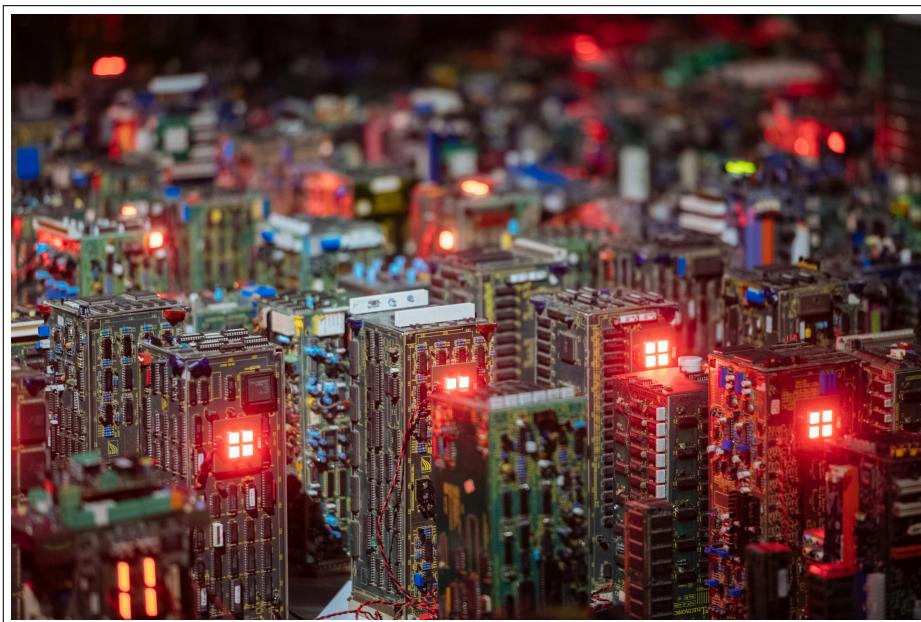




Rapport de TPs

Architecture des Processeurs 2



Alexandra BARON - Ness TCHENIO

Janvier 2025

Encadrant : Olivier Potin

ISMIN EI23

Sommaire

1	Introduction	4
2	TP 1 : RV32I architecture pipeline	5
2.1	Étude du processeur RISC-V pipeliné	5
2.1.1	Examinez le sous circuit data_path	6
2.1.2	Examinez le sous circuit control_path	6
2.1.3	Examinez l'instance dmem du composant wsync_mem	7
2.1.4	Examinez l'instance imem du composant wsync_mem	7
2.2	Exécution et simulation d'un programme	8
3	TP 2	11
3.1	Éxécution d'un programme	11
3.2	Correction du problème	12
3.2.1	Correction logicielle	12
3.2.2	Correction matérielle : Interlock	13
3.3	Correction matérielle : Bypass	19
4	TP 3	21
4.1	Cache mémoire "Direct"	21
4.2	Performance	21
4.3	Cache instruction direct	21
4.4	Cache instruction associatif à deux voies	24
5	Conclusion	27
6	Annexes	28

Table des figures

1	RV32i_top	5
2	Waveform de la simulation de l'exo1	8
3	Fichier main.S	8
4	Waveform de la simulation de main.S	9
5	Fichier main.S	9
6	Waveform de la simulation du main en prenant en compte les dépendances.	10
7	Programme Mult.S sans prendre en compte des dépendances.	11
8	Waveform Mult.S sans prendre en compte des dépendances.	12
9	Waveform Mult.S avec prise en compte des dépendances (ajout de NOPs)	13
10	Waveform Mult.S avec prise en compte des dépendances (ajout de NOPs)	13
11	Bloc combinatoire pour mettre à jour les flags	15
12	Signal <i>stall_w</i>	16
13	Signal <i>stall_w</i> lors de la simulation	16
14	Réultat de la simulation	16
15	Top avec gestion de saut.	17
16	Top avec gestion de branchemet.	17
17	Gestion des dépendances de contrôle. <i>Datapath</i>	18
18	Gestion des dépendances de contrôle. <i>Datapath</i>	18
19	Gestion des dépendances de contrôle. <i>Controlpath</i>	18
20	Simulation complète	19
21	Modification de <i>alu_src1_mux_comb</i>	19
22	Modification du stall lié à l'implémentation du bypass	20
23	Simulation avec bypass	20
24	RV32i_soc avec mémoire cache	21
25	Opcodes (fichier RV32i_pkg.sv).	28

1-Introduction

Les processeurs RISCV-V32 sont des processeurs basés sur une architecture à jeu d’instructions réduit qui utilisent des registres de 32 bits. Ils sont aujourd’hui largement utilisés dans les systèmes embarqués, en raison de leur faible consommation énergétique et leur coût réduit.

Comme les autres processeurs, ils utilisent une architecture pipeline pour améliorer leur performance, en découpant le traitement d’une instruction en plusieurs étapes exécutées en parallèle. Dans le cadre de notre cours d’Architecture des Processeurs, nous étudions l’architecture pipeline à cinq étapes des processeurs RISCV-32 : Instruction Fetch (IF), Instruction Decode (ID), Execution (EXE), Memory Access (MEM) et Write Back (WB).

Afin de mieux comprendre le fonctionnement des pipelines de ces processeurs, nous avons réalisé trois travaux pratiques, dont nous vous proposons le compte-rendu dans ce rapport. Ces TP nous ont permis de nous familiariser avec l’architecture pipeline, d’apprendre à gérer les différents conflits et dépendances pouvant survenir lors du traitement parallèle des instructions, et enfin d’implémenter divers types de cache servant de mémoire.

2-TP 1 : RV32I architecture pipeline

Ce TP était l'occasion pour nous de découvrir le fonctionnement de l'architecture pipelinée des processeurs RISC-V32, en travaillant notamment sur le datapath et le controlpath. C'était aussi l'occasion de tester des algorithmes simples et de simuler leur fonctionnement sur le logiciel *modelsim*

2.1 Étude du processeur RISC-V pipeliné

Question 1 :

Le circuit top-level est RV32i_soc. Il regroupe les sous-circuits RV32i_datapath, RV32i_controlpath, InstMem et DataMem.

Question 2 :

RV32i_controlpath : Implémentation du chemin de contrôle pour un processeur monocycle conforme à l'architecture RISC-V 32 bits (RV32I). Ce chemin de contrôle gère la logique décisionnelle qui détermine comment les différentes unités du processeur interagissent pour exécuter les instructions.

RV32i_datapath : Implémentation du chemin de données d'un processeur monocycle suivant l'architecture RISC-V (RV32I). Son rôle est d'assurer que toutes les données passent par les cinq étapes suivantes : Fetch (IF), Decode (ID), Execution (EXE), Memory Access (MEM) et Write-back (WB).

InstMem : Cette mémoire stocke l'instruction obtenue lors de l'étape de *fetch* à l'adresse donnée et renvoie l'instruction en sortie. Il est uniquement possible de lire à partir de cette mémoire.

DataMem : Cette mémoire stocke la donnée qui traverse la pipeline à l'adresse spécifiée. Il est possible d'y écrire et d'y lire.

Question 3 :

Les sous circuits instanciés dans le RV32i_top sont RV32i_datapath et RV32i_controlpath.
RV32i_datapath gère les flux de données et RV32i_controlpath coordonne les signaux de contrôle.

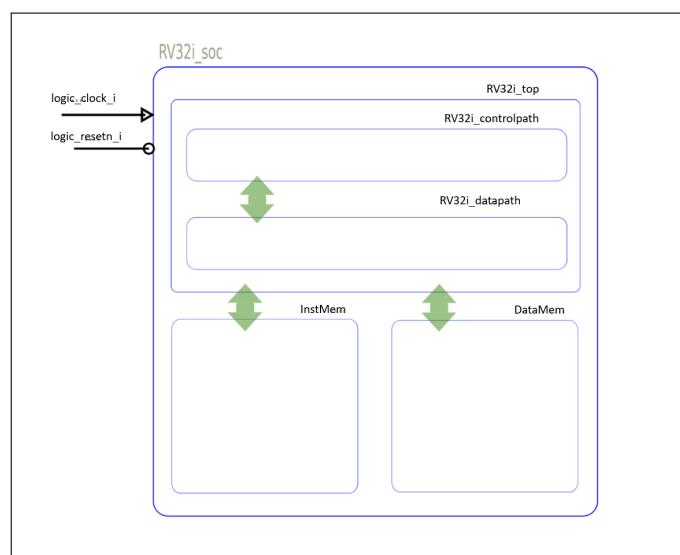


FIGURE 1 – RV32i_top

2.1.1 Examinez le sous circuit data_path

Question 4 :

Le cœur RISC-V est composé de cinq étages :

Instruction Fetch (IF) : Récupère l'instruction à l'adresse indiquée, dans la mémoire d'instructions.

Instruction Decode (ID) : Décode l'instruction et récupère les opérandes nécessaires.

Instruction Execution (EXE) : Exécute l'opération définie par l'instruction (par exemple, via l'ALU).

Instruction Memory Access(MEM) : Accède à la mémoire de données si nécessaire (lecture ou écriture).

Instruction Write-back(WB) : Écrit les résultats dans les registres.

Question 5 :

Les signaux en sortie du datapath et à destination du controlpath sont : *instruction*, *alu_zero_w* et *alu_lt_w*. *instruction* correspond à l'instruction à exécuter.

alu_zero_w est le signal binaire qui indique si le résultat de l'ALU est nul.

alu_lt_w est le signal binaire qui indique si l'opérande 1 est inférieure à l'opérande 2 selon la logique du calcul effectué dans l'ALU.

Question 6 :

Le signal en sortie du datapath et à destination de la mémoire d'instructions est : *imem_addr_o*.

Ce signal correspond à l'adresse envoyée à la mémoire d'instructions pour récupérer une instruction à exécuter.

Question 7 :

Les signaux en sortie du datapath et à destination de la mémoire données sont : *dmem_addr_o*, *dmem_di_o* et *dmem_ble_o*.

dmem_addr_o correspond à l'adresse envoyée à la mémoire de données.

dmem_di_o transporte les données à écrire dans la mémoire (dans le cas d'une écriture).

dmem_ble_o permet de sélectionner des octets spécifiques dans le mot lu/écrit.

2.1.2 Examinez le sous circuit control_path

Question 8 :

Ces signaux représentent les instructions à différents stades du pipeline.

inst_dec_r représente l'instruction au stade de décodage.

Elle est soit lue depuis l'entrée (*instruction_i*) soit remplacée par une instruction NOP si il y a blocage du pipeline (*stall_w* activé)

inst_exec_r représente l'instruction au stade d'exécution.

Elle est mise à jour à chaque cycle en fonction de la sortie de l'instruction précédente (*inst_dec_r*), sauf en cas de reset (*resetn_i = 0*).

inst_mem_r représente l'instruction au stade mémoire.

Elle est transmise depuis le stade précédent (*inst_exec_r*) et sert à réaliser les opérations, de lecture ou d'écriture, sur la mémoire.

inst_wb_r représente l'instruction au stade d'écriture des résultats.

L'instruction est transmise depuis le stade mémoire (*inst_mem_r*) et est utilisée pour écrire les résultats dans les registres du processeur.

Question 9 :

L'index du registre de destination est extrait dans l'étape de décode (ID). Il est ensuite transféré au banc de registres, où les résultats sont stockés lors de l'étape Write-Back (WB).

Question 10 :

Actuellement, le signal est défini à 0.

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

Il n'y a donc aucun mécanisme de gestion de blocage dans le pipeline.

Question 11 :

Comme expliqué brièvement à la question 8, lorsque le signal stall_w est actif, les instruction sont stoppées dans le pipeline jusqu'à ce que les conditions nécessaires soient remplies, cela afin éviter des erreurs (dépendance ou conflit).

Au stade de décode, l'instruction inst_dec_r est remplacée par une instruction NOP si le signal stall_w est à l'état actif.

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

Les autres instructions du pipeline continuent leur exécution sans perturbation permettant un maintien de la synchronisation entre les différentes étapes.

2.1.3 Examinez l'instance dmem du composant wsync_mem**Question 12 :**

La mémoire de données (dmem) est une instance du module wsync_mem avec le paramètre SIZE défini à 4096.

```
localparam DMEM_SIZE = 4096; (module RV32i_soc)
```

Chaque élément de cette mémoire a une largeur de 32 bits (4 octets). La taille totale en octets vaut $4096 \times 4 = 16384$ octets ou 16 Ko.

Question 13 :

L'adresse de base de la mémoire de données est 0x0001_0000.

Elle est définie dans le paramètre DMEM_BASE_ADDR dans le module RV32i_soc.

```
localparam DMEM_BASE_ADDR = 32'h0001_0000;
```

2.1.4 Examinez l'instance imem du composant wsync_mem**Question 14 :**

De même que pour la mémoire de données, la mémoire d'instruction (imem) a une taille de 16 Ko.

```
localparam IMEM_SIZE = 4096;
```

Question 15 :

L'adresse de base de la mémoire d'instructions est 0x0000_0000. Celle-ci est définie de la même manière que l'adresse de base de la mémoire de données.

```
localparam IMEM_BASE_ADDR = 32'h0000_0000; (module RV32i_soc )
```

2.2 Éxécution et simulation d'un programme

Question 16 :

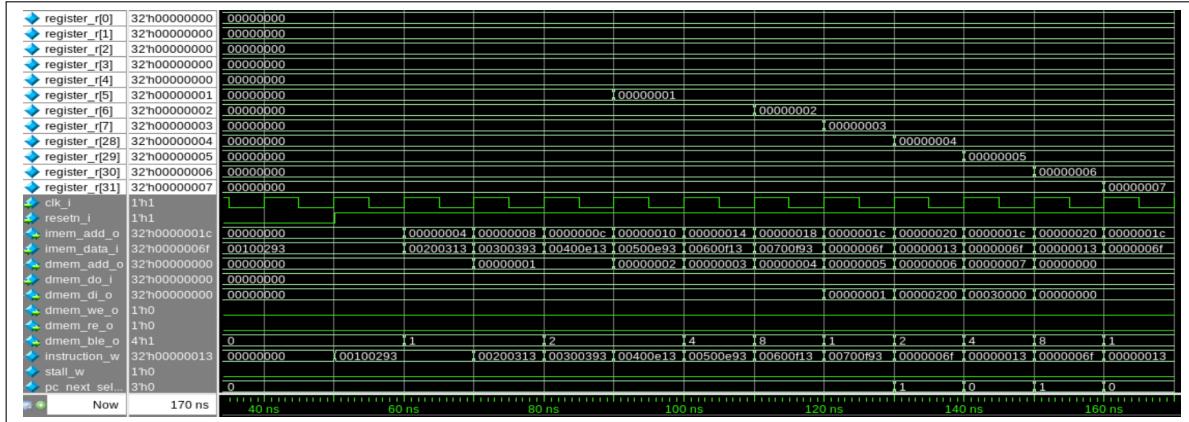


FIGURE 2 – Waveform de la simulation de l'exo1.

Le programme semble s'être déroulé correctement. On constate que tous les registres se sont vu affectés les valeurs indiquées dans le programme exo1.S.

Question 17 :

La simulation se termine grâce à la commande `lab1 : j lab1 nop`.

Effectivement, cette commande crée une boucle infinie qui force le processeur à sauter en permanence à la même adresse sans progresser dans le programme. Cela bloque l'exécution du code, ce qui implique la fin de la simulation.

Question 18 :

Voici le code du main.S :

```
.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
```

FIGURE 3 – Fichier main.S

A la fin de la simulation, on s'attend à voir les valeurs suivantes dans les registres :

$$\begin{array}{ll} t_0 = 0x3 & t_3 = 0x10 \\ t_1 = 0x8 & t_4 = 0x11 \\ t_2 = 0xB & t_5 = 0xFFFFFFFF \end{array}$$

Question 19 :

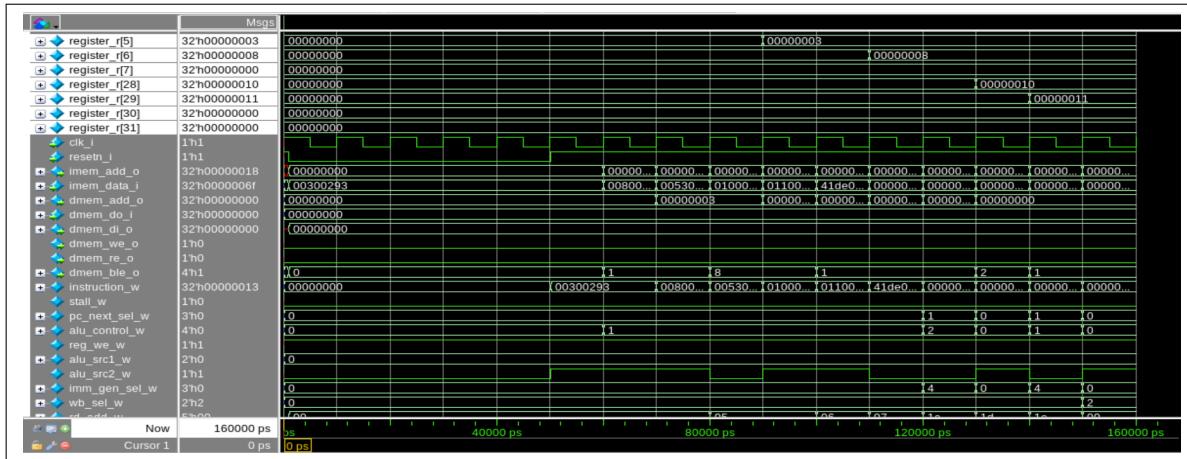


FIGURE 4 – Waveform de la simulation de main.S

On constate qu'à la fin de la simulation, les valeurs des registres ne correspondent pas aux valeurs attendues. Cela s'explique par une dépendance de données entre les différentes commandes. On observe effectivement une dépendance de données avec t_1, t_0 dans la commande $add\ t2, t_1, t_0$ et avec t_3, t_4 dans la commande $sub\ t5, t_3, t_4$. On se propose de rajouter des `nop` entre les commandes créant un conflit.

```

.section .start;
.globl start;

start:
    li t0,0x3
    li t1,0x8
    nop
    nop
    nop
    add t2,t1,t0
    li t3,0x10
    li t4,0x11
    nop
    nop
    sub t5,t3,t4
lab1 : j lab1
    nop

.end start

```

FIGURE 5 – Fichier main.S

L'ajout de ces commandes `nop` permet aux registres de s'actualiser avant d'être utilisés dans des calculs.

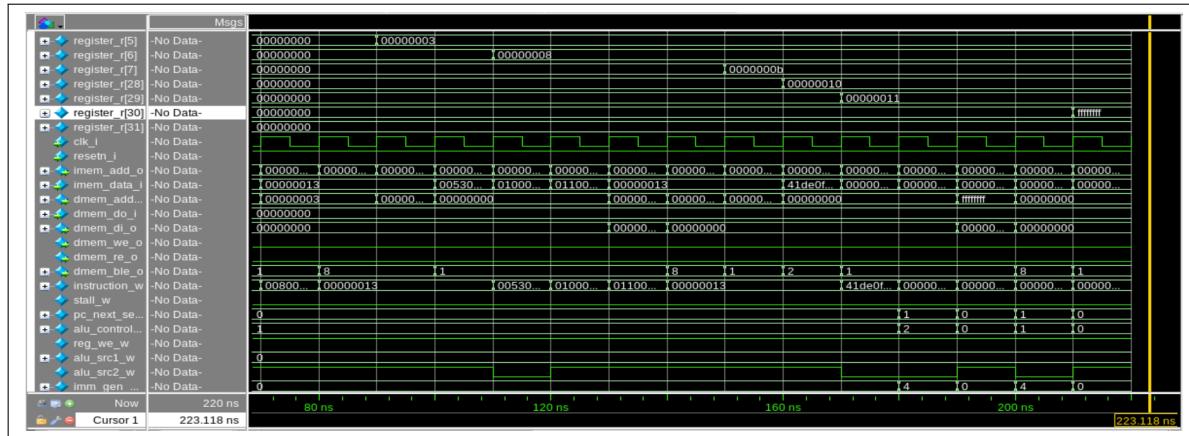


FIGURE 6 – Waveform de la simulation du main en prenant en compte les dépendances.

On constate finalement que les registres présentent la valeur attendue en fin de simulation. Les dépendances de données ont bien été résolues.

3-TP 2

Dans ce TP, nous étudions un programme permettant la multiplication de deux opérandes. Ce programme présentant des dépendances de données et de contrôle le long du pipeline, l'enjeu du TP était de mettre en place différentes méthodes permettant de les régler. Nous avons d'abord mis en place une correction logicielle en insérant des NOP dans le programme dès que nécessaire. Dans un second temps nous avons proposé une correction matérielle avec l'utilisation d'un signal Stall, avant de finalement avoir recours à un bypass.

3.1 Éxécution d'un programme

Question 1 :

```

1 .section .start;
2 .globl start;
3
4 nop ;
5     addi x0,x0,0
6
7 start:
8     li t0,0x8 #t0
9     li t1,0x7 #t1
10    li t2,0x0 #compteur
11    li t3,0x0 #resultat
12    li t4,0x10 #16 pour comparer le compteur
13
14 loop:
15
16    andi t5,t1,1 #t5 = t1 & 1
17    # t5 n'a pas eu le temps de se mettre a jour
18    beq t5,x0,pas_vrai #si ca vaut 0 on fait pas le calcul
19    #control hazard, au moment du branchemetn on prend une décision sur un truc qui a pas ete maj encore. Pour éviter de fetch l'incrémentation de t3
20    add t3,t3,t0
21 pas_vrai :
22    slli t0,t0,1
23    srli t1,t1,1
24    bne t2,t4,loop
25
26
27 lab : j lab
28 #control hazard ! So NOP for jump instruction. Je gère le controle hazard lié au jump
29     nop
30
31 .end start
32

```

FIGURE 7 – Programme Mult.S sans prendre en compte des dépendances.

Ce programme permet d'effectuer la multiplication de deux opérandes, en suivant la logique de l'algorithme 1.

Question 2 :

Décortiquons les itérations :

Itération 1 :

$t_0 = 0x8$ (0000 1000), $t_1 = 0x7$ (0000 0111).
 $t_1 \text{ } 1 = 1$, donc $t_3 = t_3 + t_0 = 0 + 0x8 = 0x8$.

Itération 2 :

$t_0 = 0x10$ (0001 0000), $t_1 = 0x3$ (0000 0011).
 $t_1 \text{ } 1 = 1$, donc $t_3 = t_3 + t_0 = 0x8 + 0x10 = 0x18$.

Itération 3 :

$t_0 = 0x20$ (0010 0000), $t_1 = 0x1$ (0000 0001).
 $t_1 \text{ } 1 = 1$, donc $t_3 = t_3 + t_0 = 0x18 + 0x20 = 0x38$.

Itération 4 :

$t_0 = 0x40$ (0100 0000), $t_1 = 0x0$ (0000 0000).
 $t_1 \text{ } 1 = 0$, donc pas d'ajout à t_3 .

Sur les 16 itérations effectuées, seules les 3 premières influencent le calcul, le nombre d'étapes utiles étant déterminé par le nombre de 1 dans le registre t_1 .

A l'issue de la simulation, le registre t_3 contient donc la valeur $0x38$.

Cela est cohérent avec la valeur attendue. En effet, $8 * 7 = 56 = 0x38$.

Question 3 :

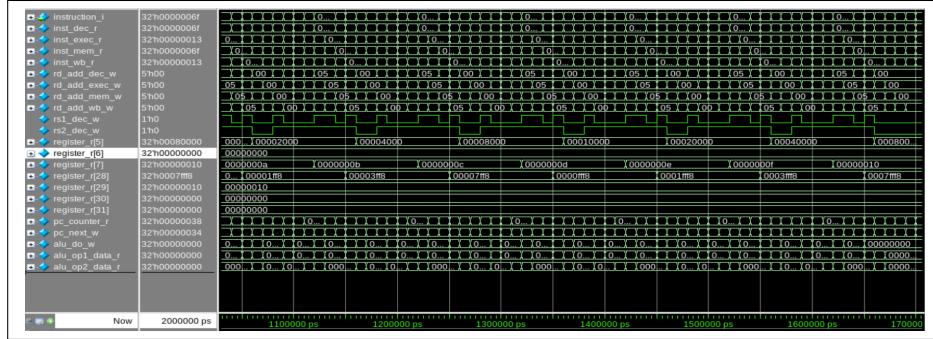


FIGURE 8 – Waveform Mult.S sans prendre en compte des dépendances.

Le résultat lu au registre 28 ne correspond pas à la valeur attendue dans t3. Cela est dû aux dépendances dans le programme. On constate effectivement une dépendance de données entre les lignes 16 et 18 ainsi qu'une dépendance de contrôle entre les lignes 18 et 20.

Question 4 :

Les dépendances observées dans ce programme sont principalement liées aux étapes ID (décodage) et EXE (exécution). Les instructions utilisent des données encore en cours de traitement dans l'ALU (dépendance de données), ou bien les branchements changent le flux d'exécution et nécessitent du temps pour s'actualiser (dépendance de contrôle).

3.2 Correction du problème

3.2.1 Correction logicielle

Question 5 :

Le principe de l'instruction NOP est une instruction qui ne modifiera rien au programme mais qui comportera pour un cycle d'horloge.

Ainsi, pour coder une instruction NOP, on peut par exemple utiliser l'instruction suivante :

$$\text{addi } x_0, x_0, 0$$

où x_0 est un registre pré-enregistré qui vaut toujours 0.

Question 6 :

On utilise des NOP pour régler les dépendances de données ainsi que les dépendances de contrôle.

Après la ligne 16, on insère trois NOPs. En effet, le résultat de l'instruction *andi* n'est disponible qu'à l'étape Write-Back, alors que l'instruction suivante dépend de ce résultat dès l'étape Decode. Cela crée une dépendance de données, et il est nécessaire de décaler l'instruction suivante de trois cycles d'horloge pour permettre à l'instruction *andi* de passer par les étapes Execute (EXE), Memory (MEM) et Write-Back (WB), avant que son résultat ne soit utilisé.

Après la ligne 20 on insère 2 NOPs. Le résultat du branchement étant obtenu à l'étape EXE, l'insertion de 2 NOPS permet de bloquer l'arrivée d'instructions qui pourraient s'avérer érronées en cas de branchement, dans les étapes précédentes (IF et ID). Cela permet de résoudre une dépendance de données (instruction de type B).

```

1 .section .start;
2 .globl start;
3
4 nop ;
5      addi x0,x0,0
6
7 start:
8      li t0,0x8 #t0
9      li t1,0x7 #t1
10     li t2,0x0 #compteur
11     li t3,0x0 #resultat
12     li t4,0x10 #16 pour comparer le compteur
13
14 loop:
15     addi t2,t2,1
16     andi t5,t1,1 #t5 = t1 & 1
17     nop
18     nop
19     nop # t5 n'a pas eu le temps de se mettre a jour
20     beq t5,x0,pas_vrai #si ca vaut 0 on fait pas le calcul
21     nop #control hazard, au moment du branchemet on prend une décision sur un truc qui a pas ete maj encore. Pour éviter de fetch l'incrémentation de t3
22     nop
23     add t3,t3,t0
24 pas_vrai:
25     slli t0,t0,1
26     srli t1,t1,1
27     bne t2,t4,loop
28
29
30 lab : j lab
31 #control hazard ! So NOP for jump instruction. Je gère le controle hazard lié au jump
32     nop
33
34 .end start
35

```

FIGURE 9 – Waveform Mult.S avec prise en compte des dépendances (ajout de NOPs)

Question 7 :

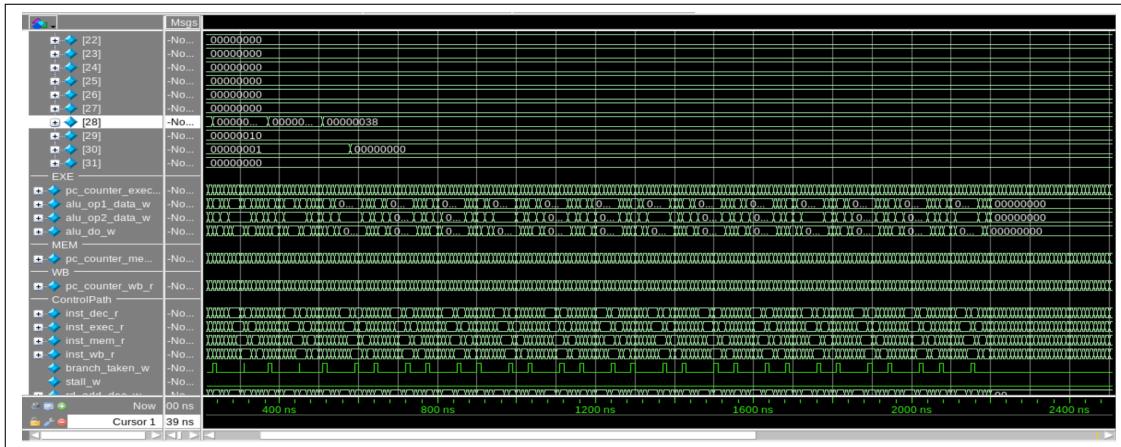


FIGURE 10 – Waveform Mult.S avec prise en compte des dépendances (ajout de NOPs)

On observe qu'à la fin de la simulation, la valeur associée au registre r28 correspondant à t3 est 0x38. Cela correspond bien au résultat attendu de la multiplication.

3.2.2 Correction matérielle : Interlock

Question 8 :

Les signaux *rs1_dec_w* et *rs2_dec_w* correspondent respectivement aux registres sources Rs1 et Rs2. Plus précisément, les bits 15 à 19 de l'instruction sont extraits pour définir Rs1, tandis que les bits 20 à 24 sont utilisés pour définir Rs2.

```

assign rs1_dec_w = instruction_i[19 :15];
assign rs2_dec_w = instruction_i[24 :20];

```

Question 9 :

Le signal *rd_exe_w* représente l'adresse du registre de destination (Rd) pour l'instruction en cours d'exécution dans l'étage EXE. Cette adresse est extraite des bits 7 à 11 de l'instruction :

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

Le signal *rd_mem_w* correspond à l'adresse du registre de destination pour l'instruction à l'étage MEM. Il s'agit de la propagation de l'adresse Rd depuis l'étage précédent (EXE) :

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

Le signal *rd_wb_w* désigne l'adresse du registre de destination dans l'étage WB. Cette adresse est propagée depuis l'étage précédent (MEM) :

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

Question 10 :

Pour générer le signal *stall_w*, il est nécessaire de détecter une dépendance de données entre une instruction en cours dans l'étage ID et une instruction située dans les étages EXE, MEM ou WB.

Un *stall* est déclenché si l'un des registres sources, Rs1 ou Rs2, de l'instruction en ID correspond au registre destination, Rd, d'une instruction présente dans les étages EXE, MEM ou WB.

Lorsque le signal *stall_w* est actif dans le *control_path*, l'instruction en ID est remplacée par une instruction *NOP* :

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

Cette substitution empêche l'instruction en ID d'avancer dans le pipeline tant que la dépendance de données n'est pas résolue.

Question 11 :

Les instructions de type U et J ne lisent pas Rs1.

Question 12 :

Les instructions de type U, J et I ne lisent pas Rs2.

Question 13 :

Les instructions de type B et S ne lisent pas Rd.

Rédaction et logique du signal *stall_w* :

En prenant en compte les réponses aux questions 10, 11, 12 et 13, nous décidons d'introduire des flags (signaux booléens) : *uses_rs1_flag*, *uses_rs2_flag*, et *writes_rd_flag*. Ces flags déterminent si l'instruction en cours lit dans Rs1, lit dans Rs2, ou écrit dans Rd.

Un bloc combinatoire est ajouté pour mettre à jour ces flags en fonction de l'opcode de l'instruction en cours de décodage (*opcode_dec_w*). La liste des opcodes utilisés pour ce traitement est issue du fichier *RV32i_pkg.sv* (cf. Annexes).

```

always_comb begin
    case (opcode_dec_w)
        // R-Type
        RV32I_R_INSTR: begin
            uses_rs1_flag = 1'b1;
            uses_rs2_flag = 1'b1;
            writes_rd_flag = 1'b1;
        end
        // I-Type
        RV32I_I_INSTR_OPER,
        RV32I_I_INSTR_LOAD,
        RV32I_I_INSTR_JALR: begin
            uses_rs1_flag = 1'b1;
            uses_rs2_flag = 1'b0;
            writes_rd_flag = 1'b1;
        end
        // S-Type
        RV32I_S_INSTR: begin
            uses_rs1_flag = 1'b1;
            uses_rs2_flag = 1'b1;
            writes_rd_flag = 1'b0;
        end
        // B-Type
        RV32I_B_INSTR: begin
            uses_rs1_flag = 1'b1;
            uses_rs2_flag = 1'b1;
            writes_rd_flag = 1'b0;
        end
        // U-Type
        RV32I_U_INSTR_LUI,
        RV32I_U_INSTR_AUIPC: begin
            uses_rs1_flag = 1'b0;
            uses_rs2_flag = 1'b0;
            writes_rd_flag = 1'b1;
        end
        // J-Type
        RV32I_J_INSTR: begin
            uses_rs1_flag = 1'b0;
            uses_rs2_flag = 1'b0;
            writes_rd_flag = 1'b1;
        end
        // I-Type Fence/CSR (ne modifient pas de registres)
        RV32I_I_INSTR_FENCE,
        RV32I_I_INSTR_ENVCSR: begin
            // Pas de dépendances sur rs1/rs2/rd
            uses_rs1_flag = 1'b0;
            uses_rs2_flag = 1'b0;
            writes_rd_flag = 1'b0;
        end
        default: begin
            uses_rs1_flag = 1'b0;
            uses_rs2_flag = 1'b0;
            writes_rd_flag = 1'b0;
        end
    endcase
end

```

FIGURE 11 – Bloc combinatoire pour mettre à jour les flags

En utilisant ces flags, le calcul du signal *stall_w* vérifie uniquement les registres nécessaires à l'instruction en cours, ce qui permet de réduire les stalls inutiles.

Le registre x_0 est une constante toujours égale à zéro en RISC-V. Toute lecture de ce registre renvoie 0, et toute tentative d'écriture dans ce registre est ignorée par le processeur.

Par conséquent, x_0 ne doit pas être pris en compte dans les calculs de dépendances pour le signal **stall_w**. Ainsi, le signal **stall_w** est modifié pour intégrer cette particularité, ce qui donne la proposition suivante :

```
assign stall_w = (((rs1_dec_w != 5'b00000) && (uses_rs1_flag && (rs1_dec_w == rd_add_exec_w || rs1_dec_w == rd_add_mem_w || rs1_dec_w == rd_add_sb_w))) || ((rs2_dec_w != 5'b00000) && (uses_rs2_flag && (rs2_dec_w == rd_add_exec_w || rs2_dec_w == rd_add_mem_w || rs2_dec_w == rd_add_sb_w))));
```

FIGURE 12 – Signal *stall_w*.

Question 14 :

Nous avons lancé la simulation après avoir retiré les NOPs liés aux dépendances de données, tout en conservant ceux associés aux dépendances de contrôle. Dans ce contexte, nous nous attendons à ce que le signal **stall_w** s'active afin de remplacer les NOPs précédemment utilisés.

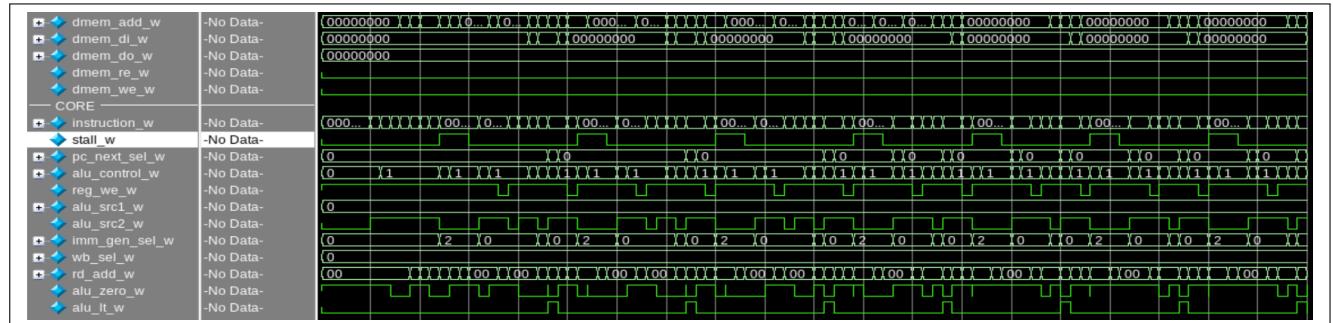


FIGURE 13 – Signal *stall_w* lors de la simulation

On constate ici que le signal **stall_w** s'active correctement tout au long de la simulation.

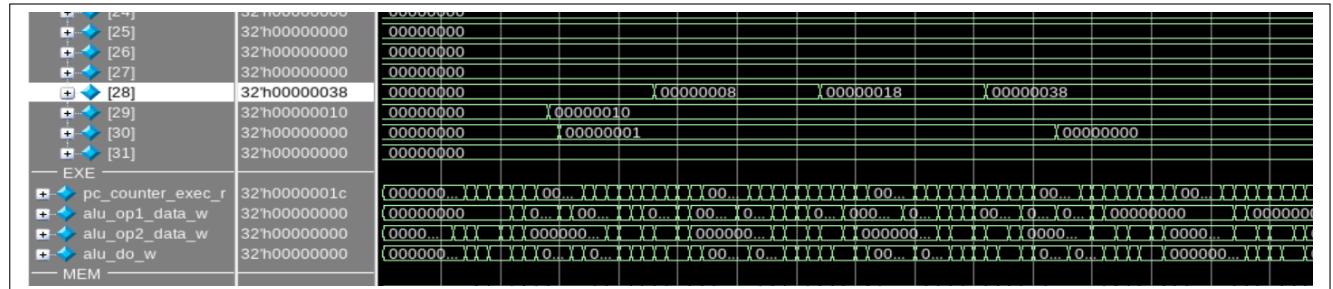


FIGURE 14 – Résultat de la simulation

Le résultat de la simulation correspond bien au résultat attendu de la multiplication, visible dans le registre r28.

Cela confirme l'efficacité du signal **stall_w** défini précédemment.

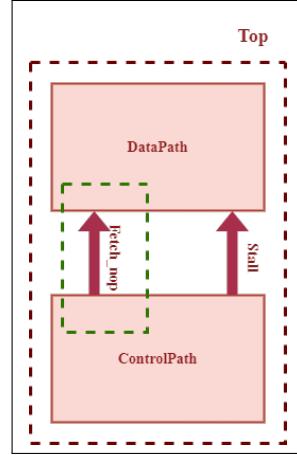
Question 15 :


FIGURE 15 – Top avec gestion de saut.

Lorsqu'un stall est activé, il bloque tout le pipeline, y compris le compteur de programme (PC), pour attendre que les données soient prêtes.

En revanche, pour un saut (dépendances de contrôle de type J), il est crucial de ne pas bloquer le PC, car cela arrêterait tout le pipeline. À la place, on crée un nouveau signal (Fetch_nop) qui insère un NOP dans l'étape Fetch.

La logique derrière l'insertion de ce NOP est que le résultat d'une instruction de type J est récupéré à l'étape Decode. Il est essentiel de "bloquer" l'arrivée de nouvelles instructions dans les étapes précédentes (ici seulement l'étape Fetch), car celles-ci pourraient s'avérer érronnées en fonction du résultat de l'instruction J. Insérer un NOP à cette étape permet au pipeline d'avancer tout en laissant le temps de calculer le résultat de l'instruction J, sans qu'aucune instruction inutile ou erronée ne progresse dans les étapes suivantes.

Ainsi, la gestion des sauts ne requiert pas une modification du signal stall_w.

Question 16 : cf question 18

Question 17 :

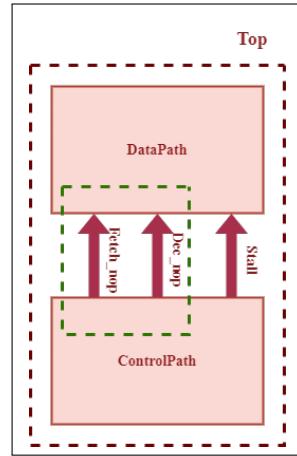


FIGURE 16 – Top avec gestion de branchemet.

Pour les dépendances de contrôle de type B (branchements), comme pour celles de type J, il ne faut pas bloquer l'incrémentation du PC counter. Ainsi, on introduit un nouveau signal (Dec_nop) qui insère un NOP dans l'étape de Decode.

En effet, les instructions de branchement nécessitent une évaluation conditionnelle pour déterminer si un branchement doit être fait ou non. Le résultat de cette évaluation étant obtenu dans l'étape EXE, l'insertion d'un NOP dans les étapes précédentes (IF et ID) empêche l'avancée dans le pipeline, d'instructions qui pourraient s'avérer erronées en cas de branchement.

Ainsi pour les instructions de type B, on active les signaux Fetch_nop et Dec_nop afin de bloquer l'avancée de nouvelles instructions dans le pipeline avant l'obtention du résultat de l'instruction de branchement.

```

always_ff @(posedge clk_i or negedge resetn_i) begin : exec_stage
  if (resetn_i == 1'b0)
    begin
      alu_op1_data_r <= 0;
      alu_op2_data_r <= 0;
      rs2_data_r <= 0;
      func3_exec_r <= 0;
      pc_br_target_r <= 0;
      pc_counter_exec_r <= 0;
    end
  if ( dec_nop_i == 1'b1) //NNNNNNNNNNNEEEEEEEEEEEEEEEEEE
    begin
      alu_op1_data_r <= 0;
      alu_op2_data_r <= 0;
      rs2_data_r <= 0;
      func3_exec_r <= 0;
      pc_br_target_r <= pc_br_target_w;
      pc_counter_exec_r <= pc_counter_dec_r;
    end
  else
    begin
      alu_op1_data_r <= alu_op1_data_w;
      alu_op2_data_r <= alu_op2_data_w;
      rs2_data_r <= rs2_data_w;
      func3_exec_r <= func3_dec_r;
      pc_br_target_r <= pc_br_target_w;
      pc_counter_exec_r <= pc_counter_dec_r;
    end
end

```

FIGURE 17 – Gestion des dépendances de contrôle. *Datopath*

```
assign inst_w = (fetch_nop_i == 1'b0) ? imem_data_i : 32'h000000013;
```

FIGURE 18 – Gestion des dépendances de contrôle. *Datapath*

```
assign fetch_nop_o = branch_taken_w || jump_taken_w ;
assign dec_nop_o = branch_taken_w;
```

FIGURE 19 – Gestion des dépendances de contrôle. *Controlpath*

Question 18 :



FIGURE 20 – Simulation complète

On constate que le résultat obtenu en fin de simulation correspond bien au résultat de la multiplication effectuée. Cela implique que les dépendances ont été corrigées et donc que linstanciation des signaux Fetch_nop et Dec_nop est correcte.

On observe dailleurs bien sur la capture quils sactivent correctement tout au long de la simulation.

3.3 Correction matérielle : Bypass

Question 19 :

Le bypass gère les dépendances de données entre les instructions, permettant à une instruction suivante de lire une valeur produite par une instruction précédente sans attendre létape de Write-Back.

Question 20 :

```
//AVEC BYPASSSSSSSSSSSS
  always_comb begin : alu_src1_mux_comb
    case (alu_src1_i)
      SEL_OPL_RS1: alu_op1_data_w = rsl_data_w;
      SEL_OPL_IMM: alu_op1_data_w = imm_w;
      SEL_OPL_PC: alu_op1_data_w = pc_counter_r;
      SEL_OPL_BYPASS: alu_op1_data_w = alu_do_w;
      default: alu_op1_data_w = 0;
    endcase
  end
```

FIGURE 21 – Modification de *alu_src1_mux_comb*

Question 21 :

Oui, limplémentation du bypass a un impact sur le signal stall_w. Le bypass permet de résoudre certaines dépendances de données sans nécessiter linsertion de stalls, ce qui réduit les cas où stall_w doit être activé.

```
//stall avec bypass
assign stall_w =((rs1_dec_w != 5'b00000) && (uses_rs1_flag && ((rs1_dec_w == rd_add_mem_w) || (rs1_dec_w == rd_add_wb_w)))|||((rs2_dec_w != 5'b00000) && (uses_rs2_flag && ((rs2_dec_w == rd_add_exec_w) || (rs2_dec_w == rd_add_mem_w) || (rs2_dec_w == rd_add_wb_w))));
```

FIGURE 22 – Modification du stall lié à l’implémentation du bypass

Comme vu en cours, on sait que le bypass permet de fournir la donnée rs1 depuis l’étage d’Execution directement à l’étage de Décode. Ainsi, l’implémentation d’un bypass supprime la dépendance de données entre $rs1_dec_w$ et $rd_add_exec_w$ ce qui permet de supprimer l’activation du stall à ce niveau.

Question 22 :

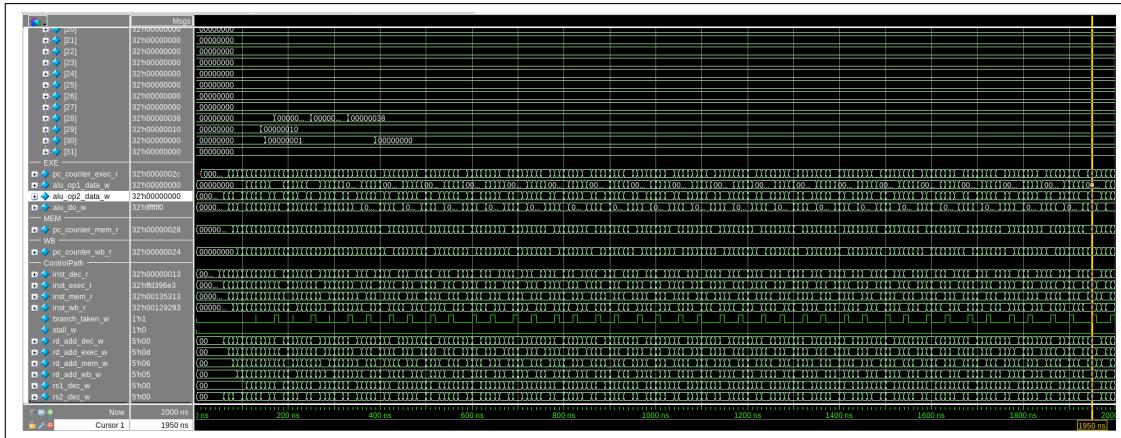


FIGURE 23 – Simulation avec bypass

On constate sur la simulation qu’on obtient bien le résultat attendu. La multiplication s’effectue correctement avec l’implémentation du bypass.

On constate par ailleurs que le résultat de la multiplication est obtenu plus rapidement que dans les simulations sans le bypass (différence d’environ 60ns).

Question 23 :

Dans ce TP nous avons mis en oeuvre plusieurs méthodes permettant de régler les conflits et les dépendances de données et d’instructions.

Dans la première partie nous avons utilisé une correction logicielle en insérant des NOPs dans notre programme. Cette méthode, bien que facile à mettre en place, réduit l’efficacité du pipeline en y ajoutant un grand nombre de cycles. De plus, cette méthode n’est pas optimisée car même lorsque les données sont obtenues en avance, le pipeline reste bloqué jusqu’à ce que tous les cycles NOP soient écoulés.

Dans la deuxième partie, nous avons mis en place une correction matérielle avec la création d’un signal Stall. Ce signal insère un NOP dans l’étape ID dès qu’une dépendance de données est détectée. Bien que cette méthode nécessite une logique matérielle supplémentaire pour la détection et la gestion de conflits, elle permet d’optimiser l’utilisation des NOPs et permet de réduire la durée pendant laquelle l’avancée du pipeline est bloquée. La perte de cycle est réduite par rapport à la correction logicielle.

Dans la dernière partie, nous avons exploité l’implémentation d’un bypass dans le pipeline. Le bypass permet la lecture d’une valeur produite par une instruction précédente avant qu’elle ne soit inscrite dans la mémoire. Il permet de réduire certaines dépendances de données limitant l’utilisation du Stall. Bien que nécessitant une logique complexe, le bypass minimise les interruptions du pipeline et accélère les opérations.

Finalement, le bypass semble être la méthode la plus fructueuse, optimisant et accélérant le traitement de données dans un pipeline. Il aurait été intéressant d’ajouter des bypass entre d’autres étapes du pipeline, ce qui aurait conduit à une accélération encore plus importante du traitement de données.

4-TP 3

Un cache est une petite mémoire dont la fonction est de stocker des données fréquemment utilisées afin de faciliter leur accès lors de l'exécution des instructions.

L'objectif de ce TP était d'implémenter différents types de cache : d'abord un cache d'instruction direct, puis un cache d'instruction associatif à deux voies.

4.1 Cache mémoire "Direct"

Taille de la ligne de cache :

Selon le découpage des adresses (TAG, INDEX, OFFSET), la partie OFFSET est utilisée pour adresser les octets à l'intérieur d'une ligne de cache. On a $b = 3 - 0 + 1 = 4$. $K = 9 - 4 + 1 = 6$. $T = 31 - 10 + 1 = 22$.

Taille de ligne : $2^b = 2^4 = 16$ mots = 128 bits.

Nombre d'entrées dans la mémoire cache :

INDEX $k = 6$ bits indique le nombre de lignes dans la mémoire cache, soit $2^k = 2^6 = 64$ lignes.

Avec 16 octets par ligne, la mémoire cache contient donc $64 \times 16 = 1024$ octets.

4.2 Performance

Taux de MISS :

Taux de MISS = 1 - HIT rate = 1 - 0.9 = 0.1 (10%).

Temps d'accès moyen :

Temps d'accès moyen = HIT time + (MISS rate × MISS penalty)

HIT time = 1 cycle, MISS penalty = 10 cycles.

Temps d'accès moyen = $1 + (0.1 \times 10) = 1 + 1 = 2$ cycles.

La mémoire n'étant pas très grande, la traversée du multiplexeur pour accéder au cache est très rapide, ce qui permet un accès en un seul cycle.

4.3 Cache instruction direct

Question 0 :

On représente le RV32i_soc ainsi :

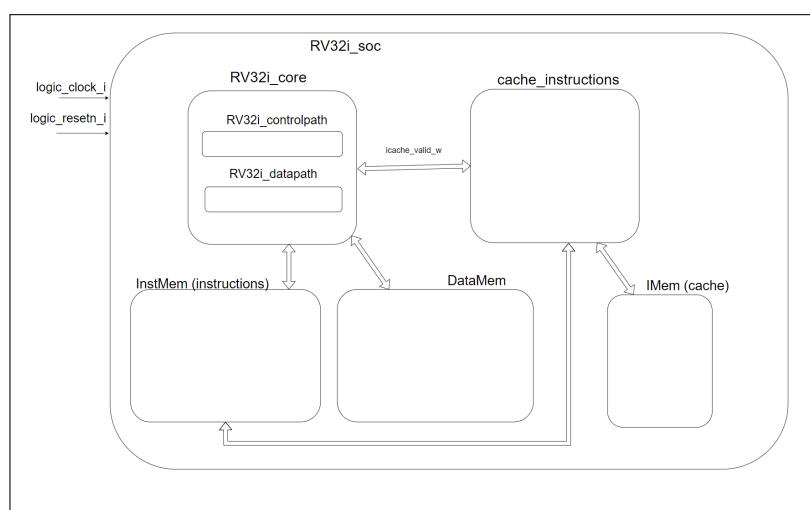


FIGURE 24 – RV32i_soc avec mémoire cache

Le signal de données transférées entre Imem et le cache d'instruction est défini ainsi :

logic [3 :0]/[31 :0] imem_cache_data_w ;

Ceci correspond à 4 blocs de 32 bits, soit 128 bits.

Ainsi, le bus de données entre Imem et le cache d'instruction a une largeur de 128 bits.

Dans *RV32i_top.sv*, le signal *imem_valid_i* indique si les données provenant de la mémoire d'instructions sont valides et prêtes à être utilisées par le processeur.

Question 1 : cf question 10

Question 2 :

Le cache est constitué de trois éléments : les instructions, les tags et les bits de validité.

Dans la mémoire cache, on stocke uniquement les instructions. Il est en effet essentiel de les y placer, car elles sont souvent temporaires et nécessitent une exécution rapide. Les stocker dans la cache est donc judicieux, car il s'agit de la mémoire offrant le temps d'accès le plus court.

Chaque instruction occupant 32 bits, il est possible d'en sauvegarder 4 par ligne de cache (128 bits).

Le Tag (22 bits) et le bit de validité (1 bit) ne sont pas stockés directement dans la mémoire cache, ce sont des métadonnées.

Question 3 : cf question 10

Question 4 : cf question 10

Question 5 :

La requête est *hit* si la ligne de cache est valide et si le tag stocké dans la ligne correspond au tag de l'adresse. Sinon, on dit que la requête est *miss*.

Question 6 :

Si la requête est hit, alors la donnée est lue depuis le cache. On récupère la donnée ainsi : *cache_data [index]/[offset]*. On vérifie ensuite que la donnée renvoyée est bien valide.

Question 7 :

Si la requête est miss, on lit la donnée depuis la mémoire : *mem_read_data_i [offset*32 + : 32]*.

On vérifie si la donnée renvoyée est bien valide.

On prend aussi soin d'aligner l'accès à la mémoire sur la taille de la ligne avec la commande *assign mem_addr_o = {tag, index, ByteOffsetBits1'b0}*.

Question 9 :

Avant de transmettre le résultat dans la mémoire, on commence par réinitialiser le cache. La transmission du résultat ne peut se faire qu'à condition que la mémoire ne réponde (*mem_read_valid_i ==1*). Si cette condition est vérifiée, on met à jour le cache avec la nouvelle ligne.

Question 10 :

Finalement, le programme *cache_module.sv* se présente ainsi.

```

1 module cache #(
2     localparam ByteOffsetBits = 4,
3     localparam IndexBits = 6,
4     localparam TagBits = 22,
```

```

5
6     localparam NrWordsPerLine = 4,
7     localparam NrLines = 64,
8
9     localparam LineSize = 32 * NrWordsPerLine
10    ) (
11        input logic clk_i,
12        input logic rstn_i,
13
14        input logic [31:0] addr_i,
15
16        // Read port
17        input logic read_en_i,
18        output logic read_valid_o,
19        output logic [31:0] read_word_o,
20
21        // Memory
22        output logic [31:0] mem_addr_o,
23
24        // Memory read port
25        output logic mem_read_en_o,
26        input logic mem_read_valid_i,
27        input logic [LineSize-1:0] mem_read_data_i
28    );
29
30 // **REGISTRES POUR LE CACHE**
31 // Tableau pour stocker les tags
32 logic [NrWordsPerLine-1:0][31:0] cache_data [NrLines]; //donnees de chaque ligne
33 logic [TagBits-1:0] cache_tag [NrLines]; //tags de chaque ligne
34 logic cache_valid [NrLines]; //bits de validite
35 //decoupage de l'adresse addr_i
36 logic [TagBits-1:0] tag; //MSB
37 logic [IndexBits-1:0] index; //bits du milieu pour select la ligne de cache.
38 logic [ByteOffsetBits-1:0] offset; //LSB pour select le mot dans la ligne.
39
40 assign tag = addr_i[31:32-TagBits];
41 assign index = addr_i[32-TagBits-1:ByteOffsetBits];
42 assign offset = addr_i[ByteOffsetBits-1:0];
43
44 logic hit;
45 assign hit = cache_valid[index] && (cache_tag[index] == tag);
46 // si c'est un hit on renvoie le mot correspondant a l'offset dans la ligne de cache.
47 assign read_word_o = hit ? cache_data[index][offset] : mem_read_data_i[offset*32
+ 32]; //Si HIT alors la donnee est lue depuis le cache, si MISS la donnee
est lue depuis la m moire.
48 assign read_valid_o = (hit && read_en_i) || mem_read_valid_i; //determine si la
donnee renvoyee est valide.
49 //requete a la memoire si on a un MISS
50 assign mem_addr_o = {tag, index, {ByteOffsetBits{1'b0}}}; //adresse alignee sur
la ligne
51 assign mem_read_en_o = !hit && read_en_i; //activation de la lecture de memoire
52
53 //Stockage du resultat dans la memoire du cache
54 always_ff @(posedge clk_i or negedge rstn_i) begin
55     if (!rstn_i) begin //reinitialisation du cache
56         for (int i = 0; i < NrLines; i++) begin
57             cache_valid[i] <= 1'b0;

```

```

58     end
59   end else if (mem_read_valid_i) begin //maj du cache avec la nouvelle
60     ligne
61     cache_data[index] <= mem_read_data_i;
62     cache_tag[index] <= tag;
63     cache_valid[index] <= 1'b1;
64   end
65 end
66 endmodule

```

Listing 1 – Cache direct en lecture seule

4.4 Cache instruction associatif à deux voies

Question 11 :

On déclare la deuxième voie du cache de la même manière que la ligne de cache dans la partie précédente, tout en prenant soin de numérotter les deux lignes.

La mise en place d'une priorité de type LRU nécessite l'instanciation d'une variable qui prendra la valeur 0 ou 1, selon la ligne de cache qui a été modifiée le plus récemment.

Question 12 :

Pour qu'une requête soit un hit, il est nécessaire de vérifier les deux voies du cache, car elle peut se trouver dans l'une ou l'autre. Ainsi, on modifie le code en effectuant la vérification d'abord sur une voie, puis sur l'autre. Le signal hit sera alors vrai si le hit de la voie 0 ou celui de la voie 1 est détecté.

Question 13 : cf question 16

Question 14 : cf question 16

Question 15 : cf question 16

Question 16 :

Finalement, l'implémentation d'un cache instruction associatif à deux voies se présente ainsi :

```

1 module cache #(
2   localparam ByteOffsetBits = 4,
3   localparam IndexBits = 6,
4   localparam TagBits = 22,
5
6   localparam NrWordsPerLine = 4,
7   localparam NrLines = 64,
8
9   localparam LineSize = 32 * NrWordsPerLine
10 ) (
11   input logic clk_i, //horloge
12   input logic rstn_i, //reset
13
14   input logic [31:0] addr_i, //adresse de la memoire a laquelle on doit acceder
15
16   // Read port
17   input logic read_en_i, //signal de lecture

```

```

18   output logic read_valid_o,
19   output logic [31:0] read_word_o,
20
21   // Memory
22   output logic [31:0] mem_addr_o,
23
24   // Memory read port
25   output logic mem_read_en_o,
26   input logic mem_read_valid_i, //reponse de la memoire si MISS
27   input logic [LineSize-1:0] mem_read_data_i //reponse de la memoire si MISS
28 );
29
30   // ***REGISTRES POUR LE CACHE**
31   logic [NrWordsPerLine-1:0][31:0] cache_data_way0 [NrLines];
32   logic [NrWordsPerLine-1:0][31:0] cache_data_way1 [NrLines];
33   logic [TagBits-1:0] cache_tag_way0 [NrLines];
34   logic [TagBits-1:0] cache_tag_way1 [NrLines];
35   logic cache_valid_way0 [NrLines];
36   logic cache_valid_way1 [NrLines];
37   logic lru [NrLines]; // 0 = voie 0 est LRU, 1 = voie 1 est LRU
38
39   // Decoupage de l'adresse
40   logic [TagBits-1:0] tag;
41   logic [IndexBits-1:0] index;
42   logic [ByteOffsetBits-1:0] offset;
43
44   assign tag = addr_i[31:32-TagBits];
45   assign index = addr_i[32-TagBits-1:ByteOffsetBits];
46   assign offset = addr_i[ByteOffsetBits-1:0];
47
48   // Signaux de hit pour chaque voie
49   logic hit_way0, hit_way1;
50   assign hit_way0 = cache_valid_way0[index] && (cache_tag_way0[index] == tag);
51   assign hit_way1 = cache_valid_way1[index] && (cache_tag_way1[index] == tag);
52
53   // Signal global de hit
54   logic hit;
55   assign hit = hit_way0 || hit_way1;
56
57   // Selection de la voie a lire en cas de hit
58   assign read_word_o = (hit_way0) ? cache_data_way0[index][offset] :
59     (hit_way1) ? cache_data_way1[index][offset] :
60       mem_read_data_i[offset*32 +: 32]; // En cas de miss, lire
61       depuis la memoire
62
63   // Requete a la memoire en cas de miss
64   assign mem_addr_o = {tag, index, {ByteOffsetBits{1'b0}}};
65   assign mem_read_en_o = !hit && read_en_i;
66
67   // Determiner la voie evicter en cas de miss
68   logic way_to_replace;
69   assign way_to_replace = lru[index]; // 0 = voie 0, 1 = voie 1
70
71   // Mise a jour du cache et du LRU
72   always_ff @(posedge clk_i or negedge rstn_i) begin
73     if (!rstn_i) begin
74       // Reinitialisation du cache et du LRU

```

```

74      for (int i = 0; i < NrLines; i++) begin
75          cache_valid_way0[i] <= 1'b0;
76          cache_valid_way1[i] <= 1'b0;
77          lru[i] <= 1'b0; // Initialement, la voie 0 est LRU
78      end
79  end else if (mem_read_valid_i) begin
80      // Mise a jour du cache avec la nouvelle ligne
81      if (way_to_replace == 1'b0) begin
82          cache_data_way0[index] <= mem_read_data_i;
83          cache_tag_way0[index] <= tag;
84          cache_valid_way0[index] <= 1'b1;
85      end else begin
86          cache_data_way1[index] <= mem_read_data_i;
87          cache_tag_way1[index] <= tag;
88          cache_valid_way1[index] <= 1'b1;
89      end
90      // Mise a jour du LRU
91      lru[index] <= ~way_to_replace; // La voie remplacée devient la plus
92      // récemment utilisée (~ renvoie l'inverse)
93  end else if (hit) begin
94      // Mise a jour du LRU en cas de hit
95      if (hit_way0) begin
96          lru[index] <= 1'b1; // La voie 1 devient LRU
97      end else if (hit_way1) begin
98          lru[index] <= 1'b0; // La voie 0 devient LRU
99      end
100 end
101
102 // Signal de validité de la donnée lue
103 assign read_valid_o = (hit && read_en_i) || mem_read_valid_i;
104
105 endmodule

```

Listing 2 – Cache instruction associatif à deux voies

5-Conclusion

L'ensemble de ces TP nous aura permis, dans un premier temps, de comprendre le fonctionnement du pipeline à 5 étapes d'un processeur RISC-V32, puis de le manipuler afin d'améliorer son efficacité.

Après avoir identifié les potentiels conflits et dépendances rencontrés le long d'un pipeline à l'issue du TP1, nous avons cherché à les résoudre.

Ainsi, dans le TP2, nous avons mis en place différentes solutions pour gérer ces dépendances de données et de contrôle. Après avoir constaté les limites de la correction logicielle, nous avons introduit un stall avant d'implémenter un bypass. Cela a mis en évidence les avantages d'une correction matérielle, en particulier l'efficacité du bypass pour résoudre les problèmes sur un pipeline et accélérer le traitement des données.

Enfin, dans le TP3, nous avons travaillé sur des caches, en implémentant deux versions dans le but de réduire le temps d'accès à la mémoire et d'améliorer les performances globales du pipeline.

Ces travaux pratiques nous ont permis d'acquérir une compréhension approfondie des mécanismes d'optimisation des processeurs modernes et des outils pour les concevoir de manière plus efficace.

6-Annexes

```

// RV32I opcodes
// R instruction opcode
const logic [6 : 0] RV32I_R_INSTR = 7'b0110011;

// I instruction opcode
const logic [6 : 0] RV32I_I_INSTR_JALR = 7'b1100111;
const logic [6 : 0] RV32I_I_INSTR_LOAD = 7'b0000011;
const logic [6 : 0] RV32I_I_INSTR_OPER = 7'b0010011;
const logic [6 : 0] RV32I_I_INSTR_FENCE = 7'b0001111;
const logic [6 : 0] RV32I_I_INSTR_ENVCSR = 7'b1110011;

// S instruction opcode
const logic [6 : 0] RV32I_S_INSTR = 7'b0100011;

// B instruction opcode
const logic [6 : 0] RV32I_B_INSTR = 7'b1100011;

// U instruction opcode
const logic [6 : 0] RV32I_U_INSTR_LUI = 7'b0110111;
const logic [6 : 0] RV32I_U_INSTR_AUIPC = 7'b0010111;

// J instruction opcode
const logic [6 : 0] RV32I_J_INSTR = 7'b1101111;

```

FIGURE 25 – Opcodes (fichier RV32i_pkg.sv).