Titre: Title:	Définition, spécification et génération automatique d'architectures de modules réseau ciblant des FPGA
Auteur: Author:	Rajat Singh Rajput
Date:	2025
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Singh Rajput, R. (2025). Définition, spécification et génération automatique d'architectures de modules réseau ciblant des FPGA [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/62574/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/62574/
Directeurs de recherche: Advisors:	François-Raymond Boyer, & Yvon Savaria
Programme: Program:	Génie électrique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Définition, spécification et génération automatique d'architectures de modules réseau ciblant des FPGA

RAJAT SINGH RAJPUT
Département de génie électrique

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie électrique

Janvier 2025

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Définition, spécification et génération automatique d'architectures de modules réseau ciblant des FPGA

présenté par **Rajat SINGH RAJPUT**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Tarek OULD-BACHIR, président
François-Raymond BOYER, membre et directeur de recherche
Yvon SAVARIA, membre et codirecteur de recherche
Guy BOIS, membre

DÉDICACE

Merci à mes proches pour le support...

REMERCIEMENTS

Je tiens à exprimer ma sincère gratitude à tous ceux qui m'ont aidé ou encouragé au cours de deux dernières années. J'aimerais remercier mon directeur de recherche, Yvon Savaria, ainsi que mon codirecteur, François-Raymond Boyer de m'avoir soutenu et aidé tout au long de ce projet. Je voudrais également remercier les membres du laboratoire pour l'aide aux installations d'appareils et les discussions. Je remercie Bill Pontikakis pour son aide dans l'orientation du projet.

De plus, je dois mes plus sincères remerciements à ma famille bien-aimée pour un support financier et moral.

RÉSUMÉ

L'utilisation des langages de programmation pour optimiser toutes sortes de travaux complexes est la norme. Le langage P4 a besoin d'un tel support pour se développer avec l'aide de langages de haut niveau tel que le C++ et Python. En effet, il est présentement compliqué de réaliser des communications accélérées sur circuit logique programmable (FPGA) sans une seule manipulation manuelle du code en langage de description matériel (HDL). Dans cette étude, nous présentons une méthode de communication plus simple pour tous les utilisateurs d'externes des processeurs de paquets (P4). Notre objectif est d'utiliser des interfaces Python générées par une architecture personnalisée de P4. La génération automatique des interfaces va simplement permettre au programmeur de haut niveau d'avoir accès à des fonctionnalités de bas niveau sans avoir à programmer du HDL. C'est important, car généralement les programmeurs P4 ne connaissent pas ou ne veulent pas manipuler le HDL. La génération et l'interconnectivité des modules matériels pour les pipelines P4 sont effectuées sur une base automatique pour les utilisateurs d'externes qui veulent avoir plus de fonctionnalités dans leur réseau de communication. Les interfaces générées incluent l'accès direct à la mémoire (DMA) et l'interconnexion des composants périphériques, permettent la mise en œuvre de pont de communication multipériphériques flexibles. Les résultats de simulation valident la véritable fonctionnalité de notre méthodologie d'interface de pointe. En effet, une fonction externe au programme P4 a été connectée à l'aide d'interface générée automatiquement. Cette fonction externe communique avec le pipeline P4 à travers les modules matériels générés. En plus d'améliorer l'efficacité de développement, ce projet a une faible consommation de ressources matérielles et la solution est fonctionnelle dans l'industrie, car celle-ci est validée par les outils d'Intel. La compilation avec Quartus des modules matériels générés à partir du compilateur d'Intel et celles générées par l'outil Python montre une utilisation de ressources de 1,81 % de la logique (table de recherche ou LUT) pour une puce Arria10. De plus, les ressources sont limitées à 1,62 % pour la mémoire (bloc de RAM) et 0,96 % pour les bascules (registres). La différence avec le pipeline sans interface générée est de 281 registres et 194 LUT, donc une différence de 1,64 % et de 4,54 % pour le nombre de registres et le nombre d'unités logique (LUT) respectivement. La méthode proposée est efficace pour l'utilisateur et améliore la flexibilité du langage P4.

ABSTRACT

Optimizing and automating complex tasks is crucial in this world of fast advancement. Indeed, existing software-only programmable switches offer low performance, and those based on complex ASICs lack flexibility. The solution to this scientific concern was the arrival of the P4 packet processing language. Once again, developing networking hardware using traditional hardware description language is complex and time-consuming. P4 is a domain-specific language proposed to facilitate network equipment development and programmable switches. Software developers still consider the language to be missing some important external functions. This thesis proposes a simple communication method to extend software and ASIC-based switches to improve the performance of software switches and the flexibility of ASIC switches. The best way to visualize this scenario for a packet processing language like P4 is to have a flexible autogenerated communication procedure. The desired functionality is assumed to be expressed using the P4 language for all users of external packet processors (P4). We aim to use Python interfaces generated by a custom architecture in P4 and Python tools. Auto-generation of interfaces is proposed to reduce the development effort required from application developers to obtain hybrid solutions leveraging FPGAs. It is important because, generally, P4 programmers do not know or do not want to manipulate a hardware description language (HDL). The generation and interconnectivity of hardware modules for P4 pipelines are performed automatically. The proposed solution exploits direct memory access (DMA) and peripheral component interconnection interfaces to implement flexible multi-peripheral communication bridges. Simulation results validate the proper functionality of our state-of-the-art P4 interface methodology. Indeed, an external function to the P4 program was connected using an autogenerated interface. In addition to improving development efficiency, this project has low consumption of hardware resources, and the solution is functional in the industry because Intel tools validate it. Compiling with Quartus the hardware modules generated by Intel's compiler and those generated by the Python tool show a resource usage of 1.81 % of the logic (lookup table or LUT) for an Arria10 chip. Additionally, resources are limited to 1.62 % for memory (RAM block) and 0.96 % for flip-flops (registers). The proposed method is efficient for the user and improves the flexibility of the P4 language. The difference with the pipeline without generated interface is 281 registers and 194 LUTs, so a difference of 1.64 % and 4.54 % for the number of registers and the number of logical units (LUT) respectively. The proposed method is efficient for the user and improves the flexibility of the P4 language.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

ALM	Adaptive Logic Module
AMBA	Advanced Microcontroller Bus Architecture
AOE	Architecture-Owned Extern
API	Application Programming Interface
ARM	Advanced RISC Machine
AXI	Advanced Extensible Interface
BRAM	Block Random Access Memory
CPU	Central Processing Unit
DFA	Deterministic Finite Automaton
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DUT	Device Under Test
EOP	End Of Packet
FIFO	First In First Out
FPGA	Field Programmable Gate Arrays
HDL	Hardware Description Language
HLS	High Level Synthesis
IP	Intellectual Property
LUT	Look-Up Tables
NIC	Network Interface Card
PAC N3000	Programmable Acceleration Card N3000
PCIe	Peripheral Component Interconnect express
PIE	Position-Independent Executables
P4	Programming Protocol-independent Packet Processors
RTL	Register Transfer Level
SDN	Software-Defined Networking
SOC	System-on-a-Chip
SOP	Start Of Packet
VHDL	Very High-Speed Integrated Circuit Hardware Description Language

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

1.1 Contexte

Le matériel utilisé dans les réseaux de données, également appelé équipement réseau ou périphériques du réseau informatiques, est un appareil électronique nécessaire à la communication et à l'interaction entre les appareils d'un réseau informatique. Plus précisément, ces équipements assurent la transmission de données dans un réseau de données informatique. Les unités qui sont le dernier récepteur ou celles qui génèrent des données sont appelées hôtes, systèmes finaux ou équipements terminaux de données. Dans une perspective plus large, les téléphones mobiles, les tablettes et les appareils associés à l'Internet des objets peuvent également être considérés comme du matériel de réseau. À mesure que la technologie progresse et que les réseaux IP sont intégrés aux infrastructures des bâtiments et aux services publics domestiques, le terme matériel réseau devient ambigu en raison du nombre considérablement croissant de points finaux compatibles avec le réseau.
Les applications réseau se développent rapidement en raison de la demande de vitesses de transfert de données plus élevées et les marchés des serveurs et de l'infonuagique. En conséquence, les centres de données continuent d'évoluer pour répondre à ces demandes. Ils optent pour une solution qui est plus universelle, comme le langage P4, que spécifique. Les programmes P4 sont conçus pour être indépendants de l'implémentation, donc ils peuvent être compilés sur de nombreux types de support matériels tels que des processeurs à usage général, des FPGA, des systèmes sur puce, des processeurs réseau et des ASIC. Les périphériques réseau jouent un rôle essentiel en facilitant la communication entre les ordinateurs et les réseaux Internet.
Le marché des réseaux continue d'exiger une transmission de données plus fluide et de nombreux défis sont propres au marché des réseaux comme la perte de canal et la compensation de perte de canal sur la ligne de transmission. Cependant, dans certains systèmes, le délai de traitement peut être assez important, en particulier lorsque les routeurs exécutent des algorithmes de chiffrement complexes et examinent ou modifient le contenu des paquets [1]. L'inspection approfondie des paquets effectuée par certains réseaux examine le contenu des paquets pour des raisons de sécurité, juridiques ou autres, ce qui peut entraîner des retards très importants et n'est donc effectuée qu'à des points d'inspection sélectionnés. Le langage P4 n'est pas la solution au déchargement du réseau lors de ces situations. Dans le cas où le nombre de calculs est important, il n'est essentiellement pas possible de faire le déchargement vers un processeur ou FPGA de haute performance et augmenter le parallélisme avec d'autres

équipements du réseau ou périphériques du réseau informatiques.
En fait, le réseau est composé des interfaces comme le PCIe, l'AXI et SPI ou l'I2C qui relient les composants de la topologie très efficacement. La communication peut se faire pour la distribution de fonctionnalité qui n'est pas réalisable en P4 ou celle qui demande plus de matériel. Cette fonctionnalité peut être transférée à un autre composant comme FPGA ou le CPU de l'hôte [2]. En effet, les interfaces sont programmables en matériel et elles permettent l'échange d'informations à l'aide des ports en entrés et en sorties. Elles sont très visibles, dans le marché, configurées en Verilog (HDL).

1.2 Problématique

Le problème avec les interfaces configurées en langage matériel est la consommation de temps et d'énergie. Des études montrent que les programmeurs des langages matériels consacrent beaucoup de temps dans la conception du matériel exact sans problème ou bogue [3]. Tout simplement parce qu'il n'est pas possible de valider le code matériel sans les déploiements en matériel et sans effectuer une vérification avec le logiciel adéquat. Les modules programmés en matériel sont accessibles au logiciel et font des échanges de données pour effectuer la logique du matériel implémenté. Il est à noter que les FPGA favorisent l'implémentation flexible du matériel. Les interfaces qui sont utilisées pour communiquer l'information d'un FPGA à l'autre ou vers un hôte sont aussi programmées en HDL (RTL) [2].

Par conséquent, ce principe de fonctionnement n'est pas favorable pour les langages descriptifs comme le P4 qui désirent avoir accès à la configuration des interfaces matérielles. Les limites de P4 font en sorte qu'il n'est pas possible de programmer les interfaces flexibles directement en P4 en passant simplement par l'architecture de celle-ci.

Ces interfaces sont possibles si un module décrit en P4 collabore avec une autre langage de haut niveau comme le C++ ou le Python. D'ailleurs, Python est utilisé de plus en plus dans la programmation et l'automatisation, car il favorise la simplicité de développement pour le programmeur.
Bref, le problème vient du fait que les compilateurs applicables au programme P4 sont toujours dépendants des langages comme le VHDL et le Verilog pour la configuration des interfaces. Il est aussi vrai que les interfaces sont parfois pré-implémentées et disponible dans une bibliothèque que le compilateur consulte pour la synthèse des interfaces. Donc pour interconnecter un pipeline P4 avec le réseau, tenter de le faire avec des interfaces préimplémentées est très limité.

1.3 Objectifs de recherche

Ce mémoire a pour objectif premier de proposer un outil permettant d'étendre l'accessibilité du langage P4. Les applications écrites en HDL venant du compilateur P4 auront accès aux interfaces flexibles. Ces modules HDL devront faire la communication à une fonctionnalité externe au travers d'un code d'interface codé d'un langage de haut niveau au lieu d'être dans un langage de bas niveau (VHDL, Verilog). Les programmes P4 peuvent, quant à eux, avoir un code qui permet l'utilisation des externes.

Les externes personnalisés interconnectés avec le pipeline P4 dans l'architecture du programme P4 permettent plus de fonctionnalité aux actions des tableaux P4. Les actions ayant une fonction externe favorisent l'encapsulation et la conception de code modulaire en permettant de spécifier et d'implémenter différents comportements séparément. Cela accélère les temps de compilation, car apporter une petite modification nécessite uniquement de recompiler cette unité de traduction et de recréer les liens, et non de recompiler l'intégralité de la base de code.
D'ailleurs, il est aussi possible de voir, lors de la compilation, que le code P4 compilé, dans ce projet, utilise le compilateur HLS pour générer le code HDL pour des tableaux P4.
La déclaration de fonctions via un externe sera visible en langage matériel, après la compilation du code P4, la fonction externe sera accessible dans le code HDL à travers les IPs HDL du tableau P4 du programme P4 de l'utilisateur. Donc, ce code HDL peut, à ce moment-là, être connecté aux interfaces provenant d'un autre langage de haut niveau comme le C++ ou le Python.

1.4 Contributions

La principale contribution de ce projet est un générateur d'interface qui étend un langage pour la description du plan de données P4. Premièrement, le code P4 est utilisé pour générer des modules matériels pour une application réseau. Ensuite, ce projet permet de générer automatiquement les interfaces pour cette application P4 à l'aide de l'outil de Python «PyXHDL». Deuxièmement, l'environnement réalise les interconnexions avec ces interfaces aux externes de l'utilisateur. Un externe peut être écrit en C++ et peut faire un retour de la valeur calculée à travers ces interfaces générées pour la fonctionnalité de l'externe. La gestion de ces interfaces est sous le contrôle du script Python d'Intel.

1.5 Vérification

L'environnement de vérification est basé sur l'utilisation d'un externe personnalisé dans le code P4. Cet externe sera utilisé pour automatiquement générer des interfaces codées en PyXHDL. Ce projet utilise Python et la simulation matérielle pour illustrer le bon fonctionnement des externes communiquant avec le pipeline P4.
Les interfaces performent des échanges à partir du module de la mémoire BRAM et en communiquant avec le processeur émulé à travers le module DPI-C utilisés en SystemVerilog.

1.6 Organisation du mémoire

La suite de ce mémoire est organisée comme suit. Tout d'abord, une revue des travaux sur les recherches existantes est résumée dans le chapitre II. Par la suite, le programme P4 pour le compilateur d'Intel et l'interface Python sont exposés dans le chapitre III. Dans le chapitre IV, une discussion est présentée sur les résultats du transfert de données. Finalement, le chapitre V conclut ce mémoire et propose des travaux futurs dans le but de favoriser le développement matériel (bas niveau). Avant tout, faisons un petit aperçu de la composition du langage P4 à la section suivante.

1.7 Langage P4

Cette section présente les notions préliminaires concernant le projet. Il s'agit de la nouvelle structure et le langage de programmation de réseaux en P4, la logique des externes ainsi que les méthodes utilisées pour l'autogénération par la programmation de haut niveau.
L'émergence des commutateurs programmables a gagné du terrain parmi les principaux fournisseurs de réseaux. L'autogénération de modules de bas niveau utilisant un langage de haut niveau comme P4 a révolutionné la conception des systèmes de communication et les générations d'IP [2]. Dans ce type de gestion, l'utilisation de Python a constitué une avancée majeure dans la synthèse moderne de bas niveau.

1.7.1 Trois composants du plan de donné P4

Le langage P4 décrit le comportement de la procédure de traitement des paquets en trois parties. La première partie est liée à l'extraction de l'en-tête du paquet lors de l'étape de l'analyseur [4].

La deuxième partie est associée à la logique de contrôle des paquets lors de la phase de correspondance et d'action. Ici, les en-têtes de paquets sont utilisés pour créer des clés de

recherche en utilisant des champs de paquets ou des métadonnées calculées.

La troisième partie principale implique l'inverse de l'analyse, à savoir la construction de paquets.

1.7.2 Architecture P4

Le fichier d'architecture P4 sert de descripteur principal pour les interfaces P4 à générer par le compilateur P4. Ce fichier est valide pour tous les blocs et externes de P4 [4]. Les connexions actuelles pour les externes sont rendues possibles via un bus déjà disponible en HDL, tel que l'AXI [2]. Les modules d'interface sont instanciés par le compilateur pour établir la connexion au pipeline P4. L'architecture peut être différente pour différents FPGA et ceci est aussi valide pour différentes versions de P4 comme le $P4_{16}$.

Listing 1.1 La définition de mon externe dans un architecture P4

```
1 #include <core.p4>
2 extern Custom_Extern_Component{
3  Custom_Extern_Component();
4  void update(in bit<256> data)
5 }
```

1.7.3 Externe en P4

Comme les unités de somme de contrôle et bien d'autres, le langage P4 possède différents éléments externes qui rendent le langage performant. Ces externes sont utilisés dans le langage P4 pour exécuter des API définies. Leurs comportements internes sont fixes. Un objet externe définit un ensemble de méthodes accessibles aux programmes P4, offrant un moyen d'interagir avec ces composants spécifiques à l'architecture sans exposer les détails internes de leurs implémentations internes [4,5]. Ce code 1.1 est la définition d'externe dans le fichier d'architecture P4. Cela inclut l'utilisation du type «extern» pour définir le constructeur d'objet externe personnalisé Custom_Extern_Component() comme en C++ puis une déclaration de la méthode externe nommée update().

Comme dans le code 1.2 la déclaration de la méthode Custom_Architecture<Header, Metadata>(), créée l'instance de l'objet externe Custom_Extern_Component(). Il est alors possible de faire un appel à l'instance externe (comme une fonction en C++) dans les actions du bloc de contrôle du programme P4 illustrés au code 1.3. L'objet Custom_Pipeline<Header, Metadata>() déclare la structure du pipeline comme faisant partie d'une entité en utilisant un type tel que «package». Il en va de même pour l'entité de niveau supérieur de l'architec-

ture, Custom_Architecture<Header, Metadata>(), qui inclut désormais l'instance externe de l'utilisateur (custom_extern) ainsi que l'instance d'objet de ce pipeline appelée pipe0.

Listing 1.2 La liaison de l'externe au pipeline P4 dans l'architecture P4

```
package Custom_Pipeline<Header, Metadata>(
 Parser<Header, Metadata> parser,
 Ingress<Header, Metadata> ingress,
 Deparser<Header> deparser
);
package Custom_Architecture<Header, Metadata> (
 Custom_Pipeline<Header, Metadata> pipe0,
 Custom_Extern_Component custom_extern = Custom_Extern_Component()
);
```

Listing 1.3 Utilisation l'externe dans l'action P4

```
action port_ctrl(bit<8> eport) {
    md_intr.egress_port = eport;
    bit<256> AOE_data= 7;
    aoe_inst.update(AOE_data);
}
```

CHAPITRE 2 REVUE DE LA LITTERATURE

Dans ce chapitre, nous mettons en évidence certaines découvertes importantes liées à ce projet. Donc, cette section abordera les sujets suivants : la génération de HDL à l'aide d'outils Python, les interfaces dans les réseaux de communication, l'interface P4, ainsi que les interfaces du projet Pigasus.

2.1 HDL en Python

Les langages de haut niveau facilitent la génération des langages matériels comme VHDL [6]. Python a connu un regain de popularité pour plusieurs raisons. Facilité d'apprentissage et d'utilisation comme la syntaxe de Python est claire et intuitive, ce qui la rend facile à apprendre et à utiliser pour les débutants. Il permet aux développeurs d'écrire moins de code pour accomplir plus de tâches par rapport à de nombreux autres langages de bas niveau RTL [7].

Python prend en charge plusieurs paradigmes de programmation, notamment la programmation procédurale, orienté objet et fonctionnelle. Il a une large gamme d'applications utilisées dans le développement Web, l'analyse de données, l'intelligence artificielle, le calcul scientifique, l'automatisation, etc. Python dispose d'une vaste bibliothèque standard et d'un riche écosystème de bibliothèques tierces pour pratiquement toutes les applications. De plus, une communauté vaste et active contribue à une richesse de ressources, de documentation et de support.

Python peut facilement s'interfacer avec des langages comme C/C++ (via des extensions) et Java (via Jython). Idéal pour le prototypage en raison de sa nature de haut niveau et de son cycle de développement rapide. Pour la science des données et apprentissage automatique, c'est le langage préféré des scientifiques grâce à des bibliothèques telles que NumPy, pandas, TensorFlow et scikit-learn.

2.1.1 Compatibilité de Python

Python est indépendant de la plate-forme et peut fonctionner sous Windows, macOS et Linux. Python gagne également du terrain dans le domaine de la conception matérielle, notamment en tant qu'outil de génération RTL. Python permet aux concepteurs d'écrire des descriptions abstraites de haut niveau du matériel qui peuvent ensuite être converties en RTL. Les constructions de haut niveau et les fonctionnalités orientées objet permettent un

code réutilisable et maintenable.

La nature interprétative et la syntaxe concise de Python facilitent le prototypage et l'itération rapides des conceptions matérielles. Les outils de débogage robustes de Python facilitent l'identification et la résolution des problèmes dès le début du processus de conception [8].

Python peut s'intégrer aux outils EDA (Electronic Design Automation) existants, facilitant ainsi un flux de travail transparent.

Des bibliothèques telles que MyHDL, PyMTL et Chisel (qui utilisent Scala mais ont un objectif similaire) permettent aux concepteurs de décrire le matériel en Python ou dans des langages de haut niveau similaires, qui sont ensuite convertis en Verilog/VHDL. Entrée de conception flexible avec ces outils qui offrent une entrée de conception plus flexible et expressive par rapport au DSL (Domain Specific Language) HDL traditionnel.

2.2 HDL généré par les outils Python

Les interfaces dans certains autres projets utilisent aussi python, mais avec les librairies telles que le PyHDL ou le MyHDL. Ces librairies permettaient un accord parfait prédéfini pour la génération de code matériel. Bref, il y a plusieurs outils qui favorisent l'utilisation du langage de haut niveau (Python) pour faire la génération du code en bas niveau (RTL). Les outils et leurs environnements sont tous différents les uns des autres, ainsi que les communautés les supportant impactent grandement la flexibilité et l'extensibilité des outils, voire la table 2.1. L'outil PyXHDL est considéré faible pour la catégorie de Soutien communautaire, car son développement n'est pas fait par un grand nombre de personnes. Il est considéré de qualité moyenne pour la flexibilité, l'extensibilité et ainsi que pour la performance et l'efficacité. Cet outil gratuit est assez puissant, rapide et facile à utiliser malgré son apparition récente sur le marché.

En effet, PyXHDL prend avantage d'être compatible avec les outils EDA existants. Cet outil de l'environnement Python permet aussi, comme le Chisel une bonne prise en charge de la vérification et des tests.

Les outils comme le MyHDL permettent d'avoir pratiquement les mêmes fonctionnalités que les autres de la liste, donc de faire la génération du code matériel avec un langage de haut niveau [9]. Plus spécifiquement, les générateurs MyHDL sont similaires aux blocs «Always» dans Verilog et aux processus en VHDL. Un module matériel (appelé bloc dans la terminologie MyHDL) est modélisé comme une fonction qui renvoie des générateurs. Cette approche facilite la prise en charge de fonctionnalités telles que la hiérarchie arbitraire, l'association de ports, les tableaux d'instances et l'instanciation conditionnelle. De plus, MyHDL propose des classes

Tableau 2.1 Utilisation de Python comme langage de programmation pour le matériel RTL

Outils	Soutien communautaire	Flexibilité et extensibilité	Performance et efficacité	Licence et coût
MyHDL	Moyen	Élevé	Élevé	gratuite
PyMTL	Moyen	Élevé	Élevé	gratuite
PHDL	Faible	Moyen	Moyen	gratuite
Migen	Moyen	Élevé	Élevé	gratuite
Chisel	Élevé	Élevé	Élevé	gratuite
SpinalHDL	Moyen	Élevé	Élevé	gratuite
NGEN	Faible	Moyen	Moyen	gratuite
HWT	Faible	Moyen	Moyen	gratuite
PyXHDL	Faible	Moyen	Moyen	gratuite

qu'implémentent les concepts traditionnels de description du matériel. Il fournit une classe de signaux pour prendre en charge la communication entre les générateurs, une classe pour prendre en charge les opérations orientées bits et une classe pour les types d'énumération.

D'un autre côté, en PyMTL le concepteur commence par développer une conception sous test (DUT «Device Under Test») au niveau fonctionnel et un banc de test entièrement en Python. Ensuite, le DUT est affiné de manière itérative au niveau du cycle et au niveau du transfert de registre, ainsi que par la vérification et l'évaluation à l'aide d'une simulation basée sur Python et du même banc de test. Le concepteur peut, après la traduction d'un modèle PyMTL en Verilog et utiliser le même banc de test pour la cosimulation. Notez que les concepteurs peuvent également co-simuler le code source SystemVerilog existant avec un banc de test PyMTL.

D'ailleurs, Chisel (Constructing Hardware In a Scala Embedded Language) est un langage de construction de matériel intégré dans le langage de programmation de haut niveau Scala [10]. Chisel est une bibliothèque de définitions de classes spéciales, d'objets prédéfinis et de conventions d'utilisation au sein de Scala. Ainsi, lors de la conception du Chisel, le programmeur écrit en réalité un programme Scala qui construit un graphe matériel.

Enfin, le HWT est le pont entre HLS et HDL. Il offre un style de codage comme HLS, mais en même temps il vous permet de manipuler des objets HDL [11]. Cela signifie qu'il est un peu plus lent d'écrire un prototype que dans HLS, mais vous savez toujours quoi, comment et pourquoi se passe. HWT utilise le «netlist» de haut niveau pour la représentation interne de la conception cible. Les «netlist» optimisées sont générées à partir d'instructions de code habituelles, d'appels de fonctions, d'instructions, etc. Les processus matériels sont automatiquement résolus. Cette «netlist» est facile à utiliser et facile à modifier ou à analyser

par l'utilisateur s'il manque quelque chose dans la bibliothèque principale. Les modes de sérialisation permettent également de modifier le comportement du composant pendant la sérialisation.

Tous les outils présentés dans cette section sont très semblables. Ils permettent de faire la génération du code de bas niveau en utilisant les langages de haut niveau comme le Python. Ces outils n'ont pas les mêmes plates-formes de fonctionnement et ne sont pas sous la gestion du même propriétaire et sont compatibles avec de différents composants comme les FPGA, les CPU ainsi que divers logiciels. Alors, une comparaison de fiabilité dépend surtout sur la méthode de l'implémentation et les besoins en matière de fonctionnalité (le besoin d'un code simple, efficace et facile à comprendre).

2.3 Interfaces dans les réseaux de communication

Dans les réseaux et les systèmes numériques, une interface fait référence à une méthode standardisée permettant aux composants de communiquer entre eux. Ces interfaces définissent les règles et protocoles de transfert de données, garantissant l'interopérabilité entre les différents appareils et systèmes.

2.3.1 PCIe (Interconnexion de composants périphériques Express)

PCIe est une interface de communication haute vitesse utilisée pour connecter des composants haute vitesse telle que des cartes graphiques, des SSD et des cartes réseau à la carte mère. Ayant une bande passante élevée, faible latence, prend en charge plusieurs voies (x1, x4, x8, x16) pour des performances évolutives [12, 13].

2.3.2 DMA (module d'accès direct à la mémoire)

Les DMA permettent aux périphériques matériels de transférer des données vers/depuis la mémoire sans impliquer le processeur, libérant ainsi les ressources du processeur. Les DMA sont capables d'effectuer le transfert de données à grande vitesse. Ils sont efficaces pour les transferts de données volumineux et ils réduisent la surcharge du processeur [13].

2.3.3 AXI (interface extensible avancée)

AXI fait partie de la spécification ARM AMBA (Advanced Microcontroller Bus Architecture), utilisée pour le transfert de données à grande vitesse dans les SoC (System on Chips).

Performant de haute performance et à débit élevé, les bus AXI prennent en charge plusieurs transactions en cours, longueurs de rafale flexibles.

2.4 Interfaces P4

Dans le contexte de notre recherche, les interfaces servent de conduits intégraux permettant d'isoler le réseau P4 par rapport au plan de données. Une interface est définie comme un point de communication facilitant l'interaction entre des entités distinctes au sein des modules P4 [4]. Il faut noter ici que chaque entité est supposée fonctionner indépendamment selon l'architecture définie par l'utilisateur. Les interfaces permettent aux diverses entités de d'échanger des données sur des mécanismes de connexion reconnus tels que le PCIe exploitée dans notre projet. Traditionnellement, P4 fait des connexions en utilisant des interfaces AXI.

Les interfaces en P4 découlent directement du code P4 dans le fichier de son architecture [4,5]. Le fichier comme expliqué auparavant est lié au «core» du code P4 et permet de former les liens en utilisant des liens possibles dans les fichiers du compilateur. Le code P4 est compilé traditionnellement à une version JSON pour la partie de l'application logicielle du plan de contrôle, mais les compilateurs modernes génèrent aussi les modules matériels.

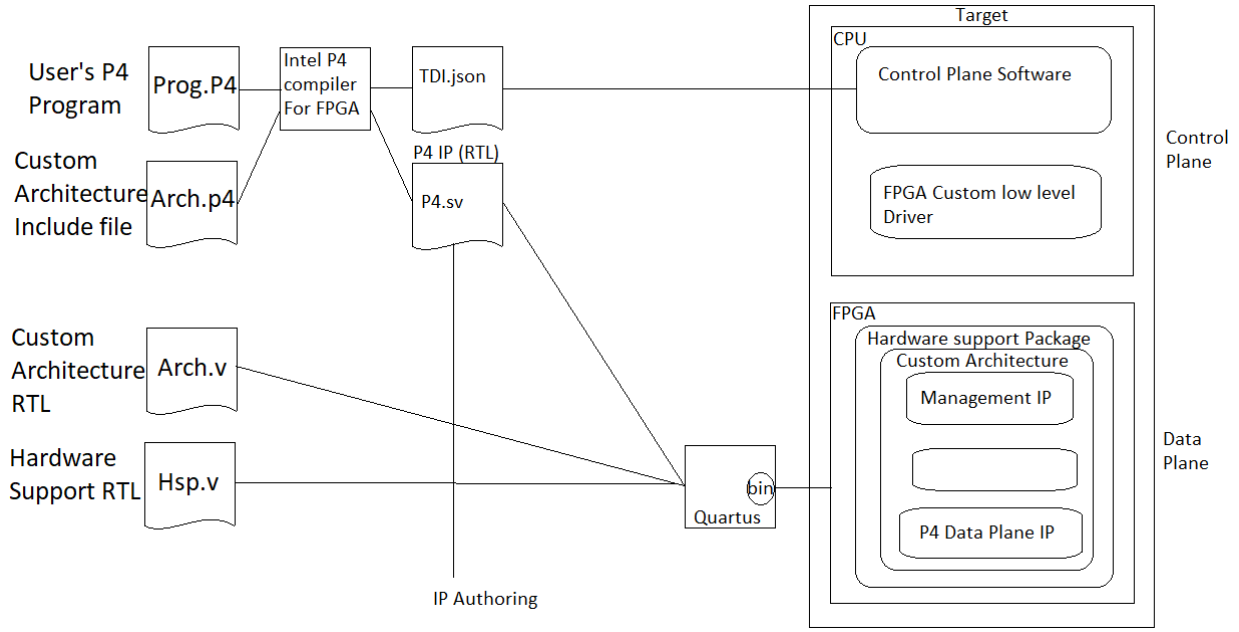

FIGURE 2.1 Le processus de compilation du programme P4 d'Intel jusqu'à déploiement de gauche à droite

Les compilateurs d'Intel et de Xilinx ont le pouvoir de prendre les bibliothèques à leur disposition pour traduire un code P4 en HDL en utilisant le passage par HLS comme pour le

projet eHDL [14]. Ceci permet de minimiser le temps de compilation ainsi que cette nouvelle façon de faire permet de réduire le stress d'écrire du code RTL. Dans ce projet, il est question d'utiliser l'architecture du FPGA Arria10 d'Intel (PAC N3000). Le compilateur d'Intel tente de faire un alignement simple pour le passage du code P4 à du code HDL [15]. La bonne structure de compilation du compilateur P4 d'Intel est visible à la figure 2.1. Le FPGA est la cible de l'implémentation pour le pipeline P4 une fois convertie en HDL après la synthèse Quartus.

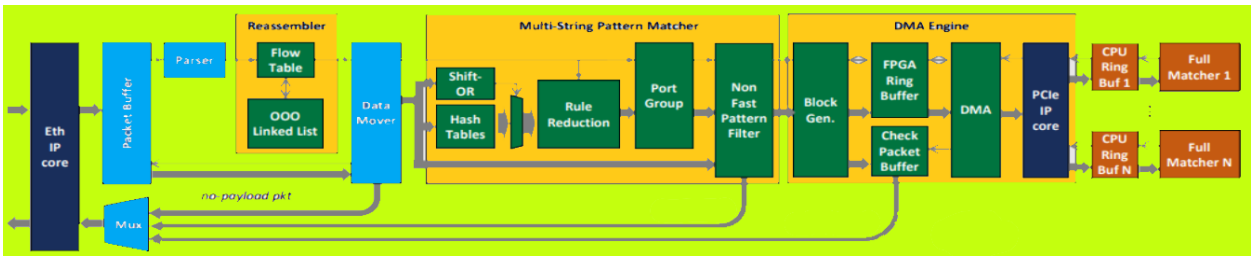

FIGURE 2.2 La distribution de charges au niveau du projet Pigasus

2.5 Interface pour le projet Pigasus

Les interfaces ont été implémentées avec Python dans le projet de Pigasus [2]. Dans ce projet, l'équipe a utilisé les templates Jinja2 comme à l'exemple du code 2.1 pour faire la génération automatique d'interfaces HDL comme VHDL et Verilog. Pigasus avait comme objectif de rendre le processus plus rapide pour coder les modules HDL de la détection de l'intrusion [16]. De plus, le projet utilise ses interfaces générées pour distribuer la charge d'un FPGA sur d'autres composants de la topologie comme un deuxième FPGA ou l'hôte de la communication du réseau (voir figure 2.2). Le projet a réussi à implémenter la logique de la détection d'intrusion à l'aide de Python et pouvait atteindre des vitesses atteignant 100 Gbps.

Listing 2.1 «Template» de Jinja2 en Python

```
1 Hallo {{ nom }}!
2
3 Dans le test d'aujourd'hui {{ nom_du_test }}.
4 Tu a obtenue {{ note }} parmit {{ note_max }} points.
5
6 Salut,
7 Rajat
8
9
```

```
10
11 from jinja2 import Environment, FileSystemLoader
12
13 note_max = 100
14 nom_du_test = "Externe en C++"
15 participants = [
16     {"nom": "Jean", "note": 100},
17     {"nom": "Fred", "note": 87},
18     {"nom": "Luis", "note": 92},
19 ]
20
21 environment = Environment(loader=FileSystemLoader("templates/"))
22 template = environment.get_template("message.txt")
23
24 for participants in participants:
25     filename = f"message_{participants['nom'].lower()}.txt"
26     content = template.render(
27         participants,
28         note_max=note_max,
29         nom_du_test=nom_du_test
30     )
31     with open(filename, mode="w", encoding="utf-8") as message:
32         message.write(content)
33         print(f"... wrote {filename}")
```

Pigasus a utilisé le Python pour toute sa plate-forme de conception et de génération de codes HDL ainsi que pour la simulation des différentes configurations de tests en utilisant le système de détection et de prévention des intrusions.

Le projet Pigasus est basé sur l'utilisation de l'architecture du FPGA Startix10. Ce FPGA est utilisé comme une première ligne de défense, en étant considéré comme une ressource de détection d'intrusion pour les premiers processus de filtrage, avant même l'intervention du CPU dans la méthode traditionnelle, comme illustré à la figure 2.2. Le partage de tâches entre le CPU et le FPGA est notamment viable dans d'autres applications. Le projet de [17] utilise la reconnaissance par automates finis déterministes (DFA) avec des performances élevées de prétraitement. Le projet de [18] permet la reconfiguration pour répondre aux demandes d'inférence de réseaux neuronaux profonds en temps réel sur un SmartNIC photonique-électronique. Plus proche de notre utilité est le projet de [19], qui traite du calcul hybride FPGA et CPU à l'aide de leur outil Spork («hybrid computing scheduler») pour exploiter les avantages en termes d'efficacité énergétique des FPGA pour ce type de charges de travail à un coût raisonnable. Par ailleurs, de nouvelles méthodologies permettent d'atteindre les objectifs de

télécommunications issus de la 5G [20].

Alors, les paquets entrent en contact avec le FPGA directement sans passer par le processeur d'hôte. Quoique, au besoin (selon le fonctionnement du réseau) ceux-ci sont dirigés vers un deuxième FPGA ou l'hôte. Il est aussi important de noter que cette transition est aussi possible pour des paquets qui n'ont pas été tout à fait décodés.

2.6 Synthèse de la littérature

Lors de cette section, il était question de faire une revue de la littérature pour le choix des outils de génération automatique de code HDL pour les interfaces (DMA, PCIe). La comparaison des outils de génération a été effectuée sur la performance et la facilité d'utilisation des outils comme MyHDL. D'ailleurs, le projet de référence Pigasus avait aussi été présenté dans ce chapitre. Pigasus est un projet d'extension de pipeline de détection d'intrusion utilisant le « Jinja template » de Python. Il faut savoir que l'outil est un bon choix pour générer du matériels pour FPGA [21]. Bref, nous verrons dans la prochaine section que l'utilisation du langage P4 est destinée à la génération de pipeline P4, mais peut aussi servir à déclarer une fonction externe pour étendre le pipeline P4.

CHAPITRE 3 ARCHITECTURE DE L'ENVIRONNEMENT DES INTERFACES

La mise en œuvre de certaines fonctionnalités dans les programmes P4 nécessite l'utilisation de modules appelés externes. De cette façon, le programme P4 du plan de données («Data Plane») devient plus efficace. Un FPGA peut avoir des périphériques comme d'autres FPGA. Les externes vont communiquer avec ces périphériques au travers d'interfaces. Alors, l'exécution de l'externe prend place sur d'autres FPGA ou processeurs. L'objet de ce mémoire est la génération automatique de ces interfaces. Une interface comme le PCIe permet une communication entre divers périphériques comme les FPGA, le CPU, etc. Ce chapitre traitera des thèmes suivants : les outils de développement pour le P4 offerts par Intel, la génération automatique d'interfaces avec PyXHDL, la compilation des interfaces, ainsi qu'une analyse de haut niveau des réseaux.

3.1 Pipeline P4 d'Intel

La conception des interfaces commence par l'écriture d'un programme P4 qui utilise des blocs fonctionnels, souvent appelés modules IP externes. C'est la fonctionnalité de cet externe qui peut être déployée sur un autre périphérique. Un système exploitant cette organisation est illustré à la figure 3.1. Le compilateur d'Intel propose un modèle différent de celui offert par le BMv2 («Behavioral Model version 2»). Une différence importante vient du fait qu'il accepte des IP externes. L'autre différence, c'est qu'il génère une description de matériel exprimée en HDL pour le programme P4 fourni par l'utilisateur. De plus, certaines différences importantes sont à la base de l'écriture de l'architecture du programme P4. Plus spécifiquement, dans les deux prochaines sections, il sera question de faire une analyse du code P4 d'Intel qu'il est essentiel de compiler pour faire fonctionner un externe avec l'interface générée.

3.1.1 Architecture du pipeline de traitement P4 d'Intel

Premièrement, pour qu'une interface soit compatible en matériel avec le programme d'utilisateur à la fin de la compilation, il faut un programme P4 compatible au compilateur d'Intel. Le programme P4 accepté par le compilateur d'Intel et celui accepté par le compilateur BMv2 ont aussi des similitudes comme les blocs P4 traditionnellement disponibles dans le langage. Par exemple, le programme P4 doit contenir les éléments essentiels pour le fonctionnement d'un pipeline de traitement de paquets («parser», «match&action», «deparser»). Le

programme P4 compatible au compilateur d'Intel est essentiellement le même que le P4$_{16}$ de la communauté P4 [4]. La principale différence est qu'il est maintenant possible d'y ajouter plus de composant comme des IP externes dans les blocs «action» et «table». Ceci facilite l'ajout de l'interface qui permet de faire des transferts tels qu'illustré à la figure 3.2. L'interface est un module matériel, donc écrite dans un langage comme le VHDL ou Verilog qu'il sera possible de connecter au pipeline P4.

Deuxièmement, pour qu'une interface soit compatible en matériel avec l'architecture P4 personnalisée, cette architecture doit être compatible au compilateur d'Intel. L'architecture P4 d'Intel doit être utilisée avec des outils de liaison des externes d'utilisateur exprimée par des annotations spécifiées via la syntaxe @. L'architecture personnalisée est une conception conventionnelle pour FPGA compatible au P4 et une couche d'adaptateur logiciel. Une fois que l'architecture personnalisée est créée, le système mis en œuvre dans un FPGA ressemble à un dispositif P4 programmable conventionnel. Le concepteur d'architecture P4 personnalisée a la même flexibilité qu'un concepteur qui développe du matériel pour un FPGA lorsqu'il est nécessaire de personnaliser les parties non programmables en P4 de la couche matérielle («Data Plane»).

Le concepteur d'architecture personnalisée peut instancier les IP du plan de données P4 sans connaître le code P4 source fourni par le concepteur de l'application. Cela fonctionne parce que le compilateur P4 d'Intel pour FPGA infère les noms de module IP du plan de données P4 à partir du fichier d'architecture personnalisée. La figure 3.1 montre un pipeline P4 qui accède à un module externe élaboré par un utilisateur à travers une interface dont la conception sera l'objet de la prochaine sous-section.

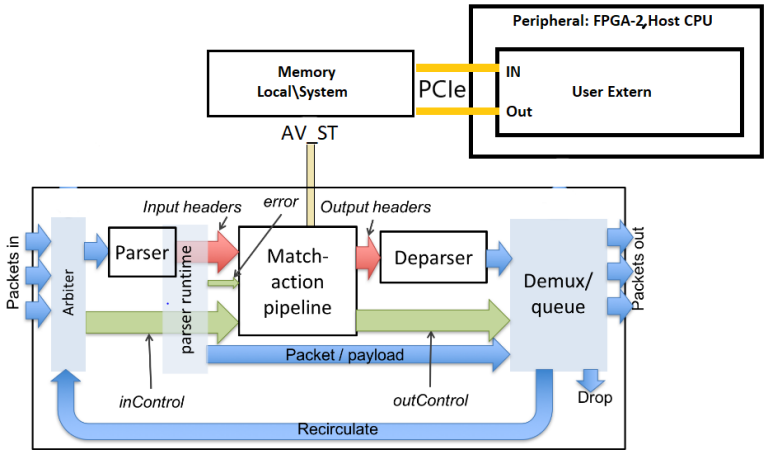

FIGURE 3.1 Fonctionnalités exprimées avec un module externe au pipeline P4

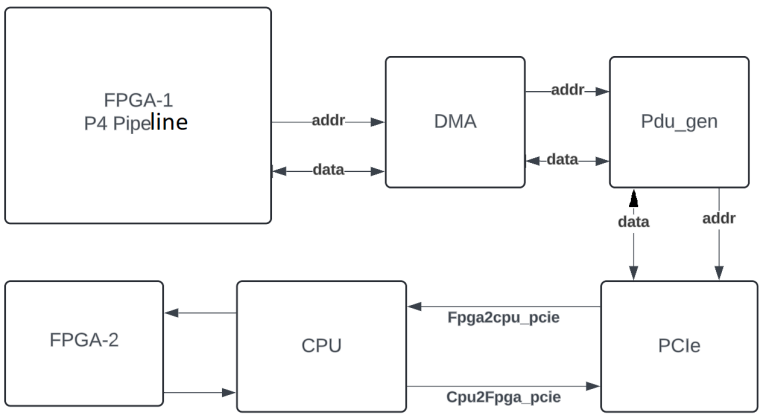

FIGURE 3.2 Génération du chemin jusqu'au CPU

La figure 3.1 met aussi en évidence le lien qui est utile pour faire fonctionner une interface avec un pipeline P4 et avec les périphériques exploitant la fonctionnalité de l'externe. L'interface prend son accès au pipeline principal à partir du bloc d'action, là où se trouvent les tables de comparaison P4. Comme mentionné, l'architecture joue le rôle de gérer les bonnes liaisons au pipeline P4. Une fois la compilation terminée, il est possible de faire le lien entre l'architecture et ses composants à l'aide d'un langage de haut niveau comme le Python.

Donc, ayant le programme P4 et l'architecture P4 personnalisée dans une bonne forme permet de faciliter l'utilisation du code matériel généré par le compilateur d'Intel pour le connecter à l'interface générée à l'aide de l'outil PyXHDL exploité dans le présent projet.

Un important élément d'originalité du présent projet réside dans l'utilisation d'un programme P4 avec du code matériel généré automatiquement par un langage de haut niveau (Python et HLS). Ce mémoire introduit des extensions à un programme P4 applicable dans le contexte du compilateur d'Intel pour les SmartNIC utilisant les pipelines P4.

3.2 Génération automatique d'interfaces avec PyXHDL

L'outil PyXHDL génère du code HDL compatible avec divers langages de description de matériel cibles populaires tel que le VHDL, le Verilog, le SystemVerilog. Il est à noter que les sections précédentes ont permis d'identifier l'outil PyXHDL comme un outil flexible (section 2.2 du chapitre précédent) et facile d'utilisation à cause de sa syntaxe facile à comprendre. La programmation d'une interface est effectuée à l'aide des classes héritant de celles de l'outil PyXHDL introduit dans le code 3.1 par exemple.

La figure 3.3 montre le chemin des ressources que parcourt l'ensemble de données de ce

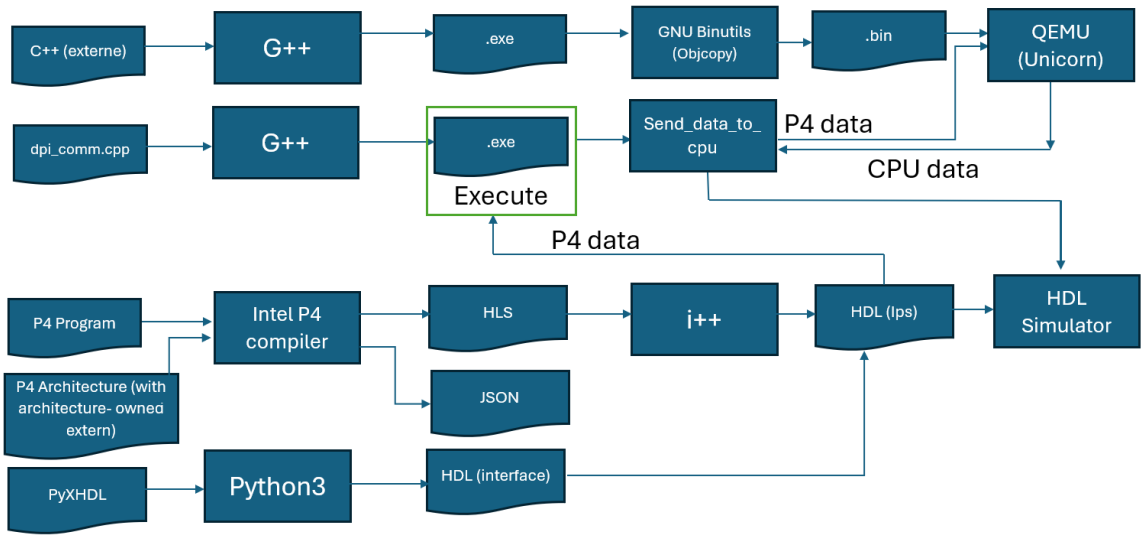

FIGURE 3.3 Flot de données complet pour l'implémentation et le test de solutions en P4

projet. La propagation se fait, normalement, de gauche vers la droite et plus spécifiquement selon le sens des flèches. Les blocs courbés représentent les données et les blocs rectangulaires représentent les processus. D'abord, il est possible de voir le programme P4, l'architecture P4 et l'outil PyXHDL pour la génération d'interface. En effet, cette partie de bas de la figure 3.3 montre l'implémentation des interfaces et l'emplacement de l'interconnectivité (bloc HDL compilé à travers le code HLS) avec l'interface. La partie du haut montre le cas de la validation de l'interface à l'aide du code DPI-C (dpi_comm.cpp), qui fait le lien entre l'émulateur exécutant l'externe et la simulation RTL.

3.2.1 Module DMA

La toute première étape du processus proposé pour la génération d'interface entre un programme P4 et ses externes, effectuée avec l'outil PyXHDL, exploite le code HDL pour un module DMA cité plus tôt (listing 3.1).

Cette approche se distingue de celle adoptée dans Pigasus [2] que l'on utilise comme projet de référence. C'est une méthode qui se distingue de la génération de code HDL avec «Jinja Template», car l'utilisateur peut aussi faire des modifications à haut niveau en Python.

De cette façon, il est possible de voir le passage d'un code Python vers un code matériel Verilog. Donc, l'utilisateur peut comprendre la provenance des générations du code de ce projet ce qui est aussi utile pour les concepteurs électronique. Pigasus utilise la génération d'IP pour ses pipelines en HDL à travers les «Jinja Template» de Python et utilise les modules

DMA et PCIe comme interface entre ses périphériques (FPGA, hôte). Cette solution permet de faire un accès rapide aux données des ressources externes (calculs C++) à Pigasus.

L'interface générée automatiquement dans ce projet est le DMA et le PCIe, illustrés à la figure 3.2 par le bloc mémoire et les deux canaux jaunes respectivement. De plus, le code HLS permet de générer un «Avalon Stream» qui est directement lié au DMA (bloc mémoire), et qui sera discuté à la section 3.3.3. Compte tenu du caractère central du module DMA cité pour la méthode de génération d'interfaces proposée, il est important de bien le comprendre. Le module DMA, visible à la figure 3.2 est relié à la mémoire. Ceci favorise un échange de données avec le CPU sans l'alourdir. Le module DMA fait le pont entre le pipeline de traitement P4 et les périphériques liés à l'aide de l'interface PCIe. Cette dernière peut faire des liens selon l'adresse attribuée (pour le FPGA et le CPU). La flexibilité offerte par ce modèle vient notamment du fait que les bus PCIe sont connectés avec la mémoire vers laquelle pointe le DMA.

Le code fourni au listing 3.1 illustre un cas où le générateur de PyXHDL génère le code RTL pour le module DMA. Cet exemple est une des éléments clés du présent projet. Ce module écrit en Python est directement lié avec le compilateur HDL d'Intel. Donc, la génération d'interface effectuée de cette façon doit simplifier la création de ponts entre le langage Python et le langage matériel. Le langage matériel dans lequel le module DMA sera généré est le Verilog. Ce choix est principalement dû à l'intégration aux modules IPs Verilog déjà disponible après la compilation du programme P4 avec le compilateur d'Intel. Il est à noter que la génération automatique des interfaces dans ce projet est pleinement généralisable pour des applications de transferts de paquets. L'interface peut avoir un usage plus spécifique comme pour la détection d'instruction (Pigasus [2]). Alors, ces interfaces sont personnalisées pour l'application du transfert de données respective. En effet, l'interface sera utile pour la connexion à l'externe généré par le programme P4 d'Intel fourni par l'utilisateur.

Avant d'aller dans les détails de la conception du module DMA à l'aide de l'outil PyXHDL, il est important de mentionner que le code matériel (HDL) généré par cet outil est synthétisable et multi-usage. Un utilisateur peut apporter des modifications au code Python de l'outil PyXHDL et le code sera toujours synthétisable. Il est donc possible de faire des changements sur les ports d'entrée ou de sortie ainsi que sur différentes variables, tout en maintenant la compatibilité du code pour diverses applications. Il faudra également modifier les autres modules dépendants dans le pipeline P4 (ou dans toute autre application) pour assurer la compatibilité. C'est pourquoi le protocole de communication PCIe-DMA est universel à travers différents systèmes embarqués, en réseau ou autre.

Listing 3.1 Classe de PyXHDL pour l'implémentation des modules d'interface HDL

```
class DMA(X.Entity):
    PORTS = 'Clk, Rst_n, =pcie_rb_wr_data, =pcie_rb_wr_addr,
       =pcie_rb_wr_en, \
    pcie_rb_wr_base_addr, pcie_rb_almost_full, =pcie_rb_update_valid,
       =pcie_rb_update_size, \
    disable_pcie, pdumeta_cpu_data, pdumeta_cpu_valid,
       pdumeta_cpu_ready, \
    =pdumeta_cpu_csr_readdata, =ddr_wr_req_data, =ddr_wr_req_valid,
       ddr_wr_req_almost_full, \
    =ddr_rd_req_data, =ddr_rd_req_valid, ddr_rd_req_almost_full,
       ddr_rd_resp_data, \
    ddr_rd_resp_valid, =ddr_rd_resp_almost_full, \
    in_pkt, in_meta, in_usr, nomatch_pkt'

    @X.hdl_process(kind=X.ROOT_PROCESS)
    def run():

        lis = []
        lis2 = []
        for i in range(6):
            lis.append(vs.mkwire(ExternIP_channel_if
            (_P=dict(WIDTH=512)), "channel_if_"+str(i+1)))

        for i in range (2):
            lis2.append(vs.mkwire(ExternIP_channel_if(
            _P=dict(WIDTH=388)), "channel_if_"+str(i+7)))

        A = Clk
        QQ = vs.mkwire(tp.Uint(1))
        ...
        ExternIP_unified_fifo(in_clk = A,
            in_reset          = B,
            out_clk           = EEE,
            out_reset         = EEE,
            in_data           = pdumeta_cpu_data,
            in_valid          = pdumeta_cpu_valid,
            in_ready          = pdumeta_cpu_ready,
            out_data          = MM,
            out_valid         = NN,
            out_ready         = OO,
            fill_level        = PP,
            almost_full       = EEE,
```

```
38              overflow          = EEE,
39              _P=dict(FIFO_NAME        = name1,
40                      MEM_TYPE         = "M20K",
41                      DUAL_CLOCK       = 0,
42                      USE_ALMOST_FULL  = 0,
43                      FULL_LEVEL       = 450,
44                      SYMBOLS_PER_BEAT = 1,
45                      BITS_PER_SYMBOL  = 28,
46                      FIFO_DEPTH       = 512))
```

Le module DMA conçu est présenté au listing 3.1. Dans ce listing, on remarque que la classe «class DMA(X.Entity)» hérite de la classe X.Entity à la ligne 1. Le moule X.Entity fait appel à la classe «Entity» qui hérite de la classe «_CoreEntity» contenant la composition complète de code d'une entité matérielle comme la déclaration des ports d'entrée et de sortie. Une fois qu'une classe comme la DMA est déclarée dans le code PyXHDL, il est important d'y ajouter les ports et pour ce faire, il faut utiliser la variable de classe «PORTS» en majuscules et d'y attribuer les noms des ports comme à la ligne 2.

D'ailleurs, il faut aussi faire ce type de déclaration pour les modules qui seront instanciés dans notre module principal comme dans le cas du module «ExternIP_unified_fifo» à la ligne 25 du listing 3.1. Il est notable que le code Python reste le même, peu importe le choix de langage précisé plus tard pour la compilation (le VHDL ou le Verilog). La compilation produit la définition suivante pour l'entête du module DMA en Verilog : «module DMA(Clk, Rst_n, pcie_rb_wr_data, pcie_rb_wr_addr, pcie_rb_wr_en, ...) ; input logic [0 : 0] Clk ; input logic [0 : 0] Rst_n ; output logic [513 : 0] pcie_rb_wr_data ; output logic [11 : 0] pcie_rb_wr_addr ; output logic [0 : 0] pcie_rb_wr_en ; ...»

La logique principale produite par PyXHDL est combinatoire. Cette logique effectue des assignations en continu. La logique combinatoire n'est pas bloquante, mais parallèle. Ces circuits sont tous activés en parallèle. Dans un cas réel, ceci conduit à des assignations en continu (au changement d'état de la partie de droite, il y a changement immédiatement à la partie de gauche.) Pour le cas du Verilog, les assignations sont avec le «assign» dans le corps du module en définition. Sachant que l'outil peut générer du Verilog ou du VHDL à partir d'un même code Python (PyXHDL), il faut donc respecter le code Python compatible aux deux langages.

L'appel du processus «@X.hdl_process(kind=X.ROOT_PROCESS)» dans le code de la ligne 10 du listing précédent (listing 3.1) instancie la logique combinatoire du code matériel implémenté pour le module DMA. De cette ligne de code, on peut faire ressortir l'utilisation du décorateur de Python avec le nom X.hdl_process. Il est bien connu que les décorateurs

agissent pour faire de la transformation sur le code d'une méthode ou d'une classe.

Donc, la fonction sous un décorateur en Python entre en paramètre à la fonction décorateur et cette fonction décorateur modifie la sortie ou l'entrée de la fonction décorée (agis sur son comportement). Ceci ce fait de la manière suivante :
@decorateur_avec_arguments
def décorée(ar1, ar2)
En effet, la fonction «decorateur_avec_arguments» avec le symbole «@» se place au-dessus d'une fonction qu'il veut utiliser pour modifier ses entrées et ses sorties. Les arguments «ar1» et «ar2» sont tous les deux des entrées de la fonction «décorée».

Cet exemple met en lumière le fonctionnement du décorateur «@X.hdl_process()» de l'outil PyXHDL. Regardons comment utiliser ce décorateur.

Certains tests avec l'outil PyXHDL montrent qu'il est possible de générer plusieurs types de routines séquentielles et combinatoires en HDL. Dans le listing 3.1 on utilise, à la ligne 10, la variable «kind» qui est un paramètre du décorateur «@X.hdl_process» et le «kind» établit le type de processus à générer. Le «kind» sert au placement du processus parmi différentes catégories comme le «ROOT_PROCESS» et le «INIT_PROCESS».

Premièrement, il est possible de remarquer que la logique combinatoire est générée avec l'utilisation du «kind» de type «ROOT_PROCESS» comme au cas 1 et au cas 2 du tableau 3.1. Deuxièmement, il est visible au cas 3 et au cas 6 du tableau 3.1 que l'outil PyXHDL génère du code séquentiel (sans l'utilisation du paramètre «kind»). Le cas 4 du tableau 3.1 ne compile pas avec le générateur de l'outil PyXHDL parce qu'il n'est pas possible d'avoir un HDL valide pour le décorateur «@X.hdl_process(sens=clk, kind=X.INIT_PROCESS)». Par conséquent, le cas 5 est possible et génère du code séquentiel avec le bloc «Initial begin» utile lors de la simulation avec des délais assignations comme le cas 6 qui lui génèrerait aussi une logique séquentielle. La routine générée par le cas 6 permet au bloc «always» d'avoir du délai comme «always #10 A = B» (aussi utile en simulation). D'ailleurs, «sens» est une autre variable en paramètre d'entrée pour «@X.hdl_process()» qui ajoute la liste de sensibilité à la routine séquentielle du langage matériel qui sera générée. Donc, le cas 1 utilise les deux paramètres du décorateur «@X.hdl_process» soit le «kind» et le «sens», mais génère un code combinatoire (le paramètre «kind» est considéré plus significatif que le paramètre «sens»). Par conséquent, le cas 1 du tableau 3.1 a le même effet qu'au cas 2. Si le «@X.hdl_process()» était sans le «kind», donc «@X.hdl_process(sens='refclk_clk')» la logique séquentielle serait générée comme au cas 3 du tableau 3.1. Le «sens» sert à prendre le nom des signaux qui seront considérés comme sensibles (liste de sensibilité) pour exécuter la routine séquentielle. Alors, l'exécution va réveiller la routine selon le changement d'état du signal sensible. Cette

routine est une logique contenant de la mémoire (logique procédurale). Elle effectue donc des assignations bloquantes (sans symbole «<=») là où les signaux devront conserver leur valeur précédente avant l'arrivée du front, produisant un changement d'état d'un signal de la liste de sensibilité.

Tableau 3.1 Génération de la logique combinatoire et de la logique séquentielle à partir de l'outil PyXHDL

Cas	Décorateur «@X.hdl_process()»	Code Verilog généré après la compilation
Cas 1	@X.hdl_process(sens=clk, kind=X.ROOT_PROCESS)	Logique combinatoire sans le bloc «always@(clk)», donc assignation continue
Cas 2	@X.hdl_process(kind=X.ROOT_PROCESS)	Logique combinatoire sans le bloc «always@(clk)», donc assignation continue
Cas 3	@X.hdl_process(sens=clk)	Logique séquentielle avec le bloc «always@(clk)» et avec le signal de sensibilité, donc assignation bloquantes (sans symbole «<=»)
Cas 4	@X.hdl_process(sens=clk, kind=X.INIT_PROCESS)	Pas de génération du code (Erreur)
Cas 5	@X.hdl_process(kind=X.INIT_PROCESS)	Logique séquentielle avec le bloc «Initial begin», donc cette assignation est bloquante (sans symbole «<=»)
Cas 6	@X.hdl_process()	Logique séquentielle avec un bloc «always», mais sans le signal de sensibilité, donc cette assignation est également bloquante (sans symbole «<=»)

Un module matériel peut être instancié dans le module matériel principal, mais à partir du code Python de l'outil PyXHDL. Pour ce faire, le module est instancié directement comme une instanciation de classe en Python avec son nom. Dans cette instance, il faut d'abord assigner à ses variables en paramètres comme le «in_clk» une variable initialisée comme «A» («A = Clk» à la ligne 23 du listing 3.1). Par la suite, il faut utiliser «_P» comme paramètre de la liste des paramètres de l'instance pour assigner des variables aux variables globales du module matériel. Donc, une variable globale sert de constante pour certaines déclarations de ports ou des signaux en HDL. Le «_P» s'assigne un dictionnaire avec toutes les variables globales que le module aurait de besoin. Alors, la ligne 39 à 46 du listing 3.1 résulte en une instance en Verilog comme la suivante : «unified_fifo #(.FIFO_NAME("[top] pdumeta_cpu_FIFO"),

.MEM_TYPE("M20K"), .DUAL_CLOCK(0), ...) unified_fifo_1(.in_clk(Clk),

.in_reset(Rst_n), ...)»

Les signaux comme «QQ = vs.mkwire(tp.Uint(1))» sont initialisés en PyXHDL où le «vs» est le module «vars» de l'outil PyXHDL qui est aussi importé dans «pyxhdl» et «mkwire» est la fonction «make wire» qui retourne une instance de signal de type «tp.Uint(1)». Ce qui veut dire «type unsigned integer» d'un seul bit. Le signal «QQ = vs.mkwire(tp.Uint(1))» résulte dans un signal en Verilog comme «wire logic [0 : 0] QQ ;». La liste des signaux continue aux trois petits points à la ligne 25. Les signaux sont tous génériques et utilisables pour des FPGA et l'hôte comme pour le projet Pigasus. L'utilisateur peut utiliser le module DMA, une fois généré par PyXHDL, comme une interface entre la mémoire et le module PCIe. En somme, la figure 3.1 montre une relation entre un module DMA et une bus d'interface PCIe. Dans la prochaine sous-section, le module PCIe est décrit. PCIe est l'interface qui est aussi générée par l'outil PyXHDL dans ce projet. PCIe fait l'interface entre l'externe et le pipeline P4 généré par le programme P4.

3.2.2 Module PCIe

L'utilisation du module DMA est indispensable pour le transfert de données d'un périphérique à l'autre. Cette technique est beaucoup utilisée dans les systèmes embarqués comme les microcontrôleurs et FPGA [13]. Ceci permet au CPU de déléguer la gestion de la mémoire au système de DMA. Un module PCIe a également été implémenté avec l'outil PyXHDL. Il est exprimé avec un code Python semblable à celle du module DMA (comme au listing 3.1). Ce dernier comprenait des instanciations comme le module matériel channel_if «channel interface» qui est utile pour les interfaces de simulation. Ce canal est utilisé avec le type «Modport» de deux catégories «tx» et «rx» (transmission et réception).

Le module PCIe fait appel à plusieurs modules comme le module fpga2cpu_pcie, le module cpu2fpga_pcie et le module pcie_ed. Le module pcie_ed a pour objectif de faire un transfert ou une écriture des données sur le bus PCIe de la carte. Ce module est souvent spécifique au FPGA en question dans notre cas c'est pour un Arria10 de la série 10AX115H1F34I1SG. Donc, il faut configurer les ports pour spécifier leurs caractéristiques de transferts dans «pcie_ed.tcl». Le module cpu2fpga_pcie est le module qui sert à prendre la réponse du CPU en lien avec la demande que l'utilisateur lui transmet à l'aide du module fpga2cpu_pcie en utilisant les ports d'entrée wr_data, wr_addr et wr_en principalement tout en lisant l'adresse du port de sortie wr_base_addr.

Le module de PCIe est composé de plusieurs routines séquentielles qui font que les échanges de valeurs sont bloquants (sans symbole «<=»). Ces échanges se font de manière procédurale.

Les routines sont écrites avec l'annotation «@X.hdl_process(sens='refclk_clk')» avec l'outil PyXHDL (comme au cas 3 du tableau 3.1), ce qui résulte en une génération de code matérielle séquentielle comme pour la routine pdu_buffer dans le listing 3.2. Cette routine est sensible au changement d'état du signal d'horloge «refclk_clk» et en observant le bit à la position sept du signal «address_0» (ligne quatre du listing 3.2) on effectue une concaténation des signaux de registres PCIe (48 à 63) pour l'attribution au «readdata_0_» (ligne six du listing) qui lui est attribué avec la logique combinatoire à l'entrée du module «PCIe_ed».

Listing 3.2 Une des Routines séquentielle générée par le code PyXHDL du module PCIe

```
1  always @(refclk_clk)
2    pdu_buffer : begin
3      if (&cpu_reg_region_r2) begin
4        case (address_0[6: 6])
5          unsigned'(1'(3)):
6            readdata_0_ = {{{{{{{{{{{{{{{{pcie_reg63_pcie,
                 pcie_reg62_pcie}, pcie_reg61_pcie}, pcie_reg60_pcie},
                 pcie_reg59_pcie},
7            pcie_reg58_pcie}, pcie_reg57_pcie}, pcie_reg56_pcie},
                 pcie_reg55_pcie}, pcie_reg54_pcie}, pcie_reg53_pcie},
8            pcie_reg52_pcie}, pcie_reg51_pcie}, pcie_reg50_pcie},
                 pcie_reg49_pcie}, pcie_reg48_pcie};
9        endcase
10       readdatavalid_0_ = read_0_r2;
11     end else begin
12       readdata_0_ = frb_readdata;
13       readdatavalid_0_ = frb_readvalid;
14     end
15   end
```

Dans le module PCIe, à plusieurs endroits, on retrouve des routines qui font une réassignation du signal d'entrée à un autre signal local comme ceci :
always @(refclk_clk)
clock_crossing_to_jtag1 : begin
pcie_reg64_r1_ = pcie_reg64_status ;
pcie_reg64_pcie_ = pcie_reg64_r1 ;
end.
En effet, le temps de stabilité de la valeur lue est plus long, donc en matériel il y a ajout d'un «Flip-Flop» pour stabiliser cette donnée. D'ailleurs, les routines séquentielles sont nécessaires dans le cas de changement de domaine d'horloge, car elles permettent à la valeur d'intérêt de traverser d'un domaine dans lequel l'horloge est plus rapide vers un domaine dans lequel

l'horloge est moins rapide. Il existe plusieurs autres méthodes pour stabiliser une donnée pour en permettre le passage d'un domaine d'horloge à l'autre comme l'étirement de l'horloge, l'ajout de FIFO, etc.

3.3 Compilation des Interfaces

La compilation de l'interface se fait dans l'outil Python PyXHDL. Une fois que le module est complété avec les instanciations et les assignations nécessaires, il est temps de faire la compilation de ce code Python en code HDL (matériel). Pour ce faire, on utilise l'instance suivante : «emitter = X.Emitter.create(args.backend)» pour créer un objet venant du module «Emitter» importé dans le module «pyxhdl» qui utilise le décorateur @classmethod de la fonction «create».

Les paramètres qui sont envoyés à la fonction «create» sont les arguments comme le type de générateur «–backend» à utiliser (dans notre cas, c'est le Verilog), la description (une phrase en «string») et la taille des données (lue à partir du data du code P4) qui sera associée au paramètre de l'externe «–p4_data_size». Après, on prépare la génération du code matériel avec la création de l'objet codegen «codegen = X.CodeGen(emitter, X.create_globals(PCIE, source_globals=globals()))» qui lui est une instance de la classe CodeGen du module importé «pyxhdl». Cette classe est composée de plusieurs des fonctions comme generate_entity, generate_process, run_code, load_var, assign_value, _handle_array, _get_sensitivity, etc. Dans le cas de ce projet, on utilise une des fonctions énumérées «codegen.generate_entity(ar1, ar2)» là où l'argument un (ar1) est le nom du module principal et l'argument deux (ar2) est un dictionnaire qui contient la liste des ports qui ont été assignés avec le bon nombre de bits par exemple «dict(refclk_clk=X.mkwire(X.Uint(1)), ...)», donc le port refclk_clk est assigné à un «wire» d'un seul bit.

De plus, l'émetteur fait appel à un émetteur de code matériel (soit en VHDL ou en Verilog) selon le paramètre «–backend». Dans le cas de ce projet, le choix était le Verilog, donc l'émetteur appelle le module verilog_emitter qui fait toute la génération du code d'interface en Verilog. En analysant le code Python de l'outil PyXHDL, il est possible de faire certaines modifications pour faciliter la génération des instances de modules d'interfaces. Le code source présente les classes et les fonctions de l'outil PyXHDL qu'il est possible de modifier. Les prochains paragraphes vont montrer les modifications dans le code original de l'outil PyXHDL au niveau du module verilog_emitter qui nous ont permis de faire l'instanciation de notre module principal dans le code matériel généré par le compilateur P4 simultanément. Le but était de faire ressortir tous les ports d'entrée et de sortie que le PCIe et le DMA utilisent (pour les transformer en signaux locaux). Un module généré à travers l'outil PyXHDL

est écrit en langage HDL et il contient des ports d'entrée et de sortie. Pour qu'un autre module HDL instancie ce module, il faut utiliser une syntaxe d'instanciation comme celle générée par le code Python pour la fonction «ExternIP_unified_fifo» (ligne 26 du listing 3.1). Donc, si on génère un module DMA écrit en Python, il nous faut lire les ports de ce module et générer ses ports comme des signaux pour les instancier dans un des fichiers généré par le programme P4 comme le «fonction_wrapper.sv». Par la suite, il faut aussi créer un appel à ce module dans le fichier «fonction_wrapper.sv», donc une fois l'instance crée dans un fichier à part, il est possible de l'utiliser comme un exemple de gabarit d'instanciation. Précisément dans ces modifications, les deux modules (PCIe et DMA) ont été analysés pour faire ressortir leurs ports d'entrées et de sorties dans un fichier en format texte. Dans la classe verilog_emitter les modifications ont été apportées à la fonction emit_module_decl, car c'est la fonction responsable de faire la génération de chacun des signaux de port des modules matériels générés par PyXHDL. L'instanciation a été faite en lisant chaque nom de port soit d'entrée ou de sortie et faisant un signal «wire» de chacun en Verilog. Le signal crée a le même nom que le port original dans le but de l'instancier en Verilog par exemple ceci : «.pcie_rb_wr_data(pcie_rb_wr_data)». Alors, les modifications génèrent le signal «wire» pour ce port comme «wire logic[513 : 0] pcie_rb_wr_data». D'ailleurs, certains des ports n'ont pas de type «logic» parce qu'ils sont de taille unitaire (vecteurs de taille unitaire).

Alors, de la fonction emit_module_decl le fichier texte (interface.txt) sera généré comme gabarit d'instanciation contenant un modèle pour les deux modules matériels PCIe et DMA avec tous les signaux des ports d'entrée et de sortie ainsi que leur appel. Ce fichier sera lu par la suite par le code Python principal qui fait la génération, la compilation et l'instanciation de tout le système de ce projet. Le code Python principal a été développé pour faciliter la compilation et la génération de tous les documents nécessaires au projet P4. Ce fichier était un outil d'Intel venant avec le nouveau compilateur P4 ($P4_{16}$) en 2023. Ces fonctionnalités seront l'objet de la prochaine sous-section.

3.3.1 Compilation du code de l'externe C++

Le fichier de l'externe est ajouté au code Python qui fait la compilation du programme P4. Le script fait sa compilation en utilisant la commande «g++» comme dans le cas suivant : «g++ -m32 -o add add.cpp -nostdlib -fno-pie -no-pie -Wl,–section-start=.text=0x01000000». Les paramètres sont utilisés pour favoriser la bonne compilation du code C++ du fichier d'externe (add.cpp) et de générer un exécutable avec l'assignation -o et le nom du fichier exécutable (add). L'autre paramètre de la commande «g++», le -m32 faits la compilation pour un environnement de 32 bits. Le paramètre -nostdlib précise de ne pas utiliser les

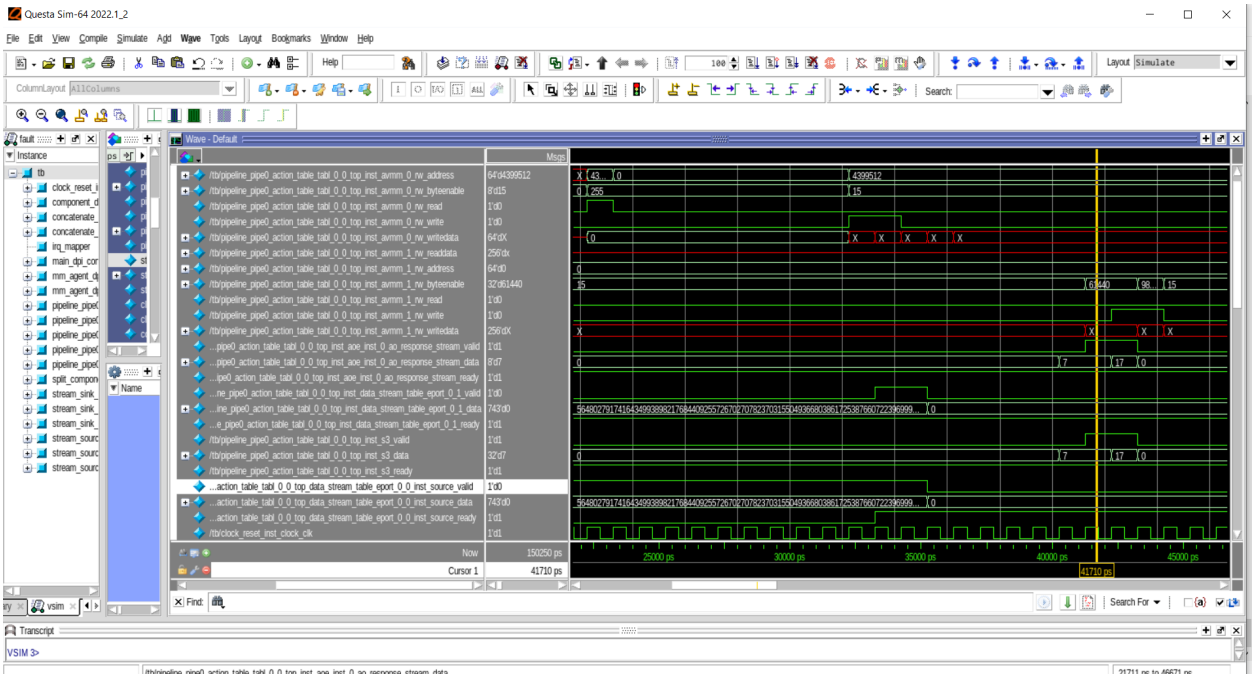

FIGURE 3.4 Lecture de données par une application P4 arrivant d'un module externe

fichiers ou bibliothèques de démarrage de système standard lors de la liaison. Donc, aucun fichier de démarrage, mais seules les bibliothèques spécifiées sont transmises à l'éditeur de liens. La commande utilise aussi le paramètre -no-pie (exécutable indépendant de la position (PIE)). Ceci est une condition préalable pour ne pas activer la randomisation de la disposition de l'espace d'adressage. Le paramètre -Wl,–section-start=.text=0x01000000 est utilisé pour permettre une adresse de départ précise à l'exécutable du code add.cpp. Dans ce cas, l'adresse de base est de 0x01000000, donc comme visible sur la figure 3.5, la zone texte débute par l'adresse 0x01000000. D'ailleurs, le code de l'externe peut être différent et sa commande peut être modifiée dans le code Python.

3.3.2 Les bits de l'externe P4

La taille de l'ensemble de données (cette taille inclut la taille des autres paramètres qui accompagne le paramètre «data») a été évaluée et extraite dans le script Python à l'aide d'une lecture du ficher d'architecture P4 comme dans le code 1.2. Pour ce faire, le script fait une recherche de la taille de l'ensemble de données en regardant toutes les valeurs des paramètres des externes (si plus qu'une fonction externe) et prend la plus grande valeur cumulative de tous les externes disponibles. En effet, une addition est faite pour chacune des valeurs de bits suivantes : les paramètres du premier externe «in bit<256> data, in bit<2>

```
SYMBOL TABLE:
01000000 l    d  .text  00000000 .text
01001000 l    d  .eh_frame_hdr  00000000 .eh_frame_hdr
0100101c l    d  .eh_frame      00000000 .eh_frame
01003000 l    d  .bss   00000000 .bss
00000000 l    d  .comment       00000000 .comment
00000000 l    df *ABS*  00000000 add.cpp
00000000 l    df *ABS*  00000000
01001000 l       .eh_frame_hdr  00000000 __GNU_EH_FRAME_HDR
01000000 g     F .text  00000010 _Z6init_cv
00000000         *UND*  00000000 _start
01003000 g     O .bss   00000004 c
01000010 g     F .text  00000025 mul
01003000 g       .bss   00000000 __bss_start
01003000 g       .bss   00000000 _edata
01003004 g       .bss   00000000 _end
```

FIGURE 3.5 Symbole de l'exécutable de l'externe C++ de l'utilisateur

iface, in bit<3> op» et les paramètres du deuxième externe «in bit<256> data, in bit<4> iface, in bit<3> op». La taille sera considérée à une valeur de 263 bits provenant de l'externe deux, qui est plus grande que les 261 bits de l'externe un. Cette taille binaire sera, par la suite, attribuée à la compilation du code d'interface en PyXHDL. Le but est d'avoir un bon ajustement des ports des interfaces générées.

3.3.3 Le code HLS pour la table d'action P4

Les fichiers HLS sont synthétisés à l'aide du compilateur HLS [22]. Il a été nécessaire d'ajouter quelques lignes de code pour personnaliser le code HLS avant de le compiler en HDL avec le compilateur «i++». Cela permet d'accéder aux paramètres d'externes P4 via le flux de sortie du composant HLS («Avalon Stream» décrit à la section suivante). D'ailleurs, le compilateur de programme P4 d'Intel a généré les fichiers HLS à partir des tables P4 de l'utilisateur. L'externe de l'utilisateur se trouve dans un composant HLS qui comporte deux tâches de lancement. Ces tâches sont de réponse et de requête implémentées pour s'exécuter en parallèle afin de faciliter l'échange de données via des flux de données («Avalon Stream»). La première tâche consiste à collecter les données d'utilisateur auprès du tableau de l'utilisateur contenant l'externe (tâche requête). C'est cette tâche qu'il va falloir configurer à travers le fichier de haut niveau «wrapper.vhd» pour récolter la donnée de l'externe. La deuxième tâche récupère la réponse de l'externe si l'utilisateur le demande dans le programme P4. Ceci constitue une

limitation actuelle du compilateur. En effet, le message que produit le compilateur P4 d'Intel lors d'une requête d'une valeur de retour par une fonction externe est présenté à la figure 3.8. Le commentaire «Call-response architecture-owned externs not supported» montre que la lecture de retour n'est présentement pas supportée.

La solution à ce problème est l'utilisation de l'interface générée automatiquement, car celle-ci peut être personnalisée pour avoir une valeur de retour.

3.3.4 Avalon Stream

Le bus Avalon Streaming est une interface prenant en charge un flux de données et est configurable via HLS, comme décrit dans la section précédente. Traditionnellement, Les données sont transférées de la Source au Drain sous le contrôle de plusieurs signaux. Le modèle standard de ce bus comprend trois signaux de commande principaux, tandis que le mode paquet ajoute des signaux supplémentaires pour identifier des éléments clés des paquets transmis, tels que le début de paquet (SOP) et la fin de paquet (EOP).

Les détails de cette interface sont illustrés à la figure 3.7. Le signal facultatif «empty» indique le nombre d'octets vides dans le dernier cycle du paquet. Deux paramètres, «readyLatency» et «readyAllowance», permettent de configurer la synchronisation : dans le cadre de ce mémoire, ces deux paramètres sont mis à zéro, signifiant que les transferts de données se produisent uniquement lorsque les signaux «ready» et «valid» sont actifs.

La 3.6 présente le chronogramme de l'interface Avalon Streaming standard.

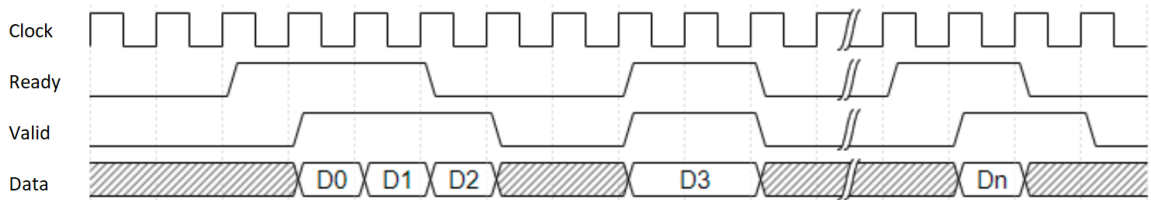

FIGURE 3.6 Le Chronogramme du fonctionnement d'un Avalon Stream

3.3.5 La génération de l'interface

L'ajout d'un composant matériel dans le fichier choisi, le fichier «function_wrapper.sv», se fait à l'aide du script Python de l'outil d'Intel qui a été modifié pour qu'il utilise la méthode read_instances(). Cette méthode permet de faire la lecture du fichier texte qui avait été généré par l'outil PyXHDL (interface.txt) et de composer une table pour tous les signaux de

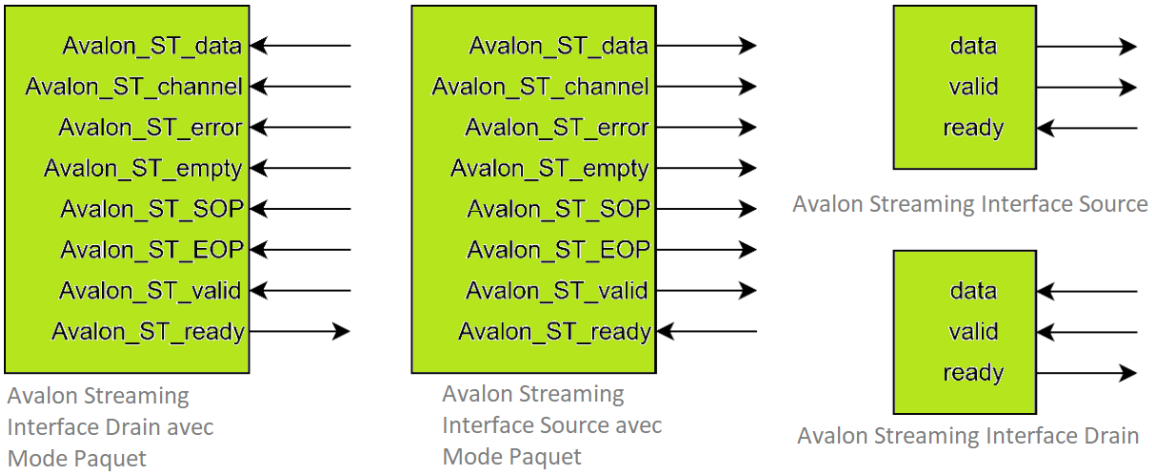

FIGURE 3.7 Les ports importants de l'Avalon Stream d'Intel

```
 This is the p4 file
 test_aoeip.p4
arch_aoeip.p4

 this is the data/buffer for hls out s3 stream:  263 , 2
[debug] Execute:
  /usr/packages/p4env/intel/intelfpga_p4c/bin/p4c-fpga-rtl-generator test_aoeip.p4 --gen-dir /tmp/tmp1qb5qbf4/ --device ARRIA10 --p4runtime-fi
les /tmp/tmp1qb5qbf4/P4Info.txt --arch IntelArch


test_aoeip.p4(246): [--Wwarn=unused] warning: Table big_acl is not used; removing
 table big_acl {
       ^^^^^^^
test_aoeip.p4(260): [--Wwarn=unused] warning: Table big_acl2 is not used; removing
 table big_acl2 {
       ^^^^^^^^
test_aoeip.p4(115): [--Wwarn=uninitialized_out_param] warning: out parameter 'meta' may be uninitialized when 'ParserImpl' terminates
            out metadata_t meta,
                           ^^^^
test_aoeip.p4(113)
parser ParserImpl(packet_in pkt,
       ^^^^^^^^^^
arch_aoeip.p4(84): warning: Output parameter is missing connection to either next block or architecture: md_user
    inout M md_user,
            ^^^^^^^
warning: Implicit parser error output is not connected to downstream block, output can be connected by defining input parameter with error typ
e in MA block
arch_aoeip.p4(37): [--Werror=unsupported] error: update: Call-response architecture-owned externs not supported
 void update(out bit<256> data, in bit<2> iface, in bit<3> op);
      ^^^^^^
error: 1 errors were detected during the compilation.

[debug] Died in invoke_p4c. Exiting...

[rajput@srvys02:.../AOE/custom_design ]%
```

FIGURE 3.8 L'architecture P4 n'accepte pas les réponses de l'externe P4

tous les ports des deux modules (DMA et PCIe). La méthode prend en paramètre une des variables «interface_signals» qu'on utilise pour écrire le nom des ports des instances. Cette fonction repère aussi les signaux qui représentent les ports du DMA qui doivent être remplacés (pour la simulation) à l'interface serveur (wire logic in_pkt, wire logic in_meta, wire logic in_usr et wire logic nomatch_pkt en server#(.SDARG_BITS(32), .DATA_BITS(512)) in_pkt(),server#(.SDARG_BITS(32), .DATA_BITS(512)) in_meta, server#(.SDARG_BITS(32), .DATA_BITS(512)) in_usr, etc). Le fichier auquel on fait référence, ici, est celui qui contient les signaux des ports de nos modules d'interfaces.

Dès que les listes sont disponibles pour les ports des signaux et les appels d'instanciations des interfaces, il est possible de prendre le fichier HDL généré par le code PyXHDL pour les interfaces DMA et PCIe afin de le rendre disponible pour la synthèse dans le fichier de simulation du pipeline P4 «/simulation». Notez que le fichier auquel on fait référence, ici, est celui des interfaces «dma.v» (Verilog) qui est différent du fichier d'instanciation «interface.txt» (servant à créer l'appel et la liste des signaux à partir des ports des interfaces générées). Le fichier d'interfaces «dma.v» comporte les deux modules d'interfaces, donc le DMA et le PCIe dans un même fichier Verilog.

La fonction generate_interface_instance() du fichier Python principal, quant à elle, écrit le code de l'interface dans un des fichiers nommés function_wrapper.sv, avec les instances des deux modules, à savoir les DMA et PCIe, ainsi que les signaux corrigés. Cette fonction a été choisie parce qu'elle se trouve dans le fichier ayant la plus grande hiérarchie par rapport aux IP P4 de l'externe utilisé dans un tableau P4. Le code HLS génère ce fichier matériel en SystemVerilog comme la plupart des IP de l'externe P4, mais il occupe le haut de cette hiérarchie de fichiers des IP (il regroupe toutes les fonctionnalités des IPs du tableau P4). Dans ce fichier, la fonction generate_interface_instance() insère aussi une routine qui permettra de faire les tests sur l'interface générée. La routine a pour but de préparer un test de transfert de données qui sera effectué pour valider le fonctionnement de l'interface et de son instanciation dans le système P4.

Il est à noter que le fichier function_wrapper.sv avait été préalablement repéré (localiser) à l'aide de la fonction get_file_location(name) en lui donnant en paramètre le nom («name») du fichier qu'on recherche.

3.3.6 Ajustements supplémentaires aux dossiers de l'interface

En ajoutant les fichiers qui résultent de la génération de l'interface dans le dossier de simulation, il faut aussi faire un ajout à la liste des fichiers de simulation nommés «P4_systemverilog _files.f» et «P4_vhdl_files.f» pour que la compilation soit complète de tous les modules ma-

tériels. Pour ce faire, le script Python de la compilation fait un repérage (localise) de ces fichiers à l'aide du module «os» de Python et y ajoute le nom et le chemin de chacun des fichiers des interfaces générées.

Avant de faire les repérages des fichiers, le script fait aussi le compte de tous les pipelines implémentés par l'utilisateur P4. Ainsi, par la suite, il est possible de faire toutes les opérations sur les fichiers de celle-ci (fichiers de simulation, fichiers d'interfaces, fichier d'instanciations, etc).

Donc, dans le dossier de simulation, un utilisateur devrait voir l'ajout d'un dossier contenant les fichiers venant de la génération de l'interface nommée «/src». Il est à noter que ce fichier doit être ajouté par l'utilisateur à son répertoire personnel et d'indiquer son emplacement au script Python d'Intel (fichier Python de compilation principal). Ce script va automatiquement considérer ce chemin et mettra dans ce dossier les fichiers d'interface venant de PyXHDL (inteface.txt et dma.sv).

3.4 Composition du réseau

Cette section présente l'utilité des SmartNIC pour le quelle on utilise une interface générée automatiquement. Dans ce projet, un programme P4 est compiler pour programmer un commutateur réseau NIC qui peut être dit SmartNIC et qui peut être utilie pour l'optimisation des performances étant un SmartNICs programmables en P4 [23, 24]. La fonction principale d'un NIC est de permettre à un appareil de se connecter à un réseau, qu'il s'agisse d'un réseau local (LAN), d'un réseau étendu (WAN) ou d'internet. Ce qui améliore les performances réseau pour des tâches comme le streaming vidéo, le transfert de gros fichiers, ou les jeux en ligne.

Le réseau est composé de plusieurs éléments de la topologie. Notamment, le FPGA PAC N3000 est un FPGA agit comme un «SmartNIC» dans le réseau et est installé dans le serveur. Il a comme principal objectif de faire la distribution de la charge parmi les paquets du serveur et peut servir d'accélérateur en cas de congestion dès qu'il y a un surplus de paquets à traiter. Il permet, donc, de faire la classification rapide des paquets dans le serveur. Un SmartNIC comme le PAC N3000, qui est compatible avec la langue P4, permet aussi de faire le remplacement parfait des commutateurs traditionnels sans avoir à changer de modèle pour une nouvelle fonctionnalité. Pour cette raison, il est simplement possible de faire un programme P4 qui va générer la langue HDL pour faire fonctionner un FPGA NIC avec plus qu'une fonctionnalité. La seule différence avec la convention est d'avoir un compilateur P4 adéquat pour l'architecture d'un FPGA de ce type.

3.4.1 Synthèse de la solution proposée

Sachant ce qu'est un SmartNIC dans un réseau, il est maintenant possible de l'utiliser pour des applications comme la détection d'anomalies, à l'image du projet Pigasus. Il est important de comprendre que la capacité de prendre en charge des fonctionnalités du réseau par des accélérateurs matériel consiste simplement à transférer certains processus vers un FPGA ou un autre périphérique. Plusieurs études considèrent ce type de solution pour des applications comme la détection d'anomalies, la classification et l'analyse de paquets pour des besoins particuliers [25–27].

Selon la figure 2.2 du projet Pigasus et le pipeline P4 à la figure 3.1, il est possible de voir que l'architecture d'un FPGA doit supporter le transfert de domaine. Les interfaces automatisées permettent la communication en tant que service infrastructure, qui augmente la productivité de la conception pour les systèmes multi-FPGA. De plus, la fonctionnalité match&action est bien pour faire le pont entre le FPGA et le CPU, car c'est l'endroit où se trouve la plus grande charge des pipelines P4.

De plus, une approche pour mettre en œuvre le débordement dynamique consiste à utiliser plusieurs FPGA. Dans ce modèle, un FPGA de sauvegarde sera réutilisé au moment de l'exécution pour gérer le trafic de débordement d'un FPGA principal. Le calcul complexe peut être acheminé via le canal de communication inter-fpga, tel que le PCIe-DMA.

Dans cet ordre d'idée, il faut faire le lien entre les périphériques et un pipeline de traitement P4 classique. Pour ce faire, des interfaces PCIe-DMA apparaissent comme une solution valable. Il est possible de prendre comme exemple d'utilisation concret le système de PCIe-DMA d'Intel de la référence [28]. Pour mettre en œuvre cette solution, une étape possible est de concevoir une application pour un générateur de paquets avec une interface Avalon à un seul port, prenant en charge plusieurs canaux sans entrelacement. Deuxièmement, Ceci peut conduire dans un deuxième temps à un DMA multi-canaux H2D et D2H via une interface Avalon-MM en mémoire. Le cœur IP DMA multicanaux pour PCI Express fournit un port maître de lecture/écriture Avalon-MM. Dans ce contexte, il est possible d'exploiter des interfaces générées automatiquement en code HDL sont souvent utilisées dans les applications d'Intel.

Dans ce chapitre, la génération automatique d'interface à l'aide de l'outil PyXHDL à été discutée. Une interface matérielle (PCIe-DMA) générée en Verilog a fait l'objet de cette partie. L'appel à la compilation du code Python de l'outil PyXHDL à partir du code Python du compilateur d'Intel a également été expliquée. D'ailleurs, ce chapitre a mis en lumière la possibilité d'interconnecter le pipeline P4 avec une interface générée automatiquement.

Ceci est effectué à l'aide des processus d'automatisation via le script Python spécifiquement

à la section 3.3.1.

Le chapitre a aussi démontré que l'interface générée est une référence fonctionnelle du projet Pigasus qui est aussi utilisable pour un code P4 avec externe du compilateur P4 d'Intel. Le prochain chapitre va présenter des expérimentations avec l'interface générée automatiquement. Cette interface effectue un simple envoi et la réception de données à l'aide de la méthodologie d'automatisation des sections précédentes (sections 3.3.1 à 3.3.4).

CHAPITRE 4 EXPERIMENTATIONS ET RESULTATS

Ce chapitre présente le test de fonctionnement qui valide les interfaces proposées, les performances du réseau qui en découlent, ainsi que les résultats obtenus.

4.1 Test et fonctionnement

Le DPI-C est un outil qui sert au SystemVerilog pour exploiter les fonctionnalités du C. Les fonctions implémentées en C peuvent être appelées depuis SystemVerilog à l'aide des déclarations d'importation "DPI". Ces fonctions sont appelées tâches et fonctions importées. Toutes les tâches et fonctions importées doivent être déclarées par le mot-clé «import» dans l'entête du fichier SystemVerilog. Donc, les fonctions et les tâches implémentées en SystemVerilog et spécifiées dans les déclarations d'exportation "DPI" peuvent être appelées depuis un code C.

4.1.1 Préparation à l'extraction du résultat de l'externe de P4

Le fichier «function_wrapper.sv» a été introduit et expliqué à la section 3.3.4 du chapitre précédent. Ce fichier fait appel à la mémoire BRAM (le module dans le fichier bram _simple2port.sv) pour enregistrer la donnée du paramètre de l'externe venant du programme P4. Dans notre code Python, on l'implémente pour générer un transfert de données entre un code C++ (dpi_comm.cpp du schéma 3.3) et le code SystemVerilog du module bram _simple2port. Principalement, ce code C++ n'est rien d'autre qu'un pont entre l'émulateur de CPU (QEMU) Unicorn et la valeur de l'externe transmise par le module bram_simple2port en simulation. La valeur de la donnée est alors attribuée à cette fonction écrite en C++ qu'on utilise dans le code SystemVerilog du module bram_simple2port à l'aide du module de DPI-C.

Le module DPI-C a pour objectif de faire la liaison entre le langage C++ et le langage SystemVerilog. Pour ce faire, il faut avoir compilé le code C++ avec un compilateur comme le «g++» et d'utiliser le DPI pour des fins de communication. Dans ce projet, ce médium est utilisé pour faire le transfert de la donnée de l'externe au Unicorn (l'émulateur de CPU).

Les listes (nommées Pipeline_pipe0_sv.f et Pipeline_pipe0_vhd.f) disponibles pour la compilation des modules matériels, avant la simulation, sont corrigées (triées selon la priorité de compilation) pour certains des modules qui ne sont pas dans le bon ordre de compilation. Cette correction est apportée surtout pour le fichier VHDL p4_top_arch.vhd qui est toujours placé pour la compilation avant certains des fichiers de bas niveau hiérarchique. Ce

sont des fichiers utilisés dans parser_pkg.vhd ce qui cause tout le temps une erreur lors de la compilation.

Dans le but de favoriser un environnement de validation, il a été obligatoire d'attribuer une valeur par défaut à la donnée entrant dans le tableau contenant l'externe P4, donc dans le fichier «wrapper.vhd» comme expliqué à la section 3.3.3. Il a fallu changer la valeur associée au signal in_data_stream_table _ePort_0_0_data qui a une taille binaire de 742 bits. Pour bien avoir accès à la valeur de l'externe, il faut que la valeur du signal in_exit_aligned qui est d'un bit soit de zéro, la valeur du signal in_enable_aligned qui est aussi d'un bit soit à 1 et la valeur du signal in_action_aligned soit elle aussi à 1, donc de cette façon les bonnes clés seront assignées au tableau P4 pour l'action d'externe.

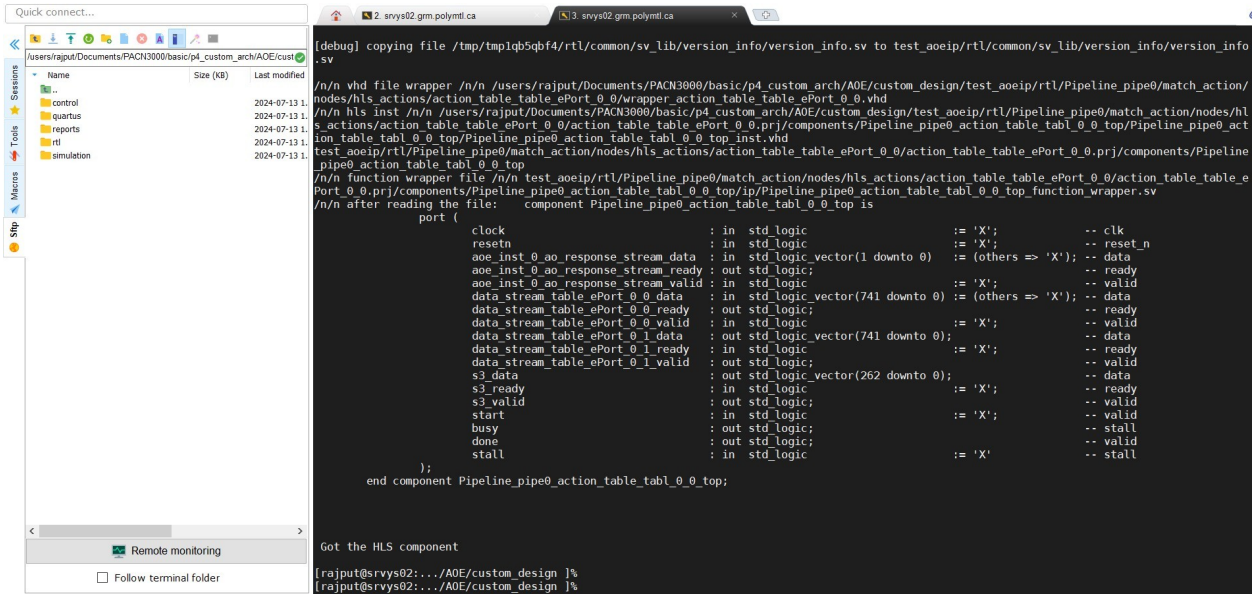

FIGURE 4.1 Génération de l'interface avec le nouveau compilateur

4.2 Résultats obtenus avec l'interface générée automatiquement

Les interfaces générées automatiquement en Verilog sont utilisées pour envoyer une valeur de l'externe à l'émulateur QEMU (Unicorn). Un calcul simple, comme une multiplication en C++ (7x20=140), est effectué pour la valeur de retour. Plus précisément, l'échange (pipeline P4 et émulateur) de valeurs est réalisé à l'aide de l'interface PCIe-DMA. De cette expérimentation, il est clair que la génération automatique d'un système d'interface PCIe-DMA est bénéfique, en effet, cette interface permet un accès universel aux périphériques du pipeline P4 avec un effort minimal.

La liste suivante présente les étapes de la réalisation d'interface avec l'utilisation de l'émulateur QEMU :

- Générer le code HLS à partir du programme P4
- Ajouter un Avalon Stream au code HLS pour la donnée de l'externe
- Générer l'interface PCIe-DMA à l'aide de PyXHDL
- Générer l'interconnectivité entre un pipeline P4 et l'interface
- Compiler et aligner le code C++ de l'externe dans l'émulateur
- Compiler et aligner le code C++ du DPI-C (pont entre la simulation matériel et logiciel)
- Envoyer la donnée à travers DPI-C dans ModelSim
- Faire le calcul à l'aide de l'émulateur x86 et retourner la valeur à ModelSim à travers DPI-C

Les modules matériels générés par l'utilisation du compilateur P4 sont à la base de la structure du pipeline P4. Il est à noter que les tableaux P4 constituent la plus grande partie de la logique du transfert de paquets. Dans le but de connaître la quantité de ressources utilisées par les modules P4 et les interfaces, il a été important pour nous de réaliser les étapes de post-synthèse, comme illustré à la figure 4.2.

Comme expliqué précédemment, la validation temporelle est réalisée en utilisant ModelSim. ModelSim prend en charge l'analyse temporelle détaillée, permettant aux concepteurs de simuler les relations de synchronisation complexes entre les signaux dans un circuit. Cela aide à identifier les problèmes de synchronisation potentiels, tels que les violations de temps de maintien et de prépositionnement des «Flip-Flop» et les conditions de course, en veillant à ce que la conception réponde aux spécifications de synchronisation requises avant la mise en œuvre matérielle. En outre, les fonctionnalités de débogage robustes de ModelSim et les formes d'onde complètes facilitent l'analyse approfondie et l'optimisation des performances temporelles du circuit, contribuant finalement à la création de systèmes numériques plus efficaces visible à la figure 3.4.

La figure 4.3 montre les différentes étapes de test effectuées pour valider le fonctionnement des interfaces générées. L'ensemble des blocs connectés après le bloc Fonction_Wrapper (un module SystemVerilog généré par le compilateur P4) sont automatiquement générés à l'aide de l'outil PyXHDL et du code Python pour la gestion des tests et des interfaces. Le transfert de données est effectué afin de créer un environnement propice à l'utilisation de l'interface. Ce transfert a lieu dans le module DMA et se poursuit jusqu'au module fpga2cpu_pcie, qui utilise le module bram_simple2port pour faire le lien entre ce module SystemVerilog et l'émulateur de CPU (QEMU).

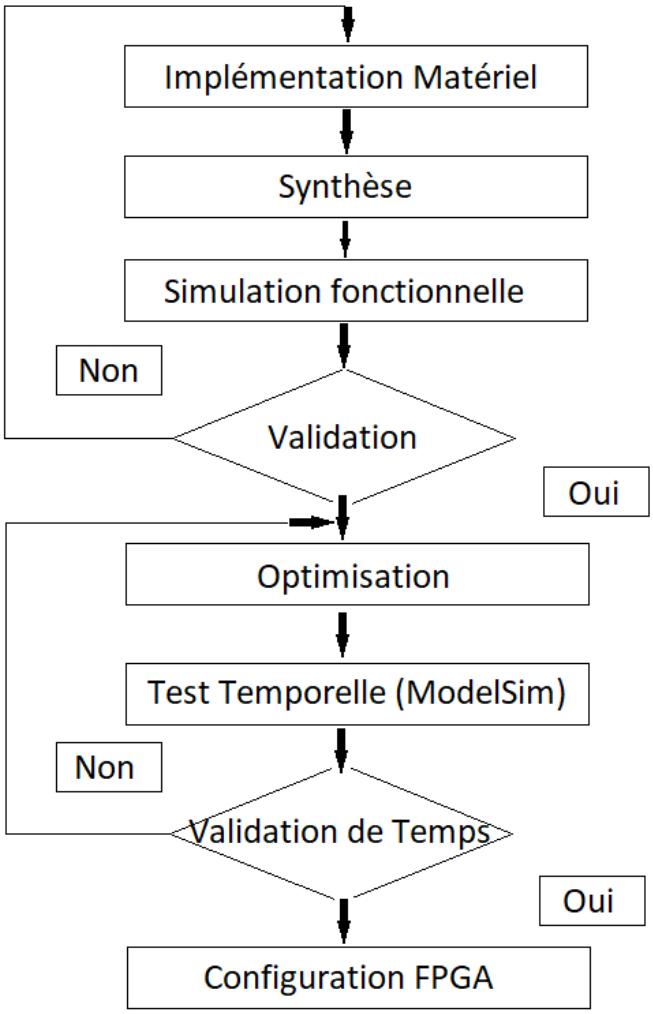

FIGURE 4.2 Étape d'implantation de matériel dérivé de P4

Il est à noter que la fonction «int send_data_to_cpu(addr, data)» écrite en C++ est réellement le lien entre le code SystemVerilog et le code C++. Cette fonction exécute un code Python qui lui utilise l'émulateur de CPU (send_data_to_cpu.py). La donnée et l'adresse envoyées en paramètre à la fonction en question sont placés dans la file d'instructions du CPU pour être exécutées. Le résultat obtenu est renvoyé au module SystemVerilog (bram_simple2port), qui transmet ensuite les données en sortie jusqu'au module PCIe vers le pipeline P4.

Les données venant de l'externe sont placées dans in_pkt (entré du module DMA) une fois extraite du module matériel Fonction_Wrapper à l'aide de la commande envoyée au module VHDL «wrapper.vhd».

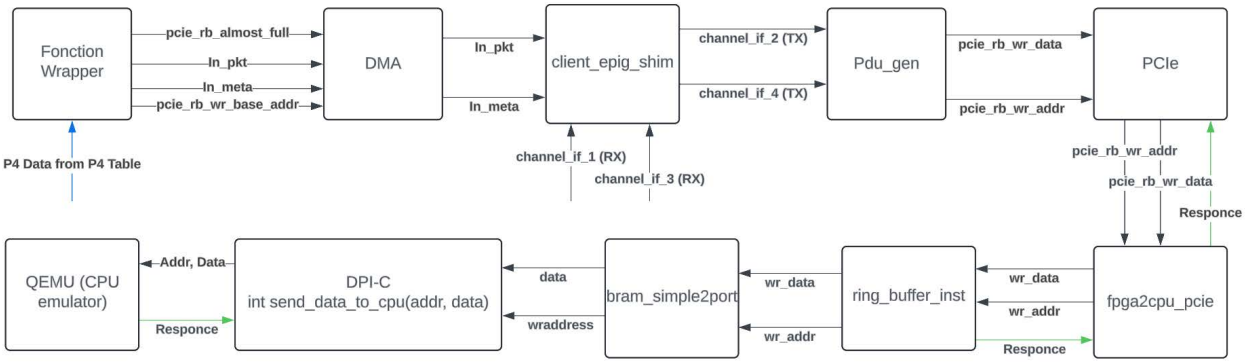

FIGURE 4.3 La validation du fonctionnement des interfaces

Il est également possible de voir les canaux (channel_if_x en RX ou TX) de communication reliés au client_epig_shim, qui effectue un transfert de données avec les interfaces System-Verilog pour la simulation matérielle.

Dans cette expérience, l'objectif était de transférer les données de l'exécution de l'externe appelé en P4, soit une valeur de 7, que nous avons multipliée par 20 dans l'émulateur, obtenant ainsi 140 comme résultat de retour pour l'interface générée. L'adresse utilisée était 3, choisie arbitrairement en fonction de la taille de la mémoire de l'émulateur. Le retour du résultat obtenu est visible dans la simulation ModelSim, semblable à la figure 3.4.

4.3 Discussion des résultats

L'analyse des ressources consommées montre que le système du FPGA de ce projet (Arria10 de série 10AX115H1F34I1SG) est efficace pour la mise en oeuvre des circuits implémentés. Le tableau 4.1 présente une vue d'ensemble pour montrer les ressources utilisées par le matériel du réseau P4 incluant les modules de l'interface générée.

TABLEAU 4.1 Ressources utilisées par l'architecture P4 modifié après synthèse sur Quartus pour le FPGA Arria10

Bloc	LUT	BRAM	Registres
Ressources utilisé	7745	44	16 352
Ressources disponibles	427 200	2 713	1 708 800
Pourcentage d'utilisation	1,81 %	1,62 %	0,96 %

La distribution des ressources pour les fonctions des diverses méthodes est détaillée au tableau 4.3.

Tableau 4.2 Différence des ressources utilisées par l'architecture P4 modifié après synthèse sur Quartus pour le FPGA Arria10

Ressources	Avec interface	Sans Interface	Différence Relatif
Lut	16,352	16071	1.64 %
Registres	7745	7551	4.54 %

Ces résultats sont obtenus quand l'outil cible un FPGA Arria10. Le tableau 4.3 n'est pas exhaustif et donc, ne montre pas tous les statistiques du rapport de synthèse telles que l'utilisation des nombres d'ALM (Adaptive Logic Module), d'ALM récupérables par regroupement dense, d'ALM non disponibles, de bits des blocs de mémoire, de blocs DSP, etc. Par ailleurs, le rapport de compilation présente plusieurs tableaux de ressources, comme celui du «Fan-Out». Dans le tableau 4.3, les «Dedicated Logic Registers» sont les registres utilisés pour ce système d'interfaces incluant le pipeline P4. Les «M20Ks» sont des blocs de mémoire. En outre, les ALMs représentent des ressources logiques du FPGA. Les données sont distribuées par entité selon un ordre hiérarchique du programme P4 compilé à travers un banc de test.

L'étape du placement attribue des emplacements spécifiques pour chaque élément logique du FPGA en fonction de la «netlist» compilée, en optimisant des facteurs tels que les performances et la surface. Le routage connecte ensuite ces éléments placés avec des fils physiques, garantissant l'intégrité du signal et respectant les contraintes de synchronisation. Ce processus est essentiel pour transformer une fonctionnalité de haut niveau en une implémentation physique, assurant ainsi son fonctionnement correct et efficace dans le FPGA.

En se basant sur le tableau 4.1, il est évident que les ressources nécessaires pour faire fonctionner les interfaces générées dans les pipelines P4 sont largement suffisantes. Dans une puce FPGA, les fonctionnalités sont généralement réparties entre plusieurs blocs logiques configurables, chacun étant capable de fonctionner de manière autonome tout en produisant des résultats cohérents et uniques. Chaque bloc logique possède son propre jeu de bascules («Flip-Flop») ainsi que des tables de recherche. Table de recherche ou LUT est comme un petit morceau de RAM qui est chargé chaque fois qu'on allume une puce FPGA. Il définit et dirige le comportement de la logique combinatoire de la puce en fonction du code VHDL ou Verilog, en se référant aux valeurs prédéterminées pour produire les résultats souhaités. Cela signifie que les entrées présentent dans la table de recherche sont essentiellement les lignes d'adresse d'une cellule RAM d'un bit de largeur. Dans le cas de ce projet, la compilation avec Quartus des modules matériels générés à partir du compilateur d'Intel et de l'outil Python

(PyXHDL) montre une utilisation de ressources de 1,81 % de la logique (table de recherche ou LUT) pour une puce Arria10. De plus, les ressources sont limitées à 1,62 % pour la mémoire (bloc de RAM) et 0,96 % pour les bascules (registres). La différence avec le pipeline sans interface généré est de 281 registres et 194 LUT, donc une différence de 1,64 % et de 4,54 % pour le nombre de registres et le nombre d'unités logiques (LUT) respectivement comme au tableau 4.2. Ces résultats confirment la faible complexité des interfaces générées. Le FPGA ciblé, un Arria10, est donc bien en mesure de les implémenter.

Tableau 4.3 L'utilisation des ressources par entités matériels (synthèse Quartus)

ALMs needed [=A-B+C]	Dedicated Logic Registers	M20Ks	Entity Name
7736.3 (1213.5)	16352 (0)	44	P4DataPlaneIP
6412.2 (0.0)	15080 (0)	44	Pipeline_pipe0
24.5 (24.5)	58 (58)	0	axi4lite2mi32
6236.9 (0.0)	13831 (0)	44	Pipeline_pipe0_NP4_ATOM
45.5 (7.9)	87 (0)	0	MI_SPLITTER_PLUS
36.4 (5.8)	87 (13)	0	MI_PIPE
30.7 (0.0)	74 (0)	0	PIPE
30.7 (30.7)	74 (74)	0	PIPE_REG
6189.2 (6.8)	13738 (0)	44	Pipeline_pipe0_p4top
3213.0 (3.6)	4209 (0)	0	Pipeline_pipe0_deparser_top
161.2 (0.0)	5 (0)	0	altdpram
161.2 (161.2)	5 (5)	0	dpram_pm32
35.0 (35.0)	49 (49)	11	FIFOX
0.0 (0.0)	0 (0)	11	SDP_BRAM_BEHAV
0.0 (0.0)	0 (0)	11	altsyncram
0.0 (0.0)	0 (0)	11	altsyncram_vi72
33.0 (33.0)	34 (34)	2	FIFOX
0.0 (0.0)	0 (0)	2	SDP_BRAM_BEHAV
0.0 (0.0)	0 (0)	2	altsyncram
0.0 (0.0)	0 (0)	2	altsyncram_nc72
542.8 (0.0)	1228 (0)	3	Pipeline_pipe0_match_action_to
542.8 (47.8)	1228 (85)	3	Pipeline_pipe0_table_table_ePort_0_0 _ent
259.3 (0.0)	557 (0)	0	Pipeline_pipe0_wrapper_action_table _table_ePort_0_0
259.3 (0.0)	557 (0)	0	Pipeline_pipe0_action_table_tabl_0_0 _top
259.3 (0.0)	557 (0)	0	Pipeline_pipe0_action_table_tabl_0_0 _top_internal
259.3 (0.0)	557 (0)	0	Pipeline_pipe0_action_table_tabl_0_0 _top_function_wrapper

CHAPITRE 5 CONCLUSION ET RECOMMANDATIONS

Le travail présenté dans ce mémoire propose un environnement des interfaces qui combine la modélisation logicielle des interfaces matérielles pour résoudre le problème d'ajout de fonctionnalités externes aux applications réseau configurables et programmables. Ce chapitre résume les travaux réalisés. Dans un premier temps, une synthèse des réalisations est proposée. Dans une deuxième section, certaines limites du travail réalisé et les contraintes associées à cette solution sont revues. Finalement, des axes de recherches futures sont proposés.

5.1 Synthèse des travaux

La conception de l'architecture, de ce projet, combine la génération et la simulation d'une interface PCIe-DMA à l'essai avec la validation d'une implémentation sur Modelsim. ModelSim fait la communication avec l'émulateur d'un CPU x86. Dans ce cas, la communication entre un ordinateur hôte et la carte FPGA est réalisée par des interfaces générées automatiquement à l'aide de l'outil PyXHDL. Cela garantit que l'échange de données sur différentes interfaces peut avoir lieu simultanément. Sur la plate-forme d'Intel, l'interface utilisée est celle de la famille Intel. Le programme P4 exécuté dans le serveur à travers le compilateur Intel permet la description des interfaces par le médium des externes (AOE). La carte PAC N3000 avec un Arria10 est une carte de développement d'accélération à usage général. Les données brutes reçues par les ports du module «function_wrapper» sont directement reçues de la part du DPI-C venant de l'émulateur QEMU.

L'exécution de la nouvelle plate-forme permet un gain significatif en temps de compilation du programme P4 et la génération automatique de l'interface à travers PyXHDL. Le temps d'exécution de la compilation du système principal est maintenant d'environ 40 secondes, ceci inclut la génération du code matériel pour le programme P4, la génération des bancs d'essais, la génération du code matériel d'interface ainsi que les interconnections du pipeline P4 à l'interface. Ces générations sont effectuées en modifiant le code Python du compilateur Intel.

5.2 Limitations et contraintes

L'implémentation de cet environnement présente deux limitations précédemment évoquées. La première concerne la flexibilité d'obtenir une réponse externe au pipeline P4 d'Intel. Étant donné que la partie matérielle est insérée dans le FPGA avec le DUT et que les données sont

stockées dans des FIFO, il est possible de lire les valeurs de retour même si le compilateur d'Intel ne permet pas de réponse directe. Pour une taille réduite, par exemple, de 44 cases de mémoire, les blocs occupent très peu de place. Cependant, cette occupation peut augmenter considérablement si le volume de données augmente.

La deuxième limitation se situe au niveau de la carte Arria10. Une modification est nécessaire pour générer le «bitstream» (binaire) avec la nouvelle structure du code matériel, sans avoir à utiliser l'ancienne méthodologie présentée dans l'annexe pour le compilateur P4 d'Intel. Donc, sans synthétiser et extraire le binaire directement par une seule commande. En effet, une fois le «bitstream» téléchargé, le FPGA PAC N3000 tombe en panne, devenant inactif et inutilisable.

5.3 Travaux Futurs

Pour la continuité de cette recherche, des informations supplémentaires pourraient être ajoutées à un bus de métadonnées pour les éléments externes. De plus, pour le système de ce projet, il est nécessaire que la solution soit vérifiée pour toutes les architectures du programme P4, y compris pour les autres composants d'Intel comme les Tofino et les autres NIC. Il est également possible de mensionner que la compilation des programmes P4 peut utiliser des solutions de la technologie de L'intelligence artificielle (IA) pour la détection et le filtrage de paquets ainsi que pour la génération des interfaces optimisées.

RÉFÉRENCES

[1] E. Papadogiannaki et S. Ioannidis, "Acceleration of intrusion detection in encrypted network traffic using heterogeneous hardware," *Sensors*, vol. 21, n°. 4, p. 1140, 2021.

[2] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar et J. Sherry, "Achieving 100gbps intrusion prevention on a single server," dans *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, p. 1083–1100.

[3] Y. Gorbounov, "Hardware design automation with python and verilog hdl," *Computer Science and Education in Computer Science*, vol. 17, n°. 1, p. 23–27, 2021.

[4] *P4Runtime Specification*, P416 Language Specification, 7 2022, rev. 1.2.3.

[5] I. Benacer, F.-R. Boyer et Y. Savaria, "A high-speed, scalable, and programmable traffic manager architecture for flow-based networking," *IEEE Access*, vol. 7, p. 2231–2243, 2018.

[6] F. Yousefifeshki, H. Li et F. Khomh, "Studying the challenges of developing hardware description language programs," *Information and Software Technology*, vol. 159, p. 107196, 2023.

[7] D. Castells-Rufas, G. Rotger et D. Novo, "Streamlining fpga circuit design and verification with python and py4hw," *XI SOUTHERN PROGRAMMABLE LOGIC CONFERENCE (SPL)*, 2023.

[8] M. Su, J.-P. David, Y. Savaria, B. Pontikakis et T. Luinaud, "An fpga-based hw/sw co-verification environment for programmable network devices," dans *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2022, p. 2529–2533.

[9] K. Jaic et M. C. Smith, "Enhancing hardware design flows with myhdl," dans *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, p. 28–31.

[10] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek et K. Asanović, "Chisel : constructing hardware in a scala embedded language," dans *Proceedings of the 49th Annual Design Automation Conference*, 2012, p. 1216–1225.

[11] V. Chandran, I. Mamatha et S. Tripathi, "Neda based hybrid architecture for dct—hwt," dans *2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA)*. IEEE, 2016, p. 1–6.

[12] S. Tian, I. Giechaskiel, W. Xiong et J. Szefer, "Cloud fpga cartography using pcie contention," dans *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021, p. 224–232.

[13] Y. Zhao, M. Li, Y. Zhang, Q. Lin et Z. Chen, "Research on fpga timing optimization methods with large on-chip memory resource utilization in pcie dma," dans *2016 CIE International Conference on Radar (RADAR)*. IEEE, 2016, p. 1–4.

[14] A. Rivitti, R. Bifulco, A. Tulumello, M. Bonola et S. Pontarelli, "Ehdl : Turning ebpf/xdp programs into hardware designs for the nic," dans *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, p. 208–223.

[15] Z. Cao, H. Su, Q. Yang, J. Shen, M. Wen et C. Zhang, "P4 to fpga-a fast approach for generating efficient network processors," *IEEE Access*, vol. 8, p. 23 440–23 456, 2020.

[16] G. Roopa et M. S. Reddy, "A study on pattern matching intrusion detection system for providing network security to improve the overall performance of security system." *Indian Journal of Public Health Research & Development*, vol. 9, nº. 11, 2018.

[17] J. Zhong, S. Chen et B. Han, "Fpga-cpu architecture accelerated regular expression matching with fast preprocessing," *The Computer Journal*, vol. 66, nº. 12, p. 2928–2947, 2023.

[18] Z. Zhong, M. Yang, J. Lang, C. Williams, L. Kronman, A. Sludds, H. Esfahanizadeh, D. Englund et M. Ghobadi, "Lightning : A reconfigurable photonic-electronic smartnic for fast and energy-efficient inference," dans *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023, p. 452–472.

[19] P. Patel, K. Lim, K. Jhunjhunwalla, A. Martinez, M. Demoulin, J. Nelson, I. Zhang et T. Anderson, "Hybrid computing for interactive datacenter applications," *arXiv preprint arXiv :2304.04488*, 2023.

[20] M. Abbasmollaei, T. Ould-Bachir et Y. Savaria, "Normal and resilient mode fpga-based access gateway function through p4-generated rtl," dans *2024 20th International Conference on the Design of Reliable Communication Networks (DRCN)*, 2024, p. 32–38.

[21] A. Li et D. Wentzlaff, "Prga : An open-source fpga research and prototyping framework," dans *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, p. 127–137.

[22] *Intel® High Level Synthesis Compiler Standard Edition : Reference Manual*, 2023, https://www.intel.com/content/www/us/en/docs/programmable/683310/19-1/component-memories-memory-attributes.html, Accessed : 2023-10-08.

[23] E. F. Kfoury, S. Choueiri, A. Mazloum, A. AlSabeh, J. Gomez et J. Crichigno, "A comprehensive survey on smartnics : Architectures, development models, applications, and research directions," *IEEE Access*, 2024.

[24] J. Xing, Y. Qiu, K.-F. Hsu, S. Sui, K. Manaa, O. Shabtai, Y. Piasetzky, M. Kadosh, A. Krishnamurthy, T. E. Ng *et al.*, "Unleashing smartnic packet processing performance in p4," dans *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023, p. 1028–1042.

[25] Z. Hu, H. Hasegawa, Y. Yamaguchi et H. Shimada, "Enhancing detection of malicious traffic through fpga-based frequency transformation and machine learning," *IEEE Access*, vol. 12, p. 2648–2659, 2024.

[26] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak et N. Zilberman, "Iisy : Hybrid in-network classification using programmable switches," *IEEE/ACM Transactions on Networking*, vol. 32, n°. 3, p. 2555–2570, 2024.

[27] M. Scazzariello, T. Caiazzi, H. Ghasemirahni, T. Barbette, D. Kostić et M. Chiesa, "A {High-Speed} stateful packet processing approach for tbps programmable switches," dans *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, p. 1237–1255.

[28] *Multi Channel DMA Intel® FPGA IP for PCI Express* Design Example User Guide*, 2023, https://www.intel.com/content/www/us/en/docs/programmable/683517/24-2/single-port-avalon-st-packet-generate-check.html, Accessed : 2024-10-18.

ANNEXE A FICHIER PYTHON DE COMPILATION PRINCIPAL

Le fichier de compilation d'Intel est un outil indispensable lorsqu'on utilise le nouveau compilateur d'Intel. La compilation avec l'exécutable précédemment disponible avait davantage de capacité pour générer un binaire («Bitstream») pour un FPGA et de le charger dans le FPGA. Selon la figure A.1 la compilation du code P4 prenait un peu plus de temps. La compilation prenait environ deux à trois heures et ceci même si le code P4 (programme P4) contenait des erreurs. La consommation de temps était principalement due à toute la procédure exhaustive dans une seule ligne de commande : «./bsp-fw.sh.v0.7.3/bsp-fw.sh –gen-p4 p4-sw-sdk.src.v1.1.1/examples/p4-16/vlan_tagging_counter/p4-programs/top.p4 –use-flr –use-ext-stat –rtl /usr/packages/p4env/intel/inteldevstack/rtl/n3000_1_3_v1.5.7/ –gen-dir gen_16 –gen-qip –gen-fw –compiler p4-16».

Pour la commande précédente, il est possible de changer quelques paramètres comme le type de compilateur pour une version de code P4 de la génération $P4_{14}$ en indiquant «–compiler p4-14» et changer le nom du fichier généré en conséquence, donc la commande deviendrait quelque chose comme celle-ci : «./bsp-fw.sh.v0.7.3/bsp-fw.sh –gen-p4 p4-sw-sdk.src.v1.1.1/examples/p4-14 /vlan_tagging_counter/p4-programs/top.p4 –use-flr –use-ext-stat –rtl /usr/packages/p4env /intel/inteldevstack/rtl/n3000_1_3_v1.5.7/ –gen-dir gen_14 –gen-qip –gen-fw –compiler p4-14».

Dans la nouvelle syntaxe du $P4_{16}$, il est important d'utiliser les annotations pour AOEIP pour faire la génération des externes qui n'ont pas de fonctionnalité interne au programme P4. De la même façon qu'avant, ce script Python (intelfpga-p4c.py) prend aussi le fichier de l'architecture en considération du programme P4.

Il est possible d'utiliser le «–cppextern» pour attribuer un programme externe C++ au script Python. Donc, la commande pour la compilation de ce script devient la suivante : «python3.10 /users/rajput/ Documents/PACN3000/p4env/intel/intelfpga_p4c/bin/intelfpga-p4c.py – device ARRIA10 –debug-msg –temp-dir /tmp/tmp1qb5qbf4/ test_aoeip.p4 –cppextern /users/rajput/Documents/PACN3000/basic/p4_custom_arch/AOE/custom_design/test

_aoeip_v37/ simulation/src/add.cpp». Lors de la compilation avec cette commande, il est aussi important d'utiliser un attribut «–debug-msg» pour avoir plus d'informations du compilateur lors de l'exécution de la commande. La compilation demande aussi le modèle de la carte disponible (FPGA) comme dans notre cas c'est l'Arria10, donc il suffit de l'associer au paramètre «–device» de ce script.

Les ajouts qui ont pris place lors de la conception du système de ce projet sont pour l'automatisation de la génération du code HDL des interfaces écrites en langage de haut niveau à l'aide de l'outil PyXHDL. Alors, comme visible à la figure 3.2, le système des interfaces est connecté au pipeline P4 et tout ceci est effectué à l'aide du code Python offert avec le compilateur.

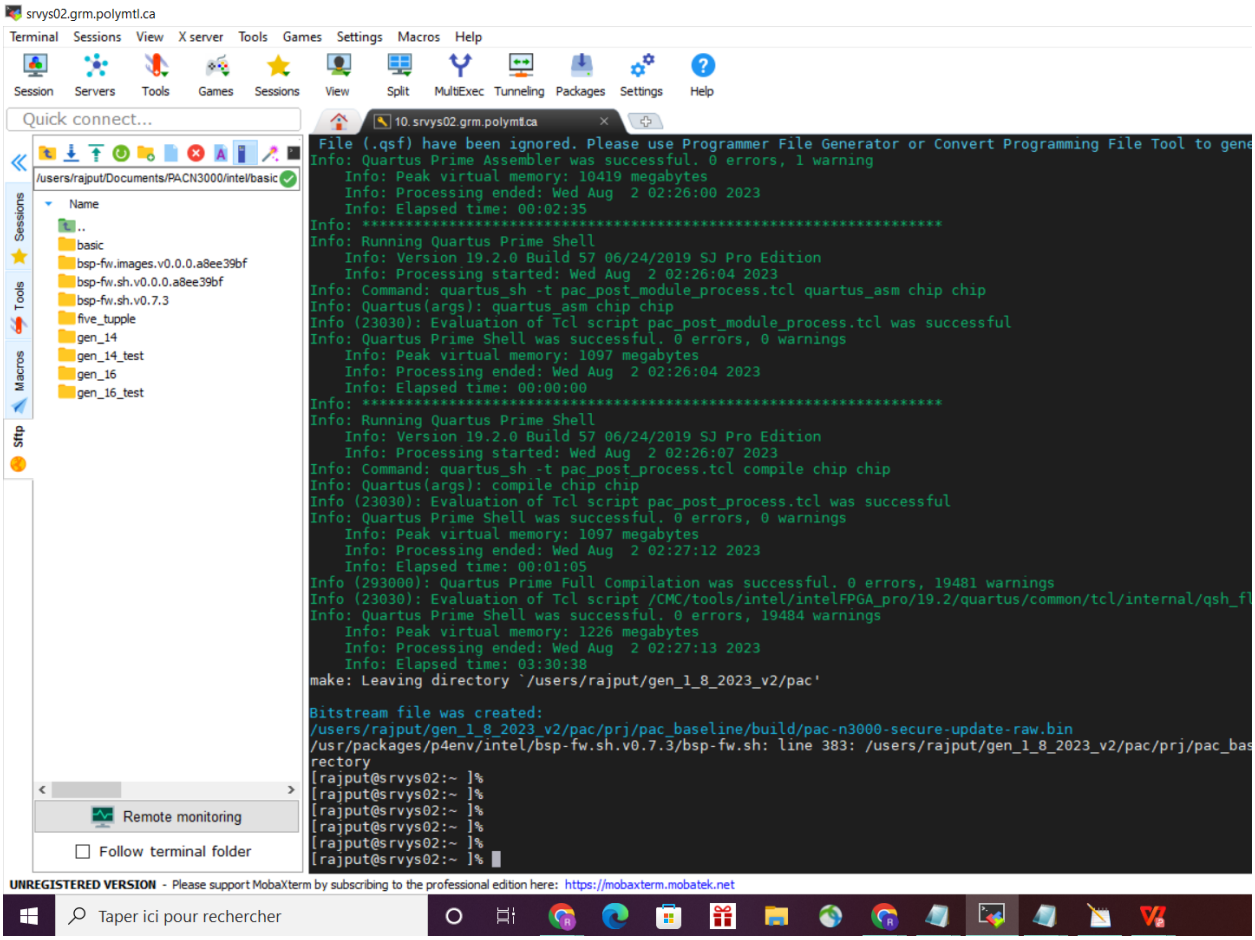

FIGURE A.1 Génération de bitstream avec l'ancien compilateur