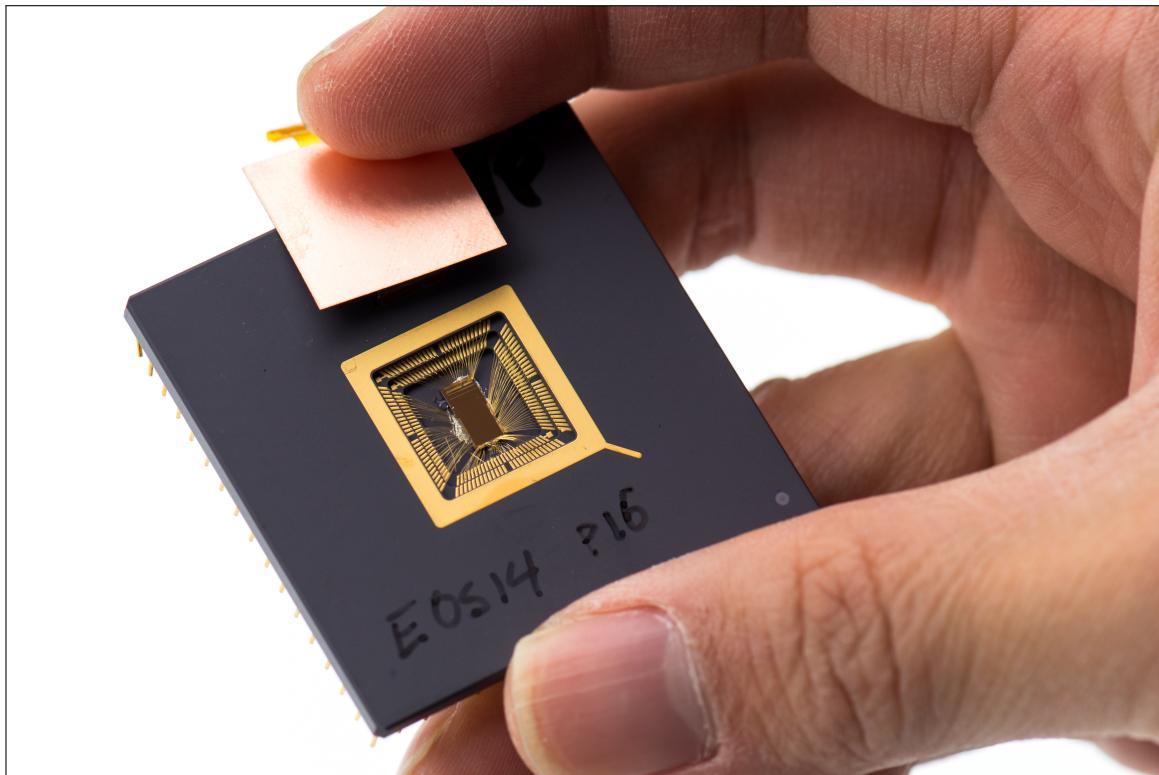




## Architecture des processeurs 2

Pipeline d'un processeur scalaire RISCV



Martin RABIER - Tristan PANHELLEUX

ISMN EI23

# Sommaire

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Architecture pipeline</b>	<b>3</b>
2.1	Outils et compilation . . . . .	3
2.2	Appropriation de l'architecture . . . . .	3
2.2.1	Architecture de base . . . . .	3
2.2.2	Introduction aux dépendances . . . . .	5
2.3	Application . . . . .	7
<b>3</b>	<b>Gestion des dépendances</b>	<b>9</b>
3.1	Exécution d'une multiplication . . . . .	9
3.1.1	Implémentation du pseudo-code . . . . .	9
3.2	Correction . . . . .	10
3.2.1	Correction logicielle . . . . .	10
3.2.2	Correction matérielle : Dépendance de données . . . . .	10
3.2.3	Correction matérielle : Dépendance de contrôle (jump) . . . . .	12
3.2.4	Correction matérielle : Dépendance de contrôle (branch) . . . . .	13
<b>4</b>	<b>Mémoire cache</b>	<b>14</b>
4.1	Cache d'instruction en lecture . . . . .	14
4.1.1	Cache d'instruction 1 voie . . . . .	14
4.1.2	Cache d'instruction associatif 2 voies . . . . .	16
<b>5</b>	<b>Conclusion</b>	<b>16</b>
<b>6</b>	<b>Annexes</b>	<b>17</b>
A	Chronogramme dépendance de données . . . . .	17
B	Chronogramme dépendance de contrôle . . . . .	17

## 1-Introduction

Après avoir, en première année, appréhendé l'architecture d'un processeur RISCV monocycle dans le cours *Architecture des processeurs 1*, puis le langage SystemVerilog avec le cours *Conception d'un système numérique*, nous allons allier ce langage avec la première discipline afin d'approfondir l'architecture d'un processeur RISCV pour voir de quelle manière on peut implémenter du pipeline.

Pour ce faire, nous utiliserons pour la compilation l'outil open source de compilation RISCV GCC [1], le logiciel de simulation Modelsim ainsi que les ressources mises à disposition dans le cadre du cours (support de travaux dirigés, GreenCard RISCV, ...). Enfin, l'objectif du cours étant uniquement l'implémentation des mécanismes de pipeline, les fichiers SystemVerilog nécessaires, majoritairement réalisés par M. Agoyan, sont fournis. Ces fichiers seront uniquement modifiés selon les exigences d'une architecture pipeline.

Le rapport suivra un plan linéaire respectant l'ordre dans lequel les tâches ont été demandées et les questions posées. Par souci de clarté, certaines questions seront regroupées avec une réponse commune. Ces regroupements ne seront pas numérotés afin de rendre le rapport plus fluide. De plus, nous nous autorisons à ajouter du contenu supplémentaire jugé pertinent pour étayer notre démarche.

## 2-Architecture pipeline

### 2.1 Outils et compilation

Pour ne pas être dépendant du réseau de l'école lors de notre travail personnel, nous avons fait le choix de travailler sur une machine virtuelle locale.

Aussi il peut être nécessaire de définir un chemin pour le compilateur RISCV GCC, ce qui n'est pas nécessaire s'il est déjà présent sur votre machine. Dans notre cas, on se place dans le dossier racine et on exécute :

```
$ source path.txt
```

Ensuite pour la compilation et la simulation du processeur, on se place dans le dossier *sim*, on exécutera la ligne suivante qui compilera à la fois le *build.sh* du répertoire *firmware* et celui du répertoire *sim*, simplifiant la simulation et lançant directement Modelsim s'il est téléchargé sur votre machine.

```
$ source dbuild.txt
```

### 2.2 Appropriation de l'architecture

#### 2.2.1 Architecture de base

Pour commencer et comme indiqué en introduction l'architecture principale nous est déjà fourni. Aussi pour commencer le travail on s'approprie cette dernière en identifiant l'organisation des fichiers, l'architecture associée ainsi que le rôle de chacun des signaux.

On commence par identifier l'organisation du processeur en identifiant le circuit **top-level** ainsi que les sous-circuits qui le compose. L'organisation du processeur est illustrée *figure 1*. On identifie le top-level comme étant le fichier **RV32i\_soc**. Ce module instancie lui-même une part **RV32i\_pipeline\_top** qui correspond au processeur et d'autre part **wsync\_mem** qui gère la partie mémoire.

RV32i\_pipeline\_top instancie un **controlpath** et un **datapath**. Enfin chacun des modules instancie et utilise au besoin des unités élémentaires comme l'*Arithmetic Logic Unit (ALU)*, des registres *regfile*, des *multiplexeurs*...

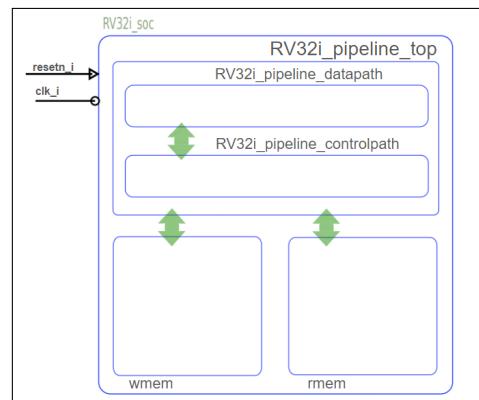


FIGURE 1 – Schéma organisation

### Tableau des modules et leur rôle

Nom du module	Rôle
SOC	Permet de lier la partie processeur (calcul) et la mémoire. Instancie les modules RV32i_pipeline_top et wsync_mem.
RV32i_top	Gère les signaux et connecte les modules entre eux sans en instancier de nouveaux.
wsync_mem	Stocke les données traitées par le processeur en mémoire à l'adresse prévue.
Datapath	Instancie deux sous-modules : les ALU (réalisation des opérations logiques selon les entrées) et les registres (manipulation des données dans le processeur).
ALU	Réalise des opérations logiques et arithmétiques selon les signaux d'entrée.
Registres	Permettent de manipuler et de stocker temporairement les données dans le processeur.

TABLE 1 – Résumé des modules et de leurs rôles

Concernant le fonctionnement global du module et plus précisément le nombre d'étage qui le compose, on utilise le schéma d'architecture fournit dans le premier cours [2] , sur lequel s'appuie l'implémentation du processeur.

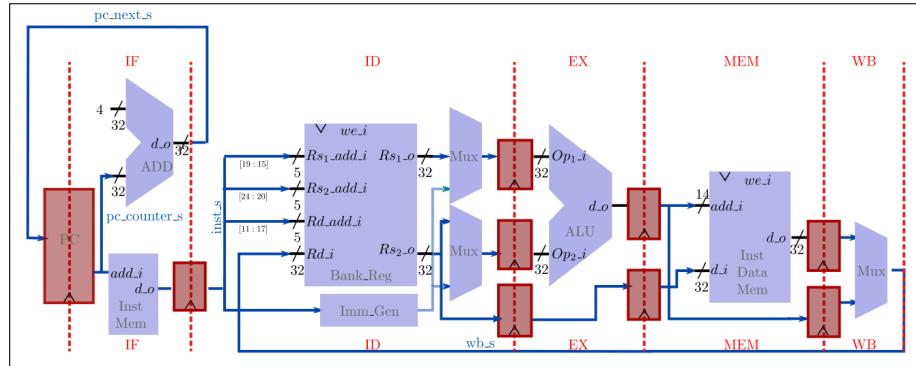


FIGURE 2 – Architecture processeur RISC-V

On retrouve les 5 étages étudié en cours d'architecture des processeurs 1, dont les noms et définitions sont détaillés ci dessous.

**INSTRUCTION FETCH** : Charge les instructions depuis la mémoire d'instruction.

**INSTRUCTION DECODE** : Interprète les instructions et adresse les registres nécessaires.

**EXECUTE** : L'ALU (Arithmetic Logic Unit) réalise l'opération logique souhaitée.

**MEMORY** : On lit ou on écrit dans la mémoire principale si l'opération le nécessite.

**WRITE BACK** : Écriture du résultat dans le registre.

Afin de comprendre la manière dont l'information est traitée et transmise on analyse ensuite les signaux utilisés par ces modules. On se concentre particulièrement sur les signaux à destination du datapath, les signaux à destination de la mémoire d'instruction et enfin les signaux à destination de la mémoire de données. Par soucis de clarté, on détaille le rôle de tous ces signaux dans la table 2.

Avant de classer les signaux, on définit le rôle de chacune des deux mémoires. Tout d'abord, la mémoire d'instruction. Elle contient les instructions machines qui devront être exécutées par le processeur. Elles sont récupérées une à une grâce au programme counter (PC) qui adresse la mémoire. Elle impose aux signaux détaillés ci-après. En effet la valeur de  $we\_i$  sera par exemple systématiquement fixée à 1'b0 puisqu'on lit uniquement

l'instruction. Les autres valeurs fixes seront détaillées dans le tableau *Instruction Memory*. D'autre part, la mémoire de données. Celle-ci est une mémoire temporaire qui stocke toutes les informations nécessaires à l'exécution des instructions machines et est couramment appelée mémoire RAM (Random Access Memory). On y retrouve différentes informations comme des opérandes ou des résultats intermédiaires. Elle est accessible aussi bien à l'écriture qu'à la lecture. Ainsi, contrairement à la mémoire d'instruction on a pas de valeur fixe pour les données de destination.

Signal	Description
<b>Data Memory</b>	
clk_i	Signal d'horloge
we_i	Signal d'écriture (write enable)
re_i	Signal de lecture (read enable)
d_i	Données d'entrée
be_i	Masquage des bits
add_i	Adresse de la mémoire utilisée pour l'accès aux données de la mémoire

Signal	Description
<b>Instruction Memory</b>	
clk_i	Signal d'horloge
we_i	Signal d'écriture (write enable, fixé à 0 car on ne fait que lire la mémoire d'instruction)
re_i	Signal de lecture (read enable)
d_i	Données d'entrée (fixé à 32'h0 pour la lecture de données)
be_i	Masquage des bits (fixé à 4'b1111 pour lire tous les octets)
add_i	Adresse de la mémoire
valid_o	Indique si les données lues dans la mémoire d'instruction sont valides

Signal	Description
<b>Entrées du Control Path</b>	
clk_i	Signal d'horloge
resetn_i	Signal de réinitialisation
alu_zero_i	Indicateur de comparaison (zéro)
alu_lt_i	Indicateur de comparaison (inférieur)
instruction_i	Instruction actuelle envoyée au processeur pour exécution

TABLE 2 – Regroupement et explication des signaux

### 2.2.2 Introduction aux dépendances

Les signaux de la formes  $inst\_X\_r$  sont respectivement associé aux étages du processeur : decode, execute, memory et write-back. Il permettent d'effectuer pour chaque étage une vérification stall condition. Si l'on remarque un conflit pouvant entraîner des dépendances de données ou de contrôle, l'opération conditionnelle retourne alors 32'h00000013, ce qui correspond à l'opération NOP.

Pour identifier les signaux gérant l'index de destination, on s'appuie tout d'abord sur la GreenCard RISCV afin d'identifier les bits correspondant [3].

CORE INSTRUCTION FORMATS														
	31	27	26	25	24	20	19	15	14	12	11	7	6	0
<b>R</b>						funct7		rs1		funct3		rd		opcode
<b>I</b>						imm[11:0]		rs1		funct3		rd		opcode
<b>S</b>						imm[11:5]		rs1		funct3	imm[4:0]			opcode
<b>SB</b>						imm[12 10:5]		rs1		funct3	imm[4:1 11]			opcode
<b>U</b>											imm[31:12]		rd	opcode
<b>UJ</b>											imm[20 10:1 11 19:12]		rd	opcode

FIGURE 3 – Format des instruction RISCV

Aussi on remarque d'après la *figure 3* que l'adresse du registre de destination sont les bits [11 :7] de l'instruction. On retrouve bien cette adresse dans le fichier '*RV32i-pipeline\_controlepath.sv*'. On déclare une variable adresse pour chaque étage. L'adresse se propage alors de la manière suivante :

**Étape Decode** : L'adresse du registre de destination est extraite directement de l'instruction (d'après *figure 3*) avec la ligne suivante :

```
assign rd_add_dec_w = instruction_i[11:7];
```

**Étape Execute** : L'adresse est propagée depuis l'étape Decode et assignée au signal correspondant, extrait de *inst\_exec\_r[11:7]*.

```
assign rd_add_exec_w = inst_exec_r[11:7];
```

**Étape Memory** : L'adresse est extraite à partir du signal *inst\_mem\_r[11:7]* pour cette étape du pipeline.

```
assign rd_add_mem_w = inst_mem_r[11:7];
```

**Étape Write Back** : Pour la dernière étape, l'adresse est assignnée à la variable *rd\_add\_wb\_w*. Une assignation supplémentaire est effectuée pour relier ce signal à la sortie finale *rd\_add\_o*.

```
assign rd_add_wb_w = inst_wb_r[11:7];
assign rd_add_o = rd_add_wb_w;
```

L'adresse *rd\_add\_o* correspond à la sortie qui sera attribuée à un banc de registre. On passe ensuite sur le fichier *regfile.sv* où la variable *rd\_add\_i* correspond à la variable *rd\_add\_o* du module précédent. L'adresse est finalement utilisée pour mettre à jour le registre souhaité.

Aussi, nous avons vu de quelle manière l'index d'instruction est géré, ainsi que la manière dont on peut effectuer une vérification pour générer un signal *stall* à chaque étage. On se penche alors sur la manière dont ce signal *stall* est généré.

Pour l'instant le signal *stall* est généré comme un signal *logic* mais n'est pas conditionnée par une équation booléenne pour sa mise à jour, ainsi le signal *Stall\_w* est toujours à 0 d'après la ligne de code suivante :

```
assign stall_w = 1'b0;
```

Cependant, si la logique booléenne du *stall* n'est pas implémentée, le cas où le signal est actif est tout de même pris en compte. Dans ce cas, la gestion des signaux concernés est assurée par la ligne suivante :

```
assign inst_dec_r = (stall_w == 1'b1) ? 32'h00000013 : instruction_i;
```

Lorsque le signal `stall_w` est actif, c'est-à-dire lorsqu'il vaut 1, la variable `inst_dec_r` prend la valeur `32'h00000013`, ce qui correspond à l'insertion d'un NOP. Cette insertion permet d'éviter les conflits et garantit le bon fonctionnement du processeur face à ce type de dépendance.

Enfin, on s'intéresse à la structure de la mémoire déjà implémentée. Tout d'abord, la mémoire de données `dmem` du composant `wsync_mem`. On définit la mémoire de données comme un tableau de taille `SIZE` de mots de 32 bits. Sachant  $SIZE = 4096$ , on a donc  $32 \times 4096 = 131072 = 2^{18}$  bits. Ainsi, `dmem` correspond à 16 Ko. On retrouve l'adresse de cette mémoire `32'h00010000` dans le fichier `RV32I_soc`.

Comme pour la mémoire de donnée, la mémoire d'instruction est un tableau de `SIZE` ligne de mots de 32 bits. Par le même calcul que ci-dessus on a donc une mémoire d'instruction de 16 Ko. On retrouve également dans le fichier `RV32I_soc` son adresse `32'h00000000`.

## 2.3 Application

Après avoir compris l'architecture du processeur, on effectue les premières simulation et on tente de prendre en main le système.

Pour commencer, on exécute le programme `exo1.S`. En analysant le programme le comportement attendu est le suivant : on souhaite charger des valeurs allant de 1 à 7 dans 7 registres différents. On lance la simulation et obtient le résultat *figures 4 et 5*.

[5]	00000001	00000000	00000001					
[6]	00000002	00000000		00000002				
[7]	00000003	00000000			00000003			

FIGURE 4 – Résultat registres 5 à 7 exo1

[28]	00000004	00000000			00000004			
[29]	00000005	00000000				00000005		
[30]	00000006	00000000					00000006	
[31]	00000007	00000000						00000007

FIGURE 5 – Résultat registres 28 à 31 exo1

On obtient bien le résultat escompté. Bien qu'on utilise des registres nommés de 0 à 6, les registres 5 à 7 et 28 à 31 sont utilisés. Ce sont en effet, d'après l'**ABI RISCV [4]**, les registres temporaires du standard RISCV. Chaque valeur est successivement associée à l'un des registres comme nous l'attendions d'après l'analyse de l'algorithme.

```

wait (inst_w == 32'h0000006F)
repeat (5) begin
  @posedge clk_r;
end
$display("Simulation stops at %t", $time);
$stop;
end

```

FIGURE 6 – Code fin de simulation

Dans le fichier `RV32i_tb.sv`, on observe le code illustré en *figure 6*. Ce dernier attend que l'instruction lue soit `32'h0000006F`, correspondant à une instruction JAL (jump and link). Dans le fichier `Exo1.S`, on retrouve cette instruction configurée pour effectuer un saut (jump) vers elle-même, ce qui revient à une boucle infinie (`while(1)` en C).

Une fois cette boucle atteinte, on attend encore 5 fronts montants de l'horloge. Ce délai permet de stabiliser l'exécution avant d'afficher le temps d'arrêt et de terminer la simulation. Ce mécanisme garantit une fin de simulation propre et évite l'apparition de problèmes.

A partir du pseudo code, on implémente l'algorithme 1 dans un fichier *main.S*. On modifie également le fichier de compilation pour bien compiler ce fichier main et non exo1.

En transposant ce code assembleur de manière brute en assembleur on obtient le résultat *figure 7*. En compilant ensuite le code on obtient un résultat qui n'est pas le résultat initialement souhaité, qui aurait été :

$$t0 = 0x3 ; t1 = 0x8 ; t2 = t1 + t0 = 0xB ; t3 = 0x10 ; t4 = 0x11 ;$$

$$t5 = t3 - t4 = -0x1$$

Cependant ce résultat était prévisible puisque nous avons réalisé tout ces calculs sans prise en compte et gestion des dépendances. Or juste après avoir initialisé les deux premiers registres, on fait une additions sur ces derniers, on doit donc faire face à une dépendance de donnée. En effet en observant le résultat *figures 8 et 9*, on observe que les registres *t0* et *t1* sont correctement initialisés à 3 et 8, mais que leur somme dans *t2* n'est jamais réalisée. De la même manière, les registres *t3* et *t4* sont correctement initialisés, mais par leur différence.

```

1 .section .start;
2 .globl start;
3
4 _start:
5     li t0, 0x3
6     li t1, 0x8
7     add t2, t1, t0
8     li t3, 0x10
9     li t4, 0x11
10    sub t5, t3, t4
11
12    lab1 : j lab1
13    nop
14
15 .end start

```

FIGURE 7 – Code assembleur avec dépendances

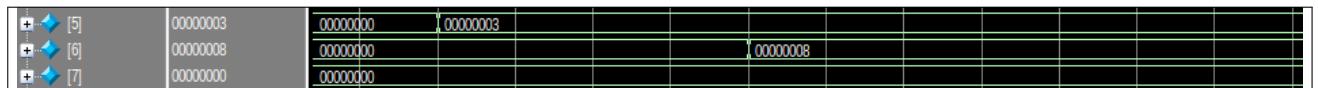


FIGURE 8 – Résultat registres 5 à 7 avec dépendances

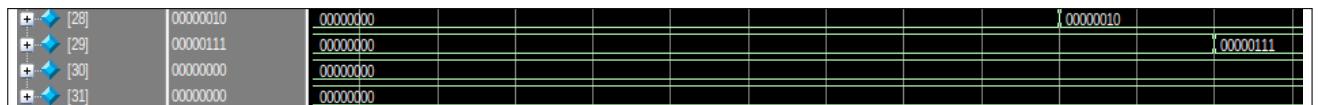


FIGURE 9 – Résultat registres 28 à 31 avec dépendances

En résolvant les problèmes de dépendances en insérant des *NOP* selon le code assembleur *figure 10*, on obtient finalement le résultat souhaité, *figure 11 et 12* avec les opérations d'addition et de soustraction correctement effectuées respectivement dans les registres 3 et 6.

```

1 .section .start;
2 .globl start;
3
4 _start:
5     li t0, 0x3
6     li t1, 0x8
7     nop
8     nop
9     nop
10    add t2, t1, t0
11    li t3, 0x10
12    li t4, 0x11
13    nop
14    nop
15    nop
16    sub t5, t3, t4
17
18    lab1 : j lab1
19    nop
20
21 .end start

```

FIGURE 10 – Algo exo1 sans dépendances



FIGURE 11 – Résultat registres 5 à 7 sans dépendances

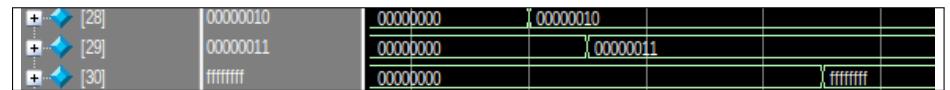


FIGURE 12 – Résultat registres 28 à 31 sans dépendances

### 3-Gestion des dépendances

L'objectif de cette partie est de constater l'influence des dépendances, de les comprendre, les gérer puis implémenter la gestion de ces dépendances directement par notre processeur.

Le support que nous utiliserons pour appréhender ces caractéristiques sera l'exécution d'un programme de multiplication par accumulation et décalage de bits. dans notre cas nous ferons la multiplication de 8 par 7 et l'objectif sera donc de pouvoir obtenir 56 en gérant les dépendances.

#### 3.1 Exécution d'une multiplication

##### 3.1.1 Implémentation du pseudo-code

En ne prenant en compte que le pseudo-code et en ne nous préoccupant pas des problématiques de dépendances pour l'instant, on obtient le code assembleur *figure 13*. Ce dernier boucle et fait le décalage de bits pour théoriquement atteindre la valeur 56. Cependant, le programme, comme pressenti, n'a pas le comportement attendu et la valeur diverge ce qui est illustré par le registre temporaire 28 en *figure 14*, qui correspond bien au registre *t3* qui doit stocker la valeur finale de notre opération d'après le code assembleur *figure 13*. Pour voir plus précisément d'où vient le problème, on observe le chronogramme pour mettre en valeur, dans un premier temps, une dépendance de donnée. Le chronogramme complet pour mettre en valeur cette dépendance est fourni en annexe A. On décrit ici la dépendance de donnée qui suit l'opération :

andi t4, t1, 1

```

1 .section .start
2 .globl start;
3
4 start:
5
6     li t1, 0x7
7     li t0, 0x8
8     li t2, 0xf
9     li t3, 0x8
10
11    loop_start:
12        andi t4, t1, 1
13        bnez t4, loop_follow # si 0 on saute l'addition
14        addi t3, t3, t0
15
16    loop_follow:
17        addi t2, t2, -1
18        srl t1, t1, 1
19        slli t0, t0, 1
20        bneq t2, loop_start # poursuit boucle si t2 != 0
21
22    lab :
23        j lab
24
25 .end start

```

FIGURE 13 – Code assembleur multiplication avec dépendances

En suivant le chronogramme, on remarque que la valeur du registre *t4* est récupérée avant d'être mise à jour. On détaille le phénomène dans l'exemple précédent en *figure 14*. On isole uniquement les deux instructions en conflit, les cases vides correspondant aux instructions précédentes et suivantes mais qui ne sont pas intéressantes pour illustrer le phénomène de dépendance.

Timecode	Opcode	Instruction	Rs1_add_i	Rs1_data_w	Rs2_add_i	Rs2_data_w	Rd_add_i	do_i (EX)	Memory	Write Back
370 ns	13	ANDI	0x01	0x01	/	/	0x1d			
380 ns	63	BEQ	0x1d	0x1	0x0		/	On effectue ANDI t4, t1, 1 On a lsb t1=0 donc do_i = 0		
390 ns								BEQZ sur t4 or t4=1 -> on n'effectue pas de saut alors qu'on devrait	Pas d'accès mémoire pour ANDI	
400 ns									Pas d'accès mémoire pour BEQZ	Mise à jour du registre t4 = 0 ( ad 0x1d )
410 ns										On renvoi l'adresse pc_counter + 4, pas de saut alors que on a bien t3 = 56

On récupère la valeur de t4 avant qu'elle ne soit mise à jour, on fait le ANDI sur l'ancienne valeur, dépendance de donnée

FIGURE 14 – Dépendance de donnée sur un ANDI

On étudie ensuite comment une dépendance de contrôle apparaît, et plus spécifiquement une dépendance liée à un branchement. Le chronogramme lié à cette dépendance est disponible en annexe B. Ce type de dépendance se produit lorsque sur une instruction de type B, on vérifie une condition qui amène à un saut, mais que l'on a déjà propagé les instructions devant être sautées dans les premiers étages du processeur. On détail un exemple en *figure 15*.

Timecode	Opcode	Instruction	Rs1_add_i	Rs1_data_w	Rs2_add_i	Rs2_data_w	Rd_add_i	EX	Memory	Write Back
370 ns	63	BEQ	0x1d	0x0	0x0	0	/			
380 ns	33	ADD	0x1c	0x38	0x05	0x40	0x1c	On effectue la comparaison, on a bien t4 = 0, on veut faire le saut		
390 ns								On exécute le add alors qu'on ne voulait pas	Pas d'accès mémoire pour BEQZ	
400 ns									Pas d'accès mémoire pour ADD	On rend le saut effectif (trop tard)
410 ns										On met à jour t4 qui passe à 78, on a une erreur

FIGURE 15 – Dépendance de contrôle sur un BEQZ

### 3.2 Correction

#### 3.2.1 Correction logicielle

Pour régler les problèmes de dépendances de manière logicielle, on utilise l'insertion d'instruction NOP (No Operation). Son fonctionnement repose sur l'exécution d'une instruction *addi x0, x0, 0*. Cette opération revient à additionner 0 au registre null. Elle n'a ainsi aucun effet mais permet d'occuper un cycle d'horloge. Cela permet à d'autres instructions de progresser dans les étages d'un processeur et de résoudre des problèmes de dépendances.

Le nombre de *nop* à insérer dépend du type de dépendance. Pour les dépendances de donnée, on insère trois *nop*, pour les dépendances de contrôle en saut, on insère un *nop* et pour les dépendances de contrôle en branchement, on insère deux *nop*. Pour le calcul de multiplication, on insère des *nop* suivant la *figure 17* et on obtient bien le résultat souhaité *figure 16*.

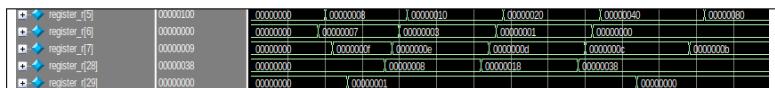


FIGURE 16 – Résultat 0x38 soit 56 en décimal

#### 3.2.2 Correction matérielle : Dépendance de données

On utilise différents signaux pour la gestion des registres. **rs1\_dec\_w** récupère *instruction\_i[19 :15]*, d'après la green\_card cette partie de l'instruction représente le registre source 1 comme cela est suggéré par le nom de la variable. De la même manière on récupère *instruction[24 :20]* pour **rs2\_dec\_w** pour le registre source 2.

Pour chaque étage, **rd\_add\_XXX\_w** est la variable qui permet de stocker l'adresse du registre de destination, laquelle correspond aux bits *[11 :7]* d'après la *green card* et le code.

```

1 .section .start
2 .globl start;
3
4 start:
5   li t1, @x7
6   li t0, @x8
7   li t2, @xf
8   li t3, @x0
9
10  loop_start:
11    andi t4, t1, 1
12    nop # dépendance de donnée sur t4
13    nop
14    nop
15    beqz t4, loop_follow # si 0 on saute l'addition
16    nop # dépendance de contrôle
17    nop
18    add t3, t3, t0
19
20  loop_follow:
21    addi t2, t2, -1
22    nop #dépendance de donnée sur t2
23    srli t1, t1, 1
24    slli t0, t0, 1
25    bnez t2, loop_start # poursuit boucle si t2 != 0
26    nop #dépendance de contrôle sur le bnez
27    nop
28
29  lab :
30    j lab
31
32 .end start

```

FIGURE 17 – Code assembleur multiplication avec nop

On s'inspire du cours pour résoudre les dépendances. L'équation booléenne du signal **stall\_w** vise à comparer l'adresse source 1 (**rs1\_add**) ou 2 (**rs2\_add**) de l'étape d'instruction *decode* avec l'adresse de destination (**rd\_add**) d'un signal d'un des étages *execute*, *instruction memory* ou *write back*, pour éviter une dépendance de données. On génère un signal **nop** si l'une des comparaisons est vraie.

L'équation booléenne est donnée par :

$$(rs1\_dec\_w == rd\_add\_exec\_w) \parallel (rs2\_dec\_w == rd\_add\_mem\_w) \parallel (rs1\_dec\_w == rd\_add\_wb\_w) \parallel \\ (rs2\_dec\_w == rd\_add\_exec\_w) \parallel (rs1\_dec\_w == rd\_add\_mem\_w) \parallel (rs2\_dec\_w == rd\_add\_wb\_w)$$

Lorsque le signal **stall\_w** est actif, il est envoyé dans un multiplexeur qui insère une instruction **nop** pour éviter un potentiel conflit dans le processeur. Ce multiplexeur est représenté dans le code par la ligne suivante :

```
assign inst_dec_r = (stall_w == 1'b1) ? 32'h00000013 : instruction_i;
```

En effet, la valeur 32'h00000013 correspond à une instruction **nop**, qui est assignée à *instruction\_i* pour résoudre le conflit.

```
function is_read_rs1(input logic [6:0] opcode);
  unique case(opcode)
    RV32I_R_INSTR: return 1'b1;
    RV32I_I_INSTR_OPER: return 1'b1;
    RV32I_I_INSTR_LOAD: return 1'b1;
    RV32I_S_INSTR: return 1'b1;
    RV32I_B_INSTR: return 1'b1;
    RV32I_B_INSTR_JALR: return 1'b1;
    default: return 1'b0;
  endcase
endfunction
```

Toutes les instructions n'utilisent pas le registre source **rs1**. Par conséquence, on définit un signal **rs1\_read** qui indique si le *registre 1* est lu par cette instruction. Pour ce faire on définit une fonction *is\_read\_rs1()* qui repose sur le package et qui permet de savoir lorsque le registre est utilisé. Dès lors on met la variable **rs1\_read** à 1'b1 ou 1'b0. On applique un raisonnement analogue pour **rs2**, avec une variable **rs2\_read** et une fonction *is\_read\_rs2()*. Enfin, on applique cette démarche au registre de destination **rd** mais avec, cette fois-ci, une variable par étage concerné, avec le nom **rd\_write\_etage\_w** et la fonction *is\_write\_rd()*. On illustre ces fonctions avec *is\_read\_1()* en figure 18.

On peut ainsi modifier l'équation. Pour plus de clarté, on sépare l'équation en deux parties, avec pour la première le hazard sur **rs1** et sur la seconde le hazard sur **rs2**. On effectue ensuite l'opération "OU" sur ces deux signaux pour obtenir **stall\_w**. On ajoute également une condition sur le registre **x0** qui n'est jamais modifié et qu'on a donc pas besoin de prendre en compte. On obtient finalement la logique booléenne suivante.

```
assign rs1_hazard_w =
  rs1_read_w && (rs1_dec_w != 5'd0) && (
    (rs1_dec_w == rd_add_exec_w && rd_write_exec_w) ||
    (rs1_dec_w == rd_add_mem_w && rd_write_mem_w) ||
    (rs1_dec_w == rd_add_wb_w && rd_write_wb_w)
  );

assign rs2_hazard_w =
  rs2_read_w && (rs2_dec_w != 5'd0) && (
    (rs2_dec_w == rd_add_exec_w && rd_write_exec_w) ||
    (rs2_dec_w == rd_add_mem_w && rd_write_mem_w) ||
    (rs2_dec_w == rd_add_wb_w && rd_write_wb_w)
  );

assign stall_w = rs1_hazard_w || rs2_hazard_w;

assign stall_o = stall_w;
```

Le reste de la logique concernant l'insertion de *NOP* suite à l'activation du signal **stall** étant déjà implémentée dans le fichier *controlpath.sv*, les dépendances de données devraient théoriquement être résolues . On réécrit donc le code assembleur sans les *NOP* résolvant les dépendances de données.

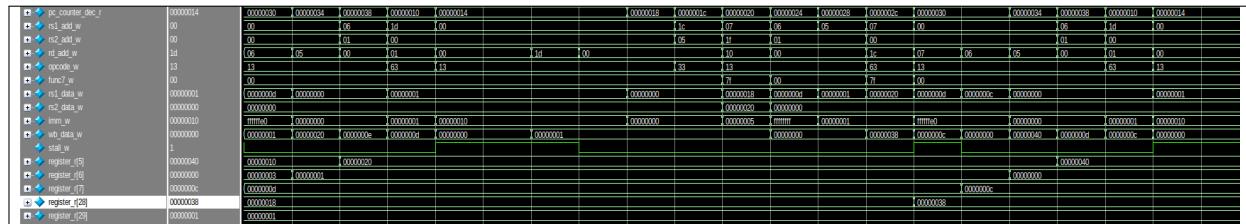


FIGURE 19 – Preuve résolution dépendance de donnée

On se place *figure 19* à l'endroit où le ANDI posait précédemment problème (3.1.1). Au moment où l'on exécutait l'instruction *BEQZ*, on chargeait  $t_4$  avec la valeur 1 alors qu'elle devait être mise à 0. Ici, l'activation du signal **stall\_w** permet au signal **wb\_data\_w** de propager la valeur  $t_4 = 0$  avant d'exécuter le *BEQZ*, et donc de résoudre le hazard de donnée.

### 3.2.3 Correction matérielle : Dépendance de contrôle (jump)

Pour la gestion des dépendances de contrôle, on ne modifie pas l'équation booléenne utilisée pour gérer les dépendances de données. En effet, la gestion par signal **stall** influe sur le comportement du *pc\_counter*, ce qu'on souhaite éviter pour les hazards de contrôle.

Aussi pour gérer les problèmes de types sauts, on s'inspire du bloc de gestion *branch* déjà implémenté dans le *controlpath*. Ainsi on crée un signal **jump\_taken\_w** dans un bloc *always\_comb jump\_taken\_comb*. Si on a une instruction de type *j*, on met **jump\_taken\_w** à 1. Cependant contrairement au bloc du *branch*, on lit l'opcode de l'instruction pour l'étage *decode* et non *execute*. On propage cette valeur au bloc *datapath* sous le nom *jump\_dep\_o*. L'objectif est "d'écraser" l'instruction qui sera sautée. Aussi il faudrait à l'étage de *decode* remplacer l'instruction chargée dans l'étage *fetch* par un *NOP*. S'il n'y a pas de bloc *always\_ff* pour le *fetch* contrairement aux autres étages, on peut assimiler l'assimation du **inst\_w** à cette étape. Ainsi, on met en place l'équation booléenne suivante.

```
assign inst_w = jump_dep_i ? 32'h00000013 : imem_data_i; // équivalent etage fetch
```

De cette manière, on érase directement l'instruction sautée à l'étage *fetch* sans influencer le *pc\_counter*, ce qui est illustré par le tableau *figure 21* issue du cours [2]. I2 est une instruction de type *J*, on érase l'instruction suivante I3. Comme nous n'utilisons pas d'instructions de type *J* pour notre multiplication en dehors de la boucle infi de fin, on crée un nouveau code assembleur dédié au test de la gestion de saut, illustré en *figure 20*.

```
1 .section .start
2 .globl start;
3
4 start:
5
6     li t1, 0x1
7     jal x0, verif
8     li t2, 0xFF
9 verif:
10    li t3, 0x2
11
12 lab :
13    j lab
14
15 .end start
```

FIGURE 20 – Code assembleur preuve résolution hazard de saut

	c1	c2	c3	c4	c5	c6	c7	c8
IF	I1	I2	I3	I5	I6			
ID		I1	I2	NOP	I5	I6		
EX			I1	I2	NOP	I5	I6	
MEM				I1	I2	NOP	I5	I6
WB					I1	I2	NOP	I5

FIGURE 21 – Tableau cours dépendances de saut

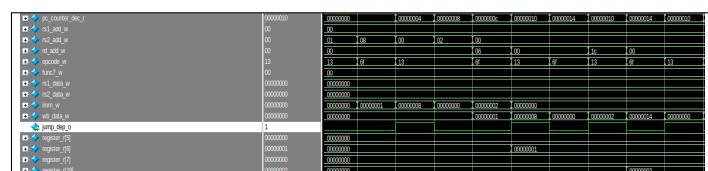


FIGURE 22 – Preuve résolution dépendance de saut

En exécutant ce code on obtient bien le résultat escompté en écrasant l'instruction problématique en *ligne 8*. De plus sur le chronogramme *figure 22*, on remarque bien qu'après une instruction de saut avec l'opcode *0x6f*, le signal **jump\_dep\_i** est activé, et que l'instruction suivante chargé dans l'étage **decode** est un *NOP*. De plus, le registre temporaire **t2** n'est jamais initialisé à la valeur *0xFF*.

### 3.2.4 Correction matérielle : Dépendance de contrôle (branch)

Comme évoqué en 3.2.3, un signal de détection des instructions de *type B* est déjà implémenté. Il cherche un opcode correspondant à ce type à l'étape **execute**. Ainsi, contrairement au *jump*, on doit injecter deux *NOP*, tout en ne bloquant pas le *PC\_counter*. On modifie l'équation précédente pour insérer un *NOP* à l'étape **fetch** en cas de *type B* également.

```
assign inser_nop = jump_dep_i || branch_taken_i;
assign inst_w = inser_nop ? 32'h00000013 : imem_data_i; // equivalent etage fetch
```

On doit cependant ajouter un second *NOP* comme illustré dans le cours [2] en *figure 23*. Puisque *I4* est déjà écrasée depuis la logique booléenne précédente, on souhaite ici effectué le remplacement de *I3* à l'étape **execute**.

	c1	c2	c3	c4	c5	c6	c7	c8
IF	I1	I2	I3	I4	I6			
ID		I1	I2	I3	NOP	I6		
EX			I1	I2	NOP	NOP	I6	
MEM				I1	I2	NOP	NOP	I6
WB					I1	I2	NOP	NOP

FIGURE 23 – tableau cours dépendance de branchement

Aussi, en utilisant à nouveau la variable **branch\_taken\_i** dans le *datapath*, on peut ajouter une nouvelle condition sur **branch\_taken\_i** dans **exec\_stage**. On continue de faire avancer le *PC\_counter*, mais en remplaçant les sortie de l'alu, la valeur de *rs2* ainsi que la *func3* par 0, ce qui revient à écraser par un *NOP*.

```
else if (branch_taken_i == 1'b1)
begin
    alu_op1_data_r <= 0; // on remplace par zero pour faire nop
    alu_op2_data_r <= 0; // même chose
    rs2_data_r <= 0; // même chose
    func3_exec_r <= 0; // même chose
    pc_br_target_r <= pc_br_target_w; // on fait tout de même progresser le pc
    pc_counter_exec_r <= pc_counter_dec_r; // même chose
end
```

Pour le test de cette gestion de dépendance et puisque les données sont déjà gérées, on peut reprendre le *Mult.S* initial et le tester en enlevant tout les *NOP*. Pour vérifier la pertinence de notre implémentation, nous observerons cette fois les signaux **stall\_w** et **branch\_taken\_i**. On observe le résultat de notre simulation *figure 24*.

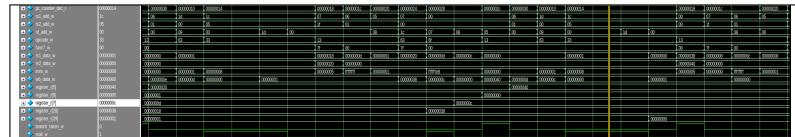


FIGURE 24 – Preuve résolution dépendance de branchement

Comme attendu, lorsqu'une instruction de *type B* atteint l'étage **execute**, les deux instructions suivantes sont remplacées par des instructions *NOP*. On remarque ainsi l'activation du signal **branch\_taken\_i** lorsqu'une instruction d'opcode *0x63* atteint l'étage **execute**.

Finalement, et en conclusion de cette partie, notre processeur est maintenant théoriquement capable d'exécuter tout type de code pseudo-assembleur sans que l'utilisateur n'ai à se préoccuper des dépendances de ses instructions. Cependant, cette gestion s'est faite au prix de la vitesse d'exécution, et la suite de l'amélioration d'un processeur en pipeline passe par une amélioration de l'efficacité temporelle.

## 4-Mémoire cache

### 4.1 Cache d'instruction en lecture

#### 4.1.1 Cache d'instruction 1 voie

Pour commencer, on caractérise les propriétés principales de notre futur cache. Ce dernier, plus rapide que la mémoire classique vient se placer comme intermédiaire entre cette dernière et le *SOC*. Pour mieux s'imaginer cette dernière, on schématisé l'architecture en *figure 25*. La nouvelle entrée **imem** permet de définir les moments où la lecture doit être effectuée dans la mémoire cache ou quand le processeur doit "attendre".

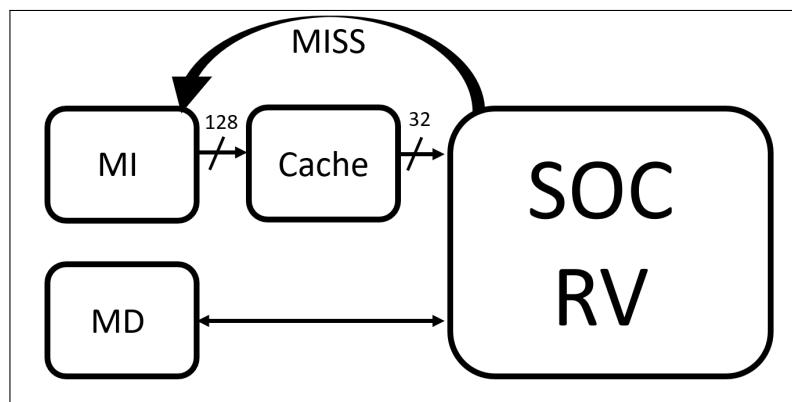


FIGURE 25 – Schéma communication SOC avec mémoire

Après avoir schématisé l'architecture, on cherche à déterminer les paramètres de notre mémoire cache. On détermine ainsi la taille d'une ligne, son nombre d'octets et de mots.

$2^K$  lignes  $\rightarrow 2^b$  blocs de  $2^4$  mots  
 Bytes = 16 (calcul des lignes)  
 4 mots par ligne = 128 bits par ligne

On calcule le nombre de Bytes d'offset **b**, le nombre de bits d'index **k** et le nombre de bits d'index **t**.

$$\begin{aligned} \mathbf{b} &= 3 - 0 + 1 = 4 \\ \mathbf{k} &= 9 - 4 + 1 = 6 \\ \mathbf{t} &= 31 - 10 + 1 = 22 \end{aligned}$$

On peut ensuite qualifier les performances d'une mémoire cache. En prenant un cache avec les performances suivantes : L1 Hit : 1 cycle ; Miss penalty : 10 cycles ; L1 Hit rate : 90% ; Miss rate : 10% ; fonctionnement à 100 MHz. en prenant 1000 accès mémoire, on calcule le taux de MISS et le temps d'accès moyen.

Sur 1000 accès mémoire :

- Taux de MISS : 10%
- Temps d'accès moyen :  $1 \text{ cycle} + (0,1 \times 10 \text{ cycles}) = 2 \text{ cycles}$

Les questions précédentes nous permettent de définir les *localparam* de notre module *cache1.sv*.

Les données stockées dans cette cache sont les instructions. Il est important de les stockées dans la mémoire cache car ces dernières sont souvent temporaires et doivent être exécutée le plus rapidement possible. Ainsi, les placer en mémoire cache sera bien plus pertinent puisque c'est la mémoire avec l'accès le plus rapide. Chaque instruction fait 32 bits et on peut donc en stocker 4 par ligne. Les informations comme le bit de validité ou le tag sont en réalité en dehors des 128 bits de données : ce sont les **métadonnées**.

On crée ainsi des lignes de 128 bits pour notre mémoire cache.

Étant donné que cette section se concentre sur l'implémentation du cache de lecture, et pour éviter de surcharger le rapport et le rendre indigeste, nous ne répondrons pas aux questions en copiant-collant directement le code. Nous nous limiterons à décrire le principe de fonctionnement. Cependant le code commenté est joint dans le rendu de projet.

Pour le découpage de l'adresse, elle se fait selon la forme suivante :

```
| 31 . Tag . 10 | 9 . Index . 4 | 3 . Offset . 0 |
```

Le fonctionnement du *hit / miss* permet d'indiquer si des données sont trouvable ou non dans la mémoire cache. Si c'est le cas on les lit directement. Dans le cas contraire, on lit la mémoire principale et on initialise de nouvelles lignes de cache avec la donnée. On implémente la logique derrière la lecture directe du cache ou de la mémoire classique en fonction du *hit / miss*. De plus on implémente un signal **read\_valid\_o**. En transmettant ce signal du cache vers le datapath et le controlpath grâce au module top et soc, on implémente une logique d'attente lorsque la valeur est *miss*. Dans le datapath et le controlpath, on peut ensuite rajouter, dans tous les blocs *always\_ff*, une condition pour chaque propagation d'instruction, qu'on illustre avec l'exemple suivant.

```
always_ff @(posedge clk_i or negedge resetn_i) begin : mem_stage
    if (resetn_i == 1'b0) inst_me_r <= 32'h0;
    else if (imem_valid_i == 1'b1) inst_mem_r <= inst_exec_r;
end
```

Enfin on peut implémenter l'ajout d'une donnée dans le cache d'instruction. Cette dernière contient également la mise à jour du tag, de la validité, mais aussi du signal de lecture pour débloquer l'avancée du CPU. Dernière chose à régler pour obtenir un cache fonctionnel, il est nécessaire de convertir le type de *imem\_cache\_data*. En effet on assigne *imem\_cache\_data\_w[3:0][21:0]* à *mem\_read\_data[127:0]*. On transforme donc *imem\_cache\_data* en *imem\_cache\_data\_128* qui permet d'avoir un même type.

On peut enfin tester notre mémoire cache. Pour vérifier son bon fonctionnement on observe notamment le signal **read\_valid\_o**. On obtient le chronogramme en *figure 26*.

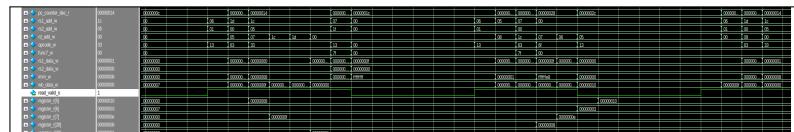


FIGURE 26 – Chronogramme mémoire cache

Tout d'abord on observe bien la bonne valeur finale avec le résultat *0x38*. Ensuite on remarque bien l'influence du signal **read\_valid\_o**, qui autorise l'avancée du processeur par "paquets".

#### 4.1.2 Cache d'instruction associatif 2 voies

On implémente ensuite un cache deux voies associatif à partir du cache simple de la partie précédente. Tout d'abord pour créer une deuxième voie on ajoute simplement une dimension au tableau précédent. Pour le hit on garde une logique similaire en divisant le signal *is\_hit*. On implémente de plus un signal *evict\_way* qui permet de respecter la logique lru et de supprimer lorsque c'est nécessaire, la donnée la plus ancienne.

Pour le signal de validité de lecture, on garde une nouvelle fois la même logique, tout comme pour l'écriture mémoire. Pour sélectionner la voie du cache, on se sert du signal **evict\_way**.

Concernant le test du cache 2 voies, il faut tout d'abord modifier le fichier *build.sh* du dossier sim et remplacer *cache1.sv* par *cache2voies.sv*. Si le fichier se compile correctement et tombe bien sur la valeur *0x38* souhaitée, on ne démontre par qu'il s'agit effectivement d'un cache 2 voies.

Pour un test qui permettrait de prouver le fonctionnement du cache 2 voies, on crée un fichier *testVoie.S*. Pour le compiler plutôt que *mult.S*, il faut remplacer les *mult.S* et *mult.o* par des *testVoie.S* et *testVoie.o* dans le *build.sh* du dossier firmware.

On imagine différentes approches pour montrer la différence entre les deux caches. Écrire deux valeurs différentes sur une même ligne : dans le cache 2 voies elles seront chacune sur une voie différente, on charge ces données avec un *lw* ultérieurement et on voit qu'elles sont toutes les deux à leur valeurs d'origine. On exécute le même code sur un cache une voie. La deuxième donnée écrasera donc la première et on lorsque que l'on chargera les valeurs, elles auront toutes les deux la valeur de la deuxième chargées dans le cache.

On imagine également une stratégie de remplissage d'une voie. Les données écrites par la suite écraseront des données dans le cache 1 voie quand elles continueront à être stockées dans le cache 2 voies.

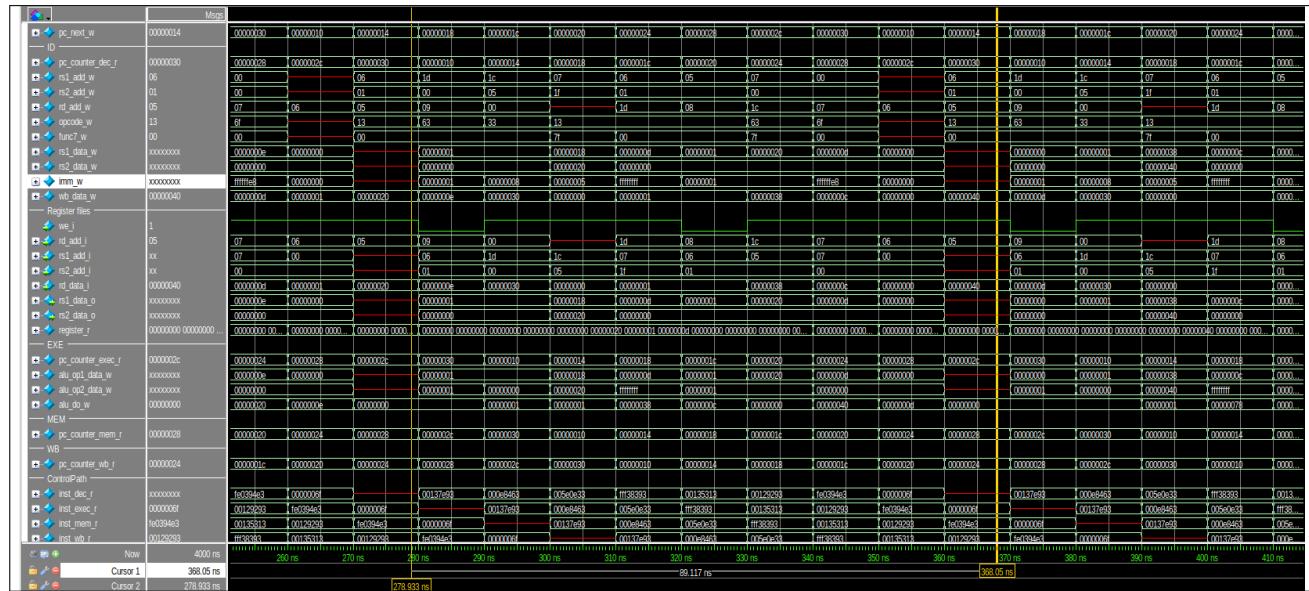
Cependant dans les faits, nous n'avons réussi à implémenter aucune de ces deux stratégies, et ne pouvons donc pas conclure quand à la nature de notre second cache.

## 5-Conclusion

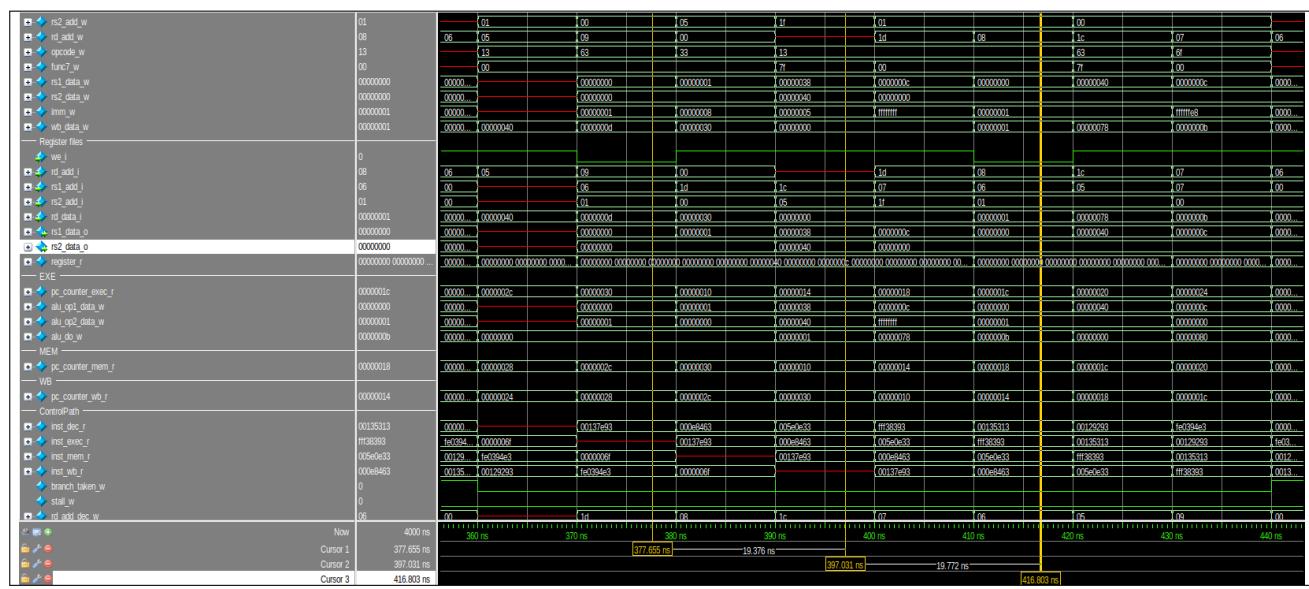
Ce projet nous a permis d'avancer dans notre compréhension des processeurs et du jeu d'instruction RISC-V. Si l'on sait désormais comment fonctionne un pipeline et un mémoire cache simple, le projet nous montre également les procédés qu'ils nous reste à apprêhender. Optimisation de la vitesse d'exécution, complexification de la mémoire pour optimiser le fonctionnement, introduction de prédition de branchement... Il nous invite de plus à réfléchir aux alternatives du processeur scalaire avec d'autres types de processeur plus complexe auquel nous pourrons nous intéressé dans le futur : architecture superscalaire pour l'exécution de plusieurs instruction en parallèle ou processeur *out-of-order*.

## 6-Annexes

### Annexe A Chronogramme dépendance de données



### Annexe B Chronogramme dépendance de contrôle



## Références

- [1] Prebuilt RISC-V GCC Toolchains for Linux. GitHub repository.  
<https://github.com/stnolting/riscv-gcc-prebuilt>
- [2] M. Agoyan, M. Lacruche, S. Pontié, et O. Potin, *Architecture des processeurs RISC-V (RV32I) - Architecture Pipeline*, Mines Saint-Étienne, 8 novembre 2024.  
<https://ecampus.emse.fr/mod/resource/view.php?id=35772>
- [3] D. A. Patterson et J. L. Hennessy, *Computer Organization and Design : The Hardware/Software Interface, RISC-V Edition*, Elsevier, 2018.  
<https://ecampus.emse.fr/mod/resource/view.php?id=2500>
- [4] RISC-V International, *RISC-V ABIs Specification, Version 1.0 : Ratified*, November 2022.  
<https://d3s.mff.cuni.cz/files/teaching/nswi200/202324/doc/riscv-abi.pdf>