Rapport #2

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Rapport du Laboratoire 2 - Conception d'un microprocesseur

ELE8304 - Circuits intégrés à très grande échelle

Automne 2023
Département de génie électrique
École Polytechnique de Montréal

Dernière mise à jour: 16 décembre 2023

Ismail **Essaidi** 2306597
Maxime **Pietrera-Ferrandini** 2312199

Table des matières

Table des figures

1 Introduction

Depuis plusieurs décennies, l'évolution des performances des microprocesseurs intégrés est réelle et au coeur de notre civilisation. Depuis l'Intel 4004 en 1971, cette évolution se déroule sur deux principaux terrains d'innovations :

- Les technologies de transistor : La réduction des tailles des transistors, l'augmentation de la densité d'intégration et la réduction de la tension d'alimentation. Cela a permis une augmentation des fréquences d'opérations des processeurs tout en gardant une densité de puissance quasi-constante.
- Les techniques de conceptions des microarchitectures de processeurs : le pipeline, les caches, les prédicteurs de branche, les bus, ... Cela a permis de tirer le meilleur parti des technologies de semiconducteurs.

Ainsi, dans ce laboratoire, nous allons concevoir et implémenter, en technologie 45 nm, un microprocesseur RISCV simple appelé mini-riscv. Son jeu d'instruction est dérivé de l'architecture RV32I, et sa microarchitecture utilise un pipeline à 5 étages. L'objectif de ce laboratoire est de permettre d'appréhender les étapes nécessaires à la conception d'un microprocesseur, de sa description matérielle en VHDL jusqu'à son implémentation en circuit intégré.

2 Présentation du mini-riscv

La mini-riscv est un processeur possédant une architecture complexe. Cette partie vise donc à décrire sommairement cette architecture, c'est-à-dire l'ensemble des spécifications fonctionnelles du processeur. Cela comprend le jeu d'instructions supporté par le processeur, les registres utilisables, l'interfaçage avec la mémoire ainsi que les éventuelles entrées/sorties. Il s'agit finalement de l'interface entre le logiciel et le matériel.
Pour décrire l'architecture du mini-riscv, nous pouvons utiliser plusieurs échelles : au niveau réalisation en logique numérique, le mini-riscv est réalisé dans une technologie CMOS 45 nm dont le jeu d'instructions est dérivé de l'architecture RV23I.
Au niveau externe, l'architecture du mini-riscv est composée d'un jeu d'instructions qui permet au *core*, ou architecture interne, de réaliser une suite d'instructions tout en intéragissant avec les mémoires d'instructions (où est stockée la suite d'instruction) et de données. Le *core*, capable de traiter et d'executer les instructions, est pipeliné sur 5 étages et utilise des registres pour effectuer les différentes opérations.

2.1 Jeu d'instructions

Le jeu d'instructions (*ISA : Instruction Set Architecture*) spécifie toutes les instructions supportées par le mini-riscv. Il définit aussi les modes d'adressage des opérandes ainsi que le format des instructions. Les instructions sont au format fixe de 32 bits.
Il existe plusieurs types d'instructions réparties en 6 formats. Ces formats permettent de séparer les différents bits d'une instruction pour y coder par exemple des portions d'identification (comme un *opcode*), des portions qui encodent l'adresse des registres ou des portions qui encodent des valeurs immédiates. La liste des 25 insctructions supportées par le mini-riscv se trouve sur la figure 35 en

annexe.
Plus précisément, les instructions se classent selon plusieurs types de sous-instructions :

— **Les instructions de Branchement :** JAL et JALR sont des sauts inconditionnels qui modifient la valeur de la prochaine instruction à effectuer dans le Program Counter. L'instruction BEQ quant à elle est un branchement contidionnel qui dépend de la valeur de la condition testée (ici *BEQ = Branch on EQual* indique que si le résultat de la comparaison donne une égalité, on effectue le saut).

— **Les instructions d'accès à la mémoire de données :** la plupart des instructions permettent de réaliser des opérations seulement sur les registres, il faut donc les instructions LW et SW pour lire et écrire des données dans la mémoire. LW permet de lire une valeur depuis la mémoire et de la sauvegarder dans un registre tandis que SW permet d'écrire une valeur depuis un registre vers une adresse de la mémoire donnée.

— **Les instructions d'opérations arithmétiques et logiques avec une valeur immédiate :** les instructions ADDI, SUBI, ORI, XORI, SLTI, SLTIU, SLLI, SRLI, SRAI réalisent les mêmes opérations que les suivantes mais utilisent une valeur immédiate à la place du deuxième registre.

— **Les instructions d'opérations arithmétiques et logiques :** ces instructions encodent les différentes opérations réalisables avec le mini-riscv. Les détails se trouvent dans le tableau suivant :

Code de l'instruction	**Description**
ADD	addition entre deux registres
SUB	soustraction entre deux registres
AND	ET logique entre deux registres
OR	OU logique entre deux registres
XOR	OU exclusif logique entre deux registres
SLL	décalage logique à gauche
SRL	décalage logique à droite
SRA	décalage arithmétique à droite
SLT	compare le registre *src1* au registre *src2* signés et fixe *dest* à '1' si $src1 < scr2$ et à '0' sinon
SLTU	équivalent à *SLT* avec des registres non signés
LUI	place les 20 premiers bits de son immédiat dans les 20 bits de poids forts du registre destination, et complète avec des 0

2.2 Interfaces Mémoires

Le mini-riscv est composé de deux types de mémoires, ce qui en fait une mémoire double-port adressable par octet. En effet, chaque octet possède une adresse unique. Les deux espaces distincts, de 1 kB chacun, permettent de gérer l'accès à la mémoire d'instructions (*imem*) sur le port a ainsi

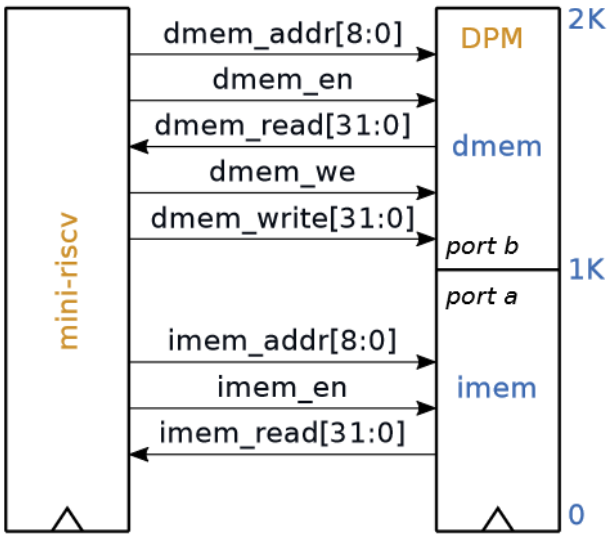

FIGURE 1 – Schéma de principe des interfaces mémoires

qu'à la mémoire de données (*dmem*) sur le port b. La mémoire d'instructions contient la suite des instructions qui doivent être exécutées tandis que la mémoire des données contient les différentes données accessibles en lecture et en écriture. Cependant, la mémoire possède une latence d'un cycle qu'il faut prendre en compte. Le schéma de la mémoire ainsi que son interface avec le mini-riscv est montré sur la figure 1.

2.3 Registres

Au niveau des registres accessibles sur le mini-riscv, le tableau suivant définit la convention utilisée :

Nom	Numéro	Utilisation
zero	x0	Valeur zéro
ra	x1	Adresse de retour des fonctions
sp	x2	Stack Pointer
gp	x3	Global Pointer
tp	x4	Thread Pointer
t0-t2	x5-x7	Temporaires
s0-s1	x8-x9	Sauvegardes
a0-a7	x10-x17	Arguments de fonctions
s2-s11	x18-x27	Sauvegardes
t3-t6	x28-x31	Temporaires

3 Modules

Dans cette partie, nous allons nous intéresser aux différents modules du processeur. Pour ce laboratoire, nous nous sommes intéressés aux modules suivants :

- Le banc de registres
- Le Program Counter (PC)
- L'Unité Arithmétique et Logique (ALU)
- L'adder 32 bits

De plus, pour pouvoir les tester, nous avons réalisé des bancs d'essais spécifiques pour ces 4 modules. Enfin, dans le cadre de ce laboratoire, les trois premiers modules ont été préalablement fournis. Ce qui fait que, concrètement, nous avons seulement créé le modèle VHDL de l'adder 32 bits.

3.1 Le banc de registres

Le premier module, le banc de registres, est un module qui gère l'écriture et la lecture sur les 32 registres de 32 bits du processeur *mini-riscv*. Son schéma de principe est présenté sur la figure 2.

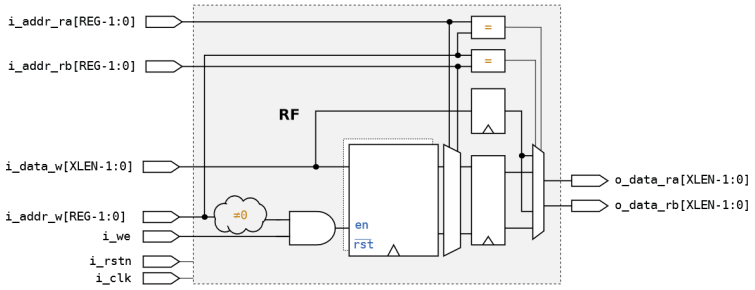

FIGURE 2 – Schéma de principe du banc de registres

Le banc de registres est utilisé avec deux paramètres génériques : **REG** et **GEN** qui correspondent respectivement à la taille des signaux d'adresses ainsi qu'à celle des signaux des données. Le signal *i_rstn* correspond au reset du banc de registres qui affecte tous les emplacements à 0 lorsqu'il vaut 0. Le signal *i_clk* correpond à l'horloge. En effet, la lecture et l'écriture des données s'effectuent sur front montant de l'horloge. Pour l'écriture, sur front montant de l'horloge, l'adresse pointée par le signal *i_adr_w* est affectée à *i_data_w* si *i_we* vaut 1, et à sa valeur précédente si *i_we* est égal à 0. En d'autres termes, le signal *i_we* correspond à *write enable* qui contrôle l'activation de l'écriture. Pour la lecture, sur front montant de l'horloge, les sorties *o_data_ra/rb* correspondent aux valeurs pointées par les adresses *i_addr_ra/rb* dans la mémoire. Pour terminer, lorsqu'une adresse de lecture est la même qu'une adresse d'écriture, alors les signaux de sortie doivent refléter la valeur de *i_data_w*. L'interface VHDL du banc de registres est le suivant :

```
entity riscv_rf is
    port (
        i_clk : in std_logic;
```

```
        i_rstn : in std_logic;
        i_we : in std_logic;
        i_addr_ra : in std_logic_vector(REG-1 downto 0);
        o_data_ra : out std_logic_vector(XLEN-1 downto 0);
        i_addr_rb : in std_logic_vector(REG-1 downto 0);
        o_data_rb : out std_logic_vector(XLEN-1 downto 0);
        i_addr_w : in std_logic_vector(REG-1 downto 0);
        i_data_w : in std_logic_vector(XLEN-1 downto 0));
end entity riscv_rf;
```

Vérification du banc de registres avec banc de test

Pour simuler et vérifier le bon fonctionnement du banc de registres, nous avons codé un banc de tests permettant de tester plusieurs situations :

- **Test 1 :** reset.
- **Test 2 :** écriture de la valeur ”00000001” à l'adresse ”0x01” avec un $we = 1$.
- **Test 3 :** écriture de la valeur ”00000000” à l'adresse ”0x01” avec un $we = 0$.
- **Test 4 :** écriture de la valeur ”00000008” à l'adresse ”0x02” avec un $we = 1$.
- **Test 5 :** écriture de la valeur ”00000004” à l'adresse ”0x04” avec un $we = 1$ et lecture des registres ”0x02” et ”0x01”.
- **Test 6 :** lecture des registres ”0x02” et ”0x04”.

Le résultat de la simulation est présenté à la figure 3.

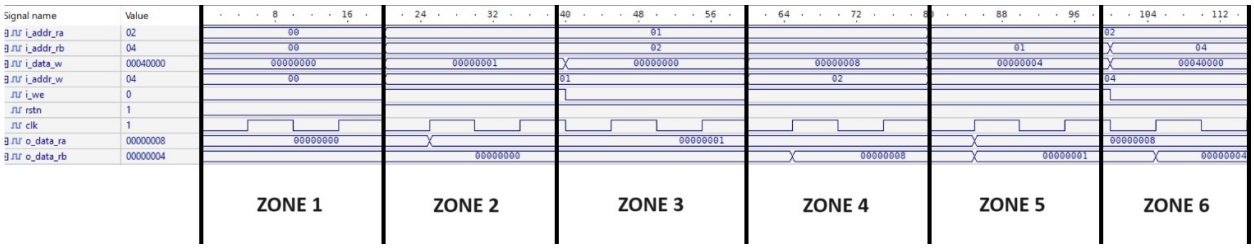

FIGURE 3 – Simulation du banc de registres

On observe maintenant les sorties du banc de registres pour vérifier le bon fonctionnement.

Zone	Valeur de *o_data_ra*	Valeur de *o_data_rb*	Interprétation
1	00000000	00000000	Le reset est bien effectué. Tous les registres sont mis à 0.
2	00000001	00000000	La valeur 1 est bien écrite à l'adresse 01 et lue au front montant d'horloge. Rien ne se passe sur la sortie b.
3	00000001	00000000	Rien ne change à l'adresse 01 (we = 0) et 1 est toujours bien lu à cette adresse.
4	00000001	00000008	8 est bien écrit à l'adresse 02 et lue au front montant de l'horloge.
5	00000008	00000001	8 est bien lu à l'adresse 02 et 1 à l'adresse 02.
6	00000008	00000004	8 est bien lu à l'adresse 2 et 4 à l'adresse 04. L'écriture réalisée à l'étape précédente à bien fonctionné.

On voit donc que le Banc de Registres fonctionne parfaitement. Il écrit uniquement lorsque $we = 1$ et lit les valeurs des bonnes adresses demandées. Les valeurs des sorties correspondent bien aux valeurs pointées par *i_addr_ra/rb*.

3.2 Le Program Counter (PC)

Le module du Program Counter permet de gérer le bus d'adresses des instructions dans la mémoire. En effet, sa sortie pointe vers la prochaine instruction à exécuter. La figure 4 montre le schéma de principe du PC.

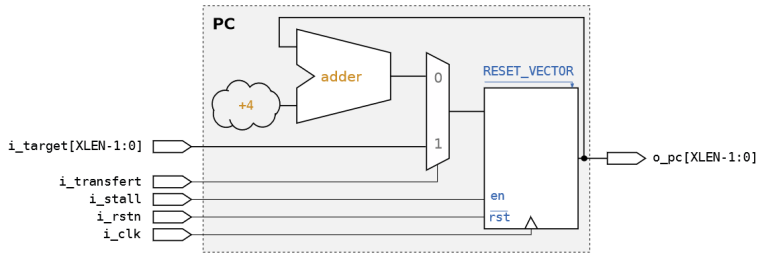

FIGURE 4 – Schéma de principe du Program Counter

La sortie pointée par le PC est sous format 32 bits. A chaque front montant d'horloge *i_clk*, la sortie *o_pc* est incrémentée de 4 grâce à un module *adder* tant que *i_transfert* est égale à 0. En effet, si ce dernier vaut 1, cela correspond à un branchement et *o_pc* doit être mis à jour avec la valeur *i_target*. Pour effecteur un reset, lorsque *i_rstn* vaut 0, *o_pc* est initialisé à **RESET_VECTOR**, soit la valeur 16#00000000#.
Enfin, l'interface VHDL du PC utilise l'architecture suivante :

```
entity riscv_pc is
    generic (RESET_VECTOR : natural := 16#00000000#);
    port (
        i_clk : in std_logic;
        i_rstn : in std_logic;
        i_stall : in std_logic;
        i_transfert : in std_logic;
        i_target : in std_logic_vector(XLEN-1 downto 0);
        o_pc : out std_logic_vector(XLEN-1 downto 0));
end entity riscv_pc;
```

Vérification du module PC avec banc de test

Pour simuler et vérifier le bon fonctionnement du PC, nous avons codé un banc de tests permettant de tester plusieurs situations :

- **Test 1 :** reset.
- **Test 2 :** incrémentation du PC en mode normal avec *i_transfert = '0'*.
- **Test 3 :** incrémentation du PC en mode branchement avec *i_transfert = '1'* et *i_target = '00000004'*.

Le résultat de la simulation est présenté à la figure 5.

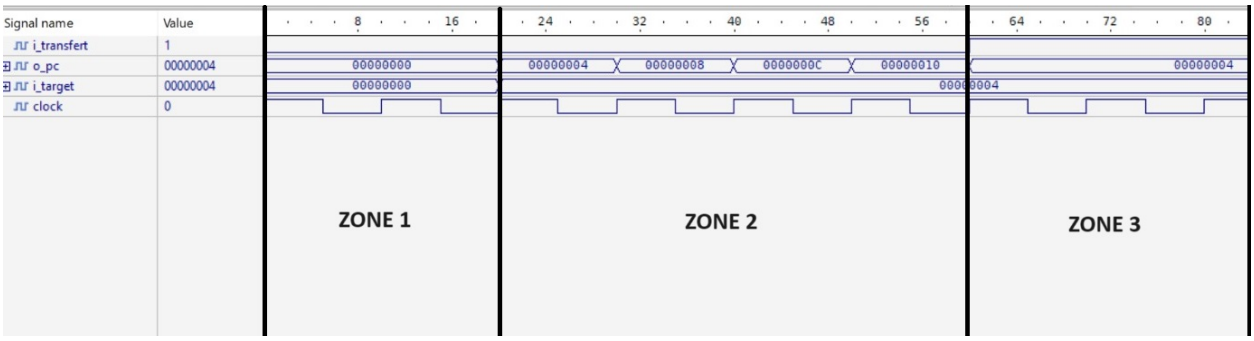

FIGURE 5 – Simulation du banc de test du Program Counter

On observe maintenant les sorties du PC pour vérifier le bon fonctionnement.

Zone	**Valeur de *o_pc***	**Interprétation**
1	00000000	Le reset est effectué. Tous les registres sont mis à 0.
2	*o_pc+ = 4*	La sortie du PC s'incrémente de 4 en 4 correctement.
3	00000004	La sortie du PC passe à l'adresse '00000004'. Le branchement est donc correctement effectué.

On voit donc que le Program Counter fonctionne parfaitement. Il s'incrémente de 4 en 4 tant que *i _transfert* n'est pas à '1'. Lorsqu'il passe à 1, il prend bien la valeur de *i_target*.

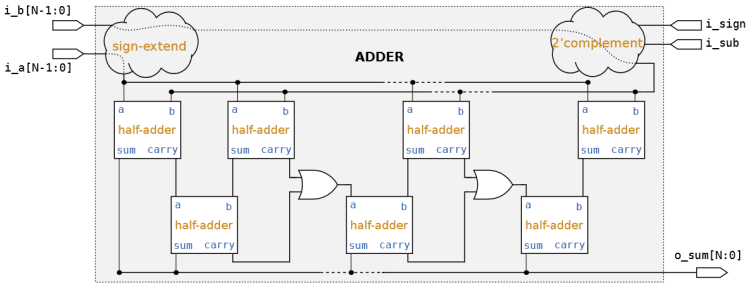

FIGURE 6 – Schéma de principe de l'adder

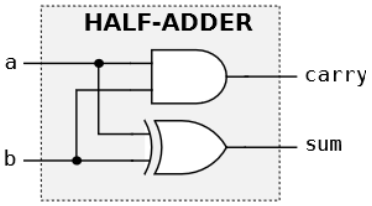

FIGURE 7 – Schéma de principe d'un half-adder

3.3 L'Adder

L'adder est un module combinatoire qui permet de réaliser l'addition sur 32 bits de deux opérandes, les signaux *i_a* et *i_b*. Le schéma de principe de l'adder est exposé sur la figure 6. Ce module utilise une technique de conception assez particulière : le *ripple-carry*. En effet, elle est basée sur une multitude de sous-modules plus petits en série, des *half-adder* qui permettent de propager une retenue pour réaliser l'addition sur plusieurs bits. Chaque *half-adder* permet de réaliser l'addition sur un bit et de générer une retenue en plus du résultat, comme on peut le voir sur le schéma de la figure 7. Ainsi, l'adder réalise l'addition des deux signaux d'entrée, sur 32 bits, pour donner le résultat *o_sum* en sortie sur 33 bits. Le 33ème bit de la sortie permet de donner le signe du résultat dans une logique signée. C'est le signal *i_sign* qui contrôle si l'opération est réalisée sur des entiers signés (lorsqu'il vaut 1) ou non signés (lorsqu'il vaut 0). De plus, le signal *i_sub* indique si l'opération à réaliser est une soustraction ou un addition. Lorsqu'il vaut 1, une soustraction doit être effectuée, mais cela consiste en réalité à additionner *i_a* avec le complément à 2 de *i_b*.
L'interface de l'adder est donc le suivant :

```
entity riscv_adder is
    generic (N : positive := 32);
    port (
        i_a : in std_logic_vector(N-1 downto 0);
        i_b : in std_logic_vector(N-1 downto 0);
        i_sign : in std_logic;
        i_sub : in std_logic;
        o_sum : out std_logic_vector(N downto 0));
```

```
end entity riscv_adder;
```

Ce module étant le seul que nous devions réaliser, nous allons détailler un peu plus sa conception. Tout d'abord, nous avons dû créer le module VHDL de l'half-adder. Pour cela, nous avons instancié l'interface suivant :

```
entity half_adder is
    port (
        a, b       : in  std_logic;
        sum, carry : out  std_logic);
end entity half_adder;
```

Les entrées a et b sont des bits au même titre que les sorties *carry* et *sum*. On a alors : $carry = a\ AND\ b$ et $sum = a\ XOR\ b$. Par la suite, nous avons généré 33 structures de deux half-adder pour réaliser le module complet de l'adder (en les connectant comme sur la figure 6). Il est aussi assez important de travailler avec des signaux de 33 bits images de *i_a* et de *i_b* pour être raccord avec la sortie. Pour faire cela, nous avons préalablement créé deux signaux intermédiaires représentant *i_a* et *i_b* en réalisant une extension de signe sur le 33ème bit.

Vérification de l'adder avec banc de test

Pour vérifier notre adder, nous avons réalisé un banc de test pour observer les sorties en fonction des tests suivants :

- **Test 1 :** réalisation de l'opération : $1 + 2$ (*i_sign = '0' et i_sub = '0'*).
- **Test 2 :** réalisation de l'opération : $1 + (-1)$ (*i_sign = '1' et i_sub = '0'*).
- **Test 3 :** réalisation de l'opération : $4 - 1$ (*i_sign = '0' et i_sub = '1'*).
- **Test 4 :** réalisation de l'opération : $(-2) - (-1)$ (*i_sign = '1' et i_sub = '1'*).

Le résultat de la simulation est présenté à la figure 8.

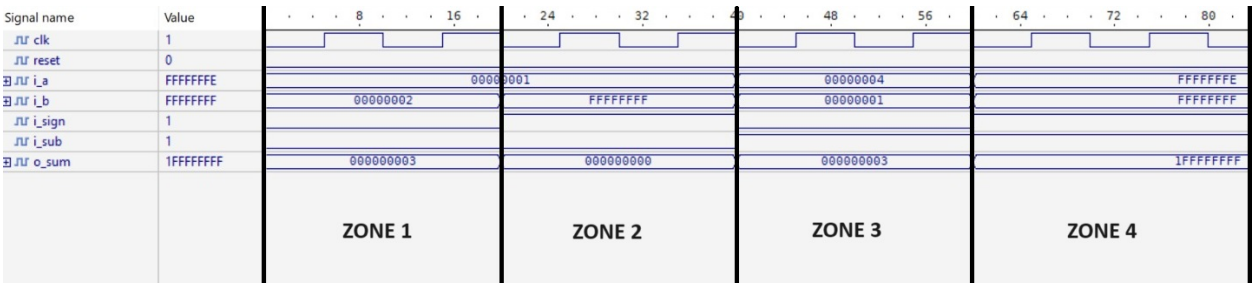

FIGURE 8 – Simulation du banc de test de l'adder

On peut donc observer les résultats pour valider le comportement des signaux de sortie. On a alors :

Zone	Valeur de *o_sum*	Interprétation
1	000000003	$1 + 2 = 3$, le résultat est validé. L'adder fonctionne pour les additions non signées.
2	000000000	$1 + (-1) = 0$, le résultat est validé. L'adder fonctionne pour les additions signées.
3	000000003	$4 - 1 = 3$, le résultat est validé. L'adder fonctionne pour les soustractions non signées.
4	1FFFFFFFF	$(-2) - (-1) = -1$, le résultat est validé. Le 33ème bit prend la valeur de '1' pour indiquer un résultat négatif. L'adder fonctionne donc pour les soustractions signées.

On voit donc que l'adder fonctionne parfaitement. Il réalise les additions et les soustractions pour des entiers signés et non signés. Le résultat est écrit sur 33 bits avec le 33ème bit qui prend la valeur de '1' lorsque le résultat est négatif.

3.4 L'Unité Arithmétique et Logique (ALU)

L'unité arithmétique et logique (ALU) est un module du *mini-riscv* qui réalise des opérations logiques et arithmétique. Le schéma de principe de l'ALU est donné sur la figure 9.

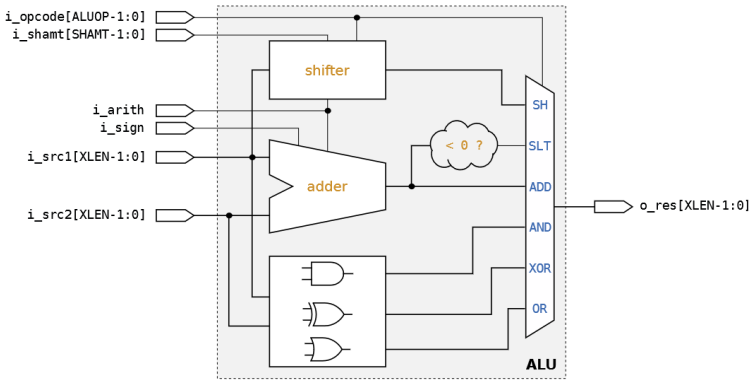

FIGURE 9 – Schéma de principe de l'ALU

Ce module combinatoire est composé de trois blocs principaux : un shifter, un adder et un bloc logique. Il utilise des signaux de commande qui agissent sur deux opérandes : *i_src1* et *i_src2*. Ces deux opérandes sont des signaux 32 bits qui sont reliés à des valeurs qui peuvent provenir des différents registres. En sortie, l'ALU retourne un signal 32 bits : *o_res* qui est le résultat de l'opération effectuée. Pour plus de détails :

— **Le module shifter** consiste à réaliser les différents décalages logiques ou arithmétiques sur la première opérande i_{src1}. Il peut réaliser des décalages à gauche comme à droite en fonction des signaux de contrôle i_{arith} et i_{opcode}. En effet, si i_{arith} vaut 0, le décalage doit être logique (simple décalage en remplissant les nouveaux bits par des 0 si le décalage est à droite) et si il vaut 1, le décalage est arithmétique (les nouveaux bits correspondent au bit de poids fort pour conserver le signe).

— **Le module adder** fait référence au module de l'Adder 32 bits décrit précedemment. Il permet de faire l'addition ou la soustraction des deux opérandes. Le signal de commande *i_arith* permet d'indiquer si il s'agit d'une addition ('0') ou d'une soustraction ('1').
— **Le module logique** réalise quant à lui les différentes opérations OR, AND ou encore XOR sur les opérandes.

Ainsi, l'interface VHDL de l'ALU est codée comme cela :

```
entity riscv_alu is
    port (
        i_arith : in std_logic;
        i_sign : in std_logic;
        i_opcode : in std_logic_vector(ALUOP_WIDTH-1 downto 0);
        i_shamt : in std_logic_vector(SHAMT_WIDTH-1 downto 0);
        i_src1 : in std_logic_vector(XLEN-1 downto 0);
        i_src2 : in std_logic_vector(XLEN-1 downto 0);
        o_res : out std_logic_vector(XLEN-1 downto 0));
end entity riscv_alu;
```

Vérification de l'ALU avec banc de test

Pour tester le comportement de l'ALU, nous avons réalisé un banc de test pour observer les sorties en fonction des tests suivants :

— **Test 1 :** ADD, 0x2, 0x1 $(2 + 1)$.
— **Test 2 :** SRA, 0x2, 0x1 $(2 >> 1)$.
— **Test 3 :** XOR, 0x1, 0x1 $(1 \oplus 1)$.

Le résultat de la simulation est présenté à la figure 10.

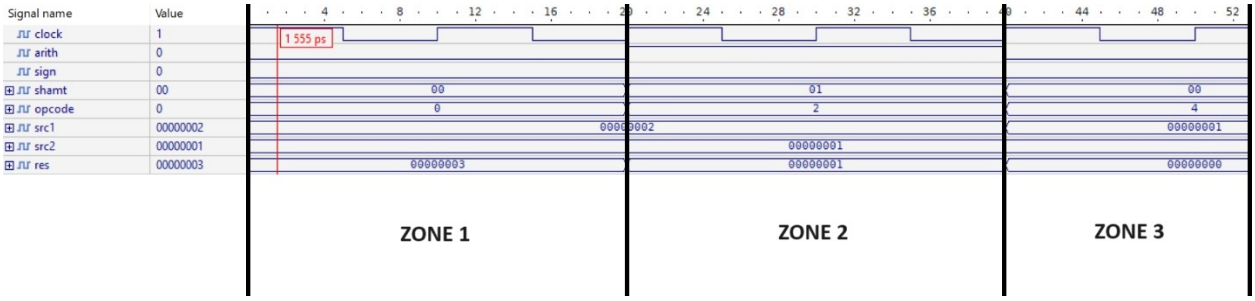

FIGURE 10 – Simulation du banc de test de l'ALU

On peut donc observer les résultats pour valider le comportement du signal de sortie *res*.

Zone	Valeur de *o_sum*	Interprétation
1	00000003	$1 + 2 = 3$, le résultat est validé. L'ALU réalise bien l'opération d'addition.
2	00000001	La décalage arithmétique fonctionne. $2 =' 0010'$ devient $1 =' 0001'$. Le signal *arith* est à '1'.
3	00000000	$'0001' \oplus' 0001' =' 0000'$ donc l'opération de XOR fonctionne.

On voit donc que l'ALU fonctionne parfaitement à travers ces trois exemples d'opérations. Chaque opération utilise un sous-module de l'ALU : l'adder, le shifter et le module logique. Les différentes autres opérations fonctionnent aussi, mais nous avons choisi de limiter les tests à ces 3 opérations pour faciliter la compréhension.

4 Core

Dans cette partie, nous allons nous intéresser au fonctionnement du *core* qui utilise les modules implémentés dans la section précédente. Le core fonctionne sur le principe de *pipelinage* qui s'appuie sur une structure en *pipeline* de 5 étages détaillée ci-dessous. Chaque étape est supposée durer un cycle et permet théoriquement au processeur de traiter 5 instructions en parallèle tout en ayant un débit d'une instruction traitée/seconde. Les étages du *pipeline* sont :

1. **Instruction Fetch :** Lit la prochaine instruction dans la mémoire d'instructions *imem*.
2. **Instruction Decode :** Détermine le type d'instruction à exécuter et lie les opérandes du banc de registres.
3. **Execute :** Exécuter l'instruction et réalise les opérations demandées sur les opérandes.
4. **Memory Access :** Accède à la mémoire pour lire ou écrire une donnée (si besoin).
5. **Write Back :** Ecrit si besoin le résultat dans le banc de registres.

4.1 Instruction Fetch (IF)

La première étape du *pipeline* est le *Fetch (IF)*. Il s'occupe de récupérer la nouvelle instruction à exécuter. Pour cela, il utilise le module PC pour trouver l'adresse de cette nouvelle instruction et intéragit avec la mémoire *imem* d'instructions pour en trouver la valeur. Ce signal doit être sauvegardé dans le registre d'état **IF/ID** pour être communiqué vers l'étage *decode*.
La sortie du PC est sur 32 bits tandis que l'entrée de la mémoire *imem* est sur 9 bits. Il faut donc s'assurer de ne garder que les 9 derniers bits du signal de sortie du PC. On remarque aussi que cet étage communique avec l'étage **EX** qui lui fournit les signaux de commande suivants : *transfert* qui indique si un branchement doit être réalisé, *target* pour l'adresse de l'éventuel branchement, *stall* et *flush* en cas de conflit dans le pipeline. En effet, dans ce cas, il peut être utile d'actionner un *stall* qui stoppe le pipeline pendant un cycle ou un *flush* qui neutralise l'action en cours d'exécution et purge ce dernier.
Le schéma du Fetch est illustré sur la figure 11

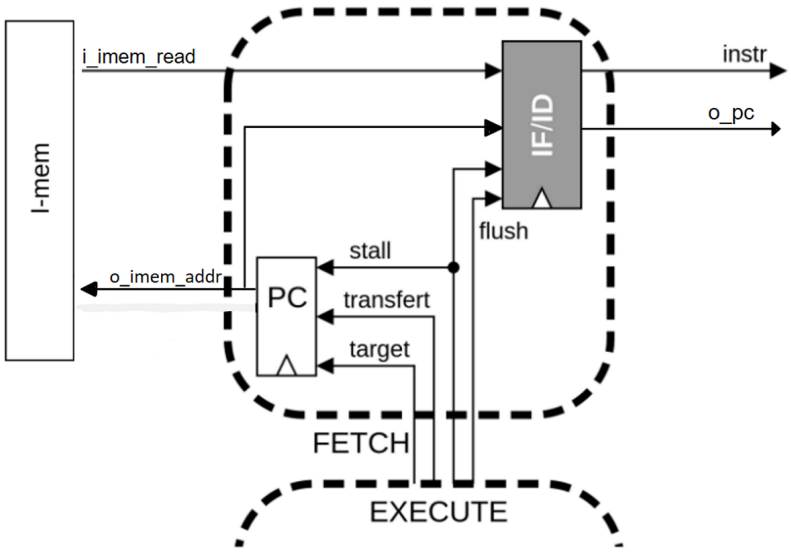

FIGURE 11 – Schéma de principe de l'étage Fetch

4.2 Instruction Decode (ID)

Cette étape du pipeline consiste à décoder l'instruction récupérée par la bascule **IF/ID** et de générer un ensemble de signaux de contrôle et de données à partir de l'information contenue dans le mot d'instruction à exécuter. Elle produit aussi les opérandes pour l'opération à réaliser et transmet tous les signaux utiles dans le registre d'état **ID/EX** permettant l'exécution de l'instruction. Plus précisément, elle permet de :

- Générer les signaux permettant d'identifier l'instruction (*opcode, funct3, funct7*).
- Intéragir avec le banc de registres pour générer les signaux d'adresses permettant de lire les opérandes dans le Banc de registres (*rs1_addr, rs2_addr*), ainsi que le signal d'adresse permettant d'écrire dans le Banc de registres (*rd_addr*). Ce signal devra être transmis à travers tous les étages du pipeline jusqu'à l'étape **WB**.
- Accéder au Banc de registres à partir de ces adresses pour produire les opérandes.
- Générer les valeurs immédiates selon le bon format.
- Générer l'ensemble des signaux de contrôle en fonction de l'instruction : est-ce un branchement ? Un saut ? Doit-on accéder (lecture ou écriture) à la mémoire de donnée ? Doit-on écrire le résultat dans le Banc de registres ? ...

Le schéma de l'étage *decode* est illustré par la figure 12.

4.3 Execute (EX)

Dans cette étape du pipeline, l'instruction est executée grâce aux différents modules de l'ALU ainsi que de l'Adder. En effet, cet étage génère le signal qui correspond au résultat des opérations

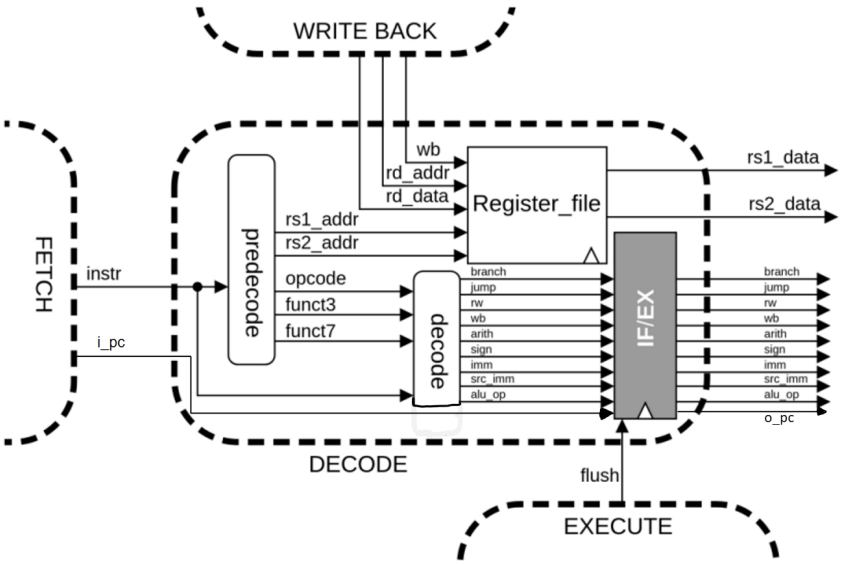

FIGURE 12 – Schéma de principe de l'étage Decode

arithmétiques et logiques réalisées sur les opérandes. Elle génère ainsi, dans le cas d'une instruction *lw* ou *sw* l'adresse de lecture ou d'écriture dans la mémoire *dmem*. Cet étage est aussi responsable de la vérification des conditions de branchement ainsi que de la génération du signal d'adresse de destination. Il communique avec l'étage **ME** via le registre d'état **EX/ME**. Le schéma de l'étape *execute* du pipeline est donné par la figure 13.

4.4 Memory Access (ME)

Le Memory Access est une étape assez particulière du pipeline. En effet, elle n'est réellement utilisée que lors d'une instruction de *Load (lw)* ou de *Store (sw)*. Danc ce cas, elle accède à la mémoire de données *dmem* pour écrire une donnée en mémoire ou récupérer une donnée stockée en mémoire. Dans tous les autres cas, cette étape consiste simplement en une transmission des résultats de l'étape précédente jusqu'à l'étage *Write Back* via la bascule **ME/WB**.
Plus concrètement, lors d'une instruction d'écriture, la donnée (*store_data*) ou le résultat (*alu_result*), est écrite en mémoire. Lors d'une instruction de lecture, la donnée lue est contenue dans le signal *load_data*. Tout cela se fait en fonction des signaux de commande *we* pour l'écriture et *rw* pour la lecture.
Le schéma de l'étage *memory access* est illustré par la figure 14.

4.5 Write Back (WB)

Dans cette dernière étape du pipeline, le mini-riscv écrit le résultat de l'instruction dans le banc de registres à l'adresse *rd_addr*. Ce résultat correspond au signal *alu_result* sauf dans le cas d'une instruction de *Load (lw)* où il correspond à *load_data*. Ce choix est dicté par la valeur du signal de

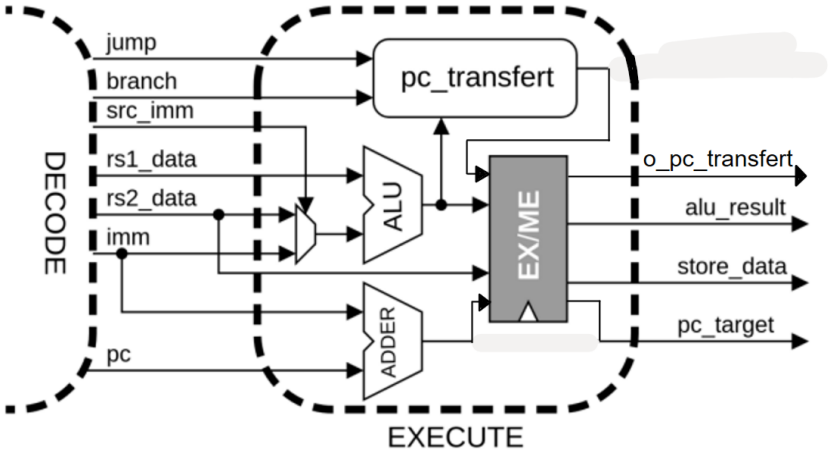

FIGURE 13 – Schéma de principe de l'étage Execute

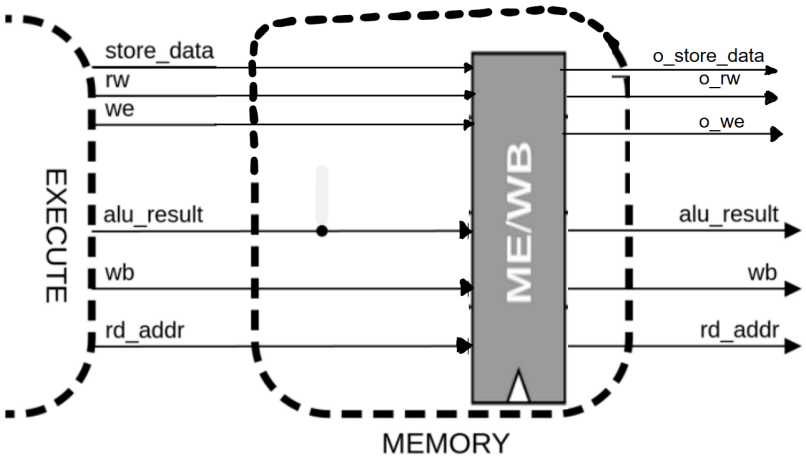

FIGURE 14 – Schéma de principe de l'étage Memory Access

commande *rw*.
Le schéma du write back est illustré par la figure 15.

4.6 Gestion des conflits

Au sein de la séquence d'instructions réalisée par le mini-riscv, il arrive que des conflits se produisent lorsque la séquence d'instructions est modifiée. C'est le cas par exemple lors de l'exécution d'une instruction de branchement. Pour remédier à cela, il existe deux types d'opérations permettant de rétablir l'ordre d'exécution des instructions pour coller avec l'ordre établi par le

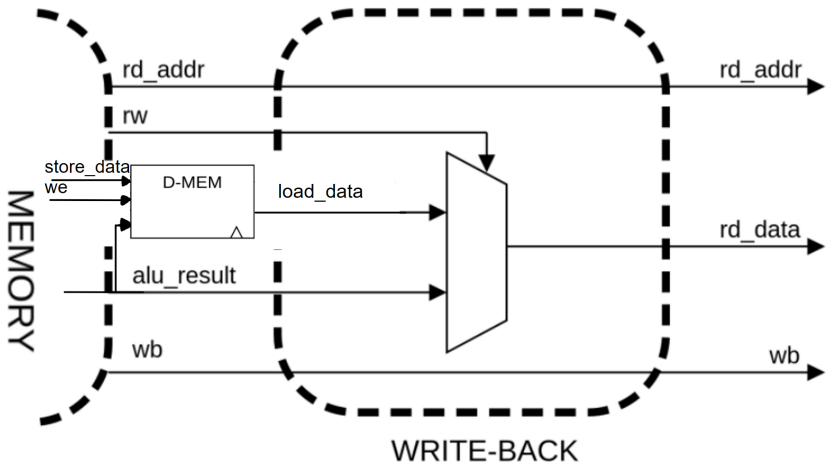

FIGURE 15 – Schéma de principe de l'étage Write Back

programmeur. Ces deux opréations sont : un **stall** qui stoppe le flot d'instructions du pipeline pour un cycle et un **flush** qui purge le pipeline. Ces deux actions peuvent intervenir lors des trois conflits suivants :

— **Un conflits structurel** qui apparaît lorsque deux instructions cherchent à accéder à une même ressource au même moment. Ce problème se produit lorsque deux instructions veulent simultanément lire et écrire une valeur à la même adresse dans le Banc de registres. Pour contrer cela, le Banc de registres a été conçu de telle sorte que la valeur à écrire soit dupliquée dans un registre indépendant.

— **Un conflit de données** qui peut se produire lorsqu'une opération à effectuer dépend du résultat d'une opération précédente. C'est notamment le cas lorsque l'on souhaite utiliser une opérande d'un registre qui n'a pas encore été mis à jour par une instruction précédente. La solution à cela est la technique du *forwarding*, c'est-à-dire le fait de transmettre le résultat d'une instruction des étages **ME** et **WB** vers l'étage **EX**. Mais ce n'est pas forcément suffisant car, lors d'une instruction de *load*, la donnée n'est accessible qu'à la sortie de l'étage **ME**, c'est pourquoi il faut ajouter un *stall* d'un cycle pour attendre que la donnée soit disponible.

— **Un conflit de contrôle** qui se produit lors d'une instruction de branchement conditionnelle par exemple. En effet, dans ce cas, c'est à partir de l'étage de l'**EX** que l'on sait si le branchement doit être pris ou non. Seulement, pendant ce temps, deux instructions ont déjà été chargées dans le pipeline. C'est pour cela que, si le branchement doit être pris, il ne faut pas exécuter ces deux instructions. Elles peuvent donc être retirées grâce à la solution du *flush*. La stratégie utilisée dans le codage du mini-riscv est de se dire que dans le doute, on précharge malgré tout les deux instructions suivantes mais que, dans le cas où le branchement doit être pris, on élimine ces deux instructions.

Dans notre programme, nous n'avons pas pu réaliser la fonction *stall* du pipeline. Malgré tout, la fonction *flush* fonctionne correctement. Pour vider le pipeline, nous remplaçons les deux instructions à effacer par l'instruction **NOP** qui ne fait rien à partir de l'étage **EX**. Pour observer un exemple de cette fonction, la section 5.2 se penche sur la réalisation d'une instruction de branchement.

5 Simulation du core à travers différentes instructions

Dans cette section, nous allons observer étage par étage le comportement du pipeline en fonction de plusieurs instructions :

1. ADD s2, t1, t0
2. BEQ a0, a1, branch_ok
3. SW a0, -4(sp)
4. LW a1, -4(sp)

5.1 ADD s2, t1, t0

Dans un premier temps, nous allons observer la réponse des étages du pipeline de notre mini-riscv à une instruction d'addition. Nous allons ainsi ajouter les registres temporaires t1 et t0 dans le registre de sauvegarde s2. L'opération réalisée est donc : $s2 = t1 + t0$.

Fetch

Pour commencer, on observe l'étage *Fetch* pour l'instruction d'addition. On voit que le signal *imem_read* vaut $0x00530933$, ce qui correspond bien à la valeur d'encodage de l'instruction. *imem_addr* correspond au *fetch_pc* et vaut 40. Cela correspond au $0x10$ème soit 16ème coup d'horloge (par une division par 4 puisque le PC est incrémenté de 4 en 4). L'instruction est donc bien lue et envoyée au coup d'horloge suivant au *Decode*.

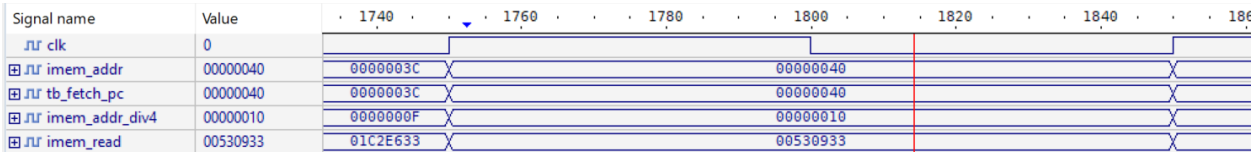

FIGURE 16 – Instruction ADD : Diagramme temporel de l'étage Fetch

Decode

Pour vérifier le fonctionnement du *Decode* pour cette instruction, nous allons observer les différents signaux internes à l'étage. Nous observons donc le coup d'horloge suivant, le 17ème. On voit déjà que le signal *alu_op* correspondant à l'opcode de l'ALU vaut $0x000$, ce qui correspond bien à

l'opcode de l'instruction ADD.
$sign = 1$ ce qui indique une addition signée, mais cela n'a pas d'importance ici.
$arith = 0$, ce qui est cohérent vu que l'on veut faire une addition.
Finalement, les signaux *rs1_addr* et *rs2_addr* valent respectivement 06 et 05, ce qui correspond bien aux adresses des registres à lire t1 et t0.

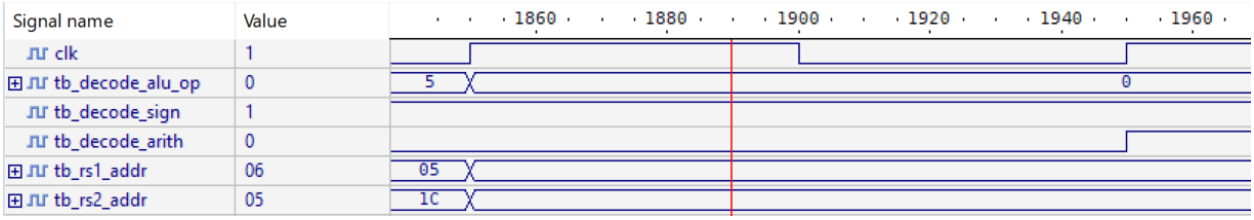

FIGURE 17 – Instruction ADD : Diagramme temporel de l'étage Decode

Execute

Nous nous rendons donc maintenant au coup d'horloge suivant (le 18ème) pour observer la bonne transmission jusqu'à l'étage *Execute*. On observe que *rs1_data* et *rs2_data* valent respectivement $0X7FF$ et $0XFFFFF000$, ce qui correspond aux valeurs contenues dans t1 et t0. On voit donc que l'étage *Decode* a bien transmis les données depuis les adresses correspondantes dans le Banc de Registres. On peut aussi regarder le résultat de l'étage avec notamment le signal de sortie de l'ALU qui vaut : $execute\_alu\_result = 0XFFFFF7FF$, ce qui est bien le résultat de l'addition de $0X7FF$ et de $0XFFFFF000$. L'étage fonctionne correctement.

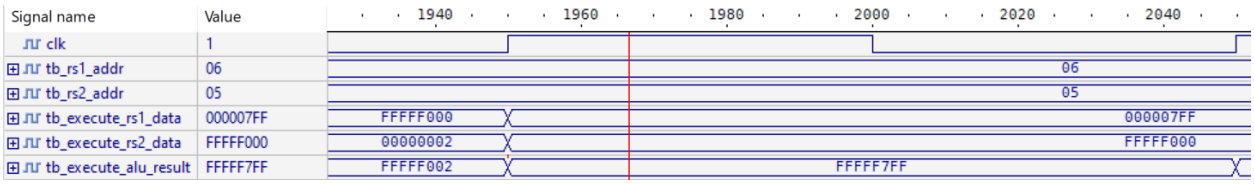

FIGURE 18 – Instruction ADD : Diagramme temporel de l'étage Execute

Memory

Pour vérifier le fonctionnement de cet étage, nous allons observer le coup d'horloge suivant (le 19ème). On voit que le signal de commande *memory_we* est à '0' car ce n'est pas une instruction de "Store" et donc on ne veut pas écrire de résultat directement dans la mémoire. Les autres signaux ne sont pas réellement important car cet étage ne fait pas grand chose puisqu'on ne souhaite pas accéder ou écrire dans la mémoire.
Write Back

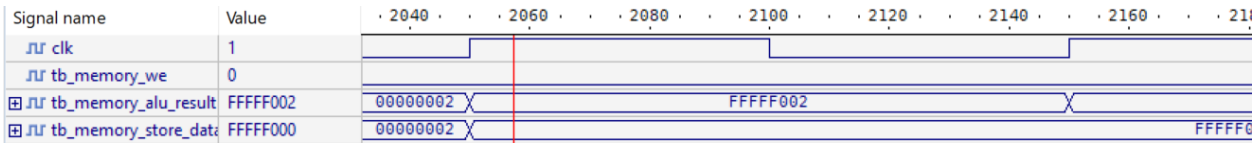

FIGURE 19 – Instruction ADD : Diagramme temporel de l'étage Memory

On observe finalement l'étage de *Write Back*. On est au coup d'horloge suivant (le 20ème). On peut tout d'abord observer que le signal de contrôle *write_back_wb* est à '1' car il faut écrire le résultat dans le registre s2. Ce signal est transmis depuis l'étage *Execute* et permet d'écrire à l'adresse *write_back_rd_addr* qui est l'adresse de s2. La valeur de ce signal est $0x12$, ce qui est validé puisque s2 est le 18ème registre, soit $0x12$ en hexadécimal. Enfin, la donnée à écrire dans ce registre est *write_back_rd_data* qui vaut bien le résultat de l'ALU.

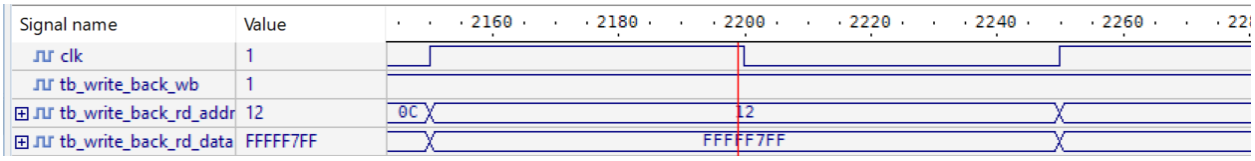

FIGURE 20 – Instruction ADD : Diagramme temporel de l'étage Write Back

On peut donc dire que notre processeur mini-riscv fonctionne parfaitement pour le passage dans le pipeline de cette première instruction.

5.2 BEQ a0, a1, branch_ok

Nous allons maintenant observer la réponse des étages du pipeline de notre mini-riscv à une instruction de branchement conditionnelle. Cette instruction vise à réaliser un saut à l'adresse $PC+3$. C'est-à-dire de se brancher 3 instructions plus loin dans la suite d'instructions si la condition est satisfaite. Ici, la condition est que les registres a0 et a1 soit égaux, ce qui est le cas.

Fetch

Pour commencer, on observe que le PC est à l'adresse *imem_addr*, ce qui correspond à $0x8C = 140$, soit la 35ème instruction si on divise par 4. Cette donnée est importante puisqu'elle nous permettra de vérifier si le branchement se fait à l'instruction $35 + 3 = 38$ème instruction. On voit donc que sur ce coup d'horloge, la valeur de *imem_data* est $0x00B50663$, ce qui correspond bien à la valeur d'encodage de l'instruction à effectuer.

Decode

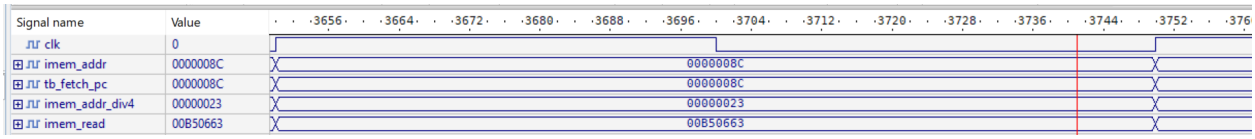

FIGURE 21 – Instruction BEQ : Diagramme temporel de l'étage Fetch

On regarde maintenant le coup d'horloge suivant pour l'étage *Decode*. On constate tout d'abord que le signal *branch* est à '1', ce qui veut dire que le décodage d'un branchement a bien été pris en compte.
Les signaux $aluop = 0x000$ (ADD ou SUB) et $arith = 1$ indiquent que l'on transmet bien à l'*Execute* une soustraction des deux registres contenus dans les adresses *rs1_addr* et *rs2_addr*. Le but est de faire une comparaison entre les valeurs des deux registres. On peut vérifier que *rs1_addr* et *rs2_addr* valent respectivement $0x0A$ et $0x0B$, ce qui est bien l'adresse des registres a0 et a1 (registre n°10 et n°11). Le signal de commande *branch* est là pour indiquer à l'*Execute* de réaliser le branchement au besoin si la condition de branchement est respectée.

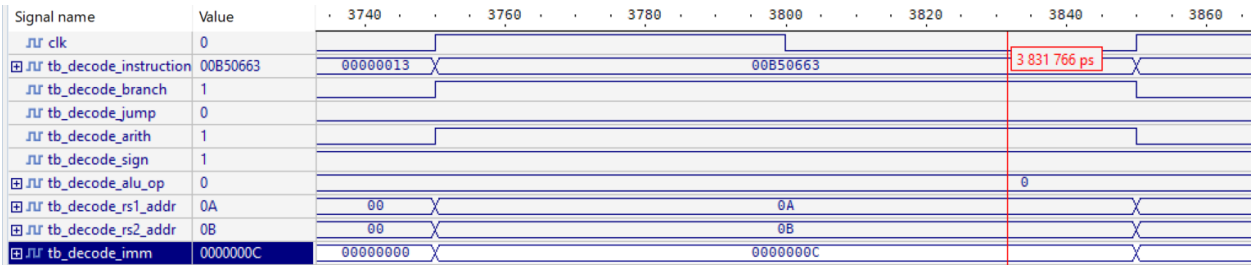

FIGURE 22 – Instruction BEQ : Diagramme temporel de l'étage Decode

Execute

Pour l'étage *Execute*, on observe que les données *rs1_data* et *rs2_data* récupérées à la sortie du Banc de Registres grâce aux adresses *rs1_addr* et *rs2_addr* valent $0X0FFF80F0$, elle sont donc bien égales et la condition de branchement est satisfaite. Le signal *alu_result* est nul et confirme bien cela.
Le signal *pc_transfert* passe alors à '1' pour indiquer qu'un branchement doit être fait.
Enfin, la valeur lue de la valeur immédiate est $imm = 0xC$ soit 12, ce qui correspond à un saut de 3 adresses si l'on prend en compte la division par 4. On voit aussi que le signal *pc_target* passe à $0x98$ soit le résultat de $pc + imm = 0x8C + 0x08 = 0x98$. Tout est encore bien cohérent pour le fonctionnement de l'*Execute*.
Visualisation des 5 étages

On voit donc que l'instruction est bien réalisée dans le pipeline. Les étages *Memory* et *Write*

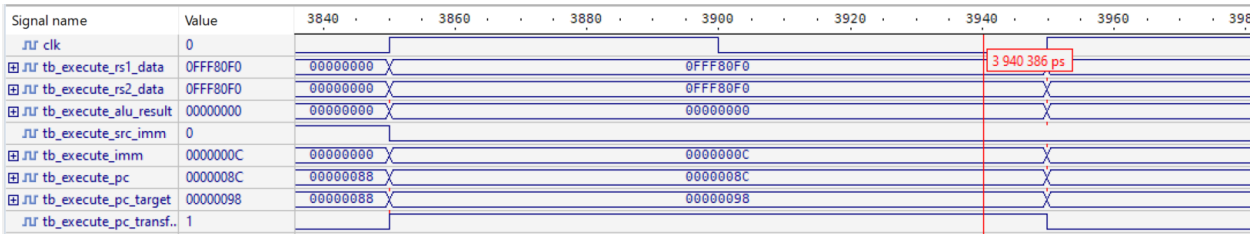

FIGURE 23 – Instruction BEQ : Diagramme temporel de l'étage Execute

Back ne sont pas intéressants puisqu'ils n'interviennent pas dans la réalisation de l'instruction. On peut aussi voir qu'après 2 cycles (par rapport à *Execute*), on a *imem_addr* qui vaut $0x98$ et *decode_instruction* qui est égal à $0x00001013$, ce qui correspond à l'instruction **NOP**. Il y a donc bien un *flush* des étages *Fetch* et *Decode* lorsque *pc_transfert* vaut '1'. *pc_transfert* est équivalent à un *flush*. La figure 24 montre le chronogramme des 5 étages pour l'instruction BEQ.

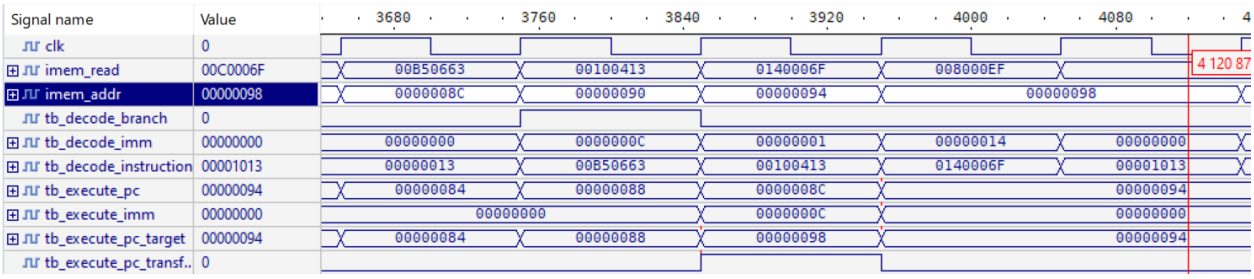

FIGURE 24 – Instruction BEQ : Diagramme temporel des 5 étages

5.3 SW a0, -4(sp)

Nous allons maintenant observer le comportement de notre mini-riscv avec une instruction de stockage "Store". Cette instruction a pour but d'écrire la donnée stockée dans le registre a0 à l'adresse contenue dans le Stack Pointer (sp) à laquelle on ajoute la valeur immédiate -4.

Fetch

Dans un premier temps, on peut observer l'étage du *Fetch*. Cet étage nous donne le signal *imem_read* qui vaut $0xFEA12E23$, ce qui correspond bien à l'encodage de l'instruction à exécuter.

Decode

Dans l'étage *Decode*, on peut observer que les signaux *rs1_addr* et *rs2_addr* valent respectivement $0x02$ et $0x0A$, ce qui correspond aux registres SP et a0 (registres n°2 et n°10 dans le Banc de Registres).

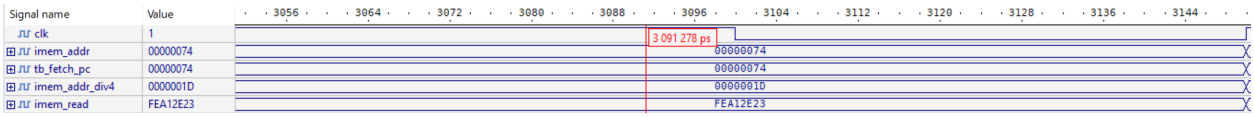

FIGURE 25 – Instruction SW : Diagramme temporel de l'étage Fetch

Le signal *decode_imm* vaut $0xFFFFFFFC$, soit -4 en binaire et correspond bien à la valeur immédiate de l'instruction.
Enfin, l'opcode de l'ALU, *alu_alu* est à '0', ce qui signifie que l'opération à exécuter est une addition.

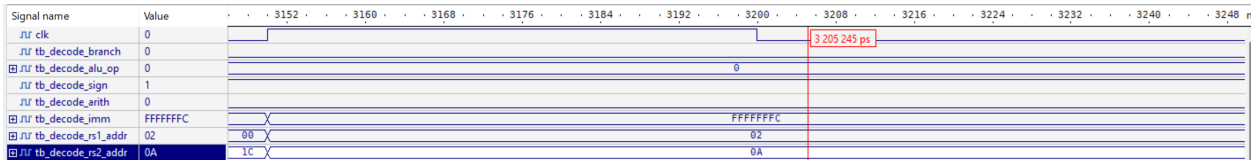

FIGURE 26 – Instruction SW : Diagramme temporel de l'étage Decode

Execute

Au coup d'horloge suivant, nous pouvons passer à l'observation de l'*Execute*. On voit que les données transmises par le banc de registres, c'est-à-dire les données contenues aux adresses *rs1_addr* et *rs2_addr*, sont *rs1_data* $= 0x000007FC$ et *rs2_data*$= 0x0FFF84F0$. La première donnée correspond à l'adresse pointée par le Stack Pointer SP. La seconde correspond à la valeur qui va être écrite en mémoire, elle est donc transmise à l'étage *Memory* via le signal *store_data*.
Le signal *execute_src_imm* passe à '1', ce qui signifie que l'on va utiliser une valeur immédiate comme deuxième opérande dans l'ALU (à la place du deuxième registre).
Enfin, le signal *alu_result* vaut $0X000007F8$, ce qui correspond à la somme $sp+(-4) = 0x000007FC + 0xFFFFFFFC$. On voit donc que le résultat de l'ALU contient l'adresse du Stack Pointer à laquelle on a enlevé la valeur immédiate 4.

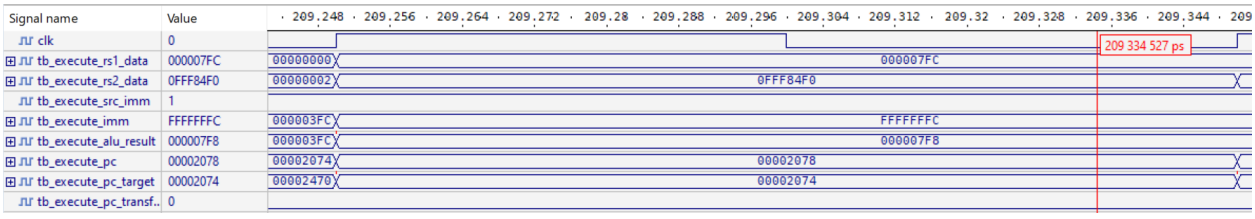

FIGURE 27 – Instruction SW : Diagramme temporel de l'étage Execute

Memory

Pour cet étage, on voit que le signal de contrôle *we* passe à '1', ce qui signifie bien que l'on veut écrire une donnée dans la mémoire de donnée *dmem*. Les signaux *store_date* et *alu_result* contiennent respectivement la donnée du registre a0 ainsi que l'adresse à laquelle l'écrire dans la mémoire de données. C'est signaux sont les mêmes que pour l'étage précédent.

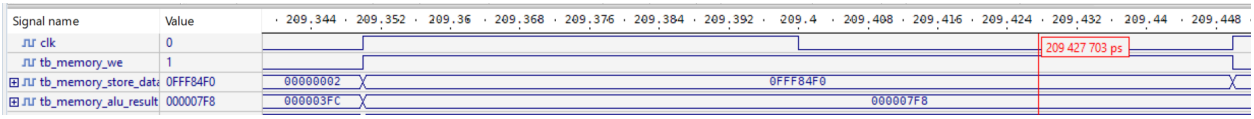

FIGURE 28 – Instruction SW : Diagramme temporel de l'étage Memory

Write Back

Pour le dernier étage du pipeline, on peut voir que les signaux de contrôle *wb* et *rw* sont à 0 puisque l'on ne veut pas écrire dans un registre du Banc de Registre ou lire une valeur depuis la mémoire de données.
Le signal *dmem_addr* contient l'adresse d'écriture $0x000007F8$ et *dmem_write* contient la donnée $0x0FFF84F0$. On a bel et bien écrit la valeur souhaitée à la bonne adresse.

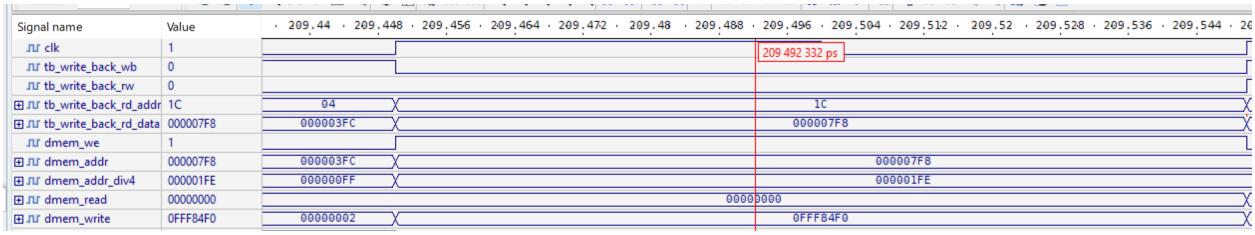

FIGURE 29 – Instruction SW : Diagramme temporel de l'étage Write Back

Notre mini-riscv fonctionne donc parfaitement pour l'opération de stockage. Nous allons donc maintenant vérifier cela pour l'opération inverse de lecture.

5.4 LW a1, -4(sp)

Pour finir, la dernière instruction que nous allons vérifier est une instruction de lecture "Load". Cette instruction permet de charger la donnée contenue à l'adresse $SP + (-4)$ dans le registre a1.

Fetch

Le premier étage du pipeline nous permet d'observer l'encodage de l'instruction. En effet, le *Fetch* nous donne le signal *imem_read* qui vaut $0xFFC12583$. Cela est donc en adéquation avec l'encodage de l'instruction souhaitée.

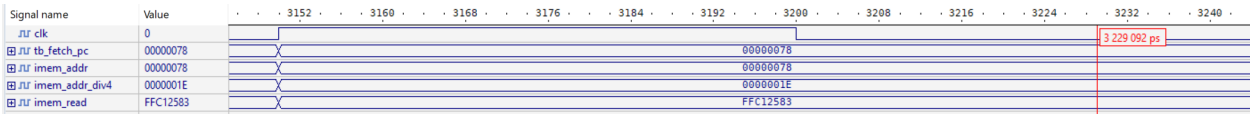

FIGURE 30 – Instruction LW : Diagramme temporel de l'étage Fetch

Decode

Au niveau du *Decode*, cela se passe relativement comme pour l'instruction précédente, *rs1_addr* pointe vers l'adresse du Stack pointer (i.e. $0x02$ car c'est le registre n°2), *decode_imm* contient la valeur immédiate -4, ou $0xFFFFFFFC$ en hexadécimal (*rs2_addr* n'a pas d'importance pour cette instruction). Cet étage décode donc l'instruction et envoie au Banc de Registres l'adresse du Stack Pointer pour y récupérer la donnée.

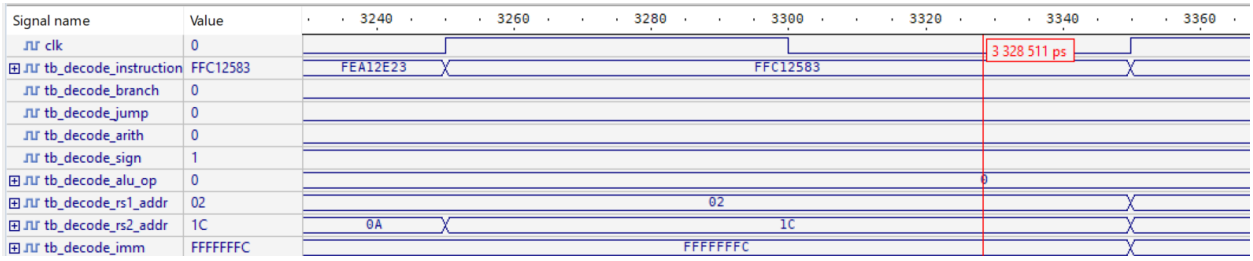

FIGURE 31 – Instruction LW : Diagramme temporel de l'étage Decode

Execute

Dans cet étage, le signal *rs1_data* représente la donnée contenue dans le Stack Pointer, $0x07FC$, cela correspond à l'adresse pointée par le registre SP. On a donc la bonne donnée à la sortie du Banc de Registres.
Le signal *src_imm* est à '1' car l'on souhaite utiliser la valeur immédiate comme deuxième opérande de l'ALU. On voit aussi que la valeur immédiate s'est correctement transmise sur le signal *execute_imm* qui vaut bien $0xFFFFFFFC = -4$. Au final, la sortie de l'ALU, *alu_result* vaut : $0x07F8$, soit $0x07FC + 0xFFFFFFFC$. On a donc bien ajouté -4 à l'adresse de lecture dans la mémoire de données.
Memory

Pour l'étage de *Memory*, au coup d'horloge, le signal de commande *we* passe à '0' car l'on ne souhaite pas écrire. Il n'y a pas grand chose d'autre qui se passe sur ce coup d'horloge puisque l'on souhaite récupérer une donnée dans la mémoire *dmem*. Avec la latence d'un cycle, on observe le résultat de l'interface avec la *dmem* au coup d'horloge suivant.
Write Back

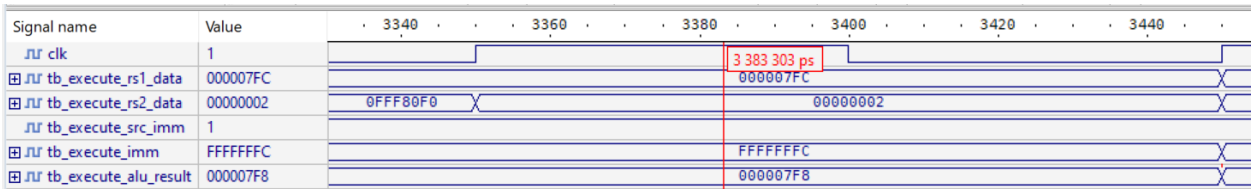

FIGURE 32 – Instruction LW : Diagramme temporel de l'étage Execute

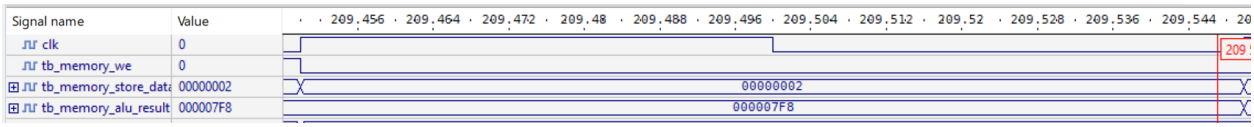

FIGURE 33 – Instruction LW : Diagramme temporel de l'étage Memory

Pour le *Write Back*, le signal de contrôle *rw* est à '1' car on souhaite lire une donnée en mémoire. Le signal *rd_addr* est à $0x0B$ qui donne l'adresse du registre dans lequel écrire la donnée (11ème registre).
Le signal *rd_data* est égal à *load_data* et à *dmem_read*. Il contient la donnée, $0x0FFF80F0$, lue depuis la mémoire de données *dmem*.
De plus, le signal *alu_result* est égal à *dmem_addr* et vaut l'adresse de la donnée lue en mémoire.

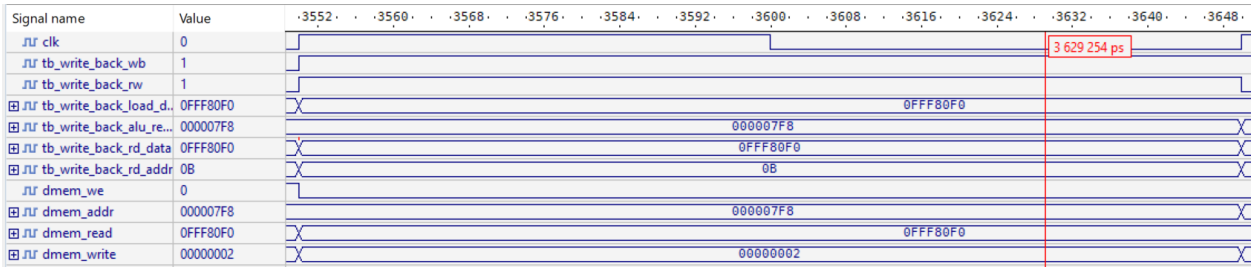

FIGURE 34 – Instruction LW : Diagramme temporel de l'étage Write Back

On voit donc maintenant que l'instruction de "load" est correctement effectuée à travers le pipeline. Nous avons donc vu, à travers les quatre instructions précédentes, que le mini-riscv que nous avons implémenté fonctionne. Que ce soit à travers des instructions d'addition, de branchement ou encore de lecture/écriture, les signaux internes aux étages du pipeline communiquent correctement et permettent le bon fonctionnement du système. Nous avons aussi implémenté le fonction de *flush* pour résoudre les conflits de contrôle lors d'un branchement.

6 Conclusion

En somme, nous avons vu, à travers ce laboratoire, l'implémentation d'un processeur mini-riscv. Nous avons observé son architecture et notamment la construction de son jeu d'instructions ainsi que de l'interface avec la mémoire et les registres. Au niveau inférieur, nous avons implémenté différents modules pour nous permettre de réaliser un jeu varié d'instructions (comme par exemple un banc de registres, une ALU ou un Adder). Enfin, nous avons implémenté la micro-architecture en pipeline du mini-riscv dans le but d'effectuer les différentes étapes de la réalisation d'une instruction.

Pour aller plus loin dans la démarche de conception, nous pouvons nous pencher sur l'implémentation de ce processeur, opération réalisable notamment avec des outils numériques comme *Cadence*. Nous avons cependant le regret de ne pas avoir eu le temps de pouvoir nous y pencher sérieusement.

***N.B.** : Certaines figures ainsi que certains passages sont tirés de la documentation du microprocesseur mini-riscv fournie au préalable du laboratoire 2 par messieurs Yvon Savaria, Mickaël Fiorentino, Justin Pabot et Timothée Trembly.*

7 Annexes

U-imm[31:12]				rd	0110111	**LUI**
J-imm[20 | 10:1 | 11 | 19:12]				rd	1101111	**JAL**
I-imm[11:0]		rs1	000	rd	1100111	**JALR**
B-imm[12|10:5]	rs2	rs1	000	B-imm[4:1|11]	1100011	**BEQ**
I-imm[11:0]		rs1	010	rd	0000011	**LW**
S-imm[11:5]	rs2	rs1	010	S-imm[4:0]	0100011	**SW**
I-imm[11:0]		rs1	000	rd	0010011	**ADDI**
I-imm[11:0]		rs1	010	rd	0010011	**SLTI**
I-imm[11:0]		rs1	011	rd	0010011	**SLTIU**
I-imm[11:0]		rs1	100	rd	0010011	**XORI**
I-imm[11:0]		rs1	110	rd	0010011	**ORI**
I-imm[11:0]		rs1	111	rd	0010011	**ANDI**
0000000	shamt	rs1	001	rd	0010011	**SLLI**
0000000	shamt	rs1	101	rd	0010011	**SRLI**
0100000	shamt	rs1	101	rd	0010011	**SRAI**
0000000	rs2	rs1	000	rd	0110011	**ADD**
0100000	rs2	rs1	000	rd	0110011	**SUB**
0000000	rs2	rs1	001	rd	0110011	**SLL**
0000000	rs2	rs1	010	rd	0110011	**SLT**
0000000	rs2	rs1	011	rd	0110011	**SLTU**
0000000	rs2	rs1	100	rd	0110011	**XOR**
0000000	rs2	rs1	101	rd	0110011	**SRL**
0100000	rs2	rs1	101	rd	0110011	**SRA**
0000000	rs2	rs1	110	rd	0110011	**OR**
0000000	rs2	rs1	111	rd	0110011	**AND**

FIGURE 35 – Liste des instructions supportées par le mini-riscv