MINES Saint-Étienne
Une école de l'IMT

Institut Mines-Télécom

RAPPORT

Architecture des Processeurs 2

Compte rendu des travaux dirigés

DERRUAU Noam
Groupe 2— 2A
ISMIN
noam.derruau@etu.emse.fr

DEKEYSER Alexis
Groupe 2— 2A
ISMIN
alexis.dekeyser@etu.emse.fr

15 janvier 2026

Table des matières

Table des figures

Introduction

L'architecture RISC-V s'est imposée comme un standard incontournable grâce à son jeu d'instructions ouvert et modulaire. Ce rapport porte sur l'étude et l'optimisation d'un processeur 32 bits implémentant l'ISA RV32I. À l'exception des instructions de synchronisation et d'exception (`FENCE`, `ECALL` et `EBREAK`), l'ensemble du jeu d'instructions de base a été implémenté.

L'enjeu principal de ce travail est de transformer une exécution séquentielle en une architecture pipelinée performante, capable de traiter les instructions en parallèle tout en garantissant la cohérence des calculs.

Nos objectifs se déclinent selon trois axes :

- **Analyse architecturale :** Compréhension approfondie de l'architecture d'un processeur RISCV single-core pipeliné
- **Gestion des aléas :** Mise en place de mécanismes pour résoudre les dépendances de données et les dépendances de contrôle afin d'éviter de manuellement insérer des `NOP` dans notre programme.
- **Optimisation mémoire :** Conception et intégration d'un *cache d'instructions* en lecture seule afin de réduire les cycles d'attente lors des accès mémoire.

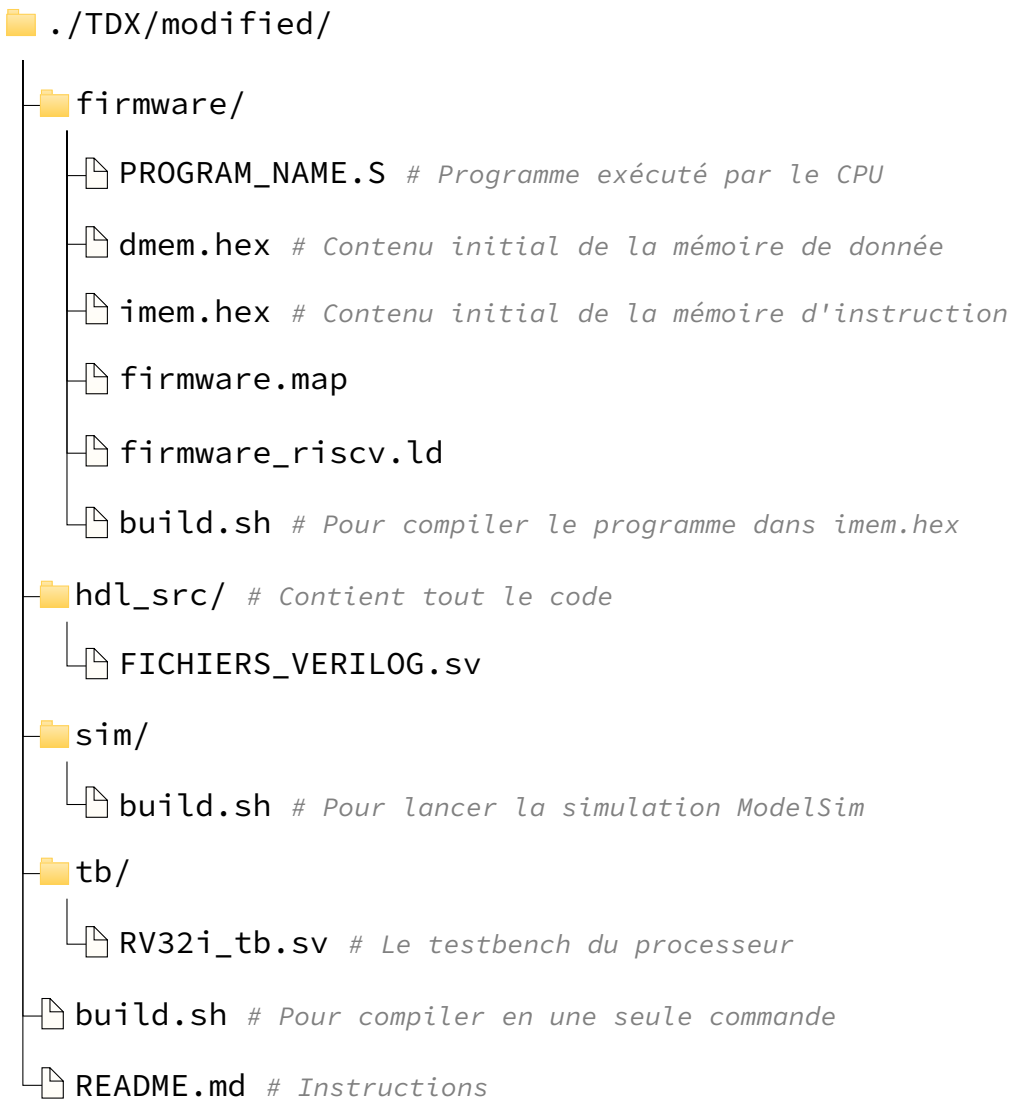
Précisions - Instructions pour les programmes Verilog

Ce rapport est accompagné d'un fichier `zip` contenant l'intégralité du code pour les trois TD. Vous pouvez retrouver une copie de l'intégralité du code au lien suivant :

https ://github.com/derruau/ismin-adp2-riscv

L'arborescence du projet est telle que :

FIGURE 1 : Arborescence du projet



Remarquez qu'avant d'accéder à cette arborescence, nous avons le choix entre les dossiers `base` et `modified` :

- **`base`** : contient le code tel qu'il était juste avant de commencer le TD.
- **`modified`** : contient le code après modification.

Pour les TD 2 et 3, plusieurs versions du code à différentes étapes du TD sont disponible. Ils se trouvent dans les dossiers : `./TD[X]/modified/hdl_src/Q[Y]/`.

De plus, vous pouvez compiler le code à n'importe quelle version de celui-ci en une seule commande lorsque vous êtes dans le dossier `./TD[X]/modified` :

```
# X peut prendre les valeurs suivantes pour les différents TD
# TD1: exo1, main
# TD2: Q3, Q10, Q12, Q14, Q15, Q17
# TD3: direct, associatif
# Ainsi, en faisant './build Q10', vous compilez et lancez
# directement la version du code telle qu'elle était à la
# question 10 dans le TD2.
./build [X]
```

Pour plus d'informations, n'hésitez pas à consulter le `README`.

TD 1

RV32I architecture pipeliné

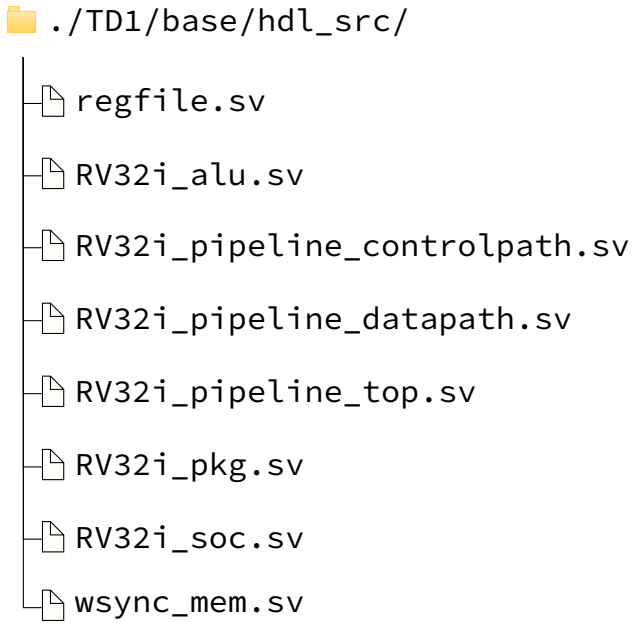
Afin de commencer l'étude du processeur, nous nous appuierons sur un projet de référence fourni sur ecampus. Cette première étape consiste à appréhender l'organisation modulaire du code et la hiérarchie des fichiers, calquée sur l'architecture matérielle du processeur. L'objectif est de valider le bon fonctionnement du socle initial et de maîtriser le flux de simulation avant d'entamer les phases de modification du pipeline et l'implémentation du cache.

1.1 Étude du processeur RISCV pipeliné

1.1.1 Découverte du processeur

L'architecture matérielle du processeur est définie par un ensemble de modules Verilog situés dans le dossier `./TD1/base/hdl_src/`. La structure se décompose comme suit :

FIGURE 1.1 : Fichiers source du processeur de référence



Nous allons tout d'abord essayer de comprendre les relations d'instanciation entre ces différents fichiers pour construire un premier schéma de l'architecture de ce processeur. En ouvrant les différents fichiers, nous observons ceci :

RV32i_soc.sv

```
1  RV32i_top RV32i_core (
2      ...
3  );
4
5  wsync_mem #(
6      ...
7  ) imem (
8      ...
9  );
10
11 wsync_mem #(
12     ...
13 ) dmem (
14     ...
15 );
```

RV32i_pipeline_top.sv

```
1  RV32i_datapath dp (
2      ...
3  );
4
5  RV32i_controlpath cp (
6      ...
7  );
```

Ainsi, nous constatons que le fichier `RV32i_soc.sv` contient deux instances du fichier `wsync_mem.sv` correspondant aux mémoires d'instruction et de données et une instance du fichier `RV32i_pipeline_top.sv`, qui contient une instance du *datapath* et du *controlpath*.

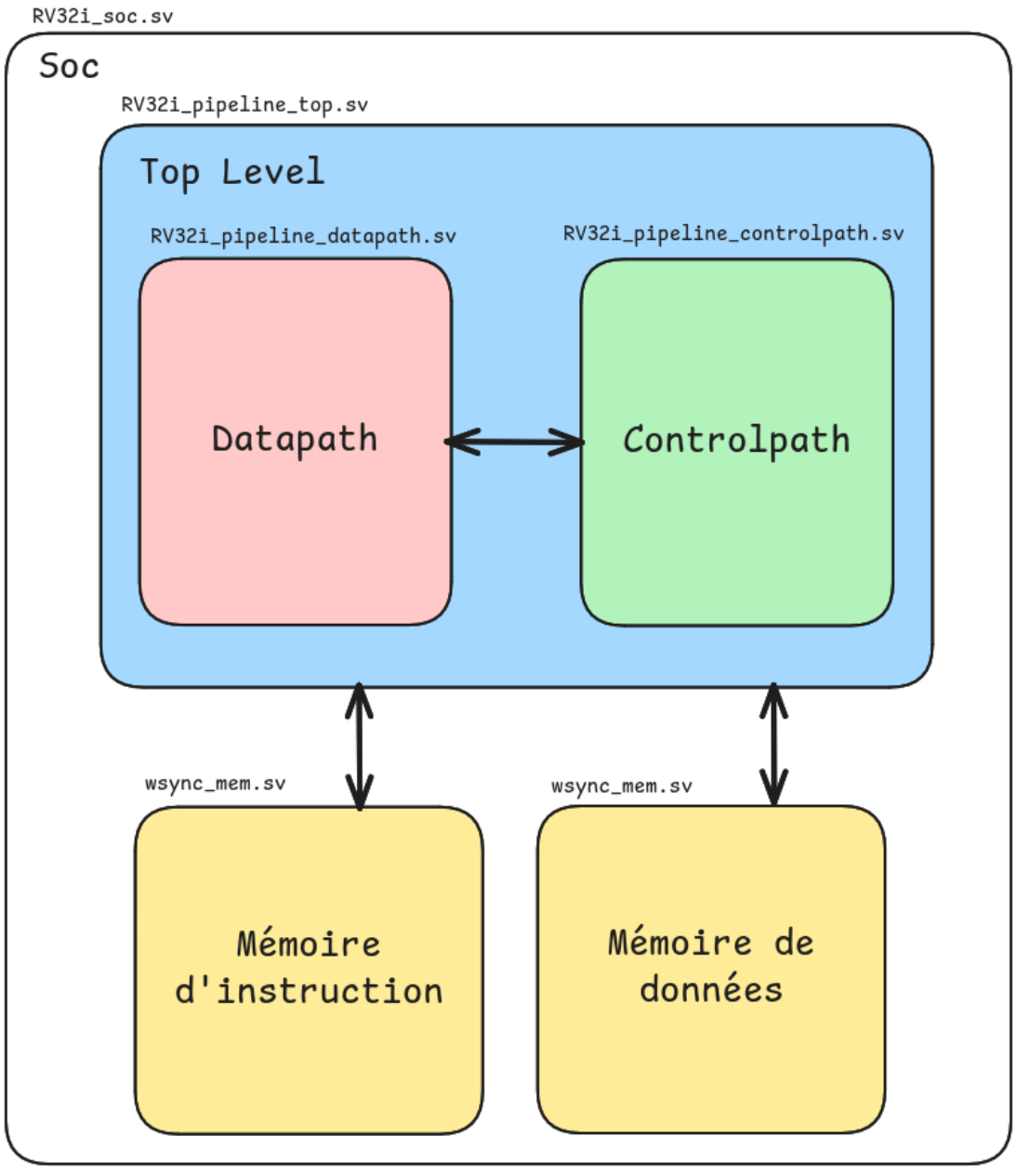
- **Le datapath** regroupe tous les circuits qui manipulent, transportent ou stockent les données binaires, par exemple l'ALU qui effectue les opérations et les registres qui stockent temporairement le résultat des opérations. C'est la partie qui exécute physiquement les opérations. Nous pou-

vons d'ailleurs voir ces circuits car dans le code de `RV32i_pipeline_datapath.sv`, nous retrouvons bien une instanciation de `RV32i_alu.sv` et `regfile.sv`.

- **Le controlpath** est le "cerveau" du processeur. Il décode les instructions et génère les signaux de commande nécessaires pour piloter le *datapath*. Il détermine, par exemple, quelle opération l'ALU doit effectuer ou si un registre doit être écrit ou lu à un instant t.

Muni de ces connaissances, nous pouvons faire un premier schéma de la hiérarchie des composants du processeur :

FIGURE 1.2 : Schéma de la hiérachie des composants du processeur



Examinons maintenant d'un peu plus près chacun de ces composants.

1.1.2 Examination du datapath

Commençons par examiner le *datapath* plus en détail. Nous remarquons que le datapath se découpe en **5 étages**, tous séparés par un registre qui contient l'instruction dans l'étage correspondant. Chaque étage est relié au suivant linéairement comme ceci :

FIGURE 1.3 : Les 5 étages du datapath

1. **FETCH** : Récupération de la prochaine instruction à insérer dans la pipeline.
2. **DECODE** : Décomposition des différentes parties de l'instruction selon son type.
3. **EXECUTE** : Exécution de l'instruction et calcul du résultat.
4. **MEMORY** : Stockage (optionnel) du résultat dans la mémoire de donnée.
5. **WRITE BACK** : Écriture dans le banc de registres du résultat.

Le pilotage du *datapath* par le *controlpath* repose sur une interaction bidirectionnelle. Si le *controlpath* orchestre l'exécution via des signaux de commande, il doit également recevoir des signaux de retour du *datapath* tels que les indicateurs de comparaison ou de statut pour adapter son comportement aux conditions dynamiques de l'exécution. D'après le code de `RV32i_pipeline_top.sv`, nous avons 4 signaux qui vont du *datapath* au *controlpath* :

RV32i_pipeline_datapath.sv

```
1  module RV32i_datapath (
2      ...
3      // Dec stage
4      output logic [31:0] instruction_o,  // Donne l'instruction
5                                          // actuellement dans
6                                          // l'étage DECODE au
7                                          // controlpath.
8      ...
9      // Exec stage
10     output logic        alu_zero_o, // | Flags de comparaison de
11     output logic        alu_lt_o,   // | l'ALU
12     output logic        alu_ltu_o,  // |
13     ...
14 );
```

De même, on peut identifier les signaux à destination de la mémoire d'instruction et de la mémoire de donnée :

Signaux à destination des mémoires

```
// À destination de la mémoire d'instruction
module RV32i_datapath (
    ...
    // Fetch stage
    output logic [31:0] imem_add_o, // Donne l'adresse de la
                                    // prochaine instruction à
                                    // fetch
    ...
);

// À destination de la mémoire de données
module RV32i_datapath (
    ...
    // Mem stage
    output logic [31:0] dmem_add_o, // L'adresse à écrire.
    output logic [31:0] dmem_di_o,  // les données à écrire.
    output logic [ 3:0] dmem_ble_o, // Le masque pour écire les
                                    // données.
    ...
);
```

1.1.3 Examination du controlpath

L'étude du code du *controlpath* montre que celui-ci suit la même logique que le pipeline. Pour fonctionner correctement, il reçoit l'instruction située à l'étage `DECODE` du *datapath*. Comme le processeur traite plusieurs instructions en même temps, le *controlpath* possède lui aussi une série de registres qui font circuler l'instruction de l'étage `DECODE` jusqu'à l'étage `WRITE BACK`. Cela permet au *controlpath* de connaître l'instruction présente dans chaque étage et d'envoyer les bons ordres au bon moment.

Système de pipleline du controlpath

```
module RV32i_controlpath (
    ...
    // Dec stage
    input  logic [31:0] instruction_i,
    ...
);
  ...
  assign inst_dec_w = stall_dec_i ? 32'h00000013 : instruction_i;
  ...
  // Exec stage
  always_ff @(posedge clk_i or negedge resetn_i) begin : exec_stage
    if (!resetn_i) begin
      inst_exec_r <= 32'h0;
    end else if (!stall_exec_i) begin
      inst_exec_r <= inst_dec_w;
    end
  end
  ...
  // Mem stage
  always_ff @(posedge clk_i or negedge resetn_i) begin : mem_stage
    if (!resetn_i) begin
      inst_mem_r <= 32'h0;
    end else if (!stall_exec_i) begin
      inst_mem_r <= inst_exec_r;
    end
  end
  ...
  // Write back
  always_ff @(posedge clk_i or negedge resetn_i) begin : wb_stage
    if (!resetn_i) inst_wb_r <= 32'h0;
    else inst_wb_r <= stall_exec_i ? 32'h00000013 : inst_mem_r;
  end
  ...
endmodule
```

Cet extrait de code illustre le mécanisme de transfert des instructions aux travers des registres `inst_dec_w`, `inst_exec_r`, `inst_mem_r` et `inst_wb_r` à chaque cycle d'horloge.

Le signal de stall

Une des difficultés majeure lors du passage d'une architecture monocycle à une architecture pipeliné est la gestion des dépendances. Les dépendances arrivent lorsqu'une instruction nécessite un résultat d'une autre instruction toujours dans la pipeline. Voici un exemple de dépendance de donnée :

Exemple de dépendance de données

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

start:
    addi  x3, x3, 10
    add  x4, x4, x3 # Dépendance de donnée avec x3
end:
    j end
    nop
    nop
```

Dans ce cas et sans gestion de dépendance de données, l'instruction `addi` se trouve à l'étage `DECODE` lorsque l'instruction `add` est exécutée et le registre `x3` n'est donc pas encore mis à jour.
Pour remédier à ce problème, une solution partielle consiste à introduire un signal, appelé signal de *stall* (`stall_o` dans le *controlpath*) qui stoppe une partie de la pipeline pour laisser le temps aux résultats dépendants de finir de traverser la pipeline.

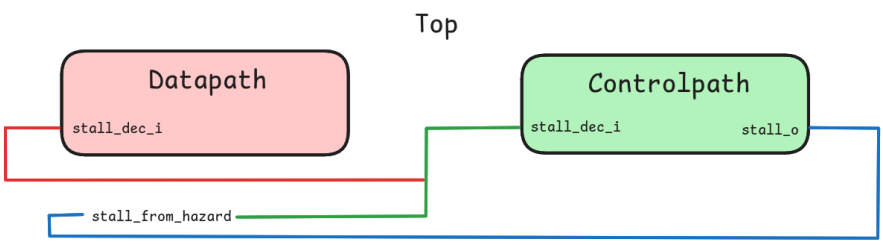
Pour l'instant, le signal de *stall* est toujours désactivé dans le *controlpath* :

RV32i_pipeline_controlpath.sv

```
assign stall_o = 1'b0;
```

Cependant, bien que les conditions d'activation du signal de *stall* n'aient pas été implémentés, son effet sur le *datapath* et le *controlpath* l'a été. Le signal de *stall* active dans le toplevel un signal appelé `stall_from_hazard` qui va lui même activer l'entrée `stall_dec_i` du *datapath* ET du *controlpath*. Ces signaux vont alors suspendre la partie `DECODE` du pipeline et laisser les instructions dans les parties suivantes continuer.

FIGURE 1.4 : Schéma de l'effet du signal de stall



1.1.4 Examination des mémoires

Les derniers composants de `RV32i_soc.sv` à examiner sont la mémoire d'instructions (`IMEM`) et la mémoire de données (`DMEM`). Ces mémoires sont des instances du même composant `wsync_mem.sv`. Voici comment elles sont définies :

Adresses de base

```
//
localparam IMEM_BASE_ADDR = 32'h0000_0000; // 0
localparam DMEM_BASE_ADDR = 32'h0001_0000; // 65536 en décimal
```

IMEM

```
assign imem_re_w = ire_w &
↪  imem_cs_w;
wsync_mem #(
    .SIZE(4096),
    .WS(0),
    .INIT_FILE(IMEM_INIT_FILE)
) imem (
    .clk_i(clk_i),
    .we_i(1'b0),
    .re_i(imem_re_w),
    .ble_i(4'b1111),
    .d_i(32'h0),
    .add_i(imem_add_w[11:2]),
    .d_o(imem_data_w),
    .valid_o(imem_valid_w)
);
```

DMEM

```
assign dmem_we_w = dwe_w &
↪  dmem_cs_w;
assign dmem_re_w = dre_w &
↪  dmem_cs_w;
wsync_mem #(
    .SIZE(4096),
    .WS(0),
    .INIT_FILE(DMEM_INIT_FILE)
) dmem (
    .clk_i(clk_i),
    .we_i(dmem_we_w),
    .re_i(dmem_re_w),
    .ble_i(ble_w),
    .d_i(dmem_di_w),
    .add_i(dmem_add_w[11:2]),
    .d_o(dmem_do_w),
    .valid_o(dmem_valid_w)
);
```

Ces mémoires ont une taille de 4096 mots chacune, comme indiqué par la ligne `.SIZE(4096)`. Ainsi, un programme ne peut pas faire plus de 4096 instructions sinon on ne peut pas le stocker complètement dans la mémoire et il ne peut pas non plus stocker plus de 4096 mots dans la mémoire de données.

Les adresses de base sont à 0 pour `IMEM` et 65536 pour `DMEM`. Ce choix permet de déterminer si une adresse appartient à la mémoire d'instruction ou la mémoire de données en ne regardant qu'un seul bit :

FIGURE 1.5 : Distinction des adresses IMEM et DMEM (en hexa)

IMEM	0	0	0	0	X_3	X_2	X_1	X_0
DMEM	0	0	0	1	X'_3	X'_2	X'_1	X'_0

1.2 Éxécution et simulation d'un programme

Maintenant que nous nous sommes familiarisé avec l'architecture de référence du processeur, nous allons le tester en faisant tourner quelques programmes.

1.2.1 Un premier programme

Le premier programme que nous allons tester nous est fourni est s'appelle `exo1.S`, voici son contenu :

exo1.S

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

start:
    li t0,1
    li t1,2
    li t2,3
    li t3,4
    li t4,5
    li t5,6
    li t6,7
lab1 :
    j  lab1      # Boucle infinie (signale la
    nop          # fin du programme

.end start
```

Ce programme n'est pas censé poser de difficultés au processeur puisqu'il n'y a aucune dépendance de données. Les opérations `li` sont des pseudo-instructions qui sont compilés comme ci-contre.

```
# Pseudo instruction
li rd, imm
# Instruction équivalente
addi rd, x0, imm
```

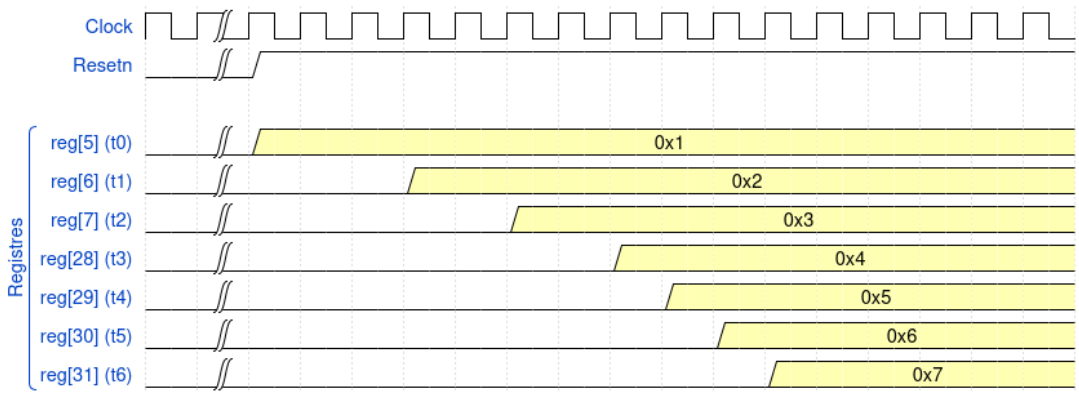
De plus, on voit que le programme finit par une boucle infinie d'instruction `J` suivie de `NOP`. Tous les programmes que nous écriverons pour ce processeur devront finir par cette boucle infinie. En effet, dans le testbench (`./TD1/modified/tb/RV32i_tb.sv`), il y a une boucle qui compte si cette séquence de `J` et de `NOP` se répète 5 fois d'affilé. Si c'est le cas, on stoppe le programme. Cette séquence d'instruction est donc un moyen de dire que la simulation est finie.

Code déterminant la fin de la simulation

```
do begin
    @(inst_w); // Attendre que inst_w change
    if (inst_w == 32'h0000006F) begin // Détection du J
        @(inst_w); // Attendre que inst_w change
        if (inst_w == 32'h00000013) i = i + 1; // Détection du NOP
        else begin
            @(inst_w);
            i = 0;
        end
    end
end while (i < 5); // Répétition 5 fois

$stop; // Arrêt de la simulation
```

Ainsi, après l'exécution du programme `exo1.S`, le résultat dans les registres est correct comme l'illustre le chronogramme ci-dessous.

FIGURE 1.6 : Chronogramme de `exo1.S`



Note : Vous pouvez voir les captures d'écran de toutes les simulations ModelSim en annexe en fin de rapport. Cliquez ici (Figure 3.6)

1.2.2 Un second programme

Le second programme de test est spécifiquement conçu pour introduire des dépendances de données. Son exécution vise à mettre en évidence les limites de l'architecture actuelle et à confirmer que le processeur, en l'absence de mécanismes de gestion des aléas, ne permet pas encore de garantir l'intégrité des résultats. Le code assembleur utilisé pour ce test se trouve ci-dessous :

FIGURE 1.7 : Code du second programme de test

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

Les dépendances de données observées proviennent d'un conflit de lecture/écriture : les résultats des chargements dans les registres`t0`, `t1`, `t3` et `t4` n'ont pas encore été stabilisés dans le banc de registres. En effet, alors que les instructions `add` et `sub` atteignent l'étage `EXECUTE`, les données dont elles dépendent transitent encore dans les étages `MEMORY` ou `WRITE BACK`. En l'absence de mécanisme de gestion des dépendances de données, le processeur utilise les valeurs obsolètes (égales à 0) présentes dans les registres, ce qui conduit à des calculs erronés. Les résultats de cette exécution sont détaillés ci-dessous :

FIGURE 1.8 : Résultats de l'exécution de `main.S`

Registre	Résultat Correct	Résultat Réel
t0 (x5)	0x03	0x03
t1 (x6)	0x08	0x08
t2 (x7)	0x0B	0x00
t3 (x28)	0x10	0x10
t4 (x29)	0x11	0x11
t5 (x30)	0x21	0x00

Note : . Accéder à la simulation ModelSim (Figure 3.7)

Comme attendu, le processeur n'a pas bien calculé les valeurs de `t2` et `t5`. Pour remédier à ce problème, nous allons donc commencer à implémenter la gestion des dépendances dans le prochain TD.

TD 2

Gestion des dépendances

L'étape précédente a mis en évidence les limites d'un pipeline sans mécanismes de contrôle : les dépendances de données mais aussi les ruptures du flux d'instructions compromettent l'intégrité de l'exécution. Cette deuxième partie est consacrée à la résolution de ces problématiques à travers l'implémentation de solutions logicielles et matérielles.

Nous travaillerons sur trois mécanismes de contrôle :

- **Résolution des dépendances de données** : Nous comparerons deux approches pour garantir la disponibilité des opérandes. D'une part, une méthode logicielle par insertion d'instructions NOP (No Operation), et d'autre part, une solution matérielle via l'implémentation d'un mécanisme d'Interlock.
- **Gestion des ruptures de flux** : Nous traiterons les dépendances de contrôle liées aux instructions de branchement et de saut, afin d'assurer la cohérence du *PC* lors des ruptures de séquence.
- **Synchronisation mémoire** : Enfin, nous intégrerons un mécanisme de *Wait State (WS)*. Ce dispositif permettra au processeur de se synchroniser avec la mémoire d'instruction en suspendant le pipeline jusqu'à la réception effective des instructions demandées.

2.1 Correction des dépendances

2.1.1 Correction logicielle

Nous allons commencer par essayer de corriger le problème des dépendances de façon logicielle en insérant des instructions `NOP` là où les dépendances peuvent

arriver.
Une instruction `NOP` (No OPeration) est une instruction dont le but est de ne rien faire du tout. On peut coder une instruction `NOP` sur l'architecture RISC-V en ajoutant 0 au registre `x0`. En effet, le registre `x0` est un registre spécial qui contient toujours 0. Voici l'équivalent de l'instruction NOP :

Instruction équivalente à un NOP

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

On distingue trois types de dépendances :

1. Les dépendances de données
2. Les dépendances de contrôle de type branchement
3. Les dépendances de contrôle de type saut

Le cas des dépendances de données

Les dépendances de données arrivent lorsque le résultat d'un calcul n'a pas eu le temps d'être mis à jour dans les registres et qu'une seconde opération dans l'étage `EXECUTE` nécessite le résultat de l'opération précédente, par exemple :

Exemple de dépendance de données

```
1 addi x8, x8, 1
2 addi x8, x8, 1
```

La première instruction `addi` doit d'abord passer par les étages de `MEMORY` et `WRITE BACK` avant de mettre à jour les registres.

Il faut donc 3 `NOP` pour gérer une dépendance de données :

```
1 addi x8, x8, 1
2 nop
3 nop
4 nop
5 addi x8, x8, 1
```

Le cas des dépendances de contrôle de type branchement

Les dépendances de contrôle de type branchement arrivent lorsque le flux d'exécution du programme est rompu, c'est à dire lorsqu'on prends un branchement. Voici un exemple pour mieux comprendre :

Exemple de dépendance de contrôle de type branchement

```
1 beq t0, t1, label  # Si t0 == t1, on saute au label
2 addi t2, t2, 1     # Instruction suivante "par défaut"
3 subi t3, t3, 1     # Instruction suivante "par défaut"
4 ...
5 label: sub t3, t4, t5
```

Pour un branchement, la prise de décision est faite à l'étage `EXECUTE`. Or, à ce moment, les instructions `addi` et `subi` juste après sont déjà dans la pipeline aux étages `DECODE` et `FETCH`. Même si on change le PC et que l'on prends bien le branchement pour executer l'opération `sub`, les instructions `addi` et `subi` seront quand même exécutées lors des prochains cycle d'horloge.

Il faut donc 2 `NOP` pour gérer une dépendance de contrôle de type branchement :

```
1 beq t0, t1, label
2 nop
3 nop
4 addi t2, t2, 1
5 subi t3, t3, 1
6 ...
7 label: sub t3, t4, t5
```

Le cas des dépendances contrôle de type saut

Pour finir, les dépendances de contrôle de type saut arrivent presque à cause des même raisons que les dépendances de branchement, cependant contrairement à ces dernières, les dépendances de contrôle de type saut font changer le PC à l'étage `DECODE`, donc seule une seule instruction est déjà dans la pipeline derrière elle.

Exemple de dépendance de contrôle de type saut

```
1 j label
2 addi x7,x7,1
3 ...
4 label: addi t3, t3, 1
```

Il faut donc 1 `NOP` pour gérer une dépendance de contrôle de type saut :

```
1 j label
2 nop
3 addi x7,x7,1
4 ...
5 label: addi t3, t3, 1
```

Nouveau programme à corriger

Pour ce deuxième TD, un nouveau programme à l'emplacement `./TD2/modified/firmware/exo2_unmodified.S` nous a été fourni. Une copie de ce programme est disponible en annexe (Figure 3.8). S'il est correctement exécuté, nous devons obtenir les valeurs suivantes dans nos registres :

FIGURE 2.1 : Valeurs des registres à la fin de `exo2.S`

Registre	Résultat Correct
`t0` (x5)	`0x0F`
`t1` (x6)	`0x10`
`t2` (x7)	`0x00`
`fp` (x8)	`0x02`

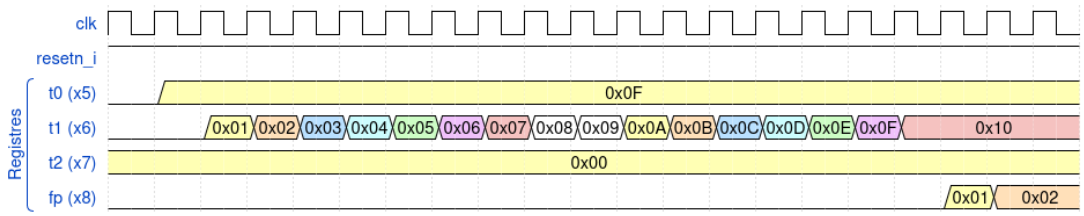
Comme nous l'avons mis en évidence auparavant, l'absence de traitement des dépendances conduit inévitablement à des erreurs d'exécution. Afin de garantir l'intégrité des calculs, nous présentons ici la version corrigée du programme intégrant des instructions `NOP`. Le nombre de cycles d'attente insérés entre chaque instruction critique respecte les délais établis lors de l'analyse des dépendances dans les sections précédentes.

Correction logicielle de `exo2.S`

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

start:
    addi x5,x0,15
    add  x6,x0,x0
    add  x7,x0,x0
    add  x8,x0,x0
lab1:
    j lab2          // Dépendance de contrôle de type J;
    nop             // On a besoin de 1 NOP
    addi x7,x7,1    //
lab2:
    addi x6,x6,1    //
    nop             // Dépendance de données:
    nop             // On a besoin de 3 NOP
    nop             //
    bge x5,x6,lab1  //
    nop             // Dépendance de contrôle avec instruction type
    ↪  B:
    nop             // On a besoin de 2 NOP
    addi x8,x8,1    //
    nop             // Dépendance de données:
    nop             // On a besoin de 3 NOP
    nop             //
    addi x8,x8,1    //
    nop             //
    nop             // Ces NOP sont ici avoir le temps de mettre le
    ↪  addi dans r8
    nop             //
lab3:
    j lab3          // Condition de stop du testbench:
    nop             // avoir 5 fois j lab3 suivi d'un nop

.end start
```

Voici là aussi le chronogramme de l'exécution de ce nouveau programme :

FIGURE 2.2 : Chronogramme de l'exécution de la correction logicielle



Note : . Accéder à la simulation ModelSim (Figure 3.9)

2.1.2 Correction Matérielle des dépendances de données

La correction logicielle n'est pas une solution souhaitable car avec cette solution un programme devrait être recompilé pour chaque type de processeur exis-

tant (qui ont chacun des pipelines différentes). Cela créerait vite des casse-tête énorme pour les développeurs de compilateurs. De plus, les exécutables seraient beaucoup plus lourds que ce qu'ils ne pourraient être. Ainsi, il y aurait beaucoup de gâchis de mémoire avec cette solution.
Il est donc préférable de corriger les dépendances de façon matérielle. Notre objectif dans cette partie sera donc d'écrire une équation booléenne correcte du signal `stall_o` du fichier `./TD2/modified/hdl_src/RV32i_pipeline_controlpath.sv` qui permet de stopper les étages `FETCH` et `DECODE` du *datapath*

Identification des opérandes et des destinations

La première étape consiste à extraire les adresses des registres impliqués dans chaque étage du pipeline. Dans l'étage `DECODE`, on identifie les sources potentielles via les signaux `rs1_dec_w` et `rs2_dec_w`. Simultanément, on surveille les adresses de destination (`rd`) des instructions plus avancées dans le pipeline (étages `EXE`, `MEM` et `WB`).

> Une dépendance brute existe si l'adresse d'une source en cours de décodage correspond à l'adresse d'une destination en cours d'exécution ou de sauvegarde.

Première écriture du `stall_o`

Nous créons deux signaux, `hazard_rs1` et `hazard_rs2` qui sont vrai lorsqu'il y a une dépendance de donnée sur `rs1` et `rs2` respectivement.
Alors, nous pouvons dire que nous devons activer le *stall* lorsqu'au moins un de ces signaux est vrai.

RV32i_pipeline_controlpath.sv

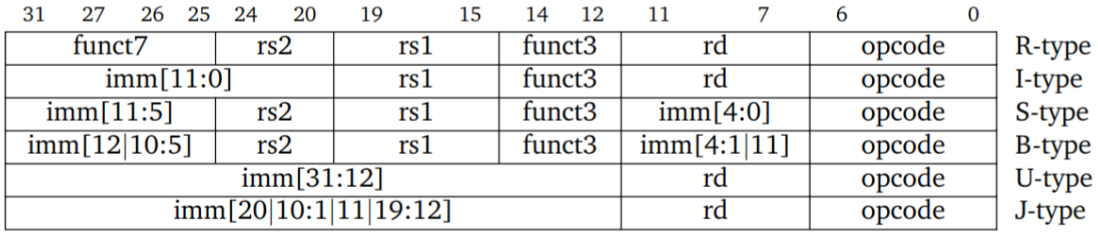
```
1  logic hazard_rs1;
2  logic hazard_rs2;
3
4  // Condition de stall sur rs1:
5  assign hazard_rs1 = rs1_dec_w == rd_add_exec_w || rs1_dec_w ==
   ↪  rd_add_mem_w || rs1_dec_w == rd_add_wb_w;
6
7  // Condition de stall sur rs2
8  assign hazard_rs2 = rs2_dec_w == rd_add_exec_w || rs2_dec_w ==
   ↪  rd_add_mem_w || rs2_dec_w == rd_add_wb_w;
9
10 assign stall_o = hazard_rs1 || hazard_rs2;
```

Cependant, comme nous allons le voir dans la prochaine partie, cette logique n'est pas du tout complète.

Correction du signal de stall

En effet, il existe des instructions RISCV RV32I qui ne contiennent pas les opérandes `rs1`, `rs2` ou `rd` :

FIGURE 2.3 : Types d'instructions de RV32i



31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Nous allons donc devoir créer cinq signaux supplémentaires :

- `rs1_re_dec_w` : qui est vrai lorsque l'opération à l'étage `DECODE` utilise `rs1`.
- `rs2_re_dec_w` : qui est vrai lorsque l'opération à l'étage `DECODE` utilise `rs2`.
- `rd_we_exec_w`, `rd_we_mem_w` et `rd_we_wb_w` : qui sont vrai lorsque l'instruction dans les étage `EXEC`, `MEMORY` ou `WRITE BACK` utilisent `rd`.

Pour ce faire, examinons les *opcodes* des instructions qui utilisent `rs1`, `rs2` et `rd` pour essayer de déterminer les différences entre les opérations qui utilisent ces opérandes et les autres.

FIGURE 2.4 : *Opcodes* et utilisation des opérandes

Type d'instruction	Opcode	`rs1` Utilisé	`rs2` utilisé	`rd` utilisé
R	0b0110011	✓	✓	✓
I	0b0010011	✓	✗	✓
I (bis)	0b0000011	✓	✗	✓
S	0b0100011	✓	✓	✗
B	0b1100011	✓	✓	✗
U	0b0110111	✗	✗	✓
J	0b1101111	✗	✗	✓

Pour `rs1` La différence entre un *opcode* qui utilise `rs1` et les autres est le bit de poids 2^2.

Définition de `rs1_re_dec_w`

```
logic rs1_re_dec_w;

// Les instructions qui utilisent rs1 ont ce bit à 0, les autres à 1
assign rs1_re_dec_w = !opcode_dec_w[2];
```

Pour `rs2` La différence entre un *opcode* qui utilise `rs2` et les autres se trouve dans le fait qu'un *opcode* qui utilise `rs2` a les bit de poids 2^2 à 0 et le bit de poids 2^5 à 1.

Définition de `rs2_re_dec_w`

```
logic rs2_re_dec_w;

// Les instructions qui utilisent rs1 ont ce bit à 0, les autres à 1
assign rs2_re_dec_w = !opcode_dec_w[2] && opcode_dec_w[5];
```

Pour `rd` Les *opcodes* qui utilisent `rd` sont les seuls à avoir la séquence `0b100` entre les bits de poids 2^5 et 2^3.

Définition de `rd_we_XXX_w`

```
// Les rd_we_XXX_w sont à 1 lorsqu'une instruction dans XXX utilise
// rd.
logic rd_we_exec_w, rd_we_mem_w, rd_we_wb_w;

// Fonction qui retourne 1 si l'instruction utilise rd, 0 sinon.
function logic rd_used(input logic [31:0] inst);
    // Cas spécial pour les instructions qui utilisent R0.
    // On DOIT les rendre non-bloquants sinon ils peuvent
    // créer un stall infini.
    //
    //              RD     !=     R0       &&   OPCODE[5:3] != 3'b100
    rd_used = ( inst[11:7] != 7'b0000000 ) && (  inst[5:3]  != 3'b100 );
endfunction

// On applique la fonction rd_used aux signaux rd des différents
// stages de la pipeline.
assign rd_we_exec_w = rd_used(inst_exec_r);
assign rd_we_mem_w = rd_used(inst_mem_r);
assign rd_we_wb_w = rd_used(inst_wb_r);
```

Écriture finale de l'expression de `stall_o` pour les dépendances de données

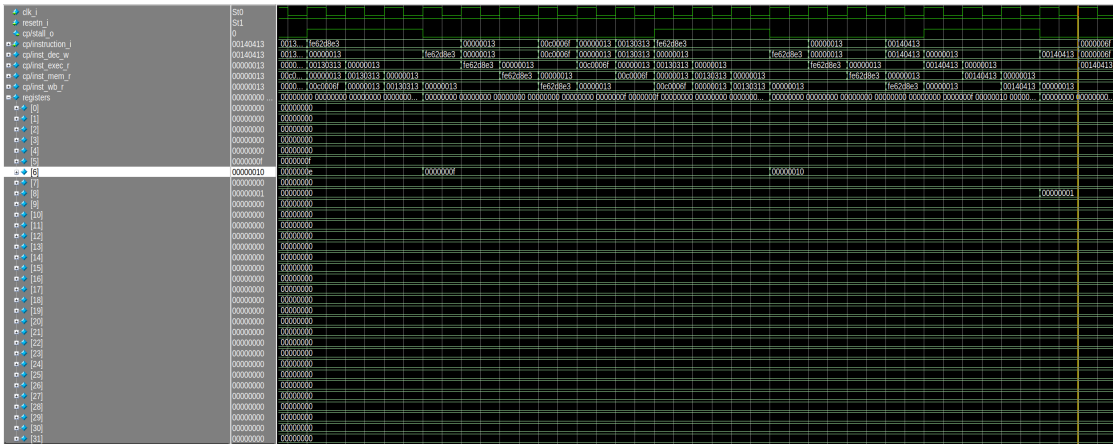
En combinant les extraits de code des parties précédentes, nous arrivons à cette expression pour `stall_o` :

FIGURE 2.5 : Expression de `stall_o`

```
// Les rd_we_XXX_w sont à 1 lorsqu'une instruction dans XXX utilise rd
// Les hazard_rsX sont à 1 lorsqu'il y a dépendance de donnée sur le registre rsX.
// Les rsX_re_dec_w sont à 1 lorsque l'instruction dans le stage DEC utilise rsX
logic rd_we_exec_w, rd_we_mem_w, rd_we_wb_w;
logic hazard_rs1, hazard_rs2;
logic rs1_re_dec_w, rs2_re_dec_w;

// Fonction qui retourne 1 si l'instruction utilise rd, 0 sinon.
function logic rd_used(input logic [31:0] inst);
    // Cas spécial pour les instructions qui utilisent R0.
    // On DOIT les rendre non-bloquants sinon ils peuvent
    // créer un stall infini.
    //
    //            RD    !=    R0      &&   OPCODE[5:3] != 3'b100
    rd_used = ( inst[11:7] != 7'b0000000 ) && (  inst[5:3]  != 3'b100 );
endfunction

// On applique la fonction rd_used aux signaux rd des différents
// stages de la pipeline.
assign rd_we_exec_w = rd_used(inst_exec_r);
assign rd_we_mem_w = rd_used(inst_mem_r);
assign rd_we_wb_w = rd_used(inst_wb_r);

// Les opcodes qui utilisent rs1 ont le bit 2^2 à 0, les autres à 1
// Les opcodes qui utilisent rs2 ont le bit 2^2 à 0 et le 2^5 à 1, ce qui n'est
// pas le cas des autres opcodes.
assign rs1_re_dec_w = !opcode_dec_w[2];
assign rs2_re_dec_w = !opcode_dec_w[2] && opcode_dec_w[5];

// Il y a une dépendance de donnée sur rsX lorsque:
// - rsX est utilisé dans le stage DEC
// - rd est utilisé dans le stage EXEC, MEM ou WB
// - rsX correspond à rd dans le stage EXEC, MEM ou WB
assign hazard_rs1 =
(rs1_re_dec_w)
&& (
        (rd_we_exec_w && (rs1_dec_w == rd_add_exec_w))
    ||  (rd_we_mem_w  && (rs1_dec_w == rd_add_mem_w))
    ||  (rd_we_wb_w   && (rs1_dec_w == rd_add_wb_w))
);
assign hazard_rs2 =
(rs2_re_dec_w)
&& (
        (rd_we_exec_w && (rs2_dec_w == rd_add_exec_w))
    ||  (rd_we_mem_w  && (rs2_dec_w == rd_add_mem_w))
    ||  (rd_we_wb_w   && (rs2_dec_w == rd_add_wb_w))
);

// On stall si on a une dépendance de données sur rs1 ou sur rs2.
assign stall_o = hazard_rs1 || hazard_rs2;
```

Une fois avoir enlevé les dépendances de données de notre programme, (maintenant enregistré sous le nom `exo2_Q10.S` (Figure 3.10), nous pouvons tester cette logique sur ModelSim en faisant `./build Q10` dans le TD2. Cela nous permet de vérifier que ce code est juste puisque les registres ont les bonnes valeurs de fin et que le signal de *stall* s'active au bon moment.

FIGURE 2.6 : Activation du stall pour les dépendances de données



Note : Retrouvez l'intégralité de la simulation en cliquant ici (Figure 3.11)

2.1.3 Correction matérielle des dépendances de contrôle de type saut

Pour assurer le bon fonctionnement du processeur lors de l'exécution d'instructions de rupture de flux, nous avons implémenté une logique spécifique pour les instructions de type `J`.

Analyse du comportement au travers d'un exemple

Considérons le programme assembleur suivant :

Programme d'exemple pour la gestion de saut

```
0x100: ADDI R1, R1, 15
0x104: ADDI R1, R1, 15
0x108: ADDI R2, R1, 15
0x10C: J 0x11C              # Instruction de saut
0x110: ADDI R4, R0, 10      # Instruction devant être annulée
0x114: ADDI R5, R0, 10
0x118: ADDI R6, R0, 10
0x11C: ADDI R7, R2, 5       # Cible du saut
0x120: NOP
0x124: ORI R7, R7, 16
```

Dans ce scénario, le flux d'exécution se déroule normalement jusqu'à ce que l'instruction `J` atteigne l'étage `DECODE`. Au front d'horloge suivant, l'instruction située à l'adresse `0x110` (`ADDI R4, R0, 10`), qui a déjà été chargée par anticipation, est remplacée par une instruction `NOP`. Simultanément, le circuit de calcul des sauts met à jour le (`PC`) à l'adresse cible `0x11C`.

L'état successif des instructions dans l'étage `DECODE` est alors le suivant :

1. `J 0x11C`
2. `NOP` (insérée pour annuler l'instruction chargée par erreur)
3. `ADDI R7, R2, 5`

Note sur les dépendances de données : Un point crucial de cette implémentation est la synergie avec le mécanisme de *stall*. On remarque que lorsque l'instruction `ADDI R7, R2, 5` arrive à l'étage `DECODE`, l'instruction `ADDI R2, R1, 15` se situe à l'étage `WRITE BACK`. En conséquence, le signal de *stall* s'active naturellement.

Il n'est donc pas nécessaire de modifier la condition de stall existante pour gérer les dépendances liées aux sauts.

À l'étage `EXECUTE`, la séquence d'exécution finale se présente ainsi :

- `ADDI R2, R1, 15`
- `J 0x11C`
- `NOP`
- `NOP` (induite par le *stall*)
- `ADDI R7, R2, 5`

Modifications matérielles

En résumé, l'implémentation de la logique des sauts repose sur deux piliers :

1. La détection des instructions de type `J` dès l'étage `DECODE`.
2. L'ajout d'un multiplexeur en amont du registre de l'étage `DECODE`. Ce multiplexeur transmet l'instruction provenant de l'étage `FETCH` par défaut, mais insère un `NOP` dès qu'un saut est détecté.

Modification du Datapath Dans le fichier `RV32i_pipeline_datapath.sv`, nous avons introduit un signal d'entrée permettant de forcer l'insertion d'un `NOP`.

RV32i_pipeline_datapath.sv

```
module RV32i_datapath (
    ...
    // Fetch stage
    input logic fetch_nop_i, // Signal du control_path
                             // sélectionnant un NOP
                             // pour le registre DEC à la place de
                             // l'instruction
                             // mémoire habituelle.
    ...
);
    ...
    // Sélectionne un NOP (0x00000013) si fetch_nop_i est actif,
    // sinon transmet imem_data_i
    assign inst_w = fetch_nop_i ? 32'h00000013 : imem_data_i;
```

Modification du Controlpath Dans le fichier `RV32i_pipeline_controlpath.sv`, la logique de détection identifie l'Opcode correspondant au type `J`.

RV32i_pipeline_controlpath.sv

```
module RV32i_controlpath (
    ...
    // Fetch stage
    output logic fetch_nop_o, // Commande l'insertion du NOP dans
                              // le datapath.
    ...
);
...
// Activation si l'opcode à l'étage de décodage correspond à une
  ↪  instruction J
// opcode_dec_w provient de l'instruction actuellement en étage DEC
assign fetch_nop_o = (opcode_dec_w == 7'b1101111);
```

Interconnexion au niveau Top Enfin, les signaux sont interconnectés dans le module *Top*.

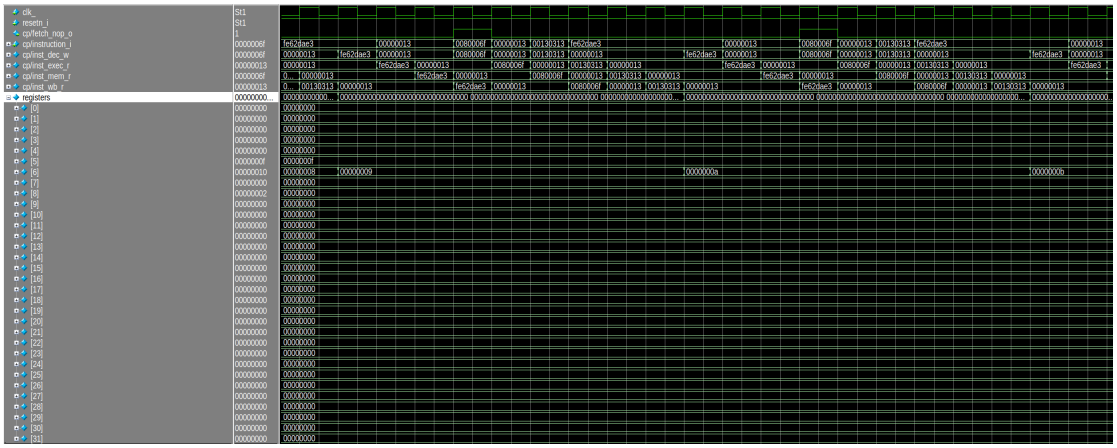
RV32i_pipeline_top.sv

```
// Signal d'interconnexion pour le Fetch Stage
logic fetch_nop_w;

RV32i_datapath dp (
    ...
    .fetch_nop_i(fetch_nop_w),
    ...
);

RV32i_controlpath cp (
    ...
    .fetch_nop_o(fetch_nop_w),
    ...
);
```

2.1.4 Test de la gestion des sauts

Pour tester notre gestion des sauts, nous créons une autre version du programme, appelée `exo2_Q12.S` (Figure 3.12) qui ne contient plus de `NOP` là où il y avait des gestions de saut. Une simulation ModelSim nous permet de voir que notre solution fonctionne puisque les registres ont les bonnes valeurs à la fin du programme et que le signal `fetch_nop_o` s'active au bon moment.
Pour lancer cette simulation, tapez `./build Q12`

FIGURE 2.7 : Activation du `fetch_nop_o` pour les dépendances de saut



Note : Retrouvez l'intégralité de la simulation en cliquant ici (Figure 3.13)

2.2 Correction matérielle des dépendances de contrôle de branchement

2.2.1 Logique de fonctionnement des branchements

Une opération de branchement suit la logique algorithmique suivante :

```
1 if (condition(rs1, rs2)) {
2     PC += imm;
3 } else {
4     PC += 4;
5 }
```

Dans notre architecture, l'adresse cible du PC en cas de branchement peut être déterminée dès l'étage `DECODE`. Cependant, la décision effective (savoir si la condition est remplie) ne peut être validée qu'après le passage dans l'ALU.

Ainsi, la décision n'est connue avec certitude qu'au moment où le résultat de la condition est stabilisé dans le registre de l'étage `MEMORY`.

En fonction du résultat, deux scénarios se présentent :

- **Si la branche n'est pas prise** : Le flot d'exécution se poursuit séquentiellement sans interruption.
- **Si la branche est prise** : Deux instructions ont déjà été chargées par anticipation dans le pipeline (aux étages `DECODE` et `EXECUTE`). Ces instructions étant invalides, elles doivent être supprimées (*flush*).

2.2.2 Analyse des dépendances de données lors d'un branchement

Lorsqu'une dépendance de données survient simultanément à un branchement effectif, la situation est la suivante :

- Le branchement se situe dans l'étage `MEMORY`.
- Une instruction dépendante se trouve dans `DECODE`.
- Une autre instruction dépendante est soit dans `EXECUTE`, soit dans `WRITE BACK`.

Si l'instruction dépendante est dans `WRITE BACK` ou `EXECUTE` et que la branche est prise, les étages `DECODE` et `EXECUTE` sont vidés. Par conséquent, la dépendance de donnée est naturellement résolu par le changement de flux de contrôle.

2.2.3 Problématique du (*stall*) actuel

Avec notre implémentation actuelle, une dépendance de donnée lors d'un branchement active le signal `stall_o = 1` dans le *controlpath*. Cela propage `stall_dec_w = 1` dans le *top*, puis `stall_dec_i = 1` dans le *datapath*.

Considérons le code de gestion du PC :

```
// Définition de la clock
always_ff @(posedge clk_i, negedge resetn_i) begin : program_counter
    if (!resetn_i) pc_counter_r <= '0;
    else begin
        if (!stall_dec_i || (pc_next_sel_i == SEL_PC_BRANCH)) begin
            pc_counter_r <= pc_next_w;
        end
    end
end

// Définition de la sélection du prochain PC
always_comb begin : pc_next_comb
    case (pc_next_sel_i)
        SEL_PC_PLUS_4: pc_next_w = pc_plus4_w;
        SEL_PC_JAL:    pc_next_w = pc_j_target_w;
        SEL_PC_JALR:   pc_next_w = pc_jr_target_w;
        SEL_PC_BRANCH: pc_next_w = pc_br_target_r;
        default:       pc_next_w = pc_plus4_w;
    endcase
end
```

Le problème réside dans le fait que le PC se retrouve bloqué par le mécanisme de *stall*. Cela empêche l'exécution effective du branchement et entraîne un gaspillage de cycles d'horloge.

Il est donc impératif de modifier la formule du signal `stall_o` !

2.2.4 Implémentation de la solution

Pour corriger ce comportement, nous devons :

- Implémenter un mécanisme permettant de remplacer le contenu des étages `EXECUTE` et `DECODE` par des instructions `NOP` (*flush*).
- Lever l'état de *stall* si un branchement est pris (étage `MEMORY`).

Modifications du Controlpath

Dans le fichier `RV32i_pipeline_controlpath.sv`, nous modifions l'équation du *stall* pour qu'elle soit ignorée si une branche est prise :

RV32i_pipeline_controlpath.sv

```
// La condition de stall est qualifiée par l'absence de
// branchement pris
assign stall_o = (hazard_rs1 || hazard_rs2) && !branch_taken_w;

// Étage DEC
assign inst_dec_w = stall_dec_i ? 32'h00000013 : instruction_i;

// Étage EXEC
always_ff @(posedge clk_i or negedge resetn_i) begin : exec_stage
    if (!resetn_i) begin
        inst_exec_r <= 32'h0;
    end else if (branch_taken_w) begin
        // Flush de l'étage EXEC en insérant un NOP
        inst_exec_r <= 32'h00000013;
    end else if (!stall_exec_i) begin
        inst_exec_r <= inst_dec_w;
    end
end

// Logique de décision du branchement
always_comb begin : branch_taken_comb
    case (opcode_exec_w)
        RV32I_B_INSTR: begin
            case (func3_exec_w)
                RV32I_FUNCT3_BEQ:  branch_taken_w = alu_zero_i;
                RV32I_FUNCT3_BNE:  branch_taken_w = !alu_zero_i;
                RV32I_FUNCT3_BLT:  branch_taken_w = alu_lt_i;
                RV32I_FUNCT3_BGE:  branch_taken_w = !alu_lt_i;
                RV32I_FUNCT3_BLTU: branch_taken_w = alu_ltu_i;
                RV32I_FUNCT3_BGEU: branch_taken_w = !alu_ltu_i;
                default:           branch_taken_w = 1'b0;
            endcase
        end
        default: branch_taken_w = 1'b0;
    endcase
end

assign branch_taken_o = branch_taken_w;
```

Modifications du Datapath

Dans le fichier `RV32i_pipeline_datapath.sv`, nous ajoutons les entrées de *flush* pour nettoyer les registres de pipeline :

RV32i_pipeline_datapath.sv

```
// Étage DEC
always_ff @(posedge clk_i or negedge resetn_i) begin : dec_stage
    if (!resetn_i) begin
        // Reset des signaux...
        opcode_r <= '0;
    end else if (flush_dec_i) begin
        // Flush : insertion d'un NOP (ADDI x0, x0, 0)
        opcode_r <= 7'b0010011;
        func3_dec_r <= '0;
        func7_dec_r <= '0;
        rs1_add_r <= '0;
        rs2_add_r <= '0;
        rd_add_dec_r <= '0;
        pc_counter_dec_r <= '0;
    end else if (!stall_dec_i) begin
        opcode_r <= opcode_w;
        func3_dec_r <= func3_w;
        func7_dec_r <= func7_w;
        rs1_add_r <= rs1_add_w;
        rs2_add_r <= rs2_add_w;
        rd_add_dec_r <= inst_w[11:7];
        pc_counter_dec_r <= pc_counter_r;
    end
end

// Étage EXEC
always_ff @(posedge clk_i or negedge resetn_i) begin : exec_stage
    if (!resetn_i) begin
        alu_op1_data_r <= '0;
        // ...
    end else if (flush_exec_i) begin
        alu_op1_data_r <= '0;
        alu_op2_data_r <= '0;
        rs2_data_r <= rs2_data_w;
        func3_exec_r <= '0;
        pc_br_target_r <= pc_br_target_w;
        pc_counter_exec_r <= pc_counter_dec_r;
    end else if (!stall_exec_i) begin
        alu_op1_data_r <= alu_op1_data_w;
        alu_op2_data_r <= alu_op2_data_w;
        rs2_data_r <= rs2_data_w;
        func3_exec_r <= func3_dec_r;
        pc_br_target_r <= pc_br_target_w;
        pc_counter_exec_r <= pc_counter_dec_r;
    end
end
```

Modifications du Module Top

Enfin, nous connectons les signaux de *flush* au signal `branch_taken` dans `RV32i_pipeline_top.sv` :

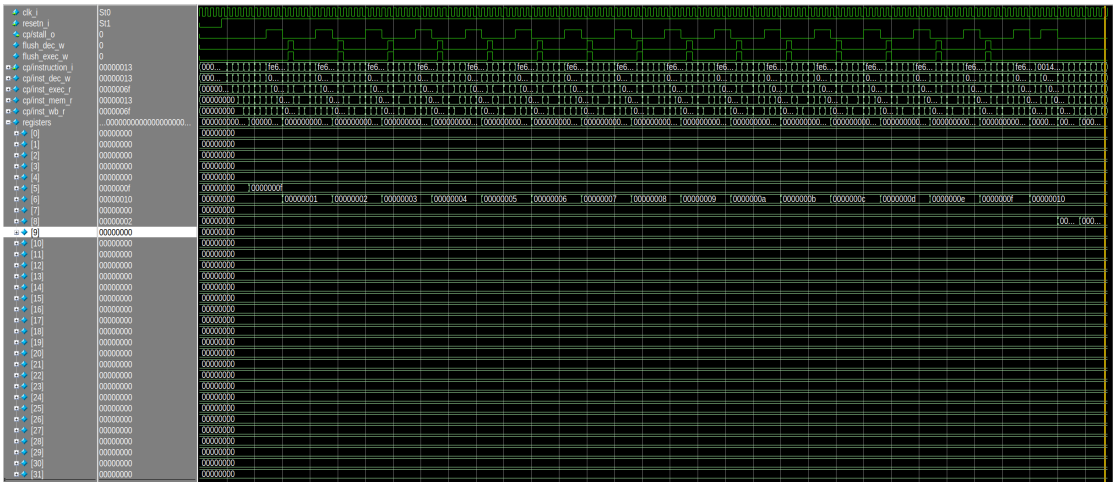
RV32i_pipeline_top.sv

```
logic branch_taken_w;
logic flush_dec_w, flush_exec_w;

RV32i_datapath dp (
    .flush_dec_i(flush_dec_w),
    .flush_exec_i(flush_exec_w),
    // ...
);

RV32i_controlpath cp (
    .branch_taken_o(branch_taken_w),
    // ...
);

// Utilisation de signaux distincts pour le futur,
// bien qu'identiques ici
assign flush_dec_w = branch_taken_w;
assign flush_exec_w = branch_taken_w;
```

2.2.5 Test de la gestion des dépendances de contrôle de type branchement

Pour tester notre gestion des branchements, nous créons une autre version du programme, appelée `exo2_Q14.S` (Figure 3.14) qui ne contient plus de `NOP` nulle part. Une simulation ModelSim nous permet de voir que notre solution fonctionne puisque les registres ont les bonnes valeurs à la fin du programme. Pour lancer cette simulation, tapez `./build Q14`

FIGURE 2.8 : Test ModelSim des dépendances de branchement



Note : Retrouvez l'intégralité de la simulation en cliquant ici (Figure 3.15)

2.3 Validation globale avec `ilock_tests.S`

Pour finaliser la validation de notre architecture, nous avons soumis le processeur à une suite de tests intensive contenue dans le fichier `ilock_tests.S`.

Le programme `ilock_tests.S` est structuré en plusieurs séquences distinctes, chacune isolée par l'insertion de quatre instructions `NOP`. Cette séparation permet de s'assurer que le pipeline est totalement vidé entre chaque test, évitant ainsi toute interférence résiduelle entre les différentes conditions de risque étudiées.

Le tableau ci-dessous récapitule les valeurs attendues dans les registres `x1` à `x5` après l'exécution des différentes étapes de validation :

Pour lancer la simulation ModelSim correspondante, utiliser la commande `./build.sh Q15`

FIGURE 2.9 : Valeurs des registres lors des tests d'interlock (`ilock_tests.S`)

Nature du Test	Étape	x1	x2	x3	x4	x5
1. Sans dépendance	-	1	2	3	4	5
2. Dépendance `rs1`	1	2	3	-	-	-
	2	3	3	-	-	-
	3	3	3	-	-	-
	4	3	3	-	-	-
	5	3	3	-	-	-
3. Dépendance `rs2`	1	4	5	-	-	-
	2	5	5	-	-	-
	3	5	5	-	-	-
	4	5	5	-	-	-
	5	5	5	-	-	-
4. Non-activation Interlock	1	`0x10800`	6	-	-	-
	2	`0x10800`	6	`0x0C`	-	-
	3	`0x10800`	6	`0x0C`	-	-
	4	`0x10800`	6	`0x0C`	-	-
	5	`0x10800`	6	`0x0C`	-	-
	6	`0x10800`	6	`0x07`	-	-
5. Vérification JAL	-	@ retour	-	-	-	-
6. Dépendance de contrôle	-	-	`0x0E`	-	-	-

2.3.1 Conclusion des tests

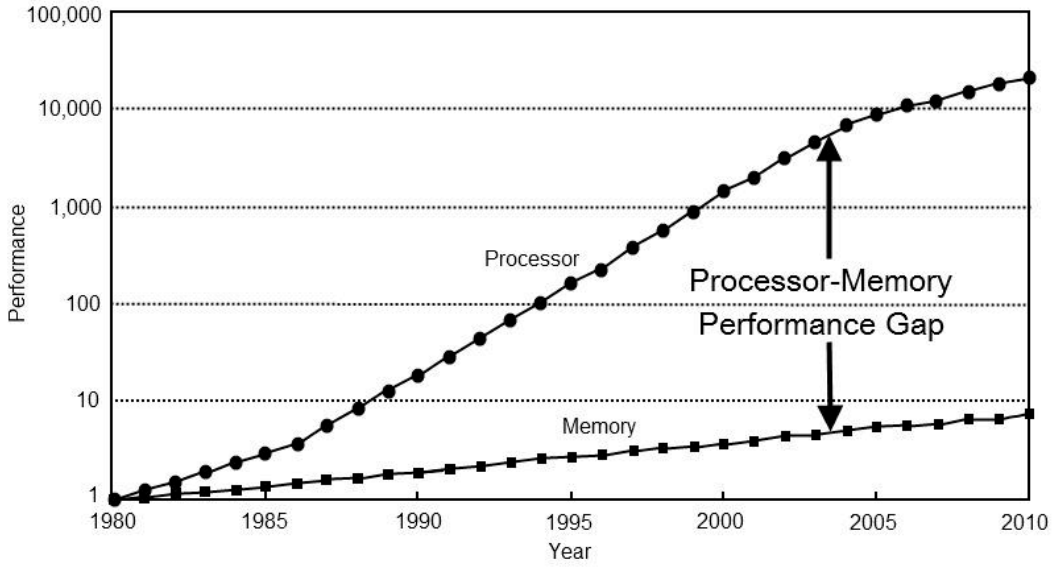
L'analyse des chronogrammes de simulation confirme que les tests sont tous passés. À l'issue de cette campagne de mesures, les résultats obtenus en simulation correspondent strictement aux valeurs théoriques présentées ci-dessus. Nous pouvons donc conclure que le processeur a passé avec succès l'intégralité des tests fonctionnels et gère désormais correctement les dépendances de don-

nées et de contrôle.

2.4 Implémentation du *Wait State*

Depuis le début des TD, les mémoires `IMEM` et `DMEM` n'avaient aucune latence. Ce comportement n'est pas très réaliste puisqu'on observe qu'un écart se creuse de plus en plus entre la vitesse des processeurs et la vitesse de la mémoire :

FIGURE 2.10 : Comparaison de la vitesse du CPU et de la Mémoire



Ainis, nous allons artificiellement ajouter 3 cycles de latence aux mémoires `DMEM` et `IMEM` dans le but de reproduire un comportement plus fidèle du processeur.

RV32i_soc.sv

```
wsync_mem #(
    .SIZE(4096),
    .WS(3), // 3 cycles de latence
    .INIT_FILE(IMEM_INIT_FILE)
) imem (
  ...
);

wsync_mem #(
    .SIZE(4096),
    .WS(3), // 3 cycles de latence
    .INIT_FILE(DMEM_INIT_FILE)
) dmem (
  ...
);
```

2.4.1 Adaptation du processeur à ce changement

2.4.2 Gestion du signal de validité

Durant l'intervalle s'écoulant entre la requête d'accès et la réponse effective de la mémoire, le bus de sortie `d_o` contient des données non significatives. Il est donc impératif de conditionner la lecture du contenu de `d_o` à l'activation du signal `valid_o`.

Problématique de la désynchronisation lors des branchements

Un cas critique survient lors de l'exécution d'une instruction de branchement. Lorsque cette instruction atteint l'étage `EXEC`, une nouvelle requête d'instruction est émise alors que la précédente n'a potentiellement pas encore validé son cycle par le signal `valid_o`.

Le module de mémoire fourni, `wsync_mem.sv`, ne réinitialise son compteur interne permettant de générer `valid_o` que lorsque les signaux de lecture (`re_i`) et d'écriture (`we_i`) sont nuls. Par conséquent, si le signal `imem_re_o` (correspondant au `re_i` de la mémoire) n'est pas forcé à 0 au moment où le branchement est validé dans l'étage `EXEC`, le signal `valid_o` sera désynchronisé pour l'instruction suivante.

Il est à noter que, bien que la mémoire `wsync_mem.sv` maintienne une valeur correcte sur le bus `d_o` même en l'absence de `valid_o`, maintenir `imem_re_o` à 1 en permanence n'affecte pas l'exécution fonctionnelle dans cet environnement de simulation spécifique. Cependant, par souci de rigueur architecturale et pour garantir la portabilité vers une mémoire réelle, une correction logicielle est nécessaire.

En résumé

Le signal `imem_re_o` doit être forcé à 0 lorsqu'une instruction de type B à l'étage `EXEC` valide un branchement.

RV32i_pipeline_top.sv

```
// Pour synchroniser le signal valid_o de la mémoire d'instruction
// lorsqu'on prends une branche. Dans le cas contraire on a une
// désynchronisation du temps d'attente pour l'instruction
// suivant le branchement (ce qui peut potentiellement mener
// à une instruction invalide dans d_o)
assign imem_re_o = !branch_taken_w;
```

TD 3

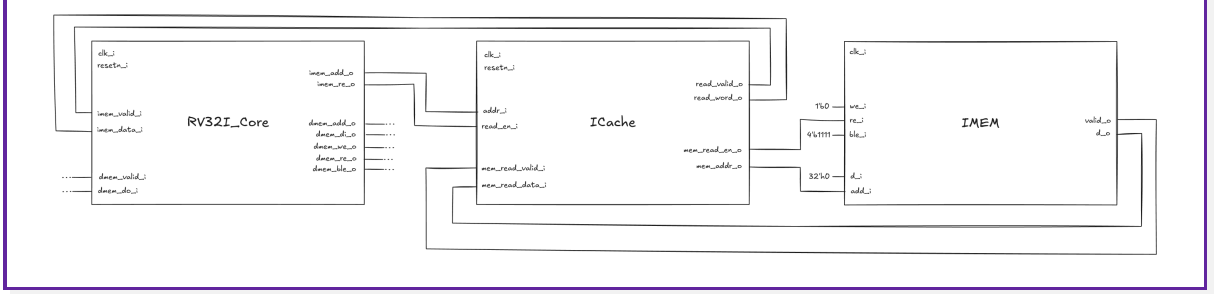
Implémentation d'une mémoire cache

L'introduction d'une latence de trois cycles d'horloge lors de l'accès à la mémoire d'instruction a mis en évidence un goulot d'étranglement majeur pour notre architecture. Dans cette configuration, chaque chargement d'instruction pénalise lourdement le débit du processeur, limitant drastiquement le nombre d'instructions exécutées par cycle.

L'objectif de cette section est de pallier cette latence en concevant et en intégrant un cache d'instructions en lecture seule. En exploitant le principe de localité spatiale et temporelle, ce module permettra de stocker localement les instructions les plus fréquemment sollicitées, réduisant ainsi le temps d'accès moyen à la mémoire.

Le cache sera situé comme ceci dans la hiérarchie des composants :

FIGURE 3.1 : Insertion du cache en tant que composant



3.1 Calculs préliminaires

Dans un premier temps, nous allons implémenter un cache mémoire direct. Afin de concevoir un cache adaptée à notre architecture pipelinée, nous avons défini un adressage mémoire segmenté. Cette étude préalable permet de déterminer les caractéristiques physiques du cache L1 ainsi que le gain de performance attendu.

3.1.1 Analyse de la structure du cache direct

L'adresse de 32 bits fournie par le processeur est décomposée selon le schéma suivant pour un cache direct :

Bits [31 :10]	Bits [9 :4]	Bits [3 :0]
Étiquette (TAG)	Index	Décalage (OFFSET)

Taille d'une ligne de cache

La taille d'une ligne de cache est déterminée par le champ `OFFSET`. Avec 4 bits dédiés, nous pouvons calculer la capacité par ligne en octets et en mots :

- **En octets :** $L_{octet} = 2^{\text{OFFSET}} = 2^4 = 16$ octets.
- **En mots :** Puisque notre architecture est basée sur des mots de 32 bits (4 octets), une ligne contient :

$$\frac{16 \text{ octets}}{4 \text{ octets/mot}} = 4 \text{ mots}$$

Nombre d'entrées (lignes)

Dans notre architecture, le nombre d'entrées correspond au nombre de lignes disponibles, défini par le champ `INDEX`. Avec 6 bits :

$$N_{lignes} = 2^{\text{INDEX}} = 2^6 = 64 \text{ entrées}$$

3.1.2 Analyse des performances attendues

Pour évaluer l'efficacité de ce cache, nous considérons un processeur cadencé à 100 MHz, soit une période d'horloge $T_{clk} = 10$ ns. Les paramètres de simulation sont les suivants :

- **L1 HIT Time :** 1 cycle (10 ns)

- **MISS Penalty :** 10 cycles ($100\,\text{ns}$)
- **L1 HIT Rate :** 90 % ($0,9$)

Miss Rate

Le taux de défaut est le complémentaire du taux de succès :

$$\text{Miss Rate} = 1 - \text{Hit Rate} = 1 - 0,9 = 0,1 \text{ (soit } 10\,\% \text{)}$$

Temps d'accès moyen

Pour un échantillon de 1000 accès mémoire, le temps d'accès moyen (T_{moy}) se calcule par la moyenne pondérée du temps de succès et du temps de pénalité :

$$\begin{aligned} T_{moy} &= (\text{Hit Rate} \times \text{Hit Time}) + (\text{Miss Rate} \times \text{Miss Time}) \\ T_{moy} &= (0,9 \times 10\,\text{ns}) + (0,1 \times 100\,\text{ns}) \\ T_{moy} &= 9\,\text{ns} + 10\,\text{ns} \\ T_{moy} &= 19\,\text{ns} \end{aligned} \tag{3.1}$$

Conclusion : L'intégration de ce cache permet d'atteindre un temps d'accès moyen de $19\,\text{ns}$ par instruction, contre $100\,\text{ns}$ en cas d'accès systématique à la mémoire principale. Cela valide l'intérêt du cache pour compenser la latence imposée.

3.2 Design et implémentation du cache direct

3.2.1 Intégration au SoC

L'architecture du `RV32i_soc` intègre désormais un module `cache_direct` agissant comme interface entre le `RV32i_core` et la mémoire d'instruction (`IMEM`).

L'interconnexion entre l'`IMEM` et le cache utilise un bus de données large de 128 bits (4 mots de 32 bits). Ce choix permet de charger une ligne de cache entière en une seule transaction mémoire, optimisant ainsi la localité spatiale.

L'introduction du signal `imem_valid_i` est cruciale : la latence mémoire n'étant plus constante (différence entre un *HIT* instantané et un *MISS* avec pénalité), ce signal permet au contrôleur de cache d'indiquer au coeur que l'instruction est prête à être consommée.

3.2.2 Caractéristiques Techniques

Le cache est directe. Ainsi, les paramètres sont définis comme suit :

Paramètres du cache

```
// Paramètres de segmentation de l'adresse [cite: 1, 2]
localparam ByteOffsetBits = 4;  // Bits 3 à 0 (16 octets par ligne)
localparam IndexBits      = 6;  // Bits 9 à 4 (64 entrées)
localparam TagBits        = 22; // Bits 31 à 10

// Calculs de structure
localparam NrWordsPerLine = (2**ByteOffsetBits) / 4; // 4 mots/ligne
localparam NrLines        = 2**IndexBits;           // 64 lignes
localparam LineSize       = 32 * NrWordsPerLine;    // 128 bits de données
```

3.2.3 Étapes de Conception

Structure de Stockage

Chaque ligne du cache doit stocker trois types d'informations pour identifier et restituer les données :

- **Le Dirty-Bit** : Indique si la ligne contient des données synchronisées avec la mémoire.
- **Le Tag** : Les 22 bits de poids fort de l'adresse originale.
- **Les données** : La ligne de 128 bits (4 mots).

Le stockage est modélisé par un tableau de registres de taille $[151 : 0]$ (128 bits de données + 22 bits de Tag + 1 bit de validité) :

```
logic [LineSize + TagBits:0] registers [NrLines] = '{default: 0};
```

Découpage de l'Adresse et Détection du Hit

L'adresse reçue `addr_i` est décomposée de manière combinatoire. La condition de *HIT* est validée si le bit de validité est à 1 et que le Tag stocké correspond au Tag de l'adresse demandée :

Système de détection de HITS

```
assign {tag_w, index_w, offset_w} = addr_i;
assign {rdirty_bit_w, rtag_w, rdata_w} = registers[index_w];

assign request_hit_w = read_en_i && (rdirty_bit_w == 1'b1) &&
  ↪  (rtag_w == tag_w);
```

Sélection du Mot

Comme le cache stocke 128 bits par ligne, nous utilisons l'offset pour extraire le mot de 32 bits spécifique demandé par le processeur. Pour optimiser les res-

sources matérielles, nous utilisons un décalage binaire :

```
assign shifted_rdata = rdata_w >> ('h8 * offset_w);
assign read_word_o = shifted_rdata[31:0];
```

3.2.4 Machine d'État et Optimisations

Conception de la FSM

Pour garantir la performance, la logique de contrôle utilise une machine d'état à trois états :

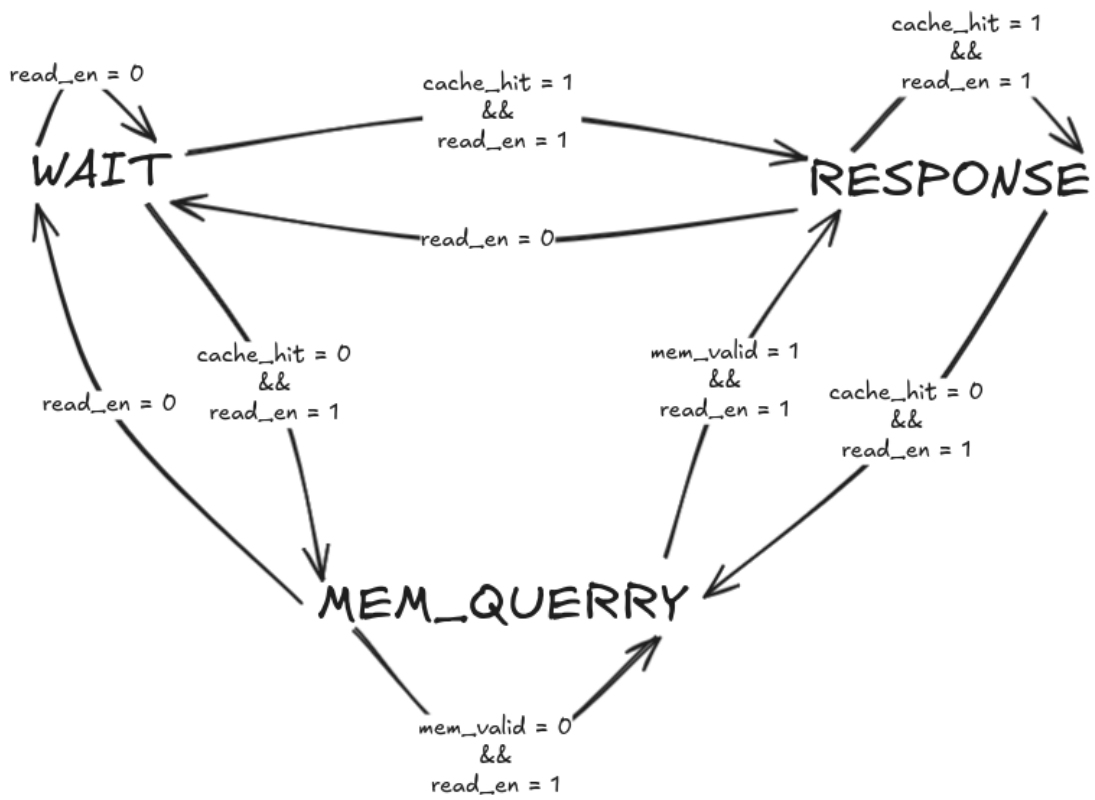
- **WAIT** : État de repos. Si un *HIT* est détecté, transition vers `RESPONSE`, sinon `MEM_QUERRY`.
- **MEM_QUERRY** : Envoi de la requête à la mémoire principale. On attend le signal `mem_read_valid_i`. À réception, la ligne de cache est mise à jour.
- **RESPONSE** : Délivrance de la donnée au processeur via `read_valid_o`.

Optimisations de Performance

Afin d'éviter de gaspiller des cycles d'horloge, plusieurs optimisations ont été implémentées :

- **Transition Directe RESPONSE → MEM_QUERRY** : Si, durant l'état de réponse, le processeur demande une nouvelle adresse qui n'est pas dans le cache, la FSM bascule directement en requête mémoire sans repasser par `WAIT`.
- **Logique Combinatoire du Prochain État** : Le calcul de `next_cache_state` est placé dans un bloc `always_comb`. Cela permet d'anticiper les signaux de contrôle mémoire (comme `mem_read_en_o`) dès le cycle actuel et nous fait gagner un cycle.
- **Gestion du Flush/Restart** : Si l'adresse `addr_i` change durant un `MEM_QUERRY`, le système redémarre la requête pour s'assurer que le pipeline ne reçoive pas une instruction obsolète.

FIGURE 3.2 : Machine d'état du cache direct



3.3 Test du cache direct

Voici une diagramme de simulation ModelSim. Le cache fait tourner le programme fourni avec le TD 3, appelé `exo3_2ways.S`, dont une copie se trouve en annexe.
Dans le but de tester le processeur, il existe aussi une version de ce programme sans les `NOP` appelé `exo3_2ways_sans_nop.S`, dont une copie est aussi disponible en annexe.

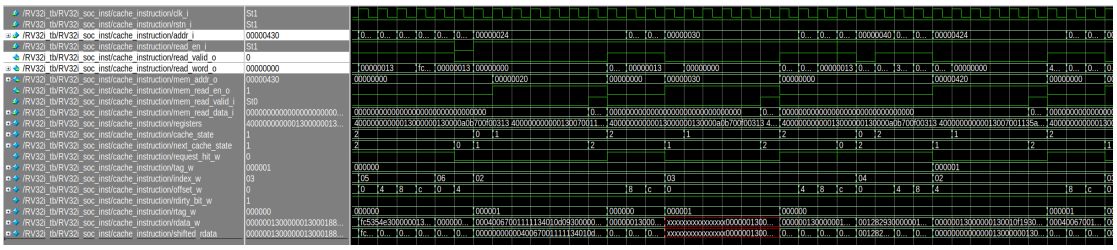
Le programme `exo3_2ways.S` est équivalent au programme ci-dessous en C. C'est un programme qui multiplie `x1` et `x2` et place le résultat dans `x7` :

Programme équivalent à `exo3_2ways.S`

```c
int main() {
      unsigned int x1, x2, x3, x4, x5, x6, x7;

      x5 = 0;
      x4 = 0;

      x6 = 15;
      x1 = 0xA5A5
      x2 = 7;

      do {
          x3 = x1 & 1;

          if (x3 != 0) {
              x4 += x2;
          }

          x1 = x1 >> 1;
          x2 = x2 << 1;

          x5++;

      } while (x6 >= x5);

      x7 = x4;

      return 0;
}
```

Pour lancer ces deux simulations, vous pouvez exécuter les commandes `./build direct` et `./build direct-sans-nop`
Voici le résultat de la simulation avec les `NOP` :

FIGURE 3.3 : Simulation du cache direct



On remarque que les changements d'état se font instantanément. Le cache ne peut physiquement pas aller plus vite puisqu'il ne gâche aucun cycle d'horloge.

3.4 Cache Instruction Associatif à Deux Voies

L'évolution de notre architecture vers un cache associatif à deux voies vise à réduire le taux de conflits d'indexation rencontrés dans le cache direct. Cette structure permet de stocker deux lignes de données différentes pour un même

index mémoire.

3.4.1 Architecture Matérielle

Le cache est désormais composé de deux structures de stockage identiques (voies), chacune capable de stocker une ligne de 128 bits, son Tag de 22 bits et un (*dirty-bit*).

```
// Les deux voies contenant les lignes de registre.
logic [LineSize + TagBits:0] registers_voie_1_r [NrLines] =
  ↪  '{default: 0};
logic [LineSize + TagBits:0] registers_voie_2_r [NrLines] =
  ↪  '{default: 0};
```

3.4.2 Stratégie de Remplacement : Algorithme LRU

Pour décider quelle voie remplacer lors d'un défaut de cache (*Miss*), nous implémentons la stratégie **LRU** (*Least Recently Used*).L'objectif est de préserver les données les plus récemment sollicitées en vertu du principe de localité.

Dans une configuration à deux voies, la gestion du LRU est simplifiée par l'utilisation d'un seul bit par index (`lru_data_r`) indiquant quelle voie est la candidate au remplacement :

- `lru_data_r = 0` : La Voie 1 sera remplacée.
- `lru_data_r = 1` : La Voie 2 sera remplacée.

Mise à jour de la priorité

La mise à jour de ce bit s'effectue de manière séquentielle à chaque succès de lecture (*Hit*). La logique suit la table de vérité suivante pour garantir que la voie non sollicitée devienne la prochaine victime :

FIGURE 3.4 : Table de vérité de la mise à jour du LRU

Hit Voie 1	Hit Voie 2	Prochain LRU	Signification
0	0	-	Pas de remplacement
0	1	0	Remplacer Voie 1 au prochain Miss
1	0	1	Remplacer Voie 2 au prochain Miss
1	1	IMPOSSIBLE	Imp. à cause de l'unicité du tag

Voici le code correspondant à cette mise à jour :

```
always_ff @(posedge clk_i or negedge rstn_i) begin
    // Ici on update la voie la plus récemment utilisée.
    // On ne peut pas avoir de hit sur les deux signaux en même
    // temps, s'il  y a un hit, c'est obligatoirement sur un
    // des deux signaux car le tag est unique pour un même index.
    // La table de vérité est équivalente à la valeur de
    // request_hit_1_w.
    if (request_hit_w) begin
        lru_data_r[index_w] <= request_hit_1_w;
    end
end
```

3.4.3 Étapes Logiques de Conception

Détection du Hit et Sélection des Données

Lors d'une requête, le cache compare simultanément le Tag de l'adresse avec ceux stockés dans les deux voies. Un multiplexeur sélectionne ensuite la donnée de la voie gagnante pour le traitement

```
// Les signaux qui indiquent si oui ou non la requète est un hit
// ou un miss.
// request_hit_1_w est pour la première voie
// request_hit_2_w est pour la deuxième voie
logic request_hit_1_w;
logic request_hit_2_w;
logic request_hit_w;

// La ligne de cache sélectionnée.
logic [LineSize-1:0] rdata_sel_w;

always_comb begin
    ...
    // Détection des Hits par voie
    request_hit_1_w = read_en_i && (rdirty_bit_1_w == 1'b1) &&
    ↪  (rtag_1_w == tag_w);
    request_hit_2_w = read_en_i && (rdirty_bit_2_w == 1'b1) &&
    ↪  (rtag_2_w == tag_w);
    request_hit_w   = request_hit_1_w || request_hit_2_w;

    // Sélection de la voie active
    rdata_sel_w = request_hit_2_w ? rdata_2_w : rdata_1_w;
    ...
end
```

Extraction du Mot et Récupération Mémoire

Comme pour le cache direct, le mot de 32 bits est extrait de la ligne de 128 bits via un décalage basé sur l'offset. En cas de *Miss*, la machine d'état (*FSM*) passe en état `MEM_QUERRY` pour solliciter `IMEM` et mettre à jour la voie désignée par le bit LRU.

3.4.4 Analyse des Performances

Les tests de simulation montrent un gain de performance significatif grâce à la réduction des défauts de cache par rapport au cache direct.
Le tableau suivant présente le temps total d'exécution du programme de test selon l'architecture utilisée :

FIGURE 3.5 : Comparaison des performances avec et sans cache

Architecture	Temps d'exécution (ns)
Sans cache	12 530
Cache direct	5 670
Cache associatif 2 voies	3 870

Ces tests ont été réalisé avec la version `sans-nop` du programme `exo3_-2ways.S`. Les commandes précises qui ont été utilisés sont :

- `./TD3/modified/build.sh direct sans-nop`
- `./TD3/modified/build.sh associative sans-nop`
- `./TD3_SANS_CACHE/modified/build.sh`

3.4.5 Conclusion

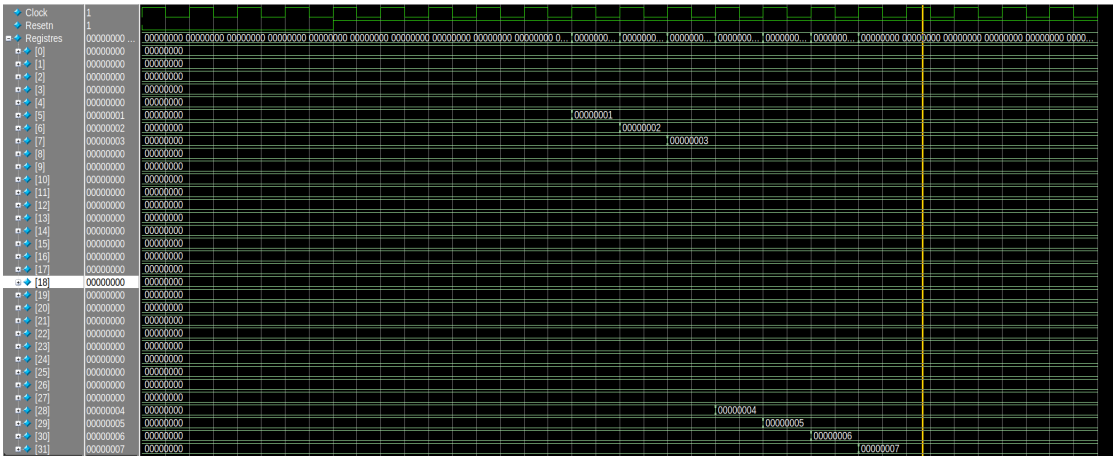
L'implémentation du cache associatif à deux voies, combinée à une gestion LRU rigoureuse, permet une réduction du temps d'exécution de près de 70 % par rapport à une architecture sans cache, et de 31 % par rapport au cache direct simple. La robustesse de la FSM, notamment la transition optimisée entre les états `RESPONSE` et `MEM_QUERRY`, assure une réactivité maximale du pipeline.

Annexe

Cette partie regroupe les ressources complémentaires nécessaires à la compréhension détaillée du rapport, mais dont le volume ne permettait pas une insertion directe dans le corps du rapport. Vous y trouverez notamment des extraits de code source complets ainsi que des captures de simulations et de chronogrammes directement prise de ModelSim. Ces documents servent de support technique aux analyses présentées précédemment.

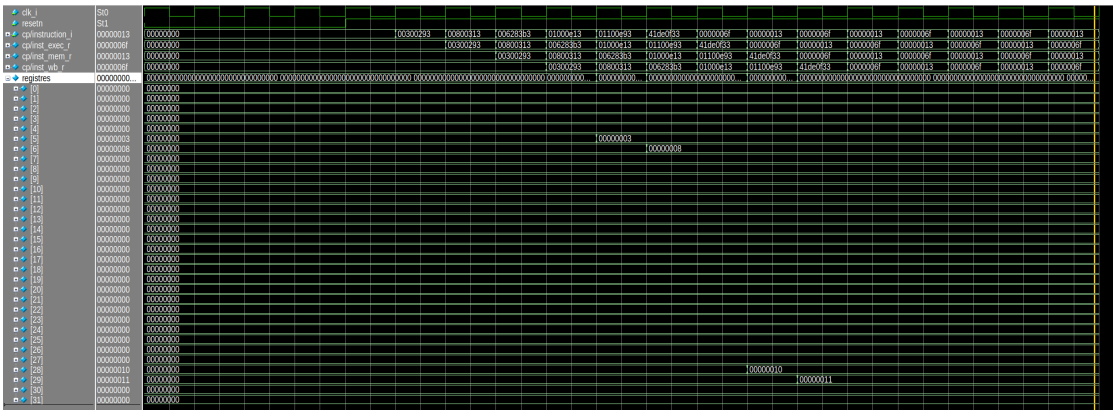
TD 1

FIGURE 3.6 : Chronogramme de `exo1.S` sur ModelSim



Note : Retour au Rapport (Figure 1.6)

FIGURE 3.7 : Chronogramme de `main.S` sur ModelSim



Note : Retour au Rapport (Figure 1.8)

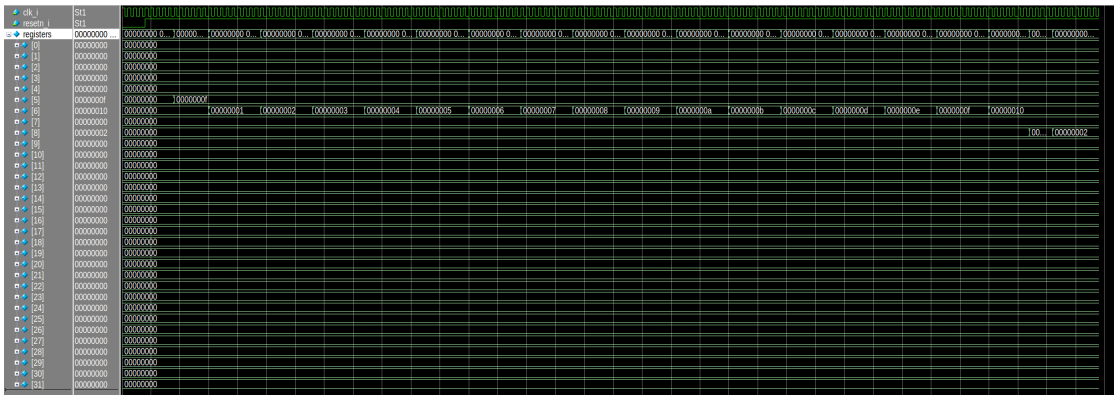
TD 2

FIGURE 3.8 : Programme `exo2_unmodified.S`

```
1  .section .start;
2  .globl start;
3
4  start:
5     addi x5,x0,15
6     add  x6,x0,x0
7     add  x7,x0,x0
8     add  x8,x0,x0
9  lab1:
10    j    lab2      # Dépendance de contrôle de type saut
11    addi x7,x7, 1
12 lab2:
13    addi x6,x6, 1   # Dépendance de données
14    bge x5,x6,lab1  # Dépendance de contrôle de type saut
15    addi x8,x8, 1   # Dépendance de données
16    addi x8,x8, 1
17 lab3:
18    j    lab3
19    nop
20    nop
21
22 .end start
23
```

Note : Retour au Rapport (Figure 2.1.1)

FIGURE 3.9 : Chronogramme de la correction logicielle sur ModelSim



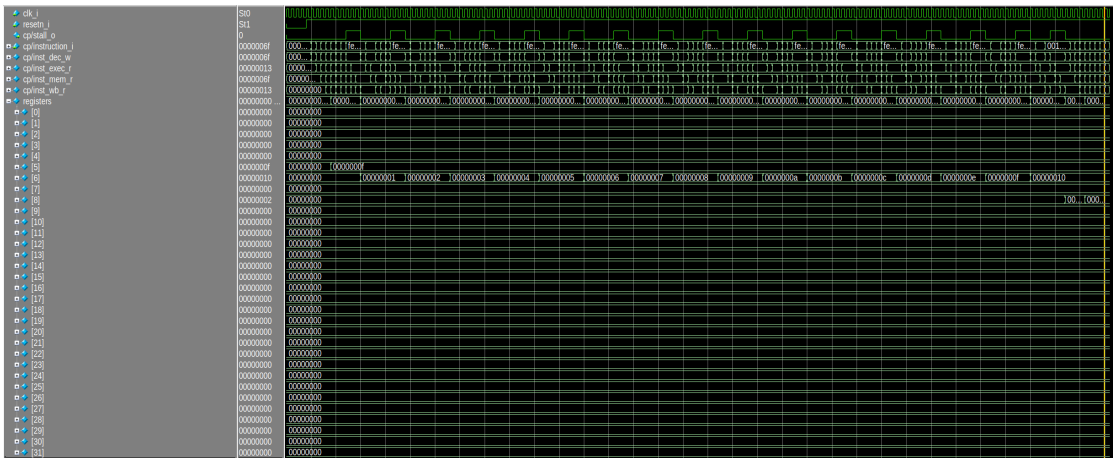
Note : Retour au Rapport (Figure 2.2)

FIGURE 3.10 : Programme exo2_Q10.S

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

start:
   addi x5,x0,15
   add  x6,x0,x0
   add  x7,x0,x0
   add  x8,x0,x0
lab1:
   j lab2        // Dépendance de contrôle de type J;
   nop          // On a besoin de 1 NOP
   addi x7,x7,1   //
lab2:
   addi x6,x6,1   // Dépendance de données
   bge x5,x6,lab1  //
   nop          // Dépendance de contrôle avec instruction type B:
   nop          // On a besoin de 2 NOP.
   addi x8,x8,1   // Dépendance de données
   addi x8,x8,1   //
lab3 :
   j lab3        // Condition de stop du testbench:
   nop          // avoir 5 fois j lab3 suivi d'un nop

.end start

```

Note : Retour au Rapport (Figure 2.1.2)

FIGURE 3.11 : Chronogramme gestion des dépendances de données par le *stall*



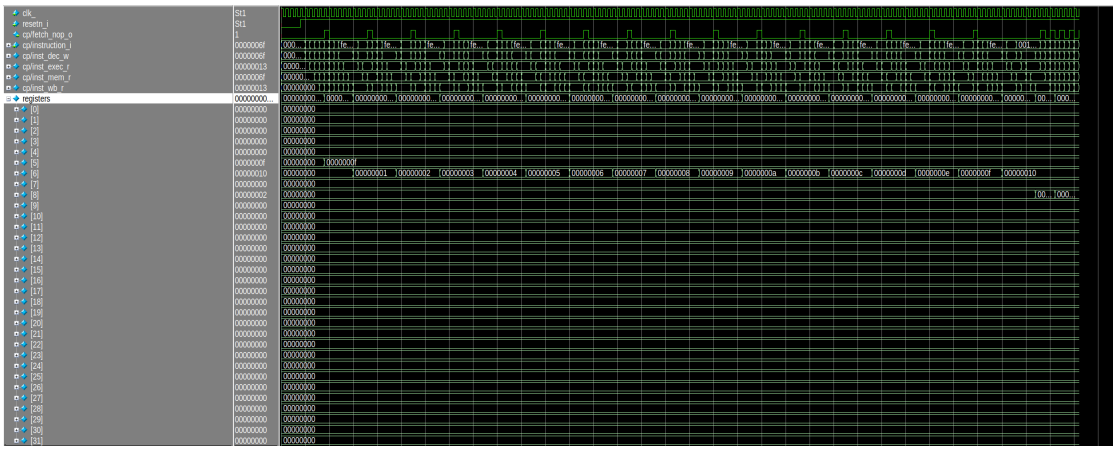
Note : Retour au Rapport (Figure 2.6)

FIGURE 3.12 : Programme exo2_Q12.S

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

start:
   addi x5,x0,15
   add  x6,x0,x0
   add  x7,x0,x0
   add  x8,x0,x0
lab1:
   j lab2
   addi x7,x7,1
lab2:
   addi x6,x6,1   // Dépendance de données
   bge x5,x6,lab1  //
   nop          // Dépendance de contrôle avec instruction type B:
   nop          // On a besoin de 2 NOP.
   addi x8,x8,1   // Dépendance de données
   addi x8,x8,1   //
lab3:
   j lab3        // Condition de stop du testbench:
   nop          // avoir 5 fois j lab3 suivi d'un nop
.end start
```

Note : Retour au Rapport (Figure 2.1.4)

FIGURE 3.13 : Chronogramme gestion des dépendances de saut par *fetch_nop_o*



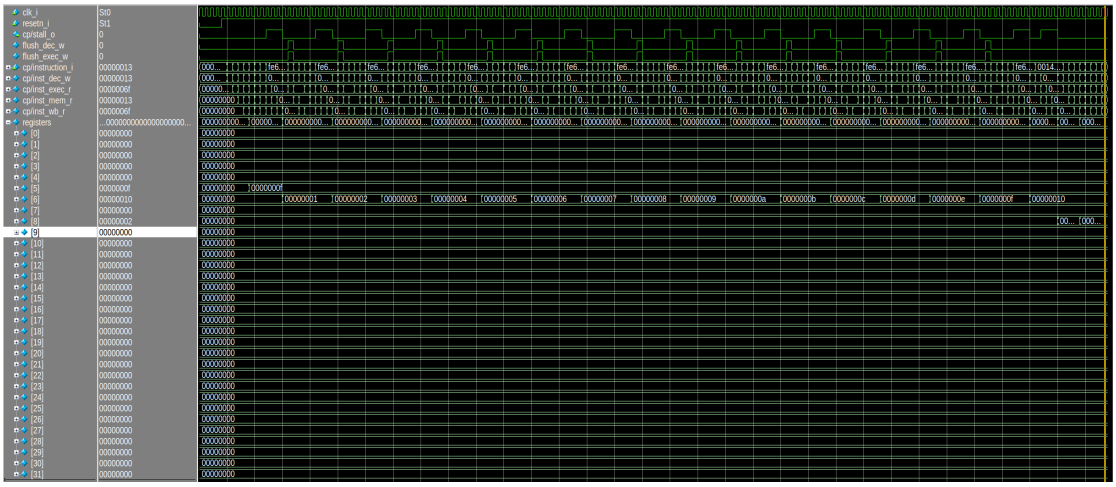
Note : Retour au Rapport (Figure 2.7)

FIGURE 3.14 : Programme exo2_Q14.S

```
1  .section .start;
2  .globl start;
3
4  start:
5     addi x5,x0,15
6     add  x6,x0,x0
7     add  x7,x0,x0
8     add  x8,x0,x0
9  lab1:
10    j lab2
11    addi x7,x7,1
12 lab2:
13    addi x6,x6,1   // Dépendance de données
14    bge x5,x6,lab1  // Dépendance de contrôle
15    addi x8,x8,1
16    addi x8,x8,1   // Dépendance de données
17 lab3:
18    j lab3        // Condition de stop du testbench:
19    nop          // avoir 5 fois j lab3 suivi d'un nop
20 .end start
```

Note : Retour au Rapport (Figure 2.2.5)

FIGURE 3.15 : Chronogramme gestion des dépendances de saut par *fetch_nop_o*



Note : Retour au Rapport (Figure 2.8)

TD 3

FIGURE 3.16 : Programme exo3_2ways.S

```
.section .start
.globl start
.globl lab
.globl shifter

start:
   addi x6,x0,15
   // a=0xA5A5
   lui x1,0xA
   nop
   nop
   nop
   addi x1,x1,0x5A5
   // b=7
   addi x2,x0,7
   nop
   nop
   //  bit 0 de a
lab:
   andi x3,x1,1
   nop
   nop
   nop
   // si le bit 0 de a est  zero on accumule b*2
   beqz x3,lab1
   // on acummule
   nop
   nop
   add x4,x4,x2
lab1:
// on decale a vers la droite
   //srai x1,x1,1
   // on decale b vers la gauche
   //slli x2,x2,1
   jal  x8,shifter
   nop
   addi x5,x5,1
   nop
   nop
   nop
   bge  x6,x5,lab
   nop
   nop
   //resultat final dans x7 =0x48783
   add x7,x0,x4
lab3:
   j lab3
   nop
   nop

.section .way2
shifter :
   srai x1,x1,1
   slli x2,x2,1
   jalr x0,x8
   nop
   nop
```

Note : Retour au Rapport (Figure 3.3)

FIGURE 3.17 : Programme exo3_2ways_sans_nop.S

```
// Programme de multiplication entre x1 et x2
// Le résultat de la multiplication est stocké dans x7
// Exemple:
// x1 = 0xA5A5 et x2 = 0x7 => x7 =0x48783

.section .start
.globl start
.globl lab
.globl shifter

start:
   // Dans x1: 0xA5A5
   // Dans x2; 0x7
   // Dans x6: x0F
   addi x6,x0,15
   lui  x1,0xA
   addi x1,x1,0x5A5
   addi x2,x0,7
lab:
   // Si x3 % 2 == 0, goto lab1
   // Sinon, x4 += x2
   andi x3,x1,1
   beqz x3,lab1
   add  x4,x4,x2
lab1:
   // Appelle la fonction shifter
   // x5 += 1
   // Si x6 >= x5, aller à lab
   // Sinon x7 = x4
   jal  x8,shifter
   addi x5,x5,1
   bge  x6,x5,lab
   add  x7,x0,x4
lab3:
   j   lab3
   nop
   nop

.section .way2
shifter:
   // Divise x1 par 2
   // Divise x2 par 2
   // Retourne après le jal  x8,shifter
   srai x1,x1,1
   slli x2,x2,1
   jalr x0,x8
   nop // La gestion de JALR n'est pas implémenté
   nop
```

Note : Retour au Rapport (Figure 3.3)