COMPTE RENDU FINAL

PROJET RISC-V

1 TABLE DES MATIERES

2 INTRODUCTION

2.1 PRESENTATION

Les processeurs RISC « *Reduced Instruction Set Computing* » sont aujourd'hui l'un des types de processeur le plus utilisé dans le milieu de la conception numérique. Énormément d'architecture de processeurs comme ARM utilisent ce type de structure. A l'origine, les processeurs RISC se reposaient sur un jeu d'instruction simple et limité. Ce qui rend bien plus facile la conception de ce type de processeur contrairement à des processeurs CISC (*Complex Instruction Set Computing*).

Nous avons choisi de faire ce projet pour plusieurs raisons. Premièrement, nous avions à cœur de creuser les notions vues en cours d'architecture des processeurs, de ne pas s'arrêter à une compréhension globales des idées mais d'aller vraiment au bout de la démarche. Deuxièmement, le défi technique nous semblait intéressant. C'est l'occasion d'approfondir nos compétences en design numérique et de mener un projet numérique d'une envergure plus conséquente que ce que nous avons pu faire précédemment. Nous avons également développé nos compétences en gestion de projet, d'autant plus qu'il s'agit d'un projet durant lequel il était facile de s'égarer dans les détails techniques et optimisations. Il est donc nécessaire de bien se répartir les tâches et de s'organiser correctement pour être efficace.

Les différentes étapes de la conception de notre processeur sont décrites pendant ce rapport. Vous pouvez retrouver l'ensemble du projet dans le GitHub suivant :

https://github.com/romainDCHD/Miniprojet_RISC_V.git

2.2 CAHIER DES CHARGES

Initialement notre objectif était de réaliser un processeur RISC-V pipeline 5 étage capable d'exécuter en simulation un programme en C. Pour ce faire nous avons défini un certain nombre d'instructions minimales qu'il nous fallait pour faire tourner un code complet. Nous avions donc pour objectif initial de pouvoir exécuter tout le jeu d'instruction le plus simple du RISC-V mais aussi de réaliser un compilateur nous permettant de passer d'un code assembleur à un code machine (binaire) correspondant au jeu d'instruction spécifique de notre processeur. L'objectif était également de pouvoir réaliser une implémentation ASIC pour notre processeur. Ce choix était motivé par plusieurs raisons : la première et que nous avons tous les trois davantage envie de travailler sur des circuits ASIC, un peu plus proches du silicium que les circuits FPGA, et la deuxième est que réaliser les tests du processeur sur FPGA aurait été assez long à faire et nous savions qu'avec notre objectif de compilateur ça ferait beaucoup. En effet sur FPGA nous n'avions pas de moyen simple d'observer l'exécution de notre processeur étant donné que les résultats seuls se trouvaient dans une mémoire locale au circuit.

Malgré tout, nous nous sommes rendus compte un peu tard que cet objectif était un peu trop ambitieux pour le temps imparti, notamment à cause de la phase de test que nous avions sous-estimée.

2.3 JEU D'INSTRUCTION

Comme dit précédemment, le jeu d'instruction choisi est le plus basique du RISC-V, RV32I. Ce jeu d'instruction a pour objectif d'être le plus simple possible tout en supportant suffisamment de fonctionnalités afin de pouvoir supporter des systèmes d'exploitation modernes. Nous avons alors réalisé de légères modifications au jeu d'instructions, étant donné que nous n'avions absolument pas

comme ambition de supporter un système d'exploitation. Nous avons donc décidé de ne pas supporter les instructions **CSR** (*Control Status Register*), qui correspondent à l'interaction avec des registres spéciaux liés à l'exécution d'autres instructions. De même pour les instructions **ECALL** et **EBRAK** qui sont des instructions de gestion d'exécution et de *debug*. Elles ne nous auraient pas été utiles étant donné que nous avions déjà tout l'environnement de *Simulink* afin d'effectuer le debug. Enfin les dernières instructions à ne pas supporter sont **FENCE** et **E.FENCE** qui sont utile dans le cas où l'on veuille sécuriser des zones mémoires.

3 COMPILATEUR

Dans notre démarche de conception nous avions prévu d'effectuer des tests assez poussés afin de pouvoir valider le fonctionnement de notre processeur. Cela devenait d'autant plus nécessaire que l'on rajoutait des optimisations au cœur. Afin de pouvoir réaliser des programmes de tests suffisamment poussés nous avions besoin d'un compilateur nous permettant de passer de code assembleur ou C à du code machine. Nous pensions initialement utiliser des compilateurs existants tel que GCC cependant nous avons dû faire face à certains problèmes tel que le fait que notre processeur supporte un nombre d'instructions très limité. Nous avons alors pensé au fait que nous avions déjà réalisé un compilateur en début d'année et qu'il suffisait de modifier une partie du projet déjà existant pour l'adapter à nos nouveaux besoins.

Nous avons donc récupéré le projet de compilateur assembleur Python et fait certaines modifications. Nous avons dans un premier temps adapté le fichier d'expressions régulières à la syntaxe de l'assembleur Intel. Nous avons ensuite réécrit l'intégralité du parser afin de, là encore, l'adapter à la syntaxe de notre assembleur, et aussi pour pouvoir stocker les informations des lignes assembleur. Il a aussi été nécessaire de créer la structure et l'ensemble des fonctions liées à ces dernières. Cela comprenait le remplacement de certaines instructions (les détails ici) , le calcules des offsets des sauts de branchement, l'ajout d'instructions NOP à la suite d'un branchement afin d'éviter de remplir le pipeline d'instructions ne devant pas être exécuter, et enfin le rebouclage du code sur lui-même. Le compilateur nous permettait enfin de générer un fichier ASCII contenant les séquences de bits de nos instructions.

```
main:
        addi    sp,sp,-32
        sw      r6,28(sp)
        addi    r6,sp,32
        lui      r3,2
        sw      r3,-28(r6)
        lui      r3,1
        sw      r3,-20(r6)
        lw      r2,-28(r6)
        lw      r3,-20(r6)
        beq     r2,r3,.L2
        sw      zero,-24(r6)
        jal     r1,.L3
.L4:
        lw      r3,-20(r6)
        addi    r3,r3,1
        sw      r3,-20(r6)
        lw      r3,-24(r6)
        addi    r3,r3,1
        sw      r3,-24(r6)
.L3:
        lw      r2,-24(r6)
        lui      r3,4
        blt     r2,r3,.L4
.L2:
        lw      r3,-20(r6)
        lw      r6,28(sp)
        addi    sp,sp,32
        jalr    ra,r2,main

;https://godbolt.org/
;int main(void) {
;    int a = 2;
;    int b= 1;
;
;    if (a != b) {
;        for (int i = 0; i < 5; i=i+1) {
;            b = b + 1;
;        }
;    }
;
;    return b;
;}
```

```
11111110000000010000000100010011
00000000011000010010111000100011
00000010000000010000001100010011
00000000000000000000000000110111
11111110001100110010001000100011
00000000000000000000000000110111
11111110001100110010011000100011
11111110010000110010000100000011
11111110110000110010000110000011
00000100001100010000000001100011
00000000000000000000000000000000
00000000000000000000000000000000
11111110000000110010010000100011
00000001110000000000000011101111
11111110110000110010000110000011
00000000000010001100000000110010011
11111110001100110010011000100011
11111110100000110010000110000011
00000000000010001100000000110010011
11111110001100110010010000100011
11111110100000110010000100000011
00000000000000000000000000110110111
11111110001100010100000011100011
00000000000000000000000000000000000
00000000000000000000000000000000000
11111110110000110010000110000011
00000001110000001001000110000011
00000010000000010000000100010011
11111001000000010000000011100111
01111000110101111111001101101111
```

Pour passer du code C au code assembleur alimentant notre compilateur nous utilisions le site Compiler Explorer avec un compilateur GCC RV32. Une fois toutes les étapes terminées nous chargions le fichier du code machine dans la mémoire instructions du processeur depuis notre bench.

4 LE PROCESSEUR

4.1 LES RECHERCHES

Pour commencer à concevoir notre processeur, nous avons dû commencer par des recherches pour voir ce qu'il se faisait et les architecture que l'on pouvait trouver. Notre méthode était la suivante :

1. Rechercher un maximum d'architecture pipeline 5 étages sans se poser plus de question et les répertorier sur un document.
2. Dans un second temps nous avons comparé les architectures pour comprendre les modules qui revenaient tout le temps. Nous avons ensuite sélectionné une architecture qui nous paraissait la plus judicieuse
3. Nous avons fait tourner des instructions dans l'architecture trouvée en ne nous basant que sur le schéma global de l'architecture trouvée et les connaissances acquises en cours pour comprendre dans les détails son fonctionnement, la modifier et l'adapter pour établir notre propre architecture.

Note : Nous nous sommes un moment égaré à vouloir trop rentrer dans les détails des architectures déjà existantes et bien détaillées. Nous nous sommes rendu compte que ça n'était pas judicieux et qu'il fallait auparavant qu'on acquiert une compréhension globale du sujet avant de rentrer dans les détails des codes. Nous n'avons finalement jamais relu ces codes par la suite.

Voici un petit exemple d'instruction qu'on a pu faire tourner à la main :

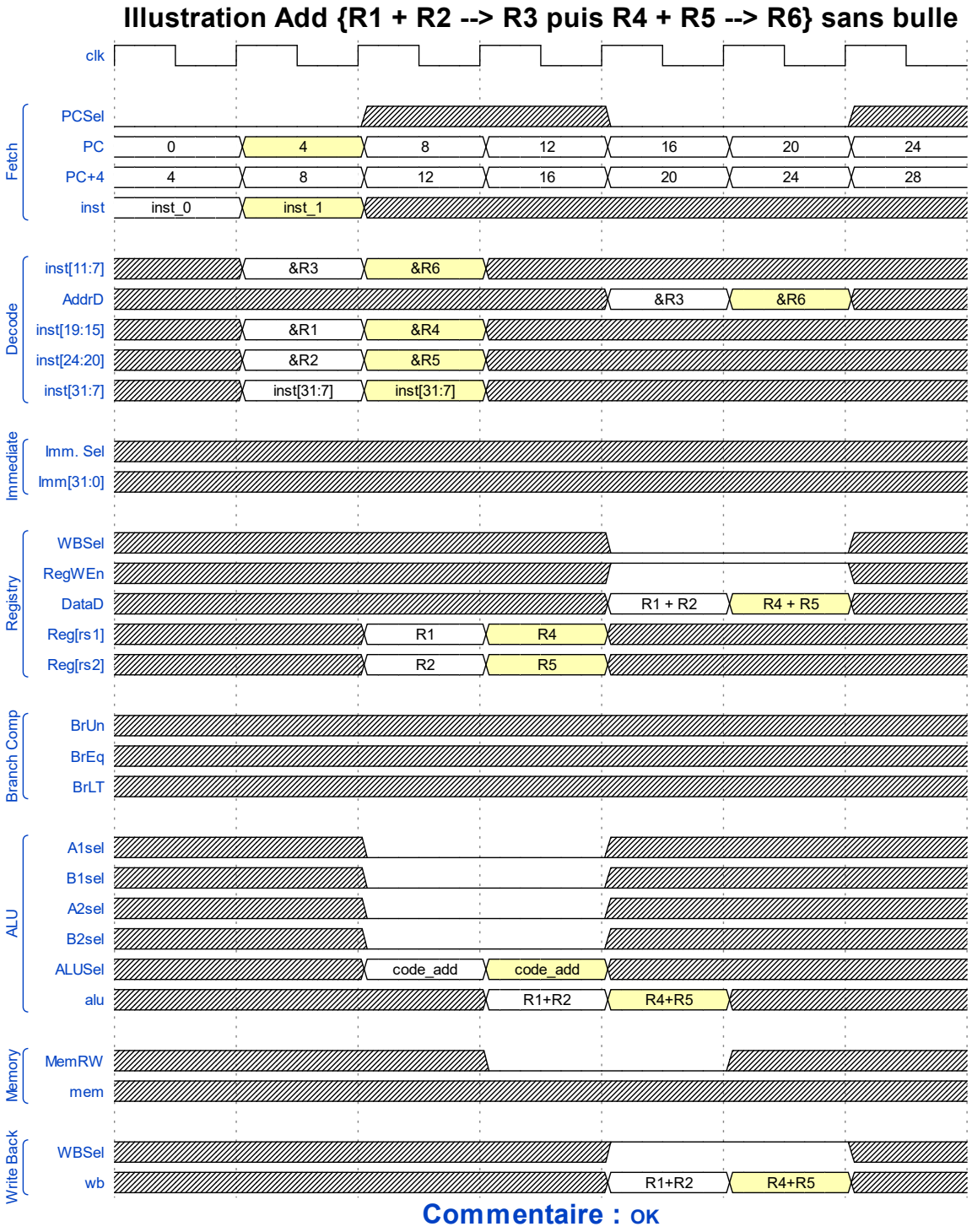

Figure 4.1 - Chronogramme d'une suite d'instruction ADD sans bulle

Cette étape fut cruciale pour nous car elle nous a permis de bien comprendre comment marche un processeur et de voir les dépendances de données que nous pouvions avoir.

Les deux points gênants que nous avons identifiés sont les suivants : les bulles dues aux dépendances de données entre les instructions et les instructions exécutées alors que nous sommes censés prendre un branchement ou un jump. Le premier problème est réglé directement en hardware par notre processeur quant au deuxième problème nous avons décidé de le régler en software (cf. partie sur le COMPILATEUR).

4.2 L'ARCHITECTURE

Comme nous l'avons dit précédemment nous avons fait un pipeline 5 étages. Voici la répartition des différents étages :

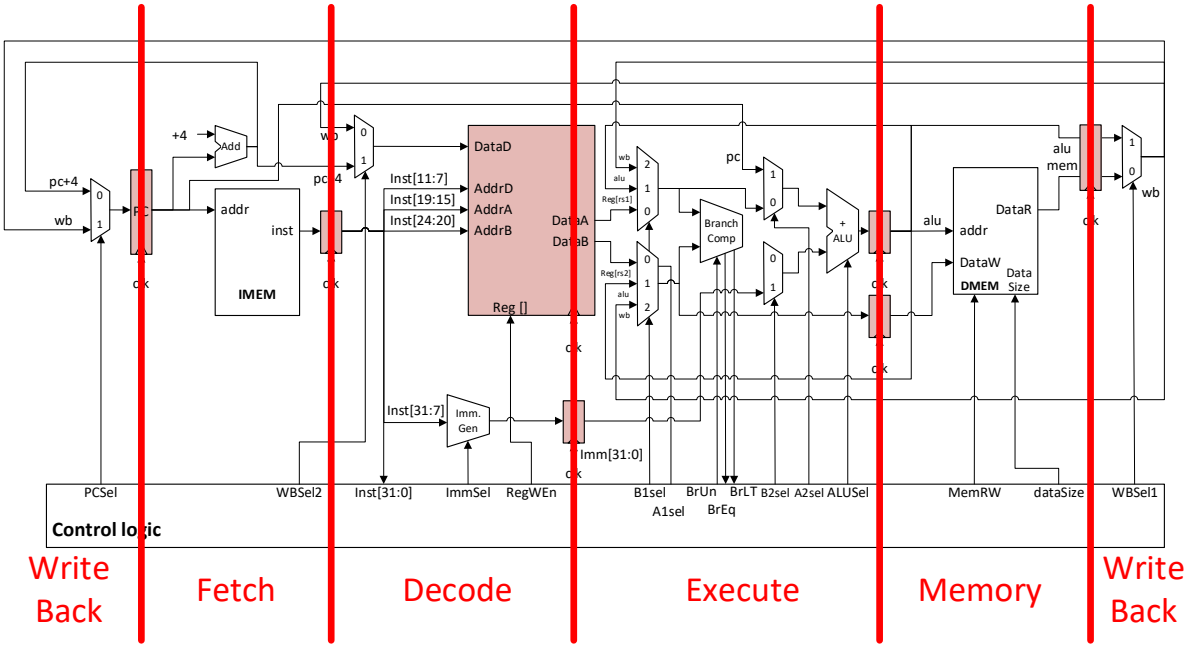

Figure 4.2 - Architecture du cœur avec les différents étages

Pour ce qui est de la fonctionnalité des différents blocs nous n'allons pas rentrer dans les détails car nous pourrions passer des heures à expliquer le fonctionnement de chacun d'eux. Néanmoins dans les grandes lignes :

- Le bloc **IMEM** est la mémoire instruction chargée directement par notre testbench
- Le bloc **Reg[]** dans l'étage de décodage est le fichier de registre constitué de 32 registres de 32 bits décrit CI-DESSOUS.
- L'étage **Execute** est l'étage qui va permettre de faire des optimisations hardware via les multiplexer récupérant les données soit à la sortie de l'étage *Write Back* pour gagner 1 bulle, soit de l'ALU pour gagner en gagner deux.
  Note : nous n'avons ici pas mis le multiplexage qu'il faudrait avoir après la mémoire pour la aussi pouvoir gagner des bulles.
- **L'ALU** va réaliser les opérations arithmétiques et logiques sur les valeurs qu'il a en entrée. Il ne prend en compte que les calculs sur entier et ne fait pas de multiplication ou division.
- L'étage de mémoire permet de prendre en charge les instructions LOAD ou STORE et comprend la mémoire de données.
- Enfin **l'étage Write Back** qui permet de faire boucler le processeur.

Tableau 1 - Description des registres du RV32I, extrait de la spécification RISC-V

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Vous pouvez retrouver le code RTL de chacun de nos modules directement sur le git.

5 LES TESTS

5.1 TESTS DES BLOCS INDIVIDUELLES

Chaque bloc composant notre processeur étaient, pour la plupart dans leur première version, tous testé individuellement pour vérifier que le fonctionnement attendu du bloc seul était respecté. Cela nous permettait rapidement de modifier ou faire une première étape de débogage sur nos codes. De plus, un testbench permet aussi de donner une meilleure compréhension aux autres membres de l'équipe sur la façon de manipuler et utiliser le bloc.

Voici un exemple simple du test du bloc **imem**, le bloc mémoire dans lequel étaient stockées les instructions :

```
initial forever begin
    imem1.tab_inst[0] = 32'haaaaaaaa;
    imem1.tab_inst[1] = 32'hbbbbbbbb;
    imem1.tab_inst[2] = 32'hcccccccc;
    imem1.tab_inst[3] = 32'hdddddddd;
    addr = 32'h00000000;
    #100
    addr = 32'h00000004;
    #100
    addr = 32'h00000008;
    #100
    addr = 32'h0000000c;
    #100;
end
```

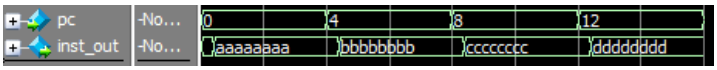

Figure 5.1 - Test et résultat du bloc imem

Le but du bloc imem est de renvoyer l'instruction correspondant au PC (*program counter*) rentrant. Sachant que le PC s'incrémente toujours de 4 en 4 pour passer d'une instruction à la suivante, le PC 0 correspond à l'adresse 0 du tableau d'instruction, le PC 4 correspond à l'adresse 1 et ainsi de suite. Ainsi on voit sur la simulation de ce testbench, effectuée sur *ModelSim*, que pour chaque PC sort l'instruction stockée à la bonne adresse appelée par PC.

5.2 Tests « a la main »

Pour réaliser les tests nous avons procédés en parallèle de deux manières : La première était de réaliser des tests à la main en entrant directement les mots binaires dans la mémoire instruction et en vérifiant à la main la sortie. Pour chaque test, une petite description du test ainsi que son avancement était rempli dans un fichier prévu à cet effet.

Voici un exemple de test mis en place :

1. ASM_ADD_PB.BIN

`but`: tester add avec dep. de données: **dépendance d'un cran**

- effectuer un load immédiat de 2 dans le registre 0
- LUI de 4 dans le reg 1
- faire un add de R0 et R1 et le stocker dans R2
- faire un add de R0 et R2 et le stocker dans R3 **dépendance de 2 crans**
- LUI de 2 dans le registre 3
- faire un add de R0 et R3 et le stocker dans R4
- faire un add de R0 et R4 et le stocker dans R5
- faire un add de R0 et R5 et le stocker dans R6

`avancement`: réussi

Figure 5.2 - Description d'un programme de test sur des instruction ADD avec dépendance (source)

La deuxième façon que nous avions de tester notre code était d'utiliser le compilateur et une vérification automatique des données mémoire et des registres comme décrit ci-dessous.

5.3 BANC DE TEST GLOBAL

Afin de pouvoir valider avec confiance le fonctionnement de notre processeur nous avons réalisé un banc de test global avec l'aide du compilateur décrit CI-DESSUS. Ce banc de test avait pour but d'exécuter un programme C permettant de tester toutes les instructions supportées dans le maximum de contextes. Cela regroupait des cas de dépendances de données, des débordements, ou encore tous les résultats possibles d'une opération dans les cas les plus simples comme pour les comparaisons. Les résultats étaient ensuite automatiquement vérifiés dans le banc de test.

Les premières difficultés que nous avons rencontrées lors de l'exécution de ce banc de test étaient d'abord dans l'exécution des instructions préliminaires permettant de préparer les données pour le banc de tests. Nous avons alors pu débugger la lecture et l'écriture dans les registres, puis dans la mémoire. Au moment où nous écrivons ce compte rendu nous avons **36%** des tests qui fonctionnent. Nous savons qu'un autre problème auquel nous devons faire face est un cas de dépendance que nous n'avons pas pris en compte.

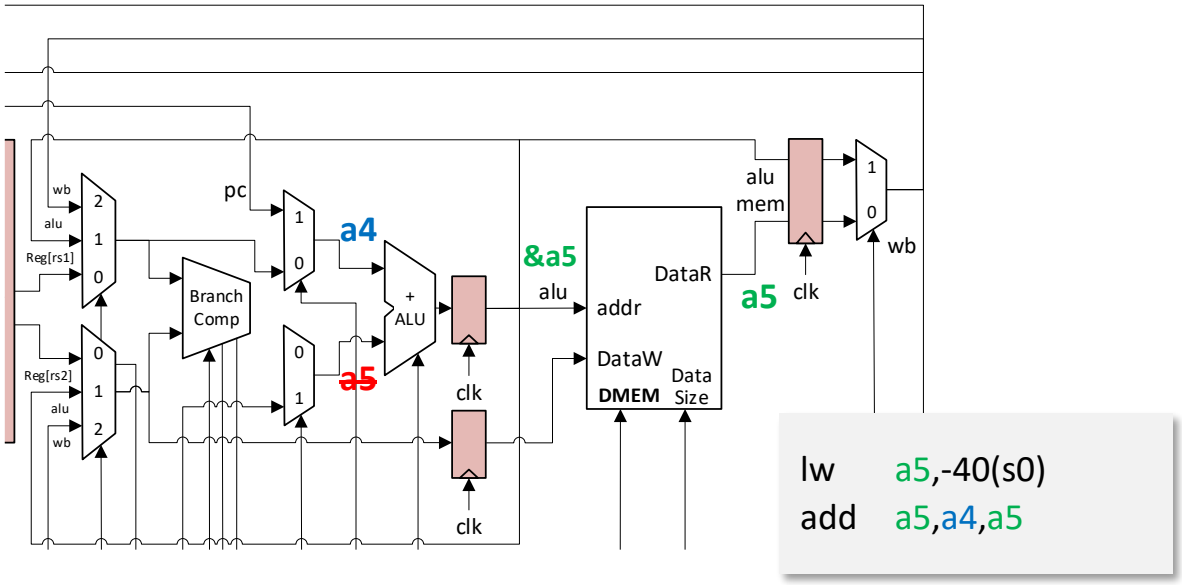

Figure 5.3 - Illustration du problème de dépendance sur la mémoire

Il peut y avoir des cas où une donnée lue en mémoire (**a5**) est requise dans l'étage d'exécution au cycle suivant et se retrouve alors manquante (~~**a5**~~). Il nous faudrait alors rajouter une entrée aux multiplexer afin de pouvoir récupérer cette donnée.

6 SYNTHESE ET PLACEMENT ROUTAGE

6.1 SYNTHESE

Une Synthèse est une étape importante dans le flot de conception. Elle permet de transformer et traduire un code Verilog, SystemVerilog ou VHDL en une *netlist*. Une *netlist* est une liste des connexions électriques entre les différents modules et composants électroniques.

Nous avons eu relativement peu de temps pour effectuer la synthèse car nous nous sommes rendu compte un peu tard qu'il serait impossible de tester le processeur entièrement. Par ailleurs, pour

pouvoir effectuer des tests facilement nous avons dû opter pour des choix hardware qui n'étaient pas synthétisables. Le gros point bloquant était la mémoire instruction. En effet, comme nous n'avons pas recréé un OS en entier, nous avons donc décidé de faire une mémoire instruction de taille variable (le point bloquant) pour pouvoir, en fonction des tests, y accueillir exactement les instructions souhaitées. La dernière instruction que l'on chargeait depuis le testbench général était une instruction de branchement inconditionnel qui revenait au début des instructions, ce qui est le choix fait par les OS quand ils n'ont plus d'instructions à envoyer. Notre programme pouvait donc tourner à l'infini. Nous avons donc simplement retiré la mémoire instruction pour la synthèse, ce qui n'est pas si gênant que ça en fin de compte car l'objectif ici n'était pas de finir avec un processeur fonctionnel mais de comprendre comment marchait la synthèse.

La synthèse s'est effectuée sur le logiciel *Design Vision*. Elle nous a permis de tirer plusieurs caractéristiques et performances possibles de notre microprocesseur :

Tableau 2 - Spécification du processeur à l'issue de la synthèse

Critère	Chemin critique	Fréquence maximale	Puissance consommée	Surface occupée
Performances	7 ns	143 MHz	120 mW	1,9 mm²

On possède, de plus, une vision en circuit logique (portes NAND, OR…) du chemin critique. On peut alors analyser ce chemin et faire en sorte de réduire le nombre de portes et donc la longueur du chemin. Il nous permet aussi d'isoler dans le code, la partie à optimiser et ainsi rendre notre processeur plus efficace, ce que nous aurions fait avec plus de temps, avec une réduction du temps de chemin critique et donc une augmentation de la fréquence de fonctionnement.

6.2 PLACEMENT ROUTAGE SUR CIBLE ASIC

Le placement et routage sont les deux étapes qui suivent la synthèse dans le flot de conception de notre micro-processeur. Elles sont réalisées avec l'outil Innovus de Cadence et sont-elles-même constituées de plusieurs étapes.

- Avant de commencer le placement, il faut d'abord importer les bibliothèques (fichiers LEF) et les librairies (.lib) contenant les informations de timing pour chaque bibliothèque
- Ensuite il a été nécessaire d'écrire deux fichiers :
  - **.v** : netlist Verilog qui instancie le top et les plots d'entrée/sortie
  - **.io** : l'agencement de la couronne et le positionnement des plots est défini dans ce fichier
- Importation du design donc de la netlist créée précédemment lors de la synthèse et de l'autre créée lors de l'étape précédente.
- Viennent maintenant les étapes de placement : représente les étapes de positionnement des entrées et sorties, des différents blocs du circuit en fonction de la taille voulue pour la puce et autres contraintes. On procède alors à la création du plan des surfaces, placement des entrées/sorties et placement des cellules :

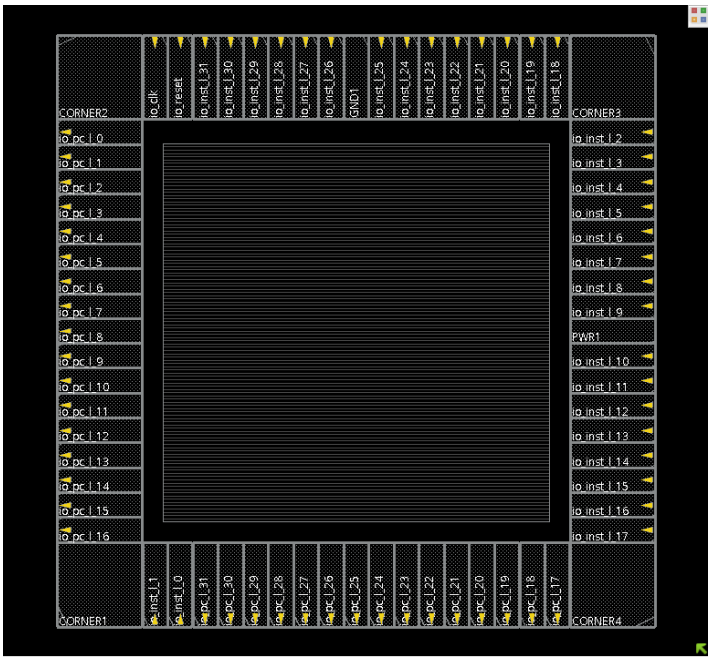

Figure 6.1 - Vue après la réalisation les étapes de placement

- Viennent enfin les étapes de routages :
  - Routages des alimentations : couronnes et rails d'alimentation et bandes placées au-dessus du cœur :

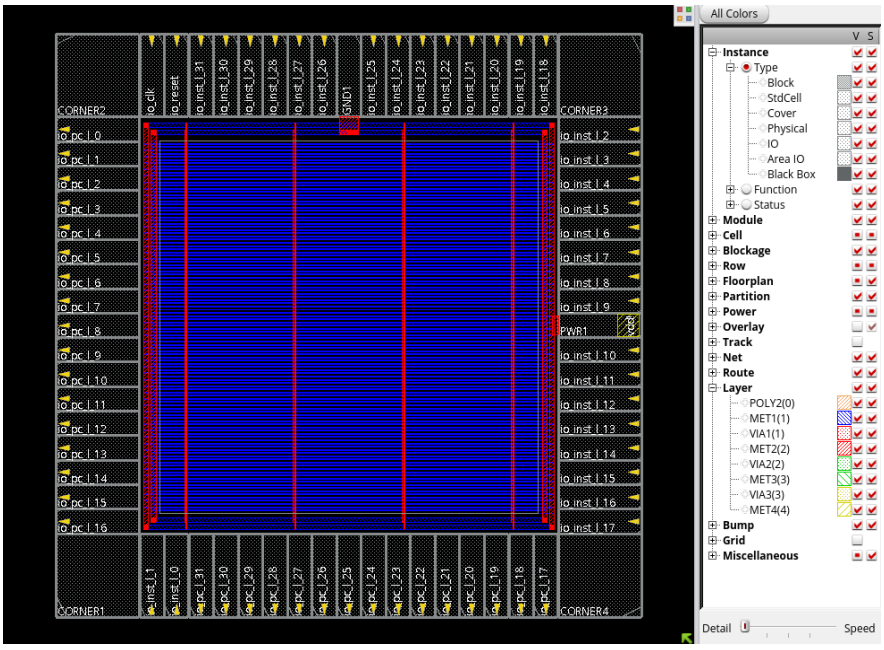

Figure 6.2 - Vue après la réalisation les étapes de routage alimentation

  - Génération de l'arbre d'horloge et routage de l'horloge : il y a malheureusement eu un petit problème avec Innovus pour cette partie.
  - Routage final (avec remplissage des espaces libres) et vérification :

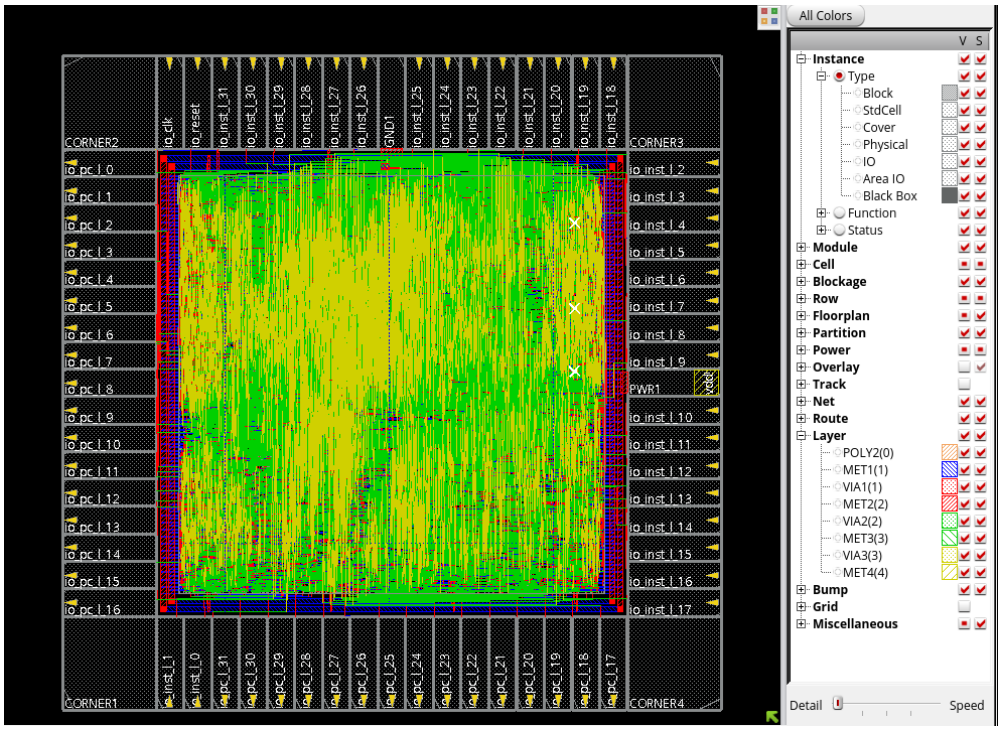

Figure 6.3 - Vue après le routage final et la vérification

- La dernière étape qui va permettre de passer à la prochaine étape du flot est l'exportation des fichiers :
  - DEF : Ce fichier permet d'importer le design du circuit dans CADENCE
  - GDSII : Ce fichier est une vue physique du circuit. Il sera importé dans cadence pour les vérifications DRC et LVS
  - Nouvelle Netlist Verilog car des buffers ont été ajoutés lors de la génération de l'arbre d'horloge (même si ici elle n'a pas marché)

7 AVANCEMENT ET GESTION DE PROJET

7.1 AVANCEMENT

Si nous devions faire un bilan de la répartition de notre temps nous dirions que nous avons passés ¼ de notre temps à rechercher une architecture, un jeu d'instruction et établir un cahier des charges, ¼ à réaliser concrètement notre processeur en System Verilog et la moitié de notre temps à réaliser des tests de fonctionnement.

Le plus long aura été d'organiser notre environnement de test pour charger correctement la mémoire, générer automatiquement des instructions binaires et vérifier automatiquement si les tests sont passés ou non. Après cette étape, débugger le processeur fut plutôt compliqué de par sa complexité globale mais néanmoins assez rapide car pour chaque problème réglé, plusieurs tests pouvaient passer. Nous avons vraiment la sensation que notre processeur est sur le point de marcher *parfaitement*, il nous manque seulement un peu de temps pour finir de débugger certaines instructions. Finalement, voilà tous les tests qui sont passés, le plus exhaustivement possible :

- Ecriture de variable de tailles différentes en mémoire
- SLL (Shift Left Logical)

- SRL (Shift Right Logical)
- SLTU vrai (Set Less Than Unsigned)
- SLTU faux
- SLT vrai (Set Less Than)
- SLT faux
- SUB sans débordement
- SUB avec débordement
- SLTIU vrai (Set Less Than Immediate Unsigned)
- SLTIU faux
- SLTI vrai (Set Less Than Immediate)
- SLTI faux

Les sauts et les branchements n'ont pas eu le temps d'être testé.

Nous avons décidé d'arrêter la phase de test un peu plus tôt car nous nous sommes rendus compte qu'il était tout bonnement impossible de tester exhaustivement le processeur dans le temps imparti. Cela nous a permis de faire avec succès la synthèse du processeur, ainsi que son placement et routage. Cette étape nous a pris environ 13h.

Le compilateur quant à lui est totalement fonctionnel et satisfaisant.

7.2 GESTION DE PROJET

Pour mener à bien notre projet nous avons mis en place un certain nombre de choses. Nous avions une communication régulière via un canal de communication Messenger pour nous mettre à jour des avancées de chacun. Ainsi, pour chaque session de travail nous tenions au courant les autres membres du groupe de nos avancés, des problèmes rencontrés et des solutions trouvées. Cela nous a permis d'être toujours à jour et de pouvoir nous entraider même à distance.

Nous nous sommes aussi bien répartis les tâches au fur et à mesure. Le diagramme de Gantt à été très utile au début pour planifier le travail sur le long terme et pour qu'on essaye d'évaluer le temps que nous prendrait chaque tâche. Il nous a permis d'être serein sur l'avancement du projet car nous savions que nous étions dans les temps à l'exception de la fin où nous avions clairement sous-estimés la phase de test.

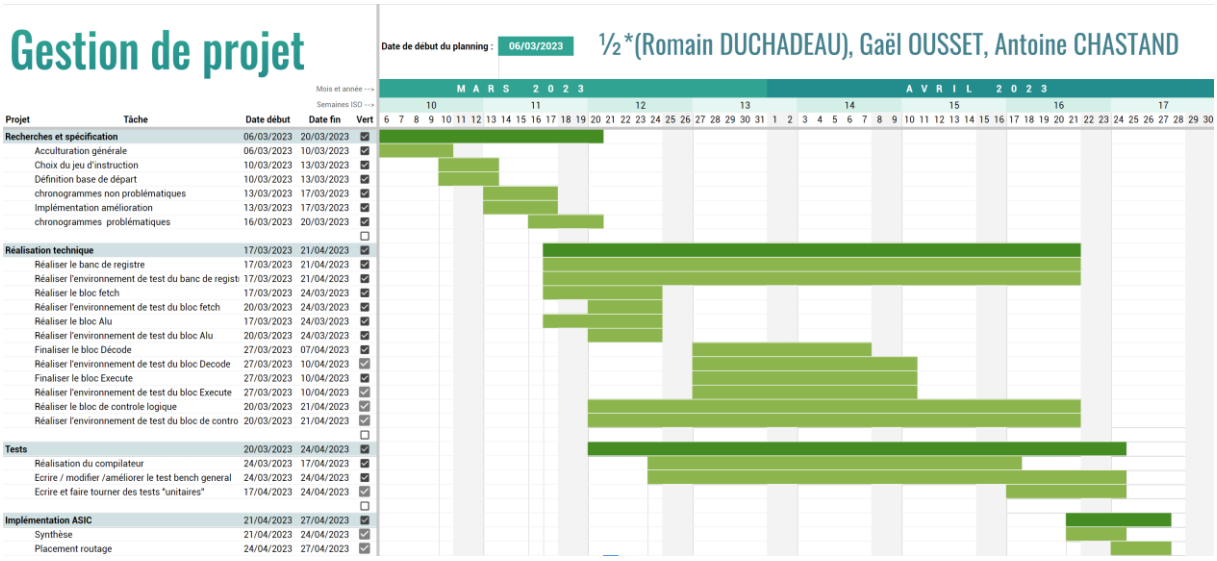

Figure 7.1 - Diagramme de Gantt

8 CONCLUSION ET BILAN DE COMPETENCES

8.1 BILAN DE COMPETENCES

8.1.1 Romain DUCHADEAU

Ce projet a été pour moi l'occasion d'apprendre énormément sur beaucoup de points et de consolider énormément de compétences acquises durant cette année à Phelma.

Sur le plan purement technique j'ai pu développer ma compétence **Concevoir ou réaliser des solutions techniques ou expérimentales, permettant de répondre à un cahier des charges** en **m'appuyant sur les connaissances acquises en cours** pour m'aider à concevoir un cahier des charges réaliste mais aussi m'aider dans la conception d'un bloc relativement complexe. J'ai beaucoup apprécié l'autonomie qu'on avait car, même si nous pouvions parfois perdre du temps par manque de connaissances / de pratique, cela nous a permis d'essayer, de se tromper et d'apprendre de ses erreurs (ce qui est bien plus profitable pour notre formation que simplement essayer quelque chose qui marche déjà). Cela m'a également permis de **savoir respecter des contraintes de temps** puisque nous étions totalement autonomes vis à vis de cela également. Nous avons également pensé à rendre notre solution compréhensible et réutilisable par d'autres puisque j'ai commenté tout le code que j'ai écrit et que notre processeur est en libre accès sur GitHub. Nous avons également pensé à un ReadMe qui permet de mieux comprendre notre architecture.

J'ai pu également, sur le plan technique toujours, pu développer ma compétence **mettre en œuvre une démarche de recherche fondamentale ou appliquée à des fins d'innovation** car j'ai pu **mettre en œuvre une méthodologie de recherche et de développement** mais également car nous avons essayé de **mettre en œuvre la solution technique la plus pertinente** aux vues du temps imparti.

Sur le plan "softskills" maintenant, j'ai pu beaucoup développer la compétence **coopérer dans une équipe ou en mode projet** car j'ai essayé d'être au **maximum moteur dans mon groupe**, car nous avons utilisés **des méthodes de gestion** de projet mais également car nous avons réalisés une présentation orale dans laquelle nous avons au maximum **adapté notre communication au public et aux enjeux**.

Pour conclure je dirais que ce projet a été vraiment important pour moi dans la formation que j'ai reçue cette année. Il m'a permis de prendre confiance en moi à propos de mes capacités à réaliser un projet dont je n'aurais pas pensé être capable il y à quelques mois. Il m'a également permis de consolider énormément mes connaissances mais aussi de beaucoup les approfondir. Enfin, il m'a permis de mieux savoir travailler en équipe, de mieux organiser la répartition des tâches et de mieux pouvoir évaluer à l'avenir la durée d'une tâche en électronique numérique.

8.1.2 Antoine CHASTAND

Ce Projet de fin d'année a été pour moi l'occasion d'utiliser et d'améliorer de nombreuses compétences acquises ces deux dernières années tout en utilisant les connaissances acquises durant mon cursus.

Tout d'abord, c'est par un bon travail d'équipe que nous avons assuré la bonne réalisation de ce projet. J'ai donc développé la compétence **Coopérer dans une équipe OU en mode projet**. En effet, pour mener à bien l'ensemble des étapes à temps, nous avons mis en place **différents outils de gestion de projet**. Le premier et le plus simple était la création d'un groupe de discussion sur lequel nous pouvions échanger rapidement et nous tenir ainsi informés de nos progrès. Le deuxième fut la mise en place d'un diagramme de Gantt qui nous a permis de répartir efficacement les tâches entre chaque membre et sur une période de temps donnée pour ainsi mieux organiser toutes les étapes du projet. Le troisième était la création d'un dossier partagé sur GitHub afin que tous les membres aient accès en même temps aux dernières modifications que chacun a apportées. Enfin, pour la présentation finale de notre projet, nous devions rester clairs et simples, ne pas compliquer nos explications et rester concis **en adaptant notre communication au public et aux enjeux**.

Ensuite, c'est bien techniquement que des progrès ont été réalisés. J'ai fait de nombreuses recherches mais j'ai aussi appliqué et approfondi tout ce que j'ai appris cette année afin de réaliser au mieux ce projet.

J'ai donc d'abord développé la compétence **Concevoir ou réaliser des solutions techniques ou expérimentales, permettant de répondre à un cahier des charges**. Nous avons choisi un processeur RISC-V à 5 étages pour notre cahier des charges. Pour le concevoir, il a fallu **définir les fonctions de chaque bloc ou module le composant**. Ensuite, chaque membre s'occupait d'un certain nombre de blocs. Nous devions alors chacun **concevoir et tester (sur ModelSim)** de notre côté les blocs créés **individuellement** avant de les mettre en commun et définir des **modifications** si nécessaire, **tout ça dans un laps de temps limité** donné lors de la répartition des tâches. J'ai donc appris **à produire une solution fonctionnelle à un problème technique complexe, avec une méthodologie d'évaluation, afin d'obtenir des résultats analysés et compris.** J'ai aussi procédé à la synthétisation du processeur entier grâce aux logiciels de synthèse **Synopsis** et **Design Vision**, cela m'a permis de déboguer et d'améliorer la structuration des codes. Enfin, j'ai réalisé le **placement et routage** sur l'outil Innovus afin d'avoir la possibilité, par la suite, d'implémenter notre processeur sur **cible ASIC**.

La compétence de **mettre en œuvre une démarche de recherche fondamentale ou appliquée à des fins d'innovation** a aussi été développée. En effet, pour créer ce processeur RISC-V nous avons tout d'abord effectué maintes recherches pour choisir quelle structure nous voulions réaliser. Ces recherches nous ont aussi servi à comprendre et choisir les meilleures solutions pour la création des blocs et la répartition des différentes fonctions du circuit entre les différents modules. Finalement, pour trouver la meilleure des solutions, nous avons **mis en œuvre une méthodologie de recherche et développement** en comparant nos **connaissances** de cours, nos **recherches** mais aussi nos **idées**. Chacun apportait son **avis** ou ses **critiques**, et c'est en **discutant** que nous avons choisi de quelle manière nous allions concevoir notre processeur.

Pour conclure, ce projet représente donc une expérience solide pour laquelle j'ai développé mes compétences, approfondi mes connaissances et construit une base solide dans la conception numérique qui m'aidera lors de mes futurs projets.

8.1.3 Gaël OUSSET

Via ce projet j'ai pu grandement améliorer de nombreuses compétences parmi celle proposées dans le référentiel de compétence de Phelma.

Sur l'aspect techniques, la compétence **concevoir ou réaliser des solutions techniques, théoriques ou expérimentales, permettant de répondre à un cahier des charges** est sûrement celle qui a le plus bénéficier de ce projet. Dans un premier temps les étapes de réalisation du processeur et du banc de tests principale, avec l'analyse des résultats et le débogage du processeur de manière global sur les différents fichiers réalisés par moi ou non m'a permis de d'apprendre **à produire une solution fonctionnelle à un problème technique complexe, avec une méthodologie d'évaluation, afin d'obtenir des résultats analysés et compris**. De plus la réutilisation du compilateur réalisé lors du premier semestre de cette année, pour l'adapter à nos nouveaux besoins a permis de **m'appuyer sur mes connaissances pour mettre en œuvre la conception de la technique**. De plus la mise en place pour moi et mes collègues de nouveaux environnements de travail sur *Windows* avec *ModelSim*, des scripts *Batch* et *VSCode*, nous permettant de travailler aussi bien au CIME qu'à l'extérieur, m'a permis de **choisir les outils les mieux adapté**. Enfin la rédaction du fichier README et autres fichier markdown présent sur le git, avec notamment l'usage de nombreux commentaires dans mes codes, et un choix commun avec mes collègues sur le nommage des variables, m'a permis de **rendre la solution proposée compréhensible et réutilisable par d'autres**.

Ce projet m'a aussi grandement permis d'améliorer mes compétences scientifiques via **la mise en œuvre d'une démarche de recherche fondamentale ou appliquée à des fins d'innovation**. Dans un premier temps, la phase de recherche bibliographique initiale sur les architectures RISC-V et la mise en place de notre cahier des charges nous a permis **de proposer et mettre en œuvre une méthodologie de recherche et de développement**. De plus toute la phase de développement de l'architecture, notamment sur les optimisations et le choix de leurs implémentations, les corrections des problèmes, ainsi que le développement du bloc *Control Logic* pour lequel j'étais responsable, m'a permis **de dimensionner et de mettre en œuvre la solution technique la plus pertinente et en la justifiant**.

Enfin l'aspect de travail de groupe dans ce projet avait une place très importante, me conduisant ainsi à développer ma compétence **de coopération dans une équipe ou en mode projet**. Pour commencer cet aspect là nous a conduit à **utiliser différents outils de gestion de projet**, tel que *GitHub* pour la mise en communs de nos différents codes ainsi que le suivie de l'avancement de chacun, *Messenger* pour pouvoir communiquer entre nous, *GDoc* pour la rédaction des différents rapports, et enfin l'usage d'un diagramme de Gantt pour gérer et prévoir l'avancement du projet. De plus le fait d'être autonome dans le cadre de ce projet, et de trouver par moi-même le travail à réaliser m'a permis **de trouver ma place et d'être moteur dans un groupe fonctionnel ou dans un projet technique**.

En conclusion je dirais que ce projet m'a grandement permis d'avancer dans la validation des différentes compétences proposées par Phelma.

8.2 Conclusion

Finalement, ce projet de conception d'un processeur RISC-V a été une très bonne expérience pour laquelle nous avons utilisé et développé nos connaissances tout en appliquant une démarche similaire

au flot de conception étudié en cours. Nous avons mis en place différents outils afin de communiquer efficacement en groupe. Ainsi nos capacités en communication et gestion de projet se sont améliorées. De plus, nous avons réussi à développer un processeur qui est fonctionnel sur beaucoup d'instructions du jeu choisi. Le processeur est de plus synthétisable et le placement et routage a pu être effectué. Pour arriver à ce résultat, chaque membre s'est occupé d'une partie du projet avant de tout mettre en commun. Nous avons donc chacun développé des capacités techniques dans un domaine distinct des autres ou commun lorsqu'il a fallu coder les modules, les tester et les déboguer, une partie qui nous a pris énormément de temps et dans laquelle le groupe entier s'est investi.

En somme toute, ce projet fut enrichissant et représente une solide expérience dans la conception numérique avant les débuts de nos stages.

Pour la répartition des points nous nous mettons 10 chacun.

9 ANNEXE

Voici ci-dessous le jeu d'instruction RV32I avec la description de chaque instruction.
Le code couleur est le suivant :

- En bleu les instructions **REGISTER-REGISTER**
- En rouge les instructions **REGISTER-IMMEDIATE**
- En rose les instructions **de branchement**
- En jaune les instructions **UPPER-IMMEDIATE**
- En vert les instructions **LOAD et STORE**
- En violet les instructions **de saut**
- Sur fond blanc les instructions **non supportées**

Tableau 3 - Instructions RV32I

Instruction	Nom	Format	Description	Implémentation
ADD		add rd,rs1,rs2	Adds the registers rs1 and rs2 and stores the result in rd. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.	x[rd] = x[rs1] + x[rs2]
ADDI	add immediate	addi rd,rs1,imm	Adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI rd, rs1, 0 is used to implement the MV rd, rs1 assembler pseudo-instruction.	x[rd] = x[rs1] + sext(immediate)
AND		and rd,rs1,rs2	Performs bitwise AND on registers rs1 and rs2 and place the result in rd	x[rd] = x[rs1] & x[rs2]
ANDI		andi rd,rs1,imm	Performs bitwise AND on register rs1 and the sign-extended 12-bit immediate and place the result in rd	x[rd] = x[rs1] & sext(immediate)
BEQ	branch equal ?	beq rs1,rs2,offset	Take the branch if registers rs1 and rs2 are equal.	if (x[rs1] == x[rs2]) pc += sext(offset)
SLL	shift left logical	sll rd,rs1,rs2	Performs logical left shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.	x[rd] = x[rs1] << x[rs2]
SRL	shift right logical	srl rd,rs1,rs2	Logical right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2	x[rd] = x[rs1] >>u x[rs2]
OR		or rd,rs1,rs2	Performs bitwise OR on registers rs1 and rs2 and place the result in rd	x[rd] = x[rs1] | x[rs2]
ORI		ori rd,rs1,imm	Performs bitwise OR on register rs1 and the sign-extended 12-bit immediate and place the result in rd	x[rd] = x[rs1] | sext(immediate)
BNE	branch not equal ?	bne rs1,rs2,offset	Take the branch if registers rs1 and rs2 are not equal.	if (x[rs1] != x[rs2]) pc += sext(offset)
SLLI	shift left logical immediate	slli rd,rs1,shamt	Performs logical left shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate. In RV64, bit-25 is used to shamt[5].	x[rd] = x[rs1] << shamt
SRLI	shift right logical immediate	srli rd,rs1,shamt	Performs logical right shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate. In RV64, bit-25 is used to shamt[5].	x[rd] = x[rs1] >>u shamt
XOR		xor rd,rs1,rs2	Performs bitwise XOR on registers rs1 and rs2 and place the result in rd	x[rd] = x[rs1] ^ x[rs2]
XORI		xori rd,rs1,imm	Performs bitwise XOR on register rs1 and the sign-extended 12-bit immediate and place the result in rd Note, "XORI rd, rs1, -1" performs a bitwise logical inversion of register rs1(assembler pseudo-instruction NOT rd, rs)	x[rd] = x[rs1] ^ sext(immediate)
BGE	branch greater or equal	bge rs1,rs2,offset	Take the branch if registers rs1 is greater than or equal to rs2, using signed comparison.	if (x[rs1] >=s x[rs2]) pc += sext(offset)

SLT		slt rd,rs1,rs2	Place the value 1 in register rd if register rs1 is less than register rs2 when both are treated as signed numbers, else 0 is written to rd.	x[rd] = x[rs1] <s x[rs2]
SLTU		sltu rd,rs1,rs2	Place the value 1 in register rd if register rs1 is less than register rs2 when both are treated as unsigned numbers, else 0 is written to rd.	x[rd] = x[rs1] <u x[rs2]
SRA	shift right arithmetical immediate	sra rd,rs1,rs2	Performs arithmetic right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2	x[rd] = x[rs1] >>s x[rs2]
LUI	load upper immediate	lui rd,imm	Build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros.	x[rd] = sext(immediate[31:12] << 12)
BGEU	branch greater or equal unsigned	bgeu rs1,rs2,offset	Take the branch if registers rs1 is greater than or equal to rs2, using unsigned comparison.	if (x[rs1] >=u x[rs2]) pc += sext(offset)
SLTI	set less than immediate	slti rd,rs1,imm	Place the value 1 in register rd if register rs1 is less than the signextended immediate when both are treated as signed numbers, else 0 is written to rd.	x[rd] = x[rs1] <s sext(immediate)
SLTIU	set less than immediate unsigned	sltiu rd,rs1,imm	Place the value 1 in register rd if register rs1 is less than the immediate when both are treated as unsigned numbers, else 0 is written to rd.	x[rd] = x[rs1] <u sext(immediate)
SRAI	shift right arithmetical immediate	srai rd,rs1,shamt	Performs arithmetic right shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate. In RV64, bit-25 is used to shamt[5].	x[rd] = x[rs1] >>s shamt
AUIPC	add upper immediate to pc	auipc rd,imm	Build pc-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register rd.	x[rd] = pc + sext(immediate[31:12] << 12)
BLT	branch less than ?	blt rs1,rs2,offset	Take the branch if registers rs1 is less than rs2, using signed comparison.	if (x[rs1] <s x[rs2]) pc += sext(offset)
LB	load byte ?	lb rd,offset(rs1)	Loads a 8-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.	x[rd] = sext(M[x[rs1] + sext(offset)][7:0])
LH	load hex ?	lh rd,offset(rs1)	Loads a 16-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.	x[rd] = sext(M[x[rs1] + sext(offset)][15:0])
LW	load word ?	lw rd,offset(rs1)	Loads a 32-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.	x[rd] = sext(M[x[rs1] + sext(offset)][31:0])
SB	store byte ?	sb rs2,offset(rs1)	Store 8-bit, values from the low bits of register rs2 to memory.	M[x[rs1] + sext(offset)] = x[rs2][7:0]
BLTU	branch less than unsigned	bltu rs1,rs2,offset	Take the branch if registers rs1 is less than rs2, using unsigned comparison	if (x[rs1] <u x[rs2]) pc += sext(offset)
LBU	load byte unsigned ?	lbu rd,offset(rs1)	Loads a 8-bit value from memory and zero-extends this to XLEN bits before storing it in register rd.	x[rd] = M[x[rs1] + sext(offset)][7:0]
LHU	load hex unsigned ?	lhu rd,offset(rs1)	Loads a 16-bit value from memory and zero-extends this to XLEN bits before storing it in register rd.	x[rd] = M[x[rs1] + sext(offset)][15:0]
SW	store word ?	sw rs2,offset(rs1)	Store 32-bit, values from the low bits of register rs2 to memory.	M[x[rs1] + sext(offset)] = x[rs2][31:0]
SH	store hex ?	sh rs2,offset(rs1)	Store 16-bit, values from the low bits of register rs2 to memory.	M[x[rs1] + sext(offset)] = x[rs2][15:0]
JAL	**jump and link**	**jal rd,offset**	**Jump to address and place return address in rd.**	**x[rd] = pc+4; pc += sext(offset)**
CSRRW	atomic read/write CSR.	csrrw rd,offset,rs1	Atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to XLEN bits, then writes it to integer register rd. The initial value in rs1 is written to the CSR. If rd=x0, then the instruction shall not read the CSR and shall not cause any of the side effects that might occur on a CSR read.	t = CSRs[csr]; CSRs[csr] = x[rs1]; x[rd] = t

CSRRS	atomic read and set bits in CSR.	csrrs rd,offset,rs1	Reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written).	t = CSRs[csr]; CSRs[csr] = t | x[rs1]; x[rd] = t
CSRRC	atomic read and clear bits in CSR	csrrc rd,offset,rs1	Reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected.	t = CSRs[csr]; CSRs[csr] = t &~x[rs1]; x[rd] = t
ECALL	Non utilisé			
JALR	**jump and link register**	**jalr rd,rs1,offset**	**Jump to address and place return address in rd.**	**t =pc+4; pc=(x[rs1]+sext(offset))&~1; x[rd]=t**
CSRRWI		csrrwi rd,offset,uimm	Update the CSR using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field.	x[rd] = CSRs[csr]; CSRs[csr] = zimm
CSRRSI	A intégrer après ?	csrrsi rd,offset,uimm	Set CSR bit using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field.	t = CSRs[csr]; CSRs[csr] = t | zimm; x[rd] = t
CSRRCI		csrrci rd,offset,uimm	Clear CSR bit using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field.	t = CSRs[csr]; CSRs[csr] = t &~zimm; x[rd] = t
EBREAK	Non utilisé			
SUB		sub rd,rs1,rs2	Subs the register rs2 from rs1 and stores the result in rd. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.	x[rd] = x[rs1] - x[rs2]
FENCE	Non utilisé ?	fence pred, succ	Used to order device I/O and memory accesses as viewed by other RISC-V harts and external devices or coprocessors. Any combination of device input (I), device output (O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same. Informally, no other RISC-V hart or external device can observe any operation in the successor set following a FENCE before any operation in the predecessor set preceding the FENCE.	Fence(pred, succ)
FENCE.I	Non utilisé ?	fence.i	Provides explicit synchronization between writes to instruction memory and instruction fetches on the same hart.	Fence(Store, Fetch)