High-level synthesis of instruction set processors

Jean-Michel Gorius

▶ **To cite this version:**

Jean-Michel Gorius. High-level synthesis of instruction set processors. Hardware Architecture [cs.AR]. Université de Rennes, 2024. English. ⟨NNT : 2024URENS068⟩. ⟨tel-04884873v2⟩

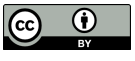
HAL Id: tel-04884873

https://inria.hal.science/tel-04884873v2

Submitted on 26 Feb 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

COLLEGE DOCTORAL BRETAGNE

MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

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

Par

Jean-Michel GORIUS

Synthèse de haut niveau de processeurs à jeu d'instructions

High-Level Synthesis of Instruction Set Processors

Thèse présentée et soutenue à Rennes, le 20 décembre 2024

Rapporteurs avant soutenance :

Luciano LAVAGNO — Full Professor, Politecnico di Torino
Frédéric PÉTROT — Professeur des Universités, Grenoble INP

Composition du Jury :

Examinateurs :	Isabelle PUAUT	Professeure des Universités, Université de Rennes
	Luciano LAVAGNO	Full Professor, Politecnico di Torino
	Frédéric PÉTROT	Professeur des Universités, Grenoble INP
	Lana JOSIPOVIĆ	Assistant Professor, ETH Zürich
	Arthur PERAIS	Chargé de Recherche, TIMA, Grenoble
	Simon ROKICKI	Maître de Conférences, ENS Rennes
Dir. de thèse :	Steven DERRIEN	Professeur des Universités, Université de Bretagne Occidentale

High-Level Synthesis of Instruction Set Processors

Jean-Michel Gorius

2021–2024

Remerciements

Je tiens à remercier toutes les personnes qui ont rendu cette thèse possible. Que ce soit par leur soutien, leurs conseils, ou encore nos discussions, chacune a contribué à faire de cette étape de ma vie un moment empli de découvertes, tant professionnelles que personnelles.

Je remercie tout d'abord mon directeur de thèse, Steven Derrien, qui a su me guider sur le chemin de la recherche académique, depuis mon premier stage jusqu'à l'accomplissement de ces trois années de travail en commun. Merci également à Simon Rokicki, dont l'encadrement et les conseils ont grandement contribué à modeler ma vision de la recherche et de l'enseignement supérieur. Merci à eux pour la confiance qu'ils m'ont accordé en me proposant ce sujet de thèse. Je n'aurais pas pu espérer meilleure équipe d'encadrement.

Merci à Isabelle Puaut, qui a accepté de présider mon jury de soutenance, et merci également à tous les autres membres du jury, Luciano Lavagno, Frédéric Pétrot, Lana Josipović, et Arthur Perais. Leurs questions et leurs commentaires ont permis d'améliorer significativement le contenu de cette thèse. Je remercie tout particulièrement les rapporteurs, Luciano et Frédéric, dont les commentaires ont permis d'éclaircir de nombreux points de détails du manuscrit.

Je suis reconnaissant d'avoir eu l'occasion de travailler au sein d'une équipe de recherche dynamique, aux côtés de nombreuses personnes motivées et talentueuses. Je salue tout particulièrement Amélie, Benoît, Cédric, Corentin, Dylan, François, Guillaume, Hery, Joseph, Laurent, Léo, Léo, Louis, Louis, Olivier, Patrice, Rémi, Romaric, Sami, Seu, et Silviu. Je remercie également Nadia Derrouault, dont la patience et la maîtrise de l'art de résoudre des problèmes administratifs sont sans limites.

Merci aux membres du département informatique de l'ENS de Rennes, qui ont su m'accompagner tout au long de ma scolarité et durant ma thèse, et dont les conseils ont façonné mon approche du travail de recherche et d'enseignement. Merci à David Pichardie, David Baelde, François Schwarzentruber et Martin Quinson. Je les remercie de m'avoir donné l'opportunité de participer au montage de la préparation à l'agrégation d'informatique, une expérience qui m'a appris beaucoup sur l'enseignement, mais également sur moi-même.

Je remercie mes parents, Marc et Elisabeth, ainsi que mon frère, Philippe, pour leur soutien inconditionnel lors de mes études, et tout particulièrement lors de ma thèse. Ils ont toujours su m'encourager et me pousser à me dépasser. Sans eux, cette thèse n'aurait probablement jamais vu le jour.

Enfin, je tiens à remercier ma fiancée, Sarah. Ces trois dernières années n'auraient pas été les mêmes sans toi, sans ton soutien, ton sourire, et tes encouragements. Merci.

Résumé en français

Portée par l'avènement de l'ordinateur personnel et le développement rapide des technologies de micro-processeurs, la révolution informatique a profondément transformé notre société, ainsi que notre manière d'interagir avec le monde. De nos jours, les dispositifs informatiques sont omniprésents : des machines multi-cœurs haute performance dans les serveurs, aux petits dispositifs embarqués faible consommation dans les équipements électroniques portables, les ordinateurs sont partout. L'Internet des objets (IoT) ouvre de nombreuses nouvelles opportunités pour les produits et applications numériques, mais présente également des défis de taille pour les concepteur d'ordinateurs : ces dispositifs doivent traiter des charges de travail computationnelles de plus en plus importantes, tout en respectant des exigences strictes en termes de coûts et d'efficacité énergétique. La grande majorité des plateformes IoT repose sur des familles de micro-contrôleurs (MCU) à faible consommation, basées sur le même jeu d'instruction (ISA), tel que ARM. Différents MCU au sein de la même famille exposent une large gamme de compromis entre consommation énergétique et performance grâce à des micro-architectures distinctes. Les concepteurs de produits peuvent choisir, parmi les implémentations de processeurs disponibles, celles qui conviennent le mieux à leur domaine d'application, en prenant en compte des contraintes de coût, d'énergie, et de performance.

La plupart des MCU existants reposent sur des ISA propriétaires, ce qui empêche les tiers de mettre en œuvre librement des micro-architectures personnalisées ou de s'écarter d'une ISA standardisée, limitant ainsi de possibles innovations. L'initiative RISC-V vise à répondre à cette problématique en développant et en promouvant un jeu d'instructions ouvert, avec son propre ensemble d'outils. L'écosystème RISC-V croît rapidement et suscite un grand intérêt auprès des concepteurs de plateformes IoT, car il permet une personnalisation libre de l'ISA et de la micro-architecture. Le succès croissant de RISC-V met en lumière la nécessité de possibilités de personnalisation dans le paysage numérique actuel. Avec la fin imminente des lois de Moore et de Dennard, les concepteurs doivent envisager des alternatives aux ISA propriétaires bien établies, et la demande pour des processeurs à jeu d'instructions (ISP) spécialisés ne cesse d'augmenter.

Les processeurs à jeu d'instructions sont des composants matériels complexes conçus pour exécuter un flux d'instructions stocké en mémoire externe. Ces processeurs offrent une interface de programmation très flexible, les rendant adaptés aux charges de travail irrégulières et hétérogènes. Les charges de travail contenant des motifs de calcul irréguliers sont inhérentes aux applications mobiles tout comme au calcul haute performance, où la plupart des applications sont dominées par leur flot de contrôle. En revanche, dans les domaines embarqués et IoT, les scénarios de calcul typiques impliquent généralement peu de variabilité et de contrôle, mais se concentrent sur le traitement de grandes quantités de données. Plutôt que de viser la programmabilité et la flexibilité, des accélérateurs matériels dédiés sont conçus pour réduire la consommation énergétique et augmenter les performances sur un ensemble bien défini d'applications (par exemple, traitement du signal, encodage et décodage vidéo). Cependant, la tendance dans les domaines de calcul spécialisé évolue progressivement. De nouvelles applications telles que l'exploration de grands volumes de données, l'analyse de graphes et l'apprentissage automatique introduisent de nouveaux besoins computationnels qui exigent du matériel dédié offrant une interface programmable et fonctionnant sur des charges de travail au flot de contrôle irrégulier. Répondre à ces nouvelles exigences pose des défis aux fabricants de matériel, qui doivent concevoir des interfaces programmables avec de nombreuses fonctionnalités personnalisées tout en maintenant des temps de

traitement rapides et une faible consommation d'énergie. Cependant, la conception des ISP reste un processus fastidieux et sujet à de nombreuses erreurs potentielles, qui doit être mené avec soin pour éviter les dysfonctionnements matériels une fois le produit finalisé.

De plus, une même spécification de jeu d'instructions peut être utilisée comme base pour concevoir une grande variété d'ISP. Ce paysage de conception s'étend des dispositifs très faible consommation basés sur des micro-architectures peu énergivores, aux processeurs haute performance à exécution dans le désordre (OoO). Les processeurs pipelinés basés sur des pipelines à exécution dans l'ordre, à une ou plusieurs voies, sont également courants dans les équipements connectés. Cette diversité d'implémentations crée un vaste espace de conception qui englobe des compromis entre la surface de la puce, sa consommation d'énergie, et sa performance. La complexité inhérente à l'exploration de cet espace en fait une cible d'automatisation idéale.

Le rythme rapide auquel les ordinateurs ont évolué au cours des dernières décennies peut s'expliquer par quelques innovations clés dans la conception des architectures de processeurs [284], dont la *spéculation* est un pilier fondamental. La spéculation est une méthode utilisée pour découvrir du parallélisme dans les programmes, en permettant au processeur de prédire le résultat de l'exécution d'une instruction donnée avant même que celle-ci ne soit complètement exécutée. Cette optimisation permet aux concepteurs de processeurs d'accélérer l'exécution du cas le plus courant. Combinée avec le pipelining, la spéculation augmente considérablement les performances des processeurs modernes [284, 154, 248]. Il est communément admis que la spéculation est indispensable aux processeurs superscalaires à exécution dans le désordre, mais même les processeurs pipelinés à exécution dans l'ordre spéculent lorsqu'ils lisent la prochaine instruction à exécuter en mémoire pour remplir leur pipeline. Le comportement matériel complexe qui découle de l'exécution spéculative fait de cette dernière un défi de conception majeur qui ne peut pas être automatisé efficacement avec les outils de conception matérielle actuels.

Bien que la question de la personnalisation et du reciblage des compilateurs vers un nouveau jeu d'instructions ait été étudiée en détails à la fin des années 1990, et que les infrastructures de compilateurs modernes telles que LLVM [217] offrent de nombreux outils qui facilitent ce processus, la synthèse automatique de micro-architectures a reçu relativement peu d'attention. Les premières approches visant à automatiser la conception des ISP ont été proposées dans le cadre des flots de conception d'*Application-Specific Instruction Set Processors* (ASIP). Les ASIP sont des processeurs ciblant un domaine d'application particulier, dont les choix de conception sont fortement liés aux contraintes de ce domaine. L'ISA des ASIP est adaptée à une application spécifique, exposant des instructions spécialisées et se concentrant sur un petit ensemble de tâches computationnelles. Les approches proposées pour la conception automatisée des ASIP reposent souvent sur des *Domain-Specific Languages* (DSL) pour modéliser la structure matérielle du processeur, ainsi que son ISA [73, 162, 196]. Le niveau d'abstraction fourni par les outils de synthèse d'ASIP est souvent proche de celui des langages de description matérielle (*Hardware Description Languages*, ou HDL) standards, tels que Verilog et VHDL. Ces derniers nécessitent une gestion explicite des choix micro-architecturaux tels que la profondeur et l'organisation du pipeline, les points de réacheminement de données (*forwarding*) disponibles, ainsi que la logique de détection et de gestion des aléas d'exécution.

Parallèlement, les outils de synthèse de haut niveau (*High-Level Synthesis*, ou HLS) qui compilent directement du code dans un langage de haut niveau en circuits matériels, n'ont cessé de s'améliorer. La HLS a été initialement proposée pour surmonter les nombreuses limitations des flots de conception matérielle basés sur les HDL. Dans ces derniers, le matériel généré par les outils de synthèse ne satisfont pas toujours aux contraintes du concepteur, nécessitant la réécriture de grandes parties de la spécification pour obtenir le résultat escompté. Ces limitations rendent l'exploration de l'espace de conception fastidieuse, voire parfois impossible. Contrairement à la synthèse de matériel basée sur les HDL, la HLS permet à son utilisateur de spécifier le comportement d'un circuit dans un langage de haut niveau (souvent C ou C++), et de se concentrer sur la spécification algorithmique du comportement du matériel. A partir de cette description de haut niveau, une chaîne d'outils HLS déduit la structure matérielle, ainsi que les besoins en ressources et les contraintes de fréquence d'horloge nécessaires au bon fonctionnement du circuit. Cette approche abstrait l'utilisateur de la plupart des détails d'implémentation et facilite l'application de modifications et personnalisations au circuit généré.

Plusieurs résultats de recherche récents montrent que les techniques de HLS peuvent être étendues

pour synthétiser des structures matérielles spéculatives efficaces [97, 186]. En particulier, le *pipeline de boucle spéculatif* (*Speculative Loop Pipelining*, ou SLP) semble être une approche prometteuse capable de gérer à la fois la spéculation de flot de contrôle et la spéculation mémoire dans le cadre des contraintes imposées par un flot de conception HLS commercial standard. Les possibilités d'exécution spéculative introduites par SLP dans la HLS, combinées aux capacités d'exploration d'espace de conception de cette dernière, ouvrent la voie à la conception automatique de processeurs à jeu d'instructions à partir d'une description algorithmique de leur comportement.

Contributions

Les travaux décrits dans cette thèse portent sur la synthèse automatique de processeurs à jeu d'instructions, dans le cadre de la synthèse de haut niveau (HLS). En particulier, nous cherchons à générer automatiquement des processeurs pipelinés à exécution dans l'ordre à partir d'une description de haut niveau en C, sous la forme d'un simulateur de jeu d'instructions (*Instruction Set Simulator*, ou ISS).

Au cours de notre travail, nous avons développé un flot de conception matérielle entièrement automatisé capable de compiler une description algorithmique en un circuit spéculatif. Nous proposons un ensemble de transformations source-à-source qui s'appuient sur le pipeline de boucle spéculatif pour découvrir des opportunités de spéculation sur le flot de contrôle et la mémoire dans du code C, et nous générons du code spéculatif qui peut être synthétisé par une chaîne d'outils commerciaux standards, AMD Vitis HLS. Nous intégrons ces transformations dans l'infrastructure de compilation GeCoS [124], développée à l'IRISA, pour créer notre flot de conception matérielle, *SpecHLS*. Ce dernier est capable de gérer plusieurs spéculations entremêlées, des spéculations indépendantes dans des modules matériels découplés, ainsi que des spéculations sur les accès à la mémoire. Nous montrons que SpecHLS génère des circuits rapides, capables de rivaliser et de surpasser des circuits conçus et optimisés manuellement dans des HDL standards.

La Figure 1 donne un aperçu du flot de conception matérielle décrit dans cette thèse. Ce flot peut être divisé en cinq parties successives :

— *Pré-transformations.* Nous appliquons plusieurs transformations préliminaires au code d'entrée ❶, qui peuvent être contrôlées via des annotations fournies par l'utilisateur. Nous nous appuyons sur l'infrastructure GeCoS pour fournir des transformations source-à-source courantes, telles que le déroulage de boucles et la partition automatique de tableaux. Ces transformations permettent d'exposer plus d'opportunités de spéculation directement dans la face avant du compilateur. Ces dernières peuvent ensuite être exploitées par les étapes suivantes de notre flot pour améliorer le temps d'exécution du matériel synthétisé.

— *Extraction de l'IDG.* Nous transformons le code source d'entrée en une variante de la forme SSA [85], appelée *Gated-SSA* (GSSA) [278, 367, 366], afin de faciliter les transformations et l'analyse de programmes. Notre travail étend la forme GSSA standard pour permettre la gestion de la spéculation mémoire, ainsi que la représentation de délais micro-architecturaux dans la logique de gestion de la spéculation (par exemple, dans des tampons permettant l'annulation d'itérations exécutées après une mauvaise spéculation). Des exemples de GSSA sont données dans les parties ❺ et ❻ de la Figure 1.

— *Analyse.* Cette phase du processus de compilation effectue deux formes distinctes d'analyse sur la représentation GSSA. La première détermine où la spéculation mémoire peut être appliquée, et quels tableaux interagissent avec des opérateurs s'exécutant de manière spéculative. Elle expose ensuite des opportunités de spéculation mémoire sous la forme d'une file d'attente (*Store Queue*, ou SQ), permettant ainsi aux étapes suivantes du flot de conception SpecHLS de spéculer sur l'absence d'alias intra ou inter-itérations entre accès tableaux. La seconde passe d'analyse effectue une exploration de l'espace de conception pour déterminer où et comment spéculer dans le circuit final. Bien qu'une énumération exhaustive de toutes les configurations de spéculation possibles soit réalisable pour de petits circuits, une telle approche ne passe pas à l'échelle pour des cœurs de processeur complets. À la place, nous proposons un algorithme d'exploration qui repose sur une analyse statique des délais, ainsi que plusieurs heuristiques, pour accélérer le développement

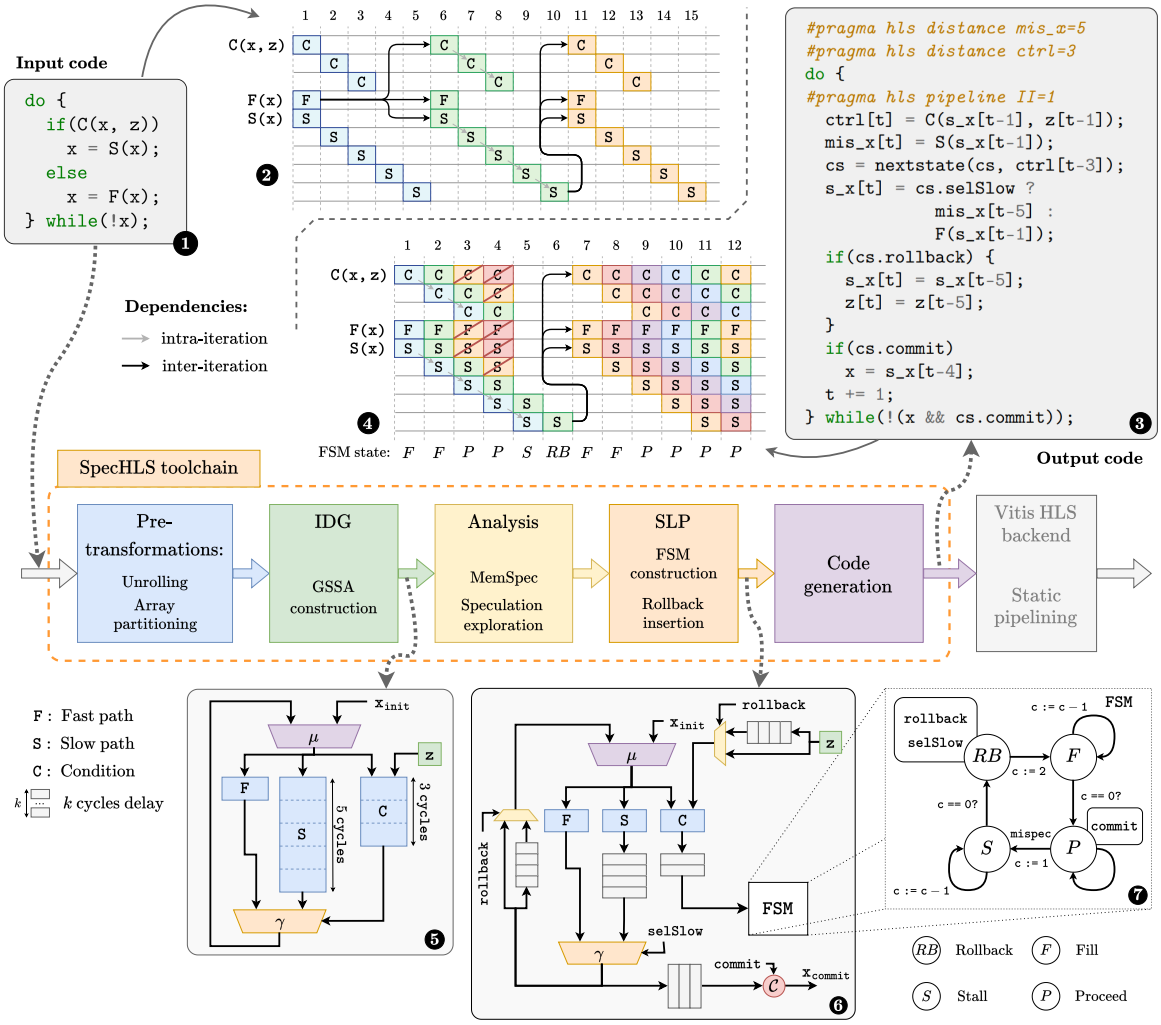


Figure 1 – Vue d'ensemble du flot de conception matérielle SpecHLS.

de matériel spéculatif complexe à l'aide de notre chaîne d'outils.

— *SLP.* L'étape qui précède la génération de code insère la logique de contrôle de la spéculation dans la représentation GSSA du programme d'entrée. Cette logique se compose d'une machine à états (*Finite State Machine*, ou FSM) qui pilote l'exécution du matériel spéculatif, ainsi que d'une logique de récupération sous forme de tampons. Nous étendons les idées initiales du pipeline de boucle spéculatif pour gérer plusieurs spéculations imbriquées, et adaptons en conséquence les mécanismes de construction de la FSM et de génération de la logique de récupération en cas de mauvaise spéculation.

— *Génération de code.* L'étape finale de SpecHLS génère du code pour la face arrière de notre flot, Vitis HLS. Le code généré (illustré dans la partie ❸ de la Figure 1) expose des distances de réutilisation explicites pour permettre un pipelining statique efficace des boucles par la face arrière. Nous générons également du code compatible avec la synthèse de haut niveau pour la logique de gestion des mauvaises spéculations, ainsi que pour la spéculation mémoire. Le code généré contient des directives qui permettent de guider la face arrière et d'activer davantage d'optimisations.

La chaîne d'outils SpecHLS génère du code C transformé, qui est ensuite envoyé à une chaîne d'outils HLS commerciale, que nous considérons comme notre face arrière. Cette dernière étape nous permet de

bénéficier d'optimisations spécifiques à la cible, implémentées par les fabricants de matériel pour leurs plateformes.

Bien que SpecHLS puisse être utilisé pour la conception d'accélérateurs spéculatifs à usage général, nous mettons particulièrement l'accent sur ses capacités de synthèse de processeurs à jeu d'instructions. Le résultat de notre travail est un flot de conception de processeurs complet, capable de générer plusieurs instances de processeurs RISC-V pipelinés à exécution dans l'ordre à partir d'une seule entrée de haut niveau sous forme de simulateur de jeu d'instructions. Nous montrons que nous pouvons explorer efficacement un espace de conception avec plusieurs centaines de milliers de configurations matérielles spéculatives possibles en quelques minutes, et que nous sommes capables de générer des processeurs compétitifs avec des cœurs de processeurs embarqués de l'état de l'art.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

So it begins.
— *Théoden, King of Rohan* (*The Two Towers*, J.R.R. Tolkien)

Spearheaded by the personal computer's advent and the rapid development of microprocessor technology, the computing revolution has profoundly transformed our society and the way we interact with the world. Nowadays, computing devices have become omnipresent: from high-performance multi-core machines in high-end servers to small, low-power embedded devices in wearable products, computers are everywhere. The Internet of Things (IoT) opens many new opportunities for digital products and applications but comes with challenges for computer designers: devices are expected to handle increasingly large computational workloads while enforcing stringent cost and energy efficiency. The vast majority of IoT platforms rely on low-power Micro-Controller Unit (MCU) families supporting the same Instruction Set Architecture (ISA), *e.g.*, ARM. Different MCUs in the same family expose a wide variety of energy to performance tradeoffs thanks to distinct micro-architectures. Product designers can choose from the available processor designs that best suit their application domain, considering price, power, and performance constraints.

Most existing MCUs rely on proprietary ISAs, which prevent third parties from freely implementing their customized micro-architecture or deviating from a standardized ISA, thereby hindering innovation. The RISC-V initiative is an effort to address this issue by developing and promoting an open instruction set architecture with its own set of tools. The RISC-V ecosystem is quickly growing and has gained much traction from IoT platform designers, as it permits free customization of both the ISA and the micro-architecture. The ever-growing success of RISC-V highlights the need for customization in today's digital landscape. With the approaching end of Moore's Law and Dennard Scaling, designers need to consider alternatives to well-established proprietary ISAs as the demand for specialized Instruction Set Processors (ISP) continues to rise.

Instruction set processors are intricate pieces of hardware designed to execute a stream of instructions stored in external memory. These processors offer a highly flexible programming interface, making them well-suited for irregular and heterogeneous workloads. Irregular workloads are inherent to desktop, mobile, and high-performance computing, where most applications are control-dominated. On the other hand, in the embedded and IoT spaces, typical computing scenarios usually involve little variability and control but focus on processing large quantities of data. Instead of targeting programmability and flexibility, special-purpose hardware accelerators are designed to reduce power consumption and increase performance on a single well-defined set of applications (*e.g.*, signal processing, video encoding, and decoding). However, the trend in specialized computational domains is slowly shifting. New applications such as data mining, graph analytics, and machine learning introduce new computational needs that

require special-purpose hardware to provide a programmable interface and operate on control-flow-dominated workloads. Addressing these new requirements challenges hardware manufacturers to design programmable hardware with many custom features while still providing fast processing times and reduced energy consumption. However, the design of ISPs is a tedious and error-prone process that needs to be conducted carefully to prevent the hardware from misbehaving once it is produced.

Additionally, a single instruction set specification can be used to design a wide variety of ISPs. This design landscape spans from very low-power devices based on low-energy micro-architectures to high-performance Out-of-Order (OoO) processors. Pipelined designs based on single-issue or multiple-issue in-order pipelines are also widespread in connected devices. This implementation diversity leads to an ample design space that encompasses tradeoffs between die area, power consumption, and performance. The inherent complexity of design space exploration makes it a good target for design automation.

The fast pace at which computers evolved during the past few decades can be explained by a few key innovations in processor architecture design [284], of which *speculation* is a core pillar. Speculation is a method used to uncover parallelism in programs by letting the processor "guess" the outcome of the execution of a given instruction before it finishes executing. This optimization allows processor designers to *make the common case fast*. Combined with pipelining, speculation drastically increases the performance of modern processor cores [284, 154, 248]. Conventional wisdom is that speculation is used in superscalar OoO processors, but even in-order pipelined processors speculate when fetching the next instruction to fill their pipeline. The intricate hardware behavior that emerges from speculative execution makes the latter a significant design challenge that cannot be automated efficiently with today's hardware synthesis tools.

While the problem of customizing and retargeting compilers to a new instruction set extension has been widely studied in the late 1990s, and modern compiler infrastructures such as LLVM [217] now offer many facilities to ease this process, the automatic synthesis of micro-architectures has received comparatively little attention. The first approaches aimed at automating the design of ISPs were proposed in the context of Application-Specific Instruction Set Processor (ASIP) design flows. ASIPs are programmable processing cores targeting a particular application domain. The ISA of ASIPs is tailored to a specific application, exposing special-purpose instructions and focusing on a small set of tasks. Proposed approaches for automated ASIP design often rely on Domain-Specific Languages (DSL) to model the processor's hardware structure, along with its ISA [73, 162, 196]. The abstraction level provided by ASIP synthesis tools is often very close to standard Hardware Description Languages (HDL), and asks for explicit management of micro-architectural choices such as pipeline depth and organization, available data forwarding points, and hazard detection and recovery logic.

In the meantime, High-Level Synthesis (HLS) toolchains, which compile high-level code directly into hardware circuits, have continuously improved. HLS was first proposed to overcome the many limitations of HDL-based hardware design flows. In the latter, iterative rewrites of large specification parts are required to achieve design goals that depend on customer or application-defined constraints. These limitations make design space exploration tedious, sometimes even rendering it impractical. Contrary to HDL-based hardware synthesis, HLS allows its user to specify the behavior of a given piece of hardware in a high-level language (often C or C++) and to focus on the algorithmic specification of the hardware's operation. An HLS toolchain infers the hardware's structure from its high-level description, hardware resource requirements, and clock frequency constraints. This approach abstracts the user away from most implementation details and makes changes to the hardware more straightforward.

Several recent research results show that HLS techniques can be extended to synthesize efficient speculative hardware structures [97, 186]. In particular, *speculative loop pipelining* (SLP) [97] appears as a promising approach that can handle both control flow and memory speculation within the constraints of a standard HLS design flow. The newfound speculative execution capabilities introduced by SLP to HLS, combined with the design space exploration capabilities of the latter, pave the way for the automatic design of instruction set processors from an algorithmic specification of their behavior.

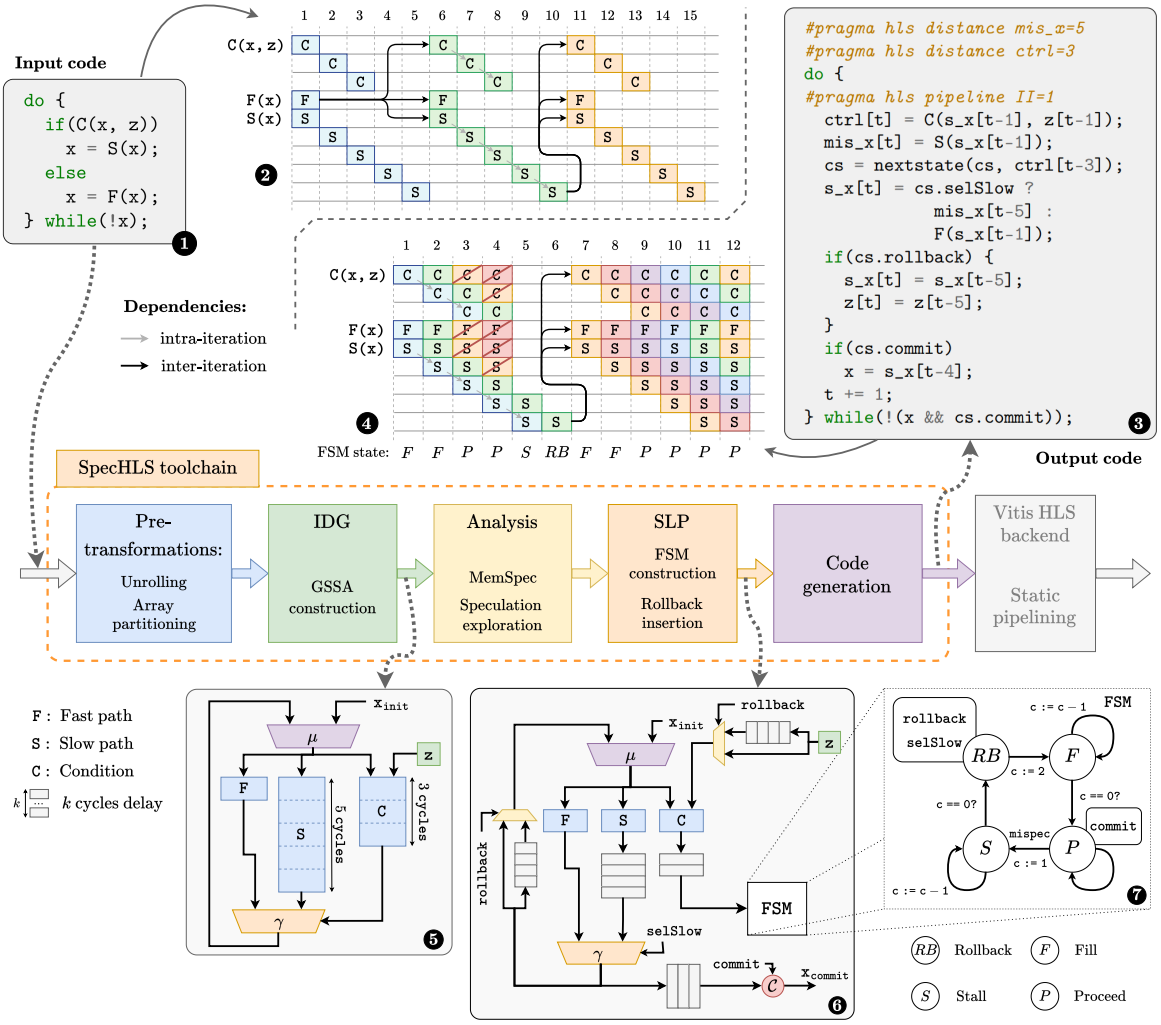


Figure 1.1 – Overview of the SpecHLS hardware design flow presented in this thesis. The details exposed in this overview are discussed in Chapter 5.

1.1 Contributions

The work described in this thesis focuses on the automatic synthesis of instruction set processors in High-Level Synthesis. In particular, we aim to automatically generate in-order pipelined processor cores from a high-level description in C in the form of an Instruction Set Simulator (ISS).

Our work has led to the development of a fully automated hardware design flow for speculative hardware accelerators and processors. It can compile an algorithmic description to speculative hardware, offering a set of source-to-source transformations that build on top of speculative loop pipelining to expose control-flow and memory speculation opportunities in C code. The generated speculation-enabled code can be synthesized using the commercial AMD Vitis HLS toolchain. We integrate these transformations into the GeCoS compiler infrastructure [124] to create our *SpecHLS* hardware design flow. The latter can handle multiple interacting speculations, independent speculations in decoupled hardware modules, and memory speculation. We show that SpecHLS generates fast hardware designs that can compete with and outperform manually optimized RTL designs in standard HDLs. Multiple speculation handling is described in more detail in Chapter 5, while memory speculation is discussed in Chapter 6.

Figure 1.1 gives an overview of the hardware design flow described in this thesis. It can be split into

five successive stages:

— *Pre-transformations.* We apply several preliminary transformations to the input code ❶, which can be controlled via user-provided annotations. We rely on the GeCoS infrastructure to provide common source-to-source transformations, such as loop unrolling and array partitioning. These transformations allow more speculation opportunities to be exposed directly in the compiler's frontend, which later stages can exploit to improve the execution time of the synthesized hardware.

— *IDG extraction.* We transform the input source code into a variant of SSA form [85] named Gated-SSA (GSSA) [278, 367, 366] to make program transformations and analysis easier. Our work extends the standard GSSA form to allow for memory speculation handling and the representation of micro-architectural delay operators in speculation handling logic (*e.g.*, in rollback buffers). Examples of GSSA are depicted in parts ❺ and ❻ of Figure 1.1.

— *Analysis.* This phase of the compilation process performs two distinct forms of analysis on the GSSA representation. The first analysis pass determines where memory speculation (see Chapter 6) can be applied and which arrays interact with speculatively executing operators. It then exposes memory speculation opportunities in the form of a Store Queue (SQ), which allows further stages of the SpecHLS flow to speculate on the absence of inter-or intra-iteration aliases between array loads and stores. The second analysis pass performs design space exploration to determine where and how to speculate in the final design. While an exhaustive enumeration of all possible speculation configurations is feasible on small designs, such an approach does not scale to fully-featured processor cores. Instead, we propose an exploration algorithm that relies on static timing analysis and heuristics to accelerate the development of complex speculative hardware using our toolchain. These scalability issues are discussed in Chapter 7.

— *SLP.* The stage that precedes code generation inserts the speculation control logic into the GSSA representation of the input program. This logic comprises a Finite State Machine (FSM) that pilots the execution of the speculative hardware and recovery logic in the form of rollback buffers. We extend the initial ideas of SLP to support multiple intertwined speculations and adapt the FSM construction and rollback generation mechanisms accordingly.

— *Code generation.* The final stage of the SpecHLS flow generates code for the HLS backend. The generated code (depicted in part ❸ of Figure 1.1) exposes explicit reuse distances to allow for efficient static loop pipelining by our backend. We also generate synthesis-friendly code for the rollback logic and memory speculation handling, along with directives to enable more backend optimizations.

The SpecHLS toolchain generates transformed C code that is then sent to a commercial HLS toolchain, which we consider our backend. This last step allows us to benefit from target-specific optimizations implemented by hardware vendors for their platforms.

While SpecHLS can be used for general-purpose speculative accelerator design, we emphasize its instruction set processor synthesis capabilities. Our work results in an end-to-end processor design flow capable of generating multiple instances of in-order RISC-V processors from a single high-level input in the form of an instruction set simulator. We show that we can efficiently explore a design space with several hundreds of thousands of possible speculative hardware configurations in a matter of minutes and that we can generate processor designs that are competitive with state-of-the-art embedded MCU-class cores.

1.2 Content Overview

This thesis is split into seven chapters. Following the current introductory chapter, Chapter 2 discusses the levels of abstraction involved in building a modern processor core. It presents fundamental concepts in digital electronics design, from logic gates to instruction set processor micro-architecture. It also discusses common hardware implementation platforms, focusing on Field Programmable Gate Arrays (FPGA), which are the main target of our work. Chapter 3 presents automated hardware design and its history. It presents standard hardware description languages, such as Verilog and VHDL,

and their extensions, as well as newer languages that introduce higher abstraction levels to hardware design. Chapter 3 also discusses High-Level Synthesis before introducing Architecture Design Languages (ADL), special-purpose languages created for processor design automation. Chapter 4 dives into the fundamental problem of scheduling in the context of HLS-based hardware design. It exposes state-of-the-art scheduling techniques used in modern HLS toolchains and discusses their limits. Dynamically-scheduled and speculatively-scheduled HLS are then presented as ways to overcome the limitations of static schedules. Speculative scheduling forms the basis of our work on the SpecHLS toolchain, presented in Chapter 5. In this chapter, we discuss the design of our hardware synthesis flow, with a particular focus on handling multiple interacting speculations, which lie at the heart of efficient processor designs. Chapter 6 follows with an exposition of a unified framework for memory dependency handling in a speculative HLS context. This second contribution unlocks the full potential of HLS for processor design by enabling both memory and value speculations to coexist in a single hardware design. Finally, Chapter 7 synthesizes our contributions and presents initial results obtained using SpecHLS for the automatic design space exploration and synthesis of RISC-V processor cores. It also presents our latest contributions on accelerated design space exploration for speculative HLS. Chapter 8 concludes this thesis and opens new research perspectives for future work on SpecHLS.

1.3 Typographical Conventions

This thesis uses several different styles of text to distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown using a teletype font, such as in "The `max` function searches and returns the maximum value from the `n`-element array `x`." Code blocks are typeset on a gray background, as follows.

```
rv32proc processor = rv32proc_init(/* ... */);
rv32instr instr;
do {
  uint32_t raw_instr = rv32proc_fetch(&processor);
  instr = rv32proc_decode(raw_instr);
} while(rv32proc_execute(&processor, &instr));
```

When particular attention should be given to a particular section of a code block, we will mark the relevant section with a squared number 1. A similar method is used to refer to certain sub-parts of figures, using circled numbers ❶ and letters ⓐ.

Notes and ancillary information are shown in boxes with an information icon.

Related work and interesting reads that are outside the scope of this thesis are shown in boxes with a magnifying glass icon.

Caveats and information that requires particular attention are highlighted in boxes with a danger sign icon.

CHAPTER 2

From Logic Gates to Processors

We are stuck with technology when what we really want is just stuff that works.
— *Douglas Adams* (The Salmon of Doubt)

Processors are complex pieces of electronic hardware that can perform demanding and diverse computational tasks very efficiently. They are the results of several decades of innovation in hardware design. From simple electronic circuits built from interconnected logic gates to highly efficient Out-of-Order processor cores with multiple cache levels, processor design is primarily a work of abstraction layering. In fact, *Use abstraction to simplify design* is one of the eight great ideas in computer architecture listed in the classic introductory computer architecture textbook by David Patterson and John Hennessy, *Computer Organization and Design* [284]. This chapter discusses the levels of abstraction involved in building a modern processor core. We start from the foundational building blocks of hardware design in Section 2.1, discussing combinational and sequential logic, synchronous and asynchronous designs, and static timing analysis. Building on this foundation, we then give an overview of hardware accelerators (Section 2.2) and some of their applications before diving into the topic of instruction set processors in Section 2.3, where we discuss processor architecture and micro-architecture. Finally, we end this chapter by discussing hardware implementation platforms (Section 2.4) such as ASICs and FPGAs.

2.1 Foundational Building Blocks

The field of electronics can be subdivided into two main sub-fields: *analog* electronics and *digital* electronics. The former deals with analog signals: continuous signals that can take on an arbitrary number of values, correlated to a time-varying, *analogous*, physical quantity. The latter deals with digital signals expressing information using a finite set of discrete values. A key advantage of digital signals over their analog counterpart is that they can be transmitted without being degraded by noise from the environment [160]. This intrinsic error correction capacity and the ability to design cheap and fast information storage solutions has led to the prevalence of digital electronics in most modern consumer electronics and computing infrastructure. While analog electronics is a fascinating field of study [160, 298], we will only focus on digital electronics in the following sections and chapters.

Section 2.1.1 presents the concepts of combinational and sequential logic, which form the basis of digital electronic circuits. Section 2.1.2 then dives deeper into sequential logic and discusses synchronous and asynchronous sequential logic. Section 2.1.3 introduces Static Timing Analysis, a technique used to estimate the timing requirements of a circuit without having to perform a full simulation of its behavior. It also gives an overview of operator pipelining, an essential concept for later chapters on scheduling and hardware synthesis techniques.

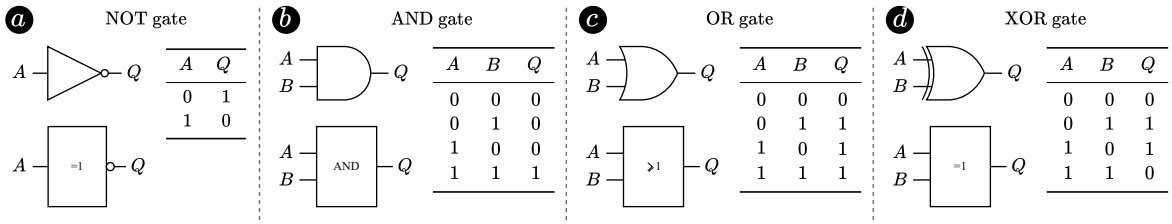


Figure 2.1 – Elementary logic gates. The top symbol gives the *distinctive shape* variant of each logic gate, and the bottom one is its *rectangular shape* counterpart. Both symbol types are standardized by IEEE/ANSI 91a-1991 [166]. The NAND (resp. NOR) gate symbol can be built from the AND (resp. OR) symbol by prepending an inverter node (○) to its output. The truth table for each operator is given on the right of its corresponding column.

2.1.1 Combinational and Sequential Logic

Digital electronic circuits are built from large assemblies of logic gates. Those gates are composed of *transistors* and are the foundation of all modern computing. A *logic gate* is a device that performs a Boolean function on some binary input and produces a binary result. Figure 2.1 illustrates the most common logic gates and their respective truth tables. As with Boolean logic, the gates depicted in Figure 2.1 do not represent a minimal *functionally complete* set of operators. $\{\text{AND}, \text{NOT}\}$, but also $\{\text{NAND}\}$, and $\{\text{NOR}\}$, are functionally complete sets of operators, *i.e.*, these operators can be used to build arbitrarily complex Boolean functions.

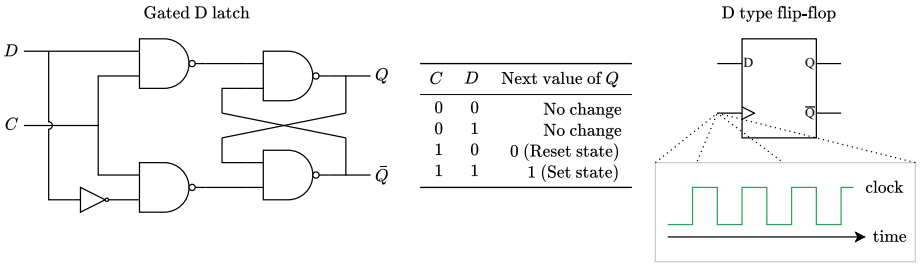
NAND and NOR gates. Digital circuit designers mostly employ NAND and NOR gates as building blocks for Boolean functions. This choice is primarily dictated by the underlying transistor technology, where the physical implementation of NAND and NOR gates uses less surface area and consumes less power than the gates depicted in Figure 2.1 [244].

The circuits one can build using logic gates produce an output signal that depends only on the value of the circuit's inputs. In other words, logic gate combinations can only express *pure functions*. This kind of circuit is commonly referred to as *combinational logic*. Another crucial part of digital electronic circuits is *sequential logic*. Sequential logic circuits produce an output that depends on the circuit's inputs and some past state of the circuit: those circuits can employ memory (*i.e.*, stateful) components to store values and retrieve them at a later stage of the computation. The primary use of sequential logic in digital circuits is to build Finite State Machines (FSM) to drive the execution of combinational logic blocks [244].

2.1.2 Synchronous and Asynchronous Sequential Logic

Two kinds of sequential logic implementations are distinguished by the way stateful components are updated in the circuit [160, 244]. On the one hand, *asynchronous* circuits rely on a *handshaking* mechanism to coordinate execution between multiple circuit components. On the other hand, *synchronous* circuits use a regular *clock* signal to synchronize updates across the entire circuit.

Designing complex asynchronous circuits requires more careful consideration than synchronous circuits, as their behavior is heavily influenced by the time required for computations in combinational logic. Additionally, asynchronous components must consider the possibility of their input signals propagating at different speeds in the circuit, leading to de-synchronized input arrival times. This inherent complexity has led most hardware designers to favor synchronous circuits over the past decades. However, with the advent of Electronic Design Automation, asynchronous designs have started to gain some relevance. The ease with which asynchronous circuits can be composed, as well as their execution flexibility, has led to a growing interest from academia and industry (see Section 4.2 for more details).



C	D	Next value of Q
0	0	No change
0	1	No change
1	0	0 (Reset state)
1	1	1 (Set state)

Figure 2.2 – Gated D latch, and the corresponding D type flip-flop. The value of the D latch output Q can only change when C is set. When C is set, the value of D is stored by the latch. The D type flip-flop transforms the level-triggered latch into an edge-triggered stateful component. Each time the clock pulse produces a rising edge, the flip-flop's internal value updates to reflect the value of D.

Synchronous circuits rely on a device called a *clock generator* to coordinate execution between different parts of the circuit. This generator produces periodic *clock pulses* and is linked to every stateful element of the circuit. The latter can only change the values they store when the clock signal changes from a low value to a high value (*rising edge*) or when it changes from a high to a low signal value (*falling edge*). The most straightforward stateful component of synchronous circuits is called a *latch*. It can store a single bit of information. It becomes a *flip-flop* when coupled to a clock signal. Figure 2.2 gives the circuit diagram for a standard latch, the gated D latch, its truth table, and the corresponding D flip-flop symbol. The gated D latch is built from four NAND gates, and a NOT gate, driven by the control signal C and the data signal D. Note that the latch can output both the stored value Q and the negation of the stored value, $\bar{Q}$.

We can easily extend the flip-flop to an arbitrary k-bit wide storage component by stacking k flip-flops, with each bit of the input stored in one flip-flop and all clock signals linked to the same clock source. This flip-flop assembly is called a *register*. Registers are at the core of complex synchronous hardware designs.

Register-Transfer Level. The register abstraction discussed in this section forms the basis of the *Register-Transfer Level* (RTL) representation of synchronous circuits. This representation is an abstract model that describes an electronic circuit as a flow of digital signals through combinational operations between hardware registers. This model underpins most Hardware Design Languages, which are discussed in more detail in Chapter 3.

2.1.3 Static Timing Analysis

Combined with a steady clock signal, combinational and sequential logic allow us to design complex synchronous digital electronic circuits, such as arithmetic operators, timers, digital signal processors, and, ultimately, fully programmable instruction set processors. However, simply wiring up components that "do the right thing" is insufficient to produce good hardware designs. The critical insight is that, as described in Section 2.1.2, synchronous circuits operate under the constraint that stateful elements may only update when a new clock pulse is emitted.

To better understand how and why synchronous hardware components' clock-directed nature may impact a circuit's design, let us consider a simple example. Suppose a, b, c, and d are k-bit numbers. We would like to compute the value of r such that

$$r = (a + b) \times (c + d).$$

This operation can be mapped to elementary arithmetic hardware components, as shown in part (a) of Figure 2.3. There are two things to note about this first circuit. First, the circuit's inputs and outputs are stored in hardware registers. These registers are made up of flip-flops, as discussed in Section 2.1.2.

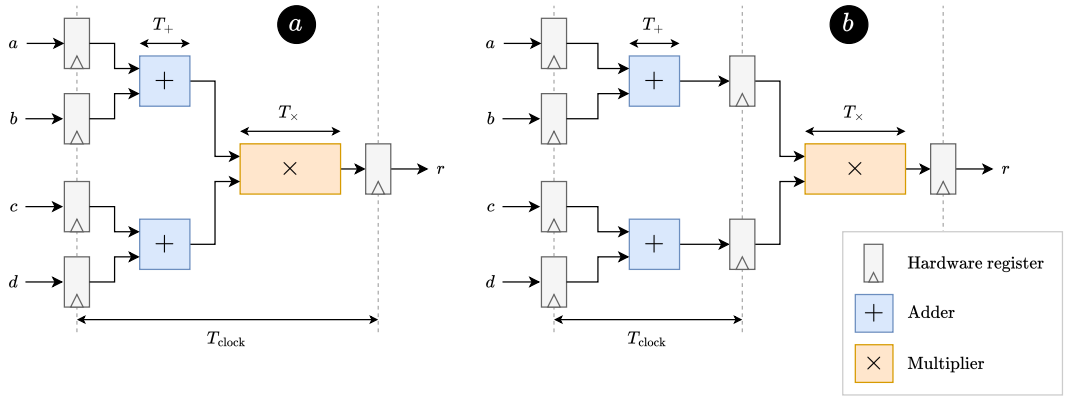


Figure 2.3 – Two implementations of $r = (a + b) \times (c + d)$ in a synchronous digital circuit. The two adders have a propagation delay of T_+, while the multiplier has a propagation delay of $T_\times = 2T_+$. Hardware registers are grouped in *register stages*, which are denoted by the dotted vertical lines.

They hold a value for the duration of a single clock cycle and are updated with each clock pulse. Second, since hardware components are physical devices, they are subject to the elementary rules of physics. In particular, the electric signals traversing each component's transistors have an inherent *propagation delay*. This phenomenon leads to each operator in Figure 2.3 having a non-zero propagation delay. The propagation delay of an operator (T_+ for adders and $T_\times$ for the multiplier in our example) corresponds to the time it takes for the operator's output to be computed from its inputs. We assume that $T_\times = 2T_+$ for the sake of this example.

Precise delay estimates. In some cases, hardware designers need to take the propagation delay on wires between components of a circuit, as well as the internal propagation delay in each logic gate, into account. Precisely timing and optimizing circuits at such a low level is out of the scope of this work. The reader may refer to existing literature for more details on this subject matter [94].

Combining the timing constraints of all the components in Figure 2.3 ⓐ, assuming that the propagation delay of signals on wires is negligible, we can deduce the minimal clock period for our first circuit:

$$T_{\text{clock}} = T_+ + T_\times = 3T_+.$$

Digital circuits are often characterized by their maximal *execution frequency*. The latter corresponds to the maximum achievable frequency of the clock signal that drives the components of the circuit and is given by

$$f_{\text{max}} = \frac{1}{T_{\text{clock}}}.$$

As we can see from our first example, the maximum achievable frequency is limited by the physical propagation delay of signals between two consecutive *register stages*. This propagation delay depends on the complexity of the combinational logic in each component. The path with the highest propagation delay limits the execution speed of the entire circuit. It is called the *critical path* of the circuit.

In order to increase the maximum execution frequency of a hardware design, we need to break up its critical path. We can insert additional registers in the circuit to reduce the propagation delay between register stages. This optimization is known as *pipelining*. An example of such a transformation is shown in part ⓑ of Figure 2.3. By adding a register at the output of each adder in our example circuit, we break the original critical path into two parts, and we now get

$$T_{\text{clock}} = \max(T_+, T_\times) = T_\times < 3T_+.$$

While we increased the execution frequency of the circuit, we need to be aware that we also increased its overall *latency*. The latency of the circuit corresponds to the number of clock cycles required to

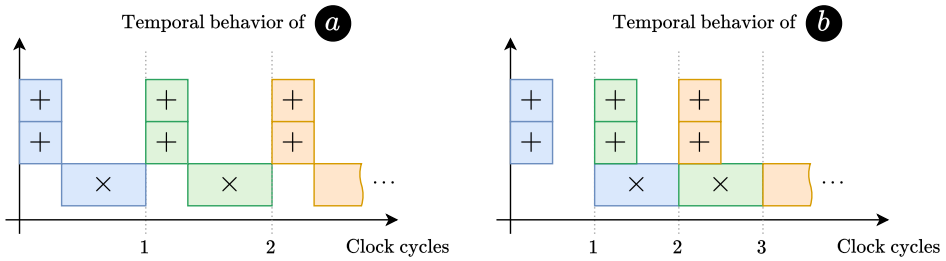


Figure 2.4 – Temporal behavior of the circuits in Figure 2.3. Each computation of r uses a different color. The execution of two successive computations of r can be partially overlapped in circuit b, which, combined with the higher execution frequency of the circuit, leads to an increased throughput.

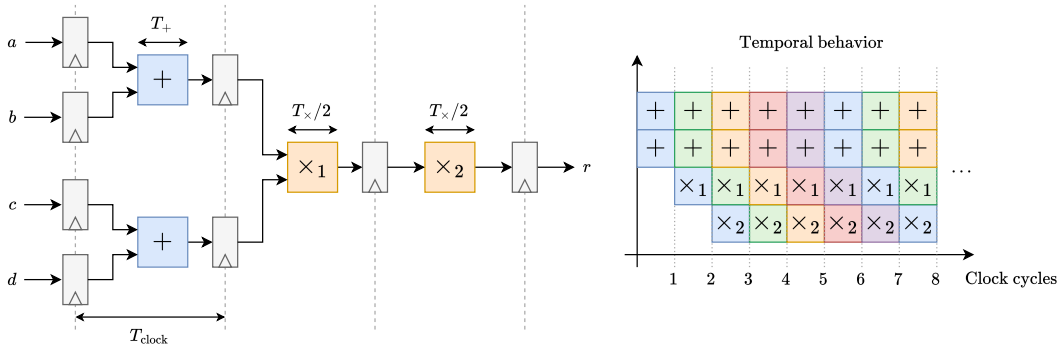


Figure 2.5 – Fully-pipelined circuit to compute $r = (a + b) \times (c + d)$. The latency of the circuit is now three cycles, but in a steady execution state, a new value of r is available every T_{clock}, with the latter equal to a third of the original clock period of the circuit in Figure 2.3 a.

obtain the result of the computation performed by the circuit. In part a of Figure 2.3, the latency is equal to one cycle, but in part b, it is equal to two cycles.

The latency and maximal execution frequency of a circuit matter when we consider how the hardware is used for computational tasks. A continuous stream of input data arrives at a, b, c, and d, and a continuous stream of output data comes out of r. The temporal behavior of the circuits in Figure 2.3 is illustrated in Figure 2.4. Since registers are updated at every clock cycle, the signal propagation in circuit b leads to a partial overlap of successive computations of r. Since the execution frequency of the circuit is also higher than its counterpart a, the circuit's overall *throughput* is increased: more values can be computed in a smaller time frame. *Throughput* and *performance* are often used interchangeably when discussing a given circuit.

While the circuit in Figure 2.3 b is faster at computing values than the circuit in Figure 2.3 a, its two adders are idle during half of the execution. This situation often arises in hardware design, where a long operator constrains the clock frequency of the entire design, leading to an under-utilization of some resources. To remedy this inefficiency, we can push pipelining even further by splitting up the long multiplication operator in two *stages*, which, when combined, compute the same result as the original multiplier. Figure 2.5 illustrates the result of splitting the multiplier into two stages to compute r. The resulting circuit has an execution frequency of

$$f_{\max} = \frac{1}{T_{\text{clock}}} = \frac{1}{T_+},$$

which is three times as high as our original execution frequency. Consequently, the throughput of the circuit is three times as high as our original implementation.

The clock frequency analysis and pipelining techniques discussed in this section form the backbone of static timing analysis and optimization in digital circuit design. The ability to quickly evaluate the

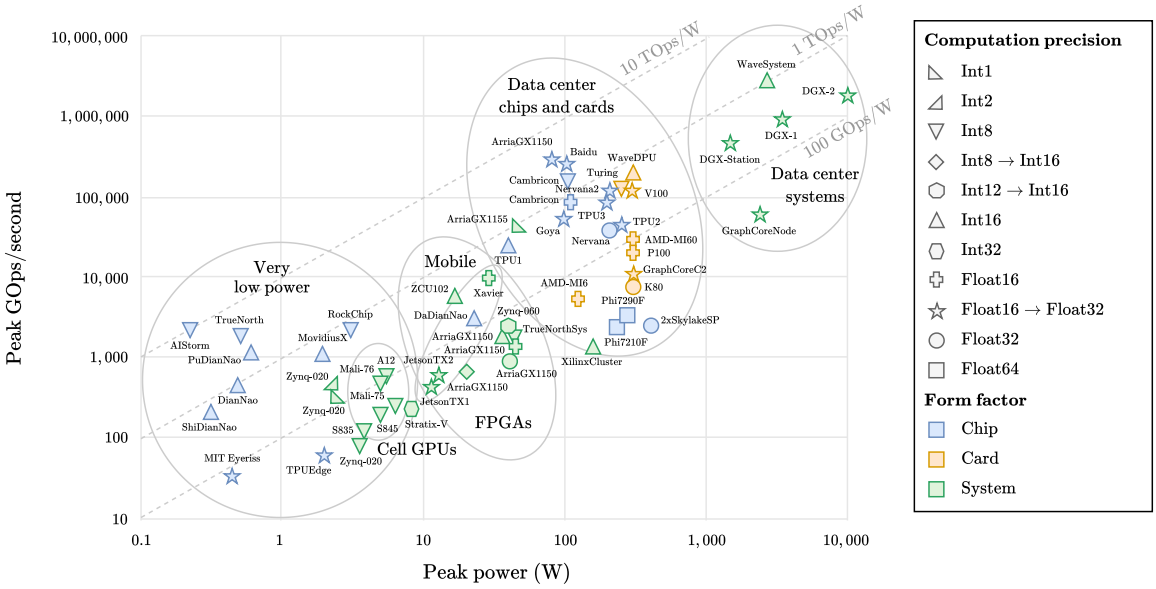


Figure 2.6 – Performance w.r.t. power scatter plot of publicly announced AI accelerators (May 2019). Both scales are logarithmic. Adapted from [316].

approximate performance of a circuit plays a critical role in the design and implementation of modern hardware, which often needs to match specific performance goals. Similar analysis techniques can be used to estimate the energy consumption of a circuit, which can be helpful in embedded computing [246, 174] and highly-constrained execution environments [317]. Scaling these techniques to large electronic designs is crucial for modern computing and requires automation. Electronic Design Automation is discussed in more detail in Chapter 3.

2.2 Hardware Accelerators

Hardware accelerators are specialized pieces of hardware that are designed to accelerate a given computing task. Such accelerators are deployed in high-bandwidth-requirement environments, where they process data for various purposes. Signal processing, image processing, cryptography, and network packet analysis are classical applications of hardware accelerators. Their high efficiency and domain-specific programmability often characterize the latter: accelerators are designed for a given task and should execute it as efficiently as possible. This characteristic can be contrasted with the general-purpose nature of instruction set processors, described in Section 2.3.

Accelerators have quickly gained traction in recent years, with the growth of new application domains such as machine learning, big data analytics, and high-frequency trading. Such applications require large amounts of computing power, with often comparatively high energy and operating costs [154]. Hardware accelerators can be a solution to increase such systems' performance while decreasing their overall energy consumption [2].

Today's most common form of hardware accelerator is the *Graphics Processing Unit* (GPU). While GPUs were originally designed to accelerate a limited number of graphics-related tasks such as *rasterization* or video decoding, they have evolved to become more general-purpose computing platforms [90]. New programming languages such as CUDA [**noauthor˙cuda˙nodate**] and OpenCL [**noauthor˙opencl˙2013**] have led to rapid growth in GPGPU (General-Purpose GPU) application development, mainly in the domains of simulation and scientific computing, all the way to machine learning.

While GPUs are still the primary computing platform for machine learning applications today [154, 180], the focus is slowly shifting to more specialized accelerators, such as *systolic arrays* [304, 190].

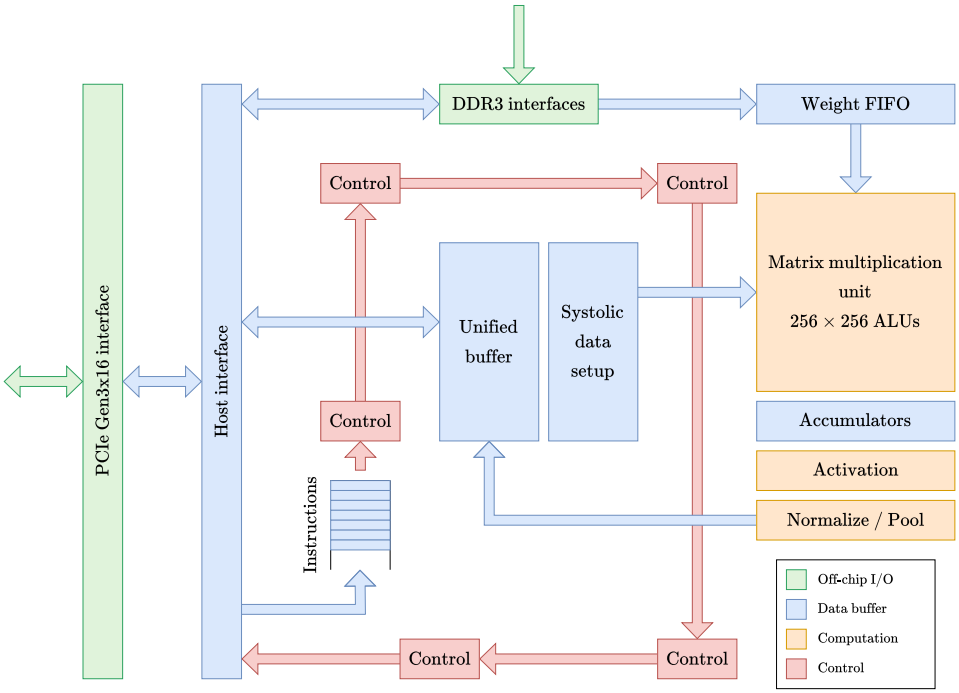


Figure 2.7 – Google's TPU block diagram. The TPU is built around a large systolic array to accelerate the matrix multiplication operations at the core of state-of-the-art deep neural networks. It is designed as a PCIe-compatible card that can be easily integrated into Google's server infrastructure. Adapted from [190] and [154].

These accelerators offer less programmable capabilities than GPUs but often significantly increase performance per Watt for accelerated applications compared to their GPU counterpart. Figure 2.6 shows an overview of the machine learning accelerator landscape, according to a survey conducted in May 2019 [316]. We observe that hardware accelerators can span a vast range of energy requirements, going from a few tens of milliwatts for chips such as the TrueNorth [6] to several kilowatts with GraphCore Intelligent Processing Unit (IPU) nodes [175]. Additionally, the AI accelerators listed in Figure 2.6 provide various computational performance characteristics. Very low-power accelerators provide 10 to 1,000 GOps/second, with the peak performers even outrunning classical hardware architectures on specific tasks [6]. Mobile chips and FPGAs (see Section 2.4.2) lie in the middle of the spectrum, with most chips being well-suited for consumer electronics. Data center chips and acceleration cards have seen rapid adoption in the past decade [154], with more and more computational workloads directed towards specialized hardware accelerators.

A prominent example of a special-purpose hardware accelerator is Google's Tensor Processing Unit (TPU) [190, 189], whose block diagram is given in Figure 2.7. The TPU is built around a large matrix multiplication unit, which contains 256 × 256 Arithmetic and Logic Units (ALU), each capable of performing an 8-bit multiply-accumulate (MAC) operation [154]. These operations are at the core of most modern deep neural networks (DNN) and form a core component of Google's infrastructure. The TPU is an interesting case study showcasing the power of hardware acceleration for critical industrial applications. Systolic arrays have long been studied as potential hardware acceleration structures for machine learning models [209, 304, 305, 210], but the TPU was the first widely deployed accelerator based on this architecture. While it is aimed exclusively at matrix multiplication acceleration, the TPU still exposes a programmable interface based on a CISC instruction set (see Section 2.3). This programmability makes it a more flexible acceleration platform, supporting a wide range of machine learning models [190, 189]. The entire operation of the TPU is directed by the control blocks depicted in Figure 2.7, which consume instructions provided by the host platform via the PCIe interface. These

control blocks orchestrate data movement between multiple buffers, optimized to feed data as fast as possible to the systolic array. Google reports that the first iteration of their TPUs provides 83× better performance per Watt than its contemporary CPU counterparts and 2.9× better performance per Watt than contemporary GPUs [328].

The current trend in hardware specialization is not new, but it has seen massive growth in recent years [287]. This growth can largely be attributed to the limits of multicore architectures, for which increasing the number of cores has diminishing returns (this is called *Amdahl's Law*). Furthermore, thermal dissipation issues in modern chips have renderer many multicore architectures to be unable to use all of their cores at full speed simultaneously, leading to so-called *dark silicon* areas on the hardware chip [113, 307].

Amdahl's Law and Moore's Law. Gene Amdahl first informally discussed what is known today as Amdahl's Law in his 1967 paper *Validity of the single processor approach to achieving large scale computing capabilities* [11]. This law can be stated as follows. Consider some computational task with a fraction of its total work $0 < s \leqslant 1$ inherently serial, while the remaining $1 - s$ fraction is p-fold parallel. Then

$$T(p) = sT(1)+(1-s)\frac{T(1)}{p}, \quad S(p) = \frac{T(1)}{T(p)} = \left(s + \frac{1-s}{p}\right)^{-1}, \quad \text{and } E(p) = \frac{T(1)}{pT(p)} = \frac{1}{sp+(1-s)},$$

where $T(n)$ is the execution time using n processors, $S(n)$ is the speedup gained by using n processors, and $E(n)$ is the resulting efficiency (*i.e.*, the ratio of the cost of running the task on a single processor vs. n processors). Consequently,

$$\lim_{p\to\infty} S(p) = \frac{1}{s}, \quad \text{and } \lim_{p\to\infty} E(p) = 0.$$

The conclusion of Amdahl's paper was quite pessimistic and led to more focus being put on advancing single-core processor performance. Another famous law coincidentally influenced the latter advancements, *Moore's Law*, which states that the number of components in integrated circuits would roughly double each year. For an in-depth discussion of the relationship between Amdahl's and Moore's Law, see [153].

Hardware accelerators shine in data-intensive applications with a lot of regularity and inherent parallelism, making them viable candidates for many data processing needs in the industry. Such accelerators are used for prototyping and small-scale deployment on Field-Programmable Gate Arrays but also for large-scale data processing acceleration with custom Application-Specific Integrated Circuits (see Section 2.4 for more details on implementation platforms). However, some workloads require programmable capabilities and flexibility levels that cannot always be achieved with specialized hardware accelerators. In such cases, instruction set processors may be the better choice.

2.3 Instruction Set Processors

Some computing tasks require more general-purpose capabilities than what hardware accelerators have to offer. A prime example is desktop and cloud computing, where users want to execute arbitrary workloads. Instruction Set Processors (ISPs, or *processors*) are programmable hardware devices that can execute arbitrary programs. Such programs are written in high-level programming languages and then compiled into machine language by a *compiler*. The flexibility and effectiveness of processors have brought them to the core of the computing revolution, where they benefited from advancements in hardware technology and computer architecture [284]. This section presents ISPs from two different angles: the Instruction Set Architecture (Section 2.3.1), which defines the programmable interface of a family of processors, and the micro-architecture (Section 2.3.2), which defines the hardware implementation details of a given processor.

Table 2.1 – Common modern processor architectures, ordered by release year.

ISA	Designer	Release year
x86	Intel Corporation	1978
ARM	ARM Ltd.	1985
Power ISA	International Business Machines Corporation (IBM)	1992
AMD64 (x86-64)	Advanced Micro Devices, Inc. (AMD)	1999
RISC-V	University of California, Berkeley	2011

2.3.1 Instruction Set Architecture

The Instruction Set Architecture (ISA) of a given processor describes the operations that said processor is capable of executing. The ISA is the primary interface between software and the underlying hardware on which it executes. While most modern ISAs are proprietary, the emergence of RISC-V, an open standard ISA specification, has rapidly grown new ISA and processor developments, from academic projects to industrial products alike [70].

The Hardware/Software Interface

The low-level programmable capabilities of processors are exposed through machine language. Machine language follows a specification the processor vendor defines as an Instruction Set Architecture (ISA). The ISA defines the instructions recognized by a processor, as well as their encoding, their operands, and the many different side effects that an instruction can have on the processor's *architectural state* (*i.e.*, its user-observable state). The ISA forms the basic description of a processor's *architecture*. Notable examples of modern consumer-grade processor architectures are listed in Table 2.1. RISC-V is the most recent of those architectures. It is an open architecture designed at UC Berkeley that boasts rapid adoption and growth. The next section gives more details about RISC-V and its potential for research and industrial applications.

Instruction Set Architectures have historically been divided into two main groups: RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer). The former designates ISAs with a reduced number of simple instructions: arithmetic, logical operations, load from memory, and store to memory. The latter describes ISAs with more complex instructions, where some may involve a combination of arithmetic, logical, and memory operations with complex addressing modes. While this distinction may still be valuable in evaluating different architectures, it has become less relevant.

Blurring the line between RISC and CISC. x86 (and its successor, AMD64) is the canonical example of a CISC instruction set, but modern x86 processors do not execute the CISC instruction directly; instead, they decode instructions into micro-operations (μ-ops, which can be thought of as elementary instructions) before executing them. The μ-ops executed by x86 processors resemble a RISC instruction set. Conversely, some RISC instruction sets, such as ARM, have long incorporated complex memory instructions, blurring the line between RISC and CISC ISAs.

Programmers can interact with machine language using *assembly language*. Assembly language provides a human-readable form of the instructions for a given processor architecture. Figure 2.8 shows the RISC-V assembly language version of a function that searches for the maximum value in an array of unsigned 32-bit integers. This example uses instructions that are common in many ISAs: conditional and unconditional branches—`ble`, `beq`, `bgeu` (respectively branch if less than or equal, equal, and unsigned greater than or equal), `j` (jump), and `ret` (return)—arithmetic instructions—`slli` (shift left logical by an immediate), `add` and `addi` (add immediate)—and data movement instructions—`mv` (move), `li` (load immediate), and `lw` (load word). Note that some of the instructions used in Figure 2.8, such as `li`, are *pseudo-instructions*, *i.e.* they are syntactical shortcuts for a short sequence of equivalent instructions. The translation from human-readable assembly instructions to machine code follows the

```
1  ; unsigned int max(unsigned int *x, int n)
2  ;   Parameters are stored in registers a0 and a1.
3  ;   The return value is stored in register a0.
4  max:
5    ble     a1, zero, .no_elements ; Return immediately if n <= 0.
6    mv      a5, a0
7    slli    a1, a1, 2              ; Compute the end iterator,
8    add     a3, a0, a1             ;   x + n * sizeof(int).
9    li      a0, 0                  ; Initialize the result.
10   j       .loop_body
11 .condition:
12   addi    a5, a5, 4              ; Advance the loop iterator.
13   beq     a5, a3, .end           ; Did we reach the end of the loop?
14 .loop_body:
15   lw      a4, 0(a5)              ; Load the next element from the array.
16   bgeu    a0, a4, .condition     ; Compare it to the current result.
17   mv      a0, a4                 ; Update the result variable.
18   j       .condition
19 .end:
20   ret
21 .no_elements:
22   li      a0, 0
23   ret
```

Figure 2.8 – Annotated RISC-V assembly code example. The `max` function searches and returns the maximum value from the `n`-element array `x`.

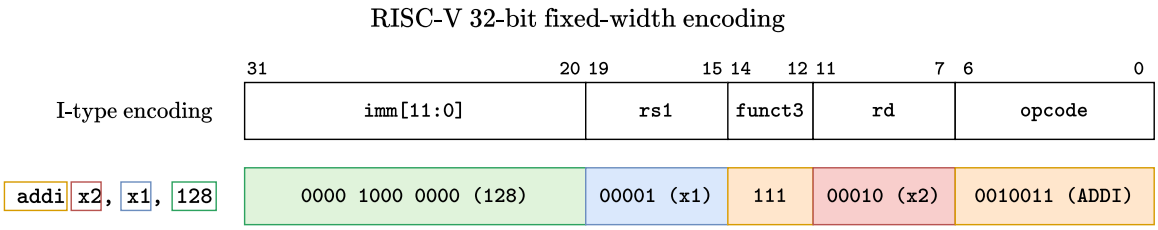


Figure 2.9 – Example RISC-V instruction encoding. The `funct3` and `opcode` fields are combined to encode the instruction's operation.

rules defined by the target platform's ISA. Some ISAs use a fixed-width encoding, where each instruction is encoded using a fixed amount of bytes, while others allow variable-length instructions. The choice of encoding is often one of trade-offs between code compactness (*i.e.*, variable-length instructions can assign a shorter encoding to frequently-used instructions) and decoding logic complexity. Figure 2.9 gives an example encoding of the `addi` instruction in the RISC-V ISA. The latter uses 32 bits for all of its instructions, which greatly simplifies decoding in hardware. However, this limitation restricts the range of values that can be directly encoded in the instruction (*e.g.*, `addi` can only use immediates of up to twelve bits, including their sign). This design contrasts with x86, which sacrifices ease of decoding for more flexibility, providing eleven different encodings for the same *add immediate* operation in 64-bit mode. More common operations are encoded using fewer bytes, and an instruction can directly encode immediates from 8 to 32 bits.

The programmer's view of an instruction set processor is strongly linked to the instruction set architecture. However, for a given architecture, many possible hardware implementations are possible.

The ISA provides an abstraction over the actual organization of the hardware and can be seen as the external API of the hardware. A processor's actual hardware implementation details form what is referred to as its *micro-architecture* (see Section 2.3.2).

RISC-V as a Bridge Between Accelerators and Processors

RISC-V is an instruction set architecture designed at UC Berkeley, which was first introduced in 2011 [283]. RISC-V distinguishes itself from most other instruction set architectures by its *open* nature. A non-profit foundation maintains the ISA and its specification, aiming to evolve the latter and ensure its long-term stability [1]. This open nature makes RISC-V a desirable target for compiler, computer architecture, and processor design research. On the other hand, many industries support the RISC-V foundation and encourage the development and adoption of RISC-V in several application domains. At the time of this writing, 372 industrial and academic partners are members of the RISC-V foundation [2].

Most modern ISAs are built incrementally: new instructions are added to the instruction set, but processors based on the new ISA iterations must support all of the instructions already supported by previous generations. This design ensures *backward binary-compatibility* but can quickly lead to ISA bloat [283]. RISC-V was designed to avoid this design path by being specified as a *modular* ISA: it is built around a core ISA, namely `RV32I` (32-bit integer instructions), which is capable of running a minimal software stack. This core ISA can be supplemented by optional standard extensions that hardware designers can include in their processor cores, depending on the needs of their clients. In addition to standard extensions, RISC-V also supports custom extensions, with parts of the instruction encoding space explicitly reserved for such usage [3]. Suppose a piece of software uses an instruction from an unsupported standard extension. In that case, the hardware generates a *trap*, and the execution of the missing instruction can be handled gracefully by software emulation.

RISC-V ISA naming. The standard naming scheme for RISC-V instruction sets is to append the width of the architectural word (*e.g.*, 32 for 32-bit architectures) to the *RV* prefix before appending the list of supported extensions. For example, the minimum set of extensions to run a typical GNU/Linux distribution is `RV32IMAFDZicsr_Zifencei`: core (I), multiply (M), atomic (A), single-precision (F) and double-precision (D) floating-point, control and status registers (Zicsr), and instruction-fetch fences (Zifencei).

The extensible nature of the RISC-V ISA makes it a perfect candidate to design processors with domain-specific characteristics, effectively bridging the gap between general-purpose instruction set processors and hardware accelerators. A recent survey by Cui, Li, and Wei [83] lists more than 20 active research projects on RISC-V extensions for domain-specific applications. These extensions span several application domains, including vector computation [352], machine learning [176, 383, 74], state-of-the-art cryptography [125, 268, 211], high-performance computing [386, 385], wireless communications [14, 364], digital signal processing [126, 12], and security [252, 151]. Far from being an extensive survey of all available custom RISC-V instruction set extensions, Cui, Li, and Wei's work [83] illustrates the thriving ecosystem that blossomed following the release of RISC-V in academic research. Additionally, RISC-V is still under active development, with new standard extensions being proposed and ratified regularly. At the time of this writing, the RISC-V unprivileged ISA specification (version 20240411) lists 42 ratified standard extensions [4], along with a ratified specification for the core 32-bit and 64-bit ISAs. A 128-bit version of the core ISA, `RV128I`, is currently in its draft version. The recently ratified V (Vector) extension [5] provides vector-processing capabilities in the form of variable-length vector operations. This approach contrasts with the common implementation of vector instructions in modern ISAs, which focus on a SIMD (Single Instruction Multiple Data) programming model with

1. https://riscv.org/
2. https://riscv.org/members/ (Accessed 2024-07-23).
3. https://github.com/riscv/riscv-isa-manual
4. https://riscv.org/technical/specifications/
5. https://github.com/riscv/riscv-v-spec

fixed-width vector registers and corresponding instructions. While controversial, the variable length vectors proposed by the RISC-V standard specification (and by the similar ARM SVE extension) are an example of the innovation in hardware acceleration that can be driven by new ISA design.

RISC-V has also been directly integrated into the design of hardware accelerators. The Vortex GPU micro-architecture [362] is a prime example of such an accelerator. Vortex combines a custom ISA extension with a micro-architecture design tailored to graphics computing acceleration. By supplementing it with a custom compilation framework, the authors of the Vortex paper developed Skybox [361]. This 3D graphics framework leverages this RISC-V-based GPU to accelerate graphics applications written using modern 3D APIs such as OpenGL ES [6] and Vulkan [7].

Through its open design philosophy, the RISC-V architecture provides new opportunities for researchers to bridge the gap between hardware accelerators and general-purpose processors. Its customization points encourage tight integration between hosts and acceleration hardware, which is crucial to providing more efficient and sustainable computing platforms in the near future [41].

2.3.2 Micro-architecture

A single instruction set specification can be used to design a wide variety of ISPs: the ISA serves as a high-level specification of the hardware operations, but the actual implementation can vary depending on usage requirements. A concrete implementation of a given architecture is called a *micro-architecture*. Contrary to the processor's architecture, the micro-architecture is not visible to the programmer and is mostly an implementation detail from the software perspective. The micro-architectural design landscape spans from very low-power devices based on low-energy *micro-coded* micro-architectures to high-performance *Out-of-Order* (OoO) processors.

Micro-architectural information leakage. While the micro-architecture is not directly exposed to programmers, some micro-architectural states may leak into an architecturally observable state. This information leakage is commonly called a *side channel*. Side channels have been a hot topic of computer security research in the past decade, with side-channel-based attacks [347] such as Spectre [199] and Meltdown [229] making it to the forefront of international news. The reader may find more information on side-channel attacks and possible countermeasures in [238, 345, 236].

A typical instruction set processor comprises several elementary building blocks, which are common among most micro-architectural implementations:

- *Registers* are used to store computation results and can be used as operands for instructions. They are grouped in a *register file*, which can hold a few dozen registers on small processors and up to several hundred registers in modern high-performance processors [154].
- *Functional units* perform the computation tasks required by each instruction the processor executes. Typical functional units include Arithmetic and Logic Units (ALU), Address Generation Units (AGU), and floating-point units (FPU). Along with registers, functional units are the backbone of a processor's datapath.
- The *control unit* is a Finite-State Machine (FSM) that orchestrates data movement between registers and functional units and governs the execution sequence of instructions inside the processor.

The simplest form of processor micro-architecture can be found in micro-coded processors, which can be divided into *single-cycle* and *multi-cycle* designs [152]. The former executes an entire instruction in a single cycle, making it a straightforward processor design. However, the clock frequency of such a design is limited by the slowest instruction that needs to be executed. A simple upgrade to the single-cycle micro-architecture is the multi-cycle design, which divides instructions into several execution cycles that are processed sequentially. This micro-architecture is often employed in very low-power settings, where performance is not critical, but energy savings are. While micro-coded processors offer low performance levels compared to other micro-architectures, they bring comparatively low power

6. https://www.khronos.org/opengles/
7. https://vulkan.org/

requirements, making them especially suitable for low-power sensors. Additionally, by splitting potentially complex instructions into multiple stages, multi-cycle micro-architectures can leverage hardware reuse, re-purposing functional units for different purposes during each execution cycle. Advances in computer architecture have nonetheless rendered micro-coded micro-architectures somewhat obsolete, with ultra-low power processors based on *pipelined* designs emerging in the past decades [106].

Pipelined micro-architectures apply the pipelining techniques discussed in Section 2.1.3 to the low-performance multi-cycle processor design. By unrolling the execution of a single instruction into multiple successive stages, pipelined processors can overlap the execution of multiple instructions. Consequently, such designs have higher throughput, achieving up to one completed instruction execution per clock cycle. Pipelined micro-architectures require additional logic to handle potential dependencies between instructions that are executed simultaneously, but this overhead is largely accounted for by the increase in execution performance [154, 152].

Abstract Pipelined Processor Execution Model

Most modern processors are designed around a standard set of high-level abstractions that form the *pipelined* execution model. The latter separates a typical processor core into several *pipeline stages*, which are executed in sequence to produce the results of the core's input instruction stream. Figure 2.10 ❶ illustrates the classic 5-stage processor pipeline model, which forms the basis of most computer architecture textbooks and courses [154, 284], for the RISC-V ISA. The control unit ⓐ is responsible for orchestrating instruction execution. The register file ⓑ has two read ports and a write port. The pipeline is divided into five distinct stages:

1. *Instruction Fetch*, where the processor reads an instruction from main memory and loads it into an internal register.
2. *Instruction Decode*, where the bits of the fetched instruction are interpreted as the instructions *opcode* and operands.
3. *Execute*, where the instruction is executed, depending on the decoded opcode and operands from the previous stage.
4. *Memory*, where loads from and stores to memory are executed. Note that the EX stage may not compute anything for load and store instructions.
5. *Write-back*, where results from the instructions' execution are written back to the processor's register file.

The fetch and decode stages form the processor's *frontend*. The frontend determines what instructions must be executed and dispatches them to the following stages. The execute and memory stages are part of the *backend*, which performs the execution. Part ❷ of Figure 2.10 shows a typical execution trace for such a processor. The horizontal axis represents time (in clock cycles), while the vertical axis represents successive instructions in program order. A new instruction is loaded into the pipeline at each cycle and starts execution. The parts of the pipelined that are relevant to the execution of the current instructions are highlighted. Instructions that a pipelined processor is currently executing are called *in-flight* instructions.

The pipelined execution model requires handling of potential *hazards* that can occur between instructions. We distinguish three types of pipeline hazards:

1. *Data hazards* occur when two overlapped instructions operate on the same data in different pipeline stages. Similarly to multi-threaded programs where two threads access the same shared variable, data hazards can lead to race conditions, which must be avoided to guarantee correct program execution. There are three types of data hazards: *Read-after-Write* (RAW), where an instruction needs to read the shared value after it has been updated; *Write-after-Read* (WAR), where an instruction should not update a value until another instruction has read it, and *Write-after-Write* (WAW), where two instructions need to write their results to the register file in program order.
2. *Structural hazards* occur when two overlapped instructions need access to the same hardware resource. When such a hazard occurs, the execution of one of the overlapped instructions needs

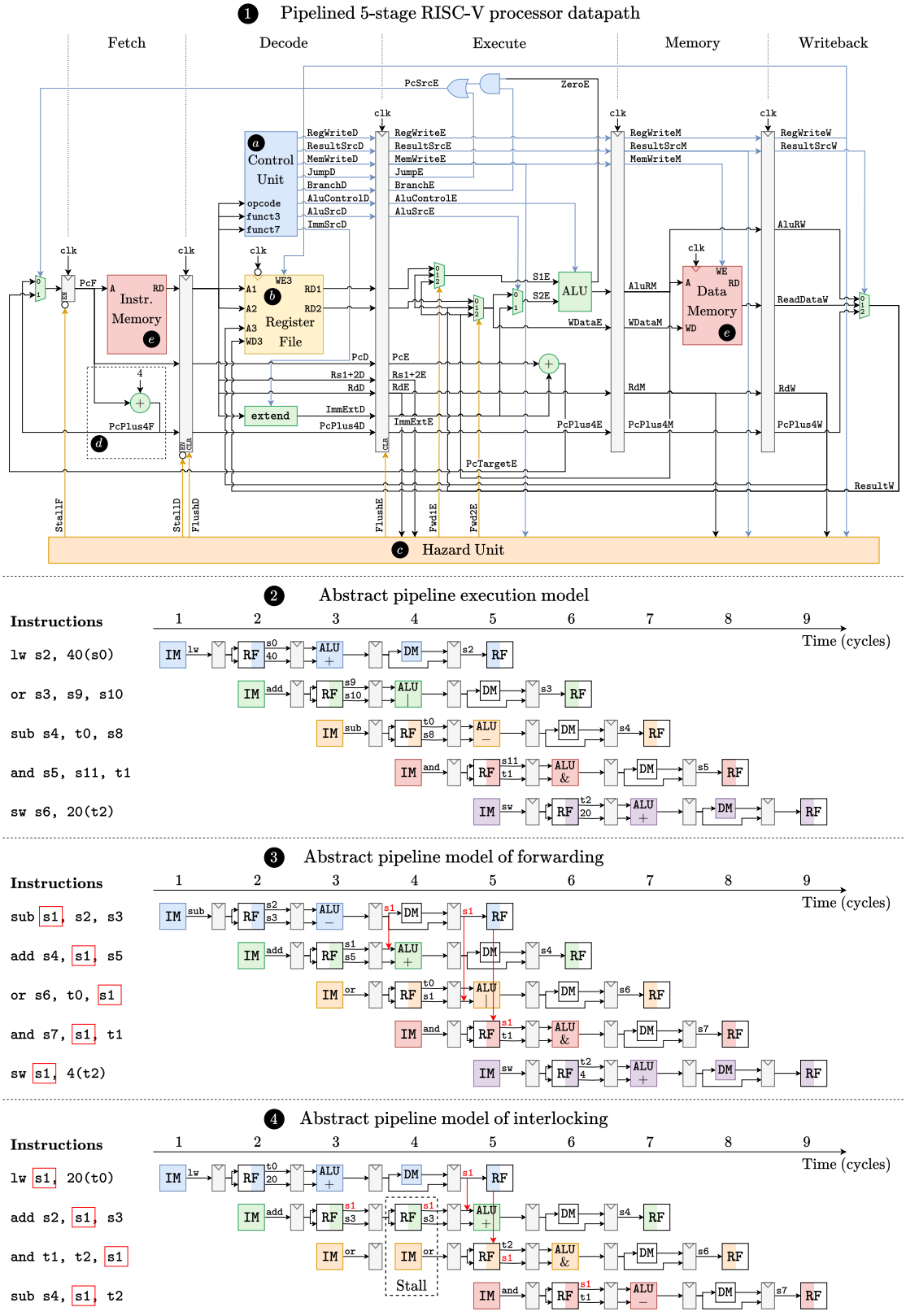


Figure 2.10 – Pipelined RISC-V datapath and abstract execution model. Datapath and trace visualization ideas adapted from [152].

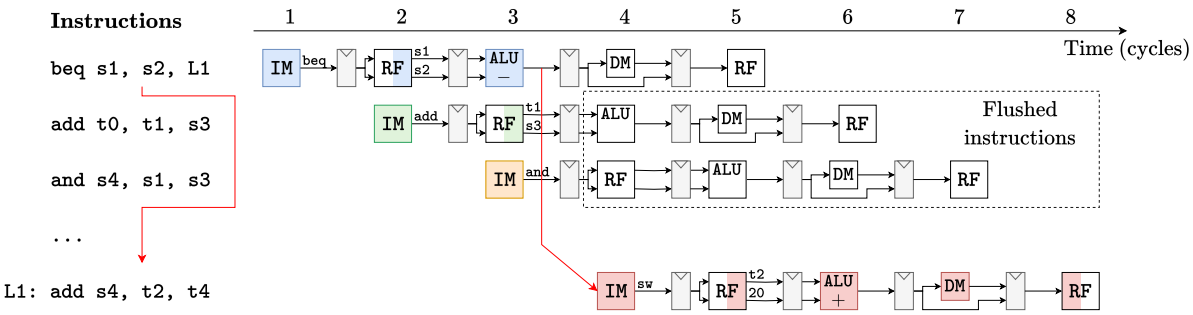


Figure 2.11 – Abstract pipeline model of flushing on branch misprediction. Flushed instructions are canceled after the branch is resolved in the execute stage. Note that branches may be resolved earlier, depending on the pipeline's design. Adapted from [152].

to be halted, waiting for the other instruction to finish using the shared resource before it can be resumed.

3. *Control hazards* occur when currently in-flight instructions need to be discarded based on the result of a previously executed instruction.

The issue of minimizing pipeline hazards while efficiently handling the ones that can occur is fundamental to efficient pipelined processor design. Data hazards can be mitigated using *forwarding* and *interlocking*, which are illustrated at the bottom of Figure 2.10. Forwarding, depicted in Part ❸, consists of linking multiple pipeline stages together to bypass some stages. In the example execution, forwarding values between the execute, memory, and write-back stages allows for a fully pipelined execution, even though data dependencies exist between successive instructions. The forwarding logic is controlled by the *hazard unit* ⓒ shown in Part ❶ of Figure 2.10. Interlocking is another technique that can be used to mitigate data hazards in case forwarding is not sufficient. In the example of Part ❹, the value of `s1` loaded from memory cannot be forwarded to its usage point in the second instruction. Thus, instead of waiting for the full execution of the `lw` instruction, the execution of some instructions is *stalled*, waiting for `s1` to be available at a forwarding point. The area marked as *Stall* in the execution trace is referred to as a *pipeline bubble*. Its value is then forwarded to the execute stage for the second instruction. Note that interlocking delays the start of the fourth instruction since only one instruction may use a pipeline stage at a given time. Consequently, structural hazards can also be avoided using interlocking. Out-of-order execution can also be used to mitigate both data and structural pipeline hazards. Control hazards, on the other hand, require *branch prediction*. These techniques are discussed in the following sections.

Predicting the Flow of Execution

The pipelined execution model depicted in Figure 2.10 requires a new instruction to start every cycle to use all available hardware resources efficiently. While this property can easily be maintained for sequential instruction streams, instructions that impact the program's control flow make this task more arduous. For example, a conditional branch may jump to two completely different locations in the program, depending on the value of its condition code. Consequently, while the branch instruction is in-flight, the next instruction to be executed is unknown. Waiting for the completion of every branch instruction in the input program is not a viable strategy for high-performance processor execution. Additionally, we note that the exact type of instruction is only known at the end of the decode stage. Consequently, a processor waiting for branches to finish execution could not pipeline its frontend.

Branch prediction [284, 154] is a hardware mechanism that tries to *predict* the flow of execution of an instruction stream in order to allow for efficient pipelined execution. A specialized hardware unit called the *branch predictor* produces predicted values of the program counter (PC) to be used by the instruction fetch stage to retrieve instructions from memory. A simple branch predictor implementation may predict

that no branches are ever taken and always instructs the pipeline to fetch and start executing the next instruction in the input stream. Such a predictor is instantiated in Figure 2.10 ❶, and is highlighted in the area marked ⓓ. Since branch prediction is a probabilistic process, some predictions may be incorrect. In such cases, a dedicated *rollback* mechanism needs to cancel the execution of wrongly predicted in-flight instructions by *flushing* the pipeline and reset it to a valid state before starting to execute the correct sequence of instructions. Figure 2.11 illustrates flushing in a simple example.

Branch prediction accuracy. State-of-the-art branch predictors achieve impressive levels of accuracy, often upwards of 98% of correct predictions. They are mostly based on perceptrons [178, 179, 177], or multiple branch histories [249, 334, 254]. There are even international competitions dedicated exclusively to determining which branch predictor is the best on a fixed workload! [a].

a. https://jilp.org/cbp2016/

The predictive machinery underlying branch prediction can be generalized to other areas of the micro-architecture. Hardware that includes generalized prediction mechanisms is said to provide *speculative execution* capabilities. Speculation lies at the core of most instruction set processor designs. Beyond branch prediction, it is used in *prefetchers* to retrieve data from memory before it is needed by in-flight instructions [294, 315, 346, 416, 309, 116, 115, 201, 193, 208, 207, 292, 15, 170, 321, 56, 322]. Prefetchers analyze the memory access patterns of the application to determine which cache lines are likely to be needed by future instructions, both for instructions and data. Further applications of speculation in instruction set processors have been studied in the field of value prediction, with the intent of increasing instruction-level parallelism by speculating on the result of certain instructions [291, 289, 290].

Speculation in in-order processors. Speculation is often associated with high-performance processors that execute instructions out of program order. However, this section shows that even simple in-order pipelines need speculation to predict the next instruction to start executing at each cycle. This observation motivates our research on speculation in the general setting of High-Level Synthesis, which is exposed in Chapter 5.

Speculative execution has many applications in the context of instruction set processor micro-architecture design. It significantly contributes to the performance of a processor [248] and enables better resource utilization while opening up opportunities for more parallel execution. As shown in Chapter 5 and Chapter 6, speculation can also bring significant performance benefits to hardware accelerators.

Deep pipelines. The 5-stage pipelined micro-architecture used throughout this section to showcase different micro-architectural designs is the canonical example of a processor pipeline [154, 284, 152]. Modern high-performance processors often target high clock speeds, which naturally leads them to have more pipeline stages: 8 to 20 stages are quite common for today's cores. While increasing the number of pipeline stages allows for greater execution frequencies, this technique has diminishing returns. In particular, deep pipelines encounter more potential hazards and suffer from larger mispeculation penalties.

Exploiting Instruction-Level Parallelism

The instruction stream of a typical program often includes sequences of instructions that do not depend on one another. Such instructions are good candidates for parallel execution on multiple functional units inside the processor. Multiple-issue pipelines and Out-of-Order processors try to maximize such parallelism (which is called *instruction-level parallelism*) to provide the best possible execution performance.

Multiple-issue micro-architectures duplicate pipeline parts to start executing more than one instruction in parallel every cycle. We distinguish two types of multiple-issue pipelined architectures: *VLIW*

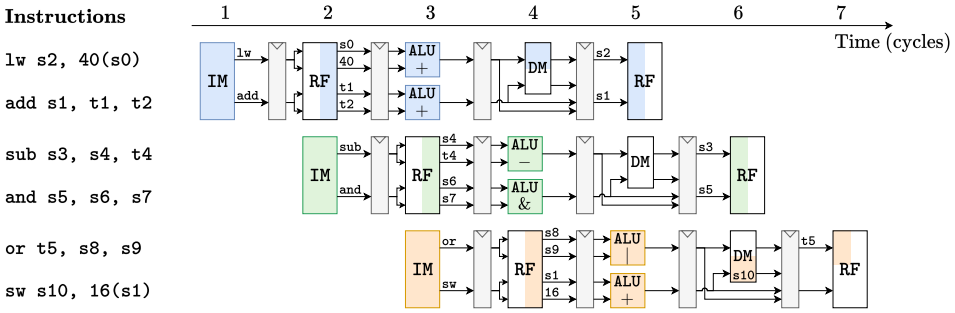


Figure 2.12 – Abstract pipeline model of a dual-issue superscalar processor. In the absence of dependencies, two instructions are started and retired at each cycle. Adapted from [152].

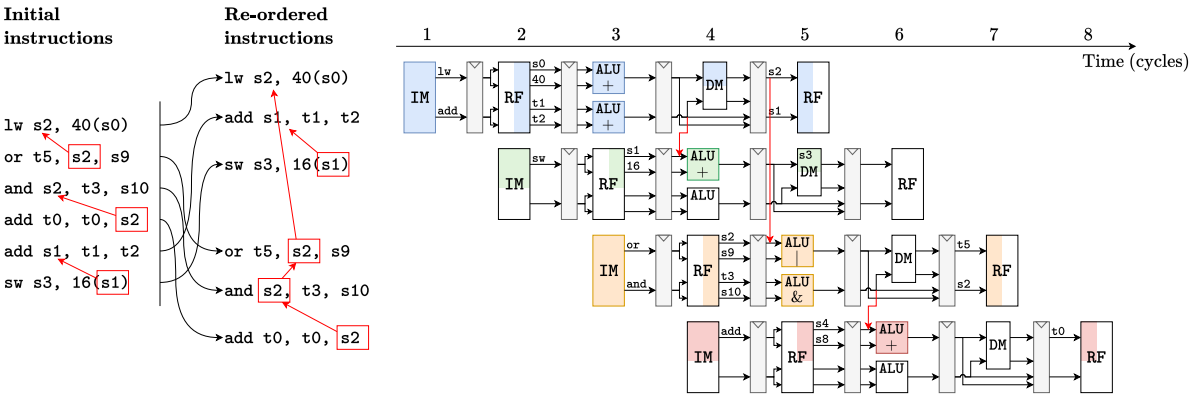


Figure 2.13 – Abstract pipeline model of an Out-of-Order processor. OoO processors can better leverage instruction-level parallelism and available forwarding opportunities by rearranging the input instruction stream on the fly. Adapted from [152].

and *superscalar* processors. VLIW (*Very Long Instruction Word*) processors are based on ISAs, where instructions are grouped into vectors and executed simultaneously. These architectures rely heavily on the compiler to schedule the execution of instruction vectors to avoid data hazards. The interested reader may find more information about VLIW processors and some historical perspectives in [123]. Superscalar processors dynamically issue multiple instructions in parallel when they detect no dependencies between successive instructions in their input stream. Figure 2.12 illustrates the execution of a dual-issue superscalar processor. The register file memory ports and the arithmetic and logic units in the execute stage are duplicated. Like VLIW processors, superscalar processors that execute instructions *in-order* require significant compiler assistance to statically schedule their instructions to minimize dependencies and subsequent pipeline hazards.

Modern high-performance processors exploit *out-of-order execution* by rearranging their instruction streams on the fly to maximize parallel execution [248, 284]. The incoming instruction stream is decoded ahead of time and analyzed by the hardware, looking for potential data dependencies (hazards). Instructions are then rearranged on the fly before being dispatched to multiple functional units. A complex mechanism is deployed to guarantee that the effects of the program's instructions still appear in order to external observers. Out-of-order execution is particularly valuable for superscalar processor designs, where the increased risk for pipeline hazard can be mitigated by rearranging the instruction stream to expose an independent series of instructions that can take advantage of the duplicated execution path and forwarding. Figure 2.13 shows the execution trace of a simple RISC-V program on a 5-stage pipelined dual-issue superscalar Out-of-Order processor. The initial program's instructions contain data dependencies that would lead an in-order superscalar design to stall after executing the first load, similarly to the single-issue execution trace of Figure 2.10 ❹. The OoO processor can reorder

these instructions to expose more parallelism and fully exploit forwarding, producing the execution trace depicted on the right of Figure 2.13.

Register renaming. Out-of-order processors often employ *register renaming* to eliminate false data dependencies that can arise between successive instructions using the same registers, which results in increased instruction-level parallelism. This technique relies on a mapping between each *logical* register (that can be referenced in an instruction) to a set of *physical* registers. The processor dynamically maps logical registers to physical ones during execution. Applying register renaming to the execution depicted in Figure 2.13, the dependency between the last `or` and `and` instructions can be eliminated by mapping both instances of `s2` to two different physical registers.

The diverse micro-architectural implementation landscape leads to an ample design space encompassing trade-offs between die area, power consumption, and performance. The inherent complexity of design space exploration makes it a good target for design automation.

Mitigating Memory Access Latency

While pipelined processor designs can bring significant execution speed improvements compared to their micro-coded counterparts, memory accesses are still notably slow when compared to the execution of other instructions. While typical instructions take 1–2 cycles to execute, accesses to main memory usually involve several hundreds of cycles, even on high-performance processors. Processor micro-architectures deploy various techniques to mitigate or hide the latency of these accesses.

The most common technique used to improve the latency of memory accesses is the use of a *cache hierarchy* [284, 154]. A *cache* is a small but fast memory buffer located near the processor core's computational units. Caches are used as transient storage for data currently used by in-flight instructions. The data stored in caches is organized in *cache lines* (whose typical size is 64 bytes for modern desktop processors). A cache maps addresses in main memory to a location in its internal buffer. Since the former is significantly larger than the latter, conflicts may arise. In such a case, the data is *evicted* from the cache and replaced by the new data, usually according to a *replacement policy*. The address to which a cache line corresponds is identified to a set of bits stored alongside the cache line, the *tag*. The latter is usually formed from the high-order bits of the memory address. An additional *valid* bit marks the cache line as containing valid data from memory. A *cache hit* happens when a memory access reads or writes data to an address already loaded into the cache. Conversely, a *cache miss* happens when data is not present in the cache and must be loaded from main memory instead. Modern processors employ sophisticated *prefetching* mechanisms and replacement policies to minimize the rate of cache misses for typical applications.

The simplest form of cache maps a memory address to a single location in its internal buffer. Such an implementation is called a *direct-mapped* cache. While direct-mapped caches are simple to implement, they can suffer from frequent evictions for applications with regular memory access patterns (*e.g.*, accesses to elements that are separated by exactly n bytes in memory, where n is the number of lines in the cache). More complex implementations, known as *associative* caches, assign addresses to multiple possible locations in the cache's buffer inside a *set*.

Multiple levels of caches of increasing size and increasing access latency are often used to improve the latency of memory accesses further. Typical desktop processors include three cache levels, some of which are inclusive of lower levels, *i.e.* they contain all the data that the lower levels in the cache hierarchy contain.

A programmer's view of memory hierarchies. Modern high-performance processors employ complex memory hierarchies and latency-hiding mechanisms that can be difficult to account for in computer programs. For more details on such issues, we strongly recommend the excellent article *What every programmer should know about memory*, by Ulrich Drepper [101].

2.3.3 Core Principles of Modern Processor Design

Processor micro-architecture design is often seen as one of the most advanced endeavors in hardware design [332]. The amount of optimizations required to achieve good performance for a wide variety of workloads makes micro-architecture design a demanding task, with significant associated costs in terms of engineering time. The core ideas on which modern processor design is built have been summarized by David Patterson in his seminal work with John Hennessy, *Computer Organization and Design* [284]. He describes them as *8 great ideas in computer architecture*. They are as follows:

1. *Design for Moore's Law.* Moore's Law is starting to reach the end of its lifetime but it has driven significant developments in computer architecture. The rapid growth in available transistors has allowed increasingly complex hardware to be integrated into processors, such as complex predictors and more advanced Out-of-Order and superscalar execution engines. The potential cost in hardware is far outweighed by the performance benefits of these optimizations, which have been made possible by Moore's Law.
2. *Use abstraction to simplify design.* Abstraction is a powerful tool that allows hardware designers to build complex micro-architectures on solid foundations. The successive levels of abstraction presented in this chapter result from decades of progress in hardware design, which have nurtured countless innovations in electronic hardware design. Chapter 3 presents another instance of this principle by describing the emergence of high abstraction levels for automated hardware design.
3. *Make the common case fast.* Along with raising the level of abstraction at which hardware designers can work, making the common case as fast as possible is an essential part of the contributions of our work. This principle is behind all advances in speculation and its multiple applications in processor design.
4. *Performance via parallelism.* Parallel execution of operations is paramount to achieving the best possible performance. Superscalar and Out-of-Order micro-architectures perfectly illustrate that striving to achieve maximum parallelism can bring unparalleled (pun intended) performance benefits.
5. *Performance via pipelining.* Virtually all modern processors are built on the abstract pipelined execution model presented in the previous sections. Pipelining enables efficient use of available hardware resources while improving performance when hazards are properly mitigated.
6. *Performance via prediction.* Branch predictors are essential to pipelined processor design. Other types of predictors enable numerous optimization opportunities in micro-architectures. Our work revolves around using speculation, which lies at the core of predictive hardware capabilities.
7. *Hierarchy of memories.* Memory has become cheap to produce and provides fast access times. However, the improvements to processor performance have far outpaced the decrease in memory access latency. Memory hierarchies built on top of caches allow modern processors to provide fast access to frequently used data, especially when compared to speculative components such as prefetchers. However, combining speculation and stateful components such as memory brings forward several challenges.
8. *Dependability via redundancy.* Physical devices such as processors can sometimes fail. In some scenarios, hardware failures can have dramatic consequences. Computers, therefore, need to be dependable, a property that is achieved by introducing redundancy.

Our contributions are closely tied to most of these micro-architecture design principles. Chapter 5 and the following chapters focus on our work to automate the task of making the common case fast in a productive high-level hardware design environment. We combine speculation with pipelining to improve the performance of hardware accelerators and instruction set processors. Chapter 6 describes our approach to automatically handling memory accesses in a high-level speculative hardware design context. Finally, we combine the ideas of Chapter 5 and Chapter 6 in Chapter 7, where we automate the synthesis of instruction set processors from a high-level description of their behavior.

Table 2.2 – Top 10 leading semiconductor foundries by market share. A large majority of the market is occupied by Taiwan Semiconductor (TSMC). The data is based on a 2024 TrendForce report [72].

Ranking	Company	Market share (4Q23)
1	Taiwan Semiconductor (TSMC)	61.2%
2	Samsung	11.3%
3	GlobalFoundries	5.8%
4	United Microelectronics Corporation (UMC)	5.4%
5	Semiconductor Manufacturing International Corp. (SMIC)	5.2%
6	Hua Hong Semiconductor	2.0%
7	Tower Semiconductor	1.1%
8	Powerchip Semiconductor Manufacturing Corp. (PSMC)	1.0%
9	Nexchip	1.0%
10	Vanguard International Semiconductor Corp. (VIS)	1.0%

2.4 Hardware Implementation Platforms

Once designed, hardware components need to be implemented on physical devices. For most hardware accelerators, the two main targets are Application-Specific Integrated Circuits (ASIC) (Section 2.4.1) and Field-Programmable Gate Arrays (Section 2.4.2). While most modern processors are built using ASIC technology, the advent of large FPGAs and more capable FPGA design tools has led to the development of numerous processor cores built for such implementation platforms. The latter are commonly referred to as *soft-core* processors. This section gives an overview of both ASIC and FPGA technological targets, focusing on FPGAs, which are the main target of our work.

2.4.1 Application-Specific Integrated Circuits

Application-Specific Integrated Circuits (ASIC) are custom integrated circuits designed to perform extremely well on a given workload. ASICs can use a fully custom design, where the designers freely choose the combination of transistors for a particular logic function. However, this design methodology is extremely costly. As a consequence, most ASICs are laid out using *standard cell libraries*, *i.e.*, collections of small transistor assemblies that are tailored to the final manufacturing process of the chip [154, 160]. The manufacturer (the *foundry*) often provides these cell libraries directly. Table 2.2 lists the world's leading foundries in terms of market share at the end of 2023.

Today, ASICs are present in a wide variety of consumer electronics, ranging from USB chargers to network switches. The typical design process for ASIC-based hardware accelerators involves the following steps:

1. *Register-Transfer Level (RTL) specification* using a hardware description language such as VHDL or Verilog (see Section 3.1 for more details).
2. *Functional validation* using logic gate simulators, and test benches [255]. This step is paramount in the design of ASICs since, once manufactured, most of them cannot be reconfigured to fix potential issues.
3. *Synthesis* of the hardware description to a logic-gate-level *netlist*. This netlist describes the interconnections and the logic cells required to build an ASIC. The foundry provides the cells as part of a standard cell library.
4. Another functional validation step, to ensure that the logic synthesis result matches the initial specification. This step often involves formal validation methods such as *static formal verification* and *logic equivalence checking* (LEC) [250].
5. *Placement* of the components on the *die* of the integrated circuit, as well as *routing* of the wires that connect those components together.

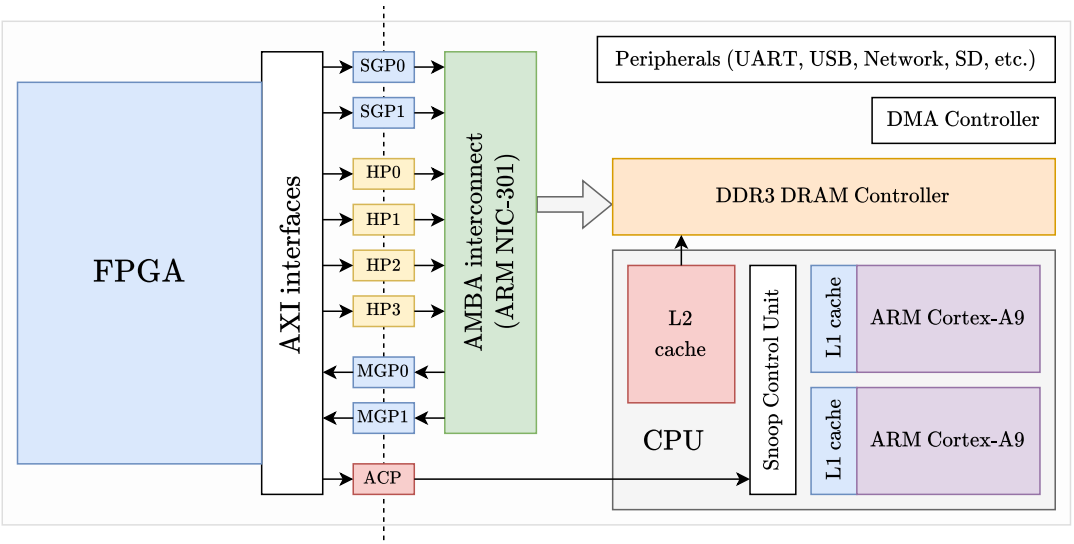


Figure 2.14 – Zynq 7000 architecture overview. The FPGA is only a part of the entire system. It can communicate with a processor comprising multiple ARM cores, external memory, and peripheral devices.

6. *Design rule checking*, where simulation and verification tools are used to ensure that the previous step did not violate any of the design rules imposed by the foundry. This step often also involves power analysis to estimate the power consumption of the final chip design.
7. *Artwork export*, where the final design is exported in a form suitable for manufacturing by a foundry (GDSII or OASIS file). The foundry will create a *mask* of the design using this file as a reference. The mask is then used in their photo-litography manufacturing process [226, 173].

Cadence Design Systems and Synopsys, Inc. are the leading ASIC design tool vendors. ASICs form the basis of most widely produced electronic products and state-of-the-art computation platforms that have stringent requirements regarding performance, electronics surface area, reliability, and energy consumption. However, their high production cost makes them unsuitable for most small-volume applications.

ASIC production costs. The high level of required expertise, coupled with state-of-the-art manufacturing processes, often drives the cost of manufacturing an ASIC up. Consequently, ASICs are often preferred in products with a large expected consumer base, relying on economies of scale to dilute the upfront manufacturing costs through large sales volumes. Field-Programmable Gate Arrays (FPGA) are often favored for smaller production volumes or prototypes.

2.4.2 Field-Programmable Gate Arrays

While ASICs provide many optimization opportunities concerning performance, energy consumption, underlying transistor technology, and surface area, one of their major drawbacks is their inflexibility. Once an ASIC has been manufactured, there is no way to add or remove some features from the hardware. This lack of flexibility makes ASICs a poor choice for contexts that require rapid iteration cycles, such as prototyping. Additionally, the high non-recurring engineering (NRE) costs of ASIC development, such as the validation checks and mask production, render them cost-inefficient for smaller production volumes. Field-Programmable Gate Arrays (FPGA) offer an alternative to ASICs for such use cases. At its core, an FPGA is an implementation platform based around reconfigurable logic blocks and programmable interconnects. It belongs to the broader category of *Programmable Logic Devices* (PLD). The elementary logic blocks inside of an FPGA can be composed to create more

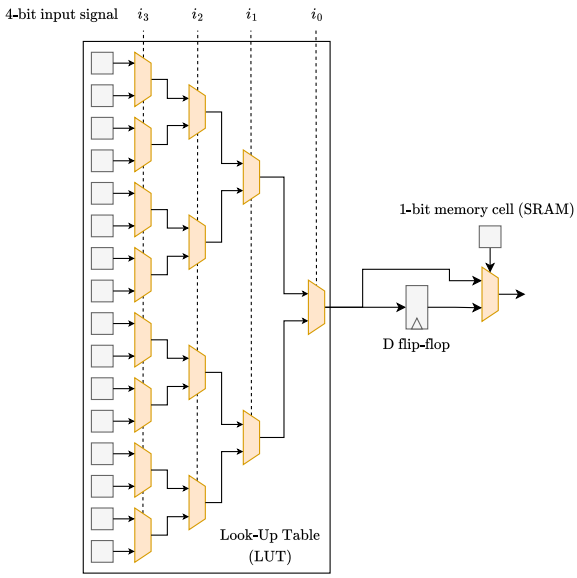


Figure 2.15 – Basic Logic Element (BLE). A 4-input Look-Up Table is used to implement arbitrary 4-input boolean functions, and is connected to the input of a D-type flip-flop. Adapted from [35, 114].

complex hardware functions while retaining maximum flexibility through the ability to reconfigure the connections between logic blocks. The first commercially viable FPGA, the XC2064, was introduced in 1985 by Xilinx [7], an American company founded one year prior, in 1984. Today, the two major FPGA vendors are AMD, through its 2022 acquisition of Xilinx, and Altera, which was acquired by Intel in 2015 and became independent again in 2024.

Most modern FPGAs are not provided as standalone implementation platforms. Instead, they are distributed as highly-integrated System-on-Chip (SoC) platforms, combining a configurable FPGA with processor cores, memory interfaces, and external peripherals. Figure 2.14 gives a simplified overview of the Zynq 7000 SoC architecture from AMD [8].

FPGA hardware components

FPGAs typically consist of tens of thousands to millions of *basic logic elements* (BLE) grouped inside of *configurable logic blocks* (CLB), embedded in a large array of interconnecting wires. They also provide on-chip memory (*block RAM*, or BRAM), arithmetic functions in the form of small Digital Signal Processors (DSP), and I/O elements. High-end FPGAs also embark High-Bandwidth Memory (HBM) banks [394]. Figure 2.15 illustrates a basic logic element similar to what can be found in most modern FPGAs. The basic logic element comprises a 4-bit Look-Up Table (LUT), which can implement any 4-input boolean function. The 4 bits of the input signal select outputs from small 1-bit memory cells, most commonly implemented using SRAM technology. The LUT's output feeds into a D-type flip-flop (see Figure 2.2), which can act as an optional storage element. A configuration bit is stored in a memory cell linked to the output multiplexer to select between the output of the LUT or the D flip-flop.

CLBs contain clusters of interconnected BLEs, typically up to 10 BLEs per CLB [114]. Additionally, modern FPGAs combine BLEs and fixed-function hardware components to provide more efficient computation for typical use cases. These fixed-function components include small Digital Signal Processors (DSP), adders, multipliers, and shift registers. This heterogeneous design allows for smaller hardware design implementation on FPGA while also increasing the performance and energy efficiency of the resulting circuit.

8. https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-7000.html

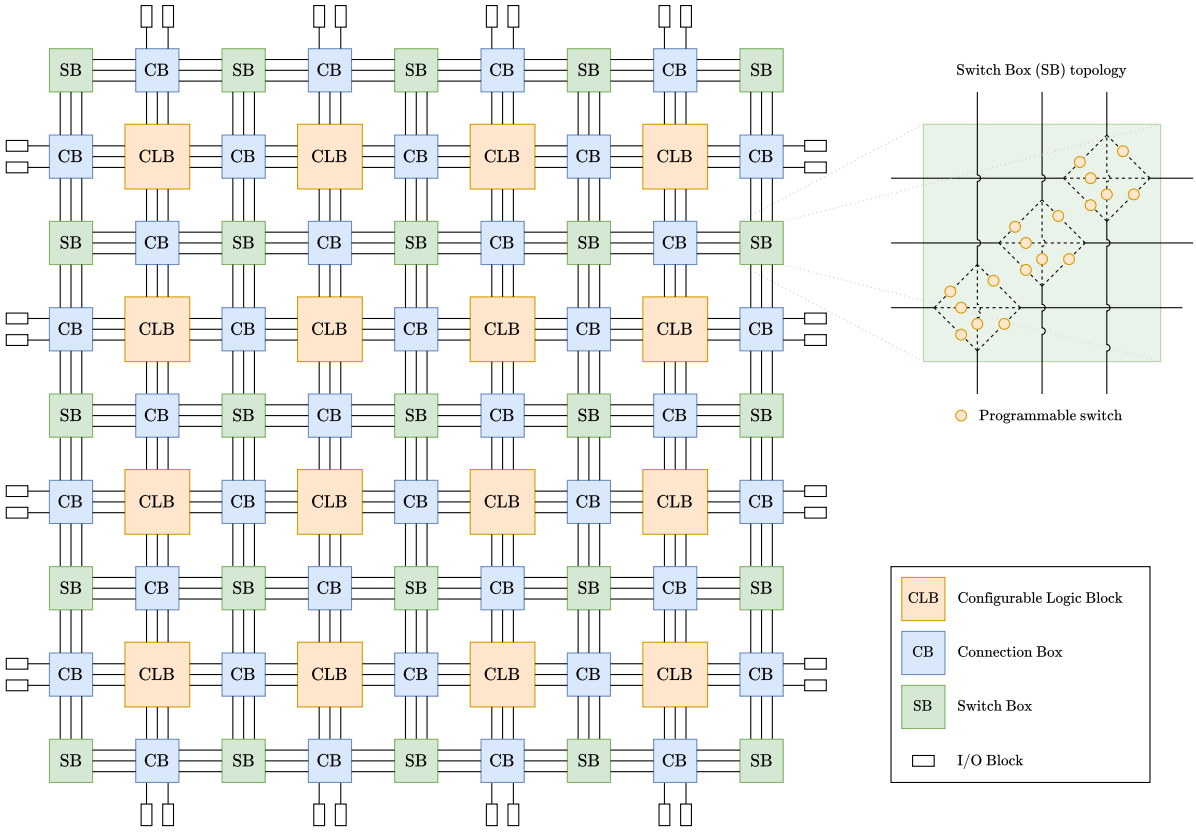


Figure 2.16 – Island-style FPGA routing network. The CLBs are interconnected using a grid of routing channels, with connection boxes distributing traffic and switch boxes serving as configurable routers on the interconnection network. The top-right shows the topology of a switch box, which contains programmable switches to route wires between its inputs and outputs. I/O blocks are located on the edge of the chip and connected to the rest of the network through connection boxes.

Routing network

While the configurable logic blocks form the core of the computational capabilities of FPGAs, they represent only about 10–20% of its total surface area [35]. The remainder of the surface is occupied by a vast routing network, which combines CLBs across the chip to perform computations. This thesis focuses on *island-style* routing architectures, which are common on Xilinx FPGAs.

An example of an island-style FPGA routing network is shown in Figure 2.16. Each CLB is akin to an island of logic elements inside a grid-shaped routing network. *Connection boxes* distribute traffic between the components of the network, while *switch boxes* route said traffic. Each switch box contains a set of programmable switches that allow the configuration of the available routing paths inside the network. I/O blocks, often located off-chip, are linked to connection boxes on the edge of the chip.

Hierarchical routing architectures. Some FPGAs provide a hierarchical routing network designed around the observation that most circuits exhibit localized computation patterns on the chip. While we do not discuss this kind of network, the reader may find a more thorough discussion of hierarchical FPGAs in [114].

The programmable switches of the switch boxes, coupled with the configurable logic functions inside of each CLB, allow FPGAs to provide a very flexible implementation platform. This platform can implement any hardware design that could be implemented on an ASIC, provided sufficient hardware

resources are available. We note that pipelining (as described in Section 2.1.3) is paramount to achieving good performance of FPGA targets. The following section describes how hardware designers can produce a configuration description for a given FPGA target.

An overview of FPGA hardware design

FPGAs contain an internal memory that stores the configuration state of each of their configurable elements. The configuration state of an FPGA is produced in a manner similar to the method used to produce the final layout of an ASIC design. A critical difference between the ASIC and FPGA design flows is the much more lightweight testing and validation steps required for FPGAs: the reconfigurable nature of this platform means that the hardware designer can correct any potential error in the design before sending the new design to the FPGA configuration memory to re-configure the chip. The typical design steps for an FPGA implementation are as follows.

1. *RTL specification* using a hardware description language such as Verilog or VHDL (see Section 3.1).
2. *Functional validation* using simulations. Contrary to ASIC design, this step can be lightweight and rarely involves formal verification methods.
3. *Logic synthesis*, where the hardware description is transformed into a netlist, similar to the ASIC design flow. However, instead of relying on foundry-provided standard cell libraries, the netlist for FPGA synthesis often only encodes the function each logic block should perform (*e.g.*, OR gate, multiplexer, or 16-bit adder)
4. *Technology mapping*, where each component in the netlist is mapped to one of the reconfigurable components provided by the target FPGA platform. This step transforms the macro-operator view of the netlist to a vendor-specific representation, where the actual implementation of each logic function is mapped to the underlying hardware. More details are provided in the next section.
5. *Place-and-route*, where the design toolchain selects the location of each logic function on the FPGA and emits configuration instructions for the FPGA routing network to route wires between the selected components.
6. *Final design export*, where the configuration parameters of the FPGA are emitted as a *bitstream* to be uploaded to the on-board configuration memory.

The preceding steps are sometimes augmented with additional simulation and verification procedures to ensure potential errors are caught early. While immensely more time-efficient than the design process of ASICs, the complete FPGA design flow from functional validation to bitstream export can last from a few minutes to a few days, depending on the design's overall complexity.

Open-source FPGA design tools. The bitstream format is often proprietary and even encrypted, which can sometimes become a security concern [407, 91] and hinders the development of open-source FPGA design toolchains [a] by requiring important reverse-engineering efforts [b] [c].

a. https://f4pga.org/
b. https://github.com/f4pga/prjxray
c. https://github.com/YosysHQ/icestorm

CHAPTER 3

Automated Hardware Design

The most exciting thing about being a scientist is that you can always change your mind.
— *Arthur C. Clarke* (Childhood's End)

Modern processor design relies heavily on Electronic Design Automation (EDA). EDA is an engineering field devoted to designing and implementing methodologies, algorithms, and software to automate parts of the electronic circuit design process. EDA has been a fantastic driver for the rapid development of modern electronics and has helped initiate many revolutions in the hardware design field [94]. This chapter delves into the subset of EDA that focuses on circuit design. We explore its successive iterations, starting with Hardware Description Languages (Section 3.1), before adding more expressive power in Section 3.2, where we describe Hardware Construction Languages. Section 3.4 then presents High-Level Synthesis (HLS), a hardware design methodology where high-level code written in programming languages such as C or C++ is translated to hardware using a *hardware compiler*, letting hardware designers focus on high-level algorithmic design decisions and fast design space exploration. Most critically, HLS abstracts away the notion of time, allowing designers to manipulate flexible *untimed* circuit representations readily amenable to automated design space exploration. The last section of this chapter, Section 3.5, looks at languages and tools specifically geared towards the design and implementation of instruction set processors, namely Architecture Description Languages (ADL). We survey a large number of ADLs and summarize their key features and discrepancies. While most processor vendors still employ low-level hardware description languages for their designs, new tools that provide better abstractions are catching on in the industry. Our work focuses on bridging the gap between processor design and the high levels of abstraction provided by modern automated hardware synthesis toolchains. We argue that the high-level nature, coupled with the flexibility and expressiveness of high-level programming languages, makes HLS a good fit for processor design.

3.1 Hardware Description Languages

The increasing complexity of hardware designs and the multiplication of possible implementation technologies renders electronics design using low-level descriptions tied to specific hardware technologies unpractical. *Hardware Description Languages* (HDL) were designed to abstract hardware designers away from the exact underlying implementation technology, allowing them to focus on the higher-level *structure* and *behavior* of their digital circuits. An HDL is akin to a programming language, with the addition of time-handling mechanisms essential to describe the behavior of digital circuits (see Section 2.1.2). The first HDLs were developed in the late 1960s and early 1970s and laid the foundations for the development of hardware description languages such as VHDL [75] and Verilog [358].

3.1.1 Register-Transfer Level Description

Register-Transfer Level (RTL) [32] is a hardware design abstraction that models electronic circuits as a flow of digital signals through combinational operations between hardware registers (Section 2.1.2). The core building blocks of RTL can be roughly divided into two categories [29]:

— *Operators* transform their input signals into one or more output signals. These operators can be simple logic gates (Section 2.1.1) or more complex combinational networks.

— *Carriers* store and transmit information inside the circuit between operators. Carriers can be stateful components, such as flip-flops, or stateless, such as wires connecting multiple operators.

Operators and carriers can be generalized to hierarchical structures, which play a crucial role in enhancing the design process efficiency. This allows for building higher-level hardware components, such as grouping individual flip-flops to form registers, and registers to form memory banks. Early HDLs introduced this idea of hierarchical structures by allowing designers to group identical components into arrays, which can then be subscripted to access their individual components [13, 31, 28].

A strong advantage of RTL over lower-level hardware descriptions is that it allows designers to efficiently build new abstractions. Components of a circuit can be grouped into more complex operators, which can then be used as a black box in other hardware designs. This ability to raise the level of abstraction for hardware design brings increased productivity and, importantly, increased opportunities for coarse-grain automation and verification (see Section 3.2).

3.1.2 Standard HDLs: VHDL and Verilog

The rapid development of consumer and industrial electronics in the 1980s and the development of new tools and languages to describe hardware prompted governments and their industrial partners to standardize a common language to describe hardware designs. The two most prominent standardized HDLs are VHDL and Verilog. While they were primarily designed for verification and simulation purposes, they quickly became the *de facto* standards for hardware design.

The US Department of Defense tasked contractors with the development of VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language) in 1983, and the language was standardized in 1988 [168]. The language was initially used to provide a unified description of hardware components used by various US DOD systems for reference and simulation purposes. VHDL is a strongly typed language, where each hardware component is described as an *entity* with a given *architecture*. Hardware designed using VHDL can be specified using either a *behavioral* description in the form of a simple algorithm, a *dataflow* specification, where each statement executes concurrently, or a *structural* definition, where components are explicitly instantiated and wired together. Since its first standardization in 1988, VHDL has become a staple of digital electronic design in both academia and industry. All major EDA tool vendors support it. The language is still actively being developed, with new revisions being standardized regularly. The latest release at the time of this writing happened five years ago, in September 2019 [165].

Verilog is a standardized hardware description language [358] that was first introduced as a proprietary design language by Automated Integrated Design Systems in 1984. The company was renamed Gateway Design Automation in 1985 and was acquired by Cadence Design Systems in 1990. Observing the success of VHDL, Cadence decided to open Verilog for standardization in 1995 [167]. In 2009, the Verilog standard was merged with the SystemVerilog standard, which has since become Verilog's default implementation for EDA. SystemVerilog is a strict superset of Verilog that brings new features to aid the verification of hardware designs. In the following, we will use the names Verilog and SystemVerilog interchangeably. Hardware designs built using Verilog consist of a hierarchy of interacting hardware modules.

Similarly to VHDL, Verilog designs can be expressed at multiple levels of abstraction. The *behavioral* level is the highest one. It allows designers to use algorithmic constructs to specify the behavior of the hardware. Verilog also offers *dataflow*-, *gate*-, and *switch*-level abstractions, with each level decreasing the layers that separate designers from the implementation. Note that many of the high-level constructs provided by the language were primarily designed for verification purposes, which means that some EDA

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter is
  port (
    clock: in std_logic;
    reset: in std_logic;
    count: out unsigned(3 downto 0)
  );
end entity counter;

architecture behavioral of counter
↪ is
  signal c: unsigned(3 downto 0);
begin
  process (clock, reset) begin
    if reset = '1' then
      -- set all bits of c to 0
      c <= (others => '0');
    elsif rising_edge(clock) then
      c <= c + 1;
    end if;
  end process;
  count <= c;
end architecture behavioral;
```

(a) VHDL implementation. The `entity` block defines the counter's input and output signals, while the `architecture` block defines its behavior.

```verilog
module counter (
  input wire clock,
  input wire reset,
  output reg [3:0] count
);
always @(posedge clock or posedge
↪ reset) begin
  if (reset) begin
    count <= 4'b0000;
  end else begin
    count <= count + 4'b0001;
  end
end
end module
```

(b) Verilog implementation. The input and output signals are directly specified as arguments to the `module` statement, and the counter's behavior is encoded in the `always` block.

Figure 3.1 – Example implementation of a 4-bit counter in VHDL and Verilog. VHDL requires separate definitions of the interface and behavior of the counter hardware component, while Verilog allows it to be defined in the same module construct.

tools may not be able to synthesize hardware from a description that is too high-level. Like VHDL, SystemVerilog is still under active development and is supported by most, if not all, EDA tools. The latest SystemVerilog standard revision was published in December 2023 [164].

Figure 3.1 shows an example of VHDL and Verilog code for a 4-bit counter. The left of Figure 3.1 gives the VHDL implementation. The `entity` statement is one of the main building blocks of VHDL designs. It defines all input and output ports of the design, with their corresponding types. In our example, `clock` and `reset` are standard logic input signals, which can be thought of as single-bit signals carried over a wire. `count` is the only output signal of our 4-bit counter. It is an `unsigned` 4-bit integer. Signals can either be inputs (`in`), outputs (`out`), or bidirectional (`inout`). The `architecture` statement specifies the `counter` entity's internal architecture. Our example uses the behavioral architectural description, which provides an algorithm to describe the execution behavior of the circuit. The `process` directive is followed by a *sensitivity list* that specifies which signals will cause the component to execute. In the case of our 4-bit counter, the value of the internal count `c` is set to zero when the `reset` signal is high, and it is incremented when the `clock` signal encounters a rising edge (see Section 2.1.2). The Verilog implementation on the right of Figure 3.1 is similar to its VHDL counterpart, albeit less verbose. The arguments to the `module` statement define the module's interface, with `clock` and `reset` defined as input `wire`s, and `count` defined as the single output of the module, stored in a 4-bit register. We note that while VHDL relies on a strong static type system, Verilog employs a static but weak type

system: the output of the `counter` module is specified as containing four bits, which could be a signed or unsigned integer or any four bits that do not represent a number. Contrast this with the VHDL definition, where the output is explicitly defined as an unsigned integer. The `always` statement specified the activation conditions of the module, similar to the process argument list in VHDL. In our example, the counter is active either when `clock` reaches a rising edge or when `reset` is triggered. The behavior of the counter is specified algorithmically, with constructs similar to the ones from software programming languages.

Verilog vs. VHDL. Verilog and VHDL are widely used in the industry, and most EDA tools support both languages to some extent. Both languages provide a host of features, some of which are rarely used in synchronous RTL designs, which make up the larger part of modern hardware designs. The Verilog vs. VHDL debate is long-running, and it has become to EDA what Vim vs. Emacs is to text editors. The reader will find more information about the trade-offs of supporting these two languages in [129].

In the context of processor design, which relies on complex hardware implementations, standard HDLs such as VHDL and Verilog introduce significant overhead regarding development speed and design space exploration. They require extensive validation to ensure the synthesized designs adhere to the designers' constraints. Additionally, changes to parts of the micro-architecture often require extensive rewrites of large portions of the code since the low-level nature and light abstraction support of HDLs make them prone to abstraction leakage and reduce opportunities for code reuse. Verilog and VHDL also provide features that may not benefit most designs: the two languages support constructs (*e.g.* asynchronous execution) that do not benefit the vast majority of hardware designs and introduce an additional cognitive load on designers nonetheless. However, Verilog and VHDL remain the primary languages used for modern processor design. This phenomenon can be partly attributed to the pervasiveness of tools that support these languages and the extensive investments from processor vendors in developing HDL-based implementation frameworks. The work we present in subsequent chapters, especially in Chapter 7, aims to ease the transition from low-level HDL descriptions to higher levels of abstraction for high-performance processor design.

3.1.3 HDL extensions

While HDLs provide all the building blocks required to create digital hardware designs, they can sometimes be cumbersome, making code reuse and verification tedious. Extensions to HDLs have been proposed to address these shortcomings, with many works extending the Verilog syntax and type system to provide stronger guarantees to the designers [413] or abstract away some of the low-level aspects of hardware design with HDLs [159].

SecVerilog is an extension to Verilog that layers a *security type system* on top of Verilog's types [413]. This type system allows security properties of the hardware to be verified statically, during the RTL design phase. More specifically, SecVerilog deals with the problem of *information flow control*, which ensures that every flow of information inside a system respects a set of security properties. This method can be used to mitigate well-known security vulnerabilities, such as timing side-channel attacks [238, 414]. SecVerilog allows hardware designers to tag each variable in their Verilog module with a *security level*. The compiler uses these tags during its type inference pass to ensure no information leaks from high-security levels to lower ones.

Transaction-Level Verilog (TL-Verilog) is an extension to SystemVerilog that abstracts away some of the time handling mechanisms of the base language, allowing hardware designers to elaborate circuits based on *transactional* behavior [159]. The design is split into abstract *stages*, which communicate through a transaction mechanism. This approach improves the reusability and maintainability of the design since fine-grained register mapping decisions are automated by the TL-Verilog synthesis toolchain, depending on the design's implementation target.

While extensions such as the ones discussed in this section enable more efficient hardware design using hardware description languages, they can still be cumbersome to use. Scaling HDL design work

to larger and more complex circuits is challenging, and this has prompted the development of new languages that raise the level of abstraction at which designers can work.

3.2 Higher Abstraction Levels

The late 1990s and early 2000s have seen the development of new hardware design techniques aimed at increasing automation in the design process [239]. With the advent of more powerful automated analysis tools and techniques, new hardware design languages emerged. Those languages focus on lifting the abstraction level of hardware design to increase the productivity of designers and the correctness of their designs [270, 25]. While they still target RTL designs, these languages provide facilities to automate parts of the design process. This section focuses on three languages designed with such goals in mind: Bluespec (2004) [270] in Section 3.2.1, Chisel (2012) [25] and SpinalHDL [1] (2014) in Section 3.2.2. We discuss examples of processor cores designed using these tools and the suitability of higher-level hardware description languages for processor design. Section 3.3 gives an overview of related works in high-level hardware description languages.

3.2.1 Bluespec SystemVerilog

Verilog and VHDL provide a very low-level view of hardware to the designer. They require a global understanding of the interactions between all the circuit's components to produce efficient hardware, which makes the design process tedious for large components. Additionally, this low-level view is prone to subtle bugs, which can arise during the initial design phase or later refactorings. Bluespec SystemVerilog (BSV) aims at providing a higher-level view to hardware designers while remaining familiar to practitioners used to Verilog or VHDL [270]. The language builds upon SystemVerilog syntactic constructions and the Verilog type system. BSV introduces the concept of *guarded atomic actions* (also called *rules*) to Verilog. These rules allow designers to express atomic transitions between states of the hardware design. They are based on a restricted form or Term Rewriting Systems (TRS) [357, 273], and guarantee that each rule will run *as-if* they were executed atomically. Actual atomic execution of rules in hardware would lead to poorly performing designs. Instead, the BSV compiler performs static analysis on the input code to determine a *schedule* of rules that allows maximum parallel execution while still preserving the atomicity guarantees of state updates.

We provide an example implementation of a 4-bit counter in BSV in Figure 3.2. Similarly to the Verilog example of Figure 3.1, the counter is defined as a hardware module. However, instead of directly manipulating wires and registers, the module's behavior is entirely specified using atomic rules. The code example of Figure 3.2 can be split into a few distinct sections:

1. The top of the file defines a BSV *interface*. BSV interfaces are similar to their counterparts in software programming languages: they define a set of methods that modules implementing the interface must provide. These interfaces specify the communication protocol between different hardware modules in the design. In our example, any hardware module implementing the `ICounter` interface must provide a `readCounter` method. Note that the `ICounter` interface is parameterized with the underlying type of the counter.
2. The storage type of the counter value, `CounterType`, is defined as a 4-bit integer.
3. The `incReg` function returns a set of rules, parametrized by its input parameter, `a`. By allowing functions to return rule sets, BSV allows hardware designers to provide common rules to multiple hardware modules generically.
4. The `counter` module defines the actual hardware component for our 4-bit counter. It implements the `ICounter` interface and provides the corresponding `readCounter` method. The `counter_value` variable is defined as a register using the `mkReg` constructor, with an initial value of 0. Note that this initial value is implicitly used to reset the counter if the `reset` signal is triggered. The module's body contains the definition of a rule, `resetCounter`. The rule is coupled to an

1. https://github.com/SpinalHDL/SpinalHDL

```
interface ICounter#(type t);
  method t readCounter;
endinterface

typedef Bit#(4) CounterType;

function Rules incReg(Reg#(CounterType) a);
  return (rules
      rule addOne;
        a <= a + 1;
      endrule
    endrules);
endfunction

(* synthesize, reset_prefix = "reset", clock_prefix = "clock",
   always_ready, always_enabled *)
module counter (ICounter#(CounterType));
  Reg#(CounterType) counter_value <- mkReg(0);
  addRules(incReg(asReg(counter_value)));

  rule resetCounter (counter_value == '1);
    action
      counter_value <= 0;
    endaction
  endrule

  method CounterType readCounter;
    return counter_value;
  endmethod
endmodule
```

Figure 3.2 – Bluespec SystemVerilog code for a 4-bit counter. Instead of directly manipulating wires and registers, the counter's behavior is specified using *guarded atomic actions* encoded in `rules`. Adapted from the *BSV Language Reference Guide* [39].

activation condition that checks if the counter value contains all ones (`'1`). The `addRules` call adds `addOne` to the module's behavior-defining rules. We note that `addOne` has no activation condition, meaning it runs on each clock cycle.

Bluespec SystemVerilog provides a wide range of utilities to make developing and maintaining hardware descriptions easier. The rule-based update system provided by BSV and its derivatives [44] allows for developing efficient verification tools able to take advantage of the guaranteed atomic update behavior of rules.

Bluespec Inc. [2], the company behind the development of the language, maintains three open-source BSV-based RISC-V processor core implementations: Piccolo [3], a 3-stage in-order pipelined core, Flute [4], a 5-stage in-order pipelined core, and Toooba [5], a superscalar out-of-order pipelined core based on the RiscyOO BSV implementation [415]. Those three implementations leverage the abstractions provided by the BSV language to provide modular and configurable processor implementations. Using the Bluespec

2. https://bluespec.com/
3. https://github.com/bluespec/Piccolo
4. https://github.com/bluespec/Flute
5. https://github.com/bluespec/Toooba

```
val cntReg = RegInit(0.U(4.W))

cntReg := cntReg + 1.U
when(cntReg === 15) {
  cntReg := 0.U
}
```

Figure 3.3 – Chisel code for a 4-bit counter. Note that the hardware specification uses regular Scala constructs to express the hardware's behavior. Assignments are executed at each clock cycle. Adapted from [332].

compiler [39], users can generate multiple variants of the same core by enabling or disabling specific ISA or micro-architectural features using compile-time definitions. Similar approaches have been used in other open-source processor designs [415, 331] and have been successfully leveraged to perform design space exploration for the synthesis of instruction set processor cores [330]. However, most of the structure of the processor's micro-architectural components is still rigidly defined, which limits the discoverability of hardware configurations that may not correspond to textbook representations of processor pipelines. For example, the Bluespec in-order RISC-V cores, Piccolo and Flute, can only be generated using their predefined pipeline depth.

3.2.2 Hardware Construction Languages

BSV lifts the level of abstraction for hardware designers, but it still lacks many modern programming language features that enable faster development cycles. Starting from a similar observation about Verilog and VHDL, researchers from UC Berkeley introduced Chisel in 2012 [25]. Chisel stands for *Constructing Hardware in a Scala Embedded Language*. It is presented as a hardware *construction* language (HCL) and was developed as a Scala eDSL (embedded Domain-Specific Language). At its core, Chisel is a collection of libraries and classes that leverage the flexibility of the Scala programming language to provide hardware design features embedded inside a high-level programming language. A Chisel design is a Scala program that *generates* an RTL description of a circuit during its execution [332]. This hardware generation paradigm is what sets HCLs apart from HDLs: much like a macro-assembler can provide high-level facilities to generate assembly language constructs (*i.e.*, *metaprogramming* features), hardware construction languages leverage high-level programming languages to generate RTL descriptions in standard HDLs.

Chisel is a peculiar hardware design language in that it caters to the needs of both hardware designers and software programmers. It provides all the low-level tools that designers experienced in standard HDLs need to express their intent while simultaneously introducing high-level concepts, such as object-oriented and functional programming, from the world of software to the world of hardware design. Chisel includes several builtin operators that make it easier to instantiate standard hardware components: arithmetic operators such as integer addition and multiplication, comparison operators, and logic operators such as AND or XOR, which are all part of the core language. Additionally, the language provides bit manipulation utilities such as bit extraction from signed and unsigned integers or bitfield concatenation. Finally, multiplexers, registers, and wire abstractions allow users to define low-level details of their design, similar to standard HDLs such as Verilog and VHDL.

Figure 3.3 gives a possible implementation of our recurring 4-bit counter example, adapted from [332]. Contrary to the HDLs presented in Section 3.1 and 3.1.3, most of the signals governing hardware execution are implicit. Chisel is built around a synthesis model where the input program's execution defines the behavior of the generated hardware component. We can leverage this synthesis model to create complex layouts and designs through the use of *generators*. Hardware generators are a fundamental building block of Chisel designs. They can be simple functions whose body is a template to generate specific hardware components. Figure 3.4 gives an example of a counter generator. The `genCounter`

```
def genCounter(n: Int, width: Int) = {
  val cntReg = RegInit(0.U(width.W))
  cntReg := Mux(cntReg === n.U, 0.U, cntReg + 1.U)
  cntReg // Return value of the function.
}

// Example counter instances.
val count10 = genCounter(10, 8)
val count1024 = genCounter(1024, 11)
```

Figure 3.4 – Chisel counter generator. The body of the `genCounter` function is a template for a generic counter that counts up from 0 to `n` (included). The `width` parameter specifies the bitwidth of the counter's internal state. Adapted from [332].

function creates a counter that counts up to the given integer. Note the use of the built-in multiplexer class, `Mux`, instead of the `when` statement of Figure 3.3. The `width` parameter gives the internal counter state's bitwidth. The full power of generators can be seen in more complex examples, where designs that would otherwise require manual unrolling in standard HDLs can be factored into simple functions, taking advantage of the Scala execution model to generate the hardware layout. For example, an *adder tree* may be specified using a recursive Scala function instead of manually wiring multiple adder stages together.

The programming facilities and the abstraction building blocks provided by Chisel make hardware design a more productive task. In his introductory textbook on Chisel titled *Digital Design with Chisel* [332], Martin Schoeberl introduces his readers to processor design using Chisel. Even though the architecture and micro-architecture of the processor used as an example in his book is simple, such an endeavor would be significantly more difficult in a standard HDL than in Chisel. The hardware construction language has also been successfully used to design complex micro-architectures. SonicBOOM [419] is an example of an Out-of-Order 64-bit RISC-V core implemented using Chisel. It features a high-performance execution engine and can compete with commercial OoO cores on standard benchmarks. Hardware design practitioners on several online discussion forums report a dramatic increase in productivity and testability after switching from HDLs such as VHDL and Verilog to Chisel.

SpinalHDL is another hardware construction language that started in 2014 as a fork of Chisel. Like its parent language, it leverages the powerful eDSL capabilities of Scala to provide a software-like development environment for complex hardware designs. It distinguishes itself from Chisel by a stronger type system for signals and more meta-programmation capabilities. However, designers using Chisel or SpinalHDL can easily switch from one to the other, depending on their needs and constraints. SpinalHDL has been used to design various processors, including several open-source cores. For example, MicroRV32 [3] is a configurable RISC-V processor designed in SpinalHDL. This core was created primarily for education and research purposes. Another successful use of this hardware construction language is the VexRiscv [6] processor, a 32-bit RISC-V CPU implementation designed explicitly for synthesis on FPGA targets. It does not use proprietary or vendor-specific hardware blocks and can be fully synthesized on various FPGA platforms. Similarly to the Bluespec cores discussed in Section 3.2.1, VexRiscv can be configured at synthesis time to enable or disable certain architectural and micro-architectural features. The core supports execution of multiple operating systems, including Linux, with the LiteX project [7]. Another notable mention of processor designed using SpinalHDL is NaxRiscv [8], an Out-of-Order design that also targets FPGAs as its primary implementation platform. VexRiscv and NaxRiscv form a good comparison basis for in-order and out-of-order pipelined soft processor cores.

6. https://github.com/SpinalHDL/VexRiscv
7. https://github.com/litex-hub/linux-on-litex-vexriscv
8. https://github.com/SpinalHDL/NaxRiscv

The hardware generation paradigm initiated by Chisel and refined by SpinalHDL can be a powerful tool for designing and implementing instruction set processors. It allows various tasks, such as decode logic generation, to be automated more easily and provides useful tools for code reuse and abstraction. By relying on a well-established programming language as their basis, both hardware construction languages can leverage compiler technology to advance the field of hardware design.

Leaky abstractions. Higher abstraction levels bring faster iteration times and more flexibility to hardware design. However, the languages described in the previous sections still allow their users to work at a low level of abstraction: they can (and sometimes even need to) manipulate low-level abstractions such as registers to build higher-level components. The potential for abstraction leakage makes it difficult to explore large design spaces without much rewriting, especially when the overarching abstractions are poorly designed.

3.3 Bridging the Gap Between Compiler and Hardware Design

The compiler community has recently seen a rising interest in hardware design problems. The development of open-source hardware construction languages such as Chisel has fostered a growing collaboration ecosystem between compiler and hardware designers [171]. In 2010, some works already advocated for a digital design methodology shift towards high-level *chip generators* instead of individual chip designs [335]. This idea was exemplified by Genesis2, a SystemVerilog extension that allows hierarchical hardware structures to be expressed procedurally [336]. This same idea was implemented in Chisel [25] and SpinalHDL. New hardware description and construction languages are still being developed, with new ones bringing innovations from compiler design and optimization to hardware synthesis [21, 127, 329, 295].

Concurrently with the development of hardware construction languages, industrial and academic efforts have led to the rapid development of High-Level Synthesis (HLS) toolchains, which enable hardware synthesis from a purely algorithmic description of the hardware's behavior in a high-level language such as C or C++. HLS is presented in Section 3.4.

Most intermediate representations in modern hardware compilers are based on SSA form, which is a program representation where each variable is assigned precisely once [5, 85, 81]. This choice is primarily pragmatic: SSA-based intermediate representations quickly became widespread in software compilers [5, 217], some of which expose interfaces for extensions. Consequently, extending a software compiler for hardware design purposes is a lower risk and lower-effort task than maintaining an entire hardware compilation stack. The search for alternative representations better suited for hardware still carries on, with derivatives of SSA such as Gated-SSA [278, 366, 367] showing promising results for parallelism extraction in high-level language programs [17]. Gated-SSA is exposed in more detail in Section 4.3.4, where we show how we leverage its explicit predicate encoding to expose speculation opportunities in hardware designs. Other works focus on encoding hierarchical constructs directly in the compiler's intermediate representation. For example, the CODESIS HLS toolchain [203, 204] relies on Hierarchical Conditional Dependency Graphs (HCDG) to improve the synthesis of hardware from control-dominated high-level code. Other approaches are built atop Hierarchical Control-and Data-Flow Graphs (HCDFG) [397, 265]. This hierarchical approach to hardware synthesis naturally lends itself to multi-level design approaches [381], where high-level abstractions are progressively lowered to lower-level views of the hardware design.

This multi-level view of compilation has its parallel in software compilers. This similarity between compiler and hardware design questions is outlined by [171] and forms the basis of novel hardware-oriented compiler design frameworks such as CIRCT [9]. The CIRCT initiative leverages the MLIR compiler infrastructure [218] to provide a framework for users to create new hardware design tools and languages. It focuses on a multi-level approach to compilation that has proven to be successful in the design of new hardware-oriented intermediate representations and compilers [333, 402, 242]. This initiative is discussed in more detail in relation to our work in Chapter 8.

9. https://circt.llvm.org/

3.4 High-Level Synthesis

The hardware description and construction languages presented in this chapter require users to learn a new programming paradigm. They emphasize low-level hardware control, with many tools requiring explicit pipelining decisions and detailed implementation decisions. Most importantly, they require designers to explicitly handle the notion of *time*, placing the burden on them to schedule operator execution and coordinate data flow in their design. High-Level Synthesis (HLS) is a hardware design technique that aims to abstract designers away from these low-level details. The core idea of HLS is that hardware designers should be able to specify the behavior of their designs from an algorithmic perspective in an *untimed* high-level programming language. With some additional semantics attached to the input programming language, different hardware structures can be inferred from syntactical constructs. This approach enables efficient design space exploration, and user-specified annotations can guide the synthesis process to generate the desired hardware components.

3.4.1 A Brief History of High-Level Synthesis

High-Level Synthesis (HLS) was first proposed to overcome the many limitations of RTL design flows w.r.t. design space exploration, code reuse, and abstraction [110]. The first commercial High-Level Synthesis toolchain was Synopsys' Behavioral Compiler [197]. This tool, released in 1994, synthesized hardware from purely behavioral Verilog or VHDL descriptions (see Section 3.1). While the idea of behavioral synthesis was appealing, the low-level nature of both Verilog and VHDL proved unsuitable for large-scale designs [105].

The Cynthesizer tool [253], released in 1998 by Forte Design Systems, used SystemC as its input language. SystemC is a set of C++ classes and utilities that allow for the design of hardware systems directly in C++. The language uses an event-driven programming model, where *processes* execute the core hardware functionality and communicate via *channels*, using dedicated *events* to synchronize their execution.

Modern commercial HLS toolchains often rely on C or C++ to describe the algorithmic behavior of hardware. An HLS toolchain infers the hardware's structure from this high-level description and the required resources and clock frequency constraints. This approach abstracts the user away from most implementation details and makes changes to the hardware easier to apply, significantly improving the designer's ability to conduct design space exploration. High-Level Synthesis tools are developed both in academia and by major EDA vendors. Commercial tools include AMD's Vitis HLS [10] Cadence's Stratus HLS [11], Siemens' Catapult HLS [12], and Intel's oneAPI DPC++/C++ compiler [13], all based on C++. These tools extend the language with implicit semantics (*i.e.*, some language constructs are translated to a specific hardware structure) and restrict the language features that can be used to describe hardware behavior. Most notably, HLS introduces strong restrictions on dynamic memory management and global variables and provides limited support for pointer arithmetic.

Early High-Level Synthesis tools often produced disappointing results compared to manual designs by experienced hardware designers. These new tools were aggressively marketed to hardware design companies, but the quality of the synthesized implementation was below many designers' expectations. After over a decade, continuous improvements to HLS toolchains have rendered them usable in commercial scenarios, with some producing designs in line with the results obtained through hand-optimized RTL. HLS has become one of the main development methods for FPGA designs [80, 79, 365], and it has become essential for the design of complex ASICs [18]. High-Level Synthesis is still an active research area, with many new user experience improvements and optimization techniques being developed in industry and academia.

10. https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html
11. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html
12. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/
13. https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html

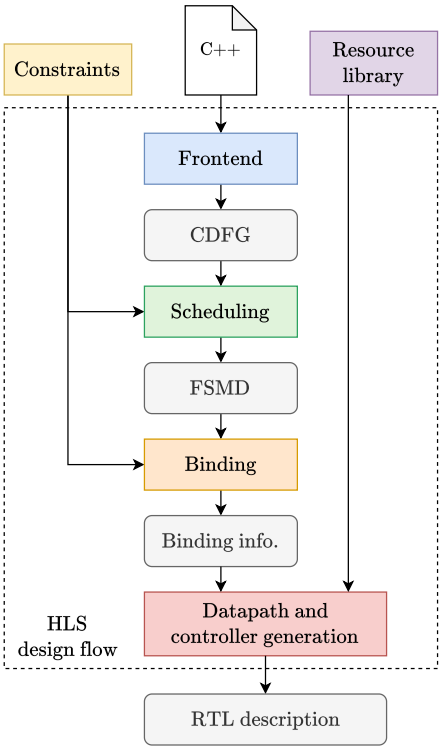


Figure 3.5 – General principle of a High-Level Synthesis toolchain. The input code is transformed into a low-level (RTL) description of the hardware through a series of transformations that progressively lower the abstraction level of the representation on which they operate. Adapted from slides by Philippe Coussy.

3.4.2 Mapping High-Level Languages to Hardware

High-Level Synthesis toolchains use high-level programming languages as their input. This approach lowers the barrier to entry for hardware design while simultaneously allowing for greater design reuse and faster iteration times. However, most programming languages are designed for execution on regular processors. Their programming model is linked to *sequential* machines that execute instructions in program order, and they often have a flat view of memory (*i.e.*, memory can be seen as a large contiguous array of addressable bytes). In contrast, efficient hardware designs are intrinsically *parallel*, and memory is often distributed into many blocks that may not all be addressed using the same interface. Additionally, software programming languages do not incorporate the notion of *time* into their programs. However, time is a critical part of synchronous hardware design, where components are updated at each pulse of a continuously operating clock.

The discrepancies between the software programming paradigms and hardware design may lead us to believe they cannot be reconciled. While high-level languages have been successfully used to create hardware designs (*e.g.*, using Scala in Chisel), they are often used as low-level hardware *generators*, with low-level hardware abstractions provided as core building blocks in the language (*e.g.*, in the form of custom classes). In contrast, HLS toolchains that rely on, *e.g.*, C or C++, are designed to be used in the same way as regular C and C++ compilers. To this end, they must extend the languages with an *implicit* semantics to incorporate timed elements into their input programs. HLS compilers attach timed execution semantics to syntactical language constructs like loops. For example, in most HLS toolchains, one loop iteration is considered to be executed in one clock cycle. Additionally, HLS compilers restrict the operations available to users for memory management and pointer manipulation: dynamic memory allocation is impossible in hardware, and the banked memory of hardware makes pointer arithmetic

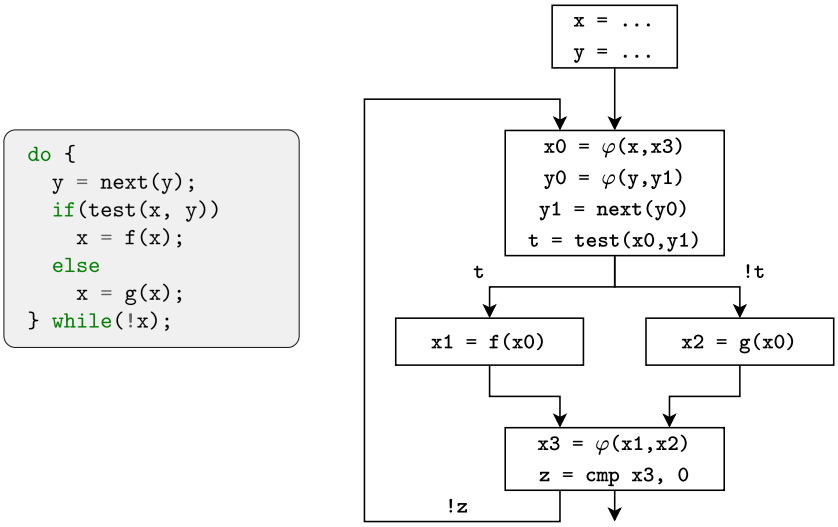


Figure 3.6 – SSA CDFG representation of a simple program. Each variable is assigned exactly once in the SSA representation, while it may be overwritten in the original program. Data-flow is encoded in the *def-use* chains of SSA form.

undefined in most cases. HLS maps program variables to stateful hardware components based on their type: scalar variables are mapped to registers, and arrays are stored in distributed memory banks or external memory, accessed through an I/O interface.

The general principle of High-Level Synthesis is depicted in Figure 3.5. The input code (in C++) is progressively lowered to an RTL description of the corresponding circuit. This process involves four key steps described in the following sections. Note that the order in which these steps are depicted in Figure 3.5 may vary from one HLS toolchain to the next since there are inherent feedback loops between the different transformation passes that can exist inside such a hardware compilation toolchain.

Frontend

The frontend converts the input code into an intermediate representation (IR) amenable to static analysis and optimization. The program is split into several *basic blocks*, corresponding to regions of sequential operations exempt from any control flow and terminated by a single control-flow operation. The targets of the latter are then linked to their source to form the *Control-Flow Graph* (CFG) of the program. In addition to the CFG, the HLS frontend also creates a data-flow representation of the program in the form of a *Data-Flow Graph* (DFG) that exposes the dependencies between variables and operators in the input program. The combination of the CFG and the DFG in each basic block forms a representation denoted as CDFG (for *Control-and Data-Flow Graph*).

Figure 3.6 shows the SSA CDFG representation (right) of a simple loop (left). The data-flow information is implicitly encoded by the variable definitions and their uses in the IR, along with what is commonly referred to as *def-use chains*. In order to follow SSA's rules about single assignment to variables, control-flow join points introduce φ-functions (or φ-nodes) to select between two possible values for a variable. The φ-functions in the loop header select either the initial values of `x` and `y`, or their updated values from the loop body, while the last φ-function selects the correct value for `x` from the conditional statement. Note that φ-functions are only used to encode control-flow join points and do not possess any semantics from the standpoint of the input program. The condition that determines which value of the variable to choose for a given φ-function is not encoded in the latter; some variants of SSA form explicitly encode predicates in φ-nodes, which can make them more attractive for hardware synthesis purposes. More details on such a representation can be found in Chapters 4 and 5.

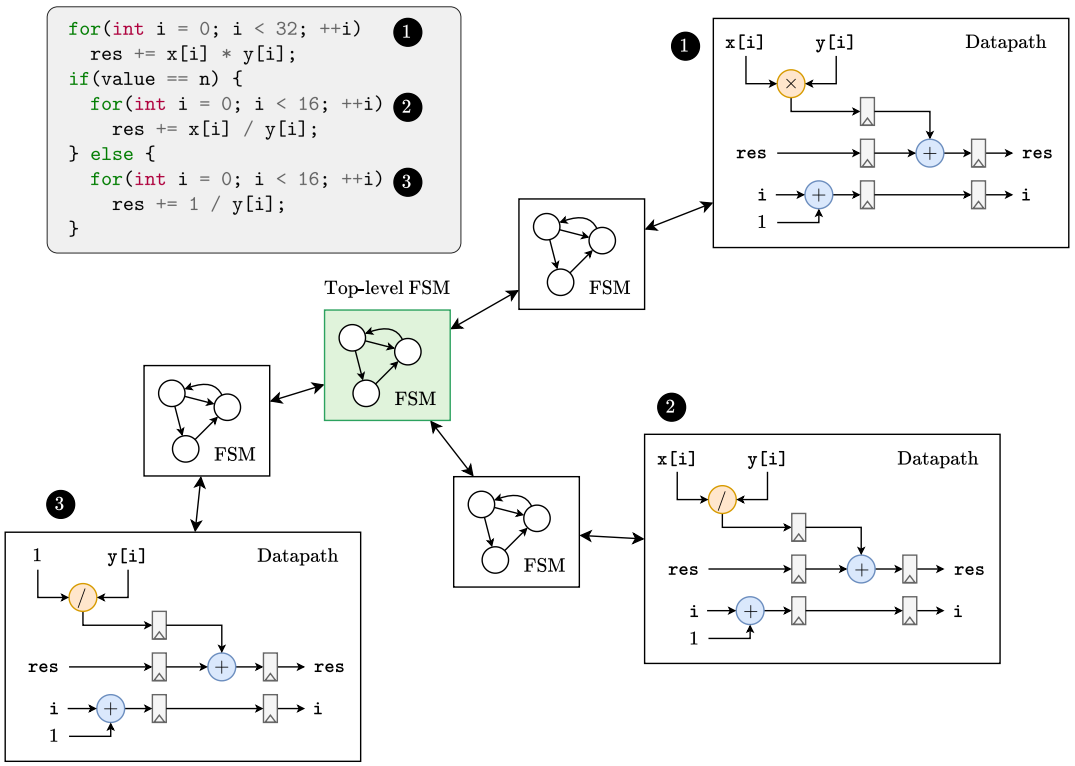


Figure 3.7 – Decomposition of a function into multiple FSM-controlled datapaths. Each loop in the input program is mapped to a separate datapath. The top-level FSM coordinates the execution of all hardware components based on the control flow of the input program. Adapted from slides by Steven Derrien.

Scheduling

Since the high-level languages used as inputs to HLS toolchains do not include any notion of timed execution, the compiler must define when each operation should start executing. This step of the synthesis process is called *scheduling*. The core idea behind scheduling is to take the data dependency constraints encoded in the DFG, coupled with constraints on the target execution frequency set by the user, and produce a schedule that fits those requirements. HLS toolchains also rely on a model of each operator's execution latency in order to be able to place each operator into its corresponding execution cycle without creating overlaps and violating dependencies.

HLS toolchains can rely on many different scheduling algorithms: from list scheduling [112] to minimize execution latency to modulo scheduling [215, 313, 312] to maximize throughput, to force-directed scheduling [285] when trying to minimize resource usage. More advanced techniques based on Integer Linear Programming (ILP) [276] and System of Difference Constraints (SDC) [77] have also become relevant to synthesize better designs, be it from a performance, energy consumption, or die area perspective. Scheduling has a profound impact on the quality of the hardware that can be synthesized by an HLS compiler, which is why it remains the subject of many research endeavors. Most modern HLS design flows use a combination of several scheduling techniques, favoring some over others when trying to achieve a synthesis goal defined by their user. The output of the scheduling algorithm is a scheduling function, a *schedule*, that associates each operator in the input program's DFG to a cycle at which it should start its execution. This schedule can then be used to derive a Finite-State Machine that will control the execution of the circuit in each basic block. These basic-block-level FSMs are linked together by a top-level FSM that pilots the execution flow of the entire program by relying on information provided by the CFG.

```
void add_round_key(int8_t n, uint8_t *result, uint8_t *state,
                   uint8_t key[240]) {
  for(int8_t i = 0; i < 4; ++i) {
    for(int8_t j = 0; j < 4; ++j)
      result[i * 4 + j] =
                state[i * 4 + j] ^ key[n * 16 + i * 4 + j];
  }
}
```

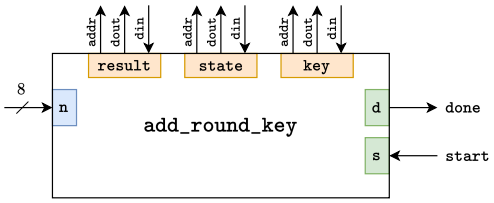


Figure 3.8 – Inferred port interface of the AES-256 `add_round_key` function. The synthesized hardware component's inputs and outputs correspond to the function's input and output variables. Memory operand interfaces have an address (`addr`), input data (`din`), and output data (`dout`) port.

Figure 3.7 gives an example of the final FSM with Datapath (FSMD) representation produced by the scheduling step of the HLS flow shown in Figure 3.5. Each loop, marked ❶ to ❸, is mapped to a separate datapath. This datapath contains combinational operators and registers at cycle boundaries, which an FSM pilots. Loop ❶ is lowered to a two-stage pipeline with a non-pipelined multiplication operator. Similarly, loops ❷ and ❸ contain two pipeline stages. Note that all of the datapaths are not required to be lowered to the same number of pipeline stages. The program's sequential execution flow drives the top-level FSM, which sends control signals to the datapath FSMs to enable or disable their execution. The conditional check in the input code is encoded in the top-level FSM, which selects the appropriate result based on the value of the `value` and `n` variables. In the example of Figure 3.7, the two conditional branches are complex loops, which result in the creation of separate datapaths. For simpler conditional structures that do not contain loops or function calls, the HLS compiler may infer a simpler construct, choosing to execute both branches simultaneously and selecting the appropriate result using a multiplexer.

Datapath merging. The hardware datapaths generated for loops ❷ and ❸ of the example in Figure 3.7 differ only in the first input to the division operator. Sufficiently advanced program analysis could detect such similarities and merge the resulting datapaths to reduce the surface area of the final hardware design. Few HLS tools incorporate such an advanced technique suitable for general-purpose cases. Automatic datapath merging is a well-studied problem, on which interested readers may find more information in the works of Moreano *et al.* [262, 261, 343].

Since the scheduling algorithms employed by HLS tools rely exclusively on statically known information encoded in the CDFG and user-defined constraints, the schedules they produce are always constrained by the worst-case execution time of any given datapath. While this limitation is not an issue for applications with regular control flow, it can become a significant performance bottleneck for applications with irregular and frequent control flow decisions. Chapter 4 delves deeper into the limitations of static scheduling in HLS and explores state-of-the-art alternatives that form the basis of our work.

Binding

The resource *binding* step that follows scheduling is responsible for assigning actual operators on the implementation target to operators in the CDFG. It is also concerned with laying out data in memory according to the access patterns that are present in the input program; contrary to compilation for regular instruction set processors, this data layout process has to deal with selecting between different types of memory banks, depending on the number of access ports they provide, their size, their access latencies, and even their position relative to other operators in the final layout of the circuit. Resource binding is often resource-constrained or time-constrained and heavily depends on the results of the scheduling step.

Assigning operators and data to particular resources impacts many characteristics of the final hardware design, including area, clock frequency, and energy consumption. Binding is thus a complex

problem, often requiring trade-offs to solve a given use case. Many trade-offs are centered around the notion of *resource sharing*, where the compiler assigns a resource to more than one operation or piece of data in the input program. For example, a potentially expensive multiplier may be shared between two datapaths if the schedule assigns them to two execution paths that are guaranteed not to cause conflicts. We note that increased clock frequency and performance constraints severely limit the potential for resource sharing since high-throughput architectures often rely on achieving as much parallel execution as possible. This interplay between scheduling and resource binding prompts for the development of compilation techniques that can increase resource sharing opportunities [262, 261, 343, 146, 150, 258]. However, sharing resources is no panacea: shared resources often require multiplexers to handle routing of data in and out of the shared part of the circuit, which can have an adverse impact on surface area or achievable clock frequency [146, 150].

Another area where resource binding and scheduling may interact is in inferring the interface of a hardware module. High-Level Synthesis operates at the function level to define hardware modules. The latter can call each other, and the appropriate communication interface will be emitted during the RTL generation step. The port interface of each module is determined from the function's prototype. Figure 3.8 illustrates the interface inferred by an HLS toolchain for the `add_round_key` function of the 256-bit AES (Advanced Encryption Standard) algorithm [355]. The scalar variable `n` is directly transmitted using an 8-bit wide wire. In contrast, the `result`, `state`, and `key` arrays use an external memory interface to communicate with the synthesized hardware module. Additional data ports, such as `d` (done) and `s` (start), are automatically added by the synthesis process to allow for external control of the component's execution. However, the port interface of the synthesized hardware component can sometimes constrain the possible execution schedule of its internal components, which is why it needs to be taken into account during the scheduling step. For example, while local memory may provide quick access to data (at least one cycle, but often not much more) in the form of local register files or memory blocks (such as the distributed SRAM blocks in FPGAs), external memory can take an unknown number of cycles to send back the requested data. The user can often tweak the communication protocol used by synthesized components using annotations in the input source code.

Datapath and Controller Generation

Once resources have been allocated to operations in the input program's CDFG, the final step of the HLS flow is to generate the circuit's RTL description. HLS compilers generate an RTL description of the datapaths and their corresponding FSMs, which can then be used to implement the synthesized circuit. These descriptions are often emitted in standard HDL formats such as VHDL and Verilog.

The usual HLS design flow is somewhat similar to RTL-based design using standard HDLs in that it requires many iterative refinement steps to achieve the desired results. HLS users often need to tweak and annotate their code after synthesis to improve the area and frequency characteristics of the final generated circuits. However, a major difference between HLS and HDL design processes is their corresponding time scales. While HDL designs may require several months of work, HLS designs often require a few days or weeks to achieve comparably performing results [18]. This decrease in iteration time enables more thorough design space exploration capabilities, which has led to the increasing popularity of HLS even for complex designs [79].

3.4.3 High-Level Synthesis Annotations

Modern High-Level Synthesis toolchains leverage decades of compiler optimization research to optimize the code provided by users in order to generate efficient hardware [57, 215, 395, 217, 42]. Modern HLS toolchains provide tools to their users to apply code transformations without sacrificing the readability (arguably) of the original code. These transformations are often described using code annotations, either in the form of `pragma` preprocessor directives (similar to OpenMP [86]) or attributes. The latter have the advantage of being an actual language construct that can directly interact with syntactical elements to modify the compilers' representation of the input program. Since our work relies

Table 3.1 – Common AMD Vitis HLS `pragma` annotations. These annotations are provided to give hints to the optimizer and hardware synthesis backend.

Name	Type
`pragma HLS inline`	Function inlining.
`pragma HLS unroll`	Loop optimization.
`pragma HLS dependence`	Loop optimization.
`pragma HLS loop_flatten`	Loop optimization.
`pragma HLS loop_merge`	Loop optimization.
`pragma HLS array_partition`	Memory access optimization.
`pragma HLS array_reshape`	Memory access optimization.
`pragma HLS latency`	Kernel optimization.
`pragma HLS pipeline`	Pipeline structure inference.

mainly on AMD Vitis HLS, we will use the rather unfortunate `pragma`-based approach in the remainder of this thesis.

Table 3.1 gives an overview of common `pragma` annotations that users can use in the AMD Vitis HLS development environment. These annotations are attached to specific regions of the input source code to provide hints and directives to the optimizer and hardware generation backend. The following provides a brief overview of the features provided by the annotations listed in Table 3.1:

- `pragma HLS inline` defines the *inlining* behavior for the annotated function. When used without arguments, the body of the annotated functions is inlined at each call site, *i.e.*, its body is pasted in-place instead of performing an actual function call. When given the `off` argument, inlining is disabled for the marked function. The `recursive` argument enables recursive inlining of all functions called inside of the body of the annotated function. This pragma has equivalent spellings in C and C++ software with the `always_inline`, `noinline`, and `flatten` attributes provided by GNU-compatible compilers.
- `pragma HLS unroll` controls unrolling for the annotated loop body. This annotation can enable or disable unrolling and specify the unrolling *factor* for the loop. For example, a loop annotated with `pragma HLS unroll factor=4` will be unrolled (*i.e.*, its body will be replicated, and the loop trip count adjusted accordingly) four times.
- `pragma HLS dependence` allows users to enforce or disable handling a particular data dependence inside a loop's body. HLS compilers perform static analysis to detect whether dependencies exist between memory accesses in the input program. However, precise alias analysis in an undecidable problem [308], so the user can use this annotation to give hints to the optimizer about the presence or absence of a RAW, WAR, or WAW memory dependency.
- `pragma HLS loop_flatten` and `loop_merge` annotate loops that must be flattened or merged by the compiler before trying to schedule their bodies. Loop flattening [128] is a standard optimization compilers use to transform a *loop nest* into a single loop that performs the same computation. Loop merging can be used to force the merging of loops inside the input program. In the example of Figure 3.7, annotating loops ❷ and ❸ with this `pragma` would allow the HLS compiler to merge the two corresponding datapaths to reduce the final design's surface area.
- `pragma HLS array_partition` and `array_reshape` perform transformations on the data layout of arrays in the input code. Code may sometimes be easier to read and maintain when using a single array of a given shape, but using large arrays is often not the best way to access data in hardware. The `array_partition` pragma allows users to split an array into multiple sub-arrays that can subsequently be stored in separate memory banks. The HLS compiler transforms array accesses into the initial array to access the memory bank associated with the corresponding sub-range of addresses manipulated by the program. Similarly, `array_reshape` changes the *shape* of the array by reorganizing its elements to match a specified data layout. This data layout change can allow for more efficient data access patterns, like interleaved accesses to multiple memory

```
void add_round_key(int8_t n, uint8_t *result, uint8_t *state, uint8_t key[240]) {
  #pragma HLS array_partition variable=key type=complete
  for(int8_t i = 0; i < 4; ++i) {
    #pragma HLS unroll factor=4 skip_exit_check
    for(int8_t j = 0; j < 4; ++j) {
      #pragma HLS unroll factor=4 skip_exit_check
      result[i * 4 + j] = state[i * 4 + j] ^ key[n * 16 + i * 4 + j];
    }
  }
}
```

Figure 3.9 – Annotated HLS code for the AES-256 `add_round_key` function.

banks.

— `pragma HLS latency` serves as a hint to the scheduler indicating the expected execution latency of a function, a loop, or a code block. If the HLS compiler manages to produce a datapath whose execution latency matches the constraints set by this annotation, then it will not try to produce a potentially more efficient schedule.

— `pragma HLS pipeline` annotates a loop with a target *initiation interval* (II) value. This value defines the maximum number of cycles separating the start of two consecutive loop iterations. The HLS compiler will try to overlap the execution of multiple loop iterations to achieve the desired value of II. If the latter is not achievable, it may fail with an error or provide a best-effort design with the closest achievable II. As demonstrated in Section 2.1.3, lower values of II produce faster circuits.

The need for HLS annotations. Typical High-Level Synthesis code relies heavily on user annotations to guide the synthesis process toward efficient results, be it in terms of performance, surface area, or energy consumption. While the need for such annotations may question the utility of HLS compared to standard HDLs, remember that the entire HLS design process is often faster by at least an order of magnitude!

Figure 3.9 shows an annotated example of the 256-bit AES `add_round_key` function from Figure 3.8. The `array_partition` annotation makes sure that the `key` array is stored using individual registers for fast access (it is *completely* partitioned). The two `unroll` annotations fully unroll the inner loops.

3.4.4 Fertile Ground for New Research

High-Level Synthesis has been successfully applied in both academic and industrial settings to synthesize complex hardware accelerators [79]. It has seen wide adoption in fields such as machine learning [412, 348, 137, 326, 23, 420, 369, 384, 401, 403, 62, 102, 417, 349, 251, 59, 297], video and image processing [374, 76, 71, 234], graph analytics [61], networking [375], bio-informatics [60, 243, 353, 140, 235, 408, 78], and scientific computing [391, 422, 221, 423, 382, 342, 222]. The productivity and iteration time gains provided by HLS hardware design flows, as well as the continuous QoR (*Quality of Results*) improvements delivered by new optimization techniques, has made them a viable alternative to standard HDLs in many settings. Infrastructure-as-a-Service (IaaS) providers such as Amazon AWS [14] even include FPGA-based nodes in their offerings for cloud-based applications [387].

HLS sits at the crossroads between hardware and compiler design, which allows it to benefit from advances in both compilation/synthesis optimizations [267, 271, 122, 213, 79], and hardware technology [394]. This unique position makes the field of HLS an active research area, with many subsisting research questions and challenges [79]. Among the latter are (i) the challenges in user interface design

14. https://aws.amazon.com/

and hardware synthesis control, (ii) the lack of open-source and standardized toolchains, and (iii) the difficulty of porting CPU-oriented applications to HLS design flows.

The typical way for HLS users to interact with the hardware synthesis engine is through code annotations, as described in Section 3.4.3. These annotations are required since the C and C++ languages were not designed for direct hardware mapping. Designing new languages specifically targeted at HLS design flows allows language designers to integrate hardware-related semantics directly into syntactical or library constructs. This approach has been successfully applied by novel languages such as Spatial [200] and Kanagawa [288]. The latter has been used at Microsoft to design data-center scale hardware accelerators. It relies on the fundamental concepts of *hardware threads* and *wavefront consistency*. The former are a mapping from the software concept of execution threads to hardware: they represent a sequence of operations executed sequentially and independently of other threads (with some occasional synchronization points). This programming model is coupled to a consistency model that ensures, through the compiler and the execution model, that access to shared resources remains ordered during the entire execution of a thread group. Wavefront consistency ensures that accesses to shared state made by a given hardware thread happen in program order and that if thread t_1 executes ahead of thread t_2, then all accesses to these shared resources by t_1 will happen before the corresponding accesses for t_2. Kanagawa offers expressive language constructs that allow hardware designers to design libraries of reusable components. The language is designed to give designers control over hardware concurrency instead of letting the synthesis flow infer parallelism from sequential language constructs.

While designing new HLS-oriented languages may be viable for hardware designed from the ground up, it may not be suitable for all HLS-related tasks. In particular, many application domains that want to accelerate computations using hardware accelerators already rely on well-established codebases and libraries. It does not seem reasonable to translate all of this existing code to a new programming paradigm to benefit from hardware acceleration [79]. Additionally, HLS can be used as a backend for domain-specific compilers that could benefit from hardware acceleration, making it possible to produce accelerators that would have been too expensive to produce without such a high-level design process. Consequently, there is a real need for tools and techniques that ease the transition between CPU-centric code and FPGA-optimized implementations. Source-to-source transformation frameworks have been proposed to address this challenge [124, 219]. By increasing the amount of automated hardware-specific optimizations that can be performed by the HLS compiler [84, 103], such techniques can be used to bridge the gap between software designed to run on classical computers and hardware-accelerated high-performance implementations [368, 97, 133, 119, 131, 118]. Our work focuses on enabling high-performance hardware accelerator design using source-to-source transformations for High-Level Synthesis. We demonstrate how these techniques can be used to synthesize speculative hardware from a sequential algorithmic specification (Chapter 5 and 6), and we apply them to the synthesis of RISC-V processors from instruction set simulators (Chapter 7). The flexibility of HLS and the familiarity of its input language to software developers make it a good target for bridging the gap between software and hardware design concerns.

Another significant challenge for the widespread adoption of High-Level Synthesis is the lack of standardized and open-source toolchains. While recent works have led to the creation of some open-source HLS design flows such as LegUp [15] [50], Bambu [117], Dynamatic [187], and ScaleHLS [404], these tools cannot compete with vendor-supplied compilers when targeting their hardware. Timing and hardware layout information are often under-specified in vendor documentation, making it hard to develop HLS toolchains with accurate timing estimates on most common FPGA boards. Additionally, modern high-performance platforms require intimate knowledge of their implementation to exploit all of their resources best. Despite the reverse engineering efforts deployed by several communities to make the development of FPGA tools more open (see Section 2.4.2), such high-end platforms are still far out of reach of reverse engineers. Some vendors are starting to open-source parts of their design flows [16], a step in the right direction for future HLS tool development.

15. LegUp was since turned into a commercial product by its original authors through LegUp Computing Inc. The company was acquired by Microchip Technology Inc. in 2021 (https://www.microchip.com/en-us/about/news-releases/products/microchip-acquires-high-level-synthesis-tool-provider-legup).

16. https://github.com/Xilinx/HLS

Future perspectives for HLS. HLS is a very active research area. New optimizations and higher-level programming languages also broaden the scope of designs suitable for HLS-based hardware design techniques by lifting the limitations of existing toolchains. The interested reader may find an excellent rendition of High-Level Synthesis' successes, challenges, and opportunities, along with some valuable insights, in [79].

3.5 Processor Design Automation

Processors are often considered an advanced topic in hardware design [332]. They are intricate pieces of electronics that require the careful design and implementation of multiple interacting components. The high-performance instruction set processors described in Section 2.3 have deservedly won a place on the list of humanity's most advanced technological achievements. Fortunately, not all processors require the level of technological expertise needed to craft the most high-performance chips: embedded processors often rely on simpler designs, trading off execution speed for lower energy consumption, surface area, and reduced production costs. Designing all but the simplest of processor cores is nonetheless a non-trivial task, making it a perfect fit for automation techniques.

3.5.1 Designing Processors With Hardware Description Languages

The history of processor design closely follows that of EDA since the relative complexity of processors makes them an excellent target for design automation. Like most digital electronics today, processors are primarily designed using hardware description languages such as VHDL or Verilog [160, 154]. Some examples of open-source processor cores designed using standard HDLs include the CV32E40P [126] and CVA6 [411] RISC-V cores from the OpenHW Group [17]. Despite the prominence of standard HDLs in processor design, newer hardware description and construction languages discussed in Section 3.2 are increasingly being used to design processors. BERI (Bluespec Extensible RISC Implementation) [388] is a RISC instruction set processor designed at the University of Cambridge and implemented using Bluespec SystemVerilog (Section 3.2.1). It is used as a teaching and research platform and was the initial implementation target for the CHERI capabilities system [396].

Designing instruction set processors comes with unique challenges, which sets them apart from many other digital electronics circuits. Processors must offer a flexible programming interface, making them well-suited for highly irregular and heterogeneous workloads. The latter are inherent to desktop, mobile, and high-performance computing, where most applications are control-dominated. On the other hand, in the embedded and IoT spaces, typical computing scenarios usually involve little variability and control but focus on processing large quantities of data. Instead of targeting programmability and flexibility, most applications in these domains focus on special-purpose hardware accelerators (see Section 2.2) designed to reduce power consumption and increase performance on a single well-defined set of tasks (e.g., signal processing, video encoding, and decoding).

However, the trend in specialized computational domains is slowly shifting. New applications such as data mining, graph analytics, and machine learning introduce new computational needs. The latter require special-purpose hardware to provide a programmable interface and to operate on control-flow-dominated workloads. Addressing these new requirements challenges hardware manufacturers to design programmable components, such as ISPs, with many custom features while still providing fast processing times and reduced energy consumption. The design of such custom ISPs is a tedious and error-prone process that must be conducted carefully to prevent the hardware from misbehaving once it is produced.

The challenges posed by processor design and the growing need for customizable instruction set processors have prompted the development of a different class of languages. The latter, which are presented in the next section, are designed around issues inherent to processor design but seldom occur in other electronics design endeavors.

17. https://github.com/openhwgroup

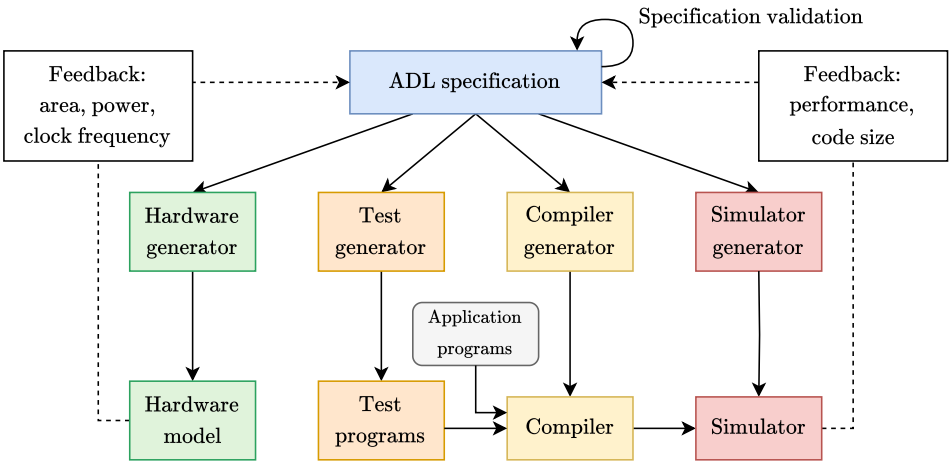


Figure 3.10 – Processor design using an ADL. The ADL specification can be used to generate hardware models, tests, compilers, and simulators that all contribute to a feedback loop that can help improve said specification. Adapted from [259] and [260].

3.5.2 Architecture Description Languages

The standard Hardware Description Languages described in Section 3.1 as well as the higher-level hardware languages presented in Section 3.2 are all well-suited to describe low-level hardware components. However, they are a poor fit to describe the high-level structure of a processor's architecture. Languages designed specifically for design work related to processors can capture more relevant information to automate tasks specific to architecture and micro-architecture design. This section discusses such languages, which are often referred to as *Architecture Description Languages* (ADL), or *Processor Description Languages* (PDL).

An Overview of Architecture Description Languages

Architecture Description Languages are specialized languages that describe the structure and behavior of instruction set processors. Contrary to HDLs, they are not destined for general-purpose hardware design. They include a variety of processor-specific constructs that can be used to describe a processor's architecture and microarchitecture. ADLs were first introduced as tools for building retargetable compilers [30, 301]. These compilers are designed to be easily adapted to a new instruction set architecture, often providing backend-agnostic optimizations on top of a target-specific backend. Most of today's most successful compilers are built on such a model. Processor specifications described using an ADL can be used to automate the generation of assemblers and instruction timing models for a target platform, which can help derive significant parts of a retargetable compiler's backend.

The early 2000s saw a rising interest in Application-Specific Instruction Set Processors (ASIP), which prompted the development of new ADLs with a more general focus than retargetable compilation [300, 259]. These languages quickly became central to the design of many embedded ASIP cores by providing automated generation capabilities for development environments, simulators, and hardware models. Later sections give more details on these early ASIP design flows. Figure 3.10 illustrates the interactions between all these components. Only the ADL specification needs to be created and updated manually: the hardware, test, compiler, and simulator generators can often be automatically derived from the latter. The hardware model and the simulator contribute to a feedback loop that improves the processor specification by providing information about area, power, clock frequency, performance, and code size to the designer.

In the following sections, we focus on the hardware synthesis part of ADLs. We follow a top-down approach, starting with ADLs built to express ISA semantics and aimed at automating the synthesis

```
storage r
  width 32
  size  4
end

instruction add
  encoding "00ddnnii"
  semantics begin
    r[d] = add(r[n], exts(i));
  end
end
```

Figure 3.11 – ViDL specification of a simple ADD instruction. The register file is specified as a storage element containing four 32-bit elements. The instruction's encoding and semantics are given by the `instruction` block. Adapted from [100].

of processor microarchitecture. We then describe ADLs that describe lower-level hardware structures and are targeted toward automatically generating specific microarchitectural features for pipelined processors. We end this section with a comparison of the features of a large number of ADLs.

Micro-Architecture Synthesis from Instruction Set Specifications

The ISA defines the high-level behavior of a processor. However, a given architecture can lead to various micro-architectural choices that will influence the final design's performance, cost, and energy consumption. For example, designers may want to integrate a *forwarding* mechanism into their core, which will bypass certain pipeline stages for pipeline hazard handling purposes. However, these mechanisms may have a surface area and energy consumption overhead that may not be suited for the target application. The different trade-offs for a given hardware component form what is known as its *design space*. Quickly exploring this design space and selecting micro-architectural designs that fit the application at hand is key to achieving high-quality processor designs at scale. Several techniques have been proposed to generate multiple micro-architectures from a single instruction set architecture specification automatically [99, 230]. These techniques rely on Architecture Description Languages to define the high-level features of the architecture.

ViDL (Versatile Instruction Set Architecture Description Language) [100] is a language designed to specify instruction sets formally. A snippet of ViDL code is shown in Figure 3.11. The language provides high-level concepts that allow designers to quickly specify the semantics of individual instructions, along with their precise bit-level encoding. The `semantics` block accepts a subset of Standard ML (SML) [257] to allow potentially complex instructions to be described expressively. ViDL is coupled to a processor generator [99]. The latter parses the instruction set description and, using a target clock frequency indicated by the user generates multiple VHDL implementations of the same processor with different micro-architectural trade-offs. The generator operates on a data-flow graph (DFG) representation that describes the interaction of instructions with storage components and I/O ports. It then automatically splits operations into multiple pipeline stages, schedules their execution, inserts pipeline registers to define the processor's pipeline stages, and infers possible forwarding and interlocking logic. The generated designs include a simple branch prediction scheme that predicts all branches to be *not-taken*. ViDL and its accompanying processor generator have been successfully used to synthesize pipelined processor micro-architectures for a wide variety of ISAs, including ARM, MIPS, and Power. However, the project seems to have been abandoned since the early 2010s, and little information is available on the internals of ViDL and its supporting infrastructure.

ASSIST [230] is an ADL embedded in Python with similar goals to ViDL, focusing on automatic RISC-V processor generation. Figure 3.12 gives an overview of the specification of a few RISC-V in-

```
# Integer addition.
def execute_add():
    create_inst('ADD'); tmp = add(rs1, rs2)
    assign(tmp, rd); inc_pc()

# Branch-if-equal.
def execute_beq():
    create_inst('BEQ'); tmp = cmp_eq(rs1, rs2)
    update_pc_with_pred(tmp, pc, imm_b)

# Load byte.
def execute_lb():
    create_inst('LB'); address = add(rs1, imm_i)
    mem_read(address, 1, rd, SIGNED); inc_pc()
```

Figure 3.12 – RISC-V instruction specifications using ASSIST. This snippet specifies the behavior of three RISC-V instructions: `add`, `beq`, and `lb`. Adapted from [230].

structions using ASSIST. `rs1`, `rs2`, `imm_b`, `imm_i`, and `rd` are user-defined fields that represent parts of the instruction's encoding. ASSIST lets users combine such custom elements with *micro-operations* provided by the language to interact with the architectural state of the processor (*e.g.*, `mem_read` to read from memory, `assign` to update the register file, `inc_pc` and `update_pc_with_pred` to update the value of the program counter (PC)). The high-level instruction set specification is fed to an *architectural synthesis engine* that automatically infers a pipeline structure for the target architecture. The synthesis engine generates a Chisel pipeline description that can then be translated to Verilog. The Chisel description and its corresponding HDL implementation are used to perform simulations to obtain performance numbers and area and timing results for the generated micro-architectures. Those results are then fed to an *autotuner* that provides a feedback loop to improve the generated design iteratively. The automated pipeline exploration using the autotuner allows ASSIST to produce hardware designs that are competitive with manually optimized processor pipelines. While ASSIST can drastically improve the design process of ISPs based on the RISC-V ISA, designers are limited by the basic building blocks provided by the language in their hardware descriptions. A major limitation of approaches like ASSIST is their introduction of only a few high-level building blocks that increase expressivity at the expense of control over the generated micro-architecture.

The rise of soft processor cores. The advent of new processor design tools and languages and the ubiquity of reconfigurable hardware in the form of FPGAs has led to the emergence of numerous *soft processor cores*. These cores are designed to be implemented on FPGA platforms. They can provide good levels of performance for some embedded applications while retaining the flexibility of FPGAs for easier integration into prototypes and low-production-volume designs [405].

Pipeline Design Automation

The first approaches aimed at automating the design of ISPs were proposed in the context of ASIP design flows [73, 162, 196, 372, 247, 58, 260, 198, 327, 274, 16, 212, 134]. ASIPs are programmable processing cores targeting a particular application domain. As a result of their specialized nature, the ISA of ASIPs is tailored to the application, exposing specially crafted instructions and focusing on a small set of tasks. This domain-specific approach is similar to some of the more advanced hardware accelerators. ASIPs can be seen as an early middle-ground between specialized co-processors and general-purpose instruction set processors. Hardware evolution has led most ASIP-related design work to be spread between modern instruction set processors, which often include specialized instruction for

domain-specific computation acceleration in their ISA, and hardware accelerators, such as the GPUs and TPUs discussed in Section 2.2.

Early works have proposed automated hazard-resolution tools based on hardware/software co-design techniques [73, 162]. The *Snapshot Method* [73] was developed to automate parts of the synthesis of pipelined instruction set processors by considering possible inter-instruction dependencies during the development phase. It considers sequences of instructions that could be in flight in the pipeline. It iteratively determines if the pipeline structure needs to be adapted to make the execution of the current sequence more efficient. This design methodology was successfully applied to the synthesis of pipelined designs for two different architectures, Mark1 and ROMP [338]. The Snapshot Method allows for iterative refinement of the pipeline structure by defining the number of stages and solving some structural hazards by adding more hardware where necessary.

Later works [205] focused on the automation of another kind of hazard management technique, namely *interlocking* and *forwarding*. The former is a mechanism by which the overlapped execution of instructions in a pipelined processor is partially halted to prevent structural hazards: part of the instruction stream's execution is made sequential until the structural hazard is resolved. The interlocking machinery is built on top of a *stall engine* that stalls (*i.e.*, pauses) the execution of certain pipeline stages while subsequent stages continue their execution. *Forwarding* is a data hazard management technique that enables bypassing some registers in the pipeline to forward values from the output of execution units to their inputs. Both interlocking and forwarding are crucial to memory access handling in pipelined hardware designs and are discussed in Chapter 7.

The pipelined execution model easily fits into the transactional model introduced with TL-Verilog in Section 3.1.3. T-spec and T-piper [272] build on this observation and provide a high-level specification language and synthesis tool to synthesize processor pipelines from a transactional behavior description automatically. T-spec is a language that can be used to define the connections between hardware modules and their interactions using a transactional semantic. Instead of mapping the execution of hardware components to clock cycles, transactions are executed in an abstract domain that decouples the processor's datapath specification from the actual implementation. Hardware designers only need to specify the transactions, and the T-piper synthesis tool will schedule them for the target implementation. A single transaction can, therefore, be mapped to multiple clock cycles. T-spec and T-piper provide an iterative design methodology, allowing hardware designers to enable forwarding in certain pipeline stages selectively. The transactional view of pipeline synthesis also emerges when considering synchronous elastic (*i.e.* latency tolerant) circuits to build processor pipelines [191]. Elastic circuits and their unique scheduling properties are discussed in Chapter 4.

Recent work [409] proposed a novel pipeline description language, PDL, that provides abstractions for micro-architectural features such as pipeline stages, data hazard resolution, and even *speculative execution*. PDL is an imperative-style programming language in which hardware designers are only required to specify the behavior of their processors. They can then annotate this behavioral description with micro-architectural primitives that control the insertion of hazard mitigation logic to increase the performance of their designs. The concurrent execution nature of processor pipelines is modeled using *threads*, whose semantics are similar to software threads executing in parallel. To model the 5-stage pipeline described in Figure 2.10, a new thread would be launched at the end of the IF stage to start processing the next instruction. Hazard management is built around a *lock* abstraction, similar to software locks in multi-threaded programs. This similarity also appears in some High-Level Synthesis toolchains [288].

The abstraction level provided by automated pipeline design tools is often very close to the one offered by standard HDLs and asks for explicit management of micro-architectural choices such as pipeline depth, organization, and available data forwarding points. While the approaches described in this section give a great level of control to hardware designers, they can become cumbersome when exploring the design space available for a given instruction set architecture.

Table 3.2 – Comparison of Architecture Description Languages. Check marks indicate full support in the language; parenthesized check marks indicate only partial support for a given feature.

ADL Name	ISA info.	μ-arch. info.	Comp. gen.	Sim. gen.	C.-a. sim.	Formal verif.	Test gen.	Hardware gen.
ADL++ [274]	✓	✓	(✓)	✓				
AIDL [263]		✓		✓	(✓)	(✓)		(✓)
ArchC [318, 24]	✓	✓		✓	✓	✓		✓
ASSIST [230]	✓			(✓)	(✓)			✓
CSDL [310, 26]	✓	✓		✓	✓	✓	✓	✓
CodAL [370]	✓	✓	✓	✓	✓	✓	✓	✓
CoreDSL [275]	✓	(✓)	✓	✓		✓		✓
EXPRESSION [260]	✓	✓	✓	✓	✓	(✓)	✓	✓
FlexWare [286]	✓	✓		(✓)	✓			✓
GNR [134]		✓		✓	✓	✓		✓
HMDES [27]	✓	✓	✓	✓	✓			
ISDL [147, 148, 27]	✓		✓	✓	✓			(✓)
LISA [58]	✓	✓	✓	✓	✓			✓
MADL [302]	✓	✓	(✓)	✓	✓			
MAML [212]	✓	✓	✓	✓	✓			✓
Maril [45]	✓		✓					
MIMOLA [247]		✓	✓	✓	✓		✓	✓
nML [372]	✓		✓	✓			✓	
PDL [409]		✓				✓		✓
PRMDL [356]	✓	✓	✓	✓	✓			
RADL [339, 27]	✓	✓		✓	✓			
SAIL [135]	✓			✓	✓	✓	✓	
TDL [192, 27]	✓		✓					
TIE [327]	✓	✓					✓	✓
T-spec/T-piper [272]		✓						✓
UDL/I [27]		✓	✓	✓	(✓)			✓
VADL [156]	✓	✓	✓	✓	✓		✓	✓
Valen-C [169, 27]	✓	(✓)		✓				
ViDL [100, 99]	✓			✓	✓			✓

Comparison of Architecture Description Languages

Architecture Description Languages make instruction set processor design less cumbersome by automating large parts of the hardware design process. Many ADLs also offer automated compiler, simulator, and test generation, which can provide helpful feedback during a processor's design phase. The past decades have seen much development around ADLs, some focusing on specific areas of the ISP design process while others try to provide all the required tools out of the box.

Table 3.2 gives an overview of the feature differences between several ADLs. The successive columns indicate whether (i) the language incorporates information about the processor's ISA, (ii) the language includes micro-architectural feature descriptions, (iii) a compiler for the described architecture can be automatically generated, (iv) an architecture simulator can be generated, (v) a *cycle-accurate* simulator can be generated, (vi) formal verification capabilities are built into the language or its surrounding framework, (vii) tests can be automatically synthesized from the processor description, and (viii) whether hardware can be generated from this description.

While most ADLs target hardware generation of instruction set processors, many such languages are also aimed at automatically generating compilers, simulators, and functional validators for the architecture they describe. This ecosystem of tools is crucial for hardware designers looking to describe

their processor cores using such languages since simulation and verification are paramount to successful complex ISP design. Most of the languages listed in Table 3.2 use a combination of architectural information about the ISA and micro-architectural information. This combination enables the automatic generation of optimizing compilers for the target architecture (*e.g.*, through the use of custom instruction schedulers and register allocators), as well as accurate processor simulators. Many ADLs offer the possibility of generating cycle-accurate simulators, which provide exact behavior information down to each clock cycle. Some languages even provide automated facilities for generating micro-architectural tests and formal verification tools to validate the behavior of the generated hardware. We note that CodAL [370] is the only ADL that checks all of the features in the table. However, most public information about this commercial language is derived from promotional material, making it hard to evaluate the extent to which each feature is supported.

While architecture Description Languages offer valuable tools to automate parts of the processor design process, they often fall short in hardware generation capabilities. As described in previous sections, ADLs capable of hardware synthesis are often HDLs in disguise, offering only limited abstraction and design space exploration capabilities to hardware designers, and making them inherently hard to use. High-level ADLs that support hardware generation, on the other hand, often constrain their users to the architectural and micro-architectural primitives provided by the language. This lack of extensibility can hinder the exploration of advanced micro-architectural features. In this thesis, we propose a shift in focus towards automating the synthesis of ISPs using High-Level Synthesis. We demonstrate that this approach strikes a good balance between expressivity, ease of use, and design space exploration capabilities, addressing the limitations of current ADLs.

CHAPTER 4

Scheduling in High-Level Synthesis

Time is not a straight line. It's all bumpy-wumpy and timey-wimey.
— *The Doctor* (Doctor Who)

High-Level Synthesis allows users to design hardware using familiar high-level languages, which can greatly benefit the productivity of hardware designers. It also contributes to reduced iteration time and faster design space exploration, which makes it a perfect fit for discovering new hardware design patterns and trade-offs. The key step in the High-Level Synthesis process is *scheduling*. During this phase, the hardware synthesis engine weaves the notion of timed execution in the hardware representation. It determines when operations should start executing relative to one another, ensuring that data dependencies are not violated and that hardware is used to its full potential. This chapter discusses efficient scheduling for High-Level Synthesis, which is a fundamental problem that prompts new research ideas on a regular basis. Section 4.1 discusses static scheduling techniques employed in modern commercial HLS toolchains. Section 4.2 presents *dynamic* scheduling techniques, which allow for higher-performance hardware designs by exploiting the dynamic nature of some workloads. Finally, Section 4.3 explores *speculative* scheduling, a technique that introduces speculative execution capabilities to the scheduling process to maximize performance and resource utilization.

4.1 Static Scheduling and Loop Pipelining

High-Level Synthesis toolchains must infer hardware structure from user input code during the synthesis process. Similarly to compilers for a typical programming language, this usage pattern implies that the synthesis process can only rely on information known at *compile time* to generate the target hardware and especially to schedule its execution. This reliance on compile time (*i.e.*, statically-known) information for scheduling is known as *static scheduling*. Most commercial tools use static scheduling and *loop pipelining* to synthesize pipelined hardware from an algorithmic description.

4.1.1 Loop Pipelining

Designing efficient accelerators using only statically known information requires HLS toolchains to uncover as much parallelism as possible from the input code. The core of the parallelism-uncovering mechanisms in modern HLS flows is centered around one particular optimization pass, known as *loop pipelining*. Given a target execution frequency and the corresponding execution cycle estimates for standard operators, loop pipelining transforms a sequential loop iteration into a set of independent operations that the hardware can execute concurrently to speed up execution. To better understand

```
1  void vec_mul(int *v1, int *v2, int *out, int n) {
2    for (int i = 0; i < n; ++i)
3      out[i] += v1[i] * v2[i];
4  }
```

Figure 4.1 – Sample data-processing code. The two `n`-sized vectors `v1` and `v2` are multiplied component-wise, and added to the output vector `out`.

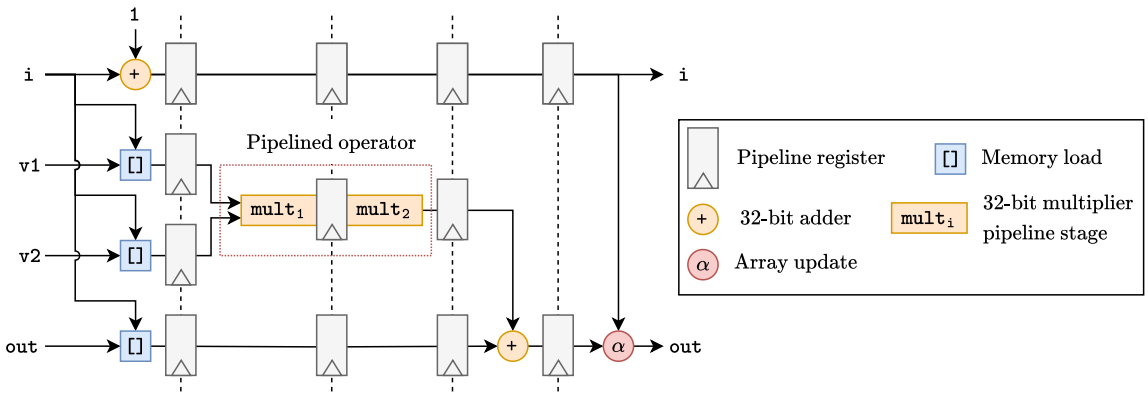


Figure 4.2 – Pipelined datapath for the code in Figure 4.1. Each operator executes in one clock cycle, except for the 32-bit multiplier. The latter is divided into two pipeline stages. The α operator symbolizes the update to the `out` array.

how loop pipelining operates, consider the example code in Figure 4.1. This code is typical of data-processing applications that are well-suited for hardware acceleration: it performs a regular operation on a potentially large dataset and does not require complex control-flow handling.

The datapath inferred by a typical HLS toolchain from this input code for a target execution frequency f is given in Figure 4.2. This datapath is organized in multiple stages that are delimited by pipeline registers. The loop pipelining pass determines these stages' boundaries and their contents. Static analysis of data dependencies between operators and static cycle time estimates are combined to produce this hardware structure, which will be piloted by a Finite State Machine (see Section 3.4). The output schedule of the loop pipelining pass applied to the code in Figure 4.1 is depicted in Figure 4.3. The horizontal axis represents execution cycles, while the vertical axis corresponds to pipeline stages. In this case, the static schedule produced by loop pipelining is optimal: it produces the fastest possible hardware implementation at frequency f, with all operators fully utilized at all times once the pipeline is filled with data. The pipelined schedules produced by loop pipelining are characterized by their *latency* Δ, *i.e.*, the number of cycles required to execute one loop iteration, and their *initiation interval* II, which corresponds to the number of cycles that separate the start of two successive loop iterations. In our example, we have $\Delta = 5$ cycles, and $\text{II} = 1$ cycle.

Scheduling constraints. The datapath organization shown in Figure 4.2 is one of many possible configurations of a circuit that computes the result of the code in Figure 4.1. The schedule for a given hardware design is always computed under a set of constraints, be it the target II, the amount of available hardware resources, or the expected energy consumption of the final design.

Loop pipelining is the HLS counterpart of another optimization technique commonly used in compilers, known as *software pipelining*. Software pipelining aims to exploit instruction-level parallelism in processors by organizing instructions to maximize the potential for parallel execution in hardware. Software pipelining is implemented using a technique known as *modulo scheduling* [312].

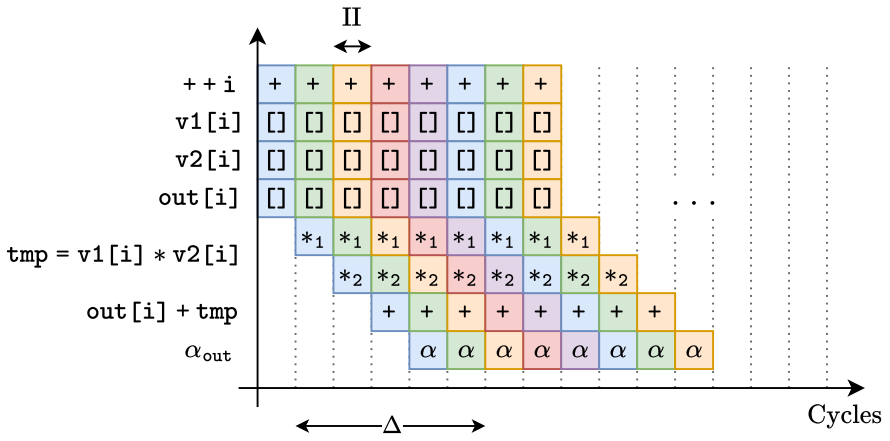


Figure 4.3 – Static schedule inferred from the code in Figure 4.1. The schedule inferred by loop pipelining is optimal, leading to the highest possible performance.

4.1.2 Modulo Scheduling

Modulo scheduling was introduced in 1981 to automate code generation for a custom scientific computing architecture [312]. Monica Lam then extended this technique and showed that it was applicable to general-purpose computing VLIW (Very Long Instruction Word) architectures in 1988 [215]. The introduction of modulo scheduling kickstarted a wave of research on automatic parallelism extraction in compilers for software programming languages. Early compilers used branch-and-bound approaches with special-purpose heuristics to prune the search space of all possible modulo schedules. At the same time, research work was underway to give an optimal formulation of the modulo scheduling problem in terms of Integer Linear Programming (ILP) [314, 324].

The main idea of modulo scheduling is to try to maximize the overlap between successive loop iterations, which are set to start execution at a fixed interval (the *initiation interval*, II). Modulo scheduling tries to extract a pattern from the unrolled loop's body that can be efficiently scheduled given time and execution resource constraints. This pattern is then repeatedly executed, and a prologue/epilogue pair is inserted into the execution stream to adjust for potential initial or trailing iterations [194]. A modulo scheduler needs to take two different sets of constraints into account: the available computing resources that can be used in parallel (*e.g.*, the number of integer arithmetic units in a processor), as well as the intra-and inter-iteration dependencies that can exist between operations. The first phase of modulo scheduling computes the *resource-constrained II*, or ResII. This value is computed by considering only the available computing resources and ignoring potential data dependencies between loop iterations. Then, the *recurrence constrained II*, RecII, is computed by considering loop-carried dependencies. The latter constrain the final schedule since RAW, WAR, and WAW dependencies need to be honored. The resulting initiation interval pair is used to determine the minimum II, MII, such that

$$\text{MII} = \max(\text{ResII}, \text{RecII}).$$

Once the modulo scheduling algorithm has determined the minimum initiation interval value, iterations can be scheduled to execute at regular intervals of at least MII cycles. Considering resources and data dependencies, this scheduling technique ensures that the resulting schedule cannot cause structural or data hazards (see Section 2.3.2). Optimization techniques that target HLS often try to improve the value of ResII while considering RecII as a fixed value computed by the compiler. Our work takes on a different perspective since we try to reduce RecII through speculative execution in HLS (see Section 4.3.4 and Chapter 5).

Most state-of-the-art modulo schedulers, both for software programming languages and for High-Level Synthesis, rely on an iterative approach to modulo scheduling. Bantwal Ramakrishna "Bob" Rau

```
1  int vec_filtered(int *v1, int *v2, int n) {
2    int result = 0;
3    for (int i = 0; i < n; ++i) {
4      if (filter(result))
5        result += v1[i] * v2[i];
6    }
7    return result;
8  }
```

Figure 4.4 – Sample data-processing code with a conditional. The two `n`-sized vectors `v1` and `v2` are multiplied component-wise, and the results are accumulated into `result`.

proposed the first iterative approach to this scheduling problem in 1994 [313]. ILP-based schedulers represent the state-of-the-art techniques in this field and can be used to obtain optimal schedules w.r.t. latency [276, 277, 344]. Optimal schedules can sometimes be challenging to compute, leading to increased computation times for problems that may not require optimal schedules to satisfy their design constraints (*e.g.*, production cost, or energy efficiency). Heuristics-based approaches built on top of System of Difference Constraints (SDC) formulations [77] can be used to reduce the modulo scheduling computation time [418, 49]. An SDC problem can be solved in polynomial time w.r.t. the problem size [77]. The SDC form of the HLS scheduling problem can be formulated as follows.

$$\text{Minimize} \sum_{i \in I} t_i \quad \text{subject to } t_i - t_j \leqslant -D_i + b_{i,j}\text{II},$$

where t_i is the starting cycle of operation i, D_i is its latency, and $b_{i,j}$ is a measure of intra-loop dependencies between instructions i and j [344]. Such SDC-based approaches have been successfully integrated into HLS toolchains that can compete with ILP-based schedulers when determining a schedule for a target II value [276].

Rational II values. While modulo scheduling techniques rely on initiation interval values expressed as integer cycle counts, there may be some benefit to considering *rational* II values. The latter have been the subject of several recent works that demonstrate the use of rational II in modulo scheduling to produce designs with even higher throughput than regular integer-based approaches. While these works are outside the scope of this thesis, we encourage the reader to take a look at [340, 121, 120].

4.1.3 Limits of Static Loop Pipelining

Static loop pipelining based on modulo scheduling can produce efficient schedules for regular loops. However, introducing control flow inside the loop body can render such schedules sub-optimal in most cases. Figure 4.4 shows a modified version of the code in Figure 4.1 that includes a conditional statement inside the loop body. Since the value of `filter` cannot be known at synthesis time, the static schedule needs to take the worst case into account in order to avoid creating any hazards during execution. Consequently, the loop needs to be scheduled *as-if* the `filter` function always returns `true`. The resulting schedule, where we assume the execution of `filter` takes one clock cycle, is shown in Figure 4.5. While some operations can be overlapped inside of a single loop iteration, the conditional loop-carried dependency on the `result` variable renders any attempt at overlapping the execution of two successive iterations impossible. Consequently, the best achievable II when statically scheduling this loop is $\text{II} = 4$.

The remainder of this chapter focuses on possible improvements to static techniques when scheduling programs with irregular control flow, such as the one depicted in Figure 4.4. In order to remove any

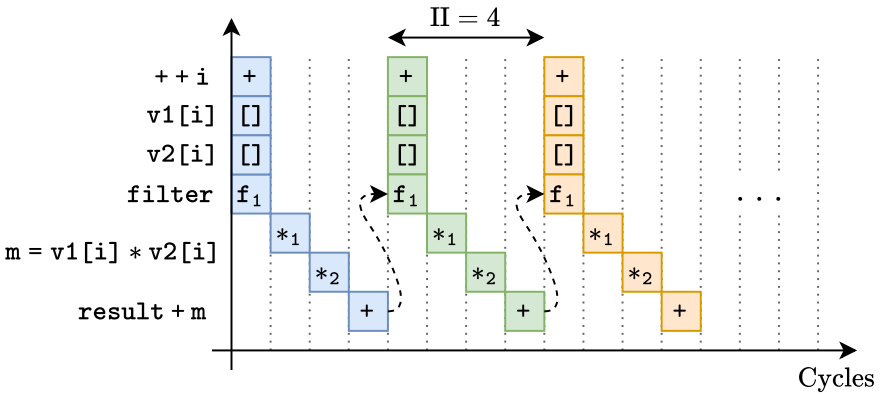


Figure 4.5 – Static schedule inferred from the code in Figure 4.4. Since the return value of `filter` cannot be known statically, the static schedule needs to consider the worst case. Dotted arrows represent conditional loop-carried data dependencies.

```
do {
  y = next(y);
  if (test(x, y))
    x = f(x);
  else
    x = g(x);
} while(!x);
```

Figure 4.6 – Sample data-processing code. This snippet will be used throughout this chapter to illustrate various HLS scheduling techniques.

potential noise and provide a generic example, we will use the code snippet given in Figure 4.6 (which readers may recall from Chapter 3) for further discussions. This example is typical of applications that are difficult to handle for standard HLS tools when any of the functions take more than a single clock cycle to execute. Figure 4.7 illustrates the schedule produced by static techniques for this input program if we assume that `g` takes three cycles to execute. In this case, II = 3 and $\Delta = 4$ cycles. Similarly to the example in Figure 4.5, conditional loop-carried dependencies constrain the schedule, which needs to consider the worst possible execution scenario for each loop iteration.

In some cases, control flow in loops can be eliminated by introducing predication [9, 282]. The resulting loop body can then be scheduled using the modulo scheduling techniques discussed in this section. In the presence of nested branches, more complex predication schemes based on hierarchical predicates or predicate matrices can be used [363, 241, 240, 256]. However, many emerging applications such as graph analytics, network packet processing, or workloads operating on sparse data structures, heavily rely on control-flow dominated implementations that are not always amenable to predication. The dynamic nature of the computation patterns involved in such workloads makes them perfect targets for techniques that introduce dynamic mechanisms into High-Level Synthesis design flows.

4.2 Dynamically-Scheduled High-Level Synthesis

As demonstrated in the previous section, the static nature of the analysis required to schedule operators using High-Level Synthesis hinders the generation of efficient hardware for input programs with irregular control flow. Dynamic scheduling is a technique first proposed in 1991 by Page and Luk to alleviate the restrictions imposed by static scheduling [280]. It was later refined and integrated into the

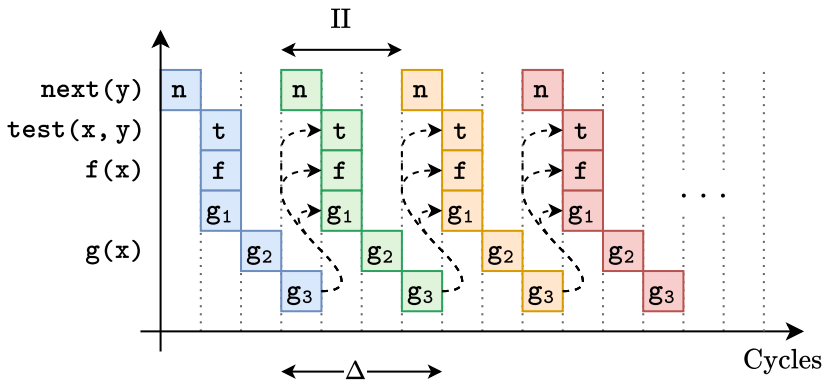


Figure 4.7 – Static schedule for the generic example of Figure 4.6. This schedule assumes that `g` takes three cycles to execute at the target frequency of the synthesized circuit. Dotted arrows represent the conditional loop-carried dependencies that constrain the schedule.

Handel-C language [20]. Later work introduces a High-Level Synthesis compiler that transforms C programs to asynchronous circuits [376]. Recent work has shown a renewed interest in dynamic scheduling, especially from the perspective of latency-insensitive design [51, 52] and dataflow circuits [183, 185, 139]. Dynamically-scheduled High-Level Synthesis [410, 184, 293] promises more efficient use of hardware resources, as well as performance gains in scenarios that are difficult to handle for standard HLS toolchains.

4.2.1 Dynamic Scheduling

Dynamic scheduling's underlying idea is that operations should be able to start as soon as the circuit can determine that their results will be needed for the execution. Translating this principle to the example exposed in Figure 4.4 means that the next iteration of the loop should be able to start as soon as the value of `filter` is known to be `false`. By relying only on statically known information, the schedule produced by a standard HLS toolchain cannot consider such runtime information. In contrast, a synthesis toolchain capable of producing dynamically scheduled hardware could produce the schedule depicted in Figure 4.8. This example assumes that `filter` returns `true` only during the third iteration of the loop. Consequently, the only loop-carried dependency happens between the third and fourth iterations. It is depicted by the arrow in Figure 4.8. The dynamic nature of such a schedule means that the representation given in Figure 4.8 is only valid for a given input data set; other initial values of `x` and `y` could produce a different outcome, leading to an altered schedule at execution time.

Dynamatic [1] is a state-of-the-art High-Level Synthesis compiler that produces dynamically-scheduled hardware from C and C++ code [184]. Dynamatic relies on a latency-insensitive approach to circuit design to start the execution of hardware paths dynamically. In place of the centralized control mechanism based on Finite State Machines generated by standard HLS tools (see Section 3.4), this HLS toolchain generates a set of components whose control logic is distributed throughout the circuit. Each component communicates data availability to its dependencies, allowing for dynamic execution. This type of hardware implementation is called *elastic* or *dataflow* circuit.

4.2.2 Dynamic Execution with Dataflow Circuits

The concept of dataflow circuits originated in the domain of asynchronous electronic design and the context of dataflow processor architectures [373, 46]. The latter deviate from the control-flow-oriented design of most instruction set processors by eliminating the need for a program counter. Instead,

1. https://dynamatic.epfl.ch/

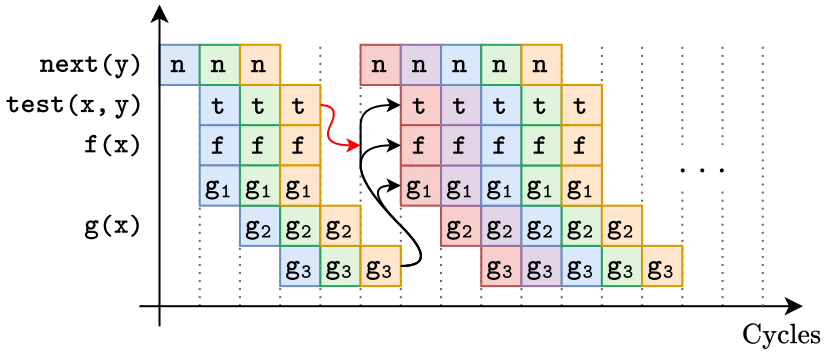


Figure 4.8 – Dynamic schedule inferred from the code in Figure 4.4 for fixed input values. In this instance, we assume that `test` always returns `true`, except for the third iteration of the loop.

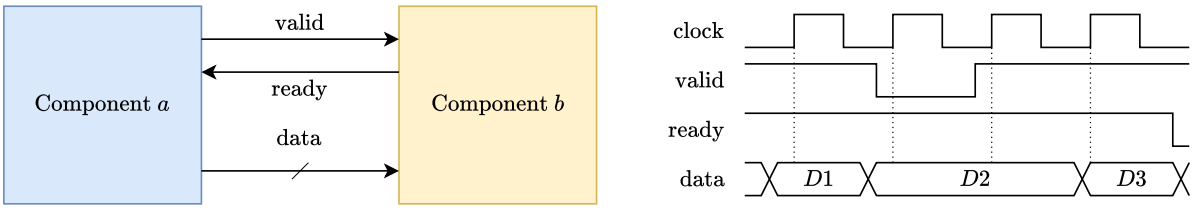


Figure 4.9 – Dataflow circuit handshake protocol. The right-hand side depicts a chronogram of a typical execution scenario. Adapted from [181].

each instruction is executed once its predecessors have finished execution. This type of architecture provides significant opportunities for latency-hiding and parallel operation execution through dynamic inter-instruction dependency detection. Such design techniques were later introduced to the world of synchronous circuits under various names, including *latency-insensitive design* [51, 52], *synchronous interlocked pipelines* [172], *elastic circuits* [82], and *dataflow networks* [378].

Synchronous dataflow circuits form the basis of the scheduling capabilities of state-of-the-art dynamic HLS tools such as Dynamatic. The latter relies on the Synchronous Elastic Flow (SELF) protocol [82] to establish communications between multiple dataflow components in the circuit. They exchange information about their readiness and the availability of data to process using a pair of handshake signals, as depicted in Figure 4.9. Component a can signal that valid data is available by setting the `valid` signal. Similarly, component b can indicate it is ready to process data by setting the `ready` signal. When both signals are set, a piece of data (a *token*, by analogy with Petri nets [266]) is transferred from a to b. The left of Figure 4.9 illustrates a possible execution scenario. While the circuit's behavior is similar to an asynchronous design, signals are only updated on the clock signal's rising edges. The handshaking protocol employed by dataflow circuits allows scheduling decisions to be made at runtime. Once all of the data is available at the inputs of a component, the latter is free to begin execution.

Dynamatic employs a small number of pre-defined elastic components to construct circuits around the concept of token exchange. These base components include storage units such as elastic buffers and FIFOs and control-flow components such as branching, merging, and path selection. Additional elastic primitives, *fork* and *join*, mimic multi-threading constructs from software programming languages. Figure 4.10 shows the dataflow representation of a simple program using Dynamatic's elastic component library. The `branch` node transfers control to either one of its outputs, depending on the value of its condition input port (linked to `test` in our example). Control-flow is merged from either of two input paths using the `merge` node, which is similar to a φ-node in Static Single Assignment (SSA) form [85]. The `fork` node distributes tokens to its outputs as soon as they are ready to process data, and the `sel` node selects one of its two inputs depending on a condition value. Finally, `buff` nodes act as storage

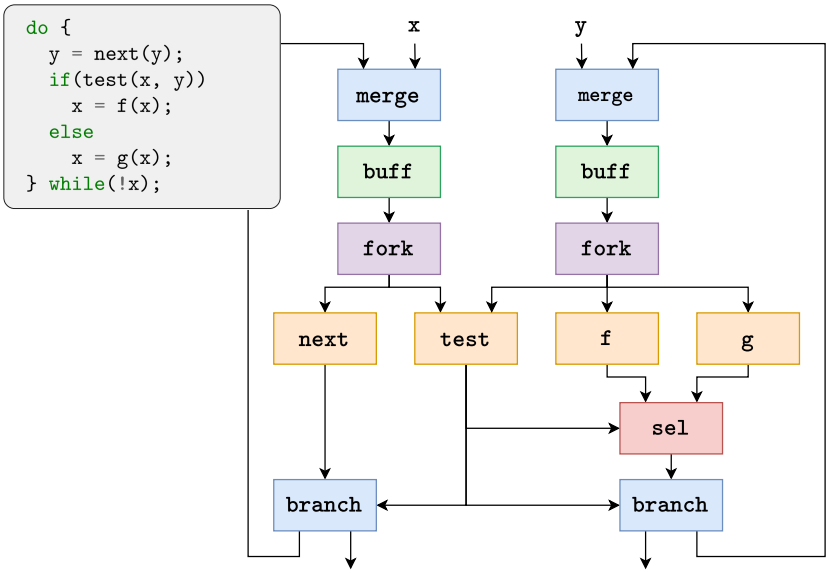


Figure 4.10 – Dataflow circuit representation of our example program (Figure 4.6). The `branch` and `merge` nodes handle control flow, while `buff`, `fork`, and `sel` nodes propagate data in the circuit.

for their input values.

For a dataflow design's execution to be semantically correct w.r.t. the input program, tokens propagating in the circuit need to follow the same order as basic blocks in the program. Dynamatic addresses this issue by propagating tokens through all basic blocks on a path in the CFG, ensuring that a given block only receives data from its immediate predecessors. This approach prevents early token consumption and subsequent deadlocks in the circuit caused by input starvation. Elastic buffers must also break up datapaths containing cycles to avoid deadlocks [188]. Later works have improved the performance of dataflow circuits by extracting more parallelism from the input program [108, 66].

Dynamically-scheduled circuits based on elastic components, such as the ones produced by Dynamatic, suffer from large design area overheads [183, 184, 187, 185] caused by the distributed handshake network required for dynamic execution. This issue has been addressed by numerous recent works, which we discuss in Section 4.2.4.

4.2.3 Memory Access Handling

When interfacing with memory, the circuits generated by dynamic HLS toolchains need to ensure the consistency of accesses w.r.t. the input program. Since each operation can start execution as soon as its predecessors are done computing their results, memory accesses may arrive out of program order. Such a scenario could cause memory dependency violations, leading to incorrect hardware execution.

A possible strategy to address this issue is to use an elastic Load-Store Queue (LSQ) to maintain consistent memory access ordering [182, 184]. This LSQ uses a dummy progress indicator token to track the currently executing basic block. The former is treated as a special variable with no data attached to it, which is live in all basic blocks of the program. Upon basic block entry, the `ready` signal of the LSQ is probed to determine if the previous basic block's memory slot allocations have all been completed. If not, the execution of the basic block is stalled until the `ready` signal is set. This allocation scheme ensures that new slots are allocated in program order, keeping accesses ordered even if the dynamic execution schedules parts of the program out-of-order.

Imposing that the input program's basic blocks be mapped directly to hardware and requiring basic block execution in program order severely limits the parallelization capabilities of dynamic HLS tools [410, 107]. Recent work has focused on alleviating this requirement using *fast token delivery* [108].

This technique performs dataflow analysis on the representation of the input program to determine which basic blocks do not require synchronization and can be bypassed. This approach effectively decouples the CFG-centric view of the program described in previous work [184] from the actual hardware implementation. This technique has been extended to support memory accesses by extending the elastic LSQ method described above [109]. Instead of coupling load and store ordering to the sequential execution of basic blocks, a separate memory dependency graph is built from the input program using static compiler analysis passes. This graph is then used to determine the order in which LSQ slots must be allocated to guarantee program correctness.

Another approach to lifting the strong requirements of in-order basic block execution in dataflow circuits was proposed as dynamic C-slow pipelining [65]. The latter adapts a technique for optimizing the critical path of synchronous circuits first proposed by Leiserson *et al.* [223], known as *C-slow pipelining*. This technique replaces each register in a circuit with a set of C registers, effectively turning the original circuit into a design with C independent threads executing concurrently. Where the circuit initially could only store a single piece of data, it can now retrieve C such data pieces and operate on them using the original combinational logic. This transformation increases the latency of the design since each computation step requires C times as many clock cycles, but it does not impact throughput. When combined with *retiming* [224, 389] (*i.e.*, register displacement inside of a circuit), C-slow pipelining can increase the execution frequency of a design by up to C times. Dynamic C-slow pipelining applies this technique to loop nests for dynamically-scheduled HLS designs. By statically proving the absence of memory dependencies between iterations of the outer loop and applying C-slow pipelining to inner loops, this approach brings significant performance improvements with little area overhead compared to standard dynamic HLS.

4.2.4 Mitigating the Cost of Dynamic Scheduling

The distributed handshaking network required to enable dynamic execution and ensure program correctness of dataflow circuits induces a significant area overhead [184, 108, 398]. This increase in resource usage may only sometimes be acceptable for constrained designs that need to run on small-scale FPGAs or need to limit their energy consumption. Several recent works have tried to address this issue by exploring opportunities to eliminate the overhead induced by dynamic scheduling in parts of the generated circuit.

The DASS HLS compiler [2] [63, 64] was the first approach aimed at reducing the overhead incurred by dynamic execution in dataflow circuits by combining dynamic and static scheduling, where such a combination does not incur a performance penalty. The core idea behind DASS revolves around including statically-scheduled hardware modules in a latency-insensitive circuit. This technique was first described by Carloni [53] under the name of *protocols and shells paradigm*. DASS starts from a fully dynamic implementation of a piece of hardware and tries to determine which regions of the design do not benefit from dynamic scheduling. It turns these regions into isolated black boxes and schedules their contents using standard static scheduling algorithms (see Section 4.1). The isolated regions are then connected to the dynamically-scheduled parts of the circuit using a wrapper that provides an elastic handshaking interface. The mixed static and dynamic circuit model employed by the DASS HLS compiler has been formalized using a probabilistic Petri net model [68]. The latter allows for automatic estimations of dynamically scheduled hardware performance and enables better target II values to be selected for statically scheduled regions. However, determining which parts of the hardware design should be statically-scheduled proves challenging. Later work proposed a heuristic-based approach to efficiently find static islands in dynamically-scheduled dataflow circuits [67].

While DASS tries to identify static islands in a dynamically scheduled design, other approaches try to uncover dynamic execution opportunities in statically scheduled hardware. Szafarczyk *et al.* propose an algorithm for identifying code regions that would benefit from dynamic scheduling in an otherwise statically-scheduled environment [350, 351]. Their approach selectively decouples parts of the Data-Dependency Graph (DDG) from the rest of the program in order to relax the constraints on the

2. https://github.com/jianyicheng/DSS

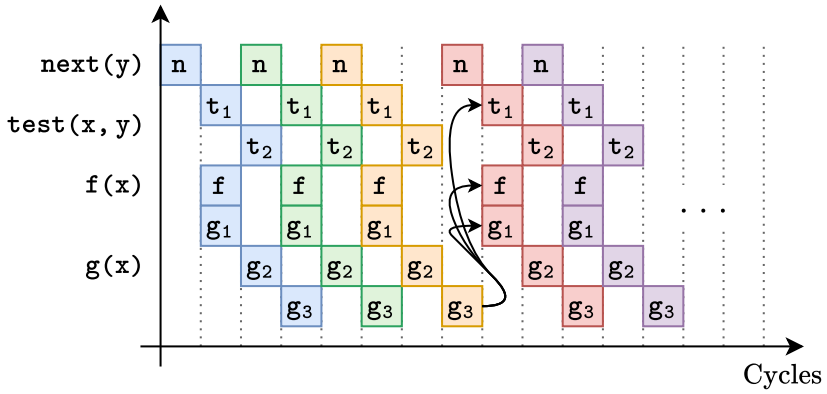


Figure 4.11 – Dynamic schedule of the code in Figure 4.6, with a slow condition. The long latency of the `test` function hinders the ability of dynamic scheduling to achieve the best execution performance.

modulo scheduling algorithm used for static schedule inference. Basic blocks and memory operations amenable to dynamic execution are isolated from the statically scheduled part of the circuit behind latency-insensitive channels. This method is not without similarities with Decoupled Software Pipelining (DSWP) [311, 279, 371, 69], a software pipelining technique that partitions loops into Strongly Connected Components (SCC) that are treated as independent threads communicating through FIFOs.

The control logic introduced by the handshake mechanism in dynamically-scheduled circuits requires two signals, `valid` and `ready`, to be computed for each dynamic component. The `ready` signal back-propagates through the circuit, stalling all predecessors of a component if the latter is not ready to perform any computation. This phenomenon is called *back-pressure*. In some instances, it is possible to prove the absence of back-pressure using a model checker to eliminate the logic required to propagate the `ready` signal in the circuit [400]. Conversely, parts of the design may not need a complex distributed network of control signals to express the validity of data. Model checking can once again be leveraged to simplify the computation of the `valid` signal in such scenarios [400]. Further handshake logic optimization opportunities can be uncovered by exploiting latency and occupancy balancing techniques [399]. The resulting designs share similarities with circuits synthesized by both statically and dynamically scheduled HLS toolchains.

Several works extend static scheduling techniques to support dynamic mechanisms. For example, ElasticFlow [354] is a hardware synthesis toolchain capable of pipelining loops with irregular control flow by dynamically distributing loop iterations to an array of Loop Processing Units (LPU). Source-to-source transformations can also be applied before the HLS synthesis flow to enable dynamic memory hazard detection and recovery [8, 232]. Dai *et al.* insert dynamic flushing logic into statically-pipelined designs to handle resource conflicts [88] and more general pipeline hazards [87, 89]. Morvan *et al.* [264] use polyhedral analysis to efficiently pipeline nested loops. Their approach flattens loop nests and statically inserts pipeline *bubbles* (*i.e.* stalls) to avoid hazards at runtime.

4.2.5 Limits of Dynamic Scheduling

Dynamically-scheduled HLS enables automated hardware synthesis for applications with irregular control flow. The dataflow circuit-based approaches described in previous sections are easily composable, making reasoning about the synthesized circuits' correctness easier [398]. However, dynamic scheduling may not always be sufficient to achieve the best possible performance for a given hardware design. Consider, for example, the code snippet of Figure 4.4, which is reproduced in Figure 4.11. Instead of assuming that the `test` function takes a single cycle to execute, as we did in Figure 4.8, let us assume that it takes two cycles to complete its execution. In this case, the decision to execute either `f` or `g`

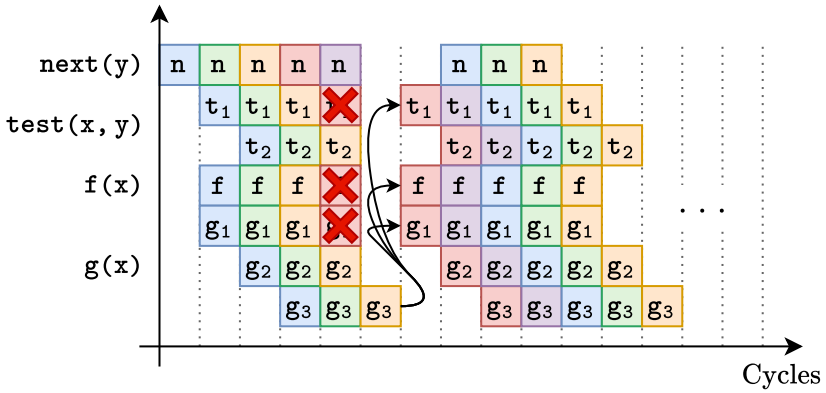


Figure 4.12 – Speculative schedule of the code in Figure 4.6, with a slow condition. By assuming that `test` is `true` most of the time, the hardware can execute one loop iteration per cycle in most cases. Crossed-out computations are rolled-back because of a mispeculation.

cannot happen before the two cycles required by `test` have elapsed. The resulting schedule is shown in Figure 4.11. The plain arrows depict a data dependency between the third and fourth iteration of the loop.

All other parameters considered equal, the schedule shown in Figure 4.11 produces faster hardware than the static scheduling approach described in Section 4.1. However, it still performs poorly w.r.t. pipeline resource utilization, with an average number of cycles per iteration (CPI) of two. The situation worsens if `test` takes three or more cycles to execute. Since the dynamic scheduling decision depends on the value of this function to trigger further loop iteration execution, the performance of the dynamic schedule would be the same as a static one. The quality of a dynamic schedule is strongly linked to the execution latency of conditions in the input program. This observation has led to the development of novel techniques for HLS, which introduce speculation in the execution flow [186, 97]. The latter are presented in the next section and expanded upon through our work in Chapters 5 and 6.

4.3 Speculative Scheduling

Speculation, in the context of hardware design, refers to a set of techniques used to improve the performance of a circuit through *speculative* execution [154, 284]. The latter refers to the execution of a piece of computation before knowing whether its result will be needed. By preemptively computing some values, hardware can avoid the latency cost of the computation, thereby improving the overall performance of the design. Speculation is often biased towards choosing the most probable path through the use of predictors or domain-specific knowledge about the data to be processed. Speculative execution is widely used in instruction set processors, and several works have tried to bring some form of speculation to HLS design flows.

4.3.1 The Case for Speculative Execution

Instruction set processors are a typical example of hardware design that heavily relies on speculative execution to enable high-performance computations. Echoing one of the eight great ideas in computer architecture coined by David Patterson [284], and discussed at the end of Section 2.3.2, speculation aims at making the common case as fast as possible. By predicting that a given execution path will be taken in a program, modern processors can significantly accelerate code execution with a relatively small area and power consumption overhead. A study by McFarlin *et al.* [248] determined that 88% of the speed-up of an Out-of-Order processor compared to an in-order one was solely due to the ubiquitous support for speculative execution built into the OoO core. They attribute the remainder of the performance

differential to the ability of the OoO processor to change the instruction's schedule during execution dynamically.

Speculative execution enables the hardware to execute computations ahead of the resolution of a potentially long condition. This ability is the perfect solution to the shortcomings of dynamically scheduled hardware demonstrated in Section 4.2.5, as shown in Figure 4.12. By assuming that the `test` function returns `true` most of the time, we can start executing a new iteration of the loop every cycle, using the result of `f` as the new value of `x`. This approach works without issues as long as the prediction is correct. However, if the prediction fails, all computations that started execution with the wrong value of `x` must be canceled and restarted with the appropriate value once `g` has finished executing. In the example trace depicted in Figure 4.12, a single misspeculation occurs at the third iteration, leading to a partial iteration needing to be canceled. If the discarded computation were to have any side effects, the latter would also need to be rolled back, ensuring that the first iteration that follows a misspeculation can execute in an environment where all state is valid. Note that the value of `y` computed by `next` needs not be rolled back since it does not depend on `x`.

Modern HLS design toolchains have difficulty synthesizing control-dominated codes [216, 97]. While dynamic scheduling addresses some of these limitations, it still produces poorly performing hardware when conditionals take several cycles to be resolved (see Section 4.2.5). Introducing speculation into HLS-produced designs would lift the restrictions of static scheduling while simultaneously bypassing the limitations of dynamic execution in control-flow-heavy applications.

Dynamic scheduling: a special case of speculative scheduling. We note that a speculative schedule is equivalent to a dynamic schedule in control-dominated applications where all conditional branches can be resolved in a single cycle.

4.3.2 Bringing Speculation to High-Level Synthesis

Path-based scheduling [48] was one of the first scheduling techniques proposed for the High-Level Synthesis of control-dominated applications. This algorithm schedules each possible execution path in the input program independently and then tries to overlap the resulting schedules by minimizing the amount of state changes required to move from one path to the next. However, path-based scheduling lacks pipelining capabilities and requires the order of operations to be chosen during the scheduling step. The latter restriction has been addressed in later works [163, 34]. Kountouris and Wolinski [202] propose a heuristic method based on list scheduling that considers resource sharing and code motion opportunities to synthesize control-flow-dominated applications. Later work [206] employs an exact method using derived constraints programming models to achieve a similar goal. Gupta *et al.* [144, 145, 143] develop a High-Level Synthesis framework named SPARK [141] that employs several techniques borrowed from VLIW compilers to synthesize hardware. In particular, it employs a set of code transformation techniques named *speculative code motion* to extract maximum parallelism from the input code. SPARK can move operations across (*trailblazing*), out of (*speculation*), and into (*reverse speculation*) conditional statements. These transformations are guided by heuristics aimed at improving the performance and surface area of the synthesized designs. Global scheduling techniques have also been used to synthesize control-heavy applications independently of control dependencies [380, 296].

Modern HLS design flows incorporate optimizations based on speculative code motion in their regular synthesis pipeline. While it improves the performance of hardware designs, this technique suffers from the intrinsic limitations of static scheduling outlined in Section 4.1. Alternative methods try to incorporate speculative execution directly into the synthesized hardware, which lifts the restrictions imposed by static schedules and allows execution paths to be chosen at runtime. Radivojević and Brewer [306] propose a symbolic formulation of the control-dependent and resource-constrained scheduling problem, which allows them to derive optimal schedules for applications with arbitrary forward-branching control paths. The execution order of conditionals is determined during execution instead of being determined statically during scheduling. Holtmann and Ernst [158, 157] introduce a static branch predictor (*i.e.* a predictor that always predicts the same value at a given point of execution, without taking any

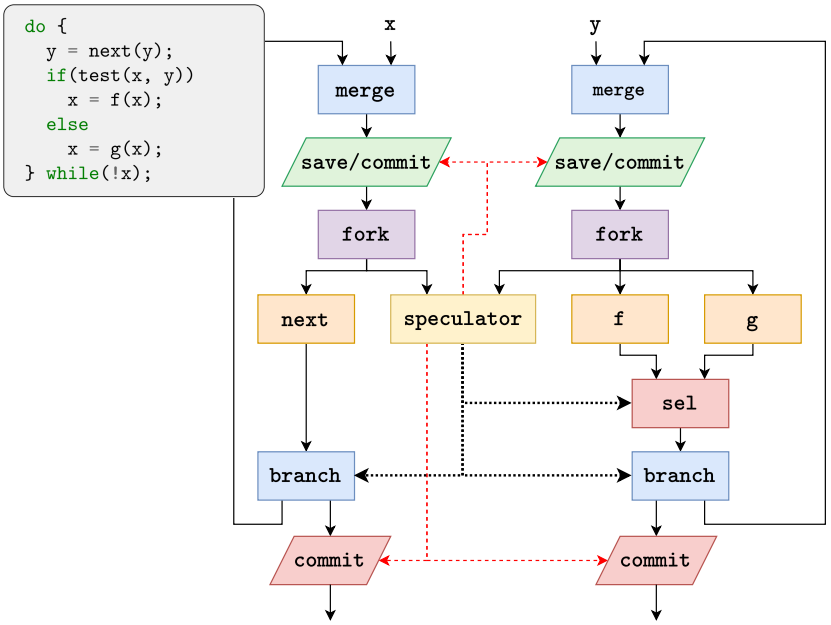


Figure 4.13 – Speculative dataflow circuit representation of our example program (Figure 4.6). Plain arrows represent the flow of data tokens, and dotted arrows mark speculative token paths. Dashed red arrows are control signals issued by the speculator.

history into account) to predict the execution of hardware and perform scheduling at runtime. Later work describes a technique to reduce the control logic overhead incurred by static branch prediction in HLS [155]. Lakshminarayana *et al.* [214] propose a scheduling algorithm that can produce speculative schedules from behavioral descriptions of hardware designs. Several works describe the integration of branch predictors in HLS designs [216, 337], which enable high-performance execution on a wide variety of workloads and for many applications. Other predictors, such as carry predictors for functional units [96], have also been integrated into HLS-generated hardware designs.

Recent work has shown a renewed interest in speculative execution support for High-Level Synthesis. Speculative dataflow circuits [186] bring speculation to the elastic hardware designs described in Section 4.2.2. On the other hand, Speculative Loop Pipelining (SLP) [97] introduces a novel scheduling technique that can generate speculative hardware designs through source-to-source transformations. The following sections describe these two techniques.

4.3.3 Speculative Dataflow Circuits

Speculative execution support implies an ability to dynamically adapt the execution path of hardware according to the result of a given speculation. The dynamic nature of such a problem is well-suited to approaches based on handshake protocols and token-passing. PreCoRe [360, 359] uses such a token-based approach to hide the potentially variable latency of external memory accesses in hardware designs. The latter are handled through speculative queue buffers combined with speculative tokens. These tokens are used to mark values produced by speculation and to handle rollbacks (resp. commits) in case of mispeculation (resp. correct speculation).

Josipović *et al.* describe a similar speculative token-based approach [186] that builds upon their previous work on dynamically-scheduled HLS [184]. They describe speculative dataflow circuits, a type of elastic hardware design capable of speculative execution. They introduce a new kind of data token to be exchanged with their handshaking protocol to enable speculative execution in elastic circuits: *speculative tokens*. The latter is generated by a dedicated hardware component, the *speculator*, which speculatively integrates all the prediction logic required to evaluate a control-flow path. In addition

to issuing speculative tokens in the circuit, speculators ensure that the predictions made during the execution are correct. If not, they control the rollback logic to revert the current state to a valid one. The rollback logic comprises two additional structural circuit elements, namely *commit units* and *save units*. The former are used to retain speculative tokens at critical parts of the circuit until the speculator has validated the corresponding prediction. The commit unit then converts the speculative token to a regular data token and forwards it to subsequent computational elements. If the prediction is incorrect, the commit unit discards the speculative token. Save units are the counterpart of commit units. They store valid data tokens that enter a circuit region where speculation may happen. The saved tokens are restored or flushed out depending on the speculator's decision regarding the corresponding speculative decision.

Figure 4.13 shows an example of a speculative dataflow circuit generated from the code first introduced in Figure 4.10. The `test` logic is replaced by the speculator, which controls the save and commit units. Speculative tokens travel through the dotted arrows, and all other paths in the dataflow circuit use regular data tokens. Each time a speculator is inserted into a dataflow circuit, it defines a speculative region. This region is delimited by save units at its entry points and commit units at its outputs. This setup allows the extent of speculation to be well-defined in the circuit and avoids possible interferences between multiple speculators. In addition to new elastic components, speculation mandates that the execution path be marked as carrying a speculative value. A simple marking bit follows the speculative datapath and indicates whether a speculator issued the value currently carried by said datapath.

Surface area overhead. As pointed out in Section 4.2.4, the surface area overhead of dataflow circuits compared to statically-scheduled circuits can be significant. The introduction of speculative execution capabilities stresses this phenomenon. Experimental results show a geometric mean increase in resource usage of 73% for LUTs and 8% for FFs, compared to the statically-scheduled design [186].

4.3.4 Speculative Loop Pipelining

Speculative hardware synthesis methods based on speculative dataflow circuits enable HLS tools to introduce speculation generically by adding speculative components to the intermediate dataflow representation of the toolchain. This approach leverages dynamic execution and prediction to achieve execution similar to what can be found in modern instruction set processors. However, this technique relies on a custom HLS middle and back-end incorporating all the required components to generate speculative dataflow circuits, making integrating with existing HLS tools harder. One way to circumvent this limitation is to rely on input program transformations to expose speculation directly at the source level.

Speculative Loop Pipelining (SLP) [97] is a hardware synthesis technique that relies on source-to-source transformations to directly expose speculative behavior in the high-level specification used as an input to an HLS toolchain. It extends traditional loop pipelining with an additional pass to expose speculation opportunities in strongly connected components of the input program's CDFG (Control and Data-Flow Graph). Figure 4.14 illustrates this code transformation on the running example introduced in Figure 4.6. The initial loop code is transformed to decouple the data and control paths in the execution, mapping data-dependent operations to per-cycle iterations and control decisions to an external FSM, whose current state is stored in `cs`. By making each iteration of the transformed loop correspond to one execution cycle, data dependencies, and reuse distances become explicit, enabling the HLS toolchain to schedule the speculative circuit efficiently. The entire control path is abstracted in an FSM depicted in Figure 4.15. This FSM handles speculation and triggers rollbacks or commits actions depending on the correctness of the predicted value. It contains four distinct active states:

— the `FILL` state corresponds to the pipeline data fill-up;
— the `PROCEED` state corresponds to the stationary state of the pipeline, where correct speculations are committed until a misspeculation is detected. In the latter case, the FSM moves to the transient `STALL` state;

```
1  do {
2  #pragma HLS pipeline II=1
3    y[t] = next(y[t-1]);
4    ctrl[t] = test(s_x[t-2], y[t-1]);
5    s_x[t] = f(s_x[t-1]);
6    mis_x[t] = g(s_x[t-3]);
7    cs = nextstate(cs, ctrl[t]);
8    if (cs.rollback)
9      s_x[t] = mis_x[t];
10   if (cs.commit)
11     x = cs.sel ? s_x[t-1] : mis_x[t];
12   t += 1;
13 } while(!(x && cs.commit));
```

Figure 4.14 – Result of the SLP transformation applied to the code in Figure 4.6. The control-flow of the loop body is factored into an FSM, whose state is stored in `cs`. Adapted from [97].

— the `STALL` state is used to pause the execution after a mispeculation to wait for the correct value to be available, after which the FSM transitions to the `ROLLB` state;
— the `ROLLB` state restores the pipeline's content in case of a mispeculation, effectively rolling back all computations relying on an incorrect prediction. Once rollback is completed, the pipeline is restarted in the `FILL` state.

The SLP source-to-source transformation allows speculative hardware to be generated with regular HLS toolchains. Its backend relies on a standard HLS toolchain to perform resource allocation and sharing. Speculative loop pipelining divides the input program's CDFG into strongly connected components (SCC) and applies speculation to each SCC. To automate the detection of potential speculative execution points, SLP relies on a derivative of SSA form (see Section 3.4.2) to represent programs, namely Gated-SSA (GSSA) [366, 367]. GSSA replaces the φ-nodes of the standard SSA representation by μ-, γ-and η-nodes, collectively referred to as *gating nodes*. SLP extends GSSA by introducing ρ-nodes to represent rollback logic. The semantics of these nodes is defined as follows:

— $\mu(x_{\text{ext}}, x_{\text{in}})$ replaces φ-nodes at the head of loops and selects either the initial value x_{ext} or the loop-carried x_{in} value for a variable x;
— $\gamma(c, x_{\text{false}}, x_{\text{true}})$ replaces φ-nodes at confluence points after conditional statements, selecting either x_{true} or x_{false} depending on the value of the condition c;
— $\eta(c, x_{\text{out}})$ replaces φ-nodes at loop exits and selects the corresponding value of x_{out} when the loop exit condition c is met;
— $\rho(d, c)$ is used to model a rollback with a data buffer d and a control signal c: when $c = 0$, the ρ-node forwards the most recent value of d to its output, and when $c > 0$, it discards the c most recent elements and forwards the value in d that was stored c iterations in the past.

This representation allows a source-to-source compiler to easily manipulate the input program's control flow and transform it through simple changes to the Gated-SSA structure. Most importantly, the fact that control-flow decisions are encoded directly in gating nodes enables reasoning about control signals in the synthesized circuit. The SLP transformation, which is implemented using the GeCoS compiler framework [124], detects γ-nodes with *unbalanced* inputs (*i.e.*, with at least one fast and one slow path), and selects them as potential speculation candidates. It modifies γ-node inputs in SCCs to expose the reuse distance for each data source, as shown in lines 3–6 of the code in Figure 4.14. A separate *shadow variable* (`mis_x` in our example) is created for each speculated live-out variable. The latter is used to compute values along non-speculatively executed paths and is used in case of mispeculation. SLP then creates the FSM controlling the speculation logic depicted in Figure 4.15 and creates an additional execution path in the program to commit values out of the current SCC. Finally, ρ-nodes are inserted on back-edges of the CDFG for all live-out variables that are not subject

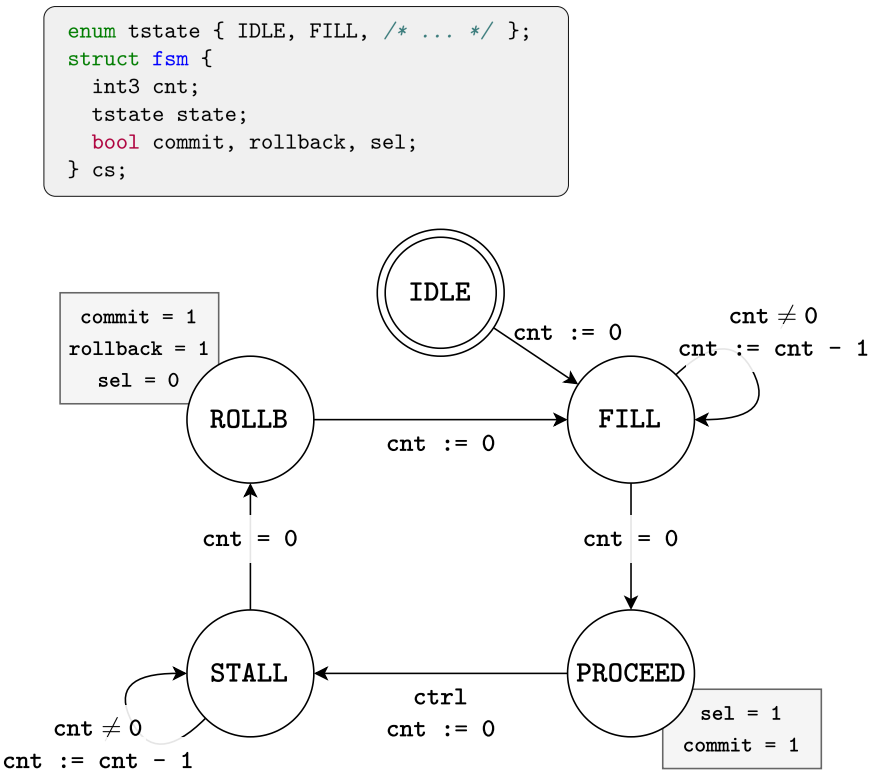


Figure 4.15 – SLP control FSM. The execution starts in the `IDLE` before filling the pipeline in the `FILL` state. The `PROCEED` state handles normal execution and delegates to the `STALL` and `ROLLB` states in case of mispeculation.

to speculation. These nodes handle the rollback logic required to recover from a mispeculation.

The Gated-SSA representation of our example program is shown on the left of Figure 4.16. Their GSSA equivalents have replaced the φ-nodes of the standard SSA form. The right of Figure 4.16 gives a graphical representation of the dependencies between operators in the input program, with an execution latency of two cycles for `test` and three cycles for `g`. We call this representation an *instruction dependency graph* (IDG). The depicted IDG features two distinct SCCs, one for variable `y` (SCC_y), and one for `x` (SCC_x). The former does not contain any γ-node (*i.e.*, it does not expose any speculation opportunity), so SLP does not consider it for speculative execution. On the other hand, SCC_x contains a conditional statement with a fast and a slow input, which makes it a candidate for speculative execution. SLP operates on the IDG representation and inserts the FSM responsible for the speculation logic between the `test` function's output and the control signal input of the γ-node. The output of SCC_y is sent to a FIFO to buffer any value that may be produced during the mispeculation recovery procedure in SCC_x. This example has no rollback nodes since a potential speculation error on `x` cannot affect any other variable that may need to be restored to a correct value during mispeculation recovery.

4.3.5 From Speculative Loop Pipelining to Processor Synthesis

SLP integrates nicely with existing HLS toolchains and can take advantage of the many target-specific optimizations such tools can perform when targetting a specific family of FPGAs. However, it suffers from several limitations when handling multiple interacting speculations and does not address the issue of memory access handling. Our work builds on the foundations laid out by SLP. We extend it to handle arbitrarily complex speculation patterns (Chapter 5) and introduce proper memory access

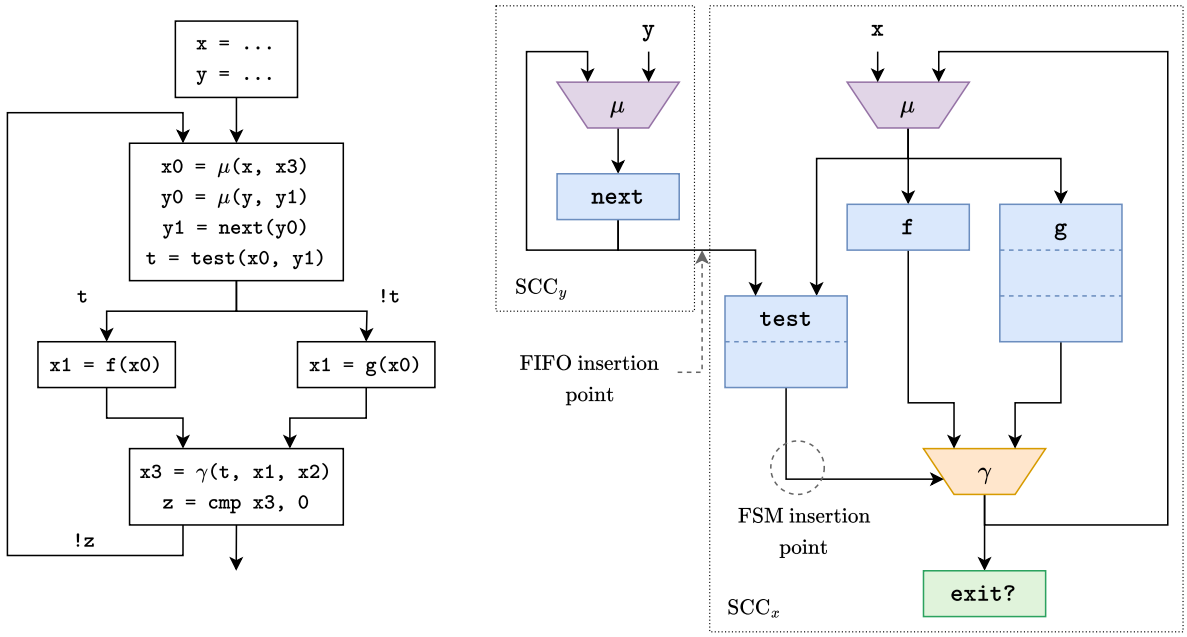


Figure 4.16 – Gated-SSA (left) and IDG (right) representation of the program in Figure 4.6. μ-nodes replace SSA form's φ-nodes in the loop header, and a γ-node represents the conditional decision in the input program. The SLP transformation decouples SCC_x from SCC_y by inserting a FIFO and inserts an FSM to handle control-flow between the output of `test` and the γ-node's control signal input.

support in a speculative context (Chapter 6). We apply these techniques to automatically synthesize pipelined RISC-V instruction set processors in Chapter 7.

CHAPTER 5

Speculative Accelerator Design using High-Level Synthesis

Hold onto your hats, this next part is new stuff.
— *Anonymous* (Marginal note, Concrete Mathematics)

Custom hardware accelerator usage is shifting toward new application domains such as graph analytics and unstructured text analysis. These applications expose irregular and complex control flow, which is challenging to map to hardware, especially when operating from a C or C++ description in High-Level Synthesis toolchains. Several approaches relying on speculative execution, described in Chapter 4, have been proposed to overcome those limitations. However, they often fail to handle the multiple interacting speculations required for realistic use cases. This chapter describes a fully automated hardware synthesis flow that can generate speculative hardware accelerators. We start by motivating our work in Section 5.1 before giving an overview of our speculative hardware design flow, SpecHLS, in Section 5.2. SpecHLS is a source-to-source compiler framework supporting speculative loop pipelining for arbitrary control flow and memory speculation patterns. Handling interacting speculations is a complex problem, which we discuss in Section 5.3. We propose an iterative speculation pattern resolution algorithm to solve this problem in Section 5.4. Finally, we evaluate our toolchain on applications that can benefit from speculative execution. We show that our proposed approach can provide an order of magnitude performance improvements without suffering from a resource usage explosion (Section 5.5).

5.1 Motivation

Modern instruction set processor architectures use aggressive speculation strategies to improve their performance, trying to extract as much instruction-level parallelism as possible from programs. Speculative execution support often comes at a drastic increase in hardware design complexity and non-negligible surface area usage. For these reasons, custom hardware accelerators rarely implement speculation support. However, acceleration techniques are becoming increasingly relevant for application domains such as graph analytics, sparse linear algebra, and machine learning. These workloads, which could benefit from speculation, require complex accelerators whose design would only become more complex if they were to add speculative execution support.

High-Level Synthesis partly addresses this issue: a designer can start from a C or C++ specification and refine it through a design space exploration step to derive a highly optimized hardware implementation. However, as demonstrated in the previous chapter, current HLS tools have severe limitations for

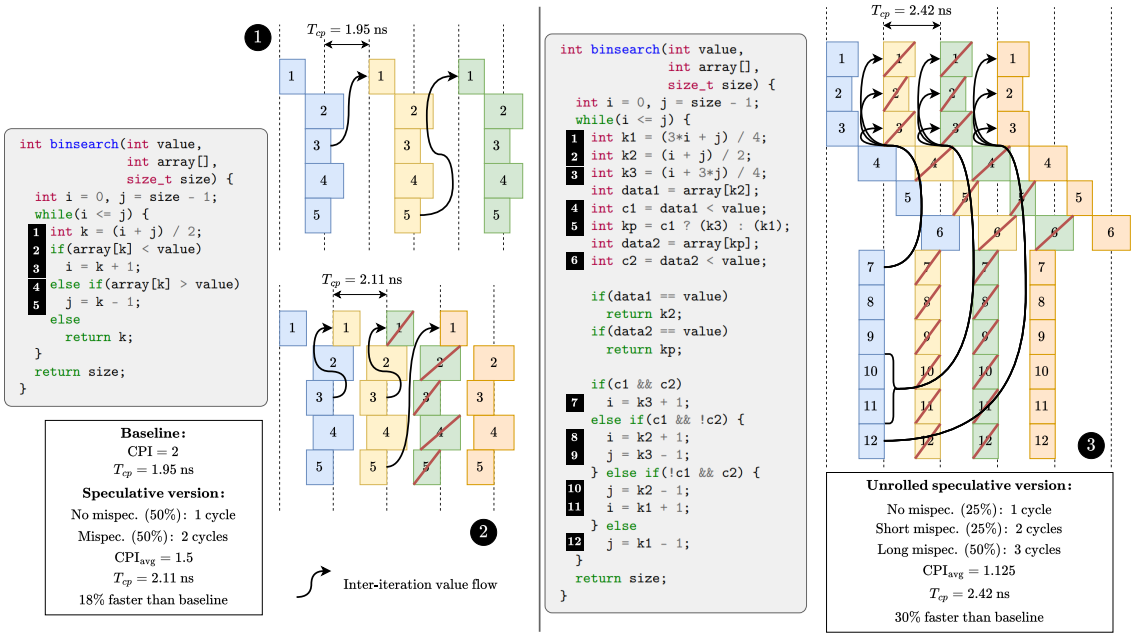


Figure 5.1 – Loop pipelining strategies for a binary search kernel application. The leftmost part represents the C code for the basic kernel and execution traces for the baseline and speculative versions. The rightmost part represents the unrolled version of the kernel and the execution trace for the speculative version.

programs with irregular control flow or data-dependent behavior. There have been attempts to better support this type of kernel in HLS, notably through the use of dynamic or speculative schedules [89, 272, 184, 186] (see Section 4.2 and Section 4.3). Nevertheless, no existing approach is general enough to support arbitrary speculation patterns.

To demonstrate why supporting complex intertwined speculations is essential to producing efficient speculative hardware designs, we consider the simple binary search algorithm shown in the leftmost part of Figure 5.1. The kernel exposes a loop-carried dependency over `i` and `j`, which prevents overlapping consecutive iterations. As a consequence, loop pipelining leads to a minimum initiation interval of II = 2, with a post-synthesis clock speed of $T_{\text{cp}} = 1.9\,\text{ns}$ on an XU280 FPGA. The corresponding execution trace is given in Part ❶ of Figure 5.1.

However, it is possible to speculate that the first conditional 2 is taken and immediately start the next iteration, as illustrated by the trace in Part ❷ of Figure 5.1. The resulting circuit operates at a lower clock speed $T_{\text{cp}} = 2.11$ and suffers from a high misspeculation rate (50%), assuming an unskewed data distribution within `array`. However, misspeculated iterations only take two cycles to execute, just as for the baseline accelerator. The average Cycles Per Iteration (CPI) is therefore $\text{CPI}_{\text{avg}} = 1.5$ instead of 2. Taking into account the increased value of T_{cp}, we can conclude that the speculative accelerator improves performance by 18%.

Let us further consider a manually unrolled version of the kernel, as shown on the right of Figure 5.1. In this instance, it is possible to speculate two iterations ahead, with a misspeculation rate of 75% and an even lower clock speed ($T_{\text{cp}} = 2.42$), as represented in Part ❸. The next iteration starts immediately with the speculated values for `i` and `j`. After one cycle, the first condition is resolved, and there is a 50% chance for the first speculation to be correct. In such a case, we immediately start the next iteration with another speculated value for `i` and `j`. This second speculation also has a 50% chance of being correct. Consequently, 25% of iterations speculate correctly and need one cycle to execute. 50% of iterations misspeculate once and need two cycles. 25% of them misspeculate twice and need three cycles. On average, the accelerator needs 2.25 cycles to execute one iteration. This duration corresponds to two

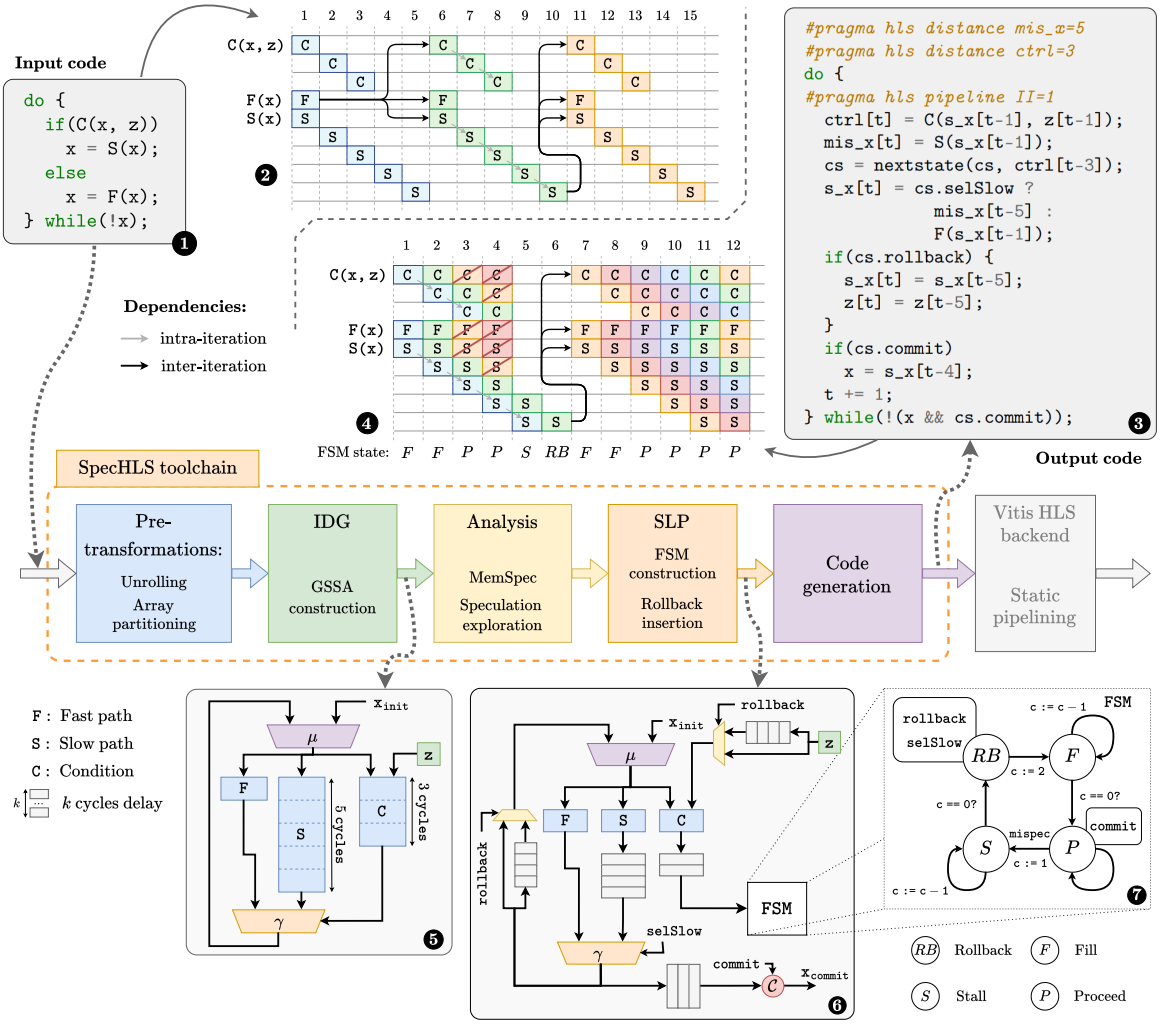


Figure 5.2 – SpecHLS source-to-source transformation flow. The toolchain takes C code ❶ as input and produces C code ❸. ❷ and ❹ show the respective schedules of the input and output code. The toolchain operates on the intermediate representation depicted in ❺ (before transformation) and ❻ (after transformation). The speculation-controlling FSM is outlined in ❼.

iterations of the non-unrolled kernel. The unrolled speculative accelerator leads to the most profitable strategy among the three shown in Figure 5.1.

There are three lessons to be learned from this simple example. First, speculative execution can be profitable without high-confidence predictions. Second, it is often necessary to resort to intertwined or nested speculations. Last, the best solution is not always the most obvious one.

5.2 SpecHLS Compiler Flow

SpecHLS is a source-to-source compiler transformation flow whose main components are shown in Figure 5.2. SpecHLS accepts a subset of C++ as input and produces transformed C++ code supporting speculative loop pipelining. Vitis HLS can then exploit the latter to obtain an accelerator design. The key idea of speculative loop pipelining [97] (SLP) is to consider speculation as a parallelizing transformation rather than a backend optimization. Consequently, SpecHLS does not address the issue of scheduling individual operations in the pipeline and delegates that task to the HLS tool it uses as a

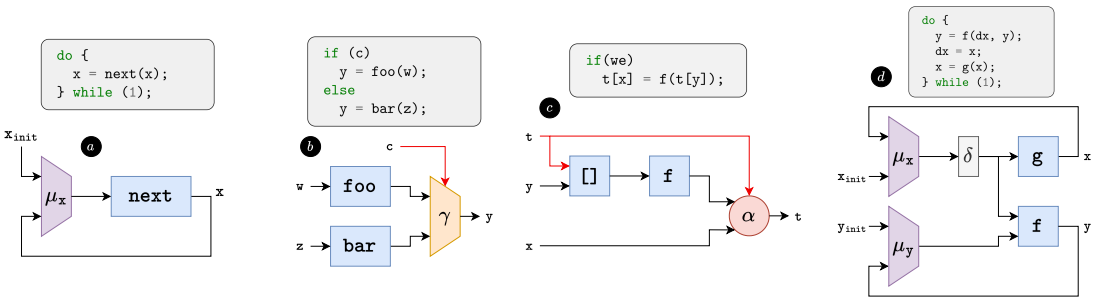


Figure 5.3 – Extended Gated-SSA operators. μ-nodes (a) represent loop headers, γ-nodes (b) encode control-flow decisions, α-nodes (c) serve as array updates, and δ-nodes (d) act as delays on a hardware path.

backend.

The C code produced by SpecHLS from the code in Part 1 is illustrated in Part 3 of Figure 5.2. A corresponding execution trace is shown in Part 4. The output code 3 exposes longer reuse distances and implements the pipeline hazard management logic. Increasing the reuse distance allows the HLS pipeliner to work more aggressively to achieve a lower II since data dependencies impose fewer constraints between iterations. The SpecHLS transformation flow automatically infers the reuse distance. The larger the distance, the more aggressive speculation can be. There is, therefore, a trade-off between (i) values of II and T_{cp} and (ii) area and mispeculation recovery cost.

Our compiler flow relies on the same Gated-SSA representation as the initial SLP implementation, presented in Section 4.3.4. We extend it with δ-nodes. The latter are akin to *delays* introduced on the hardware path in which they appear. They can be seen as micro-architectural delays in the generated hardware, while μ-nodes can be thought of as iteration delays. δ-nodes increase the reuse distance on the path where they are inserted. Lastly, α-nodes act as store operations into arrays. They take an array, a value, and an index as inputs and produce the updated array. Figure 5.3 illustrates the GSSA operators we manipulate in this and subsequent chapters. We refer to this representation as an *Instruction Dependency Graph* (IDG). The loop condition is omitted from μ-nodes to simplify our examples.

Part 5 and Part 6 of Figure 5.2 depict simplified Gated-SSA representations of the programs in 1 and 3. In this example, the conditional is exposed as a γ-node that selects the value used for the next iteration based on its control port. These values are obtained from three distinct paths: fast (`F`), slow (`S`), and conditional (`C`).

Because they capture conditionals, γ-nodes expose speculation opportunities. For example, speculating that the γ-node in 5 selects the fast path enables pipelining with $\text{II} = 1$. To be considered for speculation, a given γ-node must (i) expose a long delay over at least one of its data or control inputs, (ii) expose a short delay over one of its input data ports, and (ii) the probability for the latter path to be taken must be higher than a given threshold. The latter probability can be obtained analytically or by profiling the input code. Scaling this speculation exploration phase to larger designs is a challenging problem we address in Chapter 7.

We extend the Speculative Loop Pipelining (SLP) transformation first introduced by Derrien *et al.* [97] and presented in Section 4.3.4. The SLP transformation operates on the loop body's IR, and its result is illustrated in Part 6 of Figure 5.2. Each loop produces a strongly connected component (SCC) in the intermediate representation. Delays are inserted to increase reuse distances on non-speculated paths (e.g., the paths involving `S` and `C`). Those delays increase the reuse distance for loop-carried dependencies and provide opportunities for the HLS pipeliner to overlap the execution of independent computations. Such delays allow the back-end HLS toolchain to produce a schedule with an initiation interval of $\text{II} = 1$.

Recovery logic is also inserted to handle mispeculations. This logic consists of (i) rollback nodes for restoring past inputs and (ii) commit nodes for filtering invalid outputs. The recovery mechanism is

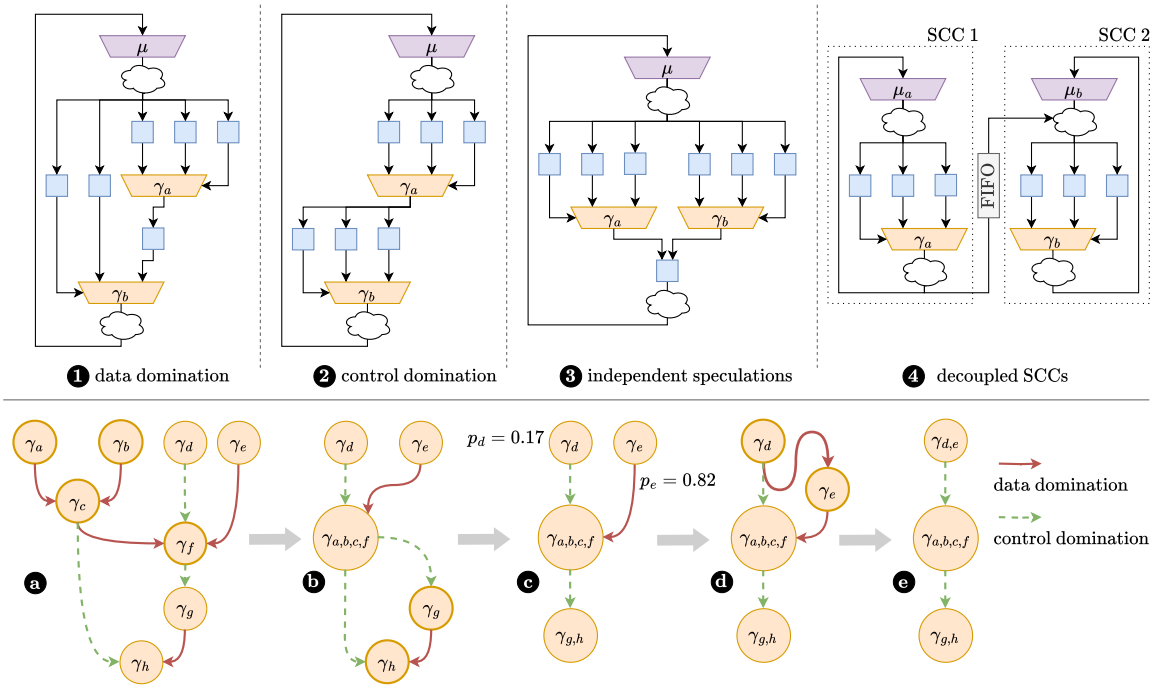


Figure 5.4 – Taxonomy of γ-node patterns handled by SpecHLS, and example γ-node graph reduction.

controlled by a Finite State Machine (FSM) depicted in ❼. Section 4.3.4 describes how those nodes are inserted.

Existing speculation techniques for accelerators [97, 186] are restricted to cases where speculations are independent and do not interfere. In practice, as exemplified in Figure 5.1, speculations are often intertwined, making recovery logic more complex. Therefore, it is necessary to characterize precisely how dependent speculations interact.

In this chapter, we generalize SLP to support arbitrary speculation configurations that involve one or more speculative decisions. We achieve this by partitioning the intermediate representation into a set of *speculation patterns*, *i.e.* disjoint sets of γ-nodes. We propose a taxonomy of speculation patterns coupled with a set of transformation rules. We use these rules to rewrite arbitrary patterns into a normalized representation supported by our control logic generation flow.

5.3 Classification of Multiple Speculation Patterns

Handling patterns involving multiple speculations is challenging: the outcome of a given speculation may depend on the outcome of another one, resulting in intricate recovery schemes. This complexity depends on the dependency graph relating the speculated γ-nodes in the target loop. This section presents how SpecHLS handles four γ-node patterns that arise when dealing with multiple speculations, namely *data domination*, *control domination*, *independent γ-nodes* and *decoupled SCCs*. The iterative resolution of these patterns from the Gated-SSA intermediate representation is described in a later section.

5.3.1 Data Domination

Data domination (DD) is a speculation pattern where speculative values only flow through the data inputs of γ-nodes. An example of data domination is provided in Part ❶ of Figure 5.4. The speculation chooses the shortest path through all nodes in this pattern. Once conditions have been resolved, and

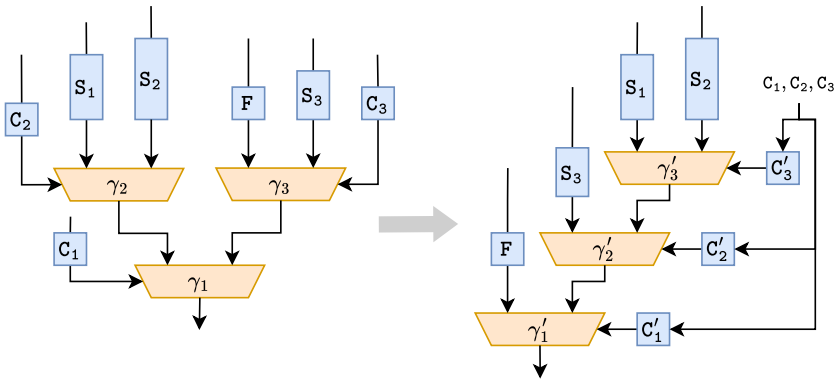


Figure 5.5 – Data-domination canonicalization. γ-nodes are rearranged to form a comb-shaped tree, with inputs of increasing execution latency.

in case of mispeculation, the control logic rolls back the computation, and the speculation chooses the second-shortest path in the pattern. The same process is repeated if another mispeculation occurs.

The execution trace depicted in Part ❸ of Figure 5.1 is an example of data domination: at cycle 2, an iteration is started using the output of computation 7 from the first iteration; during cycle 2, the condition computed by 4 reveals a mispeculation, and the iteration is started again at cycle 3 using the output of 10 and 11; during cycle 3, the condition computed by 6 reveals another mispeculation, and the iteration is started a third time at cycle 4 using the output of 12.

Data domination is handled using a single Finite State Machine (FSM), which controls all the γ-nodes alongside the rollback mechanisms. This FSM is built by composing the FSMs that correspond to single speculations for each γ-node in the pattern.

The GSSA representation may only sometimes expose data domination patterns amenable to the simple speculation logic described above. We introduce a *canonicalization* pass that transforms γ-node trees that form a data domination pattern into a canonical form that exposes paths in decreasing profitability order, as illustrated in Figure 5.5. Another challenge arises when handling data domination patterns in which values do not flow directly from one γ-node to another. This scenario is depicted in Part ❶ of Figure 5.4: a block processes the output of γ_a before sending its output to γ_b. The additional computation may de-synchronize the resolution of the conditions driving γ_a and γ_b and would require additional control logic to ensure proper speculative behavior. We simplify this pattern by retiming and duplicating this extra computation on all inputs of γ_a, thereby generating additional hardware. Better handling of this particular case of data domination is left for future work.

5.3.2 Control Domination

In a control domination (CD) pattern, the output of a γ-node impacts the control input of another one, as depicted in Part ❷ of Figure 5.4. The challenge in such a speculation pattern is correctly handling situations where a mispeculation is triggered based on incorrect data from previous mispeculations. The latter's condition may not have been resolved when the former mispeculation occurs.

Control domination is handled by inserting a rollback mechanism and an FSM for each γ-node. When a mispeculation is signaled by an FSM, it resets every FSM having a shorter condition. It prevents FSMs with longer conditions from handling a mispeculation signal resulting from the mispeculation being handled by itself. This behavior is achieved using a mask system that hides some mispeculation signals. We can handle an arbitrary number of speculative γ-nodes using control-dominated speculations, and all speculation patterns can be reduced to CD scenarios, ensuring a safe but sub-optimal rollback mechanism in every situation.

Consider the example depicted in Part ❷ of Figure 5.4, assuming that the condition controlling γ_a is longer than the one controlling γ_b. A mispeculation from γ_a resets the FSM controlling

γ_b. A mispeculation from γ_b hides mispeculation signals to γ_a for latency(`Cb`) cycles, starting after latency(`Ca`) − latency(`Cb`) cycles.

5.3.3 Independent Speculations

Two γ-nodes are independent if they belong to the same loop but do not interact within a given loop iteration, as depicted in Part ❸ of Figure 5.4. However, as both nodes are in the same SCC, the values produced by the γ-nodes are used by one another at the next iteration.

If one speculation fails at a given iteration, the values used to compute the next iteration are wrong for both iterations. Consequently, this pattern cannot be handled using two independent FSMs. We insert an artificial dependency between independent γ-nodes and consider that they form a data domination pattern. We create a link from the node with the lowest mispeculation probability to the node with the highest one, minimizing the mispeculation probability for the resulting pattern. The pattern depicted in Part ❸ of Figure 5.4 would be equivalent to a single four-input γ-node choosing between all pairs of values from the two original γ-nodes.

5.3.4 Decoupled Strongly-Connected Components

The last pattern consists of γ-nodes in different SCCs, as depicted in Part ❹ of Figure 5.4. Even if there are interactions between the SCCs, it is possible to schedule each of them using decoupled software pipelining [311]. Consequently, each γ-node has its own FSM and the different speculations are decoupled using a FIFO between the two SCCs.

5.4 Iterative Speculation Pattern Resolution

Because several patterns may be involved in the same loop, we build a directed acyclic graph capturing the interactions between all γ-nodes within an SCC. The bottom right part of Figure 5.4 represents such a graph, where edges capture interaction patterns.

We start with one node of the graph for each γ-node in the SCC. We merge nodes in this graph, with each merging operation corresponding to the insertion of an FSM. When merging patterns recursively, we build a product FSM to drive the resulting grouping. At the end of this deterministic process, we obtain a chain of control-dominated nodes for the entire loop body that we handle with an FSM for each node.

In Figure 5.4, we start from an SCC with eight distinct γ-nodes labeled γ_a to γ_h. Data dominations are represented using plain arrows, and control dominations are represented using dashed arrows. Two nodes can only be linked by a single edge. We start by merging DD patterns: in ⓐ, we merge γ_a, γ_b, γ_c, and γ_f, preserving the incoming and outgoing edges for this group of nodes. We merge γ_g and γ_h in ⓑ, but the resulting node would have two incoming edges from $\gamma_{a,b,c,f}$: a control domination and data domination. Since the former is more generic than the latter, this pattern is resolved as a single control domination to get ⓒ. At this stage, we could either merge γ_d or γ_e into $\gamma_{a,b,c,f}$. We examine the mispeculation probabilities for each node, p_d and p_e, to take a decision and insert an artificial direct data domination between γ_d and γ_e in ⓓ since $p_d < p_e$. Finally, we merge γ_d and γ_e and resolve the two edges going into $\gamma_{a,b,c,f}$ as a control domination in ⓔ.

5.5 Experimental Evaluation

We evaluate our technique based on three examples we identified as good candidates for our approach. We generate N_{conf} speculative versions using more or less aggressive speculation configurations and provide the results for the most relevant ones. We compare the performance and resource usage of the speculatively pipelined designs with the non-speculative versions. Our results were obtained with Vitis HLS, targeting an XU280 FPGA.

Table 5.1 – Results of our experimental study on selected examples. The first columns show the different accelerators' performance and relative speed-ups compared to the baseline implementation. The last columns show their area cost.

Speculation configuration	N_{conf}	II	$F_{\max}$ (MHz)	CPI	Speed-up	Area LUT	Area FF	Area SRL	Area DSP
Binary search									
Baseline	1	2	513	2	1×	128	100	0	0
Speculative	48	1	473	1.5	1.2×	353	371	0	0
Unrolled speculative	108	1	413	1.1	1.5×	813	657	94	0
Hmmer									
Baseline (M=8)	1	4	221	4	1×	5,707	3,006	0	0
Speculative (M=8)	64	1	186	1	3.4×	10,048	10,391	879	0
Baseline (M=16)	1	6	221	6	1×	11,084	5,655	0	0
Speculative (M=16)	64	1	235	1	6.4×	19,756	18,265	2,895	0
SKA gridding									
Baseline	1	8	273	8	1×	2,238	2,951	0	10
Speculative	112	1	268	1	7.9×	34,267	40,309	3,066	80
RISC-V CPU									
Baseline	1	2	287	2	1×	2,047	1,293	0	4
OpStall	12	1	248	1.3	1.3×	2,634	1,773	0	4
Kernels `dhrystone`				1.29	1.34×				
Kernels `matmul`				1.18	1.47×				
Kernels `median`				1.33	1.31×				
Kernels `gcd`				1.37	1.26×				
RegStall	37	1	256	1.3	1.4×	2,763	1,971	0	4
Kernels `dhrystone`				1.35	1.33×				
Kernels `matmul`				1.22	1.47×				
Kernels `median`				1.34	1.33×				
Kernels `gcd`				1.37	1.30×				

The results are provided in Figure 5.1. The first columns summarize the performance results, where II is the Initiation Interval achieved by the HLS tool and $F_{\max}$ is the maximal frequency of the generated hardware. The third column represents the average number of cycles needed to execute one loop iteration (CPI). The speed-ups displayed in the table summarize the performance improvement w.r.t. the baseline implementation, considering both the maximal frequency and number of cycles per iteration. The last values of the table represent the area cost of the different solutions.

The following sections describe each use case and explain how speculative execution helps improve the resulting accelerator's performance.

5.5.1 HMM-Based Sequence Comparison

The HMMER software package is used for profile HMM searches in biological sequence databases. Its kernel exposes a dependency pattern preventing parallelization using static techniques, as illustrated in the upper-right part of Figure 5.6. However, it is possible to speculate with very high confidence (99.9%) over the outcome of a *max* operation that drives `test`. This speculation removes the critical dependency, enabling wavefront parallelization [353]. Whenever a misspeculation occurs, the last safe state is restored, and execution resumes when the correct value is available. Our flow can take advantage of this speculation opportunity and automatically generate a fully parallel speculative accelerator. Results show that the speculative version outperforms the non-speculative one by up to 6.4× for a

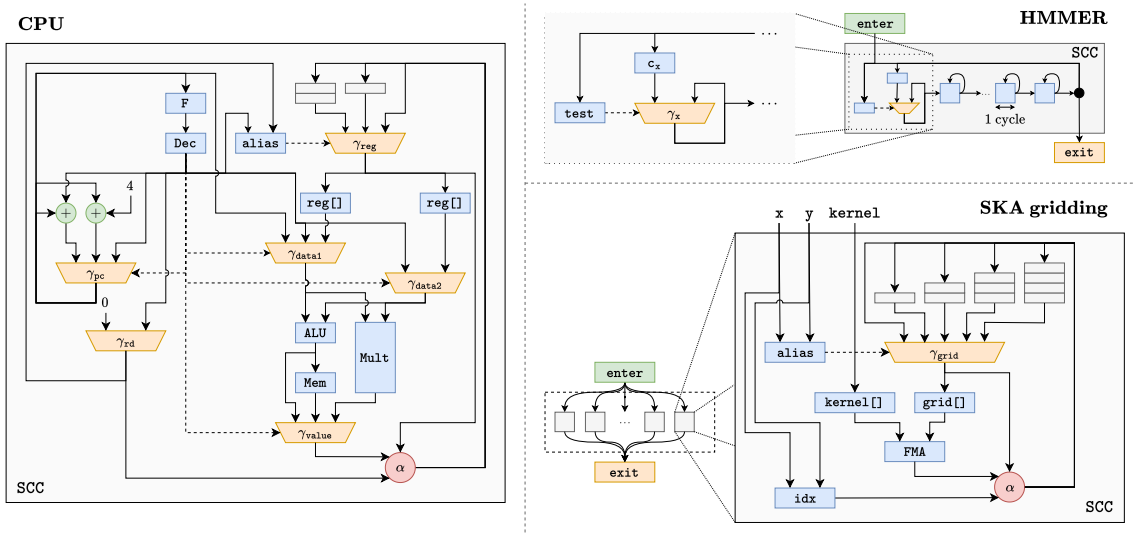


Figure 5.6 – Simplified view of the internal representation of the different use-case applications.

profile of size $M = 16$, at the price of a 2× increase in LUT count.

5.5.2 Gridding in the Square Kilometer Array

The Square Kilometer Array (SKA) aims at becoming the world's largest radio telescope. SKA relies on compute-intensive image gridding and de-gridding stages, where velocity measurements are back-propagated to reconstruct sky images [374].

The gridding algorithm's Gated SSA program IR is summarized in the lower-right part of Figure 5.6. It consists of two parallel loops (both with low trip counts), which perform the parallel updates of independent pixels, as illustrated in Figure 5.6. However, the pixel update operation involves a data-dependent Read-After-Write (RAW) loop-carried dependency for array `grid`, which prevents static pipelining.

We use the ability of SpecHLS to perform memory speculation across multiple iterations, extending the technique described in previous work [97]. Memory speculation is achieved by inserting runtime alias checks whose role is to stall the computation in case of an alias, enforce the loop-carried dependency, and speculate that no such alias occurred. A generalized framework for memory speculation based on a similar idea is exposed in Chapter 6. The bottom-right part of Figure 5.6 represents the alias check mechanism for the pixel update operation. We stall the pipeline whenever the RAW reuse distance is lower than four iterations to accommodate the FMA latency and achieve II = 1.

As shown in Figure 5.1, this approach improves performance by 8×. However, since every pixel update process operates independently, control logic and FIFOs must be duplicated, significantly impacting the design's surface area, especially regarding LUT and FF usage.

5.5.3 In-order Pipelined RISC-V CPU

Several accelerator platforms are based on many-core architectures built from simple in-order CPU micro-architectures with specialized instruction sets. These CPUs are designed at the RTL level and are difficult to evolve or customize.

This use-case shows that our flow can automatically infer an in-order pipelined micro-architecture for the RISC-V RV32IM instruction set, starting from an Instruction Set Simulator (ISS) model entirely written in C. The simulator's IR is depicted in the leftmost part of Figure 5.6, where γ-nodes expose speculation opportunities. Note that this representation is obtained by manually tuning the ISS to expose only speculations that we consider useful in this instance and manually inserting memory

speculation. Chapter 7 shows how to automate the transformations required to synthesize RISC-V processors from an ISS model.

As shown in Figure 5.1, pipelining the baseline ISS leads to II = 2, due to the loop-carried dependency over `pc` and `reg[]`. We use our flow to explore the set of possible speculation strategies in two cases: with memory speculation enabled (RegStall design) and without (OpStall design) and pick the best solution as identified by our toolchain. OpStall speculates over $\gamma_{\texttt{pc}}$ that no branch is taken and stalls the pipeline whenever a multiplication is issued. RegStall stalls in the case of a RAW dependency over a register.

Contrary to previous work on SLP [97], our approach allows for fine-grain speculation resolution by separately handling the speculations that occur in the pipeline. We avoid resolving all conditions in the execute stage of the processor's pipeline, thereby shortening this stage and improving the overall design frequency. The performance of each design was evaluated over four kernels (`dhrystone`, `matmul`, `median` and `gcd`). The results in Figure 5.1 show that SpecHLS improves performance for the OpStall configuration. The RegStall configuration shows a slight performance drop that we expect to be an artifact of the way the HLS backend handles our memory speculation code. Chapter 6 addresses those limitations.

5.5.4 Discussion

The three use cases discussed in the previous sections demonstrate that speculative execution can improve the performance of hardware accelerators, even when the kernel already exposes iteration-level parallelism. However, this technique would provide only limited performance improvements if applied to standard HLS benchmarks. The reason is that current benchmarks focus on computation patterns already well-supported by existing HLS tools [149, 421].

Our work is specifically designed to address kernels that pose challenges for standard HLS tools. Speculative pipelining is a powerful optimization that has the potential to unlock numerous unforeseen applications. However, there are several practical issues that need to be resolved. For instance, there is a need to fully support design space exploration when combining speculations. We present an efficient algorithm tailored for such exploration tasks in Chapter 7. Our approach also raises several verification issues, and we emphasize the need for a formal equivalence-checking technique for our transformations. The latter is left as future work.

The SKA gridding and RISC-V processor examples described in previous sections employ memory speculation to achieve better performance. The next chapter focuses on the automated insertion of such memory speculation mechanisms in SpecHLS.

CHAPTER 6

Memory Dependency Handling

The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka!' but 'That's funny...'
— *Isaac Asimov* (The Roving Mind)

In the previous chapter, we show that speculative execution can be inferred from a high-level behavioral description of hardware in C++ by leveraging compiler transformation and analysis techniques. Speculative HLS is a step towards bringing high-performance optimizations previously restricted to general-purpose computing to accelerator design. With its high customization potential, speculation in HLS could bring significant performance improvements with a relatively small hardware overhead. In this chapter, we focus on the issue of memory dependency handling. We give an intuition of memory speculation in Section 6.1 by showing how we can re-cast a memory dependency speculation problem to a control-flow speculation one. We then present some background and related works in Section 6.2, where we focus on data hazards in HLS designs. Section 6.3 introduces a formalized representation of memory accesses in a speculative HLS context and shows how the problem of speculating on memory accesses can be re-cast as a control-flow speculation problem in a general context. Consequently, the techniques described in Chapter 5 can be reused for memory speculation. We introduce a unified memory speculation framework that allows aggressive scheduling and high-throughput accelerator synthesis in the presence of complex memory dependencies. We discuss code generation in Section 6.4, where we expose a code-centric view of the memory speculation infrastructure and how we generate code for the SpecHLS backend. Finally, we show that our technique can generate high-throughput designs for various applications in Section 6.5.

6.1 The Path to Memory Speculation

Control-flow speculation enables many optimization opportunities in the context of HLS, as demonstrated in Chapter 5. It lifts the limitations imposed by static and dynamic scheduling, unlocking maximum performance in a wide variety of scenarios that are hard to deal with efficiently in standard HLS toolchains. However, control-flow speculation relies on the presence of *control-flow decisions* with a fast and a slow path to exploit during speculative scheduling. Therefore, the performance of some standard HLS example programs cannot improve solely with the speculation mechanisms described in the previous chapter since they do not exhibit any conditional execution pattern.

A typical example is the classical weighted `histogram` program shown in Figure 6.1. The latter is often used as an example of a loop that is hard to pipeline for standard HLS toolchains. Indeed, the inter-dependency Read-after-Write dependency on the `out` array constrains the pipelined schedule to

```
1  void histogram(int *out, const int *w, const int *idx, int n) {
2    for(int i = 0; i < n; ++i) {
3      int current_idx = idx[i];
4      out[current_idx] = out[current_idx] + w[i];
5    }
6  }
```

Figure 6.1 – Sample code for a simple weighted histogram.

```
1  void histogram(int *out, const int *w, const int *idx, int n) {
2    int old_idx = -1;
3    for(int i = 0; i < n; ++i) {
4      int current_idx = idx[i];
5      if(old_idx == current_idx) {
6        out[current_idx] = out[current_idx] + w[i]; // RAW dependency, slow path
7      } else {
8        out[current_idx] = out[old_idx] + w[i];     // No dependency, fast path
9      }
10     old_idx = current_idx;
11   }
12 }
```

Figure 6.2 – Code for a simple weighted histogram with an explicit fast and slow path. While the semantics of the code are exactly the same as in Figure 6.1, and such a transformation is likely to be reverted by any decent C or C++ compiler, this version exhibits explicit fast and slow execution paths in a conditional statement, which makes it amenable to SLP.

wait for the write operation to complete before any read can be executed. It follows that the minimum achievable II for the `histogram` loop must be at least equal to 2. The fundamental issue in this example is that it is impossible for the HLS toolchain to statically determine if there is any *aliasing* between accesses to `out` in successive iterations of the loop, *i.e.* if the location that is written to in a given iteration is read in the next one. We note that there are two possible scenarios in this case. If there is an alias, then reads from `out` need to wait for the array contents to be updated. Otherwise, the read can proceed immediately and load a value from the array before the update. We can exhibit these two paths in the code of Figure 6.1, as shown in Figure 6.2. The semantics of the code in the latter figure are strictly the same as the one in Figure 6.1, but there is now an explicit fast and slow path in the code controlled by a conditional statement. This pattern is all that is required to apply Speculative Loop Pipelining (SLP) to the `histogram` loop, making this example amenable to speculative execution.

The `histogram` example outlines the main intuition behind speculation on memory dependencies. By exposing a slow path for the aliasing case and a fast path for the non-aliasing case, we can leverage the control-flow speculation techniques described in Chapter 5 to speculate on memory dependencies. However, the transition from scalar values to arrays brings along a new set of challenges. Most importantly, we need to automate the insertion of alias checks for intra-and inter-iteration memory dependencies and adapt our rollback logic to arrays. While the by-value semantics of α-nodes in the GSSA intermediate representation of SpecHLS is convenient for high-level analysis, it is highly impractical for actual hardware generation. While the resulting circuit would be correct and behave appropriately in case of mispeculations if such value semantics were preserved for code generation, the synthesized hardware would be far from optimal. Each time a value may be written in an array, it would require array copy operations, and the cost of storing array states for potential rollbacks in case of a mispeculation would be high. The main goal of the techniques described in this chapter is to handle memory accesses in

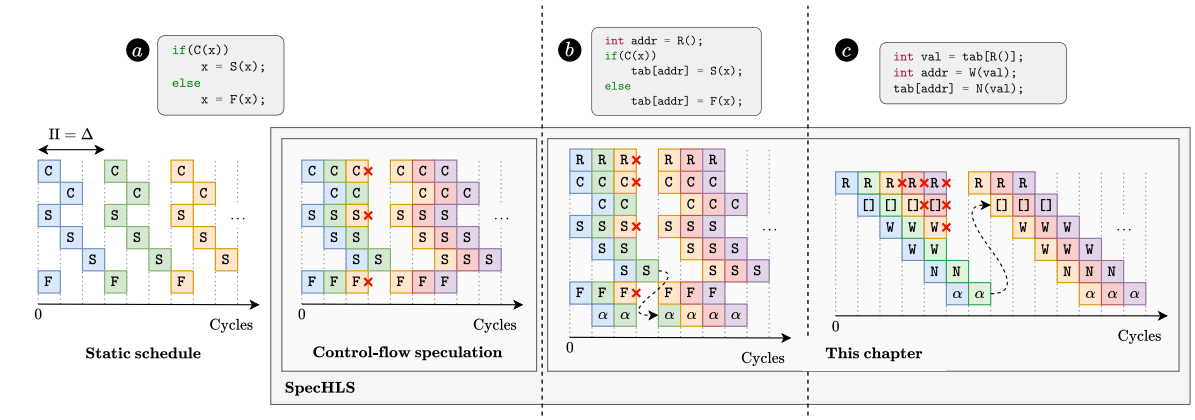


Figure 6.3 – Unified memory dependency handling in speculative HLS allows for transparent handling of array accesses in a speculative context. Part (a) illustrates control-flow speculation, on which we focus in Chapter 5. Part (b) and part (c) show the schedules obtained using the techniques presented in this chapter in the presence of memory accesses and memory dependencies. The vertical axis represents pipeline stages, and the horizontal axis represents execution cycles. `F` is a *fast* operation (1 cycle), `S` is a *slow* operation (3 cycles), and α denotes an array update.

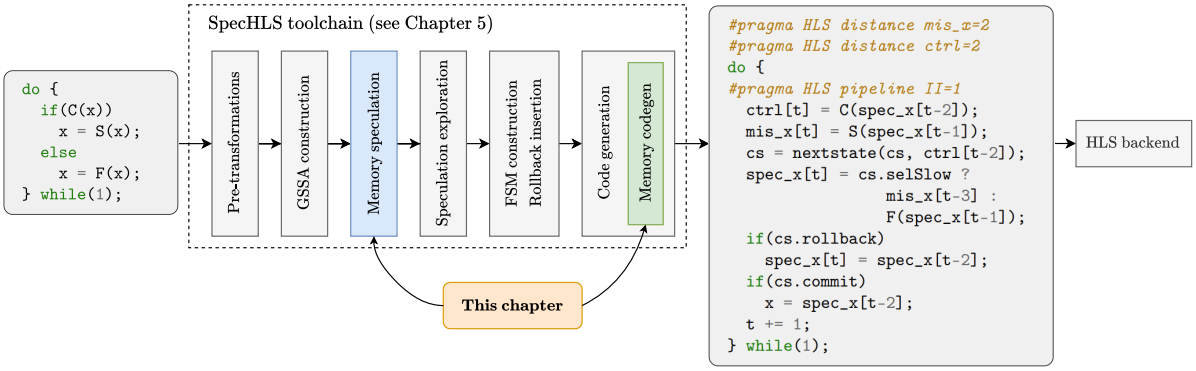


Figure 6.4 – Extensions to the SpecHLS flow to support memory speculation. This chapter introduces a memory speculation pass and its accompanying code generation extension.

a speculative context using a unified framework, as depicted in Figure 6.3. We achieve this goal by focusing on critical parts of the SpecHLS flow shown in Figure 6.4.

6.2 Background and Related Work

Speculation is a fundamental aspect of high-performance computing that has implications for compiler and low-level hardware design. Some specialized compiler frameworks leverage coarse-grain speculative execution by partitioning programs into sets of speculative threads [36, 303, 371]. This behavior leads to memory hazards between threads that must be resolved at runtime. Speculative Decoupled Software Pipelining (SpecDSWP) supports speculative memory rollbacks by leveraging a versioned memory [371]. All these approaches leverage speculation on existing hardware, while our approach is aimed at custom hardware synthesis.

Besides the work described in this thesis, several research endeavors have applied varying degrees of speculation to High-Level Synthesis flows, from automatic branch prediction synthesis [157, 216] to generalized speculative execution [186, 97]. The latter often incorporate some form of limited memory speculation. Josipović *et al.* [186] insert *save* and *commit* operators at the boundary of dataflow circuit

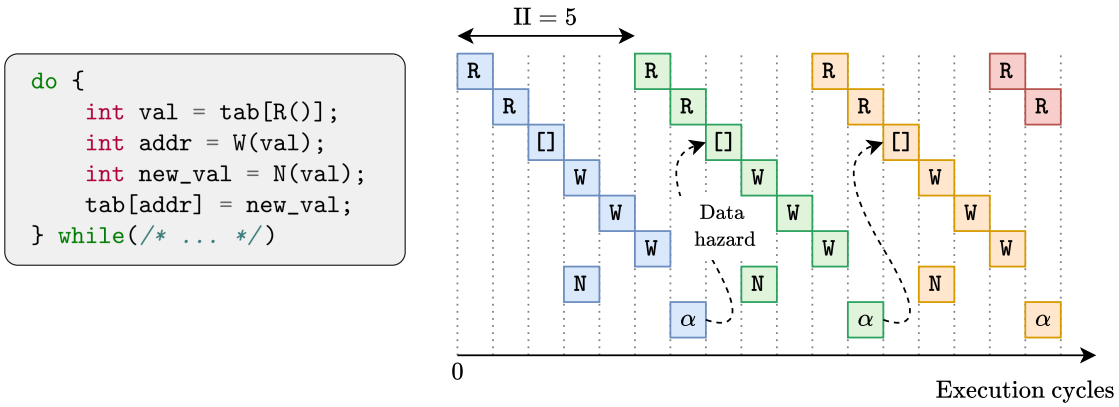


Figure 6.5 – Data-dependent memory access, along with its statically inferred execution schedule. An inter-iteration memory dependency (dotted arrow) prevents efficient overlap of iterations through standard loop pipelining.

regions where speculative computations are to take place. These regions always end before any *store* unit can interact with memory, and no speculative token can escape to memory before the result of the computation is committed. Derrien *et al.* [97] only handle simple read-after-write loop-carried dependencies in a speculative context. They use a technique based on the work of Alle *et al.* [8] for runtime alias detection insertion. Furthermore, their approach only allows for a single array update for each array in a given loop iteration.

Dynamic memory hazard resolution in HLS is an active research area, as it provides a broader range of applications to HLS. Alle *et al.* [8] use source-to-source manipulations to insert runtime alias detection and transform loops with loop-carried dependencies, allowing them to be pipelined by HLS toolchains. Dai *et al.* [89] insert dynamic hazard resolution logic at compile-time and handle data hazards at runtime through memory port arbitration and squash-and-replay. Several authors have also leveraged polyhedral compilation techniques to insert compile-time/runtime alias detection logic and stall the execution pipeline when an alias occurs [264, 233, 232]. Some of the latter approaches tend to duplicate hardware to handle memory dependencies [233, 232], whereas our approach's area overhead is limited to speculation and alias detection logic.

The issue of store queue sizing in dataflow circuits has been covered by Liu *et al.* [231] as well as Elakhras *et al.* [109]. The latter leverage basic-block analysis to tune the size of the load-store queue inserted into their dataflow designs. This approach has only been shown to be applicable in the context of *dynamic* dataflow circuits. Speculative execution would likely require additional logic in the store queue, especially to handle multiple interacting speculations, which are only briefly discussed by previous work on dataflow circuits [186].

6.3 Speculating on Memory Dependencies

Speculating on memory dependencies allows data hazards to be handled efficiently, leading to tight operator scheduling in the final hardware. This section starts by describing memory dependency handling in standard HLS flows (Section 6.3.1) before illustrating the impact of runtime memory disambiguation logic on the throughput of generated hardware (Section 6.3.2). Section 6.3.3 shows how we can integrate memory dependency information into our intermediate representation, and Section 6.3.4 describes our memory speculation framework. Finally, Section 6.3.5 presents our complete memory speculation mechanism, which can handle RAW, WAR, and WAW dependencies.

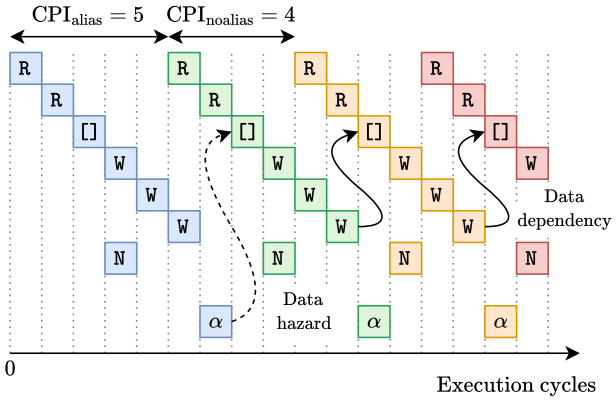


Figure 6.6 – Execution schedule of the example code in Figure 6.5 with runtime memory disambiguation. The load from memory depends on the result of the address computation `W`, thereby limiting the minimum achievable II.

6.3.1 Memory Dependency Handling in HLS

Data hazards can severely impact the scheduling efficiency of standard HLS tools. When a possible memory alias is detected in a loop body, the hardware needs to be synthesized while considering the worst-case scenario, similarly to how instructions need to be scheduled for the worst case in a VLIW compiler backend [215]. This worst case can have a tremendous impact on the value of II that can be achieved by a hardware design since the hardware must assume that an alias can happen at every iteration of the loop. Figure 6.5 illustrates a simple data-dependent memory access example that leads to poor hardware synthesis results. The dependency limits the pipelining capabilities of the HLS toolchain, especially in the presence of non-negligible address computation delays. The bottom part of Figure 6.5 illustrates the execution schedule that can be inferred by HLS, assuming that `R`, `W`, and `N` take respectively two, three, and one cycles to execute. We denote memory loads by `[]` and memory writes by α and further assume that memory accesses take one cycle. We observe that the loop pipelining transformations manage to partially overlap the execution of consecutive iterations of the loop. However, the memory dependency denoted by the dotted arrow in Figure 6.5 prevents further overlapping because of the potential data hazard if the read and write addresses are the same.

Modern HLS toolchains use static alias analysis passes to identify memory dependencies inside loops. This approach can sometimes help identify false dependencies with advanced analyses and transformations based on polyhedral compilation techniques [299, 264]. Additionally, HLS users can override alias analysis results with code annotations. The Vitis HLS `pragma dependence` is an example of such annotation, which allows the user to specify the type of dependency (RAW, WAR, or WAW) and the inter-iteration distance between potentially aliasing array accesses [10]. These approaches can cover various memory dependency scenarios but cannot account for situations where the dependency distance cannot be statically bounded. Our approach handles the latter case as well as the former one, leading to higher throughput in the generated hardware at a small area overhead cost.

6.3.2 A Note on Runtime Memory Disambiguation

Existing work has focused on runtime memory access disambiguation [8, 264, 232, 186, 109]. The latter avoids long memory accesses when possible. However, the address computation latency may still hinder performance even with runtime alias detection. An example is given in Figure 6.6, with an alias occurring only between the first and second iterations of the loop. The disambiguation logic leads to a dynamically scheduled execution [184] and an improvement in the number of cycles per iteration (CPI). Suppose data hazards represent a proportion $p < 1$ of memory accesses during execution. In that case, we now have $\text{CPI} = \text{CPI}_{\text{alias}} \times p + \text{CPI}_{\text{noalias}} \times (1-p) = 5p + 4 \times (1-p) < \text{CPI}_{\text{alias}} = 5$. In the following,

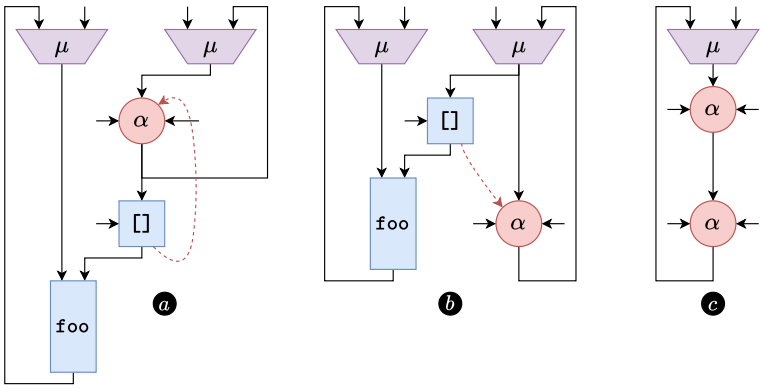


Figure 6.7 – Memory dependencies in extended Gated-SSA. The dotted arrow represents a load-store dependency that needs to be honored during code generation. The immutable array representation already ensures store-store ordering.

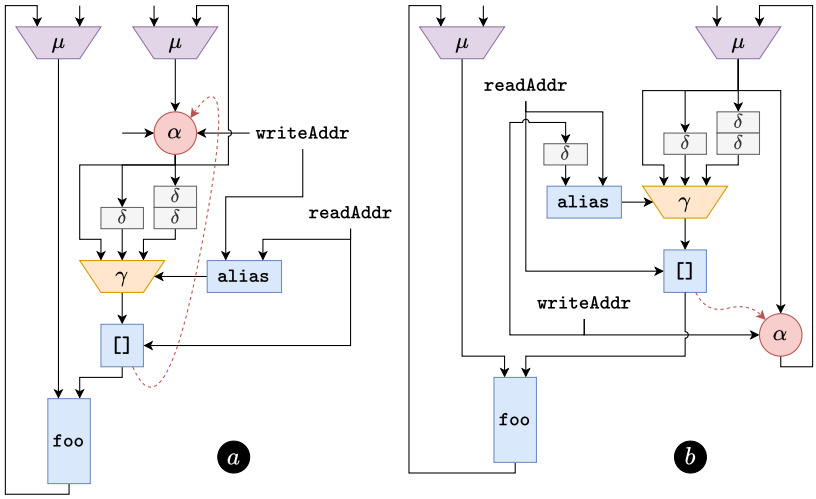


Figure 6.8 – Speculating on a Read-after-Write (a) (see Figure 6.7 (a)) and Write-after-Read (see Figure 6.7 (b)) dependencies. The shortest path exposes two δ-nodes.

we show how to improve the performance of HLS code that exhibits memory dependencies further by leveraging speculation.

6.3.3 Memory Dependencies in Gated-SSA

The GSSA representation that we use in SpecHLS provides limited support for memory operation ordering. For example, while store order is preserved through the immutable array representation, the relative order of stores and preceding loads is not preserved in the GSSA representation. This behavior can cause the generated circuit to misbehave if a Write-after-Read (WAR) dependency is violated. We extend our toolchain's intermediate representation to allow for the expression of explicit memory dependencies using non-dataflow edges in the IR graph (see Figure 6.7).

We note that this extension enforces strong load-store and store-load ordering, but not load-load ordering: loads that in a given order in the input code may be generated out-of-order in the transformed code. The reordering of load operations does not change the behavior of the circuit and allows us to not over-constrain operation scheduling.

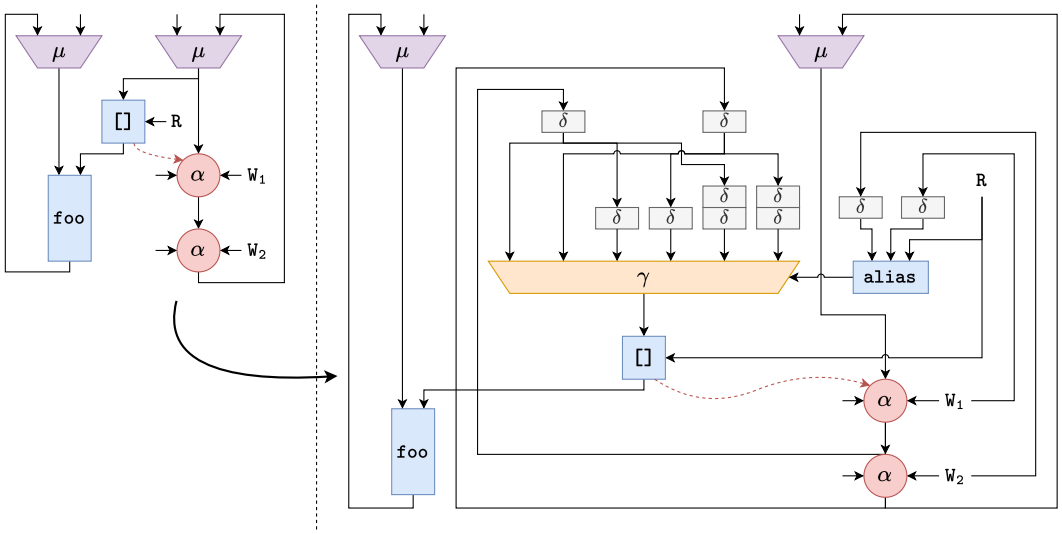


Figure 6.9 – Handling multiple array updates in the same loop iteration. The alias detection γ-node placed before the memory load handles aliases up to two iterations away with both α-nodes.

6.3.4 Exposing Speculation Opportunities

As described in Chapter 5, speculation opportunities are modeled in our IR as γ-nodes. The latter correspond to control-flow decisions, so we need to recast the memory dependency problem as a control-flow decision problem. This transformation corresponds precisely to runtime alias detection, as Section 6.1 outlines. We introduce a GSSA transformation that materializes runtime alias detection in the IR. Such a transformation aims to increase the dependency distance between iterations, allowing the HLS backend to synthesize deeper pipelines. The alias detection window depth w_d (*i.e.* the number of previous writes to check for aliases with) is determined through a combination of application profiling and design space exploration. The latter is discussed in Chapter 7.

We start by considering the case of intra-iteration Read-after-Write dependencies. Before each load, we insert a γ-node that operates on the array *value* from which we would like to read (Figure 6.8 ⓐ). The inputs to the alias detection γ-node are increasingly delayed versions of the array (*i.e.* from the current and previous iterations of the loop), and the decision is controlled by an `alias` node. The latter manages an internal buffer of addresses whose size will be determined during the code generation phase (see Section 6.4). In a simple example, such as the `histogram` kernel described in Section 6.1, the `alias` node is simply an equality comparison between the read and write addresses. In case of an alias, then the `alias` node selects the non-delayed input of the γ-node, stalling the load operation until the α-node completes its operation. Otherwise, if there is an alias $n \geqslant 1$ iterations before the load, then the input of the γ-node with n δ-nodes is selected.

Selecting an array value from a sequence of n δ-nodes can be seen as ignoring the last n stores to this array. A direct consequence of this observation is that the alias detection γ-node exhibits a slow path on its non-delayed input, and increasingly fast paths on its delayed inputs. This imbalance in path length through the γ-node naturally lends itself to speculation [97]. We speculate that there are no aliases, selecting the most delayed version of the array. If there was an alias $i < w_d$ iterations ago, we roll back the computation and select the i-th input of the γ-node. Since we speculate, we avoid the issues with runtime memory disambiguation latencies illustrated in Figure 6.6: a new loop iteration can start every cycle. The mispeculation handling logic takes care of rolling back any potentially erroneous values. This process is handled transparently by the speculation insertion mechanism in SpecHLS.

Let us now consider the case of intra-iteration Write-after-Read dependencies. Similarly to the RAW dependency case, we insert an alias detection mechanism before each memory load. The main difference with the RAW example described above is that write addresses must be delayed before entering the

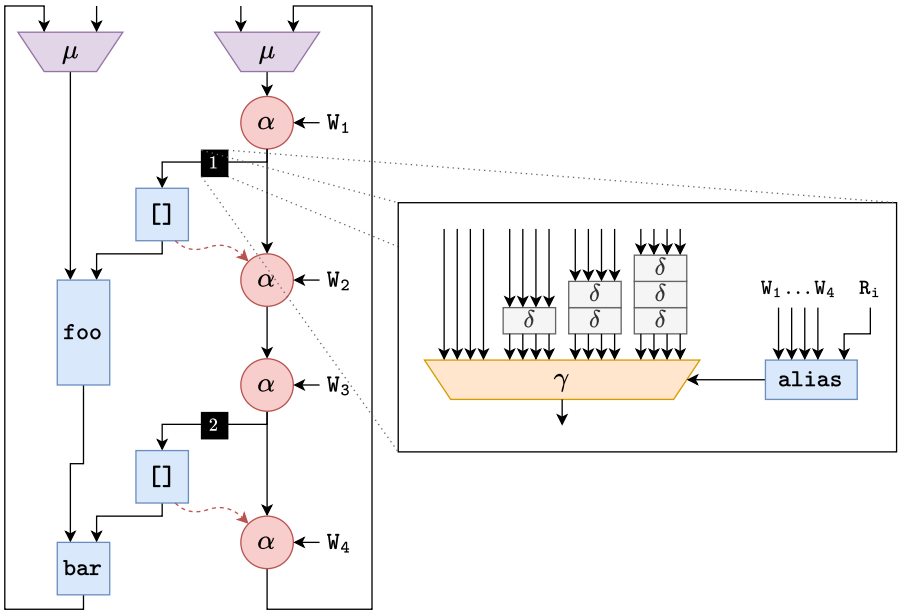


Figure 6.10 – Generalizing the memory speculation mechanism to account for multiple memory dependencies. δ-nodes on the inputs of the alias detection γ-node are grouped for readability purposes.

alias node. Figure 6.8 (b) illustrates this situation.

Finally, let us consider a more complete example that exhibits both a Write-after-Write and a Write-after-Read dependency. The left of Figure 6.9 represents the initial GSSA representation of our input program. `R`, $\texttt{W}_1$, and $\texttt{W}_2$ are the load address and the two write addresses, respectively. The right part of Figure 6.9 shows the GSSA representation after applying our memory speculation transformation pass. The alias detection γ-node is placed before the memory load and handles aliases up to two iterations away with both α-nodes. Any Write-after-Write dependency is encoded in our GSSA representation by an edge between α-nodes. This edge is considered by the code generation phase (Section 6.4) to keep array updates in order.

6.3.5 Generalizing Memory Speculation

The examples detailed in Section 6.3.4 form the basis of SpecHLS's memory speculation framework. They are the foundation from which we derive a mechanized procedure to insert memory speculation opportunities in any input code. The main idea behind this procedure is given by Figure 6.10 and is described in the remainder of this section.

In the following, $n_\alpha(s)$ denotes the number of α-nodes that operate on s in the GSSA representation. For each load l_s from s, $P_\alpha(l_s)$ (resp. $S_\alpha(l_s)$) denotes the set of α-nodes that precede (resp. follow) l_s. Once all memory dependencies are explicitly represented in GSSA, we traverse our IR, looking for memory load operations for each array symbol s in our input program. We insert the alias detection logic before each load l_s from s. The structure of the alias detection mechanism can be summarized as follows:

— the `alias` node has $n_\alpha + 1$ entries, one for l_s's read address, and one for each write address in the SCC. Write addresses from nodes in $P_\alpha(l_s)$ are directly linked to the `alias` node, while addresses from nodes in $S_\alpha(l_s)$ are connected through a δ-node (not shown in Figure 6.10).

— the γ-node has w_d groups of inputs, with each successive group adding a δ-node to its inputs. The latter inputs correspond to the array values produced by all the α-nodes operating on s in the SCC. Similarly to the write addresses, additional δ-nodes are inserted for array values produced by nodes in $S_\alpha(l_s)$.

6.4 Code Generation

Previous sections have shown how we can introduce a generalized memory speculation mechanism in the SpecHLS toolchain. In this section, we discuss the code generation aspect of memory speculation, which requires special care to make it amenable to efficient synthesis by commercial HLS toolchains. Section 6.4.1 starts by highlighting the semantic gap between the by-value array semantics we have been working with up to this point and the HLS-friendly semantics we need to generate for our backend. Section 6.4.2 then focuses on our store queue parameter inference pass, while Section 6.4.3 gives an overview of the code we generate to handle speculation on memory accesses.

6.4.1 Code Generation for Memory Speculation

For our speculation flow to generate efficient hardware, we need to lower the level of abstraction at which we manipulate arrays. So far, arrays have been considered immutable values, with α-nodes producing a new updated value after each store operation. While this representation is convenient for high-level transformations, it is impractical for realistic hardware synthesis. In the code generation phase, we lower this abstraction to a set of store queues and an underlying array. This step represents an important shift in perspective when considering arrays, as they go from a localized state to a global state that needs to be managed across the entirety of the loop we are trying to pipeline.

Strong store-load and load-store ordering. Our approach requires the relative order of stores and loads in the intermediate representation to be preserved. The code generation operates on a global state, so we require a consistent update order to maintain coherency between the array contents updates and array reads.

The following section highlights the challenges that arise from such a shift of perspective, as we need to guarantee that (i) a read from the global state has to reflect the previously localized state of the array and needs to take into account potentially pending stores to the array that may not have been committed yet; (ii) a write to the array needs to update the pending write buffer while waiting for the confirmation that the value to be written is correct; (iii) a value is committed to the global state only if is correct, and the array contents (as well as all auxiliary data structures) are rolled-back to their previous state in case of a mispeculation.

6.4.2 Inferring Store Queue Parameters

In our memory dependency handling framework, stores to memory are handled by a set of partial store queues. Each partial store queue holds a list of pending values to be stored and a list of their corresponding addresses, with a valid bit for each entry. The size of each list, `maxPendingStores` and `maxPendingAddr`, are computed by a procedure described in the following. We create a separate partial store queue for each α-node that operates on the array, with the union of all these partial queues forming a store queue structure similar to the one found in processors. Doing so allows us to have fine-grain control in the alias detection while still making it easy for the HLS toolchain in the backend to optimize our code.

A store window parameterizes the partial store queues to ensure temporal consistency during the execution (see Section 6.4.1). The size of each partial store queue and the window parameters are determined by a depth propagation pass, which we chose to encode as a type inference pass on our IR. In the following, we consider the case where there is only one α-node for each array in the input code (Section 6.4.2) before showing how to generalize our approach to multiple array updates (Section 6.4.2).

Single Array Update

We denote the type of an array with elements of type τ as $\mathcal{A}(\tau)$. We add two additional parameters to this type, w and d, to encode the length of the store queue and the end of the store queue window

(*i.e.* the number of values to *discard* from the end of the store queue) that is valid at each node. Consequently, if a node N has type $\mathcal{A}(\tau, w, d)$, then all stores in the queue in the range $[0; w - d]$ are *live* when executing N.

Algorithm 1 Store queue depth inference.

```
maxRBPending ← max_{r∈RB} depth(r)
for all sym array symbol do
  maxPendingStores[sym] ← maxRBPending + 1
  maxPendingAddr[sym] ← max(Δ_U, maxRBPending + 1)
  worklist ← {μ | type(μ) = SpecArrayType}
  while !worklist.empty do
    for n ∈ worklist do
      if n is μ-node or ∀p ∈ predecessors(n), visited(p) then
        propagate(n)
        worklist = (worklist\n) ∪ {s | s ∈ successors(n) ∧ ¬visited(s)}
      end if
    end for
  end while
end for
```

The store queue depth inference pass is a forward analysis along dataflow edges, ignoring the back-edges on μ-nodes, as shown in Algorithm 1. The maximum depth of the store queue is computed as one plus the maximum of all rollbacks that can occur for a given array. Intuitively, the depth of the rollback operators is given by the length in cycles of the computations that determine if a speculation is correct. During the execution of such an operation, values may be written to the store queue at each execution cycle. These values may be invalid if the current speculation happens to be incorrect. Thus, we need to keep at least $\texttt{maxRBPending} + 1$ values in the store queue at each time since a mispeculation may require rolling back at most `maxRBPending` values in the queue and restoring the previous state of the array. A direct consequence of this definition is that the *oldest* value in the store queue, if valid, can be committed to the underlying array since it will never be rolled back.

The store queue depth inference pass operates on a worklist of nodes, starting with all μ-nodes in the SCC corresponding to arrays. The entire procedure traverses the intermediate representation exactly once, propagating store queue parameters in the `propagate` function. The latter operates according to the following inference rules:

$$\text{Mu}\ \frac{x : \mathcal{A}(\tau) \quad p : \texttt{bool} \quad i_x : \mathcal{A}(\tau)}{\mu(p, i_x, x) : \mathcal{A}(\tau, \texttt{maxRbPending} - 1, 0)}$$

$$\text{Alpha}\ \frac{x : \mathcal{A}(\tau, w, d) \quad i : \texttt{int} \quad e : \tau}{\alpha(x, i, e) : \mathcal{A}(\tau, w + 1, d)}$$

$$\text{Gamma}\ \frac{c : \texttt{bool} \quad e_0 : \mathcal{A}(\tau, w_0, d_0) \ \ldots \ e_n : \mathcal{A}(\tau, w_n, d_n)}{\gamma(c, e0, \ldots, e_n) : \mathcal{A}(\tau, \max_i w_i, \min_i d_i)}$$

$$\text{Mux}\ \frac{c : \texttt{bool} \quad e_0 : \mathcal{A}(\tau, w_0, d_0) \ \ldots \ e_n : \mathcal{A}(\tau, w_n, d_n)}{\text{mux}(c, e0, \ldots, e_n) : \mathcal{A}(\tau, \max_i w_i, \min_i d_i)}$$

$$\text{Delta}\ \frac{x : \mathcal{A}(\tau, w, d)}{\delta(x) : \mathcal{A}(\tau, w, d + 1)}$$

$$\text{Rollback}\ \frac{x : \mathcal{A}(\tau, w, d)}{\text{RB}(x) : \mathcal{A}(\tau, w, d)}$$

The store queue window is initialized at μ-node, where the depth inference pass starts. When encountering an α-node, an additional pending store is added to the store queue window. Multiplexers

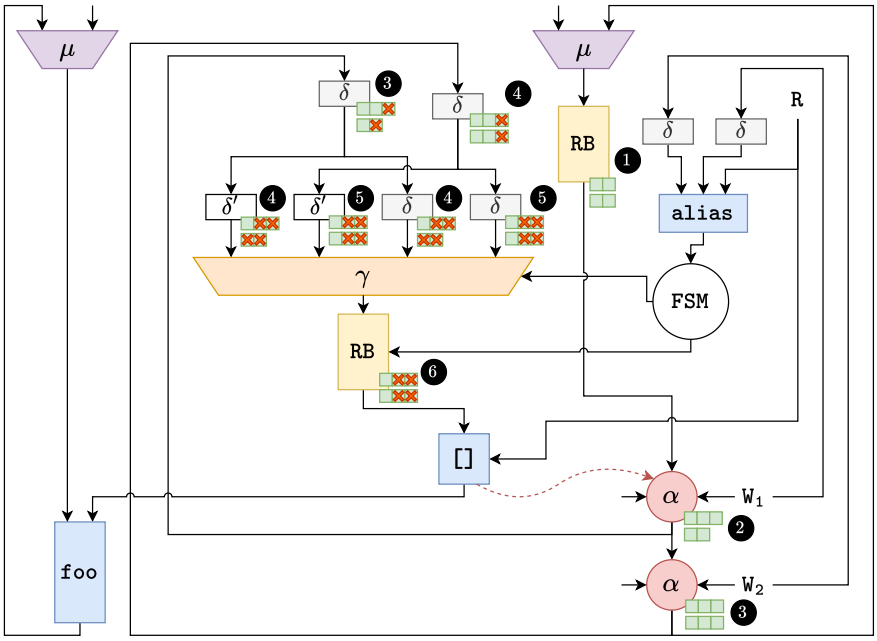


Figure 6.11 – Store queue structure inference pass. The circled numbers show the successive steps followed by the inference algorithm. Delays marked as δ are inserted by our memory speculation algorithm, while those marked δ' are the ones inserted by SLP.

and γ-node are treated similarly: we need to consider the most pending stores from all of the node's inputs and discard the least amount of elements. The latter rule encodes the interval union of all input store queue windows. δ-nodes add a discarded store, as they delay the execution by one cycle: valid stores before a delay will only be visible at the next cycle for operators dominated by the δ-node. Finally, rollbacks pass the type of their input to their output, as they only affect the maximum size of the store queue windows. These inference rules cover all the node types that may interact with arrays in our IR, except for loads.

Multiple Array Updates

The type inference pass presented in the previous section can be extended to multiple array updates by replacing the type parameter w with a list, where each element of the list corresponds to a different α-node. Only the Alpha inference rule needs to be updated, replacing it by

$$\text{Alpha}\frac{x : \mathcal{A}(\tau, w, d) \quad i : \texttt{int} \quad e : \tau}{\alpha_j(x, i, e) : \mathcal{A}(\tau, w[w_j \mapsto w_j + 1], d)},$$

with j the index of the α-node, and $w[w_j \mapsto w_j + 1]$ denoting the in-place update of the j-th element of w.

Figure 6.11 illustrates the store queue parameter inference pass on an example involving a load and two stores. The store queue windows are represented for each α-node by a sequence of squares. The output type of a node is attached to its lower-right corner. The length of the sequences corresponds to the value of the elements of the w type parameter list. Crossed-out elements in those windows represent the d parameter. Note that there are two sets of δ-nodes at the input of the γ-node: the first set corresponds to the delays inserted by our memory speculation transformation (marked δ). The second corresponds to delays inserted by the Speculative Loop Pipelining pass described in Chapter 5. Recall that the latter are inserted on slow paths at the input of γ-nodes. In this case, reading from a delayed array is fast since we avoid waiting on the last store to read the array. Consequently, paths

```
struct spec_addr {
  int address;
  bool valid;
};

int x_pending[NA][MP];
int x_pending_addr[NA][MA];
bool x_pending_valid[NA][MA];

template <int A, int MP, int MA>
int* update_x(int *x, int value, bool we, int address) {
  #pragma HLS inline
  #pragma HLS array_partition type=complete variable=x_pending
  #pragma HLS array_partition type=complete variable=x_pending_addr
  #pragma HLS array_partition type=complete variable=x_pending_valid
  {
    #pragma GCS unroll
    for (int i = 0; i < MP - 1; ++i)
      x_pending[A][i] = x_pending[A][i+1];
    #pragma GCS unroll
    for (int i = 0; i < MA - 1; ++i) {
      x_pending_addr[A][i] = x_pending_addr[A][i+1];
      x_pending_valid[A][i] = x_pending_valid[A][i+1];
    }

    x_pending[A][MP - 1] = value;
    x_pending_addr[A][MA - 1] = address;
    x_pending_valid[A][MA - 1] = we;
    return x;
  }
}
```

Figure 6.12 – Example C++ code generated for the array update operation on an integer array. `NA` denotes the number of α-nodes operating on array `x`, while the template parameters indicate the identifier of the α-node, the maximum number of pending writes, and the maximum number of pending addresses.

with fewer memory speculation delays are slow paths. The resulting IDG exposes two delays on all inputs of the γ-node, which absorb the latency of the two α-nodes and enable pipelining with a unit initiation interval.

6.4.3 Code Generation Structure

For each array for which we inserted the speculation logic described in Section 6.3, we need to generate code for five different functions: `read`, `update`, `commit`, `rollback` and `alias`. They correspond respectively to array reads, α-nodes, commit nodes, rollbacks, and the alias detection logic from our speculative IR. These functions operate on plain C pointers representing arrays and interact with global store queue structures. The following lays out the latter structures and describes the operation of each function in more detail. We only discuss operations for a single array, but the code is easily extended to multiple arrays.

To read values from an array, we need to either load it from its contents or index into our store queue

structure if an update occurred at the current read address. This operation is achieved by providing our reading stub the value of the read address alongside the array pointer and a list of value ranges. The latter correspond to the value windows to consider in each partial store queue for the current read operation. We do not traverse the entire store queue to avoid data dependency violations, as some values from later store operations may already have been written to the store queue by speculative execution. The parameters of each store queue window are computed by the procedure described in Section 6.4.2. The store queue traversals are all unrolled, and the reading logic is inlined at each call site to allow for maximum optimization potential for the HLS backend.

Array updates that happen through α-nodes in our intermediate representation are lowered to the code akin to the one in Figure 6.12. The update function is parameterized by the index of the α-node, `A`, for access to partial store queues, as well as the maximum number of pending values and addresses, `MP` and `MA`. Each α-node in our intermediate representation is replaced by a call to this function, passing its unique identifier as the first template parameter. We shift the contents of the store queue to make room for the newly written value and insert the latter in the partial store queue corresponding to the current α-node. The `we` function argument corresponds to a write-enable signal cleared by the speculation Finite State Machine (FSM) if a mispeculation is being handled and invalid values are propagating through the circuit. We leverage the GeCoS code transformation infrastructure through the `GCS` annotations to unroll the store queue shift loops. We annotate the function for the HLS backend: the update function will be inlined, and the `x_pending` and `x_pending_addr` arrays will be resolved to individual registers in the generated hardware, which will map them to fast-access memory [10].

Once the value computed by a speculative operation is deemed correct by the speculation logic, the contents of the partial store queues are committed to the underlying array. For each α-node in the code, we check if the oldest value in its partial store queue is valid (*i.e.*, the update function set its valid bit) and, if so, we write it to the array at the address held in the queue. As noted in Section 6.4.2, we can safely commit this value since it lies outside the range of potential rollback candidates for the array to which the store queue is attached.

When a mispeculation occurs in the SCC, incorrect data may have already been written to the array's store queue. This incorrect data can either be a value or an address computed by a speculative operation as a result of a wrong speculation. In such a case, our store queue can be rolled back by marking all stores that happened during the wrong speculative execution as invalid. This function is called directly by the speculative FSM.

The alias detection logic presented in Section 6.3.5 is rewired as follows during the code generation phase. The output of the alias node is linked to the FSM and informs the speculative execution process of potential mispeculations. Instead of operating on the write addresses, as illustrated in Figure 6.10, the final alias detection function operates on the partial store queues associated with all α-nodes. It checks if the read address of its corresponding load operator aliases with any store operation registered in the partial queues and produces an integer value indicating the distance (in terms of memory operations) at which the closest alias occurred. The address checks are laid out in a binary tree-like structure, with each tree level checking if an alias occurred in the lower or the upper half of a window in the partial store queues. This structure allows for efficient hardware generation and better execution frequency, as many comparisons can be parallelized. Since the alias detection function only informs the speculation decisions, it is allowed to be slightly pessimistic.

6.5 Experimental Results

Application-specific hardware accelerators often comprise loop kernels and do not encompass entire applications, making usual application-focused benchmarks such as SpecINT irrelevant. Although there exist HLS-specific benchmarks [149, 421], they are focused on kernels with regular access patterns, as they carry the legacy of standard HLS application domains.

To address these issues, we choose to evaluate our memory speculation framework on a selected set of benchmarks with data-dependent memory dependencies from previously published work [8, 97], as well as statically determinable but infrequent dependencies that prevent pipelining as discussed by Liu

Table 6.1 – Performance results for a set of benchmarks that exhibit dynamic or otherwise hard to handle memory dependencies.

	Baseline		Speculative				
Benchmark	II	$F_{\max}$ (MHz)	II	$F_{\max}$ (MHz)	CPI	**Misp. (%)**	**Speed-up**
SimpleRAW	3	101	1	101	1.2	6	2.5×
SimpleWAR	3	101	1	101	1.5	3	2.0×
SimpleWAW	7	101	1	101	1.7	9	4.1×
DoubleRAW	9	101	1	80	1.4	17	5.1×
Histogram	4	122	1	101	1.3	16	2.5×
SKA-Gridding	4	122	1	101	2.5	40	1.3×
ALU	2	110	1	99	1.3	15	1.4×
KulischAccum	3	197	1	141	1.2	3	1.8×
Floyd-Warshall	6	122	1	82	1.1	11	3.7×
Gauss	6	122	1	87	1.0	0	4.3×
BNN	5	145	1	137	1.1	10	4.3×

Table 6.2 – Area results for our benchmark set. We observe an average surface area increase close to the average speed-up presented in Table 6.1.

	Baseline			Speculative		
Benchmark	LUT+FF	BRAM	DSP	LUT+FF	BRAM	DSP
SimpleRAW	579	1	0	1584 (2.7×)	1	0
SimpleWAR	903	1	0	2035 (2.3×)	1	0
SimpleWAW	2955	0	0	5475 (1.9×)	0	0
DoubleRAW	1376	0	2	6380 (4.7×)	0	2
Histogram	723	0	2	2772 (3.8×)	0	2
SKA-Gridding	1170	0	2	1480 (1.3×)	0	2
ALU	608	0	0	1336 (2.2×)	0	3
KulischAccum	715	2	0	959 (1.3×)	2	0
Floyd-Warshall	1631	0	0	8192 (5.0×)	0	0
Gauss	994	1	5	4177 (4.2×)	3	3
BNN	330	2	0	2379 (7.2×)	2	0

et al. [232]. Standard HLS tools fail to find a satisfying operation schedule and produce hardware with II > 1 in both cases. *SimpleRAW*, *SimpleWAR*, *SimpleWAW*, and *DoubleRAW* are synthetic benchmarks that exhibit basic memory dependency patterns inside of a loop. *SimpleRAW* (resp. *SimpleWAR*, and *SimpleWAW*) is similar to Figure 6.7 ⓐ (resp. Figure 6.7 ⓑ, and Figure 6.7 ⓒ). *Histogram*, *SKA-Gridding*, *ALU*, and *KulischAccum* exhibit data-dependent memory accesses that cannot be determined statically. *Floyd-Warshall* and *Gauss* exhibit an inter-iteration memory dependency for certain iterations of the loop. Consequently, HLS tools must generate a pessimistic schedule while pipelining the loop. *BNN* contains both a data-dependent memory access and an inter-iteration memory dependency.

In all our benchmarks, SpecHLS inserts the minimum number of memory speculation delays required to achieve II $= 1$ in the speculative design. We use Vitis HLS 2021.2 to perform the High-Level Synthesis, with an XC7A200 as the target FPGA. Table 6.1 shows performance, and Table 6.2 shows area results for our benchmark set. The baseline (with no speculation applied) is shown on the left-hand side, followed by its speculative counterpart. Mispeculation rates and relative speed-ups are given at the end of each row. The CPI (Cycles Per Iteration) value that we give for the speculative design can be seen as an *effective* II. It is equal to II for the baseline and is linearly correlated with the mispeculation rate for the speculative version. The hardware utilization is illustrated using lookup tables and flip-

flops (LUT+FF), memory blocks (BRAM), and hardware multiply-add blocks (DSP). Note that some arithmetic operations can either be mapped to DSP or to LUT+FF elements, which explains some of our area results. For example, the speculative *Gauss* accelerator uses fewer DSPs than the baseline because some arithmetic operations have been mapped to LUT+FF by the HLS backend's heuristics.

We note that the proposed approach reduces the static II for each application. To measure the speed-up, we compare the CPI achieved by the baseline and the speculative versions of our benchmarks and the maximal frequency of the generated hardware. The additional control inserted to detect aliases and handle mispeculation can hurt the maximal achievable frequency. However, the important gains on the CPI side lead to a speed-up ranging from 1.3× to 5.1×. It is important to note that for synthetic benchmarks and data-dependent applications, the mispeculation rate depends on the data used and may affect the speed-up. However, the original SLP paper by Derrien *et al.* shows that even if we always mispeculate, the CPI is never worse than the pessimistic static schedule [97]. Only the $F_{\max}$ reduction may reduce the performance of the generated hardware.

We observe an area increase for most benchmarks that closely matches the speed-up. This overhead is partly due to the additional logic used to delay pending writes, detect aliases, and correctly handle mispeculations. The cost of this logic depends on the alias check depth used to achieve II = 1 and on the number of read/write operations in an iteration of the kernel. However, reducing the II also negatively impacts area utilization, as it minimizes the possibility of resource sharing. No resource sharing is possible when II = 1.

CHAPTER 7

Putting it All Together: Synthesizing RISC-V Processors

In the beginning there was nothing, which exploded.
— *Terry Pratchett* (The Science of Discworld)

Instruction set processors are complex pieces of hardware that require significant efforts to design, test, and produce. Despite the many productivity advantages that novel hardware design methodologies such as hardware construction languages (Section 3.2.2) and High-Level Synthesis (Section 3.4) bring to the table, many ISPs are still designed using standard hardware description languages. This phenomenon can be partly explained by the limitations of these tools w.r.t. intrinsic requirements of processor design, such as speculative execution. Additionally, modern processors are typically built around knowledge of past designs, often incrementally improving parts of a well-studied hardware implementation. These existing designs are often based on standard HDL implementations developed when alternative design methodologies were not viable. The effort required to create a new processor microarchitecture from scratch remains significant, especially for production-ready implementations and even for industrial-grade prototyping chips. In this chapter, we describe how the speculative High-Level Synthesis techniques we exposed in the previous chapters can be applied to synthesizing in-order pipelined RISC-V processor cores from a high-level description in the form of an instruction set simulator. After describing state-of-the-art RISC-V cores and HLS-based processor design techniques (Section 7.1), we show that instruction set simulators can be used as expressive processor models (Section 7.2) and that we can synthesize a wide variety of processors that can compete with manually-optimized designs (Section 7.3). Section 7.4 concludes this chapter by discussing the issue of scaling up design space exploration to a large number of potential speculation opportunities, which is a common scenario when trying to design instruction set processors from more complex simulators.

7.1 State-of-the-art

The rise of RISC-V as the premier open-source Instruction Set Architecture has significantly reshaped the processor design landscape and its associated market. Instead of being bound by expensive licenses for proprietary ISAs such as x86 and ARM, vendors can now develop RISC-V cores without the need for a separate ecosystem of tools. Most major compilers support RISC-V targets, often with a wide variety of ratified or planned extensions. The extensible nature of this ISA encourages the development of deeply integrated accelerators that can be directly exploited by programs targeting RISC-V. This

```
struct FtoDC ftodc;
struct DCtoEx dctoex;
struct ExtoMem extomem;
struct MemtoWB memtowb;
// ...

while (true) {
  ftodc_tmp = fetch();
  dctoex_tmp = decode(ftodc);
  extomem_tmp = execute(dctoex);
  memtowb_tmp = memory(extomem);
  writeback(memtowb);

  // Handle stalls
  bool stall[5] = stallLogic();
  if (!stall[0])
    ftodc = ftodc_tmp;
  if (!stall[1])
    dctoex = dctoex_tmp;
  // ...

  // Handle forwarding
  bool forward = forwardLogic();
  if (forward)
    dctoex.value1 = extomem.result;
  // ...
}
```

Figure 7.1 – Overview of the explicitly pipelined structure of the Comet RISC-V processor. The pipeline register state is explicitly passed around between stages, and hazard-handling logic is explicitly instantiated in the code. Adapted from [319].

change in perspective reduces the costs associated with developing and maintaining an ISA, allowing manufacturers and hardware designers to focus on micro-architectural issues. As an open standard, RISC-V also plays a crucial role in making the field more inclusive, lowering the barrier to entry for hobbyists and academics. They can now create their own processor designs that seamlessly integrate into the rich ecosystem of RISC-V tools, tests, and specifications.

Prominent examples of successful open-source RISC-V cores include OpenHW Group's Core-V family [1], some of which are based on a series of RISC-V designs that originated in academia as part of the PULP open hardware platform [2] [323]. However, these designs still rely on standard HDLs for their specification. Bluespec SystemVerilog [44, 39] was used in the design of riscy-OOO [415], an OoO superscalar processor capable booting Linux on FPGA. Hardware construction languages have also been used to design RISC-V cores, most notably VexRiscv [3], a 32-bit processor designed specifically for FPGA using SpinalHDL, and the BOOM [55, 419] family of cores, which provides several implementations of Out-of-Order RISC-V processors, designed using Chisel. The Rocket family of cores [19] was also designed using Chisel.

High-Level Synthesis has received comparatively little attention in the context of processor design. Early work by Skalicky *et al.* [341] suggests HLS as a viable alternative to HDLs for the design of

1. https://github.com/openhwgroup/core-v-cores
2. https://pulp-platform.org/
3. https://github.com/SpinalHDL/VexRiscv

processors with custom ISAs. The authors successfully synthesize several cores from a high-level functional description in C++ (an *Instruction Set Simulator* (ISS), see Section 7.2) for highly-specialized applications on FPGA using a commercial HLS toolchain, and demonstrate significant gains in surface area w.r.t. to general-purpose soft-cores, with a comparatively acceptable performance drop. These processor cores are pipelined solely using the static-analysis-based pipelining capabilities of the HLS toolchain, which leads to high initiation intervals and poor pipeline resource utilization. A similar approach was described by Ahmed *et al.* [4], who report comparable surface area gains. More recent works leverage knowledge of the inner workings of HLS tools and lower the processor description's abstraction level to achieve better performance for processors designed using HLS. HL5 [4] [245] is a 32-bit pipelined processor that implements the `RV32IM` instruction set. It was entirely designed using HLS and an extension to the C++ language called SystemC [281, 38]. While HL5 is presented by its authors as "the first 32-bit processor fully designed and implemented with HLS," they seem to have missed the announcement of Comet [5] [319], which was described in a paper published a year prior. The latter is a 5-stage pipelined RISC-V processor core that supports three ISA configurations: `RV32I`, `RV32IM`, and `RV32IMF`. Similarly to HL5, Comet's pipeline structure is directly expressed in C++, with explicit handling of stalls, forwarding, and multi-cycle operators. Figure 7.1 gives an overview of the structure of Comet's code. Pipeline registers are defined as structures, with signals contained in *e.g.*, `ftodc` corresponding to values stored between the fetch and decode stages of the pipeline. This explicitness allows Comet to bypass potential restrictions imposed by the automatic pipelining capabilities of HLS toolchains. It also enables the synthesis of processor cores that can compete with hand-written HDL designs in terms of performance and surface area. Explicit pipeline descriptions expressed in a language such as C++ have the added benefit of providing a cycle-accurate processor simulator, which can be used for performance validation and micro-architectural analysis. However, by directly describing the pipeline organization in C++, these approaches constrain the schedule of operations to certain stages, thereby severely limiting the usefulness of HLS's optimization capabilities. This RTL view of HLS often requires knowledge of the HLS compiler's internals, coupled with careful code annotation to improve the quality of the generated hardware [130]. Recent work [161] proposes to improve the results of processor synthesis in HLS by introducing static multi-threading, interleaving the execution of multiple threads to increase the reuse distance between operators.

While the approaches discussed above demonstrate that modern HLS toolchains are mature enough to synthesize complex instruction set processor cores, the design of ISPs using HLS is still hindered by the static analysis capabilities of these toolchains. Commercial HLS compilers cannot infer the speculative schedules ISPs require to execute instructions efficiently and best utilize the hardware resources in their pipeline stages. The work we present in Chapter 5 addresses this issue and brings speculation capabilities to HLS using source-to-source transformations. The memory speculation framework described in Chapter 6 further improves speculation support in HLS by providing a general framework for memory access handling in a speculative context. With these two improvements, we can supplement standard HLS toolchains to produce high-quality processor cores from a high-level description similar to the one described in earlier work on High-Level Synthesis of Instruction Set Processors [341, 4], without incurring the performance penalty that results from static scheduling, and without requiring the introduction of multiple hardware threads [161].

7.2 Instruction Set Simulators as Processor Models

The most natural way of expressing the behavior of a processor in code is through a high-level ISA simulator. Such an *Instruction Set Simulator* (ISS) is typically comprised of a loop, where instructions are read from memory, decoded into several fields, and execution is dispatched through a `switch` statement to modify the simulator's state. The program counter (PC) is then updated to point to the next instruction.

4. https://github.com/sld-columbia/hl5
5. https://gitlab.inria.fr/srokicki/Comet

```
1 rv32proc processor = rv32proc_init(/* ... */);
2 rv32instr instr;
3 do {
4   uint32_t raw_instr = rv32proc_fetch(&processor);
5   instr = rv32proc_decode(raw_instr);
6 } while(rv32proc_execute(&processor, &instr));
```

Figure 7.2 – High-level structure of a RISC-V instruction set simulator. An excerpt of the execution logic inside `rv32proc_execute` is provided in Figure 7.3

Figure 7.2 shows the typical structure of an ISS written in a C-like language. The `processor` contains the simulator state, *i.e.* the contents of instruction and data memory, the current value of PC, the register file, and any other state that the processor may require during execution. The simulation loop reads an instruction from memory using the `rv32proc_fetch` function and decodes it in `rv32proc_decode`. The `rv32instr` structure contains all the fields an instruction may require during execution. These fields are given by the RISC-V specification and are created from bit ranges in `raw_instr`. Examples of such fields include `opcode`, which specifies the type of the instruction, `funct3`, which encodes a sub-function for a given instruction type (*e.g.* `addi`, for *add immediate*, is a sub-function of the `opi` instruction type), `simm_I` and `simm_J`, signed integer immediates, or `rd` and `rs1`, which describe the destination register and register operand index into the register file, respectively.

The heart of the ISS shown in Figure 7.2 is located inside of the `rv32proc_execute` function, of which an excerpt is presented in Figure 7.3. This function takes the simulator state and the current instruction as parameters and performs the latter's actual execution. It returns a boolean to indicate whether the simulation should continue with the following instruction or be aborted. The instruction's opcode is matched against known opcodes inside the top-level `switch` statement, and the simulator state is updated accordingly in each `case`. Instructions with a straightforward encoding can directly be executed, such as for `RISCV_AUIPC`. In contrast, others may require matching additional fields to determine their exact semantics, as seen for the `RISCV_OPI` and `RISCV_LD` family of instructions. The `next_pc` variable determines the value of the program counter at the end of the execution of the current instruction. It defaults to selecting the next instruction (on line 3), but it can be updated by control flow instructions such as `RISCV_JAL`. The PC value is updated at the end of the execute function, just before returning control to the simulator's main loop.

Instruction Set Simulators are widely used in the industry and academia for simulation, validation, and prototyping. Notable examples of such simulators for RISC-V processors include Spike [6], the official RISC-V ISA simulator, and Whisper [7], which was initially developed for the verification of the VeeR RISC-V micro-controller cores, maintained by the CHIPS Alliance [8]. Both follow a structure similar to the one depicted in Figure 7.2 and Figure 7.3. The use of ISA simulation is not limited to the RISC-V ISA. Several successful simulators have been developed for a wide variety of instruction sets, including gem5 [37, 237] (which supports ARM, MIPS, Power, RISC-V, SPARC, and x86 architectures), and Sniper [54], which supports fine-grained simulation of multi-core processors.

Contrary to DSL-based approaches, which may lack the required expressivity, the availability of a general-purpose language such as C or C++ allows for modeling complex behavior directly inside the simulator's source code. This expressive power allows for rapid prototyping of ISA and micro-architectural extensions and insertion of additional simulated components such as branch predictors, shadow stacks for Control-Flow Integrity (CFI), and caches. Section 7.3.6 showcases an example of a branch predictor implemented directly at the ISS level. An ISS is, therefore, a highly flexible processor model that can act as a specification of a processor's execution behavior. Consequently, synthesizing

6. https://github.com/riscv-software-src/riscv-isa-sim
7. https://github.com/chipsalliance/VeeR-ISS
8. https://github.com/chipsalliance

```c
bool rv32proc_execute(rv32proc *processor, const rv32instr *instr) {
  bool should_continue = true;
  uint32_t next_pc = processor->pc + 4;
  switch(instr->opcode) {
  case RISCV_AUIPC:
    processor->x[instr->rd] = processor->pc + instr->imm_U;
    break;
  case RISCV_JAL:
    processor[instr->rd] = next_pc;
    next_pc = processor->pc + instr->simm_J;
    break;
  case RISCV_OPI: {
    uint32_t rs1 = processor->x[instr->rs1];
    int32_t simm_I = instr->simm_I;
    switch(instr->funct3) {
    case RISCV_ADDI:
      processor->x[instr->rd] = rs1 + simm_I;
      break;
    /* Other cases ... */
    default:
      illegal_instruction(instr, processor->pc);
      break;
    }
    break;
  }
  case RISCV_LD: {
    uint32_t value = 0;
    uint32_t offset = processor->x[instr->rs1] + instr->simm_I;
    switch(instr->funct3) {
    case RISCV_LB:
      value = (int8_t)mem_read8(processor->data_memory, offset);
      break;
    /* Other cases ... */
    default:
      illegal_instruction(instr, processor->pc);
      break;
    }
    processor->x[instr->rd] = value;
    break;
  }
  /* Other cases ... */
  default:
    illegal_instruction(instr, processor->pc);
    break;
  }
  processor->pc = next_pc;
  return should_continue;
}
```

Figure 7.3 – Execution logic of a RISC-V instruction set simulator. The decoded instruction's opcode is matched against a set of known opcodes, which determine the operation to be executed. The processor state is updated in each `case` statement, before proceeding to the next loop iteration shown in Figure 7.2.

an ISP from its ISS would increase hardware designer productivity while maintaining a single source of truth for the ISA specification, the corresponding ISA simulator, and the hardware design flow input.

Simulation vs. emulation. Instruction set simulators are built on top of a micro-architectural model of the processor they are simulating. This design allows the simulator to provide detailed characteristics of the micro-architecture during execution and enables fine-grained analysis of the processor's behavior. However, this detailed view of the system comes at a cost, which may bring no benefit to some applications. In cases where only the *observable behavior* of the processor needs to be replicated, *emulation* is often a better choice than simulation. Insights into techniques for high-performance system emulation can be found in works describing QEMU [33, 98], a popular system emulator built on top of a dynamic binary translation (DBT) [104, 95, 320, 111] framework.

7.3 Synthesizing Processors using SpecHLS

In single-issue, in-order pipelined processors, a new instruction starts executing before its predecessor instructions are finished executing. A pipeline hazard occurs when two consecutive instructions are dependent or need the same hardware resource simultaneously (see Section 2.3). In such a case, instructions are canceled or stalled until execution can resume. While this type of micro-architecture is well understood, its implementation and customization using RTL specifications are tedious: modifying the pipeline structure often requires a complete rewrite, making Design Space Exploration (DSE) very difficult.

Equipped with the multiple speculation support described in Chapter 5, and the memory speculation framework from Chapter 6, we have all the tools at our disposal to synthesize processors from instruction set simulators. This section describes the initial results obtained using a modified RISC-V ISS, which constrains the code generation of our source-to-source compiler to circumvent some scalability issues in the SpecHLS toolchain. These issues have been addressed by later work, which is discussed in Section 7.4. In the following, we show how the speculative capabilities introduced in previous chapters can serve as a foundation to perform fully automatic micro-architectural synthesis from a behavioral description of a processor in the form of an ISS. We extend SpecHLS to support the synthesis of in-order pipelined processor micro-architectures and their hazard recovery logic. We then evaluate our approach in terms of supported features (both from an ISA and micro-architectural perspective) and quality of results (performance and area). Our results show that our flow can handle complex mechanisms like branch prediction and hardware CFI while providing Quality of Results (QoR) similar to manual designs.

7.3.1 Defining the Pipeline Model

In the following, we consider a simple ISS implemented in C that follows the general structure described in Section 7.2. The processor execution is modeled using an infinite loop that performs instruction fetching, decoding, and execution within a single loop iteration. A direct correspondence exists between an iteration of the simulator loop and a complete instruction execution. Additionally, we assume that memory and the register file are represented using C arrays. The decoding logic is implemented using a `switch` statement. For the sake of simplicity, we assume that all instruction execution paths go either through a branch instruction, a load, a store, an ALU operation, or a multiplication.

Figure 7.4 shows the Gated-SSA representation of this ISS. It is comprised of several static loop-carried dependencies (over `pc`, the register file `x`, and the external memory `mem`), which would prevent a standard HLS tool from fully pipelining the kernel. The representation also exposes several speculation opportunities through γ-nodes, $\gamma_{\texttt{pc}}$ and $\gamma_{\texttt{out}}$, which we aim to explore.

7.3.2 Exposing Hardware Reuse

Because a software ISS describes every instruction's execution path individually, its GSSA representation may expose many redundant operations and memory accesses. For example, the model inferred

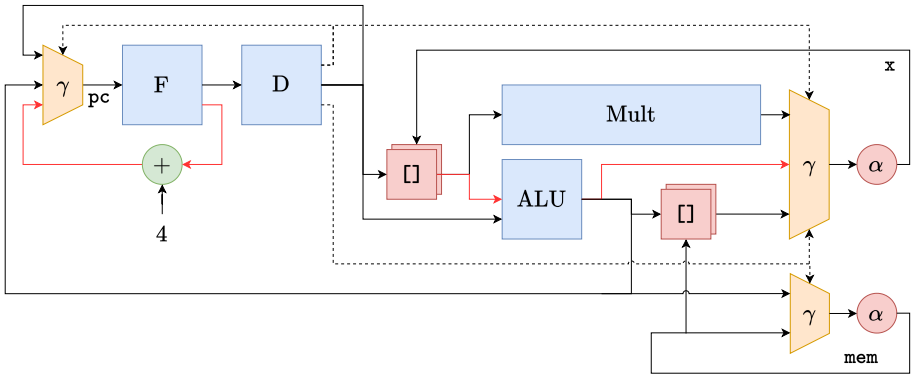


Figure 7.4 – GSSA representation of the full ISS model, including data memory. Plain arrows represent data edges, while dotted arrows represent control edges. The highlighted edges show the speculated paths.

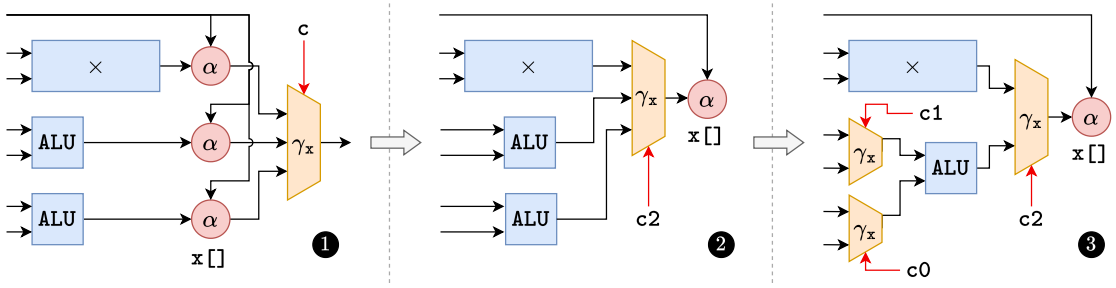


Figure 7.5 – Exposing hardware reuse through γ-node input factorization. The typical structure derived from a naive translation of an ISS to GSSA is shown in ❶. Redundant register file writes can be factored into a single α-node in ❷, and the ALU shared for two of the γ-node's inputs in ❸.

from the code in Figure 7.3 would require several adders and a register file with multiple write ports. Since only one instruction is executed at every iteration of the loop, most of these resources are redundant and can be eliminated. However, some mutually exclusive operations or resources instances do not share the same operands, and therefore, they do not benefit from standard Common Sub-expression Elimination (CSE) compiler passes [5, 81]. In our context, this limitation leads to significant area overheads in the processor's datapath, which must be mitigated.

We address this issue through a preprocessing step, identifying redundant operations and removing them to reduce area cost. Our approach uses a simple heuristic to identify resource-sharing opportunities among a given γ-node's inputs and restructures the program representation to enable resource factorization. Our transformation is illustrated in Figure 7.5, which shows how several array update operations can be factored together to save write-port resources ❷ and how further factorization enables sharing of the ALU ❸. Despite its simplicity, this optimization effectively eliminates redundant operations in many locations inside a typical ISS structure. More efficient techniques based on datapath merging [262, 261] could be employed to expose further optimization opportunities and will be considered in future extensions to the work presented in this thesis.

7.3.3 Data Hazards and Memory Speculation

Our approach can only deal with control-flow hazards by relying solely on the techniques presented so far. It cannot pipeline the execution stages due to the RAW dependencies on the register file. From a compiler's perspective, these dependencies are akin to memory dependencies over the array that *models* the register file. This memory dependency can be speculated on by using the techniques described

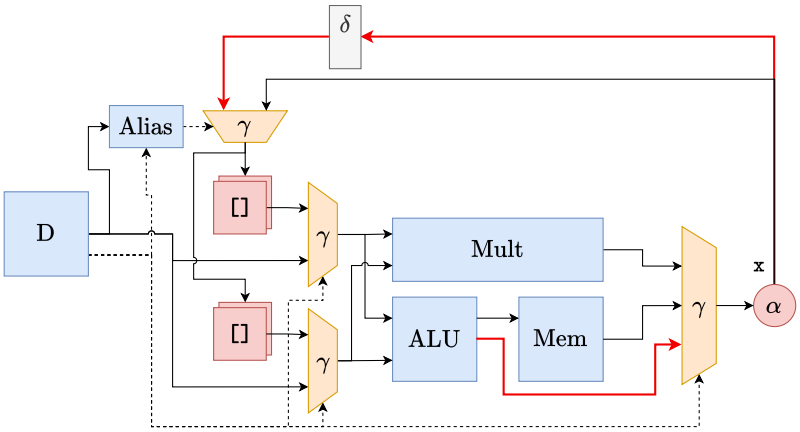


Figure 7.6 – Pipeline interlocking with runtime alias check insertion. Memory access logic is omitted for the sake of clarity.

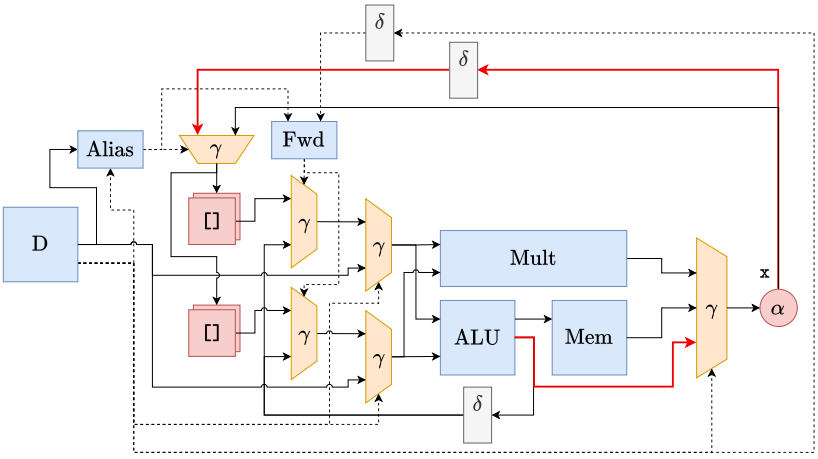


Figure 7.7 – Pipeline forwarding. Memory access logic is omitted for the sake of clarity.

in Chapter 6, *i.e.*, by treating arrays as values and recasting memory speculation as a control-flow speculation. Our toolchain leverages two techniques to handle data hazards in a processor pipeline:

1. *Interlocking.* The first approach implements pipeline interlocking by introducing runtime alias detection checks in the GSSA representation. These checks control an n-ary γ-node that selects, among p previous array values, the one containing the most recent update of the to-be-read location. This principle is illustrated in Figure 7.6 for $p = 1$. The modified GSSA representation exposes a speculation path with a reuse distance of 2, thanks to the additional γ-node. This transformation is described in more detail in Chapter 6, where it is generalized to multiple array reads and array updates.
2. *Forwarding.* The second approach implements a form of operand forwarding to bypass the write operation on the register file. An additional decision layer is added to the hazard management logic, as illustrated in Figure 7.7. The inserted γ-nodes are controlled by the `Fwd` node and select between the previous execution's ALU result and the register file contents. The forwarded value is selected if there is a RAW dependency (*i.e.*, an alias) at the current iteration and if the last iteration used the ALU. Otherwise, further computations are stalled, and the register file is read to get the correct operand value.

Note that forwarding and interlocking are complementary approaches for solving data hazards. The

former avoids unnecessarily stalling execution if the value required by an instruction is already available at a forwarding point in the pipeline. Still, it cannot resolve data hazards involving values not computed yet. For example, in a classical 5-stage pipeline where the memory access stage follows the execution stage, the result of a load cannot be forwarded to the execution stage for the next instruction in the pipeline. Interlocking, on the other hand, solves this issue by stalling execution until dependencies are resolved, but it is more expensive than forwarding in terms of performance.

7.3.4 Supporting Non-Pipelined Execution Units

The SpecHLS toolchains pipelines all operations inside a loop body, regardless of their usage pattern. This aggressive form of pipelining leads to sub-optimal hardware resource utilization for operators seldom used during execution. Consider, for example, the M extension of the RISC-V instruction set, which adds support for integer multiplication and division. Implementing these instructions through the SpecHLS flow will lead to a processor datapath with a fully-pipelined integer division operation and, consequently, a deep pipeline. Given such instructions' low utilization rate, this can be considered a poor design decision.

We replace costly and rarely-used operators with a non-pipelined multi-cycle version to address this issue. A small state machine drives the resulting operation by advancing the computation at each cycle and producing the expected result after several cycles.

7.3.5 Design Space Exploration

Combining memory speculation strategies for the register file `x` with speculation over `pc` leads to a large design space. The most effective speculation configuration likely intertwines speculations involving several γ-nodes. While the extensions to speculative loop pipelining exposed in Chapter 5 allow us to handle such complex speculation scenarios, searching the resulting design space is a non-trivial task that can quickly be computationally demanding.

In this initial attempt at synthesizing RISC-V cores from an ISS, we reduce the number of γ-nodes that may be amenable to speculation through manual annotations in the input code. We then perform an exhaustive search in the design space to determine all speculation paths leading to a pipeline with $\mathrm{II} = 1$ (*i.e.*, a pipeline that can reach a $\mathrm{CPI} = 1$ when no hazard occurs). We discard the least relevant configurations using profiling information. Changing the latencies of the different operators allows us to explore various pipeline structures. Section 7.4 describes an improved DSE framework that can scale to several dozens of intertwined speculation opportunities.

7.3.6 Expressivity

As Section 7.2 mentions, operating from an ISS model expressed in C or C++ offers more flexibility and expressivity than DSL-and code-generation-based approaches. In the following, we discuss several examples that illustrate this versatility. Our chosen examples address this aspect from two angles: the ability to easily extend the ISA with custom instructions and customize the micro-architecture directly from the C specification.

ISA Extensions

We evaluate our approach on the base RV32I instruction set and implement the M extension, which supports integer multiplication and division operations. We did not include support for CSR registers since most of the cores we compare against do not support them either. We, however, implement the ISA extensions for AES encryption described by the authors of the ASSIST [230] framework. Since our approach operates directly from C, we can add the AES instruction by copying the C code describing the instruction's operational semantics provided in the ASSIST paper and modifying only a handful of lines of code. Area and performance results for this extension are provided in Section 7.3.7. They are on par with those reported for ASSIST.

```
k = hash(pc);
pred_pc = pc + 4;
btb_hit = btb_tag[k] == pc;
// ...
if (btb_hit && btb_state[k] >= WEAK_TAKEN)
  pred_pc = btb_target[k];

switch (instr->opcode) { /* ... */ }

if (pred_pc == next_pc) {
  // Good prediction, fast path
  incr(btb_state[k]);
  pc = pred_pc;
} else {
  // Bad prediction, slow path
  if (btb_hit)
    decr(btb_state[k]);
  else if (/* The instruction was a branch */)
  ↪  {
    btb_state[k] = WEAK_TAKEN;
    btb_tag[k] = pc;
    btb_target[k] = /* Branch target */;
  }
  pc = next_pc;
}
```

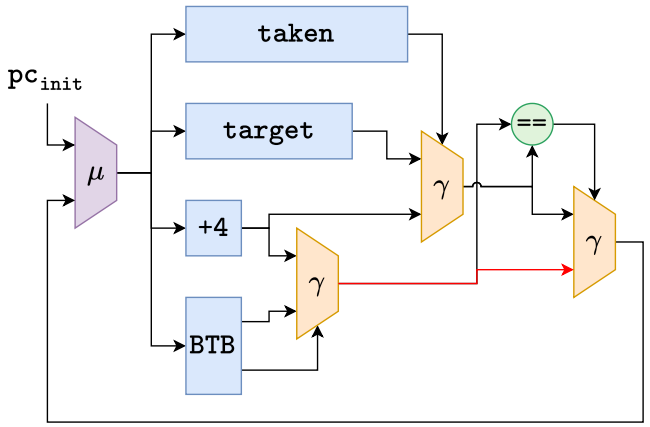


Figure 7.8 – Branch predictor implemented at the ISS level. The `pred_pc == next_pc` check in the code on the left produces a γ-node that exposes a speculation opportunity on the prediction result.

Micro-Architectural Extensions

Our approach makes it possible to describe micro-architectural features directly at the behavioral level. Our toolchain can then infer the corresponding hardware feature. In such a case, the C specification exposes both the target core's architectural and micro-architectural state. We implement two micro-architectural extensions to demonstrate the expressivity of our approach:

1. *Branch prediction.* We extend our ISS model with a branch prediction mechanism based on a bimodal branch predictor [249], built atop a 256-entry Branch Target Buffer (BTB). This predictor is expressed directly at the ISS level as a sequence of conditional statements, as shown in Figure 7.8. This model leads to a GSSA representation with an additional short speculation path. This path is frequently taken since the prediction is expected to be correct most of the time. Our toolchain can then harness this new speculation path and subsequently reduce the number of control-flow hazards.
2. *Control-Flow Integrity.* We add a CFI security mechanism [1, 47] to protect the processor against buffer overflow attacks. The mechanism consists of a shadow stack implemented in a separate on-chip memory. This shadow stack works as follows: whenever a call instruction is executed, the processor saves the return address in `ra` and pushes it to the shadow stack. When branching back to `ra`, the processor checks that the target jump address matches that at the top of the shadow stack. In case of a mismatch, an exception is raised. As for branch prediction, this mechanism is entirely described at the ISS level. From a micro-architectural perspective, the CFI adds a long path in the PC update loop, although that path should rarely be taken. This path leads to additional pipeline stages to preserve clock speed. Here again, our flow can infer the modified pipelined structure automatically.

7.3.7 Experimental Validation

To demonstrate that our approach can generate competitive pipeline micro-architectures, we generate a large set of processors by exploring different speculation setups [132]: no speculation on the register file, pipeline interlocking, or forwarding. We also modify the latency of the different operational blocks used in the ISS to explore several pipeline depths. This ability to easily explore pipeline configurations is a significant advantage of our approach over the processor description language frameworks

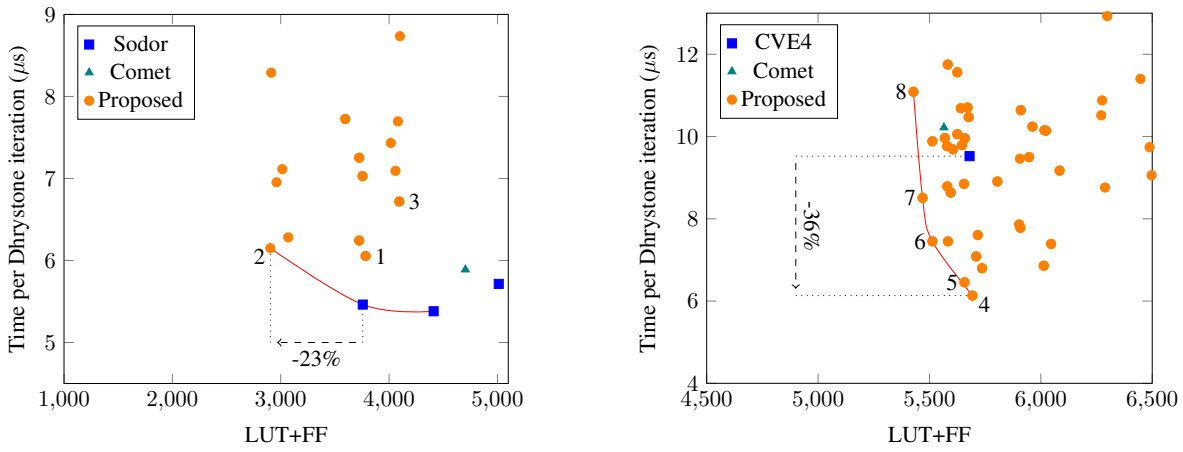


Figure 7.9 – Area and performance results of our DSE for 21 variants of RV32I cores (left) and 105 variants of RV32IM cores (right). Results for the Sodor, Comet, and CVE4 cores are reported as baselines.

described in Section 3.5.2, which often require the user to define the pipeline organization explicitly upfront. As baselines, we synthesize the three Sodor pipelined cores [9] (2-, 3-, and 5-stage pipelines), and the CV32E40P core [93, 126]. We synthesize two configurations of the Comet processor [319], namely RV32I and RV32IM. The latter core is a good baseline for understanding the level of performance that can be expected from a manually-optimized low-level pipeline description in HLS. Additionally, the output code produced by SpecHLS follows the same structure as the Comet implementation, with an explicit pipeline structure expressed directly at the source level (see Figure 7.1). As our generated cores do not implement RISC-V CSR registers, we remove the CSR unit from all cores.

Our experiments target an Artix7 XC7A200TISBG-1L FPGA and use Vitis HLS 2021.2 as the HLS backend. Performance results were obtained by executing the Dhrystone benchmark suite [390], compiled using `newlib`.

Automatic design space exploration results are provided in Figure 7.9. The leftmost part represents the results obtained for the RV32I ISA, and the rightmost part gives the results for the RV32IM ISA. The generated micro-architectures are slower than the Sodor and Comet baselines for the RV32I ISA, while we can generate faster cores for the RV32IM target (36% faster than the CVE4 baseline). Our generic approach generates extra control logic (*e.g.*, rollback mechanisms) on the critical path of the RV32I cores, reducing the maximal achievable frequency. On the other hand, the critical path of the RV32IM cores is located in the multiplication and division unit. The extra logic that limits the RV32I performance could be optimized during the speculative scheduling transformation, but this improvement is left as future work.

Regarding hardware cost, our exploration highlights smaller designs than the baselines for both the RV32I and RV32IM configurations. This discovery of such points is made possible by our toolchain's design space exploration capabilities combined with the HLS backend's ability to infer efficient memory primitives for the register file.

We report the area and performance level of several generated cores in Table 7.1. The vanilla ISS row shows results for the untransformed ISS used as an input to our toolchain to generate different core configurations. These results highlight the need for speculative scheduling to synthesize efficient processor soft cores from an ISS. The cores generated by our flow, numbered from 1 to 8, are also labeled in Figure 7.9. Cores 1, 2, 4, and 5, which are the fastest ones, implement the forwarding mechanism described in Section 7.3.3. Core 3, the core with the highest DMIPS/MHz result, does not implement any memory speculation mechanism. However, the absence of speculation on memory aliases negatively impacts the core's execution frequency. We also observe that Core 2 is much smaller than Core 1 while

9. https://github.com/ucb-bar/riscv-sodor

Table 7.1 – Area and performance results for several generated micro-architectures and baselines.

Instruction Set	Configuration	LUT	FF	DSP	$F_{\max}$ (MHz)	DMIPS/MHz
RV32I	Vanilla ISS	4399	2884	0	183	0.15
	Sodor 5	3554	1458	0	102	0.97
	Sodor 3	2540	1217	0	96	1.08
	Sodor 2	3245	1166	0	93	1.13
	Comet	2951	1753	0	102	0.95
	Core 1	2188	1595	0	91	1.03
	Core 2	1934	970	0	99	0.93
	Core 3	2461	1633	0	79	1.07
RV32IM	Vanilla ISS	10793	7674	0	84	0.04
	Comet	3829	1742	5	53	0.96
	CVE4	4463	1217	9	62	1.04
	Core 4	3279	2414	4	96	0.96
	Core 5	3211	2446	4	92	0.96
	Core 6	3096	2417	4	102	0.75
	Core 7	3112	2357	4	101	0.66
	Core 8	3107	2321	4	76	0.68
RV32IM with ext.	BPred	3952	3186	4	67	1.03
	CFI	3162	2416	4	87	0.96
	AES	3364	2439	4	88	0.96

having a similar performance level. In the former, the HLS backend instantiates a LUTRAM memory for the register file, increasing the pipeline's depth and maximal frequency and significantly reducing the number of required FFs.

Comparing against our different baselines highlights that we generate cores using fewer LUTs but similar amounts of FFs. We also note that the performance improvement against baseline cores for the RV32IM ISA is due to the increased maximal frequency of the cores. Indeed, our toolchain can balance the pipeline organization automatically to match the target timings. Lastly, Table 7.1 also provides performance and area results for the three extensions discussed in Section 7.3.6. These experiments demonstrate that our approach can handle micro-architectural customizations directly at the ISS level. We can easily add instructions or features that modify the pipeline structure directly in the input C code.

7.4 Scaling Up to a More Idiomatic ISS

One of the main strengths of SpecHLS lies in its ability to uncover complex, fine-grained speculation opportunities that other approaches would miss. This fine-grained view raises the challenge of determining *where* (*i.e.*, on which γ-node) and *how* (*i.e.*, for which input) to speculate. An exhaustive search is impractical in most cases since it is common for control-heavy kernels to reach hundreds of thousands of potential speculation configurations. The results obtained in the previous section circumvent this design space size explosion issue by heavily constraining the set of γ-nodes that should be considered by speculative scheduling. It disables speculation on all but a handful of γ-nodes. An actual unconstrained ISS written for the RV32I ISA contains at least four times as many γ-nodes, some of which have more than two possible inputs. We address this issue by proposing an efficient DSE algorithm based on a few heuristics and static analyses.

A large number of γ-nodes quickly makes the problem of finding the best speculation candidates intractable. If we denote the set of γ-nodes in our representation by Γ, and the number of inputs for

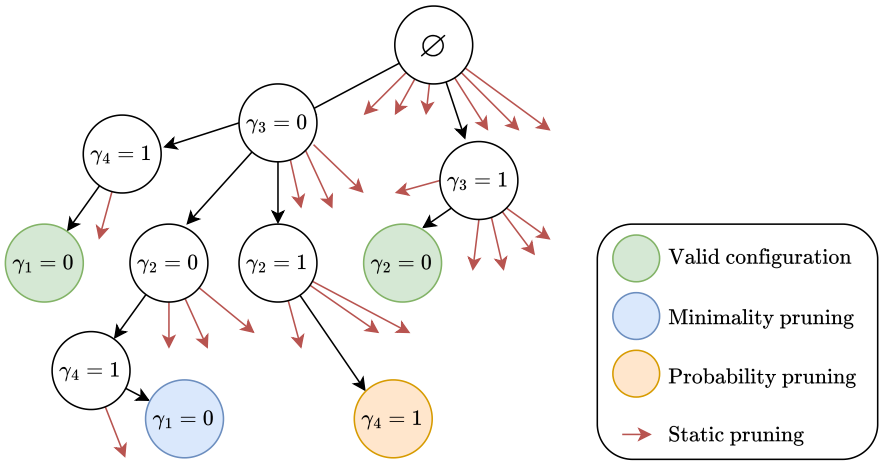


Figure 7.10 – Sample design space exploration tree. Each node represents a γ-node input selected for speculative scheduling. A path starting at the tree's root forms a speculation configuration.

node n by inputs(n), then our solution space contains up to

$$\prod_{n \in \Gamma} \text{inputs}(n)$$

distinct γ-node combinations to consider. For each γ-node, we need to determine whether it should be selected for speculative execution and, if so, which one of its inputs should be considered as taken. However, we are only interested in the subset of configurations that have the following properties:

— the configuration achieves the target II (usually II = 1) by exposing a short path in the GSSA graph representation;
— the configuration has an occurrence probability higher than a reference threshold θ, which is set by the user as a parameter to guide the design space exploration;
— the configuration is minimal: removing any speculated γ-node from it increases the achievable II.

We call a speculation configuration that satisfies all these requirements a *valid configuration*. In the context of a complete design space exploration, the goal is to obtain all such configurations. However, since the search space is intractable, we may look for the first n solutions instead. In all cases, we need to be able to (i) prune the search space as early as possible and (ii) explore the space by favoring solutions that are likely to be more profitable using heuristics.

We propose an exploration algorithm built on a branch-and-bound approach developed in joint work with Dylan Leothaud [225], the main idea of which is illustrated in Figure 7.10. Our search procedure starts at the root of the exploration tree, with no speculated γ-node. It employs three distinct pruning techniques to eliminate invalid speculation configurations:

— *Minimality pruning* excludes the configurations that are strict supersets of already explored valid solutions.
— *Probability pruning* excludes configurations whose occurrence probability on a representative input dataset is lower than the user-specified value of θ. The probability of each configuration is obtained by executing the input code to our toolchain with additional instrumentation to determine which input a γ-node selects at each iteration of the loop that will be speculatively scheduled.
— *Static pruning* uses scheduling information to statically determine which configurations will never lead to an initiation interval that matches the target II. The details of this pruning method are out of the scope of this thesis.

Each node of the exploration tree corresponds to a speculated input selected for a particular γ-node. The path leading from the tree's root to a node goes through the nodes corresponding to the current

Table 7.2 – Design space metrics for an idiomatic RISC-V ISS. The probability pruning method uses a threshold of $\theta = 10\%$. Excerpt from the results presented in our FPL'24 paper [225].

Benchmark	Design space size				Runtime
	γ-nodes	Baseline	Heuristics	SOTA [350]	
RISC-V CPU	16	752M	1.46k	11.3M	263s
	(other results omitted)				

Table 7.3 – Area and performance results for a sample speculation configuration of a RISC-V ISS. Excerpt from the results presented in our FPL'24 paper [225].

Benchmark	Type	II	T_{clk} (ns)	LUT	FF	DSP
RISC-V CPU	Baseline	5	8.89	1499	361	0
	Speculative	1	14.35	1630	953	0
	(other results omitted)					

speculation configuration being explored. Pruning alone is not sufficient to enable efficient design space exploration. The order in which γ-nodes are considered during the exploration is also important since it should favor γ-nodes that lead to more efficient schedules. A total order on γ-nodes, based on the notion of *mobility*, is exposed in the paper but is out of the scope of this thesis.

Table 7.2 shows design space metrics for an idiomatic RISC-V ISS that employs the canonical structure described in Section 7.2. The 16 γ-nodes lead to more than 750 million possible speculation configurations, which makes an exhaustive search highly impractical. Our exploration algorithm reduces the design space size by several orders of magnitude, leading to a more tractable solution space: we find a suitable speculation configuration in a little more than four minutes. As a comparison, we show the size of the explored space when following the approach of Szafarczyk *et al.* [350], who propose to automatically insert dynamic execution behavior in statically-scheduled circuits in the SOTA column. Our results show that our approach is much more scalable while natively supporting both control flow and memory speculation.

Table 7.3 provides area and performance results for our RISC-V processor, comparing the baseline implementation (the unmodified ISS) with the one found by our exploration process. Similarly to the results described in earlier sections, our experimental evaluation demonstrates a reduced initiation interval at the cost of additional hardware and a lower clock frequency. Part of the area overhead is due to the additional logic used to handle mispeculations, while the remainder is a direct consequence of reaching a lower II, leading to fewer resource-sharing opportunities. Note, however, that the area results are on par with the smallest designs exhibited by the constrained DSE approach described in Figure 7.9 and Table 7.1.

Fast design space exploration is a key selling point of High-Level Synthesis toolchains [79], which enables faster iteration times and better hardware design trade-offs. This capability is especially valuable for processor design, where most existing designs still rely on standard HDLs with little exploration capabilities outside of tedious code-rewrite-and-verification cycles. Yiannacouras *et al.* [406] provide exploration capabilities for processor designs in their SPREE framework. The latter takes a custom ADL description and generates several versions of a processor in Verilog. However, SPREE does not include support for speculation, outside of the next-PC fetch required by in-order pipelined processors. Saussereau *et al.* [331, 330] present an alternative approach to DSE for processors, where several parts of a general-purpose core can be included or excluded from the final design, based on the target application's requirements. Their exploration framework relies on a textual replacement-based preprocessor and requires a full implementation of the core to be available upfront. On the other hand, our work focuses on a more flexible approach to processor design based on a high-level functional description of the core in a language like C or C++. The results in this section show that, by taking advantage of static analysis and profiling information, we can efficiently explore large design spaces with hundreds of

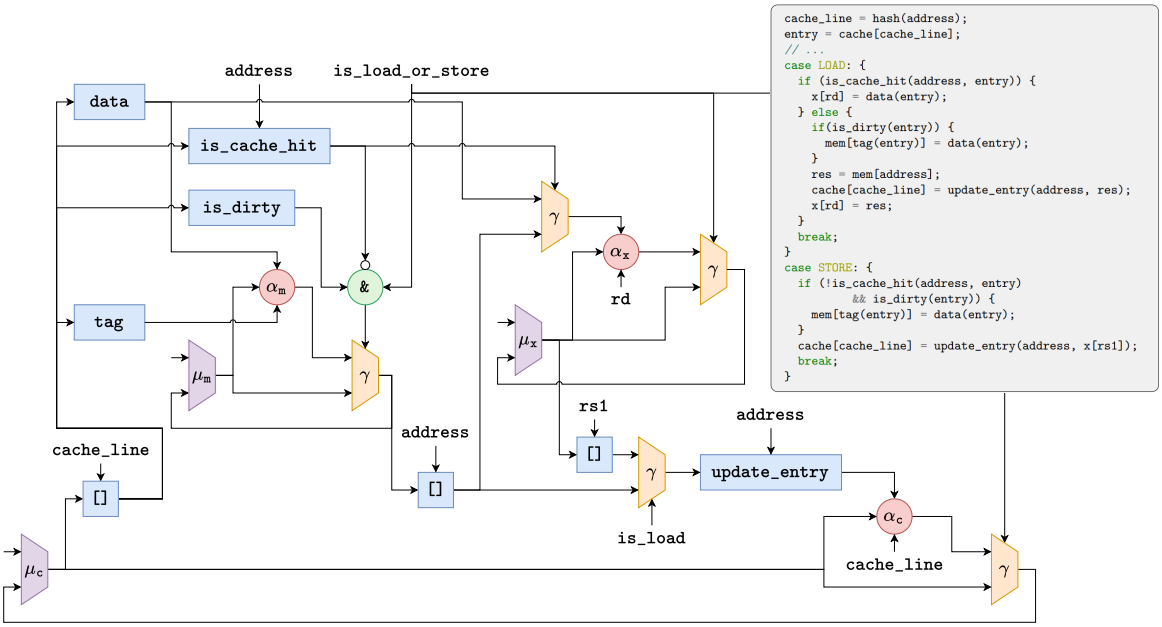


Figure 7.11 – Cache implementation inside of an ISS. A cache simulator is directly embedded into the ISS, exposing a fast speculation path on cache hit.

thousands of solutions in a matter of minutes, without costly RTL simulations. Our approach is scalable enough to automatically infer the set of speculation decisions that enable the automatic generation of an in-order RISC-V processor core from a high-level description in the form of an ISS.

7.5 Toward More Complex Micro-Architectures

As demonstrated in Section 7.3.6, instruction set simulators can be extended to include micro-architectural features such as branch prediction or CFI directly in their source code. While this approach breaks the boundaries between architecture and micro-architecture, it enables a wide range of customization points and micro-architectural exploration opportunities. With the scalable design space exploration techniques presented in the previous section, we can explore many speculation configurations, opening the door to implementing more complex micro-architectural features directly at the ISS level. Figure 7.11 illustrates such a feature by implementing a data cache in an ISS. This example exposes several new γ-nodes, which are as many potential speculation opportunities in the processor design. Additionally, the array reads are more opportunities to apply the memory speculation techniques discussed in Chapter 6, exposing even more γ-nodes to the speculation exploration algorithm.

Other micro-architectural design choices can also be expressed at the source level. For example, a superscalar processor capable of issuing n instructions concurrently may be expressed as an *unrolled* version of a single-issue processor's ISS. A first attempt at synthesizing such a core on a reduced version of the RISC-V ISA is presented in our latest paper [225]. By encoding the transition from single-issue to multiple-issue micro-architectures as a source-to-source transformation, we can also provide additional design space exploration opportunities to hardware designers, letting them focus on the high-level specification of the processor's architecture before applying the unrolling transformation to evaluate different issue widths. Such automation is part of the research perspectives that emerge from our work, which we give an overview of in the next chapter.

CHAPTER 8

Conclusion

In the darkest corners of the earth, where light dares not tread, ancient truths lie waiting to be unearthed.
— *H.P. Lovecraft* (The Dunwich Horror)

This thesis lays the foundations of a High-Level Synthesis flow capable of synthesizing fully-featured in-order pipelined RISC-V processor cores from an instruction set simulator. We extend and improve the initial ideas of Speculative Loop Pipelining (SLP) described by Derrien *et al.* [97] in late 2020 to provide both a framework for handling multiple interacting speculations, as well as a set of techniques to handle memory accesses in such a speculative context efficiently. We describe a design space exploration methodology based on a combination of static analysis, profiling, and heuristics that enables efficient exploration of possible speculative hardware configurations in large design spaces. The resulting toolchain is general enough to support complex ISA and micro-architectural features while producing designs that are performing on par with RTL designs. Our work on instruction set processor design using HLS immediately applies to synthesizing other hardware accelerators. It opens up a new avenue for the design of high-performance speculative accelerators from high-level specifications.

The SpecHLS source-to-source transformation flow described in this thesis is an open-source project, which is available on Inria's Gitlab instance under the GeCoS umbrella project [1]. It is comprised of a front-end that converts C code to the Gated-SSA representation used by our transformation passes, a source transformation middle-end that performs code optimizations, design space exploration, and discovers speculation opportunities in the input code, and a backend that generates code suitable for synthesis with the AMD Vitis HLS toolchain. The repository also contains the full implementation of the RISC-V instruction set simulators used to obtain the results described in Chapter 7.

The automatic synthesis of in-order pipelined processor cores is a significant milestone for the SpecHLS toolchain, proving our approach's viability for complex hardware design endeavors. However, several challenges still need to be addressed to enable the synthesis of more performance-demanding micro-architectures that could underpin application-class processors. Section 8.1 discusses potential future research directions for our work, and Section 8.2 outlines some of the technical challenges and opportunities that come with the design of a speculative hardware compiler like SpecHLS. Section 8.3 lists the publications that resulted from our efforts. We hope that our work can foster new interests in HLS for processor design and serve as a basis for an easier-to-use, robust, and flexible processor design flow that would make designing a processor as simple as writing an instruction set simulator.

1. https://gitlab.inria.fr/gecos/gecos-hls

8.1 Research Perspectives

Our work sits at the crossroads of three research domains: hardware, micro-architectural, and compiler design. This exposure to multiple communities is an opportunity to create new research that bridges the existing abstraction gaps between them. However, work that spans several domains that can sometimes vary widely with respect to their expectations brings along its own set of challenges. The compiler-oriented approach we took during our work may not always look like the best fit for hardware designers who are used to the low-level nature of RTL design, giving maximum control to the designer at the expense of increased iteration times and a more error-prone design process. Conversely, we still want to enable complex hardware behavior to be modeled accurately in our representation. The latter require a level of control that cannot be achieved through too high-level an abstraction, albeit raising abstraction levels is tempting for compiler engineers.

Striking a good balance between the generality of our transformation flow and the usability of the toolchain to design actual hardware proved challenging. Nevertheless, we believe that the ever-growing need for custom processors and high-performance hardware accelerators makes SpecHLS a valuable tool for bridging parts of the gap that still exists between the hardware and compiler design communities. The following sections expose several prospective research perspectives that stem from our work and largely involve a blend of both compiler and hardware design techniques. While some of these ideas have not been explored yet, some are already the subject of active research. Others are part of the LOTR ANR project [2] (ANR-23-CE25-0016), which aims at designing and implementing *a new generation of hardware synthesis flow targeted at the design of pipelined processor micro-architectures*, with a focus on the RISC-V ISA.

8.1.1 Reducing the Cost of Speculative Hardware

The results we present in Chapter 5 and Chapter 6 emphasize that the speculative execution capabilities introduced by SpecHLS into HLS designs come with a non-negligible surface area overhead. The latter is primarily due to the rollback control logic. A similar overhead issue was also outlined by related works on dynamically and speculatively scheduled HLS based on dataflow circuits [184, 186]. However, the handshaking management overhead inherent to elastic hardware designs does not affect our toolchain.

The rollback insertion logic in the SpecHLS toolchain currently relies on a systematic and naive algorithm for rollback placement. These rollback components, built using multiplexers and shift registers, must keep track of the loop iteration state to enable mispeculation recovery. Their internal buffers form a significant part of the area overhead introduced by speculative scheduling in hardware designs. Additionally, rollbacks harm the circuit's maximum achievable clock speeds since they often end up on the hardware design's critical path. Therefore, simplifying the rollback logic is paramount to producing circuits that are competitive with RTL designs on a surface area level.

We note that some of the rollbacks introduced by SpecHLS may not be necessary in the current implementation's state. In the RISC-V processor examples discussed in this thesis, the current naive approach would insert a rollback in the final design to revert any changes to the value of the program counter. However, there is no need to roll back PC, since it always gets a value from the increment to the next instruction or a branch. It never goes back to a previous value during the execution. Variables with such strong forward progress properties do not require any rollback logic, and the result of their rollback logic is only ever used by a speculative γ-node. We can eliminate the corresponding rollback by leveraging *postdominance* analysis in the GSSA representation.

Another rollback simplification opportunity arises in the presence of *reversible computations*. We say that a hardware operator Ω is reversible if there exists a finite set of hardware operators $W = \{\omega_1, \ldots, \omega_m\}$ such that

$$\exists I = (i_1, \ldots, i_n) \in \mathcal{S}_n \quad \omega_{i_1}\left(\ldots\left(\omega_{i_n}(\Omega(x))\right)\ldots\right) = x,$$

2. https://lotr.irisa.fr/

with $\mathcal{S}_n$ denoting the set of all permutations of elements in $[\![1; n]\!]$. If we suppose that operations on a speculated variable are reversible, we can avoid using a rollback and replace it with operators from W to compute the rolled-back value. A simple example of this optimization arises for loop indices incremented by a constant value s at each iteration. Instead of keeping a history of successive indices, we can insert a subtraction operator and decrement the loop index by $n \times s$ to roll back the last n iterations. We can extend the idea of reversible computations to rolling back loop indices on *polyhedral* iteration domains [136, 377], where the reverse computation of the loop index update may be easily computed.

8.1.2 Resource Sharing and Speculation

As outlined in Chapter 7, a typical C or C++ implementation of an instruction set simulator may expose many redundant computations. This redundancy is usually not an issue in a traditional compiler that can leverage compile-time optimizations to factor out or eliminate redundant computations between different branches of the ISS' main `switch` statement. Besides, generating two instructions instead of one in the compiled executable may not significantly impact the simulator's performance. The converse is true when dealing with compilers that generate hardware: requiring two potentially expensive operators in the output (*e.g.*, two multipliers) where one could have been enough can incur a large area and performance overhead in the final design. Consequently, modern HLS toolchains heavily rely on *resource sharing* [379, 195] algorithms and heuristics to reduce the size of synthesized designs.

While we currently apply a relatively straightforward resource-sharing approach for the synthesis of processor cores from an ISS, there is significant potential for improvement. By employing more advanced resource-sharing techniques, such as datapath merging, we could greatly enhance the resource utilization of our synthesized designs. Resource sharing issues also emerge when dealing with memory accesses, where certain hardware implementation resources may not expose sufficient read and write ports to accommodate for the need of the input program. Future work will need to propose a strategy that can handle such resource constraints in general-purpose designs and instruction set processors while still providing the high level of performance we have come to expect from speculative designs.

8.1.3 Formal Verification

Verification, formal or through an extensive test suite with high coverage, is essential to modern hardware design. The complexity of instruction set processors makes them an especially challenging target for verification. Additionally, verification is often performed on formal models that are *derived* from the RTL specification but need to be carefully kept synchronized to stay relevant [227]. By providing a single source of truth in the form of an instruction set simulator, SpecHLS already reduces the maintenance surface for cycle-accurate simulation, performance estimation, and hardware synthesis. Our approach could be extended to include the automatic generation of a formal model of a processor's behavior (expressed using, *e.g.*, *temporal logic* [40]).

Another critical component of the SpecHLS toolchain, the speculative scheduling engine, would benefit from more rigorous verification. The approach proposed by Lee *et al.* [220] for checking the equivalence of hardware models before and after transformations applied by a High-Level Synthesis compiler is a good starting point for such future work. A formal model of our transformation would also allow us to verify specific properties of our speculation control FSM, for example:

— making sure that there are no mispeculation deadlocks, *i.e.*, ensuring that we cannot end up in an infinite loop of mispeculation recovery;

— ensuring that the speculative execution schedule produced by our FSM is always at least as fast as the worst-case static schedule produced by a standard HLS toolchain for a given input code.

While we are reasonably confident that these properties hold, the complex interactions arising from intertwined speculations make such intuitive reasoning hard to trust.

Improving the predictability of the designs generated by SpecHLS also constitutes an interesting research direction, in line with recent work on predictable hardware generation for HLS [269].

8.1.4 Safe and Secure Micro-Architecture Design

Hardware safety mechanisms, designed to protect execution from transient faults or permanent errors, often rely on some form of redundancy [325]. Several micro-architecture designs are based on such duplication techniques that duplicate parts of the pipeline's operations to verify that the execution ended without any errors [22, 142, 228]. The insertion of such safety mechanisms in micro-architectures synthesized with SpecHLS is straightforward since the designer can directly express them in the ISS model. Additionally, the SpecHLS compiler could automatically insert safety checks, enabling more design space exploration for safety-critical processor cores.

On the security side, both software and hardware security could benefit from our proposed automated micro-architecture design. Software security mechanisms, such as Control Flow Integrity, can be readily expressed in the SpecHLS flow, as demonstrated in Chapter 7. Similarly, hardware security mechanisms often consist of the addition of functional blocks in the pipeline or instruction set extensions [393, 252, 92, 392, 138]. The latter can both be expressed in the ISS that is processed by our toolchain.

8.1.5 Synthesizing More Complex Micro-Architectures

In its current state, SpecHLS can successfully generate several micro-architectures for pipelined in-order processors from an ISS. However, this kind of micro-architecture is suitable mainly for embedded and low-power applications, where energy consumption requirements dwarf performance requirements. In contexts where performance becomes more important, micro-architectures that can issue more than one instruction per cycle (*i.e.*, superscalar micro-architectures) start becoming relevant. In its simplest form, a superscalar processor simulator can be built from a single-issue simulator by unrolling its main loop: the more iterations are unrolled, the more instructions can be fetched at once and executed simultaneously. Unconditionally unrolling the ISS' main loop body may, however, introduce new challenges, especially w.r.t. resource usage, memory port contention, and a lengthened critical path, which are issues that will need to be addressed if SpecHLS is to be used for superscalar processor synthesis.

A natural next step to superscalar processors is Out-of-Order processor synthesis. The extensible nature of the instruction set simulator approach that we describe in this thesis would allow us to easily integrate hardware components that are essential to OoO execution directly at the source level (*e.g.*, a scoreboard [154]). Such an approach may be a good first step towards OoO processor synthesis, though it leads to micro-architectural leakage in the architectural abstraction surface of the ISS. A better solution would be to automate the insertion of OoO support hardware during the SpecHLS transformation flow. Such an automation step would require building a more robust speculation FSM building framework. It may also necessitate introducing processor-design-specific transformation passes in the SpecHLS flow, effectively trading off generality for efficiency.

8.2 Technical Perspectives

In addition to the new research opportunities described in the previous section, developing the SpecHLS toolchain comes with several technical challenges. This section discusses some of the technical perspectives for the future development of our hardware design flow.

The current toolchain is based on the GeCoS compiler framework [124], which was developed in Java and later extended using the Xtend dialect of Java [3]. The infrastructure is a large piece of legacy software with several limitations that make the development of new features tedious. The MLIR compiler infrastructure [218] is a promising alternative that provides well-tested and maintained reusable components for the development of compilers. Inspired by LLVM [217] and the lessons learned from its development, MLIR provides an extensible framework for defining intermediate representations, transforming between them, and generating code. It also fostered the development of the CIRCT [4] initiative, which aims at building an open ecosystem of tools for hardware and compiler development.

3. https://eclipse.dev/Xtext/xtend/
4. https://circt.llvm.org/

CIRCT and MLIR represent an opportunity for SpecHLS to offload parts of its core infrastructure onto a more robust basis, bringing performance, ease-of-use, and visibility benefits to the toolchain.

The development of our design space exploration algorithm [225] (see Section 7.4) has already led us to use parts of the CIRCT infrastructure for scheduling operators. The scheduling information we gather during this pass is sufficient to allow us to generate Verilog or VHDL directly instead of relying on a standard HLS backend for operator scheduling. The only critical infrastructure component required to generate synthesis-ready HDL code is a register retiming pass [224]. CIRCT does not currently expose such a retiming algorithm, and developing one for SpecHLS within an MLIR-based framework would allow us to contribute it back to the community.

Since commercial HLS toolchains do not expose a reusable timing and resource allocation model for their target FPGA platforms, we would need to quantify the QoR reduction incurred by switching to an entirely open-source stack. We expect the discrepancy to be directly linked to the quality of our timing model, which we will need to improve in order to avoid *loose micro-architectural loops* [43].

Finally, numerous opportunities exist to increase the quality of the hardware produced by SpecHLS. For example, the control logic inside the speculation FSM can quickly become complex, but there are opportunities for common path fusion and simplification in this area.

8.3 Publications

Our work has led to four publications in international journals and conferences, one publication in a French conference, and several poster communications. The former are listed below, in chronological order of publication.

- Jean-Michel Gorius, Simon Rokicki, and Steven Derrien, **SpecHLS: Speculative Accelerator Design Using High-Level Synthesis**, in *IEEE Micro*, 2022.
- Jean-Michel Gorius, Simon Rokicki, and Steven Derrien, **Design Exploration of RISC-V Soft-Cores through Speculative High-Level Synthesis** (short paper), in *2022 International Conference on Field-Programmable Technology (ICFPT)*, 2022.
- Jean-Michel Gorius, Simon Rokicki, and Steven Derrien, **A Unified Memory Dependency Framework for Speculative High-Level Synthesis**, in *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC 2024)*, 2024.
- Dylan Leothaud, Jean-Michel Gorius, Simon Rokicki, and Steven Derrien, **Efficient Design Space Exploration for Dynamic & Speculative High-Level Synthesis**, in *FPL 2024 - 34th International Conference on Field-Programmable Logic and Applications*, 2024.
- Dylan Leothaud, Jean-Michel Gorius, Simon Rokicki, and Steven Derrien, **Exploration efficace d'espace de conception pour la synthèse de haut niveau dynamique et spéculative**, in *COMPAS 2024 - Conférence francophone d'informatique en Parallélisme, Architecture et Système*, 2024.

Our work has also been presented at the *Colloque National du GDR SOC²* (Rennes, France, 2021), *RISC-V Spring Week* (Paris, France, 2022), and *RISC-V Summit Europe* (Munich, Germany, 2024).

Finally, an article on *Optimizing Recovery Logic in Speculative High-Level Synthesis* has been submitted to DAC 2025 and is currently under review. It builds on some of the ideas developed in Section 8.1.1.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow integrity principles, implementations, and applications". In: *ACM Trans. Inf. Syst. Secur.* 13.1 (Nov. 6, 2009), 4:1–4:40. ISSN: 1094-9224. DOI: 10.1145/1609956.1609960.

[2] *Adding Hardware Accelerators to Reduce Power in Embedded Systems*. White Paper 650571. Altera, Sept. 30, 2009, pp. 1–5.

[3] Sallar Ahmadi-Pour, Vladimir Herdt, and Rolf Drechsler. "The MicroRV32 framework: An accessible and configurable open source RISC-V cross-level platform for education and research". In: *Journal of Systems Architecture* 133 (Dec. 1, 2022), p. 102757. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2022.102757.

[4] Tanvir Ahmed, Noriaki Sakamoto, Jason Anderson, and Yuko Hara-Azumi. "Synthesizable-from-C Embedded Processor Based on MIPS-ISA and OISC". In: *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*. 2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing. Oct. 2015, pp. 114–123. DOI: 10.1109/EUC.2015.23.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. USA: Addison-Wesley Longman Publishing Co., Inc., July 1986. 796 pp. ISBN: 978-0-201-10088-4.

[6] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, Brian Taba, Michael Beakes, Bernard Brezzo, Jente B. Kuang, Rajit Manohar, William P. Risk, Bryan Jackson, and Dharmendra S. Modha. "TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10 (Oct. 2015), pp. 1537–1557. ISSN: 1937-4151. DOI: 10.1109/TCAD.2015.2474396.

[7] Peter Alfke, Ivo Bolsens, Bill Carter, Mike Santarini, and Steve Trimberger. "It's an FPGA!" In: *IEEE Solid-State Circuits Magazine* 3.4 (2011), pp. 15–20. ISSN: 1943-0590. DOI: 10.1109/MSSC.2011.942449.

[8] Mythri Alle, Antoine Morvan, and Steven Derrien. "Runtime dependency analysis for loop pipelining in High-Level Synthesis". In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC). May 2013, pp. 1–10. DOI: 10.1145/2463209.2488796.

[9] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. "Conversion of control dependence to data dependence". In: *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL '83. New York, NY, USA: Association for Computing Machinery, Jan. 24, 1983, pp. 177–189. ISBN: 978-0-89791-090-3. DOI: 10.1145/567067.567085.

[10] AMD Xilinx. *Vitis High-Level Synthesis User Guide (UG1399)*. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls. 2023.

[11] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. AFIPS '67 (Spring). New York, NY, USA: Association for Computing Machinery, Apr. 18, 1967, pp. 483–485. ISBN: 978-1-4503-7895-6. DOI: 10.1145/1465482.1465560.

[12] Hela Belhadj Amor, Carolynn Bernier, and Zdeněk Přikryl. "A RISC-V ISA Extension for Ultra-Low Power IoT Wireless Signal Processing". In: *IEEE Transactions on Computers* 71.4 (Apr. 2022), pp. 766–778. ISSN: 1557-9956. DOI: 10.1109/TC.2021.3063027.

[13] F Anceau, P Liddell, J Mermet, and Ch Payan. "CASSANDRE: a language to describe digital systems, application to logic design". In: *Software Engineering: Proceedings of the Third Symposium on Computer and Information Sciences held in Miami Beach, Florida*. 1969.

[14] Renzo Andri, Tomas Henriksson, and Luca Benini. "Extending the RISC-V ISA for Efficient RNN-based 5G Radio Resource Management". In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020 57th ACM/IEEE Design Automation Conference (DAC). July 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218496.

[15] Ali Ansari, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. "Divide and Conquer Frontend Bottleneck". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). May 2020, pp. 65–78. DOI: 10.1109/ISCA45697.2020.00017.

[16] Guido Araujo, Sandro Rigo, and Rodolfo Azevedo. "Chapter 11 - Processor Design with ArchC". In: *Processor Description Languages*. Ed. by Prabhat Mishra and Nikil Dutt. Vol. 1. Systems on Silicon. Burlington: Morgan Kaufmann, Jan. 1, 2008, pp. 275–294. DOI: 10.1016/B978-012374287-2.50014-8.

[17] Manuel Arenaz, Juan Touriño, and Ramón Doallo. "A GSA-based compiler infrastructure to extract parallelism from complex loops". In: *Proceedings of the 17th annual international conference on Supercomputing*. ICS '03. New York, NY, USA: Association for Computing Machinery, June 23, 2003, pp. 193–204. ISBN: 978-1-58113-733-0. DOI: 10.1145/782814.782842.

[18] Arvind, R.S. Nikhil, D.L. Rosenband, and N. Dave. "High-level synthesis: an essential ingredient for designing complex ASICs". In: *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.* IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004. Nov. 2004, pp. 775–782. DOI: 10.1109/ICCAD.2004.1382681.

[19] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. Apr. 2016.

[20] Matthew Aubury, Ian Page, Geoff Randall, Jonathan Saul, and Robin Watts. "Handel-C language reference guide". In: *Computing Laboratory. Oxford University, UK* 12 (1996).

[21] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. "Lime: a Java-compatible and synthesizable language for heterogeneous architectures". In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. OOPSLA '10. New York, NY, USA: Association for Computing Machinery, Oct. 17, 2010, pp. 89–108. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869469.

[22] T.M. Austin. "DIVA: a reliable substrate for deep submicron microarchitecture design". In: *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture. Nov. 1999, pp. 196–207. DOI: 10.1109/MICRO.1999.809458.

[23] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. "An OpenCL™ Deep Learning Accelerator on Arria 10". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. New York, NY, USA: Association for Computing Machinery, Feb. 22, 2017, pp. 55–64. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021738.

[24] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. "The ArchC Architecture Description Language and Tools". In: *International Journal of Parallel Programming* 33.5 (Oct. 1, 2005), pp. 453–484. ISSN: 1573-7640. DOI: 10.1007/s10766-005-7301-0.

[25] 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: *DAC Design Automation Conference 2012*. DAC Design Automation Conference 2012. June 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[26] Mark W. Bailey. "CSDL: reusable computing system descriptions for retargetable systems software". PhD thesis. USA: University of Virginia, 2000. 150 pp.

[27] Nirmalya Bandyopadhyay, Kanad Basu, and Prabhat Mishra. "Chapter 14 - HMDES, ISDL, and Other Contemporary ADLs". In: *Processor Description Languages*. Ed. by Prabhat Mishra and Nikil Dutt. Vol. 1. Systems on Silicon. Burlington: Morgan Kaufmann, Jan. 1, 2008, pp. 369–394. DOI: 10.1016/B978-012374287-2.50017-3.

[28] Mehmet B. Baray and Stephen Y. H. Su. "A digital system modeling philosophy and design language". In: *Proceedings of the 8th Design Automation Workshop*. DAC '71. New York, NY, USA: Association for Computing Machinery, June 28, 1971, pp. 1–22. ISBN: 978-1-4503-7465-1. DOI: 10.1145/800158.805056.

[29] M.R. Barbacci. "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems". In: *IEEE Transactions on Computers* C-24.2 (Feb. 1975), pp. 137–150. ISSN: 1557-9956. DOI: 10.1109/T-C.1975.224181.

[30] Mario R. Barbacci. "Instruction set processor specifications (ISPS): The notation and its applications". In: *IEEE Transactions on Computers* C-30.1 (Jan. 1981), pp. 24–40. ISSN: 1557-9956. DOI: 10.1109/TC.1981.6312154.

[31] C. Gordon Bell and Allen Newell. "The PMS and ISP descriptive systems for computer structures". In: *Proceedings of the May 5-7, 1970, spring joint computer conference*. AFIPS '70 (Spring). New York, NY, USA: Association for Computing Machinery, May 5, 1970, pp. 351–374. ISBN: 978-1-4503-7903-8. DOI: 10.1145/1476936.1476993.

[32] CG Bell, AG Jordan, and JF Traub. "Register Transfer (RT) level: Components, Representation and Design techniques". In: *Proposal from the Electrical Engineering and Computer Science Departments, CMU, to the National Science Foundation (Computer Systems Design Program), CMU proposal* 1582 (1971).

[33] Fabrice Bellard. "QEMU, a fast and portable dynamic translator". In: *Proceedings of the annual conference on USENIX Annual Technical Conference*. ATEC '05. USA: USENIX Association, Apr. 10, 2005, p. 41.

[34] R.A. Bergamaschi, S. Raje, I. Nair, and L. Trevillyan. "Control-flow versus data-flow-based scheduling: combining both approaches in an adaptive scheduling system". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 5.1 (Mar. 1997), pp. 82–100. ISSN: 1557-9999. DOI: 10.1109/92.555989.

[35] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAS*. Boston, MA: Springer US, 1999. ISBN: 978-1-4613-7342-1 978-1-4615-5145-4. DOI: 10.1007/978-1-4615-5145-4.

[36] Anasua Bhowmik and Manoj Franklin. "A general compiler framework for speculative multithreading". In: *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. SPAA '02. New York, NY, USA: Association for Computing Machinery, Aug. 10, 2002, pp. 99–108. ISBN: 978-1-58113-529-9. DOI: 10.1145/564870.564885.

[37] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. "The gem5 simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 31, 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718.

[38] David C. Black and Jack Donovan, eds. *SystemC: From the Ground Up*. Boston: Kluwer Academic Publishers, 2004. ISBN: 978-0-387-29240-3. DOI: 10.1007/0-387-30864-4.

[39] *Bluespec Compiler (BSC)*. https://github.com/B-Lang-org/bsc. Aug. 28, 2024.

[40] Bochmann. "Hardware Specification with Temporal Logic: An Example". In: *IEEE Transactions on Computers* C-31.3 (Mar. 1982), pp. 223–231. ISSN: 1557-9956. DOI: 10.1109/TC.1982.1675978.

[41] David Bol, Julien De Vos, François Botman, Guerric de Streel, Sébastien Bernard, Denis Flandre, and Jean-Didier Legat. "Green SoCs for a sustainable Internet-of-Things". In: *2013 IEEE Faible Tension Faible Consommation*. 2013 IEEE Faible Tension Faible Consommation. June 2013, pp. 1–4. DOI: 10.1109/FTFC.2013.6577767.

[42] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A practical automatic polyhedral parallelizer and locality optimizer". In: *ACM SIGPLAN Notices* 43.6 (June 7, 2008), pp. 101–113. ISSN: 0362-1340. DOI: 10.1145/1379022.1375595.

[43] E. Borch, E. Tune, S. Manne, and J. Emer. "Loose loops sink chips". In: *Proceedings Eighth International Symposium on High Performance Computer Architecture*. Proceedings Eighth International Symposium on High Performance Computer Architecture. Feb. 2002, pp. 299–310. DOI: 10.1109/HPCA.2002.995719.

[44] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. "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*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, June 11, 2020, pp. 243–257. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385965.

[45] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. "The Marion system for retargetable instruction scheduling". In: *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. PLDI '91. New York, NY, USA: Association for Computing Machinery, May 1, 1991, pp. 229–240. ISBN: 978-0-89791-428-4. DOI: 10.1145/113445.113465.

[46] M. Budiu, P.V. Artigas, and S.C. Goldstein. "Dataflow: A Complement to Superscalar". In: *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005. Mar. 2005, pp. 177–186. DOI: 10.1109/ISPASS.2005.1430572.

[47] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. "Control-Flow Integrity: Precision, Security, and Performance". In: *ACM Comput. Surv.* 50.1 (Apr. 4, 2017), 16:1–16:33. ISSN: 0360-0300. DOI: 10.1145/3054924.

[48] R. Camposano. "Path-based scheduling for synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10.1 (Jan. 1991), pp. 85–93. ISSN: 1937-4151. DOI: 10.1109/43.62794.

[49] Andrew Canis, Stephen D. Brown, and Jason H. Anderson. "Modulo SDC scheduling with recurrence minimization in high-level synthesis". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014 24th International Conference on Field Programmable Logic and Applications (FPL). Sept. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927490.

[50] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. "LegUp: high-level synthesis for FPGA-based processor/accelerator systems". In: *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. FPGA '11. New York, NY, USA: Association for Computing Machinery, Feb. 27, 2011, pp. 33–36. ISBN: 978-1-4503-0554-9. DOI: 10.1145/1950413.1950423.

[51] L.P. Carloni, K.L. McMillan, A. Saldanha, and A.L. Sangiovanni-Vincentelli. "A methodology for correct-by-construction latency insensitive design". In: *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051)*. 1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051). Nov. 1999, pp. 309–315. DOI: 10.1109/ICCAD.1999.810667.

[52] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. "Theory of latency-insensitive design". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20.9 (Sept. 2001), pp. 1059–1076. ISSN: 1937-4151. DOI: 10.1109/43.945302.

[53] Luca P. Carloni. "From Latency-Insensitive Design to Communication-Based System-Level Design". In: *Proceedings of the IEEE* 103.11 (Nov. 2015), pp. 2133–2151. ISSN: 1558-2256. DOI: 10.1109/JPROC.2015.2480849.

[54] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. New York, NY, USA: Association for Computing Machinery, Nov. 12, 2011, pp. 1–12. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063454.

[55] Christopher Patrick Celio. "A Highly Productive Implementation of an Out-of-Order Processor Generator". PhD thesis. UC Berkeley, 2017.

[56] Gino Chacon, Elba Garza, Alexandra Jimborean, Alberto Ros, Paul V. Gratz, Daniel A. Jiménez, and Samira Mirbagher-Ajorpaz. "Composite Instruction Prefetching". In: *2022 IEEE 40th International Conference on Computer Design (ICCD)*. 2022 IEEE 40th International Conference on Computer Design (ICCD). Oct. 2022, pp. 471–478. DOI: 10.1109/ICCD56317.2022.00076.

[57] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. "Register allocation via coloring". In: *Computer Languages* 6.1 (Jan. 1, 1981), pp. 47–57. ISSN: 0096-0551. DOI: 10.1016/0096-0551(81)90048-5.

[58] Anupam Chattopadhyay, Heinrich Meyr, and Rainer Leupers. "Chapter 5 - LISA: A Uniform ADL for Embedded Processor Modeling, Implementation, and Software Toolsuite Generation". In: *Processor Description Languages*. Ed. by Prabhat Mishra and Nikil Dutt. Vol. 1. Systems on Silicon. Burlington: Morgan Kaufmann, Jan. 1, 2008, pp. 95–132. DOI: 10.1016/B978-012374287-2.50008-2.

[59] Hanqiu Chen and Cong Hao. "DGNN-Booster: A Generic FPGA Accelerator Framework For Dynamic Graph Neural Network Inference". In: *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). May 2023, pp. 195–201. DOI: 10.1109/FCCM57271.2023.00029.

[60] Yu-Ting Chen, Jason Cong, Jie Lei, and Peng Wei. "A Novel High-Throughput Acceleration Engine for Read Alignment". In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines. May 2015, pp. 199–202. DOI: 10.1109/FCCM.2015.27.

[61] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. "ThunderGP: HLS-based Graph Processing Framework on FPGAs". In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '21. New York, NY, USA: Association for Computing Machinery, Feb. 17, 2021, pp. 69–80. ISBN: 978-1-4503-8218-2. DOI: 10.1145/3431920.3439290.

[62] Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, and Deming Chen. "Cloud-DNN: An Open Framework for Mapping DNN Models to Cloud FPGAs". In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '19. New York, NY, USA: Association for Computing Machinery, Feb. 20, 2019, pp. 73–82. ISBN: 978-1-4503-6137-8. DOI: 10.1145/3289602.3293915.

[63] Jianyi Cheng, Lana Josipović, George A. Constantinides, Paolo Ienne, and John Wickerson. "Combining Dynamic & Static Scheduling in High-level Synthesis". In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '20. New York, NY, USA: Association for Computing Machinery, Feb. 24, 2020, pp. 288–298. ISBN: 978-1-4503-7099-8. DOI: 10.1145/3373087.3375297.

[64] Jianyi Cheng, Lana Josipović, George A. Constantinides, Paolo Ienne, and John Wickerson. "DASS: Combining Dynamic & Static Scheduling in High-Level Synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.3 (Mar. 2022), pp. 628–641. ISSN: 1937-4151. DOI: 10.1109/TCAD.2021.3065902.

[65] Jianyi Cheng, Lana Josipović, George A. Constantinides, and John Wickerson. "Dynamic Inter-Block Scheduling for HLS". In: *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 2022 32nd International Conference on Field-Programmable Logic and Applications (FPL). Aug. 2022, pp. 243–252. DOI: 10.1109/FPL57034.2022.00045.

[66] Jianyi Cheng, Lana Josipović, John Wickerson, and George A. Constantinides. "Parallelising Control Flow in Dynamic-scheduling High-level Synthesis". In: *ACM Transactions on Reconfigurable Technology and Systems* 16.4 (Sept. 1, 2023), 55:1–55:32. ISSN: 1936-7406. DOI: 10.1145/3599973.

[67] Jianyi Cheng, John Wickerson, and George A. Constantinides. "Finding and Finessing Static Islands in Dynamically Scheduled Circuits". In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '22. New York, NY, USA: Association for Computing Machinery, Feb. 11, 2022, pp. 89–100. ISBN: 978-1-4503-9149-8. DOI: 10.1145/3490422.3502362.

[68] Jianyi Cheng, John Wickerson, and George A. Constantinides. "Probabilistic Scheduling in High-Level Synthesis". In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). May 2021, pp. 195–203. DOI: 10.1109/FCCM51124.2021.00031.

[69] Shaoyi Cheng and John Wawrzynek. "Architectural synthesis of computational pipelines with decoupled memory access". In: *2014 International Conference on Field-Programmable Technology (FPT)*. 2014 International Conference on Field-Programmable Technology (FPT). Dec. 2014, pp. 83–90. DOI: 10.1109/FPT.2014.7082758.

[70] David Chisnall. *How to Design an ISA - ACM Queue*. Jan. 11, 2024. URL: https://queue.acm.org/detail.cfm?id=3639445 (visited on 07/22/2024).

[71] Young Kyu Choi, Jason Cong, and Di Wu. "FPGA Implementation of EM Algorithm for 3D CT Reconstruction". In: *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines. May 2014, pp. 157–160. DOI: 10.1109/FCCM.2014.48.

[72] Eden Chung. *Press Center - Global Top 10 Foundries Q4 Revenue Up 7.9%, Annual Total Hits US$111.54 Billion in 2023, Says TrendForce — TrendForce - Market research, price trend of DRAM, NAND Flash, LEDs, TFT-LCD and green energy, PV*. TrendForce. Mar. 12, 2024. URL: https://www.trendforce.com/presscenter/news/20240312-12072.html (visited on 07/24/2024).

[73] R.J. Cloutier and D.E. Thomas. "Synthesis of Pipelined Instruction Set Processors". In: *30th ACM/IEEE Design Automation Conference*. 30th ACM/IEEE Design Automation Conference. June 1993, pp. 583–588. DOI: 10.1145/157485.165053.

[74] Marco Cococcioni, Federico Rossi, Emanuele Ruffaldi, and Sergio Saponara. "A Lightweight Posit Processing Unit for RISC-V Processors in Deep Neural Network Applications". In: *IEEE Transactions on Emerging Topics in Computing* 10.4 (Oct. 2022), pp. 1898–1908. ISSN: 2168-6750. DOI: 10.1109/TETC.2021.3120538.

[75] David R. Coelho. *The VHDL Handbook*. Boston, MA: Springer US, 1989. ISBN: 978-1-4612-8902-9 978-1-4613-1633-6. DOI: 10.1007/978-1-4613-1633-6.

[76] Davide Conficconi, Eleonora D'Arnese, Emanuele Del Sozzo, Donatella Sciuto, and Marco D. Santambrogio. "A Framework for Customizable FPGA-based Image Registration Accelerators". In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '21. New York, NY, USA: Association for Computing Machinery, Feb. 17, 2021, pp. 251–261. ISBN: 978-1-4503-8218-2. DOI: 10.1145/3431920.3439291.

[77] J. Cong and Zhiru Zhang. "An efficient and versatile scheduling algorithm based on SDC formulation". In: *2006 43rd ACM/IEEE Design Automation Conference*. 2006 43rd ACM/IEEE Design Automation Conference. July 2006, pp. 433–438. DOI: 10.1145/1146909.1147025.

[78] Jason Cong, Licheng Guo, Po-Tsang Huang, Peng Wei, and Tianhe Yu. "SMEM++: A Pipelined and Time-Multiplexed SMEM Seeding Accelerator for DNA Sequencing". In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). Apr. 2018, pp. 206–206. DOI: 10.1109/FCCM.2018.00040.

[79] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. "FPGA HLS Today: Successes, Challenges, and Opportunities". In: *ACM Transactions on Reconfigurable Technology and Systems* 15.4 (Aug. 8, 2022), 51:1–51:42. ISSN: 1936-7406. DOI: 10.1145/3530775.

[80] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. "High-Level Synthesis for FPGAs: From Prototyping to Deployment". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.4 (Apr. 2011), pp. 473–491. ISSN: 1937-4151. DOI: 10.1109/TCAD.2011.2110592.

[81] Keith D. Cooper and Linda Torczon. "Chapter 10 - Scalar Optimization". In: *Engineering a Compiler (Third Edition)*. Ed. by Keith D. Cooper and Linda Torczon. Philadelphia: Morgan Kaufmann, Jan. 1, 2023, pp. 517–573. ISBN: 978-0-12-815412-0. DOI: 10.1016/B978-0-12-815412-0.00016-4.

[82] Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. "Synthesis of synchronous elastic architectures". In: *Proceedings of the 43rd annual Design Automation Conference*. DAC '06. New York, NY, USA: Association for Computing Machinery, July 24, 2006, pp. 657–662. ISBN: 978-1-59593-381-2. DOI: 10.1145/1146909.1147077.

[83] Enfang Cui, Tianzheng Li, and Qian Wei. "RISC-V Instruction Set Architecture Extensions: A Survey". In: *IEEE Access* 11 (2023), pp. 24696–24711. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3246491.

[84] Serena Curzel, Sofija Jovic, Michele Fiorito, Antonino Tumeo, and Fabrizio Ferrandi. "Higher-Level Synthesis: experimenting with MLIR polyhedral representations for accelerator design". In: *IMPACT 2022: 12th International Workshop on Polyhedral Compilation Techniaues*. 2022, pp. 1–10.

[85] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Transactions on Programming Languages and Systems* 13.4 (Oct. 1, 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320.

[86] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* 5.1 (Jan. 1998), pp. 46–55. ISSN: 1558-190X. DOI: 10.1109/99.660313.

[87] Steve Dai, Gai Liu, Ritchie Zhao, and Zhiru Zhang. "Enabling adaptive loop pipelining in high-level synthesis". In: *2017 51st Asilomar Conference on Signals, Systems, and Computers*. 2017 51st Asilomar Conference on Signals, Systems, and Computers. Oct. 2017, pp. 131–135. DOI: 10.1109/ACSSC.2017.8335152.

[88] Steve Dai, Mingxing Tan, Kecheng Hao, and Zhiru Zhang. "Flushing-Enabled Loop Pipelining for High-Level Synthesis". In: *Proceedings of the 51st Annual Design Automation Conference*. DAC '14. New York, NY, USA: Association for Computing Machinery, June 1, 2014, pp. 1–6. ISBN: 978-1-4503-2730-5. DOI: 10.1145/2593069.2593143.

[89] Steve Dai, Ritchie Zhao, Gai Liu, Shreesha Srinath, Udit Gupta, Christopher Batten, and Zhiru Zhang. "Dynamic Hazard Resolution for Pipelining Irregular Loops in High-Level Synthesis". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. New York, NY, USA: Association for Computing Machinery, Feb. 22, 2017, pp. 189–194. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021754.

[90] William J. Dally, Stephen W. Keckler, and David B. Kirk. "Evolution of the Graphics Processing Unit (GPU)". In: *IEEE Micro* 41.6 (Nov. 2021), pp. 42–51. ISSN: 1937-4143. DOI: 10.1109/MM.2021.3113475.

[91] Wafi Danesh, Joshua Banago, and Mostafizur Rahman. "Turning the Table: Using Bitstream Reverse Engineering to Detect FPGA Trojans". In: *Journal of Hardware and Systems Security* 5.3 (Dec. 1, 2021), pp. 237–246. ISSN: 2509-3436. DOI: 10.1007/s41635-021-00122-4.

[92] Jean-Luc Danger, Adrien Facon, Sylvain Guilley, Karine Heydemann, Ulrich Kühne, Abdelmalek Si Merabet, and Michaël Timbert. "CCFI-Cache: A Transparent and Flexible Hardware Protection for Code and Control-Flow Integrity". In: *2018 21st Euromicro Conference on Digital System Design (DSD)*. 2018 21st Euromicro Conference on Digital System Design (DSD). Aug. 2018, pp. 529–536. DOI: 10.1109/DSD.2018.00093.

[93] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications". In: *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS). Sept. 2017, pp. 1–8. DOI: 10.1109/PATMOS.2017.8106976.

[94] Scott Davidson. "Electronic Design Automation for IC Implementation, Circuit Design, and Process Technology and Electronic Design Automation for IC System Design, Verification, and Testing". In: *IEEE Design & Test* 35.3 (June 2018), pp. 98–99. ISSN: 2168-2364. DOI: 10.1109/MDAT.2018.2814988.

[95] J.C. Dehnert, B.K. Grant, J.P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. "The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges". In: *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* International Symposium on Code Generation and Optimization, 2003. CGO 2003. Mar. 2003, pp. 15–24. DOI: 10.1109/CGO.2003.1191529.

[96] Alberto A. Del Barrio, Maria C. Molina, Jose M. Mendias, Roman Hermida, and Seda Ogrenci Memik. "Using Speculative Functional Units in high level synthesis". In: *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010). Mar. 2010, pp. 1779–1784. DOI: 10.1109/DATE.2010.5457102.

[97] Steven Derrien, Thibaut Marty, Simon Rokicki, and Tomofumi Yuki. "Toward Speculative Loop Pipelining for High-Level Synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (Nov. 2020), pp. 4229–4239. ISSN: 1937-4151. DOI: 10.1109/TCAD.2020.3012866.

[98] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. "PQEMU: A Parallel System Emulator Based on QEMU". In: *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. 2011 IEEE 17th International Conference on Parallel and Distributed Systems. Dec. 2011, pp. 276–283. DOI: 10.1109/ICPADS.2011.102.

[99] Ralf Dreesen. "Generating interlocked instruction pipelines from specifications of instruction sets". In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. CODES+ISSS '12. New York, NY, USA: Association for Computing Machinery, Oct. 7, 2012, pp. 285–294. ISBN: 978-1-4503-1426-8. DOI: 10.1145/2380445.2380492.

[100] Ralf Dreesen. "ViDL: A Versatile ISA Description Language". In: *2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems*. 2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems. Apr. 2012, pp. 222–231. DOI: 10.1109/ECBS.2012.49.

[101] Ulrich Drepper. "What every programmer should know about memory". In: *Red Hat, Inc* 11.2007 (2007), p. 2007.

[102] Akshay Dua, Yixing Li, and Fengbo Ren. "Systolic-CNN: An OpenCL-defined Scalable Runtime-flexible FPGA Accelerator Architecture for Accelerating Convolutional Neural Network Inference in Cloud/Edge Computing". In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). May 2020, pp. 231–231. DOI: 10.1109/FCCM48280.2020.00064.

[103] 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*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, June 11, 2020, pp. 408–422. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385983.

[104] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye. "Dynamic binary translation and optimization". In: *IEEE Transactions on Computers* 50.6 (June 2001), pp. 529–548. ISSN: 1557-9956. DOI: 10.1109/12.931892.

[105] EETimes. *Behavioral synthesis crossroad*. EE Times. Apr. 5, 2004. URL: https://www.eetimes.com/behavioral-synthesis-crossroad/ (visited on 08/08/2024).

[106] Virantha Ekanayake, Clinton Kelly, and Rajit Manohar. "An ultra low-power processor for sensor networks". In: *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. ASPLOS XI. New York, NY, USA: Association for Computing Machinery, Oct. 7, 2004, pp. 27–36. ISBN: 978-1-58113-804-7. DOI: 10.1145/1024393.1024397.

[107] Ayatallah Elakhras, Andrea Guerrieri, Lana Josipović, and Paolo Ienne. "Survival of the Fastest: Enabling More Out-of-Order Execution in Dataflow Circuits". In: *Proceedings of the 2024 ACM SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '24. New York, NY, USA: Association for Computing Machinery, Apr. 2, 2024, pp. 44–54. ISBN: 9798400704185. DOI: 10.1145/3626202.3637556.

[108] Ayatallah Elakhras, Andrea Guerrieri, Lana Josipović, and Paolo Ienne. "Unleashing Parallelism in Elastic Circuits with Faster Token Delivery". In: *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 2022 32nd International Conference on Field-Programmable Logic and Applications (FPL). Aug. 2022, pp. 253–261. DOI: 10.1109/FPL57034.2022.00046.

[109] Ayatallah Elakhras, Riya Sawhney, Andrea Guerrieri, Lana Josipović, and Paolo Ienne. "Straight to the Queue: Fast Load-Store Queue Allocation in Dataflow Circuits". In: *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '23. New York, NY, USA: Association for Computing Machinery, Feb. 12, 2023, pp. 39–45. ISBN: 978-1-4503-9417-8. DOI: 10.1145/3543622.3573050.

[110] John P. Elliott. *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design*. USA: Kluwer Academic Publishers, Apr. 1999. ISBN: 978-0-7923-8542-4.

[111] Alexis Engelke, Dominik Okwieka, and Martin Schulz. "Efficient LLVM-based dynamic binary translation". In: *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE 2021. New York, NY, USA: Association for Computing Machinery, Apr. 7, 2021, pp. 165–171. ISBN: 978-1-4503-8394-3. DOI: 10.1145/3453933.3454022.

[112] Leah Epstein. "List Scheduling". In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. Boston, MA: Springer US, 2008, pp. 455–457. ISBN: 978-0-387-30162-4. DOI: 10.1007/978-0-387-30162-4_205.

[113] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark silicon and the end of multicore scaling". In: *Proceedings of the 38th annual international symposium on Computer architecture*. ISCA '11. New York, NY, USA: Association for Computing Machinery, June 4, 2011, pp. 365–376. ISBN: 978-1-4503-0472-6. DOI: 10.1145/2000064.2000108.

[114] Umer Farooq, Zied Marrakchi, and Habib Mehrez. *Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*. New York, NY: Springer, 2012. ISBN: 978-1-4614-3593-8 978-1-4614-3594-5. DOI: 10.1007/978-1-4614-3594-5.

[115] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. "Proactive instruction fetch". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. New York, NY, USA: Association for Computing Machinery, Dec. 3, 2011, pp. 152–162. ISBN: 978-1-4503-1053-6. DOI: 10.1145/2155620.2155638.

[116] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. "Temporal instruction fetch streaming". In: *2008 41st IEEE/ACM International Symposium on Microarchitecture*. 2008 41st IEEE/ACM International Symposium on Microarchitecture. Nov. 2008, pp. 1–10. DOI: 10.1109/MICRO.2008.4771774.

[117] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. "Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021 58th ACM/IEEE Design Automation Conference (DAC). Dec. 2021, pp. 1327–1330. DOI: 10.1109/DAC18074.2021.9586110.

[118] Corentin Ferry. "Automating the derivation of memory allocations for acceleration of polyhedral programs". thesis. Université de Rennes, Feb. 19, 2024.

[119] Corentin Ferry, Tomofumi Yuki, Steven Derrien, and Sanjay Rajopadhye. "Increasing FPGA Accelerators Memory Bandwidth With a Burst-Friendly Memory Layout". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42.5 (May 2023), pp. 1546–1559. ISSN: 1937-4151. DOI: 10.1109/TCAD.2022.3201494.

[120] Nicolai Fiege, Patrick Sittel, and Peter Zipf. "Improving Energy Efficiency in Loop Pipelining by Rational-II Modulo Scheduling". In: *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). May 2022, pp. 1–2. DOI: 10.1109/FCCM53951.2022.9786117.

[121] Nicolai Fiege, Patrick Sittel, and Peter Zipf. "Speeding Up Optimal Modulo Scheduling with Rational Initiation Intervals". In: *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 2022 32nd International Conference on Field-Programmable Logic and Applications (FPL). Aug. 2022, pp. 322–326. DOI: 10.1109/FPL57034.2022.00056.

[122] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. "Transformations of High-Level Synthesis Codes for High-Performance Computing". In: *IEEE Transactions on Parallel and Distributed Systems* 32.5 (May 2021), pp. 1014–1029. ISSN: 1558-2183. DOI: 10.1109/TPDS.2020.3039409.

[123] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. "VLIW processors: once blue sky, now commonplace". In: *IEEE Solid-State Circuits Magazine* 1.2 (2009), pp. 10–17. ISSN: 1943-0590. DOI: 10.1109/MSSC.2009.932433.

[124] Antoine Floc'h, Tomofumi Yuki, Ali El-Moussawi, Antoine Morvan, Kevin Martin, Maxime Naullet, Mythri Alle, Ludovic L'Hours, Nicolas Simon, Steven Derrien, François Charot, Christophe Wolinski, and Olivier Sentieys. "GeCoS: A framework for prototyping custom hardware design flows". In: *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM). Sept. 2013, pp. 100–105. DOI: 10.1109/SCAM.2013.6648190.

[125] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. "Extending the RISC-V Instruction Set for Hardware Acceleration of the Post-Quantum Scheme LAC". In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). Mar. 2020, pp. 1420–1425. DOI: 10.23919/DATE48585.2020.9116567.

[126] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (Oct. 2017), pp. 2700–2713. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2017.2654506.

[127] 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: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014 24th International Conference on Field Programmable Logic and Applications (FPL). Sept. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927454.

[128] Anwar M. Ghuloum and Allan L. Fisher. "Flattening and parallelizing irregular, recurrent loop nests". In: *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPOPP '95. New York, NY, USA: Association for Computing Machinery, Aug. 1, 1995, pp. 58–67. ISBN: 978-0-89791-700-1. DOI: 10.1145/209936.209944.

[129] Steve Golson and Leah Clark. "Language Wars in the 21st Century: Verilog versus VHDL–Revisited". In: *Synopsys Users Group (SNUG)* (2016).

[130] Bernard Goossens. *Guide to Computer Processor Architecture: A RISC-V Approach, with High-Level Synthesis*. Undergraduate Topics in Computer Science. Cham: Springer International Publishing, 2023. ISBN: 978-3-031-18022-4 978-3-031-18023-1. DOI: 10.1007/978-3-031-18023-1.

[131] Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. "A Unified Memory Dependency Framework for Speculative High-Level Synthesis". In: *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. CC 2024. New York, NY, USA: Association for Computing Machinery, Feb. 20, 2024, pp. 13–25. ISBN: 9798400705076. DOI: 10.1145/3640537.3641581.

[132] Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. "Design Exploration of RISC-V Soft-Cores through Speculative High-Level Synthesis". In: *2022 International Conference on Field-Programmable Technology (ICFPT)*. 2022 International Conference on Field-Programmable Technology (ICFPT). Dec. 2022, pp. 1–6. DOI: 10.1109/ICFPT56656.2022.9974478.

[133] Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. "SpecHLS: Speculative Accelerator Design Using High-Level Synthesis". In: *IEEE Micro* 42.5 (Sept. 2022), pp. 99–107. ISSN: 1937-4143. DOI: 10.1109/MM.2022.3188136.

[134] Bita Gorjiara, Mehrdad Reshadi, and Daniel Gajski. "Chapter 13 - GNR: A Formal Language for Specification, Compilation, and Synthesis of Custom Embedded Processors". In: *Processor Description Languages*. Ed. by Prabhat Mishra and Nikil Dutt. Vol. 1. Systems on Silicon. Burlington: Morgan Kaufmann, Jan. 1, 2008, pp. 329–367. DOI: 10.1016/B978-012374287-2.50016-1.

[135] Kathryn E Gray, Peter Sewell, Christopher Pulte, Shaked Flur, and Robert Norton-Wright. *The Sail instruction-set semantics specification language*. Tech. rep. Technical report, Cambridge University, 2017. 86 BIBLIOGRAPHY, 2017.

[136] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. "Polly – performing polyhedral optimizations on a low-level intermediate representation". In: *Parallel Processing Letters* 22.4 (Dec. 2012), p. 1250010. ISSN: 0129-6264. DOI: 10.1142/S0129626412500107.

[137] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates". In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). Apr. 2017, pp. 152–159. DOI: 10.1109/FCCM.2017.25.

[138] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. "Hardware-Software Contracts for Secure Speculation". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021 IEEE Symposium on Security and Privacy (SP). May 2021, pp. 1868–1883. DOI: 10.1109/SP40001.2021.00036.

[139] Andrea Guerrieri, Srijeet Guha, Lana Josipović, and Paolo Ienne. "DynaRapid: From C to FPGA in a Few Seconds". In: *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '24. New York, NY, USA: Association for Computing Machinery, Apr. 2, 2024, p. 40. ISBN: 9798400704185. DOI: 10.1145/3626202.3637580.

[140] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. "Hardware Acceleration of Long Read Pairwise Overlapping in Genome Sequencing: A Race Between FPGA and GPU". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). Apr. 2019, pp. 127–135. DOI: 10.1109/FCCM.2019.00027.

[141] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations". In: *16th International Conference on VLSI Design, 2003. Proceedings.* 16th International Conference on VLSI Design, 2003. Proceedings. Jan. 2003, pp. 461–466. DOI: 10.1109/ICVD.2003.1183177.

[142] Sukrat Gupta, Neel Gala, G. S. Madhusudan, and V. Kamakoti. "SHAKTI-F: A Fault Tolerant Microprocessor Architecture". In: *2015 IEEE 24th Asian Test Symposium (ATS)*. 2015 IEEE 24th Asian Test Symposium (ATS). Nov. 2015, pp. 163–168. DOI: 10.1109/ATS.2015.35.

[143] Sumit Gupta, Mehrdad Reshadi, Nick Savoiu, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. "Dynamic common sub-expression elimination during scheduling in high-level synthesis". In: *Proceedings of the 15th international symposium on System Synthesis*. ISSS '02. New York, NY, USA: Association for Computing Machinery, Oct. 2, 2002, pp. 261–266. ISBN: 978-1-58113-576-3. DOI: 10.1145/581199.581256.

[144] Sumit Gupta, Nick Savoiu, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. "Conditional speculation and its effects on performance and area for high-level synthesis". In: *Proceedings of the 14th international symposium on Systems synthesis*. ISSS '01. New York, NY, USA: Association for Computing Machinery, Sept. 30, 2001, pp. 171–176. ISBN: 978-1-58113-418-6. DOI: 10.1145/500001.500040.

[145] Sumit Gupta, Nick Savoiu, Sunwoo Kim, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. "Speculation techniques for high level synthesis of control intensive designs". In: *Proceedings of the 38th annual Design Automation Conference*. DAC '01. New York, NY, USA: Association for Computing Machinery, June 22, 2001, pp. 269–272. ISBN: 978-1-58113-297-7. DOI: 10.1145/378239.378481.

[146] Stefan Hadjis, Andrew Canis, Jason H. Anderson, Jongsok Choi, Kevin Nam, Stephen Brown, and Tomasz Czajkowski. "Impact of FPGA architecture on resource sharing in high-level synthesis". In: *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. FPGA '12. New York, NY, USA: Association for Computing Machinery, Feb. 22, 2012, pp. 111–114. ISBN: 978-1-4503-1155-7. DOI: 10.1145/2145694.2145712.

[147] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. "ISDL: an instruction set description language for retargetability". In: *Proceedings of the 34th annual Design Automation Conference*. DAC '97. New York, NY, USA: Association for Computing Machinery, June 13, 1997, pp. 299–302. ISBN: 978-0-89791-920-3. DOI: 10.1145/266021.266108.

[148] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. "ISDL: An Instruction Set Description Language for Retargetability and Architecture Exploration". In: *Design Automation for Embedded Systems* 6.1 (Sept. 1, 2000), pp. 39–69. ISSN: 1572-8080. DOI: 10.1023/A:1008937425064.

[149] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. "CHStone: A benchmark program suite for practical C-based high-level synthesis". In: *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2008 IEEE International Symposium on Circuits and Systems (ISCAS). May 2008, pp. 1192–1195. DOI: 10.1109/ISCAS.2008.4541637.

[150] Yuko Hara-Azumi, Toshinobu Matsuba, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. "Impact of Resource Sharing and Register Retiming on Area and Performance of FPGA-based Designs". In: *Information and Media Technologies* 9.1 (2014), pp. 26–34. DOI: 10.11185/imt.9.26.

[151] Austin Harris, Tarunesh Verma, Shijia Wei, Lauren Biernacki, Alex Kisil, Misiker Tadesse Aga, Valeria Bertacco, Baris Kasikci, Mohit Tiwari, and Todd Austin. "Morpheus II: A RISC-V Security Extension for Protecting Vulnerable Software and Hardware". In: *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). Dec. 2021, pp. 226–238. DOI: 10.1109/HOST49136.2021.9702275.

[152] Sarah L. Harris and David Harris. "7 - Microarchitecture". In: *Digital Design and Computer Architecture*. Ed. by Sarah L. Harris and David Harris. Morgan Kaufmann, Jan. 1, 2022, pp. 392–497. ISBN: 978-0-12-820064-3. DOI: 10.1016/B978-0-12-820064-3.00007-6.

[153] Michael T Heath. "A tale of two laws". In: *The International Journal of High Performance Computing Applications* 29.3 (Aug. 1, 2015), pp. 320–330. ISSN: 1094-3420. DOI: 10.1177/1094342015572031.

[154] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Nov. 2017. 936 pp. ISBN: 978-0-12-811905-1.

[155] D. Herrmann and R. Ernst. "Register synthesis for speculative computation". In: *Proceedings European Design and Test Conference. ED & TC 97*. Proceedings European Design and Test Conference. ED & TC 97. Mar. 1997, pp. 463–467. DOI: 10.1109/EDTC.1997.582401.

[156] Simon Himmelbauer, Christoph Hochrainer, Benedikt Huber, Niklas Mischkulnig, Philipp Paulweber, Tobias Schwarzinger, and Andreas Krall. *The Vienna Architecture Description Language*. Feb. 14, 2024. DOI: 10.48550/arXiv.2402.09087. arXiv: 2402.09087[cs].

[157] U. Holtmann and R. Ernst. "Combining MBP-speculative computation and loop pipelining in high-level synthesis". In: *Proceedings the European Design and Test Conference. ED&TC 1995*. Proceedings the European Design and Test Conference. ED&TC 1995. Mar. 1995, pp. 550–556. DOI: 10.1109/EDTC.1995.470346.

[158] U. Holtmann and R. Ernst. "Experiments with low-level speculative computation based on multiple branch prediction". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 1.3 (Sept. 1993), pp. 262–267. ISSN: 1557-9999. DOI: 10.1109/92.238440.

[159] Steven F. Hoover. "Timing-Abstract Circuit Design in Transaction-Level Verilog". In: *2017 IEEE International Conference on Computer Design (ICCD)*. 2017 IEEE International Conference on Computer Design (ICCD). Nov. 2017, pp. 525–532. DOI: 10.1109/ICCD.2017.91.

[160] Paul Horowitz and Winfield Hill. *The Art of Electronics*. 3rd. USA: Cambridge University Press, Mar. 2015. 1219 pp. ISBN: 978-0-521-80926-9.

[161] Sara Sadat Hoseininasab, Caroline Collange, and Steven Derrien. "Rapid Prototyping of Complex Micro-architectures Through High-Level Synthesis". In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Ed. by Francesca Palumbo, Georgios Keramidas, Nikolaos Voros, and Pedro C. Diniz. Cham: Springer Nature Switzerland, 2023, pp. 19–34. ISBN: 978-3-031-42921-7. DOI: 10.1007/978-3-031-42921-7_2.

[162] I.-J. Huang and A.M. Despain. "Hardware/software resolution of pipeline hazards in pipeline synthesis of instruction set processors". In: *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. Proceedings of 1993 International Conference on Computer Aided Design (ICCAD). Nov. 1993, pp. 594–599. DOI: 10.1109/ICCAD.1993.580120.

[163] S.H. Huang, Y.L. Jeang, C.T. Hwang, Y.C. Hsu, and J.F. Wang. "A Tree-Based Scheduling Algorithm for Control-Dominated Circuits". In: *30th ACM/IEEE Design Automation Conference*. 30th ACM/IEEE Design Automation Conference. June 1993, pp. 578–582. DOI: 10.1145/157485.165051.

[164] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language". In: *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)* (Feb. 2024), pp. 1–1354. DOI: 10.1109/IEEESTD.2024.10458102.

[165] "IEEE Standard for VHDL Language Reference Manual". In: *IEEE Std 1076-2019* (Dec. 2019), pp. 1–673. DOI: 10.1109/IEEESTD.2019.8938196.

[166] "IEEE Standard Graphic Symbols for Logic Functions (Including and incorporating IEEE Std 91a-1991, Supplement to IEEE Standard Graphic Symbols for Logic Functions)". In: *IEEE Std 91a-1991 & IEEE Std 91-1984* (July 1984), pp. 1–160. DOI: 10.1109/IEEESTD.1984.7896954.

[167] "IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language". In: *IEEE Std 1364-1995* (Oct. 1996), pp. 1–688. DOI: 10.1109/IEEESTD.1996.81542.

[168] "IEEE Standard VHDL Language Reference Manual". In: *IEEE Std 1076-1987* (Mar. 1988), pp. 1–218. DOI: 10.1109/IEEESTD.1988.122645.

[169] Akihiko Inoue, Hiroyuki Tomiyama, Takanori Okuma, Hiroyuki Kanbara, and Hiroto Yasuura. "Language and Compiler for Optimizing Datapath Widths of Embedded Systems". In: *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences* E81-A.12 (Dec. 25, 1998), pp. 2595–2604. ISSN: , 0916-8508.

[170] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. "Re-establishing Fetch-Directed Instruction Prefetching: An Industry Perspective". In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Mar. 2021, pp. 172–182. DOI: 10.1109/ISPASS51385.2021.00034.

[171] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations". In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). Nov. 2017, pp. 209–216. DOI: 10.1109/ICCAD.2017.8203780.

[172] H.M. Jacobson, P.N. Kudva, P. Bose, P.W. Cook, S.E. Schuster, E.G. Mercer, and C.J. Myers. "Synchronous interlocked pipelines". In: *Proceedings Eighth International Symposium on Asynchronous Circuits and Systems*. Proceedings Eighth International Symposium on Asynchronous Circuits and Systems. Apr. 2002, pp. 3–12. DOI: 10.1109/ASYNC.2002.1000291.

[173] Richard Jaeger and Travis Blalock. *Microelectronic Circuit Design*. McGraw-Hill Education, Mar. 1, 2010. 1376 pp. ISBN: 978-0-07-338045-2.

[174] R. Jayaseelan, T. Mitra, and Xianfeng Li. "Estimating the Worst-Case Energy Consumption of Embedded Software". In: *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*. 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06). Apr. 2006, pp. 81–90. DOI: 10.1109/RTAS.2006.17.

[175] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. *Dissecting the Graphcore IPU Architecture via Microbenchmarking*. Dec. 6, 2019. DOI: 10.48550/arXiv.1912.03413. arXiv: 1912.03413[cs].

[176] Qiang Jiao, Wei Hu, Fang Liu, and Yong Dong. "RISC-VTF: RISC-V Based Extended Instruction Set for Transformer". In: *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC). Oct. 2021, pp. 1565–1570. DOI: 10.1109/SMC52423.2021.9658643.

[177] D.A. Jimenez. "Fast path-based neural branch prediction". In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36. Dec. 2003, pp. 243–252. DOI: 10.1109/MICRO.2003.1253199.

[178] D.A. Jimenez and C. Lin. "Dynamic branch prediction with perceptrons". In: *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture. Jan. 2001, pp. 197–206. DOI: 10.1109/HPCA.2001.903263.

[179] Daniel A. Jiménez and Calvin Lin. "Neural methods for dynamic branch prediction". In: *ACM Trans. Comput. Syst.* 20.4 (Nov. 1, 2002), pp. 369–397. ISSN: 0734-2071. DOI: 10.1145/571637.571639.

[180] M. I. Jordan and T. M. Mitchell. "Machine learning: Trends, perspectives, and prospects". In: *Science* 349.6245 (July 17, 2015), pp. 255–260. DOI: 10.1126/science.aaa8415.

[181] Lana Josipović. *SAFARI Live Seminar - From C/C++ code to high-performance dataflow circuits*. Nov. 7, 2022.

[182] Lana Josipović, Philip Brisk, and Paolo Ienne. "An Out-of-Order Load-Store Queue for Spatial Computing". In: *ACM Transactions on Embedded Computing Systems* 16.5 (Sept. 27, 2017), 125:1–125:19. ISSN: 1539-9087. DOI: 10.1145/3126525.

[183] Lana Josipović, Philip Brisk, and Paolo Ienne. "From C to elastic circuits". In: *2017 51st Asilomar Conference on Signals, Systems, and Computers*. 2017 51st Asilomar Conference on Signals, Systems, and Computers. Oct. 2017, pp. 121–125. DOI: 10.1109/ACSSC.2017.8335150.

[184] 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*. FPGA '18. New York, NY, USA: Association for Computing Machinery, Feb. 15, 2018, pp. 127–136. ISBN: 978-1-4503-5614-5. DOI: 10.1145/3174243.3174264.

[185] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. "From C/C++ Code to High-Performance Dataflow Circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.7 (July 2022), pp. 2142–2155. ISSN: 1937-4151. DOI: 10.1109/TCAD.2021.3105574.

[186] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. "Speculative Dataflow Circuits". In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '19. New York, NY, USA: Association for Computing Machinery, Feb. 20, 2019, pp. 162–171. ISBN: 978-1-4503-6137-8. DOI: 10.1145/3289602.3293914.

[187] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. "Synthesizing General-Purpose Code Into Dynamically Scheduled Circuits". In: *IEEE Circuits and Systems Magazine* 21.2 (2021), pp. 97–118. ISSN: 1558-0830. DOI: 10.1109/MCAS.2021.3071631.

[188] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. "Buffer Placement and Sizing for High-Performance Dataflow Circuits". In: *ACM Transactions on Reconfigurable Technology and Systems* 15.1 (Mar. 31, 2022), pp. 1–32. ISSN: 1936-7406, 1936-7414. DOI: 10.1145/3477053.

[189] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. "Motivation for and Evaluation of the First Tensor Processing Unit". In: *IEEE Micro* 38.3 (May 2018), pp. 10–19. ISSN: 1937-4143. DOI: 10.1109/MM.2018.032271057.

[190] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. New York, NY, USA: Association for Computing Machinery, June 24, 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246.

[191] 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*. 2008 IEEE/ACM International Conference on Computer-Aided Design. Nov. 2008, pp. 434–441. DOI: 10.1109/ICCAD.2008.4681612.

[192] Daniel Kästner. "TDL: A Hardware Description Language for Retargetable Postpass Optimizations and Analyses". In: *Generative Programming and Component Engineering*. Ed. by Frank Pfenning and Yannis Smaragdakis. Berlin, Heidelberg: Springer, 2003, pp. 18–36. ISBN: 978-3-540-39815-8. DOI: 10.1007/978-3-540-39815-8_2.

[193] Cansu Kaynak, Boris Grot, and Babak Falsafi. "SHIFT: shared history instruction fetch for lean-core server processors". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. New York, NY, USA: Association for Computing Machinery, Dec. 7, 2013, pp. 272–283. ISBN: 978-1-4503-2638-4. DOI: 10.1145/2540708.2540732.

[194] Arun Kejariwal and Alexandru Nicolau. “Modulo Scheduling and Loop Pipelining”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1158–1173. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_65.

[195] T. Kim, N. Yonezawa, J.W.S. Liu, and C.L. Liu. “A scheduling algorithm for conditional resource sharing—a hierarchical reduction approach”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.4 (Apr. 1994), pp. 425–438. ISSN: 1937-4151. DOI: 10.1109/43.275353.

[196] Reimund Klemm, Javier Prieto Sabugo, H. Ahlendorf, and G. Fettweis. “Using LISATek for the Design of an ASIP Core including Floating Point Operations”. In: *MBMV*. 2007.

[197] David W. Knapp. *Behavioral synthesis: digital system design using the Synopsys behavioral compiler*. USA: Prentice-Hall, Inc., Aug. 1996. 231 pp. ISBN: 978-0-13-569252-3.

[198] Yuki Kobayashi, Yoshinori Takeuchi, and Masaharu Imai. “Chapter 7 - ASIP Meister”. In: *Processor Description Languages*. Ed. by Prabhat Mishra and Nikil Dutt. Vol. 1. Systems on Silicon. Burlington: Morgan Kaufmann, Jan. 1, 2008, pp. 163–182. DOI: 10.1016/B978-012374287-2.50010-0.

[199] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. “Spectre attacks: exploiting speculative execution”. In: *Commun. ACM* 63.7 (June 18, 2020), pp. 93–101. ISSN: 0001-0782. DOI: 10.1145/3399742.

[200] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. “Spatial: a language and compiler for application accelerators”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. New York, NY, USA: Association for Computing Machinery, June 11, 2018, pp. 296–311. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192379.

[201] Aasheesh Kolli, Ali Saidi, and Thomas F. Wenisch. “RDIP: return-address-stack directed instruction prefetching”. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. New York, NY, USA: Association for Computing Machinery, Dec. 7, 2013, pp. 260–271. ISBN: 978-1-4503-2638-4. DOI: 10.1145/2540708.2540731.

[202] A.A. Kountouris and C. Wolinski. “Combining speculative execution and conditional resource sharing to efficiently schedule conditional behaviors”. In: *Proceedings of the ASP-DAC ’99 Asia and South Pacific Design Automation Conference 1999 (Cat. No.99EX198)*. Proceedings of the ASP-DAC ’99 Asia and South Pacific Design Automation Conference 1999 (Cat. No.99EX198). Jan. 1999, 343–346 vol.1. DOI: 10.1109/ASPDAC.1999.760029.

[203] A.A. Kountouris and C. Wolinski. “Hierarchical conditional dependency graphs as a unifying design representation in the CODESIS high-level synthesis system”. In: *Proceedings 13th International Symposium on System Synthesis*. Proceedings 13th International Symposium on System Synthesis. Sept. 2000, pp. 66–71. DOI: 10.1109/ISSS.2000.874030.

[204] Apostolos A. Kountouris and Christophe Wolinski. “Efficient scheduling of conditional behaviors for high-level synthesis”. In: *ACM Trans. Des. Autom. Electron. Syst.* 7.3 (July 1, 2002), pp. 380–412. ISSN: 1084-4309. DOI: 10.1145/567270.567272.

[205] Daniel Kroening and Wolfgang J. Paul. “Automated pipeline design”. In: *Proceedings of the 38th annual Design Automation Conference*. DAC ’01. New York, NY, USA: Association for Computing Machinery, June 22, 2001, pp. 810–815. ISBN: 978-1-58113-297-7. DOI: 10.1145/378239.379071.

[206] Krzysztof Kuchcinski and Christophe Wolinski. “Global approach to assignment and scheduling of complex behaviors based on HCDG and constraint programming”. In: *Journal of Systems Architecture*. Synthesis and Verification 49.12 (Dec. 1, 2003), pp. 489–503. ISSN: 1383-7621. DOI: 10.1016/S1383-7621(03)00075-4.

[207] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. "Blasting through the Front-End Bottleneck with Shotgun". In: *SIGPLAN Not.* 53.2 (Mar. 19, 2018), pp. 30–42. ISSN: 0362-1340. DOI: 10.1145/3296957.3173178.

[208] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. "Boomerang: A Metadata-Free Architecture for Control Flow Delivery". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). Feb. 2017, pp. 493–504. DOI: 10.1109/HPCA.2017.53.

[209] Kung. "Why systolic architectures?" In: *Computer* 15.1 (Jan. 1982), pp. 37–46. ISSN: 1558-0814. DOI: 10.1109/MC.1982.1653825.

[210] S. Y. Kung and J. N. Hwang. "A unified systolic architecture for artificial neural networks". In: *Journal of Parallel and Distributed Computing* 6.2 (Apr. 1, 1989), pp. 358–387. ISSN: 0743-7315. DOI: 10.1016/0743-7315(89)90065-8.

[211] Yao-Ming Kuo, Francisco García-Herrero, Oscar Ruano, and Juan Antonio Maestro. "RISC-V Galois Field ISA Extension for Non-Binary Error-Correction Codes and Classical and Post-Quantum Cryptography". In: *IEEE Transactions on Computers* 72.3 (Mar. 2023), pp. 682–692. ISSN: 1557-9956. DOI: 10.1109/TC.2022.3174587.

[212] Alexey Kupriyanov, Frank Hannig, Dmitrij Kissler, and Jürgen Teich. "Chapter 12 - MAML: An ADL for Designing Single and Multiprocessor Architectures". In: *Processor Description Languages*. Ed. by Prabhat Mishra and Nikil Dutt. Vol. 1. Systems on Silicon. Burlington: Morgan Kaufmann, Jan. 1, 2008, pp. 295–327. DOI: 10.1016/B978-012374287-2.50015-X.

[213] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. "Programming and Synthesis for Software-defined FPGA Acceleration: Status and Future Prospects". In: *ACM Trans. Reconfigurable Technol. Syst.* 14.4 (Sept. 13, 2021), 17:1–17:39. ISSN: 1936-7406. DOI: 10.1145/3469660.

[214] G. Lakshminarayana, A. Raghunathan, and N.K. Jha. "Incorporating speculative execution into scheduling of control-flow-intensive designs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19.3 (Mar. 2000), pp. 308–324. ISSN: 1937-4151. DOI: 10.1109/43.833200.

[215] M. Lam. "Software pipelining: an effective scheduling technique for VLIW machines". In: *ACM SIGPLAN Notices* 23.7 (June 1, 1988), pp. 318–328. ISSN: 0362-1340. DOI: 10.1145/960116.54022.

[216] Vianney Lapotre, Philippe Coussy, Cyrille Chavet, Hugues Wouafo, and Robin Danilo. "Dynamic branch prediction for high-level synthesis". In: *2013 23rd International Conference on Field programmable Logic and Applications*. 2013 23rd International Conference on Field programmable Logic and Applications. Sept. 2013, pp. 1–6. DOI: 10.1109/FPL.2013.6645540.

[217] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* International Symposium on Code Generation and Optimization, 2004. CGO 2004. Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[218] 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)*. 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Feb. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[219] Jason Lau, Aishwarya Sivaraman, Qian Zhang, Muhammad Ali Gulzar, Jason Cong, and Miryung Kim. "HeteroRefactor: refactoring for heterogeneous computing with FPGA". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. New York, NY, USA: Association for Computing Machinery, Oct. 1, 2020, pp. 493–505. ISBN: 978-1-4503-7121-6. DOI: 10.1145/3377811.3380340.

[220] Chi-Hui Lee, Che-Hua Shih, Juinn-Dar Huang, and Jing-Yang Jou. "Equivalence checking of scheduling with speculative code transformations in high-level synthesis". In: *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*. 16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011). Jan. 2011, pp. 497–502. DOI: 10.1109/ASPDAC.2011.5722241.

[221] Dajung Lee, Andrei Hagiescu, and Dan Pritsker. "Large-Scale and High-Throughput QR Decomposition on an FPGA". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). Apr. 2019, pp. 337–337. DOI: 10.1109/FCCM.2019.00078.

[222] Seungah Lee, Emmanuel Casseau, Angeliki Kritikakou, Olivier Sentieys, Ruben Salvador, and Julien Galizzi. "On-board Payload Data Processing Combined with the Roofline Model for Hardware/Software Design". In: *2024 IEEE Aerospace Conference*. 2024 IEEE Aerospace Conference. Mar. 2024, pp. 1–12. DOI: 10.1109/AERO58975.2024.10521057.

[223] Charles E. Leiserson, Flavio M. Rose, and James B. Saxe. "Optimizing Synchronous Circuitry by Retiming (Preliminary Version)". In: *Third Caltech Conference on Very Large Scale Integration*. Ed. by Randal Bryant. Berlin, Heidelberg: Springer, 1983, pp. 87–116. ISBN: 978-3-642-95432-0. DOI: 10.1007/978-3-642-95432-0_7.

[224] Charles E. Leiserson and James B. Saxe. "Retiming synchronous circuitry". In: *Algorithmica* 6.1 (June 1, 1991), pp. 5–35. ISSN: 1432-0541. DOI: 10.1007/BF01759032.

[225] Dylan Leothaud, Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. "Efficient Design Space Exploration for Dynamic & Speculative High-Level Synthesis". In: *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*. 2024 34th International Conference on Field-Programmable Logic and Applications (FPL). Sept. 2024, pp. 109–117. DOI: 10.1109/FPL64840.2024.00024.

[226] Harry J. Levinson. *Principles of Lithography*. SPIE Press, 2005. 446 pp. ISBN: 978-0-8194-5660-1.

[227] Daniel Lewin, Dean Lorenz, and Shmuel Ur. "A methodology for processor implementation verification". In: *Formal Methods in Computer-Aided Design*. Ed. by Mandayam Srivas and Albert Camilleri. Berlin, Heidelberg: Springer, 1996, pp. 126–142. ISBN: 978-3-540-49567-3. DOI: 10.1007/BFb0031804.

[228] Jiemin Li, Shancong Zhang, and Chong Bao. "DuckCore: A Fault-Tolerant Processor Core Architecture Based on the RISC-V ISA". In: *Electronics* 11.1 (Jan. 2022), p. 122. ISSN: 2079-9292. DOI: 10.3390/electronics11010122.

[229] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. *Meltdown*. Jan. 3, 2018. DOI: 10.48550/arXiv.1801.01207. arXiv: 1801.01207[cs].

[230] Gai Liu, Joseph Primmer, and Zhiru Zhang. "Rapid Generation of High-Quality RISC-V Processors from Functional Instruction Set Specifications". In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 2019 56th ACM/IEEE Design Automation Conference (DAC). June 2019, pp. 1–6.

[231] Jiantao Liu, Carmine Rizzi, and Lana Josipović. "Load-Store Queue Sizing for Efficient Dataflow Circuits". In: *2022 International Conference on Field-Programmable Technology (ICFPT)*. 2022 International Conference on Field-Programmable Technology (ICFPT). Dec. 2022, pp. 1–9. DOI: 10.1109/ICFPT56656.2022.9974425.

[232] Junyi Liu, John Wickerson, Samuel Bayliss, and George A. Constantinides. "Polyhedral-Based Dynamic Loop Pipelining for High-Level Synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.9 (Sept. 2018), pp. 1802–1815. ISSN: 1937-4151. DOI: 10.1109/TCAD.2017.2783363.

[233] Junyi Liu, John Wickerson, Samuel Bayliss, and George A. Constantinides. "Run fast when you can: Loop pipelining with uncertain and non-uniform memory dependencies". In: *2017 51st Asilomar Conference on Signals, Systems, and Computers*. 2017 51st Asilomar Conference on Signals, Systems, and Computers. Oct. 2017, pp. 126–130. DOI: 10.1109/ACSSC.2017.8335151.

[234] Xinheng Liu, Yao Chen, Tan Nguyen, Swathi Gurumani, Kyle Rupnow, and Deming Chen. "High Level Synthesis of Complex Applications: An H.264 Video Decoder". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. New York, NY, USA: Association for Computing Machinery, Feb. 21, 2016, pp. 224–233. ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847274.

[235] Michael Lo, Zhenman Fang, Jie Wang, Peipei Zhou, Mau-Chung Frank Chang, and Jason Cong. "Algorithm-Hardware Co-design for BQSR Acceleration in Genome Analysis ToolKit". In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). May 2020, pp. 157–166. DOI: 10.1109/FCCM48280.2020.00029.

[236] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. "A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography". In: *ACM Comput. Surv.* 54.6 (July 13, 2021), 122:1–122:37. ISSN: 0360-0300. DOI: 10.1145/3456629.

[237] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. *The gem5 Simulator: Version 20.0+*. Sept. 29, 2020. DOI: 10.48550/arXiv.2007.03152. arXiv: 2007.03152[cs].

[238] Yangdi Lyu and Prabhat Mishra. "A Survey of Side-Channel Attacks on Caches and Countermeasures". In: *Journal of Hardware and Systems Security* 2.1 (Mar. 1, 2018), pp. 33–50. ISSN: 2509-3436. DOI: 10.1007/s41635-017-0025-y.

[239] D. MacMillen, R. Camposano, D. Hill, and T.W. Williams. "An industrial view of electronic design automation". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19.12 (Dec. 2000), pp. 1428–1448. ISSN: 1937-4151. DOI: 10.1109/43.898825.

[240] Scott A. Mahlke, William Y. Chen, Roger A. Bringmann, Richard E. Hank, Wen-Mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. "Sentinel scheduling: a model for compiler-controlled speculative execution". In: *ACM Trans. Comput. Syst.* 11.4 (Nov. 1, 1993), pp. 376–408. ISSN: 0734-2071. DOI: 10.1145/161541.159765.

[241] Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. "Sentinel scheduling for VLIW and superscalar processors". In: *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*. ASPLOS V. New York, NY, USA: Association for Computing Machinery, Sept. 1, 1992, pp. 238–247. ISBN: 978-0-89791-534-2. DOI: 10.1145/143365.143529.

[242] Kingshuk Majumder and Uday Bondhugula. "HIR: An MLIR-based Intermediate Representation for Hardware Accelerator Description". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. ASPLOS '23. New York, NY, USA: Association for Computing Machinery, Feb. 7, 2024, pp. 189–201. ISBN: 9798400703942. DOI: 10.1145/3623278.3624767.

[243] Gowthami Jayashri Manikandan, Sitao Huang, Kyle Rupnow, Wen-Mei W. Hwu, and Deming Chen. "Acceleration of the Pair-HMM Algorithm for DNA Variant Calling". In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). May 2016, pp. 137–137. DOI: 10.1109/FCCM.2016.42.

[244] M. Morris Mano and Charles Kime. *Logic and Computer Design Fundamentals*. 4th. USA: Prentice Hall Press, May 2007. 696 pp. ISBN: 978-0-13-198926-9.

[245] 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)*. 2020 IEEE Custom Integrated Circuits Conference (CICC). Mar. 2020, pp. 1–8. DOI: 10.1109/CICC48029.2020.9075913.

[246] Charalampos Marantos, Konstantinos Salapas, Lazaros Papadopoulos, and Dimitrios Soudris. "A Flexible Tool for Estimating Applications Performance and Energy Consumption Through Static Analysis". In: *SN Computer Science* 2.1 (Jan. 7, 2021), p. 21. ISSN: 2661-8907. DOI: 10.1007/s42979-020-00405-7.

[247] Peter Marwedel. "Chapter 3 - Mimola—A Fully Synthesizable Language". In: *Processor Description Languages*. Ed. by Prabhat Mishra and Nikil Dutt. Vol. 1. Systems on Silicon. Burlington: Morgan Kaufmann, Jan. 1, 2008, pp. 35–63. DOI: 10.1016/B978-012374287-2.50006-9.

[248] Daniel S. McFarlin, Charles Tucker, and Craig Zilles. "Discerning the dominant out-of-order performance advantage: is it speculation or dynamism?" In: *SIGARCH Comput. Archit. News* 41.1 (Mar. 16, 2013), pp. 241–252. ISSN: 0163-5964. DOI: 10.1145/2490301.2451143.

[249] Scott McFarling. *Combining branch predictors*. Tech. rep. Digital Equipment Corporation, 1993.

[250] Ashok B. Mehta. *ASIC/SoC Functional Design Verification*. Cham: Springer International Publishing, 2018. ISBN: 978-3-319-59417-0 978-3-319-59418-7. DOI: 10.1007/978-3-319-59418-7.

[251] Yuan Meng, Hongjiang Men, and Viktor Prasanna. "Accelerating Deformable Convolution Networks". In: *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). May 2022, pp. 1–1. DOI: 10.1109/FCCM53951.2022.9786173.

[252] Arjun Menon, Subadra Murugan, Chester Rebeiro, Neel Gala, and Kamakoti Veezhinathan. "Shakti-T: A RISC-V Processor with Light Weight Security Extensions". In: *Proceedings of the Hardware and Architectural Support for Security and Privacy*. HASP '17. New York, NY, USA: Association for Computing Machinery, June 25, 2017, pp. 1–8. ISBN: 978-1-4503-5266-6. DOI: 10.1145/3092627.3092629.

[253] Michael Meredith. "High-Level SystemC Synthesis with Forte's Cynthesizer". In: *High-Level Synthesis: From Algorithm to Digital Circuit*. Ed. by Philippe Coussy and Adam Morawiec. Dordrecht: Springer Netherlands, 2008, pp. 75–97. ISBN: 978-1-4020-8588-8. DOI: 10.1007/978-1-4020-8588-8_5.

[254] Pierre Michaud. "An Alternative TAGE-like Conditional Branch Predictor". In: *ACM Trans. Archit. Code Optim.* 15.3 (Aug. 28, 2018), 30:1–30:23. ISSN: 1544-3566. DOI: 10.1145/3226098.

[255] A Miczo. *Digital logic testing and simulation*. USA: Harper & Row Publishers, Inc., Nov. 1985. 414 pp. ISBN: 978-0-06-044444-0.

[256] D. Milicev and Z. Jovanovic. "Predicated software pipelining technique for loops with conditions". In: *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing. Mar. 1998, pp. 176–180. DOI: 10.1109/IPPS.1998.669907.

[257] Robin Milner. *The Definition of Standard ML: Revised*. MIT Press, 1997. 132 pp. ISBN: 978-0-262-63181-5.

[258] Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo, and Fabrizio Ferrandi. "Interprocedural resource sharing in High Level Synthesis through function proxies". In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 2015 25th International Conference on Field Programmable Logic and Applications (FPL). Sept. 2015, pp. 1–8. DOI: 10.1109/FPL.2015.7293958.

[259] Prabhat Mishra and Nikil Dutt. "Chapter 1 - Introduction to Architecture Description Languages". In: *Processor Description Languages*. Ed. by Prabhat Mishra and Nikil Dutt. Vol. 1. Systems on Silicon. Burlington: Morgan Kaufmann, Jan. 1, 2008, pp. 1–12. DOI: 10.1016/B978-012374287-2.50004-5.

[260] Prabhat Mishra and Nikil Dutt. "Chapter 6 - EXPRESSION: An ADL for Software Toolkit Generation, Exploration, and Validation of Programmable SOC Architectures". In: *Processor Description Languages*. Ed. by Prabhat Mishra and Nikil Dutt. Vol. 1. Systems on Silicon. Burlington: Morgan Kaufmann, Jan. 1, 2008, pp. 133–161. DOI: 10.1016/B978-012374287-2.50009-4.

[261] N. Moreano, E. Borin, Cid de Souza, and G. Araujo. "Efficient datapath merging for partially reconfigurable architectures". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.7 (July 2005), pp. 969–980. ISSN: 1937-4151. DOI: 10.1109/TCAD.2005.850844.

[262] Nahri Moreano, Guido Araujo, Zhining Huang, and Sharad Malik. "Datapath merging and interconnection sharing for reconfigurable architectures". In: *Proceedings of the 15th international symposium on System Synthesis*. ISSS '02. New York, NY, USA: Association for Computing Machinery, Oct. 2, 2002, pp. 38–43. ISBN: 978-1-58113-576-3. DOI: 10.1145/581199.581210.

[263] T Morimoto. "Superscalar processor design with hardware description language AIDL". In: *Proc. of APCHDL'94* (1994).

[264] Antoine Morvan, Steven Derrien, and Patrice Quinton. "Polyhedral Bubble Insertion: A Method to Improve Nested Loop Pipelining for High-Level Synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.3 (Mar. 2013), pp. 339–352. ISSN: 1937-4151. DOI: 10.1109/TCAD.2012.2228270.

[265] Yannick Le Moullec, Jean-Philippe Diguet, Thierry Gourdeaux, and Jean-Luc Philippe. "DesignTrotter: System-level dynamic estimation task a first step towards platform architecture selection". In: *Journal of Embedded Computing* 1.4 (Jan. 1, 2005), pp. 565–586. ISSN: 1740-4460.

[266] T. Murata. "Petri nets: Properties, analysis and applications". In: *Proceedings of the IEEE* 77.4 (Apr. 1989), pp. 541–580. ISSN: 1558-2256. DOI: 10.1109/5.24143.

[267] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. "A Survey and Evaluation of FPGA High-Level Synthesis Tools". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (Oct. 2016), pp. 1591–1604. ISSN: 1937-4151. DOI: 10.1109/TCAD.2015.2513673.

[268] Pietro Nannipieri, Stefano Di Matteo, Luca Zulberti, Francesco Albicocchi, Sergio Saponara, and Luca Fanucci. "A RISC-V Post Quantum Cryptography Instruction Set Extension for Number Theoretic Transform to Speed-Up CRYSTALS Algorithms". In: *IEEE Access* 9 (2021), pp. 150798–150808. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3126208.

[269] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. "Predictable accelerator design with time-sensitive affine types". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, June 11, 2020, pp. 393–407. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385974.

[270] R. Nikhil. "Bluespec System Verilog: efficient, correct RTL from high level specifications". In: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. June 2004, pp. 69–70. DOI: 10.1109/MEMCOD.2004.1459818.

[271] Mostafa W. Numan, Braden J. Phillips, Gavin S. Puddy, and Katrina Falkner. "Towards Automatic High-Level Code Deployment on Reconfigurable Platforms: A Survey of High-Level Synthesis Tools and Toolchains". In: *IEEE Access* 8 (2020), pp. 174692–174722. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3024098.

[272] Eriko Nurvitadhi, James C. Hoe, Timothy Kam, and Shih-Lien L. Lu. "Automatic Pipelining From Transactional Datapath Specifications". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.3 (Mar. 2011), pp. 441–454. ISSN: 1937-4151. DOI: 10.1109/TCAD.2010.2088950.

[273] Enno Ohlebusch. *Advanced Topics in Term Rewriting*. Springer Science & Business Media, Apr. 17, 2013. 420 pp. ISBN: 978-1-4757-3661-8.

[274] Soner Önder. "Chapter 10 - ADL++: Object-Oriented Specification of Complicated Instruction Sets and Microarchitectures". In: *Processor Description Languages*. Ed. by Prabhat Mishra and Nikil Dutt. Vol. 1. Systems on Silicon. Burlington: Morgan Kaufmann, Jan. 1, 2008, pp. 247–273. DOI: 10.1016/B978-012374287-2.50013-6.

[275] Julian Oppermann, Brindusa Mihaela Damian-Kosterhon, Florian Meisel, Tammo Mürmann, Eyck Jentzsch, and Andreas Koch. "Longnail: High-Level Synthesis of Portable Custom Instruction Set Extensions for RISC-V Processors from Descriptions in the Open-Source CoreDSL Language". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. Vol. 3. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, Apr. 27, 2024, pp. 591–606. ISBN: 9798400703867. DOI: 10.1145/3620666.3651375.

[276] Julian Oppermann, Andreas Koch, Melanie Reuter-Oppermann, and Oliver Sinnen. "ILP-based modulo scheduling for high-level synthesis". In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '16. New York, NY, USA: Association for Computing Machinery, Oct. 1, 2016, pp. 1–10. ISBN: 978-1-4503-4482-1. DOI: 10.1145/2968455.2968512.

[277] Julian Oppermann, Melanie Reuter-Oppermann, Lukas Sommer, Andreas Koch, and Oliver Sinnen. "Exact and Practical Modulo Scheduling for High-Level Synthesis". In: *ACM Trans. Reconfigurable Technol. Syst.* 12.2 (May 6, 2019), 8:1–8:26. ISSN: 1936-7406. DOI: 10.1145/3317670.

[278] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. "The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages". In: *ACM SIGPLAN Notices* 25.6 (June 1, 1990), pp. 257–271. ISSN: 0362-1340. DOI: 10.1145/93548.93578.

[279] G. Ottoni, R. Rangan, A. Stoler, and D.I. August. "Automatic thread extraction with decoupled software pipelining". In: *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05). Nov. 2005, 12 pp.–118. DOI: 10.1109/MICRO.2005.13.

[280] Ian Page and Wayne Luk. "Compiling Occam into field-programmable gate arrays". In: *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*. Vol. 15. Abingdon EE&CS Books 15 Harcourt Way, Abingdon OX14 1NV, UK. 1991, pp. 271–283.

[281] Preeti Ranjan Panda. "SystemC: a modeling platform supporting multiple design abstractions". In: *Proceedings of the 14th international symposium on Systems synthesis*. ISSS '01. New York, NY, USA: Association for Computing Machinery, Sept. 30, 2001, pp. 75–80. ISBN: 978-1-58113-418-6. DOI: 10.1145/500001.500018.

[282] Joseph CH Park and Mike Schlansker. *On predicated execution*. Hewlett-Packard Laboratories Palo Alto, California, 1991.

[283] David Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, Aug. 2017. 200 pp. ISBN: 978-0-9992491-0-9.

[284] David A. Patterson and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Mar. 2017. 696 pp. ISBN: 978-0-12-812275-4.

[285] P.G. Paulin and J.P. Knight. "Force-directed scheduling for the behavioral synthesis of ASICs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 8.6 (June 1989), pp. 661–679. ISSN: 1937-4151. DOI: 10.1109/43.31522.

[286] Pierre G. Paulin, Clifford Liem, Trevor C. May, and Shailesh Sutarwala. "FlexWare : A Flexible Firmware Development Environment for Embedded Systems". In: *Code Generation for Embedded Processors*. Ed. by Peter Marwedel and Gert Goossens. Boston, MA: Springer US, 2002, pp. 67–84. ISBN: 978-1-4615-2323-9. DOI: 10.1007/978-1-4615-2323-9_4.

[287] Biagio Peccerillo, Mirco Mannino, Andrea Mondelli, and Sandro Bartolini. "A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives". In: *Journal of Systems Architecture* 129 (Aug. 1, 2022), p. 102561. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2022.102561.

[288] Blake Pelton, Adam Sapek, Ken Eguro, Daniel Lo, Alessandro Forin, Matt Humphrey, Jinwen Xi, David Cox, Rajas Karandikar, Johannes de Fine Licht, Evgeny Babin, Adrian Caulfield, and Doug Burger. "Wavefront Threading Enables Effective High-Level Synthesis". In: *Proc. ACM Program. Lang.* 8 (PLDI June 20, 2024), 190:1066–190:1090. DOI: 10.1145/3656420.

[289] Arthur Perais. "Increasing the performance of superscalar processors through value prediction". PhD thesis. Université de Rennes, Sept. 24, 2015.

[290] Arthur Perais and André Seznec. "EOLE: Toward a Practical Implementation of Value Prediction". In: *IEEE Micro* 35.3 (May 2015), pp. 114–124. ISSN: 1937-4143. DOI: 10.1109/MM.2015.45.

[291] Arthur Perais and André Seznec. "Practical data value speculation for future high-end processors". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). Feb. 2014, pp. 428–439. DOI: 10.1109/HPCA.2014.6835952.

[292] Arthur Perais, Rami Sheikh, Luke Yen, Michael McIlvaine, and Robert D. Clancy. "Elastic Instruction Fetching". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). Feb. 2019, pp. 478–490. DOI: 10.1109/HPCA.2019.00059.

[293] Morten Borup Petersen. "A Dynamically Scheduled HLS Flow in MLIR". 2022. 92 pp.

[294] J. Pierce and T. Mudge. "Wrong-path instruction prefetching". In: *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29. Dec. 1996, pp. 165–175. DOI: 10.1109/MICRO.1996.566459.

[295] Oron Port and Yoav Etsion. "DFiant: A dataflow hardware description language". In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 2017 27th International Conference on Field Programmable Logic and Applications (FPL). Sept. 2017, pp. 1–4. DOI: 10.23919/FPL.2017.8056858.

[296] U. Prabhu and B.M. Pangrle. "Global mobility based scheduling". In: *Proceedings of 1993 IEEE International Conference on Computer Design ICCD'93*. Proceedings of 1993 IEEE International Conference on Computer Design ICCD'93. Oct. 1993, pp. 370–373. DOI: 10.1109/ICCD.1993.393350.

[297] Léo Pradels, Silviu-Ioan Filip, Olivier Sentieys, Daniel Chillet, and Thibaut Le Calloch. "FPGA-based CNN Acceleration using Pattern-Aware Pruning". In: *2024 IEEE 6th International Conference on AI Circuits and Systems (AICAS)*. 2024 IEEE 6th International Conference on AI Circuits and Systems (AICAS). Apr. 2024, pp. 228–232. DOI: 10.1109/AICAS59952.2024.10595865.

[298] R. Prasad. *Analog and Digital Electronic Circuits: Fundamentals, Analysis, and Applications*. Undergraduate Lecture Notes in Physics. Cham: Springer International Publishing, 2021. ISBN: 978-3-030-65128-2 978-3-030-65129-9. DOI: 10.1007/978-3-030-65129-9.

[299] William Pugh. "The Omega test: a fast and practical integer programming algorithm for dependence analysis". In: *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. Supercomputing '91. New York, NY, USA: Association for Computing Machinery, Aug. 1, 1991, pp. 4–13. ISBN: 978-0-89791-459-8. DOI: 10.1145/125826.125848.

[300] Wei Qin and Sharad Malik. "A Study of Architecture Description Languages from a Model-based Perspective". In: *2005 Sixth International Workshop on Microprocessor Test and Verification*. 2005 Sixth International Workshop on Microprocessor Test and Verification. Nov. 2005, pp. 3–11. DOI: 10.1109/MTV.2005.2.

[301] Wei Qin and Sharad Malik. "Architecture Description Languages for Retargetable Compilation". In: *The Compiler Design Handbook*. Ed. by Y Srikant and P Shankar. CRC Press, Sept. 25, 2002. ISBN: 978-0-8493-1240-3 978-1-4200-4057-9. DOI: 10.1201/9781420040579.ch14.

[302] Wei Qin, Subramanian Rajagopalan, and Sharad Malik. "MADL—An ADL Based on a Formal and Flexible Concurrency Model". In: *Processor Description Languages*. Ed. by Prabhat Mishra and Nikil Dutt. Vol. 1. Systems on Silicon. Burlington: Morgan Kaufmann, Jan. 1, 2008, pp. 217–245. DOI: 10.1016/B978-012374287-2.50012-4.

[303] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. "Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices". In: *SIGPLAN Not.* 40.6 (June 12, 2005), pp. 269–279. ISSN: 0362-1340. DOI: 10.1145/1064978.1065043.

[304] Patrice Quinton. "Automatic synthesis of systolic arrays from uniform recurrent equations". In: *SIGARCH Comput. Archit. News* 12.3 (Jan. 1, 1984), pp. 208–214. ISSN: 0163-5964. DOI: 10.1145/773453.808184.

[305] Patrice Quinton. "The systematic design of systolic arrays". In: *Centre National de Recherche Scientifique on Automata networks in computer science: theory and applications*. USA: Princeton University Press, Oct. 1, 1987, pp. 229–260. ISBN: 978-0-691-08479-4.

[306] I. Radivojevic and F. Brewer. "A new symbolic technique for control-dependent scheduling". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 15.1 (Jan. 1996), pp. 45–57. ISSN: 1937-4151. DOI: 10.1109/43.486271.

[307] Amir M. Rahmani, Pasi Liljeberg, Ahmed Hemani, Axel Jantsch, and Hannu Tenhunen, eds. *The Dark Side of Silicon*. Cham: Springer International Publishing, 2017. ISBN: 978-3-319-31594-2 978-3-319-31596-6. DOI: 10.1007/978-3-319-31596-6.

[308] G. Ramalingam. "The undecidability of aliasing". In: *ACM Trans. Program. Lang. Syst.* 16.5 (Sept. 1, 1994), pp. 1467–1471. ISSN: 0164-0925. DOI: 10.1145/186025.186041.

[309] A. Ramirez, O.J. Santana, J.L. Larriba-Pey, and M. Valero. "Fetching instruction streams". In: *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings*. 35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings. Nov. 2002, pp. 371–382. DOI: 10.1109/MICRO.2002.1176264.

[310] Norman Ramsey and Mary F. Fernández. "Specifying representations of machine instructions". In: *ACM Trans. Program. Lang. Syst.* 19.3 (May 1, 1997), pp. 492–524. ISSN: 0164-0925. DOI: 10.1145/256167.256225.

[311] R. Rangan, N. Vachharajani, M. Vachharajani, and D.I. August. "Decoupled software pipelining with the synchronization array". In: *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Oct. 2004, pp. 177–188. DOI: 10.1109/PACT.2004.1342552.

[312] B. R. Rau and C. D. Glaeser. "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing". In: *SIGMICRO Newsl.* 12.4 (Dec. 1, 1981), pp. 183–198. ISSN: 1050-916X. DOI: 10.1145/1014192.802449.

[313] B. Ramakrishna Rau. "Iterative modulo scheduling: an algorithm for software pipelining loops". In: *Proceedings of the 27th annual international symposium on Microarchitecture.* MICRO 27. New York, NY, USA: Association for Computing Machinery, Nov. 30, 1994, pp. 63–74. ISBN: 978-0-89791-707-0. DOI: 10.1145/192724.192731.

[314] B. Ramakrishna Rau and Joseph A. Fisher. "Instruction-level parallel processing: History, overview, and perspective". In: *The Journal of Supercomputing* 7.1 (May 1, 1993), pp. 9–50. ISSN: 1573-0484. DOI: 10.1007/BF01205181.

[315] G. Reinman, B. Calder, and T. Austin. "Fetch directed instruction prefetching". In: *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture.* MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture. Nov. 1999, pp. 16–27. DOI: 10.1109/MICRO.1999.809439.

[316] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. "Survey and Benchmarking of Machine Learning Accelerators". In: *2019 IEEE High Performance Extreme Computing Conference (HPEC).* 2019 IEEE High Performance Extreme Computing Conference (HPEC). Sept. 2019, pp. 1–9. DOI: 10.1109/HPEC.2019.8916327.

[317] Hugo Reymond, Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, Isabelle Puaut, and Erven Rohou. "SCHEMATIC: Compile-Time Checkpoint Placement and Memory Allocation for Intermittent Systems". In: *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Mar. 2024, pp. 258–269. DOI: 10.1109/CGO57630.2024.10444789.

[318] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. "ArchC: a SystemC-based architecture description language". In: *16th Symposium on Computer Architecture and High Performance Computing.* 16th Symposium on Computer Architecture and High Performance Computing. Oct. 2004, pp. 66–73. DOI: 10.1109/SBAC-PAD.2004.8.

[319] Simon Rokicki, Davide Pala, Joseph Paturel, and Olivier Sentieys. "What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications". In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). Nov. 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942177.

[320] Simon Rokicki, Erven Rohou, and Steven Derrien. "Hybrid-DBT: Hardware/Software Dynamic Binary Translation Targeting VLIW". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.10 (Oct. 2019), pp. 1872–1885. ISSN: 1937-4151. DOI: 10.1109/TCAD.2018.2864288.

[321] Alberto Ros and Alexandra Jimborean. "A Cost-Effective Entangling Prefetcher for Instructions". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).* 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). June 2021, pp. 99–111. DOI: 10.1109/ISCA52012.2021.00017.

[322] Alberto Ros and Alexandra Jimborean. "Wrong-Path-Aware Entangling Instruction Prefetcher". In: *IEEE Transactions on Computers* 73.2 (Feb. 2024), pp. 548–559. ISSN: 1557-9956. DOI: 10.1109/TC.2023.3337308.

[323] Davide Rossi, Igor Loi, Francesco Conti, Giuseppe Tagliavini, Antonio Pullini, and Andrea Marongiu. "Energy efficient parallel computing on the PULP platform with support for OpenMP". In: *2014 IEEE 28th Convention of Electrical & Electronics Engineers in Israel (IEEEI)*. 2014 IEEE 28th Convention of Electrical & Electronics Engineers in Israel (IEEEI). Dec. 2014, pp. 1–5. DOI: 10.1109/EEEI.2014.7005803.

[324] John Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. "Software pipelining showdown: optimal vs. heuristic methods in a production compiler". In: *SIGPLAN Not.* 31.5 (May 1, 1996), pp. 1–11. ISSN: 0362-1340. DOI: 10.1145/249069.231385.

[325] Sepideh Safari, Mohsen Ansari, Heba Khdr, Pourya Gohari-Nazari, Sina Yari-Karin, Amir Yeganeh-Khaksar, Shaahin Hessabi, Alireza Ejlali, and Jörg Henkel. "A Survey of Fault-Tolerance Techniques for Embedded Systems From the Perspective of Power, Energy, and Thermal Issues". In: *IEEE Access* 10 (2022), pp. 12229–12251. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3144217.

[326] Mohammad Samragh, Mohammad Ghasemzadeh, and Farinaz Koushanfar. "Customizing Neural Networks for Efficient FPGA Implementation". In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). Apr. 2017, pp. 85–92. DOI: 10.1109/FCCM.2017.43.

[327] Himanshu A. Sanghavi and Nupur B. Andrews. "Chapter 8 - TIE: An ADL for Designing Application-specific Instruction Set Extensions". In: *Processor Description Languages*. Ed. by Prabhat Mishra and Nikil Dutt. Vol. 1. Systems on Silicon. Burlington: Morgan Kaufmann, Jan. 1, 2008, pp. 183–216. DOI: 10.1016/B978-012374287-2.50011-2.

[328] Kaz Sato and Cliff Young. *An in-depth look at Google's first Tensor Processing Unit (TPU)*. Google Cloud Blog. May 12, 2017. URL: https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu (visited on 07/22/2024).

[329] Shimpei Sato and Kenji Kise. "ArchHDL: A Novel Hardware RTL Design Environment in C++". In: *Applied Reconfigurable Computing*. Ed. by Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz. Cham: Springer International Publishing, 2015, pp. 53–64. ISBN: 978-3-319-16214-0. DOI: 10.1007/978-3-319-16214-0_5.

[330] Jonathan Saussereau, Christophe Jego, Camille Leroux, and Jean-Baptiste Begueret. "Design and Implementation of a RISC-V core with a Flexible Pipeline for Design Space Exploration". In: *2023 30th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2023 30th IEEE International Conference on Electronics, Circuits and Systems (ICECS). Dec. 2023, pp. 1–5. DOI: 10.1109/ICECS58634.2023.10382774.

[331] Jonathan Saussereau, Camille Leroux, Jean-Baptiste Begueret, and Christophe Jego. "AsteRISC: A Size-Optimized RISC-V Core for Design Space Exploration". In: *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2023 IEEE International Symposium on Circuits and Systems (ISCAS). May 2023, pp. 1–5. DOI: 10.1109/ISCAS46773.2023.10181330.

[332] Martin Schoeberl. *Digital Design with Chisel*. Kindle Direct Publishing, 2019.

[333] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. "LLHD: a multi-level intermediate representation for hardware description languages". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, June 11, 2020, pp. 258–271. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3386024.

[334] André Seznec and Pierre Michaud. "A case for (partially) tagged geometric history length branch prediction". In: *The Journal of Instruction-Level Parallelism* 8 (2006), p. 23.

[335] O Shacham, O Azizi, M Wachs, Wajahat Qadeer, Zain Asgar, Kyle Kelley, John P Stevenson, Stephen Richardson, Mark Horowitz, Benjamin Lee, Alex Solomatnikov, and Amin Firoozshahian. "Rethinking Digital Design: Why Design Must Change". In: *IEEE Micro* 30.6 (Nov. 2010), pp. 9–24. ISSN: 1937-4143. DOI: 10.1109/MM.2010.81.

[336] Ofer Shacham, Megan Wachs, Andrew Danowitz, Sameh Galal, John Brunhaver, Wajahat Qadeer, Sabarish Sankaranarayanan, Artem Vassiliev, Stephen Richardson, and Mark Horowitz. "Avoiding game over: bringing design to the next level". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. New York, NY, USA: Association for Computing Machinery, June 3, 2012, pp. 623–629. ISBN: 978-1-4503-1199-1. DOI: 10.1145/2228360.2228472.

[337] Miho Shimizu, Nagisa Ishiura, Sayuri Ota, and Wakako Nakano. "Speculative execution in distributed controllers for high-level synthesis". In: *Proceedings of the 28th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype*. RSP '17. New York, NY, USA: Association for Computing Machinery, Oct. 19, 2017, pp. 99–104. ISBN: 978-1-4503-5418-9. DOI: 10.1145/3130265.3130319.

[338] R. O. Simpson and P. D. Hester. "The IBM RT PC ROMP processor and memory management unit architecture". In: *IBM Systems Journal* 26.4 (1987), pp. 346–360. ISSN: 0018-8670. DOI: 10.1147/sj.264.0346.

[339] C. Siska. "A processor description language supporting retargetable multi-pipeline DSP program development tools". In: *Proceedings. 11th International Symposium on System Synthesis (Cat. No.98EX210)*. Proceedings. 11th International Symposium on System Synthesis (Cat. No.98EX210). Dec. 1998, pp. 31–36. DOI: 10.1109/ISSS.1998.730593.

[340] Patrick Sittel, Nicolai Fiege, John Wickerson, and Peter Zipf. "Optimal and Heuristic Approaches to Modulo Scheduling With Rational Initiation Intervals in Hardware Synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.3 (Mar. 2022), pp. 614–627. ISSN: 1937-4151. DOI: 10.1109/TCAD.2021.3060320.

[341] Sam Skalicky, Tejaswini Ananthanarayana, Sonia Lopez, and Marcin Lukowiak. "Designing customized ISA processors using high level synthesis". In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig). Dec. 2015, pp. 1–6. DOI: 10.1109/ReConFig.2015.7393299.

[342] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. "Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication". In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '22. New York, NY, USA: Association for Computing Machinery, Feb. 11, 2022, pp. 65–77. ISBN: 978-1-4503-9149-8. DOI: 10.1145/3490422.3502357.

[343] Cid C. de Souza, Andre M. Lima, Guido Araujo, and Nahri B. Moreano. "The datapath merging problem in reconfigurable systems: Complexity, dual bounds and heuristic evaluation". In: *ACM J. Exp. Algorithmics* 10 (Dec. 31, 2005), 2.2–es. ISSN: 1084-6654. DOI: 10.1145/1064546.1180613.

[344] Leandro de Souza Rosa, Christos-Savvas Bouganis, and Vanderlei Bonato. "Scaling Up Modulo Scheduling for High-Level Synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.5 (May 2019), pp. 912–925. ISSN: 1937-4151. DOI: 10.1109/TCAD.2018.2834440.

[345] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. "Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices". In: *IEEE Communications Surveys & Tutorials* 20.1 (2018), pp. 465–488. ISSN: 1553-877X. DOI: 10.1109/COMST.2017.2779824.

[346] V. Srinivasan, E.S. Davidson, G.S. Tyson, M.J. Charney, and T.R. Puzak. "Branch history guided instruction prefetching". In: *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture. Jan. 2001, pp. 291–300. DOI: 10.1109/HPCA.2001.903271.

[347] François-Xavier Standaert. "Introduction to Side-Channel Attacks". In: *Secure Integrated Circuits and Systems*. Ed. by Ingrid M.R. Verbauwhede. Boston, MA: Springer US, 2010, pp. 27–42. ISBN: 978-0-387-71829-3. DOI: 10.1007/978-0-387-71829-3_2.

[348] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. New York, NY, USA: Association for Computing Machinery, Feb. 21, 2016, pp. 16–25. ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847276.

[349] Mengshu Sun, Zhengang Li, Alec Lu, Yanyu Li, Sung-En Chang, Xiaolong Ma, Xue Lin, and Zhenman Fang. "FILM-QNN: Efficient FPGA Acceleration of Deep Neural Networks with Intra-Layer, Mixed-Precision Quantization". In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '22. New York, NY, USA: Association for Computing Machinery, Feb. 11, 2022, pp. 134–145. ISBN: 978-1-4503-9149-8. DOI: 10.1145/3490422.3502364.

[350] Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. "Compiler Discovered Dynamic Scheduling of Irregular Code in High-Level Synthesis". In: 2023 33rd International Conference on Field-Programmable Logic and Applications (FPL). IEEE Computer Society, Sept. 1, 2023, pp. 1–9. ISBN: 9798350341515. DOI: 10.1109/FPL60245.2023.00009.

[351] Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. "Dynamically Scheduled Memory Operations in Static High-Level Synthesis". In: *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). May 2023, pp. 220–220. DOI: 10.1109/FCCM57271.2023.00048.

[352] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. "Design and Evaluation of SmallFloat SIMD extensions to the RISC-V ISA". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). Mar. 2019, pp. 654–657. DOI: 10.23919/DATE.2019.8714897.

[353] Toyokazu Takagi and Tsutomu Maruyama. "Accelerating HMMER search using FPGA". In: *2009 International Conference on Field Programmable Logic and Applications*. 2009 International Conference on Field Programmable Logic and Applications. Aug. 2009, pp. 332–337. DOI: 10.1109/FPL.2009.5272276.

[354] Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, and Zhiru Zhang. "ElasticFlow: A complexity-effective approach for pipelining irregular loop nests". In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). Nov. 2015, pp. 78–85. DOI: 10.1109/ICCAD.2015.7372553.

[355] National Institute of Standards Technology (NIST), Morris J. Dworkin, Meltem Sonmez Turan, and Nicky Mouha. "Advanced Encryption Standard (AES)". In: *NIST* (May 9, 2023).

[356] A.S. Terechko, E.J.D. Pol, and T.J. van Eijndhoven. "PRMDL: a machine description language for clustered VLIW architectures". In: *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*. Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001. Mar. 2001, pp. 821–. DOI: 10.1109/DATE.2001.915170.

[357] Terese. *Term Rewriting Systems*. Cambridge University Press, Mar. 20, 2003. 926 pp. ISBN: 978-0-521-39115-3.

[358] *The Verilog® Hardware Description Language*. Boston, MA: Springer US, 2002. ISBN: 978-0-387-84930-0 978-0-387-85344-4. DOI: 10.1007/978-0-387-85344-4.

[359] Benjamin Thielmann, Jens Huthmann, and Andreas Koch. "Memory Latency Hiding by Load Value Speculation for Reconfigurable Computers". In: *ACM Trans. Reconfigurable Technol. Syst.* 5.3 (Oct. 1, 2012), 13:1–13:14. ISSN: 1936-7406. DOI: 10.1145/2362374.2362377.

[360] Benjamin Thielmann, Jens Huthmann, and Andreas Koch. "PreCoRe - A Token-Based Speculation Architecture for High-Level Language to Hardware Compilation". In: *2011 21st International Conference on Field Programmable Logic and Applications*. 2011 21st International Conference on Field Programmable Logic and Applications. Sept. 2011, pp. 123–129. DOI: 10.1109/FPL.2011.31.

[361] Blaise Tine, Varun Saxena, Santosh Srivatsan, Joshua R. Simpson, Fadi Alzammar, Liam Cooper, and Hyesoon Kim. "Skybox: Open-Source Graphic Rendering on Programmable RISC-V GPUs". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, Mar. 25, 2023, pp. 616–630. ISBN: 978-1-4503-9918-0. DOI: 10.1145/3582016.3582024.

[362] Blaise Tine, Krishna Praveen Yalamarthy, Fares Elsabbagh, and Kim Hyesoon. "Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. New York, NY, USA: Association for Computing Machinery, Oct. 17, 2021, pp. 754–766. ISBN: 978-1-4503-8557-2. DOI: 10.1145/3466752.3480128.

[363] Parthasarathy Tirumalai, Meng Lee, and Michael Schlansker. *Parallelization of loops with exits on pipelined architectures*. Citeseer, 1990.

[364] Maël Tourres, Cyrille Chavet, Bertrand Le Gal, Jérémie Crenne, and Philippe Coussy. "Extended RISC-V hardware architecture for future digital communication systems". In: *2021 IEEE 4th 5G World Forum (5GWF)*. 2021 IEEE 4th 5G World Forum (5GWF). Oct. 2021, pp. 224–229. DOI: 10.1109/5GWF52925.2021.00046.

[365] Kaihui Tu, Xifan Tang, Cunxi Yu, Lana Josipović, and Zhufei Chu. *FPGA EDA: Design Principles and Implementation*. Singapore: Springer Nature, 2024. ISBN: 978-981-9977-54-3 978-981-9977-55-0. DOI: 10.1007/978-981-99-7755-0.

[366] Peng Tu and David Padua. "Efficient building and placing of gating functions". In: *ACM SIGPLAN Notices* 30.6 (June 1, 1995), pp. 47–55. ISSN: 0362-1340. DOI: 10.1145/223428.207115.

[367] Peng Tu and David Padua. "Gated SSA-based demand-driven symbolic analysis for parallelizing compilers". In: *Proceedings of the 9th international conference on Supercomputing*. ICS '95. New York, NY, USA: Association for Computing Machinery, July 3, 1995, pp. 414–423. ISBN: 978-0-89791-728-5. DOI: 10.1145/224538.224648.

[368] Yohann Uguen, Florent de Dinechin, and Steven Derrien. "Bridging high-level synthesis and application-specific arithmetic: The case study of floating-point summations". In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 2017 27th International Conference on Field Programmable Logic and Applications (FPL). Sept. 2017, pp. 1–8. DOI: 10.23919/FPL.2017.8056792.

[369] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. New York, NY, USA: Association for Computing Machinery, Feb. 22, 2017, pp. 65–74. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021744.

[370] Roddy Urquhart. *What is CodAL?* Codasip. Feb. 26, 2021. URL: https://codasip.com/2021/02/26/what-is-codal/ (visited on 09/18/2024).

[371] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. "Speculative Decoupled Software Pipelining". In: *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007). Sept. 2007, pp. 49–59. DOI: 10.1109/PACT.2007.4336199.

[372] Johan Van Praet, Dirk Lanneer, Werner Geurts, and Gert Goossens. "Chapter 4 - nML: A Structural Processor Modeling Language for Retargetable Compilation and ASIP Design". In: *Processor Description Languages*. Ed. by Prabhat Mishra and Nikil Dutt. Vol. 1. Systems on Silicon. Burlington: Morgan Kaufmann, Jan. 1, 2008, pp. 65–93. DOI: 10.1016/B978-012374287-2.50007-0.

[373] Arthur H. Veen. "Dataflow machine architecture". In: *ACM Comput. Surv.* 18.4 (Dec. 11, 1986), pp. 365–396. ISSN: 0360-0300. DOI: 10.1145/27633.28055.

[374] Bram Veenboer, Matthias Petschow, and John W. Romein. "Image-Domain Gridding on Graphics Processors". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). May 2017, pp. 545–554. DOI: 10.1109/IPDPS.2017.68.

[375] Juan Camilo Vega, Marco Antonio Merlini, and Paul Chow. "FFShark: A 100G FPGA Implementation of BPF Filtering for Wireshark". In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). May 2020, pp. 47–55. DOI: 10.1109/FCCM48280.2020.00016.

[376] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth C. Goldstein. "C to Asynchronous Dataflow Circuits: An End-to-End Toolflow". In: (Sept. 1, 2001). DOI: 10.1184/R1/6603986.v1.

[377] Sven Verdoolaege. "Presburger formulas and polyhedral compilation". In: *Polly Labs and KU Leuven* (2016).

[378] Muralidaran Vijayaraghavan and Arvind. "Bounded Dataflow Networks and Latency-Insensitive circuits". In: *2009 7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design*. 2009 7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design. July 2009, pp. 171–180. DOI: 10.1109/MEMCOD.2009.5185393.

[379] K. Wakabayashi and T. Yoshimura. "A resource sharing and control synthesis method for conditional branches". In: *1989 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*. 1989 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers. Nov. 1989, pp. 62–65. DOI: 10.1109/ICCAD.1989.76905.

[380] Wakabayashi and Tanaka. "Global scheduling independent of control dependencies based on condition vectors". In: Design Automation Conference. IEEE Computer Society, June 1, 1992, pp. 112–115. ISBN: 978-0-8186-2822-1. DOI: 10.1109/DAC.1992.227852.

[381] Haili Wang, Jinian Bian, Qiang Wu, and Yunfeng Wang. "iTuCoMe: HCDFG-based incremental tuning HW/SW co-design methodology for multi-level exploration". In: *Proceedings of the Ninth International Conference on Computer Supported Cooperative Work in Design, 2005.* Proceedings of the Ninth International Conference on Computer Supported Cooperative Work in Design, 2005. Vol. 2. May 2005, 978–983 Vol. 2. DOI: 10.1109/CSCWD.2005.194320.

[382] Jie Wang, Licheng Guo, and Jason Cong. "AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA". In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '21. New York, NY, USA: Association for Computing Machinery, Feb. 17, 2021, pp. 93–104. ISBN: 978-1-4503-8218-2. DOI: 10.1145/3431920.3439292.

[383] Shihang Wang, Jianghan Zhu, Qi Wang, Can He, and Terry Tao Ye. "Customized Instruction on RISC-V for Winograd-Based Convolution Acceleration". In: *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP). July 2021, pp. 65–68. DOI: 10.1109/ASAP52443.2021.00018.

[384] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. "C-LSTM: Enabling Efficient LSTM using Structured Compression Techniques on FPGAs". In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '18. New York, NY, USA: Association for Computing Machinery, Feb. 15, 2018, pp. 11–20. ISBN: 978-1-4503-5614-5. DOI: 10.1145/3174243.3174253.

[385] Xi Wang, John D. Leidel, Brody Williams, Alan Ehret, Miguel Mark, Michel A. Kinsy, and Yong Chen. "xBGAS: A Global Address Space Extension on RISC-V for High Performance Computing". In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). May 2021, pp. 454–463. DOI: 10.1109/IPDPS49936.2021.00054.

[386] Xi Wang, Brody Williams, John D. Leidel, Alan Ehret, Michel Kinsy, and Yong Chen. "Remote Atomic Extension (RAE) for Scalable High Performance Computing". In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020 57th ACM/IEEE Design Automation Conference (DAC). July 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218589.

[387] Xiuxiu Wang, Yipei Niu, Fangming Liu, and Zichen Xu. "When FPGA Meets Cloud: A First Look at Performance". In: *IEEE Transactions on Cloud Computing* 10.2 (Apr. 2022), pp. 1344–1357. ISSN: 2168-7161. DOI: 10.1109/TCC.2020.2992548.

[388] Robert N. M. Watson, Jonathan Woodruff, David Chisnall, Brooks Davis, Wojciech Koszek, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, Robert Norton, and Michael Roe. *Bluespec Extensible RISC Implementation: BERI Hardware reference*. UCAM-CL-TR-868. University of Cambridge, Computer Laboratory, 2015. DOI: 10.48456/tr-868.

[389] Nicholas Weaver. "Retiming, Repipelining and C-Slow Retiming". In: *Reconfigurable Computing*. Ed. by Scott Hauck and André Dehon. Systems on Silicon. Burlington: Morgan Kaufmann, Jan. 1, 2008, pp. 383–399. DOI: 10.1016/B978-012370522-8.50025-X.

[390] Reinhold P. Weicker. "Dhrystone: a synthetic systems programming benchmark". In: *Commun. ACM* 27.10 (Oct. 1, 1984), pp. 1013–1030. ISSN: 0001-0782. DOI: 10.1145/358274.358283.

[391] Dennis Weller, Fabian Oboril, Dimitar Lukarski, Juergen Becker, and Mehdi Tahoori. "Energy Efficient Scientific Computing on FPGAs using OpenCL". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. New York, NY, USA: Association for Computing Machinery, Feb. 22, 2017, pp. 247–256. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021730.

[392] Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. "Sponge-Based Control-Flow Protection for IoT Devices". In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE Computer Society, Apr. 1, 2018, pp. 214–226. ISBN: 978-1-5386-4228-3. DOI: 10.1109/EuroSP.2018.00023.

[393] Mario Werner, Erich Wenger, and Stefan Mangard. "Protecting the Control Flow of Embedded Processors against Fault Attacks". In: *Smart Card Research and Advanced Applications*. Ed. by Naofumi Homma and Marcel Medwed. Cham: Springer International Publishing, 2016, pp. 161–176. ISBN: 978-3-319-31271-2. DOI: 10.1007/978-3-319-31271-2_10.

[394] Mike Wissolik, Darren Zacher, Anthony Torza, and Brandon Da. "Virtex UltraScale+ HBM FPGA: a revolutionary increase in memory performance". In: *Xilinx Whitepaper* 3 (2017), p. 4.

[395] M.E. Wolf and M.S. Lam. "A loop transformation theory and an algorithm to maximize parallelism". In: *IEEE Transactions on Parallel and Distributed Systems* 2.4 (Oct. 1991), pp. 452–471. ISSN: 1558-2183. DOI: 10.1109/71.97902.

[396] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. "The CHERI capability model: revisiting RISC in an age of risk". In: *SIGARCH Comput. Archit. News* 42.3 (June 14, 2014), pp. 457–468. ISSN: 0163-5964. DOI: 10.1145/2678373.2665740.

[397] Qiang Wu, Yunfeng Wang, Jinian Bian, Weimin Wu, and Hongxi Xue. "A hierarchical CDFG as intermediate representation for hardware/software codesign". In: *IEEE 2002 International Conference on Communications, Circuits and Systems and West Sino Expositions*. IEEE 2002 International Conference on Communications, Circuits and Systems and West Sino Expositions. Vol. 2. June 2002, 1429–1432 vol.2. DOI: 10.1109/ICCCAS.2002.1179048.

[398] Jiahui Xu and Lana Josipović. "Automatic Inductive Invariant Generation for Scalable Dataflow Circuit Verification". In: *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD). Oct. 2023, pp. 1–9. DOI: 10.1109/ICCAD57390.2023.10323796.

[399] Jiahui Xu and Lana Josipović. "Suppressing Spurious Dynamism of Dataflow Circuits via Latency and Occupancy Balancing". In: *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '24. New York, NY, USA: Association for Computing Machinery, Apr. 2, 2024, pp. 188–198. ISBN: 9798400704185. DOI: 10.1145/3626202.3637570.

[400] Jiahui Xu, Emmet Murphy, Jordi Cortadella, and Lana Josipović. "Eliminating Excessive Dynamism of Dataflow Circuits Using Model Checking". In: *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '23. New York, NY, USA: Association for Computing Machinery, Feb. 12, 2023, pp. 27–37. ISBN: 978-1-4503-9417-8. DOI: 10.1145/3543622.3573196.

[401] Ke Xu, Xiaoyun Wang, and Dong Wang. "A Scalable OpenCL-Based FPGA Accelerator for YOLOv2". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). Apr. 2019, pp. 317–317. DOI: 10.1109/FCCM.2019.00058.

[402] Ruifan Xu, Youwei Xiao, Jin Luo, and Yun Liang. "HECTOR: A Multi-Level Intermediate Representation for Hardware Synthesis Methodologies". In: *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. ICCAD '22. New York, NY, USA: Association for Computing Machinery, Dec. 22, 2022, pp. 1–9. ISBN: 978-1-4503-9217-4. DOI: 10.1145/3508352.3549370.

[403] Yifan Yang, Qijing Huang, Bichen Wu, Tianjun Zhang, Liang Ma, Giulio Gambardella, Michaela Blott, Luciano Lavagno, Kees Vissers, John Wawrzynek, and Kurt Keutzer. "Synetgy: Algorithm-hardware Co-design for ConvNet Accelerators on Embedded FPGAs". In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '19. New York, NY, USA: Association for Computing Machinery, Feb. 20, 2019, pp. 23–32. ISBN: 978-1-4503-6137-8. DOI: 10.1145/3289602.3293902.

[404] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation". In: 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE Computer Society, Apr. 1, 2022, pp. 741–755. ISBN: 978-1-66542-027-3. DOI: 10.1109/HPCA53966.2022.00060.

[405] Peter Yiannacouras, Jonathan Rose, and J. Gregory Steffan. "The microarchitecture of FPGA-based soft processors". In: *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*. CASES '05. New York, NY, USA: Association for Computing Machinery, Sept. 24, 2005, pp. 202–212. ISBN: 978-1-59593-149-8. DOI: 10.1145/1086297.1086325.

[406] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. "Exploration and Customization of FPGA-Based Soft Processors". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (Feb. 2007), pp. 266–277. ISSN: 1937-4151. DOI: 10.1109/TCAD.2006.887921.

[407] Junghwan Yoon, Yezee Seo, Jaedong Jang, Mingi Cho, JinGoog Kim, HyeonSook Kim, and Taekyoung Kwon. "A Bitstream Reverse Engineering Tool for FPGA Hardware Trojan Detection". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. New York, NY, USA: Association for Computing Machinery, Oct. 15, 2018, pp. 2318–2320. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3278487.

[408] Tanner Young-Schultz, Lothar Lilge, Stephen Brown, and Vaughn Betz. "Using OpenCL to Enable Software-like Development of an FPGA-Accelerated Biophotonic Cancer Treatment Simulator". In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '20. New York, NY, USA: Association for Computing Machinery, Feb. 24, 2020, pp. 86–96. ISBN: 978-1-4503-7099-8. DOI: 10.1145/3373087.3375300.

[409] Drew Zagieboylo, Charles Sherk, Gookwon Edward Suh, and Andrew C. Myers. "PDL: a high-level hardware design language for pipelined processors". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. New York, NY, USA: Association for Computing Machinery, June 9, 2022, pp. 719–732. ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523455.

[410] Ali Mustafa Zaidi and David Greaves. "A New Dataflow Compiler IR for Accelerating Control-Intensive Code in Spatial Hardware". In: *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. 2014 IEEE International Parallel & Distributed Processing Symposium Workshops. May 2014, pp. 122–131. DOI: 10.1109/IPDPSW.2014.18.

[411] Florian Zaruba and Luca Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (Nov. 2019), pp. 2629–2640. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114.

[412] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '15. New York, NY, USA: Association for Computing Machinery, Feb. 22, 2015, pp. 161–170. ISBN: 978-1-4503-3315-3. DOI: 10.1145/2684746.2689060.

[413] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. "A Hardware Design Language for Timing-Sensitive Information-Flow Security". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, Mar. 14, 2015, pp. 503–516. ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694372.

[414] Jiliang Zhang, Congcong Chen, Jinhua Cui, and Keqin Li. "Timing Side-channel Attacks and Countermeasures in CPU Microarchitectures". In: *ACM Comput. Surv.* 56.7 (Apr. 9, 2024), 178:1–178:40. ISSN: 0360-0300. DOI: 10.1145/3645109.

[415] Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, and Arvind Arvind. "Composable Building Blocks to Open up Processor Design". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). Oct. 2018, pp. 68–81. DOI: 10.1109/MICRO.2018.00015.

[416] Yi Zhang, Steve Haga, and Rajeev Barua. "Execution history guided instruction prefetching". In: *Proceedings of the 16th international conference on Supercomputing*. ICS '02. New York, NY, USA: Association for Computing Machinery, June 22, 2002, pp. 199–208. ISBN: 978-1-58113-483-4. DOI: 10.1145/514191.514220.

[417] Yichi Zhang, Junhao Pan, Xinheng Liu, Hongzheng Chen, Deming Chen, and Zhiru Zhang. "FracBNN: Accurate and FPGA-Efficient Binary Neural Networks with Fractional Activations". In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '21. New York, NY, USA: Association for Computing Machinery, Feb. 17, 2021, pp. 171–182. ISBN: 978-1-4503-8218-2. DOI: 10.1145/3431920.3439296.

[418] Zhiru Zhang and Bin Liu. "SDC-based modulo scheduling for pipeline synthesis". In: *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). Nov. 2013, pp. 211–218. DOI: 10.1109/ICCAD.2013.6691121.

[419] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine". In: *Fourth Workshop on Computer Architecture Research with RISC-V* (May 2020).

[420] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. New York, NY, USA: Association for Computing Machinery, Feb. 22, 2017, pp. 15–24. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021741.

[421] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs". In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '18. New York, NY, USA: Association for Computing Machinery, Feb. 15, 2018, pp. 269–278. ISBN: 978-1-4503-5614-5. DOI: 10.1145/3174243.3174255.

[422] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. "Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL". In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '18. New York, NY, USA: Association for Computing Machinery, Feb. 15, 2018, pp. 153–162. ISBN: 978-1-4503-5614-5. DOI: 10.1145/3174243.3174248.

[423] Yu Zou and Mingjie Lin. "Massively Simulating Adiabatic Bifurcations with FPGA to Solve Combinatorial Optimization". In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '20. New York, NY, USA: Association for Computing Machinery, Feb. 24, 2020, pp. 65–75. ISBN: 978-1-4503-7099-8. DOI: 10.1145/3373087.3375298.

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

Université de Rennes

Titre : Synthèse de haut niveau de processeurs à jeu d'instructions

Mot clés : Synthèse de haut niveau, exécution spéculative, processeurs à jeu d'instructions

Résumé : Cette thèse porte sur la synthèse automatique de processeurs à jeu d'instructions en utilisant la synthèse de haut niveau (HLS). En particulier, nous visons à générer automatiquement des cœurs de processeurs pipelinés in-order à partir d'une description de haut niveau en C sous la forme d'un simulateur de jeu d'instructions (ISS). Au cours de notre travail, nous avons développé un flot de conception matérielle entièrement automatisé qui permet de compiler une description algorithmique en circuit spéculatif, SpecHLS. Nous proposons un ensemble de transformations de code basées sur le pipeline spéculatif de boucles, afin de révéler des opportunités de spéculation sur le flot de contrôle et la mémoire dans du code C, et nous générons du code spéculatif synthétisable à l'aide d'une chaîne d'outils de HLS commerciale. SpecHLS est capable de gérer plusieurs spéculations entremêlées, des spéculations indépendantes dans des modules matériels découplés, ainsi que la spéculation mémoire. Notre travail aboutit à un flot de conception capable de générer plusieurs instances de processeurs RISC-V in-order à partir d'un ISS. Nous montrons que nous pouvons explorer efficacement un espace de conception avec des centaines de milliers de configurations matérielles spéculatives possibles en quelques minutes, et générer des processeurs compétitifs avec des cœurs de processeurs embarqués.

Title: High-Level Synthesis of Instruction Set Processors

Keywords: High-Level Synthesis, Speculative Execution, Instruction Set Processors

Abstract: This thesis focuses on automatically synthesizing instruction set processors using High-Level Synthesis (HLS). In particular, we aim at automatically generating in-order pipelined processor cores from a high-level description in C in the form of an Instruction Set Simulator (ISS). During our work, we developed a fully-automated hardware design flow that can compile an algorithmic description to speculative hardware, SpecHLS. We propose a set of source-to-source transformations that build on top of speculative loop pipelining to expose control flow and memory speculation opportunities in C code, and we generate speculation-enabled code that can be synthesized using a commercial HLS toolchain. SpecHLS can handle multiple interacting speculations, independent speculations in decoupled hardware modules, and memory speculation. Our work results in an end-to-end processor design flow capable of generating multiple instances of in-order RISC-V processors from an ISS. We show that we can efficiently explore a design space with hundreds of thousands of possible speculative hardware configurations in minutes and generate processor designs competitive with state-of-the-art embedded MCU-class cores.