



ECOLE DES MINES DE SAINT ETIENNE

Architecture des Processeurs 2

Rapport de travaux pratiques

Auteurs :
Kévin GUILLOUX

Table des matières

1	RV32I architecture pipeline	1
1.1	Étude du processeur RISC-V pipeliné	1
1.1.1	Le sous circuit data_path	2
1.1.2	Le sous circuit control_path	2
1.1.3	L'instance dmem du composant wsync_mem	3
1.1.4	L'instance imem du composant wsync_mem	3
1.2	Exécution et simulation d'un programme	3
2	Gestion des dépendances	7
2.1	Exécution d'un programme	7
2.2	Correction du problème	9
2.2.1	Correction logicielle	9
2.2.2	Correction matérielle : Interlock	11
2.2.2.1	Dépendances de données	11
2.2.2.2	Dépendances de contrôle	12
2.2.3	Correction matérielle : Bypass	14
3	Implémentation d'Please insert into preamble une mémoire cache	19
3.1	Questions préliminaires	19
3.1.1	Cache mémoire "Direct"	19
3.1.2	Performance	19
3.2	Cache d'instructions direct	20
3.2.1	Création du module "direct_cache"	20
3.2.1.1	Création du module	20
3.2.1.2	Test du module	22
3.2.2	Modification de la mémoire principale	23
3.2.3	Adaptation du processeur à la mémoire cache	25
3.2.4	Les modifications apportées	25
3.2.4.1	Modifications des control et data path	25
3.2.4.2	Modification du RISCV_core	26
3.2.4.3	Modification du top layer	26
3.2.4.4	Modification de linstanciation dans la testbench	26
3.2.5	Test et analyses des performances	27
3.3	Cache d'instructions associatif à plusieurs voies	29
3.3.1	Création du module multi_way_cache	29
3.3.2	Vérification et analyse des performances	31
3.4	Cache de données avec écriture "write through"	36
3.4.1	Création du module "write_through_cache"	36
3.4.2	Adaptation du processeur RISCV	39
3.4.3	Vérification et analyse des performances	40
3.4.4	Amélioration des performances par le bypass	42
3.4.4.1	Bypass de RS2 pour les instructions de type S	42
3.4.4.2	Bypass de RS2 pour les instructions de type LOAD	43
3.4.4.3	Mise en évidence des améliorations	44
3.5	Cache de données avec écriture "write back"	45
3.5.1	Création du module "write_back_cache"	45
3.5.2	Vérification et analyse des performances	48
3.6	Amélioration complète du bypass	49
3.6.1	Modifications apportées	49

3.6.1.1	Modifications du control_path	50
3.6.1.2	Modifications du data_path	53
3.6.2	Mise en évidence de l'amélioration des performances	54
4	Annexe	55
4.1	Connexions entre les différents blocs	55
4.2	Problèmes de compilations et solutions	56
4.3	Explications de l'archive .zip	56
4.4	Limitations de l'architecture fournie	56

Table des figures

1.1	Hierarchie des composants du RISC-V pipeliné	1
1.2	Chronogramme résultant de l'execution du programme "exo1.S"	4
1.3	Chronogramme résultant de l'execution du programme "main.S"	5
1.4	Chronogramme résultant de l'execution du programme "main.S" sur une architecture monocycle	6
2.1	Chronogramme résultant de l'execution du programme "mult.S"	8
2.2	Mise en évidence du problème dans le chronogramme de "mult.S"	8
2.3	Chronogramme résultant de l'execution du programme "mult_correc.S"	10
2.4	Logique combinatoire pour 2 types d'instructions	11
2.5	Chronogramme résultant de l'execution du programme "mult_correc_data_only.S"	12
2.6	Zoom sur le chronogramme résultant de l'execution du programme "mult_correc_data_only.S"	12
2.7	Zoom sur la partie JAL du chronogramme résultant de l'execution du programme "mult_correc_data_only.S"	13
2.8	Chronogramme résultant de l'execution du programme "mult.S" corrigé via hardware	13
2.9	Zoom sur le chronogramme résultant de l'execution du programme "mult.S" corrigé via hardware	13
2.10	Zoom sur la fin du chronogramme résultant de l'execution du programme "mult.S" corrigé via hardware	14
2.11	Chronogramme résultant de l'execution du programme "bypass.S"	15
2.12	Zoom sur 2 boucles du programme "bypass.S"	15
2.13	Chronogramme résultant de l'execution du programme "bypass.S", avec l'implémentation hardware	17
2.14	Zoom sur 2 boucles du programme "bypass.S", avec l'implémentation hardware	18
3.1	Chronogramme de la testbench du "direct_cache" seul	22
3.2	Chronogramme de la testbench du "direct_cache" avec la mémoire principale	25
3.3	Chronogramme de l'execution du programme "mult.S" sans cache et avec une mémoire "parfaite"	27
3.4	Chronogramme de l'execution du programme "mult.S" avec 4 mots par ligne de cache	27
3.5	Zoom sur le chronogramme de l'execution du programme "mult.S" avec 4 mots par ligne de cache	28
3.6	Chronogramme de l'execution du programme "mult.S" avec 8 mots par ligne de cache	28
3.7	Zoom sur le chronogramme de l'execution du programme "mult.S" avec 8 mots par ligne de cache	29
3.8	Chronogramme de la testbench du "multi_way_cache" avec la mémoire principale	31
3.9	Chronogramme de la testbench spécifique au "multi_way_cache" avec la mémoire principale	32
3.10	Chronogramme de l'execution du programme "mult.S" avec la cache multi-voies	33
3.11	Chronogramme de l'execution du programme "multiway_stresstest.S" avec la cache directe	34
3.12	Zoom sur une des boucles du programme "multiway_stresstest.S" avec la cache directe	34
3.13	Chronogramme de l'execution du programme "multiway_stresstest.S" avec la cache multivoies	35
3.14	Zoom sur une des boucles du programme "multiway_stresstest.S" avec la cache multivoies	35
3.15	Chronogramme de la testbench "write_through_cache_tb"	40
3.16	Chronogramme du programme "memcpy.S" avec la cache de données "write_through_cache"	41
3.17	Zoom sur le chronogramme du programme "memcpy.S" avec la cache de données "write_through_cache"	42
3.18	Chronogramme du programme "memcpy.S" avec la cache de données "write_through_cache" et le bypass corrigé sur RS2	44
3.19	Chronogramme du programme "memcpy.S" avec la cache de données "write_back_cache" et le bypass corrigé sur RS2	48
3.20	Zoom sur une boucle du programme "memcpy.S" avec la cache de données "write_back_cache" et le bypass corrigé sur RS2	49
3.21	Chronogramme du programme "memcpy.S" avec la cache de données "write_back_cache" et le bypass complet sur RS2	54
4.1	Diagramme rapide de la "die" du soc	55

Table des codes

2.1	Modifications apportées pour le stall et bypass	15
2.2	Modifications apportées aux contrôles de l'ALU	16
2.3	Modifications apportées aux choix des entrées de l'ALU	17
3.1	Affectation des valeurs du tag, de l'index et de l'offset selon l'adresse	20
3.2	Nom des registres représentant la mémoire de la cache	20
3.3	Conditions d'écritures (synchrone) et de lecture (asynchrone) dans les registres	21
3.4	Génération du signal de hit	21
3.5	Assignation de l'adresse mémoire	21
3.6	Assignation des différents signaux de contrôle	21
3.7	Reorganisation de la ligne en mots	22
3.8	Génération des signaux de sortie pour le processeur	22
3.9	Paramètres du module de mémoire	23
3.10	Initialisation des valeurs de la mémoire	23
3.11	Mémorisation et ajout des cycles de latence en écriture/lecture	24
3.12	Entrées sorties du cœur avec la cache	26
3.13	Registres mis à jours avec la LRU	29
3.14	Conditions d'écritures (synchrone) et de lecture (asynchrone) dans les registres, avec la LRU	30
3.15	Génération du nouveau signal de hit	31
3.16	Déclaration du nouveau module	36
3.17	Assignation de la ligne à écrire en mémoire	37
3.18	Modification des conditions d'écriture (synchrone) et de lecture (asynchrone) pour la cache write_through	37
3.19	Bascule permettant de résoudre les problèmes d'écritures involontaires	38
3.20	Code de logique de sortie de la cache write_through	38
3.21	Modifications apportées dans le data_path pour le bypass type S	43
3.22	Modifications apportées dans le control_path pour le bypass type S	43
3.23	Modifications apportées dans le control_path pour le bypass type LOAD	43
3.24	Bascule gérant la latence en entrée pour l'écriture	45
3.25	Bascule gérant la latence en sortie pour l'écriture	45
3.26	Partie mémoire de la cache write_back	46
3.27	Gestion des signaux de contrôle de la cache write_back	47
3.28	Gestion des dépendances, avec le bypass complet	50
3.29	Contrôle des entrées de l'ALU, avec le bypass complet	51
3.30	Contrôle de l'opérande des instructions de type S, avec le bypass complet	52
3.31	Choix de la valeur de RD à propager, avec le bypass complet	52
3.32	Affectation de la valeur de RD suivant l'étage, avec le bypass complet	53
3.33	Affectation des valeurs OP1 et OP2 suivant l'étage, avec le bypass complet	53
3.34	Affectation de la valeur à écrire en mémoire suivant l'étage, avec le bypass complet	53

Chapitre 1

RV32I architecture pipeline

1.1 Étude du processeur RISC-V pipeliné

Le circuit **top-level** est le circuit qui "englobe" tous les autres. Il est en effet le circuit sur lequel on s'appuie pour réaliser la testbench de notre IP. Ici, le circuit **top-level** correspond au circuit "**RV32i_soc**". On décompte les sous-circuits suivants : (On utilise le nom des instances, et non des modules)

- "**RV32i_core**" : Sous-circuit rassemblant la logique et la "puissance de calcul" du processeur, respectivement sous la forme du :
- "**RV32i_controlpath**" : Ce circuit permet de décoder les instructions présentes dans la mémoire "imem" et de contrôler le circuit "**RV32i_datapath**" en fonction de ces dernières. En résumé, il s'agit des "neuronnes" du processeur.
- "**RV32i_datapath**" : Ce circuit permet de réaliser l'ensemble des opérations mathématiques, combinatoires ou logiques via l'ALU. Mais également de stocker les données dans le banc de registres ou dans la mémoire de données "**dmem**". En résumé, il s'agit des "muscles" du processeur.
- "**imem**" : Sous-circuit qui correspond à la mémoire d'instructions du processeur.
- "**dmem**" : Sous-circuit qui correspond à la mémoire dédiée aux données du processeur.

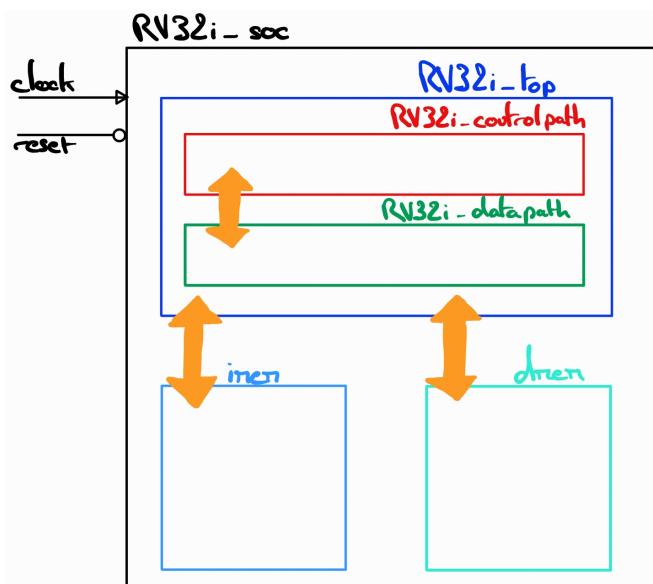


FIGURE 1.1 – Hiérarchie des composants du RISC-V pipeliné

1.1.1 Le sous circuit data_path

Q4.

Notre cœur RISCV est composé de 5 "étages" distinct, qui représentent les 5 différentes étapes de traitement d'une instruction. Ces "étages" sont différenciés par des "murs de registres", c'est à dire que l'on place un registre sur les connections entre les différents "étages". Il en résulte qu'ils sont séparés temporellement, une donnée qui passe d'un "étage" à un autre devra attendre une période d'horloge pour le faire.

Les différents étages sont :

- **IF (Instruction Fetch)** : On récupère l'instruction à traiter dans la mémoire d'instructions, suivant la valeur de l'adresse **PC** actuelle.
- **ID (Instruction Decode)** : On identifie les différents registres ou valeurs immédiates à utiliser avec l'instruction.
- **EXE (Execute)** : On effectue les opérations via l'ALU (mathématiques, combinatoires ou logiques).
- **MEM (Memory access)** : Si l'instruction le spécifie, on écrit ou on récupère une donnée dans la mémoire "dmem". (Cette étape n'est pas nécessairement présente dans les instructions)
- **WB (Write-Back)** : On réécrit (ou non) la valeur dans le banc de registres.

Q5.

Les signaux qui quittent le data_path vers le controle_path sont :

- **instruction_o** : Correspond à l'instruction présente dans l'étage **DEC**, donc celle qui était présente à l'étage **FETCH** et qui est passée par le mur de registre.
- **alu_zero_o** : Vaut 1 si la valeur de sortie de l'ALU vaut 0.
- **alu_lt_o** : Vaut 1 lorsque la valeur de sortie de l'ALU est négative.

1.1.2 Le sous circuit control_path

Q8.

Ce sont des signaux qui permettent la propagation de l'instruction entre les différents "étages" du **control_path**, donc entre les différents registres permettant de les séparer. On peut comparer ces signaux à des autoroutes à sens uniques, les registres servant alors de péage, et nous permettant ainsi de changer de portions d'autoroute (On passe de l'A7 à l'A9 par exemple, ce qui correspond alors à passer du signal **inst_exec_r** à **inst_mem_r**).

Q9.

L'adresse du registre de destination fait partie de l'instruction, elle est donc présente dans le "chemin principal" du **control_path** dès le début. Cette adresse va donc "parcourir" tout le **control_path** et, donc, passera par tous les registres. Elle arrivera donc directement au banc de registres en même temps que les commandes de l'instruction pour l'étage **WB (Write-Back)**.

Q10.

Le signal **stall_w** est actuellement simplement généré comme une "masse" constante. En effet, le signal se voit affecté la valeur **1'b0** via la ligne :

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

Il restera donc constamment à l'état bas et ne pourra pas être modifié (la gestion des dépendances n'est donc pas implémentée, du moins pas complètement).

Q11.

Si le signal **stall_w** est actif, l'instruction traitée dans le **control_path** n'est alors plus l'instruction qui provient de l'étape **IF (Instruction Fetch)**, mais une instruction par défaut, sans valeurs :

```
32'h0000 0013
```

Cette instruction correspond à une opération **ADDI (Addition Immediate)** entre la valeur 0 et le registre 0, enregistré dans le registre 0. Cette opération n'entraîne donc bien aucune dépendance, ni aucun impact sur les différentes valeurs enregistrées en mémoire ou dans le banc de registres. En effet, le registre 0 contient la valeur 0 et ne peut pas être modifié, cette opération lui ajoute donc 0 et ne le modifie pas.

1.1.3 L'instance dmem du composant wsync_mem

Cette mémoire commence par l'adresse $32'h0001\ 0000$, et elle possède 4096 "espaces" mémoires de 32bits. On a donc une mémoire de 131 072 bits, soit 16 384 octets, donc 16KiB (KibiByte).

1.1.4 L'instance imem du composant wsync_mem

Cette mémoire commence par l'adresse $32'h0000\ 0000$, et elle possède 4096 "espaces" mémoires de 32bits. On a donc une mémoire de 131 072 bits, soit 16 384 octets, donc 16KiB (KibiByte).

1.2 Éxécution et simulation d'un programme

Q16/17.

Le programme c'est executé parfaitement ici. En effet, comme on peut le voir dans l'ABI du RISC-V, les registres temporaires t0 à t6 correspondent bien aux registres 5 à 7 et 28 à 31 du banc de registres. De plus, les instructions de types **LI (Load Immediate)** sont ici effectuées via des instructions de type **ADDI (Addition Immediate)**, en remplaçant le registre source n°1 par le registre 0. La valeur immédiate tient sur 12Bits, il est donc cohérent d'utiliser cette instruction pour charger les valeurs dans les registres.

On remarque un décalage temporel d'une période entre le chargement de la première et de la seconde valeur dans le banc de registres, ceci intervient car le signal **reset** passe à l'état

actif 0.1ns après le front montant du signal d'horloge. L'instruction précédente possède donc une période d'avance sur la seconde, mais ce "décalage" se "répare" tout seul à l'instruction suivante.

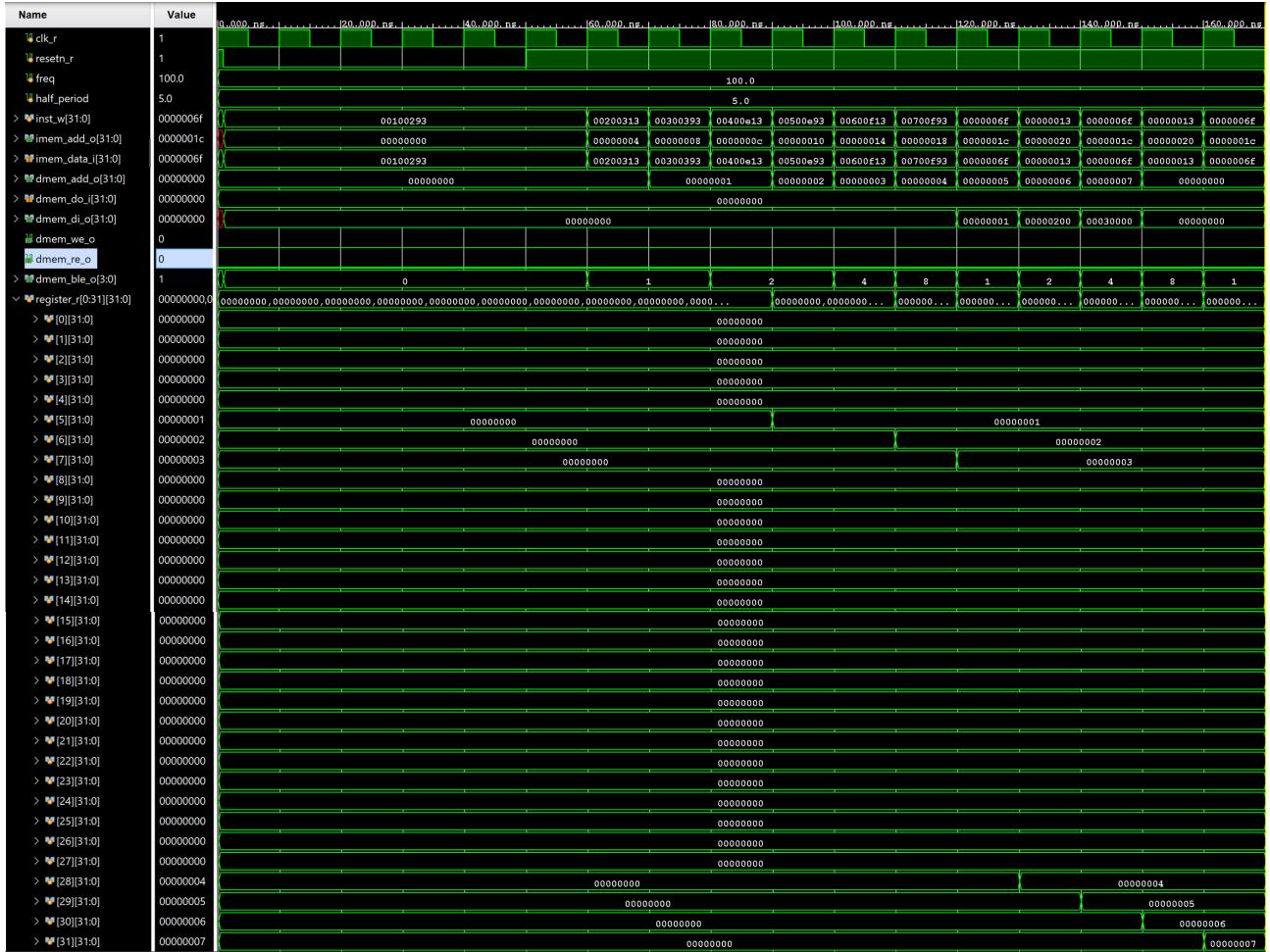


FIGURE 1.2 – Chronogramme résultant de l'exécution du programme "exo1.S"

La simulation est arrêtée après 5 périodes d'horloges (correspond au nombre d'étages de l'architecture pipeline, pour permettre à la dernière instruction de se terminer), suivant la détection de l'instruction :

$32'h0000\ 006F$

Cette instruction correspond à l'instruction de saut de type **JAL (Jump and Link)** vers la fonction lab1.

Q18.

Le fichier "main.S" regroupe les lignes suivantes :

```
.section .start;
.globl start;
```

```
start :
    li t0,0x3
```

```

    li t1,0x8
    add t2,t1,t0
    li t3,0x10
    li t4,0x11
    sub t5,t3,t4
lab1 : j lab1
    nop

```

.end start

On remplace les égalités par des **li**, afin que le compilateur décide de lui-même des instructions à utiliser pour charger les valeurs dans les registres de la manière la plus optimisée.

Q19.

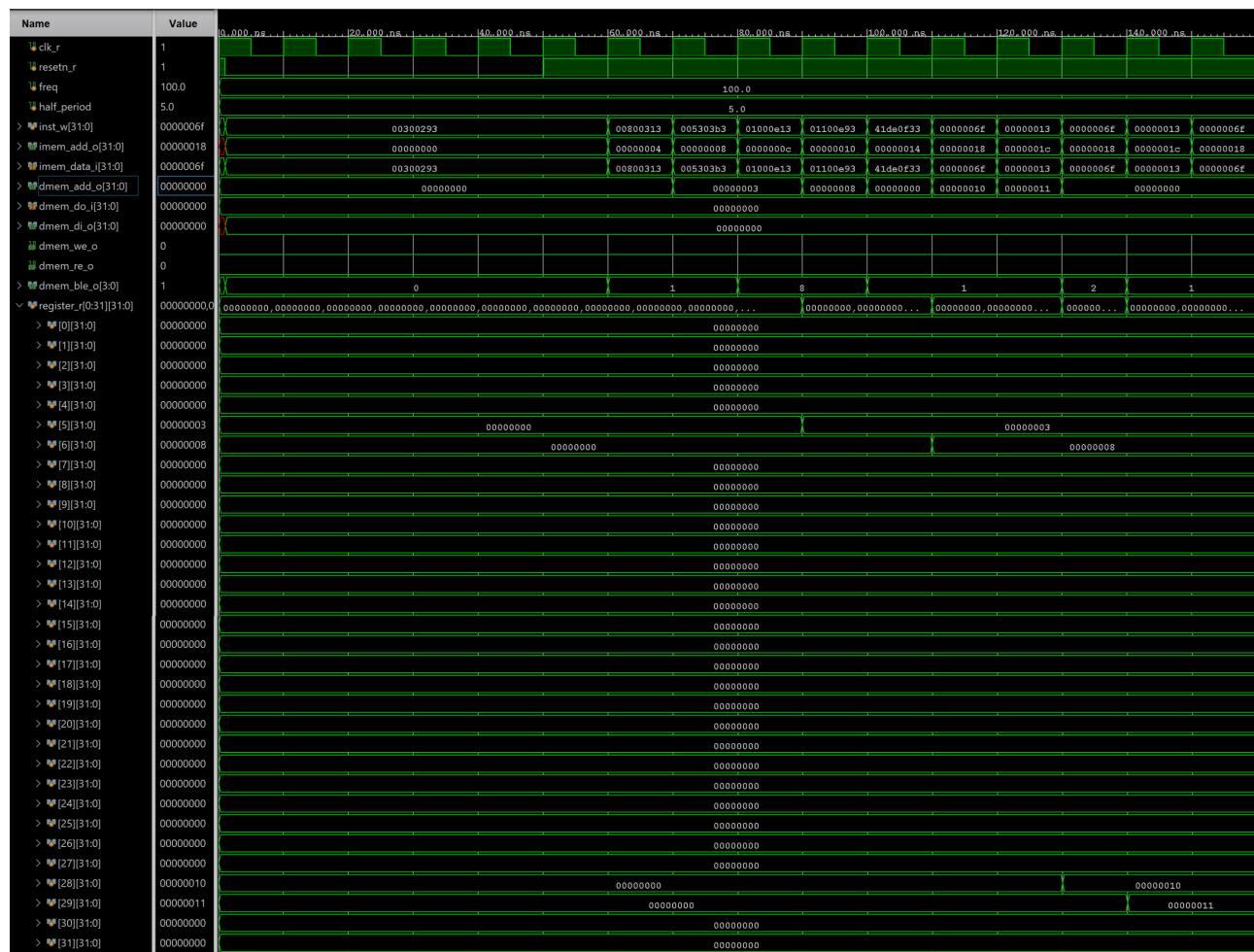


FIGURE 1.3 – Chronogramme résultant de l'exécution du programme "main.S"

On observe ici que les valeurs sont correctement chargées dans les registres **t0**, **t1**, **t3**, **t4**. Cependant, les registres **t2**, **t5** gardent la valeur 0. Le problème vient en fait des dépendances de données, en effet, lors de la sélection des valeurs dans le banc de registres par les instructions **add** et **sub**, **t0**, **t1** et **t3**, **t4** valent encore 0, en effet les valeurs ne sont pas encore chargées et sont encore respectivement à l'étape **EXE** et **MEM**. Le calcul s'effectue donc bien, mais avec

les mauvaises valeurs.

En comparant avec l'exécution du programme sur le même processeur RISC-V, mais cette fois-ci utilisant une architecture monocycle, on met bien en évidence le problème de dépendance qui survient dans l'architecture pipeline. Les valeurs sont ici chargées à temps dans le banc de registres, sans avoir besoin d'effectuer d'**interlocks** pour attendre l'arrivée des données manquantes.

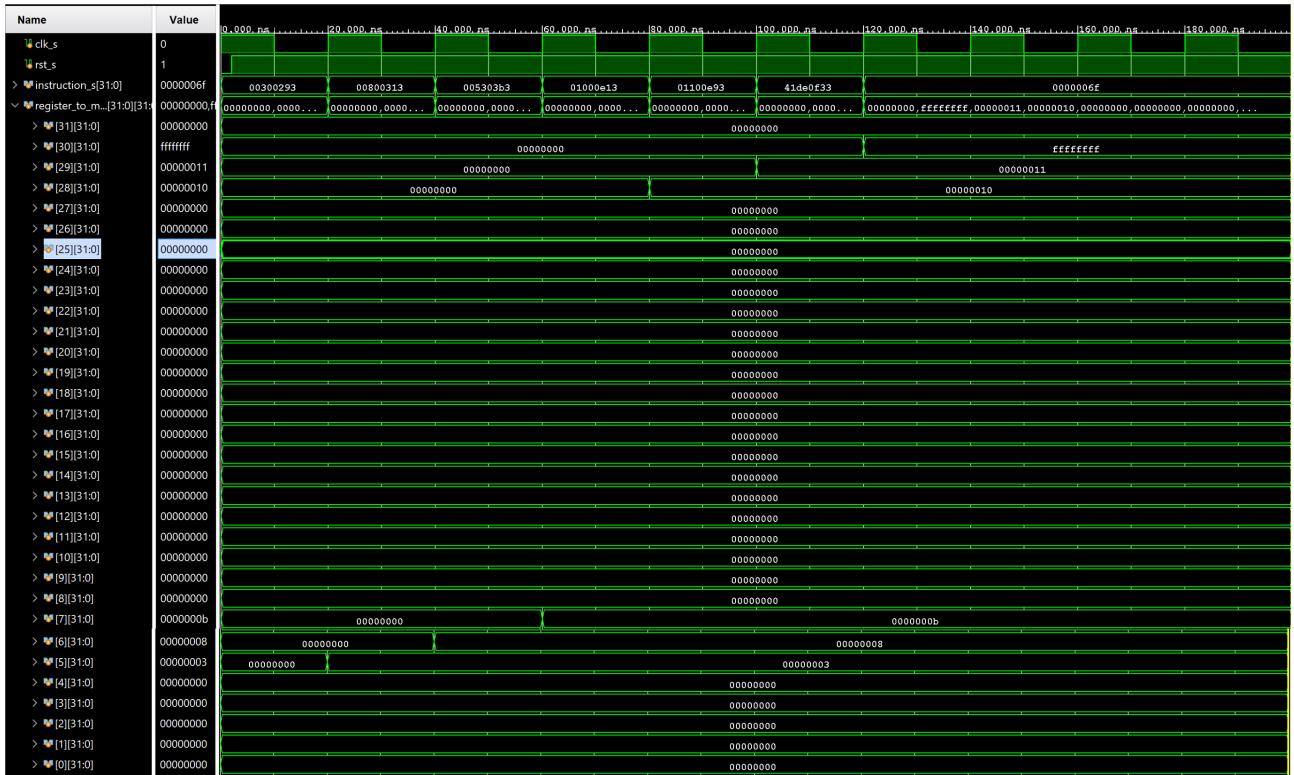


FIGURE 1.4 – Chronogramme résultant de l'exécution du programme "main.S" sur une architecture monocycle

Chapitre 2

Gestion des dépendances

2.1 Éxécution d'un programme

On réalise l'algorithme de la multiplication logicielle en assembleur. Dans un premier temps, on ne prends pas en compte les spécificités liés à l'architecture pipeline, notamment vis-à-vis des dépendances de données.

Programme en assembleur de **mult.S**

```
.section .start;
.globl start;

start :
    li t0, 0x8          //Operande 1 de la multiplication
    li t1, 0x7          //Operande de la multiplication
    li t2, 0x0          //Compteur de boucle
    li t3, 0x0          //Resultat de la multiplication
    li t4, 16           //Valeur de fin de boucle
    li t6, 0x1          //Valeur de comparaison du if

loop :
    addi t2,t2,1        //Incrementation de la boucle
    andi t5, t1, 0x001  //t5 contient uniquement l'information du bit de poids faible de t0
    bne t5,t6,endIf    //If(t1[0] == 1'b1)
    add t3,t3,t0

endIf :
    slli t0,t0,1        //Decalage a gauche de t0
    srli t1,t1,1        //Decalage a droite de t1
    bltu t2,t4,loop     //Verification de fin de boucle (t2 < t4?)

lab1 :
    j lab1
    nop

.end start
```

On s'attendrait donc à obtenir la valeur :

$$7 * 8 = 56 = 0x38$$

On compile donc ce programme et on le charge en mémoire de notre processeur RISCV pour observer le résultat final :

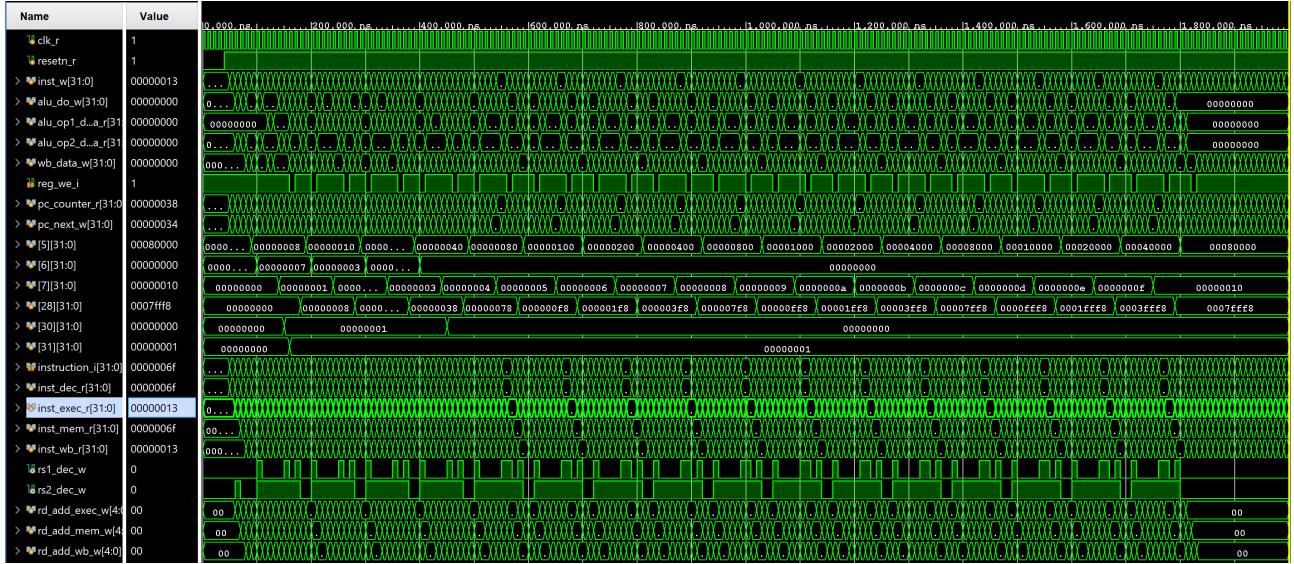


FIGURE 2.1 – Chronogramme résultant de l'exécution du programme "mult.S"

On remarque immédiatement que le programme ne s'exécute pas comme on l'attendait. En effet, on s'attend à avoir la valeur 56 en fin d'exécution (lorsque le registre n°7 atteint la valeur 0x10), mais on atteint la valeur 0x7fff8.

L'explication se trouve (encore une fois) dans l'absence de prise en compte hardware de la dépendance. On a en effet ici 3 instructions qui posent un problème de dépendance de données (en vert) ou de contrôle (en bleu).

- Le registre **t5** n'est pas encore mis à jour dans le banc de registres lorsque le branch du **if** (bne) arrive à l'étage de **decode**.
- L'instruction qui est conditionnée par la boucle s'effectue toujours, quelque soit le résultat de la condition. Le **if** est donc ici inutile et n'a pas d'impact sur le déroulement du programme.
- Les instructions qui suivent la vérification de la fin de la boucle s'effectuent dans tout les cas, car le **branch** n'a lieu que 2 cycles plus tard.

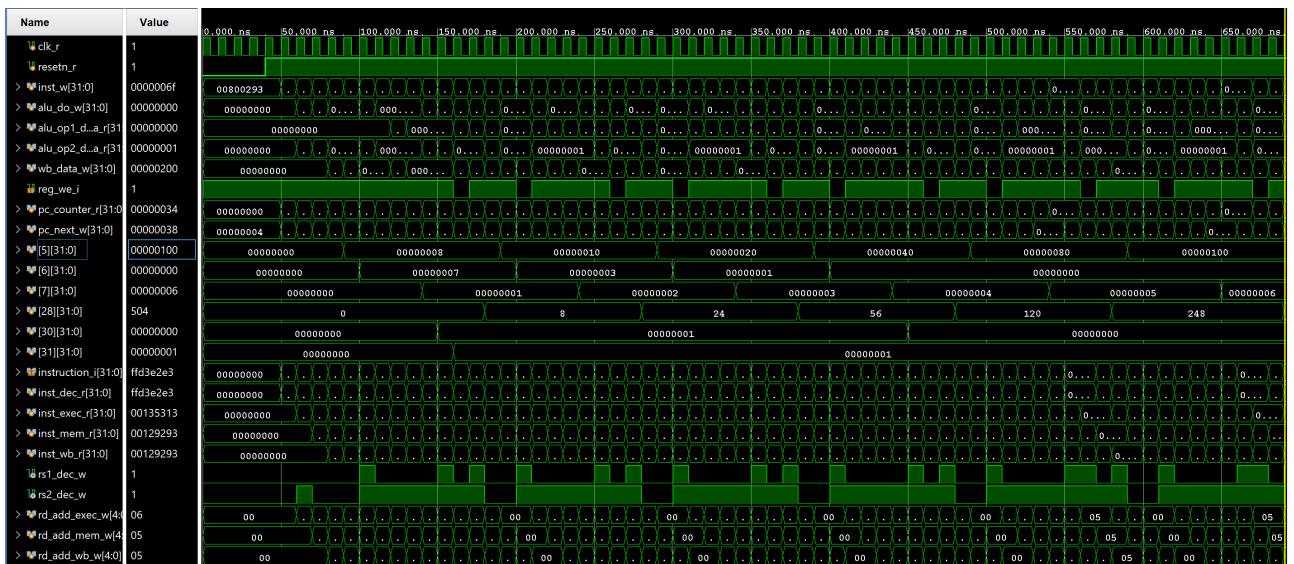


FIGURE 2.2 – Mise en évidence du problème dans le chronogramme de "mult.S"

La valeur finalement obtenue est donc :

$$524280 = 0x7ff8 = 8 + (8 << 1) + (8 << 2) + \dots + (8 << 15)$$

Les étages concernés sont donc principalement le **IF (Instruction Fetch)**, pour les dépendances de contrôles des **branch** et le **ID/EXE**, pour les dépendances de données sur les calculs et affectations de registres.

2.2 Correction du problème

2.2.1 Correction logicielle

Une instruction de type **NOP** correspond à l'instruction suivante sur l'architecture RISC-V :

```
ADDI x0,x0,0
0x0000 0013
```

En effet, le **NOP** correspond simplement à une instruction vide, qui ne fait "rien" à part incrémenter la valeur du compteur **PC**. Elle sert donc simplement à "combler" un trou entre 2 instructions.

On l'utilise majoritairement pour régler les problèmes de dépendance de données et de contrôle dans l'architecture pipeline.

Pour notre problème de dépendance dans le programme **mult.S**, on peut donc utiliser des **NOP** pour résoudre le problème. En effet, en "décalant" les instructions qui se chevauchent avec des **NOP**, on résoud la dépendance en permettant aux instructions qui les précèdent de finir leur travail au moment où l'instruction suivante en a besoin, et non pas après comme c'est actuellement le cas.

On propose donc la modification suivante :

Programme en assembleur de **mult_correc.S**

```
.section .start;
.globl start;

start :
    li t0, 0x8          //Operande 1 de la multiplication
    li t1, 0x7          //Operande de la multiplication
    li t2, 0x0          //Compteur de boucle
    li t3, 0x0          //Resultat de la multiplication
    li t4, 16           //Valeur de fin de boucle
    li t6, 0x1          //Valeur de comparaison du if

loop :
    andi t5, t1, 0x001 //t5 contient uniquement l'information du bit de poids faible de t0
    addi t2,t2,1        //Incrementation de la boucle
    NOP
    NOP
    bne t5,t6,endIf   //If(t1[0] == 1'b1)
    NOP
    NOP
    add t3,t3,t0

endIf :
    slli t0,t0,1        //Decalage a gauche de t0
    srli t1,t1,1        //Decalage a droite de t1
    bltu t2,t4,loop     //Verification de fin de boucle (t2 < t4?)
    NOP
```

NOP

```
lab1 :
    j lab1
    nop
```

```
.end start
```

Pour la dépendance de données, il est nécessaire que la donnée soit chargée au bon endroit pour l'étape **ID** de l'instruction qui est concernée par la dépendance. Donc, il faut que lorsque l'étage **IF** voit arriver l'instruction qui "subit" la dépendance, l'instruction qui "provoque" la dépendance se trouve à l'étage **MEM**.

Il faut donc insérer 3 instructions "entre" les 2 instructions "problématiques" afin de résoudre la dépendance.

Ici, on insère donc seulement 2 instructions **NOP**, car on "place" l'instruction "addi t2,t2,1" à la place d'une des 3 instructions **NOP**, afin de gagner un cycle, et donc en performance.

On s'intéresse dorénavant aux dépendances de contrôles. Ici, le problème ne vient pas du chargement des valeurs, mais de la modification (ou non) de la valeur du compteur **PC**. Ce type de dépendance concerne donc les instructions de type **BRANCH** et **JUMP (JAL/JALR)**. La dépendance s'effectue donc dès l'étage **IF**, et non à l'étage **ID** comme pour les dépendances de données.

Les étages de "décisions" sont donc :

- **MEM** : Si l'instruction est de type **BRANCH**. On aura donc besoin d'insérer 2 instructions, car la valeur du compteur **PC** est calculée lors de l'étage **ID**, on détermine la "branche" à choisir lors de l'étage **EXE** et on met à jour au début de l'étage **MEM**.
- **EXE** : Si l'instruction est de type **JAL**. On aura donc besoin d'insérer une seule instruction, car la valeur du compteur est calculée lors de l'étage **ID**, et mise à jour au début de l'étage **EXE**.
- **MEM** : Si l'instruction est de type **JALR**. On aura donc besoin d'insérer 2 instructions, car la valeur du compteur **PC** est calculée lors de l'étage **EXE** et on met à jour au début de l'étage **MEM**.

Ici, on inserera donc 2 instructions **NOP** après les instructions "bne t5,t6,endIf" et "bltu t2,t4,loop" (en bleu), ce qui permet de résoudre les problèmes de dépendances de contrôle.

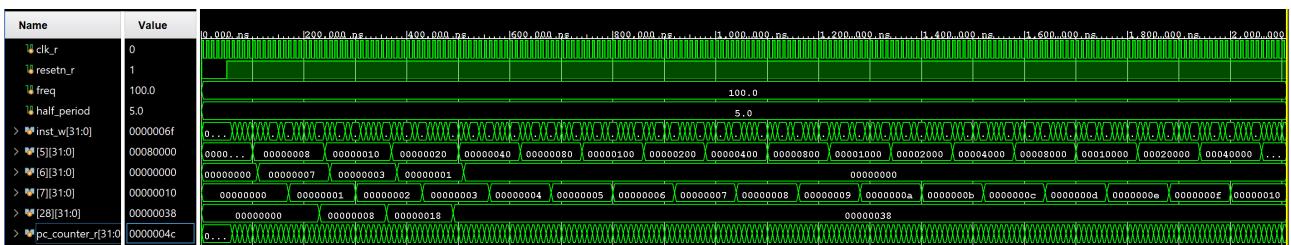


FIGURE 2.3 – Chronogramme résultant de l'exécution du programme "mult_correc.S"

On peut voir que l'on obtient bien la bonne valeur pour l'opération, on a $0x38 = 56$ à la fin de l'opération.

Comme l'on pouvait s'y attendre, l'ajout des instructions **NOP** affecte cependant les performances, et le programme met ici environ 45% plus de cycles à s'exécuter (on rajoute 6 instructions nulles), mais on obtient le bon résultat. Il existe cependant d'autres manières plus efficaces d'implémenter la gestion des dépendances, que ce soit du point de vue de la mémoire d'instructions (place prise par les programmes) ou des performances pures (temps d'exécution).

2.2.2 Correction matérielle : Interlock

L'implémentation matérielle des interlocks permet de ne plus inclure les **NOP** dans la mémoire d'instruction, ce qui permet donc de réduire la taille de ces derniers dans la mémoire, au dépend d'un peu de combinatoire supplémentaire dans le circuit.

2.2.2.1 Dépendances de données

Afin d'implémenter les interlocks, on commence par analyser certains signaux du sous-circuit **control_path** :

- **rs1_dec_w** : Adresse du registre source n°1, extrait de l'instruction pendant l'étage **ID**
- **rs2_dec_w** : Adresse du registre source n°2, extrait de l'instruction pendant l'étage **ID**
- **rd_add_XXX_w** : Adresse du registre de destination, extrait de l'instruction pendant les différents étages **ID**, **EXE**, **MEM**, **WB**

L'interlock repose sur la génération d'un signal logique **stall_w**, qui indique la détection d'une dépendance de données/contrôle. On va donc, pour détecter les dépendances de données, utiliser les signaux listés précédemment pour repérer les possibles dépendances. Pour celà, on cherche à savoir si l'instructions située à l'étage **ID** va nécessiter l'accès à un des registres qui va être modifié par une des instructions situées aux étages suivants.

Le signal est un signal logique sur 1 seul bit, donc on effectue un **xor** entre les adresses des registres sources et de destination, puis un **nor** sur tout les bits pour savoir si ils ont été exactement identiques. Ce qui donne la fonction logique suivante, qui vaut 1 si au moins un des registres sources est le même que le registre de destination d'un des étages :

$$\begin{aligned} stall_w = & (\sim \|(rs1_dec_w \oplus rd_add_exec_w)) + (\sim \|(rs2_dec_w \oplus rd_add_exec_w)) \\ & + (\sim \|(rs1_dec_w \oplus rd_add_mem_w)) + (\sim \|(rs2_dec_w \oplus rd_add_mem_w)) \\ & + (\sim \|(rs1_dec_w \oplus rd_add_wb_w)) + (\sim \|(rs2_dec_w \oplus rd_add_wb_w)) \end{aligned}$$

Néanmoins, on ne peut pas implémenter cette fonction de cette manière directement, en effet, toutes les instructions n'utilisent pas nécessairement les registres. Egalemennt, si les registres de destinations ou de sources sont le registre 0, alors il n'est pas nécessaire de générer un signal de stall. On crée donc un bloc **always_comb** qui prends en compte ceci, en modifiant la logique combinatoire du signal **stall_w** :

```
RV32I_B_INSTR : begin
    if (rs1_dec_w!=5'h00) stall_rs1_w = (~|(rs1_dec_w ^ rd_add_exec_w)) | (~|(rs1_dec_w ^ rd_add_mem_w))
        | (~|(rs1_dec_w ^ rd_add_wb_w));
    else stall_rs1_w = 1'b0;
    if (rs2_dec_w!=5'h00) stall_rs2_w = (~|(rs2_dec_w ^ rd_add_exec_w)) | (~|(rs2_dec_w ^ rd_add_mem_w))
        | (~|(rs2_dec_w ^ rd_add_wb_w));
    else stall_rs2_w = 1'b0;
    stall_w = stall_rs1_w | stall_rs2_w;
end
RV32I_I_INSTR_JALR : begin
    if (rs1_dec_w==5'h00) stall_w = 1'b0;
    else stall_w = (~|(rs1_dec_w ^ rd_add_exec_w)) | (~|(rs1_dec_w ^ rd_add_mem_w))
        | (~|(rs1_dec_w ^ rd_add_wb_w));
end
```

FIGURE 2.4 – Logique combinatoire pour 2 types d'instructions

Donc, on sépare la génération du signal **stall_w** entre les registres sources 1 et 2, qu'on ne génère que lorsque l'instruction fait appel au registre, et que son adresse est différente de 0. Pour les instructions qui ne font pas appels ni aux registres 1, ni au 2, on fixe le signal **stall_w** à 0, car il n'y aura pas de problèmes de dépendances de données.

On teste donc cette solution matérielle avec le programme de la multiplication logicielle. Pour se faire, on garde seulement la correction logicielle pour les dépendances de contrôle (**en bleu**), et on retire ceux concernant les dépendances de données (**en vert**).

On obtient alors le chronogramme suivant :

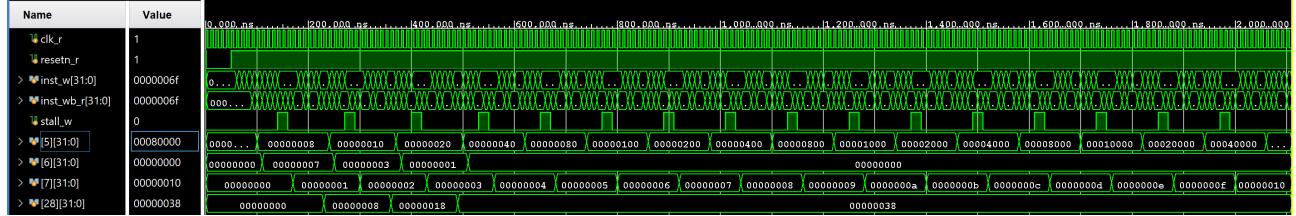


FIGURE 2.5 – Chronogramme résultant de l'execution du programme "mult_correc_data_only.S"

On peut donc bien observer la bonne génération du signal **stall_w**, et le bon résultat final. En zoomant sur les premières boucles, on peut mieux observer les NOP qui sont ajoutés sur le signal **inst_wb_r** par le signal **stall_w**. Ils apparaissent pendant 2 cycles d'horloge, après 2 cycles, ce qui correspond bien à la propagation entre les différents étages du pipeline (ID vers WB = 2 étages).

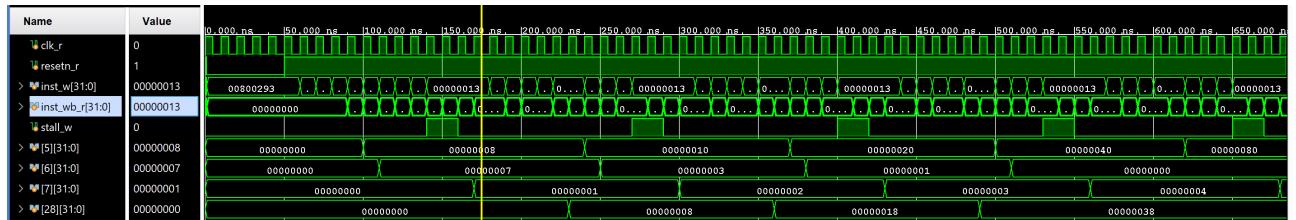


FIGURE 2.6 – Zoom sur le chronogramme résultant de l'execution du programme "mult_correc_data_only.S"

2.2.2.2 Dépendances de contrôle

GESTION DES SAUTS

La gestion des instructions de type **JALR** n'est pas implémentée, on s'intéresse donc uniquement aux dépendances causées par les instructions de type **JAL** (En effet, actuellement l'opération effectuée lorsqu'une instruction de type jalr est détectée est : "PC = PC + RS1" au lieu de "PC = RS1 + Imm"). (Ce type d'instructions est cependant correctement supporté dans le code qui est rendu avec ce rapport)

Pour gérer les dépendances, dès lors qu'une instruction de type **JAL** est détectée dans l'étage **ID**, on remplace l'instruction qui est récupérée à l'étage **IF** par une instruction **NOP**. Ainsi, étant donné que lorsque l'étage **ID** se termine, la valeur du compteur **PC** est à jour, on a simplement pas décodé, ni executé l'instruction suivante, qui de toute façon n'aurait pas du être récupérée à cause du saut.

On observe donc la partie finale du programme, une boucle infinie sur la fonction **lab1**. On vérifie bien ici la génération du signal **fetch_jump_w**, qui permet d'indiquer si l'on veut remplacer l'instruction de l'étage **IF** par un **NOP**.

GESTION DES BRANCHEMENTS

Afin de gérer les branchements, il est cette fois nécessaire de s'intéresser à l'instruction pendant l'étage **EXE**. En effet, la condition du branchement n'est calculée qu'à cet étage, on "laisse" donc les instructions suivantes s'exécuter en attendant cet étage (afin de maximiser les

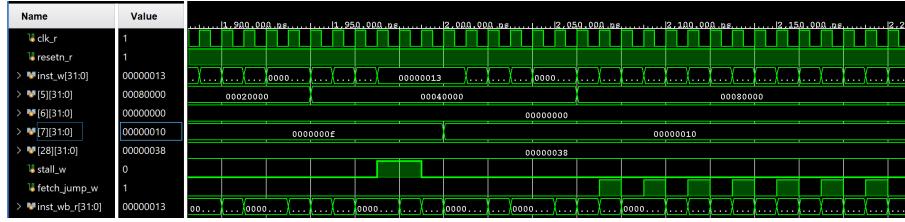


FIGURE 2.7 – Zoom sur la partie **JAL** du chronogramme résultant de l'execution du programme "mult_correc_data_only.S"

performances si le chemin n'est pas pris étant donné que nous n'avons ici pas de prédition). Dès lors qu'une instruction de type **BRANCH** est détectée à l'étage **EXE**, on regarde donc si la condition a été vérifiée via le signal **branch_taken_w**. Alors, on affecte la valeur de ce dernier signal à un nouveau signal que l'on crée, **fetch_branch_w** et à **fetch_jump_w**. Le nouveau signal permet de remplacer l'instruction de l'étage **ID** par un **NOP**, de la même manière que via le signal **stall_w**. Cependant, ici on ne bloque pas le circuit, on remplace l'instruction par un **NOP**. La différence principale entre **fetch_branch_w** et **stall_w** est donc que **stall_w** met en pause le circuit en stoppant le compteur **PC** et la mise à jour de l'étage **IF**, alors que **fetch_branch_w** ne stoppe aucune de ces mises à jours.

On corrige donc bien correctement la dépendance de contrôle, comme on peut le voir sur ce chronogramme, le programme s'exécute correctement, sans aucune correction logicielle :

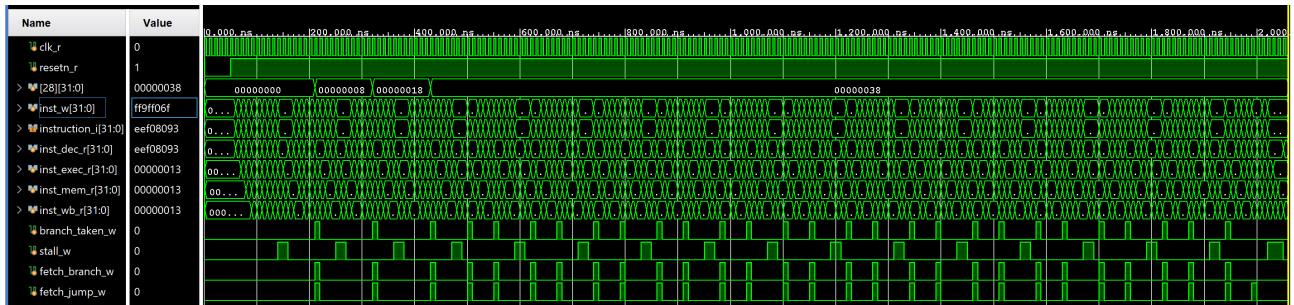


FIGURE 2.8 – Chronogramme résultant de l'exécution du programme "mult.S" corrigé via hardware

Si on zoom sur une partie de l'exécution pour voir plus précisément, on peut voir que lorsque le signal **branch_taken_w** est à 0 (donc que, par exemple, le "if" est vérifié), alors l'opération est effectuée. Si le signal est à 1 par contre, on peut bien voir l'ajout de 2 **NOP** supplémentaires pour remplacer les instructions préalablement chargée. La dépendance de contrôle est donc prise en compte et ne posera pas de problème.



FIGURE 2.9 – Zoom sur le chronogramme résultant de l'exécution du programme "mult.S" corrigé via hardware

Un problème qui peut être soulevé est la présence d'instructions **JAL** juste après une ins-

truction **BRANCH**, cependant, comme on peut le voir également sur ce zoom, on remplace correctement l'instruction **JAL** par un **NOP** et le compteur **PC** prends bien la valeur imposée par l'instruction **BRANCH**. Sauf dans le cas où l'instruction **BRANCH** ne vérifie pas sa condition, comme dans ce dernier chronogramme, où l'on peut bien voir que l'instruction **JAL** s'effectue correctement.

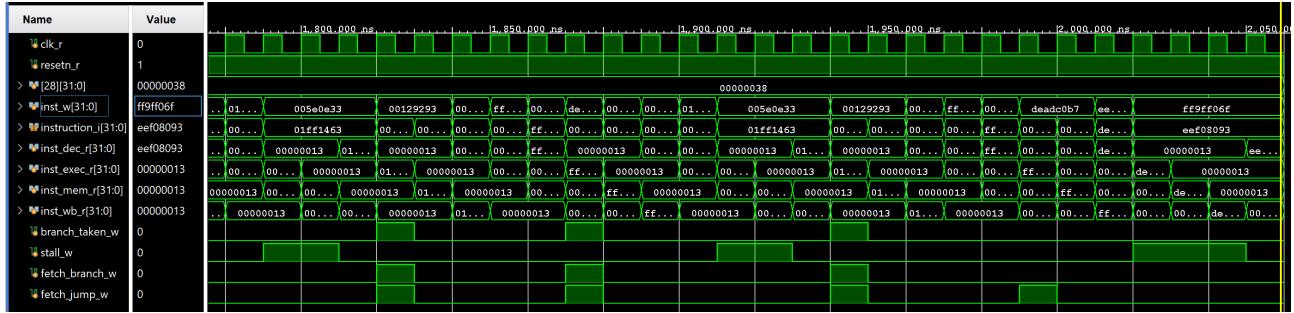


FIGURE 2.10 – Zoom sur la fin du chronogramme résultant de l'exécution du programme "mult.S" corrigé via hardware

2.2.3 Correction matérielle : Bypass

On cherche ici à implémenter un bypass entre la sortie de l'**ALU** et ses entrées **OP1** et **OP2**, afin de gérer les dépendances de données "directes" (entre 2 instructions successives). On s'intéresse donc aux étages **ID** et **EXE**, on va donc créer un "pont" entre les 2 étages, qui ne sera pas concerné par le "mur" de registres. Ainsi, la sortie de l'**ALU** sera directement reliée aux registres qui contiennent les données de ses entrées **OP1** et **OP2**.

Afin de mettre en évidence l'intérêt du bypass, on utilisera le programme suivant :

Programme en assembleur de **bypass.S**

```

.section .start;
.globl start;

start :
    li t1, 0x10          //Valeur de départ 1
    li t2, 0x20          //Valeur de départ 2
    li t6, 0x1f          //Valeur de fin de boucle
loop :
    addi t5,t5,1         //Incrementation de la boucle
    add t3,t1,t2
    or t4, t1, t3
    and t1, t3, t4
    bltu t5,t6,loop     //Verification de fin de boucle (t5 < t6?)

lab1 :
    li ra, 0xDEADBEEF
    j lab1

.end start

```

Il comporte deux dépendances de données et une dépendance de contrôle (**en rouge**). Pour les dépendances de données, on a 2 cas différents, la première (**en bleu**) est une dépendance

"simple" et "directe" avec l'instruction précédente et le 2nd opérande. La 2nde dépendance (**en vert**) est quand à elle plus complexe, en effet, les 2 opérandes dépendent des instructions précédentes, **t4** provient de l'instruction précédente et **t3** est modifié 2 instructions auparavant.

On execute donc ce programme sans implémenter le bypass, et on obtient ceci :

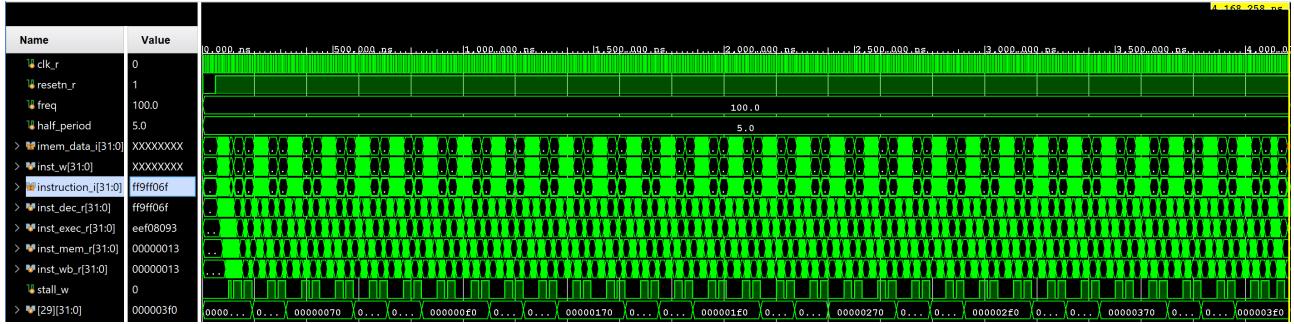


FIGURE 2.11 – Chronogramme résultant de l'exécution du programme "bypass.S"

On remarque que, pour chaque boucle, on a 2 utilisations du signal **stall_w**, et 3 insertions de **NOP**. Le programme termine de s'exécuter au bout de 4170ns, valeur que l'on comparera à celle obtenue après l'implémentation du bypass.

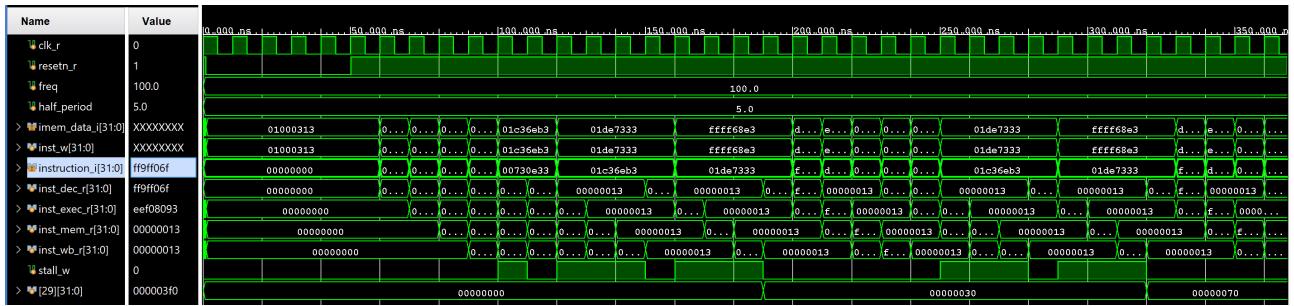


FIGURE 2.12 – Zoom sur 2 boucles du programme "bypass.S"

On voit ici très clairement les différents **NOP** qui se propagent dans les étages pour les dépendances de données (3) et de contrôle (2). On peut aussi voir que, lors de la première boucle, la ligne du programme "add t3,t1,t2" provoque un signal **stall_w** pendant 1 cycle, car le registre **t2** n'est pas encore mis à jour dans le banc de registres.

On cherche donc maintenant à implémenter le bypass afin d'améliorer les performances. On commence donc par modifier la génération du signal **stall_w**, en effet, il n'est plus nécessaire de générer ce dernier si on a les égalités :

$$\begin{aligned} \text{rs1_dec_w} &== \text{rd_add_exec_w} \\ \text{rs2_dec_w} &== \text{rd_add_exec_w} \end{aligned}$$

Cependant, on génère respectivement les signaux :

$$\begin{aligned} \text{alu_src1_bypass_w} \\ \text{alu_src2_bypass_w} \end{aligned}$$

Ceci nous permet donc, dans la sélection du signal de sélection des entrées de l'**ALU**, de choisir l'entrée qui correspond à la sortie de l'**ALU**. Voici les parties du codes modifiées (on ajoute également dans le package les valeurs du select de l'**ALU**, et on change dans le "core" la taille du signal **alu_src2_i**) :

```
1 always_comb begin : stall_comb
2   rd_used_exe_w = rd_used(opcode_exec_w);
```

```

3      rd_used_mem_w = rd_used(opcode_mem_w);
4      rd_used_wb_w = rd_used(opcode_wb_w);
5      case (opcode_dec_w)
6          RV32I_R_INSTR, RV32I_S_INSTR, RV32I_B_INSTR : begin
7              fetch_jump_dec_w = 1'b0;
8              if (rs1_dec_w==5'h00) begin
9                  stall_rs1_w = 1'b0;
10                 alu_src1_bypass_w = 1'b0;
11             end
12             else if ((rs1_dec_w==rd_add_exec_w) && rd_used_exe_w) begin
13                 stall_rs1_w = 1'b0;
14                 alu_src1_bypass_w = 1'b1;
15             end
16             else begin
17                 stall_rs1_w = ((~(rs1_dec_w ^ rd_add_mem_w)) && rd_used_mem_w) | ((~(rs1_dec_w ^ rd_add_wb_w)) && rd_used_wb_w);
18                 alu_src1_bypass_w = 1'b0;
19             end
20             if (rs2_dec_w==5'h00) begin
21                 stall_rs2_w = 1'b0;
22                 alu_src2_bypass_w = 1'b0;
23             end
24             else if ((rs2_dec_w==rd_add_exec_w) && rd_used_exe_w) begin
25                 stall_rs2_w = 1'b0;
26                 alu_src2_bypass_w = 1'b1;
27             end
28             else begin
29                 stall_rs2_w = ((~(rs2_dec_w ^ rd_add_mem_w)) && rd_used_mem_w) | ((~(rs2_dec_w ^ rd_add_wb_w)) && rd_used_wb_w);
30                 alu_src2_bypass_w = 1'b0;
31             end
32             stall_w = stall_rs1_w | stall_rs2_w;
33         end
34         RV32I_I_INSTR_JALR, RV32I_I_INSTR_LOAD, RV32I_I_INSTR_OPER, RV32I_I_INSTR_FENCE,
35         RV32I_I_INSTR_ENVCSR : begin
36             fetch_jump_dec_w = 1'b0;
37             alu_src2_bypass_w = 1'b0;
38             if (rs1_dec_w==5'h00) begin
39                 stall_w = 1'b0;
40                 alu_src1_bypass_w = 1'b0;
41             end
42             else if ((rs1_dec_w==rd_add_exec_w) && rd_used_exe_w) begin
43                 stall_w = 1'b0;
44                 alu_src1_bypass_w = 1'b1;
45             end
46             else begin
47                 stall_w = ((~(rs1_dec_w ^ rd_add_mem_w)) && rd_used_mem_w) | ((~(rs1_dec_w ^ rd_add_wb_w)) && rd_used_wb_w);
48                 alu_src1_bypass_w = 1'b0;
49             end
50         end
51         RV32I_J_INSTR : begin
52             stall_w = 1'b0;
53             fetch_jump_dec_w = 1'b1;
54             alu_src1_bypass_w = 1'b0;
55             alu_src2_bypass_w = 1'b0;
56         end
57         default : begin
58             stall_w = 1'b0;
59             fetch_jump_dec_w= 1'b0;
60             alu_src1_bypass_w = 1'b0;
61             alu_src2_bypass_w = 1'b0;
62         end
63     endcase
64     fetch_branch_w = branch_taken_w;
65     fetch_jump_exe_w = branch_taken_w;
66     fetch_jump_w = fetch_jump_dec_w | fetch_jump_exe_w;
67 end : stall_comb

```

Code 2.1 – Modifications apportées pour le stall et bypass

```

1  always_comb begin : alu_src1_comb
2      if (alu_src1_bypass_w) alu_src1_o = SEL_OP1_DIRECT_ALU;
3      else begin
4          case (opcode_dec_w)
5              RV32I_U_INSTR_LUI: alu_src1_o = SEL_OP1_IMM;
6              RV32I_U_INSTR_AUIPC: alu_src1_o = SEL_OP1_PC;

```

```

7      default: alu_src1_o = SEL_OP1_RS1;
8      endcase
9  end
10
11
12 always_comb begin : alu_src2_comb
13   if (alu_src2_bypass_w) alu_src2_o = SEL_OP2_DIRECT_ALU;
14   else begin
15     case (opcode_dec_w)
16       RV32I_I_INSTR_OPER: alu_src2_o = SEL_OP2_IMM;
17       RV32I_I_INSTR_LOAD: alu_src2_o = SEL_OP2_IMM;
18       RV32I_U_INSTR_AUIPC: alu_src2_o = SEL_OP2_IMM;
19       RV32I_S_INSTR: alu_src2_o = SEL_OP2_IMM;
20       default: alu_src2_o = SEL_OP2_RS2;
21     endcase
22   end
23 end

```

Code 2.2 – Modifications apportées aux contrôles de l’ALU

```

1 //mux to select ALU op1
2 always_comb begin : alu_src1_mux_comb
3   case (alu_src1_i)
4     SEL_OP1_RS1: alu_op1_data_w = rs1_data_w;
5     SEL_OP1_IMM: alu_op1_data_w = imm_w;
6     SEL_OP1_PC: alu_op1_data_w = pc_counter_r;
7     SEL_OP1_DIRECT_ALU: alu_op1_data_w = alu_do_w;
8     default: alu_op1_data_w = 0;
9   endcase
10 end
11
12 //mux to select ALU op2
13 always_comb begin : alu_src2_mux_comb
14   case (alu_src2_i)
15     SEL_OP2_RS2: alu_op2_data_w = rs2_data_w;
16     SEL_OP2_IMM: alu_op2_data_w = imm_w;
17     SEL_OP2_DIRECT_ALU: alu_op2_data_w = alu_do_w;
18     default: alu_op2_data_w = 0;
19   endcase
20 end

```

Code 2.3 – Modifications apportées aux choix des entrées de l’ALU

On précise ici néanmoins que cette implémentation est loin d’être parfaite, et ne permet pas d’utiliser l’ensemble des instructions de notre architecture. En effet, des instructions de type **S** ou **LOAD** ont des fonctionnements différents qui ne sont pas pris en compte ici. Nous nous contenterons donc de cette implémentation pour le moment, mais elle sera vouée à être modifiée par la suite.

On relance ainsi le programme précédent pour observer l’amélioration qu’apporte le bypass.

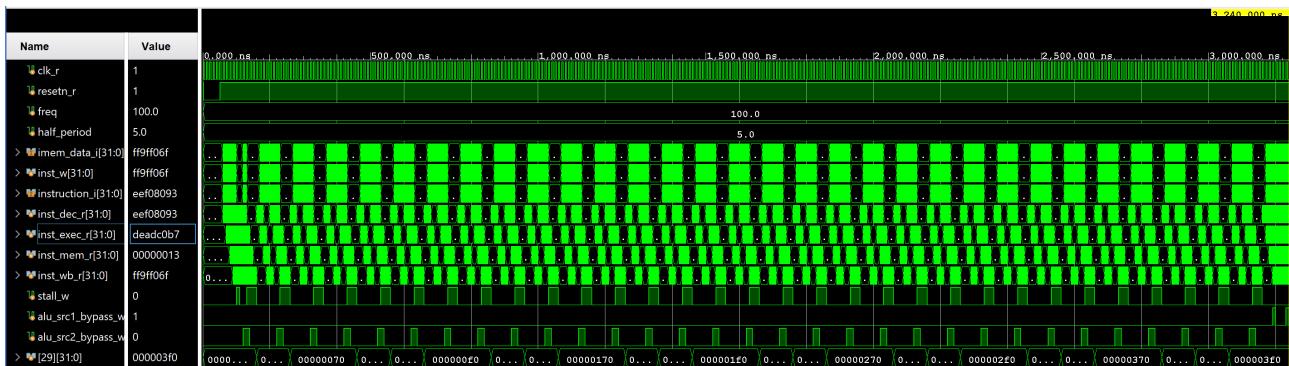


FIGURE 2.13 – Chronogramme résultant de l’execution du programme "bypass.S", avec l’implémentation hardware

On retrouve bien la même valeur finale dans le registre **t4** : `0x3f0`, donc les dépendances de données restent bien corrigées. Mais, plus intéressant, le programme se termine maintenant en

seulement 3240ns. On a donc une augmentation significative des performances. Néanmoins, on retrouve encore le signal **stall_w**.

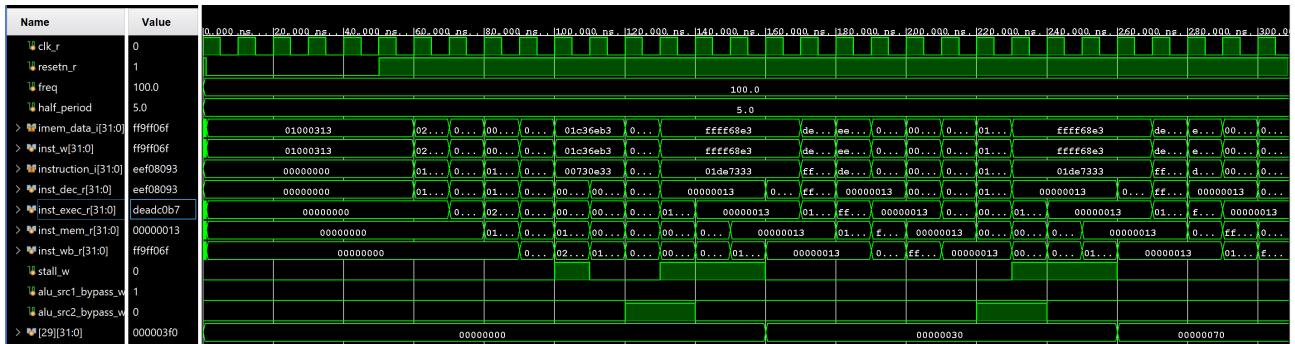


FIGURE 2.14 – Zoom sur 2 boucles du programme "bypass.S", avec l'implémentation hardware

On repère ici la raison de l'apparition du **stall_w**, en effet, comme on l'a expliqué lors de l'explication des dépendances de données du programme, la ligne verte possède une dépendance de données par rapport aux 2 instructions précédentes, donc, le bypass se déclenche pour l'instruction qui la précède, mais le signal **stall_w** se déclenche pour l'instruction encore avant. Cette dernière prends donc "le dessus" et l'instruction doit attendre au moins un cycle d'horloge. Cependant, après ce cycle d'attente, le bypass n'est plus possible car la valeur de sortie de l'**ALU** est passée à l'étage **MEM**. On se retrouve donc avec une dépendance de données "classique", et on utilise un **interlock**, donc 3 **NOP** sont insérés.

Toutes les mesures hardware que l'on a implémenter on donc eu pour but de rapprocher l'architecture pipeline de l'architecture monocycle du point de vue de la logique de programmation. On peut dorénavant coder les instructions dans l'ordre que l'on veut et sans s'inquiéter des **NOP**, tout en obtenant le résultat attendu. Cependant, seul le bypass permet d'améliorer les performances de cette architecture, dans certains cas spécifiques. L'amélioration reste assez radicale comme on l'a mis en évidence ici avec une boucle assez simple.

Chapitre 3

Implémentation d'une mémoire cache

3.1 Questions préliminaires

3.1.1 Cache mémoire "Direct"

On considère une mémoire cache dont l'adressage se fait de la manière suivante :

31 ... 10	9 ... 4	3 ... 0
TAG	INDEX	OFFSET

L'offset est de 4 bits, donc on pourra stocker 16 octets par ligne (2^4), ce qui correspond à 4 mots par ligne (1 mot = 4 octets).

L'index est de 6 bits, on aura donc $2^6 = 64$ lignes. Donc, étant donné que l'on a 4 mots par lignes, on a 256 entrées possible dans la mémoire cache.

Le tag est de 22 bits, on pourra donc adresser 2^{22} lignes de la mémoire principale, elle aura donc une capacité maximale de $2^{26} \text{ octets} = 67MB$.

3.1.2 Performance

On considère un processeur RISCV-RV32i de fréquence 100MHz et possédant une mémoire cache L1, dont les paramètres sont les suivants :

L1 HIT Time = 1 cycle

MISS Penalty = 10 cycles

Après avoir effectué un benchmark, on constate le taux d'accès suivant :

L1 HIT Rate = 90%

On en déduit donc le taux d'accès à la mémoire principale :

MISS Rate = 10%

On en déduit donc un temps d'accès moyen à la mémoire de :

$$T_{ps\ moy} = \frac{1}{100*10^6} + (0.1 * \frac{10}{100*10^6}) = 20ns$$

Attention, la pénalité est le temps d'accès SUPPLEMENTAIRE, on tente d'accéder à la mémoire L1 (1 cycle) ET on tente d'accéder à la DRAM (10 cycles), au total on a donc 11 cycles de temps d'accès si on doit accéder à la DRAM.

3.2 Cache d'instructions direct

Dans un premier temps, on se propose d'implémenter une mémoire cache relativement simple ("directe") destinée à stocker seulement les instructions du processeur.

On travaillera donc en 3 parties pour permettre l'implémentation de cette mémoire sur notre processeur. On commence donc par designer la mémoire cache en elle-même, puis on modifiera la mémoire principale et enfin on adaptera le processeur afin qu'il puisse fonctionner avec notre cache d'instructions.

3.2.1 Crédation du module "direct_cache"

3.2.1.1 Crédation du module

On commence par définir des paramètres pour l'adressage de la mémoire : **ByteOffsetBits**, **IndexBits** et **TagBits**. Ils seront donc définis comme des paramètres du module (et non des paramètres locaux), afin qu'ils puissent être modifiés à volonté. On donnera comme valeurs de base :

- ByteOffsetBits = 4
- IndexBits = 6
- TagBits = 22

Ceci nous permet donc, avec les paramètres locaux, de définir le nombre de mots par lignes, ainsi que le nombres de lignes :

- NrWordsPerLine = $\frac{2^{\text{ByteOffsetBits}}}{4}$
- NrLines = $2^{\text{IndexBits}}$
- LineSize = $32 * \text{NrWordsPerLine}$

On commence maintenant la conception de notre cache. Tout d'abord, on commence par segmenter l'adresse en 3 signaux de tailles différentes, représentant les valeurs de l'offset, de l'index et du tag. Ces signaux devront être remis à 0 en cas de reset et ne seront mis à jour que si le signal **read_en_i** est actif.

```
1 // Segmentation of the input adress into smaller signals to facilitate code writing
2 always_comb begin : adress_segmentation
3     if (rstn_i==1'b0) begin
4         read_offset = 0;
5         read_index = 0;
6         read_tag = 0;
7     end
8     else if (read_en_i) begin
9         read_offset = addr_i[ByteOffsetBits-1:2];
10        read_index = addr_i[IndexBits+ByteOffsetBits-1:ByteOffsetBits];
11        read_tag = addr_i[TagBits+IndexBits+ByteOffsetBits-1:IndexBits+ByteOffsetBits];
12    end
13 end : adress_segmentation
```

Code 3.1 – Affectation des valeurs du tag, de l'index et de l'offset selon l'adresse

L'étape suivante consiste à créer les registres représentant la mémoire de la cache. Il est donc nécessaire d'instancier des signaux, qui viendront "mémoriser" les valeurs de 3 éléments :

- Le bit de validité associé à chaque ligne, pour déterminer si cette dernière possède des informations, ou si elle a été réinitialisé et nécessite donc d'être mise à jour
- Le tag associé à la ligne
- La ligne comprenant les mots (les données, ici instructions)

```
1 //Cache elements
2 logic cache_line_validity[NrLines-1:0];
3 logic [TagBits-1:0] cache_tags[NrLines-1:0];
4 logic [LineSize-1:0] cache_words[NrLines-1:0];
```

Code 3.2 – Nom des registres représentant la mémoire de la cache

On crée donc ensuite les registres, avec les conditions permettant de réinitialiser, écrire et lire dans ces derniers.

```

1 //Registers implementation
2 generate
3   for (genvar i=0;i<NrLines;i++) begin : registers_cache
4     always_ff @(posedge clk_i or negedge rstn_i) begin : write_cache
5       if(rstn_i==1'b0) begin
6         cache_line_validity[i] <= 1'b0;
7         cache_tags[i] <= 0;
8         cache_words[i] <= 0;
9       end
10      else if (mem_read_valid_i && (read_index==i)) begin
11        cache_line_validity[i] <= 1'b1;
12        cache_tags[i] <= read_tag;
13        cache_words[i] <= mem_read_data_i;
14      end
15    end : write_cache
16    always_comb begin : read_cache
17      registers_output_lines_tags[i] <= cache_tags[i];
18      registers_output_all[i] <= cache_words[i];
19    end : read_cache
20  end : registers_cache
21 endgenerate

```

Code 3.3 – Conditions d’écritures (synchrone) et de lecture (asynchrone) dans les registres

On considèrera ici les lectures comme asynchrones pour cette cache de niveau 1, étant donné sa faible taille. Les écritures seront quand à elle synchrones, en un seul cycle.

Néanmoins, il est désormais nécessaire de déterminer si la donnée demandée par le processeur est bien présente dans la cache. On crée donc la logique qui permet de générer le signal logique **hit_w**, qui détermine si la donnée présente dans la ligne **read_index** est la bonne.

```

1 //Generation of the hit signal, stating if the data is present in the cache
2 always_comb begin : hit_or_miss
3   if (cache_tags[read_index]==read_tag && read_en_i && cache_line_validity[read_index])
4     hit_w = 1'b1;
5   else hit_w = 1'b0;
end : hit_or_miss

```

Code 3.4 – Génération du signal de **hit**

Dans le cas où l’instruction n’est pas présente dans la cache, il est donc nécessaire d’envoyer une demande à la mémoire de niveau supérieur pour cette donnée (ici une mémoire L2 ou principale). Pour celà, on renvoie l’adresse demandée par le processeur, avec la valeur de l’offset mise à zéro, car on vient chercher toute une ligne, et non un mot de 32 bits.

```

1 //Memory outputs
2 assign mem_addr_o = {read_tag,read_index,read_offset_zero};

```

Code 3.5 – Assignation de l’adresse mémoire

Il est donc désormais nécessaire de choisir entre envoyer la valeur présente dans la cache, celle renvoyée par la mémoire supérieure ou demander à la mémoire supérieure une nouvelle ligne. Pour celà, on s’appuie sur les différents signaux en entrée, ou ceux générés par la logique de notre cache : **read_en_i**, **mem_read_valid_i** et **hit_w**.

```

1 //Registers output logic and data assignation
2 always_comb begin : mux_read_out
3   if (mem_read_valid_i && read_en_i) begin
4     registers_output_line = mem_read_data_i;
5     mem_read_en_o = 1'b0;
6   end
7   else if (hit_w && read_en_i) begin
8     registers_output_line = registers_output_all[read_index];
9     mem_read_en_o = 1'b0;
10  end
11  else if(read_en_i&&rstn_i) begin
12    mem_read_en_o = 1'b1;
13  end
14  else begin
15    mem_read_en_o = 1'b0;
16  end
17 end : mux_read_out

```

Code 3.6 – Assignation des différents signaux de contrôle

La mémoire supérieure et les registres de la cache nous renvoient des lignes complètes (ici de 128bits), mais on cherche ici à obtenir seulement un mot de 32bits, il est donc nécessaire de découper cette ligne en mots de 32bits.

```

1 //Data reorganization
2 generate
3   for (genvar k=0;k<NrWordsPerLine;k++) begin : memory_input_distribution
4     assign registers_output_words[k] = registers_output_line[(k+1)*32-1:k*32];
5   end : memory_input_distribution
6 endgenerate

```

Code 3.7 – Reorganisation de la ligne en mots

Enfin, on implémente la logique permettant d'affecter le mot correspondant à l'offset inclus dans l'adresse, à la sortie de la mémoire cache, destinée au processeur. On en profite également pour créer le signal de sortie **read_valid_o**, qui indique au processeur si la valeur présente sur le bus de sortie correspond bien à la valeur qu'il a demandé.

```

1 //Read output logic
2 always_comb begin : register_output_formalized
3   if ((mem_read_valid_i || hit_w) && read_en_i) begin
4     read_valid_o <= 1'b1;
5     read_word_o <= registers_output_words[read_offset];
6   end
7   else read_valid_o <= 1'b0;
8 end : register_output_formalized

```

Code 3.8 – Génération des signaux de sortie pour le processeur

3.2.1.2 Test du module

Afin de vérifier le bon fonctionnement de notre cache, nous allons simuler via une testbench des interactions entre notre cache et un processeur, mais également avec une mémoire externe, le tout "à la main" afin de maîtriser de A à Z les interactions pour vérifier :

- Si les lignes s'actualisent bien dans la cache lors de la première lecture
- Si la valeur donnée par la mémoire est bien directement redonnée au processeur, et écrite au début du cycle suivant
- Si les lignes peuvent bien être lue après une écriture
- Si les lignes sont bien remplacées en cas de tag différents

On obtient alors le chronogramme suivant :

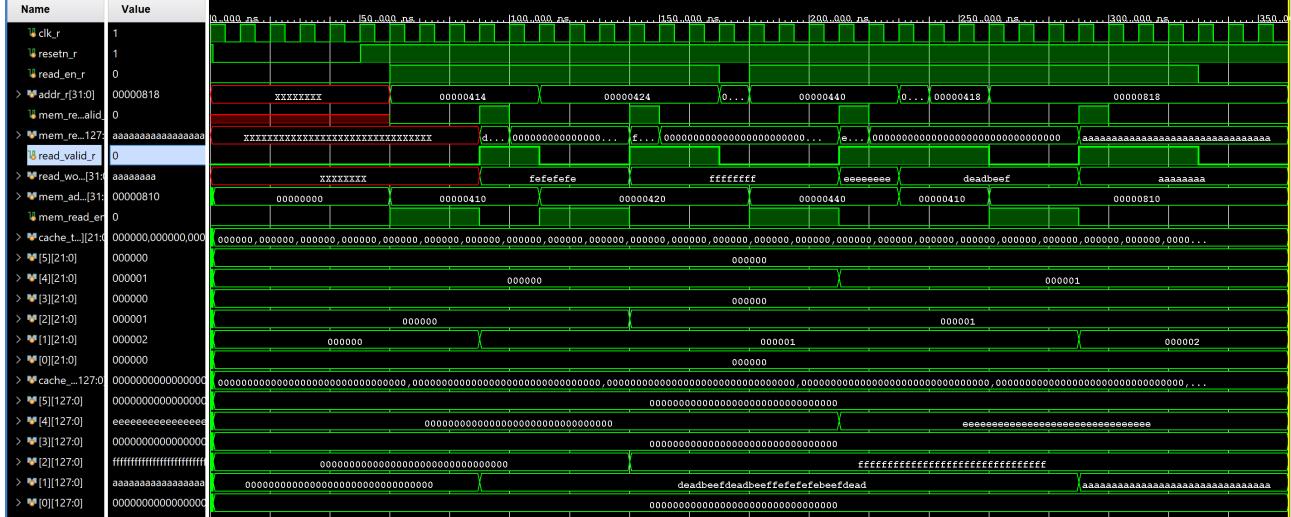


FIGURE 3.1 – Chronogramme de la testbench du "direct_cache" seul

On précise que l'on attend seulement 3 cycles entre le moment où on demande un nouveau mot à la cache, et celui où l'on envoie la ligne sur le port correspondant à l'entrée depuis la

mémoire **mem_read_data_i**.

Les données sont correctement enregistrées dans la cache et envoyée au processeur. Lors de changements de l'adresse, le signal **read_valid_o** s'actualise correctement suivant que la valeur soit présente ou non dans la cache.

On commence donc à s'intéresser à l'adaptation des autres composants, à commencer par la mémoire principale.

3.2.2 Modification de la mémoire principale

Afin d'adapter la mémoire, il est tout d'abord nécessaire d'ajouter un signal de sortie, pour indiquer que la donnée demandée est prête, on l'appellera **data_ready_o**.

Ensuite, il est nécessaire d'ajouter quelques paramètres afin de pouvoir adapter la mémoire à la cache avec laquelle elle doit communiquer, pour que la longueur des lignes notamment soit cohérente. Dans le même temps, on définit un paramètre qui nous permettra de choisir le nombre de cycle de latence que l'on veut associer à une lecture/écriture dans notre mémoire.

```

1 module dram_emulation_mem #(
2     parameter ByteOffsetBits = 5,
3     parameter SIZE = 4096,      //In bytes
4     parameter INIT_FILE = "",
5     parameter CYCLE_LATENCE = 10,
6     localparam NB_WORDS_LINE = (2**ByteOffsetBits)/4,
7     localparam LINE_SIZE = 32 * NB_WORDS_LINE,
8     localparam ADDR_LEN = $clog2(SIZE)
9 ) (

```

Code 3.9 – Paramètres du module de mémoire

Ensuite, il est désormais nécessaire de stocker des lignes de données, et non des mots pour pouvoir communiquer par "paquets" avec la mémoire cache directement inférieure. On lit donc, initialement des valeurs par mots de 32bits, puis, on les réarrange par lignes de **LINE_SIZE** bits.

```

1 //Address for the memory by lines
2 logic [31-ByteOffsetBits+1:0] add_w;
3 assign add_w = add_i[31:ByteOffsetBits];
4
5 //Signals for memorization and latency
6 logic [31:0] mem[SIZE];
7 logic [LINE_SIZE-1:0] mem_line[SIZE/NB_WORDS_LINE];
8 logic [CYCLE_LATENCE-1:0][LINE_SIZE-1:0] data_delayed_w;
9 logic [CYCLE_LATENCE-1:0] propag_bit_w;
10 logic [CYCLE_LATENCE-1:0][LINE_SIZE-1:0] data_delayed_write_w;
11 logic [CYCLE_LATENCE-1:0] propag_bit_write_w;
12
13 //Initialisation of the memory cells
14 initial begin
15     if (INIT_FILE == "") $readmemh("../../../../../firmware/zero.hex", mem);
16     else $readmemh(INIT_FILE, mem);
17     for (int lines=0;lines<SIZE/NB_WORDS_LINE;lines++) begin
18         for (int column=0; column<NB_WORDS_LINE; column++) begin
19             mem_line[lines][(column*32)+:32] = mem[NB_WORDS_LINE*lines+column];
20         end
21     end
22 end

```

Code 3.10 – Initialisation des valeurs de la mémoire

Enfin, on modifie la logique de lecture de la mémoire afin d'ajouter des registres de propagation, qui vont permettre de simuler la latence associée aux lectures. En effet, les registres étant placés en série, le nombres de cycle de latence correspond alors au nombre de registres à traverser pour "sortir" de la mémoire.

```

1 //Register for the writing in memory
2 generate
3   for (genvar wr=0;wr<CYCLE_LATENCE-1;wr++) begin : REGISTER_LATENCE_WRITE
4     always_ff @(posedge clk_i) begin
5       if (write_enable_i && propag_bit_write_w[wr]) begin
6         data_delayed_write_w[wr+1] <= data_delayed_write_w[wr];
7         propag_bit_write_w[wr+1] <= propag_bit_write_w[wr];
8       end
9       else begin
10         data_delayed_write_w[wr+1] <= 0;
11         propag_bit_write_w[wr+1] <= 1'b0;
12       end
13     end
14   end : REGISTER_LATENCE_WRITE
15 endgenerate
16
17 always_ff @(posedge clk_i) begin : wmem
18   if (write_enable_i && propag_bit_write_w[CYCLE_LATENCE-1]) begin
19     mem_line[add_w] <= data_delayed_write_w[CYCLE_LATENCE-1];
20     write_valid_o <= 1'b1;
21   end
22   else write_valid_o <= 1'b0;
23 end
24
25 //Logic for the reading operation
26 always_comb begin : rmem
27   if (read_enable_i == 1'b1) begin
28     data_delayed_w[0] = mem_line[add_w];
29     propag_bit_w[0] = 1'b1;
30   end
31   else begin
32     data_delayed_w[0] = 0;
33     propag_bit_w[0] = 1'b0;
34   end
35 end
36
37 //Propagation registers aimed at simulating the read latency of the memory
38 generate
39   for (genvar i=0;i<CYCLE_LATENCE-1;i++) begin : REGISTER_LATENCE_READ
40     always_ff @(posedge clk_i) begin
41       if (read_enable_i && propag_bit_w[i]) begin
42         data_delayed_w[i+1] <= data_delayed_w[i];
43         propag_bit_w[i+1] <= propag_bit_w[i];
44       end
45       else begin
46         data_delayed_w[i+1] <= 0;
47         propag_bit_w[i+1] <= 1'b0;
48       end
49     end
50   end : REGISTER_LATENCE_READ
51 endgenerate
52
53 //Final propagation register for the simulated latency, affects the outputs
54 always_ff @(posedge clk_i) begin
55   if (read_enable_i && propag_bit_w[CYCLE_LATENCE-1]) begin
56     data_o <= data_delayed_w[CYCLE_LATENCE-1];
57     read_valid_o <= 1'b1;
58   end
59   else begin
60     data_o <= 0;
61     read_valid_o <= 1'b0;
62   end
63 end

```

Code 3.11 – Mémorisation et ajout des cycles de latence en écriture/lecture

Il est maintenant nécessaire de tester le bon fonctionnement de cette mémoire, et notamment la communication avec la mémoire cache. Pour cela, on reprends la testbench précédente, dans laquelle on intègre simplement la mémoire que nous venons de créer, on y implémente directement les données que l'on envoyait précédemment à la main. On vient donc tester exactement les mêmes interactions entre mémoire cache et principale que précédemment.

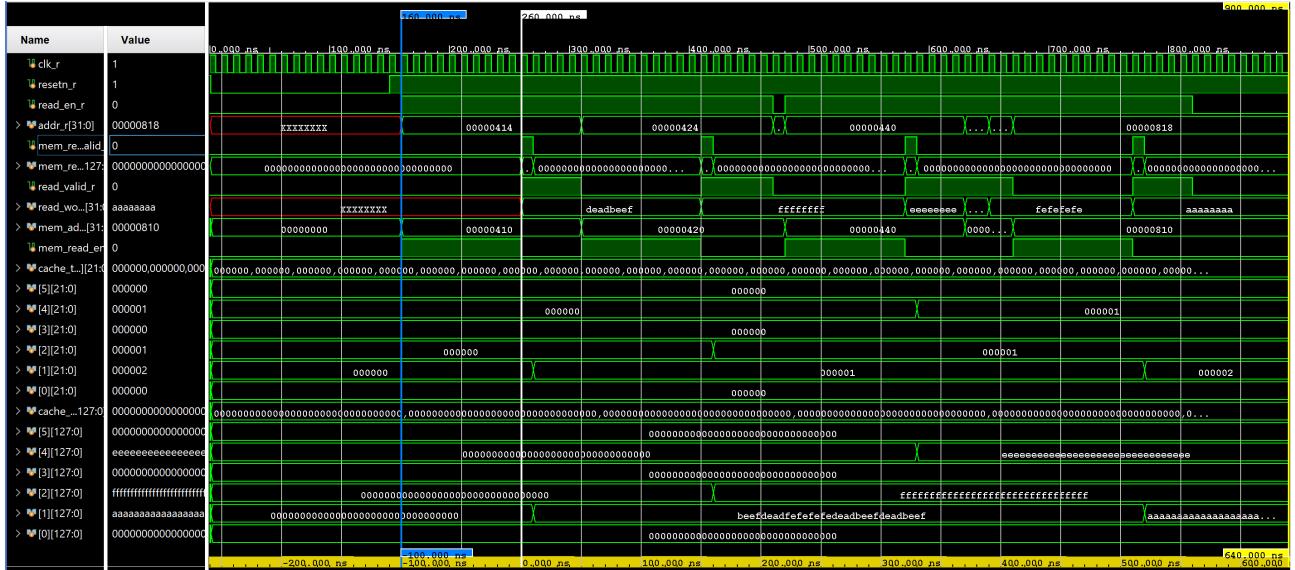


FIGURE 3.2 – Chronogramme de la testbench du "direct_cache" avec la mémoire principale

La testbench s'exécute correctement, et on observe bien les caractéristiques attendue. En effet, la mémoire met 10 cycles à envoyer la ligne de données à la mémoire cache, la donnée est bien directement disponible pour le processeur et est écrite sur la ligne de la cache au cycle suivant. Soit au bout d'un total de 11 cycles.

Les données sont bien cohérentes avec celles attendues également, et les signaux se génèrent correctement. On peut donc passer à l'étape suivante, l'implémentation avec le processeur.

3.2.3 Adaptation du processeur à la mémoire cache

3.2.4 Les modifications apportées

On intègre maintenant la cache au processeur, pour cela, plusieurs modules nécessitent une modification.

3.2.4.1 Modifications des control et data path

On considère maintenant des mémoires "non idéales", du moins pour les mémoires d'instructions pour l'instant. Ceci entraîne une modification radicale pour l'exécution des instructions dans notre processeur. En effet, précédemment, l'on considérait que les instructions étaient toujours prête instantanément lorsque l'adresse **PC** changeait. Ce n'est désormais plus le cas, en effet, dorénavant les instructions peuvent mettre jusqu'à 10 cycles d'horloge pour arriver.

Si l'on gardait le même fonctionnement dans les circuits **control_path** et **data_path**, la valeur du compteur PC continuerait de s'incrémenter alors que les instructions ne sont pas présentes sur le bus. Ceci entraînera donc 2 types de dysfonctionnements :

- Des instructions ne seront jamais exécutées, ou dans le désordre car présente seulement après leur appel.
- Des instructions seront exécutées en boucle, plusieurs fois, car l'instruction présente sur l'entrée **imem_data_i** n'est pas modifiée tant que la bonne instruction n'est pas valide

Pour résoudre ce problème, on ajoute une entrée logique supplémentaire aux 2 modules : **imem_read_valid_i**.

Cette entrée permet d'autoriser ou non l'écriture dans les registres qui séparent les différents étages de notre architecture, tant qu'elle est à l'état actif, les instructions se propagent dans

les différents étages. Si cette entrée est à l'état bas, tout les registres sont donc bloqués, et les instructions ne se propagent plus. Le processeur est en quelque sorte "bloqué", en attendant que l'instruction suivante soit disponible, réglant ainsi les problèmes.

3.2.4.2 Modification du RISCV_core

La modification de ce module est assez simple, on ajoute simplement l'entrée logique dans la déclaration des **control_path** et **data_path**. On ajoute également cette même entrée à ce module, afin de propager l'information.

Contrairement à une architecture multi-coeur, la mémoire cache n'est ici pas présente directement avec le coeur, mais dans le module global, situé au niveau d'abstraction supérieur.

3.2.4.3 Modification du top layer

Ici, on modifie le top layer afin de pouvoir utiliser la mémoire cache d'instruction. Pour celà, on retire l'instanciation de la mémoire d'instruction pour la remplacer par notre mémoire cache d'instructions.

On modifie également les entrées sorties, afin de pouvoir accepter des lignes de données en provenance de la mémoire principale et les signaux de contrôle de cette dernière. On prévoit les entrées et sorties liées à la cache de données dans le même temps.

```

1 module logic_unit_pipeline #(
2   parameter BYTE_OFF_BITS = 5,
3   parameter INDEX_BITS = 5,
4   parameter TAG_BITS = 22,
5   parameter NB_WAYS = 2,
6   localparam NB_WORDS_LINE = (2**BYTE_OFF_BITS)/4,
7   localparam NB_LINES = 2**INDEX_BITS,
8   localparam LINE_SIZE = 32 * NB_WORDS_LINE,
9   //Size of the L1 caches in Bytes
10  localparam L1_SIZE = NB_LINES * LINE_SIZE /8,
11  //Parameters for the instruction cache adress size and adresssing
12  localparam IMEM_BASE_ADDR = 32'h0000_0000,
13  localparam IMEM_SIZE = L1_SIZE,
14  //Parameters for the data cache adress size and adresssing
15  localparam DMEM_BASE_ADDR = 32'h0001_0000,
16  localparam DMEM_SIZE = L1_SIZE
17 )()
18   //General purpose input ports
19   input logic clk_i, rst_i,
20   //Input ports
21   input logic mem_instr_read_valid_i,
22   input logic mem_data_read_valid_i,
23   input logic mem_data_write_valid_i,
24   input logic [LINE_SIZE-1:0] mem_instr_read_data_i,
25   input logic [LINE_SIZE-1:0] mem_data_read_data_i,
26   //Output ports
27   output logic mem_instr_read_enable_o,
28   output logic mem_data_read_enable_o,
29   output logic mem_data_write_enable_o,
30   output logic [31:0] mem_instr_addr_o,
31   output logic [31:0] mem_data_addr_o,
32   output logic [LINE_SIZE-1:0] mem_data_write_data_o
33 );

```

Code 3.12 – Entrées sorties du coeur avec la cache

3.2.4.4 Modification de l'instanciation dans la testbench

Enfin, on instancie dorénavant le processeur RISCV, mais également le module de mémoire principale dans la testbench, car il est nécessaire de l'avoir pour que ce dernier s'exécute correctement.

3.2.5 Test et analyses des performances

Afin de vérifier le bon fonctionnement de notre cache, nous allons exécuter le programme "mult.S" du TP précédent. Cependant, afin de pouvoir réellement analyser les performances de notre cache, nous devons au préalable tester l'exécution du programme sans cette cache, mais avec la pénalité de performance de la mémoire principale.

On a une autre solution plus simple, on regarde simplement le temps d'exécution avec une mémoire "parfaite" et on multiplie le temps d'exécution par 10, pour prendre en compte les 10 cycles de latence de cette dernière.

On exécute donc ce programme et on obtient le chronogramme suivant :

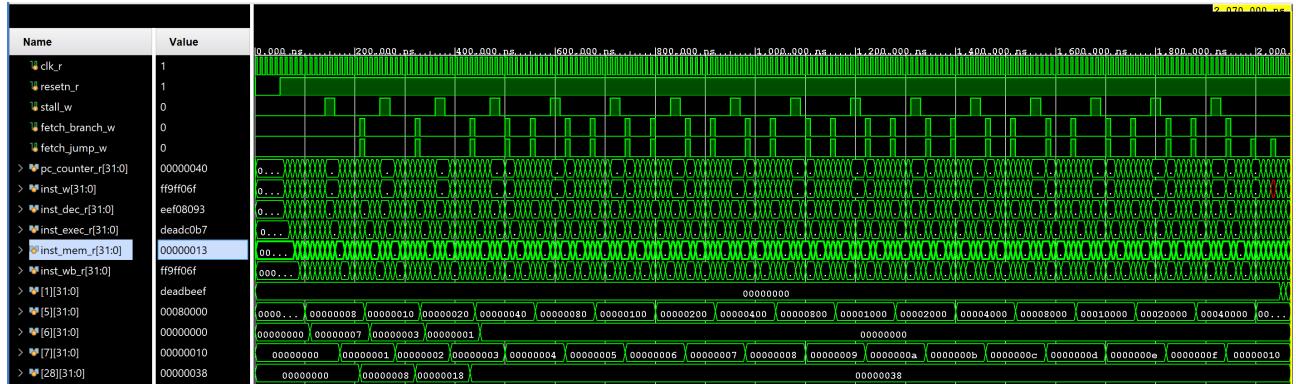


FIGURE 3.3 – Chronogramme de l'exécution du programme "mult.S" sans cache et avec une mémoire "parfaite"

On obtient un temps d'exécution total de :

$$\begin{aligned} \text{Temps d'exécution "idéal"} &= 2070\text{ns} \\ \text{Temps d'exécution "réel"} &= 20700\text{ns} \end{aligned}$$

Que l'on compare donc avec la même exécution, mais avec notre cache d'instructions.

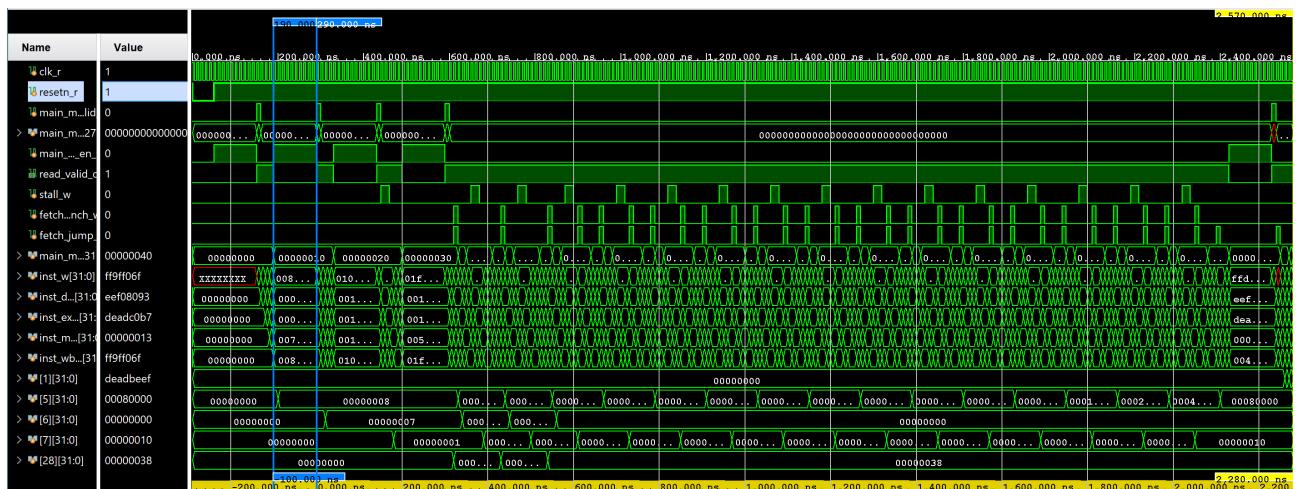


FIGURE 3.4 – Chronogramme de l'exécution du programme "mult.S" avec 4 mots par ligne de cache

On obtient un temps d'exécution total de :

$$\text{Temps d'exécution} = 2570\text{ns}$$

On peut mettre en évidence le bon fonctionnement de la cache en zoomant sur le début de l'exécution du programme. Soit le chargement initial des valeurs et la première boucle. On voit ainsi le chargement des lignes d'instructions dans la mémoire cache, puis l'exécution des instructions directement depuis la mémoire cache, et donc sans lecture supplémentaires vers la mémoire d'instructions.

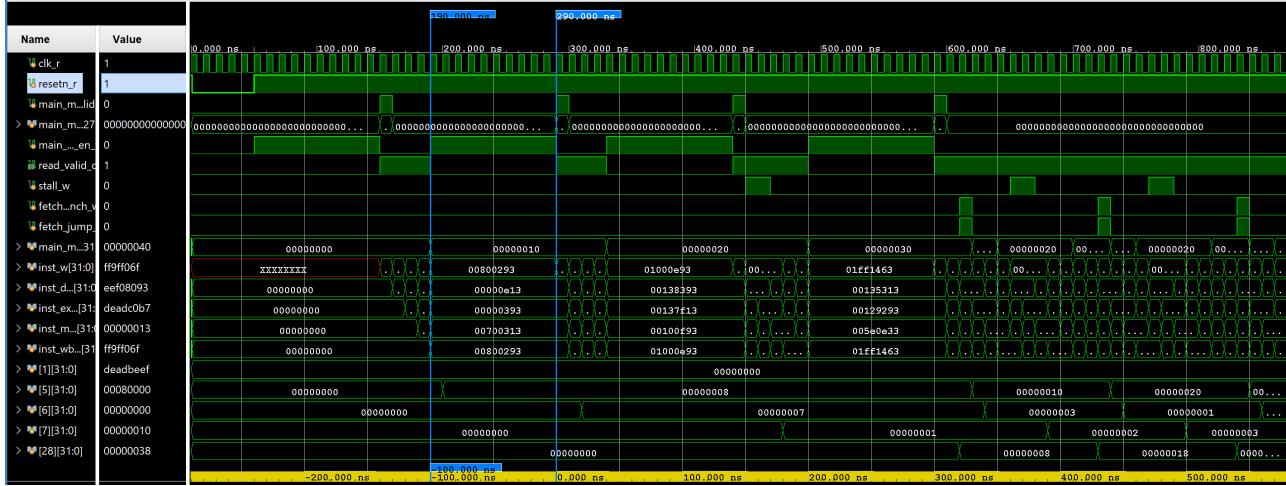


FIGURE 3.5 – Zoom sur le chronogramme de l'exécution du programme "mult.S" avec 4 mots par ligne de cache

Afin d'améliorer les performances de la cache, il peut être intéressant de réduire le nombre de lignes, mais d'augmenter la taille de ces dernières. Celà possède l'avantage de réduire le nombre de requêtes mémoires dans le cas de petites boucles de lectures d'instructions, néanmoins, on réduit la "diversité" des lignes que l'on peut stocker (lignes provenant d'adresses mémoires éloignées, donc avec des tags différents). La taille des lignes doit donc être choisie en gardant celà en tête, pour ne pas réduire les performances dans une grande partie des cas, pour les améliorer dans de rares cas.

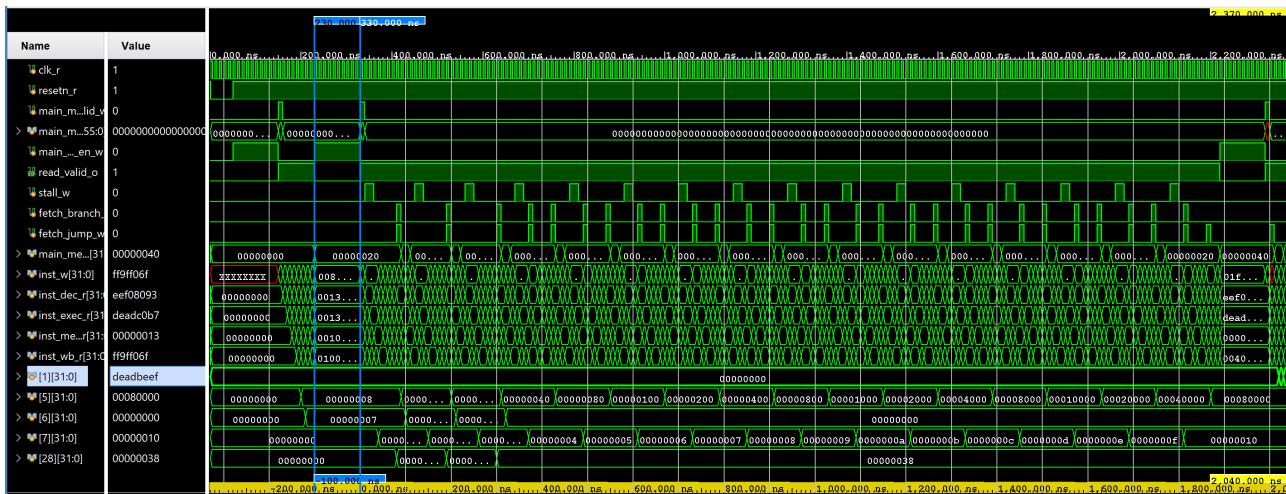


FIGURE 3.6 – Chronogramme de l'exécution du programme "mult.S" avec 8 mots par ligne de cache

On obtient un temps d'exécution total de :

$$\text{Temps d'exécution} = 2370\text{ns}$$

Ce qui correspond à un gain de : 200ns , soit 2 appels mémoires "miss".

Un zoom sur le début de l'exécution nous montre le gain de performances que l'on retrouve ici.

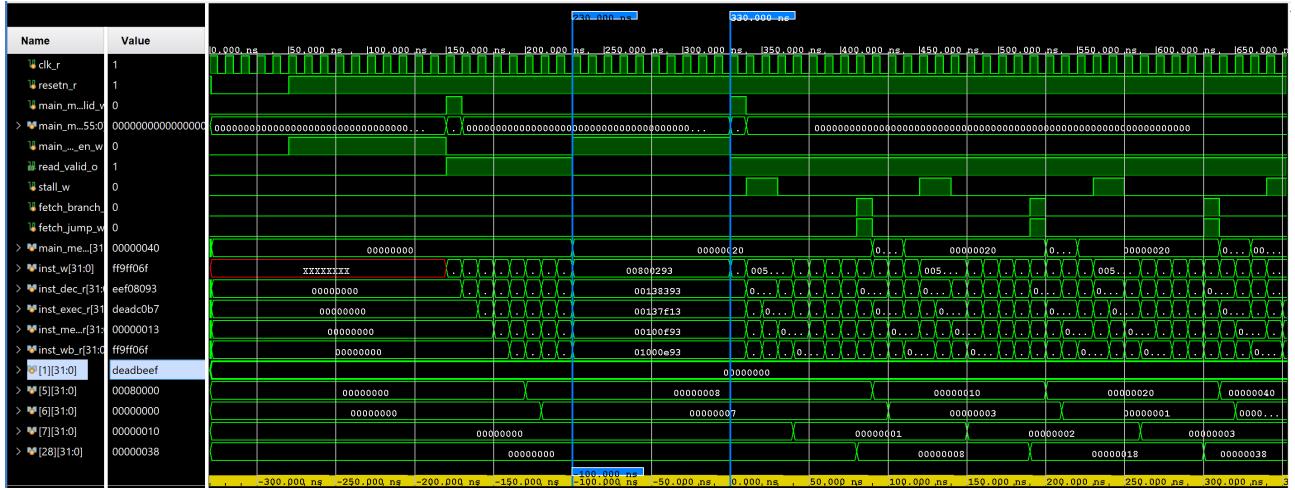


FIGURE 3.7 – Zoom sur le chronogramme de l'exécution du programme "mult.S" avec 8 mots par ligne de cache

3.3 Cache d'instructions associatif à plusieurs voies

3.3.1 Crédation du module multi_way_cache

Afin d'améliorer les performances dans le cas où plusieurs instructions sont exécutées régulièrement, et reposent sur la même ligne, mais à des adresses tags différentes, on propose l'implémentation de plusieurs voies dans notre cache. Celà se traduit donc par une amélioration de notre cache précédente, en augmentant sa capacité, sans pour autant augmenter le temps d'accès à cette dernière, car on augmente pas le nombre de lignes (ce qui pourrait également améliorer les performances si on était pas limité par la taille, le coût et le temps d'accès à des mémoires caches plus larges).

On se propose ici de ne pas se limiter à 2 voies, mais d'implémenter une cache multi-voies, où le nombre de cette dernière est paramétrable et modifiable à la volée, pour tester les améliorations de performances et l'adaptée à notre grée à notre application.

On utilisera ici un algorithme **LRU** pour le remplacement des lignes dans notre cache. Ce sera donc la ligne qui a été lue/écrite il y a le plus longtemps qui sera "écrasée" lors d'une demande d'écriture d'une nouvelle ligne dans la mémoire.

On crée donc un nouveau module, nommée **multi_way_cache.sv**, dans lequel on apporte les modifications suivantes afin de l'adapter.

Tout d'abord, on ajoute un paramètre pour connaitre le nombre de voies à implémenter, que l'on nommera pour la suite **NB_WAYS**.

Ensuite, il est nécessaire dans un premier temps de modifier les signaux qui servent à stocker les valeurs, afin d'ajouter une nouvelle dimension, pour les voies. On ajoute également le stockage de la valeur LRU pour chaque ligne, qui permet de connaitre l'index de la prochaine ligne dans laquelle écrire la donnée (la plus ancienne, en terme d'accès).

```

1 //Cache elements
2 logic cache_line_validity[NrLines-1:0][NB_WAYS-1:0];
3 logic [TagBits-1:0] cache_tags[NrLines-1:0][NB_WAYS-1:0];
4 logic [LineSize-1:0] cache_words[NrLines-1:0][NB_WAYS-1:0];
5 logic [NB_WAYS*BITS_WAYS-1:0] cache_LRU[NrLines-1:0];           //Signal used in order to
   choose which way to write the new line into

```

Code 3.13 – Registres mis à jours avec la LRU

La valeur **BITS_WAYS** correspond au nombre de bits nécessaire pour stocker les adresses des voies, et dépend donc du nombre de voies.

On modifie maintenant l'instanciation des registres, en rajoutant à la fois le registre de stockage des valeurs LRU, la lecture dans les différentes voies et la logique de mise à jour des valeurs LRU suivant l'écriture ou la lecture dans la cache.

```

1 //Registers implementation
2 generate
3     for (genvar i=0;i<NrLines;i++) begin : registers_cache
4         for (genvar j=0;j<NB_WAYS;j++) begin : registers_cache_ways
5             always_ff @(posedge clk_i or negedge rstn_i) begin : write_cache
6                 if(rstn_i==1'b0) begin
7                     cache_line_validity[i][j] <= 1'b0;
8                     cache_tags[i][j] <= 0;
9                     cache_words[i][j] <= 0;
10                end
11                else if (mem_read_valid_i && (read_index==i) && (cache_LRU[i][0+:BITS_WAYS
12 ]==j)) begin
13                    cache_line_validity[i][j] <= 1'b1;
14                    cache_tags[i][j] <= read_tag;
15                    cache_words[i][j] <= mem_read_data_i;
16                    cache_LRU[i] <= {cache_LRU[i][0+:BITS_WAYS],cache_LRU[i][BITS_WAYS+:(
17 NB_WAYS-1)*BITS_WAYS]};
18                    end
19                    else if ((hit_w!=0) && (cache_LRU[i][(j*BITS_WAYS)+:BITS_WAYS]==
20 hit_index_w) && read_en_i && (i==read_index)) begin
21                        //Ajout modification LRU suite a une lecture
22                        if (j==0) cache_LRU[i] <= {cache_LRU[i][0+:BITS_WAYS],cache_LRU[i][
23 BITS_WAYS+:(NB_WAYS-1)*BITS_WAYS]};
24                        else if (j<(NB_WAYS-1)) cache_LRU[i] <= {cache_LRU[i][(j*BITS_WAYS)+:
25 BITS_WAYS],cache_LRU[i][((j+1)*BITS_WAYS)+:(NB_WAYS-j-1)*BITS_WAYS],cache_LRU[i][0+j*
26 BITS_WAYS]};
27                        end
28                    end : write_cache
29                    always_comb begin : read_cache
30                        if (hit_w[j]) begin
31                            registers_output_lines_tags[i] <= cache_tags[i][j];
32                            registers_output_all[i] <= cache_words[i][j];
33                        end
34                    end : read_cache
35                end : registers_cache_ways
36                always_ff @(posedge clk_i or negedge rstn_i) begin : LRU_cache
37                    if(rstn_i==1'b0) begin
38                        cache_LRU[i] <= int_lru();
39                    end
40                end : LRU_cache
41            end : registers_cache
42        endgenerate

```

Code 3.14 – Conditions d'écritures (synchrone) et de lecture (asynchrone) dans les registres, avec la LRU

La logique de mise à jour de la valeur LRU est la suivante :

— Format des valeurs pour une ligne :

- Un mot de $NB_WAYS * BITS_WAYS$, qui contient les indices des différentes voies (de 0 à NB_WAYS-1)

- Par exemple, pour une cache à 4 voies, pour une ligne "i" donnée la valeur LRU initiale sera : [3210] = [11100100] (en binaire)

- La ligne la plus ancienne sera donc celle dont l'indice se place aux bits de poids faibles, la plus récente au niveau de bits de poids forts (donc dans notre exemple, la voie 0 est la plus ancienne, la voie 3 est la voie la plus récente)

- **Mise à jour en cas d'écriture** : Une écriture depuis la mémoire se déroule toujours sur la voie dont l'indice se trouve aux bits de poids faibles de la valeur LRU. On place donc cette valeur aux bits de poids forts et on décale de $BITS_WAYS$ bits vers la droite. Avec notre exemple, cela donne alors : [0321]

- **Mise à jour en cas de lecture** : En cas de lecture, l'opération est plus complexe. En effet, il est nécessaire de déterminer dans quelle voie la lecture se passe, et quelle est la

position de cette voie dans la LRU. Une fois cette position déterminée, on place cet indice au début de la LRU (poids fort), et on décale vers la droite tout les indices qui occupaient les bits de poids supérieurs à la position initiale de la cache. Ce qui correspond dans notre exemple, en cas de lecture sur la voie 2 : [2031]

Un des changements supplémentaire à apporter est la diversification du signal **hit_w**. On doit en effet générer un signal par voie, pour déterminer dans quelle voie la donnée correspond à celle qui a été demandée par le processeur. On génère également un nouveau signal **hit_index_w**, qui contient, comme son nom l'indique, l'index de la voie dans laquelle la donnée se trouve, il est utile pour mettre à jour la LRU en cas de lecture.

```

1 //Generation of the hit signal, one per way
2 generate
3     for (genvar h_1=0;h_1<NB_WAYS;h_1++) begin : hit_gen
4         always_comb begin : hit_or_miss
5             if (cache_tags[read_index][h_1]==read_tag && read_en_i && cache_line_validity[read_index][h_1]) begin
6                 hit_w[h_1] = 1'b1;
7                 hit_index_w = h_1;
8             end
9             else hit_w[h_1] = 1'b0;
10        end : hit_or_miss
11    end : hit_gen
12 endgenerate

```

Code 3.15 – Génération du nouveau signal de **hit**

Les autres parties du codes ne nécessitent pas de modifications substantielles, on ne s'attardera donc pas dessus ici.

3.3.2 Vérification et analyse des performances

On va maintenant vérifier que notre cache fonctionne de la manière souhaitée. Dans un premier temps, on reprends simplement la testbench utilisée pour tester la cache directe, car on souhaite vérifier que notre cache continue d'avoir son fonctionnement normal. De plus, cette testbench écrit 2 fois dans la ligne n°1 de la cache, ce qui va donc nous permettre de vérifier que la LRU fonctionne correctement.

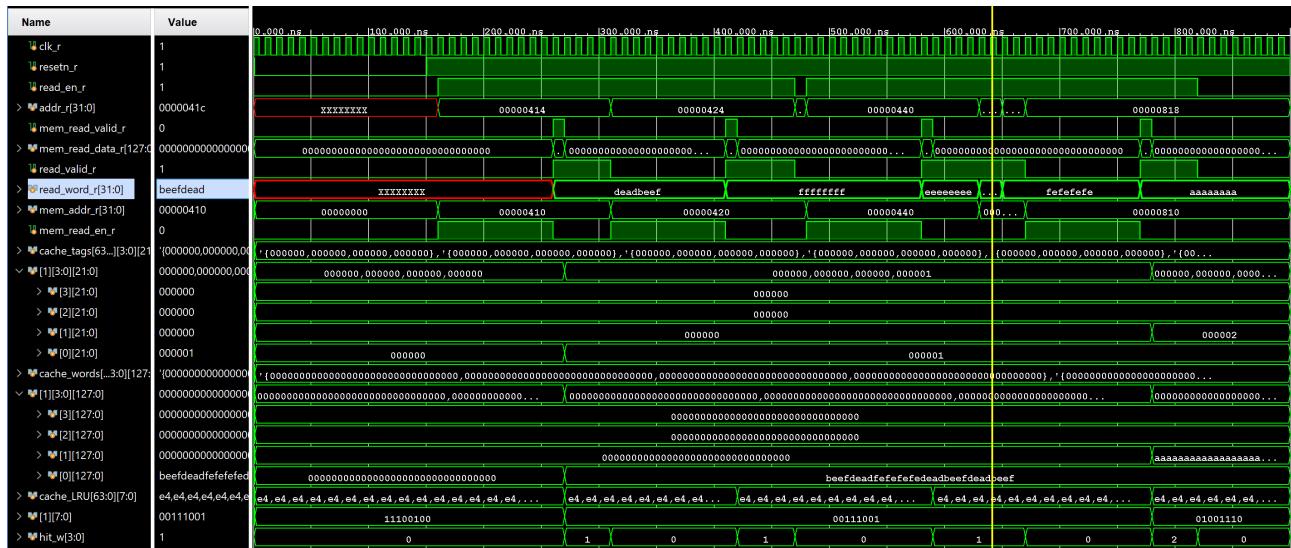


FIGURE 3.8 – Chronogramme de la testbench du "multi_way_cache" avec la mémoire principale

La donnée présente sur la ligne 1 et voie 0 n'est pas supprimée par la nouvelle donnée, qui se place correctement sur la voie 1. On a donc vérifié le bon fonctionnement de la cache, mais

on va dorénavant vérifier que la valeur de la LRU s'actualise bien.

Pour celà, on crée une testbench qui va venir charger tour à tour des lignes de données dans la cache, toujours sur la ligne n°1, afin de stresser l'implémentation multi-voie. On va également effectuer une opération de lecture entre les écritures, pour vérifier que la valeur de la LRU s'update correctement dans ce cas-ci.

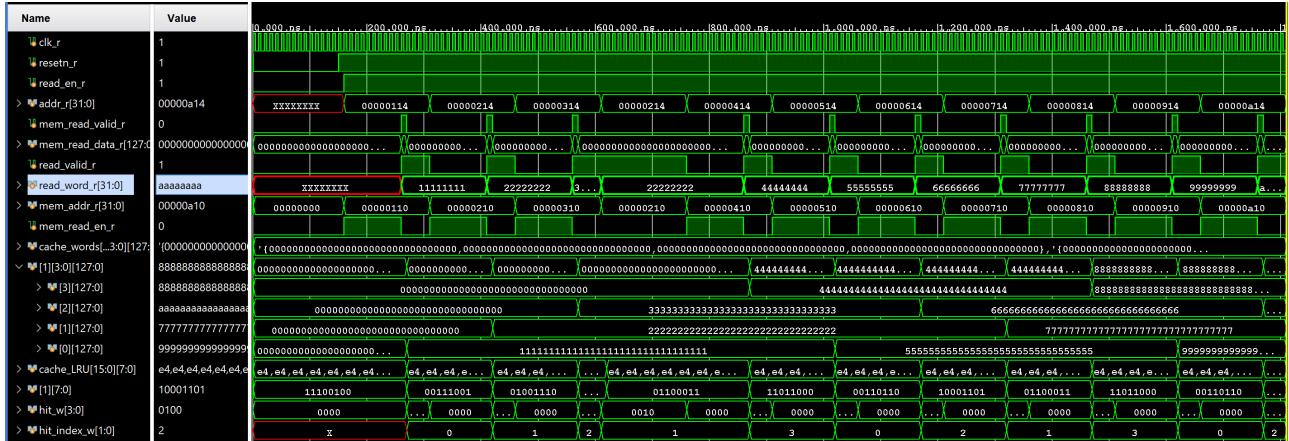


FIGURE 3.9 – Chronogramme de la testbench spécifique au "multi_way_cache" avec la mémoire principale

On observe bien le résultat attendu, en effet, on écrit dans les 3 premières voies, puis on effectue une lecture sur la voie 1, l'écriture suivante se fait bien sur la voie 3 comme attendue, puis sur la voie 0, car c'est bien la plus ancienne. Mais, l'écriture suivante s'effectue sur la voie 2, on voie alors bien que la voie 1 a été conservée, car elle est plus "récente" que la voie 2. On obtient donc, comme attendu, l'évolution suivante pour la LRU de la ligne 1 :

- Initialisation : [3210]
- Ecriture : [0321]
- Ecriture : [1032]
- Ecriture : [2103]
- Lecture : [1203]
- Ecriture : [3120]
- Ecriture : [0312]
- Ecriture : [2031]
- Ecriture : [1203]
- Ecriture : [3120]
- Ecriture : [0312]
- ...

Dorénavant, nous allons tester l'implémentation de la cache sur le processeur pour voir le gain de performance que l'on peut gagner. On commence par executer le programme "mult.S", pour voir si ce dernier bénéficie de la différence d'architecture offerte par notre cache améliorée.

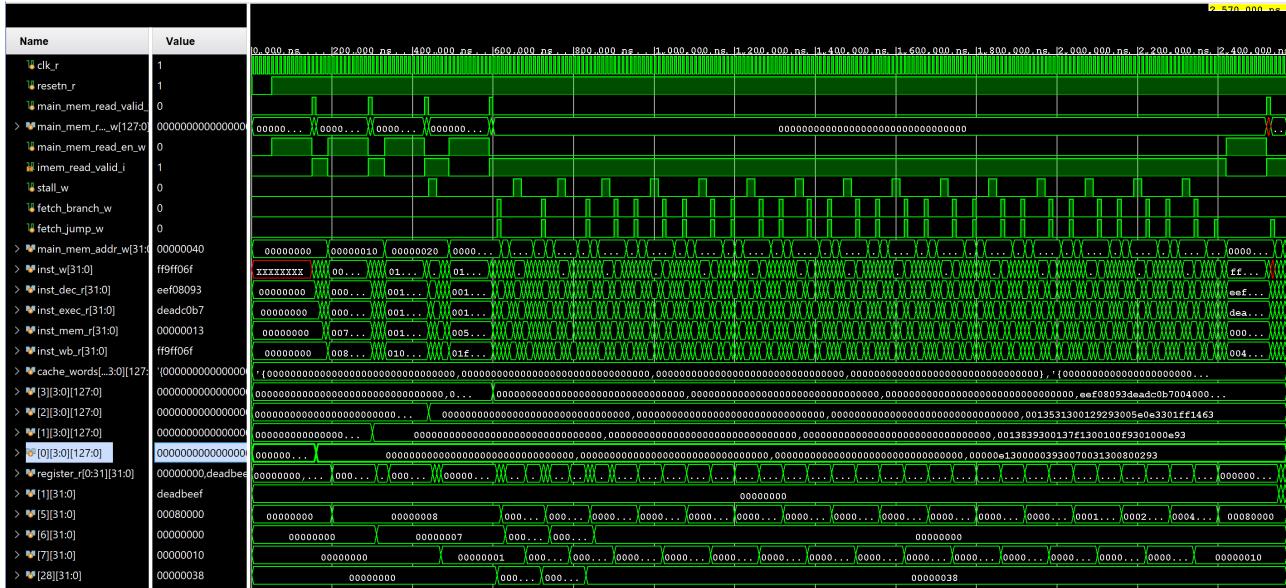


FIGURE 3.10 – Chronogramme de l'exécution du programme "mult.S" avec la cache multi-voies

Le temps d'exécution est exactement le même que pour une cache identique, sans gestion des voies. Ce programme ne posait en effet pas de problème avec la cache précédente, on va donc chercher à mettre en évidence les avantages de cette cache en écrivant un programme qui va "mettre à mal" la cache directe.

Pour cela, on crée un programme qui va effectuer une suite d'opérations (n'importe lesquelles tant qu'elles ne font pas appel à la mémoire), en boucle. Le nombre d'opération devra être suffisamment élevée pour que la taille de la cache d'instruction ne suffise pas à garder l'entièreté de la boucle en mémoire, et qu'il y ait besoin de réaliser au minimum un appel mémoire par boucle afin de la compléter.

On écrit donc le programme assembleur suivant :

Programme en assembleur de **multiway_stresstest.S**

```
.section .start;
.globl start;

start :
    li t0, 0          //Compteur de boucle
    li t1, 0x1f        //Valeur de fin de boucle
    li t2, 0xABCD      //Operande n1 pour les calculs
    li t3, 0x52        ///Operande n2 pour les calculs

loop :
    addi t0,t0,1      //Incrementation du compteur de boucle
    or t4,t2,t3        //Opération de remplissage de la boucle
    and t5,t2,t3
    add t6,t2,t3
    sub t4,t2,t3
    xor t5,t2,t3
    sll t6,t2,t3
    srl t4,t2,t3
    sra t5,t2,t3
```

```

slt t6,t2,t3
sltu t4,t2,t3
bltu t0, t1, loop //Comparateur de sortie de la boucle
lab1 :
    li ra, 0xDEADBEEF
    j lab1

.end start

```

Que l'on execute tout d'abord avec une petite cache directe, de seulement 4 lignes composée de mots de 2 mots par lignes.

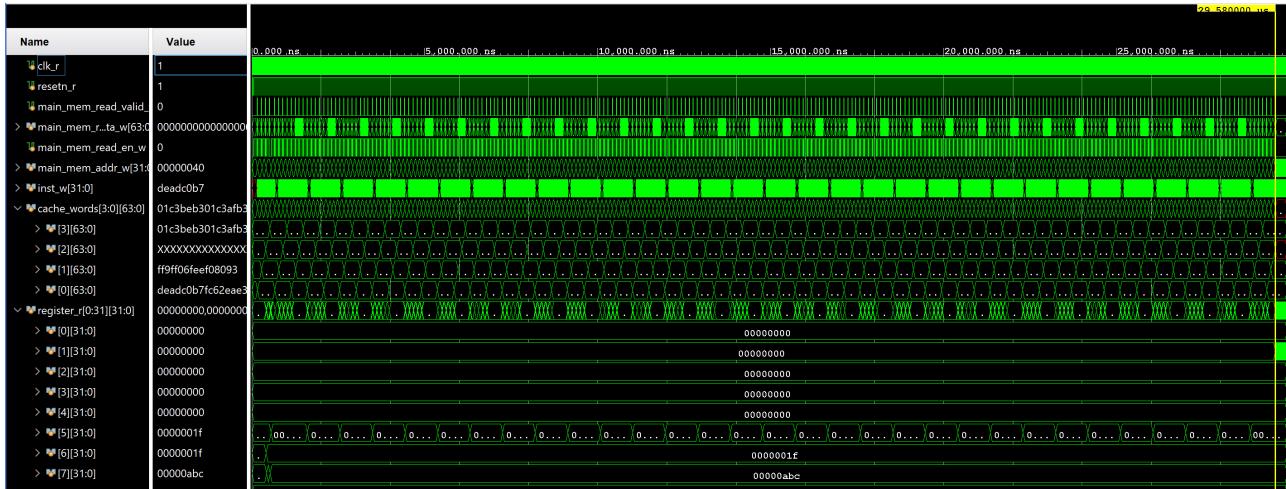


FIGURE 3.11 – Chronogramme de l'exécution du programme "multiway_stresstest.S" avec la cache directe

Le temps d'exécution est extremement long ici, plus de 20000ns. En zoomant sur l'exécution d'une des boucles du programme, on peut aisément voir pourquoi.

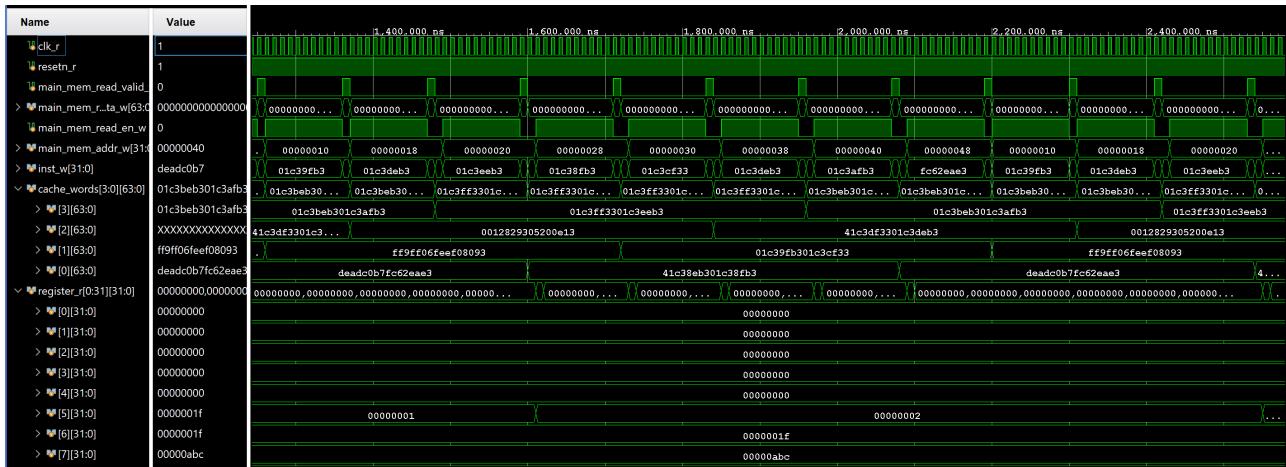


FIGURE 3.12 – Zoom sur une des boucles du programme "multiway_stresstest.S" avec la cache directe

La cache ne peut pas suivre, et doit sans cesse appeler la mémoire principale pour récupérer les données qu'elle a précédemment effacé. On perd donc énormément de cycles pour rien.

On vient donc remplacer cette cache par notre cache multivoies, qu'on configure avec le même nombre de lignes et de mots, mais on utilise ici 2 voies.

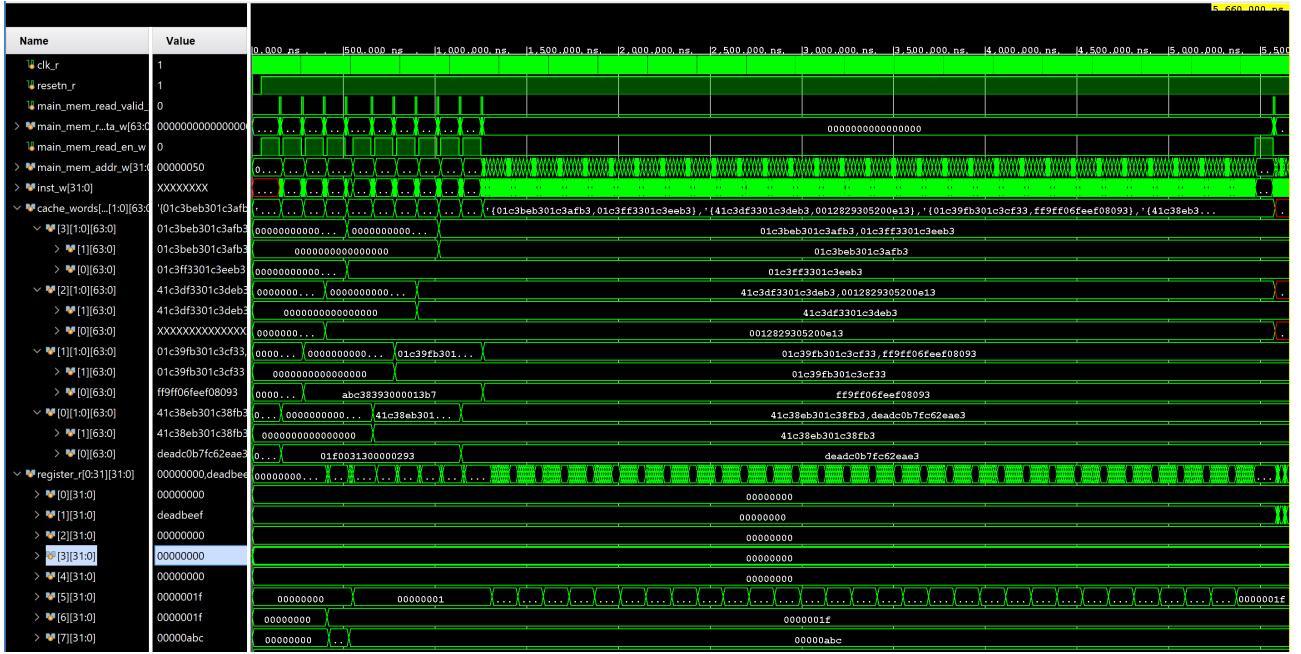


FIGURE 3.13 – Chronogramme de l'exécution du programme "multiway_stresstest.S" avec la cache multivoies

Le temps d'exécution est ici tout à fait réaliste, en effet, on se retrouve avec un temps total de 5660ns. On peut voir que la mémoire n'est appelée que pour charger les instructions lors de la première boucle, puis, lors des boucles suivantes, celle-ci n'est plus sollicitée car les instructions sont toutes présentes dans la cache. On ne perd donc plus de cycle inutilement à réécrire en permanence les mêmes instructions dans la cache.

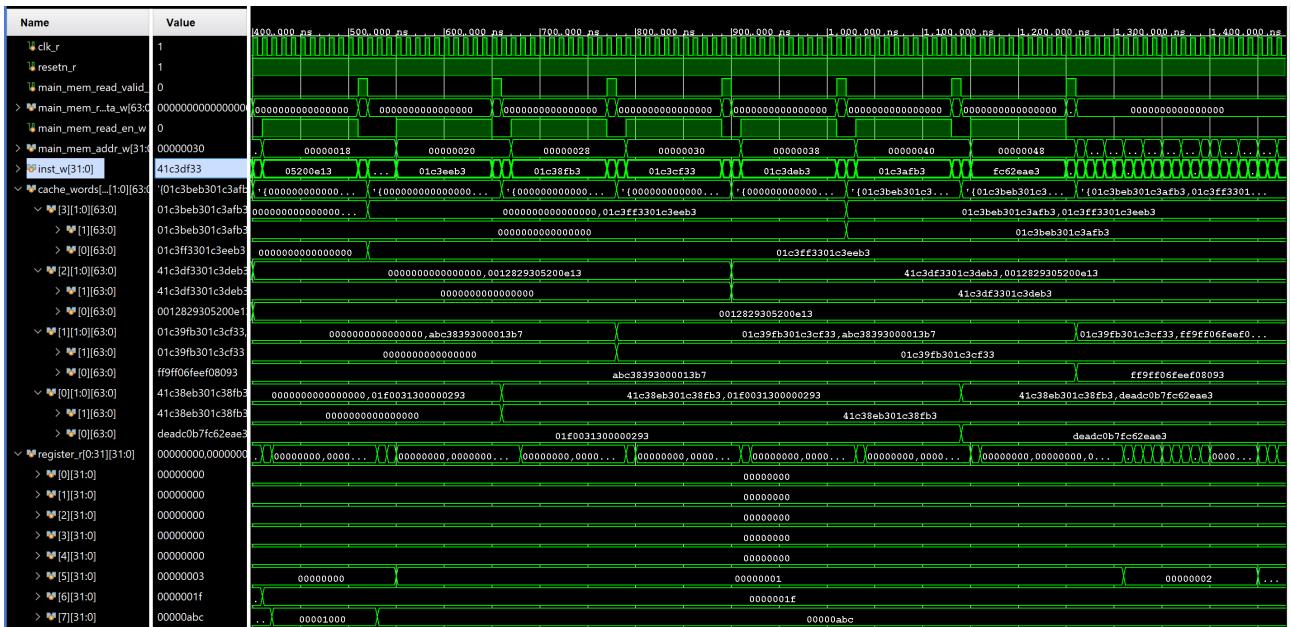


FIGURE 3.14 – Zoom sur une des boucles du programme "multiway_stresstest.S" avec la cache multivoies

3.4 Cache de données avec écriture "write through"

3.4.1 Crédation du module "write_through_cache"

On s'intéresse maintenant à l'ajout de la possibilité d'écriture dans notre cache, afin de pouvoir l'utiliser pour la mémoire de donnée (donc dans l'étage **MEM**). Pour celà, on s'intéresse dans un premier temps à un algorithme de type "write through". Cet algorithme impose, dans le cas d'une écriture dans la cache, de devoir, avant de valider l'écriture au processeur, transmettre la ligne modifiée à la mémoire supérieure afin d'assurer la cohérence avec cette dernière.

On commence donc par récupérer le fichier de notre cache multivoies, **multi_way_cache.sv**, duquel on va partir pour lui ajouter l'écriture, on l'appellera alors **write_through_cache.sv**.

On commence donc par décommenter les entrées et sorties en rapport avec l'écriture :

```
1 module write_through_cache #(
2     parameter ByteOffsetBits = 5,
3     parameter IndexBits = 5,
4     parameter TagBits = 22,
5     parameter NB_WAYS = 2,
6     localparam BITS_WAYS = $clog2(NB_WAYS),
7     localparam WORD_SIZE = 32,
8     localparam NrWordsPerLine = (2**ByteOffsetBits)/4,
9     localparam NrLines = 2**IndexBits,
10    localparam LineSize = WORD_SIZE * NrWordsPerLine
11 ) (
12     // General purpose input ports
13     input logic clk_i,
14     input logic rstn_i,
15     input logic [31:0] addr_i,
16
17     // Read port
18     input logic read_en_i,
19     output logic read_valid_o,
20     output logic [WORD_SIZE-1:0] read_word_o,
21
22     // Write port
23     input logic write_en_i,
24     input logic [WORD_SIZE-1:0] write_word_i,
25     output logic write_valid_o,
26
27     // Memory
28     output logic [31:0] mem_addr_o,
29
30     // Memory read port
31     output logic mem_read_en_o,
32     input logic mem_read_valid_i,
33     input logic [LineSize-1:0] mem_read_data_i,
34
35     // Memory write port
36     output logic mem_write_en_o,
37     output logic [LineSize-1:0] mem_write_data_o,
38     input logic mem_write_valid_i
39 );
```

Code 3.16 – Déclaration du nouveau module

Ensuite, pour la génération des signaux de **hit_w** et le découpage de l'adresse, on remplace la condition **read_enable_i** par **read_enable_i || write_enable_i**.

On s'intéresse maintenant au corps du problème pour l'autorisation de l'écriture. On commence donc par gérer le cas du **miss** en écriture. On viens chercher la ligne depuis la mémoire comme pour une lecture classique. Ensuite, on arrive dans le cas d'un **hit** en écriture. On a donc 2 actions à mener en simultanée, écrire la donnée dans la cache, et la transmettre à la mémoire supérieure.

Dans ce bloc, on affecte donc la ligne à envoyer à la mémoire à partir du mot à écrire et de la ligne qui est présente dans la cache :

```

1 //Memory write data output organizer
2 generate
3   for (genvar k=0;k<NrWordsPerLine;k++) begin : memory_write_data_distribution
4     always_comb begin
5       if (k==read_offset) mem_write_data_s[(k*WORD_SIZE)+:WORD_SIZE] = write_word_i;
6       else mem_write_data_s[(k*WORD_SIZE)+:WORD_SIZE] = registers_output_all[
7         read_index][(k*WORD_SIZE)+:WORD_SIZE];
8     end
9   end : memory_write_data_distribution
endgenerate

```

Code 3.17 – Assignation de la ligne à écrire en mémoire

Et, on l'écrit dans la mémoire de la cache en modifiant le bloc **always_ff** de la cache précédente :

```

1 //
2 //-----//  

3 //"Memory" of the module  

4 //-----//  

5 generate
6   for (genvar i=0;i<NrLines;i++) begin : registers_cache
7     for (genvar j=0;j<NB_WAYS;j++) begin : registers_cache_ways
8       always_ff @ (posedge clk_i or negedge rstn_i) begin : write_cache
9         ///////////////////////////////////////////////////////////////////Reset
10        if (rstn_i==1'b0) begin
11          cache_line_validity[i][j] <= 1'b0;
12          cache_tags[i][j] <= 0;
13          cache_words[i][j] <= 0;
14        end
15        ///////////////////////////////////////////////////////////////////Update of a way of the cache, with a data coming from a higher level
16        memory
17        else if (mem_read_valid_i && (read_index==i) && (cache_LRU[i][0+:BITS_WAYS]
18 ==j)) begin
19          cache_line_validity[i][j] <= 1'b1;
20          cache_tags[i][j] <= read_tag;
21          cache_words[i][j] <= mem_read_data_i;
22          cache_LRU[i] <= {cache_LRU[i][0+:BITS_WAYS],cache_LRU[i][BITS_WAYS+:(
23 NB_WAYS-1)*BITS_WAYS]};
24          end
25          ///////////////////////////////////////////////////////////////////Update a way of the cache according to the hit value, when it has been
26          transmitted to upper level memory successfully
27          //, with a data/word coming from the processor or a lower level memory
28          else if (hit_w[cache_LRU[i][(j*BITS_WAYS)+:BITS_WAYS]] && write_en_i && (i
29 ==read_index) && (waiting_write_finished_s==1'b0)) begin
30            cache_line_validity[i][cache_LRU[i][(j*BITS_WAYS)+:BITS_WAYS]] <= 1'b1
31            ;
32            cache_words[i][cache_LRU[i][(j*BITS_WAYS)+:BITS_WAYS]][(read_offset*
33 WORD_SIZE)+:WORD_SIZE] <= write_word_i;
34            if (j==0) cache_LRU[i] <= {cache_LRU[i][0+:BITS_WAYS],cache_LRU[i][
35 BITS_WAYS+:(NB_WAYS-1)*BITS_WAYS]};
36            else if (j<(NB_WAYS-1)) cache_LRU[i] <= {cache_LRU[i][(j*BITS_WAYS)+:
37 BITS_WAYS],cache_LRU[i][((j+1)*BITS_WAYS)+:(NB_WAYS-j-1)*BITS_WAYS],cache_LRU[i][0+:j*
38 BITS_WAYS]};
39            end
40            ///////////////////////////////////////////////////////////////////Modification of the LRU value after a succesfull read in a way
41            else if ((hit_w!=0) && (cache_LRU[i][(j*BITS_WAYS)+:BITS_WAYS]==
42 hit_index_w) && read_en_i && (i==read_index)) begin
43              if (j==0) cache_LRU[i] <= {cache_LRU[i][0+:BITS_WAYS],cache_LRU[i][
44 BITS_WAYS+:(NB_WAYS-1)*BITS_WAYS]};
45              else if (j<(NB_WAYS-1)) cache_LRU[i] <= {cache_LRU[i][(j*BITS_WAYS)+:
46 BITS_WAYS],cache_LRU[i][((j+1)*BITS_WAYS)+:(NB_WAYS-j-1)*BITS_WAYS],cache_LRU[i][0+:j*
47 BITS_WAYS]};
48              end
49            end : write_cache
50            ///////////////////////////////////////////////////////////////////Modification of the output based on the hit signal
51            always_comb begin : read_cache
52              if (hit_w[j]) begin
53                registers_output_lines_tags[i] <= cache_tags[i][j];
54              end
55            end
56          end : read_cache
57        end
58      end
59    end
60  end
61
```

```

39           registers_output_all[i] <= cache_words[i][j];
40       end
41   end : read_cache
42 end : registers_cache_ways
43 ///Initialisation of the LRU during a reset
44 always_ff @(posedge clk_i or negedge rstn_i) begin : LRU_cache
45     if(rstn_i==1'b0) begin
46         cache_LRU[i] <= int_lru();
47     end
48   end : LRU_cache
49 end : registers_cache
50 endgenerate

```

Code 3.18 – Modification des conditions d'écriture (synchrone) et de lecture (asynhrone) pour la cache write_through

Les modifications sont ici dans l'ajout de la 3ème condition du **if**, en cas d'écriture avec un **hit**, on vient écrire le mot dans la ligne, modifier la valeur de la LRU et passer le bit de validité à 1 si il ne l'avait pas déjà été.

L'ajout de la condition (**waiting_write_finished_s == 1'b0**) est ici une sécurité nécessaire pour éviter l'écriture dans une ligne qui n'est pas supposée être modifiée. En effet, la synchronisation de l'écriture lors d'un changement d'adresse/de mot à modifier n'est pas instantanée, et il est donc nécessaire de patienter 1 cycle complet pour que toutes les données soit à jour, et effectuer l'écriture au bon endroit. Cette variable permet donc celà, grâce à la bascule suivante :

```

1  // -----
2  //-----// "Post-Memory" logic of the module
3  //-----//
4  //Flip-flop checking if a writing operation is finished or if any new one is started
5  always_ff @(addr_i, write_word_i, posedge mem_write_valid_i, posedge mem_read_valid_i,
6    posedge write_en_i, posedge read_en_i) begin
7    if (mem_write_valid_i || (mem_read_valid_i && read_en_i)) waiting_write_finished_s =
8      1'b1;
9    else waiting_write_finished_s = 1'b0;
10 end

```

Code 3.19 – Bascule permettant de résoudre les problèmes d'écritures involontaires

Ce signal est donc mis à 1 dès qu'une écriture/lecture est terminée, et repasse à 0 uniquement lorsque l'on change de mot, d'adresse ou d'actions à effectuer dans la cache. Il permet donc de "resynchroniser" la mémoire à faible coup. On a alors pas d'écriture aux mauvais endroits.

Il est maintenant nécessaire d'adapter la génération des signaux logiques de contrôle vers la mémoire et le processeur. C'est ce bloc **always_comb** qui s'en occupe :

```

1  //Registers output logic and data assignation
2  always_comb begin : mux_read_out
3    ///////////////////////////////////////////////////////////////////
4    if (rstn_i==1'b0) begin
5      mem_read_en_o = 1'b0;
6      mem_write_en_o = 1'b0;
7      write_valid_o = 1'b0;
8    end
9    ///////////////////////////////////////////////////////////////////
10   else if (mem_read_valid_i && read_en_i) begin
11     registers_output_line = mem_read_data_i;
12     mem_read_en_o = 1'b0;
13     mem_write_en_o = 1'b0;
14     write_valid_o = 1'b0;
15   end
16   ///////////////////////////////////////////////////////////////////
17   else if ((waiting_write_finished_s==1'b0) && mem_read_valid_i && write_en_i) begin
18     mem_read_en_o = 1'b0;
19     mem_write_en_o = 1'b0;
20     write_valid_o = 1'b0;

```

```

21      end
22      ////In case of a line already present in cache, output of the correct word
23      else if (hit_w && read_en_i) begin
24          registers_output_line = registers_output_all[read_index];
25          mem_read_en_o = 1'b0;
26          write_valid_o = 1'b0;
27          mem_write_en_o = 1'b0;
28      end
29      ////Sending data to be written in upper level memory
30      else if ((waiting_write_finished_s==1'b0) && hit_w && write_en_i) begin
31          mem_write_data_o = mem_write_data_s;
32          mem_write_en_o = 1'b1;
33          mem_read_en_o = 1'b0;
34          write_valid_o = 1'b0;
35      end
36      ////In case of a miss, asking the upper level memory for the line
37      else if(read_en_i) begin
38          mem_read_en_o = 1'b1;
39          mem_write_en_o = 1'b0;
40          write_valid_o = 1'b0;
41      end
42      ////When a miss is triggered in a write access, asking upper level memory for the line
43      else if ((waiting_write_finished_s==1'b0) && write_en_i && (hit_w==0)) begin
44          mem_read_en_o = 1'b1;
45          mem_write_en_o = 1'b0;
46          write_valid_o = 1'b0;
47      end
48      ////When data is correctly written in cache and upper level memory, sends the validity
49      signal
50      else if (waiting_write_finished_s && write_en_i) begin
51          write_valid_o = 1'b1;
52          mem_read_en_o = 1'b0;
53          mem_write_en_o = 1'b0;
54      end
55      ////Default case
56      else begin
57          write_valid_o = 1'b0;
58          mem_read_en_o = 1'b0;
59          mem_write_en_o = 1'b0;
60      end
61  end : mux_read_out

```

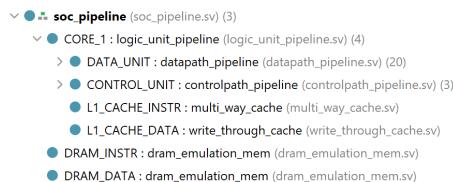
Code 3.20 – Code de logique de sortie de la cache write_through

On ajoute donc seulement les cas en rapport avec l'écriture par rapport à la cache multivoies, ce qui rajoute de nombreux cas mais ne complexifie pas les appels en écriture, qui restent complètement inchangées.

3.4.2 Adaptation du processeur RISCV

On peut donc maintenant ajouter la cache pour améliorer les performances des écritures dans la mémoire (cache de données). Cependant, cela implique de modifier les conditions de propagation des signaux dans les **control_path** et **data_path**, car, la cache et la mémoire supérieures étant "réelles" et possédant un temps de réponse non nul, il faut bloquer la propagation tant que la mémoire de données n'a pas fini son opération.

Pour cela, on modifie donc l'arborescence du processeur, qui devient donc la suivante :



Les caches sont donc placés avec les circuit logique du coeur RISCV (car si on étends notre architecture pour la rendre multicoeur, les caches de niveau L1 sont propres à chaque coeur).

On place ensuite la mémoire RAM au niveau d'abstraction supérieure, car il n'y a qu'un cœur. Pour être parfaitement cohérent, il faudrait les passer à un niveau d'abstraction encore supérieur en parlant d'un SOC, mais on se limite à cette solution ici, car l'on n'a qu'un seul cœur.

Egalement, pour limiter la complexité, étant donnée qu'on a pas de module BIU pour gérer les requêtes mémoire, on instancie 2 modules RAM, différenciées pour les instructions et les données.

On travaille maintenant sur les **control/data_path**, il suffit simplement de faire la même chose que l'on a fait avec le signal **imem_read_valid_i**, mais en ajoutant le signal **dmem_read_valid_i**.

3.4.3 Vérification et analyse des performances

On vient maintenant tester notre cache pour vérifier son bon fonctionnement, aussi bien pour les écritures que pour les lectures. Il est en effet nécessaire de vérifier que les **Corner Case** ne posent pas de problèmes, mais aussi les cas d'utilisation classique avant de l'implémenter sur notre processeur.

On propose pour cela la testbench suivante : **write_through_cache_tb**, qui vient stresser cette mémoire en lui fournissant plusieurs requêtes en écriture/lecture sur la ligne N°1, avec des adresses différentes. Les données ont été au préalable enregistrées dans une mémoire "réelle" qui est reliée à notre cache. On obtient alors le chronogramme suivant :

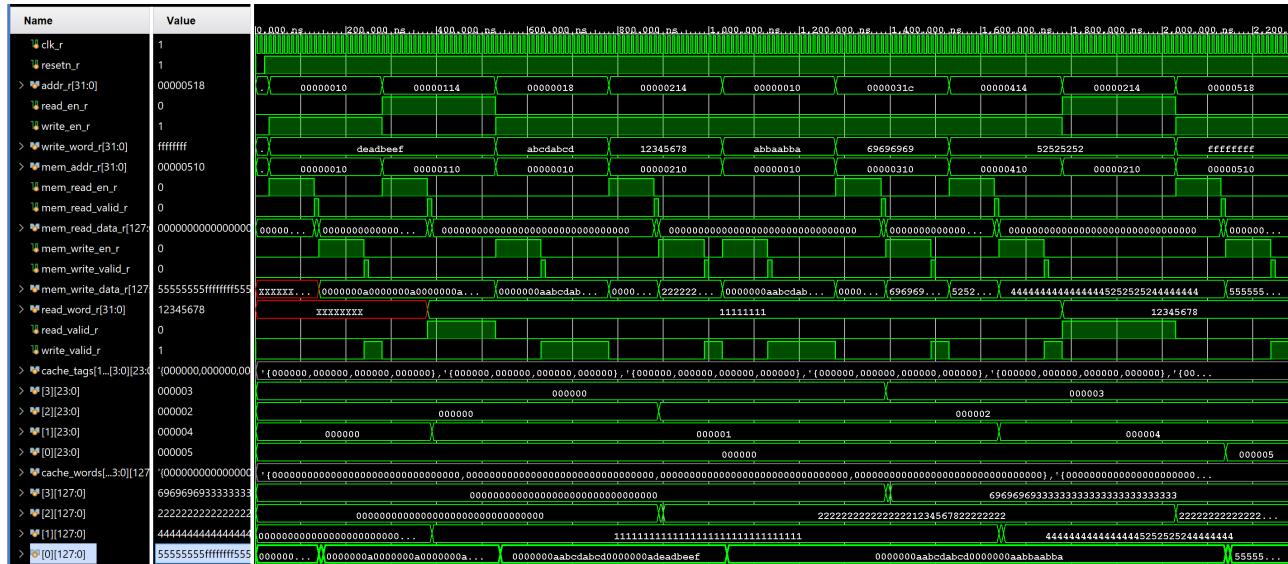


FIGURE 3.15 – Chronogramme de la testbench "write_through_cache_tb"

On observe bien la bonne écriture des données dans la cache, et leur transmissions à la RAM qui s'effectue avant la génération du signal de validité de l'écriture. On passe donc à la création d'un programme assembleur destinée à venir mettre en évidence les avantages et inconvénients de cette cache.

On va donc réaliser un programme simple, ayant pour but de copier les 64 premiers "mots" d'un tableau vers une adresse différente en ajoutant 10 à leur valeur. On va donc pouvoir mettre en évidence le bon fonctionnement des opérations d'écriture et de lecture dans notre

cache. C'est donc ici le programme suivant :

Programme en assembleur de **memcpy.S**

```
.section .start;
.globl start;

start :
    li t0, 0x0          //Base adress for the location of the old array
    li t1, 0x100         //Number of values in the array
    li t2, 0x1000        //Base adress for the location of the new array
    li t3, 0xa           //Value to add to each cell of the array

loop :
    lw t4, 0(t0)         //Load of the array cell value from memory
    add t4,t4,t3          //Adding the offset value to the cell
    sw t4,0(t2)          //Store the new array cell value in memory at the new adress
    addi t0,t0,0x4         //Incrementation of the adress for the next array cells
    addi t2,t2,0x4         //Incrementation of the adress for the next array cells
    bltu t0, t1, loop      //End of loop branch verification

lab1 :
    li ra, 0xDEADBEEF
    j lab1

.end start
```

On l'exécute et on obtient le chronogramme suivant :

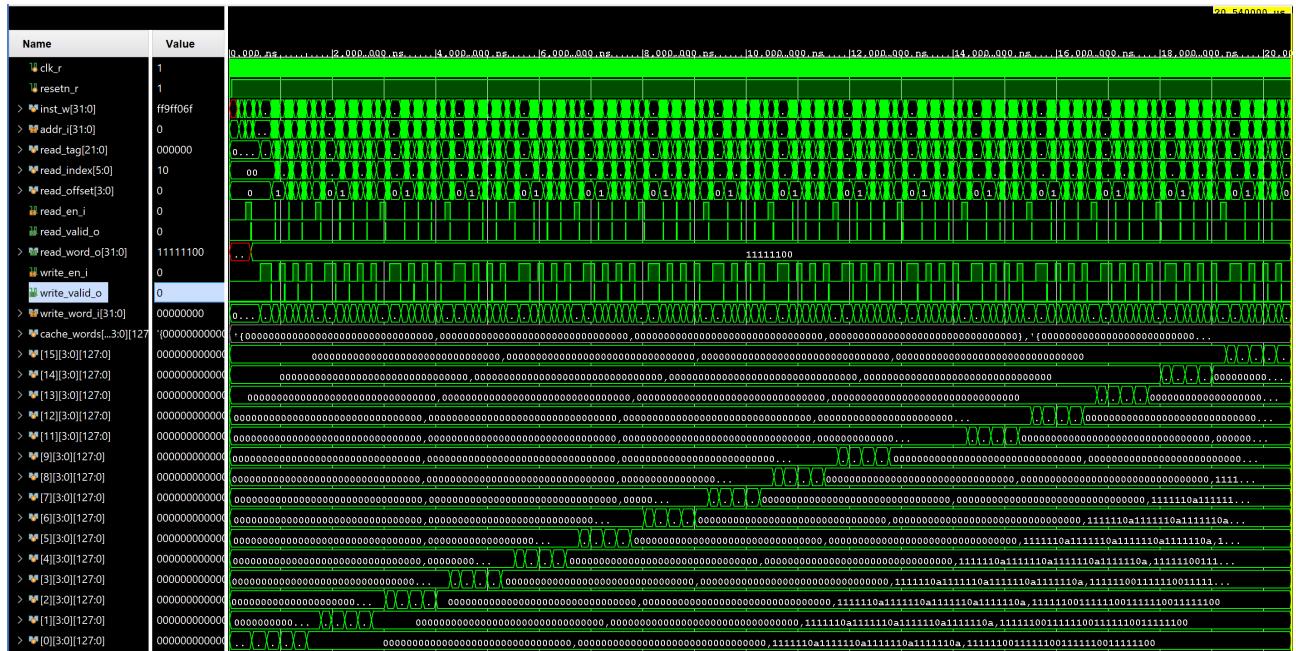


FIGURE 3.16 – Chronogramme du programme "memcpy.S" avec la cache de données "write_through_cache"

On trouve un temps d'exécution très long, de $20,54\mu s$ ce qui correspond donc à une vitesse de copie dans la mémoire d'environ :

$$12.5 Mo/s$$

Ce temps est long, mais on a ici directement une mémoire cache et RAM cohérente, on peut donc voir que les données sont bien présentes à la bonne adresse dans cette dernière :

> [256][127:0]	111110a111110a111110a111110a
> [257][127:0]	111110a111110a111110a111110a
> [258][127:0]	111110a111110a111110a111110a
> [259][127:0]	111110a111110a111110a111110a
> [260][127:0]	111110a111110a111110a111110a
> [261][127:0]	111110a111110a111110a111110a
> [262][127:0]	111110a111110a111110a111110a
> [263][127:0]	111110a111110a111110a111110a
> [264][127:0]	111110a111110a111110a111110a
> [265][127:0]	111110a111110a111110a111110a
> [266][127:0]	111110a111110a111110a111110a
> [267][127:0]	111110a111110a111110a111110a
> [268][127:0]	111110a111110a111110a111110a
> [269][127:0]	111110a111110a111110a111110a
> [270][127:0]	111110a111110a111110a111110a
> [271][127:0]	111110a111110a111110a111110a

Un zoom sur l'écriture de la première ligne dans la mémoire nous permet de mettre en évidence les pertes de performance, qui sont ici du au manque du bypass sur les instructions de type **S** et **Load** (instructions que l'on avait retiré du bypass pour le garder simple). On s'intéresse donc ensuite à les rajouter, avec la logique pour les gérer, dans le but d'améliorer les performances.

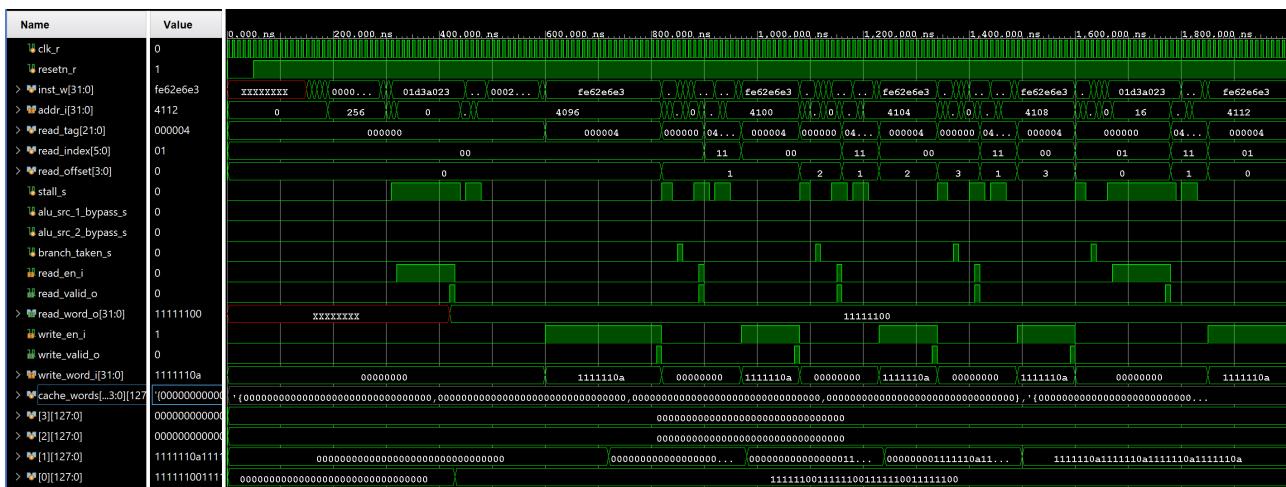


FIGURE 3.17 – Zoom sur le chronogramme du programme "memcpy.S" avec la cache de données "write_through_cache"

3.4.4 Amélioration des performances par le bypass

L'implémentation précédente du Bypass pour RS2 ne prenait pas en compte les instructions de type **LOAD** et **S**, car elles ont une complexité accrue pour leur gestion. On va donc dorénavant modifier notre circuit pour les prendre en compte.

3.4.4.1 Bypass de RS2 pour les instructions de type S

Dans une instruction de type **S**, la donnée RS2 ne sert pas à effectuer un calcul dans l'ALU, mais est utilisée dans l'étage **MEM**. En effet, cette donnée représente la donnée à enregistrer dans la mémoire. On modifie donc notre circuit **data_path** pour incorporer un signal de contrôle **alu_src_2_bypass_i**, qui nous permet de choisir entre la sortie de l'ALU et du banc de registres, pour la propagation de RS2 au travers du mur de registres, à destination de l'étage **MEM**.

```

1  ///////////////////////////////////////////////////////////////////
2  ///////////////////////////////////////////////////////////////////Choose the value to be written in memory
3  configurable_mux #(2,32) MUX_REG_RS2 (
4      .data_i({rd_data_wb_s,
5          rd_data_mem_s,
6          rd_data_exe_s,
7          rs2_data_s}),
8      .sel_i(mem_data_select_i),
9      .data_o(rs2_data_dec_s)
);

```

Code 3.21 – Modifications apportées dans le data_path pour le bypass type S

Dans le **control_path**, il est nécessaire de modifier la façon dont on génère le signal de contrôle du multiplexeur lié à l'entrée OP2 de l'ALU. En effet, il ne faut pas prendre la sortie de l'ALU en cas de bypass pour une instruction de type S, car on veut ici "mettre" la valeur immédiate (IMM) dans l'opérande 2 de l'ALU.

```

1  ///////////////////////////////////////////////////////////////////
2  ///////////////////////////////////////////////////////////////////Mux controlling the entry operand 2 of the ALU
3  always_comb begin : alu_src2_comb
4      if (rs2_bypass_exe_s && (opcode_dec_s != RV32I_OPCODE_S)) alu_src_2_o =
MUX_SEL_OP2_BYPASS_EXE;
5      else if (rs2_bypass_mem_s && (opcode_dec_s != RV32I_OPCODE_S)) alu_src_2_o =
MUX_SEL_OP2_BYPASS_MEM;
6      else if (rs2_bypass_wb_s && (opcode_dec_s != RV32I_OPCODE_S)) alu_src_2_o =
MUX_SEL_OP2_BYPASS_WB;
7      else begin
8          case (opcode_dec_s)
9              RV32I_OPCODE_I_JALR, RV32I_OPCODE_I_OPER : alu_src_2_o = MUX_SEL_OP2_IMM;
10             RV32I_OPCODE_U_AUIPC, RV32I_OPCODE_I_LOAD: alu_src_2_o = MUX_SEL_OP2_IMM;
11             RV32I_OPCODE_S: alu_src_2_o = MUX_SEL_OP2_IMM;
12             default: alu_src_2_o = MUX_SEL_OP2_RS2;
13         endcase
14     end
end : alu_src2_comb

```

Code 3.22 – Modifications apportées dans le control_path pour le bypass type S

3.4.4.2 Bypass de RS2 pour les instructions de type LOAD

Les instructions de type LOAD quand à elle, n'imposent pas de modification dans le bypass de RS2 directement. Cependant, dans ce type d'instructions, la valeur de **RD** n'est pas prête à la fin de l'étage **EXE**, en effet, elle provient de la mémoire et n'est donc prête qu'à la fin de l'étage **MEM**. Une vérification supplémentaire est donc nécessaire dans le **control_path** afin d'empêcher la génération du bypass si l'instruction présente dans l'étage **EXE** est une instruction de type **LOAD**.

```

1  ///////////////////////////////////////////////////////////////////
2  ///////////////////////////////////////////////////////////////////Logic being used to generate the dependency related signals (stall, nop and bypass)
3  always_comb begin : dependency_comb
4      rd_usage_exe_s = rd_used(opcode_exe_s);
5      rd_usage_mem_s = rd_used(opcode_mem_s);
6      rd_usage_wb_s = rd_used(opcode_wb_s);
7      case (opcode_dec_s)
8          RV32I_OPCODE_R, RV32I_OPCODE_S, RV32I_OPCODE_B : begin
9              fetch_jump_dec_s = 1'b0;
10             ///////////////////////////////////////////////////////////////////Checking for dependency between RS1 and RD
11             if (rs1_addr_dec_s==5'h00) begin
12                 stall_rs1_s = 1'b0;
13                 rs1_bypass_exe_s = 1'b0;
14                 rs1_bypass_mem_s = 1'b0;
15                 rs1_bypass_wb_s = 1'b0;
16             end
17             else if (opcode_exe_s == RV32I_OPCODE_I_LOAD) begin
18                 stall_rs1_s = (~|(rs1_addr_dec_s ^ rd_addr_exe_s)) && rd_usage_exe_s;
19                 rs1_bypass_exe_s = 1'b0;
20                 rs1_bypass_mem_s = (~|(rs1_addr_dec_s ^ rd_addr_mem_s)) && rd_usage_mem_s;
21                 rs1_bypass_wb_s = (~|(rs1_addr_dec_s ^ rd_addr_wb_s)) && rd_usage_wb_s;
22             end
23             else begin
24                 stall_rs1_s = 1'b0;
25             end
26         end
27     end
28 
```

```

24         rs1_bypass_exe_s = (~|(rs1_addr_dec_s ^ rd_addr_exe_s)) && rd_usage_exe_s;
25         rs1_bypass_mem_s = (~|(rs1_addr_dec_s ^ rd_addr_mem_s)) && rd_usage_mem_s;
26         rs1_bypass_wb_s = (~|(rs1_addr_dec_s ^ rd_addr_wb_s)) && rd_usage_wb_s;
27     end
28     ////Checking for dependency between RS2 and RD
29     if (rs2_addr_dec_s==5'h00) begin
30         stall_rs2_s = 1'b0;
31         rs2_bypass_exe_s = 1'b0;
32         rs2_bypass_mem_s = 1'b0;
33         rs2_bypass_wb_s = 1'b0;
34     end
35     else if (opcode_exe_s == RV32I_OPCODE_I_LOAD) begin
36         stall_rs2_s = (~|(rs2_addr_dec_s ^ rd_addr_exe_s)) && rd_usage_exe_s;
37         rs2_bypass_exe_s = 1'b0;
38         rs2_bypass_mem_s = (~|(rs2_addr_dec_s ^ rd_addr_mem_s)) && rd_usage_mem_s;
39         rs2_bypass_wb_s = (~|(rs2_addr_dec_s ^ rd_addr_wb_s)) && rd_usage_wb_s;
40     end
41     else begin
42         stall_rs2_s = 1'b0;
43         rs2_bypass_exe_s = (~|(rs2_addr_dec_s ^ rd_addr_exe_s)) && rd_usage_exe_s;
44         rs2_bypass_mem_s = (~|(rs2_addr_dec_s ^ rd_addr_mem_s)) && rd_usage_mem_s;
45         rs2_bypass_wb_s = (~|(rs2_addr_dec_s ^ rd_addr_wb_s)) && rd_usage_wb_s;
46     end
47     stall_s = stall_rs1_s | stall_rs2_s;
48 end

```

Code 3.23 – Modifications apportées dans le control_path pour le bypass type LOAD

3.4.4.3 Mise en évidence des améliorations

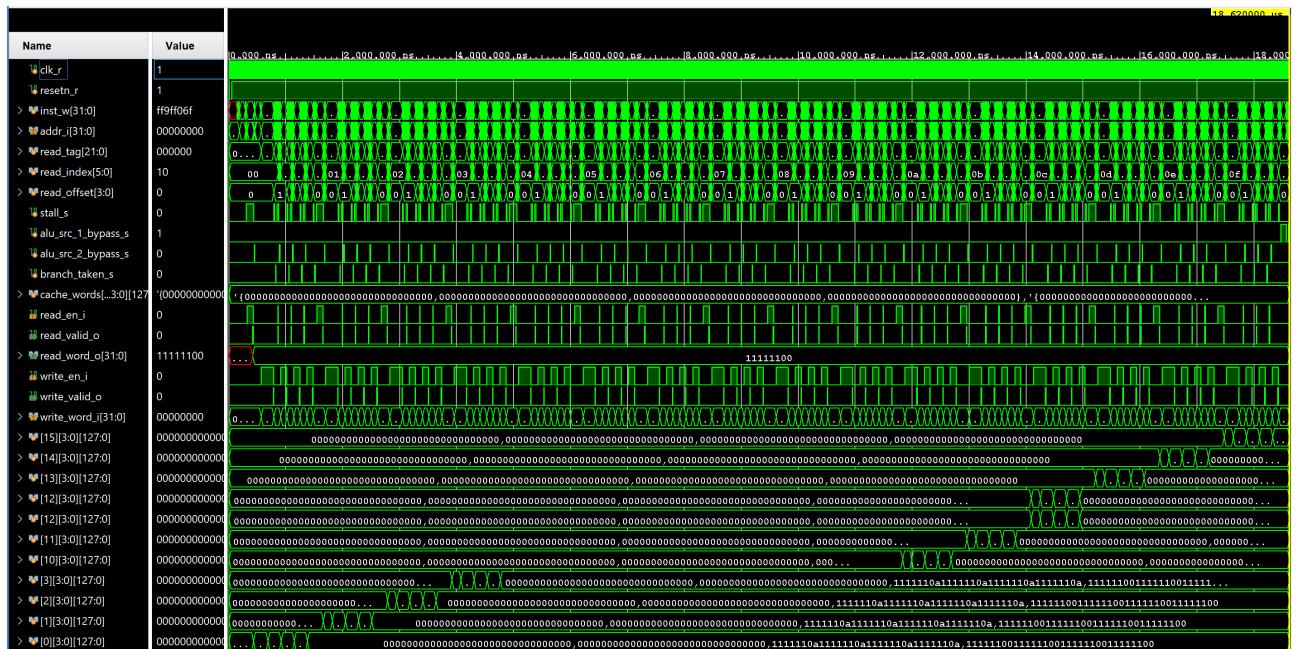


FIGURE 3.18 – Chronogramme du programme "memcpy.S" avec la cache de données "write_through_cache" et le bypass corrigé sur RS2

On observe ainsi une augmentation mesurable des performances, en effet, la copie se termine au bout de $18.62\mu s$, ce qui se traduit par une augmentation d'environ 10% des performances, et donc une vitesse de copie dans la mémoire d'environ :

$$13.7 Mo/s$$

3.5 Cache de données avec écriture "write back"

3.5.1 Crédation du module "write_back_cache"

La cache **write_through** possède un désavantage majeur, elle nécessite un appel mémoire pour chaque écriture afin de garantir la cohérence avec cette dernière. On perds donc énormément de cycles d'horloge à transmettre des morceaux de lignes à la mémoire, ce qui ralentit grandement le processeur. On viens donc implémenter une cache de type **write_back**, qui ne transmet la ligne modifiée au processeur qu'uniquement lorsque cette dernière est remplacée dans la cache. On peut donc garantir la cohérence des mémoires, tout en évitant de perdre énormément de cycles à renvoyer des lignes en cours de modification.

Pour cela, on reprends donc la cache **write_through**, dont on vient modifier le fonctionnement. Les entrées et sorties restent en effet les même, seul l'affectation de leurs valeurs change.

La première modification a apportée est la suppression de la "bascule" **waiting_write_finished**, car on ne l'utilisera pas ici. En effet, on implémente une autre solution pour gérer les écritures et lectures, en se reposant sur un cycle de latence.

```
1 //Adding a delay to the generation of the write enable input in order to avoid issues
2 always_ff @(posedge write_valid_out_s, posedge change_addr_word_s, posedge clk_i, negedge
3   write_en_i) begin : write_enable_delay
4   if ((write_en_i == 1'b0)) begin
5     write_enable_delay_s <= 1'b0;
6     write_en_s <= 1'b0;
7   end
8   else if (write_valid_out_s && (change_addr_word_s==1'b0)) begin
9     write_en_s <= write_enable_delay_s;
10    write_enable_delay_s <= 1'b0;
11  end
12  else begin
13    write_en_s <= write_enable_delay_s;
14    write_enable_delay_s <= write_en_i;
15  end
16 end : write_enable_delay
17
18 //Signal indicating if there is a change in the adress or the word to write, reinitialised
19 //on the negedge of the clock
20 always_ff @(addr_i, write_word_i, negedge clk_i) begin
21   if (clk_i == 1'b0) change_addr_word_s <= 1'b0;
22   else change_addr_word_s <= 1'b1;
23 end
```

Code 3.24 – Bascule gérant la latence en entrée pour l'écriture

```
1 //Adding a delay to the generation of the write valid output in order to avoid issues
2 always_ff @(write_valid_s, posedge clk_i) begin
3   if (write_valid_s==1'b0) begin
4     write_valid_delay_s <= 1'b0;
5     write_valid_out_s <= 1'b0;
6   end
7   else begin
8     write_valid_delay_s <= write_valid_s;
9     write_valid_out_s <= write_valid_delay_s;
10  end
11 end
12
13 //Logic outputs assignation
14 assign mem_write_en_o = mem_write_en_s;
15 assign mem_read_en_o = mem_read_en_s;
16 assign write_valid_o = write_valid_out_s;
```

Code 3.25 – Bascule gérant la latence en sortie pour l'écriture

L'ajout de cette latence sur la génération des signaux est essentielle pour éviter les écritures à des adresses non voulues. En effet, sans cette latence, l'écriture se fait instantanément dans la cache, alors que l'adresse n'est pas encore correctement modifiée. Celà ne pose pas de problème pour la lecture, car elle est asynchrone et l'adresse a donc le temps de se mettre à jour.

Cependant, l'écriture dans les registres étant synchrone, il est nécessaire de patienter un cycle pour la mise à jour des signaux afin d'écrire.

On retrouve bien cette modification dans la partie "mémoire" de notre cache, et plus précisément dans la partie dédiée à l'écriture.

```

1   //-----//  
2   //Memory" of the module  
3   //-----//  
4   //Registers implementation  
5   generate  
6     for (genvar i=0;i<NrLines;i++) begin : registers_cache  
7       for (genvar j=0;j<NB_WAYS;j++) begin : registers_cache_ways  
8           always_ff @(posedge clk_i or negedge rstn_i) begin : write_cache  
9             ///Reset  
10            if(rstn_i==1'b0) begin  
11              cache_line_validity[i][j] <= 1'b0;  
12              cache_line_dirty_bit[i][j] <= 1'b0;  
13              cache_tags[i][j] <= 0;  
14              cache_words[i][j] <= 0;  
15            end  
16            ////Update of a way of the cache, with a data coming from a higher level  
17            memory  
18            else if (mem_read_valid_i && (read_index==i) && (cache_LRU[i][0+:BITS_WAYS  
19                ==j)) begin  
20              cache_line_validity[i][j] <= 1'b1;  
21              cache_line_dirty_bit[i][j] <= 1'b0;  
22              cache_tags[i][j] <= read_tag;  
23              cache_words[i][j] <= mem_read_data_i;  
24              cache_LRU[i] <= {cache_LRU[i][0+:BITS_WAYS],cache_LRU[i][BITS_WAYS+:(  
25                NB_WAYS-1)*BITS_WAYS]};  
26              end  
27              ////Update a way of the cache according to the hit value, when it has been  
28              transmitted to upper level memory succesfully  
29              //, with a data/word coming from the processor or a lower level memory  
30              else if (hit_w[cache_LRU[i][(j*BITS_WAYS)+:BITS_WAYS]] && write_en_s && (i  
31                ==read_index)) begin  
32                cache_line_dirty_bit[i][cache_LRU[i][(j*BITS_WAYS)+:BITS_WAYS]] <= 1'  
33                b1;  
34                cache_words[i][cache_LRU[i][(j*BITS_WAYS)+:BITS_WAYS]][(read_offset*  
35                WORD_SIZE)+:WORD_SIZE] <= write_word_i;  
36                if (j==0) cache_LRU[i] <= {cache_LRU[i][0+:BITS_WAYS],cache_LRU[i][  
37                  BITS_WAYS+:(NB_WAYS-1)*BITS_WAYS]};  
38                else if (j<(NB_WAYS-1)) cache_LRU[i] <= {cache_LRU[i][(j*BITS_WAYS)+:  
39                  BITS_WAYS],cache_LRU[i][(j+1)*BITS_WAYS):(NB_WAYS-j-1)*BITS_WAYS],cache_LRU[i][0+:j*  
40                  BITS_WAYS]};  
41                end  
42                ////Modification of the LRU value after a succesfull read in a way  
43                else if ((hit_w!=0) && (cache_LRU[i][(j*BITS_WAYS)+:BITS_WAYS]==  
44                  hit_index_w) && read_en_i && (i==read_index)) begin  
45                  if (j==0) cache_LRU[i] <= {cache_LRU[i][0+:BITS_WAYS],cache_LRU[i][  
46                    BITS_WAYS+:(NB_WAYS-1)*BITS_WAYS]};  
47                  else if (j<(NB_WAYS-1)) cache_LRU[i] <= {cache_LRU[i][(j*BITS_WAYS)+:  
48                    BITS_WAYS],cache_LRU[i][(j+1)*BITS_WAYS):(NB_WAYS-j-1)*BITS_WAYS],cache_LRU[i][0+:j*  
49                    BITS_WAYS]};  
50                  end  
51                  else if (mem_write_valid_i && (i==read_index) && (cache_LRU[i][0+:  
52                    BITS_WAYS]==j)) begin  
53                    cache_line_dirty_bit[i][j] <= 1'b0;  
54                    end  
55                  end : write_cache  
56                  ////Modification of the output based on the hit signal  
57                  always_comb begin : read_cache  
58                      if (hit_w[j]) begin  
59                        registers_output_lines_tags[i] <= cache_tags[i][j];  
60                        registers_output_all[i] <= cache_words[i][j];  
61                      end  
62                  end : read_cache  
63              end : registers_cache_ways  
64              ////Initialisation of the LRU during a reset  
65              always_ff @(posedge clk_i or negedge rstn_i) begin : LRU_cache  
66                  if(rstn_i==1'b0) begin  
67                      cache_LRU[i] <= int_lru();
```

```

53         end
54     end : LRU_cache
55   end : registers_cache
56 endgenerate

```

Code 3.26 – Partie mémoire de la cache write_back

Ici, on ajoute une donnée supplémentaire à mémoriser, **cache_line_dirty_bit**. Cette donnée correspond à l'état de la ligne/voie, si une écriture a été effectuée dedans, elle doit être mise à l'état **1**. Elle n'est alors remise à **0** que lorsque la ligne "dirty" a été transmise à la mémoire pour garantir sa cohérence. On peut alors remplacer la ligne dans la cache sans perdre la cohérence.

L'étape suivante est la génération des signaux de contrôle de la mémoire, que l'on vient modifier pour coller au "protocole" de notre nouvelle cache.

```

1 //-----//  

2 // "Post-Memory" logic of the module  

3 //-----//  

4 //Registers output logic and data assignation  

5 always_comb begin : mux_read_out
6     ///////////////////////////////////////////////////////////////////  

7     if (rstn_i==1'b0) begin
8         mem_read_en_s = 1'b0;
9         mem_write_en_s = 1'b0;
10        write_valid_s = 1'b0;
11    end
12    ///////////////////////////////////////////////////////////////////  

13    else if (mem_read_valid_i && write_en_i) begin
14        mem_read_en_s = 1'b0;
15        mem_write_en_s = 1'b0;
16        write_valid_s = 1'b0;
17    end
18    ///////////////////////////////////////////////////////////////////  

19    else if (write_en_i && hit_w) begin
20        mem_read_en_s = 1'b0;
21        write_valid_s = 1'b1;
22        mem_write_en_s = 1'b0;
23    end
24    ///////////////////////////////////////////////////////////////////  

25    else if(write_en_i && rstn_i && cache_line_dirty_bit[read_index][cache_LRU[read_index]
26 [0+:BITS_WAYS]]) begin
27        mem_write_data_o = cache_words[read_index][cache_LRU[read_index][0+:BITS_WAYS]];
28        mem_read_en_s = 1'b0;
29        mem_write_en_s = 1'b1;
30        write_valid_s = 1'b0;
31    end
32    ///////////////////////////////////////////////////////////////////  

33    else if (write_en_i && (hit_w==0)) begin
34        mem_read_en_s = 1'b1;
35        mem_write_en_s = 1'b0;
36        write_valid_s = 1'b0;
37    end
38    ///////////////////////////////////////////////////////////////////  

39    else if (mem_read_valid_i && read_en_i) begin
40        registers_output_line = mem_read_data_i;
41        mem_read_en_s = 1'b0;
42        mem_write_en_s = 1'b0;
43        write_valid_s = 1'b0;
44    end
45    ///////////////////////////////////////////////////////////////////  

46    else if (hit_w && read_en_i) begin
47        registers_output_line = registers_output_all[read_index];
48        mem_read_en_s = 1'b0;
49        write_valid_s = 1'b0;
50        mem_write_en_s = 1'b0;
51    end
52    ///////////////////////////////////////////////////////////////////  

53    else begin
54        mem_read_en_s = 1'b0;
55        mem_write_en_s = 1'b0;
56        write_valid_s = 1'b0;
57    end
58 end

```

```

52      else if(read_en_i && rstn_i && cache_line_dirty_bit[read_index][cache_LRU[read_index]
53          ][0+:BITS_WAYS]]) begin
54          mem_write_data_o = cache_words[read_index][cache_LRU[read_index][0+:BITS_WAYS]];
55          mem_read_en_s = 1'b0;
56          mem_write_en_s = 1'b1;
57          write_valid_s = 1'b0;
58      end
59      ////In case of a miss, asking the upper level memory for the line
60      else if(read_en_i && rstn_i) begin
61          mem_read_en_s = 1'b1;
62          mem_write_en_s = 1'b0;
63          write_valid_s = 1'b0;
64      end
65      ////Default case
66      else begin
67          write_valid_s = 1'b0;
68          mem_read_en_s = 1'b0;
69          mem_write_en_s = 1'b0;
70      end
71  end : mux_read_out

```

Code 3.27 – Gestion des signaux de contrôle de la cache write_back

Pour les écritures et les lectures, en cas **miss**, il est alors nécessaire de vérifier l'état de la ligne (le "dirty" bit), si il est à **1**, il est alors nécessaire de demander une écriture (transmission) de ligne à la mémoire. Dès que le "dirty" bit est à **0** on peut directement demander la nouvelle ligne et, soit écrire, soit lire la donnée.

En cas de **hit**, si on a une écriture, on attend donc **1 cycle** avant d'écrire, puis on peut transmettre le bit de validité. Pour les lectures, la transmission de la donnée lue, et du bit de validité reste instantanée.

3.5.2 Vérification et analyse des performances

Afin de tester les performances, on execute le programme précédent avec notre nouvelle cache :

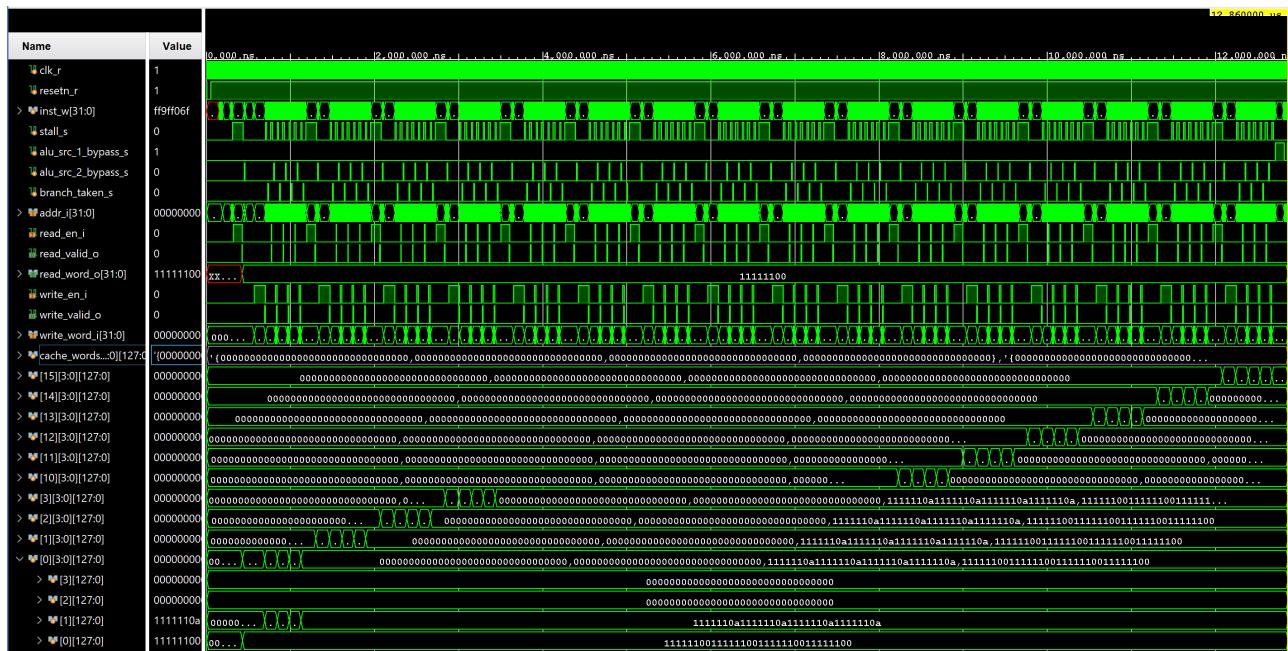


FIGURE 3.19 – Chronogramme du programme "memcpy.S" avec la cache de données "write_back_cache" et le bypass corrigé sur RS2

On observe ainsi une nouvelle augmentation des performances, bien plus drastique cette fois.

Le programme s'exécute en effet en seulement $12.86\mu s$ (amélioration de 30%). Il est cependant nécessaire de rappeler que les données ne sont présentes que dans la cache, et n'ont pas encore été copiés dans la mémoire principale. Ce temps est donc certes bien plus court, mais on aura une pénalité au moment de l'éviction des lignes qui ont été écrites dans la mémoire, de 10 cycles par lignes, ce qui représente donc $1.6\mu s$ ici. Même en prenant en compte cette pénalité qui ne sera "servie" que plus tard, on arrive à un temps total d'exécution de $14.46\mu s$, ce qui reste une amélioration d'environ 20% des performances. Ceci représente donc une vitesse de copie dans la mémoire de :

$$\begin{aligned} Vitesse maximale &= 19.9 Mo/s \\ Vitesse avec penalitee &= 17.7 Mo/s \end{aligned}$$

Un focus sur l'exécution d'une boucle permet de mettre en évidence le temps gagné sur les écritures en mémoire :

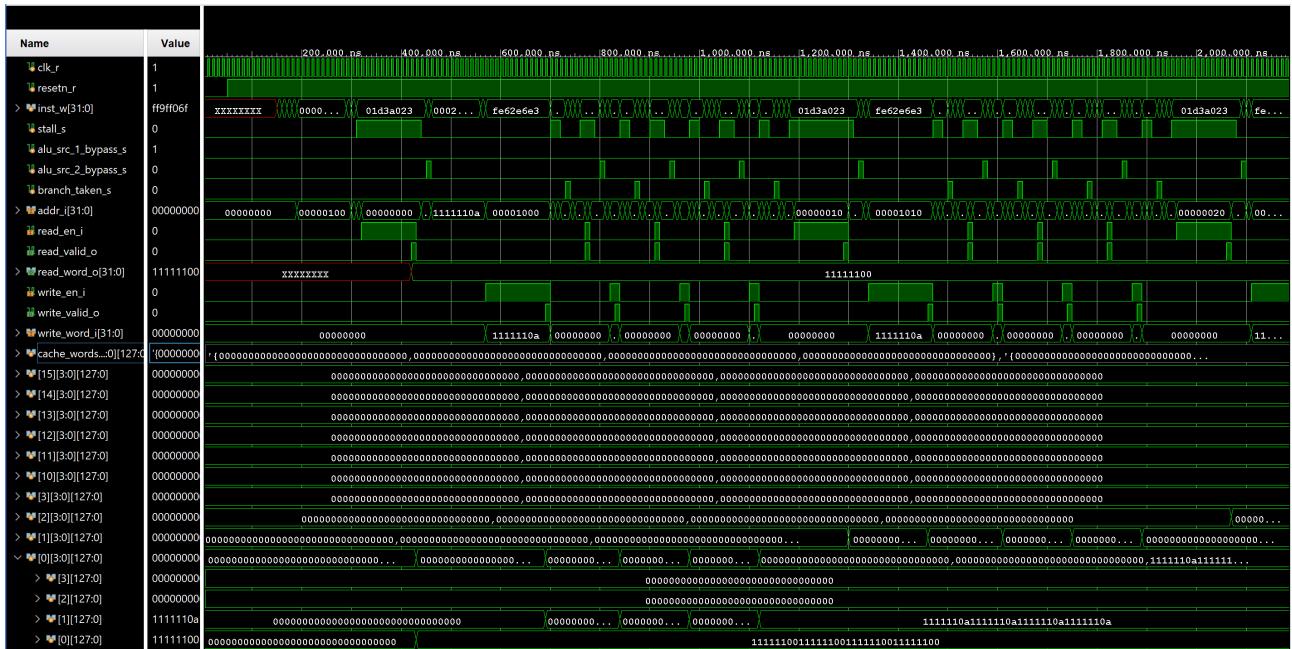


FIGURE 3.20 – Zoom sur une boucle du programme "memcpy.S" avec la cache de données "write_back_cache" et le bypass corrigé sur RS2

3.6 Amélioration complète du bypass

3.6.1 Modifications apportées

Afin d'améliorer davantage les performances de notre architecture, et avant de nous intéresser à la gestion des prédictions, il est possible de nous débarasser presque entièrement des dépendances de données. C'est en effet le **Stall** qui nous cause ici la perte de nombreux cycles, et donc des pertes de performances. Il est donc intéressant de venir améliorer l'implémentation de notre **Bypass**, afin de limiter au maximum l'utilisation de ce dernier.

Pour cela, il est nécessaire de différencier la génération du signal de **Bypass**, suivant les étages. Nous allons donc dans un premier temps nous intéresser aux modifications à apporter au **control_path**.

3.6.1.1 Modifications du control_path

On vient ici modifier la génération des signaux de contrôle qui sont liés à l'**ALU**, aux écritures mémoires et au **Bypass**. Pour celà, il est nécessaire de bien comprendre les conditions qui nous permettent de générer ce dernier suivant les étages. En réalité, on n'a besoin du signal de **stall** dans un seul et unique cas, car dans tout les autres le **bypass** suffit pour régler les dépendances de données.

Le cas "problématique" est la présence d'une instruction de type **LOAD** dans l'étage **EXE**. En effet, une instruction de type **LOAD** enregistre dans le registre **RD** une donnée présente dans la mémoire, à une adresse calculée par l'intermédiaire de l'**ALU**. La valeur de **RD** qui est concernée par la dépendance de donnée n'est donc "prête" qu'à partir de l'étage **MEM**.

Pour tout les autres cas, on génère alors des signaux de Bypass avec la formule suivante :

$$rsX_bypass_ZZZ_s = (\sim \|(rsX_addr_dec_s \oplus rd_addr_ZZZ_s))$$

Où X prends les valeurs 1/2 suivant que l'instruction de l'étage **DEC** utilise ou non RS1/RS2, et ZZZ prends les valeurs EXE, MEM ou WB pour représenter les différents étages dans lesquels la dépendance de donnée s'applique.

Ceci nous donne alors la logique suivante dans le **control_path** :

```

1 /////////////////////////////////////////////////////////////////// Logic being used to generate the dependency related signals (stall, nop and bypass)
2 always_comb begin : dependency_comb
3     rd_usage_exe_s = rd_used(opcode_exe_s);
4     rd_usage_mem_s = rd_used(opcode_mem_s);
5     rd_usage_wb_s = rd_used(opcode_wb_s);
6     case (opcode_dec_s)
7         RV32I_OPCODE_R, RV32I_OPCODE_S, RV32I_OPCODE_B : begin
8             fetch_jump_dec_s = 1'b0;
9             ////Checking for dependency between RS1 and RD
10            if (rs1_addr_dec_s==5'h00) begin
11                stall_rs1_s = 1'b0;
12                rs1_bypass_exe_s = 1'b0;
13                rs1_bypass_mem_s = 1'b0;
14                rs1_bypass_wb_s = 1'b0;
15            end
16            else if (opcode_exe_s == RV32I_OPCODE_I_LOAD) begin
17                stall_rs1_s = (~|(rs1_addr_dec_s ^ rd_addr_exe_s)) && rd_usage_exe_s;
18                rs1_bypass_exe_s = 1'b0;
19                rs1_bypass_mem_s = (~|(rs1_addr_dec_s ^ rd_addr_mem_s)) && rd_usage_mem_s;
20                rs1_bypass_wb_s = (~|(rs1_addr_dec_s ^ rd_addr_wb_s)) && rd_usage_wb_s;
21            end
22            else begin
23                stall_rs1_s = 1'b0;
24                rs1_bypass_exe_s = (~|(rs1_addr_dec_s ^ rd_addr_exe_s)) && rd_usage_exe_s;
25                rs1_bypass_mem_s = (~|(rs1_addr_dec_s ^ rd_addr_mem_s)) && rd_usage_mem_s;
26                rs1_bypass_wb_s = (~|(rs1_addr_dec_s ^ rd_addr_wb_s)) && rd_usage_wb_s;
27            end
28            ////Checking for dependency between RS2 and RD
29            if (rs2_addr_dec_s==5'h00) begin
30                stall_rs2_s = 1'b0;
31                rs2_bypass_exe_s = 1'b0;
32                rs2_bypass_mem_s = 1'b0;
33                rs2_bypass_wb_s = 1'b0;
34            end
35            else if (opcode_exe_s == RV32I_OPCODE_I_LOAD) begin
36                stall_rs2_s = (~|(rs2_addr_dec_s ^ rd_addr_exe_s)) && rd_usage_exe_s;
37                rs2_bypass_exe_s = 1'b0;
38                rs2_bypass_mem_s = (~|(rs2_addr_dec_s ^ rd_addr_mem_s)) && rd_usage_mem_s;
39                rs2_bypass_wb_s = (~|(rs2_addr_dec_s ^ rd_addr_wb_s)) && rd_usage_wb_s;
40            end
41            else begin
42                stall_rs2_s = 1'b0;
43                rs2_bypass_exe_s = (~|(rs2_addr_dec_s ^ rd_addr_exe_s)) && rd_usage_exe_s;
44                rs2_bypass_mem_s = (~|(rs2_addr_dec_s ^ rd_addr_mem_s)) && rd_usage_mem_s;
45                rs2_bypass_wb_s = (~|(rs2_addr_dec_s ^ rd_addr_wb_s)) && rd_usage_wb_s;
46            end

```

```

47         stall_s = stall_rs1_s | stall_rs2_s;
48     end
49     RV32I_OPCODE_I_JALR, RV32I_OPCODE_I_LOAD , RV32I_OPCODE_I_OPER ,
50     RV32I_OPCODE_I_ENVCSR : begin
51         fetch_jump_dec_s = 1'b0;
52         rs2_bypass_exe_s = 1'b0;
53         rs2_bypass_mem_s = 1'b0;
54         rs2_bypass_wb_s = 1'b0;
55         ////Checking for dependency between RS1 and RD
56         if (rs1_addr_dec_s==5'h00) begin
57             stall_s = 1'b0;
58             rs1_bypass_exe_s = 1'b0;
59             rs1_bypass_mem_s = 1'b0;
60             rs1_bypass_wb_s = 1'b0;
61         end
62         else if (opcode_exe_s == RV32I_OPCODE_I_LOAD) begin
63             stall_s = (~|(rs1_addr_dec_s ^ rd_addr_exe_s)) && rd_usage_exe_s;
64             rs1_bypass_exe_s = 1'b0;
65             rs1_bypass_mem_s = (~|(rs1_addr_dec_s ^ rd_addr_mem_s)) && rd_usage_mem_s;
66             rs1_bypass_wb_s = (~|(rs1_addr_dec_s ^ rd_addr_wb_s)) && rd_usage_wb_s;
67         end
68         else begin
69             stall_s = 1'b0;
70             rs1_bypass_exe_s = (~|(rs1_addr_dec_s ^ rd_addr_exe_s)) && rd_usage_exe_s;
71             rs1_bypass_mem_s = (~|(rs1_addr_dec_s ^ rd_addr_mem_s)) && rd_usage_mem_s;
72             rs1_bypass_wb_s = (~|(rs1_addr_dec_s ^ rd_addr_wb_s)) && rd_usage_wb_s;
73         end
74     end
75     RV32I_OPCODE_J : begin
76         stall_s = 1'b0;
77         fetch_jump_dec_s = 1'b1;
78         rs1_bypass_exe_s = 1'b0;
79         rs1_bypass_mem_s = 1'b0;
80         rs1_bypass_wb_s = 1'b0;
81         rs2_bypass_exe_s = 1'b0;
82         rs2_bypass_mem_s = 1'b0;
83         rs2_bypass_wb_s = 1'b0;
84     end
85     default : begin
86         stall_s = 1'b0;
87         fetch_jump_dec_s = 1'b0;
88         rs1_bypass_exe_s = 1'b0;
89         rs1_bypass_mem_s = 1'b0;
90         rs1_bypass_wb_s = 1'b0;
91         rs2_bypass_exe_s = 1'b0;
92         rs2_bypass_mem_s = 1'b0;
93         rs2_bypass_wb_s = 1'b0;
94     end
95 endcase
96 fetch_branch_s = branch_taken_s || fetch_jalr_s;
97 fetch_jump_exe_s = branch_taken_s || fetch_jalr_s;
98 fetch_jump_s = fetch_jump_dec_s | fetch_jump_exe_s;
99 end : dependency_comb

```

Code 3.28 – Gestion des dépendances, avec le bypass complet

Maintenant, il faut générer les signaux de contrôle du **data_path** afin de sélectionner les bonnes données à propager depuis l'étage **DEC**, suivant les signaux de **bypass**.

Afin d'avoir la donnée la plus "fraîche", il est nécessaire de prendre en compte en priorité le signal de **bypass** lié à l'étage **EXE**, et ainsi de suite. La donnée **RD** que l'on récupère sera alors bien la dernière modification qui lui sera apportée et sera donc bien toujours à jour, où qu'elle soit dans les étages du circuit.

```

1 //Mux controlling the entry operand 1 of the ALU
2 always_comb begin : alu_src1_comb
3     if (rs1_bypass_exe_s) alu_src_1_o = MUX_SEL_OP1_BYPASS_EXE;
4     else if (rs1_bypass_mem_s) alu_src_1_o = MUX_SEL_OP1_BYPASS_MEM;
5     else if (rs1_bypass_wb_s) alu_src_1_o = MUX_SEL_OP1_BYPASS_WB;
6     else begin
7         case (opcode_dec_s)
8             RV32I_OPCODE_U_LUI: alu_src_1_o = MUX_SEL_OP1_IMM;
9             RV32I_OPCODE_U_AUIPC: alu_src_1_o = MUX_SEL_OP1_PC;

```

```

10         default: alu_src_1_o = MUX_SEL_OP1_RS1;
11     endcase
12   end
13   end : alu_src1_comb
14
15 //Mux controlling the entry operand 2 of the ALU
16 always_comb begin : alu_src2_comb
17   if (rs2_bypass_exe_s && (opcode_dec_s != RV32I_OPCODE_S)) alu_src_2_o =
18     MUX_SEL_OP2_BYPASS_EXE;
19   else if (rs2_bypass_mem_s && (opcode_dec_s != RV32I_OPCODE_S)) alu_src_2_o =
20     MUX_SEL_OP2_BYPASS_MEM;
21   else if (rs2_bypass_wb_s && (opcode_dec_s != RV32I_OPCODE_S)) alu_src_2_o =
22     MUX_SEL_OP2_BYPASS_WB;
23   else begin
24     case (opcode_dec_s)
25       RV32I_OPCODE_I_JALR, RV32I_OPCODE_I_OPER : alu_src_2_o = MUX_SEL_OP2_IMM;
26       RV32I_OPCODE_U_AUIPC, RV32I_OPCODE_I_LOAD: alu_src_2_o = MUX_SEL_OP2_IMM;
27       RV32I_OPCODE_S: alu_src_2_o = MUX_SEL_OP2_IMM;
28       default: alu_src_2_o = MUX_SEL_OP2_RS2;
29     endcase
30   end
31 end : alu_src2_comb

```

Code 3.29 – Contrôle des entrées de l’ALU, avec le bypass complet

Il est cependant nécessaire de modifier également le signal propagé pour les instructions de type **STORE**, en effet, pour ces dernières, on propage directement la valeur **RS2** directement au travers des étages **EXE** et **MEM**. On la choisit donc avec cette partie :

```

1 always_comb begin : bypass_S_type_comb
2   if (rs2_bypass_exe_s) mem_data_select_o = RS2_BYPASS_EXE;
3   else if (rs2_bypass_mem_s) mem_data_select_o = RS2_BYPASS_MEM;
4   else if (rs2_bypass_wb_s) mem_data_select_o = RS2_BYPASS_WB;
5   else mem_data_select_o = RS2_BYPASS_REG_BANK;
6 end : bypass_S_type_comb

```

Code 3.30 – Contrôle de l’opérande des instructions de type S, avec le bypass complet

Enfin, la dernière partie du **control_path** concerne le choix de la valeur de **RD** sur laquelle s’effectue le **bypass**. Pour l’étage **WB**, ce choix est déjà effectué, car on a déjà besoin de le faire pour l’écriture dans le registre. On choisira donc directement la valeur à écrire dans le banc de registre. Cependant, pour les étages **EXE** et **MEM**, on a plusieurs choix possible en fonction de l’instruction dans cet étage.

- Pour l’étage **EXE** :
 - La sortie de l’ALU
 - PC+4
- Pour l’étage **MEM** :
 - La sortie de l’ALU
 - PC+4
 - La valeur récupérée dans la mémoire

Cela se traduit donc par cette partie du code, qui contrôle le choix de la valeur de RD à bypass pour les 2 étages explicités précédemment :

```

1 //Logic being used to know which rd to select in the EXE stage for the bypass
2 always_comb begin : rd_bypass_exe_select_comb
3   case (opcode_exe_s)
4     RV32I_OPCODE_I_JALR, RV32I_OPCODE_J : rd_bypass_exe_o = RD_EXE_BYPASS_PC_PLUS_4;
5     default : rd_bypass_exe_o = RD_EXE_BYPASS_ALU;
6   endcase
7 end : rd_bypass_exe_select_comb
8
9 //Logic being used to know which rd to select in the MEM stage for the bypass
10 always_comb begin : rd_bypass_mem_select_comb
11   case (opcode_mem_s)
12     RV32I_OPCODE_I_JALR, RV32I_OPCODE_J : rd_bypass_mem_o = RD_MEM_BYPASS_PC_PLUS_4;

```

```

13     RV32I_OPCODE_I_LOAD : rd_bypass_mem_o = RD_MEM_BYPASS_MEM_DATA;
14     default : rd_bypass_mem_o = RD_MEM_BYPASS_ALU;
15   endcase
16 end : rd_bypass_mem_select_comb

```

Code 3.31 – Choix de la valeur de RD à propager, avec le bypass complet

3.6.1.2 Modifications du data_path

Enfin, on ajuste notre circuit **data_path** afin qu'il supporte les modifications du bypass apportées dans le **control_path**.

Dans un premier temps, on ajoute donc des multiplexeurs permettant de choisir les valeurs de **RD** à choisir suivant l'étage et l'instruction présente à l'intérieur.

```

1 ///////////////////////////////////////////////////////////////////Choose the RD that is taken from the EXE stage for the bypass
2 configurable_mux #(1,32) MUX_BYPASS_RD_EXE (
3   .data_i({pc_plus4_exe_s,
4           alu_data_out_s}),
5   .sel_i(rd_bypass_exe_i),
6   .data_o(rd_data_exe_s)
7 );
8 ///////////////////////////////////////////////////////////////////Choose the RD that is taken from the MEM stage for the bypass
9 configurable_mux #(2,32) MUX_BYPASS_RD_MEM (
10   .data_i({cache_read_data_mem_s,
11            pc_plus4_mem_s,
12            alu_data_out_mem_s}),
13   .sel_i(rd_bypass_mem_i),
14   .data_o(rd_data_mem_s)
15 );

```

Code 3.32 – Affectation de la valeur de RD suivant l'étage, avec le bypass complet

Ensuite, il est maintenant nécessaire de modifier les **MUX** qui affectent les entrées **OP1** et **OP2** de l'**ALU**.

```

1 ///////////////////////////////////////////////////////////////////Choose the OP1 of the ALU
2 configurable_mux #(3,32) MUX_ALU_SRC1 (
3   .data_i({rd_data_wb_s,
4           rd_data_mem_s,
5           rd_data_exe_s,
6           pc_addr_s,
7           imm_gen_data_s,
8           rs1_data_s}),
9   .sel_i(alu_src_1_i),
10  .data_o(alu_op1_dec_s)
11 );
12 ///////////////////////////////////////////////////////////////////Choose the OP2 of the ALU
13 configurable_mux #(3,32) MUX_ALU_SRC2 (
14   .data_i({rd_data_wb_s,
15           rd_data_mem_s,
16           rd_data_exe_s,
17           imm_gen_data_s,
18           rs2_data_s}),
19   .sel_i(alu_src_2_i),
20   .data_o(alu_op2_dec_s)
21 );

```

Code 3.33 – Affectation des valeurs OP1 et OP2 suivant l'étage, avec le bypass complet

Et finalement, on modifie le multiplexeur qui permet de choisir la valeur à écrire dans la mémoire de donnée pour qu'il prenne en compte le **bypass** entre les différents étages.

```

1 ///////////////////////////////////////////////////////////////////Choose the value to be written in memory
2 configurable_mux #(2,32) MUX_REG_RS2 (
3   .data_i({rd_data_wb_s,
4           rd_data_mem_s,
5           rd_data_exe_s,
6           rs2_data_s}),

```

```

7     .sel_i(mem_data_select_i),
8     .data_o(rs2_data_dec_s)
9  );

```

Code 3.34 – Affectation de la valeur à écrire en mémoire suivant l'étage, avec le bypass complet

3.6.2 Mise en évidence de l'amélioration des performances

Cette implémentation du **bypass** complété nous permet donc d'augmenter grandement les performances de notre circuit, pour le prouver, nous allons une nouvelle fois executer le programme "memcpy.S", et observer le gain de performances.

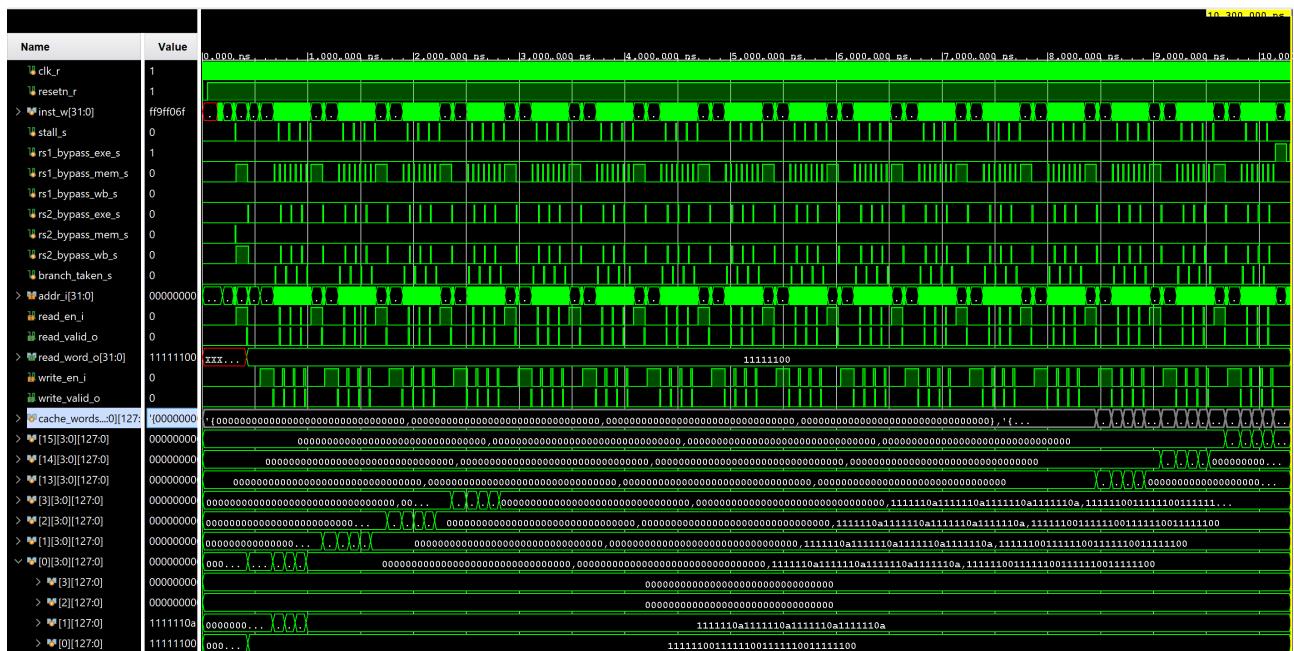


FIGURE 3.21 – Chronogramme du programme "memcpy.S" avec la cache de données "write_back_cache" et le bypass complet sur RS2

On observe dorénavant un temps d'exécution de seulement $10.3\mu s$, ce qui est encore une amélioration importante ($11.9\mu s$ si l'on prends en compte la pénalité future). Ceci correspond donc à une vitesse de copie de :

$$\begin{aligned} Vitesse maximale &= 24.8 Mo/s \\ Vitesse avec penalitee &= 21.5 Mo/s \end{aligned}$$

En résumé, on a donc réussi à presque doubles les performances de notre circuit pour la copie de données, en ajoutant un nouveau type de mémoire cache plus efficace, et en implémentant un bypass complet, permettant de réduire au minimum le nombre de **stall**, et donc le nombre de cycles perdus.

Afin d'améliorer encore les performances, il pourrait être intéressant d'implémenter un système de prédictions de branchement, ce qui pourrait augmenter encore d'avantage les performances du au grand nombre de boucles qu'effectuent les programmes.

Chapitre 4

Annexe

4.1 Connexions entre les différents blocs

Voici le diagramme représentant les connexions entre les différents blocs, avec leurs tailles et noms pour le fichier final :

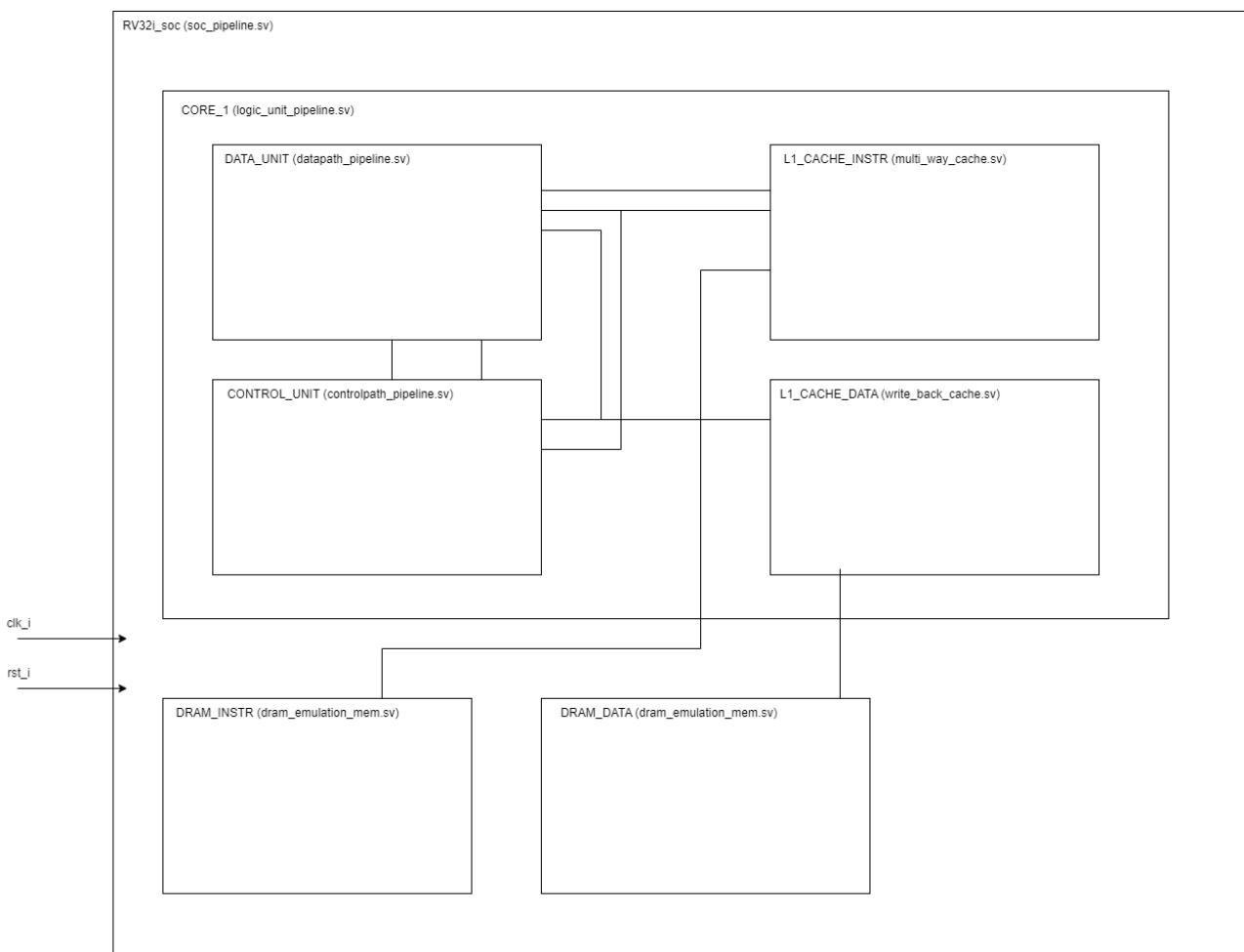


FIGURE 4.1 – Diagramme rapide de la "die" du soc

4.2 Problèmes de compilations et solutions

En cas de problème de compilation du fichier avec Modelsim (tallin), il peut être nécessaire de remplacer le module "dram_emulation_mem" par le module "dram_emulation_mem_modelsim" dans le fichier "soc_pipeline.sv".

Le problème viendrait de modelsim, qui refuse une affectation effectuée dans un bloc "initial begin", alors que c'est une initialisation de la mémoire, le 2nd module corrige ce problème en réaffectant les entrées et sorties plusieurs fois pour garder la cohérence des modules.

4.3 Explications de l'archive .zip

Dans cette archive, vous trouverez l'ensemble des codes que nous avons réalisés pour ce projet, dans leur version finale uniquement.

L'ensemble des codes a été réécrit, depuis 0, et ne contient donc plus aucune partie des codes originels fournis initialement. Ils sont néanmoins commentés dans leur intégralités, avec des explications (en anglais), et des séparations claires pour les différentes parties de l'architecture.

Des testbenches sont également présentes pour pouvoir tester indépendamment la grande majorité des modules fournis.

Le package est également différent que celui fourni (oui oui on a vraiment TOUT modifié), la logique reste similaire, mais les noms reflète mieux notre compréhension et nous semble plus naturels.

TOUT les fichiers présents dans le dossier "**hdl_src**" sont nécessaires pour la bonne compilation du code, néanmoins, seuls les fichiers présents dans le dossier "**TP_specific**" concerne directement les parties modifiées pour ce TP.

Seuls les fichiers présents dans le dossier "**tb/TP_specific**" sont néanmoins utiles pour tester les différents modules. Les fichiers présents dans le dossier "**tb/Usefull**" sont des testbenches qui concernent des modules utilisés dans ce projet, mais non modifiés.

L'ensemble des codes utilisables pour tester cette architecture sont présents dans le dossier **firmware/Assembleur**, les fichiers **.hex** leurs correspondant sont directement présents dans le dossier **firmware**. Pour les programmes qui le nécessitent, les données de la mémoire de "data" sont également présents dans le dossier **firmware**.

4.4 Limitations de l'architecture fournie

Cette architecture ne supporte actuellement pas les instructions **ecall** et **ebreak**. Les instructions de type **JALR** sont cependant supportées en théorie, mais non testées faute de temps...

PS pour Virgile : J'ai pas trouvé comment executer les benchmarks que j'ai trouvé sur internet du coup j'ai pas pu le faire.