Laboratoire #2

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Conception d'un microprocesseur

ELE8304 - Circuit intégrés à très grande échelle

Automne 2025
Département de génie électrique
École Polytechnique de Montréal

Dernière mise à jour: 8 janvier 2025

Ait Abdeslam, Massil 2153204

Table des matières

Table des figures

1 Introduction

Ce rapport présente l'implémentation et la vérification d'un mini-processeur basé sur une architecture RISC-V avec un pipeline à 5 étages. Nous avons commencé par décrire et tester les quatre modules principaux du processeur : l'Adder, l'ALU, le compteur de programme et le banc de registres. Chacun de ces modules a été analysé et validé individuellement pour s'assurer de leur bon fonctionnement.Nous avons ensuite détaillé les différents étages du pipeline du processeur, en expliquant le rôle de chaque étape dans l'exécution des instructions. À travers l'exécution de quatre types d'instructions (arithmétiques, de branchement, de gestion de la mémoire, et spécialisées), nous avons démontré comment le cœur du processeur traite les instructions dans le pipeline.Enfin, le rapport aborde les étapes de synthèse et de placement-routage du processeur. Nous avons décrit les scripts utilisés pour ces processus et effectué des simulations post-synthèse et post-placement-routage afin de vérifier la validité du design. Ces simulations ont permis de valider le bon fonctionnement du processeur dans sa version finale.

2 Modules

La présente section porte sur les modules composant le processeur. Pour chaque module, nous fournirons d'abord une description générale de son fonctionnement, suivie d'une présentation de son interface, puis des résultats de la simulation associée au module.

2.1 Adder

2.1.1 Fonctionnement

Le module adder permet d'additionner et de soustraire des entiers de N bits (N étant un paramètre générique, N=32 dans notre processeur). Il est réalisé à partir d'une série de half-adders et prend en charge l'addition et la soustraction de nombres signés ou non signés.

2.1.2 Interface

Interface de l'adder

```
entity riscv_adder is
  generic (
    N : positive := 32
  );
  port (
    i_a    : in  std_logic_vector(N-1 downto 0);
    i_b    : in  std_logic_vector(N-1 downto 0);
    i_sign : in  std_logic;  -- '0' for unsigned, '1' for signed
    i_sub  : in  std_logic;  -- '0' for addition, '1' for subtraction
    o_sum  : out std_logic_vector(N downto 0)
  );
end entity riscv_adder;
```

Comme expliqué précédemment, le module adder permet d'effectuer l'addition ou la soustraction des valeurs contenues dans les signaux i_a et i_b. Le signal i_sign détermine si l'opération porte sur des entiers signés ou non signés. Lorsque i_sign est à 1, l'opération est réalisée sur des entiers signés, ce qui a pour effet d'étendre les signaux i_a et i_b en tant qu'entiers signés. En revanche, lorsque i_sign est à 0, l'opération se fait sur des entiers non signés, ce qui modifie l'extension des signaux i_a et i_b. Si le signal i_sub est à 0, l'opération effectuée est une addition. Dans le cas contraire, le complément à deux du signal i_b est pris, ce qui revient à effectuer une soustraction. Le résultat de l'opération est stocké dans le signal o_sum, qui contient à la fois le résultat (32 bits) et le carry (1 bit).

2.1.3 Simulation

Pour tester ce module, nous avons écrit un banc de test qui vérifie que différentes opérations produisent les résultats attendus. Étant donné que le circuit est combinatoire, aucune analyse de timing n'est nécessaire. Nous avons ainsi testé l'addition et la soustraction, tant pour des entiers signés que non signés, en prenant également en compte les cas où l'opérande 2 est supérieur à l'opérande 1 dans une soustraction non signée. Les résultats de cette simulation sont présentés ci-dessous dans la console de simulation de l'outil Vivado.

```
Note: Test 1 Passed: Unsigned addition (10 + 20) = 30
Time: 40 ns  Iteration: 0  Process: /tb_riscv_adder/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_adder_tb.vhd
Note: Test 2 Passed: Unsigned subtraction (30 - 20) = 10
Time: 80 ns  Iteration: 0  Process: /tb_riscv_adder/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_adder_tb.vhd
Note: Test 3 Passed: Unsigned subtraction (15 - 20) correctly underflows
Time: 120 ns  Iteration: 0  Process: /tb_riscv_adder/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_adder_tb.vhd
Note: Test 4 Passed: Signed addition (10 + 20) = 30
Time: 160 ns  Iteration: 0  Process: /tb_riscv_adder/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_adder_tb.vhd
Note: Test 5 Passed: Signed addition (10 + -5) = 5
Time: 200 ns  Iteration: 0  Process: /tb_riscv_adder/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_adder_tb.vhd
Note: Test 6 Passed: Signed subtraction (-10 - 5) = -15
Time: 240 ns  Iteration: 0  Process: /tb_riscv_adder/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_adder_tb.vhd
Note: Test 7 Passed: Signed subtraction (-10 - -20) = 10
Time: 280 ns  Iteration: 0  Process: /tb_riscv_adder/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_adder_tb.vhd
```

Résultats de la simulation de l'adder

2.2 ALU

2.2.1 Fonctionnement

Le module ALU (Unité Arithmétique et Logique) réalise les opérations arithmétiques et logiques du processeur. Il est composé de deux sous-modules principaux : un shifter, qui effectue des décalages

à gauche, ainsi que des décalages logiques et arithmétiques à droite, et un adder, qui permet d'effectuer des additions et des soustractions. De plus, l'ALU peut réaliser des opérations logiques telles que ET, OU et XOR. Toutes ces opérations sont effectuées sur des mots de 32 bits.

2.2.2 Interface

Interface de l'ALU

```
entity riscv_alu is
  port (
    i_arith  : in  std_logic;                                  -- Arith/
       Logic
    i_sign   : in  std_logic;                                  -- Signed/
       Unsigned
    i_opcode : in  std_logic_vector(ALUOP_WIDTH-1 downto 0); -- ALU
       opcodes
    i_shamt  : in  std_logic_vector(SHAMT_WIDTH-1 downto 0); -- Shift
       Amount
    i_src1   : in  std_logic_vector(XLEN-1 downto 0);          -- Operand A
    i_src2   : in  std_logic_vector(XLEN-1 downto 0);          -- Operand B
    o_res    : out std_logic_vector(XLEN-1 downto 0));         -- Result
end entity riscv_alu;
```

Les signaux i_src1 et i_src2 représentent les opérandes. Le signal i_opcode détermine le type d'opération à effectuer encodé sur 3 bits : ADD (addition ou soustraction), SL (shift à gauche), SR (shift à droite, logique ou arithmétique), SLT (comparaison i_src1 ¡ i_src2), ainsi que les opérations logiques XOR, OR et AND. Chaque opcode correspond à un encodage sur 3 bits, par exemple ADD correspond à 000. Les différents opcode sont défini dans *riscv_pkg.vhd*. La sortie o_res dépend donc de la valeur du signal i_opcode.Pour les opérations de décalage (shift), le nombre de bits à décaler est déterminé par la valeur du signal i_shamt. La nature de l'addition ou de la soustraction dépend du signal i_arith, comme expliqué dans la section dédiée à l'addder. Enfin, dans le cas d'un décalage à droite (shift right), si i_arith est actif, un décalage logique est effectué ; sinon, un décalage arithmétique est réalisé.

2.2.3 Simulation

Pour tester l'ALU, nous avons développé un testbench afin de vérifier le bon fonctionnement des différentes opérations réalisées par l'unité. Étant donné que l'ALU est un circuit combinatoire, aucune analyse de timing n'a été effectuée. Chaque type d'opération a été testé pour garantir la correcte exécution des fonctionnalités de l'ALU. Les résultats de cette simulation sont présentés ci-dessous dans la console de l'outil de simulation Vivado.

```
Note: Test 1 Passed: Addition (20 + 10) = 30
Time: 25 ns  Iteration: 0  Process: /tb_riscv_alu/stim_process  File: C:/Users/
   User/Desktop/VLSI_lab2/sources/riscv_alu_tb.vhd
Note: Test 2 Passed: Substraction (20-10) = 10
```

```
Time: 50 ns  Iteration: 0  Process: /tb_riscv_alu/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_alu_tb.vhd
Note: Test 3 Passed: SLT result outputs 1
Time: 75 ns  Iteration: 0  Process: /tb_riscv_alu/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_alu_tb.vhd
Note: Test 4 Passed: SLT result outputs 0
Time: 100 ns  Iteration: 0  Process: /tb_riscv_alu/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_alu_tb.vhd
Note: Test 5 Passed: Shift left of 1 = 2
Time: 125 ns  Iteration: 0  Process: /tb_riscv_alu/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_alu_tb.vhd
Note: Test 6 Passed: Logical shift right of 2 = 1
Time: 150 ns  Iteration: 0  Process: /tb_riscv_alu/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_alu_tb.vhd
Note: Test 7 Passed: Arithmetic shift right of -2 = -1
Time: 175 ns  Iteration: 0  Process: /tb_riscv_alu/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_alu_tb.vhd
Note: Test 8 Passed: 1 Xor 0 = 1
Time: 200 ns  Iteration: 0  Process: /tb_riscv_alu/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_alu_tb.vhd
Note: Test 9 Passed: 1 Xor 1 = 0
Time: 225 ns  Iteration: 0  Process: /tb_riscv_alu/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_alu_tb.vhd
Note: Test 10 Passed: 1 or 1 = 1
Time: 250 ns  Iteration: 0  Process: /tb_riscv_alu/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_alu_tb.vhd
Note: Test 11 Passed: 1 or 0 = 1
Time: 275 ns  Iteration: 0  Process: /tb_riscv_alu/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_alu_tb.vhd
Note: Test 12 Passed: 1 and 0 = 0
Time: 300 ns  Iteration: 0  Process: /tb_riscv_alu/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_alu_tb.vhd
Note: Test 13 Passed: 1 and 1 = 1
Time: 325 ns  Iteration: 0  Process: /tb_riscv_alu/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_alu_tb.vhd
```

Résultats de la simulation de l'ALU

2.3 Compteur de Programme

2.3.1 Fonctionnement

Le program counter (PC) garde une trace de l'adresse de la prochaine instruction à exécuter. À chaque cycle d'horloge, il est mis à jour pour pointer vers l'instruction suivante, soit par un simple incrément, soit en conservant sa valeur précédente (stall), soit vers une adresse cible spécifiée (jump). Le program counter utilise deux paramètres génériques : XLEN (la taille du bus, ici 32 bits) et RESET_VECTOR (l'adresse de départ, fixée à 0 dans notre cas).

2.3.2 Interface

Interface du program counter

```
entity riscv_pc is
  generic (RESET_VECTOR : natural := 16#00000000#);
  port (
    i_clk       : in  std_logic;
    i_rstn      : in  std_logic;
    i_stall     : in  std_logic;
    i_transfert : in  std_logic;
    i_target    : in  std_logic_vector(XLEN-1 downto 0);
    o_pc        : out std_logic_vector(XLEN-1 downto 0));
end entity riscv_pc;
```

Le signal i_clk correspond à l'horloge, tandis que i_rstn représente le signal de réinitialisation (ici, un reset actif bas). En fonctionnement normal (avec i_stall et i_transfert à 0), le programme counter incrémente sa valeur de 4 à chaque cycle d'horloge. Lorsque i_stall est à 1, la valeur du programme counter reste inchangée. Si i_transfert est à 1, le programme counter prend la valeur contenue dans i_target au cycle suivant. L'adresse pointée par le programme counter est disponible sur le signal o_pc. À noter que si les deux signaux i_stall et i_transfert sont à 1, le signal i_stall est prioritaire.

2.3.3 Simulation

Pour tester le programme counter, nous avons développé un testbench afin de vérifier son bon fonctionnement. Étant un module séquentiel, nous avons vérifié que les signaux prenaient les valeurs attendues après chaque front montant. Nous avons testé le reset du PC, son fonctionnement normal (incrément de 4 à chaque cycle), ainsi que les scénarios de transfert et de stall. Les résultats de cette simulation sont présentés ci-dessous dans la console de l'outil de simulation Vivado.

```
Note: Test Case 1 Passed: PC reset to the reset vector.
Time: 20 ns  Iteration: 0  Process: /tb_riscv_pc/stim_process  File: C:/Users/User
    /Desktop/VLSI_lab2/sources/riscv_pc_tb.vhd
Note: Test Case 2 Passed: PC incremented correctly over multiple cycles.
Time: 100 ns  Iteration: 0  Process: /tb_riscv_pc/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_pc_tb.vhd
Note: Test Case 3 Passed: PC correctly transferred to target address.
Time: 120 ns  Iteration: 0  Process: /tb_riscv_pc/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_pc_tb.vhd
Note: Test Case 4 Passed: PC correctly remained stalled.
Time: 180 ns  Iteration: 0  Process: /tb_riscv_pc/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_pc_tb.vhd
Note: Test Case 4 Passed: PC incremented correctly after stall was removed.
Time: 200 ns  Iteration: 0  Process: /tb_riscv_pc/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_pc_tb.vhd
```

Résultats de la simulation du progam counter

2.4 Banc de registre

2.4.1 Fonctionnement

Le banc de registres (RF) permet de gérer l'accès en lecture et en écriture aux 32 registres de 32 bits. Il est responsable de stocker et fournir des données à partir de registres spécifiques en fonction des adresses d'entrée. Lors d'un cycle d'horloge, l'écriture et la lecture des données se fait sur le front montant. Lorsqu'une réinitialisation asynchrone est effectuée, tous les registres sont remis à zéro. Le module veille à ce que les valeurs des registres soient mises à jour ou lues correctement en fonction des signaux d'adresse et de contrôle.

2.4.2 Interface

Interface du Banc de re registres

```
entity riscv_rf is
  port (
    i_clk     : in  std_logic;
    i_rstn    : in  std_logic;
    i_we      : in  std_logic;
    i_addr_ra : in  std_logic_vector(REG_WIDTH-1 downto 0);
    o_data_ra : out std_logic_vector(XLEN-1 downto 0);
    i_addr_rb : in  std_logic_vector(REG_WIDTH-1 downto 0);
    o_data_rb : out std_logic_vector(XLEN-1 downto 0);
    i_addr_w  : in  std_logic_vector(REG_WIDTH-1 downto 0);
    i_data_w  : in  std_logic_vector(XLEN-1 downto 0));
end entity riscv_rf;
```

À chaque cycle, deux registres peuvent être lus : le registre désigné par i_addr_ra et sa valeur o_data_ra, ainsi que le registre i_addr_rb et sa valeur o_data_rb. Un seul registre peut être écrit, avec la valeur i_data_w au registre i_addr_w, mais l'écriture n'a lieu que si le signal i_we est à high. Si un registre est à la fois lu et écrit, le signal de sortie correspondant (o_data_ra ou o_data_rb) prend la valeur du signal i_data_w. À noter que le registre 0 contient toujours la valeur 0.

2.4.3 Simulation

Pour valider le bon fonctionnement du banc de registres, nous avons développé un testbench. Ce dernier vérifie les différents modes de fonctionnement du banc de registres, notamment l'écriture et la lecture sur divers registres. Nous avons testé spécifiquement le comportement du registre 0 (qui doit toujours contenir la valeur 0), le fonctionnement du signal write enable, ainsi que le reset. De plus, nous avons vérifié la lecture simultanée des mêmes registres (ra et rb) et testé la lecture et l'écriture du même registre sur le même cycle. Les résultats de cette simulation sont présentés ci-dessous dans la console de l'outil de simulation Vivado.

```
Note: Test 1 Passed: Register 1 correctly written with value 1.
Time: 80 ns  Iteration: 0  Process: /tb_riscv_rf/stim_process  File: C:/Users/User
   /Desktop/VLSI_lab2/sources/riscv_rf_tb.vhd
```

```
Note: Test 2 Passed: Register 2 correctly written with value 2.
Time: 140 ns  Iteration: 0  Process: /tb_riscv_rf/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_rf_tb.vhd
Note: Test 3 Passed: Register 0 remains 0 as expected.
Time: 200 ns  Iteration: 0  Process: /tb_riscv_rf/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_rf_tb.vhd
Note: Test 4 Passed: Register 3 was not written to when write enable was off.
Time: 260 ns  Iteration: 0  Process: /tb_riscv_rf/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_rf_tb.vhd
Note: Test 5 Passed: All registers correctly reset to 0.
Time: 310 ns  Iteration: 0  Process: /tb_riscv_rf/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_rf_tb.vhd
Note: Test 6 Passed: both values are at the correct value.
Time: 350 ns  Iteration: 0  Process: /tb_riscv_rf/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_rf_tb.vhd
Note: Test 7 Passed: read and right the same registry takes data_w value.
Time: 390 ns  Iteration: 0  Process: /tb_riscv_rf/stim_process  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/riscv_rf_tb.vhd
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_riscv_rf_behav' loaded.
```

Résultats de la simulation du Banc de registres

3 Core

Dans cette section, nous présenterons le fonctionnement à haut niveau ainsi que l'interface de chacun des 5 étages du pipeline. Nous exposerons ensuite les résultats des simulations réalisées pour chaque étage. Enfin, nous illustrerons le fonctionnement du cœur du processeur à travers l'exécution d'une instruction arithmétique, d'un branchement, d'une opération sur la mémoire, et enfin de l'opération spécialisée.

3.1 Pipeline

3.1.1 Instruction Fetch (IF)

L'étage Instruction Fetch (IF) est responsable de la récupération des nouvelles instructions depuis la mémoire d'instructions. À chaque cycle, il transmet l'instruction ainsi que son adresse mémoire à l'étage suivant. Cet étage utilise le program counter défini dans la section précédente. Cependant, trois points doivent être pris en compte. Premièrement, la mémoire a un cycle de latence, ce qui entraîne un décalage d'un cycle entre l'instruction fournie par la mémoire et la sortie du program counter. Pour compenser, un registre supplémentaire est utilisé afin de conserver l'adresse de l'instruction précédemment lue. Deuxièmement, l'étage doit être capable de stall si l'étage Execute le demande, par exemple, lors d'un conflit de données nécessitant un cycle d'attente. Enfin, l'étage doit pouvoir effectuer un flush si l'étage Execute le demande, notamment en cas de branchement, où il doit annuler l'instruction en cours et prendre en compte la prochaine instruction déterminée par le résultat du branchement. L'interface du module est présentée ci-dessous.

Interface de l'étage Instruction Fetch

```
entity riscv_instruction_fetch is
    port (
      i_clk       : in  std_logic;
      i_rstn      : in  std_logic;

      --From EX
      i_stall     : in  std_logic;
      i_flush     : in  std_logic;
      i_transfert : in  std_logic;
      i_target    : in  std_logic_vector(XLEN-1 downto 0);

      --Instruction memory signals
      i_imem_read : in  std_logic_vector(XLEN-1 downto 0);
      o_imem_addr : out std_logic_vector(XLEN-1 downto 0);

      --To decode
      o_pc_current   : out std_logic_vector(XLEN-1 downto 0);
      o_instr        : out std_logic_vector(XLEN-1 downto 0));
end entity riscv_instruction_fetch;
```

Les signaux i_clk et i_rstn sont respectivement les signaux d'horloge et de réinitialisation. L'étage Execute envoie le signal i_stall à l'étage IF, lui indiquant de ne pas envoyer la prochaine instruction (stall). Le signal i_flush, lorsqu'il est actif, ordonne de flush l'instruction en cours et d'utiliser celle présente dans le signal i_target (dans ce cas, i_transfert suit la valeur de i_flush). Ensuite, les signaux i_imem_read et o_imem_addr spécifient l'adresse à lire dans la mémoire d'instructions et l'instruction lue. Enfin, les signaux o_pc_current et o_instr sont transmis à l'étage suivant, contenant respectivement l'instruction et son adresse.

Pour tester l'étage, nous avons développé un testbench qui vérifie le bon fonctionnement du reset, du fonctionnement normal, ainsi que des mécanismes de stall, de flush et de transfert. Ce testbench permet de s'assurer que chaque fonctionnalité de l'étage fonctionne correctement dans différents scénarios. Les résultats de cette simulation sont présentés ci-dessous dans la console de l'outil de simulation Vivado.

```
Note: Test Case 1 Passed: PC reset to the reset vector.
Time: 100 ns  Iteration: 0  Process: /tb_riscv_instruction_fetch/stimulus_process
     File: C:/Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_fetch
    /riscv_instruction_fetch_tb.vhd
Note: Test Case 2 Passed: Instruction fetched successfully.
Time: 300 ns  Iteration: 0  Process: /tb_riscv_instruction_fetch/stimulus_process
     File: C:/Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_fetch
    /riscv_instruction_fetch_tb.vhd
Note: Test Case 3 Passed: Instruction held during stall.
Time: 400 ns  Iteration: 0  Process: /tb_riscv_instruction_fetch/stimulus_process
     File: C:/Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_fetch
    /riscv_instruction_fetch_tb.vhd
Note: Test Case 3.1 Passed: Instruction updated after stall release.
Time: 500 ns  Iteration: 0  Process: /tb_riscv_instruction_fetch/stimulus_process
     File: C:/Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_fetch
```

```
   /riscv_instruction_fetch_tb.vhd
Note: Test Case 4 Passed: NOP instruction inserted during flush.
Time: 600 ns  Iteration: 0  Process: /tb_riscv_instruction_fetch/stimulus_process
    File: C:/Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_fetch
   /riscv_instruction_fetch_tb.vhd
Note: Test Case 5 Passed: PC updated to jump target.
Time: 700 ns  Iteration: 0  Process: /tb_riscv_instruction_fetch/stimulus_process
    File: C:/Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_fetch
   /riscv_instruction_fetch_tb.vhd
```

Résultats de la simulation de l'étage Instruction Fetch

3.1.2 Instruction Decode (ID)

L'étage Instruction Decode (ID) a pour fonction de décoder l'instruction en cours, en identifiant son opcode, ainsi que les champs funct3 et funct7. Cet étage contient également le banc de registres et est chargé de lire les valeurs des registres source, lorsque cela est nécessaire. Il génère les flags qui permettront à l'étage Execute de traiter les données correctement. De plus, l'étage Instruction Decode produit la valeur immédiate sur 32 bits, si applicable. En résumé, cet étage prépare l'instruction de manière à ce que l'étage Execute puisse la traiter au cycle suivant.

Interface de l'étage Instruction Decode

```
entity riscv_instruction_decode is
    port (
      i_clk       : in  std_logic;
      i_rstn      : in  std_logic;

      -- From WB
      i_wb      : in  std_logic;
      i_rd_addr : in  std_logic_vector(REG_WIDTH-1 downto 0);
      i_rd_data : in  std_logic_vector(XLEN-1 downto 0);

      -- From EX
      i_flush   : in  std_logic;
      i_stall   : in  std_logic;


      -- From IF
      i_pc_current   : in std_logic_vector(XLEN-1 downto 0);
      i_instr        : in std_logic_vector(XLEN-1 downto 0);

      --To EX :
      -- Registers
      o_rs1_data  : out std_logic_vector(XLEN-1 downto 0); --register 1
         data
      o_rs2_data  : out std_logic_vector(XLEN-1 downto 0); --register 2
         data
      o_rs1_addr   : out std_logic_vector(REG_WIDTH-1 downto 0); --
         register 1  adress
```

```
        o_rs2_addr    : out std_logic_vector(REG_WIDTH-1 downto 0); --
           register 2  adress
        o_rd_addr    : out std_logic_vector(REG_WIDTH-1 downto 0); --
           Destination register adress

        -- ALU
        o_arith      : out  std_logic;                                --
           Arith/Logic
        o_sign       : out  std_logic;                                --
           Signed/Unsigned
        o_opcode     : out  std_logic_vector(ALUOP_WIDTH-1 downto 0); -- ALU
           opcodes
        o_shamt      : out  std_logic_vector(SHAMT_WIDTH-1 downto 0); --
           Shift Amount

        -- Immediate value
        o_imm        : out  std_logic_vector(XLEN-1 downto 0); -- Immediate
           value

        -- Flags
        o_jmp        : out std_logic;    --jmp instr
        o_jalr       : out std_logic;    --is jalr instr
        o_brnch      : out std_logic;    --branch instr
        o_src_imm    : out std_logic;    --immediate value
        o_rshmt      : out std_logic;    --use rs2 for shamt
        o_wb         : out std_logic;    --write register back
        o_we         : out std_logic;    --write memory
        o_re         : out std_logic;   --read memory

        --Special Instruction
        o_spc       : out std_logic;  --Is special instr
        o_odd       : out std_logic;  --Is func3 odd
        o_neg       : out std_logic;  --Is func3 negative

        --Direct transfert from IF
        o_pc_current    : out std_logic_vector(XLEN-1 downto 0)); --
           pc_current value


end entity riscv_instruction_decode;
```

Les signaux i_clk et i_rstn correspondent respectivement aux signaux d'horloge et de réinitialisation. L'étage Instruction Decode reçoit de l'étage Write-back la donnée à écrire, le registre cible ainsi qu'un signal indiquant si l'écriture doit être effectuée. De l'étage Execute, il reçoit les signaux de flush (en cas de branchement) et de stall (en cas de conflit de données). De l'étage Instruction Fetch, il reçoit l'instruction à décoder ainsi que son adresse, qu'il transmet directement à l'étage Execute.L'étage Instruction Decode envoie à l'étage Execute les valeurs des registres utilisées par

l'instruction, ainsi que leurs adresses, et l'adresse du registre de destination. Il transmet également les signaux relatifs à l'ALU (voir la section sur l'ALU), ainsi que la valeur immédiate étendue sur 32 bits. Enfin, il envoie les différents flags nécessaires pour que l'étage Execute puisse traiter correctement l'instruction. L'utilisation de ces flags sera détaillée dans la section dédiée à l'étage Execute. Des signaux spécifiques à l'instruction spécialisée sont également transmis.

L'étage Instruction Decode (ID) utilise deux modules : le module Banc de registres et un module appelé Decode, qui se charge du pré-décodage et du décodage des instructions. Le module Decode commence par lire l'opcode de l'instruction et en déduit son type. Il étend ensuite la valeur immédiate si nécessaire, et détermine l'instruction en fonction de l'opcode, du func3 et du func7 (si applicable). Pour les instructions arithmétiques, il génère les signaux appropriés pour l'ALU. Si l'instruction utilise une valeur immédiate, il met le flag o_src_imm à 1, indiquant à l'étage Execute d'utiliser cette valeur immédiate au lieu du registre rs2. En cas de branchement, il configure les flags de branchement avec les bonnes valeurs. L'étage Decode traite reconnait l'instructions spécialisée. Enfin, il envoie l'adresse des registres au Banc de registres pour la lecture des données.

Pour la simulation de l'étage Instruction Decode, nous avons réalisé deux simulations distinctes. La première consiste à tester le module Decode pour vérifier que chaque instruction est correctement décodée, c'est-à-dire que les signaux transmis à l'étage Execute sont bien ceux attendus pour chaque instruction. La deuxième simulation concerne l'étage Instruction Decode dans son ensemble, où nous avons testé la lecture et l'écriture des registres, ainsi que les mécanismes de flush et de stall. En cas d'anomalie dans les signaux, une erreur est affichée dans la console.

```
Note: Result for instruction LUI :
Time: 10 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction JAL :
Time: 20 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction JALR :
Time: 30 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction BEQ :
Time: 40 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction LW :
Time: 50 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction SW :
Time: 60 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction ADDI :
Time: 70 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction SLTI :
Time: 80 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction SLTIU :
Time: 90 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
```

```
Note: Result for instruction XORI :
Time: 100 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction ORI :
Time: 110 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction ANDI :
Time: 120 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction SLLI :
Time: 130 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction SRLI :
Time: 140 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction SRAI :
Time: 150 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction ADD :
Time: 160 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction SUB :
Time: 170 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction SLL :
Time: 180 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction SLT :
Time: 190 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction SLTU :
Time: 200 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction XOR :
Time: 210 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction SRL :
Time: 220 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction SRA :
Time: 230 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction OR :
Time: 240 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction AND :
Time: 250 ns  Iteration: 0  Process: /decode_tb/line__55  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
Note: Result for instruction ESWP :
Time: 260 ns  Iteration: 0  Process: /decode_tb/line__61  File: C:/Users/User/
    Desktop/VLSI_lab2/sources/pipeline/riscv_instruction_decode/decode_tb.vhd
```

Résultats de la simulation du module Decode

```
Note: Result for instruction ESWP :
Time: 400 ns  Iteration: 0  Process: /tb_riscv_instruction_decode/stimulus_process
      File: C:/Users/User/Desktop/VLSI_lab2/sources/pipeline/
    riscv_instruction_decode/riscv_instruction_decode_tb.vhd
Note: Result for instruction ADD :
Time: 500 ns  Iteration: 0  Process: /tb_riscv_instruction_decode/stimulus_process
      File: C:/Users/User/Desktop/VLSI_lab2/sources/pipeline/
    riscv_instruction_decode/riscv_instruction_decode_tb.vhd
Note: Result for stalling :
Time: 600 ns  Iteration: 0  Process: /tb_riscv_instruction_decode/stimulus_process
      File: C:/Users/User/Desktop/VLSI_lab2/sources/pipeline/
    riscv_instruction_decode/riscv_instruction_decode_tb.vhd
Note: Result for flushing :
Time: 700 ns  Iteration: 0  Process: /tb_riscv_instruction_decode/stimulus_process
      File: C:/Users/User/Desktop/VLSI_lab2/sources/pipeline/
    riscv_instruction_decode/riscv_instruction_decode_tb.vhd
```

Résultats de la simulation de l'étage Instruction Decode

3.1.3 Execute (EX)

L'étage Execute a pour rôle d'exécuter l'opération définie par l'instruction en cours. Il contient un ALU pour effectuer les opérations arithmétiques, et gère également les branchements. À l'aide d'un adder, il génère l'adresse cible du branchement et en parallèle, il génère le signal de branchement, qu'il transmet aux modules précédents pour un éventuel flush. L'étage Execute reçoit également les données en forwarding des étages précédents et gère les conflits de données, en utilisant les données forwardaient ou avec un stall. De plus, il intègre un module Special-Instruction pour traiter les instructions spécialisées. En résumé, l'étage Execute exécute les opérations prévues par chaque instruction tout en gérant efficacement les conflits de données et de contrôle.

Interface de l'étage Execute

```
entity riscv_execute is
    port (
      i_clk       : in  std_logic;
      i_rstn      : in  std_logic;

      -- Forwarding
      -- From MEM
      i_mem_rd_addr : in std_logic_vector(REG_WIDTH-1 downto 0); --
         Destination register adress from memory stage
      i_mem_rd_data : in std_logic_vector(XLEN-1 downto 0); --Destination
         register data from memory
      i_mem_rd_wb   : in std_logic; -- Will the data be written
      i_mem_rd_re   : in std_logic; -- Will the data be read from memory (
         stall)

      -- From WB
      i_wb_rd_addr : in std_logic_vector(REG_WIDTH-1 downto 0); --
         Destination register adress from write back stage
```

```
i_wb_rd_data : in std_logic_vector(XLEN-1 downto 0); --Destination
   register data from memory
i_wb_rd_wb   : in std_logic; -- Will the data be written

--From ID :
-- Registers
i_rs1_data  : in std_logic_vector(XLEN-1 downto 0); --register 1
   data
i_rs2_data  : in std_logic_vector(XLEN-1 downto 0); --register 2
   data
i_rs1_addr  : in std_logic_vector(REG_WIDTH-1 downto 0); -- register
    1 adress
i_rs2_addr  : in std_logic_vector(REG_WIDTH-1 downto 0); -- register
    2 adress
i_rd_addr   : in std_logic_vector(REG_WIDTH-1 downto 0); --
   Destination register adress

-- ALU
i_arith     : in  std_logic;                                  -- Arith
   /Logic
i_sign      : in  std_logic;                                  --
   Signed/Unsigned
i_opcode    : in  std_logic_vector(ALUOP_WIDTH-1 downto 0); -- ALU
   opcodes
i_shamt     : in  std_logic_vector(SHAMT_WIDTH-1 downto 0); -- Shift
    Amount

-- Immediate value
i_imm       : in  std_logic_vector(XLEN-1 downto 0); -- Immediate
   value

-- Flags
i_jmp       : in std_logic;   --jmp instr
i_jalr       : in std_logic;   --is jal instr
i_brnch     : in std_logic;   --branch instr
i_src_imm   : in std_logic;   --immediate value
i_rshmt     : in std_logic;   --use rs2 for shamt
i_wb        : in std_logic;   --write register back
i_we        : in std_logic;   --write memory
i_re        : in std_logic;  --read memory

--Special Instruction
i_spc      : in std_logic;  --Is special instr
i_odd      : in std_logic;  --Is func3 odd
i_neg      : in std_logic;  --Is func3 negative

i_pc_current  : in std_logic_vector(XLEN-1 downto 0); --pc_current
   value
```

```
        -- Control

        o_stall     : out  std_logic;
        o_flush     : out  std_logic;
        o_transfert : out  std_logic;
        o_target    : out  std_logic_vector(XLEN-1 downto 0);

        -- To ME
        o_we          : out std_logic;   --write memory
        o_re          : out std_logic;  --read memory

        o_alu_result : out std_logic_vector(XLEN-1 downto 0); --alu_result
        o_wb         : out std_logic; -- write back result
        o_rd_addr    : out std_logic_vector(REG_WIDTH-1 downto 0); --
           Destination register adress

        o_store_data : out std_logic_vector(XLEN-1 downto 0));-- Adress to
           store in memory


end entity riscv_execute;
```

L'étage Execute reçoit d'abord les signaux de forwarding des étages suivants. Il compare l'adresse des registres qu'il utilise avec celles fournies par les étages Mem et WB (en priorité MEM). Si l'adresse des registres correspond et que les données doivent être écrites (lorsque le signal i_stage_rd_wb est actif), l'étage utilise la donnée en forwarding. Dans le cas du forwarding depuis l'étage Mem, si le registre correspond mais que la donnée doit être lue depuis la mémoire (signal i_mem_rd_re actif), le processeur effectue un stall pour attendre que la donnée soit disponible au cycle suivant.

Ensuite, l'étage Execute reçoit les données et les adresses des registres rs1 et rs2, ainsi que l'adresse du registre de destination (qu'il transmet directement à l'étage suivant). Il reçoit également les signaux de l'ALU et la valeur immédiate étendue sur 32 bits.

En fonction des flags reçus, les opérandes de l'ALU peuvent varier. Dans le cas classique, les opérandes de l'ALU sont rs1 et rs2 (après gestion des conflits). Si le signal i_src_imm est actif, la valeur immédiate est utilisée comme deuxième opérande de l'ALU. Pour une instruction de type jump (comme JAL ou JALR), avec le flag i_jmp actif, l'adresse de l'instruction actuelle est utilisée comme premier opérande et la valeur 4 comme deuxième opérande pour calculer la prochaine instruction.

Dans le cas d'une instruction de branchement, il faut aussi gérer le signal o_transfert. Pour une instruction jump (JAL ou JALR), lorsque le flag i_jmp est actif, l'étage effectue un flush et active le transfert. Pour une instruction de branchement (i_brnch actif), l'ALU doit retourner zéro pour activer le transfert. Un adder est utilisé pour calculer l'adresse cible. Dans le cas de JALR (i_jalr actif), l'adresse cible est calculée à partir de la valeur immédiate et de la valeur de rs1, tandis que

pour JAL et BEQ, ce sont la valeur immédiate et l'adresse de l'instruction qui sont utilisées.

Enfin, pour une instruction spécialisée (i_spc actif), le résultat du module spécialisé est utilisé à la place du résultat de l'ALU.

Pour garantir le bon fonctionnement de l'étage, nous avons développé un testbench afin de vérifier les résultats de différentes instructions, en s'assurant de tester au moins une instruction de chaque type. Dans le cas d'un signal de sortie incorrect, une erreur est signalée dans le terminal.Les résultats de cette simulation sont présentés ci-dessous dans la console de l'outil de simulation Vivado.

```
Note: Result for instruction ESWP 1 (BE->LE IMM) :
Time: 50 ns  Iteration: 0  Process: /tb_riscv_execute/stimulus_process  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_execute/riscv_execute_tb.
    vhd
Note: Result for instruction ESWP 2 (LE->BE RS1) :
Time: 75 ns  Iteration: 0  Process: /tb_riscv_execute/stimulus_process  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_execute/riscv_execute_tb.
    vhd
Note: Result for instruction LUI :
Time: 100 ns  Iteration: 0  Process: /tb_riscv_execute/stimulus_process  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_execute/riscv_execute_tb.
    vhd
Note: Result for instruction JALL :
Time: 120 ns  Iteration: 0  Process: /tb_riscv_execute/stimulus_process  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_execute/riscv_execute_tb.
    vhd
Note: Result for instruction JALR :
Time: 130 ns  Iteration: 0  Process: /tb_riscv_execute/stimulus_process  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_execute/riscv_execute_tb.
    vhd
Note: Result for instruction BEQ :
Time: 150 ns  Iteration: 0  Process: /tb_riscv_execute/stimulus_process  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_execute/riscv_execute_tb.
    vhd
Note: Result for instruction LW :
Time: 170 ns  Iteration: 0  Process: /tb_riscv_execute/stimulus_process  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_execute/riscv_execute_tb.
    vhd
Note: Result for instruction SW :
Time: 190 ns  Iteration: 0  Process: /tb_riscv_execute/stimulus_process  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_execute/riscv_execute_tb.
    vhd
Note: Result for instruction ADDI :
Time: 210 ns  Iteration: 0  Process: /tb_riscv_execute/stimulus_process  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_execute/riscv_execute_tb.
    vhd
Note: Result for instruction SLTI :
Time: 230 ns  Iteration: 0  Process: /tb_riscv_execute/stimulus_process  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_execute/riscv_execute_tb.
    vhd
Note: Result for instruction SLTUI :
Time: 250 ns  Iteration: 0  Process: /tb_riscv_execute/stimulus_process  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_execute/riscv_execute_tb.
```

```
    vhd
Note: Result for instruction XORI :
Time: 270 ns  Iteration: 0  Process: /tb_riscv_execute/stimulus_process  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_execute/riscv_execute_tb.
    vhd
Note: Result for instruction ADD :
Time: 290 ns  Iteration: 0  Process: /tb_riscv_execute/stimulus_process  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_execute/riscv_execute_tb.
    vhd
Note: Result for instruction SLL :
Time: 310 ns  Iteration: 0  Process: /tb_riscv_execute/stimulus_process  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_execute/riscv_execute_tb.
    vhd
```

Résultats de la simulation de l'étage Execute

3.1.4 Memory Acces (ME)

L'étage Memory Access permet d'effectuer des opérations de lecture et d'écriture dans la mémoire de données. Il remplit deux fonctions principales : l'écriture et la lecture de la mémoire. Lors de l'écriture, il enregistre le résultat de l'ALU (provenant de l'étage Execute) dans la mémoire à l'adresse spécifiée par l'étage Execute. En cas de lecture, il accède à l'adresse indiquée par l'étage Execute et transmet le résultat à l'étage Write-back.

Interface de l'étage Memory Acces

```
entity riscv_memory_acces is
    port (
      i_clk       : in  std_logic;
      i_rstn      : in  std_logic;

     -- Forwarding (to EX)
      o_mem_rd_addr : out std_logic_vector(REG_WIDTH-1 downto 0); --
         Destination register adress from memory stage
      o_mem_rd_data : out std_logic_vector(XLEN-1 downto 0); --Destination
          register data from memory
      o_mem_rd_wb   : out std_logic; -- Will the data be written
      o_mem_rd_re   : out std_logic; -- Will the data be read from memory
         (stall)

      -- Memory signals

      i_load_data  : in std_logic_vector(XLEN-1 downto 0); --Memory data
         read
      o_store_data : out std_logic_vector(XLEN-1 downto 0); --Memory data
         to write
      o_mem_adress : out std_logic_vector(XLEN-1 downto 0); --Memory
         adress
      o_we         : out std_logic;   --write memory
      o_re         : out std_logic;  --read memory
```

```
        -- From EX
        i_we          : in std_logic;    --write memory
        i_re          : in std_logic;   --read memory

        i_alu_result  : in std_logic_vector(XLEN-1 downto 0); --alu_result
        i_wb          : in std_logic; -- write back result
        i_rd_addr     : in std_logic_vector(REG_WIDTH-1 downto 0); --
            Destination register adress

        i_store_data  : in std_logic_vector(XLEN-1 downto 0);-- Adress to
            store in memory


      --To WB
      o_load_data   : out std_logic_vector(XLEN-1 downto 0); --Memory data
          read
      o_alu_result  : out std_logic_vector(XLEN-1 downto 0); --alu_result
      o_wb          : out std_logic; -- write back result
      o_rd_addr     : out std_logic_vector(REG_WIDTH-1 downto 0); --
          Destination register adress
      o_re_wb       : out std_logic);  --memory or alu result

end entity riscv_memory_acces;
```

Les signaux i_clk et i_rstn correspondent respectivement aux signaux d'horloge et de réinitialisation. Les signaux de forwarding à l'étage EX permettent de vérifier si les données lues dans les registres rs1 ou rs2 sont prêtes à être utilisées ou si elles proviennent d'une donnée qui n'a pas encore été écrite. Si la donnée ne dépend pas d'un accès mémoire, elle peut être utilisée immédiatement. Sinon, l'étage Execute déclenche un stall et attend que la donnée soit prête.Les signaux de mémoire gèrent l'accès à celle-ci. De l'étage Execute, nous recevons le résultat de l'ALU, l'adresse du registre de destination, ainsi que l'adresse mémoire où la donnée doit être stockée. Nous recevons également des flags qui déterminent si l'accès à la mémoire est autorisé. Ensuite, l'étage suivant reçoit le résultat de l'ALU, la valeur lue en mémoire, l'adresse du registre de destination, ainsi que les flags permettant de choisir si l'on utilise le résultat de l'ALU ou la donnée lue en mémoire, et si l'on doit écrire le résultat ou non.

Pour tester le module, nous avons développer un test bench qui vérifie le bon fonctionnement de module. On vérifie la lecture et l'écriture de la mémoire et si les données sont correctement transmises à l'étage write-back. Les résultats de cette simulation sont présentés ci-dessous dans la console de l'outil de simulation Vivado.

```
Note: Test Case 1 Write to memory.
Time: 20 ns  Iteration: 0  Process: /tb_riscv_memory_acces/stim_proc  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_memory_acces/
    riscv_memory_tb.vhd
Note: Test Case 2 Read from memory.
```

```
Time: 30 ns  Iteration: 0  Process: /tb_riscv_memory_acces/stim_proc  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_memory_acces/
    riscv_memory_tb.vhd
Note: Test Case 3 Pass ALU result to WB
Time: 40 ns  Iteration: 0  Process: /tb_riscv_memory_acces/stim_proc  File: C:/
    Users/User/Desktop/VLSI_lab2/sources/pipeline/riscv_memory_acces/
    riscv_memory_tb.vhd
```

Résultats de la simulation de l'étage Memory Access

3.1.5 Write-Back (WB)

L'étage Write-back permet d'écrire les données dans le registre appropriée. Il choisit simplement entre le résultat de l'alu et la donnée lue en mémoire. Il envoie ce résultat à l'étage Instruction Decode (qui contient le banc de registre). À noter que cet étage est combinatoire.

Interface de l'étage Memory Acces

```
entity riscv_write_back is
    port (

     -- Forwarding (to EX)
     o_wb_rd_addr : out std_logic_vector(REG_WIDTH-1 downto 0); --
        Destination register adress from wb stage
     o_wb_rd_data : out std_logic_vector(XLEN-1 downto 0); --Destination
        register data from wb stage
     o_wb_rd_wb   : out std_logic; -- Will the data be written


      -- From MEM
     i_load_data  : in std_logic_vector(XLEN-1 downto 0); --Memory data
        read
     i_alu_result : in std_logic_vector(XLEN-1 downto 0); --alu_result
     i_wb         : in std_logic; -- write back result
     i_rd_addr    : in std_logic_vector(REG_WIDTH-1 downto 0); --
        Destination register adress
     i_re_wb      : in std_logic;  --memory or alu result

     -- To ID
     o_wb      : out  std_logic;
     o_rd_addr : out  std_logic_vector(REG_WIDTH-1 downto 0);
     o_rd_data : out  std_logic_vector(XLEN-1 downto 0));

end entity riscv_write_back;
```

Tout comme pour l'étage Memory Access, l'étage EX utilise des signaux de forwarding pour vérifier si les données lues dans les registres rs1 ou rs2 sont prêtes à être utilisées, et les mettre à jour si nécessaire. De l'étage précédent, nous recevons la donnée lue en mémoire, le résultat de l'ALU, l'adresse du registre de destination, ainsi que des flags permettant de déterminer si l'on utilise les

données de l'ALU ou de la mémoire, et si l'écriture du résultat doit être effectuée. Enfin, nous transmettons à l'étage ID la donnée à écrire, l'adresse du registre de destination, et si on écrit le résultat ou non.

Étant donné que l'étage Write-back est relativement simple (un multiplexeur), le testbench associé est également très simple. Nous vérifions que l'étage transmet correctement chaque valeur et que le choix entre le résultat de l'ALU et celui de la mémoire est bien effectué. Les résultats de cette simulation sont présentés ci-dessous dans la console de l'outil de simulation Vivado.

```
Note: Test 1: Select ALU result (i_re_wb = '0')
Time: 10 ns  Iteration: 0  Process: /tb_riscv_write_back/line__48  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/pipeline/riscv_write_back/riscv_write_back_tb.
    vhd
Note: Test 2: Select load data (i_re_wb = '1')
Time: 20 ns  Iteration: 0  Process: /tb_riscv_write_back/line__48  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/pipeline/riscv_write_back/riscv_write_back_tb.
    vhd
Note: Test 3: Write-back enable (i_wb)
Time: 30 ns  Iteration: 0  Process: /tb_riscv_write_back/line__48  File: C:/Users/
    User/Desktop/VLSI_lab2/sources/pipeline/riscv_write_back/riscv_write_back_tb.
    vhd
```

Résultats de la simulation de l'étage Write-Back

3.2 Simulations

La sous-section suivante décrira le fonctionnement complet du cœur du processeur. Nous y illustrerons le déroulement des opérations arithmétiques, des opérations de branchement, des accès mémoire (SW et LW), ainsi que de l'instruction spécialisée. Pour ce faire, nous suivrons la propagation de l'instruction à travers le pipeline et analyserons la signification des différents signaux, afin de valider le bon déroulement de l'exécution de l'instruction.

3.2.1 Instruction arithmétique

Instructions arithméthiques testées

```
1    and   s1, zero, zero    // addr=00, instr=000074B3, x9 = 0
2    lui   t0, 0xFFFFF       // addr=04, instr=FFFFF2B7, x5  = 0xFFFFF000
3    ori   t1, zero, 0x7FF   // addr=08, instr=7FF06313, x6  = 0x000007FF
4    addi  t2, zero, -1      // addr=0c, instr=FFF00393, x7  = 0xFFFFFFFF
5    xori  t3, zero, 2       // addr=10, instr=00204E13, x28 = 0x00000002
6    andi  t4, t1, 1         // addr=14, instr=00137E93, x29 = 0x00000001
7    slli  a0, t1, 4         // addr=18, instr=00431513, x10 = 0x00007FF0
8    srli  a1, t0, 4         // addr=1c, instr=0042D593, x11 = 0x0FFFFF00
```

La figure 1 présente les résultats de la simulation du code ci-dessus. Les lignes roses délimitent les différents étages du pipeline, tandis que les lignes rouges séparent les différentes instructions. Les numéros en bleus serviront à identifier des cycles dans l'explication. Seuls les signaux pertinents

pour démontrer le fonctionnement du processeur sont affichés. Notons qu'on représente les signaux sortant du Core donc les adresses réellement lues depuis la mémoire sont divisés par 4.

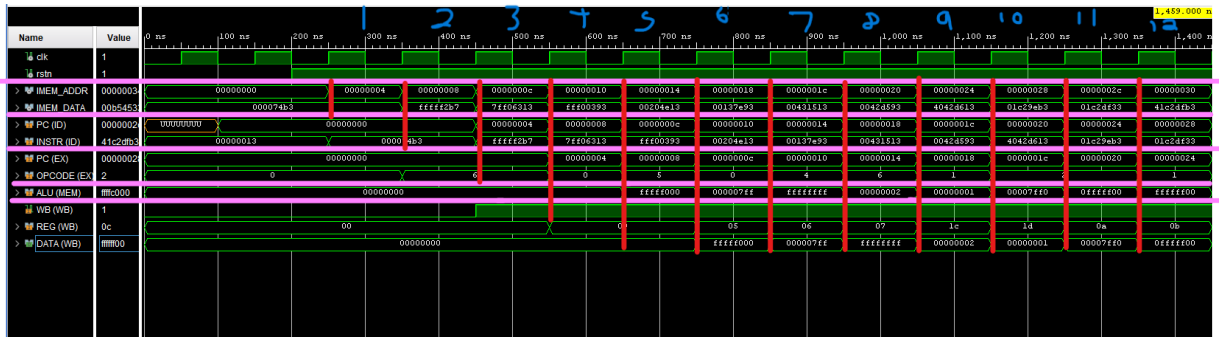

FIGURE 1 – Simulation d'instructions arithmétiques

Nous montrons d'abord la progression des instructions dans le Pipeline : **Avant Cycle 1 :** Reset

- **Cycle 1 :**
  - IF : Instruction lue 0x000074B3 (AND). Prochaine instruction à lire : 0x04.
  - ID : Vide
  - EX : Vide
  - ME : Vide
  - WB : Vide
- **Cycle 2 :**
  - IF : Instruction lue 0xFFFFF2B7 (LUI). Prochaine instruction à lire : 0x08.
  - ID : Instruction 0x000074B3 (AND). Adresse : 0x00.
  - EX : Vide
  - ME : Vide
  - WB : Vide
- **Cycle 3 :**
  - IF : Instruction lue 0x7FF06313 (ORI). Prochaine instruction à lire : 0x0C.
  - ID : Instruction 0xFFFFF2B7 (LUI). Adresse : 0x04.
  - EX : Adresse : 0x00. ALU-OPCODE : 6 (AND)
  - ME : Vide
  - WB : Vide
- **Cycle 4 :**
  - IF : Instruction lue 0xFFF00393 (ADDI). Prochaine instruction à lire : 0x10.
  - ID : Instruction 0x7FF06313 (ORI). Adresse : 0x08.
  - EX : Adresse : 0x04. ALU-OPCODE : 0 (ADD)
  - ME : Résultat ALU : 0

  - WB : Vide
- **Cycle 5 :**
  - IF : Instruction lue 0x00204E13 (XORI). Prochaine instruction à lire : 0x14.
  - ID : Instruction 0xFFF00393 (ADDI). Adresse : 0x0C.
  - EX : Adresse : 0x08. ALU-OPCODE : 5 (OR)
  - ME : Résultat ALU : 0xFFFFF000
  - WB : Valeur : 0. Registre : 9

Les instructions continuent à se propager dans le pipeline jusqu'au cycle 12, où la dernière instruction est terminée. Le bon fonctionnement du processeur est confirmé par l'observation de l'étage Write-Back, où les valeurs correctes sont écrites dans les registres appropriés.

3.2.2 Instruction de branchement

Instructions de branchement testées

```
/*****************************************************************
Branchements: x8 = BRANCH\_SUCCESS si succes, BRANCH\_FAILURE (ou 0) sinon
 *****************************************************************/
branch:
    beq a0, a1, branch_ok   // addr=88, instr=00B50663
    li  s0, BRANCH_FAILURE  // addr=8C, instr=00100413
    jal zero, next          // addr=90, instr=0140006F

branch_ok:
    jal ra, success         // addr=94, instr=008000EF
    jal zero, next          // addr=98, instr=00C0006F

success:
    li s0, BRANCH_SUCCESS   // addr=9C, instr=FFF00413
    jalr zero, ra, 0        // addr=A0, instr=00008067
```

La figure 2 présente les résultats de la simulation du code ci-dessus. Les lignes roses délimitent les différents étages du pipeline, tandis que les lignes rouges séparent les différentes instructions. Les numéros en bleus serviront à identifier des cycles dans l'explication. Seuls les signaux pertinents pour démontrer le fonctionnement du processeur sont affichés. Notons qu'on représente les signaux sortant du Core donc les adresses réellement lues depuis la mémoire sont divisés par 4.

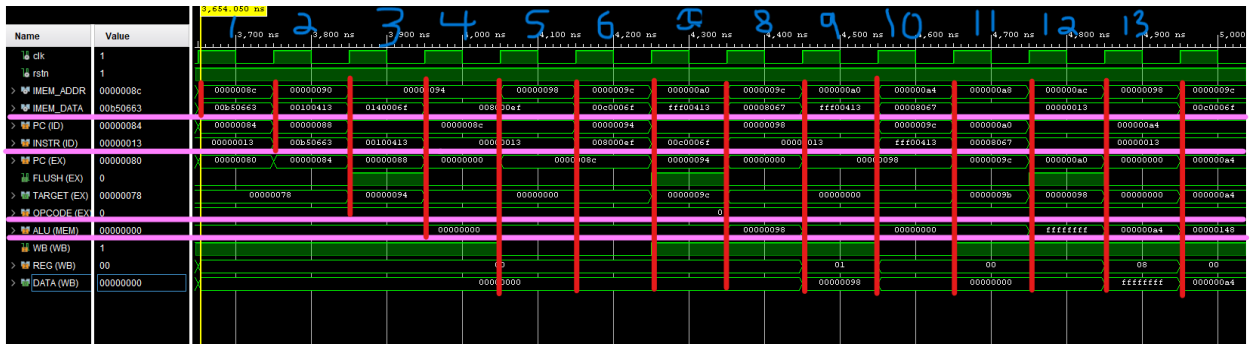

FIGURE 2 – Simulation d'instructions de branchement

Nous montrons la gestions des branchements à travers les différents cycles pertinents.

- **Cycle 1 :** La première instruction de branchement est récupérée.
- **Cycle 3 :** Le branchement est pris, et les résultats des opérations précédentes sont flushés. La target du branchement est à l'adresse `0x94` (JAL).
- **Cycles 4-5 :** Les instructions précédentes ont bien été flushées, et des `NOP` sont envoyées.
- **Cycle 6 :** L'instruction à l'étage ID est bien le `JAL` à l'adresse `0x94`.
- **Cycle 7 :** Le branchement est à nouveau pris, les résultats précédents sont flushés, et la target est à l'adresse `0x9C` (LI).
- **Cycles 8-9 :** Les instructions précédentes ont été correctement flushées, et des `NOP` sont envoyées.
- **Cycle 10 :** L'instruction à l'étage ID est bien le `LI` à l'adresse `0x9C`
- **Cycle 13 :** Le résultat du branchement réussi (`0xFFFFFFFF`) est correctement écrit dans le registre `x8`.

3.2.3 Instruction de mémoire

Instructions de mémoires testées

```
1    la   sp, _STACK     // addr=64, instr=7FC00113, x2  = 0x000007FC
2    la   gp, _HEAP      // addr=68, instr=3FC00193, x3  = 0x000003FC
3    la   tp, _HEAP      // addr=6C, instr=3FC00213, x4  = 0x000003FC
4    sw   a0, -4(sp)     // addr=70, instr=FEA12E23, DMEM[7F8] = 0x0FFF80F0
5    lw   a1, -4(sp)     // addr=74, instr=FFC12583, x11 = 0x0FFF80F0
```

La figure 1 présente les résultats de la simulation du code ci-dessus. Les lignes roses délimitent les différents étages du pipeline, tandis que les lignes rouges séparent les différentes instructions. Les numéros en bleus serviront à identifier des cycles dans l'explication. Seuls les signaux pertinents pour démontrer le fonctionnement du processeur sont affichés. Notons qu'on représente les signaux sortant du Core donc les adresses réellement lues depuis la mémoire sont divisés par 4.

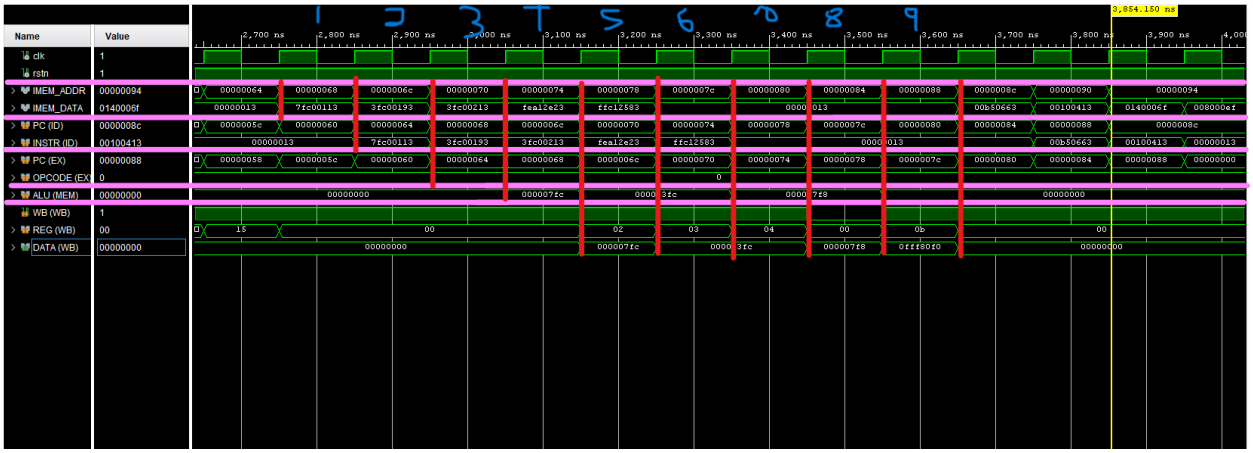

FIGURE 3 – Simulation d'instructions de mémoire

Nous montrons que nous sommes en mesures d'écrire dans la mémoire (SW) puis lire cette valeur (LW).

- **Cycles 1-4 :** Les différentes instructions entrent dans le pipeline.
- **Cycle 5 :** La valeur `0x000007FC` est correctement écrite dans le registre 2 (`LA`).
- **Cycle 6 :** La valeur `0x000003FC` est correctement écrite dans le registre 3 (`LA`).
- **Cycle 7 :** La valeur `0x000003FC` est correctement écrite dans le registre 4 (`LA`) à l'étage `WB`, et la valeur `0x0FFF80F0` est écrite dans la mémoire à l'adresse `0x7F8` (`SW`) à l'étage `MA`.
- **Cycle 8 :** Aucun registre n'est écrit à l'étage `WB` (`SW`), mais la valeur `0x0FFF80F0` est lue dans la mémoire à l'adresse `0x7F8` (`LW`) à l'étage `MA`.
- **Cycle 9 :** La valeur `0x0FFF80F0` est correctement écrite dans le registre 11 (`LW`).

3.2.4 Instruction spécialisée

Nous allons maintenant tester le fonctionnement de l'instruction spécialisée. Pour ce faire, nous générerons une instruction spécialisée et la suivrons à travers le pipeline. L'instruction sera définie comme suit :

- **IMM** : `0b101010101010` → Étendue : `0xFFFFFAAA`
- **RS1** : `0b00000`
- **funct3** : `0b101` (conversion de big-endian vers little-endian et valeur immédiate)
- **RD** : `0b00100`
- **Opcode** : `0b1010101`

Ainsi, l'instruction complète sera `AAA0D255`. Cette instruction effectue la conversion de la valeur immédiate de `big-endian` vers `little-endian`. Le résultat attendu est l'écriture de la valeur `AAFAFFFF` dans le registre 4.

La figure 4 présente les résultats de la simulation de l'instruction si dessus. Les lignes roses délimitent les différents étages du pipeline, tandis que les lignes rouges séparent les différentes instructions. Les numéros en bleus serviront à identifier des cycles dans l'explication. Seuls les signaux pertinents pour démontrer le fonctionnement du processeur sont affichés. Notons qu'on représente les signaux sortant du Core donc les adresses réellement lues depuis la mémoire sont divisés par 4.

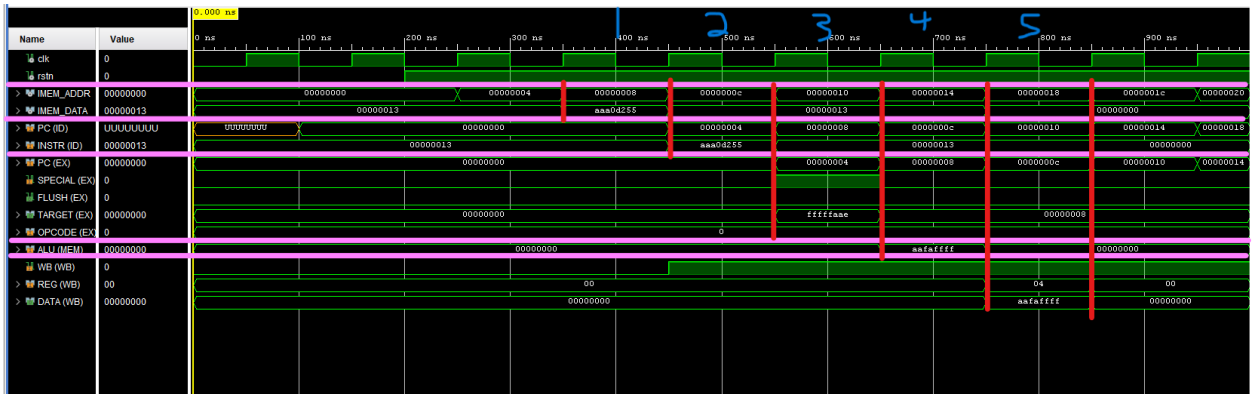

FIGURE 4 – Simulation de l'instruction spécialisée

L'instruction est suivie dans le Pipeline :

- **Cycle 1 :** L'instruction est récupérée depuis la mémoire à l'étage **IF**.
- **Cycle 2 :** L'instruction est transmise à l'étage **ID**.
- **Cycle 3 :** L'instruction spécialisée a été correctement reconnue depuis l'étage **IF**, comme le témoigne le flag `special` reçu à l'étage **EX**.
- **Cycle 4 :** La conversion a bien été effectuée, comme le montre le résultat reçu à l'étage **MEM**.
- **Cycle 5 :** La valeur convertie (`AAFAFFFF`) est correctement écrite dans le registre 4 à l'étage **WB**.

3.2.5 Conflit de données

Dans cette simulation, nous allons présenter la gestion des conflits de données.Dans notre implémentation du mini processeur RISC-V, un conflit de données survient lorsqu'une instruction tente de lire une donnée écrite par l'une des deux instructions précédentes. En effet, dans ce cas, la donnée n'a pas encore été écrite au moment de l'étage Decode, ce qui peut rendre les valeurs des registres transmises entre les étages Decode et Execute obsolètes. Pour résoudre ce problème, nous utilisons une technique appelée forwarding. Elle consiste à transmettre la valeur devant être écrite aux étages suivants en contournant l'étage Decode. Concrètement, depuis les étages Memory Access et Write Back, nous transmettons à l'étage Execute l'adresse de la donnée à écrire, la valeur de cette donnée, ainsi qu'une indication précisant si la valeur sera effectivement écrite. Il est important de noter que la valeur provenant de l'étage Memory Access est prioritaire, car elle résulte d'une instruction plus récente. Ainsi, à l'étage Execute, nous comparons l'adresse des registres source (registre 1 et registre

2) avec les adresses transmises depuis les étages suivantes. Si une correspondance est trouvée et que la donnée doit être écrite, nous utilisons cette valeur (en priorité Memory Acces) au lieu de celle transmise par l'étage Decode. Cette approche permet de corriger la majorité des conflits de données, à une exception près : le cas où la donnée à écrire provient d'une lecture en mémoire (instruction LW). Dans ce scénario, la donnée n'est pas immédiatement disponible. Si cette donnée est requise à l'étage Execute, nous introduisons un stall pour attendre qu'elle soit prête. Pour gérer ce cas particulier, l'étage Memory transmet, en plus de l'adresse, de la donnée et de l'indication de son écriture, une information supplémentaire précisant si la valeur provient d'une lecture mémoire.

Nous allons maintenant simuler les instructions suivantes.

Instructions avec conflits de données

```
1    addi sp, sp, 1                  // x2  = 0x00000001
2    sw   sp, 0(sp)                  // Conflit sans stall
3    lw   a0, 0(sp)                  // Conflit sans stall
4    lw   a1, 0(a0)                  // Conflit avec stall
```

Nous écrivons la valeur 1 dans le registre x2. L'instruction suivante (SW) utilise la valeur de x2, ce qui provoque un conflit de données. Dans ce cas, le forwarding entre les étages Memory Access et Execute suffit à résoudre ce conflit. De la même manière, l'instruction suivante (LW) lit et écrit dans le registre a0. Une fois encore, le forwarding entre les étages Write Back et Execute permet de résoudre ce conflit. Enfin, la dernière instruction lit la valeur de a0. Cependant, cette donnée provient d'une lecture mémoire, et n'est donc pas immédiatement disponible. Le processeur insère un stall pour attendre que la donnée soit prête, puis lit la valeur depuis l'étage Write Back lors du cycle suivant. La simulation présentée à la figure 5 montre le résultat de l'execution de ces instructions :

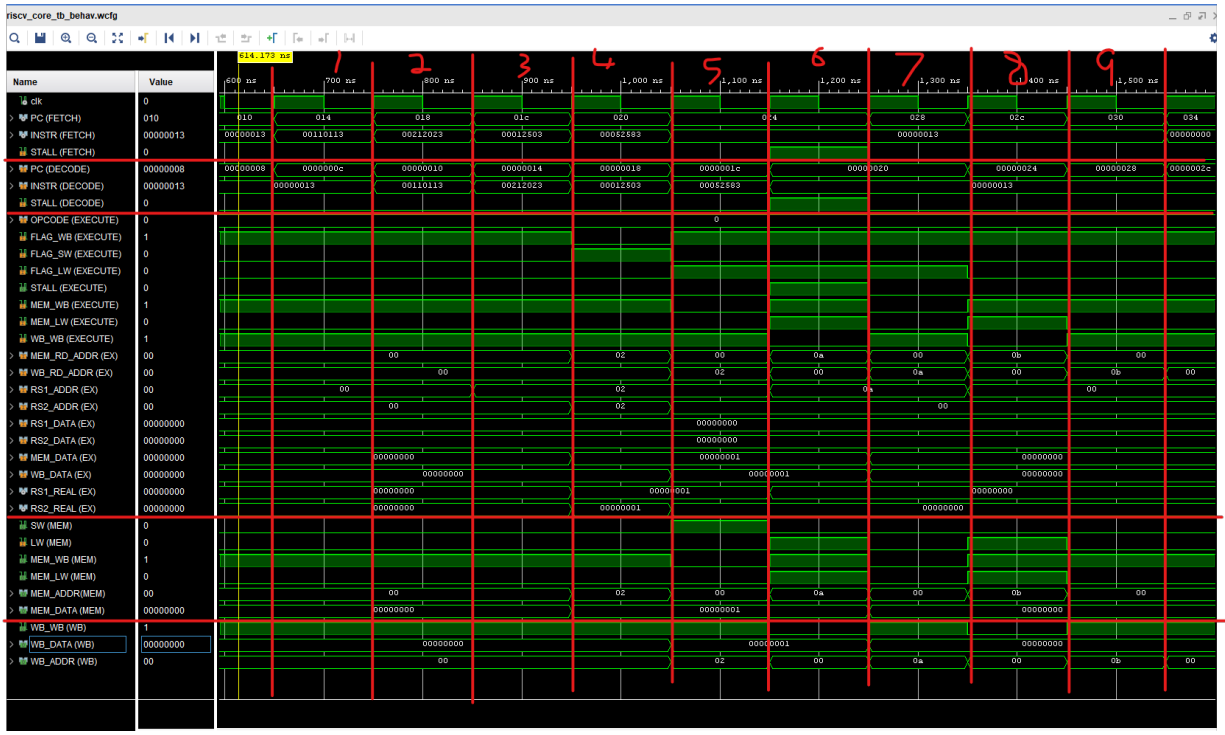

FIGURE 5 – Simulation des conflits de données

Voici une explication de ce qu'il se passe à chaque cycle :

— **Cycle 1 à 3 :** Les trois premières instructions sont dans le pipeline, et l'instruction `ADD` est exécutée à l'étape `Execute`.

— **Cycle 4 :** L'instruction `SW` est exécutée. On observe que l'étage `Memory Access` *forward* la valeur du registre `x2` ainsi que son adresse via les signaux `MEM_RD_ADDR` et `MEM_DATA`. Le signal indiquant que la donnée doit être écrite (`MEM_WB`) est également actif. La valeur utilisée par l'étage `Execute` est bien celle issue du *forwarding*, comme le montrent les signaux `RS1_REAL` et `RS2_REAL`.

— **Cycle 5 :** La première instruction `LW` est exécutée. L'étage `Memory Access` n'intervient pas dans le *forwarding* (`MEM_WB` est à `low`), car `SW` n'écrit pas de valeur. Cependant, l'étage `Write-Back` (issu de l'instruction `ADD`) *forward* bien la valeur de `x2` ainsi que son adresse via les signaux `WB_RD_ADDR` et `WB_DATA`. La valeur utilisée par l'étage `Execute` est bien celle issue du *forwarding*, comme le montre le signal `RS1_REAL`.

— **Cycle 6 :** La deuxième instruction `LW` est exécutée. On observe que le conflit est détecté et qu'un *stall* est activé. Ce *stall* se propage correctement aux étages `Fetch` et `Decode`.

— **Cycle 7 :** Le CPU est en *stall*, et les instructions précédentes sont maintenues. La valeur issue du précédent `load` est finalement *forwardée* depuis l'étage `Write-Back` (premier `LW`) grâce aux signaux `WB_RD_ADDR` et `WB_DATA`, tandis que le signal `WB_WB` est actif. La valeur

utilisée par l'étage `Execute` est bien celle issue du *forwarding*, comme le montre le signal `RS1_REAL`.

— **Cycle 8-9 :** La dernière instruction `LW` passe successivement par les étages `Memory Access` et `Write Back`.

4 Implémentation

Cette section porte sur l'implémentation du processeur. Nous commencerons par aborder la synthèse, puis le placement-routage. Pour chaque étape, nous détaillerons les scripts utilisés et expliquerons le rôle des principales commandes. Ensuite, nous présenterons les résultats des différents rapports : STA (Static Timing Analysis), DRC (Design Rule Check) et LVS (Layout Versus Schematic). Enfin, nous conclurons par une simulation temporelle de la Netlist, illustrant les délais introduits à chaque étape.

4.1 Synthèse

4.1.1 Scripts

Lors de la synthèse, nous exécutons le script suivant qui lui-même exécute différent script, chacun portant sur un élément de la synthèse. Nous détaillerons chaque sous-script :

```
source ../scripts/synthesis/setup.tcl
source ../scripts/synthesis/compile.tcl
source ../scripts/synthesis/constraints.tcl
source ../scripts/synthesis/synth.tcl
source ../scripts/synthesis/netlist.tcl
```

synthesis.tcl

Le premier script permet d'initialiser l'outil à notre configuration. On précise la version de VHDL, le niveau de verbosité ainsi que les différents emplacements et configurations de la librairie utilisée.

```
#Setup
set_db information_level 9
set_db hdl_vhdl_read_version 2008
set_db init_hdl_search_path $::env(SRC_DIR)
set_db init_lib_search_path [list $::env(FE_TIM_LIB) $::env(BE_QRC_LIB)
    $::env(BE_LEF_LIB)]
read_libs -max_libs slow_vdd1v0_basicCells.lib -min_libs
   fast_vdd1v0_basicCells.lib
read_physical -lef gsclib045_tech.lef
read_qrc gpdk045.tch
set_db interconnect_mode ple
```

setup.tcl

Le deuxième script permet de compiler les différents fichiers dans le bon ordre de compilation. Il vérifie finalement qu'aucun élément du sytème n'est manquant.

```
# Package
read_hdl -vhdl riscv_pkg.vhd

#Modules
read_hdl -vhdl riscv_adder.vhd
read_hdl -vhdl riscv_rf.vhd
read_hdl -vhdl riscv_pc.vhd
read_hdl -vhdl riscv_alu.vhd

#Pipeline
read_hdl -vhdl riscv_instruction_fetch.vhd
read_hdl -vhdl decode.vhd
read_hdl -vhdl riscv_instruction_decode.vhd
read_hdl -vhdl riscv_execute.vhd
read_hdl -vhdl riscv_memory.vhd
read_hdl -vhdl riscv_write_back.vhd

#Core
read_hdl -vhdl riscv_core.vhd

#Elaborate
elaborate riscv_core

#Checking
check_design -unresolved
```

compile.tcl

Le prochain script permet la spécification des contraintes. La seule contrainte changée par rapport au tutoriel a été la fréquence d'horloge. Elle a été réduite à 50MHz (elle aurait pue être plus élevée) pour satisfaire les contraintes de timming. On applique ainsi les contraintes et on s'assure de leur complétude.

```
read_sdc $::env(CONST_DIR)/timing.sdc
report_timing -lint > $::env(SYN_REP_DIR)/riscv_core.timing_lint.rpt
report_clocks > $::env(SYN_REP_DIR)/riscv_core.clk.rpt
report_clocks -generated > $::env(SYN_REP_DIR)/riscv_core.clk.rpt
```

constraints.tcl

Le script suivant réalise la synthèse du processeur. Il réalise la synthèse générique, l'association aux cellules standards et l'optimisation.

```
# Synthese generique
set_db syn_generic_effort high
```

```
syn_generic riscv_core
write_hdl > $::env(SYN_NET_DIR)/riscv_core.syn_gen.v

#Cellules standards
set_db syn_map_effort high
syn_map riscv_core
write_hdl > $::env(SYN_NET_DIR)/riscv_core.syn_map.v

#Optimisation du systeme
set_db syn_opt_effort high
syn_opt riscv_core
write_hdl > $::env(SYN_NET_DIR)/riscv_core.syn_opt.v
```

synth.tcl

Finalement, le dernier script produit la Netlist ainsi que le rapport STA.

```
#Netlist
write_hdl > $::env(SYN_NET_DIR)/riscv_core.syn.v
write_sdf -nonegchecks -setuphold split -version 2.1 > $::env(
    SYN_NET_DIR)/riscv_core.syn.sdf
write_sdc > $::env(CONST_DIR)/riscv_core.syn.sdc

#STA
report_timing > $::env(SYN_REP_DIR)/riscv_core.syn.timing.rpt
```

netlist.tcl

4.1.2 Contraintes et simulation

Nous nous assurons d'abord du respect des contraintes de timings à l'aide de l'outil STA. Le résultat du rapport est présenté si dessous. À noter que nous somme conservateur sur le slack time et qu'il aurait été possible d'augmenter la fréquence d'horloge.

```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 21.10-p002_1
  Generated on:           Dec 03 2024  12:00:46 am
  Module:                 riscv_core
  Operating conditions:   PVT_0P9V_125C
  Interconnect mode:      global
  Area mode:              timing library
============================================================


Path 1: MET (5391 ps) Setup Check with Pin DECODE_r_rd_addr_reg[0]/CK->D
          Group: clk
     Startpoint: (R) DECODE_r_rs2_addr_reg[0]/CK
          Clock: (R) clk
       Endpoint: (F) DECODE_r_rd_addr_reg[0]/D
          Clock: (R) clk
```

```
                  Capture       Launch
     Clock Edge:+   20000            0
    Src Latency:+       0            0
    Net Latency:+       0 (I)        0 (I)
        Arrival:=   20000            0

          Setup:-      80
    Uncertainty:-     100
  Required Time:=   19820
   Launch Clock:-       0
      Data Path:-   14428
          Slack:=    5391
```

STA post synthèse

Pour lancer la simulation temporelle post-synthèse, nous exécutons le script suivant, qui crée la librairie de travail, compile la netlist, la mémoire et le banc d'essai et lance la simulation avec le fichier SDF et la librairie des cellules standards.

```
#Setup
vmap -c
vmap gsclib045 /CMC/kits/GPDK45/simlib/gsclib045_slow
vlib syn/work
vmap work syn/work

#Net List
vlog -work work ../implementation/syn/base_netlist/riscv_core.syn.v

#Compilation
vcom -2008 -work work ../sources/dpm.vhd
vcom -2008 -work work ../sources/riscv_core_tb.vhd

#Simulation

vsim -t ps -sdfmax dut=../implementation/syn/base_netlist/
   riscv_core.syn.sdf -L gsclib045 work.riscv_core_tb
```

synth_sim.tcl

La figure 6 montre le délai introduit après la synthèse.

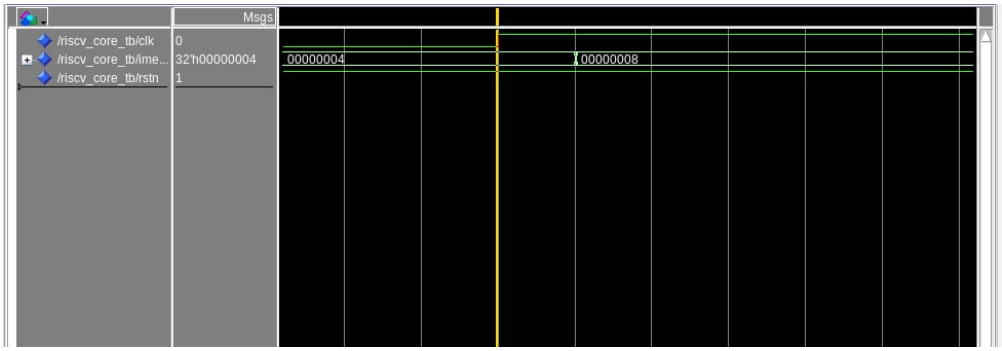

FIGURE 6 – Simulation temporelle post-synthèse

On remarque un délai de 100ps entre le front montant de l'horloge et la réaction de l'étage Fetch. Nous respectons cependant toujours le timings comme l'a montrée le rapport STA. De plus, nous nous sommes assuré du bon fonctionnement du processeur post-synthèse en nous assurant que le processeur procédait aux mêmes accès mémoire (instruction et donnée) que dans la simulation comportementale.

4.2 Placement et routage

4.2.1 Scripts

Lors du placement-routage, nous exécutons le script suivant qui lui-même exécute différent script, chacun portant sur un élément du placement routage. Nous détaillerons chaque sous-script :

```
source ../scripts/routage/setup.tcl
source ../scripts/routage/partitionnement.tcl
source ../scripts/routage/pin.tcl
source ../scripts/routage/placement.tcl
source ../scripts/routage/horloge.tcl
source ../scripts/routage/routage.tcl
source ../scripts/routage/repport.tcl
source ../scripts/routage/netlist.tcl
```

placement_routage.tcl

Le premier script permet d'initialiser l'outil à notre configuration. On initialise les librairies, la Netlist et le top level, les alimentations puis le fichier mmmc (précisant les conditions d'opération du système). Nous n'avons pas modifié les configurations présentées dans le tutoriel. On charge ensuite toutes les configurations dans Innovus.

```
#Initialisation des librairies
set init_oa_ref_lib [list gsclib045_tech gsclib045 gpdk045 giolib045]

#Initialisation de la netlist et du top-level
set init_verilog $::env(SYN_NET_DIR)/riscv_core.syn.v
```

```
set init_design_settop 1
set init_top_cell riscv_core

#Initialisation de l alimentation
set init_pwr_net VDD
set init_gnd_net VSS

#MMMC
set init_mmmc_file $::env(CONST_DIR)/mmmc.tcl

#Load
init_design
```

setup.tcl

Le deuxième script réalise le partitionnement du circuit, c'est à dire de définir la taille du circuit, l'emplacement des différents modules constituant le système, organiser l'alimentation du circuit et les pins d'entrées/sorties (script suivant). On définit le floorplan et ensuite les allimentations.

```
#Floorplan

floorPlan -site CoreSite -r 0.9 0.6 1 1 1 1

#Alimentation
globalNetConnect VDD -type pgpin -pin VDD -inst * -override
globalNetConnect VSS -type pgpin -pin VSS -inst * -override
globalNetConnect VDD -type tiehi -inst * -override
globalNetConnect VSS -type tielo -inst * -override

addStripe -nets VDD -layer Metal1 -direction vertical -width 0.6 \
    -number_of_sets 1 -start_from left -start_offset -0.8

addStripe -nets VSS -layer Metal1 -direction vertical -width 0.6 \
    -number_of_sets 1 -start_from right -start_offset -0.8


sroute -nets { VDD VSS } -connect { corePin floatingStripe }
```

partitionnement.tcl

Le script suivant permet de définir les entrée sorties. On place l'horloge et le reset sur le côté gauche. Les siganux de la mémorie d'instruction sur le côté du haut. Les signaux de la mémoire de donnée du côté droit et finalement les pins du DFT en bas (bien que non utilisé).

```
setPinAssignMode -pinEditInBatch true

# Assign reset and clock pins to the top edge (edge 0)
```

```
editPin -pin [list i_rstn i_clk] -edge 0 -layer 4 -spreadType SIDE
    -offsetEnd 2 -offsetStart 2 -spreadDirection clockwise -pinWidth 0
    .08 -pinDepth 0.335 -fixOverlap 1

# Assign instruction memory enable to the right edge (edge 1)
editPin -pin [list o_imem_en] -edge 1 -layer 4 -spreadType SIDE
    -offsetEnd 2 -offsetStart 2 -spreadDirection clockwise -pinWidth 0
    .08 -pinDepth 0.335 -fixOverlap 1

# Assign instruction memory address (9 bits) to the right edge (edge 1)
set imem_addr_pins {}
for {set i 0} {$i <= 8} {incr i} {
    lappend imem_addr_pins "o_imem_addr[$i]"
}
editPin -pin $imem_addr_pins -edge 1 -layer 4 -spreadType SIDE
    -offsetEnd 2 -offsetStart 2 -spreadDirection clockwise -pinWidth 0
    .08 -pinDepth 0.335 -fixOverlap 1

# Assign data memory enable and write enable to the right edge (edge 1)
editPin -pin [list o_dmem_en o_dmem_we] -edge 1 -layer 4 -spreadType
    SIDE -offsetEnd 2 -offsetStart 2 -spreadDirection counterclockwise
    -pinWidth 0.08 -pinDepth 0.335 -fixOverlap 1

# Assign data memory address (9 bits) to the right edge (edge 2, more
    space for data-related signals)
set dmem_addr_pins {}
for {set i 0} {$i <= 8} {incr i} {
    lappend dmem_addr_pins "o_dmem_addr[$i]"
}
editPin -pin $dmem_addr_pins -edge 2 -layer 4 -spreadType SIDE
    -offsetEnd 2 -offsetStart 2 -spreadDirection counterclockwise
    -pinWidth 0.08 -pinDepth 0.335 -fixOverlap 1

# Assign instruction memory read data (32 bits) to the right edge (edge
    2)
set imem_read_pins {}
for {set i 0} {$i <= 31} {incr i} {
    lappend imem_read_pins "i_imem_read[$i]"
}
editPin -pin $imem_read_pins -edge 2 -layer 4 -spreadType SIDE
    -offsetEnd 2 -offsetStart 2 -spreadDirection counterclockwise
    -pinWidth 0.08 -pinDepth 0.335 -fixOverlap 1

# Assign data memory read data (32 bits) to the right edge (edge 2)
set dmem_read_pins {}
for {set i 0} {$i <= 31} {incr i} {
    lappend dmem_read_pins "i_dmem_read[$i]"
}
```

```
editPin -pin $dmem_read_pins -edge 2 -layer 4 -spreadType SIDE
    -offsetEnd 2 -offsetStart 2 -spreadDirection counterclockwise
    -pinWidth 0.08 -pinDepth 0.335 -fixOverlap 1

# Assign data memory write data (32 bits) to the right edge (edge 2)
set dmem_write_pins {}
for {set i 0} {$i <= 31} {incr i} {
    lappend dmem_write_pins "o_dmem_write[$i]"
}
editPin -pin $dmem_write_pins -edge 2 -layer 4 -spreadType SIDE
    -offsetEnd 2 -offsetStart 2 -spreadDirection counterclockwise
    -pinWidth 0.08 -pinDepth 0.335 -fixOverlap 1

# Assign DFT pins (scan enable, test mode, TDI, TDO) to the left edge (
    edge 3)
editPin -pin [list i_scan_en i_test_mode i_tdi o_tdo] -edge 3 -layer 3
    -spreadType SIDE -offsetEnd 2 -offsetStart 2 -spreadDirection
    clockwise -pinWidth 0.08 -pinDepth 0.335 -fixOverlap 1
```

pin.tcl

Le prochain script permet de réaliser le placement. Il vérifie d'abord les contraintes temporelles, ouis il définit les options pour le placement. On retire les bascules à balayages (nous utilisons la Netlist de base sans chaîne de balayage). On lance ensuite le placement.

```
#Verification contraintes temporelle
timeDesign -prePlace -outDir $::env(PNR_REP_DIR)/timing

#Option de placement
setDesignMode -process 45 -flowEffort standard
setPlaceMode -timingDriven true  \
                        -place_global_cong_effort auto \
                        -place_global_reorder_scan true



#Sans balayage
setPlaceMode -place_global_reorder_scan false
deleteAllScanCells


#Lancement du placement
place_opt_design
```

placement.tcl

Le prochain script réalise la synthèse de l'arbre d'horloge. Il définit les cellules à utiliser pour l'arbre d'horloge. Il réalise ensuite la synthèse de l'arbre d'horloge ainsi que l'optimisation. Il vérifie ensuite les contraintes temporelles.

```
#Definition des cellules
set_ccopt_property buffer_cells [list CLKBUFX20 CLKBUFX16 CLKBUFX12
   CLKBUFX8 CLKBUFX6 CLKBUFX4 CLKBUFX3 CLKBUFX2]
set_ccopt_property inverter_cells [list CLKINVX20 CLKINVX6 CLKINVX8
   CLKINVX16 CLKINVX12 CLKINVX4 CLKINVX3 CLKINVX2 CLKINVX1]
set_ccopt_property use_inverters true
set_ccopt_property clock_gating_cells TLATNTSCA*

#Synthese de L'arbre
ccopt_design

#Optimisation
optDesign -postCTS

#Rapport
timeDesign -postCTS -outDir $::env(PNR_REP_DIR)/timing
timeDesign -hold -postCTS -outDir $::env(PNR_REP_DIR)/timing
```

horloge.tcl

Le script suivant réalise le routage. Il rempli les espaces vides, configure le routage et le lance. Il réalise ensuite des optimisations.

```
#Adding fillers
addFiller -cell FILL32 FILL16 FILL8 FILL4 FILL2 FILL1 -prefix FILLER

#Configuration temporelle/electrique
setNanoRouteMode -quiet -routeWithTimingDriven true
setNanoRouteMode -routeWithSIDriven true

#Routage
routeDesign -globalDetail


#Optimisation
setExtractRCMode -engine postRoute
extractRC

setAnalysisMode -analysisType onChipVariation
setAnalysisMode -cppr both

optDesign -postRoute -setup -hold -outDir $::env(PNR_DIR)/opt
```

routage.tcl

Le script suivant génère les rapports DRC,LVS et STA.

```
#DRC
```

```
set_verify_drc_mode -report $::env(PNR_REP_DIR)/riscv_core.drc.rpt
verify_drc

#LVS
verifyConnectivity -type all -report $::env(PNR_REP_DIR)/
   riscv_core.con.rpt

#STA
timeDesign -postRoute -outDir $::env(PNR_REP_DIR)/timing
report_timing > $::env(PNR_REP_DIR)/riscv_core.tim.rpt
```

repport.tcl

Finalement, le dernier script génère la Netlist.

```
saveNetlist $::env(PNR_NET_DIR)/riscv_core.pnr.v
write_sdf -version 2.1 -target_application verilog -interconn noport
   $::env(PNR_NET_DIR)/riscv_core.pnr.sdf
```

netlist.tcl

La figure 7 donne le résultat du placement routage.

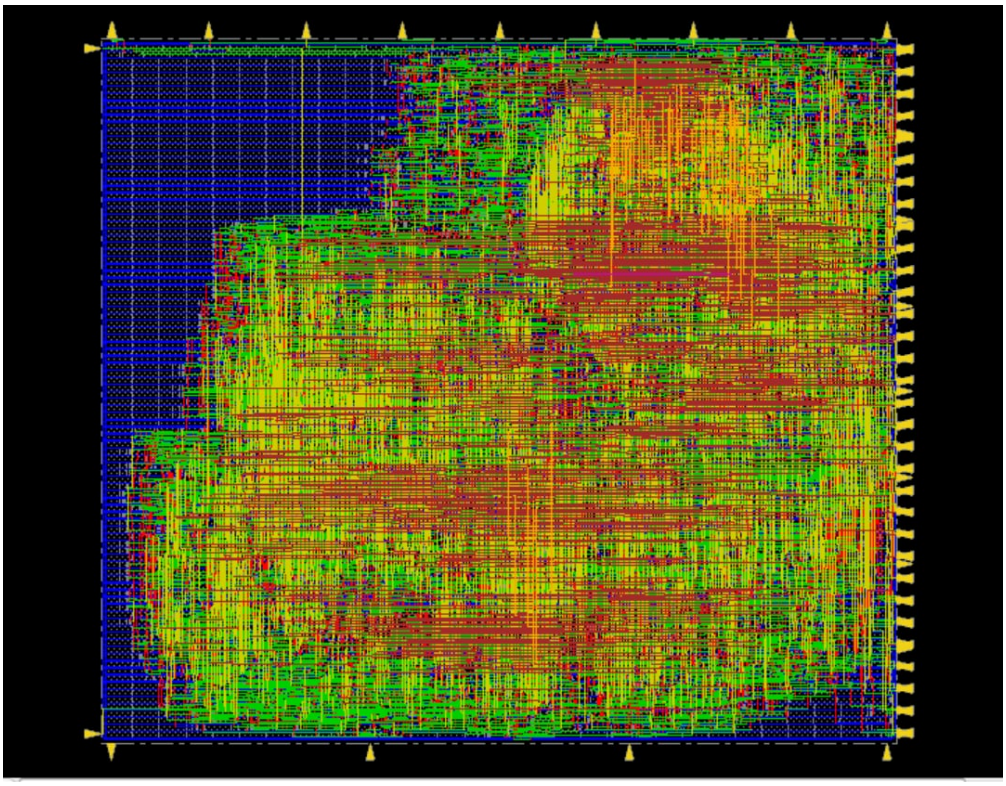

FIGURE 7 – Résultat du placement routage

4.2.2 Contraintes et simulation

Nous nous assurons d'abord du respect des contraintes de timings à l'aide de l'outil STA. Le résultat du rapport est présenté si dessous.

```
#####################################################################
#  Generated by:      Cadence Innovus 21.17-s075_1
#  OS:                Linux x86_64(Host ID vlsi401)
#  Generated on:      Mon Dec  2 23:17:07 2024
#  Design:            riscv_core
#  Command:           report_timing > $::env(PNR_REP_DIR)/riscv_core.tim.rpt
#####################################################################
Path 1: MET Setup Check with Pin DECODE_r_rd_addr_reg[0]/CK
Endpoint:   DECODE_r_rd_addr_reg[0]/D (v) checked with  leading edge of 'clk'
Beginpoint: MEMORY_o_re_wb_reg/Q      (^) triggered by  leading edge of 'clk'
Path Groups: {clk}
Analysis View: slow_av
Other End Arrival Time          0.138
- Setup                         0.030
+ Phase Shift                  20.000
+ CPPR Adjustment               0.000
- Uncertainty                   0.100
= Required Time                20.007
- Arrival Time                 11.351
= Slack Time                    8.657
     Clock Rise Edge                     0.000
     + Clock Network Latency (Prop)      0.002
     = Beginpoint Arrival Time           0.002
```

STA post placement-routage

Nous vérifions ensuite le résultat du DRC.

```
#####################################################################
#  Generated by:      Cadence Innovus 21.17-s075_1
#  OS:                Linux x86_64(Host ID vlsi401)
#  Generated on:      Mon Dec  2 23:17:03 2024
#  Design:            riscv_core
#  Command:           verify_drc
#####################################################################
#set_verify_drc_mode -report /users/Cours/ele8304/20/Labs/VLSI_lab2-vf/
    implementation/pnr/base_reports/riscv_core.drc.rpt

No DRC violations were found
```

DRC post placement-routage

Finalement, nous avons deux violations pour LVS. Ces violations correspondent à deux ground flottants. Ces violations sont tolérées dans le cadre du cours.

```
#####################################################################
#  Generated by:      Cadence Innovus 21.17-s075_1
#  OS:                Linux x86_64(Host ID vlsi413)
```

```
#  Generated on:      Mon Dec  2 19:33:54 2024
#  Design:            riscv_core
#  Command:           verifyConnectivity -type all -report /export/tmp/8304_20/
    VLSI_lab2-vf/implementation/pnr/base_reports/riscv_core.con.rpt
############################################################
Verify Connectivity Report is created on Mon Dec  2 19:33:54 2024



Net VSS: dangling Wire at (191.100, 1.140) (191.100, 1.140) on layer: Metal1
Net VSS: dangling Wire at (191.100, 168.720) (191.100, 168.720) on layer: Metal1

Begin Summary
    2 Problem(s) (IMPVFC-94): The net has dangling wire(s).
    2 total info(s) created.
End Summary
```

LVS post placement-routage

Pour lancer la simulation temporelle post placement-routage, nous exécutons le script suivant, qui crée la librairie de travail, compile la Netlist, la mémoire et le banc d'essai et lance la simulation avec le fichier SDF et la librairie des cellules standards.

```
#Setup
vmap -c
vmap gsclib045 /CMC/kits/GPDK45/simlib/gsclib045_slow
vlib pvr/work
vmap work pvr/work

#Net List
vlog -work work ../implementation/pnr/base_netlist/riscv_core.pnr.v

#Compilation
vcom -2008 -work work ../sources/dpm.vhd
vcom -2008 -work work ../sources/riscv_core_tb.vhd

#Simulation

vsim -t ps -sdfmax dut=../implementation/pnr/base_netlist/
   riscv_core.pnr.sdf -L gsclib045 work.riscv_core_tb
```

synth_sim.tcl

La figure 8 montre le délai introduit après le placement-routage.

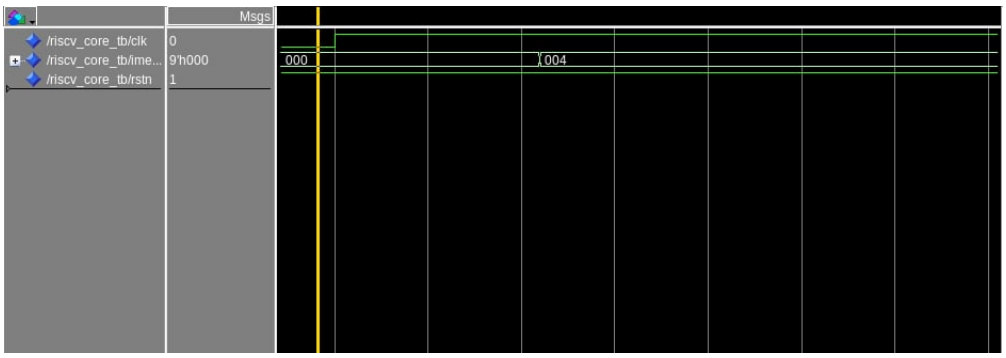

FIGURE 8 – Simulation temporelle post placement-routage

On remarque un délai de 220ps entre le front montant de l'horloge et la réaction de l'étage Fetch, un délai deux fois plus grand qu'après la synthèse. Nous respectons cependant toujours le timings comme l'a montrée le rapport STA. De plus, nous nous sommes assuré du bon fonctionnement du processeur post placement-routage en nous assurant que le processeur procédait aux mêmes accès mémoire (instruction et donnée) que dans la simulation comportementale et post-synthèse.

5 Conclusion

Ce projet a permis de concevoir et de tester un processeur basé sur l'architecture RISC-V à pipeline, en mettant en œuvre et vérifiant plusieurs modules essentiels à son fonctionnement, tels que l'Adder, l'ALU, le compteur de programme et le banc de registres. Nous avons décrit en détail le fonctionnement de chaque étage du pipeline et démontré le traitement efficace de quatre types d'instructions (arithmétiques, branchements, mémoire et spécialisées) par le cœur du processeur.Les simulations effectuées, tant au niveau des tests fonctionnels que des simulations post-synthèse et post-placement-routage, ont permis de valider le bon fonctionnement du processeur à chaque étape de son cycle d'exécution. Les résultats obtenus lors de ces simulations ont montré que le processeur fonctionne comme prévu, avec une gestion correcte des données et des instructions à travers le pipeline. La synthèse et le placement-routage ont également été réalisés avec succès, et les simulations associées ont confirmé que le design respectait les contraintes de timing et de performance. Ce projet a ainsi permis de valider à la fois la conception fonctionnelle et la faisabilité physique du processeur, tout en offrant un aperçu complet des processus de conception, de simulation et de vérification des circuits intégrés.