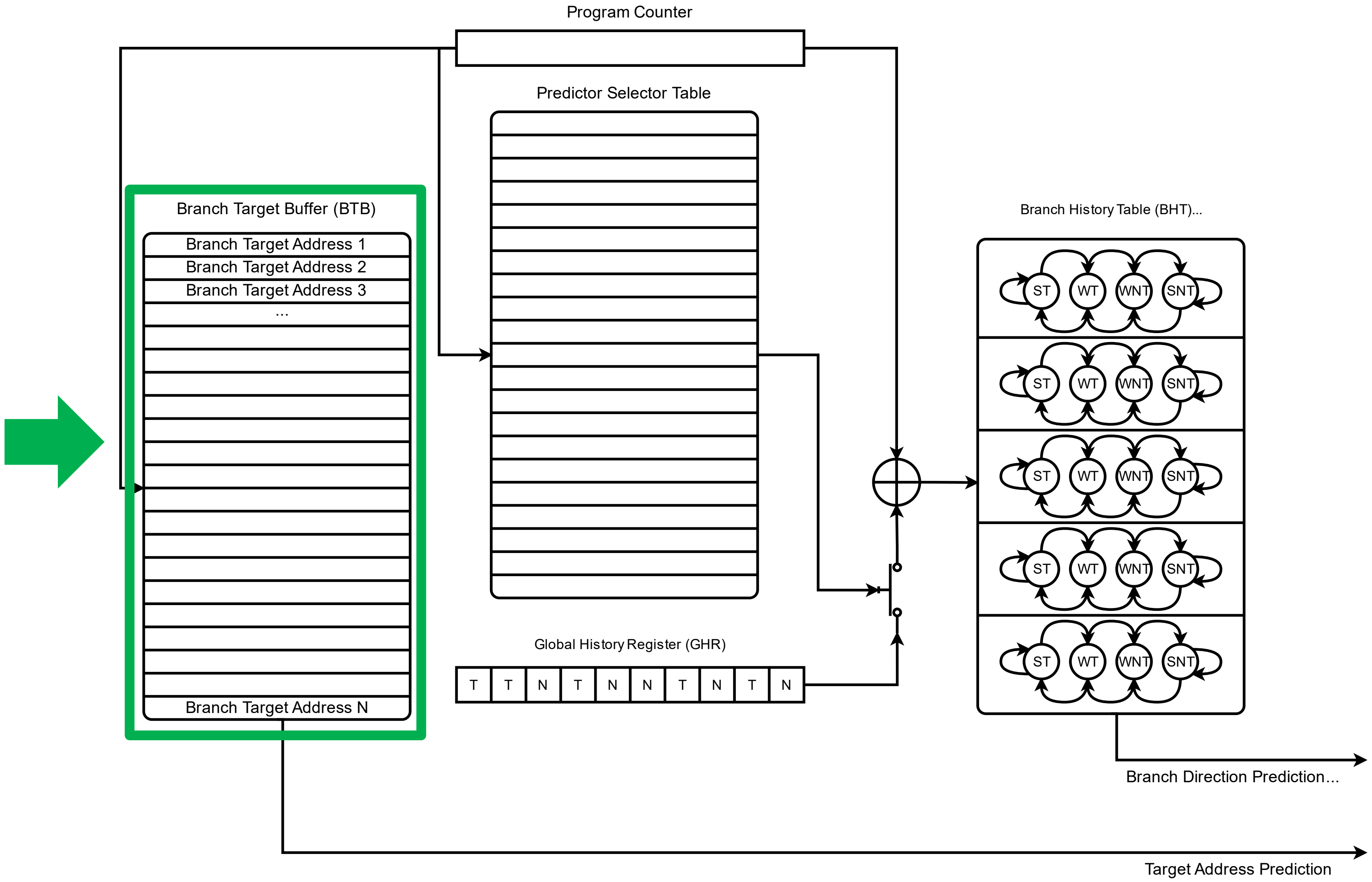
Hybrid branch predictor – building blocks



Answer: IF

Hybrid branch predictor – building blocks

Answer: Where-to



Branch predictors – different types of branches

- Direct
  - Conditional
    - Jumps
      - Taken
      - Not Taken
  - Unconditional
    - Jumps
    - Calls
- Indirect
  - Unconditional
    - Jumps
    - Calls
    - Function return

Branch predictors – different types of branches

- **Direct**
  - **Conditional**
    - **Jumps**
      - Taken
      - Not Taken
  - Unconditional
    - Jumps
    - Calls
- Indirect
  - Unconditional
    - Jumps
    - Calls
    - Function return

- x86: *Jcc* $address
  - Control flow change to the specified $address, **when** condition is met
  - Condition *cc* is based on the state of the status flags (EFLAGS register)
    - JA – jump if above
      - Status flags: CF=0 and ZF=0
    - JB – jump if below
      - Status flags: CF=1
    - JE – jump if equal
      - Status flags: ZF=1
    - JNE – jump if not equal
      - Status flags: ZF=0

Branch predictors – different types of branches

- **Direct**
    - **Conditional**
        - **Jumps**
            - **Taken**
            - Not Taken
    - Unconditional
        - Jumps
        - Calls
- Indirect
    - Unconditional
        - Jumps
        - Calls
        - Function return

- Example (Taken)

```
a = 0
if (a == 0)
    *addr = %rax
else
    %rax = *addr
```

```
xor %rdi, %rdi ; set ZF=1
test %rdi, %rdi ; set ZF=1
je END_LABEL ; if ZF==1 goto END_LABEL
mov (%rsi), %rax ; memory load
END_LABEL:
mov %rax, (%rsi) ; memory store
```

Branch predictors – different types of branches

- **Direct**
  - **Conditional**
    - **Jumps**
      - Taken
      - **Not Taken**
  - Unconditional
    - Jumps
    - Calls
- Indirect
  - Unconditional
    - Jumps
    - Calls
    - Function return

- Example (Not Taken)

```
a = 1
if (a == 0)
    *addr = %rax
else
    %rax = *addr
```

```
mov $1, %rdi
test %rdi, %rdi ; set ZF=0
je END_LABEL ; if ZF==1 goto END_LABEL
mov (%rsi), %rax ; memory load
END_LABEL:
mov %rax, (%rsi) ; memory store
```

Branch predictors – different types of branches

- **Direct**
  - Conditional
    - Jumps
      - Taken
      - Not Taken
  - **Unconditional**
    - **Jumps**
    - Calls
- Indirect
  - Unconditional
    - Jumps
    - Calls
    - Function return

- x86: JMP $address
  - Unconditional control flow change to the specified $address, without return
  - Direct – target address static
    - Part of the instruction
  - Used by compilers to implement:
    - Loops
    - Tail calls
    - Sharing common code blocks
      - Error handling code
    - …
  - Other uses:
    - RAP – jumping over meta-data in code
    - Live patching
    - …

Branch predictors – different types of branches

- **Direct**
  - Conditional
    - Jumps
      - Taken
      - Not Taken
  - **Unconditional**
    - Jumps
    - **Calls**
- Indirect
  - Unconditional
    - Jumps
    - Calls
    - Function return

- x86: CALL $address
  - Unconditional control flow change to the specified $address with return
  - Direct – target address static
    - Part of the instruction
  - CALL instruction ➔ push %rip; jmp $address
  - Execution flow is expected to resume at the CALL following instruction eventually
  - Used by compilers to implement:
    - Procedure calls
    - Recursive calls
    - …
  - Other uses:
    - __x86.get_pc_thunk.* – position independent code execution helper on i386/i686
    - …

Branch predictors – different types of branches

- Direct
    - Conditional
        - Jumps
            - Taken
            - Not Taken
    - Unconditional
        - Jumps
        - Calls
- **Indirect**
    - **Unconditional**
        - **Jumps**
        - Calls
        - Function return

- x86: JMP reg (or [mem])
    - Unconditional control flow change to the dynamic address specified via register or memory dereference, without return
    - Indirect – target address dynamic
        - May change at runtime
        - Specified by register or memory location
            - i386: absolute address
            - x64: pc-relative offset
    - Used by compilers to implement:
        - Tail calls
        - Jump tables
        - Switch-case
        - Virtual function tables (C++)
        - Multiway conditional branches

Branch predictors – different types of branches

- Direct
    - Conditional
        - Jumps
            - Taken
            - Not Taken
    - Unconditional
        - Jumps
        - Calls
- **Indirect**
    - **Unconditional**
        - Jumps
        - **Calls**
        - Function return

- x86: CALL reg (or [mem])
    - Unconditional control flow change to the dynamic address specified via register or memory dereference, with return
    - Indirect – target address dynamic
        - May change at runtime
        - Specified by register or memory location
            - i386: absolute address
            - x64: pc-relative offset
    - Used by compilers to implement:
        - Function pointers
        - Virtual functions (C++)
        - Position independent code

Branch predictors – different types of branches

- Direct
    - Conditional
        - Jumps
            - Taken
            - Not Taken
    - Unconditional
        - Jumps
        - Calls
- **Indirect**
    - **Unconditional**
        - Jumps
        - Calls
        - **Function return**

- x86: RET
    - Unconditional control flow change to the $address located on stack
    - Indirect – target address dynamic
        - May change at runtime
        - Stored on stack upon function call
    - Used by compilers to implement:
        - Function returns
        - Retpoline
    - Does not use BTB, but Return Stack Buffer (RSB) aka Return Address Stack (RAS)

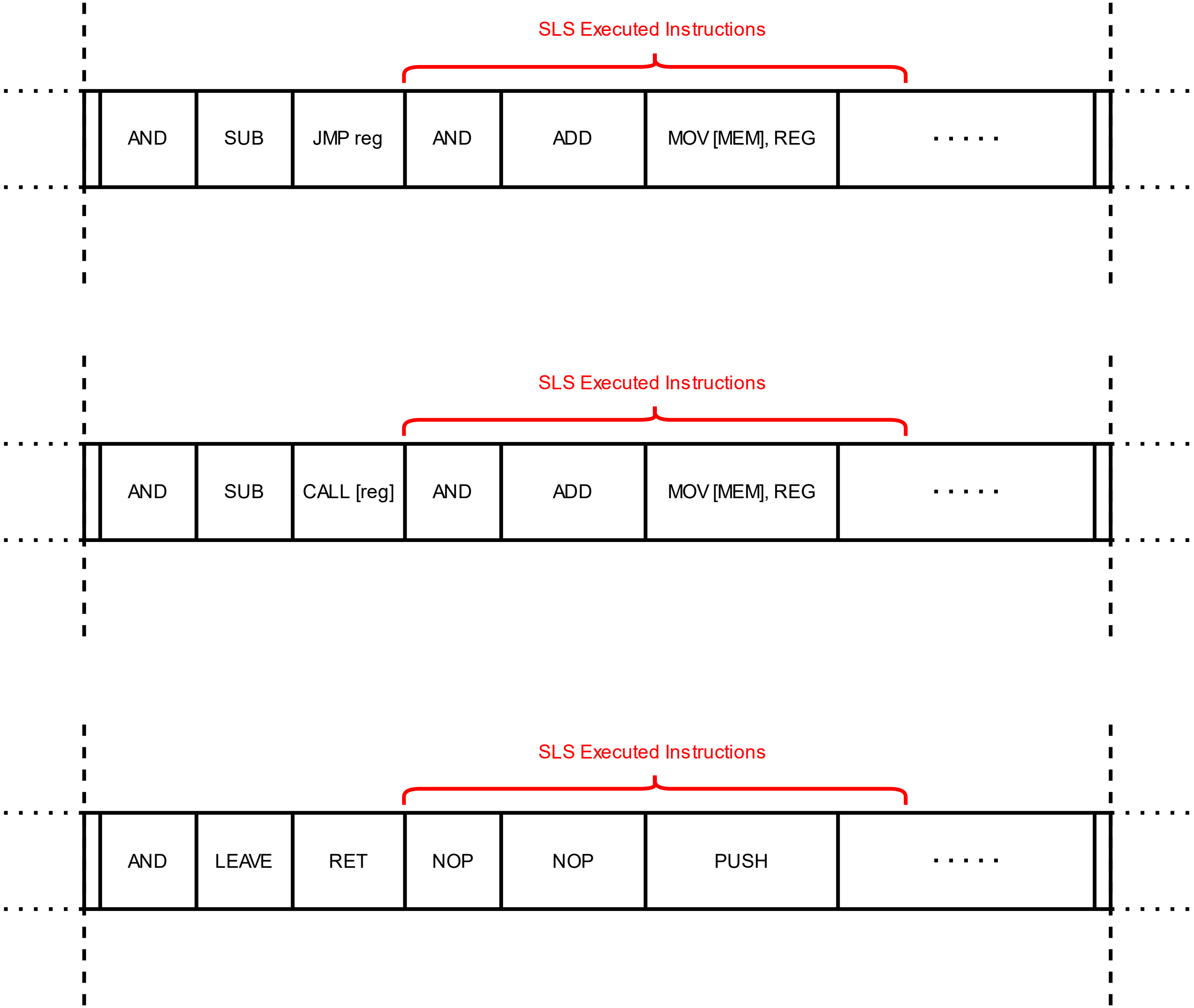
Straight-Line Speculation (SLS)

- 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



SLS Executed Instructions

AND | SUB | JMP reg | AND | ADD | MOV [MEM], REG | . . . . .

SLS Executed Instructions

AND | SUB | CALL [reg] | AND | ADD | MOV [MEM], REG | . . . . .

SLS Executed Instructions

AND | LEAVE | RET | NOP | NOP | PUSH | . . . . .

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 $address
    - CALL $address
  - Branch direction does not matter
    - Forward and backward branches suffer from the SLS
  - It is possible to trigger the 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

- SLS code example

```
; memory address 0 whose access latency allows to observe the speculative execution
0. mov CACHE_LINE_0_ADDR, %rsi

; memory address 1 whose access latency allows to observe the speculative execution
1. mov CACHE_LINE_1_ADDR, %rbx

; flush both cache lines out of cache hierarchy to get a clean state
2. clflush (%rsi)
3. clflush (%rbx)
4. mfence

5. jmp END_LABEL
; memory load to the flushed cache line; it never executes architecturally
6. mov (%rsi / %rbx), %rax
7. END_LABEL:

8. measure CACHE_LINE_0/1_ADDR access time
```

Straight-Line Speculation (SLS)

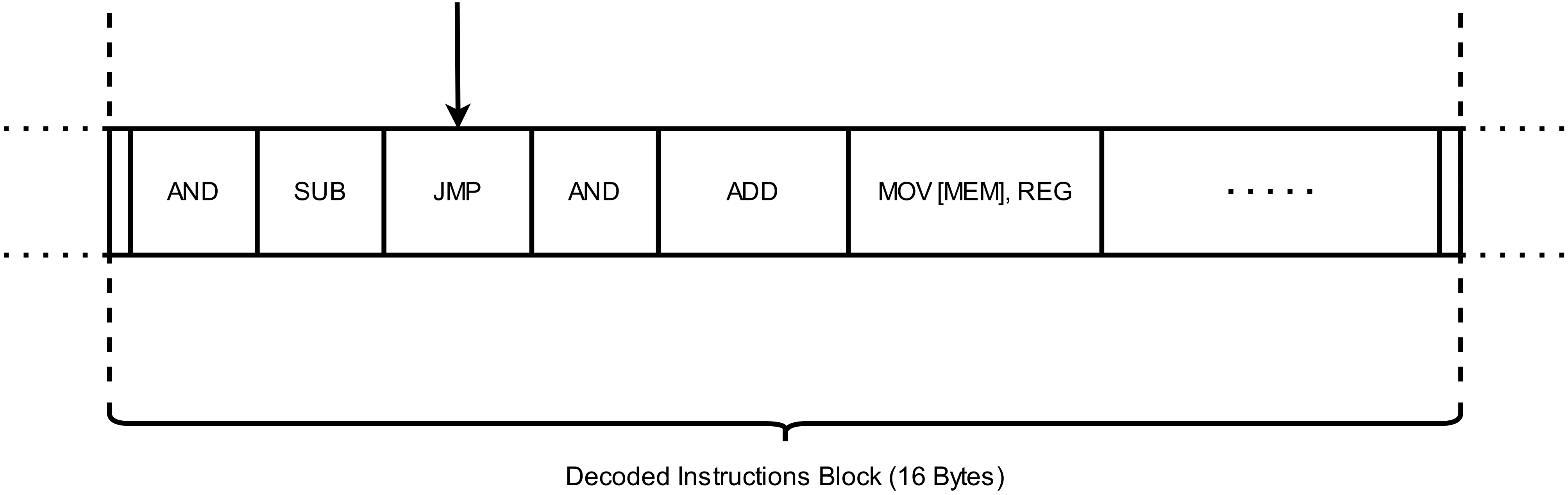
- Why would a modern CPU speculate past a direct unconditional branch?
    - After all:
        - Its target address is static!
            - And encoded as part of the instruction!
        - There is no latency involved
            - Its unconditional – no need to evaluate conditions

Straight-Line Speculation (SLS)

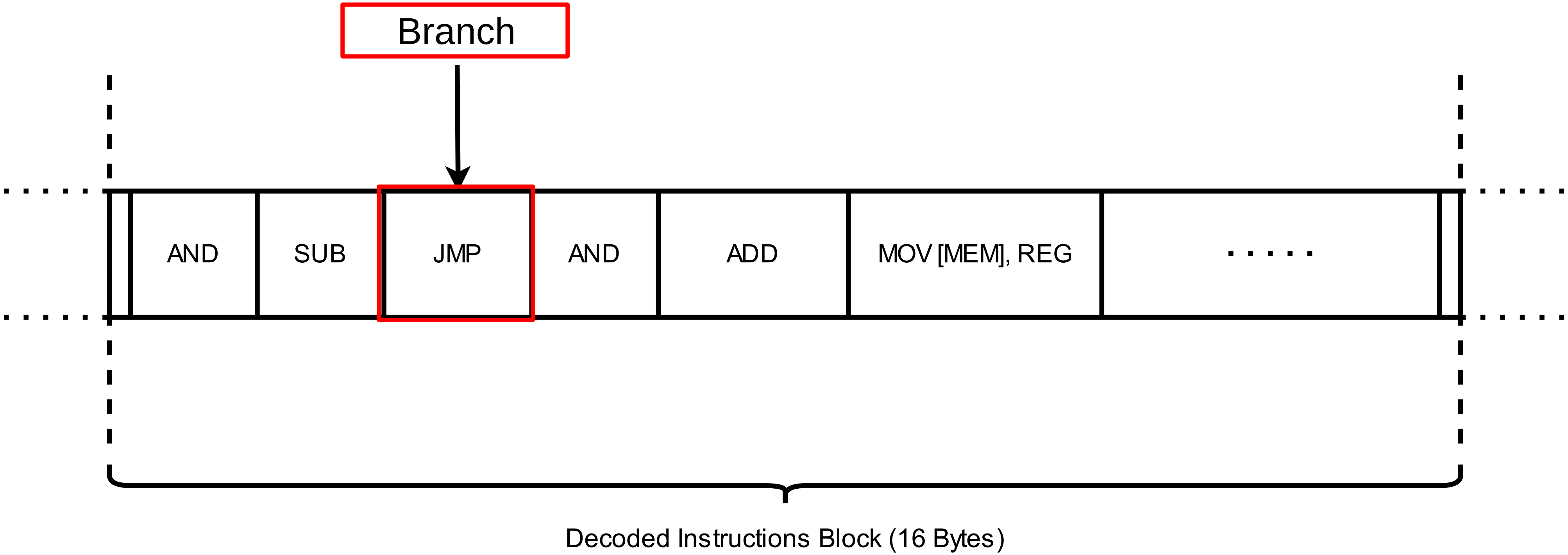
- Why would a modern CPU speculate past a direct unconditional branch?
  - After all:
    - Its target address is static!
      - And encoded as part of the instruction!
    - There is no latency involved
      - Its unconditional – no need to evaluate conditions

- Let’s see why…

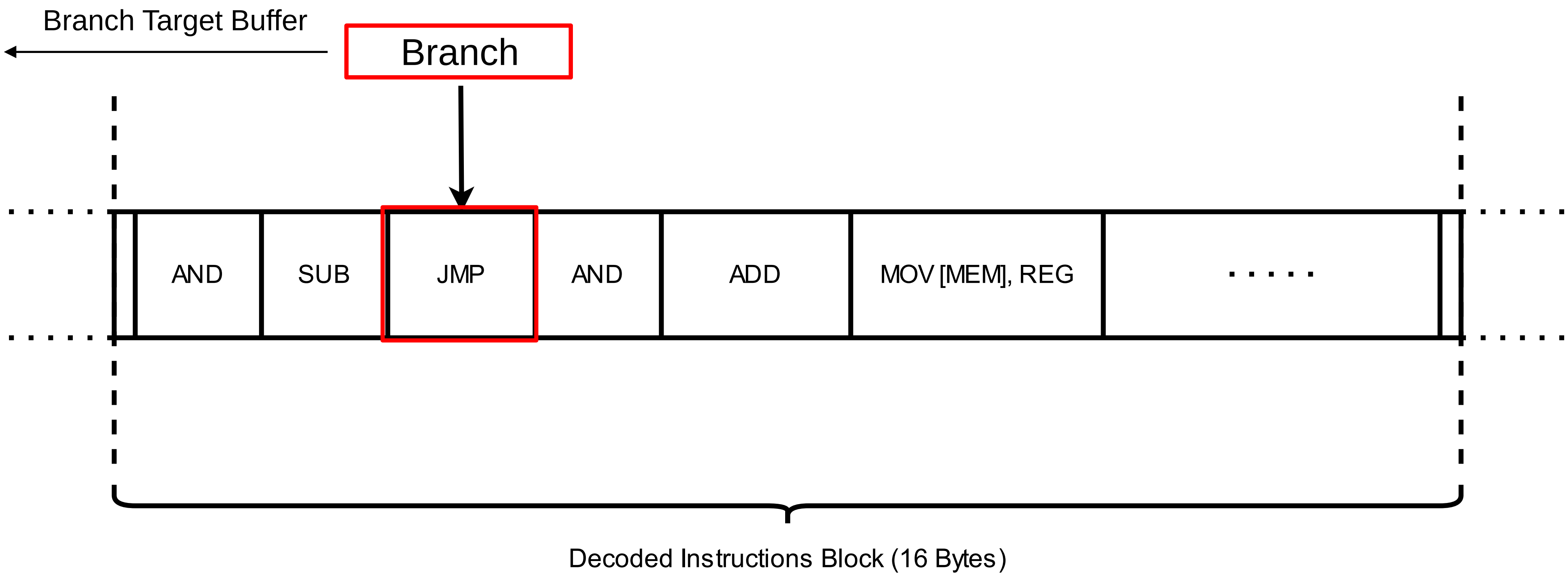
Straight-Line Speculation (SLS) - mechanics



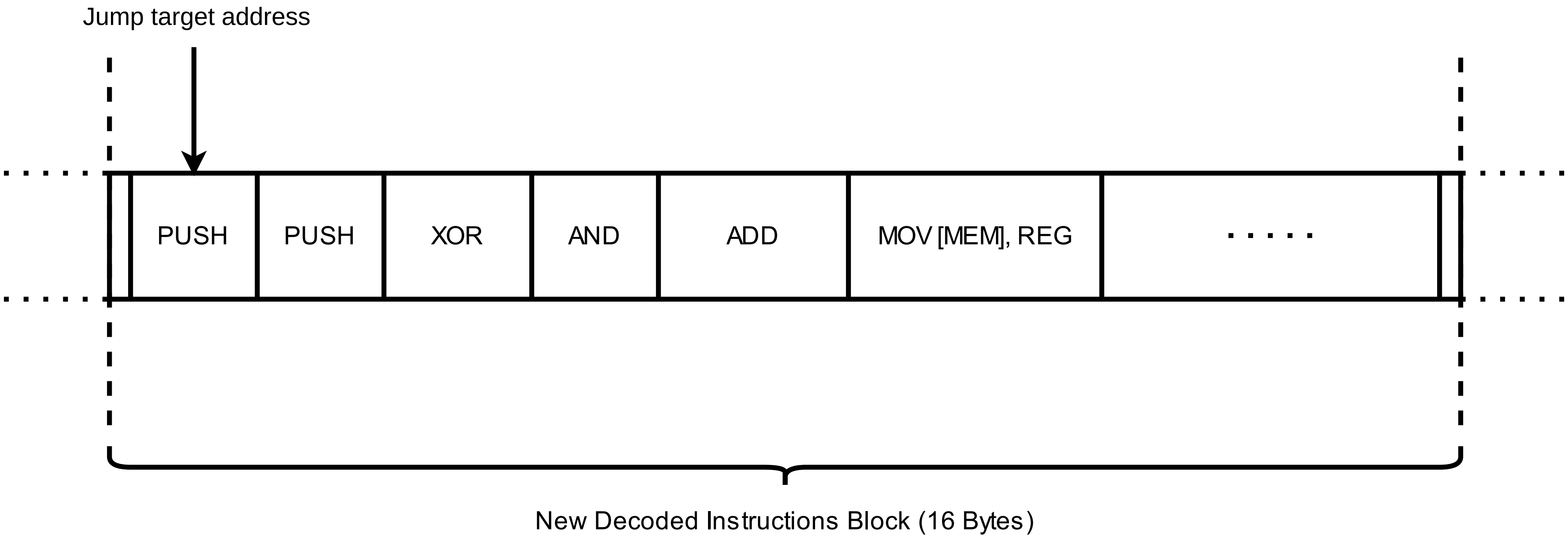
Straight-Line Speculation (SLS) - mechanics



Straight-Line Speculation (SLS) - mechanics

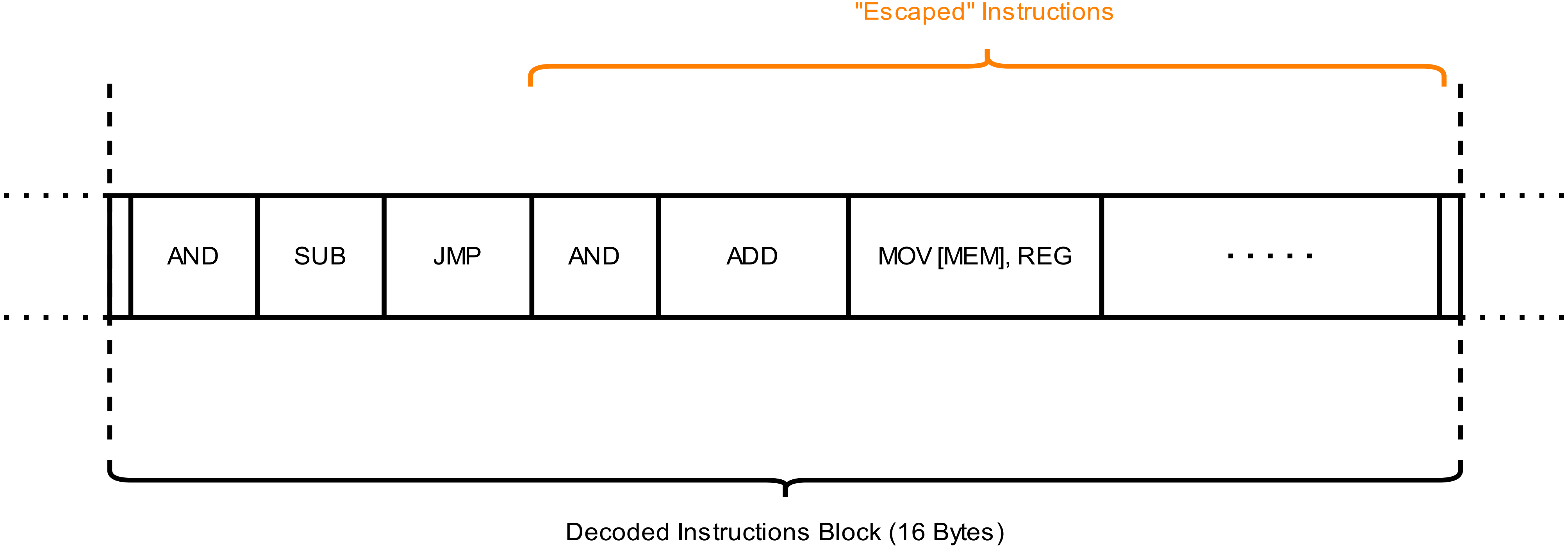


Straight-Line Speculation (SLS) - mechanics



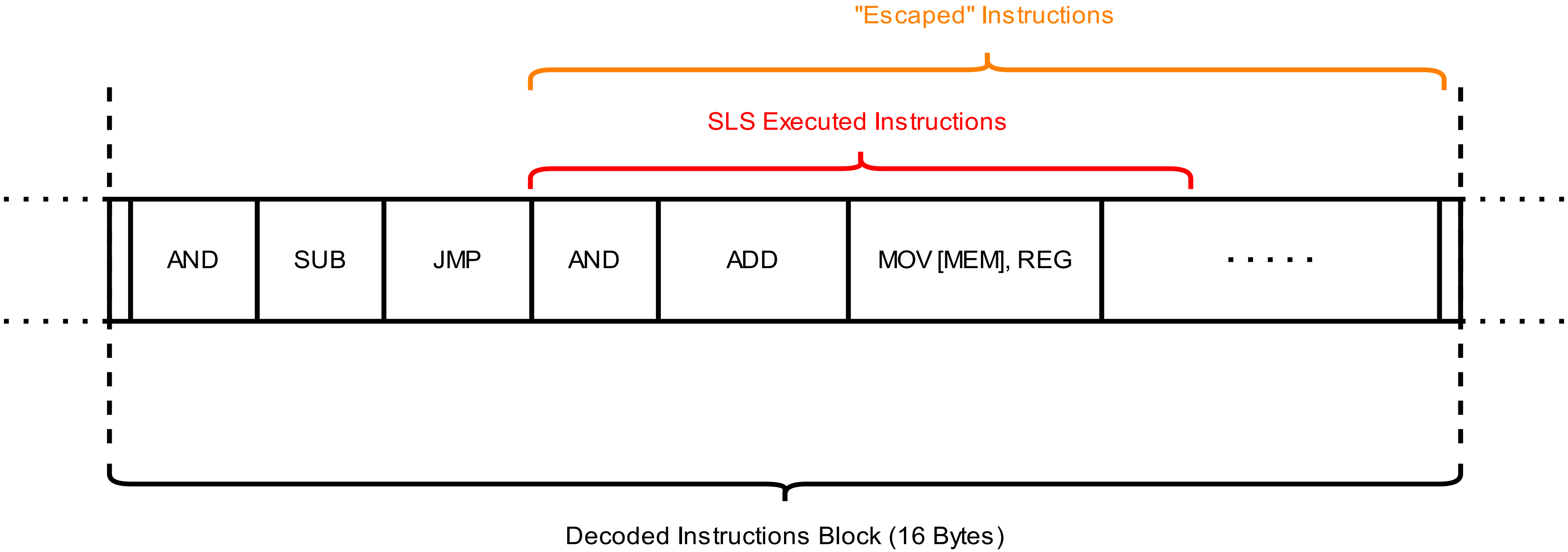
Predicted correctly

Straight-Line Speculation (SLS) - mechanics

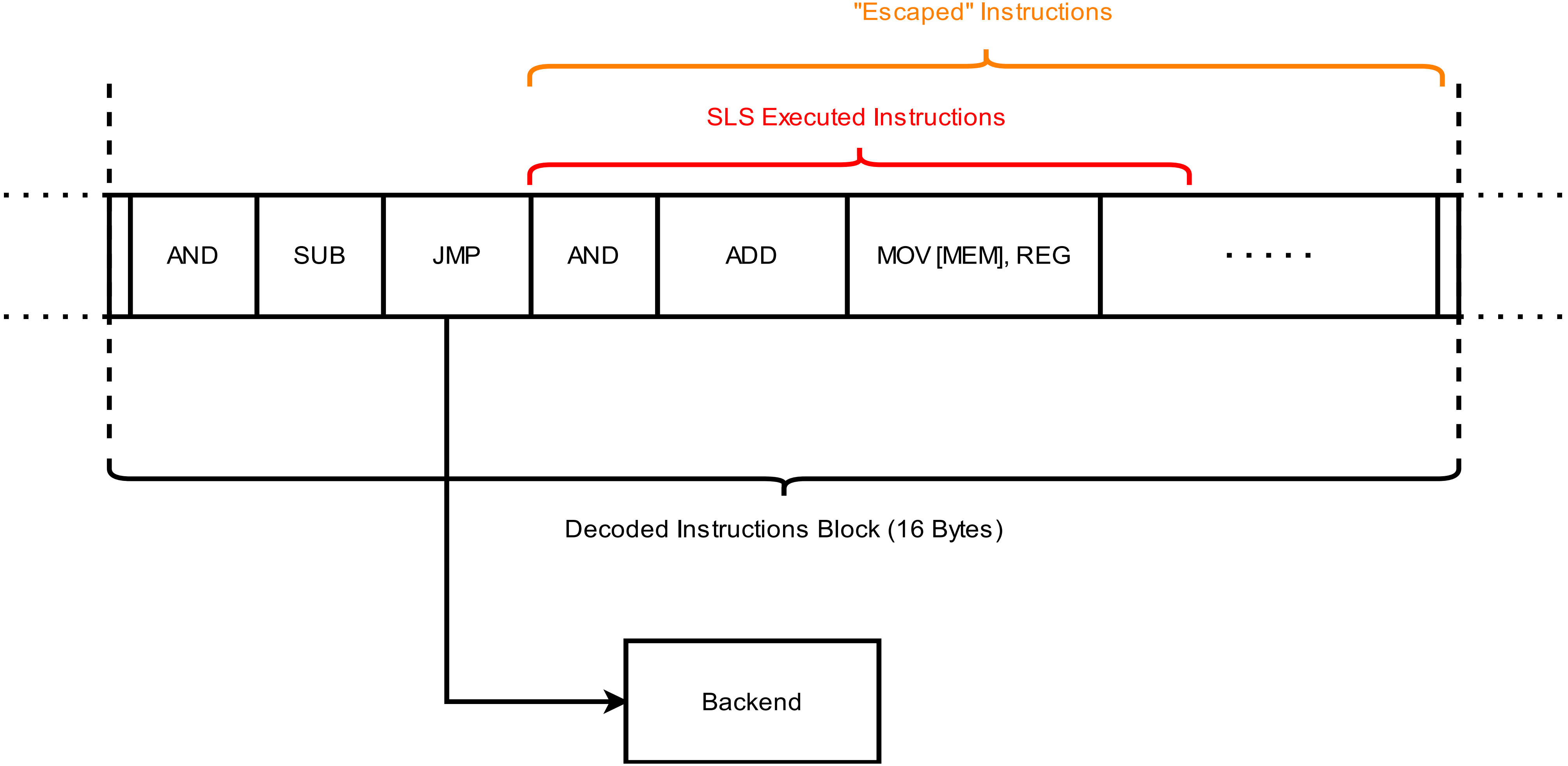


Mispredicted

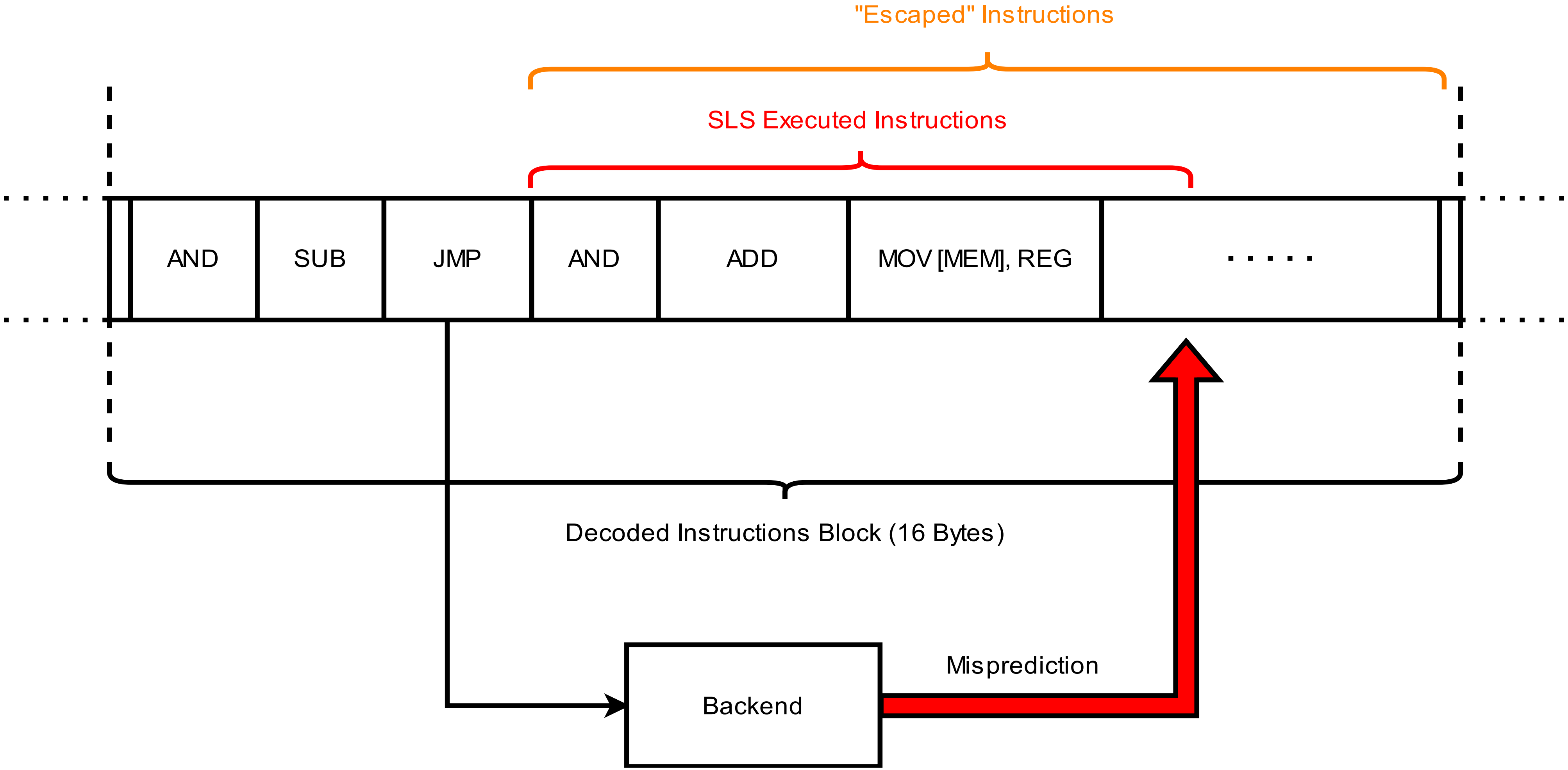
Straight-Line Speculation (SLS) - mechanics


"Escaped" Instructions
SLS Executed Instructions
AND
SUB
JMP
AND
ADD
MOV [MEM], REG
. . . . .
Decoded Instructions Block (16 Bytes)

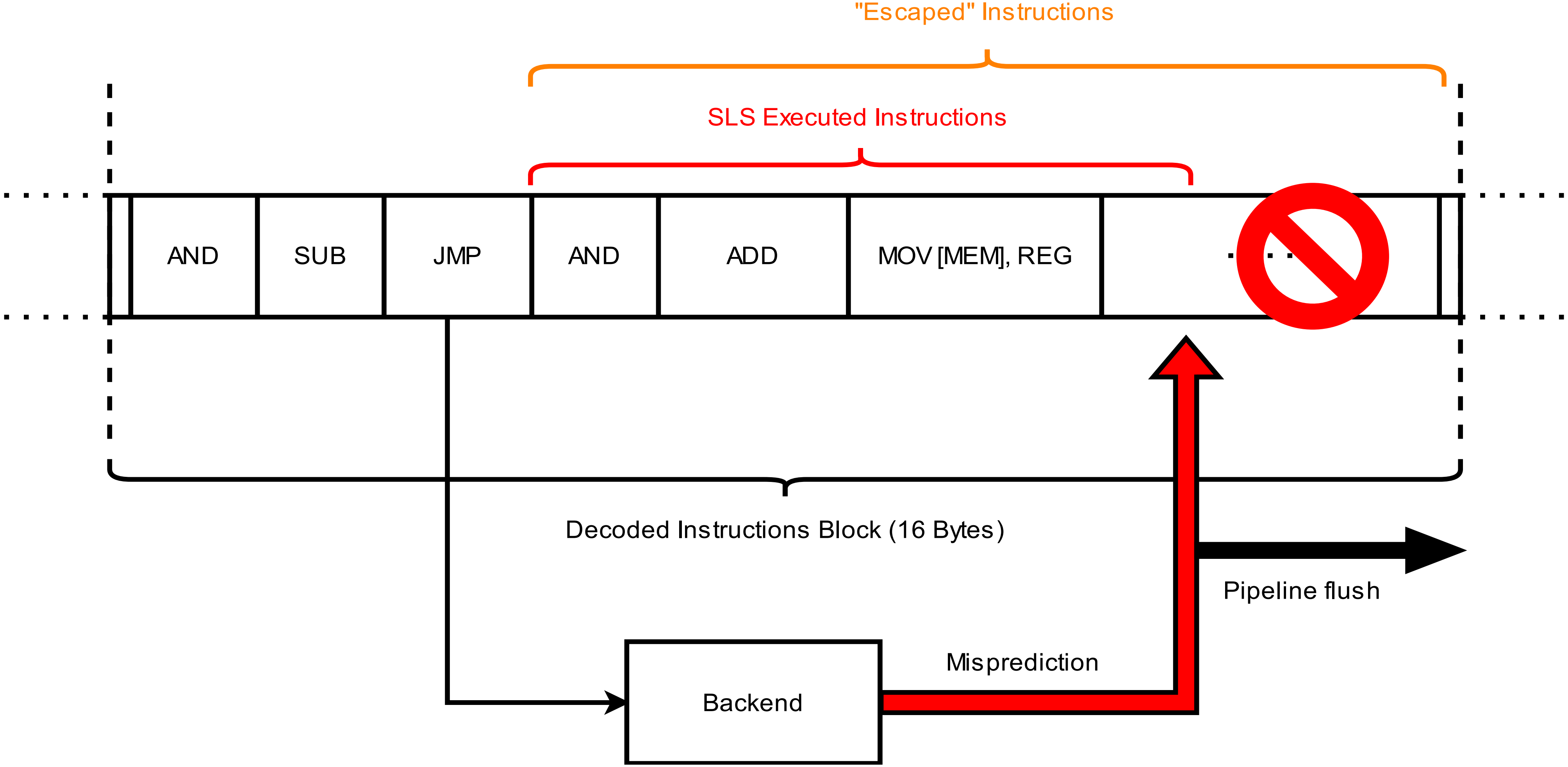
Straight-Line Speculation (SLS) - mechanics



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 the 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

```
.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 uops 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
  - Such loads do provide data to their dependent uops

- STLF requirements
  - Earlier store contains all the load’s bytes (cannot load more)
  - CPU uses address bits 11:0 to determine STLF eligibility
  - Same address space and ideally same registers, closely grouped together

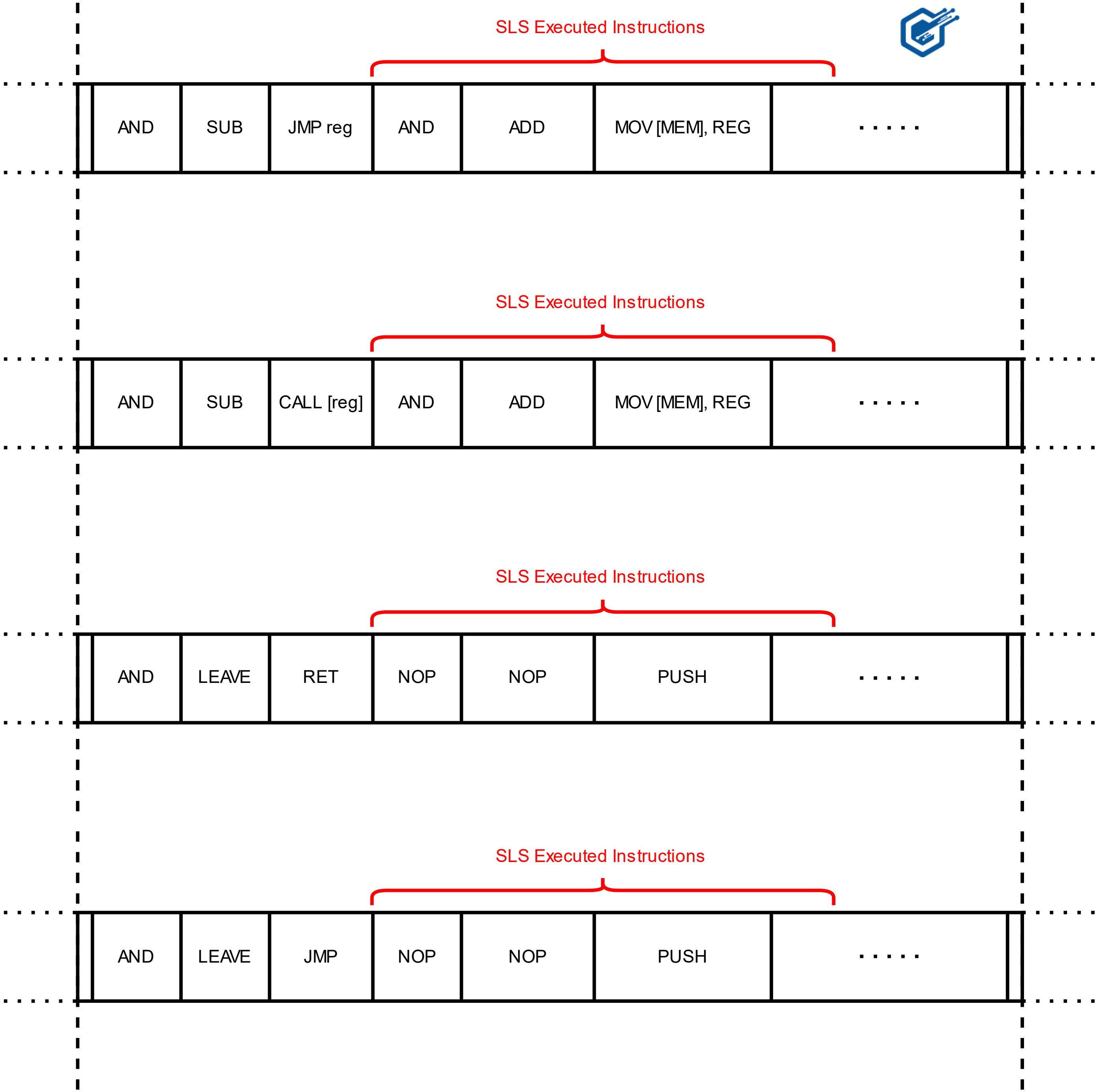
CVE-2021-26341 - Direct unconditional branch SLS

- SLS gadget 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:
```

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 $address



SLS Executed Instructions

AND | SUB | JMP reg | AND | ADD | MOV [MEM], REG | . . . . .

SLS Executed Instructions

AND | SUB | CALL [reg] | AND | ADD | MOV [MEM], REG | . . . . .

SLS Executed Instructions

AND | LEAVE | RET | NOP | NOP | PUSH | . . . . .

SLS Executed Instructions

AND | LEAVE | JMP | NOP | NOP | PUSH | . . . . .

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 $address
    - **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
- If a conditional branch is architecturally taken
  - It could be speculatively executed as not taken ➔ mispredicted
- Typical Spectre v1 situation

Spectre v1: a fall-thru speculation of conditional branches

- Spectre v1 and conditional branches relation
- A common Spectre v1 gadget
    - Out-of-bound array access
    - Speculative bypass of a bound check
    - Bound check memory access latency
- Most speculation blocking mitigation target “array-based” Spectre v1 gadgets

- But, is Spectre v1 really limited to that?

Spectre v1: a fall-thru speculation of conditional branches

- Flush BTB to trigger a fall-thru speculation for a conditional branch
  - No condition evaluation considerations necessary
  - No memory access (or any other) latency required
  - Easy to make **any** conditional branch mispredict
    - Even a trivial one
  - Speculative type confusion
    - No need for array out-of-bound

- Works also on AMD Zen3!

- Neither this nor direct unconditional branch SLS affects Intel

Spectre v1: a fall-thru speculation of conditional branches

- Gadget example

```
; memory address whose access latency allows to observe the mispredictions
0. mov $CACHE_LINE_ADDR, %rsi

; flush the cache line out of cache hierarchy to get a clean state
1. clflush (%rsi)
2. mfence

3. xor %rdi, %rdi ; set ZF=1
4. jz END_LABEL    ; if ZF==1 goto END_LABEL

; memory load to the flushed cache line; it never executes architecturally
5. mov (%rsi), %rax
6. END_LABEL:

7. measure CACHE_LINE_ADDR access time
```

Spectre v1: a fall-thru 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 uops in time

- Constructing a full Spectre v1 gadget **is** possible
- Secret data can be anywhere in memory

- Limitations
    - Shorter speculation window ➔ fewer instructions
        - More difficult to build cache oracle

SLS Mitigations

- Here 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 put a speculative execution barrier (i.e., serializing or ordering instruction) after
        - The shorter the instruction the better
        - Never gets executed architecturally

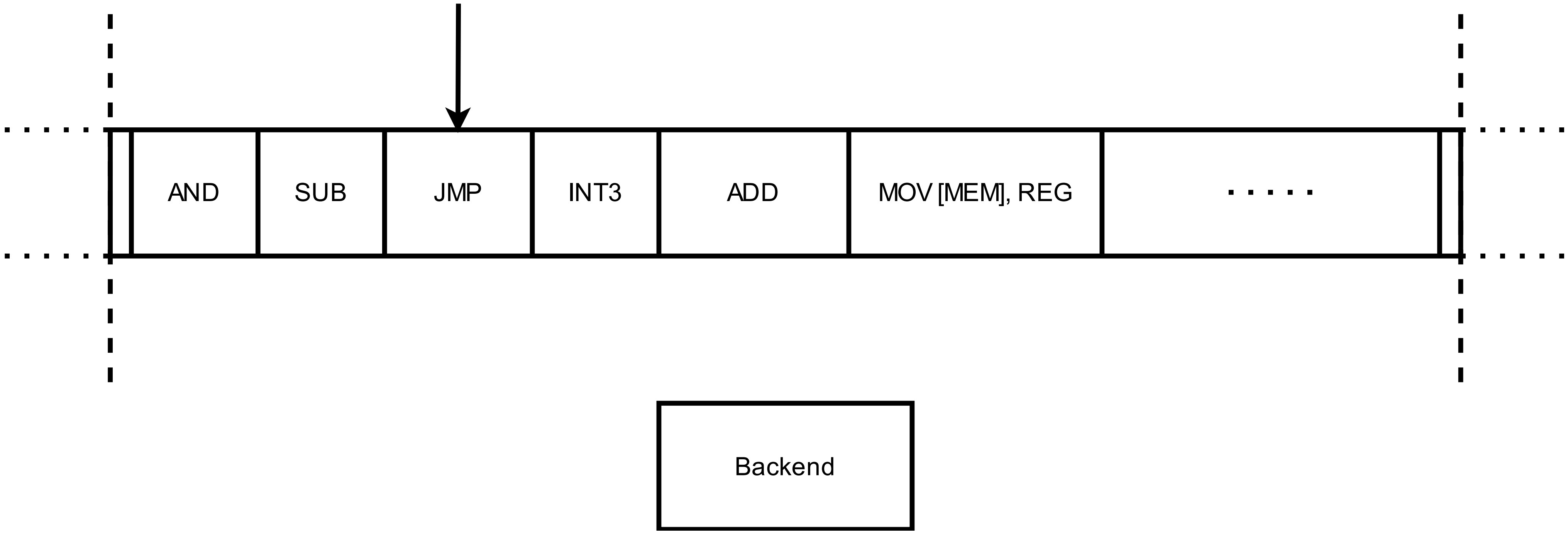
- SLS mitigation for direct and indirect unconditional **call** is not that simple
    - At some point control flow resumes execution at an instruction following the call
        - The speculative execution barrier gets executed architecturally
        - Must not have architectural “side-effects”

SLS Mitigations

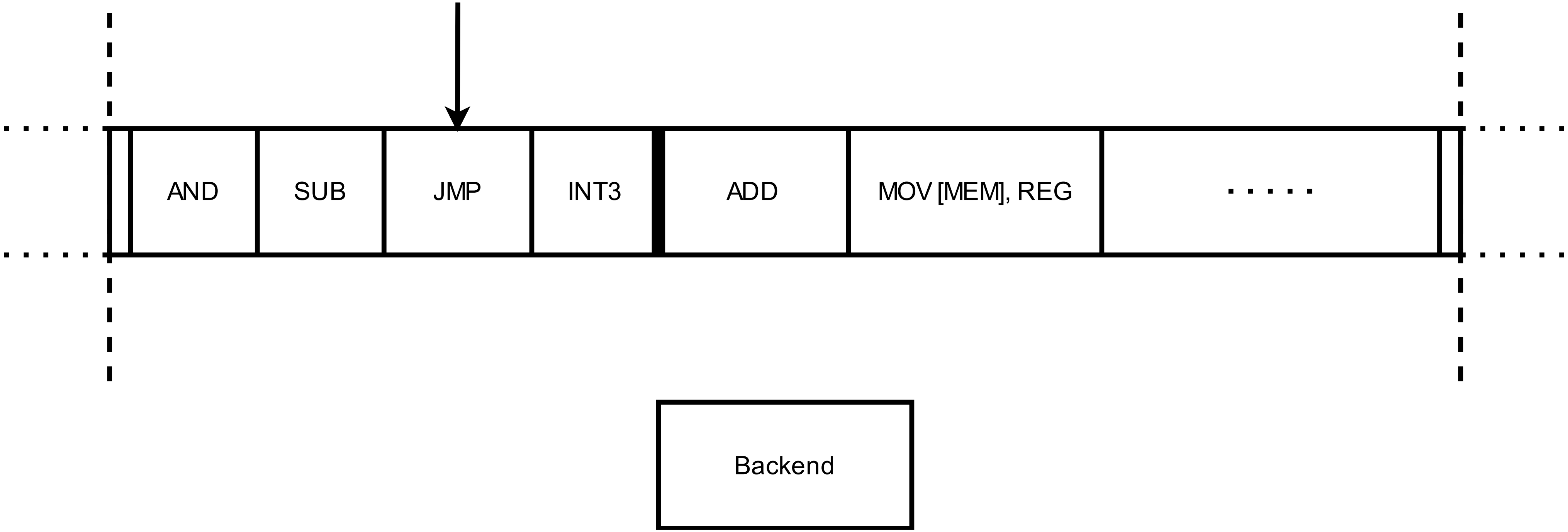
- Mitigation for
  - Direct unconditional **jump**
  - Indirect unconditional **jump**
  - Function return **RET**

  INT3 – single byte opcode (0xCC)

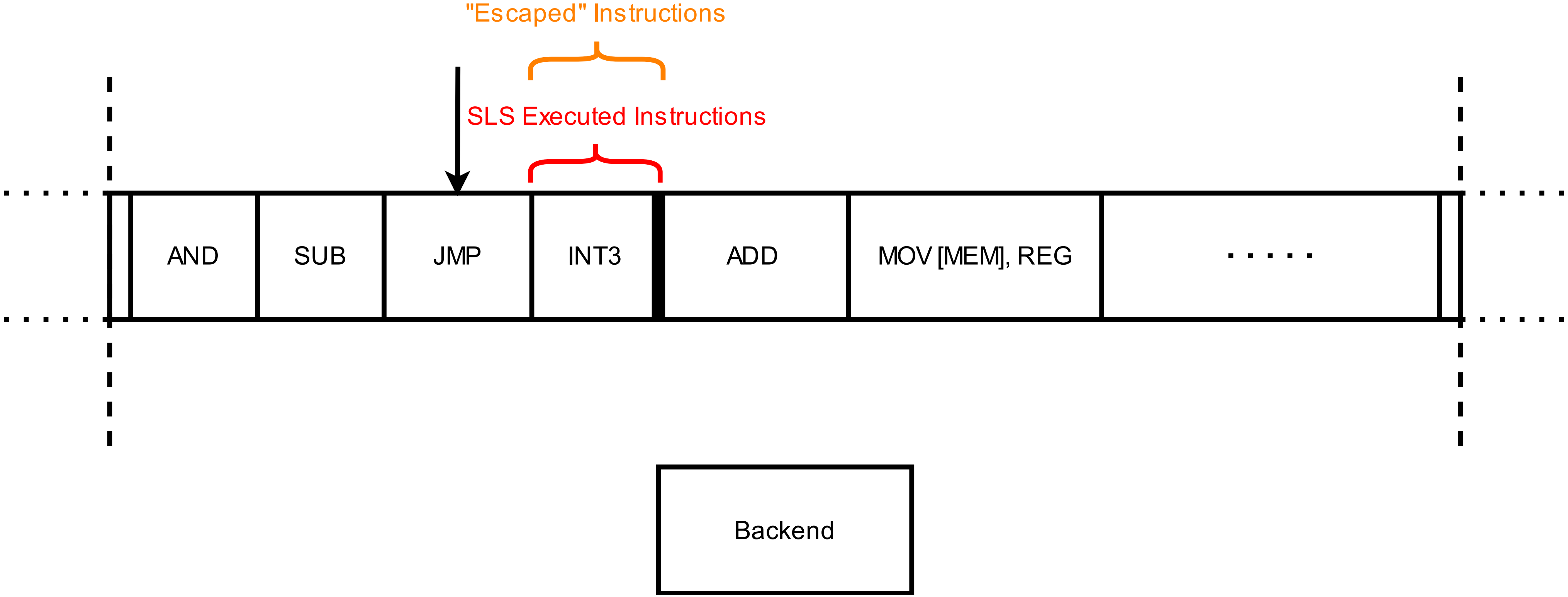
SLS Mitigations



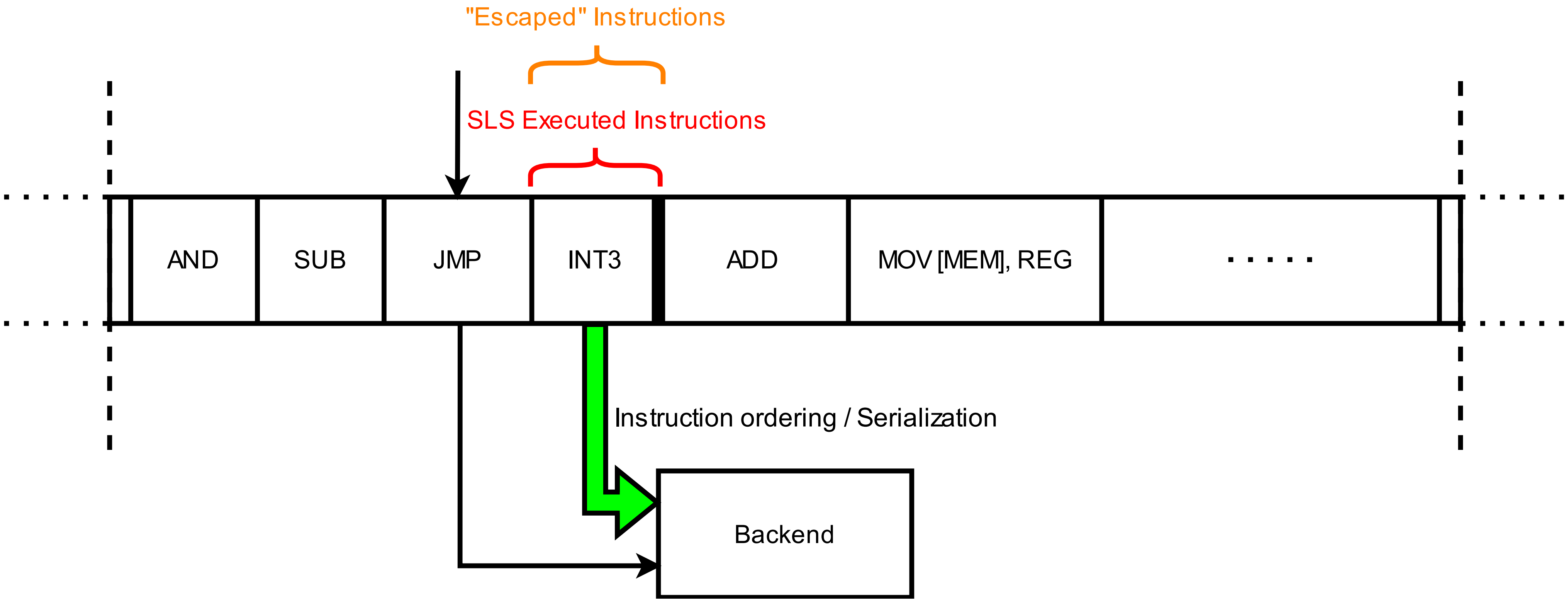
SLS Mitigations



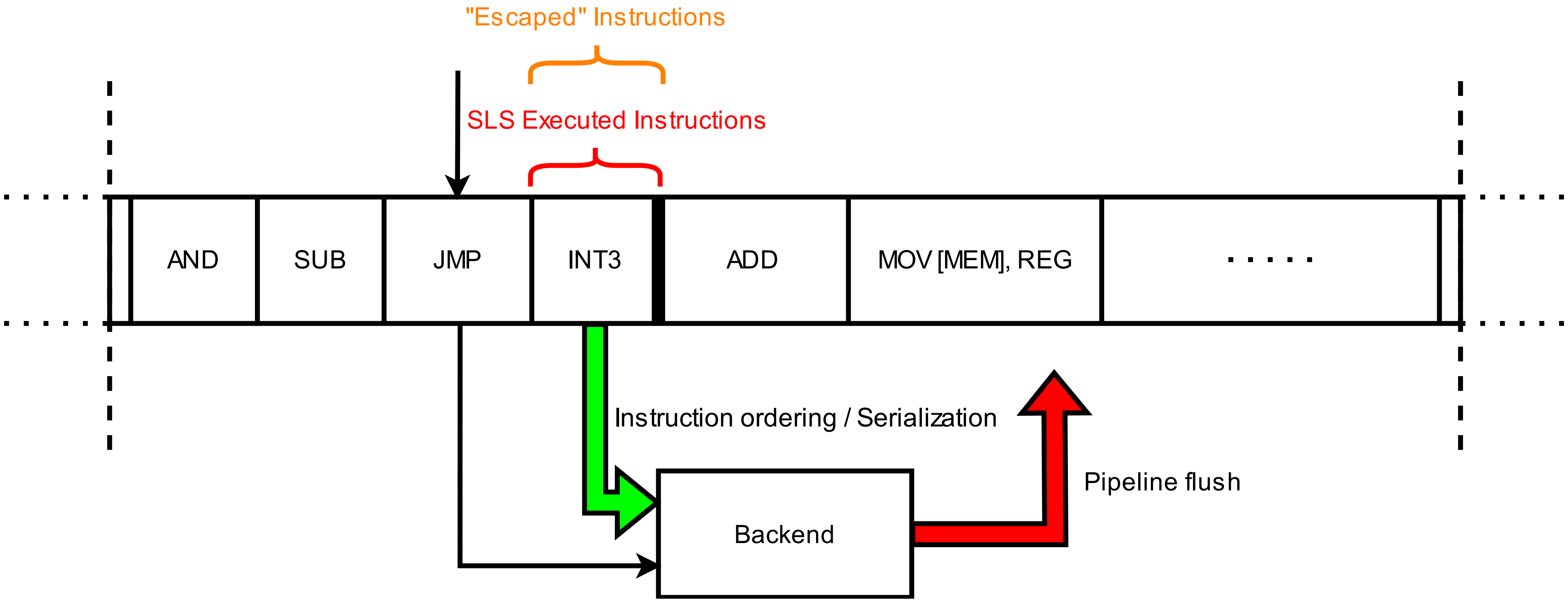
SLS Mitigations



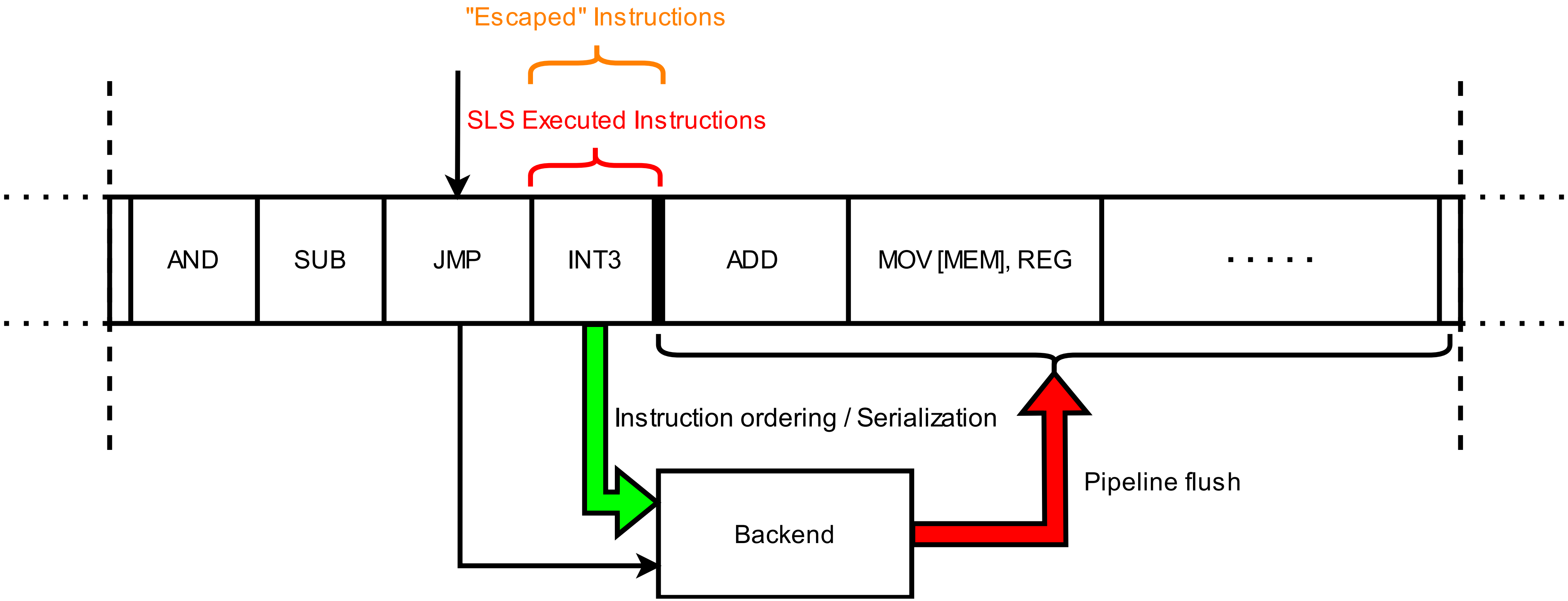
SLS Mitigations



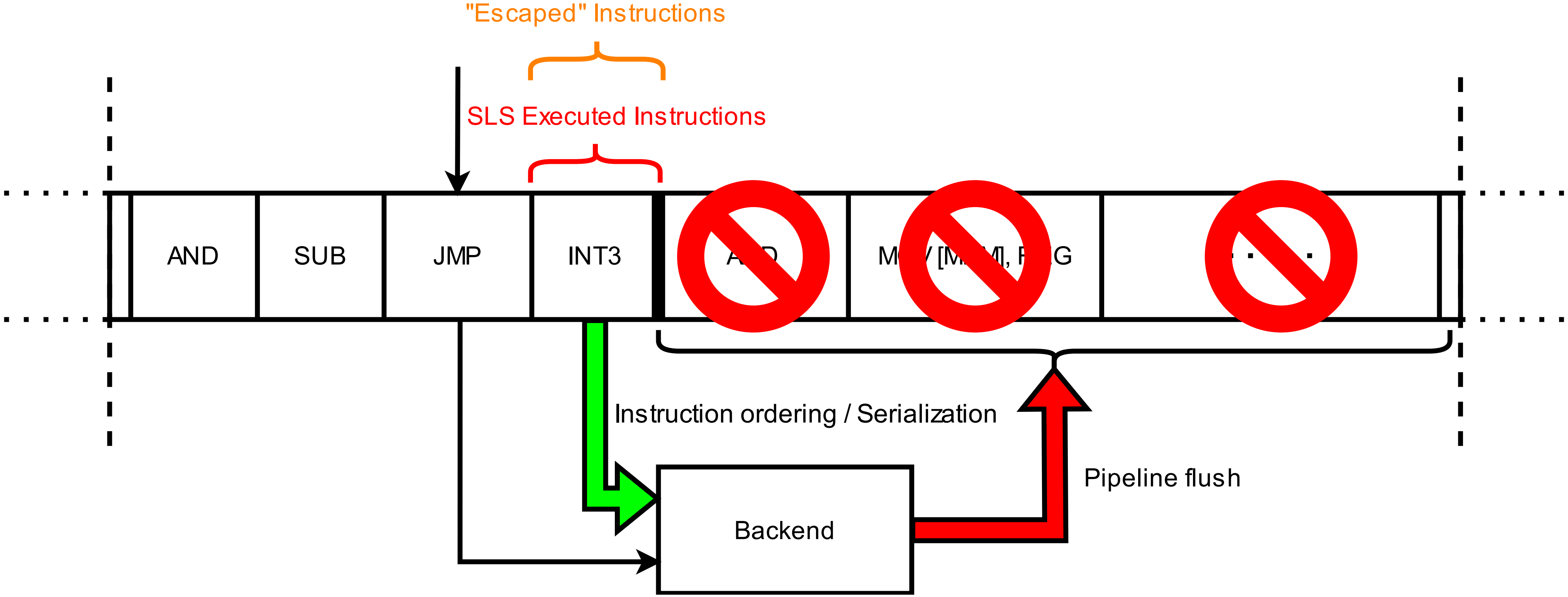
SLS Mitigations



SLS Mitigations



SLS Mitigations



SLS Mitigations

- Mitigation for
  - Direct unconditional **call**
  - Indirect unconditional **call**

  LFENCE - Not good for performance!

  XOR EAX, EAX – complicated!

SLS Mitigations

- Mitigation for
  - Direct unconditional **call**
  - Indirect unconditional **call**

  LFENCE - Not good for performance!

  XOR EAX, EAX – complicated!

- **XOR EAX, EAX**
  - Based on compiler post-call behavior assumptions
    - Callee-clobbered registers won't be used without re-write
    - Callee-preserved registers are preserved – invariant
    - Only return value register (eax) might be abused
  - Clearing return value register before the call
    - Forces eax value to 0 during SLS
    - No arbitrary content of eax

SLS Mitigations

- Mitigation for
  - Direct unconditional **call**
  - Indirect unconditional **call**

  LFENCE - Not good for performance!

  XOR EAX, EAX – complicated!

- **XOR EAX, EAX**
  - Complicated:
    - Based on compiler assumptions that might not always hold
      - Compiler implementation dependent
    - Some calling convention ABIs use eax as function input parameter
      - Fastcall / regparm(3)
    - Variadic functions may use eax as parameter
    - Small structures returned via eax + edx
    - What to do with:
      - CALL eax

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
OPEN
SOURCE
SECURITY