École nationale supérieure des mines de Saint-Étienne

GP - Conception de Systèmes Electroniques 2
Architecture des processeurs 2

Rapport : RV32I architecture pipeline

Élèves :
Ismahane Erramidi
Jade Melliti

Enseignant :
Marc Lacruche

17 janvier 2025

Table des matières

1 TP n°1 : RV32I architecture pipeline **2**
1.1 RV32I architecture pipeline . . . 2
1.1.1 Examinez le sous circuit data_path . . . 3
1.1.2 Examinez le sous circuit control_path . . . 4
1.1.3 Examinez l'instance dmem du composant wsync_mem . . . 4
1.1.4 Examinez l'instance imem du composant wsync_mem . . . 5
1.2 Éxécution et simulation d'un programme . . . 5

2 TP n°2 : Gestion des dépendances **9**
2.1 Éxécution d'un programme . . . 9
2.2 Correction des problèmes . . . 11
2.2.1 Dépendances de données . . . 14
2.2.2 Dépendances de contrôle . . . 15
2.3 Optimisation en utlisant le bypass . . . 17
2.3.1 Pipeline sans Bypass . . . 19
2.3.2 Pipeline avec Bypass . . . 20

3 TD n°3 : Implémentation d'une mémoire cache **21**
3.1 Cache mémoire "direct" . . . 22
3.2 Cache instruction associatif à deux voies . . . 25

1 TP n°1 : RV32I architecture pipeline

1.1 RV32I architecture pipeline

Question 1 : Identifier le circuit top-level et donner la liste des sous-circuits instanciés dans le top-level ?

Le circuit top-level est représenté par le fichier RV32i_pipeline_soc.sv. Le circuit top-level est le cœur du CPU en lui-même. Dans le fichier, on retrouve des sous-circuits à savoir la mémoire d'instruction (imem) et la mémoire de donnée (data memory).

Question 2 : Quel est le rôle de chacun de ces sous-circuits ?

La mémoire d'instruction et la mémoire de données jouent des rôles distincts mais complémentaires.

La mémoire d'instruction est utilisée pour stocker les instructions que le processeur doit exécuter. La mémoire de données, quant à elle, est utilisée pour stocker les données sur lesquelles les instructions vont opérer. Le processeur utilise la mémoire d'instruction pour savoir quelles opérations effectuer et la mémoire de données pour obtenir les valeurs nécessaires à ces opérations.

Question 3 : De quels circuits est composé le RV32I_top ? Remplissez le schéma de la figure 1

Le RV32I_top est composé des sous-circuits suivants :
- data_path
- control_path
- imem
- dmem

On peut représenter le RV32I_top comme cela :

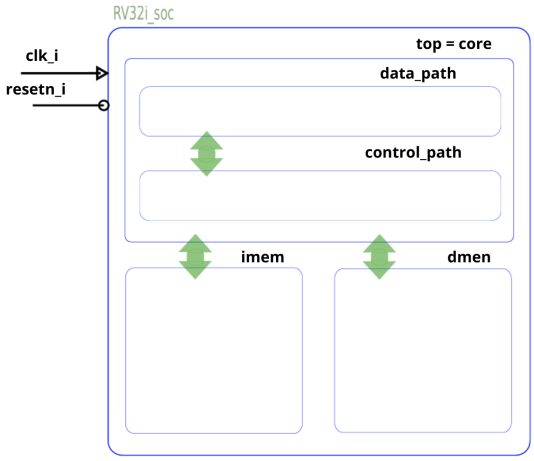


FIGURE 1 – Schéma représentatif de RV32I_top

Le control Path, aussi appelé chemin de contrôle, est responsable de générer les signaux de contrôle nécessaires pour orchestrer le fonctionnement global du processeur. Il s'assure que chaque composant fonctionne de manière synchronisée en fonction des instructions. Ainsi, ce circuit dirige tous les autres, d'où sa présence en haut du schéma.

Le data Path, aussi appelé chemin des données, gère le cheminement des données dans le processeur, en exécutant les opérations sur les registres et en connectant les différents composants, comme l'ALU, les registres, et les mémoires. Ainsi, il est placé au centre, car il connecte et transporte les données entre imem, dmem, et les unités fonctionnelles.

Donc, imem et dmem sont sur les blocs à gauche et à droite, correspondant respectivement à la mémoire des instructions et des données.

1.1.1 Examinez le sous circuit data_path

Question 4 : De combien d'étages est composé le coeur RISC-V ? Nommez les et donnez leur rôle !

Le cœur RISC-V pipeliné est composé de cinq étages :

1. IF (Instruction Fetch) : Récupère l'instruction suivante de la mémoire d'instructions.
2. ID (Instruction Decode) : Décode l'instruction récupérée.
3. EX (Execute) : Exécute l'instruction décodée.
4. MEM (Memory) : Accède à la mémoire de données si nécessaire.
5. WB (Write Back) : Écrit le résultat de l'instruction dans le registre de destination.

Question 5 : Quels sont les signaux à destination du controlpath ?

Les signaux qui sont à destination du control_path sont les suivants :
- clk_i,
- resetn_i,
- dmem_do_i

Question 6 : Quels sont les signaux à destination de la mémoire d'instructions ?

Les siganux à destination de la mémoire d'instructions :
- clock
- we
- ble_i : contrôle la taille des données à lire
- d_i : instruction en mémoire
- imen_add_w
- imen_data_w

Question 7 : Quels sont les signaux à destination de la mémoire de données ?

Les signaux à destinations de la mémoire de données :
- clock

— we : `dmem_we_w`
— re : `dmem_re_w`
— ble_i
— d_i : `dmem_di_w`
— adresse : `dmem_add_w[13:2]`
— donnée : `dmem_do_w`

1.1.2 Examinez le sous circuit control_path

Question 8 : Que représente les signaux : inst_dec_r inst_exec_r inst_mem_r inst_wb_r ?

Ce sont des signaux internes sur 4 bits qui contrôlent chaque étage du coeur.
— inst_dec_r : Registre de pipeline entre les étages IF et ID.
— inst_exec_r : Registre de pipeline entre les étages ID et EX.
— inst_mem_r : Registre de pipeline entre les étages EX et MEM.
— inst_wb_r : Registre de pipeline entre les étages MEM et WB

Question 9 : Suivez le chemin de l'index du registre de destination depuis l'étage de décode jusqu'au banc de registre. Commentez

Lorsqu'on suit le chemin de l'index du registre de destination depuis l'étage de décode, jusqu'au banc de registre, on se rend compte qu'une partie va dans Rs1_add_i et Rs2_add_i, c'est-à-dire les adresse des registres sources, puis une autre partie dans imm_gen. Cela est totalement logique lorsqu'on vérifie le tableau avec la liste des instructions et la manière dont les instructions se déroulent.

Question 10 : Comment est actuellement généré le signal stall_w ?

Le signal stall_w est généré par le control_path. Le stall est initialisé nul à la fin du code. En effet, pour l'intant, nous nous plaçons dans le cas où il n'y a aucune dépendance ou encore aucune raison pour faire une pause dans l'exécution.

Question 11 : Que se passe-t-il quand le signal stall_w est actif dans le control_path ?

On retrouve cette ligne de code :

```
inst\_dec\_r = (stall\_w == 1'b1) ? 32'h00000013 : instruction\_i;
```

Ainsi, si le stall est actif, l'instruction prend la valeur 32'h00000013, ce qui correspond à un NOP.
Ainsi, lorsque le signal stall_w est actif, le pipeline est arrêté, ce qui signifie que les instructions ne progressent pas vers les étages suivants. Cela permet de résoudre les dépendances ou les conditions qui ont causé l'arrêt.

1.1.3 Examinez l'instance dmem du composant wsync_mem

Question 12 : Quelle taille a la mémoire de données ?

La mémoire de données corresponds à un tableau de 4096, avec des données de 32 bits.

Question 13 : Quelle est son adresse de base ?

Dans le fichier RV32I_pipeline_soc.sv, on retrouve l'adresse initiale de la mémoire de données qui vaut 10000.

1.1.4 Examinez l'instance imem du composant wsync_mem

Question 14 : Quelle taille a la mémoire de données ?

La mémoire de données corresponds à un tableau de 4096, avec des données de 32bits.

Question 15 : Quelle est son adresse de base ?

Dans le fichier RV32I_pipeline_soc.sv, on retrouve l'adresse initiale de la mémoire de données qui vaut 10000.

1.2 Éxécution et simulation d'un programme

Question 16 : En examinant le programme exo1.S et le contenu des registres en fin de simulation, le programme s'est il déroulé correctement ?

Le programme que nous simulons est le suivant :

```
1  .section .start;
2  .globl start;
3
4  start:
5      li t0, 1
6      li t1, 2
7      li t2, 3
8      li t3, 4
9      li t4, 5
10     li t5, 6
11     li t6, 7
12
13 lab1:
14     j lab1
15     nop
16
17 .end start
```

Il est important de comprendre le code proposé avant de simuler pour savoir si le programme s'est correctement déroulé. Les registres à regarder, sont les registres 5,6,7 et 28,29,30,31. En effet, les t0...t6 correspondent aux registres énoncés précédents. En effet, le tableau suivant nous donne la correspondance entre ti et le registre associé.

N° de registre	Nom	Description
0	zero	Registre cablé à 0
1	ra	Adresse de retour
2	sp	Pointeur de pile
3	gp	Pointeur global
4	tp	Pointeur de "Thread"
5...7	t0...t2	Registres temporaires
8	s0/fp	Registre à sauvergarder N°0, pointeur de "Frame"
9	s1	Registre à sauvergarder N°1
10...11	a0...a1	2 premiers paramètres, valeurs de retour
12...17	a2...a7	Paramètres
18...27	s2...s7	Registres à sauvegarder
28...31	t3...t6	Registres temporaires

FIGURE 2 – Utilisation des registres

Ainsi, on comprends que le programme que l'on souhaite exécuter est censé mettre le registre R5 à 1, le registre R6 à 2, le registre R7 à 3, le registre R28 à 4, le registre R29 à 5, le registre R30 à 6 et le registre R31 à 7.
En simulant, on obtient :

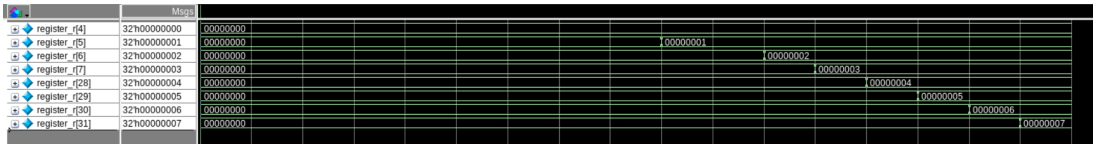


FIGURE 3 – Simulation

Ainsi, on peut conclure que le programme se déroule correctement.

Question 17 : Comment s'efectue l'arrêt de la simulation ?

On peut remarquerà la fin du programme la présence d'un jump. Ainsi, il va réaliser un jump avec un offset de 0, c'est-à-dire sauter sur lui-même. on va alors entrer dans une boucle infini. Il va le faire 5 fois et puis, il va s'arrêter.

Question 18 : Ecrire le fichier main.S qui effectue les opérations élémentaires décrit dans l'algorithme 1.

L'algorithme 1 est le suivant :

Algorithm 1 Fonctions élémentaires à assembler

1: $t0 = 0x3$
2: $t1 = 0x8$
3: $t2 = t1 + t0$
4: $t3 = 0x10$
5: $t4 = 0x11$
6: $t5 = t3 - t4$

FIGURE 4 – Algorithme 1

On écrit cet algortihme dans le fichier main.S. Le fichier main.S devient :

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

Question 19 : Lancez la simulation selon la procédure décrite précédemment. Quel est le résultat obtenu ? Pourquoi ?

Nous avons donc lancer la simulation sans vérifier au préalable les problèmes de conflits. On peut remarquer qu'il y a des conflits dans l'écriture de notre algorithme. En effet, dans la ligne 3, nous appelons t1 sans avoir vérifié au préalable que t1 est disponible. Or, comme nous avons pu le voir pendant ce tp, nous cinq étages qui s'éxécutent sur chaque coup de clock. Or, dans l'algorithme on ne prends pas en compte ce problème.
Il y a un autre conflit dans notre algorithme. En effet, à la ligne 6, on appelle t3 et t4. Or, nos registres n'ont pas eu le temps d'être enregistré dans le WB. Ainsi, il y a un conflit.

Afin de résoudre ce problème, il est nécéssaire d'ajouter des nope à chaque fois qu'on a un conflit. Afin de savoir, le nombre précis de nope à ajouter. Deux possibilités s'offrent à nous. On peut soit tester de manière aléatoire et voir quand le problème est résolu, ou alors on peut faire un schéma pour savoir le nombre exact de nopes nécéssaires. Nous avons réalisé un schéma, que l'on retrouve sur la figure suivante :

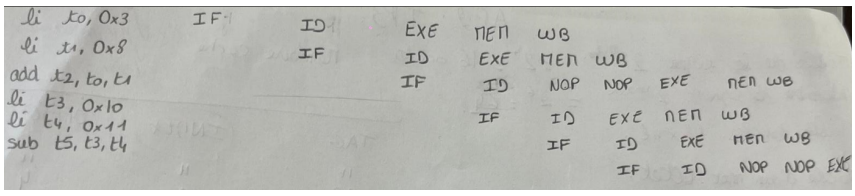


FIGURE 5 – Résolution des problèmes

Ainsi, comme on peut le voir dans le tableau, il faut deux nope entre chaque instruction pour obtenir la bonne solution. Ainsi, nous avons réécrit le programme avec les nopes placés au bon endroit.

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

2 TP n°2 : Gestion des dépendances

Dans une microarchitecture de type pipeline, il peut avoir de nombreuses dépendances. Une dépendance est générée lorsque par exemple, une instruction dépend du résultat d'une instruction précédente, elle ne peut pas être exécutée tant que la dépendance n'est pas gérée. Ainsi, dans ce travail pratique nous allons nous intéresser à la gestion des dépendances. Pour ce faire, nous allons exécuter un programme avec des dépendances. Ensuite, nous allons corriger les problèmes rencontrés. Enfin, nous allons essayer d'optimiser la correction.

2.1 Éxécution d'un programme

Le programme que nous allons mettre en place, permet de réaliser une multiplication non-signée de 2 mots de 16 bits. Le résultat est alors non-signé sur 32 bits.
Afin de réaliser une multiplication, dans ces conditions et dans un processus sequentiel, alors il faut effectuer une boucle de 16 itérations. A chaque itération, il faut accumuler l'opérande1 si le bit de poids faible de l'opérande2 est égale à '1'. De plus, il faut à chaque itération, décaler l'opérande1 à gauche et l'opérande2 à droite.
On peut retrouver l'algorithme traduisant une multiplication entre 8 et 7, sur la figure suivante :

Algorithm 1 Multiplication logicielle

1: $t0 = 0x8$
2: $t1 = 0x7$
3: **for** $t2 = 0 to 15$ **do**
4: **if** $t1[0] == 1'b1$ **then**
5: $t3 = t3 + t0$
6: **end if**
7: $t0 = t0 << 1$
8: $t1 = t1 >> 1$
9: **end for**

FIGURE 6 – Résolution des problèmes

Ensuite, nous avons alors traduit cet algorithme en code. Nous retrouvons le code suivant :

```
1   .section .start;
2   .globl start;
3
4   start:
5       li t0, 0x8
6       li t1, 0x7
7       li t2, 0x15
8       li t3, 0
9
10      loop :
11          beqz t2, .end
12          andi t4, t1, 1
13          beqz t4, .test
14          add t3, t3, t0
15      .test :
16          slli t0, t0, 1
17          srli t1, t1, 1
18          addi t2, t2, -1
19          j loop
20      .end :
21          nop
22
23  lab: j lab
24      nop
25
26  .end start
```

Le résultat normalement attendu du registre t3 est le résultat de la multiplication entre t0 et t1, à savoir entre 8 et 7. On est censé obtenir 56. Ainsi, nous pouvons réaliser la simulation. Lorsque nous simulons, nous obtenons :

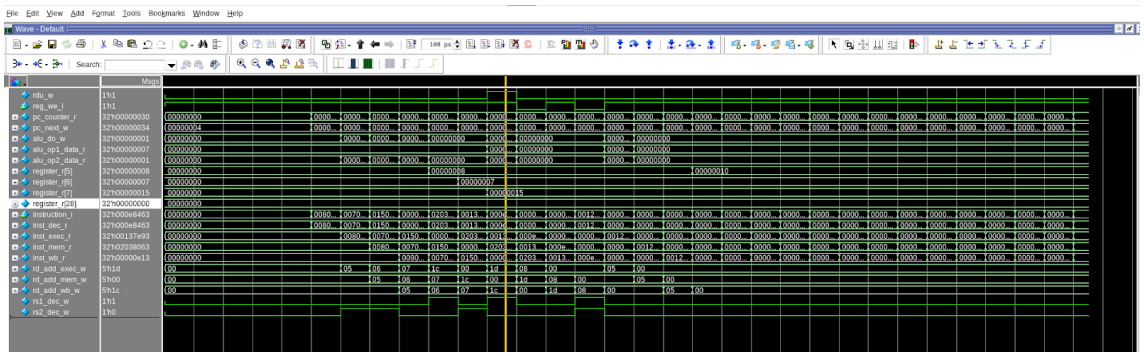


FIGURE 7 – Simulation du programme

Nous pouvons voir que nous n'obtenons pas le bon résultat. En effet, le registre 28 (correspondant à t3) est à 0 tout le long de la simulation alors qu'il est censée être à 56 à la fin de la simulation. Néanmoins, le problème ne provient pas du code, puisque le registre 7 (correspondant à t2) vaut bien 15. Ainsi, le programme réalise bien les 16 itérations. Les registres 5 et 6 correspondant respectivement à t0 et t1, prennent bien les valeurs respectives 8 et 7.
Le problème que l'on rencontre dans ce programme est un problème de dépendances. En effet, certaines instructions présentes dans le code font appel au résultat de l'instruction précédente, sans attendre que l'instruction soit terminée. En effet, les instructions doivent écrire leurs résultats dans les registres avant qu'une instruction suivante puisse les utiliser. Ainsi, dans ce programme on retrouve deux types de dépendances. Les dépendances de données apparaissent lorsque des instructions utilisent des résultats d'autres instructions. En effet, dans le programme on retrouve les deux lignes suivantes :

```
andi t4, t1, 1    # I1 : t4 dépend de la valeur actuelle de t1
beqz t4, .test    # I2 : La branche dépend de la valeur de t4
```

On peut clairement remarquer que nous utilisons t4 sur deux instructions d'affilées. Ainsi, t4 n'est pas encore disponible que nous l'utilisons déjà. On a donc des dépendances de données.
Nous avons également des dépendances de contrôle. Les dépendances de contrôle apparaissent dans les instructions de branchement conditionnel (beqz, j). Par exemple, dans les lignes suivantes :

```
beqz t4,.test #Instruction de brachement conditionnel
j loop #Instruction de saut inconditionnel
```

Pour que le programme sache s'il doit prendre ou non la branche, il doit attendre l'étage EX.
Il est totalement possible de corriger les problèmes avec des NOPs mis à la main dans le programme comme, nous l'avons déjà fait dans le TP n°1.
Néanmoins, cela demande à l'utilisateur de corriger son programme en ajoutant des nops manuellement. Ainsi, ce n'est pas optimale.
Dans la suite, on va ajouter les NOPs manuellement dans un premier temps pour enlever toutes les dépendances, les unes après les autres.

2.2 Correction des problèmes

Correction logicielle

Un NOP est une intruction permettant de corrigger les problèmes de dépendances. En effet, comme son nom l'indique, il permet de faire aucune opération. En ne faisant rien, cette dernière permet de laisser le temps à l'instruction précédente de se terminer avant de faire celle dont on a besoin.
Un nop peut être vu comme l'instruction suivante :

```
addi x0,x0,0
```

Le registre 0 est toujours cablé à 0. Ainsi, cette instruction ajoute 0 à 0, ce qui en pratique n'a aucun effet.

Un NOP est encodé binairement avec l'opcode suivant : 0x00000013. Les NOP sont utils pour enlever les problèmes de dépendance de données. Ainsi, c'est une solution pour corriger les problèmes. Nous avons alors implémenter la solution avec les NOP.

Pour ce faire, il faut dans un premier temps réfléchir à la quantité de NOP à insérer dans notre programme pour qu'il fonctionne. Nous avons alors réalisé un tableau avec chaque instruction nous permettant de comprendre quand avons-nous besoin d'un nop. On retrouve ce tableau sur la figure suivante :

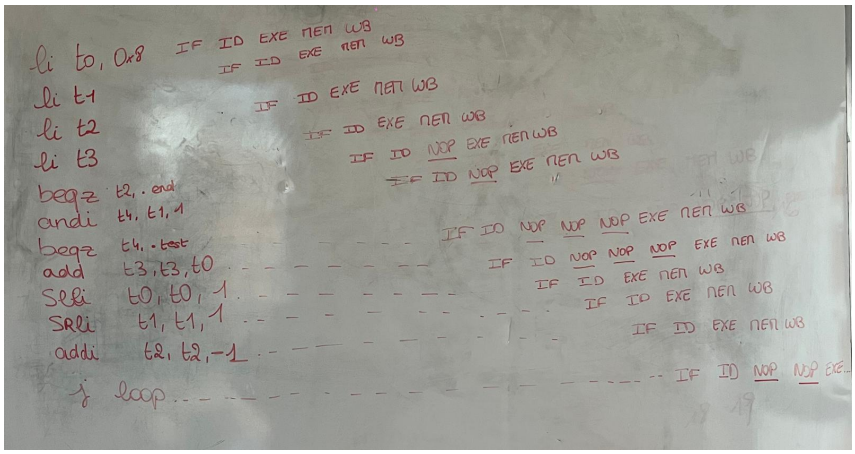


FIGURE 8 – Résolution des problèmes

Les NOP sont soulignés afin de faciliter la compréhension. Nous allons maintenant expliquer la raison de l'emplacement de chaque NOP dans le tableau.
Le premier NOP se situe à la quatrième instruction à savoir :

```
beqz t2,.end
```

En effet, lors du premier test, il faut que l'initialisation de t2 ait eu le temps de se faire. Dans le tableau, il faut attendre que l'instruction soit dans le WB. On peut voir qu'on doit attendre un coup de clock. On ajoute donc un NOP.

Ensuite, on ajoute un NOP avant l'instruction suivante :

```
andi t4,t4,1
```

Ici, il faut attendre de savoir si l'instruction précédente s'est réalisée. En effet, si elle s'est réalisée, alors on va directement à end et on ne doit pas executer l'instruction andi. Ainsi, avant d'exécuter l'instruction andi, il faut attendre que le beqz se soit exécuté.

Puis, nous trouvons des NOP dans l'instruction suivante :

```
beqz t4,.test
```

En effet, afin de tester t4, il faut savoir sa valeur. Il faut attendre l'étage WB de l'instruction précédente pour connaître sa valeur. C'est pour cette raison que nous avons ajouté 3 NOP, ici. On a ajouté le nombre nécéssaire de NOP pour que l'on éxécute cette ligne après le WB de la ligne précédente.

Pour le prochain NOP, c'est sur le même principe que pour le beqz précédent. En effet, on a du ajouter des NOP à cette instruction :

```
add t3,t3,t0
```

Comme la ligne précédente est une instruction de branchement conditionnel, on doit attendre que la ligne précédente s'exécute pour savoir ce que l'on doit faire. En effet, si T4 vaut 0, alors on ne doit pas exécuter l'instruction add.

La dernière instruction nécéssitant un NOP est la suivante :

```
j loop
```

En effet, avant de faire un saut, il faut avoir finit toutes les instructions. Ainsi, il faut que la ligne précédente ait eu le temps d'écrire dans le WB. Ainsi, nous avons ajouté deux NOP pour laisser le temps nécéssaire.

Maintenant que nous avons compris le nombre de NOP nécéssaire et les positions de chacun. Nous avons implémenté le code respectant ce qu'on a vu précédemment.

```
1  .section .start;
2  .globl start;
3
4  start:
5      li t0, 0x8
6      li t1, 0x7
7      li t2, 0x15
8      li t3, 0
9      nop
10     nop
11     loop :
12         beqz t2, .end
13         andi t4, t1, 1
14         nop
15         nop
16         nop
17         beqz t4, .test
18         nop
19         nop
20         add t3, t3, t0
21     .test :
22         slli t0, t0, 1
23         srli t1, t1, 1
24         addi t2, t2, -1
25         nop
26         nop
27         j loop
28     .end :
29         nop
30
31 lab: j lab
32     nop
33
34 .end start
```

En simulant, on obtient :

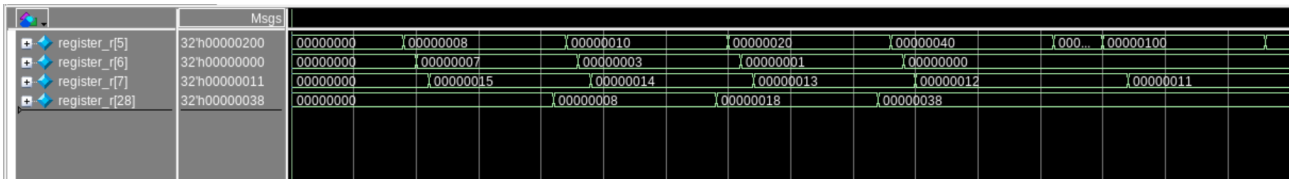


FIGURE 9 – Résolution des problèmes

On peut voir qu'on a bien retrouvé le bon résultat. En effet, on est censé trouver 56. Or 56 en héxadécimal correspond à 38.

Correction matérielle

L'objectif de cette partie est de modifier l'implémentation des différents fichiers pour corriger les problèmes de dépendances sans avoir à ajouter des nops dans le mult.S.
Pour ce faire, nous allons corriger les dépendances de données dans un premier temps, puis les dépendances de contrôle.

2.2.1 Dépendances de données

Nous allons en premier lieu, comprendre le fonctionnement des signaux internes afin de contrôler les dépendances de données.
Tout d'abord, **rs1_dec_w** et **rs2_dec_w** sont des signaux représentant les registres sources de l'instruction à l'étage de décodage(ID).

Comme vu précédemment, nous avons ensuite les étages d'exécution, de mémoire et d'écriture, représenté respectivement par EXE, MEM et WB. Il y a donc des signaux internes pour représenter le registre de destination de l'instruction dans chacun des étages. Ces signaux sont de la forme : rd_add_XXX_w avec XXX l'étage correspondant.

Maintenant que nous avons compris l'utilité des signaux internes, nous pouvons définir le signal **stall** : il sert à interrompre temporairement le pipeline en cas de dépendance de données. Lorsqu'il est actif, l'instruction en cours de décodage devient une opération **NOP**. Le contrôle du pipeline va alors insérer des cycles d'attente pour attendre que la donnée soit disponible avant de continuer l'exécution de l'instruction suivante.

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

Dans la suite de cette partie, nous allons nous demander : de quelle manière implémenter un stall efficace pour gérer les dépendances de données ?

Dans un premier temps, nous allons écrire une version simple de cette équation en ne tenat compte que des registres sources de l'instruction à l'étage DEC et du registre destination dans les étages qui suivent. L'équation du stall simplifié est la suivante :

```
stall_w = (rs1_dec_w != 5'b00000 && rs1_dec_w=rd_add_exe_w ||
           rs1_dec_w = rd_add_mem_w || rs1_dec_w = rd_add_wb_w) ||
          (rs2_dec_w != 5'b00000 && rs2_dec_w = rd_add_exe_w ||
           rs2_dec_w = rd_add_mem_w || rs2_dec_w = rd_add_wb_w)
```

Cette équation vérifie si les registres source **rs1** ou **rs2** dans l'instruction en cours (étage DEC) sont égaux à un registre de destination rd dans les instructions des étapes suivantes (EXE, MEM, WB). Si une telle correspondance existe, cela signifie qu'il y a une dépendance de données et que le signal **stall_w** sera actif, ce qui va introduire un stall dans le pipeline.

Pour affiner le stall, il faut vérifier si le registre source **rs1** est utilisé. Nous avons alors créé un signal pour vérifier si ce dernier est effectivement utlisé. L'équation booléenne du signal interne rs1_used est la suivante :

```
rs1_used = (opcode_dec_w == RV32I_I_INSTR_OPER) ||
           (opcode_dec_w == RV32I_I_INSTR_LOAD) ||
           (opcode_dec_w == RV32I_B_INSTR) ||
           (opcode_dec_w == RV32I_R_INSTR)
```

Il faut également faire le même principe pour l'autre registre source est effectivement utilisé. En effet, certaines instructions n'utilisent pas **rs2**, par exemple, les instructions de type immédiat (comme **LUI**). Ainsi, nous réalisons le même principe pour **rs2**, en créant un signal interne **rs2_used** qui a la forme suivante :

```
rs2_used = (opcode_dec_w == RV32I_R_INSTR) ||
           (opcode_dec_w == RV32I_I_INSTR_OPER) ||
           (opcode_dec_w == RV32I_S_INSTR)
```

Enfin, il faut appliquer le même principe pour vérifier que **rd** est effectivement utilisé. En effet, certaines instructions comme les branch n'écrivent pas dans **rd**. Ainsi, on créée des signaux internes pour chaque étages, c'est-à-dire : **rd_written_exec**, **rd_written_mem** et **rd_written_wb**.
On obtient :

```
rd_written = (opcode_XXX_w == RV32I_I_INSTR_LOAD) ||
             (opcode_XXX_w == RV32I_R_INSTR) ||
             (opcode_XXX_w == RV32I_I_INSTR_OPER) ||
             (opcode_XXX_w == RV32I_J_INSTR)
```

avec XXX pour chaque étage.

Enfin, notre équation booléenne du stall devient :

```
stall_w = ((rs1_used && rs1_dec_w != 5'b00000) &&
          ((rd_written_exec  && rs1_dec_w == rd_add_exec_w) ||
           (rd_written_mem && rs1_dec_w == rd_add_mem_w) ||
           (rd_written_wb  && rs1_dec_w == rd_add_wb_w))) ||
          ((rs2_used && rs2_dec_w != 5'b00000) &&
          ((rd_written_exec && rs2_dec_w == rd_add_exec_w) ||
           (rd_written_mem  && rs2_dec_w == rd_add_mem_w) ||
           (rd_written_wb && rs2_dec_w == rd_add_wb_w)))
```

Ainsi, en modifiant le mult.S c'est-à-dire en enlevant les NOP servant à résoudre les dépendances de données, on simule et on trouve :

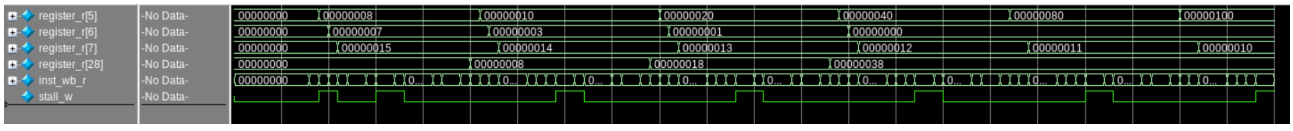


FIGURE 10 – Simulation avec la gestion de stall

2.2.2 Dépendances de contrôle

Gestion des sauts

Les instructions de saut, telles que JAL, introduisent une dépendance de contrôle dans le pipeline d'exécution. En effet, une fois l'adresse cible calculée dans l'étage de décodage

(ID), toutes les instructions chargées (fetch) après le saut deviennent incorrectes si le saut est pris. Ce problème est amplifié dans la pipeline, car plusieurs instructions peuvent déjà être en cours d'exécution.

Afin de gérer ce type de dépendance il suffit d'insérer des NOP, sans modifier l'équation de stall. Les NOPs permettent de vider le pipeline des instructions inutiles en attendant que la mise à jour du PC soit effective.

```
  assign inst_dec_r =(stall_w == 1'b1 || opcode_dec_w == RV32I_J_INSTR || opcode_dec_w == RV32I_I_INSTR_JALR) ?
32'h00000013 : instruction_i;
```

FIGURE 11 – Gestion des sauts

On a enlevé encore les NOP qui géraient les jumps dans le fichier mult.S et on a relancé le programme. Les résultats obtenus sont conformes aux attentes.

FIGURE 12 – Simulation pour la gestion des sauts

Gestion des branchements

Afin d'assurer une gestion optimale des instructions de branchement au sein du pipeline, on doit insérer un NOP (32'h00000013) lorsqu'un saut est effectué. Ce NOP correspond au signal **branch_taken_r** décalé d'un cycle d'horloge.

```
  assign inst_dec_r =(stall_w == 1'b1 || opcode_dec_w == RV32I_J_INSTR || opcode_dec_w == RV32I_I_INSTR_JALR ||
branch_taken_r == 1'b1) ? 32'h00000013 : instruction_i;
```

FIGURE 13 – Emploi du signal branch_taken_r

Le bloc **exec_stage** attribue à **inst_exec_r** une nouvelle valeur en fonction de l'état du système. En cas de réinitialisation ou de branchement pris, **inst_exec_r** est mis à jour en conséquence. Dans les autres cas, il prend simplement la valeur de **inst_dec_r**.

```
always_ff @(posedge clk_i or negedge resetn_i) begin : exec_stage
  if (resetn_i == 1'b0) inst_exec_r <= 32'h0;
  else if (branch_taken_w == 1'b1) inst_exec_r <= 32'h00000013;
  else inst_exec_r <= inst_dec_r;
end
assign rd_add_exec_w = inst_exec_r[11:7];
```

FIGURE 14 – Emploi du signal branch_taken_w

Le bloc **branch_taken_delay** introduit un retard d'un cycle d'horloge sur le signal **branch_taken_w**, alimentant ainsi le registre **branch_taken_r**. Ce mécanisme assure une synchronisation précise des décisions de branchement au sein du pipeline.

```
always_ff @(posedge clk_i or negedge resetn_i) begin : branch_taken_delay
  if (resetn_i == 1'b0) branch_taken_r = 1'b0;
  else branch_taken_r <= branch_taken_w;
end
```

FIGURE 15 – Le bloc branch_taken_delay

On a encore enlevé des NOP pour les branchements dans mult.S et on a relancé le programme. On a bien eu t3 = 0x38, le signal de branchement est là, donc notre solution fonctionne.

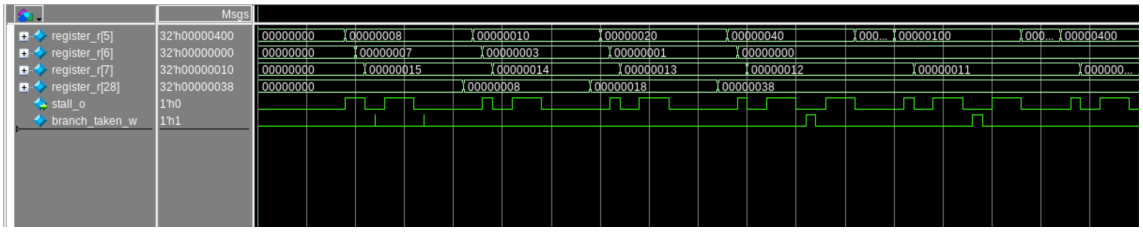


FIGURE 16 – Simulation pour la gestion des branchements

2.3 Optimisation en utlisant le bypass

Le **bypass** gère spécifiquement les dépendances de données, plus précisément celles causées par des dépendances de type RAW (Read After Write). Une dépendance RAW survient lorsqu'une instruction tente de lire une donnée qui n'a pas encore été écrite par une instruction précédente. Prenons l'exemple de deux instructions :

```
I1 : ADD t2, t1, t0
I2 : SUB t3, t2, t0
```

I2 est dépendante du résultat de I1 pour utiliser la valeur de t2. Si I2 atteint la phase d'exécution (EXE) avant que I1 n'ait fini d'écrire son résultat (WB), une dépendance RAW se produit.

Le **bypass** résout cette dépendance en transmettant directement le résultat de l'instruction I1 dans l'étage (EXE) à l'instruction I2, sautant ainsi l'étape d'écriture (WB) et évitant un stall.

Dans la suite de cette partie, nous allons implémenter le bypass dans notre architecture.

Pour ce faire, on a ajouté dans le controlpath, ce bloc qui détecte si un bypass est nécessaire pour rs1 et rs2 en comparant les registres sources de l'instruction en décodage avec le registre destination de l'instruction en exécution, à condition que cette dernière ne soit pas une instruction de type **load**.

```
always_comb begin : bypass
  bypass_rs1 = rs1_dec_w != 5'b00000 && rs1_dec_w == rd_add_exec_w && opcode_exec_w != RV32I_I_INSTR_LOAD;
  bypass_rs2 = rs2_dec_w != 5'b00000 && rs2_dec_w == rd_add_exec_w && opcode_exec_w != RV32I_I_INSTR_LOAD;
end
```

FIGURE 17 – Signaux du bypass pour rs1 et rs2

Il ne faut pas générer un bypass dans le cas d'une instruction **load**. En effet, une telle instruction écrit une valeur dans un registre rd, cette valeur n'est pas immédiatement accessible aux instructions suivantes qui en dépendent. Pour éviter d'utiliser une valeur incorrecte, le pipeline est arrêté (stall) jusqu'à ce que la valeur soit effectivement écrite dans le registre dans l'étage WB.

Après avoir implémenté la logique de détection des bypass, on définit les pins pour le bypass dans RV32i_pkg.

```
// Alu operand1 selector
const logic [1 : 0] SEL_OP1_RS1 = 2'b00;
const logic [1 : 0] SEL_OP1_IMM = 2'b01;
const logic [1 : 0] SEL_OP1_PC = 2'b10;
const logic [1 : 0] SEL_OP1_BYPASS = 2'b11;


// Alu operand2 selector
const logic SEL_OP2_RS2 = 1'b0;
const logic SEL_OP2_IMM = 1'b1;
const logic [1 : 0] SEL_OP1_BYPASS = 2'b10;
```

FIGURE 18 – définition des pins dans RV32i_pkg

Il faut également modifier alu_src1_mux_comb en conséquence.

```
always_comb begin : alu_src1_comb
   if (bypass_rs1) alu_src1_comb = SEL_OP2_BYPASS;
   else begin
       case (opcode_dec_w)
         RV32I_U_INSTR_LUI: alu_src1_o = SEL_OP1_IMM;
         RV32I_U_INSTR_AUIPC: alu_src1_o = SEL_OP1_PC;
         default: alu_src1_o = SEL_OP1_RS1;
       endcase
   end
end
```

FIGURE 19 – modification du alu_src1_mux_comb

De même pour alu_src2_mux_comb

```
always_comb begin : alu_src2_comb
  if (bypass_rs2) alu_src2_comb = SEL_OP2_BYPASSS;
  else begin
      case (opcode_dec_w)
        RV32I_I_INSTR_OPER: alu_src2_o = SEL_OP2_IMM;
        RV32I_I_INSTR_LOAD: alu_src2_o = SEL_OP2_IMM;
        RV32I_U_INSTR_AUIPC: alu_src2_o = SEL_OP2_IMM;
        RV32I_S_INSTR: alu_src2_o = SEL_OP2_IMM;
        default: alu_src2_o = SEL_OP2_RS2;
      endcase
  end
end
```

FIGURE 20 – modification du alu_src2_mux_comb

On doit également modifier l'équation de stall afin d'éviter que le bypass et le stall ne soient générés simultanément. Pour cela, on supprime de l'équation du stall les deux parties suivantes, car ces cas seront désormais pris en charge par le bypass.

```
(rd_written_exec && rs1_dec_w == rd_add_exec_w)
(rd_written_exec && rs2_dec_w == rd_add_exec_w)
```

L'équation de stall devient alors :

```
assign stall_w= ((rs1_used && rs1_dec_w != 5'b00000)&&((rd_written_mem && rs1_dec_w==rd_add_mem_w)||(rd_written_wb
&& rs1_dec_w==rd_add_wb_w)))||((rs2_used && rs2_dec_w != 5'b00000)&&((rd_written_mem && rs2_dec_w==rd_add_mem_w)||
(rd_written_wb && rs2_dec_w==rd_add_wb_w)));
```

FIGURE 21 – modification de l'équation de stall pour le bypass

On a utilisé le programme en assembleur suivant pour mettre en évidence ce type de dépendance.

```
1  .section .start
2  .globl start
3
4  start:
5      li t0, 0x2
6      li t1, 0x8
7      add t2, t1, t0
8      sub t3, t2, t0
9
10 lab1:
11     j lab1
12     nop
13
14 .end start
```

L'instruction `sub t3, t2, t0` dépend de la valeur calculée par `add t2, t1, t0`.

2.3.1 Pipeline sans Bypass

Cycle	Instruction	IF	ID	EX	MEM	WB
1	li t0, 0x2	IF				
2	li t1, 0x8	IF	ID			
3	add t2, t1, t0	IF	ID	EX		
4	sub t3, t2, t0	IF	ID	—	—	—
5	sub t3, t2, t0	—	—	ID	EX	MEM

TABLE 1 – Les étapes du pipeline sans bypass

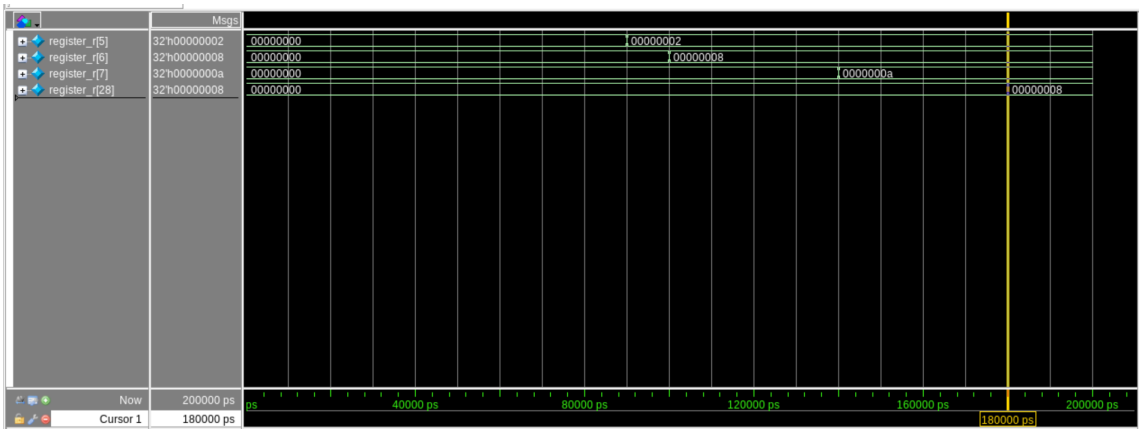


FIGURE 22 – Simulation sans le bypass

Entre le cycle 4 et le cycle 5, on doit insérer des **NOPs** pour attendre la fin de l'écriture de `t2`.

2.3.2 Pipeline avec Bypass

Cycle	Instruction	IF	ID	EX	MEM	WB
1	li t0, 0x2	IF				
2	li t1, 0x8	IF	ID			
3	add t2, t1, t0	IF	ID	EX		
4	sub t3, t2, t0	IF	ID	EX	MEM	WB

TABLE 2 – Les étapes du pipeline avec bypass

En utilisant le bypass, l'instruction `sub` peut utiliser la valeur de `t2` directement depuis l'étape EX de `add`.

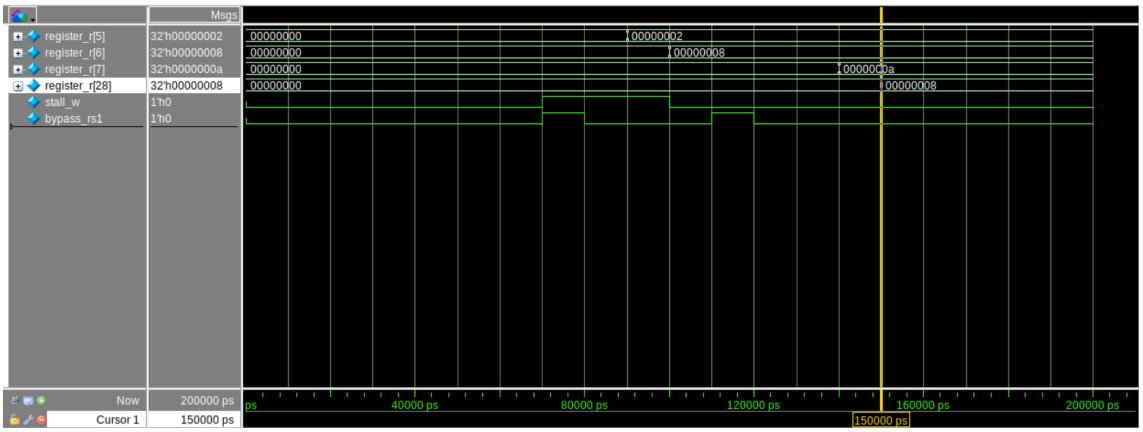


FIGURE 23 – Simulation avec le bypass

En utilisant le bypass, on peut bien remarquer une réduction du temps d'exécution (180 ns sans bypass contre 150 ns avec bypass), ce qui démontre clairement l'efficacité de cette solution.

3 TD n°3 : Implémentation d'une mémoire cache

Dans cette partie, on va implémenter deux types de mémoire cache à savoir la cache instruction direct et la cache instruction associatif à deux voies.
Une mémoire cache permet d'exploirer les propriétés de localité pour stocker les données récentes dans une mémoire plus rapide. Il existe deux types de propriétés de localité. En effet, la localité temporelle se base sur le principe suivant : Si une donnée est accédée, elle a de fortes chances d'être réutilisée prochainement. Cependant, la localité spatiale se base sur le principe suivant : Si une donnée est accédée, les données voisines ont de fortes chances d'être accédées prochainement.

Avant de se concentrer sur un des deux types de mémoire, il parait important de comprendre les données générales d'une cache. En effet, le nombre d'octets, le nombre de mots, et le nombre d'entrées sont des **paramètres** que l'on utilise dans les deux types de caches.

Nombre d'octets
La taille de la ligne de cache est déterminée par le champ OFFSET. Le champ OFFSET a 4 bits (de 3 à 0), ce qui signifie qu'il peut représenter $2^4 = 16$ valeurs différentes. Donc, la taille de la ligne de cache est de 16 octets.

Nombre de mots
Si un mot est de 4 octets (32 bits), alors la ligne de cache contient $16/4 = 4$ mots.

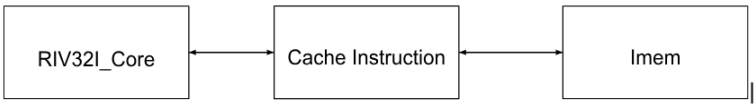
Nombre d'entrées
Le champ INDEX a 6 bits (de 9 à 4), ce qui signifie qu'il peut représenter $2^6 = 64$ valeurs différentes. Donc, le nombre d'entrées de cette mémoire cache est de 64 lignes.

On peut maintenant se poser des questions sur la **performance** d'une cache avec ces paramètres.
Tout d'abord, le taux de hit , aussi appelé taux de succés est la proportion de demandes d'accès à la mémoire qui trouvent les données recherchées dans la mémoire cache. Et inversement, le taux de miss, aussi appelé le taux d'échec, est la proportion de demandes d'accès à la mémoire qui ne trouvent pas les données rechcerchées dans la mémoire cache et doivent donc accéder à la mémoire principale.
On peut voir dans l'énoncé que le *taux de hit* est de 90 %. Comme le taux de hit et le taux de miss sont complémentaires, le *taux de miss* est de $100\% - 90\% = 10\%$.
Comme le taux de hit est élevé, ou encore le taux de miss est faible, on peut dire que la mémoire cache est efficace.

Temps d'accès moyen = (Taux de HIT × Temps de HIT) + (Taux de MISS × Temps de MISS)
Temps de HIT = 1 cycle
Temps de MISS = 10 cycles
Temps d'accès moyen = $(0.9 \times 1) + (0.1 \times 10) = 0.9 + 1 = 1.9$ cycles
Ce temps d'accès moyen de 1.9 cycles indique que, bien que la mémoire cache soit efficace dans la plupart des cas, les accès manqués (MISS) ont un impact significatif sur la performance globale.

3.1 Cache mémoire "direct"

La cache mémoire directe est caractérisée par le fait que chaque bloc de mémoire principale a une place unique dans la cache. Dans cette partie, on va implémenter une cache directe. Pour ce faire,on va faire un schéma pour représenter le lien entre la mémoire cache, RV32i_core et imem.
La cache d'instruction se siture entre le processeur est la mémoire d'instructions.



RV32i_core envoie une requête à cache_instruction pour obtenir une instruction.
Si l'instruction est présente dans cache_instruction (cache hit), elle est renvoyée directement à RV32i_core. Si l'instruction n'est pas présente dans cache_instruction (cache miss), cache_instruction envoie une requête à imem pour obtenir l'instruction.
L'instruction est ensuite renvoyée à cache_instruction, qui la stocke et la renvoie à RV32i_core.

Nous pouvons maintenant nous intéresser aux paramètres de la connexion entre la cache_instruction et imem. Comme il s'agit d'un processeur de 32 bits, la largeur du bus de données entre imem et le cache_instruction est de 32 bits (4 octets).
Dans le fichier RV32I_pipeline_top.sv, on trouve une nouvelle entrée, à savoir imem_validi_i. Elle sert à indiquer si les données provenant de la mémoire d'instructions (imem) sont valides. Cela permet au cache de savoir quand les données peuvent être utilisées.

Dans le fichier cache_direct.sv, nous devons rensigner dans un premier temps les caractéristiques des paramètres locaux du module. On trouve sur l'image suivante, les caractéristiques :

```
localparam ByteOffsetBits =4,
localparam IndexBits = 6,
localparam TagBits = 22,

localparam NrWordsPerLine = 4,
localparam NrLines = 64,

localparam LineSize = 32 * NrWordsPerLine
```

FIGURE 24 – Caractéristiques de la cache

Dans la cache, il est nécéssaire de stocker les données des tag, des données et de validité. On retrouve la création de ces signaux sur la figure suivante :

```
logic [TagBits-1:0] tag_mem [0:NrLines-1];      // Mémoire pour les tags
logic [LineSize-1:0] data_mem [0:NrLines-1];    // Mémoire pour les données
logic valid_mem [0:NrLines-1];                  // Mémoire de validité
```

FIGURE 25 – Création de signaux internes à mémoriser

Comme nous avons pu voir dans le cours, le tag, l'index et le offset sont un découpage prédéfini sur l'adrresse reçu. Ainsi, nous avons ajouté les lignes de code présentes dans la figure suivante afin de récupérer ces données :

```
assign tag_mem=addr_i[31:10];
assign offset_mem=addr_i[9:4];
assign index_mem=addr_i[3:0];
```

FIGURE 26 – Création de signaux internes à mémoriser

Maintenant que nous avons implémenté les signaux de tag, d'index et d'offset. Nous pouvons nous concentrer sur le signal modélisant la reqête hit. Ainsi, nous avons créé un signal interne hit valant 1 quand la requête est hit, et 0 quand la requête est miss.
On initialise le signal hit comme étant le fait que valid_mem[index] qui accède au bit de validité de la ligne de cache à l'index spécifié soit bien à 1 et si le tag de la ligne de cache correspond au tag de l'adresse demandé et si l'opération de lecture est active. On retrouve la ligne de code sur la figure suivante :

```
logic hit;
assign hit = (valid_mem[index] && (tag_mem[index] == tag) && read_en_i);
```

FIGURE 27 – Implémentation du hit

Maintenant que nous avons implémenté un signal hit, nous devons traiter le cas "hit" pour la sortie de réponse. Pour ce faire, on doit utiliser les blocs de logique combinatoire, d'où always_comb. Les données lues prennent la valeur de la ligne de cache à l'index spécifié en précisant que l'on s'intéresse uniquement aux 32 bits à partir de l'offset. Ainsi, on peut traduire ce principe par le code, que l'on retrouve sur la figure suivante :

```
always_comb begin
    if (hit)
        read_word_o = data_mem[index][offset * 32 +: 32];
end

assign read_valid_o = hit;
```

FIGURE 28 – Traitement du cas "hit"

De plus, il faut un signal pour savoir si la lecture est valide. La lecture est valide uniquement lorsque le hit vaut 1. C'est bien ce que l'on retrouve sur la capture d'écran.
Il ne faut pas oublier qu'en cas de miss, alors on s'intéresse à la mémoire imem. Ainsi, on créé deux signaux internes afin de gérer le cas de miss. En effet, on a accès à la mémoire qu'en cas de miss, ou encore la négation de hit. Afin d'aligner la taille de la ligne, on à juste à concaténer tag et index. On ne s'intéresse pas à l'offset. Ainsi, on ajoute quatre 0 pour remplacer l'offset. On retrouve cette idée dans le code suivant :

```
assign mem_addr_o = {tag, index, 4'b0}; // Aligner sur la taille de la ligne
assign mem_read_en_o = !hit; //accès à la mémoire qu'en cas de miss
```

FIGURE 29 – Accès à la mémoire

Maintenant, que nous avons réussi à créer les signaux internes de mémoire, nous devons uniquelent sse concentrer sur le reset et l'actualisation de la cache quand la mémoire de lecture est complète.
Il faut tout initialiser à 0 au début du programme ce que l'on fait grâce à cette partie du code :

```
if (!rstn_i) begin
    // Reset
    integer i;
    for (i = 0; i < NrLines; i = i + 1) begin
        valid_mem[i] <= 1'b0;
        tag_mem[i] <= {TagBits{1'b0}};
        data_mem[i] <= {LineSize{1'b0}};
    end
```

FIGURE 30 – Initialisation

Ensuite, il faut affecter la valeur 1'b1 au bit de validité, mettre à jour le tag stocké et implémenter le stockage du résultat dans la mémoire de cache.

```
end else if (mem_read_valid_i) begin
    // Actualisation de la cache quand la mémoire de lecture est complète
    data_mem[index] <= mem_read_data_i;
    tag_mem[index] <= tag;
    valid_mem[index] <= 1'b1;
end
```

FIGURE 31

Maintenant que nous avons réussi à implémenter la mémoire cache instruction direct, nous pouvons simuler et vérifier le bon fonctionnement de cette dernière. Afin de faire fonctionner le cache, nous avons du modifier le control_path et le data_path (nous avons été aidé par un autre binôme d'un autre groupe). En effet, il est important d'ajouter un signal imem_valid_i, en entrée dans le control_path et en sortie dans le data_path. En effet, ce signal permet de vérifier si on doit regarder la mémoire. Si c'est le cas, il faut faire une pause dans les instructions, mais il ne faut pas activer un stall, car on veut que le pc_counter continue à tourner. De ce fait, le signal imem_valid_i est utilisé dans toutes les logiques séquentielles du control_path et data_path. Ainsi, la simulation nous donne :

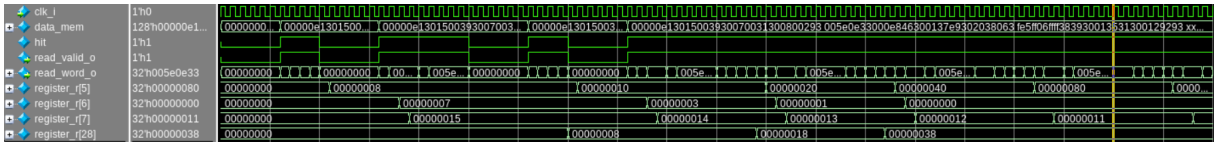


FIGURE 32 – Simulation du cache direct

Nous pouvons remarquer que la mémoire cache direct fonctionne correctement. En effet, dans le registre 28, on retrouve bien la bonne valeur. De plus, on remarque que les mots se remplissent bien dans la cache.

3.2 Cache instruction associatif à deux voies

Nous devons maintenant implémenter une mémoire cache instruction associatif à deux voies. Ce type de mémoire est caractérisé par le fait que plusieurs blocs de mémoire principale peuvent être placés dans différentes lignes de la cache.
On peut donc réutiliser le cache direct que l'on va modifier en conséquence pour le rendre associatif, avec une priorité de type LRU. Pour simplifier l'implémentation, le cache sera associatif à deux voies. Il faut stocker de nouvelles données dans le cache :

— la deuxième voie du cache
— le numéro (indice) de la voie LRU pour chaque ligne du cache

```
//paramètres à stocker
logic [TagBits-1:0] tag_mem1 [0:NrLines-1];      // mémoire du tag du cache 1
logic [LineSize-1:0] data_mem1 [0:NrLines-1];    // mémoire de données du cache 1
logic valid_mem1 [0:NrLines-1];                  // bits valides du cache 1


logic [TagBits-1:0] tag_mem0 [0:NrLines-1];      // mémoire du tag du cache 0
logic [LineSize-1:0] data_mem0 [0:NrLines-1];    // mémoire de données du cache 0
logic valid_mem0 [0:NrLines-1];                  // bits valides du cache 0
```

FIGURE 33 – Deux voies du cache

Ensuite, il faut créer le signal pour la voie LRU. Nous l'avons fait avec la ligne suivante :

```
logic lru_mem [0:NrLines-1];
```

De plus, il faut un hit pour chaque voie, codés de la même façon que pour un cache direct. Ensuite, nous avons réaliser un ou entre les deux hits. En effet, pour que la réquête soit "hit", il faut qu'elle le soit soit en passant par la voie 0, soit en passant par la voie 1. Le hit que nous avons implémenté est codé de la manière suivante :

```
assign hit0 = (valid_mem0[index] && (tag_mem0[index] == tag));
assign hit1 = (valid_mem1[index] && (tag_mem1[index] == tag));
assign hit = hit0 || hit1;
```

Il suffit maintenant d'adapter l'idée que l'on a développé pour une mémoire cache direct à notre nouvelle mémoire associatif à deux voies. Ainsi, en cas de hit, il faut savoir la voie utilisé pour écrire la bonne donnée en mémoire. Si on est dans le cas de miss, il ne faut pas oublier de mettre la variable read_word_o à 0, puisqu'on ne lit aucun mot. On retrouve cette idée grâce à la logique combinatoire suivante :

```
always_comb begin
    if (hit) begin
        if (hit0) read_word_o = data_mem0[index][(offset * 32) +: 32];
        else read_word_o = data_mem1[index][offset * 32 +: 32];
     end else read_word_o = 32'b0;
end
```

La dernière modification à faire est dans la logique séquentielle, il ne faut pas oublier d'initaliser la valeur du LRU à 0. De plus, lru_mem permet de connaître un moins récent utlisé. Ainsi, si c'est la voie 0, il faut utiliser la voie 1 et renseigner la modification de

lru_mem, et de même pour la voie 1. Ainsi, on retrouve les modifications sur la figure suivante :

```
// Logique séquentielle
always_ff @(posedge clk_i or negedge rstn_i) begin
    if (!rstn_i) begin
        integer i;
        for (i = 0; i < NrLines; i = i + 1) begin
            valid_mem0[i] <= 1'b0;
            tag_mem0[i] <= {TagBits{1'b0}};
            data_mem0[i] <= {LineSize{1'b0}};
            valid_mem1[i] <= 1'b0;
            tag_mem1[i] <= {TagBits{1'b0}};
            data_mem1[i] <= {LineSize{1'b0}};
            lru_mem[i] <= 1'b0;
        end
    end else if (mem_read_valid_i) begin
        if (lru_mem[index]==0) begin
        data_mem1[index] <= mem_read_data_i;
        tag_mem1[index] <= tag;
        valid_mem1[index] <= 1'b1;
        lru_mem[index] <= 1;
        end
        else begin
        data_mem0[index] <= mem_read_data_i;
        tag_mem0[index] <= tag;
        valid_mem0[index] <= 1'b1;
        lru_mem[index] <= 0;
        end
    end
end
```

FIGURE 34 – Mise à jour de la valeur du LRU

Ainsi, maintenant nous pouvons simuler la mémoire cache instruction associatif à deux voies et on trouve :

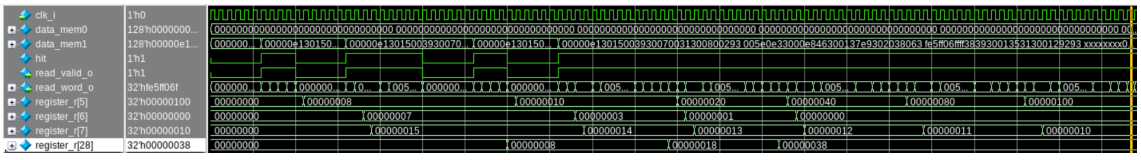


FIGURE 35 – Simulation

Le regsitre 28 a la bonne valeur attendue. De plus, les données sont écrites dans les deux voies du cache. Ainsi, nous avons réussi à implémenter la mémoire cache instruction associatif à deux voies.