ENSEIRB-MATMECA
Bordeaux-INP

3A Électronique
Systèmes embarqués
PR310

Projet RISCV'EIRB

Élèves :
Aymeric HERNANDEZ
Samuel BARIS
Florian MONDELAIN
Ayoub YAQINE

Enseignant :
Christophe JEGO

26 janvier 2024

Table des matières

Table des figures

1 Introduction

Lors des années précédentes, les élèves de l'Enseirb ont travaillé lors de différents projets à l'école sur un processeur, RISCV'EIRB. Comme son nom l'indique, ce processeur est basé sur le jeu d'instruction RISC-V 32I. L'année dernière, les élèves ont finalisé l'écriture de l'architecture du processeur. La tâche qui nous a été confiée fut de tester ce processeur de la manière la plus exhaustive possible. Pour ce faire, nous nous sommes aidés d'un livre qui nous a servi de guide et qui décrit une méthode de développement partant d'une description haut niveau. Dans ce rapport, nous allons décrire la méthode à employer afin de réaliser des tests sur un processeur à la manière d'un manuel d'utilisation. Nous verrons dans un second temps comment nous avons utilisé cette méthode pour notre RISCV'EIRB et comment les résultats de ces tests nous ont permis de corriger et de valider le processeur.

2 La carte PYNQ-Z2

La méthode que nous allons utiliser pour tester notre processeur utilise une carte PYNQ-Z2. Cette carte FPGA est un SoC (System on Chip) et possède donc un processeur ARM dont nous allons nous servir afin de realiser les tests. On va donc, sur cette carte, pouvoir programmer le FPGA avec notre processeur comme on le ferait avec n'importe quelle carte, mais ici c'est le processeur ARM de la carte qui va contrôler les entrées et sorties de notre processeur. On écrira alors un code en C qu'on fera exécuter par le processeur ARM qui chargera la mémoire de notre RISCV'EIRB, qui lancera l'exécution de ce programme chargé en mémoire et qui lira le contenu de la mémoire à la fin de l'exécution. Ceci rendra les tests bien plus faciles à réaliser une fois la configuration de la carte faite.

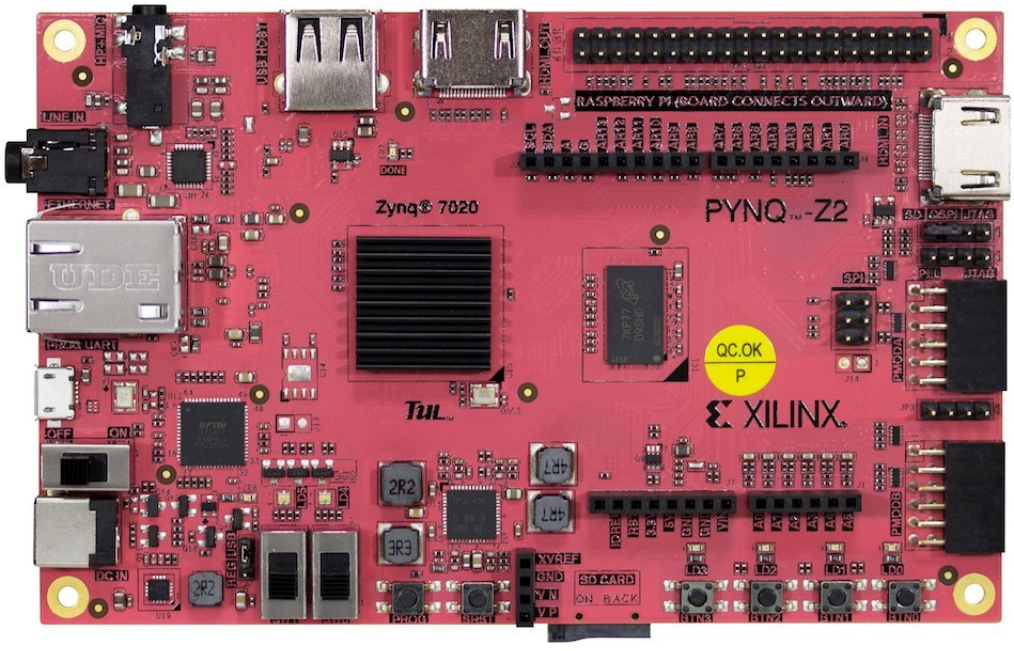

Figure 1 – PYNQ-Z2

3 Vivado

Afin de réaliser les tests, la première étape va être de générer le bitstream à partir de nos sources RTL à l'aide de l'outil Vivado. Durant ce procédé, il faudra essayer de minimiser le nom des paths car cela peut poser des problèmes. Il sera donc préférable de donner de courts noms à nos projets et de travailler dans une arborescence proche de root.

Il faut créer un projet RTL Vivado en s'assurant que la carte choisie est la bonne (ici Pynq-Z2 board). Il faut ensuite créer un block Design qui va nous permettre d'ajouter les différentes IP.

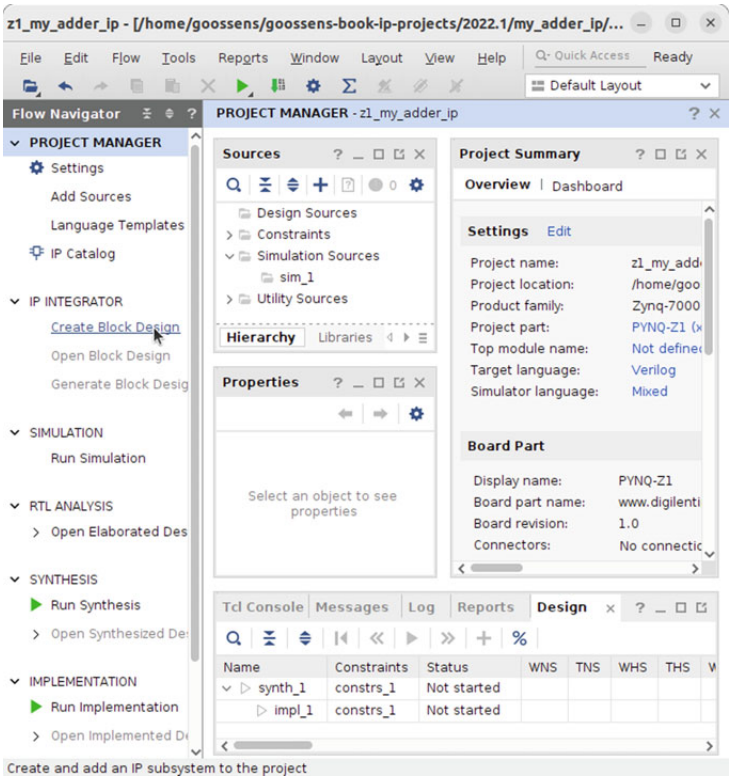

FIGURE 2 – Création du block Design

On peut ensuite ajouter l'IP ZYNQ7 Processing System à ce bloc Design qui est celle qui va permettre d'interfacer l'IP de notre RISCV'EIRB avec le processeur de la carte. On appuie sur le bouton Run Block Automation à chaque IP ajoutée.

Il va falloir créer l'IP RISCV'EIRB. Pour faire cela, on importe toutes les sources VHD du CPU et on définit comme top level la racine de ces sources. On peut ensuite créer notre IP en cliquant sur Create and Package New IP dans le menu Tools. On choisit Create a new AXI4 peripheral, on nomme cette IP comme on veut, on définit son nombre de registres et on l'ajoute au répertoire des IP. On peut finalement ajouter notre IP RISCV'EIRB et regénérer le layout. Dans notre cas, il a fallu ajouter une clock de 50Mhz à la place de la clock de 100Mhz en sortie de l'IP Zynq7. Cela ce fait en cliquant sur l'IP et en ajoutant une clock dans le menu PL Fabric Clocks de Clock Configuration. On relie à la nouvelle clock les éléments qui étaient reliés à l'ancienne. On obtient alors le block Design complet suivant :

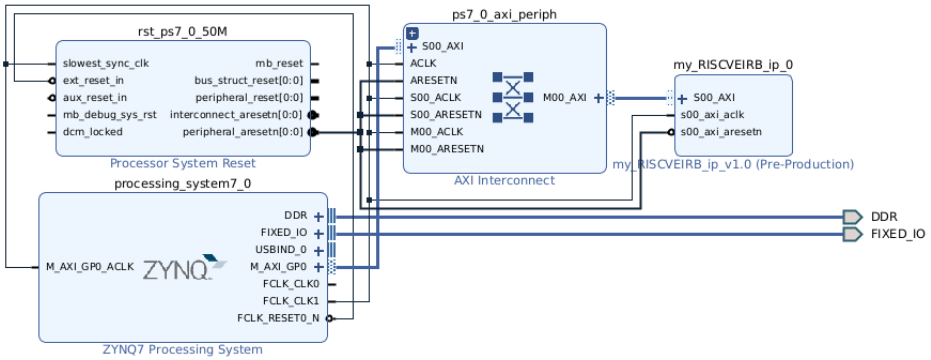

FIGURE 3 – Block Design complet

On crée ensuite le wrapper de notre design en selectionnant celui-ci (.bd) dans la fenètre des sources. On effectue un clic droit et on sélectionne Create HDL Wrapper. Une fois que le wrapper apparait, on peut générer le bitstream en laissant les options par défaut.

On vient finalement exporter le bitstream en un .xsa qui va permettre de programmer le FPGA et de faire les tests sur Vitis. On sélectionne dans les menus de la barre du haut File/Export/Export Hardware en prennant bien garde inclure le Bitstream.

4 Vitis

La première chose à faire avant de programmer la carte est de s'assurer qu'on va pouvoir afficher les messages que celle-ci va nous communiquer. On utilise pour cela Putty qu'on configure en Serial Connection type avec une Serial line /dev/ttyUSB1 et une vitesse à 115200. Les drivers USB doivent être installés pour cela. C'est dans ce terminal qu'on va lire les résultats des tests.

FIGURE 4 – Terminal Putty

On peut passer sur Vitis et créer un nouveau projet. Pour ce projet, on choisit une

nouvelle plateforme, on sélectionne notre fichier XSA et on choisit Hello World comme template, ce qui va générer un fichier C helloworld que l'on va pouvoir modifier avec nos codes tests.

Dans la barre du haut, en dessous du menu principal, on clique sur Launch Target Connection, on sélectionne Local [default] dans Hardware Server et on confirme.

On peut finalement exécuter l'affichage d'un hello world dans Putty afin de tester notre configuration. On clique droit sur le nom de notre dossier système et on sélectionne Build Project.

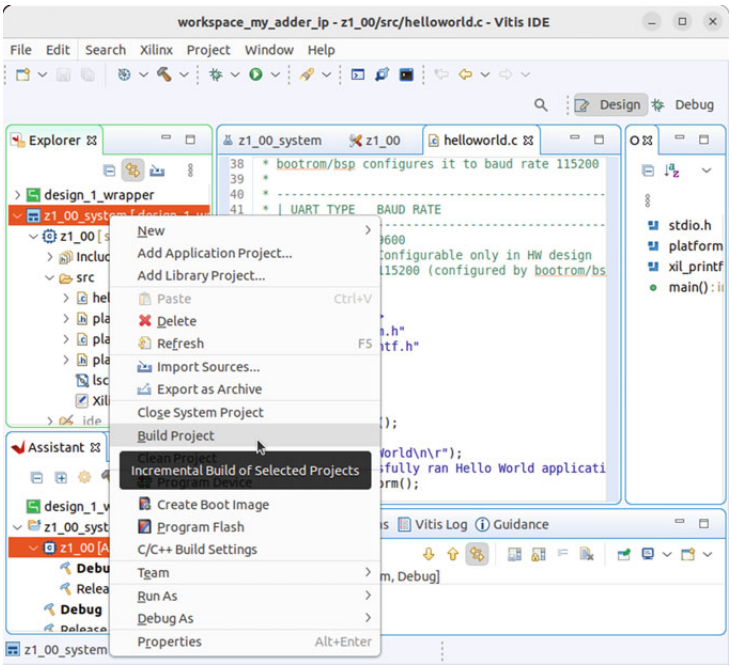

FIGURE 5 – Build Project

Une fois le Build fini, on clique de nouveau et on sélectionne Run As/1 Launch Hardware et on peut lire Hello World ainsi que Successfully ran Hello World application dans le terminal Putty. Notre FPGA est maintenant bien configuré pour effectuer des tests, on a plus qu'à modifier le fichier C, recompiler et exécuter.

5 Simulations

Afin d'avoir une référence de comportement du RISCV, il est pratique de pouvoir simuler le comportement attendu du processeur afin de pouvoir voir étape par étape les actions que le CPU est supposé exécuter, et dans quel état il est supposé être à la fin de son exécution afin de pouvoir les comparer avec le comportement que nous obtenons effectivement. Pour cela nous avons utilisé deux outils, SPIKE et le simulateur BRISC-V.

5.1 SPIKE

Spike est le simulateur RISC-V qui nous est conseillé par Bernard GOOSENS dans son livre qui nous sert de référence. Il s'agit d'une solution fonctionnant en local, qui a été installé à la demande de Monsieur Jégo sur les machines. A l'aide de l'option '-d' nous

pouvons passer spike en mode débogage et effectuer les actions du RISC-V les unes après les autres.

Cependant, au moment de rédiger ce compte-rendu SPIKE n'était plus accessible sur les machines, nous ne pouvons donc vous fournir de consignes détaillées dans son utilisation et vous renvoyons vers l'utilisation décrite dans le livre de Bernard GOOSENS. Cette solution est intéressante principalement si vous utilisez un ordinateur personnel avec droits root et que vous n'avez pas internet.

5.2 BRISCV-Simulator

BRISC-V Simulator est un projet du laboratoire ASCS qui fournit notamment un simulateur de RISCV. Il est utilisable en ligne à l'adresse : "https ://ascslab.org/research/briscv/simulator
L'interface se présente comme suit :

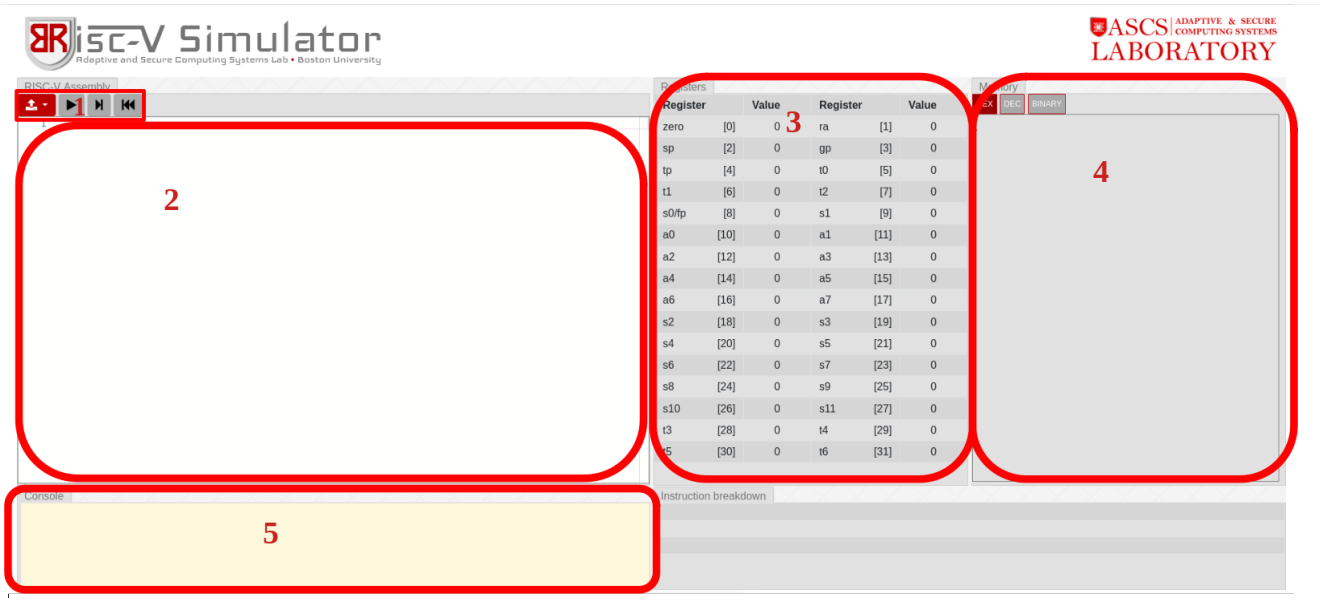

FIGURE 6 – BRISCV Interface

Avec dans la zone 1 les boutons pour uploader un fichier *.s, lancer la simulation ou exécuter une instruction, la zone 2 le code uploadé, la zone 3 les registres, la zone 4 la mémoire et dans la zone 5 la console qui permet d'avoir un retour s'il y a un problème d'upload du code assembleur.

De plus, Il peut être nécessaire de modifier le code assembleur avant de pouvoir le simuler. Cependant, le simulateur ne permet pas de modifier le code qui lui est fourni. Il faut donc bien le vérifier avant l'upload.

Tout d'abord, quelques fois le simulateur ne fonctionne pas s'il reste des commentaires dans les code. Exemple :

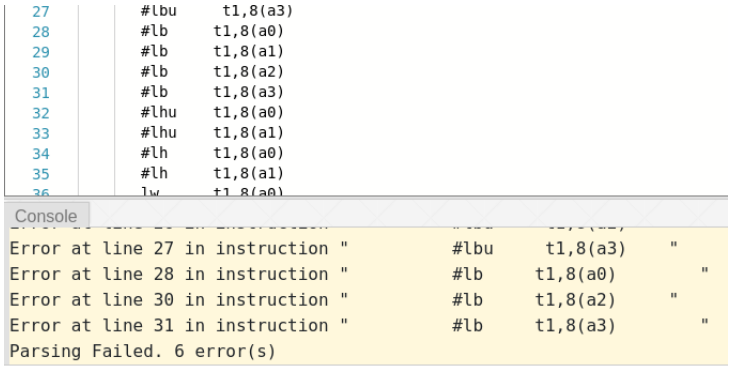

FIGURE 7 – Erreur de l'upload du code assembleur

Ensuite, contrairement à notre architecture qui sépare sa mémoire de donnée et sa mémoire d'instruction, le BRISCV ne possède qu'une mémoire et écrit ses instructions aux adresses basses de sa mémoire. Ainsi, lorsque nous réalisons une écriture en mémoire il faut prendre cela en compte afin de ne pas aller effacer des instructions dans la mémoire. Afin d'éviter ce problème, nous avons écrit nos données avec un décalage de 200 adresses (nous écrivons donc la première donnée à l'adresse 200 etc., vu que nous ne testions pas de long programmes, nous étions ainsi sûrs de ne pas effacer nos instructions, ce qui est problématique en particulier dans la situation où nous effectuons des sauts en arrières).

Une fois les modifications nécessaires effectuées, nous pouvons réaliser l'upload du fichier *.s en cliquant sur l'icone suivant :

FIGURE 8 – Boutton d'upload

Si la console affiche le message suivant c'est que le format du fichier est bien reconnu :

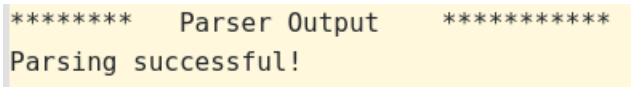

FIGURE 9 – Message d'upload correctement effectué

Le simulateur ajoute automatiquement un entête d'instructions d'initialisation qui permettent au programme entre autres d'initialiser son pointeur de pile.

FIGURE 10 – Exemple d'entête ajouté par le simulateur

Vous pouvez alors exécuter les instructions les unes après les autres en cliquant sur le bouton suivant :

FIGURE 11 – Bouton d'exécution d'une instruction

Ou bien exécuter 10 000 000 d'instructions d'un coup, ce qui nous assure d'arriver à la fin de notre programme (s'il y a une fonction d'arrêt de prévu qui réalise un saut en boucle par exemple) en cliquant sur ce bouton :

FIGURE 12 – Bouton d'exécution du code

Nous pouvons enfin vérifier les états de nos registres et de notre mémoire afin de les comparer à ce que nous obtenons effectivement avec notre processeur.

6 Les tests

L'organisation RISC-V fournit des tests officiels que chaque architecture formelle doit pouvoir passer. Ces tests permettent de tester de façon exhaustive et complète chaque instruction implémentée. Cependant, chaque test spécifique à une instruction utilise d'autres instructions. Ainsi, un processeur bogué peut ne pas passer un test spécifique sans qu'il n'y ait vraiment un problème relatif à l'instruction associée au test elle-même. Pour remédier à ce problème, il faut commencer par tester l'architecture d'une manière non exhaustive, mais assez efficace pour déboguer le processeur et résoudre les problèmes principaux liés à l'implémentation de certaines instructions. Les tests primordiaux sont listés dans le tableau ci-dessous :

Le test	Les instructions testées
test_op_imm	Les pseudo-instructions li et ret - les instructions addi, andi, ori, xori, sltiu, slli, slti, srli et srai
test_op	Les pseudo-instructions li et ret - les instructions add, and, or, xor, sub, sltu, sll, slt, srl et sra
test_load_store	Les pseudo-instructions li et ret - les instructions sw, addi, sh, sb, lb, lbu, lh, lhu et lw
test_lui_auipc	Les pseudo-instructions li,jr et ret - les instructions lui, auipc, sub, et addi
mytest_jal_jalr	Les pseudo-instructions li,jr, j, mv et ret - les instructions sw, auipc, jal, et jalr
test_branch	Les pseudo-instructions liet ret - les instructions beq, bne, blt, bge, bltu et bgeu
myPGCD_RISCV	Les pseudo-instructions li,j, mv et ret - les instructions lw, sub, bge, sw et bne

Pour pouvoir effectuer ces tests, il suffit d'inclure le fichier hexadécimal associé au test dans le fichier helloworld.c fourni, puis d'exécuter le code C. Ce code C inclut les en-têtes associés à notre IP personnalisée. Il déclare deux tableaux jouant respectivement le rôle de la mémoire d'instructions et la mémoire de données. Des boucles sont utilisées pour écrire les instructions et les données dans la mémoire de l'IP RISC-V personnalisée. Une autre boucle assure l'exécution du programme, puis des boucles assurent la lecture des instructions et des données depuis la mémoire de l'IP personnalisée. Ensuite, ces octets lus sont affichés. Il faut donc s'assurer des résultats du test en comparant ces résultats avec les résultats d'un simulateur (Voir partie Simulateur) ou avec les résultats trouvés après le déroulement manuel du programme.

Une fois ces tests passés, il est possible de tester le processeur personnalisé avec les tests officiels. Ces tests permettent de tester de manière exhaustive chaque instruction. Chaque test associé à une instruction spécifique est lui-même composé de plusieurs sous-tests. Le programme helloworld.c a la même structure que le précédent, à la différence qu'il affiche pour chaque instruction si tous les tests associés ont été passés et, le cas échéant, le numéro du premier test que le processeur n'a pas pu passer. En tout, il existe 51 tests : 50 tests pour les instructions et un test "Simple" qui ne teste aucune instruction, mais teste l'architecture elle-même. Il s'agit du test d'auto-vérification le plus basique. Si le processeur personnalisé ne réussit pas ce test, il y a peu de chances qu'il réussisse l'un des tests d'auto-vérification plus complexes. Ces tests ont été inclus au programme C sous la forme d'un fichier hexadécimal contenant toutes les instructions de l'ensemble des tests et des sous-tests les formant. Une fois un problème détecté, il faut corriger l'architecture en corrigeant les codes RTL relatifs à l'implémentation de l'instruction concernée par le problème détecté.

Lors des tests primordiaux, nous avons remarqué qu'à la suite de l'utilisation des pseudo-instructions j et jr, on a observé un comportement anormal de nos programmes. Plus précisément, les différences entre les résultats attendus et ceux observés se font lorsqu'une instruction utilise le registre X0 comme entrée. Ce registre est utilisé dans de nombreuse instruction afin de récupérer un zéro, car normalement ce registre est constamment fixé à zéro. De ces observations, on a alors pu déduire où se trouvait le problème.

En effet, les pseudo-instructions sont des dérivées des instructions jal et jalr. Ces instructions vont changer la valeur du pc, c'est-à-dire faire un saut dans le programme, tout en stockant la valeur actuelle du pc dans un registre, c'est-à-dire conserver l'emplacement du programme avant le saut. Pour les pseudo-instructions j et jr, on ne va pas sauvegarder la valeur du pc. Pour faire cela, on définit ces pseudo-instructions comme étant les instructions jal et jalr avec comme registre dans lequel sera stockée la valeur actuelle du pc le registre X0. Comme on l'a déjà dit, ce registre est normalement fixé à zéro et sert donc ici de "poubelle" car la valeur ne devrait pas être sauvegardée. Or, après plusieurs tests plus centrés sur ces instructions et sur ce problème, on a pu déterminer que cela venait justement du fait que le registre X0 pouvait être modifié, il n'était pas fixé et ne servait pas de "poubelle", ce qui est contraire aux spécifications RISC-V.

Après la correction de ce problème et d'autres problèmes principalement liés à l'implémentation des instructions de décalage, notre architecture a pu passer l'ensemble des tests officiels.

```
Data Memory Read Process
addi  : all tests passed
add   : all tests passed
andi  : all tests passed
and   : all tests passed
auipc : all tests passed
beq   : all tests passed
bge   : all tests passed
bgeu  : all tests passed
blt   : all tests passed
bltu  : all tests passed
bne   : all tests passed
jalr  : all tests passed
jal   : all tests passed
lb    : all tests passed
lbu   : all tests passed
lh    : all tests passed
lhu   : all tests passed
lui   : all tests passed
lw    : all tests passed
ori   : all tests passed
or    : all tests passed
sb    : all tests passed
sh    : all tests passed
simple: all tests passed
slli  : all tests passed
sll   : all tests passed
slti  : all tests passed
sltiu : all tests passed
slt   : all tests passed
sltu  : all tests passed
srai  : all tests passed
sra   : all tests passed
srli  : all tests passed
srl   : all tests passed
sub   : all tests passed
sw    : all tests passed
xori  : all tests passed
xor   : all tests passed

Successfully ran RISCV Program
```

FIGURE 13 – L'ensemble des tests officiels passés

7 Conclusion

Comme nous l'avons vu, cette méthode nous a permis de corriger plusieurs bugs et ça mise en place permettra dans le futur de modifier beaucoup plus facilement le processeur, car il n'y aura plus qu'à relancer une phase de tests après chaque modification pour s'assurer qu'il est toujours opérationnel. Notre IP RISC-V personnalisé a finalement réussi avec succès tous les tests officiels fournis par l'organisation RISC-V. Il est désormais prêt à être utilisé pour implémenter des programmes et des fonctionnalités plus complexes. Cela témoigne de la fiabilité et de la conformité de notre implémentation aux spécifications RISC-V, ouvrant ainsi la voie à des applications plus avancées et à des développements futurs.

8 Bibliographie

Guide to computer processor architecture a risc-v approach with high-level synthesis, Bernard Goossens