COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Par

Herinomena Henintsoa ANDRIANATREHINA

Mitigating Spectre vulnerability in modern Out-of-Order Cores.

Thèse présentée et soutenue à Rennes, le 12 Janvier 2026

Rapporteurs avant soutenance :

Guy GOGNIAT Professeur des Universités - Université de Bretagne Sud
Yuval YAROM Professeur des Universités - Ruhr University Bochum

Composition du Jury :

Examinateur·trice·s :	Lesly-Ann DANIEL	Maître de Conférences - EURECOM
	Guy GOGNIAT	Professeur des Universités - Université de Bretagne Sud
	David MONNIAUX	Directeur de recherche - CNRS
	Olivier SAVRY	Ingenieur de recherche - CEA
Dir. de thèse :	Olivier SENTIEYS	Professeur des Universités - Université de Rennes
Co-dir. de thèse :	Ronan LASHERMES	Ingenieur de recherche - Rambus

Remerciements

Je tiens à exprimer ma profonde reconnaissance à toutes les personnes qui ont contribué à l'aboutissement de cette thèse. Je remercie celles et ceux qui m'ont soutenu au cours de ces trois années, que ce soit sur les plans scientifique, administratif ou personnel. Chaque soutien m'a apporté la force, les connaissances et la confiance nécessaires pour mener ce travail à son terme.

Je souhaite exprimer ma gratitude à mon directeur de thèse, Ronan LASHERMES, pour la confiance qu'il m'a accordée, ainsi que pour toutes les discussions qui ont profondément façonné ma vision de la recherche, nourri ma réflexion scientifique et contribué à la maturation des idées développées dans ce travail. Je remercie également mes encadrants, Simon ROKICKI et Joseph PATUREL, pour leur grande disponibilité, la qualité de leurs conseils et la pertinence de leur aide, qui m'ont permis de progresser tout au long de ces trois années de recherche. Mes remerciements vont aussi à Thomas RUBIANO, qui a rejoint l'équipe en cours de thèse et a apporté une contribution précieuse, tant aux travaux réalisés qu'à l'aboutissement de ce manuscrit. Je suis reconnaissant envers l'ensemble de mes collaborateurs pour la richesse de nos échanges et leurs retours constructifs, que ce soit sur les travaux, les publications, les présentations scientifiques ou ce manuscrit.

Je tiens également à remercier chaleureusement les membres du jury, Lesly-Ann DANIEL, David MONNIAUX et Olivier SAVRY, ainsi que le président du jury, Guy GOGNIAT, d'avoir accepté d'évaluer ce travail et pour la qualité des échanges constructifs lors de la soutenance. Leurs remarques et suggestions ont grandement contribué à l'amélioration du contenu et de la clarté de cette thèse. Je souhaite enfin remercier tout particulièrement les rapporteurs, Guy GOGNIAT et Yuval YAROM, pour la pertinence et la profondeur de leurs retours, qui ont permis d'enrichir et d'approfondir le contenu de ce manuscrit.

Je souhaite ensuite remercier l'équipe TARAN pour son accueil bienveillant, ainsi que pour son aide et ses échanges, qui ont grandement facilité mon intégration dans l'environnement de recherche. Je les remercie également pour leur soutien précieux durant la préparation de la soutenance. Je tiens par ailleurs à exprimer ma gratitude à Olivier SENTIEYS pour avoir accepté mon intégration au sein de l'équipe et pour avoir contribué au bon déroulement de la soutenance. Enfin, je remercie l'ensemble du person-

nel administratif, Nadia DEROUAULT et Elodie COTTREL, pour leur efficacité et leur soutien, en particulier lors de la préparation des missions et de la soutenance.

Je souhaite adresser mes remerciements les plus sincères à ma plus grande source de motivation, celle qui m'a constamment poussé à me dépasser : MA FAMILLE. Toujours unie malgré la distance, toujours présente malgré les kilomètres, elle a rendu l'éloignement plus léger et les épreuves plus supportables.

Je pense tout particulièrement à mes parents, Andry et Vero, dont le soutien indéfectible et la présence constante ont fait que la distance qui nous séparait ne s'est jamais réellement fait sentir. Je pense également à mon frère, Haja, et à ma soeur, Hani, ainsi qu'à mes amis, Stéphanie, Nancy, Stelvio et Kanto, pour leur bienveillance, leur soutien précieux et leur aide sincère, que les mots peinent à remercier à leur juste valeur. Qu'il s'agisse de moments personnels ou professionnels, ils ont été, sont et resteront toujours présents à mes côtés, et c'est grâce à eux que j'ai pu surmonter chaque difficulté rencontrée au cours de ce parcours.

MERCI!

Résumé en français

L'automatisation des tâches a longtemps été un objectif persistant de l'humanité. Les humains ont toujours cherché des moyens de simplifier les tâches, d'accélérer les processus et de réaliser ce qui serait autrement impossible par leurs seuls moyens. Peu d'inventions incarnent cette ambition aussi clairement que les machines de calcul. Ce qui a commencé comme l'idée simple de combiner des composants de base pour effectuer des opérations arithmétiques est devenu le fondement de certains des outils les plus sophistiqués dont nous dépendons aujourd'hui.

L'un des aspects les plus frappants de l'évolution des technologies est son caractère auto-renforçant. L'augmentation des performances informatiques conduit à des améliorations des équipements de fabrication, qui produisent à leur tour des dispositifs plus rapides et plus puissants, créant ainsi un cycle d'accélération continue.

À mesure que la puissance de calcul augmente, elle devient un pilier toujours plus central de la vie moderne. Elle accélère la recherche scientifique, accroît la productivité et rend possibles des domaines entiers (par exemple, les télécommunications, l'analyse de données à grande échelle et l'intelligence artificielle). Son influence sur la société est profonde et durable : elle modifie notre manière de travailler, de communiquer et d'interpréter le monde.

L'évolution des dispositifs informatiques a convergé vers un principe commun: construire une machine capable d'exécuter uniquement un petit ensemble d'opérations simples sur des données, appelée **processeur**, et réduire tout programme complexe à une séquence de ces opérations élémentaires, appelées **instructions**. Ces instructions opèrent généralement sur des **données** qui servent d'entrées et de sorties. Un exemple simple est *1 + 1 = 2*, où + représente l'instruction et les nombres représentent les données manipulées.

De nos jours, presque tous les dispositifs électroniques utilisent un processeur pour fonctionner. Au-delà de son rôle technologique, la performance des processeurs est également devenue un moteur commercial majeur, voire un symbole de puissance au niveau international. Au fil du temps, la demande en performance et en quantité de processeurs augmente de manière exponentielle.

Un processeur est fondamentalement composé de *transistors*: des composants électron-

iques élémentaires qui agissent comme des interrupteurs. Le nombre de transistors dans un processeur donne une indication de sa performance potentielle. Ainsi, la taille des transistors joue un rôle essentiel dans le progrès des processeurs: plus elle est petite, plus il est possible d'intégrer de transistors dans un seul processeur. De plus, une taille de transistor réduite diminue la consommation d'énergie, produit moins de chaleur par transistor et réduit la distance entre les composants, ce qui diminue la latence. En conséquence, la taille des transistors est devenue un facteur clé de l'amélioration des performances des processeurs.

Cependant, la réduction de la taille des transistors se heurte à des limites physiques et technologiques: les transistors approchent des dimensions où les effets quantiques, les fuites de courant et les contraintes de fabrication posent des défis critiques.

Plutôt que de s'appuyer uniquement sur la réduction de la taille des transistors, la recherche s'est également concentrée sur l'optimisation de la manière dont les processeurs exécutent les instructions. Étant donné que les logiciels sont représentés sous forme de séquences d'instructions, une métrique clé de performance est le nombre d'instructions exécutées par unité de temps.

Pendant l'exécution, le processeur maintient un état interne contenant les données manipulées par les instructions (appelé **état architectural**). En outre, le processeur conserve un ensemble de valeurs fournissant des informations sur la manière dont les instructions ont été exécutées afin d'améliorer leur exécution future. Ces informations constituent l'**état microarchitectural** du processeur. Bien que certaines instructions puissent intentionnellement influencer ou contraindre cet état pour des raisons de performance ou de sécurité, l'état microarchitectural lui-même ne devrait pas être **directement observable** par les instructions logicielles.

Contexte et problématique

La technologie est désormais profondément intégrée dans notre vie quotidienne et elle traite inévitablement certaines de nos données les plus sensibles. Étant donné que le marché et la recherche ont constamment privilégié la performance, souvent au détriment de la sécurité, une question fondamentale se pose: **pouvons-nous être certains que les dispositifs ne trahiront pas notre confiance?**

Le facteur le plus limitant pour la performance est la dépendance entre les instructions: pour exécuter une instruction donnée, la précédente doit être terminée.

La majorité des gains de performance dans les processeurs modernes provient du mécanisme de **spéculation**. L'idée est de parier sur le résultat des instructions afin de commencer l'exécution des suivantes de manière anticipée. La performance est améliorée si la prédiction est correcte, car cela masque la latence. Dans le cas contraire, le processeur annule toutes les instructions exécutées de manière spéculative.

Lorsqu'une mauvaise spéculation se produit, le retour au dernier état correct entraîne des coûts en performance (attente de la restauration de tous les états), en consommation d'énergie (remise de chaque transistor dans son état initial) et en surface (stockage du dernier état correct de chaque transistor modifié). La complexité des processeurs modernes rend cette opération extrêmement coûteuse.

L'état microarchitectural n'est pas critique pour la correction de l'exécution, puisqu'il est principalement utilisé pour anticiper et accélérer l'exécution future. En conséquence, la grande majorité des concepteurs de processeurs choisissent de réduire ces coûts en ne restaurant que l'état architectural. Le fait de ne pas restaurer l'état microarchitectural modifié peut révéler des informations sur les instructions précédemment exécutées et les données manipulées, ce qui soulève des problèmes de sécurité. De plus, des travaux de recherche ont montré que des effets physiques peuvent divulguer l'état réel de la microarchitecture, permettant la fuite des données manipulées pendant l'exécution. Cette propriété a donné naissance à deux grandes classes de vulnérabilités bien connues, les **canaux cachés** et les **canaux auxiliaires** [9], [36], [49], [72], [85].

En influençant délibérément le processeur pour qu'il accède de manière spéculative à des données sensibles, des attaquants peuvent modifier l'état microarchitectural et divulguer des secrets via des canaux cachés. Cette vulnérabilité est connue sous le nom de **Spectre** [35].

Depuis la découverte de Spectre, de nombreuses contre-mesures ont été proposées. Ces solutions peuvent être classées en deux grandes catégories: les **solutions logicielles**, qui modifient les programmes afin de limiter les fuites lors de la mauvaise spéculation, et les **solutions matérielles**, qui impliquent des modifications de l'implémentation du processeur pour empêcher l'attaque.

Chaque approche présente des avantages et des inconvénients :

— Les **solutions logicielles** peuvent être appliquées directement au code existant, ce qui les rend relativement simples à déployer et très flexibles, mais elles entraînent généralement un surcoût de performance important et une protection incomplète.

— Les **solutions matérielles** offrent un contrôle plus fin des comportements spécu-

latifs, permettant de meilleures performances et une protection plus robuste. Cependant, elles sont fortement dépendantes des implémentations de processeurs, ce qui les rend difficiles à reproduire et à généraliser à différents processeurs.

La recherche s'est de plus en plus orientée vers des solutions matérielles. Cependant, ces solutions produisent fréquemment des résultats incohérents: les mesures de performance diffèrent souvent lorsque des études tentent de reproduire les travaux originaux, comme discuté dans section 5.2.3.

Ces incohérences peuvent être attribuées à plusieurs facteurs. Premièrement, les benchmarks choisis et les processeurs cibles ont une forte influence sur les résultats d'évaluation. Deuxièmement, les chercheurs n'ont accès qu'à des processeurs open-source simples, et de nombreux fournisseurs commerciaux ne divulguent pas leurs évaluations de sécurité. Troisièmement, des modèles de menace et des hypothèses initiales différents rendent la comparaison des solutions proposées difficile.

Ainsi, bien qu'un large éventail de stratégies de protection ait été proposé pour faire face à Spectre, les problèmes susmentionnés empêchent une évaluation et une comparaison fiables de leur efficacité.

Contributions

Évaluation de la spéculation sélective

Malgré les incohérences observées dans l'état de l'art, nous constatons que la plupart des contre-mesures existantes reposent sur une approche de **spéculation sélective**, qui consiste à autoriser l'exécution spéculative pour les instructions considérées comme *non dangereuses*, tout en bloquant celles susceptibles de contribuer à la divulgation de données sensibles.

Cependant, des contradictions apparaissent dans les résultats de performance et de sécurité des approches de l'état de l'art qui adoptent cette méthode.

Dans cette première contribution, nous évaluons l'efficacité de la spéculation sélective sur une famille de processeurs basés sur l'architecture `RISC-V` , sous un modèle de menace dans lequel le matériel n'a aucune connaissance des données secrètes. Dans ce cadre, nous faisons l'hypothèse conservatrice que **tous les contenus mémoire sont sensibles**.

Nous mettons en place un environnement de test qui simule plusieurs sémantiques de spéculation sélective et génère des statistiques permettant de comparer la performance et

la sécurité de chacune.

Cet environnement de test permet une comparaison équitable de multiples stratégies, qu'elles soient logicielles, matérielles ou hybrides, dans un cadre unifié.

Au-delà des résultats détaillés obtenus par cette évaluation, notre étude met également en évidence les limites de la spéculation sélective : **elle ne peut pas offrir une protection complète contre Spectre sans entraîner une pénalité de performance significative**.

Valeurs secrètes architecturales

En nous appuyant sur les conclusions tirées de notre première contribution, notre deuxième contribution propose une approche qui, à notre connaissance, n'a pas été introduite dans l'état de l'art existant.

Nos expériences précédentes ont montré que l'obtention de zéro instruction exploitable selon notre méthode de détection conduit à des niveaux de performance proches de ceux d'un processeur in-order lorsqu'une approche de spéculation sélective est utilisée. Les limites de la première approche peuvent provenir non seulement de la sémantique de la solution elle-même, mais également du modèle de menace.

En particulier, l'absence de connaissance matérielle sur les données réellement sensibles oblige le processeur à traiter toutes les données comme secrètes, entraînant des restrictions inutiles et une dégradation significative des performances.

Pour remédier à cette limitation, notre deuxième contribution introduit explicitement la notion de *données secrètes* : des valeurs du programme marquées comme sensibles, dont l'information de sensibilité est propagée à toutes les données dérivées.

Le matériel nécessite deux domaines de protection complémentaires pour garantir la confidentialité des données secrètes:

— Un niveau de protection mémoire afin d'assurer la confidentialité des données secrètes lorsqu'elles sont stockées en mémoire.

— Un niveau de protection des registres pour imposer des restrictions sur la manipulation des données secrètes et empêcher les opérations susceptibles de compromettre leur confidentialité.

Table of Contents

Chapter 1

Introduction

Automating tasks has long been a persistent goal of humanity. People have always looked for ways to make tasks easier, speed things up, and do what would otherwise be impossible on their own. Few inventions capture this ambition as clearly as computing machines. What began as the simple idea of combining basic components to perform arithmetic has become the basis for some of the most sophisticated tools we rely on today.

One of the most striking aspects of the progression of technologies is its self-reinforcing nature. Increased computer performant leads to improvements in manufacturing equipment, which in turn produces faster and more powerful devices, creating a cycle of continuous acceleration.

As computing power continues to grow, it becomes an ever more central pillar of modern life. It speeds up scientific research, increases productivity, and makes whole fields possible (e.g., global telecommunications, large-scale data analysis, and artificial intelligence). Its influence on society is deep and lasting: it changes how we work, communicate, and interpret the world.

The evolution of computing devices has converged toward a common principle: build a machine capable of executing only a small set of simple operations on data, a **processor**, and reduce any complex program into a sequence of these basic operations, called **instructions**. These instructions generally operate on **data** that serves as inputs and outputs. A simple example is *1 + 1 = 2*, where `+` represents the instruction and the numbers represent the data being manipulated.

Nowadays, nearly all electronic devices use a processor to function.

Beyond its technological role, processor performance has also become a major commercial driver and even a symbol of power at the international level. Over time, the demand for processor performance and quantity increases exponentially.

A processor is fundamentally composed of *transistors*: basic electronic components that act as switches. The number of transistors in a processor gives an indication of its

potential performance. Thus, the size of the transistor plays an important role in the progress of processors: the smaller it is, the more transistors can fit in a single processor. Moreover, smaller transistor size reduces power consumption, produces less heat per transistor, and reduces the distance between components, which lowers latency. As a result, transistor size has become a key factor in improving processor performance.

However, shrinking transistor size has faced some physical and technological limitations: transistors are approaching dimensions where quantum effects, power leakage, and manufacturing constraints pose critical challenges.

Rather than solely relying on transistor size, research has also focused on the optimisation of **how** processors execute instructions themselves. As software is represented as a sequence of instructions, a key performance metric is the number of instructions executed per unit of time.

During the execution, the processor holds an internal state, containing the data manipulated by the instructions (referred to as the **architectural state**). Additionally, the processor holds a collection of values providing information on how instructions were executed in order to improve their future execution. This information is referred to as the **microarchitectural state** of the processor. While certain instructions may intentionally influence or constrain this state for performance or security purposes, the microarchitectural state itself should not be **directly observable** by software instructions.

1.1 Context and problem

Technology is now deeply embedded in our daily lives and it inevitably processes some of our most sensitive data. Since both the market and research have consistently prioritized performance, often sacrificing security, a fundamental question arises: **Can we be sure that the devices will not betray our trust?**

The most limiting factor for performance is the dependencies between instructions: in order to execute a given instruction, the previous one has to be completed. Most of the performance improvement in modern processors comes from the mechanism of **speculation**. The idea is to bet on the outcome of instructions in order to start executing the next ones early.

Performance is improved if the prediction is correct because it hides latency. Otherwise, the processor discards all speculatively executed instructions.

When misspeculation occurs, returning to the last correct state induces costs in perfor-

mance (waiting for all states to be restored), power consumption (switching each transistor back to its original state), and area (storing the last correct state of every switched transistor). The complexity of modern processors makes this operation prohibitively expensive.

Microarchitectural state is non-critical for the correctness of the execution, since it is primarily used to anticipate and accelerate future execution. Consequently, the vast majority of processor designers decide to reduce these costs by restoring only the architectural state.

Not restoring the altered microarchitectural state may reveal information about previously executed instructions and manipulated data, leading to security concerns.

Moreover, research has shown that physical effects can disclose the actual state of the microarchitecture, which allows leakage of the data manipulated during execution. This property gave rise to two well-known classes of vulnerabilities, **covert** and **side channels** [9], [36], [49], [72], [85].

By deliberately influencing the processor to misspeculatively access sensitive data, attackers can influence the microarchitectural state, and disclose secrets through covert channels. This vulnerability is known as **Spectre** [35].

Since the discovery of Spectre, numerous mitigation techniques have been proposed. These mitigations can be classified into two broad categories: **software mitigations**, which modify programs to limit leakage on misspeculation, and **hardware mitigations**, which involve modifications of the processor implementation to prevent the attack.

Each approach has advantages and drawbacks:

— **Software mitigations** can be applied directly to existing code, which makes them relatively straightforward to deploy and very flexible, but they generally result in a significant performance overhead and incomplete protection.

— **Hardware mitigations** provide more fine-grained control over speculative behaviors, enabling more performance and stronger protection. However, they are mostly tied to the implementations of the processor, making them challenging to reproduce and difficult to generalize across different processors.

Research has increasingly shifted toward hardware-based solutions. However, these mitigations frequently produce inconsistent results: performance measurements frequently differ when studies attempt to reproduce the original mitigation work, as discussed in section 5.2.3.

Such inconsistency can be traced back to several factors. First, the chosen benchmarks and target processors have a strong influence on evaluation outcomes. Second, researchers

only have access to simple open-source processors, and many commercial vendors do not disclose security assessments. Third, different threat models and initial assumptions make it hard to compare proposed mitigations.

Thus, although a wide range of mitigation strategies have been proposed to address Spectre, the aforementioned issues prevent reliably assessing and comparing their effectiveness.

1.2 Contributions

1.2.1 Evaluation of selective speculation

Despite the inconsistency observed in the state of the art, we note that most existing mitigations rely on a **selective speculation** approach, which consists of allowing speculative execution for instructions that are considered *non-threatening* while blocking instructions that could contribute to the disclosure of sensitive data.

However, contradictions occur in the performance and security results of the state-of-the-art approaches that choose this method.

In this first contribution, we evaluate the effectiveness of selective speculation on a family of processors based on the `RISC-V` architecture, under a threat model in which the hardware has no knowledge of which data is secret. In this setting, we conservatively **assume that all memory contents are sensitive**.

We set up a test environment that simulates several selective speculation semantics and generates statistical figures to compare the performance and security of each. Security evaluation focuses on identifying instruction sequences that could theoretically be exploited by Spectre.

This test environment enables consistent comparison of multiple mitigation strategies, whether software-only, hardware-only, or hybrid under a unified environment.

Beyond the detailed results obtained from this evaluation, our study also underscores the limitations of selective speculation: **it cannot fully protect against Spectre without incurring a significant performance penalty**.

1.2.2 Architectural secret values

Building on the insights gained from our first contribution, our second contribution proposes an approach that, to the best of our knowledge, has not been introduced in the

existing state-of-the-art.

Our previous experiments showed that achieving zero exploitable instructions according to our detection method results in performance levels close to those of an in-order processor when using a selective speculation approach. The shortcomings of the first approach may stem not only from the semantics of the mitigation itself but also from the underlying threat model.

In particular, the lack of hardware-level knowledge about which data is actually sensitive forces the processor to treat all data as secret, leading to unnecessary restrictions and significant performance degradation.

To address this limitation, our second contribution introduces the explicit notion of *secret data*: values in the program that are marked as sensitive, with this sensitivity information propagated to all derived data. The hardware requires two complementary protection domains to ensure secret data confidentiality:

— A memory protection level to ensure the confidentiality of secret data when stored in memory.

— A register protection level to enforce restrictions on secret data manipulation to prevent operations that could compromise its confidentiality.

1.3 Structure of thesis

This manuscript is structured into six chapters.

Chapter 1, which follows this introduction, presents the different layers that a software program goes through during its execution. It begins with the compiler level, where source code is translated into machine code understood by the processor. It then describes the architectural level, an abstract view of the processor, and finally covers the microarchitectural level, which explains how instructions are executed within the processor components, and describes a general implementation of an Out-of-Order (OoO) processor.

Chapter 2 provides a detailed implementation of the NaxRISCV processor.

Chapter 3 explores microarchitectural vulnerabilities and highlights the *transient execution* vulnerability, which enables the leakage of sensitive data through unintended instruction execution, such as Spectre and Meltdown.

Chapter 4 provides a synthesis of the state-of-the-art mitigation techniques for these vulnerabilities. It emphasizes the decreasing interest in Meltdown-related mitigations and

the growing focus on Spectre.

The first contribution, which evaluates selective speculation as a mitigation strategy, is detailed in **Chapter 5**. This chapter presents the entire experimental process, from defining the threat model and semantics to implementing each semantic variant on NaxRISCV. It then discusses and analyzes the experimental results, while also highlighting the limitations of both the methodology and the selective speculation approach itself.

Finally, **Chapter 6** exposes the second contribution of this thesis, which focuses on a new mitigation approach designed to identify secret data in the program and propagate this information to enhance hardware protection. Beyond protecting processors against Spectre, this approach also enforces the confidentiality of secret data in memory, ensuring that only protected instructions can access sensitive values.

1.4 Typographical Conventions

To facilitate the reading of this thesis, different text styles and information boxes are used to help identify and categorize specific types of content.

When a box introduces a new concept or mechanism, any later occurrence of that term will be marked with the same style and icon to help the reader easily recall its definition.

This **box** introduces the definition of a new term that will be mentioned in later discussions. It is primarily used for technical terms that do not originate from this manuscript.

This paragraph uses the context defined in the box above.

This **box** is used to highlight either a specific design choice or a note that is unique to this manuscript or our proposed approach.

This paragraph refers to a note previously discussed in the box above.

This **box** serves as a reminder of information introduced earlier in the manuscript, often several pages before.

This paragraph references a term that was mentioned in the box above.

Chapter 2

Bridging Software and Hardware: From Source Code to Microarchitectural State

---

While software examples such as Firefox, Excel, or PDF readers are widely known, it is less common to consider how a computer interprets and executes software to ensure it behaves as intended.

Figure 2.1 outlines the different abstraction layers involved in executing software on hardware components. At the **high-level**, a software application is composed of various algorithms, each implementing distinct tasks. These algorithms are expressed through programming languages that provide a higher-level description, comprehensible to humans. However, this manuscript focuses more on the **low-level**: how physical electronic components are able to understand these abstract descriptions. To shed light on this process, we begin by breaking down the transition from the **high-level** to the **low-level** in section 2.1 (layers 3 to 4 in Figure 2.1). We then dive into how the system interprets and processes a program written in a machine language (layer 4), from section 2.2 to section 2.3.

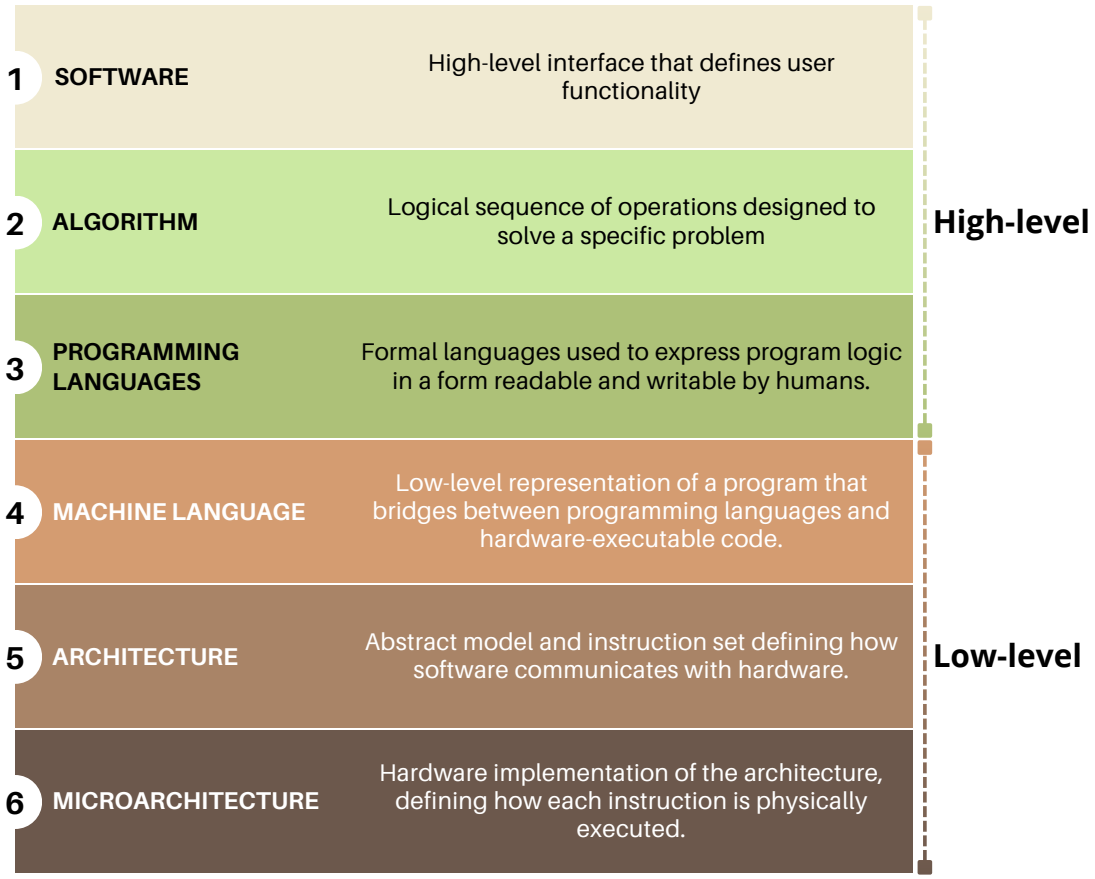


Figure 2.1 – Layered view of program execution, from software to hardware.

2.1 Translating Programming languages into Machine Instructions

A **compilers** translate programming languages (layer 3) into a machine language (layer 4) to be understood by a computer, as shown in Figure 2.2.

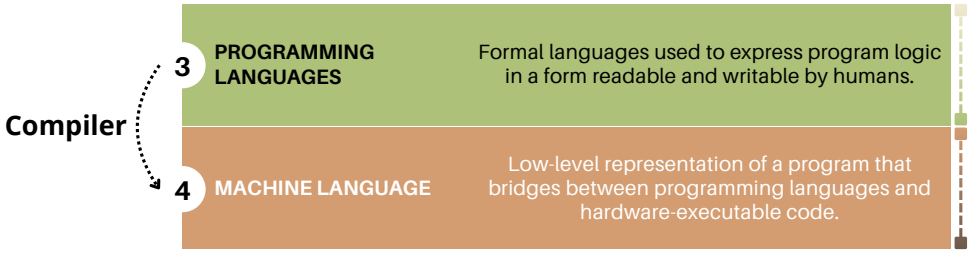


Figure 2.2 – Programming language to machine language.

A machine language is a different way of encoding and structuring instructions compared to programming languages. Instead of prioritizing readability and abstraction for human understanding, machine language encodes instructions in a form closely tied to the hardware, allowing the processor to understand them directly. The encoding and the set of authorized instructions depend on the target architecture (see section 2.2).

A **Static Single Assignment (SSA)** is a technique for representing programs in which each variable is assigned exactly once.
Intermediate Representation (IR) is a temporary representation of the program used by a compiler mainly for analyses and transformations conveniences. For instance, it can adopt the SSA property.
Machine Instruction Representation (MIR) is a medium-level representation of the program that derived from the IR, allow the compiler to perform optimizations and analyses independent of the target hardware architecture
A **`RISC-V` back end** is the part of the compiler responsible for translating the program's IR into `RISC-V` machine instructions, making the code executable on a `RISC-V` processor.

Since we target a `RISC-V` architecture in this thesis, we use *Clang/LLVM* to compile program, which is a compiler that provides a `RISC-V` back-end.

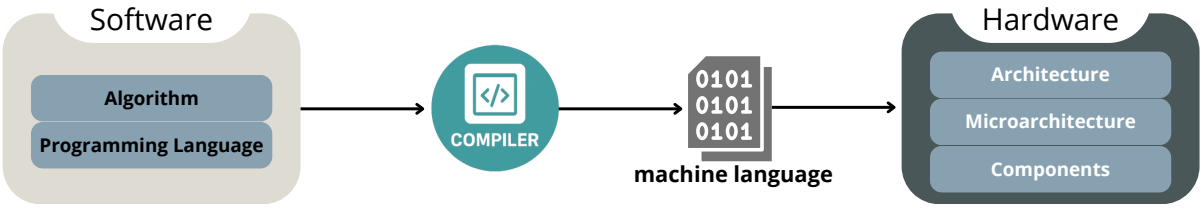


Figure 2.3 – Translating a programming language into the machine language of a target architecture.

2.1.1 Compiler description

The compiler translates programs from different programming languages to assembly code for different target architectures, as illustrated in Figure 2.3. LLVM compiler is composed of three parts:

— **Front end**: parses a program written in a specific language and translate it into LLVM-IR, which is the IR of LLVM.

— **Middle end**: this part is also known as *source and target independent*, the abstraction level of the IR allows the compiler to perform precises analysis and transformations.

— **Back end**: translates LLVM-IR into target specific assembly language. It also performs optimizations relative to the architecture and includes the **register allocation** (see subsection 2.1.2).

Compilers are designed to produce efficient code, but they do not always preserve the *entire semantic* or *behaviours* of programs it translates, provided that the final output remains unchanged. It applies several optimizations, specially in the middle end, that can inadvertently break protections or countermeasures introduced earlier during the compilation. Thus, we introduce our implementation countermeasures at the back end parts of the compilation.

2.1.2 Register allocation

A **register** is the smallest and fastest memory elements inside the hardware. Their size and number are dened by the processor's architecture.
infinite virtual registers are an abstraction used by compilers in IR, where the compiler assumes an unlimited number of registers to simplify analysis and optimization.

A **register allocator** maps the infinite virtual registers of an IR to the limited set of hardware registers.

The gap between the large number of variables typically used in applications and the limited set of hardware registers makes efficient register allocation crucial for performance. The compiler examines the control flow of the program to estimate when variables are “live", that is when their values are still needed, and uses this information to reuse hardware registers effectively.

Sometimes it may require more registers than are hardware register available, consequently, it temporarily stores registers values in memory and reloads them when needed. This operation is called *spilling*, and comes with a performance cost.

2.2 Architecture: Interface and Semantics

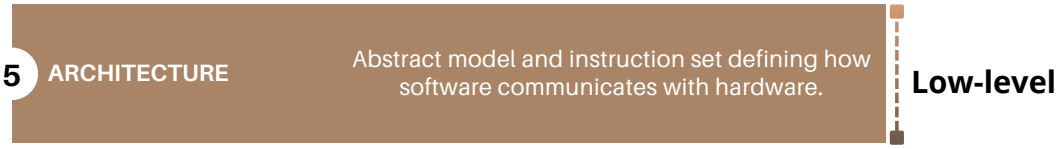


Figure 2.4 – Architecture level.

The **architecture** or Instruction-Set Architecture (ISA) (layer 5) is an abstract specification that defines which resources programs may use and modify and which parts of the hardware state are visible to them. Precisely, it specifies the rules that software must follow to control the hardware by specifying **what** the hardware does.

An **instruction set** is the set of machine-level instructions that a computer can understand and execute.

Architecture includes the instruction set, privilege levels, memory/register models, and exception handling.

As the ISA is an abstract contract between hardware and software, it does not vary across different hardware implementations of the same ISA in order to ensure program compatibility. Examples of architectures: `RISC-V` , x86, and ARM.

2.2.1 The `RISC-V` Architecture

The manuscript focuses on the `RISC-V` architecture, a modern open standard ISA designed to be simple, modular, and extensible.

Arithmetic operations include addition, subtraction, multiplication, and division. **Logical operations** include bitwise operations such as AND, OR, XOR, NOT, as well as shift operations.

Each Architecture follows distinct design philosophies and is often categorized by its data flow model. `RISC-V` adopts the **Load/Store architecture** model, one of the most popular architectures. This model uses dedicated instructions (**`load`** and **`store`**) to move data between memory and registers, while arithmetic, logical, and other operations only

operate on registers. More details on the exchange between memory and registers are given in section 2.3.1.

A **binary digit** (bit) is a data unit that can only have two possible values represented as zero or one.
A **byte** is made up of eight bits.
A **binary code** or **machine code** consists of instructions or data represented in bytes, that a processor can understand.
An **instruction format** or **instruction type** defines how the bits of a machine instruction are organized to represent its operation and operands.

The number of bytes used to encode one instruction depends on the targeted architecture. Each group of bytes that represents an instruction typically includes an **Opcode** specifying the operation to perform and, if required, source registers, a destination register, and an immediate value used as a constant operand.

For instance, `RISC-V` has fixed-length 32-bit instructions (in its base format) and supports multiple instruction formats (e.g., R-type, I-type, S-type).

A typical `RISC-V` instruction follows the format **`name_inst rd, rs1, rs2`**. Here, **`name_inst`** represent the operation to be performed, `rs1` and `rs2` are the source registers, and `rd` is the destination register that receives the result of the operation. The use of operands varies depending on the instruction type: some use immediate values, some ignore one or both source registers, and others have no destination.

A **R-type** instruction uses two source registers, has a destination register, and does not have an immediate value.

For example, the instruction **`add x1, x2, x3`** follows the R-type format. It computes `x2` + `x3`, and stores the result in `x1`.

The `RISC-V` instruction set is fully detailed in [55].

2.3 Microarchitecture: Design and Optimization

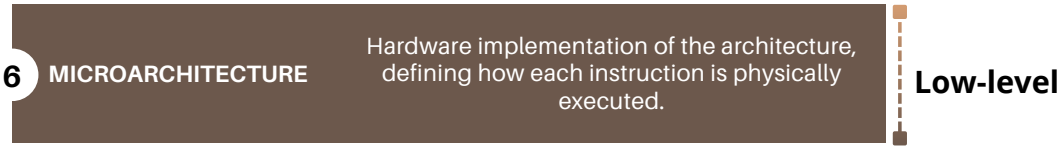


Figure 2.5 – Microarchitecture level.

The **microarchitecture** defines **how** the hardware carries out the architecture's features. It is the implementation of those architectural decisions, and the user does not necessarily need to know about this last layer to execute a program. Microarchitectures can vary widely among different processors, even within the same ISA family.

The terms *processing unit*, *CPU*, *core*, and *processor* are often used interchangeably and incorrectly in the context of computer systems, but they refer to distinct concepts. It is important to define them properly to avoid confusion:

A **processing unit** is a general term that is designed to **refer to any hardware that executes instructions**.

The term **processor** typically refers to any device capable of performing operations on data. In computing, it encompasses a wide range of processing units, which include the Central Processing Unit (CPU), Graphics Processing Unit (GPU), Neural Processing Unit (NPU), Tensor Processing Unit (TPU), etc. Although "processor" is a broader term, this manuscript uses it interchangeably with CPU as it is the only processing unit relevant to our discussion.

A **CPU** refers to the main processing hardware in a computer, the most versatile processing unit. It includes all components responsible for executing instructions (fetch, decode, and execute) for general-purpose software and handles a wide range of tasks. Originally, each CPU was a single chip or die. As semiconductor technology advanced, it became possible to place multiple processing units on a single CPU chip. Today, a CPU still refers to the entire chip, while each internal processing unit is known as a CPU core.

An Execution Unit (EU) is a hardware block that performs specific operations on instructions issued by the processor.

A **core** is one independent EU *within* the CPU chip. There can be one or more cores in each CPU.

Beginning in the 1960s, optimization of the microarchitecture became a systematic objective and a major focus for computer constructors. In this section, we present some common optimization mechanisms employed in modern microarchitectures and explain how each improves performance. These optimizations range from simple pipelined execution (see subsection 2.3.2) to more advanced mechanisms such as OoO (see subsection 2.3.4) and speculative execution (see subsection 2.3.5).

2.3.1 Unified Memory Model: Von Neumann Legacy

A basic calculator performs one operation at a time, manually provided by a user. As computational systems evolved and workload demand grew, this manual intervention became a limiting factor.

In 1945, John von Neumann introduced a more efficient architecture to enable a sequential execution model. Named after him, it is known as the Von Neumann architecture and has profoundly shaped modern computer design. It introduced an architecture with a unified memory model to store both instructions and data for the program to be executed, and an EU fetches and executes instructions in order from memory. Thus automating the execution of a list of instructions.

To grasp the developments that followed, two key concepts deserve a deeper look: the memory and the communication mechanism between memory and the EU.

Mechanics of Memory Access

Each piece of data stored in memory is associated with a memory address that indicates where it is stored. It helps the processor identify each piece of data and use it as needed during execution.

Two dedicated instructions allow the processor to communicate with the memory:

— **`Load`** instruction, with the following semantics **`load rd, rs1`**, allows the processor to read the data stored in the address `rs1` and store it on the `rd` register.

— **`Store`** instruction, with the following semantics **`store rs2, rs1`**, writes the value of `rs2` into the address stored on `rs1` on the memory.

The term **word** has a different meaning depending on the architecture. In `RISC-V` , it corresponds to 32 bits = 4 bytes.
A **signed value** represents both positive and negative values.
An **unsigned value** represents only non-negative values (zero and positive values), giving a larger positive range for the same number of bits.

In practice, there are multiple types of `load` and `store` instructions to handle different data sizes and sign interpretation. For load operation in `RISC-V` , the instructions `lb`, `lh`, and `lw` are used respectively for *byte*, *half-word* = 2 bytes, and *word-sized* data. The instructions `lbu` and `lhu` perform the same operations but interpret the loaded data as unsigned. It is similar for store operations with `sb`, `sh`, and `sw` instructions. This is illustrated in Figure 2.6.
However, we refer to the two operations as `load` and `store` instructions, as the specific size of the data is mostly not relevant to most explanations in this manuscript.

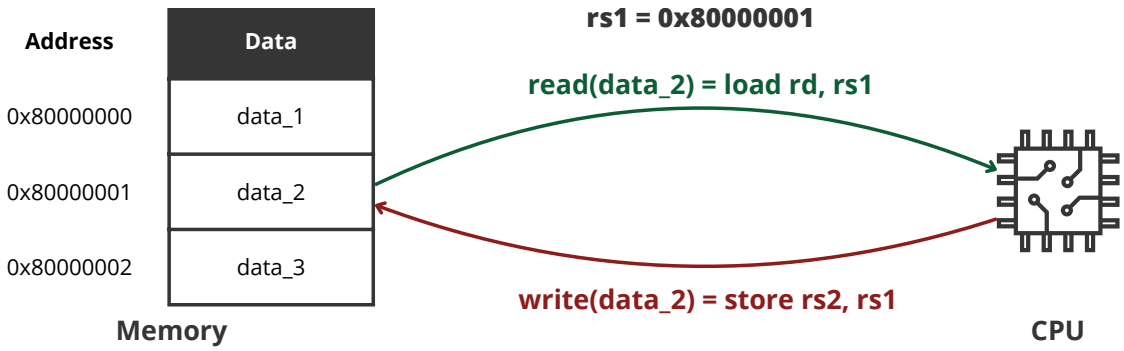


Figure 2.6 – Access memory with `load`/`store` instructions

Von Neumann Bottleneck

The main drawback of the Von Neumann architecture lies in the shared bus used to access both data and instructions, which limits the number of simultaneous operations possible. Moreover, as processors have become increasingly faster, memory access speeds have not kept pace, creating a critical performance bottleneck commonly known as the *Von Neumann bottleneck*. The delay in memory response makes the processor often idle, significantly limiting overall system efficiency. It was only a matter of time before a new memory model emerged to alleviate this performance gap.

The microarchitecture adopted a hierarchical memory system; this design incorporates small, fast, short-term memory units located closer to the processor, as illustrated in Figure 2.7. Known as **cache memory**, they store frequently or recently accessed data.

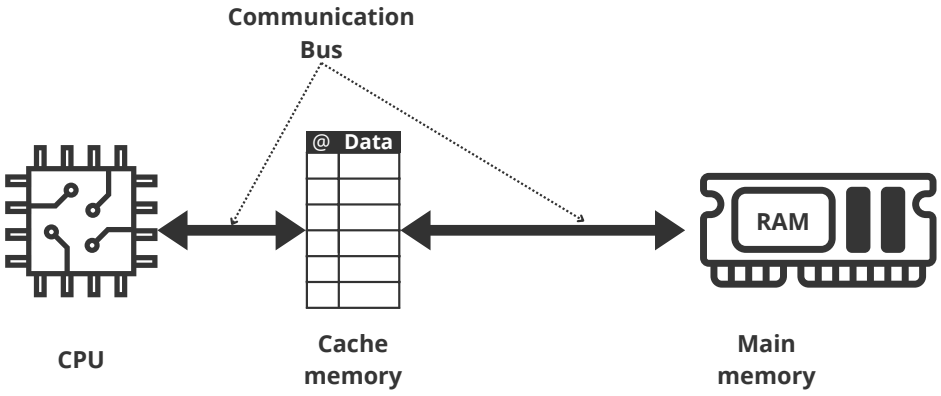


Figure 2.7 – Cache memory structure, reducing memory latency.

This dramatically reduces memory access latency and improves processor utilization.

When the cache memory is full, several replacement policies, such as least frequently used (LFU) or least recently used (LRU), are employed to select which data should be overwritten by the incoming data. **Before eviction**, the system assures that any modified data is **written back** to the higher levels of the memory hierarchy to maintain consistency.

With growing performance demands, multi-level cache systems were introduced to help balance speed and size across layers. These levels are commonly labeled with the letter L with the cache level as an index, such as L1, L2, and L3. The lower the level, the faster and smaller the memory, with each level providing a trade-off between size and speed. This multi-level cache system becomes a common implementation in modern processors.

The cache memory is organized in fixed-size blocks, called *lines*, typically 32, 64, or 128 bytes depending on the architecture. Hence, every memory access fetches an entire cache line, regardless of the type of `load` instruction (`lb`, `lh`, or `lw`). This ensures that memory blocks have a consistent granularity across the hierarchy, allowing the system to leverage spatial locality (the assumption that nearby addresses are likely to be accessed soon). Consequently, accessing a single element within a cache line brings all neighboring data into the cache. Eviction also occurs at the same granularity.

2.3.2 Through the Pipeline: From Fetch to Retire

Appearing for the first time in 1961 on the IBM 7030 processor, the **pipeline execution** technique was one of the first optimizations that the microarchitecture has applied. By dividing instruction processing into a sequence of smaller, independent stages (illustrated in Figure 2.8), the processor can overlap the execution of multiple instructions. This technique, known as pipelining, allows the processor to start executing a new instruction before the previous one has fully completed, thereby enabling multiple instructions to be in execution at the same time but in different stages (see Figure 2.9). This increases instruction throughput and reduces overall execution time. Moreover, it opens up various optimization possibilities, such as Out-of-Order (subsection 2.3.4) or speculative execution (subsection 2.3.5). A basic execution pipeline is composed of four stages; we associate each stage with a color in this manuscript to help understand further explanations:

— Fetch: Retrieve the next instruction to be executed.

— Decode: Gather all information about the instruction that the microarchitecture may need during the execution.

— Execute: Perform the operation corresponding to the instruction.

— Commit: Applies the result on the architecture side. It may consist of writing back the result value in the memory.

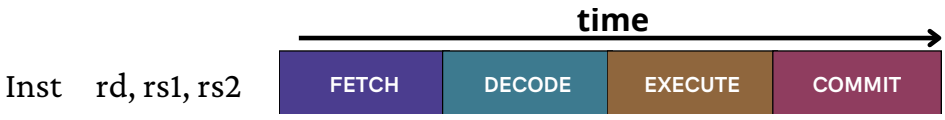


Figure 2.8 – One instruction pipelined

A white square (empty) represents a stall during pipeline execution; it can be interpreted as a wasted execution cycle that generally reduces performance.

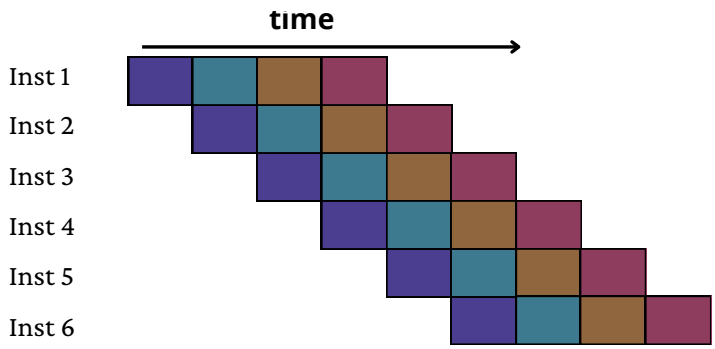


Figure 2.9 – Example of a pipeline execution.

2.3.3 Twice the Work, Half the Time: Superscalar Execution

Modern processor cores often have multiple independent EUs, which allow them to execute several instructions simultaneously. To minimize idle state, the core fetches more than one instruction at a time and attempts to execute them in parallel as long as there are no resource conflicts or interdependencies. This approach is known as **superscalar** execution.

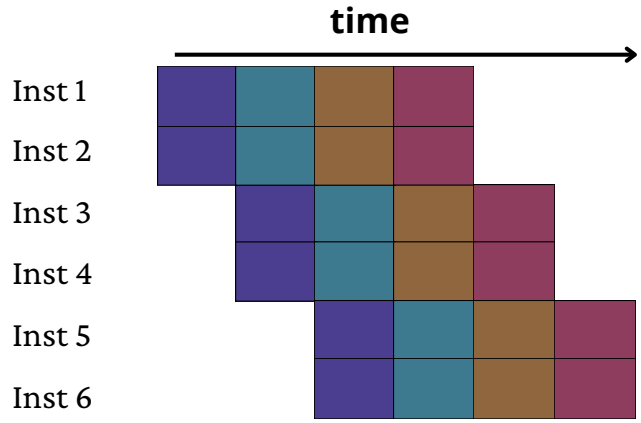


Figure 2.10 – *2*-wide superscalar execution

2.3.4 Disorder by Design: Out-of-Order Execution

Read-after-Write hazard (Read-After-Write (RAW)) occurs when a younger instruction tries to read from a register that an older instruction is still writing to. Since the younger instruction might read the value before the write is performed, it could get outdated or incorrect data. This creates a dependency between the two instructions, and the younger one must wait until the older instruction finishes writing to the register.

```
add     x2, t0, x1          # Write to x2 register
add     x4, x3, x2          # Read x2 register
```

Figure 2.11 – RAW example in `RISC-V` assembly.

In a basic pipelined execution, each instruction is executed strictly in the order defined by the program and cannot overtake others; this is known as in-order execution. Even in such a basic model, execution hazards can arise, typically the RAW hazard.

Hence, the processor stalls the pipeline for all subsequent instructions as long as the dependency is not resolved, as shown in Figure 2.12. This leads to a significant loss of performance.

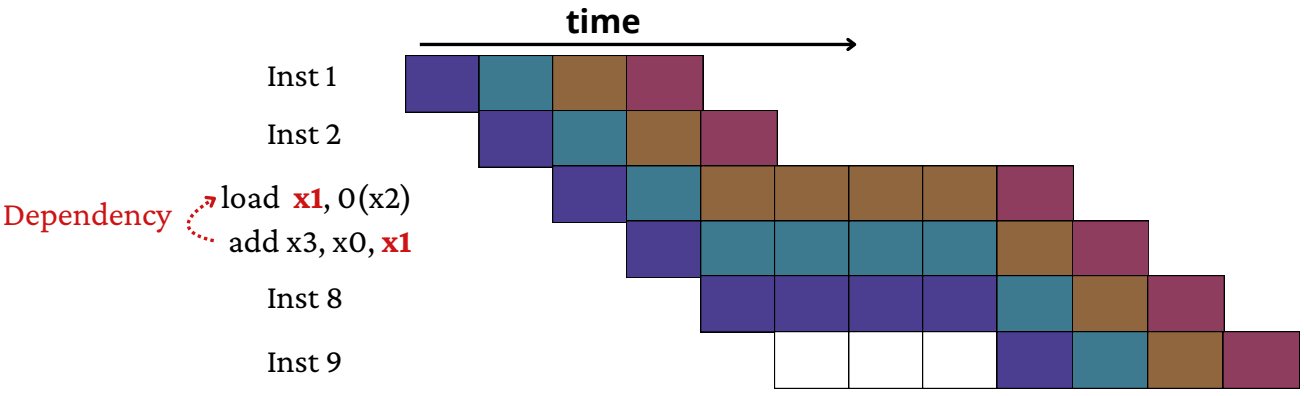


Figure 2.12 – In-Order execution, an early dependency stalls subsequent instructions.

To deal with this problem, a modern processor aims to remove program order at the microarchitecture level to reduce stalls (white squares) in the pipeline.

In this manuscript, a register or operand is considered **available** or **ready** when no pending instruction is waiting to write to it. This means that the register value has been updated and can be safely used as a source register.

Starting in 1964, the `scoreboarding` technique was developed on the CDC 6600 processor. This architecture had multiple functional units and could execute instructions as soon as their operands were available, ignoring the program order.

Write-after-Read hazard (Write-After-Read (WAR)) occurs when a younger instruction tries to write to a register that an older instruction still needs to read from. If the write happens too early, the older instruction might read the wrong (new) value instead of the one it expected.

```
add     x1, t0, x2          # Read x2 register
add     x2, x3, x4          # Write to x2 register
```

Figure 2.13 – WAR example in `RISC-V` assembly.

Write-after-Write hazard (Write-After-Write (WAW)) occurs when two instructions write to the same register, and the younger one writes before the older one. The older instruction could overwrite the result of the younger instruction, leading to incorrect final values.

```
add     x2, t0, x1          # Write to x2 register
add     x2, x3, x4          # Write to x2 register
```

Figure 2.14 – WAW example in `RISC-V` assembly.

However, ignoring the program order adds two new execution hazards: WAR and WAW. Those two dependencies are known as `false dependencies`, as they do not exist in logical program order; they appear only when the processor allows an instruction to execute before its predecessor.

Unfortunately, the scoreboarding method relies on a primitive mechanism for hazard detection and cannot handle false dependencies, typically the WAW hazard, which causes a stall in execution.

A few years later, in 1966, the `IBM System/360 Model 91` introduced the first out-of-order processor. Based on `Tomasulo's algorithm` [70], it reduced stalls and had greater

parallelism compared to its predecessors.

Section 2.4 gives a full explanation of the implementation of `Tomasulo's algorithm`, highlighting one of the innovations brought by the algorithm, known as register renaming (described in section 2.4.2), which handles the two false dependency hazards in an OoO execution.

As the name suggests, **Out-of-Order execution** allows instructions to execute independently of their program order. The mechanism fetches instructions in order and detects register dependencies between fetched instructions. It then pushes instructions to the execution stage only when: all register dependencies are resolved and the corresponding EU is not busy. Instructions are executed Out-of-Order, yet the microarchitecture ensures that they are **committed in the original order** of the program. This ordering is essential to handle events such as exceptions or misspeculation during execution, as represented in Figure 2.15.

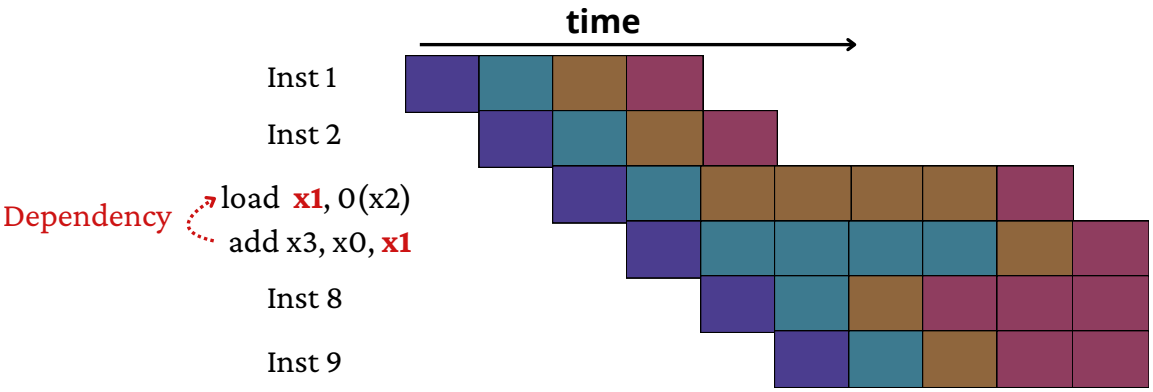


Figure 2.15 – With out-of-order execution, only the instructions with unresolved dependencies are stalled.

In all pipeline example illustrations, we assumed that the core is able to commit an unlimited number of instructions at once. However, this depends on the specific characteristics of the core.

2.3.5 The Art of Speculation: Predicting Execution Paths

The **Program Counter (PC)** is a dedicated register to store the address of the next instruction to be executed.

Some instructions, such as conditional branches and direct jumps, have the ability to alter the normal sequential flow of execution. When such an instruction is encountered, the next value of PC may point to a different address rather than simply progressing to the next sequential instruction. However, according to the principles of pipelining (see the subsection 2.3.2), an instruction is typically fetched several cycles before the previous instruction has been fully executed. This poses a challenge for the microarchitecture: either stall the pipeline and wait until the path execution is determined (see Figure 2.16), leading to a considerable loss of performance. Or guess the outcome based on historical behavior and take the risk of executing instructions that may ultimately prove invalid.

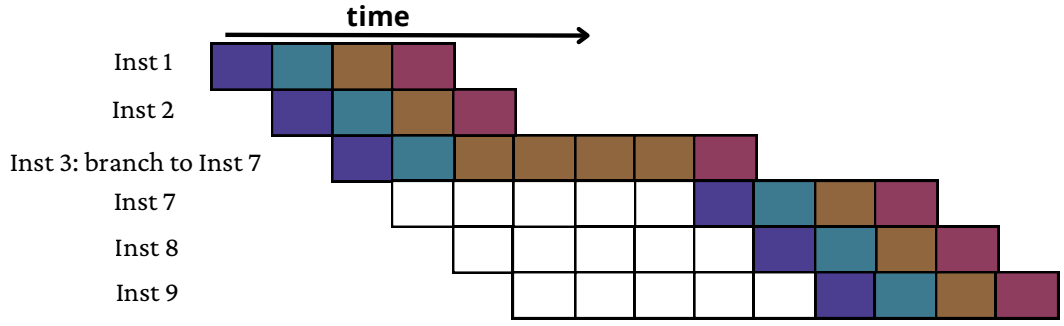


Figure 2.16 – Basic instruction pipeline without speculation.

Introduced in the mid-1990s to boost the performance, speculation techniques have since progressed to a variety of strategies that occur at different stages of the pipeline and apply on a different scenario.

We classify speculation techniques into **three categories**:

Control-flow speculation: This mechanism is dedicated to control-flow instructions, which are capable of modifying the normal execution flow of the program: conditional branches and unconditional jumps.

For an *unconditional jump*, the processor always updates the PC with the target address. The only uncertainty lies in the fact that the target address is not yet available in the early pipeline stages. To avoid stalling, the microarchitecture employs **destination**

address speculation, using a table that stores target addresses from past control-flow instructions executed, indexed by the control-flow instruction's address.

In contrast, a *conditional branch* has two possible outcomes: either `taken`, where the condition is met and the PC is updated with the target address, or `not taken`, the condition is not met and the PC is incremented to the next sequential instruction. The processor relies on **branch direction speculation** to anticipate this decision, predicting whether the branch will be taken or not in order to speculatively fetch along the most likely path. Then, destination address speculation is used if the branch is taken and the target address is not available yet.

The accuracy of both destination address speculation and branch direction speculation depends on the executed program and the design of the prediction mechanism. According to the literature [65], [86], prediction accuracy typically exceeds 80%.

An exception arises with the **return instruction**, a special form of unconditional jump that transfers control back to the instruction following the original function call. Since this target address is always known, a dedicated table is used to store return addresses. In practice, the accuracy of return prediction is limited only by the depth of the table and often approaches nearly 100%.

Dependency speculation: Another common situation where modern microarchitecture frequently employs speculation is during the execution of a `load` instruction. Instead of predicting the execution path, this form of speculation guesses the dependencies between instructions.

Since memory accesses can be relatively slow, and the instructions that follow often depend on the loaded data, waiting for the execution of a `load` instruction could significantly stall the pipeline. To mitigate this, processors tend to prioritize the execution of `load` instructions. Still, a `load` instruction may have an address dependency with an older `store` instruction if both have the same target address. In this case, executing the `load` before the `store` may result in loading stale data. A simple register dependency analysis cannot detect the dependency as the two instructions may use different source registers that store the same memory address. In this case, a `load` instruction is forced to stall until all preceding `store` instructions have completed the computation of their target addresses.

To improve the performance, the microarchitecture speculates on whether a `load` instruction has an address dependency with an older `store`. If the assumptions hold, the

`load` (and its dependent instructions) proceed early, reducing the overall latency.

Value speculation: This remains the least adopted in commercial processors. The value speculation technique aims to reduce the stall caused by dependencies on a high-latency operation by speculating on the result value of those operations. Despite the potential benefits, it is still in the experimental stage.

Although the terminology may preempt concepts discussed later, in this work we refer to instructions that trigger a speculative execution as **speculation gadgets**.

There are **two possible results** when a speculation gadget is executed:

When speculation succeeds

Good prediction helps the processor avoid idle cycles and optimize the use of the EUs. It also hides the latency of slower operations like memory accesses on `load` instructions.

In Figure 2.17, the processor predicts the branch to be taken and speculatively jumps to instruction seven while waiting for the completion of the speculation gadget (third instruction) to validate its assumption.

Since commit operations are always carried out in-order, they are forced to wait for the speculation gadget to resolve.

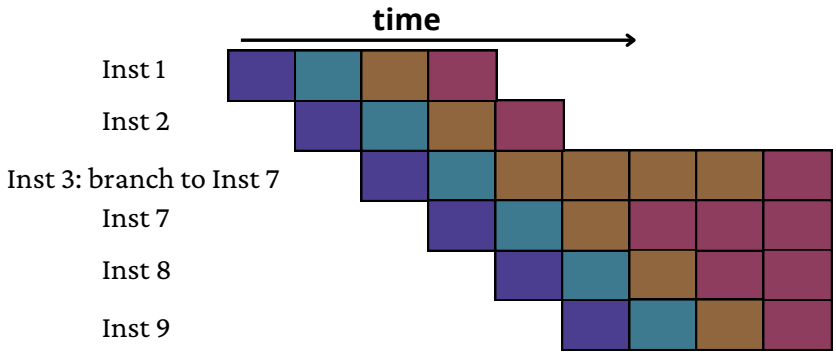


Figure 2.17 – Good prediction on branch instruction.

When speculation fails

As it is a speculation, the assumption made by the processor can be incorrect. In such cases, the processor executes instructions that should not have been executed, as in the

fourth, fifth and sixth instructions in Figure 2.18. However, as soon as the misspeculation is detected, by comparing the guessed outcome with the actual result of the speculation gadget, the microarchitecture initiates a process of *rescheduling*. It consists of flushing the pipeline, fetching the correct instruction, and discarding the commit requests of all misspeculatively executed instructions.

Rescheduling signals are issued only by the speculation gadget during its commit stage, ensuring that the instruction is indeed meant to be executed. More details about rescheduling are in section 2.4.4.

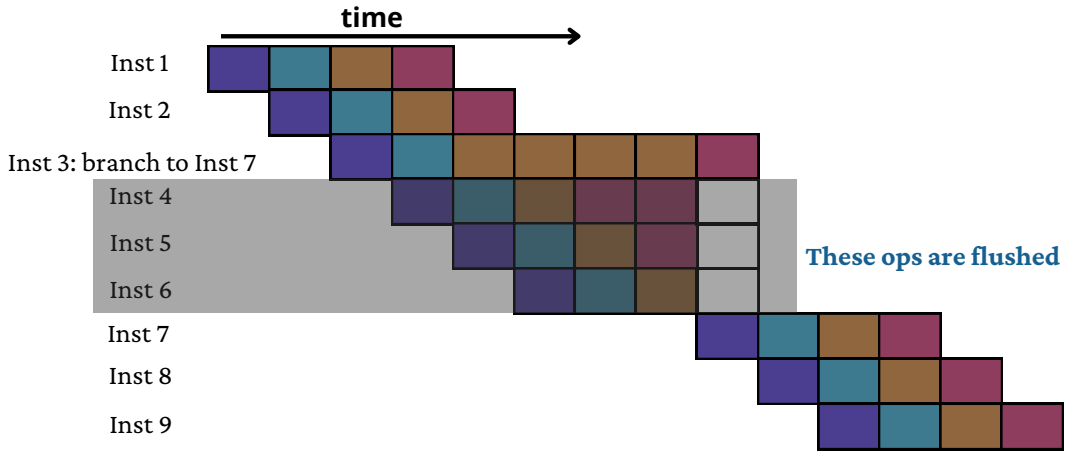


Figure 2.18 – Bad prediction on branch instruction.

In Figure 2.18, the grey area represents a **misspeculated window**, which is later flushed during a rescheduling operation. Despite the microarchitecture's efforts to hide the effect of misspeculation, some residual states may remain and be exploited to leak sensitive data. This vulnerability is explored further in chapter 5.

The remarkable aspect of speculation is that, in theory, performance remains equivalent to that of a processor without speculation (no-speculation performance is represented in Figure 2.16) when a bad speculation occurs, because a non-speculative processor would stall during the speculation phase. However, this is not always true in practice. The microarchitecture often needs additional cycles to propagate the rescheduling signal across different components, many of which require resetting internal state before the next execution. Still, the performance penalty from a misprediction is usually much smaller than

the performance gain achieved by a good speculation.

2.4 Disorder in Practice: Implementing Out-of-Order Execution

We learned what Out-of-order execution is in subsection 2.3.4, but how does a processor ensure the correctness of the program despite the lack of order? This section is dedicated to the implementation of a standard OoO core based on `Tomasulo's algorithm`.

For the sake of explanation, we use the different pipeline stages, described in subsection 2.3.2, and detail the operations that each instruction undergoes in each stage.

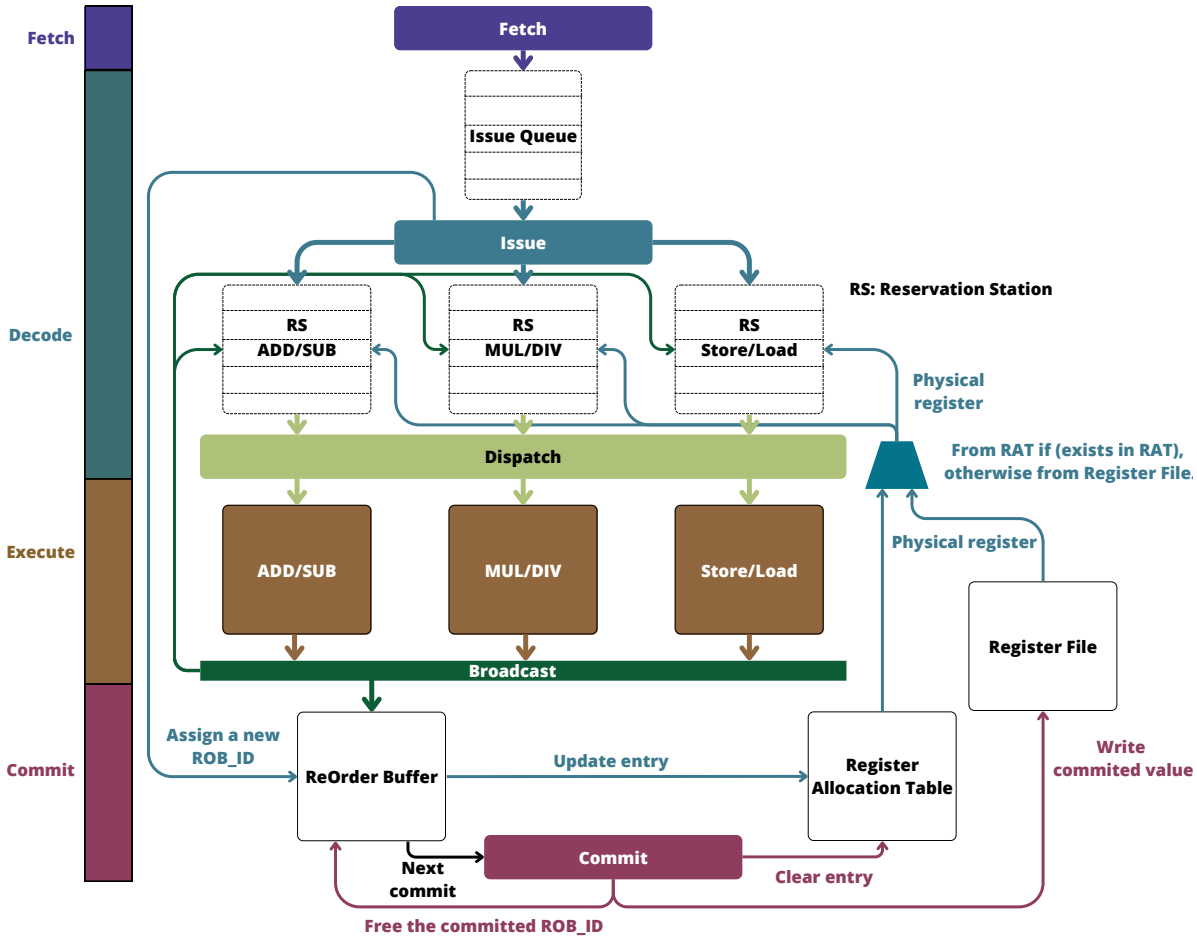


Figure 2.19 – Out-of-Order architecture.

2.4.1 Fetch Stage

Figure 2.20 – Fetch stage in Out-of-Order architecture.

The microarchitecture starts by retrieving the four bytes representing the instruction at the address pointed to by the PC register. Then the PC is either incremented to point to the next instruction or updated with a new address in the case of a jump.

In a modern processor, most speculation mechanisms begin predicting in this phase when needed. It allows the processor to modify the value of the next PC in case speculation is needed.

In a superscalar processor, it fetches more than one instruction at a time and the PC register is incremented according to the number of fetched instructions.

2.4.2 Decode Stage

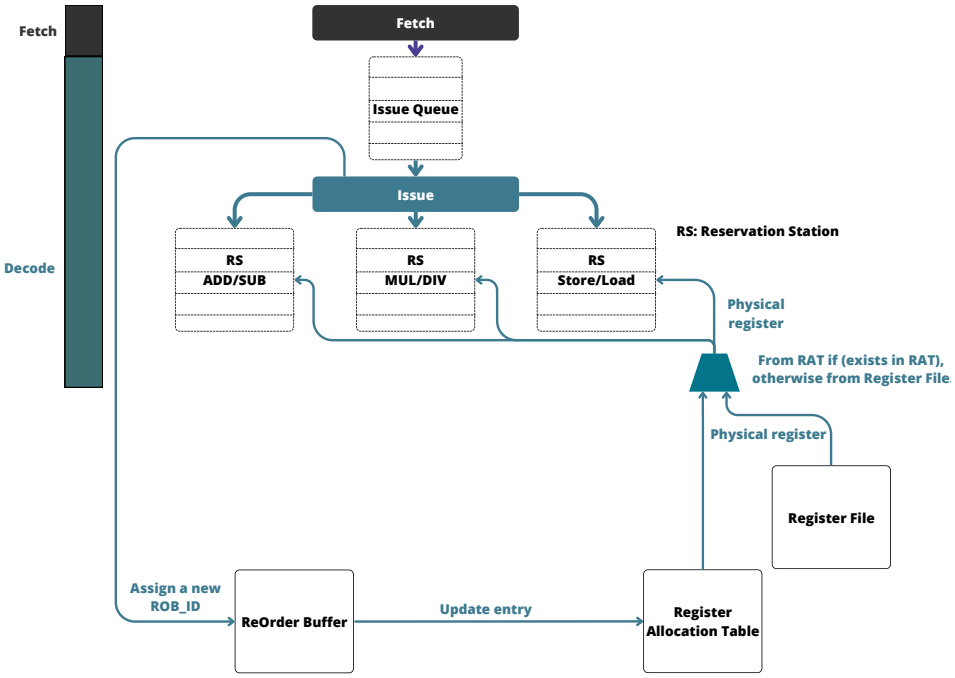


Figure 2.21 – Decode stage in Out-of-Order architecture.

A **reservation station** is a small memory table that stores temporary instructions until all conditions allowing them to execute are met.

At this stage, the core decodes the fetched instruction, in byte format, to gather relevant information. This is possible because the compiler encodes instructions according to the target architecture. All decoded instructions are stored in the Issue Queue (a First In First Out (FIFO) structure). As the fetch stage is in-order, instructions in the Issue Queue are still in program order. The oldest instruction is dispatched from the queue if its reservation station is not full.

Instructions are grouped according to the EU that can execute them, and each group has its own reservation station.

From there, the stage can be split into two steps:

Register renaming: This first step aims to remove false dependencies using register renaming. False dependencies happen when a younger instruction `I2` writes to a register that is also used by an older instruction `I1`, as explained in subsection 2.3.4. Even though there is no data dependency between the two, the processor might delay `I2` to avoid writing to the register before `I1` executes. This unnecessary delay penalizes performance, especially if `I2` is ready to execute and its EU is not busy.

Architectural registers: A register name used by a program. The number and purpose of architectural registers are fixed by the (ISA). Example of architectural registers: x0, x1, . . . , x31.
Physical registers: The actual hardware storage locations used to hold data. Generally, there are more physical registers than architectural ones to allow register renaming and support Out-of-Order execution. These registers are not visible to the program and are managed internally by the microarchitecture.

From the program's point of view, architectural registers represent the logical locations where data appears to be stored. In reality, a mechanism named **register renaming** dynamically maps each architectural register to a **unique physical register**. The apparent changes in the values of architectural registers are, in fact, an illusion. The system simply points to different physical registers that store the updated values.

In this manuscript, the term **register** refers to an architectural register unless specified otherwise.

In Figure 2.21, two structures hold physical register values. These two structures may refer to the same physical register but at different pipeline stages:

— **The Reorder Buffer (ROB):** Each entry corresponds to one instruction in the pipeline and tracks all its associated metadata. If the instruction writes to a destination register, the ROB temporarily stores the execution result along with the associated architectural register until the instruction commits. The value stored at this stage remains uncertain while disorder and speculation reign. Each entry is accessed by a unique identifier called `ROB_ID`.

— The **Register file:** Stores the value of physical registers after the commit stage. It reflects the architectural state, i.e., the value visible to the program after the last committed instruction. The number of physical registers in this structure is equal to the number of architectural registers, and each entry is indexed by the architectural register name. More detail is given in subsection 2.4.4.

A table called Register Allocation Table (RAT) points to the structure that holds the most recent value of a given architectural register. This mapping either points to a `ROB_ID` that is associated with the architectural register on the ROB or is empty, meaning that the committed value in the register file is already up to date.

When an instruction moves from the Issue Queue to the reservation station, the microarchitecture:

— Allocate a new entry in the ROB using the pointer `ptr_alloc`, which keeps track of the next available entry. Then the pointer increments to the subsequent entry in preparation for future allocations.

— If the instruction had a destination register, the RAT is updated to map this architectural register to the new `ROB_ID`.

— The source registers are looked up in the RAT. If a mapping exists for a source register, it is replaced by the corresponding `ROB_ID`.

Because the renaming operation is performed while instructions are still in program order, the table RAT always reflects the latest physical register for each architectural register, and instructions in the ROB are in program order.

By using register renaming for the WAW hazard, the younger instruction `I2` can overtake the older instruction without compromising the final state of the program. As shown in Figure 2.22, the two instructions now write to different physical registers, and the RAT table holds the latest mapping of the register `x2`, indicating `ROB_2`. As long as

`I2` is not committed and no younger instruction writes to the register `x2`, all subsequent instructions that consume the register `x2` will access the value stored in `ROB_2`.

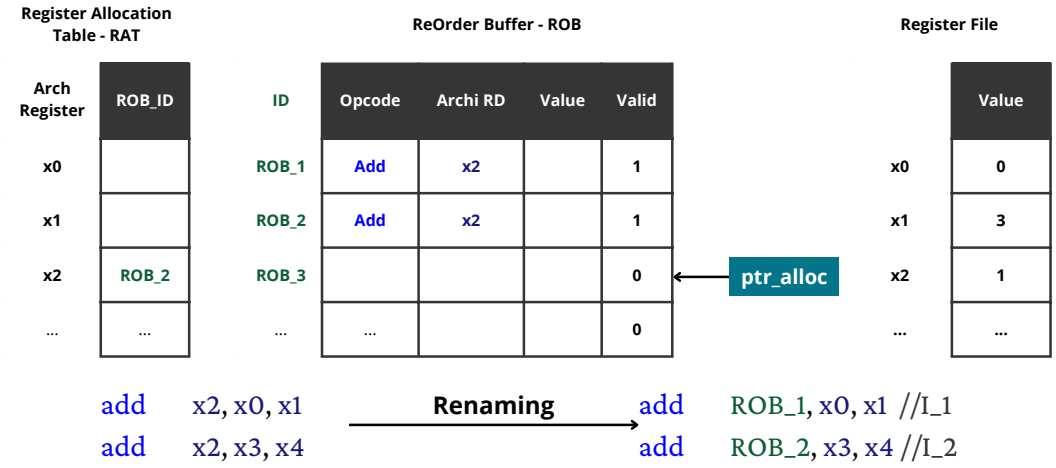


Figure 2.22 – Initialize RAT, ROB and register file during the renaming operation.

Dispatch: After the renaming operation, information about the instruction is pushed to the corresponding reservation station. It typically contains the **Opcode** and some meta-data, including the physical registers used by the instruction. The order of the instructions is meaningless at this stage; only data dependency matters. To track these dependencies, the reservation station verifies whether the source registers have been replaced by a `ROB_ID` during the renaming. If so, it waits for the availability of the corresponding values. If a source register was not replaced by a `ROB_ID`, it indicates the value is already committed and available in the register file. An instruction proceeds to the execution stage only when all its operands are available and its EU is ready.

OP	RD	RS1	RS2
Add	ROB 1	x0	x1
Add	ROB 2	x3	x4

Table 2.1 – Example of a reservation station for ADD/SUB instructions.

2.4.3 Execute Stage

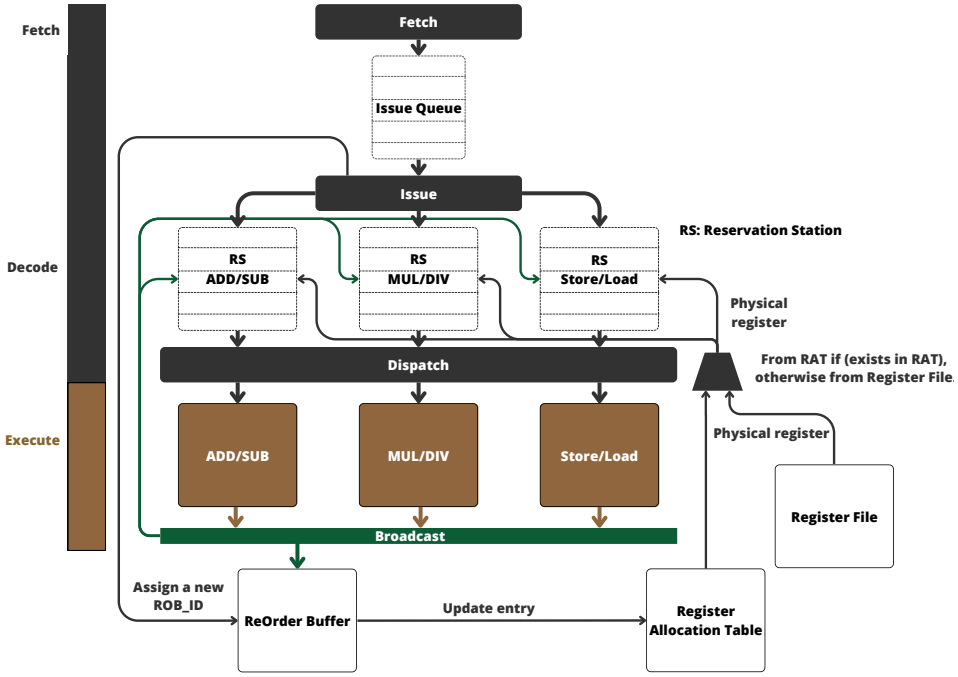


Figure 2.23 – Execute stage in Out-of-Order architecture.

Each EU can process one instruction at a time. The duration of the execution stage depends strongly on the type of instruction being processed. The time spent on simple operations like addition and subtraction is likely fixed and short. On the other hand, more complex ones, like multiplication, division, store, or load operations, may vary considerably depending on the operand values and the current state of the microarchitecture.

After an instruction completes execution, the results are broadcast to all units that may be waiting for it. In particular, the reservation stations can then clear the dependencies for subsequent instructions.

Simultaneously, the physical register in the ROB is updated with the value, and the corresponding entry is marked as executed, allowing future instructions to directly use the value if needed.

2.4.4 Commit Stage

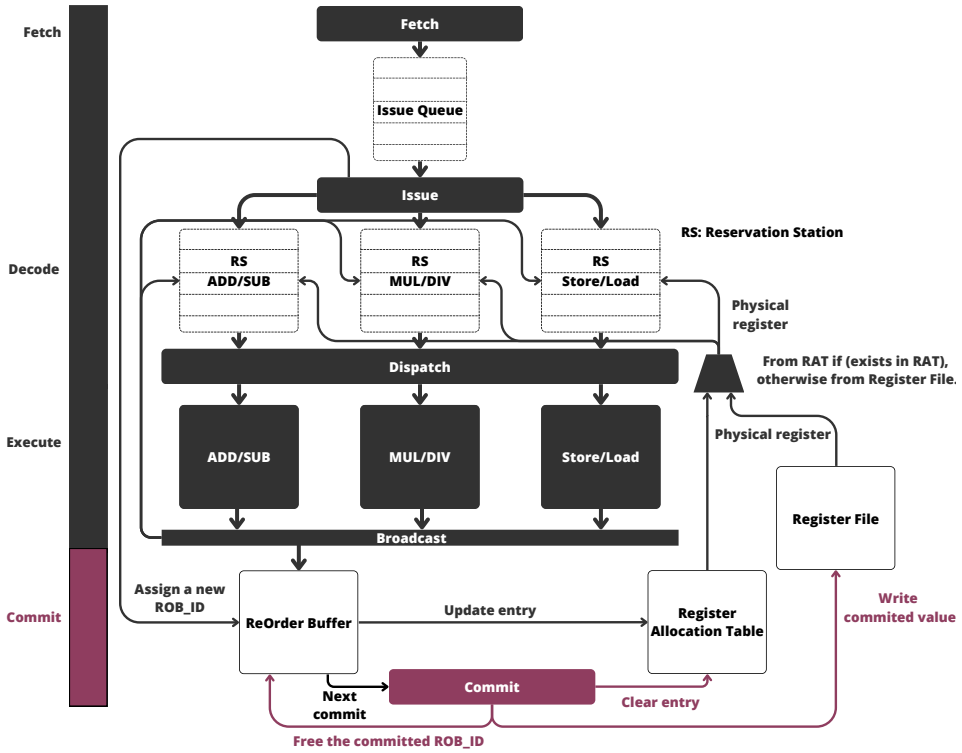


Figure 2.24 – Commit stage in Out-of-Order architecture.

This final stage of the pipeline acts as a bridge between the microarchitecture and the architectural view. Since uncertainty rules at the microarchitecture level, the commit stage re-imposes order and ensures that only valid instructions are committed.

Moreover, some instructions may be executed speculatively even if an exception is raised by an older one; hence, the microarchitecture must process commits in order. It also needs to be capable of rolling back the state of physical registers to that of the last committed instruction when necessary.

A pointer `ptr_commit` is used on the ROB, which holds instructions in program order, to track the next instruction to be committed. The microarchitecture commits an instruction only when it has completed execution and is pointed to by `ptr_commit`.

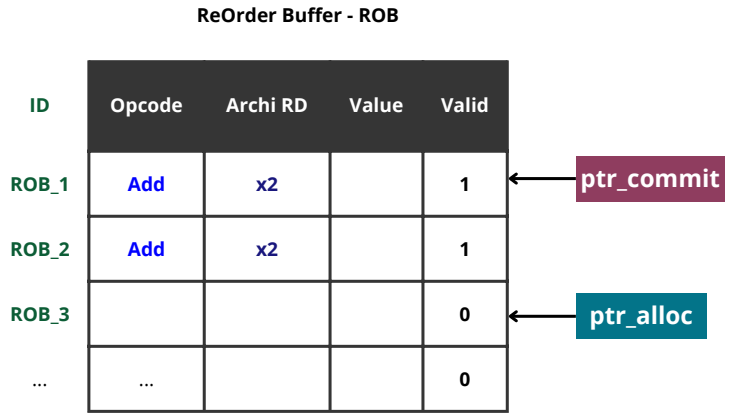


Figure 2.25 – Update ROB during the commit stage.

When an instruction is committed:

— If the instruction had a destination register, the result stored temporarily in the ROB is written to the register file using the architectural register as the address.

— The corresponding entry in the RAT is cleared to indicate that the value in the register file is the most recent.

— The instruction's `ROB_ID` is released and available for future instructions.

— The `ptr_commit` pointer is incremented to point to the next instruction.

Rescheduling: This consists of rolling back the state of the physical registers to that of the last committed operation. An instruction triggers a rescheduling operation during its last stage (commit stage), ensuring that it was not executed speculatively and that no prior instructions require rescheduling.

A rescheduling is performed in the following cases:

— a misspeculation: all speculative instructions are discarded.

— an exception: the processor needs to discard all instructions executed out-of-order after the faulting instruction.

Since the register file is only updated during commit and the ROB contains speculative values, recovering the last valid state involves:

— Clearing the RAT to force the microarchitecture to only use the register file.

— Setting `ptr_commit` to the value of `ptr_alloc` to ignore all pending commits on the ROB.

Chapter 3

Inside the NaxRISCV Core

This chapter delves into the details of all the necessary background on the implementation of the target core `NaxRISCV`. It is intended to be used to help reproduce the experience of our research.

The target core, **NaxRISCV**, is an **out-of-order**, **superscalar** processor and supports different forms of **speculation**. The NaxRISCV used in our experiments is a 64-bit processor, meaning that memory accesses are naturally aligned to 64 bits and registers store 64-bit values. Moreover, it has a cache line size of 64 bytes.

Memory cache is updated and evicted at the cache line size granularity.

In this manuscript, a **plugin** refers to a modular hardware component that encapsulates specific functionality (e.g., instruction decoding, EUs, ROB) and integrates into the processor core via a shared interface. It allows decentralized design, enabling flexible extension, customization, and separation of concerns in the hardware architecture.

NaxRISCV is developed with the **Scala** language using the **SpinalHDL library**, and it adopts a modular design where each component is declared as a **plugin** to enable straightforward integration of new features.

The project is fully open-source:

NaxRISCV project: https://github.com/SpinalHDL/NaxRiscv
Date: Nov 10, 2022.
Commit: `110338e7b62156326454baf53509c15fa98a4978`

Documentation: *https://spinalhdl.github.io/NaxRiscv-Rtd/main/index.html*.

A key advantage of NaxRISCV is its high configurability: users can, in theory, modify the number of instructions fetched, decoded, and committed per cycle, though doing so is often more complex in practice.

All results of our work are based on a version of NaxRISCV with: **two fetches, two decodes, and two commits per cycle.**

This manuscript describes only components that are either central to the system's behavior or were modified during the experiments. Figure 3.1 represents a simplified version of the NaxRISCV core.

To simplify the explanation, all **control-flow speculation** mechanisms are represented by the `prediction plugin` in Figure 3.1.

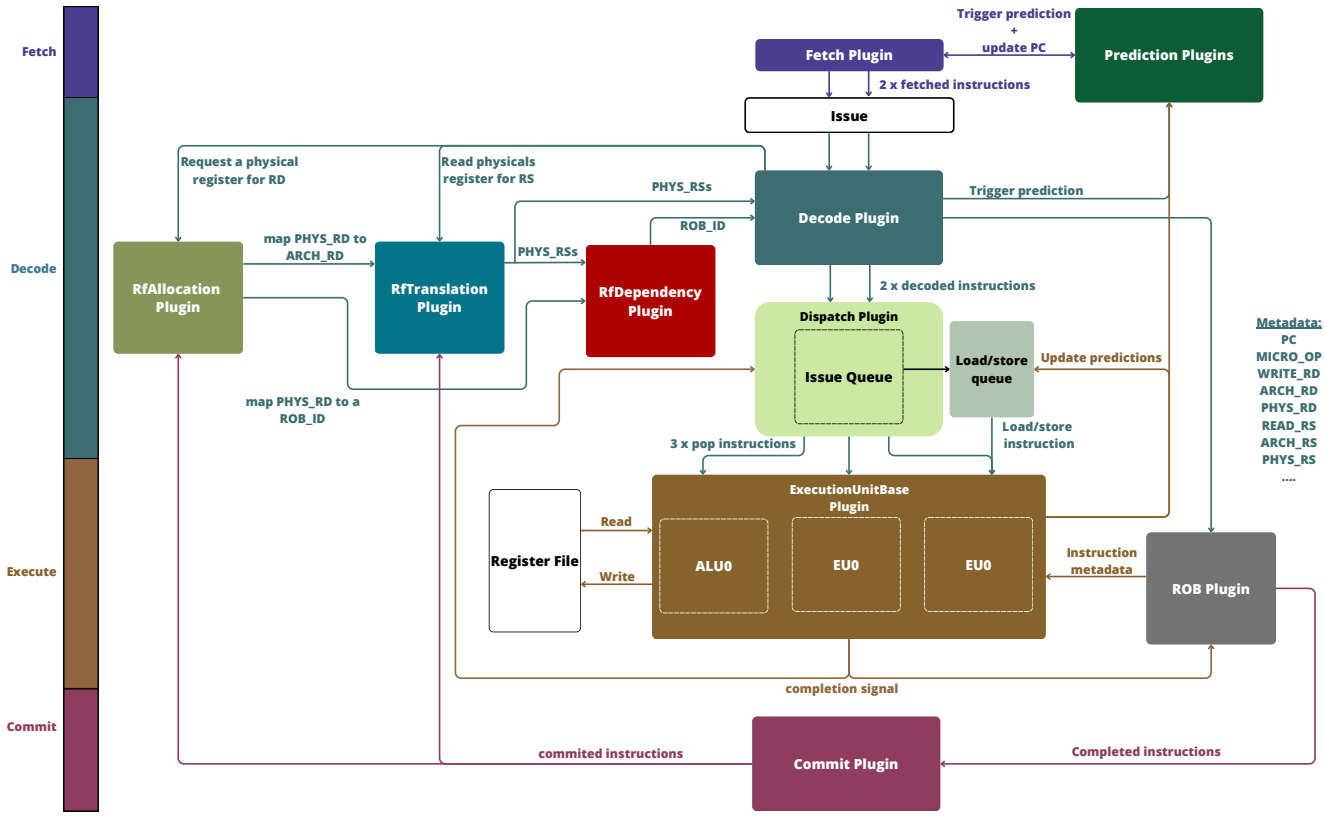


Figure 3.1 – Simplified illustration of the NaxRISCV's microarchitecture

As in the implementation of out-of-order in section 2.4, operations are described stage by stage and the plugins involved are detailed as we go along. The maximum number of fetch, decode, and commit instructions is fixed at `INST_COUNT` = **2**.

3.1 Fetch stage

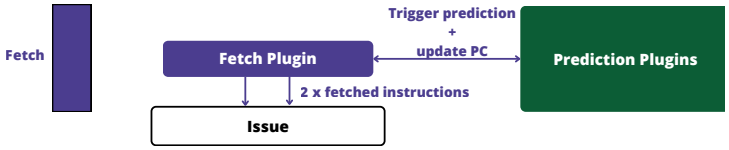


Figure 3.2 – Operations during the fetch stage

As in subsection 2.4.1, the `fetch plugin` reads `INST_COUNT` consecutive instructions starting at the current PC address, then computes the next PC value.

— When no jump request is detected from another plugin, the PC is incremented to the next instruction. As each instruction has four bytes and `INST_COUNT` instructions were fetched, the value of PC is incremented by `INST_COUNT` * 4 bytes = 8 bytes.

— When a jump request is valid, it updates the value of PC with the target address within the request.

All prediction mechanisms for control-flow instructions operate at this stage to speculatively fetch the most probable next instruction as early as possible.

A **Global History Register (GHR)** is a shift register of N bits that keeps the outcomes of recent branches. Assuming a GHR of size 4 bits, after the first four conditional branches with outcomes `taken`, `not taken`, `taken`, and `taken` respectively, the value of GHR becomes `1011`.

NaxRISCV integrates three prediction structures for control-flow instructions:

— **Branch direction speculation** (conditional branch): A table, known as **pattern history table (PHT)**, stores the result of previous conditional branches. Each entry has a 2-bit saturating counter, which is incremented if the corresponding branch is taken and decremented otherwise. PHT entries are indexed by an XOR operation between the current PC value and the GHR.

  — The branch is predicted as **taken** if the leftmost bit, also known as Most Significant Bit (MSB), is `1`.

  — Otherwise, it is predicted as **not taken**.

  This speculation mechanism is known as **GSHARE**.

— **Destination address speculation** (conditional branch/unconditional jump): A table, known as **Branch Target Buffer (BTB)**, stores recent destination addresses in a cache-like structure indexed by the control-flow instruction's address. The processor looks up the table whenever it encounters a control-flow instruction. If the instruction's address is found, a jump request is sent to the `fetch plugin` with the corresponding target address. By default, **BTB** has 512 entries. For a conditional branch, the jump request is only sent if the *branch direction speculation* predicts the instruction as *taken*.

— **Destination address speculation for return instructions:** A dedicated table, called **Return Stack buffer (RSB)**, stores the address of the next instruction following the function call instruction, as this is the instruction to be executed once the function is completed. The **RSB** is a circular buffer with 32 entries and overwrites the oldest entry if the table is full. Thus, it can achieve a perfect return speculation for up to 32 nested function calls.

Then the fetched instructions are sent to the `Decode plugin` for the next stage.

3.2 Decode stage

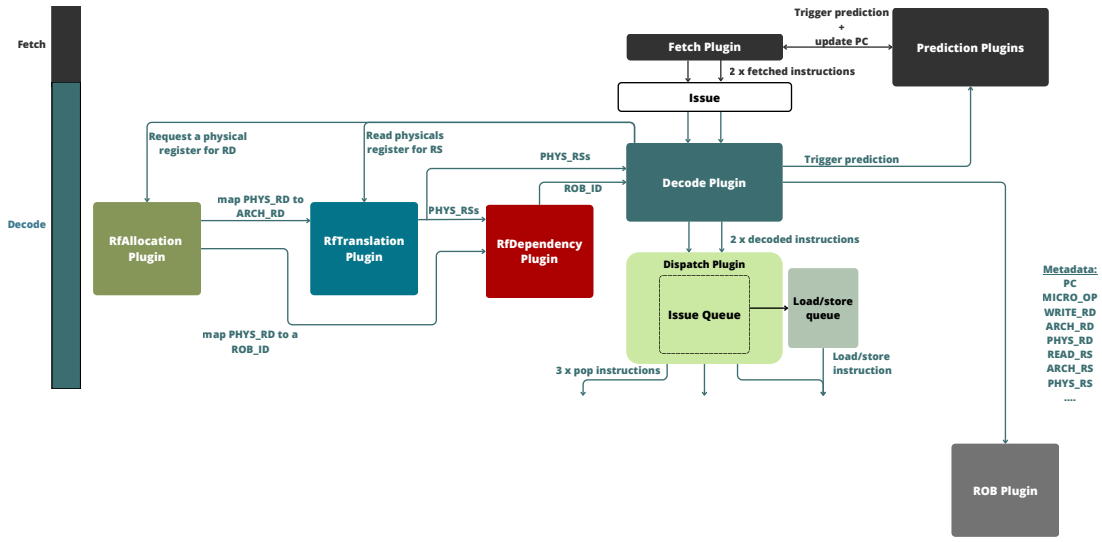


Figure 3.3 – Operations during the decode stage.

This stage is divided into three phases and each phase is performed successively.

Figure 3.4 – Three phases of decode stage

3.2.1 Decoded Phase

All authorized instructions are defined within NaxRISCV (`Rvi.scala` for integer instructions and `Rvfd.scala` for all float instructions). Based on these lists, `Decode plugin` is able to deduce the type of a fetched instruction, allowing it to interpret the instruction correctly.

Example: decoding **`add`** `x1`, `x2`, `x3` is an example of an R-type instruction. This instruction performs the operation:

$$\texttt{x1} \leftarrow \texttt{x2} + \texttt{x3}$$

It follows the standard R-type format:

Field	`funct7`	`rs2`	`rs1`	`funct3`	`rd`	`opcode`
Bits	31–25	24–20	19–15	14–12	11–7	6–0
Value for `add x1, x2, x3`	0000000	`x3`	`x2`	000	`x1`	0110011

A **variable** is a symbolic name that refers to a storage location containing a value. This value can change during the execution of a program or the operation of a hardware system.
A **Boolean variable** is a variable that stores a single bit, interpreted as either `true` (1) or `false` (0).

At this phase, each instruction possesses the following properties:

— *inst.WRITE_RD*: a `boolean` that indicates whether the instruction has a destination register.

— *inst.ARCH_RD*: the index of the destination architectural register.

— *inst.ARCH_RS1* and *inst.ARCH_RS2*: the indexes of the first and second source architectural registers.

The combination of opcode, funct7, and funct3 indicates that it is an **`add`** instruction:

— Marking the instruction as valid.

— Setting the value of *inst.WRITE_RD*, indicating that it will write to a destination register.

— Specifying the target operation as an addition.

— Setting the architectural registers to be used: *inst.ARCH_RS1* = `x1`, *inst.ARCH_RS2* = `x2`, *inst.ARCH_RD* = `x3`.

The instruction does not have physical registers associated to it yet at this phase.

3.2.2 Renaming Phase

As the microarchitecture still needs to commit in-order, a ROB table stores instructions in the program order during its lifetime in the pipeline. Moreover, it makes all information about each instruction available across different stages. ROB is a circular buffer; each slot is indexed with a unique identification `ROB_ID`.

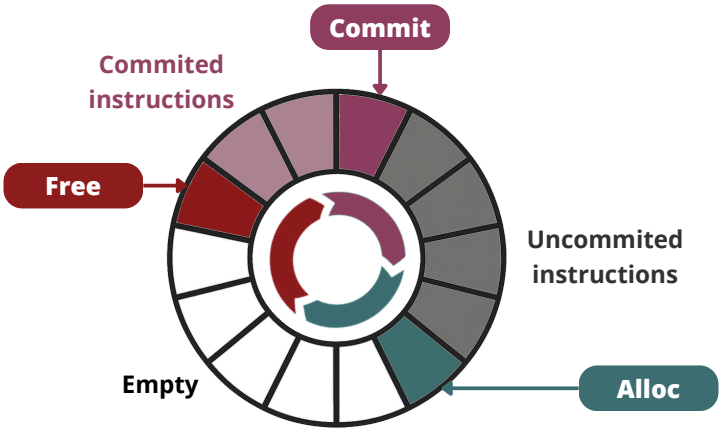


Figure 3.5 – ROB table managed by two pointers: `Alloc` and `Commit`

The ROB table is managed with three pointers:

— `Alloc` pointer: points to the next entry to be allocated. It is incremented after a new instruction is added to the table.

— `Commit` pointer: tracks the next instruction to be committed. It is incremented after the retirement of the pointed instruction. The position of the `Alloc` pointer is always ahead of or equal to the `Commit` pointer.

— `Free` pointer: points to the next entry to be released. An entry can be freed if it is both committed and pointed to by `Free`. The `Free` pointer increments as soon

as the entry is released. This pointer always lags behind, or is equal to, the `Commit` pointer, and continuously advances toward it.

The interval [`Commit`, `Alloc`) indicates the set of instructions in the pipeline that have not yet been committed. The interval [`Free`, `Commit`) indicates the set of instructions that have been committed but are still occupying entries in the ROB and need to be freed. When `Alloc` = `Commit`, no instruction is pending for commitment, and the table is considered empty.

To perform register renaming, the `decode plugin` has to gather the current physical register associated with its *inst.ARCH_RS*s and request a new physical register for the *inst.ARCH_RD* when *inst.WRITE_RD* is set. Two structures are used to manage the liaison between architectural register and physical register:

To simplify the explanation, the term *physical register* refers to its **index** within the register file not its actual hardware location. For example, *inst.PHYS_RD* holds the **index of the destination physical register**.

`RfAllocation plugin`: This plugin tracks the availability of **physical registers** and allocates an unused one when needed.

Ceiling binary logarithm, indicated as $\text{clog}_2(x)$, gives the minimum number of bits required to represent x distinct elements. Since each bit doubles the number of representable states, i.e., n bits can encode 2^n values, the smallest number of bits required is the smallest integer n such that $2^n \geq x$. By definition, this corresponds exactly to $\text{clog}_2(x) = \lceil \log_2(x) \rceil$, where $\lceil \cdot \rceil$ denotes the ceiling function.
For example, to represent 10 elements: $\text{clog}_2(10) = \lceil \log_2(10) \rceil = \lceil 3.32 \rceil = 4$ bits.

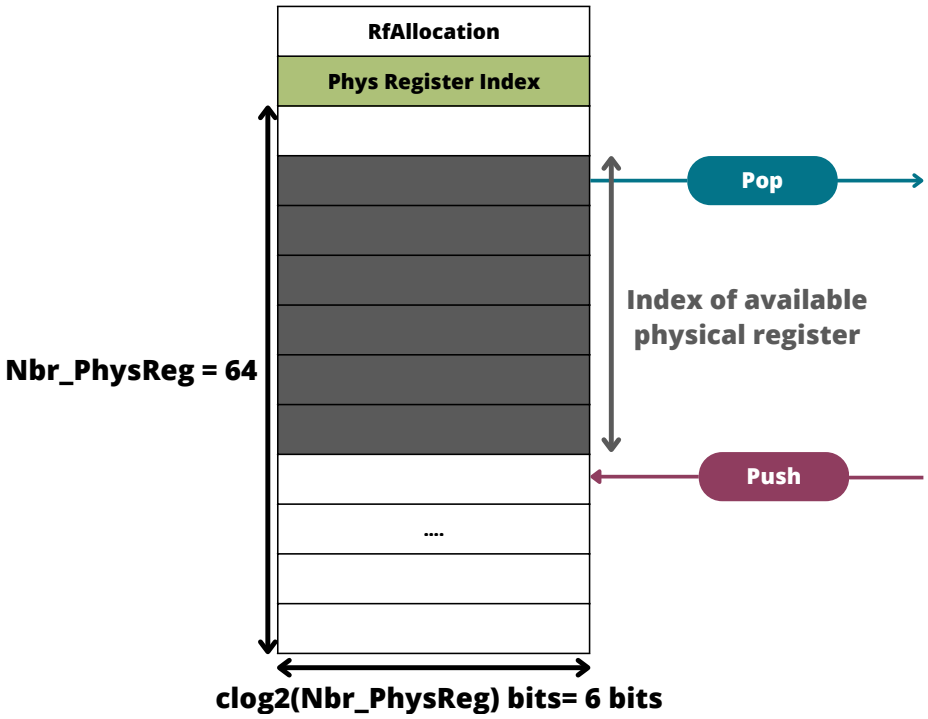


Figure 3.6 – RfAllocation Table

`RfAllocation plugin` is a circular buffer of 64 entries and uses two pointers:

— `pop`: points to the next available physical register. As NaxRISCV decodes `INST_COUNT` per cycle, it can pop up to `INST_COUNT` physical registers per cycle for each instruction that has a destination register (when *inst.WRITE_RD* is set).

— `push`: points to the entry that receives the next physical register to be released. It is also capable of pushing up to `INST_COUNT` physical registers per cycle.

`RfTranslation plugin`: Similar to the implementation of standard out-of-order described in section 2.4, the microarchitecture must track the mapping between physical and architectural registers during the execute and commit stages to support out-of-order execution. Unlike the standard implementation, which uses two separate memory locations of physical registers for the two stages, this approach gathers all values in a unified, larger *register file*. It contains 64 physical registers. Only the mapping between architectural registers and physical registers is tracked separately by `RfTranslation plugin`. This eliminates the extra copy of the physical register value from ROB to register file during the commit stage and reduces the tail-end latency of the pipeline.

This plugin uses two tables of 32 entries.

Speculative table: reflects the in-flight (tentative) rename mappings during the execute stage, a state of uncertainty caused by speculative and out-of-order execution. Each entry is indexed by an architectural register and stores the current physical register assigned by the most recent instruction.

Commit Table: stores the final mapping for each register (post-commit), used for recovery and rescheduling. Each entry is indexed by an architectural register and stores a physical register. This table is only consulted during recovery operations. After the first update of an entry, the Speculative table already contains the up-to-date mapping and takes over mapping duties.

A register of 32 bits, `Location`, selects which mapping table should be used for each register. Each bit represents one architectural register: if the bit is set, the Speculative table is used; otherwise, the commit table is used.

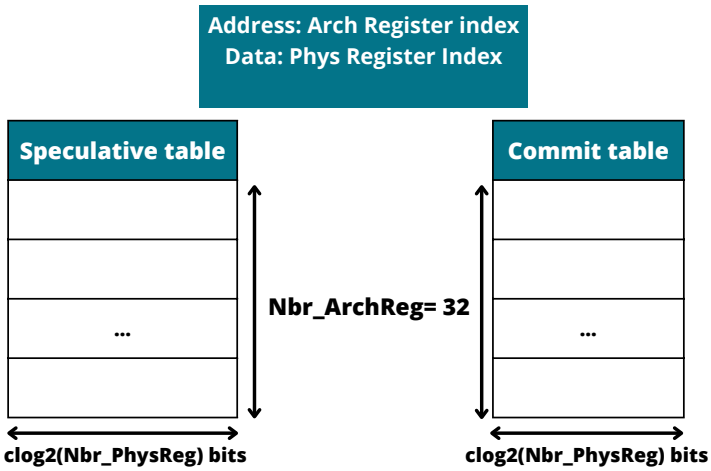


Figure 3.7 – RfTranslation tables, **Speculative table** for execution stage states and **Commit table** for commit stage states.

Beyond register renaming, each fetched instruction must be able to identify the most recent instruction that produces its source operands. This establishes the necessary dependencies, ensuring that the fetched instruction waits until those earlier instructions have executed. This task is assigned to `Rfdependency plugin`.

`Rfdependency plugin` : A table of 64 entries (number of physical registers) that ties the destination physical register directly to the last instruction that modifies it, identified by its *inst.ROB_ID*. Each entry has a `valid` bit:

— `valid = 1`, the instruction represented with its *inst.ROB_ID* was not executed yet. Thus, the associated physical register is not ready to be used by subsequent instructions. Later instructions that consume the physical register will depend on the current instruction.

— `valid = 0`, the physical register is ready and the associated *inst.ROB_ID* can be ignored.

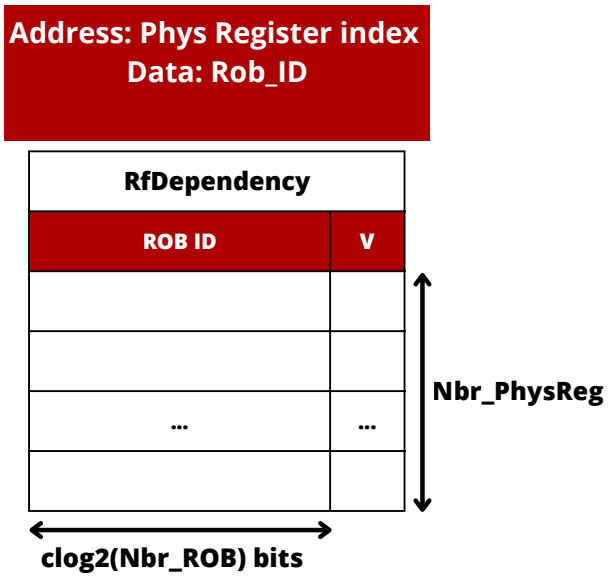


Figure 3.8 – RfDependency Table

During the register renaming operation, the `ROB plugin` allocates the next available entry to the instruction using its `Alloc` pointer. The index of the entry, `ROB_ID`, is used for other components to identify the instruction along the pipeline.

If the instruction has a destination register (*inst.WRITE_RD* is set), a new physical register is linked to the *inst.ARCH_RD*:

— The `RfAllocation plugin` releases the next available physical register using the `pop` pointer before incrementing it. The physical register popped becomes the destination physical register of the instruction.

— The `Decode plugin` backs up the old physical register associated with the *inst.ARCH_RD* in the speculative table, as it must be freed if the instruction commits.

— The `RfTranslation plugin` updates the Speculative table entry at the index *inst.ARCH_RD* with the physical register released by `RfAllocation plugin`. More-

over, the corresponding bit in the `Location` is set to one, indicating that the Speculative table has the most updated physical register.

— The `Dependency plugin` links the instruction's `ROB_ID` to the physical register released by `RfAllocation plugin`, allowing subsequent instructions to find the current instruction in case a dependency occurs.

For the source registers, the `Decode plugin` retrieves the most updated physical register associated with the *inst.ARCH_RS*s: from the Speculative table if the corresponding bit in `Location` is set, otherwise from the Commit table.

The following properties are added to the instruction:

— *inst.ROB_ID*: the index of the entry allocated to the instruction in the ROB table.

— *inst.PHYS_RD*: the index of the destination physical register.

— *inst.FREE_PHYS_RD*: the index of the old physical register associated with the *inst.ARCH_RD* before the current instruction overwrote it.

— *inst.PHYS_RS1* and *inst.PHYS_RS2*: the indexes of the first and second source physical registers.

Before the instruction is sent to the `Dispatch plugin` for the next phase, the corresponding entry in the ROB is updated with all gathered information by `Decode plugin`.

3.2.3 Dispatch Phase

The next step is to track dependencies between instructions and forward them to the next stage only when no dependency remains and the corresponding EU is not busy. During this process, decoded instructions are temporarily held in an issue queue table.

The issue queue table has 32 slots, with `INST_COUNT` columns and 32/`INST_COUNT` lines.

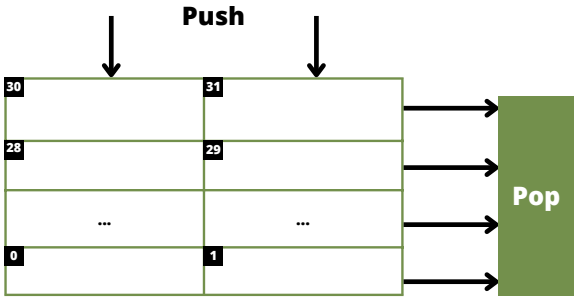


Figure 3.9 – Issue Queue table, the highest slot number (31st slot) holds the youngest instruction, while the lowest (0th slot) holds the oldest instruction. The first line (slots 31 and 30) receives new instructions, and existing instructions are shifted downward to free space for incoming ones.

Each slot has the following fields:

— **Trigger**: A sequence of bytes giving operands dependencies with older slots. Each bit represents the dependency of the instruction with the instruction in the same position of the bit on the issue queue. If the x-th bit is set, it means that the instruction depends on the instruction on the x-th slot. As a slot contains older instructions, closer to slot 0, the number of possible dependencies decreases. The `trigger` fields for each slot have a decreasing size, as the number of slots it could depend on is decreasing. Actually, the size of the `trigger` field on the x-th slot is equal to x + 1 bits: the first bit (the MSB) indicates whether the instruction writes to a destination register, and the remaining bits indicate the dependency of the instruction with older slots.

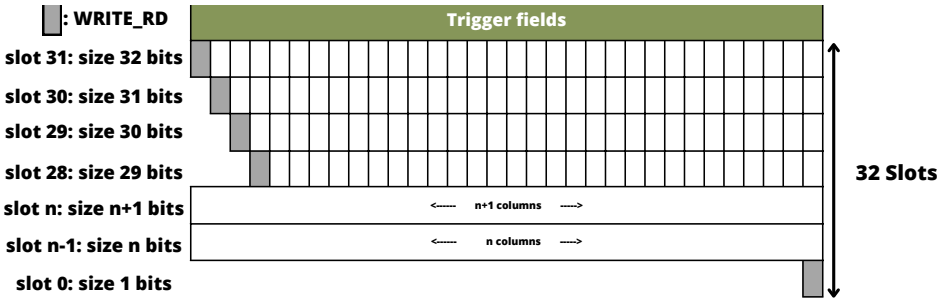


Figure 3.10 – Trigger fields

— **Sel**: identifies the EU responsible for the execution. Its value is equal to zero if the corresponding slot is not valid.

— **Context**: Regroups information about the instruction, including operands' physical registers, *inst.ROB_ID*, etc.

The issue queue is full if at least one of the `INST_COUNT` slots on the first column (0 or 1st slots) is still valid.

The `Dispatch plugin` has three main roles:

Computing dependency: It computes the `trigger` 's values for the `INST_COUNT` fetched instructions.

Thanks to the `Dependency plugin`, each instruction can retrieve the *inst.ROB_ID* of the most recent instruction that produces its required *inst.PHYS_RS* (see section 3.2.2). Since instructions are inserted in program order in both the ROB and the issue queue, the *inst.ROB_ID* uniquely identifies the provider instruction's slot in the issue queue. To compute the `trigger` field, each bit corresponding to a provider instruction's slot is set to one, thus encoding the dependency. Additionally, the MSB is set whenever the instruction writes to a destination register (*inst.WRITE_RD*).

A Load Store Unit (LSU) is a specialized EU responsible for handling all memory accesses, i.e., `load` and `store` instructions.
Store-to-load forwarding allows the LSU to directly forward the value produced by an older `store` in the pipeline to a younger `load` targeting the same target address.

In NaxRISCV, the LSU's role extends beyond executing memory operations. It manages load and store dependencies through a dedicated load/store queue, computes the addresses of memory instructions, performs **dependency speculation**, and handles store-to-load forwarding.

To improve address checking between `Load` and `Store` instructions, a particular treatment is applied to `store`: dependency on the second operand (the base address register) is ignored during the computation of the `trigger` field and is delegated to the LSU. This optimization allows early address computation during the dispatch stage and enhances the dependency handling on memory addresses.

Pushing Instructions into the Issue Queue: The issue queue is full if at least one of the `INST_COUNT` slots on the last line (0 or 1st slots) is still valid. If the table is not full, the data in each slot is shifted `INST_COUNT` times inside the table to make room for

new instructions. Then the `INST_COUNT` new instructions are placed on the first line (31st and 30th slots) of the issue queue table.

Popping Instructions from the Issue Queue An instruction is **ready** when all bits of its `trigger` field, except the MSB, are zero and the `Sel` field is not equal to zero. It can only pop if the instruction is ready **and** the EU indicated by the `Sel` is not busy.

The dependency speculation on memory instructions is performed during this operation. When a **`Load`** is ready to issue, the LSU checks all older **`Store`** instructions in the program order. If their addresses are already computed, a precise comparison is performed: if no conflict exists, the **`Load`** can safely proceed; otherwise, it must wait or receive the value through **store-to-load forwarding**. If one or more older **`Store`** instructions have unresolved addresses, the **`Load`** can speculatively issue, and the LSU revalidates the decision later. In case a conflict is detected, the **`Load`** and all younger instructions are discarded and re-executed. This mechanism allows **dependency speculation** between **`Load`**/**`Store`** instructions to improve performance, while correctness is ensured by LSU conflict detection.

3.3 Execution stage

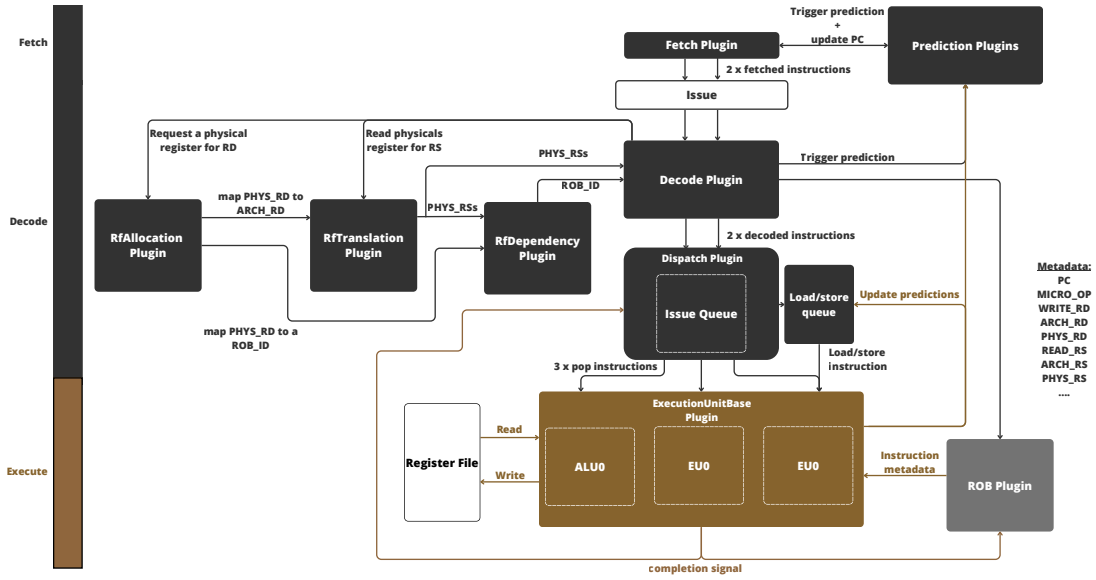


Figure 3.11 – Operations during the execute stage.

The EU identifies the instruction and executes the associated operation. It can access information related to the instruction from the ROB table using its *inst.ROB_ID*, and operand values directly from the register file. In fact, EUs are the only components capable of accessing the register file.

NaxRISCV has three EUs, each is a pipeline-based skeleton that connects with:

— The `Dispatch plugin` to read the next instruction to be executed.

— The `Rob plugin` to read the instruction data and to notify the completion of an instruction.

— The `register file` to read the value of operand registers and write the result if the instruction has a destination register.

Specific operation structures are *attached* to an EU that declares the supported instruction and links it to the operation that should be performed (e.g., Addition, Subtraction, Multiplication, etc.).

An **Arithmetic Logic Unit (ALU)** is an EU responsible for carrying out logical and arithmetic operations.

The NaxRISCV EUs are:

— **ALU0** and **ALU1**, both perform exactly the same instructions: addition, subtraction, logical operation, shift operation, branch operation.

— **EU0** performs the remaining operations: (Multiplication, Division, memory access operation (`load/store`), Csr access operations, and system call operations.

Some arithmetic operations are delegated to the EU **EU0** for the optimization of operation distribution. It aims to improve the parallelism of the execution and reduce latency.

When an instruction finishes its execution, the EU notifies the `Dispatch plugin` with a *wake signal* containing the *inst.ROB_ID* of the executed instruction. The *inst.ROB_ID* allows identifying the slot where the executed instruction is stored; thus, younger instructions set the corresponding bit in their trigger field to zero to remove the dependency. The system loses two cycles during this operation, as the EU can only send the wake signal in the cycle after completion, and updating the trigger fields takes one more cycle.

To reduce this latency penalty, the `Dispatch plugin` anticipates the latency of the execution for some operations that always take the same number of cycles to execute,

regardless of the operand values. In this case, the `Dispatch plugin` does not need to wait for the signal from the EU; it stores the amount of cycles necessary to execute the fixed latency instruction and updates the `trigger` as soon as possible. The dependency is then updated exactly at the same time as the instruction is executed.

This is a list of terms used to simplify further explanation:

- **Static Latency**: latencies of instruction with fixed execution time.
- **Dynamic Latency**: latencies of instruction with variable execution time.
- **Static Wake**: signals mechanism within `Dispatch plugin` to track and remove dependencies, only associated with instruction having static latency.
- **Dynamic Wake**: signals mechanism sent by EUs to notify the `Dispatch plugin` to remove dependencies, only associated with instruction having dynamic latency.

Simultaneously, the execution notifies the `ROB plugin` to mark the instruction's slot as executed.

When a **`store`** instruction is executed, the processor does not immediately write the **data** to the target memory address. Instead, the value is temporarily placed in a buffer called the *store queue*, located within the LSU. The data is written back to memory only when the instruction *commits*, ensuring that misspeculated stores cannot overwrite valid memory contents. To reduce latency, however, the LSU can perform *store-to-load forwarding*, allowing a subsequent **`load`** instruction targeting the same address to read the pending value directly from the store queue instead of waiting for the memory update.

Speculation gadgets refers to instructions that are able to trigger speculative execution (section 2.3.5).

If the executed instruction is a **speculation gadget**, a feedback signal is sent to the prediction mechanism, which then corrects its prediction if necessary. When the prediction differs from the result, the responsible prediction mechanism issues a rescheduling request to the `commit plugin`, specifying the *inst.ROB_ID* of the instruction.

3.4 Commit stage

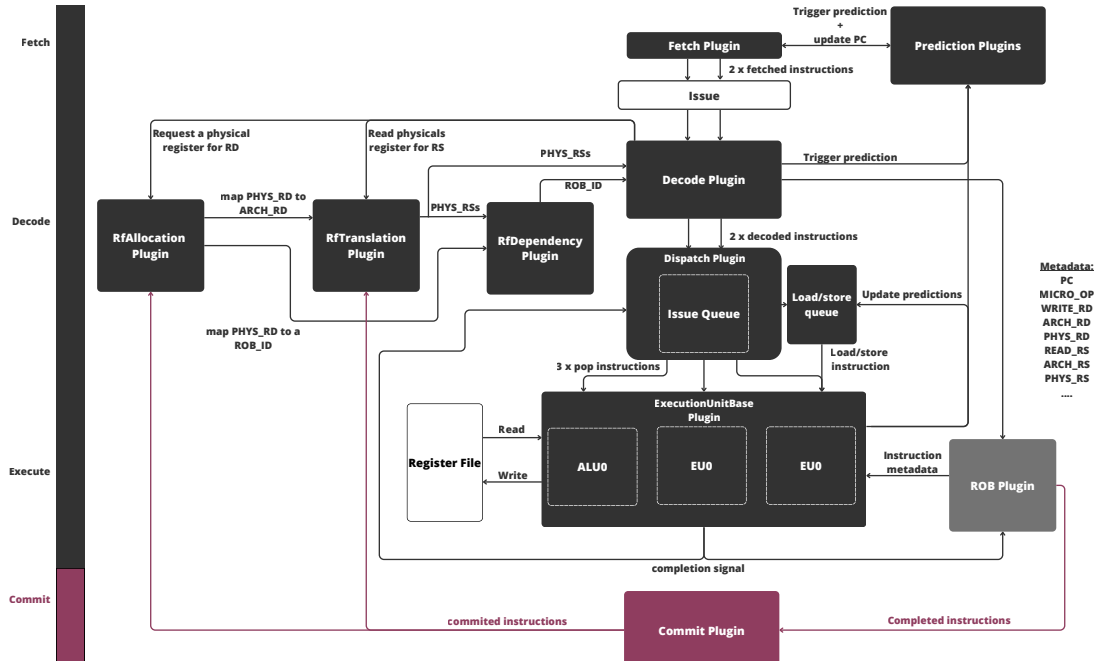


Figure 3.12 – Operations during the commit stage.

The `Commit plugin` checks whether the instruction pointed to by the `Commit` pointer in the ROB table is marked as executed by the EU.

During the commit operation of an instruction:

— The entry indexed by the *inst.ARCH_RD* in the *Commit table* (`RfTranslation plugin`) is updated with the *inst.PHYS_RD* of the instruction.

— The *inst.FREE_PHYS_RD* is released by pushing it into the *RfAllocation table*, making it available for future instructions.

— The `Commit` pointer is incremented by up to `INST_COUNT`, advancing to the next instruction to be committed.

— The old slot used for the instruction is released when the `Free` pointer reaches it.

When a misspeculation or exception occurs, the **rescheduling operation** is triggered during the commit stage of the responsible instruction, ensuring that recovery is always managed in program order. To squash subsequent instructions, the `Commit` pointer in the ROB table jumps to the `Alloc`'s position. Since the two pointers are incremented in only one direction on the circular buffer, uncommitted instructions located between the old

position of the `Commit` pointer and the `Alloc` pointer are ignored and will eventually be overwritten by future instructions. However, all destination physical registers allocated by the squashed instructions must be released. Because the *RfAllocation table* can only push up to `INST_COUNT` instructions per cycle, the `Free` pointer advances entry by entry, releasing physical registers, until it catches up to the new position of the `Commit` pointer.

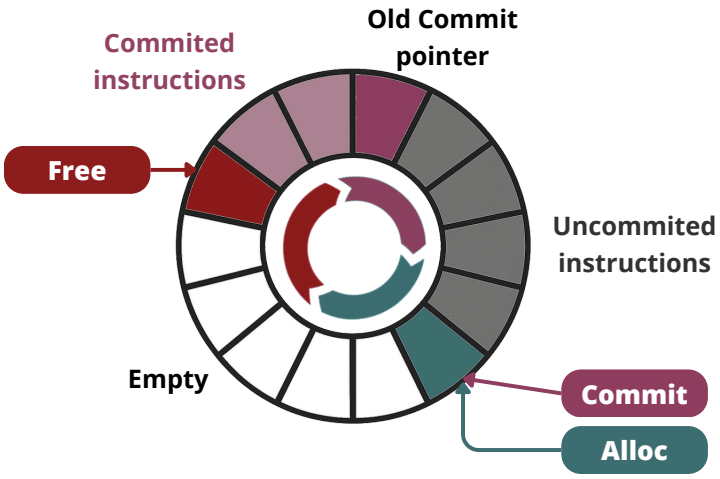


Figure 3.13 – ROB table during rescheduling operation

Chapter 4

Transient Security Flaws in Modern Architectures

Microarchitectures have constantly undergone modifications to optimize performance and efficiency. Unfortunately, security was never a primary concern during the first generations of computers, and even today it is often balanced against performance. The lack of focus on security has led to several fundamental flaws in modern architectures, potentially enabling the leakage of sensitive information within the system.

This chapter details some of the security issues related to microarchitecture design.

4.1 Silent Leaks: Side and Covert Channels in Modern CPUs

The **transistor** is an elementary building block of a computer at the hardware level. It acts as a voltage-controlled switch; when a transistor is "on", it completes a circuit (logic-1); when "off", it opens it (logic-0).

Each hardware component within the microarchitecture (registers, prediction mechanisms, cache memory, control logic, etc.) is built from logic gates, themselves built from transistors. Each data and state is physically stored as a voltage level: high voltage for logic 1 and low for 0. Every logical operation toggles these transistors, causing current to flow, electromagnetic fields to shift, and signals to propagate through the chip, which inherently consumes power, induces small delays, generates heat, alters the states of microarchitecture components, etc. Crucially, these analog side effects depend on the exact operations and data values: all computation perturbs the hardware's physical state in measurable ways. An observer may measure and correlate these side effect signals to the secret data being processed; it is known as a **side-channel** attack. These unintended physical side channels are a fundamental, system-wide challenge in microarchitectural

design.

One of the best-known side channels is based on timing differences caused by the performance advantage of cache memories[9], [51] (cache memory is detailed in section 2.3.1). Recently accessed data are stored temporarily in cache memory, making the next access to the same data much faster. The timing gap between a **cache miss**, when the data is absent from the cache, and a **cache hit**, when the data is already stored there, is large enough for an observer to accurately infer the data presence in the cache and deduce whether it was recently used.

A **covert channel** is a specific exploitation of the side effects, designed to establish an unauthorized communication method between two different security domains. While side-channel attacks leverage side effects unintentionally produced by legitimate programs, covert-channel attacks deliberately execute malicious code on the target system to reproduce side effects in the microarchitecture, aiming to exfiltrate information to an observer.

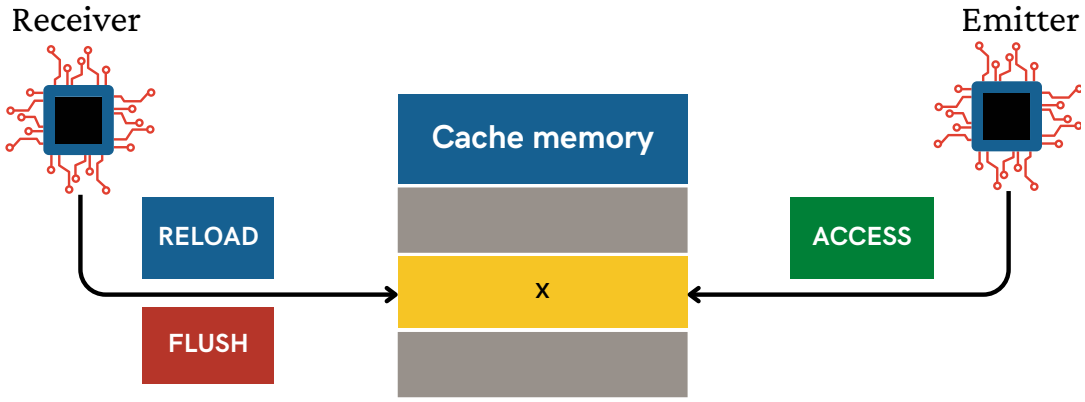


Figure 4.1 – A covert channel is an unauthorized communication channel where the attacker controls both the emitter and receiver. Here, the attacker leverages shared data in the cache memory to create a covert communication.

An **Operating System (OS)** is a system software that provides standardized interfaces to applications and manages hardware resources (e.g., processor, memory, privilege levels, input/output device, etc.) available to each application. Operating systems are commonly used in general-purpose systems to simplify resource management, e.g., Linux, Windows, macOS/iOS, but may still be absent in certain use cases, since programs can also run directly on bare hardware.

As illustrated in Figure 4.1, the two processes should not be able to communicate with

each other without the operating system's permission. However, the programs executed on both processes may rely on a shared library. For efficiency, the operating system maps the code segments **X** of the library into a shared memory page, which can be exploited as a communication medium.

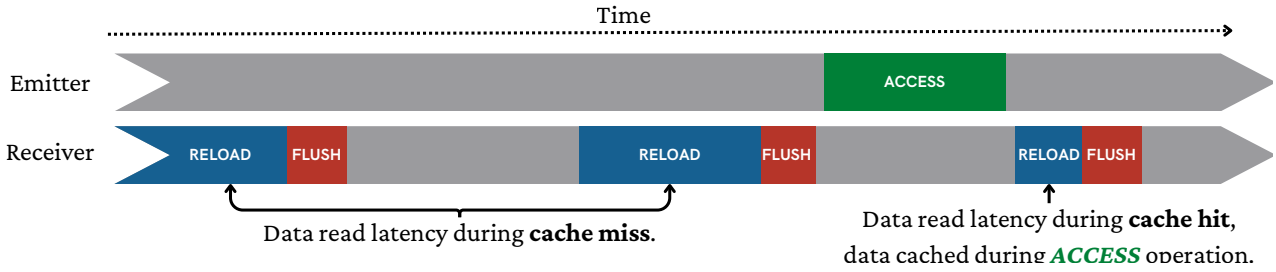


Figure 4.2 – Time evolution during covert channel exchange. The latency of the *reload* operation on the receiver is controlled by the emitter. It has low latency when the emitter performs the **ACCESS** operation.

The *receiver* repeatedly reloads and evicts **X** from cache memory while measuring the time spent during the load operation. Meanwhile, the emitter accesses **X**, thereby bringing it into the cache memory. Thus, the receiver's subsequent reload times for **X** reveal whether the emitter has accessed the data or not: a low latency access indicates that the emitter accessed the shared data, while a high latency indicates the absence of such access. Despite the exchanged information being minimal, this mechanism is sufficient to establish a sophisticated covert communication channel.

An example of a covert channel exploitation is leaking data across privilege levels. Privilege levels are defined by the architecture itself, but they are mainly exploited by operating systems in general-purpose systems to enforce isolation. In bare-metal or embedded systems without an operating system, software often runs entirely at the highest privilege level, leaving the others unused.

A **firmware** is a low-level software stored in permanent memory that initializes and manages hardware before or beneath the operating system.

In the `RISC-V` architecture, three privilege levels are defined:

— **Machine mode** (M-mode): the highest privilege level, typically reserved for low-level firmware and system management tasks.

— **Supervisor mode** (S-mode, also referred to as *kernel mode* in operating systems): used by the OS to manage hardware resources and provide services to applications.

— **User mode** (U-mode): the least privileged level, corresponding to application execution in operating systems, with restricted access to hardware resources.

> A **system call**, or **syscall**, is a mechanism allowing a *user program* to request the *kernel* to perform a privileged operation on its behalf, such as handling an exception.

User privilege is restricted and must request services from the kernel through system call mechanisms.

The kernel data must remain protected and inaccessible from user mode. However, an attacker may gain control of a small piece of privileged code that reads and leaks kernel data through side effects, e.g., using the cache memory. In this scenario, the privileged block of code acts as the *emitter*, while a user privilege program acts as the *receiver*.

The cache memory is only one of the many microarchitecture components that can be exploited with covert and side channel attacks; other examples include page tables[11], Translation Lookaside Buffer (TLB)[26], BTB[2], branch direction predictors[2], shared EUs[2], prefetchers[64], port contention[4], cache controllers[63], etc.

4.2 Out-of-Order Chaos: Meltdown 

> In out-of-order processors, the program order is meaningless at the microarchitecture level as long as dependencies are respected. Even so, commit operations are always performed in order, ensuring that only instructions allowed to execute can commit.

It was long assumed that an instruction removed from the pipeline prior to commitment posed no security risk as its effects would never propagate beyond the microarchitecture. Hence, it is common in modern processors to execute instructions that may never actually be executed, since the program path often takes time to resolve. The execution of an instruction can only be confirmed during the commit stage.

Crucially, even a flushed instruction can still alter certain states in the microarchitecture. These states may subsequently be observed through side-channel measurements.

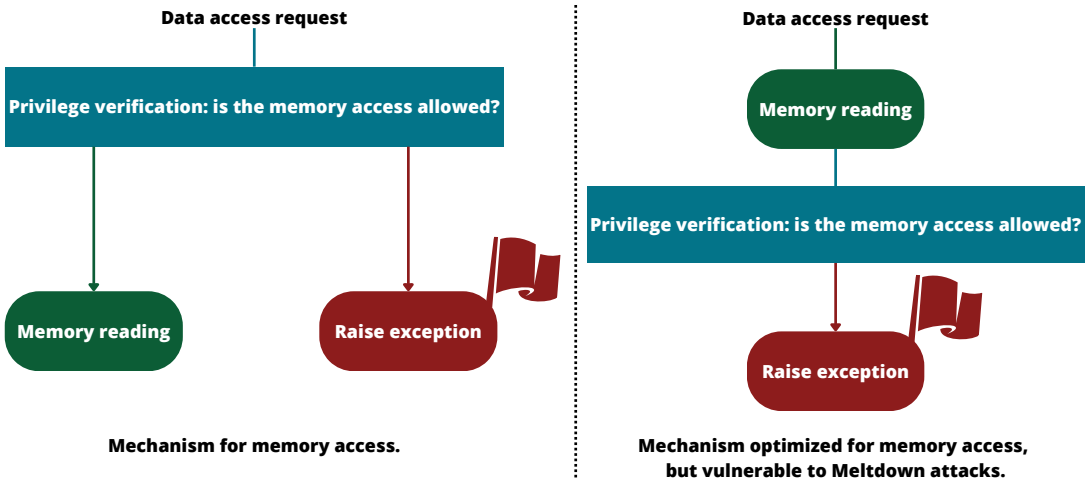
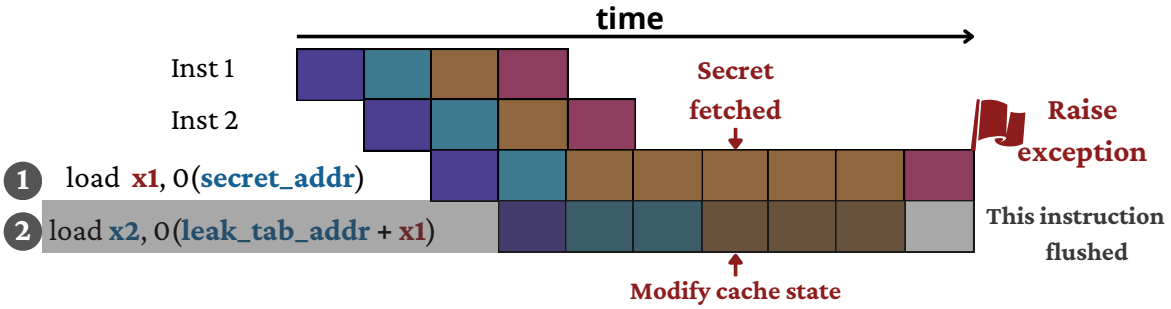


Figure 4.3 – Mechanism for memory access. Some processors rearrange the steps to reduce latency, but this creates a new vulnerability to Meltdown.

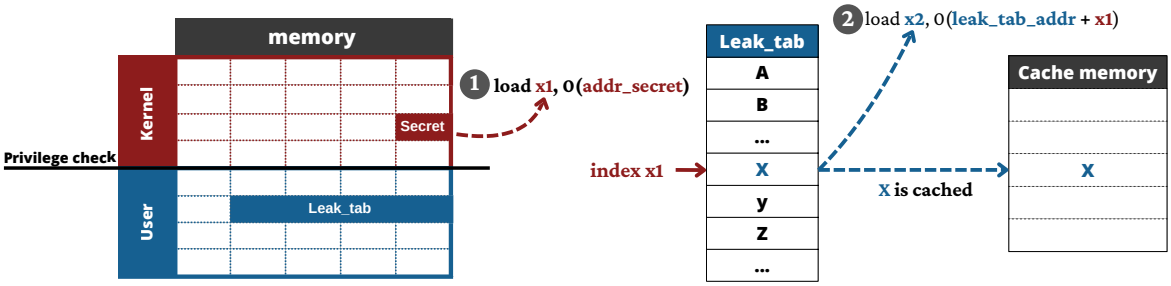
Transient execution is a broad term referring to any execution that happens "provisionally" and may later be discarded, whether due to misspeculation or out-of-order scheduling.

To improve performance, some out-of-order processors perform memory accesses without first verifying that the access has the right privilege (as illustrated in Figure 4.3). In any case, such an instruction can only retire once the privilege verification has been resolved. Thus, the loaded value is forwarded to dependent instructions without waiting for the privilege verification, reducing stall states. This creates a window of transient execution in which subsequent dependent instructions execute before the exception is raised, potentially leaking data. This vulnerability is known as **Meltdown** [13], [40].

As it exploits a mechanism that bypasses architectural checks by loading data without prior privilege verification, Meltdown is capable of leaking data *across privilege boundaries*.



(a) Load access to secret data before the privilege check.



(b) The secret loaded is used as an index in a user table data (`leak_tab`) to leave a trace measurable by an observer in the cache memory.

Figure 4.4 – On some processors, a load always reaches the data despite it not having the authorization. The data can be leaked in microarchitecture state (e.g., cache memory) before the exception is raised.

Considering a secret data that should be only accessible with kernel mode. If the access is attempted by a user privilege process, an exception should be raised, redirecting execution to the exception handler.Meanwhile, a shared table, `leak_tab`, is accessible at any privilege level (*user data*).

Since the memory access of the `load x1, 0(addr_secret)` is performed regardless of the outcome of the privilege verification, subsequent instructions can manipulate the secret value and leave observable traces in the microarchitecture, as illustrated in Figure 4.4a. Figure 4.4b shows the data flow during transient execution. The following instruction, `load x2, 0(leak_tab_addr + x1)`, uses the secret value stored in `x1` as an index to fetch an element (`X`) from the shared table `leak_tab`. At the same time, a copy of `X` is placed in the cache memory for future access. The exception is triggered only when the first `load` reaches the commit stage, flushing the incorrectly executed instructions from the pipeline and redirecting the execution. Unfortunately, `X` remains in the cache memory. By measuring the access latency of each element in the shared table, an attacker

can detect which entry is cached (low latency) and thereby deduce the value of the secret, which corresponds to the index of `X`.

In practice, constructing such a leakage channel is more involved than this simplified description suggests. A more detailed explanation of this mechanism is provided in section 4.3, which is dedicated to the Spectre vulnerability, the main focus of this manuscript.

4.3 The Security Risks of Speculation: Spectre

A **speculation gadget** refers to instructions that are able to trigger speculative execution.

A transient execution flaw can also occur during misspeculation. In this case, the processor incorrectly predicts the outcome of a speculation gadget, causing some instructions to be speculatively executed with the wrong assumption. Such misspeculative execution allows an attacker to leak sensitive data and is known as the **Spectre** [35] attack. Spectre is particularly challenging to mitigate, as speculative mechanisms are deeply integral to modern processor design, and each can be independently exploited to leak data.

Spectre processes in three stages:

— Predictor Manipulation

— Transient Execution and Data Leakage

— Covert Channel Extraction

4.3.1 Predictor Manipulation

To induce the misspeculation, an attacker can deliberately influence the speculation mechanism to increase the likelihood of misspeculation.

Speculation mechanisms base their prediction on the historical outcomes of past executions, though their adaptation strategies differ.

The **GSHARE** is a branch direction speculation on conditional branches that index the PHT with the combination, by an XOR operation, between GHR and the PC.

To illustrate how Spectre works, the explanation here is based on an example attack that exploits the branch direction speculation using a **GSHARE** predictor in

NaxRISCV.

```
if(s < array1_size){
    y = array2[array1[s] * 64];
}
```

Figure 4.5 – Example of C code vulnerable to the Spectre-PHT attack.

The goal is to train the prediction to misspeculate by making the prediction mechanism believe that the outcome is likely to execute a specific block of code chosen by the attacker. For clarity, we refer to this attacker-controlled block as the **Spectre gadget**.

In Figure 4.5, the `if` condition is repeatedly executed with `s` < `array1_size`, making the condition outcome **`true`** and the `if` body run each time. Each such outcome updates the branch's 2-bit saturating counter in the PHT. With every repetition, the counter of the corresponding entry increments toward its maximum value, setting its MSB to `1`. After sufficient repetitions, the predictor will *confidently predict* the branch condition as **`true`** in future executions of the same `if` condition.

4.3.2 Transient Execution and Data Leakage

The same code is executed again, but now with a condition that should evaluate to false (`s` ≥ `array1_size`), where `s` is controlled by the attacker. Still, the **GSHARE** is trained to predict *true* during the predictor manipulation, leading the processor to miss-predict and speculatively execute the code inside the `if` body.

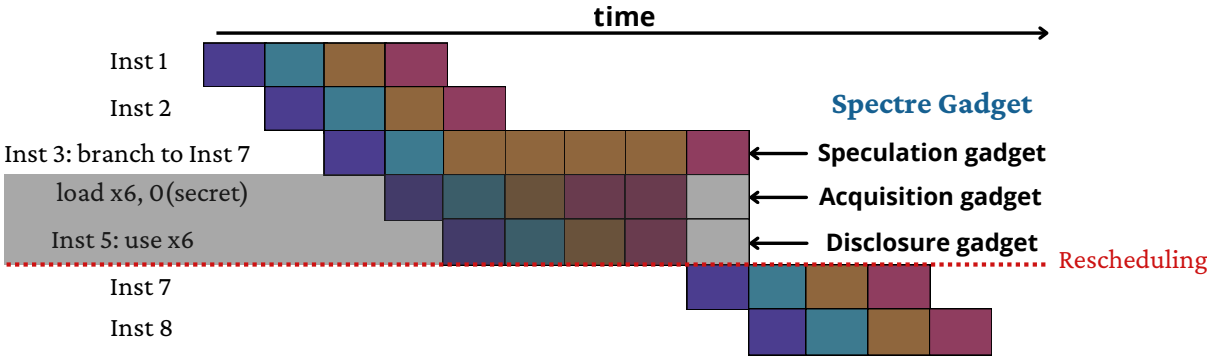


Figure 4.6 – Pipeline of Spectre gadget execution. The speculation gadget triggers a misspeculation, while the two last instructions leak a secret in the microarchitecture during the misspeculation window.

During this phase, a `Spectre gadget` is executed, structured around three key elements (illustrated in Figure 4.6):

Speculation gadget: This term was already introduced earlier and has been used throughout the previous chapters. It indicates an instruction capable of triggering speculative behavior. In Figure 4.5, this is the `if` condition, which corresponds to a conditional branch. Its role is critical as the last two steps of the attack must occur within a misspeculative window. In practice, any instruction that compels the microarchitecture to speculate can be classified as a *speculation gadget*.

Spectre variants are categorized by the speculation mechanism exploited during the attack. The four most well-known Spectre variants are listed below. Importantly, NaxRISCV is vulnerable to these variants, as each exploits a mechanism present in the processor:

— **Spectre-PHT** [14], [35] leverages the branch direction prediction via the PHT. This variant targets conditional branch speculation, as the case for Figure 4.5.

— **Spectre-BTB** [8], [14], [35] leverages the branch destination speculation via a BTB. This variant targets indirect branch speculation.

— **Spectre-RSB** [14], [37], [42] exploits the specific branch destination speculation on return instruction using RSB. This variant targets return operation speculation.

— **Spectre-STL** [14], [30], [37] exploits the dependency speculation in LSU. This variant targets store/load dependency speculation.

This adaptability of Spectre to exploit different speculative mechanisms makes comprehensive mitigation difficult without negatively impacting processor performance.

An optional **delay block** pre-stage may be performed before the execution of the `speculation gadget` stage to increase the success rate of Spectre. It consists of inserting dummy instructions with high latency and strict inter-instruction dependencies, forcing them to execute sequentially and thereby deliberately increasing their execution time. Then making the *speculation gadget* depend on this block of code instructions, thereby delaying its execution. In other words, this technique enlarges the misspeculation window, providing sufficient time for subsequent stages to leak data.

Acquisition gadget: An instruction that loads the secret data from memory during the misspeculation. On most architectures, and particularly on `RISC-V` , the only instruction capable of reading data outside the register file is the `load` instruction. In our example,

this corresponds to **`array1[s]`**, which reads a single **byte** at index `s` in the table `array1`. It can be interpreted as a **`load`** from the address `array1` + `s` in assembly language.

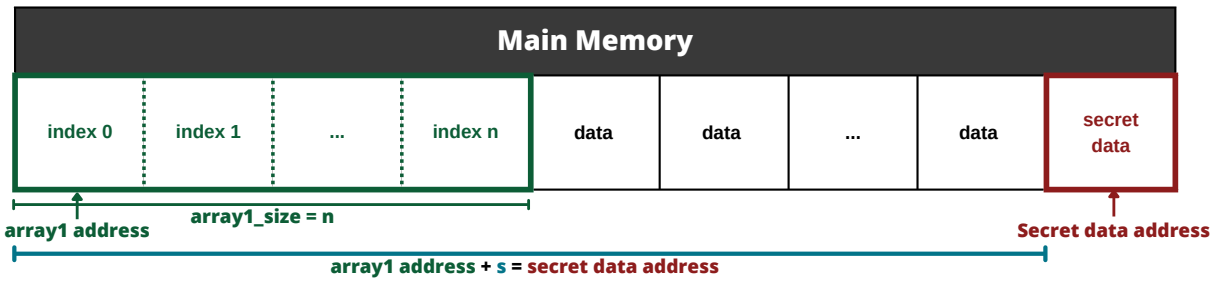


Figure 4.7 – Acquisition gadget targeting secret data.

Since `s` is out-of-bounds and controlled by the attacker, its value can be set to reach speculatively one byte from secret data located at `addr_secret` = `array1` + `s`, denoted as `S`.

Disclosure gadget: Reaching the secret data is not enough because the *acquisition gadget* instruction is flushed as soon as the responsible **speculation gadget** commits. Furthermore, the secret itself is not shared with other processors, making a side-channel based directly on the cached copy of the secret data ineffective. Thus, a second instruction is required to translate the secret byte `S`, which temporarily exists in the register, into a persistent microarchitecture state that survives the rescheduling operation. A state that the attacker can later measure.

It can be achieved with various instructions, as long as they affect the state of the microarchitecture. The disclosure gadget is the **`array2[array1[s] * 64]`** in Figure 4.5, or more precisely **`array2[S * 64]`**, since **`array1[s]`** equals `S`. Here, the table **`array2`** is a shared data accessible by other processors, which may, for example, be a shared library deliberately set up by the attacker. By using `S` as an index, the accessed element in **`array2`** depends directly on the secret's value.

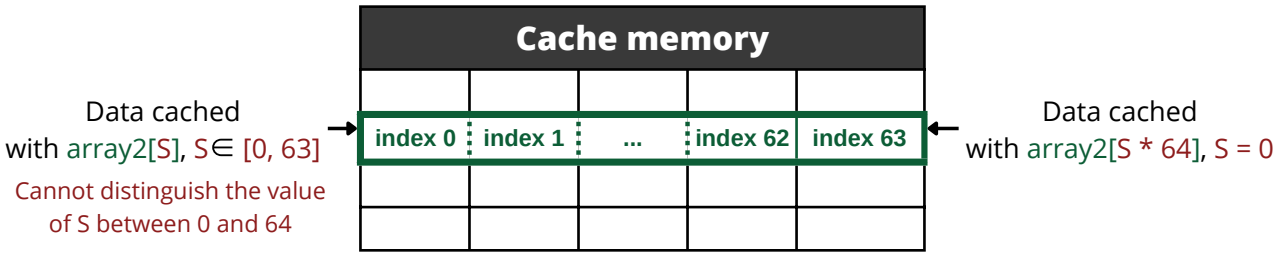


Figure 4.8 – Data are organized into cache lines. On NaxRISCV, each cache line is 64 bytes, so accessing a single byte also loads the remaining 63 bytes of the same line.

One complication arises from the fact that cache memory accesses fetch entire cache lines rather than a single byte. Assuming the **`array2`** is aligned (the first element is placed at the start of a cache line). In the NaxRISCV processor with a 64-byte cache memory size, addresses from 0 to 63 are mapped in one line, 64−127 to the next, and so forth. Therefore, 64 different values of `S` fall into the same cache line, making it impossible for the attacker to distinguish them by measuring access time. To resolve this, `S` is multiplied by the cache line size (64 in our example), ensuring that each possible `S` value maps to a distinct cache line.

The disclosure process described above was simplified for explanatory purposes. Modern cache memories use techniques such as set-associative cache mapping to improve performance, at least in NaxRISCV. These details were omitted here to avoid unnecessary complexity, since they do not affect the fundamental principle being explained.

4.3.3 Covert Channel Extraction

If a section of the **`array2`**, equal to one cache line size, is stored in cache memory, its index has a direct correlation with the secret byte. Unlike the secret access, retrieving this element is allowed since **`array2`** is shared data. Similar to Meltdown, measuring the access time of each element allows an attacker to precisely identify which entry was speculatively cached during the transient execution, and thereby deduce the value of the secret byte.

While the example leverages the cache memory to exfiltrate the secret data from the microarchitecture, Spectre can exploit a variety of microarchitecture states. As discussed in section 4.1, covert channels are not limited to cache memory; securing cache memory alone does not mitigate Spectre, although it reduces the number of available leakage

paths. Indeed, research has exposed variants of Spectre attacks exploiting components beyond the cache memory, such as branch target buffers[43], vector instructions[62], port contention [10], [24], etc.

Variant	**Attack Name**	**Code Name**	**Category**
Variant 1	Bounds Check Bypass	PHT (Pattern History Table)	Spectre
Variant 2	Branch Target Injection	BTB (Branch Target Buffer)	Spectre
Variant 3	Rogue Data Cache Load	Meltdown	Meltdown
Variant 3a	Rogue System Register Read	Meltdown Variant 3a	Meltdown
Variant 4	Speculative Store Bypass	STL (Store-to-Load)	Spectre
Variant 5	Return Stack Buffer Injection	RSB (Return Stack Buffer)	Spectre

Table 4.1 – Some representative transient execution vulnerabilities (including Spectre and Meltdown variants; **not exhaustive**).

Chapter 5

Securing Microarchitectures: State-of-the-Art Mitigations

This manuscript focuses on transient execution vulnerabilities. Although side channels and covert channels are integral components of both Spectre and Meltdown attacks, they remain outside the scope of this work and are treated as another kind of vulnerability within the overall attack.

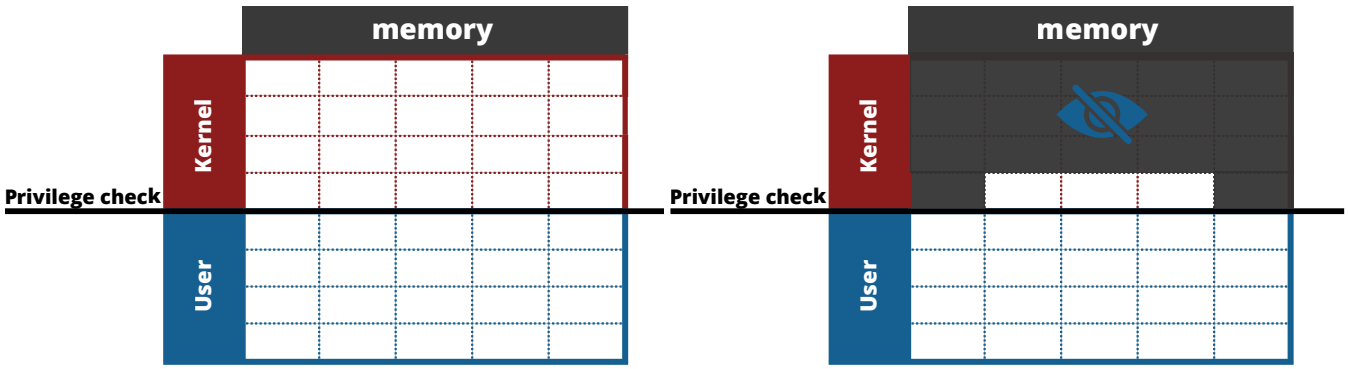
Since the discovery of Spectre and Meltdown in 2018, numerous patches and mitigations have been proposed to protect microarchitectures. This chapter presents the state-of-the-art in proposed mitigations against transient execution attacks. We place greater focus on Spectre than on Meltdown, since its variants are more diverse, harder to mitigate comprehensively, and continue to pose significant challenges for both hardware and software mitigations.

5.1 Containing the Chaos: Mitigations for Meltdown

The kernel often exposes its entire memory within the address space of all programs, relying on privilege checks to verify that only programs with the appropriate permissions can access kernel data. This design makes the system call operation faster by reducing the cost of context transitions between user and kernel modes, since the kernel memory remains directly addressable.

On some processors, this allows data leaks via Meltdown when memory access is performed before privilege verification.

To address this vulnerability, operating systems reinforce the separation between user programs and the kernel, complementing traditional privilege checks. It no longer maps the kernel memory during the execution of user-privilege programs. Hence, access is granted only when the processor switches to a privileged context during a system call. This mechanism prevents Meltdown from transiently reading kernel data, even in the absence of



(a) Without Meltdown mitigation (b) With Meltdown mitigation

Figure 5.1 – In Figure 5.1a, a user-mode program can point to kernel data even when it is forbidden; on some processors, this can lead to a Meltdown vulnerability if the privilege check is performed too late. In contrast, as shown in Figure 5.1b, the KPTI mitigation protects kernel data by restricting user-mode visibility to a minimal portion of the kernel, preventing access to the remaining kernel memory during transient execution. The exposed portion is sufficient to allow user processes to perform system calls safely.

explicit privilege verification.

The most widely deployed fix with this approach is called Kernel Page Table Isolation (KPTI), used by both Intel and vulnerable Advanced RISC Machines (ARM) processors.

This mitigation is implemented in software and integrated directly into the OS. Only later processors integrate hardware protection enforcing isolation between privilege levels.

In practice, a portion of the kernel must remain accessible so that user programs can still perform system calls. When the user requests the intervention of the kernel through a system call, the OS switches control to the kernel, which performs the privileged operation requested by the user. This exposed part of the kernel never contains sensitive information; hence, no secrets are leaked even if Meltdown reads it.

Since Meltdown stems from a performance optimization that does not respect the architecture rules, it is relatively easier to mitigate compared to Spectre, which leverages fundamental behavior deeply embedded in modern processors. This significantly reduces the impact of Meltdown and also lessens its relevance in the context of this research, which focuses on `RISC-V` .

5.2 In Pursuit of Security: Spectre Mitigations

In contrast to Meltdown, Spectre lacks any mitigation that clearly stands out as both effective and practical. Despite numerous proposals, no defense has demonstrated complete protection with acceptable performance overhead while remaining adaptable across architectures. As a result, there is still no definitive path forward, and the handling of Spectre remains an open challenge.

Since the first disclosure of Spectre, numerous research efforts have focused on developing mitigations aimed at disabling the attack or at least reducing its scope. However, Spectre's ability to adapt by exploiting different components and weaving its way through the microarchitecture not only forces protections to evolve constantly but also makes it extremely difficult to validate that any single solution can cover all Spectre variants.

The existing solutions can be categorized into two broad implementation domains: those that only use software structures and those that involve modifications to the processor hardware. Some hardware-based mitigations may have a software-side component, but we consider a solution to be a hardware mitigation as long as it requires a hardware modification.

Selective speculation is a technique consisting of preventing certain instructions from executing speculatively while still allowing speculation for others.

In this section, mitigations of Spectre are primarily grouped by their implementation domain (software or hardware), each of which encompasses multiple approaches to addressing Spectre. For clarity, we further organize the mitigations within each domain according to their technical strategy. That said, many mitigations can also be categorized in multiple ways. Whenever relevant, we highlight whether a given technique belongs to the broader class of **selective speculation** approaches, which represents the main focus of our discussion.

Since performance results in the literature are often contradictory (see section 5.2.3), mitigation performance-loss values are omitted here, as they would not be generalizable.

5.2.1 Software Mitigation

Software mitigations are mostly used as temporary defenses since they are simpler to deploy and more flexible to adapt across different architectures. However, because the principal cause of Spectre lies in microarchitectural flaws, protections applied at higher

levels such as software often lack the precision of hardware-based mitigations, which can be observed when comparing their effectiveness.

A **Spectre gadget**, the block of code that can be exploited by an attacker to leak data speculatively, is composed of three key elements:

- *Speculation gadget:* any instruction that triggers speculative execution.
- *Acquisition gadget:* any speculatively executed instruction and access the targeted secret value from memory (e.g., a `load` instruction executed speculatively).
- *Disclosure gadget:* any speculatively executed instruction that translates the data loaded by the *acquisition gadget* into a change in microarchitectural state.

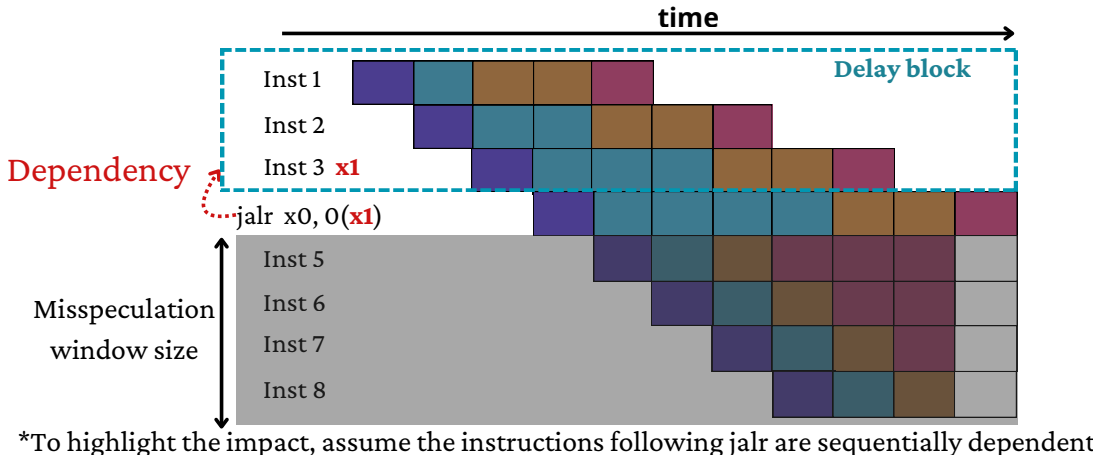
Software mitigations generally fall into three distinct approaches:

Barrier Instruction-Based Insertion

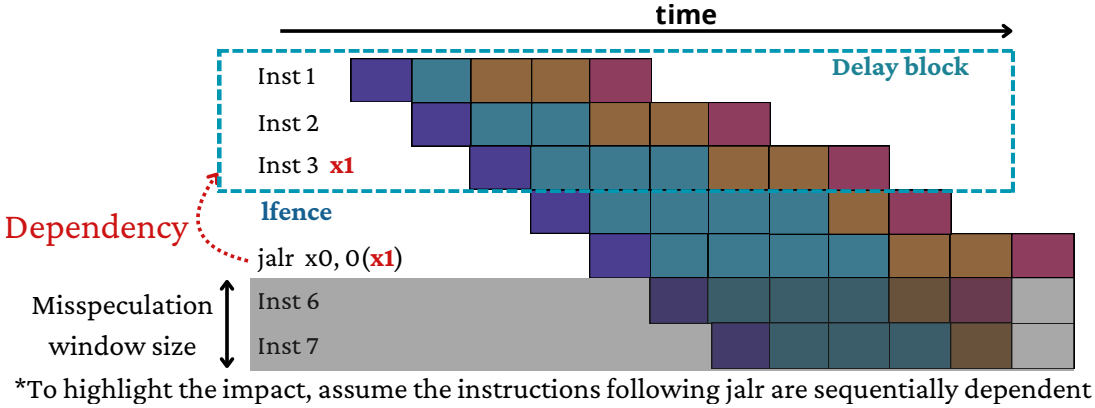
A barrier instruction is a special instruction that adds an ordering constraint on the execution of instructions. It prevents certain reordering by the processor or memory system, ensuring that specified instructions complete its execution before subsequent instructions are allowed to proceed. It can make the Spectre gadget unused when perfectly placed inside the code, but it all depends on the position where it is inserted.

A mitigation that uses such an approach is the **LFENCE/JUMP** mitigation. It is also one of the first software mitigations proposed shortly after the discovery of Spectre, in 2018.

The **Jump And Link Register (jalr)** instruction performs an indirect jump to the target address by adding an immediate offset to a base register; the full semantic is detailed in [55].
An **indirect branch** instruction is a control-flow instruction in which the target address is not fully encoded in the instruction itself but instead taken from or computed using a register, such as `jalr` in `RISC-V` . Some branch/jump instructions embed constant values, called **immediate**, within the instruction encoding.



(a) The execution of `jalr` is delayed by the **delay block**, increasing the width of the mispeculative window. This is a Spectre-BTB like.



(b) Spectre-BTB with **Load Fence (LFENCE)/JUMP** mitigation. The `lfence` instruction delays all subsequent instructions to execute before the **delay block** is executed, reducing the speculation window during the branch execution.

Figure 5.2 – `lfence` makes sure all operands and dependencies are resolved before the execution of the branch

The `LFENCE` instruction is a serializing barrier: no subsequent instruction can execute, not even speculatively, until all prior instructions have completed. It is implemented on both AMD and Intel processors, though historically with differences in their microarchitectural interpretation. After the discovery of Spectre, however, its semantics were revised on both sides and aligned more closely to better address the threat.

The goal is to provide protection against the *Spectre-BTB* style by inserting a `LFENCE` instruction before every indirect branch in the program. Instead of restricting the speculative execution, **LFENCE/JUMP** mitigation aims to reduce the speculative window,

preventing Spectre from having enough time to leak data. The **`LFENCE`** instruction is placed before the *Spectre gadget* to delay its execution until all previous instructions (*delay block*) have committed, ensuring that all registers are available before the execution of the speculation gadget.

Some other mitigations followed this path using different insertion policies of **`LFENCE`**, such as **Speculative Execution Side Effect Suppression (SESES)** . Instead of delaying the *speculation gadget* as LFENCE/JUMP, SESES delays the *acquisition gadget* by placing the barrier instruction before all possible *acquisition gadget* instructions [73]. It is considered the *last resort* mitigation, as it ensures that no **`load`** can be executed speculatively, providing full protection against Spectre but imposing a very high performance impact.

A pass is a single analysis or transformation step applied by the compiler to a program's intermediate representation, typically as part of a sequence of passes that progressively optimize or lower the code toward the final executable.

SESES is a form of *selective speculation*, since it restricts **`load`** instructions from executing speculatively. It is implemented as an optional compiler pass in LLVM.

Thunking the Control-Flow

Some mitigations modify the control-flow to remove the vulnerable code from the program. This necessitates a source code modification to change the control flow shape rather than inserting barrier instructions.

```
                    jal set_up_target; # ra <- (capture_spec)
                    capture_spec:
                    pause;
jr a0;              j capture_spec;
                    set_up_target:
                    mv ra, a0; # ra <- a0
                    ret; # jr ra
```

Figure 5.3 – Assembly code of the Retpoline mitigation, which replaces an indirect jump with a secure return-based sequence.

An example that follows this principle is the **retpoline** [1] (short for *return tram-*

poline) mitigation. Retpoline is designed to defend against speculative execution attacks that target indirect branches, particularly the mechanism of branch destination speculation (*Spectre-RSB*). It works by transforming the program's return paths into a carefully crafted sequence that forces speculative execution into a harmless loop while the actual return target is resolved correctly along the architectural execution path. By confining speculative execution to this safe sequence, retpoline prevents attackers from redirecting it to access or leak sensitive data.

Figure 5.3 illustrates the retpoline sequence, which replaces an indirect jump (`jr` with target in `a0`) by the following operations:

1. Execute `jal set_up_target`. This saves the next instruction, `capture_spec`, which is the return address, in `ra`, and pushes it onto the **RSB** (RSB is explained in section 3.1). It then jumps to `set_up_target`.
2. Copy the intended jump target from `a0` into `ra`.
3. Execute `ret` (an alias for `jalr x0, ra`). The return mechanism triggers speculative execution while the processor predicts the return address.
4. The RSB contains the saved return address from step 1, so the processor is likely to predict a return to `capture_spec`, not the address now in `ra`.
5. As a result, the processor speculatively continues execution at `capture_spec` until the misprediction is detected and the correct target (the value in `a0`) is used.

Although this approach was widely deployed during the first attempt to mitigate Spectre, it introduces additional control-flow overhead and specifically protects only against attacks that manipulate indirect jumps. Moreover, retpoline mitigation broadly suppresses speculation on returns instead of precisely countering the attack itself. Several studies discuss the effectiveness and security of this mitigation [38], [47]. They reach the same conclusion: the security provided by the solution is not worth the performance sacrifice. The main drawback of retpoline is that it converts one variant of Spectre by "disabling" (using an infinite loop) the speculative execution of an indirect jump. The performance penalty depends strongly on the specific workload, in the range of negligible overhead [69] to 150% [45]. Furthermore, retpoline is not applicable on some architectures that use different mechanisms for the prediction of return instructions.

Compiler Hardening

Another family of software mitigations does not prevent speculation itself but instead neutralizes its effects by rewriting the data flow of programs. The key idea is to ensure that values obtained under speculation cannot influence microarchitectural state in a way that would leak secrets.

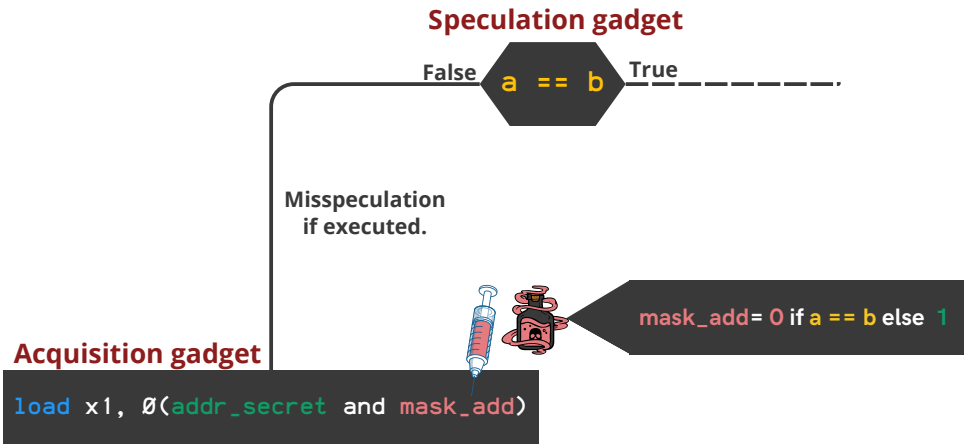


Figure 5.4 – Poison the address of the acquisition gadget (`load` instruction) according to the branch condition, ensuring it cannot be exploited during misspeculation.

For instance, **Speculative Load Hardening (SLH)** mitigation is an LLVM compiler pass that protects loads by *poisoning* their result if the outcome of the controlling branch is false, so that misspeculated values are not exploitable by the attacker. As the *poison* is based on a conditional branch, SLH mitigation is efficient in protecting against Spectre-PHT. In the illustration Figure 5.4, the `mask_add` is a predicate used to poison or not the target address of the *acquisition gadget* based on the condition of the branch condition. This adds an indirect dependency between **the condition** in the *speculation gadget* and the *acquisition gadget* that may leak the data. We implemented the SLH mitigation due to its relatively straightforward integration, allowing us to assess it in our test environment and compare its effectiveness with our own results in the chapter chapter 6. SLH is well documented in LLVM and implemented for x86 [19].

Similarly, *Blade* [74] is a software mitigation that analyzes data flows and breaks paths through which secret data loaded speculatively (*acquisition gadget*) could reach observable outputs (*disclosure gadget*). Different variants of Blade exist depending on the target architecture and available tooling, and implementations can use fence-based or SLH-like masking strategies. Static-analysis approaches such as *oo7* [75] further refine this

idea by identifying only vulnerable load-use pairs and inserting hardening instructions selectively. These methods provide better precision, but their effectiveness depends on accurate program analysis and may vary across microarchitectures.

Both *Blade* and *oo7* are also classified as selective speculation as they both detect instructions vulnerable to Spectre and block their speculative execution using different techniques such as `fence` instructions.

5.2.2 Hardware Mitigation

Hardware mitigations offer more advantages as they can directly modify the microarchitecture and thus provide finer control in addressing Spectre attacks. In theory, they can deliver stronger security with lower performance overhead [31] compared to software-based solutions.

Hardware mitigations are more numerous, as they have greater access to the processor's internal mechanisms and can therefore address Spectre vulnerabilities through a wider variety of approaches. To simplify the discussion, all **selective speculation** techniques are grouped together in a single section, rather than being detailed separately as in the case of software mitigations.

Selective speculation principle

With an understanding of how Spectre operates, a logical strategy is to identify the occurrence of a Spectre gadget and suspend the speculative execution of instructions that may contribute to a Spectre success. Numerous proposals have followed this principle, differing in how they identify potentially unsafe instructions, the granularity at which speculation is restricted, and the mechanisms used to enforce safety. The key idea is to recognize risky behavior corresponding to the three elements of a Spectre gadget: the speculation gadget, the acquisition gadget, and the disclosure gadget. Mitigations may intervene at different stages, either by preventing speculation altogether, forbidding speculative `load` instructions, or blocking the covert channel used for disclosure.

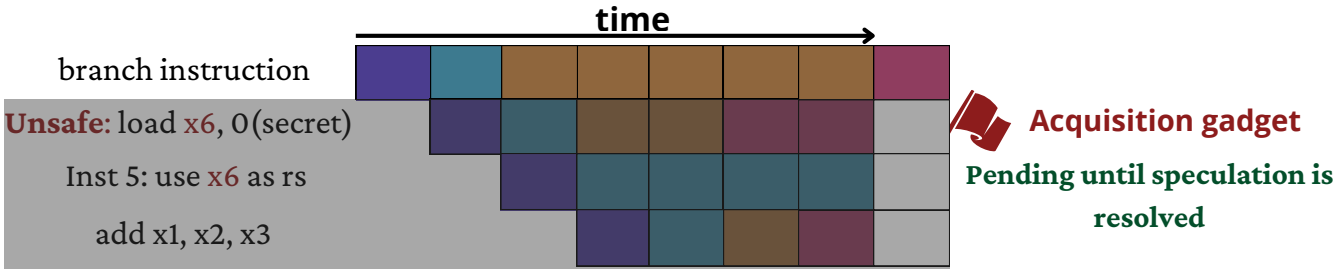


Figure 5.5 – Tag any acquisition gadget (`load` executed speculatively) and block subsequent instructions from using its destination register (which contains the speculatively loaded data).

Most mitigations focus on tracking a potential acquisition gadget and preventing the execution of subsequent disclosure gadgets. This can be achieved by marking any speculative `load` as **unsafe**, yet still allowing its execution. However, all instructions that depend on the loaded value are delayed until speculation is resolved. This prevents potential leaks from any speculative load while still permitting speculation on unrelated instructions. Multiple mitigation proposals adopt this approach, such as SpecTerminator [32], CondSpec [39], STT [87], SpecShield [7], NDA [78], and DOLMA [41]. The key difference lies in their level of permissiveness, creating a trade-off between performance and security. Some mitigations adopt a more conservative policy, such as delaying any acquisition gadget or treating all speculative instructions as unsafe, thus maximizing security at the expense of performance. Others prefer a more permissive policy, stalling only those instructions that depend on the acquisition gadget and might influence either execution timing or microarchitectural state (potential disclosure gadget), thereby improving performance but reducing adaptability since the classification of disclosure gadget is tied to the specific mechanisms of the target microarchitecture.

Hybrid Hardware-Software Approaches for Enhanced Precision

The main drawback of the previous approach is that the hardware executes programs indiscriminately and can only base its protection on the current state of the pipeline. This increases the number of false positives since protections are applied globally throughout the program. To address this lack of precision, some mitigations combine hardware mechanisms with a software component to achieve better performance.

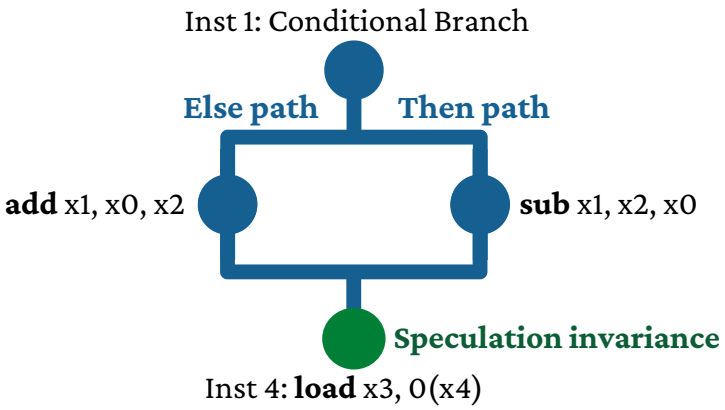


Figure 5.6 – Mark any `load` instruction that does not depend on speculation, either through its operands, as **safe**, and allow all safe instructions to execute speculatively.

InvarSpec [89], for example, introduces static analysis at the compiler level to identify instructions whose speculative execution cannot affect whether they are executed or the operands they depend on, a property called *speculation invariance*. Instructions proven to satisfy speculation invariance are classified as safe and may execute speculatively. In Figure 5.6, the execution of the `load` instruction, as well as the values of its operands, does not depend on the outcome of any speculation. In this case, it can be considered *speculation invariant* and may safely execute speculatively since it will always execute with the same operands regardless of speculation. InvarSpec acts as an optimization layer that can be combined with earlier mitigations such as STT, InvisiSpec, or NDA to reduce performance overhead. However, the arbitrary behavior of speculative execution, particularly in the presence of indirect jumps, poses a significant limitation. In such cases, the analysis often fails to prove security even when the instructions are in fact harmless, forcing InvarSpec to adopt a pessimistic stance. As a result, potential performance gains are diminished.

Instead of relying on compiler analysis, other proposals push the hybrid hardware-software approach further by requiring programs to explicitly annotate sensitive data. ProSpeCT [17], ConTExT [61], and SpectreGuard [23] follow this principle. The annotations guide the hardware in applying speculation restrictions only where needed, significantly reducing performance overhead.

Delay-on-Miss Mitigations

Instead of stalling the *disclosure gadget* itself, some approaches delay only the microarchitectural state changes it may trigger. Most solutions in this category concentrate on

the most common covert channel: the cache. In particular, they focus on `load` operations, which can modify cache-line entries. A cache line is replaced only if the requested data is not already present in the cache. During a Spectre attack, the secret-dependent data accessed speculatively by the *disclosure gadget* must not be cached beforehand in order to create a detectable state change.

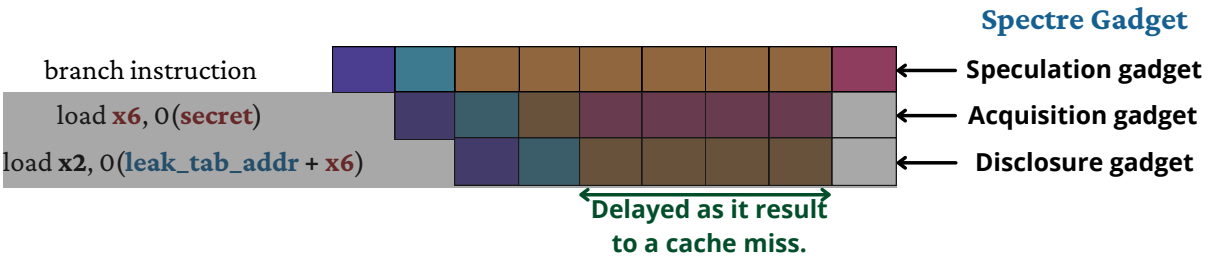


Figure 5.7 – Delay all `load` instructions that result in a cache miss, protecting against Spectre attacks that use the cache as a covert channel.

Based on this observation, certain mitigations delay the `load` operation of a *disclosure gadget* only if it would result in a cache miss, since this is the condition that causes a state change dependent on secret data. Similarly, the execution of the *acquisition gadget* may be delayed if the secret has not yet been cached.

This approach reduces performance overhead but is effective only when the secret resides outside the cache or when the *disclosure gadget* exploits cache-based covert channels. Delay-on-miss techniques have been proposed in mitigations such as *DOLMA* [41], *Understanding Selective Delay* [59], and *Efficient Invisible Speculation Execution* [58].

Clean Reversal of Misspeculated Microarchitectural State: Redo-Based Approach

Spectre attacks are possible because transient instructions persistently modify the microarchitectural state, even when they are eventually squashed due to misprediction. Therefore, a valid solution would be to revert the microarchitectural state exactly as it was before the misspeculated execution. While the concept is straightforward, implementing it is challenging since all the possible states must be reverted. That includes caches, branch predictors, Load-Store Queues, etc.

Several works have explored this line of defense, such as InvisiSpec [84], which prevents speculative loads from modifying the cache hierarchy. A small buffer, named *speculative buffer*, is placed between the core and the cache. Speculative loaded data is stored tem-

porarily in this buffer until the speculation is confirmed to be correct, e.g., once the *speculation gadget* is resolved. Speculative data can only be stored in the cache when the speculation results in a correct prediction. This mechanism prevents a *Disclosure gadget* from speculatively fetching cache lines and leaking data. However, InvisiSpec mitigation introduces noticeable latency penalties on memory operations.

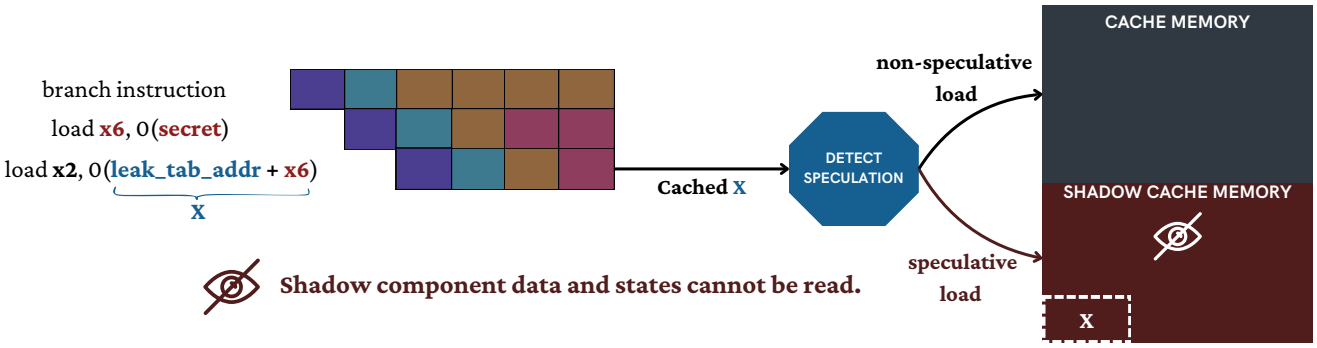


Figure 5.8 – All speculative data are placed in the shadow cache, making them invisible from outside, even from the processor itself

Instead of delaying the visibility of the state changes, CleanupSpec [57] and MuonTrap [3] propose rolling back the cache contents to remove the effect of misspeculative `load` instructions. These mitigations introduce a small cache structure, called *speculative filter cache* or *shadow cache*, placed at the lowest level of the cache hierarchy. Only this cache holds speculative data, allowing the microarchitecture to precisely isolate and control speculative state in the memory system. Data inside the *shadow cache* is not accessible to the processor, protecting speculative values from being exposed through covert channels.

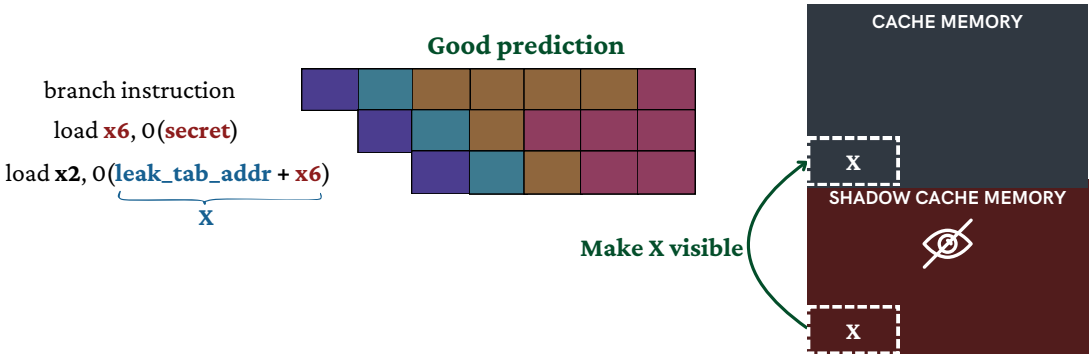


Figure 5.9 – Speculatively loaded data remain in the shadow cache and are transferred to the visible cache only if the speculation proves correct, at which point they become accessible to the rest of the processor.

The data is transferred to the main cache hierarchy and becomes visible only if it is deemed safe to commit, i.e., when the speculation path of the responsible load resolves to a correct prediction. However, this mechanism requires costly rollback logic and incurs additional storage overhead to track speculative modifications.

SafeSpec [33] mitigation extends this idea by introducing shadow structures that can be applied to each component in the microarchitecture to hold speculative execution. Unlike earlier proposals, which focus specifically on caches, SafeSpec generalizes its concept to the entire microarchitecture. Similar to MuonTrap and CleanupSpec, these shadow components isolate speculative updates until speculation is resolved and the state is deemed safe to commit.

The main drawback of SafeSpec mitigation is that the more components are equipped with shadow structures, the more complex the microarchitecture becomes, as it must synchronize each shadow component with its visible counterpart, which can significantly increase performance overhead. SafeSpec not only duplicates the complexity of the control logic but also inflates the overall hardware footprint of the microarchitecture.

Isolation Across Execution Contexts

Intel has proposed several solutions to strengthen isolation across privilege levels and mitigate Spectre-BTB, which exploits destination branch speculation on indirect jumps. Each BTB entry is tagged with the privilege level that created or last updated it. Indirect Branch Restricted Speculation (IBRS) and its improved version Enhanced IBRS (eIBRS) use this information to stop lower-privileged code from influencing the branch predictor state used by higher-privileged code. The OS enables IBRS when switching to privileged execution. While IBRS is active, the hardware does not consult predictor entries attributed to less-privileged contexts, ensuring that user-mode poisoning cannot steer privileged indirect-branch predictions.

However, running IBRS continuously (in *always-on* mode) introduces a non-negligible performance overhead [5], [20], so most operating systems enable it selectively, activating it only when necessary, for instance during kernel execution or sensitive context switches. Nonetheless, this protection alone is incomplete, as it does not prevent predictor influence from code executing at the same privilege level.

To complement IBRS, Intel introduced Indirect Branch Predictor Barrier (IBPB), a fence-like instruction that flushes the branch predictor state before allowing subsequent instructions to execute. This mechanism is recommended during context switches (user-

to-kernel or program-to-program transitions) to eliminate residual predictor state from the previous context. Placing an IBPB instruction immediately before executing a new program removes the residual state in the speculation mechanism of the previous one, thereby preventing branch target poisoning even between programs running at the same privilege level.

These mitigations specifically address Spectre-BTB and do not protect against other Spectre variants.

Dynamically Allocated Way Guard (DAWG) [34] and **Cache Allocation Technology (CAT)** [16] are cache memory protection mechanisms designed to defend against covert and side-channel attacks. They introduce a partitioning scheme that divides the last-level cache into independent regions associated with different protection domains (kernel/user), effectively breaking the communication channel between a *sender* and a *receiver*. DAWG extends this idea by enforcing isolation not only for cache allocation, but also for cache hits, misses, and metadata. For example, a cache miss in one program cannot influence the cache behavior of another program. This enhancement allows DAWG to protect the cache even during speculative execution.

5.2.3 Limitation of Spectre Mitigations

While the mitigation proposals for Spectre are numerous and technically interesting, they suffer from three fundamental limitations:

Rapid Adaptation and Diversification of Spectre Attacks: Spectre is a class of techniques that exploit speculative behaviors across multiple microarchitectural structures. The adaptation and diversification of Spectre across different prediction units, buffers, and execution resources make it challenging to confirm the reliability of existing mitigations. As a result, mitigations tailored to one mechanism are often bypassed by slightly different exploitation strategies. This is not just theoretical: [56], [64], [79], [81] show that common Intel mitigations (IBRS/eIBRS and IBPB), which aim to protect against *Spectre-BTB*, can still be bypassed.

Likewise, the *LFENCE/JMP* countermeasure was soon shown to be insufficient: speculative execution can still be triggered at the start of the pipeline even when registers are unavailable, leaving a residual exploitable window. This remaining window was demonstrated to be exploitable in [46]. Similarly, the control-flow modification introduced by *retpoline* to mitigate Spectre-BTB was later shown to introduce a new gadget exploitable

by RSB-like variants (the RETBLEED attack) [80]. In short, Spectre's adaptability forces mitigations to continually evolve and be re-evaluated.

Reproducibility and Evaluation Challenges: Comparing proposed mitigations is complicated by differences in microarchitectures, threat models, and benchmarks. Performance results reported by authors can vary widely depending on workload, processor model, and evaluation methodology. This produces contradictions in the literature and makes fair comparisons between mitigations difficult.

For instance, InvisiSpec reports a +72% performance overhead, while Efficient InvisiSpec [58] measured a 46% overhead for an improved design, and NDA reports +32% under different conditions.

Recent surveys [53], [54] provide detailed evaluations and expose many of these contradictions regarding security claims and measured performance. Our results, discussed in subsection 6.5.2, reach the same conclusion.

Tight Coupling to Current Microarchitectural Implementations: Most proposed mitigations are designed to protect particular speculation mechanisms, or prevent leakage channels in specific components (e.g. cache memory). This leads to a patchwork of protections (fences, predictor partitioning, buffer flushes, compiler rewrites) that must be combined for broader coverage. That combination is complex to deploy and can have large performance penalties. Moreover, as processor designs evolve, new speculative paths or microarchitectural optimizations can create previously unseen leakage channels, rendering prior defenses incomplete.

In practice, this means there is no single, simple, architecture-agnostic mitigation: solutions either accept residual risk or impose substantial runtime cost.

Outlook and Implications: These three issues explain why Spectre remains an open problem despite intense research: proposed mitigations can be bypassed in practice, evaluation results are often inconsistent, and defenses must continually adapt to shifting hardware designs. The community therefore increasingly favors hardware mitigations because they generally provide more stable, higher-performance mitigations with greater precision to enhance security.

In both software- and hardware-based mitigations, the selective speculation approach is widely adopted as it reduces performance overhead by restricting speculative execution

to security-sensitive instructions while allowing normal speculation for non-threatening instructions. However, the effectiveness of this approach heavily depends on the precision of the mechanism, creating a trade-off between security and performance. The earlier the intervention within a Spectre gadget, the more secure the solution, but also the more costly in terms of performance.

For example, stalling all the execution immediately after the speculation gadget eliminates all speculative execution, providing full protection against Spectre but significantly degrading performance and rendering much of the processor's speculation mechanism unnecessary. Conversely, stalling only the disclosure gadget requires detailed knowledge of the microarchitecture to identify which instructions can modify microarchitectural state and carries the risk of leaving some potential Spectre vulnerabilities undetected.

Nonetheless, a universally accepted, low-overhead, and provably complete mitigation does not yet exist.

Chapter 6

Evaluation of selective speculation

Despite the wide use of the selective speculation approach, uncertainty remains about the real effectiveness of this strategy, as studies often report large discrepancies between mitigations that follow the same principle. Such differences can be traced back to several factors, including the choice of benchmarks, the way the mechanism is implemented, its semantics, and the policies guiding its use.

The first contribution of this work is a thorough and comprehensive evaluation of selective speculation within an environment framework that allows these factors to be varied independently. To achieve this, we rely on a set of fence instructions that enable fine-grained control over speculation.

These instructions provide flexibility for compilers to enforce different placement policies and are backed by a dedicated hardware implementation, ensuring that the processor can be precisely guided from higher software layers.

Within this environment, we compare multiple barrier semantics on a common benchmark suite while also exploring different hardware implementations and insertion strategies. This setup makes it possible to conduct a more thorough examination of selective speculation as a defense approach.

The overarching question we address is:

Can selective speculation, when applied with dedicated fence instructions, serve as a practical and effective mitigation against Spectre attacks?

This contribution covers:

— The design of various fence instructions for selective speculation and their integration into the open-source out-of-order core NaxRISCV [67].

— The extension of an LLVM-based toolchain with multiple policies for inserting these fences automatically.

— The definition of a quantitative security metric, based on execution traces, that counts the number of vulnerable instruction sequences to enable a fair comparison of Spectre mitigations.

6.1 Fence instructions semantics

An **instruction-based serialization** barrier is an instruction that ensures that all previous instructions in program order are fully completed (including their effects on microarchitectural state) before any subsequent instruction can begin execution.
A **register-based serialization** barrier is an instruction that adds artificial register dependencies to order instruction execution.
A **speculation fence** is an instruction that refuses to execute speculatively and delays its execution until it is guaranteed to commit.

In this evaluation, we only consider register-based serialization instructions. Another possibility is the use of an instruction-based serialization fence, where ordering constraints are derived purely from program order. It is worth noting that in modern x86 cores, the **`LFENCE`** instruction, discussed in section 5.2.1, is an instruction-based serialization fence, in addition to acting as a speculation fence.

Register-based ordering provides the microarchitecture with finer-grained control over instruction dependencies. Instead of blocking all subsequent instructions, it can restrict only those involving a specific register. This selective restriction enables greater flexibility in instruction reordering, which helps preserve performance.

This approach also benefits from existing dependency-tracking mechanisms in NaxRISCV, thereby limiting the amount of additional hardware required. Implementation details are presented in section 6.3.

Our study examines three distinct fence semantics: **speculation fences** (**`fence_spec`**), **serialization fences** (**`fence_ser`**), and **conditional speculation fences** (**`fence_cond`**). From an architectural perspective, these fences behave like **`nop`** (no-operation) instructions, whose only role is to restrict instruction reordering across their position. They perform no functional operations themselves and follow the generic syntax: **`inst`** `rd`, `rs1`.

For clarity, the first two fences (speculation fence and serialization fence) are presented together, as their semantics are closely related.

6.1.1 Serialization and Speculation fences

This section describes the part of the semantics shared by serialization fences and speculation fences.

A register is considered as **architecturally correct** when the value of the architectural register visible to the program matches the value stored in the current physical register associated with the architectural register. This condition is satisfied only when all instructions in the pipeline that write to the register have committed.

In this manuscript, a register or operand is considered **available** or **ready** when no pending instruction is waiting to write to it. This does not guarantee that the value is architecturally correct; it only indicates that all instructions writing to it have completed their execution.

The two operands are interpreted as follows:

— The source register `rs1` allows the fence instruction to selectively **depend** on a specific register. In other words, the fence instruction cannot execute until `rs1` is ready.

— The destination register `rd` allows the fence instruction to enforce a dependency with subsequent registers.

In some processors, such as x86, a `mov rd, rs1` instruction copies the value from `rs1` into `rd`.

In NaxRISCV, the first register `x0` is immutable and always holds the value 0. This is common in some microarchitectures, as a fixed zero register is necessary to simplify certain operations. For example, the `RISC-V` architecture does not include a dedicated `mov` instruction; a move can be emulated with an `add` instruction that adds `rs1` to `x0` and writes the result into `rd`. This produces the same effect as a move operation.

Consequently, all dependencies involving `x0` are ignored, as this register never changes. To extend the semantics of serialization and speculation fences, the `x0` register is repurposed as a special operand: when used, it indicates that the dependency, normally applied to `rd` or `rs1`, should instead apply to all architectural registers.

This results in the following semantics:

1. **Predecessor Dependencies**: The fence instruction depends on the availability of register `rs1`. It cannot be executed until `rs1` is ready. If `rs1` = `x0`, then the fence is considered to depend on all architectural registers (`x1`–`x31`).
2. **Successor Dependencies**: Subsequent instructions that use `rd` must wait for the fence to execute. If `rd` = `x0`, the processor assumes that the fence instruction has established dependencies on all architectural registers (`x1`–`x31`), even though no values are actually modified.

In this manuscript, the term **Full Wait** refers to a fence instruction that depends on all registers (`rs1` = `x0`).
The term **Full Fence** refers to a fence instruction that forces all subsequent instructions to depend on it (`rd` = `x0`).

```
//1 case: fence cannot depend on beq
//as it does not have a destination register.
beq x1, x2, .target_address
fence.X x0, x0
lw      x2, 0(x3)

//2 case: reciprocal is false,
//fence.ser forces the beq to wait using the branch source register.
fence.X x0, x0
beq x1, x2, .target_address
lw      x2, 0(x3)
```

Figure 6.1 – Register-based fence relies on instruction operands to create dependencies. The symbol **`fence.X`** is used as a generic notation that refers to either **`fence.spec`** or **`fence.ser`**. In the first case, the fence cannot depend on the branch instruction (**`beq`**), even with **Full Wait** semantics, because the branch does not produce a destination register. However, the fence can still make the branch depend on it (second case) by using the branch's source register.

It is important to note that although serialization fences and speculation fences are encoded with a destination register operand, they do not alter the register's values. This

design ensures that the program state remains unchanged while still enforcing the required ordering constraints.

The two fence instructions reduce potential divergences between the architectural and microarchitectural control flows, but they do not eliminate them entirely. An instruction that has only source registers or only a destination register cannot be fully ordered with respect to others, since the dependency relation lacks sufficient operands.

The example in Figure 6.1 illustrates this limitation. In the first case, the `fence.X` (either speculative or serialization) cannot depend on a following `branch` instruction because a branch does not produce a destination register. The reciprocal is false; in the second case, the fence can delay any instruction that does have a destination register, as the dependency relation can then be established. In both cases, the speculative behavior of the branch is not stopped by the fence.

When a speculation gadget does not write to a destination register, a register-dependency fence cannot stop the spectre gadget. Ensuring that registers are available neither prevents the gadget from being executed speculatively nor stops an attacker from transiently manipulating its value, as illustrated by the instruction `lw x2, 0(x3)` in Figure 6.1. With the same example, even stronger fence semantics that guarantee the architectural correctness of the target address of the load (see subsection 6.3.4), ensuring that the load legitimately reads the correct address, do not eliminate the leak if the instruction can still execute on a misspeculated path.

The two fences diverge in their behavior.

Serialization Fence (`fence.ser`): The serialization fence is defined as `fence.ser rd, rs1`. It may execute speculatively, provided that all predecessor dependencies are resolved. As a result, this fence ensures ordering with respect to predecessor instructions but does not guarantee that successor-dependent instructions will avoid speculative execution.

Speculation Fence (`fence.spec`): The speculation fence extends the semantics of a serialization fence. Unlike `fence.ser`, it can **execute only in a non-speculative state**, i.e., when the processor is certain that the instruction will commit. It is defined as `fence.spec rd, rs1` with the following additional semantics:

1.

3. **Non-speculative Execution**: `fence.spec` terminates only once the processor has resolved speculation and guarantees that the instruction will commit (considering

exceptions, interrupts, etc.).

In practice, the speculation fence provides a stricter guarantee than serialization fence: it enforces that targeted subsequent registers are exclusively used in non-speculative mode, provided no speculation gadget is inserted between the fence and the target register. This reduces the risk of speculative side effects at the cost of higher performance overhead.

6.1.2 Conditional speculation fence

Spectre-PHT is the Spectre variant that exploits speculative execution following conditional branches.

The last fence implemented in NaxRISCV is the *conditional speculation fence*, defined as `fence.cond rd, rs1`. This instruction is specifically designed to mitigate Spectre-PHT attacks. Conditional speculation fences are inspired by the SLH mitigation but introduce specific semantic differences, which are detailed in section 6.5.

Unlike serialization or speculation fences, the conditional speculation fence requires a *predicate* stored in `rs1`, which indicates whether the outcome of a conditional branch could have been mispredicted. The predicate computation is illustrated in Figure 6.2. Therefore, an additional computation step is necessary beforehand to derive this predicate from the branch condition.

The conditional speculation fence enforces two main behaviors:

1. **Terminate Speculative Window:** If the predicate evaluates to a nonzero value, the fence instruction stalls execution until the processor reaches a non-speculative state, effectively closing the speculative window.

2. **Successor Dependencies:** Subsequent instructions that consume `rd` must wait for this fence to execute. If `rd` = `x0`, the processor assumes that the fence depends on **and** affects all architectural registers (`x1`–`x31`), even though no register values are actually modified.

```
        # if (x1 < x2) -> Jump to L_EQUAL
        blt     x1, x2, .L_EQUAL
        # Predicate computation for misspeculation detection
        #x3 = (x1 < x2) ? 1 : 0;
        sltu    x3, x1, x2        #if x3 = 1 then the prediction was wrong
        # x2 = to create a dependency with the address of the load
        # x3 = predicate
        fence.cond x2, x3         #if non-zero then stall
        lb      x4, 0(x2)         # acquisition gadget (guarded by fence.cond)

    .L_EQUAL:
```

Figure 6.2 – Predicate computation: the branch condition is used to compute a predicate that equals zero when the condition is true. This allows the **`fence.cond`** to decide whether execution should stall, preventing execution of the **`lb`** in the case of misprediction.

In Figure 6.2, the branch instruction **`blt x1, x2, .L_EQUAL`** may be mispredicted. The predicate register `x3` is computed with **`sltu`**, which sets the value of `x3` to `1` if the condition `(x1 < x2)` is true, indicating that the branch is mispredicted since it should have jumped to `.L_EQUAL`, and `0` otherwise.

If `x3` is not equal to 0, the **`fence.cond`** blocks the execution until the processor returns to a non-speculative state. This ensures that the *acquisition gadget* (**`lb`**) cannot leave any observable trace in the microarchitectural state when the branch has been mispredicted.

6.2 Security policies

As mentioned earlier in section 5.2.3, the main weakness of selective speculation lies in its strong reliance on the decision of *when* and *which* instructions should be stalled in order to block Spectre.

A Spectre attack consists of several operations executed in sequence, meaning that interrupting any one of them is sufficient to break the attack chain. However, the microarchitecture itself has no knowledge of which data are confidential. As a result, any defense must conservatively treat *all* potential Spectre gadgets as threats, even when they operate on seemingly non-sensitive data.

Delaying the *speculation gadget* (the earliest stage) only reduces the size of the speculation window. The availability of registers does not imply that execution is non-speculative: speculation begins as soon as the speculation gadget is detected, typically during the fetch stage, and at the latest during decode. For instance, a branch may have all its operands ready, but its condition has not yet reached the execution stage. Speculative execution on this branch can still occur in that short window (Figure 6.1). Thus, stalling at this early point can suppress many speculative paths but at the cost of significantly reducing useful speculation, harming performance, and, as shown previously, even the remaining small window can still be exploited for Spectre.

An alternative strategy is to allow acquisition and disclosure instructions to execute speculatively while preventing them from leaving persistent traces in the microarchitectural state (the so-called "redo" or clean-reversal approaches, described in section 5.2.2). This approach preserves more performance but comes with the major drawback of requiring protection of *every* microarchitectural structure that could serve as a covert channel. Since caches, predictors, buffers, and other resources may all leak information, such a solution becomes heavily tied to the specific microarchitecture and requires continual reassessment as new channels are discovered.

The remaining option is to delay either the *acquisition gadget* (the speculative load) or the *disclosure gadget* (the instruction that converts the loaded value into a persistent microarchitectural change).

Stalling speculative loads prevents secret values from being read speculatively but also delays all dependent instructions, thereby severely limiting the benefits of speculation.

Precision can be improved by stalling only the disclosure gadget. In this case, most non-leaking speculative instructions are allowed to proceed, which preserves more performance. However, this approach requires precise knowledge of which instructions can modify observable microarchitectural state. Any missed cases leave residual vulnerabilities, and the effectiveness of the defense becomes dependent on the specific processor. Nevertheless, because it operates at the architectural level (instruction set) rather than on the microarchitectural level, this strategy is less sensitive to microarchitectural evolution, though it still incurs a strong dependency on the underlying ISA design.

A more permissive variant is to allow the *acquisition gadget* to execute while preventing its result register from being used speculatively. This reduces the performance cost but still delays many benign instructions that are not directly involved in the attack.

Based on this knowledge, we implemented several insertion policies targeting different

operations within Spectre gadgets. The main weakness of Spectre lies in the *acquisition gadget*. While *speculation* and *disclosure* gadgets can be realized through many types of instructions that vary across microarchitectures, there is typically only one instruction that consistently serves as the acquisition gadget: the **`load`** instruction, through its variants (**`lw`**, **`lh`**, **`lb`**, etc.), which is capable of bringing secret data into a register.

This raises important design questions: should fences be placed before or after the target instruction? Could fences also be inserted around the end of branch blocks, function calls, or indirect jumps to specifically target certain Spectre variants?

These considerations give rise to numerous insertion policies, defined by combinations of placement options and the fence semantics introduced earlier. An overview of all policies is provided in Table 6.2.

Fence insertion is implemented in the LLVM compiler, which provides fine-grained control over the IR. To this end, we reproduced the SESES approach that inserts an **`LFENCE`** instruction before each load, already available in LLVM [18]. We then replaced **`LFENCE`** with our new instructions, leveraging their similar semantics. The naming of each policy reflects the fence type, the operand configuration that defines its behavior, and its placement relative to the target instruction.

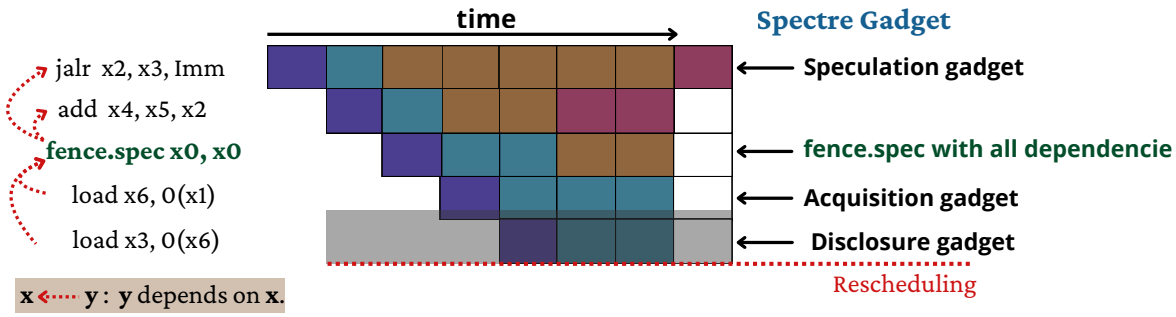


Figure 6.3 – `specall-before_load` inserts a **`fence.spec`** `x0, x0` before every load instruction, ensuring that all preceding registers are ready and that execution is no longer speculative before the load is issued.

For instance, Figure 6.3 shows an example of how `specall-before_load` is used:

— `specall-`: the speculative fence **`fence.spec`** is used. The postfix `-all` expresses a full register dependency: both `rs1` and `rd` are set to `x0`, The corresponding semantics are described in item 1 and item 2.

— `before_load`: the instruction is inserted immediately before every load in the program.

The prefixes `ser` and `cond` correspond to the use of **`fence.ser`** and **`fence.cond`**, respectively. We then modified this pass to handle the cases where `rd` and `rs1` are not equal to `x0`.

We implemented `spec-after_load` to compare placing fences after the target instructions with placing them before.

Additionally, the `dependency` approach adds fences with operands based on branch conditions and a load following it, such as **`fence.spec rd, rs1`** (`spec-dep_load`) or **`fence.ser rd, rs1`** (`ser-dep_load`). Basic optimizations prevent redundant protection of non-redefined registers within the same basic block.

SLH mitigations (introduced in section 5.2.1) adapt well to `RISC-V` and provide a useful performance baseline. On `x86`, the SLH predicate is typically updated with conditional-move instructions (**`cmov`**). Because base `RISC-V` lacks **`cmov`**, the same effect is emulated using simple integer comparisons and bit-masking (for example, with **`slt`** to produce a 0/1 flag, then turning that flag into a full-word mask and using it to select the guarded value). This emulation preserves the SLH semantics while remaining cheap in hardware.

Inter-procedural means *across functions or procedures.* In compiler or program analysis, an inter-procedural technique allows information (such as variables, states, or security predicates) to be passed from one function to another, so that the effect is preserved across function calls instead of being limited to a single function's body.

Our work builds on an earlier SLH translation for `RISC-V` by Moein Ghaniyoun [25]; we extended it to produce two policies, `slh` and `slh-ip`. The `-ip` suffix stands for inter-procedural, which means the SLH predicate survives function calls. It assures that even if a function is called during the transient execution, the predicate is still available in the function to poison any **`load`** instructions.

The conditional speculation fence is designed to be an alternative to the SLH strategy: instead of poisoning the load's address, a **`fence.cond`** following every conditional branch uses the SLH predicate (if a load exists) to prevent speculative execution in case of misspeculation. This can also be extended inter-procedurally, forming the `spec_cond_ip` policy. Unlike SLH, this technique halts speculative execution entirely during misspeculation, ensuring no speculative traces are left on the microarchitecture.

6.3 Fence Instruction Implementation

The implementation of the proposed fences requires modifications to several microarchitectural components. The changes needed for the speculative fence, serialization fence, and conditional speculation fence are largely similar, with some modifications shared across all three.

To improve clarity, the explanation of the implementation follows the order of operations that an instruction undergoes during pipeline execution. Additionally, icons are used to indicate which modifications are needed for each fence:

— SR marks modifications required for **`fence.ser`**.

— SP marks modifications required for **`fence.spec`**.

— CF marks modifications required for **`fence.cond`**.

6.3.1 Decode-Stage Modifications SR SP CF

Different modifications are applied during the three phases of the Decode stage, as described in section 3.2.

Decode Phase:

Each fence instruction is assigned a unique combination of `opcode`, `funct7`, and `funct3` to ensure correct identification:

Speculative fence - `fence.spec`:

funct7	rs2	rs1	funct3	rd	opcode
0000000	···	···	011	···	0001111

(a) Fence.spec instruction encoding.

Serialization fence - `fence.ser`:

funct7	rs2	rs1	funct3	rd	opcode
0000000	···	···	111	···	0001111

(b) Fence.ser instruction encoding.

Conditionnal speculative fence - `fence.cond`

funct7	rs2	rs1	funct3	rd	opcode
0000000	···	···	100	···	0001111

(c) Fence.cond instruction encoding.

Figure 6.4 – Encoding of each fence instruction added to the `RISC-V` ISA

These encodings are added to the list of authorized instructions in NaxRISCV (`Rvi.scala`).

Since this phase reads the instruction code received from the Fetch stage, it can precisely determine the instruction being decoded.

In addition to the general instruction properties defined in subsection 3.2.1, new attributes are introduced in the decode stage to represent fence-specific information:

— `IsFence`: a boolean flag indicating whether the instruction is one of the three fence types.

— `FenceType`: specifies the type of fence (speculative, serialization, or conditional).

— `IsFullWait`: a boolean flag set when the source register of a fence instruction is `x0`. This field is always set to zero for conditional speculation fence, since they do not support the semantics *Full Wait*.

— `IsFullFence`: a boolean flag set when the destination register of a fence instruction is `x0`.

During the renaming phase (subsection 3.2.2), each new architectural destination register is mapped to an available physical register, while the previous physical register is unlinked from the architectural register.

Since all three fence instructions define a destination register, the microarchitecture must continue tracking the old physical register and copy its value into the new physical register. This ensures that the architectural value of the destination register from the program perspective remains unchanged.

However, only the EU has access to the register file. To address this, during the renaming stage, the second source register `rs2`, which is unused from the program's perspective, is overwritten with the destination architectural register. This allows the execution to retrieve the value from the old physical register associated with the destination architectural register and copy it into its newly assigned physical register. The entire operation is described in subsection 6.3.5. This mechanism allows the microarchitecture to preserve the correct architectural state of the register.

Dispatch phase:

This stage undergoes the most significant modifications to support fence instructions since dependency tracking occurs here.

During instruction dependency resolution, the `Dispatch plugin` stage may need to inject additional dependencies into the fence's `trigger` field in the issue queue (see subsection 3.2.3), depending on the instruction's operands. If the fence is neither a *Full Wait* nor a *Full Fence*, no extra work is required: the default dependency logic already enforces correct ordering. In contrast, *Full Wait* and *Full Fence* instructions require mask-driven updates to the `trigger` field in order to introduce additional dependencies.

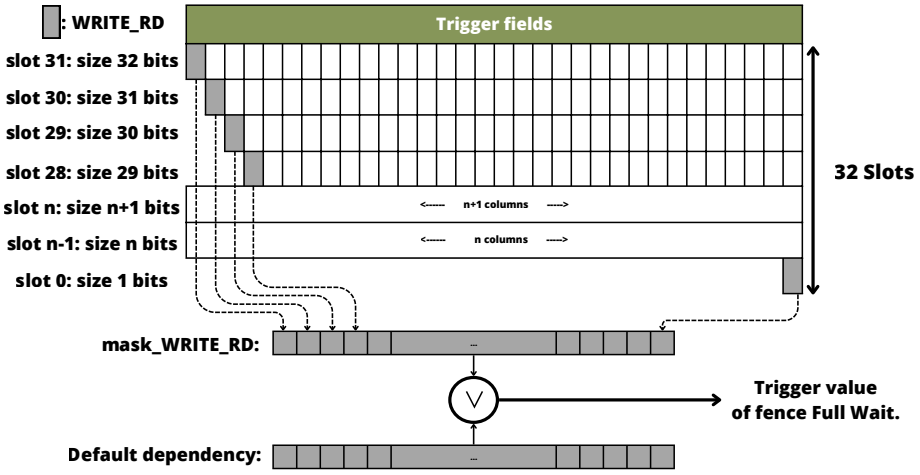


Figure 6.5 – The MSB of each trigger entry in the issue queue is used to create a bit-mask. When a *Full Wait* fence is received, its trigger value is OR-ed with the bit-mask to make it depend on all pending instructions that have a destination register.

A *Full Wait* must depend on every pending instruction in the issue queue that produces a result, any instruction with a destination register. Each issue-queue entry stores a copy of its *inst.WRITE_RD* flag in the MSB of its `trigger` field. The dispatch unit can therefore construct a bitmask that marks all entries with *inst.WRITE_RD* set. When a *Full Wait* is dispatched, this mask is OR-ed into the fence's `trigger` field, ensuring that the fence instruction waits for all currently pending writers in the queue. This operation is illustrated in Figure 6.5.

Because NaxRISCV fetches two instructions per cycle, a special case must be handled: if the younger instruction of the pair is a *Full Wait* and the older instruction writes a register, the dispatcher inserts an explicit dependency from the younger instruction to the older fence. This ensures that the intended ordering is preserved.

A *Full Fence* is managed using a similar mask-based mechanism. The issue queue is extended with a new field, `IsFullFence`, which stores a copy of the flag whenever an instruction is a Full Fence. From this field, the dispatch unit constructs a bitmask marking the positions of all *Full Fence* entries currently in the queue. Each newly dispatched instruction then has this mask OR-ed into its `trigger`, ensuring that it implicitly depends on every outstanding *Full Fence*.

In the dual-fetch scenario, if the older instruction is a *Full Fence*, the dispatcher also adds an explicit dependency from the younger instruction to that fence, guaranteeing correct instruction ordering.

6.3.2 Speculation State Detection SP

In NaxRISCV, there is no explicit "speculative" bit for each instruction. Instructions are issued and executed the same way whether they are on the architecturally correct path or on a predicted path. The only time the microarchitecture intervenes is when a *speculation gadget* is resolved and its predicted outcome disagrees with the actual result: this triggers a pipeline reschedule (rollback) to the correct path.

Because the Reorder Buffer (ROB) holds all in-flight instructions in program order and the commit pointer advances as instructions retire, we can define a speculation state: an instruction is **speculative** exactly when there exists an older speculation gadget that has not yet committed. In other words, any instruction that sits after an unresolved speculation gadget in the ROB is executing speculatively.
Let:

— $ROB\_ID(I)$ be the ROB index (position) of instruction I.

— $\mathcal{G}_{\text{spec}}$ be the set of ROB indices corresponding to speculation gadgets currently in the ROB.

— between(a, b, c) denote that b is ordered after a and before c in the ROB order.

An instruction I is *speculative* if there exists a speculation gadget $gadget \in \mathcal{G}_{\text{spec}}$ such that g sits between the commit pointer and I in ROB order:

$$\text{Speculative}(I) \iff \exists\, g \in \mathcal{G}_{\text{spec}} : \text{between}(Commitpointer,\ g,\ ROB\_ID(I)).$$

The **speculation detector** takes the `RobId` of an instruction as input and returns a boolean value indicating if the instruction is speculative.

6.3.3 Stall mechanisms SP CF

The *speculation fence* and the *conditional speculation fence* both require stalling the pipeline under specific conditions: the speculation fence must stall when the processor is in speculative execution, while the conditional speculation fence must stall when in speculative execution **and** its predicate is non-zero.

This stall behavior can be implemented in two possible stages: the *dispatch stage* or the *execute stage*. However, the conditional speculation fence depends on the value of its source register to determine whether stalling is necessary. Since only the EUs can operate

on register values, the conditional speculation fence can only enforce its stall during the execute stage.

Dispatch-Stall fence: SP

This stall mechanism is only supported by the speculative fence. The `Dispatch plugin` unit applies additional rules to prevent any **`fence.spec`** instruction from being sent to the EU while the processor is in a speculative state, even if all dependencies are resolved and the EU is available.

> The **MSB** of the trigger field indicates whether the instruction writes to a destination register.
> The **`Sel`** field specifies the EU where the instruction should be executed, with `0` indicating that the entry is not valid.
> An instruction is considered **ready** when all bits of its `trigger` field, except the MSB, are zero, and the `Sel` field is nonzero.

For a speculative fence, readiness requires not only that these default conditions are satisfied but also that the instruction is confirmed to be in a non-speculative state, thanks to the speculation detector (subsection 6.3.2).

With this implementation, the speculative fence can be treated as a *static-latency* instruction, since the remaining work in the execution stage has a fixed latency. This design allows the speculative fence to benefit from optimizations provided by *static-latency* techniques: the `Dispatch plugin` stage anticipates the completion of the instruction, removing dependencies on pending instructions earlier.

Execute-Stall Fence: SP CF

This approach is supposed to minimize performance loss by delaying the stall until the execution stage. It is supported only by fences that need to stall, namely the speculative and conditional speculation fences:

— The *speculative fence* cannot exit the execution stage of the pipeline while the processor is in speculative mode, thanks to the speculation detector.

— The *conditional speculation fence* stalls in the execution stage if its predicate is nonzero. Only a rescheduling operation can unblock its execution, as the predicate is treated as a misspeculation.

However, this method has a significant drawback: under certain conditions, it can deadlock the pipeline.

```
S:   blt        x2, x3, .L_EQUAL
F:   fence.spec x1, x1

.L_EQUAL:
```

Figure 6.6 – Example of deadlock with two instructions.

Consider the example in Figure 6.6, where a *speculation gadget* `S` and a fence instruction `F` (either speculative or conditional) share the same EU *ALU0*. If `F` is speculatively issued before `S`:

— `F` stalls on *ALU0* until speculation is resolved, waiting for `S` to commit.

— `S` waits for *ALU0* to become available, which is occupied by `F`.

This creates a circular dependency and a deadlock. One solution is to execute any fence that can stall in a separate EU from the *speculation gadget*.

Still, even with a dedicated EU, a similar deadlock can arise.

```
F1:  fence.spec x2, x4
S:   blt        x2, x3, .L_EQUAL
F2:  fence.spec x1, x1

.L_EQUAL:
```

Figure 6.7 – Example of deadlock with three instructions.

In the example shown in Figure 6.7, a *speculation gadget* `S` is placed between two fences `F1` and `F2`, both capable of stalling. `F1` and `F2` share the same EU *FENCE0*, and `F1` has a dependency on `S`. If `F2` is speculatively executed before both `S` and `F1`:

— `F2` stalls on *FENCE0* until speculation resolves, waiting for `S` to commit.

— `S` cannot execute yet, as it depends on `F1`.

— `F1` waits for *FENCE0*, which is held by `F2`.

This cycle of dependencies results in a full pipeline deadlock. To prevent this, the `Dispatch plugin` stage enforces in-order execution of all fences that can stall by making them mutually dependent. While this avoids deadlock, it reduces pipeline flexibility and leads to performance loss.

Finally, since stalls occur in the execution stage, fence instructions exhibit indeterminate latency, forcing them to be treated as *dynamic-latency* operations.

6.3.4 Operand-Stall SP

An alternative stall mechanism was implemented for the speculative fence, diverging from the semantics originally defined in section 6.1, which stipulate that a speculative fence cannot be executed speculatively. Rather than validating the global non-speculative state, this mechanism prioritizes the architectural correctness of the source register. Put simply, a speculative fence cannot proceed until its source register `rs1` has been committed.

Ensuring architectural register correctness does not directly eliminate *Spectre*, as it does not prevent speculation and still allows sensitive data to be read and disclosed. Nevertheless, it reduces the set of exploitable Spectre gadgets, since forcing the convergence between the architecture and the microstructure on the *acquisition gadget* protects its operands from being manipulated by transient operations occurring before the fence.

During the *dispatch phase*, the `Dispatch plugin` consults the `RfDependency plugin` table (Figure 3.8), indexed by *inst.PHYS_RS1*. Each entry contains the *ROB_ID* of the last instruction that wrote to the physical register, along with a `valid` flag equal to `1` if the instruction has not yet retired. If the `valid` flag is set, the corresponding *ROB_ID* is copied into a new issue-queue field called *LAST_WRITE*.

By definition, this variant of the speculative fence requires that `rs1` be committed before the fence itself can execute. Thus, the dependency between the **`fence.spec`** and the last writer to `rs1` is resolved only at commit, not at completion as in the default behavior.

Concretely, a *wake signal* (section 3.3) is ignored by subsequent *fence.spec* instructions if the *ROB_ID* carried by the signal matches the fence's *LAST_WRITE*. This prevents the dependency from being cleared at instruction completion. Instead, the corresponding bit in the fence's `trigger` field is only cleared when the commit stage retires the instruction whose *ROB_ID* matches *LAST_WRITE*.

In effect, this implementation allows a `fence.spec` to execute speculatively as soon as its source register `rs1` has been committed.

6.3.5 Execute-Stage Modifications SR SP CF

Even if the EU does not write to the destination register during the execution of fence instructions, the *inst.ARCH_RD* still loses its old value because it is associated with a new physical register during the renaming operation.

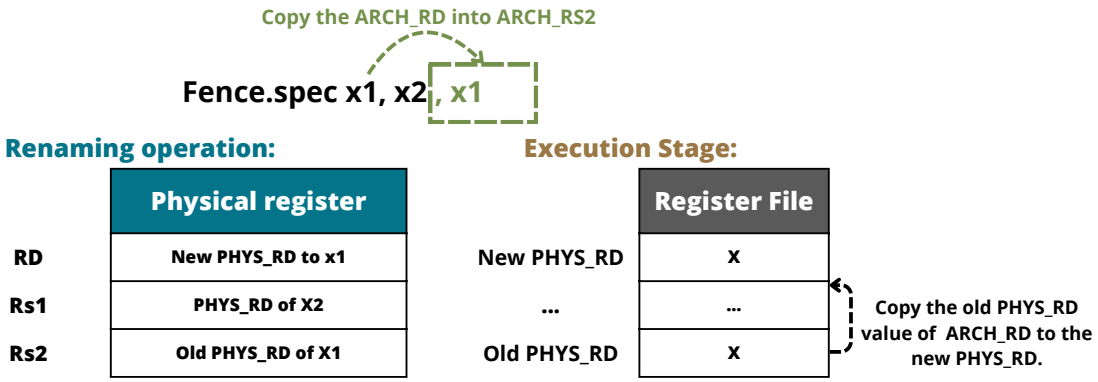


Figure 6.8 – During the Decode stage, *inst.Arch_Rs2* gets a copy of *inst.Arch_RD*. This allows the microarchitecture to track the old physical register of *inst.Arch_RD* during the renaming. Then, the value of the old physical register of *inst.Arch_RD* stored in *inst.Arch_Rs2* is copied to the new *inst.PHYS_RD* register.

To preserve the architectural value during execution, the microarchitecture uses the second source register as a backup. The EU reads the value from the register file at the index *inst.PHYS_RS2* and writes it back to the new location indexed by *inst.PHYS_RD*, as illustrated in Figure 6.8.

Some fences require dynamic latency, such as *conditional speculation fence* and certain implementations of *speculation fences* that stall in the execution stage. In these cases, the EU must send a *dynamic wake* to the `Dispatch plugin` to signal instruction completion. However, any wake signal is automatically ignored if the destination register of the executed instruction is `x0`, since instructions consuming `x0` theoretically have no need to check for updates (its value being immutable).

Therefore, the *dynamic wake* is made mandatory for any fence instruction with dynamic latency, ensuring that even a fence with the *Full Fence* option is properly recognized and not ignored.

6.4 Security metrics

Most of the previous work considers security as a binary value; the execution is either secure or unsecure with respect to a set of exploits (Spectre-PHT, Spectre-BTB, etc.) known at the time.

We propose a different strategy that is not tied to any specific exploit by identifying a Spectre gadget.

During experimentation, various insertion policies combined with fence instructions are applied to a program, which is then executed on a simulator of NaxRISCV. The methodology is detailed in section 6.5. The simulator produces a detailed trace file that provides a complete view of the pipeline execution, including misspeculated instructions. This enables the detection of Spectre gadgets by analyzing the different components that form the gadget.

To achieve this, taint tracking is applied to follow all potential secrets resulting from a speculative **`load`** (acquisition gadget) up to a speculative disclosing instruction (disclosing gadget). In this context, a *disclosed value* is defined as any source value used as the address of an issued **`load`**, the address or value of an issued **`store`**, or as an operand of an issued **`branch`**.

The **TLB** is another component involved in memory access operations. Since it is not central to the focus of this work and would require extensive background explanations, it is mentioned here only for completeness.

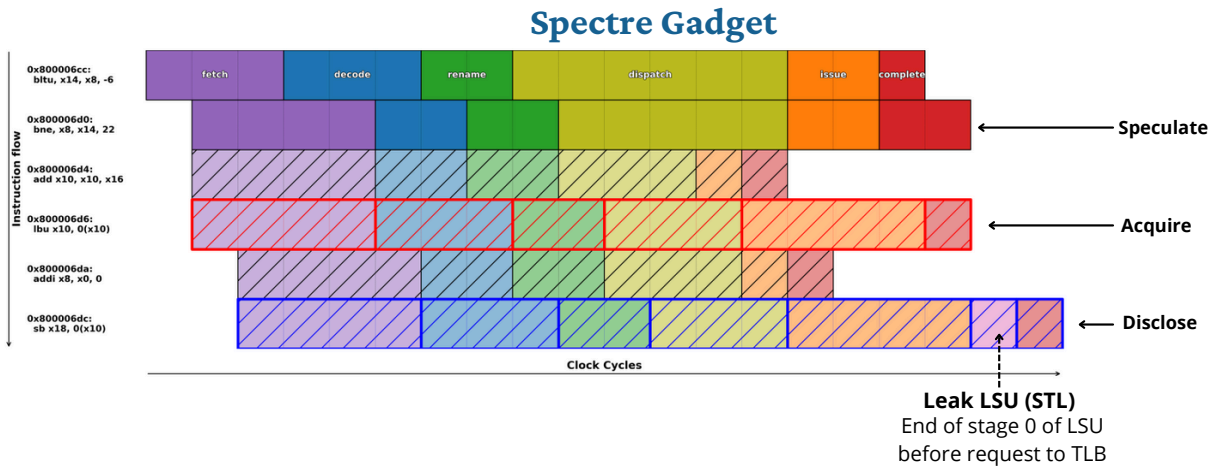


Figure 6.9 – Example of an interpreted trace of a random gadget from an Embench execution. The figure is automatically generated by our test environment (see section 6.5). It highlights the exact moment when the `sb` instruction, used as the disclosure gadget, leaks data into the TLB.

Figure 6.9 illustrates a real Spectre gadget detected by our test environment, described in section 6.5. It originates from a trace generated after executing random code, later interpreted by analysis tools within the test environment (see section 6.5). In this example, the LSU modifies the TLB state during the execution of the `sb` instruction, which acts as the disclosure gadget.

The SLH mitigation poisons the `load` address by zeroing its value during misspeculation. To accurately assess this approach, the cases where `load` instructions target the null address are detected and excluded from gadget consideration. However, any identified gadget is assumed to potentially lead to an exploit and must be mitigated.

Therefore, it is possible to count the number of Spectre gadgets in execution traces across all benchmarks and configurations.

A gadget is identified by the address of its acquisition instruction. If two gadgets in the execution trace share the same acquisition instruction, they are counted only once. This approach is necessary because all our benchmarks consist of loops performing repeated operations: the same gadget may appear multiple times in the trace for each iteration. Counting each occurrence would measure loop iterations rather than security impact.

6.5 Evaluation Environment

To evaluate the effectiveness of the proposed fence semantics and insertion policies, hardware implementing these semantics is required. Three different stalling mechanisms (execute-, dispatch-, and operand-stall) were incorporated into the NaxRISCV core, resulting in three distinct processor configurations.

A set of benchmark programs is also necessary to assess the potential security and performance implications of these methods. The **Embench** suite [12] was selected for this purpose, as it is open-source and covers a wide range of applications.

During evaluation, each program undergoes the following steps:

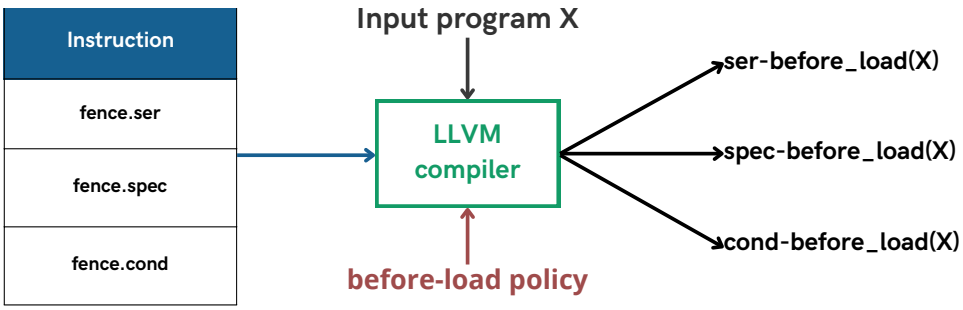


Figure 6.10 – Example of the `before_load` policy applied to program X. It generates one program for each fence instruction combined with the policy.

Instruction Insertion and compilation Different insertion policies were defined based on our understanding of how Spectre operates, as described in section 6.2. Each policy specifies where fence instructions should be inserted in a program. For each policy, all three fence types (`fence.spec`, `fence.ser`, and `fence.cond`) are applied separately, resulting in three distinct versions of the program for each fence-policy combination, as illustrated in Figure 6.10.

To measure performance costs, we implemented a `nop` policy inserting `nop`s (a pseudo-instruction for `addi x0, x0, 0`), which advances the program counter without architectural impact. It gives the impact of just adding instructions to the program, without the impact of the fence instructions' semantics. Despite its simplicity, it affects both security and performance, serving as a baseline.

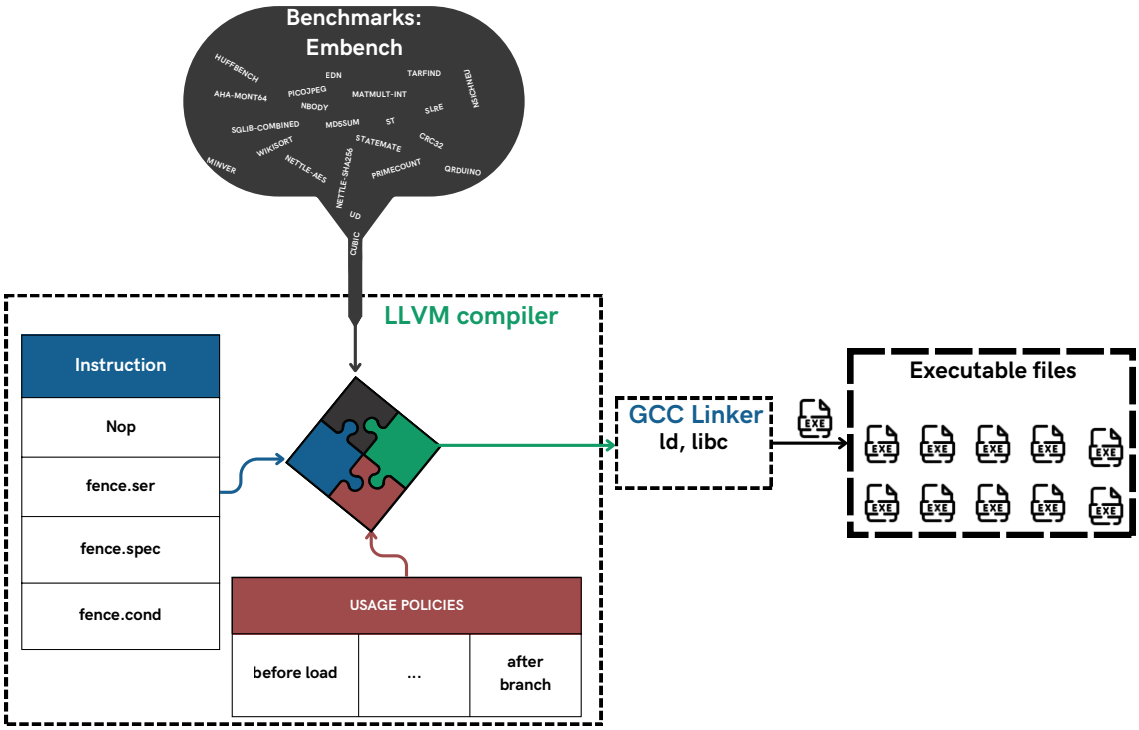


Figure 6.11 – The *LLVM compiler* inserts each instruction according to different insertion policies into the Embench programs. It then generates executable files ready for simulation.

Figure 6.11 illustrates the compiler side during the insertion of fences and `NOP` instructions

Simulation: The program resulting from each fence-policy combination is sent to a simulator that models the three different processor configurations: execute-, dispatch-, and operand-stall.

A `.gem5o3` file is an execution trace generated by the Verilator-based simulator. It contains a detailed record of the instruction flow through the processor pipeline, including fetch, issue, execution, and commit stages.

The simulations were carried out using the Verilator [66] simulator, and execution traces were collected in a format `.gem5o3`, which provides a human-readable timeline of instructions as they move through the out-of-order pipeline.

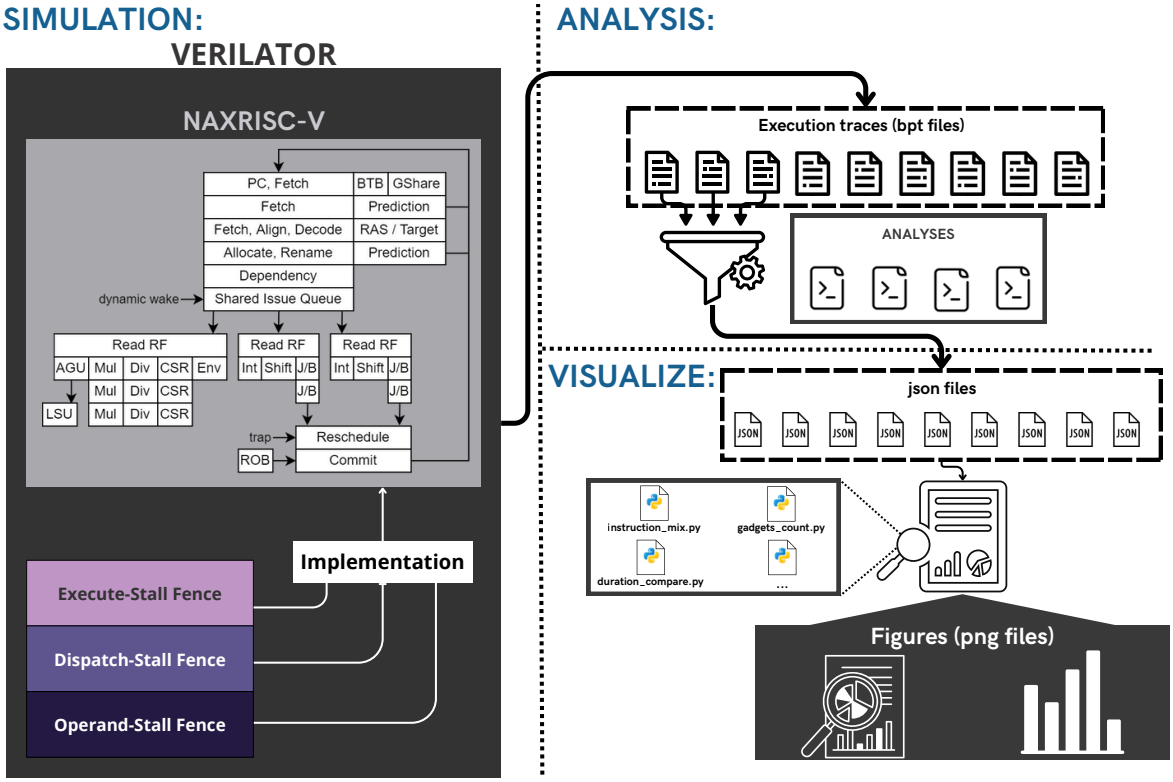


Figure 6.12 – The executable files generated by the compiler are run one by one on the three different NaxRISCV implementations simulated in Verilator. The resulting execution traces are then processed to generate statistical figures.

Analyse: Different analysis passes were run to extract metrics from the raw execution trace data: instruction mix, performances, etc. These traces also enable extraction of the microarchitectural state when a vulnerability is detected. The workflow is illustrated in Figure 6.12.

Visualize: A set of programs that take the generated data and produce clear statistical figures.

A "semantic-policy_hardware" configuration thus refers to benchmark binaries compiled with a given **semantic**, inserted according to a specific **policy**, and executed on the simulator representing the corresponding **hardware**.

For example, `specall-before_load-execute` indicates that the `specall-before_load` fence-policy (section 6.2) is applied to each program in **Embench** and executed on the processor using the *execute-stall* mechanism (section 6.3.3).

The environment's modular design enables direct comparison of mitigation strategies irrespective of their approach, mechanism, or implementation domain (hardware or software), and supports evaluations that combine multiple mitigations within a single experiment.

6.5.1 Results & Observations

For any semantic-policy_hardware configuration, the security metric is the sum of Spectre gadgets detected across all benchmarks for this configuration.

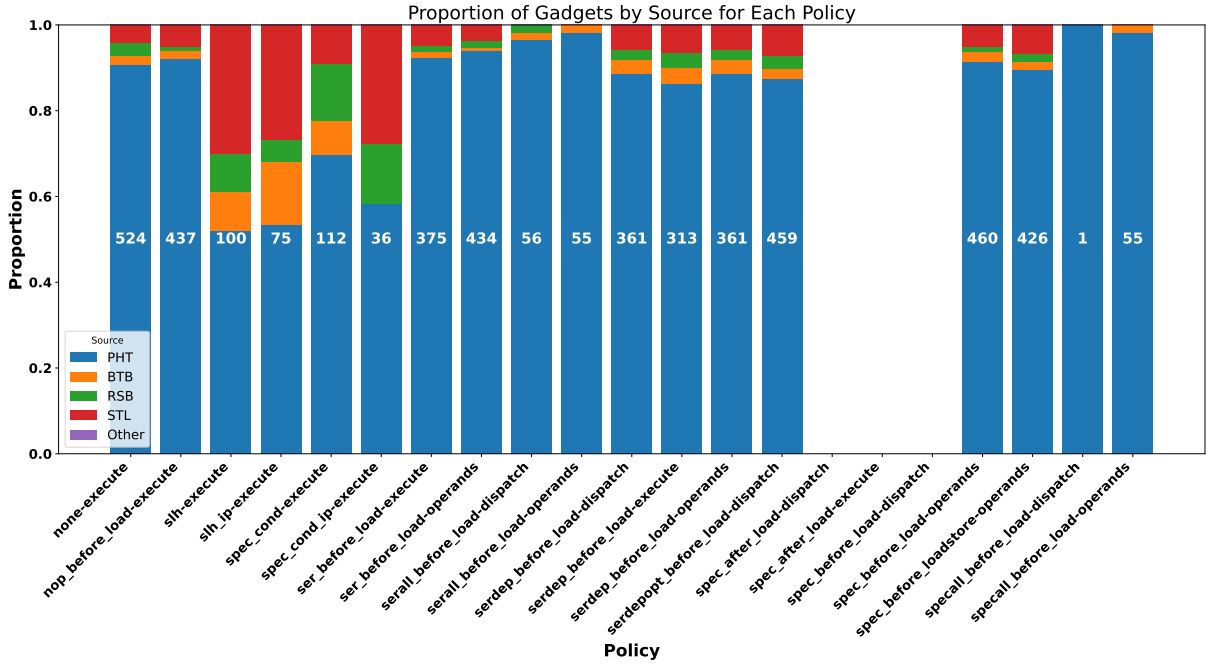


Figure 6.13 – Spectre gadgets by speculation sources, the center number is the gadgets count.

Figure 6.13 depicts the proportions of speculation sources for the Spectre gadgets that were detected for several semantic-policy_hardware configurations.

As shown in Figure 6.13, most of the gadgets found originate from a mispredicted branch direction (Spectre-PHT). It is also worth noting that some implementations are able to mitigate all kinds of gadgets while others focus on disabling only specific gadgets.

For example, Figure 6.13 illustrates that SLH has the highest relative proportion of other Spectre variants compared to the number of Spectre-PHT gadgets. This is expected:

SLH is tuned to harden PHT-type gadgets, so it primarily affects that class; other variants are untouched.

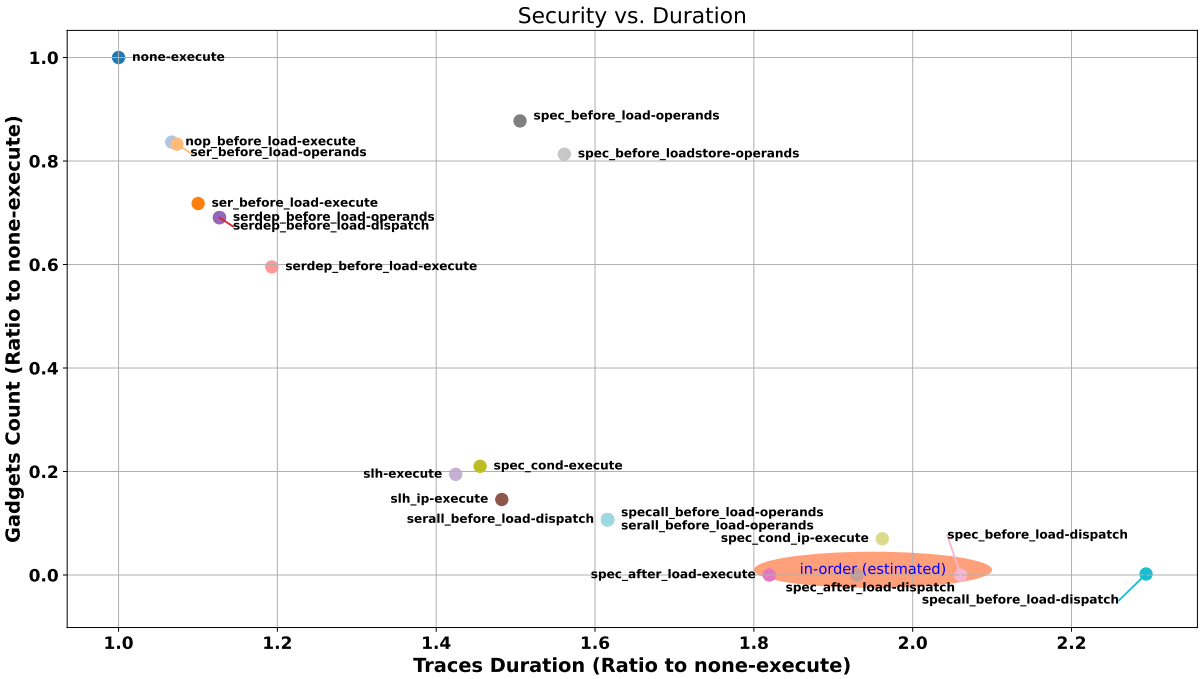


Figure 6.14 – The semantic-policy_hardware configurations according to both their security and performance metrics.

Figure 6.14 represents where the different semantic-policy_hardware configurations stand in terms of gadget-count to performance loss ratio, compared to an unmodified processor. It is a graphical representation of the data contained in Table 6.2. An evident Pareto front can be observed, ranging from the best-performing, unprotected configuration to the worst-performing, most-protected configuration.

The red area in Figure 6.14 represents an estimated range for an in-order processor, as reported in previous studies [6], [44].

The SLH mitigation, which does not rely on speculation barriers, lies on the Pareto front. While this mitigation is designed to target only Spectre-PHT gadgets, Figure 6.13 shows that SLH does not eliminate all PHT gadgets, with a small number persisting and indicating incomplete coverage. Upon closer inspection, permitted by our tooling, it appears that most of these gadgets are due to the `slh` inter-procedural predicate being loaded from the stack. Furthermore, `slh` is applied before register allocation, with spilling that introduces new unmitigated `load`s. Our tests correctly determine that these `load`s

could potentially be hijacked by an attacker, thus forming a Spectre gadget.

More generally, proving the correctness of a compiler-level mitigation pass can never provide a perfect guarantee: the arbitrary control flow of the microarchitecture can circumvent assumptions made in proofs applied to the binary.

Our policy `spec-cond_execute` implements the same predicate technique but, using our new instruction **`fence.cond`**, has a security-performance trade-off close to SLH. However, instead of poisoning load pointers, the **`fence.cond`** disables any speculative execution, incurring a slight overhead compared to SLH.

```
    blt         x2, x3, .L_EQUAL
    lb          x1, 0(x4)
    fence.spec  x1, x1

.L_EQUAL:
```

Figure 6.15 – The `spec-after_load` policy inserts a **`fence.spec`** immediately after every **`load`**. Since the **`fence.spec`** cannot execute speculatively, the `x1` register cannot be used by subsequent instructions until all speculation gadget before fence commits.

Moreover, `spec-after_load` policies, which insert a *fence.spec* after every **`load`** and use its destination register as operands (illustrated in Figure 6.15), have successfully prevented all gadgets in our benchmarks, as they prevent the use of any data loaded speculatively.

In contrast, placing a fence instruction before the **`load`**, as in `spec-before_load`, is one of the least effective policies in terms of both security and performance. Although it delays the execution of the **`load`** until all older speculative instructions complete, it does not guarantee that the load address itself is non-speculative, since NaxRISCV performs dependency speculation with memory instructions (section 2.3.5). Consequently, the **`load`** can simultaneously act as a speculation gadget (during loadstore dependency speculation) and as a disclosure gadget (by loading data from a speculative address). Notably, this represents the only Spectre gadget scenario that involves just two instructions.

This policy suffers from the same limitation as the *LFENCE/JUMP* mitigation discussed in section 5.2.1, which merely reduces the length of transient execution but remains exploitable.

6.5.2 Discussion & Implications

These results address many of the questions regarding the performance of selective speculation.

First, they confirm that the effectiveness of selective speculation is highly dependent on the precise definition of the policy. Moreover, the results reveal a clear trade-off between performance and security, forcing any approach to balance the two. Figure 6.14 shows that some policyhardware combinations achieve better security and performance than others.

However, since diverse techniques targeting load instructions, whether compiler-based (SLH) or hardware-based (most fence policies), form a clear Pareto front (see Figure 6.14), it is evident that the ability to execute memory requests speculatively remains the main performance driver in the NaxRISCV core.

Within our threat model, which assumes that all loaded data must be protected, configurations that eliminate all Spectre gadgets reduce performance to levels comparable to those of an in-order core.

This directly answers the central question regarding the effectiveness of selective speculation:

The indiscriminate use of selective speculation, which is the only viable strategy without microarchitectural knowledge of which data is confidential, does not allow for secure execution at out-of-order performance levels.

6.5.3 Limitations

Given the significant effort required to experiment with both custom out-of-order cores and custom compilers, some choices were dictated by technical feasibility and may limit the scope of the results.

First, to achieve a secure implementation, the number of gadgets must be zero. Any non-zero value implies an attacker could potentially exploit the specific conditions that produce a gadget. The security metric therefore indicates whether an implementation appears secure for the observed executions, but it does not constitute a formal proof. Execution traces represent only executed code, so the results apply only to those executions; the absence of Spectre gadgets in the traces is not a proof that gadgets cannot appear, only that they did not appear in the observed runs (and thus are likely less probable).

Second, a larger NaxRISCV configuration with wider issue width and a larger issue queue could yield better performance at the same security level by enabling more re-

ordering. However, in the hardware implementations that adopt execute-stall semantics for fence instructions (e.g., **`fence.cond`**), those fence instructions are not reorderable relative to one another, which imposes a strict limit on overall reordering.

Third, the `delay-on-miss` mitigation, delaying execution of speculative **`load`**s only when the target is not in cache, cannot be efficiently evaluated here because the Embench benchmarks exercise hot execution paths where instructions and data are typically already in cache. Moreover, `delay-on-miss` assumes a different threat model (that data in cache is public). Under the present threat model, where cached data may be confidential, `delay-on-miss` behaves like `none-execute`: fast but not secure. We implemented `delay-on-miss` on NaxRISCV and confirmed this behavior.

Finally, since microarchitectural control flow can be arbitrary, and thus any fence might be speculatively bypassed, one might expect that no policy could prevent all gadgets. Therefore, even if no speculation barrier-based policy can guarantee the complete absence of gadgets, it may still offer strong security in practice as the semantics of the fence can be integrated directly into the target instruction. For example, a **`fence.spec`** semantic can be applied directly in **`load`** instructions, which is simulated as `spec-before_load-execute` in our test.

6.5.4 Area comparison

Since all the changes that were brought to the NaxRISCV core are synthesizable, we can also measure the hardware costs of the proposed mechanisms. As a baseline, we considered the core configured as dual-issue with a 32-level deep Issue Queue, with support for single and double precision floating point operations, compressed instructions, and three EUs in total.

Table 6.1 – FPGA Resources Comparison for Fence Implementations

Fence Implementation	**Look-Up Tables (LUTs)**	**Flip-Flops (FFs)**	**FMAX**
Baseline	25811	15105	51.6 MHz
`execute-stall`	32375 (+25%)	16241 (+8%)	38.1 MHz
`dispatch-stall`	29076 (+13%)	15437 (+2%)	50.0 MHz
`operand-stall`	31024 (+20%)	15318 (+1%)	32.9 MHz

A *Xilinx XC7K325T* was used as the target Field-Programmable Gate Array (FPGA), and synthesis was done using Xilinx Vivado 2023.1. In its base configuration, the NaxRISCV

uses ≈ 26k LUTs and ≈ 15k FFs, and the maximum operating frequency reported by Vivado is 51.6 MHz.

As shown in Table 6.1, the overhead is relatively significant with +6.5k LUTs or +25% at best for the `execute-stall` hardware variant. Much of this overhead comes from adding a dedicated EU to support speculative fences, a unit that exists only in the `execute-stall` implementation (see section 6.3.3).

The maximum operating frequency of the `execute-stall` hardware also degrades most noticeably. This drop is driven by the use of a dynamic-latency mechanism for speculative fences, which requires synchronous communication between the `Dispatch plugin` and the EU. By contrast, a static-latency implementation can be handled internally inside `Dispatch plugin` and imposes fewer timing constraints.

The `dispatch-stall` variant has the smallest area overhead: it reuses much of the base implementation and requires only a few additional structures. Both the `execute-stall` and `dispatch-stall` designs do pay an extra cost (≈ 0.9k LUTs) for the speculation detector implementation.

The `operand-stall` implementation shows a different cost profile and suffers a marked frequency penalty. To satisfy its semantics, the design must delay dependency resolution for speculative fences: if the executed instruction is the last one to write to the fence's `rs1`, the dependency is not cleared immediately but only once that instruction reaches the commit stage (see subsection 6.3.4). Implementing this behaviour requires extra control operations and larger issue-queue entries (e.g. an additional $LAST\_WRITE$ field), which both increase the critical-path length and resource usage.

Note that the cost shown in Table 6.1 corresponds to the total overhead caused by the three fences.

Table 6.2 – List of evaluated policies and results

		Gadget counts			Benchmark duration geomean (hot cycles)		
Policy name	**Description**	`execute`	`dispatch`	`operands`	`execute`	`dispatch`	`operands`
`none`	No policy applied (baseline).	514			3.6 M		
`nop-before_load`	Insert **`nop`** instruction *before* each **`load`**.	430			3.8 M		
`ser-before_load`	Insert **`fence.ser rA, rA`** instruction *before* each **`load rB, offset(rA)`**.	369	428	428	3.9 M	3.8 M	3.8 M
`serall-before_load`	Insert **`fence.ser x0, x0`** *before* each **`load`**.	16	55	55	7.0 M	5.8 M	5.8 M
`serdep-before_load`	Insert **`fence.ser rB, rA`** with `rB` used as address by the following **`load`** and `rA` register from the "most dominant branching instruction operands". It is a naive approach to add serialization between **`load`**s and most dominant branching instruction if there is one.	306	355	355	4.3 M	4.0 M	4.0 M
`slh`	SLH implementation for RISCV.	100			5.1 M		
`slh-ip`	SLH implementation with inter-procedural predicate transfer via a stack pointer.	75			5.3 M		
`spec-after_load`	Insert **`fence.spec rB, rB`** instruction *after* each **`load rB, offset(rA)`**.	0	0	0	6.5 M	6.9 M	6.6 M
`spec-before_load`	Insert **`fence.spec rA, rA`** instruction *before* each **`load rB, offset(rA)`**.	0	0	451	7.4 M	7.4 M	5.4 M
`spec-before_loadstore`	Insert **`fence.spec rA, rA`** instruction *before* each **`load rB, offset(rA)`** or **`store rB, offset(rA)`**.	3	3	418	8.1 M	8.0 M	5.6 M
`spec_cond`	Insert **`fence.cond rA`** at the beginning of each basic block that contains **`load`**. It takes as operand `rA`, an always updated predicate computed as in `slh` policy.	108			5.2 M		
`spec-cond-ip`	As `spec_cond` with inter-procedural predicate transfer.	36			7.0 M		
`specall-before_load`	Insert **`fence.spec x0, x0`** *before* each **`load`**.	0	1	55	8.4 M	8.2 M	5.3 M

Gadget counts in Table 6.2 are the same ones as used to generate Figure 6.14 but different from those in Figure 6.13. Indeed, in Figure 6.13 two gadgets occurring at the same acquisition address are counted twice if they have different speculation sources, but are counted once in Table 6.2 and Figure 6.14.
Performance is measured as the geometric mean of the number of hot cycles (after the warmup) for the benchmark suite.

Chapter 7

Architectural Secret Values

This second contribution is not yet published.

The first contribution (Chapter 6) shows that the selective speculation approach fails to efficiently protect the microarchitecture against Spectre. This inefficiency is not solely a flaw of the selective speculation approach itself but also reflects an inadequate threat model. As the speculative execution is completely arbitrary, any data in the program can theoretically be accessed speculatively by any `load` in the program. Because the hardware is oblivious to which data are confidential, treating all data as secret imposes a large performance penalty.

Motivated by this observation, a new threat model and semantics are proposed.

7.1 New Threat model

Metadata is data that provides information *about other data*. In simple terms, it's a label that describes information.

To prevent this, we introduce a new approach: **Secret data are marked in source code and this metadata is propagated through the compiler into the microarchitecture, allowing hardware to focus protection only on explicitly flagged values.**

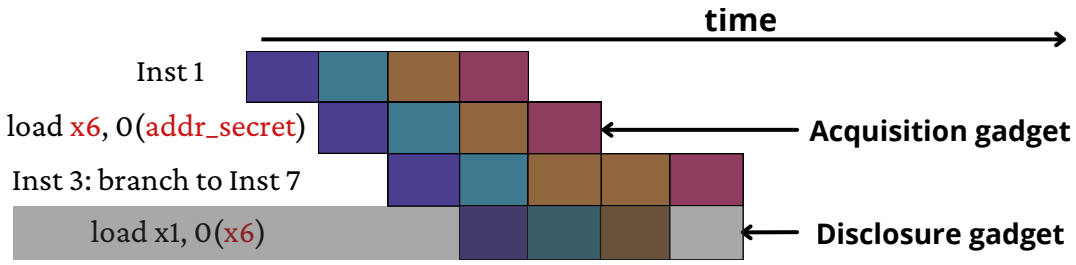


Figure 7.1 – A legitimately loaded secret can be exposed through a subsequent disclosure gadget. The second `load` instruction that leaks the secret does not necessarily need to execute speculatively. The data loaded by the second `load` have a tied dependency to the secret data, which allows an attacker to infer the value.

Confidentiality is a fundamental security property that ensures information is accessible only to authorized entities and protected from unauthorized disclosure.

With this richer program-level knowledge in the microarchitecture, beyond protecting flagged values against Spectre attacks, we can extend protections to threats omitted from the initial threat model: the cases where confidential data are legitimately read but leaked via microarchitectural state, either during non-speculative or speculative execution. This additional threat model is illustrated in Figure 7.1.

7.2 The semantics of Architectural Secret Values

We introduce the concept of architectural secret values: secret data within the program are explicitly marked as confidential. This information is then propagated from the compilation stage down to the microarchitectural level. The microarchitecture must ensure that confidential data are manipulated only through constrained operations and accessed exclusively by authorized instructions in non-speculative mode.

This approach is achieved by assigning clear responsibilities to each layer of the software-hardware stack:

7.2.1 Compiler semantics

Source-level tags

```
int __attribute__((annotate("confidential"))) secret = 42;
```

Figure 7.2 – Annotation of the `secret` variable as confidential, allowing the compiler to identify and apply protection to the data, see subsection 7.3.1.

At source level, secret values must be explicitly marked. Figure 7.2 shows an example of a secret declaration in **C**. These annotations serve as metadata for the compiler. It is the programmer's responsibility to indicate which values must remain secret so that subsequent compilation steps and the hardware can apply the appropriate protections.

Declassification is the process of removing the security classification of information. In our case, it indicates the operation to remove the *confidential* flag on a data after ensuring that doing so does not violate security policies.

The compiler must also provide an operation allowing the programmer to declassify data. A binary value resulting from a comparison with secret data does not reveal the secret value itself and can therefore be declassified, allowing it to be manipulated without restriction.

Compiler tracking

The compiler carries the heaviest burden: it must ensure that confidential labels are correctly transferred to the hardware level. Sensitive values are not limited to variables explicitly tagged in the source: any value derived from tagged data that could be used to infer the original secret must inherit the confidential attribute. Thus, the compiler performs taint-style propagation of the confidential attribute and only permits declassification through an explicit declaration in the source code.

7.2.2 Hardware semantics

Memory protection is required to prevent any unauthorized access to secret data. However, data can still be transiently read through a legitimate `load`, leading to its leakage via Spectre. To address this, the microarchitecture needs to identify legitimate

`load` instructions that are authorized to access sensitive data and prevent their speculative execution. This is achieved through the introduction of **confidential registers**, a dedicated subset of registers that are exclusively allowed to load and manipulate sensitive data. Therefore, the compiler must be aware of this set of **confidential registers** and use them exclusively for handling sensitive values.

Confidential register

The hardware exposes a dedicated class of registers, called *confidential registers*, for holding secret data.

A **Control and Status Register (CSR)** is a special-purpose register used to configure and monitor processor behaviour. For instance, the `mstatus` CSR defines and manages privilege-related fields.
A General Purpose Registers (GPR) is a register used to store temporary data, operands, or results during program execution.

The 32 integer GPR registers can be classified as either public or confidential. A dedicated `CSR`, named `MSECRET`, tracks this status: each of its lowest 32 bits corresponds to one register, with the *i-th* bit set to `1` if register `xi` is confidential, or `0` if it is public.

Under this design, the hardware treats confidential registers as **static** secret holders: once a register is flagged confidential, every access to it must respect the confidentiality policy.

To simplify the implementation, we restrict confidential registers to the integer registers only.

When a register is marked confidential, the microarchitecture enforces strict usage rules to prevent any leak of the value stored during manipulation:

Non-speculative secret Speculative execution assumes optimistic conditions (predictions are correct, data read are legitimate, no exceptions occur, etc.), which improves performance but opens transient-execution channels exploited by Spectre/Meltdown. To eliminate such transient leaks, any load that writes into a confidential register must be executed non-speculatively.

It is not necessary to delay a store operation, as the value is only written to the target memory during the retirement of the instruction; see section 3.3.

Operation restriction Operations involving confidential registers are restricted to prevent side effects that could leak secret values:

Branch on secret: Using a confidential register as a branch operand (either in the condition or as the target of an indirect jump) is forbidden, since control-flow outcomes reveal information about the secret.

Secret as memory address: Using a confidential register as (part of) a memory address is forbidden. Employing a secret as an address allows attackers to influence cache or memory state based on the secret (the classic *disclosure gadget* used by Spectre).

Variable-time instructions: Confidential registers must not be operands of instructions whose timing depends on input values (implementation-defined).

The compiler is responsible for generating code that respects these restrictions, and any attempt to violate them triggers a security exception. This assures that only compiler-validated programs that respect the confidential semantics can execute.

Memory protection

However, protecting only register contents is not sufficient to guarantee the confidentiality of sensitive data, since any `load` may speculatively fetch a secret into a non-confidential register.

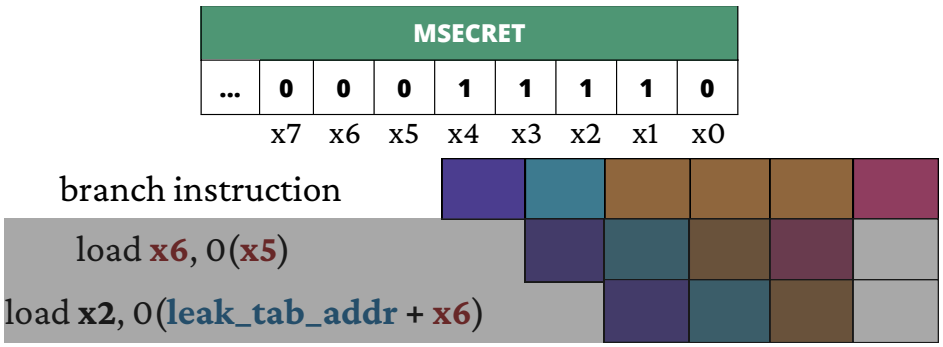


Figure 7.3 – Registers `x1` to `x4` are classified as confidential registers. However, despite this protection, any load using a non-confidential register, such as `x6` in this example, may still speculatively read a confidential address.

In the example in Figure 7.3, `x5` is not a confidential register, so it can be used as the address of a **`load`** instruction. Assuming the compiler generates a correct program that uses confidential registers exclusively for confidential data. However, this does not prevent `x5` from holding the address of sensitive data, whether speculatively or not. As a result, the **`load`** may speculatively read the actual value into a non-confidential register, `x6`, which may leak the secret value.

Thus, protection must extend to the memory.

Two different approaches can be used to protect the confidentiality of data in memory:

Access control mechanism: A more intuitive approach is to use memory tagging to label sensitive data directly in memory and verify each memory access to ensure it has the proper authorization.

Memory-tagging architectures associate a small tag (Confidential / Public) with each fixed memory granule (e.g., 4 or 16 bytes). These tags are stored in a separate tag store and therefore require a small amount of extra metadata memory.

When a load targets tagged data, the microarchitecture can check the tag and enforce additional rules: for instance, a load into a non-protected register may be disallowed if the data is marked Secret.

The flag can be updated by a dedicated instruction executed by software, and the operation can be performed only in a non-speculative state. Nevertheless, modifications to the Confidential/Public tag must be protected to prevent an attacker from clearing the "Confidential" state of sensitive data, which would make it vulnerable again to a speculative unprotected load.

To strengthen this mechanism, capability-based architectures such as Capability Hardware Enhanced RISC Instructions (CHERI) [76], [77], [83] can be applied to enforce permission checks on Secret/Public flag updates.

CHERI introduces the notion of a capability: a hardware-protected pointer that carries not only an address but also metadata describing its valid region and allowed operations.

In this context, the term **bounds** refers to the range of memory addresses that a program is allowed to access.

A capability typically contains:

Address the location it refers to.

Bounds the inclusive range of memory the capability may access.

Permissions the operations allowed (read, write, execute, confidential, etc.).

Because the hardware checks these properties on memory operations, *capabilities* prevent out-of-bounds accesses, assuring that only the legitimate program can modify its Confidential/Public flags.

Note that, despite the protection added by memory tagging, it does not stop accesses that are *authorized but contextually incorrect*. For example, a speculatively executed path might legitimately have permission to read an address but should not do so in the current control flow (misspeculation), as illustrated in Figure 7.1. This lack of protection can be completed by the confidential register mechanism, which prevents any speculative use of sensitive data.

Encryption is a security mechanism that transforms information (*plaintext*) into a scrambled form (*ciphertext*) to protect its confidentiality, using mathematical operations and a secret key.
Decryption is the reverse operation, converting ciphertext back into *plaintext* using the appropriate cryptographic key.
In cryptography, a **key** is a secret piece of information used to perform *encryption* and *decryption*.
The term **plaintext** refers to the original data before any encryption.
The term **ciphertext** refers to the transformed version of the plaintext after encryption, which can only be converted back using the correct decryption key.

Encryption memory mechanism: Another option is to use memory encryption techniques: ensuring that secret values resident in memory are stored as ciphertext and are only decrypted by the processor under controlled conditions.

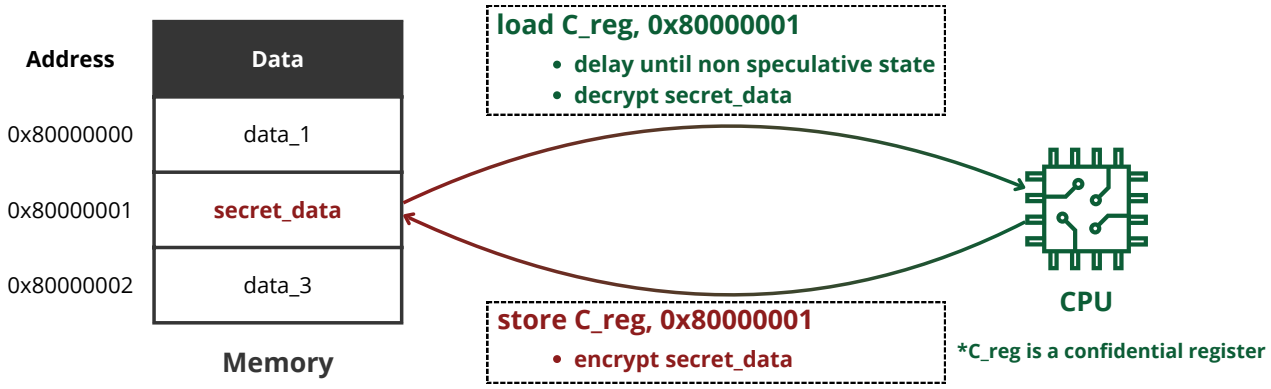


Figure 7.4 – Data are stored in encrypted form in memory. Any data loaded into a confidential register is decrypted during the operation, which only occurs in a non-speculative state. Any store from a confidential register must encrypt the data before writing it back to memory.

As illustrated in Figure 7.4, encryption and decryption operations are applied to confidential registers:

— Stores sourcing from a confidential register encrypt the value before writing to memory.

— Loads that will write into a confidential register cause the hardware to fetch the encrypted value and perform an in-core decryption step that places plaintext only into the confidential register; this step executes only when the load is non-speculative.

Therefore, a misspeculated `load` that reads sensitive data into public registers yields ciphertext, preventing transient leakage of plaintext secrets.

Unlike memory-tagging, memory encryption protects the memory value instead of adding access control to it. However, it requires key management to generate and securely maintain the key during execution. For better security, a dedicated key is randomly generated at the start of the program.

Since memory tagging entails an additional layer of mechanisms, its implementation is inherently more complex and demanding than memory encryption. Therefore, this manuscript focuses solely on memory encryption to validate the proposed protection workflow.

7.3 Implementation in the compiler

7.3.1 Annotations from source to `RISC-V` back-end

We developed a static taint analysis in the `RISC-V` LLVM back-end using annotated global variables as sources of secrets. Two labels are used to inform the compiler about the instruction state:

Confidential: Mark for a variable that is (or can be the source of) a secret.

Public: To indicate that the value is not sensitive and does not need protection.

By default, data are considered *public* and only labeled as *confidential* explicitly by the program. For instance, in C the annotation can be done using an attribute as illustrated in Figure 7.2.

```
#define declassify(X)
({
int out;
asm volatile("declassify %0, %1" : "=r"(out) : "r"(X));
out;
});
```

Figure 7.5 – This inserts a pseudo-instruction that works as an anchor for the compiler since it is marked as volatile. It therefore cannot be moved. The *declassify* function takes a *confidential* value `X` as input and returns the same value as output, but labeled as *public*.

The `declassify` function shown in Figure 7.5 provides an explicit mechanism to remove the *confidential* tag associated with a sensitive value. It takes a confidential register as input X and writes the resulting value into a public register *out*.

It can be used as

```
// Tag start_as_secret as confidential
int __attribute__((annotate("confidential"))) start_as_secret = 42;

// declassify start_as_secret, tagging it as safe.
int not_a_secret_anymore = declassify(start_as_secret);
```

Figure 7.6 – Declassifying the variable *start_as_secret* in C.

In more modern languages, such as *Rust*, the type system would be responsible for the declassification.

A **SSA** is a technique for representing programs in which each variable is assigned exactly once.
IR is a temporary representation of the program used by a compiler mainly for analyses and transformations conveniences. For instance, it can adopt the SSA property.
MIR is a medium-level representation of the program that derived from the IR, allow the compiler to perform optimizations and analyses independent of the target hardware architecture
A **`RISC-V` back end** is the part of the compiler responsible for translating the program's IR into `RISC-V` machine instructions, making the code executable on a `RISC-V` processor.

7.3.2 Taint analysis

The taint analysis is performed over a MIR just before register allocation. At that specific location in the compilation pipeline, the MIR uses virtual registers (e.g., %v1, %v2, etc.) that keep the SSA property.

The SSA properties bring precision and simplicity to taint analysis and ensure that no aggressive transformations will sabotage the analysis, since it is placed just before register allocation.

The taint is implemented using a new **subclass of register** in LLVM called Secret General Purpose Registers (SGPR), which represents the set of confidential registers available at this level. The taint propagates the secret naively throughout all instructions. It also uses LLVM alias analysis in order to taint pointers that may refer to the same secret value in memory. We acknowledge that the static taint analysis may be very pessimistic, but as a proof of concept it can demonstrate the utility of confidential registers.

7.3.3 Split Register allocation

After the taint analysis, the register allocation is called function-by-function on the SGPR (e.g., the virtual registers that are tainted). This will reserve, in a specific order, the architectural registers needed to carry the secret values of the program. Then it is called again on the rest of the registers.

The set of confidential registers defined in the microarchitecture is manually integrated into the compiler to allow it to adapt the program accordingly.

7.4 Implementation in the microarchitecture

7.4.1 Confidential register

As discussed earlier in chapter 4, the `RISC-V` architecture defines a third privilege level, *machine privilege*, which is the most privileged level, above user and supervisor/kernel privileges. Machine mode has full access to all hardware resources, including registers such as GPRs and CSRs.

Writing to the `MSECRET` register, which defines the confidentiality status of each general-purpose register, must be restricted to machine mode and kept coarse-grained. Allowing frequent updates from lesser privileged levels (e.g., if each function preamble rewrote `MSECRET`) could enable an attacker to clear the confidentiality of sensitive registers. A safer design is to make `MSECRET` read-only, with the mapping of confidential registers statically defined by hardware.

During the execution stage, the EU checks the `MSECRET` register to determine whether operands are confidential and to ensure that the restrictions described in section 7.2.2 are enforced. The EU raises an exception if any of these conditions are violated, halting the pipeline immediately to prevent further leakage of confidential data.

In NaxRISCV, CSR registers are managed as memory-like accesses; the EU must issue a read request with the `MSECRET` address to get the registers' confidentiality information. This repeated lookup introduces additional latency for every instruction operating under confidentiality constraints.

To minimize this performance cost, a mirror of the `MSECRET` value, a lightweight 32-bit register, is implemented locally inside the EU. This local copy is automatically updated whenever the `MSECRET` CSR is modified. As a result, the EU can directly access the confidentiality status of each register without performing memory-like access to the CSR.

7.4.2 Encryption mechanism

A **symmetric encryption** is an encryption method that uses the same secret key for both encryption and decryption. It is computationally fast and primarily used to ensure data confidentiality.

In our case, symmetric encryption is better suited to our constraints: it is a lightweight and fast operation that ensures the confidentiality of secret data. Among existing symmetric encryption schemes, a hardware feature known as Inline Memory Encryption (IME) corresponds to our needs, as it allows on-the-fly encryption: the encryption and decryption happen immediately and automatically as data moves, without needing separate software steps or explicit delays.

The **National Institute of Standards and Technology (NIST)** is a federal agency responsible for developing and maintaining technical standards that ensure the security and interoperability of cryptographic systems.
The **Advanced Encryption Standard (AES)** is a symmetric encryption standardized by NIST.
The **XEX-based Tweaked CodeBook mode with Ciphertext Stealing using AES (XTS-AES)** is a disk encryption mode defined by NIST that provides confidentiality for data stored in fixed-size sectors by combining AES encryption.

The NIST recommends the XTS-AES [21] mode for securing data at rest, particularly in storage devices and memory systems.

A LSU is a specialized EU responsible for handling all memory accesses.

This approach requires robust key management within the LSU in order to apply the encryption mechanism directly to each memory operation. The key management generates a new key at the start and retains it throughout the execution of each program. A secret value must remain in plaintext only inside registers and be stored in encrypted form in higher levels of the memory hierarchy, such as cache memory.

7.5 Limitations and Discussion

7.5.1 Register allocation limitation

Dividing the available registers into separate public and confidential sets reduces the flexibility of register allocation. For instance, a cryptographic workload, such as an AES implementation, may ideally require many registers to hold confidential values. If no registers remain available, the allocator is forced to spill values to memory to free registers for other critical operations. This can occur in both the confidential and non-confidential register sets. Since memory accesses have high latency, such spilling can significantly degrade performance.

The performance impact depends on both the number of registers reserved for confidential use and the register pressure of the workload. To minimize the performance loss, the number of registers reserved for secrets can be made adaptive: the compiler first measures or estimates a program's secret-register demand, then requests an appropriate `MSECRET` configuration from machine privilege.

7.5.2 Encryption limitation

The XTS-AES mode is not without limitations. In particular, it operates on a fixed granularity of 128 bits, meaning that smaller pieces of data cannot be encrypted independently of their 128-bit block. To modify only a portion of the data, the entire block must first be decrypted, then updated, and finally re-encrypted. Furthermore, encrypting the same 128-bit block produces identical ciphertext, which can potentially expose patterns in the encrypted data.

7.5.3 Approach limitation

The main obstacle to adopting architectural secrets is less about technology and more about economics and developer behavior. This model relies on developers correctly annotating sensitive data in their applications. Yet, can this be realistically expected when skipping annotations may boost performance, and the resulting security flaws could remain hidden for years?

In practice, the complexity of proper annotation would likely drive developers to rely on specialized, security-audited secret-handling libraries rather than annotating secrets

themselves. Just as developers today are advised to "never implement your own cryptographic library", future best practices may well evolve into "never implement your own secret-handling library".

From a security perspective, this shift would be beneficial. Moreover, the confidential register offers a smooth migration path toward architectures supporting secrets at the hardware level. By writing zero to the `MSECRET`, architectural secrets can be disabled, maintaining backward compatibility with conventional cores.

Chapter 8

Conclusion

Despite a strong understanding of how Spectre leaks data, research on mitigation continues to face significant challenges in achieving complete protection while maintaining acceptable performance overhead. One major obstacle to clear progress in this field is the lack of standardized evaluation methods, which makes fair comparisons between state-of-the-art mitigations difficult.

In the first contribution of this thesis, we provide an in-depth evaluation of the *selective speculation* approach, one of the most widely adopted mitigation strategies. In our threat model, every `load` instruction is assumed to potentially access sensitive data and therefore requires protection. Through experimentation, we identified a key challenge of this approach: the timing of the mitigation intervention has a critical impact on performance, as discussed in section 6.2. Our evaluation highlights the need for standardized evaluation environments in mitigation research and demonstrates that the selective speculation approach is inefficient at providing full protection against Spectre while maintaining acceptable performance under this strong threat model (see subsection 6.5.2).

To address this trade-off between performance and security, the second contribution of this thesis introduces the concept of **architectural secret values** (see chapter 7). In this approach, confidential data are marked from the source code level and tracked down to the EU. This additional semantic information enables the microarchitecture to focus security only on confidential data, providing more fine-grained protection and reducing performance loss.

The architectural secret value approach combines register and memory protection to ensure the confidentiality of sensitive data regardless of its location. Moreover, by enriching the microarchitecture's understanding of the program's data, this contribution allows us to redefine the threat model and expand the protection surface. By preventing speculative use of secret values, the approach not only protects against all existing Spectre variants but also provides a strong foundation against potential future transient vulnerabilities as processors evolve.

We hope that this work contributes to a better understanding of the effectiveness of the selective speculation approach for mitigating Spectre without discriminating between data at the microarchitectural level.

8.1 Research Perspective

As far as we know, no exploitation of Spectre has been reported in the real world to date. This is due to the complexity of such an attack: an external adversary must possess a deep understanding of the target microarchitecture and know precisely where the sensitive data resides in memory in order to leak it speculatively.

The fact that Spectre operates at the microarchitectural level is both an advantage and a limitation for attackers and defenders alike. On one hand, the diversity of microarchitectures makes it difficult for attackers to design a fully generalizable exploit. On the other hand, this same diversity complicates the design of universal mitigation mechanisms.

Spectre has drawn increasing attention from the security research community for several reasons:

— It represents a fundamentally new class of microarchitectural attacks that may conceal even more severe security threats in the future.

— Spectre attacks are indeed feasible with sufficient knowledge of the underlying microarchitecture and the executed program.

— The continuous drive for higher processor performance tends to favor attackers more than defenders. As processors become faster, more instructions can be executed within the speculative window, and high-latency instructions are predicted earlier to maintain performance. This enlarges the attack surface by giving attackers more time to leak data speculatively, as well as more potential speculation gadgets and microarchitectural states that can be exploited as disclosure gadgets.

The exponential improvement in processor performance highlights the urgent need for parallel progress in security to keep pace with new optimization features that may introduce additional vulnerabilities. This concern goes beyond Spectre itself and encompasses a broader category of microarchitectural threats that are becoming increasingly real as processors evolve. Optimizations tend to increase the divergence between the microarchitecture and the architecture, creating more arbitrary and uncontrollable states in the microarchitecture that can eventually be used to leak data.

8.1.1 Standardized test environment

Several years after the first publication of Spectre, despite numerous efforts, no clear or unified progress has been achieved in mitigation research. This lack of consolidation has created a fragmented landscape of mitigation results, often leading to confusion among researchers and a dispersion of initiatives across different directions. The resulting uncertainty has made it increasingly challenging to establish new research projects in this field and may even discourage further contributions.

Establishing a standardized test environment is therefore essential to enable coordinated and cumulative progress. Such an environment may include representative benchmarks, a common set of processors, and a well-defined categorization of the threat model. It would provide a shared foundation for evaluating and comparing mitigation strategies, improving both the reproducibility and the relevance of research results.

Our current test environment may not yet be sufficiently complete to fulfill this role entirely. Nonetheless, it highlights the clear benefits of adopting a common framework for mitigation research. For instance, in our experience, we are able to compare the SLH mitigation to the selective speculation approach in the same test environment. A standardized environment has the potential to accelerate innovation, reduce duplication of effort, and build a more coherent body of knowledge in mitigation research.

Moreover, it may go beyond Spectre-like attacks, allowing the comparison of other protections against different vulnerabilities. It can be applied, for example, to Meltdown and side- or covert-channel protections.

8.1.2 Extending architectural secret value

The capability of the microarchitecture to identify secret values opens several possibilities for protecting them. Since the microarchitecture can adapt its semantics for each operation applied to these values, it can enhance security in a more fine-grained manner.

The second contribution ensures that no secret value is ever executed speculatively, thereby preventing speculation gadgets from operating. However, from the first contribution, we learned that **the later the intervention occurs, the better the performance**.

With the ability to detect secret values, the microarchitecture is theoretically capable of achieving strong protection with significantly improved performance: speculation gadgets may execute as usual, and the system intervenes only **when a disclosure gadget**

attempts to use a confidential register that is not architecturally correct.

This approach delays intervention to the latest possible stage of speculative execution, theoretically enabling complete protection against Spectre.

Obviously, the implementation of such protection would be tightly coupled to the target microarchitecture to ensure complete coverage of all disclosure gadgets. Nevertheless, this strategy can achieve the best performance of a selective speculation approach, **if** the performance overhead caused by register spilling remains negligible, which depends on the workload characteristics of the program.

Bibliography

[1] M. F. Abdul Kadir, J. K. Wong, F. Ab Wahab, A. F. A. Abidin Bharun, M. A. Mohamed, and A. H. Zakaria, « Retpoline technique for mitigating spectre attack », *in 2019 6th International Conference on Electrical and Electronics Engineering (ICEEE)*, 2019, pp. 96–101. DOI: 10.1109/ICEEE2019.2019.00026.

[2] O. Aciiçmez, Ç. K. Koç, and J. Seifert, « Predicting secret keys via branch prediction », *in Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings*, M. Abe, Ed., ser. Lecture Notes in Computer Science, vol. 4377, Springer, 2007, pp. 225–242. DOI: 10.1007/11967668_15. [Online]. Available: https://doi.org/10.1007/11967668%5C_15.

[3] S. Ainsworth and T. M. Jones, « Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state », *in 47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020*, IEEE, 2020, pp. 132–144. DOI: 10.1109/ISCA45697.2020.00022. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00022.

[4] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, « Port contention for fun and profit », *in 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, IEEE, 2019, pp. 870–887. DOI: 10.1109/SP.2019.00066. [Online]. Available: https://doi.org/10.1109/SP.2019.00066.

[5] N. Amit, F. Jacobs, and M. Wei, « JumpSwitches: Restoring the performance of indirect branches in the era of spectre », *in 2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA: USENIX Association, Jul. 2019, pp. 285–300, ISBN: 978-1-939133-03-8. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/amit.

[6] Author(s), « Dnb: Delay-and-bypass scheduling to achieve 95% of ooo performance at in-order complexity », *in Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[7] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, « Specshield: Shielding speculative data from microarchitectural covert channels », *in 28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019*, IEEE, 2019, pp. 151–164. DOI: 10.1109/PACT.2019.00020. [Online]. Available: https://doi.org/10.1109/PACT.2019.00020.

[8] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, « Branch history injection: On the effectiveness of hardware mitigations against Cross-Privilege spectre-v2 attacks », *in 31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 971–988, ISBN: 978-1-939133-31-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/barberis.

[9] D. J. Bernstein, « Cache-timing attacks on aes », 2005.

[10] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, *et al.*, « Smotherspectre: Exploiting speculative execution through port contention », *CoRR*, vol. abs/1903.01843, 2019. arXiv: 1903.01843. [Online]. Available: http://arxiv.org/abs/1903.01843.

[11] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, « Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution », *in 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, E. Kirda and T. Ristenpart, Eds., USENIX Association, 2017, pp. 1041–1056. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck.

[12] A. Burgess, A. Whetter, G. Field, *et al.*, *Embench iot: Open benchmarks for embedded platforms*, https://github.com/embench/embench-iot, GitHub repository, 2020. [Online]. Available: https://github.com/embench/embench-iot.

[13] C. Canella, D. Genkin, L. Giner, *et al.*, « Fallout: Leaking data on meltdown-resistant cpus », *in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds., ACM, 2019, pp. 769–784. DOI: 10.1145/3319535.3363219. [Online]. Available: https://doi.org/10.1145/3319535.3363219.

[14] C. Canella, J. Van Bulck, M. Schwarz, *et al.*, « A systematic evaluation of transient execution attacks and defenses », *in Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19, Santa Clara, CA, USA: USENIX Association, 2019, pp. 249–266, ISBN: 9781939133069.

[15] R. Choudhary, J. Yu, C. W. Fletcher, and A. Morrison, « Speculative privacy tracking (SPT): leaking information from speculative execution without compromising privacy », *in MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*, ACM, 2021, pp. 607–622. DOI: 10.1145/3466752.3480068. [Online]. Available: https://doi.org/10.1145/3466752.3480068.

[16] I. Corporation, « Improving real-time performance by utilizing cache allocation technology », Intel Corporation, Tech. Rep., 2015, Accessed: 2025-09-25. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf.

[17] L. Daniel, M. Bognar, J. Noorman, S. Bardin, T. Rezk, and F. Piessens, « Prospect: Provably secure speculation for the constant-time policy », *in 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, J. A. Calandrino and C. Troncoso, Eds., USENIX Association, 2023, pp. 7161–7178. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/daniel.

[18] L. Developers, *Llvm seses - speculative execution side effect suppression*, https://groups.google.com/g/llvm-dev/c/EL8rUhvRCgo, Accessed: 2024-10-25.

[19] L. Developers, *Speculative load hardening*, https://llvm.org/docs/SpeculativeLoadHardeni
html, Accessed: 2024-10-25.

[20] M. Dillon, *Clarifying the spectre mitigations* DragonFlyBSD Users Mailing List, Online; posted January 9, 2018, Jan. 2018. [Online]. Available: https://lists.dragonflybsd.org/pipermail/users/2018-January/335637.html.

[21] M. Dworkin, « Recommendation for block cipher modes of operation: The xts-aes mode for confidentiality on storage devices », National Institute of Standards and Technology, Special Publication 800-38E, Jan. 2010, (cited on page 83). DOI: 10.6028/NIST.SP.800-38E. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-38e/final.

[22] M. Escouteloup, R. Lashermes, J. Fournier, and J. Lanet, « Under the dome: Preventing hardware timing information leakage », *in Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers*, V. Grosso and T. Pöppelmann, Eds., ser. Lecture Notes in Computer Science, vol. 13173, Springer, 2021, pp. 233–253. DOI: 10.1007/978-3-030-97348-3_13. [Online]. Available: https://doi.org/10.1007/978-3-030-97348-3%5C_13.

[23] J. Fustos, F. Farshchi, and H. Yun, « Spectreguard: An efficient data-centric defense mechanism against spectre attacks », *in Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, ACM, 2019, p. 61. DOI: 10.1145/3316781.3317914. [Online]. Available: https://doi.org/10.1145/3316781.3317914.

[24] J. Fustos and H. Yun, « Spectrerewind: A framework for leaking secrets to past instructions », *CoRR*, vol. abs/2003.12208, 2020. arXiv: 2003.12208. [Online]. Available: https://arxiv.org/abs/2003.12208.

[25] M. Ghaniyoun, *Moein ghaniyoun's website*, https://moeinghaniyoun.github.io/, Accessed: 2024-10-25.

[26] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, « Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks », *in 27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 955–972, ISBN: 978-1-939133-04-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/gras.

[27] C. Green, C. Nelson, M. Thottethodi, and T. N. Vijaykumar, « Safebet: Secure, simple, and fast speculative execution », *CoRR*, vol. abs/2306.07785, 2023. DOI: 10.48550/ARXIV.2306.07785. arXiv: 2306.07785. [Online]. Available: https://doi.org/10.48550/arXiv.2306.07785.

[28] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, « Hardware-software contracts for secure speculation », *in 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, IEEE, 2021, pp. 1868–1883. DOI: 10.1109/SP40001.2021.00036. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00036.

[29] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2017.

[30] J. Horn, « Speculative execution, variant 4: Speculative store bypass », 2018.

[31] G. Hu, Z. He, and R. B. Lee, « Sok: Hardware defenses against speculative execution attacks », *in 2021 International Symposium on Secure and Private Execution Environment Design (SEED), Washington, DC, USA, September 20-21, 2021*, IEEE, 2021, pp. 108–120. DOI: 10.1109/SEED51797.2021.00023. [Online]. Available: https://doi.org/10.1109/SEED51797.2021.00023.

[32] H. Jin, Z. He, and W. Qiang, « Specterminator: Blocking speculative side channels based on instruction classes on RISC-V », *ACM Trans. Archit. Code Optim.*, vol. 20, *1*, 15:1–15:26, 2023. DOI: 10.1145/3566053. [Online]. Available: https://doi.org/10.1145/3566053.

[33] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. B. Abu-Ghazaleh, « Safespec: Banishing the spectre of a meltdown with leakage-free speculation », *in Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, ACM, 2019, p. 60. DOI: 10.1145/3316781.3317903. [Online]. Available: https://doi.org/10.1145/3316781.3317903.

[34] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas, and J. S. Emer, « DAWG: A defense against cache timing attacks in speculative execution processors », *in 51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, IEEE Computer Society, 2018, pp. 974–987. DOI: 10.1109/MICRO.2018.00083. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00083.

[35] P. Kocher, J. Horn, A. Fogh, *et al.*, « Spectre attacks: Exploiting speculative execution », *in 2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[36] P. C. Kocher, « Timing attacks on implementations of diffie–hellman, rsa, dss, and other systems », *in Proceedings of CRYPTO '96*, Also available as cryptography/9611025, Springer (LNCS), 1996.

[37] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, « Spectre returns! speculation attacks using the return stack buffer », *IEEE Design Test*, vol. 41, *2*, pp. 47–55, 2024. DOI: 10.1109/MDAT.2024.3352537.

[38] M. Larabel, *Linux retpoline benchmarks*, Accessed: 2023-10-01, 2018. [Online]. Available: https://www.phoronix.com/review/linux-retpoline-benchmarks.

[39] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, « Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks », *in 25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, IEEE, 2019, pp. 264–276. DOI: 10.1109/HPCA.2019.00043. [Online]. Available: https://doi.org/10.1109/HPCA.2019.00043.

[40] M. Lipp, M. Schwarz, D. Gruss, *et al.*, « Meltdown: Reading kernel memory from user space », *in 27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[41] K. Loughlin, I. Neal, J. Ma, *et al.*, « DOLMA: securing speculation with the principle of transient non-observability », *in 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. D. Bailey and R. Greenstadt, Eds., USENIX Association, 2021, pp. 1397–1414. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/loughlin.

[42] G. Maisuradze and C. Rossow, « Ret2spec: Speculative execution using return stack buffers », *in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 2109–2122, ISBN: 9781450356930. DOI: 10.1145/3243734.3243761. [Online]. Available: https://doi.org/10.1145/3243734.3243761.

[43] A. Mambretti, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, and A. Kurmus, « Two methods for exploiting speculative control flow hijacks », *in 13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA: USENIX Association, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/woot19/presentation/mambretti.

[44] D. S. McFarlin, C. Tucker, and C. B. Zilles, « Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism? », *in Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, V. Sarkar and R. Bodík, Eds., ACM, 2013, pp. 241–252. DOI: 10.1145/2451116.2451143. [Online]. Available: https://doi.org/10.1145/2451116.2451143.

[45] R. McIlroy, J. Sevcík, T. Tebbi, B. L. Titzer, and T. Verwaest, « Spectre is here to stay: An analysis of side-channels and speculative execution », *arXiv preprint arXiv:1902.05178*, 2019. [Online]. Available: https://arxiv.org/abs/1902.05178.

[46] A. Milburn, K. Sun, and H. Kawakami, « You cannot always win the race: Analyzing the lfence/jmp mitigation for branch target injection », *arXiv preprint arXiv:2203.04277*, 2022. [Online]. Available: https://arxiv.org/abs/2203.04277.

[47] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer, « You shall not bypass: Employing data dependencies to prevent bounds check bypass », *CoRR*, vol. abs/1805.08506, 2018. arXiv: 1805.08506. [Online]. Available: http://arxiv.org/abs/1805.08506.

[48] H. Omar and O. Khan, « IRONHIDE: A secure multicore that efficiently mitigates microarchitecture state attacks for interactive applications », *in IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*, IEEE, 2020, pp. 111–122. DOI: 10.1109/HPCA47549.2020.00019. [Online]. Available: https://doi.org/10.1109/HPCA47549.2020.00019.

[49] O. Osvik, A. Shamir, and E. Tromer, « Cache attacks and countermeasures: The case of aes », *in Proceedings of the 6th International Conference on Cryptographic Hardware and Embedded Systems (CHES 2006)*, Introduces Prime+Probe style cache attacks and countermeasures, Springer (LNCS), 2006.

[50] M. Patrignani and M. Guarnieri, « Exorcising spectres with secure compilers », *in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 445–461, ISBN: 9781450384544. DOI: 10.1145/3460120.3484534. [Online]. Available: https://doi.org/10.1145/3460120.3484534.

[51] C. Percival, *Cache missing for fun and profit*, 2005.

[52] Phoronix, *Intel cxl r lvi benchmarking*, https://www.phoronix.com/review/intel-cxlr-lvi, Accessed: 2024-10-25.

[53] A. Randal, « This is how you lose the transient execution war », *CoRR*, vol. abs/2309.03376, 2023. DOI: 10.48550/ARXIV.2309.03376. arXiv: 2309.03376. [Online]. Available: https://doi.org/10.48550/arXiv.2309.03376.

[54] A. Randal, « Transient execution vulnerabilities in the security context of server hardware », University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-992, Dec. 2023, Accessed: 2025-09-24. DOI: 10.48456/tr-992. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-992.pdf.

[55] RISC-V Foundation, *The risc-v instruction set manual, volume i: User-level isa*, Version 2.2, October 2019, 2019. [Online]. Available: https://riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications.

[56] S. Rüegge, J. Wikner, and K. Razavi, « Branch privilege injection: Compromising spectre v2 hardware via predictor race conditions », *in Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2025, pp. -–-. [Online]. Available: https://comsec.ethz.ch/wp-content/files/bprc_sec25.pdf.

[57] G. Saileshwar and M. K. Qureshi, « Cleanupspec: An "undo" approach to safe speculation », *in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, ACM, 2019, pp. 73–86. DOI: 10.1145/3352460.3358314. [Online]. Available: https://doi.org/10.1145/3352460.3358314.

[58] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, « Efficient invisible speculative execution through selective delay and value prediction », *in Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, S. B. Manne, H. C. Hunter, and E. R. Altman, Eds., ACM, 2019, pp. 723–735. DOI: 10.1145/3307650.3322216. [Online]. Available: https://doi.org/10.1145/3307650.3322216.

[59] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, « Understanding selective delay as a method for efficient secure speculative execution », *IEEE Transactions on Computers*, vol. 69, *11*, pp. 1584–1595, 2020. DOI: 10.1109/TC.2020.3014456.

[60] S. van Schaik, A. Milburn, S. Österlund, *et al.*, « RIDL: rogue in-flight data load », *in 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, IEEE, 2019, pp. 88–105. DOI: 10.1109/SP.2019.00087. [Online]. Available: https://doi.org/10.1109/SP.2019.00087.

[61] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, « Context: A generic approach for mitigating spectre », *in 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/context-a-generic-approach-for-mitigating-spectre/.

[62] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, *Netspectre: Read arbitrary memory over network*, 2018. arXiv: 1807.10535 [cs.CR]. [Online]. Available: https://arxiv.org/abs/1807.10535.

[63] B. Semal, K. Markantonakis, R. N. Akram, and J. Kalbantner, « Leaky controller: Cross-vm memory controller covert channel on multi-core systems », *in ICT Systems Security and Privacy Protection - 35th IFIP TC 11 International Conference, SEC 2020, Maribor, Slovenia, September 21-23, 2020, Proceedings*, M. Hölbl, K. Rannenberg, and T. Welzer, Eds., ser. IFIP Advances in Information and Communication Technology, vol. 580, Springer, 2020, pp. 3–16. DOI: 10.1007/978-3-030-58201-2_1. [Online]. Available: https://doi.org/10.1007/978-3-030-58201-2%5C_1.

[64] Y. Shin, H. C. Kim, D. Kwon, J. Jeong, and J. Hur, « Unveiling hardware-based data prefetcher, a hidden source of information leakage », *in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds., ACM, 2018, pp. 131–145. DOI: 10.1145/3243734.3243736. [Online]. Available: https://doi.org/10.1145/3243734.3243736.

[65] J. Silc, T. Ungerer, and B. Robic, « Dynamic branch prediction and control speculation », *Int. J. High Performance Systems Architecture Int. J. High Performance Systems Architecture*, vol. 1, pp. 2–13, Jan. 2007. DOI: 10.1504/IJHPSA.2007.013287.

[66] W. Snyder, *Verilator: Open-source systemverilog simulator and lint system*, https://www.veripool.org/verilator/, Accessed: 2025-10-03, 2025. [Online]. Available: https://www.veripool.org/verilator/.

[67] SpinalHDL, *Naxriscv: An out-of-order risc-v cpu core*, https://github.com/SpinalHDL/NaxRiscv, Accessed: 2024-07-11, 2024.

[68] M. Taram, A. Venkat, and D. M. Tullsen, « Mitigating speculative execution attacks via context-sensitive fencing », *IEEE Des. Test*, vol. 39, *4*, pp. 49–57, 2022. DOI: 10.1109/MDAT.2022.3152633. [Online]. Available: https://doi.org/10.1109/MDAT.2022.3152633.

[69] G. C. Team, *Protecting our google cloud customers from new vulnerabilities without impacting performance*, Google Cloud Blog, 2018. [Online]. Available: https://cloud.google.com/blog/topics/inside-google-cloud/protecting-our-google-cloud-customers-new-vulnerabilities-without-impacting-performance.

[70] R. M. Tomasulo, « An efficient algorithm for exploiting multiple arithmetic units », *IBM Journal of Research and Development*, vol. 11, *1*, pp. 25–33, 1967. DOI: 10.1147/rd.111.0025.

[71] K. Tran, C. Sakalis, M. Själander, A. Ros, S. Kaxiras, and A. Jimborean, « Clearing the shadows: Recovering lost performance for invisible speculative execution through HW/SW co-design », *in PACT '20: International Conference on Parallel Architectures and Compilation Techniques, Virtual Event, GA, USA, October 3-7, 2020*, V. Sarkar and H. Kim, Eds., ACM, 2020, pp. 241–254. DOI: 10.1145/3410463.3414640. [Online]. Available: https://doi.org/10.1145/3410463.3414640.

[72] E. Tromer, D. A. Osvik, and A. Shamir, « Efficient cache attacks on aes, and countermeasures », *in Journal of Cryptology / Proceedings (various versions exist)*, Follow-up work refining cache attacks and mitigation discussion, 2010.

[73] J. Van Bulck, D. Moghimi, M. Schwarz, *et al.*, « Lvi: Hijacking transient execution through microarchitectural load value injection », *in 2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 54–72. DOI: 10.1109/SP40000.2020.00089.

[74] M. Vassena, C. Disselkoen, K. von Gleissenthall, *et al.*, « Automatically eliminating speculative leaks from cryptographic code with blade », *Proc. ACM Program. Lang.*, vol. 5, *POPL*, pp. 1–30, 2021. DOI: 10.1145/3434330. [Online]. Available: https://doi.org/10.1145/3434330.

[75] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, « Oo7: Low-overhead defense against spectre attacks via program analysis », *IEEE Transactions on Software Engineering*, vol. 47, *11*, pp. 2504–2519, 2021. DOI: 10.1109/TSE.2019.2953709.

[76] R. N. M. Watson, S. W. Moore, P. Sewell, and P. G. Neumann, « An introduction to cheri », University of Cambridge, Computer Laboratory, Technical Report UCAM-CL-TR-941, Sep. 2019. DOI: 10.48456/tr-941. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf.

[77] R. N. M. Watson, P. G. Neumann, J. Woodruff, *et al.*, « CHERI: A hybrid capability-system architecture for scalable software compartmentalization », *in Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, IEEE, 2015, pp. 20–37. DOI: 10.1109/SP.2015.9. [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/.

[78] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, « NDA: preventing speculative execution attacks at their source », *in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, ACM, 2019, pp. 572–586. DOI: 10.1145/3352460.3358306. [Online]. Available: https://doi.org/10.1145/3352460.3358306.

[79] S. Wiebing and C. Giuffrida, « Training solo: On the limitations of domain isolation against spectre-v2 attacks », *in 2025 IEEE Symposium on Security and Privacy (SP)*, 2025, pp. 3599–3616. DOI: 10.1109/SP61157.2025.00253.

[80] J. Wikner and K. Razavi, « RETBLEED: Arbitrary speculative code execution with return instructions », *in 31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 3825–3842, ISBN: 978-1-939133-31-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/wikner.

[81] J. Wikner and K. Razavi, « Breaking the barrier: Post-barrier spectre attacks », *in 2025 IEEE Symposium on Security and Privacy (SP)*, 2025, pp. 3516–3533. DOI: 10.1109/SP61157.2025.00089.

[82] N. Wistoff, M. Schneider, F. K. Gürkaynak, L. Benini, and G. Heiser, « Microarchitectural timing channels and their prevention on an open-source 64-bit RISC-V core », *in Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*, IEEE, 2021, pp. 627–632. DOI: 10.23919/DATE51398.2021.9474214. [Online]. Available: https://doi.org/10.23919/DATE51398.2021.9474214.

[83] J. Woodruff, R. N. Watson, D. Chisnall, *et al.*, « The cheri capability model: Revisiting risc in an age of risk », *in Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 457–468, ISBN: 9781479943944.

[84] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, « Invisispec: Making speculative execution invisible in the cache hierarchy », *in 51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, IEEE Computer Society, 2018, pp. 428–441. DOI: 10.1109/MICRO.2018.00042. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00042.

[85] Y. Yarom and K. Falkner, « Flush+reload: A high resolution, low noise, l3 cache side-channel attack », *in Proceedings of the 23rd USENIX Security Symposium (USENIX Security 2014)*, USENIX Association, 2014.

[86] T.-Y. Yeh and Y. N. Patt, « Alternative implementations of two-level adaptive branch prediction », *SIGARCH Comput. Archit. News*, vol. 20, *2*, pp. 124–134, Apr. 1992, ISSN: 0163-5964. DOI: 10.1145/146628.139709. [Online]. Available: https://doi.org/10.1145/146628.139709.

[87] 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 », *IEEE Micro*, vol. 40, *3*, pp. 81–90, 2020. DOI: 10.1109/MM.2020.2985359. [Online]. Available: https://doi.org/10.1109/MM.2020.2985359.

[88] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom, « Ultimate SLH: Taking speculative load hardening to the next level », *in 32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 7125–7142, ISBN: 978-1-939133-37-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-slh.

[89] Z. N. Zhao, H. Ji, M. Yan, *et al.*, « Speculation invariance (invarspec): Faster safe execution through program analysis », *in 53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, IEEE, 2020, pp. 1138–1152. DOI: 10.1109/MICRO50266.2020.00094. [Online]. Available: https://doi.org/10.1109/MICRO50266.2020.00094.

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : Assurer la confidentialité dans les cœurs Out-of-Order.

Mot clés : Spectre attaque, Canal caché, Exécution spéculative, `RISC-V` , Microarchitecture

Résumé : Dans la quête permanente d'une puissance de calcul plus rapide, les processeurs modernes utilisent des techniques permettant d'exploiter au maximum leurs ressources. Parmi ces techniques, l'exécution spéculative tente de prédire le résultat des instructions dont l'issue n'est pas encore connue, mais dont dépend la suite du programme. Cela permet au processeur d'éviter d'être inactif. Cependant, elle a ouvert une faille dans la microarchitecture: les mauvaises spéculations peuvent être exploitées pour accéder à des données sensibles, donnant naissance à la vulnérabilité **Spectre**. L'état de l'art propose diverses protections matérielles et logicielles. Les solutions matérielles sont généralement plus complètes, mais leur impact réel sur les performances reste débattu en raison des différences d'architecture et de méthodologie d'évaluation. Cette thèse vise à proposer une protection contre la vulnérabilité Spectre sur un cœur `RISC-V` . En commençant par évaluer les protections existantes, notamment la spéculation sélective, une approche largement déclinée en solutions logicielles et matérielles. Nous partons du principe que la microarchitecture est incapable de distinguer les données secrètes des données publiques dans un programme. Les résultats montrent que parvenir à une protection parfaite grâce à la spéculation sélective a un coût prohibitif en termes de performances. Face à ces limites, nous proposons une nouvelle solution qui fournit davantage d'informations à la microarchitecture sur les données sensibles.

Title: Ensuring confidentiality in modern Out-of-Order cores

Keywords: Spectre attack, Covert channel, Speculative execution, `RISC-V` , Microarchitecture

Abstract: In the continuous pursuit of faster computing, modern processors employ techniques to keep their resources as busy as possible. One such technique, speculative execution, predicts the outcome of instructions whose results are not yet available but are needed to determine the next steps of the program. This allows the processor to avoid being idle. However, it also introduced a flaw in the microarchitecture: mispredictions can be exploited to access sensitive data, giving rise to the **Spectre** vulnerability. The state-of-the-art offers various hardware and software protections. Hardware solutions are generally more comprehensive, but their real impact on performance remains debated due to differences in architecture and evaluation methodology. This thesis proposes a protection against Spectre on a `RISC-V` core. We first evaluate existing protections, particularly selective speculation, an approach widely explored in both hardware and software. We assume that the microarchitecture cannot distinguish between secret and public data within a program. The results show that achieving perfect protection through selective speculation comes at a prohibitive performance cost. To overcome these limits, we propose a new solution that provides the microarchitecture with additional information about sensitive data.