TODO

TODO

Bachelor Thesis

by

TODO

in fulfillment of requirements for degree

Bachelor of Science (B.Sc.)

submitted to

Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik IV
Arbeitsgruppe für IT-Sicherheit

in degree course

Computer Science (M.Sc.)

First Supervisor:	TODO University of Bonn
Second Supervisor:	TODO TODO
Sponsor:	TODO University of Bonn

TODO

Abstract

TODO

Contents

1 Introduction

As an advancement over older processors, modern Intel CPUs implement a number of optimization techniques that increase their efficiency. One of which is the concept of out-of-order execution, which takes advantage of the mutual independence of instructions that would normally be executed sequentially. A second optimization technique is called speculative execution and involves the prediction of whether or not a given branch is taken. With either technique, the CPU might encounter cases where the current CPU state must be rolled back to a previous one to ensure correct execution. For out-of-order execution this happens when an instruction raises an exception (e.g. accessing inaccessible memory). For speculative execution, this happens when a branch is mispredicted. A rollback causes some instructions that are currently being executed (in-flight) to continue execution for a short amount of time.

Even though rollbacks are meant to make sure in-flight instructions do not cause any lasting side effects on the microarchitectural state of the CPU, it was discovered that they can affect the contents of caches and other buffers. The disclosure of both Spectre and Meltdown in early 2018 introduced a whole family of vulnerabilities that take advantage of both out-of-order and speculative execution to leak secrets over the processor's caches, or temporarily alter the program flow of other processes. And while the performance losses introduced by software and hardware mitigations are measurable, neither vulnerability can be exploited freely on a fully patched system. As a result, however, the process of trying to exploit one of the vulnerabilities for the sake of learning how they work in detail can be challenging. Apart from the software, which may be obtained by installing an older version of an operating system that does not implement any mitigations, one must also make sure their CPU is affected by the vulnerabilities and has not yet received any relevant microcode updates from Intel. Often times, this means that a user's personal computer does not meet these requirements.

We design and implement a graphical CPU emulator that supports single-step out-of-order and speculative execution and is vulnerable to select variants of Spectre and Meltdown. While it is a simplification in comparison to real hardware, the emulator allows its users to gain a better understanding of how exactly the two vulnerabilities work and can be exploited. Furthermore, the user may experiment with (ineffective) mitigations or implement their own microcode-programs that are executed once rollbacks are completed. We supply example programs that can be run by the emulator and serve both as an entry point for the user as well as the basis of our evaluation.

Firstly, chapter 2 briefly gives an overview of relevant components of vulnerable Intel CPUs, presents the concepts of out-of-order and speculative execution in greater detail, and introduces both Meltdown and Spectre to which the resulting emulator is designed to be vulnerable. Secondly, chapter 3 further describes the target audience of the emulator and to which variants of the vulnerabilities the emulator is vulnerable, while chapter 4 documents the implementation of the emulator by describing each main component and explaining how rollbacks are implemented. Additionally, it contains an overview of the set of ISA instructions available to the user. Furthermore, chapter 5 explores the graphical user interface by defining the goals its design is supposed to accomplish and documenting design choices and important features. Chapter 6 provides a demonstration of the emulator, which includes example programs, and determines how effective the implemented mitigations are. Finally, chapter 7 summarizes the other chapters, briefly reflects on how valuable the emulator might be to our target audience, and gives ideas for future improvements.

2 Background

This chapter briefly covers the theoretical background needed to use the presented emulator and continue its development. The reader is assumed to have an understanding of elementary CPU concepts, such as pipelining and caching.

2.1 CPU

A CPU consists of a frontend, an execution engine, and a memory subsystem. As per Intel's Skylake architecture [Wik], the frontend fetches the instructions, maintains a queue of instructions that are to be executed, and decodes them into simpler microinstructions, which are then communicated to the execution engine. Additionally, it is responsible for the branch prediction (see sec. 2.3). [Gru20]

The execution engine consists of multiple sets of execution units, each set being responsible for a specific type of microinstruction, such as loads, stores, or arithmetic. Further, the scheduler allows the execution units to work on independent instructions in parallel, while the reorder buffer makes sure that instructions retire in the correct order. A common data bus (CDB) connects the reorder buffer, scheduler, and execution units. Its purpose is further described in sec. 2.2. [Gru20]

Lastly, the memory subsystem handles all memory accesses of the execution units by maintaining caches and ensuring data is fetched from lower level caches or DRAM if needed [Gru20]. Most importantly, requests to data that is present in the caches is served faster than if it were not.

A visualization of the aforementioned components can be found in fig. 1.

2.2 Out-of-order execution

As the name implies, out-of-order execution refers to the idea of executing instructions in a different order than the one in which they are given [Gru20]. With multiple execution units that run in parallel (as described in sec. 2.1), CPUs can take advantage of mutually independent instructions and execute them at the same time.

The basic realization of this concept is provided by Tomasulo's algorithm [Tom67]. It introduces two components, the first of which is the reservation station, which collects the operands of instructions until they are ready to be executed by the execution units. Crucially, the corresponding execution units do not need to wait until all operands are present and can instead compute other instructions. The second component of Tomasulo's algorithm is the Common Data Bus (CDB), which connects all reservation stations and execution units. Whenever a result is computed by an execution unit, the result is broadcast onto the CDB and thus made available to all reservation stations that are

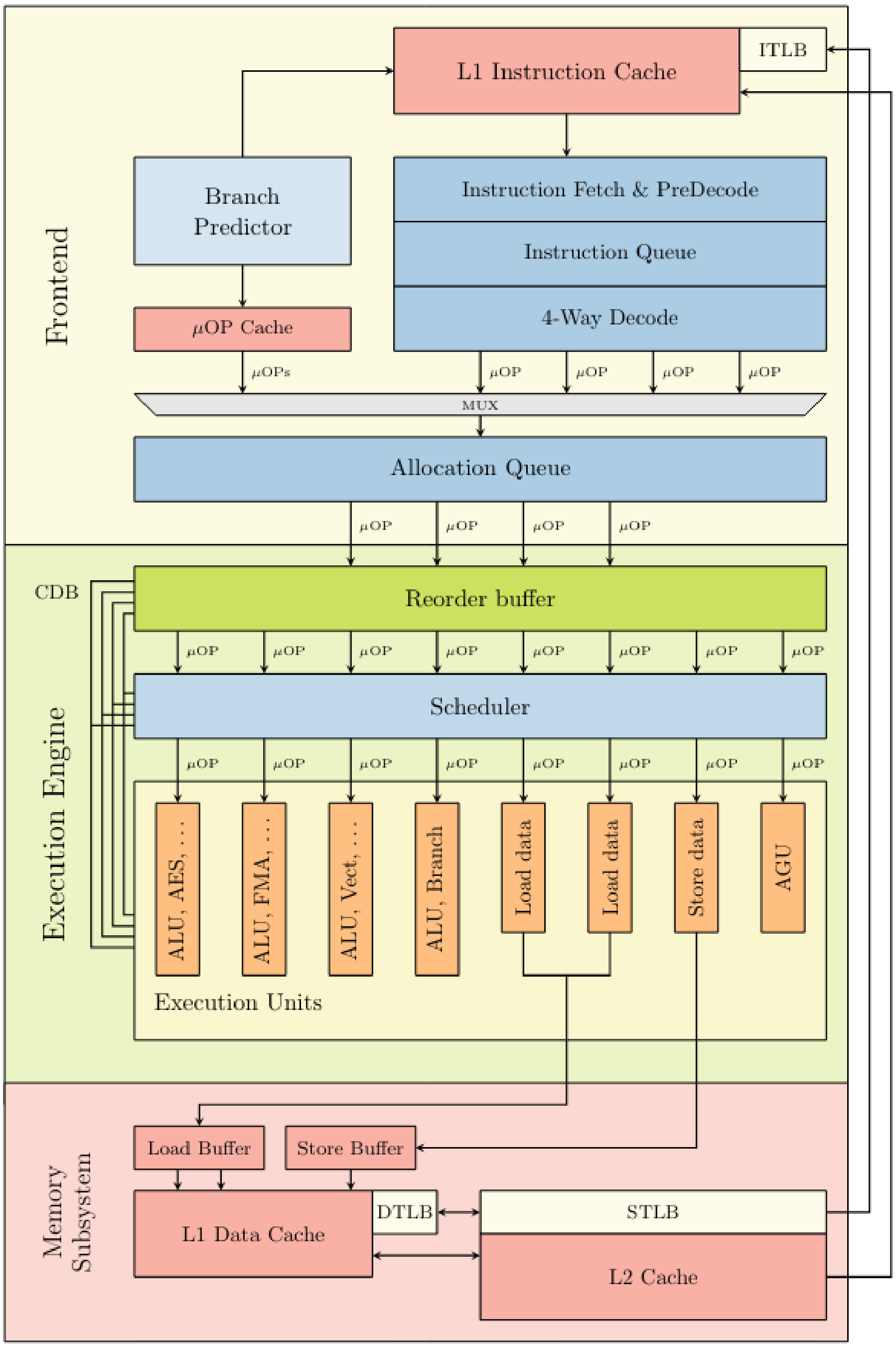

Figure 1: *Simplified overview an Intel Skylake CPU [Gru20, fig. 2.1]. For the memory subsystem, detailed knowledge of the load and store buffers, as well as the TLBs, is not required. The same applies to the allocation queue of the frontend.*

waiting for it. This important step ensures that results are not written to registers first, just to be read again by other instructions that need them as operands.

Initially, according to Tomasulo, each set of execution units needed its own reservation station, however, more modern implementations by Intel use a single unified reservation station, called scheduler, that handles all types of instructions, rather than just one [Tom67] [Wik] [Gru20]. This can also be seen in fig. 1.

2.3 Speculative execution

Speculative execution allows a CPU to predict the outcome of comparisons and other branch instructions. This prevents stalls when waiting for the instruction that determines which branch is taken to finish. Similarly to out-of-order execution, rollbacks are needed in some cases. However, in addition to exceptions, they also occur if a branch was mispredicted. [Gru20]

To predict the outcome of branch instructions, CPUs include a branch prediction unit (BPU) [Gru20]. While available in different configurations, many modern CPUs record the most recent outcomes of a branch with a counter [Gru20] that is either incremented if the branch is taken, or decremented.

2.4 Meltdown and Spectre

Meltdown [Lip+18] and Spectre [Koc+19] abuse out-of-order and speculative execution to leak data from memory addresses that are normally inaccessible to the attacker over the caches of the CPU.

2.4.1 Meltdown

On a high level, Meltdown [Lip+18] works by forcing exceptions when reading data inaccessible to the attacker and transiently encoding this data into the cache to retrieve it once the rollback has completed. What enables Meltdown is a small time window between an invalid memory access and the raising of an exception [Lip+18].

The Meltdown-US-L1 variant of Meltdown [Lip+18] to which the presented emulator is vulnerable works by accessing a memory address for which the attacker has no permission. Firstly, the attacker allocates an oracle array and ensures none of its entries are present in the cache. Upon loading the contents of the inaccessible memory location into a register, the attacker uses the value to access the array at a specific offset. When the rollback caused by the access violation has completed, the attacker measures the access time to each of the array entries to determine which one has been accessed. It is important that the data to be stolen is currently cached. While there are numerous other variants of Meltdown that differ from the basic variant mainly by how they force an exception and from which microarchitectural buffer they leak data, these types of attacks are out of scope.

2.4.2 Spectre

Spectre [Koc+19], on the other hand, relies on the CPU mispredicting a branch and transiently executing instructions that are not part of the correct execution path. This misprediction in a victim process can be induced by maliciously configuring the BPU (sec. 2.3). Depending on the instructions that are wrongfully executed by the victim, traces may later be found in the processor's cache. Similarly to Meltdown, different variants of Spectre exist. The presented emulator is vulnerable to variant 1 of Spectre, which takes advantage of the CPU mispredicting the outcome of comparison instructions.

2.4.3 Mitigations

The mitigations for Meltdown are available in both software and hardware, some of which are implemented in the emulator to allow users to experiment with them and determine their effectiveness. A first, rather simple, mitigation for Meltdown is to disable out-of-order execution [Lip+18], which would completely prevent an attacker from encoding the normally inaccessible data into the cache. Later revisions of Intel's architectures introduced further mitigations. Although undocumented by Intel, researchers suspect the processor still performs the illegal read, but zeros out the data that is given to dependent instructions before raising an exception [Can+20]. The emulator implements this mitigation, which may be enabled by the user. Other mitigations deployed by operating systems, such as KTPI (or KAISER) [Lip+18], are highly effective, but out of scope, as there exists no operating system.

Unlike Meltdown, Spectre appears to be a design flaw. While one might argue that transiently computing on real values after it is already known an exception will occur is a bug, the behavior abused by Spectre is a direct consequence of speculative execution (sec. 2.3). As a result, an effective yet questionable mitigation is to simply disable speculative execution [Koc+19]. Alternatively, Intel recommends potential victim's of Spectre v1 to use an "lfence" instruction where appropriate, which ensures prior load instructions retire before continuing, thus effectively disabling speculative execution for certain parts of an application [Int18]. A fence instruction is provided as part of the emulator's instruction set.

An additional mitigation that works against both Meltdown and Spectre is to flush the entire cache after a rollback. While highly inefficient, it does prevent the retrieval of otherwise inaccessible data after the transient execution phase of the attacks. This mitigation can be realized using the emulator by a sequence of instructions (microprogram) that are executed after each rollback.

3 Specification of our task

todo

4 CPU emulator/ backend

In this chapte, we introduce the backend of our emulator program. It contains the elements of our program that emulate actual CPU components. Our emulator is based on information about modern real life CPUs, especially the x86 Intel Skylake architecture [Wik].

In our program, we model the distinct components of a real life CPU with a modular setup making use of the object oriented functionalities of Python3. Breaking up the source code into individual CPU components also makes it easier to understand and maintain. Additionally, we have made simplifications and modifications in comparison to a real life CPU, so the emulator is as clear and easy to understand as possible while still implementing an actual out of order execution and providing the necessary functionality for Meltdown and Spectre attacks.

In this chapter we firstly introduce the components of our CPU emulator, how they work (together) and which part of a real life CPU they emulate sec. 4.1. Then we explain how our emulator provides out of order execution and how it may differ from the general Tomasulo algorithm introduced in sec. 2.2. Subsequently, we show how we implemented rollbacks and exception handling, especially with regards to how out the implementation allows for Meltdown and Spetre attacks sec. 4.3. Then we give an overview over our instruction set architecture sec. 4.4. Lastly we show how our emulator can be adapted for different demonstrations and attacks without changing the source code via a config file sec. 4.5.

4.1 CPU Components and our equivalents/ models (10-11 pages)

todo

4.1.1 CPU (F)

The primary purpose of the CPU module is to allow all other components to work together. That is, the CPU initializes all other components and provides interfaces to their functions, which allows the GUI to visualize the current state. In addition, the CPU provides functions which load user programs and a tick function that is called each cycle and does the following: instructions from the instruction queue are fetched from the frontend and forwarded to the execution engine, until either no more slots in the unified reservation station are available, or the instruction queue is empty. It then calls the tick function of the execution engine. In case of a rollback, the instruction queue maintained by the frontend is flushed. If the rollback was caused by a faulting load instruction, execution is resumed at the next instruction (as described in sec. 4.3). In case a branch was mispredicted, the frontend is notified and refills the instruction queue accordingly. If configured, corresponding microprograms are sent to the instruction queue. Lastly, a snapshot of the current state of the CPU is taken.

The second purpose of the CPU is to provide the snapshot functionality, which allows a user of the emulator to step back to previous cycles. The snapshot list is simply a list that grows at each cycles, where ech entry is a deepcopy of the CPU instance. To simply be able to deepcopy the CPU class, the snapshot list is held separately and not one of its members. Instead, the CPU class, and therefore each snapshot, maintains an index to its own entry in the snapshot list. This reference makes traversing the snapshot list one step at a time easier. Due to the low complexity of the programs we expect our users to run, at the moment, no maximum number of available snapshots is configured.

4.1.2 Instructions and Parser

todo

- Instruction mnemonic already determines the exact instruction, including the types of its operands
- Possible operand types: reg, imm, label
- Instructions distinguished based on instruction category: reg, imm, branch load, store, flush, special: cyclecount, fence
- Instruction object knows its mnemonic, types of its operands, which category of instruction it belongs to, and some category-specific information
  - For register-register and register-immediate instructions the concrete computation performed
  - For branch instructions the branch condition
  - For load and store instructions the width of the memory access
- Execution Engine only has to handle each instruction category, not all concrete instructions
- New instructions that fit an existing category can be added easily by the user
- Parser consumes abstract description of instructions, only mnemonic and operand types
  - Doesn't need to know about every concrete instruction
- Strips comments, introduced by //
- Handles labels, label name followed by :
  - Labels can be referenced before they are defined without special indication
- Two passes: first to extract all labels, second to actually parse instructions

4.1.3 Data representation

todo

how we model and handle data

byte, word

4.1.4 CPU FRONTEND

In modern CPUs, the CPU frontend provides an interface between code and execution engine. It contains components to fetch and decode the instructions from a cache and supply them to the CPU in a queue. It is also involved in speculative execution by predicting the result of conditional jumps and supplying instructions to the execution engine accordingly. [Wik]

In our emulator we simplify the components and procedures. We also separate the branch prediction unit (BPU), that manages and predicts the results of conditional jumps, from the rest of the frontend. This makes our code easier to understand and makes potential future changes or additions to the BPU more convenient.

BRANCH PREDICTION UNIT (BPU)

The BPU of modern CPUs plays a vital role in enabling speculative execution. It stores information about previously executed conditional branch instructions and predicts the outcome of future branch instructions accordingly. The rest of the CPU can then speculatively execute further instructions at an address based on the predicted outcome of the branch instead of stalling until the branch instruction is processed by the execution engine. If the prediction was false, a rollback is performed on the speculatively executed instruction sec. 4.3. If the prediction was true, the execution is overall faster than without speculative execution.

Detailed information about the BPU of our modern base CPU is not widely available [Wik]. In general, the Gshare BPU of modern CPU consists of multiple components. It contains a branch target buffer (BTB) that holds the predicted target addresses for conditional branches. It also uses a pattern history table (PTH), that can be implemented as a 2-bit-saturating counter, and a global history register (GHT) to predict whether a conditional jump will be taken or not. [**SCA**]

In our emulator, we forgo the BTB entirely. Real life CPUs benefit from stored target addresses since they have to expensively decode each branch instruction before they can work with the target address [Wik]. In our emulator, the parser decodes the jump labels from the assembler code and directly provides them to the frontend, so storing them in an additional buffer is unnecessary sec. 4.1.2. We further forgo the GHT because it is not strictly necessary to execute a Meltdown or Spectre attack. Additionally, our emulator and its behaviour are easier to understand and predict without it, which is important when implementing microarchitectural attacks for didactic purposes. The BPU of our emulator only consists of a PHT, which is enough for simple Spectre-PHT variants [**reference-eval-spectre**].

The default PHT used in our emulator holds an array called counter of configurable length to store several predictions. The instructions are assigned to different prediction slots by the last bits of their index in the instruction list. For each of the slots, the prediction can take the four different values from zero to three, where zero and one indicate that the branch will probably not be taken and two and three indicate that the branch is likely to be taken.

The source code for our emulator also contains a more simple BPU with only one slot for all instructions. Since the number of slots in the default BPU is freely configurable by the user, this simple BPU is now obsolete.

When the BPU is updated with an actual branch outcome from the execution engine, the prediction in the PHT is updated by a 2-bit counter. This means, that if the prediction was right, the counter remains at or updates to zero (strongly not taken) or three (strongly taken) respectively. If the prediction counter is at zero but the branch is actually taken, the counter is updated to one (weakly not taken). If it is at a one when the branch is taken, it is directly updated to three. The counter behaves similarly when it has tha value two or three and the branch is not actually taken.

Instruction Queue

In a real life CPU, the overall purpose of the frontend is to provide the execution unit with a steady strean of instructions so the backend is busy as much as possible and therefore efficient. In a modern x86 CPu, the frontend has to fetch x86 macro-instructions from a cache and decode, optimize and queue them repeatedly to provide the backend with a queue of μ-instructions ready for issuing in the execution engine. [Wik]

In our emulator, except for the BPU, the functionality of the CPU frontend is bundled in frontend.py. It is significantly simplified compared to a real life x86 CPU, especially since the we use only one type of instructions instead of distinguishing between macro- and μ-instructions sec. 4.4. They are already provided as a list by the parser, which renders the decoding and optimization steps in the frontend unnecessary sec. 4.1.2.

The main task of our frontend is to acts as interface between instruction list provided by the parser and the execution engine sec. 4.1.6. It provides and manages the instruction queue, which holds the instructions that the execution engine should issue next. Conditional branches with their respective BPU predictions are taken into account when filling the queue. This enables speculative execution which is needed for Spectre attacks sec. 2.4.

The central component of our emulated frontend ist the instruction queue. In our version, it does not only hold the instructions themselves, but also for every instruction in the queue, the respective index in the instruction list is stored. For branch instructions it also holds the respective branch prediction from the BPU at the time that the instruction was added to the queue. This additional information is needed by the execution engine to handle mispredictions and other exceptions sec. 4.1.6.

When adding instructions to the queue, the frontend selects them from the instruction list, adds the additional information for the execution engine and places them into the instruction queue until the queue's maximum capacity is reached. The frontend maintains a program counter (pc) that points to the next instruction in the list that should be added to the queue. When the frontend encounters a branch instruction and the branch is predicted to be taken, the frontend adjusts the pc to resume adding instructions at the branch target. If a branch was mispredicted, the frontend provides a special function to reset the pc and refill the instruction queue with the correct instructions.

Additionally, the frontend provides a function to add a μ-program to the queue. It consists of a list of instructions separate from the parser instruction list. When adding the μ-program to the queue, the frontend may exceed the maximum queue capacity. This functionality can be used to implement mitigations against microarchitectural attacks, e.g. by adding a µ-program as part of the exception handling after an illegal load [**reference-eval-mitigations**].

The frontend provides interfaces to both read and take instructions from the queue. It also provides a function that combines taking an instruction from the queue and refilling it, . Additionally, the frontend has an interface for flushing the whole queue at once without taking the instructions from the queue. This can be used when demonstrating mitigations against microarchitectural attacks [**reference-eval-mitigations**].

The frontend provides further basic interfaces, e.g. for reading the size the instruction queue and reading and setting the pc. These are used by the other components during regular execution, e.g. when issuing instructions to the execution engine, but also to reset the queue to a certain point in the program after an exception has occured sec. 4.3. Since our emulator only executes one program at a time, the other components can check via another interface whether the frontend has reached the end of the program.

4.1.5 Memory (F)

Memory is primarily managed by the Memory Subsystem (MS). The class contains a simple array that has $2^{WordWidth}$ entries, half of which are initialized to 0. Since our emulator does not run an operating system and therefore does not support paging, a different method is needed to model a page fault that allows attackers to enter the transient execution phase of the Meltdown-US-L1 attack (as explained in sec. 2.4). To solve this, we have decided to make the upper half of the address space (32768 to 65535, by default) inaccessible. Any reads to an address within the upper half result in a fault which causes a rollback a couple of cycles later. Naturally, the value written to the inaccessible addresses is $0x42$.

To handle memory accesses, *read_byte*, *read_word*, *write_byte*, and *write_word* functions are available, which do what their names suggest. Each function returns a *MemResult* object, which contains the data (0 for *write* functions), the number of cycles this operation takes, whether the operation should raise a fault (i.e. memory address is inaccessible), and, if so, how many cycles this should take. These values allow users to configure the width of the transient execution window.

Other functions that allow the UI to visualize the memory contents are provided. More specifically, *is_addr_cached* and *is_illegal_access* return whether an address is currently cached and whether a memory access to a specific address would raise a fault, respectively. Further, the MS includes functions that handle the cache management, such as *_load_line*, *flush_line*, and *flush_all*.

Meltdown Mitigation (F)

As explained in sec. 2.4.3, one of the mitigations implemented by Intel is believed to zero out data illegally read during transient execution. To model this, both the *read_byte* functions still perform the read operation, but provide 0 as the data in the returned *MemResult*, if the mitigation is enabled. As of now, the read operation still changes the cache, but since only the contents of the inaccessible memory address are cached and not the corresponding oracle entry of the attacker, the mitigation still works. The reason for this is that we believe a consequence of the CPU still performing the read operation but zeroing out the result should have side effects on the cache. If desired, this behavior can be changed easily in the *read_byte* functions.

Cache (F)

To enable tattackers to encode transiently read data, the MS maintains a single cache. The number of sets, ways, and entries per line can be configured via the config file (see sec. 4.5)

By default, there are three available cache replacement policies.

4.1.6 Execution Engine

The Execution Engine is the central component of a CPU. It is the component responsible for actually performing computations, by executing the stream of instructions provided by the frontend. Just like the Execution Engine of modern x86 processors, our Execution Engine executes instructions out-of-order, i.e. not necessarily in the order of the incoming instruction stream. In order to preserve the semantics of the program, any data dependencies have to be honored during reordering. For this we use a modified version of Tomasulo's Algorithm, that is described in detail in sec. **??**.

The Execution Engine contains the Reservation Station with a fixed number of instruction slots. Each slot contains an instruction that is currently being executed. We call such instructions *in-flight*. Our Reservation Station is unified, i.e. each slot can contain any kind of instruction. The same is often found in modern CPUs. The slots of our Reservation Station are also used to model Load Buffers and Store Buffers; the specifics of executing memory accesses are handled by the slots directly instead of separate components. We also have no concept of Execution Units that instructions need to be dispatched to, which means that instructions' ability to execute concurrently is only limited by the number of available slots.

All instructions pass through two phases during execution: In the first phase the instruction is said to be *executing*. It waits for any source operands to become available and computes its result. Once the result is computed, it is made available to waiting instructions, and the instruction transitions to the second phase.

In the second phase the instruction is said to be *retiring*. It determines if it causes a *fault*, which in this case means a *microarchitectural* fault. These can be architecturally visible faults like memory protection violations or architecturally invisible faults like branch mispredictions; both are handled the same way in the Execution Engine. Once the instruction finishes retiring its slot becomes available again and may be used to execute a new instruction.

In each clock cycle only a single instruction may finish execution or retirement. This models the contention of the Common Data Bus, which is used to provide information about computation results inside the Execution Engine and to other components and can only transmit information about a single result each clock cycle.

Besides the Reservation Station, the Execution Engine also contains the Register File, with one entry for each register. Each register entry either contains the concrete value of the register or references a slot of the Reservation Station that will produce the register's value. Since instructions are issued in program order, the state of the register file at a single point in time represents the architectural register state at that point in time, with yet-unknown register values present as slot references.

4.2 Out of Order Execution (2 pages)

todo

4.3 Exceptions and Rollbacks (2-3 pages)

todo

4.4 ISA (2 pages)

Real life Intel x86 CPUs differenciate between two types of instructions or operations. Macro-operations refer to the relatively easily human readable and convenient but complex instructions that are described by the x86 ISA. Their length differs between the instructions. Internally, in the execution units, the CPU works on µ-operations, which are small operations of a fixed length. One macro-operation contains one or multiple µ-operations. The CPU frontend has to decode the macro-operations into µ-operations in an expensive multi step process. sources: [Wik], https://en.wikichip.org/wiki/macro-operation, https://en.wikichip.org/wiki/micro-operation

Our CPU emulator only uses one type of instructions. They are directly read from our assembler code by the parser and passed to the execution engine without further decoding, splitting or replacing sec. 4.1.2, sec. 4.1.4. To show basic Meltdown and Spectre variants, we do not need overly complex instructions, e.g. instructions that contain multiple memory accesses in one or that are used to perform encryption in hardware [ref_evaluation_meltdown], [ref_evaluation_spectre]. Basic arithmetic operations, memory accesses, branches and a few special operations are sufficient for the demonstrated attacks and are both easy to implement as single instructions and to use in assembler code that should be well understood by the author. Using the same operations throughout the emulator also makes the visualization more clear and easier to follow, e.g. when the same operations appear, one after the other, in the visualization of the assembler code, the instruction queue and the reservation stations [ref_ui].

4.4.1 Default Instruction Set

In order that our CPU emulator can recognize and work with an instruction, it has to be registered with the parser sec. 4.1.2. In our default setting default, we register a basic set of instructions with the parser so students can start writing assembler code and using the emulator right away. This basic instruction set is also used in our example programs in [ref_UI].

Our relatively small instruction set is based on a subset of the RISC-V ISA. It offers a selection of instructions that is sufficient to implement Meltdown and Spectre attacks as well as other small assembler programs while still being of a manageable size so students can start to write assembler code quickly without spending much time to get to know our ISA. The syntax of the assembler representation is also based on RISC-V (as introduced in the "RISC-V Assembly Programmer's Handbook" chapter of the RISC-V ISA) [ref_RISC-V]. If needed, students can add further instructions by registering them with the parser sec. 4.1.2.

In the following subchapters we introduce the instructions of our default ISA. They are grouped according to their respective instruction type in the emulator except for the special instructions which are grouped together sec. 4.1.2. All default instructions are summarized in the appendix into a quick reference sheet [**ref_appendix**].

Arithmetic and Logical Instructions without Immediate

These are basic arithmetic and logical instructions that operate solely on register values, i.e. both source operands and the destination operand reference registers. For simplicity, we write, for example, Reg1 when referring to the value read from or stored in the register referenced by the first register operand.

Each of these default operations uses the respective python standard operator on our Word class to compute the result, except for the right shifts. For the logical and the arithmetic right shift, the python standard right shift operator is used on the unsigned and the signed version of the register value respectively. When returning the result as a Word, it is truncated to the maximal word length by a modulo operation, if necessary sec. 4.1.3. This means, that any potential carry bits or overflows are effectively ignored.

Arithmetic and Logical Instructions without Immediate		
Instr. Name	Operators	Description
add	Reg1, Reg2, Reg3	Reg1 := Reg2 + Reg3
sub	Reg1, Reg2, Reg3	Reg1 := Reg2 − Reg3
sll	Reg1, Reg2, Reg3	Reg1 := Reg2 << Reg3
srl	Reg1, Reg2, Reg3	Reg1 := Reg2 >> Reg3 logical
sra	Reg1, Reg2, Reg3	Reg1 := Reg2 >> Reg3 arithmetical
xor	Reg1, Reg2, Reg3	Reg1 := Reg2 xor Reg3
or	Reg1, Reg2, Reg3	Reg1 := Reg2 or Reg3
and	Reg1, Reg2, Reg3	Reg1 := Reg2 and Reg3

Arithmetic and Logical Instructions with Immediate

These are basically the same instructions as in sec. 4.4.1. The main difference is, that the second source register is replaced by an immediate operand which is set directly in the Assembler code. This immediate is used as the value of a Word in the execution engine, so it is truncated by a modulo operation to be in the appropriate range sec. 4.1.3, sec. 4.1.6.

Arithmetic and Logical Instructions with Immediate		
Instr. Name	Operators	Description
addi	Reg1, Reg2, Imm	Reg1 := Reg2 + Imm
subi	Reg1, Reg2, Imm	Reg1 := Reg2 − Imm
slli	Reg1, Reg2, Imm	Reg1 := Reg2 << Imm
srli	Reg1, Reg2, Imm	Reg1 := Reg2 >> Imm logical
srai	Reg1, Reg2, Imm	Reg1 := Reg2 >> Imm arithmetical
xori	Reg1, Reg2, Imm	Reg1 := Reg2 xor Imm
ori	Reg1, Reg2, Imm	Reg1 := Reg2 or Imm
andi	Reg1, Reg2, Imm	Reg1 := Reg2 and Imm

Memory Instructions

These instructions provide basic interactions with the emulated memory sec. 4.1.5. Load and store instructions exist in two versions, one that operates on Word length data chunks, for convenience, and one that operates on Byte length data chunks, for the fine granular access needed in micro architectural attacks. The flush instruction flushes the cache line for the given address sec. 4.1.5. The address is calculated in the same way for all memory instructions: addr:=Reg2+Imm, and addr:=Reg+Imm for the flush instruction respectively.

Memory Instructions		
Instr. Name	Operators	Description
lw	Reg1, Reg2, Imm	Reg1:=Mem_word[addr]
lb	Reg1, Reg2, Imm	Reg1:=Mem_byte[addr]
sw	Reg1, Reg2, Imm	Mem_word[addr]:=Reg1
sb	Reg1, Reg2, Imm	Mem_byte[addr]:=Reg1
flush	Reg, Imm	flush cache line of addr

Branch Instructions

All branch instructions compare the values of two source registers. If the comparison evaluates to true, the execution of the program is resumed at the given label in the assembler code. If it evaluates to false, the next instruction in the program is executed. Depending on the instruction, the register values are interpreted as signed or unsigned integers. Labels in the assembler code are automatically resolved by the parser sec. 4.1.2, [rev_eval_and_example_code].

Branch Instructions		
Instr. Name	Operators	Description
beq	Reg1, Reg2, Label	jump to Label if Reg1==Reg2
bne	Reg1, Reg2, Label	jump to Label if Reg1! =Reg2
bltu	Reg1, Reg2, Label	jump to Label if u(Reg1)<u(Reg2)
bleu	Reg1, Reg2, Label	jump to Label if u(Reg1)<=u(Reg2)
bgtu	Reg1, Reg2, Label	jump to Label if u(Reg1)>u(Reg2)
bgeu	Reg1, Reg2, Label	jump to Label if u(Reg1)>=u(Reg2)
blts	Reg1, Reg2, Label	jump to Label if s(Reg1)<s(Reg2)
bles	Reg1, Reg2, Label	jump to Label if s(Reg1)<=s(Reg2)
bgts	Reg1, Reg2, Label	jump to Label if s(Reg1)>s(Reg2)
bges	Reg1, Reg2, Label	jump to Label if s(Reg1)>=s(Reg2)

Special Instructions

Rdtsc acts like a basic timing instruction. It returns the number of ticks the execution unit has executed so far in the given register.

The fence instruction acts as a fixed point in the out of order execution. All instructions that are already issued in the execution unit at the point of issueing the fence instruction are executed before the fence is executed. No new instructions are issued before the execution of the fence instruction is complete.

Special Instructions		
Instr. Name	Operators	Description
rdtsc	Reg	Reg:=cyclecount
fence	none	add execution fixpoint at this code position

4.5 Config Files (1 page)

todo

5 User Interface and Usage

todo

6 Evaluation

todo

7 Conclusion

todo

8 Catchy Lab title

8.1 Abstract

8.2 Introduction (1 page)

```
general task/ goal
    CPU emulator with actual out of order execution, maybe speculative execution and comprehensible implementation and documentation
    implement different µ-architectural features like out-of-order exe., speculative exe. b.c. we want to observe and demonstrate µ-arch. attacks
brief context of the task/ goal (e.g. a sentence on Meltdown and why it is important to understand it)
    why is it important to have the emulator: so far no possibility to observe microarchitectural attacks in real life
    rauscharme Umgebung, die macht was man erwartet
structure of the report
```

8.3 Brief Theoretical Background (2-3 pages)

```
premise according to Felix: reader knows SCA lecture -> brief recaps/ reminders to reference back to from the other chapters

### really brief introduction to CPU

    multiple components
        maybe mostly via picture

    how they work together

    maybe data-flow from instruction to result

    a lot of hypothticals due to trade secrets

    suggested literature:
        maybe Gruss Diss. and other works on cbsca
        maybe textbooks
        maybe CPU wiki for pictures

### out-of-order execution
```

```

    a bit more in depth b.c. this is not content of the sca lecture

    maybe why do we need it/ advantages

    brief explanation of Tomasulo and how it works

    suggested literature:
        original Tomasulo paper from 1965
        maybe a more recent/ didactically edited explanation (is this in texbooks?)

    also something about speculative execution

### Meltdown and Spectre

    brief introduction to Meltdown (and Spectre) in general

    slightly more in-depth introduction to the Meltdown attack we picked and want to run in our emulator

    highlight the relevant parts of the CPU, maybe mention how they interact in Meltdown (and Spectre)

    mitigations and maybe how successfull they are

    suggested literature:
        ggf. Dissertation Gruss if we mention basic cbsca
        Gruss et al. papers on Meltdown and Spectre e.g. https://gruss.cc/files/meltdown.pdf or https://gruss.cc/files/meltdown_cacm.pdf
        Canella, Gruss et al. on mititgations https://gruss.cc/files/transient-attacks.pdf
```

8.4 maybe Specification of the task (1 page)

```
rauswerfen, in Introduction mit übernehmen

emulator to execute and teach Meltdown
    who is the target audience´
    which Meltdown attacks specifically
    etc. further concretisations

are there existing solutions/ related works?
    Felix' old emulator? citable?
    suggested literature:
        todo
```

8.5 CPU emulator/ backend (17-18 pages)

```
maybe general info e.g. that we wrote it in python, if we used special
    tools/ libraries
for everything we implemented:
        which part of a real life cpu does this model/ how is this done in
    real life cpus
        briefly how does it work
        why did we choose to model it like we do
            nice code structure/ code easy to understand by students
            some features more or less relevant for meltdown
            etc.
        maybe what did we leave out
        challenges?

### CPU Components and our equivalents/ models (10-11 pages)

    modular setup based on real life CPUs and nice coding conventions

    #### CPU

        contains/ controls the rest

        preparation (loading the program, init the rest)

        ticks

        coordinate rollbacks?

    #### Instructions and Parser

        instructions

        parser

    #### Data representation

        how we model and handle data

        byte, word

    #### CPU frontend

        Branch Prediction Unit (BPU)

        Instruction Queue

    #### memory

```

```
        no virtual adresses

        mmu

            how do we store data

            how do we model data access (i.p. wrt Meltdown)

        cache

            what kinds of caches can we represent

            how do we model if something is cached and the access times (i.p. wrt Meltdown)

    #### Execution Engine

        Reservation Station/ Slots

        unlimited execution units

### Out of Order Execution (2 pages)

    our version of out-of-order execution/ Tomasulo
        how much details on Tomasulo? our choice

    where in our program do we implement which components

    what did we leave out/ do differently

### Exceptions and Rollbacks (2-3 pages)

        in particular wrt making Meltdown possible

        general goal/ concept of rolling back after a misprediction or an exception

        snapshots and interaction of the CPU components
            components that are fully reset after a rollback: cache, BPU

### ISA (2 pages)

    overview of ISA

    µ-instr. vs. ISA
        mixture of both in one: only µ-code, but more abstract than in real life
        do not need to think about page faults etc. anyways, do not need overly complex instructions
```

```

    reasoning behind the choice of instructions (manageable size and instructions (e.g. no divide by zero) balanced with functionality particularly wrt Meltdown)


### config files (1 page)

    what can be configured without changing the source code
        why these variables? relevant for Meltdown?
```

8.6 Our Visualisation and Usage/ Frontend (10 pages)

```
ggf. gemäß Anmerkung von Lenni umsortieren: die idee das UI nicht mit in die 'unsere designetnscheidungen' zu nehmen, sondern im prinzip so als Manual abzukapseln finde ich eigentlich ganz gut. Müssen wir aber dann mal in der Praxis schauen. Soll ja keine didaktischen begründungen das manual zu sehr aufblähen, vllt wird das sinnvoll, dann einen teil des UI im "backend" kapitel einzubauen, und dann wirklich ein cleanes manual kapitel zu haben

zwei Strukturierungsmöglichkeiten:
    z.B. alphabetisch
    z.B. nach einzelnen Angriffen etc. aufgeteilt und aufeinander aufbauend
erklären was technisch passiert nachdem man gezeigt hat wie es aussieht
man kann auch explizit sagen, "ich gehe davon aus, dass du GDB kannst, daran ist das angelehnt, hier sind die Unterschiede"

### general concept

    goal: UI for the emulator with visualisation of CPU/ memory components and their contents

    in terminal
        maybe comparison to existing analysis tool/ debugger gdb

    triggers/ controls the actual emulator
        overview features, e.g. breakpoints, step-by-step and stepback

    maybe which tools/ libraries were used?

### features in more detail and their didactic purpose

    breakpoints, step-by-step, stepback and more

    challenges/ design choices during the implementation
```

8.7 Demonstration/ evaluation (7 pages)

```
what kind of system do we need/ did we use to run this?
    which python Version?
    which program version ? git commit
    other dependencies?

### general demonstration

    brief example program showing all the features in a "normal" execution, e.g. adding stuff

### Meltdown und Spectre demonstration

    #### Example Program Meltdown

        show example program

        maybe compare to example program for real life architecture from SCA or literature (Gruss) if available

        explain which Meltdown variant it implements

        briefly highlight which components (we expect to) interact to make it work

        how well does it work?

    #### maybe example program spectre

        same as Meltdown

    #### mitigations

        it is known which mitigations exist, here is what we have in our emulator:

        what is possible in our program as is
            planned:    cache flush: microcode -> config file
                        mfence im assembler (normally in compiler)
                        aslr directly in program -> config (es gibt ja auch mitigations, die keine echte mitigation sind; nice to have -> könnte demonstrieren dass es nicht der Fall ist; war eh schon einige Jahre vor Meltdown vorhanden/ in Gebrauch; KSLR brachen kann man auch als Angriff verkaufen)
                        flush IQ -> passiert eh schon, ist das überhaupt eine echte mitigation?
                        disable speculation (nice to have, lassen wir weg) -> config
```

```
                        out of order -> in config RS mit nur einem Slot

    is our meltdown/ spectre variant still possible?
    ggf. how does this affect the performance?
    vorsichtig sein, dass man dann auch die richtige Frage für die Antwort stellt
        in real life (already in background)
        in our program

    what would be the necessary steps/ changes to the program for further mitigations
        compare to changes in hardware by the manufacturers
```

8.8 Conclusion (1 Page)

```
recap goals
main goal of this chapter: to which extend did we reach our goals?

did we reach the goal of implementing a CPU emulator where a user can perform a Meltdown attack

how many Meltdown attacks are possible?

is Spectre possible?

mitigations
    which mitigations did we implement
    is this a good amount/ sample of real world mitigations or do we miss important ones?
    how well do they perform (also in comparison to how they perform in the real world)

value to students:
    how easy to use and convenient do we think our program is?
    do we think this will be a good tool for teaching?
    how accessible is it wrt different host architectures?
    1 überzeugter Satz: wir haben sehr gute Arbeit geleistet und das Werkzeug ist sehr gut. wenig hin schreiben, damit keiner fragt, ob wir das nicht weiter evaluieren müssten. nicht weit aus dem Fenster mit Behauptungen lehnen sondern klar und kurz unsere Meinung formulieren

further work
    more (detailed/ realistic) functionality for more Meltdown and Spectre variants
        more elaborate BPU with btb (and ghr) for more spectre variants
    more mitigations
    maybe nice to haves wrt to the visualisation/ general UI functionality/ ISA?
```

References

[Can+20] Canella, Claudio et al.: "KASLR: Break it, fix it, repeat". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 2020, pp. 481–493.

[Gru20] Gruss, Daniel: "Transient-Execution Attacks". 2020. url: https://gruss.cc/files/habil.pdf (visited on 03/11/2022).

[Int18] Intel: *Intel Analysis of Speculative Execution Side Channels*. 2018. url: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf (visited on 03/11/2022).

[Koc+19] Kocher, Paul et al.: "Spectre attacks: Exploiting speculative execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019.

[Lip+18] Lipp, Moritz et al.: "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[Tom67] Tomasulo, Robert M: "An efficient algorithm for exploiting multiple arithmetic units". In: *IBM Journal of research and Development* 11.1 (1967), pp. 25–33.

[Wik] WikiChip: *Skylake Microarchitecture*. url: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client) (visited on 03/10/2022).

List of Figures

Statement of Authorship

I hereby confirm that the work presented in this bachelor thesis has been performed and interpreted solely by myself except where explicitly identified to the contrary. I declare that I have used no other sources and aids other than those indicated. This work has not been submitted elsewhere in any other form for the fulfilment of any other degree or qualification.

TODO

TODO