الجمهورية الجزائرية الديمقراطية الشعبية
République Algérienne démocratique et populaire

وزارة التعليم العالي والبحث العلمي
Ministère de l'enseignement supérieur et de la recherche scientifique

جامعة سعد دحلب البليدة
Université SAAD DAHLAB de BLIDA

كلية التكنولوجيا
Faculté de Technologie

قسم الإلكترونيك
Département d'Électronique

Mémoire de Master

Domaine : Sciences et Technologie
Filière : Electronique
Spécialité : Electronique Des Systèmes Embarqués

Présenté par

Chanane Yasmina & Chanane Narimane

Thème

Conception et implémentation d'un processeur RISC-V à 5-étages sur FPGA

Encadré par :

M.Bakiri & H.Bougherira

Année Universitaire 2023-2024

الجمهورية الجزائرية الديمقراطية الشعبية
République Algérienne démocratique et populaire

وزارة التعليم العالي والبحث العلمي
Ministère de l'enseignement supérieur et de la recherche scientifique

جامعة سعد دحلب البليدة
Université SAAD DAHLAB de BLIDA

كلية التكنولوجيا
Faculté de Technologie

قسم الإلكترونيك
Département d'Électronique

Mémoire de Master

Domaine : Sciences et Technologie
Filière : Electronique
Spécialité : Electronique Des Systèmes Embarqués

Présenté par

Chanane Yasmina & Chanane Narimane

Thème

Conception et implémentation d'un processeur RISC-V à 5-étages sur FPGA

Encadré par :

M.Bakiri & H.Bougherira

Année Universitaire 2023-2024

Remerciements

Nous commençons par exprimer notre gratitude à Allah, dont la guidance et le soutien ont été nos sources d'inspiration tout au long de ce projet. Nous adressons nos remerciements à nos deux promoteurs, MR M. BAKIRI et Mme H. BOUGHERIRA, pour leur expertise, et leur soutien qui ont permis l'accomplissement de ce modeste travail.

À nos chers parents, nous sommes profondément reconnaissantes pour leur soutien indéfectible et leurs prières constantes qui ont été nos piliers durant tout notre parcours. Nous tenons à remercier également notre famille pour leurs aides et encouragements.

Nous sommes reconnaissantes envers l'Université SAAD DAHLAB BLIDA 1 et ses enseignants du département d'électronique pour leur engagement envers la qualité de l'enseignement et les ressources qu'ils ont mises à notre disposition. Un merci particulier au Centre de Développement des Technologies Avancées « CDTA » pour avoir accepté notre demande de stage pour la réalisation de notre projet au sein de leur établissement.

Enfin, nous sommes reconnaissantes envers toutes les personnes qui ont joué un rôle, direct ou indirect, dans ce projet, ainsi qu'à tous ceux qui nous ont soutenus de quelque manière que ce soit. Que chacun trouve ici l'expression de notre profonde gratitude.

Narimane et Yasmina

Dédicace

Nous dédions ce modeste travail à notre chère mère, symbole de l'amour et de la tendresse, qui nous a toujours soutenus et encouragés pendant toute notre scolarité.

À celui qui a sacrifié pour que nos études puissent se dérouler dans des meilleures conditions morales et financières, notre cher père, symbole de sagesse et de sacrifice.

À toute notre famille. À nos tantes.

À nos chers cousins Iman, Mohamed et Hassina. À nos amis Radia, Yasmine, Nesrine, Tahar, Célia et Madina.

À nos collègues du département de l'électronique avec qui nous avons passé des moments exceptionnels.

À toutes nos connaissances et personnes que nous aimons.

ملخص:

المصطلح "المصدر المفتوح" ينتشر بشكل متزايد في مجال المعالجات والبرمجيات. يشير إلى الوصول المفتوح إلى التصاميم والأكواد للجمهور العام دون الحاجة إلى دفع تكاليف التراخيص التجارية.

يقدم هذا العمل تصميم وتنفيذ معالج مفتوح المصدر من نوع RISC-V RV32I مع خط أنابيب من خمس مراحل. يتكون هذا المشروع من ثلاث أجزاء. الجزء الأول يتعلق بتصميم المعالج ذو الدورة الواحدة (single cycle). الجزء الثاني يتضمن تنفيذ خط الأنابيب من خمس مراحل استنادًا إلى المعالج ذو الدورة الواحدة الذي تم تطويره في الجزء الأول. أخيرًا، الجزء الثالث مكرس للتحقق من الأداء من خلال العديد من الاختبارات.

كلمات المفاتيح: المصدر المفتوح؛ خط أنابيب؛ المعالج أحادي الدورة

Résumé :

Le terme open source se répand de plus en plus dans le domaine des processeurs et des logiciels. Il désigne l'accès ouvert aux conceptions et aux codes pour le grand public, sans avoir à payer les frais des licences commerciales.

Ce travail présente la conception et l'implémentation d'un processeur open source RISC-V RV32I avec un pipeline de 5 étapes. Ce projet se compose de trois parties. La première partie concerne la conception du processeur à cycle unique (single cycle). La deuxième partie consiste en la réalisation du pipeline de 5 étapes à partir du processeur à cycle unique développé dans la première partie. Enfin, la troisième partie est consacrée à la vérification du fonctionnement par plusieurs tests.

Mots clés : Open source ; single cycle ; pipeline, RISC-V RV32I

Abstract:

The term open source is becoming more and more widespread in the field of processors and software. It means open access to designs and codes for the general public, without having to pay commercial licensing fees.

This work presents the design and implementation of an open source RISC-V RV32I processor with a 5-stage pipeline. This project consists of three parts. The first part concerns the design of the single cycle processor. The second part consists of the creation of the 5-step pipeline from the single-cycle processor developed in the first part. Finally, the third part is devoted to verifying operation by several tests.

Keywords: pipeline; RISC-V RV32I; single cycle

Liste des figures

Liste des tableaux

Table des matière

Introduction Générale

Le terme open source se répand de plus en plus dans le domaine des logiciels et des processeurs. Il désigne l'accès ouvert aux conceptions et aux codes pour le grand public sans avoir à payer les frais des licences commerciales. De nombreuses entreprises et organisations entrent dans l'aventure en proposant des versions open sources flexibles. L'open source contribue au partage des idées et des ressources, ce qui accélère le développement technologique et permet aux étudiants d'exploiter les différentes conceptions matérielles des processeurs ainsi les codes sources des logiciels.

Plusieurs projets ont été élaborés dans le domaine des microprocesseurs afin de créer un microprocesseur open source performant, notamment l'architecture RISC-V. RISC-V est une architecture d'ensemble d'instructions (ISA) open source basée sur les principes établis de l'ordinateur à jeu d'instructions réduit (RISC). Elle a été développée à l'université de Berkeley en Californie, et elle s'est imposée dans ce domaine grâce à sa simplicité et sa flexibilité.

Dans le cadre de ce projet de fin d'études, nous avons conçu et implémenté un processeur RISC-V de 32 bits (RV32I), nous avons commencé par des recherches préliminaires afin de comprendre les concepts de cette architecture ainsi que l'architecture des processeurs, leurs composants et leur fonctionnement interne.

Dans le premier chapitre, nous nous concentrons sur les termes fondamentaux utilisés dans le contexte de notre projet, pour avoir une idée générale du domaine, notamment l'architecture RISC-V et ses spécificités.

Le deuxième chapitre est consacré à la conception de notre processeur. Nous avons commencé par la réalisation des datapaths pour chaque type d'instruction, puis les avons assemblés pour former un datapath complet. Une fois le datapath global obtenu, nous avons ajouté l'unité de contrôle et l'unité de fetch d'instruction (lecture de l'instruction) pour obtenir notre processeur désiré.

Dans le troisième chapitre, nous avons utilisé plusieurs outils pour simuler notre réalisation. Nous avons écrit les codes avec Icarus Verilog, effectué les simulations et analysé les résultats avec GTKWave pour visualiser les signaux, et utilisé Vivado pour générer les schémas RTL.

Chapitre 1

Etat de l'art et généralités

1.1. Introduction

Aujourd'hui, les microprocesseurs sont partout dans notre vie quotidienne : dans nos ordinateurs, nos téléphones, nos voitures et même nos appareils électroménagers. Des entreprises comme Intel et AMD les ont rendus de plus en plus puissants. Récemment, des processeurs open source comme RISC-V sont apparus. Ces processeurs peuvent être utilisés et modifiés librement par tout le monde, ce qui encourage l'innovation et la collaboration.

Dans ce premier chapitre, nous allons explorer les différentes architectures des microprocesseurs ainsi leurs concepts. Ensuite nous allons aborder les architectures open source en appuyant sur RISC-V et ses spécificités.

1.2. Définitions générales

Cette section fournit des explications et des descriptions des concepts clés et des terminologies utilisées dans le domaine des architectures de processeurs.

1.2.1. Microprocesseur

Le microprocesseur, considéré comme le cerveau de l'ordinateur ou de n'importe quel appareil électronique, est un circuit intégré qui regroupe tous les éléments nécessaires à l'interprétation et l'exécution des instructions arithmétiques, logiques, de contrôle et transfert des données. Il est utilisé dans une application ou la tâche n'est pas prédéfinie ou nécessite un traitement intensif. [1]. Il utilise du différent type de bus pour assurer la communication et la gestion de données, définies ci-dessous :

- **Bus d'adresse :** En utilisant le bus d'adresse, le microprocesseur peut choisir la cellule mémoire ou le périphérique auquel il souhaite accéder pour lire ou écrire des informations (instructions ou données).

- **Bus de données:** ce bus est bidirectionnel, utilisé pour transférer des informations entre les différents blocs. Ces informations peuvent être soit des instructions, soit des données provenant ou à destination de la mémoire ou des périphériques.

- **Bus de contrôle:** ce bus transporte les signaux de contrôle nécessaires pour coordonner et gérer les différentes opérations dans le système. Par exemple, vérifier si l'opération en cours est lue ou écrite et si un périphérique nécessite une interruption, etc.

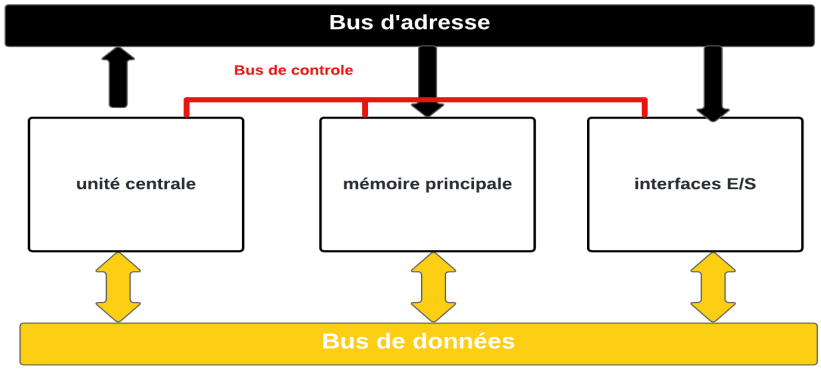

Figure 1. 1: Schéma illustrant le microprocesseur

1.2.2. Microcontrôleur

Un microcontrôleur est un circuit intégré contenant les éléments principaux de l'ordinateur tels qu'un microprocesseur, une mémoire et des périphériques d'entrée/sortie sur un seul boîtier. Selon sa conception, il peut traiter des données en blocs de 8, 16 ou 32 bits, ce qui détermine la quantité de données qu'il peut traiter simultanément et la taille des instructions qu'il peut exécuter. On le trouve dans des systèmes embarqués, dans les véhicules, machine à laver et dans d'autres appareils médicaux et ménagers. Il est utilisé pour des tâches spécifiques en fonction des entrées fournies au microcontrôleur, qui les traite pour donner des résultats. Les entrées peuvent provenir des capteurs.[3]

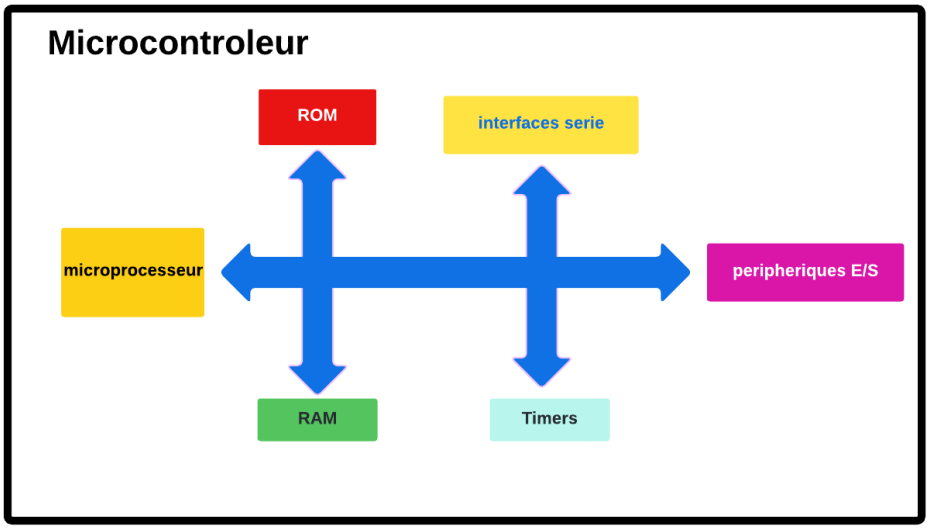

Figure1.2: Schéma illustrant le microcontrôleur

1.2.3. Architecture

L'architecture est la vue du programmeur d'un processeur. Elle est définie par le jeu d'instructions (langue) et les emplacements d'opérande (registres et mémoire). De nombreuses architectures différentes existent, telles que, MIPS, SPARC, CISC et RISC. [4]

Cette section présente les différents types d'architectures de processeurs Harvard, von Neumann, RISC et CISC ainsi leurs concepts.

1.2.3.1. Von Neumann

L'architecture de Von Neumann est une architecture d'ordinateur qui a été présentée dans un article écrit par le mathématicien John Von**.** Diffusé en 1945, le document discutait d'un projet de construction d'un ordinateur particulier nommé EDVAC. Il a décrit une architecture pour une machine numérique électronique qui offre la possibilité de conserver à la fois ses programmes et ses données dans une seule mémoire.

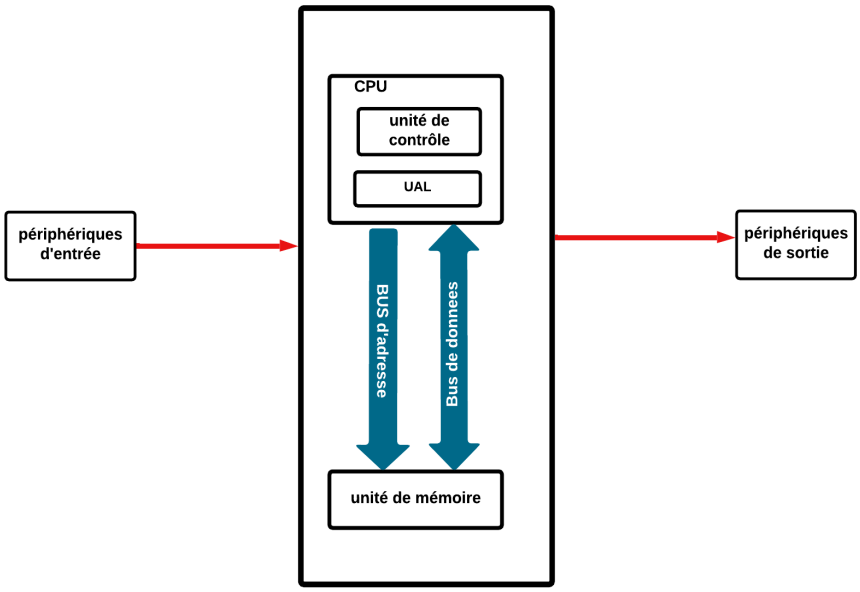

Figure 1.3: Schéma illustrant l'architecture Von Neumann

1.2.3.2. Architecture Harvard

L'architecture Harvard nommée d'après l'université prestigieuse de Harvard, États-Unis où elle a été conçue et inventée pour un premier projet nommé Harvard Mark-1 développé en 1944.

La caractéristique principale de cette architecture est la séparation des unités de mémoire de données et d'instruction permettant un accès simultané aux deux types de mémoire. Conceptuellement, les instructions et la mémoire occupent différents espaces de mémoire et une seule adresse ne peut pas identifier de manière unique l'information. Cela nécessitait des instructions d'accès mémoire similaires pour les instructions et la mémoire de données séparément. Ce qui permet d'alimenter simultanément l'unité centrale avec les instructions du

programme et les informations correspondantes. Il y a donc deux bus dans cette structure, l'un pour les programmes et l'autre pour les données. En outre, le noyau central gère directement l'unité de calcul et gère également les entrées et sorties.

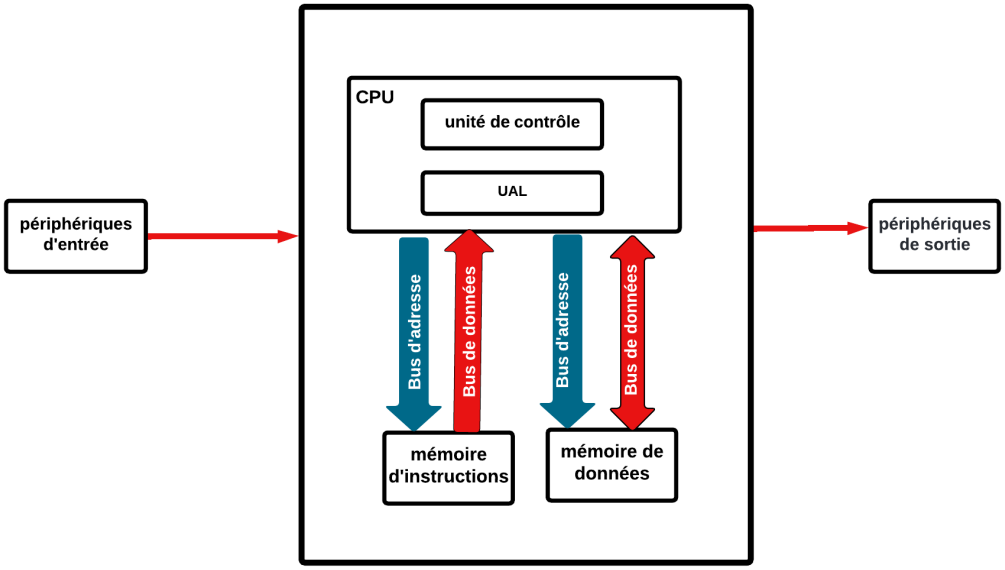

Figure 1.4 :Schéma illustrant l'architecture Harvard

Comparaison entre l'architecture Von Neumann et Harvard

Architecture Von Neumann	Architecture Harvard
- Les codes et les données sont dans la même mémoire.	- Code et données dans une mémoire séparée.
- Une seule mémoire connectée au CPU	- La mémoire de données et la mémoire de programme sont connectées séparément à la CPU
- Pipeline plus complexe à implémenter.	- Pipeline simple à implémenter
- Il existe un bus commun pour le transfert de données et d'instructions.	- Des bus séparés sont utilisés pour transférer les données et les instructions.
- Coûte moins cher	- Plus cher que Von Neumann Architecture

Tableau 1.1: Comparaison entre l'architecture Von Neumann et Harvard

1.2.3.3. Architecture CISC

L'architecture de processeur CISC se distingue par un jeu d'instructions large et complexe considéré comme complet, intégrant les opérations arithmétiques, logiques, de mouvement de

données et de contrôle des flux reflétant une philosophie de conception complexe, cependant, la complexité accrue peut également entraîner des délais de traitement plus lents.

L'objectif principal de l'architecture CISC est d'accomplir une tâche en aussi peu de lignes d'assembleur que possible. Ceci est réalisé en construisant du matériel de processeur capable de comprendre et d'exécuter une série d'opérations. [4]

Les projets de L'architecture CISC majeurs qui ont marqués l'histoire des processeurs sont mentionnés dans la figure 1.5 :

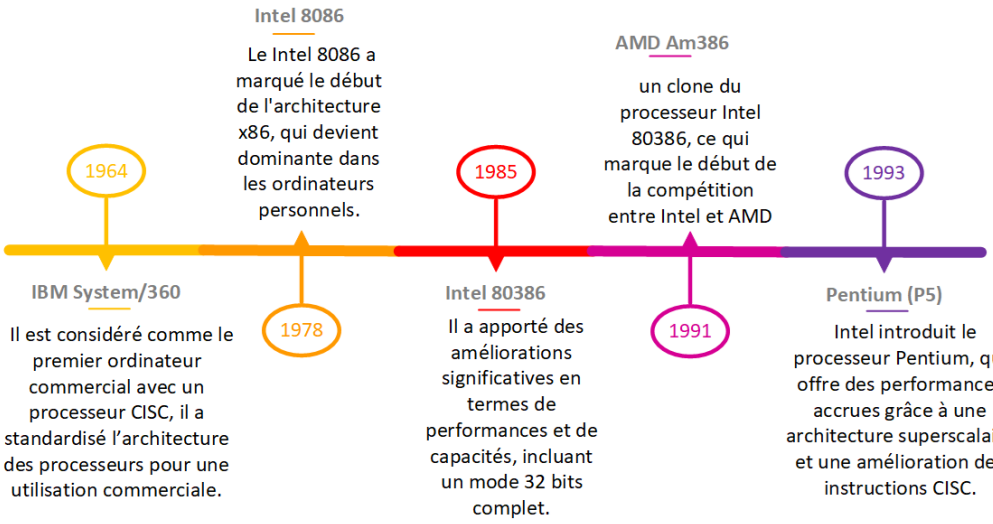

Figure 1.5: Chronoloigie des processeurs de l'architectures CISC

Malgré la période florissante de l'architecture CISC, la complexité liée à l'intégration de plusieurs instructions nécessitant un processus de décodage et d'exécution plus complexe et demandant plus de matériel a conduit à l'apparition de l'architecture RISC.

1.2.3.4. Architecture RISC

RISC signifie un processeur à jeu d'instructions réduit, un type d'architecture qui utilise des instructions simples et rapides pour effectuer des tâches complexes. Les racines du concept RISC remontent en effet aux années 1960, avec le projet IBM System/360, était un système informatique influent qui a évoqué certains principes plus tard vus dans les architectures RISC. Bien qu'il s'agisse d'un système CISC, ses décisions de conception ont influencé les développements futurs de l'architecture des processeurs, soulignant l'importance d'une exécution efficace des instructions.

L'idée de l'architecture RISC est d'aller vers la simplicité, de revenir à une petite philosophie, qui était d'essayer de rendre les processeurs plus simples pour les rendre plus rapides et plus rentables, en effet les processeurs passent la majorité de leur temps environ 80%, à faire quelques simples instructions laissant les instructions plus complexes à la

minorité de leur temps, et en abordant l'idée d'optimiser la machine pour faire les choses les plus simples autant que possible, il se peut qu'il arrive à créer un processeur efficace et moins couteux.

L'un des principaux travaux a été effectué à IBM Research grâce à l'équipe de Cocke gérée par John Cocke, un des pionniers de l'architecture RISC, ils ont développé en 1974, le concept original de RISC en prouvant avec le principe de Pareto, qu'environ 20% des instructions dans un ordinateur faisant 80% du travail. En 1980, l'équipe de Cocke a produit un premier projet prototype qui présente pour la première fois l'architecture RISC sous le nom d'IBM 801, nommé d'après le numéro du bâtiment où elle a été créée en lui donnant des instructions simples qui pouvaient être exécutées en un seul cycle.

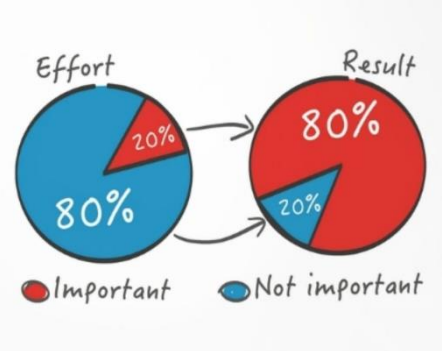

Figure1.6 : Le principe de Pareto

Le principe de Pareto : également connu sous le nom de la règle 80-20, spécifie que 80% des résultats sont issus de 20% d'effort, nommé d'après l'économiste italien Vilfredo Pareto, le principe de Pareto rappelle de manière générale que la relation entre les intrants et les extrants n'est pas équilibrée.

Les principaux projets de l'architecture RISC qui ont influencé l'évolution des processeurs sont présentés dans la figure 1.7 :

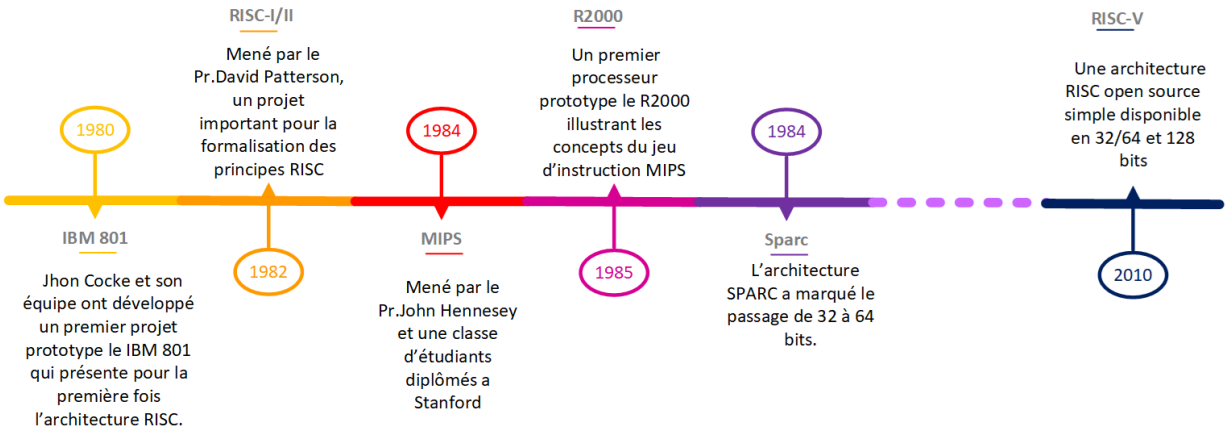

Figure1.7 : Chronologie de quelques processeurs RISC

1.2.4. Microarchitecture

La microarchitecture qui est la connexion entre logique et architecture. C'est l'implémentation de la manière dont les instructions vont être exécutées, et l'organisation spécifique des registres, des ALUs, des mémoires et d'autres blocs logiques nécessaires à la mise en œuvre d'une architecture. Il existe plusieurs types des microarchitectures notamment le single-cycle, multi-cycle et le pipeline.[3]

1.2.4.1. Single-cycle vs multi-cycle vs pipeline

Ces trois architectures sont différentes par la manière dont elles traitent les instructions. Le processeur a cycle unique exécute chaque instruction en un seule cycle d'horloge, le processeur a cycles multiples divise l'exécution en plusieurs étapes sur plusieurs cycles d'horloge, tandis que le pipeline permet l'exécution simultanée de plusieurs instructions à différentes étapes.

- **Implémentation à cycle unique (Single cycle)**

Dans un processeur a cycle unique l'instruction doit être exécutée en un seul cycle d'horloge fixe pour toutes les instructions. Le cycle d'horloge est déterminé par le délai de l'instruction la plus lente.

Dans la figure 1.8 nous avons illustré un exemple sur l'exécution de l'instruction de chargement « Load » et de stockage « store » en cycle unique.

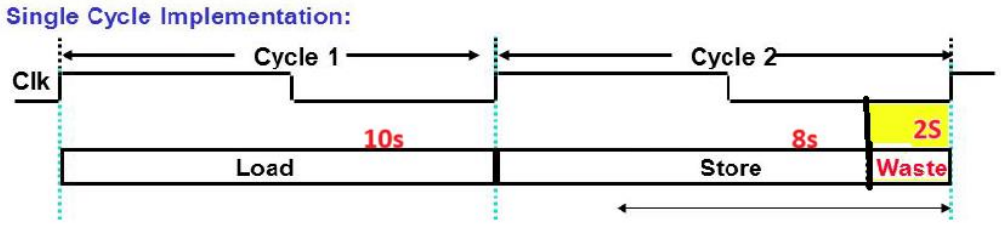

Figure 1.8: exemple d'une l'implémentation Single cycle

Ici, nous remarquons que l'instruction « Load » prend le cycle d'horloge le plus long ce qui force l'instruction store à s'exécuter dans la même durée, bien qu'elle n'ait besoin que de 8 secondes au lieu de 10 secondes. Ce qui a mené à une perte de temps de 2 s. Pour éviter ce problème, il existe l'implémentation multi-cycle.

➢ Implémentation à cycles multiples (Multi-cycle)

Dans l'implémentation à cycles multiples**,** l'instruction est divisée en cinq petits cycles d'horloge distincts (fetch, decode, execute, memory access, write back), tous ayant la même durée.

Le nombre des cycles d'horloge nécessaires pour chaque instruction n'est pas fixe, comme illustré dans la figure 1.9 avec l'exécution de l'instruction « store » qui n'a demandé que 4 cycles, Ce qui représente un avantage par rapport à l'implémentation à cycle unique (single cycle).

En revanche, Les cycles d'horloge doivent avoir la même durée. Le cycle de l'accès à la mémoire prend toujours plus de temps à s'exécuter par rapport aux autres, ce qui entraine une perte de temps dans chaque cycle.

Par exemple, si le cycle de l'accès à la mémoire demande 3 s, et que le fetch, decode, execute et write back demandent chacun 2 s, tous les cycles doivent alors être exécutés en 3 s, ce qui provoque 1s perdue dans chaque cycle, soit un total de 4s Supplémentaires.

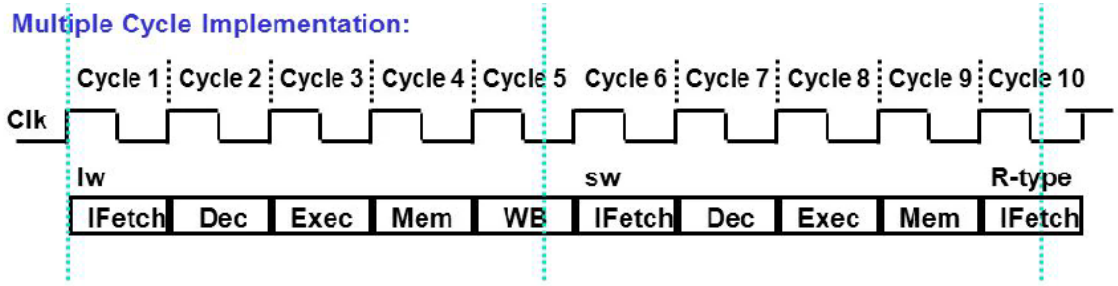

Figure 1. 9: Exemple d'une l'implémentation multi-cycle

➢ Implémentation avec Pipeline :

Le pipeline a le même principe que le multi-cycle chaque instruction est divisée en cinq étages distincts, fetch, decode, execute, memory access et write back.

Le pipeline implique le chevauchement des instructions dans l'exemple de l'exécution des deux instructions de chargement « load » et de stockage « store » illustrées dans la figure1.10 suivante :

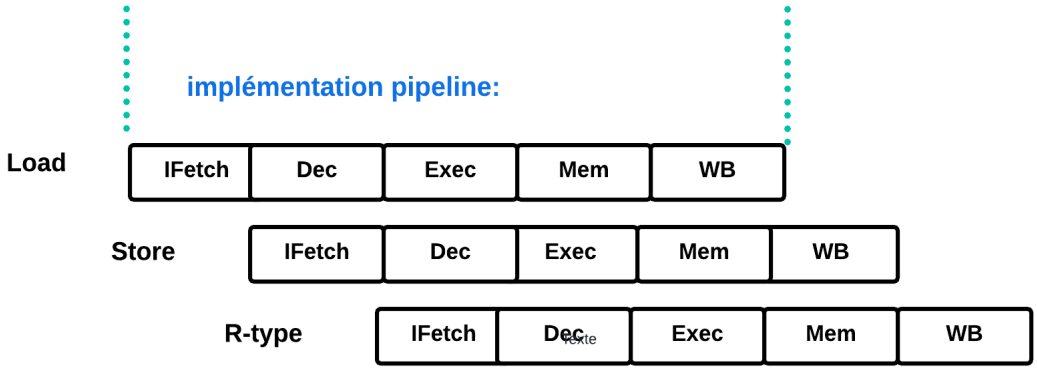

Figure 1.10: Exemple d'une implémentation pipeline

Lorsque l'instruction load commence son exécution par l'étape de fetch, l'instruction store entame également cette phase au moment où load passe à l'étape de décodage. Ainsi, load et store sont exécutées en parallèle. Par exemple, si chaque phase prend 1 seconde, load prendra 5 secondes pour compléter son exécution. Cela signifie que l'exécution totale de load et store prendra 6 secondes. [4]

1.2.5. Architecture open source

Le terme du processeur open source, est un type de processeur ou sa conception matérielle est ouverte et accessible au grand public. Ça veut dire que n'importe qui peut modifier, examiner et distribuer la conception de processeur sans avoir besoin de recourir à des licences propriétaires.

Quelques exemples des projets open source :[5]

- **OpenSPARC :** Sun MicroSystems a été le premier acteur majeur à se lancer dans cette aventure, en présentant son processeur OpenSPARC T1 en 2005 et OpenSPARC T2 en 2007. Il a été doté de puces 64 bits à huit cœurs très performants à cette époque.

- **RISC-V :** le projet RISC-V a été développé à l'université de Berkeley en Californie par le Prof.Krste Asanović et ses étudiants Yunsup Lee et Andrew Waterman en 2010. C'est une architecture de jeu d'instructions RISC open source caractérisée par sa conception simple et extensible. Disponible en 32 bits ,64bits et 128bits.

- **OpenPower :** En 2013, IBM a lancé la plateforme OpenPOWER, une initiative visant à créer une plateforme ouverte autour de son architecture Power, destinée spécifiquement aux serveurs. Bien que le jeu d'instructions Power (Power ISA) soit bien documenté, ce n'est qu'en été 2019 qu'IBM a annoncé qu'il serait accessible sous licence ouverte, avec la publication définitive de cette licence en février 2020. En juillet et septembre 2020, IBM a livré ses deux premiers cœurs Power sous licence ouverte : l'A2I et l'A2O.

- **OpenRISC :** L'openRISC est une architecture de jeu d'instructions ISA open-source développée par le projet openCores à la fin des années 1990 et au début des années 2000. Il s'agit d'une ISA 32 bits ou 64 bits qui prend en charge un sous-ensemble du jeu d'instructions MIPS, ainsi que des instructions supplémentaires pour le traitement en virgule flottante.[6]

1.2.5.1. Architecture RISC-V

Dans le cadre de notre projet nous avons choisi de travailler avec l'architecture open source RISC-V, spécifiée ci-dessous :[7]

1.2.5.2. Description de l'architecture RISC-V

L'architecture d'un processeur définie le jeu d'instructions et les emplacements d'opérande dans les registres ou les mémoires, considérée comme la vue du programmeur. L'architecture ne définit pas l'implémentation matérielle mais plutôt l'organisation du processeur, de la mémoire et d'autres périphériques, la performance et des concepts généraux. Plusieurs architectures de différents processeurs peuvent exécuter les mêmes programmes, mais elles utilisent différentes implémentations matérielles.

Pour comprendre une architecture d'un processeur, il faut d'abord apprendre sa langue. Les mots sont appelés « Instructions », le vocabulaire est appelé « Instructions set » ou l'ensemble de ces instructions.

1.2.5.2.1.1. Instructions

Les instructions indiquent à la fois l'opération à effectuer et les opérandes à utiliser. Les opérandes peuvent provenir de la mémoire, des registres ou de l'instruction elle-même. Le matériel ne comprend que 1 et 0, donc les instructions sont codées sous forme de nombres binaires dans un format appelé langage machine.

1.2.5.2.1.2. Langage machine

Les microprocesseurs sont des systèmes numériques qui utilisent les nombres binaires pour encoder, lire et exécuter les instructions en langage machine.

1.2.5.2.2. Langage assembleur

La lecture du langage machine est considéré comme ennuyeuse et fastidieuse, c'est pourquoi le langage assembleur intervient pour représenter les instructions de façon lisible par le programmeur.

1.2.5.2.3. Opérandes : Registres, mémoires et valeurs immédiates

Un opérande est une donnée sur laquelle une instruction s'exécute, ces opérandes sont stockés soit dans des registres ou des mémoires codées en langage machine.

- **Les Registres**

Les instructions ont besoin d'un accès rapide aux opérandes pour qu'elles puissent s'exécuter rapidement, les opérandes stockés dans une mémoire prennent beaucoup de temps à être récupérer, c'est pour cette raison que l'architecture RISC-V propose l'utilisation des registres.

Les registres sont un type de mémoire utilisés pour stocker et transférer des données temporairement par le processeur pour un accès immédiat. Ils peuvent tenir soit des données ou des adresses, identifiés par la lettre 'X' indiquant le numéro du registre.

- **La Mémoire**

La mémoire fournit un espace de stockage beaucoup plus important qu'un registre, elle peut stocker les données temporairement ou permanent. Par rapport au registre, la mémoire a de nombreux emplacements de données, mais l'accès prend plus de temps. Alors que le registre est plus petit et plus rapide, l'utilisation d'une combinaison des deux composants, un programme peut accéder à un grand nombre de données et l'exécutées assez rapidement.

L'architecture RISC-V est une architecture Load et Store, ce qui veut dire qu'il n y'a que les instructions de chargement et de stockage qui accèdent la mémoire. RISC-V spécifie également que l'accès à la mémoire est byte-adressable et little-endien.

- **Les valeurs immédiates**

L'architecture RISC-V possède plusieurs types d'instructions, parmi elle, il y a des instructions qui utilisent un champ pour les valeurs immédiates, des constantes que leur valeur se trouve dans l'instruction elle-même et ne nécessite pas un accès au registre ou à la mémoire. Pour utiliser ce champ dans des opérations arithmétiques ou de stockages, il faut étendre ce champ pour atteindre le nombre de bit des opérandes.

1.2.5.3. Les cores du RISC-V

Il existe un nombre considérable des processeurs bases sur l'ISA RISC-V, chacun offrant des caractéristiques spécifiques adaptées à différentes applications.

Dans cette section nous allons examiner brièvement quelques cores afin de comparer leurs principales caractéristiques et utilisations.

- **PicoRV32**

C'est un cœur de processeur RISC-V open-source simple et compact conçu par Clifford wolf. PicoRV32 est écrit en Verilog et Il peut être configuré en tant que noyau RV32E, RV32I, RV32IC, RV32IM ou RV32IMC. Il consomme moins d'énergie grâce à sa simplicité architecturale et sa petite taille ce qui le rend idéal pour les systèmes embarques. Souvent utilisé dans des projets basés sur FPGA en raison de sa taille compacte et de sa simplicité.

Il utilise une architecture sans pipeline et la plupart des instructions sont exécutées en un cycle d'horloge.[8]

- **Boom (The Berkeley Out-of-Order Machine)**

Boom (Out-of-Order Machine), est un core RISC-V RV64GC open source synthétisable et paramétrable écrit dans le langage de construction matérielle Chisel. Le terme « Out-of-order » indique une méthode d'exécution des instructions dans un processeur, qui offre une flexibilité dans l'ordre d'exécution des instructions. Contrairement à l'exécution traditionnelle « in-order » qui suit un ordre strict.[9]

- **ORCA**

ORCA est un noyau RV32IM RISC-V écrit en VHDL et développé par VectorBlox. Il a été conçu pour être adapté aux FPGA. Avec un pipeline en 5 étages. [10]

- **Potato**

Potato est un simple noyau RISC-V écrit en VHDL adapté aux FPGA, il implémente le sous-ensemble entier 32 bits (RV32I) de la spécification RISC-V version 2.0 avec un pipeline classique en 5 étapes.[11]

Le tableau 1.2 illustre la comparaison entre ces quatre cores :

cores	ISA	Langage de description matériel	Pipeline	Fréquence d'horloge	Consommation d'energie
PicoRV32	RV32I, RV32IM, RV32IC, RV32IMC, RV32E	Verilog	-	200 MHz	Faible
Boom	RV64G (I, M, A, F, D)	Chisel	Pipeline out-of-order en 6 étages	100 MHz	Modérée à élevée
ORCA	RV32I, RV32IM	VHDL	Pipeline in-order en 5 etages	125MHz	Faible à modérée
Potato	RV32I	VHDL	Pipeline in-order en 5 étages	50 MHz	Faible

Tableau 1.2: Comparaison entre les cores du RISC-V

1.3. Technologies d'implémentation d'un processeur

Ces technologies d'implémentation d'un processeur englobent les méthodes et les matériaux utilisés pour concevoir et fabriquer les processeurs. Ces technologies influencent la performance, la consommation d'énergie et les coûts de production des processeurs.

1.3.1. Circuit intégré spécifique à une application (ASIC)

Un ASIC est un circuit intégré sur mesure conçu pour un objectif ou une tâche spécifique, mais il n'est pas programmable pour effectuer une grande variété de tâches différentes. Il est plus rapide et consomme moins d'énergie que les autres circuits intégrés. Il existe deux méthodes principales pour réaliser des ASIC : les ASIC entièrement personnalisés et les ASIC semi-personnalisés.

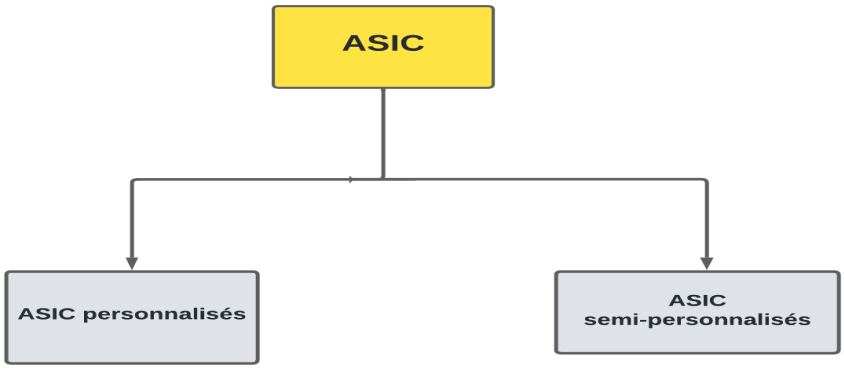

Figure 1.11: Les deux méthodes de réalisation de l'ASIC

- **ASIC entièrement personnalisés :** Ces ASIC sont créés du zéro pour une application précise. Ils offrent le plus haut niveau de performances, faible consommation d'énergie et de rentabilité. En revanche, la conception de ce type des ASIC est compliquée et requiert beaucoup de temps avec une expertise approfondie.

- **ASIC semi-personnalisés** : Les ASIC semi-personnalisées permettent certaines modifications ou configurations afin de répondre aux besoins spécifiques d'un projet sans être entièrement conçues sur mesure. Ils sont moins complexes et moins longs à concevoir que les ASIC entièrement personnalisés, et ils sont également plus rentables. Cependant, ils peuvent ne pas offrir le même niveau de performances et d'efficacité énergétique que les ASIC entièrement personnalisés.

1.3.2. Field programmable gate array (FPGA)

FPGA signifie Field Programmable Gate Arrays , est une puce reprogrammable avec un ensemble de centaines de milliers de portes logiques qui se connectent en interne entre elles pour construire un circuit numérique complexe. Les FPGA peuvent être reprogrammés selon les exigences d'application ou de fonctionnalité souhaitées après la fabrication. Cette fonctionnalité distingue les FPGA des circuits intégrés spécifiques à une application (ASIC), qui sont fabriqués sur mesure pour des tâches de conception spécifiques.

1.4. Conclusion

Les processeurs open source présentent de nombreux avantages, tels que la possibilité de les personnaliser et leur flexibilité. Nous avons choisi l'architecture RISC-V car elle est simple, modulaire et de plus en plus utilisée. Dans le prochain chapitre, nous étudierons en détail l'architecture RV32I de RISC-V, qui constitue une base solide pour comprendre les principes fondamentaux de cette architecture 32 bits.

Chapitre 2

Conception du processeur

2.1. Introduction

Ce chapitre se concentre sur les détails de l'implémentation d'un processeur RV32I. Nous commençons par étudier les spécificités du modèle open source, de son architecture et microarchitecture.

Notre objectif est d'explorer chaque étape, de la planification initiale à la réalisation du processeur. Nous nous intéressons à la compréhension de la manière de concevoir et de mettre en œuvre un processeur RV32I, en mettant l'accent sur la simplicité et l'efficacité de l'architecture RISC-V afin d'assurer des résultats fonctionnels et performants.

2.2. Architecture RV32I []

Dans le premier chapitre, nous avons examiné de manière générale l'architecture RISC-V et ses caractéristiques. Dans cette partie, nous nous intéressons au jeu d'instructions RV32I.

Le jeu d'instruction RV32I est considéré comme une base solide pour les autres variantes notamment pour RV64I (RISC-V 64 bits) et RV128I (RISC-V 128 bits) et ainsi leurs extensions. Ces variantes s'appuient sur les instructions du RV32I en ajoutant de nouvelles instructions spécifiques pour chaque extension.

2.2.1. Spécificités du RV32I

Dans cette partie, nous explorerons les différentes spécificités du jeu d'instructions RV32I, une architecture RISC simple et efficace.

2.2.1.1. Registres

Comme nous avons choisi le RV32I, il existe donc 32 registres X de 32 bits, dont 31 registres X1-X31 à usage général contenant des valeurs entières alors que le registre X0 est câblé à la constante 0. De plus, un autre registre PC « Program Counter » qui comporte l'adresse de l'instruction actuelle.

Les registres sont divisés en catégories en fonction de leur usage, comme illustré dans le tableau 2.1 :

Registre X	ABI	Description
X0	Zéro	Câblé à zéro
X1	ra	Adresse de retour
X2	sp	Pointeur de pile
X3	gp	Pointeur global
X4	tp	Pointeur de fil
X5-X7	t0-t2	Temporaires
X8	s0/t0	Registre-enregistré/pointeur de cadre
X9	s1	Registre enregistré1
X10-X17	a0-a7	Argument de fonction
X18-X27	s2- s11	Registre enregistré
X28-X31	t3- t6	Temporaires
PC	-	Compteur de programme

Tableau 2.1 : Description des registres

2.2.1.2. Capacité et taille des mémoires

Les mots de mémoire dans l'architecture RV32I sont de 32 bits, chaque mot est composé de 4 octets, et comme indiqué précédemment, l'adressage en mémoire est byte-adressable et little-endian. L'architecture RV32I utilise également un espace d'adresse de 32 bits, permettant d'adresser jusqu'à 4 Go de mémoire.

2.2.1.2.1. Mémoire d'instructions

Nous avons utilisé dans notre architecture une mémoire d'instruction de dimensions un octet sur 4096 d'emplacement, cela signifie qu'on a une capacité de 1ko afin d'aligner correctement les adresses et permettre une récupération de données saine.

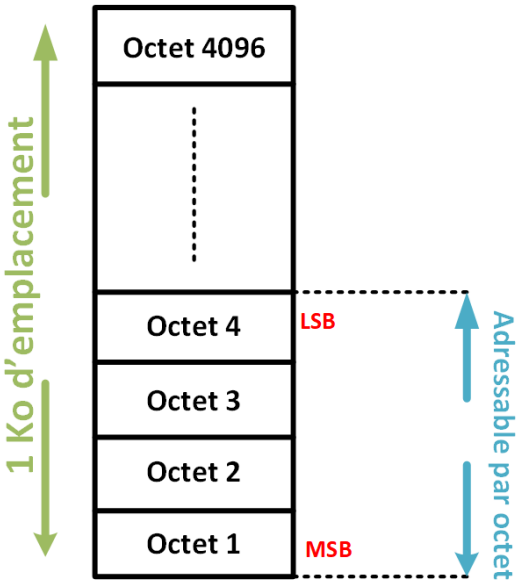

Figure 2.1: Mémoire d'instructions

Le bus de mémoire d'instructions permet le transfert des instructions de la mémoire vers le datapath. Il est généralement de 32 bits de large. Le processeur envoie une adresse d'instruction via le bus d'adresse, et la mémoire renvoie l'instruction correspondante via le bus de données. Cette instruction est ensuite décodée et exécutée par l'unité de contrôle et le datapath.

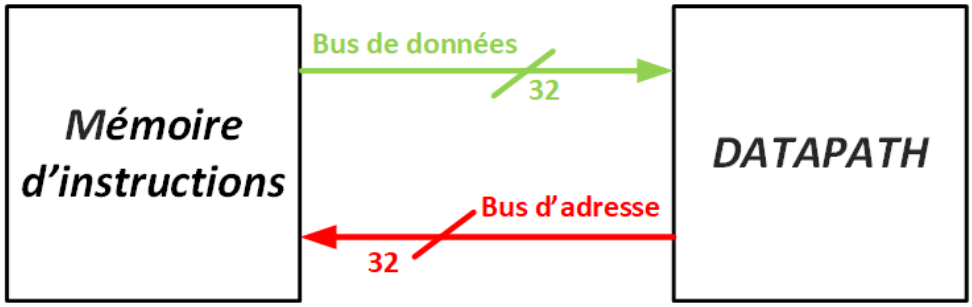

Figure2.2: Partage des données et des addresses

2.2.1.2.2. Mémoire de données

Nous avons utilisé une mémoire de données de 1024 emplacements chacun de 32 bits, elle a une capacité de 1ko et elle a également des bus de données et des bus d'adresse de 32 bits de large, correspondant à la taille des mots de l'architecture.

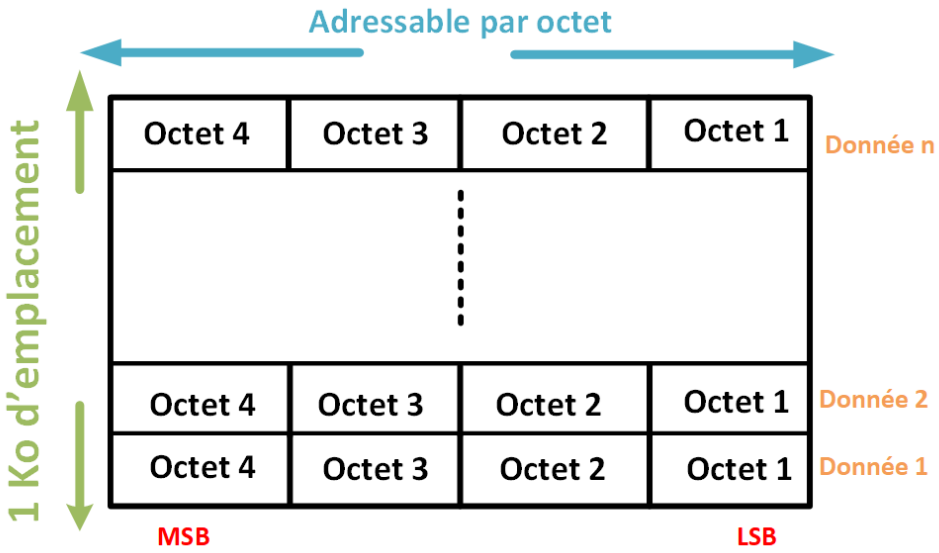

Figure2.3: Mémoire de données

Le processeur utilise la mémoire de données pour transférer des données entre le processeur et la mémoire, les instructions de chargement (Loads) reçoivent une adresse mémoire via le bus d'adresse et résultent la donnée traitée via le bus de donnée. Les instructions de stockage (store) écrivent une donnée dans la mémoire de données via le bus de donnée et envoient une adresse vers la mémoire via le bus d'adresse.

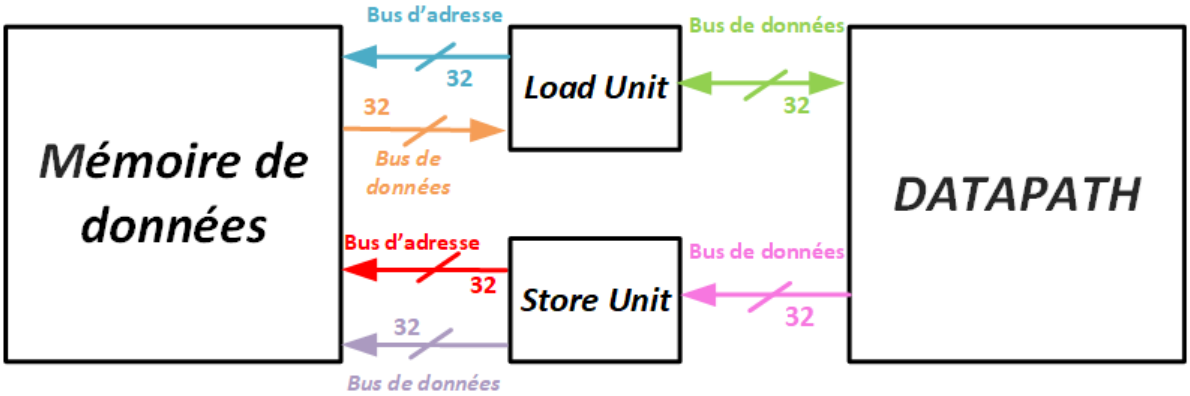

Figure2.4: Partage des données et des addresses

2.2.1.3. Le jeu d'instruction RV32I

RV32I est une architecture de jeu d'instructions conçue spécialement pour les processeurs RISC-V de 32 bits. Elle comporte 47 instructions, mais elles peuvent être réduites à 38, si les instructions SCALL/SBREAK/CSRR* sont regroupées avec une seule instruction SYSTEM, et si l'implémentation du FENCE et FENCE.I en tant que NOP.

Pour notre cas, nous avons implémenté les instructions de type R, les instructions de type I arithmétiques et logiques, de chargement et de saut incondionnel (JALR), les instructions de type B, l'instruction du saut inconditionnel (JAL), et enfin les instructions de stockages.

Comme mentionné précédemment, l'ensemble du jeu d'instruction possède plusieurs catégories d'instructions, notamment des instructions arithmétiques et logiques, de la gestion de la mémoire et aussi celles effectuant des sauts conditionnels et non conditionnels.

Category	Name	Fmt	RV32I Base	
Loads	Load Byte	I	LB	rd,rs1,imm
	Load Halfword	I	LH	rd,rs1,imm
	Load Word	I	LW	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm
	Load Half Unsigned	I	LHU	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm
	Store Halfword	S	SH	rs1,rs2,imm
	Store Word	S	SW	rs1,rs2,imm
Shifts	Shift Left	R	SLL	rd,rs1,rs2
	Shift Left Immediate	I	SLLI	rd,rs1,shamt
	Shift Right	R	SRL	rd,rs1,rs2
	Shift Right Immediate	I	SRLI	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI	rd,rs1,shamt
Arithmetic	ADD	R	ADD	rd,rs1,rs2
	ADD Immediate	I	ADDI	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm
	Add Upper Imm to PC	U	AUIPC	rd,imm
Logical	XOR	R	XOR	rd,rs1,rs2
	XOR Immediate	I	XORI	rd,rs1,imm
	OR	R	OR	rd,rs1,rs2
	OR Immediate	I	ORI	rd,rs1,imm
	AND	R	AND	rd,rs1,rs2
	AND Immediate	I	ANDI	rd,rs1,imm
Compare	Set <	R	SLT	rd,rs1,rs2
	Set < Immediate	I	SLTI	rd,rs1,imm
	Set < Unsigned	R	SLTU	rd,rs1,rs2
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm
Branches	Branch =	SB	BEQ	rs1,rs2,imm
	Branch ≠	SB	BNE	rs1,rs2,imm
	Branch <	SB	BLT	rs1,rs2,imm
	Branch ≥	SB	BGE	rs1,rs2,imm
	Branch < Unsigned	SB	BLTU	rs1,rs2,imm
	Branch ≥ Unsigned	SB	BGEU	rs1,rs2,imm
Jump & Link	J&L	UJ	JAL	rd,imm
	Jump & Link Register	UJ	JALR	rd,rs1,imm

Figure2.5: Liste des instructions du RV32I

2.2.1.4. Formats d'instructions

Les formats d'instructions définissent la structure et le comportement des instructions qui sont exécutées par le processeur, ainsi la manière dont les opérations sont effectuées et les données manipulées.

Le jeu d'instruction RV32I possède un ensemble de 6 formats d'instructions (R/I/S/B/U/J), chaque format indique des catégories spécifiques. Les instructions sont de taille fixe 32 bits et alignées sur une frontière de 4 octets en mémoire, classifiées en six types différents :

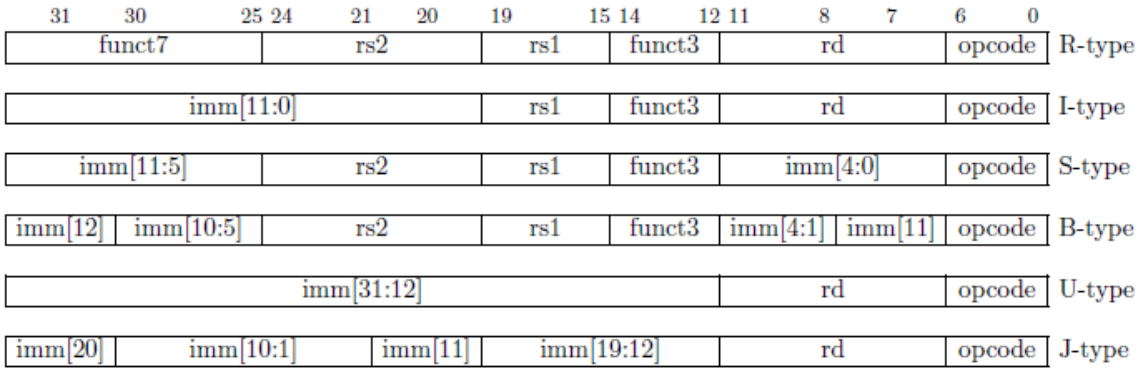

Figure2.6: Les formats d'instructions de base

- Les instructions de **type R** sont de types register-register, effectuent des opérations arithmétiques et logiques avec deux opérandes sources et mettent le résultat dans le registre de destination.
- Les instructions de **type I** sont de types immediate-register, effectuent des opérations arithmétiques et logiques, de chargement et même une instruction de saut inconditionnel en utilisant une opérande source et une valeur immédiate codé sur l'instruction elle-même, et mettent le résultat dans le registre de destination.
- Les instructions de **type S** sont de type Store, des instructions qui stockent une valeur d'un registre dans la mémoire.
- Les instructions de **type B** sont de type Branch, comparent deux registres et calculent une adresse cible pour effectuer des sauts conditionnels.
- Les instructions de **type U** sont de type Upper immediate, utilisées pour le chargement d'adresses immédiates dans les registres qui manipulent des valeurs de 20 bits dans les registres.
- Le **type J** possède une instruction de type Jump, qui effectue un saut inconditionnel après le calcul de l'adresse cible.

Les champs **Opcode**, **funct3** et **funct7** désignent le type de l'instruction, ainsi l'opération à effectuer. Quand on a des instructions qui ont le même Opcode, le champ funct3 différencie les unes des autres, et si besoin on utilise le champs funct7 pour mieux les distinguer. Par exemple les instructions ADD et SUB qui ont le même encodage pour les champs Opcode et funct3, le champ funct7 intervient pour les différencier, la figure 2.3 représente la différence :

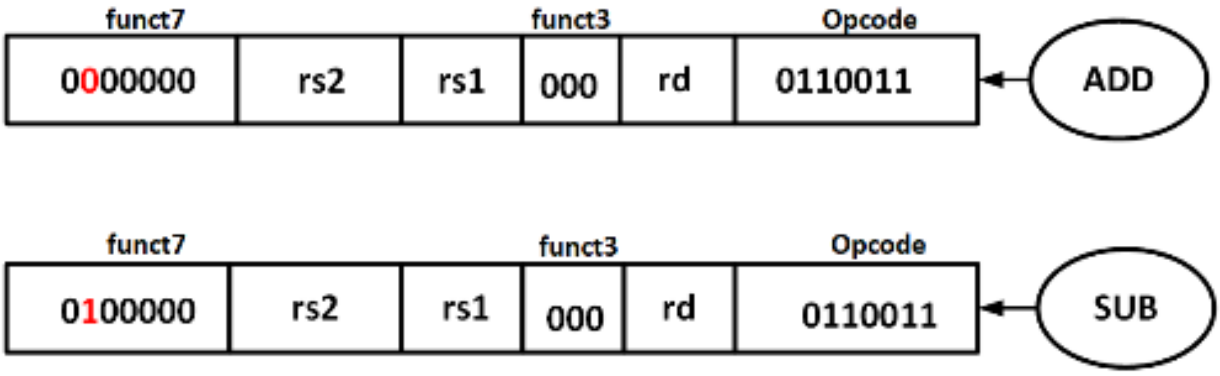

Figure 2.7: les opcodes de l'instruction ADD et SUB

2.2.1.5. L'encodage d'immédiat

Les immédiats résultants par chacun des formats d'instruction de base sont étiquetés pour montrer quel bit d'instruction (inst [y]) produit chaque bit de la valeur immédiate. Cela veut dire que le bit le plus significatif (MSB) de la valeur immédiate est copié dans les bits de poids plus élevé pour avoir l'extension juste. L'extension de signe utilise toujours le bit 31 de l'instruction étiqueté inst [31].

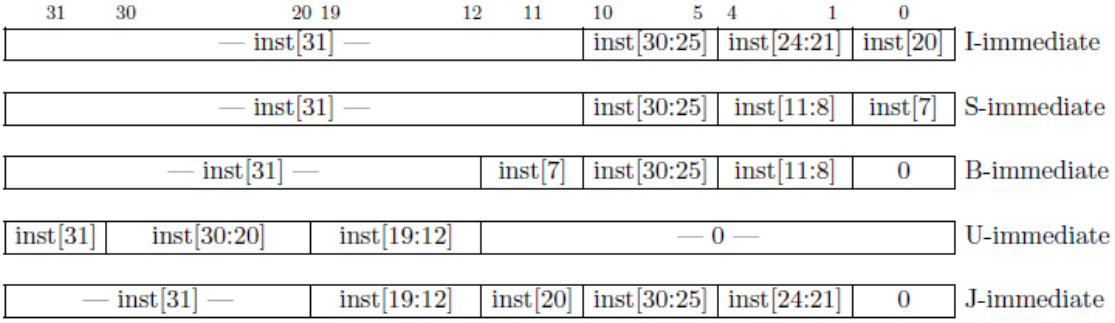

Figure 2.8: Types d'immédiates produites par RISC-V

Type I

La valeur immédiate du type I est placée dans les 12 bits MSB de l'instruction, elle est placée ensuite dans les 12 bits LSB et le bit signe est étendu jusqu'à atteindre 32 bits.

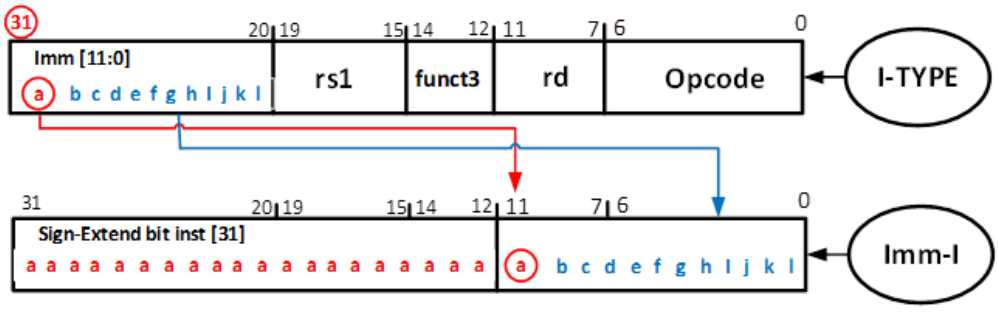

Figure 2.9: Encodages de l'immediat de type I

Type S

Pour avoir la valeur immédiate de type S, le champ de l'instruction [11 :7] est mis en position [4 :0] de la valeur étendue, et le champ de l'instruction [31 :25] est mis en en position [11 :5], et le bit de signe est étendue jusqu'à atteindre 32 bits.

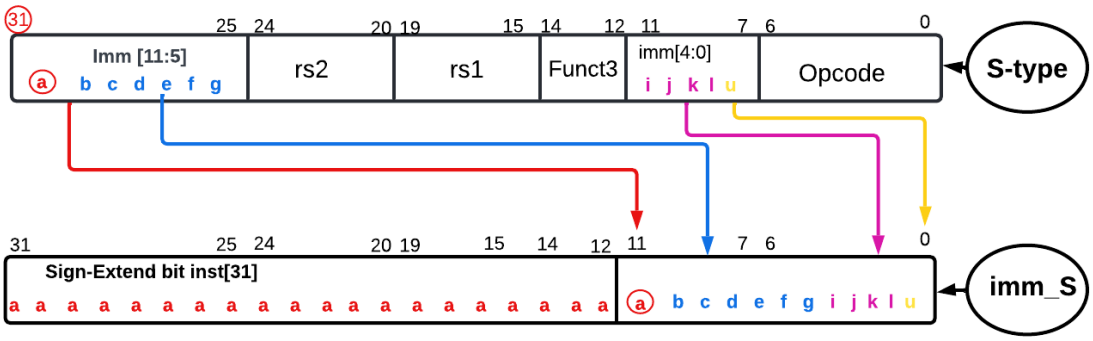

Figure 2.10: Encodages de l'immediat de type S

Type-B

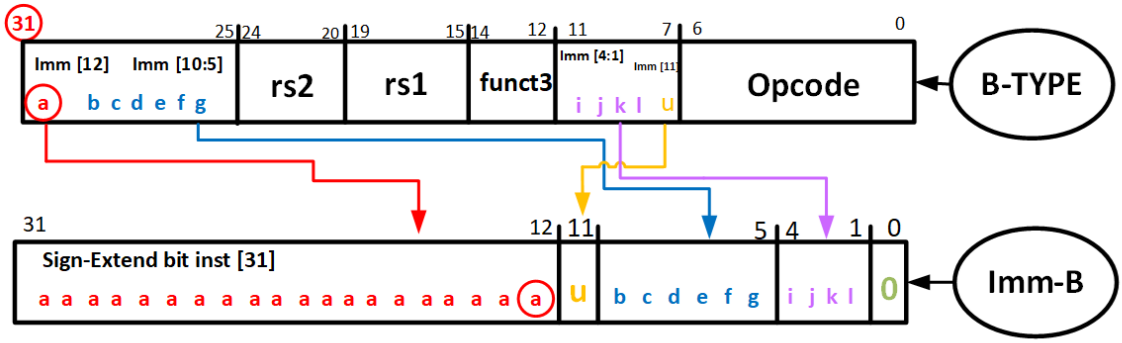

Figure 2.10: Encodages de l'immediat de type B

2.2.1.6. Instructions de calcul

Les instructions de calcul sont des instructions arithmétiques et logiques, effectuent des calculs sur des registres. Elles sont encodées soit en tant qu'opération register- register utilisant le format R ou register- immediats utilisant le format I. Dans les deux cas, le résultat est stocké dans le registre de destination rd.

2.2.1.6.1. Opérations Registre-Registre

RV32I définit plusieurs opérations arithmétiques et logiques de type R. Toutes les opérations lisent les registres rs1 et rs2 en tant qu'opérandes source et écrivent le résultat dans register de destination rd. Les champs funct7 et funct3 sélectionnent le type d'opération.

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Figure 2.11: Instructions arithmetiques et logique de format R

- **ADD** et **SUB** effectuent l'addition et la soustraction respectivement.
- **SLT** et **SLTU** effectuent des comparaisons signées et non signées respectivement, en écrivant 1 à rd si rs1 < rs2, sinon on écrit 0.
- **AND**, **OR** et **XOR** effectuent des opérations logiques bitwise.
- **SLL**, **SRL** et **SRA** effectuent des décalages logiques gauche, logique droite et arithmétique droite respectivement sur la valeur dans le registre rs1 par la quantité de décalage détenue dans les 5 bits inférieurs du registre rs2.
- **SLTU** rd, x0, rs2 définit rd à 1 si rs2 n'est pas égal à zéro, sinon définit rd à zéro (assembleur pseudo-op **SNEZ** rd, rs)

2.2.1.6.2. Opération Registre-Immédiat

RV32I définit également des instructions arithmétiques et logiques de type I, effectuant des opérations avec des registres et des valeurs immédiates encodées directement dans l'instruction elle-même. L'encodage des instructions de ce type comprend généralement deux champs (Opcode et funct3) pour spécifier l'opération, un champ pour spécifier le registre source et un autre pour le registre de destination, ainsi un autre pour spécifier la valeur immédiate. Cette valeur est codée sur 12 bits les plus significatifs (MSB), elle doit être étendue au signe afin d'atteindre la taille de 32 bits pour qu'elle puisse être utiliser.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

Figure 2.12: Instructions arithmetiques et logique de format I

- **ADDI** effectue l'addition d'une valeur immédiate étendue au signe au registre rs1.
- **SLTI** place la valeur 1 dans le registre rd si le registre rs1 est inférieur à la valeur immédiate étendue au signe lorsque les deux sont traités comme des numéros signés, autrement 0 est écrit à rd.
- **SLTU** est similaire mais compare les valeurs comme des nombres non signés, autrement dit, la valeur immédiate est traitée autant qu'une valeur non signée.
- **ANDI**, **ORI**, et **XORI** effectuent des opérations exécutant logiques bitwise **AND**, **OR**, et **XOR** sur le registre rs1 et la valeur immédiate étendue au signe et mettent le résultat dans rd.
- **ADDI** rd, rs1, 0 est utilisé pour implémenter la MV rd, rs1 assembler pseudo-instruction.

- **XORI** rd, rs1, -1 effectue une inversion logique bitwise du registre rs1 (assembleur pseudo-instruction **NOT** rd, rs).
- **SLTIU** rd, rs1, 1 définit rd à 1 si rs1 est égal zéro, sinon définit rd à 0 (assembleur pseudo-op **SEQZ** rd, rs)

Les opérations qui effectuent les décalages par des constantes sont encodées comme des cas particuliers du format. L'opérande à décaler est en rs1, et la quantité de décalage est codée sur les 5 bits inférieurs du champ de l'immédiat. Les bits supérieurs de la valeur immédiate encodent le type de décalage approprié.

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

Figure 2.13: Instructions de decalage logique de format I

- **SLLI** effectue un décalage logique à gauche et les zéros sont déplacés dans les bits inférieurs (LSB).
- **SRLI** effectue un décalage logique à droite les zéros sont déplacés dans les bits supérieurs (MSB).
- **SRAI** est décalage arithmétique à droite et le bit de signe original est copié dans les bits supérieurs évacués

2.2.1.6.3. Opération NOP

L'instruction **NOP** (No Operation) est une instruction qui ne fait rien, elle effectue une opération qui n'a aucun effet réel sauf le PC en utilisant des valeurs qui ne changent pas l'état du processeur. **NOP** est codé comme **ADDI** x0, x0, 0.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	ADDI	0	OP-IMM

Figure 2.14: Instruction Nop

2.2.1.7. Instructions de Transfert de Contrôle

Les instructions de transfert de contrôle dans le RV32I sont disponibles sous forme de sauts inconditionnels et de sauts conditionnels. Les sauts inconditionnels permettent le transfert du contrôle du programme à une nouvelle adresse sans aucune condition préalable, alors que les sauts conditionnels sont effectués uniquement si la condition est satisfaite.

2.2.1.7.1. Sauts inconditionnels

Les deux instructions JAL et JALR composants les instructions à sauts inconditionnels, permettent de transférer le contrôle du programme à une nouvelle adresse sans condition préalable. Elles utilisent l'adressage relatif au PC pour permettre au code de fonctionner correctement, quelle que soit son positionnement en mémoire.

Les instructions JAL et JALR peuvent générer une erreur d'alignement lors la lecture d'instruction si l'adresse cible n'est pas alignée sur une frontière de quatre octets.

Instruction JAL (Jump And Link)

L'instruction JAL utilise le format du type J, ou le champ de la valeur immédiate encode un décalage signé de 20 bits, multiplié par 2 (pour aligner l'adresse sur une limite de 4 octets) et étendue au signe et ajoute au PC pour calculer l'adresse cible du saut. JAL stocke l'adresse de l'instruction après le saut (PC+4) dans le registre rd, généralement standardisé avec x1 comme registre d'adresse de retour et x5 comme registre de lien alternatif.

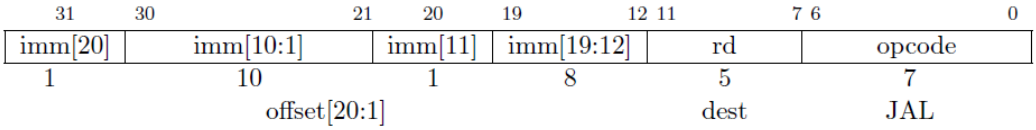

Figure 2.15: Instruction du saut inconditionnel JAL

- Les sauts inconditionnels simples (assembleur pseudo-op **J**) sont codés comme un **JAL** x0, offset.

Instruction JALR (Jump And Link Register)

L'instruction **JALR** utilise l'encodage du type I, l'adresse cible est obtenue en additionnant la valeur immédiate signée de 12 bits étendue au signe au registre rs1, ensuite, on établit au bit le moins significatif du résultat un zéro. L'adresse de l'instruction suivant le saut (PC+4) est écrite dans le registre de destination rd.

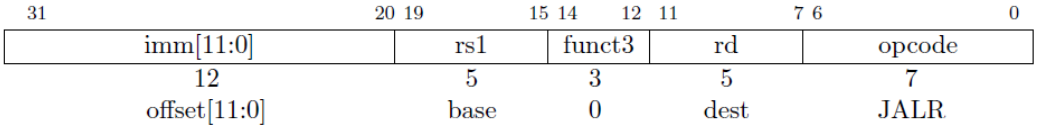

Figure 2.16: Instruction du saut inconditionnel JALR

- Si le résultat n'est pas nécessaire, on peut utiliser le registre x0 comme registre de destination.

2.2.1.7.2. Branchements conditionnels

Les instructions effectuant des sauts conditionnels utilisent le format de type B. la valeur immédiate de 12 bits qui est signée et multipliée par 2 est additionnée PC actuel pour obtenir l'adresse cible. Les instructions de branchement comparent deux registres et en fonction du résultat, le saut serait pris ou pas.

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12,10:5]		src2	src1	BEQ/BNE	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BLT[U]	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BGE[U]	offset[11,4:1]		BRANCH

Figure 2.17: Instructions du saut conditionnel de type B

- **BEQ** et **BNE** prennent le branchement si les registres rs1 et rs2 sont égaux ou non respectivement.
- **BLT** et **BLTU** prennent le branchement si le registre rs1 est inférieur à rs2, en utilisant une comparaison signée et non signée respectivement.
- **BGE** et **BGEU** prennent la branche si le registre rs1 est supérieur ou égale à rs2, en utilisant une comparaison signée et non signée respectivement.
- **BGT**, **BGTU**, **BLE**, et **BLEU** peuvent être obtenues en inversant les opérandes à **BLT**, **BLTU**, **BGE**, et **BGEU**, respectivement.

2.2.1.8. Instructions de chargement et de stockage (Load and Store)

Instructions de chargement (Load)

Les instructions de chargement (Loads) utilisent l'encodage du format I-type. L'adresse effective est obtenue en additionnant le register rs1 à la valeur immédiate de 12 bits étendue au signe. Lorsqu'une instruction de chargement est exécutée, elle lit et charge une valeur depuis la mémoire vers un registre de destination rd. Les champs Opcode et funct3 distinguent l'opération spécifique à effectuer.

RV32I possède 5 instructions de chargement : **LW/LH/LHU/LB/LBU**.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

Figure 2.18: Instructions de chargement de type I

- **LW** charge un mot de 32 bits de la mémoire dans le registre rd.
- **LH** charge une valeur de 16 bits de la mémoire et ensuite étend au signe jusqu'à atteindre 32 bits avant de stocker dans le registre rd.
- **LHU** charge une valeur de 16 bits de la mémoire, mais étend au zéro jusqu'à atteindre 32 bits.
- **LB** et **LBU** chargent des valeurs de 8 bits depuis la mémoire dans le registre rd. Elles sont définies de la même manière que **LH** et **LHU** respectivement.

Instructions de stockages (Store)

Les instructions de stockages (Stores) utilisent l'encodage du format type S. L'adresse effective est obtenue en additionnant le register rs1 à la valeur immédiate de 12 bits et étendue au signe jusqu'elle atteint 32 bits. Lorsqu'une instruction de stockage est exécutée, elle écrit la valeur du

registre rs2 dans la mémoire. Les champs Opcode et funct3 distinguent l'opération spécifique à effectuer.

RV32I comprend trois instructions de stockage: **SW**/**SH**/**SB**

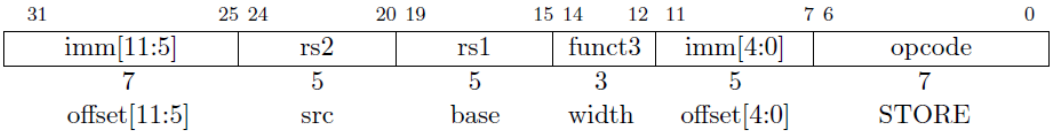

Figure 2.19: Instructions de stockage de type S

- Les instructions **SW**, **SH** et **SB** stockent des valeurs de 32 bits, 16 bits et 8 bits depuis les bits les moins significatifs (LSB) du registre rs2 à la mémoire respectivement.

2.3. Microarchitecture

Notre processeur RV32I se compose de 32 registres à usage général, un registre spécial pour le PC, une mémoire d'instructions et une autre de données.

2.3.1. Registres

Le fichier registre (Register File) est de 32 bits et 32 emplacements, il possède deux ports de lecture et un port d'écriture. Le port de lecture prend une adresse de 5 bits comme entrées adressant $2^5 = 32$ registres. Chacun des deux ports de lecture (read_reg_num1/ read_reg_num2) spécifie une opérande source. Ils lisent les valeurs des registres et mettent leurs données dans les registres read_data1 et read_data2.

Le port d'écriture prend une entrée d'adresse 5 bits (write_reg), le registre possède également une entrée de données d'écriture 32 bits (write_data), une entrée d'activation d'écriture (regwrite) et une horloge. Si regwrite est à 1, le fichier de registre écrit les données dans le registre spécifié sur le front montant de l'horloge.

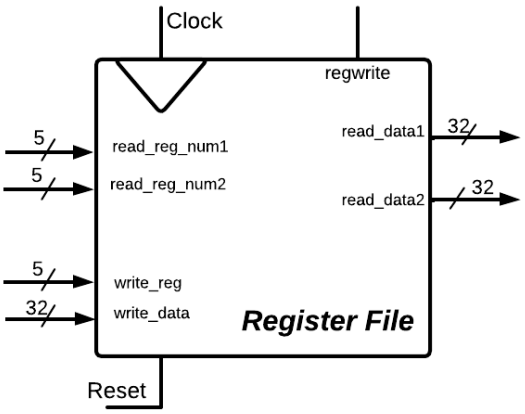

Figure 2.20: Schéma synoptique du registre

2.3.2. Program counter

Le compteur de programme, également connu sous le nom de pointeur d'instruction ou simplement PC, est un composant fondamental de l'unité centrale de traitement (CPU). C'est un registre spécial qui garde l'adresse mémoire de l'instruction suivante à exécuter dans un programme.

Le PC enregistre et pointe vers l'instruction à exécuter par la suite. Le PC ne peut pas être écrit ou lu directement en utilisant les instructions de chargement / stockage. Il ne peut être influencé que par l'exécution d'instructions qui changent le PC comme un saut inconditionnel ou un branchement, où il est mis à jour pour refléter l'adresse cible.

Le CPU utilise le compteur de programme pour suivre son emplacement dans le code, par exemple lors de la récupération de l'instruction suivante. Habituellement, le PC est incrémenté par 4 pendant l'exécution de l'instruction : les adresses sont en octets, de taille 32 bits et chaque instruction de RV32I est de quatre octets de long.

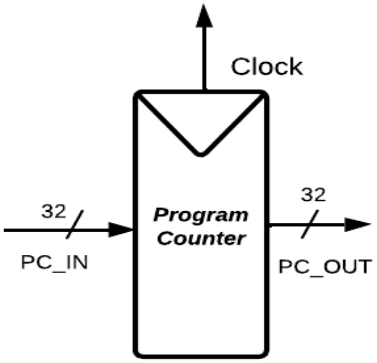

Figure 2.21: Schéma synoptique du compteur de programme

2.3.3. Mémoire d'instruction

La mémoire d'instruction a un seul port de lecture. Elle prend comme entrée l'adresse *A*, et lit la donnée de 32 bits à partir de cette adresse et la met dans le port de sortie.

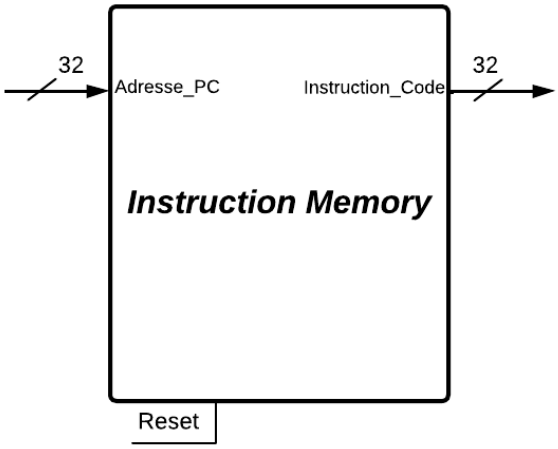

Figure 2.22: Schéma synoptique de la mémoire d'instructions

2.3.4. Mémoire de données

La mémoire de données à un seul port de lecture / écriture. Si l'activation d'écriture (WE) est 1, il écrit des données (WD) dans l'adresse A sur le front montant de l'horloge. Si l'activation d'écriture est 0, il lit l'adresse A sur RD.

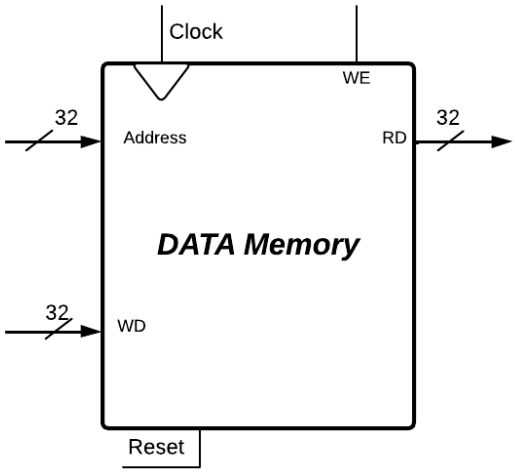

Figure 2.23: Schéma synoptique de la mémoire de données

2.4. Etages d'exécution d'une instruction

L'exécution d'une instruction se fait généralement en 5 étapes distinctes (Fetch, Decode, Execute, Memory Access et Write Back), nous allons examiner ce qui se passe dans chaque étape. La figure 2.23 :

Fetch Instruction	Decode Read Reg	Execute ALU	Memory Read/Write	Write Reg

Figure 2.23: Etages d'exécution d'une instruction

Cycle de lecture (Fetch)

La première étape d'exécution d'une instruction quel que soit son type est le cycle de lecture ou instruction fetch, l'instruction est récupérée grâce à l'adresse indiquée par le compteur de programme (PC), il est ensuite incrémenté par 4 pour pointer vers l'instruction suivante en mémoire. C'est ainsi qu'on charge l'instruction en cours depuis la mémoire d'instructions dans un registre pour pouvoir effectuer les opérations nécessaires.

Cycle de décodage

La deuxième étape d'exécution d'une instruction est le décodage, une étape essentielle pour définir quelles sont les opérandes nécessaires et les opérations spécifiques à effectuer.

Le cycle de décodage collecte les données à partir des champs Opcode, les registres rs1, rs2 et rd. Elle lit l'opcode pour déterminer le type d'instruction et extraire les données des registres. Par exemple si on a une instruction de type R, elle lit deux registres sources, si on a une instruction de type arithmétique I, elle lit un registre source et la deuxieme opérande comme une valeur immédiate, et si on a une instruction JAL, on aura aucune lecture de registre.

Cycle d'exécution

Le cycle d'exécution est la troisième étape, où l'unité arithmétique et logique (ALU) ou une autre unité de traitement effectue les opérations correspondantes. À ce stade, les opérations arithmétiques et logiques, les branchements, les sauts inconditionnels ainsi que les instructions de chargement et de stockage sont exécutés. Les opérandes sont traités et le résultat est produit.

Cycle de l'accès à la mémoire

La quatrième étape est l'accès à la mémoire où on utilise que les instructions Load/Store, cette étape gère la lecture ou l'écriture des données si nécessaire, l'adresse mémoire est calculée, et l'opération de lecture ou d'écriture est exécutée.

Cycle d'écriture (Write Back)

Le cycle d'écriture est la dernière étape d'exécution d'une instruction, où le résultat des opérations de calcules ou de stockages est écrit dans le registre de destination spécifié par l'instruction. Cela peut comporter des résultats de calculs arithmétiques, des valeurs lues de la mémoire ou des adresses de retour pour les instructions de saut. Cette étape assure que les résultats des opérations sont disponibles pour les instructions suivantes.

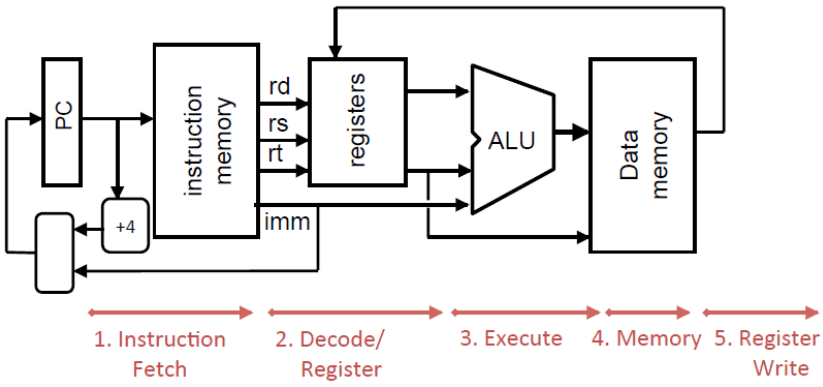

Figure 2.24: 5 Etages d'exécution d'une instruction dans un Datapath

2.5. Signaux de contrôle

Les signaux de contrôle sont des signaux qui contrôlent et maîtrisent le flux d'exécution des instructions. Ils sont essentiels pour assurer le bon fonctionnement d'un processeur, produit par

l'unité de contrôle pour déterminer les comportements attendus des composants. Ils assurent également la coordination des différentes étapes d'exécution des instructions.

L'unité de contrôle génère des signaux de contrôle en fonction du type d'instruction actuelle pour sélectionner les composants à utiliser et exploiter les fonctionnalités nécessaires. Voici un aperçu sur l'ensemble des types de signaux :

- **Signaux de contrôle de l'ALU : indiquent l'opération à effectuer.**
- **Signaux de sélection de multiplexeur (MUX) : utilisés pour sélectionner la bonne sortie du multiplexeur.**
- **Signaux d'activation des modules (Enable) : utilisés pour activer ou désactiver un module.**

D'autres signaux peuvent être introduits en fonction des besoins spécifiques de chaque architecture, ou fonctionnalités supplémentaires intégrées tels que les signaux de contrôle du registre, la prédiction de branchement, ou encore la gestion de mémoire.

2.6. Conception du RV32I

La conception de l'architecture RV32I se compose de trois unités principales : l'unité de récupération des instructions (instruction Fetch Unit), le chemin de données (Datapath) et l'unité de contrôle (Control Unit). La figure montre les interconnexions entre ces trois composants :

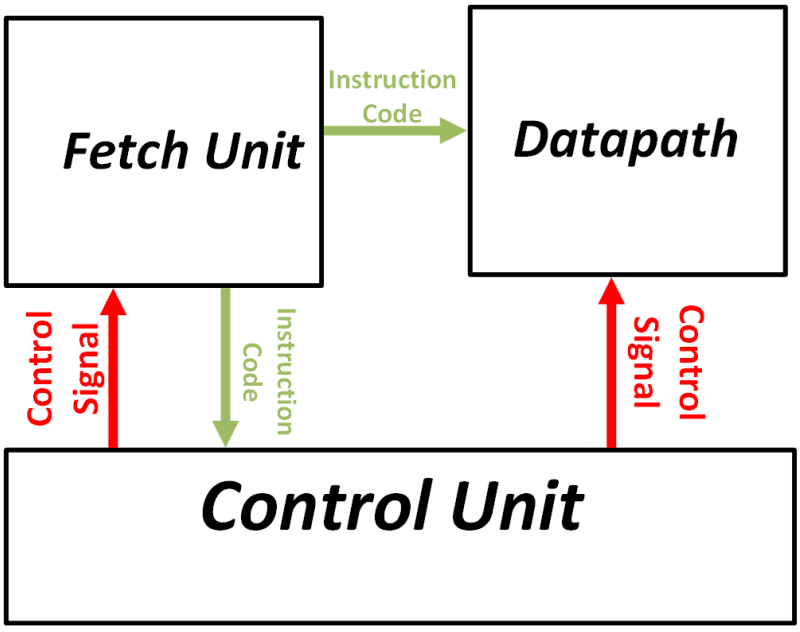

Figure 2.25: Schéma synoptique des interconnexions entre les elements du processeur

2.6.1. Instruction fetch unit

Le cycle de lecture est généralement le même pour tous les types d'instructions, puisque la longueur fixe de 32 bits qui permet une récupération uniforme des instructions à partir de la mémoire d'instructions.

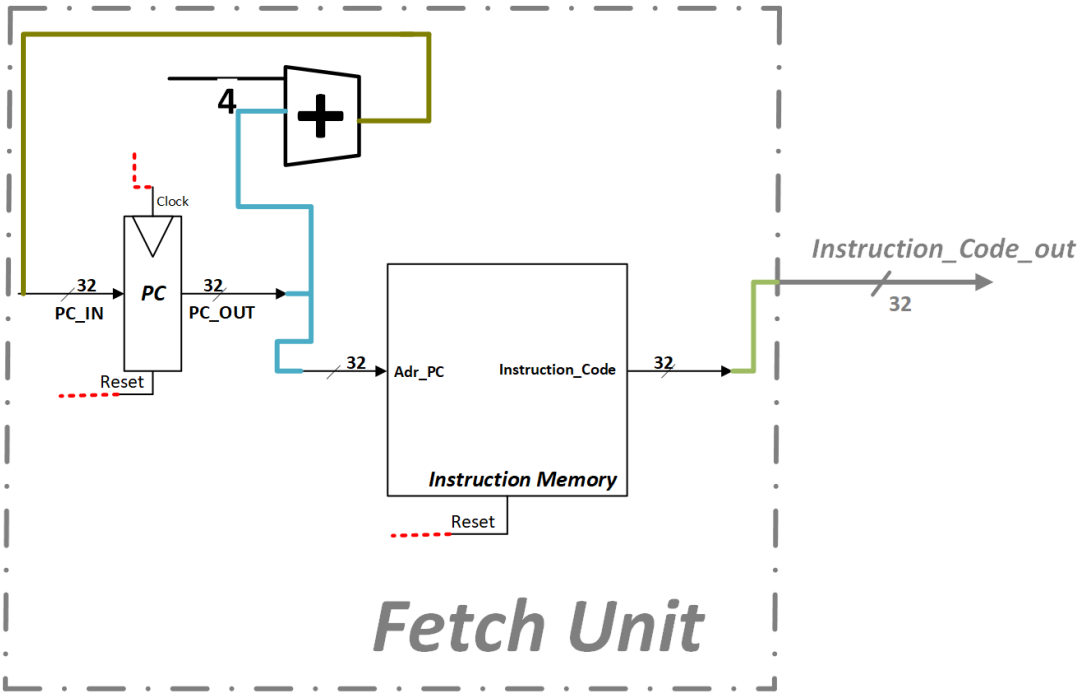

Figure 2.26: Datapath du cycle de lecture

2.6.2. Conception du datapath RV32I

Un datapath est un ensemble d'unités fonctionnelles telles que l'unité logique arithmétique (ALU), des registres, des additionneurs qui effectuent des opérations de traitement de données, des registres et des bus. Avec l'unité de contrôle, il compose l'unité centrale de traitement (CPU).

Dans cette section, Notre premier objectif dans ce projet était de concevoir l'architecture hardware capable d'exécuter les instructions du jeu d'instructions RV32I Base Integer. Les instructions incluses dans cet ensemble sont présentées à la figure 2.1.

Nous allons à présent concevoir un datapath pour chaque type d'instruction commençant par le format-R.

2.6.2.1. Instructions de type format R

Une instruction de type R effectue une opération arithmétique ou logique en lisant deux registres rs1 et rs2 comme opérandes sources et écrivant le résultat dans le registre de destination rd. Les champs funct3 et funct7 sélectionnent le type d'opération.

Par exemple, l'instruction « ADD rd, rs1, rs2 » est censée lire deux registres, les additionner et finalement écrire le résultat dans un registre de destination rd.

La figure 2.24 montre le datapath des instructions de type R :

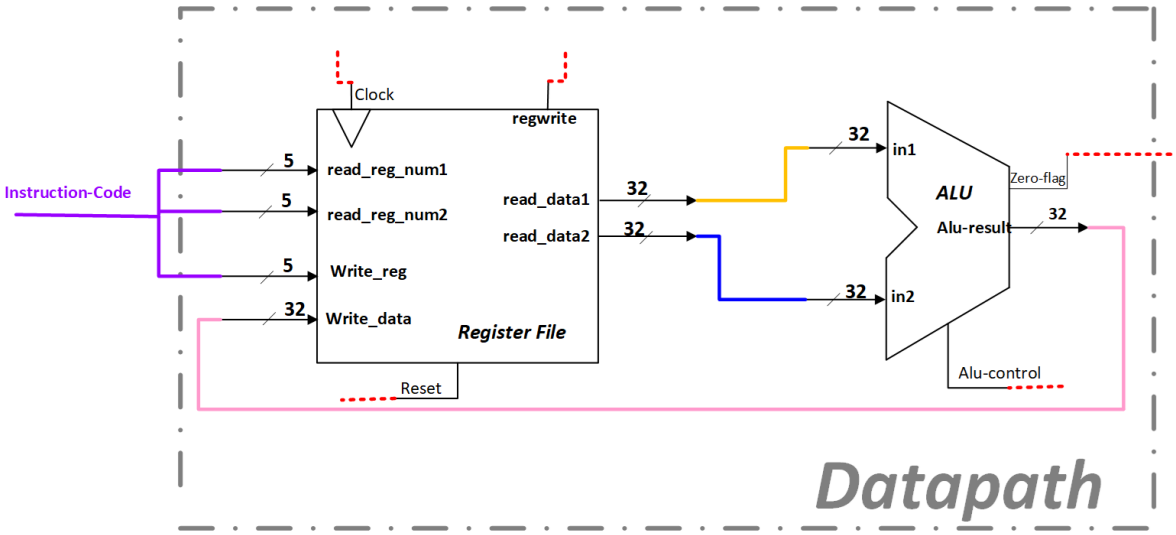

Figure 2.27: Datapath des instructions de type R

2.6.2.2. Instructions de type format I

Le format I fournit plusieurs catégories d'instructions. Dans notre conception, nous avons conçu un datapath pour chaque catégorie, un pour les instructions arithmétiques et logique, un autre pour les instructions de chargements et enfin un datapath pour l'instruction du saut inconditionnel JALR.

2.6.2.3. Instructions arithmétiques et logiques

Les instructions arithmétiques et logiques de type I lit un opérande source et une opérande immédiate 12 bits étendu au signe, effectue l'opération correspondante et écrivent le résultat dans le registre de destination.

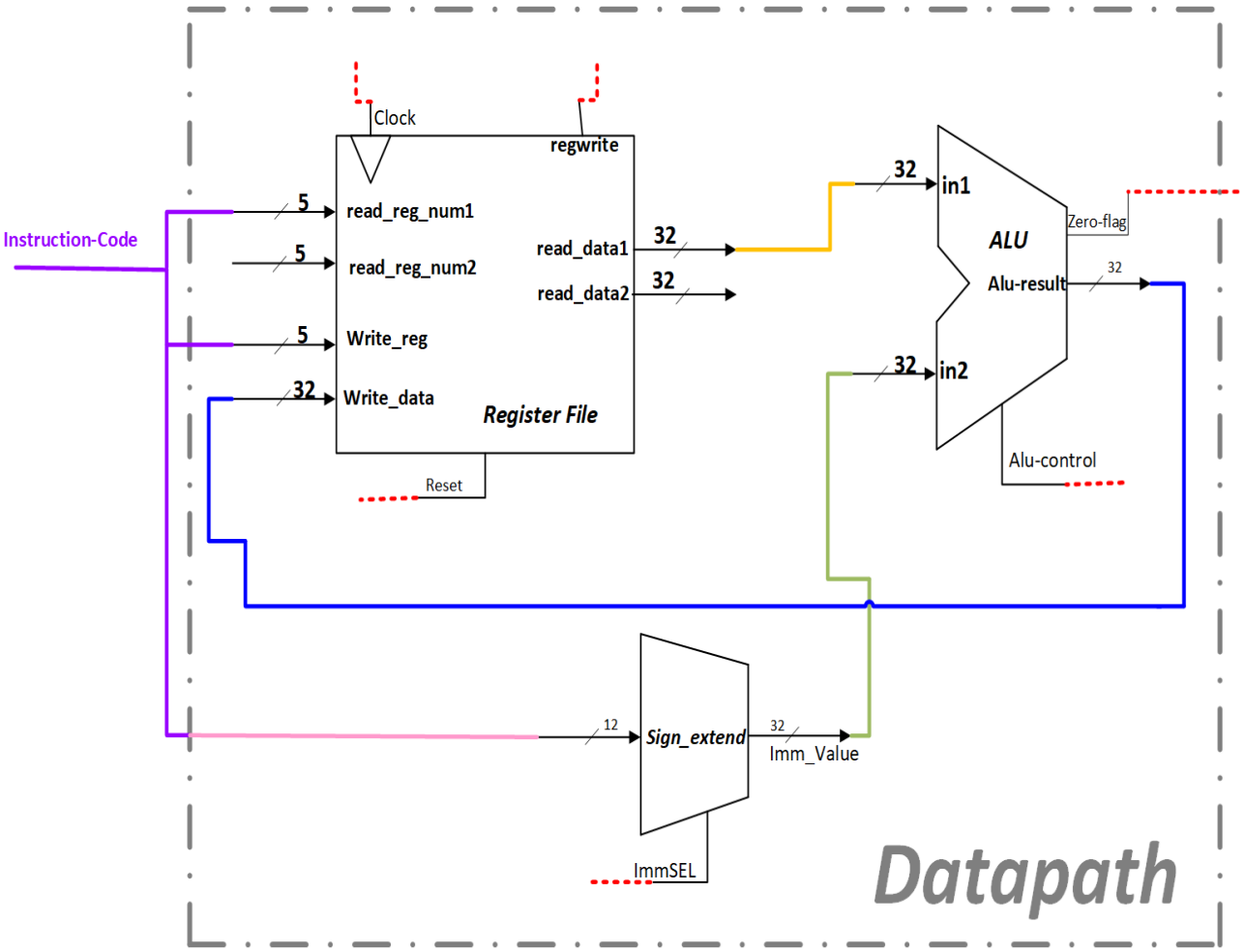

Figure 2.28: Datapath des instructions de type I arithmetiques et logiques

2.6.2.4. Instructions Load

Les instructions de chargement sont de type I.

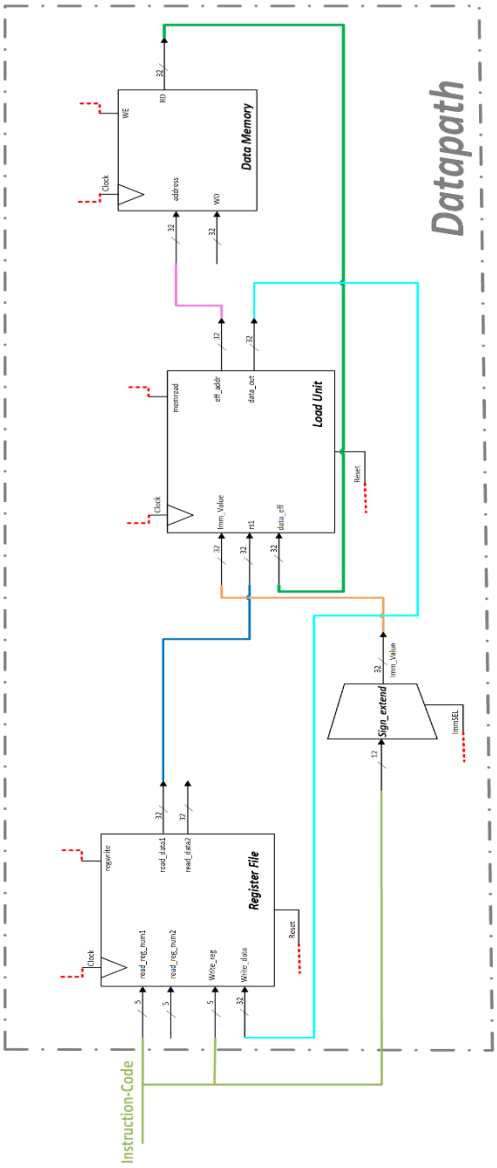

Figure 2.29: Datapath des instructions de chargement de type I

2.6.2.5. Instruction JALR

L'instruction JALR permet de réaliser un saut inconditionnel a une adresse calculée et sauvegardée dans le registre de destination rd déjà spécifié dans l'instruction. Elle reçoit comme entrée l'adresse du PC courante et un décalage (offset) étendu en signe, et calcule la nouvelle adresse en additionnant PC et Offset, résultant une nouvelle adresse du PC. L'adresse du retour PC+4 est ensuite sauvegardée dans le registre rd.

Nous avons conçu une unité qui traite cette instruction indépendamment nomme Jump Register Unit.

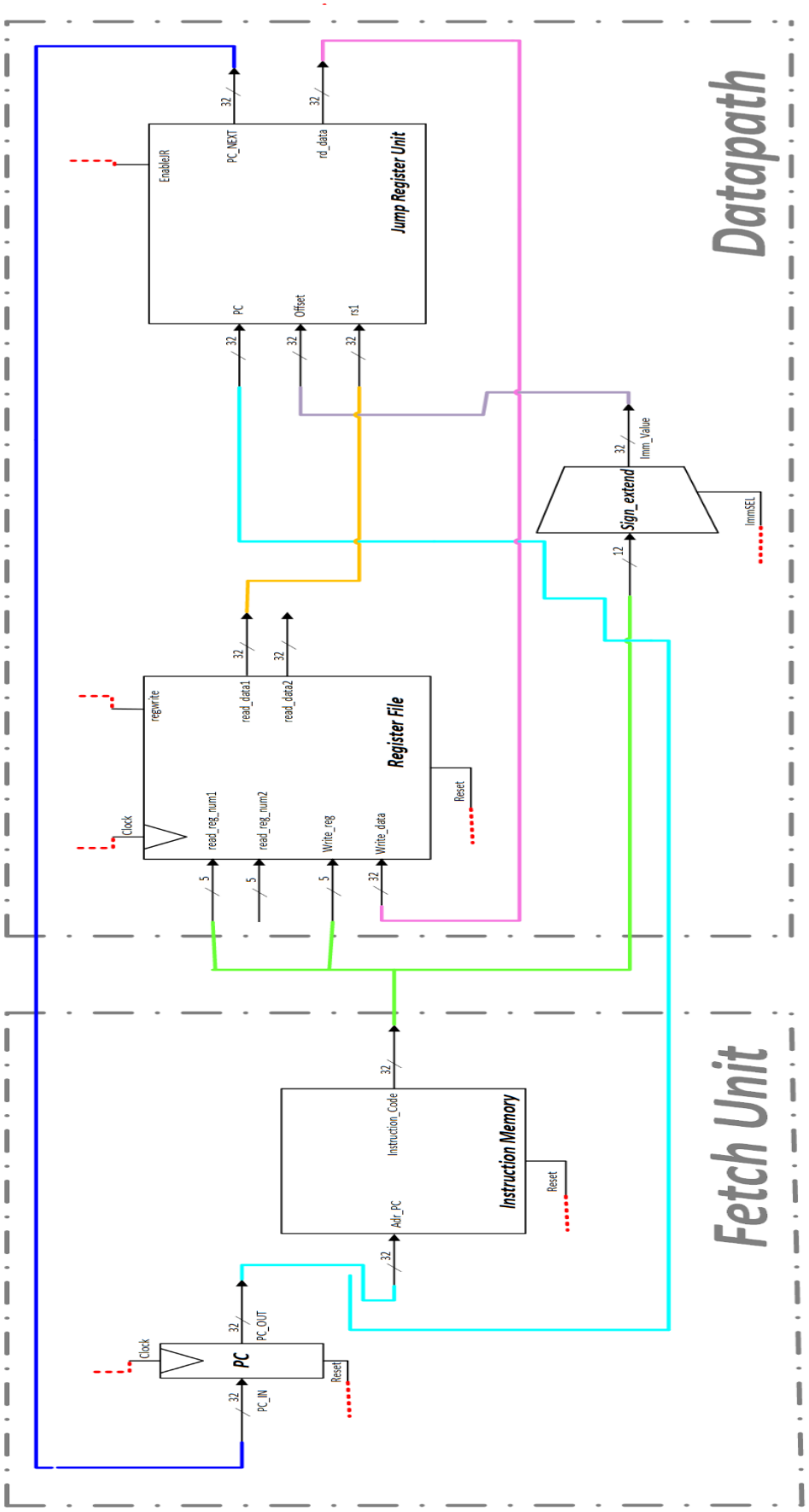

Figure 2.30: Datapath des instructions de saut inconditionnel de type I

2.6.2.6. Instructions de type format S

Les instructions de type format S sont des instructions de stockage, utilisées pour stocker des valeurs depuis un registre vers la mémoire. L'adresse de mémoire cible est calculée en ajoutant un décalage signé, spécifié dans l'instruction et étendu au signe, à la valeur du registre rs1.

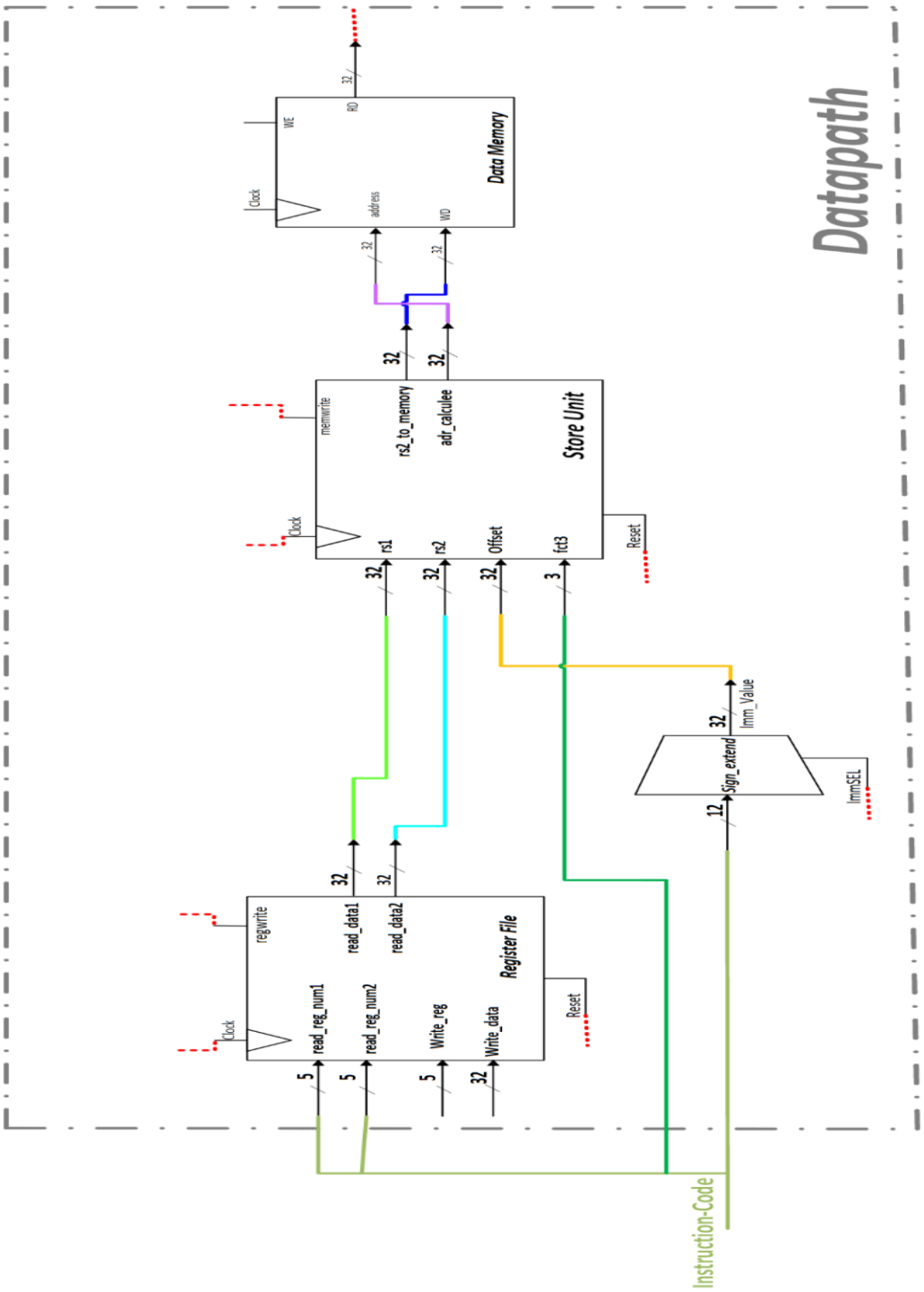

Figure 2.31: Datapath des instructions de stockage de type S

2.6.2.7. Instructions de type format B

Les instructions de branchement comparent deux registres sources rs1 et rs2, et un décalage qui est étendu au signe est ajouté au compteur de programme (PC) pour calculer l'adresse cible du saut.

BEQ et BNE prennent le branchement si les registres rs1 et rs2 sont respectivement égaux ou différents. BLT et BLTU prennent le branchement si rs1 est inférieur à rs2, en utilisant respectivement une comparaison signée et non signée. BGE et BGEU prennent le branchement si rs1 est supérieur ou égal à rs2, en utilisant respectivement une comparaison signée et non signée.

Pour concevoir une unité de branchement qui fonctionne correct, nous avons choisi de concevoir trois modules chacun avec une tache spécifique un comparateur, un conditionneur et un calculateur d'adresse.

Commençant par le comparateur qui doit effectuer la comparaison de deux valeurs entrantes chacune 32 bits, il résulte trois signaux indiquant la nature de la comparaison, Br-EQ pour une comparaison d'égalité, Br-LT pour une comparaison d'infériorité, et en dernier BR-UN pour une comparaison de deux valeurs non signées.

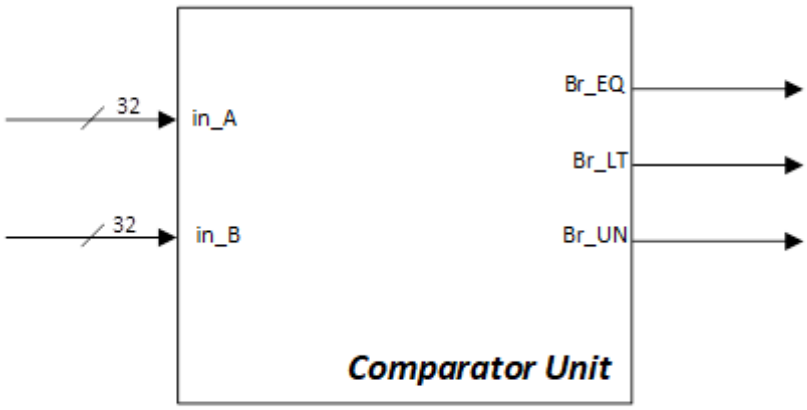

Figure 2.32: Schéma synoptique du comparateur

Le deuxième module est un conditionneur. Il reçoit comme entrées les signaux provenant du comparateur et, d'après la fonction "funct3" de l'instruction, spécifie le type de comparaison (égalité, supériorité ou infériorité). Par exemple, nous avons l'instruction d'égalité BEQ qui compare deux valeurs. Si le résultat de la comparaison est vrai, le conditionneur attribue la valeur 1 à la sortie ; sinon, il attribue la valeur 0.

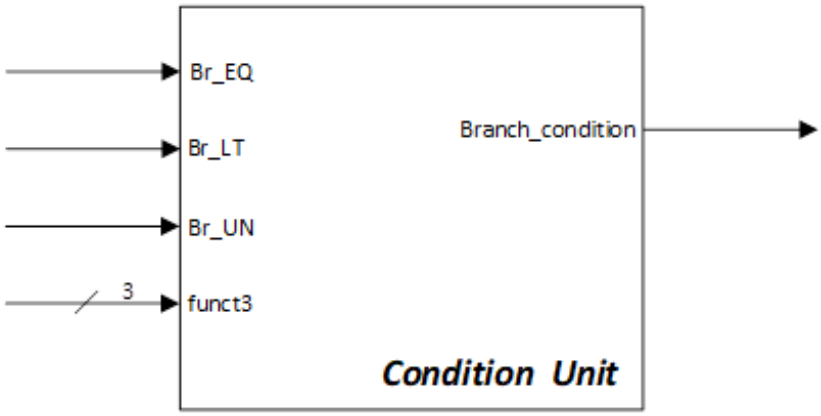

Figure 2.33: Schéma synoptique du conditionneur

Le troisième module est un calculateur d'adresse cible. Il prend comme entrées le signal du conditionneur. Si ce signal est vrai, le calculateur d'adresse cible calcule l'adresse cible en additionnant la valeur actuelle du PC et le décalage « Offset » étendu en signe pour obtenir la valeur correcte de l'adresse cible. Enfin, il génère un signal indiquant si le branchement est pris ou non.

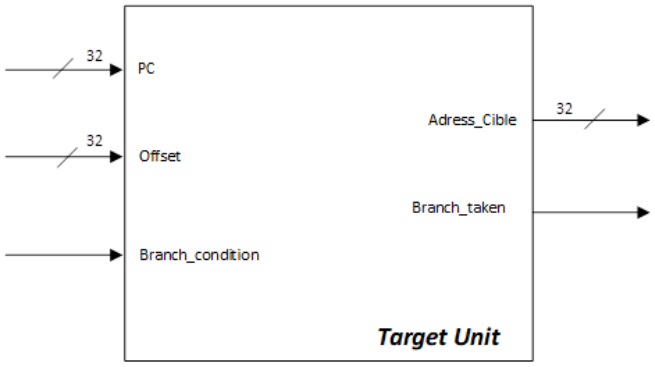

Figure 2.34: Schéma synoptique du calculateur d'adresse cible

Pour assembler l'unité de branchement, nous devons interconnecter les trois modules : comparateur, conditionneur et calculateur d'adresse cible. Les entrées in_A et in_B de l'unité de branchement seront les entrées du comparateur. L'entrée funct3 est connectée à celle du conditionneur. Le PC et l'Offset sont connectés aux entrées du calculateur d'adresse cible. Les signaux de sortie du comparateur sont connectés aux entrées du conditionneur. Enfin, les sorties de l'unité de branchement seront les résultats calculés par le calculateur d'adresse cible.

La figure 2.32 montre toutes les interconnexions :

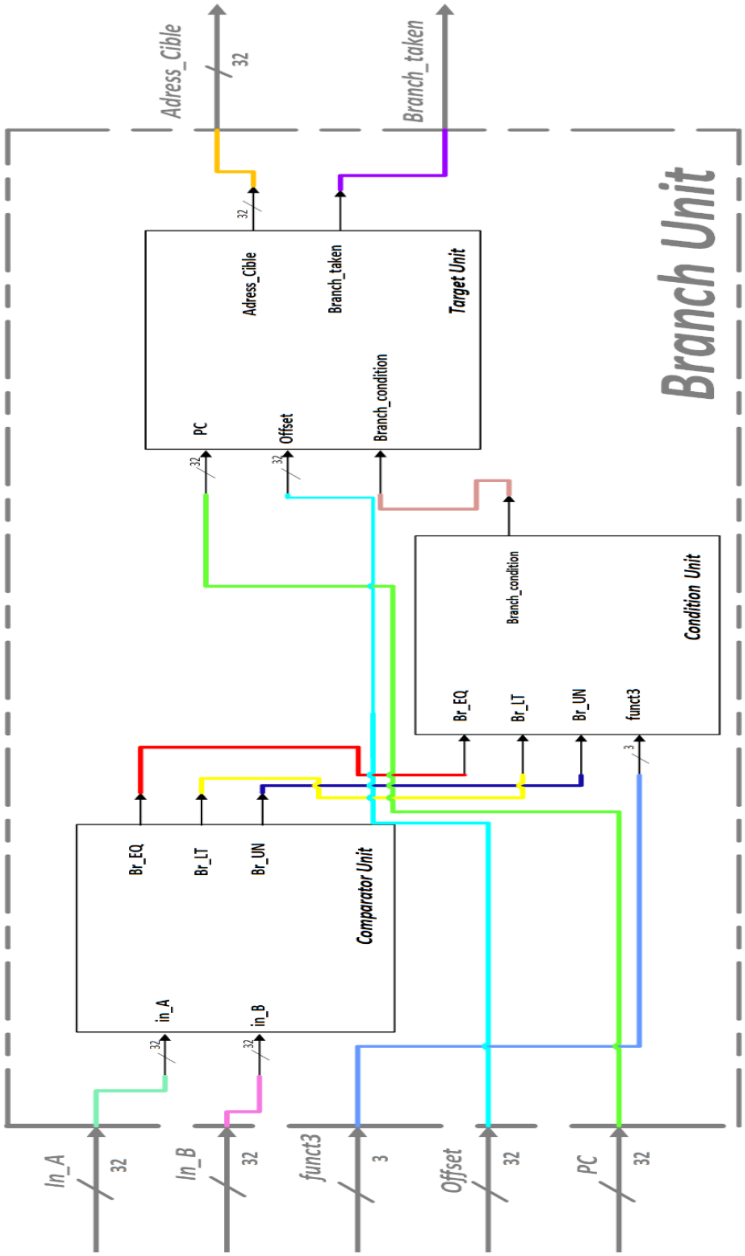

Figure 2.35: Schéma synoptique de l'unité de branchement

Le datapath des instructions de branchement est présenté dans la figure si dessous, les valeurs à comparer viennent du registre, le décalage est étendu au signe par le générateur d'i immédiat.

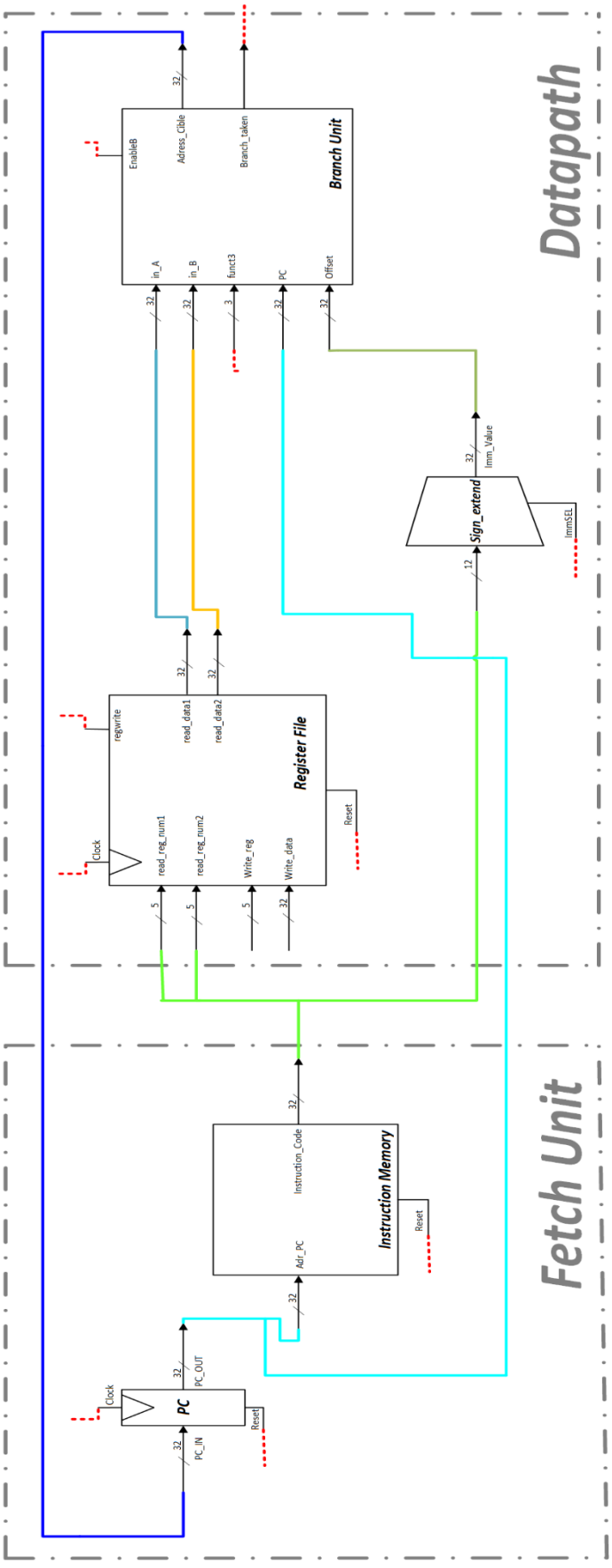

Figure 2.36: Schéma synoptique des instructions de type B

2.6.2.8. Instructions de type format J

Le format de type J contient une seule instruction JAL « jump and link », dans lequel l'immédiat J encode un décalage signé en multiples de 2 octets. Ce décalage est étendu au signe et ajouté au compteur de programme (PC) pour former l'adresse cible du saut.

L'instruction JAL rd, offset calcule l'adresse cible en additionnant l'offset avec le PC et stocker stocke l'adresse de l'instruction suivante (PC+4) dans le registre rd.

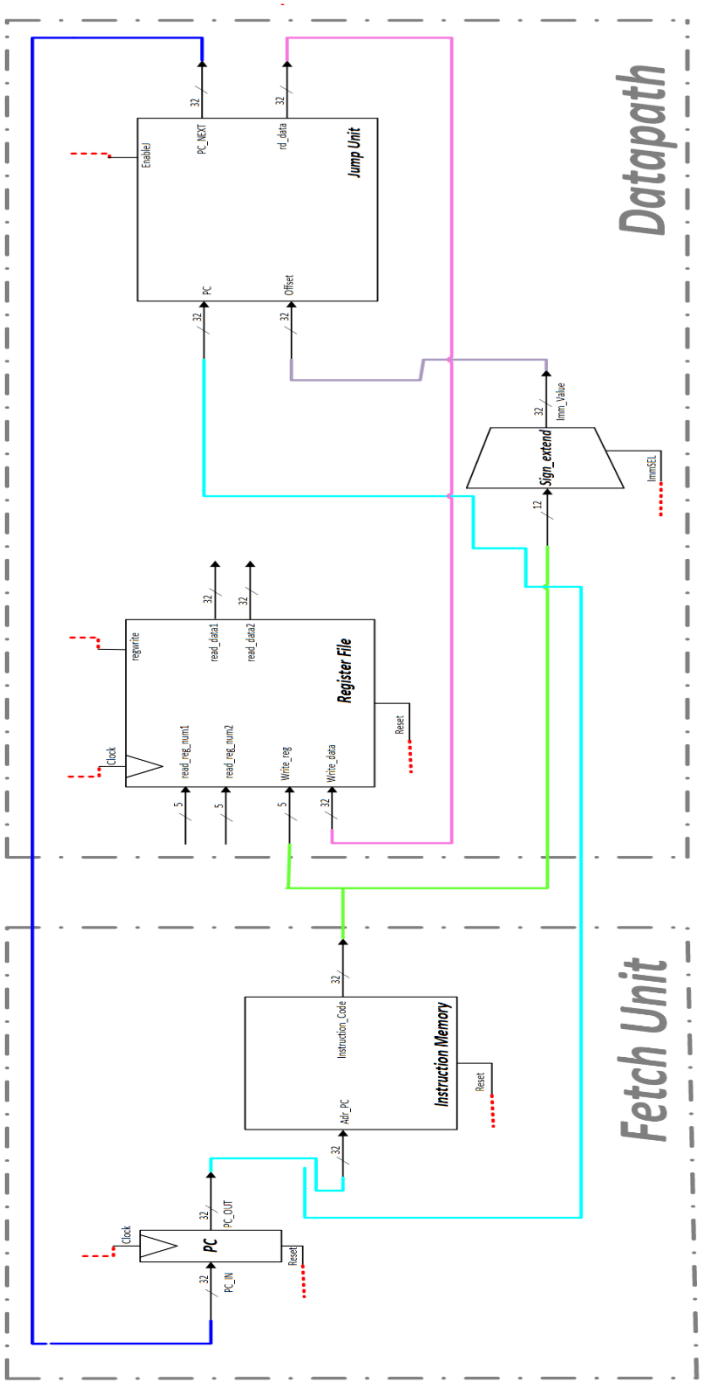

Figure 2.37: Schéma synoptique d'instructions de saut de type J

2.6.2.9. Unité de contrôle

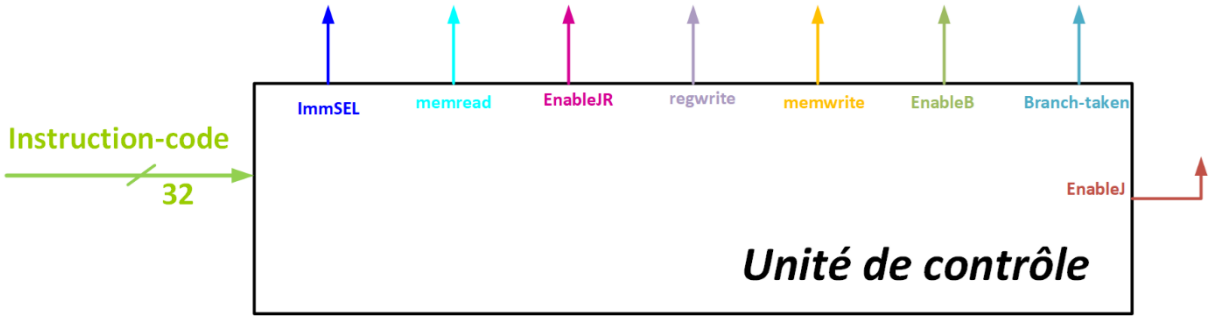

Figure 2.38: Schéma synoptique de l'unité de contrôle

2.7. Datapath global

Une fois que nous avons déterminé les datapaths pour toutes les instructions, nous les avons combinés pour former le datapath complet. Nous avons également ajouté des multiplexeurs si nécessaires. La figure 2.4 montre le datapath global du jeu d'instructions RV32I.

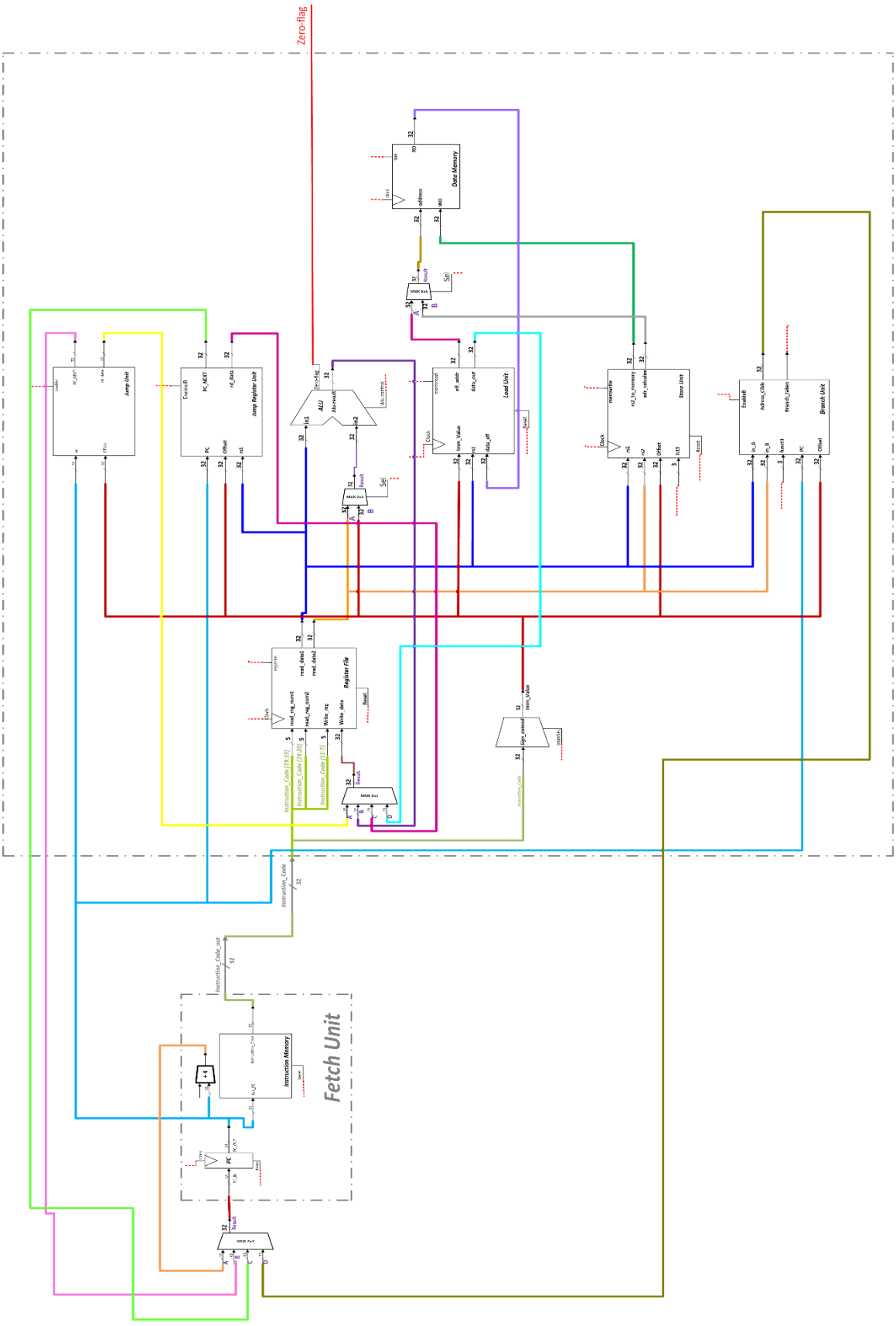

Figure 2.49: Schéma synoptique de Datapath global

L’entrée de l’ALU “in2” peut etre une entrée pour les instructions de type R et de type I arithmetiaues et logiques, ce qui signifie que nous devons ajouter un multiplexeur a cette entrees pour que l’ALU ait les deux types de donnees regitre et immediate.
L’entrée du registre Write-data nessecite un multiplexeur car l’nstruction JALR, JAL, les instructions de chargements et l’ALU font tous le besion d’utiliser cette entrée.
Un autre multiplexeur se trouve a l’entrée du PC, car les instructions de sauts conditionnels et non conditionnels renvoient des addresses cibles.

2.8. Conclusion

Dans ce chapitre, nous avons présenté la conception de l’architecture du jeu d'instructions RV32I, en créant un datapath pour chaque type d'instruction, puis en les assemblant dans un seul datapath global. Ensuite, nous avons introduit l’unité de contrôle, qui est responsable de générer des signaux de contrôle permettant la gestion totale des composants. Le chapitre 3 portera sur l’implémentation de ces datapaths en langage Verilog.

Chapitre 3

Implémentation du processeur et discussion des résultats

3.1. Introduction

Dans ce chapitre, nous présentons le processus de conception et de simulation de notre architecture proposée. Nous aborderons les outils et les méthodologies utilisés pour développer, simuler et vérifier la fonctionnalité du processeur. Ce chapitre vise à fournir une compréhension des différentes étapes de la conception, depuis la description RTL (Register Transfer Level) jusqu'àla validation des performances à travers des simulations post-synthèse.

3.2. Environnements de travail

Nous avons utilisé dans notre projet plusieurs environnements de travail afin d'avoir exploiter les fonctionnalités et les avantages de chaque un, nous avons essayé de compléter l'un avec l'autre, Vivado IDE pour la vérification de syntaxe et le « Elaborated Design », Icarus Verilog pour les tests bench et GTKwave pour visualiser et analyser les formes d'onde issues des simulations.

3.2.1. Vivado IDE

Vivado Design Suite est une suite logicielle avancée pour la synthèse et l'analyse de conceptions en langage de description de matériel (HDL), principalement Verilog et VHDL. Remplaçant Xilinx ISE avec des fonctionnalités supplémentaires pour le développement de systèmes sur puce et la synthèse de haut niveau. Introduit en avril 2012, Vivado offre un environnement de conception intégré (IDE) doté d'outils de niveau système à circuit intégré, construits sur un modèle de données évolutif partagé et un environnement de débogage commun. [16]

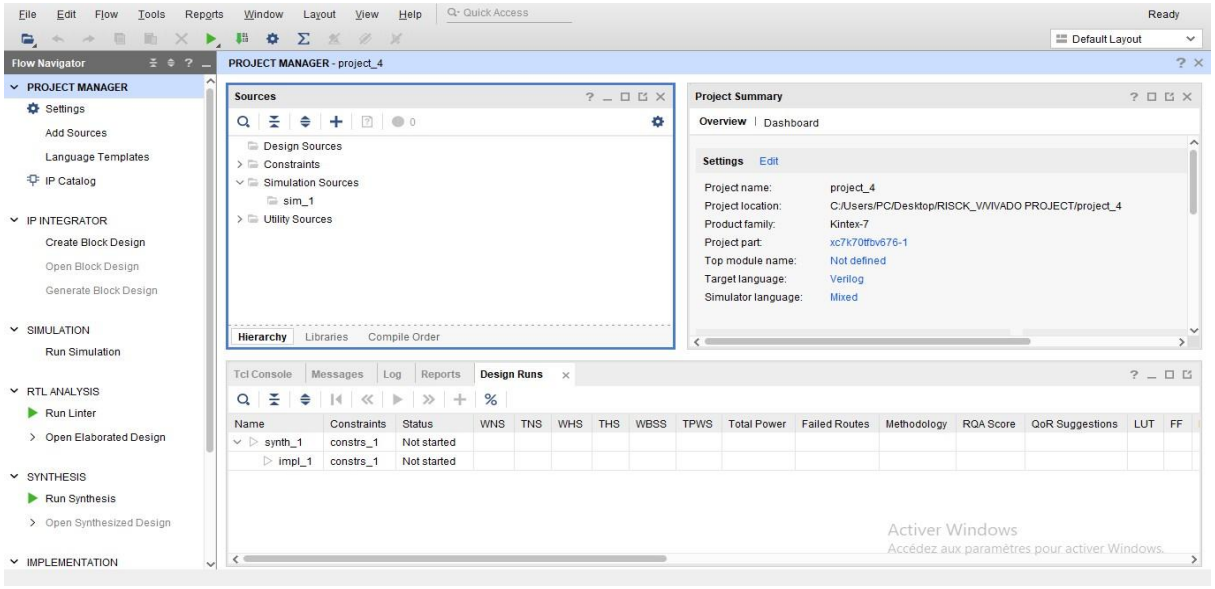

Figure3.1 : IDE Vivado

Dans le contexte de notre projet, Vivado a été utilisé pour visualiser les designs élaborés au niveau RTL (Register Transfer Level). Cela inclut la création et la vérification de descriptions détaillées des circuits numériques à l'aide de langages de description matérielle « verilog ».

3.2.2. Icarus Verilog

Figure 3.2: Logo icarus

Icarus Verilog est un outil de simulation et de synthèse de circuits numériques utilisé dans le domaine de la conception de circuits intégrés numériques. Il permet de créer, de simuler et de vérifier des designs en langage Verilog HDL (Hardware Description Language). C'est un outil open-source largement utilisé pour le développement et la vérification de matériels électroniques. [17]

Nous avons utilisé Icarus verilog pour simuler les tests benchs des codes, et pour avoir des résultats lisibles. En utilisant les deux commandes suivantes :

```
C:\iverilog\bin>iverilog.exe -o dsn ALU_UNIT.v ALU_UNIT_tb.v
```

Cette commande est utilisée pour compiler les fichiers verilog, « ALU_UNIT.v » qui est le code de l'unité arithmétique et logique ainsi « ALU_UNIT_tb.v » le code de test bench de cette unité à l'aide d'icarus verilog .

-o dsn ou n'importe quelle autre appellation est utilisée pour spécifie le nom du fichier de sortie après la compilation.

```
C:\iverilog\bin>vvp.exe dsn
```

Après avoir compiler les fichiers Verilog Cette commande est utilisée pour exécuter cette simulation et observer le comportement du code dans un environnement de simulation numérique. La figure 3.5 illustre un exemple avec le code verilog de l'unité arithmétique et logique.

```
C:\iverilog\bin>iverilog.exe -o dsn ALU_UNIT.v ALU_UNIT_tb.v

C:\iverilog\bin>vvp.exe dsn
Test case 1 (AND): Result = xxxxxxxx, Zero flag = 0
Test case 2 (OR): Result = ffffffff, Zero flag = 0
Test case 3 (ADD): Result = 80000000, Zero flag = 0
Test case 4 (SUB): Result = 7fffffff, Zero flag = 0
Test case 5 (SLT): Result = 00000000, Zero flag = 1
Test case 6 (SLL): Result = 00000000, Zero flag = 1
Test case 7 (SRL): Result = 04000000, Zero flag = 0
Test case 8 (XOR): Result = ffffffff, Zero flag = 0
Test case 9 (Zero flag): Result = ffffffff, Zero flag = 0
Test case 10 (Non-zero flag): Result = ffffffff, Zero flag = 0

C:\iverilog\bin>
```

Figure 3.3 : Exemple de icarus avec l'unité arithmétique et logique

3.2.3. GTKwave

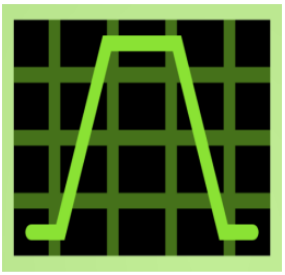

Figure3.4 : Logo GTKwave

GTKWave est un outil et destiné à l'analyse des formes d'onde générées par les simulations de circuits numériques. Il prend en charge de nombreux formats de fichiers de traces et offre une interface utilisateur graphique pour visualiser et explorer les données. Dans notre projet, GTKWave a été utilisé pour inspecter et analyser les formes d'onde résultant des simulations afin de vérifier le comportement attendu des signaux et identifier les éventuelles erreurs. Grâce à ses fonctionnalités de zoom, de marquage et de mesure précise des temps, nous avons pu examiner en détail les transitions des signaux et les relations temporelles entre eux.[18]

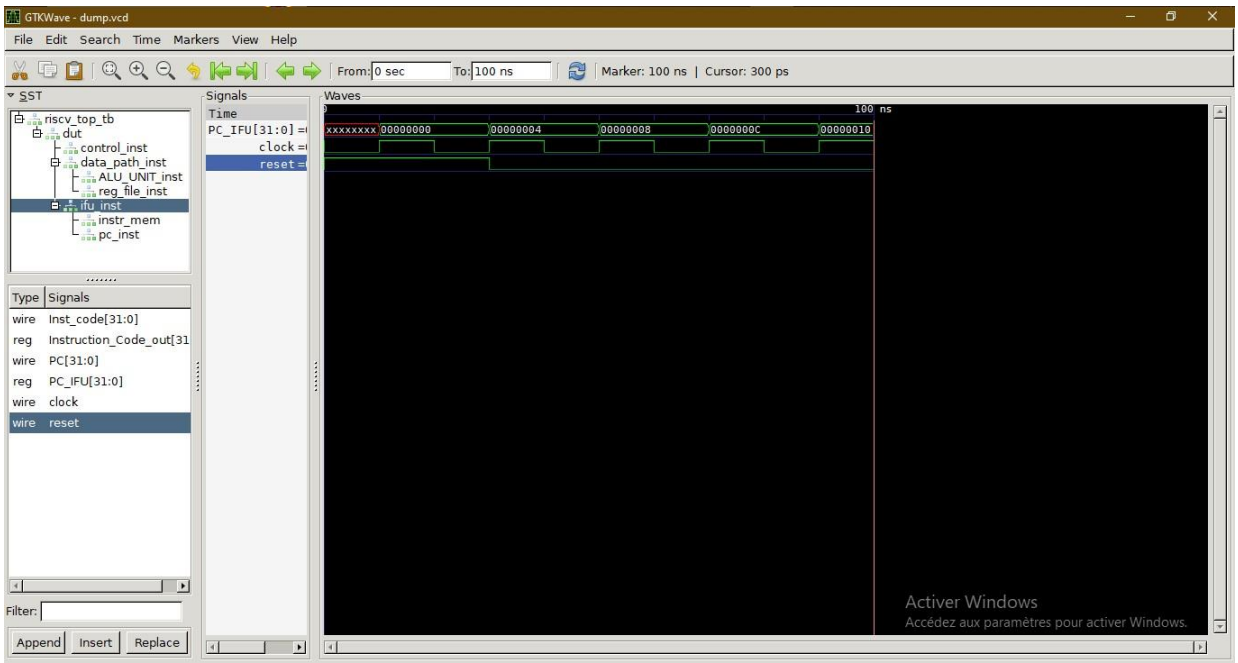

Figure 3.5: GTKWAVE

```
C:\iverilog\bin>gtkwave.exe dmp
```

Cette commande est utilisée pour ouvrir et visualiser les résultats de simulation stockésdans le fichier **dmp**.

3.2.4. Description Verilog

Verilog est un langage de description matérielle (HDL), utilisé pour décrire les systèmes et circuits numériques sous forme de code. Il a été développé par Gateway Design Automation au milieu des années 1980 et plus tard acquis par Cadence Design Systems. Il décrit la fonctionnalitéde la conception matérielle et l'outil de synthèse convertit les descriptions matérielles en une conception réelle comportant des éléments combinatoires et séquentiels.

Le langage Verilog est utilisé pour décrire les circuits numériques hiérarchiquement, en commençant par les éléments les plus basiques tels que les portes logiques et les bascules et en construisant des blocs et des systèmes fonctionnels plus complexes. Il prend également en chargeune gamme de techniques de modélisation, y compris la modélisation au niveau de la porte, au niveau RTL et au niveau du comportement.

Dernière norme Verilog : IEEE Standard 1364-2005.

3.3. Implémentation des types d'instructions

Exécution d'un petit programme

Pour tester le bon fonctionnement de notre processeur, nous avons essayé un petit programme de deux instructions de type R. Pour le faire, nous avons chargé la mémoire d'instructions manuellement.

Exemple d'encodage des instructions

L'instruction « ADDI x1, x2, 10 », est une opération du type immédiat-registre donc elle est encodée selon le format décrit auparavant.

L'instruction `ADDI` (Add Immediate) ajoute une valeur immédiate signée de 12 bits au contenu d'un registre source et stocke le résultat dans un registre de destination. (addi rd , rs1,imm0. Le convertisseur fournit l'encodage binanire, hexadecimale, etl'emplacemenr des registres sources ou destination.

La figure 3.6 montre l'exemple :

Figure 3.6 : Convertiseur RISC-V

Nous la figure 3.7 montre le decodage de l'instruction SUB X7, X5, X10

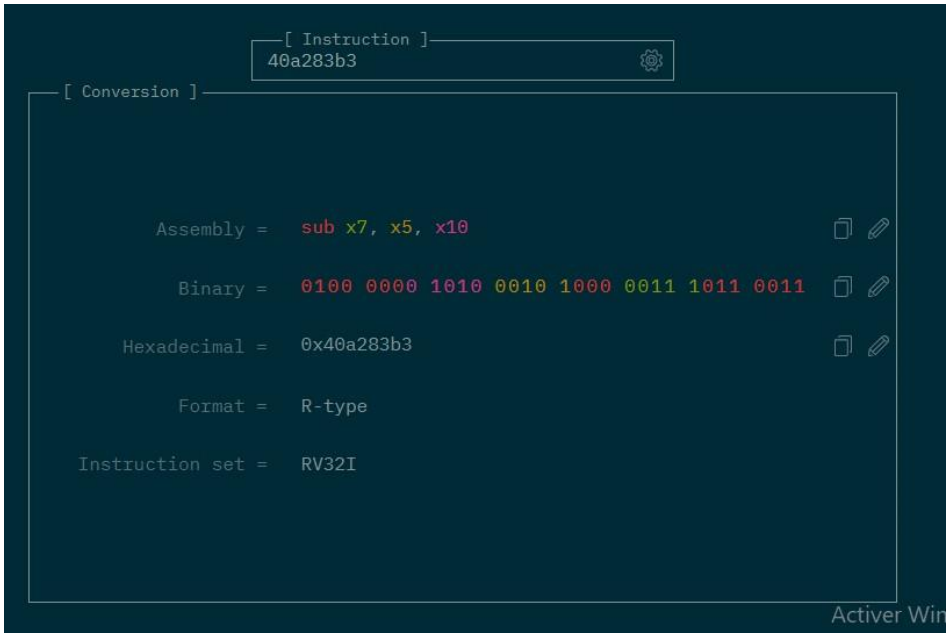

Figure 3.7: Encodage de l'instruction SUB X7, X5, X10

La figure 3.8 montre le décodage de l'instruction ADD X5, X3, X7

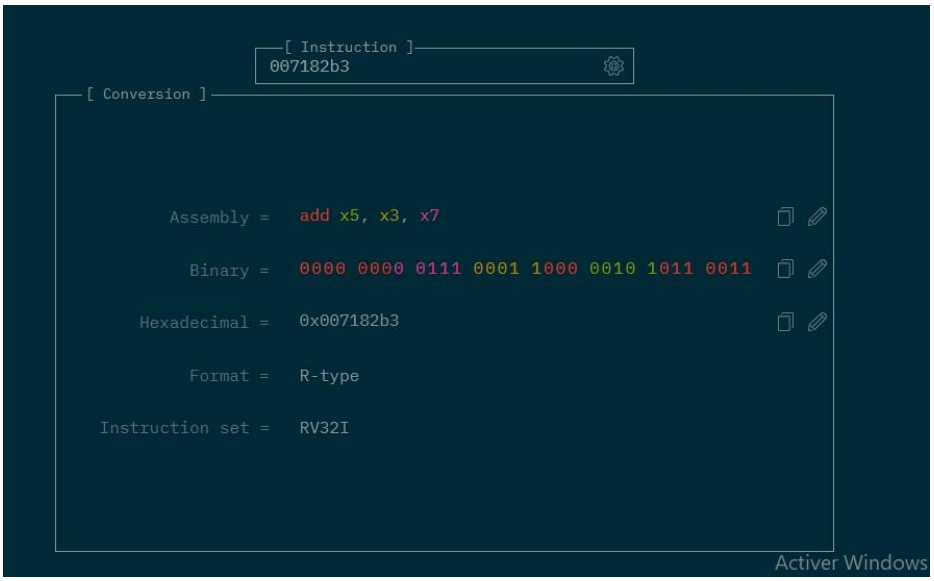

Figure 3.8: Encodage de l'instruction ADD X5, X3, X7

3.5. Discussion des résultats

3.5.1. DATAPATH du type R

Nous avons utilisé Vivado IDE pour générer le RTL de notre processeur des instructions de type R, et nous remarquons que le résultat est identique à notre conception théorique, les interconnexions sont correctement liées. La figure 3.8 montre le résultat :

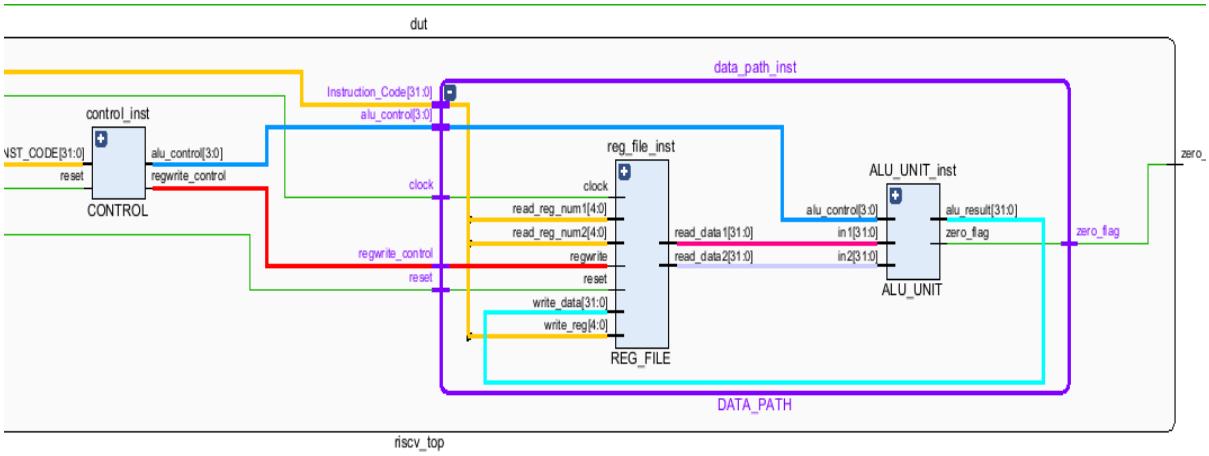

Figure 3.9: RTL du datapath de type R

Exécution d'un petit programme

00000000 // Example Register to register R-Type
007182b3 // add x5, x3, x7
405503b3 // sub x7, x10, x5
40a283b3 // sub x7, x5, x10

Figure 3.10 : Programme de test type R

Ces instructions sont stockées dans un fichier appelé Programm_R_app.mem. Le fichier .mem est utilisé pour initialiser la mémoire d'instructions du processeur dans une simulation Verilog. Chaque instruction de 32 bits est décomposée en 4 octets de 8 bits chacun, puis écrite en format hexadécimal dans le fichier. mem.

Le fichier .mem est utilisé pour stocker les instructions en mémoire de manière à ce que chaque ligne du fichier représente un octet de la mémoire. Cela permet à la fonction $readmemh de

Verilog de lire les instructions et de les charger dans la mémoire du processeur pour la simulation.

Ce programme en assembleur RISC-V exécute diverses opérations arithmétiques et logiques de type R entre les registres du processeur, comme l'addition, la soustraction et les décalages. Les instructions sont stockées dans un fichier. mem pour être chargées en mémoire lors de la simulation Verilog.

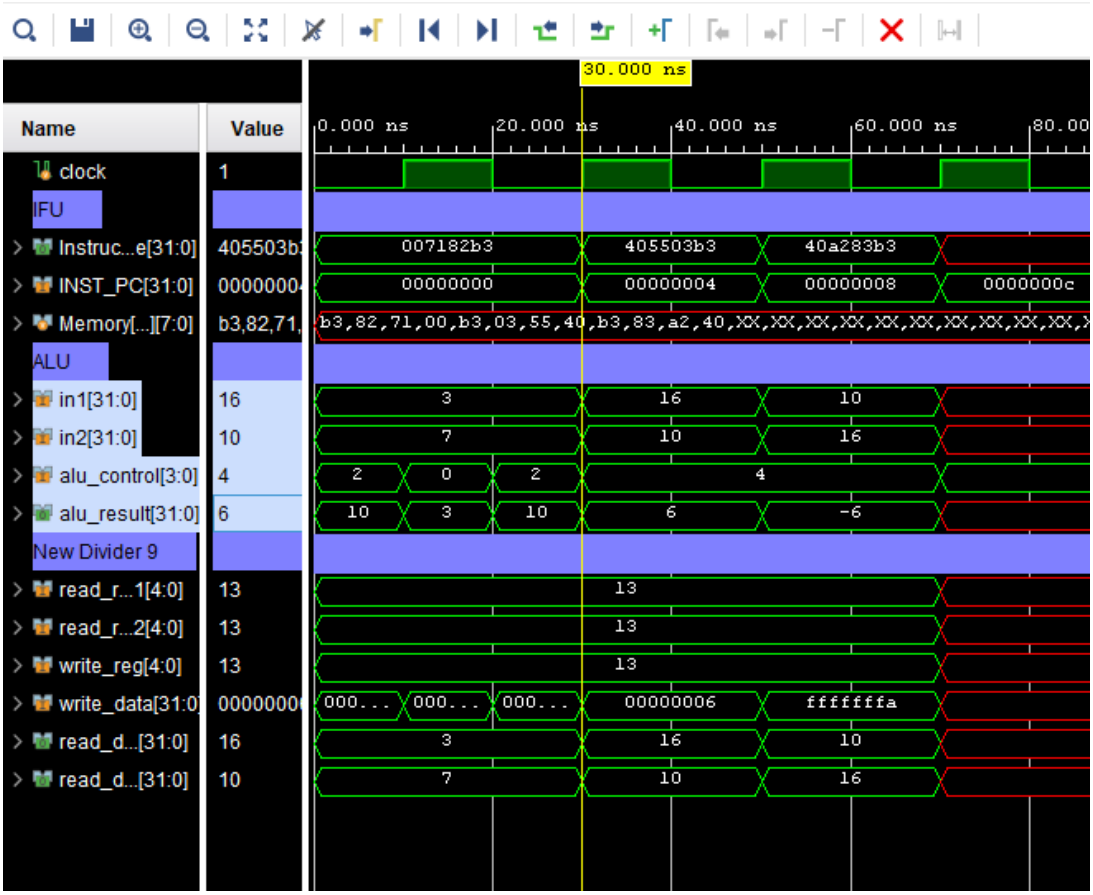

Figure 3.11 : Visualisation des signaux de type R

Ces instructions RISC-V exécutent les opérations suivantes : add x5, x3, x7 ajoute x3 et x7 pour obtenir 10, ce qui est correct. L'ALU (Unité Logique Arithmétique) sélectionne automatiquement l'opération correspondante pour chaque instruction, assurant ainsi que les résultats des soustractions sub x7, x10, x5 et sub x7, x5, x10 sont également calculés et reflétés correctement dans les registres x7.

3.5.2. Datapath de type B

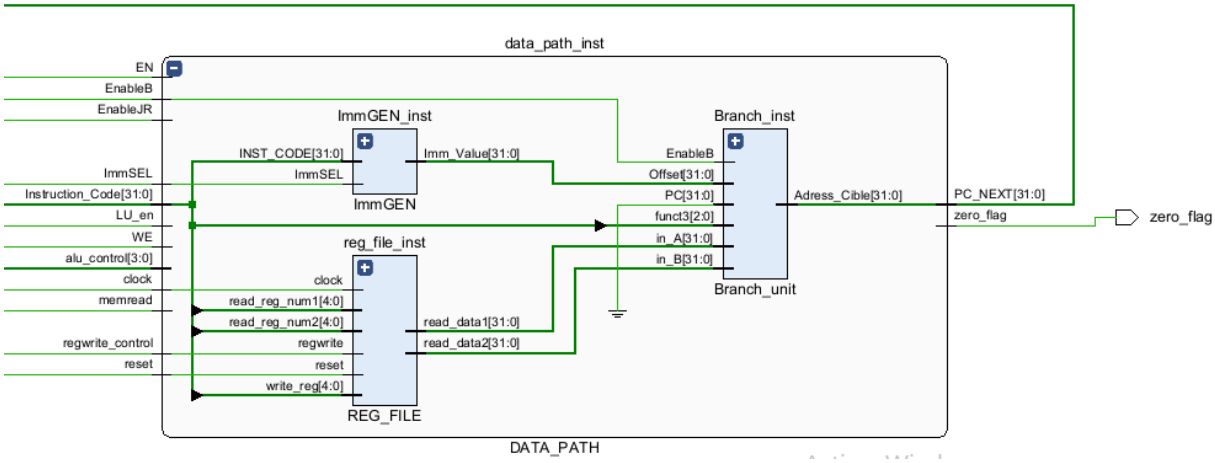

Figure 3.12 : RTL de l'unité de branchement

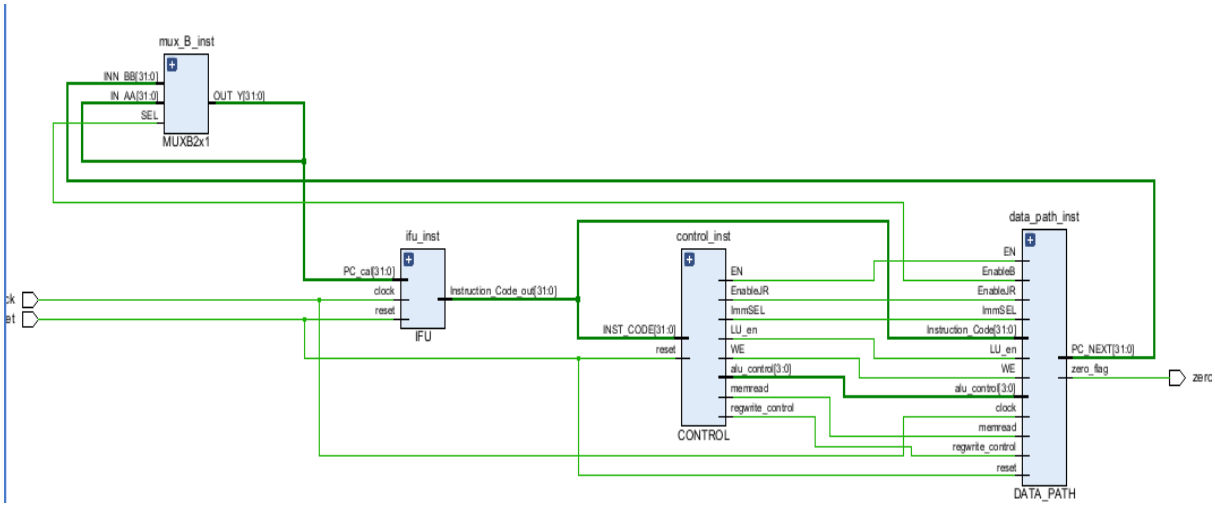

Figure 3.13 : RTL des instructions de type B

00000000 // Assume x5 is used for the loop counter and x6 holds the constant value 10 for comparison

0x00A00313 // addi x6, x0, 10 # Load immediate value 10 into register x6

```
0x00000293 // addi x5, x0, 0        # Initialize x5 to 0. Loop start
0x0062c263 // blt x5, x6, 8         # If x5 < x6, branch to the instruction to increment x5 (offset 8 bytes ahead)
0x00000013 // nop                   # Placeholder for the end of the loop
0x00000013 // nop                   # Placeholder for the end of the loop
0x00000013 // nop                   # Placeholder for the end of the loop
0x00128293 // addi x5, x5, 1        # Increment x5 by 1
0xfe629ee3 // bne x5, x6, -5        # If x5 != x6, branch back to the start of the loop (offset -12 bytes)
0x00000013 // nop                   # Placeholder for the end of the loop
```

Figure 3.14 : Programme de test type B

Ce programme en assembleur RISC-V initialise un registre avec la valeur 10 et utilise un autre registre comme compteur de boucle. Il incrémente le compteur à chaque itération jusqu'à ce qu'il atteigne 10, en effectuant une boucle qui se répète tant que le compteur est différent de 10. Des instructions `nop` servent de placeholders pour le bon alignement des branches dans la boucle. Voici le fonctionnement en détail :

- Initialisation : Le registre `x5` est initialisé à 0 (`addi x5, x0, 0`).

- Condition de boucle : La condition `blt x5, x6, 8` vérifie si `x5` est inférieur à `x6` (qui vaut 10). Si c'est le cas, il y a un branchement (`blt`) vers l'instruction `addi x5, x5, 1`, sinon la boucle se termine.

- Sortie de boucle : L'instruction `bne x5, x6, -5` permet de revenir au début de la boucle tant que `x5` n'est pas égal à `x6` (10).

Ainsi, la boucle s'arrête lorsque `x5` est égal à `x6` (10), car à ce moment-là, la condition de branchement `blt x5, x6, 8` ne sera plus satisfaite, et le programme poursuivra son exécution normale à partir de l'instruction suivante après la boucle.

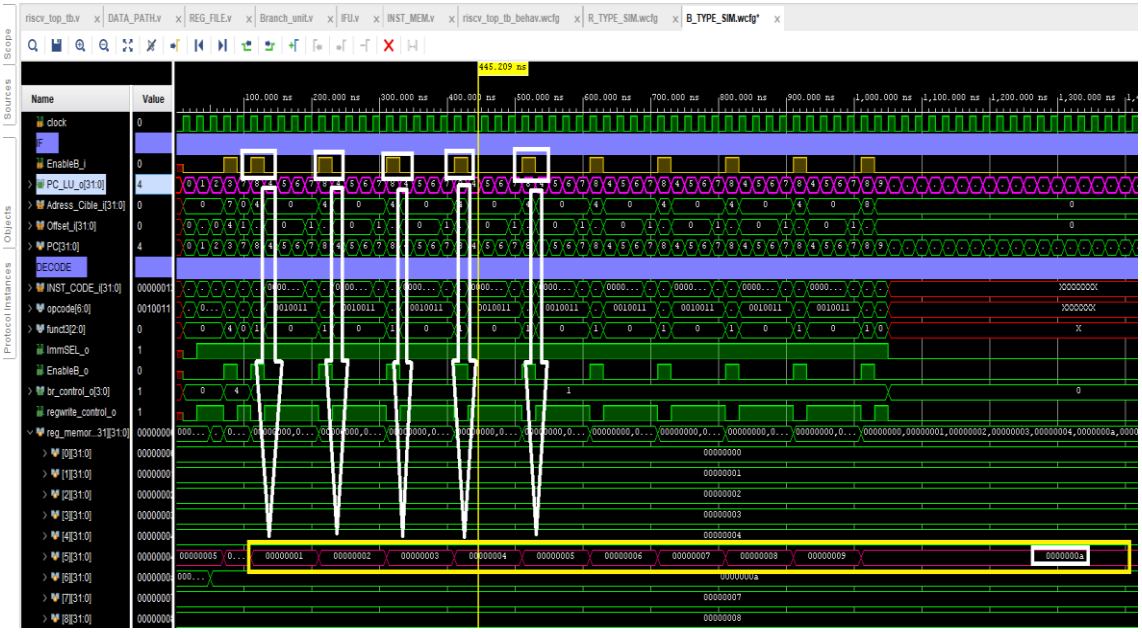

Figure 3.15 : Visualisation des signaux des instructions de type B

Dans la simulation de ce programme en assembleur RISC-V, l'image montre clairement que le compteur `x5` s'arrête à la valeur 10. L'activation du comptage se synchronise avec le signal `EnableB` et `active`, ce qui se produit tous les 5 cycles d'horloge comme spécifié. On observe que chaque itération du compteur est visible sur la simulation, confirmant que le programme fonctionne correctement selon sa logique de boucle contrôlée par `blt` et `bne`. Cette visualisation en simulation valide la précision et la fiabilité de l'exécution du programme dans un environnement virtuel, garantissant son comportement conforme aux attentes et aux spécifications définies.

3.5.3. Datapath du type I

00000000
00178793 //addi x15, x15, 1
00a00293 //addi x5, x0, 10
00528333 //add x6, x5, x5
0ff2c513 //xori x10, x5, 255

```
00229513 // slli x10, x5, 2
```

Figure 3.16 : Programme de test type I

Ce code assembleur RISC-V utilise des instructions de type I pour manipuler les registres du processeur. L'instruction `addi` est utilisée pour incrémenter `x15` de 1, montrant l'ajout d'une valeur immédiate à un registre existant. Ensuite, `addi` initialise `x5` avec la valeur constante 10. L'instruction `add` ajoute la valeur actuelle de `x5` à elle-même et stocke le résultat dans `x6`, illustrant une opération de type I qui utilise deux registres comme sources d'opération. Ensuite, `xori` réalise une opération XOR entre `x5` et 255, mettant le résultat dans `x10`, démontrant une autre manipulation efficace de données avec une valeur immédiate. Enfin, `slli` décale logiquement à gauche le contenu de `x5` de 2 bits et le place dans `x10`, montrant comment les décalages logiques peuvent être réalisés de manière simple avec des instructions de type I.

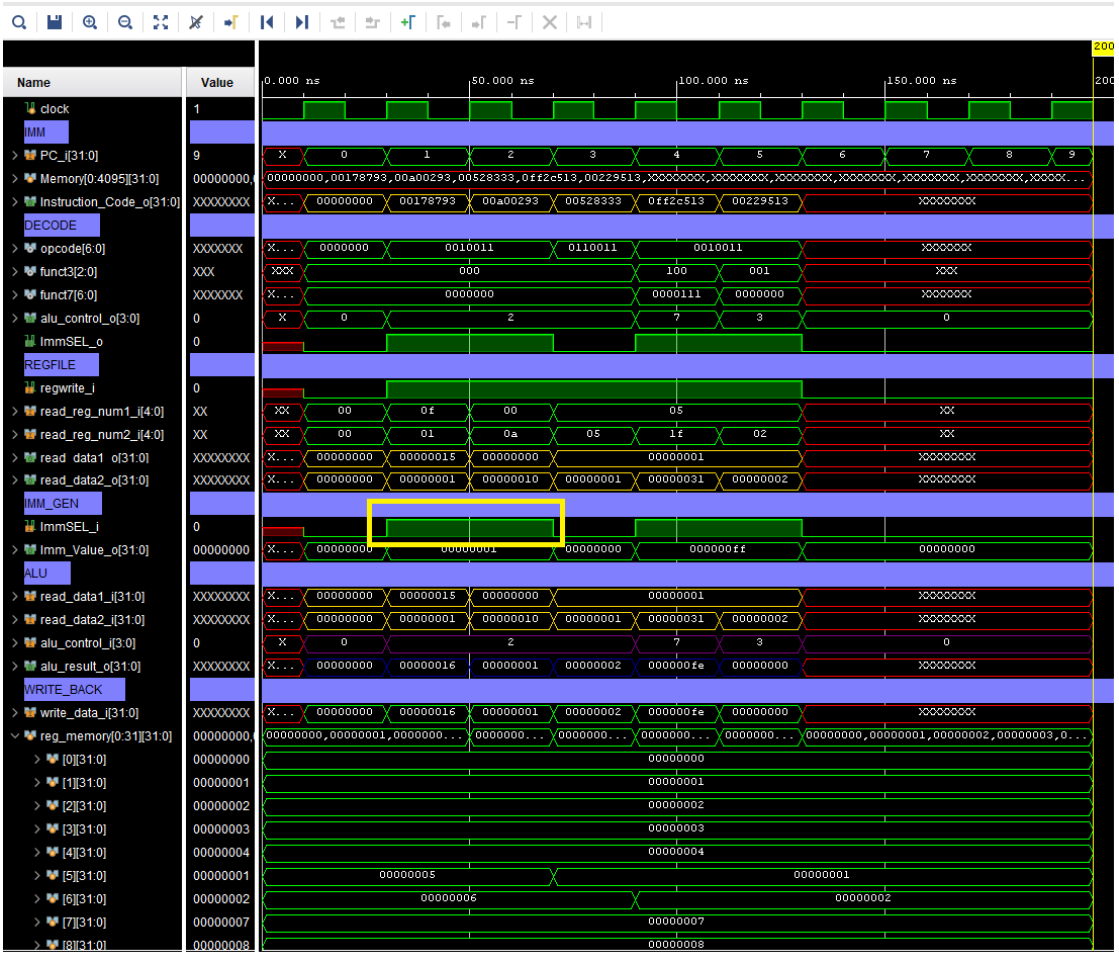

Figure 3.17 : Visualisation des signaux pour le type I

Dans la simulation de ce code assembleur RISC-V, les instructions spécifiées ont été exécutées pour vérifier leur fonctionnement sur les registres du processeur. La première instruction, `addi x15, x15, 1`, a incrémenté le registre `x15` de 1, actualisant sa valeur. Ensuite, `addi`

x5, x0, 10` a initialisé `x5` avec la valeur immédiate 10. L'instruction `add x6, x5, x5` a ensuite été exécutée, ajoutant la valeur de `x5` à elle-même et stockant le résultat 20 dans `x6`. Lors de l'exécution des instructions de type I comme `addi`, le générateur d'immédiat dans le processeur RISC-V s'active pour faciliter l'ajout de valeurs constantes aux registres, montrant ainsi la gestion efficace des opérations de données simples mais cruciales pour le traitement des instructions dans l'architecture RISC-V.

3.5.4. RTL du processeur global

La figure 3.19 montre le processeur global qui comporte le datapath, l'unité delecture et l'unité de contrôle

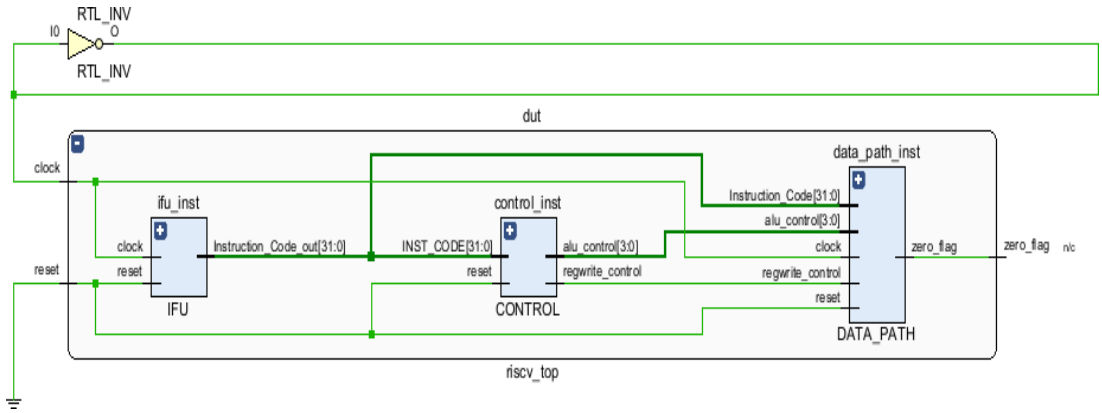

Figure 3.19 : RTL du datapath global

Dans le contexte de notre simulation, il est crucial que le processeur maintienne le contrôle sur l'Unité de Gestion des Instructions (IFU) et le Chemin de Données (Datapath). Dans notre cas, le processeur exécute principalement des instructions de type R, I et B. L'IFU s'assure de récupérer correctement les instructions depuis la mémoire, tandis que le Datapath effectue les opérations spécifiées par ces instructions, telles que les opérations arithmétiques, les accès mémoire et les branchements conditionnels. Ce contrôle garantit que chaque instruction est correctement traitée et que le flux d'exécution du programme est maintenu conformément à sa logique et à son ordre d'exécution attendus.

3.6. Perspective

Jusqu'à présent, nous avons effectué avec succès des tests sur les instructions de type I, R et B du processeur RISC-V, confirmant ainsi leur

bon fonctionnement pour les opérations arithmétiques, logiques et les branchements conditionnels. La prochaine étape devait initialement inclure les tests des instructions de stockage de type S, des instructions de chargement de type I ainsi que des sauts inconditionnels pour compléter la validation des fonctionnalités du processeur. Cependant, en raison de contraintes de temps, nous n'avons pas pu achever ces tests ni les implémenter sur un FPGA comme prévu dans le cadre de ce mémoire.

Conclusion

Dans ce chapitre, nous avons effectué des tests sur notre processeur et obtenu des résultats corrects pour les instructions de type R. Cependant, en raison de contraintes de temps, nous n'avons pas pu tester les autres types d'instructions. Malgré cela, les résultats obtenus pour le typeR nous donnent confiance dans la fonctionnalité de base de notre processeur. Pour une évaluationcomplète et détaillée, il serait nécessaire de réaliser des tests supplémentaires sur les autres types d'instructions, ce qui pourrait être une perspective pour les travaux futurs.

Conclusion générale

Les processeurs open source représentent une révolution dans le domaine de l'architecture matérielle, offrant des avantages significatifs tels que la flexibilité, la réduction des coûts et la possibilité d'innover grâce à une communauté collaborative. L'architecture RISC-V, en particulier, incarne ces avantages en proposant une instruction set architecture (ISA) libre et extensible, permettant aux concepteurs de matériel de créer des processeurs sur mesure sans les contraintes des licences propriétaires. Dans le cadre de notre projet, nous avons choisi de nous concentrer sur la conception et l'implémentation d'un processeur basé sur l'architecture RV32I, un sous-ensemble de RISC-V, en utilisant le langage de description matérielle Verilog.

En choisissant RV32I, nous avons eu l'opportunité de travailler avec une architecture bien documentée et soutenue par une vaste communauté d'utilisateurs et de développeurs. Cela nous a permis de tirer parti des ressources disponibles tout en ajoutant notre propre contribution à cette technologie émergente.

La conception de notre processeur RV32I a impliqué plusieurs étapes clés. Tout d'abord, une compréhension approfondie de l'architecture et des spécifications de RISC-V a été essentielle. Nous avons ensuite modélisé le processeur en Verilog, décrivant chaque composant essentiel, y compris l'unité de traitement des instructions, les registres et les unités arithmétiques et logiques. L'utilisation de Verilog nous a permis de créer une description matérielle précise et modulaire, facilitant les modifications et les extensions futures.

La phase de simulation et de vérification a été cruciale pour garantir le bon fonctionnement de notre processeur. En utilisant Icarus Verilog, nous avons réalisé des simulations extensives pour tester chaque composant.

Malgré que nous n'avons pas pu obtenir les résultats attendus en raison du manque du temps , nous avons eu des résultats corrects concernant le type R .

L'un des principaux bénéfices de ce projet a été l'acquisition de compétences pratiques en conception et implémentation de processeurs. Nous avons appris à utiliser des outils de description matérielle, à effectuer des simulations détaillées et à optimiser les performances du processeur. Ces compétences seront inestimables pour notre future carrière dans le domaine des systèmes embarqués et de l'architecture des processeurs.

En conclusion, ce projet a démontré la faisabilité et les avantages de concevoir un processeur open source basé sur l'architecture RISC-V. Le choix de RV32I nous a permis de tirer parti d'une architecture flexible et bien supportée, tout en acquérant une expérience précieuse en conception matérielle. Nous avons non seulement atteint nos objectifs techniques, mais nous avons également contribué à l'avancement de la technologie open source, ouvrant la voie à de futures innovations et collaborations dans ce domaine passionnant.

Bibliographies

Chapiter 1

[1] ALL ABOUT ELECTRONICS. *Difference between Microprocessor and Microcontroller*. (23 mars 2017). Consultée le 12 mars 2024. [Vidéo en ligne]. Disponible : https://www.youtube.com/watch?v=dcNk0urQsQM

[2] JDN , « Microcontrôleur : définition et composants ». (13 juillet 2020) Consulté le 23 mars 2024. [En ligne]. Disponible : https://www.journaldunet.fr/web-tech/dictionnaire-de-l-iot/1440684-microcontroleur-definition-et-composants/

[3] david monet Harris and sarah l.Harris , digital design and computer architecture

[4] Tahia Tabassum. *Single Cycle, Multi Cycle, and Pipelining*. (9 janv. 2020). Consulté le 26 avril 2024. [Vidéo en ligne]. Disponible : https://www.youtube.com/watch?v=wXo5eyeJZcU

[5] D. Feugey. « Open Hardware : l'Open Source à l'assaut des processeurs | Silicon ». Silicon. Consulté le 26 avril 2024. [En ligne]. Disponible : https://silicon.fr/futur-de-lit-open-hardware-lopen-source-a-lassaut-des-processeurs-356853.html

[6] « Comment RISC-V se compare-t-il à d'autres architectures open source comme OpenRISC et RISC-V ? » LinkedIn : s'identifier ou s'inscrire. Consulté le 2 mai 2024. [En ligne]. Disponible : https://fr.linkedin.com/advice/3/how-does-risc-v-compare-other-open-source-architectures?lang=fr

[7] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. Berkeley, CA: EECS Department, University of California, Berkeley, 2017.

[8] "PicoRV32 - A Size-Optimized RISC-V CPU," YosysHQ, [en ligne]. Disponible: https://github.com/YosysHQ/picorv32/blob/main/README.md. [Consulte le: Juin. 2, 2024].

[9] "BOOM Core Documentation," [Online]. disponible: https://docs.boom-core.org/en/latest/index.html. [Consulte le : Jun. 2, 2024].

[10] "ORCA: An Open-Source RISC-V CPU," kingcard1131, [en ligne]. disponible: https://github.com/kingcard1131/orca. [consulte : juin. 2, 2024].

[11] "Potato - A Simple RISC-V CPU," skordal, [en ligne]. disponible: https://github.com/skordal/potato. [consulte: Jun. 2, 2024].

[12] "ASIC Full Form," GeeksforGeeks, [en ligne]. disponible: https://www.geeksforgeeks.org/asic-full-form/. [consulte: Jun. 2, 2024].

[13] « What is an FPGA ? Field Programmable Gate Array ». AMD. Consulté le 15 mai 2024. [En ligne]. Disponible : https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

Chapitre 2

[7] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. Berkeley, CA: EECS Department, University of California, Berkeley, 2017.

[14] D. M. Harris and S. L. Harris, *Digital Design and Computer Architecture*, 2nd ed. San Francisco, CA: Morgan Ka ufmann, 2012.

[15] PASSLAB, "RISC-V Implementation Notes," *CSE564: Computer Systems Engineering*, Arizona State University, 2018. [en ligne]. disponible: https://passlab.github.io/CSE564/notes/lecture08_RISCV_Impl.pdf. [Consulté: Jun. 2, 2024].

Chapitre 3

[16] Arduino Factory. "Introduction à Xilinx Vivado." Arduino Factory. disponible: https://arduinofactory.fr/xilinx-vivado/#:~:text=Vivado%20a%20%C3%A9t%C3%A9%20introduit%20en,un%20environnement%20de%20d%C3%A9bogage%20commun. Consulté le 2 juin 2024.

[17] S.-L. Hsu. “Installing Icarus Verilog (Iverilog): A Quick Guide.” Circuit Cove., 2024. [Online]. Available: https://circuitcove.com/tools-iverilog-installation/ consulté le 2 juin 2024 .

[18] “Waveforms With GTKWave — Icarus Verilog documentation.” · GitHub Pages. Consulté le 2 juin 2024 . [en ligne]. disponible: https://steveicarus.github.io/iverilog/usage/gtkwave.html