Etude et mise en oeuvre d'un environnement d'exécution pour architecture hétérogène reconfigurable

Jonathan Dechelotte

université de BORDEAUX

THÈSE PRÉSENTÉE
POUR OBTENIR LE GRADE DE

DOCTEUR DE L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE N° 209 : SCIENCES DE L'INGÉNIEUR
SPÉCIALITÉ : ÉLECTRONIQUE

présenté par

M. Jonathan DÉCHELOTTE

Etude et mise en oeuvre d'un environnement d'exécution pour architecture hétérogène reconfigurable

préparée au Laboratoire IMS

Soutenue le **12/03/2020** devant le jury composé de :

Lilian BOSSUET	-	Professeur	-	Université Jean Monnet — *Rapporteur*
Jérémie CRENNE	-	Maître de Conférences	-	Bordeaux INP — *Co-encadrant*
Dominique DALLET	-	Professeur	-	Bordeaux INP — *Directeur*
Jean-Philippe DIGUET	-	Directeur de Recherche	-	Université Bretagne Sud — *Examinateur*
Guy GOGNIAT	-	Professeur	-	Université Bretagne Sud — *Rapporteur*
Russell TESSIER	-	Professor	-	University of Massachusetts — *Examinateur*

Thèse réalisée au

Laboratoire de l'Intégration du Matériau au Système (IMS)
de Bordeaux, au sein de l'équipe CSN du groupe Conception.

Université de Bordeaux, Laboratoire IMS
UMR 5218 CNRS - Bordeaux INP
351 Cours de la Libération
Bâtiment A31
33405 Talence Cedex
FRANCE

À ma famille,

Résumé

Aujourd'hui, les systèmes embarqués ont pris une part hégémonique dans notre monde. Leurs utilisation est prépondérante, que ce soit pour communiquer, se déplacer, travailler ou se divertir. Des efforts dans le domaine de la recherche et de l'industrie n'ont de cesse de faire évoluer les parties qui composent ces systèmes dont le processeur, le *FPGA*, la mémoire et le système d'exploitation. D'un point de vue architectural, l'apport d'une architecture généraliste couplée à une architecture reconfigurable positionne le *SoC FPGA* comme une cible préférentielle pour une utilisation dans les systèmes embarqués. Leurs adoption est cependant difficile du fait de leur complexité d'implémentation. L'abstraction des couches de bas niveau semble un axe d'investigation qui tend à inverser cette tendance. Au premier abord, l'utilisation d'un système d'exploitation paraît idoine. En effet, il possède l'écosystème de drivers et services disponibles pour l'accès aux ressources matérielles, la capacité d'ordonnancement natif ainsi que des bibliothèques pour la sécurité. Toutefois, cette solution engendre des contraintes qui poussent à évaluer d'autres approches. Ce manuscrit évalue la capacité d'un langage de haut niveau tel que Lua à fournir un environnement d'exécution dans le cas d'une implémentation sans système d'exploitation. À travers un écosystème nommé Lynq, cet environnement d'exécution procure les briques nécessaires à la gestion et l'allocation des ressources présentes sur le *SoC FPGA*, ainsi qu'une méthode proposant une isolation entre applicatifs. La capacité des architectures généralistes que sont les *CPU*s à devenir spécialisés lorsqu'ils sont implémentés sur un *FPGA* a été exploré par la suite, ceci au travers d'une contribution permettant la génération d'un *CPU* RISC-V ainsi que son microcode associé.

Mots clefs : Systèmes Embarqués, *FPGA*, *SoC FPGA*, Lua, Architecture Reconfigurable, RISC-V, Microcode ;

Abstract

Today, embedded systems have taken a leading role in our world. Whether for communication, travel, work or entertainment, their use is preponderant. Together, research and industry efforts are constantly developing various parts that make up these systems : processor, *FPGA*, memory, operating system. From an architectural point of view, the contribution of a generalist architecture coupled with a reconfigurable architecture positions *SoC FPGA*s as popular targets for use in embedded systems. However, their implementation's complexity makes their adoption difficult. The abstraction of low-level layers seems to be an investigation's axis that would tend to reverse this trend. The use of an operating system seems suitable at first glance because they deliver an ecosystem of drivers and services for access to hardware resources, native scheduling capacities and libraries for security. However, this solution brings constraints and lead to evaluate other approaches. This manuscript evaluates the ability of a high-level language, Lua, to provide an execution environment in such a case that the implementation does not provide operating system. It gives, through an ecosystem named Lynq, the necessary building blocks for the management and allocation of resources present on the *SoC FPGA* as well as a method for isolation between applications. Besides the adoption of this execution environment, our work explores the capacity of generalist architectures such as *CPU*'s to become specialized when implemented on a *FPGA*. This is done through a contribution allowing the generation of a RISC-V *CPU*and its associated microcode.

Keywords : Embedded Systems, *FPGA*, *SoC FPGA*, Lua, Reconfigurable Architecture, RISC-V, Microcode ;

Table des matières

Table des figures

Liste des tableaux

Introduction

Contexte

Notre monde est de plus en plus tourné vers le numérique et les technologies associées ont pris une part prépondérante dans les activités industrielles, mais également dans notre société. Pour faire face à cette numérisation à grande vitesse, des efforts ont été fournis sur les outils et technologies permettant de développer rapidement des architectures matérielles dédiées, dont les performances surpassent celles des architectures généralistes sans connaissance spécifique. Rentabilité énergétique, performance, sécurité et sûreté sont les maîtres mots d'un marché en forte croissance et en constante évolution. À ces besoins s'ajoute une problématique de coût dans une industrie toujours plus concurrentielle.

Pour se rapprocher au mieux d'une solution idéale, des modèles d'architectures ont émergé, et plus particulièrement celui du *SoC FPGA*. Alliant architectures reconfigurables et généralistes, elle permet l'agrégation en une architecture hétérogène de deux architectures aux domaines d'utilisation initialement différents. Cette nouvelle solution a pour intérêt d'utiliser un bloc matériel dédié car il est développé sur mesure. Elle est couplée à un composant généraliste, généralement un *CPU*, qui peut être à la fois utilisé comme organe de calcul ou de contrôle. Dans le cadre de l'utilisation d'un *CPU hardcore* à des fins calculatoires, il peut être intéressant d'utiliser un *CPU softcore* pour le contrôle de l'architecture. L'utilisation d'un *CPU* microcodé qui implémente un jeu d'instructions *au plus juste* permet également *in fine* d'avoir la maîtrise des instructions exécutables.

Environnement d'exécution

Ce manuscrit de thèse utilisera comme base de travail le *SoC FPGA* avec comme objectif de proposer un environnement d'exécution sur *SoC FPGA* pour tirer profit des capacités de l'architecture hétérogène reconfigurable cible. Cela devra se faire en respectant les besoins de rentabilité énergétique, de performance calculatoire, de sûreté et de sécurité.

Rentabilité énergétique

De nos jours, la complexité des architectures est croissante, notamment au niveau des couches de bas niveau. On y retrouve des microarchitectures processeur et des architectures à base de cellules logiques sur *FPGA*. Cela résulte d'un compromis lors de l'optimisation architecturale entre le gain de performance et la diminution de consommation. Cependant, la rentabilité énergétique tient compte de l'architecture en elle-même, tout comme de son utilisation. Celle-ci doit permettre de tirer profit des spécificités architecturales afin d'optimiser le rapport performance / énergie consommée.

Performance calculatoire

Plusieurs niveaux de performance peuvent être étudiés. Lors de la phase de développement d'un accélérateur matériel, nous nous intéressons à sa performance propre. Or, celui-ci a pour vocation d'être utilisé dans un environnement plus général pour un contexte d'exécution spécifique. C'est pourquoi, un second niveau d'évaluation de la performance en découle. Il est basé sur l'optimisation du contexte d'exécution dont l'objectif est de gérer l'utilisation des différents accélérateurs matériels et programmes logiciels. La finalité est de pouvoir appliquer un schéma d'exécution proposant des optimisations en performance.

Sécurité

La sécurité est devenue primordiale dans le processus de conception des architectures. En effet, elle protège le développeur du vol de sa propriété intellectuelle ou l'utilisateur pendant l'utilisation de celle-ci. L'intégration de primitives qui garantissent la sécurité du système est complexe car elle doit apporter une protection contre des vecteurs d'attaques pouvant être d'origine logicielle, matérielle ou mixte. Ces éléments sont à considérer pendant la phase de conception. De ce fait, cela complexifie le processus alors que celui-ci est déja contraint par des problématiques liées à leur intégration.

Notre environnement d'exécution posé, nous avons étudié les possibilités offertes par les architectures *CPU* microcodées. L'objectif étant d'explorer les apports offert par l'implémentation sur une architecture reconfigurable de type *FPGA* d'une architecture généraliste de type *CPU* spécialisée via le microcode. Pour cela, nous avons développé un outil de génération d'architecture à partir de la définition des instructions à implémenter.

Génération d'architecture microcodée

La génération d'architecture est un axe de recherche grandissant, notamment dans le domaine de la High Level Synthesis (*HLS*). Les études relatives à ce sujet ont été mises de côté par l'écosystème de recherche universitaire après avoir été étudié dans les années 1980. Cependant, les industriels ont maintenu leurs efforts de recherche [9] sans pour autant démontrer leurs savoir au vu des architectures *CPU* disponibles dans le commerce. À travers cela, ils asseoient leur propriétés intellectuelles et leur parts de marché. Néanmoins et depuis peu, de nombreux travaux rapportent les efforts des chercheurs dans le *reverse engineering* [10, 11]. Ceci démontre un regain d'intérêt pour la compréhension du microcode présent dans les architectures *CPU* actuelles, notamment pour répondre à des problématiques relatives à la sécurité [12].

Avec le développement de la spécification du jeu d'instruction RISC-V [13] open-source, le développement d'un *CPU* implémentant ce jeu d'instruction de manière microcodé semble pertinent. Tant pour les aspects qui permettraient une automatisation de cette génération que sur des aspects qui ouvriraient la voie à l'utilisation d'outil de vérification formelle.

Aperçu des résultats

Ce manuscrit présente les résultats du développement d'un environnement d'exécution répondant à des problématiques de performance calculatoire et de sécurité pour architecture hétérogène reconfigurable. Celui-ci offre plus largement les outils et services nécessaires au contrôle des accélérateurs matériels, des périphériques et des ressources logicielles présentes dans l'architecture depuis un langage de programmation de haut niveau à partir d'une librairie modulaire permettant l'uniformisation des méthodes d'accès. Le développement de cet environnement d'exécution a suivi la méthodologie suivante. Dans un premier temps le coeur de cet environnement, basé sur le langage Lua, a été développé. Celui-ci offre un accès aux accélérateurs matériels, aux ressources logicielles ainsi qu'aux périphériques du système en fournissant une couche d'abstraction matérielle via une bibliothèque. Chaque module de cette bibliothèque peut être chargé pour permettre l'utilisation des différentes fonctionnalités telles que la reconfiguration du FPGA, l'accès aux services réseau ou encore à la gestion des transferts mémoires. La littérature n'offrant pas de solution comme celle-ci sans l'utilisation d'un système d'exploitation, nous avons également développé les couches bas niveau permettant de s'en abstraire. Ceci permet d'améliorer la rentabilité énergétique de notre environnement d'exécution en proposant une gestion fine de la consommation. Dans un second temps, l'effort s'est concentré sur

l'optimisation de l'environnement pour en améliorer les performances globales. En effet, il a pour but d'améliorer non pas la performance intrinsèque des accélérateurs matériels et ressources logicielles, mais l'amélioration de la gestion et de l'allocation dynamique de ses ressources à des fin d'utilisation des slots disponibles via un multiplexage temporel. Pour accroître la rentabilitée des ressources disponibles sur le *SoC FPGA*, nous avons développé une version modifiée de l'algorithme dit *Completely Fair Scheduler* (*CFS*). Elle prend en compte la gestion de ressources de différents types, à savoir matériels ou logiciels, et arbitre leurs allocation. Puis, dans un troisième temps la finalisation de l'environnement d'exécution traite des aspects sécurité. Notre travail s'est axé sur l'ajout de primitives pour permettre l'isolation matérielle et logicielle complète de plusieurs environnements d'exécution sur une même architecture. Enfin, des travaux relatifs à la génération d'architecture et la génération de microcode associé sont présentés dans la dernière partie de ce manuscrit. Ceux-ci ont vocation à proposer une solution compléte afin de génèrer d'une part le code *HDL* permettant l'implémentation du *CPU* et, d'autre part, le microcode. À noter que le point de départ de cette génération est une simple description du microcode à implémenter.

Organisation du manuscrit

L'organisation de ce manuscrit suit la logique de développement de l'environnement d'exécution présenté jusqu'ici. Le chapitre 1 propose un état de l'art permettant d'avoir une approche globale des problématiques adressées à travers cette thèse. Nous définissons les bases d'une architecture hétérogène reconfigurable puis les structures de gestion de ressources applicables à ce type d'architecture. Ensuite, le panel des techniques d'isolation matérielle et logicielle est dressé. Après cela, une introduction aux architectures microcodées est proposée. Enfin, ce chapitre se conclue par un état de l'art sur génération d'architecture et de microcode.

Le chapitre 2 présente l'étude des bases de l'environnement d'exécution proposé. Les résultats démontrant la pertinence de cet environnement y sont discutés. Ils permettent d'afficher une rentabilité énergétique cohérente avec les objectifs initiaux. Un outil de développement et de visualisation des performances est également mis en avant.

Le chapitre 3 se base sur l'environnement d'exécution proposé au chapitre précédent et y insère un mécanisme de gestion de l'allocation dynamique des ressources pour également répondre à l'objectif de performance globale du système. Celui-ci apporte également des mécanismes d'isolation permettant d'atteindre les objectifs initiaux de sécurité du système.

Le chapitre 4 présente un outil de génération d'architecture *CPU* microcodée à partir d'une description du microcode à implémenter. Il en résulte un outil capable de générer des architectures *CPU softcore* RISC-V microcodées sur mesure en embarquant seulement les instructions définies en microcode.

Enfin, nous concluons ce manuscrit et proposons des axes d'évolutions des travaux présentés.

Contributions

L'ensemble des contributions de ce manuscrit forment un environnement d'exécution complet permettant un fonctionnement performant et sécurisé. Ces contributions peuvent cependant être étudiées de manière individuelle, c'est à dire :

Chapitre 2

- **Lynq** Un environnement d'exécution de base permettant l'accès aux différents périphériques de l'architecture ainsi que le contrôle des exécutions logicielles et l'utilisation des accélérateurs matériels. Pour son utilisation, côté utilisateur, une librairie modulaire est fournie.
- **Lynq webserver** Un serveur web pouvant s'interfacer avec Lynq a été développé. Il permet le codage en temps réel ainsi que le contrôle de l'exécution via l'utilisation d'un module de visualisation. Ceci afin de visualiser en temps réel les signaux internes des architectures et les résultats associés.

Chapitre 3

- **Lynq ADvanced** Se basant sur Lynq, un module de gestion et d'allocation dynamique de ressources a été développé. Il permet, selon des critères définis par l'utilisateur, d'ordonnancer des ressources logicielles et matérielles pour améliorer les performances globales de l'architecture.
- **ISOLynq** Se basant également sur Lynq, un module a été développé afin d'apporter à l'environnement d'exécution des mécanismes d'isolation et de confinement. Il permet l'utilisation de plusieurs environnements d'exécution sur une même cible tout en assurant la segmentation des applicatifs.

Chapitre 4

- **Générateur d'architecture** Un générateur d'architecture *CPU* RISC-V microcodé capable de générer, à partir d'une description du microcode en méta-langage dérivé du Lua, une architecture complète d'un *CPU* pouvant être implémenté sur *FPGA*.
- **Générateur de microcode** Un générateur de microcode qui, en fonction de signaux

de contrôle présent sur l'architecture *CPU* ciblé, génère le microcode associé aux instructions définies.

Collaborations

L'ensemble des travaux de cette thèse ont été effectués en collaboration avec le Reconfigurable Computing Group (RCG) de l'University of Massachusetts, Amherst (UMASS Amherst) et, plus particulièrement avec son Directeur, le Pr. Russell Tessier qui a contribué à ces travaux et m'a accueilli dans un environnement de travail stimulant ce qui a entraîné la naissance de la collaboration exposée ci-après.

Le serveur web et le générateur d'architecture microcodée présentés à travers le chapitre 2 et 4 sont le fruit d'une collaboration avec l'UMASS Amherst mais également, avec le Chair for embedded security (EMSEC) de Ruhr-Universität Bochum (RUB) et plus particulièrement Nils Albartus, Doctorant en sécurité des systèmes embarqués.

Publications

J. Déchelotte, R. Tessier, D. Dallet and J. Crenne, **Lynq : A Lightweight Software Layer for Rapid SoC FPGA Prototyping**, *in the Proceedings of the International Conference on Field-Programmable Logic and Application* (FPL'18), 2018, Dublin, Ireland.

M. Fyrbiak, S. Wallat, *J. Déchelotte*, N. Albartus, S. Böcker, R. Tessier and C. Paar, **On the Difficulty of FSM-based Hardware Obfuscation**, *in IACR Transactions on Cryptographic Hardware and Embedded Systems Volume 3*, (TCHES'18 Vol. 3), 2018, Amsterdam, Netherland.

1 Les System-on-Chips

« Ouvrir la route »

Devise du 31e Régiment du Génie

L'objectif de ce premier chapitre est de présenter un état de l'art sur les *SoC FPGA* qui est nécessaire à la bonne compréhension de ce manuscrit. Les problématiques pour lesquelles des solutions seront apportées dans les prochains chapitres seront également clairement établies.

À ce titre, dans la première section, nous commencerons par présenter l'architecture de base du *SoC FPGA*. Puis, la littérature relative aux environnements d'exécution qui y sont associés sera étudiée. Les mécanismes de sécurité des *SoC FPGA* et plus particulièrement les travaux relatifs à l'isolation seront ensuite exposés. Enfin, les deux dernières sections introduiront les éléments favorisant la compréhension du chapitre 4, à savoir les architectures microcodés et les travaux portant sur les outils de génération d'architectures.

1.1 Le *SoC FPGA* comme architecture hétérogène

Le 75ème anniversaire de la publication par Herman Goldstine - un des créateurs du premier ordinateur électronique - du document écrit par John von Neumann - *First Draft of a Report on the EDVAC* [14] - approche. Le monde de l'électronique analogique et numérique n'a cessé de se développer depuis. Que ce soit à travers le développement de processeur multi-cœurs ou l'essor de circuits intégrés toujours plus performants, de nouvelles architectures sont apparues pour répondre à des critères de latence, de débit ou de surface comme le montre la figure 1.1. Jusqu'au milieu des années 1980, la performance des *CPU* était principalement due à l'évolution des technologies d'implémentation. Ensuite, et ce jusqu'au début des années 2000, cette vitesse d'évolution a doublé pour atteindre 52% grâce à l'amélioration des architectures. Depuis 2003, cette augmentation a diminué à 22% du fait de certaines limitations. Elles sont, entre autres, relatives à la latence des accès mémoires, au parallélisme des instructions ainsi qu'à des limitations de dissipation énergétique.

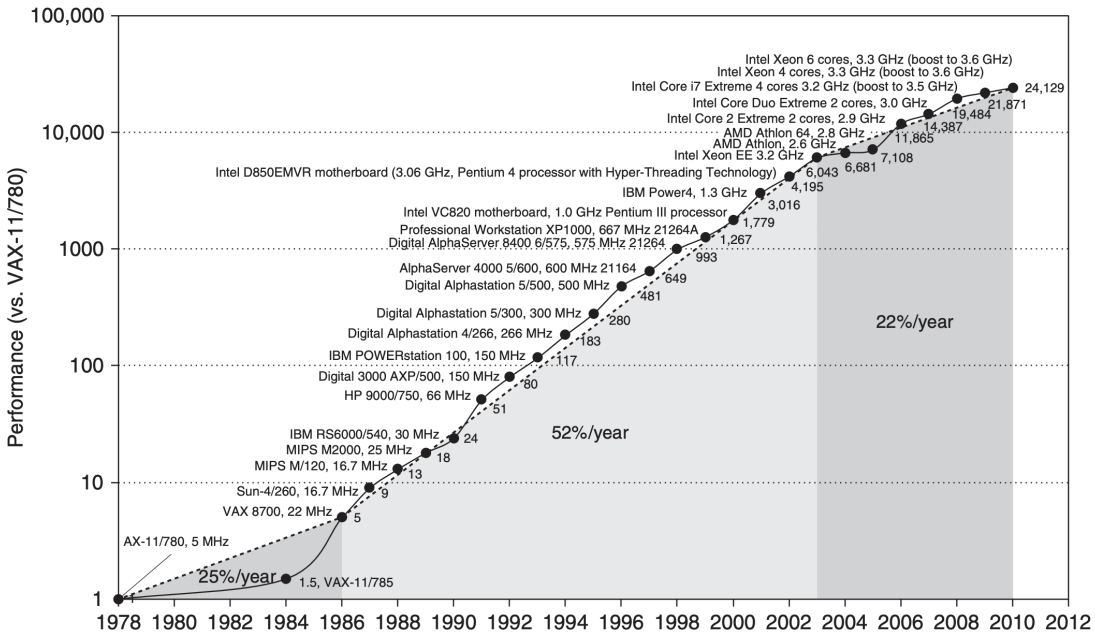


FIGURE 1.1 – Évolution de la performance des processeurs depuis la fin des années 1970 [3]

L'architecture de von Neumann est une architecture de processeur mise en évidence par le mathématicien du même nom en 1945 dans [14]. Celle-ci, présentée en figure 1.2 est composée d'une unité de contrôle ainsi que d'une Unité Arithmétique et Logique (ALU). Bien que celle-ci offre une grande flexibilité, sa particularité est qu'elle ne dispose que d'un bus partagé et d'une mémoire unique pour le stockage des instructions et des

données. Celles-ci doivent donc transiter par le même bus multiplexé pour entrer ou sortir du processeur. Ainsi le *CPU* (*Central Processing Unit*) se retrouve tôt ou tard privé d'instructions et/ou de données ce qui implique l'introduction de problématiques d'optimisation et de rendement. C'est le cas de ce qui fût appelé par la communauté scientifique le *von Neumann Bottleneck (VNB)*[15] sous l'impulsion de Backus, informaticien qui a notamment dirigé l'équipe qui inventa le langage Fortran. Il en résulte une attente qui pénalise les performances calculatoires du *CPU*. L'architecture d'Harvard a vu le jour à la même période lors du développement du premier calculateur numérique construit aux Etats-Unis. Elle diffère de l'architecture de von Neumann car la partie stockage est composée de deux mémoires distinctes (instruction et donnée) impliquant l'utilisation de deux bus séparés et améliorant la vitesse du *CPU* au détriment de la complexification de l'architecture. Des architectures sur-mesure dont l'objectif est d'être optimisé pour une application précise ont également émergé. Elles offrent les performances de latence, débit et surface recherchées, mais avec une réduction de la flexibilité si implémentées sur *ASIC* (Application Specialised Integrated Circuit). Ce type d'architecture peut également être implémenté sur cible *FPGA* (Field-Programmable Gate Array) ou à l'aide de coprocesseur dédié. En 1960, Gerald Estrin [16] proposa de combiner un *CPU* suivant l'architecture de von Neumann avec un circuit reconfigurable sur une même puce.

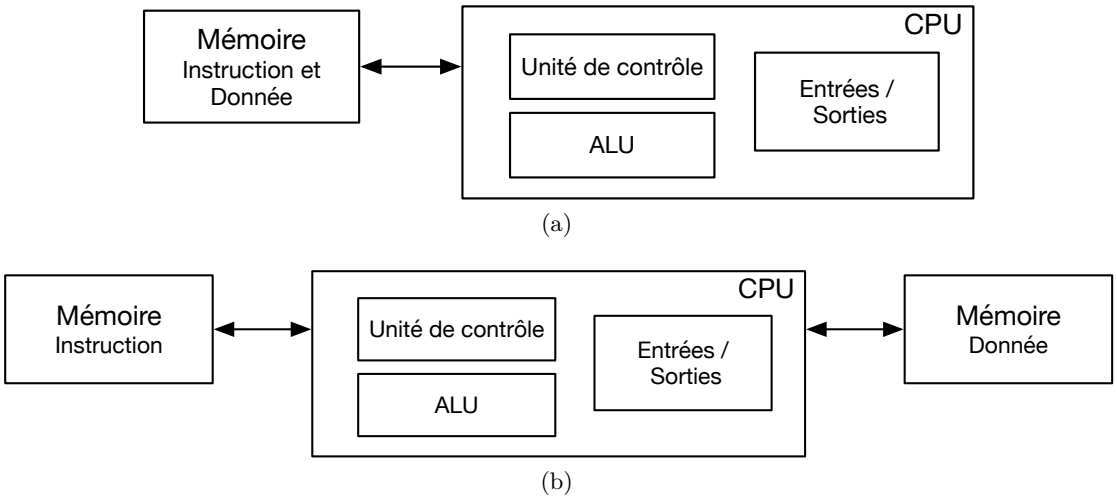


FIGURE 1.2 – Architecture de (a) von Neumann et de (b) Harvard

Des avancées comme la diminution des finesses de gravure, l'adoption du multicoeur ou encore l'augmentation des fréquences de fonctionnement (ce dernier point tend à être moins utilisé au profit de l'utilisation de plusieurs coeurs de *CPU* du fait de la proportionnalité entre fréquence et puissance consommée) ont permis d'obtenir des *CPU* plus performants, des circuits intégrés plus petits avec des performances calculatoires en constante croissance. Pour corroborer cela et étant donné les attentes croissantes des

utilisateurs qui nécessitent des performances exigeantes, les évolutions technologiques, notamment la réduction des finesses de gravures des *CPU*s, tendent également à remettre en cause la loi de Moore [17]. De plus, elles orientent les systèmes utilisant des circuits initialement conçus pour être spécialisés vers une utilisation "General Purpose". Aussi, l'utilisation des *FPGA* est privilégiée à celle des *CPU* pour certains applicatifs, notamment pour des applicatifs nécessitant l'utilisation de calcul massivement parallèle. Les premiers *CPLD* (Complex Programmable Logic Device), ancêtres des *FPGA*s, ont vu le jour en 1984 par Altera [18]. Un an après, Xilinx commercialisait le premier *FPGA* [19].

Cela a donc mené l'évolution du domaine des semi-conducteurs à voir s'imposer les *SoC FPGA*s comme une des solutions principales pour l'implémentation d'**architectures reconfigurables** dans la conception de systèmes hétérogènes complexes. Le concept d'**architecture reconfigurable** [7] est associé à l'implémentation d'architecture répondant au paradigme de "Calcul Reconfigurable (*RC* pour reconfigurable computing)" dans le domaine de la conception de systèmes numériques. À noter qu'une architecture est dite reconfigurable si au moins l'un de ses composants peut être reconfiguré. En ce qui concerne les *SoC FPGA*, cela se fait principalement via le chargement d'une nouvelle mémoire dite de configuration.

Les travaux abordés dans ce manuscrit utilisent le *SoC FPGA* comme architecture hétérogène de base. Il est important de classifier les principaux circuits logiques afin de pouvoir positionner les composants étudiés indispensables dans le cadre de nos recherches. Il s'agit du *FPGA*, du *CPU*, qui peut être multi-coeurs, et de la mémoire partagée. Des éléments complémentaires tels que les contrôleurs d'entrées / sorties peuvent s'ajouter pour compléter l'architecture.

Acronyme	Signification en Anglais	Signification en Français
ASIC	Application Specific Integrated Circuit	*Circuit Intégré propre à une application*
CPLD	Complex Programmable Logic Device	*Circuit Logique Programmable complexe*
CPU	Central Processing Unit	*Unité centrale de traitement*
DSP	Digital Signal Processor	*Processeur de signal numérique*
FPGA	Field-Programmable Gate Array	*Réseau de porte programmable*

Tableau 1.1 – Signification des acronymes pour la classification des circuits logiques

Le disque présenté dans la figure 1.3 accompagné du tableau 1.1 classifient les principaux circuits logiques numériques.

Une première partie sur la gauche du disque fait état des architectures généralistes, c'est à dire qui n'ont pas été développées pour une application précise mais pour exécuter un jeu d'instructions répondant à l'exécution d'applicatifs divers et variés. Dans cette catégorie se trouvent les *CPU*s qui sont l'implémentation d'un ou plusieurs coeurs de processeurs

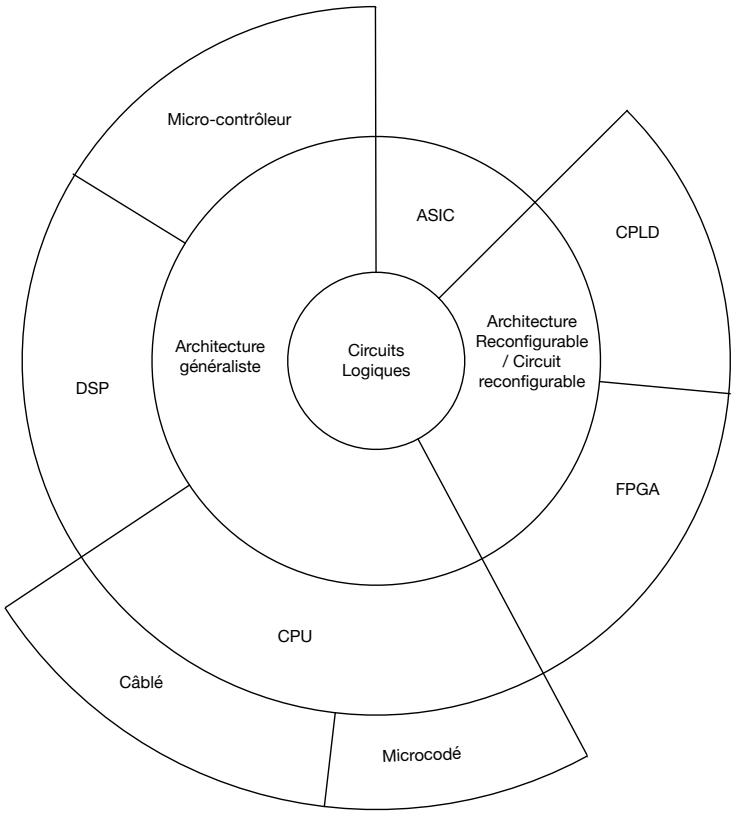


FIGURE 1.3 – Classification des circuits logiques

respectant un standard de jeu d'instructions [20, 21] et peuvent être câblés ou microcodés. Un *CPU* dit câblé met en oeuvre chaque instruction du jeu d'instructions de manière séparée. Quant au *CPU* dit microcodé, il permet d'implémenter des instructions qui utilisent du microcode depuis une description architecturale générique. Cette solution est optimale en terme de flexibilité et permet d'effectuer les mises à jour des instructions (sans reconfiguration du processeur), en modifiant seulement la mémoire contenant le microcode. Les architectures microcodées seront abordées en section 1.4. Les *DSP* sont un autre type d'architecture généraliste. Ce sont des processeurs répondant à des besoins calculatoires spécialisés pour les calculs numériques, plus particulièrement dans le traitement du signal. En effet, celui-ci utilisant de nombreux tableaux et types de données régulières ainsi que des matrices, le parallélisme trouve son sens avec l'exécution d'instruction SIMD (*Simple Instruction Multiple Data - Instruction Simple, Données Multiples*). Le dernier type d'architecture généraliste est le microcontrôleur. C'est une architecture à base de *CPU*, de mémoire ainsi que de périphériques avec leurs entrées / sorties. Il a vocation à être embarqué pour effectuer des tâches spécifiques dans des environnements contraints

en taille. Il est conçu pour être économe en énergie et embarque donc des *CPU*s à faible consommation.

Dans la seconde partie du disque en figure 1.3, les *ASIC*s sont des circuits spécialisés, développés pour une application spécifique et qui, une fois sortis de fonderie ne peuvent varier, c'est à dire dont l'architecture ne peut pas être modifiée a postériori.

La troisième et dernière partie sur le côté droit du disque schématise les architectures reconfigurables / circuits programmables qui regroupent le *CPLD* et le *FPGA*. Le *CPLD* est une architecture reconfigurable qui contient des blocs logiques reliés entre eux par un unique bloc d'interconnexion qui centralise les connexions entre ces blocs. Contrairement au *FPGA*, il contient des blocs logiques de base plus complexes et a pour avantage d'avoir un temps de propagation prédictible. Cependant, les besoins d'implémenter des circuits de plus en plus complexes ont poussé à la mise en avant du *FPGA*. Ce dernier est composé de blocs logiques entrecoupés de canaux d'interconnexion décentralisés qui nécessitent une phase de routage plus complexe. Le *FPGA* fait l'objet d'une présentation approfondie dans la prochaine section. Un trait en pointillé est positionné entre le *CPU* et le *FPGA* car les implémentations de *CPU softcore* (présenté plus loin dans ce chapitre) peuvent être microcodées. Une modification du microcode permet notamment dans les nouveaux jeux d'instructions comme RISC-V [22], de spécialiser pour les besoins d'une application le *CPU* qui de base était généraliste. Une contribution sera développée dans le chapitre 4 à ce propos.

1.1.1 De l'architecture reconfigurable *FPGA*..

Un *FPGA* est un circuit intégré reprogrammable composé d'un réseau de ressources logiques programmables entrecoupées de canaux d'interconnexion. Il permet la réalisation d'architectures numériques complexes. La logique et les canaux d'interconnexion sont (re)programmables. Entièrement reconfigurables, ils offrent une grande souplesse algorithmique et permettent de tirer parti d'un parallélisme dit massif. Les *FPGA*s sont composés de deux couches, une première de logique et une seconde de mémoire permettant de configurer la couche logique infra. L'architecture du circuit et/ou l'utilisation des blocs IP pré-existants sont décrits par l'utilisateur.

FIGURE 1.4 – Répartition du marché des dispositifs logiques reprogrammables en fonction de leurs revenus selon le rapport [4]

Le marché du *FPGA* est dominé par deux grands du secteur de l'électronique que sont Xilinx [23] et intelFPGA [24] (suite à l'acquisition d'Altera par Intel [25]) qui se partageaient en 2015 environ 85% du marché [4]. Le reste de marché est essentiellement pourvu par le constructeur Lattice [26] avec 10% ou par d'autres acteurs comme Microsemi [27], QuickLogic [28]. Le montant total du marché représentait 5,83 milliards de dollars en 2017. La projection est favorable puisqu'elle annonce un marché à 9,5 milliards de dollars en 2023, soit une évolution en croissance de 8,5% par an.

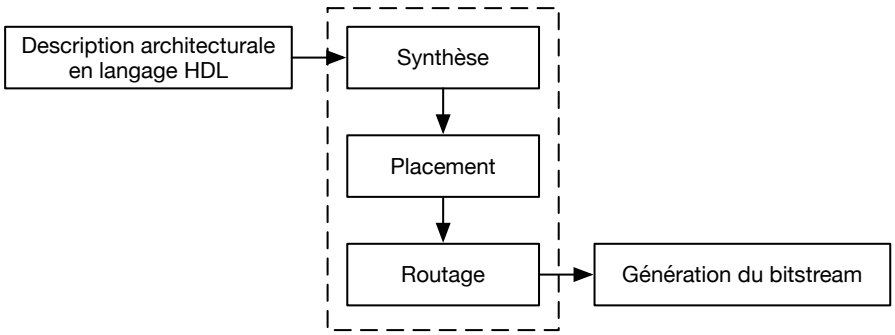


FIGURE 1.5 – Chaîne de génération d'une architecture de sa description à la génération du bitstream

La génération d'une architecture sur *FPGA* s'effectue en plusieurs étapes qui vont de sa description jusqu'à la génération du bitstream de configuration comme présenté à travers la figure 1.5. Il s'agit d'un fichier binaire qui contient les informations de configuration d'un *FPGA*. Un langage de description architecturale (Hardware Description Language, HDL) est tout d'abord utilisé. D'un point de vue historique, le *VHSIC* Hardware Description Language (*VHDL*) [29] et le Verilog [30] aident à décrire l'architecture matérielle souhaitée et ont pour intérêt de pouvoir détailler jusqu'à la porte logique l'architecture à réaliser.

La figure 1.6 présente le diagramme en Y de Gajski et Kuhn [5]. Elle classifie dans les domaines fonctionnel, structurel et physique, tous les niveaux de granularité existants dans la définition d'un système.

Les langages HDL permettent d'effectuer une description de l'architecture depuis le niveau logique jusqu'au niveau architectural. D'autres langages dérivés existent. C'est par exemple le cas du SystemVerilog [31] qui est une extension du Verilog et qui permet une description au niveau fonctionnel. Il s'en suit la phase de synthèse du code *HDL* qui précède les phases de placement et de routage. Ces trois étapes sont généralement exécutées en utilisant la chaîne d'outils propriétaire du constructeur de *FPGA* [32, 33].

Cependant, l'implication de la communauté de l'open source est croissante dans le domaine de la conception d'outils pour le développement d'architecture matérielle.

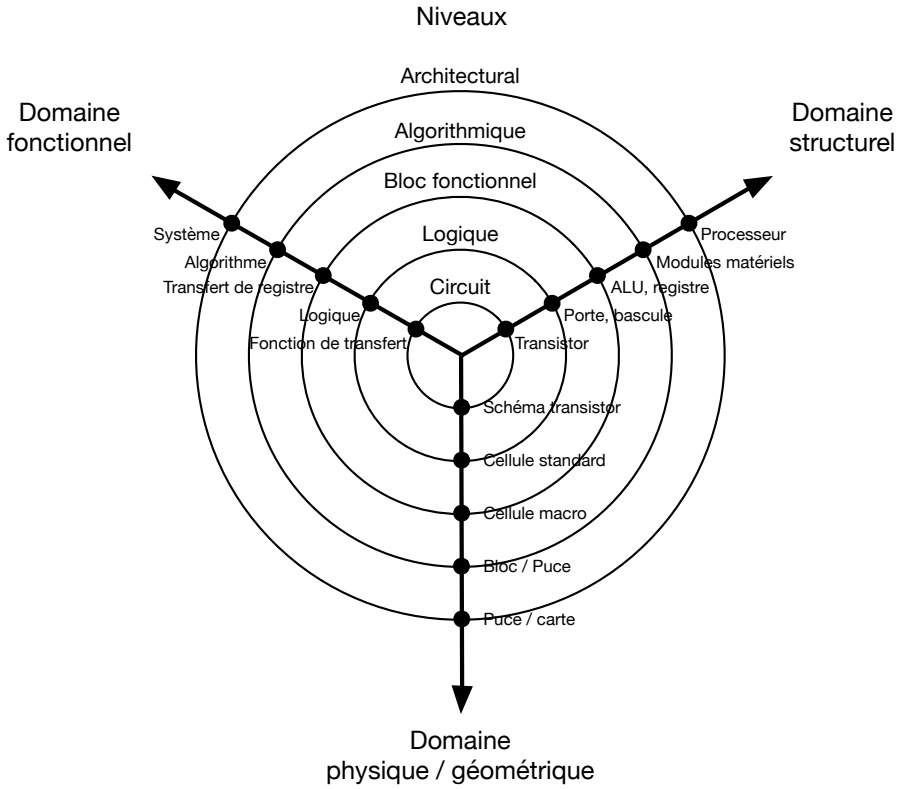


FIGURE 1.6 – Diagramme en Y de Gajski et Kuhn [5]

De ce fait, des outils comme Yosys [34] pour la synthèse ou nextPnR pour le placement routage gagnent peu à peu l'adhésion [35] et proposent des outils autres que ceux du constructeur pour une meilleure maîtrise de la conception du système. Enfin, la dernière étape consiste à générer le bitstream. À l'heure actuelle, à part pour le vendeur Lattice dont les bitstreams peuvent être générés à partir d'outils open source [36], ils doivent tous être générés à partir de la chaîne de développement du constructeur du *FPGA* cible.

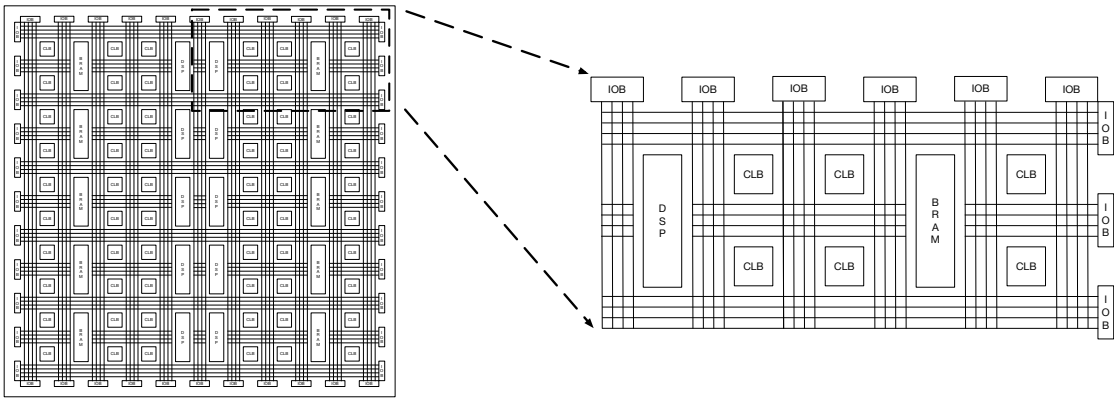


FIGURE 1.7 – Matrice d'un *FPGA*

Acronyme	Signification en anglais	Signification en Français
BLE	Basic Logic Element	*Elément Logique de Base*
BRAM	Block Random-Access Memory	*Bloc de mémoire à accès non séquenciel*
CLB	Configurable Logic Block	*Bloc de logique configurable*
DFF	Data Flip-Flop	*Bascule D*
DSP	Digital Signal Processing (Element)	*Unité de calcul numérique spécialisé*
IOB	Input / Output Block	*Bloc d'entrée / sortie*
LUT	Look-Up Table	*Table de correspondance*

Tableau 1.2 – Signification des acronymes des blocs élémentaires composant un *FPGA*

La capacité à décrire précisément l'implémentation souhaitée repose sur l'architecture interne du *FPGA* comme présentée figure 1.7. Celle-ci sert de base pour comprendre le très haut niveau de reconfigurabilité d'un *FPGA*.

Les *FPGA*s ont une structure matricielle et ont pour élément de base le *CLB*. Un *CLB* est généralement composé de *BLE*. Un *BLE* contient des éléments de logiques combinatoires, des *LUT*s, des *DFF*s et des mécanismes de diffusion de retenu. Les dénominations de *BLE* et *CLB* peuvent varier d'un constructeur à un autre. Ces blocs, constitués de ressources logiques sont entrelacés avec un réseau de canaux d'interconnexions. Le fait que les *CLB*s et les canaux d'interconnexions soient reconfigurable font du *FPGA* un circuit entièrement reconfigurable. De plus, les architectures reconfigurables sont aujourd'hui capables de fournir des performances d'un ordre de grandeur supérieur aux solutions logicielles exécutées sur processeur [37] dans le cadre d'une utilisation pour systèmes embarqués. En ce sens, elles s'imposent comme une architecture préférentielle dans de nombreux domaines d'applications. Des applications tirant parties de celle-ci sont exposées dans [7] et [38]. Le tableau 1.3 en reprend des exemples.

Finance	Évaluation du risque d'un crédit [39] Évaluation de cotation d'une valeur boursière [40]
Traitement du signal	Reconnaissance faciale [41] Détection d'arrière plan pour la reconnaissance d'objet [42]
Sécurité	Cryptographie [43]

Tableau 1.3 – Exemple d'application *FPGA* classés par catégories selon [7]

Les *FPGA*s embarquent aussi un ensemble de composants hétérogènes pour améliorer le niveau de performance calculatoire et de complexité des circuits qu'il est possible d'implémenter. Ceux-ci sont intégrés et irrigués par des canaux d'interconnexion. Les principaux composants hétérogènes que l'on retrouve sont les *BRAM*s et les blocs *DSP*s. Les *BRAM*s sont des mémoires qui fournissent des ressources de stockage au plus proche

des éléments logiques afin de réduire les latences en lecture et écriture (implicitement d'accélérer les vitesses de traitement des accélérateurs matériels). Une description autre que celle présentée dans le tableau 1.1 de *DSP* sera utilisée dans ce document. Il s'agit de Digital Signal Processing. Elle désigne un bloc élémentaire dans un *FPGA* spécialisé dans le calcul numérique. Les blocs *DSP*s sont optimisés, ils peuvent être préférés aux ressources logiques lorsque l'architecture nécessite l'utilisation d'opérateurs arithmétiques complexes. De plus, et du fait de leur intégration et des fréquences de fonctionnement supérieures qu'ils peuvent atteindre en comparaison aux autres ressources logiques, ils peuvent obtenir des débits plus importants. Ainsi, la figure 1.7 illustre le fait que les *IOB*s sont situées sur les extrémités Nord, Est, Sud et Ouest du *FPGA*. Ceux-ci servent d'interface vers l'architecture de niveau supérieur au *FPGA* à savoir, la carte sur laquelle le *FPGA* est par exemple utilisé.

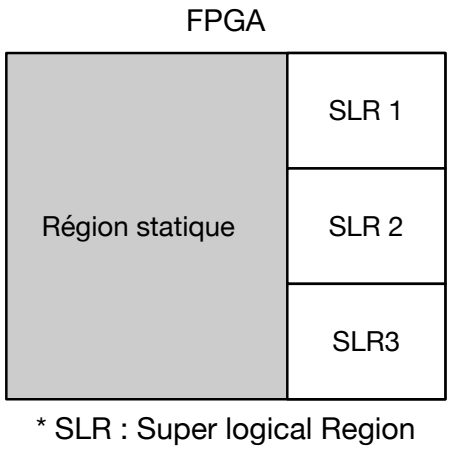


FIGURE 1.8 – Floorplan simplifié d'un composant reconfigurable dynamiquement

Comme exposé précédemment, la structure très régulière des *FPGA* et leurs conceptions architecturales en couches logiques et de configuration [44] permettent une reconfiguration de la couche logique pour changer l'architecture implémentée via le changement de la mémoire de configuration. Avant l'utilisation du *FPGA*, la mémoire de configuration afférente au circuit implémenté est chargée. Celui-ci peut être reconfiguré avant chaque fonctionnement. Ce type de reconfiguration est dit "statique" du fait qu'il ne soit pas en fonctionnement lors de la reconfiguration.

Les *FPGA* sont capables d'implémenter plusieurs accélérateurs matériel sur la même cible. Certains peuvent être coûteux en ressources. Une solution proposant une approche modulaire consiste à mutualiser des sous-parties du *FPGA*. Malgré une partie commune à chaque applicatif devant fonctionner sur le *FPGA*, un module complémentaire est implémenté au travers des SLRs (pour Super Logical Region) de Xilinx comme étant

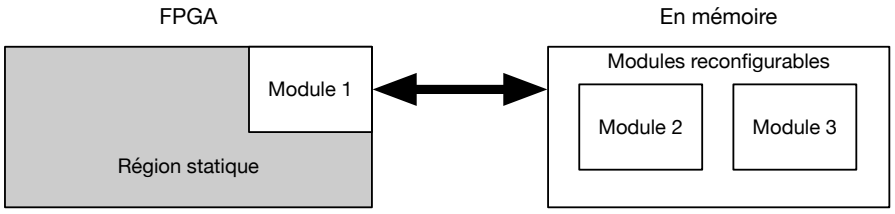


FIGURE 1.9 – Principe de la Reconfiguration Dynamique Partielle

une partie de la matrice *FPGA*. La figure 1.8 expose cela. De part la mutualisation de la partie commune cette approche permet l'économie de ressource logique. Cependant, il n'est pas possible de diminuer la taille du *FPGA* ce qui entraine une sous-utilisation des ressources. Cela sous-entend que tous les modules complémentaires ne sont pas utilisés à chaque instant pendant la phase d'exécution. Une solution, apparue avec l'évolution des technologies *FPGA* [45], consiste à introduire de la reconfiguration dynamique partielle [46, 47] comme exposé figure 1.9. Cette méthode permet de reconfigurer dynamiquement une partie du *FPGA* en basculant d'un module vers un autre lorsque les modules sont utilisés dans le temps de manière exclusive. La reconfiguration dynamique partielle introduit deux types de zone dans le *FPGA* :

- Une première, dite "statique", qui après le démarrage du *FPGA* ne varie plus dans le temps.
- Une seconde, dite "dynamique", qui est en mesure d'être reconfigurée durant le fonctionnement de la zone statique.

Avantages	
CONTINUITÉ DU SERVICE	Durant la phase de RDP, le *FPGA* peut continuer son fonctionnement sans interruption, seule la partie en cours de reconfiguration ne peut être utilisée.
MUTUALISATION DES RESSOURCES DISPONIBLES	La RDP permet la réutilisation d'une même région du *FPGA* pour l'exécution de plusieurs applicatifs lorsqu'ils n'ont pas une exécution parallèle.
Inconvénients	
STOCKAGE DES BITSTREAMS	Le stockage des bitstreams utilisés pour la RDP consomme de l'espace mémoire et son emplacement est directement lié à la latence d'accès.
TEMPS DE RECONFIGURATION	Il y a proportionnalité entre la taille du bitstream à reconfigurer et son temps de reconfiguration. Ce temps est contraint par la fréquence de l'ICAP.

TABLEAU 1.4 – Principaux avantages et inconvénients de la Reconfiguration Dynamique Partielle

La reconfiguration dynamique partielle s'effectue via l'utilisation d'un port permettant l'accès à la mémoire de configuration du *FPGA* : ICAP pour Internal Control Access Port (Port de contrôle d'accès Interne) chez Xilinx et PR IP Bloc pour Partial Reconfiguration IP bloc chez intelFPGA. Celui-ci prend le bitstream à configurer en entrée et est en charge du chargement de la mémoire de configuration du *FPGA*. Le tableau 1.4 précise les avantages et inconvénients relatifs à l'utilisation de la Reconfiguration Dynamique Partielle.

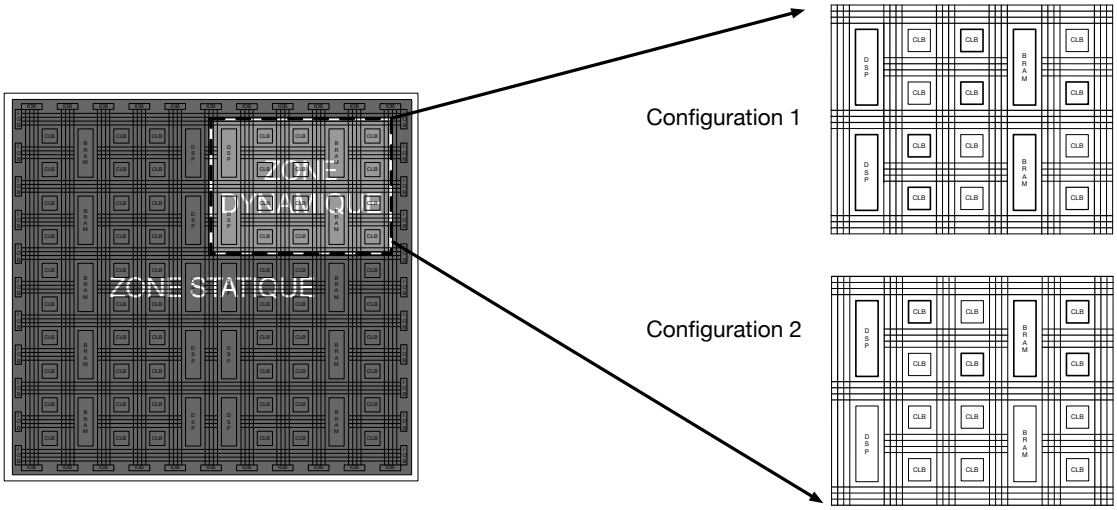


FIGURE 1.10 – Matrice de *FPGA* avec zones reconfigurables

Conceptuellement, la reconfiguration dynamique partielle permet le partage des ressources matérielles conformément à la temporalité imposée par les besoins de l'applicatif. Durant la phase de conception de l'architecture reconfigurable, il est défini une première zone du *FPGA* qui a vocation à être mutualisée pour l'ensemble du/des applicatifs présent sur le *FPGA*. D'autres zones dynamiques sont identifiées et ont vocation à être reconfigurées durant l'exécution selon les besoins de l'applicatif. La figure 1.10 expose le principe de la reconfiguration dynamique partielle appliquée à l'architecture présentée en figure 1.7. On y observe deux configurations (routage et utilisation) de ressources possibles pour une même région du *FPGA*.

1.1.2 .. au *SoC FPGA*

Les architectures combinant un *CPU* et un *FPGA* ont la possibilité d'utiliser le *FPGA* pour implémenter des accélérateurs matériels dans le cadre de l'exécution d'un programme sur le *CPU*. Pour réduire les temps de communication entre le *FPGA* et le *CPU* et, ainsi diminuer les temps de latence des calculs, les constructeurs de *FPGA* se sont concentrés sur des solutions proposant un *FPGA* et un *SoC* sur le même substrat. Les

deux principaux vendeurs proposent leurs solutions, la série Zynq pour Xilinx [48], les séries Cyclone V [49], Arria 10 [50] et, plus récemment Agilex [51] pour intel*FPGA*. La figure 1.11 décrit l'architecture en bloc d'un *SoC FPGA*. On y retrouve donc le *SoC* ainsi que le *FPGA*. Ceux-ci sont reliés par un bus connectant une mémoire partagée.

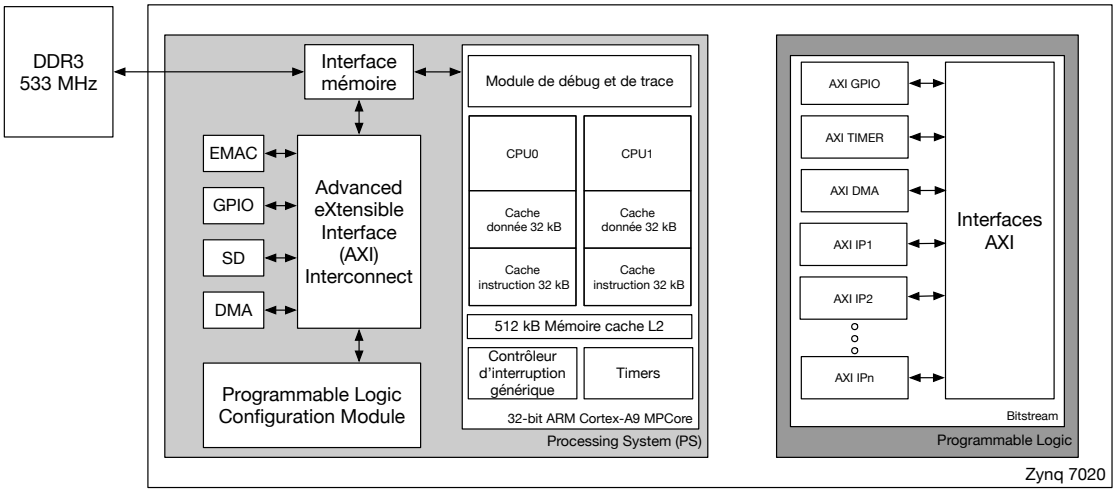


FIGURE 1.11 – Exemple de *SoC FPGA* : le Zynq-7020 de Xilinx

Ce fonctionnement ouvre la possibilité de penser que, du fait de leur aspect généraliste, les *CPU* pourraient être utilisés comme organe de contrôle malgré les versions de *SoC FPGA* qui existent avec des *CPU*s orientés vers le calcul. Le Zynq UltraScale + RFSoC de Xilinx [52] en est un exemple. Ils peuvent embarquer un système d'exploitation et l'ensemble des drivers nécessaires tandis que les *FPGA*, quant à eux, pourraient être exclusivement utilisés comme organe de calcul.
Cependant, l'intérêt croissant pour ce type de circuit logique programmable [53] a poussé l'industrie et la recherche à fournir des efforts pour apporter des solutions permettant de tirer parti des capacités des *FPGA*, et ceci même lorsqu'ils ne sont pas couplés à des *CPU*s au sein d'un même *SoC*. Un exemple concret consiste à développer des *CPU*s *softcore* pour effectuer des calculs. C'est un *CPU* qui, à partir de sa description dans un langage *HDL*, peut être implémenté sur un *FPGA*. À l'inverse d'un *CPU* durci dans le silicium qui prendra l'appelation de *CPU hardcore*. La fréquence du *CPU* sera plus élevé dans un *hardcore* car sa puce est spécifique à l'architecture du processeur contrairement au *softcore* qui est une configuration particulière de la couche logique du *FPGA* et répond donc aux limites de fréquence de celui-ci.

Le contrôle de la gestion des ressources et de l'exécution était initialement géré par des contrôleurs, implémentés dans le *FPGA* , qui prenaient généralement la forme d'une machine à état, que ce soit de Mealy ou de Moore. L'implémentation de *CPU softcore* a ouvert la route au support de systèmes d'exploitation sur *FPGA*. À première vue,

ceci pourrait tendre à remettre en cause l'intérêt des *SoC FPGA*s embarquant des *CPU*s du fait de l'autonomie acquise par les *FPGA*s qui sont maintenant capables d'exécuter du code généraliste. Cependant, l'intérêt des *SoC FPGA*s est maintenu car les *CPU softcore* n'égalent pas les *CPU hardcore*. D'autant plus qu'il est possible d'aborder l'utilisation des *CPU softcore* sous un autre angle qui est celui du *CPU* de contrôle, embarquant simplement du code logiciel ou un système d'exploitation léger pour le contrôle de l'exécution et des ressources. L'objectif principal est de bénéficier de toute la puissance de calcul du *CPU hardcore* présent sur le *SoC FPGA* pour du calcul généraliste. En effet, les mécanismes de reconfiguration dynamique partielle des *FPGA*s sont intéressants en terme de gain de place. Du fait de la reconfigurabilité des *FPGA*s, il est possible d'y exécuter un large spectre d'application. Toutefois, le *FPGA* seul, même s'il embarque un *CPU softcore*, ne peut pas contenir tout l'écosystème de driver et de contrôle qu'est capable d'apporter l'exécution de code sur un *CPU hardcore*, que ce soit en standalone ou avec un système d'exploitation. **En cela, les optimisations architecturales des *CPU*s couplées aux technologies *FPGA* offrent une alternative intéressante.**

L'utilisation de *SoC FPGA* nécessite des outils capables de tirer profit des capacités propres à ce type d'architectures. Les flots de conception et d'utilisation ont donc évolué de pair avec les *SoC FPGA*s. Ceux-ci sont complexes et nécessitent des savoirs propres aux technologies matérielles et logicielles. Le spectre des utilisateurs est alors contraint par ce besoin de savoir complexe. De plus, le manque de plateforme de développement standardisée pour les *SoC FPGA* ne favorisant pas l'apprentissage, l'évolution de l'acculturation à ce type d'architecture prend du temps.
Pour accélérer cela et diminuer les savoirs nécessaires à la conception et l'utilisation des *SoC FPGA*, Xilinx a développé le projet PYNQ [54]. Il s'agit d'une carte de développement à base de *SoC FPGA* Zynq 7-series, d'un framework constitué d'un ensemble de bibliothèques, de pilotes logiciels écrits en Python et de l'image d'un système d'exploitation Linux : Ubuntu. Ce projet est basé sur des tutoriels, de la documentation ainsi que sur une couche logicielle permettant d'abstraire le bas niveau matériel afin de le rendre plus accessible via l'utilisation de librairie Python. Le but est d'ouvrir l'accès au développement sur *SoC FPGA* aux néophytes de la programmation sur cible matérielle. Les tutoriaux prennent la forme de programme Python directement exécutable via un serveur web local auquel la carte est connectée. Cette plateforme a rapidement été adoptée par le milieu de l'enseignement [55]. Des améliorations ont été portées à ce projet. Il s'agit, par exemple, des solutions permettant l'utilisation de la reconfiguration dynamique partielle afin de modifier la conception de base [56].

Le fonctionnement du système d'exploitation Ubuntu sur PYNQ permet l'accès à de nombreux services et sert de base à l'ajout de bibliothèques utilisées pour le développement

et l'exécution d'applicatifs.
L'accès à ces deux ressources hétérogènes depuis une même interface nécessite une connaissance précise de l'environnement, ce qui réduit la population en mesure de développer sur ces cibles. **L'objet du chapitre 2 est donc de proposer une solution capable de rendre accessible l'utilisation de *SoC FPGA* à des non spécialistes.** Plus particulièrement, il s'agit de tirer partie d'un langage de haut niveau pour le développement et le contrôle du *FPGA*. De manière plus générale, cette solution doit permettre la gestion de l'architecture hétérogène reconfigurable en préservant, ou en limitant fortement a minima, la perte de performances calculatoire de celle-ci.

Le développement de PYNQ ainsi que d'autres solutions citées précédement démontrent un gain d'intérêt pour les *SoC FPGA*, plus particulièrement des efforts ont été fournis pour développer des environnements d'exécution permettant de tirer partie de leurs capacités et notamment de leurs caractéres reconfigurables. La section suivante en fait état.

1.2 Les environnements d'exécution pour l'implémentation rapide

Au fil des ans, l'évolution des architectures reconfigurables à base de *FPGA* et de processeurs a été conditionnée à l'évolution des outils et technologies d'implémentation ainsi qu'à leur facilité d'utilisation. Pour tirer profit des performances offertes par les *SoC FPGA*, deux approches sont possibles. Une première, dite généraliste, consiste en l'utilisation d'un système d'exploitation fonctionnant sur un coeur de processeur du *SoC FPGA* cible. Cette solution apporte des facilités de développement de l'applicatif. L'adoption d'un mode de fonctionnement en standalone, c'est à dire sans système d'exploitation, constitue la seconde approche. Celle-ci permet d'implémenter sur la cible uniquement le matériel nécessaire à la bonne exécution de l'applicatif. Malgré le bénéfice que peut apporter cette dernière solution pour les systèmes embarqués, sa complexité de développement pousse la communauté à poursuivre ses efforts pour : 1) améliorer la première approche, à savoir rendre plus léger les systèmes d'exploitation et 2) proposer des couches d'abstraction du matériel (*HAL* pour Hardware Abstraction Layer) afin de simplifier l'utilisation des architectures reconfigurables.

Les performances de tels *FPGA* font d'eux des cibles privilégiées pour les applications nécessitant un haut niveau de parallélisme. Leurs reconfigurabilités permettent également l'exécution de différents applicatifs en parallèle ou de manière concurrente. Cette gestion de la distribution des applications dite reconfigurables sur la matrice *FPGA* nécessite une

approche plus avancée que pour des applications dites statiques. Généralement, ceci est géré par l'ajout de briques logicielles au système d'exploitation exécuté sur le *CPU*, ou par l'utilisation d'un bloc matériel dédié. Les services de gestion du matériel reconfigurable et systèmes d'exploitation se sont développés comme en témoigne le tableau 1.5 avec la croissance de leur intérêt.

Ref	Auteurs	Année	Compatible *SoC FPGA*	Permet la RDP	Basé sur un système d'exploitation
[2]	Ismail *et al.*	2011	•	•	•
[57]	Burns *et al.*	1997	•	○	○
[58]	Brebner	1996	○	○	•
[59]	Danne *et al.*	2006	•	○	○
[60]	Gotz *et al.*	2006	○	•	•
[61]	Walder *et al.*	2004	•	•	•
[62]	Garcia *et al.*	2007	•	NA	•
[63]	Fu *et al.*	2005	•	•	•
[64]	Steiger *et al.*	2004	•	•	•
[65]	Fahmy *et al.*	2009	○	•	•
[66]	Brodersen *et al.*	2006	•	•	•
[67]	Agne *et al.*	2014	•	○	○
[68]	Pellizzoni *et al.*	2007	•	•	○
[69]	Diguet *et al.*	2011	•	•	•

Tableau 1.5 – Synthèse des solutions de gestion du matériel reconfigurable

Le support d'un système d'exploitation pour exécuter dynamiquement des parties d'une application sur le *FPGA* est apparu plus tard [63]. Ceci s'explique par le fait que le *FPGA* était considéré comme un coprocesseur dans une architecture reconfigurable à base de *CPU*. L'environnement d'exécution logiciel gérait donc sa configuration. Ce type d'allocation de ressource dynamique est difficile pour les systèmes embarqués qui ont des contraintes temps réel strictes [64]. La planification des tâches peut être envisagée grâce à un mécanisme d'affectation de tâches matérielles ou logicielles, celle-ci étant elle même dépendante des ressources disponibles et des contraintes en latence du système [65].
Des systèmes d'exploitation comme Linux ont été améliorés pour permettre à l'ordonnanceur de disposer de nouveaux modules matériels basés sur *FPGA* [2] et avoir accès au système de fichiers Linux [66]. Récemment, les systèmes d'exploitation pour les plateformes comprennant un ou plusieurs microprocesseurs et des ressources reconfigurables ont gagné en popularité [70]. Des systèmes d'exploitation dédiés comme ReconOS [67] ont ainsi vu le jour. Ils permettent au matériel et au logiciel d'intéragir en utilisant des primitives de synchronisation initialement dédiées au logiciel comme les sémaphores et mutex. Enfin, [68] a développé une solution d'allocation de ressource dynamique considérant un système embarqué à contrainte temps réel.

Compte tenu de ces éléments, notre comparaison se portera sur deux solutions développées dans les parties qui suivent. Il s'agit de BORPH [1] et FUSE [2].

1.2.1 BORPH [1]

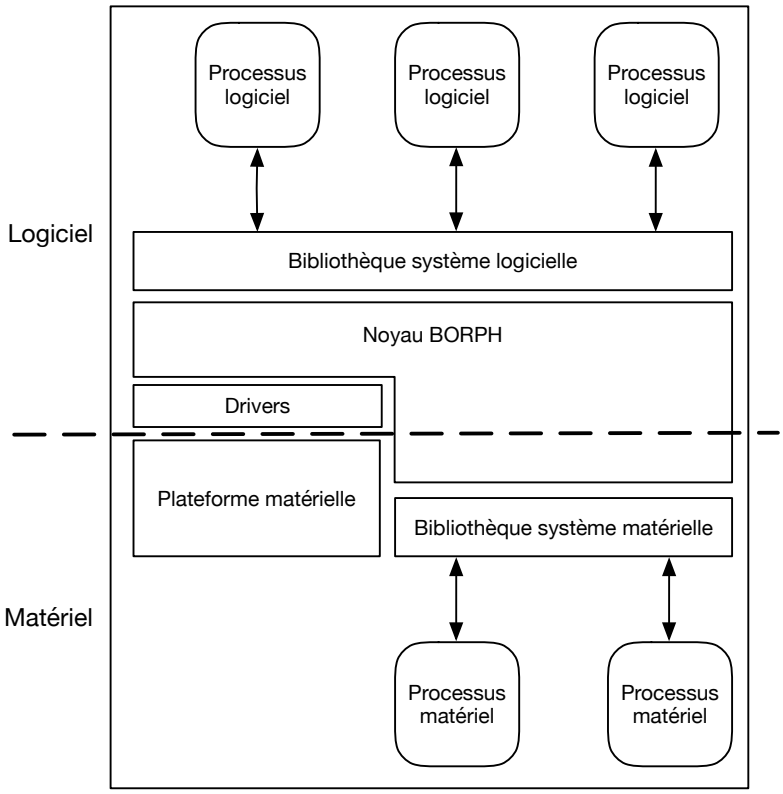


FIGURE 1.12 – Positionnement du noyau BORPH

BORPH pour Berkeley Operating System for ReProgrammable Hardware est un système d'exploitation spécifiquement développé pour les architectures reconfigurables à base de *FPGA* [71]. BORPH étend la sémantique UNIX aux architectures reconfigurables [66]. Il fournit le noyau nécessaire à l'exécution d'applicatifs sur le *FPGA* comme le montre la figure 1.12. Cette gestion du *FPGA* peut être divisée en deux parties dont la première est la gestion de la matrice *FPGA*. L'intérêt est qu'elle puisse être administrée comme une autre ressource telle que la mémoire ou le temps alloué au processeur par applicatif. La seconde est l'abstraction de détails bas niveau afin que l'utilisateur soit en mesure de se concentrer sur le développement de son application.
Le choix de la sémantique UNIX revêt deux aspects : historique et technique. Historique car UNIX est connu des ingénieurs logiciels et matériels. En ce sens, il simplifie le processus de développement et d'utilisation des architectures reconfigurables par des personnes ne possédant pas de connaissances avancées dans ce domaine. L'aspect technique se traduit par le fait que certaines sémantiques, telles que la gestion des pipes ou des flux de données sont très proches de celle utilisée dans le traitement du signal. C'est par exemple le cas pour l'utilisation des *DSP*. Enfin, le système d'exploitation Linux et ses implémentations UNIX ont été très largement étudiés par la communauté du logiciel libre et portés sur

des cibles *FPGA*. Cela facilite le déploiement de *BORPH* sur des architectures déjà existantes.

BORPH offre donc une solution qui modifie le noyau Linux et l'étend avec une interface matérielle. Il fournit les mécanismes de communication UNIX conventionnels. Le matériel est abstrait en processus UNIX et peut accéder au service offert par le système d'exploitation en utilisant des communications par FIFO. BORPH permet le choix de la ressource d'exécution d'une tâche sur processeur ou dans une zone reconfigurable. Cependant, il ne fournit pas de mécanisme pour permuter une tâche en cours d'exécution sur le matériel.

1.2.2 FUSE [2]

Dans le même esprit que BORPH, FUSE pour Front-end USEr framework abstrait les couches matérielles inhérentes aux architectures reconfigurables pour les concepteurs de logiciels et d'applicatifs. Le but est de permettre aux développeurs d'accélérateurs matériels de créer ou mettre à jour des *IP*s sans que le concepteur logiciel ne soit impacté. Le dispositif virtualise donc les accélérateurs matériels en tâches matérielles, de la même manière que ReconOS [67] ou Hthreads [72] le font dans un contexte multithread.

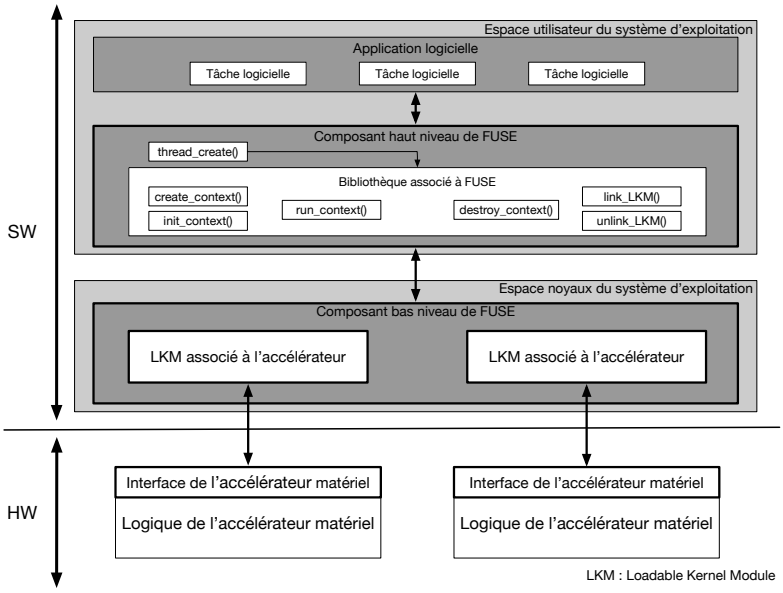


FIGURE 1.13 – Architecture de FUSE

Cette solution, présentée figure 1.13 adopte une structure modulaire. Elle offre ainsi des interfaces de contrôle et de données personnalisables entre les accélérateurs matériels et leurs drivers dans le noyau via l'utilisation de Loadable Kernel Module (*LKM*). Ces *LKM* font office de couches d'abstractions logicielles de l'accélérateur matériel pour un

accès à celui-ci. Ils utilisent des fonctions logicielles simples.
Cette approche en couche modulaire permet une séparation nette de la couche logicielle et applicative, ainsi que l'architecture reconfigurable ciblée pour les développeurs. Ainsi, la modification d'un accélérateur implique seulement la mise à jour de son *LKM* et ne nécessite pas de modification de l'applicatif. Ce dernier point en fait un avantage considérable pour son adoption pour un applicatif dont les accélérateurs matériels nécessitent des mises à jours régulières.

FUSE offre donc une solution basée sur le système d'exploitation Petalinux [73] qui est complété par l'apport d'une bibliothèque propre à FUSE. L'abstraction de la configuration architecturale sous-jacente est fournie par l'application de *LKM* associée à chaque IP. L'intérêt de FUSE réside dans le fait que les accélérateurs au niveau matériel ne nécessitent pas d'interface uniformisée.

1.2.3 De la gestion des ressources à celle des environnements d'exécution

La problématique relative à la gestion des ressources est également applicable à la gestion des environnements d'exécution sur une architecture hétérogène de type *SoC FPGA*. Le développement de solutions permettant cela est, à l'heure actuelle, essentiellement basée sur l'utilisation d'un système d'exploitation. **C'est pourquoi le chapitre 3 présentera une solution permettant l'exécution d'applicatifs en optimisant le multiplexage temporel pour améliorer la rentabilité des ressources disponibles en standalone.** En comparaison aux élèments présentés dans le tableau 1.6, cette solution proposera un gestionnaire d'allocation de ressources dynamique doté de capacité de préemption et qui permet le passage efficace de processus du logiciel vers le matériel, et inversement.

	Système d'exploitation	Préemption	Moyen de synchronisation et communication	Méthode d'abstraction	Interconnection
BORPH	•	○	Passage de message	Processus Unix	Point à point
FUSE	•	○	Mécanisme intégré au driver	Pthread	Bus commun

Tableau 1.6 – Synthèse des solutions BORPH et FUSE

Une architecture hétérogène reconfigurable permet l'exécution de multiples applicatifs que ce soit de manière concurrente ou parallèle. Ceci permet à plusieurs applicatifs d'être exécutés sur une même cible indépendament du niveau de criticité de chacune d'entre elle. Cela engendre des problématiques relatives à leur sécurisation. En effet, que ce soit de part la sensibilité des informations traité ou l'impact d'une défaillance sur le système, certains applicatifs nécessitent une protection. Il est donc nécessaire de fournir des primitives permettant un environnement d'exécution sécurisé. Cet intérêt est

démontré par la mise à disposition par l'*Agence Nationale de la Sécurité des Systèmes d'Information* (*ANSSI*) de fiches pédagogiques [74] visant à sensibiliser, autant le grand public que les industriels, des menaces qui émergent dans ce domaine.
Dans la prochaine section seront présentés les éléments liés à la sécurisation des environnements d'exécution. Une attention particulière sera portée sur les mécanismes d'isolation, qu'ils soient matériels ou logiciels.

1.3 La sécurisation des environnements d'exécution

Les domaines de la sécurité de l'information se concentrent principalement sur la sécurité des systèmes tels que la sécurité réseau, la sécurité des systèmes d'exploitation et la sécurité des bases de données. Cependant, avec l'utilisation généralisée des circuits intégrés, les problèmes de sécurité du matériel représentent des menaces pour la vie privée et la sécurité des personnes, comme pour l'économie nationale et la sécurité de la défense nationale. L'étude [75] expose qu'en 2007, le système de défense radar syrien n'a pas été en mesure d'identifier et d'alerter un missile guidé israélien. D'un point de vue scientifique, il en résulte la mise en cause d'une puce du commerce utilisée dans le système radar défectueux dans laquelle il avait été introduit une porte d'entrée dérobée qui a permis l'insertion d'un trojan durant les phases de développement. Pendant son utilisation et après réception du code activant le malfonctionnement, celui-ci n'a pas fonctionné. [76] expose également une porte dérobée délibérément insérée dans une puce *FPGA* Microsemi permettant l'extraction d'information sensible ainsi que l'ajout de fonctionnalités supplémentaires.

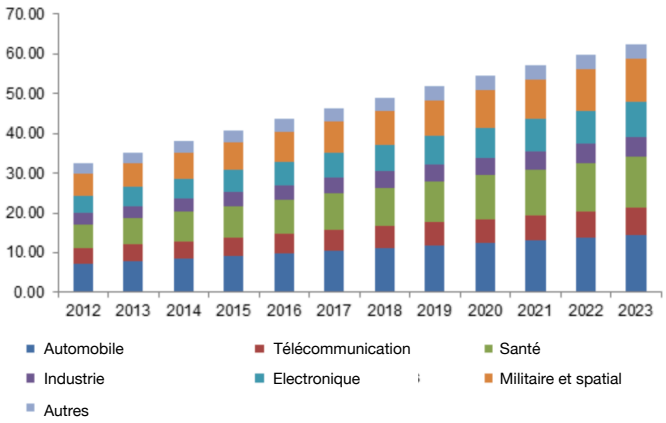


FIGURE 1.14 – Étude de 2016 sur la taille du marché européen des systèmes embarqués en milliard de dollars selon [6]

La sécurisation des environnements d'exécution, du plus bas niveau matériel jusqu'à

l'applicatif, est essentielle du fait de l'évolution des utilisations faites des *SoC FPGA*s. Cet aspect sécurisation évolue de manière croissante avec l'augmentation de l'utilisation des *SoC FPGA*, que ce soit dans le but de séparer des applicatifs à criticité différente ou de partager une même cible entre plusieurs utilisateurs. Ainsi, travailler à cette sécurisation est un élément prépondérant dans l'adoption de ce type d'environnement d'exécution pour les *SoC FPGA*s. L'augmentation des systèmes embarqués faisant fonctionner plusieurs applications sur un même système d'exploitation a ouvert la route à de nouveaux vecteurs d'attaques. Que ce soit à des fins de vol de données ou d'altération du fonctionnement initial d'une application, le développement de ces attaques va de pair avec celles des architectures numériques et de leur complexification. En effet, l'augmentation de la part de marché des systèmes embarqués démontre un effort de la part des industriels pour accroitre ces technologies comme le montre la figure 1.14. Cet intérêt est complémentaire à la part prépondérante qu'occupent les nouvelles technologies aujourd'hui, où tout système devient embarqué et vise à être réduit. À contrario, cela pousse également à la mise en place de mesures et contre-mesures pour accéder au contenu de ces systèmes, à des fins d'espionnage ou de contrefaçon, par des entités étatiques ou industrielles, pour des raisons financières ou de sécurité nationale. Les attaques que peut subir un système sont donc nombreuses et sont susceptibles d'apparaître en tout cycle de l'offre et de la demande. C'est également le cas sur le marché des systèmes basés sur le FPGA.

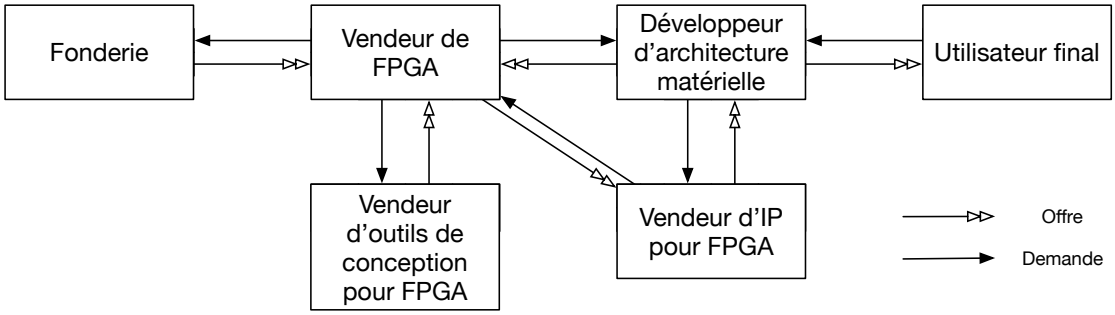


FIGURE 1.15 – Le marché de l'offre et de la demande des systémes à base de *FPGA*

La figure 1.15 est basée sur [77]. Elle montre l'ensemble des interlocuteurs du marché des architectures basées sur *FPGA*, de la fonderie jusqu'à l'utilisateur final. Les interlocuteurs de cette chaînes sont de gauche à droite :

- **1 - Fonderie** Le fondeur est en charge de la fabrication industrielle de la puce à partir de l'architecture fournie par le vendeur de *FPGA*.
- **2 - Vendeur de *FPGA*** Le vendeur de *FPGA* est l'entreprise conceptrice de l'architecture *FPGA*.
- **3 - Vendeur d'outils de conception pour *FPGA*** Le vendeur d'outil de conception pour *FPGA* est l'entreprise qui conçoit des outils pour faciliter le développement

d'architecture basée sur *FPGA*.

- **4 - Développeur d'architecture matérielle** Le développeur d'architecture matérielle développe des produits pour les marchés commerciaux à partir des *FPGA*s.
- **5 - Vendeur d'*IP* pour *FPGA*** Le vendeur d'*IP* pour *FPGA* est une entreprise qui développe des blocs matériels pouvant être implémentés sur *FPGA* pour des applications spécifiques.
- **6 - Utilisateur final** Qu'il soit une entreprise ou un utilisateur isolé, il est le créateur du besoin initial. Son objectif est d'utiliser la solution à base de *FPGA* pour répondre à un besoin.

Les interactions entre les différents acteurs de cette chaîne peuvent représenter des potentielles sources de vulnérabilité. De la même référence [77], les tableaux 1.7 et 1.8 en font la synthèse.

	Vulnérabilités		Acteurs du marché
a	Attaque par canaux auxiliaire	1	Fonderie
b	Attaque par rejeu	2	Vendeur de *FPGA*
c	Clonage	3	Vendeur d'outils de conception pour *FPGA*
d	**Fuite d'information**	4	Développeur d'architecture matérielle
e	Ingénierie inverse	5	Vendeur d'*IP* pour *FPGA*
f	**Trojan matériel**	6	Utilisateur final

Tableau 1.7 – Vulnérabilités et acteurs du marché

	(2 ← 1)	(2 ← 5)	(2 ← 3)	(4 ← 2)	(4 ← 3)	(4 ← 5)	(4 ← 6)	(5 ← 4)
a	○	○	○	○	○	○	●	○
b	○	○	○	○	○	○	●	○
c	○	○	○	○	○	○	●	○
d	●	○	●	●	●	○	○	○
e	○	○	○	○	○	○	●	●
f	●	●	○	●	○	●	○	○

Tableau 1.8 – Synthèse des vulnérabilités présentes dans le marché *FPGA*.
(x ← y) lire x vulnérable à y par rapport à l'ordonnée

Le tableau 1.8 expose la fuite d'information et le trojan matériel comme les deux vulnérabilités les plus présentes dans le marché d'architecture à base de *FPGA*.

Cette thèse se concentrera sur les problématiques liées à ces deux vulnérabilités et plus particulièrement aux moyens de défense possibles pour parer à celles-ci. Une contre-mesure visant à garantir l'isolation d'un environnement d'exécution logiciel et son matériel

sous-jacent sera identifiée pour contrer les tentatives d'intrusion et l'insertion de trojans matériels ou logiciels. Les architectures concernées sont celles à base de *FPGA* et plus particulièrement dans les *SoC FPGA*s, puisqu'il peuvent garantir l'exécution de plusieurs applicatifs sur une même cible physique. Dans ce cadre, l'étude de l'existant sera divisée en deux parties dont la première traitera de l'isolation matérielle et la seconde de l'isolation logicielle.

La littérature expose des travaux relatifs à l'isolation physique et géographique de tâches sur *FPGA*, mais également ceux en lien avec la gestion et la restriction d'accès. Cette méthode renvoie plus communément au terme de *sandboxing*. Elle permet d'isoler l'exécution de différents processus logiciel et d'accélérateurs matériel par l'interdiction et la restriction d'accès.

L'isolation matérielle est le fait d'interdire l'accès à une partie du matériel (logique ou mémoire) par des choix architecturaux et des mécanismes de contrôle d'accès. Au contraire, l'isolation logicielle interdit des accès logiciels par des choix algorithmiques avec l'implémentation logicielle de mécanismes de contrôle/restriction d'accès. Ces deux types d'isolation sont appliqués sur des architectures reconfigurables *FPGA*s pour l'isolation matérielle et sur des *CPU*s pour l'isolation logicielle.

Avant d'aller plus loin, une introduction à des notions de base de cryptographie est nécessaire. Selon [78], la cryptographie est la science de l'écriture secrète qui a pour objectif de cacher la signification d'un message. Cette action est appelée chiffrement. Il en existe deux types : symétriques et asymétriques.

- Chiffrement symétrique : Deux parties ont un algorithme de chiffrement ou déchiffrement pour lequel ils partagent une clé secrète. Le plus connu de ces algorithmes symétriques est l'AES [79] pour Advanced Encryption Standard.

- Chiffrement asymétrique : Une partie possède une clé privée et une clé publique. La clé publique est diffusée et les autres parties peuvent l'utiliser pour chiffrer des données. La clé privée est, quant à elle, utilisée pour le déchiffrement. Le plus connu des algorithmes asymétriques est le RSA [80] pour Rivest-Shamir-Adleman du nom de css auteurs.

- Authentification : Aussi connue sous le nom de fonction de hashage, cette primitive cryptographique permet à partir d'un message, d'obtenir une unique représentation de ce message appelé hash. La plus connue d'entre elles est le SHA [81].

1.3.1 Sandboxing matériel

[82] présente un mécanisme capable de gérer les restrictions de lecture/écriture et abstraire la mémoire physique via l'utilisation d'un contrôleur servant d'interface entre le monde physique et les interfaces virtualisées accessibles par l'utilisateur. Cette solution offre une isolation contre la contamination par cheval de Troie à partir d'une IP infectée. Pour cela, chaque IP est encapsulée dans des sandboxes et l'ensemble des transactions surveillées. Olson et al., dans [83], présentent un mécanisme de *sandboxing* de l'accès mémoire des accélérateurs matériels ajoutant un délai négligeable par rapport au temps d'exécution. Cette méthode implique l'ajout d'un bloc matériel entre l'accélérateur et la mémoire partagée.
La protection du noyau est étudiée dans [84]. Un module de protection est inclus dans le processeur ainsi que des modules de surveillance du matériel. Cette approche a un faible impact sur les performances, bien qu'elle soit optimisée spécifiquement pour les processeurs softcore.
[85] prend en considération la conception d'application conjointe matérielle/logicielle. Il en résulte une méthode permettant au matériel d'hériter des politiques de sécurité logicielle au moment de l'exécution. Une faible surcharge temporelle est introduite dans cette solution. Elle est obtenue en démarrant l'exécution en parallèle du contrôle de sécurité. Enfin, [86] présente une approche pour protéger les communications dans une architecture multiprocesseur utilisant des pares-feux matériels. Cette solution permet le chiffrement de données ainsi que des contrôles d'intégrité et de confidentialité. Cependant, elle introduit une pénalité en ressource proportionnelle au nombre de pare-feux présent dans le système.

1.3.2 Sandboxing logiciel

En 1996, [87] introduit un mécanisme de *sandboxing* pour restreindre l'accès des applications à un système d'exploitation en interceptant et en filtrant les appels système.
[88] propose une sandbox destinée au code natif x86 non fiable s'exécutant sur un navigateur web, tout en réduisant les performances globales de l'application de 2% à 5%. [89] présente un mécanisme de *sandboxing* pour les programmes exécutés par un utilisateur sans accès privilégiés au système d'exploitation.

La popularité croissante des appareils connectés ces dernières années a conduit les fabricants d'appareils à intégrer au processus de conception les questions de sécurité de manière plus prioritaire qu'auparavant. Afin d'aborder ces questions de manière appropriée, une spécification a été élaborée pour définir un moyen de garantir l'intégrité et la confidentialité des données circulant dans l'entité mettant en oeuvre cette spécification.

Pour aborder la question de la sécurisation des systèmes, il a dans un premier temps été défini la notion d'environnement d'exécution de confiance (TEE pour Trusted Execution Environment). Ce TEE est à opposer à l'environnement d'exécution enrichi (REE pour Rich Execution Environment). Un TEE est une zone sécurisée à l'intérieur d'un *CPU*. Il fonctionne en parallèle du système d'exploitation du REE, dans un environnement isolé. Il garantit que le code et les données chargés dans le TEE sont protégés en terme de confidentialité et d'intégrité. Ce système parallèle est destiné à être plus sûr que le système classique (REE) en utilisant à la fois du matériel et des logiciels pour protéger les données et le code. Les applications de confiance s'exécutant dans un TEE ont accès à la pleine puissance du *CPU* et de la mémoire, tandis que l'isolation matérielle protège ces composants des applications installées par l'utilisateur et s'exécutant dans le système d'exploitation principal. Les isolations logicielles et cryptographiques à l'intérieur du TEE protègent également les différentes applications de confiance les unes des autres.

Intel a pour sa part proposé un environnement d'exécution sécurisé pour ses processeurs. Intel SGX pour Intel Software Guarded eXtention [90] a été introduit comme étant un ensemble d'instructions fournissant des primitives pour exécuter une application dans une zone mémoire privée prédéfinie (à savoir des enclaves). Pour renforcer les enclaves proposées par Intel SGX, Ryoan [91] décrit une technique permettant de confiner les modules de traitement de données non fiables et empêcher la fuite des données d'entrées des utilisateurs. Comme présenté en figure 1.16, cette technique est basée sur une chaîne de confiance permettant de confirmer l'authentification du binaire devant être exécuté. Ceci est effectué à partir d'un hash fournit par le développeur du binaire devant être exécuté couplé au Méta qui contient la configuration SGX. Ryoan permet d'assurer le lancement du module contenant le binaire à exécuter après vérification du hash signé. La chaîne de confiance démarre comme suit : ❶, SGX atteste qu'une instance valide d'une sandbox Ryoan existe ❷. La sandbox Ryoan valide le module contenant le binaire à exécuter à partir du hash associé à ce module ❸.

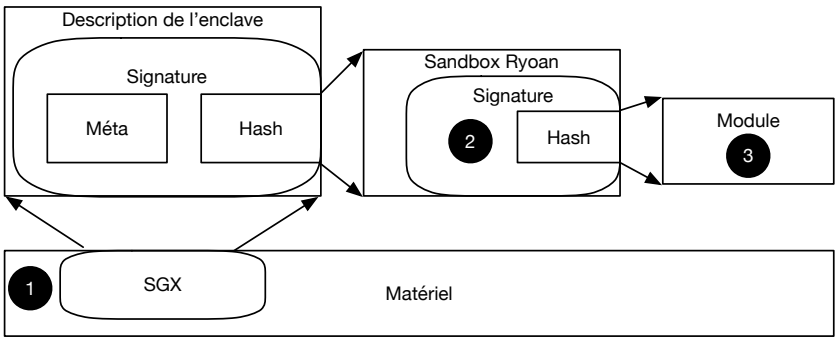


FIGURE 1.16 – La chaine de confiance de Ryoan

ARM a également proposé une solution pour permettre une isolation entre différent modes d'exécution : ARM Trustzone [92]. Cette solution est basée sur des extensions architecturales de sécurité et une couche logicielle. Un coeur de *CPU* physique fournit deux coeurs de *CPU* virtuels, l'un étant considéré comme non sécurisé, l'autre étant considéré comme sécurisé, ainsi qu'un mécanisme de changement de contexte entre les deux, connu sous le nom de moniteur. La figure 1.17 offre une représentation de l'environnement d'exécution offert par la Trustzone.

Dans le TEE se trouve le moniteur ainsi qu'un système d'exploitation et des applications optionnelles. Une implémentation Trustzone peut être constituée de tous ces composants comme par exemple la solution Trustonic [93], ou seulement d'un moniteur comme Nintendo l'utilise dans la Nintendo Switch.

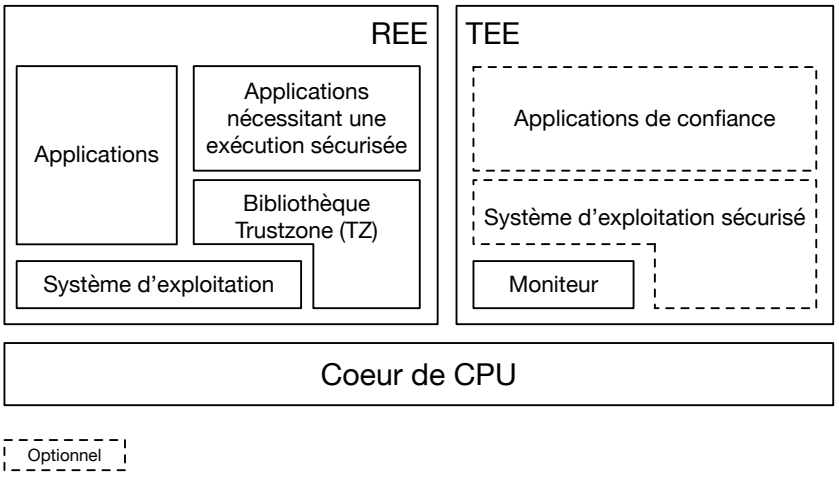


FIGURE 1.17 – L'implémentation logicielle de la Trustzone

L'implémentation d'un système d'exploitation sécurisé permet d'ajouter des fonctionnalités de confiance qui sont destinées à fournir un service sécurisé supplémentaire au monde normal. L'exécution côté REE est appelée mode normal et côté TEE mode sécurisé. Un troisième mode existe et sert d'interface entre les deux, le mode moniteur. Le mécanisme par lesquels le *CPU* peut entrer en mode moniteur depuis le mode normal peut être déclenché par l'exécution d'une instruction dédiée appelée Secure Monitor Call (SMC). Un bit dans le registre de configuration sécurisé indique dans quel environnement d'exécution est le programme (TEE ou REE) à chaque instant de l'exécution. Celui-ci est diffusé à la mémoire principale afin d'isoler la mémoire entre les mondes Sécurisés et Non-Sécurisés. La solution Trustzone possède également une procédure de démarrage sécurisée. Partant de la ROM de démarrage, les signatures du bootloader, du système d'exploitation sécurisé ainsi que celle du système d'exloitation sont successivements vérifiées pendant la séquence de démarrage.

Bien qu'apportant une solution clé en main pour l'isolation d'applications à différents niveaux de criticité sur un même *CPU*, des failles ont été identifiées. C'est le cas notamment de la compromission apportée par [94] qui expose une méthode pour compromettre un *SoC FPGA* utilisant Trustzone pendant la phase de démarrage. [95] et [96] offrent une évaluation de la solution proposée par ARM et proposent des scénarios de compromission de la solution d'isolation. Celle-ci bénéficiant de faiblesse introduite par la propagation de la solution Trustzone dans une architecture hétérogène de type *SoC FPGA*.

Cependant, de même que les méthodes de sandboxing matériels qui ne s'appliquent qu'au matériel, ces techniques de sandboxing logiciels ne s'appliquent qu'à l'exécution logicielle sur *CPU*. A l'heure actuelle, l'utilisation de mécanisme de sandboxing sur une architecture reconfigurable hétérogène composée de *CPU* et *FPGA* de type *SoC FPGA* nécessite des méthodes de sandboxing matériel **et** des méthodes de sandboxing logiciel. Notre attention s'est donc portée sur le développement d'une solution permettant une isolation complète du bas niveau matériel jusqu'à l'applicatif. C'est l'approche qui est présentée dans le chapitre 3. Elle porte sur l'apport d'un mécanisme de sandboxing matériel/logiciel pour prévenir de l'insertion de chevaux de Troie.

Les architectures reconfigurables hétérogènes de type *SoC FPGA* ont la flexibilité nécessaire pour que le *FPGA* soit mis à jour afin d'en modifier le fonctionnement ou d'en corriger un problème de conception. Or, ces mises à jour ne sont pas disponibles sur le *CPU* du *SoC FPGA* qui, de par sa conception est *hardcore* et ne peut donc pas être reconfiguré. Il est possible d'effectuer ce type de reconfiguration sur un *CPU softcore* qui serait implémenté sur le *FPGA*. Ceci est à mettre en relief avec la possibilité d'utilisation de microcode, afin de pouvoir proposer une solution permettant, via la modification de celui-ci, de pouvoir corriger un problème de conception ou ajouter des instructions dans une certaine mesure. Ceci fait l'objet de la section suivante.

1.4 Le microcode dans une architecture reconfigurable

Le composant principal en charge du contrôle de l'exécution des instructions d'un *CPU* est l'unité de contrôle. Deux méthodes existent pour l'implémenter. Une première consiste en l'implémentation de chaque instruction de manière figée dans le circuit. Une seconde est l'implémentation d'un circuit de contrôle configurable via l'utilisation de microcode. Les *CPU*s utilisant ce circuit de contrôle configurable sont dit microcodés.

Le microcode est une couche intermédiaire qui fait le lien entre le jeu d'instructions et l'architecture matérielle ciblée. Son rôle est de faire le lien entre les signaux électriques qui contrôlent le circuit physique et l'utilisation de celui-ci via des instructions logicielles.

L'architecture du jeu d'instruction (ISA pour Instruction Set Architecture) fournit une interface cohérente avec le logiciel et définit les instructions, les registres, l'accès mémoire, les E/S et la gestion des interruptions. Nous désignons les instructions de l'ISA par macroinstructions. La microarchitecture décrit comment le fabricant a mis à profit les techniques de conception des processeurs pour mettre en œuvre l'ISA. D'un point de vue haut niveau, les composants internes d'un processeur peuvent être subdivisés en chemin de données et unité de contrôle. Le chemin de données est composé d'un ensemble d'unités fonctionnelles telles que les registres, les bus de données et l'unité logique arithmétique (ALU). L'unité de contrôle contient le compteur de programmes (PC), le registre d'instructions (IR) et l'unité de décodage des instructions. Elle utilise diverses unités fonctionnelles afin de piloter l'exécution du programme. Plus précisément, l'unité de contrôle traduit chaque macro-instruction en une séquence d'actions, c'est-à-dire qu'elle récupère les données d'un registre, effectue une certaine opération ALU, puis réécrit le résultat. Le signal de contrôle est l'ensemble des signaux électriques que l'unité de contrôle envoie aux différentes unités fonctionnelles au cours d'un cycle d'horloge. Le microcode est donc une concaténation des signaux de contrôles utilisés à un cycle d'horloge pour l'exécution d'une microinstruction. Une somme de microinstruction est appelé une macroinstruction.

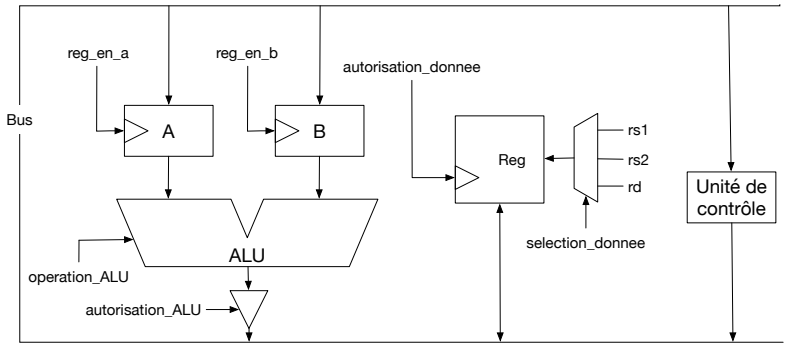


FIGURE 1.18 – Schéma de l'architecture microcodée relative aux signaux de contrôle figure 1.19

Dans cet environnement, une macroinstruction est la traduction en une somme de microinstructions d'une instruction du langage assembleur. Celles-ci, développés en fonction de l'architecture ciblée fournissent les signaux pour effectuer l'opération souhaitée. Dans la figure 1.18 est présenté un exemple d'architecture qui utilise les signaux de contrôle décrit dans la figure 1.19. Dans cet exemple qui n'est pas basé sur un jeu d'instructions particulier, il est présenté une décomposition possible de l'instruction ADD en micro-instruction. Pour exécuter une addition en assembleur, il faut dans un premier temps récuperer la valeur du premier registre de l'addition (Rs1) dans la valeur A qui sera en entrée de l'ALU, faire de même dans un second temps avec le second

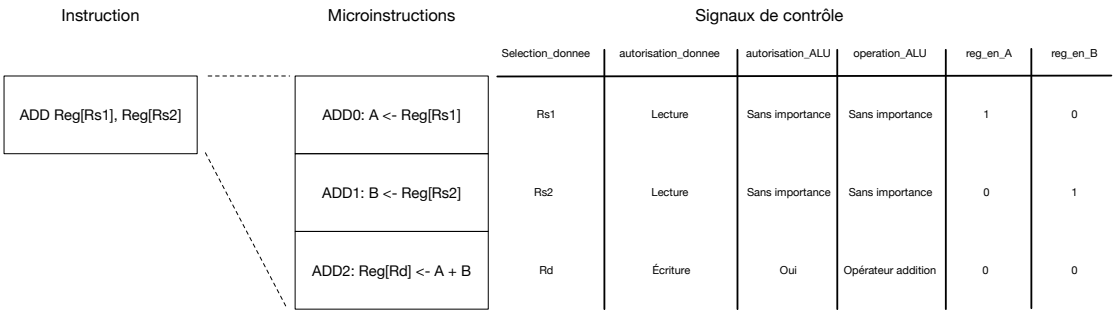


FIGURE 1.19 – Exemple de la décomposition d'une instruction en un ensemble de microinstructions traduisant des signaux de contrôle

registre (Rs2) qui sera positionné dans B. Enfin il faut effectuer l'addition est récupérer le résultat (Rd). Un enchaînement de microinstructions est une macroinstruction qui traduit une instruction assembleur de l'architecture cible. Celle-ci est donc divisée en une séquence de microinstructions exécutées en plusieurs cycles horloge. Le microcode est généralement stocké dans une mémoire physiquement très proche du circuit dans lequel doivent être diffusés les signaux de contrôle pour limiter la latence d'accès. L'objectif est de pallier la pénalité introduite par l'utilisation de plusieurs cycles d'horloge pour exécuter une instruction assembleur. Avant de pouvoir exécuter une macroinstruction, il est préalablement nécessaire de récupérer l'adresse à laquelle celle-ci est stockée en mémoire. Pour cela, une instruction spécifique est utilisée avant chaque appel à une instruction microcodé. Il s'agit de l'instruction de "Fetch".

1.4.1 Exemples d'utilisation du microcode

Les deux exemples présentés ont pour objectif de démontrer la flexibilité et l'évolutivité offertes par l'utilisation du microcode. Un premier exemple expose la flexibilité introduite par l'utilisation du microcode dans les jeux vidéo, et notamment au coeur du coprocesseur graphique de la Nintendo 64. Un second exemple, plus récent, expose le bénéfice que peut apporter l'utilisation de microcode pour développer des instructions complexes dans les architectures processeurs via la présentation d'une extension au jeu d'instructions présent sur *CPU* Intel, Intel SGX [90].

La Nintendo 64 a été mise en vente en 1996. Celle-ci était essentiellement composée d'un processeur NEC VR4300 cadencé à 93.75 MHz ainsi que d'un coprocesseur en charge des graphismes et de l'audio nommé Reality CoProcessor (RCP) cadencé à 62.5 MHz. Ce dernier était décomposé en deux sous-blocs : le Realidy Drawing Processor (RDP) en charge de la rastérisation de l'image à afficher et le Reality Signal processor (RSP) en charge des calculs graphiques et de l'audio. Le RSP était programmable via

l'utilisation de microcode. Ce dernier était décomposé en deux parties : une première nommée Graphical Binary Interface (GBI) regroupant les instructions relatives à la gestion graphique et une seconde nommée Audio Binary Interface (ABI) en charge de la gestion audio. Initialement le microcode était fourni par Nintendo et utilisé par les éditeurs de jeux. Cependant, des éditeurs comme Factor 5 développèrent des mises à jour du microcode GBI qu'ils intégrèrent dans les cartouches de jeux, afin de proposer des améliorations graphiques comme dans les jeux Star Wars : Rogue Squadron où un moteur de génération de paysage fut intégré ou encore Star Wars : Episode I : Battle for Naboo où le moteur de paysage et de particules fut amélioré.

Intel SGX pour Software Guard eXtention est un ensemble d'instructions fonctionnant sur des architectures de microprocesseurs Intel et dont le but est de fournir des primitives garantissant l'intégrité et la confidentialité du calcul effectué dans un environnement où toutes les couches logicielles (noyaux, hyperviseur, etc..) sont potentiellement corrompues. L'utilisateur peut exécuter une application de manière sécurisée via l'allocation de zone mémoire privée (appelée enclave). Cette extension contient dix-huit nouvelles instructions (cinq accessibles par l'utilisateur et treize par l'hyperviseur), toutes décrites en microcode qui devient le mode d'implémentation privilégié pour la description d'instructions complexes [97]. Cependant, cette solution part du principe que les enclaves sont sûres. L'étude menée par Schwarz et al. [98] propose un modèle de menace tangible pour outrepasser les protections offertes par cette extension et en démontre une mise en oeuvre. Il se pose donc la question d'un modèle de menace ne garantissant pas la confiance du code source exécuté dans l'enclave.

1.4.2 Le microcode et la sécurité

Aujourd'hui, le microcode est une couche d'abstraction présente dans la plupart des processeurs du marché. Elle permet l'implémentation d'une sous-partie des instructions du processeur, couplée avec l'utilisation d'instructions dites "hardwired" directement exécutées par un circuit fixe [99]. Ces instructions microcodées ont pour but de faciliter la mise en place d'instructions plus complexes, mais également la mise à jour du processeur sans besoin de matériel spécifique, c'est à dire via une mise à jour effectuée en logiciel. En ce sens, l'article [12] propose une méthode permettant de faire l'ingénierie inverse du microcode des processeurs AMD K8 et K10 utilisant une architecture x86. Dans cette étude, il est également proposé une méthode de mise à jour sur-mesure du microcode offrant des possibilités de modification de celui-ci. Il est enfin démontré que cette méthode permet d'introduire un cheval de Troie qui, lors de l'utilisation du processeur peut être accédé via un navigateur web et permettre l'exécution de code malveillant à distance.

L'utilisation de microcode a donc un impact majeur sur la sécurité d'un système [100, 101] que ce soit à des fins offensives avec l'introduction de code malveillant, mais également à des fins défensives via la possibilité de mise à jour post-production pour répondre à des failles de type Spectre ou Meltdown [102].
Les principaux avantages et inconvénients liés à l'utilisation d'une architecture microcodée sont présentés dans le tableau 1.9. Au delà des limites en fréquence d'utilisation que peut introduire l'utilisation d'un *CPU softcore* microcodé, celui-ci apporte plusieurs avantages significatifs en terme de flexibilité et sécurité. Il se pose cependant la question du mode de développement de tels *CPU*s. Celui-ci étant la somme de 1) une description architecturale du *CPU* et 2) le microcode associé à l'architecture. C'est pourquoi, la section suivante fixe les éléments relatifs à la génération de ce type d'architecture.

Avantages	
Evolutivité	Il est possible de corriger des erreurs post production ou apporter des optimisations.
Flexibilité	Plusieurs jeux d'instructions peuvent être utilisés avec la même architecture. Également, le développement d'instruction complexe sur mesure est possible via l'utilisation d'une séquence de microinstruction simple.
Sécurité	L'utilisation du microcode permet de pallier des failles de sécurité qui seraient identifiées via la mise à jour de celui-ci.
Inconvénients	
Vitesse	La vitesse d'un processeur microcodé est directement liée à la vitesse du contrôleur de microcode du processeur. Bien que celui-ci soit architecturalement optimisé, sa limite de vitesse est implicitement liée à la fréquence d'accès à la mémoire contenant le microcode.
Temps de fetch	Entre chaque instruction, une instruction de fetch doit être appelée afin de récupérer l'instruction suivante devant être exécutée.

Tableau 1.9 – Avantages et inconvénients liés à l'utilisation d'une architecture microcodée comparé à l'utilisation d'une architecture câblant directement chaques instructions.

1.5 La génération d'architecture microcodée

Des méthodes améliorant l'accessibilité au développement matériel par des ingénieurs non spécialistes émergent et tentent d'apporter des solutions à différents niveaux d'abstraction.

1.5.1 Description architecturale

Des efforts ont d'abord été fournis pour améliorer les langages de description matérielle en réduisant leur expressivité. PyMTL [103], SysPy[104], MyHDL[105], PyHDL[106] ou PyRTL[107] en sont des exemples. En parallèle, des outils comme la synthèse de haut niveau (*HLS* pour High Level Synthesis) [108, 109] décrivent l'architecture depuis un langage de programmation comme le C, C++ ou encore SystemC. Le code source est ensuite compilé pour générer du code *HLS* [110, 111]. Du fait de l'amélioration des outils relatifs à la *HLS* et l'intérêt grandissant de l'industrie pour l'utilisation des *FPGA*s, d'autres outils permettant la génération de code HDL depuis des langages de programmation ont émergé. L'objectif premier est de proposer une solution plus simple d'utilisation d'une technologie qui est jusqu'à présent exclusivement destinée aux experts [112, 113]. Enfin, Chisel [114] a été développé par UC Berkeley dans le but de fournir un outil capable, à partir d'une description architecturale en Scala, de générer l'architecture en verilog. Depuis sa création, il a suscité l'adhésion de la communauté et s'est entre autre développé pour la génération d'architecture processeur softcore [115]. Les suites d'outils utilisant la HLS pour la génération de blocs matériels intègrent également des primitives permettant l'intéraction et l'inter-communication des blocs [116].

1.5.2 Automatisation du développement de microcode

La simplification des méthodes de description architecturale a ouvert la porte au développement de jeux d'instructions pouvant être implémentés sur *CPU*. Ils sont basés sur un modèle *" base + extension "* et offrent des possibilités de création d'instructions personnalisées [117]. Outre le développement de l'architecture, l'utilisation de ce type de jeu d'instructions avec une architecture microcodée nécessite la génération du microcode associé. Des solutions basées sur une traduction dynamique en microcode ont été présentées en 1981. Baba et Hagiwara [118] ont présenté des travaux sur la génération de microprogramme, plus spécifiquement de microinstructions heuristiques et independantes de l'architecture ciblée. Cette solution permet la génération de signaux de contrôle offrant une grande flexibilité. [119] expose également un outil de génération indépendant de l'architecture cible. Ces deux solutions offrent de la flexibilité, cependant elles introduisent une perte de performance. Une autre approche étudiée par Sheraga dans [120] consiste en la traduction statique du code source en microcode via l'utilisation de macro. Cette solution pallie les pertes de performances en mappant directement via des macros des blocs élémentaires de microcode. Cette solution améliore les performances mais introduit une perte de flexibilité. [121] tente d'aporter une solution à ces lacunes en présentant un générateur de microcode prenant en entrée le code source d'un algorithme et une

description de l'architecture cible et génère le microcode associé.

Dernièrement des outils de descriptions architecturales ont été étudiés en profondeur pour proposer une solution complète en accord avec les évolutions du matériel [114]. Il est proposé des solutions qui permettent la génération d'architecture matérielle performante. D'un autre côté, les outils de génération de microcode ont principalement été étudiés dans les années 1980, mais très peu depuis l'avènement des solutions *FPGA* et *SoC FPGA* modernes. **Dans le chapitre 4 une solution permettant de proposer une description d'architecture en langage *HDL* implémentable sur cible *FPGA* ou *SoC FPGA* et son microcode associé seront présentés.**

Conclusion

Une première section expose l'architecture des *SoC FPGA* ainsi que les étapes de conception d'architectures sur de tels systèmes. Par la suite, la section 1.2 relative aux environnements d'exécution pour architectures hétérogènes reconfigurables offre un panel de solutions pour la gestion de ceux-ci. Que ce soit via l'ajout dynamique au noyau des drivers ou la proposition d'une couche d'abstration du matériel, elles se basent sur un système d'exploitation ou a minima une version allégée d'un noyau linux. Les solutions faisant exception ne développent pas un système d'exploitation mais un contrôleur amélioré. Dans une optique de développement d'un système léger capable d'implémenter des mécanismes de sécurité, nous exposons nos contributions au chapitre 2. Nous proposons ensuite une solution qui permet l'exécution de matériel et logiciel embarquant seulement les primitives nécessaires. Celle-ci permet de limiter les vecteurs d'attaques potentiels dûs à la gestion des resources de l'architecture hétérogène reconfigurable ciblée.

La sécurité des architectures reconfigurables via des mécanismes d'isolation a largement été étudiée dans ses aspects logiciels. Elle est également devenue un axe de recherche évoluant de manière croissante et proportionnellement à l'intérêt que porte l'industrie aux architectures reconfigurables. Le développement d'architecture de type *SoC FPGA* permettant l'exécution de code logiciel et d'accélérateur matériel conduit à proposer dans le chapitre 3 une solution abordant la problématique avec une vue combinée matérielle/logicielle.

La résolution de certaines failles de sécurité récentes comme Spectre ou Meltdown dans les processeurs du commerce Intel via la mise à jour du microcode a démontré deux choses. Premièrement, du microcode est présent dans une majorité des processeurs du commerce même s'il n'implémente pas l'ensemble des instructions disponibles sur ceux-ci.

Deuxièmement, l'utilisation d'architecture microcodée offre la flexibilité nécessaire à la mise à jour post-production, notamment de faille de sécurité ou de fonctionnalité. Pour tirer partie des bénéfices précédemment cités, le chapitre 4 exposera une architecture de *CPU* à jeu d'instructions RISC-V. De plus, peu de solutions relatives à la génération d'une combinaison architecture et microcode associé n'a fait l'objet de contributions significatives. Ce même chapitre proposera donc un générateur associé à ce *CPU*. Il permettra la génération de l'architecture ainsi que la génération du microcode associé.

2 Lynq, un environnement d'exécution

Ce chapitre présente Lynq, un environnement d'exécution qui répond aux problématiques de développement et de gestion d'architectures hétérogènes reconfigurables sans système d'exploitation, programmable depuis un langage de haut niveau. Les bases de cet environnement d'exécution sont présentées dans une première section. La seconde traite des éléments et de la méthodologie relatifs à la génération de Lynq. Puis, les deux sections suivantes décrivent la manière d'utiliser cet environnement ainsi que les performances atteintes. Enfin, la dernière partie présente le serveur web adossé à la solution proposée.

2.1 Introduction

2.1.1 Objectifs

Comme présenté dans le chapitre 1, des approches existent pour faciliter le développement de *SoC FPGA*. De plus, la génération automatisée d'accélérateurs matériels à l'aide des technologies HLS ainsi que l'utilisation de nouveaux outils d'intégration [122] réduisent la complexité de conception des *SoC FPGA*. L'utilisation de langage de programmation de haut niveau comporte donc plusieurs intérêts. Entre autre, elle permet de simplifier ces approches et d'accélerer le temps de développement qui demande un effort continu depuis des décennies.

Dans cette direction, Xilinx a récemment mis sur le marché l'environnement PYNQ [54]. Cette solution utilise le langage de programmation Python et est basée sur des architectures de base (overlays) que l'utilisateur peut utiliser et modifier. L'API PYNQ se présente sous la forme d'une bibliothèque Python permettant l'accès au *FPGA* par le biais d'entrées / sorties "mappées" en mémoire (*MMIO*) et d'accès en mémoire directe (*DMA*). Les concepteurs d'applications peuvent donc les développer sans la nécessité de concevoir des accélérateurs matériels en y associant des pilotes de périphérique. Ainsi, le concepteur devient simplement un utilisateur de bibliothèque d'accélérateurs matériels (*IP*) pour lesquels PYNQ fournit une interface afin d'en simplifier l'accès. Les blocs *IP* sont contrôlés par un interpréteur Python fonctionnant dans la partie système de traitement (*PS*) du *SoC FPGA*. Ils peuvent être chargés dynamiquement dans la partie logique programmable (*PL*) au moment de l'exécution afin de fournir les fonctionnalités requises par une application logicielle spécifique.

Schmidt [123] a évalué la pertinence du PYNQ pour le développement rapide d'applications en étudiant son impact sur les performances ainsi que la localisation des goulots d'étranglement associés. Un algorithme de détection de contours a été implémenté en Python. Il en résulta un temps d'exécution 334 fois plus lent que la version C. Une accélération de 11.5 fois par rapport à la version C a été observée dans le cas où la bibliothèque OpenCV était utilisée. Elle améliore le temps d'exécution jusqu'à 30 fois si elle est couplée à l'utilisation d'accélérateur matériel. Le résultat montre que les avantages de Python sont limités si la performance en vitesse est une nécessité. Cette limitation est principalement causé par la faible efficience de l'interpréteur Python mais le choix de celui-ci comme langage pour le contrôle d'accélérateur matériel est judicieux pour faciliter le prototypage rapide.

Pour pallier le problème lié à la performance, Rigo [124] a proposé PyPy. Il s'agit d'une

implémentation alternative qui utilise un interpréteur et un compilateur *JIT*. Cependant, PyPy implémente un sous-ensemble restreint du langage Python et inclut des dépendances fortes ainsi que la présence d'un système d'exploitation. La synthèse relative aux solutions énoncées jusqu'a présent est faite dans le tableau 2.1.

				Architecture			Plateforme				Compatibilité	
	Ref.	Language	License	x86	x64	ARM	Win.	Lin.	OSX	Standalone	*JIT*	PL
Python	[125]	Python	PSF	●	●	●	●	●	●	○	○	○
PyPy	[124]	Python	MIT	●	●	●	●	●	●	○	●	○
PYNQ	[54]	Python	BSD	○	○	●	○	●	○	○	○	●
BITS	[126]	Python	n/a	●	○	○	○	○	○	●	○	○
Lua	[127]	Lua	MIT	●	●	●	●	●	●	○	○	○
Lua*JIT*	[128]	Lua	MIT	●	●	●	●	●	●	○	●	○
(notre contribution)		Lua	MIT	○	○	●	○	○	○	●	●	●

JIT = Just-In-Time (compiler)
PL = Programmable Logic

Tableau 2.1 – Positionnement de Lynq par rapport à l'existant

Quelle est donc la meilleure manière de tirer profit d'un langage de haut niveau pour le développement et le contrôle d'accélérateur matériel ? Est-il possible d'améliorer les mécanismes de gestion d'une architecture hétérogène reconfigurable tout en préservant ou en limitant fortement a minima la perte de performances ? Pour répondre à ces questions, le firmware d'intégration et de contrôle d'architecture hétérogène reconfigurable open-source Lynq est proposé et présenté dans ce chapitre. Cet environnement de développement logiciel est basé sur le langage de programmation de haut niveau Lua.
Il offre une alternative légère à PYNQ de par ses interfaces de contrôle matériel et logiciel unifiées dans le sens où les accès matériels et logiciels sont rendus possibles depuis une même bibliothèque de haut niveau. Lynq limite les pertes en performance grâce à l'utilisation d'un compilateur juste à temps (*JIT* pour Just-In-Time). D'autre part, les outils permettant l'utilisation de ces solutions sont devenus une composante favorisant ou entravant leur adoption. Une des solutions mise en oeuvre, notamment quand un système d'exploitation n'est pas disponible consiste en l'utilisation d'un serveur web.

De nombreux serveurs web pour systèmes embarqués sont aujourd'hui disponibles. Certains sont économes en ressources, par exemple, LightTPD [129] est un serveur web fonctionnant sous Linux et Windows. Son objectif est d'avoir une empreinte mémoire faible tout en administrant un grand nombre d'applications et en conservant une charge *CPU* faible. Il existe également NGINX [130] qui utilise une architecture asynchrone basée sur les événements. Cependant, le système nécessite Linux et une implémentation consomme à minima 10 Mo de RAM. Monkey Server [131] a une petite empreinte mémoire de 100 Ko mais a besoin de Linux ou OSX pour fonctionner. Cherokee [132] et Hiawatha [133] sont deux serveurs web supplémentaires pour des systèmes embarqués

ayant initialement été conçus pour les serveurs plutôt que pour les systèmes embarqués. Ces deux solutions ne sont pas applicables à nos besoins.
Les interfaces de contrôle, telles que les serveurs Web, même s'ils ont une empreinte mémoire faible, utilisent encore généralement des systèmes d'exploitation consommateurs en mémoire. Ceci constitue un obstacle à une utilisation efficace de l'environnement de développement de bout en bout. De plus, les serveurs Web actuels pour les systèmes embarqués nécessitent souvent un code compilé plutôt qu'un code interprété.
La table 2.2 reprend les caractéristiques importantes des serveurs web présentés précèdemment. Un compromis compatible avec nos besoins, à savoir une solution standalone, légère et fonctionnant sans système d'exploitation pour exécuter du code compilé et interprété n'existe pas. Pour pallier cela, nous avons développé une solution capable de contrôler et configurer les ressources présentes sur le système embarqué cible telles que les fonctionnalités du *SoC FPGA* et ses interfaces.

	NGINX [130]	**LightTPD** [129]	**Monkey** [131]	**Cherokee** [132]	**Hiawatha** [133]
Asynchrone	●	○	○	○	○
Basé sur un système d'exploitation	●	●	●	●	●

Tableau 2.2 – Spectre des différentes fonctionnalités des serveurs web disponibles pour systèmes embarqués

2.1.2 Caractéristiques des contributions

Pour répondre aux problématiques exposées en introduction et atteindre les objectifs fixés section 2.1.1, nous avons développé Lynq, un environnement d'exécution léger basé sur Lua*JIT* et capable de fonctionner sur *CPU softcore* et *hardcore*. Cet environnement d'exécution offre aux développeurs de systèmes embarqués la possibilité de bénéficier des capacités spécifiques des *SoC FPGA*s, tout en conservant un haut niveau de performances. Associé à Lynq, nous avons développé un serveur web qui fonctionne en standalone et qui peut être exécuté sur le *SoC FPGA*. Les caractéristiques de nos contributions, à savoir Lynq et son serveur web, sont résumées ci-dessous.

- **Standalone.** Lynq est un firmware qui s'appuie directement sur les drivers et bibliothèques bas niveaux du constructeur du *SoC FPGA* cible. Cependant, il est à même de s'intégrer comme un applicatif dans un dispositif pourvu d'un système d'exploitation Linux.
- **Léger.** Lynq a été conçu pour pouvoir être embarqué dans des systèmes pauvres en ressources matérielles. Notamment dans des *SoC*s dont la part accordée à

l'interface de contrôle doit être très peu consommatrice de mémoire (que ce soit pour le stockage ou l'utilisation).

- **Économe en énergie.** Lynq, de par ses spécificités d'implémentation et la capacité de son compilateur juste à temps (*JIT*) offre des performances calculatoires supérieures au langage Python en MFLops/Watts.
- **Unifié.** Lynq propose une couche d'abstraction du matériel permettant l'accès aux différents périphériques du *SoC FPGA* depuis une *API*. Celle-ci a été développée de manière modulaire pour permettre à l'utilisateur de charger uniquement les modules utiles à l'applicatif visé. L'utilisateur a également la possibilité de créer et d'ajouter des modules selon ses besoins.
- **Démarrage rapide.** Lynq a une phase de démarrage et d'initialisation de 20 ms. Cela se fait au détriment de l'écosystème de services disponibles nativement avec les solutions basées sur un système d'exploitation.
- **Version standard.** Lynq propose une version embarquant un interpréteur Lua compatible avec toutes les architectures *CPU*, qu'elles soient *hardcore* ou *softcore*.
- **Version *JIT*.** Lynq propose également une version basée sur Lua*JIT* offrant un interpréteur optimisé et un compilateur *JIT*. Ceci permet d'obtenir des performances calculatoires supérieures aux versions Lua standards. Par contre, cela se fait au détriment d'une compatibilité avec certaines architectures *CPU*.
- **Performant.** Lynq, dans sa version *JIT*, tire parti d'un compilateur *JIT* pour proposer des performances lui permettant de se positionner, en plus de ses capacités de contrôle de *SoC FPGA*, comme une alternative logicielle calculatoire intéressante pouvant se rapprocher des performances d'un programme C compilé.
- **Flexible.** Lynq, dans sa version *JIT*, propose également l'exécution de code C via l'utilisation d'une interface d'appel à des fonctions externes (FFI). Ceci permet, dans un programme Lua, de faire des appels transparents, à des fonctions en C préalablement compilées.

Le serveur web a été développé pour fonctionner sur *SoC FPGA* tels que les *SoC* Zynq 7-serie de chez Xilinx ou la famille Arria 10 de chez intel*FPGA*. Nous l'avons développé afin qu'il réponde aux trois caractéristiques suivantes :

- **Standalone.** Un des points principaux était la capacité du serveur web à fonctionner sans le support d'un système d'exploitation. Pour arriver à cela, le serveur web est basé sur les couches d'abstraction et de services fournis par Lynq. Cela permet au serveur web de tirer parti des capacités de compilation JIT et d'interprétation, ainsi que de la librairie permettant l'accès au matériel.
- **Faible empreinte mémoire.** Un second point était la diminution la plus importante possible de l'empreinte mémoire nécessaire au stockage et au bon fonctionne-

ment de ce serveur web afin de pouvoir l'embarquer dans des systèmes embarqués à faibles ressources matérielles. La taille du serveur web basé sur Lynq est de 80kB et il nécessite 150kB de mémoire RAM pour fonctionner.

- **Modulaire.** Pour rester dans l'esprit de faible consommation, l'architecture du serveur web a été développée en suivant une logique modulaire afin que l'utilisateur puisse charger et utiliser seulement les modules nécessaires à ses besoins.

Pour fonctionner, un module permettant la gestion des connections à un réseau est nécessaire. Deux modules peuvent être chargés pour proposer des fonctionnalités complémentaires :

- **Module de développement.** Ce module permet d'écrire et de tester du code en direct. Il supporte l'écriture de script, leur exécution en temps réel et le contrôle de la cible *SoC FPGA*.
- **Module de visualisation.** Ce module permet la visualisation de données. Utilisé avec le module de programmation en direct, il permet de créer un environnement de développement avec obtention de résultats en cours d'exécution.

2.2 Les bases de l'environnement d'exécution

2.2.1 Architecture cible

Dans le cadre de notre étude, un des objectifs était de pouvoir contrôler finement une architecture contenant plusieurs zones reconfigurables et profiter des capacités de la Reconfiguration Dynamique Partielle du *FPGA*. La figure 2.1 décrit l'architecture *SoC FPGA* utilisée dans le cadre de notre étude. Celle-ci est composée d'une partie logicielle (PS), de mémoire DDR et d'une partie matérielle (PL). L'architecture matérielle propose deux zones reconfigurables (PRR1 et PRR2). Elle implémente une AXI DMA utile aux transferts de mémoires entre la DDR et les zones reconfigurables ainsi qu'un contrôleur de reconfiguration décrit au paragraphe suivant. Un timer peut être ajouté si l'applicatif le nécessite.

Un contrôleur de reconfiguration dynamique partielle (*PRC*) personnalisé, décrit en figure 2.2, a été conçu en complément du module de gestion de *FPGA* de Lynq pour tirer profit de la vitesse maximale de reconfiguration autorisée. Le *PRC* est composé d'un Advanced eXtensible Interface (*AXI*) Direct Memory Access (*DMA*), d'un bloc pour mettre en forme les données et d'un port d'accès à la mémoire de configuration.
L'*AXI DMA* transfère les données de la mémoire DDR jusqu'au bloc de remise en forme de la mémoire par mots de 32 bits. Une fois ceux-ci remis en forme pour répondre aux

spécifications requises par le port de reconfiguration, ils y sont transférés pour démarrer la reconfiguration du *FPGA*. L'objet de ce contrôleur n'est pas d'améliorer le temps de reconfiguration, qui peut être obtenu en augmentant la fréquence d'horloge en entrée du port au-delà de ses spécifications comme dans [134], mais de bénéficier en pratique du débit maximum théorique annoncé par le vendeur.

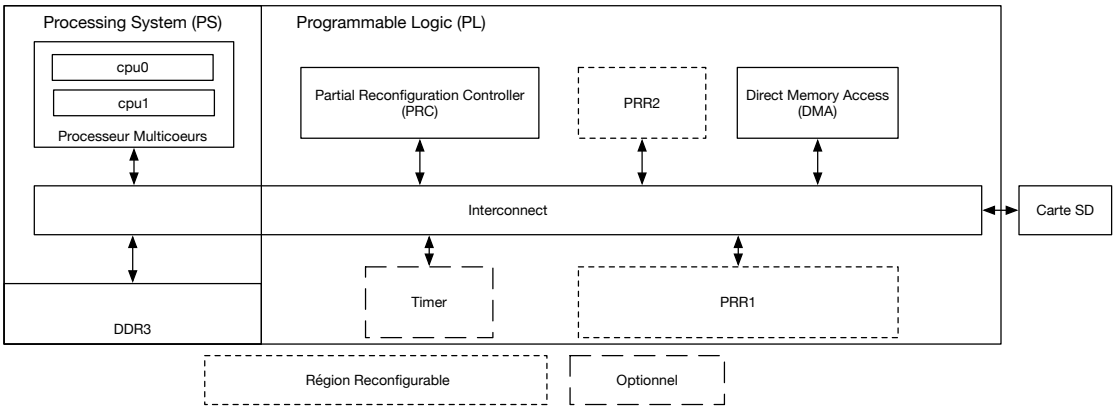


FIGURE 2.1 – Architecture du *SoC FPGA* utilisée pour le développement de Lynq

Le *PRC* personnalisé est inséré dans l'architecture de base et connecté au bus partagé. Il est ensuite directement contrôlé par l'utilisateur final à l'aide de scripts Lua.

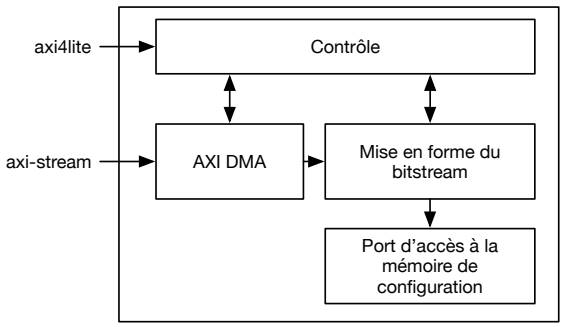


FIGURE 2.2 – Contrôleur de Reconfiguration Dynamique Partielle (PRC)

Notre environnement d'exécution cible actuellement les *SoC FPGA*s Xilinx de la série Zynq-7000. Son développement a tenu compte de deux éléments : proposer une couche d'abstraction *HAL* (Hardware Abstraction Layer) entre le domaine applicatif et le bas niveau matériel (figure 2.3), proposer une bibliothèque permettant l'accès unifié aux ressources logicielles du *PS* et matérielles de la *PL* depuis un langage de haut niveau. Ce firmware est basé sur trois couches logicielles :

- Le Board Support Package (*BSP*) qui contient les drivers et les bibliothèques de bas niveau ;
- Un compilateur Just-In-Time (Lua*JIT*) ;

- Une bibliothèque Lua suivant une approche modulaire.

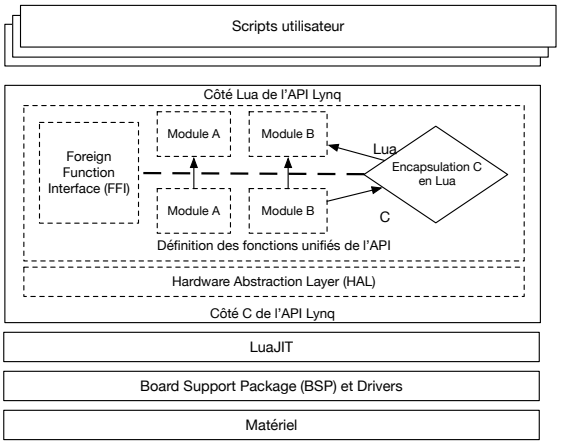


FIGURE 2.3 – Couches logicielles de Lynq

La figure 2.3 expose ces trois couches logicielles citées. La prochaine sous-section 2.2.2 ainsi que les sections 2.3 et 2.4 en exposent les détails.

2.2.2 Lua*JIT*

Lua

Pour créer une interface efficace entre le logiciel et le matériel, nous utilisons Lua [127], un langage de script léger, portable et pouvant être facilement embarqué. Celui-ci supporte les programmations procédurales, orientées objets, fonctionnelles, orientées données et la description de données. Le langage combine la syntaxe procédurale avec des constructions de description de données basées sur des tableaux associatifs et une sémantique extensible. Lua est également typé dynamiquement ; il fonctionne en compilant et interprétant le bytecode avec une machine virtuelle à base de registres (VM). De plus, il dispose d'une gestion automatique de la mémoire avec une collecte incrémentielle des déchets mémoires. Du simple interpréteur embarqué, dans un système d'exploitation ciblant les systèmes embarqués [135] jusqu'au calcul scientifique [136], Lua est devenu un langage largement utilisé depuis son introduction en 1993.

Lua*JIT*

En principe, l'interprétation d'un programme Lua se fait en lisant et décodant le bytecode et en exécutant les instructions assembleur correspondantes. Pour améliorer le temps d'exécution et les performances de ce langage, Pall [128] a proposé Lua*JIT*. Il s'agit d'un compilateur *JIT* pour le langage de programmation Lua dont le principe est de

transformer le bytecode à la volée en instructions assembleur qui peuvent être exécutées sur le *CPU* cible.

Lua*JIT* est un compilateur *JIT* utilisant les traces d'exécution. Il fonctionne en découvrant, identifiant et optimisant des séquences linéaires de bytecodes (traces) pendant l'exécution. Pendant celle-ci, l'interpréteur Lua*JIT* enregistre les informations d'exécution dans le bytecode Lua*JIT* et une représentation intermédiaire (IR pour Intermediate Representation) est émise par ce bytecode. La phase de compilation de l'exécution du *JIT* effectue des traductions spécifiques à l'architecture pour générer du code machine à partir de l'IR préalablement enregistrée. Puis, d'autres itérations exécutent le code compilé en utilisant un tableau de pointeur vers le code assembleur généré ; celui-ci ayant pour indice les instructions de bytecode qu'ils implémentent. Lorsque Lua*JIT* ne peut pas gérer un morceau de code spécifique, il revient à l'interpréteur optimisé pour l'architecture cible. Ceci implique une transition entre deux modes d'exécution : non optimisé (interprété) et optimisé (jitted). Pour résoudre ce problème, Lua*JIT* dispose d'une table de correspondance entre l'emplacement dans la pile d'une instruction et son instruction IR à correspondance unique SSA (Static Single Assignment) pour chaque instruction devant être exécutée. Cette table est utilisée pour restaurer l'état de la pile d'exécution si nécessaire.

Lua*JIT* surpasse la plupart des langages interprétés en matière de performance calculatoire pour l'exécution de code interprété ou compilé à la volée [137]. Notre étude va également dans ce sens.

Invalidation du cache instruction

Lua et Lua*JIT* n'ont pas été initialement développé pour fonctionner en standalone mais en temps qu'applicatif s'exécutant sur un système d'exploitation Windows, Linux, BDS, OSX ou POSIX OS. Plus spécifiquement, la bibliothèque C associée à Lua et Lua*JIT* effectuent des appels aux services des systèmes d'exploitation hôtes. L'utilisation de Lua et Lua*JIT* sans système d'exploitation a donc nécessité la réimplémentation des appels système nécessaires pour que les programmes fonctionnent correctement. La gestion d'un compilateur JIT implique une gestion du cache d'instruction afin que les instructions compilées à la volée puissent être directement exécutées. Ceci est permis par l'utilisation d'appel système. Nous avons été confrontés à des problèmes relatifs à la gestion des caches lors de l'utilisation de Lua*JIT* qui nous ont amené à devoir implémenter des appels système, non présent initialement sans l'utilisation d'un système d'exploitation. Sans ces appels systèmes, après chaque compilation *JIT*, le cache d'instruction n'était pas mis à jour. Ceci ne permettait donc pas au *CPU* d'exécuter l'instruction nouvellement

compilé. À la place, les instructions précédemment en cache étaient éxécutées.

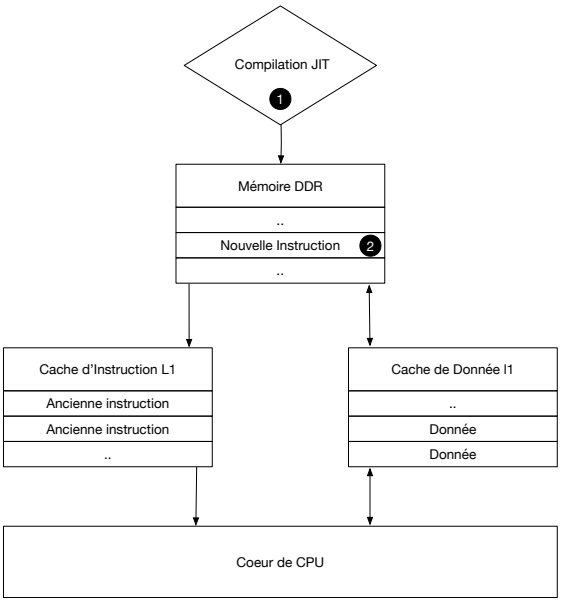


FIGURE 2.4 – Sans invalidation du cache d'instruction

Pour pallier ce problème, nous avons proposé des modifications au code source de Lua*JIT* afin de supporter l'exécution de Lua*JIT* sans système d'exploitation sur architecture ARM Cortex A9 en 32 bit. En raison des problèmes de privilège dans le mode utilisateur, l'appel aux fonctions d'invalidations et de flush du cache du *CPU* ARM doit être effectué via un appel système. Un lien vers des fonctions externes a été créé dans la bibliothèque Lua*JIT* et les fonctions ont été implémentées dans le code source de Lynq à partir des fonctions de bas niveau contenues dans le *BSP* du *FPGA* ciblé. Une description du phénomène relatif au manque d'invalidation du cache d'instruction peut être fait à partir des figures 2.4 et 2.5.

En se basant sur la figure 2.4, l'étape ❶ consiste en l'action d'utiliser le compilateur *JIT* sur un script Lua. Puis, le résultat de cette compilation est sauvegardé dans la mémoire principale à l'étape ❷.
Dans cette configuration sans appel système, les intructions générées ne peuvent pas être déplacées vers le cache d'instruction et donc bénéficier du gain de vitesse lors de l'exécution sans invalidation du cache d'instruction. La conséquence de ce phénomène est l'exécution des anciennes instructions contenues par le cache d'instruction.
Au contraire, basée sur la figure 2.5, la plage adresse des instructions qui ne seront plus exécutées est invalidée à l'étape ❸ par invalidation du cache d'instruction. Cela permet aux instructions nouvellement générées d'y accéder et d'être exécutées à l'étape ❹.
Comme pour l'invalidation de cache, les tâches habituellement effectuées par le système

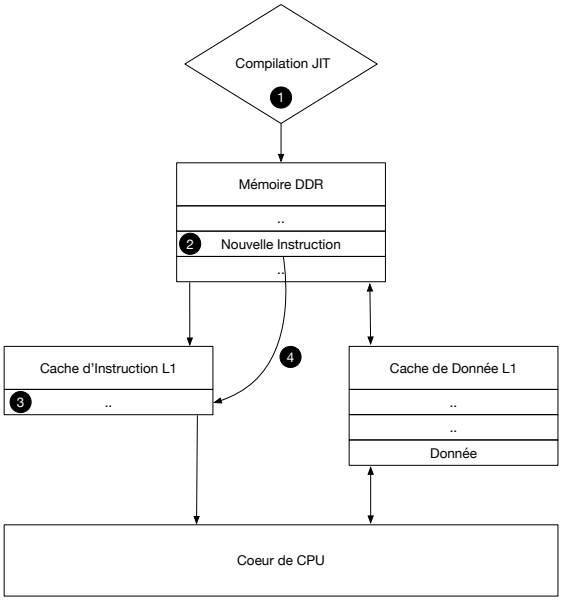


FIGURE 2.5 – Avec invalidation du cache d'instruction

d'exploitation sont réalisées via l'utilisation d'appels système. De ce fait, pour pouvoir utiliser Lua*JIT*, l'accès à un tableau contenant toutes les adresses des symboles relatifs aux appels système est nécessaire. Plus précisément, il s'agit d'être en mesure d'accéder à un tableau de symboles contenu dans la bibliothèque Lua*JIT* depuis l'extérieur de celle-ci, via l'utilisation d'appel système.

2.3 Génération de l'environnement / Pré-exécution

2.3.1 Board Support Package (*BSP*)

Le *BSP* contient tous les drivers bas niveaux, ainsi que les bibliothèques permettant d'accéder aux services offerts par le *FPGA* cible pour l'architecture développée. Pour le générer, les outils propriétaires se basent sur un fichier recensant la totalité des blocs contenus dans l'architecture ainsi que les informations relatives à leurs interconnexions. À partir de ce fichier ("hardware handoff"), le *SDK* génère un *BSP*, suite à quoi il est possible d'ajouter les bibliothèques logicielles pour accéder aux services requis par l'architecture cible. Comme exposé en figure 2.3, le *BSP* de Lynq s'appuie sur deux bibliothèques logicielles. La première est FatFS et permet la manipulation de fichiers et la seconde, lwIP, offre une interface réseau. Ces bibliothèques remplacent les services fournis par le système d'exploitation quand l'exécution n'est pas en standalone. Lynq pouvant fonctionner sans système d'exploitation, il doit donc les fournir.

La manipulation et la gestion d'une arborescence de fichiers est une des opérations basiques qu'un système d'exploitation fournit. En effet, il permet de stocker et récupérer des données en mémoire non volatile tout en offrant une arborescence pour leur classement. Même s'il a été modifié pour fonctionner sans système d'exploitation, Lua*JIT* utilise de manière intensive la manipulation de fichiers à partir de la bibliothèque standard d'entrée-sortie, notamment pour charger et exécuter des scripts et gèrer les bibliothèques utilisateurs. Sans la possibilité d'accéder au système de stockage, Lynq serait limité à une sous partie des possibilités offertes par Lua*JIT*. Pour contourner cette limitation, tous les appels système ont été implémentés dans la bibliothèque standard C (libc) du *BSP* de Lynq en utilisant un système de fichier de type File Allocation Table (*FAT*). Pour cela, nous avons utilisé la bibliothèque FatFS, une implémentation *FAT* générique écrite en C qui est optimisée pour les systèmes embarqués. La bibliothèque FatFS a été utilisée pour permettre l'utilisation d'entrées/sorties génériques. En effet, dans un premier temps les drivers donnant l'accès aux périphériques de type carte SD étaient fournis par le *BSP*. Dans un second temps FatFS était la couche logicielle qui venait les utiliser.

Des services réseaux basiques sont aussi fournis par Lynq. Pour cela, lwIP, une pile logicielle TCP/IP libre développée pour les systèmes embarqués a été utilisée. Le principal avantage de cette bibliothèque logicielle est qu'elle se concentre sur la réduction des ressources utilisées par le système tout en proposant une pile réseau complète. Comme pour l'utilisation de FatFS, lwIP utilise les drivers de bas niveau fournis par le *BSP*. Ici, la bibliothèque réseau se base sur le driver Ethernet Media Access Controller (*EMAC*). Les protocoles TCP et UDP sont implémentés comme module Lua dans l'*API* (plus de détails seront donnés en section 2.4.2). Ceci permet l'accès aux différents services réseau pour toutes les applications Lua, à partir du moment où elles ont chargé et initialisé le module de la bibliothèque afférent.

Se basant sur le bas niveau énoncé jusqu'à présent, la prochaine partie expose le processus de génération d'une image de Lynq.

2.3.2 Génération de l'image

Lynq démarre depuis une carte SD. La figure 2.6 décrit le processus d'intégration de Lynq avec les outils propriétaires du *FPGA* cible, ainsi que la génération de l'image de démarrage pour la carte SD.

D'après la figure 2.6, l'étape ❶ consiste à compiler le code source modifié de Lua*JIT*. Cela ce fait à l'aide d'une chaine de compilation ARM afin de générer une bibliothèque statique compatible avec le *SoC FPGA* cible. Cette version modifiée permet l'utilisation

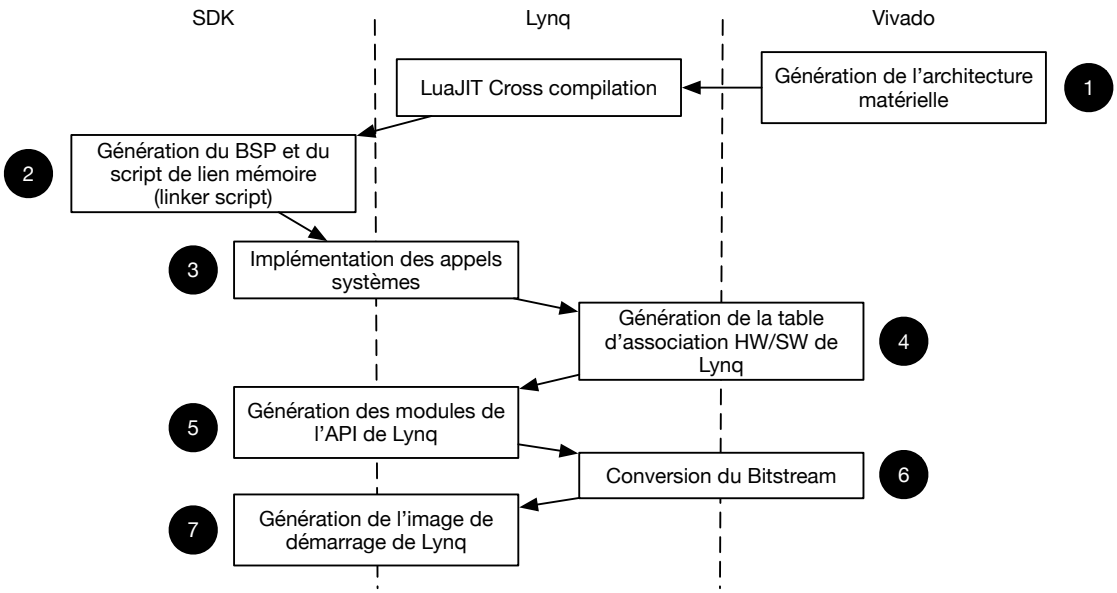


FIGURE 2.6 – Étape de génération de Lynq

des fonctions d'invalidation de cache exposées en section 2.2.2 et l'accès aux appels système via l'utilisation d'un tableau de symboles. En parallèle, l'architecture matérielle (présentée en section 2.1 est décrite en utilisant le logiciel de développement associé au *FPGA* cible. Le *BSP* contient les drivers bas niveau et les bibliothèques qui sont des prérequis pour le fonctionnement de Lynq. Celui-ci est généré en fonction du *FPGA* cible, de même que le fichier en charge de la description des espaces mémoires alloués. Il s'agit de l'étape ❷.
L'étape ❸ consiste en l'implémentation des appels système. L'étape ❹ consiste en l'utilisation d'un script qui extrait du fichier "hardware handoff" une table de liaison HW/SW en Lua. Celle-ci répertorie tous les blocs et plages adresses présents dans l'architecture. Charger ce fichier Lua dans l'applicatif conduit à ce que les fonctions accédent au matériel de bas niveau via l'utilisation des variables de la table de mapping. L'étape ❺ est le développement de la bibliothèque Lua de Lynq. Celle-ci sera présentée section 2.4.2. L'étape ❻ consiste en la conversion de tous les bitstreams du type de fichier de format *.bit* vers un format *.bin* qui seront utilisés dans le *SoC FPGA*. La conversion est nécessaire afin que le bitstream à l'entrée du *PRC* soit au bon format pour pouvoir être reconfiguré.

Notons que les étapes ❹, ❺ et ❻ peuvent être effectuées sans ordre particulier.

Enfin, à partir de tous ces éléments, nous avons été en mesure de générer l'image de démarrage de Lynq et d'y associer les fichiers de base pour son bon fonctionnement. Notre image de base peut contenir ou non un bitstream pour le *FPGA*. De ce fait, il

est possible de générer une image de démarrage permettant le bon fonctionnement du *CPU* présent sur le *SoC FPGA*, sans initialiser le *FPGA*.

2.4 Utilisation de l'environnement : Exécution

2.4.1 Séquence de démarrage

La séquence de démarrage est contrôlée par le *CPU* ARM en suivant un séquencement contenu en mémoire ROM. Celle-ci est exécutée sur le premier coeur du *CPU*. La ROM de démarrage lit l'en-tête contenue dans l'image de démarrage sur la carte *SD* pour déterminer la séquence à appliquer. Il s'en suit l'exécution du code contenu dans la ROM et le chargement du *FSBL* (First Stage Boot Loader). Ce dernier initialise la partie *CPU* du *SoC FPGA* avec des données générées par les outils Xilinx, notamment les allocations mémoires. Il charge ensuite un bitstream dans *PL*. L'environnement d'exécution Lynq est enfin chargé en mémoire RAM et prend la main sur le fonctionnemet du *SoC FPGA*. Après cette étape, Lynq est en mesure de compiler ou interpréter du code en utilisant Lua*JIT* et peut commencer l'exécution de scripts Lua. La figure 2.7 expose cette séquence de démarrage.

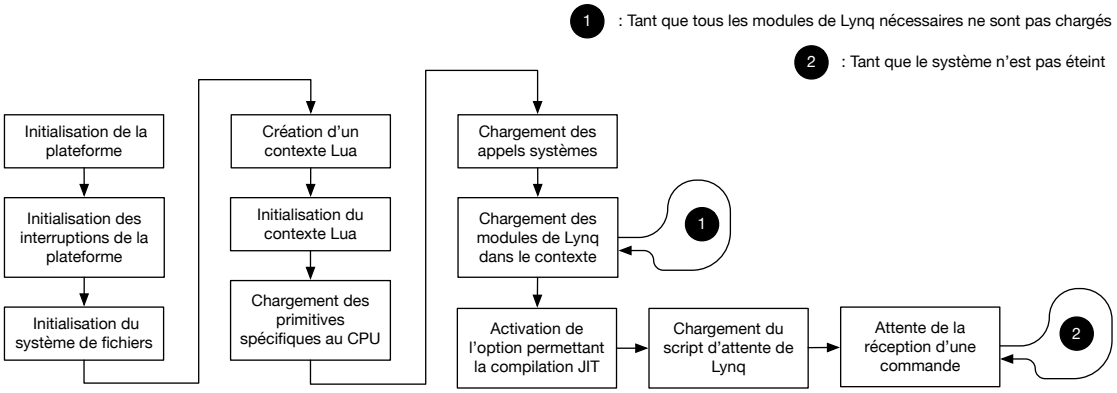


FIGURE 2.7 – Séquence de démarrage de Lynq

2.4.2 Présentation de l'*API*

Une contribution principale de Lynq est la mise à disposition d'une bibliothèque Lua qui servira d'interface de haut niveau pour l'utilisateur final. Elle permet également la gestion de l'environnement d'exécution offert par le *SoC FPGA*, que ce soit la partie *PS* ou *PL*. La bibliothèque abstrait les détails de bas niveau, tels que les transactions mémoires ou l'accès aux *MMIO*. Cette bibliothèque est écrite en C et en Lua, et elle offre des fonctions et méthodes en Lua. Lua étant un langage orienté objet, chaque

module chargé dans l'environnement permet la création d'objets sur lequel nous pouvons appliquer des méthodes. Prenons l'exemple d'un GPIO du *SoC FPGA*. Lorsqu'un GPIO doit être utilisé, par exemple un bouton poussoir, un objet associé à ce bouton poussoir est créé. Durant sa création, celui-ci est déterminé à partir des identifiants et adresses extraits de la table de liaison HW/SW. Une fois créé, il est possible de le manipuler en récupérant par exemple sa valeur via une méthode de lecture appliquée à l'objet.
La figure 2.8 décrit la bibliothèque de Lynq. Elle se lit de gauche à droite (la gauche représentant la couche la plus basse et la droite la couche la plus haute). En se basant sur le *BSP* contenant tous les drivers de bas niveau relatifs à l'architecture et les bibliothèques associées aux services ainsi que le fichier de mapping Lua généré précédemment, une couche d'abstraction du matériel (*HAL*), appelé *dev* sert de couche de base à notre bibliothèque. Elle permet de regrouper les pilotes de différents *BSP* par fonctionnalité pour cibler les cartes d'autres fournisseurs avec la même *API* de haut niveau. Par cela, nous entendons que pour une fonctionnalité donné, chaque vendeur fournit ses fonctions de bas niveau. L'objectif de la couche d'agrégation (*HAL*) est donc, à partir des fonctions de bas niveau de chaque vendeur, d'unifier les fonctions de la bibliothèque par fonctionnalité. Au dessus de celle-ci, se trouve la couche *lib* qui est la plus haute et qui permet l'utilisation dans l'environnement d'exécution Lua de toutes les fonctionnalités offertes par la bibliothèque en rendant accessible les modules à l'utilisateur.

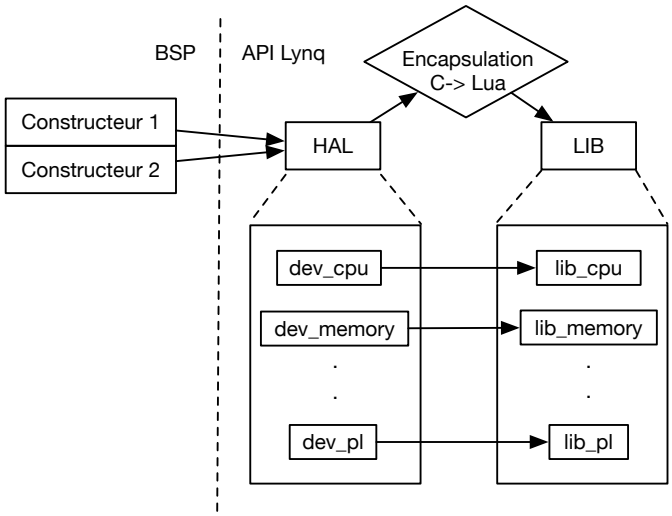


FIGURE 2.8 – Architecture de la bibliothèque Lynq

La figure 2.9 présente la vue en couches, de bas en haut, suivi durant le développement de la bibliothèque Lynq. Nous nous sommes basés sur le matériel cible ❶ pour lequel nous avons généré le *BSP* ❷. Après cela, nous avons développé le *HAL* ❸ pour agréger toutes les fonctionnalités. Puis, la dernière couche expose les fonctionnalités développées en C à l'environnement d'exécution Lua sous la forme de bibliothèques.

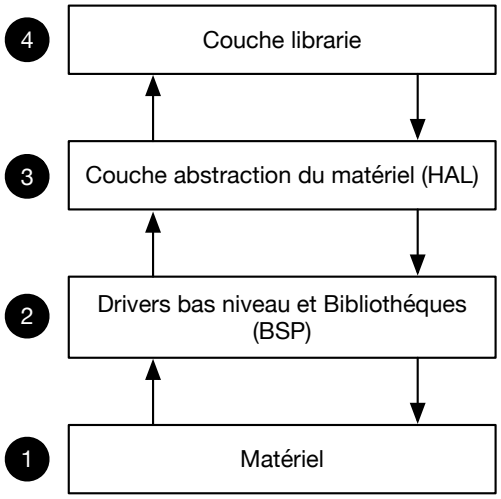


FIGURE 2.9 – Développement de l'API Lynq suivant une logique de bas en haut

L'architecture de la bibliothèque a été pensée de manière à ce que la couche de haut niveau suive une logique modulaire. L'utilisateur peut donc à l'usage, ne charger que les modules dont il a besoin pour son applicatif. L'objectif est de réduire son empreinte mémoire globale. La bibliothèque gère et contrôle les transferts entre l'environnement d'exécution Lua dans le *PS* et les accélérateurs matériels (*IP*) dans la *PL*. Également, la gestion des ports d'entrées/sorties à usage général (*GPIO*), les *DMA*, les interruptions, la gestion réseau, la configuration du *FPGA* ainsi que la gestion du CPU ont également leurs propres modules. Les fonctions, méthodes et constantes sont exportées sous forme de modules chargés ultérieurement dans un script Lua. Ci-dessous sont décrits les différents modules de Lynq présents dans la version de base de Lynq.

- **CPU** - Ce module apporte les fonctionnalités de contrôle du *CPU* hardcore présent sur le *SoC FPGA*. Il permet à l'utilisateur de sélectionner le coeur sur lequel son applicatif est exécuté et d'en gérer son fonctionnement.
- **GPIO** - Ce module permet l'accès au *GPIO*, que ce soit de la partie *PS* via le ARM ou de la partie *PL* via *AXI*.
- **Timer** - Ce module apporte le support des timers du ARM pour la partie PS mais également des timers implémentables dans la partie *PL* via *AXI*.
- **DMA** - Ce module permet le contrôle des transferts *DMA* des parties *PS* et *PL* via *AXI*.
- **Socket** - Ce module offre le support des protocoles *TCP* et *UDP*.
- **Programmable logic (PL)** - Ce module permet le support des capacités de reconfiguration (partielle et totale, dynamique et statique).

L'utilisation de la version de Lynq *JIT*, permet d'accéder aux fonctionalités *FFI* via l'utilisation de Lua*JIT*. Cette fonctionnalité permet de faire appel à des fonctions C préalablement compilées. Le code C est directement intégré au code source de Lynq sous forme de bibliothèque durant sa génération. En se basant sur les figures 2.3 et 2.8, nous implémentons les fonctions FFI dans la couche dev / (*HAL*). Celles-ci n'ont pas besoin d'une interface dans la couche lib et sont mise à disposition de l'utilisateur pour le *FFI*. Contrairement aux fonctions présentes dans la bibliothèque de la couche lib, les symboles des fonctions mises à disposition via FFI doivent être ajoutés à la table de symboles internes de Lua*JIT*. En effet, elle est chargée durant la phase de démarrage et rendue accessible au moment de l'exécution via Lua*JIT*.

2.4.3 Interface de commande Lynq

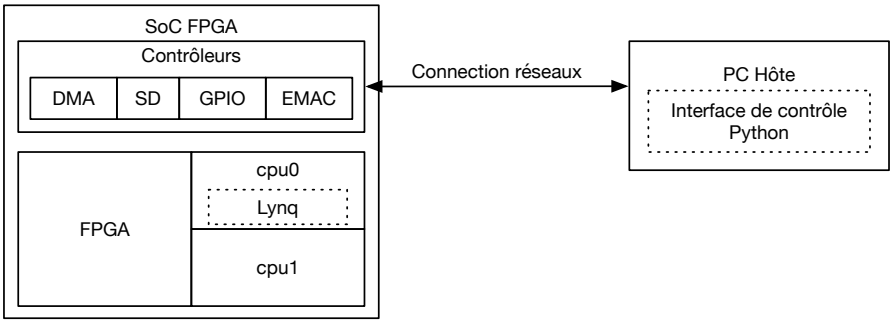


FIGURE 2.10 – Interface *SoC FPGA*- PC hôte

Afin de contrôler l'environnement d'exécution dans les phases de développement, nous avons développé en Python une interface de contrôle en ligne de commande portable compatible avec la bibliothèque Lynq. Les commandes disponibles sont présentées dans le tableau 2.3. Le but est de se connecter au *SoC FPGA* via une connection TCP et contrôler l'exécution du *SoC FPGA*(2.10). Le PC hôte via la connection réseau permet à partir de l'interface de commande en Python d'envoyer des commandes à l'environnement d'exécution Lynq qui s'exécute sur un coeur de *CPU* du *SoC FPGA*.

2.5 Performances

Les bases de notre environnement d'exécution ainsi que ses phases de génération et d'exécution présentées, nous allons dans cette section évaluer les performances de Lynq. Pour cela nous avons mené une étude portant sur les aspects utilisation de ressources matérielles, performances calculatoires, temps de démarrage, consommation énergétique et consommation mémoire.

Command	Description	Arguments
configure	Configure la partie *PL*	NOM_CIBLE, BITSTREAM
connect	Connecte le *SoC FPGA* à l'interface de commande (*TCP*)	NOM_CIBLE, ADDRESSE_IP, PORT
disconnect	Déconnecte le *SoC FPGA* de l'interface de commande	NOM_CIBLE
dow	Télécharge un fichier sur la carte SD du *SoC FPGA* Cible	NOM_CIBLE, FICHIER
run	Exécute un script Lua sur un coeur de *CPU* ARM du *SoC FPGA*	NOM_CIBLE, FILE
start	Démarre un coeur de *CPU* sur le *SoC FPGA* cible	NOM_CIBLE, CPUID
stop	Arrête un coeur de *CPU* sur le *SoC FPGA* cible	NOM_CIBLE, CPUID
reset	Redémarre le *SoC FPGA* cible	NOM_CIBLE
memread	Lit une donnée à l'adresse spécifiée	NOM_CIBLE, ADDR_LECTURE
memwrite	Ecrit une donnée à l'adresse spécifiée	NOM_CIBLE, ADDR_ECRITURE, DONNEE
repl	Lance le mode interpreteur interactif (*REPL*)	NOM_CIBLE

Tableau 2.3 – Résumé des principales commandes disponibles dans l'interface de contrôle en ligne de commande

2.5.1 Ressources utilisées par les PRRs

Deux types de *PRR* ont été utilisés dans l'architecture de base présenté en section 2.2.1 Chaque *PRR* est configurée afin de supporter 1) une région d'horloge complète du *FPGA* et, 2) une demi région d'horloge du *FPGA*. Cette approche vient compenser le temps de reconfiguration d'une *PRR* qui n'est pas dépendant de l'application cible puisqu'il est constant et proportionnel à la taille de la *PRR*. Ainsi, la reconfiguration de la *PRR*2 sera privilégiée si le bitstream pour un accélérateur A est disponible pour la *PRR*1 et pour la *PRR*2 afin que la reconfiguration soit plus rapide. Les ressources disponibles dans chaque *PRR* sont présentées dans le tableau 2.4. Cette approche est confortée par son utilisation dans les travaux de Goeder et al. [56].

	LUT	DFF	BRAM18k	DSP
Capacité du *FPGA*	53200	106400	140	220
Architecture de base (Partie statique)	6020 (11.3%)	7778(7.3%)	10(7,1%)	0
PRR1	10400 (19.5%)	20800(19.5%)	60 (43%)	40(18.2%)
PRR2	4800 (9%)	9600(9%)	40(28.6%)	40(18.2%)
Multiplication matricielle 32x32	1304	804	3	0
FFT 1024 pts	1540	2369	7	9

Tableau 2.4 – Ressources matérielles disponibles et utilisées pour la reconfiguration dynamique partielle sur une cible *SoC FPGA*à base de Zynq-7020

Comme évoqué précédemment, la taille de la *PRR* impacte directement celle du bitstream. Elle influe donc sur le temps de reconfiguration. Le tableau 2.4 expose, pour un Zynq-7020, une utilisation de 10 à 15 % des ressources disponibles dans la *PRR*1. Cette utilisation est doublée si elle est mise en place dans la *PRR*2. Cependant, la taille du bitstream de *PRR*1 étant de 1237kB et celle de *PRR*2 étant de 635kB, il est plus rapide de reconfigurer cette dernière comme le démontre le tableau 2.5, cela même si son

pourcentage d'utilisation est supérieur à celui de $PRR1$. À titre d'exemple, les ressources utilisées par deux implémentations, respectivement une multiplication matricielle 32x32 et un calcul de FFT 1024 points, ont été mesurées et toutes deux peuvent être implémentées dans la PRR2.

	PRC	PCAP
PRR1	3.1 ms	9.7 ms
PRR2	1.6 ms	5 ms

Tableau 2.5 – Temps de reconfiguration des PRR en millisecondes **[ms]** en utilisant notre Contrôleur de Reconfiguration (PRC)

2.5.2 Performance calculatoire

Pour comprendre et évaluer les avantages de l'utilisation de LuaJIT par rapport aux solutions existantes, la suite de benchmarks SciMark 2.0 [138] a été utilisée. Il s'agit d'une bibliothèque de test qui mesure la performance des codes logiciels utilisés dans les applications scientifiques. Ce kernel donne une indication sur la performance de la Machine Virtuelle (VM) ou du compilateur (JIT) sous-jacent de LuaJIT. SciMark expose ses résultats en MFlops. Certains benchmarks exercent des fonctions transcendantes (non algébriques comme le calcul d'un sinus ou d'un cosinus), ainsi que des opérations entières. Cela induit un calcul de performance calculatoire en MFlops approximatif. Toutefois, le même nombre de MFlops est utilisé de façon constante pour que les comparaisons soient valides. La taille des problèmes est également réduite afin d'isoler les effets relatifs à la hiérarchie mémoire et se concentrer sur les mécanismes internes de VM/JIT et de CPU. SciMark comporte cinq kernels que sont :

- La Fast Fourier Transform (FFT) / Transformée de Fourier Rapide effectue une transformée unidimentionnelle de nombres complexes. Ce kernel utilise des fonctions complexes d'arithmétique, de mélange, de mémoire et de trigonométrie.
- Jacobi Successive Over-Relaxation (SOR) résout des équations de Laplace en 2D avec des conditions aux limites de Dirichlet sur une matrice. Il permet d'exercer des schémas d'accès mémoire pour calculer des moyennes de matrice.
- L'intégration de Monte Carlo (MC) est une méthode de calcul de Pi qui utilise une approximation basée sur le calcul de l'intégrale d'un quart de cercle. Ce kernel utilise des générateurs de nombres aléatoires, des appels de fonctions synchronisés et des inlining de fonctions.
- Sparse Matrix Multiply (SPARSE) utilise une matrice dans un format compressé

avec une structure à faible densité. Ce kernel exerce un adressage indirect et des références de mémoire non régulières.

- Lower-Upper matrix factorization (LU) calcule la factorisation d'une matrice dense à l'aide d'un pivotement partiel. Ce kernel exerce des fonctions d'algèbre linéaire (BLAS pour Basic Linear Algebra Subroutines) sur une matrice dense.

Chaque kernel a été implémenté en C, Python et Lua. Pour fournir une comparaison équitable, le temps d'exécution de Lua et Lua*JIT* avec Python interprété et PyPy *JIT*ted est exposé. Les kernels C ont été compilés avec ARM GCC 6.2.1, avec et sans optimisation du compilateur (O0 et O3). Les kernels Python ont été implémentés dans Python 3.4 et PyPy 5.8. L'ensemble des kernels de Python 2.7 ont également été implémentés. Les kernels Lua ont été implémentés dans Lua 5.1.5 et LuaJIT 2.0.5 sans aucune optimisation du compilateur. Ils ont été exécutés en mode interprété ou JITted.
La fonctionnalité FFI a permis d'évaluer l'impact de l'allocation et de la manipulation de mémoire de bas niveau sur le temps d'exécution. L'exécutable Lynq a été compilé avec GCC 6.2.1 sans optimisation du compilateur.

Langage	Spécification	FFT	SOR	MC	SPARSE	LU
Python	Python 3.4	0.35	0.095	0.135	0.61	0.097
	PyPy 5.8.0	12.41	34.85	8.54	11.47	17.33
C	GCC 6.2.1 -o0	25.79	71.34	12.17	26.66	44.04
	GCC 6.2.1 -o3	79.61	165.77	54.42	75.59	111.64
Lua (Notre travail)	Lua 5.1.5	1.73	3.96	1.05	2.38	2.78
	LuaJIT 2.0.5	3.98	10.27	2.49	6.49	8.74
	LuaJIT 2.0.5 (jit)	47.60	128.29	28.97	22.42	67.93
	LuaJIT 2.0.5 (jit, ffi)	58.91	140.50	41.55	36.63	84.50

Tableau 2.6 – Performance de calcul en [**MFlops**]

L'ensemble des résultats présentés est fourni à partir d'implémentations logicielles exécutées dans un seul processus logiciel de sorte qu'un seul coeur de CPU ARM a été utilisé. PYNQ et notre système ont été démarrés à partir d'une carte SD de classe 10 de 8 Go. Pour les expérimentations avec PYNQ, aucun module hyperviseur n'a été chargé et toutes les fonctions de gestion de l'alimentation ont été désactivées. Tout le code exécutable a été mis en cache en mémoire avant chaque mesure.

Les résultats présentés dans cette section donnent les performances des implémentations logicielles et de la VM. Ils donnent un aperçu de la façon dont les implémentations des VM Python et Lua fonctionnent par rapport à d'autres approches.

Le tableau 2.6 expose les résultats de calcul en MFlops des kernels de référence SciMark

(FFT, SOR, MC, SPARSE et LU) dans les environnements PYNQ et Lynq utilisant des implémentations C, Python et Lua. En commençant par comparer les interpréteurs Python 3.4 et Lua 5.1.5, il apparait que Lua est 3.9 fois à 41.7 fois plus rapide. La comparaison entre Python 3.4 et l'interpréteur LuaJIT 2.0.5 permet ensuite d'accentuer les différences entre les implémentations de machines virtuelles Python et Lua (10.6 fois à 108.1 fois plus rapides). Lorsqu'il est activé, le mode LuaJIT JITted exécute 36.8 fois à 1350.4 fois plus rapidement. Lors de l'utilisation de FFI, LuaJIT pousse l'interpréteur Python à sa limite de performance. La performance brute de calcul est de 60 fois à 1478.9 fois plus rapide que Python. Dans ce cas, les résultats du temps d'exécution des implémentations Lua JITted sont comparés avec les implémentations interprétées en Python, ce qui augmente l'écart de performances. Pour faire une comparaison plus juste, la performance de PyPy 5.8.0 est comparée à celle de LuaJIT avec FFI. La performance est de 3.2 fois à 4.9 fois meilleure en faveur de LuaJIT.

LuaJIT avec FFI est aussi 1.4 fois à 3.4 fois plus rapide que les implémentations C créées avec l'optimisation du compilateur O0. Cependant, comme prévu lors de l'utilisation de GCC avec l'optimisation du compilateur O3, C surpasse LuaJIT par un facteur de 1.2 fois à 2.1 fois. Les options -O0 et O3 indiquent au compilateur le niveau d'optimisation qu'il peut utiliser, l'option -mthumb quant à elle (qui sera utilisée plus loin dans cette section) indique au compilateur qu'il peut également utiliser des instructions sur 16 bits lorsque cela est possible.

Les résultats sont aussi comparés du point de vue de l'efficacité énergétique. Ainsi, comme indiqué au bas du tableau 2.7, les performances de LuaJIT sont en moyenne 2.5 fois à 4.8 fois meilleures que PyPy sur l'ensemble de la bibliothèque de test. Le choix de LuaJIT pour l'implémentation de Lynq présente des avantages significatifs en termes de performances calculatoire et d'énergie par rapport à l'interpréteur Python du PYNQ et des améliorations par rapport à PyPy. Les performances Lua obtenues avec JIT et FFI activées tirent profit des capacités propres de LuaJIT qui intégre une utilisation de FFI pour les transferts de mémoire. L'objectif est d'améliorer les performances de calcul. Celles-ci peuvent toutefois être améliorées avec JIT et FFI activés. Pour cela, du code C est directement appelé via des appels FFI depuis Lua. Le tableau 2.8 en expose les détails en performances.

La différence de performance entre LuaJIT 2.0.5 (jit) et LuaJIT 2.0.5 (jit, ffi) s'explique par le fait que, lorsque FFI est activée, même si LuaJIT exécute un script Lua, toutes les structures de données et les allocations de mémoire de l'exécutable sont optimisées avec l'option FFI. Ceci implique une accélération du transfert de donnée et une augmentation des performances. Cependant, il est possible d'obtenir de meilleures performances en

language	specifications	FFT	SOR	MC	SPARSE	LU
Python	Python 3.4	1.71	0.50	0.71	3.14	0.51
	PyPy 5.8.0	69.18	310.86	101.73	64.84	193.92
C	GCC 6.2.1 -o0	160.92	405.68	74.70	151.98	262.48
	GCC 6.2.1 -o3	529.80	1199.54	306.39	440.31	722.64
Lua (Notre travail)	Lua 5.1.5	10.02	20.46	5.26	12.63	14.94
	LuaJIT 2.0.5	21.51	56.87	13.90	33.01	42.99
	LuaJIT 2.0.5 (jit)	255.85	719.63	168.28	133.30	359.73
	LuaJIT 2.0.5 (jit, ffi)	330.26	893.39	249.28	203.89	476.32

Tableau 2.7 – Performances de calcul en Megaflops par Watt [**MFlops/W**]

	FFT	SOR	MC	SPARSE	LU
Code C compilé avec GCC 6.2.1 -O0 utilisant LuaJIT 2.0.5 (jit, ffi)	23.85	67.21	12.09	25.64	41.30
Code C compilé avec GCC 6.2.1 -O0	25.79	71.34	12.17	26.66	44.04
Code Lua compilé avec LuaJIT 2.0.5 (jit, ffi)	58.91	140.50	41.55	36.63	84.50
Code C compilé avec 6.2.1 -O3 utilisant LuaJIT 2.0.5 (jit, ffi)	79.57	165.55	51.28	69.55	110.42
Code C compilé avec 6.2.1 -O3	79.61	165.77	54.42	75.59	111.64

Tableau 2.8 – Comparaison en [**MFlops**] entre l'exécution LuaJIT d'un script en Lua avec l'option FFI activé et l'exécution d'un script faisant des appels en C avec les mêmes options.

utilisant la fonctionnalité FFI sur l'ensemble de la bibliothèque SciMark. Cela revient à décrire chaque kernel en fonction C et les appeler dans des scripts Lua. Le tableau 2.8 montre une comparaison entre un script Lua avec l'algorithme écrit en Lua utilisant LuaJIT avec les options FFI et JIT activées et un script Lua avec la même option activée mais utilisant directement par FFI des fonctions écrites en C. La comparaison a été faite avec les résultats précédents compilés avec GCC comprenant différents niveaux d'optimisation et notre meilleure configuration pour Lua (options JIT et FFI activées). Le code Lua utilisant LuaJIT 2.0.5 (jit, ffi) a été pris comme point de référence. Cette configuration est comparée à un code C pur exécuté directement en C avec LuaJIT via FFI. Comme prévu, le code C compilé avec l'optimisation -O0 et exécuté directement ou via LuaJIT, obtient de moins bonnes performances que notre référence. Le code C compilé avec l'optimisation -O3 réalise quant à lui les meilleures performances. Le code Lua qui utilise LuaJIT 2.0.5 (jit, ffi) atteint entre 92% et 99,9% des performances obtenues via l'exécution de code C avec l'optimisation -O3. Les performances de LuaJIT par rapport au C non optimisé exposés auparavant sont réduites. Cette expérience appui les performances de LuaJIT par rapport à celles de PyPy sur PYNQ en augmentant la plage d'accélération entre 4.75 fois et 6.4 fois par rapport à celle qui est exposée précédement (de 3.2 fois à 4.9 fois).

L'évaluation des performances calculatoire de Lynq comparée à des exécutions en Python ou C confirme la pertinence de l'utilisation de Lua dans un environnement d'exécution sans système d'exploitation. La prochaine partie continuera notre étude en exposant une évaluation du temps de démarrage de notre solution ainsi que la consommation énergétique associée.

2.5.3 Temps de démarrage et consommation énergétique

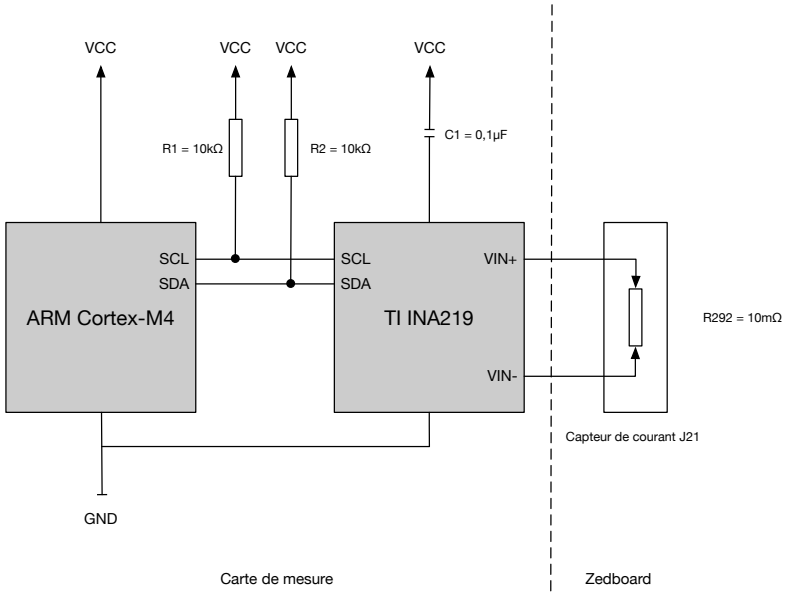


FIGURE 2.11 – Circuit de mesure de la consommation de la Zebboard

Le circuit de mesure de consommation présenté en figure 2.11 comprend un *CPU* ARM Cortex-M4 qui collecte les données de courant et de puissance surveillées par un composant Texas Instrument INA219. Les mesures ont été effectuées avec une fréquence d'échantillonnage de 1 kHz et à température ambiante.
La figure 2.12 montre deux graphiques Lynq et PYNQ qui représentent la consommation d'énergie de la ZedBoard pendant le processus de démarrage de PYNQ et de Lynq. La phase de démarrage du PYNQ (1a) comprend deux étapes qui sont le chargement du kernel Linux et le chargement de l'espace utilisateur (14,1 secondes). Il s'en suit une phase de repos (1b) du *CPU* ARM après avoir terminé l'exécution de la séquence de démarrage. Comme il fonctionne sans sytème d'exploitation, Lynq peut démarrer en 22 msec (2a) lorsqu'aucune connection réseau n'est requise et que le *FPGA* n'est pas configuré. À noter que dans sa version initialisant seulement la bibliothèque de base de Lynq et ne chargeant aucun module, le temps de démarrage descend jusqu'à 10 ms. Au repos (2b), l'environnement Lynq affiche une consommation moyenne de 3,4 W, soit 0,1 W de plus que le PYNQ au repos. Dans cette expérience, LuaJIT exécute un script Lua

implémentant une structure de contrôle qui fonctionne à l'infini sans effectuer d'opération. PYNQ exécute Linux et lorsqu'il n'y a pas de processus à exécuter sur le *CPU* ARM, une tâche inactive est programmée par l'OS pour réduire la consommation d'énergie. Le coeur de *CPU* est arrêté jusqu'à ce qu'une interruption se produise.

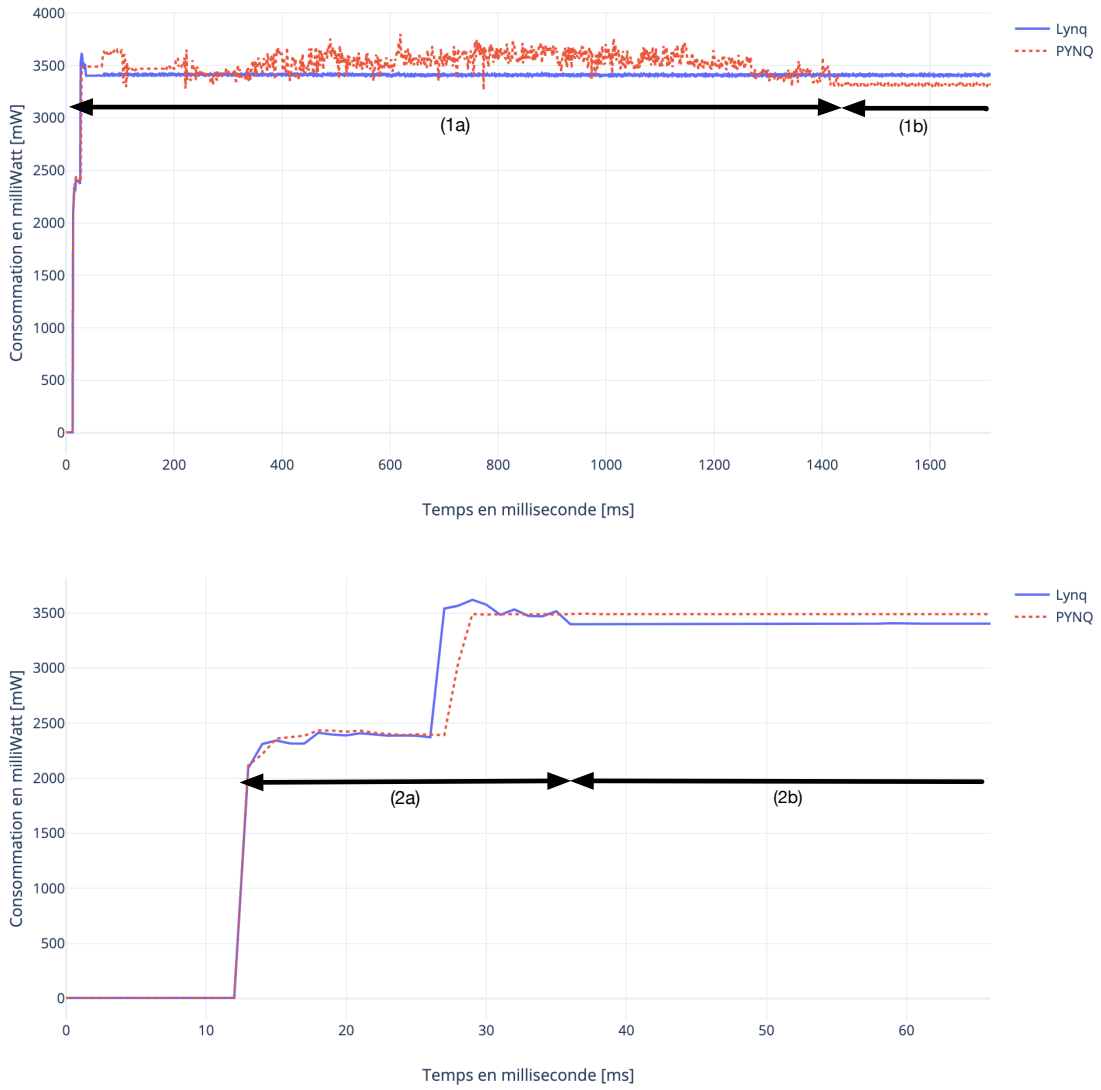


FIGURE 2.12 – Consommation énergétique de la Zedboard durant la phase de démarrage de PYNQ et de Lynq

2.5.4 Consommation mémoire

Une image de Lynq a une taille de 800kB lorsqu'elle a été compilé avec les options -O3 et -mthumb. Cette taille ne tient pas compte de celle du bitstream total du *FPGA* qui peut être chargé à posteriori. Alors qu'il est de 3.86MB pour un Zynq-7020, il est de 5.7MB

pour un Zynq-7030.

Durant son fonctionnement, nous avons mesuré la consommation mémoire de Lynq pendant l'exécution des différents kernels de la bibliothèque de test SciMark. Celle-ci a été comparé avec les kernels exécutés en Python sur la carte PYNQ. Les résultats sont présentés dans le tableau 2.9. La consommation mémoire moyenne de LuaJIT est de 1.15 fois à 7.02 fois inférieure (moyenne 3.12 fois).

	FFT	SOR	MC	SPARSE	LU
Python	113.21	89.84	191.30	11.72	585.92
LuaJIT	16.13	78.45	81.59	2.33	163.56

Tableau 2.9 – Consommation mémoire en **[kB]** pour Scimark

La consommation mémoire est mesurée en kB. Afin d'avoir des mesures équitables, nous avons soustrait les consommations mémoires statiques relatives aux fonctionnement des VM Python et Lua. Nous constatons une consommation de mémoire de 1.15 fois à 7.02 fois supérieure pour l'environnement Python comparé à notre environnement Lua. L'interface de commande présentée section 2.4.3 nous a permis d'effectuer les expérimentations précédemment exposées à partir des commandes existantes. Cependant, celui-ci ne permettait pas l'exécution de code complexe ainsi que la visualisation en temps réel des résultats et des rapports de débug. En effet, l'exécution d'un applicatif nécessitant l'utilisation de plusieurs scripts était impossible via l'invite de commande. De même que la visualisation des résultats calculatoire. Pour pallier cela nous avons développé un serveur web pour Lynq.

2.6 Serveur Web Lynq

Nous avons développé un serveur web pour les systèmes embarqués. L'accent a été mis sur les *SoC FPGA*s. Celui-ci se base sur l'environnement d'exécution Lynq et se positionne comme une couche intermédiaire avec l'applicatif. Intégralement écrit en Lua, il tire parti des bénéfices de l'API Lua de Lynq. Il permet, tout en gardant une empreinte mémoire ainsi qu'une charge processeur faible, de fonctionner sans système d'exploitation et d'exécuter du code compilé ou interprété tout en visualisant les données relatives à l'exécution. Notre serveur Web nécessite moins de 100 kB de mémoire de stockage. Il a besoin de 125 kB de mémoire RAM pour fonctionner ce qui lui permet d'afficher une consommation mémoire entre 25 fois et 135 fois inférieure aux serveurs Web cités. La contrepartie à cette faible empreinte mémoire réside dans les performances relatives à son temps de réponse. En effet, notre serveur web répond en moyenne en 40 microsecondes,

ce qui est 6.25 fois à 11.1 fois plus lent que les serveurs web cités précédemment.

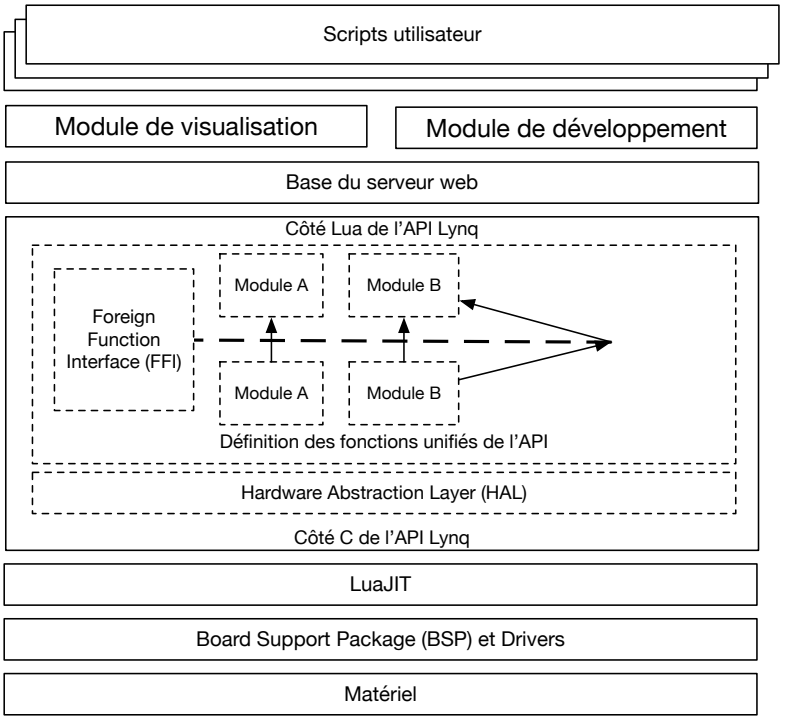


FIGURE 2.13 – Positionnement du serveur Web dans la stack de Lynq

2.6.1 Fonctionnalités principales

Cette partie expose les fonctionnalités principales offertes par le serveur web :

- **Exécution sur processeur softcore et hardcore.** Le serveur web est en mesure de fonctionner sur des processeurs softcore ou hardcore, afin de s'adapter au système embarqué cible. Cependant, les capacités d'amélioration de performances propres à LuaJIT ne peuvent être utilisées que si l'architecture du processeur cible est compatible LuaJIT.
- **Support d'architectures hétérogènes.** Une des fonctionnalités principales du serveur web est sa capacité à contrôler des ressources de nature différente depuis une interface de contrôle centralisée. Le serveur web peut en effet, depuis la même interface, se connecter à plusieurs cibles de nature différente et les administrer.

Le serveur web se base sur la stack logicielle de Lynq. Celle-ci est, comme présentée en figure 2.13, utilisée comme base. Il utilise les fonctions disponibles avec Lynq pour son fonctionnement. Le module de base implémente les fonctions réseau nécessaires à l'établissement de la connection client/serveur. Basé sur la bibliothèque lwIP du *BSP*,

des sockets web sont abstraites et rendues accessible en Lua par Lynq. Le module de développement offre une interface web permettant le codage et l'exécution de code Lua depuis une interface graphique. Enfin, le module de visualisation encapsule la bibliothèque JavaScript plotly [139] et offre des fonctions Lua permettant la visualisation de donnée en cours d'exécution.

2.6.2 Évaluation des performances

Dans cette partie les performances de consommation mémoire sont analysées ainsi que le temps de réponse à une requête de notre solution.

- **Utilisation mémoire** : Le tableau 2.10 compare la mémoire minimale nécessaire pour différents serveurs web incluant notre solution. Les quatre serveurs web présentés et comparés doivent pouvoir fonctionner sur un système d'exploitation. La plupart d'entre eux est supporté par différent système d'exploitation et tous sont en mesure de fonctionner sous Linux. Nous avons effectué nos différentes comparaisons de ces serveurs web fonctionnant sur une distribution Ubuntu 16.04 64-bit.

Nom	Notre solution	NGINX	LightTPD	Cherokee	Apache
1-Taille du binaire du serveur web	80kB	956kB	112kB	24kB	579kB
2-Taille de ces dépendances	800kB	>1MB	>1MB	>1MB	>1MB
Taille de 1+2	880kB	>1MB	>1MB	>1MB	>1MB

TABLEAU 2.10 – Comparaison de la mémoire de stockage minimale requise pour les serveurs Web

  Il est difficile d'évaluer précisément la taille des dépendances de ces solutions car celle-ci ne prennent pas seulement en compte les bibliothèques dont elle dépendent, mais aussi certains services fournis par le système d'exploitation dont la taille est difficilement quantifiable. Pour les serveurs web dans ce cas, la notation utilisée sera *>1MB*.

  La nécessité d'utiliser un système d'exploitation induit une quantité de mémoire minimale supérieure à notre solution. Cependant, ces données sont à mettre en perspective avec le fait que tous ces serveurs Web bénéficient de l'écosystème que peut offrir un système d'exploitation. Ceci permet l'accés à des bibliothèques qui permettent l'introduction de fonctionnalités complémentaires relatives notamment à la sécurité comme par exemple OpenSSL.

- **Consommation mémoire RAM** Nous avons développé un script qui mesure séparément la consommation de Lynq et du serveur web pour apporter une compa-

raison raisonnable. Les résultats sont présentés dans le tableau 2.11. Les chiffres exposés sont la représentation de la mémoire nécessaire pour faire fonctionner le serveur web dans un mode d'attente avec un client connecté. La consommation mémoire de base, composé des utilisations mémoires du système d'exploitation et de ses services de base y est soustraite.

Nom	Notre solution	NGINX	LightTPD	Cherokee	Apache
Mémoire RAM Minimum	125kB	16.8MB	3.1MB	4.7MB	13.2MB

Tableau 2.11 – Moyenne de la mémoire RAM nécessaire pour le fonctionnement des serveur Web.

Nous avons mesuré une consommation de base de 250kB pour Lynq, à ceci s'ajoute la consommation mémoire moyenne du serveur web de 125kB. Dans le tableau 2.11 les résultats comparant les différents serveurs web sont exposés. Notre approche consomme entre 25 fois et 135 fois moins de mémoire RAM. Ces résultats sont à coupler avec la figure 2.14 qui montre la consommation mémoire RAM en fonctionnement de notre serveur web.
Durant les tests conduits, la consommation RAM du serveur web était situé autour de 125kB. À cela s'ajoute la consommation mémoire de l'applicatif en cour d'exécution. Les variations de consommation mémoire RAM de cet applicatif présentées figure 2.14 viennent du ramasse-miette (GC pour Garbage collector) qui agit de manière incrémentale, c'est à dire que celui-ci tente de récolter la mémoire non utilisée en alternance avec l'exécution du programme. La consommation mémoire la plus faible relevée est de 93kB.

- **Temps de réponse moyen.** Le temps de réponse moyen à une requête d'un serveur web est un paramêtre important. Nous avons mesuré le temps moyen entre l'envoi de la requête et l'accusé de réception de notre système. Notre serveur web obtient un temps moyen de réponse de 40μs, ce qui est entre 6.25 fois et 11.1 fois plus lent que les autres serveurs web présentés tableau 2.12. Ce ralentissement vient du fait que sans système d'exploitation, les sockets doivent être implémentés manuellement et ne sont pas optimisés, ceci n'étant pas l'objectif du développement de notre serveur Web.

Nom	Our work	NGINX	LightTPD	Cherokee	Apache
Temps de réponse moyen	40μs	5.7μs	3.60μs	3.9μs	6.4μs

Tableau 2.12 – Temps de réponse moyen à une requête en [μs]

L'évaluation des consommations mémoires et temps de réponse positionnent notre serveur comme une alternative standalone au serveur Web déjà présent sur le marché pour des applications sans contrainte temps réel forte.

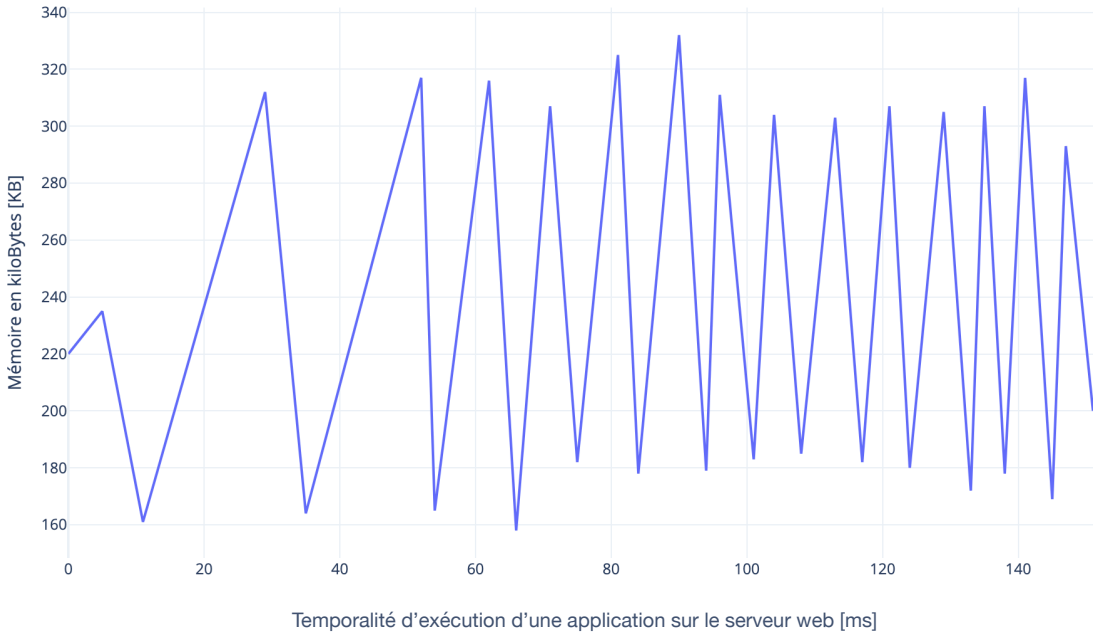


FIGURE 2.14 – Consommation mémoire RAM du serveur web en cours d'exécution d'un applicatif

Celui-ci a été entièrement écrit en Lua et se positionne comme une surcouche à Lynq. Des performances de temps de réponses inférieures aux solutions du marché sont présentées. Au regard de l'objectif initial, qui est de disposer d'un serveur web fonctionnant sans système d'exploitation capable d'être implémenté sur un *SoC FPGA*, couplé au fait que le nombre de client sera largement inférieurs aux serveurs web présenté, ceux-ci ont vocation à être utilisés à grande échelle. Le serveur web développé remplit nos objectifs, d'autant plus que ses performances en terme de consommation mémoire lui permettent d'être embarqués dans des systèmes devant être économe en ressources.

2.7 Conclusion

À travers ce chapitre, il a été présenté Lynq qui est un environnement d'exécution doté d'une interface de contrôle uniformisée. Cette solution répond également à des problématiques de rentabilité énergétique.
Lynq permet le contrôle et l'utilisation de ressources matérielles et logicielles depuis un langage de haut niveau au travers d'une API. Voici la synthèse des contributions

originales qui constituent la solution Lynq.

(i) À l'inverse des solutions de type PYNQ basées sur un système d'exploitation, Lynq se base sur une version modifiée de LuaJIT en mesure de fonctionner en standalone ;

(ii) Une couche d'abstraction du matériel haut niveau qui se base sur les résultats de la génération du BSP et permet l'accès au bas niveau matériel (*CPU* ou *FPGA*) depuis les couches supérieures de Lynq ;

(iii) L'API est décrite de manière modulaire, se basant sur la couche d'abstraction du matériel, permettant l'accès aux différents blocs et périphériques du *SoC FPGA* ;

(iv) Un serveur web qui tire parti des couches basses de Lynq pour proposer un outil standalone permettant une exécution de code distante ainsi que des outils de visualisation.

Ces contributions ont fait l'objet :
1) d'un article [140] accepté à la conférence *28th International Conference on Field Programmable Logic and Applications (FPL 2018)* ; 2) le serveur web est en cours de publication.
Au delà des objectifs pour lesquels Lynq a été développé, ses performances exposées en section 2.5.2 ouvrent la porte à une utilisation de Lua non seulement comme langage de contrôle, mais également comme langage orienté calcul, plus particulièrement à travers l'utilisation du FFI.
En outre, la simplicité d'utilisation et de contrôle des couches de bas niveau depuis Lua qui est couplé à l'utilisation du serveur web confirment la possibilité d'utiliser une telle solution comme base pour les travaux futurs relatifs à l'amélioration des performances globale du système et l'ajout de mécanismes pour l'amélioration de la sécurité.

3 Lynq ADvanced & ISOLynq

Dans le but de permettre un multiplexage temporel des ressources du *SoC FPGA* de manière sécurisée, ce chapitre présente des améliorations apportées à l'environnement d'exécution Lynq. Plus particulièrement, il introduit des mécanismes dont l'objectif est d'assurer des exécutions concurrentes ou parallèles sécurisées.

Pour répondre à cela, un rappel des objectifs et une présentation des caractéristiques principales sera fait en introduction. Puis, un contrôleur d'allocation de ressources dynamiques sera présenté dans la section 3.2. Quant à la section 3.3, elle décrira les mécanismes nécessaires aux environnements d'exécution pour offrir une isolation matérielle et logicielle.

3.1 Introduction

3.1.1 Objectifs

L'utilisation d'un environnement d'exécution permet de développer des programmes logiciels et des accélérateurs matériels depuis une interface logicielle unique. Les performances obtenues dépendent de plusieurs facteurs dont la capacité de transfert mémoire (fréquence, latence, débit) ainsi que les performances calculatoires intrinsèques de chaque bloc exécuté, qu'il soit matériel ou logiciel. Suite à la présentation de l'environnement d'exécution dans le chapitre précédent, la question de l'optimisation de l'utilisation des ressources logicielles ou matérielles disponibles se pose.

La littérature offre des solutions à cette problématique, notamment via l'utilisation de système d'exploitation dédié aux systèmes embarqués. Depuis 1998, avec [141] et cela jusqu'à aujourd'hui, tirer parti du parallélisme et de l'hétérogénéité des architectures reconfigurables a fait l'objet de nombreuses recherches [60, 61, 62]. En passant par l'amélioration du masquage du temps de reconfiguration dynamique partiel pour les *FPGA* [142] ou par le développement de systèmes d'exploitations [59, 58] pour les processeurs, des solutions ont émergé afin de tirer parti de l'hétérogénéité des *SoC FPGA*s. Celles-ci doivent répondre au paradoxe "simplicité d'utilisation" versus "optimisation et performances". Un axe d'investigation réside donc dans l'optimisation du multiplexage temporel des ressources reconfigurables hétérogénes disponibles sur le *SoC FPGA*. De même, la capacité d'un système à manipuler et restaurer ces contextes dans un environnement d'exécution est importante. Ceci afin de pouvoir basculer d'un applicatif à un autre selon des critères définis par rapport à la cible (priorité, temps alloué, dépendance d'une tâche à une autre). Généralement, le système d'exploitation qui fonctionne sur un CPU d'une architecture reconfigurable occupe cette fonction.

Or, la spécificité de notre environnement d'exécution Lynq réside dans l'absence de système d'exploitation. Il n'y a donc pas de contrôleur en charge de la gestion de l'exécution des applicatifs dans sa version de base. L'utilisateur doit ainsi définir explicitement l'ordre d'exécution ainsi que les ressources utilisées par son applicatif.

Le premier objectif de ce chapitre est donc de proposer une amélioration à l'environnement d'exécution présenté dans le chapitre 2 afin de pouvoir allouer dynamiquement les ressources rendues disponibles aux applicatifs. Il est important de noter que l'environnement d'exécution fonctionne en standalone. De ce fait, la gestion de l'allocation des ressources ne pourra pas utiliser les services communément disponibles d'un système d'exploitation.

Les bases de notre environnement d'exécution posées, nous avons ensuite étudié la

sécurisation de l'exécution de plusieurs applications sur la même cible. En cela, le second objectif de ce chapitre est de proposer l'agrégation de plusieurs méthodes de sandboxing logicielles et matérielles en une solution unique, pouvant être contrôlée depuis une interface de configuration administrée par Lynq. Elle s'appuie sur l'utilisation de Lynq, et introduit une pénalité logicielle minime pour les performances applicatives.

3.1.2 Caractéristiques

Afin de répondre aux problématiques et objectifs énoncés précédemment, des améliorations ont été apportés à l'environnement d'exécution présenté dans le chapitre 2. Cette solution est articulée autour d'un contrôleur permettant la gestion de l'allocation des ressources matérielles et logicielles disponibles de manière dynamique. Les principales caractéristiques de notre solution sont décrites ci-dessous :

- **Lynq ADvanced : Multiplexage temporel.** Lynq ADvanced se base sur des méta données relatives à un applicatif et offre une gestion de l'allocation des ressources disponibles de manière dynamique sur l'architecture hétérogène ciblée.
- **Lynq ADvanced : Mise à jour automatisé.** Lynq ADvanced garde une trace des temps d'exécution et en fait une analyse, pour permettre une mise à jour des données d'exécutions utilisées pour l'allocation de ressources dynamiques.
- **ISOLynq : Isolation d'Environnement d'Exécution.** ISOLynq se base sur l'environnement d'exécution Lynq et propose des mécanismes qui permettent l'isolation d'application utilisant des ressources matérielles et logicielles. Il n'implique pas d'utilisation de ressources matérielles supplémentaires. Il est entièrement configuré et contrôlé depuis Lynq.

3.2 Lynq ADvanced : allocation de ressources dynamiques

Cette partie consiste à ajouter les capacités à allouer dynamiquement les ressources du *SoC FPGA* à l'environnement d'exécution Lynq. Pour cela, chaque partie constituante d'un applicatif est considérée comme une tâche et matérialisée par une coroutine [143]. Celle-ci représente une exécution avec son pointeur d'instructions, ses variables locales et sa propre pile logicielle. Les coroutines offrent un système collaboratif. En effet, chacune d'entre elle passe la main à une autre coroutine aprés son exécution. À la différence des threads, plusieurs coroutines ne peuvent être exécutées que de manière concurrente.
Une coroutine comporte un ou plusieurs points d'entrées (ou de re-entrées) et, un ou plusieurs points de restauration. Un autre bloc de code (autrement dit une autre coroutine) est autorisé à s'exécuter pendant un temps donné lorsqu'une coroutine atteint un point de

restauration, l'exécution lui est ensuite rendu à un de ses points de re-entrée. Lorsqu'une coroutine rend la main au *CPU* (yield), la coroutine initiale reprend son exécution.

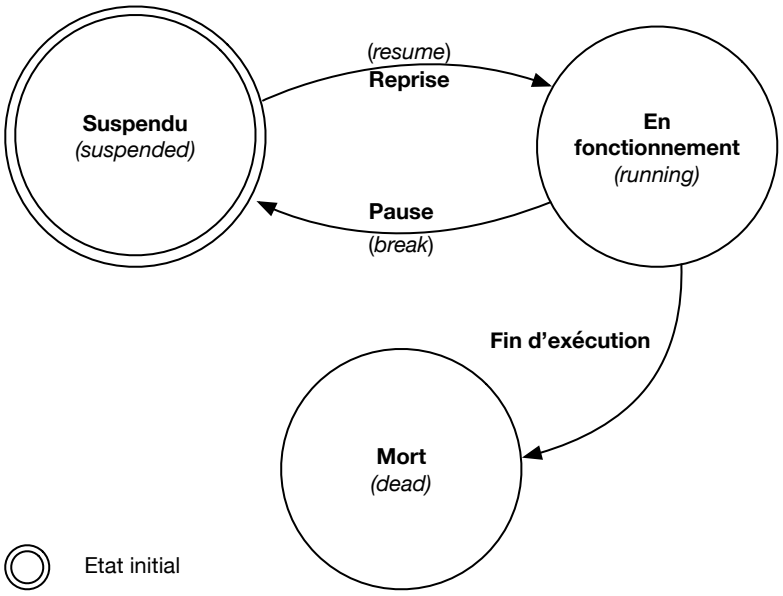


FIGURE 3.1 – Les états d'une coroutine

S'il y a plus de deux coroutines, un ordonnanceur détermine quelle coroutine peut être exécutée après soit un yield ou soit la mort d'une coroutine. Cette propriété implique que les paramètres et les données soient préservés (et restaurés si nécessaire) lorsque l'exécution est cédée à une autre coroutine. La figure 3.1 expose les états possibles d'une coroutine. Après sa création, une coroutine se retrouve dans un état suspendu (*suspended*). Elle entre ensuite dans un état de fonctionnement (*running*), jusqu'à ce qu'elle atteigne un point de restauration ou entre dans un état mort (*dead*) à cause de la fin de son exécution. Elle peut également s'interrompre via l'utilisation d'un yield qui permet de stopper une coroutine de manière explicite durant son exécution.

Le tableau 3.1 présente les avantages et inconvénients relatifs aux coroutines utilisées pour notre gestionnaire d'allocation de ressource dynamique pour Lynq. Celui-ci est associé à une comparaison entre coroutine et thread dans le tableau 3.2. La coroutine possède un système d'allocation mémoire dynamique à l'inverse des threads dont l'allocation mémoire est faite de manière statique avec l'allocation fixe à la création d'une quantité de mémoire. L'utilisation de coroutine permet donc de seulement consommer la quantité de mémoire nécessaire à son fonctionnement. La portabilité des coroutines en fait un atout. Plus spécifiquement, les coroutines ne dépendent pas d'un système d'exploitation, contrairement aux threads qui eux ont besoin d'un modèle d'implémentation de thread disponible sur l'OS (POSIX [144] pour le plus connu).

Avantages	
Portable	La mise en oeuvre de structure de gestion est facilement implémentable avec l'utilisation de coroutines. En effet, elles ne nécessitent pas l'utilisation de mécanismes provenant d'un système d'exploitation.
Économe en mémoire	Contrairement aux threads fonctionnant sur système d'exploitation et pour lesquels une quantité de mémoire fixe est alloué à la création, l'espace nécessaire à une coroutine est dynamiquement alloué.
Asynchrone	L'utilisation de coroutine permet la gestion de ressources à caractère asynchrone telles que les entrées et sorties du *SoC FPGA*.
Inconvénients	
Méthode non préemptive	À l'inverse des systèmes d'exploitation préemptifs qui ont la faculté d'exécuter ou stopper une tâche en cours, l'utilisation des coroutines implique une gestion non préemptive. Ceci se traduit par une incapacité du gestionnaire à interrompre une coroutine en fonctionnement, celle-ci étant la seule à en avoir la capacité.
Bloquant	Une mauvaise utilisation des coroutines peut entrainer un programme à ne jamais terminer son exécution du fait d'une coroutine ne rendrant jamais la main.

Tableau 3.1 – Avantages et inconvénients relatifs à l'utilisation des coroutines

	Portabilité	Type d'allocation mémoire	Multitâches
Coroutine	Oui	Dynamique	Coopératif, Asynchrone
Thread	Dépendence à l'OS	Allocation Statique	Préemptif, Synchrone

Tableau 3.2 – Comparaison coroutine / thread

3.2.1 Types de coroutine

Il existe différents types de coroutines permettant l'exécution de code Lua, en utilisant un interpréteur ou un compilateur JIT. Ceci peut se faire grâce à des appels aux langages C ou C++ via la fonctionnalité FFI ou la sollicitation de ressources matérielles. Ces quatre types de coroutines sont détaillées ci-dessous :

- **Coroutine pré-compilée avant exécution :** Cette coroutine permet l'exécution d'un binaire Lua pré-compilé. Ce type de binaire est généré avec le compilateur Just-

In-Time et atteint des performances proches de celle du C compilé sans optimisation. Ils sont plus rapides que ceux compilés pendant l'éxécution car aucun temps de compilation n'est nécessaire.

- **Coroutine interprétée ou compilée pendant l'exécution :** Cette coroutine exécute du code Lua via l'interpréteur Lua ou le compilateur Just-In-Time (JIT) intégré. En utilisant un compilateur JIT, les même performances que précédemment (proches du code C compilé sans optimisation) peuvent être atteintes grâce à la détection et à l'optimisation des points chauds pendant le processus de compilation comme présenté dans le chapitre 2.
- **Coroutine C / C++ utilisant la fonctionnalité FFI :** De la même manière que pour l'exécution de code Lua, il est possible d'appeler des fonctions C/C++ compilées à partir de code interprété ou pré-compilé via une interface d'accès aux fonctions étrangères (FFI).
- **Coroutine matérielle sur *FPGA* :** Une coroutine est créée même lorsqu'un accélérateur matériel doit être exécuté. Sa création est nécessaire pour permettre l'appel à une ressource matérielle. Ceci pour gérer la ressource reconfigurable dans le cas où l'emplacement prévu n'est pas déjà chargé avec l'accélérateur requis. La coroutine envoie un signal au contrôleur d'allocation de ressources dynamiques afin de demander l'autorisation d'utiliser une zone reconfigurable ou la reconfigurer le cas échéant. Si la ressource n'est pas disponible, le contrôleur sera en mesure d'utiliser un autre type de coroutine pour exécuter la tâche.

3.2.2 Allocation dynamique

Notre gestionnaire d'allocation de ressources dynamiques (DRA pour Dynamic Ressources Allocation) est basé sur l'environnement d'exécution Lynq qui abstrait le matériel de bas niveau et offre une interface d'accès unique au *FPGA* et au *CPU* à partir d'une seule *API*. Notre DRA utilise des coroutines pour gérer les tâches, c'est à dire qu'une coroutine est l'implémentation d'une tâche à exécuter. L'architecture ciblée est la même que celle pour laquelle Lynq a été développé.

La figure 3.2 présente les différentes couches depuis le bas niveau matériel jusqu'aux tâches de l'utilisateur devant être exécutées. Cette approche permet de gérer les ressources de niveau inférieur en associant les briques bas niveau de Lynq présentées au chapitre 2 (appelé Lynq Core dans la figure). Elle utilise un module en charge de la gestion de l'allocation de ressource dynamique. Notre approche de gestion des ressources s'appuie sur quatre caractéristiques :

- **Léger.** DRA consomme moins de 15 kB de code Lua et utilise 126 kB en plus de

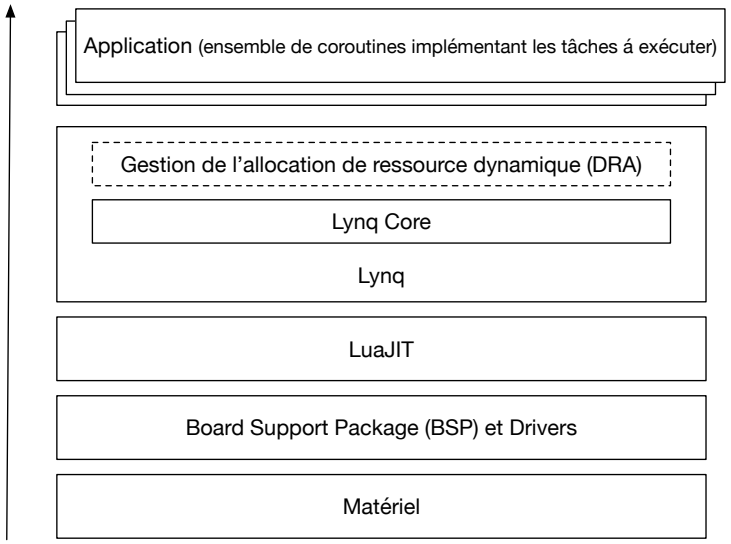


FIGURE 3.2 – Pile d'exécution

la mémoire nécessaire à la coroutine pour son exécution. La mémoire nécessaire varie selon l'algorithme de la coroutine. Comme déjà expliqué, la mémoire allouée à la coroutine (implicitement sa consommation mémoire) est dynamique selon ses besoins.

- **Ordonnancement collaboratif.** DRA utilise une méthode d'ordonnancement non préemtive dite collaborative. Pendant l'exécution d'une coroutine, seul un yield peut rendre la main au DRA.
- **Synchronisation explicite.** Les applications peuvent inclure des yields dans leur code pour définir explicitement les interactions entre les tâches. Cette méthode de synchronisation permet également d'anticiper l'utilisation d'un slot de reconfiguration partielle avant l'exécution pour masquer le temps de reconfiguration.
- **Gestion des priorités.** Une priorisation dans la gestion des coroutines peut être utilisée avec notre DRA. Un utilisateur peut attribuer une priorité à chaque tâche au cours du processus de conception à l'exécution. Au moment de l'exécution, le DRA peut également arbitrer et modifier les priorités au besoin. Dans le cas où une coroutine a besoin d'une reconfiguration partielle et que le gestionnaire ne reçoit pas le signal indiquant que la reconfiguration est terminée, celle-ci a une priorité moindre. Les tâches peuvent ainsi être exécutées durant les phases d'attente et de reconfiguration.
- **Résultats partiels.** Une coroutine Lua peut générer des résultats partiels. Pendant l'exécution de la tâche, lorsqu'un yield se produit, des données peuvent être transmise à d'autre coroutines au moment où celle-ci rend la main.

Notre gestionnaire d'allocation de ressources dynamique est basé sur une version adaptée

de l'algorithme de planification équitable (de l'anglais CFS pour Completely Fair Scheduling) [145]. Derrière cet algorithme se cache l'idée que chaque tâche devant être exécutée doit bénéficier d'un temps d'accès au *CPU* équitable par rapport aux autres tâches. Si à un instant donné, des tâches n'ont pas assez de temps alloué pour leur exécution, on parle de tâche en état de famine. Dans ce cas, il faut leur donner la main pour qu'elles puissent s'exécuter.
Pour maintenir cet équilibre, l'algorithme enregistre les temps alloués à chaque tâche dans ce qui est appelé le temps d'exécution virtuel. Plus une tâche présente un temps d'exécution virtuel petit (sous entendu moins une tâche a eu accès au *CPU*), plus elle a besoin d'accéder au *CPU*. Cet algorithme prend également en compte les éléments extérieurs à l'exécution propre de la tâche comme l'attente d'une entrée / sortie ou dans notre cas de la disponibilité d'une zone reconfigurable ou de sa reconfiguration. Cela se traduit par la mise à disposition de temps d'exécution *CPU* quand la tâche peut s'exécuter. Pour maintenir une vision du temps d'exécution virtuel des tâches, cet ordonnanceur utilise un arbre rouge et noir (de l'anglais RB Tree pour Red and Black Tree). Sa structure est basée sur deux propriétés qui le rendent approprié à cet usage : (1) la structure des données est auto-équilibrée ; aucun chemin dans l'arborescence n'est plus de deux fois plus long que tout autre chemin et (2) les opérations d'insertion et de suppression dans l'arbre se produisent en temps O(log n), où n est le nombre de nœuds dans l'arborescence. Cette propriété permet à notre DRA d'insérer ou de supprimer rapidement des nœuds représentant les coroutines de son arbre d'exécution.

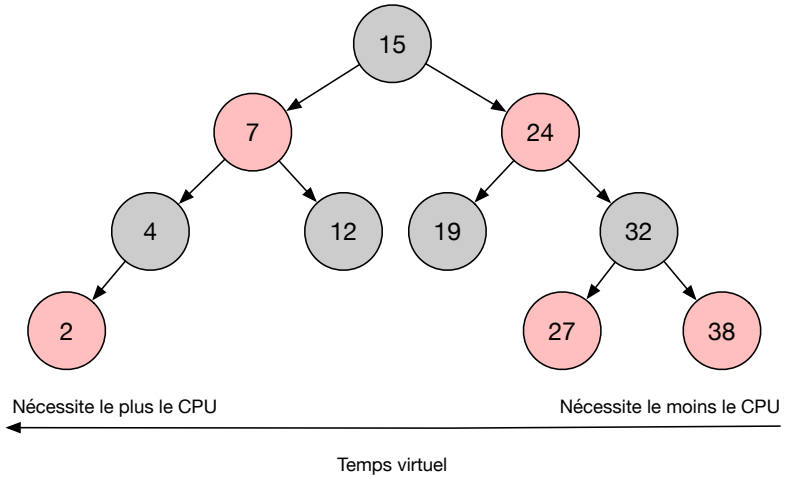


FIGURE 3.3 – Arbre Rouge et Noir

La figure 3.3 présente la structure de données en arbre. Les tâches qui ont le plus grand besoin du *CPU* (durée d'exécution virtuelle la plus faible) sont stockées vers la gauche de l'arborescence et les tâches qui en nécessitent le moins (durée d'exécution virtuelle la plus élevée) vers la droite. Pour être équitable, l'algorithme choisit donc le noeud (ici le

2 rouge) le plus à gauche. Si la tâche doit à nouveau être exécutée, son temps virtuel d'exécution est mis à jour puis, elle est à nouveau insérée dans l'arbre. Les noeuds de l'arbre se déplacent donc de droite à gauche pour maintenir l'équité.

Lorsque le DRA (en suivant l'algorithme CFS) donne la main à une coroutine, celle-ci entre dans un état de fonctionnement. Après une yield, le contrôle est restitué au DRA et la coroutine en cours d'exécution entre dans un état suspendu. Une coroutine nouvellement sélectionnée se met alors en mode de fonctionnement.

Basée sur le gestionnaire DRA, la prochaine partie propose une amélioration de l'algorithme CFS pour prendre en compte la particularité de Lynq. Ce dernier permet l'utilisation de ressources logicielles et matérielles et offre une solution visant à améliorer le multiplexage temporel des ressources du *SoC FPGA* lorsqu'une tâche peut être exécutée en utilisant plusieurs ressources.

3.2.3 Amélioration du DRA

Durant l'exécution, le DRA enregistre les temps virtuels alloués à chaque coroutine. Deux cas peuvent être distingués pour connaître le temps d'exécution virtuel des coroutines pré-exécution. Premièrement, si la coroutine utilise du code logiciel, une approximation du temps d'exécution peut être faite à partir de la fréquence du *CPU*. Le second cas est celui de l'utilisation d'un accélérateur matériel pour lequel les informations de latence ne peuvent être récupérées par le DRA que si elles sont fournies par le concepteur. À partir des données disponibles au démarrage, le DRA affecte les ressources en fonction des coroutines. Puis, il adapte ses régles d'allocation en fonction des temps virtuels d'exécution. Pour cela, chaque fois que le DRA exécute une tâche pour la première fois, il calcule le temps d'exécution d'une implémentation spécifique (coroutine matérielle, code interprété ou code JIT).

L'exemple de la figure 3.4 est utilisé pour présenter ce mécanisme. Il décrit les mécanismes mis en place pendant l'exécution. Dans cet exemple les temps d'exécution logiciels de l'ensemble du benchmark Scimark (*FFT, SOR, MC, SPARE, LU*) sont connus pour les deux modes logiciels. Le temps d'exécution matériel de la *FFT* est également connu. Aucune donnée relatives aux ressources matérielles pour les quatre autres kernels du benchmark ne sont disponibles avant le début de l'exécution. Nous considérons la première exécution d'une tâche qui consiste en l'utilisation du kernel *LU* pour laquelle sont disponibles les deux versions logicielles et la version matérielle de la tâche. Lorsque le DRA doit éxécuter la tâche associée à cette application, il donne la main à la coroutine 1. La zone reconfigurable étant disponible, la coroutine demande la reconfiguration puis se

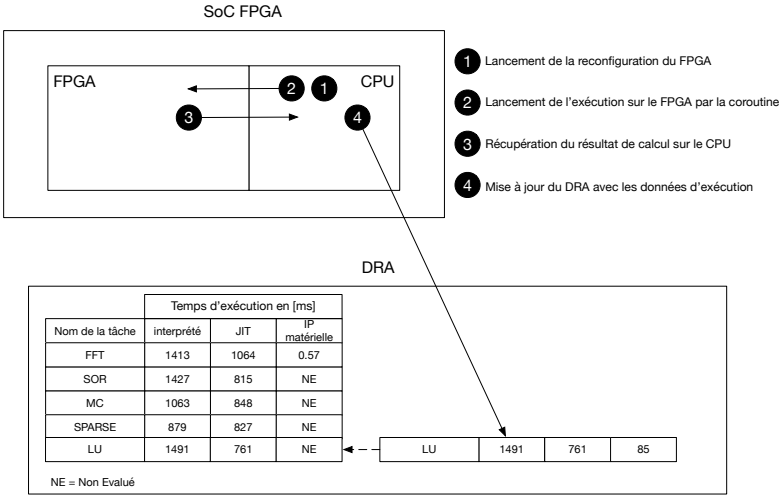


FIGURE 3.4 – Mécanisme d'amélioration du contrôleur d'allocation de ressources dynamiques

met en pause via l'appel à la fonction yield ❶. Une seconde coroutine (2) devant exécuter le kernel *FFT* prend la main, celle-ci utilise le code JIT car la zone reconfigurable est en cours d'utilisation. La coroutine 1 reprend la main suite à la fin de la reconfiguration et lance l'exécution sur le *FPGA* puis se met à nouveau en pause ❷. Ensuite une coroutine (3) exécutant un kernel *MC* prend la main et utilise également le code JIT car la version matérielle n'est pas disponible. Une fois que l'accélérateur matériel a terminé son exécution et qu'un signal est remonté au DRA, la coroutine utilisant la ressource matérielle peut reprendre la main et traiter le résultat obtenu sur le *CPU* ❸. Suite à cela, le temps d'exécution matériel est calculé par le DRA ❹. Le chronogramme qui reprend cette chronologie est disponible en figure 3.5.

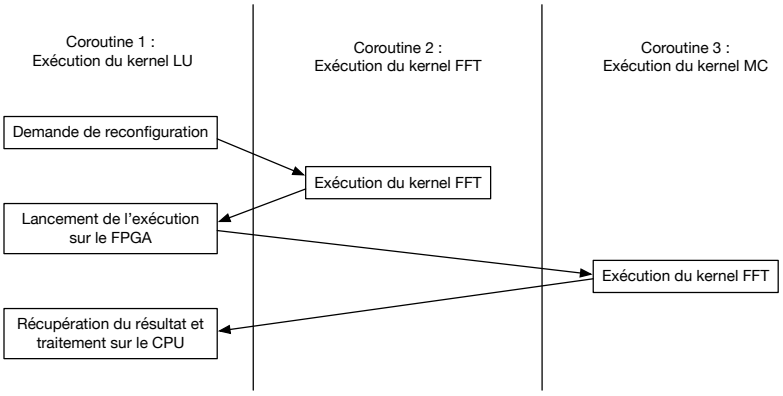


FIGURE 3.5 – Chronogramme de l'exécution d'un programme à 3 coroutines

Puisque le DRA dispose à présent du temps d'exécution du kernel (LU), il favorisera l'utilisation de la version matérielle quand la ressource sera disponible lors de prochaine

exécution. Dans le cas où l'IP sera mise à jour, un signal sera envoyé au DRA et conduira à une réévaluation de la latence matérielle.

3.2.4 Évaluation des performances

Pour conduire l'évaluation des performances offertes par le DRA, nous avons repris le benchmark Scimark et avons mené une étude comparative entre les versions interprétées et JIT. Nous avons également intégré l'utilisation d'un accélérateur matériel pour les kernels FFT et LU. À ce titre, le tableau 3.3 présente les temps d'exécution des différents kernels du benchmark Scimark évoquées précédemment. Les temps d'exécution des 5 benchmarks, lorsqu'ils sont exécutés en logiciel, sont toujours en faveur de la version LuaJIT 2.0.5 (jit, ffi). Il en est de même pour la consommation mémoire, à l'exception de MC dont la mémoire consommée est inférieure lors de l'utilisation de Lua 5.1. L'amélioration du temps d'exécution des kernels entre les versions interprétées et JIT est de 1.47 fois plus rapide [1.06 fois - 1.96 fois]. Ces informations apportent donc au DRA l'information permettant de privilégier l'utilisation de LuaJIT 2.0.5 comparé à une exécution interprété. L'ajout des temps d'exécution matériels des deux kernels (FFT et LU) permet au DRA de les utiliser s'ils sont déjà configurés sur la cible *SoC FPGA*. Par comparaison des temps d'exécution des implémentations logicielles, ils seront directement utilisés. S'ils ne sont pas déjà configurés, le DRA prendra en compte la disponibilité d'une zone reconfigurable ainsi que le temps de reconfiguration associé.

	Interpréteur de LuaJIT 2.0.5		LuaJIT 2.0.5 (jit, ffi)		IP matérielle
	Temps d'exécution [ms]	Mémoire consommée [kB]	Temps d'exécution [ms]	Mémoire consommée [kB]	Temps d'exécution [ms]
FFT	1413	1.95	1064	25.13	0.57
SOR	1427	81.78	815	78.70	-
MC	1063	0.76	848	19.60	-
SPARSE	879	153.70	827	84.38	15.16
LU	1491	105.11	761	14.74	85

Tableau 3.3 – Temps d'exécution et consommation mémoire des coroutines en versions logicielle utilisant LuaJIT 2.0.5 en version interprétée et JIT ainsi qu'une version matérielle appliquée au benchmark Scimark 2.0

Dans cette section, Lynq ADvanced a été présenté comme un gestionnaire d'allocation de ressources dynamiques qui permet un multiplexage temporel de celles-ci sur un *SoC FPGA* à partir d'une liste de tâches. Ceci permet à plusieurs applicatifs de pouvoir être exécutés sur le même *SoC FPGA* en bénéficiant de l'aspect collaboratif offert par les coroutines disponibles en Lua couplées à Lynq. Cependant, l'introduction de la capacité à utiliser plusieurs applicatifs soulève une problématique relative à la sécurisation de l'environnement d'exécution de ces derniers. Comment permettre à deux applicatifs ayant

un niveau de criticité différent de pouvoir être exécutés de manière sûre et plus particulièrement de manière isolée ? La collaboration permet aux coroutines de communiquer entre elles, ce qui est intéressant au sein d'un même applicatif. Cependant, mal utilisé, ceci pourrait tendre à des dérives conduisant à du vol d'information ou l'introduction de code malveillant compromettant l'intégrité d'une application. En ce sens, la seconde contribution de ce chapitre apporte donc des mécanismes d'isolation des environnements d'exécution de Lynq pour pallie cela.

3.3 ISOLynq : isolation inter environnement d'exécution

Afin de répondre aux problématiques exposées dans la section précédente (3.1.1), le choix de sécuriser un environnement d'exécution en se basant sur Lynq a été fait. Cela signifie que des mécanismes de restrictions bas niveau sont rendus accessibles depuis un langage de haut niveau, en l'occurence Lua. L'objectif est de permettre l'exécution de plusieurs applicatifs tout en garantissant un accès maitrisé aux ressources, c'est-à-dire en donnant l'accès à un applicatif aux ressources dont il a besoin. Ces choix et leurs justifications font l'objet de cette section. Pour cela, des précisions sur le fonctionnement interne de LuaJIT seront apportées. En effet, les mécanismes implémentés se basent sur l'architecture logicielle de Lua. Il est donc important de les exposer pour comprendre la solution proposée. A la suite de cela, le modéle de menace sera exposé puis notre solution présentée.

Comme décrit dans le section 2.4 du chapitre 2, Lynq associe un état (state) Lua à un contexte d'exécution. Chaque application peut utiliser des ressources logicielles et matérielles durant son exécution dans un environnement d'exécution. Un état Lua contient également le contexte d'exécution de l'interpréteur Lua associé.

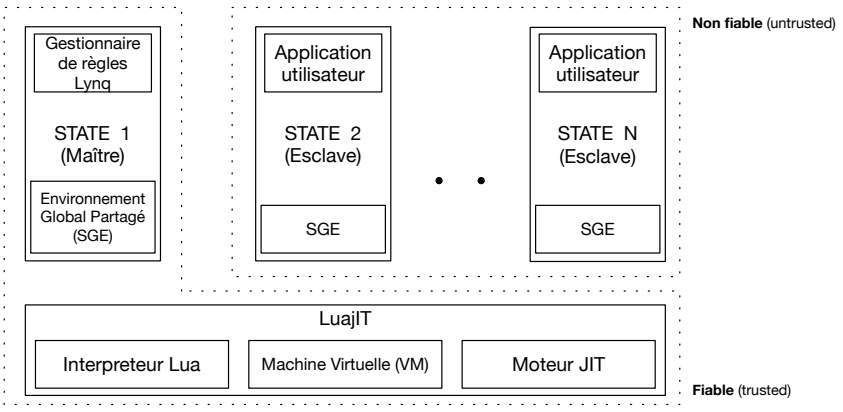


FIGURE 3.6 – Définition des zones sécurisées dans la pile logicielle Lua lors d'une utilisation avec LuaJIT

La figure 3.6 expose le positionnement des états (structure logicielle) par rapport à la pile logicielle Lua au travers duquel l'ensemble des scripts de Lynq sont exécutés. Cette pile logicielle Lua est composée d'un interpréteur, d'une machine virtuelle (*VM*) et, pour LuaJIT, d'un compilateur JIT. Au démarrage du Lynq, durant la phase de création d'un contexte Lua (cf. chapitre 2 figure 2.7), un état maître est créé auquel s'ajoute la contribution de ce chapitre : un environnement d'exécution sécurisé (*TEE*). Ce dernier est en charge de la gestion des mécanismes de sandboxing et fera l'objet de la section 3.3.2.

Un environnement global partagé (*SGE*) existe localement dans chaque état Lua. Celui-ci contient des règles qui s'appliquent à toutes les applications exécutées dans cet état. L'implémentation de ce *SGE*, qui sera présentée plus en détail en section 3.3.2, se traduit par une table associative résolvant l'accès à toutes les variables de l'état Lua. Schématiquement, le *SGE* autorise l'accès aux différents modules, zones mémoires et I/O de la cible. Dans la suite de cette section, Nous utiliserons un accélérateur matériel AES pour présenter le fonctionnement de notre mécanisme. À noter que celui-ci est applicable aux autres accélérateurs matériels. La solution que nous avons retenue est basée sur la modification du *SGE* des états esclaves par le gestionnaire de sandboxing de Lynq.

3.3.1 Modèle de menace

Dans notre modèle de menace, nous considérons que LuaJIT est *fiable*. Le processeur ARM du *SoC FPGA* cible exécutant Lynq est également protégé contre les accès physiques directs. Un attaquant n'a pas d'accès à l'état Lua maître, ni à aucun des *SGE* des états Lua esclaves. Les attaques physiques ne font pas l'objet de cette l'étude. Nous partons du postulat que l'adversaire est en mesure d'accéder aux fichiers sources des applicatifs en Lua ainsi qu'aux bitstreams stockés en mémoire DDR et sur la carte SD. Une connexion Ethernet est également disponible et permet le téléchargement de fichiers provenant d'une source de stockage non sécurisée vers la mémoire de stockage du *SoC FPGA*. L'attaquant peut donc charger une application malveillante en mémoire et l'exécuter à l'aide de script Lua, de binaire précompilé Lua ou de bitstream de configuration. Ces vecteurs sont chargés en mémoire DDR ou carte SD en amont de l'exécution et sont accessibles durant l'exécution via des appels à l'API de Lynq. Le but de l'attaquant est d'accéder aux données des applications exécutées de manière concurrente ou parallèle. L'attaquant ne peut pas exécuter de code malveillant tant que la séquence de démarrage de Lynq n'est pas terminée, c'est-à-dire tant que l'état maître Lua n'est pas créé et l'environnement d'exécution sécurisé (TEE) configuré. Les figures 3.6 et 3.7 exposent les éléments du modéle de menace et représentent l'environnement d'exécution implémenté

dans l'état maître tout comme dans les états esclaves (respectivement environnement d'exécution fiable et non fiable). Pour chaque application, nous supposons l'allocation d'au moins un état esclave pour héberger un environnement d'exécution non fiable (UEE). Le gestionnaire de sandboxing de Lynq applique ensuite un profil de sécurité au SGE de l'état esclave. Lorsque l'application se termine, le gestionnaire supprime l'état esclave. Il est important de rappeler que la gestion de création des états, la gestion des profils de sécurité ainsi que la décision d'exécuter un applicatif Lua est uniquement possible depuis l'état maître Lua.

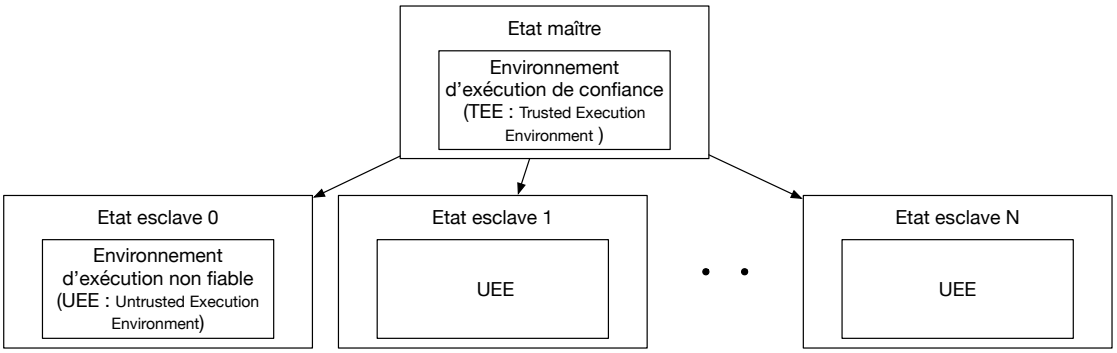


FIGURE 3.7 – Vue globale des environnements d'exécution **fiable** (*trusted*) et **non fiable** (*untrusted*) implémentant les états maîtres et esclaves

3.3.2 Fonctionnement d'ISOLynq

Notre mécanisme de sandboxing est illustré figure 3.8. Il porte sur l'utilisation d'un gestionnaire de sandboxing et se compose de cinq étapes :

1. Création de l'état maître au démarrage de Lynq ;
2. Exécution du programme maître sécurisé chargé de gérer l'exécution des applications de manière sécurisée à l'aide du gestionnaire de sandboxing ;
3. Création d'un état esclave par le biais de l'état maître Lua lorsqu'une application est prête à être exécutée ;
4. Application d'un profil de sécurité prédéfini au SGE de l'état esclave ;
5. Exécution de l'application dans l'état esclave.

Le gestionnaire de sandboxing de Lynq est une extension de son API. Il est chargé de fournir à l'état maître les fonctions de haut niveau pour manipuler les SGE des états esclaves exécutant les applications non sécurisées.

Les règles de sécurité applicables aux environnements d'exécution peuvent être regroupées dans des profils prédéfinis qui sont supervisés par le gestionnaire de sandboxing. Celui-ci peut également ajouter, modifier ou supprimer des règles de sandboxing dans le profil

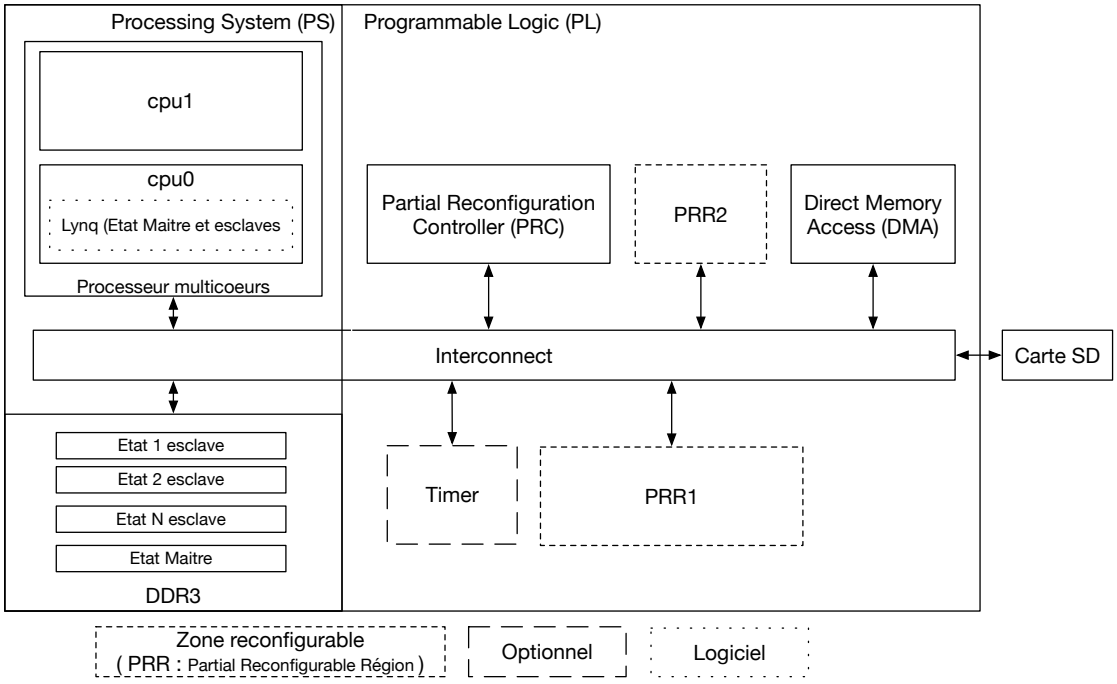


FIGURE 3.8 – Diagramme bloc de l'architecture de base qui implémente les mécanismes de sandboxing

d'un *SGE* esclave dynamiquement durant l'exécution d'une application. Tout comme un *SGE*, un profil est implémenté à l'aide d'une table associative. Trois types de règles peuvent être appliquées :

- **Sur les modules** : Les modules de l'API Lynq sont seulement disponibles à partir d'un état esclave lorsque cela est nécessaire. Les applications accèdent aux modules autorisés au moment de l'exécution.
- **Sur la mémoire** : L'état maître Lua gère l'allocation de la mémoire et la résolution des adresses. Chaque état esclave possède une plage de mémoire physique abstraite séparée et manipule une plage d'adresses virtuelles. Les applicatifs non sécurisés n'ont donc jamais accès directement aux adresses mémoires physiques. Ceci peut être comparé à la fonction de protection de la mémoire présente dans les unités de gestion de la mémoire (MMU pour Memory Management Unit). Cependant, notre environnement d'exécution fonctionnant sans système d'exploitation, l'utilisation d'une MMU n'est pas possible. Elle est remplacée par notre mécanisme de sandboxing.
- **Sur les I/O** : Les règles appliquées aux *SGE* permettent une restriction des entrées / sorties et des accès aux fichiers stockés en mémoire. Cette limitation permet la restriction des reconfigurations dynamiques des PRR en interdisant

l'accès aux bitstreams. Toutes les fonctions sont également inclus dans les I/O. En effet, elles peuvent provenir soit du langage soit de scripts externes (fichiers stockés en mémoire). Nous avons considéré l'ensemble des fonctions comme des I/O qu'il est possible de restreindre.

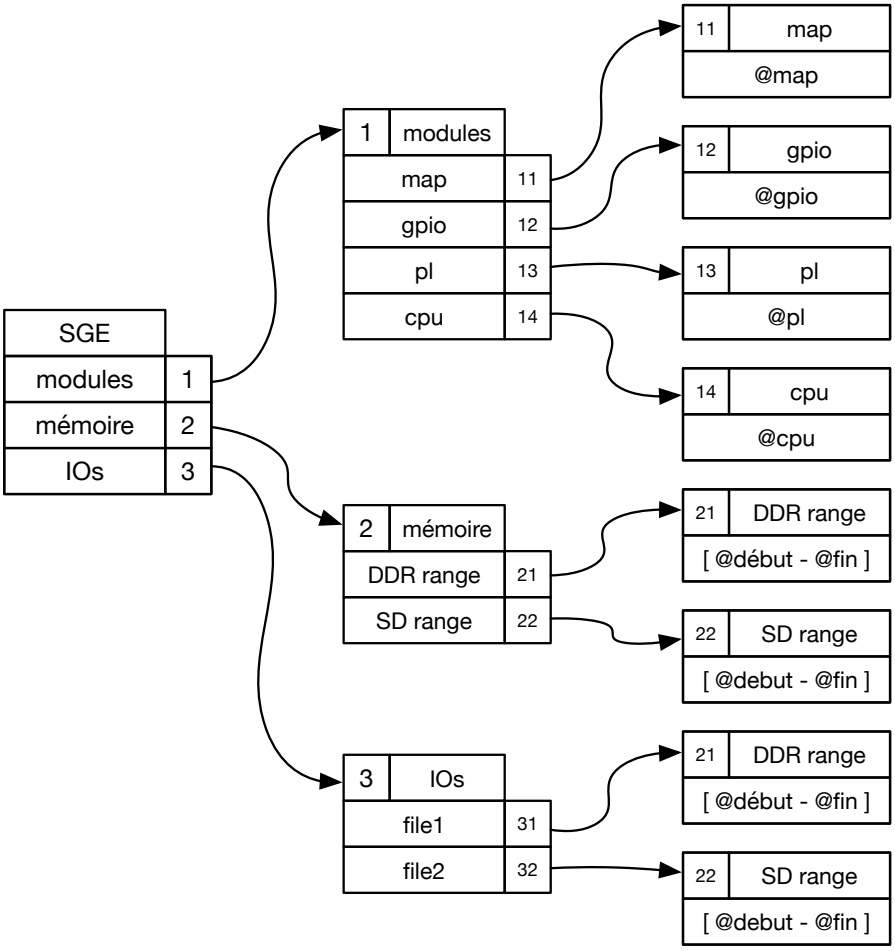


FIGURE 3.9 – Exemple de profil SGE prédéfini

Afin de prendre en charge le sandboxing, des profils prédéfinis fournissent des règles d'isolation de base qui sont appliquées suite à la création d'un état esclave. Le profil définit les modules de l'API Lynq autorisés, les règles applicables aux entrées / sorties ainsi que les segments mémoires accessibles. La figure 3.9 présente un exemple de profil qui peut être appliqué (Le symbole @ indiquant *adresse de*). Ces profils initialisent la table associative du *SGE* avec des valeurs de base. Pour cela, une liste des modules accessibles est initialisée, de même que la plage adresse accessible pour l'exécution de l'application ainsi qu'une liste d'entrées / sorties. Dans notre exemple, les modules MAP, GPIO et PL sont autorisés à être utilisés et le module *CPU* ne l'est pas. L'utilisation du bitstream AES est autorisé, toute autre tentative d'utilisation d'un autre bitstream (matérialisé

dans la table avec un **.bin*) provoquerait la reconfiguration du bitstream AES. Enfin, une plage d'adresse en mémoire DDR est définie et le profil autorise l'accès à la plage mémoire de la mémoire SD contenant le bitstream AES. Durant l'exécution, les accès et actions non définis ne sont pas exécutés dans la configuration par défaut. Cependant, il est également possible de rediriger toute action non définie vers une action de remplacement définie dans le *SGE* comme présenté dans l'exemple avec le bitstream AES. Lors de l'exécution, deux scénarios sont donc possibles lorsqu'un attaquant essaye d'accéder à une zone mémoire pour laquelle il n'a pas les droits : reconfigurer dynamiquement le *FPGA* avec un bitstream non autorisé ou accéder à un module de l'API restreint. Un premier scénario peut être appliqué quand une règle de redirection est définie. Par exemple, si une application tente de configurer un accélérateur AES spécifique mais que l'accès au bitstream est interdit, une règle de redirection peut définir la reconfiguration de la zone avec un bitstream alternatif. Dans le cas où une règle de redirection n'est pas définie, un deuxième scénario est appliqué. Il consiste à ne pas exécuter l'action demandée.

Pour chaque appel à une fonction logicielle, une entrée / sortie ou mémoire, l'environnement d'exécution vérifie l'approbation de chaque opération via une comparaison avec les règles du *SGE*. Pendant l'exécution, ces contrôles sont fonctionnellement transparents pour l'utilisateur car les mécanismes de sandboxing sont directement intégrés à Lynq et exécutés dans le moteur de LuaJIT. Une pénalité temporelle est introduite mais aucune ressource matérielle supplémentaire n'est nécessaire. Cette pénalité est induite par le temps de comparaison avec les règles de l'environnement d'exécution.

3.3.3 Cas d'étude

La figure 3.10 établit le diagramme d'exécution du cas d'étude présenté ci-après. Deux utilisateurs exécutent leurs applications respectives sur le même *SoC FPGA* : un utilisateur lambda (Blanc) et un autre avec de mauvaises intentions (Rouge). Blanc exécute une de ses *IPs* propriétaires sur le *SoC FPGA*. Rouge connaît l'adresse où Blanc stocke le résultat des calculs effectués par son *IP*. Son objectif est de les chiffrer et demander une rançon à Blanc en échange de la clé de déchiffrement. Ce type d'attaque est communément appelé *ransomware*. Rouge est en mesure d'utiliser un algorithme de chiffrement logiciel dont il a l'implémentation en Lua. Après la création de l'état Lua esclave pour Rouge, un profil est appliqué au *SGE* de son environnement d'exécution. Celui-ci limite la plage mémoire et l'accès aux IP qui ne lui appartiennent pas. Au moment de l'exécution, Rouge n'est donc pas en mesure d'accéder à la plage mémoire de Blanc.

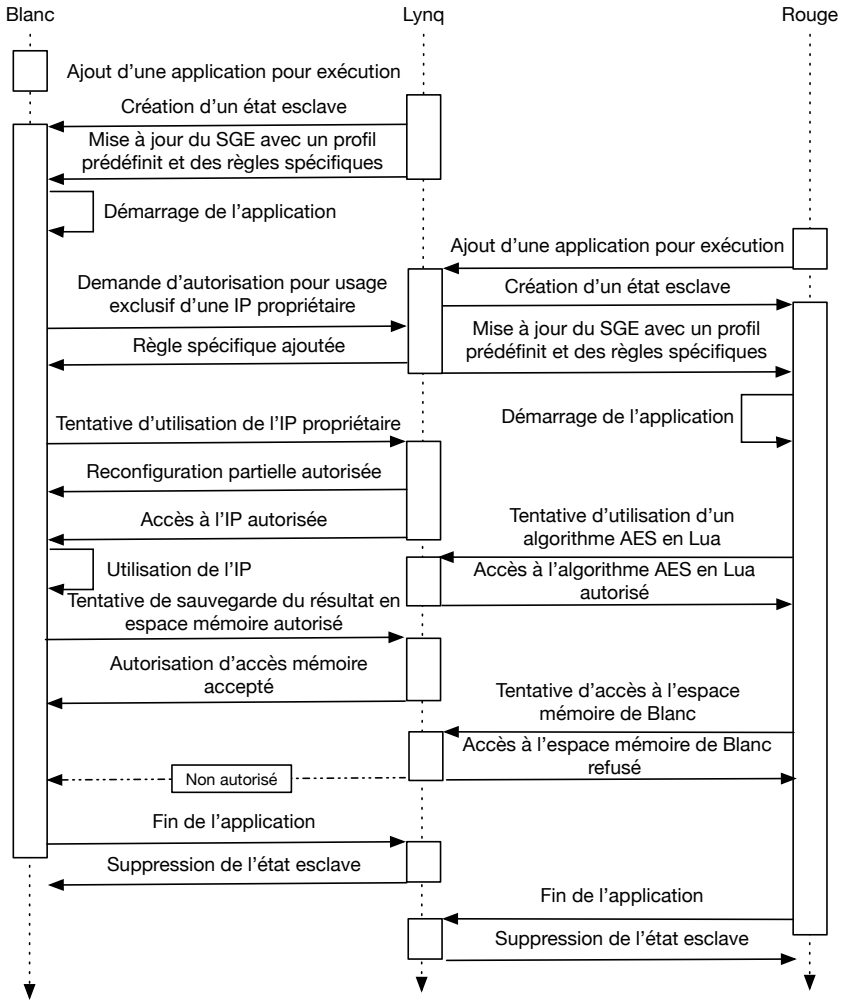


FIGURE 3.10 – Exemple d'exécution avec un attaquant souhaitant accéder une zone mémoire privée

3.3.4 Compromis sur l'utilisation du FFI

L'exécution de script Lua est exposée figure 3.11. Les fonctions C externes sont directement appelées via l'interface FFI de LuaJIT.
Contrairement aux fonctions C encapsulées en Lua, celles qui sont appelées via l'interface FFI sont ajoutées à une table de symboles chargée au démarrage de LuaJIT. Elles sont ensuite accessibles pendant l'exécution.

Pour pleinement supporter la fonctionnalité FFI sans système d'exploitation, la mise en oeuvre d'appel système bas niveau est nécessaire. En effet, pour fonctionner Lynq doit récupérer un pointeur vers une table de hachage interne au moteur JIT de LuaJIT. Cela afin de pouvoir accéder aux symboles des fonctions FFI depuis l'extérieur. Cette table est similaire à une table de symboles que l'on retrouve dans les systèmes d'exploitation.

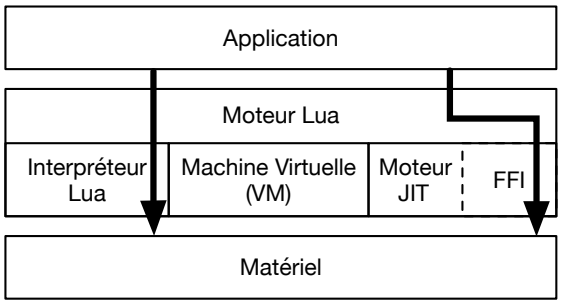


FIGURE 3.11 – Pile fonctionnelle pour l'utilisation du FFI

Elle est utilisée par le moteur de LuaJIT afin d'énumérer toutes les fonctions C externes qu'il est possible d'appeler via FFI. Pendant l'exécution d'une application, le code est interprété ou compilé via le compilateur JIT puis, le code machine émis est exécuté. Pour les appels *FFI*, le moteur LuaJIT a un accès direct à la table de symboles pour exécuter la fonction spécifiée. Si la table de symboles pointe vers une mémoire non-sécurisée, un attaquant peut modifier (et potentiellement exécuter) une fonction malveillante dans l'environnement d'exécution sécurisé. De plus, la fonctionnalité FFI permet un accès à tous les types de données C et rend possible une manipulation spécifique de la mémoire, à savoir à l'adresse près. L'attaquant peut donc contourner les régles de sandboxing et accéder à des plages mémoires non autorisés. Il peut aussi déréférencer des pointeurs NULL ou changer le prototype d'une fonction [146]. Par conséquent, il est nécessaire d'appliquer des mesures de protection complémentaires pour utiliser la fonctionnalité FFI. Une alternative à la restriction de l'utilisation du FFI pourrait également être l'encapsulation des appels FFI dans des fonctions Lua et l'application de restriction sur la plage d'accès mémoire. Cette méthode réduit le gain de performances offert par l'utilisation du FFI et elle allonge le temps de développement puisqu'elle rajoute une étape qui consiste à développer les fonctions d'encapsulations. Pour pallier cela, la seule utilisation du FFI autorisée et celle permettant l'appel de fonctions mises à disposition par l'état Lua maître.

3.3.5 Evaluation des performances

Pour évaluer l'impact des mécanismes de sandboxing, nous avons appliqué le même dispositif expérimental qui est décrit dans le chapitre 2.5.2. La performance du système avec les mécanismes de sandboxing est comparée aux performances de Lynq dans sa version initiale présentée au chapitre 2. Avant d'exposer les performances, il est à noter que l'implémentation de la solution ISOLynq n'utilise pas de ressources matérielles, son coût est purement logiciel et ne demande pas de modification de son architecture par l'utilisateur. Notre approche se base exclusivement sur l'ajout de briques logicielles. Le

module permettant l'utilisation d'environnement sécurisé est chargé dans l'état Lua maître au démarrage. Le temps nécessaire à ce chargement est de 18μs. Ce temps est négligeable au regard du temps de démarrage de Lynq qui, pour rappel, est de 10ms. Le tableau 3.4 résume les différents temps associés aux actions relatives au management des mécanismes de sécurité. Pour ajouter des règles à un profil, 7μs par règle sont nécessaires. Une fois toutes les règles ajoutées au profil, 13μs sont nécessaires pour ajouter le profil à un environnement d'exécution esclave. De même, lorsque le profil d'un état esclave doit être mis à jour, ce délai est proportionnel au nombre de règles à mettre à jour (7μs par règle à modifier/ajouter et 13μs de mise à jour du *SGE* de l'état Lua esclave). Durant l'exécution, la pénalité introduite par le mécanisme de sandboxing est également directement influencée par le nombre de règles à comparer. Chaque éxécution de code nécessitant une comparaison avec le profil de sandboxing introduit une pénalité moyenne de 1μs. Cette valeur a pu être déterminée par comparaison entre l'exécution d'une application avec et sans profil de sandboxing.

Création		18	
Ajout		13	
Mise à jour		7×N + 13	
	Min	Max	Average
Comparaison	0.7	1.2	1

Tableau 3.4 – Pénalité temporelle introduite par l'utilisation d'un *SGE* en [μ**s**]. La valeur N fait état du nombre de règles

Le tableau 3.5 propose une comparaison des performances avec et sans sandboxing du benchmark Scimark. Pour catégoriser plus précisément les pénalités, trois cas d'études ont été pris en compte. Un premier cas applique des restrictions sur les modules et I/O. Un second applique des restrictions sur les accès mémoire et un dernier combine les deux premiers cas d'étude. Pour implémenter ces restrictions, nous avons utiliser le benchmark en Lua. Tout d'abord nous avons identifié l'ensemble des modules, des fonctions et des variables utilisés par les kernels, puis nous avons procédé en quatre temps. Dans un premier temps, une version sans restrictions a été implémentée, puis dans un second temps l'ensemble des restrictions sur les modules et accès I/O ont été appliquées lors de la seconde implémentation.

Dans un troisième temps nous avons implémenté une version avec l'ensemble des restrictions mémoire identifiées préalablement. Enfin, une version avec toutes les restrictions (modules, accès I/O et mémoire) a été utilisée. Ces dernières sont contrôlées à chaque tentative de l'applicatif d'y accéder. Avant l'analyse des résultats, il est important de noter que les accès mémoires sont les plus pénalisants du fait de l'occurence de leurs contrôles.

	FFT	SOR	MC	SPARSE	LU
(1) Lynq avec LuaJIT 2.0.5 (jit)	47.60	128.29	28.97	22.45	67.93
(2) Lynq avec contrôle d'accès I/O et module	46.91	125.72	28.06	21.87	66.81
(3) Lynq avec contrôle d'accès mémoire	44.98	122.88	27.09	21.56	64.27
(4) Lynq avec contrôle d'accès I/O, module et mémoire	44.31	121.31	26.12	20.91	63.25
(2)/(1) ratio (av. 97.84%)	98.55%	98.00%	96.86%	97.42%	98.35%
(3)/(1) ratio (av. 94.89%)	94.50%	95.78%	93.51%	96.04%	94.61%
(4)/(1) ratio (av. 92.81%)	93.09%	94.56%	90.16%	93.14%	93.11%

TABLEAU 3.5 – Comparaison en **[MFlops]** pour différents niveaux de test d'isolation de la suite de benchmark Scimark exécutée avec LuaJIT 2.0.5 (jit) dont les résultats ont été présentés en Figure 2.7.

Dans un script, l'accès au module est généralement testé une fois en début d'exécution. À ceci s'ajoute des itérations supplémentaires si pendant l'exécution d'un script qui fait appel à d'autre fichier source Lua, d'autres modules sont appelés. Le contrôle de l'accès aux I/O est plus fréquent car en plus d'intervenir lors de toute tentative d'utilisation d'un fichier, il intervient pour l'appel de fonctions. À ce niveau, l'impact du contrôle de ces fonctions dépendra de la granularité de contrôle souhaité. En effet, cette pénalité qui peut être décidée par le gestionnaire du système sera le fruit d'un compromis entre une restriction dure impliquant un test de toutes les fonctions bas niveau et, une restriction plus souple où seul l'accès aux fonctions de haut niveau sera testé. Quant aux accès mémoires, ils interviennent à chaque tentative d'accès à une variable du programme et ils sont donc les plus fréquents. Pour effectuer ces évaluations nous avons repris les benchmarks exécutés dans le chapitre 2 section 2.5.2 avec les mêmes paramètres.

La figure 3.12 expose visuellement les résultats présentés dans le tableau 3.5. La base de comparaison est l'exécution du kernel avec Lynq avec LuaJIT 2.0.5 (jit) sans sandboxing.

Le premier cas d'étude sur la restriction des appels aux I/O et module expose une pénalité moyenne de 2.16% [1.45% - 3.14%]. Les restrictions sur les accès mémoires introduisent une pénalité moyenne de 5.11% [3.96% - 6.09%]. Ces résultats confirment les prévisions concernant l'occurence des contrôles faite précédemment. En effet, la pénalité introduite est proportionnelle aux nombres de contrôles sur les aspects mémoires. Enfin, le dernier cas d'étude nous montre une pénalité moyenne de 7.19% [5.44% - 9.84%].

L'ensemble de ces résultats confirment la proportionnalité de la perte de performance en rapport avec l'occurence des comparaisons au profil de sandboxing.

Ces résultats sont à mettre en relief avec les solutions proposées par Intel avec SGX et ARM avec Trustzone. En effet, l'utilisation d'Intel SGX ne peut pas être directement comparée vis à vis de notre solution car nous ciblons une architecture ARM. De plus

FIGURE 3.12 – Performance en pourcentage [%] pour les 5 noyaux de test : FFT, SOR, MC, SPARSE et LU selon le niveau de restriction appliqué. (2), (3) et (4) représentent les niveaux de tests d'isolations présentés dans le tableau 3.5

Intel SGX a été développé pour des architectures Intel et lors des entrées et sortie des TEE et REE, nos deux solutions ne font pas les mêmes opérations. En se basant sur les travaux de [147], deux résultats à titre de comparaison sont intéressants : entrer ou sortir d'une enclave via les mécanismes de ecall/ocall prend entre 8200 et 17000 cycles horloge. Avec les optimisations proposées par leurs travaux entrer ou sortir d'une enclave via les mécanismes de HotEcall/HotOCall prend entre 620 et 1400 cycles horloge. Soit à la même fréquence d'horloge que nos mesures sur le Cortex-A9 (667MHz) : entre 12.3 μs et 25.5 μs et entre 0.93 μs et 2.1 μs. La solution proposée avec l'utilisation de HotEcall/HotOcall est donc 1.33 à 1.75 fois plus lente que notre solution. Cependant cette comparaison est à regarder au spectre des opérations que chacune des solutions effectue. Là où nous faisons une comparaison avec le *SGE*, la solution HotCalls applique une procédure plus complexe de requête entre le TEE et le REE.

Quant à l'utilisation d'ARM Trustzone, nous nous sommes basés sur les travaux de [148]. Ces résultats ont été obtenu grâce à l'utilisation du framework Open Portable Trusted Execution Environment [149] sur une cible ARM Cortex A53 cadencé à 1.2GHz. Les résultats de l'étude sont les suivants : le temps de bascule du REE au TEE est de 110 μs et le temps de bascule du TEE au REE est de 47 μs. Cependant, de même que pour l'étude des résultats d'Intel SGX, les résultats obtenus sont à pondérer avec le nombre d'opérations qu'effectuent ARM Trustzone lors d'un changement TEE/REE.

3.4 Conclusion

Dans ce chapitre, des améliorations complémentaires à la solution initiale Lynq ont été présentées pour répondre aux problématiques de performance et de sécurité présentées en introduction. La synthèse des contributions originale est présentée ci-dessous :

(i) Lynq Advanced, une couche logicielle qui répond à des problématiques de multiplexage temporel en tirant partie non seulement des performances intrinsèques de chaque bloc, mais également de leur ordonnancement pour proposer une solution semi autonome dans la gestion de l'allocation des ressources du *SoC FPGA* ;

(ii) ISOLynq, une couche logicielle qui propose une solution de confinement pour plusieurs environnements d'exécution afin qu'ils soient isolés, tant au niveau matériel que logiciel, des autres environnements d'exécution en cours sur le *SoC FPGA* ;

4 De la description du microcode jusqu'au *CPU*

L'objectif de ce chapitre est de présenter un outil capable de générer un processeur microcodé. À partir d'une description dans un langage basé sur le langage Lua, il génère d'une part, une description architecturale en langage *HDL* d'un processeur microcodé implémentant les signaux de contrôle relatifs au jeu d'instruction RISC-V et, d'autre part, le microcode permettant l'utilisation des instructions. Pour présenter ces contributions, la première section complétera l'état de l'art du chapitre 1. Elle apportera une présentation du jeu d'instruction RISC-V et décrira le fonctionnement interne des compilateurs. Par la suite, le générateur Atlas sera présenté. Il est capable de générer une description architecturale en verilog d'un *CPU* RISC-V et de son microcode à partir d'une description du microcode à implémenter.

4.1 Introduction

4.1.1 Objectifs

Au-delà de méfaits pouvant être commis par une personne tierce, des problèmes de conception peuvent être à l'origine de bogues compromettant l'intégrité ou le fonctionnement d'un système [150]. Ceci ouvre la porte à des failles pouvant être mise à de mauvais profits. Elles peuvent avoir de graves conséquences sur la sécurité du système. Par exemple l'escalade de privilège, une manoeuvre qui vise à acquérir des droits permettant un accès à des ressources qui sont usuellement non accessible aux utilisateurs non administrateurs. Pour les défauts identifiés, les constructeurs fournissent des solutions alternatives permettant le contournement des bogues en question [151]. Cependant, cette méthode n'est pas la solution à toutes les erreurs de conception qui, quand elles sont plus complexes, nécessitent une prise en compte à plus bas niveau, généralement par la modification du matériel. Le développement d'un processeur nécessitant une description matérielle complexe, le coût de réparation d'une telle erreur de conception est non négligeable, car au delà de la réparation en elle même du bogue, toutes les phases de vérifications fonctionnelles sont à refaire. Pour pallier à cela, une des solutions utilisées par les constructeurs est l'ajout de microcode dans leurs architectures. Cette utilisation du microcode introduit une flexibilité dans le développement d'instruction complexe en plus de permettre la correction de bogue. Elle réduit également drastiquement le coût de telles corrections. Ces mécanismes de mise à jour du microcode sont utilisés par Intel depuis le processeur Pentium Pro P6 datant de 1995 et AMD depuis le processeur K7 en 1999 [152]. Le monde des jeux d'instructions fait état de nombreux standard les définissant. Cependant très peu d'entre eux ont des spécifications ouvertes et permettent l'utilisation par des tiers.

C'est le cas du standard RISC-V, né à UC Berkeley et présenté pour la première fois dans un rapport technique de l'université [13] en 2011. Gagnant très vite en popularité, il a suscité l'adhésion de la communauté et il est devenu un standard incontournable de ces dernières années. Preuve en est la parution des premiers *SoC FPGA*s qui utilise un *CPU* RISC-V : Polarfire de Microsemi [153]. De nombreuses architectures permettant l'implémentation de *CPU* à jeu d'instruction RISC-V existent déjà [154, 155]. Cependant, celles-ci ne sont pas microcodées. Des solutions à des fins éducatives ont également été développées, la plus connue étant la suite Sodor [115], développé par UC Berkeley Architecture Research team. Ceux-ci ont utilisé comme base leurs propre langage, Chisel, afin de décrire des architectures RISC-V possédant différents niveaux de pipeline ainsi qu'une description microcodée. Nous avons donc sélectionné ce standard de jeu d'instruction pour nos travaux relatifs au développement d'un générateur d'architecture de processeur

microcodé. L'objectif de ce chapitre est de proposer un processeur microcodé utilisant le jeu d'instruction RISC-V ainsi que son générateur afin d'être en mesure de générer à la demande et en fonction du microcode voulu, l'architecture matérielle et le microcode adéquat.

4.1.2 Caractéristiques

Les caractéristiques principales de ces contributions sont les suivantes :

- **Générateur d'architecture microcodée** Un générateur d'architecture de processeur RISC-V microcodé a été développé conjointement avec un générateur de microcode. Celui-ci fournit une description architecturale dans le langage Verilog depuis une description du microcode à implémenter.
- **Générateur de microcode** Capable de convertir en signaux de contrôle une description synthétique de l'instruction désirée, le générateur de microcode effectue la traduction et la mise en forme du microcode depuis un outil décrit en Lua.

4.2 Complément à l'état de l'art

En complément du chapitre 1, des informations relatives au standard de jeu d'instruction RISC-V ainsi qu'à l'architecture interne des compilateurs sont apportées dans cette section .

4.2.1 L'architecture du jeu d'instruction RISC-V

L'interface matériel - logiciel est importante dans un système informatique. Elle est définie dans l'architecture du jeu d'instruction, plus communément connu sous l'acronyme *ISA* pour *Instruction Set Architecture*. Ces travaux se focalisent plus particulièrement sur RISC-V, une des dernières nées dans le domaine des ISA. Elle a vu le jour en 2010 dans un groupe de recherche de l'University of California, Berkeley. Depuis, le monde industriel et académique se sont réunis autour d'une fondation pour développer ce qui est devenu un standard en libre accès. L'objectif de ce standard de jeu d'intruction est de pouvoir être utilisé sur le plus grand nombre de cible et pour tous types d'applicatifs. Cet *ISA* a également vocation à être ouvert et accessible à tous sans restriction. Cela a pour conséquences : 1) le standard doit définir une architecture qui soit la plus simple possible pas parti pour des optimisations architecturales sur des cibles particulières. Ceci peut induire des restrictions sur les cibles ou être trop coûteux en ressources pour des cibles ne bénéficiant pas de ces optimisations ; 2) le nombre d'implémentations disponibles de

manière gratuite et open-source à largement contribué à la popularité de cet *ISA* ; 3) les coûts de développement de telles solutions ont également été réduis. La concurrence entre le marché du libre qui propose des implémentations ouvertes et les implémentations propriétaires dynamisant l'innovation de la recherche en microarchitecture. Enfin, 4) les entités qui ont des doutes sur le respect et l'application de certaines normes, à des fins d'espionnage industriel ou d'ingèrences gouvernementales, peuvent adopter un standard ouvert. Cela leur permet, du fait du haut niveau de confiance dont ils ont besoin, d'être en mesure d'implémenter eux mêmes leurs architectures.

Extension	**Nom de l'extension**
Zicsr	Instructions des Registres de Contrôle et Status (CSR)
Zifencei	Instructions de Synchronisation Mémoire
I	Instructions Entières
E	Instructions Entières (pour Systèmes Embarqués)
M	Instructions de Multiplication et Division
A	Instructions Atomiques
F	Instructions Flottantes Simple Précision
D	Instructions Flottantes Double Précision
Q	Instructions Flottantes Quadruple Précision
C	Instructions Compressés

Tableau 4.1 – Extension du standard de jeu d'instruction RISC-V

Lors du développement de ce standard, il a été décidé de proposer différentes extensions du standard permettant l'accès aux instructions par catégorie. Afin de fournir un processeur fonctionnel, l'extension de base I est toujours présente (ou sa variante E). À cela s'ajoutent ensuite les extensions présentées dans le tableau 4.1. Ce format avec extension permet également la mise en œuvre de RISC-V à très faible consommation énergétique, notamment pour des applications embarquées. Il est également possible d'utiliser cet ISA pour implémenter des *CPU* RISC-V spécialisés. Plus généralement, l'approche modulaire offerte par les extensions RISC-V permet d'adapter la compilation d'un programme aux extensions disponibles sur la cible.

La communauté a permis l'adoption de ce nouveau standard en faisant des efforts dans le développement des compilateurs associés, notamment GCC [156] et Clang[157] qui le supportent aujourd'hui tous deux. Le port des systèmes d'exploitations a également été abordé par la communauté [158, 159].

4.2.2 Architecture simplifiée de compilateur

Le code source d'un programme ne peut pas être exécuté tel quel par un processeur car celui-ci n'est en l'état qu'une simple suite de caractères et non une suite d'instructions à exécuter. Ce code source est une retranscription dans un langage spécifique d'un algorithme. Outre le fait de rendre le langage de programmation intelligible pour le programmeur, cette structure dite statique du programme permet une analyse dans les premières étapes de compilation. Un compilateur est un outil qui traduit un programme d'un langage de programmation spécifique appelé code source vers une suite d'instructions exécutables sur l'architecture cible appelé code objet. La première étape consiste à utiliser un Lexer ou Tokenizer pour diviser le code source en sous-parties élémentaires appelées *Token*s. Ceux-ci sont ensuite consommés un par un à l'étape suivante où est utilisé un Parser. Celui-ci possède une connaissance du langage utilisé en entrée et il est responsable de 1) la traduction en structure de donnée de celui-ci, 2) l'identification des erreurs de syntaxe. Il résulte de cette étape un *AST* pour *Abstract Syntax Tree*, traduit en français par *Arbre Syntactique Abstrait* (*ASA*). Celui-ci a pour objectif de donner une structure flexible en entrée du générateur de code. La figure 4.1 présente une chaine de compilation simplifiée.

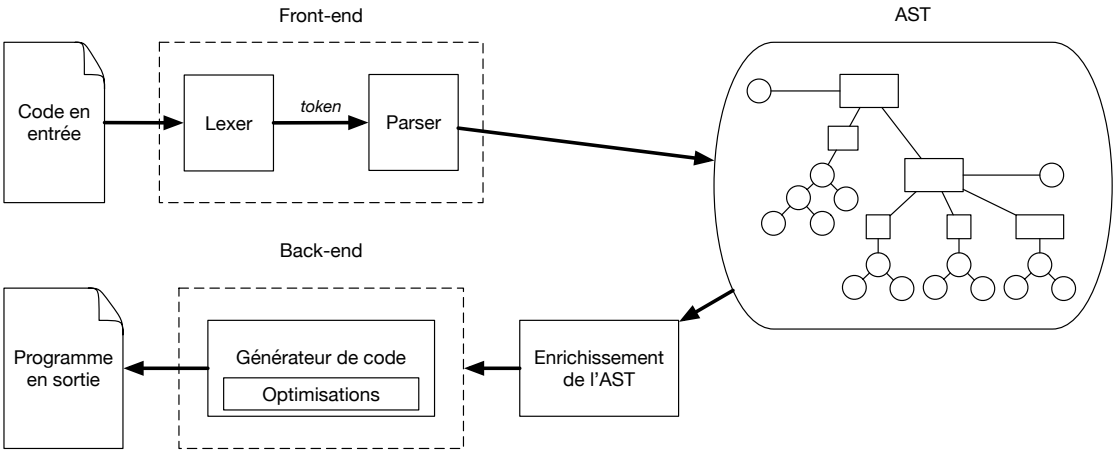


FIGURE 4.1 – Vue interne du compilateur

Le Front-end du compilateur est composé du Lexer (ou Tokenizer) et du Parser. En sortie du Front-End est produit un *AST*. Celui-ci est ensuite enrichi avec des informations complémentaires. Cet enrichissement, dont on ne traitera pas dans la suite de ce document, consiste généralement en deux phases. Une première phase dite d'identification qui consiste à vérifier les régles de portée des identificateurs et à connecter toutes ses occurences. La portée d'un identificateur est la région du programme où celui-ci est actif, celle-ci est décrite visuellement grâce à la figure 4.2. La seconde phase consiste à inférer les types,

vérifier les régles de typage et enrichir les expressions décrites dans l'*AST* avec leurs types. À partir de cet *AST* enrichi, le back-end du compilateur est en mesure de générer le code. Des optimisations peuvent avoir lieu durant cette étape. Nous ne parlerons pas de celles-ci dans la suite de ce chapitre. Elles sont de nature à améliorer le temps d'exécution, mais aussi la taille du programme. Pour être efficace, ces optimisations tirent parti et utilisent des connaissances de l'architecture cible jusqu'aux plus bas niveaux. Ces détails ne sont pas toujours accessibles via un langage de haut niveau pour l'utilisateur.

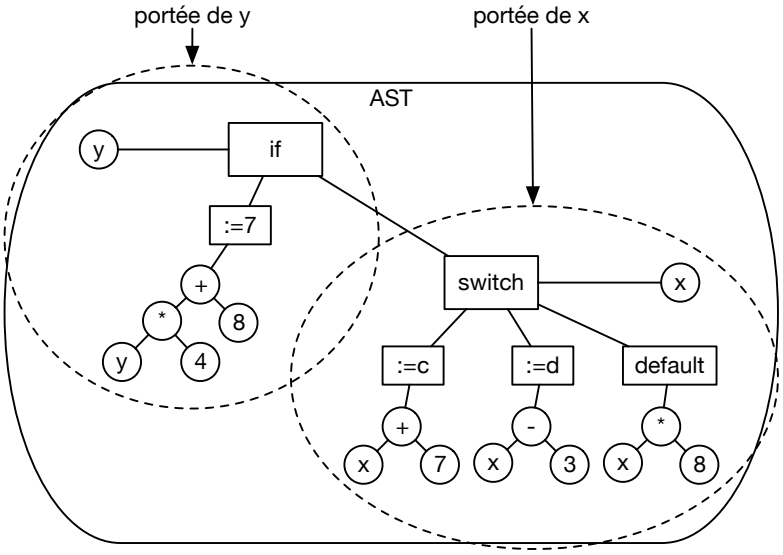


FIGURE 4.2 – Porté d'un identificateur

Au-delà de ces optimisations, la phase de génération a pour objectif de traduire l'*AST* enrichi en un code machine exécutable sur l'architecture cible. Pour présenter le fonctionnement du compilateur, nous allons prendre comme exemple le code source dans le listing ci-dessous.

Listing 4.1 – Un exemple d'algorithme

```
if ( a == 17 )
        x = a * 2 + 1
else
        switch ( y )
        {
                case 1 : x = a - 1 ;
                case 2 : x = a - 10 ;
                default : x = a + 8 ;
        }
```

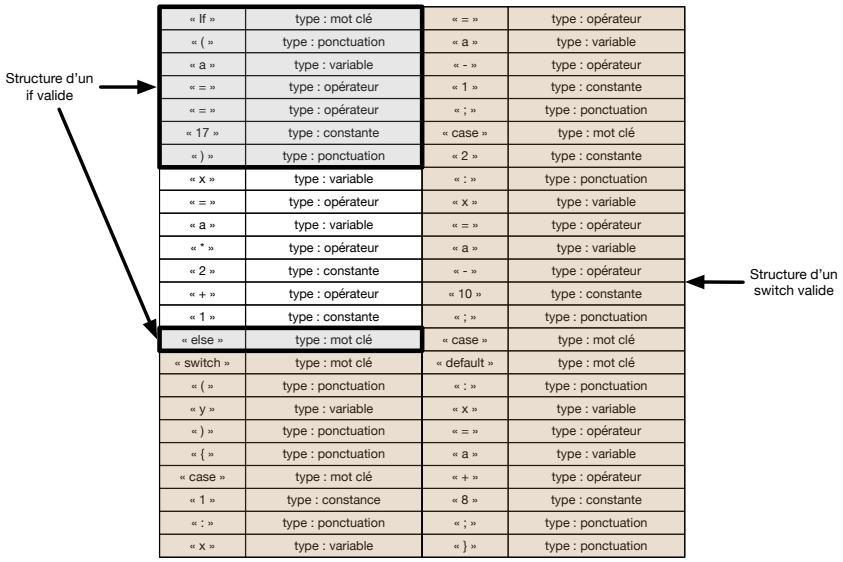
Tout d'abord l'analyse lexicale du code source d'un programme logiciel est effectué par le

« If »	type : mot clé	« = »	type : opérateur
« (»	type : ponctuation	« a »	type : variable
« a »	type : variable	« - »	type : opérateur
« = »	type : opérateur	« 1 »	type : constante
« = »	type : opérateur	« ; »	type : ponctuation
« 17 »	type : constante	« case »	type : mot clé
«) »	type : ponctuation	« 2 »	type : constante
« x »	type : variable	« : »	type : ponctuation
« = »	type : opérateur	« x »	type : variable
« a »	type : variable	« = »	type : opérateur
« * »	type : opérateur	« a »	type : variable
« 2 »	type : constante	« - »	type : opérateur
« + »	type : opérateur	« 10 »	type : constante
« 1 »	type : constante	« ; »	type : ponctuation
« else »	type : mot clé	« case »	type : mot clé
« switch »	type : mot clé	« default »	type : mot clé
« (»	type : ponctuation	« : »	type : ponctuation
« y »	type : variable	« x »	type : variable
«) »	type : ponctuation	« = »	type : opérateur
« { »	type : ponctuation	« a »	type : variable
« case »	type : mot clé	« + »	type : opérateur
« 1 »	type : constance	« 8 »	type : constante
« : »	type : ponctuation	« ; »	type : ponctuation
« x »	type : variable	« } »	type : ponctuation

FIGURE 4.3 – Décomposition du code source en *Token*s

Lexer. Cette étape consiste à séparer le code source, vu initialement comme une chaine de caractère, en une liste d'éléments appelés *Token*s. Plus spécifiquement, le lexer est chargé de mettre en forme le code source pour qu'il soit utilisé par le reste du compilateur. Celui-ci décompose donc en *Token*s les éléments constituant le code en entrée du compilateur. C'est la plus petite entité que le parser peut comprendre. Ces *Token*s contiennent les noms de variables, les constantes, les mots-clés ainsi que les signes de ponctuation. La figure 4.3 présente le résultat de l'analyse lexicale avec l'ensemble des *Token*s identifiés et typés selon leur nature : mot clé, ponctuation, variable, opérateur et constante.

Ensuite, l'analyse grammaticale des *Token*s est faite par le parser, c'est une étape qui analyse les *Token*s pour en déterminer le sens. Par analogie à la langue française, le lexer travaille pour déterminer les mots constituant une phrase et le parser en fait une analyse grammaticale pour en déterminer le sens. Un exemple est présenté à la figure 4.4. Basé sur l'exemple présenté dans le listing 4.1, les *Token*s en sortie du lexer sont analysés et il en ressort la structure d'instructions conditionnelles if et switch. La figure 4.5 présente un exemple d'*AST* pouvant résulter du front-end du compilateur pour le même exemple que présenté précédement. L'*AST* est un arbre de représentation de la structure statique d'un programme dont les différents noeuds et liens représentent les différents opérateurs et arguments présents.



« If »	type : mot clé	« = »	type : opérateur
« (»	type : ponctuation	« a »	type : variable
« a »	type : variable	« - »	type : opérateur
« = »	type : opérateur	« 1 »	type : constante
« = »	type : opérateur	« ; »	type : ponctuation
« 17 »	type : constante	« case »	type : mot clé
«) »	type : ponctuation	« 2 »	type : constante
« x »	type : variable	« : »	type : ponctuation
« = »	type : opérateur	« x »	type : variable
« a »	type : variable	« = »	type : opérateur
« * »	type : opérateur	« a »	type : variable
« 2 »	type : constante	« - »	type : opérateur
« + »	type : opérateur	« 10 »	type : constante
« 1 »	type : constante	« ; »	type : ponctuation
« else »	type : mot clé	« case »	type : mot clé
« switch »	type : mot clé	« default »	type : mot clé
« (»	type : ponctuation	« : »	type : ponctuation
« y »	type : variable	« x »	type : variable
«) »	type : ponctuation	« = »	type : opérateur
« { »	type : ponctuation	« a »	type : variable
« case »	type : mot clé	« + »	type : opérateur
« 1 »	type : constance	« 8 »	type : constante
« : »	type : ponctuation	« ; »	type : ponctuation
« x »	type : variable	« } »	type : ponctuation

FIGURE 4.4 – Interprétation des *Token*s par le parser

Enfin, le générateur de code transforme enfin l'*AST* en code objet (instruction assembleur) pour l'architecture cible. Durant cette étape, le générateur choisi les instructions dans l'ISA ciblé, leurs ordonnancements ainsi que l'allocation des registres.

4.3 Le générateur d'architecture RISC-V et microcode associé

L'objectif est de décrire une association architecture microcodé/microcode à partir des instructions ciblées. Pour cela, il faut être en mesure de décrire les instructions à

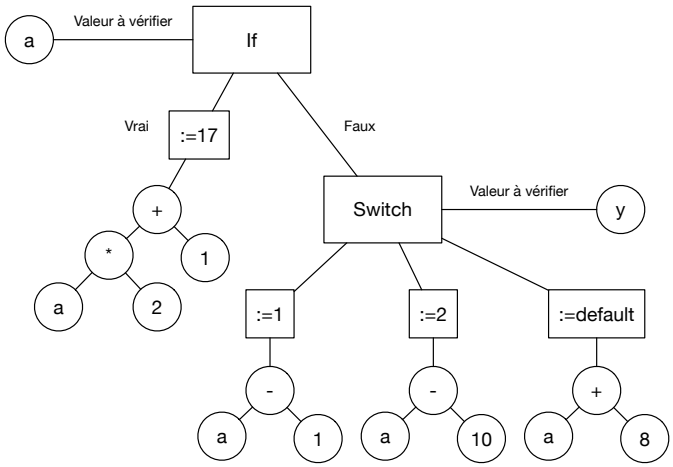


FIGURE 4.5 – Décomposition de l'algorithme du listing 4.1 en *AST*

implémenter. À cette fin, nous avons développé un langage pour décrire les instructions microcodés ainsi que l'ensemble de la chaine de compilation : lexer, parser et générateur de code. Les parties suivantes décrivent le langage, le lexer puis le parser et enfin le générateur de code.

4.3.1 Le langage

Décrire des instructions assembleurs afin qu'elles puissent être décomposées en microcode puis exécutées depuis une architecture microcodé nécessite une connaissance du jeu d'instruction. Le langage doit être doté d'une expressivité qui permet, lors de la phase de compilation, de distinguer les éléments propres à l'architecture de ceux relevant des signaux de contrôle, le microcode. Il est donc nécessaire de permettre au langage de décrire des aspects architecturaux via la description des signaux nécessaire au contrôle ainsi que des aspects fonctionnels via le microcode associé.

var pc : register<31 :0> = 0	Déclaration du registre pc sur 32 bits.
var ir : register<31 :0> = 0	Déclaration du registre ir sur 32 bits.
var a : register<31 :0> = 0	Déclaration du registre a sur 32 bits.
var b : register<31 :0> = 0	Déclaration du registre b sur 32 bits.
var imem : register<31 :0><0 :511> = {}	Déclaration du registre imem sur 512x32 bits.
var rf : register<31 :0><0 :31> = {}	Déclaration du registre pc sur 32x32 bits.
var rs1 : signal< 4 :0> = ir<19 :15>	Déclaration du signal rs1 sur 5 bits (les bits 19 à 15 de ir).
var rs2 : signal< 4 :0> = ir<24 :20>	Déclaration du signal rs2 sur 5 bits (les bits 24 à 20 de ir).
var rd : signal< 4 :0> = ir<11 :7>	Déclaration du signal rd sur 5 bits (les bits 11 à 7 de ir).

Tableau 4.2 – Déclaration de signaux internes

Notre langage est capable de décrire d'une part, les signaux internes du processeur utilisés pour la génération de l'architecture et d'autre part, les instructions qui utilisent ces signaux internes pour la génération du microcode. Les tableaux 4.2 et 4.3 via un exemple de code source pour la description de signaux internes (tableau 4.2) et la description d'instructions (tableau 4.3) présentent les fonctionnalités de notre langage.

Dans le tableau 4.2 nous avons défini un exemple de déclaration de signaux internes. La figure 4.6 en présente la syntaxe. Cette déclaration se compose de trois parties. Une première partie permet de nommer le signal, une seconde en permet son dimensionnement et enfin la dernière partie l'initialise à sa valeur par défaut. La déclaration des instructions suit également une syntaxe qui est présenté à la figure 4.7. Elle se compose aussi de trois parties. Une première partie permet le nommage de l'instruction. Une seconde consiste à décrire les microinstructions. Elle peut avoir une taille variable en fonction de la complexité de l'instruction à décrire. Enfin, la dernière partie consiste

def fetch	Déclaration de l'instruction fetch
f0 : ir ← imem[pc] ;	Récupération de l'instrution assembleur à exécuter dans la mémoire instruction à l'adresse pc ;
f1 : a ← pc ;	pc est positionné dans l'opérateur a de l'alu ;
f2 : pc ← alu(a+4) ; dispatch ;	pc récupére la valeur a+4 et dispatch
def add	Déclaration de l'instruction add
a0 : a ← rf[rs1] ;	Le registre rs1 est positionné dans l'opérateur a de l'alu ;
a1 : b ← rf[rs2] ;	Le registre rs2 est positionné dans l'opérateur b de l'alu ;
a2 :rf[rd] ← alu(a+b) ; goto f0 ;	Positionne dans rd le résultat de a+b et envoie pc vers l'instruction de fetch.

TABLEAU 4.3 – Deux exemples de déclaration d'instructions

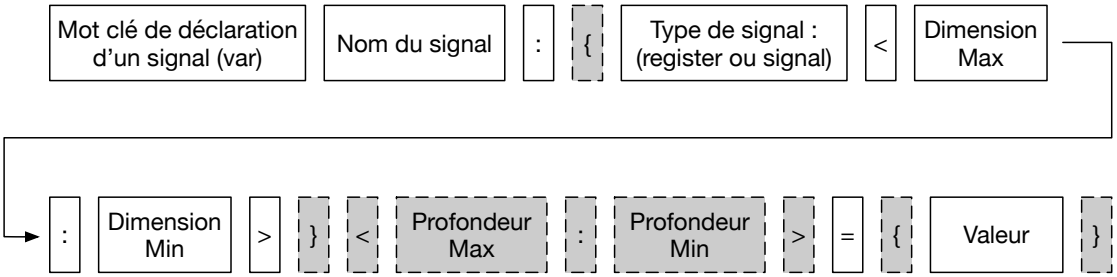


FIGURE 4.6 – Déclaration d'un signal

en une instruction pour le compteur programme. Par exemple pour la déclaration de l'instruction add présenté dans le tableau 4.3, l'instruction add est nommée, puis elle est définie en trois lignes et enfin le compteur programme est renvoyé à l'adresse de l'état f0.

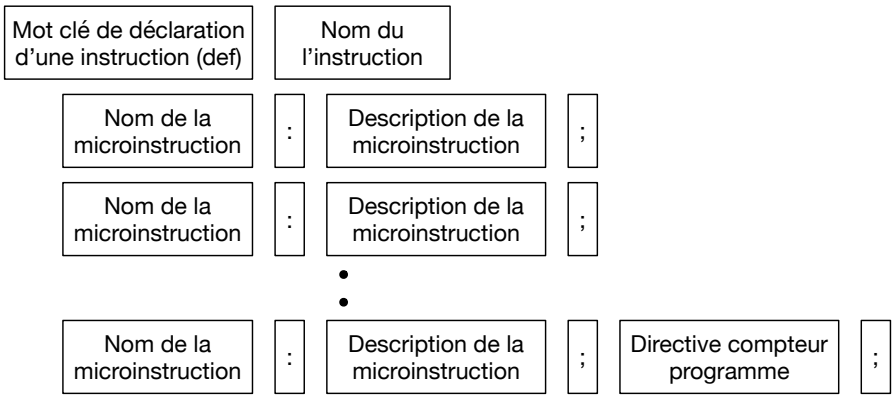


FIGURE 4.7 – Déclaration d'une instruction

4.3.2 Le compilateur de microcode : Atlas

À partir d'une description du microcode, nous avons utilisé comme base Titan [160] et Pallene [161] (un compilateur pour le langage de programmation Pallene, basé sur Titan) pour analyser le code source et générer l'architecture *HDL* ainsi que le microcode associé. Ce sont des compilateurs spécialement développés pour être utilisé avec des langages qui intéragissent avec Lua. Nous avons adapté le compilateur écrit en Lua et l'avons enrichi afin qu'il soit en mesure de prendre en entrée du code source dans notre langage.

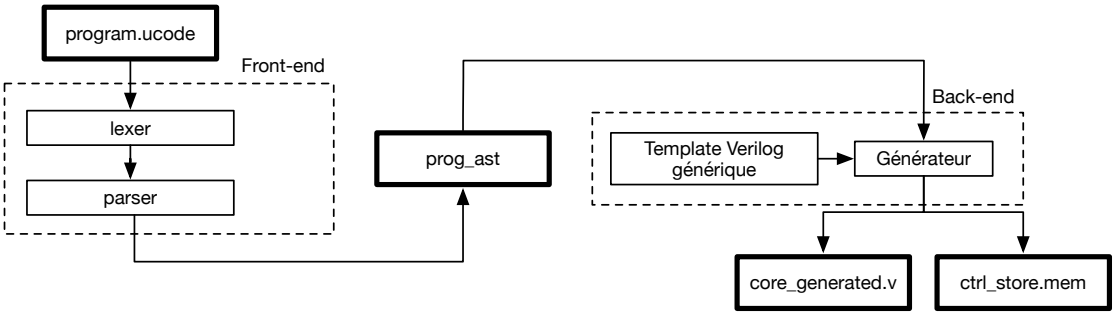


FIGURE 4.8 – Séquence de compilation

La séquence de génération est présentée figure 4.8. Elle débute avec, en entrée, un fichier source contenant une description du microcode que l'utilisateur veut implémenter et elle se termine avec la génération de deux fichiers : un premier qui décrit l'architecture du processeur et un second fichier mémoire qui contient le microcode. Le front-end démarre avec le lexer pour séparer le code source en une chaine de caractère, elle n'a pas été modifié. Cependant l'étape d'identification est enrichie avec les éléments relatifs à notre langage. À ce titre, l'enrichissement grammatical consiste à l'ajout de symboles et de mots-clés propres à la structure de notre langage.

Le tableau 4.4 recense l'ensemble des mots-clés initialement définis dans Pallene ainsi que les modifications que nous avons apportées. Le tableau 4.5 fait de même avec les symboles. Concernant le tableau 4.4, les mots-clés qui ne sont pas présents dans Atlas apparaissent en italique (*any, local, type*). Les mots-clés ajoutés se reconnaissent en gras (**def, dispatch, let, register, signal, value, var**). Ils permettent d'apporter de nouvelles fonctionnalités. Comme énoncé dans la description du langage, **[def, var]** permettent la définition d'un signal ou d'une instruction ; **[signal, register]** donnent lieu au typage d'un signal.

Le tableau 4.5 quant à lui expose les modifications apportées à la signification des symboles dans Atlas. On peut noter trois modifications : l'ajout du symbole ADDIF qui permet la définition de microinstruction conditionnelle en utilisant l'ALU ; l'ajout du

Mots-clés	Pallene	Atlas	Mots-clés	Pallene	Atlas	Mots-clés	Pallene	Atlas
any	•	○	for	•	•	**register**	○	•
and	•	•	function	•	•	repeat	•	•
as	•	•	goto	•	•	return	•	•
boolean	•	•	if	•	•	**signal**	○	•
break	•	•	import	•	•	string	•	•
def	○	•	in	•	•	then	•	•
dispatch	○	•	integer	•	•	true	•	•
do	•	•	**let**	○	•	*type*	•	○
else	•	•	*local*	•	○	until	•	•
elseif	•	•	nil	•	•	**value**	○	•
end	•	•	not	•	•	**var**	○	•
false	•	•	or	•	•	while	•	•
float	•	•	record	•	•			

TABLEAU 4.4 – Mots-clés définis dans Pallene et Atlas

Symboles	Pallene	Atlas	Symboles	Pallene	Atlas	Symboles	Pallene	Atlas
ADD	+	+	SHL	<<	<<	LBRACKET	[	[
SUB	−	−	SHR	>>	>>	RBRACKET	]	]
MUL	*	*	CONCAT	..	..	LCURLY	{	{
MOD	%	%	EQ	==	==	RCURLY	}	}
DIV	/	/	LT	<	<	SEMICOLON	;	;
IDIV	//	//	GT	>	>	COMMA	,	,
POW	^	^	NE	~=	~=	DOT	.	.
LEN	#	#	LE	<=	<=	DOTS	...	...
BAND	&	&	GE	>=	>=	DBLCOLON	::	::
BXOR	~	~	**ASSIGN**	**=**	**<-**	COLON	:	:
BOR	‖	‖	LPAREN	(	(	RARROW	− >	− >
ADDIF		**+?**	RPAREN	)	)	**DECLARE**		**=**

TABLEAU 4.5 – Symboles définis dans Pallene et Atlas

symbole DECLARE matérialisé par l'utilisation du symbole =; et enfin, la modification du symbole ASSIGN de = vers <-.

La figure 4.9 présente la décomposition en *Token*s de l'exemple issu du tableau 4.2 et 4.3. On y retrouve la décomposition du code source en *Token*s et l'association *Token* / type de *Token*.

Suite à la décomposition en *Token* du code source, le parser génére l'*AST*. Le parser d'Atlas est modifié afin de pouvoir analyser les *Token*s spécifiques suite à l'enrichissement de la grammaire présentée précédement.

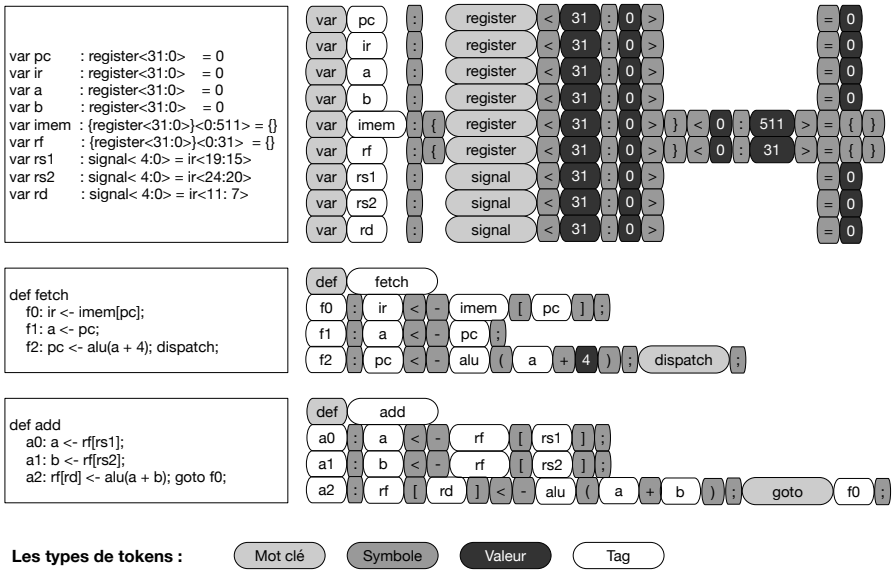


FIGURE 4.9 – Fonctionnement de notre Lexer

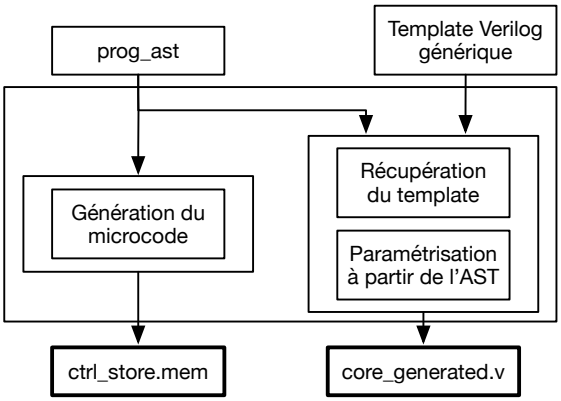


FIGURE 4.10 – Back-end du compilateur

Le back-end du compilateur d'Atlas génère la description architecturale en Verilog ainsi que le microcode associé à partir de l'*AST*. La figure 4.10 présente les étapes de la génération. Celle-ci se décompose en deux séquences parallèles. Concernant la génération de la description *HDL*, une description Verilog générique du *CPU* RISC-V est utilisé et celle-ci est paramétrée en fonction des données de l'*AST*. Ceci avec pour objectif d'ajouter les ressources logiques nécessaires à l'implémentation de l'instruction. Par exemple, pour chaque opérateur d'ALU (add, sub, or, xor, and, etc) détecté dans l'*AST*, le matériel nécessaire à son implémentation est ajouté comme paramètre au template. Car, dans ce cas précis, le signal de contrôle issue du microcode viendra commander l'utilisation de l'opération désirée. Cependant celle-ci doit être disponible sur l'architecture matérielle. La génération du microcode suit le même principe et un fichier binaire qui contient

l'ensemble des microinstructions décrites initialement est généré.

4.3.3 L'architecture et le microcode associé

Dans cette partie, nous allons présenter l'architecture du processeur RISC-V générée à partir du compilateur précédemment présenté ainsi que le microcode associé. La génération de l'architecture *HDL* à partir d'un template permet au développeur de microcode de se concentrer directement sur les instructions à implémenter. Le template contient déjà les éléments de base de l'architecture.

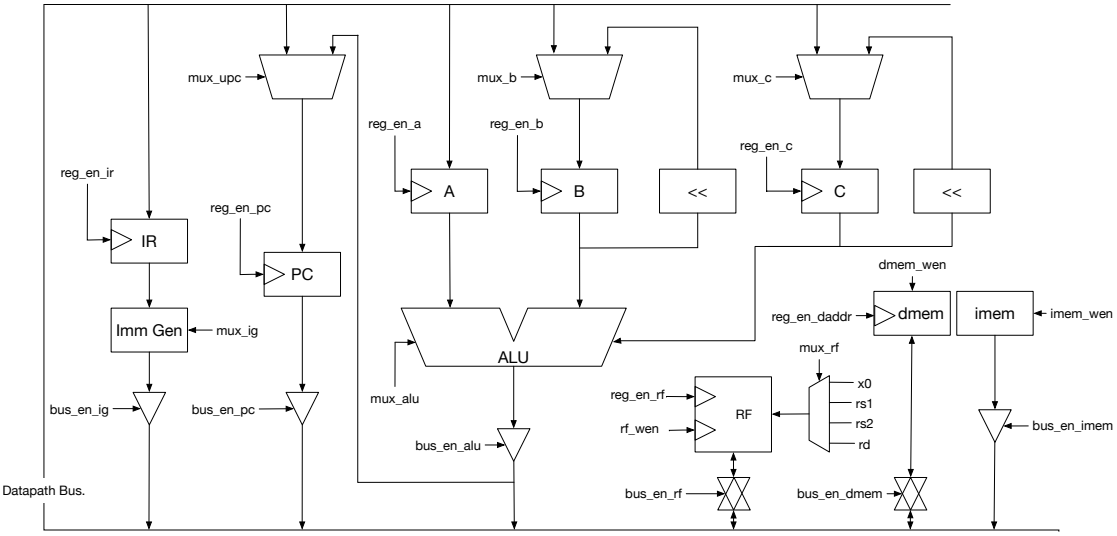


FIGURE 4.11 – Architecture du processeur RISC-V

La figure 4.11 présente l'architecture du processeur capable d'implémenter les instructions de l'extension I du jeu d'instruction RISC-V présenté dans le tableau 4.6.

Types d'instruction	Instructions
Instruction de fetch*	FETCH
Instruction Immédiat à Registre	ADDI, SLTI, SLTIU, ANDI, ORI, XORI, SLLI, SLRI, SRAI, AUIPC, LUI
Instruction Registre à Registre	ADD, SLT, SLTU, AND, OR, XOR, SLL, SRL, SRA, SUB
Instruction de Chargement	LB, LH, LW, LBU, LHU
Instruction de Stockage	SB, SH, SW
Opération de saut inconditionnel	JAL, JALR
Opération de branchement conditionnel	BEQ, BNE, BLT, BLTU, BGE, BGEU

* → Ne fais pas officiellement partie de l'extension I.

TABLEAU 4.6 – Liste des instructions que peut implémenter notre processeur

Cette architecture est composée d'un bus de données ainsi que de l'ensemble des signaux de contrôle qui sont commandés grâce au microcode. On retrouve également les éléments constitutifs d'un *CPU*, à savoir l'*ALU*, le compteur de programme et les registres. Les

mémoires de donnée et d'instructions sont séparées car cette architecture suit celle d'Harvard. Le tableau 4.7 présente la décomposition en signaux de contrôle associés aux instructions fetch et add.
Dans les figures 4.12, 4.13 et 4.14, nous allons reprendre l'instruction add lors des trois phases de son exécution.

	bus_en_alu	bus_en_rf	bus_en_imem	bus_en_pc	bus_en_ig	bus_en_dmem	imem_wen	dmem_wen	rf_wen	reg_en_pc	reg_en_ir
f0	-	-	1	-	-	-	-	-	-	-	1
f1	-	-	-	1	-	-	-	-	-	-	-
f2	1	-	-	-	-	-	-	-	-	1	-
a0	-	1	-	-	-	-	-	-	-	-	-
a1	-	1	-	-	-	-	-	-	-	-	-
a2	1	-	-	-	-	-	-	-	1	-	-

	reg_en_a	reg_en_b	reg_en_c	reg_en_daddr	reg_en_rf	mux_upc	mux_b	mux_c	mux_rf	mux_ig	mux_alu
f0	-	-	-	-	-	upc_plus_one	-	-	-	-	-
f1	1	-	-	-	-	upc_plus_one	-	-	-	-	-
f2	-	-	-	-	-	instr_addr	-	-	-	-	plus_a_4
a0	1	-	-	-	-	upc_plus_one	-	-	rs1	-	-
a1	-	1	-	-	-	upc_plus_one	data_bus	-	rs2	-	-
a2	-	-	-	-	1	addr_f0	-	-	rd	-	plus_a_b

TABLEAU 4.7 – Microcode des instructions fetch et add

La figure 4.12 présente les signaux de contrôle actifs durant la première microinstruction de l'instruction add. Elle positionne le contenu du registre rs1 en entrée de l'ALU dans l'opérande A.

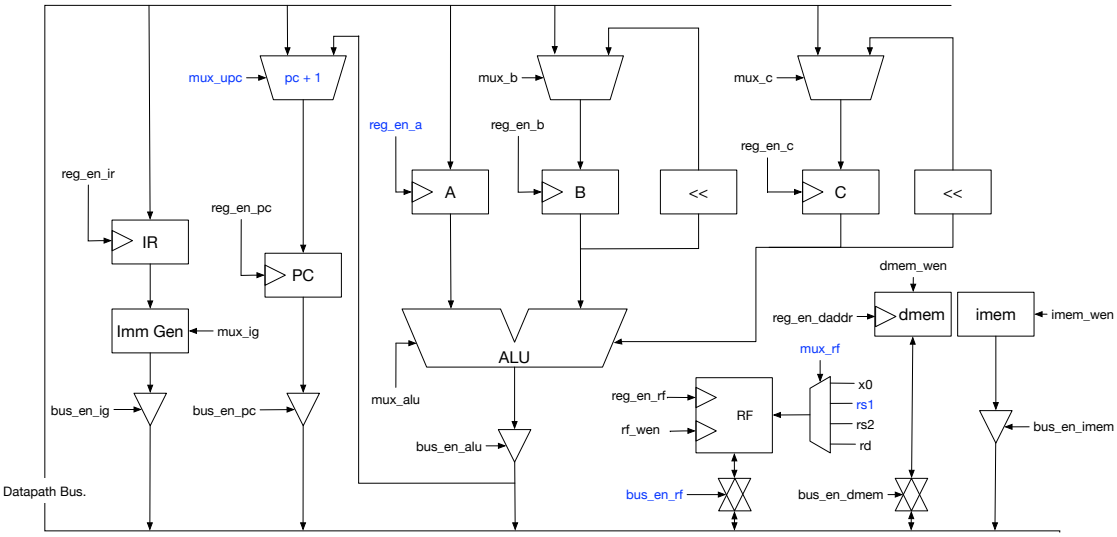


FIGURE 4.12 – Signaux de contrôle actif durant a0

La figure 4.13 présente la seconde microinstruction de l'instruction add. Celle-ci est similaire à a0. Elle positionne le contenu du registre rs2 dans l'opérande B de l'ALU.

La figure 4.14 présente la dernière microinstruction en charge du positionnement du résultat de l'ALU dans le registre sélectionné. Elle renvoie également PC à l'adresse de Fetch.

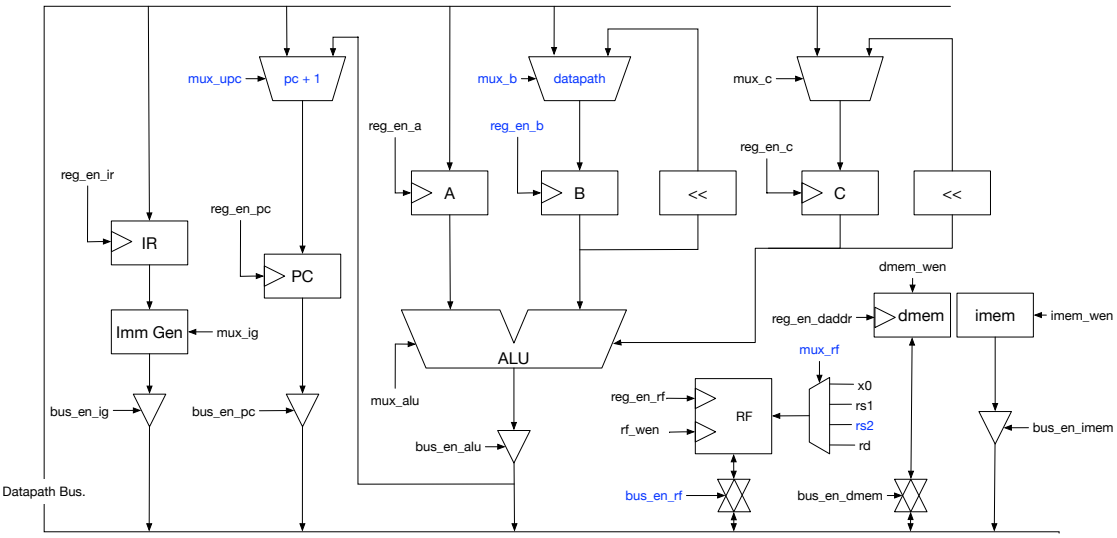


FIGURE 4.13 – Signaux de contrôle actif durant a1

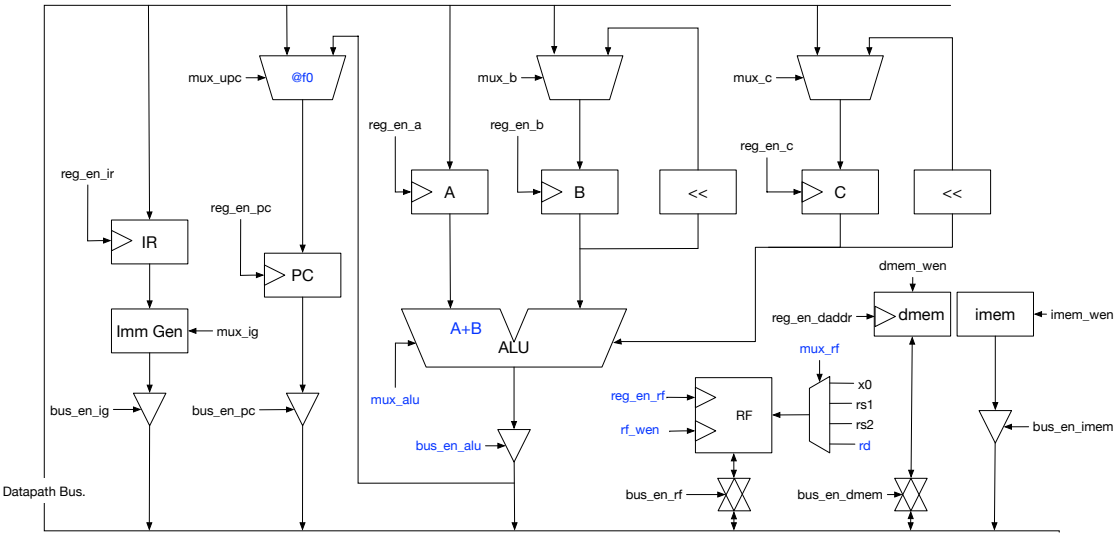


FIGURE 4.14 – Signaux de contrôle actif durant a2

4.3.4 Évaluation de l'architecture microcodée

Pour valider le fonctionnement de notre architecture microcodé, nous l'avons dans un premier temps simulée à partir des outils de la suite Vivado. Pour faire cela, nous avons généré la mémoire instruction du processeur. À ce titre, nous avons développé un outil en Lua qui prend en entrée des instructions assembleurs et fournit en sortie un fichier binaire qui peut être chargé à l'initialisation en mémoire instruction du processeur.

Ensuite nous avons implémenté notre *CPU* sur cible *SoC FPGA* Zynq-7020. Les ressources utilisés sont présentées dans le tableau 4.8. Les valeurs des mémoires de

données et d'instructions ainsi que le microcode sont stockés en BRAM. Les LUT et DFF consommés définissent l'architecture du processeur en incluant les signaux de contrôles, les registres internes et le microcode.

	LUT	DFF	BRAM18k	DSP
Capacité du *FPGA*	53200	106400	140	220
Notre *CPU* RISC-V microcodé	418 (0.79%)	1193 (1.12%)	1 (0.71%)	0
PicoRV32 (regular)	917 (1.72%)	583 (0.55%)	0	0

Tableau 4.8 – Ressources matérielles utilisées par le *CPU* RISC-V généré sur une cible *SoC FPGA* à base de Zynq-7020

Pour évaluer les performances de notre *CPU* microcodé, il était important de le comparer avec un processeur câblé afin d'avoir un aperçu de la perte de performances engendrée par l'utilisation d'une architecture microcodée. Pour cela, nous effectuons une comparaison avec le PicoRV32 [155] dans sa version *regular*. Cette version du PicoRV32 est la version fournie par défaut qui implémente les extensions I, M et C. Plusieurs facteurs expliquent que nos ressources utilisées soient plus faibles. Premièrement, le PicoRV32 est câblé, ce qui signifie que toutes les instructions sont définies en logique. Ceci contrairement à notre implémentation dont l'architecture est plus généraliste et où les instructions sont spécifiées via le microcode. Ensuite, il implémente des interfaces supplémentaires car il a vocation à être intégré dans un *SoC*. Enfin, ce *CPU* implémente les extensions du jeu d'instruction RISC-V I, M et C là où nous n'implémentons que les instructions de l'extention I (sans les instructions relatives aux registres de contrôle). Notre *CPU* fonctionne à une fréquence de 178.57MHz post placement et routage. Le PicoRV32 sur la même cible propose une fréquence de fonctionnement de 196MHz. Les fréquences de fonctionnement ne peuvent cependant pas être prises telles quelles pour juger de la vitesse de fonctionnement du *CPU*. En effet, l'utilisation d'une architecture microcodée impose une décomposition des instructions en plusieurs cycles d'horloge ainsi qu'une instruction de fetch avant chaque nouvelle instruction. Pour comparer notre implémentation microcodée au PicoRV32, nous avons utilisé la suite de test pour l'ISA RISC-V extention I d'Andrew Watterman [8], un des fondateurs de l'ISA RISC-V. Le tableau 4.9 reprend l'ensemble des temps d'exécution des tests pour chaque instruction assembleur.

La moyenne de l'amélioration du temps d'exécution sur un PicoRV32 comparée à notre architecture est de 1.84 fois [1.66 - 2.19]. Les meilleures performances du PicoRV32 sont expliquées par deux points : la fréquence d'horloge du *CPU* est supérieure ; le nombre de cycle par instruction (CPI) des instructions câblées est inférieur à celui de "instruction de fetch + instruction". Le PicoRV32 est donc avantagé pour des instructions de type

Instruction	Architecture microcodée	PicoRV32	Instruction	Architecture microcodée	PicoRV32
ADD	7.83	4.27	LB	4.99	3.01
AND	7.83	4.27	LH	4.99	3.01
OR	7.83	4.27	LW	5.97	2.99
XOR	7.83	4.27	LBU	5.00	3.02
SUB	7.83	4.27	LHU	5.00	3.02
SLT	8.16	4.35	SB	8.79	5.29
SLTU	8.18	4.39	SH	8.79	5.29
SLL	8.54	4.61	SW	8.71	5.24
SRL	8.54	4.61	JAL	1.58	0.81
SRA	8.50	4.57	JALR	2.88	1.40
ADDI	3.98	1.82	BEQ	7.39	4.20
ANDI	3.98	1.82	BNE	7.44	4.22
ORI	3.98	1.82	BLT	7.38	4.19
XORI	3.98	1.82	BLTU	7.40	4.21
AUIPC	1.19	0.67	BGE	8.34	4.74
LUI	1.68	0.98	BGEU	8.38	4.75

TABLEAU 4.9 – Temps d'exécution des tests de la suite [8] en μ**s**

instruction registre / registre ou instruction registre / immédiat avec respectivement des nombres de cycles par instruction de 3 contre 6 et 4 contre 6. L'écart est moins important pour les instructions de chargement et stockage. Cette constatation coïncide avec le nombre de cycle par instruction 33% supérieur (6 contre 8).

Cependant, ces pertes de performances dues à l'utilisation d'une architecture microcodée sont à mettre en relief avec l'intérêt d'une telle architecture lorsque la flexibilité est recherchée. Un compromis quant à l'utilisation de celle-ci semble avoir été adopté par le marché industriel avec l'utilisation du microcode dans les processeurs pour un nombre restreint d'instruction répondant à des critères de complexité ou d'évolutivité identifiés en amont de la phase de production.

4.4 Conclusion

Dans ce chapitre nous avons présenté un compilateur capable, à partir de la description dans le langage de haut niveau Lua des instructions à implémenter, de générer une architecture *HDL* d'un *CPU* RISC-V en complétant un template ainsi que le microcode associé. Les principales contributions de ce conmpilateur sont :

(i) Un langage qui permet la description de signaux de contrôle et d'instruction ;

(ii) Un outil de compilation capable de générer :

— Dans un langage *HDL* l'architecture d'un *CPU* RISC-V microcodé ;

— Le microcode permettant la gestion des signaux de contrôles du *CPU*.

(iii) Un outil de décomposition d'assembleur RISC-V en mémoire instruction.

À titre d'ouverture, deux objectifs à long terme ont été identifiés. Tout d'abord, l'étude de l'impact d'un malware introduit dans une architecture microcodé via la modification de son microcode. Ensuite, avec l'aspect communautaire de l'ISA RISC-V, l'utilisation de microcode peut être une solution rapide pour tester l'efficacité de nouvelles instructions en vue d'une proposition d'extension pour ajout au standard RISC-V.

Conclusions et perspectives

Les architectures hétérogènes reconfigurables, du plus bas niveau matériel jusqu'à l'applicatif sont au coeur de toutes les contributions présentées dans ce document. En particulier, l'utilisation de LuaJIT apporte une solution bas coût pour la gestion d'un environnement d'exécution capable d'offrir aux développeurs d'applications une couche d'abstraction de l'architecture. Dans le cadre des expérimentations qui visent à caractériser le gain de cette solution, les performances calculatoires et énergétiques tendent à ouvrir la route à une utilisation de Lynq comme base pour le développement de solutions complémentaires. Outre la capacité à contrôler les ressources constitutives d'un *SoC FPGA*, à savoir le *CPU*, l'interface mémoire et les contrôleurs pour la partie PS et, le *FPGA* pour la partie PL, la question de l'amélioration de leur rendement s'est posé.

A ce titre, il est important de rappeler le fil rouge de l'ensemble des travaux présentés dans ce manuscrit. En effet, les propositions devaient être indépendantes du système d'exploitation pour obtenir une solution ayant une empreinte mémoire la plus faible possible.

L'intérêt des applicatifs pour architectures reconfigurable hétérogènes est de faire usage des ressources matérielles et logicielles proposées par la cible, de manière concurrente ou parallèle. Dans le cadre d'un usage concurrent des ressources du *SoC FPGA*, une partie de celle-ci n'est pas utilisée à chaque instant et peut être mise à la disposition d'un autre applicatif. Cependant, l'utilisation de la solution Lynq ne permet pas en l'état d'allouer des ressources en mesure de fournir cela. La suite de nos travaux s'est donc concentré sur une méthode de multiplexage temporel ayant pour objectif d'améliorer le rendement de ces ressources. Exclusivement développé en Lua et basé sur Lynq, Lynq ADvanced a tenu compte des spécificités induites par l'hétérogénéité de l'architecture cible. Grâce à l'adaptation d'un algorithme d'ordonnancement pour l'utilisation de coroutines, cette

méthode propose une alternative à l'utilisation d'un système d'exploitation pour le multiplexage temporel de ressources hétérogènes.

La faculté pour un environnement d'exécution d'être utilisé par des applicatifs induit une augmentation du risque pour chaque application, vue séparément, de pouvoir être perturbé par l'exécution des autres. Que ce soit de manière intentionnelle avec l'utilisation d'une application spécialement développée pour cela, ou de manière non intentionnelle avec une erreur de codage pouvant emmener à un bogue compromettant l'intégrité du système, l'état de l'art n'apportait pas de contre-mesure garantissant la sécurité de notre environnement d'exécution. Pour répondre à cela, ISOLynq se base sur Lynq et apporte une méthode qui offre une séparation en environnement d'exécution distinct de chaque applicatifs devant être exécuté. Cette solution introduit une notion d'environnement d'exécution maitre / esclave et met à disposition des primitives de sandboxing qui complètent l'isolation naturelle qu'offre la séparation des applicatifs dans différents environnements d'exécution.

Lynq Advanced et ISOLynq peuvent être vus comme des éléments complémentaires à la solution Lynq dont ils dépendent. Ces trois solutions profitent de la structure du langage de programmation Lua et des apports de LuaJIT. Il serait intéressant de faire bénéficier l'écosystème Lynq au *SoC FPGA* de chez IntelFPGA car la majorité des *SoC FPGA* de ce constructeur utilisent des processeurs ARM. De la même manière, le portage sur processeur Intel semble dès à présent possible, LuaJIT étant compatible nativement avec les architectures x86 et x86_64. Il est également envisageable de bénéficier de l'écosystème Lynq sur processeur RISC-V avec des implémentations de Lua compatibles. Cependant à l'heure actuelle il est impossible d'utiliser LuaJIT et donc de bénéficier des performances calculatoires et de la fonctionnalité FFI offertes. Le portage est néanmoins techniquement possible et il consisterait principalement à modifier la partie en charge de la conversion d'une représentation intermédiaire en code assembleur de LuaJIT.

Cet écosystème est complété par un serveur web qui permet le contrôle du *SoC FPGA* depuis un ordinateur distant. L'administration de plusieurs *SoC FPGA* via cet outil permettrait un déploiement à plus grande échelle de notre solution. Ceci est partiellement fonctionnel à l'heure actuelle. Cela nous donnerais la capacité d'exécuter des applicatifs en utilisant des ressources externes au *SoC FPGA* mais présentes sur un réseau. Cette externalisation du traitement impliquerait également un enrichissement de Lynq pour la partie gestion des ressources et une modification de Lynq ADvanced pour l'allocation dynamique de celles-ci. De même, un protocole réseau adapté devrait être mis en place. ISOLynq n'ayant pas été pensé pour une utilisation sur plusieurs cibles, une refonte de celui-ci semble nécessaire.

Un dernier axe de recherche étudié dans ce manuscrit concerne la génération de la description architecturale d'un *CPU* RISC-V microcodé et de son microcode associé. Ces travaux trouvent leurs motivations dans 1) le souhait d'apporter une solution de génération d'architecture à partir des instructions à implémenter ; 2) la nécessité d'avoir une base pour mener des travaux exploratoires sur l'impact de l'insertion de trojan dans le microcode sur une architecture *CPU* microcodée. Ce deuxième point n'a pas été présenté dans ce manuscrit. Cet outil de génération s'appuie sur un compilateur préexistant dont le front-end a été enrichi pour permettre la définition de notre langage et le back-end modifié pour y intégrer la capacité à générer une description architecturale depuis un template et le microcode. Cet outil est également entièrement écrit en Lua et une exécution sur *SoC FPGA* est possible, ce qui permet d'envisager la re-génération du microcode d'un *CPU* RISC-V pendant le fonctionnement. La seule condition est que ces changements n'entraitent pas de modification de la description architecturale. Dans le cas où la description du microcode entrainerait une modification de la description architecturale, il faudrait avoir recours à un ordinateur capable de générer le bitstream de manière distante puis téléverser et reconfigurer le *FPGA*. Idéalement cela pourrait être effectué en utilisant de la reconfiguration dynamique partielle.

En conclusion, les possibilités offertes par l'utilisation du langage Lua nous ont permis de nous abstraire de l'utilisation d'un système d'exploitation. Ces travaux n'avaient cependant pas pour objectif de remettre en cause l'utilité de tels systèmes mais plutôt d'y apporter une alternative. Il n'existe en effet pas de solution idoine pour la gestion d'une architecture hétérogène reconfigurable. De plus, le choix de l'organe de gestion des ressources est souvent lié à un compromis entre les besoins de l'applicatif et les ressources à administrer. L'arrivée de la 5G, qui se positionne en rupture face aux précédents standards de communication sans fil, va largement favoriser la multiplication des systèmes embarqués et l'intégration de l'IoT. A l'ère de cette évolution, les solutions standalones auront autant leur place sur le marché de l'industrie que les solutions qui se basent sur un système d'exploitation. De même que le développement d'architecture dédiée fera la part belle aux architectures hétérogènes reconfigurables.

Bibliographie

[1] H. K.-H. So and R. W. Brodersen, "Borph : An operating system for fpga-based reconfigurable computers," Ph.D. dissertation, EECS Department, University of California, Berkeley, Jul 2007. [Online]. Available : http://www2.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-92.html

[2] A. Ismail and L. Shannon, "Fuse : Front-end user framework for o/s abstraction of hardware accelerators," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2011, pp. 170–177.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition : A Quantitative Approach*, 5th ed. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2011.

[4] MarketsandMarkets, "Fpga market by technology (sram, antifuse, flash), node size (less than 28 nm, 28-90 nm, more than 90 nm), configuration (high-end fpga, mid-range fpga, low-end fpga), vertical (telecommunications, automotive), and geography - global forecast to 2023," MarketsandMarkets, Tech. Rep., Oct. 2017.

[5] G. Herrmann and G. Dost, "Entwurf und technologie von mikroprozessoren," in *Taschenbuch Mikroprozessortechnik, 2. Auflage*, T. Beierlein and O. Hagenbruch, Eds. Fachbuchverlag Leipzig im Carl Hanser Verlag Mnchen Wien, 2001.

[6] P. B. Ankita Bhutani, "Embedded system market size by application (automotive, industrial, consumer electronics, telecommunication, healthcare, military and aerospace), by product (software, hardware) industry outlook report, regional analysis, application development potential, price trends, competitive market share and forecast, 2016 – 2023," Global Market Insights, Tech. Rep., Mars 2016.

[7] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable computing architectures," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 332–354, March 2015.

[8] A. Watterman. (2019) riscv-test/isa/rv32ui. [Online]. Available : https://github.com/riscv/riscv-tests/tree/master/isa/rv32ui

[9] L. Gwennap, "P6 microcode can be patched," *Microprocessor Report*, 1997.

[10] Anonymous. (2004, July) Opteron exposed : Reverse engineering amd k8 microcode updates. [Online]. Available : http://securitem.com/securityreviews/5FP0M1PDFO

[11] ——. (2016, Feb) Reverse engineering the arm1 processor's microinstructions. [Online]. Available : http://www.righto.com/2016/02/reverse-engineering-arm1-processor.html

[12] P. Koppe, B. Kollenda, M. Fyrbiak, C. Kison, R. Gawlik, C. Paar, and T. Holz, "Reverse engineering x86 processor microcode," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC : USENIX Association, 2017, pp. 1163–1180. [Online]. Available : https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/koppe

[13] K. Asanović and D. A. Patterson, "The risc-v instruction set manual, volume i : Base user-level isa," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-62, May 2011. [Online]. Available : http://digitalassets.lib.berkeley.edu/techreports/ucb/text/EECS-2011-62.pdf

[14] J. von Neumann, "First draft of a report on the edvac," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[15] J. Backus, "Acm turing award lectures." New York, NY, USA : ACM, 2007, ch. Can Programming Be Liberated from the Von Neumann Style ? : A Functional Style and Its Algebra of Programs. [Online]. Available : http://doi.acm.org/10.1145/1283920.1283933

[16] G. Estrin, "Organization of computer systems : The fixed plus variable structure computer," in *Papers Presented at the May 3-5, 1960, Western Joint IRE-AIEE-ACM Computer Conference*, ser. IRE-AIEE-ACM '60 (Western). New York, NY, USA : ACM, 1960, pp. 33–40. [Online]. Available : http://doi.acm.org/10.1145/1460361.1460365

[17] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff." vol. 11, no. 3, Sep. 2006, pp. 33–35.

[18] Altera. (1984) Ep310 datasheet. [Online]. Available : http://www-inst.eecs.berkeley.edu/~cs294-59/fa10/resources/Altera-history/altera_ep300.pdf

[19] Xilinx, "The evolution of programmable logic design technology," in *Xcell*, vol. 32, Second Quarter 1999, pp. 5–8.

[20] ARM. (2014) Arm architecture reference manual, armv7-a and armv7-r edition. [Online]. Available : https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf

[21] Intel. (2019) Intel® architecture instruction set extensions and future features programming reference. [Online].

Available : https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf

[22] A. Waterman, "Design of the risc-v instruction set architecture," Ph.D. dissertation, EECS Department, University of California, Berkeley, Jan 2016. [Online]. Available : https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf

[23] Xilinx. (2019) Xilinx - adaptable. intelligent. [Online]. Available : https://www.xilinx.com/

[24] Intel. (2019) Intelfpgas and programmable devices - intelfpga. [Online]. Available : https://www.intel.com/content/www/us/en/products/programmable.html

[25] ——. (2015) Intel completes acquisition of altera. [Online]. Available : https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/

[26] L. Semiconductor. (2019) Lattice semiconductor. [Online]. Available : http://www.latticesemi.com/en

[27] Microsemi. (2019) Microsemi. semiconductor and system solution. power matters. [Online]. Available : https://www.microsemi.com/

[28] QuickLogic. (2019) Quicklogic - provider of end-to-end solutions for endpoint ai. [Online]. Available : https://www.quicklogic.com/

[29] VHDL, "Ieee standard vhdl language reference manual," *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pp. c1–626, Jan 2009.

[30] Verilog, "Ieee standard for verilog hardware description language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–590, April 2006.

[31] SystemVerilog, "Ieee standard for systemverilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, Feb 2018.

[32] Xilinx. (2019) Vivado design suite. [Online]. Available : https://www.xilinx.com/products/design-tools/vivado.html

[33] intelFPGA. (2019) intel quartus prime software suite. [Online]. Available : https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html

[34] C. Wolf. (2019) Yosys open synthesis suite. [Online]. Available : http://www.clifford.at/yosys/

[35] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic, "Yosys+nextpnr : An open source framework from verilog to bitstream for commercial fpgas," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2019, pp. 1–4.

[36] SymbiFlow. (2019) Projet treillis. [Online]. Available : https://github.com/SymbiFlow/prjtrellis

[37] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung, "Reconfigurable computing : architectures and design methods," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 2, pp. 193–207, March 2005.

[38] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing : A survey," *J. VLSI Signal Process. Syst.*, vol. 28, no. 1-2, pp. 7–27, 2001. [Online]. Available : https://doi.org/10.1023/A:1008155020711

[39] D. B. Thomas and W. Luk, "Credit risk modelling using hardware accelerated monte-carlo simulation," in *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '08. Washington, DC, USA : IEEE Computer Society, 2008, pp. 229–238. [Online]. Available : https://doi.org/10.1109/FCCM.2008.41

[40] G. W. Morris and M. Aubury, "Design space exploration of the european option benchmark using hyperstreams," in *2007 International Conference on Field Programmable Logic and Applications*, Aug 2007, pp. 5–10.

[41] R. McCready, "Real-time face detection on a configurable hardware system," in *Field-Programmable Logic and Applications : The Roadmap to Reconfigurable Computing*, R. W. Hartenstein and H. Grünbacher, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2000, pp. 157–162.

[42] M. Genovese and E. Napoli, "Asic and fpga implementation of the gaussian mixture model algorithm for real-time segmentation of high definition video," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 3, pp. 537–547, March 2014.

[43] S. Drimer, T. Güneysu, and C. Paar, "Dsps, brams, and a pinch of logic : Extended recipes for aes on fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 1, pp. 3 :1–3 :27, Jan. 2010. [Online]. Available : http://doi.acm.org/10.1145/1661438.1661441

[44] T. Becker, W. Luk, and P. Y. K. Cheung, "Enhancing relocatability of partial bitstreams for run-time reconfiguration," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, April 2007, pp. 35–44.

[45] Xilinx. (2011) Ds586 : Logicore ip xps hwicap(v5.01a). [Online]. Available : https://www.xilinx.com/support/documentation/ip_documentation/xps_hwicap/v5_01_a/xps_hwicap.pdf

[46] K. Vipin and S. A. Fahmy, "Fpga dynamic and partial reconfiguration : A survey of architectures, methods, and applications," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 72 :1–72 :39, Jul. 2018. [Online]. Available : http://doi.acm.org/10.1145/3193827

[47] D. Koch, J. Torresen, C. Beckhoff, D. Ziener, C. Dennl, V. Breuer, J. Teich, M. Feilen, and W. Stechele, "Partial reconfiguration on fpgas in practice — tools and applications," 01 2012.

[48] Xilinx. (2019) Xilinx soc portfolio. [Online]. Available : https://www.xilinx.com/products/silicon-devices/soc.html

[49] intelFPGA. (2019) intelfpga cyclone v. [Online]. Available : https://www.intel.com/content/www/us/en/products/programmable/fpga/cyclone-v.html

[50] ——. (2019) intelfpga arria 10. [Online]. Available : https://www.intel.com/content/www/us/en/products/programmable/fpga/arria-10.html

[51] ——. (2019) intelfpga agilex. [Online]. Available : https://www.intel.fr/content/www/fr/fr/products/programmable/fpga/agilex.html

[52] Xilinx. (2019) Zynq ultrascale + rfsoc. [Online]. Available : https://www.xilinx.com/products/silicon-devices/soc/rfsoc.html

[53] G. I. Analysts, "Global fpga industry," ReportLinker, Tech. Rep., July 2019.

[54] Xilinx. (2016) PYNQ : Python productivity for Zynq. [Online]. Available : http://www.pynq.io

[55] B. Hutchings and M. Wirthlin, "Rapid implementation of a partially reconfigurable video system with pynq," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–8.

[56] J. Goeders, T. Gaskin, and B. Hutchings, "Demand driven assembly of fpga configurations using partial reconfiguration, ubuntu linux, and pynq," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2018, pp. 149–156.

[57] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. De Wit, "A dynamic reconfiguration run-time system," in *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*, April 1997, pp. 66–75.

[58] G. Brebner, "A virtual hardware operating system for the xilinx xc6200," in *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, R. W. Hartenstein and M. Glesner, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 1996, pp. 327–336.

[59] K. Danne, R. Muhlenbernd, and M. Platzner, "Executing hardware tasks on dynamically reconfigurable devices under real-time conditions," in *2006 International Conference on Field Programmable Logic and Applications*, Aug 2006, pp. 1–6.

[60] M. Gotz and F. Dittmann, "Reconfigurable microkernel-based rtos : Mechanisms and methods for run-time reconfiguration," in *2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, Sep. 2006, pp. 1–8.

[61] H. Walder and M. Platzner, "A runtime environment for reconfigurable hardware operating systems," in *Field Programmable Logic and Application*, J. Becker, M. Platzner, and S. Vernalde, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, pp. 831–835.

[62] P. Garcia and K. Compton, "A reconfigurable hardware interface for a modern computing system," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, April 2007, pp. 73–84.

[63] W. Fu and K. Compton, "An execution environment for reconfigurable computing," in *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, April 2005, pp. 149–158.

[64] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms : online scheduling of real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1393–1407, Nov 2004.

[65] S. A. Fahmy, J. Lotze, J. Noguera, L. Doyle, and R. Esser, "Generic software framework for adaptive applications on fpgas," in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, April 2009, pp. 55–62.

[66] R. Brodersen, A. Tkachenko, and H. K. So, "A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph," in *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*, Oct 2006, pp. 259–264.

[67] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, "Reconos : An operating system approach for reconfigurable computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71, Jan 2014.

[68] R. Pellizzoni and M. Caccamo, "Real-time management of hardware and software tasks for FPGA-based embedded systems," *IEEE Transactions on Computers*, vol. 56, no. 12, pp. 1666–1680, Dec. 2007.

[69] J.-P. Diguet, Y. Eustache, and G. Gogniat, "Closed-loop–based self-adaptive hardware/software-embedded systems : Design methodology and smart cam case study," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 3, May 2011. [Online]. Available : https://doi.org/10.1145/1952522.1952531

[70] EETime and Embedded, "2019 embedded markets study," Website Study, 2019.

[71] H. K. So and R. W. Brodersen, "Improving usability of fpga-based reconfigurable computers through operating system support," in *2006 International Conference on Field Programmable Logic and Applications*, Aug 2006, pp. 1–6.

[72] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews, "Hthreads : A computational model for reconfigurable devices," in *2006 International Conference on Field Programmable Logic and Applications*, Aug 2006, pp. 1–4.

[73] Xilinx. (2019) Petalinux tools. [Online]. Available : https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html

[74] CyberEdu, "Fiches pédagogiques sur la cybersécurité au sein des composants électroniques," ANSSI, Tech. Rep., Fev 2017.

[75] S. Adee, "The hunt for the kill switch," *IEEE Spectrum*, vol. 45, no. 5, pp. 34–39, May 2008.

[76] S. Skorobogatov and C. Woods, "Breakthrough silicon scanning discovers backdoor in military chip," in *Cryptographic Hardware and Embedded Systems – CHES 2012*, E. Prouff and P. Schaumont, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, pp. 23–40.

[77] J. Zhang and G. Qu, "Recent attacks and defenses on fpga-based systems," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, no. 3, pp. 14 :1–14 :24, Aug. 2019. [Online]. Available : http://doi.acm.org/10.1145/3340557

[78] C. Paar and J. Pelzl, *Understanding Cryptography : A Textbook for Students and Practitioners*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[79] P. S. P. 197, "Announcing the advanced encryption standard (aes)," Federal Information, Tech. Rep. NIST.FIPS.197, Nov 2001. [Online]. Available : https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[80] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978. [Online]. Available : http://doi.acm.org/10.1145/359340.359342

[81] P. S. P. 202, "Sha-3 standard : Permutation-based hash and extendable-output functions," Federal Information, Tech. Rep. NIST.FIPS.202, Aug 2015. [Online]. Available : https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

[82] F. Hategekimana, T. Whitaker, M. J. H. Pantho, and C. Bobda, "Shielding non-trusted ips in socs," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–4.

[83] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control : Sandboxing accelerators," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 470–481.

[84] D. Hwang, M. Yang, S. Jeon, Y. Lee, D. Kwon, and Y. Paek, "Riskim : Toward complete kernel protection with hardware support," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 740–745.

[85] F. Hategekimana, J. Mandebi Mbongue, M. J. H. Pantho, and C. Bobda, "Secure hardware kernels execution in cpu+fpga heterogeneous cloud," in *2018 International Conference on Field-Programmable Technology (FPT)*, Dec 2018, pp. 182–189.

[86] P. Cotret, G. Gogniat, and M. J. S. Flórez, "Protection of heterogeneous architectures on fpgas : An approach based on hardware firewalls," *CoRR*, vol. abs/1602.05106, 2016. [Online]. Available : http://arxiv.org/abs/1602.05106

[87] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications confining the wily hacker," in *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, ser. SSYM'96. Berkeley, CA, USA : USENIX Association, 1996, pp. 1–1. [Online]. Available : http://dl.acm.org/citation.cfm?id=1267569.1267570

[88] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client : A sandbox for portable, untrusted x86 native code," in *2009 30th IEEE Symposium on Security and Privacy*, May 2009, pp. 79–93.

[89] T. Kim and N. Zeldovich, "Practical and effective sandboxing for non-root users," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13. Berkeley, CA, USA : USENIX Association, 2013, pp. 139–144. [Online]. Available : http://dl.acm.org/citation.cfm?id=2535461.2535478

[90] V. Costan and S. Devadas, "Intel sgx explained."

[91] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan : A distributed sandbox for untrusted computation on secret data," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA : USENIX Association, 2016, pp. 533–549. [Online]. Available : https://www.usenix.org/conference/osdi16/technical-sessions/presentation/hunt

[92] A. S. Technology, "Building a secure system using trustzone technology," ARM, Tech. Rep., apr 2009. [Online]. Available : http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf

[93] Trustonic, "Trustonic security for mobile vehicle applications on android," Trustonic, Tech. Rep., 2017.

[94] N. Jacob, J. Heyszl, A. Zankl, C. Rolfes, and G. Sigl, "How to break secure boot on fpga socs through malicious hardware," 08 2017, pp. 425–442.

[95] E. M. Benhani, C. Marchand, A. Aubert, and L. Bossuet, "On the security evaluation of the arm trustzone extension in a heterogeneous soc," in *2017 30th IEEE International System-on-Chip Conference (SOCC)*, Sep. 2017, pp. 108–113.

[96] E. M. Benhani, L. Bossuet, and A. Aubert, "The security of arm trustzone in a fpga-based soc," *IEEE Transactions on Computers*, vol. PP, pp. 1–1, 02 2019.

[97] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad : Cross-privilege-boundary data sampling," *arXiv :1905.05726*, 2019.

[98] M. Schwarz, S. Weiser, and D. Gruss, "Practical enclave malware with intel SGX," *CoRR*, vol. abs/1902.03256, 2019. [Online]. Available : http://arxiv.org/abs/1902.03256

[99] A. Fog, "The microarchitecture of intel, amd and via cpus," Technical University of Denmark, Tech. Rep., 2018.

[100] A. C. S. Center. (2018) Update on processor vulnerabilities (meltdown/spectre). [Online]. Available : https://www.cyber.gov.au/news/update-on-processor-vulnerabilities-spectre-meltdown

[101] B. Kollenda, P. Koppe, M. Fyrbiak, C. Kison, C. Paar, and T. Holz, "An exploratory analysis of microcode as a building block for system defenses," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA : ACM, 2018, pp. 1649–1666. [Online]. Available : http://doi.acm.org/10.1145/3243734.3243861

[102] Microsoft. (2019) Kb4090007 : Intel microcode updates. [Online]. Available : https://support.microsoft.com/en-us/help/4090007/intel-microcode-updates

[103] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL : A unified framework for vertically integrated computer architecture research," in *IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 280–292.

[104] E. Logaras and E. S. Manolakos, "SysPy : using Python for processor-centric SoC design," in *IEEE International Conference on Electronics, Circuits and Systems*, Dec 2010, pp. 762–765.

[105] J. Decaluwe, "Myhdl : a Python-based hardware description language," in *Linux journal*, vol. 127, Nov 2014, p. 5.

[106] P. Haglund, O. Mencer, W. Luk, and B. Tai, "PyHDL : Hardware scripting with Python," pp. 288–291, Jan. 2003.

[107] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, "A pythonic approach for rapid hardware prototyping and instrumentation," in *International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2017, pp. 1–7.

[108] D. D. Gajski and L. Ramachandran, "Introduction to high-level synthesis," *IEEE Design Test of Computers*, vol. 11, no. 4, pp. 44–54, Winter 1994.

[109] G. Martin and G. Smith, "High-level synthesis : Past, present, and future," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 18–25, July 2009.

[110] Xilinx. (2019) Vivado hls. [Online]. Available : https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[111] intelFPGA. (2019) Intel hls compiler. [Online]. Available : https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[112] A. Canis *et al.*, "Legup : High-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA : ACM, 2011, pp. 33–36.

[113] E. Martin *et al.*, "Gaut : An architectural synthesis tool for dedicated signal processors," in *Proceedings of EURO-DAC 93 and EURO-VHDL 93- European Design Automation Conference*, Sep. 1993, pp. 14–19.

[114] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel : Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, June 2012, pp. 1212–1221.

[115] U. B. A. Research. (2019) Risc-v sodor - educational microarchitectures for risc-v isa. [Online]. Available : https://github.com/ucb-bar/riscv-sodor

[116] V. Kathail, J. Hwang, W. Sun, Y. Chobe, T. Shui, and J. Carrillo, "Sdsoc : A higher-level programming environment for zynq soc and ultrascale+ mpsoc," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA : ACM, 2016, pp. 4–4. [Online]. Available : http://doi.acm.org/10.1145/2847263.2847284

[117] K. Asanović and D. A. Patterson, "Instruction sets should be free : The case for risc-v," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, Aug 2014. [Online]. Available : http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html

[118] Baba and Hagiwara, "The mpg system : A machine-independent efficient microprogram generator," *IEEE Transactions on Computers*, vol. C-30, no. 6, pp. 373–395, June 1981.

[119] P. Marwedel, "A retargetable microcode generation system for a high-level microprogramming language," in *Proceedings of the 14th Annual Workshop on Microprogramming*, ser. MICRO 14. Piscataway, NJ, USA : IEEE Press, 1981, pp. 115–123. [Online]. Available : http://dl.acm.org/citation.cfm?id=800075.802443

[120] Sheraga and Gieser, "Experiments in automatic microcode generation," *IEEE Transactions on Computers*, vol. C-32, no. 6, pp. 557–569, June 1983.

[121] M. Balakrishnan, P. C. P. Bhatt, and B. B. Madan, "An efficient retargetable microcode generator," in *Proceedings of the 19th Annual Workshop on Microprogramming*, ser. MICRO 19. New York, NY, USA : ACM, 1986, pp. 44–53. [Online]. Available : http://doi.acm.org/10.1145/19551.19536

[122] Xilinx. (2019) Vitis unified software platform. [Online]. Available : https://www.xilinx.com/products/design-tools/vitis.html

[123] A. G. Schmidt, G. Weisz, and M. French, "Evaluating rapid application development with python for heterogeneous processor-based FPGAs," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 121–124.

[124] A. Rigo, M. Fijałkowski, C. F. Bolz, A. Cuni, B. Peterson, A. Gaynor, A. Håkan, H. Krekel, and S. Pedroni. (2003) PyPy : A fast, compliant alternative implementation of the python language. [Online]. Available : https://pypy.org/

[125] G. Van Rossum *et al.*, "Python programming language," in *USENIX Annual Technical Conference*, vol. 41, 2007, p. 36.

[126] Intel. Bios implementation test suite. [Online]. Available : http://biosbits.org

[127] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho, "Lua-an extensible extension language," *Softw., Pract. Exper.*, vol. 26, no. 6, pp. 635–652, 1996.

[128] M. Pall. (2008) The LuaJIT project. [Online]. Available : http://luajit.org

[129] Lighttpd. (2006) Lighttpl, fly light . [Online]. Available : https://www.lighttpd.net

[130] NGINX. (2004) NGINX : High Performance Load Balancer . [Online]. Available : https://www.nginx.com

[131] Monkey. (2001) Monkey Server. [Online]. Available : http://monkey-project.com

[132] A. L. Ortega. (2014) Cherokee. [Online]. Available : http://cherokee-project.com/

[133] H. Leisink. (2002) Hiawatha. [Online]. Available : https://www.hiawatha-webserver.org/

[134] R. Bonamy, H. Pham, S. Pillement, and D. Chillet, "Uparc—ultra-fast power-aware reconfiguration controller," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2012, pp. 1373–1378.

[135] F. Fainelli, "The OpenWrt embedded development framework," in *Proceedings of the Free and Open Source Software Developers European Meeting*, 2008.

[136] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7 : A Matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[137] C. F. Bolz and L. Tratt, "The impact of meta-tracing on VM design and implementation," *Science of Computer Programming*, vol. 98, pp. 408–421, 2015.

[138] R. Pozo and B. Miller. (2000) Scimark 2.0. [Online]. Available : http://math.nist.gov/scimark2

[139] Plotly. (2019) Plotly javascript open source graphing library. [Online]. Available : https://plot.ly/javascript/

[140] J. Dechelotte, R. Tessier, D. Dallet, and J. Crenne, "Lynq : A lightweight software layer for rapid soc fpga prototyping," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2018, pp. 372–3723.

[141] P. Merino, M. Jacome, and J. C. Lopez, "A methodology for task based partitioning and scheduling of dynamically reconfigurable systems," in *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*, April 1998, pp. 324–325.

[142] R. Cattaneo, R. Bellini, G. Durelli, C. Pilato, M. D. Santambrogio, and D. Sciuto, "Para-sched : A reconfiguration-aware scheduler for reconfigurable architectures," in *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, May 2014, pp. 243–250.

[143] A. L. de Moura, N. Rodriguez, and R. Ierusalimschy, "Coroutines in lua," in *Journal of Universal Computer Science*. J.UCS, Jul. 2004, pp. 910–925. [Online]. Available : http://lua.org/doc/jucs04.pdf

[144] T. O. Group, "1003.1 standard for information technology — portable operating system interface (posix)," IEEE and The Open Group, Tech. Rep., sep 2001.

[145] IBM. (2009) Inside the linux 2.6 completely fair scheduler. [Online]. Available : https://developer.ibm.com/tutorials/l-completely-fair-scheduler/

[146] M. Pall. (2017) Ffi sementics. [Online]. Available : http://luajit.org/ext_ffi_semantics.html

[147] O. Weisse, V. Bertacco, and T. Austin, "Regaining lost cycles with hotcalls : A fast interface for sgx secure enclaves," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA : Association for Computing Machinery, 2017, p. 81–93. [Online]. Available : https://doi.org/10.1145/3079856.3080208

[148] J. Amacher and V. Schiavoni, "On the performance of arm trustzone," in *Distributed Applications and Interoperable Systems*, J. Pereira and L. Ricci, Eds. Cham : Springer International Publishing, 2019, pp. 133–151.

[149] Linaro. (2019) Open portable trusted execution environment. [Online]. Available : https://www.op-tee.org/

[150] T. register. (2018) Kernel-memory-leaking intel processor design flaw forces linux, windows redesign. [Online]. Available : https://www.theregister.co.uk/2018/01/02/intel_cpu_design_flaw/

[151] Techtarget. (2018) Bug intel : le fondeur savait et a continué à vendre ses puces. [Online]. Available : https://www.lemagit.fr/actualites/450432587/Bug-Intel-le-fondeur-savait-et-a-continue-a-vendre-ses-puces

[152] D. D. Chen and G.-J. Ahn. (2014) Security analysis of x86 processor microcode. [Online]. Available : https://www.dcddcc.com/docs/2014_paper_microcode.pdf

[153] Microsemi. (2019) Polarfire soc. [Online]. Available : https://www.microsemi.com/product-directory/soc-fpgas/5498-polarfire-soc-fpga#overview

[154] E. Zurich and U. di Bologna. (2019) Pulp platform. [Online]. Available : https://pulp-platform.org

[155] C. Wolf. (2019) Picorv32 - a size-optimized risc-v cpu. [Online]. Available : https://github.com/cliffordwolf/picorv32

[156] G. M. Eclipse. (2019, Sept) The risc-v embedded gcc. [Online]. Available : https://gnu-mcu-eclipse.github.io/toolchain/riscv/

[157] llvm dev. (2019, Sept) Llvm 9.0.0 releases. [Online]. Available : https://lists.llvm.org/pipermail/llvm-dev/2019-September/135304.html

[158] R.-V. Foundation. (2019) Linux/risc-v. [Online]. Available : https://riscv.org/software-tools/linux/

[159] seL4. (2019) Risc-v. [Online]. Available : https://docs.sel4.systems/Hardware/RISCV

[160] T. P. Language. (2019) The titan programming language. [Online]. Available : https://github.com/titan-lang/titan

[161] P. P. Language. (2019) Pallene compiler. [Online]. Available : https://github.com/pallene-lang/pallene