PROJET AVANCE EN SYSTEMES EMBARQUES

Evaluation et extension d'un processeur RISC-V

POUPARD Hugo
DIVET Yann
ALMORIN Hugues
MONIÈRE Camille

3A Electronique Option SE
ENSEIRB-MATMECA

Table des matières

I. INTRODUCTION

Les performances d'un ordinateur dépendent de la qualité des programmes qu'il exécute, mais aussi des performances du matériel, en particulier de son processeur. Or les performances d'un processeur dépendent de son implémentation, qui dépend elle de l'architecture de jeu d'instructions du processeur. Or les architectures dominantes sur le marché (x86 pour les ordinateurs personnels, ARM pour les systèmes embarqués) sont protégées et sous des licences propriétaires, empêchant de développer des processeurs à partir de celles-ci et forçant la dépendance vis-à-vis de ceux disponibles. C'est dans ce contexte qu'est apparue l'architecture libre RISC-V.

Le but de ce projet est, après avoir réalisé un état de l'art du RISC-V, de choisir une méthode d'évaluation de l'architecture et de comparer les résultats de certaines implémentations entres elles voire à d'autres architectures. Les sources peuvent être trouvées à la fin de ce document et sont identifiées à l'aide du signe suivant : [x].

Note : Au cours de ce document, "RISC-V" est utilisé par abus de langage pour désigner les processeurs basés sur l'architecture de jeu d'instructions RISC-V.

II. ETAT DE L'ART

A. DÉFINITION ET PRÉSENTATION

RISC-V (pour *Reduced Instruction Set Architecture - Five*) est un projet d'architecture de jeu d'instructions libre et ouverte, sous licence Creative Commons attribution 4.0 International[1]. Comme le nom de l'architecture l'indique, elle est de type RISC, c'est-à-dire qu'elle ne définit qu'un nombre restreint d'instructions, chacune s'exécutant en un faible nombre de cycles d'horloge.

RISC-V est actuellement maintenue par une société à but non-lucratif, la RISC-V Foundation[2], qui dirige le développement de l'architecture et approuve ou non ses versions. L'architecture jouit surtout d'une communauté forte, composée à la fois de grands acteurs industriels et académiques (Bluespec, Inc. ; Google ; Microsemi ; NVIDIA ; NXP ; University of California, Berkeley ; Western Digital ; et autres…) mais aussi de nombreux volontaires, le caractère libre du projet permettant à tout le monde de participer, via GitHub entre autres. Et même si ce n'est pas la première architecture libre, elle se distingue par ses objectifs.

B. LE BUT DU PROJET RISC-V

Le but originel du projet était de développer une architecture RISC libre et open-source à but académique, facilitant la recherche. Cependant, le projet s'est rapidement transformé en standard pour l'industrie[3]. Pour cause, l'ambition du projet de définir une architecture polyvalente, pouvant s'implémenter dans un

microcontrôleur embarqué jusque dans un processeur multicœurs de supercalculateur en passant par un softcore complètement virtualisable.

C. SPÉCIFICATIONS

L'architecture n'est pour l'instant pas entièrement ratifiée par la RISC-V Foundation, mais plusieurs de ses spécifications sont en cours de ratification ou "gelées" ce qui signifie qu'elles ne devraient plus être modifiées d'ici leur ratification.

Une spécificité du RISC-V est sa modularité. L'architecture se compose d'une architecture de base de traitement des entiers à laquelle peut être adjointe des extensions. L'architecture résultante est donnée par un code : RV*XY*, avec RV pour RISC-V, X la longueur en bits des instructions et Y la base et les extensions ajoutées.

Base	Version	Status
RVWMO	2.0	Ratified
RV32I	**2.1**	**Ratification**
RV64I	**2.1**	**Ratification**
RV32E	*1.9*	*Draft*
RV128I	*1.7*	*Draft*
Extension	**Version**	**Status**
Zifencei	**2.0**	**Ratification**
Zicsr	**2.0**	**Ratification**
M	**2.0**	**Ratification**
A	**2.0**	**Ratification**
F	**2.2**	**Ratification**
D	**2.2**	**Ratification**
Q	**2.2**	**Ratification**
C	**2.0**	**Ratification**
Ztso	*0.1*	*Frozen*
Counters	*2.0*	*Draft*
L	*0.0*	*Draft*
B	*0.0*	*Draft*
J	*0.0*	*Draft*
T	*0.0*	*Draft*
P	*0.1*	*Draft*
V	*0.4*	*Draft*
N	*1.1*	*Draft*
Zam	*0.1*	*Draft*

Figure 1 : Bases et extensions ainsi que leur statut, version de décembre 2018[1].

Le détail de chaque module ne sera pas donné ici mais peut être facilement trouvé, que ce soit dans les spécifications de l'architecture ou sur Wikipédia. On notera tout de même les trois longueurs 32-64-128 bits comme premiers indicateurs de polyvalence de l'architecture, les deux bases possibles "I" pour gestion des entiers et "E" indiquant un jeu d'instructions réduit visant l'embarqué ("E" pour "Embedded") ainsi que l'existence de la lettre "G" (absente de la figure) qui est l'acronyme pour "General [purpose]" et incluant les lettres "IMAFD" soit "Integer Multiply-divide Atomic operations Floating-point single precision and Double precision". Le plus simple processeur RISC-V pour l'embarqué serait le RV32E et le généraliste le plus commun le RV64G.

III. EXPOSITION DES TRAVAUX

A. STRUCTURE

Nous avons choisi de rechercher plusieurs implémentations de RISC-V, sans restriction d'extension. Chaque membre du groupe a ainsi recherché une ou plusieurs implémentations en différents langages.

L'idée était d'étudier ces implémentations, de déterminer s'il était possible de les implémenter sur des FPGA en notre possession (Artix-7 et Zynq de Xilinx, iCE40 de Lattice, Cyclone-IV d'Altera). Si oui, de les implémenter et de les tester à l'aide d'une méthode permettant de comparer leurs performances. Cette méthode de test doit donc établir une grandeur de référence, ce qui indique l'utilisation d'un benchmark (signifiant "référence").

Il existe plusieurs types de benchmark pour mesurer les performances d'un système à différents niveaux, par exemple les tests de performance pour systèmes de calculs parallèles qui peuvent nous donner une mesure de la capacité à exécuter du code en parallèle. Le type qui convient le mieux à notre projet le benchmark synthétique : il consiste en l'exécution d'une suite d'instructions en proportions relatives à ce que contiennent statistiquement les programmes usuels. Il en existe plusieurs : Whetstone pour les processeurs capables de calcul sur des flottants, Dhrystone pour le calcul sur les entiers et CoreMark une version similaire au Dhrystone mais plus récente. Afin de ne pas compliquer le projet, nous avons choisi d'utiliser un benchmark à la fois simple et très utilisé, le Dhrystone. Il consiste en l'exécution de plusieurs instructions de calculs d'entiers en boucle, et à la mesure du nombre de tours de boucle par secondes en "Dhrystone/s". Ce nombre est ensuite divisé par 1757 (le nombre de Dhrystone/s d'une machine fonctionnant nominativement à 1 MIPS) pour obtenir des DMIPS puis par la fréquence d'horloge pour obtenir des DMIPS/MHz[3], une unité permettant une comparaison des performances des différentes implémentations.

Compte-tenu des différents choix d'optimisations dans chacune des implémentations, les résultats doivent être considérés avec précaution.

B. DESCRIPTION ET ANALYSE

1. f32c

a) Présentation

Le f32c est un softcore 32-bits décrit en VHDL "paramétrable" à l'origine développé pour exécuter des instructions MIPS et adapter ensuite à l'architecture RISC-V[4]. Il peut maintenant exécuter des instructions RISC-V ou MIPS. Dans les 2 cas, c'est un processeur scalaire et avec 5 étages de pipeline. Ces implémentations du RISC-V sont dites "paramétrables" par leurs créateurs, car il est possible de choisir la taille des mémoires programme et de données, ainsi que d'activer ou non différentes

options à ajouter au processeur, tel que le support de l'extension de signe ou la prédiction de branchement. Le GitHub du projet fournit des codes VHDL permettant d'implémenter ce softcore sur de nombreuses cible FPGA de chez Xilinx, Altera et Lattice. Tous les modules VHDL sont disponibles sont licence BSD.

b) Implémentation

Nous avons pour notre part voulu tester l'implémentation du f32c pour la carte Basys-3, dont le circuit FPGA est un Artix-7. Cette implémentation est une RV32IM qui permet donc la manipulation d'entier et leur multiplication. Deux versions de celle-ci sont disponibles dans le projet, une plus légère que l'autre en termes de mémoire et de ressources matérielles utilisées. Nous avons choisi celle des deux avec le plus de mémoire BRAM pour éviter au maximum de rencontrer des problèmes à cause d'une mémoire restreinte. Un bitstream de cette implémentation est directement disponible sur internet, cependant nous avons préféré le générer à nouveau à partir des sources pour avoir accès aux résultats de synthèse et être certain des options choisies. Les résultats de synthèse sont les suivants.

Les options choisies pour la synthèse sont les options par défaut du projet, sauf pour l'architecture où nous avons sélectionné RISC-V et non MIPS. Le choix des options se fait en définissant des valeurs pour des paramètres "generic". Le fait que ces derniers puissent être définis dans plusieurs fichiers avec des valeurs différentes et soient parfois écrasées par d'autres lors des affectations de signaux (portmap) rend difficile la compréhension des options choisies. De plus certaines options ne sont disponibles que pour l'architecture MIPS et non pour l'architecture RISC-V. Les seules options activées par défaut pour le RISC-V sont :

- Operand forwarding
- Load aligner
- Pipeline
- Prédiction de branchement

Ressource	Utilisation	Utilisation %
LUT	1663	8,00
LUTRAM	48	0,50
FF	1447	3,48
BRAM	32 Ko	64,00
DSP	0	0,00
IO	76	71,70
BUFG	1	3,13

Figure 2 : Ressources utilisées par le f32c.

On constate que le f32c n'utilise qu'un faible pourcentage des ressources de la carte en termes de LUT et de FF, en revanche il utilise une majorité des blocs RAM et des entrées/sorties.

La programmation du processeur f32c se fait à l'aide d'une l'IDE Arduino. Les créateurs du projet ont en effet développé une bibliothèque Arduino contenant les outils logiciels nécessaires à la programmation du f32c[5]. Cela présente l'avantage de pouvoir utiliser les bibliothèques Arduino pour programmer la carte. Il a grâce à cela été très simple de faire clignoter les Leds d'utiliser la liaison série de la carte. Il suffit d'utiliser les fonctions des bibliothèque Arduino :

- `pinMode()`
- `digitalWrite()`
- `delay()`

pour faire clignoter les leds et

- `Serial.begin()`
- `Serial.print()`
- `Serial.println()`

pour utiliser la liaison série de la carte. À noter que les programmes compilés avec l'IDE Arduino doivent respecter la syntaxe Arduino et notamment utiliser les fonctions setup() et loop().

c) Test

Nous avons ensuite voulu tester les performances du f32c à l'aide du Dhrystone. Le code du Dhrystone est fourni sur le GitHub du projet. Malheureusement, le code fourni n'a pas été adapté pour être compilé avec l'IDE Arduino. Il a donc fallu commencer par rassembler les fichier dhry_1.c dhry_2.c et dhry.h en un seul, adapter les déclarations de fonction, car l'IDE Arduino ne supporte pas la syntaxe obsolète utilisée dans ces fichiers. Il a également fallu ajouter toute la librairie standard du projet à l'IDE. Après ces différentes étapes, il restait encore des erreurs de compilation, notamment une macro assembleur non définie malgré le fait que le fichier la définissant ait bien été ajouté à la librairie Arduino. Nous n'avons malheureusement pas eu le temps de résoudre ce problème avant la fin du projet. Nous n'avons donc pas de résultat de test pour le f32c.

2. Comet

a) Présentation

Comet est un projet particulier de RISC-V décrit en C/C++ pour de la synthèse de haut niveau (HLS pour "High Level Synthesis")[6]. C'est un RV32I qui supporte en partie l'extension "M" (Multiply-divide integer operations), les opérateurs de division et de modulo étant hors du cœur, ces instructions s'exécutant en plus d'un cycle d'horloge. Il est divisé en 5 étages de pipeline, et inclut 64 Mo de mémoire programme

et 64 Mo de mémoire de données. Le projet est dépourvu de licence, le principe de propriété intellectuelle rend donc propriété exclusive de son auteur[7].

b) Analyse

L'utilisation de cette implémentation semblait simple au premier abord. En effet, le code source inclut un répertoire Vivado (l'outils de synthèse de Xilinx) ainsi qu'un script TCL exécutable par Vivado HLS. Cependant, une analyse plus poussée du projet permet de comprendre que le déploiement sur FPGA Xilinx n'est plus maintenu. Le projet se concentre aujourd'hui sur l'utilisation de Catapult HLS[8], un puissant outil de Mentor Graphics. Cet outil permet d'effectuer des synthèses de haut niveau visant une implémentation ASIC. Cependant cet outil est loin d'être gratuit.

Nous avons essayé de tout de même utiliser Vivado HLS, en créant un projet Vivado personnalisé incluant les sources nécessaires. Plusieurs problèmes ont été soulevés :

- Les bibliothèques utilisées sont propres à Catapult. Or si elles sont accessibles, rien ne prouve qu'elles seront synthétisées de la même manière par Vivado.
- Les sources incluent la description d'un cache. Cependant, ça synthèse n'aboutit pas sur Vivado HLS. Nous avons dû le retirer de la synthèse.
- Les sources ne décrivent clairement que le cœur du processeur. Un SoC est émulé lors de la simulation, mais il n'est pas dans les sources en C/C++, et les branchements nécessaires entre le cœur Comet et la glue nécessaire à son utilisation ne sont pas indiqués.

Ces problèmes et l'absence de fichier de contraintes permettant (dans l'éventualité d'une synthèse réussi d'un SoC) de lier l'IP aux connexions physiques du FPGA ont conduit à l'abandon du test de cette implémentation.

Si la perspective d'étudier une implémentation en HLS est intéressante, Comet ne répond pas aux contraintes de notre projet (temps limité, utilisation de Vivado HLS). Nous avons donc cherché une autre piste.

3. Piccolo

a) Présentation

Piccolo est un processeur paramétrable à trois étages de pipeline développé par Bluespec Inc. pour les systèmes embarqués. Il existe en deux versions, 32 et 64 bits, supporte les extensions "G", "C" (instructions compressées) et les niveaux de privilèges (utilisateur, superviseur et machine)[9] et supporte nativement les bus AXI4-Lite Fabric. Il est annoncé que les codes sources sont testés sur des cartes Xilinx, ce qui semble encourageant.

De plus, si les codes sont au format ".bsv", un langage développé par Bluespec pour réaliser de la synthèse de haut niveau à l'aide du Bluespec compiler (non libre), des codes sources Verilog pour un SoC RV32IACMU et un SoC RV64GCSU (avec

support de modes privilégiés) sont disponibles. Ces codes sont indiqués synthétisables et simulables sur iverilog et verilator, des logiciels de simulation pour les langages de description matérielle.

Enfin, la version 32 bits est sensée pouvoir démarrer FreeRTOS et la 64 bits, le noyau Linux.

b) Analyse

En suivant la procédure expliquée sur GitHub, il a été possible de simuler les implémentations dans iverilog. Ces simulations consistent en l'exécution de plusieurs instructions, et à la vérification du résultat. Cependant, contrairement à ce qui était annoncé, la synthèse Vivado a échoué. Il s'est avéré, que la taille des registres et de la mémoire allouée était gigantesque. Cela n'apparaît pas à la simulation, car le simulateur ne vérifie pas si la mémoire peut physiquement être allouée. Nous aurions pu changer manuellement cette valeur, mais les fichiers Verilog étant générer par un outil, ils ne sont pas écrits pour être facilement manipulés par un humain. Ainsi, il est très difficile de connaitre l'impact de ce changement sur les autres fichiers. Il faudrait pour cela remonter à la source, les codes Bluespec. Malheureusement, le logiciel de synthèse n'étant pas libre, ceci n'est pas possible. Ainsi, si la synthèse est possible après modification des codes sources, rien n'indique que le SoC synthétisé soit fonctionnel. Aucun testbench pour l'IP n'est fourni, et le fichier de contraintes devrait aussi être rédigé.

Ces problèmes ont conduit à l'abandon de cette implémentation.

4. PicoRV32

a) Présentation

Le PicoRV32 a été conçu en Verilog par Clifford Wolf[10], aussi concepteur de la chaîne d'outils *Yosys*, *arachnepnr*/*nextpnr* et *IceStorm*, permettant de faire des synthèses Verilog, du placement routage et du chargement de bitstream pour FPGA. Ce design vise une implémentation légère d'un RISC-V 32 bits pour un utilisation en processeur auxiliaire sur des projets sur FPGA ou ASIC. Dans la majorité des cibles, sa haute fréquence d'utilisation maximale lui permet de s'ajouter dans n'importe quel projet sans gêner les contraintes de timing. Les fonctionnalités de configuration du cœur lui permettent d'être généré en tant que RV32I (Integer) avec facultativement les options M et C (Integer Multiplication and Division, 16-bits compressed instructions) ou bien en RV32E (Embedded 32 bits core, 16 registers) en désactivant une bonne partie des options. Ce cœur n'est pas conçu autour d'un pipeline. Il existe également en deux versions : bus processeur natif ou bien bus AXI4-Lite Master. L'ajout d'un adaptateur est possible pour utiliser l'AXI en plus du bus natif.

Le projet propose un SoC (System on Chip) ciblant les FPGA Lattice iCE40, il contient un module UART pour la communication série, un module SRAM de 1Ko (par défaut), un contrôleur SPI (principalement pour communiquer avec la mémoire SPI-

Flash de la carte). Spécifiquement à la cible, un module de contrôle des GPIO est aussi présent.

Un programme d'exemple pour ce SoC est fourni, ainsi qu'un benchmark Dhrystone visant à s'exécuter sur une simulation Verilog du core à l'aide de *iverilog*.

b) Tests réalisés et résultats

Une carte iCE40-HX8K conçue par Lattice nous a été fournie, elle est une des cibles d'exemple du projet, sa prise en main a fait partie des expérimentations. La première étape du test de la plateforme a été de générer le SoC pour l'implanter dans cette carte. Après analyse du Makefile, on peut voir que l'outil *Yosys* est utilisé pour la synthèse et *arachnepnr* pour le placement routage. On peut remarquer que l'ordre de placement routage utilisé par *arachnepnr* est fixe par défaut et peut mener à un échec de la procédure, il a donc été choisi de modifier le Makefile pour y ajouter un ordre aléatoire de placement routage. Une fois l'application d'exemple compilée, le bitstream et le programme peuvent être chargés dans la mémoire Flash de la carte à l'aide de l'outil *iceprog* de la suite *IceStorm*. On peut constater que dans la description du SoC, l'adresse de base du programme dans la mémoire flash est à *0x00100000* (1Mo après le début), c'est pourquoi il est spécifié à *iceprog* de charger le programme à cet endroit (option *-o 1M*). Le programme d'exemple permet, via la communication série, d'afficher du texte et d'exécuter un test simple permettant de vérifier le bon fonctionnement du système.

A ce stade des manipulations il est possible de quantifier l'utilisation des ressources de la cible : 5761/7680 Luts (LUT 4 entrées) et 6/32 BRAM (Block RAM simple port de 4kb)[11]. L'analyse du fichier *sections.lds* permet d'identifier qu'une architecture de type Harvard est utilisée, en effet la mémoire de données est séparée de la mémoire programme située en Flash.

L'étape suivante est de faire exécuter au système un Dhrystone. Un code est notamment fourni avec le projet, il est destiné à la base à s'exécuter dans une simulation Verilog (utilisation de *iverilog* et *vvp*). On pourra en comparer les résultats dans le tableau final. L'adaptation de ce code a été faite en adaptant la séquence d'amorçage (fichier *start.s*), la réutilisation du linker script *sections.lds* du programme d'exemple et l'utilisation d'une librairie standard personnalisée. Un Makefile a également été conçu. Durant ce développement, il a été remarqué que la mémoire de donnée présente sur la cible (1Ko) était trop faible pour accueillir le benchmark qui nécessite environ 11Ko. Après une synthèse en modifiant la taille de la mémoire dans le SoC et la modification du linker script, le programme est prêt à l'exécution.

Le Dhrystone fonctionne correctement et l'ombre planant au-dessus des fonctions time() et instr() a été réglée en plongeant dans le Verilog. Ces deux fonctions sont en fait des appels à des instructions custom (mais obligatoires dans un RV32I) qui permettent de récupérer le nombre de coup d'horloge et le nombre d'instructions décodées.

c) Résumé

Le tableau ci-dessous nous permet de comparer les performances mesurées, les performances annoncées et simulées :

	Annoncé (optimal)	Simulé (mêmes conditions que la mesure)	Mesuré
Cycles par instruction	4.1	3.883	79
Dhrystone/s/MHz	908	710	45
DMIPS/MHz	0.516	0.404	0.025

Figure 3 : Résultats des tests de performances du PicoRV32

On peut y voir que les performances mesurées sont très inférieures à celles simulées. Cela est en fait dû à une différence dans les options de génération du SoC par rapport à celles utilisées pour les performances annoncées. Les calculs de multiplication et de division sont effectués par des coprocesseurs connectés sur le Pico Co-Processor Interface (PCPI) ce qui offre le choix d'en instancier des blocs différents facilement, au prix d'un peu de logique pour l'interface. Il existe alors deux versions du multiplicateur utilisables (option *ENABLE_FASTMUL*) donc la plus rapide permet un calcul en environ 4 à 5 cycles. Il n'est hélas pas possible d'implémenter pour la cible iCE40 le cœur avec cette option activée, il manque environ 1000 Luts.

Il y a également une différence non négligeable entre le système simulé et le système mesuré. En effet, le système simulé ne prend en compte que le cœur, accédant à une RAM qui contient directement le programme, or dans le système mesuré, le SoC est conçu pour accéder à la mémoire programme à travers un contrôleur SPI-Flash, ce qui pourrait expliquer la valeur excessive de cycles par instruction

Pour conclure cette partie des expérimentations, on peut confirmer qu'en effet l'utilisation pour laquelle ce cœur RISC-V a été prévu est celle d'un processeur auxiliaire pouvant se placer dans n'importe quel design. La cible possède un nombre de ressource qui convient très bien à une version dégradée de ce RV32IMC et surtout à un RV32E, à des utilisations plutôt microcontrôleur embarqué que systèmes permettant d'accueillir un OS linux. Les difficultés de ces expériences ont résidé en la complexité de génération d'un programme baremetal, avec notamment l'utilisation d'une librairie standard spécifique et de linker scripts.

5. lowRISC

a) Présentation

Le projet lowRISC[12], fondé notamment par Robert Mullins (co-fondateur du projet Raspberry Pi) et Alex Bradbury (contributeur au projet Raspberry Pi), a pour but

la création d'une implémentation du processeur RISC-V 64 bits à bas coûts. Il reste ainsi dans la lignée des projets précédents de ces fondateurs. Cette implémentation se veut personnalisable, et capable de faire tourner un système d'exploitation Linux. Destiné à être implémenté sur une Nexys 4 DDR de chez Xilinx, le lowRISC prend en charge presque l'intégralité des entrées/sorties présentes sur la carte. Cela permet une multitude d'applications, allant du simple programme baremetal utilisant des interrupteurs ou des leds jusqu'au Linux embarqué.

Le projet lowRISC est une implémentation de type RV64GC (ou RV64IMAFDC). Elle comprend donc une architecture de base 64 bits, avec les extensions suivantes :

- C : 16-bits compressed instructions
- G : General, comprenant :
    - I : Integer
    - M : Integer Multiplication and Division
    - A : Atomic operations
    - F : Single-precision floating-point
    - D : Double-precision floating-point

De plus, le lowRISC est fourni avec divers programmes d'exemple, notamment des exemples baremetal tels qu'un "Hello World" ou un test de mémoire. Il contient également un exemple permettant de faire booter la carte sur un système d'exploitation Linux Debian depuis une carte SD ou en NFS via liaison Ethernet. La documentation inclut les étapes pour la compilation de l'image Linux. La toolchain permettant de compiler des exécutables pour une architecture RISC-V 64 bits est également fournie avec ce projet.

b) Tests réalisés et résultats

Les différents exemples fournis avec le lowRISC ont d'abord été testé. L'exemple baremetal "Hello World" génère un bitstream qui, lors du démarrage de la carte, envoie sur la liaison UART un message "Hello World". On s'assure ainsi du bon fonctionnement du RISC-V sur la carte. D'autres exemples baremetal ont été testé, mais ne seront pas présentés ici.

L'exemple permettant de faire booter la carte sur un Linux a également été testé. Après avoir compilé l'image Linux Debian en suivant la documentation et généré le bitstream correspondant, un script fourni permet de formater correctement la carte SD pour utilisation et de copier l'image sur la carte. Afin de ne pas avoir à reprogrammer la carte à chaque mise hors tension, la Quad-SPI flash a été programmé avec le bitstream.

L'exemple fonctionne alors parfaitement, la carte permet de booter sur l'image Linux. La communication peut alors se faire via UART (méthode la plus facile) mais il est également possible de brancher un clavier et un écran VGA sur la carte (les contrôleurs VGA et USB étant implémenté). Il est aussi possible de se connecter à la carte via SSH, en branchant la carte en Ethernet et en suivant le tutoriel pour

paramétrer le réseau. Le système Linux inclut avec lui la toolchain RISC-V64, permettant de compiler des exécutables pour ce type d'architecture.

Au niveau des résultats sur les ressources matérielles utilisées, l'outil Vivado fourni le tableau suivant.

Ressources	Utilisation	Utilisation %
LUT	47679	75.20
LUTRAM	2946	15.51
FF	31249	24.64
BRAM	83	61.48
DSP	15	6.25
IO	125	59.52
BUFG	9	28.13
MMCM	2	33.33
PLL	2	33.33

Figure 4 : Résultats de l'utilisation matérielle issus de l'outil Vivado.

On observe ainsi que l'implémentation lowRISC nécessite une quantité assez conséquente de ressources matérielles, aussi bien au niveau cellules logiques qu'au niveau mémoire. Cependant, il est toujours possible de rajouter d'autres modules à cette implémentation.

Niveau performances, le benchmark *Dhrystone* a été utilisé. Malheureusement, il a été impossible de générer un bitstream pour faire tourner un Dhrystone baremetal, dû à la complexité des Makefiles et de problèmes de librairies. Le Linux embarqué sur la carte contenant la toolchain RISC-V64, cet exécutable a donc été compilé et testé sous l'OS. Avec 50 000 runs à travers le Dhrystone, on obtient un résultat de **46 992 Dhrystones/s**, soit l'équivalent de **0,134 DMIPS/MHz**.

On est alors loin des résultats annoncés pour certains softcore commercialisés, tels que le Nios II/f (0,9 DMIPS/MHz) ou le MicroBlaze "Microcontroller" (1,1 DMIPS/MHz).

c) Résumé

L'implémentation lowRISC est donc assez intéressante et riche, grâce notamment aux nombreux exemples fournis avec celui-ci. Destiné à être implémenté sur des cartes relativement accessibles (ou en tous cas disponibles au sein de l'école), il permet de travailler facilement sur une architecture RISC-V. Point négatif de cette implémentation : la complexité dans ses Makefile, et l'organisation des dépôts. Il

devient alors assez compliqué de comprendre exactement comment sont générés les exemples baremetal et surtout comment en rajouter. La documentation encore trop incomplète de ce projet le rend complexe de compréhension. Au niveau des performances obtenues, le lowRISC reste bien en dessous des performances d'une majorité des softcores disponibles sur le marché, comme le MicroBlaze. Ses performances sont néanmoins comparables au Nios II/e, destiné à des implémentations bas coût (tout comme le lowRISC). Ce projet reste pour autant très intéressant et prometteur.

C. COMPARATIFS

Nous souhaitions comparer les performances RISC-V à d'autres architectures. Celles que nous avons choisis sont celles d'Intel (Nios II), Xilinx (MicroBlaze) et ARM (Cortex).

Processeur	DMIPS/MHz Annoncés	DMIPS/MHz Mesurés
Nios II/f	0.9[13]	
Nios II/e	0.1[13]	
MicroBlaze "Applications Processor"	1.4[14]	
MicroBlaze "Microcontroller"	1.1[14]	
Cortex-A35	1.78[15]	
Cortex-A5	1.57[15]	
RISC-V Piccolo	1.9[16]	
RISC-V f32c	1.31[4]	
RISC-V LowRISC		0.134
RISC-V Picorv32	0.516	0.025

Figure 5 : Comparatifs entre des processeurs RISC-V et d'autres architectures (MicroBlaze de Xilinx, Nios II de Intel, Cortex de ARM).

Le Cortex-A5, le Nios II et le MicroBlaze sont en 32 bits. Le Cortex-A35 et la version du Piccolo sont en 64 bits. Le Nios II/e ("economy") est une version allégée ne contenant que les composants indispensables, alors que le Nios II/f ("fast") est optimisé pour la vitesse d'exécution. Le même principe est appliqué aux MicroBlaze "Microcontroller" et "Applications Processor". Les Cortex sont tous les deux les

processeurs les moins puissants de leur catégorie, mais les plus puissant dispose d'un pipeline "Out-of-Order" et de prédictions de branchement rendant la comparaison incohérente.

Les performances annoncées des RISC-V en font de sérieux concurrents, surtout en considérant les futures évolutions prévues (pipeline "Out-of-Order" par exemple). De plus, contrairement aux autres, certains softcore RISC-V sont complètement libres (de droits et open-source).

Enfin, les performances mesurées du lowRISC sont très similaires (en termes de DMIPS/MHz) à celles annoncées du Nios II/e. Ainsi nous avons pu observer nous-même ces performances.

IV. CONCLUSION

Nous avons pu constater au cours de cette étude de RISC-V les avantages de cette architecture mais aussi rencontré certains problèmes. En effet, les processeurs RISC-V ont des performances comparables à leurs concurrents tout en étant plus accessible. Le bénéfice de l'open-source est une plus grande réactivité aux bugs et une amélioration permanente, via la communauté. Cela s'accompagne bien sûre d'une foule d'implémentations possibles, certaines très fonctionnelles, d'autres moins.

Néanmoins, si un bon nombre de logiciels ont été portés sur RISC-V (compilateurs, debuggers, noyaux de systèmes d'exploitation, etc…), certains portages restent en développement et même si l'architectures est plutôt stable désormais, il reste des points cruciaux à établir (standardisation des registres de contrôle et de statuts, jeu d'instruction limité pour l'embarqué, etc…). De plus, si un OS permet de s'affranchir du matériel (grâce à un environnement standardisé) et donc de développer des programmes plus simplement, l'attribution des adresses physiques est laissée au concepteur, ce qui peut poser problème lors du développement d'applications matérielles ("baremetal").

Il n'en reste pas moins que RISC-V a bousculé le monde du microprocesseur, à une époque où Intel et ARM monopolisent chacun leur marché.

V. CONCLUSIONS PERSONNELLES

➢ Hugues Almorin

Lors de ce projet, j'ai eu l'opportunité de me familiariser avec une plateforme de développement FPGA autre que celles de Xilinx et Altera/Intel. Les erreurs de manipulations m'ont permises de mieux comprendre le fonctionnement de cette carte Lattice. Le projet en lui-même est très intéressant du point de vue de la variété des designs autour de l'ISA. La prochaine étape que j'aurai apprécié aborder serait la conception d'un RISC-V.

➢ Yann Divet

Ce projet m'a permis de découvrir plus en profondeur les processeurs softcore, je n'avais en effet pas eu cette opportunité en 2ème année car j'avais suivi l'option traitement du signal et de l'image. Durant ce projet j'ai pu prendre le temps de me familiariser avec le vocabulaire relatif aux architectures de processeur, les différentes fonctionnalités qu'un processeur peut posséder (support des flottants ou non, prédictions de branchement ou non, etc…) ainsi que les différent outils logiciels nécessaires à la conception et à la programmation d'un processeur softcore. Je suis cependant un peu déçu de ne pas avoir réussi à tester les performances du f32c. J'ai la sensation de ne pas avoir produit grand-chose, mais d'avoir appris beaucoup.

➢ Camille Monière

Dès sa présentation, j'ai été intéressé par ce projet. Je souhaitais découvrir un peu mieux le milieu des microprocesseurs. Or, cela m'a permis en plus de découvrir de nombreux acteurs de ce milieu ainsi que tous les enjeux autour du RISC-V, notamment l'appréhension de ARM.

Je pense que l'architecture RISC-V a énormément de potentiel, notamment pédagogique, puisqu'elle permet, de chez soi, de réaliser une ébauche de processeur, à la manière d'un "Linux From Scratch" mais pour un processeur.

C'était de plus la première fois que j'assumais le rôle de responsable de projet et ce fut une expérience intéressante.

➢ Hugo Poupard

J'ai trouvé ce projet très intéressant, notamment l'aspect "modularité" des processeurs basés sur le jeu d'instruction RISC-V. En effet, nous avons pu observer au cours de nos recherches la grande diversité des implémentations disponibles, allant du simple microprocesseur au processeur capable de faire tourner un système Linux. Cette modularité en fait leur force, permettant à chacun de réaliser son processeur RISC-V adapté à ses besoins.

D'un point de vue personnel, je trouverai intéressant d'étudier ce type d'implémentation dans le cadre de l'UE de conception conjointe matériel/logiciel avec M. Kadionik (dans laquelle nous étudions déjà comment compiler des images Linux et implémenter des softcores sur FPGA ou Zynq).

VI. SOURCES

[1] : "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20181221-Public-Review-draft", Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation, December 2018.

[2] : Site officiel de la RISC-V Foundation, https://riscv.org/

[3] : Wikipédia l'encyclopédie libre, https://fr.wikipedia.org/

[4] : GitHub officiel f32c, https://github.com/f32c

[5] : Site du projet fpgarduino, http://www.nxlab.fer.hr/fpgarduino/

[6] : Page Gitlab INRIA de Comet, https://gitlab.inria.fr/srokicki/Comet

[7] : Comparatif des licences par GNU project, https://www.gnu.org/licenses/license-list.fr.html

[8] : Site officiel Mentor Graphics, page de Catapult HLS, https://www.mentor.com/hls-lp/catapult-high-level-synthesis/

[9] : GitHub official Bluespec Piccolo, https://github.com/bluespec/Piccolo

[10] : GitHub official du picoRV32 de Clifford Wolf, http://github.com/cliffordwolf/picorv32

[11] : Site officiel de Lattice, https://www.latticesemi.com/view_document?document_id=49312

[12] : Site officiel du projet lowRISC, https://www.lowrisc.org/

[13] : Nios®II Performance Benchmarks, DS-N28162004 | 2018.10.15

[14] : The MicroBlaze Soft Processor: Flexibility and Performance for Cost-Sensitive Embedded Designs, WP501 (v1.0) April 13, 2017, page 5, www.xilinx.com

[15] : Listes des architectures ARM et leurs performances sur Wikipédia, https://en.wikipedia.org/wiki/List_of_ARM_microarchitectures

[16] : Site officiel de Bluespec Inc., https://bluespec.com/piccolo-risc-v-core/