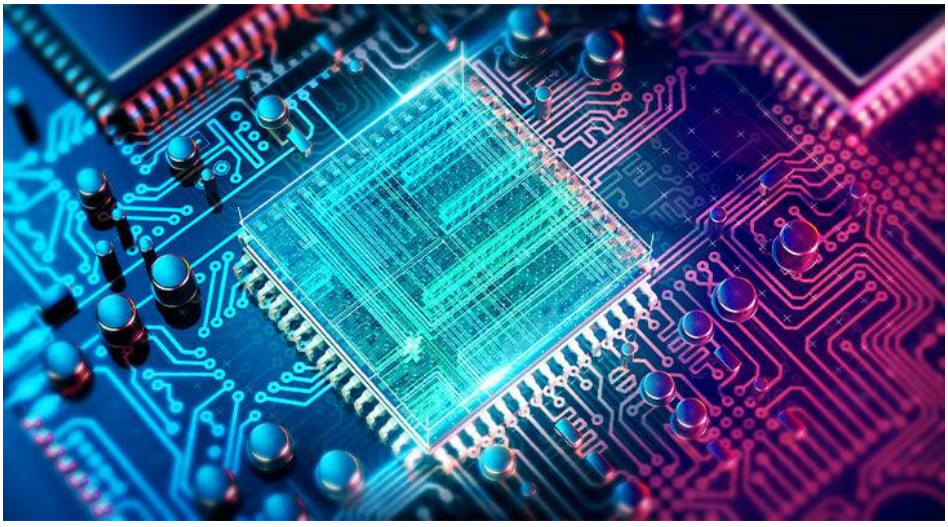
Mines Saint-Etienne

Rapport Architecture des Processeurs II

Enseignants : Olivier POTIN, Simon PONTIE



Yasser EL KOUHEN

Promotion EI23

Table des matières

Table des Illustrations

Remerciements

Je souhaite exprimer ma profonde gratitude à Ibrahim Hadj-Arab, dont l'aide précieuse m'a permis d'acquérir une meilleure compréhension des concepts fondamentaux et avancés en Architecture des Processeurs II. Grâce à ses explications claires et à sa disponibilité, j'ai pu surmonter de nombreuses difficultés théoriques et mieux appréhender les subtilités du pipeline, des dépendances et des caches.

Je tiens également à remercier Simon Pontie et Olivier Potin pour la flexibilité et la compréhension dont ils ont fait preuve à mon égard. Face à une situation qui me semblait désespérée, ils ont su faire preuve d'une grande bienveillance, me permettant ainsi d'aborder ce travail avec plus de sérénité. Leur compréhension m'a été d'une aide importante pour avancer malgré les contraintes rencontrées.

À travers ce rapport, j'espère témoigner du sérieux et de l'implication que j'ai pu mettre dans l'étude et la mise en application des concepts abordés tout au long de ces travaux dirigés

Introduction

Ce rapport présente l'ensemble des travaux dirigés réalisés dans le cadre du cours « Architecture des Processeurs RISC-V » (ISMIN 2A, 2024–2025). Il se divise en trois parties distinctes, chacune abordant des problématiques essentielles à la conception et à l'optimisation d'un processeur moderne.

Dans un premier temps, nous étudions en détail le processeur RISC-V pipeliné (TD1). Nous analysons la hiérarchie du système sur puce (SoC), en identifiant les modules clés tels que le cœur du processeur (RV32i_top), la mémoire d'instructions (imem) et la mémoire de données (dmem). Nous examinons également la structure du pipeline à cinq étages (IF, ID, EX, MEM, WB) et la propagation des signaux, notions fondamentales pour comprendre comment les instructions et les données circulent dans le processeur.

Dans un second temps, nous nous intéressons aux dépendances qui surviennent dans un pipeline (TD2). Les hazards, notamment de type RAW (Read After Write) et de contrôle, représentent un défi majeur pour la performance des processeurs pipelinés. Pour y remédier, nous avons exploré diverses stratégies : l'insertion d'instructions NOP, la mise en œuvre d'un interlock matériel et l'implémentation d'un bypass (forwarding). Ces techniques visent à réduire les cycles perdus et à garantir la bonne synchronisation entre les différentes étapes du pipeline.

Enfin, le troisième volet du rapport (TD3) porte sur l'implémentation d'une mémoire cache d'instructions. Dans cette partie, nous développons d'abord un cache direct (direct mapped) en lecture seule, puis nous étendons cette solution à un cache associatif à deux voies dotées d'une politique de remplacement LRU (Least Recently Used). L'étude des caches est primordiale, car ils jouent un rôle déterminant dans la réduction des temps d'accès à la mémoire et, par conséquent, dans l'amélioration globale des performances du processeur.

Pour mener à bien ces travaux, nous avons utilisé ModelSim pour la simulation numérique et le compilateur RISC-V GCC pour la compilation des programmes en assembleur. Ces outils nous ont permis d'observer de près le comportement des architectures mises en œuvre, d'analyser les waveforms et de valider nos choix techniques.

Ce rapport se veut ainsi une synthèse détaillée et pédagogique des concepts clés de l'architecture des processeurs modernes, allant de la gestion du pipeline et des hazards jusqu'à l'optimisation des accès mémoire par le biais des caches. Dans les sections suivantes, chaque question posée lors des TD est traitée de manière exhaustive afin d'apporter des éclaircissements sur les enjeux théoriques et pratiques rencontrés.

1. TD1 – Étude du Processeur RISC-V Pipeliné

1.1. Analyse du SoC et de sa hiérarchie

Q1 – 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 module **RV32i_soc**. L'analyse des fichiers montre que ce module instancie trois sous-circuits principaux :

- **RV32i_top** : le cœur du processeur qui englobe le datapath et le control_path.
- **wsync_mem imem** : la mémoire d'instructions, qui stocke le programme à exécuter.
- **wsync_mem dmem** : la mémoire de données, qui contient les données manipulées par le programme.

Cette organisation hiérarchique permet de séparer la logique de traitement (RV32i_top) de la gestion des mémoires.

Q2 – Quel est le rôle de chacun de ces sous-circuits ?

- **RV32i_top :**

  – Il constitue le cœur du processeur RISC-V.

  – Il exécute le jeu d'instructions RV32I en coordonnant le datapath (traitement des données) et le control_path (décodage et génération de signaux de commande).

  – Il gère le pipeline en répartissant l'exécution sur cinq étages (IF, ID, EX, MEM, WB).

- **wsync_mem imem :**

  – Cette mémoire contient le programme à exécuter.

  – Elle est de taille 4096 mots de 32 bits, soit 16 Ko, et est mappée à l'adresse de base **0x0000_0000**.

– Son accès en lecture est asynchrone (lecture immédiate dès l'activation du signal) tandis que l'écriture est synchronisée (pour l'initiation).

- **wsync_mem dmem :**

  – Elle stocke les données manipulées par le programme.

  – Elle possède également 4096 mots de 32 bits (16 Ko) et est mappée à **0x0001_0000**.

  – Elle supporte les accès par octet grâce au signal **ble_i** et réalise les écritures de manière synchronisée.

Q3 – De quels circuits est composé le RV32i_top ? Remplissez le schéma de la Figure 1.

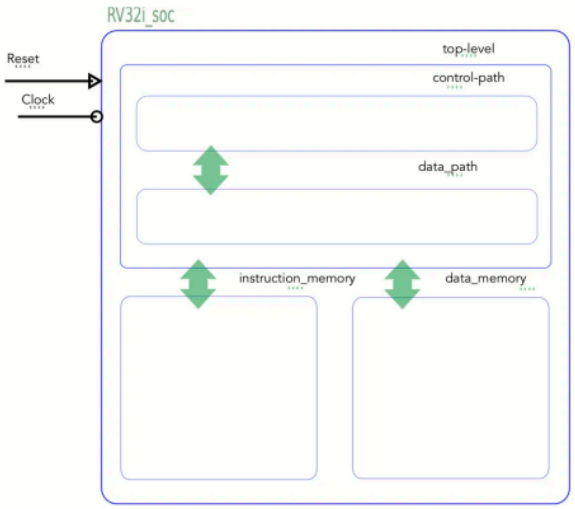


Figure 1 Hiérarchie des composants du RISC-V pipeliné

Le module **RV32i_top** se compose principalement de deux sous-circuits :

- **Data_path :** qui effectue toutes les opérations arithmétiques et logiques, gère les transferts de données et réalise les calculs d'adresses.
- **Control_path :** qui décode l'instruction, génère les signaux de commande pour les autres modules (ALU, accès mémoire, etc.) et orchestre le déroulement du pipeline.

Ainsi, dans le schéma global du SoC, on retrouve :

- **RV32i_soc** au sommet, contenant **RV32i_top** (qui lui-même intègre Data_path et Control_path), ainsi que les mémoires **imem** et **dmem** reliées par des bus de données et d'adresses.

1.2. Étude du Pipeline

Q4 – De combien d'étages est composé le cœur RISC-V ? Nommez-les et donnez leur rôle.

Le processeur pipeliné se divise en **5 étages** :

1. **IF (Instruction Fetch) :**

   – Récupère l'instruction depuis imem.

   – Met à jour le compteur de programme (PC).

2. **ID (Instruction Decode) :**

   – Décode l'instruction lue.

   – Extrait les champs d'opérandes et les signaux de contrôle.
   – Lit le banc de registres pour récupérer les valeurs des registres sources.

3. **EX (Execute) :**

   – Effectue les opérations arithmétiques et logiques à l'aide de l'ALU.
   – Calcule l'adresse pour les opérations de mémoire.

4. **MEM (Memory Access) :**

   – Gère les accès mémoire pour les instructions Load (lecture) et Store (écriture).

5. **WB (Write Back) :**

   – Écrit le résultat final (issu de l'ALU ou de la mémoire) dans le registre destination.

Ce découpage permet une exécution simultanée et fluide de plusieurs instructions en traitant chacune à une étape différente du pipeline.

Q5 – Quels sont les signaux à destination du control_path ?

Les principaux signaux entrants dans le module **RV32i_controlpath** sont :

- **clk_i** : l'horloge qui synchronise l'ensemble du système.
- **resetn_i** : le signal de réinitialisation (actif bas) pour remettre le processeur dans un état connu.
- **instruction_i** : l'instruction à décoder, extraite de la mémoire d'instructions.
- **alu_zero_i** et **alu_lt_i** : signaux fournis par l'ALU, qui indiquent respectivement si le résultat est nul ou négatif, et sont essentiels pour la gestion des branchements conditionnels.

Ces signaux permettent au control_path de générer les commandes nécessaires pour guider l'exécution dans les différents étages.

1.3. Interfaces Mémoire

Q6 – Quels sont les signaux à destination de la mémoire d'instructions ?

L'instance **imem** reçoit les signaux suivants :

- **clk_i** : pour synchroniser les opérations.
- **we_i** : signal d'écriture, qui est maintenu désactivé (1'b0) puisque imem est en lecture seule.
- **re_i** : signal de lecture, activé en fonction de l'activation du chip-select (imem_cs).
- **add_i** : l'adresse envoyée par le processeur, généralement alignée sur la taille d'un mot ou d'une ligne.
- **ble_i** : permet d'activer les 4 octets constituant une instruction.
- **d_i** : données à écrire (non utilisées ici).

Ce jeu de signaux permet à imem de fournir rapidement l'instruction demandée par le processeur.

Q7 – Quels sont les signaux à destination de la mémoire de données ?

L'instance **dmem** reçoit :

- **clk_i** : synchronisation de la mémoire.
- **we_i** : signal d'écriture, activé pour les opérations Store.
- **re_i** : signal de lecture, activé lors des opérations Load.
- **add_i** : adresse générée par le processeur pour accéder aux données.
- **ble_i** : permet d'activer uniquement certains octets, utile pour des accès partiels.
- **d_i** : données à écrire lors des Store.

Ces signaux garantissent un accès précis et contrôlé aux données stockées dans dmem.

1.4. Propagation des Instructions et des Registres

Q8 – Dans le control_path, que représentent les signaux inst_dec_r, inst_exec_r, inst_mem_r, et inst_wb_r ?

Ces signaux correspondent à la même instruction qui circule à travers les différents étages du pipeline :

- **inst_dec_r :** Instruction actuellement décodée à l'étape ID.
- **inst_exec_r :** Instruction transmise à l'étape d'exécution (EX) pour pilotage de l'ALU et autres calculs.
- **inst_mem_r :** Instruction en transit vers l'étape MEM, pour les opérations d'accès à la mémoire.
- **inst_wb_r :** Instruction qui atteint le stade WB et dont le résultat sera écrit dans le banc de registres.

Ils permettent ainsi de suivre l'évolution et la propagation de chaque instruction dans le pipeline.

Q9 – Suivez le chemin de l'index du registre de destination depuis l'étage de décode jusqu'au banc de registre. Commentez.

L'index du registre destination (extrait des bits [11:7] de l'instruction) suit le parcours suivant :

1. **ID :** Lors du décodage, l'index est extrait et stocké dans un signal (par exemple, **rd_add_dec_w**).
2. **EX :** Cet index est propagé sous forme de **rd_add_exec_w**, où il peut être utilisé par l'ALU pour identifier le registre cible dans certaines opérations.
3. **MEM :** L'index est ensuite transmis en tant que **rd_add_mem_w** afin de préparer un accès mémoire si l'instruction le requiert (cas des Load par exemple).
4. **WB :** Enfin, en write-back, l'index apparaît comme **rd_add_wb_w** et détermine le registre à mettre à jour dans le banc de registres.

Ce chemin garantit que le résultat calculé est correctement écrit dans le registre correspondant.

1.5. Gestion des Hazards et Arrêt du Pipeline

Q10 – Comment est actuellement généré le signal stall_w ?

Initialement, la génération de **stall_w** se base sur des comparaisons simples entre les registres sources (rs1_dec_w et rs2_dec_w, extraits en ID) et les registres destination (rd) présents dans les étages EX, MEM et WB.

Dans la version de base, l'équation booléenne de stall_w peut être simplifiée et, souvent, dans le design initial le signal est fixé à 0 pour ne pas introduire de blocage, ce qui signifie que le pipeline ne subit pas de stall même en cas de dépendances.

Q11 – Que se passe-t-il quand le signal stall_w est actif dans le control_path ?

Quand le signal stall_w est actif dans le **control_path**, le pipeline est temporairement bloqué :

1. **Instruction NOP injectée** : Une instruction neutre (32'h00000013, souvent une instruction "NOP") est injectée au stade de décodage pour éviter des comportements indésirables.
2. **Blocage de l'avancement** : Les instructions actuelles restent bloquées dans leur étage respectif, empêchant leur progression normale dans le pipeline.
3. **Gestion des conflits** : Ce mécanisme est conçu pour résoudre des dépendances de données ou conflits structurels. Toutefois, dans l'état actuel du design, stall_w est toujours désactivé (1'b0) et n'a donc aucun effet.

1.6. Caractéristiques des Mémoires

Q12 – Quelle taille a la mémoire de données (instance dmem) ?

La mémoire d'instructions/données est déclarée avec SIZE = 4096, chaque mot étant de 32 bits (4 octets). Ainsi, la capacité totale de dmem est :

4096×4= 16 384 octets soit **16 Ko**.

Q13 – Quelle est l'adresse de base de la mémoire de données ?

La mémoire de données (dmem) est mappée à l'adresse **0x0001_0000** selon les spécifications fournies dans les fichiers source (défini dans le SoC).

Q14 – Quelle taille a la mémoire d'instructions (instance imem) ?

De même que dmem, imem a SIZE = 4096 mots de 32 bits. La taille totale est donc également : 4096×4= 16 384 octets. Ce qui correspond bien à **16 Ko**.

Q15 – Quelle est l'adresse de base de la mémoire d'instructions ?

Imem est mappée à l'adresse **0x0000_0000**.

1.7. Simulation et Validation

Q16 – En examinant exo1.S et le contenu final des registres, le programme s'est-il déroulé correctement ?

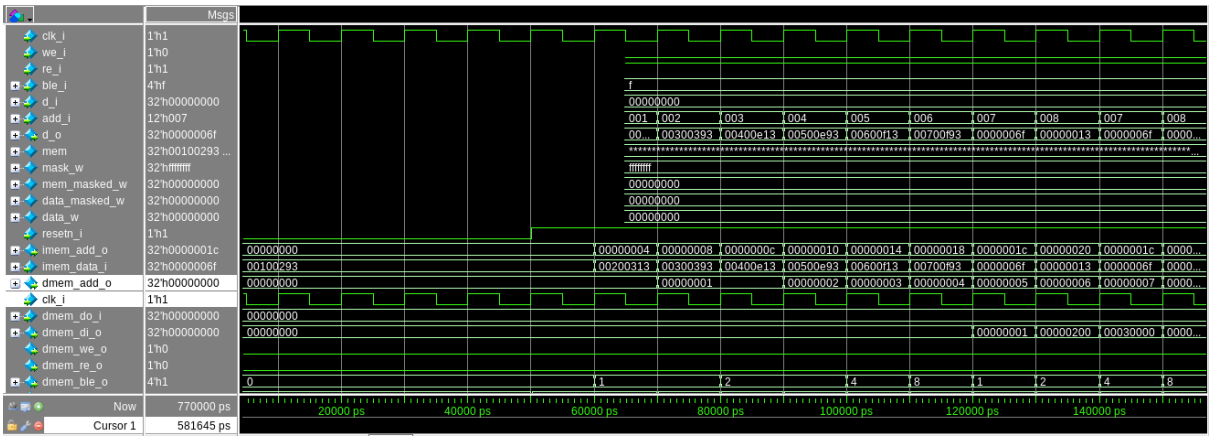


Figure 2 Résultat en sortie de exo1.S (Assembleur)

Pour le signal dmem_add_o, Les valeurs des registres correspondent aux attentes :

- On voit les instructions li t0,1, li t1,2, etc. qui indiquent que les registres ont été initialisés.

Le signal imem_add_o montre une exécution normale :

- On observe que l'adresse de la mémoire d'instruction évolue correctement, ce qui montre que les instructions sont bien récupérées.

Le PC semble boucler sur lab1 :

- Il y a un motif répétitif après l'exécution des li (période de 2 coups de clock) qui suggère une boucle infinie.

Q17 – Comment s'effectue l'arrêt de la simulation ?

Le programme exo1.S inclut un saut inconditionnel :

```
lab1 :  j  lab1
        nop

.end start
```

Figure 3 Saut inconditionnel exo1.S (NOP)

Ce saut crée une boucle infinie, ce qui empêche l'arrêt naturel de la simulation. L'arrêt doit donc être déclenché manuellement ou par une commande spécifique (ex. « run -all » suivie d'une commande d'arrêt).

Q18 – Écrire le fichier main.S qui exécute l'algorithme élémentaire (Algorithm 1)

Un exemple de fichier **main.S** correspondant à l'algorithme donné peut être rédigé ainsi :

```
.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
    nop
    sub t5, t3, t4

lab1:
    j lab1
    nop

.end start
```

Une instruction qui dépend du résultat d'une autre instruction doit **attendre que la première atteigne WB** pour lire la bonne valeur. En d'autres termes l'étape ID d'une ligne faisant appel à une variable précédente doit se dérouler après l'étape WB de ladite variable.

D'où la nécessité de 3 nop minimum pour l'étape d'addition et et l'étape de soustraction.

Figure 4 Code Assembleur main.S

Q19 – Simulation en sortie du code assembleur main.S

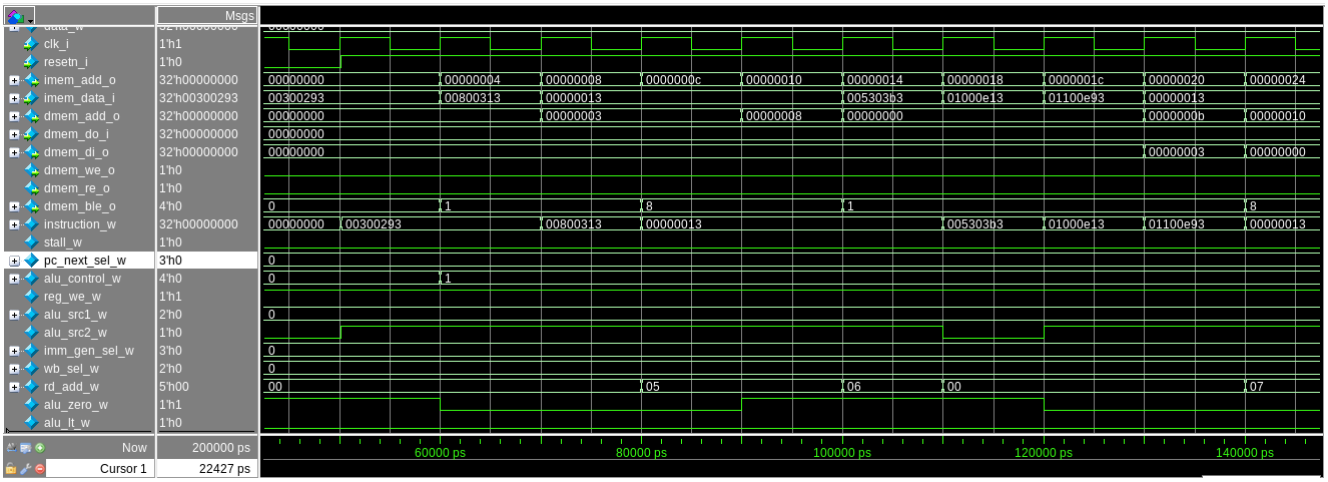


Figure 5 Résultat en sortie de main.S (Assembleur)

Comme on peut le remarquer dans la simulation ci-dessus, celle-ci réalise parfaitement ce que le code assembleur main.S avait donné pour instructions de réaliser. Cela vient du fait d'avoir pris en compte les différentes dépendances liés à notre algorithme assembleur directement dans notre programme assembleur. Notamment, en ajoutant le nombre minimal de NOP nécessaire pour prendre en compte correctement chaque dépendance

2. TD2 – Gestion des Dépendances et Correction d'un Programme de Multiplication

2.1. Modification et Résultat Attendu

Q1 – Modifier le fichier Mult.S pour réaliser la multiplication

```
.section .start
.globl _start

_start:
    li t0, 0x8       # Charger l'opérande 1 (multiplicande) dans t0
    li t1, 0x7       # Charger l'opérande 2 (multiplicateur) dans t1
    li t2, 16        # Initialisation du compteur de boucle à 16
  (multiplication sur 16 bits)
    li t3, 0         # Initialisation de l'accumulateur du résultat
    nop

loop:
    andi t4, t1, 1   # Tester le bit de poids faible de t1
    nop
    nop
    nop
    beqz t4, skip    # Si LSB == 0, ne pas additionner t0 à t3
    add t3, t3, t0   # Additionner t0 à l'accumulateur si LSB == 1

skip:
    slli t0, t0, 1   # Décalage à gauche du multiplicande (t0 <<= 1)
    srli t1, t1, 1   # Décalage à droite du multiplicateur (t1 >>= 1)
    addi t2, t2, -1  # Décrémenter le compteur
    nop
    nop
    nop
    bnez t2, loop    # Répéter jusqu'à ce que t2 atteigne 0

lab1:
    j lab1          # Boucle infinie pour arrêter l'exécution
    nop

```

Figure 6 Code Assembleur Mult.S

Le fichier **Mult.S** est modifié pour :

- Utiliser **t0** (x5) et **t1** (x6) pour stocker les deux opérandes (par exemple, 0x8 et 0x7 respectivement).
- Utiliser **t2** (x7) comme compteur pour itérer 16 fois.
- Utiliser **t3** (x28) comme accumulateur pour additionner les résultats partiels. Ainsi, le programme implémente une boucle qui, pour chaque itération, vérifie le bit de poids faible de t1, ajoute t0 à t3 si ce bit vaut 1, puis décale t0 à gauche et t1 à droite.

De la même manière que dans le TD précédent avec main.S, la présence de NOPs est justifié par une gestion logicielle des dépendances.

Q2 – Quel est le résultat normalement attendu du registre t3 ?

Pour la multiplication de 0x8 par 0x7, le résultat attendu dans t3 est **0x38** (56 en décimal).

2.2. Analyse des Problèmes et Simulation

Q3 – Simulation : Quel résultat est obtenu et pourquoi ?

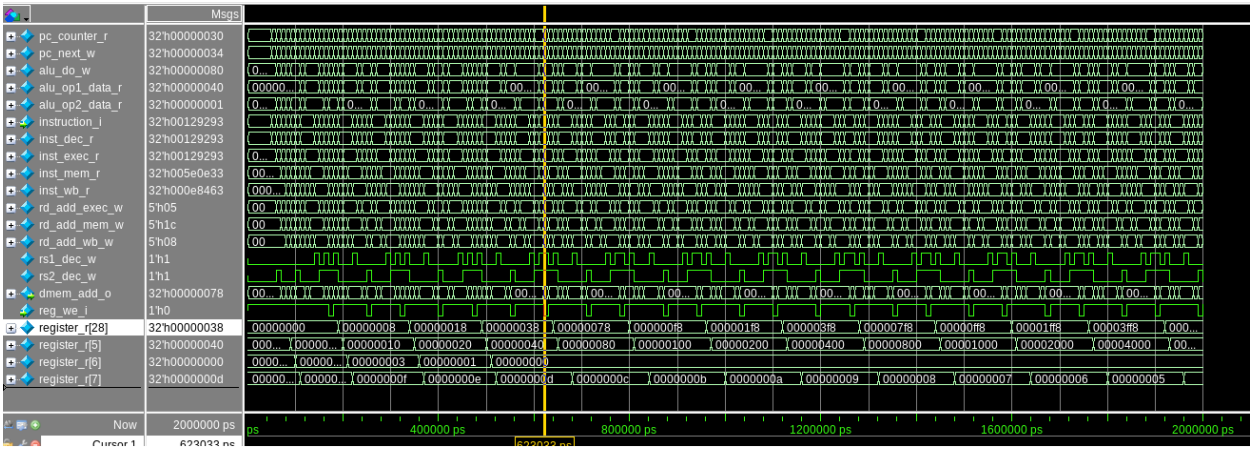


Figure 7 Résultat Initial en sortie de Mult.S (Assembleur)

Le programme est censé recommencer voir la valeur de t3 finir à 0x38, et dès que t3 atteint cette valeur, le registre 28 associé à t3 garde pour valeur 0x38 indéfiniment comme prévu (registre 28= 0x38). Cependant ce n'est pas le cas, la valeur de t3 diverge. Cela est sûrement dû à des problèmes de dépendance (d'un ou plusieurs des types suivants : Dépendance de données ou Dépendance de contrôle ou encore Dépendance structurelle).

Q4 – Quels étages du pipeline sont concernés par les dépendances identifiées ?

Si nous n'avions pas ajouté de nops initialement pour lutter face à la dépendance de données, nous aurions eu des dépendances concernant principalement les étages suivants du pipeline :

1. **ID → EX** (Instruction Decode → Execute) :
    - Dépendances de données (RAW) entre li, andi, beqz, addi et bnez.
    - L'opérande n'est pas encore calculé au moment où il est lu.

2. **EX → MEM → WB** (Execute → Memory → Write Back) :
    - La valeur produite en EX doit être écrite en WB avant d'être lue dans ID, causant un retard.
3. **IF → ID** (Instruction Fetch → Decode) :
    - Dépendance de contrôle après beqz, nécessitant un vidage du pipeline si le branchement est mal anticipé.

En résumé, les dépendances impactent **ID, EX, MEM et WB**, avec des retards dus à l'absence de forwarding et à la gestion du branchement.

2.3. Solutions Logicielles : Insertion de NOP

Q5 – Comment coder une instruction NOP en RISC-V ?

Un NOP en RISC-V se code de la manière suivante :

```
addi x0, x0, 0   # Instruction qui n'a aucun effet, codée en 0x00000013
```

La pseudo-instruction nop est généralement traduite en cette forme.

Q6 – Utilisation du NOP pour éliminer les dépendances – Type et nombre requis

Les dépendances à éliminer sont des dépendances de données et une dépendance de contrôle.

- Dépendances de données :
    - Entre li t1, 0x7 et andi t4, t1, 1
    - Entre andi t4, t1, 1 et beqz t4, skip
    - Entre addi t2, t2, -1 et bnez t2, loop
- Dépendance de contrôle :
    - Entre beqz t4, skip et add t3, t3, t0

Nombre d'instructions NOP nécessaires :

- Sans forwarding, il faut insérer 3 NOP entre chaque dépendance de données si les 2 instructions en question se suivent directement.
- Pour la dépendance de contrôle, il faut insérer 3 NOP après beqz t4, skip.

Q7 – Vérification par simulation avec insertion des NOP

Le code Mult.S après correction logicielle est donc :

```
.section .start
.globl _start

_start:
    li t0, 0x8       # (x5) Charger l'opérande 1 (multiplicande) dans t0
    li t1, 0x7       # (x6) Charger l'opérande 2 (multiplicateur) dans t1
    li t2, 16        # (x7) Initialisation du compteur de boucle à 16 (multiplication sur 16 bits)
    li t3, 0         # (x28) Initialisation de l'accumulateur du résultat
    nop

loop:
    andi t4, t1, 1   # Tester le bit de poids faible de t1
    nop
    nop
    nop
    beqz t4, skip    # Si LSB == 0, ne pas additionner t0 à t3
    nop
    nop
    add t3, t3, t0   # Additionner t0 à l'accumulateur si LSB == 1

skip:
    slli t0, t0, 1   # Décalage à gauche du multiplicande (t0 <<= 1)
    srli t1, t1, 1   # Décalage à droite du multiplicateur (t1 >>= 1)
    addi t2, t2, -1  # Décrémenter le compteur
    nop
    nop
    nop
    bnez t2, loop    # Répéter jusqu'à ce que t2 atteigne 0

lab1:
    j lab1          # Boucle infinie pour arrêter l'exécution
    nop

.end start
```

Figure 8 Code Assembleur Mult.S - Après Correction Logicielle

La simulation associée en sortie est alors :

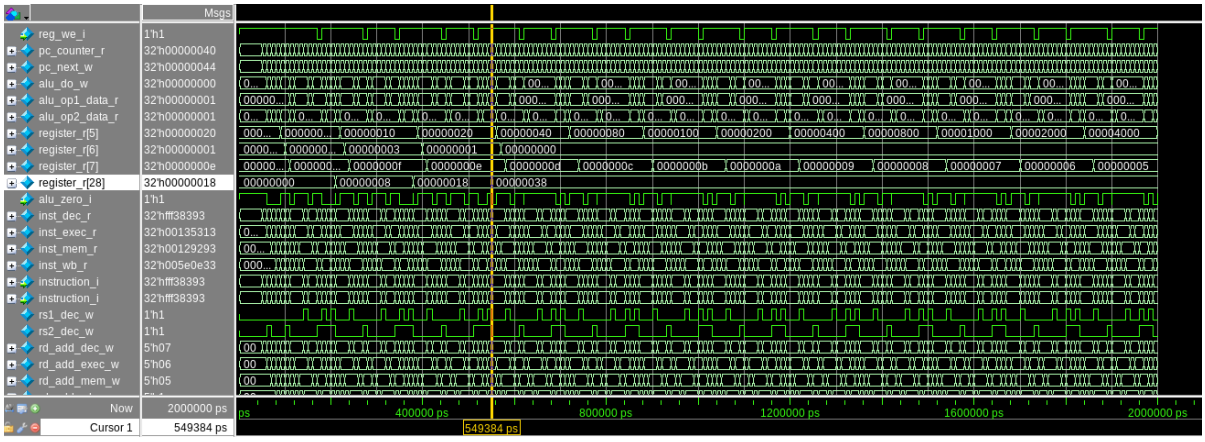


Figure 9 Résultat en sortie de Mult.S (Assembleur) - Après Correction Logicielle

On remarque donc que la valeur de t3 (registre 28) converge vers 0x38 (le résultat correct de notre multiplication) et reste constante en 0x38 comme nous le souhaitons. C'est la preuve que les dépendances de données et la dépendance de contrôle ont été gérés correctement en ajoutant le nombre minimal de nop nécessaires à chaque étape. Et que le résultat est donc maintenant conforme aux attentes

2.4. Analyse des Signaux de Dépendances

Q8 – Dans le control_path, que représentent rs1_dec_w et rs2_dec_w ?

Les signaux **rs1_dec_w** (instruction_i[19:15]) et **rs2_dec_w** (instruction_i[24:20]) représentent les indices des registres source (rs1 et rs2 respectivement) de l'instruction en cours, extraits lors de l'étape ID (Instruction Decode). Ils sont utilisés pour lire les valeurs correspondantes dans le banc de registres avant l'exécution (EX).

Q9 – Que représente le signal rd_add_XXX_w dans les étages EX, MEM et WB ?

Dans le control path, les signaux rd_add_XXX_w aux différentes étapes du pipeline (ID, EXE, MEM et WB) représentent l'indice du registre destination (rd), qui est extrait à partir du champ instruction_i[11:7] lors de l'étape de décodage (ID).
rd_add_dec_w est généré durant l'étape de décodage (ID) et contient l'indice du registre destination (rd), qui recevra potentiellement un résultat plus tard dans le pipeline.
Ce signal est ensuite propagé à l'étage EXE sous le nom de rd_add_exec_w, où il est transmis à l'ALU, qui peut l'utiliser pour identifier le registre cible d'une opération arithmétique ou logique.
Après l'exécution, rd_add_mem_w conserve cet indice lorsqu'un accès mémoire est nécessaire, notamment pour les instructions Load où le registre rd devra recevoir une valeur depuis la mémoire.
Enfin, rd_add_wb_w est utilisé lors de l'étape WB pour s'assurer que le bon registre est mis à jour avec le bon résultat final dans le banc de registres.

Ainsi, ces signaux assurent le suivi et l'écriture correcte des résultats des instructions à travers les différentes étapes du pipeline.

2.5. Implémentation d’un Interlock Matériel (stall_w)

Q10 – Écrire l'équation booléenne générant stall_w et décrire son effet

Une version simple de l’équation est la suivante :

stall_w = ((rs1_dec_w == rd_add_exec_w) || (rs2_dec_w == rd_add_exec_w)) && reg_we_exec _w || ((rs1_dec_w == rd_add_mem_w) || (rs2_dec_w == rd_add_mem_w)) && reg_we_mem_w ((rs1_dec_w == rd_add_wb_w) || (rs2_dec_w == rd_add_wb_w)) && reg_we_wb_w;

Quand stall_w est actif, le pipeline est bloqué pour éviter une dépendance de données. L'instruction en **ID** est remplacée par une **NOP**, le **PC ne progresse pas**, et l'exécution attend que la donnée soit disponible avant de continuer. Cela évite les erreurs dues aux dépendances **RAW (Read After Write)**.

Q11 – Toutes les instructions utilisent-elles rs1 ? Qualifier l’équation

Non, toutes les instructions n'utilisent pas rs1. Par exemple, certaines instructions comme les instructions de type U (LUI, AUIPC) et les sauts inconditionnels (JAL) ne lisent pas rs1.

rs1_dec_w n'est ainsi pas toujours pertinent car certaines instructions ne l'utilisent pas. Il faut donc ajouter une condition pour vérifier que l'instruction en ID utilise rs1 avant de le comparer :

- stall_w = ((use_rs1_dec && (rs1_dec_w == rd_add_exec_w)) || (rs2_dec_w == rd_add_exec_w)) && reg_we_exec_w || ((use_rs1_dec && (rs1_dec_w == rd_add_mem_w)) || (rs2_dec_w == rd_add_mem_w)) && reg_we_mem_w || ((use_rs1_dec && (rs1_dec_w == rd_add_wb_w)) || (rs2_dec_w == rd_add_wb_w)) && reg_we_wb_w;

Où use_rs1_dec est un signal qui vaut 1 uniquement pour les instructions qui utilisent rs1.

Ajouter une vérification use_rs1_dec permet donc d'éviter des dépendances inutiles sur rs1. Cela optimise ainsi la pipeline en réduisant les stalls superflus pour les instructions qui n’ont pas besoin de rs1.

Q12 – Toutes les instructions utilisent-elles rs2 ? Qualifier l'équation

De manière analogue, non, toutes les instructions n'utilisent pas rs2. Un exemple serait les instructions de type I (comme ADDI, LW, JALR) et les instructions de saut (JAL) qui n'ont pas de second opérande rs2.

rs2_dec_w n'est donc pas toujours pertinent, car certaines instructions ne l'utilisent pas. Nous devons donc ajouter une condition pour vérifier que l'instruction en ID utilise rs2 avant de le comparer :

stall_w = ((use_rs1_dec && (rs1_dec_w == rd_add_exec_w)) || (use_rs2_dec && (rs2_dec_w == rd_add_exec_w))) && reg_we_exec_w || ((use_rs1_dec && (rs1_dec_w == rd_add_mem_w)) || (use_rs2_dec && (rs2_dec_w == rd_add_mem_w))) && reg_we_mem_w || ((use_rs1_dec && (rs1_dec_w == rd_add_wb_w)) || (use_rs2_dec && (rs2_dec_w == rd_add_wb_w))) && reg_we_wb_w;

Où use_rs2_dec est un signal qui vaut 1 uniquement pour les instructions qui utilisent rs2.

Cet ajout de use_rs2_dec permet d'éviter des stalls inutiles pour les instructions qui n'utilisent pas rs2. Cela améliore l'efficacité du pipeline en réduisant les blocages inutiles.

Q13 – Toutes les instructions écrivent-elles dans rd ? Qualifier l'équation

Non, toutes les instructions n'écrivent pas dans rd. Par exemple, les instructions de saut conditionnel (BEQ, BNE, BLT, etc.) et les instructions de stockage (SW) n'écrivent pas de résultat dans un registre de destination.

Dans l'équation initiale Q10, reg_we_exec_w, reg_we_mem_w et reg_we_wb_w indiquent si l'instruction en EXE, MEM ou WB écrit dans rd, mais ces signaux sont **activés pour toutes les instructions, même celles qui n'écrivent pas dans rd**. Ajouter une condition permet alors de vérifier que l'instruction en EXE, MEM et WB **écrit dans rd** avant de comparer avec rs1_dec_w et rs2_dec_w :

stall_w = ((use_rs1_dec && (rs1_dec_w == rd_add_exec_w)) || (use_rs2_dec && (rs2_dec_w == rd_add_exec_w))) **&& reg_we_exec_w && writes_rd_exec** || ((use_rs1_dec && (rs1_dec_w == rd_add_mem_w)) || (use_rs2_dec && (rs2_dec_w == rd_add_mem_w))) **&& reg_we_mem_w && writes_rd_mem** || ((use_rs1_dec && (rs1_dec_w == rd_add_wb_w)) || (use_rs2_dec && (rs2_dec_w == rd_add_wb_w))) **&& reg_we_wb_w && writes_rd_wb;**

Où writes_rd_exec, writes_rd_mem et writes_rd_wb sont des signaux qui indiquent si l'instruction de l'étage correspondant **écrit dans rd**.

Q14 – Tester la solution avec le programme d'origine (sans NOP pour données) et comparer

D'après la question 4, les dépendances de données apparaissent entre les instructions suivantes :

- **Entre** li t1, 0x7 **et** andi t4, t1, 1
- **Entre** andi t4, t1, 1 **et** beq t4, zero, .skip_add
- **Entre** addi t2, t2, -1 **et** bne t2, zero, .loop

Donc en supprimant les NOP destinés à résoudre les dépendances de données (tout en conservant ceux pour les dépendances de contrôle), Mult.S devient :

```
.section .start
.globl _start

_start:
    li t0, 0x8       # (x5) Charger l'opérande 1 (multiplicande) dans t0
    li t1, 0x7       # (x6) Charger l'opérande 2 (multiplicateur) dans t1
    li t2, 16        # (x7) Initialisation du compteur de boucle à 16 (multiplication sur 16 bits)
    li t3, 0         # (x28) Initialisation de l'accumulateur du résultat

loop:
    andi t4, t1, 1   # Tester le bit de poids faible de t1
    beqz t4, skip    # Si LSB == 0, ne pas additionner t0 à t3
    nop
    nop
    add t3, t3, t0   # Additionner t0 à l'accumulateur si LSB == 1

skip:
    slli t0, t0, 1   # Décalage à gauche du multiplicande (t0 <<= 1)
    srli t1, t1, 1   # Décalage à droite du multiplicateur (t1 >>= 1)
    addi t2, t2, -1  # Décrémenter le compteur
    bnez t2, loop    # Répéter jusqu'à ce que t2 atteigne 0
    nop
    nop

lab1:
    j lab1          # Boucle infinie pour arrêter l'exécution
    nop

.end start
```

Figure 10 Code Assembleur Mult.S - Après Modification du Stall

Ce qui nous permet d'obtenir la simulation :

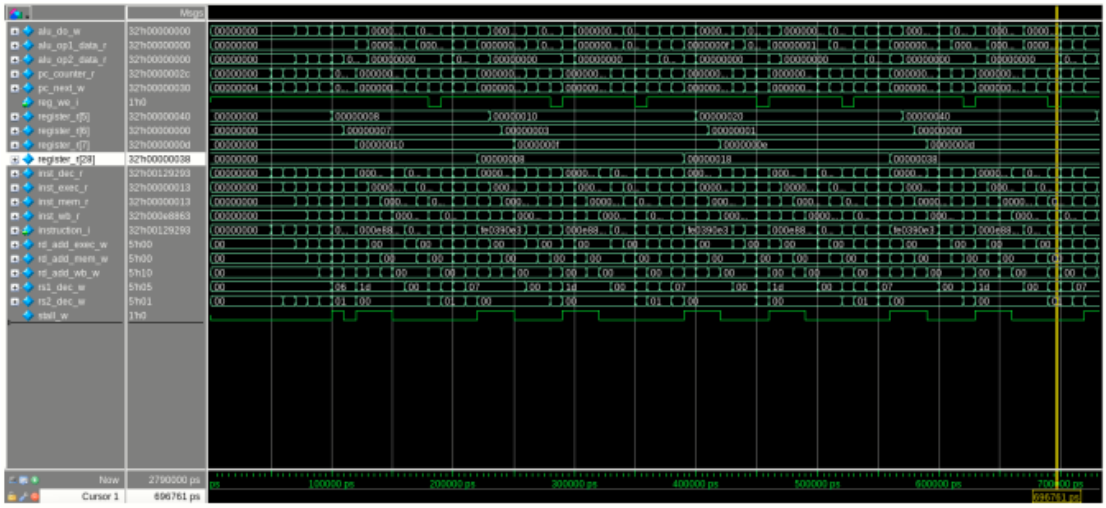


Figure 11 Résultat en sortie de Mult.S (Assembleur) - Après Correction Matérielle des Dépendances de Données

On remarque que les dépendances des données ont bien été gérés correctement de manière matérielle. Avec le registre 28 qui finit par stocker de manière constante la valeur 0x38.

2.6. Gestion des Sauts et des Branchements

Q15 – Gestion des sauts – Ajout de logique dans control_path

Les instructions de saut (JAL et JALR) modifient directement la valeur du compteur de programme (PC). Leur gestion nécessite un calcul rapide de l'adresse cible, sinon cela peut entraîner des dépendances de contrôle.

Pour gérer ces instructions, on introduit un signal jump_o dans le control_path, qui détecte la présence d'une instruction de saut :

assign jump_o = (opcode_dec_w == RV32I_I_INSTR_JALR) || (opcode_dec_w == RV32I_J_INSTR) ? 1'b1 : 1'b0;

Ce signal est ensuite utilisé dans le datapath pour invalider l'instruction suivante en insérant un NOP (32'h00000013) lorsqu'un saut est détecté :

assign inst_w = (jump_i == 1'b1) ? 32'h00000013 : imem_data_i;

Cela permet d'éviter l'exécution incorrecte d'une instruction après un saut et garantit que la mise à jour du PC se fait correctement.

Enfin, il n'est pas nécessaire de modifier la logique de stall (question Q10), car les sauts ne provoquent pas de dépendances de données nécessitant une mise en pause du pipeline.

Q16 – Tester la solution pour les sauts et comparer

Sans gestion matérielle des sauts et enlevant le NOP du lab1, on remarque l'apparition de bulles à la toute fin puis des signaux qui ne renvient plus rien de cohérent :

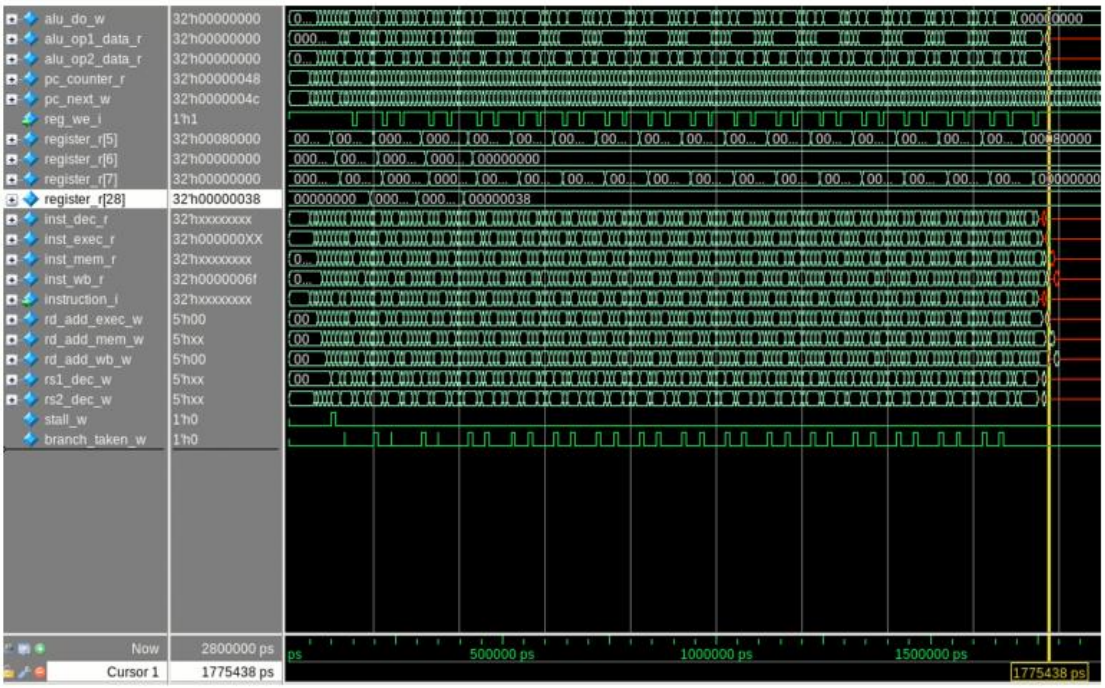


Figure 12 Résultat en sortie de Mult.S (Assembleur) - Sans Gestion matérielle des Sauts

Alors qu'en testant le programme d'origine sans insérer de NOP dans la partie lab1 de notre Mult.S. Mais avec une gestion des dépendances de contrôle relatives aux sauts (conservation de la logique ajoutée pour détecter les sauts), la simulation montre que le pipeline gère correctement le changement de PC. Les signaux inst_wb_r et stall_w indiquent qu'aucune interruption supplémentaire n'est nécessaire, et la séquence d'instructions correspond bien à celle obtenue avec l'insertion de NOP pour les sauts :

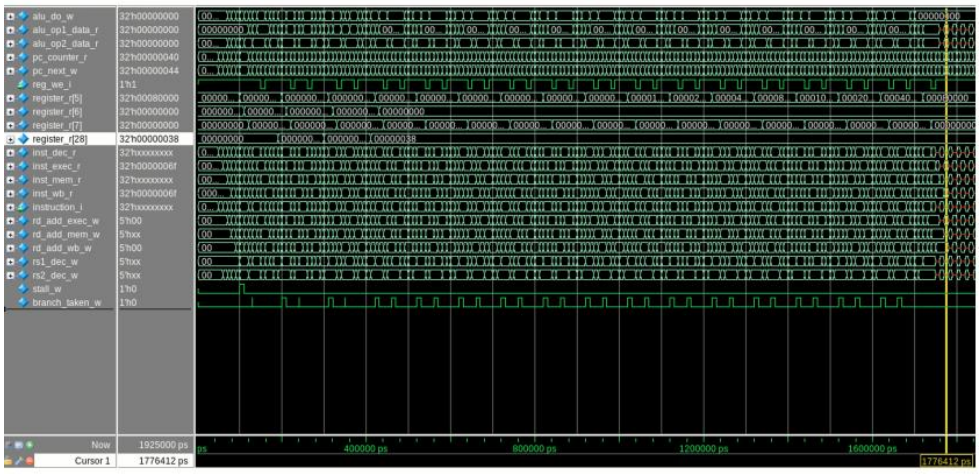


Figure 13 Résultat en sortie de Mult.S (Assembleur) - Après Gestion matérielle des Sauts

Q17 – Gestion des branchements – Ajout de logique dans control_path et data_path

D'après la question 4, la dépendance de contrôle concerne principalement la relation entre l'instruction BEQ (branchement conditionnel) et l'instruction ADD qui la suit. Pour gérer cette dépendance, il est nécessaire de prendre en compte le signal branch_taken_w, qui indique si un branchement a été pris.

Ainsi, pour gérer cette dépendance de contrôle, on modifie l'assignation de inst_dec_r pour injecter une instruction NOP (32'h00000013) lorsque soit stall_w est actif (pour les dépendances de données), soit branch_taken_w est actif (pour les dépendances de contrôle). Cela permet de vider le pipeline correctement en cas de branchement :

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

Q18 – Tester la solution pour les branchements et comparer

En simulant avec la nouvelle logique de branchement (sans NOP insérés pour ces dépendances), on observe via les signaux inst_wb_r et stall_w que le pipeline gère correctement les branchements. La séquence d'instructions obtenue est conforme à celle observée avec l'insertion de NOP pour le contrôle, confirmant ainsi l'efficacité de la solution, avec en sortie du registre 28 la valeur de convergence 0x38 :

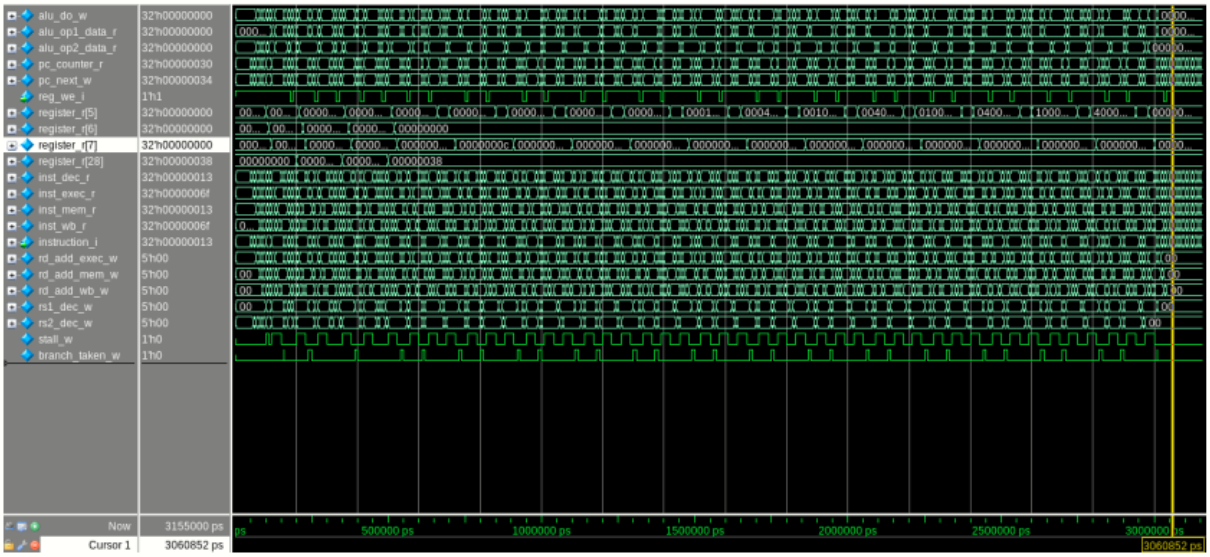


Figure 14 Résultat en sortie de Mult.S (Assembleur) - Après Correction Matérielle des Dépendances de Branchements

2.7. Implémentation d'un Bypass (Forwarding) – BONUS incomplet

Q19 – Bypass – Quelle dépendance est gérée par ce bypass ?

Le bypass (ou forwarding) permet de gérer la dépendance entre la sortie de l'ALU dans l'étage EX et une instruction qui en a immédiatement besoin dans son calcul (dépendance EX → EX). Ce mécanisme permet de transmettre directement le résultat sans attendre l'étape WB, réduisant ainsi le besoin d'insertion de NOP.

Q20 – Modifier alu_src1_mux_comb en conséquence

Q21 – Y a-t-il un impact sur stall_w ?

Oui, l'implémentation d'un bypass réduit le besoin de générer un signal stall_w pour certaines dépendances (notamment celles entre EX et EX). Par conséquent, le nombre de stalls (injections de NOP) diminue et le pipeline s'exécute plus efficacement.

Q22 – Écrire un petit programme en assembleur mettant en évidence cette dépendance

Un exemple de programme pouvant illustrer une dépendance résolue par bypass pourrait être :

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


start:
        add   t0, t1, t2   # t0 reçoit le résultat de l'addition de t1 et t2
        sub   t3, t0, t4   # t3 utilise immédiatement le résultat de t0

lab1 :  j  lab1
        nop

.end start
```

Dans ce cas, sans bypass, un stall serait nécessaire. Avec le bypass, la valeur calculée par l'ALU est immédiatement transmise à l'instruction suivante.

2.8. Conclusion sur l'Efficacité des Solutions

Q23 – Conclusion sur l'efficacité de toutes ces solutions

- **Insertion de NOP :** Simple à implémenter, mais pénalise les performances en introduisant des cycles morts.
- **Interlock (stalls calculés via stall_w) :** Permet de bloquer le pipeline uniquement lorsque cela est nécessaire, mais la détection doit être précise.
- **Bypass (forwarding) :** La solution la plus performante, car elle transmet directement les résultats entre étages et réduit considérablement les stalls.

La combinaison de ces techniques, en particulier l'implémentation du bypass couplée à une gestion intelligente des stalls, permet d'optimiser le débit du pipeline tout en garantissant la correction des dépendances.

3. TD3 – Implémentation d'une Mémoire Cache d'Instructions

3.1. Introduction et Schéma Global

Q0 – Établir un schéma du RV32i_soc et déterminer la largeur du bus entre imem et cache_instruction

Le schéma du RV32i_soc comporte :

- **RV32i_core (RV32i_top)** : le cœur du processeur contenant datapath et control_path,
- **cache_instruction** : placé entre RV32i_core et imem pour accélérer l'accès aux instructions,
- **imem** : mémoire d'instructions qui fournit des lignes complètes.

La largeur du bus de données entre imem et cache_instruction est de **128 bits** (correspondant à une ligne de 16 octets). Par ailleurs, dans RV32i_pipeline_top.sv, la nouvelle entrée **imem_valid_i** sert à indiquer que la donnée lue depuis imem est valide et peut être utilisée par le cache.

3.2. Cache Direct (Direct Mapped)

Q1 – Saisir les caractéristiques du cache dans les localparam

Les caractéristiques du cache direct sont définies par les paramètres suivants :

- **ByteOffsetBits = 4** : pour 16 octets par ligne.
- **IndexBits = 6** : ce qui donne $2^6 = 64$ lignes.
- **TagBits = 22** : car 32 – (4+6) = 22.
- **NrWordsPerLine = 4** : 4 mots de 32 bits chacun.
- **LineSize = 32 × 4 = 128 bits.**

Q2 – Quelles données doit-on stocker dans chaque ligne ? Précisez leur taille

Chaque ligne de cache doit contenir :

- Un **bit de validité** (1 bit) indiquant si la ligne contient des données valides.
- Le **tag** (22 bits) pour identifier l'adresse de la ligne.
- La **ligne de données complète** (128 bits, soit 16 octets) qui contient les mots d'instructions.

Q3 – Créer les registres de stockage

Les registres sont créés sous forme de tableaux pour modéliser la mémoire du cache. Par exemple, en SystemVerilog :

```
logic       valid    [NrLines-1:0];
logic [TagBits-1:0] tag_mem  [NrLines-1:0];
logic [LineSize-1:0] data_mem [NrLines-1:0];
```

Ces registres stockent respectivement le bit de validité, le tag et la ligne de données pour chaque ligne du cache.

Q4 – Implémenter le découpage de l'adresse en tag, index et offset

Pour une adresse 32 bits, le découpage est réalisé de la manière suivante :

- **Tag :** bits [31:10] (22 bits).
- **Index :** bits [9:4] (6 bits).
- **Offset :** bits [3:0] (4 bits).

Ce découpage permet d'identifier la ligne du cache (via l'index), de vérifier la correspondance de l'adresse (via le tag) et de sélectionner le mot dans la ligne (via l'offset).

Q5 – Créer un signal interne indiquant si la requête est hit ou miss

On définit un signal, par exemple :

wire hit = valid[addr_index] && (tag_mem[addr_index] == addr_tag);

Ce signal indique si, pour l'index donné, la ligne est valide et si le tag stocké correspond à celui extrait de l'adresse (hit) ou non (miss).

Q6 – Assigner les sorties de réponse pour le cas hit

En cas de hit, le cache extrait le mot demandé en utilisant l'offset (converti en indice de mot) et transmet cette donnée au processeur via le signal **read_word_o**, tout en activant **read_valid_o** pour indiquer que la donnée est valide.

Q7 – Assigner les sorties de requête à la mémoire en cas de miss

Lors d'un miss, le cache doit :

- Aligner l'adresse pour accéder à une ligne complète (en mettant les bits d'offset à 0).
- Activer le signal **mem_read_en_o**.
- Transmettre l'adresse alignée via **mem_addr_o** afin de récupérer la ligne de données depuis la mémoire.

Q8 – Ajuster les sorties de réponse pour transmettre le résultat de la mémoire

Quand la mémoire répond (signal **mem_read_valid_i** actif), le cache extrait le mot demandé (selon l'offset ou le word offset) et l'affecte à **read_word_o**, tout en activant **read_valid_o**. Ainsi, le processeur reçoit la donnée issue de la mémoire en cas de miss.

Q9 – Implémenter le stockage de la valeur récupérée dans le cache

Dès réception des données par la mémoire, le cache doit :

- Stocker la ligne complète dans le registre **data_mem** à l'index correspondant,
- Mettre à jour le **tag_mem** avec la partie tag de l'adresse,
- Affecter le bit de validité correspondant à 1 pour signaler que la ligne contient des données valides.

Q10 – Démontrer le bon fonctionnement du cache direct

La simulation du programme "Mult.S" doit montrer que :

- En cas de hit, le cache renvoie immédiatement le mot demandé au processeur.
- En cas de miss, le cache envoie la requête à la mémoire, récupère la ligne, met à jour sa mémoire interne et transmet ensuite la donnée au processeur.
  Les résultats observés en simulation confirment que le cache direct fonctionne correctement.

3.3. Deux Voies avec LRU

Q11 – Ajout des registres pour la deuxième voie et le LRU (cache associatif)

Pour transformer le cache en un cache associatif à deux voies, il faut déclarer de nouveaux registres pour la deuxième voie ainsi qu'un tableau de bits LRU pour chaque ensemble. Par exemple :

```
logic valid    [1:0][NrLines-1:0];
logic [TagBits-1:0] tag_mem  [1:0][NrLines-1:0];
logic [LineSize-1:0] data_mem [1:0][NrLines-1:0];
logic lru [NrLines-1:0];
```

Ici, la première dimension indique la voie (0 ou 1) et le signal **lru** indique, pour chaque index, la voie qui a été la moins récemment utilisée.

Q12 – Mettre à jour la condition de hit pour le cache à deux voies

Pour un cache à deux voies, on définit un hit pour chaque voie, par exemple :

```
wire hit_way0 = valid[0][addr_index] && (tag_mem[0][addr_index] == addr_tag);
wire hit_way1 = valid[1][addr_index] && (tag_mem[1][addr_index] == addr_tag);
wire hit = hit_way0 || hit_way1;
```

Le signal **hit** est activé si l'une des deux voies contient la donnée correspondant au tag et à l'index.

Q13 – Dans le cas d'un hit, quelle voie lire ?

Lorsqu'un hit est détecté, la voie à lire est celle pour laquelle le signal hit_wayX est vrai. Par exemple, si **hit_way0** est vrai, le cache lira la donnée de la voie 0 ; sinon, il la lira de la voie 1. Une logique conditionnelle (if-else) dans le code détermine cette sélection.

Q14 – Dans le cas d'un miss, quelle voie évincer ?

En cas de miss, on procède comme suit :

- Si l'une des deux voies est invalide, celle-ci est choisie pour être remplie.
- Si les deux voies sont valides, on utilise le bit **lru** pour choisir la voie la moins récemment utilisée, qui sera alors évincée et remplacée par la nouvelle ligne.

Q15 – Mise à jour du LRU après un accès

Après un accès (hit ou après rechargement en cas de miss), le bit **lru** associé à l'ensemble est mis à jour afin d'indiquer que la voie utilisée devient la plus récente (MRU). Par exemple, si la voie 0 est utilisée, on peut mettre lru[index] à 1 pour indiquer que la voie 1 sera évincée en cas de besoin futur.

Q16 – Démontrer le bon fonctionnement du cache associatif à deux voies

Pour un comportement fonctionnel, la simulation (avec le programme "Mult.S") doit montrer que :

- Le cache détecte correctement les hits dans l'une ou l'autre voie.
- En cas de miss, la voie à évincer est choisie selon le critère LRU.
- La mise à jour des registres et du bit LRU fonctionne comme attendu. Les waveforms obtenues confirment que le cache associatif améliore le taux de hit et fonctionne correctement.

4. Conclusion

Ce compte-rendu présente une analyse complète et détaillée des trois TDs.

- **TD1** a permis d'analyser la structure hiérarchique du processeur RISC-V pipeliné (RV32i_soc, RV32i_top, imem, dmem), de comprendre le rôle de chaque sous-circuit et de suivre la propagation des instructions à travers les cinq étages du pipeline.
- **TD2** a mis en évidence les problèmes de dépendances (hazards RAW et de contrôle) dans le pipeline et a proposé plusieurs solutions : insertion de NOP, interlock via stall_w et bypass. La simulation a confirmé l'impact positif de ces solutions sur la correction des erreurs et l'amélioration du débit.
- **TD3** a conduit à l'implémentation d'une mémoire cache d'instructions, d'abord en mode direct mapped puis en mode associatif à deux voies avec gestion LRU, démontrant ainsi l'importance d'une hiérarchie mémoire bien conçue pour optimiser la performance du processeur.

Ces travaux illustrent de manière concrète les défis liés à l'architecture des processeurs pipelinés et les techniques d'optimisation associées, offrant une base solide pour la compréhension des systèmes modernes de traitement.