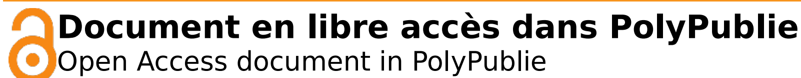
PolyPublie
Polytechnique Montréal

Titre: Title:	Élaborer un environnement de test pour la vérification et la validation d'applications réseaux configurables sur FPGA
Auteur: Author:	Mengyue Su
Date:	2022
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Su, M. (2022). Élaborer un environnement de test pour la vérification et la validation d'applications réseaux configurables sur FPGA [Master's thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/10301/


Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/10301/
Directeurs de recherche: Advisors:	Yvon Savaria, & Jean Pierre David
Programme: Program:	Génie électrique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Élaborer un environnement de test pour la vérification et la validation d'applications réseaux configurables sur FPGA

MENGYUE SU

Département de génie électrique

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

Génie électrique

Avril 2022

© Mengyue Su, 2022.

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

Élaborer un environnement de test pour la vérification et la validation d'applications réseaux configurables sur FPGA

présenté par **Mengyue SU**

en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

a été dûment accepté par le jury d'examen constitué de :

François LEDUC-PRIMEAU, président

Yvon SAVARIA, membre et directeur de recherche

Jean Pierre DAVID, membre et codirecteur de recherche

Guy BOIS, membre

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, Jean Pierre David de m'avoir soutenu et aidé tout au long de ce projet. Je voudrais également remercier Thomas Luinaud pour les suggestions apportées tout au long de ce projet, ainsi que tous les membres du laboratoire pour l'aide aux installations d'appareils et les discussions. Je remercie aussi Mickael Lallart et tous les membres de la Service de Relation Internationale à INSA Lyon de m'avoir aidé pendant le programme de Double Diplôme. Je remercie Bill Pontikakis pour son aide à l'écriture de l'article.

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

RÉSUMÉ

Le *Software-Defined Networking* (SDN) est une architecture importante qui a émergé au cours des années 2010. Elle introduit la séparation de plan de contrôle et de plan de données dans les périphériques réseau tel que les commutateurs et les routeurs. Cette architecture permet à l'utilisateur de définir la logique du plan de données et de centraliser le contrôle de ressource. La programmabilité des applications réseau est largement augmentée donc divers types d'équipements reprogrammables sont développés. Les FPGA, parmi d'autres approches, sont largement utilisés comme plateformes SDN en raison de leur haute reprogrammabilité, de leur flexibilité et de leur coût raisonnable.

Cependant, avec l'augmentation rapide de la complexité du matériel, le débogage devient de plus en plus difficile. Les bogues dans les périphériques réseau se traduisent par des pertes financières pour les fournisseurs de services réseau et dégradent la qualité de l'expérience pour les utilisateurs. Les outils de simulation ne peuvent pas garantir une couverture complète des pannes, car les bogues peuvent se manifester à tout moment dans le matériel. Pour atténuer ces problèmes, nous proposons un outil de co-vérification matériel/logiciel (HW/SW) qui cible les périphériques réseaux programmables. Le système intègre une simulation logicielle et une mise en œuvre matérielle précises au cycle d'horloge. Pour la mise en œuvre du logiciel, des outils ouverts tel que *CocoTB* et GHDL ont été utilisés. La partie matérielle est implémentée dans un FPGA, incluant le modèle à tester (*Device Under Test* (DUT)) et les interfaces de test. Les données peuvent être extraites des ports d'entrée/sortie (I/O) du DUT pendant l'exécution en temps réel. L'insertion des données produites par le banc de test logiciel est également prise en charge. Pour l'implémentation matérielle, des expériences rapportées ont été menées sur la plateforme *DE10-Standard* et la plateforme *NetFPGA-SUME*. Cette première est une carte de développement générale sous l'environnement d'Intel. Cette dernière est une nouvelle carte spécifique au développement de programme réseau équipée avec les ports Internet de haut débit. Pour identifier la source d'un bogue, les assertions matérielles sont supportées par l'environnement. Il permet de capturer les données avant et après la présentation d'une assertion.

ABSTRACT

During the last decade, new architectures in data network systems have been applied. Software-Defined Networking (SDN) introduces control plane and data plane separation in network devices such as switches and routers. This architecture allows the user to define their own logic of the data plane and centralize resource control. The programmability of network applications is greatly increased therefore various types of reprogrammable equipment are developed. Among various hardware platforms, due to their high reprogrammability, flexibility and reasonable cost, FPGAs are widely used to implement SDN components.

However, with the rapid increase of complexity in hardware, debugging becomes more and more difficult. Bugs in network devices translate to financial losses for the network service providers and degrade the quality of experience for the users. Simulation tools cannot guarantee complete fault coverage, as bugs can manifest themselves at any time in live hardware. To mitigate these issues, we offer a hardware / software (HW / SW) co-verification tool that targets programmable network devices. The system integrates precise software simulation and hardware implementation providing cycle accurate modeling. For the software implementation, open-source tools such as *CocoTB* and GHDL were used. The hardware part embeds in programmable hardware, including the Device Under Test (DUT), and its test interfaces. Data can be extracted from the input/output (I/O) ports of the DUT during real-time execution. Data insertion, produced by the software testbench, is also supported. For the hardware implementation, reported experiments were implemented on the *DE10-Standard* platform and the *NetFPGA-SUME* platform. The first is a general development board operating under the Intel environment. The latter is a new specific network programming development card equipped with high-speed Internet ports. To simplify the bug targeting, the hardware assertion is supported by the environment. It allows data to be captured before and after an assertion has triggered.

TABLE DES MATIÈRES

REMERCIEMENTS III

RÉSUMÉ IV

ABSTRACT V

TABLE DES MATIÈRES VI

LISTE DES TABLEAUX IX

LISTE DES FIGURES X

LISTE DES SIGLES ET ABRÉVIATIONS XIII

LISTE DES ANNEXES XV

CHAPITRE 1 INTRODUCTION 1

1.1 Contexte 1

1.2 Problématique et motivation 2

1.3 Objectifs de recherche 3

1.4 Contributions 4

1.5 Co-vérification 4

1.6 Organisation du mémoire 4

CHAPITRE 2 CONTEXTE 6

2.1 « Software-Defined Networking » 6

2.2 P4: « Programming Protocol-independent Packet Processors » 7

2.3 Cartes et Plateformes 7

2.3.1 La carte DE10-Standard et la plateforme d'Intel 8

2.3.2 La carte NetFPGA-SUME et la plateforme de Xilinx 8

2.4 *CocoTB* & GHDL 9

2.5 « *Open Verification Library* » 10

CHAPITRE 3 REVUE DE LA LITTERATURE 11
3.1 Les testeurs de réseau 11
3.1.1 Les testeurs de réseau open-source 11
3.1.2 Les testeurs de réseaux commerciaux 13
3.2 Les outils de débogage 13
3.2.1 Les simulateurs RTL 13
3.2.2 Les analyseurs logiques embarqués (ELA) 14
3.2.3 Combinaison de simulation et d'exécution 15
3.3 Retour sur la problématique 16
CHAPITRE 4 ARCHITECTURE DE L’ENVIRONNEMENT DE TEST 17
4.1 Architecture générale 17
4.2 Interfaces d’entrée et de sortie 19
4.3 L’environnement CocoTB 20
4.4 L’architecture de la plateforme d’Intel 20
4.4.1 Les interfaces utilisées 20
4.4.2 Détails de l’architecture 22
4.4.3 Mise en œuvre du module matériel 22
4.4.4 La communication avec « socket » 26
4.4.5 Le compteur de cycles 28
4.4.6 Modification de l’interface pour vérifier la « MAT » 28
4.4.7 Le module de traitement de sortie 33
4.5 L’architecture sur la plateforme de Xilinx 34
4.5.1 Les interfaces utilisées 35
4.5.2 Présentation d’un exemple : le projet « Reference NIC » 36

4.5.3 Détails de l’architecture 37

4.5.4 La conception du matériel 38

4.5.5 Communication à l’aide d’un UART 42

4.5.6 Modification au programme de test dans CocoTB 46

CHAPITRE 5 EXPERIMENTATIONS ET RESULTATS 47

5.1 Résultats sur la plateforme d’Intel 47

5.1.1 Système général 47

5.1.2 Vérification de la « MAT » 48

5.1.3 Le fichier avec les formes d’ondes 51

5.2 Résultats sur la plateforme de Xilinx 52

5.2.1 Configuration de l'implémentation 52

5.2.2 Test avec CocoTB 53

5.2.3 Test du bloc d’extraction pour la vérification d’assertions 55

5.2.4 Ajout du compteur d’horloge 57

5.3 Utilisation des ressources 59

CHAPITRE 6 DISCUSSION 60

CHAPITRE 7 CONCLUSION ET RECOMMANDATIONS 63

7.1 Synthèse des travaux 63

7.2 Limitations et contraintes 64

7.3 Travaux futurs 64

RÉFÉRENCES 66

ANNEXES 69

LISTE DES TABLEAUX

Tableau 4.1 : Distribution d'adresse de l'interface d'entrée avec bus de 128 bits........................24

Tableau 4.2: La configuration des modes de MAT..30

Tableau 4.3 : Les signaux de l'interface *AXI4-Stream*..35

Tableau 4.4 : Adresse en mémoire du bloc d'insertion...40

Tableau 4.5 : Adresse de mémoire de bloc d'extraction..40

Tableau 4.6 : Liste des fonctions utilisées pour la communication en *AXI4-Lite*..........................41

Tableau 4.7 : Liste des fonctions utilisées pour la communication en UART..............................42

Tableau 4.8 : Commande et fonctionnement 43

Tableau 4.9 : Fonctions crées pour la communication UART..44

Tableau 5.1 : Liste des entrées 47

Tableau 5.2 : Clés et leur valeur...49

Tableau 5.3 : Adresse MAC et IP 53

Tableau 5.4 : List des ports 56

Tableau 5.5: Utilisation des ressources..59

Tableau 6.1 : Comparaison avec les projets de recherche existants..62

LISTE DES FIGURES

Figure 4.1 : Architecture proposée par Luinaud [25]....17

Figure 4.2 : Architecture de la partie logicielle....18

Figure 4.3 : Architecture de la partie matérielle....18

Figure 4.4 : Exemple de bus d'interface mappée en mémoire....19

Figure 4.5: L'interface *Avalon Streaming*....21

Figure 4.6 : Chronogramme de l'interface *Avalon Streaming*....21

Figure 4.7 : Programmes et langages utilisés sur la plateforme d'Intel....22

Figure 4.8 : Architecture matérielle sur la plateforme Intel....23

Figure 4.9: Modèle des interfaces pour un canal....24

Figure 4.10 : Organisation des bits du signal de contrôle....24

Figure 4.11 : Arrangement des bits d'un bus de données de 128....25

Figure 4.12 : Logique de l'écriture de l'interface d'entrée....25

Figure 4.13: Process de communication en *socket*....26

Figure 4.14: Format de données dans un fichier d'entrée....27

Figure 4.15 : Format de données dans le *socket*....27

Figure 4.16 : Machine à états qui permet de caractériser le passage des données....28

Figure 4.17: L'interface de bloc MAT....29

Figure 4.18 : Architecture pour vérifier la MAT....31

Figure 4.19 : Organisation des signaux sur le PHV....31

Figure 4.20: L'interface de vérificateur « ovl_always »....32

Figure 4.21 : Insertion de l'assertion....32

Figure 4.22 : Format de données vcd....34

Figure 4.23 : Régénération de fichier VCD....34

Figure 4.24: Les signaux de l’interface *AXI4-Lite* 35

Figure 4.25 : Architecture générale sur la plateforme Xilinx 37

Figure 4.26 : Architecture matérielle pour la plateforme Xilinx...38

Figure 4.27 : Interfaces des blocs de communication de données a) bloc d’insertion b) bloc d’extraction c) bloc d’extraction avec assertion..39

Figure 4.28 : Format des champs qui expriment le niveau de remplissage 42

Figure 4.29 : Architecture de la partie logicielle..44

Figure 4.30 : Logigramme de la fonction a) write_to_insertion b) read_from_extraction 45

Figure 4.31 : Ordre des données dans le bus AXI4-Stream 46

Figure 4.32 : Format des données attendues dans les fichiers d’entrée ou de sortie.....................46

Figure 5.1: DUT pour le test du système général..47

Figure 5.2 : Résultats obtenus avec *CocoTB*..48

Figure 5.3 : Résultat en FPGA 48

Figure 5.4 : Processus de test en *CocoTB* pour la vérification du module MAT..........................49

Figure 5.5 : Résultats de la simulation dans *CocoTB* (MAT) 49

Figure 5.6 : Résultat de la partie de FPGA (MAT)...50

Figure 5.7 : Résultat de comparaison (MAT) 51

Figure 5.8: Formes d’ondes regénérées..52

Figure 5.9 : Schéma de mise en œuvre réalisé avec *Vivado* 52

Figure 5.10 : Configuration de l'implémentation 53

Figure 5.11 : Architecture pour le test avec *CocoTB* 53

Figure 5.12 : Procédure dans *CocoTB*...54

Figure 5.13 : Module de communication 54

Figure 5.14 : Résultat de test produits avec *CocoTB* 54

Figure 5.15 : Données récupérées par le bloc d'extraction pour la vérification d'assertions 55

Figure 5.16 : Séparation de données analysées 56

Figure 5.17 : Décodage d'un paquet 56

Figure 5.18 : Données de sortie du test avec le détecteur de l'adresse MAC 57

Figure 5.19 : Compteur de l'horloge 57

Figure 5.20 : Distribution des temps d'arrivé de paquet 58

LISTE DES SIGLES ET ABRÉVIATIONS

API	Application Programming Interface
ARP	Address Resolution Protocol
ASIC	Application-Specific Integrated Circuit
BMv2	Behavioral Model version 2
BRAM	Block Random Access Memory
CAM	Content-Addressable Memory
CBR	Constant Bit Rate
CocoTB	COroutine based COsimulation TestBench
DDR3 SDRAM	Double Data Rate 3 Synchronous Dynamic Random-Access Memory
DMA	Direct Memory Access
DSP	Digital Signal Processor
DUT	Device Under Test
ELA	Embedded Logic Analyzer
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
GHDL	G Hardware Design Language
GUI	Graphical User Interface
HDL	Hardware Description Language
HPS	Hardware Processor System
IP (core)	Intellectual Property
IPv4	Internet Protocol version 4
IRC	Industrial Research Chair
JTAG	Joint Test Action Group

MAC	Media Access Control
MAT	Match Action Table
MAU	Match Action Unit
NIC	Network Interface Controller
OCRAM	On-Chip Random Access Memory
OVL	Open Verification Library
P4	Programming Protocol-independent Packet Processors
PC	Personal Computer
PCIe	Peripheral Component Interconnect express
PHV	Protocol Header Vector
PISA	Protocol Independent Switch Architecture
QSFP	Quad Small Form-factor Pluggable
RTL	Register Transfer Level
SDE	Software Development Environment
SDN	Software-Defined Networking
SFP+	Small form-factor pluggable
SoC	System on a chip
UDP	User Datagram Protocol
VBR	Variable Bit Rate
VCD	Value Change Dump
VHDL	VHSIC Hardware Description Language

LISTE DES ANNEXES

Annexe A – Schéma du système sur plateforme de Xilinx 69

CHAPITRE 1 INTRODUCTION

1.1 Contexte

Les bogues dans les périphériques réseau se traduisent par des pertes financières pour les fournisseurs de services réseau et ils dégradent la qualité de l'expérience pour les utilisateurs. Les outils de simulation actuels ne permettent pas de garantir une couverture complète des bogues, car ils peuvent se manifester à tout moment au niveau matériel. Il est donc important de proposer des solutions permettant de vérifier l'implémentation complète (logicielle et matérielle).

Le *Software-Defined Networking* (SDN) [1] est une architecture importante qui a émergé au cours des années 2010. Elle introduit la séparation du plan de contrôle et du plan de données dans les périphériques réseaux tels que les commutateurs et les routeurs. Cette architecture offre davantage de flexibilité dans la gestion du réseau. Par la suite, grâce à cette caractéristique, de nombreux chercheurs ont fourni de grands efforts concentrés vers la programmabilité du plan de données pour le transfert des paquets. En conséquence, le langage P4 [2], un langage de programmation permettant d'exprimer la structure du traitement des paquets, a été introduit. Plusieurs fournisseurs proposent des équipements ; par exemple le commutateur programmable Tofino [3] offre de hautes performances tout en étant programmable. Il supporte l'implémentation des programmes P4 permettant aux utilisateurs de modifier le plan de données sans connaitre le matériel physique. Les compilateurs de P4 à HDL tels que p4-sdnet [4] sont aussi rendus disponibles par les producteurs des cartes programmables de réseaux.

Les FPGA sont largement utilisés comme plateformes SDN en raison de leur reprogrammabilité, de leur flexibilité ainsi que de leurs coûts d'ingénierie non-récurrents inférieurs à ceux des ASIC (*Application Specific Integrated Circuits*). Par conséquent, le SDN et les plans de données programmables sont devenus des technologies essentielles pour les systèmes de réseaux à débits de données élevés et avec une latence ultra-faible, telle que nécessaire dans les centres de données modernes et les réseaux 5G.

En réalité, la forme des paquets qui circulent sur les réseaux est très variable. Elle varie en termes de longueur, de contenu, de priorité, de latence tolérable, etc. À cause de cette diversité, il est pratiquement impossible de fournir une couverture de test complète aux fonctionnalités des périphériques réseau par des moyens de simulation. Actuellement, les interfaces SFP+ et QSFP

sont installées sur les périphériques réseau haut débit qui permettent d'atteindre un débit d'au moins 10 Gbit/s. Les cartes FPGA spécifiques au réseau peuvent fournir une précision/résolution temporelle de 6,4 ns ou mieux. Étant donné que les erreurs de synchronisation au niveau matériel ne peuvent pas nécessairement être observées en simulation, les outils de test en circuit physique tels que des analyseurs logiques intégrés (ELA) et JTAG peuvent être utilisés pour observer des signaux réels. Cependant, de telles méthodes offrent une visibilité limitée du signal dans le circuit et consomment beaucoup de ressources FPGA, dépendamment de la complexité de signaux à capturer. Les testeurs de performances de réseau commerciaux, par exemple IxNetwork [5], supportent un débit très élevé, mais ils sont coûteux et offrent une faible programmabilité. Les projets de recherche open-source comme OSNT [6] et son projet dérivé OP4T [7] ont des limites, car ils se concentrent sur la précision des mesures de latence. La plupart des testeurs, indépendamment de l'équipement à tester, visent à la mesure de performance. Ils ne donnent donc pas accès à des signaux matériels internes.

Ce projet fait partie de la chaire de Polytechnique Montréal intitulé : « *Industrial Research Chair (IRC) for High Speed and Programmable Packet Processing* ». Ce groupe de recherche se concentre sur le développement des applications programmables de traitement de paquets à grande vitesse. Les chercheurs travaillent sur des modules divers du plan de données tel que le parseur, le déparseur et le *Match Action Table* (MAT). Il est donc nécessaire de vérifier le fonctionnement de ces modules créés.

Le travail présenté dans ce mémoire s'intéresse à établir un environnement de test en coopération matériel/logiciel pour vérifier et valider les applications de type réseaux programmables.

1.2 Problématique et motivation

Des études montrent que les concepteurs passent 35 à 50% de leur temps à valider et à déboguer des applications. Selon certaines estimations, le coût du débogage, des tests et de la vérification peut représenter de 50 à 75% du budget total des projets de développement [24]. Durant le débogage des programmes sur FPGA, plusieurs outils sont utilisés par les ingénieurs, dans la partie logicielle et la partie matérielle.

La simulation est généralement la première étape de débogage. Les ingénieurs appliquent les signaux d'entrée à la conception et vérifient la sortie attendue par rapport à la sortie de conception simulée. C'est un moyen facile de détecter les erreurs de conception les plus évidentes. L'avantage de la simulation par rapport au débogage matériel est que la simulation peut se réaliser en logiciel, sans besoin de matériel de test. Cependant, créer une suite de tests qui couvre un système complexe à 100 % est quasiment impossible pour les grands programmes. De plus, comme il ne s'agit pas d'une implémentation matérielle, la simulation ne peut généralement pas détecter les événements asynchrones, les interactions du système à vitesse élevée ou les violations de la synchronisation.

Pour une meilleure vérification de la conception, les ingénieurs ajoutent du matériel de débogage dans leur circuit afin d'observer et de contrôler les signaux clés au sein de l'architecture. La technique la plus utilisée est l'analyseur logique intégré (ELA). Les grands fabricants de FPGA offrent leurs propres analyseurs pour faciliter le travail de développeur, par exemple le SignalTap II [8] d'Intel et le ChipScope [9] de Xilinx. Toutefois, ils requièrent une licence coûteuse et consomment les ressources du FPGA selon la complexité du système et les signaux à observer.

Ce mémoire introduit un environnement de test qui combine les modèles matériel et logiciel afin de conserver leurs avantages tout en limitant leurs inconvénients.

1.3 Objectifs de recherche

Ce mémoire a pour objectif premier de proposer des outils permettant de vérifier et de valider des applications écrites en HDL. Pour avoir une vérification et une validation complètes, l'outil devrait combiner la simulation logicielle et l'implémentation matérielle. L'architecture est implémentée sur deux plateformes différentes : *Intel Altera* FPGA et *Xilinx NetFPGA*. Le système a l'avantage du simulateur sur l'observabilité et la contrôlabilité, et évite son inconvénient au niveau de la détection d'erreurs de synchronisation en implémentant le même test en matériel. En plus, sur la plateforme *Xilinx NetFPGA*, un moniteur permet de capturer les données en pleine vitesse. La vérification par l'assertions est aussi supportée par l'environnement de test. L'utilisateur peut insérer les blocs d'assertion puis les connecte avec l'environnement de test afin de récupérer les données ciblées.

1.4 Contributions

La principale contribution de ce projet est une plateforme qui offre une vérification matérielle/logicielle des applications réseau. Premièrement, l'environnement réalisé sur la plateforme d'Intel permet de capturer les données du banc d'essai défini par l'utilisateur et les transmet au matériel. Ensuite, il récupère les données après traitement par le DUT en matériel, et les renvoie au banc d'essais.

Deuxièmement, l'environnement réalisé sur la plateforme de Xilinx réalise le même fonctionnement, en plus du débogage du plan de données réseau en temps réel. Il permet enfin de capturer les données entrantes et sortantes, avec ou sans un contrôle d'assertion.

1.5 Co-vérification

Dans la littérature existante, la co-vérification conjointe logiciel/matériel consiste à s'assurer que le logiciel du système embarqué fonctionne correctement avec le matériel et que le matériel a été correctement conçu pour exécuter le logiciel [31]. Le SoC est défini comme une puce unique qui équipe un ou plusieurs microprocesseurs, des fonctions logiques personnalisées spécifiques à l'application et un système embarqué. Les microprocesseurs et les processeurs de signal numérique (DSP) dans une puce ont obligé les ingénieurs à considérer le logiciel comme une partie du processus de vérification. Par contre, dans ce mémoire, nous utilisons cette expression avec un sens différent, à savoir la vérification en combinaison d'une partie logicielle et d'une partie matérielle. Le système à vérifier est une application réseau implémentée dans un FPGA. L'environnement de test combine un premier processus de vérification exécuté par un simulateur et un deuxième est une implémentation en FPGA. Les résultats sont ensuite collectés et comparés pour obtenir un rapport de vérification complet.

1.6 Organisation du mémoire

La suite de ce mémoire est organisée comme suit. Tout d'abord, les notions préliminaires théoriques sont présentées dans le chapitre 2. Par la suite, une revue des travaux sur les recherches existantes est résumée dans le chapitre 3. L'architecture implémentée sur les plateformes d'Intel et

de Xilinx sont exposées dans le chapitre 4. Dans le chapitre 5, une discussion sur les résultats est donnée. Finalement, le chapitre 6 conclut ce mémoire et propose des travaux futurs.

CHAPITRE 2 CONTEXTE

Ce chapitre présente les notions préliminaires concernant le projet. Il s'agit de la nouvelle structure et le langage de programmation de réseaux, le circuit à tester et les technologies ainsi que les plateformes utilisées.

2.1 « Software-Defined Networking »

Le *Software-Defined Networking (SDN)* [1] est un nouveau modèle d'architecture réseau. En utilisant le protocole *OpenFlow*, il est possible de séparer le plan de contrôle du plan de données. Le plan de contrôle est typiquement implémenté en logiciel, ce qui permet de centraliser sa gestion de façon distribuée sur chaque équipement du réseau. Cela permet aux administrateurs de réseau de replanifier le réseau en utilisant des programmes de type de contrôle central sans changer l'équipement matériel. Le SDN [1] fournit une nouvelle solution pour contrôler le trafic réseau, et également il fournit une bonne manière de développer un réseau central. Plus important encore, grâce à la séparation des plans de contrôle et de données, cette architecture permet au contrôle du réseau de devenir directement programmable.

Le plan de contrôle spécifie la manière dont les paquets de données sont transférés. Cette partie définit comment les données sont envoyées d'un équipement à un autre. La création d'une table de routage est aussi considérée comme une partie du plan de contrôle. Par exemple, les routeurs utilisent divers protocoles pour identifier les chemins réseau et ils enregistrent ces chemins dans des tables de routage. D'autre part, le plan de données prend en charge la transmission des paquets en accord avec la configuration définie par le plan de contrôle.

Dans les réseaux conventionnels, le trafic de contrôle et le trafic de données sont couplés dans les périphériques réseau. La plupart des fonctions de contrôle sont réparties sur de nombreux appareils. Au contraire, le paradigme SDN [1] permet de retirer les fonctions de contrôle des périphériques réseau et de les consolider dans un contrôleur centralisé. Avec ce modèle, une fois qu'un contrôleur centralisé obtient les règles de transfert, les instructions de transfert des paquets sont téléchargées vers les périphériques réseaux appropriés. Cette communication entre le contrôleur et les périphériques réseau facilite la programmation standardisée des périphériques réseau.

2.2 P4: « Programming Protocol-independent Packet Processors »

Le langage spécifique au domaine des applications réseau connu comme le *Programming Protocol-independent Packet Processors* (P4) [2] est un langage qui permet de décrire la méthode de traitement des paquets dans les périphériques réseau. Il est considéré comme une solution de SDN [1] car il permet de programmer la méthode de traitement de flux sur les équipements de transmission de paquets réseau que ce soit du matériel ou du logiciel. P4 [2] vise trois buts : l'indépendance du protocole, l'indépendance au matériel et la reconfigurabilité pendant le fonctionnement du réseau. Il permet donc de : 1) définir un parseur et un ensemble de *Match Action Table* (MAT), 2) programmer des équipements réseau sans connaître les détails du périphérique de traitement de paquets et laisser les compilateurs configurer le périphérique cible et 3) laisser les utilisateurs modifier le parseur.

Le *Protocol Independent Switch Architecture* (PISA) est une architecture de transfert à pipeline unique supporté par P4 [2]. Il comporte trois types de blocs programmables, le parseur, le MAT et le déparseur. Dans le parseur, les en-têtes doivent être reconnus selon l'ordre et la taille déclarés dans la structure du paquet. Les MAT servent à définir les tables d'appariement et l'algorithme de traitement exact a effectué quand la condition spécifiée est rencontrée. Le déparseur définit le format du paquet de sortie. À part ces blocs, un bus de données contenant les metadonnées traverse les blocs. Il est programmable et représente les informations propres au paquet telles que le port d'entrée/sortie, la longueur de paquet, etc.

Cette architecture permet aux programmeurs de programmer séparément les étapes dans le pipeline avec le langage de haut niveau, P4 [2]. Le compilateur P4, ensuite, sert à commander ce pipeline logique. L'organisation qui soutient le langage P4, le consortium P4, fournit un compilateur basique ciblé, le BMv2 (*Behavioral-Model*) [10], un commutateur P4 logiciel de référence. Les autres fournisseurs de matériel développent leurs propres compilateurs pour appliquer des programmes sur leur plateforme, telle que le *P4-SDNet* [4] de Xilinx.

2.3 Cartes et Plateformes

Le projet de ce mémoire cible deux cartes programmables en exploitant deux plateformes différentes. Le présent chapitre, présente ces cartes et les plateformes.

2.3.1 La carte DE10-Standard et la plateforme d'Intel

La trousse de développement *DE10-Standard* est une plateforme de conception matérielle ciblant les FPGA d'Intel de la classe *System-on-Chip* (SoC). Cette carte est équipée d'un banc FPGA Cyclone V et un processeur câblé (HPS) basé sur l'ARM. Elle est aussi équipée d'une mémoire DDR3 haute vitesse, de modules d'interface vidéo et audio, d'un réseau Ethernet ainsi que d'autres modules qui facilitent la mise en œuvre de solutions applicables à des nombreuses applications.

Dans la plateforme d'Intel, le logiciel Quartus donne les fonctionnalités de programmation de la carte *DE10-Standard*. Ainsi, les interfaces Avalon simplifient la conception du système, elles permettent de connecter facilement des composants au FPGA Intel®. La famille d'interfaces Avalon définit des interfaces appropriées pour la transmission de flux de données à grande vitesse, la lecture et l'écriture de registres et de mémoire, et le contrôle de périphériques dans la puce ou hors de la puce.

2.3.2 La carte NetFPGA-SUME et la plateforme de Xilinx

La carte *NetFPGA-SUME* est une plateforme spécifique très populaire pour la conception de réseaux hautes performances et haut débit. Développée grâce à une collaboration entre *Digilent*, *l'Université de Cambridge* et *l'Université de Stanford*, la carte permet de développer diverses applications autour d'un FPGA configuré pour offrir plusieurs ports Internet de haut débit. Cette carte est équipée un FPGA Xilinx® Virtex®-7 690T, quatre ports SFP+ 10 Gb/s, cinq banques de mémoire haute vitesse indépendant accessibles à partir de divers périphériques compatibles avec les normes 1866 MT/s SoDIMM DDR3 et 500MHz QDRII+. Un PCIe de troisième génération à huit voies et autres connecteurs d'extension, sont également disponibles. Elle peut donc atteindre un débit maximal de 4x10Gbps. Une organisation a été établie pour prendre en charge une large collection de blocs IP (*Intellectual Property*) gratuits développés par une importante communauté qui sont disponibles sur GitHub [11]. Une série d'exemples de projets et leur tutoriel sont aussi fournis par cette organisation pour aider les débutants à maîtriser la plateforme ou à développer leurs propres applications.

Comme le FPGA équipé dans cette carte appartient la famille Xilinx Virtex-7 FPGA, la plateforme Xilinx *Vitis* qui inclut le logiciel Xilinx *Vivado* propose un environnement de programmation. Les interfaces AXI4 sont également offertes sur cette plateforme pour établir les échanges de données.

2.4 *CocoTB* & GHDL

CocoTB [13] est un environnement de banc de test en cosimulation basé sur coroutine (*COroutine based COsimulation TestBench environment*) pour vérifier des modules décrits avec des langages matériels tels que VHDL et SystemVerilog RTL, ainsi qu'à l'aide de Python. Un banc d'essai codé en *CocoTB* est structuré de façon semblable à un banc d'essai VHDL. Le circuit à l'essai, *Design Under Test* (DUT), est instancié en tant que « *Top Level* » dans le simulateur. L'utilisateur définit les données d'entrée pour chaque port d'entrée du DUT. *CocoTB* pilote les entrées appariées dans le simulateur et récupère les sorties. L'utilisateur peut donc observer et traiter les sorties directement depuis un programme Python. Ce système a plusieurs avantages. Premièrement, Python est un langage de haut niveau, donc l'utilisateur n'a pas à maîtriser un langage matériel. Deuxièmement, Python supporte différentes bibliothèques de traitement. À l'aide de la variété des modules logiciels offerts, le traitement de données devient beaucoup plus simple et cela donne aussi la possibilité d'établir un pont de communication entre le banc d'essai et les autres interfaces telles qu'un UART ou une interface Internet.

Le GHDL [28] est un simulateur à sources ouvertes pour le langage VHDL. Il permet de compiler et d'exécuter le code VHDL directement sur PC. Il supporte plusieurs types de sorties, par exemple le format GHW qui est son propre type de fichier de format d'ondes, le format VCD, etc. Il est un des simulateurs supportés par *CocoTB*. Parmi les divers simulateurs tels que Questa [18], le GHDL n'a pas de meilleure performance, mais il supporte tous les fonctionnements dont nous avons besoin. En plus, il est le plus simple à installer et à intégrer avec l'environnement CocoTB. Il n'offre pas une interface graphique, donc il a beaucoup moins de bibliothèques dépendantes à installer en comparaison avec les autres simulateurs. Dans notre projet, le simulateur échange les données avec le CocoTB et certains aspects de l'interface graphique sont donc inutiles.

2.5 « *Open Verification Library* »

Open Verification Library (OVL) [14] est une bibliothèque de modules informatiques pour la vérification des descriptions de circuits numériques écrites dans les langages de description de matériel (HDL). Il est actuellement maintenu par *Accellera*. Dans cette bibliothèque, plusieurs modules permettent de vérifier des propriétés spécifiques du circuit. Ils sont appelés vérificateurs. Ils sont ajoutés aux modules à vérifier et sont liés aux signaux de ce dernier via des ports. Certains aspects de la fonctionnalité du module peuvent être modifiés en ajustant ses paramètres. La bibliothèque fournit des modules de vérification pour différents types de propriétés par exemple une condition qui doit toujours être accordés, une valeur de type spécifique, un signal qui ne doit jamais être activé, un changement de valeur approprié, une synchronisation appropriée de l'événement, etc. L'avantage principal de cette bibliothèque est qu'il permet d'introduire des concepts de vérification de haut niveau dans les modules à vérifier sans nécessité de définir un nouveau langage. Notre système accepte les signaux d'assertion comme les signaux d'entrée. Ces assertions sont insérées dans la plateforme par l'utilisateur. Par conséquent, les vérificateurs proposés par cette bibliothèque nous aident à générer les signaux d'assertion de la manière la plus simple. En comparant avec un autre langage de vérification tel que PSL [29], cette bibliothèque fournit les composants matériels sans forcer l'utilisateur à en décrire la logique. Dans ce mémoire, nous avons utilisé le vérificateur « ovl_always » qui permet de surveiller un signal ciblé. Il sera détaillé dans le chapitre 4.4.6.

CHAPITRE 3 REVUE DE LA LITTERATURE

Cette section présente les articles et les solutions existants concernant les différentes techniques liées au projet.

3.1 Les testeurs de réseau

Cette section détaille les testeurs de réseau commerciaux et les projets de recherche open-source dans ce domaine.

3.1.1 Les testeurs de réseau open-source

Le premier testeur de réseau open-source est “OSNT: Open source network tester” [6] publié par Gianni Antichi et al. en 2014. Il propose la structure de base pour vérifier un dispositif de réseau. Sa structure est composée d’un générateur de paquets et d’un moniteur de paquets. Le système sert à principalement 1) générer les flux de paquets, 2) capturer les flux de paquets et 3) insérer et retirer des marqueurs de temps. Deux structures sont proposées par cet article : l’une implémente le générateur et le moniteur sur deux cartes FPGA séparément et l’autre les rassemble sur une seule carte. Le projet est réalisé sur la plateforme *NetFPGA-SUME*. Un marqueur de temps de 64 bits appelé *timestamp* par la suite est inséré à une position prédéfinie de chaque paquet par le générateur de paquet. Ce marqueur permet de mesurer le temps d’arrivé d’un paquet. Le générateur de *timestamp* dans le moniteur de paquet permet de le retirer et puis de l’analyser. À cause de la fréquence de SPF+ port, 156.25MHz, la résolution du système est limitée à 6.4ns. Un filtre de paquets est implémenté dans le moniteur, permettant d’éliminer les paquets non désirés d’après des règles prédéfinies. Tous les paquets peuvent aussi être transférés à un ordinateur de soutien par DMA (*Direct Memory Access*). Mais cette fonctionnalité a un débit limité, à cause de la performance du module DMA.

Dans le cadre de recherches récentes, l’outil OSNT [6] a été proposé. Dans la même veine, “FlueNT10G : A Programmable FPGA-based Network Tester for Multi-10-Gigabit Ethernet” [15] décrit un testeur qui se concentre sur la relecture précise des traces du réseau. Son générateur de paquets permet de rejouer les flux des paquets préenregistrés ou les flux créés synthétiquement et d’insérer des *timestamps* dans chaque paquet. Le moniteur sert à mesurer le temps d’arrivée des

paquets et à retirer le *timestamp*. Il offre une résolution de 6,4 ns pour les *timestamps* et la mesure de temps. Le système utilise 28 bits pour compter le temps d'arrivée inter-paquets et 32 bits pour le temps de transmission inter-paquets. Le module de filtrage de paquets dans le moniteur fournit un filtre basé sur l'adresse MAC Ethernet prédéfinie. Une interface de programmation d'application (API) est désignée pour automatiser la configuration de DUT, le retour d'informations et l'échange de données entre le FPGA et le PC. Une fonctionnalité optionnelle nommée *"FlueNT10G Agent"* peut être exécutée dans le DUT permettant de faciliter la caractérisation de performance. La portion matérielle de ce système est implémentée sous *NetFPGA-SUME* et le logiciel est écrit sous Go (Google Go) [27].

Un autre testeur à source ouverte, "Formullar: An FPGA-based network testing tool for flexible and precise measurement of ultra-low latency networking systems" [16], fait une contribution sur la précision de temps et le contrôle du trafic. Formullar génère des paquets dans un temps précis, en fonction des différents modèles de trafic. La génération de paquets est contrôlée de manière programmable, ce qui rend les tests automatisés et flexibles. Ensuite, la latence est mesurée en lien avec la synchronisation de la génération et de la réception des paquets. Étant donné que la latence dépend fortement des modèles de trafic, particulièrement la taille des paquets, le débit, les rafales, etc., il y a une demande essentielle de tester les DUT avec des modèles de trafic variables. Le modèle de trafic générique proposé dans ce système permet de configurer divers paramètres par exemple CBR (*constant bit rate*) et VBR (*variable bit rate*). Cette caractéristique garantit la haute programmablité sur le format de paquet. L'implémentation en FPGA fournit une précision en cycle prêt. L'API définie en langage C établit la communication entre un FPGA et un ordinateur de soutien. Bien que le modèle offert permette de configurer ses paramètres, il est difficile de générer certaines caractéristiques désirées à cause de la limitation de vitesse d'échange de données.

Un projet dérivé d'OSNT [6] est "OP4T : Bringing Advanced Network Packet Timestamping into the Field" [7] qui optimise la programmabilité de ses modules de filtrage et la précision des *timestamp*. OP4T offre une architecture matérielle partiellement reconfigurable. Il agit comme un NIC (*Network Interface Controller*) qui échange des traces de paquets entre le serveur et le support physique. Le *timestamp* est inséré après l'en-tête UDP. Fonctionnant à une fréquence maximale de 250 MHz, il peut atteindre une résolution de 4 ns. La contribution principale est son bloc "*Packet Processor*" reconfigurable. Ce bloc définit les règles de filtrage afin que le système offre une

programmabilité avancée. En utilisant le flux de reconfiguration partielle en Xilinx *Vivado*, ce bloc peut être partiellement reprogrammé en temps d'exécution. Il offre aussi un parseur de paquet pour identifier les adresses comme un filtre ou un détecteur d'un signal permettant de réaliser des fonctionnalités particulières. La complexité de ce bloc est cependant limitée par celle du processeur de paquets initial.

3.1.2 Les testeurs de réseaux commerciaux

Plusieurs fournisseurs proposent des testeurs de réseaux commerciaux. Ils sont généralement multifonctionnels, avec une interface d'utilisateur adaptée et ils supportent différents types de protocoles. Ces systèmes sont généralement inabordables pour les groupes de recherche académique ou les petites entreprises.

Le testeur IxNetwork [5], fabriqué par Keysight, offre une couverture de test de 1G à 400G Ethernet. Il permet de générer des flux de trafic qui imitent des applications et des scénarios de test. IxNetwork supporte aussi des environnements réseau virtualisés et peut s'exécuter à partir de tout environnement opérationnel disponible. De plus, il fournit une couverture de protocoles complète pour le routage et la commutation. Une analyse approfondie du flux de trafic est aussi appliquée. Ce testeur est assez puissant, mais le prix de l'appareil est d'environ $10,000 et son fonctionnement nécessite une licence d'environ $5,000 par an. Spirent [17] est une entreprise qui offre des outils et services de mesure de la performance des réseaux. Mais les solutions qu'elle offre visent souvent au matériel sur mesure qui n'est pas convenable au budget des instituts de recherche ou des petites entreprises.

3.2 Les outils de débogage

Cette section présente les outils de débogage existant incluant les simulateurs logiciels, les méthodes embarquées et les projets avec la coopération matérielle/logicielle.

3.2.1 Les simulateurs RTL

Les modèles dits *Register-Transfer Level* (RTL) soutiennent une abstraction de la conception qui modélise un circuit numérique synchrone en termes de flux de signaux échangé entre les registres et les opérations logiques effectuées sur ces signaux. Il est principalement utilisé dans les langages

de description de matériel (HDL) tels que Verilog et VHDL afin de créer des représentations d'un circuit à un niveau plus élevé que les simples réseaux logiques exprimés au niveau *porte logique* communément appelées *netlist*. Au premier stade de création d'une description RTL dit présynthèse, les simulations ne permettent généralement pas de valiser les contraintes de temps.

Actuellement, divers simulateurs RTL offerts commercialement ou à sources ouverte sont proposés par des fournisseurs de matériel. Du côté commercial, Questa [18] est l'un des simulateurs les plus populaires. Il offre un environnement qui soutient plusieurs langages permettant de spécifier la fonctionnalité de modules matériel. Questa [18] est développé et commercialisé par Siemens. Il permet notamment de simuler des modules exprimés à l'aide de langages de description de matériel tels que VHDL, Verilog et SystemC, et il inclut un débogueur de langage C intégré. Ce simulateur peut être utilisé indépendamment ou en coopération avec Intel Quartus Prime, PSIM, Xilinx ISE ou Xilinx *Vivado*. Une interface utilisateur graphique (GUI) est fournie pour afficher les formes d'ondes. Les grands fournisseurs de matériel proposent également leurs propres simulateurs, par exemple *Quartus II Simulator (Qsim)* et *Xilinx Simulator (XSIM)*.

Les simulateurs open-source supportent aussi la simulation des langages matériels. Mais ils sont normalement spécifiques pour un seul langage, par exemple, Verilator compile seulement le Verilog et le GHDL de limite au VHDL. Ils ne sont peut-être pas les meilleurs choix, car ils offrent des caractéristiques limitées en comparaison des simulateurs commerciaux. En plus, ils ne fournissent pas d'interface graphique, mais ils génèrent seulement les formes d'onde en sortie sous forme de fichiers.

3.2.2 Les analyseurs logiques embarqués (ELA)

Les analyseurs logiques embarqués sont également utilisés pour le débogage. Ces blocs analyseurs logiques sont insérés dans le module à vérifier, en utilisant les ressources d'un FPGA cible. Ces blocs permettent de facilement sélectionner et de capturer des signaux qui reflètent son fonctionnement.

Les analyseurs logiques embarqués commerciaux sont généralement fournis par les environnements de développement de FPGA ou les outils de simulation. Citons à titre d'exemple, le Signal Tap II [8] proposé par Intel dans Quartus et Chipscope [9] proposé par Xilinx. L'accès à

ces analyseurs pour les contrôler et y transférer des données se fait généralement via un port JTAG standard afin de simplifier les exigences d'interface. Les signaux capturés peuvent être affichés sur un PC à l'aide d'un logiciel de visualisation, présentant les formes d'ondes via un simulateur logique. L'avantage de cette approche est qu'elle permet de visualiser des signaux recueillis dans le matériel en utilisant seulement l'interface JTAG standard. En plus, ils sont intégrés aux environnements de développement et ont une interface d'utilisateur complète. Donc les utilisateurs peuvent y accéder très facilement. Cependant, l'un des inconvénients de cette approche est qu'une grande quantité de ressources FPGA est requise. Ce surcoût est significatif sur les petites plateformes ou lorsque les mémoires sont occupées. Ainsi, une recompilation est requise après chaque modification de la sonde et cela peut prendre beaucoup de temps si le module à tester est complexe. De plus, en raison de la logique supplémentaire ajoutée, les contraintes de performance temporelle peuvent être plus difficiles à rencontrer. Une licence est aussi nécessaire pour avoir la permission d'utiliser analyseurs embarqués.

Plusieurs chercheurs proposent leurs ELA adaptés aux différentes conditions d'utilisation. Zet et Fosalau [22] proposent un analyseur basé sur FPGA. L'analyseur acquiert les données par les ports d'entrées et les enregistre dans les FIFOs. La communication avec le PC de commande s'effectue par le port série. Elle est bidirectionnelle afin que le PC puisse envoyer les données de configuration vers l'analyseur. Cette solution supporte la configuration de la fréquence d'horloge. Une interface graphique basée sur Labview [32] est utilisée pour configurer l'analyseur et pour afficher les chronogrammes. Notons que la fréquence d'échantillonnage maximale est de 25MHz. Pour résoudre la limitation de vitesse des communications via JTAG ou USB, Popa et al. [23] proposent un analyseur basé sur un réseau de haut débit. La communication se réalise par les paquets passant le 10Gbps Ethernet. Ces auteurs définissent un protocole spécial pour transférer les données d'échantillonnage, l'horloge et les informations de déclenchement, etc. L'inconvénient principal de cette solution est qu'elle consomme beaucoup de ressources et qu'elle utilise le port Ethernet à 10Gbps pour transférer les résultats.

3.2.3 Combinaison de simulation et d'exécution

Attia et Betz ont publié un article présentant leur outil de débogage en combinant la simulation et l'exécution sur matériel. Ces travaux sont documentés dans l'article “StateMover: Combining

Simulation and Hardware Execution for Efficient FPGA Debugging” [19]. Cet outil relie un simulateur et un FPGA, pour permettre le débogage dans le système avec une observabilité et une contrôlabilité complète. Aussi, il supporte l'interruption sécurisée des tâches d'un FPGA et le transfert des états vers un simulateur s'exécutant sur un hôte. En inverse, cet outil extrait l'état du simulateur et l'écrit sur le FPGA pour l'exécution sur le matériel. L’implémentation proposée réalise une précision au cycle prêt et garde le temps de téléchargement des états entre le FPGA et le simulateur à l’intérieur de quelques secondes. Cette idée a également été adoptée dans Simpoint proposé par l’article “Coupling Simulation and Hardware for Interactive Circuit Debugging”[20] pour le débogage du matériel analogique.

3.3 Retour sur la problématique

Dans l'ensemble, le domaine des outils de débogage reste toujours l'un des points chauds de la recherche. Les chercheurs et les fabricants proposent diverses solutions adaptées à différents besoins. Le développement des architectures réseaux, tel que SDN [1], apporte un aspect de programmabilité qui introduit de nouveaux défis de débogage. Les outils existants ne s’adressent pas aux applications réseau [19] [20] [22] [23] et n’arrivent pas à accéder aux signaux internes dans la mise en œuvre matérielle [6] [7] [15] [16]. Dans la suite de ce mémoire, nous en présentons les contributions. Nous proposons un environnement de test, composé d'une partie logicielle et d’une partie matérielle, spécifié pour les équipements réseaux.

CHAPITRE 4 ARCHITECTURE DE L'ENVIRONNEMENT DE TEST

Ce chapitre introduit une architecture d'environnement de test combinant une partie logicielle et une partie matérielle pour vérifier les applications réseau configurables et reprogrammables basées sur FPGA. Avec les interfaces créées, l'environnement doit être capable de réaliser la communication entre un PC et l'application à tester, implémentée dans un FPGA. Ces canaux de communication permettent de passer d'abord un test dans le simulateur logiciel et puis de le passer dans la carte FPGA. Par conséquent, le comportement logique et l'exigence de synchronisation peuvent être vérifiés.

4.1 Architecture générale

L'architecture générale suit la structure de système de vérification de programme P4 proposée par Luinaud [25], un collaborateur de la Chaire KIN. Comme présenté dans la figure 4.1, deux modes de test peuvent être appliqués sur le DUT. Le flux de vérification proposé et mis en œuvre dans le banc de test consiste à générer un flux de paquets à tester et à l'appliquer sur les ports d'entrée dans le simulateur logiciel. Cette partie du système de vérification obtient les résultats de simulation. Ensuite, le flux de paquets produit par le simulateur est envoyé au FPGA qui implémente l'interface vers le DUT matériel pour obtenir les résultats produit par le prototype en matériel sur FPGA. Les deux résultats peuvent finalement être comparés par un module de comparaison qui se trouve dans l'environnement logiciel sur un ordinateur hôte.

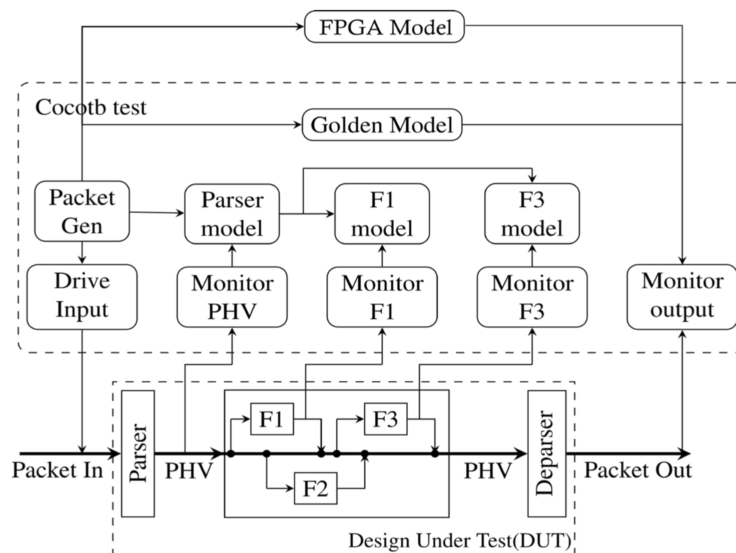


Figure 4.1 : Architecture proposée par Luinaud [25]

La figure 4.2 présente l'architecture du programme Python. Il est le centre du système et prend en charge le test logiciel. Il sert à : 1) générer les paquets à tester, 2) les envoyer au simulateur RTL, 3) générer un modèle de référence pour comparaison, 4) envoyer les paquets générés au FPGA, 5) récupérer les sorties du simulateur et du prototype matériel sur le FPGA et 6) comparer les données de sortie. Les données récupérées ou générées par chaque module sont enregistrées dans des fichiers, donc quatre fichiers sont finalement créés.

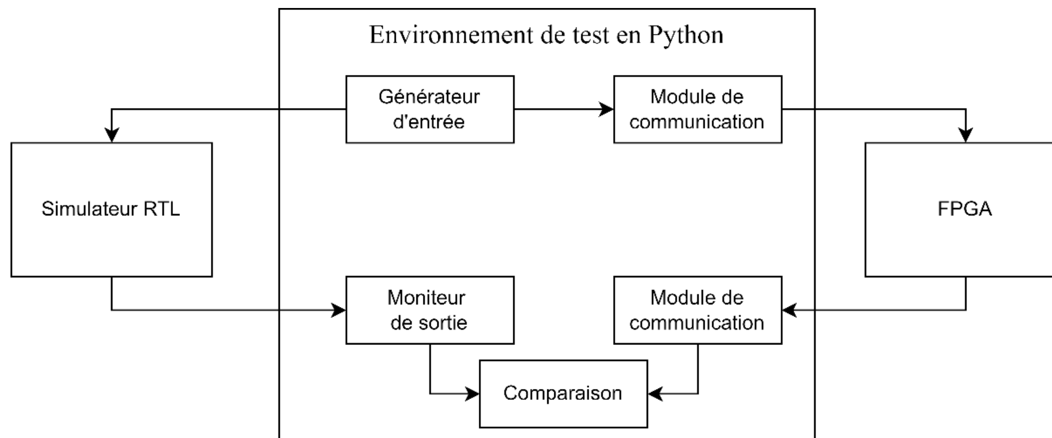


Figure 4.2 : Architecture de la partie logicielle

La figure 4.3 présente l'architecture du côté matériel. Le module de communication envoie les données vers un processeur embarqué dans le FPGA. Ensuite, le processeur écrit les données dans une interface d'entrée. Cette interface enregistre les données dans une FIFO. Une fois que toutes les données sont stockées, la sortie de la FIFO se déclenche par un signal de contrôle. Les données traversent d'abord le DUT pour être ensuite récupérées par une interface de sortie. Cette interface contient aussi une FIFO. Finalement, le processeur lit les données de sortie et les renvoie vers le PC.

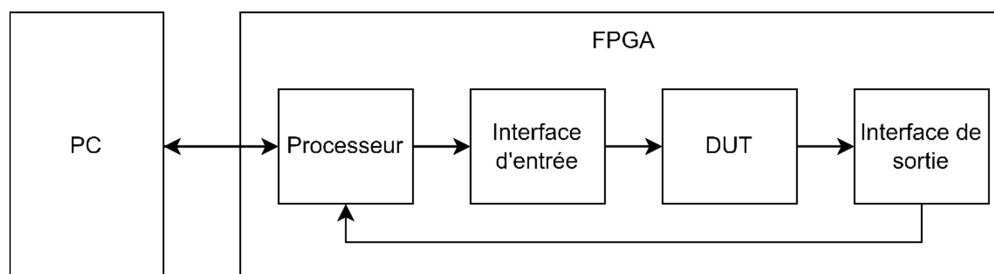


Figure 4.3 : Architecture de la partie matérielle

4.2 Interfaces d’entrée et de sortie

L’interface d’entrée et l’interface de sortie servent à établir la communication entre le processeur et le DUT. Ces interfaces sont basées sur des FIFOs permettant d’enregistrer les données.

Pour communiquer avec le processeur, les blocs sont équipés d’un bus de données accessible en mémoire. Ce bus est commandé via les signaux de contrôle « lire » et « écrire » qui permettent d’identifier l’opération. Les signaux « données » et « adresse » servant à transférer les données et les adresses correspondantes. Cette adresse détermine dans quel registre la donnée se trouve. Avant la compilation du DUT, une adresse de base unique est distribuée à chaque bloc équipé de ce type de bus. Du côté processeur, les blocs sont considérés comme les blocs de mémoire. La fonction d’écriture prend une adresse et une donnée comme paramètres. Cette adresse est la somme de l’adresse de base et de l’adresse du registre. La fonction de lecture prend seulement une adresse et renvoie une donnée. La figure 4.4 présente un exemple de deux interfaces. Si une donnée doit être écrite dans le registre R1 de l’interface d’entrée, la fonction d’écriture doit contenir la donnée et l’adresse « 0x000A+1 ».

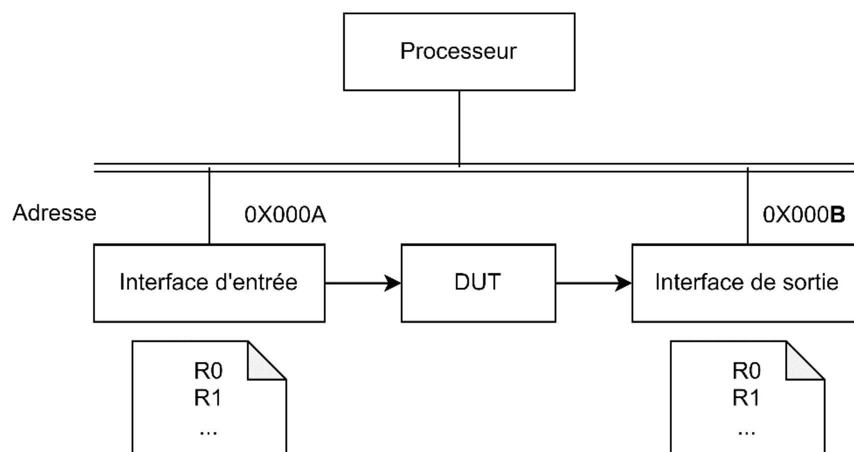


Figure 4.4 : Exemple de bus d’interface mappée en mémoire

Pour passer les données au DUT, un bus de flux est utilisé. Il permet de transférer les données à chaque cycle d’horloge de façon unidirectionnelle. Ce bus contient les signaux suivants : « *ready* », « *valid* » et « *data* ». Le signal « *ready* » est piloté par la destination et le signal « *valid* » est piloté par la source. Ce sont les signaux de contrôle qui réalisent le « *handshake* » afin d’indiquer si les données sont valides ou pas. Lors de l’activation de lecture de la FIFO, les données sortent de la FIFO à chaque cycle d’horloge. Cela est cohérent avec le bus de flux. Donc, pour l’interface

d'entrée, la sortie de la FIFO est connectée directement avec le signal « *data* ». Lorsque le DUT est prêt à traiter les données (« *ready* » activé) et que le passage de donnée est déclenché, la lecture de la FIFO s'effectue et le « *handshake* » s'établit (« *valid* » activé).

Comme le processeur supporte des données de 32 bits, l'interface d'entrée prend en charge de combiner les données de 32 bits à la largeur de bus de flux. Cela est réalisé par l'adresse de registre. Le registre 0 est réservé pour la lecture de niveau de remplissage de FIFO. Les autres registres sont utilisés pour identifier la position d'une donnée dans un bus. Le détail de la distribution de registre est présenté dans les sections suivantes.

4.3 L'environnement CocoTB

CocoTB est un environnement qui fournit des méthodes pour assigner les données aux ports d'entrée du DUT et collecter les données à sa sortie. Ce simulateur offre des coroutines permettant de réaliser des travaux en parallèle.

Trois fonctions sont créées sous sa bibliothèque : le test principal, un moniteur d'entrée et un moniteur de sortie. La fonction de test principale définit la structure du test. Elle contient les générateurs d'horloge, les mécanismes d'activation des moniteurs, l'assignation des données et finalement l'appel du module de test mis en œuvre dans un FPGA.

4.4 L'architecture de la plateforme d'Intel

La présente section décrit l'environnement de test implémenté sur la plateforme d'Intel.

4.4.1 Les interfaces utilisées

La famille *Avalon* est utilisée dans la plateforme d'Intel ciblée.

4.4.1.1 « Avalon Streaming »

Le bus *Avalon Streaming* est une interface qui prend en charge un flux unidirectionnel de données. Les données sont transférées de la *Source* au *Drain* sous le contrôle de certains signaux. Le modèle standard de ce bus contient simplement trois signaux de commande et le mode de paquet ajoute les autres signaux nécessaires pour identifier les éléments importants des paquets transmis, par exemple le début de paquet (SOP) et la fin de paquet (EOP), etc. Les détails de cette interface sont

présentés dans la figure 4.5. Le signal vide (*empty*) facultatif indique le nombre d'octets qui sont vides dans le cycle de fin de paquet. Deux paramètres *readyLatency* and *readyAllowance* sont utilisés pour configurer la règle. Dans le cas de ce mémoire, les deux paramètres sont mis à zéro, ce qui signifie que les transferts de données ne se produisent que lorsque « *ready* » et « *valid* » sont détectés. La figure 4.6 représente le chronogramme de l'interface *Avalon Streaming* standard. Les signaux « *error* » et « *channel* » sont optionnels, ils ne sont pas utilisés dans le projet.

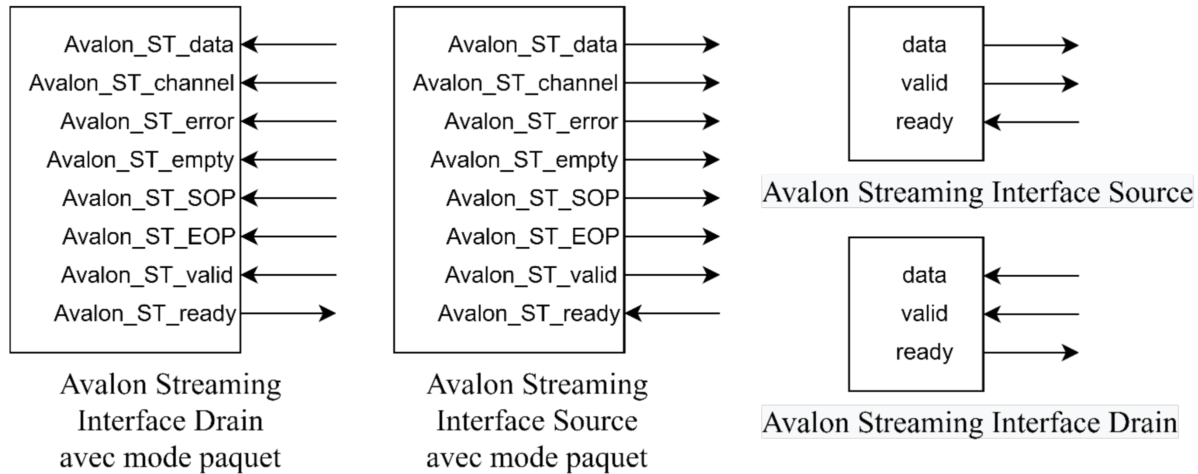


Figure 4.5: L'interface *Avalon Streaming*

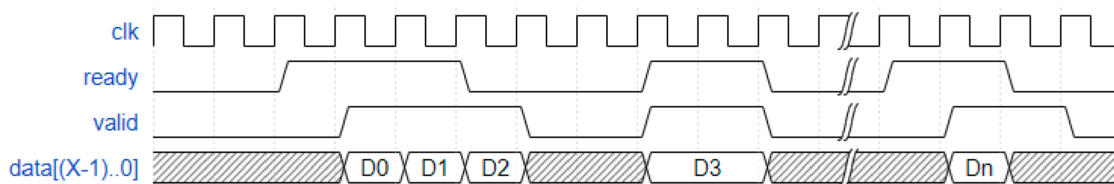


Figure 4.6 : Chronogramme de l'interface *Avalon Streaming*

4.4.1.2 Interface Mappée en Mémoire

Le bus *Avalon Memory Mapped Interface* (Avalon-MM) est une interface de lecture/écriture basée sur les adresses typiques des connexions entre les périphériques maitre-esclave. Cette interface complète inclut plusieurs signaux permettant de contrôler toutes les fonctionnalités de lecture/écriture. Mais pour faire fonctionner cette interface, il n'est pas nécessaire d'utiliser tous les signaux existants. Les signaux nécessaires sont *address*, *readdata* et *read* pour une interface en lecture seule, ou *address*, *writedata* et *write* pour une interface en écriture seule. Dans le projet

présenté dans ce mémoire, le signal *waitrequest* est aussi ajouté dans l'interface pour indiquer la situation d'occupation du bus de données.

4.4.2 Détails de l'architecture

La carte utilisée pour cette plateforme est la *DE10-Standard* (Section 2.3.1). Cette carte possède un seul port Ethernet connecté avec le processeur matériel embarqué dans le FPGA. Ce système permet donc d'un côté de configurer l'interface Internet et de récupérer les paquets et de l'autre côté, d'échanger des données avec les périphériques implémentés dans le FPGA. La communication entre l'ordinateur hôte et le processeur matériel embarqué appelé HPS se réalise par des *sockets*. D'autre part, l'interface entre le HPS et le bloc d'interface au DUT est faite via des lectures et écritures en mémoire.

La figure 4.7 décrit en détail la partie FPGA du système et le langage utilisé dans les différents programmes.

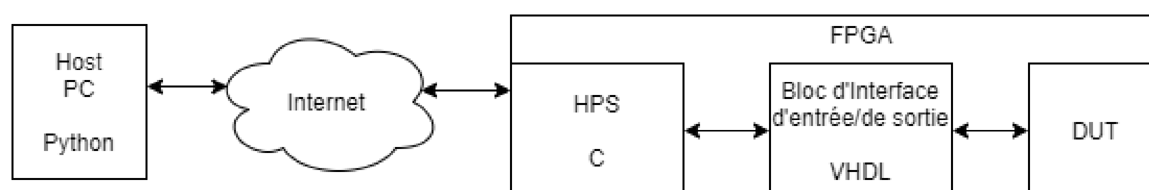


Figure 4.7 : Programmes et langages utilisés sur la plateforme d'Intel

4.4.3 Mise en œuvre du module matériel

Cette section décrit les détails de la manière d'implémenter le module en matériel. Le bus *Avalon Streaming* permet de transférer des données à chaque cycle d'horloge, mais la communication en *socket* et les opérations en mémoire sont beaucoup plus lentes. Une interface basée sur des FIFOs est donc insérée pour stocker les données et synchroniser l'envoi de données. Une caractéristique importante des applications réseau telles que les routeurs ou les commutateurs sont leurs ports d'entrée et de sortie multiples. Cependant, un seul port Internet est disponible et il peut seulement communiquer avec le processeur ARM. Pour simuler des scénarios différents, les interfaces peuvent être considérées comme des ports d'entrée ou de sortie. La figure 4.8 présente un modèle qui permet de simuler un environnement de N ports. Les données sont préremplies dans chaque interface et les sorties de FIFO sont bloquées par une barrière de synchronisation. Une fois que les

données à injecter sont prêtes, un signal « GO » est envoyé par l'utilisateur. Ce signal sert à faire passer les données dans toutes les interfaces au DUT en même temps.

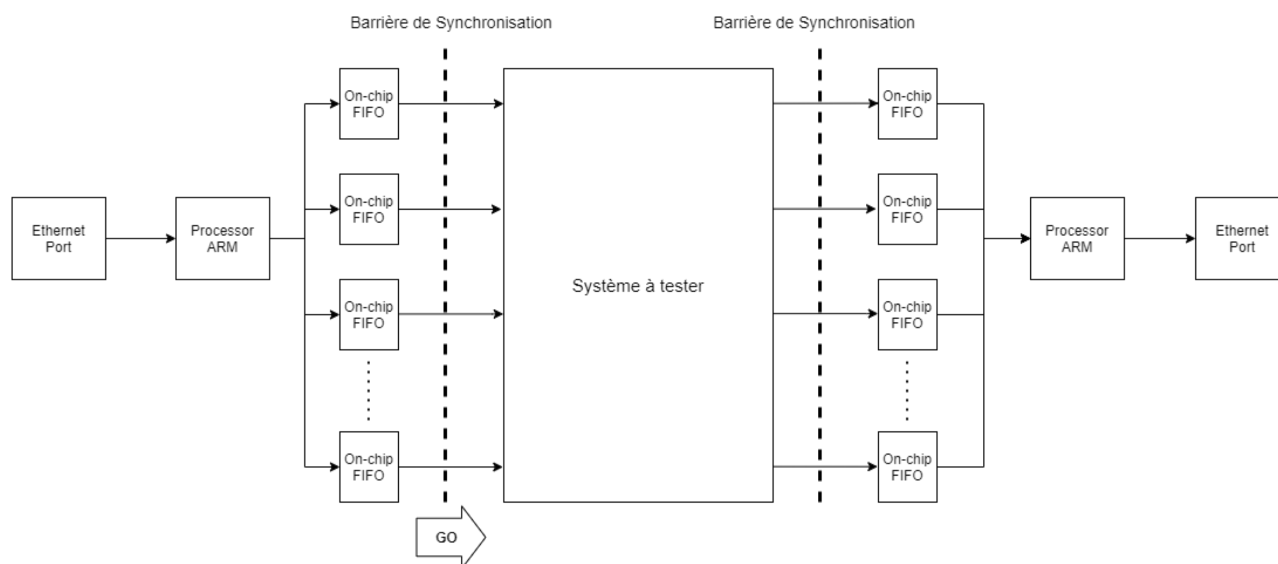


Figure 4.8 : Architecture matérielle sur la plateforme Intel

4.4.3.1 Les interfaces d'entrée et de sortie

Cette interface établit le lien entre le DUT et le HPS. Comme présenté dans la figure 4.9, l'interface d'entrée est placée à l'entrée du DUT permettant de combiner les données en format 32 bits écrit par le HPS aux bus *Avalon Streaming* configurables. De l'autre côté, l'interface de sortie récupère les données du bus *Avalon Streaming* après le DUT, puis les organise en mots de 32 bits.

Le bus de communication entre l'HPS et l'interface est de type *Avalon Memory Mapped*. Comme expliqué dans la section 4.2 et 4.4.2, chaque fois qu'une donnée valide est écrite dans ce bloc, il contient deux informations : une donnée en 32 bits et une adresse en 10 bits. Le bus *Avalon MM* est fixé à 32 bits, donc pour un bus *Avalon Streaming* de X bits (multiple de 32), au moins X/32 écritures sont nécessaires.

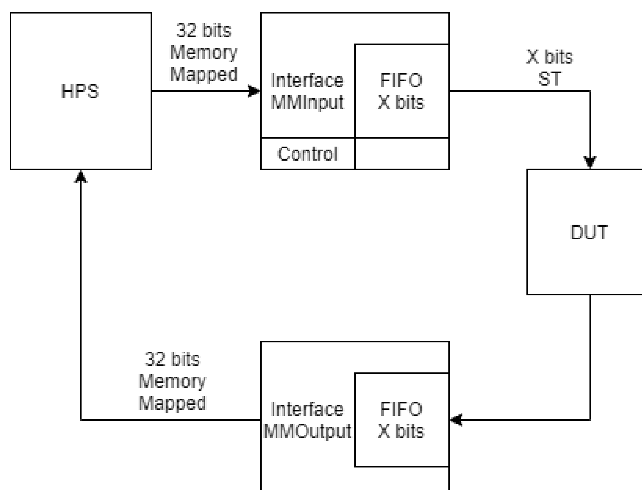


Figure 4.9: Modèle des interfaces pour un canal

Pour une application réseau, si les données à tester sont sous la forme d'un paquet, l'interface *Avalon Streaming* en mode paquet est utilisée. Donc, il faut aussi passer les informations des signaux de début et de fin de paquet telles que SOP et EOP à l'interface d'entrée. Afin d'éviter le dépassement de la profondeur de la FIFO, le HPS doit être capable de lire le niveau de remplissage. La distribution de l'adresse pour le bloc d'interface d'entrée est présentée dans le tableau 4.1. L'organisation du signal de contrôle est détaillée dans la figure 4.10.

Tableau 4.1 : Distribution d'adresse de l'interface d'entrée avec bus de 128 bits

Adresse	Type	Données
0	Lecture	Niveau de remplissage
1	Ecriture	Signal de contrôle
2	Ecriture	Data[127..96]
3	Ecriture	Data[95..64]
4	Ecriture	Data[63..32]
5	Ecriture	Data[31..0]

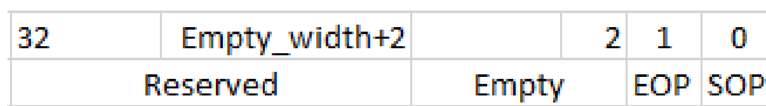


Figure 4.10 : Organisation des bits du signal de contrôle

La FIFO devrait enregistrer tous les signaux requis sauf les signaux de contrôle « *ready* » et « *valid* ». Sa largeur est donc la somme du nombre de bits des signaux, arrondi à un multiple de 8. Le bloc FIFO génère cette limitation. Prenant comme exemple un bus de données de 128 bits, l'arrangement des bits est présenté dans la figure 4.11.

00 143..142	data 141..14	empty 13..10	EOP 9	SOP 8	channel 7..3	error 2..0

Figure 4.11 : Arrangement des bits d’un bus de données de 128

Chaque fois qu’une donnée est écrite, elle est décalée à la position selon son adresse. Pour une donnée à l’adresse X, le nombre de bits à décaler est

$$32 \times (\mathrm{MAX_ADDR} + 1 - \mathrm{X}) = (\mathrm{MAX_ADDR} + 1 - \mathrm{X}) \ll 5$$

où le paramètre $\mathrm{MAX_ADDR} = DATA_BUS_WIDTH \div 32$

Par exemple, une donnée à l’adresse 2 dans un bus de 128 bits est traduite en effectuant un décalage à gauche de 96 bits, correspondant à la position [127..96]. Un compteur s’incrémente à chaque fois qu’une donnée valide est écrite. Le compteur est forcé à (MAX_ADDR -1) si le bit de fin de paquet est écrit pour terminer l’écriture de données. Lorsqu’il y a assez de données écrites (par exemple, le compteur est égal à 4 pour un bus de 128 bits), l’écriture dans la FIFO est activée, ce qui complète un cycle de données dans la FIFO. La machine à états utilisée dans ce processus est présentée à la figure 4.12.

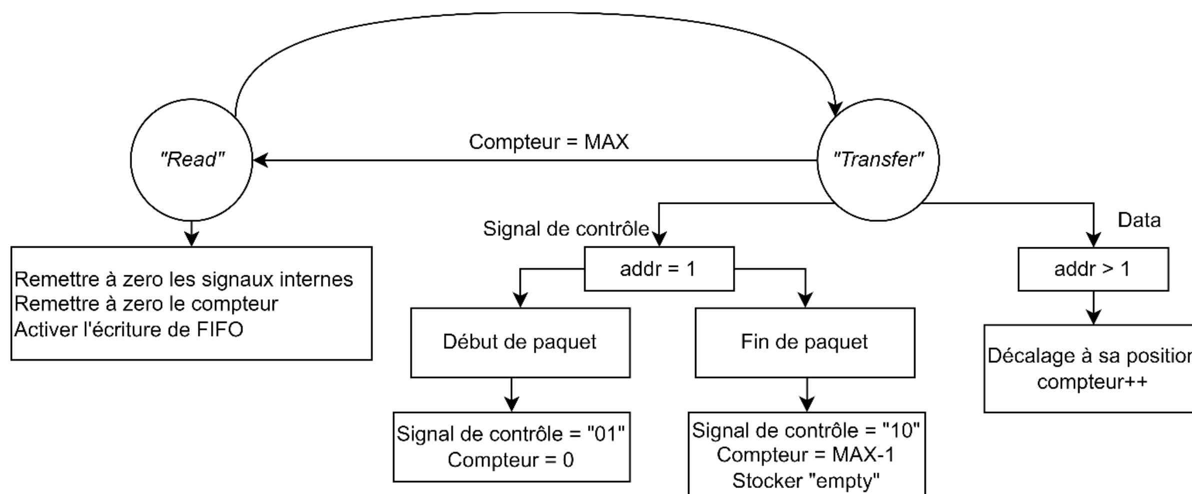


Figure 4.12 : Logique de l’écriture de l’interface d’entrée

L’interface de sortie suit le même principe de fonctionnement que l’entrée, mais elle prend les données enregistrées dans la FIFO de format *Avalon Streaming* et les transfère en mots de 32 bits qui sont prêts à lire par le HPS. La distribution des adresses est similaire à celle présentée dans le tableau 4.1, sauf que le type d’opération est changé d’écriture à lecture. De plus, le décalage à

gauche fait dans l'interface d'entrée devient un décalage à droite. Une différence importante se trouve au niveau de la lecture de la FIFO : dans cette interface, lors d'une lecture à la dernière partie de données, une lecture de la FIFO s'effectue.

4.4.3.2 Le programme dans le HPS

Du côté de HPS, au début du programme, l'adresse correspondante à chaque interface est calculée selon l'adresse de base du périphérique. Cette adresse est distribuée automatiquement par le SDE Qsys. Après ce transfert de l'adresse, la lecture et l'écriture dans les interfaces se traduisent comme des opérations de tableau de mémoire.

4.4.4 La communication avec « socket »

Cette section explique la communication avec socket entre le HPS et un ordinateur hôte (PC). La figure 4.13 illustre le flux de travail. Le programme serveur s'exécute dans le HPS et attend une connexion. Chaque bloc de l'interface est assigné à un identificateur (ID), par exemple « Input 0 », « Output 0 ». Après avoir vérifié l'ID, un socket est associé à une interface.

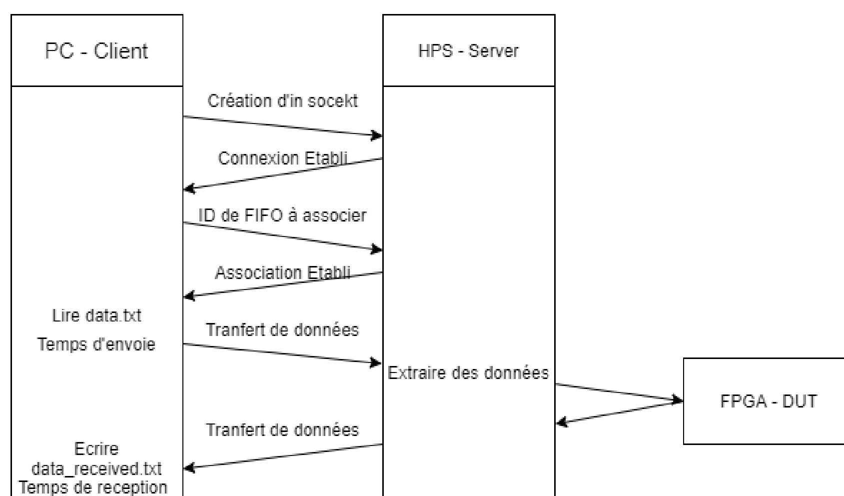


Figure 4.13: Process de communication en *socket*

4.4.4.1 Au niveau du PC

Pour éviter la confusion des données dans chaque *socket*, la bibliothèque dite de « *threading* » [30] est utilisé. Les opérations en *socket* peuvent être réalisées en parallèle. La fonction de l'interface d'entrée lit les données d'entrée dans le fichier crée par le générateur de paquets. Il reforme ces données sous un format d'une suite de paquets contenant les données à transmettre. La figure 4.14

présente le format de données dans le fichier et celui dans le *socket*. Les « paquets » sont séparés par « ; », chaque ligne de ce fichier correspond à un paquet à envoyer. Ce format permet de réduire le nombre de paquets échangé entre la carte et le PC, en plus de définir individuellement les données, sans être influencé par les protocoles de transmission. La figure 4.15 explique le format des données dans le *socket*. Plus précisément, le programme lit les données dans le fichier, puis calcule la taille. La plage 32 bits avant chaque paquet est réservée pour indiquer sa taille, permettant ainsi au programme du HPS de bien identifier les paquets de données.

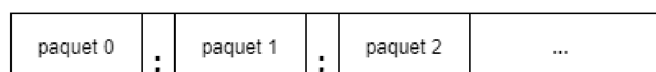


Figure 4.14: Format de données dans un fichier d'entrée

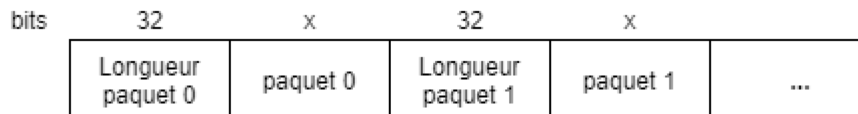


Figure 4.15 : Format de données dans le *socket*

Pour récupérer les données venues du FPGA, une fonction prend en charge d'extraire les données venues du *socket* puis les enregistre dans le fichier de sortie prédéfini en respectant les formats mentionnés dans la figure 4.14.

4.4.4.2 Au niveau du « HPS »

Du côté du HPS, lorsqu'il a reçu une nouvelle connexion, un fil est créé pour faire les opérations suivantes. Dans ce fil, les données sont extraites du *socket* et puis elles sont séparées en identifiant chaque paquet. Le programme envoie ensuite les données paquet par paquet avec le signal de contrôle vers l'interface d'entrée. Pareillement, du côté sortie, après avoir établi l'association, le fil lit continuellement le niveau de remplissage de la FIFO dans l'interface de sortie. Dès qu'il y a des données dans la FIFO, il les lit, les reforme selon la figure 4.15 et les renvoie vers le PC. Ce mécanisme peut être réalisé avec l'interruption, mais le fil dans ce cas attend seulement la réception des données donc l'interruption n'est pas nécessaire.

4.4.5 Le compteur de cycles

Pour mesurer la durée utilisée par le DUT, un compteur détermine le temps à l'échelle du cycle d'horloge. Une entrée supplémentaire est ajoutée dans l'interface d'entrée et de sortie. Un processus dans l'interface d'entrée permet de contrôler le temps de passage des données et aussi la durée entre deux paquets en accord avec le signal « GO ». D'autre part, le paramètre *STARTPASS* marque le cycle de passage des données, et celui de *InterPac* définit la différence de temps entre deux cycles de données. La figure 4.16 présente la machine à états qui met en œuvre ce mécanisme. Si le paramètre *InterPac* est égal à 0, les données passent continuellement. Sinon, un compteur compte le nombre de cycle entre le passage de deux données et fait passer la donnée prochaine dès que le nombre de cycles est atteint.

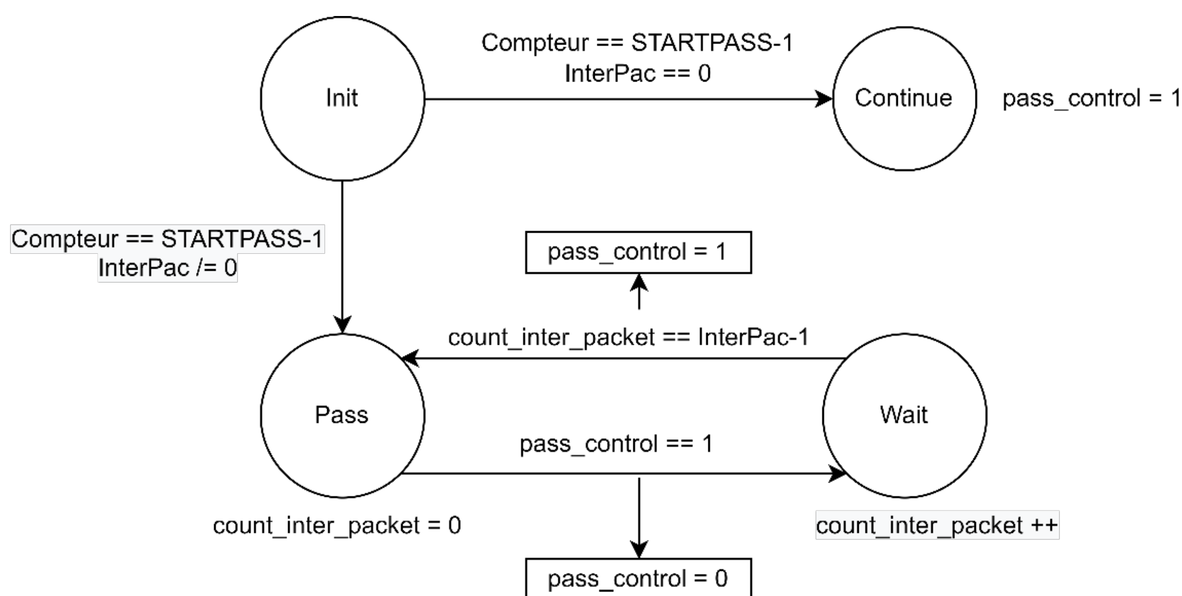


Figure 4.16 : Machine à états qui permet de caractériser le passage des données

Dans l'interface de sortie, une nouvelle FIFO enregistre la valeur de compteur lors de l'arrivée d'une donnée valide. Le programme HPS lit ces données de temps à l'adresse suivante.

4.4.6 Modification de l'interface pour vérifier la « MAT »

Pour appliquer cet environnement dans un scénario réel, le MAT est utilisé comme le DUT. Cette section présente les modifications faites pour intégrer ce module dans l'environnement de test.

4.4.6.1 Présentation de la « Match Action Table »

Comme mentionné précédemment, le groupe de recherche vise actuellement au développement d'applications réseau programmables. Une MAT réalisée a été mise en œuvre par Richer St-Onge [26], un collaborateur de la Chaire KIN. Il avait besoin d'en faire la vérification. Nous l'avons testé avec l'environnement présenté ci-haut.

Cette MAT est un bloc basé sur de la mémoire permettant de reconnaître les clés ciblées et de sortir les résultats correspondants. Dans une application réseau, les paquets entrants sont premièrement traités par le parseur permettant d'identifier les protocoles. Les entêtes de protocoles sont extraits sous un format défini par le vecteur d'entête de protocole (PHV). La MAT prend ensuite ces vecteurs comme entrée et recherche les informations correspondantes telles que le port de sortie ou le protocole suivant. Il supporte deux modes de fonctionnement qui sont configurés par certains signaux d'entrée. Dans le mode d'insertion, le bloc récupère la clé et la valeur présentées aux entrées et les enregistre dans le tableau de mémoire. Dans le mode de recherche, il prend la clé d'entrée et cherche dans le tableau de mémoire. Une fois cette clé trouvée, la valeur appariée est présentée à la sortie accompagnant avec un signal « *hit* » activé. Dans ce mémoire, la MAT implémentée utilise principalement une « CAM » qui est une mémoire adressable par le contenu. De plus, la MAT permet de prédéfinir des tableaux de type *On-Chip RAM* « OCRAM » pour initialiser le bloc.

L'interface de ce bloc et la configuration des modes sont présentées à la figure 4.17 et dans le tableau 4.2.

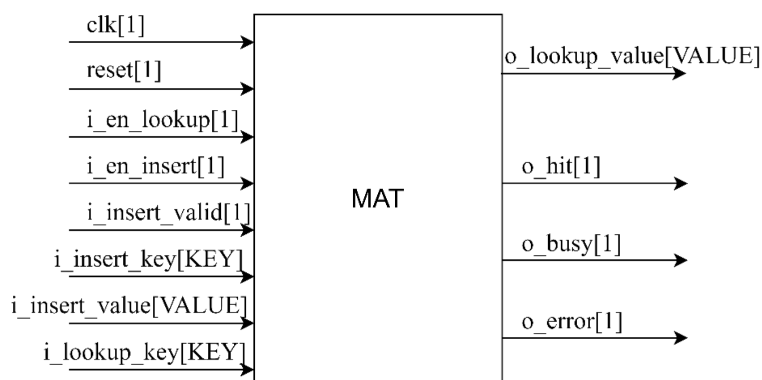


Figure 4.17: L'interface de bloc MAT

Tableau 4.2: La configuration des modes de MAT

	Insertion	Recherche
i_en_lookup	-	X
i_en_insert	X	-
i_insert_valid	X	X
i_insert_key	X	-
i_insert_value	X	-
i_lookup_key	-	X

4.4.6.2 L'unité « *Match Action* »

Comme les données d'entrée dans ce cas sont les vecteurs PHV, le mode paquet est inutile donc les signaux de paquets sont enlevés. Afin de connecter les ports d'entrée avec le bus *Avalon Streaming,* une interface « *Match Action Unit* » (MAU) est créée en enveloppant la MAT comme présenté à la figure 4.18. Elle sert à connecter les ports de la MAT aux bus de données *Avalon Streaming* et à générer les signaux de contrôle « *ready* » et « *valid* ». La taille des signaux « *Key* » et « *Value* » sont configurables, mais dans ce mémoire, ils sont fixés à 48 et 8. Donc la somme des nombres de bits des ports d'entrée est de 107. La taille du bus d'*Avalon Streaming* est donc mise à 128 bits. L'organisation des signaux dans le vecteur est illustrée à la figure 4.19.

À part de cette interface ajoutée, une autre modification est apportée à l'interface d'entrée. Dans ce cas, à la fin des transferts des données en *socket*, les données devraient être envoyés vers le DUT pour une interface qui contient un seul canal. Pour simplifier le mécanisme de contrôle, le signal « GO » est intégré dans cette interface. Donc, dans le programme du HPS, une fois que les données sont écrites dans l'interface, une valeur 0x1 est écrite à l'adresse maximale : 0x3FF. Du côté de l'interface, dès qu'elle a reçu cette donnée, elle fait passer les données vers le MAU.

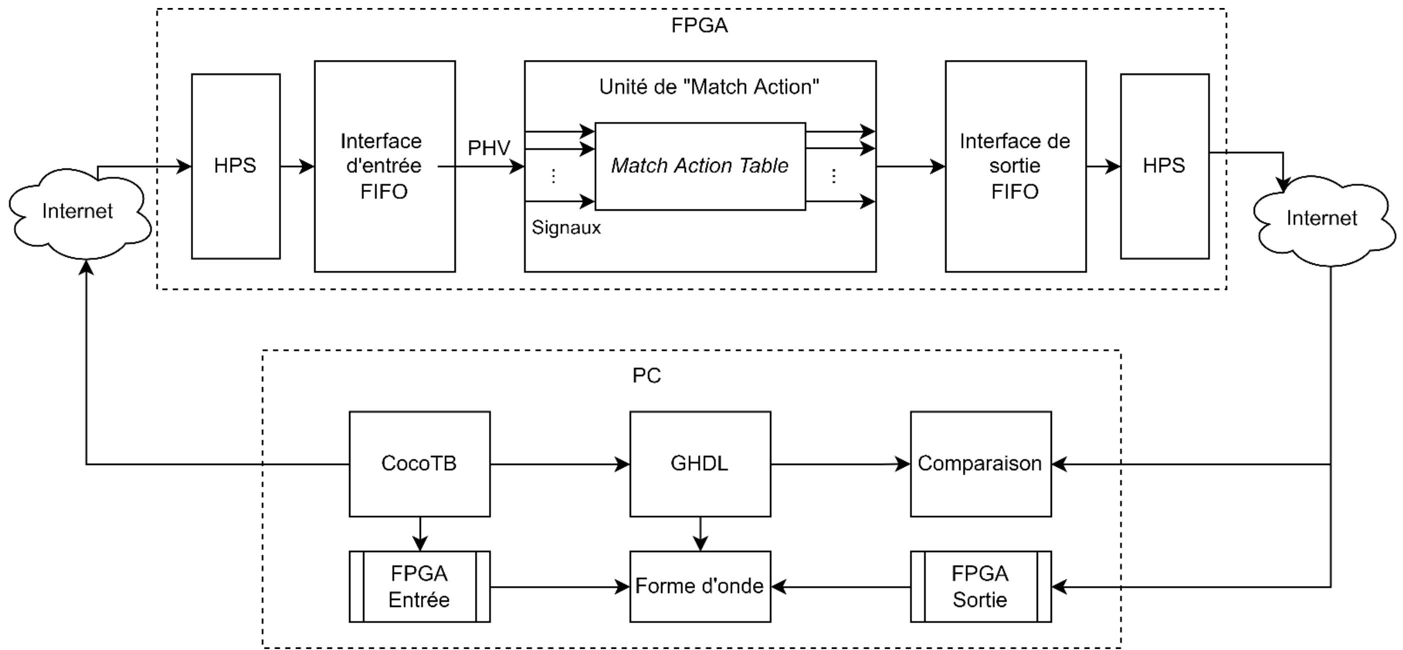


Figure 4.18 : Architecture pour vérifier la MAT

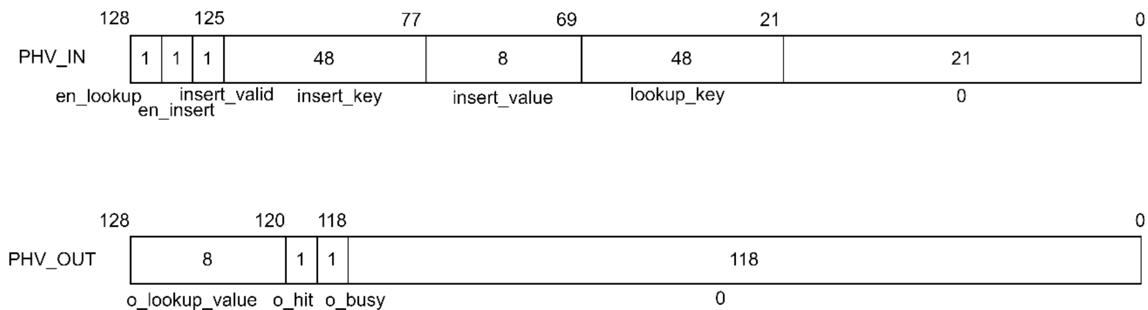


Figure 4.19 : Organisation des signaux sur le PHV

4.4.6.3 Intégration des assertions

L'utilisation d'assertions est une méthode importante dans le domaine de la vérification. Une assertion permet de surveiller un signal ou un événement afin de cibler plus facilement la source d'un bogue. Plusieurs organisations comme OVL (*Open Verification Library*) [14] proposent les méthodes pour mettre en œuvre des assertions. Il est intéressant d'ajouter une fonction d'assertion dans le système. En effet, la MAT fournit un signal « *hit* » affirmant la correspondance d'une clé avec une valeur contenue dans la table. Ce signal est activé lorsqu'une valeur valide est présentée à la sortie. Ce mécanisme peut devenir une condition à surveiller pour tester l'implémentation de l'assertion. Il y a trois tableaux de recherche dans la MAT, donc trois signaux « *hit* » sont produits.

Dans ce mémoire, l'OVL est utilisé comme générateur d'assertion qui permet de surveiller un signal ciblé. Le vérificateur choisi est le « ovl_always » qui sert à confirmer que le signal ciblé est toujours actif. Lorsque le signal entré dans le port « test_expr » n'est plus actif, la sortie « fire » est activée. L'interface de ce composant est présentée à la figure 4.20 [14].

Figure 4.20: L'interface de vérificateur « ovl_always »

Dans le but d'éviter de modifier la MAT, au lieu d'insérer cette fonction directement dans le bloc MAT, le module d'assertion est placé dans l'interface MAU et les signaux à surveiller sont extraits du MAT comme présenté par la figure 4.21. C'est donc dire qu'un ensemble de signaux de sortie en 3 bits est ajouté dans le bloc MAT qui rassemble les signaux « *hit* ». Trois blocs d'assertion « ovl_always » sont insérés dans le MAU. Chaque composant est connecté à un inverseur et puis un signal « *hit* ». Les signaux restent donc actifs quand il n'y a pas de détection. Lorsqu'une clé est trouvée, le signal correspondant est désactivé et une assertion apparait. Les sorties de chaque bloc sont ajoutées une par une dans le bus *Avalon Streaming* permettant de les renvoyer vers le PC. Comme le composant est synchronisé avec l'horloge, chaque fois qu'il y a une valeur valide présentée à la sortie, une assertion devrait apparaître au cycle prochain selon le tableau.

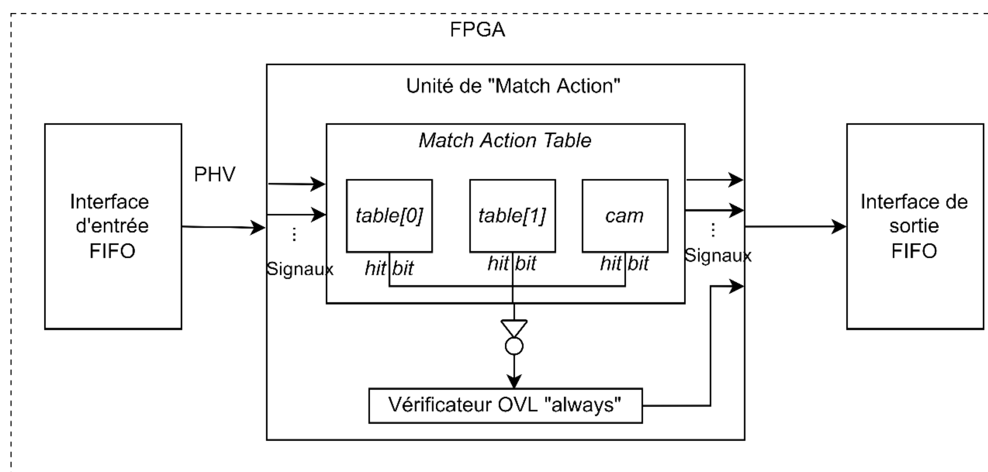


Figure 4.21 : Insertion de l'assertion

4.4.7 Le module de traitement de sortie

Les mécanismes présentés ci-dessus peuvent faire le même test dans le simulateur et dans le FPGA. Pour analyser les résultats, un module de comparaison permet de vérifier la cohérence entre les sorties et un générateur de fichier de forme d'onde permet d'ajouter les signaux de sortie du FPGA dans le résultat de simulation.

4.4.7.1 Le module de comparaison

Ce module sert à récupérer les données enregistrées dans les fichiers de sortie, les comparer et puis les afficher. Durant le test sur FPGA, des retards ou des erreurs peuvent apparaître pour diverses raisons telles que la déstabilisation. Donc une comparaison exacte cycle par cycle est inappropriée. Ce module permet à utilisateur de configurer la tolérance de retard maximale. Une fois qu'il y a une différence entre les deux sorties dans un cycle, il compare la donnée dans le cycle suivant jusqu'à la tolérance. Ce module permet de générer un rapport qui affiche le nombre de cycles total, le nombre de cycles de retard et d'erreur s'il y a eu lieu et les numéros de cycles correspondants. De plus, les données sont affichées dans deux colonnes et les cycles problématiques sont mis en couleur.

4.4.7.2 La régénération du fichier de forme d'ondes

Le simulateur utilisé dans ce projet, le GHDL, génère un fichier de forme d'ondes à la fin de la simulation. Ce fichier contient les informations de tous les signaux internes du DUT. Pour ne pas perdre ces informations et mieux visualiser les sorties, un programme est réalisé pour ajouter les données d'entrée et de sortie obtenues du FPGA dans le fichier.

Le fichier généré par le GHDL est en format de *fst*. Ce fichier de forme d'ondes est difficile à modifier. Donc la commande « *fst2vcd* » est utilisée pour le transformer au format vcd. Les données dans le fichier vcd sont en ordre chronologique. Un exemple est fourni dans la figure 4.22.

```
$scope module module_name $end   Déclaration de module
$var wire 1 ! signal1 $end       Déclaration des signaux sous le
$var wire 1 " signal2 $end       format: type, taille, ID, nom
$upscope $end
$enddefinitions $end
#0     0 ps
0!     Données de signal1 à 0ps
1"     Données de signal2 à 0ps
#3125  3125 ps
1!     Données de signal1 à 3125ps
0"     Données de signal2 à 3125ps
```

Figure 4.22 : Format de données vcd

L'ajout de nouveaux signaux permet la régénération du fichier total. Deux bibliothèques sont importées : le *VCDVCD* [33] qui est un parseur de fichiers vcd et le *vcd.VCDWriter* [34]. Le fichier original est lu par le lecteur *VCDVCD* [33] et les données sont enregistrées dans un tableau de structure « Signal ». Cette structure contient toutes les informations d'un signal, contenant l'ID, le temps de la fin, le nom, la taille, le type, un tableau pour les données et un tableau pour les temps correspondants. Ensuite, les fichiers d'entrée et de sortie du test en FPGA sont lus. Les signaux sont extraits et aussi, les informations contenues dans cette structure sont complétées. Les informations sont enfin écrites dans un nouveau fichier dans l'ordre utilisant le rédacteur de VCD. Le flux de travaux est présenté dans la figure 4.23.

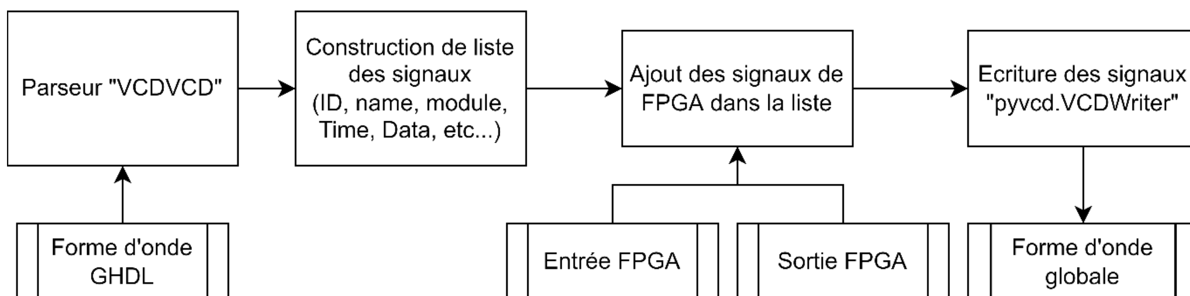


Figure 4.23 : Régénération de fichier VCD

4.5 L'architecture sur la plateforme de Xilinx

Cette section décrit l'environnement de test mis en œuvre sur la plateforme de Xilinx qui est réalisé sur la carte *NetFPGA-SUME*. L'architecture principale suit l'ensemble des idées introduites précédemment. Les modifications nécessaires pour ajuster le système de test à la plateforme sont aussi détaillées.

4.5.1 Les interfaces utilisées

La famille d'interfaces AXI est utilisée dans la plateforme de Xilinx.

4.5.1.1 « AXI4 Stream »

Le protocole *AXI4-Stream* est utilisé comme interface standard pour connecter les composants et transférer les données d'un maître à un esclave dans le FPGA Xilinx. Cette interface est similaire à *Avalon Streaming*, les signaux et leurs fonctions sont présentés dans le tableau 4.3. Le bus de métadonnées inclut les informations de paquet tel que les ports source et destination.

Tableau 4.3 : Les signaux de l'interface *AXI4-Stream*

Nom	Direction	Largueur	Fonction
ACLK	entrée	1	Horloge
ARESETN	entrée	1	*Reset*
TVALID	Source->Sink	1	*Handshake*
TREADY	Sink->Source	1	*Handshake*
TDATA	Source->Sink	256	Bus de Données
TLAST	Source->Sink	1	Fin de paquet
TKEEP	Source->Sink	32	Longueur de paquet
TUSER	Source->Sink	128	Bus de Metadonnées

4.5.1.2 « AXI4-Lite »

L'interface *AXI4-Lite* est une interface de type *Memory Mapped* plus simple. Elle ne nécessite pas toutes les fonctionnalités d'AXI4. L'*AXI4-Lite* est plus complexe de l'interface *Avalon MM* et ses signaux sont présentés dans la figure 4.24 [12].

Global	Write address channel	Write data channel	Write response channel	Read address channel	Read data channel
ACLK	AWVALID	WVALID	BVALID	ARVALID	RVALID
ARESETn	AWREADY	WREADY	BREADY	ARREADY	RREADY
–	AWADDR	WDATA	BRESP	ARADDR	RDATA
–	AWPROT	WSTRB	–	ARPROT	RRESP

Figure 4.24: Les signaux de l'interface *AXI4-Lite*

Le système de « *handshake* » entre les signaux *ready* et *valid* est nécessaire dans chaque canal pour vérifier les données et les adresses séparément.

4.5.2 Présentation d'un exemple : le projet « Reference NIC »

Dans ce mémoire, l'exemple *reference_nic* fourni par l'organisation *NetFPGA* est utilisé comme projet de base. Les blocs sont insérés dans ce projet pour répondre à nos besoins. Ce projet fait de cette carte une carte réseau (NIC) transférant les données venues de l'extérieur au PC et vice versa. Le flux de paquets traverse quatre blocs IP. Son module *nf_10g_interface* combine le sous-système Ethernet Xilinx AXI 10G et un adaptateur *AXI4-Stream*. Chaque instance de ce module correspond à un port Internet. Les paquets arrivant des ports SFP+ externes sont traités par les parties PMA et PCS et puis MAC dans l'IP Ethernet 10G. Chaque paquet entrant est annoté avec des métadonnées et est finalement transformé en *AXI4-Stream* 256 bits. Le côté TX suit exactement le même chemin, mais dans la direction opposée. Le module *input_arbiter* est connecté avec le RX des interfaces 10G. Il a cinq interfaces d'entrée : les quatre modules *nf_10g_interface* et un module DMA pour communiquer avec le port PCIe du PC. Chaque entrée se connecte à une file d'attente d'entrée. Le module sert à rassembler les files à une seule file de sorties. Le module *output_port_lookup* récupère les données sorties de *input_arbiter* et prend en charge de décider la direction d'un paquet. Une fois cette décision prise, les paquets sont transmis au module *output_queues*. Selon la destination, ce module distribue les paquets au port approprié, soit les interfaces TX des module 10Gbps soit le DMA pour transférer au PC. L'interface utilisée par ces blocs est l'*AXI4-Stream* qui sera présentée ci-dessous.

À part ce flux de paquets, un processeur logiciel *Microblaze* est aussi ajouté pour faire des tests de puissance. La communication entre *Microblaze* et les autres composants sont réalisés par l'interface *AXI4-Lite*.

4.5.3 Détails de l'architecture

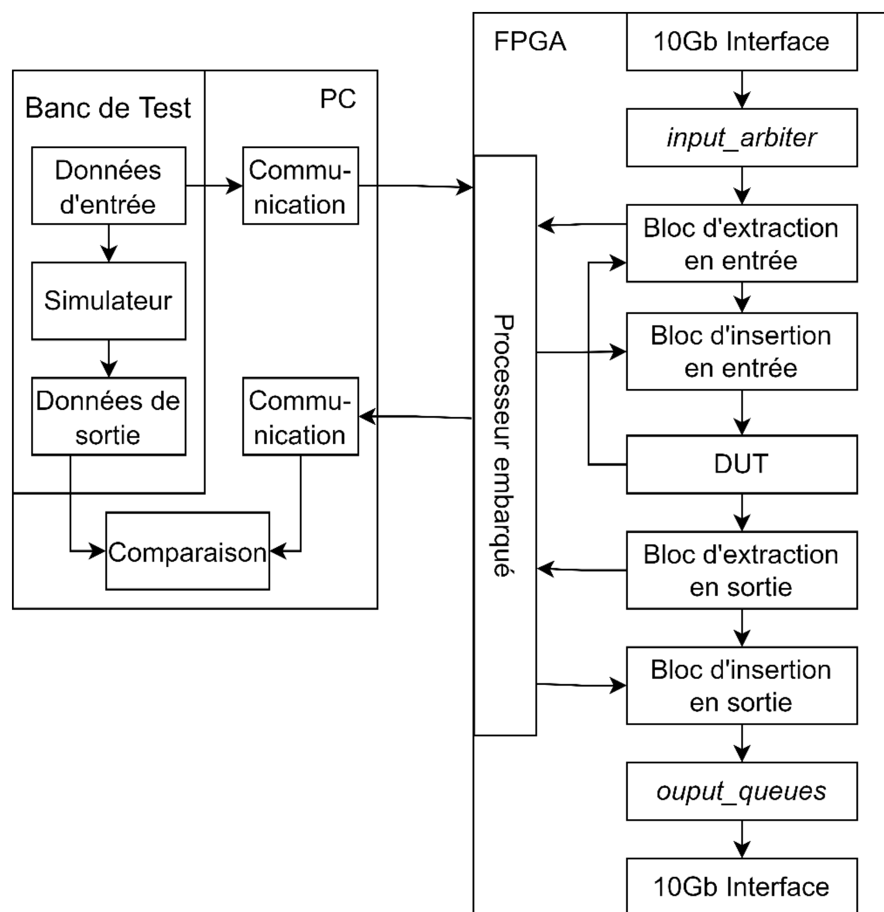


Figure 4.25 : Architecture générale sur la plateforme Xilinx

La Figure 4.25 présente l'architecture générale de l'environnement de test réalisé. Contrairement à l'architecture précédente, la carte *NetFPGA-SUME* permet de recevoir les paquets et de traiter directement les données pures sans les passer à un processeur ou à un noyau. Le système permet donc : 1) d'insérer les données dans les signaux d'entrée du DUT et de capturer les données de sortie et 2) de récupérer les données pendant l'exécution en temps réel.

La partie droite de la figure 4.25 représente l'implémentation matérielle du système réalisant les tests physiques. Cette partie définit comment les paquets traversent la carte et d'où les données sont récupérées ou sont insérées. Le flux de paquets entre dans la carte traversant les ports Internet SFP+. Les blocs d'interface 10Gb servent à analyser les paquets et à transformer les données à l'*AXI4-Stream* de 256 bits avec un bus de métadonnées et ses 128 bits associés. Ce dernier contient le port source et le port destination. Les quatre bus de données sont ensuite combinés en un seul bus par

le bloc d'arbitre d'entrée. Le bloc de données d'entrée permet de récupérer ou d'insérer les données à partir du processeur logiciel, *Microblaze*. Par la suite, les données traversent le centre du système donc le DUT. Après avoir été traitées par le DUT, les données de sortie sont récupérées. Finalement, les données entrent dans la file d'attente de sortie et sont distribuées aux interfaces 10Gb selon les ports indiqués dans les métadonnées. Les blocs d'extraction n'ajoutent pas de registre dans le flux de paquets, donc ils n'introduisent normalement pas de délai.

La partie gauche représente la simulation logicielle. Comme présenté dans les sections précédentes, le banc de test en Python écrit à la base de la bibliothèque *CocoTB*. Il constitue le cœur de la partie logicielle qui prend en charge 1) de récupérer les informations définies par l'utilisateur, 2) d'activer le simulateur GHDL, 3) de collecter les données de sortie du simulateur et 4) d'échanger les données avec le processeur logiciel sur la carte FPGA en traversant le module UART.

4.5.4 La conception du matériel

Pour simplifier le travail, les blocs d'interface 10Gb, le bloc d'arbitre d'entrée, le DUT et le bloc de file d'attente de sorties sont réutilisés de l'exemple « Reference NIC ». Les blocs d'insertion et d'extraction de données prennent en charge la communication avec la partie logicielle. Ils sont donc notre contribution principale dans cette plateforme. La figure 4.26 représente l'architecture matérielle complète. Trois blocs basés sur les FIFOs sont créés pour atteindre l'objectif. Leur interface est illustrée dans la figure 4.27.

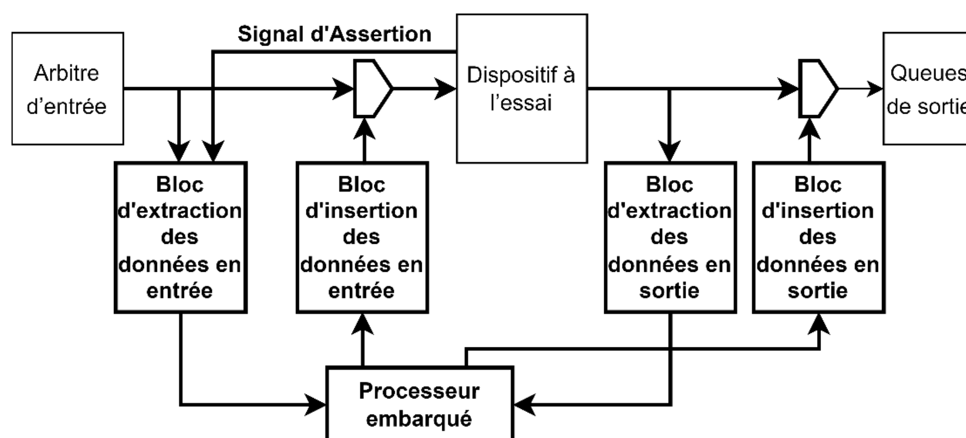


Figure 4.26 : Architecture matérielle pour la plateforme Xilinx

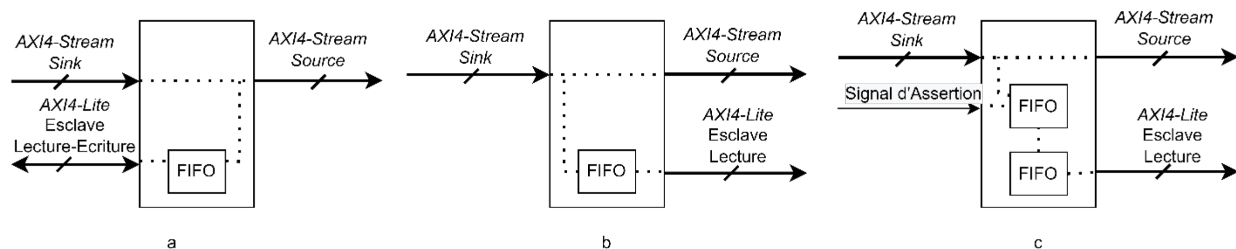


Figure 4.27 : Interfaces des blocs de communication de données a) bloc d'insertion b) bloc d'extraction c) bloc d'extraction avec assertion

4.5.4.1 Le bloc d'insertion des données

Le bloc d'insertion de données permet d'insérer les données enregistrées dans la FIFO au bus *AXI4-Stream*. Comme illustré dans la figure 4.27.a, deux modes sont offerts pour soutenir différents fonctionnements. En mode normal, les données du simulateur sont transmises au processeur logiciel. Puis, elles sont écrites dans ce bloc et stockées dans la FIFO. Pendant ce temps, le bloc est transparent dans le système. Les données venues du bloc précédent traversent ce bloc sans aucune modification. Lors de la réception d'une commande "changement de mode", qui est transmise par le processeur logiciel, à chaque cycle d'horloge, les données stockées dans la FIFO sont envoyées vers le DUT. La commande "changement de mode" peut donc être considérée comme un signal « GO », qui est utilisé pour contrôler la synchronisation des données sortant de la FIFO. Le processeur logiciel lit également le niveau de remplissage de la FIFO pour éviter l'écrasement des données.

Plus en détail, le bus *AXI4-Stream* dans ce système comporte 256 bits de données, 128 bits de métadonnées, 32 bits de « *keep* » qui représentent la longueur de paquet, 1 bit de fin de paquet donc en totalité 417 bits. La largeur de FIFO est aussi de 417 bits. Le tableau 4.4 détaille l'adresse de mémoire et son contenu pour le bus *AXI4-Lite* qui sert à réaliser la communication entre le processeur logiciel et le bloc. Chaque fois qu'une donnée est écrite dans l'adresse 56, le bloc pousse les 417 bits dans la FIFO, une case de la FIFO est remplie avec succès. Si la FIFO est déjà remplie, la nouvelle donnée écrite sera perdue. Ce niveau de remplissage peut être lu sur l'adresse 0. Le bit 0 à l'adresse 60 sert au changement de mode.

Tableau 4.4 : Adresse en mémoire du bloc d'insertion

Adresse	Type	Données
0	Lecture	Niveau de remplissage de FIFO
4,8,12,…,32	Ecriture	FIFO bit 161 à bit 416 (Données)
36,…,48	Ecriture	FIFO bit 33 à bit 160 (Metadonnées)
52	Ecriture	FIFO bit 1 à bit 32 (Keep)
56	Ecriture	FIFO bit 0 (Fin de paquet)
60	Ecriture	Configuration de mode

4.5.4.2 Le bloc d'extraction des données

Le bloc d'extraction de données prend en charge de collecter les données valides transitant par le système. La figure 4.27.b et c montrent que les données venues du bus *AXI4-Stream Sink* sont, d'un côté, passées vers le prochain bloc et, d'un autre côté, stockées sans modification dans des FIFOs. Le processeur logiciel peut accéder à ces données via un bus de données *AXI4-Lite*. Les données sont accessibles dans l'ordre et le niveau de remplissage des FIFOs peut être lu à l'adresse prédéfinie. Pour satisfaire divers scénarios de test, deux types de fonctionnalités sont pris en charge. Dans le premier cas, le bloc d'extraction de données reçoit un signal d'assertion en entrée. En l'absence de déclencheur, la FIFO est maintenue à moitié pleine. Pour maintenir ce niveau de remplissage, les données les plus anciennes sont supprimées avec l'arrivée de nouveaux paquets. Une fois qu'une assertion est détectée, c'est-à-dire que le signal d'assertion d'entrée est activé, la FIFO est remplie par la suite de données. Ainsi, les données avant et après l'assertion sont disponibles à partir de la même FIFO. Cela permet de détecter les données problématiques et de les comparer aux données attendues pour la découverte de bogues. Le deuxième cas est un simple stockage de données. Le bloc d'extraction de données n'a pas de signal de déclenchement. Les paquets sont stockés jusqu'à ce que les FIFOs soient complètement remplies. La construction de l'adresse mémoire suit le même principe que celle du bloc d'insertion, représentant dans le tableau 4.5. La lecture de la FIFO se fait lors d'une lecture de AXI4-Lite valide sur l'adresse 56.

Tableau 4.5 : Adresse de mémoire de bloc d'extraction

Adresse	Type	Données
0	Lecture	Niveau de remplissage de FIFO
4,8,12,…,32	Lecture	FIFO bit 161 à bit 416 (Données)
36,…,48	Lecture	FIFO bit 33 à bit 160 (Metadonnées)
52	Lecture	FIFO bit 1 à bit 32 (Keep)
56	Lecture	FIFO bit 0 (Fin de paquet)

4.5.4.3 Détail de l'implémentation

Les FIFOs utilisées dans les trois types de blocs sont générés par l'IP « générateur de FIFO » fourni par Xilinx *Vivado*. Il permet de générer divers types de FIFO construite à base de ressource différente, de l'horloge de lecture et d'écriture, la largeur et la profondeur. La carte supporte quatre ports Internet à 10Gbps, offrant donc en totalité 40Gbps, le bus *AXI4-Stream* fonctionne à une fréquence de 160MHz. Avec le bus de données de 256 bits, le flux de données peut atteindre 40.96Gbps qui est adapté au débit maximal d'ensemble des quatre ports. Cependant, le processeur logiciel fonctionne à une fréquence inférieure, 100MHz. Par conséquent, les ports connectés avec le bus *AXI4-Stream* demandent une fréquence de 160MHz et ceux connectés avec le bus *AXI4-Lite* requièrent une fréquence de 100MHz. Dans le bloc d'insertion et d'extraction sans assertion, une seul FIFO avec des horloges d'entrée et de sortie différentes est suffisant. Pour le bloc d'extraction avec assertion, l'implémentation est plus complexe à cause du système de déclenchement. Les données sont d'abord stockées dans une FIFO synchrone avec une seule horloge permettant d'écrire et de lire sous 160MHz. Cette FIFO est gardée à moitié remplie avant l'activation de l'assertion et est remplie après. Une fois qu'elle est remplie, les données dans cette FIFO sont envoyées dans une deuxième FIFO avec une horloge d'écriture et de lecture différente prenant en charge la communication avec le processeur logiciel.

Enfin, comme mentionné ci-dessus, les blocs d'extraction ne modifient pas les données qui les traversent. Le bloc d'insertion offre un mode transparent. Par conséquent, le système de débogage n'influence pas le fonctionnement normal de l'application lorsqu'il n'est pas activé.

4.5.4.4 Communication entre le processeur logiciel et les blocs

Le programme sur le processeur logiciel est écrit en C. Le tableau 4.6 liste les fonctions de système utilisées pour la lecture ou l'écriture de mémoire.

Tableau 4.6 : Liste des fonctions utilisées pour la communication en *AXI4-Lite*

Fonction	Fonctionnement
Xil_Out32(IP_BASEADDR+ADDR, data)	Ecrire 32 bits à l'adresse ciblée
Xil_In32(IP_BASEADDR+ADDR)	Lire 32 bits à l'adresse ciblée

Notons que l'adresse en mémoire est basée sur une résolution en octet et la largeur du bus de données d'*AXI4-Lite* est de 32 bits, chaque registre prend quatre adresses et chaque lecture ou écriture est en mot de 32 bits. Pour un bus de 417 bits, 14 échanges sont requis.

Plus précisément, la lecture de niveau de remplissage est toujours sur l'adresse 0. Car le compteur de ce niveau n'est pas toujours précis, les drapeaux « vide » et « plein » sont aussi nécessaires. Le format des informations lues à l'adresse 0 est présenté dans la figure 4.28. Dans ce mémoire, la profondeur des FIFOs implémentés avec deux horloges est de 1024, donc le niveau de remplissage peut s'exprimer par une adresse de 10 bits. Deux FIFO sont utilisés dans le bloc d'extraction avec l'assertion, donc leurs niveaux de remplissage sont lus en même temps. À cause du IP de générateur, une FIFO avec une seule horloge dont la profondeur est de 1024 qui prend 11 bits pour représenter le niveau.

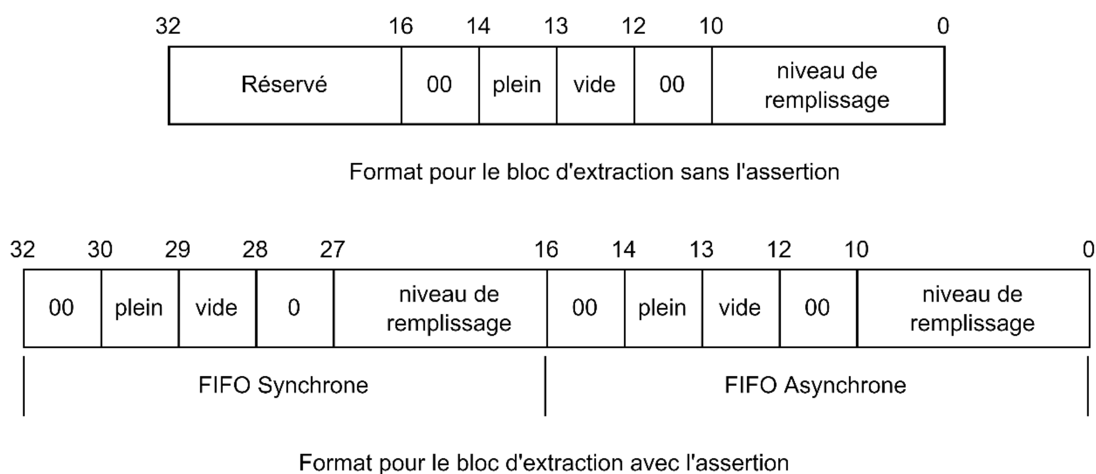


Figure 4.28 : Format des champs qui expriment le niveau de remplissage

4.5.5 Communication à l'aide d'un UART

Cette section détaille la communication à l'aide d'un UART entre le programme de banc de test sur PC et le processeur logiciel sur FPGA.

4.5.5.1 Communications au niveau du processeur embarqué

Le tableau 4.7 liste les fonctions de système utilisées pour la communication en UART.

Tableau 4.7 : Liste des fonctions utilisées pour la communication en UART

Fonction	Fonctionnement
XUartLite_RecvByte(UARTBASEADDR)	Recevoir un octet par l'interface UART
Xil_printf(data)	Envoyer les données à l'interface UART

La communication à l'aide d'un UART se réalise de façon « Commande – Réponse ». À cause de la limitation de la taille de la RAM, le programme doit être le plus simple possible. Les commandes

ne contiennent donc qu'une lettre. Le tableau 4.8 liste les commandes implémentées. Le système complet contient quatre blocs comme présenté à la figure 4.26: un bloc d'insertion à l'entrée, un bloc d'insertion à la sortie, un bloc d'extraction incluant le module assertion et un bloc sans ce module.

Tableau 4.8 : Commande et fonctionnement

Bloc	Commande	Fonctionnement
insertion à l'entrée	a	Lecture du niveau de remplissage
	w	Ecriture des données dans le bloc
	g	Signal « Go » qui déclenche l'envoi de données
	s	Signal « Stop » qui retourne au mode normal
extraction avec l'assertion	c	Lecture du niveau de remplissage
	t	Lecture des données
extraction sans l'assertion	b	Lecture du niveau de remplissage
	r	Lecture des données
	i	Initialiser (vider) la FIFO
insertion à la sortie	d	Lecture du niveau de remplissage
	o	Ecrire les données dans le bloc
	e	Signal « Go » pour déclencher l'envoi de données
	f	Signal « Stop », qui retourne l'interface au mode normal

Une fois qu'une commande d'écriture des données est reçue, le programme attend ensuite 56 octets donc 448 bits. Cette chaîne de caractères est écrite dans le bloc d'insertion 32 bits par 32 bits. Un changement de boutisme est fait avant l'écriture, car les données de réseaux sont sous le format de petit-boutiste. Lorsqu'il y a une demande de lecture des données, une boucle lit de l'adresse 4 à 56 dans un bloc. Une chaîne de 112 caractères est envoyée vers le PC après le changement de boutisme.

4.5.5.2 Communication au niveau du PC

La communication UART du côté du PC est écrite en Python. Comme présenté dans la figure 4.29, un module appelé « communication » est réalisé pour intégrer les fonctions utiles. La bibliothèque « Pyserial » [21] est importée pour établir le lien avec le UART. Les fonctions *write*, *read* et *readline* permettent d'écrire les octets, de lire un nombre d'octets paramétré et de lire une ligne de données. En effet, les données qui sont écrites dans le UART doivent être retrouvées à la réception, par exemple, si la lettre 'w' est écrite, la lecture prochaine sera 'w\r\n', donc une lecture doit être faite après chaque écriture pour jeter cette ligne inutile.

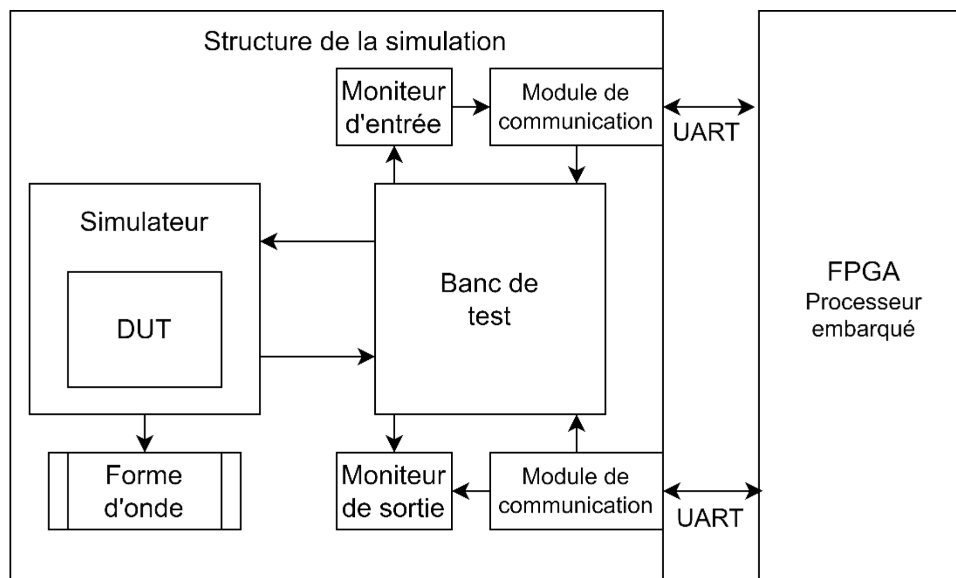


Figure 4.29 : Architecture de la partie logicielle

L'architecture dans la figure 4.29 sert à faire à la fois le test simulé en logiciel par la coopération *CocoTB* et GHDL et le test en FPGA. Le flux de travail consiste à insérer les données d'entrée au DUT et puis à récupérer les données à sa sortie. Donc, ce module utilise seulement le bloc d'insertion à l'entrée et le bloc d'extraction sans assertion qui sont présentes avant et après le DUT. Le tableau 4.9 liste les fonctions créées dans ce module.

Tableau 4.9 : Fonctions crées pour la communication UART

Fonction	Fonctionnement
write_to_insertion	Lire les données par le fichier d'entrée et les envoyer dans le bloc d'insertion d'entrée
read_from_extraction	Lire les données par le bloc d'extraction sans l'assertion et les enregistrer dans le fichier de sortie
get_extraction_fill_level	Obtenir le niveau de remplissage du bloc d'extraction
get_insertion_fill_level	Obtenir le niveau de remplissage du bloc d'insertion
init_extraction	Initialiser le bloc d'extraction
send_go	Envoyer le signal « GO » au bloc d'insertion
send_stop	Mettre le bloc d'insertion au mode normal

Pour simplifier la description, la figure 4.30 illustre les diagrammes logiques de la fonction write_to_insertion (a) et read_from_extraction (b).

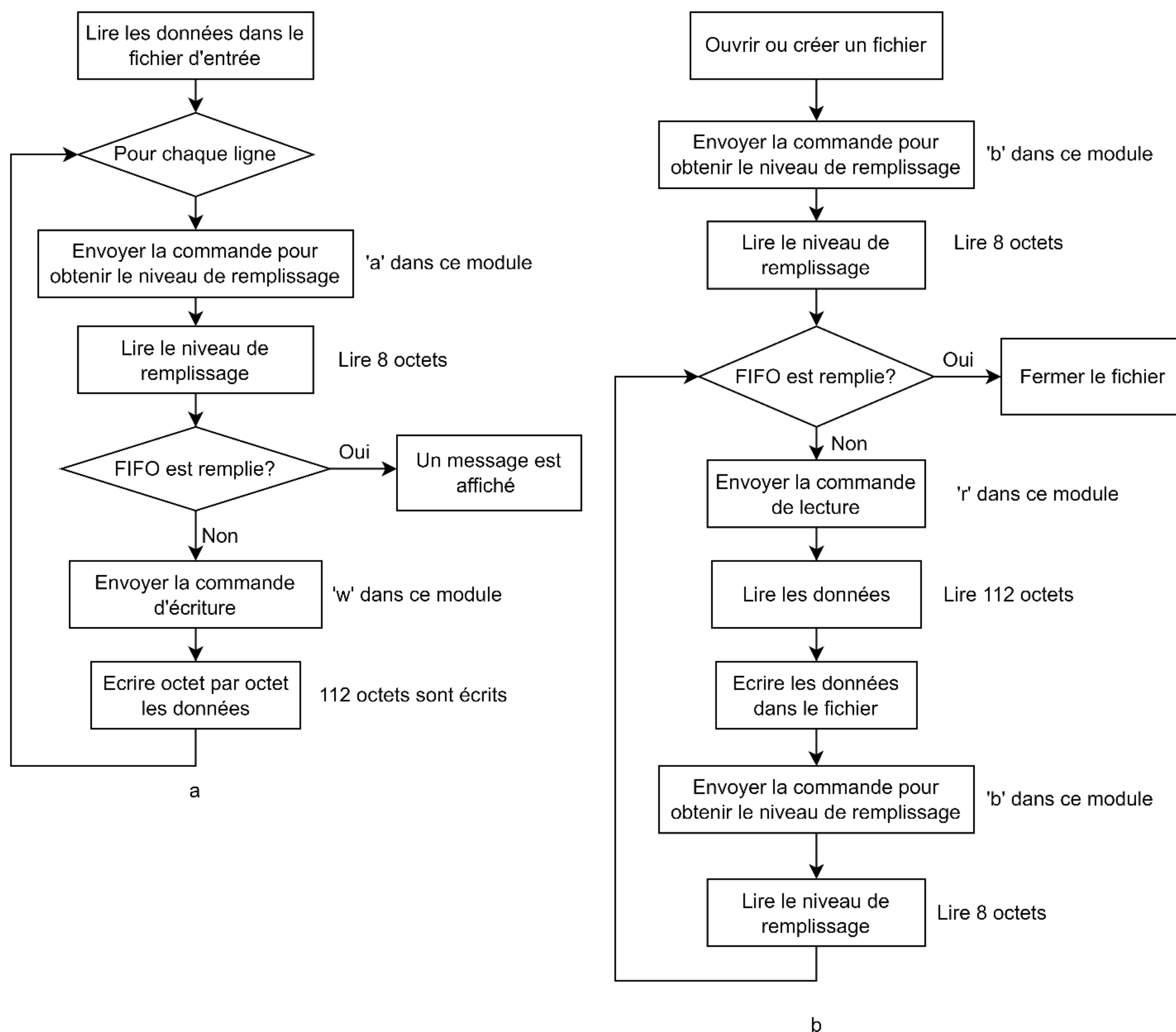


Figure 4.30 : Logigramme de la fonction a) write_to_insertion b) read_from_extraction

Les autres fonctions servent simplement à l'envoi de commandes (précisées dans le tableau 4.8).

À part ce module, deux autres programmes sont créés pour faire fonctionner les deux autres blocs : le bloc d'extraction avec assertion et le bloc d'insertion à la sortie. L'un permet d'initialiser le bloc d'extraction avec assertion, lire les données et les enregistrer dans un fichier. L'autre lit des données par un fichier et les écrits dans le bloc d'insertion à la sortie. La logique principale est similaire, sauf que les commandes sont changées.

4.5.6 Modification au programme de test dans CocoTB

Le programme de banc de test suit le même principe de celui présenté dans la section 4.3, mais certaines modifications sont faites pour adapter le nouvel environnement. Tout d'abord, les signaux sont changés de ceux du bus *Avalon Streaming* à ceux du bus *AXI4-Stream*. De plus, une fonction de changement de l'ordre des données est ajoutée pour traiter le problème de boutisme. La figure 4.31 présente l'ordre de données dans le bus *AXI4-Stream*. La ligne FIFO spécifie à quel bit les données se positionnent et la ligne *AXI4-Lite* identifie l'adresse mémoire où on doit lire ou écrire ces données. L'exemple proposé correspond à la transmission d'un paquet UDP contenant un message « Hello World ! ». Le début de paquet fournit donc l'adresse MAC de destination (20 :35 :55 :4D :45 :00) injectée à partir de poids faible dans le bus TDATA. Mais pour que les données soient lisibles et compréhensibles, elles devraient être en format de gros-boutisme comme présenté dans la figure 4.32. Une fonction de changement de boutisme d'un mot 32 bits et une fonction pour réorganiser les données sont implémentées. Elles sont appelées avant l'assignation des signaux d'entrée et l'enregistrement des signaux de sortie durant la simulation logicielle.

	TDATA								TUSER				TKEEP	TLAST
FIFO	416							161	160			33	32 1	0
AXI4-Lite	32	28	24	20	16	12	8	4	48	44	40	36	52	56
	A8C05003	A8C00763	11400040	D64F2900	00450008	00380000	00020045	4D555302	00000000	00000000	00000000	0001003C	FFFFFFFF	00000000
	C9545362	00000000	0021646C	726F5720	2C6F6C6C	65489B8A	15000A1A	0B924603	00000000	00000000	00000000	0001003C	0FFFFFFF	00000001

Figure 4.31 : Ordre des données dans le bus AXI4-Stream

```
0253554D45000200000038000800450000294FD6400040116307C0A80350C0A83C00010000000000000000000000000FFFFFFFF00000000
0346920B1A0A00158A9B48656C6C6F2C20576F726C6421000000000000625354C93C00010000000000000000000000000FFFFFF0F01000000
```

Figure 4.32 : Format des données attendues dans les fichiers d'entrée ou de sortie

CHAPITRE 5 EXPERIMENTATIONS ET RESULTATS

Ce chapitre présente les méthodes de vérification et les résultats obtenus pour l'environnement de test sur les deux plateformes.

5.1 Résultats sur la plateforme d'Intel

5.1.1 Système général

Le système à tester est implémenté avec deux canaux de bus d'*Avalon Streaming* avec la taille de données de 128 bits. Le DUT utilisé est un bloc simple d'addition et de soustraction. Comme présenté dans la figure 5.1, la sortie de canal 1 est la somme des canaux d'entrée et celle de canal 2 est la soustraction.

Figure 5.1: DUT pour le test du système général

Dans ce test, pour aligner le passage des données, celui-ci s'exécute approximativement en 1 minute (0xB2D05F00 cycles) après la programmation de la carte. Les données d'entrée sont listées dans le tableau 5.1. Un autre module est ajouté permettant de calculer les résultats.

Tableau 5.1 : Liste des entrées

Canal 1	Canal 2
0A0A0A0A0A0A0A0A0A0A0A0A0A0A0A0A	A0A0A0A0A0A0A0A0A0A0A0A0A0A0A0A0
0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	11111111111111111111111111111111
0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	11111111111111111111111111111111
0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	11111111111111111111111111111111
0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	11111111111111111111111111111111

Figure 5.2 : Résultats obtenus avec *CocoTB*

La figure 5.2 présente les résultats simulés et espérés. Après avoir finalisé la simulation, le transfert vers le FPGA est activé. La figure 5.3 présente les données envoyées et les résultats reçus. Les deux résultats sont identiques. Cela montre que le DUT fonctionne correctement et sans problème de synchronisation.

Figure 5.3 : Résultat en FPGA

5.1.2 Vérification de la « MAT »

Cette partie présente le résultat généré par l'architecture de vérification de la MAT. La procédure de test réalisée par le programme dans *CocoTB* est décrite dans la figure 5.4.

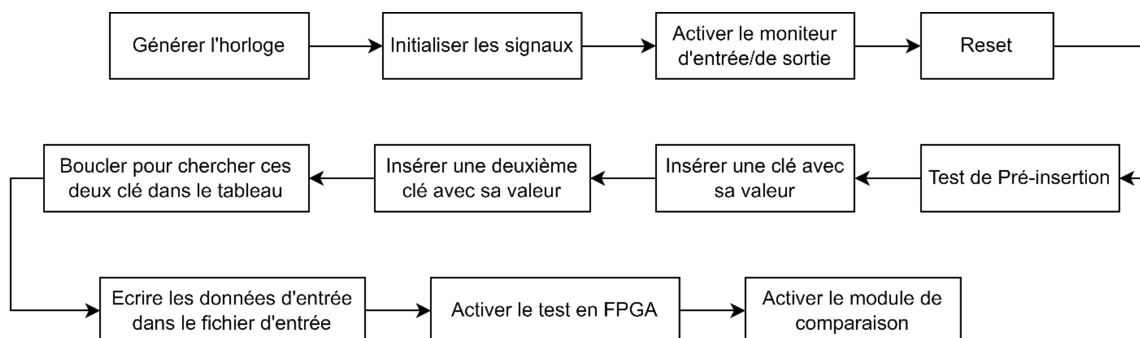


Figure 5.4 : Processus de test en *CocoTB* pour la vérification du module MAT

Les clés et leur valeur insérées dans ce test sont listées dans le tableau 5.2.

Tableau 5.2 : Clés et leur valeur

Mode d'insertion	Clé	Valeur
Pré-insertion	0x000000000004	0x04
Pré-insertion	0x000000000005	0x05
Insertion	0xBB00BB00BB02	0xAB
Insertion	0xAA00AA00AA01	0xCD

En plus du mode d'insertion, le MAT supporte une pré-insertion de clés et de valeurs. Elles sont enregistrées dans un fichier spécifique. Lors de la synthèse du projet, ce fichier est lu et les données sont stockées directement dans la mémoire de la MAT. Dans ce test, la pré-insertion est utilisée pour produire manuellement des différences dans le résultat. Les données de pré-insertion sont ajoutées seulement dans le projet de simulation logicielle, mais pas dans le projet implémenté dans le FPGA. C'est-à-dire, durant le test, quatre clés sont dans la MAT du côté logiciel et seulement deux dans la MAT à l'intérieur du FPGA. Cela cause les différences à la sortie pour que le module de comparaison puisse les détecter.

```
VPI module loaded!
     -.--ns INFO     cocotb.gpi                        ..mbed/gpi_embed.cpp:240  in _embed_sim_init          Python interpreter initialized and cocotb loaded!
     0.00ns INFO     cocotb                                  __init__.py:220  in _initialise_testbench_   Running on GHDL version 2.0.0-dev (1.0.0.r419.g9832
36ac) [Dunoon edition]
     0.00ns INFO     cocotb                                  __init__.py:226  in _initialise_testbench_   Running tests with cocotb v1.5.2 from /usr/local/li
b/python3.8/dist-packages/cocotb
     0.00ns INFO     cocotb                                  __init__.py:247  in _initialise_testbench_   Seeding Python random module with 1641441467
     0.00ns WARNING  cocotb                                  __init__.py:266  in _initialise_testbench_   Pytest not found, assertion rewriting will not occu
r
     0.00ns INFO     cocotb.regression                    regression.py:127  in __init__                 Found test test_hashtable.test_hashtable
     0.00ns INFO     cocotb.regression                    regression.py:468  in _start_test              Running test 1/1: test_hashtable
     0.00ns INFO     ..est.test_hashtable.0x7fb6414f4760  decorators.py:312  in _advance                 Starting test: "test_hashtable"
                                                                                                          Description: None
     0.00ns INFO     cocotb.hashtable                  test_hashtable.py:182  in init                    Start clock and initialize input signals to zero
    25.00ns INFO     cocotb.hashtable                  test_hashtable.py:354  in test_hashtable          Preinsert Test
    30.00ns INFO     cocotb.hashtable                  test_hashtable.py:358  in test_hashtable          Output value: 0x0
    35.00ns INFO     cocotb.hashtable                  test_hashtable.py:360  in test_hashtable          Output value: 0x0
    40.00ns INFO     cocotb.hashtable                  test_hashtable.py:362  in test_hashtable          Output value: 0x4
    45.00ns INFO     cocotb.hashtable                  test_hashtable.py:365  in test_hashtable          Output value: 0x0
    50.00ns INFO     cocotb.hashtable                  test_hashtable.py:367  in test_hashtable          Output value: 0x0
    55.00ns INFO     cocotb.hashtable                  test_hashtable.py:369  in test_hashtable          Output value: 0x5
    55.00ns INFO     cocotb.hashtable                  test_hashtable.py:371  in test_hashtable          Insert Test
   100.00ns INFO     cocotb.hashtable                  test_hashtable.py:392  in test_hashtable          Lookup Test
Received b'Connection Build! FIFO Input 0'
Start Read file: Input 0
End thread: Input 0
```

Test de CocoTB

Communication avec FPGA

Figure 5.5 : Résultats de la simulation dans *CocoTB* (MAT)

La figure 5.5 présente le résultat affiché après avoir exécuté le programme dans *CocoTB*. Dans le cadre jaune, les messages sont affichés durant la simulation logicielle. Ces messages sont programmés en Python donc l'utilisateur peut les définir individuellement. Dans ce cas, il affiche les données de sortie dans le test de pré-insertion et puis lors du changement de mode de la MAT. Après avoir finalisé la simulation, la communication avec le FPGA est activée dans le cadre bleu. La figure 5.6 présente les messages suivants : les sept premières lignes signifient que les *sockets* pour chaque interface sont établis et les données d'entrées sont bien envoyées. Les tailles des paquets reçus lorsque la réception de données de sortie est affichée. Dans ce cas, la taille totale des données est de 8840 octets. Dans le premier paquet, il y a seulement 2896 octets donc la réception de paquets se poursuit. Une fois tous les octets reçus, le module de comparaison est activé.

```
Start thread: Input 0
Received b'Connection Build! FIFO Input 0'
Start Read file: Input 0
End thread: Input 0
Start thread: Output 0
Received b'Connection Build! FIFO Output 0'
data_total_len =  8840
length of data_update: 2896
length of data_update: 4500
data_origin_len =  8840
Start Write file: Output 0
End thread: Output 0
```

Communication avec FPGA

```
    No.                 Simulation Output                       FPGA Ouput
  Total                               441                              441
      0  00000000000000000000000000000000  00000000000000000000000000000000
      1  00000000000000000000000000000000  00000000000000000000000000000000
      2  00000000000000000000000000000000  00000000000000000000000000000000
      3  00000000000000000000000000000000  00000000000000000000000000000000
      4  00000000000000000000000000000000  00000000000000000000000000000000
      5  00000000000000000000000000000000  00000000000000000000000000000000
      6  00000000000000000000000000000000  00000000000000000000000000000000
      7  04800000000000000000000000000000  00000000000000000000000000000000
      8  00000020000000000000000000000000  00000000000000000000000000000000
      9  00000000000000000000000000000000  00000000000000000000000000000000
     10  05800000000000000000000000000000  00000000000000000000000000000000
     11  05800020000000000000000000000000  00000000000000000000000000000000
     12  05c00020000000000000000000000000  00400000000000000000000000000000
     13  05c00020000000000000000000000000  00400000000000000000000000000000
     14  05c00020000000000000000000000000  00400000000000000000000000000000
     15  05800020000000000000000000000000  00000000000000000000000000000000
     16  05c00020000000000000000000000000  00400000000000000000000000000000
     17  05c00020000000000000000000000000  00400000000000000000000000000000
     18  05c00020000000000000000000000000  00400000000000000000000000000000
     19  05800020000000000000000000000000  00000000000000000000000000000000
     20  00400020000000000000000000000000  00400000000000000000000000000000
     21  00400000000000000000000000000000  00400000000000000000000000000000
     22  abc00400000000000000000000000000  abc00400000000000000000000000000
     23  00004020000000000000000000000000  00004020000000000000000000000000
     24  00000400000000000000000000000000  00000400000000000000000000000000
     25  cd800400000000000000000000000000  cd800400000000000000000000000000
```

Figure 5.6 : Résultat de la partie de FPGA (MAT)

Le cadre rose dans les figures 5.6 et 5.7 met en évidence le résultat du module de comparaison. La première colonne numérote les lignes de données. La deuxième et la troisième colonne listent les sorties de simulation et celles en provenance du FPGA. Les lignes en rouge présentent une différence entre les deux colonnes. Cette différence est causée par la pré-insertion. Les valeurs trouvées sont 0x4 et 0x5, qui sont correspondant avec le tableau 5.4. Puis, les valeurs 0xAB et 0xCD sont retrouvées dans les deux tests, signifiant que la MAT fonctionne correctement dans la simulation et l'implémentation. Un bit d'assertion est relevé au prochain cycle d'horloge quand il y a une valeur trouvée.

```
     429        00000400000000000000000000000000        00000400000000000000000000000000
     430        ab800400000000000000000000000000        ab800400000000000000000000000000
     431        00000402000000000000000000000000        00000402000000000000000000000000
     432        00000400000000000000000000000000        00000400000000000000000000000000
     433        cd800400000000000000000000000000        cd800400000000000000000000000000
     434        00000480000000000000000000000000        00000480000000000000000000000000
     435        00000400000000000000000000000000        00000400000000000000000000000000
     436        ab800400000000000000000000000000        ab800400000000000000000000000000
     437        00000402000000000000000000000000        00000402000000000000000000000000
     438        00000400000000000000000000000000        00000400000000000000000000000000
     439        cd800400000000000000000000000000        cd800400000000000000000000000000
     440        00000480000000000000000000000000        00000480000000000000000000000000
mismatch count : 13
mismatch cycle nomber:  [7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
  2215.00ns INFO     cocotb.regression                         regression.py:487  in _log_test_summary
  2215.00ns INFO     cocotb.regression                         regression.py:557  in _log_test_summary
************************************

ME(NS)  REAL TIME(S)  RATIO(NS/S) **

************************************

15.00         11.10       199.59  **

************************************

  2215.00ns INFO     cocotb.regression                         regression.py:574  in _log_sim_summary
```

Figure 5.7 : Résultat de comparaison (MAT)

À la fin de la comparaison, un rapport fournit le nombre de cycles problématiques et liste leur numéro.

5.1.3 Le fichier avec les formes d'ondes

La figure 5.8 présente le fichier de forme d'onde régénéré. Dans le cadre jaune, le module dans le FPGA et celui en simulation peuvent être sélectionnés. Ensuite, dans la liste en-dessous, les signaux sont listés et peuvent être ajoutés. Les signaux sélectionnés sont affichés dans la partie droite de la fenêtre.

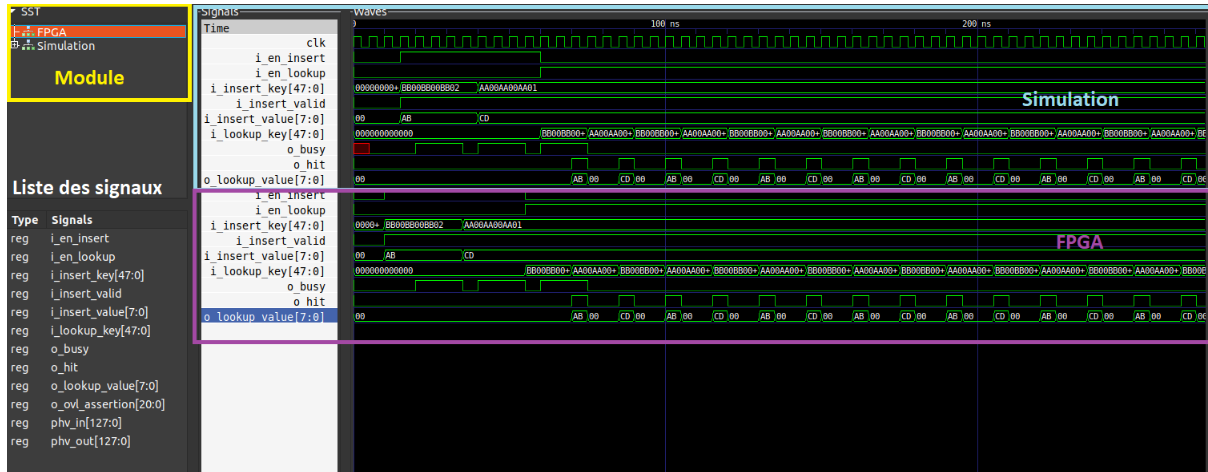


Figure 5.8: Formes d’ondes regénérées

5.2 Résultats sur la plateforme de Xilinx

La figure 5.9 présente l’architecture réalisée dans l’environnement de conception *Vivado*. L’architecture générale est disponible en annexe.

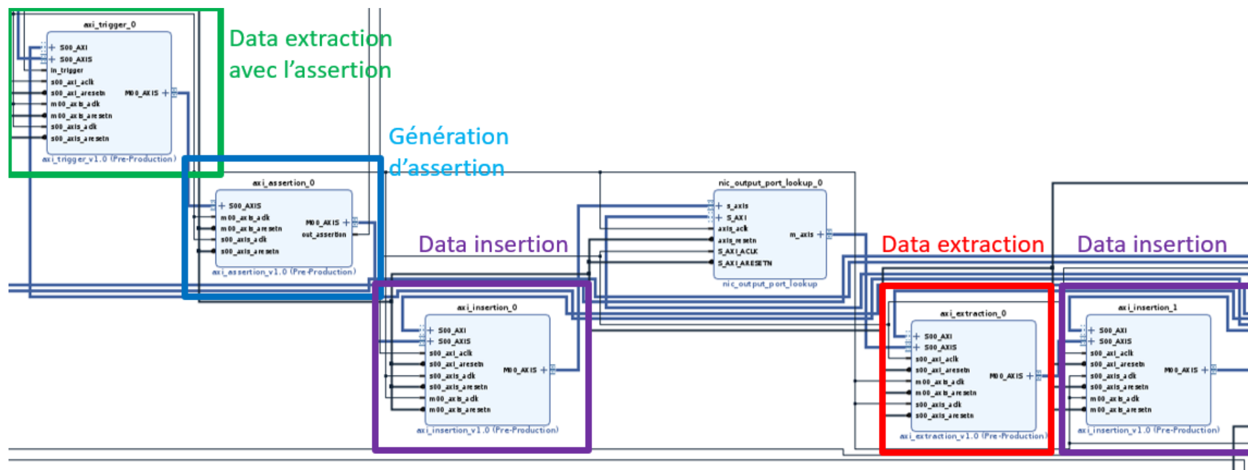


Figure 5.9 : Schéma de mise en œuvre réalisé avec *Vivado*

5.2.1 Configuration de l'implémentation

L'environnement de test est constitué de deux PC, chacun équipé d'une carte *NetFPGA*. Les ports nf0 et nf1 sont connectés ensemble comme présenté sur la figure 5.10. Les adresses MAC et IP correspondantes sont listées dans le tableau 5.3. La plupart du temps, la carte sur le PC2 agit comme

une carte NIC pour envoyer les paquets à tester. Le système à tester est implémenté sur la carte *NetFPGA-SUME* dans le PC1.

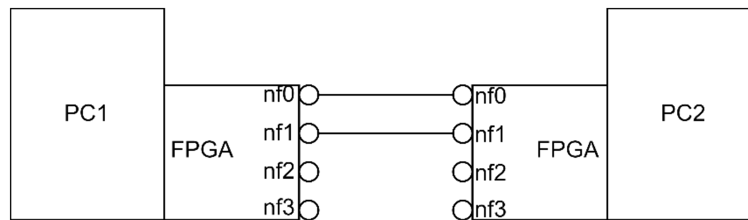


Figure 5.10 : Configuration de l'implémentation

Tableau 5.3 : Adresse MAC et IP

PC	Port	MAC	IP
PC1	nf0	02:00:00:00:37:00 20:35:55:4D:45:00	192.168.3.70
	nf1	02:00:00:00:37:01	192.168.3.71
PC2	nf0	02:00:00:00:38:00	192.168.3.80
	nf1	02:00:00:00:38:01	192.168.3.81

5.2.2 Test avec CocoTB

Pour tester le système avec *CocoTB*, un DUT simple qui ajoute '1' aux valeurs présentes sur les bus TDATA, TUSER et TKEEP est inséré dans le système. La figure 5.11 présente l'architecture pour ce test.

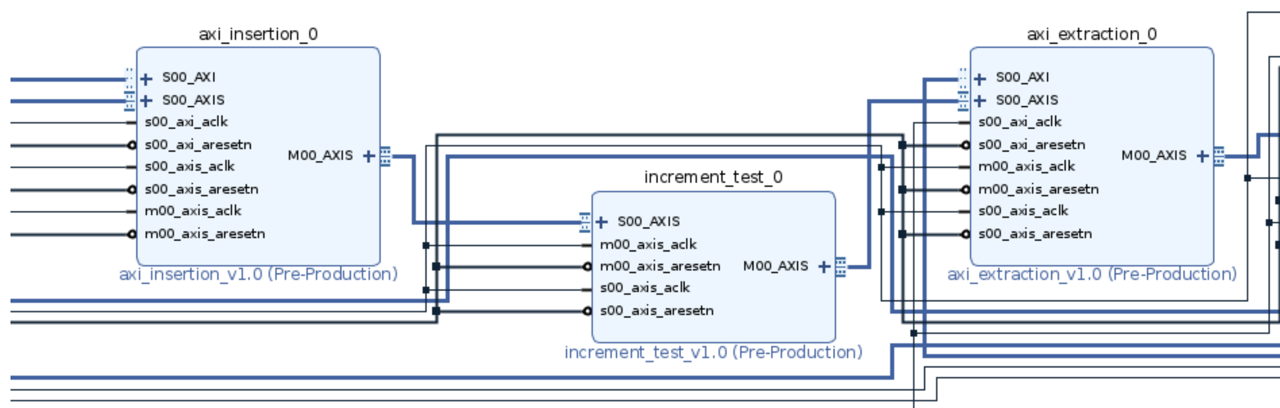


Figure 5.11 : Architecture pour le test avec *CocoTB*

La figure 5.12 décrit les travaux réalisés par le programme *CocoTB* et la figure 5.13 décrit le flux de travail du module de communication. Les données d'entrée définies dans le fichier sont d'abord envoyées au simulateur GHDL pour obtenir les résultats de simulation.

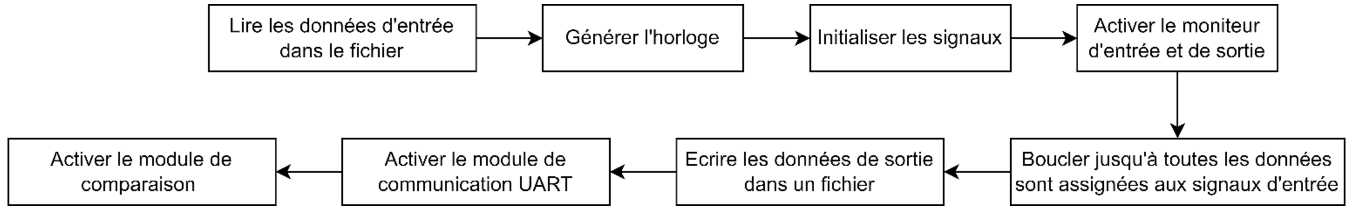


Figure 5.12 : Procédure dans *CocoTB*

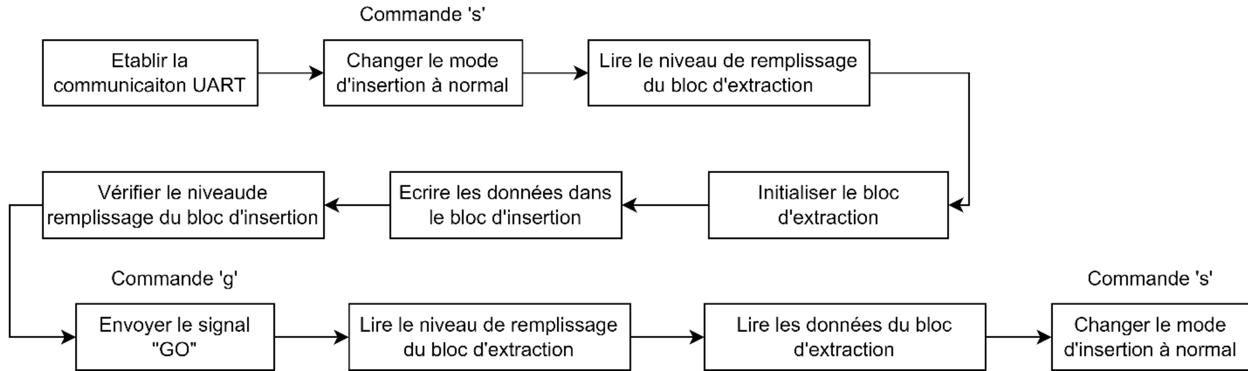


Figure 5.13 : Module de communication

Les données traversent l'interface UART pour se rendre au FPGA et faire le test dans le FPGA. Les résultats de l'implémentation physique sont ainsi obtenus. Le module de comparaison permet de donner un rapport selon les deux fichiers de sortie. La figure 5.14 présente le contenu récupéré dans chaque fichier. Dans le cadre jaune, les données sont augmentées de '1' à cause du DUT implémenté.

Données d'entrée

Données de sortie de simulateur

Données de sortie de FPGA

Figure 5.14 : Résultat de test produits avec *CocoTB*

5.2.3 Test du bloc d'extraction pour la vérification d'assertions

Ce test est réalisé avec l'architecture présentée sur la figure 5.9. Le but est de vérifier si le bloc d'extraction pour la vérification d'assertions peut récupérer les données ciblées.

Dans un premier temps, le message ciblé est « Find Key Word! » et le mot clé « Find » est retrouvé à la position 80 – 111 dans le bus TDATA. Une fois que le mot est détecté, le signal de sortie « *trigger* » est activé. Ce signal est connecté avec le signal d'entrée d'assertion du bloc d'extraction donc il s'agit d'un vérificateur d'assertion. Le flux de paquets est généré par deux programmes en Python : l'un envoie continuellement les paquets UDP avec un message « Hello World! », l'autre envoie une seule fois le message ciblé « Find Key Word! ».

```
498   020000003700020000003800080045000029CDCA40004011E512C0A80350C0A83C00010000000000000000000000000FFFFFFFF00000000
499   03469B7A1A0A0015812C48656C6C6F2C20576F726C64210000000000BC452F503C00010000000000000000000000000FFFFFF0F01000000
500   020000003700020000003800080045000029CDCB40004011E511C0A80350C0A83C00010000000000000000000000000FFFFFFFF00000000
501   03469B7A1A0A0015812C48656C6C6F2C20576F726C642100000000000004EBEB333C00010000000000000000000000000FFFFFF0F01000000
502   020000003700020000003800080045000029CDCC40004011E510C0A80350C0A83C00010000000000000000000000000FFFFFFFF00000000
503   03469B7A1A0A0015812C48656C6C6F2C20576F726C64210000000000680580093C00010000000000000000000000000FFFFFF0F01000000
504   020000003700020000003800080045000029CDCD40004011E50FC0A80350C0A83C00010000000000000000000000000FFFFFFFF00000000
505   03469B7A1A0A0015812C48656C6C6F2C20576F726C642100000000003C1D61673C00010000000000000000000000000FFFFFF0F01000000
506   020000003700020000003800080045000029CDCE40004011E50EC0A80350C0A83C00010000000000000000000000000FFFFFFFF00000000
507   03469B7A1A0A0015812C48656C6C6F2C20576F726C6421000000000056BDB64B3C00010000000000000000000000000FFFFFF0F01000000
508   0200000038000200000037000806000108000604000102000000370 0C0A803463C0002010000000000000000000000000FFFFFFFF00000000
509   000000000000C0A80350000000000000000000000000000000000000000000003C0002010000000000000000000000000FFFFFF0F01000000
510   020000003700020000003800080600010800060400020200000038 00C0A803503C00010000000000000000000000000FFFFFFFF00000000
511   020000003700C0A8034600000000000000000000000000000000000091188C023C00010000000000000000000000000FFFFFF0F01000000
512   02000000370002000000380008004500002AE16440004011D177C0A80350C0A83C00010000000000000000000000000FFFFFFFF00000000
513   0346CCAD1A0A001662A646696E64204B657920576F72642100000000 4487C58F3C00010000000000000000000000000FFFFFF0F01000000
514   0200000038000200000037000800 45C000463BAE00004001B662C0A80346C0A854000201000000000000000000000000FFFFFFFF00000000
515   0350030A85040000000004500002AE16440004011D177C0A80350C0A80346CCAD54000201000000000000000000000000FFFFFFFF00000000
516   1A0A001662A646696E64204B657920576F72642100000000000000000000000054000201000000000000000000000000FFFF0F0001000000
517   020000003700020000003800080045000029E64840004011CC94C0A80350C0A83C00010000000000000000000000000FFFFFFFF00000000
518   0346D55F1A0A001547474 8656C6C6F2C20576F726C64210000000000F64F768F3C00010000000000000000000000000FFFFFF0F01000000
519   020000003700020000003800080045000029E64940004011CC93C0A80350C0A83C00010000000000000000000000000FFFFFFFF00000000
520   0346D55F1A0A00154747 48656C6C6F2C20576F726C64210000000000ECB228643C00010000000000000000000000000FFFFFF0F01000000
```

Figure 5.15 : Données récupérées par le bloc d'extraction pour la vérification d'assertions

Comme présenté dans la figure 5.15, le paquet ciblé « Find Key World ! » se retrouve dans la ligne 512 qui correspond à la moitié de la profondeur de la FIFO (dans le cadre bleu). Les paquets ARP pour chercher les adresses sont dans le cadre rose. Le paquet « Hello World ! » se situe dans le cadre jaune.

Une analyse des données est réalisée sur les paquets ciblés donc les lignes 512 et 513. La figure 5.16 sépare les données issues des bus *AXI4-Stream*. La figure 5.17 analyse les protocoles différents utilisés par le paquet UDP : la partie jaune est le protocole Ethernet où se trouvent les adresses MAC et le type de protocole d'IP ; la partie bleue est le protocole IPv4 où les adresses IP

se situent ; le reste est le protocole UDP contenant le message ciblé. Dans le bus de metadonnées (TUSER), le premier octet 0x003C représente la longueur de paquet, soit 60 octets. L'octet prochain 0x01 représente le port de la source et le suivant est le port de destination 0x00. Le port source et le port de destination sont en encodage « *one-hot* », c'est-à-dire que chaque bit du champ représente un port différent, comme listé dans le tableau 5.4. Le paquet ciblé est donc venu du port MAC0 (nf0), à destination d'un port indéterminé. Le bit TLAST permet d'identifier la fin d'un paquet.

TDATA	TUSER	TKEEP	TLAST
02000000370002000000380008004500002AE16440004011D177C0A80350C0A8	3C000100000000000000000000000000	FFFFFFFF	00000000
0346CCAD1A0A001662A646696E64204B657920576F726421000000004487C58F	3C000100000000000000000000000000	FFFFFF0F	01000000

Figure 5.16 : Séparation de données analysées

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
02	00	00	00	37	00	02	00	00	00	38	00	08	00	45	00
00	2A	E1	64	40	00	40	11	D1	77	C0	A8	03	50	C0	A8
03	46	CC	AD	1A	0A	00	16	62	A6	46	69	6E	64	20	4B
65	79	20	57	6F	72	64	21	00	00	00	00	44	87	C5	8F

Figure 5.17 : Décodage d'un paquet

Tableau 5.4 : List des ports

Bit	0	1	2	3	4	5	6	7
Port	MAC0	CPU0	MAC1	CPU1	MAC2	CPU2	MAC3	CPU3

Pour simuler un scénario plus réaliste, un deuxième test est réalisé avec un détecteur de l'adresse MAC. Le vérificateur d'assertions est remplacé par un autre bloc permettant de détecter si l'adresse source MAC est le port 1 (nf1) de PC2 (02 :00 :00 :00 :38 :01). Cette information se situe 6 octets après le début de paquet donc aux bits 48 – 95 dans le bus de données. Pour identifier le début du paquet, le signal de fin de paquet TLAST est surveillé par ce bloc. C'est le même principe que pour le test précédent, les paquets sont envoyés continuellement par le port 0 et l'un des paquets est envoyé par le port 1. Dans la figure 5.18, le paquet à la ligne 512 est un paquet « Broadcast ARP » demandant l'adresse MAC de l'adresse 192.168.3.71 envoyé par l'adresse MAC 02 :00 :00 :00 :38 :01 (dans le cadre jaune). Le paquet avec l'adresse ciblée est trouvé à la moitié des données de sortie. Cela signifie que le système de détection de l'assertion et le bloc de collection des données ciblées fonctionnent correctement. Afin de simplifier le travail, nous avons créé des blocs de détection très ciblés, mais des tests aléatoires peuvent être réalisés pour améliorer la couverture.

```
505  0346C61E1A0A0015568848656C6C6F2C20576F726C64210000000000A4A12CFB3C000100000000000000000000000000FFFFFF0F01000000
506  0253554D4500020000003800080045000029F81B40004011BAC1C0A80350C0A83C000100000000000000000000000000FFFFFFFF00000000
507  0346C61E1A0A0015568848656C6C6F2C20576F726C642100000000001C0FE8983C000100000000000000000000000000FFFFFF0F01000000
508  0253554D4500020000003800080045000029F81C40004011BAC0C0A80350C0A83C000100000000000000000000000000FFFFFFFF00000000
509  0346C61E1A0A0015568848656C6C6F2C20576F726C6421000000000070E183A23C000100000000000000000000000000FFFFFF0F01000000
510  0253554D4500020000003800080045000029F81D40004011BABFC0A80350C0A83C000100000000000000000000000000FFFFFFFF00000000
511  0346C61E1A0A0015568848656C6C6F2C20576F726C64210000000000DF660EB73C000100000000000000000000000000FFFFFF0F01000000
512  FFFFFFFFFFFF020000003801080600010800060400010200000038 01C0A803513C000400000000000000000000000000FFFFFFFF00000000
513  000000000000C0A80347000000000000000000000000000000000000C29E19313C000400000000000000000000000000FFFFFF0F01000000
514  0200000038010253554D4501080600010800060400020253554D4501C0A803473C000804000000000000000000000000FFFFFFFF00000000
515  020000003801C0A80351000000000000000000000000000000000000000000003C000804000000000000000000000000FFFFFF0F01000000
516  0253554D4501020000003801080045000 02A2BB340004011 8727C0A80351C0A83C000400000000000000000000000000FFFFFFFF00000000
517  0347D7641A0A001657ED46696E64204B657920576F726421000000002ED5737B3C000400000000000000000000000000FFFFFF0F01000000
518  0200000038010253554D4501080045C000465489000040019D85C0A80347C0A854000804000000000000000000000000FFFFFFFF00000000
519  0351030A8506000000004500002A2BB3400040118727C0A80351C0A80347D76454000804000000000000000000000000FFFFFFFF00000000
520  1A0A001657ED46696E64204B657920576F72642100000000000000000000000054000804000000000000000000000000FFFF0F0001000000
521  0200000038010253554D4501080600010800060400010253554D4501C0A803473C000804000000000000000000000000FFFFFFFF00000000
522  000000000000C0A80351000000000000000000000000000000000000000000003C000804000000000000000000000000FFFFFF0F01000000
523  0253554D450102000000380108060001080006040002020000003801C0A803513C000400000000000000000000000000FFFFFFFF00000000
```

Figure 5.18 : Données de sortie du test avec le détecteur de l'adresse MAC

5.2.4 Ajout du compteur d'horloge

Comme mentionné dans l'explication de l'interface *AXI4-Stream*, le bus de données est contrôlé par deux signaux de contrôle « ready » et « valid ». Le bloc d'extraction collecte seulement les données valides donc le débit de données est variable. Pour mesurer le débit, un bloc de compteur de cycles d'horloge est finalement ajouté dans le système (voir la figure 5.19).

Figure 5.19 : Compteur de l'horloge

Ce bloc incrémente un nombre de 64 bits à chaque cycle d'horloge. Une fois qu'un nouveau paquet est arrivé contrôlant le signal TLAST, le nombre de cycles d'horloge est envoyé dans une FIFO.

Ce bloc équipe aussi l'interface *AXI4-Lite* pour communiquer avec le processeur. Les commandes 'k' et 'h' sont distribuées pour faire la lecture du niveau de remplissage et des données. Un programme permet de lire les données suivant la même logique que celle décrite dans la figure 4.30. Pour analyser les temps d'arrivée des paquets, un analyseur simple est écrit en Python. Il utilise la longueur des paquets pour calculer le débit et analyser la distribution de données. Afin de rendre la figure plus lisible, le programme supprime les valeurs excessives.

Pour un flux de paquet avec le message « Hello World ! », la distribution des temps d'arrivée des paquets est présentée dans la figure 5.20.

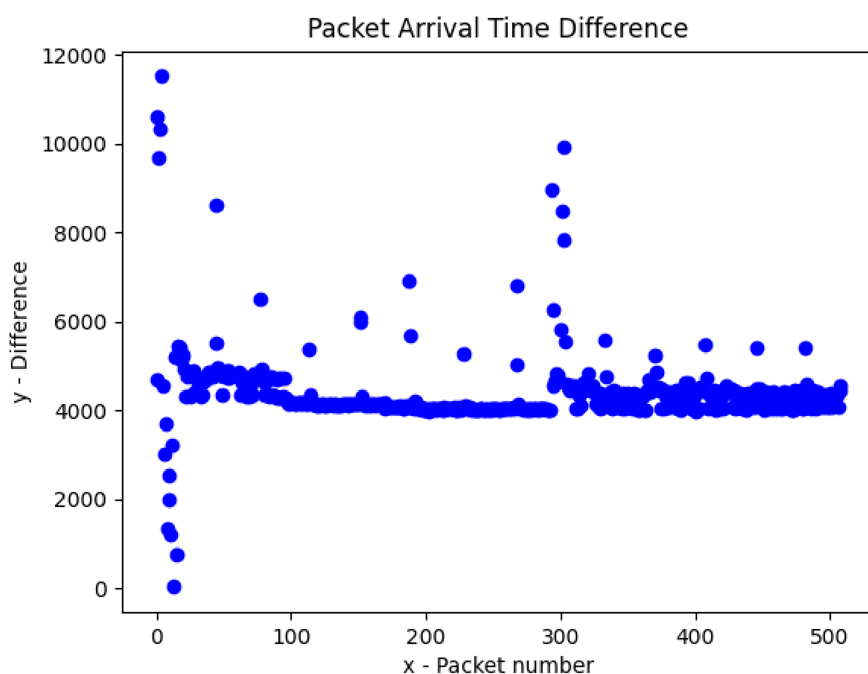


Figure 5.20 : Distribution des temps d'arrivé de paquet

Le temps d'arrivée moyen est d'environ 4386, c'est-à-dire chaque 4386 cycles, un paquet est reçu par la carte. La longueur des paquets est 64 octets, soit 2 cycles pour chacun. Le débit est estimé à 18.67Mbps. Cela est raisonnable, car le programme Python utilisé pour générer les paquets utilise le noyau pour construire les paquets et puis les envoie vers l'interface réseau. Il ne supporte pas les flux à haut débit. En plus, les paquets utilisés ont une taille très petite, ce qui peut aussi limiter la bande passante.

5.3 Utilisation des ressources

Le tableau 5.5 rassemble les rapports d'utilisation des ressources pour chaque bloc créé. La profondeur de FIFO dans ces blocs est configurée à 1024.

Tableau 5.5: Utilisation des ressources

Bloc	LUT*	BRAM**	Registres
Extraction	247	12	1025
Extraction avec l'assertion	308	24	1509
Insertion	330	12	1012
NetFPGA-SUME disponible	433200	1470	866400

* Lookup Table **Block RAM

Les blocs occupent très peu de ressources. Chaque bloc prend moins de 1% des LUT et des registres disponibles dans le FPGA. Plus de 1% de BRAM est utilisée, car la FIFO en a besoin. Le FPGA monté sur la carte *NetFPGA-SUME*, *Virtex-7 XC7VX690T*, a 1470 blocs BRAM de 36ko. L'utilisation de BRAM est déterminée par la taille des FIFOs et peut être calculée selon la profondeur. Selon la datasheet de Block RAM, la largeur et la profondeur maximales d'une BRAM de 36 Ko sont de 72 bits×(512+1). Dans le cas d'exemple, la largeur est 417 bits et la profondeur est 1024. Donc le nombre de BRAM occupé pour adapter la largeur est $417/72=5.79\approx6$, pour la profondeur $1024/(512+1)\approx2$. Une FIFO de 417×1024 a besoin de 6×2=12 blocs de BRAM de 36ko.

CHAPITRE 6 DISCUSSION

Les recherches récentes sur les testeurs de réseaux se concentrent sur la vérification des équipements entiers. Les travaux sur la génération et la surveillance de paquets se concentrent principalement sur des solutions à débit élevé, l'obtention d'une faible latence et les mesures de latence avec une précision à la nanoseconde. Ces testeurs sont normalement implémentés à l'extérieur du DUT. Ainsi, seuls les paquets aux ports d'entrée et de sortie peuvent être analysés. Cependant, il est important d'avoir accès aux signaux situés à l'intérieur du DUT pour une meilleure observabilité.

Avec la solution proposée, les données brutes peuvent être capturées à partir du DUT, ou être insérées dans le DUT par l'environnement, et les données de sortie sont collectées pour observation. Comme des modules de test sont insérés dans le circuit à tester, l'environnement permet d'opérer directement sur les signaux internes. Grâce aux blocs d'insertion et d'extraction insérés, les données s'échangent entre un ordinateur hôte et un FPGA. Avec cet environnement, un banc de test physique est donc réalisé. Cette solution facilite la détection de bogues de synchronisation durant la simulation RTL, en effectuant le même test en matériel. La visibilité de la simulation est gardée dans les formes d'ondes avec les signaux obtenus par le test en FPGA. En insérant les blocs d'interfaces, cette solution permet un test complet. De plus, un signal d'assertion d'entrée est maintenant supporté afin que les données ciblées puissent être enregistrées et rejouées pour aider à découvrir d'éventuels bogues. Les opérateurs de réseau peuvent utiliser ces données pour reproduire les scénarios problématiques.

L'environnement a été implémenté sur deux plateformes différentes. À cause de la structure différente entre deux cartes utilisées dans ce mémoire, les détails de l'architecture ont dû être modifiés. La conception sur la plateforme d'Intel contient les interfaces de taille configurable, un bloc compteur et une interface pour adapter la MAT. La communication entre le HPS et le FPGA est l'un des points les plus importants dans cet environnement. La version implémentée prend un bus de données de 128 bits, ce qui impose donc 4 fois d'écritures à faire dans le HPS pour un cycle de données. La transmission en *socket* prend encore plus de temps. Par contre, le passage de données vers le DUT se fait cycle par cycle. En ajoutant des FIFOs pour enregistrer les données, cette architecture résout le problème de différence entre l'efficacité de transmission de la

partie logicielle et celle de la partie matérielle. Ce système est testé avec une « MAT » en ajoutant une interface de connexion simple.

Toutefois, une difficulté dans la partie de traitement des formes d'ondes est la synchronisation des signaux. Les sorties récupérées du FPGA contiennent seulement des données valides, mais le simulateur génère les données à partir de l'initialisation du module. Cela peut causer des désynchronisations dans le fichier de formes d'ondes.

L'environnement de test basé sur la plateforme Xilinx suit le même principe. Mais, cette carte est spécifique au développement d'équipements réseau permettant de traiter les données brutes. La partie HPS est donc enlevée et est remplacée par le processeur embarqué *Microblaze*. Les données n'ont pas besoin de passer par le *socket* Internet, mais sont envoyées directement par un UART. De plus, un bus de données plus large et un bus de métadonnées à côté offrent plus de possibilités pour stimuler le DUT.

À part cela, les blocs d'extraction récupèrent les données en temps réel et permettent de déboguer durant le fonctionnement. Ce système occupe très peu de ressources et les utilisateurs peuvent ajouter des blocs selon la nécessité. Quand ils ne sont pas requis, les blocs sont transparents dans le système après une simple configuration.

Le tableau 6.1 présente une comparaison des caractéristiques de nos travaux avec différentes solutions publiées. Les caractéristiques d'intérêt incluent le support pour la simulation, le débit, la génération de paquets, la relecture de paquets et la précision des travaux existants. Au lieu de rejouer tous les trafics comme avec le projet FlueNT10G [15], notre outil proposé permet soit de rejouer tous les trafics, soit de reprendre seulement les données ciblées à l'aide de l'assertion. Cela simplifie le débogage. En plus, certains outils permettent de fonctionner avec les quatre ports sur la carte *NetFPGA*, donc ils supportent un débit de 40 Gbit/s. Dans notre cas, un seul port est utilisé durant le test, où le débit est de 10 Gbit/s, mais ce n'est pas une limitation réelle de la solution proposée. Notre outil est le seul qui supporte une simulation comparative. Elle permet de fournir une haute observabilité des signaux internes. Nous offrons une précision d'un cycle d'horloge qui est 6.4ns sur la carte *NetFPGA-SUME.* La plupart des outils ont cette même limitation, sauf le projet OP4T [7] qui peut opérer à une fréquence de 250MHz. La génération de paquets de notre outil se réalise par le programme Python. Les utilisateurs peuvent construire des données à l'aide

d'un générateur logiciel ou par écriture manuelle. Cette façon de faire offre une haute programmabilité en ce qui a trait aux trafics à tester en comparaison avec les autres outils.

Tableau 6.1 : Comparaison avec les projets de recherche existants

	Outil proposé	FlueNT10G [15]	OP4T [7]	Formullar [16]
Combinaison avec la simulation	Oui	Non	Non	Non
Débit	1x10 Gbit/s	3 ou 4x10 Gbit/s	Dépendre de la plateforme utilisée	1x10 Gbit/s
Génération de paquet	Partiel à l'aide de l'assertion ou complet	Rejouer les trafics	Non	Basé sur Modèle
Rejouer des paquets	Oui	Oui	Non	Basé sur Modèle
Précision	Cycle d'horloge (6.4ns)	6.4ns	Cycle d'horloge (250MHz Maximal donc 4 ns)	6.4ns

CHAPITRE 7 CONCLUSION ET RECOMMANDATIONS

Le travail présenté dans ce mémoire propose un environnement de test qui combine la modélisation logicielle et la validation matérielle pour vérifier les 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.

7.1 Synthèse des travaux

L'architecture combine la simulation d'un module à l'essai avec la validation d'une implémentation sur FPGA ainsi que la communication entre eux. Le but est de passer le même test dans un simulateur et sur une version d'un modèle mis en œuvre sur un FPGA pour mieux vérifier le fonctionnement d'une application. L'environnement de test a été implémenté sur deux plateformes FPGA : le *DE10-Standard* d'Intel et le *NetFPGA-SUME* de Xilinx. Dans les deux plateformes, la partie basée sur la simulation reste semblable en utilisant le banc de test *CocoTB* et le simulateur GHDL. L'utilisateur écrit le programme d'un banc de test qui définit les données d'entrée des ports du DUT. Les moniteurs d'entrée et de sortie collectent les données des ports du DUT et les enregistrent dans des fichiers. L'environnement fournit aussi des entrées pour insérer les assertions. L'utilisateur peut ajouter des modules d'assertion dans la plateforme permettant de capturer les données ciblées.

La carte *DE10-Standard* est une carte de développement à usage général. À cause de la structure de la carte, les données entrant par le port Internet sont forcées de passer par le HPS. Ce dernier exécute un système d'exploitation Linux et prend en charge la configuration et l'échange de données du port Ethernet. Dans ce cas, la communication entre un ordinateur hôte et la carte FPGA est réalisée par des *sockets*. Le programme exécuté dans le HPS agit comme un serveur. Il attend les connexions venues de l'ordinateur hôte. Chaque interface dans le FPGA correspond à un *socket*. Cela garantit que l'échange de données sur différentes interfaces peut avoir lieu simultanément. Sur la plateforme Intel, l'interface utilisée est celle de la famille d'*Avalon*. Les blocs d'interfaces sont créés avec des ports *Avalon Memory Mapped* qui servent à communiquer avec le HPS et le mécanisme *Avalon Streaming* qui gère les données transférées au DUT et celles reçues du DUT.

Les données d'entrées définies par l'utilisateur traversent les interfaces, le programme à tester et puis reviennent dans le programme *CocoTB*.

Une seconde plateforme utilisée dans cette recherche est la *NetFPGA-SUME* qui est une carte spécifique au développement des applications réseau. Les données brutes reçues par ses ports Internet peuvent être traitées directement. Sans passer par le HPS, les données, dans ce cas, traversent l'interface UART et sont stockées dans un processeur embaqué qui gère le système de façon logicielle. Avec ces nouvelles caractéristiques, l'environnement permet de collecter les données en temps réel.

7.2 Limitations et contraintes

L'implémentation de cet environnement possède deux limitations précédemment exprimées. La première est la consommation de ressources. Comme la partie matérielle est insérée dans le FPGA avec le DUT et que les données sont stockées dans des FIFOs, les ressources occupées dépendent du volume nécessaire pour le test. Avec une taille petite, par exemple, 1024 cases implémentées dans ce mémoire, les blocs occupent très peu de place. Mais cette occupation augmente lorsque le volume de données s'élève.

La deuxième est la gestion des mises à jour. Sur la carte *DE10-Standard*, chaque fois qu'il y a une reconfiguration des paramètres, le flux de compilation, synthèse et implémentation est exécuté. Sur la carte *NetFPGA-SUME*, une modification devrait se faire sur le bloc d'interconnexion des bus *AXI4-Lite* si le nombre de blocs insérés augmente. Cela prend longtemps et à cause de la reprogrammation de l'interface PCIe, une mise à jour demande un redémarrage de l'ordinateur hôte après chaque programmation de la *NetFPGA*. De plus, la complexité du programme dans le HPS est aussi limitée par la RAM distribuée.

7.3 Travaux futurs

Cette section présente les travaux futurs. Le chapitre 5.2.4 décrit un bloc de compteur d'horloge. Dans la solution proposée ici, les temps d'arrivée sont enregistrés dans le compteur contenant une FIFO indépendante. Dans une suite à cette recherche, ces informations pourraient être ajoutées sur un bus de métadonnées. Cela permettrait d'ajouter un marqueur de temps pour chaque paquet arrivé

pour ainsi réduire les ressources occupées. De plus, il serait intéressant de développer l'interface utilisateur afin d'en faciliter l'utilisation.

RÉFÉRENCES

[1] The Open Networking Foundation, “Software-Defined Networking: The New Norm for Networks”, April 2012

[2] BOSSHART, Pat, DALY, Dan, GIBB, Glen, et al. P4: Programming protocol-independent packet processors. ACM SIGCOMM Computer Communication Review, 2014, vol. 44, no 3, p. 87-95.

[3] Intel® Tofino™ 2 : Second-generation P4-programmable Ethernet switch ASIC that continues to deliver programmability without compromise

[4] P4-SDNet User Guide https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1252-p4-sdnet.pdf

[5] KeySight Technologies. 2019. IxNetwork. Retrieved October 2021 from https://www.keysight.com/ca/en/products/network-test/protocolload-test/ixnetwork.html

[6] ANTICHI, Gianni, SHAHBAZ, Muhammad, GENG, Yilong, et al. OSNT: Open source network tester. IEEE Network, 2014, vol. 28, no 5, p. 6-12.

[7] HAWARI, Mohammed et CLAUSEN, Thomas. OP4T: Bringing Advanced Network Packet Timestamping into the Field. In: 2021 International Conference on Information Networking (ICOIN). IEEE, 2021. p.137-142.

[8] Intel, Inc. Intel Quartus Prime Pro Editio User Guide, Debug Tools, UG-20139, October 2021.

[9] Xilinx, Inc. Chipscope Pro Software and Cores: User Guide, October 2012.

[10] BMV2 Release 1.14. 2020. Retrieved October 2021 from https://github.com/p4lang/behavioral-model

[11] NetFPGA GitHub Organization, https://github.com/NetFPGA/NetFPGASUME-public/wiki/NetFPGA-SUME-Reference-NIC-Vivado-2020.1-and-Ubuntu-2020.4

[12] AXI Reference Guide https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

[13] COroutine based COsimulation TestBench, https://docs.cocotb.org/en/stable/

[14] Open Verification Library (OVL) Working Group https://www.accellera.org/activities/working-groups/ovl

[15] OELDEMANN, Andreas, WILD, Thomas, et HERKERSDORF, Andreas. FlueNT10G: A programmable FPGA-based network tester for multi-10-gigabit ethernet. In : 2018 28th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2018. p.178-1787.

[16] PARK, Taejune, SHIN, Seungwon, SHIN, Insik, et al. Formullar: An FPGA-based network testing tool for flexible and precise measurement of ultra-low latency networking systems. Computer Networks, 2021, vol. 185, p. 107689.

[17] Spirent: Automated Testing and Assurance Solutions https://www.spirent.com/

[18] Siemens EDA Software - Questa Advanced Simulator https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/

[19] ATTIA, Sameh et BETZ, Vaughn. StateMover: Combining simulation and hardware execution for efficient FPGA debugging. In : Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2020. p. 175-185.

[20] STRASNICK, Evan, AGRAWALA, Maneesh, et FOLLMER, Sean. Coupling Simulation and Hardware for Interactive Circuit Debugging. In : Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. 2021. p. 1-15.

[21] Python Serial Port Extension for Win32, OSX, Linux, BSD, Jython, IronPython: pyserial 3.5 https://pypi.org/project/pyserial/

[22] ZET, Cristian et FOSALAU, Cristian. FPGA Based Logic Analyzer. In: 2018 International Conference and Exposition on Electrical And Power Engineering (EPE). IEEE, 2018. p. 0335-0340.

[23] POPA, Stefan, IVANOVICI, Mihai, et COLIBAN, Radu-Mihai. Time-multiplexed 10Gbps Ethernet-based Integrated Logic Analyzer for FPGAs. In: 2020 International Symposium on Electronics and Telecommunications (ISETC). IEEE, 2020. p. 1-4.

[24] O'DELL, Devon H. The Debugging Mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. Queue, 2017, vol. 15, no 1, p. 71-90.

[25] Mengyue Su, Jean-Pierre David, Yvon Savaria, Bill Pontikakis, Thomas Luinaud. An FPGA-based HW/SW Co-Verification Environment for Programmable Network Devices. Publiera prévu en ISCAS 2022.

[26] Patrick Richer St-Onge. Mémoire de maitrise en préparation, Ecole Polytechnique Montréal 2022.

[27] Build fast, reliable, and efficient software at scale, GO language, https://go.dev/.

[28] GHDL: free and open-source analyzer, compiler, simulator and (experimental) synthesizer for VHDL https://ghdl.github.io/ghdl/index.html

[29] Property Specification Language http://www.project-veripage.com/psl_tutorial_1.php

[30] Threading — Thread-based parallelism https://docs.python.org/3/library/threading.html

[31] Jason R. Andrews, Chapter 4 - Hardware/Software Co-Verification, Co-verification of Hardware and Software for ARM SoC Design, Newnes, 2005, Pages 119-163, ISBN 9780750677301, https://doi.org/10.1016/B978-075067730-1/50009-8.

[32] What Is LabVIEW? - Graphical programming environment. https://www.ni.com/en-us/shop/labview.html#

[33] Pypi - vcdvcd 2.3.2 https://pypi.org/project/vcdvcd/

[34] Pyvcd documentation ! - vcd.writer https://pyvcd.readthedocs.io/en/latest/index.html

ANNEXES

Annexe A – Schéma du système sur plateforme de Xilinx