Bordeaux INP
ENSEIRB
MATMECA

Filière Électronique - PR310

Rapport de projet avancé en systèmes embarqués

Comparaison d'implémentations RISC-V

Élèves :
Arthur DUCHEIX
Thomas GUARETON
Léo PAJOT
Filipe POUGET

Professeur :
Jérémie CRENNE

28 janvier 2022

Table des matières

Introduction

Les récents développements dans le domaine des architectures de processeurs ont changé la façon dont chacun crée et utilise de telles technologies : la tendance du matériel open-source n'a cessé de croître, et, si nos formations sont à l'image des besoins de l'industrie, le "libre" (qu'il soit matériel ou logiciel) semble à présent incontournable. Plus spécifiquement, dans le domaine des architectures de micro-processeurs, de très nombreuses alternatives aux solutions propriétaires proposées par les leaders du marché voient le jour, avec comme ISA (Instruction Set Architecture) populaire, le RISC-V. Il incombe alors à l'ingénieur de faire le bon choix parmi la multitude d'alternatives proposées !

Objectifs et organisation

Au travers de ce projet, nous avons souhaité non seulement étudier plusieurs implémentations de processeurs RISC-V différentes, mais également prendre en main la *chaîne d'outils* (toolchain) nécessaire à leur bonne exploitation : les architectures RISC-V étant par nature modulaires, les outils de compilation sont également paramétrables afin de cibler l'architecture adéquate.

Nous avons donc divisé notre équipe en deux binômes afin, d'une part, de faire correspondre nos activités aux affinités personnelles de chacun, et afin, d'autre part, de pouvoir travailler en parallèle sur les deux sujets suivants :

- implémentation d'un processeur RISC-V micro-codé en SystemVerilog et enrichissement de celui-ci (Filipe et Léo),
- mise en œuvre de la toolchain nécessaire à la compilation de programmes pour cibles RISC-V et étude d'architectures publiques (Thomas et Arthur).

Nous tâcherons donc, au travers de ce rapport, de décrire les progrès que chacun a fait au cours de ce projet, qui malgré sa courte durée, nous a permis de progresser et de nous préparer à nos expériences futures, en particulier au Projet de Fin d'Études.

1 Comparaison de différentes implémentations de processeur RISC-V

1.1 Introduction

Dans cette partie nous allons comparer différentes implémentations de processeurs RISC-V open source trouvés sur Internet. Cette comparaison porte sur les ressources utilisées sur le FPGA et sur les performances évaluées. Pour cela il faut prendre quelques outils en main afin de pouvoir correctement les utiliser. La première étape a été d'installer Linux afin de pouvoir utiliser ces différents outils.

1.2 La Toolchain

Un des principaux outils à comprendre et à prendre en main a été la **toolchain** associée au RISC-V. Mais avant de s'intéresser à une **toolchain** en particulier il est important d'en présenter le principe de fonctionnement :

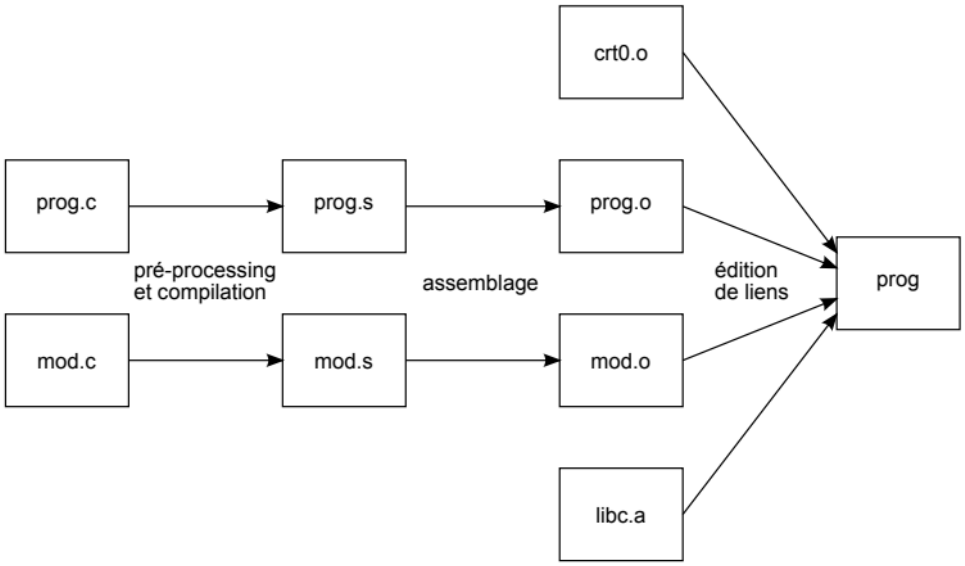

FIGURE 1 – Principe de fonctionnement d'une toolchain

Une toolchain est un programme qui met en lien et appelle dans le bon ordre différents autres programmes (le pré-processeur, le compilateur, l'assembleur, l'éditeur de lien). Si on donne en entrée un fichier .c tous ces sous programmes seront appelés, si on lui donne un fichier .s (assembleur) la partie pré-processing et compilation sera ignorée. En sortie, nous récupérons un exécutable binaire.

Dans le cadre de notre projet, nous avons du installer une toolchain ciblée RISC-V. Il existe une multitude d'extensions au RISC-V et pour chaque extension, il existe une toolchain. Nous nous sommes pour notre part concentrés sur l'extension classique **RV32I** et avons donc installé la toolchain associée.

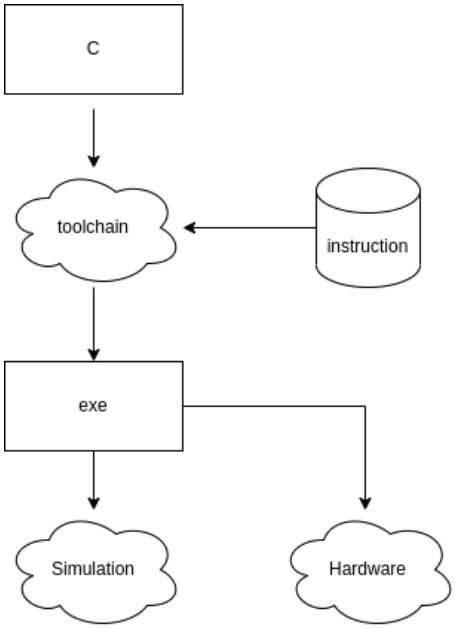

FIGURE 2 – Utilisation de la Toolchain RV32I

Grâce à cette toolchain, nous pouvons écrire un programme en C, nous récupérons ensuite un exécutable que nous pourrons tester soit en simulation soit sur une cible réelle telle que la carte Nexys A7.

1.3 Dhrystone

Un autre outils utile à la comparaison des différents processeurs est Dhrystone. C'est un benchmark, c'est-à-dire un programme de test de performances, crée par Reinhold Weicker en 1984. L'unité utile est le DMIPS/MHz, le Drhystone Million Instructions Per Second Per Mega Hertz qui représente donc le nombre d'instructions Dhrystone réalisées par seconde et par MHz. Si le processeur est à 100 DMIPS et qu'il fonctionne à 100 MHz alors il est à 1 DMIPS/MHz. Dans la mesure du possible nous utiliserons ce benchmark afin d'évaluer les performances des différents processeur pour pouvoir les comparer.

1.4 Nomenclature et extensions

Afin de pouvoir savoir quel processeur comparer il faut tout d'abord clarifier la nomenclature. Le nom d'un processeur RISC-V est régit par une nomenclature stricte.
Tout d'abord, la base est de renseigner les lettres "RV" afin d'indiquer que c'est un processeur **R**ISC-**V**.
Ensuite il faut préciser le nombre de bits (32,64 ou 128).
Enfin il faut préciser l'extension du jeu d'instructions, il en existe un certains nombre dont voici une liste non-exhaustive :

— **I** jeu d'instruction standard pour manipuler des entiers (integers)

— **C** extension standard d'instructions compressées
— **M** extension standard de multiplications et divisions entières
— **P** extension standard d'instructions SIMD compactées (Packed SIMD)

Ainsi, le processeur RV32IM est un processeur RISC-V sur 32 bits pouvant faire des multiplication et divisions entières.

1.5 PICORV32

1.5.1 Présentation

Le PICORV32 est un processeur open source créé par Clifford Wolf, il peut implémenter les jeux d'instructions RV32I, RV32M, RV32C, RV32IMC. Il a été créé dans le but d'être intégré dans des designs sur FPGA ou ASIC, ainsi l'objectif de ce processeur est d'être léger. Dans notre cas nous nous focalisons sur le jeu d'instruction basique c'est à dire le RV32I.

1.5.2 Expérimentation

Ce processeur a été le premier que nous avons testé, il nous a permis d'installer la toolchain et nous avons fait face à de nombreux obstacles lors de l'installation de celle-ci. Au final, nous n'avons pas testé cette implémentation sur FPGA cependant nous avons quand même pu obtenir des résultats de synthèse sur Vivado.

1.5.3 Ressources

Pour le domaine des ressources utilisées ce processeur tient ses promesses. Voici ce qu'il occupe :

PICORV32	NEXYS 4 A7
LUT	835 (1.3%)
LUTRAM	48 (0.35%)
FF	583 (0.46%)

FIGURE 3 – Ressources utilisées par le PICORV32 sur cible Nexys 4 A7

Le FPGA utilisé n'est pas celui qui possède le plus de ressources (par rapport à la Genesys 2 de chez Digilent par exemple) pourtant on note que le processeur utilise une partie infime de ces ressources, en occupant moins de 2% des LUTs disponibles par exemple. Cependant qu'en est-il des performances ?

1.5.4 Performances

Dhrystone nous est fourni dans le GitHub du projet, il ne reste plus qu'à le lancer. Cependant il faut s'assurer de bien utiliser la toolchain riscv32i et non pas la riscv32im qui est

utilisée de base, pour cela une petite modification du Makefile est suffisante. Les résultats sont les suivants :

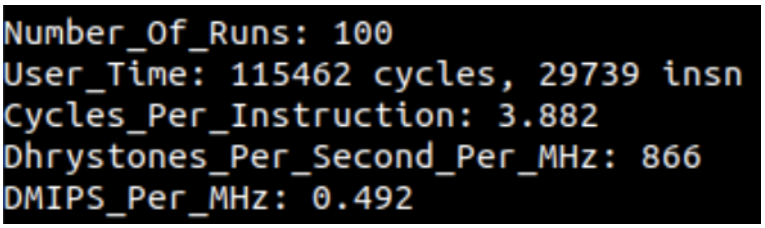

FIGURE 4 – Résultat du dhrystone pour le PICORV32

On note ainsi moins de 0.5 DMIPS/MHz en travaillant à 100MHz, nous notons cette valeur afin de la comparer plus tard avec les autres processeurs.

1.6 NEORV32

1.6.1 Présentation

Le NEORV32 est un processeur open source implémentant le jeu d'instruction RISC-V, le but est d'être un processeur auxiliaire prêt à l'emploi dans un design SoC. Il possède 14 extensions du RISC-V et encore une fois nous nous focalisons sur le RV32I.

1.6.2 Expérimentation

Ce processeur a été le plus simple à prendre en main, nous avons pu le simuler mais aussi le mettre sur carte Nexys 4 A7, les seuls ajouts d'entrées sorties furent un liaison UART, un bouton restart et une LED.
Nous avons pu exécuter des programmes sur cette carte utilisant le NEORV32 tels que le célèbre *Hello World* ou encore un *Game of Life*.
Nous avons pu voir les ressources utilisées mais nous n'avons pas pu exécuter le *dhrystone* nous allons donc analyser les valeurs données par le créateur.

1.6.3 Ressources

NEORV32	NEXYS 4 A7
LUT	1995 (3%)
BAM	8 (6%)
FF	1482 (1%)

FIGURE 5 – Ressources utilisées par le NEORV32 sur cible Nexys 4 A7

Nous pouvons observer que ce processeur est lui aussi très léger bien qu'un peu plus lourd que le précèdent, cependant ces valeurs sont aussi à relativiser puisque ces ressources sont aussi utilisées par la liaison UART, le bouton et la LED.

1.6.4 *Performances*

```
NEORV32: DMIPS/s:              76923
NEORV32: DMIPS/MHz:            769
```

FIGURE 6 – Résultat du dhrystone pour le NEORV32

Contrairement aux ressources utilisées, nous voyons ici une grande différence de performances. En effet sans être beaucoup plus gourmand que le PICORV32 ce processeur est plus de 1500 fois plus performant atteignant 769 DMIPS/MHz, à une fréquence de travail de 100MHz.

1.7 POTATO

1.7.1 *Présentation*

Le processeur POTATO est un processeur RISC-V très simple décrit en VHDL qui cible les FPGA. Comme les processeur précédents, c'est un projet prêt à l'emploi grâce à un design SoC qui nous est fourni. Il supporte le RV32I ce qui est idéal dans le cadre de nos expérimentations.

1.7.2 *Expérimentation*

Le projet est présenté comme prêt à l'emploi. Pour le tester une section **software** est présente sur le git du projet. Nous avons réussi à générer un bitstream ainsi qu'à charger des programmes tels qu'un **hello world** ou un **check sum sha256**. Cependant, la partie communication UART du projet n'était pas fonctionnelle en l'état et malgré de nombreux tests, nous n'avons pas réussi à visualiser quoi que ce soit sur le port de communication série de notre carte Nexys A7.

1.7.3 *Ressources*

POTATO	NEXYS 4 A7
LUT	2883 (5%)
LUTRAM	80 (0.5%)
FF	2035 (1.5%)
BRAM	40 (30%)

FIGURE 7 – Ressources utilisées par le POTATO sur cible Nexys 4 A7

Ce processeur est lui aussi assez léger, bien que plus gourmand en LUT et FF. Il utilise aussi beaucoup de BRAM ce qui peut sembler assez surprenant. Mais cela s'explique sûrement d'une part à cause du design SoC et d'autre part par le fait que ce type de projet est rarement optimisé et que le seul but du désigner était de se familiariser avec l'écriture d'un processeur RISC-V et de le tester avec des programmes minimalistes.

1.7.4 Performances

Étant donné que la communication UART ne fonctionnait pas et qu'aucunes données ne sont mises à disposition par le créateur, nous ne pourrons donc pas comparer les performances au **dhrystone** de ce processeur avec les autres que nous avons testé.

1.8 RVSoC

1.8.1 Présentation

Le projet RVSoC (RISC-V System on Chip) est un projet de recherche et développement qui concerne l'architecture RISC-V et qui cible les FPGA. C'est un projet décrit en Verilog HDL. Il vise à faire marcher un Linux sur la carte.

1.8.2 Expérimentation

Ce projet a été pensé pour fonctionner sur notre carte, la Nexys A7. Un guide utilisateur était fourni, nous avons donc pu générer le bitstream du projet puis l'exécuter sur la carte. Après avoir suivi convenablement le tutoriel, nous avons pu exécuter les commandes linux classiques directement sur la carte.

1.8.3 Ressources

RVSOC	NEXYS 4 A7
LUT	10421 (16%)
BRAM	38 (28%)
FF	6379 (5%)

FIGURE 8 – Ressources utilisées par le RVSOC sur cible Nexys 4 A7

Ce projet est celui qui nécessite le plus de ressources, mais cela s'explique simplement par le fait que ce projet représente fait fonctionner un linux embarqué sur FPGA.

1.8.4 Performances

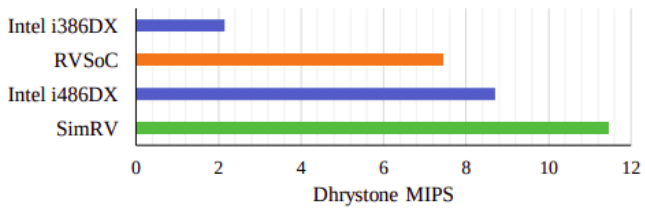

FIGURE 9 – Comparaison des performances du RVSoC avec des processeurs connus

Lorsque l'on s'intéresse aux performances de ce processeur, on oeut voir qu'il se situe entre deux processeurs Intel, cependant il faut mettre cela en perspective avec le fait que ce soit des processeurs vieux de plus de 30 ans.

2 Implémentation d'un processeur RISC-V micro-codé

2.1 Avant-propos

Nous avons également choisi, en plus de l'analyse de différents processeurs *off-the-shelf*, de nous attacher à la description de notre propre architecture, et ce pour plusieurs raisons :

— afin d'apprendre le langage SystemVerilog,
— afin de manipuler un type d'architecture encore jamais pratiqué,
— afin d'expérimenter la difficulté du processus d'itération sur une architecture pré-existante,
— afin de découvrir de nouveaux concepts et leurs implémentations, comme la manipulation d'instructions SIMD et le masquage d'empreinte de consommation.

Comme nous l'indiquons dans l'introduction de ce rapport, nous n'avons par pu, au terme de ce projet, présenter une réalisation fonctionnelle incorporant toutes ces propositions.

Justification du choix d'architecture Le processeur que nous avons choisi de développer est un processeur micro-codé, basé l'architecture proposée par M Jérémie Crenne pendant son cours EN226 à l'ENSEIRB-MATMECA. Nous avons certes fait ce choix car reprendre cette proposition d'architecture nous économisait des efforts de développement, mais aussi et surtout parce que nous avions déjà réalisé un travail similaire au cours du module EN210 de M Camille Leroux, consistant en la "conception d'un processeur [simple cycle] avec jeu d'instruction élémentaire".

Les architectures micro-codées possèdent cependant un avantage de taille par rapport à leurs pendants simple cycle : contrairement à ces dernières, l'ajout ou la modification des instructions existantes est simplifié, étant donné que l'exécution des instructions supportées par le processeur repose sur une machine d'état passant par une séquence de micro-états propre à chaque instruction. Si la modification à apporter ne nécessite pas de bloc matériel supplémentaire pour son exécution, sa mise en œuvre ne nécessite que la modification du micro-code. De même, une nouvelle instruction (ou même macro-instruction) peut potentiellement être implémentée sans coût supplémentaire (autre que celui de l'espace mémoire nécessaire pour stocker les micro-instructions la décrivant).

2.2 Implémentation du processeur basique

Ce processeur basique se divise en trois unités distinctes : **l'unité de traitement** qui a pour rôle d'effectuer des modifications sur les données à partir de l'instruction en cours, **l'unité de mémoire** qui stocke les différentes données, et **l'unité de contrôle** qui génère les signaux permettant aux données de suivre le bon chemin.

2.2.1 Unité de traitement

Comme dit précédemment, nous avons pris exemple l'architecture de M. Crenne pour notre processeur. Nous pouvons donc voir Figure 10 le datapath de l'unité de traitement sur laquelle nous nous sommes basés :

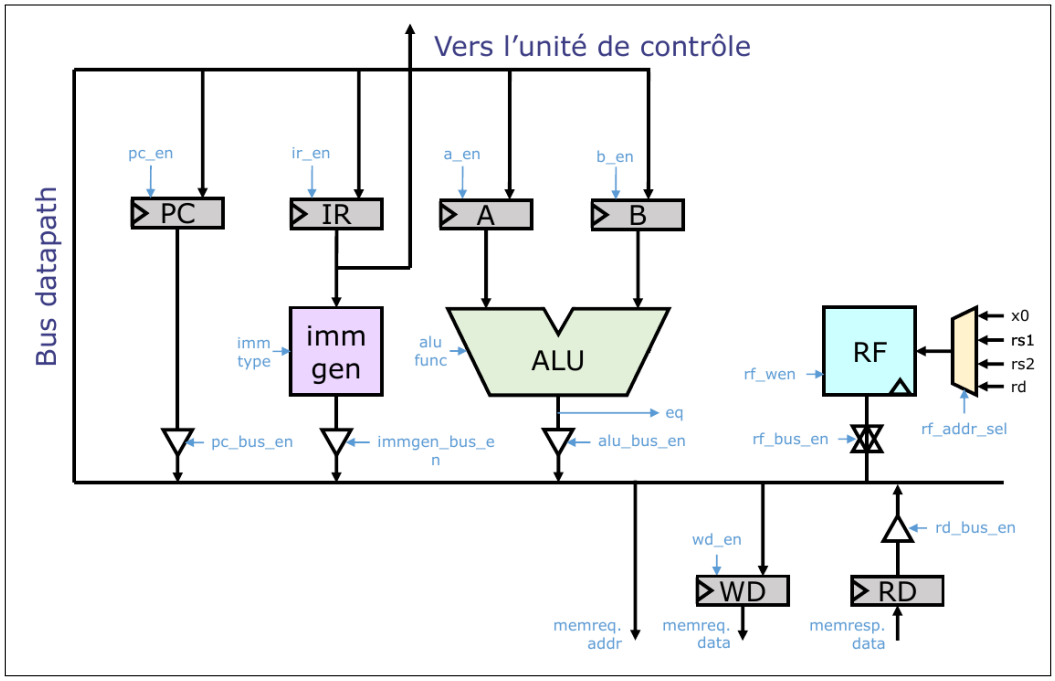

FIGURE 10 – Unité de traitement du processeur microcodé

Nous pouvons retrouver des similarités avec un processeur simple cycle : un registre pour mémoriser le PC, un bloc de génération de valeur immédiate, un ALU et une file de registre. Cependant on remarque que les entrées de tous ces blocs sont connectées aux sorties de tous les autres par un bus d'interconnexion. Seuls des registres en entrée et des buffers en sorties, contrôlés par des signaux venant de l'unité de contrôle, régulent le chemin que doivent prendre les données. Voyons comment fonctionne chaque partie de cette unité.

Bus d'interconnexion Ce bus permet de faire circuler les données à traiter dans toute l'unité. À chaque cycle d'horloge, il connecte des blocs différents. Les blocs connectés dépendent de l'étape (fetch, décode, ...) ou microétape (lorsqu'une étape s'effectue en plusieurs cycles) dans laquelle l'exécution se situe. Par exemple lorsque l'on veut lire une donnée de la file de registre pour s'en servir d'opérande à l'ALU, le registre A (ou B) et la sortie de la file de registre seront activés. Ce bus est également connecté à l'unité de mémoire lors de la lecture et l'écriture de celle-ci.

Un changement que nous avons effectué par rapport au schéma Figure 10 est qu'au lieu d'utiliser des buffers en sortie des blocs, nous avons mis un multiplexeur pour contrôler la valeur du bus.

Registre PC Ce registre sert tout simplement à mémoriser le PC. La valeur stockée sert à lire la prochaine instruction à exécuter. Elle est donc en sortie du mux lors du fetch pour lire la mémoire, mais c'est également à ce moment qu'elle sert d'opérande à l'ALU pour pouvoir l'incrémenter. Le registre est donc activé après l'incrémentation et lors d'une instruction `JUMP`.

Générateur de valeur immédiate Ce bloc sert comme son nom l'indique à générer une valeur immédiate à partir de l'instruction lue dans la mémoire. Le registre est donc actif lors

du fetch après la lecture en mémoire. De plus, comme nous pouvons le voir sur le datapath, c'est à partir de cette branche que l'unité de contrôle a accès à l'instruction.

La position des bits à lire étant différente selon l'opération, le type de valeur immédiate dépend de l'instruction. Cette valeur se retrouve sur le bus lors d'une opération sur une valeur immédiate : `ADDI`, offset lors d'une écriture/lecture de la mémoire, ...

Les instructions que nous avons implanté nous obligent à définir deux types de valeurs immédiates :

— `ADDI/JR/LW : imm = {20'h0, instr[31:20]}`
— `SW : off = {20'h0, instr[31:25], instr[11:7]}`

ALU Ce bloc permet d'effectuer différentes opérations sur les opérandes en entrée. Avant de lire le résultat, les deux opérandes doivent bien évidemment être chargés l'un après l'autre. Ceux-ci viennent soit de la file de registres, soit du bloc de valeur immédiate. Seules deux opérations ont été implantées :

— simple addition non signée: `temp = opA + opB`, addition des deux opérandes lors des instructions `ADD` et `ADDI`, mais aussi lors du calcul d'une adresse avec un offset
— `+4 : temp = opA + 4`

File de registres Tout comme dans un processeur simple cycle, il s'agit de la mémoire interne au processeur. L'adresse à laquelle nous souhaitons accéder est définie par l'instruction. La lecture et l'écriture se font depuis le bus d'interconnexion : le mux empêche les données lues d'aller sur le bus lorsque ce n'est pas voulu, mais l'entrée en écriture est toujours connectée au bus (l'activation de l'écriture se fait avec les signaux de contrôle).

2.2.2 Unité de mémoire

Pour ce qui est de la mémoire, nous avons choisi une architecture de Von Neumann, c'est-à-dire que le programmes et les données sont stockées dans la même mémoire. Nous avons pris comme base de description un exemple fourni par Xilinx.

Pour ce qui est des entrées/sorties, nous avons comme seule sortie la donnée lue. En entrée, en plus du signal d'horloge, il y a :

— l'adresse dont la longueur dépend de la profondeur de la mémoire (arbitrairement nous avons choisi une profondeur de 1024 mots, l'adresse est donc sur 10 bits)
— la donnée d'entrée sur 32 bits, servant lorsque l'on souhaite écrire dans la mémoire
— *write enable* permettant d'activer l'écriture à l'état haut

L'adresse et la donnée à écrire sont toutes les deux lues sur le bus d'interconnexion. Il faut donc un registre pour mémoriser l'une des deux, avant d'envoyer la suivante.

Dans notre cas, l'adresse est sur 10 bits alors que nous avons un bus sur 32. Il faut donc "rogner" la donnée. Il faut également noter que du point de vue du processeur, comme nous manipulons des données sur 32 bits, soit 4 octets, passer d'une adresse à la suivante revient à faire `+4`. Il faut donc également enlever les 2 bits de poids faible.

2.2.3 *Unité de contrôle*

Tout comme pour l'unité de traitement, nous nous sommes basés sur un exemple pour l'unité de contrôle :

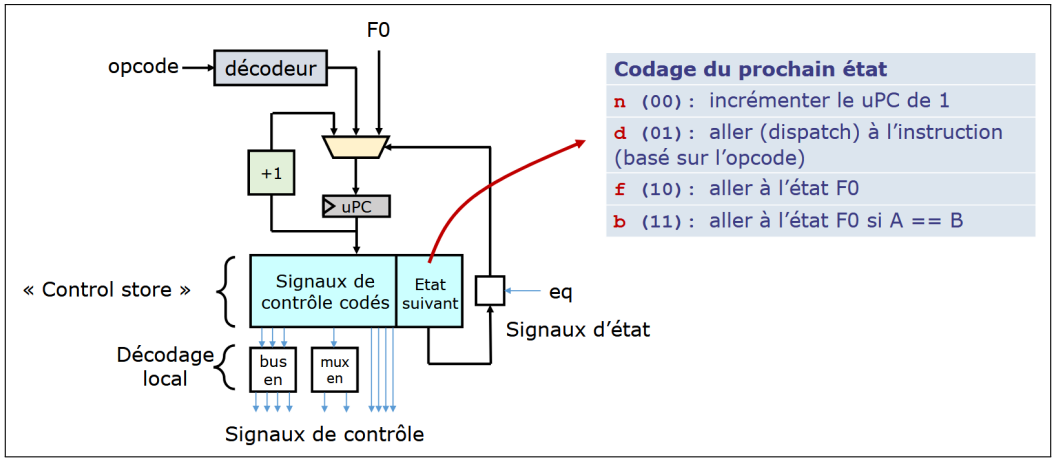

FIGURE 11 – Unité de contrôle du processeur microcodé

Il s'agit simplement d'un FSM couplée avec une table de contrôle. Le but de cette unité est de générer les signaux de contrôle en fonction de l'étape de l'exécution dans laquelle se trouve le processeur. Ces étapes sont FETCH, DECODE, EXECUTE, MEMORY et WRITE BACK dans un processeur simple cycle. Dans notre cas, ces étapes sont divisées en microétapes à cause des limites de l'architecture microcodée. Le fonctionnement reste tout de même assez simple.

```
FETCH
F0 : memreq.addr ← PC; A ← PC
F1 : IR ← RD
F2 : PC ← A + 4; goto INST

ADDI
AI0 : A ← RF[rs1]
AI1 : B ← sext( imm_i )
AI2 : RF[rd] ← A + B; goto F0
```

FIGURE 12 – Exemple de microétapes de l'instruction `ADDI`

Tout d'abord la table de contrôle sert à avoir les signaux de contrôle en fonction de l'étape, mais sert également à indiquer la prochaine étape. Il y a 3 possibilités pour la nouvelle étape :

— `F0` : début du fetch, atteint à l'initialisation ou après avoir fini d'exécuter une instruction
— sortie du décodeur : une fois le fetch fini, nous connaissons l'opcode de l'instruction, nous pouvons donc commencer à l'exécuter
— `+1` : passage d'une microétape à la suivante

2.3 Masquage de consommation

Le processeur basique étant maintenant fonctionnel, nous avons cherché à l'améliorer. La première modification était de masquer la consommation en homogénéisant le nombre le microé-

tapes nécessaires à chaque instructions. C'est une modification assez simple, il faut cependant faire attention à ce que l'ajout de microétapes n'apporte pas des disfonctionnements, comme l'écriture dans un registre de la file de registre alors que sa valeur est nécessaire au bon fonctionnement du programme.

Voyons d'abord les instructions implémentées :

— `ADD` : 3 microétapes
— `ADDI` : 3 microétapes
— `SW` : 4 microétapes
— `LW` : 4 microétapes
— `JR` : 1 microétape

Il faut donc atteindre 4 microétapes par instruction.

Pour `ADD` et `ADDI` il n'y a qu'une étape à ajouter. Nous avons donc juste rajouté la lecture d'un registre de la file de registre.

Pour `JR`, il faut rajouter 3 microétapes. Or `ADD` en a 3. Nous avons donc juste ajouter une instruction `ADD` après `JR`. Pour ne pas changer la valeur d'un registre nous avons juste eu à faire une addition d'un registre quelconque avec le registre `x0` (cablé à 0), puis de réécrire dans le registre de départ.

2.4 Résultats de synthèse

2.4.1 Processeur basique

Nous avons réalisé une première synthèse de notre processeur après avoir fini d'implémenter les instructions suivantes:

`ADD, ADDi, LW, SW, JR.`

Processeur maison	Nexys 4 A7
LUT	160 (0.25%)
FF	162 (0.13%)
BRAM	1 (0.74%)

TABLE 1 – Ressource estimées par le synthétiseur de Vivado pour notre processeur "minimaliste"

2.5 Processeur avec masquage de l'empreinte de consommation

Nous avons ensuite implémenté la méthode de masquage de la consommation décrite ci-dessus, puis synthétisé à nouveau l'architecture.

Processeur maison	Nexys 4 A7
LUT	170 (0.27%)
FF	162 (0.13%)
BRAM	1 (0.74%)

TABLE 2 – Ressource estimées par le synthétiseur de Vivado pour notre processeur "minimaliste" implémentant une technique de masquage de la consommation

La variation très légère des ressources nécessaires prouve ici la force de modularité de l'architecture micro-codée : les instructions peuvent être totalement repensées/altérées sans que cela impacte le design matériel. Cependant, ces résultats ne permettent pas de voir l'impact de cette modification du comportement du processeur sur ses performances : en effet, en égalisant par le haut le nombre de micro-instructions par instruction, nous diminuons le débit effectif du processeur, avec un impact bien plus disproportionné sur les instructions les plus courtes, qui se voient fortement rallongées.

Il est à noter que nous sommes ici dans un cas très simple où l'on ne manipule que des additions... Si nous avions implémenté la multiplication, et ses 36 micro-étapes, nous aurions sûrement dû choisir de la masquer ou non, car allonger toutes les instructions à 36 cycles aurait été catastrophique pour les performances de notre processeur.

2.6 Ajout d'instructions SIMD

Nous avons ensuite souhaité explorer la possibilité de manipuler des données avec des instructions de type Single Instruction Multiple Data (SIMD). De telles instructions ajoutent la possibilité, lorsque la précision des données à manipuler le permet, d'appliquer simultanément à plusieurs données différentes la même instruction, fournissant ainsi plusieurs résultats à la fois. En clair, elles permettent d'effectuer des calculs en parallèle sur une même unité de calcul. Dans le cas de notre processeur 32 bits, nous pouvons par exemple imaginer effectuer non pas une addition 32 bits mais 2 additions 16 bits, ou encore 4 additions 8 bits. Leur utilisation augmente alors le débit théorique en terme de nombre de données traitées par instruction.

	Nombre de cycles	Taille des opérandes	Nombre d'opérandes	Débit (nombre d'opérandes par cycle)
ADD	6 (en comptant le fetch)	32	2	1/3
ADD16		16	4	2/3
ADD8		8	8	4/3

TABLE 3 – Illustration des gains permis par l'utilisation d'instruction SIMD

Afin de trouver une définition cohérente avec notre architecture, nous nous sommes ba-

sés sur la spécification de l'extension RISC-V (encore non officielle) *Packed*[1], permettant le traitement et la manipulation de telles données.

Ce type d'instruction est présent dans de nombreux processeurs aujourd'hui, qu'ils soient destinés au calcul haute performance ou à l'embarqué, au vu des optimisations qu'elles permettent dans les domaines du traitement du signal, de la simulation physique, ou encore du chiffrement de données. Pour preuve, nous pouvons par exemple citer le mémoire de Master de D.A.M. Koene, *Implementation and Evaluation of Packed-SIMD Instructions for a RISC-V Processor*. Celui-ci nous renseigne notamment sur les possibles optimisations logiques à notre disposition, proposées par l'auteur à travers sa démarche d'extension d'un processeur RISC-V open-source CVA6[2]. On y retrouve un grand nombre d'informations clé, la démarche de l'auteur étant similaire à (et bien plus approfondie que) la nôtre, comme par exemple le tableau suivant.

Table 3.1: Overview of the basic SIMD instructions used in the investigated applications, the numbers denote the element size the instruction operates on.

		Machine learning	Video encoding	Computer vision
Add and subtract	Add	16, 32, 64	8, 16, 32, 64	
	Saturated add	16	16	16, 32
	Horizontal add	32	16, 32	
	Sub	16, 32	8, 16, 32	
	Saturated sub		8	8, 16
	Horizontal sub		16	
Shift	Shift left logical	16, 32, 64	16, 32, 64	
	Shift right logical	16, 32, 64	16, 32, 64, 128	16
	Shift right arithmetic	32	32	
Data movement	Pack signed	32 to 16	32 to 16	16 to 8
	Pack unsigned	16 to 8	16 to 8	16 to 8
	Unpack	8, 16, 32, 64	8, 16, 32, 64	
	Shuffle single input	16, 32	8, 16, 32, 64	
	Shuffle two inputs		8, 16, 32, 64	16, 32
	Zero extend elements		8 to 16, 8 to 32	8 to 16, 16 to 32
	Sign extend elements		8 to 16, 16 to 32	16 to 32
	Insert element	16		
Comparison	Min unsigned	8	16	8
	Min signed		8	16, 32
	Max unsigned	8		8
	Max signed		8, 16	16
	Greater than	32	8, 16, 32	16, 32
	Less than			16
	Equal		8, 16, 32	16, 32
Multiplication	Multiply	16, 32	16, 32	16

FIGURE 13 – Tableau extrait du mémoire de D.A.M. Koene analysant les cas d'utilisation de différentes instructions SIMD tirées de la proposition d'extension *Packed*

On constate ici que de nombreux domaines peuvent en effet bénéficier des gain permis

1. Lien vers le dépôt GitHub de la proposition d'extension
2. Lien du dépôt GitHub du processeur CVA6

par l'utilisation d'instruction SIMD. On remrque également que l'idéal serait de proposer un processeur permettant des calculs sur des éléments de taille variées, étant donné que nous ne pouvons prédire à l'avance la précision demandée par l'application.

Format des données En effet, cette extension ne serait pas complète si elle ne fournissait que les instructions de calcul en elle-mêmes. Pour mettre en œuvre ces calculs parallélisés, un travail de mise en forme est nécessaire pour regrouper plusieurs données initialement réparties dans plusieurs registres, dans un seul registre. Il faut alors faire appel à des instructions de *packing* afin de préparer les données, mais aussi afin de les "décompresser".

	31 16	15 0
Registre r1	a[15:0]	b[15:0]
Registre r2	c[15:0]	d[15:0]
Registre r3	a[15:0]	d[15:0]

TABLE 4 – Exemple d'application de l'instruction PK**TB**16 r3, r1, r2 : le **T**op du registre 1 et le **B**ottom du registre 2 sont concaténés dans le registre 3

La décompression des données est cependant plus délicate car elle nécessite de prendre en compte le format des données : ce n'est plus une simple sélection de bits.

	31 24	23 16	15 8	7 0
Registre r1	a[7:0]	b[7:0]	c[7:0]	d[7:0]
Registre r2	sext16(b[7:0])		sext16(d[7:0])	

TABLE 5 – Exemple d'application de l'instruction **S**UNPKD8**20** r2, r1 : les octets **2** et **0** du registre 1 sont étendus sur une représentation 16 bits **S**ignée, et concaténés dans r2

Le très grand nombre de variations possibles autour de ces instructions fait que la proposition de spécification *Packed* est très fournie. Des choix ont tout de même dû être faits par les rédacteurs de cette spécification afin de faire rentrer ce grand nombre d'instructions dans l'espace restreint des combinaisons disponibles, dicté par la spécification RISC-V à travers les formats d'instruction.

Table 37. Instruction Encoding for funct3[14:12]==0b000 and funct7[31:25]==0b1010110 (oneop).

ONEOP == 1010110								
subf5	**[2:0]**							
[4:3]	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
00	insb (RV32/RV64)				insb (RV64)			
01	sunpkd810	sunpkd820	sunpkd830	sunpkd831	zunpkd810	zunpkd820	zunpkd830	zunpkd831
10	kabs8	kabs16	kabs32	sunpkd832	kabsw			zunpkd832
11	swap8							

FIGURE 14 – Tableau extrait de la proposition de spécification *Packed*

On observe notamment que certaines combinaisons de décompression n'existent pas, comme SUNPKD821. Si un tel comportement est nécessaire, il faudra alors faire appel à de nouvelles instructions, comme SWAP8, qui échange la position des octets de chacun des demi-mots du registre source (SWAP16 existe également).

Surcoût de manipulation Dans tous les cas, dès lors que des instructions SIMD sont utilisées, une pénalité en terme de temps de traitement est donc inévitable pour la préparation puis l'interprétation des données manipulées. Ces pénalités viennent bien entendu réduire le débit effectif de l'architecture, et nuancent donc les chiffres présentés dans le tableau 3. Il revient donc aux personnes chargées de l'optimisation de l'application développée de donner les directives adéquates au compilateur afin de minimiser ce surcoût au regard des gains permis par la parallélisation des calculs.

Compromis performance/surface silicium Cependant, cette flexibilité supplémentaire a un coût. En effet, si on veut conserver le comportement attendu de l'opération à transposer en SIMD, une adaptation de l'unité arithmétique est nécessaire : pour l'exemple de l'addition, il est essentiel de s'assurer que la possible retenue d'un des deux résultats ne "contamine" pas les calculs effectués en parallèle!

Arithmétique spécifique Cette adaptation de l'unité arithmétique peut prendre multiples formes, mais la création d'opérateurs dédiées (additionneur, soustracteur, etc) est probablement l'une des solutions les plus simples à développer. Reste alors à concevoir leur logique interne !

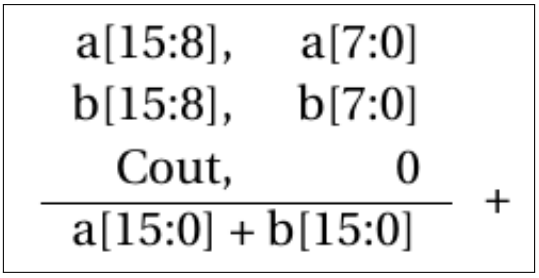

FIGURE 15 – Traitement de l'addition 16 bits proposé par D.A.M. Koene

L'addition est le cas le plus simple, et le découpage en blocs de 8 bits proposé par D.A.M. Koene permet de gérer les additions sur différents formats d'éléments (8, 16, 32 bits) : seul le traitement de la retenue du bloc de 8 bits précédent change en fonction du format traité.

Enfin, l'auteur porpose aussi une méthode optimisée de gestion des *overflow* ou *underflow* qui peuvent avoir lieu au cours des calculs. Ces cas spécifiques sont d'ailleurs prévus dans l'extension *Packed*, où figurent de multiples versions des instructions de base, suivant le comportement voulu : saturation, *wrap-around*, *signed* et *unsigned halving* (perte du LSB au profit du débordement), etc.

Conclusion

Ce projet a été séparé en deux parties : prise en main et manipulation de la toolchain d'une part, et implémentation d'un processeur d'autre part. Cette séparation avait, à l'origine, été pensée pour pouvoir les faire converger vers la compilation d'un exécutable pour notre processeur "maison", supportant seulement un sous-ensemble des instructions des spécifications visées. Cependant, nous n'avons pas pu atteindre cet objectif, qui aurait pourtant été l'occasion d'explorer les solutions à notre disposition pour configurer de manière personnalisée la *toolchain*, ainsi que d'appréhender les différents formats exécutables fournis par celle-ci, dans l'optique de les charger dans notre processeur.

Même si nous n'avons pas atteint ces objectifs, probablement trop ambitieux, nous avons tout de même, au cours de ce projet, eu la possibilité de nous familiariser avec les méthodes employées par les développeurs *softcore* afin de rendre leurs architectures modulables. Nous avons également découvert la démarche de la mesure de performance de ces architectures à travers des outils de *benchmarking*, ainsi que les écueils à éviter lorsqu'on compare différents systèmes (processeur VS SoC par exemple). Pour ce qui est de la description de notre processeur "maison", celle-ci nous a permis de nous entraîner davantage à la manipulation de cet outil souvent capricieux (parfois à raison!) qu'est Vivado, et ce dans un nouveau langage de description. Cet exercice a enfin été pour nous le prétexte pour découvrir de nouveaux sujets, comme le calcul parallèle grâce aux instructions SIMD, ou encore le masquage de l'empreinte de consommation du processeur.

Bibliographie

https://fr.wikipedia.org/wiki/Dhrystone

https://github.com/YosysHQ/picorv32

https://fr.wikipedia.org/wiki/RISC-V#Nomenclature_des_extensions

https://github.com/stnolting/neorv32

https://stnolting.github.io/neorv32/#_rationale

https://chamilo.grenoble-inp.fr/courses/ENSIMAG3MM1LDB/document/c99_compil.pdf

https://github.com/skordal/potato

https://www.arch.cs.titech.ac.jp/wk/rvsoc/doku.php

https://arxiv.org/pdf/2002.03576

https://github.com/riscv/riscv-p-spec

https://repository.tudelft.nl/islandora/object/uuid:c4162ff8-9419-4434-852d-c1c3297df808/datastream/OBJ/download