



This work is distributed
under the CC BY-SA license,
unless stated otherwise.

HardwareSecurity

7. Microarchitectural attacks

Ján Jančár

Summer 2026

Today

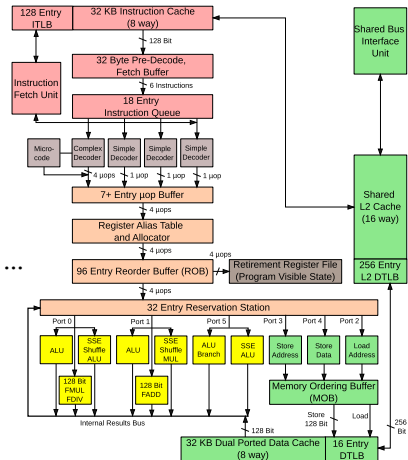
- Microarchitecture and microarchitectural state
- Microarchitectural attacks
- Defences

Microarchitecture

What the processor does to be fast

Microarchitecture

- **Architecture** (The contract)
 - Specified by the vendor (ISA)
 - Programs execute architecturally
 - Registers, instructions, memory, flags
- **Microarchitecture** (The implementation)
 - Programs should not need to care about it
 - It should be invisible
 - Caches, branch predictors, out-of-order execution, ...
 - *How the sausage gets made*
- **Microarchitectural state**
 - Runtime state of the processor's microarchitecture



Intel Core 2 Architecture

Microarchitecture

Why?

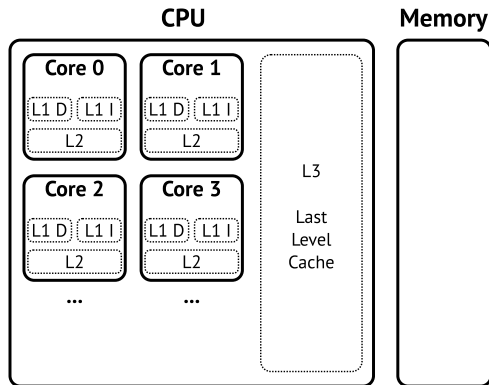
So that you can build [Minecraft in Minecraft](#)

- [Moore's law](#) slowdown
 - Hard to make transistors smaller: Physics become weird
 - Hard to fit more transistors: Heat, electricity consumption
 - ▶ Power scales non-linearly
- Optimizations
 - More cores
 - More caches
 - More parallelization
 - More complex pipelines
- Specific to the CPU
 - We will be discussing generic elements

Microarchitecture

Caches

- Last lecture
- Part of the microarchitecture
- Cache attacks are microarchitectural attacks



Microarchitecture

Instruction lifecycle

■ Fetch (IF)

- Program Counter (PC) pulls the next instruction from memory or the I-Cache

■ Decode (ID)

- Translates raw instruction bits into internal signals/control words (μ ops)

■ Execute (EX)

- The ALU performs the math, logic, or address calculation

■ Memory (MEM)

- Data is read from or written to RAM (via L1/L2/L3 cache hierarchy)

■ Writeback (WB)

- Results are committed permanently to the architectural registers

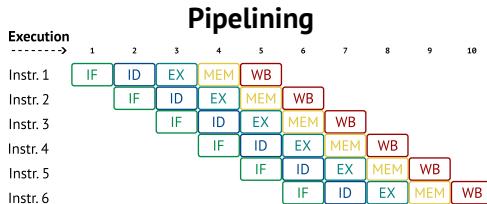
■ One clock cycle → One stage executed

■ Modern processors can have many more stages

- 10 - 30 stages
- More granular split → Smaller/faster stage → Higher clock speed

Microarchitecture

Pipelining



Microarchitecture

Pipelining

- Overlap instruction stages to increase throughput ([pipelining](#))
- Multiple instructions in-flight at once
 - Improves performance without increasing clock speed
- **Hazard**: Instruction can not execute in parallel
 - Data hazard: Data dependence between instructions
 - Control hazard: Branch
- ? What do you do when there is a branching instruction?
 - Wait (stall)
 - Always guess it is not taken (or taken) and continue, resolve the mess later
 - Remember whether it was taken previously, guess the same, resolve the mess later
- Hazard handling introduces complexity
 - **Stall**: Stop until ready
 - **Bypass**: Route data as soon as possible
 - **Reorder**: Find other instructions to execute
 - **Speculate**: Guess the dependence (branch)

A lot of work goes into keeping the pipeline and execution engine saturated.

Microarchitecture

micro-ops (μ ops)

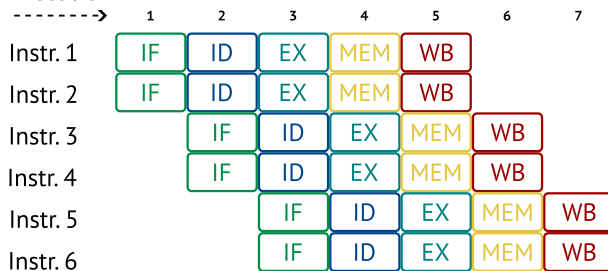
- Complex ISA instructions
 - **Split** into simpler internal operations
 - ▶ [micro-ops](#)/ μ ops
 - **Fused** into faster internal operations
- Enables flexible scheduling and execution
 - μ ops can be more regular-sized and easier to execute
 - Instructions can be kept developer-friendly
 - μ ops are what actually executes
- Architecturally, this is invisible
 - But, timing varies

Microarchitecture

Superscalar execution

- Multiple instructions executed per cycle ([superscalar](#))
- Uses multiple execution units (ALUs, load/store units)
 - **Execution unit:** hardware that performs an operation (e.g., add, multiply, memory access)
 - **Execution port:** channel to the execution unit
- Parallelism limited by dependencies

Execution

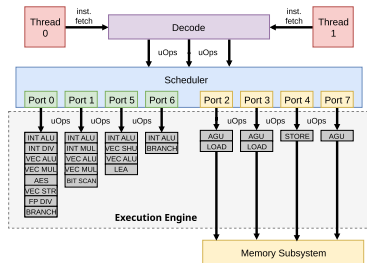


Microarchitecture

Simultaneous multithreading

■ Simultaneous multithreading

- Multiple threads (logical cores) share one physical core
 - Instructions are intermixed (even within the same cycle)
- ### ■ Different to multiprocessing
- There is no context-switch/time slicing
- ### ■ Improves utilization of execution units
- ### ■ Threads share caches and execution resources

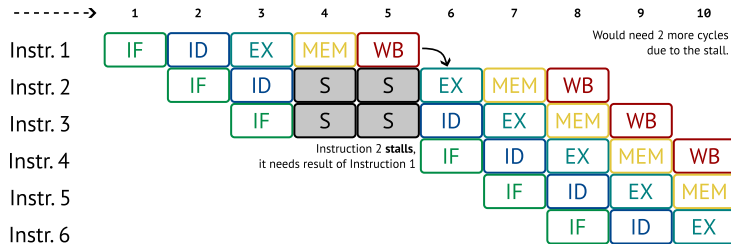


Microarchitecture

Out-of-order execution

- Instructions may not come in the best order for the CPU
 - Lots of instructions of the same type: Does not saturate all execution units
 - Dependencies: Some instructions may not be executable yet → stall

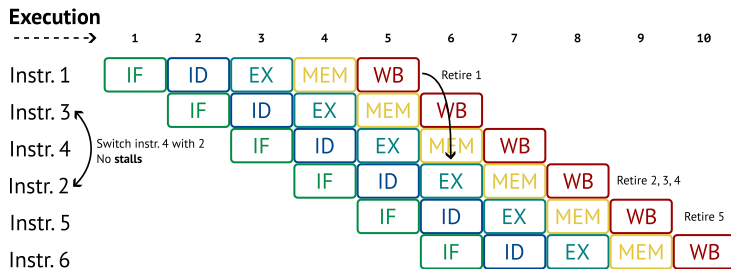
Execution



Microarchitecture

Out-of-order execution

- Instructions may not come in the best order for the CPU
 - Lots of instructions of the same type: Does not saturate all execution units
 - Dependencies: Some instructions may not be executable yet → stall
- **Out-of-order execution**
 - Execute instructions as soon as inputs are ready
 - Reorder execution to avoid stalls
- Results committed in-order (retirement)
 - **Retirement:** point where results become permanent (visible to the program)



Microarchitecture

Out-of-order execution

- Instructions may not come in the best order for the CPU
 - Lots of instructions of the same type: Does not saturate all execution units
 - Dependencies: Some instructions may not be executable yet → stall
- **Out-of-order execution**
 - Execute instructions as soon as inputs are ready
 - Reorder execution to avoid stalls
- Results committed in-order (retirement)
 - **Retirement**: point where results become permanent (visible to the program)
- ? What if you executed an instruction you should not have?
 - An older instruction actually caused a trap (div by zero/illegal memory access), or
 - You speculated wrong
 - ➡ **Squash** all instructions executed wrongly instead of retiring them
 - ▶ These are called *transient* instructions

Architecturally, this is invisible: Instructions retire in-order

Microarchitecture

Speculation

```
if (x == 5) {  
    do_this();  
} else {  
    do_that();  
}
```

```
mov eax, DWORD PTR x[rip]  
cmp eax, 5  
jne .L2  
call do_this  
jmp .L3  
.L2:  
call do_that  
.L3:  
...
```

- Stalling on branches is very inefficient
- **Speculation**
 - Proceed with execution as if branch was/was not taken
 - ✓ If you guessed correctly, hurray, retire instructions
 - 🔥 If you guessed incorrectly, burn, squash all transient instructions
- Architecturally, act as if nothing happened
- **Transient execution:** Any execution like this, that "never happened"

Microarchitecture

Branch prediction

- Do not just guess branches, learn patterns and **predict their outcome**
- **Branch History Buffer (BHB)**
 - Predicts **whether** conditional branches are taken or not
 - Global
 - Two level with a **Pattern History Table (PHT)**
- **Branch Target Buffer (BTB)**
 - Predicts addresses of **where** indirect branches jump to
- **Return Stack Buffer (RSB)**
 - Predicts addresses of **where** functions **return** to (**ret**)
 - Buffer containing return addresses of last N **call** instructions

Microarchitecture

Pre-fetching

- Predict future memory accesses and load data into cache early ([prefetching](#))
- Reduces memory latency if prediction is correct
- Based on access patterns (array, stride)

```
char arr[bound];  
for (int i = 0; i < bound; i++) {  
    out[i] = arr[i];  
}
```

```
char arr[bound*stride];  
for (int i = 0; i < bound; i++) {  
    out[i] = arr[stride*i];  
}
```

- Also, *data memory-dependent* prefetchers (pointer chasing)

```
char *arr[bound];  
for (int i = 0; i < bound; i++) {  
    char *a = arr[i];  
    out[i] = *a;  
}
```

Covert channel

- **Side channel:** *victim* leaks to *attacker* by accident
- **Covert channel:** *attacker* leaks to themselves across some security boundary

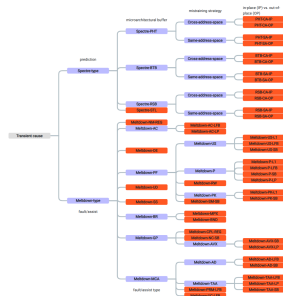
Microarchitectural attacks

What happens when you see the microarchitecture

Transient execution attacks

- **Aim:** Leak data from a different security domain
- **Scenarios**
 - User to Kernel
 - User to User
 - Guest to Host
 - Guest to Guest
 - Cross-thread
- **Idea:** Exploit *transient execution* somehow
 - Make the processor execute some code transiently
 - It will roll-back the architectural effects when it detects it should not have executed
 - It will **not** roll-back the microarchitectural effects
 - Observe the microarchitectural effects somehow
 - ...
 - Profit
- Spectre, Meltdown, Foreshadow, Downfall, Fallout, RIDL, ZombieLoad, LVI, BHI, Retbleed, Zenbleed, ...

transient.fail



Spectre

- **Origin:** Transient execution due to speculation
- Bypasses **software-defined** properties
 - Bounds checks, function call-return pairs, etc.
- Many variants
 - V1: Bounds Check Bypass (Spectre-PHT)
 - V2: Branch Target Injection (Spectre-BTB)
 - Return Address Injection (Spectre-RSB)
 - V4: Speculative Store Bypass (Spectre-STL)
 - ...

Spectre V3 is actually Meltdown.

Microarchitectural attacks

Spectre V1: Bounds Check Bypass

```
int array[array_len];
int victim(uint64_t index) {
    if (index >= array_len)    // Bounds check
        return -1;
    return array[index];
}
int wrapper(uint64_t index) {
    for (int i = 0; i < 128; i++) // Clear BHB
        ;
    return victim(index);        // Run victim
}
int spectre(uint64_t index) {
    wrapper(0);                 // Mistrain branch predictor
    wrapper(0);

    _mm_clflush(&array_len);    // Flush array_len
    _mm_mfence();
    int rv = wrapper(index);     // Access out of bounds
    cc_transmit(rv);             // Leak value to yourself, transiently
                                // via some covert_channel

    return cc_receive();
}
```

Mistrain the conditional branch predictor of a victim (BHB + PHT), then access out of bounds and leak the value transiently to yourself (covert channel)

Microarchitectural attacks

Spectre V2: Branch Target Injection

- **Idea:** Mistrain the indirect branch predictor of a victim (BTB) to speculatively execute code anywhere in its memory
 - Use “*gadgets*” from victim’s memory to leak values to yourself
- Achieves speculative code execution

Microarchitectural attacks

Spectre-RSB: Return Address Injection

- **Idea:** Mistrain the return address predictor of a victim (RSB) to speculatively return to code anywhere in memory
- Like Spectre-BTB, but on **ret**
- Achieves speculative code execution

Meltdown

- **Origin:** Transient execution due to faulting instruction
 - Disallowed memory access (i.e., segfault)
- Bypasses **hardware-defined** security properties
 - Memory access control
- Many variants (Meltdown-like attacks)
 - Foreshadow, RIDL, ZombieLoad, Fallout, ...

Microarchitectural attacks

Meltdown

- **Idea:** Just access disallowed memory directly
- ? What happens with speculative execution?
 - Architecturally, it will not work
 - Transiently, it will (for a bit)

1 Setup probe_array for a cache covert channel (Flush+Reload)

2 Access and leak kernel memory transiently

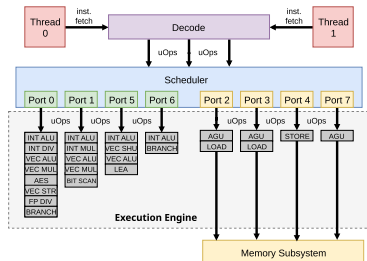
```
uint8_t secret = *kernel_address;           // Access kernel memory, will fail
uint8_t dummy = probe_array[secret*64];     // Transiently leak value
```

3 Observe access in cache covert channel

Microarchitectural attacks

Port contention

⚠ Not transient execution



- Port Contention for Fun and Profit
- **Origin:** Separate threads share the same physical resources (execution units & ports) in SMT
- **Idea:** Execute instructions on the same physical core as the victim (SMT)
 - Then, execute instructions targetting a specific execution port
 - Measure their timing → observe victim's usage of that port

Microarchitectural attacks

Augury & GoFetch

⚠ Not transient execution



GoFetch

- Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest
- GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers
- **Origin:** Data memory-dependent prefetcher fetches *pointer-looking* values into the cache
- **Idea:** Adaptively craft inputs to cryptographic algorithm that look like pointers, based on some secret key bits
 - Then, detect whether value was prefetched (Probe)

Defences

Mitigating what is possible

- **Just** disable speculation, out-of-order execution, SMT, DMP, ...
- **Reality**
 - Modern processors are fast **because of** these features
 - Performance slow down would be huge: years of setback

Can have our cake and eat it too?

Defences

Who should fix these?

■ CPU manufacturers

- Have the strongest capabilities
- Yet, changing hardware takes time
- They can provide [mitigations](#)
- They do not know all the ways their hardware is used

■ Compiler developers

- Can introduce some hardening options ([\[1\]](#), [\[2\]](#), [\[3\]](#))
- They also do not know all the ways their compiler is used

■ Kernel developers

- Can introduce hardening options
- Their systems may be used in different settings

■ Application developers

- Knows how their app works
- Yet, has the least capabilities in fixing vulnerabilities

Defences

Some mitigations (1)

■ **lfence (Load Fence)**

- Waits until all prior instructions are done (serializing)
- Put it in every branch
- **Problem:** Very slow

■ **Speculative Load Hardening (SLH)**

- Mask the loaded value using a mask that is only set in the correct branch
- Mis-speculation has all zero mask, clears value
- **Problem:** Not enough

```
int victim(uint32_t index) {  
    if (index >= array_len) {  
        return -1;  
    }  
    return array[index];  
}
```

```
int victim(uint32_t index) {  
    uint32_t mask = 0xFFFFFFFF;  
    if (index >= array_len) {  
        return -1;  
    } else {  
        mask = (index >= array_len) ? 0 : 0xFFFFFFFF;  
    }  
    index &= mask;  
    return array[index];  
}
```

Defences

Some mitigations (2)

- **Retpoline**
 - Convert indirect branches into returns
 - Thus, no BTB is involved, only RSB, which can be controlled
 - Speculation is trapped inside an infinite loop until squashed
 - **Problem:** On some processors, RSB falls back to BTB when it underflows
- **Indirect Branch Restricted Speculation (IBRS)**
 - Behavior of unprivileged code will not influence speculation in privileged code
 - Kernel [entry](#)/exit
- **Indirect Branch Predictor Barrier (IBPB)**
 - Speculation barrier, used during [context switches](#)
- **Single Thread Indirect Branch Predictors (STIBP)**
 - Isolate branch predictors during SMT

Defences

Testing your machine

■ Linux

- Look into `/proc/cpuinfo`: bugs entry
- Read files in `/sys/devices/system/cpu/vulnerabilities/`: mitigations
- Test the attacks from the [transientfail](#) repository
- [Disable](#) the mitigations and see what happens

■ Windows

- Use the [SpeculationControl](#) script

■ MacOS

- Intel processors: `sysctl machdep.cpu.leaf7_features` (some mitigations)
- Apple silicon: Not mitigated

Takeaways

- How modern processors achieve their performance via complicated microarchitecture
- What microarchitectural attacks are
- How microarchitectural attacks work
- How to defend against them

Recommended reading

- For **this** lecture
 - Read [A Systematic Evaluation of Transient Execution Attacks and Defenses](#)
 - ▶ A great overview of transient execution attacks
- For the **next** lecture
 - Read [Differential Power Analysis](#)
 - ▶ The original SPA/DPA paper by Kocher, Jaffe, Jun

References

■ Microarchitecture

- [The microarchitecture of Intel, AMD, and VIA CPUs](#)
- [MIT 6.191 lecture on Modern Processor Architecture](#)
- [Explanatory video on microarchitecture by Intel](#)

■ Transient execution attacks

- [MIT 6.S983 lecture on Transient Execution Attacks](#)
- [Meltdown: Reading Kernel Memory from User Space](#)
- [Spectre Attacks: Exploiting Speculative Execution](#)
- [Overview of CPU vulnerabilities and their mitigations in Linux](#)
- [Overview of some software mitigation techniques from AMD](#)
- [A high level overview of Spectre and Meltdown \(+ variants\)](#)