Abstracting Hardware Architectures for Agile Design of High-performance Applications on FPGA

Jean Bruant

THÈSE

Pour obtenir le grade de

Université Grenoble Alpes

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École Doctorale : Électronique, Électrotechnique, Automatique et Traitement du Signal (EEATS)
Spécialité : Nano Électronique et Nano Technologies
Unitée de recherche : Laboratoire Techniques de l'Informatique et de la Microélectronique pour l'Architectures des systèmes intégrés (TIMA)

Abstracting Hardware Architectures for Agile Design of High-performance Applications on FPGA

Abstractions d'architectures matérielles pour une conception *Agile* d'applications performantes sur FPGA

Présentée par :

Jean BRUANT

Encadrement de Thèse :

Olivier Muller Maître de conférences, Grenoble INP, Université Grenoble Alpes	Directeur de thèse
Frédéric Pétrot Professeur des Universités, Grenoble INP, Université Grenoble Alpes	Co-encadrant de thèse
Pierre-Henri Horrein Ingénieur Docteur, OVHcloud	Co-encadrant de thèse

Thèse soutenue publiquement le **8 décembre 2022**, devant le jury composé de :

Florence Maraninchi Professeur des Universités, Grenoble INP, Université Grenoble Alpes	Présidente
Matthieu Arzel Professeur des Universités, Lab-STICC, IMT Atlantique Bretagne - Pays de la Loire	Rapporteur
Steven Derrien Professeur des Universités, ISTIC, Université Rennes 1	Rapporteur
Roselyne Chotin Maître de conférences, LIP6, Sorbonne Université	Examinatrice
Megan Wachs Ph.D., VP of Engineering, Sifive	Examinatrice
Olivier Muller Maître de conférences Grenoble INP, Université Grenoble Alpes	Directeur de thèse
Frédéric Pétrot Professeur des Universités, Grenoble INP, Université Grenoble Alpes	Co-encadrant de thèse
Pierre-Henri Horrein Ingénieur Docteur, OVHcloud	Co-encadrant de thèse

To our 24 sq.m,
at least as dense as this manuscript

Abstract

In a context of ever-growing worldwide communication traffic and fast deployment of IoT devices, network attacks have become a daily challenge with record-breaking throughput levels. Compared to software solutions based on general purpose CPUs, FPGA-based mitigation appliances appear as an energy-efficient alternative which combines configurability with guaranteed high-throughput and low-latency. However, implementation of such dedicated hardware accelerators based on the register-transfer level (RTL) abstraction is a much slower and tedious process than functionally equivalent software developments. The latter have indeed benefited from the introduction of countless high-level paradigms over the past decades, whereas traditional hardware description languages (HDLs) have consistently remained rigid and verbose. As a result, the agility gap between hardware and software developments is expanding at a steady pace, leaving hardware design experts frustrated by the lack of re-usability of their carefully crafted architectures.

This thesis tackles this generic hardware development issue within the context of high-performance networking appliance design at OVHcloud. Mimicking the successful trajectory of software evolution, it aims at leveraging a stack of abstraction levels to instill flexibility within hardware descriptions. As a key enabler, Hardware Construction Languages (HCLs) apply some existing software abstractions to hardware design, which permits descriptions of circuit generators with high-level software paradigms, such as object-oriented and functional programming. This thesis first exhibits the relevance of such software inherited paradigms to develop highly re-usable network functions, inspecting both implementation and design perspectives. Based on this powerful base layer, we introduce an additional hardware-oriented abstraction focusing on high-performance pipelined applications. Finally, the integration ability of these novel design methodologies within existing HDL hierarchies is reviewed in detail, yielding two final contributions aiming at ensuring a smooth cohabitation of both methodologies. The first one provides a direct path from existing HDL sources to their functionally equivalent HCL counterparts, thanks to an automated translation tool. This word-for-word translation is intended as the first step of an iterative manual upgrade to truly benefit from high-level abstractions of HCLs. The second one focuses on the smooth integration of HCL-generated hierarchies back into a top-level HDL hierarchy, which is a key acceptance factor for these new methodologies in long-running projects.

Résumé[1]

Dans un contexte de forte augmentation des communications numériques à travers le monde et de déploiement rapide de l'internet des objets (IoT), les attaques sur les réseaux de données sont devenues un défi quotidien avec des niveaux de trafic record. Par rapport aux solutions logicielles basées sur des processeurs généralistes, les dispositifs de mitigation construits à partir de FPGA apparaissent comme une alternative économe en énergie qui combine la configurabilité avec à la fois la garantie d'un haut débit et d'une faible latence. Cependant, la mise en œuvre de ces accélérateurs matériels dédiés, basée sur l'abstraction des circuits numériques au niveau registre (RTL), est un processus beaucoup plus lent et fastidieux que les développements logiciels fonctionnellement équivalents. Ces derniers ont en effet bénéficié de l'introduction de nombreux paradigmes de haut niveau au cours des dernières décennies, alors que les langages de description du matériel (HDL) traditionnels sont restés rigides et verbeux. En conséquence, l'écart d'agilité entre les développements matériels et logiciels se creuse à un rythme soutenu, laissant les experts en conception matérielle frustrés par le manque de réutilisabilité de leurs architectures si soigneusement élaborées.

Cette thèse aborde ce problème générique au développement matériel dans le contexte de la conception d'équipements réseau haute-performance chez OVHcloud. En imitant la trajectoire réussie de l'évolution des langages logiciels, elle vise à tirer parti d'un empilement de niveaux d'abstraction pour insuffler de la flexibilité au sein des descriptions matérielles. En particulier, les langages de construction matérielle (HCL) appliquent déjà certaines abstractions logicielles à la conception matérielle, ce qui permet de décrire des générateurs de circuits avec des paradigmes logiciels de haut niveau, tels que la programmation orientée objet et fonctionnelle. Cette thèse montre d'abord la pertinence de l'utilisation de tels paradigmes hérités du monde logiciel pour développer des fonctionnalités réseau hautement réutilisables, en s'intéressant à la fois aux perspectives de mise en œuvre et de conception. Sur cette base, nous présentons une abstraction supplémentaire, spécifique aux développements matériels, qui se concentre sur les applications *pipelinées* à haute-performance. Enfin, la capacité d'intégration de ces nouvelles méthodologies de conception dans les hiérarchies HDL existantes est examinée en détail, ce qui donne lieu à deux contributions finales visant à assurer une cohabitation harmonieuse entre ces deux méthodologies de développement. Grâce à un outil de traduction automatique, la première fournit un chemin direct depuis des sources HDL existantes vers une version HCL fonctionnellement équivalente. Cette traduction mot à mot est conçue comme la première étape d'une mise à niveau manuelle et itérative pour réellement bénéficier des abstractions de haut niveau fournies par les HCL. La seconde se concentre sur l'intégration sans accrocs des hiérarchies générées par les HCL dans une hiérarchie HDL, ce qui constitue un facteur d'acceptation essentiel de ces nouvelles méthodologies dans les projets au long cours.

[1] *Traduction basée sur un premier jet fourni par le traducteur DeepL à partir du résumé en langue originale.* https://www.deepl.com

Contents

Acknowledgments

I would first like to thank the members of my examining committee for the time they invested in my work: Florence Maraninchi, who accepted to chair the committee, Steven Derrien and Matthieu Arzel, who both accepted to report on the manuscript after being also part of three monitoring committees, Roselyne Chotin and Megan Wachs. The numerous questions they had prepared for the defense led to passionate discussions. A special mention to Megan Wachs, who accepted to be part of the committee of an unknown French PhD candidate on the other side of the world, despite a non-negligible logistical constraint: a 9-hours time difference.

Speaking of a considerable time difference, many thanks to my brother Hugues and his life companion Lilly who assiduously attended the defense, at a very early time in the morning for them as well.

I would then like to thank the members of the Chisel/FIRRTL open-source community, who have warmly welcomed me, despite my small contributions and sparse presence at the dev meetings. From the very beginning of this PhD journey, we knew we needed to pick a mature solution and a dynamic ecosystem to build our academic contributions while retaining viable industrial perspectives. A solution that would not let us down after a few years, and Chisel did not. Special thanks to Jack Koenig and Schuyler Eldridge who introduced me to the power of Chisel/FIRRTL annotations and custom transforms in the Scala FIRRTL compiler. I guess I will again need your help now that Chisel is moving to Circt and the MLIR-based FIRRTL compiler.

My research journey toward this PhD started well before my time at OVHcloud, with my one-year internship at WideNorth in Oslo, Norway. I am especially grateful to Helge, Hallvard and Nicolas who offered my fellow intern Benoît and I the opportunity of a one-year research project with full autonomy and absolute trust. Thank you, Helge for inviting me to give my very first scientific talk at the FPGA forum in Trondheim.

Finally, and before continuing in French, I must acknowledge the tremendous proofreading work provided by my beloved wife Clothilde. Her meticulous and rigorous reading aloud definitively improved the English writing quality of this manuscript.

Remerciements

La thèse est une aventure, certes personnelle, mais non sans jalons, sans repères, sans un balisage parfois patiemment anticipé par ceux qui en ont été les instigateurs. Tout a commencé dans le bâtiment *K2* de Telecom Bretagne au printemps 2015, par la découverte de l'électronique numérique et des FPGA, encadrée par notamment Matthieu Arzel et Pierre-Henri Horrein. Alors jeune étudiant, profitant du bon air marin breton et bien loin de m'imaginer docteur, je m'extasiais devant les résultats – quelques points rouges sur un fond noir – de l'algorithme de détection de contours Sobel que nous venions fièrement d'implémenter avec Benoît. Aux côtés de ce binôme indéfectible et passionné, nos chers encadrants nous vantaient, en toute honnêteté, les mérites de la recherche, tant académique qu'industrielle. Les perspectives d'une thèse apparaissaient alors loin, vraiment loin, au bout d'une formation d'ingénieur qui n'en était alors qu'à son commencement. Bien enterrée, la graine était semée. Une graine que Matthieu et Pierre-Henri ne manqueraient pas de soigner et d'éveiller à la liberté offerte par la recherche. C'est ainsi que, toujours aux côtés de mon fidèle compère Benoît, Matthieu nous offrit l'opportunité d'une année de stage en Norvège, pendant laquelle nous avons pu goûter au grand frisson d'une totale autonomie de recherche. Dès mon retour à Brest, Pierre-Henri me proposa un contrat de professionnalisation chez OVH, un cadre définitivement plus industriel, mais toujours guidé par l'ambition d'innover. C'est dans ce contexte que la thèse est apparue comme une suite logique à ce parcours de formation. Conçue sur mesure dans une confiance presque aveugle avec Tristan, Olivier, Frédéric et Pierre-Henri, l'aventure débuta finalement à l'automne 2018 : quatre saisons plus tard, la dormance était levée. Je manque de mots pour remercier les protagonistes de ce chemin vers la recherche, qui ne manquèrent d'ailleurs pas de rester des acteurs présents pendant la thèse.

Merci Matthieu pour le partage de ta passion, ton suivi régulier et toutes les opportunités que tu m'as offertes jusqu'au surlendemain de la soutenance.

Merci Benoît pour ton amitié sans faille, allant jusqu'à sacrifier les premières heures de tes vacances pour *voler* au secours de ma détresse en LaTeX. Merci pour ta rigueur, ton goût donné pour LaTeX et les présentations bien faites. Sans cette année à tes côtés en Norvège, je n'aurais sans doute jamais sauté le pas de la thèse.

Merci Pierre-Henri pour ton accompagnement et ton écoute attentive chaque jour depuis les prémices de la thèse, jusqu'à la soutenance et encore aujourd'hui pour poursuivre cette aventure qu'est la recherche, quelles que soient les conditions adverses sur ce noble chemin. Merci pour ces discussions passionnantes qui attisent ma curiosité et mon envie d'aller toujours plus loin. Merci pour ce tout, dont je ne me risquerais pas ici à dresser une liste exhaustive.

Merci à l'équipe FPGA qui m'a accueilli chez OVH, Tristan, Thibault, Clément et Pierre-Henri, puis Paul-Louis. Merci d'avoir été réceptifs à ma démarche de recherche en marge de l'équipe, avant, pendant et encore aujourd'hui après ces quatre années de thèse. Merci également à mes collègues OVHcloud de P19, qui m'ont soutenu jour après jour, en particulier pendant les longs mois de rédaction, entre cafés et plantations.

La thèse est une aventure, certes personnelle, mais surtout faite de rencontres. Merci à toute l'équipe SLS de TIMA qui malgré ma présence plus que sporadique à Grenoble, m'a réservé un accueil chaleureux lors de mes séjours trop peu nombreux. Merci Laurence, Julie, Arthur, Frédéric R., Liliana, Adrien, Maxime, Nathan, Marie, Benjamin et tous les compères de passage pour ces discussions passionnantes autour d'un thé, d'un café ou d'une viennoiserie.

Merci Bruno d'avoir été un fidèle co-bureau malgré la distance, et de m'avoir montré la voie jusqu'à la soutenance – *The best thesis defense is a good thesis offense*. Merci pour ton soutien tout au long de ces années avec ces discussions rafraichissantes au milieu de nos péripéties personnelles.

Merci Frédéric P. pour tes références éclairées dès l'écriture du sujet de thèse, quand *Chisel* et *HCL* étaient encore des termes obscurs. Merci pour tes relectures à toute heure, mais toujours attentives et efficaces de nos différents articles et de ce manuscrit.

Merci Olivier pour ta confiance absolue, ou quand une seule discussion désintéressée autour d'un verre suffit à engager une thèse à distance quelques mois plus tard. Merci pour ton accompagnement et ta présence continue malgré la distance maintenue par le Covid. Merci pour ton temps et ton dévouement sans faille, notamment dans les derniers jours jusqu'à la soutenance. Je n'oublierai pas non plus ta relecture marathon du manuscrit !

La thèse est une aventure, certes personnelle, mais partagée. Merci à mes parents, de m'avoir donné le goût de poursuivre de longues études, à mon Papa de m'avoir initié à la recherche, dès mon plus jeune âge lorsque je bricolais de petits circuits électriques pour Playmobil. Merci à mes amis et à ma famille pour leur soutien, j'ai été particulièrement touché par l'engouement suscité par la soutenance, assidûment suivie à distance par un grand nombre d'entre vous.

La thèse est une aventure, mais est-elle finalement si personnelle ? Elle finit par embrigader, bon gré mal gré, ceux qui gravitent de trop près autour. En quatre ans, la vie avance, d'étape en étape, péripéties après péripéties. Merci Clothilde d'avoir pris part à cette aventure à mes côtés, depuis 2015, dans le bonheur et dans les épreuves, dans la santé et dans la maladie.

Chapter 1

Introduction

While the need for specialized hardware accelerators arises in various contexts, this preliminary chapter intentionally focuses on the concrete industrial motivations which have led to the use of Field Programmable Gate Arrays (FPGAs) at the core of high-performance network devices at OVHcloud.

Built on universality and equality principles, the Internet is expected to provide a free and unrestricted access to its contents to all individuals. As stated by the IRPC[1] in their *10 Internet Rights and Principles*, "*everyone has an equal right to access and use a secure and open Internet*". This ideal and philanthropic vision is unfortunately endangered by many players, from dictatorships and authoritarian states to greedy companies to malicious individuals. Either to protect their interests or to arbitrary restrain freedom of others, they are determined to enforce their own rules by all means.

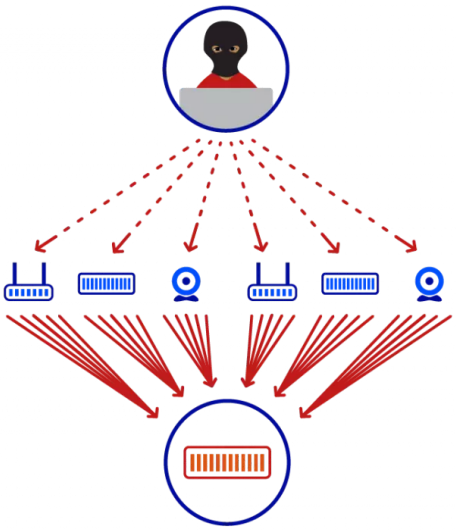


Figure 1.1: Distributed Denial of Service (DDoS) principle

Among the various attack strategies used to disrupt the Internet, one consists in saturating a given service, e.g. a webpage, by throwing at it more requests than it is capable to process. This results in a *Denial of Service* for legitimate users whom are not able to use the service anymore, e.g. they are unable to access the content of the webpage and instead

[1] "*The Internet Rights and Principles Dynamic Coalition is an open network of individuals and organizations based at the UN Internet Governance Forum (IGF) committed to making human rights and principles work for the online environment.*" Website: https://internetrightsandprinciples.org

reach a connection timeout without response. As the capacity of web services are sized to handle numerous requests from many users at the same time, generating a sufficient amount of traffic to reach saturation requires a considerable computational power and a solid network connection. To overcome this technical limitation, attackers instead rely on a large group of previously compromised devices, referred as a *botnet*, and order all devices to send as many requests as they can to the same target at a precise time [MSB+06]. Figure 1.1 illustrates such a *Distributed Denial of Service*, based on several kinds of compromised devices such as unsecured IP cameras or misconfigured home routers. Some malicious organizations continuously track down vulnerable devices to create large botnets and rent their harmful powers to the highest bidder, generating attacks above 1.5 terabits per second (Tbps) [Net20, MXSJ17]. The most famous DDoS victims include public administrations and controversial websites, however most attacks are now profit-driven [WCCM18]. In particular, the online gaming industry, which relies on low latencies to provide a satisfying experience to players, appears particularly prone to attacks [BGG+22].

OVHcloud Network Defense Infrastructure

As there are not only legitimate users of services on the Internet, part of the traffic directed to cloud provider services such as OVHcloud is ill-intentioned. Service providers must protect themselves from DDoS attacks, either targeting specific hosted services or even their own network infrastructure for a larger impact in case of successful disruption. Detecting rogue traffic and mitigating its impact as soon as it enters the network infrastructure, i.e. at the external connection points referred as Points of Presence (POP), is required to defend the network against high-throughput attacks. Early detection and mitigation of such attacks gives them no opportunity to saturate some network nodes before reaching a dedicated mitigation infrastructure.

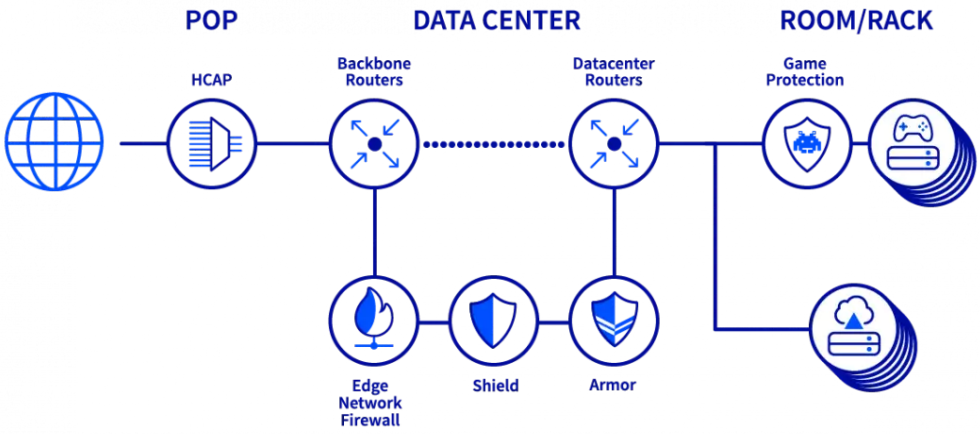


Figure 1.2: OVHcloud's network defense infrastructure

To protect a worldwide network with more than 20 Tbps connected to the Internet, OVHcloud has developed its own anti-DDoS protection system which successfully mitigates thousands of attacks a day. Figure 1.2 illustrates this defense infrastructure, beginning in POPs with *HCAP*, a first coarse-grained mitigation stage based on Field Programmable Gate Arrays (FPGAs). To complete the filtering, it then relies on detection algorithms (omitted in the figure) which redirects suspicious traffic to a mitigation center composed of three fine-grained mitigation elements. The last one, *Armor*, is of particular interest in the context of this thesis as it combines FPGAs and CPUs to deliver high-performance and

advanced mitigation mechanisms at hundreds of gigabits per seconds. This combination of heterogeneous technologies requires keeping hardware developments synchronized with agile software development flows and their continuous improvements.

Thesis Goals

In this context of pressing need for high-performance network devices, hardware accelerators such as FPGAs appear as energy-efficient alternatives to functionally equivalent software solutions. However, hardware developments struggle to reach the efficiency of their software counterparts and improved hardware development flows are required to deliver regular updates at a steady pace. Among the various available options, this thesis focuses on raising the level of abstraction of the design description, without compromising neither on performance nor on control over the hardware implementation. Representative of this approach, Hardware Construction Languages (HCLs), propose to describe hardware generators with modern software paradigms such as functional and object-oriented programming.

The overall goal of this thesis consists in evaluating the ability of HCLs to abstract hardware architectures, first natively and then as a platform for additional hardware-oriented abstractions. Experimenting with Chisel HCL [BVR+12], this thesis reviews the relevance of such abstractions to design highly flexible and re-usable high-performance network devices at OVHcloud.

Thesis Outline

Chapter 2 introduces in greater details the context and the approach of this thesis, and raises several precise issues we attempt to address with this manuscript.

Chapter 3 reviews prior research regarding hardware-oriented abstraction levels with a comparison to the evolution of abstraction levels in software. In particular, it focuses on Hardware Construction Languages (HCLs) as a first level of zero-cost hardware abstraction.

Chapters 4, 5 and 6 detail the contributions of this thesis. Considering both implementation and design perspectives, we first focus on the abstractions provided by HCLs to develop high-performance network functions. We then propose our own hardware-oriented abstraction to improve parameterization and re-usability of pipelined applications. We finally review the integration of these novel design methodologies within existing HDL hierarchies, which results in the introduction of two tools aiming at smoothening the cohabitation of both methodologies.

Chapter 7 combines the previous contributions in an industrial-scale experimentation. It follows the successive upgrades of a packet classifier from its original SystemVerilog hierarchy, to its re-integration as a pipeline-oriented hardware generator within the surrounding network devices.

Chapter 8 concludes this manuscript with a summary of contributions and details the perspectives of potential future works.

Chapter 2

Problem Statement

In this chapter, we present the issues of existing hardware design techniques with regard to agility and high-performance requirements of a constantly evolving network environment. Network devices are expected to flawlessly process very high throughput while providing configurability for both daily-basis operations and long-term evolutions. Designing devices matching both requirements is a complex challenge.

In order to apprehend the needs for more *agile* hardware design techniques, we first inspect the network ecosystem. Secondly, we detail network device design requirements which arise from the ecosystem, then we review existing hardware design solutions. Finally, we focus on hardware generation techniques and question their ability and means to provide the agility hardware designers are looking for.

Contents

2.1 Introduction

The problem tackled in this thesis lies at the confluence of two usually opposed considerations: performance and flexibility. On the one hand, high-speed networks set high standards of throughput and latency for their devices. On the other hand, their operation requires flexibility due to a constantly evolving usage, from both user and provider perspectives. Before digging into the association of these two considerations, this section first introduces the concept of *agility* and then provides applicative context about the topology of the Internet and the typology of the Internet traffic.

2.1.1 Agile Methodology Definitions

Following the publication of the *Manifesto for Agile Software Development* in 2001 [agi11], the terms *agile* and *agility* have become widely used adjectives to refer to many design and project management methodologies, not exclusively restricted to software [LWC+16]. These methods are grounded on the spirit of the manifesto, which emphasizes: *"individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan"*. In practice, agile approaches focus on **small and fast iterations**, each delivering an incomplete but working prototype. The adjectives *small* and *fast* are deliberately vague here as they heavily depend on the implementation of the *agile methodology* in a given context. Goals and principles set up for *front-end* software development might indeed largely differ from the one chosen for hardware tape-out.

In this thesis, the terms *agile* and *agility* are used to refer to this common base principle: *"small and fast iterations"*, without restricting to a specific implementation. As iterations are at the core of hardware development process, regardless of any project management considerations, we focus exclusively on agility within the hardware design flow itself. From design to implementation, to simulation, to synthesis and final testing, each step indeed requires fine-tuning of the architecture with back and forth to the drawing board. To accelerate these necessary iterations, responsiveness and flexibility of hardware descriptions, tools and design flows are key enablers.

The expression *"user-story"* can sometimes be used to refer to one design iteration, with the idea to demonstrate a user-impacting feature at the end of the fixed timeframe. Large and diverse requirements are split into such user-stories, quickly and regularly delivered as successive minimal functioning prototypes rather than as a single finished product down the line. Flexibility in the development process is required to allow such deliveries without impairing the overall consistency and slowing down the entire process. On the contrary, as these user-stories focus on implementing independent and well-identified features, they are expected to catch and address issues more efficiently. This leads to an overall improvement of the design quality and development time.

Practicing agile methodology requires an appropriate development environment to reach its goals, notably to avoid full redesign for each increment and potential regressions due to the frequent modifications. In particular, this thesis focuses on applying these agile principles to the implementation of high-speed networking devices. The following subsection provides an overview of the wide and heterogeneous ecosystem in which such devices are intended to be integrated: the Internet.

2.1.2 Overview of The Internet

The Internet is a giant computer network connecting just over half of the world population, with more than 3.8 billions users as of 2020 for an estimated global bandwidth of 700 terabits

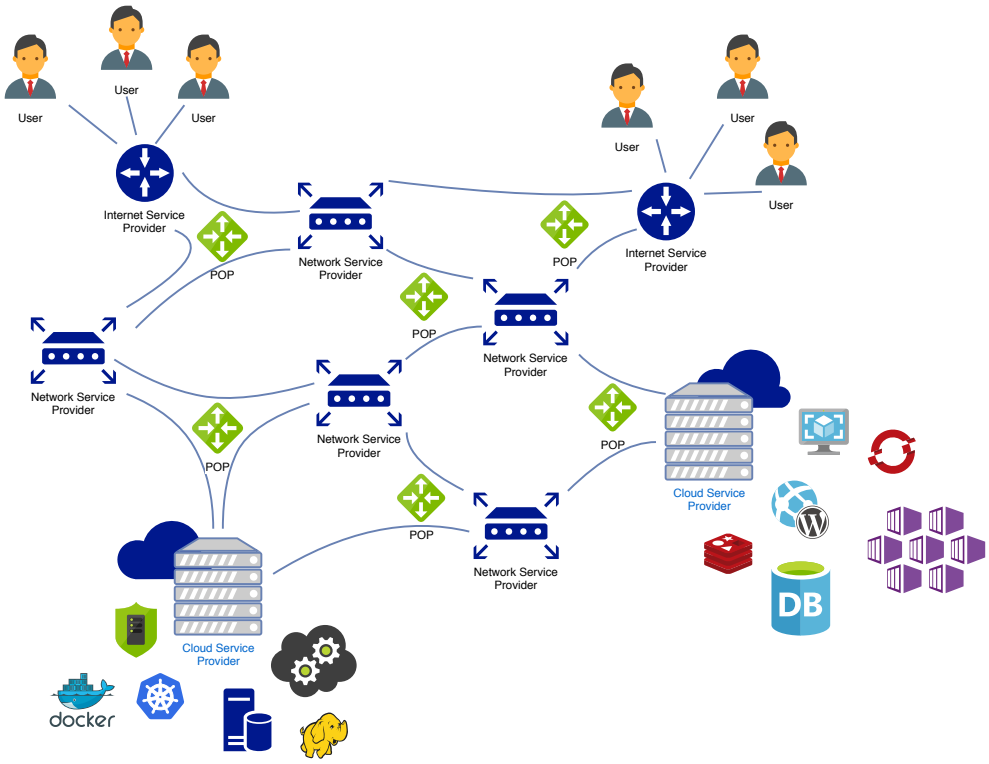


Figure 2.1: Internet topology

per second (Tbps) [BM+20]. At the opposite of a homogeneous and centralized system, it gathers thousands of heterogeneous subnetworks respectively providing access to parts of the world.

2.1.2.1 Topology

Such gigantic—and constantly rising—numbers are made possible thanks to the very nature of the Internet: *a multi-tiered, non-hierarchical network topology.* It means that there is no single point in the network centralizing and processing all these 700 Tbps. Instead, a multitude of actors operating infrastructures of various scale, contribute to this worldwide connectivity as Figure 2.1 illustrates. Users usually get an Internet access through a subscription with an *Internet Service Provider (ISP)*. ISPs are connected to multiple *Network Service Providers (NSPs)*, in order to reach the various places where services are located. While contents once used to be hosted on-premise by their respective authors, they are now largely centralized in datacenters managed by hosting companies. As these hosting companies diversified their activities during the last decade, offering many services such as compute, storage or network, they are now best known as *cloud service providers*. OVHcloud is one of such cloud service provider, the largest in Europe and worldwide competing with actors such as Amazon Web Service (AWS), Microsoft Azure and Google Cloud in America or Alibaba Cloud in Asia.

Each interconnection between two service providers requires a physical connection between their respective infrastructures which takes place in *Points-of-Presence* (POPs). In the decentralized topology of the Internet, these POPs are critical centralization points of the traffic, in charge of routing high traffic throughput, ranging from tens of gigabits per second to several terabits per second per interconnection. Due to their strategic value for the network stability, providers generally rely on redundant and oversized links to guarantee

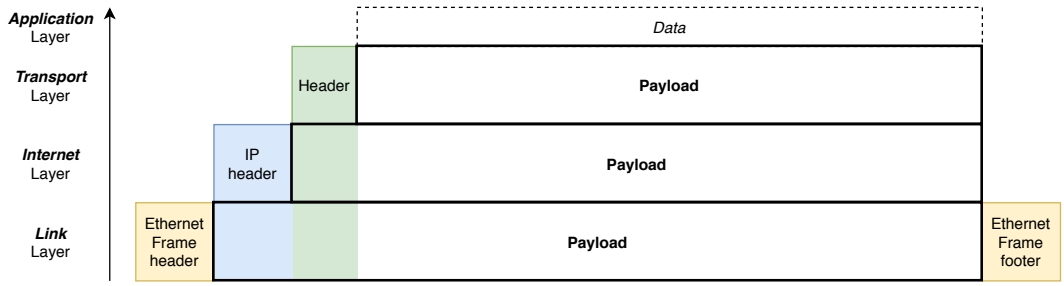


Figure 2.2: Typical IP packet structure in the *Internet protocol suite (TCP/IP)* model from *Link* layer to *Application* layer

their coverage of any given part of the Internet at any time.

2.1.2.2 Traffic typology

In all places of the network infrastructure, and in particular in POPs, measuring and forecasting the volume of traffic is crucial. This traffic consists in a raw flow of bits, grouped in independent *packets* which transit through the network from one *source* to a *destination*. Following the *Internet protocol suite (TCP/IP)* model, Figure 2.2 illustrates how raw bits are first interpreted as *Ethernet frames* whose payloads are in turn interpreted as *IP packets*, usually simply referred as *packets*. To measure throughput and understand the traffic typology, two key characteristics are considered.

Bitrate (bps) The most commonly used scale to report throughput figures is the amount of bits passing through a link or a device in a given unit of time. Called *bitrate* and measured in bits per second (bps), it offers a raw quantization of the traffic. In particular, at the *link level* this value takes into account all the bits required to transmit a packet over the communication medium, independently of any applicative intents. This physical bitrate is necessary and sufficient to dimension links capacity between nodes of the network. On the other hand, applications only consider the useful payloads and compute a bitrate which is more meaningful to the final user.

Packet rate (pps) This second scale reports the number of packet passing through a link or a device per second. As each packet requires its own set of processing steps, this scale is key to dimension the processing capacity of network devices.

Combining *packet rate* and *bitrate* offers further insight into the traffic typology. While bitrate is used to size the capacity of links and buffers, packet rate is directly linked to the computational power required to process the traffic. For example file storage services are associated with a flow of big packets meaning relatively high bitrate for relatively low packet rate. On the contrary logging platforms process a lot of individual events each consisting of a small packet with little data, leading to relatively high packet rate for low bitrate.

2.2 Network Application Design

Following the introduction of general considerations related to network integration context, this section delves into the design of network devices, from principles to detailed requirements.

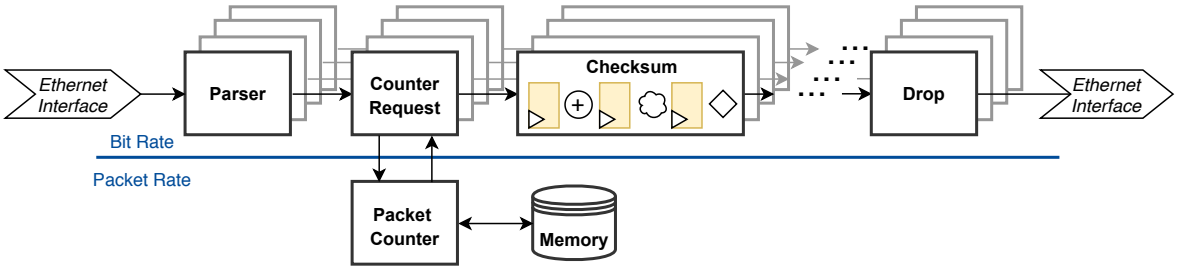


Figure 2.3: Network application representation

2.2.1 Design Principles

2.2.1.1 Definitions

Network Application The expression *network application* relates to the overall functionality offered by a set of network devices, regardless of the choices made for the actual implementation of this functionality. Network devices are *computers* in charge of processing network flows and can indeed be implemented with various hardware and software technologies. Moreover, some functionalities might either be implemented within a single device or shared across multiple devices, each approach providing its own advantages.

Network Flow A network flow is a generic term used to identify a communication between two nodes of the network. Depending on the network context and the application goals, these flows will be strictly defined as a set of fields related to the network protocol in use. For example, in the context of IP networks, a flow is often defined with its so-called *5-tuple*, consisting of:

- 1 field for transport protocol identifier[1] (most commonly used are TCP, UDP and ICMP),
- 2 fields for source and destination port of transport protocol (if any),
- 2 fields for source and destination IP addresses.

The 5-tuple is not fully exhaustive given the diversity of transport protocols, however it is a strong basis to identify and keep track of a connection from a user to a service.

Stateless The most basic network applications are stateless, meaning that they do not need to keep track of flows over a period of time. An example of such application would be a set of static rules in charge of allowing or denying access to some flows: e.g. accept only traffic towards a given range of public IPs on port number 443 (https), and deny all incoming packets that are not TCP.

Stateful More complex applications require keeping track of the flows over time. For example, to compute a throughput, a counter is associated to each flow in a *(key, value)* manner. Appropriate memories are then required and their limitations in terms of latency and throughput are key factors to dimension the implementation according to performance requirements.

[1]The transport protocol is the protocol corresponding to the transport layer of the *TCP/IP* model illustrated in Figure 2.2. The protocol identifier is contained into the header of this layer.

2.2.1.2 Generic Architecture

In a network application, some operations such as checksum computation are done at bitrate, while others such as packet filtering or state storage are done at packet rate. This leads—independently of any implementation choice—to a conceptually simple architecture represented in Figure 2.3.

Network packets are provided by the Ethernet interfaces as successive chunks sequentially processed in the application. A reliable and fast interconnection with these interfaces is crucial to meet performance requirements in terms of bitrate. Operations occurring at bit level are chained as a pipeline and a sufficient number of these pipelines are instantiated to reach the required bitrate. These pipelines then access shared operations executed at packet rate.

Packet rate and bitrate requirements for a network application hence have critical and distinct implications with respect to solution implementation. For stateful applications such as Figure 2.3 illustrates, the choice of a memory technology and its characteristics deeply impacts the worst-case achievable performance in terms of packet rate.

2.2.2 Performance Requirements

2.2.2.1 Throughput

At OVHcloud, the network entry points consist of a few dozens of POPs worldwide, providing a peering capacity of over 24 Tbps. Typical peering links capacity can range from tens of gigabits per second (Gbps) for minor actors to tens of terabits per second for large and tightly coupled actors. A typical value used to dimension network devices in POPs, is a peering capacity of 1 Tbps. However, while bitrate is sufficient to scale network links, packet rate is also a primary element to consider for network device design.

Minimal and maximal packet sizes are the key to draw a relationship between these two rates. Without exposing all the structural details of successive protocol layers, the main figure to consider is that the smallest *IP packet* requires 84 bytes[2] on the wire. In contrast, the biggest packets would usually take up to 1538 bytes or even 9038 bytes for so-called *jumbo frames*.

A continuous flow of theses *jumbo frames* at 1 Tbps would require an equivalent packet rate of:

$$\frac{1 \cdot 10^{12}}{9038 \cdot 8} = 13 \cdot 10^{6} \text{ pps} = 13 \text{ Mpps}$$

However the required packet rate quickly increases as soon as the packet size decreases. Considering the worst-case scenario of a continuous flow of the smallest possible packets, a device in charge of a 1 Tbps link should be able to process as many packets per second as:

$$\frac{1 \cdot 10^{12}}{84 \cdot 8} = 1.5 \cdot 10^{9} \text{ pps} = 1.5 \text{ Gpps}$$

This higher value shall be used, at least as reference, to define the packet rate requirement of any network device. In the context of attack mitigation, the worst case scenario for designer is indeed the best-case scenario for any ill-intentioned user and shall be considered accordingly. Based on statistics of actual traffic typology this requirement might be partially loosened, following a risk analysis which is beyond the scope of this thesis.

[2]Minimal size of an Ethernet Frame is 64 bytes (Link layer in Figure 2.2). Transmitting it on the physical layer—as a full Ethernet packet—requires a 8 bytes header and a 12 bytes inter-packet gap (IPG), adding up to 84 bytes [I+18].

2.2.2.2 Latency

Latency of a network device is the time for a given packet to fully pass through it, from the input interface to the output interface. Latency is a significant characteristic of network applications, as packets transit and are processed by many network devices before reaching their final destination. Several kinds of Internet uses deeply suffer from high or fluctuant latency, such as Voice over IP (VoIP) communications or video games.

Apart from the raw compute latency induced by the network application itself, a key component of the latency is the time spent between the application and the interface. This duration increases for each intermediate layer in the processing. A latency-optimal network application architecture, as presented in Figure 2.3 would connect as directly as possible the application to the interfaces without any third parties.

2.2.3 Agility Requirements

2.2.3.1 Daily Configurability

Immediate configurability of network applications is a key for fast incident response. Application must therefore be designed in a way to provide instantly adjustable parameters for network operation. Configurability can also be a service for users, with for example configurable firewall rules, manipulated through an Application Programming Interface (API) and potentially updated in a matter of seconds depending on the user's requirements.

However, this kind of instant configurability allows no time for application upgrade. This implies either careful planning of all use-cases at the time of application design, in order to expose all relevant parameters—which would be the opposite of agile methodology, or designing the application in such way that regular upgrades are part of its lifecycle, through *small and fast iterations*. Each of such iteration can then bring new parameterization options for the end-user, with fast response to emerging and unplanned needs. In this thesis we focus on this mid- and long-term upgrade requirements, further detailed in the next subsection.

2.2.3.2 Mid- and Long-term Design Upgrades

High-speed network devices, at the core of the Internet *backbone*, are expensive and expected to last for numerous years, when at the same time services and technologies evolve much faster.

A practical and long-running issue partially caused by outdated network devices is the slow adoption rate of IPv6 protocol, aiming at superseding IPv4 since 1995 but still very unevenly deployed in the world [CAZ+14]. Most new network devices now provide IPv6 support, but very few offer the same advanced features for both IPv4 and IPv6 traffic.

Alongside this actually very long-term evolution, many others, with smaller impact but occurring on a regular basis, punctuate a device lifecycle, such as evolution of the Internet uses. An example is the introduction of new technologies and platforms, such as video streaming and videoconferencing services, whose usage has been increasing for years [BM+20]. Another factor is the implementation of new higher level network protocols which impact the traffic shape at lower level. The rising utilization of *QUIC* protocol, an equivalent of HTTP/TLS over UDP [IT21, LRW+17], is a contemporary example. In addition, some network applications are built on top of various pieces of hardware, and their respective lifecycles impact the application evolution. Older hardware might indeed progressively be discarded, while the application must be adapted to support new references, from various vendors and slightly different specifications. Finally, the above-mentioned

incident response which triggers immediate actions sometimes also requires structural changes to the application, to sustainably mitigate newly detected threats.

Assuming that upgrading application design is an option, confidence in the design flow, from language expressiveness to automated validation toolchains, is decisive to achieve an efficient upgrade. The time spent to first, implement a design upgrade, and then validate it, is indeed a considerable factor of responsiveness. The fewer lines of code are impacted and the more localized they are, the faster the upgrade is implemented. From then on, an automated and exhaustive validation flow is a key enabler to efficiently deliver this upgrade. Confidence stands on the ability of a language and its associated toolchain to catch bugs and developer mistakes as early as possible in the development flow. Strongly typed languages with strict compilers and appropriate linters are valuable assets to grow confidence in a toolchain.

2.2.4 From Architecture to Efficient Implementation

Performance requirements of network devices are measured on various scales: bitrate, packet rate, latency and resilience to traffic surge. This thesis focuses on the network design requirements of core network devices, providing the highest standards for each of these requirements with respect to the power-consumption. Such devices are indeed aiming at handling continuously large amount of traffic, at network edge in POPs or within cloud provider network infrastructure. Considering the replication of devices across network infrastructure to process traffic at scale, performance per watt is a key criterion to compare their implementation.

2.2.4.1 Network Processing Units

In a network device context, solutions dedicated to packet processing are known as Network Processing Units (NPUs). One of their key characteristic is their close access to high-speed network interfaces in order to achieve high throughput and low latency.

The generic architecture presented in Figure 2.3, which serves as reference for the implementation of NPUs, is based on a pipeline of processing elements. In particular, usual NPU implementations can be divided in three main categories, depending on the configurability of both their pipeline and their processing elements [Sta15]:

***Fixed pipeline* NPUs** exhibit the best performance per watt but little configurability and no evolution ability,

***Configurable pipeline elements* NPUs** still constrain the network processing pipeline but support greater configurability and minor midterm evolutions, notably at the level of their processing elements,

***Fully-configurable pipeline* NPUs** are not restricted to any structure of network pipeline nor to a predefined subset of processing operators.

2.2.4.2 Available Implementation Targets

The diversity of configurability intents of NPU architectures leads to various corresponding implementation targets.

ASICs Application Specific Integrated Circuits (ASICs) are custom-made integrated circuits, designed and optimized for the application they serve. Typically chosen to implement fixed pipeline NPUs, they achieve the best performance per watt at the cost of limited

configurability and no long-term evolution ability. Their manufacturing cost as well as the time required from design to production makes them hardly suitable for small and fast design iterations.

CPUs General purpose Central Processing Units (CPUs) offer full flexibility, with fast short-term update capabilities and the ability to support long-term evolutions. In charge of both scheduling and processing, CPUs provide the implementation target for fully-configurable NPUs. In such case, the NPU is implemented as a software application, leveraging tools and framework such as DPDK [CAR14], eBPF with XDP [VCP+20] or VPP among many others [CDPA+18]. These implementations aim at shortening and accelerating the path between network interfaces and user-space where the application runs. However, this approach suffers from the physical distance: entire packets have to transit up and down the software network stack. As a result, this stack becomes a throughput bottleneck which craves for hardware acceleration.

Mixed Approach To benefit from both ASIC performance and CPUs flexibility, some NPUs combine both to offer configurability ranging from *Configurable pipeline elements* NPUs to *Fully-configurable pipeline* NPUs. A big agility advantage of these platform is the provided toolchain used to program them: both daily update and long term update can be compiled from the provided language(s) to the device in a matter of minutes. They are hence the most suitable NPUs to match our requirements, however, they come with two pitfalls. First, the extended configurability comes at the cost of lower performance per watt. Second, the long-term evolutions remain strictly restricted to the underlying architecture, including but not restricted to bus sizes and coarse-grain operators available. With respect to the previous IPv6 deployment example, these restrictions would lead to accept a considerable loss of throughput in order to enable IPv6 support on an earlier device. Indeed, the support of IPv6 requires much larger metadata processing: not less than 256 bits instead of 64 bits for a source-destination IPs tuple. Assuming an existing metadata bus of 64 bits, IPv6 packets require 3 additional cycles to be processed on such device.

FPGAs Field Programmable Gate Arrays (FPGAs) are reconfigurable devices composed of numerous programmable logic blocks and interconnects. Arbitrary large FPGAs are able to implement virtually any digital circuit, at the cost of limited running frequencies due to their fixed layout of configurable resources. They exhibit an interesting trade-off between performance per watt and lifetime management, which makes them well-suited to build network processing systems. In particular, they are able to implement efficient *fixed pipeline* NPUs while retaining the ability to support a complete architecture upgrade for an updated version of the network application. Compared to ASICs, they do not require the high investments associated with tape-out and are hence able to address niche markets with cost-effective chips.

2.2.4.3 Conclusion

While providing the best configurability, software solutions based on general purpose CPUs are falling behind in terms of performance per watt compared to specialized hardware accelerators [NSS+16, CS12]. As a result, dedicated hardware solutions are generally preferred, at least for the wire-rate processing, for development of high-speed network processing units. As ASICs can hardly be described as agile due to their inherent manufacturing cost, FPGAs appear as the best suited implementation targets given our requirements. The following section therefore focus on way to match both agility and performance requirements in the context of hardware design.

2.3 Towards Agile and Efficient Hardware Design

2.3.1 Limitations of Leading Hardware Design Methodologies

First steps of digital circuit design flows are identical for ASIC and FPGA targets. They both rely on a description of the circuit at Register Transfer Level (RTL). This RTL description can be manually crafted or produced by various tools from a *higher-level input*. It can then be used interchangeably for simulation, FPGA synthesis or ASIC synthesis. In this thesis we focus on the expressiveness and reusability of hardware descriptions at their various abstraction levels. Therefore, without loss of generality, in our experiments we specifically report result of FPGA synthesis due to the previously discussed network context and requirements.

2.3.1.1 Context

Two languages, *VHDL* and *(System)Verilog*, have emerged in the 80s to model digital circuits at the Register Transfer Level. Originally designed for simulation rather than synthesis, they intend to offer behavioral models of circuits, which are both Cycle-Accurate and Bit-Accurate (CABA). This level of detail offers the highest fidelity with the actual digital hardware, and was found precise-enough to be directly used as actual hardware description and synthesized into hardware netlist.

This high precision level makes these languages quite verbose, and their roots in behavioral simulation leads to two major issues for hardware description. First, only a restricted subset of the language is supported by synthesis tools, with significant differences among vendors. Second, the event-driven paradigm, particularly fitted to behavioral modeling, leaves the inference of events into actual hardware by the tools, which in practice requires sticking to predefined syntax and code snippets in order to obtain consistent results.

2.3.1.2 HDL Agility Struggle through simple User-stories

In this subsection, we intend to demonstrate that classical HDL such as VHDL and Verilog are hardly compatible with agile methodology. In order to highlight their low agility, we review the code modification required by simple user-stories at the level of a basic addition module. Without loss of generality, the following examples are presented as Verilog code.

Iteration #0 A first version of the module is created, in its simplest expression, as follows:

```
1  module add(
2      input [5:0] a, input [5:0] b, output [6:0] r
3    );
4    assign r = a + b;
5  endmodule
```

This module *add* is computing the arithmetic *operation* $r = a + b$. The associated architecture is fully combinational and does not involve any protocol signaling.

Iteration #1 After synthesis of this first module, assuming integration in a larger design, the timing report indicates a long combinational path linked to this module with regard to the rest of the design. For the second design iteration a register stage is hence added to store the result of the addition directly in the module as follows:

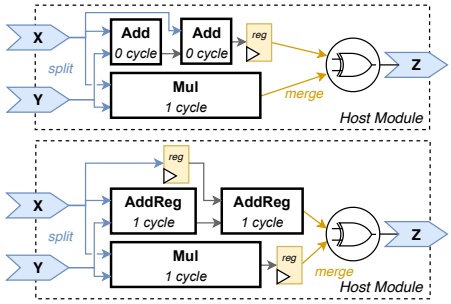


Figure 2.4: Potential integration issue with submodule latency evolution

```
module addReg(
    input clock,
    input [5:0] a, input [5:0] b, output [6:0] r
  );
  logic [6:0] res;
  always @(posedge clock) begin
      res <= a + b;
  end
  assign r = res;
endmodule
```

The presence of this additional register is to be considered carefully when integrating this module in larger designs. While breaking the combinational path helps the place & route tool, the latency of the module in clock cycles is increased, with no automated propagation of this information to the rest of the design. Figure 2.4 illustrates how the difference of latency between `add` and `addReg` modules must be compensated according to surrounding context in the upper hierarchy. However, such simple example is not representative of the complexity of a real module, in which moving the register in the upper hierarchy might simply not be feasible.

Iteration #2 While the surrounding design evolves with various needs, a third iteration is now required to add a ready/valid signaling to the *data* signal.

```
module addRegDecoupled(
    input  clock,
    input  [5:0] a, output a_ready, input  a_valid,
    input  [5:0] b, output b_ready, input  b_valid,
    output [6:0] r, input  r_ready, output r_valid
  );
  logic [6:0] res;
  logic res_valid = '0;
  always @(posedge clock) begin
    if (r_ready) begin
      res       <= a + b;
      res_valid <= a_valid & b_valid;
    end
  end
  assign a_ready = r_ready;
  assign b_ready = r_ready;
  assign r_valid = res_valid;
  assign r = res;
endmodule
```

In this new description the three *data* signals *a*, *b* and *r* are respectively associated with two protocol signals: *ready* and *valid*. This widespread pattern for signaling *data/ready/valid* is also referred as a *decoupled* interface. The introduction of this signaling protocol drastically increases the number of lines and reduces the readability of the user intent, now only counting for a single line of code over 19 of them, while at the first iteration it was also a single line of code but over only 5 in total.

Iteration #3 Later on, the target FPGA for the design is changed for a more recent technology, allowing longer combinational paths. It is now counter-productive to add a register stage for every instance of *add* in the design. For this fourth iteration, the register stage is therefore removed from the module.

```
1  module addDecoupled(
2      input  [5:0] a, output a_ready, input  a_valid,
3      input  [5:0] b, output b_ready, input  b_valid,
4      output [6:0] r, input  r_ready, output r_valid
5    );
6    assign r = a + b;
7    assign r_valid = a_valid & b_valid;
8    assign a_ready = r_ready;
9    assign b_ready = r_ready;
10 endmodule
```

This leads to substantial modification of the module and as above-mentioned, special care is once again required to carefully adapt the surrounding design to keep all signals synchronized and meaningful.

Conclusion These four user-stories highlight the major impact of basic implementation-related design upgrade with traditional hardware description languages. In the current example, this impact comes from two main factors. First, protocol signaling is very verbose, it dilutes the essence of the user intent in a lot of additional protocol management, decreasing clarity of code. As a consequence, swapping one protocol for another means an almost complete rewrite of the module which drastically limits reusability and increases validation cost. Second, as Figure 2.4 illustrates, the designer is in charge of mentally inferring and manually propagating latency in terms of register stages from the innermost modules to the top module. This time-consuming and error-prone task further increases the cost of design upgrades, both in terms of development and validation. Developed a few decades ago assuming fixed requirements and specifications, traditional HDLs such as Verilog do not prove suitable for the *small and fast iterations* of agile methodology.

2.3.1.3 Performance Limitations of High-level Design Methods

To overcome the limitations of traditional hardware description language, previous research focused on increasing the abstraction level of hardware development [BRS13, SAW+10]. A popular approach consists in expressing functionality instead of describing hardware. This includes High-Level Synthesis (HLS), where software languages are compiled into hardware architectures, and Domain-Specific Languages (DSL), where domain-dependent primitives are provided to describe the design.

High-Level Synthesis High-Level Synthesis is an active research area, in quest of efficient and generic algorithms to infer hardware architecture from software algorithms [NSP+16]. One of the goal of HLS approach is to open hardware development—in particular FPGA

as acceleration platform—to software developers with little hardware development experience, leveraging widespread languages such as C or C++. A second objective is the abstraction of the target: allowing a single piece of code to run indifferently on various target (CPUs, GPUs or FPGAs), even considering real-time context-switching between these targets depending on the needs [BMR16]. However, these promising features come at the expense of the development of a complex compilation toolchain. Inferring a functionally-equivalent hardware architecture from a given algorithm is indeed a first challenge in itself but finding an appropriate compromise that fits design constraints among the numerous possible hardware architecture is even harder. To tackle this issue, hints can be provided to the compiler as code annotations. However, this requires to manually provide an architecture for many pieces of code, hence defeating the original purpose. This also considerably slows down the design process, ultimately making it harder to upgrade as annotation might require to be manually adapted at each functional iteration. As a result, high-level synthesis is struggling to bring software agility to hardware design by reusing the exact same languages. In particular this applies to control-oriented design used in network applications, where architecture annotations are required to match throughput performance [NSP+16, MMGC20, GRDT+16]. Due to these annotations, the portability of HLS hardware descriptions between vendors is a complex challenge, which further reduces the actual reusability of such descriptions [XCC21]. While efficiently closing the gap between software algorithms and hardware implementations in various contexts, HLS is not suited to match both performance and agility requirements of high-speed networking applications.

Application-Specific DSLs *Domain-Specific Languages* regroup all languages designed for a restricted set of task or tied to a particular application domain. With respect to the strong network context, we focus here on network application DSLs which are promising for defining a network device at application level. FlowBlaze [SRC+19] is a DSL that provides a way to represent stateful applications. Another popular language is P4 [BDG+14], which is designed to program network pipelines. It relies on an architecture representation which defines the capabilities of the underlying processing system, and on an application representation based on required fields and operations applicable on packets. Toolchains implementing P4 offer real improvements over baseline HDL implementation. They automate the tedious process of protocol implementation, thus limiting the usual errors in bit manipulation, and speeding up integration of new protocols. They also provide an implementation agnostic view of the application, which eases discussion with non-hardware specialists and helps to focus on the functionality instead of implementation details. However, while P4 increases agility and expressiveness in this context, it is not sufficient. It relies on vendor IPs for function implementation, or on HLS when none exists. As HLS is not really adapted for these control-oriented applications, this leaves the implementation of missing functions unresolved. The tight performance constraints then lead to the need for custom implementations of critical functions. This means that agile development cannot be achieved only through the use of a network-related DSL.

Conclusion Providing advanced ways of representing network devices at application level, top-down design methods based on compilation of software languages still struggle to demonstrate clear agility gain when performance guarantees are set as key requirement. High-level hardware design methodologies presented so far fail to improve agility without compromising on architecture control and application performance. To that extend, these approaches are not further considered in this thesis.

2.3.2 Promises of Advanced Hardware Construction Languages

Hardware Construction Languages—or Frameworks (HCFs)—regroup all languages and toolchains that aim at providing generation constructs to produce pieces of digital hardware.

To that extent, thanks to their slight evolution since their creation, traditional HDLs such a (System)Verilog or VHDL can somehow be included in this category as they do provide some basic generation capabilities. However, as we previously demonstrated, their capabilities are too restrictive to match agile methodology requirements.

In order to increase abstraction level and provide advanced generation constructs without losing any control on the resulting architecture, three main approaches have been explored. The first consists in processing some pieces of existing hardware descriptions with an external tool to adapt them to various contexts. The second is about introducing new languages, supporting similar paradigms as traditional HDLs as well as new ones and enhanced constructs. The third one is about meta-programming: embedding hardware description languages into existing High-Level Languages (HLLs) which are providing the constructs to generate and further manipulate the circuits. Differences of these approaches will be further reviewed in the next chapter, as we now focus on the principles and objectives they all share.

2.3.2.1 Principles

Generation HCLs are sometimes known as *Hardware Generation Languages (HGLs)* [Max11] to highlight one of their common principle: provide advanced generation capabilities rather than simply describe hardware. Generation capabilities consist of constructs able to instantiate, duplicate and connect programmatically pieces of circuit.

Parameterization Another shared principle is to focus on providing extended parameterization where traditional HDLs are restricted to fixed patterns and limited types. Parameterization is no longer to be limited to basic module parameters and HCLs offer to propagate them in new ways, such as automated negotiation between communicating parts of the design [CTL17, AAB+16].

Additional paradigms While traditional HDLs hardly evolved since their first introduction, software languages, originally very close to machine language and its imperative programming paradigm have since benefited from many innovations. These innovations include among others object-oriented programming and functional programming paradigms. HCLs attempt to make these paradigms available to hardware designers as relevant tool in a hardware generation context [BKK+10].

Elaboration The previous principles come with the need to close the gap between high-level parameterized generators, and existing toolchains for hardware simulation and synthesis. While some works suggest introducing and standardizing an Intermediate Representation (IR) common to all the hardware design tools [SKGB20], the original principle of HCLs consist in leveraging the existing toolchains without requiring them to change. The uneven to nonexistent support among vendors of the features progressively introduced in traditional HDLs has indeed been a considerable issue for hardware designer for years. In order to fit in the existing and quite passive hardware tool ecosystem, HCLs come with their own toolchain to elaborate the high-level generator into low-level hardware description in the form of traditional HDL code.

2.3.2.2 Objectives

Reusability A first objective of HCLs is to promote code-reuse rather than code duplication. Due to their limited parameterization capabilities, code factorization is often hard to achieve with standard HDLs, which often then leads to code duplication rather than code refactoring. This prevents the concurrent versions of an almost identical design to benefit from the improvements made to one of them, whereas HCLs aim at supporting small and fast iterations on the same pieces of code.

Expressiveness A second objective of HCLs consists in easing the design of complex pieces of hardware based on expressive description patterns. Such patterns aim at improving code readability and reducing boilerplate code to describe complex circuits.

Conciseness A transversal objective of HCLs is code conciseness, which helps with both reusability—shorter code is faster to update and integrate—and expressiveness—as to achieve conciseness the language must provide powerful constructs able to efficiently describe circuits.

2.4 Conclusion

In a context of network application setting strong requirements on both performance and agility, traditional HDLs as well as top-down high-level abstractions do not prove sufficient to produce efficient hardware at a steady pace.

While providing base hardware primitives close to HDL ones, Hardware Construction Languages follow a bottom-up approach to increase circuit abstraction. In particular, they support extended constructs to generate hardware, and promise more expressiveness and better code-reusability thanks to these new capabilities. Based on solid principles, HCLs notably claim to efficiently improve the hardware design development flow compared to traditional HDLs, which has been partially verified on some large projects [AAB+16]. However, HCLs have not yet fully demonstrated their claims, which raise several questions we aim at answering in this thesis:

- How far can the abstraction level provided by HCLs instill agility into hardware design flow while preserving fine-grained control over implementation?
- How can this abstraction level significantly remodel the design of hardware circuits from the earliest stages?
- What higher-level hardware design abstractions and paradigms can be relevant to improve agility of hardware design?
- How can HCLs provide the foundations to introduce and implement these additional abstraction levels?
- How to integrate HCLs generation-based flow into large existing codebases and their established hardware development flow?

Chapter 3

State of the Art

In this chapter we first establish the position of hardware construction languages (HCLs) in the hardware design ecosystem among the various existing abstraction levels. In particular, we compare the progressive raise in abstraction levels in hardware design to similar evolutions in software-engineering.

As the layering of successive abstraction levels has indeed unlocked new design opportunities in software, in a second part we study candidate high-level design paradigms aiming at remodeling hardware design methodologies. This study focuses notably on pipeline programming and paradigms leaning towards separation of concerns.

Finally, we review how, from shared principles, the diversity of HCLs implementations provides a suitable platform to develop and extend these high-level design paradigms towards increased flexibility and agile development flows.

Contents

3.1 Raising the Abstraction Level

Recipient of the Turing Award for her work on data abstraction and *abstract data types*, Barbara Liskov[1] defines the concept of abstraction as follows: *"An abstraction is a many-to-one map. It 'abstracts' from 'irrelevant' details, describing only those details that are relevant to the problem at hand"* [LG+86]. In this section we review which details are relevant to preserve when raising the abstraction level, first in a software engineering context for reference, and then comparatively in a hardware design context.

3.1.1 Abstraction Levels in Software Engineering

3.1.1.1 Iterative Physical Constraints Abstraction

Since the early days of computer engineering, *von Neumann* architecture has been widely adopted to build soft-programmable computers. It remains today the standard architecture of general purpose computers. At the core of this architecture lies a processor or central processing unit with a control unit and an arithmetic/logic unit [VN93]. This processor is associated with a memory containing both data and instructions. From this point onwards, computers were ready for software programming. However, processor specifications are unique to each implementation, from the number of registers and their widths to the available arithmetic and logic operations to architecture of the control unit itself. These fine architectural details had to be fully understood by the developer to design a software, that in the end, could only execute on a very specific machine.

To ease the development and improve the reusability of software, additional abstraction levels are required. Instruction Set Architectures (ISA) provide an abstract model of the computer which, without constraining all the details of processor implementations, requires them to execute the same set of instructions. Based on ISAs, assembly languages enabled the design of software in a human-readable style. Assembler programs support the use of additional abstractions such as macros, enabling them to convert some generic patterns into appropriate machine code depending on the actual ISA implementation.

Figure 3.1 illustrates on bottom left these first abstraction levels, progressively alleviating the physical constraints from the processor and memory to human-readable languages, easier to develop and maintain. This trend continues with the development of compilers and languages such as FORTRAN, ALGOL, COBOL and C, steadily providing further abstractions based on algorithmic concepts such as variables, functions or loops.

Higher-level languages introduce additional algorithmic concepts and new programming paradigms such as functional, object-oriented or concurrent programming. These paradigms intend to increase abstraction by providing ways to model programs and focusing on algorithm expressiveness. Runtime and virtual machines provide further physical abstractions of the underlying hardware. For example languages such as Java—virtual machine—and Python—interpreted—entirely hide the concept of memory allocation and management thanks to a garbage collector. This mechanism greatly eases software development at the cost of runtime overhead. While fine-grained control of performance is definitely a *relevant piece of information* and first-class concern of programming language such as C or Rust, languages such as Python or Java made the choice to forgo this concern in the development environment they provide. To still provide a respectable performance to their users, substan-

[1] In 1968, Barbara Liskov became one of the first women in the United States to be awarded a computer science PhD. She earned the Turing Award in 2009, computer science's equivalent of the Nobel Prize, "for contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing." She currently heads MIT's Programming Methodology Group in the Computer Science and Artificial Intelligence Laboratory. [ACM09]

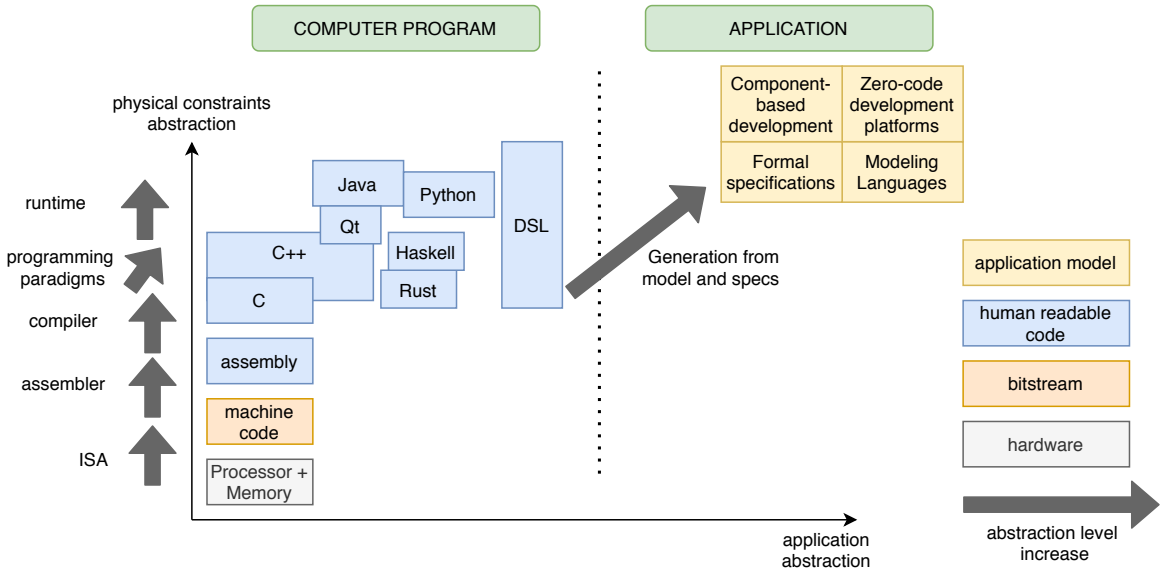


Figure 3.1: Partial visualization of abstraction levels in software engineering focusing on selected programming languages and techniques

tial efforts regarding this matter have rather been poured into the development of efficient interpreters and virtual machines.

At this stage, the abstraction is entirely oriented towards building software more efficiently, with a focus on automation of low-level tasks such as choices of registers to use or fine-grained instruction scheduling.

3.1.1.2 Towards Application Abstraction

Some programming languages shift from the sole abstraction of physical constraints towards abstraction of the application itself. Expressive code intends not only to model the computer program but also the applicative context it serves. Relevant information in such context includes domain-specific information and formalism. A lot of initiatives follow this trend, ranging from graphical libraries such as Qt to Domain-Specific Languages (DSL) [DKS+12, Vis07], intentionally forgetting the general purpose context to focus on the tailored expression of precise applicative matters.

All the above-mentioned languages and abstractions take their ground into the context of computer program development, with more or less abstraction of the execution target. They are primarily intended for a developer to build some piece of software to match some requirements to solve a given problem.

3.1.1.3 Application Models

On the other hand, the right side of Figure 3.1 presents techniques which focus on modelling an application, without integrating the constraints of the target in their representations. This way of abstracting exhibits a strong discontinuity with the previous approach. In this context, relevant information only regards the application requirements and design, and disregards any implementation matter in the model. One of the objectives of such methodologies is to provide a solution to specify applications without software development skills, enabling domain-specialists to design applications tailored to their needs. While previous abstractions remained in the software paradigm, hence leveraging common bases, application models stand apart. They might be used only to represent the application

specification, and might not be sufficient for complete code generation.

The most advanced generative approach consists in formally-proven code generation from formal specifications. It does not focus on improving the easiness to build an application out of general or domain-specific requirements, but aims at generating safe code for life-critical systems such as avionic software development [BBF+00].

Component-based development consists in assembling in a high-level representation some components, picked out of a library of precisely specified and already implemented components. The generation process is then limited to known patterns for interfacing and instantiating the components. Zero-code and no-code approaches are typical examples of component-based development [KMF01, CL+17].

3.1.1.4 Conclusion

Software-engineering began with manual programming of custom computer architectures and progressively raised the abstraction level to focus, first, on algorithms, and then, on applicative needs. Highest levels of abstraction enable designers to accurately model applications while intentionally hiding implementation details. Final code might then be automatically generated or left to developers, leveraging the advanced software-engineering techniques available in high-level languages to implement the application.

The history of hardware engineering follows somehow similar evolutions in the definition and usage of various abstraction levels. We now explore these hardware abstraction levels and detail how they compare to software abstraction levels.

3.1.2 Abstraction Levels in Hardware Design

This thesis focuses on hardware development agility in the context of reprogrammable hardware targets. Results and conclusions have been drawn exclusively from FPGA experimentations, therefore we mainly focus here on abstractions levels above FPGA architectures.

3.1.2.1 Base Circuit Abstractions on FPGA

The base abstraction level of digital circuit design is the transistor, an electronic component serving as a controlled switch. Several transistors can be assembled together to create logic gates, providing the abstraction level of Boolean algebra as a collection of logic operators, such as *NAND* and *NOR*. Assembly of logic gates in more advanced circuit provide various functionalities such as memorization with flip-flops (FFs) and logic operations with Look-Up Tables (LUTs).

FPGAs provide a collection of such functionalities as base reprogrammable elements: FPGA resources. They include registers, various memory styles, arithmetic operators and logic operators. The count of each resource and their fine-grained placement with connection busses on the chip, are fixed by the FPGA architecture and define absolute capacity limitation for each FPGA. Exact mapping of elements and their routing must be explicitly specified to implement a circuit. From this point onwards, this limited and bound set of resources will be considered the base abstraction for the FPGA developer.

Similarly to machine code, FPGAs are programmed with a raw binary file, called *bitstream* whose format depends on each device architecture. Most commercial FPGA architectures and their associated bitstreams are closed-source and proprietary as they are considered very valuable intellectual property. While a bitstream abstraction level per FPGA family exists for constructors, FPGA developers have very limited access to this level.

The netlist abstraction makes the physical layout of the circuit irrelevant, and is somehow equivalent to assembly language of software. This abstraction level enable FPGA developers

to instantiate base logic elements and to define their connections while the painful task of mapping the circuit to an FPGA architecture is left to a *place and route* tool.

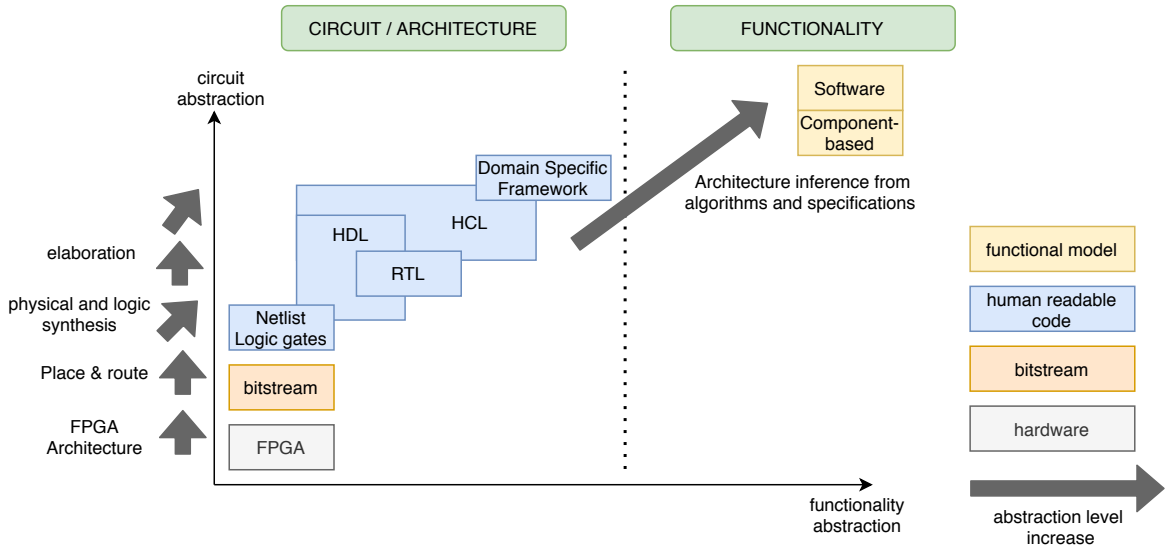


Figure 3.2: Visualization of abstraction levels in hardware design for FPGAs

3.1.2.2 Architectural Abstractions

To abstract circuits as hardware architectures in a generic way, hardware description languages (HDLs) introduce a basic structural abstraction based on modules. A module is a simple container with Input and Output ports (IOs), which may contain arbitrary complex logic and instances of other modules, then referred as *submodules*. In particular, it may model any circuit, from a simple transistor to a complex processor. To that extend, HDLs are able to describe circuits at various levels of abstraction, from the netlist composed of basic primitives to complex user-defined hierarchies.

To abstract the most basic of these primitives such as adders, multiplexers, and registers, the Register Transfer Level (RTL) provide generic operators and algorithmic concepts, such as conditional statements and basic loops. Figure 3.2 illustrates the raise in the abstraction level in terms of both circuit and application abstractions, thanks to synthesis tools. First, logic synthesis transforms algorithmic and arithmetic statements, closer to applicative expression, into basic logic operators. Then, physical synthesis is in charge of mapping these generic elements to the primitives available on a given target. HDLs also introduce hierarchical structure of circuits with modules, allowing modular programming and improving code reusability on a module basis. Modules, also referred as IP for intellectual property, are used as base components of several higher level abstractions such as system-level design.

Register Transfer Level (RTL) focuses mainly on the description of synchronous circuits as combinational operations in between clocked registers. Within HDLs such as Verilog or VHDL, this abstraction is described as behavioral and event-driven patterns to model registers as processes, triggered by signal value changes. This description style was originally intended for simulation purposes but has since become the most commonly used abstraction level.

Elaboration provides a structural abstraction level by offering generation capabilities and parameterization of the circuit at a module granularity. In particular, it can be applied to create parameterized RTL modules, which greatly improves the reusability of architectures. This approach has been partially integrated within traditional HDLs with basic generation

and parameterization constructs.

Hardware Construction Languages (HCLs) are also based on the RTL abstraction to describe the base hardware primitives, but they raise elaboration capabilities to a whole new level with highly parameterized hardware generators.

3.1.2.3 Software and Applicative Models

On the other side of the spectrum, the right side of Figure 3.2 presents solutions which fully abstract the circuit and architectural considerations from the design intent.

High-level synthesis aims at inferring circuit architectures from software-style algorithm expressions [NSP+16]. This abstraction level places the circuit functionality as the most relevant information and dismiss all implementation details from the description. This approach gives access to FPGAs as accelerators to software developers and decreases the development time for hardware designers thanks to more concise code. However, with implementation and architectural details removed from the description, control over these elements—crucial to match performance requirements—becomes quite cumbersome if not impossible.

Another approach consists in component-based descriptions, providing a high-level representation of the expected functionalities of a system [GTSB08]. It leverages base blocks, implemented with any aforementioned methods, and provides an interface, sometimes graphical [Eri17], to connect and parameterize them. Similarly to component-based software developments, this technique limits the hardware generation to known patterns at the interfaces of existing blocks [KVDWDK+08, RKH+20].

3.1.2.4 Transversal Domain-Specific Abstractions

Similarly to software DSLs, some hardware abstractions are tailored for specific applicative domains and their needs [RMMV10, MML13, SBL+14]. Theses additional application abstractions can be added on top of any other level of hardware abstractions, including HCL with Domain-Specific Frameworks [PKB+16] and HLS [GLN+14]. Specific applicative contexts range from signal processing [TŽ17, CCB+08], to machine learning [GRDT+16], to networking [AB09, LDC12, BDG+14, LTL+16, SRC+19].

3.1.3 Conclusion

While software abstractions have raised towards algorithm expression first, and then application models, circuit abstractions have focused on architecture first, and then functionalities. However, in both domains, a strong discontinuity appears between the two abstractions axes illustrated in Figures 3.1 and 3.2. This gap is explained in both cases by opposing conceptions of abstraction, and associated choices of relevant information to preserve. The choice of forgetting implementation details in any high-level abstractions opens a huge design space which reveals hard to fully explore when mapping the model to an actual implementation. This leads in both domains to a reduction of the operational scope of the models or to a limitation of performance as finding the optimal solution to any general purpose problem is considered a NP-hard problem.

Software developers strongly interested in performance and still aiming at improving language expressiveness have overcome these limitations with zero-runtime-cost abstractions provided for example with the Rust programming language or C++ templates. In this approach, the cost of abstraction is assumed by the compiler, resulting in longer compilation time but guaranteeing no impact on the runtime. Similarly, advanced elaboration capabilities of HCLs are aiming at providing a zero-cost abstraction in terms of FPGA resource usage

and resulting performances, the abstraction cost being paid by slightly longer elaboration time. Latest development effort regarding the elaboration stack are following closely the techniques offered by software compilation stacks, strengthening the comparison [L+21, LAB+21].

Based on the current panorama of abstraction levels, highlighting the relevance of zero-cost architectural abstraction to bring agility into hardware design, the next section explores in further details how another architectural abstraction level could be introduced with higher level paradigms.

3.2 High-level Hardware Design Paradigms

In this section we attempt to provide a comprehensive overview of high-level hardware design paradigms. Contributions in this domain are quite scattered and do not follow any linear evolution, often emerging from domain-specific applicative needs. To our knowledge, no previous work has provided a classification of such paradigms in hardware. We focus on paradigms dimmed relevant in the context of high-speed packet processing applications and their discrete flow of independent packets interspersed with idle periods.

3.2.1 Pipelining Paradigm

Digital circuits are inherently parallel as each part of the circuit is able to process data independently at the same time. Describing an algorithm as a digital circuit opens a wide space of possible implementations. The pipelining approach implements it as a chain of computational units, instantiated and connected in the same order as the original algorithm. Computational unit are connected through buffers, allowing independent operations. Each unit in between two buffers is called a pipeline stage. Execution of the sequential algorithm then consists in successive executions of the pipeline stages. For one execution of the algorithm, each pipeline stage is only used once. As soon as the input pipeline stage is empty, another execution flow can begin which follows the previous one all along the pipeline.

Exploiting chained operation parallelism, the pipeline paradigm maximizes throughput of streaming applications at the cost of dedicated resources for every stage. It is hence widely used both for software development [GTA06, LLS+15, KST13] and to leverage the inherent parallelism of digital circuits, from their earliest expressions [Cot65, HF72].

3.2.1.1 Pipeline Patterns

In traditional HDLs, description of a pipeline follows two main recognizable code patterns. The first identifiable pattern is the behavioral process associated to the update of a given signal, *de facto* describing a register. The second pattern consists of flow control signals, providing a communication protocol between successive pipeline stages, and might be omitted. Such signals are used to indicate when some data is available at the output of current stage, and when current stage can accept some incoming data. A common implementation consists of a *valid* signal to indicate upstream data availability and a *ready* signal to indicate downstream ability to process new data. A successful handshake occurs when both signals are high at a same rising clock edge. Apart from this *ready-valid* protocol signaling, several well-known protocols can be implemented around the same pipeline stage functionality, such as Credit-Based, or Carloni [AB18]. The following Verilog excerpt exhibits a pipeline stage with a single data signal and ready-valid control signaling.

```
1  // stage1 data and control signal declaration
2  reg  [DATA_WIDTH-1:0] stage1_data;
3  reg                   stage1_valid;
4  wire                  stage1_ready;
5
6  // stage1 synchronous computational unit and buffering as register
7  always @(posedge clock) begin
8    if (stage1_ready) begin
9      stage1_data  <= in_data;
10     stage1_valid <= in_valid;
11   end
12 end
13 // stage1 combinational operations: here control back-propagation
14 assign in_ready = stage1_ready;
```

Sticking to such patterns makes the pipeline stages appear very clearly in the architecture description. However, the pipeline only emerge to the trained eyes of the designer and has no concrete definition within the language. As is, pipelines and pipeline stages do not provide any reflexivity[2]over their signals, operations and structure, restricting their existence to a mental abstraction for the developer. In particular, this prevents any parameterization and instrumentation of this pipeline pattern.

3.2.1.2 Pipeline Stage Abstractions

To raise the pipeline stage abstraction level above a subjective pattern, a first step consists in providing a dedicated syntax to describe pipeline stages as first class citizen in the hardware description. TL-Verilog [Hoo17] provides such a stage-based description, enabling easier identification and iteration over pipeline stages, with the ability to implement a stage as either register or wire. However, this approach is based on a simple register pipeline view, and do not tackle the issue of protocol signaling nor the construction of interconnected pipelines.

To bring additional consistency to pipelines as formalized abstraction elements, an approach consists in leveraging a double-sided API. One side defines pipeline stages as base elements, while their instantiation and structure is specified with its counterpart. In *sc_pipes* SystemC library [HP14, Har09], pipeline stages are modeled as independent processes, which are then connected together with a set of operators. This formalization, originating from software modeling of pipelines, is based on a strong split between the stages—providing base functionalities—and their organization—providing both the architecture and the overall functionality. In practice the framework encapsulates software functions as pipeline stages, and then focuses exclusively on providing a DSL to model functional pipelines from pipeline stages. The expressiveness of the operators is quite advanced, most notably providing support for feedback loops and stage reuse with appropriate control and stall of the pipeline. The pipeline elaboration consists in generating protocol signaling and registers from the pipeline expression. However, there is no fine-grained model of the pipeline stages, seen as arbitrary functions. In particular, latency is not inferred from stages description but must be explicitly declared by the developer.

Finally, both approaches also lack some formalization of the overall pipeline and its signals as reflexive objects with extended attributes, still preventing parameterization and instrumentation.

[2] *Reflexivity* stands for the ability to provide meta-data about an object being described, such as the latency between two points of the circuit or protocol signals associated to a data signal. Further descriptions can then be based on these elements, e.g. to conditionally generate parts of the circuit.

3.2.1.3 Auto-pipelining

While stages are the base component of pipelines, they are no longer considered a relevant piece of information in further abstractions providing automated pipelining. Such abstractions indeed aim at removing the split into stages from developer scope. They provide automated pipeline details generation based on a higher-level expression of the functionality and the associated timing constraints.

In its most basic form auto-pipelining consists in cutting combinational paths by inserting registers in order to match timing requirements [IdD17, PE17, Kem20].

More advanced auto-pipelining algorithms are found in HLS, taking advantage of raw algorithm expression to infer the entire pipeline in association with other parallelization techniques such as loop unrolling [GCW+21, MR01, KP01, DFH+20]. Auto-pipelining is notably used for micro-architectures, modeled with highly constrained pipeline specifications, in order to lower these specifications into actual pipelines implementations [GOCBK10, KKCGO08].

Recent works on dynamic scheduling of dataflow circuits [JGI18], based on the theory of latency-insensitive designs [CMSV01], target higher circuit performance by inserting buffers on critical back-pressure paths [JSG+21]. This approach, which forgoes the static scheduling in favor of protocol signaling in the inferred architecture, provides additional degrees of freedom to HLS algorithms for enhanced design space exploration. Instead of ensuring a consistent static scheduling of the complete design, it reproduces the choice of traditional hardware designers who intentionally release latency-awareness constraints of individual pipeline stages, in order to keep the description flexible and maintainable. While this approach provides interesting results, it stands at the precise opposite to our own approach of latency-aware pipeline modeling, further described in Chapter 5.

Auto-pipelining is suited to infer pipelines or pipeline stages from a loose algorithmic or arithmetic specification, providing a lot of flexibility to the designer. However, in a context of packet switching involving only very simple algorithms, it does not provide the relevant abstraction to handle both performance and complex synchronization needs.

3.2.1.4 Applicative Pipelines

While the aforementioned approaches defined fine-grained pipeline stages, other abstractions leverage the same paradigm to model coarse-grain domain-specific application pipelines. A typical application example matches the context of this thesis as it models network application pipeline. As discussed in the previous chapter, a generic model for any network application consists of successive steps a network packet goes through from input interface to output interface. This architecture is used as basis in several network DSLs with pipeline stages being each a complex function, such as a classifier or an associative counter [BDG+14, SRC+19]. Similar approaches exist in other applicative domains, such as signal processing [VG14]. These DSLs focus exclusively on high-level applicative modeling and aim at constraining as little as possible the implementation of pipeline stages. The direct consequence is that such applicative pipelines disregard the underlying implementation of stages, only considering overall specified properties such as throughput or latency. This approach is perfectly suited to generate complex application based on IPs with a top-down approach, however it does not provide an upper abstraction level in the bottom-up approach we are following in this thesis.

Nonetheless, despite current top-down approach of network applicative DSLs, the high abstraction level they model is extremely relevant for network application design. In particular, a bottom-up approach reaching such applicative level with appropriate intermediate abstractions for implementation would be very consistent. Such stacked abstraction levels

would indeed enable extended parameterization and design space exploration at applicative level while leveraging a model actually inferred from the implementation itself.

3.2.1.5 Conclusion

Pipelining paradigm is suited to describe fine-grained parallel computations as well as a large variety of applications at a coarse-grained level. In particular, network application and packet processing can benefit from this model at all abstraction levels. However, there is no pipelining-oriented abstraction unifying fine-grained pipeline stages with coarse-grained pipeline application models. This limits the control over implementation details provided by high-level pipeline models.

3.2.2 Dataflow-oriented Programming

Dataflow abstraction can be seen as a generalization of pipeline modeling. Both focus on data movement between various operators, but where pipeline stages share a homogeneous throughput, dataflow processes are heterogeneous and might consume and produce an arbitrary dynamic number of tokens. Dataflow processes are connected through infinite buffers to handle any difference of production and consumption. Synchronous Dataflow (SDF) reduces the scope to constant numbers of tokens consumed and produced by processes over time, enabling static scheduling [LM87a, LM87b, LP95]. To provide a consistent implementation with finite buffers of SDFs in hardware, static scheduling consists in duplicating some processes to match production and consumption rate at any stage of the flow. This very same approach is used in Fleet framework [THZ20] to provide massive parallelism based on a sequential user-specified token processor, with automatic generation of all the duplication and routing logic.

While dataflow is a relevant abstraction level to model applications such as signal processing of continuous data-streams, it is not suited for network applications and their discrete streams of variable-size packets interspersed with idle periods. Continuous homogeneous data-streams indeed benefit from a complex data flow to model advanced applications. However, the discrete and changing flow of chunks of packets leads to the use of very basic operators with for example no feedback loops at fine-grained level, i.e. within the same packet. Data flow modeling does not bring any relevant rise in abstraction level at this scale.

Nonetheless, the dataflow abstraction can still be relevant in the network context at applicative level where dataflow tokens consist of complete packets allocated on various independent packet processors [CB04].

3.2.3 Transaction-level Modeling

While the dataflow approach focuses on modeling data continuously flowing in and out of operators, transaction-level modeling (TLM) focuses on specifying the interactions between the operators. This decoupled approach aims at providing a system-level abstraction which specifies the communication between various modules of the circuit, modeled as concurrent processes [RSP+05]. These processes are communicating through channels, which provides an abstraction of protocol implementation details by presenting a common interface to processes. Data transfers are initiated within processes through function calls on the interface and the resulting *transaction* is completed when the function returns.

This functional approach is particularly suited to simulation and verification purposes [Str17, BFP07]. The generalized use of a common interface abstracting the implementation details simplifies the description of data transfers between modules and also considerably

helps with design space exploration at system-level to experiment with various communication protocols and architectures.

While TLM abstraction is tightly linked to SystemC language [Pan01] in which it has been formalized, other languages such as TL-Verilog [Hoo17] and PDVL [Str17] are relying on analogous concepts. Similarly, Bluespec Verilog language introduces the concept of guarded atomic actions to model operations and transactions of various actors in a circuit [Nik04, Nik08]. Although transaction-level modeling is not explicitly based on nor intended for pipelines design, both TL-Verilog and sc_pipe leverage TLM concepts within a stage-based pipeline abstraction.

While TLM incidentally uses interfaces to model and abstract transactions, other abstractions, specifically based on interface and port modeling, are presented in the next section.

3.2.4 Port and Interface Modeling

Carefully specified and generic interfaces are the basis of reusable modules and subsequently improved agility. This led to the standardization of interfaces and busses such as AXI and Avalon, as well as international effort such as the Virtual Socket Interface Alliance [SB99], aiming at providing specifications ranging from analog interfaces to system on chip design.

While these standards greatly help with ensuring compatibility of hardware IPs for system-level integration, they do not explicitly aim at providing interface abstraction in hardware development. In particular, standards are oriented towards design and not verification. Instrumentation of interfaces with additional constraints such as typing has been proposed [dMHV19] to help with system-level verification of proper interactions between IPs. Similarly, BaseJump STL [Tay18] provides a standard template library to handle communication protocols between modules, defining the notion of demanding and helpful module interface endpoints. Wire Sorts [CSBH21] is a language abstraction that further extends these concepts to enforce additional safety in hardware composition, in particular focusing on avoiding any combinational loop risk when interconnecting modules.

Further abstractions of communication channels provide higher-level interfaces which handle the abstraction of complex structured data being transmitted on a constrained communication busses [PBVS+20]. This abstraction level is very relevant in a network context, where packets are typical complex structured data, passed over constrained busses as series of data chunks.

3.2.5 Orthogonalization and Separation of concerns

Whereas the paradigms we reviewed so far focused on hardware description and abstraction, we now focus on pure software engineering and code organization, pursuing the same effort to improve agility of development. As mentioned in preamble, abstraction is all about preserving relevant information in a given context. From that definition, we presented a relatively linear path, progressively preserving less and less low-level details thanks to successive abstraction levels. However, at any given abstraction level, programs and architectures are in charge of multiples tasks at the same time. A typical example is ensuring security of a computation: some input data validation is required to ensure the security of the program, although it is not required for the computation itself. Dealing all at once with these various concerns leads to tangled code, more difficult to maintain and upgrade, hence reducing the development agility. In particular, pieces of code regarding a given concern are scattered across the entire codebase. Orthogonalization aims at untangling the unrelated concerns and dealing with them in explicitly separated contexts [HL95]. Various

implementations of orthogonalization have been explored, the next paragraphs further detail the most relevant ones.

Aspect-oriented Programming (AOP) To distinguish the various concerns a piece of code is dealing with, Aspect-oriented Programming defines a concern as an *aspect* and draws a boundary between the main code and additional cross-cutting concerns with the following implementation and its associated vocabulary [KLM+97]. On one side, each *aspect* contains *advice* about some implementation details. On the other side, the codebase contains *join points*, grouped as a *pointcut*. Each aspect specifies on which *pointcuts* their pieces of advices shall be applied. The action of applying the implementation information is referred as *weaving*, which in practice results in the generation of an updated version of the codebase or of its intermediate representation that is then processed as if the developer had addressed the concern on each join point. Aspect-oriented programming has been successfully leveraged in hardware design, most notably for verification [HMN01, Str17], and collateral generation of a physical floorplanning design [IBA+19].

Feature-oriented Programming (FOP) Presented at the exact same *European Conference on Object-Oriented Programming* as AOP, Feature-oriented Programming introduces itself as an extension of object-oriented programming [Pre97]. It aims at producing programs as a composition of independent features that can be activated or not at compilation time depending on the final applicative context. This approach offers high parameterization and the resulting program is lighter as it ships only with the required features. Original implementation relies on aggregation or multiple-inheritance of the independent features, and introduces so-called *lifters* to resolve feature interactions. A hardware implementation proposed to parameterize performance counters in a RISC-V core relies on an aspect-oriented formalization [DC21]. *Aspects* can indeed be used to capture *features*, and the concrete implementation relying on code injection at some *join points* is similar in both cases.

Separation of concerns is very relevant in hardware design and has been explored to mix properly cross-cutting concerns into a main design while avoiding both code scattering for the injected concern and code tangling of the main design. However, as far as we know, separation of concerns has only focused yet on the introduction of secondary cross-cutting concerns while the designer is at any time in charge of 1. Describing an overall functionality, 2. Choosing the appropriate hardware primitives, 3. Guaranteeing performance (timing, latency, throughput), and 4. Keeping all signals consistent and synchronized. Creating boundaries between some of these various concerns could highly benefit to design reusability.

3.2.6 Conclusion

Several high-level paradigms have been proved suitable to hardware design. Based on different concepts, they aim at achieving a common objective: increasing development agility by most notably improving the reusability of the designs. While none of them emerge as an additional architectural abstraction level, combining multiple paradigms and software engineering techniques remains yet to be explored to introduce consistent design patterns and abstractions.

Introducing and developing these higher-level abstractions requires a concrete medium to experiment with. As the highest architectural abstraction level available to design hardware, HCLs are the natural candidates for the task. The next section reviews in detail their advanced elaboration capabilities and how they can help with implementing further abstractions in practice.

3.3 Implementing Abstractions with Hardware Construction Languages

Hardware construction languages stand as an architectural evolution over traditional hardware description languages, most notably providing advanced elaboration capabilities. Generation and parameterization are key to achieve improved reusability but are also a tool to develop further abstractions.

This section reviews software engineering techniques at work behind the diversity of HCLs and explore how these techniques can help with implementing further abstractions. It concludes on the choice of an HCL that will be used as a foundation for the contributions of this thesis.

3.3.1 Review of techniques

Traditional HDLs focus on the description of basic hardware primitives such as modules, registers and wires. Introduced in the 90s, VHDL and (System)Verilog remain today the most widespread hardware description languages in the industry, due to their standardized and efficient support by virtually all Electronic Design Automation (EDA) tools. These languages are declarative and provide very little reflexivity over the circuit being described. However, they do provide basic generation and parameterization capabilities, introducing the concept of circuit elaboration from a somehow generic description [SM13]. This includes for example, parameterization of signal widths, and conditional or loop generation of certain parts of the circuit upon parameters combinations. Inclusion of these features in traditional HDLs lays the ground for hardware generation. In that respect, they could be seen as forerunner of hardware constructions languages. Nonetheless, by many aspects—including rather limited tool support—this generality remains quite restricted compared to the expressiveness available in high level languages used for software programming, which HCLs aim towards.

We describe in the following paragraphs the evolution towards hardware construction languages, leveraging various implementations, with more advanced elaboration capabilities.

3.3.1.1 External Pre-Processing

As a first step towards code configurability, external pre-processing offers the ability to produce various versions of a single hardware description. The result depends on parameterization provided to the external pre-processing tool which leverages manually inserted tokens within the description to interfere with the latter. The hardware descriptions in between these tokens are considered as simple text strings and are not syntactically nor semantically analyzed. This approach is quite powerful to structurally modify the description without any limitations imposed by the underlying description language semantics. For example, it enables swapping one keyword for another, changing the boundaries of a module or conditionally inserting Input or Output ports. In practice, external pre-processing occurs before actual lexing and parsing of the hardware descriptions, and their use in the standard hardware development flow, notably by simulation and synthesis tools. In that respect, it is supposed to produce syntactically and grammatically correct descriptions, but has by construction no means to guarantee such properties nor to provide feedback on the correctness of its actions. This highly impairs the ease of debugging of description which heavily relies on such pre-processing directives.

While VHDL descriptions have remained monolithic across all standards, Verilog has included pre-processor directives since standardized version IEEE 1364-1995, exhibiting similar features to C pre-processor system [Sny10]. Despite being restricted to very basic

definitions, macros and conditions, it outlines the basis of external code generation and is still heavily in use as part of the standard.

However, the external property of pre-processing systems make them fairly independent of any standardization which has led to many works on the subject, introducing various functionalities within hardware descriptions. While most of these efforts have remained held back in the privacy of research or corporate laboratories to solve very specific issues, some others have aimed at addressing more generic needs. Those are largely based on templates and processing languages providing convenient string manipulation capabilities such as Perl for *Genesis 2* [SAW+10, Sha11, SGS+12], Scheme for *Verischemelog* [JB99], or Python for *PyHDL* [HMLT03] and *BaseJump STL* [Tay18].

While still based on templates and pre-processing, this last example foreshadows the evolution from external pre-processing directives towards integrated language constructs which progressively blurs the lines between pre-processor and language features.

3.3.1.2 Enhancing HDLs with New Constructs

One issue with external pre-processing lies in the strict separation of pre-processor directives and language constructs. Massive usage of macros and templates leads to complex hardware descriptions and impairs code-base maintainability. To overcome this issue, a solution consists in introducing the new features as new language constructs, in order to fully integrate them as part of a single consistent new description language. The description is monolithic and consistent, forgoing the flexibility of code pre-processing but offering an integrated experience oriented towards the actual needs of hardware designers. To achieve this integration, the processing step now requires a full syntactical and semantic analysis, upgrading its awareness of the code strings as a consistent hardware description. It becomes a compiler, which leverages various lowering transformations to elaborate a low-level version of the enhanced description, directly usable by standard EDA tools. This approach takes advantage of the hardware designers' existing habits and aims at improving their design experience, which eases the adoption of new concepts and languages. For example, *Bluespec SystemVerilog* [Nik04, Nik08] extends SystemVerilog[3]while *TL-X* aims at introducing similar language extensions for various existing HDLs [Hoo17]. On the other hand, *THDL++* [SWW11] retains VHDL semantics and aims at providing a C++-style language which focuses on improving the parameterization of hardware modules.

This integrated approach yet requires providing consistent development environments, which has lead so far to solutions maintained by private companies in the form of closed-source commercial toolchains such as *Bluespec SystemVerilog* [Nik04, Nik08] or *TL-Verilog* [Hoo17] compilers[4]. As a result, while these solutions do provide new ways of designing hardware, they are hardly extendable by their users to cope with their own specific needs, which calls for more flexible solutions.

3.3.1.3 Extended Meta-programming with Embedded HCLs

Meta-programming consists in having control over the description from an upper perspective, with ability to interfere with it before it is actually interpreted or compiled in its standard flow. To that extend, the pre-processing approach we mentioned earlier is a first step in that direction but while it provides unrestricted modification abilities, it usually lacks the understanding of the underlying description. To further extend the meta-programming

[3]The extensions brought by Bluespec and its classification as HCL is discussed in paragraph 3.3.2.1.

[4]Bluespec Inc. publicly released *bsc*, the Bluespec compiler on GitHub in January 2020 under BSD-3 license [Inc20]. Earlier works had to rely on their own Bluespec compiler [G+14] in order to make further extensions to it [Gre19].

ability of external pre-processing, *pyverrilog* [TY15], and its *veriloggen* code generator, base the manipulation of the description on a fully qualified intermediate representation (IR) of the Verilog language.

On the other hand, building a brand-new hardware construction language as a library of a high-level programming language means creating a consistent hardware environment within this language, including defining all base hardware primitives from scratch. Such approach requires a tremendous amount of development to offer a viable alternative to existing HDLs, but gives in exchange full control over the semantics of this new language. Such HCLs are said to be *embedded* in their host high-level language and usually imply a much more invasive change in designers' development habits. While the two previous approaches introduced respectively external and internal extensions to existing HDLs, embedded HCLs often appear to hardware designers as completely new languages with a steep learning curve.

A diversity of host languages The syntax of embedded HCLs is rooted in their host languages which indeed often largely differs from traditional HDLs. As first HCLs emerged out of researches undertaken by language and compilation specialists, they are based on high-level languages such as SML with *HML* [LL95], OCaml with *Hardcaml* [uja19], Agda with *Pi-Ware* [PFSS15] and the widespread Haskell with *Lava* [BCSS98] and *CλaSH* [BKK+10]. As one of the first HCL, *Lava* [BCSS98] has been further extended by research initiatives such as *Kansas Lava* [GBK+09] and *blarney* [Nay18]. The widespread usage of Python as a scripting language has made it popular among hardware designer, advantageously replacing bash or Tcl within hardware development flows and widely used for integration of hardware designs as applications. Originally targeting ease of use for software beginners, it has logically been chosen as the host language for many HCLs including *MyHDL* [Dec15, JS15], *(n)migen* [Bou13], *PyMTL* [LZB14, JPOB20], *Mamba* [JIB18], *PyRTL* [CTD+17] to name only a few among an abundance of proposals [ES16, DSFE17, MML13, LM10, LKM16, Mas07, VER19]. Other high-level languages, well-known in the world of software development, have been considered as hardware description hosts, notably due to the additional programming paradigms they offer. These include Java with *JHDL* [BH98], *Hardware Join Java* [KH06] or *MaxJ HGL* [Max11, PM11] and its functional-oriented evolution, Scala, with *Chisel* [BVR+12], *Spinal HDL* [Pap16], *Scala HDL* [LLX+14], *Veriscala* [LLX+17] and *DFiant* [PE17].

Integration of inherited software-engineering features A key advantage of embedded HCLs is indeed the immediate access they offer to existing high-level language constructs and programming paradigms of their host. The most commonly exploited feature is the object-oriented programming paradigm with the definition of hardware primitives as classes and objects, providing the ground to many HCLs. The functional programming paradigm is also leveraged in most HCLs embedded in functional languages such as Haskell and Scala, but also more surprisingly in the more general purpose language Python [VER19]. To provide distinguishable features compared to traditional HDLs, polymorphism and high-order functions are also key enablers and widely used when available in the host language. The former provides extended type parameterization of circuit descriptions while the latter enables much more complex module parameterization, for example passing parts of the module behavior as functional parameters while its overall architecture remains identical. Both features bring reusability and flexibility of hardware descriptions to a whole new level.

Two fundamentally different implementation approaches Embedded HCLs share a common vision towards highly flexible hardware descriptions, however their actual imple-

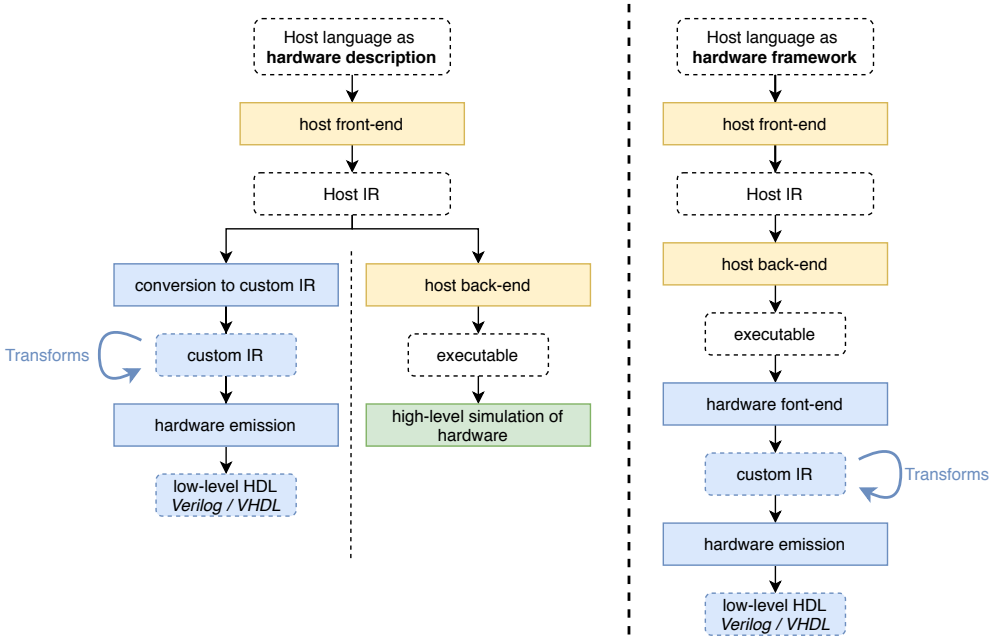


Figure 3.3: Implementation principles for embedded HCLs

mentations follow one of two fundamentally different principles, which respectively lead to various outcomes. Figure 3.3 illustrates the way they respectively operate to generate hardware in greater details.

1. Integrated approach The flow based on the direct use of the host language IR is described on the left and presents the way a first category of HCL operate, including notably *CλaSH* [BKK+10], *Scala HDL* [LLX+14] and *MyHDL* [Dec15, JS15]. The generation of hardware from the description follows the following steps:

1. Construction of host IR by host front-end,
2. Conversion of this IR into a hardware-oriented custom IR
3. Transformation of this IR, to only retain language constructs expressible with traditional HDL,
4. Emission of the resulting low-level HDL.

During this hardware generation process, the code of the description **is not executed** as a standard program. In particular, the host language constructs are directly mapped into hardware constructs, de facto providing a high-level description language. However, it might also be executed as a standard program, then providing a high-level simulation of the described hardware. As the host IR might arbitrary grow in complexity with advanced software engineering features, the generic transformation of host IR into hardware is a considerable challenge. As a result, such embedded HCLs are actually restricted to a particular subset of their host language. This leads to the same issue of unclear separation between synthesizable subset and remaining constructs encountered within traditional HDLs.

2. Explicit construction On the right side of Figure 3.3 is depicted the flow followed by the second category of HCLs, including notably *Chisel* [BVR+12], *(n)migen* [Bou13], and *PyRTL* [CTD+17]. It is based on the construction of a custom IR from scratch

during the execution of the description as a standard program within the host flow. The execution then proceeds as follows:

1. An entry point for the generation is explicitly specified within the host language, consisting of a call to the HCL generation API. It takes as parameters a top level module and its hardware parameterization,
2. The internal part of the HCL, provided as a library of the host language and referred as the *builder*, is activated and initializes an empty circuit,
3. The code corresponding to the hardware description is executed. Hardware primitives on which the user-code is based are interacting with the *builder* process upon execution, progressively building a custom representation of the circuit,
4. Once the execution of user code is over, the circuit in the form of the custom IR is complete and can be further processed. The compilation of this custom IR into low-level traditional HDL leverages iterative lowering transformations,
5. The resulting low-level HDL is emitted.

Within this flow, the integrality of the host language can be used, not effectively to *describe hardware* but rather to *implement a hardware generator* based on a predefined set of hardware primitives provided by the framework. In that respect, the lowering from initial custom IR into low-level HDL is as complex as required by the offered hardware primitives, i.e. fully controlled by the designers of these HCLs. On the other hand, this constructive flow does not provide high-level simulation of the circuit and implies to create a consistent set of hardware primitives from scratch to provide a viable hardware description language.

Various description intents All the internal machinery in action within HCLs aims at providing a flawless hardware description or generation experience to the designer. To that extend, several approaches are considered.

Behavioral Inherited from traditional HDLs which were initially conceived for hardware simulation, the behavioral description style is based on event-driven constructs to model clock, reset and registers without explicitly declaring them as such. This style of description is notably used by HCLs focusing on simulation such as *MyHDL* [Dec15, JS15].

Structural Most HCLs provide explicit hardware primitives as objects, following a structural approach. Among others, *Chisel* [BVR+12] leverages explicit hardware *bindings* as respectively registers, wires or modules. It also introduces specific `Clock` and `Reset` types.

Siloed Focusing exclusively on synchronous logic, *(n)migen* [Bou13] introduces a siloed approach between combinational and synchronous statements, whereas most hardware description interleaves both.

Gate-level From an entirely different perspective, some approaches such as *PHDL* [Mas07] focus on the description of fine-grained circuits, down to the details of operators implementation as elementary logic gates.

System-level At the other end of the spectrum, some other approaches such as *SysPy*, focus on system-level integration and System-on-Chip (SoC) designs [LKM16].

Multi-level While various languages mix several approaches, some are focused on offering a multi-level approach, with a clearly specified granularity such as functional, cycle and RTL levels in *PyMTL* [LZB14, JPOB20].

Towards architectural abstractions On top of the functioning details and design intents, some HCLs build further abstractions of the circuit. A common built-in feature consists of the abstraction of clock and reset management, with automatic connection of clocks to registers and parameterization of reset types across designs. The introduction of more advanced abstractions is discussed in Section 3.3.2.

3.3.1.4 New languages

The reuse of existing languages, either hardware ones to be pre-processed or extended, or software ones as hosts, comes with restrictions of syntax, grammar and semantics. These limitations can be bent to some extent but fuels the willingness to design new languages from scratch, such as Lola [GL98] or Pyrope [SWS+17, Ski18]. While they do not necessarily introduces new hardware description or generation concepts, they aim at solving some practical issues, such as language artifacts of embedded HCLs. However, as pure UFOs to both hardware engineering and software engineering worlds, they do not benefit from the library-reuse pattern neither from a hardware description perspective, nor from a software perspective. They require tremendous amount of effort to implement and maintain, while lacking community support which comes with the reuse of existing languages. As a result they are usually short-lived initiatives, hence preventing the ability to explore and implement further abstractions within these languages.

3.3.2 Usage of HCLs towards Agile Development

This thesis explores the idea that expressiveness and flexibility of hardware descriptions are providing development agility thanks to improved reusability and code concision. Traditional HDLs have failed to provide such expressiveness and flexibility and hardware construction languages, notably embedded HCLs, now appear as a solution [NS96, TH19]. In addition, introducing new design paradigms is expected to unlock higher abstraction levels and ease expression of complex applicative needs. The previous sections presented numerous Hardware Construction Languages and detailed their functioning principles, we now focus on the opportunities they offer towards higher abstractions levels and agile uses.

3.3.2.1 Towards New Paradigms

Design introspection Design introspection is the ability to extract meta-information from a design, such as the latency between two points of the design and to use this information for further processing. The latency is indeed not explicitly specified by the designer but is the result of its description and the relation he describes between signals with registers and various operators. The ability to extract this information has been successfully experimented with embedded HCLs, leading to various applications, either directly impacting the generation process [Pap16] or driving downstream analysis [FMR21]. Reflexivity over the circuit and free access to its internal representation—permitted by most embedded HCLs—are key enablers to increase the flexibility of the design flow.

System-on-Chip and Interface Oriented Libraries As part of the re-usability quest among hardware design world, the support of usual communication interfaces both within the design and externally is subject to extensive work. At design-level, the most advanced

circuits are often made of large subcomponents, either developed as part of the project or integrated from third-parties. Proper communication between these heterogeneous elements is key to successful integration and validation. In particular, homogenization of parameters among components communicating over shared busses is crucial and greatly benefits from automatized approach such as parameter negotiation [CTL17]. From a system-level design point of view, the integration of many components within the same design is a challenge faced by SoC designers. Approaches such as Chipyard [ABG+20] based on *Chisel* [BVR+12] and the *Rocket-chip RISC-V generator* [AAB+16], or Lite-X [KBBLL19] based on *(n)migen* [Bou13] are aiming at providing tools to generate and validate such complex designs.

Rule Based Hardware: the Case of Bluespec Bluespec SystemVerilog (BSV) introduces a new way of designing hardware with *Atomic Guarded Actions* [Nik04, Nik08]. It radically changes the way to express concurrent combinational operations, which are described as a set of independent rules to be executed within a clock cycle. The compiler is then in charge of scheduling these concurrent rules, in particular when reading or writing the same registers. It generates a circuit taking care of rules priorities thanks to *guard* signals associated to rule triggers. To that extend, Bluespec falls outside the scope Hardware Construction Languages, taking a step closer to High-Level Synthesis, as some control over the circuit is left to heuristics. The description replaces manual specification of combinational operations by functional expression of these operations. Despite available only through proprietary and closed-source tools[5], this hardware programming paradigm inspired further works [BPCC20], most notably in the formal verification domain with Kami, a coq library which implements a BSV-style hardware description [CVS+17]. According to the authors, this paradigm abstracts some details of actual hardware implementation, such as explicit parallelism, which eases the verification of the circuit in its functional form.

HCLs as Experimental Laboratory The previous paragraphs introduced numerous design concepts and abstractions developed independently, often due to specific applicative needs. As a conclusion of this part oriented towards new paradigms, HCLs appear to provide a great development platform to experiment and integrate new design paradigms. In particular, the flexibility of the embedded HCL was demonstrated with the successful integration of numerous paradigms, including Bluespec constructs within Chisel [Gre15].

3.3.2.2 The Role of Intermediate Representations (IR)

To serve the introduction of new paradigms, circuit intermediate representations (IRs) stand at the core of hardware design flows. These representations are the ground for any analysis and automation of the circuit from an external perspective. They can allow early reporting of i.e. combinational loops, design introspection as we mentioned earlier, and also serve as the medium for optimization algorithms such as *dead-code elimination* and *constant propagation*. Executed early in the design flow, these optimizations are crucial to save time for both simulation and synthesis. A wide diversity of such hardware IR has been developed, including most notably RTLIL [WGK13], FIRRTL [IKL+17], CoreIR [DTH18], LGRAPH [WPC+19, PWSR18], LNAST [WSR19], LLHD [SKGB20] and Ahir [SRAD07]. While each of these IR is tightly closed to its own original application, they intend to provide generic hardware representations, which has lead to some conjoint toolchains such as CIRCT [L+21] featuring both FIRRTL and LLHD.

[5]The Bluespec compiler was open-sourced after the cited studies. See footnote 4 on page 34 for further details.

Hardware IRs are at the core of hardware development flows and are perfectly suited to analysis and automation, however they stand apart from the designer intent by one big step: elaboration. In design flows, hardware IRs are indeed the result of elaboration of high-level hardware generators and hence do not retain generation intents such as programmatic instantiation. As a result, analysis of the circuit at this level cannot be easily reflected in the user description and cannot serve as parameter for further elaboration.

3.3.2.3 Agile Development with HCL

From their principles to their implementations, HCLs are oriented towards providing reusability and flexibility to describe and generate digital circuits. Furthermore, agile development focuses on delivering small functioning iterations which is achievable but extremely costly with traditional HDLs. Previous work demonstrated the relevance of HCLs usage in an agile development context. In particular, a specialized Tensor Processing Unit (TPU) has been efficiently designed by Google teams with HCLs [JYP+17, JYPP18, LTN+18]. Similarly, HCLs demonstrate fast iterations from design to tape-out of general purpose applications such as a multicore RISC-V chip [LWC+16, IKL+17], thanks to the flexibility of the toolchain.

3.4 Conclusion

Providing the current highest architectural abstraction level among hardware design methodologies, HCLs and their enhanced elaboration capabilities pave the way for introducing new hardware-oriented abstractions and paradigms. Numerous of such high-level design paradigms have been recently explored, and some of them appear particularly relevant in the context of networking hardware design.

The abstraction level provided by HCLs does instill some agility into hardware development flows and preserves, by-design, close control over the implementation. However, several aspects of HCL-based development flows call for further investigations:

1. While the fine-grained implementation opportunities offered by HCLs are assumed to introduce flexibility, side-by-side comparisons with traditional HDLs in an iterative context have not been clearly drawn until now.

2. Beyond implementation details, in-depth transformations of hardware design methodologies with higher-level abstractions based on HCLs have not yet been exhibited.

3. Among the various high-level abstractions already proposed with HCLs, a wide diversity of domain-specific applications context remains to be explored, such as networking and streaming applications.

4. Although HCLs are presented as a viable replacement of traditional HDLs, their cohabitation with large existing HDL codebases has been neglected so far. In particular, neither the translation of existing HDL hierarchies into equivalent hardware generators, nor the automated integration of HCL-generated hierarchies have been considered.

The next chapters intend to explore these uncharted territories, which first involves to pick an HCL as foundation of our experiments.

Choice of an HCL for further experimentation

As we previously mentioned, most HCLs share common principles and only differ in their implementations. The most flexible and advanced implementations are found with

embedded HCLs, in particular those building their custom IR out of hardware primitive upon execution. To integrate and extend high-level design paradigms within a given HCL, we expect the following characteristics:

1. An open-source HCL and toolchain to allow unrestricted experimentation,
2. A high degree of maturity to focus on integration of high-level paradigms rather than HCL engineering,
3. A well documented and flexible hardware intermediate representation to build analysis and transformations,
4. A widely used host language—in particular in the context of meta-programming—to benefit from community support and a rich ecosystem of existing software libraries.

The Scala-embedded HCL *Chisel* [BVR+12], backed by *FIRRTL* [IKL+17] hardware compiler framework, perfectly matches these requirements. The next chapters—without loss of generality—detail our contributions based on this HCL, starting with the application of agile methodologies to hardware design, from precise implementation matters to high-level abstractions.

Chapter 4

Agile Hardware Design

This chapter reviews the wide range of applicative usages opening up to Hardware Construction Languages (HCLs) and how they provide the foundations for an in-depth remodeling of hardware development flows. The induced evolution is not only restricted to hardware descriptions themselves but applies to the earliest architectural drafts, long before digging into implementation details.

After a review of existing HCL usages, we first demonstrate how Chisel HCL enables generating more flexible and more reusable hardware modules, in a network applicative context while preserving designers' control on the architecture.

Then, with this technical background gained as simple users of the new language constructs, we exhibit that HCLs can unleash designers' creativity, as early in the design flow as the initial coarse grained architectural sketches.

Contents

4.1 Hardware Construction Languages Usages and Applications

As newcomers in hardware development flows, Hardware Construction Languages (HCLs) have to prove by example their relevance and their suitability to address complex design issues in practice. To evaluate the applicability of their bottom-up approach, both sides of the latter must be studied.

At the bottom, an equivalence with existing HDLs In their lowest forms, HCLs must be able to describe complex designs without loss of control or generality compared to the traditional HDLs they intend to replace.

Upward, the ability to provide useful abstractions HCLs introduce the latest software development techniques, aiming at providing flexible and reusable hardware generators. This improved expressiveness shall nonetheless prove its relevance in specific business logic development contexts.

4.1.1 Scope Reduction

Introduced almost four decades ago and widespread in both research and industry since then, traditional HDLs have been used for a wide range of applications, with varying needs and intents. The Verilog language in particular is quite versatile, originally intended for behavioral description of circuits in a simulation context and then extended in numerous ways with SystemVerilog standards. (System)Verilog usages range from advanced simulation framework with object-oriented programming [SJR+16, KC11, DCF+14] to low level netlist structural circuit description to analog and mixed-signal modeling [FO00, PLV05]. While HCLs might appear as potential drop-in replacements of traditional HDLs, the scope of such replacement opportunity should be clearly defined. Among the numerous features and uses of traditional HDLs, the only target context of the HCLs considered in our study is strictly limited to *digital synchronous logic descriptions*, based on user-described hardware generators. In particular, there is no intended analog and mixed signals descriptions support, and HCLs do not intend to be a universal circuit descriptions used at netlist level. They instead still rely on Verilog to play this role, although numerous auxiliary researches efforts are focused on providing a replacement for the use of Verilog as intermediate circuit representation [SKGB20, WPC+19, WSR19, MB21]. With clock and reset abstractions provided in most HCLs, the synchronous logic description scope is often further reduced to positive-edge synchronization. While this is an issue to implement some low-level interfaces such as DDR memory controllers, it still covers the vast majority of custom circuit design requirements. This dependency towards the existing EDA stack based on using traditional HDLs is largely assumed, and HCLs should be regarded as front-ends for description of advanced hardware generators rather than drop-in replacement of traditional HDLs in all cases. Chapter 6 further reviews the integration issues raised by HCLs to fit within the existing EDA stacks, in particular in cases where a large number of traditional HDL descriptions is already in use.

4.1.2 Previously Validated Use Cases

Despite their inherent limitations, a wide range of applications remains available to HCLs. The current review, along with the experiments detailed in this chapter, are part of larger research effort, which started before the introduction of HCLs, exploring concrete use-cases and analyzing actual benefits of higher level abstractions for hardware design. HCLs introduce such abstractions but have usually been developed with a more or less concrete application context in mind. This is the case for Chisel, whose initial developments have

been closely related to those of the rocket-chip RISC-V generator [AAB+16], and hence oriented towards highly-parameterized processors description, with ASIC tape-out target. These studies also aim at demonstrating that HCLs can be used to describe hardware generators, not tied to a specific applicative domain but flexible enough to reveal their convenience in various use-cases.

Without loss of generality, we only consider here the numerous Chisel-related publications, which already highlight the large coverage of many applicative use-cases:

- Processor Design [LWC+16, APC15, RPPS19],
- Network-on-Chip (NoC) Design [FFDMS16, TOJ+19, KK17],
- Signal Processing [WRI+18, MP19],
- Other domain specific applications such as DNA sequence alignment [DTCC+18].

These works use Chisel from a hardware designer perspective, and mostly take advantage of advanced generation features such as extensive parameterization of complex designs, functional parameterization and type polymorphism. Leveraging these features reportedly enables their authors to build reusable and flexible designs, while contributing to the quality of their original researches in their respective domains. However, these works do not aim at providing further abstractions within the Chisel language nor at comparing implementation results between their Chisel implementations versus equivalent HDL implementations. The usability and flexibility of Chisel is demonstrated, which validates the relevance of HCL bottom-up approach, but still lacks some comparison points between Chisel and Verilog development flows in terms of performance, resource usage and reusability.

Last but not least, to our knowledge no prior research have explored the relevance of HCL usage within the specific context of network device design and implementation. In particular the introduction of high-level software abstractions as an integral part of the design process remains yet to be explored in this context.

4.1.3 Expected HCLs Contributions to Network Devices Design

To protect a worldwide network with more than 24 Tbps connected to the Internet, at OVHcloud we have developed a custom anti-DDoS protection system which successfully mitigates thousands of attacks a day. It consists of multiple custom layers featuring both high-performance FPGAs and CPUs. Our aim being to keep hardware development synchronized with software continuous improvement, HDLs have not proven to be efficient enough, raising a need for an improved agile hardware development flow. We expect such development flow to be based on simple iterations, following the *principle of least power*, which comes with the following philosophy:

1. Complexity is your enemy,
2. Do not fear refactoring,
3. Do not overengineer.

Providing explicit hardware primitives, HCLs have already demonstrated their ability to exhaustively describe custom digital designs in a quite similar fashion as traditional HDLs for a large range of applications. Their extensive generation and parameterization capabilities instill flexibility in hardware implementations, which augurs first-class compliance with our agility objectives. Besides hardware implementation convenience, we also expect software

engineering advances provided by HCLs to transpire and expand their influence to the way hardware architecture are designed from their earliest stages.

As part of the global research effort to exhibit the relevance of HCLs usage in various fields, our two contributions in this chapter detail the application of the agile methodologies to networking hardware device design at OVHcloud. The next section details the iterative design of a core functionality of stateful network devices: a per flow associative storage implemented with a *hash-table*. The following one discusses the ability to design another version of such associative storage from a radically remodeled perspective based on *Abstract Data Types*, a software engineering abstraction natively offered by HCLs.

4.2 Bringing Agility to Networking Hardware Development

In this section we aim at presenting how Hardware Construction Languages can increase agility and allow an iterative process when implementing hardware network applications. Specifically the proposed network-oriented use-case enables us to:

- demonstrate how HCLs can benefit to hardware development and help build higher architectural abstractions of circuits,
- evaluate the quality of HCLs generated circuits in terms of latency, throughput and resource usage against equivalent HDL implementations.

For this first industrial experiment with Chisel HCL, we focus on a central function, a hash-table, largely used in our network device to store states associated to network flows. It is a core feature for all stateful network devices, which are much more powerful than simple stateless network packet processing. We describe the classic cuckoo hashing algorithm and dig into its iterative implementation as a Chisel hardware module, incrementally introducing features and parameterizations. We show that the generated hardware is on par with our current, thoroughly optimized, SystemVerilog implementation both in terms of performance and resource usage. Finally, we present our analysis of the advantages and drawbacks in the use of Chisel for this design.

4.2.1 Cuckoo Hash-Table Algorithm

As detailed in Section 2.2, many network applications need to store per flow states, where a flow is usually defined by its *5-tuple*, consisting of source and destination IP addresses, transport protocol and, source and destination ports. Given the wide range of potential IP addresses (2^{128} unique IPv6 addresses available), the total number of flows is far too large to use a memory with one entry per potential flow. This problem further increases when the flow is defined with additional parameters, such as traffic category, or internal profile. In practice, a network device does not process such a wide range of traffic, hence not all slots are required, and dictionaries can be used instead of exhaustive associations.

While specialized Content-Addressable Memories (CAMs) or Ternary CAM (TCAMs) are perfectly suited for dictionary hardware implementation, these highly specialized circuits are ill-suited for FPGAs. Efficient dictionary implementation is usually obtained through the use of hash-tables, which can be based on external memories, decoupling logic and storage functions.

We here focus on one particular hash-table implementation, based on the cuckoo hashing algorithm [PR04]. This dictionary implementation provides worst-case constant lookup time while focusing on efficient memory space utilization. The present cuckoo hashing algorithm is based on N independent memory banks, respectively associated with N corresponding

hashing functions. Cuckoo hashing is well suited to hardware implementation, as memory banks can be accessed in parallel.

Given a $(key, value)$ pair, the N hash functions are applied to the *key*. Each resulting hash is the address of a slot in the associated memory bank. Any $(key, value)$ pair can hence be stored in N different slots.

Given a *key*, the lookup operation is quite straightforward:

1. N hashes of the *key* are computed,
2. slots pointed by each hash are retrieved,
3. if the *key* is found in one of the retrieved slots, the associated *value* is returned.

The lookup operation execution time does not depend on where or when the data was stored, ensuring worst-case constant lookup time.

The insert operation is a bit more complex. Given a $(key, value)$ pair to be inserted, the last step is replaced by:

3. lookup for free slots:
   (a) if at least one slot is free, randomly pick one of them for insertion,
   (b) otherwise swap the content of one randomly chosen slot with the $(key, value)$ pair and go back to step 1).

The re-insert operations of overwritten $(key, value)$ pairs are called *moves*. Having *moves* creates a possibly infinite loop in the design, if no free slots are found for each successive move. This is avoided by limiting the number of re-insertions, dropping the last $(key, value)$ pair when the limit is reached.

This hash-table was already implemented and used in our anti-DDoS solution, with many improvements and variations to fit our use-cases. As part of an ongoing migration of our applications from SystemVerilog to Chisel, we decided to re-implement it, starting with a very basic specification, and iteratively adding the required features. We target the following performance requirements:

- a throughput of one operation per clock cycle,
- a latency within hundreds of cycles, which is quite transparent at network scale.

The first design iteration is a core cuckoo hash-table module depicted in Figure 4.1. *Move* operations are blocking incoming requests until the move is successful, or when the limit is reached.

The implementation process for this module is the same with Chisel or with SystemVerilog. The hardware description closely follows the linear pipeline architecture. Expressing this simple assembly of modules from a coarse-grained point of view is straightforward in both languages. While this is expected from SystemVerilog, it validates Chisel ability to provide the necessary constructs to serve as a standard hardware description language. It also enables the inclusion of existing SystemVerilog modules as *black-boxes*, a mechanism used in this initial implementation to integrate our existing custom hash functions. While there is no significant gain in using Chisel at this point, there is also no loss in hardware expressiveness.

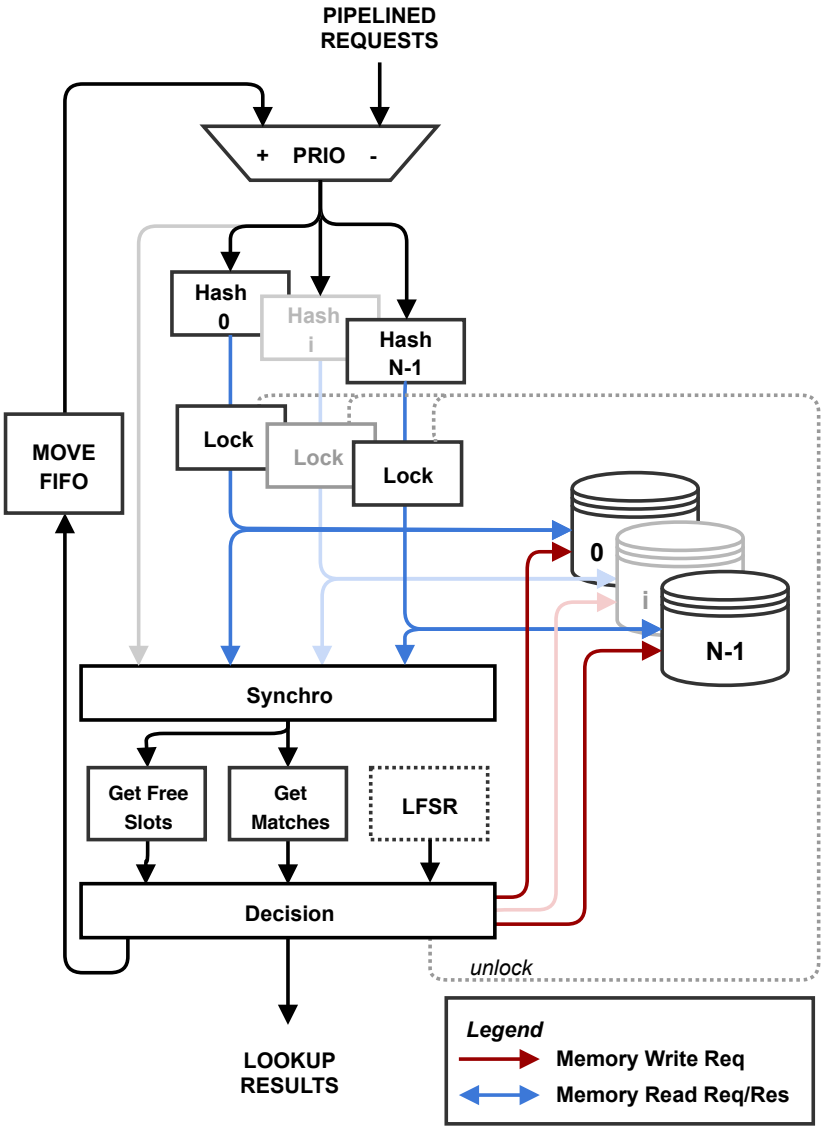


Figure 4.1: Global cuckoo hash-table architecture

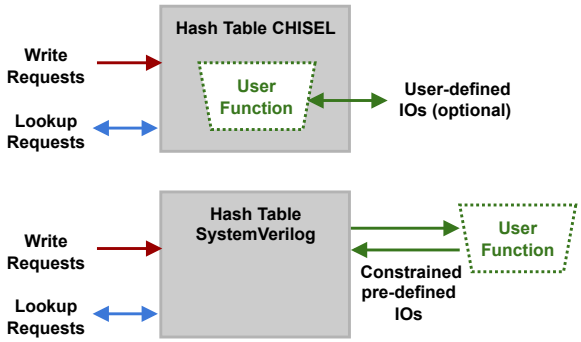


Figure 4.2: Illustration of functional parameterization in Chisel

4.2.2 Read-Modify-Write Iteration

As the second design iteration, we extend the initial implementation with the ability to process user-defined *read-modify-write* operations. For example, this allows the increment of a counter with a single request. A *read-modify-write* operation consists of 3 successive steps:

1. lookup a $(key, value)$ pair,
2. compute updated *value* given user-defined algorithm,
3. write result back to original memory slot.

The complexity of this iteration lies in the user-defined aspect of the modification. Adding an atomic *read-modify-write* is simply a matter of locking the table and inserting the user-defined function between the decision stage and the actual write to memory. But being able to externally provide the operation is not as simple. Our SystemVerilog implementation uses an external module for this operation, and carefully designed hardware ports, which comes with many pitfalls. First, increasing the number of ports increases the complexity, reducing overall expressiveness of the source code. Modifying the function means ensuring that the connection is done correctly. It also implies a complex validation of the modifier, mocking the expected in-place integration. Secondly, providing a default modifier can only be done with a wrapper module or with *generate* statements, which limits code reuse. Lastly, the resulting hierarchy seen during verification and synthesis analysis presents the modifier outside the main hash-table pipeline, resulting in harder debug and resources enumeration. In our anti-DDoS application, we use several hash-tables, with different modifier functions. Limitations encountered using SystemVerilog for this simple *read-modify-write* upgrade are symptomatic of design patterns that prevent code reusability with HDLs. As upgrades come at high maintainability and verification costs, full rewrites are often preferred in practice.

In contrast, functional parameterization and object inheritance in Chisel provides the ability to split roles. Figure 4.2 illustrates the difference between both approaches. Users can instantiate the hash-table with any modifier function, provided that it matches the software interfaces documented below (for simplicity, we use a `UInt` as data type).

```
trait CuckooModifier {
   def build(valid: Bool, input: UInt): (UInt, Bool)
}
```

The build function is the main function called in the hash-table to generate modifier hardware. An example of concrete modifier implementation is defined as follows:

```
object Increment extends CuckooModifier {
   def build(valid: Bool, input: UInt): (UInt, Bool) = (input + 1.U, valid)
}
```

Finally, the `HashTable` module takes a `CuckooModifier` as parameter, and uses it in its implementation.

```
class HashTable(
  val modifier : Option[CuckooModifier] = None
  // ...
```

```
) extends Module {
   //...
   modData, modValid = modifier match {
      case Some(mod) => mod.build(lookupValid, lookupData)
      case None      => (lookupValid, lookupData)
   }
   //..
}
```

Possible flavors of this parameterization are endless. For example, we could have used the *build* function as a parameter, but using a trait allows for more complex modifiers. We could also use a list of modifiers, along with a modifier selector mechanism, to implement different modifications in a single hash-table. With this approach, hardware interfaces are not growing out of control, hierarchy is preserved and a default child module generator can be provided to the module constructor, thus increasing reusability. Changes in code are quite limited and confidently integrated into the code base thanks to associated unit-tests. The flexibility offered by this parameterization hardly suffers from any limitation and greatly helps with integrating the Chisel implementation within our complex code base.

4.2.3 Experimenting With Hash Functions

The hash function choice is a balance between resource usage and latency on one hand, and collision avoidance on the other hand. While redesigning our hash-table, we wanted to experiment with other hash functions, which is the subject of this third iteration.

In SystemVerilog, new hashes can be integrated in several ways: using hardware ports as above-mentioned; manually selecting the desired hash function within hash-table code; or with a generic parameter, coupled to an enumeration of known hash functions using *generate* statements. However, none of these solutions avoid modification of original code for each new hash function—or only at the cost of complex and constrained interfaces. In Chisel, as described through the second iteration, functional parameterization allows fast integration of user-defined code without internal changes, thus easing design exploration and increasing reuse.

An interesting hash function candidate is the SipHash algorithm [AB12] which was deemed one of the most performant hash [SNO13]. Originally designed for software, it is highly sequential with multiple iterations of a computation—the *siproun*d—over the input message. To integrate SipHash in our fully pipelined architecture we need to unroll the sequential loops while duplicating the *siprounds* and chaining them through register stages. The number of *siprounds* is configurable and impacts the hash function properties. Within SystemVerilog, the usual way to deal with such variable length pipelines is to define an array of register stages. These stages are then connected using their respective indexes which results in low readability and highly error-prone code. Resorting to a function to factorize *siproun*d computations between registers stages is appealing. Unfortunately, SystemVerilog functions can hardly integrate registers, which prevents exploration of different registers configuration inside the stages.

In contrast, Chisel supports recursive generating functions which are able to describe both the functionality and the pipeline details by instantiating either registers or wires between stages. Hardware stages are defined during elaboration (*i.e.* the execution of these generating functions), leading to a very comprehensive match between the algorithm steps and the generated hardware stages.

This third iteration shows two other aspects of the usefulness of Chisel: the extended use of functions for both code generation and advanced software configuration. First,

the possibility to instantiate hardware inside a function allows more natural expression of algorithms. Secondly, configuration functions can be defined as software within the hardware description, whereas for our SystemVerilog hash functions, a configuration file is generated using a Python script prior to hardware elaboration. This limits the number of errors, and interestingly speeds up simulation and synthesis, as the hardware is generated in a static form, as opposed to the SystemVerilog version in which the configuration file is read dynamically by the tools.

4.2.4 Hardware Evaluation

The primary goal of languages is to allow efficient implementation in terms of resource usage and performance. In resource-constrained FPGAs, a language must be chosen carefully in this respect. As we observed the same trend over numerous possible parameterizations, this subsection focuses on the result corresponding to the following configuration:

- 75-bit key width
- 21-bit address width
- 69-bit data width
- Synthesis frequency: 200 MHz
- FPGA target: Xilinx VU9P

		Base	+ SipHash		+ *increment*		
Verilog	Total LUTs	9162	23722	*+14560*	23894	*+172*	+0.7%
	LLUTs	9162	22043	*+12881*	22215	*+172*	+0.8%
	MLUTs	0	1679	*+1679*	1679	=	=
	FFs	16705	24742	*+8037*	25135	*+393*	+1.6%
	BRAMs	11.5	11.5		11.5		
Chisel	Total LUTs	9411	24331	*+14920*	24401	*+70*	+0.3%
	LLUTs	9266	22371	*+13105*	22441	*+70*	+0.3%
	MLUTs	145	1960	*+1815*	1960	=	=
	FFs	15393	23636	*+8243*	23806	*+170*	+0.7%
	BRAMs	11.5	11.5		11.5		
Diff	Total LUTs	+249	+609	*+360*	+507	*-102*	-0.4pt
	LLUTs	+104	+328	*+224*	+226	*-102*	-0.5pt
	MLUTs	+145	+281	*+136*	+281		
	FFs	-1312	-1106	*-206*	-1329	*-223*	-0.9pt
	BRAMs	=	=		=		
Diff %	Total LUTs	+2.7%	+2.6%	*+2.5%*	+2.1%	*-59%*	
	LLUTs	+1.1%	+1.5%	*+1.7%*	+1.0%	*-59%*	
	MLUTs	+∞	+16.7%	*+8.1%*	+16.7%		
	FFs	-7.9%	-4.5%	*-5.5%*	-5.2%	*-57%*	
	BRAMs	=	=		=		

Table 4.1: Hardware resource usage comparison through iterations

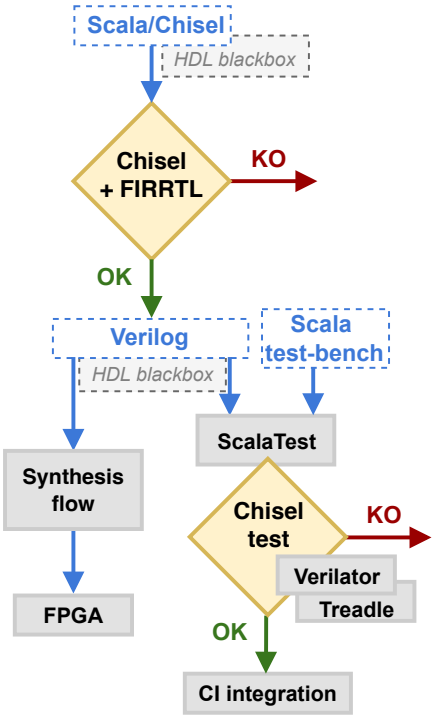


Figure 4.3: Chisel design and validation flow

Table 4.1 summarizes the resource usage for both SystemVerilog and Chisel implementations at each iteration. We took care to implement similar pipeline in terms of register stages. This shows that even if SystemVerilog allows to spare a few resources, both technologies display comparable usage. The discrepancies come from slightly different design choices, and from the randomization of hash functions. Overall, using Chisel does not come at a significant cost in resources, which is remarkable given the maturity of the SystemVerilog implementation.

Moreover, synthesis time, omitted in this table, is up to 5 times shorter for Chisel-generated Verilog than for SystemVerilog. This is mainly due to the fact that no generation occurs within Chisel-generated Verilog, which limits the required operations—and possible tool-related errors. Ease of use is also enhanced: a single Verilog file, without any external dependency, is generated and can then be used by usual EDA tools, in particular simulators and synthesis tools.

4.2.5 Agile-Friendly Design

As demonstrated with this study, Chisel parameterization process greatly improves the flexibility and reusability of hardware modules. While agility is achievable with SystemVerilog, iterations are limited by the low generic nature of the language thus impacting the entire flow. Low flexibility leads to harder iterations, which in turn lead to bigger increments, as the integration of a feature overwhelms the time required to develop the said feature. When using Chisel, the entire focus is on small increments.

This improved agility also applies to the design and validation flow, presented in Figure 4.3, through the elaboration step. In SystemVerilog, each simulation or synthesis tool has its own elaboration process, supporting its own subset of the language. Simulation tools are usually more feature-rich, and synthesis tools limitations are found late in the flow, possibly requiring in-depth modifications of already validated modules. To avoid

this, we usually stick to a very basic subset of the language, further limiting code reuse. Chisel elaboration ensures transformation of Chisel code to this basic subset. It also provides numerous checks such as types or combinational loop checking. As it occurs before functional simulation, this flow brings three major benefits. First, these checks do not need to be asserted during functional validation anymore. Secondly, many mistakes can be caught in a matter of seconds before any lengthy simulation start-up. Thirdly, the generated Verilog is compatible with all synthesis and simulation tools.

On the validation side, *ScalaTest* library natively provides test-cases management and straightforward integration of pass/failed results into continuous integration (CI) systems. It allows testing of collections of programmatically defined parameter sets. For example, it is possible to routinely check all existing hash functions within a single test. Test-benches, also written in Scala, are fed to the Chisel-tester which provides bindings with fast open-source hardware simulators such as *treadle* and *verilator*.

In conclusion HCLs introduce more checking steps, taking place earlier in the validation flow, enabling faster iterations and hence improving the overall hardware development agility and predictability.

4.2.6 Conclusion

In a challenging context of hardware network device design, which targets strong performance, an agile hardware development flow is required to cope with constantly evolving network traffic. This first positive experiment with Chisel in the design of a complex hardware module showcases the relevance of Hardware Construction Languages at the scale of industrial network device design. The high-level constructs brought by Chisel permit an actual iterative development, with small increments, and improved validation at each step.

Designing generators instead of describing circuits proves to be an efficient approach which fully exhibits its agility as soon as successive design iterations occur. HCLs integrate powerful software engineering concepts into hardware development, unlocking higher abstraction levels while still mastering generated hardware. As a direct benefit, it becomes easier to build highly reusable design libraries. Furthermore, these accessible higher abstractions levels call for a more integrated approach, not limited to an upgrade of the hardware description itself but with an impact at the earliest architecture design stages.

4.3 Towards In-depth Transformation of Circuit Design

Based on the improved flexibility gained in practice with the use of Hardware Construction Languages for architecture descriptions, this section reviews their ability to provide the foundations of a more radical hardware design flow transformation. In particular, the introduction of development paradigms inherited from software such as Object-Oriented Programming (OOP) enables hardware designers to model architecture in ways that would have never been envisioned otherwise.

To illustrate these modeling abilities, we first describe a network-oriented applicative use-case used as context for this experiment. Then we outline its implementation based on an Abstract Data Type (ADT) model, which is permitted by Chisel Object-Oriented and Functional paradigms inherited from its host language Scala. We next showcase the flexibility of this decoupled implementation, with a design iteration introducing a new complex feature. Finally, we compare the synthesis results between the original SystemVerilog version of the module and its Chisel counterpart.

4.3.1 Overcoming Hash-table Overflow Scenarios

The previous hash-table iterative experiment, depicted in Section 4.2.1, presents the need to rely on associative memories or dictionaries to store per flow states. The underlying limitation is driven by the relatively small size of available memories compared to the gigantic number of potential flows a network device should be able to process. With hardware dictionaries, which can be implemented as hash-tables, a predefined memory size is allocated to store the same gigantic potential amount of flows. Besides the issue of hash conflicts caused by limited memory depth, total capacity of a hardware hash-table is finite, and data overflow might occur as soon as too many distinct flows are attempted to be stored.

With regard to the current network context focused on per flow storage, any hash-table implementation could be potentially overflown by an attacker able to craft network packets. With a single traffic generator connected to the Internet through a unique IP address, it is indeed possible to craft packets towards a given destination IP address with any arbitrary source IP address. Crafting network packets with an arbitrary source IP addresses which does not correspond to the address of the device sending them is referred as *IP spoofing*, with four purposes:

1. hide sender identity,
2. impersonate another computer system identity,
3. redirect the expected response of a request to a selected target (reflective attacks), or
4. artificially generate distinct flows from a single device.

Some transport protocols such as *TCP* prevent such behavior, or at least mitigate the risk in practice, with the establishment of a connection through an initial handshake. *UDP*, a widely used protocol, does not provide such mechanism and is hence largely exploited by attackers. Dictionaries used in the implementation of an attack mitigation infrastructure must be resilient against those phenomena with appropriate protection or fail-over mechanisms.

Besides intentional spoofing, the wide diversity of flows received by a network device might have a varying relevance to the actual applicative functionality provided by a hash-table. Considering an application monitoring the amount of traffic per flow, only flows reaching a predefined threshold are eventually of interest to activate a rate-limiting mechanism. In particular, in the context of protecting the network against massive traffic surges, most of the *legitimate* traffic does not reach one-tenth or even one-hundredth of those thresholds. Maintaining exact counts for every single flow is thus not required from an applicative point of view.

Probabilistic counting strategies such as count-min sketches (CMS) have notably been designed to provide solution to discriminate *heavy-hitters* in large streams [CM05]. In a nutshell, for each $(key, value)$ entry, N hashes of the *key* are computed, and the N corresponding memory entries are updated by adding *value* to their respective counts. If the minimum of the N resulting counts reaches a given threshold, an alert is raised. This algorithm can be used as front-end protection to discriminate flows whose traffic reach a given threshold, and select only these flows for accurate counting with a standard hash-table. The overall resulting architecture of such *Protected Hash-table* is the following:

Probabilistic counting Flows are first counted with a probabilistic approach, such as count-min sketch, raising alert as soon a given threshold is reached. This approach may trigger false positive cases, which are due to hash conflicts. When this first stage is triggered, flows are further processed by a deterministic stage (namely, a hash table) to provide an accurate count.

Accurate counting Packets for which the alert has been raised are then precisely counted using a deterministic approach, which maintains an accurate count for each distinct flow inserted. Based on this accurate information, appropriate action can be taken as soon as a flow reaches a threshold.

This two-stage strategy provides a decent protection of the deterministic counting structure against spoofing cases[1] and permits a proper sizing of memories for storage depending on applicative needs.

A notable limitation of this methodology lies in its periodic behavior: alert are triggered when a counter reach a threshold **within a given period of time**. In practice, all counters are reset to zero at the beginning of each period, only preserving the knowledge of an existing alert. For any flow, the first period is spent only in the probabilistic stage whose alert does not imply an action, resulting in an initial *grace period*. Then, if an alert was previously raised on the given flow, this flow is inserted in the standard hash-table, to provide an accurate counting and eliminate potential false-positive alerts. In network devices, a one-second period provides a respectable trade-off between aggressive reaction on short peaks and permissiveness towards dangerous traffic. It also eases the implementation as the rates (volume per second) are directly equal to the value of the threshold, expressed in number of bytes or packets.

4.3.2 Towards a Generic *Protected Hash-table* Architecture

Providing an overflow protection for hash-tables, tailored to their actual purpose, is a desired feature for several network functions, such as *rate-control* or *zombie detection*.

Figure 4.4 illustrates the generic 2-stage architecture of such *Protected Hash-table*. It is based on a degenerated version of the count-min sketch algorithm, with a simple sketch using a single bank of memory for probabilistic stage. Two independent modules are in charge of the periodic cleaning of the counters. They share their memory accesses with the modules of the main pipeline through a memory interface manager which is in charge of guaranteeing the consistency of the memory access at all times. From a *black-box* perspective, this architecture exposes only three data structures through its interfaces: a *request*, a *result* and some elements stored within the memory (*StoredData*).

Based on this simplified block-level view, several functionalities could be implemented. However, while the *business logic* would be very different for rate-control or zombie detection functions, the overall *architecture implementation*, pipelined and based on the *read-modify-write* pattern detailed in Section 4.2.2, is quite similar. This observation calls for an explicit separation of the *business logic* and the *architecture implementation*.

Business Logic *Functionality*	Architecture Implementation *Structure*
Data Definition & Interpretation	Data Routing & Synchronization
Algorithms	Pipeline Hierarchy
Decisions	Protocol Signaling

This separation of concerns between functionality and architecture intends to provide additional flexibility to the resulting implementation, with the following goals:

[1] This strategy is only intended for self-protection against spoofing and does not aim at protecting destination network and devices against spoofed packets. Note that the size of memories shall be dimensioned in accordance with the minimum intended thresholds.

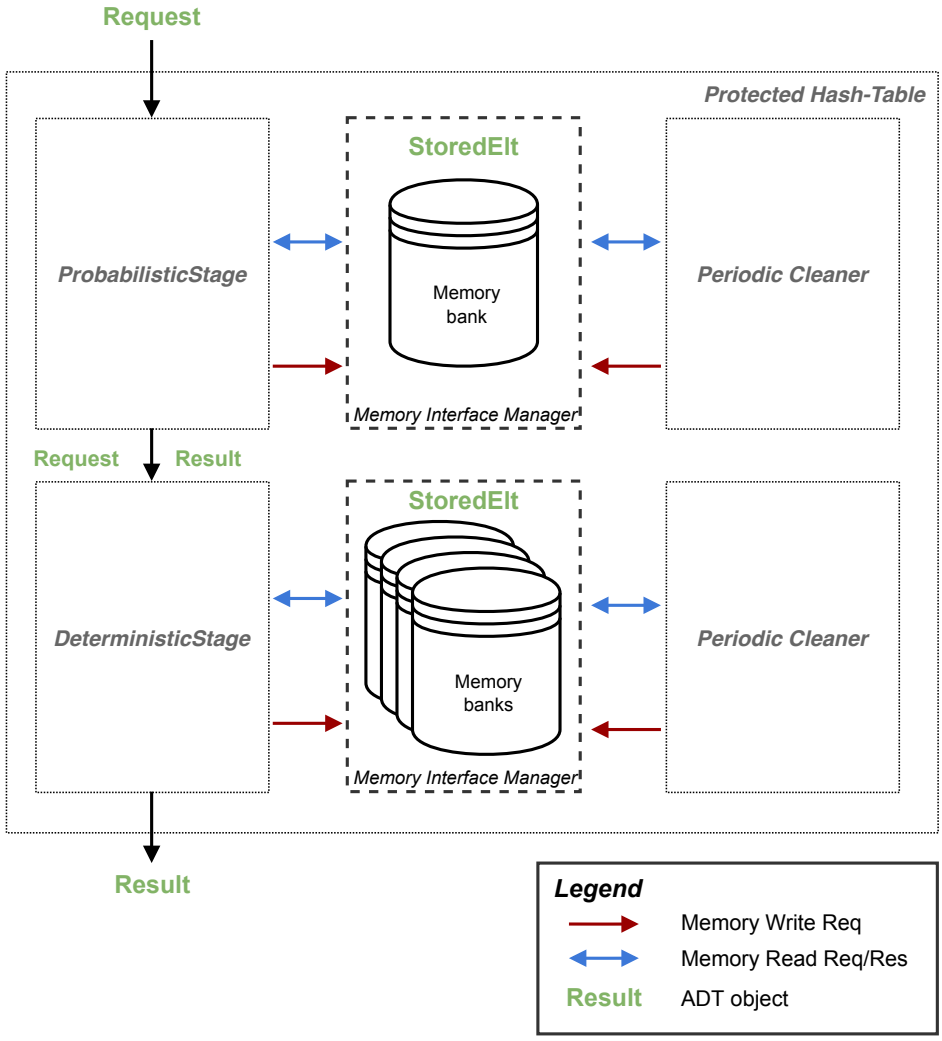


Figure 4.4: Generic *Protected Hash-table* architecture

Reusability The generic architecture can be reused directly in different and non-anticipated functional contexts.

Readability The business logic is concentrated in a single point, identifiable and not scattered. Similarly, the architecture implementation is no longer polluted with functional-related details, suppressing code tangling.

Iterative Independence Thanks to a clearly defined interface, each concern can be iteratively upgraded without taking its counterpart into account.

On one hand, such separation of concerns could theoretically be implemented with traditional HDLs, at a very high engineering cost, as these languages lack implementation flexibility. However, it would lead to an increased complexity of the resulting implementation, *de facto* defeating the aforementioned purposes of the separation.

On the other hand, HCLs bring software engineering techniques to hardware generator implementation. This not only permits to describe hardware with more flexibility but also gives access to advanced software engineering methodologies, in particular design abstraction principles. The next section reviews how *Abstract Data Types* could help with the integration of this separation of concerns from the earliest design stages.

4.3.3 Designing with Abstract Data Types

Describing hardware generators within a high-level software engineering language unleashes the power of the software design techniques for hardware architectures and data flows modelling. Among numerous software engineering concepts available, *Abstract Data Types* define a data-oriented paradigm as follows:

> "An *abstract data type* defines a class of abstract objects which is completely characterized by the operations available on those objects. This means that an abstract data type can be defined by defining the characterizing operations for that type." [LZ74]

Applied to our generic *Protected Hash-table* architecture, we can define the three main data objects as follows:

Request Input of *Protected Hash-Table* pipeline, the request provides the $(key, value)$ pair,

Stored Data (StoredData) At the core of the processing lies the stored data which is retrieved from the memory and updated based on the current request,

Result Computed from both previous objects, the result is presented at pipeline output.

Schematically, architectures of probabilistic and deterministic stages are almost identical and to preserve the conciseness of the code snippets, we consider a single ADT scheme which can be applied to both stages. The overall architecture can be expressed by the following equations, based on our three ADT objects:

$$\textbf{Request} \xrightarrow{\text{fetch}} \textbf{StoredData} \tag{4.1}$$

$$\textbf{StoredData} + \textbf{Request} \xrightarrow{\text{compute}} \textbf{StoredData} + \textbf{Result} \tag{4.2}$$

In practice, the internal architectures can be further detailed as follows:

$$\begin{cases} \textbf{Request} \xrightarrow{\text{getRawID}} identifier \xrightarrow{\text{hash}} address \\ \textbf{Request} \xrightarrow{\text{doLock}} lock \end{cases} \tag{4.3}$$

$$lock + address \xrightarrow{\text{read memory}} \textbf{StoredData} \tag{4.4}$$

$$\textbf{StoredData} + \textbf{Request} \xrightarrow{\text{update}} \textbf{StoredData} \tag{4.5}$$

$$\textbf{StoredData} + \textbf{Request} \xrightarrow{\text{getResult}} \textbf{Result} \tag{4.6}$$

The three main data types are printed in bold font, while intermediate variables—belonging to the implementation domain—are printed in italic font. Data oriented functions are printed in green and implementation related functions remain black.

We observe the central position of **StoredData** ADT object in these equations, and select it as the main abstract data type to implement the interface. From a design perspective, it sets the methods at the core of the computation. In particular, the **StoredData** might contain *private* fields, only stored for business purposes and unrelated to the other ADT objects. In rate counter case, the running counters and the concept of previous alert are business critical but are neither related to the **Request** which only carries a $(key, value)$ and associated thresholds, nor to the **Result** which returns the current alert status. Similarly, the **Result** is only interested in final binary result: alert raised or not.

As a result, business logic is concentrated within **StoredData** abstract data type, allowing to reduce the specifications of **Request** and **Result** to a bare minimum. For current ADT scheme implementation, **Result** is a bare Chisel type without particular methods or attributes and **Request** is detailed in the following code listing.

```
1  /** Specification of Protected HashTable requests */
2  trait Request {
3    /** returns the raw ID to be hashed for memory addressing */
4    def getRawID: UInt
5
6    /** returns true if the current request require a lock during memory access */
7    def doLock: Bool
8  }
```

Methods presented within this listing and the following one are documented with comments above their declaration, and their usage within probabilistic and deterministic stages implementation is illustrated respectively in Figures 4.5 and 4.6. These architectures follow the principles of equations 4.3 to 4.6 and exhibit a structure very close to the cuckoo hash-table pipeline detailed in Section 4.2.1, Figure 4.1.

Here the `Request`, with methods `getRawID` and `doLock`, is providing the values required to create a memory request to retrieve the main `StoredData` object.

We retain a mathematical vision of the operations, based on the $(key, value)$ pair, rather than exhaustively applying abstract data type principles. In particular, usual software abstract data types would natively provide the *hash* associated to their object. This approach is relevant in software because hashes have a predefined size, generally a multiple of 32 bits because their value is computed by 32 or 64 bits processors. These constraints are platform related and do not depend on the usage of the data type, e.g. when a smaller hash is required, the same hash is computed and only a subset is used. In hardware, computations are fully custom and not constrained to any predefined width, it is hence possible to provide efficient hash implementations at a very low resource cost and precisely sized to the local needs. In the context of associative memories, the width of hashes usually equals the memory address width, which is a highly implementation-dependent parameter.

Finally, `StoredData`, the main ADT object is defined as follows:

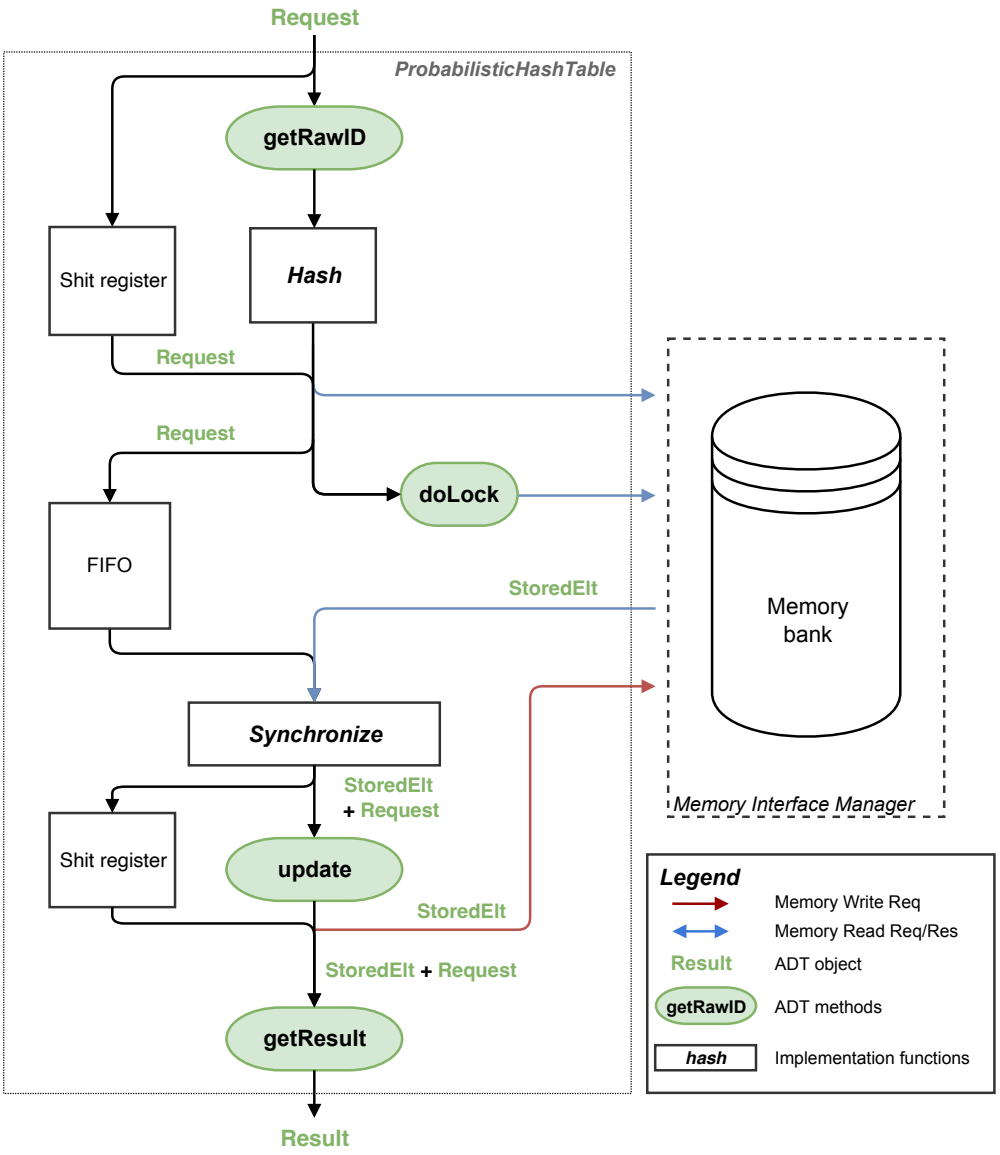


Figure 4.5: Probabilistic hash-table implementation

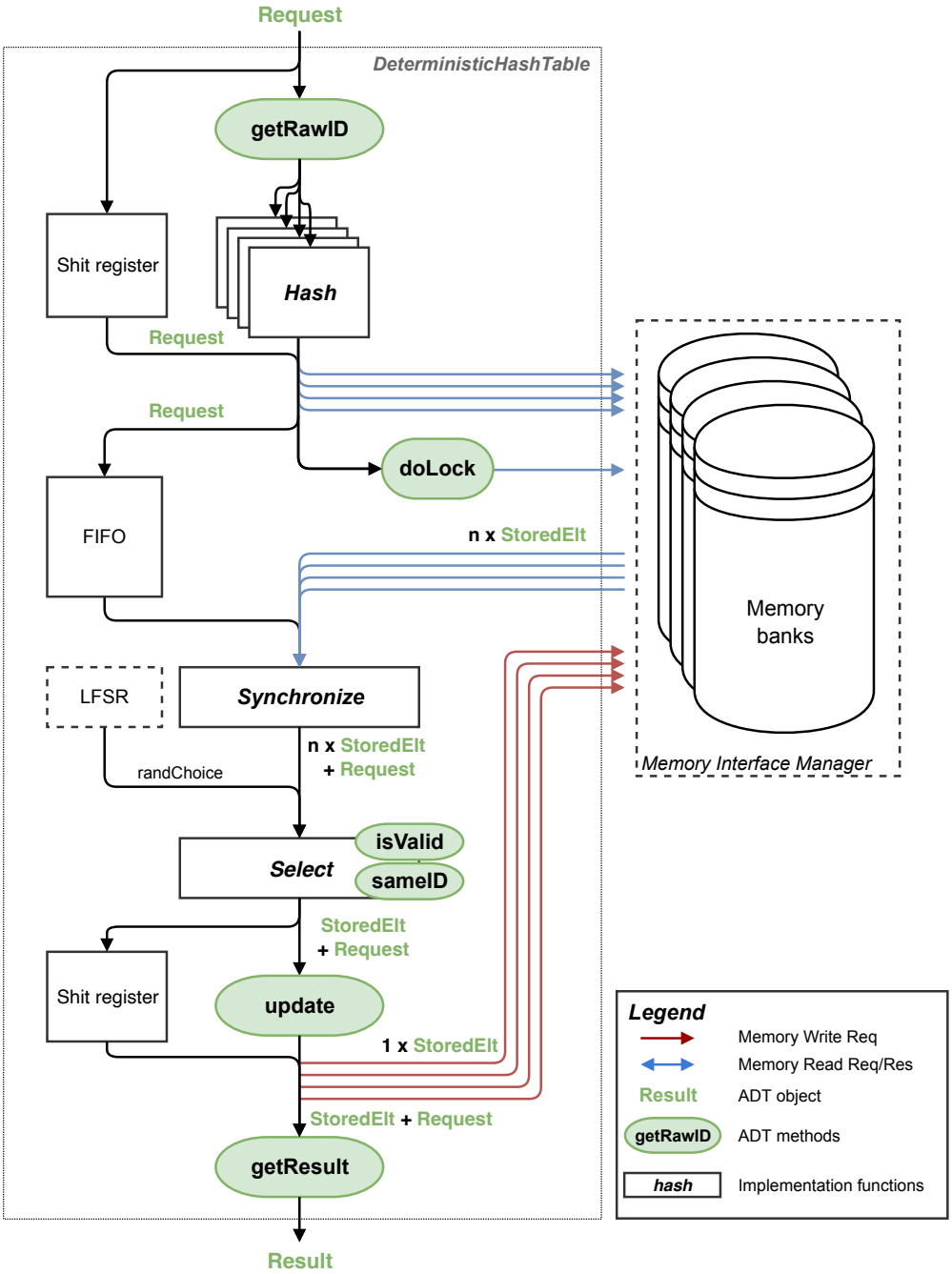


Figure 4.6: Deterministic hash-table implementation

```
/** Specification of Hashtable Stored Element
 *  type parameter R = underlying Data Type mixed into HTRequest
 *  type parameter T = underlying Data Type mixed into this trait
 *  type parameter U = simple Data Type used as output result type
 */
trait StoredData[R <: Data, T <: Data, U <: Data] {
  /** D: full data type of request */
  type D = R with Request

  /** hardware function returning true.B if this is a valid memory entry */
  def isValid(req: D): Bool

  /** hardware function returning true.B if this element id equals request id */
  def sameID(req: D): Bool

  /** Indicate the number of clock cycles needed for the update function.*/
  val updateDelay: Int

  /** Hardware function returning an updated version of this elt */
  def update(req: D, en: Bool): T with StoredData[R, T, U]

  /** hardware function returning the result expected at output of hash-table */
  def getResult(req: D): U
}
```

While the `Request` methods are focused on the identification of the $(key, value)$ pair to initiate the appropriate memory read request (equation 4.3), the `StoredData` contains the core *business logic* with all operations related to the `StoredData` update and final `Result` computation (equations 4.5 and 4.6). Most `StoredData` methods are taking a `Request` as argument, as the computations require both values. As a result, `Requests` are forwarded through shift registers and FIFOs to the lower part of implementation pipelines where `StoredData` operations take place, as illustrated respectively for probabilistic and deterministic stages in Figures 4.5 and 4.6.

Despite the focus of ADT objects on *business logic*, some implementation details still emerge in the definition of `StoredData` due to the limited reflexivity over externally defined hardware generation functions. The attribute `updateDelay` and the enable signal `en` illustrate the need for the implementation side to keep control over the `update` function, regarding both its latency and its ability to support back-pressure. The interface specified here assumes a uniform back-pressure within the `update` function and constant latency. These requirements are quite strong and might hide complex issues as they are purely declarative. In particular, the implementation side has no ability to check their consistency against the actual implementation of the `update` function on the `StoredData` definition.

The overall ADT structure assumes here that the designer in charge of the architecture implementation is the *user* of abstract data type attributes and methods. This implementation is intended to be compatible with any concrete instantiation of ADT objects and thus provides a data-agnostic functionality such as deterministic or probabilistic associative memories. On the other side of the ADT interface, the *provider* of concrete types defines the *business logic* of an application. The final user, integrates and configure both parts of the ADT scheme to generate an actual piece of hardware with the expected functionality. As a result, based on an ADT scheme, there are up to three completely decoupled roles in the construction of a final piece of hardware:

ADT User Design configurable *implementations* such as probabilistic and deterministic stages,

ADT Provider Design *business logic* such as probabilistic rate-counting or zombie detection,

Integrator Configure *implementation* and *business logic* as an actual hardware function, and integrate the resulting piece of hardware within a larger hierarchy.

This highly decoupled organization, based on one common data-oriented and abstract specification enables developers and integrators to progress concurrently on their work without strong timeline synchronization requirements. Similarly, update of respective parts can occur simultaneously which provides great flexibility in the development flow.

4.3.4 Implementation & Results

The data-oriented design of this new protected hash-table architecture is promising from the perspective of design flexibility and reusability. Considering the surrounding industrial context in which this design is to be integrated, and the willingness to preserve an agile development with reasonable iterations, the following successive steps have been planned for its integration within the existing code base:

1. Design of the initial ADT scheme,
2. Implementation of a drop-in replacement of the existing *probabilistic* counting stage, based on the ADT scheme and integrated as an independent *ProbabilisticHashTable* module,
3. Implementation of a drop-in replacement of the existing *deterministic* counting stage, based on the ADT scheme and integrated as an independent *DeterministicHashTable* module,
4. Design and implementation of the *memory interface manager* in charge of sharing memory access between multiple clients (namely here between main pipeline and periodic cleaner), with lock management,
5. Integration of both stages with memory interface manager and periodic cleaners as feature-equivalent replacements of the existing rate-counting solution,
6. Implementation of a *zombie detection* feature, self-contained within the *business logic* and requiring no modification to neither probabilistic nor deterministic stage respective implementations,
7. Unification of the ADT methods and objects to provide a unified *Protected Hash-table* module, with ability to store deterministic elements within the probabilistic stage, in an attempt to reduce the average number of memory access for each packet,
8. Attempt to merge implementations of both stages which exhibit a high degree of similarities, in order to reduce code duplication. This final step is highly experimental as reducing duplication within two simple implementations might result in the creation of one complex and hardly maintainable implementation, at the exact opposite of the original agility research.

In order to further demonstrate the usability of Chisel outside research-oriented experimentations, these developments have been carried out by a team of three hardware engineers of our FPGA team at OVHcloud. To get them started with the Chisel/Scala stack, and description of hardware generator, we set up several training sessions with short lectures and various coding exercises, from hardware description basics in Chisel to complex generation patterns exhibiting the power of Scala. We developed custom pieces of training

material, adjusted to the uneven experience and appetence for hardware abstraction among the teammates, and based on *Jupyter Notebooks*[2]provided by Chisel community[3].

The developments and integrations corresponding to steps 1 to 6 have been achieved within a reasonable timeline (around 4 months[4]), however due to external constraints, iterations 7 and 8 have not been attempted.

The following paragraphs discuss the resource usage reported after place and route on a *Xilinx VU9P* FPGA at 220 MHz. The impact of this major upgrade of development flow and language is observed through the following iterations:

1. Initial Verilog version,
2. Chisel version, after integration of probabilistic and deterministic stages, and with equivalent features to Verilog version
3. Chisel version, with additional *zombie detection* feature

The figures present the resource usage of the complete network device in which this new design has been integrated. As a consequence, the difference in resource between iterations remain relatively limited in all cases.

		Verilog	Chisel		
Resources	Total LUTs	534,699	537,418	*+2,719*	+0.5%
	LLUTs	478,315	481,934	*+3,619*	+0.76%
	MLUTs	19,901	19,001	*-900*	-4.5%
	SRLs	36,483	36,483	=	=
	FFs	843,859	843,111	*-478*	-0.05%
	BRAM	1,829	1,856	*+27*	+1.5%
	URAM	156	156	=	=

Table 4.2: Verilog and Chisel resource usage comparison after place & route on a *Xilinx VU9P* FPGA

Table 4.2 illustrates the first iteration and compares the resource usage of the original Verilog implementation against its feature-equivalent Chisel counterpart, designed with the Abstract Data Type methodology. The resource usage is roughly similar for both versions, with increased Logic LUTs usage and slightly reduced FFs usage in the Chisel version. These results are almost in line with the evolution observed in the previous cuckoo hash-table experimentation in Section 4.2.4. The newly introduced custom memory interface manager, in the Chisel version, also contributes to the considerable Logic LUT count increase. Last but not least, as the overall implementation pipelines are different, we observe a reduction of Memory LUTs usage, in favor of an increased usage of Block RAMs primitives.

[2]A *Jupyter Notebook* is a web-based interface allowing to experiment with software without requiring user installation and configuration. It provides a customizable playground and all code snippets are run on the server. Open-source software freely available at https://jupyter.org/

[3]Online Chisel Bootcamp https://mybinder.org/v2/gh/freechipsproject/chisel-bootcamp/master

[4]Time elapsed between design and validation on the FPGA. Internal time tracking reports a grand total of 271 days of work, spread across 8 months, which is equivalent to 14.5 human.months with current OVHcloud holiday policy (7 weeks/year). These figures highlight the substantial effort poured, not only into this design but also the development flow. In particular, it includes all the training sessions and many internal tool developments, which were required to match developers' requirements in terms of user experience with the Chisel stack.

		Chisel	+ *Zombie*		
Resources	Total LUTs	537,418	537,590	*+172*	+0.03%
	LLUTs	481,934	481,956	*+22*	+0.00%
	MLUTs	19,001	19,005	*+4*	+0.02%
	SRLs	36,483	36,629	*+146*	+0.40%
	FFs	843,111	844,012	*+901*	+0.11%
	BRAMs	1,856	1,856	=	=
	URAMs	156	156	=	=

Table 4.3: Impact of *zombie detection* feature addition, after place & route on a *Xilinx VU9P* FPGA

Table 4.3 illustrates the impact of the *zombie detection* feature addition on the Chisel version. This addition logically contributes to an increase of all resource usage, in relatively small proportion compared to the size of the design.

However, the key result in this iteration does not regard the amount of resource at stake, but the validation of the ADT design methodology goals:

Reusability The exact same architecture has been used to introduce the *zombie detection* feature, without any change to the implementation side,

Readability The business logic is concentrated in a single point, identifiable and not scattered. Only several ADT methods have been modified to implement the feature,

Iterative Independence The new feature has been developed as an independent iteration while other unrelated development were carried out on the implementation side, such as synchronization issue resolutions and performance optimization.

As a conclusion, the abstract data type methodology meets the objectives initially stated. The modification of the design flow occurs as soon as the earliest coarse grained block diagrams and propagates down to the implementation details, providing advanced flexibility and decoupling implementation and *business logic* concerns.

4.3.5 Limitations of Abstract Data Types in Hardware Design

The complete ADT scheme corresponding to the current implementation status is included in Appendix A.1. It differs from the unified approach presented above by introducing specific functions for probabilistic and deterministic schemes. It also introduces additional functions that were found useful during optimization phase of the design, such as additional signals returned by the update result, which allows reducing the number of write operations. As a result, the current implementation of the abstract data type is not as smooth as originally expected. In particular, and as detailed in the previous sections, the abstract data type methodology application to hardware generators suffers from the following limitations:

Limited hardware function reflexivity While high-order functions and object-oriented programming enable defining *business logic* related hardware functions in the data object, the implementation side requires additional knowledge to integrate them in its fine-grained architecture. Unfortunately, metadata associated to the function cannot be natively retrieved from the function itself, due to limited reflexivity towards the circuit under elaboration in this particular context. As a result, and despite the intended

exclusive focus of ADT objects on *business logic*, some implementation details are still required to be defined by the abstract data type provider, such as latency or control signals.

Cost of standardization Carrying additional information has a very high cost in hardware, especially for data structures intended to be stored in memory. With abstract data types, standardization of data structures is quite appealing to provide a smooth and generic interface, which often involves providing additional methods and attributes. As a result, while full control over implementation is always preserved, such generic interface might quickly shift away from a zero-cost abstraction if no extra care is taken in their implementations.

Deviation from high-level architecture principles Inferred from the earliest coarse grained sketches of the architecture, ADT objects are defined with methods based on the expected implementation needs. However, our experience tends to show that in practice implementation needs cannot be fully anticipated from the drawing board and some of them arise with the actual development and validation of the circuit. As a result, the expected iterative independence of implementation and business logic tends to be limited in practice. Moreover, with back-and-forth between both sides, the final implementation might progressively deviate from the generic decoupling with introduction of tangled warts, defeating the original purposes of ADT use.

While exhibiting powerful flexibility and reusability capabilities, implementations of Abstract Data Types in practice might prove to be more difficult than expected. In particular, the inherent complexity of hardware development leads towards the introduction of more fine-grained constraints than expected. This results in a real risk of creating *Single-use Abstract Data Types*, with so-called generic interfaces, which in practice appear to remain tightly coupled between implementation and business logic.

4.4 Conclusion

In this chapter, as part of a global research effort on high-level hardware design methodologies and in particular on hardware construction languages, we validated the relevance of HCL design flow in an industrial network context and demonstrated that thanks to HCLs, software design methodologies can be applied to hardware design.

In direct line with the numerous existing HCL use cases presented in the introduction of this chapter, our contribution towards the validation of HCLs relevance in a broad range of applicative contexts has been developed and published in a very active context [BHM+21]. Several other research efforts, published concurrently to our work have contributed to enlarge the applicative spectrum of HCLs as a relevant high-level hardware development paradigm. Without loss of generality, here is a curated list of the latest published Chisel applications, ranging across a wide variety of domains such as:

- Cryptography [GEHM+22],
- Matrix-Multiplication [FMR20],
- Data Serialization [KLK+21],
- Database Query Processing [MFL+21, VRO21],
- Signal Processing [DLL+21, DPM21],
- System-On-Chip [KJA+21],

- Neural Network [Dan22],
- Secure Processors [MBM+20],
- Memory-oriented Architectures [XLT+22],
- Optimized Graph Traversal [LSW+21],
- Tensor Computation [XZW+21].

These works showcase similar results to ours, highlighting the positive impact of Chisel on design reusability while preserving equivalent resource (or area) usage in comparison to feature-equivalent implementations with traditional HDLs [KH21, GEHM+22].

Besides the flexibility gain offered by HCLs for architecture implementations, we demonstrate that high-level software design methodologies can be applied to remodel the hardware design flow as early as coarse grained architectural sketches. From the main data structure appearing on these sketches, *Abstract Data Types* are defined and take the central role in the design process, naturally resulting in a data-oriented implementation, with a high-level of decoupling between implementation and business logic. HCLs natively support this design paradigm thanks to object-oriented and functional programming paradigms provided by their host languages. These implementation paradigms indeed provide the appropriate constructs to define interfaces and objects, to extend them depending on implementation details and finally to allow advanced parameterization, based on type inheritance or high-order functions. As a result, separation of concerns offered by abstract data types further improves reusability, readability and iterative independence of the implementations.

Beyond being yet another way of describing hardware, hardware construction languages introduce high-level software engineering constructs into architecture descriptions, which highly improves both flexibility and reusability of these architectures. Application of advanced software engineering paradigms can even remodel the way of designing hardware architectures, but they remain distant from core hardware concerns and might induce a progressive shift away from the zero-cost abstraction desired by hardware designers. To overcome these limitations, the next chapter focuses on introducing hardware-oriented abstractions, in order to provide powerful design abstraction and enhanced flexibility while still guaranteeing control and performance.

Chapter 5

Pipeline Design Methodology

This chapter focuses on raising the abstraction level of hardware descriptions based on a widespread hardware design pattern: pipeline stages.

After a short discussion regarding the relevance of Hardware Construction Languages as base layer for development of hardware-oriented abstractions, we introduce our abstraction proposal based on a fine dissection of the pipeline stage pattern.

Then, we detail the implementation of our methodology, first focusing on the construction of a synchronization model from a pipeline-oriented description and then detailing the resolution algorithms used to obtain a fully-synchronized synthesis-ready circuit.

Finally, we validate our methodology and its implementation as a Chisel library on concrete pipeline examples.

Contents

5.1 Introduction

The previous chapter reviewed usage of HCLs, first from a hardware designer perspective and then from a software developer perspective, applying a software abstraction pattern to the description of hardware architectures. While showcasing the relevance of HCLs to introduce such complex abstraction patterns, it highlights the limitations of applying software defined abstractions in the specific context of hardware design.

To overcome these limitations and achieve the best design agility and circuit reusability, hardware description languages must take actual hardware designer needs into account. Leveraging the flexibility of HCLs as software frameworks, several studies focus on providing such an enhanced development experience, from hardware description expressiveness to verification to circuit optimization. Without loss of generality, a summary of such studies is listed below, based on Chisel HCL stack and providing extension libraries with various intents.

- Front-end Constructs [CTL17, Gre15].
- Hardware/Software Co-design [VLWA13],
- Test and Verification Frameworks [KMK+18, LKK+18, DGX+19, DGX+17],
- Target-oriented Circuit Optimizations [WIS+18, SHR+19, KZBA19],

All these extensions, libraries and frameworks demonstrate the flexibility, not only of Chisel as a Hardware Construction Framework, but also of its entire compilation stack down to Verilog generation. Chisel underlying Intermediate Representation (IR) and its compiler *FIRRTL* [IKL+17], have indeed been designed as a highly flexible stack, offering by design the ability to execute user-defined circuit transformation and analysis, with a comprehensive API.

Based on this ability to introduce abstraction layers within HCLs, in this chapter we aim at raising the abstraction level from a hardware development perspective, with a focus on the pipeline stage pattern. We first introduced this pattern in our problem statement to exhibit the limitations of traditional HDLs in iterating on such a simple pattern. The next section details the considerable impact of this pattern in architecture implementations and introduces some abstraction targets to improve the design reusability of pipelined architectures.

5.2 Towards Latency-aware & Protocol-Polymorphic Pipelines

5.2.1 Motivations

To introduce the need for latency-aware and protocol-polymorphic designs, and provide a definition for these terms, we first reproduce and extend the iterative design example initially detailed in Section 2.3.1.2. The first iteration starts with the following example of an adder module written in Verilog HDL:

```
1  module add(
2      input [5:0] a, input [5:0] b, output [5:0] r
3    );
4    assign r = a + b;
5  endmodule
```

As the second iteration in the design process, a register is inserted to break the combinational path from `a` and `b` inputs to `r` output, creating a simple pipeline stage:

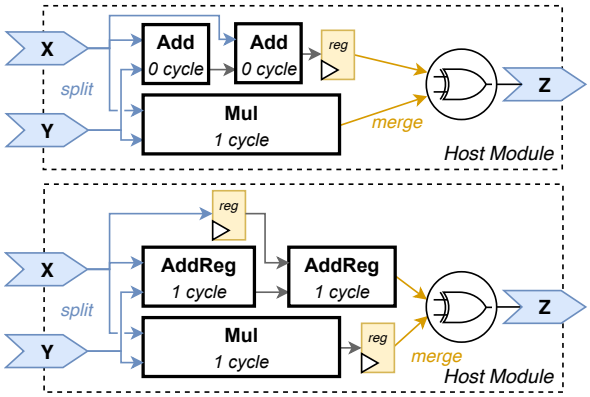


Figure 5.1: Potential integration issues with submodule latency evolution

```
module addReg(
    input clock,
    input [5:0] a, input [5:0] b, output [5:0] r
  );
  logic [6:0] res;
  always @(posedge clock) begin
      res <= a + b;
  end
  assign r = res;
endmodule
```

Insertion of such a register requires the modification of no less than 5 non-configurable lines of code but remains quite straightforward with this single operation. However, the impact increases once the module is integrated in larger designs, as latency of the module is increased by 1 clock cycle. Figure 5.1 illustrates an integration example in which the expected result is $z = (2x + y) \oplus xy$. Hardware implementation of $2x$ with an adder $(x + x)$ rather than a shift $(x \ll 1)$ is a deliberate choice in order to provide a concise example in which the latency change requires appropriate compensation to preserve the design consistency and correctness. The upper part of Figure 5.1 presents a first version of the design including two instances of the purely combinational `add` module in parallel of a *1-cycle* multiplier. A flattened version of this design, based on operators rather than module instantiation, can be written as follows:

```
  module FlatComputeAdd(
      input clock,
      input [5:0] x, input [5:0] y, output [5:0] z
    );
    logic [5:0] sumXY, sum2XY, mulXY;
    assign sumXY = x + y;
    assign sum2XY = sumXY + x;
    always @(posedge clock) begin
        mulXY <= x * y;
    end
    assign z = sum2XY ^ mulXY;
  endmodule
```

The lower part of Figure 5.1 shows the update required to preserve the correctness of the computation when using `addReg` instead of `add`. Similarly, the flattened version below inserts a register after each sum (1-cycle delay), which calls for compensation of this increased latency with the insertion of two additional registers: `regX` to delay `x` and `regMulXY` to delay `mulXY`.

```
1  module FlatComputeAddReg(
2      input clock,
3      input [5:0] x, input [5:0] y, output [5:0] z
4    );
5    logic [5:0] sumXY, sum2XY, mulXY;
6    logic [5:0] regX, regMulXY;
7    always @(posedge clock) begin
8        sumXY <= x + y;
9        regX <= x;
10       sum2XY <= sumXY + regX;
11       mulXY <= x * y;
12       regMulXY <= mulXY;
13   end
14   assign z = sum2XY ^ regMulXY;
15 endmodule
```

Colored lines highlight differences from the previous `FlatComputeAdd` version: green background for additions and yellow background for modifications. Among the 7 lines of the original module body (lines 5 to 11 included, in between `module();` and `endmodule`) only 4 remain unchanged whereas 3 lines were added and 3 were modified in the updated version, resulting in a 85% rewrite of the original module body.

In practice the designer is left alone in charge of mentally inferring any latency changes as tools do not provide automated computation of this information. These changes might require both local updates—as in these short code snippets—and global updates—throughout the hierarchy from the innermost modules to the top. To preserve design correctness, latency changes must then be manually compensated and propagated still without any additional help from EDA tools.

The verbosity and complexity of required changes highly worsen with protocol signaling usage, i.e. handshake between stages, omitted here for conciseness. From this simple *raw* description—without signals dedicated to control the dataflow—several well-known protocols can be used to provide various control implementation of the same functionality, such as Ready-Valid, Credit-Based, or Carloni [AB18]. Similarly to register insertion, introducing or swapping protocols on an existing hardware design is a laborious and error-prone task.

As a conclusion, this simple design iteration showcases the intricate link between pipeline operations and implementation in standard descriptions. Inserting a register stage or introducing protocol signaling induces a complete local refactoring due to the declarative nature of descriptions and foreshadows probable integration struggles, requiring full designer's attention to reflect any latency change to all related parts of the design. To overcome these fundamental limitations of standard hardware description languages, we aim at providing two major evolutions over standard descriptions:

Latency-Aware Design Providing reflexivity over latency for integration and guaranteeing synchronization within an explicit *Pipeline* framework,

Protocol-Polymorphism More concise and configurable descriptions, providing elaboration-time choice of protocols, which enables extensive reuse of base design blocks over a broad range of applications and hardware targets.

5.2.2 Relations as First-Class Citizen

Both latency and protocol relate to the relation between parts of a design. Identifiable parts are made of consistent signals, i.e. signals on which direct operations are meaningful from a functionality point of view. Such signals are said to be *synchronized* or *in-sync*. In

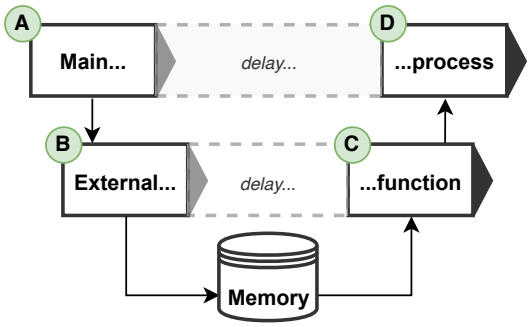


Figure 5.2: Example pipeline requiring delay for signal synchronization

practice, synchronization is fundamental to provide consistent protocol signaling and to propagate signals downstream for later use in the pipeline. Figure 5.2 illustrates a typical example of this synchronization need with a request from a *main process* to an *external function*. The *main process* retains information associated to the request which is required to process the response from the *external function*. Based on an explicit pipeline-oriented representation, this example highlights the introduction of delays, required to keep signals synchronized with respective responses of external elements before proceeding to the next stages. In particular the *external function* relies on a memory, further extending the required delay within the *main process*. Keeping signals synchronized by manually specifying and propagating those delays is a considerable burden and a source of error.

In that respect, a signal synchronization model based on latency information can provide the basis for automatic inference of those delays with simple strategies. For constant-latency operations, a shift-register of the equivalent depth is a solution to keep signals synchronized, while a FIFO with a depth equivalent to the maximum expected latency is required for variable latency.

Modeling of pipelines synchronization has been proposed in the literature from a low-level point of view, down to the synthesis flow to netlist for retiming purposes [LS91] or down to technology-dependent perspectives [SMBD93]. Based on Leiserson's [LS91] graph approach for retiming, we introduce a graph representation of the pipeline as a support to illustrate how data signals relate to each other. Our approach features protocol and latency, defined as the number of cycles or number of register stages. Our focus on cycle latency largely differs from the original study—based on combinational latency—which aims at minimizing clock periods by moving some registers to cut the longest combinational paths. We instead aim at enabling insertion of appropriate delays and protocol signals, omitted from the user description, and provided as backend transformations. To that extend, we retain the following definitions to establish our signal synchronization model:

Signal At a given point of a circuit, a signal is the carrier of electrical variations which encode some pieces of information. Within a circuit, a signal is known by its *unique* name, it retains *no state* and its sole property is its *type*, e.g. simple boolean or complex structured data types. A signal has a single driver (source) and might drive one or multiple components of the circuit (sinks).

TimeZone A *TimeZone* groups signals which are fully synchronized together. It means that direct operations on these signals is sensible from a functionality perspective, without requiring any delay or protocol synchronization. All those signals share the exact same protocol signals, e.g. validity and back-pressure signals.

Relation A *Relation* represents the synchronization requirements between two given *TimeZones*. To express the synchronization, a *Relation* defines two properties: latency

model and protocol signaling. A *Relation* does not always imply an actual hardware connection between signals nor does it convey the actual operational logic actually connecting the *Signals*. In particular, an equivalent *Relation* can be computed between any couple of *TimeZones* as soon as there are actual successive *Relations* from the source *TimeZone* to the sink *TimeZone*. Synchronization resolution is based on this ability to compute an equivalent *Relation*, and will be further detailed in the next section.

Step A step—or *PipeStep* within a *Pipeline*—is an oriented functional transition between a source *TimeZone* and a sink *TimeZone*. It defines both the *Relation* between the two *TimeZones* and the actual hardware logic connecting signals of each *TimeZones*. A *PipeStep* can use any signal available in its source *TimeZone* and might declare new signals in its sink *TimeZone*.

Pipeline A pipeline is a functionality-oriented description of streaming hardware operations, split into steps. A pipeline consists in a collection of *PipeSteps* and their associated *TimeZones*, with sink *TimeZones* of *PipeSteps* being the source *TimeZones* of other *PipeSteps*. Multiple *PipeSteps* sharing a same source *TimeZone* represent a *split* of the pipeline between distinct *branches*. On the contrary, multiple *PipeSteps* sharing a same sink *TimeZone* represent a *merge* of pipeline *branches*.

Synchronization Model The synchronization model consists of a Directed Acyclic Graph (DAG) composed of *TimeZones* as nodes and *Relations* as edges. Obtained from the pipeline description, this graph is used as intermediate representation of the hardware and is aimed at being solved to provide a fully-synchronized circuit.

From a pipeline point of view, any delayed version of a given signal retains the same name and type. As a corollary, any reference by name to a given signal at any stage of the pipeline can be provided as an appropriately delayed version of the original source. To guarantee this property and conform with the *single driven* definition of a signal, the pipeline is expected to be expressed in a Single-Static-Assignment (SSA) form, in which each signal is defined and connected only once at a given *PipeStep*, and can then be used multiple times downstream in the pipeline.

Within this model, the pipeline description does not retain any implementation details, such as transfer protocol from one stage to the next. These critical implementation details often differ from one application context to the next and must be omitted in the pipeline description to allow the generation of a wide variety of hardware based on the same functional description.

5.2.3 Circuit Resolution Process

Based on these definitions and in order to provide latency-aware and protocol-polymorphic designs, synchronization of the circuit can be achieved with the following 3-stage process:

1. **Model Construction** From an *ad-hoc* pipeline-oriented description of the circuit, this first step produces a synchronization model made of *TimeZones* and *Relations*. Construction principles and implementation are further described in Section 5.3.

2. **Synchronization Resolution** Iterating on the synchronization model as an intermediate representation, this next step is dedicated to listing all additional hardware and connections required to achieve complete circuit synchronization. Both compensating delays and protocol signals are specified in this list. Various resolution strategies and their implementation are detailed in Section 5.4.

3. Circuit Update Based on the results of synchronization, this final step consists in updating the actual circuit description, inserting the additional hardware and connections throughout the circuit.

Some of these steps might get implemented together, most notably hardware can be updated directly during resolution, however this clear separation insists on the complete independence of the synchronization model and its resolution from any framework or hardware description language.

5.3 Model Construction

The goal of this front-end step is to build the synchronization model which ultimately aims at providing automated signal delaying and protocol-polymorphism in a multi-branch pipeline context. As a logical starting point, a pipeline-oriented description is designed to give enough degrees of freedom to both make the description more configurable and the synchronization resolution relevant. The key challenge here consists in setting the appropriate level of implementation constraints fixed by the designer in the pipeline description. On the one hand, each omitted piece of information becomes a potential configuration. On the other hand, a minimal subset is still required to guarantee a purely deterministic generation, based on parameters and not on explorative heuristics.

5.3.1 Pipeline Design Methodology

To set this appropriate level of description, and with regard to the code tangling analysis from the previous section, we review the tight link between operations (*i.e.* functional intent) and architecture description. The following coarse-grained classification is an attempt to define the boundaries between these two concerns, based on usual operation and implementation matters of stream applications.

Operation type *What?*	Implementation *How?*
Arithmetic, logic	Signal forward/bypass
Branching	Protocol
States	Synchronization
Correctness	Cycle-accuracy
Register Stages (for performance)	Register Stages (for buffering & synchronization)

Register stages take a prime position for both concerns. On the operative side, as part of the design intent and structure, they provide local memory and are essentials to meet timing requirements. This classification retains the *pipelining for performance* on the user-intent side, as a critical design decision which is not to be automated or inferred.

On the implementation side, register stages are used as buffers and synchronization elements to maintain consistency between data and control signals originating from various places of the design. A typical use-case is a pipeline split in several branches which after some respective operations, get merged again: the *split/merge* pattern, previously illustrated by Figures 5.1 and 5.2. Synchronization of such signals is a well-known source of issues in complex pipelines.

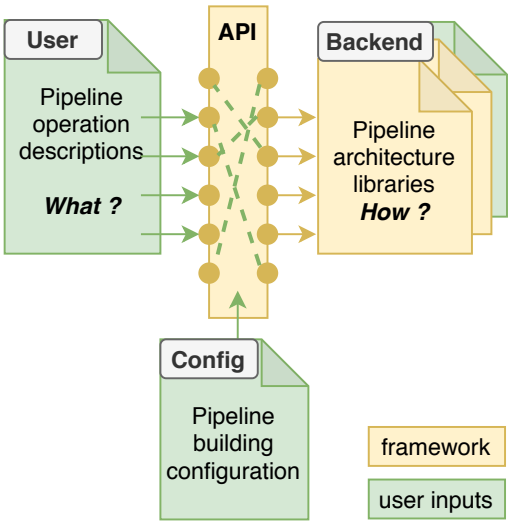


Figure 5.3: Decoupled pipeline description methodology

To improve the reusability of the pipeline descriptions, we aim at reducing the weight of implementation within the operational description while providing advanced parameterization to retain full control over the generated circuit, with a particular focus on synchronization and protocol management.

To that extent, Figure 5.3 illustrates our pipeline description methodology which introduces a triple-sided Application Programming Interface (API):

User Pipeline structure and operations are user-described as a dedicated set of classes and objects, based on a *user-facing API* defined by the framework. This API provides explicit pipeline structure creation methods,

Backend Pipelines macro-architectures implement a *backend API* as another set of classes and objects, most notably for protocols and signal propagation strategies. These macro-architectures are intended to be as generic as possible and while usual ones are included as part of the framework, users retain the ability to develop their own libraries,

Config An elaboration-time *configuration API* provides fine-grained configuration abilities in the mapping between *user-facing API* and *backend API*. It drives the actual code generation from classes and objects available in user and backend APIs.

This entire decoupled methodology is built around the signal synchronization model. It is the key to bridge the gap between user-API—describing signal interactions—and backend-API—in charge of guaranteeing their synchronizations. Section 5.4 details the algorithms used to provide signal synchronization on the back-end side while we focus first on the implementation of this 3-sided API as a pipeline-oriented hardware generation framework.

5.3.2 Pipeline Construction Framework Overview

The implementation of our methodology requires providing 1. hardware primitives to describe signals and pipeline stages, 2. hardware modeling and configurable generation capabilities. Traditional HDLs perfectly fulfill the first criterion but lack reflexivity and programming paradigms support to conveniently implement hardware modeling and generation. In line with our previous experimentations, we leverage the generation power of HCLs to implement our framework as a Chisel library. The following code excerpts

are hence presented in Scala language, including hardware primitives provided by *chisel3* library such as `UInt` type or `Module` definition.

To distinctly materialize the pipeline as an object, we describe it as a sequence of transition functions —*Relations*: edges of the graph—between implicit states—*TimeZones*: nodes of the graph.

The application of this design pattern to the motivating example of Section 5.2.1 is presented on Figure 5.4. This pipeline-oriented version is equivalent to the minified Verilog module `FlatComputeAddReg`, which inserts a register after each adder.

This code excerpt introduces the main objects and classes provided by the user API. To ease code apprehension, code background is colored with the following correspondences: 1. orange for explicit operation on the pipeline object (`init`, `split`, `merge`, `build`), 2. green for operations (*PipeSteps*) on the main (default) branch of the pipeline, and 3. yellow for *PipeSteps* on the explicit `mul` branch.

The concrete class `Pipe` is the core of this API. Its instantiation at line 5 specifies the source relation (`in`) of the pipeline, with type `XYZone` whose declaration—omitted for conciseness—simply provides signals `x` and `y` with their respective types. These two initial signals are then available to all downstream operations.

The implementation of `Pipe` objects provides a default branch—*main*—on which operations are successively recorded as explicit functions. To implement the example, three of such functions (*PipeSteps*) are declared on the main branch, on lines 13–16 (first adder), 17–20 (second adder) and 23–26 (final xor). Each *PipeStep* consists of an anonymous function taking two *TimeZones* as arguments, here `p` for previous (source) and `n` for next (sink). In the function body at line 14, the `Step.Reg` function describes the creation of a new signal `sumXY` in *TimeZone* `n`, with a *Relation* from *TimeZone* `p` of 1-cycle latency, equivalent to a usual register stage. Actual connection of this signal, as a result of the addition of `x` and `y` signals, is described at line 15. Similarly, at line 24, in the function body of the final xor operation, the `Step.Wire` function describes the creation of a new signal `z` in *TimeZone* `n`, with a *Relation* from *TimeZone* `p` of 0-cycle latency, equivalent to a usual wire—i.e. combinational—stage. *TimeZones* `p` and `n` are arguments of the *PipeStep* functions, and not yet actual *TimeZones* of the resulting pipeline. Creation of the actual *TimeZones* occurs during pipeline elaboration.

The `Pipe` object also provides explicit pipeline-oriented operations such as creation of new branches, split from the main branch at its current state, i.e. after any previously recorded operation. That is why the split towards branch `mul` occurs on line 7, splitting from the very beginning of the pipeline and creating a parallel branch. Lines 8–11 perform an operation on this new branch, described in the same manner as the main `Pipe` operations. Explicit merge of this branch into the main pipeline occurs on line 22. A last operation, for which signals from both branches are now available, occurs on lines 23–26.

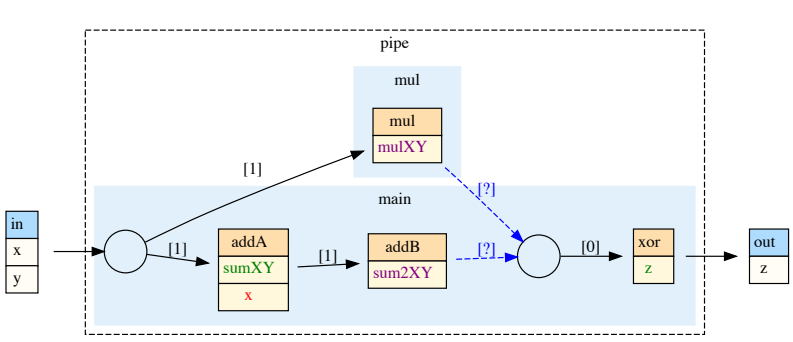
To return the computation result, on line 27, the `Pipe` instance declares connection to external *TimeZone* at its current state: here, signal `n.z` declared in last step `pipe.xor` is connected to sink *Relation* `out`. Underlying type of this relation `ZZone`—omitted here—simply consists of the singleton signal `z`.

Finally, line 28 triggers the pipeline elaboration and actual build of the synchronization model, based on all the operations previously recorded from initialization to operations on respective branches to final feed. Actual *TimeZones* objects are created and passed as arguments of the *PipeStep* functions to record signal creation and usage within the synchronization model.

At this stage, the pipeline description focuses exclusively on the design intent: signal declarations, connections and operations between these signals. Some register stages have been explicitly defined after each adder and multiplier, based on prior knowledge of target performance and application needs. However, there are no mention of protocol signaling

```
class FlatComputeAddReg(
    in: SourceRelation[TZ[_, XYZone]],
    out: SinkRelation[TZ[_, ZZone]]
) extends Module {
  val pipe = Pipe(in, "pipe")
  // explicitly defining branch "mul"
  val mul = pipe.split("mul")
  mul.mul = (p, n) => {
    n.mulXY = Step.Reg(p.x)
    n.mulXY := p.x * p.y
  }
  // 2 adder stages on main branch
  pipe.addA = (p, n) => {
    n.sumXY = Step.Reg(p.x)
    n.sumXY := p.x + p.y
  }
  pipe.addB = (p, n) => {
    n.sum2XY = Step.Reg(p.sumXY)
    n.sum2XY := p.x + p.sumXY
  }
  // explicit branch merge and final xor
  pipe.merge(mul)
  pipe.xor = (p, n) => {
    n.z = Step.Continue(p.sum2XY)
    n.z := p.sum2XY ^ p.mulXY
  }
  pipe.feed(out)
  pipe.build
}
```

Figure 5.4: Running example implemented with our pipeline-oriented framework based on Chisel HCL



(a) Graph representation corresponding to Figure 5.4

Signal Status Color Scheme	
1	Pending: Local
2	Pending: Downstream
3	Pending: Local & Downstream
4	Available: Unused
5	Available: Local Use
6	Available: Downstream Use (Forward)
7	Available: Local & Downstream Uses
8	Declared: Unused
9	Declared: Local Use
10	Declared: Downstream Use
11	Declared: Local & Downstream Uses

(b) Signal color scheme

Figure 5.5: Design intent

associated to any operation, and no register compensation, in particular to guarantee proper synchronization between `main` and `mul` branches. Figure 5.5a illustrates the corresponding design intent, with explicit *split/merge* operations on two branches of the pipeline, while some relations are intentionally left unspecified. Edge labels illustrate respective *Relation* latencies, either unknown *[?]* or explicitly defined here as registers *[1]* or simple wires *[0]*. Each *PipeStep* results in a *Relation* towards a node—its output `n` *TimeZone*—labelled with its name as table heading. The node then lists all signals either defined by the upstream *PipeStep* or required by the downstream *PipeStep*. A color code, presented in Figure 5.5b, provides further insight on the current status of each signal in each *TimeZone*. Here signal `sumXY`, which is defined by `addA` *PipeStep* and is immediately used by `addB` *PipeStep*, is colored in green corresponding to status 9: *Declared: Local use*. Signal `x`, which is also used by `addB` *PipeStep* but is not provided by `addA` *PipeStep*, is colored in red corresponding to status 1: *Pending: Local*.

As a result, Figure 5.5a illustrates the base synchronization needed to convert this simple design intent into a working circuit: delayed connection of signal `x` and balancing of branches `main` and `mul`.

The pipeline description presented in Figure 5.4 is a hardware generator which requires parameterization to generate a circuit, here with `in` and `out` *TimeZones*. This elaboration-time parameterization enables designers to generate various circuits based on the same design intent and is leveraged here to provide two degrees of freedom: 1. protocol polymorphism, and 2. extra signal propagation. Protocol polymorphism is available as part of the relation definition while extra signal propagation is based on the ability to extend the surrounding `in` and `out` *TimeZones* for additional signal synchronization. The following code excerpt illustrates the generation of the circuit in its minimal form, without protocol signaling (`RawIO`).

```
1   new FlatComputeAddReg(
2     RawIOFrom(new XYZone),
3     RawIOTowards(new ZZone)
4   )
```

Generating a different circuit based on ready/valid handshakes between each TimeZone requires swapping `RawIOFrom` and `RawIOTowards` for, respectively, `ReadyValidIOFrom` and `ReadyValidIOTowards`. The following code excerpt illustrates this protocol swap and also introduces the extra signal propagation by extending both `XYZone` and `ZZone` TimeZones with signal `e`.

```
1  trait Extra {
2    val e = UInt(5.W)
3  }
4  new FlatComputeAddReg(
5    ReadyValidIOFrom(new XYZone with Extra),
6    ReadyValidIOTowards(new ZZone with Extra)
7  )
```

Figure 5.6 illustrates the corresponding generation intent, with the introduction of signal `e`. Protocol signals, namely *ready* and *valid*, are automatically inserted at both input and output and are intentionally omitted on the graph representation as they are already implicitly part of the *Relations* and *TimeZones*.

Both changes are very concise, yet yielding very different circuits without requiring any modification of the original pipeline description.

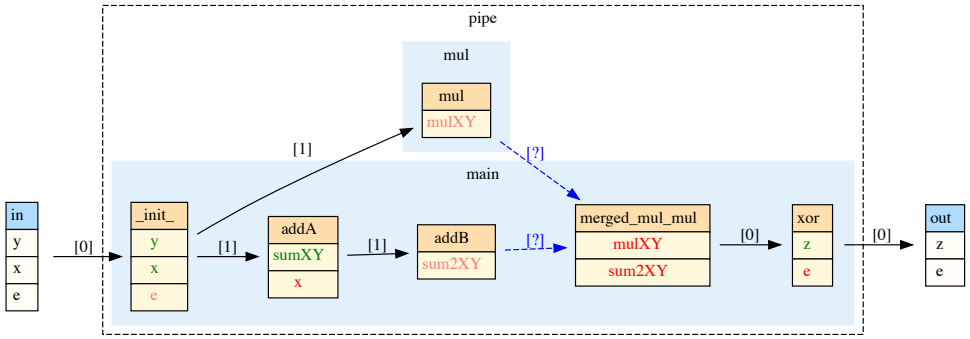


Figure 5.6: Generation intent with additional signal `e`

Similarly, the pipeline description itself is resilient to local changes without requiring remote manual compensations as previously illustrated with the register insertion example in Section 5.2.1. To toggle register stage insertion off, the `Step.Reg` function, used to define values `sumXY` and `sum2XY`, respectively on lines 14 and 18, can be replaced by the `Step.Wire` function. The version with `Step.Reg` and `RawIO` is equivalent to the Verilog module `FlatComputeAddReg`, while the `Step.Wire` version is equivalent to the Verilog module `FlatComputeAdd`. These functions define appropriate relations between signals, allowing automatic latency propagation, path balancing and latency-awareness for surrounding descriptions. Instead of complete refactoring for each use case, the module is now configurable by protocol and pipeline stages can be converted from register to wire with very little and local-only effort.

The algorithms leveraged to provide these advanced generation features, based on the resolution of this elaborated synchronization model, are detailed in the next section.

5.4 Model Resolution: Signal Synchronization

As the pipeline construction is based on partial descriptions, it contains intentional omissions aimed at providing parameterization as an afterthought or within evolving applicative contexts. The incomplete description and its associated synchronization model are then relying on parameterized resolution strategies to solve the synchronization and complete the circuit accordingly. To introduce the resolution principles of the synchronization model obtained from the pipeline description, we first explore resolution of the pipeline request presented on Figure 5.2, and then detail various signal synchronization algorithms and their application to our example.

5.4.1 Base Principles

Given a *pipeline operation description* expressing the design intent of Figure 5.2, the model construction produces an initial incomplete and unbalanced version of the graph representation as showcased in Figure 5.7.a. Letters *A*, *B*, *C* and *D* correspond to the four main *TimeZones* of this example. Edges illustrate *Relations* either unknown *[?]* or with labels displaying their latencies, which are either constant integers *[n]* or ranges between minimum and maximum latencies *[min:max]*. Latency ranges shall be bound to allow the generation of strictly latency-equivalent hardware, which is a key requirement in our applicative case to ultimately guarantee worst-case performance.

From this point onward, a successful synchronization consists in producing a balanced

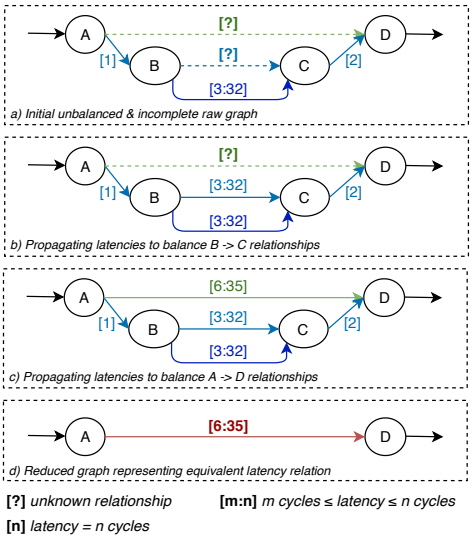


Figure 5.7: Graph corresponding to Figure 5.2 and its transformation

graph without any relation left unspecified. This task is achieved by a set of transformations, described in the *pipeline architecture libraries* and set up by the *pipeline builder configuration*:

1. A first analysis computes the equivalent latencies for each path, in particular following independent branches from their split to their merge points,

2. Based on signal usage analysis, the main synchronization transformation creates latency-equivalent hardware in place of equivalent relations as illustrated in Figures 5.7.b and 5.7.c,

3. Then, a transformation is in charge of propagating and connecting protocol signals uniformly across the balanced graph,

4. A final pass validates overall synchronization consistency, providing early feedback to the designer.

Based on this initial setup, transformations can be further edited and extended depending on specific requirements, development principles or target specificity.

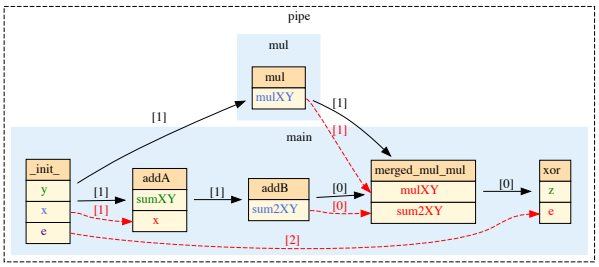
Finally, Figure 5.7.d reveals the equivalent relation of the entire balanced path between A and D. It contains all the details required to delay a signal around this whole block in a larger integration.

Following these steps, synchronization of the graph consists of hardware additions in the circuit from which the graph was originally inferred.

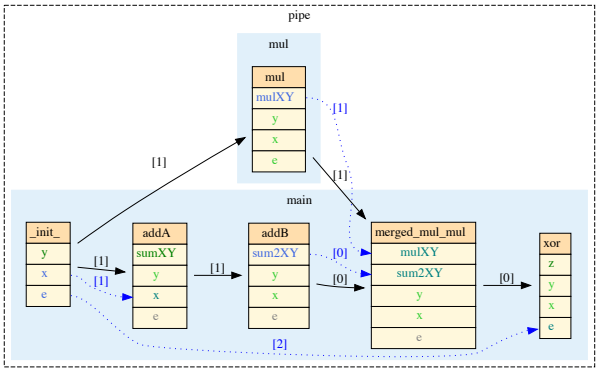
5.4.2 Resolution Strategies

The previous resolution principles outline the coarse-grained steps required to synchronize *TimeZones*. In practice, a finer-grained approach, based on signal availability and usage is required to provide various relevant synchronization algorithms and strategies. Following our running example, this section exhibits some strategies that can be used to solve the synchronization, from naive implementations to fully-configurable algorithms.

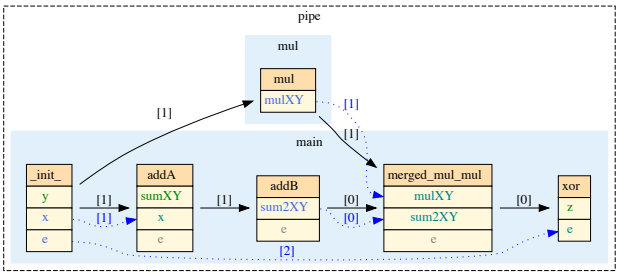
Pre-requisite The algorithms presented below assume the availability of some base operations on the graph. First of all, the graph is expected to be a Directed Acyclic Graph (DAG), i.e. edges (*Relation*) are oriented and there are no cycles in the graph. Such cycles would



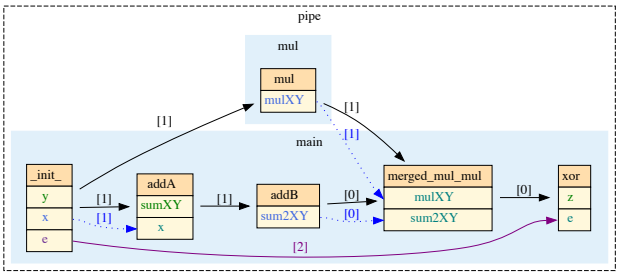
(a) Merge resolution, missing relations & no signal propagation



(b) Exhaustive forward propagation strategy



(c) Peer-to-Peer propagation strategy



(d) Direct propagation strategy

Figure 5.8: Resolution steps and options for the running example

have no meaning in our applicative context, as they would defeat the ability to provide throughput guarantees. Nodes can hence be ordered, and the synchronization model can provide a forward path among nodes (*TimeZone*) from a unique input to one or several outputs. Similarly, this order can be reversed to provide a meaningful backward path from outputs to input. In practice, the forward path is created with the following traversal: from input *TimeZone* on main branch, proceed with the next *TimeZone* until a split, in which case proceed with external branches first, then continue on the initial main branch. For our running example forward order is the following: 1. *_init_*, 2. *mul* (on branch *mul*), 3. *addA*, 4. *addB*, 5. *merged_mul_mul*, 6. *xor*. The backward path order is simply the reversed version of this list. Such traversal strategies ensure that all *TimeZone* are processed in a consistent order in terms of signal declaration and usage: the forward order ensures signal declarations to be processed before any usage whereas backward order ensures all usage to be inspected prior to the declaration, which is convenient for single-pass signal propagation algorithms.

Graph edges (*Relations*) can be composed to form equivalent relations as presented in the resolution principles and in Figure 5.7. Each signal within each *TimeZone* retains an availability property, representing its usage and connection, as illustrated with corresponding colors on Figure 5.5. With these two last properties, *equivalent missing relations* can be defined from a *TimeZone* where a signal is available to a *TimeZone* where it is used. Figure 5.8a shows the graph representation, still unresolved and ready for signal propagation, of the running example.

For the sake of conciseness, the pseudocode algorithms depicted below to illustrate strategies use branches as inputs instead of the full graph, hence not detailing split and merge management.

Algorithm 1: Naive approach: exhaustive forward

```
1  DeclaredSignals := []
2  foreach TimeZone in Branch do
3      foreach Signal in TimeZone do
4          if Signal not in DeclaredSignals then
5              DeclaredSignals += Signal
6      foreach Signal in DeclaredSignals do
7          if Signal not in TimeZone then
8              TimeZone.addAndConnect(Signal)
```

Exhaustive Forward Signal Propagation Algorithm 1 details this first naive approach, which consists of forwarding all signals from *TimeZone* to *TimeZone*, indiscriminately of actual downstream usage. The number of signals forwarded along the pipeline grows linearly stage after stage and a lot of unused hardware is generated. Figure 5.8b illustrates this phenomenon, resulting in several available but unused signals in most *TimeZones*. While synthesis tools might be able to partially remove this *dead-code* in simple cases, such as unconnected wires and registers, useless signals won't necessarily be stripped out if they are stored in a memory, e.g. in a FIFO explicitly implemented with a vendor primitive. This first naive strategy is reviewed here mainly to demonstrate the ease of providing automated signal propagation while highlighting the importance of providing a relevant implementation to avoid poor results.

Algorithm 2: Backward Peer-to-Peer approach:
Requesting unavailable signals to previous TimeZone

```
1  MissingSignals := []
2  foreach TimeZone in reversed Branch do
3      foreach Signal in MissingSignals do
4          if Signal not in TimeZone then
5              TimeZone.createPlaceHolder(Signal)
6          TimeZone.createConnectionTo(Signal)
7          if Signal.isAvailable(TimeZone) then
8              MissingSignals -= Signal
9          else
10             MissingSignals.updateTarget(Signal)
11     foreach Signal in TimeZone do
12         if not Signal.isAvailable then
13             MissingSignals += Signal
```

Peer-to-Peer Backward Propagation Backward graph traversal has a key advantage for signal propagation: inspecting missing signals as they are used, the resolution consists in fetching them upstream in the pipeline. The difference between both algorithms lies on the source selection for the connection. Algorithm 2 presents a first approach which makes no exhaustive search of an appropriate source. With this backward *peer-to-peer* approach, unavailable signals are simply requested to upstream *TimeZones*. For conciseness in the

algorithm presentation, *Signal* objects are to be understood as by-name references to fully-qualified hardware signals associated to their respective *TimeZones*. If the current *TimeZone* cannot provide some pending requested signals, requests are forwarded to the closest upstream *TimeZone*, i.e. the next to be processed by the backward traversal of the graph. In practice, a placeholder signal is created within the current *TimeZone*, this placeholder is connected to the signal placeholder from the downstream requesting *TimeZone*, and finally the request is updated with actual reference to the newly created placeholder. This is the role of *updateTarget* function on line 10. Figure 5.8c illustrates the resolution of the running example with this strategy. Previously missing relations (red dashed arrows in Figure 5.8a) have been implemented through the existing relations between *TimeZones* and now appear as blue dotted arrows.

Algorithm 3: Backward Direct propagation:
Requesting unavailable signals upstream

```
1  MissingSignals := []
2  foreach TimeZone in reversed Branch do
3      foreach Signal in TimeZone do
4          if Signal in MissingSignals then
5              TimeZone.createConnectionTo(Signal)
6              if Signal.isAvailable then
7                  MissingSignals −= Signal
8              else
9                  MissingSignals.updateTarget(Signal)
10         else if not Signal.isAvailable then
11             MissingSignals += Signal
```

Direct Backward Propagation Algorithm 3 presents a second backward strategy which directly requests connections from the closest *TimeZone* where the missing signal is available. There are two upstream availability cases: either signal declaration itself or simply any upstream usage. The main difference compared to the previous strategy is the suppression of local placeholder mechanism for signal propagation in favor of direct connections. Figure 5.8d highlights this difference, as the signal `e` no longer appears in *addA*, *addB* and *merged_mul_mul TimeZones*. Moreover, the missing relation between *_init_* and *xor TimeZones* is replaced by a concrete relation (plain purple arrow) with a latency of 2.

Hardware Primitive Selection Implementation of direct relations between two *TimeZones*, as in the previous strategy, raises the issue of which hardware primitive to pick for a given delay. While some correspondences are obvious, other offer an interesting parameterization potential to finely tune the implementation depending on resource availability. As example, we propose a parameterization based on a single integer parameter strategy called *fifoThreshold*. Variable latency paths, which only appear under certain external constraints and require appropriate protocol signaling, are always implemented using FIFOs. An implementation principle for constant latencies is proposed as follows:

Latency Value (Cycles)	Primitive
Constant $= 0$	Wire
Constant $= 1$	Register
Constant $<$ *fifoThreshold*	ShiftRegister
Constant $>=$ *fifoThreshold*	FIFO

This strategy can then be further adjusted by taking the width of signals into account, e.g. reserving FIFOs for larger signals.

Towards More Configurable Strategies The three signal propagation strategies aforementioned are very generic and focus exclusively on propagating individual signals. Another potential parameterization lies in the management of multiple signals following an identical relation: should their propagation be implemented as a single concatenated group of bits (e.g. one large FIFO) or as individual signals (e.g. one FIFO per signal). Some target-dependent choices might be relevant here depending on depth of the signal propagation and respective widths of the signals. Thanks to the methodology presented on Figure 5.3, many distinct implementations of the very same pipeline description can be generated with target-dependent strategies. Based on the simple strategies presented above, the next section illustrates the ability to obtain distinct synthesis result with the exact same pipeline description.

5.5 Results

The following results are obtained through usage of the framework in various conditions. They aim at validating the relevance of the framework in the high-speed networking application context, considering the two following criteria:

Architectural Parameterization The choice of a resolution strategy leads to distinct and overall predictable resource usage,

Zero-cost Pipeline Abstraction Compared to an equivalent exhaustive description, a circuit generated with the framework exhibits similar resource usage. Strictly tied to the pipeline description, latency and throughput are identical.

5.5.1 Running Example

Synthesis algorithms leverage inference of hardware patterns with fixed or configurable thresholds. For example in Xilinx FPGA synthesis, Shift Register LUT (SRL) primitives are usually inferred for a signal propagation across at least three registers stages sharing the same write-enable signal. For this very reason, to highlight differences between strategies even with a very minimal example, the `xor` stage is here specified with a delay of 2 (equivalent to 2 successive registers) instead of 0 (combinational wire) as originally specified in Figure 5.4. The total resulting latency of the circuit becomes 4 cycles instead of 2, hence enabling inference of SRLs. The following result table presents post-synthesis resource usage on Xilinx VU9P FPGA, exploring generation parameterization with three protocols and three strategies.

Protocol / Strategy	Total LUT	LLUT	LUTRAM	SRL	FF
No protocol / Peer-to-Peer	28	23	0	5	45
No protocol / Direct ShiftReg	28	23	0	5	45
No protocol / Direct FIFO	43	35	8	0	45
ReadyValid / Peer-to-Peer	27	27	0	0	61
ReadyValid / Direct ShiftReg	36	30	0	6	52
ReadyValid / Direct FIFO	50	42	8	0	48
Uniform ReadyValid / Peer-to-Peer	29	23	0	6	47
Uniform ReadyValid / Direct ShiftReg	29	23	0	6	47
Uniform ReadyValid / Direct FIFO	41	33	8	0	42

The first protocol implementation, namely *No Protocol*, corresponds to the absence of protocol signal (`RawIO`), while *ReadyValid* protocol is implemented in two variations:

ReadyValid Branch-level protocol resolution with conservative implementation of split and merge operations, resulting in noticeable resource usage overhead,

Uniform ReadyValid Overall pipeline-level protocol resolution which requires a single output TimeZone, i.e. a single downstream back-pressure constraint to be applied uniformly throughout the pipeline.

The three strategies correspond to the algorithms introduced in the previous section, with *Direct ShiftReg* and *Direct FIFO* corresponding to two distinct *fifoThreshold* in constant latency implementations, resp. 16 and 3. Despite being largely suboptimal in terms of resource for this very simple example, *Direct FIFO* strategy exhibits the ability to generate very different circuits from the same pipeline description.

The following paragraphs review the synthesis results obtained with these various strategies and protocols.

Expected Equivalent Resource Usage *No Protocol* and *Uniform ReadyValid* protocols exhibit identical resource usage for *peer-to-peer* and *Direct ShiftReg* strategies. This is expected due the systematic inference of successive registers as SRLs. On the contrary, with standard *ReadyValid* protocol, *peer-to-peer* strategy prevents SRL inference by generating independent enable signals for each branch and between each split and merge operation. A contrasting behavior regarding SRL inference can be observed with *Quartus* synthesizer for *Intel/Altera* FPGAs, and consequently leads to different impacts of resolution strategies on resource usage. This phenomenon is better exposed through a larger experimentation, thoroughly detailed in Chapter 7. In any case, both strategies result in distinct structures in the Chisel-generated Verilog description. A hierarchy with a submodule per omitted relation is created in the direct propagation case, whereas peer-to-peer propagation results in a single large module in which all relations are implemented. While the distinct strategies might have no impact on synthesis in this particular case, the hierarchical structure of a design still has a direct impact on debugging. Isolating propagated signals in submodules indeed avoids diluting the user intent in the generated Verilog and improves the readability. Last but not least, the ability to generate equivalent synthesized design with various strategies paves the way to experiment more exotic strategies in direct bypass context, such as the *Direct FIFO* strategy.

Expected Resource Usage Variations The impact of *Direct FIFO* strategy appears clearly with the use of memories (LUTRAMs primitives) instead of registers (FFs) or SRLs. A signaling protocol is always required around a FIFO to drive read and write control signals.

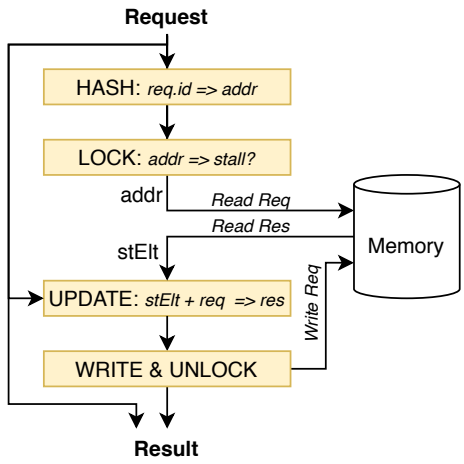


Figure 5.9: Associative memory pipeline block-diagram

In order to use FIFO strategy without protocol and still guarantee the exact expected latency through the FIFO, an additional piece of hardware is inserted. A counter, initially reset to the expected latency through the FIFO, is decremented at each cycle until it reaches and then maintains the value 0. The back-pressure signal, i.e. read enable, of the FIFO is activated only if the counter is equal to 0. Compared to a register-based solution, this mechanism has a considerable impact on resource usage due to both FIFO and counter overheads. By forcing usage of memory primitives in register-oriented context, this convoluted strategy appears to be useful when dealing with local congestion issues on complex design. Local congestion is indeed due to an unbalanced usage of resource, e.g. an area with a very high demand for registers whereas the LUTs remain barely used. Leveraging such an enigmatic strategy can reduce the tension on register by instead implementing the exact same signal propagation with LUTRAMs. Further exploration of this resolution strategy is detailed in Section 7.3.3.

Performance The throughput and latency of the resulting pipeline are intended to be guaranteed by design. The resulting circuit follows exactly the design intent expressed by the designer without any alteration of the stages and explicit delays between *TimeZones*. The resolution process enforced here first precisely balances all concurrent paths—merge resolution—then propagate the required signals with exact latencies. Then protocol signals are implemented and connected as part of the fully synchronous circuit. Following this pattern, there are no compromises on performance: FIFOs are always dimensioned for the worst case, as this is a requirement for high-throughput network applications. As a side note, some applicative contexts, which targets average rather than worst-case performance, could reduce the depth of delaying FIFOs based on expected duty cycle. Implementing such strategies would provide a trade-off between resource footprint and performance.

As a conclusion, this example highlights the ability to experiment various synchronization resolution strategies while preserving both original functionality and performance.

5.5.2 Industrial Use Case

The previous simplified examples illustrated the results of the framework, but not its capabilities. To further demonstrate the usefulness and performance of the framework, we apply it to the implementation of the *Probabilistic Stage* of the *Protected Hash-table* earlier

presented in Section 4.3. Figure 5.9 illustrates the main pipeline stages of this associative storage module, which is part of an industrial network processing device.

To set the ground for fair comparison, we first implement this module with vanilla Chisel, i.e. without using the framework. To highlight the upcoming impact of the framework on the pipeline management, this first version follows a segmented approach: 1. an independent memory module provides read and write methods with locking capabilities, 2. an abstract data type scheme is implemented in order for the pipeline implementation to only handle scheduling and synchronization. These initial choices make the pipeline stages appear clearly and separately from data computations. This early separation principle confirmed that the issues we encountered to validate the module came mostly from timing and synchronization management. One of these issues came from the need to accommodate the application on two FPGA targets, one of them requiring additional registers after memory read to meet timing requirements. While being trivial, this difference implies adaptation of all synchronized paths to maintain performance. In particular in this example, Figure 5.9 highlights the need for synchronization of the input request, used in several stages which depend on the memory read response. Many other signals throughout the design faced similar synchronization issues, not detailed here for the sake of clarity.

The following code excerpt present the outline of the upgraded version leveraging the framework.

```
1  val mem = new MemoryInterface()
2  val pipe = new Pipe(req, "main")
3  pipe.hash = (p, n) => {
4    n.addr = HashPipe(p.req)
5  }
6  pipe.read = (p, n) => {
7    n.stElt = mem.readLock(p.addr, p.req.doLock)
8  }
9  pipe.update = (p, n) => {
10   (n.res, n.updatedStElt) = p.stElt.update(p.req)
11 }
12 pipe.feed(mem.writeBack, res)
13 pipe.build
```

This streamlined version illustrates the *PipeSteps* presented in Figure 5.9 while only evading most parameterization and some verbose explicit type-cast temporarily[1] required by Scala and Chisel. It allows designers to focus on expression of pipeline stage behaviors rather than verbose and error-prone signal synchronizations. This statement was confirmed by experiment as no synchronization fixes were required during the development of this version.

Complete version of this description generates a resulting Verilog code quite close to the one previously obtained without the framework, which passes all the existing validation tests and exhibits similar throughput and latency. As the framework leaves full control of the pipeline structure to the designer and does not alter it during the synchronization transformations, identical latency is indeed expected by construction, and has been confirmed experimentally. The proper throughput confirms that the dimensioning of synchronization delays is sufficient to reach the expected performance.

With these observations on this simplified example, the methodology and the framework implementation prove their relevance to improve description flexibility while achieving similar performance. However, the potential overhead, induced by the framework and

[1] These temporary casts are only due to the current dynamic implementation of our pipeline objects in Scala, which prevents static typing at compile in its current version. This limitation is expected to be lifted with an appropriate integration of the pipeline framework as a compiler plugin.

its automated hardware generation, remains to be asserted. To answer this question both versions have been synthesized in the same conditions, targeting a Xilinx *Virtex Ultrascale+ (VU9P)* FPGA with a 200 MHz clock frequency. The table below shows a resource usage comparison between the original—*Vanilla Chisel*—and the updated version of the pipeline, leveraging the framework in its most simple *Peer-to-peer (P2P)* strategy.

	∑ LUT	LLUT	LUTRAM	SRL	FF	BRAM
Vanilla Chisel	6,077	5,314	88	675	6,793	5
Framework (P2P)	6,063	5,300	88	675	6,743	5
Difference	-14	-14	=	=	-50	=
Difference (%)	-0.2%	-0.3%	=	=	-0.7%	=

Reported resource usage after synthesis of the updated design is on-par with resource usage reported for the original implementation, with a minor advantage in favor of the framework. This slight difference is explained by the packing of delayed signals in the original version whereas the framework treat each signal independently.

A notable side effect is an important reduction in the number of lines of code (excluding comments and blank lines in both cases), between the original description and the updated version, of almost 50% in favor of the framework. While less lines of code do not necessarily mean a more flexible description, as we extracted synchronization and protocol concerns, this simplification was expected on the description side. More importantly, the evicted lines are part of the repetitive and error-prone patterns illustrated in Section 5.2.1, as these concerns are now independently managed by the synchronization backend.

5.6 Conclusion

In this chapter, we have introduced a pipeline design methodology aiming at improving pipeline description flexibility by providing a readily available pipeline functional abstraction. It comes in the form of a fully decoupled API, with on one hand expression of pipeline stages behavior, and on the other hand synchronization strategies and protocol signaling generic implementations. We close the gap between these two interfaces with a graph-based synchronization model and by providing, at elaboration-time, user-access for fine control over the generated design. By leveraging Chisel HCL to implement our methodology as an automation framework, we have successfully migrated a complex existing pipeline at no additional resource costs while considerably relieving developers' synchronization burden. Moreover, decoupling protocol signaling and synchronization concerns from the pipeline description highly improves code readability and reusability while advantageously reducing its volume.

Through this experiment and the ones presented in Chapter 4, Hardware Construction Languages have demonstrated their ability to produce efficient circuits while improving architecture reusability. At this stage, we have focused on the design of several complex architectures, packaged as hardware modules. In our industrial context, these modules are intended to be integrated within larger hardware hierarchies to build network devices and provide high-level network functionalities such as rate-limiting or specific DDoS mitigations. The next chapter discusses the issue related to the integration of these Chisel-generated blocks into existing HDL hierarchies.

Chapter 6

Integration of Hardware Construction Languages

This chapter focuses on integrating Hardware Construction Languages within existing hardware development flows. We claim that the adoption of HCLs, as newcomers in hardware ecosystems, lies in their capacity to integrate themselves effortlessly within existing flows. Contributions in this chapter arise from pure engineering matters, and therefore do not intend to provide a generic solution to integrate HCLs, but rather explore some options for both upstream and downstream integrations.

First, we review the ability to translate existing Verilog projects to Chisel, as the first step of an iterative upgrade flow, and to integrate existing HDL blocks within Chisel.

Then, we detail the integration of Chisel-generated circuits within larger SystemVerilog hierarchies, while preserving the ability to parameterize the module instantiations from SystemVerilog.

Contents

6.1 Problem Statement

Integration of Hardware Construction Languages within an existing flow and legacy codebase, immediately raises three questions depending on the point of view of the observer.

HCL as Top Viewing HCLs as top level hierarchy replacement raises the question of the integration of existing traditional HDL sources as IP within the HCL hierarchy:

- *"How can existing traditional HDL sources be instantiated within HCLs?"*

HCL as IP Viewing HCLs as an external IP generator raises the questions of the interoperability of this generator with the development flow, and of the generated content with the existing code:

- *"How can HCL-generated sources be instantiated within traditional HDLs?"*
- *"How can generation be integrated within the existing flow?"*

The first of these three questions finds a straight answer in most HCLs, which provide the ability to integrate parameterized traditional HDL *black-boxes* in their descriptions, simply emitted as module instantiation with parameters and port maps. This typically works well in Chisel HCL with (System)Verilog *black-boxes* integration instantiated and mapped in the emitted Verilog. While this process enables designers to quickly integrate existing sources within their new HCL-based projects, the interoperability remains restricted to the underlying emission language capabilities. In particular, it requires maintaining interface definition on both sides and maintaining sources in both languages. This is not a disqualifying issue for cohabitation with single-instantiated IP blocks, however it becomes rather constraining for integration of multiple instances of the same module with various parameterizations, as further detailed in Section 6.3.

The left side of Figure 6.1 illustrates the integration of such existing traditional HDL IP blocks within an HCL top level hierarchy. With this approach, the knowledge (configuration of parameters) can be centralized at the top level of HCL hierarchy and shared to external IP blocks through parameter maps. The generation process is straightforward, with HCL top level generator as the single entry point.

While this first approach is perfectly suited to new projects, retaining the ability to integrate existing IP blocks, it is not an option for large existing projects. Introducing such a massive dependency at top level, requires a complete upgrade of the development flow and a massive amount of work to encapsulate existing blocks as IPs, which is simply not worth it. Advantages of HCLs will indeed not appear until actual *business logic* is designed with HCLs. A top level replacement does not even guarantee in itself the ability to integrate such developments because additional layers of existing HDL hierarchy would also need to be rewritten to reach actual *business logic* layers. While the ability to integrate existing traditional HDL IP blocks is a key functionality for cohabitation between HDLs and HCLs, the *HCL as Top* perspective is not suitable for drop-in top level replacement at scale.

The right side of Figure 6.1 illustrates the alternative point of view, with HCLs as external IP block generators. As generated IP blocks cannot be parameterized from the HDL hierarchy, this approach requires additional synchronization between parameters passed to the generators and the parameter-dependent interfaces expected on the top HDL hierarchy side. While this requirement is constraining and requires custom tool development for smooth integration of HCL as IP, it enables designers to bring the power of HCLs directly where the core *business logic* can take advantage of it. As a result, it showcases the relevance of HCL-based design integration even in large projects where traditional HDL hierarchy development benefits from a strong legacy.

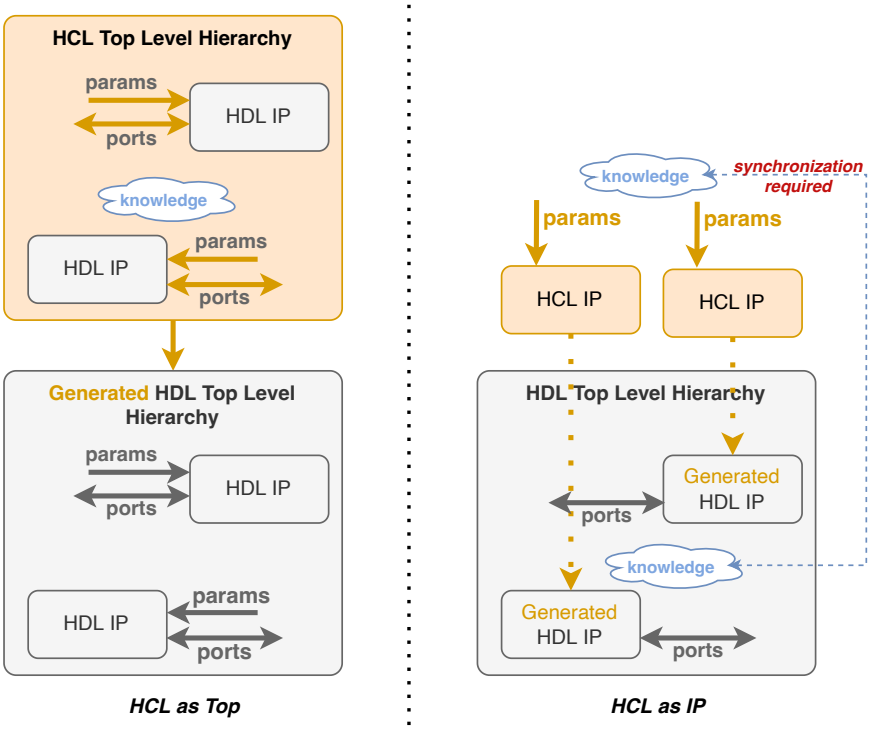


Figure 6.1: HCL integration perspectives

Integration of HCLs in large existing code-bases still raises two main engineering challenges as these code-base cannot simply be rewritten with HCLs over-night. First, beyond the integration of existing HDL IP blocks as *black-boxes*, how can these existing blocks be included in the intended agile upgrade presented in the previous chapters? Secondly, how can the synchronization between generator parameters and expected parameterization from HDL hierarchy be solved?

Previous researches regarding these integration issues have focused on component based integration, such as *Chipyard* [ABG+20], an integrated System-on-Chip (SoC) design, simulation and implementation framework. While *Chipyard* is articulated around Chisel/FIRRTL HCL stack and in particular the rocket-chip RISC-V core generator, *LiteX* [KBBLL19] is a similar framework based on (n)migen [Bou13]. Focusing on open-source cores available as git repositories, *FuseSoC* [Kin19] is another component-based integration system. *ESP* [MGDG+20] extends this compositional approach to integration of IP blocks, ranging HLS generated to higher-level designed IP to custom RTL blocks, within a custom Network-on-Chip grid. These solutions provide the ability to integrate heterogeneous components such as RTL descriptions and HCL generators within a common project, following agile development principles. However, as they introduce their own top level management to integrate components together, they are not a solution for a progressive transition of existing traditional HDL code-bases to HCLs. In addition, their integration of HCL with HDL is restricted to assembling a collection of independent top level blocks together and is not the mixed HDL/HCL cohabitation we are targeting here.

To address these unresolved integration challenges, the next section introduces an automated translation tool to provide an agile transition from existing HDL hierarchies to idiomatic HCL generators. Then, Section 6.3 discusses the integration of Chisel as IP, in particular regarding multiple distinct parameterized IP instantiations.

6.2 (System)Verilog Upstream Integration by Translation

6.2.1 Intended Flow

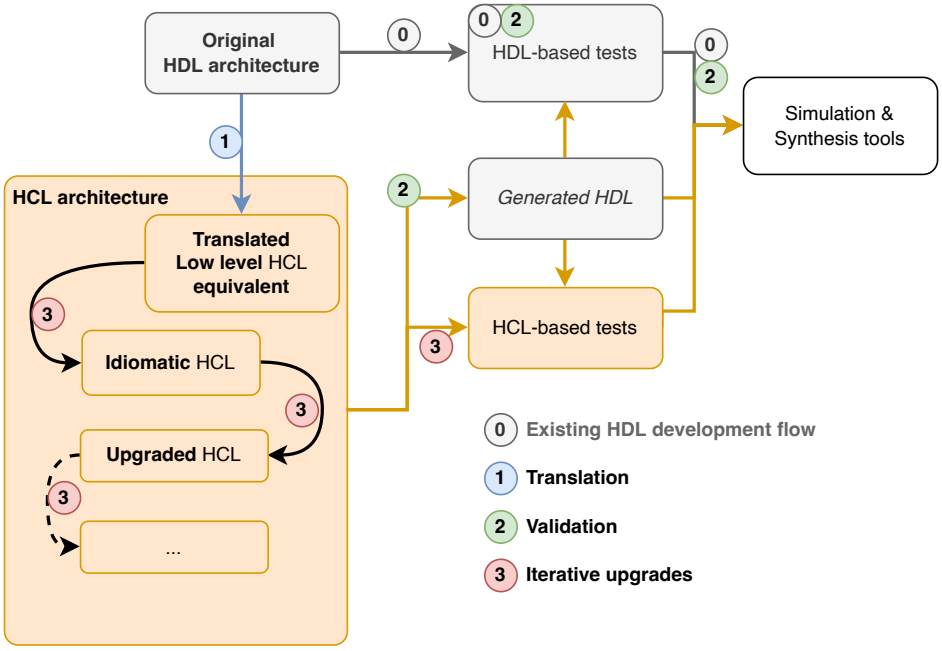


Figure 6.2: Translation process towards agile integration

While HCL hierarchies can integrate existing HDL blocks, their interoperability remains limited to the parameterization abilities and interface flexibility offered by the HDL side, which is admittedly quite low. To push the integration beyond a simple cohabitation and bring all the benefit of HCLs to the existing HDL architectures, the latter should be rewritten within the HCL. To automate this tedious task in the context of Chisel integration within our SystemVerilog hierarchy, this section introduces a (System)Verilog to Chisel translator. This translator is part of a larger agile upgrade flow from HDL architectures to idiomatic HCL architectures, illustrated in Figure 6.2 and articulated as follows from an existing HDL development flow:

Translation Original HDL architecture is translated to a low-level HCL version of the architecture, whose expression is extremely close to the original description. While most of the translation is intended to be automatic, some parts might require manual adjustments due to an absence of a direct equivalence between both languages.

Validation Before further design iterations, the quality of the translated architecture is validated with the pre-existing HDL test-benches.

Idiomatic Adaptation Automated translation of highly optimized and generic architectures often results in verbose HCL generators which can be easily improved by leveraging native HCL generation abilities.

Iterative Upgrade In the direct line of our previous contributions, HCL architectures have demonstrated their ability to be integrated in agile upgrades to extend their functionalities.

Once the initial process is complete, or as soon as the validation is complete, the resulting HCL architecture can be integrated within a larger HCL hierarchy. Then, the complete hierarchy, including this new block, might be iteratively upgraded, for example to benefit from the generation abilities such as functional parameterization.

With these integration objectives clearly stated, we now focus on the implementation of the translation tool and its application on several architectures.

6.2.2 sv2chisel Translator Structure

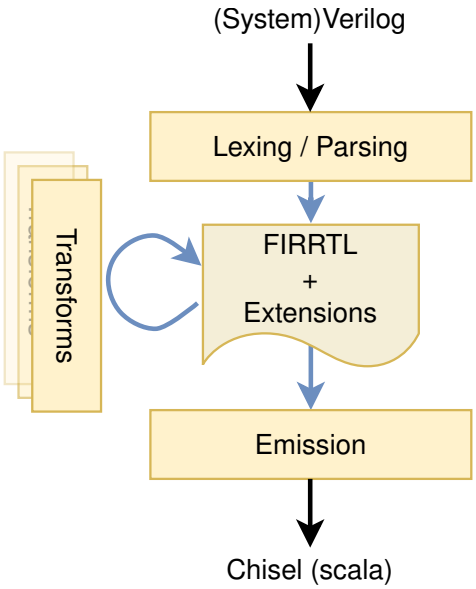


Figure 6.3: Main processing steps of *sv2chisel*

Figure 6.3 presents the structure of our translator *sv2chisel*, with the classical stages of any translator or compiler. We parse a synthesizable (System)Verilog file using ANTLR 4 [PF11], which builds an abstract syntax tree that we eventually map on our custom IR. It is based on FIRRTL which has been extended most notably to support Verilog generative statements and syntactic-sugars requiring a few transformations to be mapped to Chisel constructs. Four kinds of analysis and transformations are then performed on this IR: 1. clock inference, 2. reset inference, 3. types inference, 4. (System)Verilog syntactic-sugar translation.

Finally, the emitter outputs Chisel code, relocating comments and retaining original layout as much as possible. To enable comment relocation, every node of the IR retains its initial position in the token stream of the ANTLR lexer. Transformations then have to carefully take these indexes into account whenever inserting or removing nodes, in order not to disturb comments relocation. To achieve the final relocation, the emission goes through two steps. First, every IR node is converted to its Chisel equivalent string and the IR is flattened as a stream of those Chisel strings, each associated to its position in the original (System)Verilog token stream. Then, this stream is merged, based on the token indexes, with the original token stream which retains comments and spaces in special sub-streams.

6.2.3 Main Transformations

6.2.3.1 Clock Inference

Clock inference is a crucial part in the success of a proper translation from (System)Verilog to Chisel. As a direct consequence of (System)Verilog event-driven paradigm, the distinction between wires and registers does not come from their declaration but from their assignments either in a clocked process or as a continuous assignment outside a clocked process.

```
module clock_example(
    input clock,
    input rst,
    input i,
    output o_w,
    output o_r
  );
  /* behavioral description of a wire */
  logic w;
  assign w = i;
  /* event-driven behavioral description of
   * register r with reset value '0 */
  logic r;
  always @(posedge clock) begin
    if (rst) begin
      r <= '0;
    end else begin
      r <= i;
    end
  end
  /* output connections */
  assign o_r = r;
  assign o_w = w;
endmodule
```

Listing 6.1: Verilog for wire and synchronous-reset register

```
class ImplicitClockOnly() extends Module {
  // implicit input clock, connected to all `Reg`
  val rst = IO( Input  ( Bool() ))
  val i   = IO( Input  ( Bool() ))
  val o_w = IO( Output ( Bool() ))
  val o_r = IO( Output ( Bool() ))

  /* declaration of a wire object */
  val w = Wire(Bool())
  w := i // combinational assignment

  /* declaration of a register object */
  val r = Reg(Bool()) // type only

  when(rst){ // explicit synchronous reset
    r := false.B
  } .otherwise {
    r := i // sequential assignment @(clock)
  }

  /* output connections */
  o_w := w
  o_r := r
}
```

Listing 6.2: Translated Chisel for Listing 6.1 with clock inference

Listing 6.1 illustrates how signals `w` and `r` are to be described in the standard synthesizable (System)Verilog such that a wire is inferred for `w` and a register for `r`.

As a side note, we thoughtfully chose the use of `logic` keyword in our Verilog example for both declaration of `w` and `r`, instead of the misleading `reg` or `wire` keywords. Despite providing those keywords (System)Verilog does not enforce any restriction in their usage, consequently a register can be inferred from a signal declared as `wire` but then sequentially assigned, and conversely a wire can be inferred from a signal declared as `reg` but then combinationally assigned. Some tools raise warnings on such misusage, however the rule remains the same for all of them: behavioral inference is preferred over user `wire` or `reg` hint.

Whereas wire and register are inferred from signal assignment behavior in Verilog, Chisel provides concrete objects `Reg` and `Wire` for signal declaration. These objects are self-sufficient when it comes to assignments: both `Reg` and `Wire` are assigned with the same operator, in the same context as shown on Listing 6.2.

To infer the main clock of a module, *sv2chisel* traverses the IR a first time, looking for `ClockRegion` blocks corresponding to (System)Verilog `always @(posedge clock)` blocks. For each assignment inside such a block, left hand side references are recorded as assigned within a process clocked by `clock`. The declaration of these references will then be converted as a `Reg` object during a second IR traversal that will also remove `ClockRegion` blocks in favor

```
class ExplicitClockExample() extends RawModule {
  val reset = IO(Input(Reset()))
  val clock = IO(Output(Clock()))
  // inputs as shown Listing 6.2

  withClockAndReset(clock, reset){
    // statements as shown Listing 6.2
  }
}
```

Listing 6.3: Explicit clock and reset area in Chisel

of simple collections of statements.

Clock might also be inferred from submodules instantiations whenever the actual module declaration can be processed recursively by *sv2chisel*.

At the end of this transformation, zero, one or several clocks might have been discovered. When zero or one clocks are discovered, the clock might be fully implicit within the module as Listing 6.2 shows. When several distinct clocks have been inferred, no implicit clock is used for the module and an explicit clock area must surround the register declarations of a same clock region, leveraging Chisel syntax `withClockAndReset` as Listing 6.3 introduces.[1]

This syntax might also be required to deal with some constrained board clock and reset naming—for example at design top level. Listing 6.3 illustrates this syntax to explicit the implicit constructs provided by `Module` in the previous examples.

6.2.3.2 Reset Inference

Several kinds of reset methods can be found in Verilog descriptions:

- Synchronous Reset
- Asynchronous Reset
- FPGA Power-on Reset

Synchronous resets are hard to gracefully infer from a design as they might be hidden in deep and intricate `if`/`else` hierarchies [PK16]. For the simplest and most common Verilog pattern we perform synchronous reset inference, leveraging `RegInit` object as shown in Listing 6.4. In more complex cases, the code is translated literally, keeping the conditional assignment tree as shown in Listing 6.2. This is less idiomatic but still perfectly acceptable.

Asynchronous resets, on the other hand, are easily understood with a method equivalent to the one used for clock inference in Section 6.2.3.1, based on the Verilog `always` blocks events `@(posedge(rst))` or `@(negedge(rst))`.

Leveraging power-on resets in an FPGA design can save a lot of hardware resources used by reset trees. However, as Chisel originally targeted ASIC development, these resets were not natively present in Chisel/FIRRTL stack. We contributed to FIRRTL by adding their support as an overload of asynchronous resets, with an annotation of the source reset signal and a FIRRTL transformation to remove the reset tree in favor of declaration of signals with assignment to the specified reset value.[2] This addition allows to straightforwardly translate them to Chisel `RegInit` declarations, with the ability for the user to annotate the top level as power-on reset for equivalent behavior with the Verilog.

[1] Multi-clock modules support is a work in progress at the time of writing.

[2] Pull-request on FIRRTL's GitHub repository: https://github.com/chipsalliance/firrtl/pull/1050

```
class ImplicitClockReset() extends Module {
  // implicit inputs clock & reset
  val i   = IO( Input  ( Bool() ))
  val o_w = IO( Output ( Bool() ))
  val o_r = IO( Output ( Bool() ))
  /* declaration of a register object with reset */
  val r = RegInit(false.B) // reset value and type
  r := i // sequential assignment @(clock)
  /* declaration of a wire object */
  val w = Wire(Bool())
  w := i // combinational assignment
  /* output connections */
  o_w := w
  o_r := r
}
```

Listing 6.4: Translated Chisel for Listing 6.1 with reset inference

At the time of writing the *sv2chisel* support for mixed reset methods within the same design is still a work in progress.

6.2.3.3 Types Inferences

While (System)Verilog requires explicit clocks and reset signals, it is very permissive towards data types: there is no typing or explicit cast in most cases. As a base principle every signal is an array of bits and if it needs to be used in an arithmetic computation, it is automatically cast as an unsigned, unless explicitly specified by the designer with the `$signed` cast.

This is very different in Chisel which comes embedded in the strongly typed language Scala. Chisel has several distinct basic hardware types, such as arrays of hardware boolean (`Vec[Bool]`) and hardware (un)signed integers (`UInt` and `SInt`).

While both Chisel `Vec[Bool]` and `UInt` types would be a valid translation of any (System)Verilog signals, Chisel enforces very constraining rules for the usage of these distinct types. Chisel's `Vec[Bool]` are meant for individual bit manipulations and do not provide arithmetic operations. On the other hand `UInt` are intended for such arithmetic operations and are to be considered as a whole, they hence do not support subrange or subindex assignments.

To provide the most sensible translation—or at least the less verbose—*sv2chisel* counts the uses of each reference in arithmetic or bit operations to choose the proper declaration types.

Then for each expression, appropriate casting is inserted wherever necessary to guarantee type consistency.

Listing 6.5 shows a simple Verilog snippet illustrating the absence of typing: both `sign` and `cnt` signals are declared as packed arrays of bits. However, while `cnt` is used in an arithmetic expression, `sign` is assigned one bit at a time. Listing 6.6 is the result of *sv2chisel* translation for Listing 6.5 and illustrates type inference of the signals with respect to their usage: `cnt` is now declared as a `UInt` while `sign` is declared as `Vec[Bool]`. Moreover, as `sign` is declared as `Vec[Bool]` but used to assign the `UInt` signal `cnt`, an explicit `asUInt` cast is inserted accordingly.

Although *sv2chisel* is designed to produce ready-to-use code, the resulting code is mainly intended to serve as a transition step between (System)Verilog and handcrafted Chisel code, after manual inspection and refactoring. Listing 6.7 illustrates a small code refactoring, using a more Chisel-ish way of expressing the same computation. Instead of this complex iteration on subranges, a more idiomatic expression consists in a simple cast of the 32

```
module type_example #(param en)(
    input  clock,
    input  reset,
    output [31:0] counter,
    output [3:0]  sign
  );
  logic [31:0] cnt;
  always @ (posedge clock) begin
    cnt <= &cnt ? sign : cnt + 1;
  end
  assign counter = cnt;
  for( i=0 ; i < 4 ; i++) begin
    assign sign[i] = en ? ^cnt[(i+1)*8-1:i*8] :
                          '0;
  end
endmodule
```

Listing 6.5: Untyped verilog input

```
class TypeExample(en: Int) extends Module {
  // implicit inputs clock & reset

  val counter = IO( Output ( UInt(32.W)    ))
  val sign    = IO( Output ( Vec(4,Bool()) ))

  val cnt = Reg(UInt(32.W))

  cnt := Mux(cnt.andR =/= 0.U, sign.asUInt, cnt + 1.U)

  counter := cnt
  for(i <- 0 until 4){
    sign(i) := if(en != 0) cnt((i+1)*8,i*8).xorR
               else        false.B
  }
}
```

Listing 6.6: Typed raw Chisel translation of the untyped Verilog input of Listing 6.5

```
class TypeExample(en: Boolean) extends Module {
  /* see statement above */

  val bytes = cnt.asTypeOf(Vec(4, UInt(8.W)))
  sign := if(en) VecInit(bytes.map(_.xorR)) else Zeroes
}
```

Listing 6.7: Chisel for Listing 6.5 after manual refactoring

bits counter `cnt` to the equivalent array of 4 bytes. Then Scala functional paradigm can be leveraged to apply a xor reduction on every byte with the `map` function. The special object `Zeroes` is a custom object that is equivalent to `0.U.asTypeOf(sign)` here.

This kind of refactoring which makes the code clearer cannot be inferred by the tool but greatly helps with code maintainability. It confirms that both automated translation and manual expertise are essential to produce a finely-tuned piece of hardware. The former makes the conversion task realistic even for large code-bases while the latter ensures simplicity and readability of resulting code.

6.2.3.4 Syntactic Sugar Removal

(System)Verilog is also very permissive with connections of signals of mismatching widths. Wherever required, it will implicitly infer left-padding, with sign extension for signed signals and with zeroes otherwise. This automatic padding is then widely used as a feature, enabling some concise syntactic-sugars.

For example, SystemVerilog *bit patterns* `'0` and `'1` are such context-dependent widths. They are equivalent to a signal whose width is inferred and all bits are set to 0, respectively all bits set to 1.

```
  wire [31:0] w;
  assign w = '0;
  assign w = '1;
  assign w = '{4{8b'010111}};
```

When translating to Chisel, actual width is required for all ones bit pattern as follows:

```
val w = Wire(UInt(32.W));
w := 0.U
w := ~0.U(32.W)
w := VecInit.tabulate(4)(_ => "b010111".U(8.W))
```

A second example of (System)Verilog concision is the left-hand-side assignment on a concatenation, which allows straightforward expression of bit unpacking.

```
wire [15:0] a;
wire [7:0]  b;
wire        op;
wire [15:0] sum;
assign {op, b, a} = signal;
assign sum = op ? a + b : a - b
```

Chisel does not offer a word-for-word equivalent, however casting the right-hand-side into a `Bundle` provides an equivalent as shown below:

```
val bdl = Wire(new Bundle {
  val a   = UInt(16.W)
  val b   = UInt(8.W)
  val op  = Bool()
})
val sum = UInt(16.W)
bdl := signal.asTypeOf(bdl)
sum := Mux(bdl.op, bdl.a + bdl.b, bdl.a - bdl.b)
```

Translating (System)Verilog syntactic-sugars into Chisel code is not always as concise as the original code but thanks to the previously introduced type inference system, correct translations are achievable at the cost of a few additional lines. These verbose translations are then good starting points for manual code refactoring, as the power of Scala and Chisel constructs bring functional and object-oriented refactoring opportunities.

6.2.4 Evaluation

6.2.4.1 Methodology

Figure 6.4 illustrates all the steps taken to validate a translated design from its original description to the synthesized design.

The first step is to run the *sv2chisel* tool to translate the original (System)Verilog into Chisel. The tool appears to be a great stateful linter, catching wrongly declared and undeclared signals. Some manual adjustments might be required on the original description to fix caught issues and to overcome the current limitations of the tool as detailed in Section 6.2.4.3.

The resulting Scala must then go through the entire Chisel generation flow, each step coming with its own correctness checks:

1. Scala compilation catches types inconsistencies
2. Chisel elaboration complains on mismatching widths
3. FIRRTL compilation stops on uninitialized references and detects combinational loops

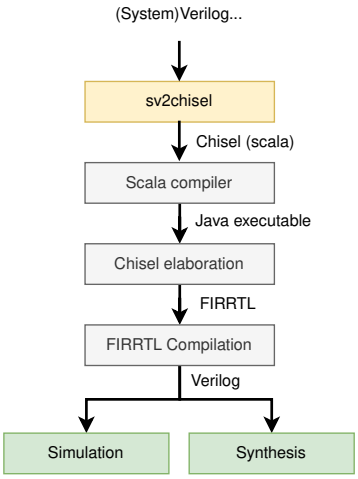


Figure 6.4: Validation process

While the two first steps are expected to pass flawlessly as long as *sv2chisel* does not complain during translation, FIRRTL compilation step raises errors that might only be manually corrected or ignored depending on the intended behavior of the result. Unexpected combinational loops combined with warnings of *sv2chisel* about unsupported blocking assignments might indicate an inaccurate translation and require further inspection of the original source code. Regarding uninitialized references, among classical examples stand big arrays, only half-connected, used to implement binary reduction operations. Such partial initialization can easily be fixed by adding the explicit default connections either in the original source or in the translated one. However, a rather simple manual refactoring operation could leverage Chisel power to implement such reduction operation in a functional way while getting rid of these partially initialized arrays.

To validate the resulting Verilog and hence the correctness of the translation, we integrate it into the existing test system, leveraging usual simulators.

Last but not least, we compare the resource usage after synthesis on FPGAs.

6.2.4.2 Results

We successfully ran our tool on three different projects to guarantee a consistent coverage of usual Verilog synthesizable constructs: 1. An internal module, currently used in production at OVHcloud, 2. A simple MIPS core implementation [Beh17], functional but not actually suited for synthesis, 3. The size-optimized RISC-V Core *PicoRV32* [Wol20].

Translated code for our internal module and the MIPS core passed their respective tests with minor manual modifications of the input source code. To pass PicoRV32 RISC-V core tests, some structural code refactoring was necessary to translate the concept of variable and their blocking assignments, as this feature does not find a straightforward Chisel equivalent.

The table below shows a resource usage comparison between original and translated versions of the two synthesizable projects. The SimpleCPU MIPS Core is indeed not intended for synthesis, which results in similar but inconsistent resource evaluation for both versions.

		Verilog	Chisel
OVHcloud Module	LUTs	1681	+7
	FFs	2733	=
	BRAMs	0	=
	DSPs	0	=
PicoRV32	LUTs	2349	+27
	FFs	1276	=
	BRAMs	0	=
	DSPs	0	=

Reported resource usage after synthesis of the translated Chisel designs are on-par with resource usage reported for their respective original (System)Verilog implementation. The 27 additional LUTs for the Chisel PicoRV32 are inferred in a quickly refactored submodule which is relying on Verilog's blocking assignments in its original implementation. Moreover, as FIRRTL compilation to Verilog is flattening complex structures and expressions, some slight differences between the two resource counts can be expected.

6.2.4.3 Limitations

The development of *sv2chisel* followed a test-driven design methodology, starting with a reduced IR and limited syntactic-sugars support that were sufficient for our first internal SystemVerilog module. Applying the tool on external open-source examples ensured to enlarge the diversity of syntax and concepts properly translated up to a decent subset of (System)Verilog constructs. Each supported input construct comes with a corresponding unit-test while the presented real world examples serve as integration tests to validate the correctness of the translation. As a next step, further semantic analysis of Verilog and Chisel could be leveraged to formally design and prove the correctness of the transformation rules. At the time of writing our tool empirically covers the (System)Verilog synthesizable subset except for compiler directives (Verilog pre-processor) and blocking assignments which find no direct equivalent in Chisel.

Among manual adjustments required to achieve a correct translation, some are due to currently missing concepts in Chisel while others could be supported and properly translated without designer's help. Other translation issues reflect missing concepts in Chisel, such as (System)Verilog variables associated with blocking assignments. This pattern requires EDA tools to automatically infer multiple implicit intermediate signals and has no straightforward equivalent in Chisel. In Chisel, recursive functions or other functional constructs can be leveraged to implement the same hardware architectures with a more idiomatic expression.

6.2.5 Conclusion

In this section we introduced *sv2chisel*, a synthesizable (System)Verilog to Chisel translator. Producing Chisel code close to a word-for-word translation of the original source and preserving overall layout and comments, this tool is intended as a first step in migrating valuable legacy Verilog code-bases to Chisel.

Although it does not aim at fully replacing the fine human expertise required to achieve a correct translation in the most advanced cases, it appears capable of producing very decent translation drafts. These drafts are then expected to be manually inspected and refactored to leverage the power of HCLs, e.g. with the use of object-oriented and functional constructs.

As a result, *sv2chisel* enables designers to integrate SystemVerilog IPs within Chisel, avoiding the restrictive *black-boxing* pattern, and permitting to effortlessly propagate any upgrade through the entire hierarchy.

While this tool brings a solution to the upstream integration challenge, the next section focuses on its downstream counterpart: integration of HCL-generated HDL sources within a top level HDL hierarchy.

6.3 HCL-as-IP: Downstream Integration of HCL-Generated Architectures

6.3.1 Motivations

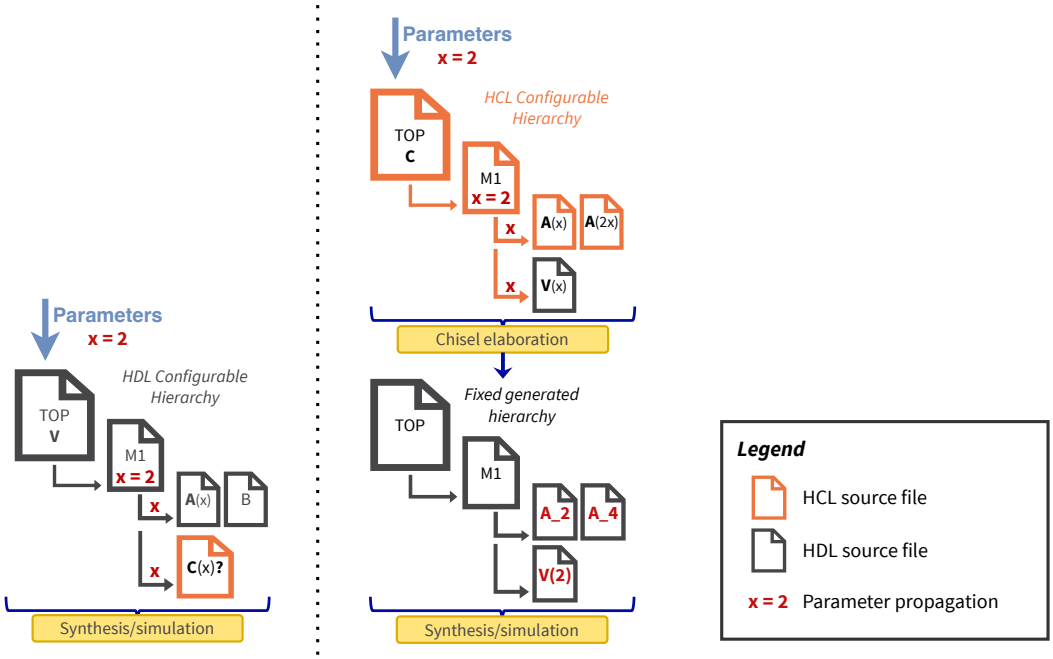


Figure 6.5: Parameter propagation compared in HDL and HCL hierarchies

A common principle to most HCLs consists in the elaboration of a high-level description, during the execution or compilation phase provided by the HCL toolchain. This process—further detailed in Section 3.3.1.3—produces a lowered version of the description, which is then usually emitted as a low-level HDL. As this elaboration stage takes place in the software domain, HCLs are able to natively provide many software engineering features, such as functional parameterization, type polymorphism and object-oriented programming.

In the case of embedded HCLs based on explicit circuit construction, such as Chisel, the resulting circuit is elaborated through the software execution of theses high-level constructs. As a result, these constructs are no longer available in the elaborated hierarchy. In particular, module parameterizations are lost, making such HCLs unable to produce configurable hierarchies. The configuration, which can include complex parameters such as high-order functions, is indeed intended to be passed to the generator, producing a distinct HDL output for each parameter set.

Figure 6.5 summarizes the differences between HDL hierarchies and HCL-generated ones. The left side presents parameter propagation in a traditional HDL hierarchy. Parameters are provided at top level and propagated down to the parameterized instantiations of HDL

module `A`, in one case parameterized with x and in the other case with $2x$. As this very basic elaboration process is directly handled by simulation and synthesis tools, this parameter propagation is effortless for the designer. The ability to instantiate a parameterized HCL IP such as `C(x)` is the objective of the current section.

To highlight the integration issue, the right side presents a similar parameter propagation in an HCL hierarchy, which is also effortless for the designer. The lower part of the figure illustrates the HDL hierarchy obtained after HCL elaboration. The parameterized module instantiation has been replaced with elaborated instances for each given parameter set, here for $x = 2$ and $2x = 4$. Integration of HDL *black-box* `V(x)` is also pictured, retaining a parameterized instantiation but with a constant parameter, here $x = 2$.

Integration of such fixed hierarchy as a configurable HCL IP within an HDL hierarchy requires to maintain a synchronization between HCL generators and HDL instantiations, as previously illustrated in Figure 6.1. To solve this synchronization issue at the scale of an industrial HDL hierarchy, two parameterization cases must be considered:

Global Parameterization Relying on a common source of knowledge

Local Parameterization Based on parameter propagation within the HDL hierarchy, such as bus widths, which are often locally computed from other parameters.

Implementing a global synchronization mechanism is a pure engineering matter and highly depends on the overall project organization and needs. For our network device projects we were already relying on a common configuration defined in Python and producing SystemVerilog packages for the hardware side and Python packages for the software side. To integrate Chisel developments in this system, a Scala package generation can be added as a new generation backend. Another option consists in fully migrating the configuration in Scala with reflexive generation of SystemVerilog and Python versions of the constants. This more expensive and intrusive approach would be the most logical from the perspective of typing: Scala is highly typed and carries the largest amount of information for each constant. Dropping this information to generate Python and SystemVerilog packages is then more efficient than annotating a poorly typed Python definition to generate a proper Scala package.

While a global parameterization might be enough to integrate one parameterized instance within the HDL hierarchy, local parameterizations remain required to integrate multiple instances of a same HCL generator. Local parameterizations are highly context dependent. Migrating all this fine-grained knowledge to a global parameterization would be extremely constraining and would defeat the reusability purpose of parameterization.

Last but not least, Chisel generation produces flattened versions of all structured signals. For example, a `Vec(N, Bool())` is emitted as N 1-bit width signals instead of an N-bit width addressable vector `[N-1:0]`. Similarly, a structured type `new Bundle { val a, b = Bool()}` is emitted as two independent signals rather than a SystemVerilog `struct`. This discrepancy has a high impact when integrating Chisel-generated modules within a traditional HDL hierarchy, especially when width parameterization results in a varying number of ports.

As a conclusion, integration of HCL-generated architecture within a top level HDL hierarchy requires a fine-grained solution which preserves the ability to locally instantiate parameterized IPs with automated mapping of flattened Input and Output ports (IOs) with their structured counterparts.

6.3.2 Pseudo-generic Wrapper-based Solution

To solve the integration of multiple parameterized instances of a Chisel generator within a SystemVerilog hierarchy, without requiring modification of the latter, we rely on a pseudo-generic wrapper. It is intended as a drop-in replacement of an equivalent parameterized

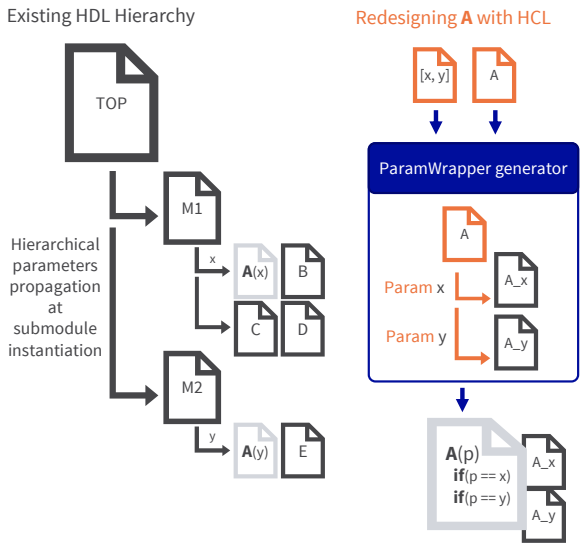


Figure 6.6: Paramwrapper principles

SystemVerilog module. This approach enables designers to replace widely replicated instances with their Chisel-generated counterparts with no impact on the existing hierarchy and at a very low integration cost.

Figure 6.6 illustrates this principle. The wrapper is only generic as far as the corresponding parameterizations have been explicitly generated. From an integration perspective, the module instantiation in the SystemVerilog hierarchy remains rigorously identical. The only additional constraint consists in centralizing the parameter sets. While we have not worked on an automation of this phase, the required parameterization can be extracted from any EDA tool able to elaborate the HDL hierarchy. More trivially, utilization of an empty shell simply reporting the expected parameterizations highly reduces this tedious task.

In practice, this pseudo-generic wrapper consists in a basic switch case structure, as illustrated in listing 6.8.

The input parameters of the wrapper are matched against predefined parameter sets, and in case of matching parameters the corresponding Chisel-generated module is selected and connected to the Input and Output ports (IOs) of the wrapper.

While such a wrapper could be written manually, this would defeat the intended parameterization flexibility by requiring extensive manual modifications. It would also require maintaining the duplication of configuration parameter sets between the generation side and this wrapper. Figure 6.6 presents a more elegant and integrated solution, based on a single entry point to generate both the various Chisel configuration and the associated wrapper. This entry point is designated as *ParamWrapperGenerator* and produces a self-contained SystemVerilog file with the wrapper alongside all the generated modules for integration convenience. Its implementation is detailed in the next section.

6.3.3 *ParamWrapperGenerator* Implementation

Figure 6.7 illustrates the implementation of the *ParamWrapperGenerator*. It first takes as input a set of parameterized Chisel generators associated to an equivalent Verilog parameterization, as illustrated in Listing 6.9. Then, each generator is successively processed in a loop which consists of the following steps:

Chisel Elaboration Handled by Chisel front-end, Chisel elaboration produces an intermedi-

```
module ip_wrapper #(
  parameter INPUT_WIDTH,
  parameter OUTPUT_WIDTH
)(
  input [INPUT_WIDTH-1:0] in,
  output [OUTPUT_WIDTH-1:0] out,
);
  generate
    if(INPUT_WIDTH == 85 && OUTPUT_WIDTH == 8) begin
      ip_0 ip(
        .in(in),
        .out(out),
      );
    end else if(INPUT_WIDTH == 32 && OUTPUT_WIDTH == 11) begin
      ip_1 ip(
        .in(in),
        .out(out),
      );
    end
  endgenerate
endmodule
```

Listing 6.8: Wrapper example for a simple *ip* module with two parameterizations

```
// listing generation config
val paramsMap = Seq(
  Map("inWidth" -> 85, "outWidth" -> 8),
  Map("inWidth" -> 32, "outWidth" -> 11)
)

// mapping generation config against actual parameters and provide associated generator
val instances = paramsMap.map(param =>
  ParamSet(Seq(
    "INPUT_WIDTH" -> IntParam(param("inWidth")),
    "OUTPUT_WIDTH" -> IntParam(param("outWidth")),
  )) -> (() => new XorHash(param("inWidth"), param("outWidth")))
)

// emit the wrapper and generate all parameterized instances
ParamWrapperGenerator.emit(instances, "chisel_xor_hashes")
```

Listing 6.9: Front-end of *ParamWrapperGenerator* for a dual configuration of `XorHash` module

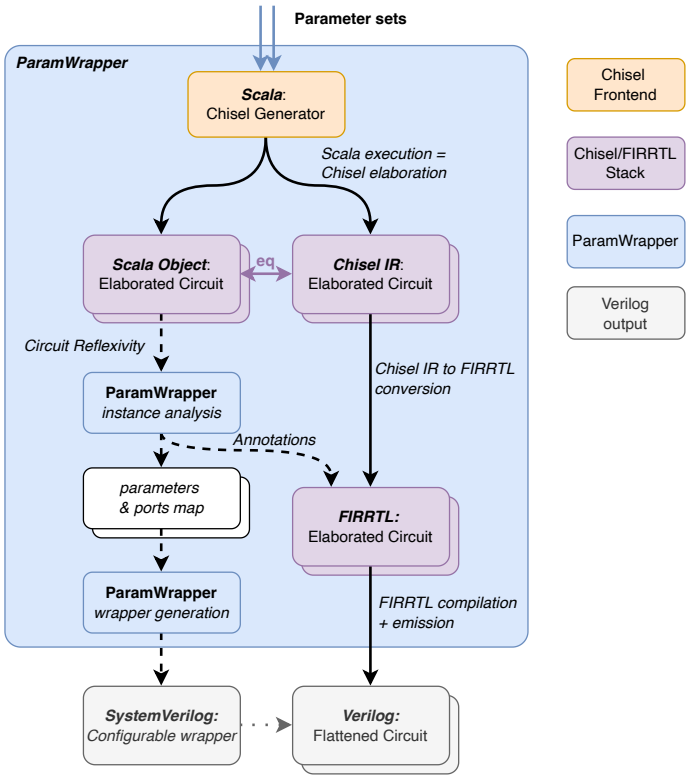


Figure 6.7: *ParamWrapperGenerator* implementation

ate representation of the circuit along with the equivalent Scala objects which provide reflexivity over the circuit,

Elaborated Circuit Analysis This reflexivity over the elaborated circuit is leveraged to retrieve the port lists in their two versions: structured and flattened. This information is internally stored for the final emission of the wrapper.

FIRRTL compilation and emission The elaborated circuit is emitted as Verilog, with a custom name mangling transformation to ensure uniqueness of all module and submodule names between parameter sets.

Finally, the wrapper is emitted based on port map information retrieved during the processing loop. Listing 6.10 presents the obtained wrapper. An error is raised here in case of mismatch, but a similar pattern could be used to record the required parameter sets, for example by appending them to a file. The wrapper IO ports are defined with the largest width encountered among the variety of instances IO ports. To ensure compliance with various EDA tools, inputs are bit-selected wherever necessary and unused output bits are connected to the ground.

While implementing this generation system requires a substantial engineering effort, its principle is permitted by the reflexivity over the elaborated circuit provided by Chisel/FIRRTL stack. Thanks to this generation API, a fully integrated solution can be provided to the IP integrator.[3]

[3]Appendix B presents a more detailed example, focused on the mapping between flattened and structured ports for proper integration within the existing HDL hierarchy.

```
module chisel_xor_hashes #(
  parameter INPUT_WIDTH,
  parameter OUTPUT_WIDTH
)(
  input clock,
  input [84:0] io_req_bits,
  output [10:0] io_res_bits,
);
  generate
    if(INPUT_WIDTH == 85 && OUTPUT_WIDTH == 8) begin
      chisel_xor_hashes_XorHash_0 XorHash(
        .clock(clock),
        .io_req_bits(io_req_bits),
        .io_res_bits(io_res_bits[7:0]),
      );
      assign io_res_bits[10:8] = '0;
    end else if(INPUT_WIDTH == 32 && OUTPUT_WIDTH == 11) begin
      chisel_xor_hashes_XorHash_1 XorHash(
        .clock(clock),
        .io_req_bits(io_req_bits[31:0]),
        .io_res_bits(io_res_bits),
      );
    end else begin: unmapped_param_set
      initial begin
        $info(">INPUT_WIDTH = %d\n >OUTPUT_WIDTH = %d\n", INPUT_WIDTH, OUTPUT_WIDTH);
        $error("CRITICAL FAILURE: Unmapped parameter set");
      end
    end
  endgenerate
endmodule
```

Listing 6.10: Wrapper generated for the parameterization given in Listing 6.9

6.3.4 Practical Evaluation: Replicated Hash Function Replacement

To demonstrate the relevance of a drop-in replacement of an existing widely replicated HDL module, we applied our *ParamWrapperGenerator* to a hash module notably used within the multiple hash-table instances introduced in the previous chapters.

The original version of this module is based on a custom hash generation algorithm written in Python. From parameters such as input and output widths of the hash, the Python script generates a binary configuration for each hash instance, corresponding to several levels of logic operators between input and output bits. The SystemVerilog module reads this binary configuration from a file and generates the appropriate connections. Without diving further in the SystemVerilog implementation, the architecture is fully generic and based on huge multidimensional vectors which are the usual way of building generic design with standard HDLs. As a result, the elaboration of this highly generic module and in particular constant propagation is costly in time for simulation and synthesis tools.

The Scala/Chisel replacement is based on the Python generation script and provides a fully integrated solution with the exact same limitation: hashes have to be generated in advance based on a predefined list of configurations. Integration of these Chisel hashes within our top level SystemVerilog has been effortless and came with interesting benefits, summarized as follows:

Original Verilog Version	**Chisel Replacement**
Slow Elaboration Requires High-end Simulation Tools Python-based config generation	Fast Chisel-driven elaboration Tool Agnostic Integrated Scala/Chisel Solution
Pseudo-generic integration (due to python generation)	Pseudo-generic integration (due to *ParamWrapper*)

As a conclusion, our *ParamWrapperGenerator* enables designers to smoothly integrate Chisel modules as IP within an existing SystemVerilog hierarchy. In particular, it unlocks access to Chisel high-level generating constructs, even when a complete replacement of upper hierarchy would require too much work to be considered in an industrial context.

6.4 Conclusion

This chapter introduces two contributions to interface Chisel with an existing SystemVerilog hierarchy. As migration of large legacy projects cannot happen overnight, we keep the perspective of SystemVerilog as top level hierarchy with integration of Chisel IPs as submodules.

The first contribution occurs upstream of the HCL flow and focuses on an agile upgrade flow from existing HDL architectures to advanced HCL generators.

The second one takes place downstream of the generation and focuses on integration of multiple parameterized instances of Chisel-generated architecture within the SystemVerilog hierarchy. It also solves the integration of generated modules and their flattened IO ports.

As part of the larger research effort aiming at validating the relevance of HCLs as part of the EDA ecosystem, we published our two contributions as open-source software on GitHub.[4]We claim that the adoption of HCLs within existing HDL projects could greatly benefit from this cohabitation perspective, aiming at a smooth and agile migration of existing HDL code-bases towards advanced HCL generators. The next chapter presents a complete experiment, showcasing this approach and integrating our previous agile design upgrades.

[4] *sv2chisel*: https://github.com/ovh/sv2chisel; *ParamWrapperGenerator* as library of *sv2chisel*: https://github.com/ovh/sv2chisel/tree/master/helpers/src/main/scala/tools

Chapter 7

Experimentation

This chapter illustrates the three main contributions presented in this thesis, through a unified experimentation at an industrial scale. From the large legacy SystemVerilog codebase at the heart of our network devices at OVHcloud, we select a core network functionality, a packet classifier, and follow its upgrade journey from Chisel translation, to pipeline oriented upgrade, to re-integration within the original hierarchy.

First, we detail the tree architecture of this packet classifier, whose original SystemVerilog implementation turns out to be extremely complex to ensure proper fit on several FPGA targets.

Then, we review its translation to Chisel, based on our automated translator *sv2chisel*, and completed with the necessary manual adjustments to guarantee on-par resource usage and a reasonable elaboration overhead.

Next, we refactor the implementation of this module to leverage our pipeline framework and drastically improve the flexibility and readability of the architecture.

Finally, we explore the various resolution strategies and their parameterizations to offer a very fine-grained control over the generated architectures to the designers, and help with timing closure.

Contents

7.1 Tree Filters Design and SystemVerilog Implementation

7.1.1 Context

Network devices are responsible for packet processing at various place of the network infrastructure. They are based on various analysis and operations on the packet stream flowing through their interfaces. These *network functions* can be sorted in four main categories: parsers, classifiers, flow analyzers and operators. The following table details the role of each of these blocks along concrete practical examples and their implementations.

Network Function	Input	Output	Implementation
Parsers	Packet	Metadata	*Ad-hoc*
Classifiers	Metadata	Stateless Labels *Flow Element*	Trees, Matchers
Flow Analyzers *e.g. counters, patterns*	Stateless Labels *Flow Element*	Stateful Labels *Flow Behavior*	Hash-table, CAMs
Operators *e.g. drop, extract, route*	Labels	*Ad-hoc Action*	*Ad-hoc*

Based on these main network functions, Figure 7.1 illustrates a classical pipelined architecture for network devices. It highlights data dependencies, and the availability of various extracted and computed elements through the network pipeline. The order in which network functions are placed is indeed critical due to these dependencies, from packet to metadata, to label, to concrete actions.

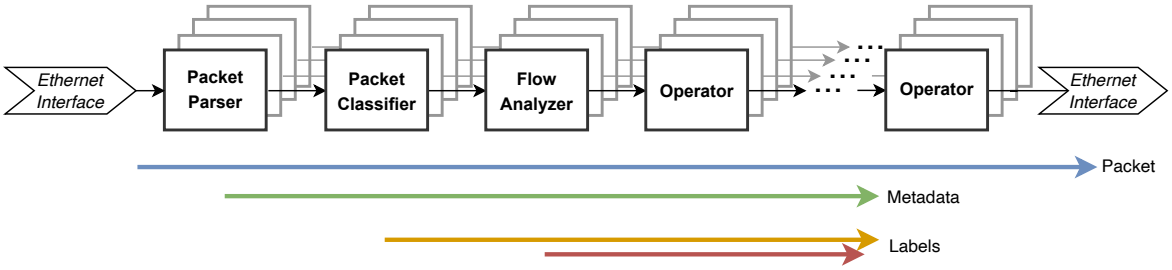
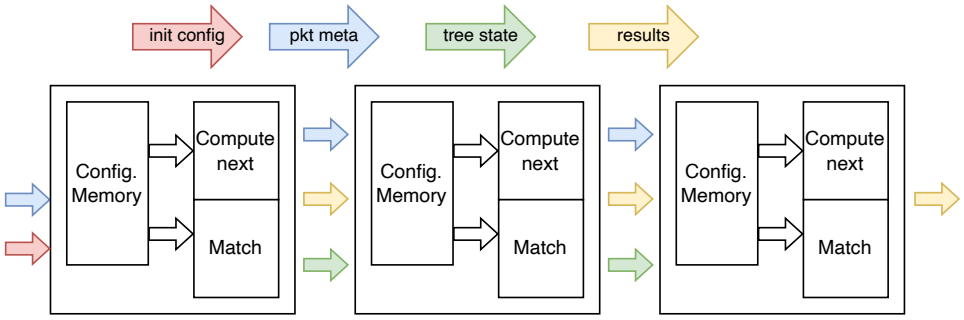


Figure 7.1: Coarse-grained generic architecture of network devices

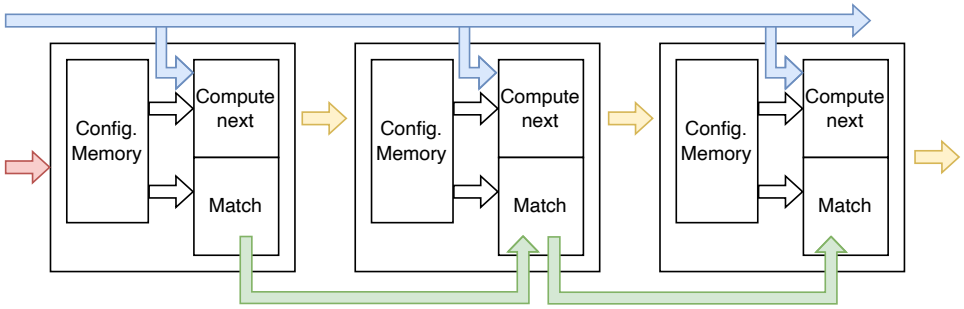
In this architecture, packets remain available throughout the pipeline, whereas metadata are only available after *ad-hoc* parsing and labels are created based on the metadata by classifiers and flow analyzers. In practice, multiple parsers, classifiers, analyzers might be used and interleaved in between operators as long as data dependency requirements are fulfilled.

Implementations of network functions are relying on a relatively small number of critical base components. In previous chapters we explored and detailed several implementations of state-based storage functionality, at the core of flow analysis functions. This chapter focuses on an upstream critical network function: packet classification, i.e. the association of a packet with one or several labels, also known as packet filtering. These labels are then used by downstream operators to select the behavior of the device, such as dropping the packet, extracting it or encapsulating it.

A packet classifier enables defining a flow as a sequence of packets associated to the same label. A given classification might correspond exactly to a given *5-tuple* or match more broad



(a) Conceptual design of the classifier illustrated on 3 stages



(b) Actual implemented architecture of the classifier

Figure 7.2: Packet classifier architecture design versus implementation

criteria, such as a range of IP addresses or a given protocol. Many algorithms and associated architectures can be used to implement packet classifiers, ranging from linear search to geometric algorithms to advanced heuristics [GM01, Tay05]. As always in hardware design and implementation, configurability and performance induce considerable resource costs, and a balance must be found between these parameters to provide efficient packet classifiers. Using a tree approach such as the one proposed in HyperCuts [SBVW03], classification rules can be compressed to keep the implementation affordable in resources, even with a one-packet-per-cycle throughput. The tree structure requires strong synchronization between packet metadata, internal state and results, which makes it an interesting application for our framework. Figure 7.2a illustrates its base principle: a succession of configurable processing stages, each corresponding to a level in the tree. While only three stages are depicted, the number of stages is fully configurable and depends on the maximum number of rules expected for a given application. At each stage, based on the result of the previous stage, the current configuration is retrieved from a memory. This configuration is then combined with packet metadata and internal tree state to produce the updated tree state, a classification result, and the address of the configuration for the next stage.

As aforementioned, a network packet metadata usually includes the so-called *5-tuples*: source and destination IP address, protocol, and source and destination ports. For an IPv6-enabled network device, with 128-bit IP addresses, 8-bit protocol identifier, and 16-bit ports, it results in a bare minimum of a 296-bit data bus to be distributed to each stage with proper synchronization. In practice, with an experimentally reasonable number of 40 stages and 6-cycle latency per stage (2 for memory access, 4 for computations), it requires already more than 70k registers only to forward and provide metadata to each stage. The impact on resource usage of the actual implementation of this signal forwarding across the design is considerable and such architectural choices become critical to eventually meet timing requirements.

7.1.2 Verilog Implementation

The original implementation of this classifier has been written in SystemVerilog and targets both Intel and Xilinx/AMD FPGAs, respectively *Stratix V* and *Ultrascale Plus* architectures [HG21]. As the number of registers in an ideal implementation, in which data is forwarded along with computations, is too high to allow efficient placement and routing of the design, we instead leverage vendor-provided shift registers, which comes with their own capabilities and limitations. Xilinx provides LUT-based SRL primitives, able to manage up to 32-bit long and 1-bit wide shift registers. Its toolchain, Vivado, is able to efficiently infer shift registers as long as they do not cross a hierarchy boundary. On the contrary, Intel toolchain, Quartus, avoids shit-register inference in most cases because it uses M20K memory blocks, which are a rather rare and precious resource in the FPGA, often harder to route. To force the use of appropriate vendor primitive, we had to manually extract data forwarding parts of the design, and integrate them directly in the top level module, which breaks the *black-box* paradigm and requires spreading information on internal pipelines lengths.

Based on the theoretical architecture presented in Figure 7.2a, Figure 7.2b depicts the *ad-hoc* implementation required to set up this explicit signal forward strategy. The three different streams of data are passed along the stages with their own explicit forward implementations:

Packet Metadata Read once at the beginning of the stage and not modified internally. A tapped shift-register is effective here, to forward metadata aside the pipeline, and supply them synchronously to each stage, after the configuration retrieval.

Compute Results Forwarded within the main pipeline, as they are used and modified all along the stages. Proper split with computation here is difficult, and can only be done locally in the design, e.g. in parallel with memory access.

Tree State Modified only once during the compute stage, after the configuration retrieval. This modification is simple and done in a combinational way, permitting the implementation of a separate shift-register-based bypass of the depth of a complete stage latency.

In the end, modularity of the original architecture is greatly restricted with this optimized implementation, which is hard to maintain and upgrade. For example, adding or removing a register within a compute block requires recomputing and propagating the latency change across the entire pipeline, in order to ensure that data remains properly synchronized.

7.1.3 Fine-grained Reverse Engineering

The fine-grained optimizations leveraged by the current SystemVerilog implementation have been carefully crafted through numerous trials and errors, by a single experienced hardware engineer. As often in such cases, the documentation is sparse and remains focused on algorithms and concepts behind the optimizations rather than their practical implementations. As a result, the module hierarchy is complex and hard to apprehend for an external observer. To get a precise grasp at the actual synchronization requirements between each part of this hierarchy, we leveraged a modified version of our *sv2chisel* tool to draw the existing hierarchy with a focus on signals and their synchronizations. In particular, we are interested in registered paths and combinational paths across the hierarchy.

As detailed in Section 6.2.3.1, inference of registers from a SystemVerilog description requires a semantic analysis, and cannot simply be achieved from a basic parsing and its resulting syntax tree. Based on the existing capabilities of *sv2chisel transforms* regarding this

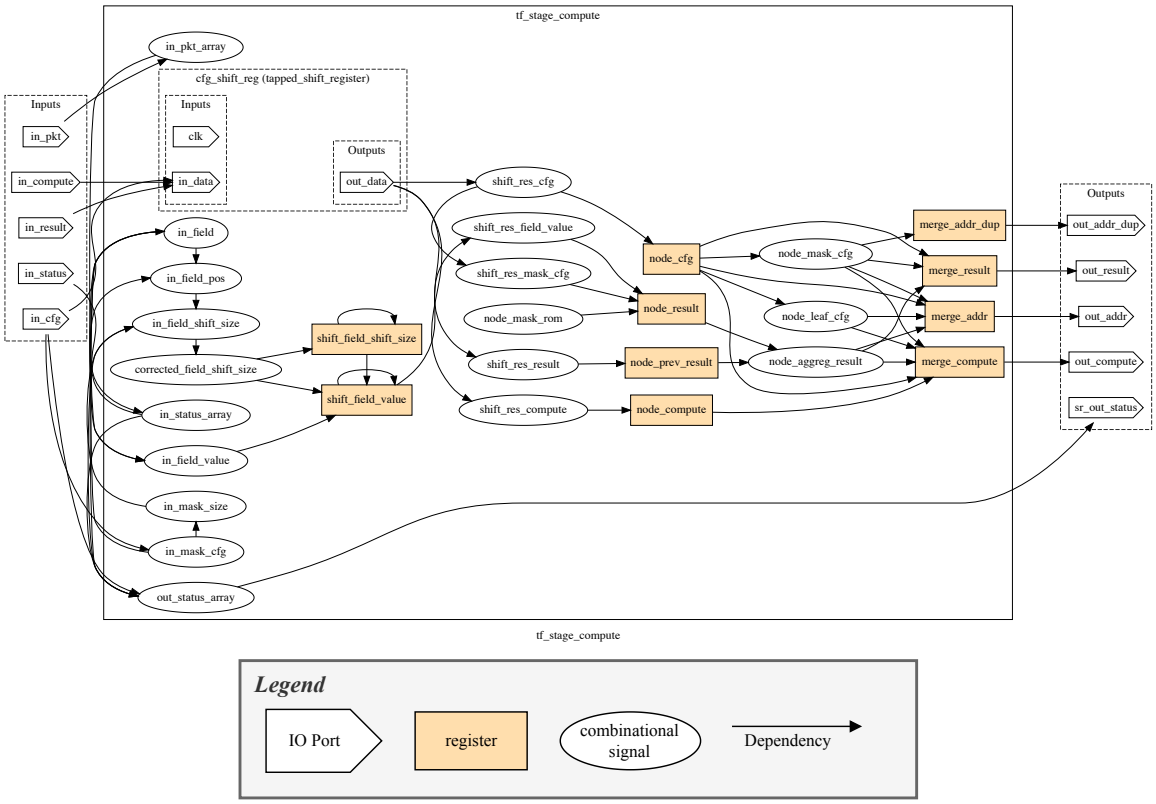


Figure 7.3: Generated synchronization-oriented representation of one inner submodule of *Tree-Filters* hierarchy

matter, we developed an alternative back-end of *sv2chisel*, able to draw signal relationships through registers. Unlike existing circuit drawing utilities, this one is not focused on the operators and logic operations between signals but only on their synchronizations. Figure 7.3 illustrates the result obtained for one of the internal component of the *tree filters* hierarchy.[1]

This preparatory study of the existing SystemVerilog hierarchy highlights how the strong signal synchronization needs, mixed with target considerations, have led to a highly complex implementation. A considerable part of the implementation overhead is dedicated to delay signals from a stage of the pipeline to the next. This is the very context in which our pipeline framework is relevant to simplify the description and regain control over architecture parameterization for enhanced flexibility and reusability. However, leveraging our framework first requires translating the existing implementation to Chisel. The next section details this translation process.

7.2 Chisel Translation with *sv2chisel*

As a first step towards advanced evolution of the *tree filters* architecture, translation of this complex existing SystemVerilog hierarchy into a more flexible language is a tedious task. To avoid any regression, it begins with a word-for-word translation and validation of this initial translated version against the existing test-benches. Then, the validated version, later referred as *Vanilla Chisel*, can be confidently upgraded with advanced features such as pipeline management.

Chapter 6 highlighted some inherent challenges of SystemVerilog to Chisel translation,

[1] Appendix C relates the fine-grained architecture of all modules in further details.

and detailed the limitations of test-driven development to provide exhaustive coverage of SystemVerilog constructs. In particular, the existing *tree filters* hierarchy makes an extensive use of advanced synthesizable SystemVerilog constructs such as enumerations, functions, packages and integration of vendors IPs. The description also leverages custom tricks to circumvent some generation limitations such as iteration over `struct` fields. As a result, a substantial effort is required to achieve the proper translation of this hierarchy. First, additional development of *sv2chisel* is required to support these constructs, then appropriate translation of some data-types and manual refactoring of anti-idiomatic patterns is further discussed.

7.2.1 Additional Translating Automation

Unlike previous uses of *sv2chisel* which achieved the translation of several independent (System)Verilog modules into Chisel, the translation of *tree filters* architecture requires a project based approach. In particular, as the hierarchy makes an extensive use of packages to define global variables which are used in various place of the design, type inference algorithms must take all usages into account. Similarly, the translation must guarantee consistent type inference for module instantiations, with appropriate casts of ports and parameters wherever required.

Aside from such a structural improvement of the automated translation process, several other developments are required, from support of new constructs to enhanced semantic analysis. Due to the very nature of test-driven development approach, several SystemVerilog constructs had been left unsupported in previous versions of *sv2chisel*. Among them, the following constructs are required to accurately translate the existing *tree filters* architecture and are now supported by *sv2chisel*: `function`, `enum` and `package`. The remaining content of the present section reviews two semantic analysis improvements, both related to generation conditions.

```
module wire_or_reg(parameter INSERT_REG = 0)(
  input clock,
  input in,
  output out,
);
  logic signal;
  generate
    if (INSERT_REG) begin
      always @(posedge clock) begin
        signal <= in;
      end
    end else begin
      assign signal = in;
    end
  endgenerate
  assign out = signal;
endmodule
```

Listing 7.11: Conditional register behavior of a signal in SystemVerilog

```
class wire_or_reg(INSERT_REG:Boolean = false){
  // implicit clock
  val in = IO(Input(Bool()))
  val out = IO(Output(Bool()))

  val signal = if(INSERT_REG) Reg(Bool())
               else Wire(Bool())
  // Tautologic condition: should be removed
  if(INSERT_REG){
    signal := in

  } else {
    signal := in
  }

  out := signal
}
```

Listing 7.12: Translation of Listing 7.11 with *sv2chisel*

A first automation requiring semantic analysis consists in the correct translation of parameterized inference of signals, either as simple `wire` or as `register`. Listing 7.11 illustrates such parameterized behavioral description in Verilog and Listing 7.12 presents the equivalent Chisel translation as obtained with *sv2chisel*. In the Chisel translation, the condition is now required at the declaration of `signal`. With additional analysis of equivalent

source in both cases, lines 7–12 could be simplified to merge equivalent statements as a single one and get rid of the tautological condition.

```
1  generate
2    if (PARAM_A) begin
3      always @(posedge clock) begin
4        if(signal) begin
5          signal <= in;
6        else begin
7          if (PARAM_B) begin
8            signal <= '0;
9          end else begin
10           signal <= '1;
11         end
12       end
13     end
14   end else begin
15     // ...
16   end
17 endgenerate
```

Listing 7.13: Nested generation and circuit conditions in SystemVerilog

```
// outside a process
if (PARAM_A) { // => generation statement
  // within a process and signal is hardware
  when(signal) { // => actual hardware multiplexer
    signal := in
  } .otherwise {
    // within a process but PARAM_B is software
    if (PARAM_B) { // => generation statement
      signal := false.B
    } else {
      signal := true.B
    }
  }
} else {
  // ...
}
// NB: no explicit generate block required
```

Listing 7.14: Nested generation and circuit conditions in translated Chisel

Another semantic improvement consists in the analysis of nested conditions within a `generate` block and a clocked process. While this improvement is not required for the proper translation of *tree filters*, it generates more idiomatic Chisel code. In a nutshell, in Chisel there is a distinction between two conditional constructs. The first, `if`, is provided by Scala and corresponds to an elaboration or generation condition which does not exist anymore in the final circuit. The second, `when`, is provided by Chisel and results in hardware multiplexers. To provide an idiomatic translation, the designer intent must be retrieved from the overall context, and in particular take the *kind* of operands into account. A *software kind* corresponds to parameters and all elaboration-related logic, whereas a *hardware kind* corresponds to actual signals in the resulting circuits. Listing 7.13 illustrates nested `if` conditions in SystemVerilog, where the interpretation as elaboration-time or actual hardware conditions is left to the designer. Listing 7.14 presents the equivalent translation in Chisel with the design intent retrieved from operands *kinds*.

7.2.2 Investigating Translation Overhead

While the previous paragraphs focused on providing effortless and functionally equivalent translations, this part reviews performance of the translated Chisel in terms of generation efficiency, simulation speed and resource usage. The following setup has been used for all experiments in this section:

Generation Chisel/FIRRTL Stack 3.5.0-RC2

- sbt 1.5.7 *with* `-mem 8192` *option forwarded to java, required for FIRRTL compilation*
- scala 2.12.15
- Ubuntu Java 11.0.13

Simulator Verilator 4.106 *with wave tracing enabled, unless otherwise specified*

Test Framework cocotb 1.5.1

FPGA Targets with two different architectures

		Baseline SV	Initial Chisel	Diff	%
Generated Files (MB)	*Chisel Stack*				
	top.fir	-	189		
	top.v	0.31[3]	22	+21.7	+7,233%
	Verilator				
	top_tb[4]	22	118	+98	+445%
	*.cpp	95	347	+252	+265%
	*.h	1.4	15	+13.6	+971%
	dump.fst[5]	2.5GB	4.2GB	+1.7GB	+68%
Duration (mm:ss)	Scala compilation	-	00:10		
	Chisel elaboration	-	00:18		
	FIRRTL compilation	-	02:11		
	Verilation[6]	00:20	∼16:00	+15:40	+4,705%
	C++ compilation	00:40	∼11:00	+10:20	+1,550%
	TOTAL	01:00	∼29:30	+28:30	+2,850%
	Simulation rate[7]	916ns/s	300ns/s	-616ns/s	-67%

Table 7.1: Initial overhead of translated *Tree filters* architecture

- Xilinx Virtex Ultrascale+ @ 200MHz *(Vivado 2018.2)*
- Altera/Intel Stratix V @ 150MHz *(Quartus 15.1)*

Table 7.1 compares the initial translation against the original SystemVerilog version, with a focus on duration of main steps of the test flow and on the volume of generated files. These first results show a major increase in setup time, from generation to simulation effective startup, with more than an order of magnitude between SystemVerilog baseline and initial Chisel translation. The simulation is also drastically slowed down, with a 67% reduction of the simulation rate. These observations match the massive increase in volume of the C++ simulation model[2]generated by Verilator. Comparison of such generated file sizes without ability to analyze their content is empiric but showcases the structural differences between handcrafted and generated Verilog, despite an identical functional behavior.

An execution overhead was expected due to the complexity of the original hierarchy and the loss of parameterizations for the simulator, however, these results make the translation unfit for re-integration in its current form. Serious actions must be taken to contain this overhead under reasonable limits.

The comparison in size between handcrafted SystemVerilog and generated Verilog is included in Table 7.1 to highlight the fact that generated files are already much larger before their compilation into a simulation model. While the generated Verilog is expected to be larger than its handcrafted counterpart, its conciseness should still be inspected to ensure a proper integration with downstream EDA tools.

[2]*Verilator* simulator first compiles the Verilog hierarchy into a C++ simulation model, which is in turn compiled to a self-sustained simulation executable.

[3]Total handcrafted SystemVerilog files, this total also includes one global generated configuration file of 116K whose content is only partially used here.

[4]Final simulation executable file size

[5]Size of wave dump for simulation of 100,000 requests

[6]Compilation of Verilog sources into a C++ simulation model by Verilator

[7]For simulation of 10,000 requests

		Initial Chisel	Optimized Inference	Diff	%
Occurrences	`UInt(<w>.W)`	149	328	+179	+120%
	`Vec(<n>, Bool())`	216	37	-179	-82%
	`Vec + UInt`	*365*	*365*	=	=
	`asTypeOf`	186	151	-35	-18%
Generated Files (MB)	*Chisel Stack*				
	top.fir	189	80	-109	-58%
	top.v	22	6.5	-15.5	-70%
	Verilator				
	top_tb	118	44	-74	-62%
	*.cpp	347	141	-206	-59%
	*.h	15	4.9	-10.1	-67%
	dump.fst	4.2GB	3.6GB	-0.6GB	-14%
Duration (mm:ss)	Scala compilation	00:10	00:08	$\simeq$	$\simeq$
	Chisel elaboration	00:18	00:12	-00:06	-33%
	FIRRTL compilation	02:11	00:50	-01:21	-61%
	Verilation	~16:00	00:54	-15:00	-93%
	C++ compilation	~11:00	01:08	-10:00	-91%
	TOTAL	~29:30	03:12	-26:15	-89%
	Simulation rate (trace)	300ns/s	610ns/s	+310ns/s	+103%

Table 7.2: Positive impact of hierarchical type inference on *Tree filters* architecture

7.2.2.1 Improving Semantic Analysis For Performance

Close inspection of the generated files reveals the critical impact of one particular semantic choice during the translation process: type inference of Verilog bit-vectors. The semantic differences regarding the translation of one-dimension bit-vector to Chisel either as `UInt` or as `Vec[Bool]` have been previously detailed in Section 6.2.3.3. While the impact of this choice on small designs remains limited to a user convenience matter, it appears to be critical for performance on larger designs. From an elaboration perspective, both hardware objects are completely different. On the one hand, `UInt` is a single hardware object, that will always be processed and emitted as a whole. On the other hand, `Vec[Bool]` is a collection of *independent* hardware objects, that are individually treated at each stage of the compilation process, down to an emission as individual signals in generated Verilog.

While the individual processing of each bit of a bit-vector might enable further optimizations, such as constant propagation or dead-code elimination, it comes with a considerable overhead, from elaboration to emission to simulation.

Inspection of the initial Chisel translation reveals a high count of `Vec[Bool]` signal declarations, as reported in the first lines of Table 7.2. In *tree filter* hierarchy, these declarations are concentrated in global type declarations, in particular in data structure declarations, which are shared between many modules and used at their interfaces. Such an intense usage of suboptimally typed signals mechanically leads to a very high count of independent signals in the generated Verilog.

To tackle this issue, a cross hierarchy type inference, with support for structural types, has been added to the translation tool. Table 7.2 showcases the comparison between the initial translation, and the translation obtained after optimization of the type inference

[7] A simple optimization case, comparing `UInt` and `Vec[Bool]` behavior is presented in appendix D.1.

		Initial Chisel	Final Chisel	Diff	%
Generated Files (MB)	*Chisel Stack*				
	top.fir	189	17	-172	-91%
	top.v	22	6.4	-15.6	-71%
	Verilator				
	top_tb	118	41	-77	-65%
	*.cpp	347	134	-213	-61%
	*.h	15	4.8	-11.2	-75%
	dump.fst	4.2GB	3.0GB	-1.2GB	-29%
Duration (mm:ss)	Scala compilation	00:10	00:06	-00:04	-40%
	Chisel elaboration	00:18	00:07	-00:11	-61%
	FIRRTL compilation	02:11	00:30	-01:41	-77%
	Verilation	∼16:00	00:45	-15:15	-95%
	C++ compilation	∼11:00	01:03	-10:00	-91%
	TOTAL	∼29:30	02:31	-27:00	-92%
	Simulation rate (trace)	300ns/s	656ns/s	+356ns/s	+118%

Table 7.3: Total improvements of successive iterations on translated *Tree filters* architecture

algorithm. The considerable reduction of `Vec[Bool]` signal declarations in favor of their `UInt` counterparts has a very beneficial effect of the entire stack from generation to simulation. As an immediate consequence, the size of Chisel-generated files is substantially reduced. With much less independent signals to handle, the simulator is now faster to generate its C++ model, which in turn is faster to compile. As a result, the simulation is more than twice as fast as the original version. While not yet as fast as the original handcrafted SystemVerilog, this iteration greatly improves the overall user-experience. It enables the integration of the translated version with a consequent but reasonable overhead.

As this simple change in Chisel description has considerable consequences on the entire stack, the remaining 37 `Vec[Bool]` signal declarations require close inspection. In particular, due to bit-level manipulation of IP addresses in some part of the design, the latter are still globally declared as `Vec[Bool]`, generating no less than 128 signals for each usage. However, as part of the current experimentation, we do not attempt to optimize each signal declaration and focus on other potential improvements with more idiomatic Chisel/Scala generators.

7.2.2.2 From Antipatterns to Idiomatic Approaches

In an attempt to further improve the translation process, a close inspection of the translated Chisel code reveals some antipatterns, originating from SystemVerilog limited expressiveness. Appendix D.2 first details the complexity of word-for-word translation of such antipatterns and provides workarounds to nonetheless achieve a correct automated translation. It then discusses a manual rewrite in order to better express the original intent in a more idiomatic Scala/Chisel version, which results in substantial performance increase. Finally, appendix D.2.3 relates an optimization of the elaboration process based on Chisel idioms. The respective impact of these manual iterations on the translation overhead is detailed in associated appendices and included in the final comparison tables below.

		Baseline SV	Final Chisel	Diff	%
Generated Files (MB)	*Chisel Stack*				
	top.fir	-	17		
	top.v	0.31	6.4	+6.1	+2,065%
	Verilator				
	top_tb	22	41	+19	+86%
	*.cpp	95	134	+39	+41%
	*.h	1.4	4.8	+3.4	+242%
	dump.fst	2.5GB	3.0GB	+0.5GB	+20%
Duration (mm:ss)	Scala compilation	-	00:06		
	Chisel elaboration	-	00:07		
	FIRRTL compilation	-	00:30		
	Verilation	00:20	00:45	+00:25	+125%
	C++ compilation	00:40	01:03	+00:23	+58%
	TOTAL	01:00	02:31	+01:31	+155%
	Simulation rate (trace)	916ns/s	656ns/s	-260ns/s	-28%
	Simulation rate (no traces)	2719ns/s	2350ns/s	-369ns/s	-14%

Table 7.4: Final overhead of translated *Tree filters* architecture

7.2.2.3 Conclusion

Table 7.3 summarizes the positive impact of all successive iterations compared to the initial translation. It highlights the potential hidden cost of a *lazy translation* which does not take advantage of Chisel native types and constructs. Following our original approach with *sv2chisel* development, efficient translation of hardware architectures does require a human expertise, with an in-depth analysis of the designer intent.

7.2.3 Final Results

The evaluation of a translation from SystemVerilog to Chisel must take several criteria into account to guarantee a smooth user-experience. While the most obvious criteria are a functionally equivalent circuit and similar resource usage after synthesis, the development flow overhead must first be considered carefully.

7.2.3.1 Development Flow Overhead

From an initial translation with an unacceptable overhead, we detailed successful improvements whose combined impact makes the resulting Chisel a serious candidate for further developments and re-integration within the surrounding SystemVerilog hierarchy.

Table 7.4 presents the final overhead of Chisel translation, compared to the original SystemVerilog implementation. Despite substantial efforts to improve the translator and additional iterations over the resulting Chisel code, this final version still does not achieve on-par results with the original SystemVerilog implementation. Due to the very nature of HCLs, elaboration and compilation steps are required to convert the high-level description into a low-level circuit which can then be fed to usual simulation tools. Both steps necessarily introduce an initial overhead which might then be compensated by an acceleration of simulation tools as mentioned in Chapters 4 and 6 with translation of a custom complex generic hash function into Chisel. However, such compensation cannot be expected here, and the translated version appears even slower than the original version for two main

```
logic [MASK_W-1:0]       node_mask_rom [MASK_W:0];

initial begin
    for (int mask_rom_id = 0; mask_rom_id <= MASK_W; mask_rom_id++) begin
        node_mask_rom[mask_rom_id] = (1 << mask_rom_id) - 1;
    end
end
```

Listing 7.15: Original Verilog version with ROM initialization loop within initial statement

reasons. First, the original SystemVerilog implementation of *tree filters* does not contain such existing elaboration bottleneck that its Chisel counterpart could take charge of. Secondly, the architecture of *tree filters* highly relies on massively replicated instantiations of parameterized modules, which get emitted as distinct lowered Verilog modules. This duplication notably contributes to the large size of the emitted *top.v* file. The size of this circuit description as input of the simulator leads to increased parsing and analysis times to build the simulation model. This lowered and flattened circuit has lost a part of the designer intent which can no longer be used by the simulator. In particular, efficient simulators are able to take advantage of parameterized architectures to re-use some evaluating functions in various parameterized contexts. In the case of Verilator, some parts of a given parameterized module might get expressed as a single C++ function which is then called with the appropriate parameters. This results in an extensive re-use of the same instructions in the simulation executable, and incidentally in a better use of processor instruction cache. Such optimizations are not available for elaborated circuits.

As a conclusion, HCL generated circuits are expected to require additional setup time and to impair execution performance, except for specific cases in which non-elaborated designs appears too generic for efficient simulation. However, the current overhead remains acceptable to pursue experiments with the pipeline automation framework.

7.2.3.2 Circuit Quality Evaluation

While the flow overhead is critical from an agile development perspective, the most important criterion for hardware integration remains guaranteeing equivalent functionality and similar resource usage.

Regarding functionality, the automated translation successfully achieved functionally equivalent translation, only requiring a single manual fix, with a warning issued by *sv2chisel*. It was due to a memory initialization occurring within an `initial` statement as illustrated in Listing 7.15. Raw translated version and functionally equivalent manual fix are presented in Listing 7.16.

However, with this first functionally equivalent version, the reported resource usage after synthesis is quite different from the original SystemVerilog version. As presented in Table 7.5, this `RegInit` version results in a notable increase of Logic LUTs usage in place of RAM blocks. This implementation change appears to impact design ability to meet timing requirements, with only 7 synthesis strategies able to achieve successful routing.

To match the original implementation intent, the initialization of `node_mask_rom` has been upgraded to leverage a custom `MemInit` construct, as illustrated in Listing 7.17. Based on Chisel `Mem` primitive, this construct leverages FIRRTL annotations to preset initial values in the so-created ROM.[8] It enforces a Verilog generation close to the original SystemVerilog

[8]This construct is provided as part of *sv2chisel-helpers* library available at https://github.com/ovh/sv2chisel/blob/master/helpers/src/main/scala/MemInit.scala

```
// Raw Translated version (initial statements are ignored by sv2chisel)
val node_mask_rom = Reg(Vec(MASK_W + 1, UInt(MASK_W.W)))
// Functionally equivalent manual fix
val node_mask_rom = RegInit(VecInit.tabulate(MASK_W+1)(i => ((BigInt(1) << i) - 1).U))
```

Listing 7.16: Manual fix of translated Chisel to provide pre-initialized Vector

```
import sv2chisel.helpers.MemInit
val node_mask_rom = MemInit.tabulate(MASK_W + 1, UInt(MASK_W.W))(i => (BigInt(1) << i) - 1)
```

Listing 7.17: Final version forcing ROM inference for equivalent resource usage

implementation and results in almost similar resource usage after synthesis, as detailed in Table 7.5. In particular, it leverages the same behavioral pattern for `node_mask_rom` description, which is expected to be inferred as a memory block and implemented with block RAMs rather than registers. Table 7.5 confirms this expectation and reports the same amount of block RAM for both implementations.

This new version almost catches up with original SystemVerilog on routing, still unexpectedly failing to meet timing requirements with one routing strategy. While block RAM count is now equal to the original one, this imperfect result is probably linked to the substantial increase in Logic LUTs usage. Closer investigation reveals that this increase is concentrated in a widely replicated submodule, justifying the global impact of what seems to be a single translation discrepancy. However, a manual comparison of the original and translated versions of this submodule was not sufficient to identify the reason of this mismatching synthesis behavior. Additional comparisons, e.g. with other synthesis tools or various parameterizations of the module, could probably help to understand the root cause.

As a conclusion, the translation of the complex *tree filters* hierarchy from its original handcrafted and optimized SystemVerilog version to a functionally-equivalent low-level Chisel version is satisfactory. After several iterations, the initial development flow overhead has been contained under reasonable limits and the resource usage remains similar. Several areas for improvements have been identified to provide a smoother development experience and contain resource usage overhead. Nonetheless, based on this decent translation, the next section pursues the experimentation with the integration of our pipeline framework to replace all complex and manual signal synchronizations by a pipeline-oriented description.

	∑ LUT	LLUT	LUTRAM	SRL	FF	RAMB36	RAMB18	URAM	Timing
SystemVerilog	90,316	59,266	0	31,050	63,945	20	190	92	14/14
Chisel RegInit	95,119	64,127	0	30,992	62,793	20	0	92	7/14
	+4,803	*+4,861*	=	*-58*	*-1,152*	=	*-190*	=	
	+5.3%	*+8.2%*	=	*-0.1%*	*-1.8%*	=	*-100%*	=	
Chisel MemInit	92,650	61,658	0	30,992	63,363	20	190	92	13/14
	+2,334	*+2,392*	=	*-58*	*-582*	=	=	=	
	+2.5%	*+4.0%*	=	*-0.1%*	*-0.9%*	=	=	=	

Table 7.5: Comparison of resource usage after implementation between original SystemVerilog and two Chisel equivalent translations

7.3 Integrating Pipeline Framework

7.3.1 Implementation

The Chisel code resulting from abovementioned iterative translation process stands as reference through this section. Based on this *Vanilla Chisel* version of the *tree filters* hierarchy, we aim at implementing a pipeline-oriented version of this hierarchy with our pipeline automation framework. While the pipeline design methodology associated with this framework is detailed in Chapter 5, this section intends to demonstrate its relevance on a large industrial application. As detailed in Section 7.1, *tree filters* architecture is a natural candidate for pipeline-oriented implementation because it heavily relies on complex signal synchronization across its hierarchy.

7.3.1.1 Description Conciseness

```
// input signals
  input            in_compute,
  input            in_result,
  input tf_word_t  in_cfg,
/*****************************************
 * several in_cfg usage in
 * combinational computations
 *****************************************/

// explicit propagation with shift-registers
shift_register #(
  // ISSUE #1: user-managed synchronization
  .DEPTH(SHIFTER_LENGTH+1),
  // ISSUE #2: user-managed widths
  .WIDTH($bits(tf_word_t) + 2),
) cfg_shift_reg (
  .clk(clk),
  .in_data({in_cfg, in_result, in_compute}),
  .out_data({sr_cfg, sr_result, sr_compute})

/*****************************************
 * several computation stages
 * providing sft_value signal
 *****************************************/
// ISSUE #3: additional signal declarations
logic      node_prev_result;
logic      node_compute;
tf_word_t  node_cfg;

logic      node_result = '0;
always_ff @(posedge clk) begin
  // computation requiring synchronization
  node_result <= sr_cfg.mask & sft_value
  // explicit forward of unmodified signals
  // with on-the-fly renaming
  node_prev_result <= sr_result;
  node_compute     <= sr_compute;
  node_cfg         <= sr_cfg;
end
/** final stage using all node_* signals */
```

Listing 7.18: Original SystemVerilog with explicit signal propagations

```
// pipeline aware naming to guarantee SSA
  val prevCompute = Bool()
  val prevResult  = Bool()
  val cfg = new TreeFilterWordT
/*****************************************
 * several in_cfg usage in
 * combinational computations
 *****************************************/
val pipe = Pipe(in, "main")
// no need for explicit propagation

// on-demand synchronization of signals

// width-inference based on reflexivity

/*****************************************
 * several computation stages
 * providing sftValue signal
 *****************************************/
// no noisy forwarded signal declarations


pipe.node = (p, n) => {
  n.currentResult = Step.RegInit(p, false.B)


  n.currentResult := p.cfg.mask & p.sftValue
}
// no need for explicit signal forward


/** final stage using previous signals */
```

Listing 7.19: Pipeline-oriented Chisel version with implicit signal propagations

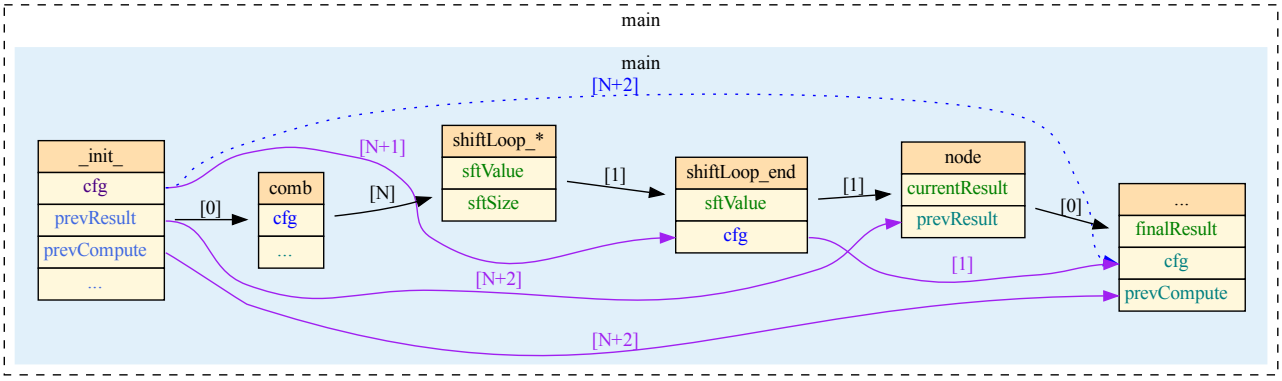


Figure 7.4: Pipeline representation generated by the framework (edited for conciseness)

Listing 7.18 presents an explicit signal propagation to match such synchronization requirements as originally implemented in the SystemVerilog version. For the sake of conciseness, the SystemVerilog version is preferred here over the translated Chisel version due to the verbosity of the translation of concatenations within instance port maps. This snippet illustrates several pipeline stages:

Input stage l.1–4, *Only three signals considered here for conciseness*

Initial combinational stage l.5–8, *(omitted)*

Explicit signal propagation l.10–19, *Shift registers synchronizing inputs with computations*

Intermediate computation stages l.21–24, *(omitted)*

Node stage l.26–39, *Requiring synchronization of previous computations with inputs*

Final stage l.40, *(omitted) Requiring signals from several previous stages*, line 40

The explicit signal propagation used in this implementation raises several issues, detailed as comments in the listing. The root cause for all these issues is the full responsibility of the developer to manage all details of the synchronization, which results in an error-prone and verbose description.

To address all these issues at once, Listing 7.19 illustrates the equivalent pipeline-oriented description of the same processing stages. Based on the explicit `Pipe` object, this implementation natively solves the synchronization requirements between computation stages, without requiring explicit signal propagation. As a result, this refined version is focused on actual processing operation, and freed of noisy statements whose sole purpose is to guarantee proper synchronizations. Applied to the complete module, it incidentally reduces the amount of lines of code by 20%. Omitted signal propagations are then programmatically implemented, allowing elaboration-time parameterizations and experiments with various synchronization strategies.

Based on a higher-level abstraction, the framework is able to provide an enriched representation of pipeline stages and theirs signals, focused on synchronization requirements. Figure 7.4 presents such a generated representation, manually edited to match the stages presented in Listing 7.19.[9] In particular, generated figures do not include any literal labels

[9]Full version of the pipeline representation of the current module it depicted in Appendix C.3.

and merged stages, as the framework only accesses elaborated values and treat each stage independently.

7.3.1.2 Off-the-shelf Resource Usage Optimization of Signal Propagations

This pipeline-oriented representation highlights a potential optimization natively provided by the framework. The purple arrows illustrate the respective synchronization requirements for the three signals this snippet focuses on: `cfg`, `prevCompute` and `prevResult`. More specifically, `cfg` requires $(N+1)$-cycle propagation from *init* to *shiftLoop_end*, whereas `prevCompute` and `prevResult` requires $(N+2)$-cycle propagation to other stages. The original SystemVerilog implementation groups them for a first propagation across the intermediate computation stages (*shiftLoop* stages with $N+1$ total delay in Figure 7.4). Considering the case of input signal `in_compute` in the SystemVerilog version, it is first delayed through explicit shift registers and comes out as `sr_compute`. Then, it is explicit forwarded with the *node* stage as `node_compute`.

In the pipeline-oriented description which enforces Single-Static Assignment, these successive copies of the same input signal correspond to the `prevCompute` signal. Unlike in the SystemVerilog implementation, the framework does not group signals for propagation. It implements a dedicated propagation for each purple arrow as they appear on Figure 7.4, with the corresponding delay.

Different hardware resources might be inferred by synthesis tools for both implementations. Considering Vivado synthesis in the SystemVerilog case, a wide shift register will be inferred for the grouped propagation, then individual registers will be inferred for additional explicit forward of `sr_compute` and `sr_result`. On target FPGA architecture, a single *SRL* primitive is able to implement shift-registers of up to 16-cycle depth, leading to a non-linear resource usage. Moreover, the last register of inferred shift-registers is extracted and implemented with a standard Flip-Flop (FF) to avoid impacting the timing of combinational operations connected to shift-register output. As a result, the actual resource usage of a shift-register of width W and depth $n \leq 16$ is W SRL + W FFs. With respective widths of W_{cfg}, W_c and W_r bits for `cfg`, `compute` and `result` signals, the total resource usage of the manual SystemVerilog propagations is given as follows:

$$\begin{array}{rc} \textit{grouped propagation} & (W_{cfg}+W_c+W_r)\times(1\,\text{SRL}+1\,\text{FF}) \\ \textit{individual registers} & W_c\,\text{FF}+W_r\,\text{FF}+W\,\text{FF} \\ \hline \textbf{Total} & (W_{cfg}+W_c+W_r)\,\text{SRL}+2(W_{cfg}+W_c+W_r)\,\text{FF} \end{array} \tag{7.1}$$

Due to shift-register inference for the grouped propagation within a submodule, the subsequent explicit forward of `compute` and `results` signals requires a flip-flop which cannot be absorbed as part of the shift-register. As a result, two successive FFs are used for propagation of `compute` and `results` signals after the explicit SystemVerilog propagation. To optimize resource usage of this snippet, an individual shift-register should be described for each signal, which would result in an even more verbose and tangled SystemVerilog implementation. With the framework and its individual signal propagations, this optimization is natively guaranteed, sparing (W_c+W_r) FFs in the current case:

$$\begin{array}{rc} \textit{cfg} & W_{cfg}(1\,\text{SRL}+1\,\text{FF})+W_{cfg}\,\text{FF} \\ \textit{compute} & W_c(1\,\text{SRL}+1\,\text{FF}) \\ \textit{result} & W_r(1\,\text{SRL}+1\,\text{FF}) \\ \hline \textbf{Total} & (W_{cfg}+W_c+W_r)\,\text{SRL}+(2W_{cfg}+W_c+W_r)\,\text{FF} \end{array} \tag{7.2}$$

While the savings remain limited here, such independent signal propagation management is able to provide considerable savings in larger hierarchies. Moreover, sparing a single

flip-flop at the right place can have a substantial impact on routing and ability to meet timing requirements for highly-congested areas of a design.

Last but not least, the pipeline-oriented implementation remains very close to the coarse-grained architecture, as Figure 7.2a illustrates in Section 7.1. As a result, the implementation preserves the intended flexibility of this stage-based design while exploration and optimizations are made easier. Modification of the processing part does not require dealing with the intricate data forwarding, and improvements of the latter can be independently achieved with appropriate implementation strategies. In particular, it allowed us to explore custom data forwarding solutions, one of which showcases the ability to improve resource usage by using FIFOs instead of shift registers for some signal propagations. This improvement was designed and implemented without requiring any update of the pipeline-oriented description, whereas the equivalent exploration on the SystemVerilog implementation would have been laborious and error-prone.

7.3.2 Initial Synthesis Results

Extending the results presented in Section 5.5, this part details various resource usage obtained after synthesis of the pipeline-oriented implementations of the *tree filters* architecture. The *Vanilla Chisel* version serves as the reference to evaluate the impact of the framework and its multiple implementation strategies. While Section 7.2.3 concludes with a noticeable overhead of the latter against the original SystemVerilog implementation, the framework does not intend to compensate existing translation discrepancies.

Three implementation strategies are experimented here, based on the corresponding synchronization algorithms presented in Section 5.4 and summarized as follows:

Peer-to-Peer Starting with the last stage, each missing signal is requested to immediate upstream stages. Propagations are implemented with the same hardware primitives as the existing relation between the current stage and its source, i.e. generally either simple wire or a register.

Direct Also starting with the last stage but based on knowledge of missing and equivalent relations, for each missing signal it implements a direct latency- and protocol-equivalent relation from the closest upstream source. The hardware primitive is chosen depending on further parameterization of the strategy. As the *tree filters* hierarchy does not require protocol signaling and relies only on constant-latency relations, we experimented with the following configurations:

- **SRL** Implement all constant latency paths based on Chisel-generated shift-registers, regardless of their depth and width of the signals.
- **FIFO** Implement constant latency paths with a FIFO and a counter as soon as the equivalent depth reaches 6 cycles and the signal width is larger or equal to 16 bits; default to shift-registers otherwise.
- **TSR** Leverage a custom shift-register primitive *TappedShiftRegister* (TSR), either relying on inference for Xilinx FPGAs or based on vendor primitives for Intel FPGAs.

As abovementioned, the synthesis results presented in this section target two distinct FPGA architectures which are currently deployed in production at OVHcloud and implement the same device, except for performance:

- Xilinx *Virtex Ultrascale+* at 200 MHz, for a 400 Gbps throughput. *(Vivado 2018.2)*
- Altera/Intel *Stratix V* at 150 MHz, for a 40 Gbps throughput. *(Quartus 15.1)*

	∑ LUT	LLUT	LUTRAM	SRL	FF	xRAM[10]
Vanilla Chisel	8,962	5,480	0	3,482	8,166	91
Peer-to-peer	8,774	5,306	0	3,468	8,159	91
Direct SRL	8,756	5,288	0	3,468	8,161	91
Direct TSR	8,838	5,370	0	3,468	8,150	91
Direct FIFO	7,898	5,642	1824	432	5,301	91
Peer-to-peer	-2.1%	-3.2%	=	-0.4%	-0.1%	=
Direct SRL	-2.3%	-3.5%	=	-0.4%	-0.1%	=
Direct TSR	-1.4%	-2.0%	=	-0.4%	-0.2%	=
Direct FIFO	-11.9%	+3.0%	∞	-87.6%	-35.1%	=

Table 7.6: Resource usage comparison for an 8-stage packet classifier on Xilinx Ultrascale+

	∑ LUT	LLUT	MLUT	FF	M20k	ALM
Vanilla Chisel	10964	7372	3592	4853	64	7116
Peer-to-peer	5044	5044	0	27294	64	8607
Direct SRL	5061	5061	0	27280	64	8625
Direct TSR	11031	7437	3594	4821	64	7311
Direct FIFO	7865	7865	0	8873	132	5464
Peer-to-peer	-54.0%	-31.6%	-100%	+462%	=	+21%
Direct SRL	-53.8%	-31.3%	-100%	+462%	=	+21%
Direct TSR	+0.6%	+0.9%	+0.06%	-0.7%	=	+2.7%
Direct FIFO	-28.3%	-6.7%	-100%	+82.8%	+106%	-23%

Table 7.7: Resource usage comparison for an 8-stage packet classifier on Intel Stratix V

We first discuss synthesis results for the Xilinx *Virtex Ultrascale+* FPGA. While the current industrial-scale versions leverage between 40 and 60 filtering stages, to ensure resource usage readability, Table 7.6 presents resource usage of an 8-stage classifier. In accordance with the original SystemVerilog implementation, this pipeline-oriented version is here generated without protocol signaling.

Xilinx toolchain reports three categories of LUT: Logical LUT (LLUT), LUTRAM and Shift register LUT (SRL). The last two are internally sharing the same *Memory LUTs* resource. The sum of these three primitives, reported in the leftmost column, is a relevant comparison criterion between resource usage. Indeed, Xilinx suggests maintaining the ratio between LUTs and FFs around 1 for best placement and routing performance. All various categories of block memories are here summarized as *xRAM* for conciseness as their quantity and distribution is the same for all implementations.

Due to Vivado default SRL inference, the three first strategies (*Peer-to-Peer*, *Direct SRL* and *Direct TSR*) show very little difference against the original implementation. The slight improvement is explained by the flattened hierarchy of the tree stages with the framework, which enables additional signal optimizations.

However, this flattened hierarchy does not benefit in the same way to Intel toolchain. Table 7.7 illustrates this behavior after the fitting step with *Quartus Prime 20.1* of the same 8-stage classifier, with no protocol signaling, targeting *Intel/Altera Stratix V* architecture.

[10] *xRAM* sums RAMB18, RAMB36 and URAM primitives based on the following equivalences: $1\,\text{RAMB18} = 1\,\text{xRAM}$; $1\,\text{RAMB36} = 2\,\text{RAMB18}$; $1\,\text{URAM}(4k \times 72b) = 8\,\text{RAMB36}(4k \times 9b)$.

The only equivalent comparison point with Vanilla Chisel in this table is the *Direct TSR* strategy, which shows a slight increase in resource usage. The excessive use of registers (FFs) with *Peer-to-peer* and *Direct SRL* strategies is due to the absence of shift registers inference. This default behavior of Quartus synthesis experimentally leads to routing issues. The rightmost column of the table reports an estimate number of logic elements (ALMs) actually used on the FPGA. A lower value means a reduced circuit footprint and more efficient resource packing. Despite a much higher LUT count, the *Direct TSR* strategy eventually appears much easier to route within a complete network device.

Last but not least, on both targets, the *Direct FIFO* strategy exhibits very different resource usage. On Intel, it highly reduces the overall logic element usage at the cost of additional memory blocks. On Xilinx, it also reduces the overall footprint, even with FIFO implemented as LUTRAMs, requiring less memory LUTs (2256 against 3482) and fewer registers. Interestingly, the *Direct FIFO* strategy balances the resource usage distribution between the various FPGA primitives. This preliminary result calls for further experiments with finer-grained parameterization of this strategy to fully expose its ability to generate a wide range of architecturally-specific implementations.

7.3.3 Further Architectural Parameterization

Controlling the balance of resource usage distribution between the various FPGA primitives is a very powerful capability which can help with timing closure of complex designs. Such control enables the designer to take full advantage the recommendation provided by FPGA vendors, either generically in their documentation or through advanced synthesis reports of a given design. As the *Direct FIFO* strategy already demonstrates the ability to implement the same functionality—from the exact same pipeline-oriented description—with a very different distribution of resource usage, this last experiment intends to further explore its parameterization to offer a wide range of resource usage distributions. In particular, the goal of this experiment is to provide parameterizations able to balance the use of LUTRAMs against SRLs in the context of Xilinx FPGA synthesis.

The *Direct FIFO* strategy introduces two configurable thresholds above which a given signal propagation will be implemented with a FIFO rather than shift registers.

1. Minimum signal propagation depth threshold
2. Minimum signal width threshold

Leveraging minimum thresholds is a first naive approach which is justified by the non-linear resource cost of FIFOs in terms of both depth and width. FIFOs are indeed based on memories which must be selected from a restricted set of *(depth, width)* couples, and as further detailed below, they require additional protocol signals which increase their resource usage.

7.3.3.1 FIFO Implementation as Constant Latency Path without Protocol Signaling

FIFOs (queues) are tailored to act as variable-latency paths, as they provide two independent interfaces, *write* (*enqueue*) and *read (dequeue)* interfaces. In that regard, they are often used as buffers to compensate or synchronize with other variable-latency paths, e.g. to accept a back-pressure at the output of a memory which itself does not accept such a back-pressure. To play this role, such variable latency paths require protocol signaling to synchronize with one another. In the case of FIFO, the *write* interface is connected to the upstream protocol signals and the *read* interface is connected to the downstream protocol signals as follows:

```
upstream.ready := !fifo.full
fifo.write_enable := upstream.valid
```

```
downstream.valid := !fifo.empty
fifo.read_enable := downstream.ready
```

From then on, if we attempt to remove the protocol signaling around this FIFO, considering that the circuit is uniformly exempt of back-pressure and its data is always valid, the following connections are observed:

```
fifo.write_enable := true.B
fifo.read_enable := true.B
```

As soon as the circuit powers up[11], this results in a continuous flow of data through the FIFO with a 1-cycle latency[12]between its *write* and *read* interfaces, regardless of the depth of the FIFO. To create a constant N-cycle latency path with such FIFO, we need to delay the `fifo.read_enable` activation by N cycles. Such behavior can be obtained with a register `read_start`, initially reset to the value N and decremented at each cycles until it reaches and maintains the value 0. The read interface is then connected as follows:

```
fifo.read_enable := read_start === 0.U
```

This results in an equivalent behavior to a shift-register of N-cycle, but it is described based on a FIFO primitive and thus uses different FPGA primitives.

As a side note, until this point the FIFO is presented with fully decoupled interfaces, enforcing at least 1-cycle delay between its input and output interfaces and requiring a FIFO depth of at least $N + 1$. Maintaining the independence of both interfaces is convenient to cut the path of back-pressure signals and reduce their fan-out, especially when back-pressure is combinational and drives numerous register enables. However, without these protocol signaling constraints, it is possible to couple both interfaces such that a full FIFO will still be able to accept a *write* request if there is also a *read* request at the same cycle. From an external point of view, this results in the following coupling of back-pressure signals:

```
upstream.ready := !fifo.full || downstream.ready
```

With such a behavior, it is possible to operate the FIFO without protocol signaling, only relying on the `full` signal provided by the FIFO itself as follows:

```
fifo.read_enable := fifo.full
```

Nonetheless, as the internal implementation of `full` signal usually relies on a counter driven by both `read_enable` and `write_enable`, this second approach has an equivalent resource overhead. As a result, the first approach is preferred as it makes no assumption on the FIFO behavior, and hence can be confidently used with any generic or vendor-locked FIFO primitive.

As previously illustrated in Tables 7.6 and 7.7, leveraging these constant latency FIFOs enables balancing resource distribution between SRL and LUTRAMs in large designs. To further demonstrate the flexibility of this FIFO strategy, a fine-grained parameterization of its two minimum thresholds is to be explored, but it first requires an analysis of the signal propagation depths and widths through the design. Beyond minimum thresholds, more advanced parameterization strategies could be proposed, but also require this analysis to sketch relevant rules.

7.3.3.2 Exhibiting Exploration Opportunities

Based on the figures reported by the framework, Table 7.8 details the occurrences of the respective *(depth, width)* couples of all direct signal propagations being implemented by the

[11]Or is reset, in such case, `true.B` should be read as `!reset` for an active-high reset.

[12]1-cycle latency is achieved with simple memory implementation. Without loss of generality, this latency might be increased as long as this higher latency remains below the signal propagation depth.

Specification		Occurrences		
Depth	Width	TfStageCompute	TfTop	Total
4	64	1		7
5	1	2	1	15
6	12		1	
6	62	1		7
6	264		1	
7	6		6	6
7	12		1	1
7	132		6	6
7	246		7	7
7	264		1	1
60	1		1	1

Table 7.8: Distribution of direct signal propagations to be implemented in the design, as reported by the framework after pipeline elaborations

framework. The module *TfStageCompute*, appearing in each stage of the classifier with the exact same parameterization is elaborated only once as an independent pipeline, and is then instantiated 7 times in the final circuit.

7.3.3.3 Architectural Parameterization for Balancing Resource Usage

From the previous report, generating the relevant parameterization to obtain various resource usage is as simple as manipulating these numbers to produce the corresponding *(depth, width)* minimum thresholds:

```
val relevantThresholds = Seq(
  4 -> Seq(1, 6, 12, 62, 64, 132, 246, 264),
  5 -> Seq(1, 6, 12, 62, 132, 246, 264),
  6 -> Seq(1, 6, 12, 62, 132, 246, 264),
  7 -> Seq(1, 6, 12, 132, 246, 264),
  10 -> Seq(1, 2),
).flatMap { case (depth, sWidth) => sWidth.map(w => depth -> w) }
```

These thresholds are then used as parameterization of the strategy applied to the pipeline-oriented classifier, and generates in a few minutes, the 30 corresponding Chisel-generated Verilog hierarchies. A simple script is then sufficient to generate the parameterization, to launch the corresponding synthesis and to report their respective resource usage, unattended and in a matter of hours.

Table 7.9 presents the result of this overnight exploration study after synthesis on the abovementioned Xilinx toolchain. Results are ordered by ascending count of LUTRAMs whose presence corresponds to FIFOs. As expected, the resource usage of the highest thresholds *(10, 2)*—which match none of the signal propagation in this design—is precisely equal to the resource usage of *Direct SRL* strategy previously reported in Table 7.6. Similarly, resource usage for the default parameterization *(6, 16)* of the *Direct FIFO* strategy is equal to the *(5, 62)* parameterization as both match the same signal propagations for this design. Interestingly, this default parameterization is close to the median value in terms of LUTRAM count which confirms the relevance of these naively and arbitrarily selected default thresholds for the current design.

With this exploration of architectural parameterizations, our pipeline automation framework demonstrates its ability to provide both guidance – to select the relevant parameteriza-

Depth				Width	LUTRAMs	SRLs	Logic LUTs	∑LUTs	FFs	RAMB18
			10	2	0	3468	5288	8756	8161	14
			DirectSRL		*0*	*3468*	*5288*	*8756*	*8161*	*14*
			10	1	6	3466	5231	8703	8307	14
		7		264	88	3336	5393	8817	8065	14
4	5	6		264	240	3072	5401	8713	7757	14
		7		246	952	1860	5451	8263	6649	14
4	5	6		246	1104	1596	5454	8154	6339	14
		7		132	1432	1068	5537	8037	5910	14
		7		12	1440	1062	5546	8048	5839	14
		7		6	1488	1026	5599	8113	5853	14
		7		1	1494	1024	5625	8143	5871	14
4	5	6		132	1584	804	5547	7935	5587	14
	5	6		62	1824	432	5642	7898	5301	14
Default DirectFIFO (d6;w16)					*1824*	*432*	*5642*	*7898*	*5301*	*14*
	5	6		12	1840	414	5682	7936	5313	14
4				64	1864	454	5862	8180	5342	0
	5	6		6	1888	378	5716	7982	5319	14
		6		1	1894	376	5752	8022	5347	14
	5			1	1922	362	5917	8201	5482	14
4				62	2104	82	5950	8136	5065	0
4				12	2120	64	5971	8155	5054	0
4				6	2168	28	6024	8220	5074	0
4				1	2202	12	6225	8439	5237	0

Table 7.9: Resource usage for the relevant threshold parameterizations

tions – and efficiency – to generate different pipeline implementations with a considerable impact on resource usage. After synthesis of the various implementations, their resource usage is widely spread across the design space which provides a great flexibility to the designer. In particular, with these parameterizations, the ratios of LUTs against FFs can be tuned, which might help the routing step with timing closure.

Finally, Table 7.9 reveals the odd behavior of *Vivado 2018.2* which infers a BRAM-based shift-register for a *(5,64)* shift-register, effectively swapping 64 SRLs for 2 BRAMs. This behavior is altered with *Vivado 2019.1*[13]which results in a major difference in the ability to meet timing requirements on some of our larger designs. Based on the exploration capabilities of the framework, this barely documented difference is not only highlighted, but the framework also provides means to compensate for it. In this precise case, this would be achieved by implementing 64-bit shift-register-based propagations with explicit Xilinx BRAM primitives (instantiated as *black-boxes*) to recover the previous behavior.

While raising the level of hardware abstraction to a convenient pipeline-oriented modeling, our framework also increases designers' control over the finest architectural details of their designs.

7.4 Conclusion

This chapter delves into the design of a complex packet classifier, which is a core function of network devices. From a simple high-level architecture based a succession of independent stages, its original SystemVerilog implementation requires fine-grained signal synchronizations all along the processing pipeline. As a result, its verbose and tangled SystemVerilog description appears unflexible and hard to maintain, defeating the original purpose of stage independence.

To overcome these limitations by leveraging the pipeline automation framework, the complex *tree filters* SystemVerilog hierarchy first needs to be translated into a functionally equivalent Chisel hierarchy. Based on *sv2chisel*, this word-for-word translation process turns out to be challenging and requires a substantial effort to first improve the automated translation tool, and then manually iterate on the result. Several areas for improvement remain in this *Vanilla Chisel* version, to reduce overhead on the development flow and resource usage after synthesis.

Nonetheless, the pipeline-oriented description of this packet classifier delivers considerable flexibility, enabling experiments with new signal propagation strategies at no redesign cost, simply as a new parameterization of the pipeline generation. Without these automation capabilities, the exotic FIFO strategy, which appears surprisingly relevant in our case, would have been ruled out without further consideration due to the amount of work needed to implement it.

Integration of Chisel version within its surrounding SystemVerilog hierarchy is required early in the process to validate its functionality within existing test suites. Thanks to the wrapper generators, automatically configured by *sv2chisel*, it happens effortlessly and requires very few manual adjustments.

While uncovering several challenges and associated areas for improvement in the translation process, this chapter demonstrates the relevance of HCLs such as Chisel to base high-level hardware abstractions. In particular, our Chisel-based pipeline design methodology exhibits enhanced design and implementation flexibility, while increasing the control over the generated architectures to its finest level of detail.

[13]Reportedly starting with *Vivado 2018.3*.

Chapter 8

Conclusion

The overall goal of this thesis consists in evaluating the ability of Hardware Construction Languages (HCLs) to abstract hardware architectures, either natively or as a platform for additional abstractions. Experimenting with Chisel HCL [BVR+12], this thesis focuses on the relevance of such abstractions to design highly flexible and re-usable high-performance network devices at OVHcloud. The detailed problem statement concludes on five precise questions which have driven the research effort, and to which we have attempted to answer along this manuscript. We now summarize the solutions and perspectives brought by our contributions to each of these questions, starting with the abstractions inherently featured in HCLs, as follows:

How far can the abstraction level provided by HCLs instill agility into hardware design flow while preserving fine-grained control over implementation?

Our detailed review of HCL principles and their various implementations in Chapter 3, first exhibits the control guarantees which arise from the very nature of parameterized hardware generation, as opposed to architecture inference. This answer is supported by numerous developments which rely on HCLs to generate complex, reusable and highly flexible designs while maintaining a close control on the generated architectures as illustrated in Chapter 4. Anchored in this research effort, our first contribution depicted in Section 4.2, demonstrates the agility of successive design iterations to build a configurable associative memory. The iterative HCL-based implementations match the performance of their SystemVerilog counterparts, with similar resource usage after FPGA synthesis.

Bolstered by this conclusive preamble, which showcases the efficient tools brought by HCLs to implement agile hardware architectures, we next question their ability to impact development flows as early as preliminary design phases.

How can this abstraction level significantly remodel the design of hardware circuits from the earliest stages?

Our second contribution presented in Section 4.3 illustrates how the application of an existing software development paradigm, in the design of a two-stage associative memory, enables decoupling architecture implementation from functionality. Based on an Abstract Data Type (ADT) interface, the exact implementation can be re-used with arbitrary functionalities, pushing the parameterization to a complete new level of abstraction. This software pattern is natively supported by embedded HCLs such as the Scala-embedded Chisel thanks to the high-level paradigms provided by their host languages, such as functional and object-oriented programming (OOP).

However, direct application of such software-oriented paradigms to hardware design comes with some limitations due to their original nature, which raises our next question:

What higher-level hardware design abstractions and paradigms can be relevant to improve agility of hardware design?

A close examination of the multi-layered hardware abstractions in Chapter 3 reveals that despite a continuous research effort, their evolution have stalled for decades at the register transfer level (RTL). Emergence of mature HCLs and inspirations from the evolution of software abstractions have paved the way for viable hardware-oriented abstractions, aiming at iterative improvements of hardware development flows.

Our core contribution, detailed in Chapter 5, arises from repeated observations of verbose, inflexible and error-prone RTL patterns as expressed within traditional Hardware Description Languages (HDLs). In particular, we focus on the register stage pattern which stands at the heart of pipelined architectures, and is a desired implementation pattern to build high-performance streaming applications from network to signal processing. To overcome the limitations of this pattern, we introduce a pipeline design methodology, based on explicit pipeline objects and a graph modelling of signals and their respective relations. Thanks to configurable resolution of signal synchronizations, our pipeline design methodology provides highly flexible descriptions, able to generate finely tuned implementations for various targets.

This design methodology could be implemented in the form of a brand-new *pipeline description language*, with the pre-processing of customized HDLs, or any other mean detailed in Chapter 3. However, as part of the iterative improvement of hardware development flows, HCLs appear as a natural starting point, which the next question explores:

How can HCLs provide the foundations to introduce and implement these additional abstraction levels?

From our initial exploration of Chisel parameterization capabilities in Section 4.2 to ADT-based design in Section 4.3 to implementation of pipeline design methodology in Chapter 5, HCLs demonstrate their ability to stand as foundations of further abstractions. Throughout experiments with the Scala-embedded Chisel HCL, our implementations leverage the features of the host language and specifically object-oriented and functional programming.

Last but not least, as newcomers in hardware development flows, the ability to integrate HCLs within existing hierarchies is discussed by the final question:

How to integrate HCLs generation-based flow into large existing code-bases and their established hardware development flow?

Design methodologies and tools are designed to address domain-specific issues, no matter how wide these domains might be. As a result, they rarely focus on their interoperability with existing solutions, or only as leading from a top-level perspective. Our two final contributions detailed in Chapter 6 tackle this issue by providing upstream and downstream integration of Chisel with existing SystemVerilog hierarchies. On the one hand, we introduce *sv2chisel*, an automated translation tool enabling a fast integration process of existing SystemVerilog hierarchies as functionally-equivalent Chisel ones. This translation is intended as a first step towards manually optimized Chisel generators, benefitting from all the flexibility and reusability detailed all along this manuscript. On the other hand, we smoothen integration of Chisel-generated hierarchies into top-level HDL hierarchies with automated wrapper generators to close the gap between parameterized HDL and low-level generated

Verilog. From our industrial perspective, guaranteeing a frictionless cohabitation between HCLs and HDLs is a key enabler toward the widespread adoption of HCLs and the enhanced hardware abstractions they deliver.

Finally, we condense our contributions in an integrated experimentation, detailed in Chapter 7. From a complex SystemVerilog packet classifier design, we first highlight the desperate need for an automated management of signal synchronizations such as the one our pipeline design methodology is able to provide. We then review the translation process of this carefully handcrafted SystemVerilog hierarchy into Chisel, based on *sv2chisel* and extended with manual iterations. We finally refactor this Chisel hierarchy, leveraging our pipeline management framework, and successfully integrate it back into the surrounding SystemVerilog top-level which defines one of our critical network device.

Perspectives

The adoption of HCLs within both academic and industrial contexts is gaining momentum. In particular, the Chisel/FIRRTL ecosystem is a mature and evolving alternative to traditional HDLs in many cases, able to provide real agility to hardware designers, and supported by a vibrant open-source community. Throughout this thesis, we contributed to some developments and discussions, and this manuscript is the proof that research efforts can be iteratively built on top of such massive pieces of engineering.

The scope of HCLs' recommended uses is widening, and many design domains are already able to benefit from their advanced functionalities. Based on existing or new paradigms, developments of custom abstractions are able to provide concrete improvements for hardware designers. Many software paradigms remain yet to be explored as potential hardware abstractions, based on the strong base layer provided by existing HCLs. To that regard, a comprehensive study of software paradigms with evaluation of their relevance as hardware design methodologies would certainly uncover valuable hardware abstractions.

Already providing pretty decent (System)Verilog to Chisel translations, our *sv2chisel* translator keeps track of a long list of desired features; beginning with an enhanced support of several SystemVerilog constructs from blocking assertions to pre-processor directives. Beyond simple word-for-word translation, the support of these statements requires a fine-grained semantic analysis of the description to retrieve the design intent and translate it into an idiomatic form. In particular, translating looped blocking statements as recursive hardware generators is a conceptual challenge which requires a subtle mastery of both traditional circuit descriptions and software paradigms.

One significant challenge for *sv2chisel* project remains its open-source future activity and viability. As of today, it has raised limited interest and feedback, even if its sources, packages and binaries cumulate numerous downloads. To gain visibility and gather a community, a partnership with some existing open-source libraries eager to move away from SystemVerilog could help.

Designed with the same modularity as the Scala FIRRTL compiler, *sv2chisel* relies on a hardware IR supporting basic elaboration constructs, and a collection of independent transformations of the latter. To that extend, other input languages such as VHDL could be supported at the simple cost of a dedicated parser with proper mapping to *sv2chisel* IR. Similarly, targeting other backends seems perfectly reasonable at the cost of some adjustments in the IR and in the existing transformations. As detailed in Chapter 3, there are countless such potential HCL targets. More importantly, development of this hardware IR has required the integration of basic elaboration statements such as conditional hardware generation, *de facto* introducing one of the first somehow elaboration-aware hardware IR. Designing a hardware IR able to handle generic high-level elaboration constructs is another

conceptual challenge that would provide major benefits to hardware compiler frameworks and to the quality of the HDL they generate. In particular, retaining width parameterizations of busses and basic generation capabilities would promote the substantial work poured into elaboration-aware simulators and thus contribute to enhanced simulation speeds.

At the core of this thesis, our pipeline design methodology introduces an efficient hardware-oriented abstraction which could be extended in numerous ways. A first desired feature would be native pipeline generation constructs such as stage replication with proper namespaces and signal visibility control. Next, automated signal synchronization could be extended to pipelines which process data at various levels of granularity, such as bit and packet levels for network devices. Adding support for automated conversion from one level to another would greatly ease interaction between cross-level modules and enable enhanced architectural parameterizations. A direct application of such support would enable remodeling our current network pipeline architecture, as initially introduced in Figure 2.3. While retaining the same high-level representation, many optimizations could be handled by the framework, based on efficient signal synchronizations across the hierarchy. Last but not least, as our pipeline automation framework provides highly flexible architectural parameterizations, it could be paired with automated architecture exploration frameworks such as QECE [Fer22, FMR21]. With a fine-grained parameterization of resolution strategies, the resulting system would be able to automatically explore an extended architectural design space and provide optimized implementations.

Abstractions promoting flexibility and re-usability are key enablers for competitive hardware developments. In a context of required and desired energy efficiency, the thoroughness and precision required to build specialized circuits shall not hide their high performance per watt when compared to equivalent software solutions. The massive efforts poured into efficient hardware design of processors are too often wasted by deficient software relying on always more performant CPUs to support the cost of their unoptimized computations. As reconfigurable hardware acceleration platforms, FPGAs are undeniable assets in high-performance device design, certainly not restricted to the network perspective we have thoroughly explored in this thesis. Following one particularly promising software evolution trend, zero-cost hardware abstractions are the future of both energy-efficient and agile developments. By committing to decrease the overhead of custom hardware developments, we sincerely hope this work will boost their attractiveness and contribute to informed decisions across organizations. More than ever, it is a timely moment to take energy efficiency into account in an attempt to contain the carbon footprint of cloud providers and all industries relying on heavy computational power.

Appendices

Appendix A

Abstract Data Type Schemes

A.1 Complete *Protected Hash-Table* ADT Scheme

This appendix reproduces the complete Abstract Data Type (ADT) whose excerpt has been used to illustrate this methodology on the design of protected hash-table as introduced in section 4.3. While principles and limitations of ADTs are presented in the same section, this complete code disclosure provides a finer-grained look at the precise implementation. Inline comments detail the role of each method and attribute.

```
/** Specification of HashTable requests
 *
 *  HTRequest implementation are the request given to a module. Allow the usage of stateful
 *  request, of type S, using the getState to get the state of a request. By default a
 *  request will trigger a write to the memory, this mechanism can be deactivated using the
 *  doNotWrite function.
 *
 *  @tparam R
 *    underlying Data Type mixed into this trait
 *  @tparam S
 *    data type of the state of a request
 */
trait HTRequest[R <: Data, S <: Data] {
  this: R => // ensure it gets constructed mixed in a concrete Chisel Data Type

  /** returns the raw ID to be hashed for memory addressing */
  def getRawID: UInt // for Hash input

  /** returns the width of the rawID */
  def idWidth: Int // for Hash param

  /** returns true if the current request require a lock during memory access not required
   *  for example for a simple "read" operation
   */
  def doLock: Bool

  /** return the current state of the request, this is useless for a stateless request */
  def getState: S

  /** Indicate whether this request requires writing to the memory. */
  def doNotWrite: Bool

  /** returns true if the request has to go through without using external memory   */
  def ignore: Bool
}

/** Specification of Hashtable Stored DataElement
```

```
38    *
39    *  HTStoredData implementation are to be entirely stored into memory. If a function may
40    *  change an element it must return  an element of type Modifiable[_].
41    *
42    *  *Dev Warning*: HashTable implementation SHALL NOT, on purpose, take for granted that
43    *  the returned type for <prob|det|demote>Update functions is strictly E (it can have
44    *  been extended / overloaded).
45    *
46    *  See example below, assuming updated has been overloaded from a default value as in
47    *  RateCounterHTData:
48    *  {{{
49    *  val w : E = Wire(<genE>)
50    *  val tmp = <this>.<prob|det|demote>Update(req)
51    *  w := tmp
52    *  assert(res.updated === tmp.updated) // will trigger in some cases
53    *  }}}
54    *  Note : if this becomes to constraining for implementation we can upgrade this
55    *  DataScheme with an explicitly differentiated type for A:memory storage & B:data
56    *  manipulation where B is a direct extension of A
57    *
58    *  @tparam R
59    *    underlying Data Type mixed into HTRequest
60    *  @tparam S
61    *    underlying Data Type mixed into HTRequest
62    *  @tparam T
63    *    underlying Data Type mixed into this trait
64    *  @tparam U
65    *    simple Data Type used as output result type
66    */
67  trait HTStoredData[R <: Data, S <: Data, T <: Data, U <: Data] {
68    this: T => // ensure it gets constructed mixed in a concrete Chisel Data Type
69
70    /** D: full data type of request */
71    type D = R with HTRequest[R, S]
72
73    /** E: this concrete type deliberately not using this.type as it would prevent extending
74     *  return type as demonstrated in example implementation RateCounter & Determinist
75     */
76    type E = T with HTStoredData[R, S, T, U]
77
78    /** hardware function returning true.B if this elt is deterministic */
79    def isDeterministic: Bool
80
81    /** hardware function returning true.B if this element requires a deterministic lookup
82     *
83     *  Ex: would be false if threshold are not reached in a RateCounter Implem
84     */
85    def needsDeterministic(req: D): Bool
86
87    /** hardware function returning true.B if this element id equals request id */
88    def sameID(req: D): Bool
89
90    /** Allow the usage of sequential detUpdate function.
91     *
92     *  Indicate the number of clock cycles needed for the detUpdat function.
93     */
94    val detUpdateDelay: Int
95
96    /** Hardware function returning an updated copy of this in the case of a deterministic
97     *  update.
98     *
99     *  This function is either combinatorial or sequential. The en parameter must be the ready
100    *  signal where this function is called. Meaning that all the clocked processes must happen
101    *  when en is true.
102    *
103    *  Determinist updates happen in the following cases:
```

```
104     *    - probStage : this.isValid & this.isDeterministic & this.sameID(req)
105     *    - probStage : !this.isValid
106     *    - deterStage : before every write to memory
107     */
108   def detUpdate(req: D, en: Bool): Modifiable[E]
109
110   /** Allow the usage of sequential probUpdate function.
111     *
112     *  Indicate the number of clock cycles needed for the probUpdate function.
113     */
114   val probUpdateDelay: Int
115
116   /** Hardware function returning an updated copy of this in the case of a probabilistic
117     *  update.
118     *
119     *  This function is either combinatorial or sequential. The en parameter must be the
120     *  ready signal of the pipeline stage which call this function. Meaning that all the
121     *  clocked processes must happen when en is true.
122     *
123     *  probabilistic updates happen in the following cases:
124     *    - probStage : this.isValid & !this.isDeterministic & !this.sameID(req)
125     */
126   def probUpdate(req: D, en: Bool): Modifiable[E]
127
128   /** hardware function returning an updated copy of this in the case of a demote update
129     *
130     *  demote updates happen in the following cases:
131     *    - probStage : this.isValid & this.isDeterministic & !this.sameID(req)
132     */
133   def demoteUpdate(req: D): Modifiable[E]
134
135   /** hardware function returning the equivalent request for the current storedElt in the
136     *  case of a move to deterStage
137     *
138     *  This function is called on data to be demoted, if needsDeterministic(req), before
139     *  demoteUpdate(req)
140     */
141   def toDetReq: D
142
143   /** hardware function returning the result expected at output of HashTable Implementation */
144   def getResult(req: D): U
145
146   /** data type of the result expected at output of HashTable Implementation */
147   def getResultType: U
148
149   /** hardware function returning true.B if this is a valid memory entry */
150   def isValid(req: D): Bool
151
152   /** hardware dictionnary of functions used to clean an element, can be stateful */
153   val tidying: Map[String, S => Modifiable[E]]
154
155   /** data type of the state of a request if needed */
156   def getStateType: S
157 }
158
159 /** Utility class to indicate if a data has been modified or cleaned. A modified data will
160 *  be written in the memory. The cleaned bit should be only used for stats computation.
161 *  The cleaned bit must be up when the data erased one of the previous period.
162 */
163 case class Modifiable[+D <: Data](private val d: D) extends Bundle {
164   val modified = Bool()
165   val cleaned  = Bool()
166   val data     = d.cloneType
167 }
```

A.2 ADT Usage Example

The following code listing showcases an excerpt of a hardware architecture based on the ADT: the probabilistic stage of the protected hash-table introduced in section 4.3. The declaration of the module as a type-parameterized Chisel `class` retains the ability to be used with any ADT implementation, as illustrated on lines 20–28. Several ADT methods are presented within this excerpt such as `idWidth` on line 40, `getRawID` on line 42, `probUpdateDelay` on lines 59 and 64, `probUpdate` on line 69, and `getResult` on line 84.

```
1  /** Probabilistic stage of ProtectedHT.
2   *
3   *  This stage is based on a single hash, and is sensitive to collision.
4   *  It does not provide any collision resolution mechanism
5   *  (hence the probabilistic characteristic).
6   *
7   *  @param _addrWidth
8   *    Width of an address
9   *  @param _simultReq
10  *    Maximum number of simultaneous request
11  *  @param _reqGen
12  *    Request data type generator.
13  *  @param _storedEltGen
14  *    Stored element data type generator, used for the memory's data.
15  *  @param _resGen
16  *    Result data type generator.
17  *  @param genHash
18  *    Hash Module generator
19  */
20 class ProbabilisticHT[R <: Data, S <: Data, T <: Data, U <: Data](
21     val _addrWidth: Int,
22     val _simultReq: Int,
23     private val _reqGen: R with HTRequest[R, S],
24     private val _storedEltGen: T
25       with HTStoredElt[R, S, T, U],
26     private val _resGen: U,
27     val genHash: ((Int, Int, Int) => HashModule) = hashes.default,
28 ) extends Module {
29
30   val io = IO(new Bundle {
31     val req = Flipped(DecoupledIO(_reqGen.cloneType))
32     val res = DecoupledIO(_resGen.cloneType)
33     // External memory
34     val memReadReq  = Decoupled(new ReadReqFlow(UInt(_addrWidth.W)))
35     val memReadRes  = Flipped(Decoupled(new ReadResFlow(_storedEltGen.cloneType)))
36     val memWriteReq = Decoupled(new CommitFlow(UInt(_addrWidth.W), _storedEltGen.cloneType))
37   })
38
39   // Hash module provided as argument
40   val hash = Module(genHash(_reqGen.idWidth, _addrWidth, 3645))
41   hash.io.req.valid := io.req.valid
42   hash.io.req.bits := io.req.bits.getRawID
43   io.req.ready := hash.io.req.ready
44
45   // Shift register in parallel of hash-module (non-generic known depth of 2)
46   val hashExtra = ShiftRegister(
47       io.req.bits,
48       2,
49       hash.io.res.ready
50   )
51
52   val fetchMem = /* ... memory access logic omitted ... */
53
54   /* synchronization logic based on probUpdateDelay */
```

```
55
56    val updateReady = Wire(Bool())
57    val updateExist = ShiftRegister(
58        io.memReadRes.bits.data.isValid(fetchMem.bits.req),
59        io.memReadRes.bits.data.probUpdateDelay,
60        updateReady
61    )
62    val updateReq = ShiftRegister(
63      fetchMem.bits,
64      io.memReadRes.bits.data.probUpdateDelay,
65      updateReady
66    )
67
68    /* call to probUpdate */
69    val updateRes = io.memReadRes.bits.data.probUpdate(fetchMem.bits.req, updateReady)
70
71    /* excerpts of internal decision logic based on ADT methods */
72
73    // Decision
74    val dec = RegDecoupled(new Bundle {
75      val res            = _resGen.cloneType
76      val exist          = Bool()
77      val write          = Bool()
78      val addr           = updateReq.addr.cloneType
79      val stored         = updateRes.data.cloneType
80      val ignoreMemReady = Bool()
81    })
82    when(updateReady) {
83      dec.valid := updateValid
84      dec.bits.res := updateRes.data.getResult(updateReq.req)
85      dec.bits.exist := updateExist
86      dec.bits.write := updateRes.modified
87      dec.bits.addr := updateReq.addr
88      dec.bits.stored := updateRes.data
89      dec.bits.ignoreMemReady := updateReq.req.ignore
90    }
91    updateReady := dec.ready
92
93    io.memWriteReq.bits.data := dec.bits.stored
94    io.memWriteReq.bits.address := dec.bits.addr
95    io.memWriteReq.bits.memWrReq := dec.bits.write
96
97    // Send result
98    io.res.bits := dec.bits.res
99
100   // Split pipeline synchronization: dec --> io.res + io.memWriteReq
101   io.res.valid := dec.valid && (io.memWriteReq.ready || dec.bits.ignoreMemReady)
102   io.memWriteReq.valid := (dec.valid && !dec.bits.ignoreMemReady) && io.res.ready
103   dec.ready := io.res.ready && (io.memWriteReq.ready || dec.bits.ignoreMemReady)
104
105 }
```

Appendix B

Port Wrapper

This appendix presents the top level generator of the translated *tree filters* architecture as produced by *sv2chisel*, and which was then manually edited. In particular, it includes additional parameters on lines 8–11.

```
1  import sv2chisel.helpers.tools.{ChiselGenMain, VerilogPortWrapper}
2
3  object TfTopGen extends ChiselGenMain {
4    VerilogPortWrapper.emit(
5        () => new TfTop(),
6        renameWrapperPorts = Map("clock" -> "clk"),
7        forcePreset = true,
8        initialStatements = Seq("import globals_p::*;"),
9        args = args ++ Seq(
10         "--target:fpga" // enable MemInit in synthesis context (required for nodeMaskRom)
11       )
12   )
13 }
```

From this configuration, the following wrapper is generated, focusing here only on structural port mapping. In this example, the output port `out_label` is mapped from its structural type `tree_filter_label_t [5:0]` to the equivalent flattened output ports of the underlying instance (12 of them) on lines 52–63. The same behavior applies to signal `in_pkt` whose custom type `tree_filter_pkt_data_t` is a `packed struct` defined in the SystemVerilog package `globals_p`.

```
1  import globals_p::*;
2  module tf_top (
3      input clk,
4      output in_rdy,
5      input in_valid,
6      input tree_filter_pkt_data_t in_pkt,
7      output out_valid,
8      output [5:0] out_matched,
9      output tree_filter_label_t [5:0] out_label,
10     output ctrl_rdy,
11     input ctrl_commit,
12     input [17:0] ctrl_addr,
13     input ctrl_write,
14     input [63:0] ctrl_write_data,
15     input ctrl_read,
16     output ctrl_read_data_valid,
17     output [63:0] ctrl_read_data
18   );
```

```
19    tf_top_raw inst (
20      .clock(clk),
21      .in_rdy(in_rdy),
22      .in_valid(in_valid),
23      .in_pkt_is_portmap_res(in_pkt.is_portmap_res),
24      .in_pkt_is_portmap_req(in_pkt.is_portmap_req),
25      .in_pkt_is_ntp_monlist(in_pkt.is_ntp_monlist),
26      .in_pkt_is_http_res(in_pkt.is_http_res),
27      .in_pkt_known_src(in_pkt.known_src),
28      .in_pkt_is_ts3init(in_pkt.is_ts3init),
29      .in_pkt_is_dns_query(in_pkt.is_dns_query),
30      .in_pkt_is_http_req(in_pkt.is_http_req),
31      .in_pkt_is_fragment(in_pkt.is_fragment),
32      .in_pkt_status(in_pkt.status),
33      .in_pkt_icmp_code(in_pkt.icmp_code),
34      .in_pkt_icmp_type(in_pkt.icmp_type),
35      .in_pkt_valid_ipv6(in_pkt.valid_ipv6),
36      .in_pkt_valid_ipv4(in_pkt.valid_ipv4),
37      .in_pkt_valid_icmp(in_pkt.valid_icmp),
38      .in_pkt_valid_net(in_pkt.valid_net),
39      .in_pkt_valid_trans(in_pkt.valid_trans),
40      .in_pkt_net_size(in_pkt.net_size),
41      .in_pkt_size(in_pkt.size),
42      .in_pkt_ether_type(in_pkt.ether_type),
43      .in_pkt_tcp_flags(in_pkt.tcp_flags),
44      .in_pkt_profile(in_pkt.profile),
45      .in_pkt_proto(in_pkt.proto),
46      .in_pkt_dst_port(in_pkt.dst_port),
47      .in_pkt_src_port(in_pkt.src_port),
48      .in_pkt_dst_ip(in_pkt.dst_ip),
49      .in_pkt_src_ip(in_pkt.src_ip),
50      .out_valid(out_valid),
51      .out_matched(out_matched),
52      .out_label_0_prio(out_label[0].prio),
53      .out_label_0_value(out_label[0].value),
54      .out_label_1_prio(out_label[1].prio),
55      .out_label_1_value(out_label[1].value),
56      .out_label_2_prio(out_label[2].prio),
57      .out_label_2_value(out_label[2].value),
58      .out_label_3_prio(out_label[3].prio),
59      .out_label_3_value(out_label[3].value),
60      .out_label_4_prio(out_label[4].prio),
61      .out_label_4_value(out_label[4].value),
62      .out_label_5_prio(out_label[5].prio),
63      .out_label_5_value(out_label[5].value),
64      .ctrl_rdy(ctrl_rdy),
65      .ctrl_commit(ctrl_commit),
66      .ctrl_addr(ctrl_addr),
67      .ctrl_write(ctrl_write),
68      .ctrl_write_data(ctrl_write_data),
69      .ctrl_read(ctrl_read),
70      .ctrl_read_data_valid(ctrl_read_data_valid),
71      .ctrl_read_data(ctrl_read_data)
72    );
73  endmodule
```

Appendix C

Tree Filters Architecture Details

C.1 Top Level Architecture

Figure C.1 illustrates the overall architecture of the *tree filters* module. This representation has been manually reverse-engineered from the SystemVerilog implementation. The parameterization of the number of tree stages in the pipeline is based on a huge partially initialized table which provides indexed intermediate signals for each stage. To our knowledge, in SystemVerilog this is the only available approach to provide such configurable pipeline length. However, it is very verbose and error-prone due to the extensive manipulation of signal indexes.

C.2 Generated Synchronization-oriented Representations

To further understand the signal synchronization requirements within the *tree filters* hierarchy, we automatically generated a pipeline-oriented representation of the numerous modules, as illustrated in Figures C.2–C.6. As creating this representation requires a semantic analysis of the pipeline, we leveraged an *ad-hoc* modified version of our *sv2chisel* translator tool to generate these figures. In addition to the symbols detailed in Figure 7.3, Figure C.6 introduces a new symbol: diamond, which represents an optional register, depending on the parameterization of the module.

C.3 Pipeline-oriented Representation

Visualization of the pipeline stages and their relations is an interesting feature provided by the pipeline automation framework we introduced in Chapter 5. Figure C.7 illustrates the representation generated by the framework of the pipeline-oriented version of the module *TfStageCompute*. This representation features colored signals, following the color scheme detailed in Figure 5.5b, and highlights the dependencies in signal usage and their synchronization resolutions. Thanks to our framework, it provides much more information than the pipeline-oriented representation generated from the original SystemVerilog in Figure 7.3.

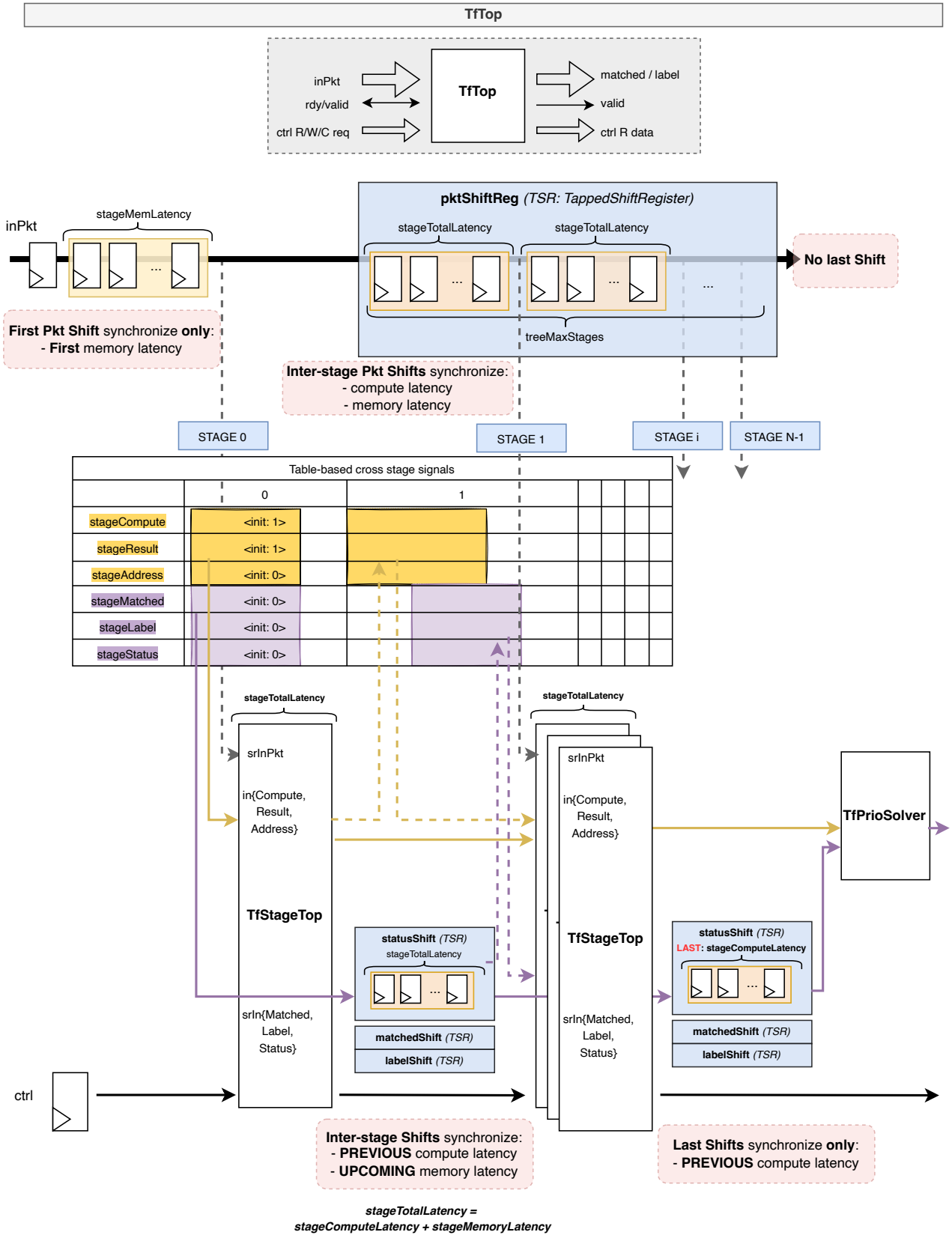


Figure C.1: Manual reverse engineering of Top-Level *Tree Filters* Hierarchy

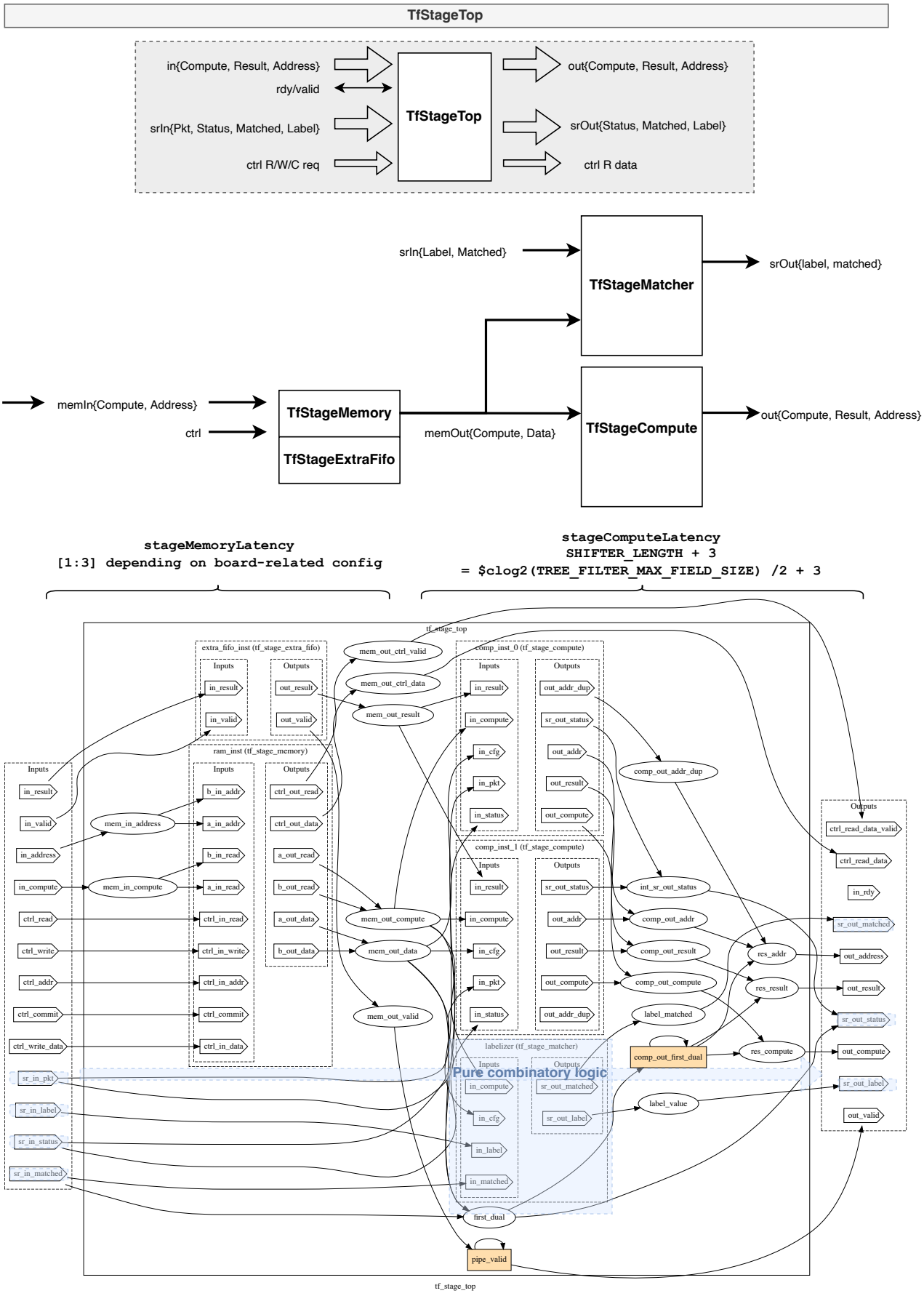


Figure C.2: Synchronization-oriented representation of the replicated stages of *Tree Filters* Hierarchy

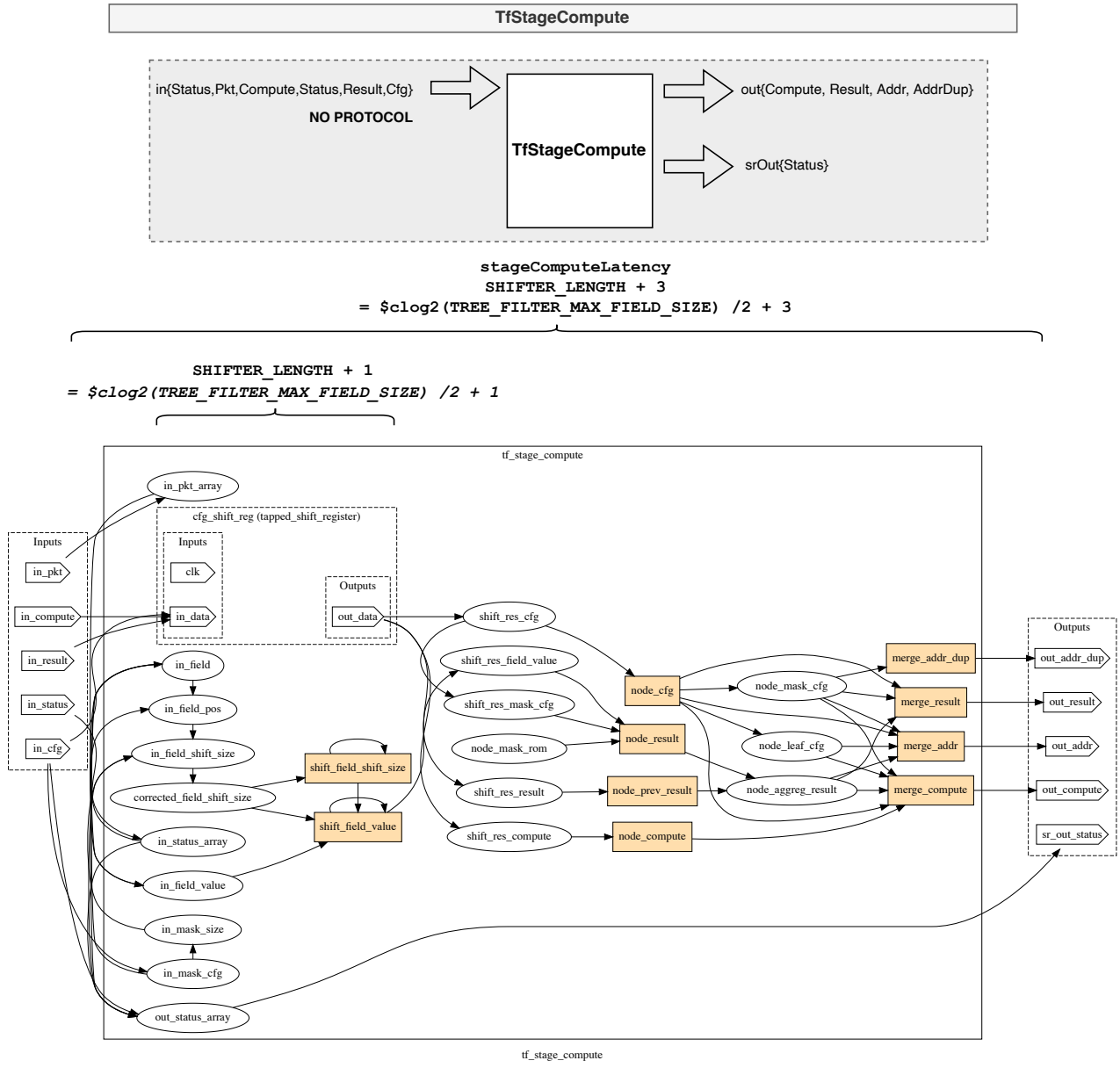


Figure C.3: Synchronization-oriented representation of the core compute submodule *TfStage-Top* of *Tree Filters* Hierarchy

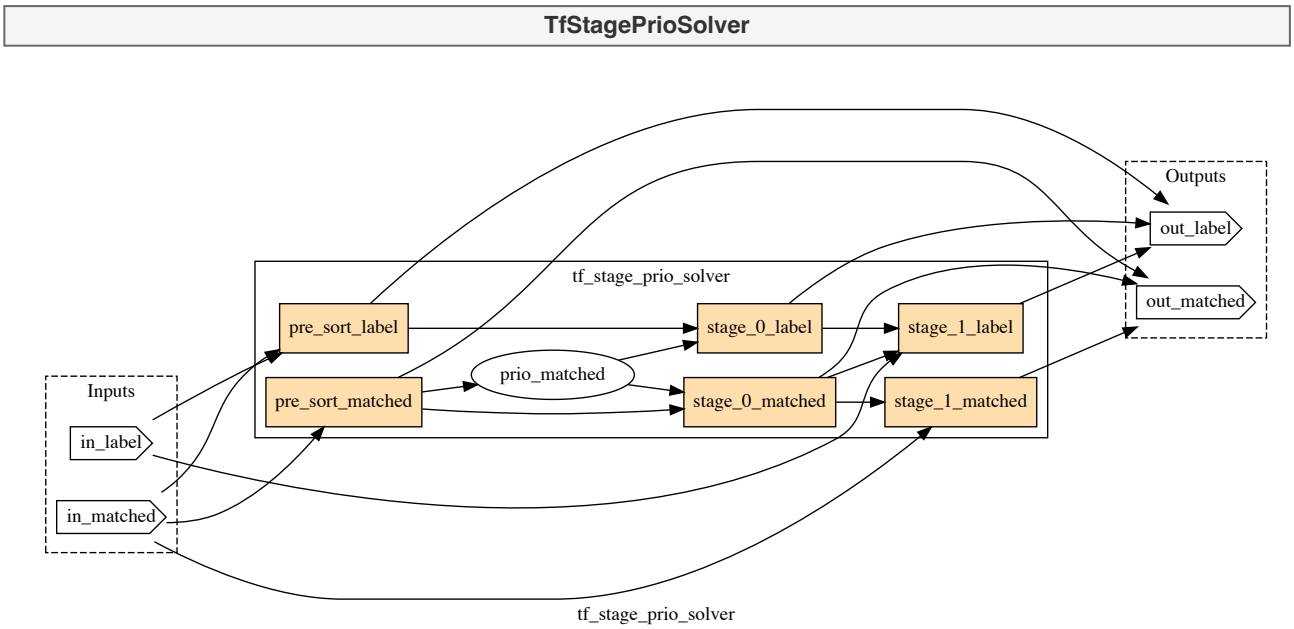


Figure C.4: Synchronization-oriented representation of *TfStagePrioSolver* module

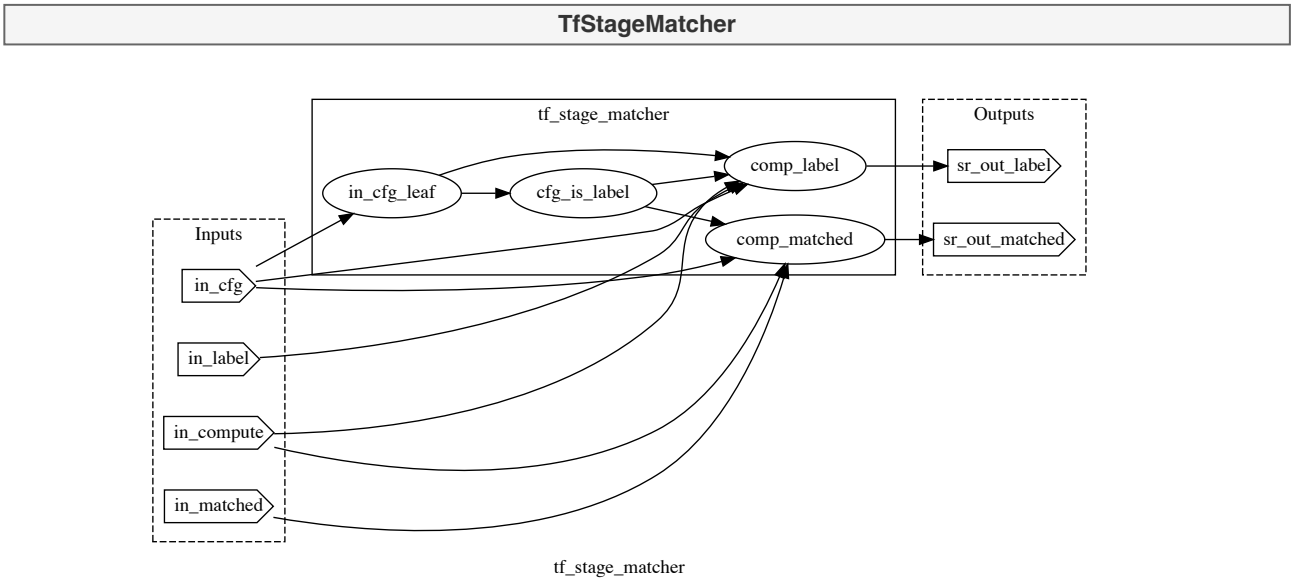


Figure C.5: Synchronization-oriented representation of *TfStageMatcher* module

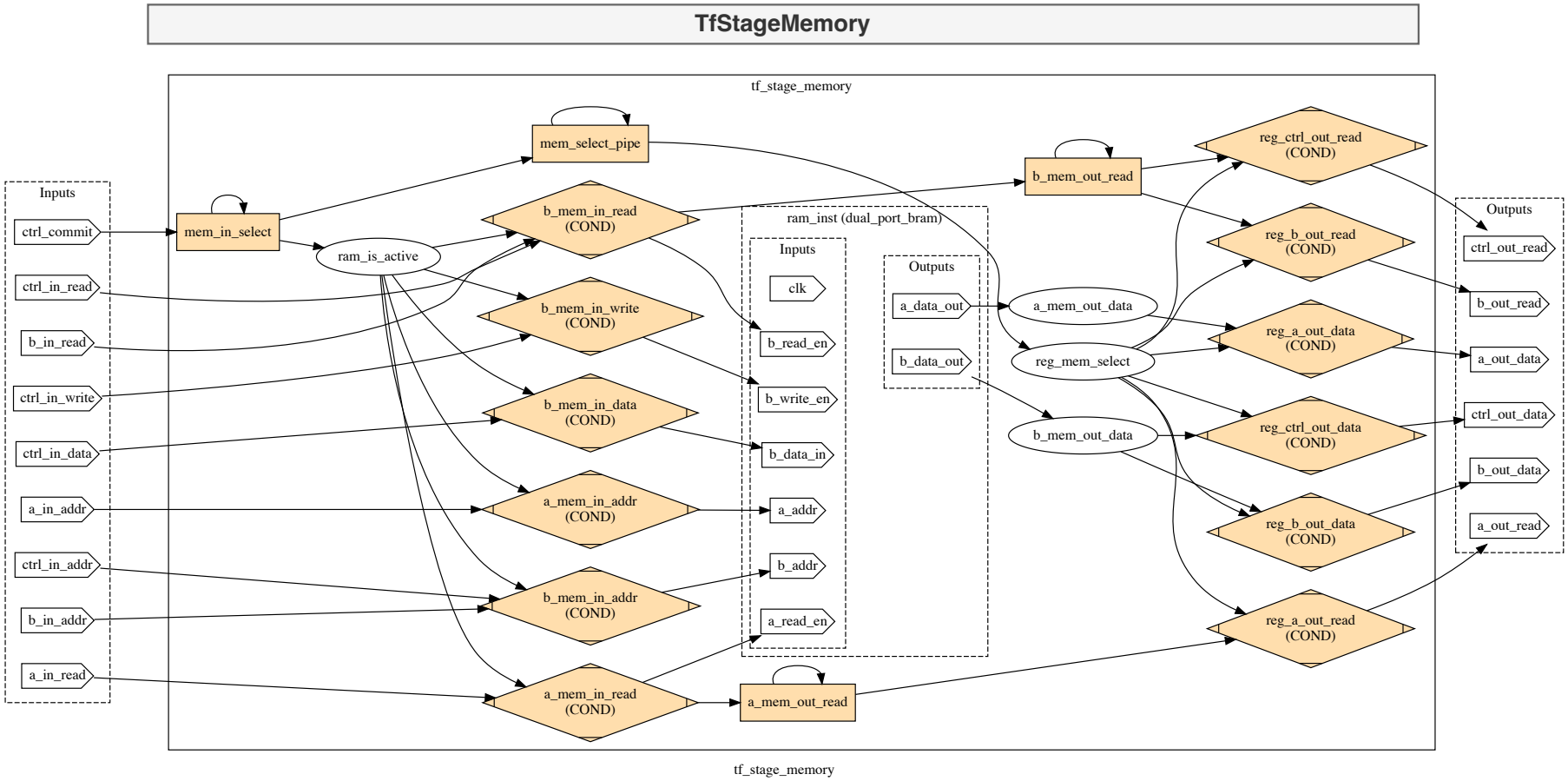


Figure C.6: Synchronization-oriented representation of *TfStageMemory* module

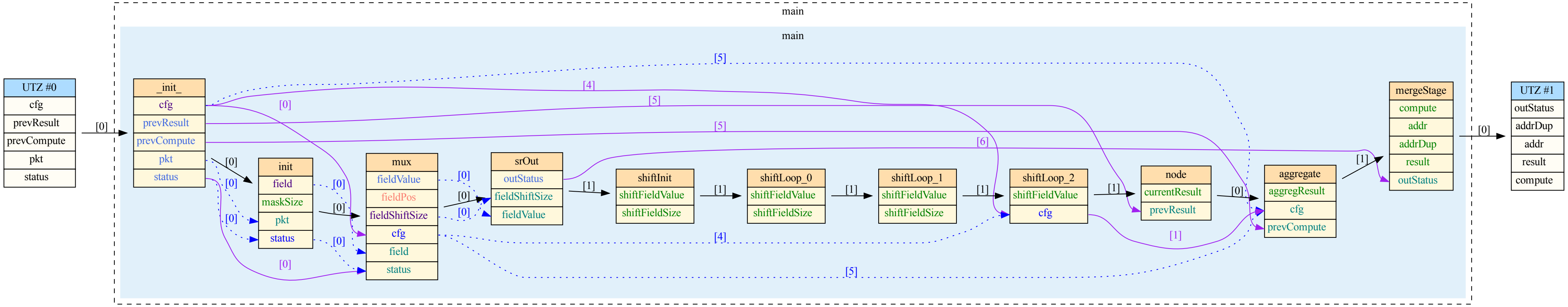


Figure C.7: Framework generated representation of the pipeline-oriented version of *TfStageCompute* module

Appendix D

Chisel Insights

This appendix presents several behaviors of Chisel, from basics to advanced idioms. The first section focuses on the difference between `Vec[Bool]` and `UInt` to represent bit vectors. The second section details two manual iterations over the *sv2chisel*-translated version of the *tree filters* architecture. These two iterations aim at producing a more idiomatic Chisel code, in an attempt to reduce the development flow overhead of the translated version against the original SystemVerilog version.

D.1 UInt vs Vec[Bool] and Flattening

In Chisel, `UInt` is a type only intended for arithmetic and logic operations. It is always processed as a whole by both the Chisel front-end and the FIRRTL compiler. This enables designers to leave the width of a `UInt` signal unspecified, letting the compiler in charge of inferring the required width depending on the surrounding context. However, it is not possible to assign a bit or a range of bits within an `UInt`, therefore to complete such bit-level operations, Chisel requires the usage of a `Vec` type, whose most simple form is a `Vec[Bool]`. A `Bool` is a base type corresponding to a single bit.

From a hardware bit-vector perspective the two following declarations are equivalent as they both manipulate the same number of bits:

```
1 val uintSignal  = Wire(UInt(4.W)) // UInt of 4 bits
2 val vecbSignal  = Wire(Vec(4, Bool())) // Vec of 4 elements of 1 bits
```

In particular, with the appropriate cast, these signals can be connected to one another as follows:

```
1 uintSignal := vecbSignal.asTypeOf(UInt(4.W))
2 vecbSignal := uintSignal.asTypeOf(Vec(4, Bool()))
```

However, from the perspective of Chisel/FIRRTL compilation and emission, these two types are fundamentally different. While the first—`UInt`—is treated as a whole, `Vec[Bool]` is flattened by the FIRRTL compiler as independent `Bool` signals, and emitted as such.

This makes the `UInt` more performant than its `Vec[Bool]` counterpart as demonstrated in Section 7.2.2.1. However, this also prevents some bit-level optimizations that are illustrated through the following code listings.

Beginning with the `UInt`, the following `ExampleUInt` module does two computations. The first, on line 6, computes a *bitwise and (&)* between input signal `in` and the constant value 9.

This value is returned as output `out`, and also used to compute the second output, `condout`, based on its second-lowest significant bit.

```
class ExampleUInt extends Module {
  val in       = IO(Input(UInt(4.W)))
  val out      = IO(Output(UInt(4.W)))
  val condout  = IO(Output(Bool()))

  val tmp = in & "b1001".U
  out := tmp
  condout := Mux(tmp(1), false.B, in(1))
}
```

From this description, Chisel/FIRRTL produces the following Verilog output:

```
module ExampleUInt(
  input        clock,
  input        reset,
  input  [3:0] in,
  output [3:0] out,
  output       condout
);
  wire [3:0] tmp = in & 4'h9;
  assign out = in & 4'h9;
  assign condout = tmp[1] ? 1'h0 : in[1];
endmodule
```

This version is almost a word-for-word translation of the Chisel description, except for the `in & 4'h9` operation, which is intentionally duplicated here for readability.

The following code listing presents the equivalent version based on `Vec[Bool]`. As `Vec` is a generic container type, it does not provide operations involving the interpretation of its components as simple bits, such as the *bitwise and (&)* operator. The description of this basic operation is then more convoluted as it requires to explicitly iterates on the underlying bits of the `Vec`, to literally apply the *bitwise and (&)* operator on each couple of bits from both signals:

```
class ExampleVec extends Module {
  val in       = IO(Input(Vec(4, Bool())))
  val out      = IO(Output(Vec(4, Bool())))
  val condout  = IO(Output(Bool()))

  val tmp = in.zip("b1001".U.asBools).map { case (i,m) => i & m }
  out := tmp
  condout := Mux(tmp(1), false.B, in(1))
}
```

From this `Vec[Bool]` description, Chisel/FIRRTL produces the following Verilog output:

```
module ExampleVec(
  input   clock,
  input   reset,
  input   in_0,
  input   in_1,
  input   in_2,
  input   in_3,
  output  out_0,
  output  out_1,
```

```
10    output  out_2,
11    output  out_3,
12    output  condout
13  );
14    assign out_0 = in_0;
15    assign out_1 = 1'h0;
16    assign out_2 = 1'h0;
17    assign out_3 = in_3;
18    assign condout = in_1;
19  endmodule
```

We observe here a complete simplification of the module, with simple mapping of output bits to input or constant values. This is easily explained by the constant propagation of the zeroes contained in `"b1001".U` constant value. After a *bitwise and* of this value against input signal `in`, the two center bits of the result `tmp` are always equal to 0. This simplification not only applies to the direct outputs `out_1` and `out_2`, but also propagates to the multiplexer `Mux`, thus always selecting the default branch `in(1)`.

With this small example, we highlight the conciseness of `UInt` against `Vec[Bool]` in both the Chisel description and the generated Verilog, with a single signal instead of N in the generated Verilog. While processing `UInt` signals as a whole is more performant, some fine-grained optimization can only be obtained with independent processing of each bit of the signal.

The next section further delves into Chisel, first focusing on structured types and then exploring some idiomatic patterns of the language.

D.2 Antipatterns Translation

D.2.1 Automated Word-for-Word Translation for Correctness

SystemVerilog suffers from a major limitation when it comes to its elaboration capabilities: it is neither able to iterate over structure fields nor to programmatically get or set a field. As a result, to build a somehow generic architecture with SystemVerilog, the implementation of a similar functionality consists in a workaround based on pre-computed bit-level sub-range accesses to the structure as illustrated in the code listing below.

```
1  /******* pre-computed global param (python-based centralized config generation) ********/
2  localparam MAX_FIELD_SIZE   = 64;
3  localparam MAX_STATUS_W = 6;
4  localparam PKT_STATUS_W = 62;
5  localparam FIELDS           = 27;
6
7  localparam integer STATUS_OFFSETS [26:0] = '{61, /* ... */ 16, 12, 6, 0};
8  localparam integer STATUS_SIZES   [26:0] = '{ 1, /* ... */  4,  4, 6, 6};
9
10 typedef struct packed {
11   logic [5:0] src_ip;
12   logic [5:0] dst_ip;
13   logic [3:0] src_port;
14   logic [3:0] dst_port;
15   /* ... */
16   logic [0:0] is_pm;
17 } pkt_status_t;
18
19 /********************************* main verilog file *********************************/
20 // vectorized version of pkt_status_t most notably used for dynamic assignment
21 logic [0:FIELDS-1][MAX_STATUS_W-1:0] in_status_array;
22 // packed structure
```

```
23  logic pkt_status_t  out_status,
24
25  genvar fid;
26  generate
27      for (fid = 0; fid < FIELDS; fid++) begin
28          localparam FIELD_HIGH = PKT_STATUS_W - STATUS_OFFSETS[fid] - 1;
29          localparam FIELD_LOW  = PKT_STATUS_W - STATUS_OFFSETS[fid] - STATUS_SIZES[fid];
30
31          assign out_status[FIELD_HIGH:FIELD_LOW] = in_status_array[fid];
32      end
33  endgenerate
```

Semantic analysis of this pattern to translate it into a more idiomatic Chisel is complex as it would require a generic elaboration mechanism, it is hence not worth the effort. Such pieces of code generally require a larger manual refactoring to benefit from native generation capabilities as detailed later in this section. However, providing a word-for-word translation of this bit-level assignment of a structured type is not that obvious with Chisel. The language indeed has a strong willingness to prevent such direct access to raw bits within structured types because these antipatterns defeat the purpose of functional and object-oriented paradigms. Nonetheless, to provide an equivalent version in Chisel, a workaround based on several implicit conversions is provided as an additional library to be included in the translated result. This piece of code, too large and specific to Chisel/Scala to be reproduced in this manuscript[1], consists of the following main steps:

1. An implicit class provides a sub-range access method on Chisel structured types which returns a custom intermediate class,
2. This custom intermediate class implements two methods: direct assignment and conversion to Chisel type,
3. A final implicit function maps the intermediate class to Chisel type wherever required.

This complex implementation is due to the need to cover both direct usage and assignment use-cases. As further detailed below, this workaround appears to induce a significant overhead which calls for an in-depth manual refactoring.

D.2.2 Manual Idiomatic Translation For Performance

As aforementioned, iteration over structured types is not an obvious task in SystemVerilog, neither is the translation to Chisel of bit-level operations on structured types. Nonetheless, to guarantee a smooth translation process, we provide a rather complex extension of Chisel to allow some bit-level operations on structured types. These operations are based on a hidden complete enumeration of the structured type and several levels of computations to match the requested bit-level access with the available fields. Once replicated through the design, this pattern appears to come with a consequent overhead.

With a manual analysis and appropriate idiomatic fix, based on the original design intent rather than the simple translation workaround, the performance of the resulting Chisel is substantially improved. Table D.1 showcases the positive impact of removing this translation hack in favor of a more idiomatic manipulation of structured types. In particular, it improves Chisel elaboration, dividing by two both the size of generated FIRRTL and the duration of this step. While focused on enhancing the overall user-experience, this iteration also slightly benefits the simulation execution speed.

[1] Full source code available at https://github.com/ovh/sv2chisel/blob/master/helpers/src/main/scala/bundleconvert.scala

		Optimized Inference	Optimized Bundle Iterations	Diff	%
Generated Files (MB)	*Chisel*				
	top.fir	80	26	-54	-68%
	top.v	6.5	6.4	-0.1	-2%
Duration (mm:ss)	Scala compilation	00:08	00:06	≃	≃
	Chisel elaboration	00:12	00:06	-00:06	-50%
	FIRRTL compilation	00:50	00:32	-00:18	-36%
	Verilation	00:54	00:45	-00:09	-16%
	C++ compilation	01:08	01:03	-00:05	-7%
	TOTAL	03:12	02:32	-00:40	-20%
	Simulation rate (trace)	610ns/s	652ns/s	+42ns/s	+7%

Table D.1: Positive impact of antipattern removal on *Tree filters* architecture

D.2.3 Experimenting With Optimized Patterns

As a final Chisel-oriented idiomatic improvement, this iteration focuses on the high replication level of the *tree filters* hierarchy.

A standard Chisel hierarchy generates a FIRRTL module for each instance of a Chisel module, regardless of the parameterization of these instances. In particular, even if several instances share the exact same parameter sets, each instance will be independently elaborated and results in module duplication in the resulting FIRRTL. To reduce this code duplication in the generated Verilog, the FIRRTL compiler is then able to merge such duplicated instances within the same Verilog module, instantiated several times through the design. This approach is counterproductive, with wasted resources for duplicated elaboration, and additional resources needed to merge the duplicated instances.

		Optimized Bundle Iterations	Optimized Instantiation	Diff	%
Generated Files (MB)	*Chisel*				
	top.fir	26	17	-9	-34%
	top.v	6.4	6.4	=	=
Duration (mm:ss)	Scala compilation	00:06	00:06	≃	≃
	Chisel elaboration	00:06	00:07	+00:01	≃
	FIRRTL compilation	00:32	00:30	-00:02	≃
	Verilation	00:45	00:45	=	
	C++ compilation	01:03	01:03	=	
	TOTAL	02:32	02:31	-00:01	≃
	Simulation rate (trace)	652ns/s	656ns/s	+4ns/s	≃

Table D.2: Impact of Definition/Instance pattern on *Tree filters* architecture

To get rid of this suboptimal process, a dedicated API has recently been introduced in Chisel. It consists of several annotations and two keywords, `Definition` and `Instance`, used in place of the usual instantiation pattern. For duplicated modules, a single `Definition` of a module is created. It elaborates the module only once and returns a reference to be

instantiated with `Instance` keyword in several places of the design.

While this pattern yet remains very manual and verbose, it might get implemented as default behavior at some point in the Chisel stack.

With many replicated modules sharing identical parameterization through *tree filters* hierarchy, we expected a noticeable improvement on the generation stack. However, as illustrated in Table D.2, implementation of this pattern only substantially impacts the size of the generated FIRRTL. We incidentally notice a slight reduction of FIRRTL compilation time, which comes at the cost of a roughly equivalent increase of Chisel elaboration time. This latest paradoxical observation, is probably explained by the quality of JVM implementation, which is able to cache the result of previous computations, such as previous module elaboration in the current case.

Publications

Journal

D&T2021 IEEE Design & Test, 2021 [BHM+21]
Towards Agile Hardware Designs with Chisel: a Network Use-case

International Conference

RSP'2020 *Virtual event* [BHM+20]
(System)Verilog to Chisel Translation for Faster Hardware Design

Bibliography

[AAB+16] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016. (Cited on pages 18, 19, 39, and 45.)

[AB09] Michael Attig and Gordon Brebner. High-level programming of the fpga on netfpga. In *Proc. NetFPGA Developers' Workshop*, 2009. (Cited on page 26.)

[AB12] Jean-Philippe Aumasson and Daniel J Bernstein. Siphash: a fast short-input prf. In *International Conference on Cryptology in India*, pages 489–508. Springer, 2012. (Cited on page 50.)

[AB18] Mustafa Abbas and Vaughn Betz. Latency insensitive design styles for fpgas. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 360–3607. IEEE, 2018. (Cited on pages 27 and 70.)

[ABG+20] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, et al. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 2020. (Cited on pages 39 and 91.)

[ACM09] ACM. Barbara Liskov, A.M. Turing award laureates. Online, https://amturing.acm.org/award_winners/liskov_1108679.cfm, 2009. (Cited on page 22.)

[agi11] Manifesto for agile software development. Online, http://agilemanifesto.org/, 2011. (Cited on page 6.)

[APC15] Krste Asanovic, David A Patterson, and Christopher Celio. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical report, University of California at Berkeley Berkeley United States, 2015. (Cited on page 45.)

[BBF+00] Gérard Berry, Amar Bouali, Xavier Fornari, Emmanuel Ledinot, Eric Nassor, and Robert De Simone. Esterel: A formal method applied to avionic software development. *Science of Computer Programming*, 36(1):5–25, 2000. (Cited on page 24.)

[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. In *ACM SIGPLAN Notices*, volume 34, pages 174–184. ACM, 1998. (Cited on page 35.)

[BDG+14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014. (Cited on pages 17, 26, and 29.)

[Beh17] Raul Behl. SimpleCPU. Retrieved July 2020, 2017. (Cited on page 99.)

[BFP07] Nicola Bombieri, Franco Fummi, and Graziano Pravadelli. Incremental abv for functional validation of tl-to-rtl design refinement. In *2007 Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6. IEEE, 2007. (Cited on page 30.)

[BGG+22] Clément Boin, Xavier Guillaume, Gilles Grimaud, Tristan Groléat, and Michaël Hauspie. One year of ddos attacks against a cloud provider: an overview. In *4th International Conference on Advances in Computer Technology, Information Science and Communications*, 2022. (Cited on page 2.)

[BH98] Peter Bellows and Brad Hutchings. Jhdl-an hdl for reconfigurable systems. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 175–184. IEEE, 1998. (Cited on page 35.)

[BHM+20] Jean Bruant, Pierre-Henri Horrein, Olivier Muller, Tristan Groleat, and Frédéric Pétrot. (system) verilog to chisel translation for faster hardware design. In *2020 International Workshop on Rapid System Prototyping (RSP)*, pages 1–7. IEEE, 2020. (Cited on page 161.)

[BHM+21] Jean Bruant, Pierre-Henri Horrein, Olivier Muller, Tristan Groleat, and Frédéric Pétrot. Towards agile hardware designs with chisel: a network use-case. *IEEE Design & Test*, 2021. (Cited on pages 65 and 161.)

[BKK+10] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. Clash: Structural descriptions of synchronous hardware using haskell. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 714–721. IEEE, 2010. (Cited on pages 18, 35, and 36.)

[BM+20] Doreen Bogdan-Martin et al. Measuring digital development facts and figures 2020. Technical report, International Telecommunication Union, 2020. (Cited on pages 7 and 11.)

[BMR16] Alban Bourge, Olivier Muller, and Frédéric Rousseau. Generating efficient context-switch capable circuits through autonomous design flow. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(1):9, 2016. (Cited on page 17.)

[Bou13] Sébastien Bourdeauducq. Migen: A python toolbox for building complex digital hardware. Online, https://m-labs.hk/migen/manual/introduction.html, 2013. (Cited on pages 35, 36, 37, 39, and 91.)

[BPCC20] Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. The essence of bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 243–257, 2020. (Cited on page 39.)

[BRS13] David F Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013. (Cited on page 16.)

[BVR+12] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. CHISEL: Constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221. IEEE, 2012. (Cited on pages 3, 35, 36, 37, 39, 41, and 133.)

[CAR14] Ivano Cerrato, Mauro Annarumma, and Fulvio Risso. Supporting fine-grained network functions through intel dpdk. In *2014 Third European Workshop on Software Defined Networks*, pages 1–6. IEEE, 2014. (Cited on page 13.)

[CAZ+14] Jakub Czyz, Mark Allman, Jing Zhang, Scott Iekel-Johnson, Eric Osterweil, and Michael Bailey. Measuring ipv6 adoption. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 87–98, 2014. (Cited on page 11.)

[CB04] Jakob Carlstrom and Thomas Bodén. Synchronous dataflow architecture for network processors. *IEEE Micro*, 24(5):10–18, 2004. (Cited on page 30.)

[CCB+08] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. Gaut: A high-level synthesis tool for dsp applications. In *High-Level Synthesis*, pages 147–169. Springer, 2008. (Cited on page 26.)

[CDPA+18] Danilo Cerović, Valentin Del Piccolo, Ahmed Amamou, Kamel Haddadou, and Guy Pujolle. Fast packet processing: A survey. *IEEE Communications Surveys & Tutorials*, 20(4):3645–3676, 2018. (Cited on page 13.)

[CL+17] Valter Costa, Bertrand Lefort, et al. Inspector, a zero code ide for control systems user interface development. In *Proc. 16th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'17)*, pages 861–865, 2017. (Cited on page 24.)

[CM05] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005. (Cited on page 54.)

[CMSV01] Luca P Carloni, Kenneth L McMillan, and Alberto L Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 20(9):1059–1076, 2001. (Cited on page 29.)

[Cot65] Leonard W Cotten. Circuit implementation of high-speed pipeline systems. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 489–504, 1965. (Cited on page 27.)

[CS12] Doris Chen and Deshanand Singh. Using opencl to evaluate the efficiency of cpus, gpus and fpgas for information filtering. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 5–12. IEEE, 2012. (Cited on page 13.)

[CSBH21] Michael Christensen, Timothy Sherwood, Jonathan Balkind, and Ben Hardekopf. Wire sorts: a language abstraction for safe hardware composition. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 175–189, 2021. (Cited on page 31.)

[CTD+17] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. A pythonic approach for rapid hardware prototyping and instrumentation. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–7. IEEE, 2017. (Cited on pages 35 and 36.)

[CTL17] Henry Cook, Wesley Terpstra, and Yunsup Lee. Diplomatic design patterns: A tilelink case study. In *1st Workshop on Computer Architecture Research with RISC-V*, 2017. (Cited on pages 18, 39, and 68.)

[CVS+17] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, et al. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):24, 2017. (Cited on page 39.)

[Dan22] Shibo Dang. Chisel-based implementation of high-performance pipelined priority queue generator. In *2022 14th International Conference on Computer Research and Development (ICCRD)*, pages 317–322. IEEE, 2022. (Cited on page 66.)

[DC21] Justin Deters and Ron Cytron. Performance counter design variation in rocket chip via feature-oriented programming. In *Proceedings of Fifth Workshop on Computer Architecture Research with RISC-V (CARRV 2021)*, 2021. (Cited on page 32.)

[DCF+14] Rolf Drechsler, Christophe Chevallaz, Franco Fummi, Alan J Hu, Ronny Morad, Frank Schirrmeister, and Alex Goryachev. Panel: Future soc verification methodology: Uvm evolution or revolution? In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–5. IEEE, 2014. (Cited on page 44.)

[Dec15] Jan Decaluwe. MyHDL website. http://www.myhdl.org, 2015. (Cited on pages 35, 36, and 37.)

[DFH+20] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 408–422, 2020. (Cited on page 29.)

[DGX+17] Shuwen Deng, Doguhan Gümüsoglu, Wenjie Xiong, Y Serhan Gener, Onur Demir, and Jakub Szefer. Secchisel: Language and tool for practical and scalable security verification of security-aware hardware architectures. *IACR Cryptology ePrint Archive*, 2017:193, 2017. (Cited on page 68.)

[DGX+19] Shuwen Deng, Doğuhan Gümüşoğlu, Wenjie Xiong, Sercan Sari, Y Serhan Gener, Corine Lu, Onur Demir, and Jakub Szefer. Secchisel framework for security verification of secure processor architectures. In *Proceedings of the*

8th International Workshop on Hardware and Architectural Support for Security and Privacy, page 7. ACM, 2019. (Cited on page 68.)

[DKS+12] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. Robotml, a domain-specific language to design, simulate and deploy robotic applications. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 149–160. Springer, 2012. (Cited on page 23.)

[DLL+21] Yue Dai, Greg LaCaille, Harrison Liew, James Dunn, and Borivoje Nikolić. A scalable massive mimo uplink baseband processing generator. In *ICC 2021-IEEE International Conference on Communications*, pages 1–6. IEEE, 2021. (Cited on page 65.)

[dMHV19] Jan de Muijnck-Hughes and Wim Vanderbauwhede. A typing discipline for hardware interfaces. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019. (Cited on page 31.)

[DPM21] Vukan D Damnjanović, Marija L Petrović, and Vladimir M Milovanović. A parameterizable chisel generator of numerically controlled oscillators for direct digital synthesis. In *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 141–144. IEEE, 2021. (Cited on page 65.)

[DSFE17] Keerthikumara Devarajegowda, Johannes Schreiner, Rainer Findenig, and Wolfgang Ecker. Python based framework for hdsls with an underlying formal semantics. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 1019–1025. IEEE Press, 2017. (Cited on page 35.)

[DTCC+18] Lorenzo Di Tucci, Davide Conficconi, Alessandro Comodi, Steven Hofmeyr, David Donofrio, and Marco D Santambrogio. A parallel, energy efficient hardware architecture for the meraligner on fpga using chisel hcl. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 214–217. IEEE, 2018. (Cited on page 45.)

[DTH18] Ross Daly, Lenny Truong, and Pat Hanrahan. Invoking and linking generators from multiple hardware languages using coreir. In *Proceedings of the first Workshop on Open-Source EDA Technology*, 2018. (Cited on page 39.)

[Eri17] Jack Erickson. *HDL CoderTM Evaluation Reference Guide*. The MathWorks, r2017b edition, 2017. (Cited on page 26.)

[ES16] Wolfgang Ecker and Johannes Schreiner. Introducing model-of-things (mot) and model-of-design (mod) for simpler and more efficient hardware generators. In *Very Large Scale Integration (VLSI-SoC), 2016 IFIP/IEEE International Conference on*, pages 1–6. IEEE, 2016. (Cited on page 35.)

[Fer22] Bruno Ferres. *Leveraging Hardware Construction Languages for Flexible Design Space Exploration on FPGA*. PhD thesis, Université Grenoble Alpes, 2022. (Cited on page 136.)

[FFDMS16] Farzad Fatollahi-Fard, David Donofrio, George Michelogiannakis, and John Shalf. Opensoc fabric: On-chip network generator. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–203. IEEE, 2016. (Cited on page 45.)

[FMR20] Bruno Ferres, Olivier Muller, and Frédéric Rousseau. Chisel usecase: Designing general matrix multiply for fpga. In *International Symposium on Applied Reconfigurable Computing*, pages 61–72. Springer, 2020. (Cited on page 65.)

[FMR21] Bruno Ferres, Olivier Muller, and Frédéric Rousseau. Integrating quick resource estimators in hardware construction framework for design space exploration. In *2021 International Workshop on Rapid System Prototyping (RSP)*, pages 1–7. IEEE, 2021. (Cited on pages 38 and 136.)

[FO00] Peter Frey and Donald O'Riordan. Verilog-ams: Mixed-signal simulation and cross domain connect modules. In *Proceedings 2000 IEEE/ACM International Workshop on Behavioral Modeling and Simulation*, pages 103–108. IEEE, 2000. (Cited on page 44.)

[G+14] David J. Greaves et al. Toy bluespec compiler. https://www.cl.cam.ac.uk/~djg11/wwwhpr/toy-bluespec-compiler.html, 2014. (Cited on page 34.)

[GBK+09] Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing kansas lava. In *International Symposium on Implementation and Application of Functional Languages*, pages 18–35. Springer, 2009. (Cited on page 35.)

[GCW+21] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. Autobridge: Coupling coarse-grained floorplanning and pipelining for high-frequency hls design on multi-die fpgas. In ACM, editor, *2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '21), February 28–March 2, 2021, Virtual Event, USA.*, 2021. (Cited on page 29.)

[GEHM+22] Xinfei Guo, Mohamed El-Hadedy, Sergiu Mosanu, Xiangdong Wei, Kevin Skadron, and Mircea R Stan. Agile-aes: Implementation of configurable aes primitive with agile design approach. *Integration*, 85:87–96, 2022. (Cited on pages 65 and 66.)

[GL98] Stephan W Gehring and Stefan H-M Ludwig. Fast integrated tools for circuit design with fpgas. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 133–139. ACM, 1998. (Cited on page 38.)

[GLN+14] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J Brown, Arvind K Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8. IEEE, 2014. (Cited on page 26.)

[GM01] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2001. (Cited on page 111.)

[GOCBK10] Marc Galceran-Oms, Jordi Cortadella, Dmitry Bufistov, and Mike Kishinevsky. Automatic microarchitectural pipelining. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 961–964. IEEE, 2010. (Cited on page 29.)

[GRDT+16] Giulia Guidi, Enrico Reggiani, Lorenzo Di Tucci, Gianluca Durelli, Michaela Blott, and Marco D Santambrogio. On how to improve fpga-based systems design productivity via sdaccel. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 247–252. IEEE, 2016. (Cited on pages 17 and 26.)

[Gre15] David J Greaves. Layering rtl, safl, handel-c and bluespec constructs on chisel hcl. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*, pages 108–117. IEEE, 2015. (Cited on pages 39 and 68.)

[Gre19] David J Greaves. Further sub-cycle and multi-cycle schedulling support for bluespec verilog. In *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design*, page 2. ACM, 2019. (Cited on page 34.)

[GTA06] Michael I Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGPLAN Notices*, 41(11):151–162, 2006. (Cited on page 27.)

[GTSB08] Sébastien Gérard, François Terrier, Bran Selic, and Pierre Boulet. Marte, the uml standard extension for real-time and embedded systems. Technical report, CEA LIST, Laboratory of Model Driven Engineering for Embedded Systems (LISE), 2008. (Cited on page 26.)

[Har09] Ed Harcourt. Policies of system level pipeline modeling. *Electronic Notes in Theoretical Computer Science*, 238(2):13–23, 2009. (Cited on page 28.)

[HF72] Thomas G Hallin and Michael J Flynn. Pipelining of arithmetic functions. *IEEE Transactions on Computers*, 100(8):880–886, 1972. (Cited on page 27.)

[HG21] Pierre-henri Horrein and Tristan Groleat. Method and system for classifying data packet fields on FPGA, 2021. US Patent App. 17/105,996. (Cited on page 112.)

[HL95] Walter L. Hürsch and Cristina Videira Lopes. Separation of concerns. Technical report, College of Computer Science, Northeastern University, 1995. (Cited on page 31.)

[HMLT03] Per Haglund, Oskar Mencer, Wayne Luk, and Benjamin Tai. Hardware design with a scripting language. In *International Conference on Field Programmable Logic and Applications*, pages 1040–1043. Springer, 2003. (Cited on page 34.)

[HMN01] Yoav Hollander, Matthew Morley, and Amos Noy. The e language: A fresh separation of concerns. In *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 38*, pages 41–50. IEEE, 2001. (Cited on page 32.)

[Hoo17] Steven F Hoover. Timing-abstract circuit design in transaction-level verilog. In *Computer Design (ICCD), 2017 IEEE International Conference on*, pages 525–532. IEEE, 2017. (Cited on pages 28, 31, and 34.)

[HP14] Ed Harcourt and James Perconti. A systemc library for specifying pipeline abstractions. *Microprocessors and Microsystems*, 38(1):76–81, 2014. (Cited on page 28.)

[I+18] IEEE 802.3 Working Group et al. IEEE std 802.3-2018 standard for ethernet. *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, pages 118,161, 2018. (Cited on page 10.)

[IBA+19] Adam Izraelevitz, Jonathan Bachrach, Krste Asanović, Simon Schleicher, and Jonathan Ragan-Kelley. *Unlocking Design Reuse with Hardware Compiler Frameworks*. phdthesis, University of California at Berkeley, 2019. (Cited on page 32.)

[IdD17] Matei Iştoan and Florent de Dinechin. Automating the pipeline of arithmetic datapaths. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 704–709. IEEE, 2017. (Cited on page 29.)

[IKL+17] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 209–216. IEEE Press, 2017. (Cited on pages 39, 40, 41, and 68.)

[Inc20] BlueSpec Inc. Github page of bluespec compiler. Online, https://github.com/B-Lang-org/bsc, 2020. (Cited on page 34.)

[IT21] J. Iyengar and M. Thomson. QUIC: A UDP-based multiplexed and secure transport. RFC 9000, RFC Editor, May 2021. (Cited on page 11.)

[JB99] James Jennings and Eric Beuscher. Verischemelog: Verilog embedded in scheme. In *ACM SIGPLAN Notices*, volume 35, pages 123–134. ACM, 1999. (Cited on page 34.)

[JGI18] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 127–136. ACM, 2018. (Cited on page 29.)

[JIB18] Shunning Jiang, Berkin Ilbeyi, and Christopher Batten. Mamba: closing the performance gap in productive hardware development frameworks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018. (Cited on page 35.)

[JPOB20] Shunning Jiang, Peitian Pan, Yanghui Ou, and Christopher Batten. Pymtl3: A python framework for open-source hardware modeling, generation, simulation, and verification. *IEEE Micro*, 2020. (Cited on pages 35 and 38.)

[JS15] Keerthan Jaic and Melissa C Smith. Enhancing hardware design flows with myhdl. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 28–31. ACM, 2015. (Cited on pages 35, 36, and 37.)

[JSG+21] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. Buffer placement and sizing for high-performance dataflow circuits. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)*, 15(1), nov 2021. (Cited on page 29.)

[JYP+17] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017. (Cited on page 40.)

[JYPP18] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. Motivation for and evaluation of the first tensor processing unit. *ieee Micro*, 38(3):10–19, 2018. (Cited on page 40.)

[KBBLL19] Florent Kermarrec, Sébastien Bourdeauducq, Hannah Badier, and Jean-Christophe Le Lann. Litex: an open-source soc builder and library based on migen python dsl. In *Proceedings of Workshop on Open Source Design Automation (OSDA 2019), colocated with DATE 2019 Design Automation and Test in Europe*, 2019. (Cited on pages 39 and 91.)

[KC11] Alexander S Kamkin and Mikhail M Chupilko. Survey of modern technologies of simulation-based verification of hardware. *Programming and Computer Software*, 37(3):147–152, 2011. (Cited on page 44.)

[Kem20] Julian Kemmerer. PipelineC. https://github.com/JulianKemmerer/PipelineC, 2020. (Cited on page 29.)

[KH06] David Kearney and John Hopf. Hardware join java: a unified hardware/software language for dynamic partial runtime reconfigurable computing applications. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 277–280. IEEE, 2006. (Cited on page 35.)

[KH21] Matti Käyrä and Timo D Hämäläinen. A survey on system-on-a-chip design using chisel hw construction language. In *IECON 2021–47th Annual Conference of the IEEE Industrial Electronics Society*, pages 1–6. IEEE, 2021. (Cited on page 66.)

[Kin19] Olof Kindgren. A scalable approach to ip management with fusesoc. In *Proceedings of Workshop on Open Source Design Automation (OSDA 2019) colocated with DATE 2019 Design Automation and Test in Europe*, 2019. (Cited on page 91.)

[KJA+21] Muhammad Hadir Khan, Aireen Amir Jalal, Sajjad Ahmed, Ali Ahmed Ansari, and Syed Roomi Naqvi. Ibtida: Fully open-source asic implementation of chisel-generated system on a chip. 2021. (Cited on page 65.)

[KK17] Hyoukjun Kwon and Tushar Krishna. Opensmart: Single-cycle multi-hop noc generator in bsv and chisel. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*, pages 195–204. IEEE, 2017. (Cited on page 45.)

[KKCGO08] Timothy Kam, Michael Kishinevsky, Jordi Cortadella, and Marc Galceran-Oms. Correct-by-construction microarchitectural pipelining. In *2008*

IEEE/ACM International Conference on Computer-Aided Design, pages 434–441. IEEE, 2008. (Cited on page 29.)

[KLK+21] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. A hardware accelerator for protocol buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 462–478, 2021. (Cited on page 65.)

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997. (Cited on page 32.)

[KMF01] Emre Kiciman, Laurence Melloul, and Armando Fox. Position summary: towards zero-code service composition. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, page 172. IEEE, 2001. (Cited on page 24.)

[KMK+18] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 29–42. IEEE Press, 2018. (Cited on page 68.)

[KP01] Daniel Kroening and Wolfgang J Paul. Automated pipeline design. In *Proceedings of the 38th annual Design Automation Conference*, pages 810–815, 2001. (Cited on page 29.)

[KST13] Md Kamruzzaman, Steven Swanson, and Dean M Tullsen. Load-balanced pipeline parallelism. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013. (Cited on page 27.)

[KVDWDK+08] Wido Kruijtzer, Pieter Van Der Wolf, Erwin De Kock, Jan Stuyt, Wolfgang Ecker, Albrecht Mayer, Serge Hustin, Christophe Amerijckx, Serge De Paoli, and Emmanuel Vaumorin. Industrial ip integration flows based on ip-xact standards. In *2008 Design, Automation and Test in Europe*, pages 32–37. IEEE, 2008. (Cited on page 26.)

[KZBA19] Donggyu Kim, Jerry Zhao, Jonathan Bachrach, and Krste Asanović. Simmani: Runtime power modeling for arbitrary rtl with automatic signal selection. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1050–1062. ACM, 2019. (Cited on page 68.)

[L+21] Christopher Lattner et al. CIRCT / Circuit IR Compilers and Tools. online, 2021. (Cited on pages 27 and 39.)

[LAB+21] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021. (Cited on page 27.)

[LDC12] Maysam Lavasani, Larry Dennison, and Derek Chiou. Compiling high throughput network processors. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 87–96. ACM, 2012. (Cited on page 26.)

[LG+86] Barbara Liskov, John Guttag, et al. *Abstraction and specification in program development*, volume 180. MIT press Cambridge, 1986. (Cited on page 22.)

[LKK+18] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: coverage-directed fuzz testing of rtl on fpgas. In *Proceedings of the International Conference on Computer-Aided Design*, page 28. ACM, 2018. (Cited on page 68.)

[LKM16] Evangelos Logaras, Evangelos Koutsouradis, and Elias S Manolakos. Python facilitates the rapid prototyping and hw/sw verification of processor centric socs for fpgas. In *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on*, pages 1214–1217. IEEE, 2016. (Cited on pages 35 and 37.)

[LL95] Yanbing Li and Miriam Leeser. Hml: an innovative hardware description language and its translation to vhdl. In *Design Automation Conference, 1995. Proceedings of the ASP-DAC'95/CHDL'95/VLSI'95., IFIP International Conference on Hardware Description Languages. IFIP International Conference on Very Large Scal*, pages 691–696. IEEE, 1995. (Cited on page 35.)

[LLS+15] I-Ting Angelina Lee, Charles E Leiserson, Tao B Schardl, Zhunping Zhang, and Jim Sukha. On-the-fly pipeline parallelism. *ACM Transactions on Parallel Computing (TOPC)*, 2(3):1–42, 2015. (Cited on page 27.)

[LLX+14] Yao Li, Antonio Roldao Lopes, Zhouyun Xu, Zhengwei Qi, and Haibing Guan. Scalahdl: Express and test hardware designs in a scala dsl. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pages 521–524. IEEE, 2014. (Cited on pages 35 and 36.)

[LLX+17] Yanqiang Liu, Yao Li, Weilun Xiong, Meng Lai, Cheng Chen, Zhengwei Qi, and Haibing Guan. Scala based fpga design flow. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 286–286. ACM, 2017. (Cited on page 35.)

[LM87a] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. (Cited on page 30.)

[LM87b] Edward Ashford Lee and David G Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers*, 100(1):24–35, 1987. (Cited on page 30.)

[LM10] Evangelos Logaras and Elias S Manolakos. Syspy: using python for processor-centric soc design. In *Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on*, pages 762–765. IEEE, 2010. (Cited on page 35.)

[LP95] Edward A Lee and Thomas M Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995. (Cited on page 30.)

[LRW+17] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 183–196, 2017. (Cited on page 11.)

[LS91] Charles E Leiserson and James B Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1-6):5–35, 1991. (Cited on page 71.)

[LSW+21] Chenhao Liu, Zhiyuan Shao, Zeke Wang, Kexin Li, Minkang Wu, Jiajie Chen, Xiaofei Liao, and Hai Jin. Scalabfs: A scalable bfs accelerator on hbm-enhanced fpgas. *arXiv preprint arXiv:2105.11754*, 2021. (Cited on page 66.)

[LTL+16] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 1–14. ACM, 2016. (Cited on page 26.)

[LTN+18] Derek Lockhart, Stephen Twigg, Ravi Narayanaswami, Jeremy Coriell, Uday Dasari, Richard Ho, Doug Hogberg, George Huang, Anand Kane, Chintan Kaur, Tao Liu, Adriana Maggiore, Kevin Townsend, and Emre Tuncer. Experiences building edge tpu with chisel. 1st Chisel Community Conference, CCC2018, 2018. (Cited on page 40.)

[LWC+16] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Stevo Bailey, Milovan Blagojevic, et al. An agile approach to building risc-v microprocessors. *IEEE Micro*, 36(2):8–20, 2016. (Cited on pages 6, 40, and 45.)

[LZ74] Barbara Liskov and Stephen Zilles. Programming with abstract data types. *ACM Sigplan Notices*, 9(4):50–59, 1974. (Cited on page 57.)

[LZB14] Derek Lockhart, Gary Zibrat, and Christopher Batten. Pymtl: A unified framework for vertically integrated computer architecture research. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 280–292. IEEE Computer Society, 2014. (Cited on pages 35 and 38.)

[Mas07] Ali Mashtizadeh. Phdl: A python hardware design framework, 2007. (Cited on pages 35 and 37.)

[Max11] Maxeler. Maxcompiler white paper, 2011. (Cited on pages 18 and 35.)

[MB21] Kingshuk Majumder and Uday Bondhugula. Hir: An mlir-based intermediate representation for hardware accelerator description. *arXiv preprint arXiv:2103.00194*, 2021. (Cited on page 44.)

[MBM+20] Kotaro Matsuoka, Ryotaro Banno, Naoki Matsumoto, Takashi Sato, and Song Bian. Virtual secure platform: A five-stage pipeline processor over tfhe. *arXiv preprint arXiv:2010.09410*, 2020. (Cited on page 66.)

[MFL+21] Mehdi Moghaddamfar, Christian Färber, Wolfgang Lehner, Norman May, and Akash Kumar. Resource-efficient database query processing on fpgas. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*, pages 1–8, 2021. (Cited on page 65.)

[MGDG+20] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G Cota, Michele Petracca, Christian Pilato, and Luca P Carloni. Agile soc development with open esp. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020. (Cited on page 91.)

[MMGC20] Paolo Mantovani, Robert Margelli, Davide Giri, and Luca P Carloni. Hl5: A 32-bit risc-v processor designed with high-level synthesis. In *2020 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8. IEEE, 2020. (Cited on page 17.)

[MML13] Fabian May and Friedrich Mayer-Lindenberg. Modhdl: A modular and expandable language for developing synchronous hardware. In *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*, pages 1–6. IEEE, 2013. (Cited on pages 26 and 35.)

[MP19] VM Milovanović and ML Petrović. A highly parametrizable chisel hcl generator of single-path delay feedback fft processors. In *2019 IEEE 31st International Conference on Microelectronics (MIEL)*, pages 247–250. IEEE, 2019. (Cited on page 45.)

[MR01] Maria-Cristina V Marinescu and Martin Rinard. High-level automatic pipelining for sequential circuits. In *Proceedings of the 14th international symposium on Systems synthesis*, pages 215–220, 2001. (Cited on page 29.)

[MSB+06] David Moore, Colleen Shannon, Douglas J Brown, Geoffrey M Voelker, and Stefan Savage. Inferring internet denial-of-service activity. *ACM Transactions on Computer Systems (TOCS)*, 24(2):115–139, 2006. (Cited on page 2.)

[MXSJ17] Tasnuva Mahjabin, Yang Xiao, Guang Sun, and Wangdong Jiang. A survey of distributed denial-of-service attack, prevention, and mitigation techniques. *International Journal of Distributed Sensor Networks*, 13(12):1550147717741463, 2017. (Cited on page 2.)

[Nay18] Matthew Naylor. Blarney github page. https://github.com/blarney-lang/blarney, 2018. (Cited on page 35.)

[Net20] Netscout. Worldwide infrastructure security report. Web, 2020. (Cited on page 2.)

[Nik04] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70. IEEE, 2004. (Cited on pages 31, 34, and 39.)

[Nik08] Rishiyur S Nikhil. Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions. In *High-Level Synthesis*, pages 129–146. Springer, 2008. (Cited on pages 31, 34, and 39.)

[NS96] Wolfgang Nebel and Guido Schumacher. Object-oriented hardware modelling-where to apply and what are the objects?'. In *Proceedings EURO-DAC'96. European Design Automation Conference with EURO-VHDL'96 and Exhibition*, pages 428–433. IEEE, 1996. (Cited on page 38.)

[NSP+16] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016. (Cited on pages 16, 17, and 26.)

[NSS+16] Eriko Nurvitadhi, Jaewoong Sim, David Sheffield, Asit Mishra, Srivatsan Krishnan, and Debbie Marr. Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2016. (Cited on page 13.)

[Pan01] Preeti Ranjan Panda. Systemc-a modeling platform supporting multiple design abstractions. In *International Symposium on System Synthesis (IEEE Cat. No. 01EX526)*, pages 75–80. IEEE, 2001. (Cited on page 31.)

[Pap16] Charles Papon. Spinalhdl. Online, https://spinalhdl.github.io/SpinalDoc/, 2016. (Cited on pages 35 and 38.)

[PBVS+20] Johan Peltenburg, Matthijs Brobbel, Jeroen Van Straten, Zaid Al-Ars, and Peter Hofstee. Tydi: an open specification for complex data structures over hardware streams. *IEEE Micro*, 2020. (Cited on page 31.)

[PE17] Oron Port and Yoav Etsion. Dfiant: A dataflow hardware description language. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–4. IEEE, 2017. (Cited on pages 29 and 35.)

[PF11] Terence Parr and Kathleen Fisher. LL(*) the foundation of the ANTLR parser generator. *ACM Sigplan Notices*, 46(6):425–436, 2011. (Cited on page 93.)

[PFSS15] João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. Pi-ware: Hardware description and verification in agda. In *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:27. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015. (Cited on page 35.)

[PK16] Panagiotis Patros and Kenneth B. Kent. Automatic detection and elision of reset sub-circuits. In *Proc. of the 27th International Symposium on Rapid System Prototyping*, pages 26–32, 2016. (Cited on page 95.)

[PKB+16] Raghu Prabhakar, David Koeplinger, Kevin J Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating configurable hardware from parallel patterns. *ACM SIGARCH Computer Architecture News*, 44(2):651–665, 2016. (Cited on page 26.)

[PLV05] François Pêcheux, Christophe Lallement, and Alain Vachoux. Vhdl-ams and verilog-ams as alternative hardware description languages for efficient

modeling of multidiscipline systems. *IEEE transactions on Computer-Aided design of integrated Circuits and Systems*, 24(2):204–225, 2005. (Cited on page 44.)

[PM11] Oliver Pell and Oskar Mencer. Surviving the end of frequency scaling with reconfigurable dataflow computing. *ACM SIGARCH Computer Architecture News*, 39(4):60–65, 2011. (Cited on page 35.)

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004. (Cited on page 46.)

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming*, pages 419–443. Springer, 1997. (Cited on page 32.)

[PWSR18] Rafael Trapani Possignolo, Sheng Hong Wang, Haven Skinner, and Jose Renau. Lgraph: A multi-language open-source database for vlsi. In *WOSET: Workshop on Open-Source EDA technology 2018*, 2018. (Cited on page 39.)

[RKH+20] Antti Rautakoura, Matti Käyrä, Timo D Hämäläinen, Wolfgang Ecker, Esko Pekkarinen, and Mikko Teuho. Kamel: Ip-xact compatible intermediate meta-model for ip generation. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 325–331. IEEE, 2020. (Cited on page 26.)

[RMMV10] Erik Rubow, Rick McGeer, Jeff Mogul, and Amin Vahdat. Chimpp: A click-based programming and simulation environment for reconfigurable networking hardware. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, page 36. ACM, 2010. (Cited on page 26.)

[RPPS19] Simon Rokicki, Davide Pala, Joseph Paturel, and Olivier Sentieys. What you simulate is what you synthesize: Designing a processor core from c++ specifications. In *ICCAD 2019 - 38th IEEE/ACMInternational Conference on Computer-Aided Design*, 2019. (Cited on page 45.)

[RSP+05] Adam Rose, Stuart Swan, John Pierce, Jean-Michel Fernandez, et al. Transaction level modeling in systemc. *Open SystemC Initiative*, 1(1.297), 2005. (Cited on page 30.)

[SAW+10] Ofer Shacham, Omid Azizi, Megan Wachs, Wajahat Qadeer, Zain Asgar, Kyle Kelley, John P Stevenson, Stephen Richardson, Mark Horowitz, Benjamin Lee, et al. Rethinking digital design: Why design must change. *IEEE micro*, 30(6):9–24, 2010. (Cited on pages 16 and 34.)

[SB99] Howard Sachs and Mark Birnbaum. Vsia technical challenges. In *Proceedings of the IEEE 1999 Custom Integrated Circuits Conference (Cat. No. 99CH36327)*, pages 619–622. IEEE, 1999. (Cited on page 31.)

[SBL+14] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):134, 2014. (Cited on page 26.)

[SBVW03] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 213–224, 2003. (Cited on page 111.)

[SGS+12] Ofer Shacham, Sameh Galal, Sabarish Sankaranarayanan, Megan Wachs, John Brunhaver, Artem Vassiliev, Mark Horowitz, Andrew Danowitz, Wajahat Qadeer, and Stephen Richardson. Avoiding game over: Bringing design to the next level. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 623–629. IEEE, 2012. (Cited on page 34.)

[Sha11] Ofer Shacham. *Chip multiprocessor generator: automatic generation of custom and heterogeneous compute platforms*. phdthesis, Stanford University, 2011. (Cited on page 34.)

[SHR+19] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. μir-an intermediate representation for transforming and optimizing the microarchitecture of application accelerators. *MICRO-52*, 2019. (Cited on page 68.)

[SJR+16] R Sethulekshmi, S Jazir, Riyaz A Rahiman, Ragipati Karthik, MS Abdulla, et al. Verification of a risc processor ip core using systemverilog. In *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, pages 1490–1493. IEEE, 2016. (Cited on page 44.)

[SKGB20] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. Llhd: a multi-level intermediate representation for hardware description languages. *arXiv preprint arXiv:2004.03494*, 2020. (Cited on pages 18, 39, and 44.)

[Ski18] Haven Blake Skinner. *Pyrope: A Latency-Insensitive Digital Architecture Toolchain*. PhD thesis, UC Santa Cruz, 2018. (Cited on page 38.)

[SM13] Stuart Sutherland and Don Mills. Synthesizing SystemVerilog: busting the myth that SystemVerilog is only for verification. In *Proceedings of SNUG Silicon Valley*, page 24, 2013. (Cited on page 33.)

[SMBD93] Karem A Sakallah, Trevor N Mudge, Timothy M Burks, and Edward S Davidson. Synchronization of pipelines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(8):1132–1146, 1993. (Cited on page 71.)

[SNO13] Won So, Ashok Narayanan, and David Oran. Named data networking on a router: Fast and dos-resistant forwarding with hash tables. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 215–226. IEEE Press, 2013. (Cited on page 50.)

[Sny10] Wilson Snyder. The verilog preprocessor: Force for good and evil. In *Proceedings of SNUG Boston*, 2010. (Cited on page 33.)

[SRAD07] Sameer D Sahasrabuddhe, Hakim Raja, Kavi Arya, and Madhav P Desai. Ahir: A hardware intermediate representation for hardware generation from high-level programs. In *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*, pages 245–250. IEEE, 2007. (Cited on page 39.)

[SRC+19] Pontarelli Salvatore, Bonola Roberto, Bifulcoand Marco, Cascone Carmelo, Spaziani Marco, Bruschi Valerio, Sanvito Davide, Siracusano Giuseppe, Capone Antonio, Honda Michio, Huici Felipe, and Bianchi Giuseppe. Flowblaze: Stateful packet processing in hardware. In *In proceedings of 16th USENIX Symposium on Networked Systems Design and Implementation*, 2019. (Cited on pages 17, 26, and 29.)

[Sta15] Gavin Stark. Introduction to the nfp architecture. In *Proceedings of P4DEVCON 2015, San Jose*, 2015. (Cited on page 12.)

[Str17] Tobias Strauch. An aspect and transaction oriented programming, design and verification language (pdvl). In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 30–39. IEEE, 2017. (Cited on pages 30, 31, and 32.)

[SWS+17] Haven Skinner, Sheng Hong Wang, Akash Sridhar, Rafael Trapani Possignolo, and Renau Jose. Pyrope, a modern hdl with a live flow. Computer Engineering Department, University of California, Santa Cruz, https://masc.soe.ucsc.edu/pyrope.html, 2017. (Cited on page 38.)

[SWW11] Ivan Shcherbakov, Christian Weis, and Norbert Wehn. Bringing c++ productivity to vhdl world: From language definition to a case study. In *Specification and Design Languages (FDL), 2011 Forum on*, pages 1–7. IEEE, 2011. (Cited on page 34.)

[Tay05] David E Taylor. Survey and taxonomy of packet classification techniques. *ACM Computing Surveys (CSUR)*, 37(3):238–275, 2005. (Cited on page 111.)

[Tay18] Michael Bedford Taylor. Basejump stl: systemverilog needs a standard template library for hardware design. In *Design Automation Conference*, 2018. (Cited on pages 31 and 34.)

[TH19] Lenny Truong and Pat Hanrahan. A golden age of hardware description languages: Applying programming language techniques to improve design productivity. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019. (Cited on page 38.)

[THZ20] James Thomas, Pat Hanrahan, and Matei Zaharia. Fleet: A framework for massively parallel streaming on fpgas. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 639–651, 2020. (Cited on page 30.)

[TOJ+19] Cheng Tan, Yanghui Ou, Shunning Jiang, Peitian Pan, Christopher Torng, Shady Agwa, and Christopher Batten. Pyocn: A unified framework for modeling, testing, and evaluating on-chip networks. In *Proceedings of the 37th IEEE Int'l Conf. on Computer Design (ICCD-37), November 2019*, 2019. (Cited on page 45.)

[TY15] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *International Symposium on Applied Reconfigurable Computing*, pages 451–460. Springer, 2015. (Cited on page 35.)

[TŽ17] Andrej Trost and Andrej Žemva. Pipeline circuit synthesis from python code. In *Embedded Computing (MECO), 2017 6th Mediterranean Conference on*, pages 1–4. IEEE, 2017. (Cited on page 26.)

[uja19] HardCaml: Register transfer level hardware design in OCaml. http://www.ujamjar.com/hardcaml, 2019. (Cited on page 35.)

[VCP+20] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020. (Cited on page 13.)

[VER19] Bogdan Vukobratovic, Andrea Erdeljan, and Damjan Rakanovic. Pygears: A functional approach to hardware design. In *Proceedings of OSDA*, 2019. (Cited on page 35.)

[VG14] Girish Venkataramani and Yongfeng Gu. System-level retiming and pipelining. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 80–87. IEEE, 2014. (Cited on page 29.)

[Vis07] Eelco Visser. Webdsl: A case study in domain-specific language engineering. In *International summer school on generative and transformational techniques in software engineering*, pages 291–373. Springer, 2007. (Cited on page 23.)

[VLWA13] Huy Vo, Yunsup Lee, Andrew Waterman, and Krste Asanovic. A case for os-friendly hardware accelerators. In *Workshop on the Interaction between Operating System and Computer Architecture (WIVOSCA), at the International Symposium on Computer Architecture (ISCA)*, 2013. (Cited on page 68.)

[VN93] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993. (Cited on page 22.)

[VRO21] Matthew Vilim, Alexander Rucker, and Kunle Olukotun. Aurochs: An architecture for dataflow threads. In *Proceedings of 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021. (Cited on page 65.)

[WCCM18] An Wang, Wentao Chang, Songqing Chen, and Aziz Mohaisen. Delving into internet ddos attacks by botnets: characterization and analysis. *IEEE/ACM Transactions on Networking*, 26(6):2843–2855, 2018. (Cited on page 2.)

[WGK13] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013. (Cited on page 39.)

[WIS+18] Edward Wang, Adam Izraelevitz, Colin Schmidt, Borivoje Nikolic, Elad Alon, and Jonathan Bachrach. Hammer: Enabling reusable physical design. In *Proceedings of the first Workshop on Open-Source EDA Technology*, 2018. (Cited on page 68.)

[Wol20] Clifford Wolf. PicoRV32 - a size-optimized RISC-V CPU. Retrieved July 2020, 2020. (Cited on page 99.)

[WPC+19] Sheng-Hong Wang, Rafael Trapani Possignolo, Qian Chen, Rohan Ganpati, and Jose Renau. Lgraph: A unified data model and api for productive open-source hardware design. In *Proceedings of Workshop on Open-Source EDA technology (WOSET) 2019*, 2019. (Cited on pages 39 and 44.)

[WRI+18] Angie Wang, Paul Rigge, Adam Izraelevitz, Chick Markley, Jonathan Bachrach, and Borivoje Nikolić. Aced: a hardware library for generating dsp systems. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018. (Cited on page 45.)

[WSR19] Sheng-Hong Wang, Akash Sridhar, and Jose Renau. Lnast: A language neutral intermediate representation for hardware description languages. In *Proceedings of Workshop on Open-Source EDA technology (WOSET) 2019*, 2019. (Cited on pages 39 and 44.)

[XCC21] Zhili Xiao, Roger D Chamberlain, and Anthony M Cabrera. Hls portability from intel to xilinx: A case study. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2021. (Cited on page 17.)

[XLT+22] Mufan Xiang, Yongjian Li, Sijun Tan, Yongxin Zhao, and Yiwei Chi. Parameterized design and formal verification of multi-ported memory. In *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 33–41. IEEE, 2022. (Cited on page 66.)

[XZW+21] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. Hasco: Towards agile hardware and software co-design for tensor computation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1055–1068. IEEE, 2021. (Cited on page 66.)

Abstracting Hardware Architectures for Agile Design of High-performance Applications on FPGA

Abstract — In a context of ever-growing worldwide communication traffic and fast deployment of IoT devices, network attacks have become a daily challenge with record-breaking throughput levels. Compared to software solutions based on general purpose CPUs, FPGA-based mitigation appliances appear as an energy-efficient alternative which combines configurability with guaranteed high-throughput and low-latency. However, implementation of such dedicated hardware accelerators based on the register-transfer level (RTL) abstraction is a much slower and tedious process than functionally equivalent software developments. The latter have indeed benefited from the introduction of countless high-level paradigms over the past decades, whereas traditional hardware description languages (HDLs) have consistently remained rigid and verbose. As a result, the agility gap between hardware and software developments is expanding at a steady pace, leaving hardware design experts frustrated by the lack of re-usability of their carefully crafted architectures.

This thesis tackles this generic hardware development issue within the context of high-performance networking appliance design at OVHcloud. Mimicking the successful trajectory of software evolution, it aims at leveraging a stack of abstraction levels to instill flexibility within hardware descriptions. As a key enabler, Hardware Construction Languages (HCLs) apply some existing software abstractions to hardware design, which permits descriptions of circuit generators with high-level software paradigms, such as object-oriented and functional programming. This thesis exhibits the relevance of both software inherited paradigms and hardware-oriented abstractions to develop highly re-usable network functions, inspecting design, implementation and integration perspectives.

Keywords: *FPGA, EDA, Modelling, Abstraction, HCL, HDL, Network, High-performance*

Thesis prepared between **OVHcloud**
6 bis rue Riquet, 75019, PARIS, France
and **TIMA laboratory**
46 Avenue Félix Viallet, 38031, GRENOBLE Cedex, France