THÈSE DE DOCTORAT

CENTRALESUPÉLEC

ÉCOLE DOCTORALE Nº 601
Mathématiques, Télécommunications, Informatique,
Signal, Systèmes, Électronique
Spécialité : **Informatique**

Spécification et vérification formelle de mécanismes de sécurité pour l'architecture RISC-V

Par
Matthieu BATY

Thèse présentée et soutenue à **Rennes, le 10 décembre 2024**
Unité de recherche : **IRISA**
Thèse Nº : 2024CSUP0021

Rapporteurs avant soutenance :

Emmanuelle ENCRENAZ-TIPHENE	Full Professor, Sorbonne Université
Magnus MYREEN	Full Professor, Chalmers University of Technology

Composition du Jury :

Présidente :	Sandrine BLAZY	Full Professor, Université de Rennes
Examinateurs :	Pierre WILKE	Assistant Professor, CentraleSupélec
	Clément PIT-CLAUDEL	Assistant Professor, EPFL
Dir. de thèse :	Guillaume HIET	Full Professor, CentraleSupélec
Invités :	Alix TRIEU	Formal methods expert, ANSSI

Abstract

In this thesis, we consider the problem of hardware verification, with a focus on security properties of RISC-V processors. We target proofs at the microarchitectural level, reasoning directly on the hardware's definition. We follow an approach based on proof assistants, versatile tools used for producing high-confidence proofs. Their flexibility allows us to express and verify arbitrary properties on designs. This differs from the more rigid forms that formal verification commonly takes in the industry, where it tends to rely on specialized tools with set scopes. The flexibility of proof assistants comes at a cost. First, they are not specialized to the problem of hardware verification — all the domain-specific knowledge needs to be formalized before any verification work can proceed. Furthermore, they are complex tools with a prohibitive learning curve. In this work, we consider both of these concerns.

The first concern we outlined is that proof assistants must be taught about hardware design. Before anything else, the required notions must be defined within the system — critically, a formal semantics of a hardware description language must be given. In this thesis, we work both with an academic language built from the ground up with a formal semantics (Kôika) and a more industrial language whose semantics we had to formalize ourselves (FIRRTL).

The second concern is related to the complexity of proof assistants. Our answer consists of frameworks built around the semantics of hardware description languages. Both manual and automatic (chiefly SMT-based) options are explored. In principle, such frameworks can cover the same ground as formal tools used in the industry and some more.

We illustrate this methodology by formally verifying security properties of a RISC-V processor within the Coq proof assistant.

Keywords: formal methods • Coq • hardware verification • microarchitecture • RISC-V

Table of contents

Table of Figures

Remerciements (acknowledgment)

Une thèse, c'est à la fois très court et très long (surtout sur la fin). Ça serait plus long encore s'il s'agissait d'un travail solitaire, ce dont je ne peux pas me plaindre : j'ai eu le plaisir de passer ces années bien entouré, et qui plus est dans un environnement des plus agréables. Les quelques paragraphes qui suivent sont dédiés aux personnes qui ont rendu cette étape de ma vie plaisante et stimulante.

Tout a commencé par un stage à l'ANSSI, qui détonnait thématiquement avec mon parcours préalable : Thomas LETAN, Arnaud FONTAINE, merci d'avoir fait confiance à un étudiant un peu perdu et d'avoir guidé ses premiers pas dans les méthodes formelles. Merci encore de l'avoir aiguillé vers ce qui était alors l'équipe CIDRE, et merci enfin (à vous personnellement et à l'ANSSI en général) de lui avoir permis de mettre du pain sur la table pendant ces trois et quelques années à travers le financement de thèse que vous lui avez proposé.

Il n'y a pas de thèse sans équipe d'encadrement et la mienne était formidable. Merci à Guillaume HIET pour son style d'encadrement efficace et droit au but, et à Pierre WILKE pour sa disponibilité et sa productivité remarquable. J'ai beaucoup apprécié nos séances de développement communes, nos discussions et nos digressions. Côté ANSSI, je remercie Arnaud FONTAINE et Alix TRIEU pour leur suivi bienveillant et leurs perspectives précieuses.

Cette thèse m'a permis d'assister au réarmement démographique progressif d'un ~~open space~~ bureau paysager globalement déserté durant la période COVID, et je souhaite rendre hommage à sa population. Merci pour les références trop de fois recyclées, les randos, la coinche, le cidre, les CTFs, OxE5C474D3, les tours en vélo et les brousseries. Merci de m'avoir fait voir le baby-foot sous tous les angles, merci pour les mercredi pizzas, les projections secrètes, la décoration intérieure, les tentatives de rapprochement diplomatique avec les étages inférieurs, les jeudi midi jeux, la course à pied, Urban Terror, les bonhommes en talons de carnets de tickets repas, les chutes de cactus, la littérature de salle de pause et les énigmes nulles. J'ai hâte d'assister aux soutenances futures pour lever la main et vous poser des questions très compliquées.

Merci aussi aux permanents et autres adultes présents à l'étage pour éviter que la récré dégénère, ainsi qu'au personel de l'Hôpital Privé Sévigné pour leurs soins attentionnés lorsqu'elle dégénérait malgré tout.

Clément PIT-CLAUDEL m'a accueilli pour une césure de six mois à l'EPFL. Je le remercie tout particulièrement pour son encadrement de qualité au long de ces six mois denses en nouvelles expériences, mais je n'oublie pas les administrations de l'EPFL, de l'ANSSI et de l'Inria, ainsi que les membres de la FIPDAC qui ont œuvré pour rendre cette césure possible.

Cette thèse a été jugée par un jury présidé par Sandrine BLAZY, avec Clément PIT-CLAUDEL et Pierre WILKE comme examinateurs, et Emmanuelle ENCRENAZ-TIPHENE et Magnus MYREEN comme rapporteurs. Merci pour vos questions pertinentes et votre regard critique sur mes travaux. Je remercie plus particulièrement mes rapporteurs pour leur temps et leurs retours très justes.

J'ai également une pensée pour ma famille. Maman, papa, sans vous je ne serais probablement pas né. Merci de m'avoir épargné ce désagrément et merci plus globalement d'avoir toujours été présents, que ce soit à distance ou physiquement. Sophie, Angélique, je suis toujours heureux de constater que notre proximité transcende la géographie (Angélique, puisse ce manuscrit t'inspirer pour la fin de ton Masterarbeit). Erzsi mama, nagyon élveztem a Szekszárdon eltöltött időt az írás időszaka alatt. Mindent köszönök!

Enfin, je remercie toutes les personnes qui mériteraient mes remerciements, mais que j'ai oublié de citer. En lieu d'excuses, je leur dédie ce manuscrit.

Chapter recap

Merci !

Résumé substanciel en français

Contexte

Nos sociétés ont récemment été le théâtre d'une informatisation brusque et massive. Les aspects stratégiques ne sont pas en reste. Désormais, la gestion de transactions bancaires et de données médicales, la surveillance de réacteurs nucléaires, ou encore la communication privée des membres du gouvernement sont régies par des systèmes informatiques.

L'informatique est fragile. La moindre erreur de conception peut ouvrir la porte à des vulnérabilités, des failles qui sont autant d'armes dans l'arsenal d'attaquants que l'échelle des enjeux ne manque pas d'attirer. La recherche et la suppression de ces vulnérabilités est l'une des priorités de la sécurité informatique, qui occupe de fait une place croissante dans nos sociétés.

Les concepteurs de langages informatiques tiennent compte de ces aspects sécuritaires. Ainsi, la majorité des compilateurs modernes sont équipés de mécanismes tels que des vérificateurs de typage et de vérificateurs d'emprunt. Les garanties réduites qu'offrent ces outils peuvent être étendues par le biais de jeux de tests, dont le rôle est d'assurer que le comportement effectif d'un programme donné dans des circonstances précises correspond bien à celui escompté. Telle quelle, cette approche reste limitée : le champ des possibles est trop vaste pour être couvert par un jeu de tests réaliste.

Ces mécanismes ne sont en fait que la partie émergée des méthodes formelles, un domaine d'étude portant sur l'obtention de preuves à haut degré de fiabilité. Les méthodes formelles englobent un large éventail de techniques et d'outils, allant de vérificateurs automatiques pour des fragments simples de la logique aux assistants de preuves, des outils généralistes dont le rôle est de guider un utilisateur dans la construction de preuves de propositions arbitraires.

L'application des méthodes formelles à la validation de logiciel a démontré son efficacité : par le passé, des systèmes complexes tels que des compilateurs ou des micro-noyaux ont pu être vérifiés de manière exhaustive.

Toutefois, que le logiciel soit vérifié ou non, son exécution effective est nécessairement incarnée : les calculs qu'il requiert sont réalisés électroniquement, et le matériel qui réalise ces calculs est un assemblage délicat de composants hétérogènes (processeur, carte mère, périphériques, etc.). Chacun de ces composants expose une interface spécifiée de manière plus ou moins libre au reste du système. *In fine*, le logiciel dépend lui aussi de ces interfaces, à travers la vue abstraite que lui en donne le système. Par conséquent, les preuves portant sur des comportements logiciels reposent sur des hypothèses, souvent implicites, au sujet du comportement du matériel.

Les hypothèses matérielles sont cruciales, dans la mesure où la moindre erreur dans les fondations peut mettre en péril tout l'édifice. Elles le sont d'autant plus si l'on considère qu'il n'est pas possible d'apporter de correctifs au matériel, contrairement au logiciel. Les méthodes formelles s'intègrent dès la phase de conception du matériel et constituent une réponse adaptée à ces besoins de sécurité et de stabilité.

Malgré certaines différences, la vérification de logiciel et la vérification de matériel restent des sujets connexes. Tout comme pour la conception de logiciel, certaines formes limitées de vérification formelle sont intégrées à l'outillage usuel. Ainsi, les langages de description du matériel sont typés. Par ailleurs, il existe un certain nombre d'outils formels exclusifs au matériel. C'est par exemple le cas des vérificateurs d'équivalence, des outils visant à établir l'équivalence fonctionnelle de deux circuits (un tel outil peut notamment être utilisé pour valider une optimisation).

Bien que les méthodes actuellement utilisées dans l'industrie remplissent de façon satisfaisante les attentes placées en elles, elles restent rigides et donc limitantes. Au demeurant, les outils employés ne sont pas vérifiés formellement, ce qui restreint quelque peu la fiabilité de leurs résultats. La mise à contribution d'assistants de preuves permettrait de répondre à ces deux points, mais les approches basées sur ces outils se doivent de répondre à certains défis pour être viables, notamment sur des questions de performance et d'ergonomie.

Objectifs et contenu

Le contexte introduit ci-dessus mène naturellement à la question de recherche au centre de cette thèse.

Question de recherche

Comment vérifier formellement, avec un assistant de preuve, des implémentations de mécanismes de sécurité pour processeurs décrites au niveau des transferts de registres ?

Il y a principalement trois raisons pour lesquelles nous souhaitons que la vérification ait lieu dans un assistant de preuve :

1. Du fait de la base de confiance plus limitée sur laquelle ils reposent, les assistants de preuve peuvent être considérés comme plus fiables que d'autres types d'outils que l'on trouve dans l'industrie ;
2. En France, l'ANSSI, qui est l'autorité en charge de la certification de sécurité informatique Critères Communs [1], reconnaît uniquement certains outils comme l'assistant de preuves Coq pour EAL7, le degré de certification le plus élevé [2] ;
3. L'expressivité des assistants de preuve dépasse celle des outils plus spécialisés —– en particulier, ils peuvent introduire des abstractions pour contourner le problème d'explosion combinatoire des états [3].

Le principal défi de la vérification formelle basée sur des assistants de preuve est la performance du processus de vérification. Les méthodologies proposées ne doivent pas fonctionner uniquement sur des exemples simples mais doivent supporter pour des charges de travail réalistes. À cette fin, nous nous appuyons sur des méthodes automatiques utilisées dans l'industrie, telles que les solveurs SMT. Nous leur déléguons des objectifs de bas niveau, tout en continuant à nous appuyer sur l'expressivité des assistants de preuve pour construction de la structure de haut niveau de la preuve.

Nous concentrons nos efforts sur la vérification de la définition même de processeurs, et plus précisément de leur microarchitecture. Ainsi, nous évitons les écueils des raisonnements basés sur des modèles : il existe souvent un écart entre le modèle et l'objet qu'il représente.

Nous nous limitons à la vérification de propriétés portant sur la définition microarchitecturale du matériel. En particulier, nous ne traitons pas la vérification de la préservation de la sémantique au long de la chaîne de production qui conduit cette définition au silicium.

Contributions

Nos contributions peuvent être résumées comme suit :

a) Nous introduisons une infrastructure pour vérifier des propriétés arbitraires sur des circuits définis dans le langage de description du matériel Kôika —

la méthodologie de vérification est basée sur des applications séquentielles de transformations de code vérifiées, menant à une simplification progressive du circuit ;

b) Nous démontrons la viabilité de cette méthodologie en l'appliquant à la vérification formelle d'un mécanisme de sécurité matériel attaché à un processeur RISC-V pipeliné et synthétisable ;

c) Nous enrichissons cette méthodologie avec des procédures automatiques basées sur des solveurs SMT ;

d) Nous portons le langage de description du matériel FIRRTL au sein de l'assistant de preuve Coq sous le nom de COQQTL — en d'autres termes, nous formalisons sa sémantique .

Les points a) et b) correspondent aux travaux accepté pour présentation à la conférence CSF'23 [4]. Ils avaient précédemment fait l'objet d'une présentation au workshop SILM'22 [5]. Le point c) décrit une extension non-publiée de ces travaux. d) se rapporte à un projet en cours que nous entendons soumettre prochainement.

Résumé

Le corps de cette thèse est divisé en six chapitres rédigés en anglais.

Le Chapitre 1 introduit le sujet de cette thèse avant de résumer succinctement son contenu, selon le modèle établi par le chapitre courant.

Dans le Chapitre 2, nous posons les bases nécessaires à la compréhension de nos travaux. Plus concrètement, nous introduisons des notions relatives à l'architecture des ordinateurs, à la sécurité informatique, ainsi qu'aux méthodes formelles. Nous discutons brièvement de ce qui se trouve à l'intersection de ces trois domaines, avant de détailler le fonctionnement de Kôika, un langage de description du matériel à la sémantique formalisée.

Dans le Chapitre 3, nous dressons un état de l'art des domaines pertinents compte tenu de l'orientation de cette thèse. Nous positionnons par ailleurs nos travaux relativement à l'existant.

Le Chapitre 4 concerne nos travaux liés au langage Kôika précédemment mentionné. En particulier, nous décrivons comment nous avons étendu le projet Kôika pour le rendre adapté à la vérification matérielle. La méthodologie résultant de ces efforts est applicable à des problèmes très généraux de vérification de circuits. Nous démontrons ses capacités au travers de la vérification d'un mécanisme de sécurité intégré à un processeur RISC-V. Par la suite, nous expliquons comment nous avons enrichi cette méthodologie avec des procédures automatiques. Ces procédures se sont révélées très efficaces en pratique pour réduire la quantité d'effort manuel requise.

Dans le Chapitre 5, nous présentons COQQTL, notre port du langage FIRRTL dans l'assistant de preuve Coq. En comparaison avec Kôika, qui est un projet académique, FIRRTL a un certain nombre de caractéristiques qui le rendent adapté à l'industrie, où il a déjà été employé avec succès. En particulier, il permet la définition de modules, des composants réutilisables d'un circuit à l'autre. Cette notion est utile tant pour la conception de circuits que pour leur vérification, une preuve réalisée sur un module s'appliquant à chacune de ses instances.

Pour finir, le Chapitre 6 résume le travail effectué et propose un certain nombre de perspectives à plus ou moins long terme.

Chapter 1

Introduction

1.1 Context

We live in a time when software is omnipresent, for better or worse. From the handling of banking transactions and the management of medical data to the communication of heads of state and the surveillance of nuclear reactors, few critical sectors are not deeply impacted by it. This pervasiveness of software makes it a prime target for malevolent actors.

Software is fragile. Design oversights, programming mistakes, or even hostile intents may lead to defects, each an additional tool in the attackers' arsenal. Techniques for ensuring that software is defect-free are therefore of paramount importance.

Most programming languages include facilities for avoiding common classes of bugs, such as typing systems or borrow checkers. These basic checks can be complemented with test suites, ensuring that programs behave as expected in all explicitly given cases. The testing process usually cannot be exhaustive — the state space is too large.

These techniques are shadows of the much richer domain of formal methods, which is concerned with constructing high-assurance proofs. Formal methods encompass a wide gamut of tools and methods, ranging from fully automatic decision procedures for simple fragments of logic to proof assistants, tools assisting a user in constructing mechanically checked proofs for arbitrary properties.

In their full generality, formal methods can reach further than traditional tools: they open ways of generalizing, for instance by reducing huge state spaces to a manageable set of representatives. These tools make it possible to formally show that the behavior of a piece of software corresponds to its specification. Large software projects such as compilers or microkernels have been verified this way.

However, software does not run in a vacuum. There needs to be a physical basis for the computations — the hardware. In this complex heterogeneous system, each component exposes an interface to the others based on more or less formal contracts about its behavior. The actual complexity of the system is usually hidden from the programmer, who accesses it through abstractions, which come with a set of assumptions. That is, software can only ever be defect-free under the hypotheses it expects the hardware to fulfill. These hypotheses about hardware properties should better hold, as shaky foundations jeopardize the whole structure. Unlike in software, errors in hardware design cannot be patched away: once the physical device is produced, its design is fixed. The unalterable nature of hardware means that its verification is a critical problem.

Hardware is described through specialized languages. A piece of hardware starts as a bunch of text files before ending up as a physical implementation. There is a large gap between these text files and the physical world, which must be crossed at some point. Along the way, many things can go wrong: even if the initial definition is correct, applying a faulty optimization pass further down the line or mixing up two wires in the physical implementation would void any formal guarantee. Trusting hardware is as much trusting the initial definition as it is trusting the process that takes this definition down to a physical object.

Hardware verification is not entirely removed from software verification, yet subtle differences remain between these tasks. Just like in software, some limited forms of formal methods are commonly applied to hardware design. Some of these are the same as those used in software, such as type systems or testing facilities. Others are hardware exclusives. For instance, equivalence checkers are automatic tools for verifying that two hardware descriptions are equivalent. They come in handy for verifying that an optimization pass does not change the functional behavior of hardware, a risk we previously outlined. Another example is assertion-based verification, wherein formal expectations about the behavior can be inlined directly within the description. Checking procedures may be used for verifying the assertions, though sometimes only in a bounded way (it may, for instance, only check only the states reachable in the first x ticks).

Although the current manifestations of formal methods in hardware are efficient, they have their limits. There are whole families of problems that are effectively out of their reach. On the other hand, despite their flexibility, proof assistants are not commonly used for hardware verification. This absence results from the insufficient support from proof assistant for this task (there are no prominent ready-made, lowfriction libraries for importing and verifying circuits) and the complexity of these tools. This complexity is also a problem for software verification: building a verified program costs much more than building its unverified counterpart. These additional costs can be seen as the price to pay for security; however, they are so high that they are generally not seen as worth the benefits outside of some critical applications such as avionics or nuclear safety.

1.2 Objectives and contents

The context described above naturally leads to the question that underpins this thesis.

Research question

How to formally verify implementations of security mechanisms for processors at the register transfer level of description within a proof assistant?

There are three core reasons why we want verification to take place within a proof assistant:

1. Proof assistants have a limited Trusted Computing Base (TCB) compared with the more specialized tools found in the industry, making them more trustable;
2. ANSSI, the French certifying authority for the Common Criteria for Information Technology Security Evaluation [1], recognizes certain formal tools for the highest level of certification (EAL7) — the Coq proof assistant is among these tools [2];
3. Proof assistants are generalist systems with an expressiveness that goes beyond that of specialized tools — in particular, they can introduce arbitrary abstractions to work around the state explosion problem [3].

The main challenge of proof assistant-based formal verification is the performance of the verification process: getting a methodology that works on simple examples to scale to realistic workloads is non-trivial. To improve this state of affairs, we rely on automatic methods used in the industry, such as SMT-solvers. We dispatch them goals that are within their reach, while the expressiveness of proof assistants gives us fine control over the high-level structure of the proof.

We focus on microarchitectural verification, as reasoning on models leaves a distance that allows errors to creep in. More specifically, our work considers the direct verification of the very definitions of circuits, which ensures that this distance is kept at a minimum. We do not prove the whole compilation chain correct for the physical implementation: our focus is only on the higher strata.

1.3 Contributions

Our contributions can be summarized as follows:

a) We design a framework around the Kôika hardware description language to specify and implement security mechanisms. In addition, it makes it possible to define and verify security properties that the implementation is supposed to enforce — the methodology we implement relies on a series of verified code transformation passes that progressively simplify the designs;
b) We showcase this methodology by verifying a hardware-based security mechanism integrated into a synthesizable processor;
c) We add support for automatic procedures for handling most of the proof without the need for user intervention;
d) We port the FIRRTL language to the Coq proof assistant — i.e., we mechanize the semantics of this language.

a) and b) correspond to work that was published at the CSF'23 conference [4], an early version of which had previously been presented at the SILM'22 workshop [5]. c) is an unpublished extension of this research. d) describes ongoing work that we plan to submit shortly.

1.4 Outline

The remainder of this thesis is split into five chapters.

We begin by introducing background information in Chapter 2. The rest of the thesis builds upon this information. In particular, we introduce basic notions of computer architecture, cybersecurity, and formal methods. We briefly discuss their intersection in a dedicated section before considering a concrete example of a formal hardware description language.

Then, in Chapter 3, we explore the state of the art of relevant domains and position this thesis relative to it.

Chapter 4 is about our work involving Kôika, a hardware description language with a formal semantics. We describe a methodology for formally verifying arbitrary hardware properties, and we apply it to verifying a security mechanism on a pipelined RISC-V processor. Furthermore, we explain how we add support for a workflow that delegates most proofs to an automatic procedure. This makes verification less tedious than through the original, essentially manual process.

Chapter 5 presents our work porting the FIRRTL language to Coq as COQQTL. This language has some features that Kôika lacked to make it a serious contender for industrial hardware design. In particular, it supports the definition of reusable modules, which give a natural way of abstracting designs. Like in Kôika, we design a framework for reasoning about COQQTL designs. Although this framework looks quite similar to the one we built for Kôika, it follows a much more automation-heavy approach from the outset.

Finally, Chapter 6 concludes this thesis. It summarizes the work we described in the previous chapters and introduces a number of ideas for future work.

Chapter 2

Background

In this chapter, we introduce the ground concepts on top of which the rest of this thesis builds. This thesis is about enabling hardware designers to formally verify that security properties hold for their designs. As such, it draws upon ideas stemming from three distinct fields of computer science:

- Computer architecture
- Cybersecurity
- Formal methods

Chapter outline

We begin with an introduction to these three fields, starting with hardware architecture in Section 2.1, continuing with cybersecurity in Section 2.2, and finishing with formal methods in Section 2.3. We explore their intersection in more detail in Section 2.4. We finish by presenting Kôika, a formal language for designing hardware, in Section 2.5. We adapted this language for our use during this thesis, as discussed in Chapter 4.

2.1 Computer architecture

Computer architecture is the branch of computer science that focuses on the design and implementation of computing machines. In this thesis, we focus more specifically on processors.

Section outline

This section introduces all relevant computer architecture-related concepts. It opens with a brief outline of the history of computer architecture in Subsection 2.1.1 before delving into Hardware Description Languages (HDLs) in Subsection 2.1.2. Finally, Subsection 2.1.3 turns to Instruction Set Architectures (ISAs).

2.1.1 A brief history

The first computer widely recognized as Turing-complete (i.e., expressive enough) was the ENIAC [6], built in 1948. Unlike some other early computing machines, the ENIAC was purely electronic, having abandoned the (electromechanical) relay in favor of the faster (electronic) vacuum tube. Early computers were designed and assembled manually. They were costly and bulky objects, which limited their use to projects funded by governments or large companies.

The following two decades were marked by significant innovations, such as transistors replacing vacuum tubes or the development of integrated circuits. These technologies allowed processors to grow more powerful, compact, affordable, and energy-efficient, eventually leading to the widespread adoption of the computer. These breakthroughs culminated in the advent of the microprocessor, with the Intel 4004 being released in 1971: over twenty-odd years, processors had gone from behemoths taking up whole rooms to something that could be held on a finger.

The rise of the microprocessor also signified the end of the human-sized era of computer electronics, with transistors reaching a size of 10µm. At this point, processors could no longer be built by hand and started relying on a photolithographic process. The photolithographic masks for early microprocessors, such as the Intel 4004, were still produced manually; however, this soon became impractical. The placement and routing of components and circuitry were delegated to increasingly automatic tools.

It was only during the 1980s that Electronics Design Automation (EDA) became pervasive: hardware could now be designed and simulated entirely digitally before being sent to production. Combined with the growing affordability of computers, this ushered in an era of accessible hardware design.

During these times, HDLs became dominant, with Verilog being released in 1984 [7]. Through these languages, the functionality of hardware could be described independently of implementation concerns such as placement and routing. At first, these languages were exclusively used for modeling and simulation. After a few years, they started being used as "silicon compilers" able to turn (relatively) high-level descriptions of circuits into netlists that could be used to generate the actual hardware.

Not only did the 1980s see the rise of EDA and HDLs, but it was also during this decade that Field-Programmable Gate Arrays (FPGAs), reconfigurable hardware allowing for efficient emulation of arbitrary circuits, made their apparition. As FPGAs could be configured using the same HDL-based descriptions as the actual designs to be

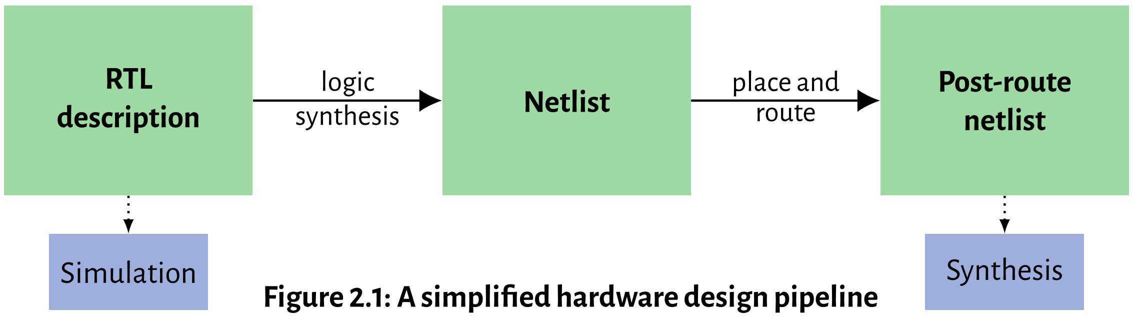


Figure 2.1: A simplified hardware design pipeline

put into production, the testing and debugging phase of hardware development could now proceed at a fraction of the previous cost, further contributing to its accessibility.

Although the overall process of hardware production has since been refined, the fundamentals of hardware design are mostly unchanged, and many industrial tools in use today date back to the 1980s. Most historical synthesis tools are proprietary, although some open-source toolchains such as SIS [8] and the related VIS [9] appeared as early as the 1990s. Modern takes on open-source synthesis toolchains such as Yosys [10] make it possible to produce a Verilog processor and get it onto an FPGA using only open-source technology.

2.1.2 Hardware Description Languages

As its name implies, an HDL is a language made to describe electronic circuits. HDLs are an integral part of modern hardware development, as they describe a circuit's behavior in isolation from low-level concerns such as placement and routing. They also help make hardware design modular: components can be easily packaged for reuse in other designs.

There are several types of hardware descriptions, the main ones being structural, behavioral, and dataflow-based. Structural descriptions are formulated in terms of basic building blocks such as gates and wires. They are often represented as Register Transfer Level (RTL) descriptions, where systems are depicted as a collection of registers and rules that govern how these registers are updated. In contrast, behavioral descriptions focus on the system's functionality, leaving the task of generating the components to the compiler. Dataflow-level descriptions, on the other hand, emphasize the flow of data between components, detailing how data moves through the system and how operations are performed on that data. Many languages incorporate multiple paradigms, allowing designers to choose the most suitable approach for their needs.

As illustrated in Figure 2.1, modern hardware development usually starts from an RTL description. This description can be used directly for simulation or synthesis (leading to FPGA-based testing or actual production). To synthesize a design, we acquire a netlist, a description of all components to be implemented. At this stage, the netlist is a graph devoid of information regarding physical layout. Only in the next phase are the target's specificities considered. For instance, different FPGAs may require different layouts. A physical layout is proposed by the place and route procedure, resulting in a post-route netlist that can be synthesized directly.

In the rest of this subsection, we provide an overview of HDLs at large, going from the classical, historical HDLs that dominate the industry to relative newcomers with new ideas regarding how hardware could be constructed.

Classical HDLs Verilog [11] and VHDL [12] are two classical HDLs that originated in the 1980s and continue to play a significant role in the industry today. VHDL is more closely related to the Ada programming language, as it was initially developed as a documentation language for the U.S. Department of Defense, while Verilog has more in common with C. For example, VHDL is strongly typed, whereas Verilog is weakly typed. Despite these differences, both languages adopt similar approaches: they allow designers to describe electronic systems at various levels of abstraction.

In 2009, Verilog's standard was merged with that of SystemVerilog [13], which started as an extension of this language, bringing facilities for verification (such as assertions or clocking domains) as well as features for higher-level description (such as C types, unions or casting) to the language.

High Level Synthesis High-Level Synthesis (HLS) is a design process where high-level programming languages such as C or C++ are used to generate hardware designs in languages like (System)Verilog or VHDL. The high level of abstraction of this process improves its expressiveness to the benefit of productivity. Furthermore, if the compiler is trusted, the verification can proceed directly on the software description of hardware.

SystemC [14] is arguably the most popular framework for HLS. It is not a language per se but a collection of C++ classes; compilers can generate HDL code from SystemC. Direct generation of hardware from C/C++ code is also possible through tools such as LegUp [15]. This approach has some traction in the industry, as illustrated by the fact that both Intel [16] and AMD [17] offer tools for HLS.

Modern HDLs Modern alternatives to these classical languages have been developed but have yet to achieve the same level of presence in the industry. The hardware industry has significant inertia due to several factors, including the extensive investment in existing tools and workflows and the amount of legacy code relying on established languages. Most modern languages use (System)Verilog or VHDL as a backend, guaranteeing a base level of compatibility with the established tooling.

Recently, LLVM Circt [18] was introduced as an open-source compiler leveraging the LLVM methodology for circuit compilation. This experimental tool provides a framework that accommodates various circuit representations and transformations. LLVM Circt can reduce the implementation cost of new HDLs.

Some modern HDLs are advertised as Hardware Construction Languages (HCLs) due to their emphasis on high-level, programmatic descriptions. For instance, Chisel [19] is an HCL embedded within the Scala programming language. It leverages the functional programming capabilities of Scala to make hardware design more expressive. The related SpinalHDL [20] iterates on Chisel, attempting to fix perceived flaws within this language. Other comparable HDLs include the Haskell-based Clash [21], [22] and Lava [23], the OCaml-based Hardcaml [24], and the Python-based MyHDL [25] and Magma [26].

Rule-based HDLs Rule-based HDLs are a class of languages centered on rules, a set of building blocks expressed sequentially but executed concurrently. Each rule may or may not run on each given cycle. The compiler synthesizes the scheduler that picks which rules run on each cycle in such a way as to maximize work done while avoiding incompatible actions. Bluespec [27] is the primary representative of this class of languages.

A more in-depth description of this class of languages is deferred to a later section, where we go through an in-depth presentation of a formal HDL of this kind (Section 2.5.1).

2.1.3 Instruction Set Architectures

Compilation is the process through which a program written in a programming language is translated to machine language, the underlying language of a given processor. Compiling programs can be long and tedious. Therefore, software is usually shipped in binary form. A program compiled for a specific processor will not run on all processors, as distinct models may not include the same instructions or encode them in the same way, they may have different sets of registers altogether, etc.

Building a binary for each different processor model would be impractical, and interoperability of different machines would be hampered. Practicality concerns led to vendors coalescing around a limited set of processor interfaces. Although these interfaces characterize the functional behavior of machine code, they leave total flexibility regarding the actual implementation. A processor could implement such an interface independently of being sequential or pipelined, scalar or superscalar, in-order or out-of-order, supporting speculative execution or not, having different sizes of cache, etc.

Processor interfaces are called ISAs. In contrast, concrete implementations are called microarchitectures.

Rather than examining one of the currently commercially successful ISAs, such as x86 [28] or Arm [29], we focus on the RISC-V ISA [30], a relative newcomer with a negligible market share but a promising future. The most important thing about RISC-V is not its architecture but the fact that it is an open-standard ISA. Unlike x86 and Arm, anyone can implement RISC-V without paying for a license. This openness alone is a great boon for amateurs and commercial implementers alike.

As its name implies, RISC-V is an ISA of the RISC tradition (Reduced Instruction Set Computer). As such, it is based on a minimal core. The simplicity of this core could be a liability. Although this makes the architecture well-suited to simple embedded systems, the lack of specialized instructions is significantly limiting in most other situations — some minor added complexity in the ISA may result in clear benefits on the side of performance. To keep the best of both worlds, RISC-V has a notion of extensions. The base standard includes only the strict minimum of features. Extensions can be implemented to extend the feature set. For instance, there are extensions for adding vector instructions or atomic instructions. There is also a privileged version of the RISC-V ISA [31], adding features such as different levels of privilege, interrupts, and hardware threads. These facilities are required to build actual operating systems.

The notion of extensions is not a specificity of the RISC-V ISA. For instance, both x86 and Arm include extensions for vector operations. Such extensions incur a risk of fragmenting the ISA. This risk is mitigated in several ways. First, extensions are not freely picked and chosen. There are some packages of extensions that certain classes of processors are expected to include. For instance, x86 extensions are consistently implemented by all processors produced after their introduction (although extensions sometimes get deprecated). Similarly, desktop-class processors are expected to implement a family of RISC-V extensions. Programs can also check whether an instruction is supported before using it and include appropriate fallbacks.

RISC-V's characteristics make it a perfect candidate for research purposes. Starting a RISC-V-based project is relatively easy, legally and practically speaking: tens of open RISC-V processor designs can be found online, and the simplicity of the base version of the standard makes it easy to work with as well as a good fit for education and students onboarding. Doing the same for x86 would be a different story. Overall, RISC-V is in a uniquely favorable position for future success.

2.2 Cybersecurity

This section introduces some cybersecurity notions that will feature throughout this document. Cybersecurity focuses on upholding critical properties of computer systems:

- **Confidentiality**: access to sensitive data must be restricted;
- **Integrity**: data must not be subject to undesired modifications;
- **Availability**: the system should remain operational.

Cyberattacks exploit hardware, software, or even social vulnerabilities for violating the properties listed above. There exist two broad family of approaches to cybersecurity:

- **Preventive approaches** are concerned with the construction of defect-free systems;
- **Reactive approaches** are concerned with the construction of fail-safe systems or the detection of abnormal situations.

In this thesis, we focus on the former approach. We formally verify that properties hold for designs before synthesizing them. Indeed, prevention is preferable whenever feasible. One of our key objectives is to explore ways of guaranteeing that hardware is safe from broad classes of vulnerabilities: removing vulnerabilities means depriving attackers of precious tools. Formal methods help us guarantee that systems are genuinely defect-free.

Section outline

We start with Subsection 2.2.1, a brief overview of the history of cybersecurity, before considering a concrete vulnerability (buffer overflows) and an adapted countermeasure (shadow stacks) in Subsection 2.2.2. Subsection 2.2.3 then turns to the notion of TCBs. Finally, Subsection 2.2.4 presents the Common Criteria for Information Technology Security Evaluation (CC), an international standard for computer security certification.

2.2.1 A brief history

Computer security is a relatively recent domain, which only took off with the introduction of the Internet. Prior to that, computers were both less numerous and more isolated. Therefore, they were less interesting and less reachable targets, although they were pretty unprotected: systems lacked primary security mechanisms. In a world without much in the way of security incentives, vulnerabilities abounded.

The first large-scale cybersecurity incident was the Morris worm, developed and released in the wild in 1988 [32]. It is estimated that about 6'000 out of the approximately 60'000 computers connected to the Internet were impacted, resulting in a temporary network partition. In reaction to this event, the first of many Computer Emergency Response Teams (CERT) was founded. The earliest commercial antivirus software was released just one year before this event, fueled by the rise of viruses (usually transmitted by floppy disks). The inadequate state of cybersecurity ensured a period of healthy growth in this market.

Cybersecurity matured during the 1990s and early 2000s, but so did cybercriminality. Organized actors, sometimes state-funded, sought to exploit vulnerabilities for monetary or political reasons. Throughout the 2010s, a string of high-profile attacks raised public awareness of the stakes of cybersecurity, such as WannaCry (2017), a ransomware that hit the industry and hospitals alike, or the Equifax data breach (2017), wherein the private data of close to 150 million persons were compromised. In parallel to this, the discovery of the Spectre [33] (CVE–2017–5753 and CVE–2017–5715) and Meltdown [34] (CVE–2017–5754) vulnerabilities, published in 2018, showed that most hardware released in the prior decades was flawed.

Not everything in cybersecurity has to do with malicious intents: just a few months ago, in July 2024, a faulty update was pushed by the Crowdstrike cybersecurity company. The impacted devices were left unable to boot, impacting industries worldwide for an estimated loss of billions of euros. This feat goes beyond what most successful attacks achieve.

After decades of relentless growth, computing has become a keystone of our societies. Alas, this foundation is not as solid and stable as we may wish. In this context, the role of cybersecurity is indeed a central one.

2.2.2 Buffer overflows and shadow stacks

Memory corruption is still one of the most significant vulnerabilities in software developed in low-level languages like C or C++. Indeed, developers using such languages are in charge of the application memory management, which can lead to spatial and temporal memory safety errors. Attackers can exploit such vulnerabilities to leak confidential data or modify the application's intended behavior.

Subsection outline

In Subsection 2.2.2.a, we introduce buffer overflows, one of the most commonly exploited vulnerabilities for memory corruption attacks. Then, in Subsection 2.2.2.b, we turn to shadow stacks, a hardware-based countermeasure targeting this vulnerability.

2.2.2.a Buffer overflows

A *buffer overflow* is a vulnerability that allows writes to reach past the end of a buffer, overwriting the subsequent memory locations in the process. When the buffer is on the stack, this opens the door to the modification of return addresses of procedures; the new address may point to arbitrary code, which gets executed.

Figure 2.2 shows a C program exhibiting a trivial buffer overflow. Buffer `buf` in function `f` is 16-byte long, and the `strcpy` function performs no bounds checking before copying the buffer `attack_buf`, which is 24-byte long. If attackers are aware of the program's memory layout, they can carefully craft a specific input that leads to an overwrite of the vulnerable function's return address.

In 2023, this vulnerability made the top of MITRE's most dangerous software weaknesses list [35].

```
void bad(){%
  puts("Bad!\n");
}

int f(char* s){%
  char buf[16];
  strcpy(buf,s);
}

int main(int argc, char* argv[]){%
  int attack_buf[6];
  attack_buf[5] = (intptr_t)&bad;
  f((char*)attack_buf);
}
```

Figure 2.2: A program vulnerable to buffer overflows
This program overwrites its return address

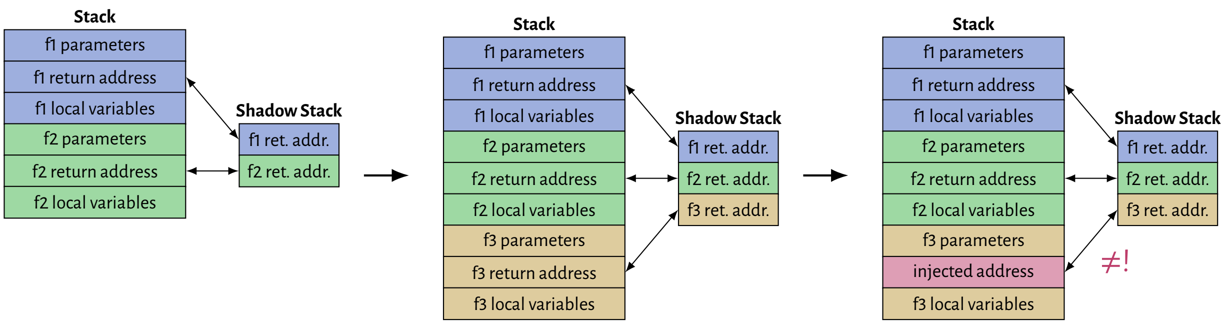


Figure 2.3: Working principle of shadow stacks
A shadow stack implicitly stores a copy of the return address on every function call and keeps it until the function returns. There is no other way to manipulate it. Whenever a function returns, the return address in the classical stack can be compared to the one in the shadow stack. If they differ, then the classical return address has been modified.

2.2.2.b Shadow stacks

There are both compiler and hardware-based countermeasures for buffer overflow vulnerability. For instance, arbitrary values called canaries may be inserted by compilers before return addresses in the stack. Considering the location of the canaries, naive attempt to overwrite return addresses would also affect them (although this approach is not foolproof [36]). The compiler inserts code to check whether return addresses have been modified.

Hardware-based solutions offer some advantages over purely compiler-based ones. In particular, they usually incur lower delays, if any, due to the parallel nature of hardware. An example of such a hardware-based security mechanism is Control-

Flow Enforcement Technology [37] (CET), a solution introduced by Intel for its 11th generation of processors.

The shadow stack is a critical component of the CET. Its working principle is illustrated in Figure 2.3. Buffer overflow resilience is achieved by storing return addresses redundantly in a space that cannot be written to through normal means. In Intel's CET, the only way through which pages marked as shadow stack pages can be altered is when an instruction corresponding to a function call or return is detected. In this situation, the stack is implicitly pushed to or popped from. Otherwise, the write fails: there is no way of modifying these pages explicitly.

When a function return is detected, the shadow and standard stacks are compared. If they disagree on the return address of a procedure, then the standard stack is compromised, and an exception gets raised — it is left to the system to handle. Return addresses are always checked before jumping to them.

2.2.3 Trusted Computing Bases

The notion of TCBs is used in cybersecurity to outline limitations to security guarantees in a system. The TCB of a system encompasses all hardware and software components whose correction is critical to its security. For instance, if a piece of software assumes that a specific external function is implemented correctly, then this function becomes part of the software's TCB. Bugs or vulnerabilities in a TCB can break everything that relies on it.

Another view of TCBs is that they are a set of logical hypotheses on which the proof of the security of a system relies. The simpler the TCB, the better the security: reduced hypotheses can be falsified in fewer ways.

2.2.4 Common Criteria and Evaluation Assurance Levels

CC [1] refers to an international standard for computer security certification. This standard introduces a way of describing security properties alongside criteria for evaluating adherence to these properties. As evaluation can be a lengthy and costly process, and different systems do not warrant the same level of guarantees, the standard describes seven Evaluation Assurance Levels (EALs). EALs indicate the rigor of the evaluation applied to a system. Each level increases the stringency of the process, with EAL7, the highest level requiring proof assistant-based formal proofs of claimed properties. In practice, this level is rarely reached: proof assistants are complex tools and verifying commodity systems within them remains challenging. One of the objectives of this thesis is to make the construction of hardware that meets the most stringent common criteria requirements more accessible.

2.3 Formal methods and proof assistants

In this section, we introduce the notions related to formal methods that are critical to the understanding of this thesis. Formal methods are about the obtention of high-assurance proofs. They encompass a family of tools that involve mechanized logic — SAT-solvers, proof assistants, and everything in between. Formal methods usually are a means more than they are an end, and this is especially true in this thesis: we want high-assurance guarantees about hardware security, and we obtain these guarantees through formal methods.

A proof is a purported sufficient argument for the truth of a proposition. All proofs promote belief in propositions, but not all are created equal: some favor terseness, and others take a more long-winded approach, usually to the benefit of understandability. Importantly, proofs may be incorrect.

Most proofs are first developed informally. The overall structure of the argument is given, but details perceived as simple enough are left to the recipient's imagination — pedantry and readability do not go hand in hand; details do not matter so long as they can be reconstructed on the fly. The issue is that the handwaviness of informal arguments leaves room for errors to sneak into demonstrations.

On the other hand, formal proofs (the ones formal methods are concerned with) are spelled out in such a way as to preclude ambiguity. Although formal proofs may still be wrong, it is possible to verify the correctness of a proof purely mechanically. Instead of trusting the whole edifice of mathematics, one only has to trust some simple verification procedures.

Section outline

As is customary by this point, we start with a brief overview of the history of formal methods in Subsection 2.3.1. We then discuss the Coq proof assistant in detail in Subsection 2.3.2. This tool plays a central role in this thesis. In Subsection 2.3.3, we present SAT-solvers, efficient tools to automate reasoning about some simple fragments of logic. We finish with Subsection 2.3.4, a short introduction to the notation we use for logical inference throughout this thesis.

2.3.1 A brief history

Formal methods are about the development and the mechanical verification of formal proofs. Formal proofs are not the standard in mathematics; indeed, the historical development of mathematics was mostly anarchic. That is not to say that early works were entirely devoid of rigor — for instance, Euclid's *Elements* [38], compiled around 300 BC, follows an axiomatic structure, where (most of) the required fundamental assumptions are laid out clearly and separately from the argumentation proper.

Modern logic can be dated back to the nineteenth century. At the time, concerns about mathematics' lack of proper foundations led to a search for a system of axioms that could finally ground this structure. Such a hypothetical system comes with a list of desiderata: it should be expressive enough to encode all of mathematics while being both syntactically complete (meaning that for each proposition, either it or its negation should be provable) and consistent (meaning that at most one out of a proposition and its negation should be provable). The result would be decidable: we could devise a procedure which, given a proposition, returns either a proof of it or a proof of its negation. And, just like that, mathematics would become a solved problem (barring the fact that the existence of such a procedure would not say anything of its practicality).

Some naive mistakes thwarted the first attempts at building such systems. In 1902, Bertrand Russell was the first to publish proof of the inconsistency of the historical version of set-based axiomatic systems, a result now known as Russell's paradox [39], [40]. However, his own efforts alongside Alfred Whitehead in formalizing mathematics as Principia Mathematica [41] also fell short of their complete goal. In fact, after a few decades of research and many attempts, there was still no solution in sight.

In his seminal 1931 paper [42], Kurt Gödel crushed many hopes. He showed that past a degree of complexity, a system of axioms must necessarily remain incomplete to avoid inconsistency. Nevertheless, the related problem of taking a formal proof and verifying its correctness is decidable.

As troubling as Gödel's theorem and its successors may be on the surface, they rarely pose a problem in practice. These theorems do not reveal mathematical logic to be vacuous, only that a commonly held belief about its shape was wrong. The existence of undecidable propositions rules out end-all procedures for logic, but undecidable propositions are absent from most workloads where formal logic can be applied anyway. End-almost-all procedures do exist: there are procedures for deciding all decidable statements. They are too inefficient for practical purposes, but that is another question. Besides, there are many fragments of logic for which efficient solvers can be found.

The prominence of questions of logic in the early twentieth century impacted the development of early computers. The fact that logicians, concerned with questions related to the nature and reach of computability, were involved is not surprising: computers are the concrete manifestations of the hitherto abstract notion of computation. For instance, a certain Alan Turing described the mathematical model of computation known as the Turing machine in 1936 [43]. Less than ten years later, he became directly involved in early computer design. Similarly, his PhD advisor, Alonzo Church, developed lambda calculus, another computation model based on functions [44]. This formalism was later credited as an influence of the Lisp programming language and functional languages.

Formal methods were not developed immediately in the wake of the development of early computers. Automath [45], the first proof assistant (a tool used both for certifying the validity of proofs and for helping users build these proofs in the first place), was started by Nicolaas Govert de Bruijn in the 1960s. Around the same time, DPLL [46], an efficient procedure for the boolean satisfiability problem (the problem of finding whether a valuation for boolean variables that satisfies a given boolean formula exists, also known as the SAT problem), was devised. Similar tools are used to this day. In fact, throughout the following two subsections, we will get acquainted with a modern proof assistant and SMT-solvers, an extension of SAT-solvers.

2.3.2 The Coq proof assistant

Coq [47] used to be the name of the Rocq theorem prover. At the time of this writing, the renaming has yet to go through — by the time of your reading, it likely has. We refer to this proof assistant using its historical name because it is still the only official one here in the past. Also, some names used for tools introduced in this thesis are puns with critical dependencies on this old name.

Coq, then, "is a formal proof management system[, which] provides a formal language to write mathematical definitions, executable algorithms, and theorems together with an environment for semi-interactive development of machine-checked proofs"[1].

With a tool such as Coq, users can define (or import) domain-specific knowledge relevant to their research and start proving properties involving those concepts. The proofs thus produced can be verified by what is called Coq's kernel, a conceptually simple proof checker. The proofs are unassailable, assuming that the domain-specific knowledge is soundly encoded in the proof assistant and that the Coq's kernel is sound. The checker provides solid guarantees — mechanically verifying proofs this way is the gold standard of high-assurance demonstration.

Coq is a formal tool. Although formal proofs are fully explicit, this does not mean that all the details always need to be written in full: what matters is that the kernel can reconstruct them. Therefore, although formal, Coq proofs can be written in a human-readable way (classical, informal proofs are better at this, however). Furthermore, Coq has good support for automation, which helps with writing terser proofs. Users can define custom tactics to automate away the tedium.

Coq's syntax is flexible, letting users introduce custom syntactical constructs for their needs. This feature is handy for embedding Domain Specific Languages (DSLs) such as HDLs. That way, circuits can be written in Coq mostly transparently, they can be simulated within the system, and we can prove properties both about the language itself and concrete circuits.

While Coq is a versatile tool, it does have its drawbacks. One of the most common issues is performance [48]. Reasoning about constructs that require significant memory can be challenging, and there are numerous subtle factors that can negatively impact performance. Additionally, when considered as a programming language, Coq falls short in providing essential debugging tools.

2.3.3 SMT-solvers

An SMT-solver is an extension of an SAT-solver, a tool that checks whether a formula over boolean variables is satisfiable. When the formula is satisfiable, the solver usually outputs a concrete valuation, which makes the formula true. Although the problem of SAT solving is NP-complete, SAT-solvers are very efficient in practice, in addition to being a widely applicable tool. Indeed, although most problems are not naturally expressed in the form expected by SAT-solvers, a large part can be translated into such

[1]https://coq.inria.fr/

$$\frac{A \Rightarrow B \qquad A}{B}\ \textsc{ModusPonens}$$

Figure 2.4: Inference rule for *modus ponens*

a form.

SMT-solvers add so-called *theories* to SAT-solvers. These theories are handy for handling problems that do not lend themselves well to representation as formulas over boolean variables. For instance, most SMT-solvers include a theory for reasoning about linear arithmetic. Through it, statements such $((a > 8) \wedge (a < 6))$ can be trivially shown to be unsatisfiable.

Whereas proof assistants are interactive tools, SMT-solvers are automatic tools. It almost looks like they sit at two opposite ends of a spectrum, with proof assistants being very expressive but complex and SMT-solvers limited to some simpler fragment of logic but very easy to use. However, some SMT-solvers said to be proof-producing justify their answer. Such proofs can then be imported into proof assistants. That way, we can have the best of both worlds: we can manually carry out high-level reasoning tasks into proof assistants and delegate the rote parts to automatic procedures.

2.3.4 Notations for logic

This section introduces notations for logical inference that are used in the rest of this document. Inference rules describe which deductions follow from given states of information. The presentation of inference rules follows the format introduced in Figure 2.4. There, we present the *modus ponens* rule, which states that $((A \Rightarrow B) \wedge A) \Rightarrow B$ (or, in plain English, “for any A and B, when we know that A implies B and that A holds, then B must hold as well”). Premises are written above the line, and the conclusion stands below it.

We present the semantics of computer languages operationally; in other words, we focus on *how* programs are evaluated. Our definitions rely on transition relations. We give several inference rules which can be applied to generate transitions. Operational semantics can be split into two categories: big-step semantics and small-step semantics.

$$\textsc{Sequence}\quad \frac{(\mathbf{i_1}, \sigma) \Downarrow (\sigma') \qquad (\mathbf{i_2}, \sigma') \Downarrow (\sigma'')}{(\mathbf{i_1}; \mathbf{i_2}, \sigma) \Downarrow \sigma''}$$

Figure 2.5: Inference rule for a sequence of instructions (big-step version)

$$\textsc{SequenceStep}\quad \frac{(\mathbf{i_1}, \sigma) \downarrow (\mathbf{i'_1}, \sigma')}{(\mathbf{i_1}; \mathbf{i_2}, \sigma) \downarrow (\mathbf{i'_1}; \mathbf{i_2}, \sigma')} \qquad \textsc{SequenceSkip}\quad \frac{}{(\mathbf{skip}; \mathbf{i_2}, \sigma) \downarrow (\mathbf{i_2}, \sigma)}$$

Figure 2.6: Inference rule for a sequence of instructions (small-step version)

Big-step semantics In big-step semantics, transitions relate instructions to values $(\mathbf{i}, \sigma) \Downarrow \sigma'$ ("the interpretation of instruction $\mathbf{i}$ in the environment σ results in the environment σ'"). In Figure 2.5, we present the big-step version of the rule used for evaluating sequences of instructions. In plain English, it can be read as "if the evaluation of $\mathbf{i_1}$ in an environment σ results in an environment σ', and if the evaluation of $\mathbf{i_2}$ in this environment σ' results in an environment σ'', then the evaluation of $\mathbf{i_1}; \mathbf{i_2}$ in an environment σ results in an environment σ''".

Small-step semantics On the other hand, in small-step semantics, transition rules return new transition states $(\mathbf{i_1}, \sigma) \downarrow (\mathbf{i_2}, \sigma')$ ("the interpretation of instruction $\mathbf{i_1}$ in environment σ is the same as the interpretation of instruction $\mathbf{i_2}$ in environment σ'"). Steps progressively simplify the instruction until it becomes **skip**, a special construct used to indicate that nothing is left to evaluate. At this stage, the value of the environment is the result. In Figure 2.6, we present the small-step version of the sequence rule. Here, it comes with two rules. The first one, SequenceStep, simplifies the left branch of the sequence. It gets applied repeatedly until this branch is fully simplified (i.e., its instruction becomes **skip**). At this point, the SequenceSkip rule can be applied to eliminate the first branch.

Small-step semantics have some advantages over big-step semantics. In particular, they can handle non-termination and are a more natural suit for modeling non-deterministic scenarios. In this thesis, only small-steps semantics are used.

2.4 Formal verification and hardware design

In this shorter section, we give a high-level overview of the problem of hardware verification — the whole detail is given in the state of the art, which can be found in Section 3.1

Formal verification is an established practice in hardware design [49], where it takes various forms. For instance, model checking [50] is a method whereby properties of hardware are verified on a finite state machine-based representation of the system. Another common method is equivalence checking [51] where the functional equivalence of two distinct designs is checked. Typical implementations rely heavily on automated procedures (this is often used to check that optimizations preserve the semantics of a design; it is usually possible to show what the differences are when they exist). Both techniques are well-suited to the needs of commercial hardware designers: they are efficient and relatively approachable. Other common formal techniques include abstraction refinement [52] and temporal logic-based tools (linear temporal logic [53], computation tree logic or Property Specification Language [54]). Various implementations of all these techniques are proposed in industry-grade tools, as discussed in greater detail in Section 3.1.

Despite their versatility, these approaches come with limitations. The main one is their sensitivity to the state explosion problem [3]: the number of states often increases exponentially with the complexity of the design. As outlined by Clarke et al. [52], abstraction is an important technique for working around this problem. Tools made for a certain category of usecases are typically limited in their abstraction ability. Furthermore, industrial tools have an impact on the TCB (a notion we introduced in Subsection 2.2.3) that scales with their complexity.

In contrast, proof assistants can give high-trust proofs about approximately everything. The versatility of proof assistants comes at a cost, mainly paid in accessibility. Overall, the practice of proof assistant-based hardware verification is still at an early stage of development. Given the current state of tooling, the properties usually checked by chip designers are much easier to verify with established systems than with these tools.

The TCB can be easily surveyed in proof assistant-based developments. Indeed, hypotheses must be introduced explicitly before they can be used by such a tool. In a sense, it becomes part of the TCB. One of the fundamental goals of proof assistants is to reduce the trusted base of reasoning in a more general sense: a simple proof-checking procedure must be trusted, but the correctness of checked proofs then comes for free.

Improvements to the tooling could make proof assistants a realistic alternative to classical checkers. Right now, the barrier of entry is high enough to skew the cost-benefit balance the wrong way. Improving usability means pushing this balance towards something more acceptable for manufacturers. As a concrete example, there is little in the way of integrating tactics for, e.g., verified equivalence checking from within proof assistants, with about the same performance as the tools more commonly used today. The initial implementation cost of verified procedures is high, but that of their subsequent uses is negligible.

The first challenge is that the industry is dominated by only a few HDLs, none built with formal methods in mind. Several projects aim to retrofit formal semantics onto classical languages, and research HDLs built with formal semantics from the ground up also exist. In the upcoming section, we discuss an example of such an HDL.

2.5 The Kôika language

Kôika [55] is an open-source formal HDL. Being embedded in Coq, this language provides a firm basis for verifying hardware. It is based on Bluespec [27], a general-purpose, high-level, non-formal HDL.

The Kôika compiler is formally verified and outputs Verilog code. One can thus apply Verilog-compatible tools (e.g., simulators and FPGA bitstream generators) to Kôika designs. The project's custom simulator "Cuttlesim" [56] allows for efficient direct simulation of Kôika designs. Indeed, Kôika sits at a level of abstraction that lends itself better to efficient simulation than RTL: its form is close to that of regular software, so standard software optimization techniques apply.

Figure 2.7 presents the classical Kôika hardware development pipeline for developing unverified hardware. Indeed, although the language itself is formalized and some interesting metaproperties have been proved about it, hardware produced with Kôika is not necessarily verified. Using a verified compiler brings benefits even when the object being compiled is not itself verified. Namely, we can safely assume that the compiler will respect a specification and in particular that it will not be a source of bugs. In practice, verifying a compiler for a language and verifying a program/circuit written in that language are two quite distinct tasks. They are related in that they both

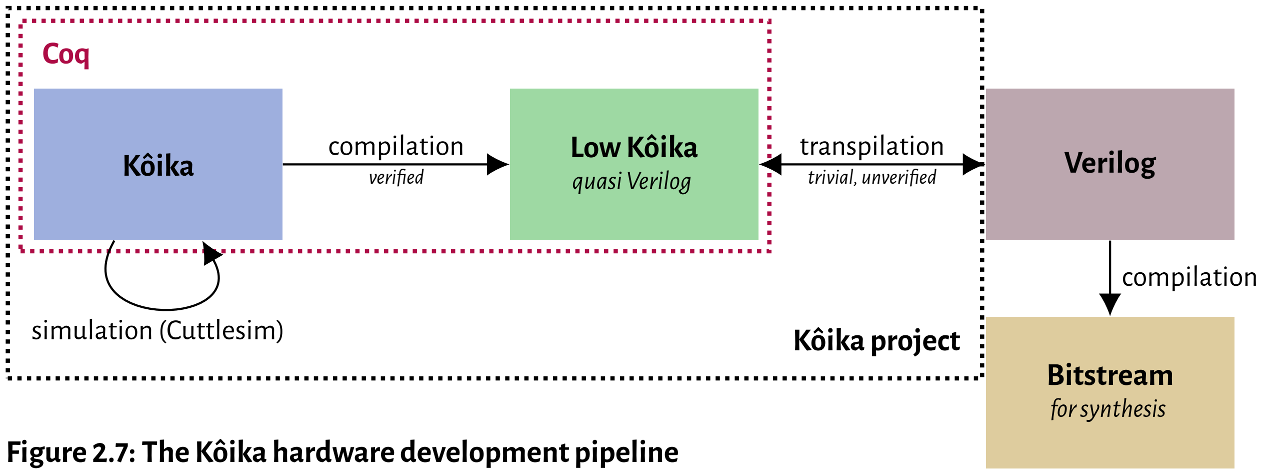


Figure 2.7: The Kôika hardware development pipeline

require a formal semantics of the language in question, but the similarity stops there.

Section outline

This section aims to arm the reader with background knowledge on the fundamentals of the Kôika language. We start with a high-level overview of the language in Subsection 2.5.1 before giving a more detailed description of the language's syntax and semantics in Subsection 2.5.2. Finally, in Subsection 2.5.3, we turn to its limitations when using it to prove hardware designs.

2.5.1 Rule-based HDLs

Just like Bluespec [27], Kôika is a rule-based language. A Kôika design is expressed as a set of independent rules, expressed sequentially but run concurrently. The compiler must generate the actual hardware, maximizing parallelism while avoiding hazards — most of the control logic is generated automatically by the compiler, with some user guidance. This way of describing hardware is particularly convenient for concurrent systems such as pipelined processors, where one rule approximately corresponds to one stage.

Kôika rules can be viewed as atomic transactions, which either execute fully or not at all during a cycle. On every cycle, the automatically derived control logic indicates whether each rule should run. This scheduler is constructed in such a way as to avoid *rule failure*s: some actions are declared incompatible by the semantics. If running a rule in a given cycle would lead to a failure, then the rule must be dropped from the schedule for that cycle.

A failure is either triggered explicitly or arises from the incompatibility of two attempted actions. We call this latter situation a conflict. For instance, a conflict occurs whenever a rule attempts to access the value of a register that is already written to by a rule scheduled with a higher priority (scheduling priority is defined explicitly by the user).

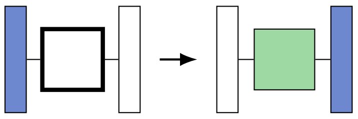
In pipelined designs, most of the communication between stages occurs across cycles. In Kôika, a pipeline stage typically writes its results onto a stack of depth one, and the stage following it pops the same stack during the next cycle to access this information. Crucially for pipelining, both pushing to a full stack and popping from an empty stack result in a failure. Figure 2.8 shows how such failures lead to implicit stalling in such designs. In this figure, stages are represented by squares ($\boxed{\text{A}}$) and stacks are represented by rectangles (▯).

On a given cycle, each rule either succeeds (🟩) or fails (🟥). The stacks between the different stages contain information about instructions. They are colored with different shades of blue. The colors indicate which instructions' data are stored in which stage (▯, ▮, etc.), which helps with tracking individual instructions over time. Later instructions are associated with darker colors. If a stack is empty, it is left uncolored (▯). A full pipeline may then look something like this: 🟩─▮─🟩─▯─🟩.

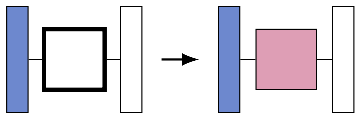
Subfigure 2.8a shows how the success of a rule influences its environment: when a rule succeeds, it takes the data from the stack before it, handles it, and then outputs relevant data related to the same instruction to the next stack. For this to work, the previous stack needs to have contents to pop, and the next one needs space to push into.

Subfigure 2.8b shows that when a rule fails, it does not impact its environment in any way. In particular, the stack in front of it stays full. Subfigure 2.8c shows how this behavior leads to failures propagating down the pipeline. Here, there are two rules with their associated priorities. Note that the rule that comes up later in the pipeline is the one that has the higher priority. Therefore, it will be considered earlier in the scheduling process. However, this rule fails and leaves the stack before it in its filled state. The other rule, not being able to output anything, also fails.

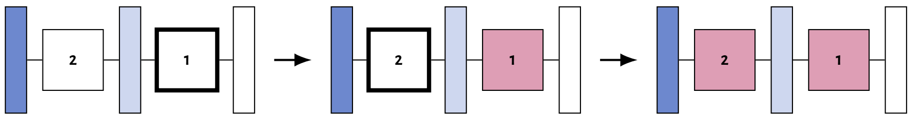
It is typical to schedule rules so that the ones that appear more to the right have higher priority: the stack following them needs to be emptied before they write to it,



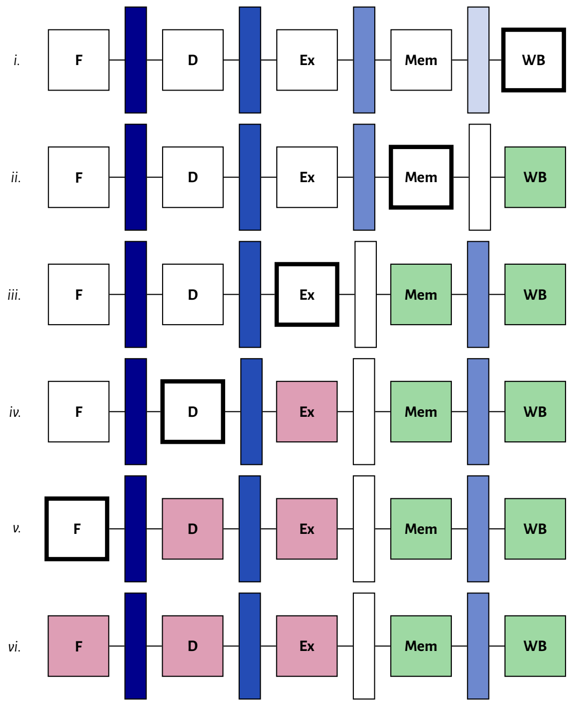
(a) Successful execution of a rule



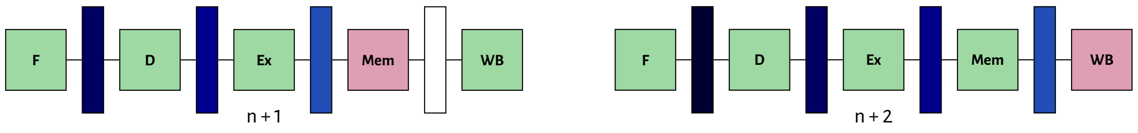
(b) Rule failure — the stack stays full



(c) One cycle of execution of a longer subset of a pipeline.
Rules are annotated with their scheduling order. Note how the failure propagates backwards: after rule **1** failed, the queue between it and rule **2** remained full. Rule **2** was bound to fail, as its write could not succeed.



(d) Failure propagating while evaluating cycle n of a pipelined processor
The propagating nature of failures shown in (c) implicitly leads to stalling behavior here (we assume that Ex fails due to a read after write hazard). We split the evaluation of the cycle into a number of discrete steps (i. to vi.), building a schedule for that cycle. Step vi. is the final state of the pipeline on cycle n.



(e) Results of cycles n + 1 **and** n + 2

Figure 2.8: Rules failures and implicit control logic

conceptually speaking.

Subfigure 2.8d shows a complete example of a failure propagating through a classic RISC pipeline during a cycle. This pipeline is composed of five stages: fetch (F), decode (D), execute (Ex), memory access (Mem) and Writeback (Mem). Although the last few stages successfully execute (see *ii.* and *iii.*), stage Ex encounters a failure (*iv.*). This failure ends up affecting the two stages before it, D (*v.*) and F (*vi.*). Kôika's semantics gives us this implicit stalling behavior for free.

Finally, possible states for the pipeline after the following cycle and the one after that are shown in Subfigure 2.8e. On cycle $n+1$, even when assuming that all other rules succeed, Mem is bound to fail as the Ex to Mem stack was left empty after cycle n. Note that the rest of the data works its way towards the pipeline's exit. At the end of cycle $n+1$, Ex has written to this stack. Therefore, Mem can succeed on cycle $n+2$. This time, however, WB having popped the Mem to WB stack and Mem not having filled it with new contents, the rule fails. Thus, the bubble also progresses toward the pipeline's exit. In real pipelined processors, optimizations such as operand forwarding can help limit unnecessary stalls by allowing data to move against the flow of the conventional data path. Kôika handles such optimizations, as we shall soon see.

2.5.2 Syntax and semantics

2.5.2.a Syntax

A Kôika design manipulates a set of stateful elements that form its microarchitectural state. In Kôika parlance, these stateful elements are called "registers" — not to be conflated with CPU registers. Although CPU registers can be Kôika registers, other objects count as Kôika registers, such as the stacks mentioned in Section 2.5.1.

As shown in Figure 2.9, a design is additionally composed of a set of rules alongside a schedule. Each rule can be seen as an atomic transaction describing a transition in the microarchitectural state from one clock cycle to the next. The schedule is a list of such rules. Its role is to constrain the hardware generation. Indeed, although the hardware circuit will execute the rules concurrently, it is described sequentially. This linear description gives rise to a simple semantics of reference, a behavior to which the hardware must conform. The compiler's role is to generate control logic to enforce the resulting constraints.

Figure 2.10 provides a sample action illustrating Kôika's syntax. This action updates

$$
\begin{array}{llcl}
\text{Actions} & a & ::= & v \mid x \mid \texttt{skip} \\
 & & \mid & \texttt{read}\ r \mid \texttt{write}\ r\ a \\
 & & \mid & \texttt{let}\ x = a\ \texttt{in}\ a \\
 & & \mid & x := a \\
 & & \mid & \texttt{if}\ a\ \texttt{then}\ a\ \texttt{else}\ a \\
 & & \mid & f(a, \ldots, a) \mid a ; a \\
 & & \mid & \texttt{abort} \\
\text{Registers} & r & & \\
\text{Variables} & x & & \\
 & & & \\
\text{Program} & P & ::= & [\texttt{Rule}\ \mathit{name} = a]* \\
 & & + & \texttt{Schedule}\ = \overrightarrow{\mathit{name}} \\
 & & & \\
\text{Values} & \mathit{val} & ::= & \vec{b} \mid \{(k : \mathit{val})*\} \mid \{\mathit{val}*\} \mid \mathit{enum\_val} \\
\text{Types} & \tau & ::= & \texttt{Bits}\ n \mid \texttt{Struct}\ [(k, \tau)*] \\
 & & \mid & \texttt{Array}\ n\ \tau \mid \texttt{Enum}\ [\mathit{variant}*]
\end{array}
$$

Figure 2.9: Syntax of Kôika designs

the state either of register a or of register b depending on the state of register a at the beginning of the cycle. A variable x is introduced in this example. The difference between registers and variables is that registers are stateful components. In contrast, variables are transient values local to a single rule.

```
let x = (read a) + 1 in
if x == 5
then write a 0
else
  x := x + 4;
  write b x
```

Figure 2.10: A Kôika action

This action writes either to register a or to register b

2.5.2.b Semantics

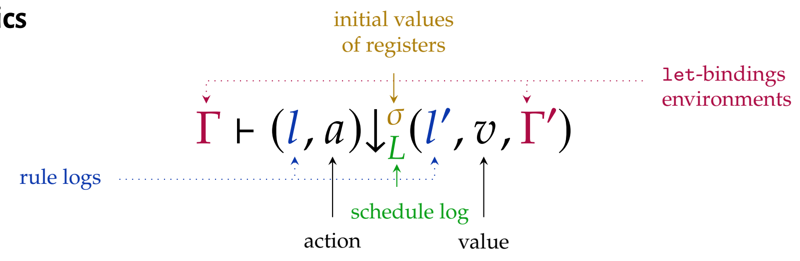

initial values of registers
let-bindings environments
$\Gamma \vdash (l, a) \downarrow^{\sigma}_{L} (l', v, \Gamma')$
rule logs
schedule log
action
value

Figure 2.11: Anatomy of a judgment

Before turning to the semantics of full schedules, we must consider the semantics of the actions from which they are built. Indeed, a Kôika rule is simply a named action. The successful execution of an action produces a value and updates the log that stores the sequence of read and write events on registers. Figure 2.11 details the general form of an action judgment ($\Gamma \vdash (l, a) \downarrow^{\sigma}_{L} (l', v, \Gamma')$). The full semantics is given in Figure 2.12, by way of a set of such judgments, where Γ is an environment for `let`-bound variables, l is a "rule log", i.e. a log of events (read or write accesses on registers) that occurred so far during the execution of the current rule, a is the action to be executed; and l' is an updated rule log, v is the value computed by action a and Γ' is an updated environment. The environment Γ is a mapping taking variable names to their associated values. Moreover, these judgments use two components that are never updated in the semantics of actions: an environment σ that gives the value of each register at the beginning of the cycle, and a "schedule log" L that contains the read and write events produced by previous rules (as defined by the schedule) during the same cycle. For the sake of clarity, when there is no ambiguity about which σ and L should be used, we write $\downarrow$ instead of the full $\downarrow^{\sigma}_{L}$.

The environment (Γ) gets updated in rule Bind. In this rule, we start by adding a binding to the environment ($(x, v_1) :: \Gamma'$). This binding gets removed at the end when the variable goes out of scope (`tl` Γ'', where `tl` is the function that returns a list's tail). Rule Funcall, which describes the semantics of function calls, creates a new environment for it by using `zip`, the function that combines two lists into a list of pairs (i.e., `zip [x1; ...; xn] [v1; ...; vn] = [(x1, v1); ...; (xn, vn)]`). `args` f and `body` f give the names of the arguments and the body of a function f.

The events in the schedule log and rule log are of the form $\mathbf{rd}(r)$ or $\mathbf{wr}(r, v)$, denoting respectively a **`read`** on register r, and a **`write`** of value v on register r. Operator ++ is the concatenation of lists. The separation between the schedule log and the rule log is necessary because the effects of rules may be canceled in the presence of conflicts

Semantics of actions:

$$\textsc{Cst}\ \dfrac{v \in val}{\Gamma \vdash (l, v) \downarrow (l, v, \Gamma)} \qquad \textsc{Var}\ \dfrac{\Gamma(x) = v}{\Gamma \vdash (l, x) \downarrow (l, v, \Gamma)} \qquad \textsc{Assign}\ \dfrac{\Gamma \vdash (l, a) \downarrow (l', v, \Gamma')}{\Gamma \vdash (l, x := a) \downarrow (l', tt, \Gamma'[x \mapsto v])}$$

$$\textsc{Bind}\ \dfrac{\Gamma \vdash (l, a_1) \downarrow (l', v_1, \Gamma') \qquad (x, v_1) :: \Gamma' \vdash (l', a_2) \downarrow (l'', v, \Gamma'')}{\Gamma \vdash (l, \texttt{let}\ x = a_1\ \texttt{in}\ a_2) \downarrow (l'', v, \texttt{tl}\ \Gamma'')}$$

$$\textsc{Cond-True}\ \dfrac{\Gamma \vdash (l, a_1) \downarrow (l', [\texttt{true}], \Gamma') \qquad \Gamma' \vdash (l', a_2) \downarrow (l'', v, \Gamma'')}{\Gamma \vdash (l, \texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3) \downarrow (l'', v, \Gamma'')}$$

$$\textsc{Cond-False}\ \dfrac{\Gamma \vdash (l, a_1) \downarrow (l', [\texttt{false}], \Gamma') \qquad \Gamma' \vdash (l', a_3) \downarrow (l'', v, \Gamma'')}{\Gamma \vdash (l, \texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3) \downarrow (l'', v, \Gamma'')}$$

$$\textsc{Funcall}\ \dfrac{\begin{array}{c}\Gamma_0 \vdash (l_0, a_1) \downarrow (l_1, v_1, \Gamma_1)\ \ \ldots\ \ \Gamma_{n-1} \vdash (l_{n-1}, a_n) \downarrow (l_n, v_n, \Gamma_n) \\ \texttt{zip}(\texttt{args}\ f, [v_1, \ldots, v_n]) \vdash (l_n, \texttt{body}\ f) \downarrow (l', v, \Gamma')\end{array}}{\Gamma_0 \vdash (l_0, f(a_1, \ldots, a_n)) \downarrow (l', v, \Gamma_n)}$$

$$\textsc{Seq}\ \dfrac{\Gamma \vdash (l, a_1) \downarrow (l', v_1, \Gamma') \qquad \Gamma' \vdash (l', a_2) \downarrow (l'', v, \Gamma'')}{\Gamma \vdash (l, a_1\ ;\ a_2) \downarrow (l'', v, \Gamma'')}$$

$$\textsc{Read}\ \dfrac{\mathbf{may\_read}(L, r)}{\Gamma \vdash (l, \mathbf{read}\ r) \downarrow^{\sigma}_{L} (l{+}{+}[\mathbf{rd}(r)], \sigma(r), \Gamma)} \qquad \textsc{Write}\ \dfrac{\Gamma \vdash (l, a) \downarrow (l', v, \Gamma') \qquad \mathbf{may\_write}(L, l', r)}{\Gamma \vdash (l, \mathbf{write}\ r\ a) \downarrow^{\sigma}_{L} (l'{+}{+}[\mathbf{wr}(r, v)], tt, \Gamma')}$$

$$\begin{array}{lcl}\mathbf{may\_read}(L, r) & \triangleq & \mathbf{wr}(r, *) \notin L \\ \mathbf{may\_write}(L, l, r) & \triangleq & \mathbf{wr}(r, *) \notin (L{+}{+}l)\end{array}$$

Semantics of a schedule:

$$\dfrac{\Gamma_0 \vdash (l_0, a) \downarrow^{\sigma}_{L} (l, v, \Gamma) \qquad (L{+}{+}l, sch) \Downarrow L'}{(L, a :: sch) \Downarrow L'} \qquad \dfrac{\Gamma_0 \vdash (l_0, a) \not\downarrow^{\sigma}_{L} \qquad (L, sch) \Downarrow L'}{(L, a :: sch) \Downarrow L'}$$

$$\dfrac{}{(L, [\,]) \Downarrow L}$$

Figure 2.12: Semantics of Kôika actions and schedules

between register reads and register writes. More precisely, a conflict happens in two situations: *read-after-write* (see rule READ in Figure 2.12), i.e. when a read on a register happens after a write on the same register by a previous rule (this information is found in the schedule log); or *write-after-write* (see rule WRITE in Figure 2.12), i.e. when a write happens after another write on the same register by a previous rule (schedule log) or by the same rule (rule log). The conditions under which reads and writes are permitted are decided by the **may_read** and **may_write** functions. Note that reading a register after a write has occurred within the same rule is allowed, and the value thus obtained is the value of the register at the beginning of the cycle.

The *read-after-write* conflict exists in order to respect the One-Rule-At-A-Time compiler property (ORAAT), i.e., the parallel execution of several rules is only allowed if it cannot be distinguished from a sequential execution of these rules, in the order described by the user-provided schedule. This property characterizes the constraints put on the compiler for hardware generation. With it, we can show that any property of designs proven by considering the rules in the scheduling order also holds for the actual, parallel design output by the compiler.

For an example of the impact of conflicts such as *read-after-write*s on scheduling, consider the rules `A: write r 0` and `B: read r`, and assume register `r` initially holds the value 1. With the schedule $[A; B]$, rules A and B cannot run concurrently as this would lead to a *read-after-write* conflict on register `r`. On the other hand, with the schedule $[B; A]$, the two rules may execute concurrently and result in the register `r` having value 0.

On the other hand, the *write-after-write* conflict is not necessary to satisfy ORAAT. Rather, it is a design choice of the authors of Kôika, who consider that overwriting registers by shadowing them is an antipattern. For instance, the rule `C: write r0 1; write r1 2; write r0 3` would write twice to register `r0`.

At the bottom of Figure 2.12, we introduce three rules that describe the semantics of a schedule, built on top of the semantics of actions. We run each rule with an initially empty environment Γ_0 and an empty rule log l_0. We write $\Gamma \vdash (l, a) \not\downarrow$ to denote that the execution of action a fails, i.e. does not result in a triple (l', v, Γ'), e.g. because of a conflict. Conflicts result in the cancellation of all the effects of the conflicting rule. In the example of rule `C`, this means not only that none of the writes to `r0` happen, but also that the write to `r1` is also ignored. The cause of this behavior is that rules are considered atomic, i.e., they execute entirely or not at all.

```
Rule A:
  if (read a == 0) then write r 0;
  if (read b == 0) then write r 1.
```

Figure 2.13: An action with a potential *write-after-write* conflict

These conflicts must be detected at run-time because the conflicting register reads or writes may be nested inside conditional expressions and thus cannot be decided during compilation. For instance, in Figure 2.13, rule A is bound to fail only when both `a` and `b` are `0` (as there is a double write on `r`).

Kôika's compiler introduces the appropriate control logic in the generated circuit to run rules in parallel, and only *commit* each rule's effects when there are no conflicts. This behavior and the notion of scheduling simplify the definition of pipelined systems: conflicts help determine how to pipeline a design without the user needing to give all the details explicitly.

The rules that run in a given cycle depend on the registers' initial value. These values are stored in σ, the association map that stores the register values at the beginning of the cycle. The constant changes in the environment throughout the execution of a design give rise to a dynamic system. As shown in Figure 2.8, rule cancellations can lead to stalling behavior arising on its own in the context of pipelined systems such as processors.

Finally, the semantics of a cycle is given by a function `interp_cycle`, taking as parameters a schedule *sch* and an initial state for registers σ, and producing as a result a new state for registers. For instance, `interp_cycle`$(sch, \sigma)(r)$ is the value of register r at the end of a cycle. This function implements the semantics of Kôika schedules presented in Figure 2.12 (this is possible because the semantics is both total and deterministic).

2.5.2.c Ports

In the language we have described thus far, passing information between two different rules during a single cycle is impossible, effectively precluding classical optimizations such as operand forwarding.

Kôika lifts this restriction through the port mechanism. Two ports, 0 and 1, are added to each register. They correspond loosely to versions of the associated register across time (see Figure 2.14). Reads and writes are associated with a specific port. One

$$
\begin{array}{rcll}
\text{Ports} & p & ::= & 0 \,|\, 1 \\
\text{Actions} & a & ::= & \ldots \\
& & | & \texttt{read}_p\ r \,|\, \texttt{write}_p\ r\ a
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{may\_read}(L, r, \texttt{P0}) & \triangleq & \mathbf{wr}(r, *, *) \notin L \\
\mathbf{may\_read}(L, r, \texttt{P1}) & \triangleq & \mathbf{wr}(r, \texttt{P1}, *) \notin L \\
\mathbf{may\_write}(L, l, r, \texttt{P0}) & \triangleq & \mathbf{wr}(r, *, *) \notin (L{+}{+}l) \\
& & \wedge\, \mathbf{rd}(r, \texttt{P1}) \notin (L{+}{+}l) \\
\mathbf{may\_write}(L, l, r, \texttt{P1}) & \triangleq & \mathbf{wr}(r, \texttt{P1}, *) \notin (L{+}{+}l)
\end{array}
$$

Figure 2.14: Syntax and semantics of Kôika designs with added ports

can only read the value of a register on port 0 if no write has occurred on that register in the current cycle, but we can read on port 1 if no write has occurred on port 1. If a write has occurred on port 0, a read on port 1 will retrieve the value that was written, whether the read and write occur in the same rule or later.

The conflicts between reads and writes are now more subtle: reads on some port p are only allowed if no write has already occurred in a rule before the current one on any port $p' \geq p$. Writes on some port p are allowed neither after writes on ports $p' \geq p$ nor after reads on ports $p' > p$. This last part means that, for example, once a read on port 1 occurred, we cannot have a write on port 1.

The notion of ports is crucial for the performance of hardware designs but it adds some complexity to the semantics of Kôika. In the rest of this chapter, we ignore them for the sake of simplicity, although they are fully supported in the implementation.

2.5.3 Verifying Kôika designs

Although it is possible to prove properties about the behavior of simple Kôika circuits using the language in its current state, this becomes impractical as the designs grow in complexity. Indeed, performance deteriorates very fast. There are multiple causes for this issue, which is the consequence of design choices in the Kôika language and of properties of the Coq proof assistant, in which the reasoning is carried out.

First, the semantics of Kôika is inherently non-modular. Consider, for instance, a single register written inside one of the rules of a Kôika design. In order to determine whether this register write occurs, one needs to check whether this rule runs or is canceled for the current cycle. Cancellation conditions, which depend on the evaluation

```
Registers: {r0, ..., r15}.

Rule tick:
  write r0 (read r0);
  write r1 (read r1);
  ...
  write r14 (read r14);
  write r15 (read r15).

Schedule: [tick].
```

Figure 2.15: A simple design with many independent writes

of all the previous rules, can get quite involved. Not only is this tricky to reason with, but it can also degrade the performance of proofs involving complex designs.

Another limiting factor is related to the techniques for proofs by computation in Coq. When considering the behavior of programs with concrete inputs, efficient Coq tactics for normalization, such as `cbv` or `vm_compute`, can be applied. However, although these tactics are usually very efficient, their behavior cannot be controlled in great detail. For programs whose inputs are not all known, this approach often runs naively into combinatorial explosions. On the other hand, tactics such as `cbn` or `simpl` are an apt tool for reasoning about programs whose inputs are at least partially abstract. Indeed, these tactics can be parameterized to try and avoid running into the performance traps that would block `cbv` or `vm_compute`. Alas, even for these tactics, the level of control falls short of our needs: we often need to be able to apply transformations to some subterms of our context and these subterms only. Eagerly applying a tactic to the wrong subterm may take our context size from very large to unmanageable.

We introduce an example of a Kôika design in Figure 2.15. It contains a single rule with a sequence of writes, each targeting a different register. The design contains a total of twenty-one registers. Real-life examples usually contain more registers and actions than this one and include more complex constructs (e.g., conditionals and let-defined variables). Note that there cannot be a conflict in this rule as all writes target different registers.

Consider the following property about this design, which says that the initial and final values of r are distinct:

$$(\texttt{interp\_cycle}([\texttt{tick}], \sigma))(r_{15}) = \sigma(r_{15})$$

It should be relatively easy to demonstrate. The first fifteen writes can safely be ignored, as they do not influence the final value of `r15`, the only register considered in the property. The value of `r15` is preserved on each cycle, meaning the property is trivially true.

However, there is no straightforward way of implementing this proof using vanilla Kôika and Coq. Proofs built using the constructs provided by the original developers of the language run for about ten minutes on a machine equipped with an Intel i5-1240P CPU, even for such a simple program and property[2]. This problem worsens when larger designs with more registers, actions, and complex constructs are considered. We present our methodology to find a way around Coq's and Kôika's performance issues when reasoning on large designs in Section 4.1.

Section recap

In this section, we presented the Kôika language. After a high-level overview of what it means for Kôika to be a rule-based language, we discussed the syntax and semantics of the language before turning to its limitations regarding proofs. Kôika's semantics is well-suited to its initial purpose as a backbone for proving metaproperties of the language (e.g., for verifying that the semantics is preserved under compilation or that the ORAAT property is indeed enforced). However, we showed how its general structure might make it ill-suited to the formal verification of large-scale designs.

[2]See https://gitlab.inria.fr/SUSHI-public/FMH/koika/-/blob/bad_performance_example/examples/bad_performance.v

Chapter 3
State of the art

In this section, we discuss the state of the art in domains relevant to this work.

Chapter outline

The current state of verification in the industry is described in Section 3.1. Both formal and non-formal tools and methods are considered. We discuss proof assistant-based formal verification of hardware in Section 3.2.

3.1 Verification in the industry

Hardware security forms the backbone of computer security. The weight of this responsibility, compounded by the fact that vulnerable hardware cannot be patched like software, compels hardware designers to meticulously verify their designs before moving them into production.

Section outline

In this section, we describe the state verification within the industry. Section 3.1.1 considers non-formal approaches while Section 3.1.2 considers formal ones.

3.1.1 Non-formal verification

Not all verification is formal. In addition to the guarantees built into HDLs (e.g., type systems), test suites are standard in the industry. Universal Verification Methodology [57] (UVM) is a standardized methodology for hardware verification that is supported by most EDA software vendors. The SystemVerilog framework that UVM defines is used for building reusable testbenches.

Fuzzing-based [58] approaches, such as Constrained Random Verification (CRV), are also standard [59]. With CRV, test cases matching some constraints are generated randomly. The constraints guide the fuzzer toward relevant portions of the state space.

Synopsys VCS [60], Cadence Incisive Functional Safety Simulator [61], Siemens Questa Advanced Simulator [62], and ALDEC's Riviera-PRO [63] are proprietary tools that are widely used for simulation, debugging, and testing, relying on the methods outlined above.

Recap

While testing and fuzzing-based approaches are essential components of the hardware design workflow and precious tools for verification, they come with some strict limitations. The main one is their lack of exhaustivity: they can probe individual paths in the state space but offer no way of generalizing. These tools are approachable, but guarantees about the global behavior of the device under test are beyond their reach. This approach is therefore insufficient for reaching the highest EAL, as discussed in Section 2.2.4. In this thesis, we consider the complimentary formal approach, which lifts this limitation at the cost of a significant increase in complexity.

3.1.2 Formal verification

As mentioned in Section 2.4, although proof assistants are not used in the industry, formal methods are represented under different forms. Bounded Model Checking (BMC) can be applied to verify that properties hold for n cycles. It works by considering the states that can be reached in n steps or less and checking that the properties hold in all these states. Properties generated from assertions that designers put into their designs can be automatically checked through this approach. Its main drawback is its boundedness: it offers no way of proving that a property holds forever. Temporal induction and k-induction [64] build on top of BMC to remedy this issue. However, this methodology does not capture every provable property of designs. Siemens Questa OneSpin Formal Verification Suite [65] and the Jasper FPV App [66] are industrial tools embodying these ideas.

Equivalence checking is used to determine whether two designs are functionally equivalent. This process involves comparing the outputs of both designs for all possible inputs to ensure they behave identically, be it through binary decision diagrams or SAT-solvers. Equivalence checking is particularly valuable in verifying that optimizations or transformations applied during the design process do not alter the intended functionality. Equivalence checking is a mature technique with good industrial support. Nonetheless, it has limitations: the heuristics these tools follow work well in typical scenarios, but they struggle to handle radical departures in design. Cadence Conformal [67] and Synopsys Formality Equivalence Checking [68] are two industrial-grade tools for formal equivalence checking.

Up to this point, all the tools we gave as examples were proprietary. This is representative of the state of the industry, where such tools dominate. However, some open alternatives exist. The Hardware Model Checking Competition [69] is a competitive event where open systems for hardware model checking are compared. It consists of word-level problems described using the BTOR2 [70] format. The latest iteration of this competition was won by AVR [71], a push-button model checker that relies on the IC3 model checking algorithm [72]. NuSMV [73] is another model checker with good performance. Unlike AVR, it is based on binary decision diagrams.

More complete open suites include ABC [74], which targets both synthesis and verification (in the form of equivalence checking), and UCLID5 [75], a modeling, synthesis, and verification tool. SimbiYosys [76] is a verification suite maintained by Yosys, the team behind the Yosys open synthesis framework [10]. It is mainly geared toward safety and liveness properties.

Moroze et al. [77] propose rtlv, a push-button approach that relies on Rosette [78], a solver-aided language embedded within Racket that provides constructs for program verification and synthesis. To achieve this, the authors developed a compiler that translates a subset of Verilog into Rosette. They report strong performance in tasks involving software running on hardware for a potentially large but bounded number of cycles.

Recap

Formal tools in use within the industry are effective and meet their objectives satisfactorily. However, they are less versatile than proof assistants and come with a greater TCB imprint. Proof assistant-based formal methods address these limitations at the cost of more manual procedures.

3.2 Proof assistant-based formal verification

Section outline

We start with an overview of ISA-level security proofs in Subsection 3.2.1 before turning our attention to work pertaining to the design of HDLs with built-in security features in Subsection 3.2.2. Subsection 3.2.3 is concerned with proofs about models of hardware, while Subsection 3.2.4 reviews formal languages that compile down to Verilog, which is almost a prerequisite for a formal HDL to be usable in practice. Finally, Subsection 3.2.5 provides commentary on micro-architectural security proofs. Subsection 3.2.6 closes this section with a presentation of work related to the notion of hardware contracts.

3.2.1 ISA-level security proofs

ISAs are the specification that CPUs must implement. As such, they form the foundation for software-level security. Recent publications have focused on proving security properties at the ISA level. One prominent line of work aims to prove various security properties for capability-enabled ISAs. Nienhuis et al. [79] prove capability monotonicity for CHERI-MIPS, while Bauereiss et al. [80] prove it for the Arm Morello prototype. The work of Georges et al. [81], [82] and Skorstengaard et al. [83], [84] prove a variety of stack safety properties that can be enforced on capability machines. Similarly, Van Strydonck et al. [85] propose a library of verified wrappers around drivers leveraging capabilities for enforcing security properties. While these publications also define and prove security properties about hardware, all their reasoning is done at the ISA level. Their results do not carry to concrete hardware implementations.

3.2.2 HDLs for security

Multiple HDLs have been proposed in the literature for securely designing circuits. For instance, Caisson [86] and SecVerilog [87] are HDLs that use information-flow types to ensure that generated circuits are secure. They allow users to define a lattice of security levels with which components are labeled. The system enforces that the information flow respects the lattice of security levels in the sense that information may only flow from higher to lower security levels. Iodine [88] and Xenon [89] use SMT-solvers to check that cryptographic circuits execute in constant time. While these approaches are powerful enough for verifying *specific* security properties, they cannot be used to establish more general properties of CPUs.

3.2.3 Verification of models

There is a substantial body of literature on verifying hardware models using proof assistants. Lisboa Malaquias et al. [90] present a formal framework for verifying models of real-time DRAM controllers. Another work with the same first author [91] describes a generic framework for verifying hardware models. Similarly, Letan et al. [92] introduce SpecCert, a framework for specifying and verifying hardware-based security enforcement mechanism models.

An important limitation of these works is the distance between these models and the actual hardware they represent. Lisboa Malaquias et al. [90] propose to generate HDL code from the model to answer this concern. Hsia et al. [93], [94] consider the problem from the other side. They propose to automatically derive models from RTL descriptions of hardware (this work was not connected to proof assistants, however).

3.2.4 Formal languages compiled to Verilog

We introduced the Kôika [55] language in Section 2.5. Kami [95] is another language with a formal semantics that can be compiled down to Verilog. It was developed within the team that published Kôika and follows the same rule-based approach. Kôika has superseded this project, as this latter work offers a more precise, cycle-accurate approach. Thomas Bourgeat's PhD thesis [96] describes Fjfj, another project derived from Kami with a focus on modularity. This thesis explores how modular proofs can be constructed in practice within this framework through the use of simulation relations between specifications and implementations. Although this is illustrated on examples tied to functional properties of processors, many of the principles at play are relevant in the context of the verification of security properties. It should be noted that Fjfj is not yet publicly available.

Vericert [97], [98] is a formally verified high-level synthesis tool based on CompCert that transforms C code into Verilog but lacks support for implementing pipelines, which is crucial for implementing efficient CPUs.

Lööw et al. [99] present a HOL4-based system for the verified synthesis of Verilog code as part of the CakeML project [100]. Specifically, it has been used for designing a basic sequential CPU implementing a custom instruction set. Dong et al. [101], [102] propose a pipelined implementation of the same ISA. They verify that the result is valid with regard to the Silver specification (a functional property) using the HOL4 proof assistant. Furthermore, they guarantee the absence of unexpected side-channels (a security property) through an analysis relying on observational models, although this result is not backed by a machine-checked proof. In contrast, our work focuses on machine-checked proofs of such security properties (we also use the more popular RISC-V ISA, although the previous results are not tied to the Silver ISA).

Lööw extends the work with Lutsig [103], [104], a verified compiler from a subset of Verilog down to a technology-mapped netlist. As such, they cover ground beyond what we cover in this thesis, bringing verification one step closer to the metal.

Cava [105] is a Lava-inspired [23] DSL embedded within Coq. It was developed as part of Google Research's Silver Oak project, which was about building high assurance variants of some peripherals used in OpenTitan [106] (an open-source silicon Root of Trust project). The stated design goals of this project are close to what we advocate for in this thesis. Alas, this project appears to have been abandoned before reaching maturity.

The Esterel project embodies another line of work with both software and hardware applications. These distinct applications are handled through two different languages that are offshoots of an earlier, unified one: Esterel [107] is hardware-oriented (compiling to Verilog or VHDL) whereas Lustre [108] is software-oriented (compiling to Ada or C). Esterel and Lustre are synchronous programming languages, i.e., they were conceived to help with the design of reactive systems. Not only do these languages enforce guarantees required by synchronous systems (determinism, boundedness in time and space), they are also connected to various automatic verification tools. Lustre is actively used for software verification in the industry, especially in the automotive and aerospace sector. For instance, the Lustre-based Ansys SCADE suite [109] is commonly used to verify critical embedded software; in particular, it is certifiable according to safety DO-178C DAL A, the most stringent level attainable in this certification scheme of software-based systems for aerospace [110], [111]. Alas, the same cannot be said of Esterel, as the rights to this language are currently held by a private owner who keeps the project in stasis. The most recent publicly available version of the Esterel, v5_92 [112], was released in 2000. One of the most recent work in this area is a 2019 preprint by G. Berry et al. [113] discussing the formalization of Esterel's semantics in the

Coq proof assistant. Additionally, methods used to verify Lustre designs could apply to the verification of hardware, owing to the closeness that exists between synchronous languages in general and rule-based HDL.

3.2.5 Micro-architectural security proofs

Erbsen et al. [114] describe the implementation of a certified IoT lightbulb with the Kami HDL. This work also blends formal verification of hardware and software. The main theorem it defines relates to the validity of the behavior of the application controlling the lightbulb. Properties of the hardware, compiler, drivers, and applications are formally verified and contribute to the final proof. Since those elements may vary independently of the others, special attention was given to the proof modularity. While this work does not focus on security properties and relies on Kami, which lacks cycle-accurate semantics, it focuses on aspects of hardware verification that complement our research. Although we are not yet addressing hardware/software interactions, the methodology it explores could inform our future work.

Choi et al. [115] describe Hemiola, a Coq framework composed of a DSL for specifying cache-coherence protocols embedded within Coq alongside machinery for reasoning about such protocols. As such, this project has a more restricted scope than what we are aiming for. Hemiola can generate Kami-based microarchitectural descriptions of hardware implementing such protocols.

Lau et al. [116] verify isolation properties of hardware defined within Coq through the Kôika HDL. They introduce a methodology that relies on MTIsolation, a custom tool that discharges goals to an external SMT-solver. They apply this methodology to the verification of a multicore, pipelined RISC-V processor equipped with an enclave mechanism. They prove that this mechanism is not affected by contention-related timing side channels. Although their methodology is focused on specific attacks, it is pretty close to what we advocate for in this thesis. Nevertheless, how easily their approach could be generalized to other security properties is unclear.

Lisboa Malaquias et al. [117] use Cava to construct formally verified memory controllers. They prove a bisimulation relation between a simple formal model of a controller and a more complex RTL circuit that refines it. It is unclear how well this methodology scales beyond the provided example, as processors typically do not have formal models as simple as those for memory controllers. The controller's verification already takes on average more than 20 minutes on an i5-10210U CPU clocked at 1.60GHz, with memory usage peaking at more than 5GiB, although it should be noted that the authors of this work emphasized that their focus was not on performance.

Knox [118], [119] is a framework for building high assurance Hardware Security Modules (HSM) that is built on top of Rosette [78]. It can be used to prove that a Verilog implementation correctly refines a functional specification defined as a state machine. In contrast to our work, theirs is specific to HSM and cannot be used to prove general-purpose security properties.

3.2.6 Hardware contracts

Bidmeshki et al. [120] introduce VeriCoq, a framework for generating Coq definitions from Verilog designs. It is based on PCHIP [121], a hardware adaption of the principles of proof-carrying code. The core idea of this approach is that a set of formal requirements about the hardware design should be specified *a priori* and that the vendor should present formal certificates to customers. VeriCoq provides a Coq representation of a subset of Verilog, but it stops short of providing a verification framework.

Guarnieri et al. [122] focus on the specification side of things. They put forward a framework for specifying hardware-software contracts for security properties and provide a security properties-oriented specification of hardware mechanisms for secure speculation. However, they are not concerned with *how* the specification is enforced.

The CLI stack represents an early and influential take on processor verification, starting in the late 1980s and culminating with the fabrication of a verified FM9001 processor [123] in the early 1990s. The CLI stack encompasses both hardware and software verification. It includes a simple verified operating system [124] as well as verified compilers [125], [126]. The FM9001 itself is a very simple sequential processor implementing a custom ISA and lacking I/O mechanisms [127] that was built using the custom RTL HDL DUAL-EVAL [128]. The authors include unverified tools for converting DUAL-EVAL designs to Verilog.

The CLI stack was built in the Lisp-based proof assistant Nqthm. Work on this project led to an improved, industrial strength version of Nqthm called ACL2 [129] that is still used to this day in academia as in the industry [130]. For instance, Goel et al. [131] describe the verification of the implementation of a subset of the x86 instructions. They verify relevant RTL blocks and assume a certain behavior for the rest of the system.

Although their project does not tackle end-to-end verification, it should be noted that the processor being verified is a complex, modern one, that was being developed by Centaur Technology for commercial purposes.

The Verisoft [132] stack is a project from the 2000s with a scope comparable to that of the CLI project. VAMP [133], the Verisoft processor, is more complex than FM9001. It proposes an out-of-order implementation of the DLX ISA. The processor can be converted to Verilog through unverified tools, with the result running on an FPGA. The verification of VAMP was initially done in the PVS proof assistant [134], [135] but eventually ported to Isabelle/HOL [136]. Like the CLI stack before it, the Verisoft project covers the software side of things, including a basic verified micro-kernel [137] and a verified compiler for C0 [138], a garbage-collected C-like programming language. In a more recent work (2014), Kovalev et al. [139] describe a formally verify pipelined multicore RISC machine. Alas, the proofs they present are paper-and-pencil only.

Lööw et al. [140] describe a modern example of cross-stack formal verification, where proofs about the software rely on formally certified hardware properties, in the line of the CLI and Verisoft stacks. The stack is implemented in CakeML [100] and targets a custom architecture instead of a standardized ISA [99]. The processor is a simple sequential one that is not too different from the CLI one. In particular, it is simpler than the Verisoft one, although it should be noted that the CakeML stack brings verification down to the level of technology-mapped netlists [103], as mentioned in 3.2.4.

Neither the CLI nor the Verisoft nor the CakeML stack are concerned with security properties. Nonetheless, they consider the related problem of functional verification of hardware. Techniques that apply to this problem usually also apply to the verification of security properties. In the case of the CLI and of the CakeML stack, it is unclear whether the proposed methods could scale beyond the simple examples they consider. On the other hand, the methodology employed in the Verisoft stack does not suffer from this concern, as the complexity of the example it is applied validates that it indeed scales. The CLI and the Verisoft stacks are both inactive projects, with the latest hardware artifacts having been released around two decades ago.

Chapter recap and take-away

This section gave an overview of the state of the art of the different domains relevant to this work. The key takeaway is that a trade-off exists between user-friendly yet limited automatic methods and more complex, general techniques based on proof assistants. This divide is not fundamental: automatic tools can generate proofs compatible with proof assistants, bringing the benefits of automation to these high-trust environments. Indeed, we advocate for such proof assistant-based, solver-assisted approaches throughout this thesis. We believe these approaches are the best option for the task at hand, as they allow users to focus on the high-level structure of the proof, abstracting over the rote mechanical parts.

Chapter 4
Proving properties of Kôika designs

The formal verification of a piece of hardware at the microarchitectural level calls for a formal semantics of the HDL in which it is formulated. Alas, the HDLs that are prevalent in the industry (such as Verilog or VHDL) were not designed with formal methods in mind. To achieve our goal of verifying security properties of processors at the microarchitectural level, we could either retrofit a formal semantics onto a non-formal language or start from a formally defined language. The former option would allow us to target one of the main languages of the industry. However, the a posteriori formalization of the semantics of such languages is a complex endeavor (although there is prior work about this [141], [142], [143], there is no complete formal semantics for these languages). The latter option is more direct than the former. Starting from a language with a formally defined semantics, all that is left to do is to import a design and verify whatever properties we set our sights on. Consequently, this approach allows us to directly identify the challenges of formal verification at such a scale and gives us insight into how an efficient formal semantics could be designed for an industrial-grade HDL.

This chapter is about our experience with this latter approach. We start from Kôika [55], an HDL with a formal semantics and a verified compiler down to some subset of Verilog. Kôika is embedded within the Coq proof assistant. Conveniently, it is shipped with an implementation of a pipelined RV32I processor. We describe this language and how we extend it to make it more amenable to proofs, using the verification of the processor mentioned above augmented with a hardware-based shadow stack as our running example.

Chapter outline

We introduced the Kôika language and its semantics back in Section 2.5, mentioning challenges that must be overcome for formal verification. Section 4.1, the central part of this chapter, discusses how we had to modify Kôika to make it better suited to formal reasoning about hardware. Section 4.2 is a concrete showcase of an application of the methodology we implemented to verify a processor extended with a hardware-based shadow stack. In Section 4.3, we present an extension of the verification process that includes automatic procedures to reduce the reliance on user guidance. Finally, Section 4.4 discusses the limitations of this project and future work. We close this chapter in Section 4.5.

4.1 A methodology for verifying Kôika designs

We propose a low-level representation, which we call an Intermediate Representation for Reasoning (IRR). This representation allows us to circumvent the limitations exposed in the previous section. With it, rules cancellation and conflict resolution become explicit. While this produces rather large terms, reifying these computations makes simplifying and storing them possible. This way, for instance, the condition describing when a rule is canceled is computed precisely once rather than each time we consider the value of any register. The subterms corresponding to registers impacted by a rule would then reference the subterm corresponding to its cancellation condition to check whether or not to apply its effects.

Section outline

This section describes this alternative semantics and how it fits in the bigger picture of the formal verification of properties of Kôika designs. Subsection 4.1.1 starts by giving a high-level overview of our proof methodology before discussing the representation in detail in Subsection 4.1.2. In Subsection 4.1.3, we describe our compilation pass from Kôika designs down to this alternative representation. The compiler is verified, and we describe our semantics preservation proof in Subsection 4.1.4. Finally, Subsection 4.1.5 presents our tooling for working with IRR terms.

4.1.1 A high-level overview

Our methodology for proving security properties on Kôika designs is illustrated in Figure 4.1. Starting from the Kôika design, we obtain the first IRR, IRR$_0$, from a verified compiler that we describe in Subsection 4.1.3. IRR$_0$ is then transformed into

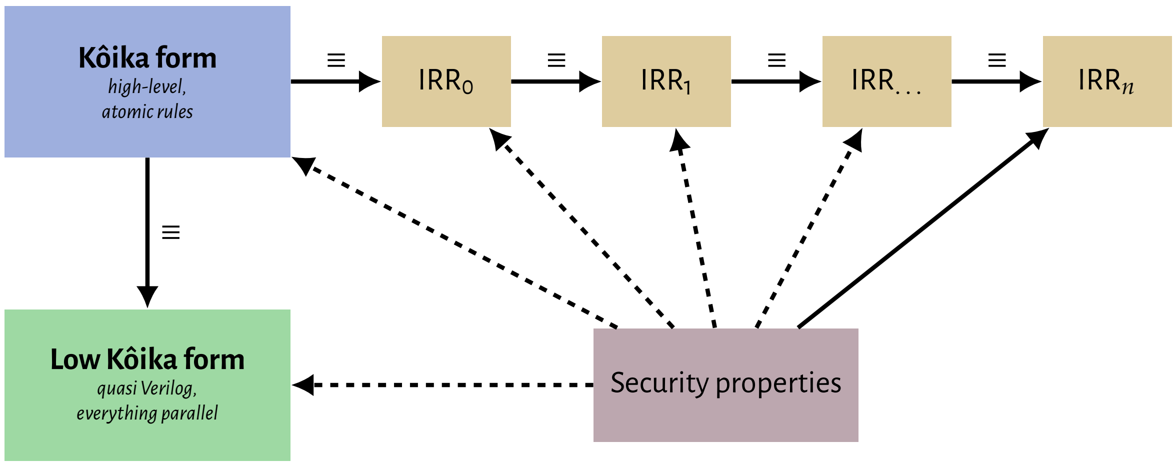


Figure 4.1: Overall structure of our proofs on Kôika designs

a sequence of IRRs, each simpler to reason about than the previous one. Applying these transformations passes is part of the proof development process. It can be done manually or through tactics that automatically apply appropriate passes depending on the form of the goal and the hypotheses. Users can easily define additional tactics tailored to their work using Coq's built-in facilities. We describe these transformations in Subsection 4.1.5. Each transformation pass is formally verified. Therefore, the proofs we carry out on the transformed IRRs also hold for the initial IRR and the initial Kôika design. Starting from the initial IRR$_0$, we repeatedly apply transformation passes until we end up with an IRR$_n$ that is amenable to a classical proof of our security property. Finally, from Kôika's compiler correctness, we know that the security property holds for the Verilog code that will be simulated or synthesized.

4.1.2 An intermediate representation for reasoning

We describe the syntax and semantics of our alternative representation in Figure 4.2: an Intermediate Representation for Reasoning (IRR) is composed of a variable map V, which maps variable identifiers to IRR actions, and a mapping R from register names to variable identifiers. We use natural numbers for the variable identifiers, although any other type equipped with a strict order relation would do. $R[r]$ holds the variable identifier corresponding to the value of register r after one clock cycle, whose contents can be recovered from the variable map V.

An IRR action is a static counterpart to Kôika actions. These IRR actions can be con-

IRR actions

$$\begin{array}{rcl} \mathit{irra} & ::= & v \in \mathit{val} \mid x \mid r \mid \;\triangleright \mathit{irra} \mid \mathit{irra} \bowtie \mathit{irra} \\ & \mid & \mathit{irra} \;?\; \mathit{irra} : \mathit{irra} \end{array}$$

IRR objects

$$\mathit{irr} \;::=\; \{V : \mathbb{N} \hookrightarrow \mathit{irra}\;;\; R : \mathtt{reg} \to \mathbb{N}\}$$

Semantics of IRR actions

$$\begin{array}{rcl} [\![v]\!]^{\mathit{irr}}_{\sigma} & := & v \quad (v \in \mathit{val}) \\ [\![x]\!]^{\mathit{irr}}_{\sigma} & := & [\![a]\!]^{\mathit{irr}}_{\sigma} \text{ if } \mathit{irr}.V(x) = a \\ [\![r]\!]^{\mathit{irr}}_{\sigma} & := & \sigma(r) \\ [\![\triangleright \mathit{irra}]\!]^{\mathit{irr}}_{\sigma} & := & \triangleright [\![\mathit{irra}]\!]^{\mathit{irr}}_{\sigma} \\ [\![\mathit{irra}_1 \bowtie \mathit{irra}_2]\!]^{\mathit{irr}}_{\sigma} & := & [\![\mathit{irra}_1]\!]^{\mathit{irr}}_{\sigma} \bowtie [\![\mathit{irra}_2]\!]^{\mathit{irr}}_{\sigma} \\ [\![\mathit{irra}_1 \;?\; \mathit{irra}_2 : \mathit{irra}_3]\!]^{\mathit{irr}}_{\sigma} & := & [\![\mathit{irra}_1]\!]^{\mathit{irr}}_{\sigma} \neq \vec{0} \;?\; [\![\mathit{irra}_2]\!]^{\mathit{irr}}_{\sigma} : [\![\mathit{irra}_3]\!]^{\mathit{irr}}_{\sigma} \end{array}$$

Figure 4.2: Syntax and semantics of IRR actions

stant values (v), variables (x), registers (r), unary operations ($\triangleright\mathit{irra}$), binary operations ($\mathit{irra} \bowtie \mathit{irra}$), or conditional expressions ($\mathit{irra} ? \mathit{irra} : \mathit{irra}$), as illustrated in Figure 4.2. Compared with Kôika actions, the main difference is that register read and write operations have disappeared. Indeed, IRR actions are free of side effects.

IRR variables do not always represent values of registers or let-bound variables. Instead, they can be arbitrary expressions. For instance, a variable may be defined to store some recurring subterm. Then, instances of the subterm in question in other variables could be replaced with references to this variable, which would help keep the memory footprint in check. The IRR action associated with a variable x in our IRR can reference only variables whose identifier y is strictly below x. This well-formedness condition ensures that there is no cyclic dependency between variables. Hence, the evaluation of these expressions terminates. We can therefore define the evaluation function $[\![\mathit{irra}]\!]^{\mathit{irr}}_{\sigma}$, which evaluates an IRR action *irra* into a value. It is parameterized by an IRR *irr* (for resolving variables) and an environment σ (that holds the initial values of registers). The definition is given in Figure 4.2.

Let us consider, for instance, the action $\mathtt{a} + v7$. Its evaluation $[\![\mathtt{a} + v7]\!]^{\mathit{irr}}_{\sigma}$ would be decomposed into $[\![\mathtt{a}]\!]^{\mathit{irr}}_{\sigma} + [\![v7]\!]^{\mathit{irr}}_{\sigma}$. The first part $[\![\mathtt{a}]\!]^{\mathit{irr}}_{\sigma}$, i.e. the evaluation of register a will simply be a lookup into the environment: $\sigma(\mathtt{a})$. The second part will look up variable $v7$ in the IRR and then recursively apply the evaluation function on the IRR action associated with $v7$.

Although the representation we introduce is tailored to languages of the Bluespec family, similar representations can be more generally applied to control partial interpretation in Coq. Indeed, there are many similar situations that Coq's built-in interpretation tactics cannot handle conveniently. Making everything explicit helps with making the performance cost of tactics predictable. In fact, IRR is very close to gated static single assignment form [144], a representation with applications in static analysis and compiler optimization passes.

4.1.3 From Kôika rules to IRR

Our first objective is to build an IRR object from a Kôika design. We handle each rule in the Kôika design according to its scheduling order. For each rule, we perform a kind of abstract interpretation of the action, as shown in the definition of $K2I$ in Figure 4.3. The parameters of the compiler are:

- The Kôika action a to be compiled
- A projection $\Pi : \texttt{Reg} + \texttt{Var} \longrightarrow \mathbb{N}$ that maps Kôika variables and registers to IRR variable identifiers
- A mapping $V : \mathbb{N} \longrightarrow \mathit{irra}$ from variable identifiers (natural numbers) to IRR actions
- The next fresh variable identifier f
- P, an IRR action representing the condition guarding the current path
- An abstract schedule log $L^\sharp$
- An abstract rule log $l^\sharp$

The result of compiling an action with $K2I$ is a 6-tuple $(\mathit{irra}, \Pi', V', f', F, l'^\sharp)$ where *irra* is the IRR action corresponding to the input Kôika action, Π', V', f' and $l'^\sharp$ are the updated values of Π, V, f and $l^\sharp$, and F is the action's failure condition, i.e., an IRR action expressing the conditions under which the current action will fail. The abstract logs record all the potential read and write events, together with an IRR action representing the condition under which these read and write events occurred. More precisely, abstract logs ($L^\sharp$ and $l^\sharp$) are of type $\{\texttt{wr}, \texttt{rd}\} \times \texttt{Reg} \longrightarrow \mathit{irra}$. For example, $l^\sharp(\texttt{wr}, r)$ is an IRR action that represents the conditions under which a write has occurred on register r. The logs are updated when compiling register reads and writes (see Figure 4.3): the notation $l^\sharp[(k,r) \overset{\vee}{\mapsto} P]$ means the mapping $l^\sharp$ updated for key (k,r), where the new value is $\mathit{old} \vee P$, where *old* is the old value of the mapping for that key. The **may_read**$^\sharp$

```
$K2I(a, \Pi, V, f, P, L^\sharp, l^\sharp) =$
 match $a$ with
 | $v \rightarrow (v, \Pi, V, f, \texttt{false}, l^\sharp)$
 | $x \rightarrow$
   let $v = \Pi(x)$ in $(v, \Pi, V, f, \texttt{false}, l^\sharp)$
 | $\texttt{read}\ r \rightarrow$ let $v = \Pi(r)$ in
   let $mr$ = may_read$^\sharp(L^\sharp, r)$ in
   $(v, \Pi, V, f, \neg mr, l^\sharp[(\texttt{rd}, r) \overset{\vee}{\mapsto} P])$
 | $\texttt{write}\ r\ a \rightarrow$
   let $(irra, \Pi, V, f, F_a, l^\sharp) =$
     $K2I(a, \Pi, V, f, P, L^\sharp, l^\sharp)$
   in
   let $mw$ = may_write$^\sharp(L^\sharp, l^\sharp, r)$ in
   let $(V, \Pi, f) =$
     $(V[f \mapsto irra], \Pi'[r \mapsto f], f + 1)$
   in
   $(\bot, \Pi, V, f, F_a \vee \neg mw, l^\sharp[(\texttt{wr}, r) \overset{\vee}{\mapsto} P])$
 | $x := a \rightarrow$
   let $(irra, \Pi, V, f, F, l^\sharp) =$
     $K2I(a, \Pi, V, f, P, L^\sharp, l^\sharp)$ in
   let $(\Pi, V) = (\Pi[x \mapsto f], V[f \mapsto irra])$ in
   $(\bot, \Pi, V, f + 1, F, l^\sharp)$
 | $\texttt{if}\ c\ \texttt{then}\ ta\ \texttt{else}\ fa \rightarrow$
   let $(irra_{cond}, \Pi, V, f, F_{cond}, l^\sharp) =$
     $K2I(c, \Pi, V, f, P, L^\sharp, l^\sharp)$ in
   let $(v_{cond}, f) = (f, f + 1)$ in
   let $V = V[v_{cond} \mapsto irra_{cond}]$ in
   let $(irra_{tb}, \Pi_{tb}, V, f, F_{tb}, l^\sharp) =$
     $K2I(ta, \Pi, V, f, P \wedge v_{cond}, L^\sharp, l^\sharp)$
   in
   let $(irra_{fb}, \Pi_{fb}, V, f, F_{fb}, l^\sharp) =$
     $K2I(fa, \Pi, V, f, P \wedge \neg v_{cond}, L^\sharp, l^\sharp)$
   in
   let $(\Pi_{merge}, V, f) = \Pi_{tb} \bigcup_{V,f}^{v_{cond}} \Pi_{fb}$ in
   $(irra_{cond}\ ?\ irra_{tb} : irra_{fb}, \Pi_{merge}, V, f,$
     $F_{cond} \vee (irra_{cond}\ ?\ F_{tb} : F_{fb}), l^\sharp)$
 | $\texttt{let}\ x = a\ \texttt{in}\ body \rightarrow$
   let $(irra, \Pi, V, f, F_a, l^\sharp) =$
     $K2I(a, \Pi, V, f, P, L^\sharp, l^\sharp)$ in
   let $(\Pi, V) = (\Pi[x \mapsto f], V[f \mapsto irra)$ in
   let $(irra_{body}, \Pi, V, f, F_{body}, l^\sharp) =$
     $K2I(body, \Pi, V, f + 1, P, L^\sharp, l^\sharp)$ in
   $(irra_{body}, \Pi[x \mapsto \bot], V, f, F_a \vee F_{body}, l^\sharp)$
```

```
$K2I^{sched}(s, \Pi, V, f, L^\sharp) =$
 match $s$ with
 | $[] \rightarrow (\Pi, V, f, L^\sharp)$
 | $r :: s \rightarrow$
   let $(\_, \Pi', V, f, F_r, l^\sharp) =$
     $K2I(r, \Pi, V, f, \texttt{true}, L^\sharp, l_0^\sharp)$ in
   let $(v_{conflict}, f) = (f, f + 1)$ in
   let $V = V[v_{conflict} \mapsto F_r]$ in
   let $(\Pi_{merge}, V, f) = \Pi \bigcup_{V,f}^{v_{conflict}} \Pi'$ in
   let $L_{merge}^\sharp =$
     $\lambda k \rightarrow L^\sharp(k) \vee (l^\sharp(k) \wedge \neg v_{conflict})$
   in
   $K2I^{sched}(s, \Pi_{merge}, V, f, L_{merge}^\sharp)$
```

```
$K2I^P(s) =$
 let $(\Pi, V, \_, \_) =$
   $K2I^{sched}(s, \Pi_0, V_0, 1, L_0^\sharp)$
 in
 $\{\ R := \texttt{fun}\ r \Rightarrow \Pi(r);\ V := V\ \}$
```

Figure 4.3: The Kôika to IRR compiler

and $\textbf{may\_write}^\sharp$ are the abstract counterpart to the **may_read** and **may_write** in Kôika semantics. For instance, $\textbf{may\_read}(L, r) = \textbf{wr}(r, *) \notin L$ (see Figure 2.14). The abstract version $\textbf{may\_read}^\sharp(L^\sharp, r)$ is defined as the IRR action $\neg L^\sharp(\textbf{wr}, r)$. The condition under which a read action fails is exactly the negation of the result of the $\textbf{may\_read}^\sharp$ function. For an action **write** $r\ a$, this condition is the logical disjunction between the negation of $\textbf{may\_write}^\sharp$ and the failure condition associated with action a.

The projection Π maps Kôika variables and registers to IRR variable identifiers. More precisely, for each register r, $\Pi(r)$ corresponds to the variable holding the final value of register r at the end of the current cycle. Initially, $\Pi(r)$ maps to a variable containing only the IRR action r. When compiling a register write (see Figure 4.3), the projection Π is updated so that $\Pi(r)$ points to the new value, per the semantics of Kôika.

One of the most involved cases for $K2I$ is the management of conditional expressions. We construct an IRR expression for each case of the conditional expression: one when the condition evaluates to `true` and another when the condition evaluates to `false`. This process is materialized by the two recursive calls to $K2I$, one with the path condition augmented with the condition, the other with its negation. These two calls generate two distinct projections Π_{tb} and Π_{fb}, that we merge using the binary operator $\bigcup_{V,f}^{v_{cond}}$. For every Kôika variable or register k, suppose $\Pi_1(k) = v_1$ and $\Pi_2(k) = v_2$, then $\Pi_1 \bigcup_{V,f}^{v_{cond}} \Pi_2$ will be a triple (V', Π_r, f') such that $\Pi_r(k) = n$ and $[\![n]\!]_\sigma^{V'} = [\![v_{cond}\,?\,v_1 : v_2]\!]_\sigma^{V'}$. For instance, let us consider the first conditional expression in rule `r1` from Figure 4.4a. Recursively analyzing the **then** and **else** branches yield two different IRRs: one when `read a == 0` holds and register `a` receives the value 1, the other where register a keeps its previous value. The merged IRR will associate to register a the IRR action $(a == 0)\ ?\ 1 : a$.

Next, depending on whether conflicts occur, the effects of rules are discarded or applied. The function $K2I^{sched}$ compiles a Kôika schedule into an IRR. Each rule is treated sequentially. For each rule, we call $K2I$ with the current projection Π and the current set of variables V, the current abstract schedule log $L^\sharp$ and an empty initial rule log $l_0^\sharp$. The failure condition F_r gives us the condition under which the rule failed. Like in the conditional expression case, we need to merge the projections using $\bigcup_{V,f}^{v_{conflict}}$. We also need to patch the abstract schedule log $L_{merge}^\sharp$ as described in Figure 4.3 so that the condition under which a read or write occurred is the disjunction of whether it already occurred in the previous schedule log $L^\sharp$ and whether it occurred in the new rule log $l^\sharp$, while not having a conflict. This step was unnecessary in the if-then-else case because we injected the condition (or its negation) in the path condition, which we cannot do here. After all, the failure condition is the result of the compiler itself.

Figures 4.4b and 4.4c illustrate a complete example of compiling a Kôika schedule into an IRR. Initially, we have one variable for each of the three registers a, b and c. The let-bound variable x in rule `r1` gives rise to $v4$ in the IRR, whose value is $v2$, the initial value of register b. The conditions of if-then-else actions also produce variables ($v5$, $v8$ and $v13$ in our example). For each register write, a variable is created that contains the IRR action being written ($v6$, $v9$ and $v14$), and yet another variable is created that contains the new value of the register, often an if-then-else IRR action dependent on the path condition that led us to the write ($v7$, $v10$, and $v15$). At the end of rule `r1`, $v11$ contains the rule's failure condition, i.e., $v8 \wedge v5$, i.e., "there have been two writes on a"; and $v12$ contains the value of register `a` after rule `r1`: "if the rule fails, the value of `a` is the same as at the beginning of the cycle, otherwise it is the value after the second **`if`**". Similarly, $v16$ contains the failure condition for rule `r2`: "the rule fails if a write occurred in this rule (condition $v13$) and a write occurred in rule `r1` ($\neg v11 \wedge (v5 \vee v8)$, i.e. rule `r1` did not fail and one of the writes occurred)".

4.1.4 Correctness of the Kôika to IRR compiler

Before reasoning on the IRR, we must ensure that our compiler is correct. This section walks through a series of definitions and lemmas that culminate in the proof of Theorem 2, which enables one to reason about an IRR to obtain properties of a Kôika design.

We first define a relation $\overset{log}{\sim}$ between a concrete and an abstract log, establishing that a read or write event occurs in the concrete log if and only if the condition associated with that event in the abstract log evaluates to true.

$$L \overset{log}{\sim}_{V} L^{\sharp} \triangleq \begin{cases} \forall r,\ \mathtt{wr}(r,*) \in L \Longleftrightarrow [\![L^{\sharp}(\mathtt{wr},r)]\!]_{\sigma}^{V} = \mathtt{true} \\ \forall r,\ \mathtt{rd}(r) \in L \Longleftrightarrow [\![L^{\sharp}(\mathtt{rd},r)]\!]_{\sigma}^{V} = \mathtt{true} \end{cases}$$

We also define a relation $\overset{reg}{\sim}$ between a concrete log and an IRR. This relation states that the projection Π sends registers to IRR variables that represent their value (as returned by a read in a Kôika action). This is captured by the $\mathtt{do\_read}(l, \sigma, r)$ function, which returns a value v if $\mathtt{wr}(r,v) \in l$, or $\sigma(r)$ if no such write occurred.

```
Registers : {a, b, c}.

Rule r1 :
  let x := read b in
  if read a == 0
    then write a 1;
  if x == 1
    then write a (x + 1).

Rule r2 :
  if read c == 1
    then write a 3.

Schedule : [r1, r2].
```

(a) Kôika design

Register	Variable Id
a	17
b	2
c	3

(b) IRR representation (R)

Id	Value	Description
1	a	initial value of register a
2	b	initial value of register b
3	c	initial value of register c
4	$v2$	let-binding in r1
5	$v1 == 0$	first condition in r1
6	1	value written in the first `if` in r1
7	$v5 \, ? \, v6 : v1$	value of register a after first `if` in r1
8	$v4 == 1$	second condition in r1
9	$v4 + 1$	value written in the second `if` in r1
10	$v8 \, ? \, v9 : v7$	value of register a after second `if` in r1
11	$v8 \wedge v5$	failure condition for r1
12	$v11 \, ? \, v1 : v10$	value of register a after r1
13	$v3 == 1$	first condition in r2
14	3	value written in the `if` in r2
15	$v13 \, ? \, v14 : v12$	value of register a after the `if` in r2
16	$v13 \wedge \neg v11 \wedge (v5 \vee v8)$	failure condition for r2
17	$v16 \, ? \, v12 : v15$	final value of register a after r2

(c) IRR representation (V)

Figure 4.4: Compilation from Kôika to IRR

$$l \overset{reg}{\sim} (\Pi, V) \triangleq \ \forall r,\ \exists n,\ \Pi(r) = n \land [\![n]\!]^V_\sigma = \texttt{do\_read}(l, \sigma, r)$$

We now define a matching relation between Kôika states (Γ, L, l) (variable environment, schedule log and rule log) and IRR states $(V, L^\sharp, l^\sharp)$ (variable mapping, abstract schedule log and abstract rule log). This relation, denoted $\sim_\Pi$, is indexed by a projection Π and states that the concrete and abstract logs are related by $\overset{log}{\sim}_V$, both for the schedule and rule logs; that the projection Π sends Kôika variables to IRR variables that evaluate identically; and that the registers are accurately projected by Π, as per the $\overset{reg}{\sim}$ relation.

$$(\Gamma, L, l) \sim_\Pi (V, L^\sharp, l^\sharp) \triangleq$$
$$\begin{cases} L \overset{log}{\sim}_V L^\sharp \\ l \overset{log}{\sim}_V l^\sharp \\ \forall x,\ \exists n,\ \Pi(x) = n \land [\![n]\!]^V_\sigma = \Gamma(x) \\ (L \mathbin{++} l) \overset{reg}{\sim} (\Pi, V) \end{cases}$$

We prove the following lemma about the correctness of the *K2I* function, which is the essence of the proof of the compiler correctness theorem.

Lemma 1 (Correctness of *K2I*)**.** *Consider two matching states* (Γ, L, l) *and* $(V, L^\sharp, l^\sharp)$ *related by projection* $\sim_\Pi$*. Compiling action a with path condition P produces a IRR action irra together with a new abstract state* $(V', L^\sharp, l'^\sharp)$*, and a failure condition F.*

If the Kôika semantics of action a produces a new Kôika state (Γ', v, l')*, then the IRR action irra produced by the compiler evaluates to Kôika value v, the failure condition F evaluates to* `false`*, and* (Γ', L, l') *and* $(V', L^\sharp, l'^\sharp)$ *stay in the matching relation.*

Otherwise, if action a fails according to Kôika semantics, the failure condition produced by the compiler evaluates to `true`*. More formally,*

$$\begin{aligned}
\forall\ \ & a\ \Pi\ V\ f\ L^\sharp\ l^\sharp\ \Gamma\ L\ l\ irra\ P\ F\ \Pi'\ V'\ f'\ l'^\sharp\ \Gamma'\ \sigma, \\
& K2I(a, \Pi, V, f, P, L^\sharp, l^\sharp) = (irra, \Pi', V', f', F, l'^\sharp) \Rightarrow \\
& (\Gamma, L, l) \sim_\Pi (V, L^\sharp, l^\sharp) \Rightarrow [\![P]\!]^V_\sigma = \texttt{true} \Rightarrow \\
& \left(\begin{aligned} \forall\ \ & l'\ v\ \Gamma', \\ & \Gamma \vdash_L (l, a) \downarrow (l', v, \Gamma') \Rightarrow \\ & [\![irra]\!]^{V'}_\sigma = v \land [\![F]\!]^{V'}_\sigma = \texttt{false} \land \\ & (\Gamma', L, l') \sim_{\Pi'} (V', L^\sharp, l'^\sharp) \end{aligned} \right) \\
& \bigwedge \left(\Gamma \vdash_L (l, a) \not\downarrow\ \Rightarrow [\![F]\!]^{V'}_\sigma = \texttt{true} \right)
\end{aligned}$$

Proof. By structural induction on a. □

We use this lemma to prove Theorem 1 about $K2I^{sched}$:

Theorem 1 (Correctness of $K2I^{sched}$). *$K2I^{sched}$ is a correct compiler of the semantics of Kôika schedules. More precisely,*

$$\begin{array}{ll}\forall & s\ \Pi\ V\ f\ L^\sharp\ \Pi'\ V'\ f'\ L'^\sharp,\\ & K2I^{sched}(s,\Pi,V,f,L^\sharp) = (\Pi',V',f',L'^\sharp) \Rightarrow\\ & (L,s) \Downarrow L' \Rightarrow\\ & L \overset{log}{\sim}_V L^\sharp \wedge L \overset{reg}{\sim} (\Pi,V) \Rightarrow\\ & L' \overset{log}{\sim}_{V'} L'^\sharp \wedge L' \overset{reg}{\sim} (\Pi',V')\end{array}$$

Proof. By induction on the schedule s.

Base case. If the schedule is empty, the theorem holds trivially because $(\Pi', V', L'^\sharp) = (\Pi, V, L^\sharp)$ and $L' = L$.

Inductive case. The schedule is of the form $r::sch$, and we have as an induction hypothesis that our theorem holds for schedule sch. By the definition of $K2I^{sched}$, we have that:

1 $(\_,\Pi_1,V_1,f_1,F_r,l^\sharp) = K2I(r,\Pi,V,f,\texttt{true},L^\sharp,l_0^\sharp)$

2 $V_2 = V_1[f_1 \mapsto F_r]$

3 $(\Pi_m,V_m,f_2) = \Pi \bigcup^{f_1}_{V_2,f_1+1} \Pi'$

4 $L_m^\sharp = \lambda k \longrightarrow L^\sharp(k) \vee (l^\sharp(k) \wedge \neg v_{conflict})$

5 $(\Pi',V',f',L'^\sharp) = K2I^{sched}(sch,\Pi_m,V_m,f_2,L_m^\sharp)$

From the semantics of a Kôika schedule (Figure 2.12), the next concrete schedule log, L', will be either $L{+}{+}l'$ if the execution of r yields a rule log l', or L if the execution of a fails to produce a rule log. To apply our induction hypothesis and finish the proof, all we need to show is that $L' \overset{log}{\sim}_{V_m} L_m^\sharp \wedge L' \overset{reg}{\sim} (\Pi_m, V_m)$.

Applying Lemma 1 on line 1 above gives the following:

$$\begin{array}{l}\forall\ \ L\ V\ L^\sharp\ \Pi\ r\ \Pi_1\ V_1\ \Gamma_0\ \sigma,\\ L \overset{log}{\sim}_V L^\sharp \wedge L \overset{reg}{\sim} (\Pi,V) \Rightarrow\\ \left(\begin{array}{l}\forall\ \ l'\ l^\sharp,\ \Gamma_0 \vdash_L (l_0,r) \downarrow (l',\_,\_) \Rightarrow\\ \quad [\![F_r]\!]^{V_1}_\sigma = \texttt{false} \wedge\\ \quad l' \overset{log}{\sim}_{V_1} l^\sharp \wedge (L \mathbin{++} l') \overset{reg}{\sim} (\Pi_1,V_1)\end{array}\right)\\ \bigwedge \left(\ \Gamma_0 \vdash_L (l_0,r) \not\downarrow\ \Rightarrow [\![F_r]\!]^{V_1}_\sigma = \texttt{true}\ \right)\end{array}$$

By disjunction of cases:

- **Case** $\Gamma_0 \vdash_L (l_0, r) \downarrow (l', \_, \_)$.
  Then, $[\![f_1]\!]_\sigma^{V_2} = \texttt{false}$, hence $\Pi_m(k) = \Pi'(k)$ for every k, and $[\![L_m^\sharp(k)]\!]_\sigma^{V_m} = [\![L^\sharp(k) \vee l^\sharp(k)]\!]_\sigma^{V_m}$ for every k. It follows that $(L{+}{+}l') \overset{log}{\sim}_{V_m} L_m^\sharp$, and $(L{+}{+}l') \overset{reg}{\sim} (\Pi_m, V_m)$.
- **Case** $\Gamma_0 \vdash_L (l_0, r) \not\downarrow$.
  Then, $[\![f_1]\!]_\sigma^{V_2} = \texttt{true}$, hence $\Pi_m(k) = \Pi(k)$ for every k, and $[\![L_m^\sharp(k)]\!]_\sigma^{V_m} = [\![L^\sharp(k)]\!]_\sigma^{V_m}$ for every k. It follows that $L \overset{log}{\sim}_{V_m} L_m^\sharp$, and $L \overset{reg}{\sim} (\Pi_m, V_m)$.

□

Finally, the theorem we want to prove relates the interpretation of a cycle and the IRR obtained by compiling the schedule. We define a function $C_\sigma(irr)$ which, given an IRR *irr* and an initial mapping for registers σ, results in an updated register environment σ'. $C_\sigma(irr)(r) \triangleq \textbf{let } n = irr.R(r) \textbf{ in } [\![n]\!]_\sigma^{irr}$

Theorem 2 (Kôika to IRR compiler correctness)**.** *Retrieving the final value of a register r through the IRR (compiled from a schedule s and an initial state of registers σ) or through the Kôika semantics gives identical results. More formally,*

$$\forall s\ \sigma\ r,\ C_\sigma(K2I^P(s))(r) = \texttt{interp\_cycle}(s, \sigma)(r)$$

Proof. By applying Theorem 1 and unfolding definitions. □

4.1.5 Verified transformation passes

We develop a toolbox of theorems and tactics in Coq for reasoning about circuits in our IRR. Among other things, we define a set of transformation passes that can be applied to a design in the IRR. In particular, the transformation passes provide a way of exploiting hypotheses about the state of the environment. A combination of simplifications and case analysis is often sufficient for discharging a goal. The reasoning is carried out inside of Coq, so the usual facilities and methods used in this language remain available at any time.

For instance, consider the design of Figure 4.5a, composed of a single rule. In this rule, register `b`'s value is updated on every cycle. Suppose we want to prove that when the value of `a` is 0 at the beginning of a cycle, the value of `b` will be 0 at the end of this cycle. Formally, we would write this as:

$\forall\ \sigma\ irr,\ \sigma(\texttt{a}) == 0 \Rightarrow C_\sigma(irr)(\texttt{b}) = 0.$

We compile the design of Figure 4.5a into an equivalent IRR, as shown in the second column of Table 4.5b. This IRR is then processed by a sequence of transformations, which progressively simplify it. In this example, we employ the following transformation passes:

- `Prune`(b): remove all variables not used in the computation of the final value of register b. We collect all the variables that may participate in the evaluation of register b in our IRR and prune the others. Here, the register b is represented by $v8$. Variables $v6$ and $v7$ can be pruned.
- `ExploitReg`(r, v): replace register r with value v. Applying this transformation generates a proof obligation: register r holds value v at the beginning of the cycle. In our example, we have the hypothesis that register a holds the value 0.
- `Collapse`: replace all variables with their value if this symbolic value is simple enough (a constant, a reference to another variable, or a register). In our example, we replace variables $v1$, $v2$ and $v5$ by their value.
- `Simplify`: compute the value of unary or binary operations for all variables in the IRR, so long as enough of their arguments are known. This process is similar to constant folding in traditional compilers. We also simplify boolean conjunctions or disjunctions when one operand is known to be true or false.

All of our transformation passes preserve the semantics under some assumptions. For instance, assuming that the value of register `a` is known to be `0` at the beginning of a cycle, then replacing its appearances with `0` is correct. Coq verifies that this assumption is met in the current context before it can apply the transformation. Similarly, evaluating a register with an IRR in which some of the required variables have been pruned away would be invalid. Therefore, pruning can only be applied in contexts where the final value of a single register is being considered.

We offer additional transformation passes:

- `PruneList`(l): remove all variables that are not used in the computation of the final value of at least one register appearing in list l
- `ReplaceVar`(x, v): replace variable x with value v, given a proof that this substitution is correct
- `ReplaceSubact`(se, v): same as `ReplaceVar` but for a subexpression se. All occurrences of this expression se in any variable of the tree can then be replaced with the given value v.

```
Registers : {a, b}.

Rule r1 :
  let x := read a in
  let y := read b in
  if x == 0 then
    write b (y - y)
  else
    (write b 1; write a 2).

Schedule : [r1].
```

(a) Example rule

	Initial	Prune(reg b)	ExploitReg	Collapse	Simplify	Collapse + Simplify
1	a		0	▨	▨	▨
2	b			▨	▨	▨
3	$v1 == 0$			$0 == 0$	1	▨
4	$v2$ - $v2$			b - b		▨
5	1			▨	▨	▨
6	2	▨	▨	▨	▨	▨
7	if $v3$ then a else $v6$	▨	▨	▨	▨	▨
8	if $v3$ then $v4$ else $v5$			if $v3$ then $v4$ else 1		b - b

(b) Successive transformations

Figure 4.5: Example rule, its transformation into an IRR, and successive transformations

- `ExploitPartialInformation`: same as `ExploitReg` but works when only some bits of a register are known.

Coq makes it possible to define custom tactics to automate away part of the tedium. We define some general tactics that take our hypotheses into account and then attempt to simplify our design as much as possible, and that can even recognize some simple subgoals and solve those automatically. For simple properties, proofs can run entirely automatically.

Section recap

In this section, we presented our methodology for reasoning on Kôika designs. We had previously explained how Kôika's native semantics are not adapted to formal reasoning on large designs. Here, we presented a new, lower-level, and more explicit representation of Kôika designs that is better suited to this task. We call this representation an Intermediate Representation for Reasoning, or IRR for short. We presented our Kôika to IRR compiler and discussed how we formally verified that it is semantics-preserving. Finally, we went through different verified transformation passes for IRR that we defined and verified. These transformation passes can be applied to simplify the IRR design progressively. They lie at the core of our proving strategy.
In the next section, we will show how the IRR can be used in practice for verifying a hardware mechanism targeting a RISC-V processor.

4.2 Implementing and verifying a shadow stack for a RISC-V processor

This section illustrates the proof methodology we described in the previous section by showing how it can be used to verify a hardware-based shadow stack (a security mechanism described in Section 2.2.2.b) for an RV32I processor.

Kôika's original developers designed a simple pipelined RISC-V processor that can be specialized to cover part of the RV32I or the RV32E part of the standard. This design does not aim for exhaustiveness and is not proven to conform to the RISC-V specification (although it passes the RISC-V test suite for all the instructions it implements). It demonstrates some of Kôika's more advanced features. Our design (source) is a tweaked and extended version of this example.

Section outline

In Subsection 4.2.1, we describe how we expand Kôika's RV32I processor by fitting it with a simple shadow stack mechanism. The formal definition of the properties we set out to verify is given in Subsection 4.2.2. The proofs are described in the same subsection, following the methodology outlined in the previous section. We close this section with Subsection 4.2.3, a quantitative summary of the proof effort.

4.2.1 The shadow stack mechanism

We are interested in the property that functions return to the instruction following their call. A possible way of guaranteeing such a property is to maintain a shadow stack, as described in Subsection 2.2.2. As a reminder, the processor pushes the expected return address onto this stack for each function call and pops it whenever it returns. If the address a function tries to return to using the regular function stack is not equal to the one on top of the shadow stack, we can deduce that something went wrong and react accordingly[1]. Of course, we must also protect this shadow stack and prevent regular writes to memory performed by application code to modify the shadow stack contents.

Shadow stacks can be implemented in software [145] or hardware. Although software implementations provide some benefits (chief among them being their compatibility with existing hardware), we will focus on hardware implementations. These offer the advantage of working with any program with no patching necessary.

We added a shadow stack module to the processor provided by the Kôika project. We can prove its isolation from the core design in that the only way of acting on it is through its two methods, `push` and `pop`. These methods are called automatically when the current instruction corresponds to a function call or return. They both expect one argument: the address of the instruction following the current function call for `push` and the stack's return address for `pop`.

The `push` function checks for potential overflows and pushes the address onto the shadow stack, whereas the `pop` function checks for potential underflows, verifies that

[1]Note that shadow stacks are a reactive mechanism (see Section 2.2). When we turn to the verification of this mechanism, we are in effect following a preventive approach to get guarantees about a reactive mechanism.

the address passed as argument matches the top of the shadow stack, and pops it. If one of these checks fails, we jump to an exception handler.

4.2.1.a Detecting function calls and returns in machine code

Contrary to CISC (Complex Instruction Set Computer) ISAs such as x86, which have dedicated call and return instructions, RISC-V uses the same instruction for multiple purposes. This choice is common for RISC ISAs. The JAL (*jump and link*) and JALR (*jump and link register*) instructions implement both unconditional jumps and function calls. However, the arguments that are passed to them make their role clear. The Application Binary Interface (ABI) describes which JAL/JALR instructions should be interpreted as function calls or returns, depending on their arguments. The RISC-V specification [30] includes information regarding how shadow stacks (which they call return-address stacks) should behave:

> For RISC-V, hints as to the instructions' usage are encoded implicitly via the register numbers used. A JAL instruction should push the return address onto a return-address stack (RAS) only when rd = x1/x5. JALR instructions should push/pop a RAS as shown in the table [that follows].
>
> | rd | rs1 | rs1 = rd | RAS action |
> |---|---|---|---|
> | !*link* | !*link* | – | none |
> | !*link* | *link* | – | pop |
> | *link* | !*link* | – | push |
> | *link* | *link* | 0 | pop, then push |
> | *link* | *link* | 1 | push |
>
> Return-address stack prediction hints encoded in register specifiers used in the instruction. [. . .] *link* is true when the register is either `x1` or `x5`.

Hence, we implement our shadow stack so that we push a return address when the destination register is `x1` or `x5`, and we pop when the source register is `x1` or `x5` and the destination register is different from the source register.[2] Note that that for instructions such as `jalr x1, 0(x5)`, the shadow stack is popped before a new address is pushed onto it. We handle this accordingly.

[2]Registers `x1` and `x5` are also respectively known as `ra` (for return address) and `t0`.

4.2.1.b Dealing with a detected stack buffer overflow

On a system with a full-fledged operating system, a stack buffer overflow detected through a shadow stack mechanism could be left for the system to manage. For instance, the affected program could be killed, and an error could be logged or displayed to the user. In our simple embedded system, these options are not all open. The two main possibilities for our exception handler are:

- ending the current execution;
- correcting the return address using the shadow stack information (or just relying purely on it and ignoring return arguments).

The latter option might be tempting. However, it comes with significant downsides. If the return address has been modified, the rest of the stack will likely be impacted and cannot be considered safe.

Our verified stack implementation halts execution on a mismatch. In order to prove anything about our processor halting, we first need to define what this means for Kôika designs — this is tricky since Kôika does not have a notion of halting the execution of a design. We consider that a system is halted when it is in a sink state:

$$\texttt{is\_halted}(\sigma) \triangleq \forall n,\ r,\ C_\sigma^n(irr)(r) = \sigma(r)$$

where $C_\sigma^n(irr)$ performs n iterations of the C_σ function, i.e. computes the state of each register after n cycles. In fact, it suffices to demonstrate the following property to obtain a proof of `is_halted` for an environment.

$$\begin{aligned}&\texttt{nochange}(\sigma) \triangleq \forall r,\ C_\sigma(irr)(r) = \sigma(r)\\&\texttt{nochange\_halted}(\sigma) : \forall \sigma,\ \texttt{nochange}(\sigma) \Rightarrow \texttt{is\_halted}(\sigma)\end{aligned}$$

We add a register called `halt` to our processor, and we equip each of the rules with a guard checking the value of this register. When it is true, no rules are run. We can prove that it behaves as expected:

$$\texttt{halt\_1\_implies\_halted} : \forall \sigma,\ \sigma(\texttt{halt}) = 1 \Rightarrow \texttt{is\_halted}(\sigma)$$

Applying `nochange_halted` as well as transformation passes `ExploitReg`, `Simplify` and `Prune` gets us most of the way for this proof. We then need to show that the value of any individual register is left unchanged during the next cycle. At this stage, the demonstration of this property is trivial.

For simulation and synthesis, we emit an external call bound to a Verilog module, which halts the processor's execution whenever we set the value of `halt` to 1.

4.2.2 Formally verified properties

We are interested in proving four properties. In plain English, they may be worded as:

- "any overflow in the shadow stack leads to the immediate halting of the processor";
- "any underflow in the shadow stack leads to the immediate halting of the processor";
- "when returning from a procedure, if the stack and the shadow stack disagree on the return address, then the processor halts immediately";
- "in all other situations, the behavior of the processor remains unchanged".

Our processor now contains an additional variable (`sstack`) that stores the shadow stack data (a vector for the stack data and a variable for a count of how many items are currently stored on the stack). Hereafter are some helpful definitions related to this variable that we will use throughout our proof:

$$\begin{aligned}
\texttt{sstack\_empty}(\sigma) &\triangleq \sigma(\texttt{sstack.sz}) == 0\\
\texttt{sstack\_full}(\sigma) &\triangleq \sigma(\texttt{sstack.sz}) == \texttt{sstack.capacity}\\
\texttt{sstack\_top}(\sigma) &\triangleq\\
&\begin{cases} \emptyset & \text{if } \sigma(\texttt{sstack.sz}) \text{ is } 0\\ \sigma(\texttt{sstack.stack}[\sigma(\texttt{sstack.sz})]) & \text{otherwise}\end{cases}
\end{aligned}$$

Our processor is pipelined, which implies that several instructions are in-flight at the same time. Nonetheless, there is at most one instruction at the execute stage at any point, and all calls to shadow stack functions occur there.

Predicates `sstack_push` and `sstack_pop` express the conditions under which a push or a pop takes place. Their definitions amount to checking whether the instruction in the execute stage corresponds to a procedure call or return, as per the specification excerpt introduced in Subsection 4.2.1.a.

$$\begin{aligned}
\texttt{sstack\_push}(\sigma) &\triangleq \texttt{is\_call\_instruction}(\sigma(\texttt{d2e.dInst.inst}))\\
\texttt{sstack\_pop}(\sigma) &\triangleq \texttt{is\_ret\_instruction}(\sigma(\texttt{d2e.dInst.inst}))
\end{aligned}$$

Functions is_call_instruction and is_ret_instruction are in turn defined in Coq. For a more concrete example, the definition of is_call_instruction is given in full below (source):

```
Definition is_call_instruction (instr: bits_t 32) : bool :=
  let bits := vect_to_list (bits_of_value instr) in
  let opcode_ctrl := List.firstn 3 (List.skipn 4 bits) in
  let opcode_rest := List.firstn 4 (List.skipn 0 bits) in
  let rs1 := List.firstn 5 (List.skipn 15 bits) in
  let rd := List.firstn 5 (List.skipn 7 bits) in
  (eql opcode_ctrl (rev [true; true; false]))
  && (
    (
      (eql opcode_rest (rev [true; true; true; true]))
      && (
        (eql rd (rev [false; false; false; false; true]))
        || (eql rd (rev [false; false; true; false; true]))))
    || (
      (eql opcode_rest (rev [false; true; true; true]))
      && (
        (eql rd (rev [false; false; false; false; true]))
        || (eql rd (rev [false; false; true; false; true]))))).
```

The no_mispred construct is used for dealing with the mispredictions that can result from branch instructions. The effects of a mispredicted instruction have to be ignored. When an instruction reaches the execution stage of the pipeline, it is already known whether or not it belongs to a mispredicted branch and, therefore, whether or not it has to be ignored. The function candidate_return_address gives the address that the current instruction attempts to return to, assuming it is a procedure return.

We give formal definitions for the first three properties we mentioned earlier:

$$\begin{aligned}
&\texttt{sstack\_uflow}(\sigma) \triangleq \texttt{no\_mispred}(\sigma) \land \texttt{sstack\_empty}(\sigma) \land \texttt{sstack\_pop}(\sigma)\\
&\texttt{sstack\_oflow}(\sigma) \triangleq\\
&\quad \texttt{no\_mispred}(\sigma) \land \texttt{sstack\_full}(\sigma) \land \neg\texttt{sstack\_pop}(\sigma) \land \texttt{sstack\_push}(\sigma)\\
&\texttt{sstack\_violation}(\sigma) \triangleq\\
&\quad \texttt{no\_mispred}(\sigma) \land \texttt{sstack\_pop}(\sigma) \land \texttt{candidate\_return\_addr}(\sigma) \neq \texttt{sstack\_top}(\sigma)
\end{aligned}$$

Note that we explicitly mention both `sstack_pop` and `sstack_push` in `sstack_uflow`. Indeed, the chart presented in Subsection 4.2.1.a shows that for instructions such as `jalr x1, 0(x5)`, the shadow stack is popped before a new address is pushed onto it. This situation is the only one where something may get pushed to a stack that was full at the beginning of the cycle without running into an overflow.

We show that these three ways of violating the shadow stack policy result in the halting of the processor.

- `sstack_uflow_implies_halt`: $\forall\sigma$, `sstack_uflow`$(\sigma) \Rightarrow$ `is_halted`$(C_\sigma(irr))$
- `sstack_oflow_implies_halt`: $\forall\sigma$, `sstack_oflow`$(\sigma) \Rightarrow$ `is_halted`$(C_\sigma(irr))$
- `sstack_addr_violation_implies_halt`:
  $\forall\sigma$, `sstack_violation`$(\sigma) \Rightarrow$ `is_halted`$(C_\sigma(irr))$

The proofs of `sstack_uflow_implies_halt`, `sstack_oflow_implies_halt` and `sstack_addr_violation_implies_halt` share some similarities. In all of them, we start by applying `halt_1_implies_halted`. We must then prove that the final value of `halt` is 1. A logical first step is to apply the `Prune` transformation pass: we do not care about the variables irrelevant to the final value of `halt`. We can also exploit our hypotheses. For instance, for `sstack_underflow_implies_halt`, we know, among other things, that the shadow stack is empty and that the instruction in the execute stage of the pipeline corresponds to a return instruction. We can exploit this information to simplify the design further. Some case analysis is required to fully exploit the information about the executed instruction. Indeed, as was shown in Subsection 4.2.1.a, a pop should occur in two situations:

- when `rd` is neither `x1` or `x5` and `rs1` is either `x1` or `x5`;
- when both `rd` and `rs1` are `x1` or `x5` but `rd` $\neq$ `rs1`.

In all branches, the rest of the proof is trivial. The proofs of the two other properties follow a similar pattern.

There remains a last property to demonstrate. To prove that our shadow stack does not interfere with the rest of the processor, we need to show that, starting from the same environment and after one cycle, in the absence of a shadow stack violation, the value of the registers that were not introduced for the shadow stack is the same in the vanilla design as in the modified one.

Kôika	**48971**	LOC
…of which proof framework	18999	LOC
Processor	**8679**	LOC
…of which core	1478	LOC
…of which shadow stack	103	LOC
Properties and proofs	**1740**	LOC

Figure 4.6: Table of Lines Of Code (LOC) for different tasks

Formally, we write this as:

$$\begin{array}{l}
\texttt{sstack\_no\_interferences}: \\
\quad \forall\, \sigma,\ \sigma(\texttt{halt}) = 0 \\
\quad \Rightarrow \neg\texttt{sstack\_violation}(\sigma) \\
\quad \Rightarrow \neg\texttt{sstack\_uflow}(\sigma) \\
\quad \Rightarrow \neg\texttt{sstack\_oflow}(\sigma) \\
\quad \Rightarrow \forall\, r,\ (r \neq \texttt{sstack}[\cdot]) \\
\quad \Rightarrow C_\sigma(\mathit{irr\_basic})\ r = C_\sigma(\mathit{irr\_sstack})\ r
\end{array}$$

Once again, we start by exploiting some known values through the `ExploitReg` transformation and keep simplifying the design with `Simplify` and `Prune` (using the variant `PruneList` this time, with all the registers, except those that are related to the shadow stack).

The shadow stack is only ever accessed from the rule corresponding to the execute stage in our design. Furthermore, accesses to the shadow stack do not modify registers of the basic Kôika design except for `halt`. The only variables specific to the shadow stack-enabled version that remain in the IRR after the call to `PruneList` are those that impact `halt`. However, it can be shown that ¬`stack_violation` implies that the value written to `halt` is 0. In other words, the write in question does not update the value of `halt`. Therefore, some surgical applications of `ReplaceVar` allow us to remove precisely the parts that differ between our two designs. Then, our two IRRs are equal, and the goal is trivially true.

4.2.3 Quantitative summary

Figure 4.6 gives a breakdown of the code footprint of different parts of this work. This footprint, given in lines of code, considers the original Kôika code and our modifications.

Complete verification of the proofs takes slightly more than 10 minutes and uses a substantial amount of RAM (around 16GB) on a 12th Gen Intel i5-1240P processor running at 4.4GHz.

A considerable amount of human effort was put into verifying the shadow stack. Putting an exact figure on how much time this represented would be hard — work on the proof proceeded in parallel with work on the framework. Regardless, this verification process was more complex than we had hoped: in addition to being entirely manual, it necessitated a high degree of expertise. In Section 4.3, we will see how we remedied this issue through the integration of powerful automatic procedures that can handle the bulk of the proof obligations in practice.

4.2.4 Experimental evaluation

This section explains our experimental setup and how to use our verified hardware in practice.

Simulation of Kôika with Cuttlesim We verify the overall functional correctness of our modified RISC-V processor by running it on Cuttlesim, the C++ simulator provided by the Kôika project that directly interprets and simulates the Kôika language. We run a test suite that targets all the instructions in the RV32I subset separately. All the tests provided with the original Kôika still pass with our modified processor.

Experimental validation of the shadow stack Even with the proof of Section 4.1, it is valuable to test whether our shadow stack detects a trivial overwriting of a function's return address. This process cross-validates the theorems we proved earlier, which indeed entail a security property.

Figure 4.7 (which we already met as Figure 2.2) shows a C program that exhibits a trivial buffer overflow: buffer `buf` in function `f` is 16-byte long, and the `strcpy` function performs no bounds checking before copying the buffer `attack_buf`, which is 24-byte long. Because we know the memory layout of this program, we know where in `attack_buf` we should place the address we want to jump to so that it will overwrite the vulnerable function's return address. Our shadow stack module should detect this violation and halt our processor before we even jump to that new return address. We run this program with the Cuttlesim simulator, once with the shadow stack deactivated

```
void bad(){
  puts("Bad!\n");
}

int f(char* s){
  char buf[16];
  strcpy(buf,s);
}

int main(int argc, char* argv[]){
  int attack_buf[6];
  attack_buf[5] = (intptr_t)&bad;
  f((char*)attack_buf);
}
```

Figure 4.7: A program vulnerable to buffer overflows
This program overwrites its return address

Shadow stack	Clock frequency (MHz)	Used logical cells (out of 7680)	Critical path (ns)
Without	22.07	7049 (91%)	45.4
With	20.49	7463 (97%)	48.8

Figure 4.8: Impact of the addition of the shadow stack on performance metrics

and once with the shadow stack activated. In the first case, the buffer overflow succeeds, and we observe that the program writes the string "Bad!" to the console. On the other hand, when run with the shadow stack activated, we observe that the execution exits abruptly. By inspecting the final state of the Kôika design manually, we see that the halt register is set, and the instruction that was being executed at that point is precisely the ret instruction in the f function.

Synthesis on the TinyFPGA BX board We successfully ran the Verilog output of the Kôika compiler on our modified RISC-V processor through the Yosys synthesis suite. Figure 4.8 gives a rundown of the effect of this addition on different metrics. The shadow stack that we could fit on this board alongside the processor has a capacity of only seven return addresses and incurs an overhead of 5.9% in the number of logical cells (LUTs) used on the FPGA. This limitation in size is due to the limited number of LUTs and internal RAM of this inexpensive FPGA, as well as to the fact that Kôika does not provide a way to store data in the board's block RAM. As could be expected, the addition of a shadow stack has a moderate negative effect on the clock frequency as well as the length of the critical path.

Section recap

In this section, we illustrated our proof methodology on a complex example. We started with an existing RISC-V processor design and expanded it with a shadow stack mechanism. We then specified some properties that our design was required to enforce and proved that it indeed does enforce them. Although we successfully proved our design correct, this came at the cost of significant work.
In the upcoming section, we discuss some changes we made to our framework, extending it with automatic procedures to reduce the expenditure of human effort. As we will see, we will be able to get rid of thousands of manually-written tactic calls: instead of building proofs manually, we rephrase the properties we wish to verify in a way that is suitable for SMT-solvers and we let these tools handle the verification automatically.

4.3 SMT-based automation

This section describes how we extended our framework with automatic, SMT-based procedures to make verification more accessible.

Section outline

In Subsection 4.3.1, we start by discussing our motivation for adding automatic procedures to our framework alongside a brief overview of what this change would require and entail. We turn to a description of the actual changes to our framework in Subsection 4.3.2. Finally, we evaluate how our extended process behaves in practice in Subsection 4.3.3. In particular, we verify the shadow stack using the new process and compare this version with the previously established baseline.

4.3.1 Motivation and overview

The framework we built on top of Kôika provides a way of doing proof assistant-based hardware verification. However, the process we described up to this point was rather costly in terms of human effort. Although the proofs we presented were logically simple, they were not easy to express in a viable way: performance was the main limiting factor, but the overall verbosity did not help.

SMT-based procedures are an answer to both of these problems. Although SMT-solvers have limitations, they can be helpful for the more significant part of the proofs in which we may be interested. In an SMT-enriched proof process, the high-level structure of the proof can still be done in Coq, and the simpler branches can be delegated to the automatic procedure. This process would be a good fit for our hardware verification process.

Hybridizing the proof assistant with an unverified external tool increases the size of the TCB. There are ways of generating proof certificates from specific SMT-solvers. Such certificates can be checked by the Coq kernel, eliminating the TCB imprint. We have not gone this far in this work: we assume that the SMT-solvers are correct and trust their results.

4.3.2 Conversion to SMT form

We must turn Coq propositions about Kôika designs into SMT queries to interface with SMT-solvers. These forms differ quite starkly: Kôika is a high-level language (e.g., the conditions for rules cancellation are implicit), whereas SMT queries expect everything to be explicit. Luckily, we already tackled the problem of rendering a Kôika design fully explicit — this is the essence of the conversion to IRR.

Therefore, the first step in generating appropriate SMT queries is to convert the circuit to IRR form. IRR nodes are very close to what solvers take as input: they are built out of basic operations (all of them well-supported by solvers; e.g., addition or negation), conditionals, constants, and references to other nodes or registers (see Figure 4.2).

Before describing the process in detail, let us consider a concrete example: assume that we want to prove that the processor halts whenever there is an underflow (i.e., we handle pops from an empty shadow stack correctly). SMT-solvers take an expression containing free variables as input and are tasked with finding a valuation for these variables that makes this expression true. They answer "SAT" and show a concrete example when they succeed, and "UNSAT" to express the fact that no satisfying valuation exists. Remember that our design has a *halt* register, which is true if and only if the processor halts on the current cycle. Therefore, we can feed the solver a query along the lines of "there is a state of the processor where the shadow stack is empty and gets popped from, yet the final value of *halt* is false" (i.e., there is a state where our conditions are met yet the processor does not halt). Of course, assuming that underflows are detected correctly, this cannot be satisfied — we expect the solver to answer with an "UNSAT".

The IRR contains nodes representing every entity of interest in the system, such as components, cancellation conditions or branch guards. We previously let on that conversion of an IRR node to an SMT expression would be straightforward. This process is, in fact, so cheap that we can register all IRR nodes of the design as named expressions. References to registers in our query can be replaced with references to the expression generated from the appropriate nodes of the IRR. Registers whose value is not known are naturally represented as variables in the SMT formula, meaning that the system will attempt to find values for these registers so as to satisfy the property.

Beyond this, the conversion process for propositions is manual. We already gave an example of what this looks like in practice: in our underflow example, we kept our preconditions and negated the conclusion. More generally, an implication $A \Rightarrow B$ can be modeled as $\neg A \vee B$. We define aliases for implication and other constructs to make the conversion more natural.

Coq provides an extraction mechanism that can be used to generate OCaml objects from Coq code. We rely on this mechanism to get all the data relevant to our queries out in the real world. We then use OCaml to generate an SMT-LIB source file [146], [147] from the extracted data. These files can be handled by SMT solvers such as Z3 [148].

Figure 4.9 summarizes our SMT-based verification process. It shows that a user can automatically go from a property about a Kôika design to one about an IRR. We detailed the Kôika-to-IRR conversion previously. Since this conversion is verified, we can trivially rephrase properties about Kôika designs as properties about their corresponding IRRs. However, producing appropriate SMT queries remains the user's duty. As mentioned, the rest of the process is quite manual, although we provide a Makefile that automates most of it. Note that we do not have a way of importing the results of the solvers back into Coq — for the time being, the verification process is split into a Coq part and an external part. In effect, we add the SMT-solver to the TCB.

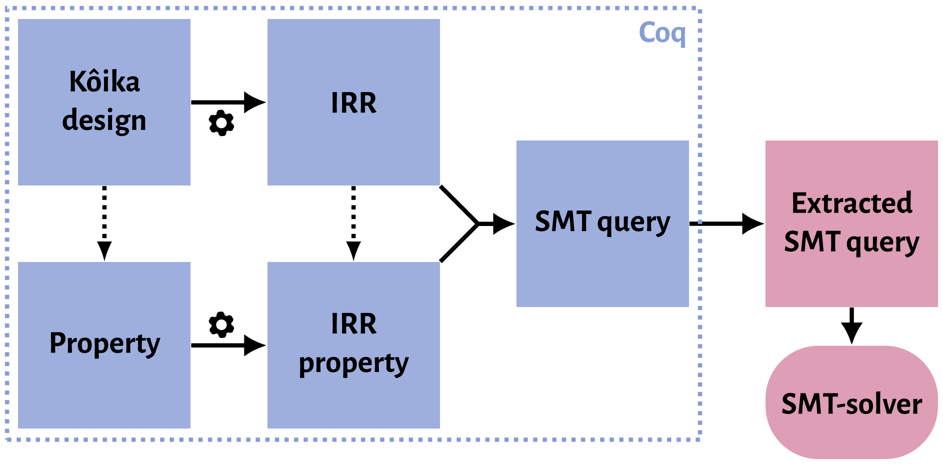


Figure 4.9: SMT-based reasoning in Kôika

The gear icon (⚙) indicates transitions that occur automatically

4.3.3 Experimental evaluation and comparison

4.3.3.a Replaying the same proofs

The most direct way of evaluating the efficiency of the hybrid Coq-SMT workflow is to verify the correctness of the shadow stack a second time. We start by proving within Coq that the queries we extract entail the statements we verify manually. We then pass these queries to the solver.

The SMT-aided verification of the shadow stack took about 4.3 seconds on the same processor as previously (for a speedup of about 150). Gains went beyond a reduction of the time required by Coq to validate the proof. The process of writing the proof in the first place was also made much swifter and smoother, with RAM usage never being an issue.

The sources for this work can be found online [3]. Note that this version of the proof relies on a primitive version of the process we describe in this section. In particular, there is no user-friendly Makefile for directly verifying the proofs — instead, we relied on a manual and risky copy-and-paste process. The modern version of the process is used in the example presented in the following subsection.

[3]https://gitlab.inria.fr/SUSHI-public/FMH/koika/-/tags/og_smt_proofs

4.3.3.b Proofs on an extended processor

Throughout his two-month internship in the SUSHI team in the summer of 2024, Gabriel Desfrenes successfully extended the processor with interruptions and exception support. He also modified the shadow stack to make it emit an exception instead of simply turning the whole processor off whenever an anomaly is encountered. The solver handles this more complex example in around 40 seconds.

The sources for this work can also be found online [4].

Section recap

In this section, we discussed how the processor we described, extended, and verified in Section 4.2 can be used in the real world.
In the next section, we turn to the limitations of the system and perspectives for its further development.

4.4 Limitations and perspectives

4.4.1 Limitations

4.4.1.a Sensitivity of the IRR

Some transformation passes such as `ReplaceVar` (see 4.5a) take identifiers of IRR variable as arguments. Proofs using these passes are very sensitive to changes to the design and tend to break even when changes occur in unrelated parts of the design. A way around this problem would be to allow referring to variables through a more persistent naming scheme. The workflow described in Section 4.3 is beneficial in this regard, as it involves automatic tools that are not sensitive to names, reducing the need for manual intervention.

4.4.1.b Kôika and memory

Kôika does not offer a way of using the block RAM of FPGAs. All the registers of a Kôika design are stored in the usually much smaller LUTs. Currently, the only way of storing registers in BRAM would be to build an interface relying on external calls, as

[4] https://gitlab.inria.fr/SUSHI-public/FMH/herve

was done for the data and instruction memory in the RISC-V example. However, this makes reasoning about such registers harder.

4.4.1.c Configuration of the security mechanism

Our implementation of the shadow stack mechanism is hardcoded. Such an approach means that parameters, such as the size of the shadows stack, cannot be adjusted at runtime. Moreover, there is only one shadow stack. These choices ease the verification of the mechanism and are consistent with the type of processor we use, which corresponds to a microcontroller without privileged mode. Such a CPU often executes a single application in an embedded device. An interesting extension would be to provide some configuration mechanisms to adapt the mechanism at runtime. This configuration mechanism must not be accessible to untrusted application code. Thus, this requires considering a more complex CPU with privileged execution mode and OS support. This evolution also poses the challenge of formal reasoning on the interaction between hardware and some trusted code, i.e., OS kernel code.

4.4.1.d Handling of security property violations

Violations of the shadow stack result in the processor halting. This behavior is rather abrupt, but it helped us validate our approach. A more adequate response would be to emit a hardware exception that an operating system could handle at its discretion.

4.4.1.e Industrial applicability

The Kôika language is an academic language that has not yet seen industrial use. It currently misses some essential features to be a viable option for this purpose, the most critical among them being the support of reusable modules. For complex circuits, code duplication leads to larger memory footprints and degraded performance. A module system would also help with the proving process: the ability to define lemmas about modules could lead to a natural way of compartmentalizing designs by decoupling an implementation from its interface. Indeed, a lemma proven once about a module would hold for any of its instances. With the right lemmas, looking at a module's implementation may not be necessary to reason about its behavior.

4.4.2 Perspectives

4.4.2.a Functional verification

There is an official formal version of the RISC-V specification based on the Sail language [149], which includes facilities to export definitions to Coq. Proving that our processor design conforms to this specification would be a logical next step.

4.4.2.b Generalizing the processor design

Our target processor is relatively simple (unprivileged ISA, 32 bits, minimal extensions). We could generalize our results by working with a family of processors instead of a single concrete instance. Our proof should work the same way for any legal combination of RISC-V extensions. We have progressed in generalizing the processor design by generating a Kôika processor design from a list of RISC-V extensions. However, the semantics of many new instructions are yet to be defined. Moreover, we were limited in our implementation because only the unprivileged part of the specification had been implemented. Adding support for the privileged part of the specification would open possibilities for interacting with the operating system.

4.4.2.c Other security mechanisms

Another research direction would be considering more ambitious security mechanisms, such as a more complex version of shadow stacks or capabilities. We would also like to tackle security mechanisms requiring proper software configuration, e.g., the isolation mechanism provided by OS kernels or hypervisors leveraging hardware features. Such approaches require formalizing the interactions between software and hardware components [92].

We managed to verify the properties we defined about the shadow stack entirely within the SMT-solver. This is unfortunate, as it means that this example does not illustrate the benefits of working within a proof assistant. Indeed, many classical verification tools include no less powerful SMT-solvers and should therefore be able to handle this example just as well. An ideal example would leave room for a proof assistant to shine. The mechanisms mentioned in the previous paragraph may be suitable for this purpose on account of their complexity.

4.4.2.d Timing side-channels

Kôika's cycle-accurate semantics makes it possible to reason about some timing side-channels. Targeting mechanisms that enforce more complex policies, such as Information Flow Tracking mechanisms, is more challenging. Indeed, such mechanisms are supposed to guarantee some forms of non-interference, which correspond to predicates on sets of traces, i.e., hyperproperties [150].

4.4.2.e Automatic procedures

The hybrid Coq-SMT workflow described in Section 4.3 has much better ergonomics than the manual one. Nevertheless, it could still be improved.

The main weakness of this process is its TCB imprint: we blindly trust an unverified tool. Pragmatically speaking, this problem is minor — this process is already as secure as formal tools in use within the industry. Nonetheless, we could reduce the TCB imprint by automatically generating proof terms and returning them to Coq. The whole process would be up to Coq's security standards.

Integrating the SMT-based procedure leaves room for improvement. We have to extract code and run commands manually outside of Coq. Ideally, we should never have to step out of the Coq environment. A tactic providing access to SMT-solvers and handling proof certificates would solve our TCB concerns and the cumbersomeness of the process.

Additionally, it should be possible to automatically handle the conversion from a subset of Coq propositions to SMT queries (this would have to happen outside of Coq, although this may be made transparent to the user). With our current approach, the user manually does most of the conversion work in Coq. Queries are defined using some syntactical constructs defined in the assistant. In addition to being unwieldy, this conveys some risks: this manual conversion process may go wrong. This risk can be mitigated by proving that the query implies the original proposition, as described in Subsection 4.3.3.a, but this comes at the cost of even more manual labor.

SMTCoq [151] is a third-party project whose aim is to provide a clean interface between Coq and SMT-solvers. Alas, Kôika manipulates bitvectors, and SMTCoq's support for bitvectors is insufficient: only some very trivial queries are handled successfully.

4.5 Conclusion

This chapter proposes a methodology for building synthesizable hardware with formally verified security mechanisms. We base our work on the Kôika formal HDL, which we modify in depth to make it practical for reasoning on hardware designs. We implement a verified compiler from Kôika designs to a lower-level, more explicit representation, which is more amenable to proving. In addition, we define a set of verified transformation passes on these low-level representations that can be applied to simplify objects in this representation, as needed for each proof.

We then apply our methodology to implement a verified shadow stack for a simple pipelined RISC-V processor. Notably, we show that detecting the overwrite of a return address results in the halting of the processor. This result is further confirmed by simulating the processor and running a simple example code performing a buffer overflow, which is indeed detected by our shadow stack.

We also present an extension of this methodology relying on an automatic, SMT-based procedure. This extension significantly lowers the human effort required for formal verification. On the downside, the procedure relies on an unverified external solver.

While the security mechanism verified here is relatively simple, it forms the foundation for possible future work and exemplifies how more complex mechanisms could be tackled.

Chapter 5
COQQTL

In Chapter 4, we described how we adapted the Kôika language to make it more amenable to proofs. However, this language was an academic one and had yet to be used in the industry; in fact, it lacked essential features to make it a realistic choice for this purpose (such as proper support of modules and instances, making the language inherently non-modular).

In this chapter, we describe COQQTL, our Coq implementation of FIRRTL. This latter language is an intermediate representation used by Chisel, a modern HDL that is used in academia [152] as well as in the industry [153]. Our intention is to enable the verification of Chisel designs (after their compilation to FIRRTL) without disturbing the classical Chisel workflow, as shown in Figure 5.1. Alternatively, COQQTL designs can be defined within Coq and only later exported to FIRRTL.

We describe how we generate an IRR from COQQTL designs as we did for Kôika ones. This is a first step towards building a framework for the formal verification of COQQTL designs in the line of our previous work.

Chapter outline

This chapter is structured as follows. Section 5.1 introduces COQQTL, our Coq implementation of FIRRTL, alongside its semantics. In Section 5.2, we describe how we compile COQQTL code to a form more amenable to formal reasoning — almost the same as the IRR we implemented in Kôika. We discuss limitations and future work in Section 5.3. Section 5.4 closes this chapter.

5.1 The COQQTL language

COQQTL (pronounced [kɔktəl]) is an implementation of the FIRRTL language embedded within the Coq proof assistant. It is based on version 3.2.0 of the language [154] In this section, we describe the syntax and semantics of this language.

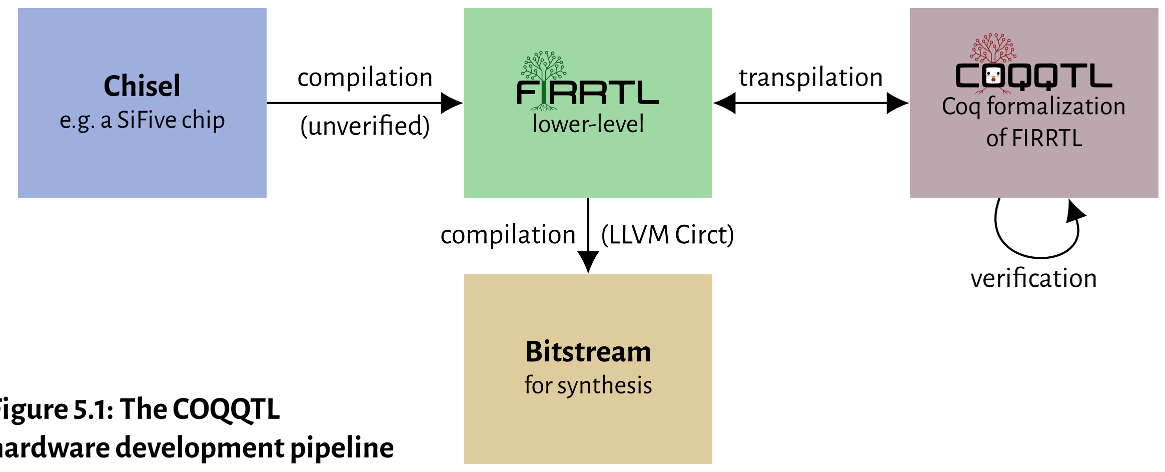


Figure 5.1: The COQQTL hardware development pipeline

Section outline

We begin with an overview of the original language and our port in Subsection 5.1.1. COQQTL's syntax and semantics are introduced in Subsection 5.1.2. Subsection 5.1.3 discusses how COQQTL designs are interpreted directly within Coq.

5.1.1 Overview

FIRRTL (Flexible Intermediate Representation for Register-Transfer Level) originated as an Intermediate Representation (IR) for Chisel, an open-source HDL embedded within Scala (which "adds hardware construction primitives to the Scala programming language, providing designers with the power of a modern programming language to write complex, parameterizable circuit generators that produce synthesizable Verilog"[1]). Nowadays, it is less tied to Chisel. Indeed, it is used as an IR by the LLVM CIRCT (Circuit IR Compilers and Tools) compiler, an (unverified) hardware compiler built following the LLVM methodology [18].

COQQTL can be used by importing FIRRTL code obtained by compiling designs written in languages supported by CIRCT (such as Chisel) or even directly as a low-level HDL. Indeed, although FIRRTL originates as an IR, it is also a full, human-readable HDL in its own right. Furthermore, as COQQTL is embedded within Coq, we can define higher-level constructs using the facilities provided by this system. Going back to the abstraction level of Chisel would mostly be a matter of designing a hardware

[1] https://github.com/chipsalliance/chisel/blob/v6.0.0/README.md

$$\begin{array}{rrll} \text{Bitvector} & bv. & & \\ \text{Types} & t & ::= & \texttt{UInt}\ n \mid \texttt{SInt}\ n \mid \texttt{Clock} \mid \texttt{AsyncReset} \\ & & \mid & \texttt{Vector}\ sz\ t \mid \texttt{Bundle}\ \{(name, t, \mathit{flipped})_{\mathit{field}}*\} \\ & & \mid & \texttt{Enum}\ \{(name, t)_{\mathit{variant}}*\} \\ & & \mid & \texttt{Const}\ t. \\ \text{Values} & val & ::= & (bv, t). \end{array}$$

Figure 5.2: Syntax of COQQTL values

design library on top of COQQTL (just like FIRRTL code roughly corresponds to what is left after the Scala portion of a Chisel design has been fully evaluated).

We implement the core features of the FIRRTL language, including its rich type system and its intricate semantics. They are presented in the rest of this chapter. It should be noted that our support for modules is only partial as of this writing[2].

We declare features that are not critical for our purposes to be out of scope, although there is no major obstacle to their implementation. Some of these features provide ways of doing a limited form of verification — our approach mostly supersedes them.

5.1.2 Syntax and semantics

This section focuses on the syntax and semantics of COQQTL, which aligns closely with the FIRRTL specification, version 3.2.0 [154]. We outline COQQTL's type system, the components it introduces, and its definition of circuits and expressions, providing a comprehensive overview of the language's structure and meaning.

5.1.2.a Types and values

COQQTL is a typed language. Its type system, described in Figure 5.2, is rather expansive, including both **simple** and **aggregate** types. It defines signed and unsigned integers (`SInt` and `UInt`), clocks (`Clock`), reset signals (`AsyncReset`), as well as fixed size arrays (`Vector`, parameterized by a type and a size), structures (`Bundle`, parameterized

[2]In effect, we support single modules only, although everything that we implemented up to this point is required for and compatible with modules. What is missing for going from single modules to multiple modules is more implementation time. In fact, we already implemented several modules-related mechanisms, e.g., for detecting that instances are not used in a way that leads to dependency loops.

Expressions e :=
| $\texttt{uconst}\ \overrightarrow{bv}$ | $\texttt{sconst}\ \overrightarrow{bv}$
| $\texttt{enum}\ t\ name\_tag\ value$
| $\texttt{ref}\ r$ | $\texttt{sfield}\ value\ subfield$
| $\texttt{sindex}\ value\ n$ | $\texttt{saccess}\ value\ e$
| $\texttt{as\_uint}\ e$ | $\texttt{as\_sint}\ e$
| $\texttt{as\_clock}\ e$ | $\texttt{as\_async\_reset}\ e$
| $\texttt{shl}\ n\ e$ | $\texttt{shr}\ n\ e$
| $\texttt{andr}\ e$ | $\texttt{orr}\ e$ | $\texttt{xorr}\ e$
| $\texttt{bits}\ hi\ lo$ | $\texttt{head}\ n\ e$ | $\texttt{tail}\ n\ e$
| $\texttt{pad}\ n\ e$ | $\texttt{cvt}\ e$ | $\texttt{neg}\ e$ | $\texttt{not}\ e$
| $\texttt{add}\ e_1\ e_2$ | $\texttt{sub}\ e_1\ e_2$ | $\texttt{mul}\ e_1\ e_2$
| $\texttt{div}\ e_1\ e_2$ | $\texttt{rem}\ e_1\ e_2$
| $\texttt{eq}\ e_1\ e_2$ | $\texttt{neq}\ e_1\ e_2$ | $\texttt{lt}\ e_1\ e_2$
| $\texttt{leq}\ e_1\ e_2$ | $\texttt{gt}\ e_1\ e_2$ | $\texttt{geq}\ e_1\ e_2$
| $\texttt{and}\ e_1\ e_2$ | $\texttt{or}\ e_1\ e_2$ | $\texttt{xor}\ e_1\ e_2$
| $\texttt{dshl}\ e_1\ e_2$ | $\texttt{dshr}\ e_1\ e_2$ | $\texttt{cat}\ e_1\ e_2$
| $\texttt{mux}\ e_{sel}\ e_1\ e_2$.

Statements s ::=
| $\texttt{wire}\ name\ t$ | $\texttt{node}\ name\ e$
| $\texttt{register}\ name\ t\ e_{clk}$
| $\texttt{register\_reset}\ name\ t\ e_{reset}\ e_{v\_init}\ e_{clk}$
| $\texttt{memory}\ name\ mem\_info$
| $\texttt{instance}\ name\ module\_name$
| $\texttt{when}\ e_{cond}\ s_{then}\ s_{else}$
| $\texttt{match}\ e\ \{(name_{variant}, ?name_{binder}, s)_{branch}*\}$
| $\texttt{seq}\ s_1\ s_2$ | $\texttt{skip}$
| $\texttt{stop}\ e_{clock}\ e_{cond}\ n\ ?name$
| $\texttt{connect}\ e_{to}\ e_{from}$ | $\texttt{invalidate}\ e$.

$$\begin{array}{rcl l}
\text{Port\_direction} & d & ::= & input \mid output. \\
\text{Ports} & p & ::= & \{(name, d, t)_{port}*\}. \\
\text{Module} & m & ::= & name \times ports \times statement. \\
\text{Circuit} & c & ::= & \{m*\} \times toplevel\_module\_name.
\end{array}$$

Figure 5.3: Syntax of COQQTL circuits

by a list of fields with a certain name, type and flippedness — we will get back to the role of flippedness when discussing modules and their interfaces) and unions (`Enum`, parameterized by a list of variants with a certain name and type). This rich type system makes it possible to write expressive COQQTL code despite the language's lack of high-level constructs. Furthermore, types (of components or subcomponents, such as that of a specific field in a bundle) may be declared `constant` (`Const`). A COQQTL value is nothing more than a bitvector tagged with a type.

5.1.2.b Syntax of circuits and modules

The syntax of the COQQTL language is described in Figure 5.3. A COQQTL circuit contains a set of modules. Modules are design bricks that are described only once but can be instantiated multiple times in a design — this is an essential part of what makes the language modular. One of the modules is marked as the toplevel: simulating or emitting a bitstream for the circuit is the same as simulating or emitting a bitstream for this module. The other ones are included implicitly as direct or indirect dependencies of this main module. Indeed, modules may contain instances of other modules. Three elements characterize a module:

- A name;
- A list of input and output ports (particular components that represent the interface of the module);
- A statement, characterizing its behavior.

The name and the input/output ports can be seen as the module's public interface. Everything else is contained in the module's statement, whose role is chiefly confined to describing two elements:

- The components that make up the module in addition to its ports;
- The connections between components/ports or constants: where they exist, under which conditions they are used, etc. These connections provide a way of computing the effects of input changes on both the outputs and the internal state of the module.

5.1.2.c Components, flow and connections

A COQQTL module contains various types of components. We already mentioned its ports while talking about its interface. These are components that represent communi-

Flow f ::=
| `source`
| `sink`
| `duplex`.

$$flip\_flow(f): \begin{cases} \texttt{sink} & \text{if } f = \texttt{source} \\ \texttt{source} & \text{if } f = \texttt{sink} \\ \texttt{duplex} & \text{if } f = \texttt{duplex} \end{cases}$$

Components *comp* ::=
| `Input` *name t* | `Node` *name e*
| `Memory` *name mem_info*
| `Instance` *name module_name*
| `Output` *name t* | `Wire` *name t*
| `Register` *name t* e_{clk}
| `Register_reset` *name t* e_{reset} e_{v_init} e_{clk}.

$$get\_flow\_comp(comp): \begin{cases} \texttt{source} & \begin{array}{l}\text{if } comp = \texttt{Input}\ \_\ \_ \lor comp = \texttt{Node}\ \_\ \_ \\ \lor\ comp = \texttt{Memory}\ \_\ \_ \lor comp = \texttt{Instance}\ \_\ \_\end{array} \\ \texttt{duplex} & \begin{array}{l}\text{if } comp = \texttt{Output}\ \_\ \_ \lor comp = \texttt{Wire}\ \_\ \_ \\ \lor\ comp = \texttt{Register}\ \_\ \_\ \_ \\ \lor\ comp = \texttt{Register\_reset}\ \_\ \_\ \_\ \_\ \_\end{array} \end{cases}$$

$$get\_flow(e): \begin{cases} get\_flow\_comp(get\_comp(r)) & \text{if } e = \texttt{ref}\ r \\ get\_flow(value) & \begin{array}{l}\text{if } e = \texttt{sindex}\ value\ \_ \\ \text{or } e = \texttt{saccess}\ value\ \_\end{array} \\ \begin{array}{l} flow_v := get\_flow(value) \\ t_v := get\_type(value) \\ \texttt{if}\ is\_field\_flipped(t_v, field) \\ \texttt{then}\ flip\_flow(f) \\ \texttt{else}\ f \end{array} & \text{if } e = \texttt{sfield}\ value\ field \\ \texttt{source} & \text{otherwise} \end{cases}$$

Figure 5.4: Syntax of COQQTL components

cation channels between the module and its environment. Ports come in two variants depending on their flow, the flow being the notion that characterizes whether a component can be read from (`source`), written to (`sink`), or both (`duplex`). `Input` ports are `source`, whereas `Output` ports are `duplex` (`sink`s are more complex, as we will see).

Components other than ports are introduced in a module's statement. For instance, a `Register` (`duplex`) is a stateful component that can store data between the ticks of the clock signal which drives it (the e_{clk} expression is expected to have type `Clock`). A `Register_reset` is a variant of this component, which is reset to some default value whenever some signal is received — this reset signal must either have type `UInt<1>` or `AsyncReset`, depending on whether resets should run in sync with the `Register`'s clock signal or not. On the other hand, a `Wire` (`duplex`) is a combinatorial component, which merely binds a value to a name and is unable to store state. A `Node` (`source`) is a combinatorial component that represents a named expression. As such, writing to it after it has been initialized is impossible. Finally, an `Instance` (`source`) represents a concrete instantiation of a submodule, and a `Memory` (`source`) is used for storing large amounts of data (the data is not directly accessible — instead, this component gives us an interface for storing data on a nondescript memory).

Components are named, and distinct components may share the same name. In scopes where several components sharing the same name are accessible, the one defined most recently shadows all the older ones.

The `ref` expression is used to access components by name. There are also expressions (`sfield`, `sindex`, `saccess`) for accessing subcomponents. For instance, `sfield (ref "c") "f"` means that we want to access field `f` of the component corresponding to `c` in the current scope. For this expression to be valid, a component `c` must be in-scope, and we need to check its type. Indeed, all components are associated with a type. For most components, this type is given explicitly. For instance, the value associated with a `Wire` defined as `wire w: UInt<8>` will be of type `UInt<8>`.

Going back to our example where we wanted to access field `f` of component `c`, we need to ensure that the type of the value associated with the `c` component is a `Bundle` with a field called `f`. The expression then returns the value of this field. If the type of component `c` is something other than a `Bundle` with a `c` field, then this expression is invalid.

Sometimes, the type of the value associated with a component does not appear in its declaration. For instance, COQQTL synthesizes a value of type `Bundle` for com-

```
circuit c:
  module m:
    input  in : UInt<8>
    output out: UInt<8>

    out <= add(in, UInt<8>(1))
```

Figure 5.5: Listing — increment input

```
circuit c:
  module m:
    input  in : UInt<8>
    output out: UInt<8>

    in <= add(out, UInt<8>(1))
```

Figure 5.6: Listing — flow error

ponents of class `Instance` and `Memory`. This `Bundle` represents their interface, as the FIRRTL specification describes. A `Bundle` associated with an `Instance` contains all of the `Instance`'s ports. Of course, input and output ports cannot be treated uniformly. Critically, input ports can only be read from — this is where flippedness comes into play. A flipped field has a flow opposite to that of the `Bundle` that contains it. Since input ports of `Instance`s should have `sink` flow, the corresponding fields in the `Bundle` are flipped. Similarly, the type of the value obtained when accessing a `Memory` is a `Bundle` representing an interface to the actual component (for instance, it contains a field for each read port that the memory offers).

The notion of flow is used only by the `connect` statement. This statement connects e_{to} to e_{from}. There are constraints on the flow of these two expressions. e_{to} should be writable (`sink` or `duplex`) and e_{from} should be readable (`source` or `duplex`). The `get_flow` function provides a way of associating a flow to expressions. Most expressions are assigned the `source` flow. Only expressions accessing components or subparts of objects of complex types may have a different flow. In particular, a `connect` cannot take a `mux` as its left-hand-side expression.

Figure 5.5 is our first concrete example of a COQQTL circuit. The `<=` symbol is syntactic sugar for the `connect` statement. In this example, we introduce a circuit `c` composed of a single module named `m`. This module has two ports: an input port called `in`, and an output port called `out`. Both ports share the same type, namely `UInt<8>`. This module takes the value it received in `in`, increments it, and outputs the result to `out`.

Figure 5.6 is almost the same, except that the occurrences of `in` and `out` are swapped in the statement. This trivial change is enough to make the circuit invalid. Indeed, the flow of `in` is `source`, and that of `out` is `duplex`. Although building the value `add(out, UInt<64>(1))` is valid, resulting in a value of flow `source` (like most expressions), connecting this value to `in` is disallowed — `in` has flow `source`, and `source`s are invalid target for writes.

For the sake of readability, later examples only show the definition of the ports and statement when a single module is used, skipping the `circuit` and `module` declaration.

The language supports both combinatorial and sequential assignments. Unlike other languages, this is done through a single construct: the `connect` statement. Whether an assignment is blocking or nonblocking depends solely on the component it targets: for instance, assignments to a `Register` are nonblocking, whereas those to a combinatorial component such as a `Wire` are blocking. Therefore, FIRRTL only requires one form of assignment.

5.1.2.d Expressions

Not all syntactically valid definitions are semantically valid. For instance, we already mentioned how `Register`s expect `Clock` typed arguments. Similarly, not all expressions support all types of arguments. Most expressions only handle integer arguments (e.g., most numerical operations such as `add` accept either two `UInt`s or two `SInt`s). Furthermore, the type of expressions is unambiguous (assuming two inputs of type `UInt`/`SInt` w_1 and `UInt`/`SInt` w_2, `add` always returns an `UInt`/`SInt` $\max(w_1, w_2) + 1$).

The syntax of expressions is given in Figure 5.3. They can be classified into five broad categories:

- **Constant definitions**, such as `uconst` or `enum`. These expressions make it possible to introduce a new value out of nowhere. Note that there is no way of generating a `Bundle` or a `Vector` this way — the only way of getting an element of these types is by accessing appropriately typed components.
- **References and subaccesses**, such as `ref` or `sindex`. These expressions allow us to access components in scope (with `ref`) and to focus on some precise part of their value when their types allow it (with `sindex` and `saccess` for accessing `Vector` items, and `sfield` for accessing `Bundle`s).
- **Conversion operations**, such as `as_uint` or `as_clock`, to obtain a value of some from a value of a different type. These work only with `UInt`s/`SInt`s.
- **Numeric operations**, such as `add` or `and`, to generate values by combining other ones.
- The `mux` statement, an outlier. Its first argument, e_{sel}, must be a `UInt`<1>, and its other two arguments, e_1 and e_2 should share the same type. The value of the first expression determines to which of these two values the expression evaluates.

5.1.2.e Components initialization

There are other conditions that a COQQTL circuit has to respect. For instance, the value associated with each component has to be clearly defined. Four different mechanisms influence this definition:

- explicit `connect`;
- explicit `invalidate`;
- implicit conservation of the previous value (stateful components only);
- implicit resetting (`Register_reset` only).

The first occurs whenever a `connect` statement is used. In this situation, the (sub)component that is the target of a `connect` statement gets its value set to the value of the expression specified in the statement.

Alternatively, a (sub)component may be the target of an `invalidate` statement. This statement can be viewed as a `connect` to an unspecified value of the appropriate type. The compiler can pick any concrete value for this as long as it is used coherently.

Finally, by default, the value of stateful components (such as `Register`s) is preserved from one cycle to the next. Running a `connect` statement connected to a `Register` is not bound to affect the `Register`. Indeed, `Register`s and `Register_reset`s take an e_{clk} argument. The value of the `Register` can be updated through a `connect` only on a rising edge of the clock.

Depending on how the `Register_reset` is defined (and in particular on the type of its e_{reset} argument, which is either `UInt`<1> or `AsyncReset`), the reset may or may not need to wait for a rising edge of the register's clock to take effect. In any case, the reset takes precedence over the conservation of the `Register_reset`'s default value.

5.1.2.f Absence of loops

Valid COQQTL designs are exempt from loops. For instance, it is forbidden to bind a wire to itself (although it is possible to bind a register to itself, due to the non-blocking nature of assignments to registers: the new value of the register from the next clock cycle on would then be defined to the one it already held). This is a problem that can be found in other languages as well. For instance, the synchronous language Esterel forbids programs that require an unbounded amount of computation in a tick, as described by Potop-Butucaru et al. [155].

The checks are fine-grained, making it, for instance, possible to bind the second

```
input in: UInt<8>[3]
wire w: UInt<8>[3]
w     <= in
w[2]  <= w[1]
```

Figure 5.7: Listing — cycle-free module

```
input in: UInt<8>[3]
wire w: UInt<8>[3]
w     <= in
w[2]  <= w[1]
w[1]  <= w[2]
```

Figure 5.8: Listing — module with cycle

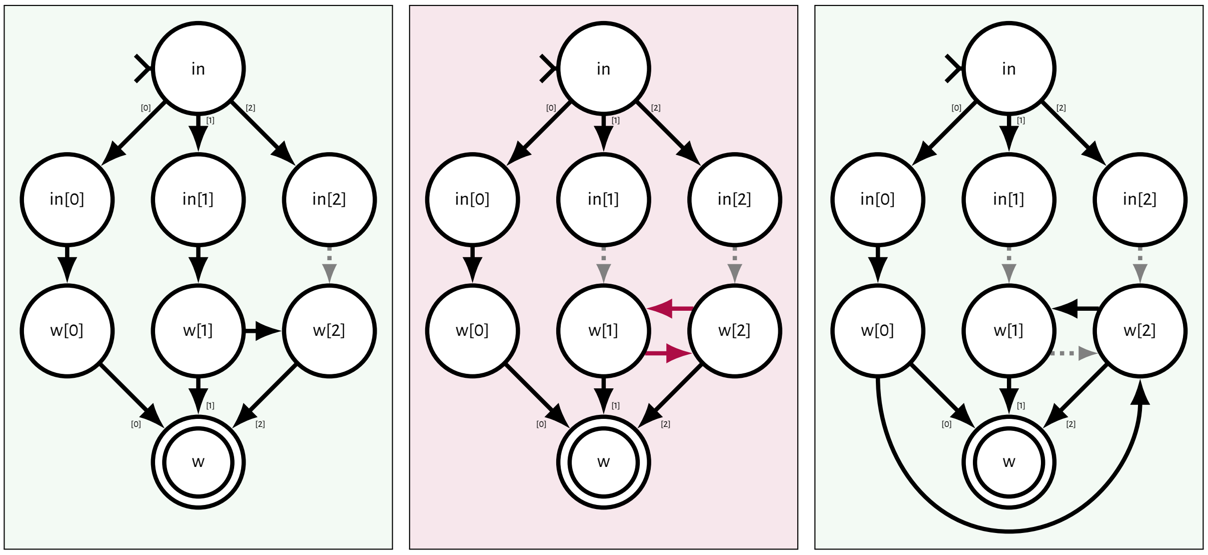
```
input in: UInt<8>[3]
wire w: UInt<8>[3]
w     <= in
w[2]  <= w[1]
w[1]  <= w[2]
w[2]  <= w[0]
```

Figure 5.9: Listing — cycle-free module

element of a vector-typed wire to the third element of the same vector (Figure 5.7; note that the `v[x]` notation is syntactic sugar for the vector access expression). However, adding a further connection from the third element to the second one leads to a cycle, and the compiler thus rejects the design (Figure 5.8).

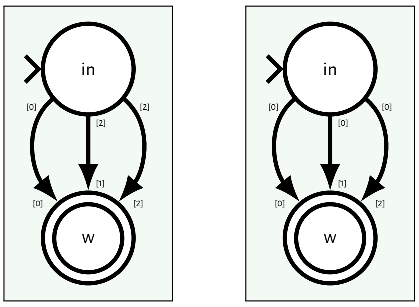
Figure 5.9 shows that, perhaps surprisingly, adding a specific connection at the end of this incorrect design can make it correct again. This behavior is because the third connection overrides the first one, removing the cycle. In COQQTL, later connections take precedence over earlier ones. This behavior makes specializing connections easy. For instance, we may want to connect two registers `v1` and `v2` corresponding to matching vectors of size 10 through `v1 <= v2`, before overwriting the connection of the 5^{th} item only in a subsequent connection `v1[4] <= x`. The entirety of the mechanisms involved in the handling of this case are introduced throughout this chapter.

Connections between (sub)components may be depicted graphically. In Figure 5.10, we show what this looks like for Figure 5.7, Figure 5.8 and Figure 5.9. The `in` component is an input port, so its value is always known at the start. We mark such given values with a ❯ symbol. Then, we split `in` into its subcomponents (`in[0]`, `in[1]` and `in[2]`). These elements only depend on the value of `in`. We represent such dependencies using an arrow (⟶). We also split the `w` component into its constituents (`w[0]`, `w[1]` and `w[2]`). Of course, the final value of component `w` depends on all 3 of its subcomponents. We also represent the dependencies introduced by connections. For instance, the first connection in all three statements is `w <= in`: `w[0]` takes the value of `in[0]`, `w[1]` takes the value of `in[1]` and `w[2]` takes the value of `in[2]`. In all three examples, the next connection is then `w[2] <= w[1]`. This connection overrides the first one as far as `w[2]` is concerned. Overridden dependencies are represented with a dotted, gray arrow (⇢). Seeing whether there is a loop in the statement is merely a matter of checking whether there are cycles in the graph (not taking dotted arrows into account).



(a) for Figure 5.7 (b) for Figure 5.8 (c) for Figure 5.9

Figure 5.10: Connections graphs



(a) for Figure 5.7 (b) for Figure 5.9

Figure 5.11: Compact connection graphs

Dependency arrows that are parts of cycles are represented in a different color (→). Nodes representing components whose value needs to be determined are represented with a double circle (◎).

The systematic splitting of all components is not strictly necessary. More compact representations of the acyclic connection graphs presented in Figure 5.10c are given in Figure 5.11. Although both graphs presented are equivalent to those from the previous figure, the second representation is much more compact: all dependencies of the component `w` are there. Ultimately, all the information required to compute the values that need to be determined comes from the values given at the start (this should not be surprising).

```
input   which: UInt<1>
input   v0: UInt<8>
input   v1: UInt<8>
output  out: UInt<8>

when which:
  out <= v0
else:
  out <= v1
```

Figure 5.12: Listing — conditional connections

```
input   in : UInt<8>
output  out: UInt<8>



when UInt<1>(1):
  out <= in
else:
  skip
```

Figure 5.13: Listing — initialization issue

Note that this kind of graph can show the presence or the absence of loops, but it is inadequate for ensuring that a component's value is explicitly given. Indeed, as we shall now see, connections may be conditional. A (non-overwritten) conditional connection adds dependencies to the (sub)component on its left-hand side, yet it does not by itself guarantee that the (sub)component is indeed always initialized: the condition guarding the `connect` may be false.

5.1.2.g Conditions, aggregate components and component initialization

The examples presented thus far were simplistic in that they always involved direct connections between components. However, any (correctly typed) expression may appear on the right-hand side of a `connect` statement. In particular, a connection may introduce dependencies to several components.

The whole story of component initialization is more complex than we initially let on. Two important notions need to be discussed in more detail:

- Conditions may guard some connections;
- Components of aggregate types may be initialized one piece at a time.

Figure 5.12 and Figure 5.13 illustrate how conditional connections impact components initialization and loop detection.

Figure 5.12 shows two conditional assignments (we introduce some syntactic sugar for the `when` statement). The first assignment appears in the first branch of the `when`, and the other appears in the alternative branch. This example is correct: the value of `out` is defined no matter which branch is taken (and `out` is the only value we need to define).

To a first approximation, the criterion for rejecting a circuit is the existence of at least

```
input   in : UInt<32>[3]
input   x  : UInt<32>
output  out: UInt<32>[3]
out     <= in
out[2]  <= x
```

Figure 5.14: Listing — aggregates connection

```
input   in1: UInt<32>
input   in2: UInt<32>
output  out: UInt<32>[3]
out[0]  <= in1
out[1]  <= in2
```

Figure 5.15: Listing — piecewise connections

one state of the circuit and its inputs such that the value of at least one subcomponent is undetermined.

However, Figure 5.13 is rejected by the COQQTL compiler. Indeed, the second branch of the `when` statement does not explicitly define a value for `out`. The second branch should never run in practice: the e_{cond} argument of the `when` is literally `UInt<1>(1)`. However, the compiler does not analyze condition expressions, viewing these parts of condition statements as opaque. Therefore, the system cannot guarantee that a value is assigned to `out` in all situations, and the module must be rejected.

The actual criterion is that the value of all components should be defined no matter which branches of conditionals are taken. Therefore, the only components that conditional statements initialize are those in the intersection of the set of components initialized by each branch.

The second notion that needs more explanation is the initialization of components of aggregate type. Indeed, these components can be initialized in multiple, possibly overlapping steps.

Figure 5.14 and Figure 5.15 illustrate how aggregate components are handled from the standpoint of component initialization validation and loop detection.

Figure 5.14 shows how an aggregate type can be defined from another. This example is correct: all parts of the component are initialized. Note that `out[2]` is connected twice: first through the `out <= in` statement, and then through the `out[2] <= x` one. As we already discussed, the latter initialization takes precedence over the former.

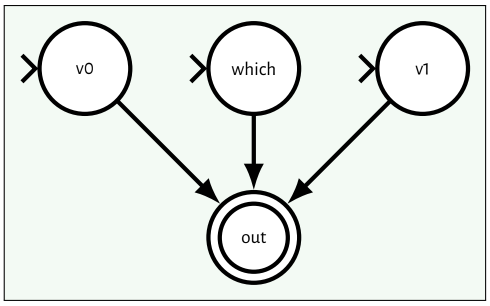
On the other hand, the example presented in Figure 5.15 has to be rejected. There, the `out` component is initialized piecewise, with the values of its first and second elements explicitly defined. However, the value of its third and last element is left undefined. Adding an (unguarded) connection to this third item would make this definition legal.

Note that the components appearing in the condition expression must be added as dependencies for all the connections under it. Figure 5.16 shows an example of a module rejected due to a condition-induced loop.

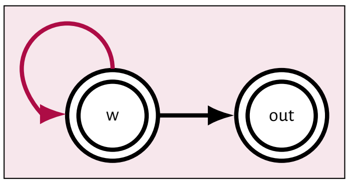
```
output out: UInt<1>

wire w: UInt<1>
when w:
  w <= UInt<1>(0)
else:
  w <= UInt<1>(1)
out <= w
```

Figure 5.16: Listing — condition induced loop



(a) for Figure 5.12



(b) for Figure 5.16

Figure 5.17: Dependencies graphs for examples involving conditions

In Figure 5.17, we give the dependencies graphs of two recent examples, which included conditions. Of particular interest is Subfigure 5.17b, which makes the issue Figure 5.16 immediately visible.

All in all, there are only a few expressions for accessing subcomponents, namely:

- the `sfield` expression, used for accessing fields of `Bundle` typed elements;
- the `sindex` expression, used for accessing the nth item of a `Vector` typed element, where n is an *integer*;
- the `saccess` expression, used for accessing the nth item of a `Vector` typed element, where n is an *expression*.

As to conditional constructs, there are:

- the `when` statement, as illustrated in Figures 5.12 and 5.13;
- the `match` statement, used for pattern matching on values of `Enum` type;
- perhaps more surprisingly, the `saccess` expression.

The `saccess` expressions are in both categories, as they act as a subcomponent accessor and a conditional. Indeed, this is the subcomponent accessor that returns the e^{th} subcomponent of a `Vector` typed element (such an element must necessarily be a component — there is no way of defining a literal `Vector`, for instance). Resolving the expression *e* is not straightforward. For the same reasons why we do not analyze the expression of `when`s, we do not analyze the expressions of `saccess`es (the only thing we know about it is that it is well-typed). We cannot even guarantee that the index it resolves to is within bounds. When the value of `saccess` is read, it returns the value of the designated element when it is within bounds, and it may return any well-typed value when it is out of bounds.

On the other hand, when it sits on the left-hand side of a `connect` statement, it turns into a conditional assignment: if the expression resolves to 0, then the assignment concerns the first item of the `Vector`, if it is 2, then it concerns the second item, and so on for all in-bounds indices. If the expression resolves to an out-of-bounds index, no connection occurs. Therefore, connections to `saccess`es always add dependencies to all the elements of the `Vector`, yet they can never be guaranteed to define the value of any concrete subelement (they may always be out-of-bounds; remember that we do not analyze the expression).

Additionally, `saccess`es may appear outside of the final position in statements. Consider for instance `c[e].b` and `d[e][e'][2]`. In these examples, `[e]` and `[e']` represent `saccess`es. All of these are followed by additional elements. Assuming that the type of `c` is:

```
Vector(Bundle([("a", UInt<8>); ("b", UInt<5>)]), 3)
```

And that of `d` is:

```
Vector(Vector(Vector(UInt<8>, 8), 3), 2)
```

The ids matched by the first expression are `c[0].b`, `c[1].b` and `c[2].b`. Those matched by the second expression are more numerous:

```
d[0][0][2]  d[0][1][2]  d[0][2][2]
d[1][0][2]  d[1][1][2]  d[1][2][2]
```

The interplay of these factors makes checking that there are no components with undefined values and no dependencies loops rather complex.

5.1.2.h Multiple clocks

An essential difference between Kôika and COQQTL is that the latter language does not have a clear notion of ticks. In Kôika, the semantics used a notion of cycles based

```
circuit c:
  module aux:
    input  in : UInt<8>
    output out: UInt<8>
    out <= in

  module main:
    instance m of aux
    wire w: UInt<8>
    m.in <= w
    w <= m.out
```

Figure 5.18: Listing — loop with instance

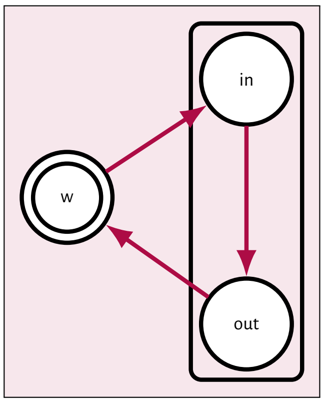


Figure 5.19: Connections in Figure 5.18

on an implicit clock that drove the entire circuit. In contrast, in COQQTL, the state is updated based on input variations. COQQTL, therefore, supports multiple clocks, and it is up to the user to ensure that clock domains are crossed safely.

In COQQTL, the `Clock` type simply represents one bit of data. The clock itself is not modelled directly in COQQTL. Instead, `Clock` typed ports of the toplevel module may be driven from the environment. It is possible to build derived clocks: the language provides an `as_clock` function, which casts any object of a simple type into a clock.

5.1.2.i Modules

Functionally speaking, `instance`s of modules are inlined in the modules that include them. They are viewed as bundles, with a flipped field for each output and a non-flipped field for each input. This abstracted view ensures that interface and implementation are kept distinct. In particular, internal components cannot be accessed directly.

Module `instance`s may be part of connection loops. Consider, for instance, Figure 5.18. There, module `aux` has two ports: an input port `in` and an output port `out`. Furthermore, the output port value is continuously set to that of the input port. Then comes the `main` module, in which the `aux` module is instantiated as `m`. This module contains a loop: `wire` `w` is driven by `m.out` while also driving `m.in`. Figure 5.19 illustrates this cycle more clearly.

The public interface of a module consists of a name and a list of inputs and outputs. We expand this interface with dependencies relating the module's inputs to its outputs to enable tracking connection loops in modules without breaking encapsulation. This dependency information is automatically derived from a module's description and then used whenever the module gets instantiated. Of course, the module itself is checked for internal loops at the point where it is defined.

5.1.3 Interpreting circuits

Having the current value of inputs is not enough to evaluate a module's new state. Indeed, registers are updated on the rising edge of their associated clock — i.e., when their previous value was low and is now high. The previous state needs to be remembered for edge detection.

COQQTL supports derived clocks — i.e., clocks can be built dynamically from arbitrary data through the `as_clock` function, which can convert data of any simple type to `Clock` type. This feature makes it easy to define clock dividers and clock signals with, e.g., half the ticking rate of input. As derived clocks may be based on any input, remembering the previous state of the `Clock`-typed inputs is insufficient.

More generally, having information regarding the previous state of the inputs is helpful for performance: not all inputs impact all outputs. Knowing which inputs changed helps limit computations for obtaining the new state.

Note that COQQTL's semantics is not deterministic. In situations involving the `invalidate` statement, the interpreter is given free reins in picking a value. In particular, it may choose to always set invalidated components to zero. Compilers may instead choose values in such a way as to simplify the implementation, for instance by replacing an `invalidate` guarded by a `when` with an assignment corresponding to the one in the other branch.

Section recap

This section presented COQTTL, our Coq port of the FIRRTL HDL. We took the time to introduce its syntax and semantics and how it can be interpreted directly within Coq. In the next section, we turn to reasoning on COQQTL circuits. We build an IRR that follows the same principles we used when dealing with Kôika designs.

5.2 The COQQTL IRR

In this section, we introduce our Intermediate Representation for Reasoning for COQQTL. As indicated by its name, this representation is meant to prove things about COQQTL designs. It is similar to the IRR we introduced for reasoning about Kôika designs in Section 4.1.2.

Section outline

We start by justifying our use of IRRs in this different context in Subsection 5.2.1: we want to shift our focus further in the direction of automation (and SMT-based proving in particular) and the relevance of IRR in this setting is not immediately apparent. We then talk about the structure of the IRR in Subsection 5.2.2. Finally, we describe how the conversion to IRR is handled in practice in Subsection 5.2.3.

5.2.1 Why an IRR for COQQTL?

We mentioned how we intend to build an interface for automated solvers to make verification more accessible and efficient. In particular, we mentioned SMT-solvers.

Most properties are expressed in terms of components and their values. For instance, we may want to show that the value of register r never drops below 2, assuming the input i was always positive. Although converting expressions to an SMT-compatible form is easy, generating a formula for the state of entire COQQTL components is more complex. This state not only depends on expressions on the right-hand side of connection statements but also on the conditions that guard these statements and the precise subcomponents that they target, the fact that later connections can override earlier ones, etc. In addition, some high-level features need to be simplified. For example, COQQTL components are referred to by name. We must check which in-scope component was last defined to solve the reference. These concerns are too high-level for SMT-solvers and must be handled upstream.

All in all, we want to turn a high-level representation into a low-level, explicit one. This situation is reminiscent of the one described in Chapter 4: we had a high-level language with a complex semantics, leading to performance issues when doing proofs manually in Coq. To make reasoning on Kôika circuits tractable, we converted them to a low-level explicit form. A representation like the one we introduced for reasoning in Kôika fits the bill.

```
circuit c:
  module m:
    input   in1: UInt<8>
    input   in2: UInt<8>
    output out: UInt<8>

    wire w1: UInt<8>
    wire w2: UInt<8>
    w1  <= add(in1, in2)
    when lt(in1, in2):
      w2  <= sub(in2, in1)
    else:
      w2  <= sub(in1, in2)
    out  <= mod(w1, w2)
```

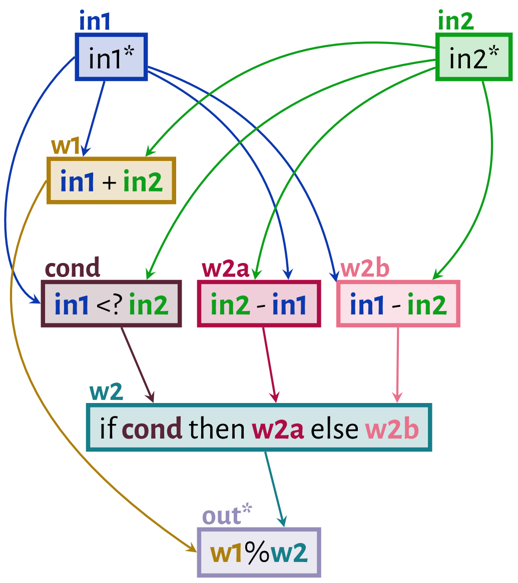


Figure 5.20: Example module and its associated IRR

5.2.2 The IRR format

Like our Kôika IRR, our COQQTL IRR is based on an ordered set of nodes. The nodes in the IRR are built out of three ingredients:

- expressions: these mostly correspond to COQQTL expressions;
- conditionals: some (sub)components may be conditionals;
- references: nodes can refer to each other, with the proviso that a node may only refer to nodes smaller than it — this guarantees that there are no loops in the structure and makes sharing possible (which helps with keeping memory usage in checks).

Despite the many similarities, two crucial differences exist between the Kôika IRR and its COQQTL counterpart.

First, there are differences between Kôika and COQQTL related to the way in which they are updated. These differences are reflected in the IRR. Kôika assumes that there is a single clock controling the entire system. All registers are updated unconditionally on every cycle. Furthermore, it includes notions such as that of ports: we introduce nodes in the IRR that correspond to values that registers transiently hold during a cycle. In contrast, COQQTL does not have a notion of cycles, as designs are evaluated in an event-driven way: we update their state based on changes in the toplevel inputs. Clocks are not modeled inside of COQQTL. Instead, the toplevel module may take clock-typed signals as input[3]. Registers updates are tied to specific clock-typed signals, which we need to consider when building the corresponding IRR nodes.

The second difference between the two IRRs is related to the type systems of both languages and, more specifically, the handling of aggregate types in COQQTL. Indeed, when considering such a COQQTL object, we need a high level of granularity: this is because connections may be of the form `v[5] <= in`, only updating part of the fuller component. We can refer to whole components as well as to subsets of them. When referring to, e.g., `v`, we want to see the value of all of its subcomponents, including `v[5]`. We, therefore, need to add constructs for building or updating aggregate values based on simpler values. For instance, we have an expression that takes a list of n values of type `t` and returns a vector composed of those values. We also have a construct for updating single values of **`Vector`**s and single fields of **`Bundle`**s: that way, we could describe `v` as `update_index(v', 5, in)`.

Outside of these changes, the IRR we end up with is essentially the same as the one we described in the previous chapter.

Figure 5.20 showcases an example of a simple COQQTL circuit and its associated IRR. This circuit contains a single module with two inputs (`in1` and `in2`) and one output (`out`), which is set to $(\texttt{in1} + \texttt{in2})\%|\texttt{in1} - \texttt{in2}|$. To obtain the absolute value of $\texttt{in1} - \texttt{in2}$, we use a conditional to detect which of the two inputs is the smallest. It is then subtracted from the other, larger input.

For each expression that appears in the code, a node is generated in the IRR. Nodes are also created for every right-hand side expression of a `<=` statement and condition expression. Furthermore, the circuit has a node for each component and port.

Nodes corresponding to input ports of the main module are particular: their value is always supposed to be defined in the environment. For practical purposes, they are defined as constants. In practice, these constants may not have a concrete value but may be Coq variables.

5.2.3 Generating IRRs

In this section, we describe turning a COQQTL circuit into an IRR. Figure 5.21 gives a high-level overview of the process for converting a circuit from COQQTL to IRR: starting from a single COQQTL module, we first build a log, tracking its components

[3]Note that enough delay should be left between input updates for the circuit to remain stable.

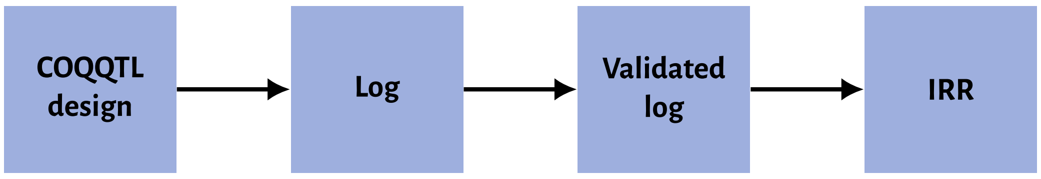


Figure 5.21: Generation of an IRR from a module

and the connections between them, enforcing some properties such as well-typedness are along the way. Using this log, we check that the values of all components are indeed defined and that the module does not contain dependency loops. When the log successfully passes these checks, we consider it validated. It is only then that we generate the actual IRR.

To generate an IRR starting from a (validated) log, we begin by defining a function for defining an IRR node for a single COQQTL expression, which is relatively easy (all COQQTL-expressions can be expressed using a few SMT-friendly primitive bitvector operations). This function needs to be called for each node in the IRR. Perhaps surprisingly, this process is not entirely trivial: we generate nodes in a specific order, ensuring that all dependencies have been handled before creating it. In contrast to that process, evaluation is straightforward and terminates trivially: evaluating all nodes in order is sufficient.

Note that we only described how single modules are handled. In general, however, modules can include other modules. It turns out that just like module instances are inlined in their parents, their IRR can be included in that of their parent (for some definition of inclusion). Modularity extends to reasoning about modules: a result proved once on a module's IRR holds for all instances.

In the rest of this section, we cover the transformation from circuits to IRR in more detail. We first consider simple modules that do not include instances before turning to the general case.

5.2.3.a Simple module to log

The first step of taking a syntactically valid simple COQQTL module to an IRR is to convert its statement into a log, a structure in which component declarations and connections are kept distinct. During its construction, all name-based references are turned into id-based references, simplifying further analysis. A log contains two elements:

- A map of ids to components (we introduce unique ids to avoid having to deal

```
module m:
  input  ina: UInt<1>
  input  inb: UInt<1>[2]
  output out: UInt<1>

  wire w: UInt<1>
  when ina:
    w <= inb[1]
  else:
    wire z: UInt<1>
    z <= add(inb[0], inb[1])
    w <= z
  out <= w
```

(a) Original module

Components:

```
① Input  UInt<1>
② Input  UInt<1>[2]
③ Output UInt<1>
④ Wire   UInt<1>
⑤ Wire   UInt<1>
```

Connections:

```
when ①:
  ④ <= ②[1]
else:
⑤ <= add(②[0], ②[1])
  ④ <= ⑤
③ <= ④
```

(b) Resulting log

Figure 5.22: Module to log

with scope resolution later on — remember that names can be shadowed)

- A "connection tree", a data structure storing the connections and the conditions under which they occur; it is a syntax tree with only the `connect` and `invalidate` statements remaining (as well as the conditions guarding these statements)

We also check for some semantic properties along the way. Non-exhaustively, we guarantee that:

- There are no typing errors
- There are no flow errors
- The names are valid and unique
- Patterns cover all variants of an enum

In Figure 5.22, we give an example of a (somewhat contrived) statement next to its corresponding log. This illustration highlights that the connections section of the log is simply a subset of the original statement. The key differences are that name-based references have been replaced with index-based ones, and component declarations have been filtered out.

5.2.3.b Cleanup and validation

At this point, we have started digesting the statements. We have logs where all the information is easy to access, but we still have to check that the following properties

```
a <= b
when UInt<1>(0):
  skip
else:
  a <= c
```

Figure 5.23: Listing — conditions and overriding
Conditions are not analyzed, so the first connection is not detected as necessarily overridden

are respected:

- All components are initialized (or invalidated)
- There are no cycles

We already described most of what there is to know about these checks back in Subsection 5.1.2.e (for components initialization) and Subsection 5.1.2.f (for loops detection).

In addition to performing these checks, necessarily overridden statements are also removed during this step. We already discussed conditions for removal when we first presented loop detection. In particular, we mentioned how overridden connections are not considered when looking for cycles. A statement is necessarily overridden if and only if there is no possible valuation of the context such that no later connection overrides it.

The usual caveat about conditions not being analyzed applies here. The detection process of necessarily overridden connections is an underapproximation: there may be false negatives. An example is shown in Figure 5.23. Just like Figure 5.13, this is a situation where COQQTL misses something that a human reader could trivially see: the same branch will always end up being taken, as the condition is the "false" constant. The first write on a is necessarily overridden in practice, but COQQTL does not remove it.

Note that this behavior is not explicitly described in FIRRTL's specification [154]. Here, we mostly align with the `firtool` FIRRTL compiler, with one key distinction: unlike COQQTL, `firtool` performs some basic condition analysis. For example, in the scenario mentioned above, `firtool` would recognize the first connection as necessarily overridden because the expression consists of a single constant. However, its condition analysis does not extend beyond this level of complexity.

At the end of this stage, we have a verified log (without cycles and with all components guaranteed to be initialized in all circumstances) and normalized (with all conditions that match COQQTL's underapproximate criterion for being necessarily overridden removed).

5.2.3.c Log to IRR

We want to generate an actual IRR expression starting from a verified and normalized log. We proceed in two steps.

First, we define a function for resolving a single id, i.e., generating the corresponding IRR node. This node should represent the id's value at the end of the cycle, which depends on many elements (the id may be the target of many connections that require the final values of other ids to be known).

The function we define assumes that all the ids required to resolve the current id have already been resolved. We expect to find the nodes associated with these ids in a map dedicated to the storage of resolved ids. Note that these dependencies correspond to the arrows seen in Figure 5.20 and that there exists a way of picking ids that does not lead to a cycle: the logs we are handling have been verified to be loop-free earlier.

We build another function from isolated IRR nodes to a full-blown IRR for a given log. Its role is to pick an appropriate order for sending the ids to the resolution function; in other words, it does topological sorting on our dependency graph. Generating the nodes in order is not strictly required: inserting a reference can be done before the related id is resolved. Our implementation stores the IRR graph as a hashmap from integers to IRR nodes, with the integers acting as identifiers and sorted by topological order. Computing this order makes later proofs easier.

Some expressions that appear in the initial design impact multiple ids. Repeating the translation of the same expressions for all the ids they impact would be wasteful. Of course, we already manage repetitions related to ids, as discussed in the previous paragraph — we do not inline the definition of referenced ids; instead, we transform references to ones to the associated IRR nodes. The conditions introduced in COQQTL's branching expressions (e.g., `when`) are another source of redundancy. Just like in Kôika, we introduce nodes for storing the IRR form of these expressions. These nodes are generated as a side effect of the function for handling ids. Inserting such a node in the IRR shifts the identifiers of all the following nodes, but the overall topological order between component ids is maintained.

Resolving a single id Generating the node corresponding to a single id is done in a series of steps. We start with a target id and the entire log obtained for the circuit. In particular, we can access the whole tree of connections.

Our first step is to collect the relevant connections in the connections tree. We do this by filtering the tree and removing all connections that do not impact the current id. For instance, we would remove a connection of the form `b <= x` when handling the a id. Perhaps more surprisingly, we also remove the connections that target subcomponents of our id. For instance, we would remove a connection of the form `a[2] <= x` when handling the a id. However, we do not remove connections to prefixes of the id. For instance, we would not remove a connection of the form `a <= x` when handling the `a[2]` id. Abstract connections that may be prefixes of the id are also relevant. For instance, `a[m + n] <= x` is kept when handling id `a[2]`.

The next step is similar to the removal of necessarily overridden connections mentioned in the previous subsection. Although we removed such connections before, we did not do it from the standpoint of the precise id under focus. For instance, assuming we are focusing on id `a[2]`, a sequence of connections `a <= x`, `a[2] <= y` would survive the initial filter (assuming it is not overridden as far as, e.g., `a[0]` is concerned) as well the removal of connections irrelevant to `a[2]`. Nevertheless, the second one would necessarily override the first connection in this context. The overall filtering process is identical to the one described previously, except that we act as if all connections targeted precisely the same id.

Afterward, we analyze all the conditions that appear in the tree. For each of them, we check whether we handled them before, i.e., if an IRR node representing their value already exist. We handle each condition that needs handling, inserting them in the IRR as appropriate and memorizing the binding.

We have two kinds of connections left in the tree: some target the id directly, whereas others target one of its prefixes. As we care about the effect of connections on the id itself, we make the latter connections more precise. For instance, assuming we care about id `a[2]`, we turn connection `a <= x` into `a[2] <= x[2]`.

Similarly, some of the connections left in the tree are abstract. `a[m + n] <= x` only impacts the value of `a[2]` when `m + n` equals `2`. To fix this, we add conditions in the tree for guarding such connections. Unlike what we did for the conditions that were present in the base tree, we do not generate nodes in the IRR for the new ones (although some more sharing here may technically be applied).

IRR expression e :=
| `irr_uconst` $\overrightarrow{bv}$ | `irr_sconst` $\overrightarrow{bv}$ | `irr_vconst` $type\ n\ \overrightarrow{bv}$ | `irr_bconst` $fields\ \overrightarrow{bv}$
| `irr_econst` $variants\ \overrightarrow{bv}$
| `irr_ref` n
| `irr_sfield` $e\ subfield$ | `irr_sindex` $e\ n$ | `irr_saccess` $e\ es$
| `irr_as_enum_tag` $variants\ tag\ e$ | `irr_as_uint` e | `irr_as_sint` e
| `irr_as_clock` e | `irr_as_async_reset` e
| `irr_shl` $n\ e$ | `irr_shr` $n\ e$
| `irr_andr` e | `irr_orr` e | `irr_xorr` e
| `irr_bits` $hi\ lo\ e$ | `irr_head` $n\ e$ | `irr_tail` $n\ e$
| `irr_pad` $n\ e$ | `irr_cvt` e
| `irr_neg` e | `irr_not` e
| `irr_add` $e_1\ e_2$ | `irr_sub` $e_1\ e_2$ | `irr_mul` $e_1\ e_2$ | `irr_div` $e_1\ e_2$ | `irr_rem` $e_1\ e_2$
| `irr_eq` $e_1\ e_2$ | `irr_neq` $e_1\ e_2$ | `irr_lt` $e_1\ e_2$ | `irr_leq` $e_1\ e_2$ | `irr_gt` $e_1\ e_2$
| `irr_geq` $e_1\ e_2$
| `irr_and` $e_1\ e_2$ | `irr_or` $e_1\ e_2$ | `irr_xor` $e_1\ e_2$
| `irr_dshl` $e_1\ e_2$ | `irr_dshr` $e_1\ e_2$
| `irr_cat` $e_1\ e_2$
| `irr_mux` $selection\ port_a\ port_b$ | `irr_when` $cond\ t\_branch\ f\_branch$
| `irr_pattern` $on\ branches$
| `irr_overwrite_index` $e\ n\ v$ | `irr_overwrite_field` $e\ field\ v$
| `irr_invalidated`
| `irr_module_input` $instance\ port$ | `irr_module_output` $instance\ port\ deps$.

Figure 5.24: Syntax of IRR expressions

After this tedious pretreatment, we are free to focus on generating the IRR expression. We will not describe this process in detail: remember that there is almost a one-to-one correspondence between COQQTL and IRR expressions. At this point, only conditions are left to handle through the conditional constructor of IRRs — the syntax of IRR expressions is given in Figure 5.24.

Finally, we extend it with information about the subcomponents of the current id. Indeed, the real value of `a[2]` depends on that of `a[2].f`. However, we removed all such connections back at the filtering stage. We expect that all of the subcomponents of the id have been resolved before the id itself is resolved. Resolving every single id would be wasteful: for instance, if there is a single connection targeting a vector of size 1024, e.g., `v <= w`, then it would be wasteful to introduce 1024 ids which would all restate a specialized version of this connection. Here, resolving only id `v` would suffice. In general, we resolve the ids we strictly have to.

Generating a topological order Generating a topological order is relatively straightforward. Starting from a dependency graph (which we already generated when checking for loops during the log validation), we apply a classical depth-first search algorithm. We call the resolution function directly from the one that does the sorting and tracks the identifier to be used for the next IRR node — we are careful to consider the fact that a call to the resolution function may result in the insertion of several nodes in the IRR (that function returns a tally of the inserted nodes).

Section recap

In this section, we presented our COQQTL IRR. In particular, we explained why we turn to this approach, how the IRR is structured, and how the conversion is handled in practice.
Building this IRR is a first step towards the construction of a full-fledged verification framework in the line of our work with Kôika. Indeed, an IRR is an explicit form that contains all the information necessary to the generation of, e.g., SMT queries. Our earlier work about generating such queries from the IRR defined for the Kôika language can be largely reused in the context of COQQTL. More generally, the work that we present in this section is an important stepping stone towards efficient and modular verification.
In the next section, we consider both short-term and long-term perspectives.

5.3 Perspectives

Section outline

In this section, we describe perspectives for future work. We start by mentioning two short-term prospects, with Subsection 5.3.1 presenting our plans for handling modules and Subsection 5.3.2. The later sections describe long-term plans. We discuss the development of higher-level languages built on top of COQQTL in Subsection 5.3.3. In Subsection 5.3.4, we outline the benefits that a Coq implementation of a generic IRR form could have. We then turn to the missing features of COQQTL in Subsection 5.3.5. With these features implemented, we could import arbitrary designs expressed in FIRRTL into COQQTL. Finally, in Subsection 5.3.6, we mention potential targets for verification.

5.3.1 Managing modules

Until now, we have been considering a simplified version of COQQTL that does not include modules. This subsection discusses the changes required to handle them.

Although module instances are semantically inlined in their parent, actually doing the inlining systematically in practice would be wasteful: it would make exploiting the fact that different instances of a module share properties harder to exploit. The interface of an instance is a bundle where only ports appear. We also know which input ports impact the value of which output ports. Input and output ports appear in the IRR through special constructors (`irr_module_input` and `irr_module_output`). Unlike classical dependencies, those between inputs and outputs appear explicitly in the IRR (this is the *deps* argument of the `irr_module_output` expression). Both expressions contain a link to their module of provenance.

Even at this level of representation, modules remain independent. In particular, each module has its own IRR. Modules in which another module is instantiated can access the third-party IRR to evaluate its output ports.

When reasoning manually, high-level results about modules can be used to limit the evaluation cost. For instance, if a particular module is known to implement multiplication in one cycle, then a read to its output can be replaced with an expression multiplying its inputs directly — the actual definition of the module does not even have to be read.

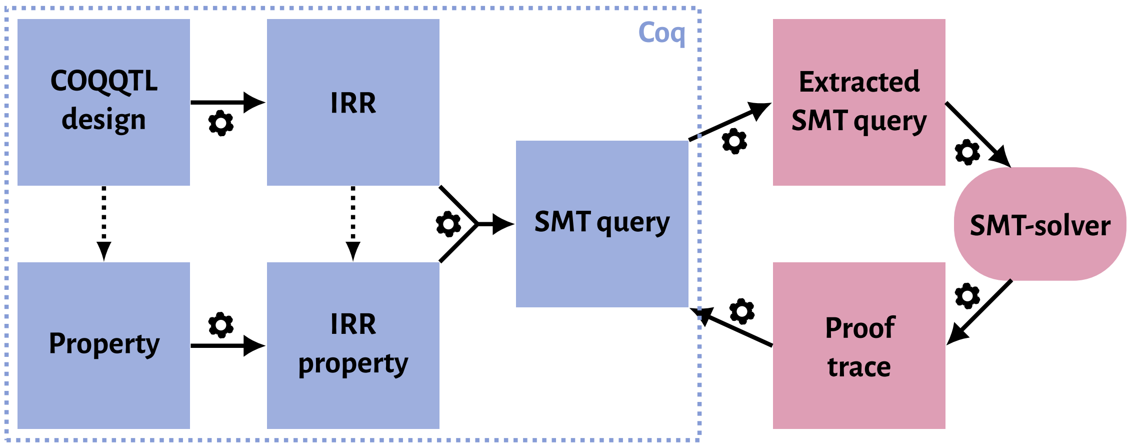


Figure 5.25: SMT-based reasoning in COQQTL
The gear icon (⚙) indicates transitions that occur automatically

5.3.2 Integrating SMT-solvers

This section describes how IRRs can generate SMT obligations. We propose a pipeline for directly interacting with solvers within the Coq proof assistant. At the time of this writing, work on the concrete implementation of a link between an external SMT-solver and our implementation of COQQTL has yet to start. However, the more complex problem of converting a COQQTL design to an explicit form has been handled for a large subset of the language (i.e., support for modules is still a work in progress). In other words, we already have the basic constructs in place to build SMT queries similar to those we built in Kôika (as described in Section 4.3) for this subset of the language.

What is left to do, then, is to implement a pipeline for discharging Coq goals involving COQQTL constructs to external solvers. We introduce the pipeline we intend to implement in Figure 5.25. This pipeline differs from the one we implemented in Kôika (described in Figure 4.9) in several regards. Indeed, although this first attempt at integrating SMT-solvers yielded satisfactory results, the process through which these results were obtained was cumbersome. Users were required to step out of the Coq environment and launch scripts to pass extracted data to external solvers. The Coq kernel did not validate any proof. In effect, the external tools were trusted.

The process we envision would run as follows. The user would start by defining a property about a COQQTL design. As represented by the dotted arrow, this property could reference concrete elements of the design directly. As described in Section 5.2, IRRs can be derived automatically for COQQTL designs. As the semantics of COQQTL designs is defined in terms of IRRs, all properties of COQQTL designs admit IRR-based formulations. For instance, references to components of the designs become references to IRR nodes. This process corresponds to the one we described for Kôika. It is only for the actual proving process that the methods begin to diverge. We plan to include a tactic for turning Coq goals about COQQTL designs (in IRR form) into SMT queries. This tactic would come as part of a plugin capable of calling an external, proof-producing SMT-solver. The proof trail generated by the solver should then be converted to a term compatible with Coq's calculus and incorporated into the proof (a problem already solved by existing plugins [151]).

Note that the SMT-solver would not be a substitute for classical Coq tactics. In situations where the solver fails, all the facilities of the proof assistant remain available. Goals simple enough to be solved by the solver should be deferred to it, while the high-level structure of the proof should be built manually in Coq.

5.3.3 Higher level of abstraction

Although the FIRRTL language is human-readable, it remains low-level. After all, it started as an intermediate representation of the Chisel language. Implementing this language on top of COQQTL is a natural next step. However, although Chisel would probably be the easiest one to port to Coq, there are other languages we may want to consider on account of their traction within the industry, such as SystemVerilog or VHDL. In order to connect COQQTL to these languages, a verified compiler is required (which in turn requires a formal semantics of the language in question).

Since we already have a formal semantics and much infrastructure around Kôika, we may also consider this language a viable target. We could also reuse existing implementations of subsets of Verilog, such as the one used in Kôika.

5.3.4 Generalizing IRRs

During this thesis, we implemented IRRs twice: once for Kôika and again for COQQTL. We did not use a unified representation for both applications since we elected to tailor the IRRs to our precise workloads. Switching to a more generic representation would make the conversion slightly more complex, but it would make the conversion to SMT form and interaction with SMT-solvers free.

SMT formulas already have the property of being generic. The reason why we do not content ourselves with them is that they are unwieldy. What the IRR brings over such formulas is a representation that can be handled by a proof assistant, as shown by the manual verification that we did back in Chapter 4.

5.3.5 Supporting the whole specification

Back in the introduction of this chapter, we mentioned how we wanted to open the possibility of verifying imported Chisel designs directly (after an unverified compilation down to FIRRTL and an unverified but trivial transpilation from FIRRTL to COQQTL, that is). For this to work in all generality, implementing the entirety of the specification is required. The features our implementation is missing are:

- Verification statements used for testing designs, such as assertion statements checked during simulation;
- Formatted prints for logging information during simulation;
- Probes, a feature used for accessing the values of internal components of modules, used for verification statements based verification;
- Layers, a feature used for keeping secondary parts of modules (e.g., those related to testing and debugging) distinct from the base definition;
- A limited form of type inference — our types are always given explicitly;
- Interplay with externally defined modules;
- Annotations (for storing arbitrary metadata about portions of a design).

Some parts of the specification may be ignored. For instance, we could skip assertions — after all, we provide a system that supersedes this mechanism. However, even these parts would be valuable additions. Assertions constitute an accessible form of formal methods. For instance, such assertions could be automatically verified when detected in circuits.

5.3.6 Validating the methodology on non-trivial examples

There are many targets that we could pick to illustrate our methodology. Properties that fall outside of the scope of traditional verification tools are especially interesting.

Several high-profile examples of open-source projecs related to RISC-V CPU design were produced in Chisel, including the Rocket Chip Generator [152] and the BOOM [156] processor. These designs could be compiled to FIRRTL and import. The verification of security properties on these designs would make for a great illustration of our methodology.

5.4 Conclusion

This chapter discusses our work on COQQTL, our Coq implementation of the FIRRTL IR. The primary objective of this project is to establish a basis for a framework for general-purpose hardware verification based on a real HDL with some industrial presence. We implement most of the language's specification, barring some non-critical features such as assertions, which we plan to address in future work.

We develop a pair of transpilers between FIRRTL and COQQTL. This makes it possible to import real Chisel designs for verification within Coq and to export designs built within Coq for real-world use.

Additionally, we construct an IRR for COQQTL in the spirit of the one we developed for Kôika. This IRR gives a way of reasoning about FIRRTL designs efficiently from within the Coq proof assistant.

Compared with Kôika, we plan to introduce a more comprehensive integration of automatic solvers. We aim to streamline the verification process, making it less manual and reducing its TCB footprint. By enhancing automation and minimizing reliance on manual labor, we hope to improve the overall efficiency of hardware verification within the Coq environment.

Future work will focus on refining this integration, ensuring that the entire FIRRTL standard is covered, and applying our methodology to the verification of large-scale designs.

Chapter 6

Conclusion

In this thesis, we set out to answer a specific research question.

Research question

How to formally verify implementations of security mechanisms for processors at the register transfer level of description within a proof assistant?

In this final section, we summarize the work presented in this thesis and introduce perspectives for future work.

Section Outline

Section 6.1 summarizes the main ideas introduced in this thesis. Concrete avenues for short-term improvements of our work are outlined in Section 6.2, whereas Section 6.3 describes broader perspectives for further research work in the area.

6.1 Summary

The growing dependence of our societies on computers makes them susceptible to cyberattacks. Incorrect software can introduce vulnerabilities that can have serious real-world consequences. By using formal verification, we can precisely specify the behavior of software and verify that they are free of vulnerabilities. Software relies on hardware, and software verification typically assumes the underlying hardware correct. If this hypothesis is incorrect, the proofs built on top of it are essentially worthless. The need for software verification, in turn, induces a need for practical methods to verify hardware.

Given the importance of hardware security, manufacturers verify their designs using testing-based and formal processes. However, these formal processes rely on limited tools that are not fit for handling the problem in its full generality.

In recent years, fundamental assumptions about processor behavior have been shown to be incorrect for most modern hardware. The most striking examples were not related to functional correctness but to temporal side channels [33], [34]. Checking that a processor is free of such defects requires reasoning based on low-level definitions. Proof assistants are well-suited for this task. However, their complexity and the various performance pitfalls they present can be significant drawbacks.

We propose a methodology based on an intermediate representation for hardware where all the information is encoded explicitly. We call this representation the Intermediate Representation for Reasoning, or IRR for short. We define transformation passes for simplifying hardware in IRR form according to the available hypotheses. We prove each such pass semantics preserving. Users can define additional passes according to their needs. The controllability of this process is a way of sidestepping the performance pitfalls of the proof assistant. Furthermore, we include a connection to a fully automatic resolution procedure, which can solve many goals by passing them to an external SMT-solver. To add support for an HDL, defining a verified compiler from it to the IRR is all that is required.

6.2 Short-term improvements

In the following, we present various short-term improvements we could apply to our work. These improvements are about adding everything required for what we feel would be a production-ready verification framework.

6.2.1 Extending COQQTL

The COQQTL implementation presented in Chapter 5 did not cover the entire specification of the language. Having a reliable port of FIRRTL would be an excellent starting place for hardware verification. It could provide a solid shared foundation for different hardware verification projects, enabling researchers to focus on higher-level verification concerns.

Supporting assertions would bring about a natural low-complexity way of incorporating hardware verification in hardware design. Proof obligations could be automatically generated from such assertions — and perhaps even automatically verified.

6.2.2 Improving the integration of automatic procedures

Our current bindings to SMT-solvers leave room for improvement. They require users to step out of the Coq environment and execute unverified scripts that pass extracted data to unverified tools. While suitable for a proof of concept, this approach makes the TCB needlessly bigger, and the user experience could be enhanced.

TCB concerns Many automatic procedures are proof-producing, meaning that they generate proof trails that can be used to derive Coq proof terms. Proof-producing procedures are not added to the TCB, as their results are not taken at face value but checked by the Coq kernel, as any user-written Coq proof would be. There exist tools for generating Coq proofs from external solvers [151], [157]. We could adapt their results to our needs.

User experience concerns Coq supports user-defined plugins that can interact with the environment and introduce custom tactics. This feature enables the approach described in Section 5.3.2, allowing users to conduct their developments entirely within Coq.

Calling external solvers asynchronously could be beneficial: such calls may last a few minutes, disrupting the workflow. Unfortunately, while Coq supports the asynchronous processing of proofs, this capability does not extend to individual tactics.

6.2.3 Illustrating our methodology

We currently lack examples of non-trivial verification work performed in COQQTL. Numerous targets could prove interesting for this purpose. Verifying either functional or security properties of third-party processors such as those of the Rocket Chip family [152] the BOOM core [156] would be a good test of the capabilities of our framework and methodology in practical applications. These processors are more complex than those we considered in our earlier work, allowing us to incorporate more sophisticated security mechanisms. For instance, we could turn to verifying an extended shadow stack mechanism with tighter integration into the system.

6.3 Perspectives

In the following, we discuss ways to expand our work. In contrast to what we presented previously, this section is about long-term goals.

6.3.1 Full verification of processors

Despite numerous projects in this field, we are unaware of any fully verified realistic processor. By “realistic”, we mean a processor that goes beyond a toy model; e.g., it should be capable of running a barebones Linux-based operating system. By “fully verified”, we mean that adherence to a specification must be formally enforced. The Sail project [149] provides formal specifications for many standard ISAs. Straightforward reference implementations can be derived from these specifications. The execution traces of valid processors must align with those of these implementations, according to some notion of compatibility. Furthermore, this project covers the generation of Coq definitions from such descriptions.

In the late 1990s and early 2000s, there was an active research community focused on validating microarchitectural optimizations, such as pipelining or Tomasulo’s algorithm [158], [159], [160], [161], [162]. However, their results were obtained on models, not RTL descriptions. Furthermore, they used simplistic custom specifications rather than standard ISAs. Nonetheless, these works provide insight into how we may approach our proofs. We could follow similar arguments to demonstrate that concrete RTL implementations are valid refinements of the Sail specification.

More recently, Reid et al. [163] described a verification workflow used at Arm, introducing techniques for working with complex microarchitectures. While they verify RTL descriptions, their process relies on bounded model checking. Therefore, their results come with the limitations inherent to this form of verification.

Although our workflow is general enough to accommodate the verification of functional properties, we did not consider such properties in this thesis. The Kôika processor we verified (heRVé [1]) is also limited: it implements the base version of RISC-V and lacks many privileged features. Considering the target we outlined, we would need to implement a particular set of extensions (adding native support for multiplications and divisions, single- and double-precision floating point numbers, atomic instructions,

[1]https://gitlab.inria.fr/SUSHI-public/FMH/herve

and compressed instructions that add 16-bit encodings for common operations) and extend our support of the privileged version of the specification [31].

Although this objective is distant, it can be split into actionable, incremental steps. We could start from an RV32I core and gradually ramp up to more complex designs (adding extensions one at a time). At first, our focus should be on feature-completeness. The introduction of advanced optimizations can be handled separately from this task.

Achieving this would represent a significant milestone, enabling the formal verification of interesting hardware-software stacks in the line of Lööw et al. [140] and the Verisoft stack [164]. The prospect of running CompCert [165] on a seL4 microkernel [166], itself running on a verified processor, is particularly alluring. Reaching that goal would allow us to ground the hardware hypotheses that are often implicit in the formal verification of software.

6.3.2 Validation of security mechanisms

In the previous section, we scratched the surface of what formal hardware verification may enable. Verifying the adherence of a processor to a formal ISA specification can be used to ground hardware-software contracts: the software may safely assume that the underlying hardware behaves in a certain way. However, there are tasks for which additional guarantees are required. For instance, we may want to prove that a cryptographic process executes in constant time and that the information it manipulates cannot be leaked to other processes.

In 2015, Intel introduced SGX (Software Guard Extensions), a set of extensions related to trusted execution environments (i.e., environments for running code in isolation from the rest of the systems). In 2021, after a string of exploits [167], this extension was deprecated for the main line of Intel processors.

Lau et al. [116] describe verifying a comparable mechanism for process isolation. They formally prove that their system provides a strong notion of isolation, which implies that they are effectively clear of leakage issues that weakened SGX. This work clearly illustrates the benefits of a proof assistant-based verification process.

The additional guarantees that security mechanisms claim to bring are a natural target for formal verification. In this thesis, we gave the example of verifying a shadow stack. We could replicate this mechanism in COQQTL and extend it incrementally to bring it closer to what Intel's CET [37] includes. The UEFI secure boot process [168] would also be an interesting target for verification. Indeed, a vulnerable boot process strongly limits any OS-provided guarantees. In particular, formal guarantees provided by systems such as seL4 [166] may be broken. Verifying such mechanisms on realistic processors (as described in the previous section) would set this work aside from the prior art.

6.3.3 Flexibility and automation

In this thesis, we consider the verification of fixed designs, which are often intentionally structured to simplify the verification process. The same is true for broader academia. While verifying such designs is a significant challenge in its own right, it is different from verifying industrial designs that are constantly evolving throughout their development. The value of a hardware verification methodology is closely linked to its usability. For a methodology to be compatible with industry needs, it must be cost-efficient and flexible enough to accommodate frequent changes.

Part of the answer comes in the form of methodological adaptations. Hardware changes tend to be local, which should be reflected in our verification methodology: small changes should have a small impact on the overall verification process. Modular reasoning and refinement have been successfully used in various projects to this end [114], [119].

We view quality integration of automation as a prerequisite for the further success of proof assistants-based verification. While Coq may not excel in this area, as it lacks native general-purpose tactics comparable to Isabelle's Hammer, third-party plugins are being developed within the community to meet this need. Automation contributes to flexibility: powerful automatic tactics may foster a highly succinct style of proof scripts that do not describe functional details. These details are filled in automatically at the last moment; changes are implicity taken into account.

We expect improvements in methodology to arise naturally as we tackle increasingly complex examples. Automation is a key component of the path of least resistance for such endeavors.

Section recap

In this section, we summarized the work done in this thesis and presented avenues for further research. For the short term, we outlined the essential implementation steps required for building a production-ready verification framework around COQQTL. For the long term, we stressed the importance of automatic procedures. We proposed the full verification of a processor capable of running Linux as a meaningful milestone to aim for.

Closing remarks

Like any other good basis for a thesis subject, hardware verification is a challenging problem. Considering its importance, it is also woefully understudied: hardware security forms the basis of all cybersecurity. Although chip designers have identified this issue, the solutions applied in the industry fall short of answering this challenge in its most general form. Proof assistants are powerful and trustworthy enough for this task. However, they are also prohibitively complex tools. Efficient, tried and tested verification methodologies and good support for delegation to automatic tools are required to make this approach compatible with industrial requirements.

Acronyms

ABI Application Binary Interface. 83

BMC Bounded Model Checking. 58

CC Common Criteria for Information Technology Security Evaluation. 32, 35

CERT Computer Emergency Response Team. 32

CET Control-flow Enforcement Technology. 35, 139

CISC Complex Instruction Set Computer. 83

CPU Central Processing Unit. 47, 55, 60–62, 96, 132

CRV Constrained Random Verification. 57

DSL Domain Specific Language. 39, 62, 63

EAL Evaluation Assurance Level. 35, 58

EDA Electronics Design Automation. 26, 57

FPGA Field-Programmable Gate Array. 26–28, 43, 90, 95

HCL Hardware Construction Language. 29

HDL Hardware Description Language. 25–29, 39, 43, 44, 57, 60, 61, 63, 64, 67, 99, 101, 102, 118, 133, 136

HLS High-Level Synthesis. 28

HSM Hardware Security Modules. 64

IR Intermediate Representation. 102, 133

IRR Intermediate Representation for Reasoning. 68–71, 73, 74, 76, 78, 79, 81, 88, 92, 93, 95, 101, 118–122, 125, 126, 128–133, 136

ISA Instruction Set Architecture. 25, 30, 60, 62, 64, 65, 83, 97, 138, 139

LUT LookUp Table. 90, 95

ORAAT One Rule At A Time. 51, 55

OS Operating System. 96, 97, 140

RISC Reduced Instruction Set Computer. 30, 47, 65, 83

RTL Register Transfer Level. 27, 28, 43, 61, 63, 64, 138

SAT boolean SATisfiability problem. 36, 38–40, 58

SGX intel Software Guard eXtensions. 139

SMT Satisfiability Modulo Theory problem. 3, 15, 16, 21, 38–40, 60, 63, 91–94, 98, 99, 119, 122, 128, 130–132, 136, 137, 158

TCB Trusted Computing Base. 21, 32, 35, 42, 59, 92, 93, 98, 133, 137

UVM Universal Verification Methodology. 57

Bibliography

[1] "Information security, cybersecurity and privacy protection — Evaluation criteria for IT security", ISO/IEC 15408-1:2022, 2022. https://www.iso.org/standard/72891.html (pages 15, 21, 35).

[2] A. Djoudi et al., "A bottom-up formal verification approach for common criteria certification: application to JavaCard virtual machine", in *European Congress Embedded Real Time System*, 2022. https://hal.science/hal-03695829 (pages 15, 21).

[3] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem", in *LASER Summer School on Software Engineering*, 2011. DOI: 10.1007/978-3-642-35746-6_1 (pages 15, 21, 42).

[4] M. Baty, P. Wilke, G. Hiet, A. Fontaine, and A. Trieu, "A generic framework to develop and verify security mechanisms at the microarchitectural level: application to control-flow integrity", in *IEEE Computer Security Foundations Symposium*, 2023. DOI: 10.1109/csf57540.2023.00029 (pages 16, 22).

[5] M. Baty, G. Hiet, and P. Wilke, "Work in progress: a formally verified shadow stack for RISC-V", in *Workshop on the Security of Software/Hardware Interfaces*, 2022. https://silm-workshop.github.io/2022-papers/silm2022-shadowstack.pdf (pages 16, 22).

[6] J. J. P. Eckert and J. W. Mauchly, "Electronic numerical integrator and computer", US Patent 3,120,606, https://patents.google.com/patent/US3120606A/en, Google Patents, 1964 (page 26).

[7] P. Flake, P. Moorby, S. Golson, A. Salz, and S. Davidmann, "Verilog HDL and its ancestors and descendants", *Proceedings of the ACM on Programming Languages*, 2020. DOI: 10.1145/3386337 (page 26).

[8] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Sequential circuit design using synthesis and optimization", in *IEEE International Conference on Computer Design: VLSI in Computers & Processors*, 1992. DOI: 10.1109/ICCD.1992.276282 (page 27).

[9] R. K. Brayton et al., "VIS: a system for verification and synthesis", in *Computer Aided Verification*, 1996. DOI: 10.1007/3-540-61474-5_95 (page 27).

[10] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic, "Yosys+nextpnr: an open source framework from Verilog to Bitstream for commercial FPGAs", in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2019. DOI: 10.1109/fccm.2019.00010 (pages 27, 59).

[11] "IEEE standard for Verilog hardware description language", *IEEE Std 1364-2005*, 2006. DOI: 10.1109/IEEESTD.2006.99495 (page 28).

[12] "IEEE standard for VHDL language reference manual", *IEEE Std 1076-2019*, 2019. DOI: 10.1109/IEEESTD.2019.8938196 (page 28).

[13] "IEEE Standard for SystemVerilog – unified hardware design, specification, and verification language", *IEEE Std 1800-2023*, 2024. DOI: 10.1109/IEEESTD.2024.10458102 (page 28).

[14] "IEEE Standard for Standard SystemC® Language Reference Manual", *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)*, 2023. DOI: 10.1109/IEEESTD.2023.10246125 (page 28).

[15] A. Canis et al., "LegUp: high-level synthesis for FPGA-based processor/accelerator systems", in *ACM International Symposium on Field Programmable Gate Arrays*, 2011. DOI: 10.1145/1950413.1950423 (page 28).

[16] *Intel HLS webpage*, https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html (page 28).

[17] *AMD Vitis HLS webpage*, https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html (page 28).

[18] *Circt website*, https://circt.llvm.org/ (pages 29, 102).

[19] *Chisel website*, https://www.chisel-lang.org/ (page 29).

[20] *Spinal HDL website*, https://spinalhdl.github.io/SpinalDoc-RTD/master/index.html (page 29).

[21] *Clash website*, https://clash-lang.org/ (page 29).

[22] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "Cλash: structural descriptions of synchronous hardware using Haskell", in *Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, 2010. DOI: 10.1109/DSD.2010.21 (page 29).

[23] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in Haskell", *ACM SIGPLAN Notices*, 1998. DOI: 10.1145/291251.289440 (pages 29, 62).

[24] A. Ray, B. Devlin, F. Y. Quah, and R. Yesantharao, "Hardcaml: An OCaml hardware domain-specific language for efficient and robust design", *arXiv preprint arXiv:2312.15035*, 2023. DOI: 10.1145/3626202.3637586 (page 29).

[25] *MyHDL website*, https://myhdl.org/ (page 29).

[26] *Magma repository*, https://github.com/phanrahan/magma (page 29).

[27] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications", in *ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, 2004. DOI: 10.1109/MEMCOD.2004.1459818 (pages 29, 43, 44).

[28] *Intel x86 manuals*, https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (page 30).

[29] *Arm manuals*, file:///home/mbty/inb/DDI0553A_k_armv8m_arm.pdf (page 30).

[30] A. Waterman and K. Asanović, *The RISC-V instruction set manual, Volume I: unprivileged ISA*, https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf, 2019 (pages 30, 83).

[31] A. Waterman, K. Asanović, and J. Hauser, *The RISC-V instruction set manual, Volume II: privileged ISA*, https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf, 2021 (pages 30, 139).

[32] E. Spafford, “The internet worm program: an analysis”, *Purdue University*, 1988. DOI: 10.1145/66093.66095 (page 32).

[33] P. Kocher et al., “Spectre attacks: exploiting speculative execution”, *Communications of the ACM*, 2020. DOI: 10.1145/3399742 (pages 32, 136).

[34] M. Lipp et al., “Meltdown: reading kernel memory from user space”, *Communications of the ACM*, 2020. DOI: 10.1145/3357033 (pages 32, 136).

[35] *2023 CWE top 25 most dangerous software weaknesses*, https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html (page 33).

[36] B. Bierbaumer, J. Kirsch, T. Kittel, A. Francillon, and A. Zarras, “Smashing the stack protector for fun and profit”, in *ICT Systems Security and Privacy Protection*, 2018. DOI: 10.1007/978-3-319-99828-2_21 (page 34).

[37] V. Shanbhogue, D. Gupta, and R. Sahita, “Security analysis of processor instruction set architecture for enforcing control-flow integrity”, in *International Workshop on Hardware and Architectural Support for Security and Privacy*, 2019. DOI: 10.1145/3337167.3337175 (pages 35, 139).

[38] Euclid, “The Elements of Euclid”. https://archive.org/details/euclid_heath_2nd_ed (page 37).

[39] B. Russell, “Letter to Frege”, 1902, ISBN: 9780674324497 (page 37).

[40] B. Russell, “Principles of Mathematics”, 1903. DOI: 10.4324/9780203822586 (page 37).

[41] A. Whitehead and B. Russell, “Principia Mathematica to 56”, 1927. DOI: 10.1017/CBO9780511623585 (page 37).

[42] K. Gödel, “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I”, *Monatshefte für Mathematik und Physik*, 1931. DOI: 10.1007/BF01700692 (page 37).

[43] A. M. Turing, “On computable numbers, with an application to the Entscheidungsproblem”, *Journal of Mathematics*, 1936. DOI: 10.1093/oso/9780198250791.003.0005 (page 38).

[44] A. Church, “A set of postulates for the foundation of logic”, *Annals of mathematics*, 1932. DOI: 0.2307/1968337 (page 38).

[45] N. G. De Bruijn, “A survey of the project AUTOMATH”, in *Studies in Logic and the Foundations of Mathematics*, 1994. DOI: 10.1016/S0049-237X(08)70203-9 (page 38).

[46] M. Davis, G. Logemann, and D. Loveland, “A machine program for theorem-proving”, *Commun. ACM*, 1962, ISSN: 0001-0782. DOI: 10.1145/368273.368557 (page 38).

[47] *Coq website*, https://coq.inria.fr/ (page 38).

[48] J. S. Gross, “Performance engineering of proof-based software systems at scale”, Ph.D. dissertation, Massachusetts Institute of Technology, 2021. http://adam.chlipala.net/theses/jgross.pdf (page 39).

[49] E. Seligman, T. Schubert, and M. V. A. K. Kumar, “Formal verification: an essential toolkit for modern VLSI design”, San Francisco, CA, USA, 2015, ISBN: 9780128008157 (page 42).

[50] E. M. Clarke, “Model checking”, in *Foundations of Software Technology and Theoretical Computer Science*, 1997. DOI: 10.1007/BFb0058022 (page 42).

[51] H. Garavel and F. Lang, “Equivalence checking 40 years after: a review of bisimulation tools”, *A Journey from Process Algebra via Timed Automata to Model Learning: Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, 2022. DOI: 10.1007/978-3-031-15629-8_13 (page 42).

[52] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement”, in *Computer Aided Verification*, 2000. DOI: 10.1007/10722167_15 (page 42).

[53] K. Y. Rozier, “Linear temporal logic symbolic model checking”, *Computer Science Review*, 2011. DOI: 10.1016/j.cosrev.2010.06.002 (page 42).

[54] “IEEE Standard for Property Specification Language (PSL)”, *IEEE Std 1850-2010*, 2010. DOI: 10.1109/IEEESTD.2010.5446004 (page 42).

[55] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, “The essence of Bluespec: a core language for rule-based hardware design”, in *ACM International Conference on Programming Language Design and Implementation*, 2020. DOI: 10.1145/3385412.3385965 (pages 43, 61, 67).

[56] C. Pit-Claudel, T. Bourgeat, S. Lau, Arvind, and A. Chlipala, “Effective simulation and debugging for a high-level hardware language using software compilers”, in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021. DOI: 10.1145/3445814.3446720 (page 43).

[57] “IEEE standard for Universal Verification Methodology language reference manual”, *IEEE Std 1800.2-2020*, 2020. DOI: 10.1109/IEEESTD.2020.9195920 (page 57).

[58] M. Sutton, A. Greene, and P. Amini, “Fuzzing: brute force vulnerability discovery”, Pearson Education, 2007. DOI: 10.5555/1324770 (page 57).

[59] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, “Fuzzing hardware like software”, in *USENIX Security Symposium*, 2022. https://www.usenix.org/conference/usenixsecurity22/presentation/trippel (page 57).

[60] *Synopsys VCS webpage*, https://www.synopsys.com/verification/simulation/vcs.html (page 58).

[61] *Cadence Incisive Functional Safety Simulator webpage*, https://newstaging.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-functional-safety-simulator.html (page 58).

[62] *Siemens Questa Advanced Simulator webpage*, https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/ (page 58).

[63] *Aldec Riviera-PRO webpage*, https://www.aldec.com/en/products/functional_verification/riviera-pro (page 58).

[64] L. De Moura, H. Rueß, and M. Sorea, "Bounded model checking and induction: from refutation to verification", in *International Conference on Computer Aided Verification*, 2003. DOI: 10.1007/978-3-540-45069-6_2 (page 58).

[65] *Siemens Questa OneSpin Formal Verification Suite webpage*, https://eda.sw.siemens.com/en-US/ic/questa/formal-verification/ (page 58).

[66] *Jasper FPV App webpage*, https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-verification-platform/formal-property-verification-app.html (page 58).

[67] *Cadence Conformal webpage*, https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/conformal-overview.html (page 59).

[68] *Synopsys Formality Equivalence Checking webpage*, https://www.synopsys.com/implementation-and-signoff/signoff/formality-equivalence-checking.html (page 59).

[69] *Hardware model checking competition 2020*, https://fmv.jku.at/hwmcc20/index.html (page 59).

[70] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "BTOR2, BtorMC and Boolector 3.0", in *International Conference on Computer Aided Verification*, 2018. DOI: 10.1007/978-3-319-96145-3_32 (page 59).

[71] A. Goel and K. Sakallah, "AVR: abstractly verifying reachability", in *Tools and Algorithms for the Construction and Analysis of Systems*, 2020. DOI: 10.1007/978-3-030-45190-5_23 (page 59).

[72] A. R. Bradley, "Understanding IC3", in *International Conference on Theory and Applications of Satisfiability Testing*, 2012. DOI: 10.1007/978-3-642-31612-8_1 (page 59).

[73] A. Cimatti et al., "NuSMV 2: An opensource tool for symbolic model checking", in *Computer Aided Verification*, 2002. DOI: 10.1007/3-540-45657-0_29 (page 59).

[74] A. Mishchenko et al., "ABC: a system for sequential synthesis and verification" (page 59).

[75] S. A. Seshia and P. Subramanyan, "UCLID5: integrating modeling, verification, synthesis and learning", in *ACM/IEEE International Conference on Formal Methods and Models for System Design*, 2018. DOI: 10.1109/MEMCOD.2018.8556946 (page 59).

[76] *SimbiYosys documentation*, https://symbiyosys.readthedocs.io/en/latest/index.html (page 59).

[77] N. Moroze, A. Athalye, M. F. Kaashoek, and N. Zeldovich, "rtlv: push-button verification of software on hardware", 2021. https://pdos.csail.mit.edu/papers/rtlv:carrv21.pdf (page 59).

[78] E. Torlak and R. Bodik, "Growing solver-aided languages with Rosette", in *International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2013. DOI: 10.1145/2509578.2509586 (pages 59, 64).

[79] K. Nienhuis et al., "Rigorous engineering for hardware security: formal modelling and proof in the CHERI design and implementation process", in *IEEE Symposium on Security and Privacy*, 2020. DOI: 10.1109/SP40000.2020.00055 (page 60).

[80] T. Bauereiss et al., “Verified security for the Morello capability-enhanced prototype Arm architecture”, in *Programming Languages and Systems*, 2022. DOI: 10.1007/978-3-030-99336-8_7 (page 60).

[81] A. L. Georges et al., “Efficient and provable local capability revocation using uninitialized capabilities”, *Proceedings of the ACM on Programming Languages*, 2021. DOI: 10.1145/3434287 (page 60).

[82] A. L. Georges, A. Trieu, and L. Birkedal, “Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities”, *Proceedings of the ACM on Programming Languages*, 2022. DOI: 10.1145/3527318 (page 60).

[83] L. Skorstengaard, D. Devriese, and L. Birkedal, “Reasoning about a machine with local capabilities: provably safe stack and return pointer management”, *ACM Transactions on Programming Language and Systems*, 2020. DOI: 10.1145/3363519 (page 60).

[84] L. Skorstengaard, D. Devriese, and L. Birkedal, “StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities”, *Journal of Functional Programming*, 2021. DOI: 10.1017/S095679682100006X (page 60).

[85] T. V. Strydonck et al., “Proving full-system security properties under multiple attacker models on capability machines”, in *Computer Security Foundations Symposium*, 2022. DOI: 10.1109/CSF54842.2022.9919645 (page 60).

[86] X. Li et al., “Caisson: a hardware description language for secure information flow”, in *Conference on Programming Language Design and Implementation*, 2011. DOI: 10.1145/1993498.1993512 (page 60).

[87] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, “A hardware design language for timing-sensitive information-flow security”, in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015. DOI: 10.1145/2694344.2694372 (page 60).

[88] K. von Gleissenthall, R. G. Kici, D. Stefan, and R. Jhala, “IODINE: verifying constant-time execution of hardware”, in *USENIX Security Symposium*, 2019. https://www.usenix.org/conference/usenixsecurity19/presentation/von-gleissenthall (page 60).

[89] K. von Gleissenthall, R. G. Kici, D. Stefan, and R. Jhala, “Solver-aided constant-time hardware verification”, in *Conference on Computer and Communications Security*, 2021. DOI: 10.1145/3460120.3484810 (page 60).

[90] F. Lisboa Malaquias, M. Asavoae, and F. Brandner, “A Coq framework for more trustworthy DRAM controllers”, in *International Conference on Real-Time Networks and Systems*, 2022. DOI: 10.1145/3534879.3534907 (page 61).

[91] F. Lisboa Malaquias, G. Giantamidis, S. Basagiannis, S. Fulvio Rollini, and I. Amundson, “Towards a methodology to design provably secure cyber-physical systems”, *ACM SIGADA Ada Letters*, 2023. DOI: 10.1145/3631483.3631499 (page 61).

[92] T. Letan, P. Chifflier, G. Hiet, P. Néron, and B. Morin, “SpecCert: specifying and verifying hardware-based security enforcement”, in *International Symposium on Formal Methods*, 2016. DOI: 10.1007/978-3-319-48989-6_30 (pages 61, 97).

[93] Y. Hsiao, D. P. Mulligan, N. Nikoleris, G. Petri, and C. Trippel, “Synthesizing formal models of hardware from RTL for efficient verification of memory model implementations”, in *IEEE/ACM International Symposium on Microarchitecture*, 2021. DOI: 10.1145/3466752.3480087 (page 61).

[94] Y. Hsiao et al., “RTL2MμPATH: multi-μPATH synthesis with applications to hardware security verification”, *arXiv preprint arXiv:2409.19478*, 2024. DOI: 0.48550/arXiv.2409.19478 (page 61).

[95] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, “Kami: a platform for high-level parametric hardware specification and its modular verification”, *Proceedings of the ACM on Programming Languages*, 2017. DOI: 10.1145/3110268 (page 61).

[96] T. Bourgeat, “Specification and verification of sequential machines in rule-based hardware languages”, Massachusetts Institute of Technology, 2023. https://hdl.handle.net/1721.1/150194 (page 61).

[97] Y. Herklotz, J. D. Pollard, N. Ramanathan, and J. Wickerson, “Formal verification of high-level synthesis”, *Proceedings of the ACM on Programming Languages*, 2021. DOI: 10.1145/3485494 (page 61).

[98] Y. Herklotz and J. Wickerson, “Hyperblock scheduling for verified high-level synthesis”, *Proceedings of the ACM on Programming Languages*, 2024. DOI: 10.1145/3656455 (page 61).

[99] A. Lööw and M. O. Myreen, “A proof-producing translator for Verilog development in HOL”, in *International Workshop on Formal Methods in Software Engineering*, 2019. DOI: 10.1109/FormaliSE.2019.00020 (pages 61, 65).

[100] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, “CakeML: a verified implementation of ML”, in *ACM Symposium on Principles of Programming Languages*, 2014. DOI: 10.1145/2535838.2535841 (pages 61, 65).

[101] N. Dong, R. Guanciale, M. Dam, and A. Lööw, “Formal verification of correctness and information flow security for an in-order pipelined processor”, in *Formal Methods in Computer-Aided Design*, 2023. DOI: 10.34727/2023/isbn.978-3-85448-060-0_33 (page 62).

[102] N. Dong, “Towards a trustworthy stack: formal verification of low-level hardware and software”, Ph.D. dissertation, KTH Royal Institute of Technology, 2024. https://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A1821827&dswid=4955 (page 62).

[103] A. Lööw, “Lutsig: a verified Verilog compiler for verified circuit development”, in *ACM International Conference on Certified Programs and Proofs*, 2021. DOI: 10.1145/3437992.3439916 (pages 62, 65).

[104] A. Lööw, “Reconciling verified-circuit development and Verilog development”, in *Conference on Formal Methods in Computer-Aided Design*, 2022. DOI: 10.34727/2022/isbn.978-3-85448-053-2_15 (page 62).

[105] *Project Silver Oak repository (public archive)*, https://github.com/project-oak/silveroak (page 62).

[106] *OpenTitan website*, https://opentitan.org/ (page 62).

[107] G. Berry and G. Gonthier, "The Esterel synchronous programming language: design, semantics, implementation", *Science of computer programming*, 1992. https://hal.science/inria-00075711 (page 62).

[108] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE", *Proceedings of the IEEE*, 1991. DOI: 10.1109/5.97300 (page 62).

[109] *SCADE website*, https://www.ansys.com/products/embedded-software/ansys-scade-suite (page 62).

[110] *DO-178C website*, https://www.rtca.org/do-178/ (page 62).

[111] J.-L. Colaço, B. Pagano, C. Pasteur, and M. Pouzet, "SCADE 6: From a Kahn semantics to a Kahn implementation for multicore", in *Forum on Specification & Design Languages*, 2018. DOI: https://hal.science/hal-01960410 (page 62).

[112] *Esterel website*, https://www-sop.inria.fr/meije/esterel/esterel-eng.html (page 62).

[113] G. Berry and L. Rieg, "Towards Coq-verified Esterel semantics and compiling", *arXiv preprint arXiv:1909.12582*, 2019. https://www.arxiv.org/pdf/1909.12582v2 (page 62).

[114] A. Erbsen, S. Gruetter, J. Choi, C. Wood, and A. Chlipala, "Integration verification across software and hardware for a simple embedded system", in *ACM International Conference on Programming Language Design and Implementation*, 2021. DOI: 10.1145/3453483.3454065 (pages 63, 140).

[115] J. Choi, A. Chlipala, and Arvind, "Hemiola: a DSL and verification tools to guide design and proof of hierarchical cache-coherence protocols", in *International Conference on Computer Aided Verification*, 2022. DOI: 10.1007/978-3-031-13188-2_16 (page 63).

[116] S. Lau, T. Bourgeat, C. Pit-Claudel, and A. Chlipala, "Specification and verification of strong timing isolation of hardware enclaves", 2024. http://adam.chlipala.net/papers/IsolationCCS24/ (pages 63, 139).

[117] F. L. Malaquias, M. Asavoae, and F. Brandner, "From the standards to silicon: formally proved memory controllers", in *NASA Formal Methods Symposium*, 2023. DOI: 10.1007/978-3-031-33170-1_18 (page 63).

[118] A. Athalye, M. F. Kaashoek, and N. Zeldovich, "Verifying hardware security modules with information-preserving refinement", in *Symposium on Operating Systems Design and Implementation*, 2022. https://www.usenix.org/conference/osdi22/presentation/athalye (page 64).

[119] A. Athalye, H. Corrigan-Gibbs, M. F. Kaashoek, J. Tassarotti, and N. Zeldovich, "Modular verification of secure and leakage-free systems: from application specification to circuit-Level implementation", in *ACM Symposium on Operating Systems Principles*. https://pdos.csail.mit.edu/papers/parfait:sosp24.pdf (pages 64, 140).

[120] M.-M. Bidmeshki and Y. Makris, "VeriCoq: a Verilog-to-Coq converter for proof-carrying hardware automation", in *IEEE International Symposium on Circuits and Systems*, 2015. DOI: 10.1109/ISCAS.2015.7168562 (page 64).

[121] E. Love, Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition", *IEEE Transactions on Information Forensics and Security*, 2011. DOI: 10.1109/TIFS.2011.2160627 (page 64).

[122] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation", in *IEEE Symposium on Security and Privacy*, 2021. DOI: 10.1109/SP40001.2021.00036 (page 64).

[123] W. A. J. Hunt and B. C. Brock, "A formal HDL and its use in the FM9001 verification", *Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences*, 1992. DOI: 10.1098/rsta.1992.0024 (page 64).

[124] W. R. Bevier, "Kit: a study in operating system verification", *IEEE Transactions on Software Engineering*, 1989. DOI: 10.1109/32.41331 (page 64).

[125] W. D. Young, "A verified code generator for a subset of Gypsy", Ph.D. dissertation, 1988. DOI: 10.5555/915193 (page 64).

[126] A. D. Flatau, "A verified implementation of an applicative language with dynamic storage allocation", The University of Texas at Austin, 1992. DOI: 10.5555/166802 (page 64).

[127] J. S. Moore, "A grand challenge proposal for formal methods: a verified stack", in *Formal Methods at the Crossroads. From Panacea to Foundational Support: 10th Anniversary Colloquium of UNU/IIST*, 2003. DOI: 10.1007/978-3-540-40007-3_11 (page 64).

[128] B. C. Brock and W. A. Hunt Jr, "The DUAL-EVAL hardware description language and its use in the formal specification and verification of the FM9001 microprocessor", *Formal Methods in System Design*, 1997. DOI: 10.1109/ASPDAC.1995.486381 (page 64).

[129] *ACL2 website*, https://www.cs.utexas.edu/~moore/acl2/ (page 64).

[130] W. A. Hunt Jr, M. Kaufmann, J. S. Moore, and A. Slobodova, "Industrial hardware and software verification with ACL2", *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 2017. DOI: 10.1098/rsta.2015.0399 (page 64).

[131] S. Goel, A. Slobodová, R. Sumners, and S. Swords, "Verifying x86 instruction implementations", in *International Conference on Certified Programs and Proofs*, 2020. DOI: 10.48550/arXiv.1912.10285 (page 64).

[132] E. Alkassar, M. A. Hillebrand, D. Leinenbach, N. W. Schirmer, and A. Starostin, "The Verisoft approach to systems verification", in *Verified Software: Theories, Tools, Experiments*, 2008. DOI: 10.1007/978-3-540-87873-5_18 (page 65).

[133] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul, "Putting it all together – formal verification of the VAMP", *International Journal on Software Tools for Technology Transfer*, 2006. DOI: 10.1007/s10009-006-0204-6 (page 65).

[134] S. Owre, J. M. Rushby, and N. Shankar, “PVS: a prototype verification system”, in *International Conference on Automated Deduction*, 1992. DOI: 10.1007/3-540-55602-8_217 (page 65).

[135] *PVS website*, https://pvs.csl.sri.com/ (page 65).

[136] T. Nipkow, M. Wenzel, and L. C. Paulson, “Isabelle/HOL: a proof assistant for higher-order logic”, 2002. DOI: 10.1007/3-540-45949-9 (page 65).

[137] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul, “On the correctness of operating system kernels”, in *Theorem Proving in Higher Order Logics*, 2005. DOI: 10.1007/11541868_1 (page 65).

[138] D. Leinenbach and E. Petrova, “Pervasive compiler verification–from verified programs to verified systems”, *Electronic Notes in Theoretical Computer Science*, 2008. DOI: 10.1016/j.entcs.2008.06.040 (page 65).

[139] M. Kovalev, S. M. Müller, and W. J. Paul, “A pipelined multi-core MIPS machine: hardware implementation and correctness proof”, 2014. DOI: 10.1007/978-3-319-13906-7 (page 65).

[140] A. Lööw et al., “Verified compilation on a verified processor”, in *Conference on Programming Language Design and Implementation*, 2019. DOI: 10.1145/3314221.3314622 (pages 65, 139).

[141] P. Meredith, M. Katelman, J. Meseguer, and G. Roşu, “A formal executable semantics of Verilog”, in *ACM/IEEE International Conference on Formal Methods and Models for Codesign*, 2010. DOI: 10.1109/MEMCOD.2010.5558634 (page 67).

[142] Q. Chen et al., “The essence of Verilog: a tractable and tested operational semantics for Verilog”, *Proceedings of the ACM on Programming Languages*, 2023. DOI: 10.1145/3622805 (page 67).

[143] J. Choi, “Formal semantics of Verilog, revisited for deductive verification”, https://joonwon.net/c/docs/pfv-draft.pdf (page 67).

[144] Y. Herklotz, D. Demange, and S. Blazy, “Mechanised semantics for gated static single assignment”, in *ACM International Conference on Certified Programs and Proofs*, 2023. DOI: 10.1145/3573105.3575681 (page 71).

[145] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity”, in *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, 2018. DOI: 10.1145/3129743.3129748 (page 82).

[146] C. Barrett, P. Fontaine, C. Tinelli, et al., *The SMT-LIB standard: version 2.6*, 2021. https://smt-lib.org/papers/smt-lib-reference-v2.6-r2021-05-12.pdf (page 93).

[147] *SMT-LIB website*, https://smt-lib.org/ (page 93).

[148] L. De Moura and N. Bjørner, “Z3: an efficient SMT solver”, in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008. DOI: 10.1007/978-3-540-78800-3_24 (page 93).

[149] A. Armstrong et al., “ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS”, *Proceedings of the ACM on Programming Languages*, 2019. DOI: 10.1145/3290384 (pages 97, 138).

[150] M. R. Clarkson and F. B. Schneider, “Hyperproperties”, *Journal of Computer Security*, 2010. doi: 10.3233/JCS-2009-0393 (page 98).

[151] B. Ekici et al., “SMTCoq: A plug-in for integrating SMT solvers into Coq”, in *Computer Aided Verification*, 2017. doi: 10.1007/978-3-319-63390-9_7 (pages 98, 131, 137).

[152] K. Asanovic et al., “The Rocket chip generator”, *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.pdf (pages 101, 132, 137).

[153] D. Lockhart et al., “Experiences building Edge TPU with Chisel” (page 101).

[154] *FIRRTL specification*. https://github.com/chipsalliance/firrtl-spec (pages 101, 103, 124).

[155] D. Potop-Butucaru, S. A. Edwards, and G. Berry, “Compiling Esterel”, 2007. doi: 10.1007/978-0-387-70628-3 (page 110).

[156] *BOOM website*, https://boom-core.org/ (pages 132, 137).

[157] Ł. Czajka and C. Kaliszyk, “Hammer for Coq: automation for dependent type theory”, *Journal of Automated Reasoning*, 2018. doi: 10.1007/s10817-018-9458-4 (page 137).

[158] P. J. Windley, “Formal modeling and verification of microprocessors”, *IEEE Transactions on Computers*, 1995. doi: 10.1109/12.368009 (page 138).

[159] K. L. McMillan, “Verification of an implementation of Tomasulo’s algorithm by compositional model checking”, in *Computer Aided Verification*, 1998. doi: 10.1007/BFb0028738 (page 138).

[160] D. Kroening, W. J. Paul, and S. M. Mueller, “Proving the correctness of pipelined micro-architectures”, in *MBMV*, 2000. https://www-wjp.cs.uni-saarland.de/publikationen/KMP00.pdf (page 138).

[161] R. Jhala and K. L. McMillan, “Microarchitecture verification by compositional model checking”, in *Computer Aided Verification*, 2001. doi: 10.1007/3-540-44585-4_40 (page 138).

[162] J. Sawada and W. A. Hunt, “Verification of FM9801: an out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability”, *Formal Methods in System Design*, 2002. doi: 10.1023/A:1014122630277 (page 138).

[163] A. Reid et al., “End-to-end verification of processors with ISA-Formal”, in *Computer Aided Verification*, 2016. doi: 10.1007/978-3-319-41540-6_3 (page 138).

[164] D. Leinenbach and E. Petrova, “Pervasive compiler verification — from verified programs to verified systems”, *Electronic Notes in Theoretical Computer Science*, 2008. doi: 10.1016/j.entcs.2008.06.040 (page 139).

[165] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, “CompCert — a formally verified optimizing compiler”, in *Embedded Real Time Software and Systems*, 2016. https://inria.hal.science/hal-01238879v1 (page 139).

[166] G. Klein et al., “Comprehensive formal verification of an OS microkernel”, *TOCS*, 2014. doi: 10.1145/2560537 (pages 139, 140).

[167] A. Nilsson, P. N. Bideh, and J. Brorsson, “A survey of published attacks on Intel SGX”, *arXiv preprint arXiv:2006.13598*, 2020. DOI: 10.48550/arXiv.2006.13598 (page 139).

[168] R. Wilkins and B. Richardson, “UEFI secure boot in modern computer security solutions”, in *UEFI forum*, 2013. https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2014/06/20082511/UEFI_Secure_Boot_in_Modern_Computer_Security_Solutions_2013-1.pdf (page 139).

[169] L. De Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer, “The Lean theorem prover (system description)”, in *International Conference on Automated Deduction*, 2015. DOI: 10.1007/978-3-319-21401-6_26.

Titre : Spécification et vérification formelle de mécanismes de sécurité pour processeurs RISC-V

Mots clés : méthodes formelles, Coq, vérification matérielle, microarchitecture, RISC-V

Résumé : Dans cette thèse, nous considérons la vérification de mécanismes de sécurité pour processeurs RISC-V décrits au niveau des transferts de registres. Nous proposons une approche basée sur des assistants de preuve, des outils généralistes produisant des preuves à haut degré de fiabilité. Cette approche présente deux avantages clés par rapport aux méthodes employées dans l'industrie : les assistants de preuve reposent sur une base de confiance limitée et permettent d'exprimer et de vérifier des propriétés générales au sein d'un seul et même environnement. Cependant, il s'agit d'outils complexes et sujets à des problèmes de performance. Par ailleurs, ils ne sont pas spécialisés sur la vérification de matériel. La formalisation des éléments pertinents pour ce problème, tels que la sémantique de langages de description du matériel, est un prérequis pour la vérification. Nos contributions sont doubles. 1/ Nous proposons un cadriciel de vérification formelle basé sur l'assistant de preuve Coq et Kôika, un langage de description de matériel avec une sémantique formelle. Nous illustrons ces travaux par la vérification d'une pile fantôme intégrée à un processeur RISC-V. Par ailleurs, nous explorons l'intégration de solveurs SMT pour automatiser le processus de vérification. 2/ Nous proposons COQQTL, un portage du langage de description du matériel FIRRTL au sein de Coq.

Title: Formal specification and verification of security mechanisms for RISC-V processors

Keywords: formal methods, Coq, hardware verification, microarchitecture, RISC-V

Abstract: In this thesis, we consider the verification of security mechanisms for processors described at the register-transfer level. We propose an approach based on interactive theorem provers, generalist tools used for producing high-assurance proofs. This approach has two key benefits over the current state of formal methods in the industry. First, interactive theorem provers come with a very small trusted computing base. Second, they can express and verify very general properties in a single, unified environment. Our reliance on these tools comes with distinct challenges, such as the fact that all the domain-specific knowledge about hardware needs to formalized before any verification work can proceed. Furthermore, interactive theorem provers are arcane tools that come with a number of performance issues. Our contributions are twofold. 1/ We design a verification framework around Kôika, a hardware description language with a formal semantics, in the Coq prover. We use this framework to verify the implementation of a shadow stack added to a RISC-V processor. We explore the integration of SMT-solvers into this framework as a means of automating the verification process. 2/ We port FIRRTL, a hardware description language with use in the industry, into Coq.