John Askeland Lauvdal

Investigating Speculative Side-Channel Protection

Master's thesis in Computer Science
Supervisor: Magnus Själander
Co-supervisor: Amund Bergland Kvalsvik
June 2023

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

John Askeland Lauvdal

Investigating Speculative Side-Channel Protection

Master's thesis in Computer Science
Supervisor: Magnus Själander
Co-supervisor: Amund Bergland Kvalsvik
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

Speculation is a widely used optimization implemented in processor cores to improve performance by getting more work done and preventing the pipeline from being idle. It has however later been discovered that the great performance gains comes with the cost of being vulnerable to speculative side-channel attacks. This is a significant security flaw, which enables attackers to read sensitive data, like passwords and encryption keys, from victims. Almost all newer processors use speculation, and are thus vulnerable to these attacks. This is a problem since both the attributes of fast performance and security against speculative side channels attacks are considered as desired when designing processor cores. However, since speculation now make the processor vulnerable to these attacks, there is a conflict between these two attributes and a compromise have to be made.

This report explores different mitigation strategies against the attack, and describes the implementation of the mitigation strategies *Delay-on-Miss* (DoM) and *Speculative Taint Tracking* (STT) implemented in the *Berkeley Out-of-Order Machine* (BOOM) processor core. It additionally explores a new variation of the DoM implementation, referred to as DoM-RW, which are more restrictive on waking up the delayed loads when they have missed in the L1 cache. The STT implementation was not completed in time, and is only partially implemented. The results from the experiments performed on this implementation can therefore not be seen as representative for the mitigation strategy. They are, however, included in the report since they can still give insight on the partially implemented solution.

The implemented solutions have been evaluated on an FPGA. When exploring the implemented mitigations, experiments testing the implementations for performance and security are performed. The test for performance are evaluated by executing benchmarks from the SPEC2017 benchmark suite. To test for security, the Spectre attacks Spectre Variant 1 and Spectre Variant 2 have been evaluated. It is additionally looked at how large the implementation overhead of the implemented mitigation strategies are in regard to physical size by reviewing the FPGA utilization of the implemented strategies.

Since STT is only partially implemented, it is not tested as comprehensive as the other implementations. It is not tested for performance, and the test for security is not performed on the same platform as the other implementations. The Spectre attacks performed are also terminated before the attack is finished.

Both the DoM implementations are demonstrated to protect data from being leaked against the two speculative side-channel attacks. The partially implemented STT implementation however leaks data. The normal DoM implementation get a slowdown in performance of 14% and the DoM-RW implementation get a slowdown of 20.9%. The implementation overhead of both the DoM implementations is measured to be 0.1% increase in the number of look-up-tables (LUTs) used for logic on the FPGA and 0.02% increase in the number of flip-flops used. The STT implementation have an increase of 1.2% in the number of LUTs used for logic, and 0.49% increase in flip-flops used compared to the baseline implementation.

The key contribution of this report is to bring new insight about how well the implemented mitigation strategies fulfills the desire for processor cores to be both fast and secure.

Sammendrag

Spekulasjon er en mye brukt optimalisering i prosessor kjerner og har bidratt til å forbedre ytelsen ved å utføre mer arbeid og forhindre at pipelinen står stille og venter. Det er imidlertid i senere tid blitt oppdaget at de den forbedrede ytelsen kommer med kostnaden av å være sårbar for spekulative sidekanalangrep. Dette er en betydelig sikkerhetssårbarhet som gjør det mulig for angripere å stjele sensitiv informasjon, som passord og krypteringsnøkler fra ofre. Nesten alle nyere prosessorer bruker spekulasjon og er dermed sårbare for slike angrep. Dette er et problem siden både rask ytelse og sikkerhet mot spekulative sidekanalangrep anses som ønskelig når man designer prosessor kjerner, men siden spekulasjon nå gjør prosessoren sårbar for disse angrepene, oppstår det en konflikt mellom disse to egenskapene.

Denne rapporten utforsker ulike strategier for å unngå sårbarheten ved å beskrive implementasjonen av forsvarsstrategiene *Delay-on-Miss* (DoM) og *Speculative Taint Tracking* (STT) implementert i *Berkeley Out-of-Order Machine* (BOOM) prosessor-kjernen. Den utforsker også en ny variant av DoM-implementasjonen, som i denne rapporten blir omtalt som DoM-RW. Denne er mer restriktiv når det gjelder å vekke opp de spekulative load instruksjonene som har blitt hindret fra å kjøre. Implementeringen av STT-strategien ble ikke fullført i tide og er derfor bare delvis implementert. Resultatene fra eksperimentene utført på denne implementasjonen kan derfor ikke ansees som representative for strategien. Likevel er de inkludert fordi de fortsatt kan gi innsikt i den delvis implementerte løsningen.

De implementerte løsningene er evaluert på en FPGA. For å utforske de ulike strategiene har det blitt utført eksperimenter for å teste implementasjonene med hensyn til ytelse og sikkerhet. Ytelsestestene evalueres ved å kjøre benchmarkene fra SPEC2017-benchmark-suitten. For å teste sikkerheten blir Spectre-angrepene, Spectre Variant 1 og Spectre Variant 2, evaluert. Det blir også sett på hvor stor implementasjonsoverhead de implementerte strategiene har med tanke på den fysiske størrelsen.

Siden STT bare er delvis implementert, er det ikke testet like omfattende som de andre implementasjonene. Den er ikke testet for ytelse, og sikkerhetstesten er heller ikke testet på samme plattform som de andre implementasjonene. Spectre-angrepene som ble utført terminerer også før angrepet er fullført.

Det viser seg at begge DoM-implementasjonene beskytter mot lekkasje fra begge de to spekulative sidekanalangrepene. Den delvis implementerte STT-implementasjonen lekker imidlertid data. Den vanlige DoM-implementasjonen får en reduksjon i ytelse på 14%, og DoM-RW-implementasjonen får en reduksjon på 20,9%. Implementasjonsoverheaden for begge DoM-implementasjonene måles til å være en økning på 0,1% i antall oppslagstabeller brukt til logikk på FPGA-en og en økning på 0,02% i antall vippere brukt. STT-implementasjonen har en økning på 1,2% i antall oppslagstabeller brukt til logikk og en økning på 0,49% i antall vippere sammenlignet med utgangsimplementasjonen.

Hovedbidraget fra denne rapporten er å gi ny innsikt i hvor godt de implementerte løsningene oppfyller ønsket om at prosessorkjerner skal være både raske og sikre.

Acknowledgements

I have to start this thesis by thanking my supervisors professor Magnus Själander and PhD candidate Amund Bergland Kvalsvik for their contribution to the work. The help I have got both with the implementations performed, and feedback when writing the report have been very valuable, and the thesis would not have been what it is without it.

I would also like to give a thanks to PhD candidate Björn Gottschall and PhD candidate David Metz for being available for questions on Slack and helping when performing the tests on the FPGA.

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

In 2018, the discovery of attack strategies exploiting some architectural optimizations caused fright in the computer architecture community. The reason for this was that the newly discovered attacks made for a conflicting compromise between the two desired properties of security and performance. It was discovered that the optimizations speculation and out-of-order execution could leak data trough side-channels in the microarchitectural state. Mitigating the attacks by removing these optimizations would lead to a non-negligible decrease in performance, which would set the development of fast processor cores several steps back.

After the discovery of the first Spectre attacks, several mitigation strategies have been suggested. They all try to find a solution which provides security with as small slowdown of the processor core as possible. This project aims to give new knowledge about how large of an overhead, in regard to performance and physical size, it is to secure processor cores against speculative side channel attacks. This is done by implementing already existing mitigation strategies discovered by other researchers. The mitigation strategies explored are two variations of *Delay-on-Miss* (DoM) [1], in addition to *Speculative Taint Tracking* (STT) /[2]. These are implemented on the Berkeley Out-of-Order Machine (BOOM) [3].

1.1 Motivation

As the technology advances, there is an increased demand for fast processors. Over the last couple of decades, the demand for heavy computations have increased massively. At the time of writing this report, it is also not expected to decline. To satisfy this demand, computer architects are constantly looking at new methods to increase the speed of computer cores in addition to finding new clever ways of doing more computation in a shorter amount of time. Dennard scaling states that as transistors gets smaller, the power density stays close to constant. Architects have until the end of Dennard scaling [4] relied mostly on reducing the size of the transistors to increase the clock rate and thus the performance. This was the case until the mid 2000s. Since then, architects have been required to improve the efficiency in single cores with other methods. These methods, for example, include pipelining, out-of-order execution, speculation amongst others. The strive for more effective processor cores have resulted in an increased complexity in the cores.

In addition to performance, the demand for security in digital devices have also increased. It is safe to assume that the more of the facets of the society turn digital, the more enticing is the digital space for criminals and others with malicious motives. Nowadays, almost all money transactions in the most developed countries are digital. For this to be safe, sensitive data like encryption keys and passwords cannot be leaked when being processed on the hardware. As processor cores have become more complex, the discovery of speculative side-channel attacks have proved that some of the optimization employed in processor cores can be exploited to leak sensitive data. If an attacker is able to trick you into running malicious code, it can potentially read your entire main memory, which may include encryption keys and passwords. The code to perform these types of attacks can easily be hidden within the other code in a normal application.

While it may, for most hardware designs, not be as relevant as having performance and security, it is also interesting to review the implementations overhead with regard to the physical size for different the designs. A large implementation overhead of mitigation strategies could potentially lead to a larger physical size of the core. Physical size is mostly a concern for devices which are expected to be lightweight and have a large degree of portability, like for example mobile phones. Cores in embedded systems are sometimes also desired to be of a small size, depending on the use case of the embedded system. However, for processor cores designed for desktop machines and warehouse scale computing, physical size is not a limiting factor in the development.

Knowing the traits desired in computer cores, the motivation of this work is therefore to explore some mitigation strategies securing the core against speculative side-channel attacks and look at the efficiency and security provided. The work aim to compare relevant solutions in regard to the performance, security provided and implementation overhead on the same platform to give new knowledge about how well the implementations provides these desired traits.

1.2 Scope

The project described in this report is the continuation of the author's specialization project [5] in the autumn semester in 2022. The specialization project implemented DoM in *Verilator* to explore the security of the implementation. Some benchmarks were also run, but as mentioned in the report from the specialization project, the tests are not credible enough to conclude anything about the performance of the modified processor core. As this project is the continuation of that project, some of the text in that report is reusable for this report, as there have been a minimal amount of change in the implementation of DoM. With permission from NTNU, the reusable parts are directly copied from the specialization project's report and used for this report. As the scope of the specialization project was much smaller, there is not a lot of copied text, and it is only text regarding the implementation of DoM, in addition to some background theory.

This project continues exploring the DoM implementation by implementing it on an FPGA to get more credible data about the performance and also the implementation overhead. This project additionally aims to implement the strategies DoM-Restricted Wakeups (DoM-RW) and STT on an FPGA and compare them with DoM and the baseline implementation. DoM-RW is a modification of DoM, to the author's knowledge, first implemented for this research. The difference of these are described in Section 3.1.4.

Due to the time limit of the project, the scope has to be narrowed down to a manageable amount of work. The work performed should also be relevant to the overall goal stated in Section 1.3. Implementing the mitigation strategies DoM, DoM-RW and STT is part of the scope as it is very important for collecting data used to accomplish the overall goal. Evaluating the implementations on an FPGA is also in the scope and is considered as indispensable for this research. While implementing STT initially is part of the scope of this research, it is worth mentioning that the implementation did not get finished within the time limit of the project, and is therefore only partly implemented. Any of the planned experiments that is possible to perform on the partly completed implementation is however performed and evaluated like on the other implementations.

Measuring performance and implementation overhead is also crucial for accomplishing the overall goal. The performance is measured in instructions per clock cycle (IPC) and the implementation overhead measured is measured in the amount of hardware of the FPGA that is utilized. Any other measurements of the performance and implementation overhead is considered out of scope.

Running attack code to test the security of the implementations is also considered a crucial part of accomplishing the goal and are for this reason considered to be part of the scope. While it would be relevant for giving greater insight to the level of protection the implementations provide, other Spectre attacks than Spectre Variant 1 and Variant 2 are not performed. It could be interesting to see if the implementations additionally are secure against the Meltdown attack and Spectre Variant 4. However, creating attack code for these attacks would be too time-consuming to meet the deadline of the delivery. Other attacks than Spectre Variant 1 and Variant 2 are therefore considered out of scope for this project.

It could also be valuable to evaluate the implementations for other traits than performance and implementation overhead in addition to the security against Spectre Variant 1 and 2. For example, energy efficiency is also a desired trait in processor cores. However, it is not considered as part of the scope for this project.

Although evaluating other traits than performance and implementation overhead in addition to the security against Spectre Variant 1 and 2 like energy efficiency, could also make for interesting research, it is, however, considered out of scope for this project.

It is also not considered as part of the scope to try to answer why the performance and implementation overhead is as it is. Only measuring and comparing the desired obtained results are considered as part of the scope. There are therefore not performed any experiments analyzing how the benchmarks are executed on the hardware to get the measured result, or experiments trying to find out why the different implementations get the implementation overhead it gets. Although finding out why the result is what it is being out of scope, it is still discussed as the report should facilitate for future research.

1.3 Contrubution

The overall goal of the project is to find out how the protection schemes implemented compare to the baseline and each other in terms of protection, performance, and implementation overhead. Experiments are set up to quantify and measure these traits and generate data reflecting how the implementation affects any of the specific traits.

The main contribution of the project is gaining new knowledge of how the implemented solutions perform with regard to the suggested desired properties. This knowledge can give useful new insight about which of the implemented solutions that are the best choice for mitigating speculative side-channel attacks. The insight can also be useful for further research on the field of speculative side-channel attacks. The contribution is summarized in the list below.

- It is discovered that the DoM and DoM-RW implementation provide security against the speculative side-channel attacks Spectre Variant 1 and Spectre Variant 2.
- The solutions are measuered to have a slowdown 14% for DoM and 20.9% for DoM-RW. The implementation overhead is also measured to be so small that it can be considered negligible.
- It is discovered that the DoM implementation with a relaxed wakeup policy may be considered more preferable than the implementation with restricted wakeups, as it gives better performance on the benchmarks, while having equal implementation overhead and security provided.

Chapter 2

Background

As the end of Dennard scaling [4] has been reached, architectural optimizations have become more important to increase performance in processor cores. Manufacturers can rely less on getting performance from only shrinking transistors, and therefore have to improve the processor cores' design to gain performance [6]. In this chapter, some technical background information will be provided about fundamental computer architecture, optimizations implemented in processor cores, and how some of these optimizations can be exploited by attackers. This information will supply the required knowledge to understand Delay-on-Miss and Speculative Taint Tracking which is implemented for this project. The chapter will also give some technical background on some of the tools used to implement the security mechanisms.

2.1 Computer Architecture Fundamentals

This section will present some fundamental concepts within computer architecture. This fundamental knowledge gives insight into the mechanisms employed in processor cores that makes them vulnerable to transient execution attacks. This knowledge is necessary to be aware of to understand how the attacks described in Section 2.2 exploits the hardware, why it is hard to mitigate them, and why the mitigations affect the performance of the processor core.

2.1.1 Processor Core Pipeline

In most processor cores, instructions are processed in a pipeline. That means that the cores are implemented with several pipeline stages. When an instruction is processed, it goes through the stages one by one before its execution is completed [6]. The classic *Reduced Instruction Set Computer (RISC)* pipeline has 5 stages, which are shown in Figure 2.1. The stages are instruction fetch, instruction decode, execution, memory access, and write-back. Although most processor

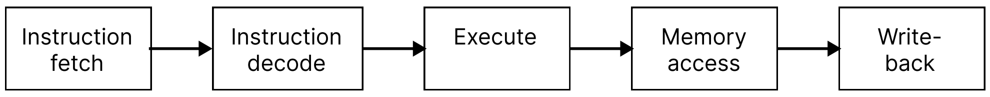


Figure 2.1: RISC five-stage pipeline

cores are more advanced than this pipeline, it is a good example to understand the fundamental concept of how instructions are processed. This pipeline will be used as a reference when describing optimizations later in this chapter.

In one clock cycle, each stage gets some input, processes the input, and writes the output to the next stage. Between each stage in the pipeline, there are registers that hold the output from the previous stage, and which also is the input in the next stage. For example, in Figure 2.1, the output from the instruction decode stage, is written to the registers between the instruction decode and the execute stage. In the next cycle, the execution stage gets its input from the same register.

By processing instructions in a pipeline with several stages, like the RISC pipeline, one exploits *instruction level parallelism (ILP)*. Instead of having only one instruction processed at a time, one can have one instruction in each pipeline stage concurrently. By having several instructions in the pipeline at a time, each at a different stage, one can say that the instructions run in parallel.

Pipelining is a cheap optimization that can improve performance by several factors. If the pipeline for example has five stages, the performance increases five times with pipelining compared to executing one instruction at a time, if there are no dependencies between the instructions.

2.1.2 Super-Scalar Processors

Super-scalar pipelines exploit ILP to gain performance. In super-scalar pipelines, each stage has extended the micro-architecture to be able to handle several instructions each cycle. This optimization of the pipeline enables the processor cores to achieve IPCs higher than 1 [6]. The amount of instructions each step can handle is called the *issue width*. Figure 2.2 shows the first 6 cycles of a program with 12 instructions running on a processor core with an issue width of two. As the figure shows, all twelve instructions have been issued after six cycles, doubling the throughput, compared to what the same program would have taken on a processor core with the reference single-issue RISC pipeline. It also gives an IPC of 1.2, on the condition that instruction 11 and 12 only use one cycle on their last four stages.

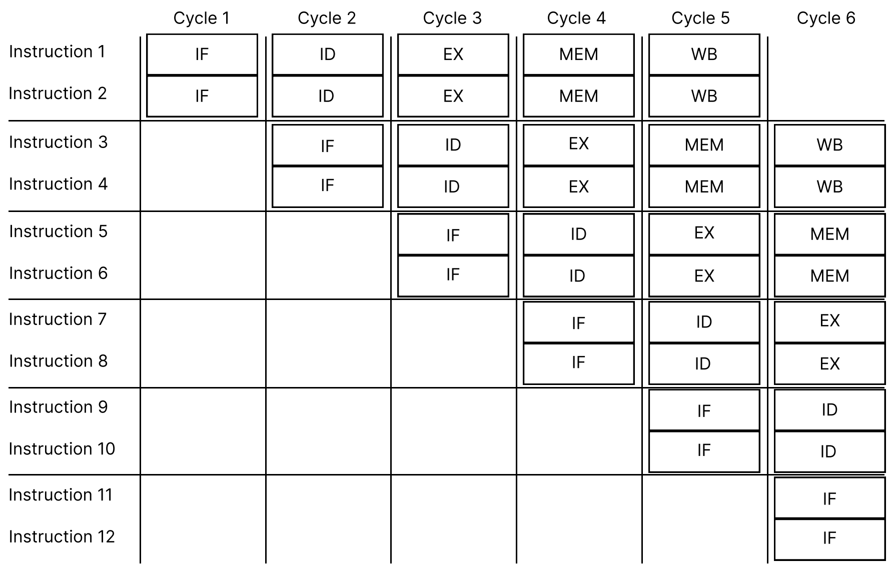


Figure 2.2: 2-wide super-scalar pipeline

2.1.3 Out-of-Order Execution

Out-of-order execution improves the ILP, and thus the processor cores' performance, compared to the in-order RISC pipeline which is used as a reference. Figure 2.3 shows a simple example of an implementation of a pipeline with out-of-order execution, similar to the RISC pipeline. The sketch is just an example of how an out-of-order pipeline can be implemented, and only tries to illustrate the most common structures and concepts employed to support out-of-order execution.

A reason why executing instructions out-of-order may be useful is because sometimes the execution time on certain execution units may vary and cause the pipeline to stop fetching instructions. If one instruction stops at a stage in the pipeline, it blocks younger instructions from moving further in the pipeline, despite the younger instructions being ready to move to the next stage [6]. If for example a load is issued and misses in the cache, it may take a lot more cycles to execute it. The data have to be loaded from lower levels in the memory hierarchy, which are further away from the core, thus giving larger latencies. In a pipeline executing all instructions in-order like the RISC pipeline, all younger instructions have to wait before continuing to the next stage until the load has finished the memory access. In that case, fetching new instructions also stops.

The problem with instructions making the pipeline stall can be solved by allowing instructions to be executed in a different order than the correct program order in part of the pipeline. An out-of-order processor core allows younger instructions to bypass the stalling instructions if they are ready to execute. The problem with variation in execution time almost always occurs at the stages where the instructions' operation is done. In the RISC example, that would be in the execute and memory access stages. Fetching, decoding, and writing the result back to the output register is the same for every instruction. Because of this, in out-of-order cores, these stages are still done in-order, and only the stages where difference in execution times occurs are done out of order, as shown in Figure 2.3. It is actually also necessary for getting the correct result to have fetching and write-back done in order [6]. While out-of-order execution succeeds in keeping the processor busy with more work, the benefits from this are limited unless the new work is relevant to program execution. Out-of-order processors, therefore, need mechanisms to handle data dependencies and also make sure that the instructions commit in program order.

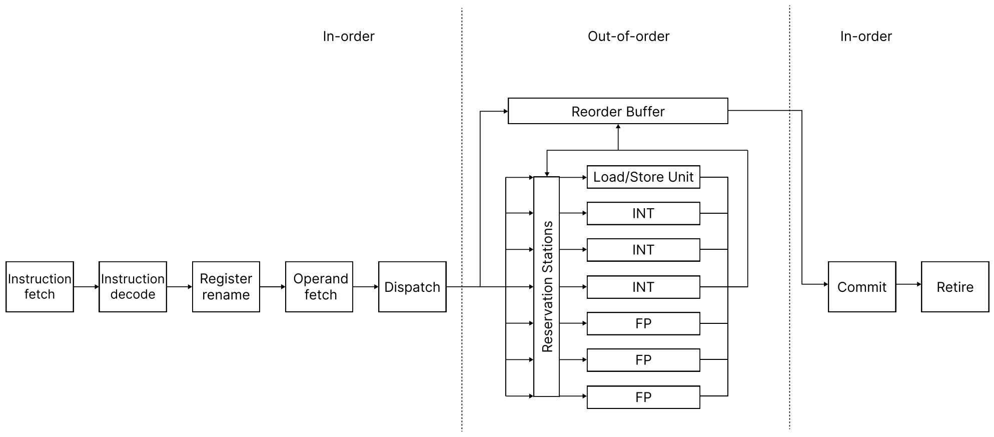


Figure 2.3: Out-of-order pipeline

To maintain correctness when executing out of order, some new hardware structures have to be introduced. The pipeline is additionally extended with some extra stages. These changes are shown in Figure 2.3. The new stages in the example pipeline are register renaming, operand fetch, and dispatch, at the frontend. At the backend of the pipeline, the write-back stage in the RISC pipeline is replaced by the two stages complete and retire. The two most important structures are the reorder buffer (ROB) and a reservation station [7] for each functional unit. The ROB is used to make sure the instructions commit in order, and the reservation stations prevent the instructions from executing unless its operands are ready.

Although the frontend of the pipeline is executed in-order and has some of the same stages as the RISC pipeline, it is still a little different. Register renaming maps logical registers with physical registers. The logical registers are available to the programmer or compiler, while the physical is hidden. It is necessary that there are more physical registers than logical ones. The purpose of having both logical and physical registers is to remove output and anti-dependencies [6]. Output

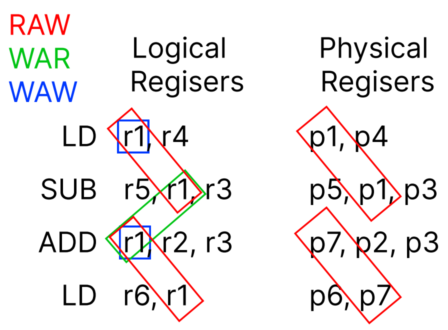


Figure 2.4: Hazardous program renamed to remove output and anti-depenencies

dependencies are instructions writing to the same register, and anti-dependencies are write-after-read dependencies. These dependencies are not a problem in in-order pipelines, since all execution is in program order. In an out-of-order pipeline, on the other hand, instructions may overwrite each other registers, and cause an incorrect result.

Figure 2.4 is an example of how register renaming can remove anti-dependencies and output dependencies. The first instruction is a load and writes its output to the logical register `r1`. A load can potentially miss in the cache, and it may take many clock cycles before the value is loaded to register `r1`. The second and third are both arithmetic instructions, which are often executed in very few cycles. The last instruction is also a load with a real data dependency with the third instruction. If the first load misses in the cache, and the add instruction having an output dependency on the load bypasses it, the add instruction will write its result to the output register before the load has loaded the value. That will lead to the load overwriting the result from the add when it is finished. The same would happen with the anti-dependency between instruction two and three if the third instruction bypasses the second.

The right side of the figure shows that the anti-dependency and output dependency have been removed by assigning another physical register to instruction three and four. The example also demonstrates that the real data dependency is remaining.

When the registers rename is finished and the logical registers have been mapped to physical registers, the input operands can be fetched from the physical registers in the operand fetch stage. In the dispatch stage, information about incoming instructions is written into an entry in the ROB and as an entry in the reservation station of the functional unit the instruction is going to execute on. Both the ROB and an execution unit's reservation station are structured as a queue, with each entry holding information about a dispatched instruction.

To maintain program order, entries in the ROB are retired in a FIFO order. The oldest instruction is at the head of the ROB ready to commit, and no younger instruction is allowed to commit before the oldest have been retired. Each entry in the reservation station also contains information about its register dependencies and prevents the instructions from executing unless their operands are ready.

2.1.4 Speculative Execution and Branch Prediction

Speculative execution is one optimization architects have employed in processors to keep the pipeline busy. Speculation involves fetching instructions by taking an educated guess on what instruction address is correct to fetch, while speculative execution is the execution of these speculatively fetched instructions. Speculation is needed when the fetch unit doesn't know the correct future instruction stream of a program. A common example of when speculation is needed is on conditional branches. Branch instructions are created, for example, when there is an *if-sentence* in a program. These instructions have a condition deciding whether the fetch unit should start to fetch instructions from the branch target or continue to increase the program counter. It may take a lot of cycles before the outcome of the condition in a conditional branch is known. Instead of waiting until it is known, wasting cycles, predictors speculate on which instructions should be executed and fetch instructions instead of making the pipeline stall. If the speculation turns out to be wrong, the pipeline rolls back to where the speculative fetching of instructions started.

As mentioned in Section 2.1.1, some instruction may cause other instructions to stall because it has a long execution, forcing the younger instructions to wait, but another reason the pipeline may stall is because the instruction fetch unit may not know where to continue fetching instructions. This happens for example at conditional branches, where the branch instruction has to wait for its dependencies to resolve before the outcome of the branch is known. To overcome this, the pipeline speculates where the execution is going to continue. The speculation is however not random. A mechanism called branch predictor and branch target buffer is implemented to take an educated decision on where the execution will continue. The branch predictor speculates on both which direction the branch takes, and also at which address to continue on if the branch is taken, also called the branch target.

There are various different methods used for speculatively deciding if a branch is taken or not taken. Two types of branch predictors exist; static and dynamic. Static predictors always predict the same outcome each time a specific branch is predicted, and the outcome is decided at compile-time based on profile information from previous runs. Dynamic predictors, on the other hand, learn the patterns during run-time from the previously executed branches. Static predictors are easier to implement and require less hardware structures than dynamic predictors, but in return, they perform worse than dynamic predictors. Dynamic predictors are the most used in modern processors. These can get very good accuracy, and many use both global branch history and the history local to a specific branch to make a decision on taking the branch or not.

To continue execution at the correct instruction, in addition to the direction of the branch, the fetch unit also needs to know the target instruction's address. Most processors use a structure called the branch target buffer (BTB) [8]. The BTB works as a cache for branch targets, where the program counter (PC) of the branch is used as a tag. The BTB entry with a matching tag stores information about the previous target address for that branch instruction.

As mentioned, speculation may sometimes lead to execution of instructions that turns out to be fetched from the wrong destination. If the speculation turns out to be incorrect when the branch resolves, the traces of the instructions executed speculatively have to be removed. Since the ROB makes the instructions commit in program order, none of the speculatively executed instructions are committed before the branch is resolved. The pipeline gets squashed, to remove these traces. Squashing reverts the architectural state of the pipeline back to the state it was in before the speculative execution was performed. Squashing involves removing all younger instructions from the ROB, in addition to setting the register values back to their old values.

2.2 Transient Execution Attacks

Execution of instructions that are incorrectly fetched speculatively, and bound to be squashed, are called *transient execution*. These instructions are executed on their respective functional unit, but never committed [9]. Transient execution is not visible in the architectural state, since the squash reverts the architectural state back to what it was before the transient execution started. However, there may still be traces left in the microarchitectural state. The microarchitectural state involves the state in structures that are not directly visible to the programmer. Microarchitectural state, for example, includes structures like the cache, branch predictor, etc. Since traces of the transient execution are not removed in the microarchitectural state, it may be used as a side channel to give an attacker information about the transient execution [10].

Figure 2.5 shows how the state in some architectural and micro-architectural structures is affected when squashing the pipeline after mispredicting the branch's outcome and executing some transient instructions. Micro-architectural state is the state of the structures which are not visible to the programmer, like the cache or branch predictor. The architectural state is the state of the structures which are visible to the programmer, like for example the registers.

The program in the figure starts at the `main` tag. The `bne` instruction is a branch instruction that branches to `br_target` if the value in register `r1` is not equal to zero. It is not known if the condition is true or not before the load instruction has loaded the value into the register. In the example in the figure, the load misses in all cache levels, and the branch is speculatively taken by the branch predictor. In the branch, a value is loaded into register `r2`. The result from the execution of the transient instructions is visible in the architectural state, by observing the ROB and the register file, and in the microarchitectural state by observing the L1 cache. When the load after the main tag has loaded the value from the main memory, it is known that the branch was mispredicted. The pipeline is flushed to start the execution of the correct instructions. Now, there are no traces left of the speculative execution in the architectural state. The ROB and registers only hold values from the correct execution, and no one can know that speculation ever happened by observing the state of these structures. On the other hand, the microarchitectural state is unaffected by the pipeline squash. The microarchitectural state is, as mentioned, not directly observable to the programmer, so to actually exploit this behavior one needs to find a way to be able to observe the changes in these structures. One way of observing the transient execution can, for example, be to use the secret value to index an array in the transient execution. The secret value can later be observed by accessing each element in the array and timing the accesses to know which of the array indexes hit in the cache. Only the index is equal to the secret value should hit in the cache, given that none of the array elements was already present in the cache. This example has demonstrated how the architectural and micro-architectural state are affected by speculation. How to be able to observe the micro-architectural state to read secret data will be further explained later in this chapter.

Most of the transient execution attacks follow the same five phases. The first phase is to prepare the microarchitecture to behave a certain way. This can for example be to train the branch predictor to speculate in a certain way. The second is to execute a trigger instruction, which triggers the transient execution of instructions. This would be the execution of the branch instruction in the example in Figure 2.5 The third is the transient execution of instructions loading unauthorized data into a microarchitectural side channel. In the example, that is done with the load instruction in the branch. The fourth is when the trigger instruction retires and the pipeline is squashed. As mentioned, when describing the example in Figure 2.5, the architectural state is reverted to what it was before executing the transient instructions, while the micro-architectural state is left unchanged. The fifth is when the attacker recovers the secret from the microarchitectural side-channel [10]. This step is not demonstrated in the previous example but will be explained when looking at the specific attacks. There are different ways of doing this, depending on the attack performed, but it could for example be to time memory access latencies to observe cache side-channels. All these phases will be explained further with a code example of a Spectre attack in Section 2.2.1.

Although most of the transient execution attacks follow the same five phases, they can be classified into two main categories, with several subcategories. The two main categories are Meltdown [11] and Spectre [9] types. Classification of the attacks is based on the cause of transient execution. Spectre attacks exploit mispre-

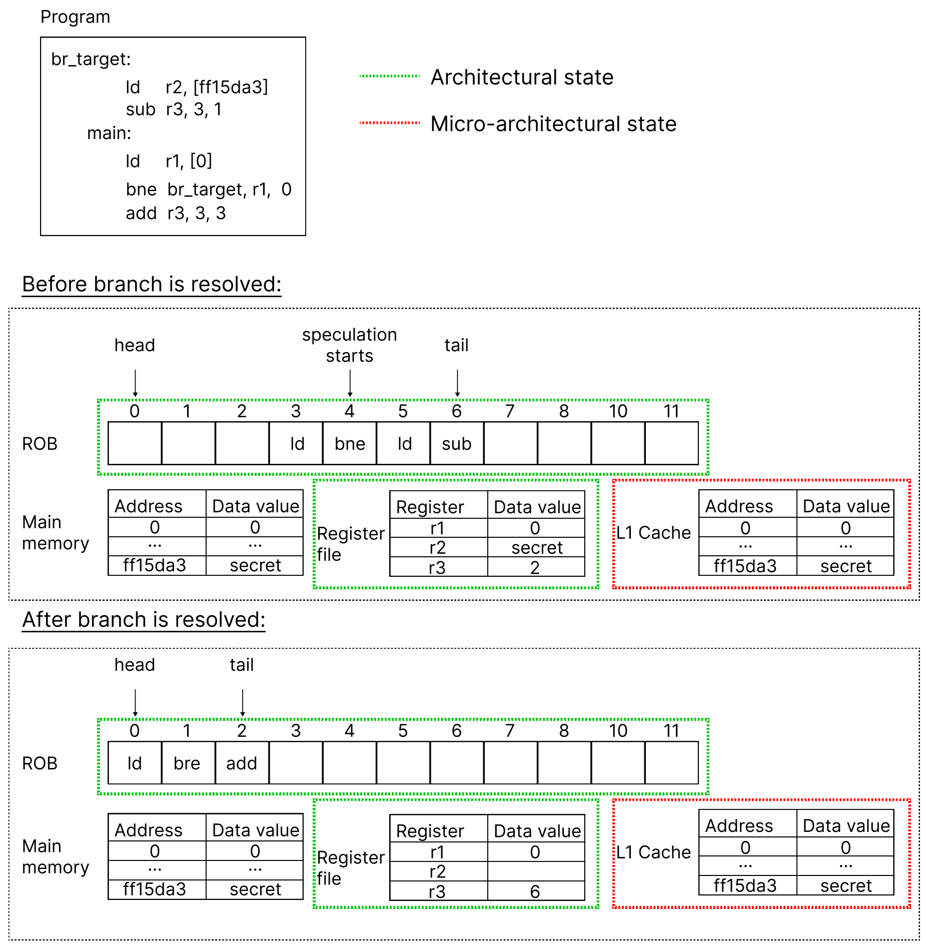


Figure 2.5: Architectural and micro-architectural state right before branch misprediction is resolved, and after the pipeline has been squashed

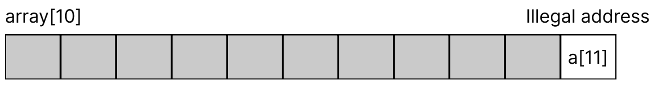


Figure 2.6: Memory layout of an array of 10. Gray elements represent addresses that are allocated to the array, and the white element at the end represents a memory address outside the allocated memory space

diction in control or data flow. Meltdown types exploit transient execution being squashed by a faulting instruction, like for example by an exception [10].

2.2.1 Spectre Variant 1

Spectre Variant 1 exploits conditional branches to leak secrets [9]. Figure 2.6 shows the memory layout of `array` from the pseudocode of the attack in Code listing 2.1. It has ten elements, making the 11th element seemingly inaccessible. This is however only if it is accessed directly. The Spectre attack makes it possible to read this value by exploiting transient execution. The attack can actually be performed many times to read the entire main memory, leaking sensitive information like cryptographic keys.

The phases presented in Section 2.2 can also be found in Code listing 2.1. The enumeration below explains how the five phases are implemented in Spectre Variant 1.

Code listing 2.1: Pseudo code of the Spectre Variant 1 attack

```c
uint8_t array[10];
uint8_t probe[256*4096];
int secret;

uint8_t dummy; // Used to do accesses, but its value is never used.

int acces_array(int index)
{
    if (index < 10)
        dummy = array[index];
}

void main()
{
    // Train branch predictor
    for (int i = 0; i < 1000; i++)
    {
        acces_array(i % 10);
    }

    // Removes all the elements in probe from the cache,
    // to make sure it is empty
    clflush(probe);

    // Do access of illegal index
    uint8_t illegal_access = acces_array(11);
    dummy = probe[illegal_access*4096];

    // Find the illegal access value by finding out which
    // element in the probe array which is in the cache
    for (int i = 0; i < 256*4096; i+=4096)
    {
        int start = time();
        dummy = probe[i];
        int stop = time();

        if ((stop - start) < cache_time_treshold)
            secret = i;
    }
}
```

1. First microarchitectural state is prepared. This is done by training the branch predictor to mispridict the branch when the trigger instruction is executed. The cache is also cleared so that none of the elements in the probe array are present there at the point where the transient execution starts.
2. The trigger instruction is the branch instruction from the `if` statement in the `access_array` function. The trigger instruction ensures that the transient execution will start, since the branch predictor has been trained to take the branch, despite the condition being false.
3. In the transient execution, an access is done to an illegal memory address, which in this case is `array[11]`. This is later in the transient execution used to index the array called `probe`, which is within the legal address space. A

side effect of the transient load of the element in `probe` is that it will be loaded into the cache before the pipeline is squashed.

4. When the trigger instruction resolves, the pipeline squashes, and the architectural state is removed. However, the `probe[array[11]*4096]` is however still present in the cache.
5. The last phase involves retrieving the value from the side-channel. In this attack, this is done by checking which of the elements in `probe` are present in the cache. This is done by making an access to each element and time the latency of the load. Since the probe array has been flushed with `clflush`, the cache does not contain any of the array elements when the attack starts. Therefore, only the element indexed by the secret value should be present at the end of the attack. When the cached element in `probe` is identified, its index reveals the secret to the attacker.

2.2.2 Other Spectre Variants

In addition to Spectre Variant 1, the original Spectre article [9] also introduced two other variants of Spectre attacks. It additionally introduces Variant 2 and Variant 4, in addition to other minor variations. The Spectre Variants presented in the original article all use the cache as its side channel, the difference is how they initiate the transient execution to load the secret into the probe array. It has however later been discovered attacks which also uses other side-channels than the cache.

Variant 2 works similarly to Return-Oriented Programming (ROP), which is a previously discovered attack. ROP tries to manipulate the return stack by performing a stack buffer overflow to overwrite the values already present in the return stack. With this, it can dictate the control flow and decide what instructions should be run. ROP analyzes instructions of, for example, the C standard library or some other code already present in executable memory regions to find suitable instructions to be able to achieve the wanted behavior. ROP then points control flow to these instructions to create a sequence of transient execution. The sequence doesn't necessarily have to be on adjacent addresses, but can be at arbitrary locations of the executable memory regions [12]. Like ROP, Spectre Variant 2 also uses instructions already present in the executable memory regions to create sequences of transient execution. The difference between ROP and Spectre Variant 2 is how it directs the control flow. While ROP performs a stack buffer overflow to overwrite the return stack, Spectre Variant 2 exploits misprediction by the BTB to dictate the control flow. Variant 2 exploits indirect branches. By flushing the cache line containing the indirect branch's target address, one has ensured that speculation starts, and the correct execution will not continue before the branch's target is loaded from main memory.

Spectre Variant 4 [13] exploit an optimization called memory disambiguation. If a load instruction follows a store instruction in some program, one can not be sure if there is a true data dependency if the destination of the store is not known. If it is a dependency, the data from the store should be forwarded to the load, and if not the load should execute as normal. Normally processors would wait with performing the load until it is known whether there is a dependency or not, but with memory disambiguation loads are executed speculatively, and the correct value is chosen when the outcome of the dependency is ready. By exploiting this speculation, one can perform a transient load while waiting for the dependency to resolve, and later retrieve the value through a cache side-channel.

The Meltdown attack doesn't exploit speculation like the Spectre attacks. This attack exploits faulting instructions together with out-of-order execution. As mentioned in the original Meltdown paper [11], the Meltdown attack only works on Intel processors. This is because the attack exploits Intel's exceptions handling. Since all the processors tested are commercial designs, the details about the micro-architecture is disclosed. It is however assumed that the processors from AMD and ARM, which were also tested, handle exceptions differently, and thus are vulnerable to the attack. The attack has also later been patched by Intel, and are now considered a less serious threat than the Spectre attacks.

2.3 Transient Execution Attack Defense Mechanisms

Several measures have been taken to mitigate both these attack types, but since speculation and out-of-order execution is very important to achieve performance in modern processors, complete mitigation is a tradeoff between security and performance. Also, since the attacks exploit hardware mechanisms, they can't be fully mitigated only with software patches. Software strategies can make it harder to perform the attack, but the way speculation and out-of-order execution are done in hardware has to be changed to completely stop it from leaking data.

2.3.1 Software Mitigations

Although software mitigation strategies against Spectre attacks exist, it does not provide the level of security that you get from hardware mitigation strategies. A lot of the software strategies are also tailored to fit a specific attack, and does not provide a general security. To the author's knowledge, there are also no software mitigation strategies providing security against Spectre Variant 1.

Retpoline [14] is a software mitigation strategy introduced by Google to mitigate Spectre Variant 2. Retpoline stands for *Return trampoline*. This strategy replaces indirect branches with new instructions, which makes sure that the speculation is not controlled by the attacker when finding the branch target.

2.3.2 Delay-on-Miss

The Delay-on-Miss (DoM) [1] mitigation strategy delays the execution of loads that miss in the L1 cache until they are no longer speculative. Since loading data from the highest cache level, doesn't load new elements into the cache and thus don't change the microarchitectural state, it is safe to do even speculatively. Loads on lower cache levels and main memory will however change the microarchitectural state by loading the address into higher cache levels, and are thus not safe to issue. To delay speculative loads, the processor needs some way of knowing if a load is speculative or not. That is done by giving the loads executed speculatively a property called a speculative shadow. The shadowed property is assigned to load instructions executed in the shadow of an instruction that has triggered speculation. The shadow is not removed until the instruction causing speculation has been resolved. Loads with the shadowed property will not be issued to levels below the L1 cache, but rather delayed on an L1 cache miss until they are safe to issue. This means that missing load instructions will not be part of the transient execution required to perform a Spectre attack, but rather delayed until the speculation is resolved [1].

There are four different types of shadows [15]; E-shadow, C-shadow, D-shadow, and M-shadow. The E-shadow is cast by instructions that may cause exceptions. The C-shadow is caused by instructions that predict control flow and is cast over instructions speculatively executed. D-shadows are caused by memory dependencies, for example, stores cast shadows over younger instructions depending on the stored value. The M-shadows are only present in systems where the memory consistency models which respect load-load order. If a load is delayed, then younger loads also will have to be delayed to respect the memory model. The most important one for Spectre variant 1 is the C-shadow, since it exploits control flow speculation.

An improvement of this technique also provides value prediction on L1 cache misses. The value prediction logic tries to predict the value of the requested data, and uses this prediction in the computations performed, without loading data into the cache. DoM is estimated, by its inventors, to have a slowdown of only 11% with value prediction, and 19% performance loss without value prediction [1].

2.3.3 Speculative Taint Tracking

Speculative Taint Tracking (STT) [2] is a mitigation strategy to mitigate transient execution attacks. STT is claimed by its authors to have a slowdown of only 8.5%. Like DoM, STT is implemented in the microarchitecture, making it a hardware mitigation strategy. Some important terms to understand Speculative Taint Tracking are *access instructions* and *transmit instructions*. These terms are used to classify instructions that have certain attributes which can be exploited to do a transient execution attack. An instruction is classified as an access instruction if the instruction is executed speculatively and therefore may depend on unauthorized secret data. In the example in Code listing 2.1 the access instruction would be the instruction created by the code in line 26. Transmit instruction is the instructions transmitting the unauthorized accessed, potentially secret, data from the access instructions through a side channel. In Code listing 2.1 the transmit instruction would be the instruction generated from the code in line 27.

The inventors of STT have found that accessing the secret data speculatively is not the dangerous part of a speculative side-channel attack. The data is not visible to the attacker unless it is transmitted through a side channel. STT tries to improve the performance over previous strategies by executing the speculative access instruction and rather delay the transmit instruction depending on the access instruction until the access instruction is safe. This is different from Delay-on-Miss which is explained in Section 2.3.2, which delays the access instruction. STT is therefore less strict than DoM and should in theory give a greater throughput of instructions compared to DoM. By allowing the access instruction to execute, one can continue the execution of dependent instructions which are not transmit instructions. It is for example safe to execute arithmetic instructions like `add` since the execution of this instruction does not leak data by manipulating the microarchitectural state. The output register of the add however has to be tainted if any of the input registers are tainted. Even though the add instruction does not leak data, an instruction depending on the result from the add could be used to leak the data from the access instruction.

To know if a transmit instruction depends on an access instruction, the registers written to by an access instruction are *tainted*. The taint is propagated to the output registers of instructions having tainted input registers. Loads may be used to transmit secrets to the cache, and therefore if a load has a tainted register as its input register, one knows that this depends on a value obtained by speculative execution. Because of this, the instruction is classified as a transmit instruction and is delayed until the access instruction is no longer speculative. The visibility point of an access instruction is when it goes from being speculative to non-speculative. When an access instruction is past the visibility points its registers get untainted, and the untainting also has to be propagated to the registers which have been tainted by younger instructions depending on the access instruction. This also untaints the input registers of potential transmit instructions. When a transmit instruction gets its input register untainted, it is no longer delayed and can be executed.

2.3.4 Other Hardware Defence Mechanisms

InvisiSpec [16] is a mitigation strategy reading speculative data into a buffer, instead of loading it into the cache. The authors claim to have a slowdown of 21%. Another defense mechanism is InvarSpec [17]. This is intended to be used to get better performance out of other strategies like InvisiSpec or DoM, by identifying if it is safe to execute loads before the speculation has resolved. This technique has, however, later shown to be insecure [18].

2.4 Tools

Several resources, frameworks, and other tools have been used when designing, simulating, and evaluating the speculative side-channel mitigation strategies Delay-on-Miss and Speculative Taint Tracking. Many of these tools are provided by the Electrical Engineering and Computer Science (EECS) department at the University of California, Berkeley. This section will present the technical background of the tools used for this project.

2.4.1 FPGA

A Field Programmable Gate Array (FPGA) is a circuit that can be configured to model hardware specified in a design. FPGAs are re-programmable, and the circuit is configured by a hardware description language (HDL), like for example Chisel or Verilog. If the hardware designer wishes to do changes in the design, the designer can simply change the implementation in the HDL, and then create a new circuit on the same hardware device from the new specification. This makes FPGAs well-suited for prototyping since the designs are tested on something which is closer to what the finished physical realization of the design would be, rather than what software simulations can provide. The process of creating something that can run on the FPGA is called synthesis. This involves transforming the HDL program into something called register transfer level (RTL). This can be compared to a computer program being compiled into an assembly program. The building blocks of FPGAs are configurable logic blocks (CLBs). The CLB also consists of smaller components like a lookup table (LUT) and a flip-flop.

2.4.2 Chipyard

Chipyard [19] is a framework that integrates various parts of hardware design. It provides various open-source designs like the BOOM [3] and Rocket [20] cores. It also provides simulation and evaluation tools like Verilator [21] and Firesim [22]. The framework, in other words, provides several independently developed building blocks and puts them together to build and simulate a complete SoC design. Chipyard is an open-source project started by researchers at Berkeley. That is also the case for BOOM, Rocket, and Firesim.

2.4.3 Firesim

Firesim [22] is a hardware simulation platform for evaluating hardware designs on FPGAs. Firesim enables hardware designers to create a bitstream of their design which can be flashed to the FPGA. This is also possible to do with other tools like Vivado, but this often requires a lot of manual configuration. Firesim works as an abstraction layer, simplifying the process of building and deploying FPGA simulations. The platform is intended to run on Amazon EC2 F1 in the public cloud. It is also intended to be used to simulate entire clusters, but can also be used to simulate single processor cores on private FPGA cards as well.

Firesim uses the terms target and host to differentiate between the simulated core and the hardware used to run the simulation. The target platform refers to the simulated environment. If an implementation of the BOOM core would be simulated on an FPGA, the BOOM core would be the target platform. The host platform is the physical CPU needed to execute the Firesim simulation, in addition to the FPGAs.

Firemarshal [23] is a tool that can be used together with Firesim to build workloads that can run on the FPGA. Running workloads on the FPGA simulation can be useful to, for example, testing the design's performance by running benchmarks. Out-of-the-box, Firemarshal offers some basic workloads which the user can either build and run on their FPGA simulation or use as a base to create their own custom workloads. These workloads include, for example, *Build-Root Linux* [24] and *Fedora Linux* [25]. Custom workloads are configured with a configuration file, which has to be either `JSON` or `YAML`. In this file, the base workload is specified, in addition to the startup scripts that should run. It can also be included a path to a folder with files that should be copied into the root file system on the FPGA simulation.

2.5 Boom Core

The Berkeley Out-of-Order Machine (BOOM) [3] is an out-of-order processor core. It is an open-source project initially created at the University of California, Berkeley [26]. The core is developed to be a platform for research in high-performance core design. According to its main contributors, it was the fastest available open-source core at the time the last version came out, measured in IPC. At the time this report is written, there are three releases of BOOM. The second version has also been physically realized [27]. All versions are written in the hardware description language *Chisel* [28], and the core is integrated with the chipyard [19] framework for hardware simulation, which is also developed by Berkeley.

BOOM is designed to execute instructions from the RISC-V [29] instruction set architecture. This ISA was also developed at Berkeley and is an attempt to provide a free and open ISA. The fact that it is free makes it suitable for academic purposes, since proprietary ISAs require licenses which can cost several millions of dollars. RISC is an acronym for *Redused Instruction Set Computer*, which is also discussed in Section 2.1. A feature of this type of instruction sets are that they have a smaller amount of instructions, and the instructions are also less complex, compared to the *Complex instruction set computers* (CISC) which may be seen as the opposite alternative. An advantage of RISC ISAs is that due to their low complexity instruc-

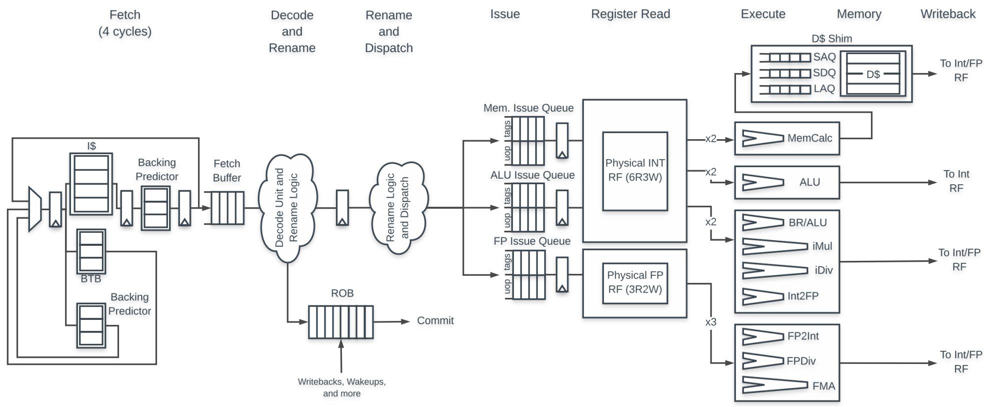


Figure 2.7: Overview of the BOOM pipeline [30]

tions, the design of the processor core also needs to be less complex to support all the instructions in the ISA.

Figure 2.7 shows an overview of the stages and components in the BOOM pipeline. The figure is acquired from the BOOM documentation [30], and published by Abraham Gonzalez, one of the PhD students at Berkeley. The pipeline has 10 stages, which are Fetch, Decode, Register Rename, Dispatch, Issue, Register Read, Execute, Memory, Writeback and Commit. Many of these stages are similar to what have already been presented in the example out-of-order core in Section 2.1.

In addition to be out of order, BOOM can also be configured to be super-scalar. In the BOOM documentation, it is stated the BOOM is a “parameterizeable core”. What this refers to is that several attributes of the core are decided by parameters. This means that it is easy to make different configurations of the core by changing these parameters. Some of these parameters are, for example, `fetchWidth` and `decodeWidth`. As presented in Section 2.1.2, a condition for a pipeline to be super-scalar is that it is able to issue more than one instruction at a time. Since the issue width is decided by the parameter `fetchWidth`, the core can be configured to be either only issue a single instruction per clock cycle or be able to issue multiple, depending on what the hardware designer wants.

In the front-end, the instructions are first fetched from the instruction memory. In the next stage, they are decoded, which involves transforming them into micro-ops. Programmatically, a micro-op is an instance of the `Micro-Op` chisel class, which contains all the necessary information relevant to that micro-op as variables. Once the instruction is decoded, the micro-ops generated get allocated an entry in the ROB. In the register renaming stage, the decoded micro-ops have their logical registers mapped to physical register to get their dependencies resolved. Lastly, the micro-ops when the micro-ops reach the dispatch stage, which means that they are ready to execute on the functional unit. In the dispatch stage, the micro-ops are therefore written into the reservation station.

The structures required to perform branch prediction also remain in the front-end of the pipeline. BOOM implements two levels of branch predictors: a *Next-Line-Predictor* (NLP) and a *Backing Predictor* (BPD). The NLP is a combination of a *Branch Target Buffer* (BTB), *Bi-Modal Table* (BIM), and *Return Address Stack* (RAS). These structure predicts the branch target if the incoming branch instructions. The BPD makes prediction on whether branch instructions are taken, or not taken.

As mentioned, dispatched micro-ops have already been assigned an entry in the ROB. The micro-ops are also given a branch-mask. This is a mask over the unexecuted branches. BOOM has a maximum number of allowed branch instructions in the pipeline concurrently. This is decided by a parameter in the configuration and also sets the width of the branch-masks. Each bit in the mask represents one older branch micro-op, and it is set to 1 if the branch is unresolved and 0 if the branch is resolved, or the slot is unused. This mask can be used as to know which branched the current micro-op speculate under. This mask becomes useful when a mispredict occurs, and the micro-ops depending on the mispredicted branch should be flushed.

After the micro-ops have been dispatched, they wait in the queue in the reservation station, or issue queue as it is called in BOOM, at their respective functional unit. For load and store micro-ops, the address is first calculated on the ALU, and then the micro-op is put in the load queue or store queue in the *Load-Store Unit* (LSU). The LSU resides in the memory stage. This is also where the actual memory operation happens. The *Translation Lookaside Buffer* (TLB) also resides in the LSU. The TLB is a cache for translation of virtual memory addresses to physical memory addresses. If an address has its address translated, a bit in the load queue and store queue entry is set.

The load queue and store queue are designed as circular buffers. They both have pointers pointing to the head and the tail of the queue. The queues always commit at the head, and add new entries to the tail. The head is the oldest entry, and when it leaves the queue, the pointer increases with one. The tail works similarly, only that it points to the next available entry in the queue, and increases every time an in-flight memory micro-op arrives. That means that at any point in time, all the valid entries in the queue reside between the head and the tail. Another feature of circular buffers is that they wrap once the end of the buffer is reached. Wrapping involves starting at the beginning of the buffer. Figure 2.8 is an example of a circular buffer. When the tail is on the last index, and increases by one, it gets index 0 again and start increasing from there.

In the memory system, only the L1 cache is integrated in the BOOM core. The lower levels in the memory hierarchy are maintained outside BOOM. If a load is issued to the cache, and the cache for some reason is not able to issue the requests to lower memory hierarchy levels, a *not acknowledged* (NACK) signal is returned to the LSU. This signal informs the LSU that the memory request was not accepted,

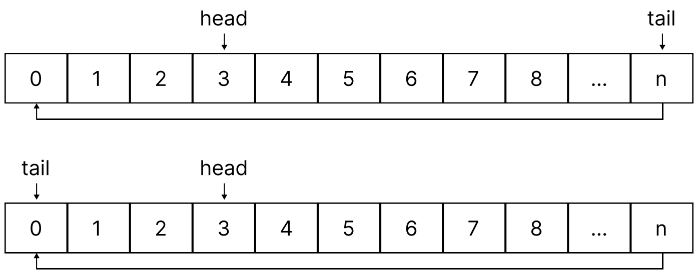


Figure 2.8: A circular buffer before and after the tail is increased by one when it is at the end of the buffer

and have to be re-issued later. The wakeup logic ensures that loads returning in a NACK signal gets issued again later. Not acknowledged loads sleeps for three cycles, before they is waken-up again. Each cycle, the oldest woken up load is selected as a candidate to be re-issued.

As mentioned in Section 2.1, it is necessary for correctness to commit and retire instructions in order. The ROB is designed similar to the load and store queue, as a circular buffer. Also here, the head is the oldest micro-op in the pipeline, and the tail is where new in-flights micro-ops arrive. Since the instructions are allocated in the ROB when the pipeline is still in-order, the ROB will always commit instructions in order.

The pipeline stages are for the most part described in separate modules and implemented in the file `core.scala`. To communicate, chisel uses bundles to describe the wires used for communication between the connected components. The bundles contain the signals sent between the components, as variables.

Chapter 3

Implementation

The modifications done to the BOOM core made for this project can be found in the GitHub repository master-project [31]. The Delay-on-Miss (DoM) implementation is localized in the branch *dom* and the STT design is in the branch *stt*. This chapter will present and explain the design choices and the outcome of these will be discussed in Chapter 6.

3.1 Delay On Miss

Two versions of DoM are implemented for this project. They are mostly similar, but one of the versions try to optimize DoM by restricting the wakeup policy to not check the L1 cache until the load has lost its shadow. The implementation of DoM with the restricted wakeup policy is referred to as *Delay-on-Miss—Restricted Wakeups* (DoM-RW) in this thesis. The RW optimization implemented will be described in more detail later in this chapter.

The modules modified to implement DoM are the Load Store Unit (LSU) and the L1 cache. There are three main steps the loads go through to secure the processor core in this implementation. These are explained in more detail in separate subchapters, but will be introduced with the illustration in Figure 3.1. The paragraphs below will refer to the numbers in the figure to explain the figure step by step, starting at one and counting upwards.

The loads are first inserted into the load queue in the LSU (1). When the load arrives, it is checked if it is speculative. Each entry in the load queue has a bit called `shadowed`, which is set if the load is speculative (2). When the load is ready to be issued, a memory request is may be issued (3).

In the cache, a cache lookup is done. If the address is present in the L1 cache, it is safe to load the value despite it having a shadow (4). The cache may not be used as a side channel if the address is already present. If the requested address is not present in the cache and the load also has a shadow, it is not safe to continue issuing the load (5). The load's micro-op may be squashed at a later point and leave changes in the microarchitectural state.

To make sure that the load is not issued, the cache returns a *not acknowledged* (NACK) signal to the LSU (6). When the LSU receives the NACK signal, it knows that the load have to be re-issued to the cache at some later point (7). At that point, it hopefully has lost its shadow, and are thus no longer speculative.

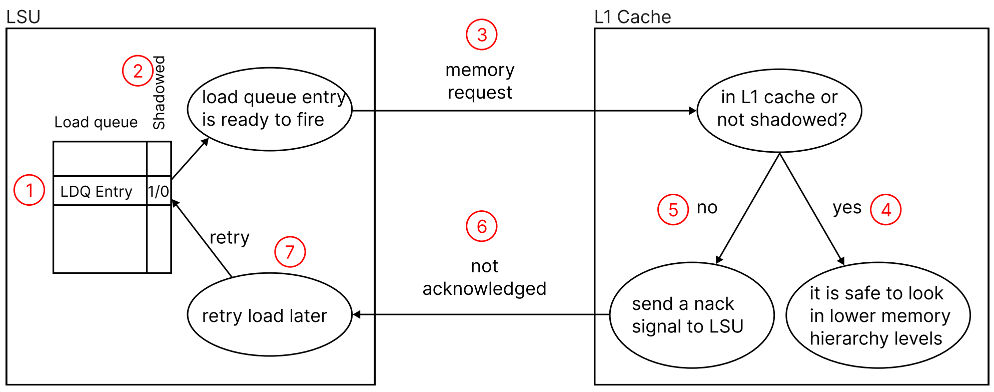

LSU
L1 Cache
Load queue
Shadowed
LDQ Entry
1/0
load queue entry is ready to fire
retry
retry load later
memory request
not acknowledged
in L1 cache or not shadowed?
no
yes
send a nack signal to LSU
it is safe to look in lower memory hierarchy levels
1
2
3
4
5
6
7

Figure 3.1: Flow of the DoM implementation

3.1.1 Detecting Speculative Loads

The first step is to mark the loads as shadowed. With every clock tick, the implementation checks every element in the load queue and marks the elements as either shadowed or not shadowed. The entries can either have a data shadow (D-shadow) or a control shadow (C-shadow). If it has any of the two, the shadowed-bit is set to true. Revisiting the attacks from Section 2.2, the C-shadows protect from Spectre Variant 1 and Variant 2, and D-shadows protect against Variant 4.

The load has a C-shadow if the execution of the instruction depends on a speculated branch instruction, which not yet has been resolved. The `MicroOp` class, has a field called `br_mask`. This is a mask over all the entries in the ROB. A bit is set to one if the ROB entry is an unresolved branch instruction. This makes it easy to detect C-shadows by simply looking at the micro-op of each element in the load queue and checking whether any of the bits in the branch mask is set. If it is, the micro-op depends on an unresolved branch, which casts a C-shadow over that instruction.

A load has a D-shadow if an older store has an unresolved address. Information about two things is needed to decide if a load has a D-shadow. That is, which elements in the store queue are older than the load, and also which of the store queue entries have not yet calculated their address. Two bit-masks over the store queue is used to track the information about older stores and stores with unresolved addresses. The baseline BOOM implementation already has a field in each load queue entry called `st_dep_mask` specifying which of the entries in the store queue that are older than that specific load. The modified implementation also has a mask that keeps track of stores with unresolved addresses implemented. There is already a field in each store queue entry indicating if the entry's address is valid or not. The chisel `map()` function [32] is used to read the valid-value from each entry in the store queue and set the value on the corresponding bit in the bit-mask based on that value. To check for D-shadows, the mask for unresolved stores is bitwise ANDed with the store dependency mask, which keeps track of which of the entries in the store queue that are older. The result of the AND operation is a new mask over load queue. If any of the bits in the result after the AND operation is set, it means that there exists a store entry that is both older than the load and has not yet calculated its address. That implies that there is a D-shadow.

3.1.2 Sending Memory Request

The second step in the algorithm comes when the loads in the load queue are issued with a memory request. The cache module needs to have information about whether a load is shadowed to know if it should return a acknowledged (NACK) signal or not. A shadowed field is therefore added is to the `BoomDCache-Req` bundle, which holds the information about wires connecting the LSU with the cache. Since information about shoadows are already present in the load queue entry, a load that becomes issued can drive the field in the `BoomDCacheReq` bundle from this.

3.1.3 Check if Memory Request Can Be Issued

If a speculative load is issued to the cache and not present in the cache, the cache module should return a not acknowledged (NACK) signal to the LSU. In the cache, there is already code present for returning a NACK signal for other cases like, for example, if the MSHR is full. The information needed to know that the request result in a NACK because it is speculative is; if the request is not a cache hit, and the request is shadowed. If both of these conditions are true, a NACK signal should be returned to the LSU.

Between the LSU and the L1 cache, there are several memory channels, all of which can have shadowed requests coming in to the cache. To keep track of the channels, a mask is used to track which of the channels have requests that are shadowed. This is done with the chisel `map()` function, similar to how the mask over unresolved store is created. A mask indicating if a channel has a request which hits in the cache already exist in the baseline implementation, since it is needed for the other cases where requests could result in a NACK. To know which channels have memory requests where the two conditions are met, a new mask over the memory channels is created. In the new mask, the channels with requests which should return NACK signals because of speculation have the bit in this mask set. The mask is created by performing a bitwise AND of the two other masks. If a memory requests results in a NACK signal because of speculation, no `MSHR` are allocated.

When the LSU receives the NACK signal, the load continues to reside in the load queue in the LSU and are later issued again. The LSU have wakeup logic that delays the load for three cycles, and then makes it available again to be issued. In the normal DoM implementation, the woken-up load can be re-issued independently of whether it still has a shadow or not.

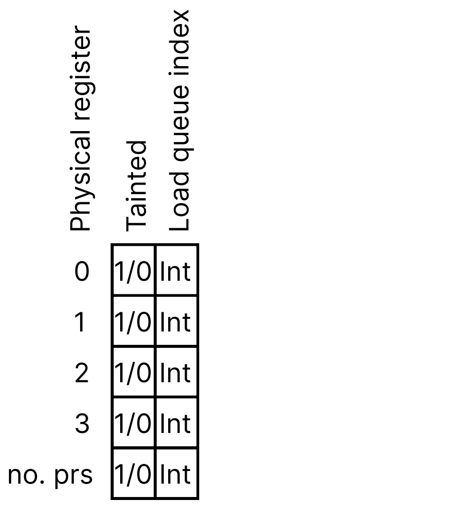
3.1.4 Restricting Wakeups

One hypothesis for this project is that DoM can perform better if it restricts the wakeup policy after a NACK has been returned for a load. The wakeup logic for waking up loads which previously have been not acknowledged by the cache is explained in Section 2.5. If the wakeup candidate still has a shadow after it has been wakened up, it is in the DoM-RW implementation, not issued to the L1 cache. The hypothesis made before performing any experiments on any of the implementations is that it is not likely that the load's address is present in the cache at the time a load is wakened up. The reason the load resulted in a NACK signal in the first place is that the load was not in the cache when it first was issued. It is therefore assumed that it is not likely that it have arrived there in the meantime when the load has been sleeping.

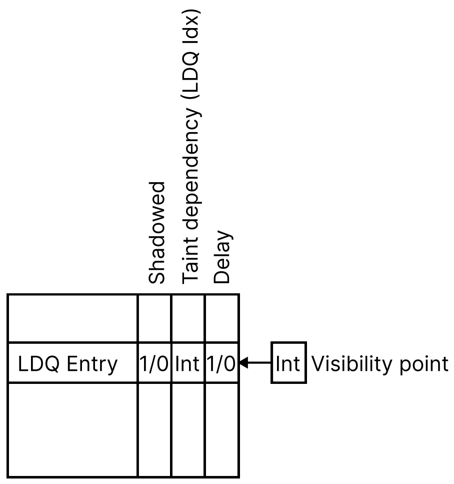
If the hypothesis is true, one could also assume that the performance would increase since less unnecessary cache lookups are performed. One reason for assuming that less cache lookups increases the performance is because when less NACK signals are returned, less loads are put to sleep. For example, if a load is issued with a shadow-bit in the memory request, the load would return with a NACK signal and sleep for three cycles if it is not in the cache. If the shadow is lost in the cycle after the memory request is issued, an unshadowed load is unnecessarily put to sleep for three cycles. It can also take several cycles before the load is issued again after it has been wakened up. This can potentially lead to some loads being delayed for several cycles more than necessary. What DoM-RW would do different is that it would wait until the shadow is lost before load is issued again. With this, one can ensure that the load is not sleeping when the shadow is lost, and the load is ready to be issued right away. If a large enough amount of loads lose its shadow when it is sleeping in the normal DoM implementation, the hypothesis is that this may have a measurable impact on the overall performance.

3.2 Speculative Taint Tracking

Some new microarchitectural structures are introduced to implement Speculative Taint Tracking in the BOOM Core. This section will explain these structures, which



(a) Taints and load dependencies are tracked through the taint mask and load queue index map



(b) The load queue is modified with new fields and a visibility point

Figure 3.2: Main overview of the microarchitectural changes needed to implement STT in BOOM

are also illustrated in Figure 3.2. Some structures are independent of the existing microarchitecture, and some are extensions of already existing structures. The microarchitectural changes are mainly in the LSU.

One of the key features of STT is to track taints on physical registers. To track tainted registers, a bit-mask over the physical register file is created. This is illustrated in Figure 3.2a. Entries, where the bit is one, indicate that the register on the same index in the register file is tainted. The ones that are not tainted have the value zero. In addition to the taint mask, a map over the register file is also made to map physical registers with the load queue entry from the access instruction that caused the taint.

The load queue has also been extended with several new fields. These are displayed in Figure 3.2b. A boolean field in the load queue entry is set if the instruction is an access instruction. This is called *shadowed* and operates identically to how loads are marked as shadowed in DoM, which is explained in Section 3.1.1. The fields *taint_dependency*, and *delay* apply to transmit instructions. Taint dependency notes which access instruction, with regard to the index in the load queue, the transmit instruction depends on. The *delay* field states whether the access instruction the transmit instruction depends on is still speculative or not. If it is no longer speculative, the transmit instruction is safe to issue. Otherwise, the transmit instruction should be delayed. In a register outside the load queue, there is also a value called *visibility_point* which is a pointer to the oldest speculative load in the load queue. This is similar to the head and the tail of the load queue. All entries which are younger than the visibility point are known to be non-speculative. The register holds the integer value pointing to an index in the load queue and is shown in Figure 3.2b.

The implementation of Speculative Taint Tracking can be described in four parts with different responsibilities. An instruction first gets classified as an access instruction if it is a load. The output register of this instruction gets tainted. Another part of the pipeline makes sure that the taint gets propagated to registers used by instructions depending on the output from an access instruction. As will be explained later in this report, this part is not yet completed when the thesis is submitted. A third part of the algorithm classifies loads with tainted inputs as transmit instructions and makes sure that these get delayed until the speculative instruction it depends on no longer is speculative. Lastly, when the visibility point goes past an access instruction, it is known to be non-speculative and the registers which have taints springing from that access instruction have to be untainted.

Due to the time limit of the project, the complete STT implementation is not fully finished. A further description of what part of the core is lacking is presented in Section 3.2.2. The most notable limitation is that, since the microarchitectural changes are implemented in the LSU, the taints are not propagated to other instructions than memory instructions.

3.2.1 Check if a Load is an Access Instruction

This STT implementation has a stricter policy for classifying instructions as access instructions than what is actually necessary. In this implementation, the output register of all loads gets tainted. The reason for that is because it is not yet known whether a load is speculative or not at the time micro-ops are classified as access instructions. However, non-speculative loads get their registers untainted as soon as the visibility point has gone past the load in the load queue. The module `MicroOP`, which contains the decoded information about an instruction, has information about whether an instruction is a load or not. The module also has information about which registers the micro-op uses and its load queue index. Once it is detected that an instruction is a load and hence also an access instruction, the entry at the load's output register in the taint mask is set to one. The map over the load queue indexes (root of taint) also gets set to the load queue index of the access instruction. If, for example, a load enters the pipeline. It writes its output to physical register 16 and has a load queue index of 2. In this example, the bit at position 16 in the taint mask would be flipped to 1 and the value at position 16 in the map of load queue indexes would be set to 2.

3.2.2 Propagate Taints

To fully implement STT, one should be able to propagate taint to every incoming instruction. As mentioned, this implementation has some limited functionality compared to what was originally planned. One of the problems has been that, since all different types of instructions can have taint propagated to it, taint propagation is required to be in a part of the pipeline which all instructions go through. And at the same time, it would also have to be late enough in the pipeline, for the shadow to be known. One possible alternative would be to have it in the dispatch stage. The logic for detecting shadows and transmit instructions, in addition to the logic involved with moving the visibility point and the logic for delaying loads, all need to read or write or both to the load queue. These parts, therefore, need to be in the LSU, since that is where the store queue resides. The situation is therefore that the propagation needs to be done outside the LSU, but most of the other parts need to be inside the LSU, and both the propagation and the other parts need to read and write from the same taint mask and load queue index map.

The implementation of STT implemented for this project has taint propagation implemented, but it is located in the LSU. This means that taints only get propagated to other loads. The implementation therefore only blocks direct load-load dependencies. An example of this kind of dependency is shown in Code listing 3.1. If the dependency is a load-add-load dependency, like in Code listing 3.2, the taint is not propagated to the add's output register, register `r3`. The load in line 3 is therefore not delayed, since its input is not tainted. To fully implement STT, the implementation should propagate the taint, so the load in line 3 gets delayed until the load in line 1 is no longer speculative. The rest of this subchapter will be used to discuss how taint propagation could be done correctly.

Code listing 3.1: Load-Load dependency

```
1 LD   r1, r2
2 LD   r3, r1
```

Code listing 3.2: Load-Add-Load Dependency

```
1 LD   r1, r2
2 ADDI r3, r1, #45
3 LD   r4, r3
```

To be able to propagate other instructions than loads, the logic for propagation should be in another module than the LSU, like for example located at the dispatch stage, in the `core.scala` file. Since most of the other parts of STT need to access the load queue, it should be in the LSU. It is possible to send information about the taint mask and load queue index map between the different components in the core. Figure 3.3 shows an out-of-order pipeline communicating the taint mask and load queue index map between the dispatch stage and the LSU. Since both the taint mask and load queue index map, have one entry for each physical register, the number of wires required between the components would have to be two times the number of physical registers. Although sharing information like this between components is possible, it comes with some challenges. One challenge is that it can be difficult to get the timing correct. The information about which register has been tainted from an access instruction needs to be transmitted to the part where the propagation logic resides before the next micro-op after the access

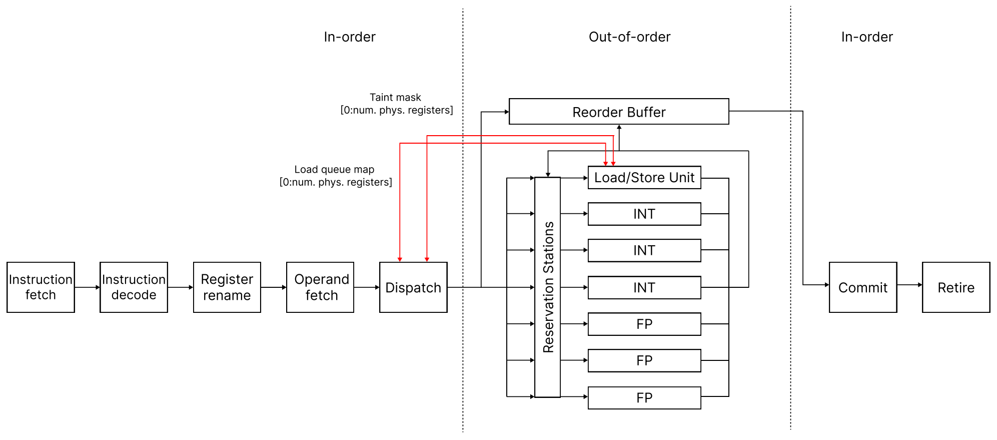


Figure 3.3: Communication between components in a core with taint propagation in the dispatch stage, and all the other taint logic in the LSU

instruction has gone through that stage. The logic for checking if an instruction is a transmit instruction also has to be done after the taint has been propagated, and this would also require transmitting information about the tainted register, from the part where register renaming happens to the LSU. Also, the logic for untainting registers would have to be synchronized.

When a micro-op gets dispatched, its input registers should be used to index the taint mask, coming from the LSU, to see if the bit indicates that the register is tainted. If it is, it means that this instruction depends on an access instruction. On that occasion, the bit in the taint mask at the position of the micro-op's output register should also be set to one. The value in the load queue index map at the position of the output register also has to be set to the same value as the position of the input register, indicating that they are coming from the same access instruction.

In some cases, it may however happen that an instruction has more than one tainted input register. If that is the case, only one of the registers has to be chosen as the root of taint. The rule for this is that the taint coming from the youngest access instruction is chosen as the root. This makes sure that the output register for that instruction gets untainted as late as possible. If the youngest root is not chosen, the register might be untainted too early when the older access instruction has reached its visibility point. In that event, the instructions' destination register would be untainted, even though the input register from the younger access instruction would still be tainted.

The map of physical registers to load queue indexes can be used to find the youngest root of taint. Since the load queue is a circular buffer, it is not as simple as choosing the lowest load queue index as the youngest root of taint. Figure 3.4 shows three different possible states the load queue can have with regard to the head, the youngest root of taint, an older root of taint, and the tail. When the tail have wrapped, but the head have not, the oldest value may have a higher index

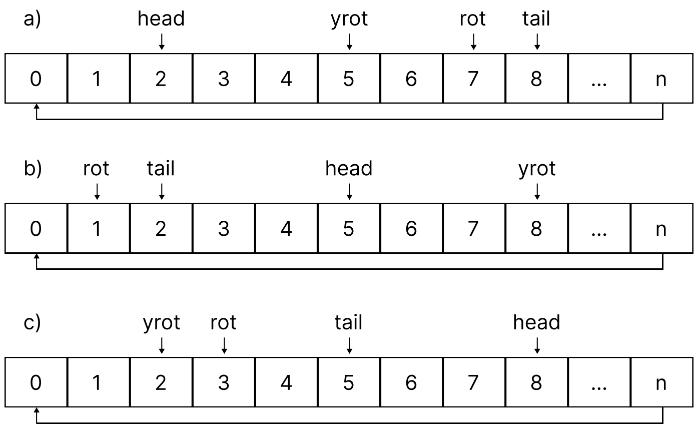


Figure 3.4: Three different states the load queue can be in when taint propagation is calculated

than the youngest, like in Figure 3.4 b). By looking at the figure, it can be found that the suggested algorithm of taking the tainted register pointing to the lowest index in the load queue works for the states shown in a) and c), but would fail to choose the youngest in example b). In this example, the load queue tail has wrapped and is at position 2, and the head is at position 5. An arriving micro-op with its input register tainted by the loads at position 1, and 8 in the load queue, would choose the older taint of 1, instead of 8 which is actually the youngest root of taint. The suggested algorithm of choosing the lowest load queue index has, with this example, proven to be inadequate with the task of choosing the youngest root of taint.

A solution to select the youngest root of taint in a future implementation could therefore be to observe the pointer positions to get select different algorithms based on whether any of the pointers have wrapped. For example, it could be something like in the pseudocode in Code listing 3.3. Here it is first checked if the load queue is in the special case of b). That is done by checking if the tail is less than the head indicating that it has wrapped, and also if any of the tainted inputs are greater than the head, indicating that the oldest root of taint is not necessarily at the lowest index. In that case, the youngest root has to be chosen by finding the lowest index, but still assuring that it is greater than the head. As seen in the code example the logic for choosing the youngest input register larger than the head is not yet implemented as it is not needed for the load-load dependencies which only have one input register. It however, would need to be implemented in the finished implementations.

Code listing 3.3: Calculate the youngest root of taint

```
if ((tail < head) && ((head < prs1) || (head < prs2) || (head < prs3))))
{
    // Find the lowest index which is larger than the head
}
else
{
    yrot := Mux(prs1 < prs2,
                Mux(prs1 < prs3, prs1, prs3),
                Mux(prs2 < prs3, prs2, prs3)
            )
}
```

3.2.3 Check if a Micro-Op is a Transmit Instruction

If a load's input value is tainted, it is classified as a transmit instruction. To find out if it input register is tainted, the taint mask is indexed with input register `prs1`. Because loads only have one input in BOOM, there is no need to check the other registers. If a microp-op is a transmit instruction, the taint dependency in the load queue entry is set to point to the index of the access instruction in the load queue. The transmit instruction can not be executed until the access instruction it depends on has lost its taint.

3.2.4 Untaint When Access Instruction is Past the Visibility Point

As mentioned, shadows are used to know whether access instructions are safe to issue, and their dependencies should be untainted. Figure 3.5 shows the load queue with the head, visibility point, and tail. The visibility point works as a pointer to the entry in the load queue which is currently the oldest shadow. It is used to indicate which entries in the load queue are safe to issue. Speculative loads older than the entry the visibility point points to are safe, while the younger ones still have to have their registers tainted. The value of the pointer increases one position each cycle if the entry it is pointing to is not shadowed. Each time it increases, it also untaints the registers tainted by the load that the visibility point moves away from. This means that if a load has been classified as an access instruction without being speculative, the registers tainted by that instruction would be untainted right away when the visibility points reach that entry. If the entry the visibility point points to is shadowed, the visibility point stays at that position until it is no longer shadowed. Since no loads can be older than the head in the load queue, and not younger than the tail, the visibility point always has to be between the head and the tail, as shown in Figure 3.5.

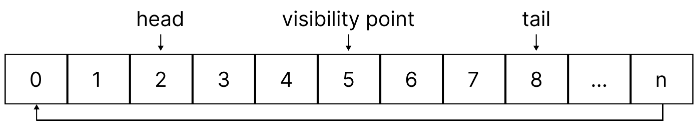
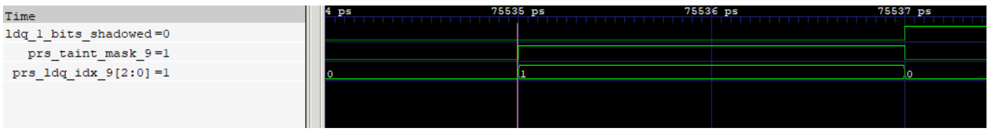
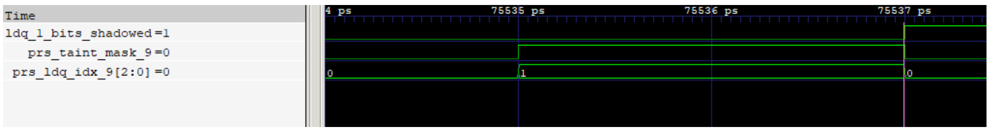


Figure 3.5: Load queue with head, visibility point, and tail



(a) A load initially taint register even if it is shadowed or not



(b) The access instruction gets shadowed after the register depending on it gets untainted

Figure 3.6: The taint is removed before it is detected that the load is shadowed

When untainting registers, the timing has also in here been a challenge. Figure 3.6 shows the output when doing waveform debugging in GTKWave. As can be seen in Figure 3.6a, register 9 gets tainted in the taint mask by the load with load queue index 1. Since every load initially is classified as an access instruction, the register gets tainted despite the load queue entry not being shadowed. Since there is no shadow, the register's entry in the bit-mask is untainted. However, as can be seen in Figure 3.6b, the load queue gets shadowed in the same cycle the register is untainted. The reason for this delay may be that, when a load arrives, it takes more than one cycle to detect the shadow. The D-shadow needs to check the store queue, and the store dependency mask before it can be calculated, and for C-shadows, the branch mask has to be checked to know if there are any speculative branches. The shadow is not set in the load queue entry until after these are calculated. It is not known how long calculating the C- and D-shadows take, but by looking at Figure 3.6 it can be assumed that it in this case takes one cycle before the shadow is calculated.

The problem can be fixed by delaying the untainting so that the access instruction can get time to be marked as shadowed before its dependencies get untainted. The solution to this delay has not been tested out enough to conclude whether it solves the problem or not. Therefore, like with taint propagation, the following paragraph is only a suggested solution to how it can be implemented in the future. The suggested solution is to block loads from untainting one cycle. This is done by implementing a block mask over the physical register file. The values in the mask are set to one when registers get tainted. Registers that are blocked in the block mask, should not get untainted before they are no longer blocked. The implementation of a one-cycle delay involves having two masks. The masks are initialized as shown in Code listing 3.4.

Each cycle, each of the entries in the block mask is defaulted to 0, except for the register which get tainted in the same cycle. These registers are set to 1. More than one register can get tainted in one cycle because the core width may allow for more than one instruction being processed at a time. There is also a new mask for the next cycle. Each cycle, the second bit mask has the same registers blocked

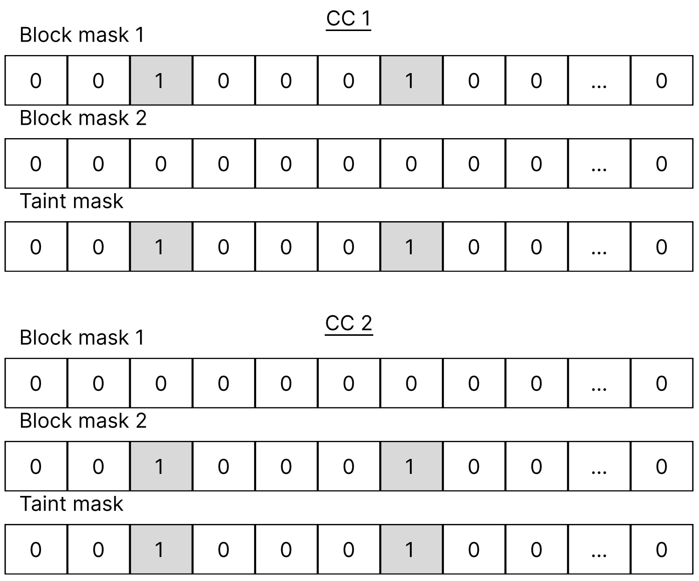


Figure 3.7: Block mask delaying untainting of registers

as the first had the last cycle. An example of this is shown in Figure 3.7. The figure shows the state of the two block masks in addition to the taint mask in two consecutive clock cycles. In cycle one, register 3 and 7 gets tainted. To ensure that they don't get untainted too early, they have to be blocked for one cycle. The bits are therefore also set in block mask 1, in addition to the taint mask. In cycle 2 no new registers are tainted, all the fields in block mask one therefore goes back to 0 which is the default value. In block mask 2 on the other hand, the bits are set to be the same as block mask 1 was in clock cycle 1. This ensures that the registers stays tainted for one more cycle and don't get untainted before the shadow is detected.

Code listing 3.4: Block mask to ensure that loads don't get untainted before the shadow is detected

```
val block_taint_mask    = WireInit(VecInit((0 until numIntPhysRegs).map(x=>false.B)))
val p1_block_taint_mask = RegNext(block_taint_mask)

...
// When tainting the destination register of an access instruction
block_taint_mask(pdst) := true.B
```

Chapter 4

Methodology

For this study, implementations to protect against speculative side-channel attacks have been made. In addition, tests are performed to collect data about the implementations and be analyzed to get empirical evidence that can be used to accomplish the overall goal of finding out how the implementations compares to each other and the baseline when looking at the attributes, performance, implementation overhead and security provided. This chapter will explain how the data, which are presented in Chapter 5, are collected and analyzed.

4.1 Experimental setup

The implementations implemented in this project have been evaluated with quantitative experiments. To evaluate the different implementations, the attributes performance, and implementation overhead have been quantified and measured. This has been done by running simulations on an FPGA. Tests to verify the security have also been performed on FPGA simulations. Since the Speculative Taint Tracking implementation is not completed, the tests on this implementation are not as comprehensive as the other implementations. The test for implementation overhead is done equally to the other, but since the implementation is not finished, the overhead measured does not reflect the overhead of the complete product. The test for security also does not reflect the result of the finished product. In addition, the test for security is performed in a software simulator, and not on an FPGA like the other implementations. The tests for performance are not performed at all on the STT implementation. This will be further discussed in Section 4.3.

To test for performance, the SPEC2017 benchmark suite is run. The number of instructions per clock cycle (IPC) is measured for each benchmark and implementation. To find the normalized IPC, the geometric mean of these measurements is calculated for each implementation. A geometric mean is a suitable measurement in this case, as the implementations are compared to the baseline implementation.

It also ensures that the result is more representative, by reducing bias towards outliers, like, for example, one benchmark performing significantly better than the rest.

To test the security of the implementations, Spectre attacks of Variant 1 and Variant 2 have been performed. These attacks verify that the implementations are secure against control flow speculation combined with cache timing attacks. It is however not tested in this project if the implementations are secure against all the discovered Spectre attacks. One can also not be sure if there are undiscovered attacks, which still are able to bypass the security of the implemented solutions. Section 4.3 will discuss some of these concerns.

To find the implementation overhead, information about the bitstream generated when simulating the implementations on the FPGA has been analyzed. Since the FPGAs use CLBs to physically implement the design, one can analyze the utilization of these to get an estimation of how large the physical realization would be. This information is generated in an output file by Vivado, which is the software created by Xilinx to generate the bitstream. It is hard to know anything about the size of the physical realization from only looking at the utilization of CLBs, therefore the modified implementations are compared to the baseline implementation to give information about the increase in hardware, rather than the actual number.

The data generated from Vivado contains information about the different subcomponents of the CLB. This is, for example, the number of LUTs used and CLB flip-flops. It also gives information about how many of the LUTs which is used for logic, and how many which is used as memory. Since none of the changes implemented are part of the memory module, the number of LUTs used as memory is expected to be the same for all of the implementations. This number is thus not used when evaluating the designs in Chapter 5. The implementation overhead is only measured in the number of LUTs used for logic, and the number of CLB flip-flops as these are the attributes expected to be affected by the modifications to the BOOM core.

4.1.1 FPGA Evaluation

The FPGA evaluation has been done on the EPIC [33] research infrastructure, which is available through the IDUN cluster [34]. EPIC has four Xilinx Alveo U250 FPGA cards [35], which master students and researchers can use to simulate hardware designs. The software simulations performed have also used the IDUN cluster, only that these are performed on CPU nodes.

To evaluate the designs on the FPGA, the *Firesim* [22] platform and the *Chipyard* framework [19] have been used. The tool's official GitHub repositories [36] [37] have been forked by the NTNU EECS research initiative. These forked repositories have been installed on the IDUN cluster, to be used for the experiments in this project. To use the FPGAs on the IDUN cluster, the description on the EECS u250 wiki-page on GitHub [38] has been followed. This provides information on how to install the correct version of Chipyard, build the bitstream for the design, flash the bitstream to the FPGA, and run the FireSim simulation.

The FireSim images used on the FPGA simulation come from NTNU EECS's fork of the official FireMarshal GitHub repository [39]. The FireSim images can provide a RISC-V Linux distribution that runs on the target platform. The images are built from a configuration file and may also be configured to provide a root file system copied to the target platform. To be able to run the benchmarks and Spectre software, the necessary binaries and other files are copied into the root file system, to make it available at the target platform.

When testing for the performance of the implementation, the image *ntnu-base* has been used. This image builds on the *Build-Root Linux* base image, which provides a Linux environment for embedded systems [24]. The ntnu-base image provides tools used to run the *SPEC 2017* benchmarks on the target platform. These tools involve a Python program created by Björn Gottschall at NTNU called *invoke* [40]. This program is used to invoke SPEC 2017 binaries. Another tool is *perf* [41] which is a profiler tool for Linux. Within perf, *perf-stat* can be used to sample the state of certain attributes in the processor core and count events like the number of clock cycles and the number of instructions when a binary is executing. These can be used to calculate the IPC, which gives an indication of the performance of the implementation.

To test the security, a custom image has been used. The *fedora-base* image is used as the base for this image. The image consists of only an installation of the Fedora RISC-V Linux distribution [25], in addition to RISC-V binaries of Spectre Variant 1 and 2. The implementation of the Spectre attack is further described in Section 4.1.3.

4.1.2 SPEC2017 Benchmark Suite

The performance of the implementations has been evaluated by running the SPEC2017 benchmark suite [42]. SPEC2017 is a collection of various workloads which are executed to test the design. It is intended to be diverse, but still a representative collection of workloads commonly executed at processors. The suite consists of both floating point benchmarks, which are programs with a higher rate of floating point operations, and integer benchmarks, which are benchmarks mostly using integer operations. Table 4.1 shows some information about the integer benchmarks executed, and Table 4.2 gives some insight to the floating point benchmarks.

The binaries for the benchmarks were acquired from NTNU EECS. They were executed with the invoke Python program. The execution of the benchmarks can be configured through a JSON file. The configuration file can be used to specify input sets, wrappers, and instruction set. Since the tests are run on BOOM, the RISC-V

Table 4.1: List of SPEC2017 integer benchmarks and a description of the programs [42]

Benchmark	**Language**	**Description**
500.perlbench	C	Perl interpreter
502.gcc	C	GNU C compiler
505.mcf	C	Route planning
520.omnetpp	C++	Discrete Event simulation
523.xalancbmk	C++	XML to HTML conversion via XSLT
525.x264	C	Video compression
531.deepsjeng	C++	Alpha-beta tree search (Chess)
541.leela	C++	Monte Carlo tree search (Go)
548.exchange2	Fortran	Recursive solution generator (Sudoku)
557.xz	C	General data compression

Table 4.2: List of SPEC2017 floating point benchmarks and a description of the programs [42]

Benchmark	**Language**	**Description**
503.bwaves	Fortran	Explosion modeling
507.cactuBSSN	C++, C, Fortran	Physics: relativity
508.namd	C++	Molecular dynamics
510.parest	C++	Biomedical imaging
511.povray	C++, C	Ray tracing
519.lbm	C	Fluid dynamics
521.wrf	Fortran, C	Weather forecasting
527.cam4	Fortran, C	Atmosphere modeling
538.imagick	C	Image manipulation
544.nab	C	Molecular dynamics
549.fotonik3d	Fortran	Computational Electromagnetics
554.roms	Fortran	Regional ocean modeling

64-bit instruction set is chosen. Wrappers, in the context of the invoke program, are software that are run together with the benchmarks. This may, for example, be profiling the execution with perf, or time the execution with the *time* program. As mentioned, the perf-stat wrapper is used to find the IPC of the benchmarks. When it comes to the input set, the set called *test* is chosen. This is the smallest of the available input sets, which in return makes for a faster execution time of the benchmarks. The *ref* input set is larger and was first tried out. But since one benchmark ran for 3 days with the ref input set, a smaller one was chosen instead. With the test set, the whole suite finishes in a couple of days on the FPGA. If the input test were going to run on a fully realized processor core, a larger input set may be more adequate, since FPGA simulation is slower than real hardware.

4.1.3 Spectre Attack

To verify that the DoM, DoM-RW, and STT implementations work as expected, Spectre attacks have been performed on the different versions, in addition to the baseline BOOM implementation. The attack software comes from the public GitHub repository boom-attacks [43]. This is made by two of the main contributors to the BOOM core from Berkeley University. It has implementations of Spectre Variant 1 and 2. This repository is not maintained and is designed for an older version of the BOOM core. Later commits to the BOOM repository have made the attack unable to retrieve the secret data through the cache side channel. The attack, however, works as intended with some changes in the attack code. The details about the changes in the attack code needed, and why they are needed, will be further explained after the attack code is explained.

The attack code for both Spectre Variant 1 and 2 tests if the characters in the string *!"#ThisIsTheBabyBoomerTest* can be retrieved through a speculative side-channel. It iterates over the characters in the string and performs the attack on each character, one by one. For each character, the attack is performed 10 times to get a more accurate result. The attack sets a threshold of 50 cycles to determine if a load is present in the cache or not. Yet, a latency of less than 50 cycles is not a guarantee that the data is present in the L1 cache. This is why the attack is run several times for each character. For each attack round, a counter is increased on the cache lines which have a latency below the threshold. The higher the counter is for a cache line, the more rounds, that cache line have a low latency. By looking at how many rounds the latency is below the threshold, one can be more certain that the test actually has resulted in a cache hit, and not randomly had a short latency.

As mentioned, the attack code needs a minor change to work every round. One of the changes in BOOM, which is implemented after the attack code was created, is a new branch predictor called the *loop* predictor. Since the attack code works after removing this predictor, it is assumed that this predictor is the cause of the trouble. Since the attack also still work for Variant 2 which exploits the branch target predictor, not the predictor predicting the direction like the loop predictor, it makes the assumption more probable. A suggestion as to why the loop predictor makes the attack not work is that it learns the branch pattern after the first round, and because of this does not make the misprediction that is crucial for executing the transient instructions. The reason is probably that the loop predictor learns the branch pattern for constant-length loops when training the branch predictor in the setup phase of the attack. The attack works as expected on the baseline implementation when the loops in the setup phase are not constant-length. With this knowledge, the only change needed is making the loop in the training phase increase the number of iterations by one each training round.

4.2 BOOM Configuration

The CPU pipeline in BOOM is parameterized, which means that it makes it possible to build it with various configurations. As previously mentioned, the configurations decide several parameters like the core width, and number of entries in the different data structures like the ROB, store queue, and load queue. The core comes with some predefined configurations, but hardware designers could also build their own custom configurations. For the experiments done on the FPGA, the *Firechip* configuration *FireSimLargeBoomConfig* is used. This implements the *LargeBoomConfig* from Chipyard and has been run on a 30Mhz clock frequency. Table 4.3 lists some of the parameters in LargeBoomConfig.

Since the Speculative Taint Tracking was unable to run on the FPGA, it was run on the software simulator *Verilator*. The implementation has, with Verilator, been tested for security with both the *SmallBoomConfig* and the *LargeBoomConfig*. Since the Spectre programs run for the longest time without terminating on the SmallBoomConfig, these results are the ones presented and discussed later in this report.

4.3 Limitations

This section will present the factors of the methodology that can make the data produced inaccurate. In general, the more data produced and the more different variations in the experiment setup, the more accurate the result would be. For example, having data about more different Spectre attacks, and having the mitigation strategies being implemented on more platforms would give more data to analyze. Also having data about more configurations of BOOM, in addition to more frequencies than 30MHz, could maybe also be valuable. To get more precise results, testing the physical realization of the designs would also be better than testing a simulation of the designs. However, the time limit and the resources available for the project are limited. With this, the scope, which is presented in Chapter 1, also has to be scaled down to be suitable for the circumstances. Having

Table 4.3: Parameters in the `LargeBoomConfig`

Parameter	**Value**
Fetch width	8
Decode width	3
No. ROB entries	96
Memory issue width	1
Int issue width	2
FP issue width	1
No. physical int registers	100
No. physical FP registers	96
No. LDQ/STQ entries	24
Max branch count	16
L1 cache size	32 KB

the designs on the FPGA at least gives more certain results than when only testing on software simulators.

As previously mentioned, the implementation of STT was not completed. As a consequence of this, the simulation of this implementation has some unwanted behavior. Some bare-metal programs tested on the Verilator simulation of STT make the simulation terminate on an assertion in the BOOM code saying *"Pipeline has hung"*. It is not known why this assertion is triggered, since after looking at the waveform generated, signals associated with the code specific for STT have the expected values when the assertion triggers. No loads are also not delayed when the assertion is triggered. The implementation also fails to be simulated on the FPGA. A bitstream is generated, but as with the software simulation, the software executed at the simulated system also here seems to hang. For this reason, the tests done on STT are only run in software simulation with Verilator. Since Verilator only executes bare-metal binaries on the simulation, a lot of extra work is needed to be able to run the benchmarks on Verilator. The results are anyway not valuable when the benchmarks have been run on a different platform than the other implementations. For this reason, the implemented STT simulation doesn't get tested for performance.

The implementations are only tested for Spectre Variant 1 and Variant 2. To fully verify that the mitigation strategies are secure against speculative side-channel attacks, more variants of the attack should be performed. It is, however, time-consuming to implement these attacks from scratch, and the resources on other attacks are also limited. Most of the attacks are likely to have to be tailored to fit the BOOM core, as with the Spectre Variant 1, where the loop predictor learns the behavior of the setup phase in the attack from the resource used. Due to the time limit of the project, there was no time to implement more attacks. It is however a limitation of the research, since it is with this not proven that the implementations are secure against other attacks than this type of side-channel attacks. It would for example be interesting to see the result from Spectre Variant 4 since the D-shadows implemented in DoM should protect against these.

Chapter 5

Result

This chapter will present the data collected from the experiments described in Chapter 4. The experiments measure the performance, and implementation overhead, in addition to verifying the security for the Baseline, DoM, DoM-RW, and STT implementations. The result will only be presented in this chapter and further discussed in Chapter 6. First, the results from the performance measurement will be presented. Then the implementation overhead will be shown. Lastly, it is shown how secure the implementations are against speculative side-channel attacks.

5.1 Performance

As mentioned, the SPEC2017 benchmark suite has both integer workloads and floating-point workloads. Table 5.1 and Table 5.2 present the IPC measured for all of the integer, and floating point benchmarks respectively. In each of the cells, next to the IPC, it is also presented how much of a slowdown the implementation has compared to the baseline BOOM implementation. Figure 5.1 and Figure 5.2 shows the data from the two tables presented as a bar chart. The tables and figures also shows the normalized performance for the implementations. Table 5.1 and Figure 5.1 shows the normalized IPC for the integer benchmarks, and Table 5.2 and Figure 5.2 shows the normalized IPC for the floating point benchmarks. Both tables and bar charts also shows the performance normalized for all benchmarks. For the DoM implementations, the slowdown across all benchmarks is 14%, and for the DoM-RW implementation it is 20.9%. For the integer benchmarks, the normalized slowdown is 11.3% for DoM and 16.5% for DoM-RW. For the floating point benchmarks, it is 16.7% for DoM and 34.4% for DoM-RW.

As expected, implementing the DoM and DoM-RW security mechanisms gives a slowdown for almost all benchmarks. This can also be deducted by reviewing the normalized slowdown. This confirms the assumption about the mitigation strategies giving a slowdown. With a total slowdown of 14% and 20.9% across all benchmarks, the slowdown can be said to be non-negligible.

One of the most interesting things to get from the result is comparing the DoM and DoM-RW performance. The hypothesis made before performing any of the experiments was that by restricting the wakepus the implementation would get a better performance. The results presented shows that across all benchmarks, DoM gets a slowdown of 14% and DoM-RW gets a performance slowdown of 20.9%. This tells that the hypothesis made is disproved. Why this is the case can not be determined with the result from the performed experiments, but the issue will be further discussed in Section 6.1 in Chapter 6.

The result presented in the tables and figures shows that the performance is generally slower for floating point benchmarks than integer benchmarks. This is expected, as it is a known fact that floating point instructions are more complex to calculate than integer instructions. It was however not as obvious that the slowdown for floating point benchmarks also turned out to be significantly larger than for the integer benchmarks. Why this may be the case can not be determined from the acquired result, but a hypothesis about it is discussed in Section 6.1.

As shown in the tables and bar charts, there are some variations on the slowdown for the different applications. For example, the *perlbench* benchmark have no slowdown for the normal DoM implementation, and only 1.6% on the DoM-RW implementation. The integer benchmark called *exchange2* are also almost the same. It, however, surprisingly gives a greater IPC with DoM than the baseline implementation. It is yet such a small increase that it can be considered negligible. On the other contrary, the integer benchmark *mcf* and the floating point benchmark *cactuBSSN* gives larger slowdown with 47.7% and 36.8% for DoM respectively and 50% and 40.8% for DoM-RW respectively. The standard deviation presented in Table 5.3 also indicates that the data varies. Across all benchmarks the IPC for the baseline implementation is measured to be 0.86 and as shown in Table 5.3 the standard deviation for this is 0.43. For the DoM implementation, the IPC is 0.74 and the standard deviation is 0.45 and for DoM-RW the IPC is 0.68 and the standard deviation is 0.44. As it is not part of the scope, experiments trying to deduct why the variation is this large is not performed. However, the proportion of instructions creating speculation combined with the proportion of instructions being loads in a program are factors that can influence the performance of the program when DoM is implemented.

Table 5.1: Performance in measured in IPC and the slowdown in percentage compared to baseline, for every INT benchmark

Benchmark	Baseline	DoM	DoM-RW
Geometric mean TOT	0.86	0.74 / **14.0%**	0.68 / **20.9%**
Geometric mean INT	0.97	0.86 / **11.3%**	0.81 / **16.5%**
500.perlbench	0.64	0.64 / **0.0%**	0.63 / **1.6%**
502.gcc	0.68	0.65 / **4.4%**	0.63 / **7.4%**
505.mcf	0.44	0.23 / **47.7%**	0.22 / **50.0%**
520.omnetpp	0.65	0.55 / **15.4%**	0.53 / **18.5%**
523.xalancbmk	0.75	0.65 / **13.3%**	0.56 / **25.3%**
525.x264	2.04	1.98 / **2.9%**	1.82 / **10.8%**
531.deepsjeng	1.40	1.37 / **2.1%**	1.24 / **11.4%**
541.leela	1.30	1.18 / **9.2%**	1.13 / **13.1%**
548.exchange2	1.97	1.99 / **-1.0%**	1.98 / **-0.5%**
557.xz	1.09	1.04 / **4.6%**	0.96 / **11.9%**

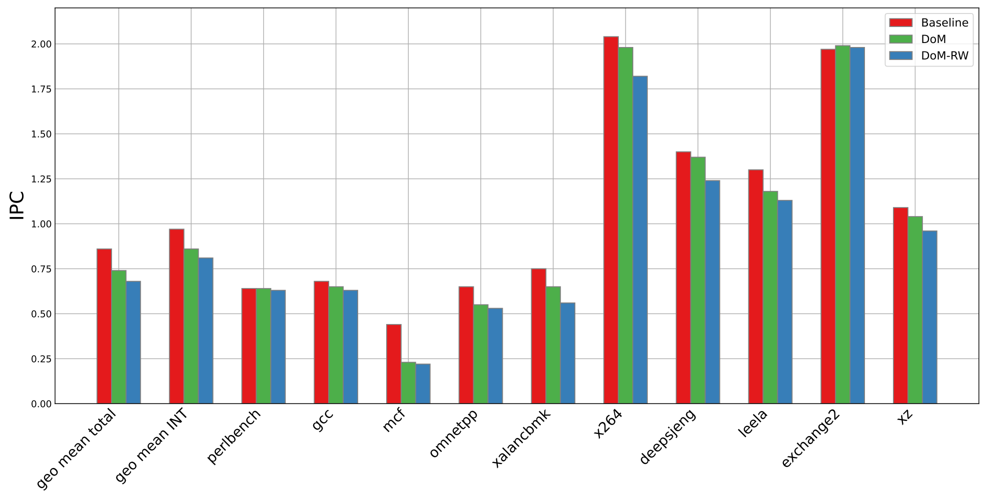


Figure 5.1: Performance measured in IPC with SPEC2017 INT benchmarks

Table 5.2: Performance in measured in IPC and the slowdown in percentage compared to baseline, for every floating point benchmark

Benchmark	Baseline	DoM	DoM-RW
Geometric mean TOT	0.86	0.74 / **14.0%**	0.68 / **20.9%**
Geometric mean FP	0.78	0.65 / **16.7%**	0.59 / **34.4%**
503.bwaves	0.42	0.39 / **7.1%**	0.38 / **9.5%**
507.cactuBSSN	0.76	0.48 / **36.8%**	0.45 / **40.8%**
508.namd	1.22	1.05 / **13.9%**	1.04 / **14.8%**
510.parest	0.91	0.75 / **17.6%**	0.65 / **28.6%**
511.povray	0.68	0.59 / **13.2%**	0.55 / **19.1%**
519.lbm	0.68	0.50 / **16.5%**	0.46 / **32.4%**
521.wrf	0.79	0.66 / **16.5%**	0.56 / **29.1%**
527.cam4	0.85	0.73 / **14.1%**	0.63 / **25.9%**
538.imagick	1.21	1.08 / **10.7%**	0.91 / **24.8%**
544.nab	0.53	0.52 / **2.9%**	0.51 / **3.8%**
549.fotonik3d	1.04	0.96 / **7.7%**	0.96 / **7.7%**
554.roms	0.66	0.50 / **24.2%**	0.41 / **37.9%**

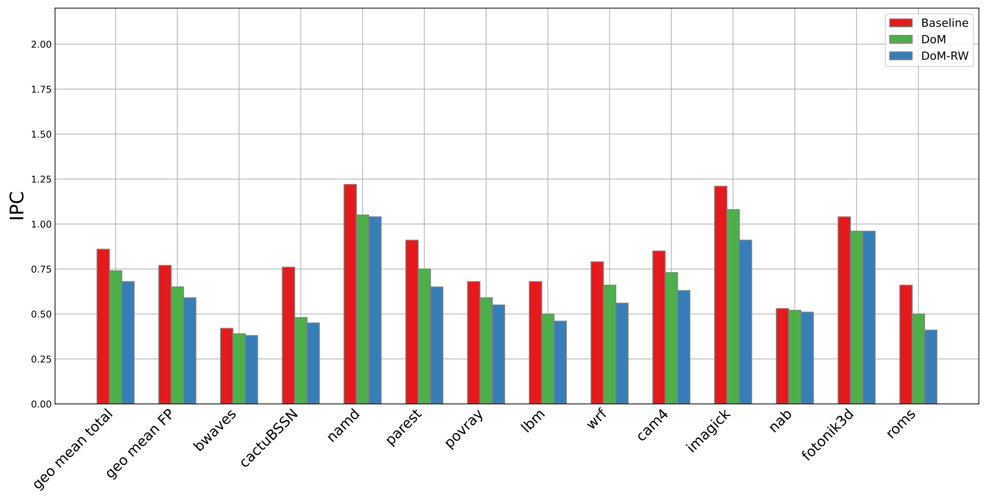


Figure 5.2: Performance measured in IPC with SPEC2017 FP benchmarks

Table 5.3: Standard deviation of the measured IPC for across all benchmarks, and integer and floating point benchmarks separately

Design	Total σ	INT σ	FP σ
Baseline	0.43	0.54	0.24
DoM	0.45	0.57	0.23
DoM-RW	0.44	0.30	0.22

5.2 Implementation Overhead

The implementations' overhead is measured with the hardware utilization of the FPGA. As mentioned in Section 4.1, the utilization is measured in number of look-up tables used as logic and the number of CLB flip-flops used. Table 5.4 shows the number of LUTs and the number of flip-flops (FFs) used for each implementation. In addition, the increase as percentage of the baseline implementation are also included for both. As the table shows, the increase in implementation overhead is minimal, with a 0.1% increase in number of LUTs as logic and 0.02% increase in number of flip-flops for both the DoM implementation and DoM-RW. This can be considered negligible. However, it is surprising that the DoM-RW implementation don't get any overhead when implementing the restricted wakeup policy. Deducting why these are the same is, as defined in Chapter 1, not considered as part of the scope for this project, and can thus not be observed by the experiments performed.

The STT implementation is also included in the table, but as previously mentioned, this implementation is not completed. This likely gives the STT implementation a lower overhead than what would have been if the implementation would also include taint propagation for all instructions. As shown in the table, the STT implementation gets a 1.2% increase in number of LUTs, and 0.49% increase in number of flip-flops used. It is hard to set the limit of when the increase in implementation overhead is non-negligible or not. Although the increase of the STT implementation is not considerably large, it is for sure considerably larger than the for the DoM implementations, especially when considering the fact that the overhead probably is larger on the finished STT implementation. For most systems, however, the overhead would probably be said to be negligible.

The entire table over the subcomponents of the CLB can be found for each of the implementations in Appendix C. As expected, there is no change in the number of LUTs used as memory in any of the implementations. The main differences are, also as expected, in the number of LUT used for logic and the number of CLB flip-flops.

Table 5.4: Implementation overhead in the different designs, measured in numbers of LUTs as logic, and number of flip-flops used

Implementation	No. LUT as Logic	No. CLB FF	LUT Increase	FF Increase
Baseline	321053 LUTs	202235 FFs	-	-
DoM	321374 LUTs	202266 FFs	0.10%	0.02%
DoM-RW	321374 LUTs	202266 FFs	0.10%	0.02%
STT	324913 LUTs	203216 FFs	1.20%	0.49%

5.3 Secutity

Code listing 5.1 and Code listing 5.2 shows the output after running the Spectre Variant 1 attack described in Section 4.1.3 for the baseline implementations and the DoM implementation respectively. All the other results from Spectre Variant 1 and Variant 2 run for this project are available in Appendix A and Appendix B. Each line in the output text comes from a Spectre attack performed on the address to the left. The secret character expected to be found at the address is stated in the parentheses in `want()`. As explained in Section 4.1.3, the attack is performed 10 times on each address, and the two most likely values are selected. The line continues to show both the best values, with the number of attack rounds the character has given a cache hit, and also the value as both a decimal and character. The character value can then be compared to the value which are expected, to see if the attack works or not.

The result is summarized in Table 5.5a and Table 5.5b showing the summary of Spectre Variant 1 and Variant 2 respectively. For each implementation, the table tells whether the attack performed on the implementation have leaked any data or not. For the answer to be “No”, the attack has to be unable to guess any of the characters. If only one of the characters is leaked, it is classified as a leaking implementation. As shown in the table, both DoM and DoM-RW succeeds to secure the processor core against both Spectre Variant 1 and Variant 2, while the Baseline implementation leaks data. This as expected before the experiments were performed.

As explained with the asterisk symbol in the table, the STT implementation is not finished and thus does not reflect the true security of the STT mitigation strategy. The implemented solution is, as explained in Section 3.2, only implemented with taint propagation on load-load dependencies. As mentioned in Section 4.1, the test for security is also only implemented on the Verilator software simulation for the STT implementation, and not on an FPGA like the other implementations. The attack code run on the STT implementation on the software simulator also terminates before it is finished. The data analyzed for the STT therefore less complete than for the other implementations, as there are less attack rounds to analyze. The data generated however reveal that the STT implementation with load-load taint propagation is not secure against the tested attacks.

Code listing 5.1: Output from a Spectre Variant 1 attack on an unsecure core

```
m[0x0x1c450] = want(!) =?= guess(hits,dec,char) 1.(10, 33, !) 2.(1, 1, )
m[0x0x1c451] = want(") =?= guess(hits,dec,char) 1.(9, 34, ") 2.(2, 1, )
m[0x0x1c452] = want(#) =?= guess(hits,dec,char) 1.(10, 35, #) 2.(3, 90, Z)
m[0x0x1c453] = want(T) =?= guess(hits,dec,char) 1.(10, 84, T) 2.(3, 7, )
m[0x0x1c454] = want(h) =?= guess(hits,dec,char) 1.(9, 104, h) 2.(2, 8,)
m[0x0x1c455] = want(i) =?= guess(hits,dec,char) 1.(8, 105, i) 2.(3, 252, |)
m[0x0x1c456] = want(s) =?= guess(hits,dec,char) 1.(9, 115, s) 2.(2, 10,
)
m[0x0x1c457] = want(I) =?= guess(hits,dec,char) 1.(7, 73, I) 2.(2, 83, S)
m[0x0x1c458] = want(s) =?= guess(hits,dec,char) 1.(7, 115, s) 2.(4, 7, )
m[0x0x1c459] = want(T) =?= guess(hits,dec,char) 1.(9, 84, T) 2.(2, 10,
)
m[0x0x1c45a] = want(h) =?= guess(hits,dec,char) 1.(9, 104, h) 2.(2, 3, )
m[0x0x1c45b] = want(e) =?= guess(hits,dec,char) 1.(9, 101, e) 2.(2, 1, )
m[0x0x1c45c] = want(B) =?= guess(hits,dec,char) 1.(7, 66, B) 2.(2, 122, z)
m[0x0x1c45d] = want(a) =?= guess(hits,dec,char) 1.(7, 97, a) 2.(3, 75, K)
m[0x0x1c45e] = want(b) =?= guess(hits,dec,char) 1.(9, 98, b) 2.(2, 7, )
m[0x0x1c45f] = want(y) =?= guess(hits,dec,char) 1.(8, 121, y) 2.(2, 7, )
m[0x0x1c460] = want(B) =?= guess(hits,dec,char) 1.(10, 66, B) 2.(2, 7, )
m[0x0x1c461] = want(o) =?= guess(hits,dec,char) 1.(9, 111, o) 2.(3, 83, S)
m[0x0x1c462] = want(o) =?= guess(hits,dec,char) 1.(10, 111, o) 2.(2, 28, )
m[0x0x1c463] = want(m) =?= guess(hits,dec,char) 1.(7, 109, m) 2.(2, 6, )
m[0x0x1c464] = want(e) =?= guess(hits,dec,char) 1.(7, 101, e) 2.(2, 3, )
m[0x0x1c465] = want(r) =?= guess(hits,dec,char) 1.(10, 114, r) 2.(2, 5, )
m[0x0x1c466] = want(T) =?= guess(hits,dec,char) 1.(8, 84, T) 2.(2, 3, )
m[0x0x1c467] = want(e) =?= guess(hits,dec,char) 1.(8, 101, e) 2.(2, 204, L)
m[0x0x1c468] = want(s) =?= guess(hits,dec,char) 1.(8, 115, s) 2.(2, 1, )
m[0x0x1c469] = want(t) =?= guess(hits,dec,char) 1.(9, 116, t) 2.(2, 0, )
```

(a) Security against Spectre Variant 1 of each implementation

Implementation	Leaks data
Baseline	Yes
DoM	No
DoM-RW	No
STT	Yes*

(b) Security against Spectre Variant 2 of each implementation

Implementation	Leaks data
Baseline	Yes
DoM	No
DoM-RW	No
STT	Yes*

*The STT implementation is not complete and does therefore not reflect the true security of the mitigation strategy

Code listing 5.2: Output from a Spectre Variant 1 attack on the DoM implementation

```
m[0x0x1c450] = want(!) =?= guess(hits,dec,char) 1.(2, 7, ) 2.(2, 10,
)
m[0x0x1c451] = want(") =?= guess(hits,dec,char) 1.(3, 252, |) 2.(2, 97, a)
m[0x0x1c452] = want(#) =?= guess(hits,dec,char) 1.(2, 62, >) 2.(2, 91, [)
m[0x0x1c453] = want(T) =?= guess(hits,dec,char) 1.(2, 1, ) 2.(2, 7, )
m[0x0x1c454] = want(h) =?= guess(hits,dec,char) 1.(2, 5, ) 2.(2, 46, .)
m[0x0x1c455] = want(i) =?= guess(hits,dec,char) 1.(2, 1, ) 2.(2, 19, )
m[0x0x1c456] = want(s) =?= guess(hits,dec,char) 1.(2, 5, ) 2.(2, 12,
                                                                     )
m[0x0x1c457] = want(I) =?= guess(hits,dec,char) 1.(2, 67, C) 2.(2, 176, 0)
m[0x0x1c458] = want(s) =?= guess(hits,dec,char) 1.(2, 5, ) 2.(2, 157, )
m[0x0x1c459] = want(T) =?= guess(hits,dec,char) 1.(2, 3, ) 2.(2, 144, )
m[0x0x1c45a] = want(h) =?= guess(hits,dec,char) 1.(2, 1, ) 2.(2, 100, d)
m[0x0x1c45b] = want(e) =?= guess(hits,dec,char) 1.(2, 5, ) 2.(2, 6, )
m[0x0x1c45c] = want(B) =?= guess(hits,dec,char) 1.(2, 111, o) 2.(2, 124, |)
m[0x0x1c45d] = want(a) =?= guess(hits,dec,char) 1.(2, 7, ) 2.(2, 10,
)
m[0x0x1c45e] = want(b) =?= guess(hits,dec,char) 1.(2, 7, ) 2.(2, 60, <)
m[0x0x1c45f] = want(y) =?= guess(hits,dec,char) 1.(3, 5, ) 2.(2, 3, )
m[0x0x1c460] = want(B) =?= guess(hits,dec,char) 1.(2, 26, ) 2.(1, 1, )
m[0x0x1c461] = want(o) =?= guess(hits,dec,char) 1.(2, 112, p) 2.(2, 153, )
m[0x0x1c462] = want(o) =?= guess(hits,dec,char) 1.(3, 7, ) 2.(3, 180, 4)
m[0x0x1c463] = want(m) =?= guess(hits,dec,char) 1.(2, 60, <) 2.(2, 110, n)
m[0x0x1c464] = want(e) =?= guess(hits,dec,char) 1.(1, 1, ) 2.(1, 2, )
m[0x0x1c465] = want(r) =?= guess(hits,dec,char) 1.(3, 3, ) 2.(2, 8,)
m[0x0x1c466] = want(T) =?= guess(hits,dec,char) 1.(3, 100, d) 2.(2, 7, )
m[0x0x1c467] = want(e) =?= guess(hits,dec,char) 1.(2, 7, ) 2.(2, 36, \$)
m[0x0x1c468] = want(s) =?= guess(hits,dec,char) 1.(1, 1, ) 2.(1, 2, )
m[0x0x1c469] = want(t) =?= guess(hits,dec,char) 1.(2, 163, #) 2.(2, 172, ,)
```

Chapter 6

Discussion

This chapter will use the sections Section 6.1, Section 6.2 and Section 6.3 to discuss the results in Chapter 5. The rest of the chapter will be used to discuss the outcome of the design choices made for the implementations of the mitigation strategies and other circumstances regarding the project.

6.1 Performance

One hypothesis made before performing any of the experiments, was that restricting wakeups would give better performance since fewer unnecessary cache lookups would be done when the loads are still shadowed. The reason why the hypothesis in this report was that issuing shadowed loads was unnecessary is that it was assumed that if the loads were not present in the cache the first time it was issued, the possibility of it being there when the load is reissued was low. The reasoning for the hypothesis is further explained in Section 3.1.4.

The performance decrease however shows that optimistically continuing to do cache lookups despite the load still being shadowed is actually not as unnecessary as anticipated. It can not be determined from the experiments performed why optimistically issuing shadowed loads to the cache yields a better performance, and it is also not part of the scope of the project. However, this section will present some new hypotheses about why the result is what it is, which can be used for later research.

One reason for why the hypothesis is disproved may be that loops have many branch instructions in addition to having many strided loads from the same cache line. The branch instructions create shadows on the successive loads, which also entails that the loads should be delayed if not present in the cache.

An assembly pseudocode example of a for-loop is shown in Code listing 6.1. The code starts on the tag `START` in line 8. Register `r2` is the register which holds the value counting the number of iterations. The register is initially set to zero with the `MV` instruction before the loop starts. The `BLE` instruction is a branch instruction which directs the control flow to the address of the first instruction after the `LOOP` tag in line 1 to start the loop. In the loop, the first iteration loads a value from the address in register `r3` into register `r1`. It then increases the value of register r3 to be ready to load the adjacent data element in the next iteration. It lastly increases the counter of the loop with one before it branches back to the start of the loop. This is done for a hundred iterations before the value in register `r2` is finally one hundred and the branch is not taken. The loop is then ended by jumping to the `END` tag.

The example in Code listing 6.1 is an example of a code snippet which is constantly encounters the reoccurring pattern of creating shadows and loops alternately. Each loop iteration the branch instruction in line 5 creates a shadow, which is cast over the load in line 2. The new hypothesis is that when the shadow cast over the first element in a cache line disappears, the entire cache line will be loaded into the L1 cache and thus making the addresses of the loads in the consecutive iterations available in the cache despite the load still being shadowed. In the DoM-RW implementation, these are delayed until the shadow disappears, which is unnecessarily late compared to the DoM implementations which optimistically issues shadowed loads and are able to hit in the cache for the elements which are in the same cache-line as an unshadowed load.

Code listing 6.1: Pseudo code of a loop in assembly

```
 1  LOOP:
 2      LD  r1, r3          // Load the data element from the adress on register e3
 3      ADDI r3, r3, #4     // Increase the index of the data element to be loaded
 4      ADDI r2, r2, #1     // Increase the loop counter
 5      BLT r2, #100, LOOP  // Loop one more time if the conter is less than 100
 6      J   END
 7
 8  START:
 9      MV r2, #0           // Set the iteration counter to zero
10      BLT r2, #100, LOOP  // Branch to the start of the loop, since r2 < 100
```

As presented in Chapter 5, the benchmarks have some variation in the slowdown of the DoM implementation. It is not known why this is the case, but the amount of speculative loads may be a key factor. The slowdown also seems to be greater for the floating point benchmarks. One reason for this may be that since floating point instructions often take longer time to execute than integer instructions, the speculative execution may as a consequence of this last longer, as the resolution of the speculation takes longer time.

6.2 Implementation Overhead

For this project, only the LUTs used as logic is analyzed, in addition to the number of flip-flops used for registers. As expected, the LUTs as memory stays constant. It is hard to know exactly how good of an estimate the number of LUTs is of the implementation overhead. While looking at the relative implementation overhead gives a good comparison between the implementations, further verifications are needed to determine how the overhead would actually be on a physical chip.

One part of accomplishing the overall goal was to answer how large of an implementation overhead the implemented mitigation strategies would make. As described in Chapter 5, the implementations show a relatively small overhead. While the measured overhead for the baseline, DoM and DoM-RW implementations are considered as a relatively good estimate, the STT implementations, as mentioned, does not reflect the overhead of the finished implementation. It can not be determined how large of an overhead the finished implementation will have. However, as the only part remaining is the logic for the taint propagation, it is likely that the implementation overhead does not get a massive increase. There can also be done changes in the current implementation of STT to reduce the implementation overhead, like for example synchronizing the timing, so that there is no longer a need for the block masks used for delaying the untaint mechanism.

6.3 Security

One of the goal for this project was to give an answer to whether the implemented defense strategies provide security against speculative side-channels or not. For the experiments performed, the DoM and DoM-RW implementations does not leak data. While Spectre Variant 1 and Variant 2 may be seen as the two most important attacks to be secured against, there are still other speculative side-channel attacks which the implementations have not been tested against. It could for example be interesting to test the implementations against Spectre Variant 4 and a Meltdown attack. It is not determined by any of the experiments performed for this research if the exception handling in BOOM can be exploited to leak data. It can also not be determined by any of the experiments if the baseline BOOM implementation speculatively allows loads to execute despite store values still waiting for their destination to be resolved. It would especially be interesting to look at Spectre Variant 4, since DoM has implemented D-shadows, which should mitigate this attack. The researchers behind STT also claims that STT is the first mitigation strategy to be secure against implicit side-channels. Implicit side-channels is when secret data implicitly dictates control-flow and executes transient instructions. An example of this is when the target of a branch instruction is decided by the secret data. The branch predictor will guess a direction, and the secret will be revealed by observing if the transient instructions have been in execution and the pipeline has been squashed when the branch resolves. In these side-channel attacks, it is not interesting to see the result of the transient execution, the secret is rather revealed simply by observing that the transient execution happened. Although executing more attacks would be a great contribution to the research as there would be more data about the security of the different implementations, the scope of the project had to be narrowed down for the project to be finished within the time limit. Writing attack code is a time-consuming task, as most attacks have to be tailored to the microarchitecture to some degree. Different processor cores may for example have different branch predictors which have to be trained a certain way. Spectre Variant 1 and Variant 2 were also the attacks with the most resources and available attack code for BOOM. The consequence of this is having less data to analyze and thus being less certain about the actual security of the implementations.

As seen in Chapter 5, the STT implementation is not secure against any of the performed attack. As mentioned before, the STT implementation is not complete and for now only implements taint propagation for load-load dependencies. The test, as mentioned, also terminates before it is finished. However, it can not be determined by any of the tests why STT leaks data. One could maybe think that securing the processor against load-load dependencies would result in some sort of security. Code listing 6.2 shows a snippet of the C-code from the attack code for Spectre Variant 1. One could maybe think that the access instruction loading `array1` would be done to a certain register, and that the transmit instruction loading `array2` would read it from the same register. In that case, the taint should be propagated to the transmit instruction even with only load-load propagation, since the second load has a direct dependency to the first load.

Code listing 6.3 shows the same snippet as in Code listing 6.2, only taht it is RISC-V assembly generated with the *gcc* compiler. It is not straight forward to understand the generated assembly, and it is therefore taken some assumptions by the author when explaining the code. The explanation must be read with the caveat that the interpretation of the assembly code may be wrong. The `bgeu` instruction in line 1 is a branch instruction. This instruction takes the control-flow to label `L14` if the value in register `a4` is greater or equal to the value in register `a5`. This is probably generated by the if-sentence and moves the control flow to `L14` if the expression in the if-sentence is false, i.e., the control-flow is moved away from the if-sentence. However, if the if-sentence is true or the branch predictor predicts that the branch is not taken, the assembly code continues. The instructions in line 2 to 5 is probably calculation of the address of `array[idx]`. It is assumed that the actual load of `array[idx]` is done in line 6. This load instruction is in STT considered as an access instruction. This instruction should taint the physical register mapped to register `a5`. Continuing in the code, it is assumed that the instructions in line 7 to 12 is the address calculation of the second load accessing the element in `array2`. The second load is considered as a transmit instruction in STT. What is interesting here is that this instruction actually uses logical register `a5` as an input register, which is the same register as the output register from the first load.

After having observed that the generated assembly code from the Spectre Variant 1 attack performed on the STT implementation, it looks like the dependencies of the access and transmit instructions are actually a load-load dependency when looking at the logical register. If they both map to the same physical register, the implemented version of STT should actually have secured the implementation against Spectre Variant 1. However, by reviewing the result obtained in Chapter 5, it does not. None of the experiments or code analysis can explain why that is the case, but some assumptions can be made. One possibility is that even though the loads use the same logical register, it is not the same physical register after the registers have been renamed. Since there are several of the instructions after the first load writing to register a5, the register-rename logic could detect that it is a write-after-write hazard and assign another physical output register to any of the instructions for calculating the address. This would result in the second load getting another register as the input register, and it would no longer have a direct load-load dependency with the first load.

Code listing 6.2: C code of the attack function

```
if (idx < array1_sz){
    dummy = array2[array1[idx] * L1_BLOCK_SZ_BYTES];
}
```

Code listing 6.3: Assembly code of the attack function

```
bgeu  a4,a5,.L14
lui a5,%hi(array1)
addi  a4,a5,%lo(array1)
ld  a5,-40(s0)
add a5,a4,a5
lbu a5,0(a5)
sext.w  a5,a5
slliw a5,a5,6
sext.w  a5,a5
lui a4,%hi(array2)
addi  a4,a4,%lo(array2)
add a5,a4,a5
lbu a5,0(a5)
sb  a5,-17(s0)
```

6.4 DoM Implementation

As can be reviewed by the results acquired, both DoM implementation provides security against the performed attacks. However, as mentioned in Section 6.3, the security testing could be more comprehensive and test for more attacks. It would, as mentioned, be useful to test against the Meltdown attack. The inventors of the DoM mitigation strategy have also described E-shadows and M-shadows, which are not implemented for this project. However, as the result shows, these shadow types are most likely not necessary to secure the processor core against Spectre Variant 1 and Variant 2. Since Meltdown exploits the exception handling, it would in future research be interesting to review whether the DoM implementation is secure against the Meltdown attack, both with and without the E-shadows implemented.

6.5 STT Implementation

This project fails to give good insight in the performance and security of the STT mitigation strategy. This section will however discuss why the STT implementation was so hard to implement, and explain what measures that can be taken to succeed in implementing the mitigation strategy in the future.

In general, the STT implementation is more complex and require more hardware structures than the DoM implementations. STT is also in the original paper [2] only implemented on the *O3* processor in the *gem5* software simulator and not in BOOM processor core. The creators may because of this not have met on the same challenges as when it is to be implemented on the BOOM core, and had an easier time implementing it. This is of course not known, can therefore only be considered as speculation.

6.5.1 Synchronization

One of the challenges when implementing STT has been to synchronize the different parts. As discussed in Section 3.2.4, the untainting mechanism is not synchronized with the shadow detection mechanism, resulting in too early untainting of some registers. The synchronization issue is in this case solved by delaying the untainting for one cycle. This is of course not an optimal solution and may contribute to poorer performance as every load instruction is blocked for one cycle.

The synchronization is also one of the challenges when it comes to taint propagation on other instructions than memory instructions. The taint have to be propagated to the Core module, and when doing this, it is hard to propagate the taint to the correct instruction.

6.5.2 Lacking STT Description

The lack of proper description of how the STT mechanism should be implemented has also been a challenge. The STT article [2] explain many of the most important concepts of taint tracking, like for example what access and transmit instructions are and how taints should be propagated to the youngest root of taint. It however don't have any detailed description of how instructions are classified as access instructions, and how it is detected that the access instruction is past the visibility point. For the STT implementation implemented for this project, it is chosen to use the shadows [15] from the DoM implementation to track the speculation of the access instruction. This is however a design choice made by the author of this report, and not something that is described by the inventors of STT. STTs untaint mechanism is also not described very well, resulting in the implemented solution for the untaint mechanism to a large degree is based on assumptions of how it should be performed made by the author.

6.6 Time Management

Most of the frameworks and tools used for this project were not familiar to the author at the start of the project. Therefore, in the start of the project period, a lot of time were spent on becoming familiar with the necessary tools. This gave less time for the actual implementation and time to write the report.

6.7 Future Work

The DoM mitigation strategy can also be extended with value prediction for the delayed loads. This is an optimization described by the inventors of the DoM mitigation strategy [1]. It would be interesting to implement this and perform the same experiments as for the other implementations in this project and compare the result with the other DoM implementation in addition to a finished STT implementation.

The mitigation strategies could also be implemented on other cores to give more insight on a general slowdown of the mitigation strategies and not only the slowdown in the BOOM core.

It is also as mentioned not part of the scope to conclude about why the result acquired is what it is, but this is something that future research can perform experiments to find out. It would for example be interesting to find out why the performance of the DoM-RW implementation is poorer than the DoM implementation.

Chapter 7

Conclusion

This work has evaluated the speculative side-channel attack mitigation strategies *Delay-on-Miss* (DoM), Delay-on-Miss with restricted wakeups (DoM-RW), and *Speculative Taint Tracking* (STT), implemented in the *Berkeley Out-of-Order Machine* (BOOM) processor core.

Experiments have been performed on an FPGA to test the mitigation strategies for performance, implementation overhead, and protection against speculative side-channel attacks. The performance is tested by running the SPEC2017 benchmark suite. The security is tested by performing attacks of Spectre Variant 1 and Variant 2. The implementation overhead is measured by looking at the utilization of the FPGA.

Although the DoM and DoM-RW implementation are completed, the STT implementation did not get fully completed before the deadline of the project. Most of the parts needed for the strategy are finished, but the taint propagation is not yet completed. This is on the contrary an important part for the strategy to work as expected. The result obtained for this implementation are thus not representative for a finished STT implementation. The experiments, where result can be obtained for the STT implementation implemented, are however still included in the report as it may be interesting to see how STT perform without the complete propagation logic competed.

The DoM and DoM-RW are proven to be secure against the speculative side-channel attacks Spectre Variant 1 and Spectre Variant 2. The partially implemented STT implementation however is not proven to be secure against these attacks. Both the DoM attacks also get a slowdown compared to the baseline implementation. For the normal DoM implementation, this slowdown is 14%, and for DoM-RW it is 20.9%. STT are not tested for performance, as it is unable to run on the FPGA. The implementation overhead of both the DoM implementations are 0.1% increase in number of look-up-tables (LUTs) used for logic, and 0.02% increase in number of flip-flops used, compared to the baseline implementation. For STT, the partially implemented solution have an implementation overhead of 1.2% increase in number of LUTs used for logic, and 0.49% in the number of flip-flops used.

It can be concluded that the DoM and DoM-RW succeeds in mitigating attacks of Spectre Variant 1 and 2, but more tests have to be performed to conclude whether it succeeds to mitigate other than these. It is not enough valid data to conclude anything about STT. It can also be concluded that the implementation overhead is small for all the mitigation strategies, and can be considered negligible. The slowdown on the other hand are not negligible for the DoM and DoM-RW implementations. There are also not enough valid data to conclude anything regarding STT when it comes to performance. Lastly, it can be concluded that the implementation of DoM with relaxed wakeups is preferred over the implementation with restricted wakeups.

Bibliography

[1] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean and M. Själander, 'Efficient Invisible Speculative Execution through Selective Delay and Value Prediction,' in *ACM/IEEE 46th Annual International Symposium on Computer Architecture*, 2019, pp. 723–735.

[2] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas and C. W. Fletcher, 'Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data,' in *52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Association for Computing Machinery, 2019, pp. 954–968.

[3] C. P. Celio, 'A Highly Productive Implementation of an Out-of-Order Processor Generator,' Ph.D. dissertation, University of California, Berkeley, 2017.

[4] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous and A. R. LeBlanc, 'Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions,' *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[5] J. A. Lauvdal, 'Exploring Efficient Speculation Tracking and Speculative Side-Channel Protection in the BOOM Core,' 2022.

[6] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Amsterdam: Morgan Kaufmann, 2012, ISBN: 978-0-12-383872-8.

[7] R. M. Tomasulo, 'An Efficient Algorithm for Exploiting Multiple Arithmetic Units,' *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[8] D. Anderson, F. Sparacio and R. M. Tomasulo, 'The IBM System/360 model 91: Machine Philosophy and Instruction-Handling,' *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 8–24, 1967.

[9] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz and Y. Yarom, 'Spectre Attacks: Exploiting Speculative Execution,' in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[10] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin and D. Gruss, 'A Systematic Evaluation of Transient Execution Attacks and Defenses,' in *Proceedings of the 28th USENIX Conference on Security Symposium*, USENIX Association, 2019, pp. 249–266.

[11] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom and M. Hamburg, 'Meltdown: Reading Kernel Memory from User Space,' in *27th USENIX Security Symposium*, 2018.

[12] H. Shacham, 'The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86),' in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery, 2007, pp. 552–561.

[13] *1528 - speculative execution, variant 4: Speculative store bypass - project-zero*, `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`, (Accessed on 06/28/2023).

[14] *Retpoline: A software construct for preventing branch-target-injection*, `https://support.google.com/faqs/answer/7625886`, (Accessed on 06/28/2023).

[15] C. Sakalis, M. Alipour, A. Ros, A. Jimborean, S. Kaxiras and M. Själander, 'Ghost Loads: What is the Cost of Invisible Speculation?' In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, Association for Computing Machinery, 2019, pp. 153–163.

[16] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher and J. Torrellas, 'InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy,' in *51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018, pp. 428–441.

[17] Z. N. Zhao, H. Ji, M. Yan, J. Yu, C. W. Fletcher, A. Morrison, D. Marinov and J. Torrellas, 'Speculation Invariance (Invarspec): Faster Safe Execution Through Program Analysis,' in *53rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2020, pp. 1138–1152.

[18] P. Aimoniotis, C. Sakalis, M. Själander and S. Kaxiras, 'Reorder Buffer Contention: A Forward Speculative Interference Attack for Speculation Invariant Instructions,' *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 162–165, 2021.

[19] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović and B. Nikolić, 'Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs,' *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[20] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo and A. Waterman, 'The Rocket Chip Generator,' EECS Department, University of California, Berkeley, Tech. Rep., Apr. 2016.

[21] *Verilator Internals #1:Debug, Stages, AstNode, V3Graph*, `https://veripool.org/papers/Verilator_Internals1_202010.pdf`.

[22] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach and K. Asanovic, 'Firesim: FPGA - Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud,' in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 29–42.

[23] Firemarshal, *FireMarshal Documentation — FireMarshal 1.6 documentation*, `https://firemarshal.readthedocs.io/en/latest/index.html`, (Accessed on 04/25/2023).

[24] *Buildroot - Making Embedded Linux Easy*, `https://buildroot.org/`, (Accessed on 05/26/2023).

[25] *Architectures/RISC-V - Fedora Project Wiki*, `https://fedoraproject.org/wiki/Architectures/RISC-V`, (Accessed on 05/26/2023).

[26] *Welcome to RISCV-BOOM's documentation! — RISCV-BOOM documentation*, `https://docs.boom-core.org/en/latest/`, (Accessed on 12/12/2022).

[27] J. Zhao, B. Korpan, A. Gonzalez and K. Asanovic, 'SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine,' in *Fourth Workshop on Computer Architecture Research with RISC-V*, May 2020.

[28] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek and K. Asanović, 'Chisel: Constructing Hardware in a Scala Embedded Language,' in *DAC Design Automation Conference 2012*, 2012, pp. 1212–1221.

[29] K. Asanović and D. A. Patterson, 'Instruction Sets Should Be Free: The Case For RISC-V,' EECS Department, University of California, Berkeley, Tech. Rep., Aug. 2014.

[30] *The BOOM Pipeline — RISCV-BOOM documentation*, `https://docs.boom-core.org/en/latest/sections/intro-overview/boom-pipeline.html`, (Accessed on 05/25/2023).

[31] J. A. Lauvdal, *Johnal18/master-project: Implementation of the speculative side channel attack mitigation strategies "Delay on Miss" and "Speculative Taint tracking" on the BOOM Core*, `https://github.com/johnal18/master-project`, (Accessed on 04/29/2023).

[32] *Vec - chisel_2.13 6.0.0-m2 javadoc*, https://javadoc.io/doc/org.chipsalliance/chisel_2.13/latest/chisel3/Vec.html#map[B](f:A=%3EB):CC[B], (Accessed on 06/26/2023).

[33] M. Själander, M. Jahre, G. Tufte and N. Reissmann, 'EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure,' Feb. 2022. arXiv: 1912.05848.

[34] *Idun – High Performance Computing Group*, https://www.hpc.ntnu.no/idun/, (Accessed on 06/03/2023).

[35] *Alveo U250 Data Center Accelerator Card*, https://www.xilinx.com/products/boards-and-kits/alveo/u250.html, (Accessed on 06/03/2023).

[36] *EECS-NTNU/firesim: FireSim: Easy-to-use, Scalable, FPGA-accelerated Cycle-accurate Hardware Simulation in the Cloud*, https://github.com/EECS-NTNU/firesim, (Accessed on 06/03/2023).

[37] *EECS-NTNU/chipyard: Pre-release starter template for custom Chisel projects*, https://github.com/EECS-NTNU/chipyard, (Accessed on 06/03/2023).

[38] *U250_firesim · EECS-NTNU/chipyard Wiki*, https://github.com/EECS-NTNU/chipyard/wiki/u250_firesim, (Accessed on 06/03/2023).

[39] *EECS-NTNU/FireMarshal: Automatically Builds a Linux Distribution for Firesim/FireChip Nodes, works with Firesim's automatic workload generation.* https://github.com/EECS-NTNU/FireMarshal, (Accessed on 06/03/2023).

[40] *PythonTools/invoke.py at master bgottschall/pythonTools*, https://github.com/bgottschall/pythonTools/blob/master/invoke.py, (Accessed on 06/04/2023).

[41] *Tutorial - Perf Wiki*, https://perf.wiki.kernel.org/index.php/Tutorial#Introduction, (Accessed on 06/04/2023).

[42] *SPEC CPU®2017*, https://www.spec.org/cpu2017/, (Accessed on 06/04/2023).

[43] *Riscv-boom/boom-attacks: Proof of concepts for speculative attacks using the BOOM core (https://github.com/riscv-boom/riscv-boom)*, https://github.com/riscv-boom/boom-attacks, (Accessed on 12/11/2022).

Appendix A

Spectre Variant 1 Results

A.1 Baseline

```
m[0x0x1c450] = want(!) =?= guess(hits,dec,char) 1.(10, 33, !) 2.(1, 1, )
m[0x0x1c451] = want(") =?= guess(hits,dec,char) 1.(9, 34, ") 2.(2, 1, )
m[0x0x1c452] = want(#) =?= guess(hits,dec,char) 1.(10, 35, #) 2.(3, 90, Z)
m[0x0x1c453] = want(T) =?= guess(hits,dec,char) 1.(10, 84, T) 2.(3, 7, )
m[0x0x1c454] = want(h) =?= guess(hits,dec,char) 1.(9, 104, h) 2.(2, 8,)
m[0x0x1c455] = want(i) =?= guess(hits,dec,char) 1.(8, 105, i) 2.(3, 252, |)
m[0x0x1c456] = want(s) =?= guess(hits,dec,char) 1.(9, 115, s) 2.(2, 10,
)
m[0x0x1c457] = want(I) =?= guess(hits,dec,char) 1.(7, 73, I) 2.(2, 83, S)
m[0x0x1c458] = want(s) =?= guess(hits,dec,char) 1.(7, 115, s) 2.(4, 7, )
m[0x0x1c459] = want(T) =?= guess(hits,dec,char) 1.(9, 84, T) 2.(2, 10,
)
m[0x0x1c45a] = want(h) =?= guess(hits,dec,char) 1.(9, 104, h) 2.(2, 3, )
m[0x0x1c45b] = want(e) =?= guess(hits,dec,char) 1.(9, 101, e) 2.(2, 1, )
m[0x0x1c45c] = want(B) =?= guess(hits,dec,char) 1.(7, 66, B) 2.(2, 122, z)
m[0x0x1c45d] = want(a) =?= guess(hits,dec,char) 1.(7, 97, a) 2.(3, 75, K)
m[0x0x1c45e] = want(b) =?= guess(hits,dec,char) 1.(9, 98, b) 2.(2, 7, )
m[0x0x1c45f] = want(y) =?= guess(hits,dec,char) 1.(8, 121, y) 2.(2, 7, )
m[0x0x1c460] = want(B) =?= guess(hits,dec,char) 1.(10, 66, B) 2.(2, 7, )
m[0x0x1c461] = want(o) =?= guess(hits,dec,char) 1.(9, 111, o) 2.(3, 83, S)
m[0x0x1c462] = want(o) =?= guess(hits,dec,char) 1.(10, 111, o) 2.(2, 28, )
m[0x0x1c463] = want(m) =?= guess(hits,dec,char) 1.(7, 109, m) 2.(2, 6, )
m[0x0x1c464] = want(e) =?= guess(hits,dec,char) 1.(7, 101, e) 2.(2, 3, )
m[0x0x1c465] = want(r) =?= guess(hits,dec,char) 1.(10, 114, r) 2.(2, 5, )
m[0x0x1c466] = want(T) =?= guess(hits,dec,char) 1.(8, 84, T) 2.(2, 3, )
m[0x0x1c467] = want(e) =?= guess(hits,dec,char) 1.(8, 101, e) 2.(2, 204, L)
m[0x0x1c468] = want(s) =?= guess(hits,dec,char) 1.(8, 115, s) 2.(2, 1, )
m[0x0x1c469] = want(t) =?= guess(hits,dec,char) 1.(9, 116, t) 2.(2, 0, )
```

A.2 Delay-on-Miss

```
m[0x0x1c450] = want(!) =?= guess(hits,dec,char) 1.(2, 7, ) 2.(2, 10,
)
m[0x0x1c451] = want(") =?= guess(hits,dec,char) 1.(3, 252, |) 2.(2, 97, a)
m[0x0x1c452] = want(#) =?= guess(hits,dec,char) 1.(2, 62, >) 2.(2, 91, [)
m[0x0x1c453] = want(T) =?= guess(hits,dec,char) 1.(2, 1, ) 2.(2, 7, )
m[0x0x1c454] = want(h) =?= guess(hits,dec,char) 1.(2, 5, ) 2.(2, 46, .)
m[0x0x1c455] = want(i) =?= guess(hits,dec,char) 1.(2, 1, ) 2.(2, 19, )
m[0x0x1c456] = want(s) =?= guess(hits,dec,char) 1.(2, 5, ) 2.(2, 12,
                                                                     )
m[0x0x1c457] = want(I) =?= guess(hits,dec,char) 1.(2, 67, C) 2.(2, 176, 0)
m[0x0x1c458] = want(s) =?= guess(hits,dec,char) 1.(2, 5, ) 2.(2, 157, )
m[0x0x1c459] = want(T) =?= guess(hits,dec,char) 1.(2, 3, ) 2.(2, 144, )
m[0x0x1c45a] = want(h) =?= guess(hits,dec,char) 1.(2, 1, ) 2.(2, 100, d)
m[0x0x1c45b] = want(e) =?= guess(hits,dec,char) 1.(2, 5, ) 2.(2, 6, )
m[0x0x1c45c] = want(B) =?= guess(hits,dec,char) 1.(2, 111, o) 2.(2, 124, |)
m[0x0x1c45d] = want(a) =?= guess(hits,dec,char) 1.(2, 7, ) 2.(2, 10,
)
m[0x0x1c45e] = want(b) =?= guess(hits,dec,char) 1.(2, 7, ) 2.(2, 60, <)
m[0x0x1c45f] = want(y) =?= guess(hits,dec,char) 1.(3, 5, ) 2.(2, 3, )
m[0x0x1c460] = want(B) =?= guess(hits,dec,char) 1.(2, 26, ) 2.(1, 1, )
m[0x0x1c461] = want(o) =?= guess(hits,dec,char) 1.(2, 112, p) 2.(2, 153, )
m[0x0x1c462] = want(o) =?= guess(hits,dec,char) 1.(3, 7, ) 2.(3, 180, 4)
m[0x0x1c463] = want(m) =?= guess(hits,dec,char) 1.(2, 60, <) 2.(2, 110, n)
m[0x0x1c464] = want(e) =?= guess(hits,dec,char) 1.(1, 1, ) 2.(1, 2, )
m[0x0x1c465] = want(r) =?= guess(hits,dec,char) 1.(3, 3, ) 2.(2, 8,)
m[0x0x1c466] = want(T) =?= guess(hits,dec,char) 1.(3, 100, d) 2.(2, 7, )
m[0x0x1c467] = want(e) =?= guess(hits,dec,char) 1.(2, 7, ) 2.(2, 36, \$)
m[0x0x1c468] = want(s) =?= guess(hits,dec,char) 1.(1, 1, ) 2.(1, 2, )
m[0x0x1c469] = want(t) =?= guess(hits,dec,char) 1.(2, 163, #) 2.(2, 172, ,)
```

A.3 Delay-on-Miss Restricted Wakeups

```
m[0x0x1c450] = want(!) =?= guess(hits,dec,char) 1.(2, 2, ) 2.(2, 6, )
m[0x0x1c451] = want(") =?= guess(hits,dec,char) 1.(2, 157, ) 2.(1, 1, )
m[0x0x1c452] = want(#) =?= guess(hits,dec,char) 1.(2, 5, ) 2.(2, 6, )
m[0x0x1c453] = want(T) =?= guess(hits,dec,char) 1.(2, 110, n) 2.(2, 206, N)
m[0x0x1c454] = want(h) =?= guess(hits,dec,char) 1.(2, 1, ) 2.(2, 10,
)
m[0x0x1c455] = want(i) =?= guess(hits,dec,char) 1.(2, 9,        ) 2.(2, 193, A)
m[0x0x1c456] = want(s) =?= guess(hits,dec,char) 1.(2, 3, ) 2.(2, 62, >)
m[0x0x1c457] = want(I) =?= guess(hits,dec,char) 1.(2, 9,        ) 2.(2, 232, h)
m[0x0x1c458] = want(s) =?= guess(hits,dec,char) 1.(2, 4, ) 2.(2, 5, )
m[0x0x1c459] = want(T) =?= guess(hits,dec,char) 1.(2, 5, ) 2.(2, 165, \%)
m[0x0x1c45a] = want(h) =?= guess(hits,dec,char) 1.(2, 3, ) 2.(2, 29, )
m[0x0x1c45b] = want(e) =?= guess(hits,dec,char) 1.(2, 5, ) 2.(2, 18, )
m[0x0x1c45c] = want(B) =?= guess(hits,dec,char) 1.(2, 150, ) 2.(2, 252, |)
m[0x0x1c45d] = want(a) =?= guess(hits,dec,char) 1.(2, 6, ) 2.(2, 49, 1)
m[0x0x1c45e] = want(b) =?= guess(hits,dec,char) 1.(2, 1, ) 2.(2, 4, )
m[0x0x1c45f] = want(y) =?= guess(hits,dec,char) 1.(2, 8,) 2.(1, 1, )
m[0x0x1c460] = want(B) =?= guess(hits,dec,char) 1.(4, 2, ) 2.(2, 5, )
m[0x0x1c461] = want(o) =?= guess(hits,dec,char) 1.(2, 2, ) 2.(2, 10,
)
m[0x0x1c462] = want(o) =?= guess(hits,dec,char) 1.(3, 193, A) 2.(2, 14, )
m[0x0x1c463] = want(m) =?= guess(hits,dec,char) 1.(2, 4, ) 2.(2, 8,)
m[0x0x1c464] = want(e) =?= guess(hits,dec,char) 1.(2, 60, <) 2.(1, 1, )
m[0x0x1c465] = want(r) =?= guess(hits,dec,char) 1.(2, 2, ) 2.(2, 3, )
m[0x0x1c466] = want(T) =?= guess(hits,dec,char) 1.(2, 8,) 2.(2, 77, M)
m[0x0x1c467] = want(e) =?= guess(hits,dec,char) 1.(2, 9,        ) 2.(2, 14, )
m[0x0x1c468] = want(s) =?= guess(hits,dec,char) 1.(2, 9,        ) 2.(2, 161, !)
m[0x0x1c469] = want(t) =?= guess(hits,dec,char) 1.(2, 5, ) 2.(2, 6, )
```

A.4 Speculative Taint Tracking

This is the result from Speculative Taint Tracking with only load-load dependencies implemented. The result is not complete since the program terminates.

```
m[0x0x80002750] = want(!) =?= guess(hits,dec,char) 1.(10, 33, !) 2.(1, 1, )
m[0x0x80002751] = want(") =?= guess(hits,dec,char) 1.(10, 34, ") 2.(1, 1, )
m[0x0x80002752] = want(#) =?= guess(hits,dec,char) 1.(10, 35, #) 2.(1, 1, )
m[0x0x80002753] = want(T) =?= guess(hits,dec,char) 1.(7, 84, T) 2.(1, 1, )
m[0x0x80002754] = want(h) =?= guess(hits,dec,char) 1.(6, 104, h) 2.(1, 1, )
m[0x0x80002755] = want(i) =?= guess(hits,dec,char) 1.(8, 105, i) 2.(1, 1, )
m[0x0x80002756] = want(s) =?= guess(hits,dec,char) 1.(9, 115, s) 2.(1, 1, )
m[0x0x80002757] = want(I) =?= guess(hits,dec,char) 1.(8, 73, I) 2.(1, 1, )
make: *** [/lhome/johnala/chipyard/common.mk:206: run-binary] Error 2
```

Appendix B

Spectre Variant 2 Results

B.1 Baseline

```
m[0x0x1c480] = want(!) =?= guess(hits,dec,char) 1.(10, 33, !) 2.(3, 178, 2)
m[0x0x1c481] = want(") =?= guess(hits,dec,char) 1.(8, 34, ") 2.(2, 2, )
m[0x0x1c482] = want(#) =?= guess(hits,dec,char) 1.(10, 35, #) 2.(2, 3, )
m[0x0x1c483] = want(T) =?= guess(hits,dec,char) 1.(9, 84, T) 2.(2, 231, g)
m[0x0x1c484] = want(h) =?= guess(hits,dec,char) 1.(9, 104, h) 2.(3, 51, 3)
m[0x0x1c485] = want(i) =?= guess(hits,dec,char) 1.(7, 105, i) 2.(2, 8,)
m[0x0x1c486] = want(s) =?= guess(hits,dec,char) 1.(10, 115, s) 2.(2, 52, 4)
m[0x0x1c487] = want(I) =?= guess(hits,dec,char) 1.(9, 73, I) 2.(2, 5, )
m[0x0x1c488] = want(s) =?= guess(hits,dec,char) 1.(8, 115, s) 2.(2, 4, )
m[0x0x1c489] = want(T) =?= guess(hits,dec,char) 1.(8, 84, T) 2.(2, 8,)
m[0x0x1c48a] = want(h) =?= guess(hits,dec,char) 1.(6, 104, h) 2.(1, 0, )
m[0x0x1c48b] = want(e) =?= guess(hits,dec,char) 1.(9, 101, e) 2.(2, 0, )
m[0x0x1c48c] = want(B) =?= guess(hits,dec,char) 1.(10, 66, B) 2.(2, 1, )
m[0x0x1c48d] = want(a) =?= guess(hits,dec,char) 1.(7, 97, a) 2.(2, 35, #)
m[0x0x1c48e] = want(b) =?= guess(hits,dec,char) 1.(8, 98, b) 2.(2, 4, )
m[0x0x1c48f] = want(y) =?= guess(hits,dec,char) 1.(7, 121, y) 2.(2, 8,)
m[0x0x1c490] = want(B) =?= guess(hits,dec,char) 1.(10, 66, B) 2.(2, 6, )
m[0x0x1c491] = want(o) =?= guess(hits,dec,char) 1.(8, 111, o) 2.(2, 1, )
m[0x0x1c492] = want(o) =?= guess(hits,dec,char) 1.(9, 111, o) 2.(2, 1, )
m[0x0x1c493] = want(m) =?= guess(hits,dec,char) 1.(10, 109, m) 2.(2, 159, )
m[0x0x1c494] = want(e) =?= guess(hits,dec,char) 1.(9, 101, e) 2.(3, 169, ))
m[0x0x1c495] = want(r) =?= guess(hits,dec,char) 1.(10, 114, r) 2.(2, 1, )
m[0x0x1c496] = want(T) =?= guess(hits,dec,char) 1.(9, 84, T) 2.(2, 6, )
m[0x0x1c497] = want(e) =?= guess(hits,dec,char) 1.(10, 101, e) 2.(2, 15, )
m[0x0x1c498] = want(s) =?= guess(hits,dec,char) 1.(7, 115, s) 2.(2, 63, ?)
m[0x0x1c499] = want(t) =?= guess(hits,dec,char) 1.(10, 116, t) 2.(2, 8,)
```

B.2 Delay-on-Miss

```
m[0x0x1c480] = want(!) =?= guess(hits,dec,char) 1.(2, 5, ) 2.(2, 142, )
m[0x0x1c481] = want(") =?= guess(hits,dec,char) 1.(2, 2, ) 2.(1, 1, )
m[0x0x1c482] = want(#) =?= guess(hits,dec,char) 1.(2, 5, ) 2.(2, 6, )
m[0x0x1c483] = want(T) =?= guess(hits,dec,char) 1.(2, 1, ) 2.(2, 2, )
m[0x0x1c484] = want(h) =?= guess(hits,dec,char) 1.(2, 4, ) 2.(2, 53, 5)
m[0x0x1c485] = want(i) =?= guess(hits,dec,char) 1.(2, 30, ) 2.(2, 142, )
m[0x0x1c486] = want(s) =?= guess(hits,dec,char) 1.(2, 0, ) 2.(2, 22, )
m[0x0x1c487] = want(I) =?= guess(hits,dec,char) 1.(2, 66, B) 2.(2, 179, 3)
m[0x0x1c488] = want(s) =?= guess(hits,dec,char) 1.(2, 8,) 2.(2, 58, :)
m[0x0x1c489] = want(T) =?= guess(hits,dec,char) 1.(2, 4, ) 2.(2, 104, h)
m[0x0x1c48a] = want(h) =?= guess(hits,dec,char) 1.(3, 88, X) 2.(3, 111, o)
m[0x0x1c48b] = want(e) =?= guess(hits,dec,char) 1.(2, 2, ) 2.(2, 47, /)
m[0x0x1c48c] = want(B) =?= guess(hits,dec,char) 1.(3, 70, F) 2.(2, 1, )
m[0x0x1c48d] = want(a) =?= guess(hits,dec,char) 1.(3, 155, .(2, 9,      )
m[0x0x1c48e] = want(b) =?= guess(hits,dec,char) 1.(2, 27, .(1, 1, )
m[0x0x1c48f] = want(y) =?= guess(hits,dec,char) 1.(2, 2, ) 2.(2, 30, )
m[0x0x1c490] = want(B) =?= guess(hits,dec,char) 1.(2, 5, ) 2.(2, 47, /)
m[0x0x1c491] = want(o) =?= guess(hits,dec,char) 1.(2, 7, ) 2.(2, 48, 0)
m[0x0x1c492] = want(o) =?= guess(hits,dec,char) 1.(2, 3, ) 2.(2, 6, )
m[0x0x1c493] = want(m) =?= guess(hits,dec,char) 1.(1, 1, ) 2.(1, 2, )
m[0x0x1c494] = want(e) =?= guess(hits,dec,char) 1.(2, 7, ) 2.(2, 21, )
m[0x0x1c495] = want(r) =?= guess(hits,dec,char) 1.(3, 124, |) 2.(3, 201, I)
m[0x0x1c496] = want(T) =?= guess(hits,dec,char) 1.(2, 137,      ) 2.(2, 142, )
m[0x0x1c497] = want(e) =?= guess(hits,dec,char) 1.(2, 1, ) 2.(2, 46, .)
m[0x0x1c498] = want(s) =?= guess(hits,dec,char) 1.(2, 63, ?) 2.(2, 69, E)
m[0x0x1c499] = want(t) =?= guess(hits,dec,char) 1.(2, 4, ) 2.(2, 166, &)
```

B.3 Delay-on-Miss Restricted Wakeups

```
m[0x0x1c480] = want(!) =?= guess(hits,dec,char) 1.(2, 1, ) 2.(2, 9,     )
m[0x0x1c481] = want(") =?= guess(hits,dec,char) 1.(2, 69, E) 2.(2, 193, A)
m[0x0x1c482] = want(#) =?= guess(hits,dec,char) 1.(2, 8,) 2.(2, 84, T)
m[0x0x1c483] = want(T) =?= guess(hits,dec,char) 1.(2, 1, ) 2.(2, 5, )
m[0x0x1c484] = want(h) =?= guess(hits,dec,char) 1.(3, 164, \$) 2.(2, 2, )
m[0x0x1c485] = want(i) =?= guess(hits,dec,char) 1.(2, 170, *) 2.(1, 1, )
m[0x0x1c486] = want(s) =?= guess(hits,dec,char) 1.(2, 6, ) 2.(2, 227, c)
m[0x0x1c487] = want(I) =?= guess(hits,dec,char) 1.(2, 1, ) 2.(2, 2, )
m[0x0x1c488] = want(s) =?= guess(hits,dec,char) 1.(2, 17, ) 2.(2, 23, )
m[0x0x1c489] = want(T) =?= guess(hits,dec,char) 1.(2, 2, ) 2.(2, 48, 0)
m[0x0x1c48a] = want(h) =?= guess(hits,dec,char) 1.(3, 4, ) 2.(3, 219, [)
m[0x0x1c48b] = want(e) =?= guess(hits,dec,char) 1.(1, 1, ) 2.(1, 2, )
m[0x0x1c48c] = want(B) =?= guess(hits,dec,char) 1.(1, 0, ) 2.(1, 1, )
m[0x0x1c48d] = want(a) =?= guess(hits,dec,char) 1.(3, 29, ) 2.(2, 4, )
m[0x0x1c48e] = want(b) =?= guess(hits,dec,char) 1.(3, 132, ) 2.(2, 5, )
m[0x0x1c48f] = want(y) =?= guess(hits,dec,char) 1.(3, 97, a) 2.(2, 6, )
m[0x0x1c490] = want(B) =?= guess(hits,dec,char) 1.(2, 2, ) 2.(2, 4, )
m[0x0x1c491] = want(o) =?= guess(hits,dec,char) 1.(2, 3, ) 2.(2, 7, )
m[0x0x1c492] = want(o) =?= guess(hits,dec,char) 1.(2, 29, ) 2.(2, 163, #)
m[0x0x1c493] = want(m) =?= guess(hits,dec,char) 1.(2, 8,) 2.(2, 10,
)
m[0x0x1c494] = want(e) =?= guess(hits,dec,char) 1.(2, 9,        ) 2.(2, 48, 0)
m[0x0x1c495] = want(r) =?= guess(hits,dec,char) 1.(2, 20, ) 2.(2, 34, ")
m[0x0x1c496] = want(T) =?= guess(hits,dec,char) 1.(2, 9,        ) 2.(2, 171, +)
m[0x0x1c497] = want(e) =?= guess(hits,dec,char) 1.(2, 4, ) 2.(2, 33, !)
m[0x0x1c498] = want(s) =?= guess(hits,dec,char) 1.(2, 17, ) 2.(2, 66, B)
m[0x0x1c499] = want(t) =?= guess(hits,dec,char) 1.(2, 88, X) 2.(2, 138,
)
```

B.4 Speculative Taint Tracking

This is the result from Speculative Taint Tracking with only load-load dependencies implemented. The result is not complete since the program terminates.

```
m[0x0x80002770] = want(!) =?= guess(hits,dec,char) 1.(10, 33, !) 2.(1, 1, )
m[0x0x80002771] = want(") =?= guess(hits,dec,char) 1.(10, 34, ") 2.(1, 1, )
m[0x0x80002772] = want(#) =?= guess(hits,dec,char) 1.(10, 35, #) 2.(1, 1, )
m[0x0x80002773] = want(T) =?= guess(hits,dec,char) 1.(7, 84, T) 2.(1, 1, )
m[0x0x80002774] = want(h) =?= guess(hits,dec,char) 1.(6, 104, h) 2.(1, 1, )
m[0x0x80002775] = want(i) =?= guess(hits,dec,char) 1.(6, 105, i) 2.(1, 1, )
m[0x0x80002776] = want(s) =?= guess(hits,dec,char) 1.(8, 115, s) 2.(1, 1, )
m[0x0x80002777] = want(I) =?= guess(hits,dec,char) 1.(7, 73, I) 2.(1, 1, )
make: *** [/lhome/johnala/chipyard/common.mk:206: run-binary] Error 2
```

Appendix C

Implementation Overhead Result

C.1 Baseline

```
+----------------------------+--------+-------+------------+-----------+-------+
|          Site Type         |  Used  | Fixed | Prohibited | Available | Util% |
+----------------------------+--------+-------+------------+-----------+-------+
| CLB LUTs                   | 338268 |     0 |          0 |   1728000 | 19.58 |
|   LUT as Logic             | 321053 |     0 |          0 |   1728000 | 18.58 |
|   LUT as Memory            |  17215 |     0 |          0 |    791040 |  2.18 |
|     LUT as Distributed RAM |  15415 |     0 |            |           |       |
|     LUT as Shift Register  |   1800 |     0 |            |           |       |
| CLB Registers              | 202236 |     2 |          0 |   3456000 |  5.85 |
|   Register as Flip Flop    | 202235 |     2 |          0 |   3456000 |  5.85 |
|   Register as Latch        |      0 |     0 |          0 |   3456000 |  0.00 |
|   Register as AND/OR       |      1 |     0 |          0 |   3456000 | <0.01 |
| CARRY8                     |   2817 |     0 |          0 |    216000 |  1.30 |
| F7 Muxes                   |  21120 |     0 |          0 |    864000 |  2.44 |
| F8 Muxes                   |   6667 |     0 |          0 |    432000 |  1.54 |
| F9 Muxes                   |      0 |     0 |          0 |    216000 |  0.00 |
+----------------------------+--------+-------+------------+-----------+-------+
```

C.2 Delay-on-Miss

```
+----------------------------+--------+-------+------------+-----------+-------+
|          Site Type         |  Used  | Fixed | Prohibited | Available | Util% |
+----------------------------+--------+-------+------------+-----------+-------+
| CLB LUTs                   | 338589 |     0 |          0 |   1728000 | 19.59 |
|   LUT as Logic             | 321374 |     0 |          0 |   1728000 | 18.60 |
|   LUT as Memory            |  17215 |     0 |          0 |    791040 |  2.18 |
|     LUT as Distributed RAM |  15415 |     0 |            |           |       |
|     LUT as Shift Register  |   1800 |     0 |            |           |       |
| CLB Registers              | 202267 |     2 |          0 |   3456000 |  5.85 |
|   Register as Flip Flop    | 202266 |     2 |          0 |   3456000 |  5.85 |
|   Register as Latch        |      0 |     0 |          0 |   3456000 |  0.00 |
|   Register as AND/OR       |      1 |     0 |          0 |   3456000 | <0.01 |
| CARRY8                     |   2817 |     0 |          0 |    216000 |  1.30 |
| F7 Muxes                   |  20992 |     0 |          0 |    864000 |  2.43 |
| F8 Muxes                   |   6645 |     0 |          0 |    432000 |  1.54 |
| F9 Muxes                   |      0 |     0 |          0 |    216000 |  0.00 |
+----------------------------+--------+-------+------------+-----------+-------+
```

C.3 Delay-on-Miss Restricted Wakeups

```
+----------------------------+--------+-------+------------+-----------+-------+
|          Site Type         |  Used  | Fixed | Prohibited | Available | Util% |
+----------------------------+--------+-------+------------+-----------+-------+
| CLB LUTs                   | 338589 |     0 |          0 |   1728000 | 19.59 |
|   LUT as Logic             | 321374 |     0 |          0 |   1728000 | 18.60 |
|   LUT as Memory            |  17215 |     0 |          0 |    791040 |  2.18 |
|     LUT as Distributed RAM |  15415 |     0 |            |           |       |
|     LUT as Shift Register  |   1800 |     0 |            |           |       |
| CLB Registers              | 202267 |     2 |          0 |   3456000 |  5.85 |
|   Register as Flip Flop    | 202266 |     2 |          0 |   3456000 |  5.85 |
|   Register as Latch        |      0 |     0 |          0 |   3456000 |  0.00 |
|   Register as AND/OR       |      1 |     0 |          0 |   3456000 | <0.01 |
| CARRY8                     |   2817 |     0 |          0 |    216000 |  1.30 |
| F7 Muxes                   |  20992 |     0 |          0 |    864000 |  2.43 |
| F8 Muxes                   |   6645 |     0 |          0 |    432000 |  1.54 |
| F9 Muxes                   |      0 |     0 |          0 |    216000 |  0.00 |
+----------------------------+--------+-------+------------+-----------+-------+
```

C.4 Speculative Taint Tracking

```
+----------------------------+--------+-------+------------+-----------+-------+
|          Site Type         |  Used  | Fixed | Prohibited | Available | Util% |
+----------------------------+--------+-------+------------+-----------+-------+
| CLB LUTs                   | 342128 |     0 |          0 |   1728000 | 19.80 |
|   LUT as Logic             | 324913 |     0 |          0 |   1728000 | 18.80 |
|   LUT as Memory            |  17215 |     0 |          0 |    791040 |  2.18 |
|     LUT as Distributed RAM |  15415 |     0 |            |           |       |
|     LUT as Shift Register  |   1800 |     0 |            |           |       |
| CLB Registers              | 203217 |     2 |          0 |   3456000 |  5.88 |
|   Register as Flip Flop    | 203216 |     2 |          0 |   3456000 |  5.88 |
|   Register as Latch        |      0 |     0 |          0 |   3456000 |  0.00 |
|   Register as AND/OR       |      1 |     0 |          0 |   3456000 | <0.01 |
| CARRY8                     |   2831 |     0 |          0 |    216000 |  1.31 |
| F7 Muxes                   |  21377 |     0 |          0 |    864000 |  2.47 |
| F8 Muxes                   |   6772 |     0 |          0 |    432000 |  1.57 |
| F9 Muxes                   |      0 |     0 |          0 |    216000 |  0.00 |
+----------------------------+--------+-------+------------+-----------+-------+
```

Appendix D

Additional Material

Masteravtale/hovedoppgaveavtale

Sist oppdatert 11. november 2020

Fakultet	**Fakultet for informasjonsteknologi og elektroteknikk**
Institutt	**Institutt for datateknologi og informatikk**
Studieprogram	**MIDT**
Emnekode	**TDT4900**

Studenten

Etternavn, fornavn	**Lauvdal, John Askeland**
Fødselsdato	**18.08.1999**
E-postadresse ved NTNU	**johnala@stud.ntnu.no**

Tilknyttede ressurser

Veileder	**Magnus Själander**
Eventuelle medveiledere	**Amund Bergland Kvalsvik**
Eventuelle medstudenter	

Oppgaven

Oppstartsdato	**11.01.2023**
Leveringsfrist	**07.06.2023**
Oppgavens arbeidstittel	**Exploring Efficient Speculation Tracking and Speculative Side-Channel Protection in the BOOM Core**
Problembeskrivelse	**Speculative side-channel attacks are attacks exploiting speculative execution done in modern processors. Mitigating these attacks has been a challenge for computer architects, as speculation is very important for the performance of CPUs. In this project, I want to continue exploring the DOM mitigation strategy, which I started on in the specialization project in the 9th semester. I also want to compare it to other mitigation strategies with regard to performance, implementation overhead and security provided.**

NTNU

Risikovurdering og datahåndtering

Skal det gjennomføres risikovurdering? **Dersom «ja», har det blitt gjennomført?**	**Nei** **Nei**
Skal det søkes om godkjenninger? (REK*, NSD)**	**Nei**
Skal det skrives en konfidensialitetsavtale i forbindelse med oppgaven? **Hvis «ja», har det blitt gjort?**	**Nei** **Nei**

*   Regionale komiteer for medisinsk og helsefaglig forskningsetikk (https://rekportalen.no)
** Norsk senter for forskningsdata (https://nsd.no/)

Eventuelle emner som skal inngå i mastergraden

Retningslinjer - rettigheter og plikter

Formål

Avtale om veiledning av masteroppgaven/hovedoppgaven er en samarbeidsavtale mellom student, veileder og institutt. Avtalen regulerer veiledningsforholdet, omfang, art og ansvarsfordeling.

Studieprogrammet og arbeidet med oppgaven er regulert av Universitets- og høgskoleloven, NTNUs studieforskrift og gjeldende studieplan. Informasjon om emnet, som oppgaven inngår i, finner du i emnebeskrivelsen.

Veiledning

Studenten har ansvar for å

- Avtale veiledningstimer med veileder innenfor rammene master-/hovedoppgaveavtalen gir.
- Utarbeide framdriftsplan for arbeidet i samråd med veileder, inkludert veiledningsplan.
- Holde oversikt over antall brukte veiledningstimer sammen med veileder.
- Gi veileder nødvendig skriftlig materiale i rimelig tid før veiledning.
- Holde instituttet og veileder orientert om eventuelle forsinkelser.
- Inkludere eventuell(e) medstudent(er) i avtalen.

Veileder har ansvar for å

- Avklare forventninger om veiledningsforholdet.
- Sørge for at det søkes om eventuelle nødvendige godkjenninger (etikk, personvernhensyn).
- Gi råd om formulering og avgrensning av tema og problemstilling, slik at arbeidet er gjennomførbart innenfor normert eller avtalt studietid.
- Drøfte og vurdere hypoteser og metoder.
- Gi råd vedrørende faglitteratur, kildemateriale, datagrunnlag, dokumentasjon og eventuelt ressursbehov.
- Drøfte framstillingsform (eksempelvis disposisjon og språklig form).
- Drøfte resultater og tolkninger.
- Holde seg orientert om progresjonen i studentens arbeid i henhold til avtalt tids- og arbeidsplan, og følge opp studenten ved behov.
- Sammen med studenten holde oversikt over antall brukte veiledningstimer.

Instituttet har ansvar for å

- Sørge for at avtalen blir inngått.
- Finne og oppnevne veileder(e).
- Inngå avtale med annet institutt/ fakultet/institusjon dersom det er oppnevnt ekstern medveileder.
- I samarbeid med veileder holde oversikt over studentens framdrift, antall brukte veiledningstimer, og følge opp dersom studenten er forsinket i henhold til avtalen.
- Oppnevne ny veileder og sørge for inngåelse av ny avtale dersom:
  - Veileder blir fraværende på grunn av eksempelvis forskningstermin, sykdom, eller reiser.
  - Student eller veileder ber om å få avslutte avtalen fordi en av partene ikke følger den.
  - Andre forhold gjør at partene finner det hensiktsmessig med ny veileder.
- Gi studenten beskjed når veiledningsforholdet opphører.
- Informere veileder(e) om ansvaret for å ivareta forskningsetiske forhold, personvernhensyn og veiledningsetiske forhold.
- Ønsker student, eller veileder, å bli løst fra avtalen må det søkes til instituttet. Instituttet må i et slikt tilfelle oppnevne ny veileder.

Avtaleskjemaet skal godkjennes når retningslinjene er gjennomgått.

Godkjent av

John Askeland Lauvdal
Student

10.01.2023
Digitalt godkjent

Magnus Själander
Veileder

10.01.2023
Digitalt godkjent

Berit Hellan
Institutt

08.02.2023
Digitalt godkjent

NTNU

Master`s Agreement / Main Thesis Agreement

Faculty	**Faculty of Information Technology and Electrical Engineering**
Institute	**Department of Computer Science**
Programme Code	**MIDT**
Course Code	**TDT4900**

Personal Information

Surname, First Name	**Lauvdal, John Askeland**
Date of Birth	**18.08.1999**
Email	**johnala@stud.ntnu.no**

Supervision and Co-authors

Supervisor	**Magnus Själander**
Co-supervisors (if applicable)	**Amund Bergland Kvalsvik**
Co-authors (if applicable)	

The Master`s thesis

Starting Date	**11.01.2023**
Submission Deadline	**07.06.2023**
Thesis Working Title	**Exploring Efficient Speculation Tracking and Speculative Side-Channel Protection in the BOOM Core**
Problem Description	**Speculative side-channel attacks are attacks exploiting speculative execution done in modern processors. Mitigating these attacks has been a challenge for computer architects, as speculation is very important for the performance of CPUs. In this project, I want to continue exploring the DOM mitigation strategy, which I started on in the specialization project in the 9th semester. I also want to compare it to other mitigation strategies with regard to performance, implementation overhead and security provided.**

Risk Assessment and Data Management

Will you conduct a Risk Assessment? **If "Yes", Is the Risk Assessment Conducted?**	**No** **No**
Will you Apply for Data Management? (REK*, NSD)**	**No**
Will You Write a Confidentiality Agreement? **If "Yes", Is the Confidentiality Agreement Conducted?**	**No** **No**

*   REK -- https://rekportalen.no/

** Norwegian Centre for Research Data (https://nsd.no/nsd/english/index.html)

Topics to be included in the Master`s Degree (if applicable)

Guidelines – Rights and Obligations

Purpose

The Master's Agreement/ Main Thesis Agreement is an agreement between the student, supervisor, and department. The agreement regulates supervision conditions, scope, nature, and responsibilities concerning the thesis.

The study programme and the thesis are regulated by the Universities and University Colleges Act, NTNU's study regulations, and the current curriculum for the study programme.

Supervision

The student is responsible for

- Arranging the supervision within the framework provided by the agreement.
- Preparing a plan of progress in cooperation with the supervisor, including a supervision schedule.
- Keeping track of the counselling hours.
- Providing the supervisor with the necessary written material in a timely manner before the supervision.
- Keeping the institute and supervisor informed of any delays.
- Adding fellow student(s) to the agreement, if the thesis has more than one author.

The supervisor is responsible for

- Clarifying expectations and how the supervision should take place.
- Ensuring that any necessary approvals are acquired (REC, ethics, privacy).
- Advising on the demarcation of the topic and the thesis statement to ensure that the work is feasible within agreed upon time frame.
- Discussing and evaluating hypotheses and methods.
- Advising on literature, source material, data, documentation, and resource requirements.
- Discussing the layout of the thesis with the student (disposition, linguistic form, etcetera).
- Discussing the results and the interpretation of them.
- Staying informed about the work progress and assist the student if necessary.
- Together with the student, keeping track of supervision hours spent.

The institute is responsible for

- Ensuring that the agreement is entered into.
- Find and appoint supervisor(s).
- Enter into an agreement with another department / faculty / institution if there is an external co-supervisor.
- In cooperation with the supervisor, keep an overview of the student's progress, the number of supervision hours. spent, and assist if the student is delayed by appointment.
- Appoint a new supervisor and arrange for a new agreement if:
  - The supervisor will be absent due to research term, illness, travel, etcetera.
  - The student or supervisor requests to terminate the agreement due to lack of adherence from either party.
  - Other circumstances where it is appropriate with a new supervisor.
- Notify the student when the agreement terminates.
- Inform supervisors about the responsibility for safeguarding ethical issues, privacy and guidance ethics
- Should the cooperation between student and supervisor become problematic, either party may apply to the department to be freed from the agreement. In such occurrence, the department must appoint a new supervisor

This Master`s agreement must be signed when the guidelines have been reviewed.

Signatures

John Askeland Lauvdal
Student

10.01.2023
Digitally approved

Magnus Själander
Supervisor

10.01.2023
Digitally approved

Berit Hellan
Department

08.02.2023
Digitally approved