

## Documentation

# Microprocesseur *mini-riscv*

Professeur:	Chargé de cours:	Chargée de laboratoire:
<b>Yvon Savaria</b>	<b>Mickaël Fiorentino</b>	<b>Érika Miller-Jolicoeur</b>
<a href="mailto:yvon.savaria@polymtl.ca">yvon.savaria@polymtl.ca</a>	<a href="mailto:mickael.fiorentino@polymtl.ca">mickael.fiorentino@polymtl.ca</a>	<a href="mailto:erika.miller-jolicoeur@polymtl.ca">erika.miller-jolicoeur@polymtl.ca</a>

Automne 2020

## TABLE DES MATIÈRES

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Architecture</b>	<b>3</b>
2.1	Jeu d'instruction . . . . .	3
2.1.1	Branchements . . . . .	4
2.1.2	Accès à la mémoire de donnée . . . . .	5
2.1.3	Opérations arithmétiques & logiques . . . . .	6
2.2	Programmation . . . . .	7
2.2.1	Interfaces Mémoires . . . . .	7
2.2.2	Compilation . . . . .	8
2.2.3	Assembleur . . . . .	8
2.2.4	Registres . . . . .	8
<b>3</b>	<b>Microarchitecture</b>	<b>9</b>
3.1	Modules . . . . .	10
3.1.1	Adder . . . . .	10
3.1.2	ALU . . . . .	11
3.1.3	Compteur Programme . . . . .	12
3.1.4	Banc de Registres . . . . .	13
3.2	Pipeline . . . . .	15
3.2.1	Instruction Fetch (IF) . . . . .	15
3.2.2	Instruction Decode (ID) . . . . .	16
3.2.3	Execute (EX) . . . . .	17
3.2.4	Memory Access (ME) . . . . .	17
3.2.5	Write-Back (WB) . . . . .	17
3.3	Gestion des conflits . . . . .	18

## 1 INTRODUCTION

L'évolution des performances des microprocesseurs intégrés—depuis l'Intel 4004 en 1971—se déroule sur deux principaux terrains d'innovations :

1. Les technologies de transistor : La réduction des tailles des transistors, l'augmentation de la densité d'intégration, et la réduction de la tension d'alimentation, a permis une augmentation des fréquences d'opérations des processeurs tout en gardant une densité de puissance quasi-constante.
2. Les techniques de conceptions des microarchitectures de processeurs : Les pipeline, les caches, les prédictors de branche, les bus, *etc.* permettent de tirer le meilleur parti des technologies de semiconducteurs.

À titre d'exemple, la FIGURE 1 montre le dessin des masques du microprocesseur RISCv *PULP*, développé par l'ETH Zurich en 2015. Il a été fabriqué par *STMicroelectronics* en technologie CMOS 28 nm FD-SOI. Il contient 2.5 millions de portes logiques (2500 *kGE*—*Gate Equivalent*) sur une surface de  $2.7\text{ mm}^2$ , et consomme 1.2 mW à 50 MHz avec une tension d'alimentation proche du seuil de 0.6 V.

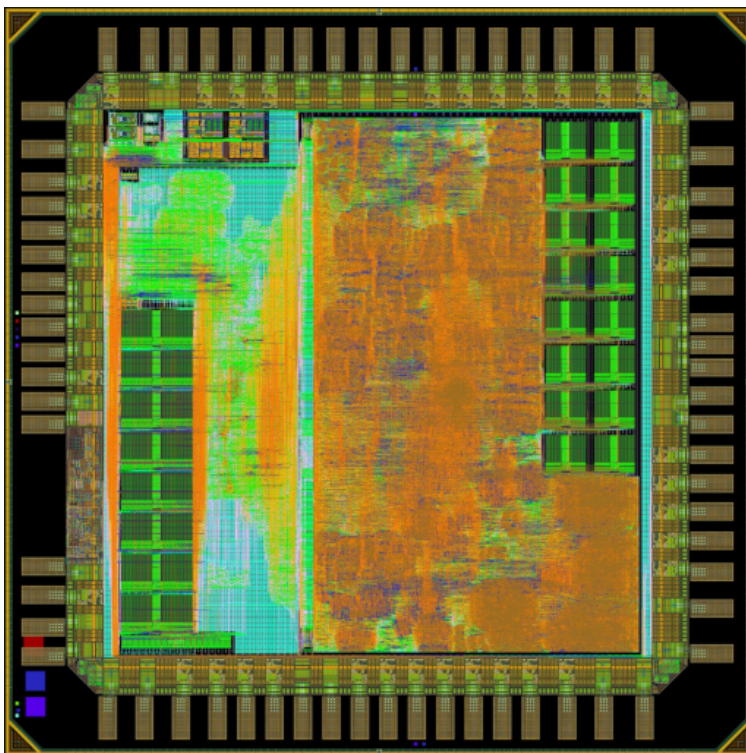


FIGURE 1: Dessin des masques du microprocesseur RISCv *PULPv3*

Ce laboratoire consiste à concevoir, et à implémenter en technologie 45 nm, un microprocesseur RISCv simple appelé *mini-riscv*. Son jeu d'instruction est dérivé de l'architecture RV32I, et sa microarchitecture utilise un pipeline à 5 étages. L'objectif de ce laboratoire est de vous permettre d'appréhender les étapes nécessaires à la conception d'un microprocesseur, de sa description matérielle en VHDL jusqu'à son implémentation en circuit intégré.

## 2 ARCHITECTURE

Cette partie présente l'architecture du processeur *mini-riscv*, c'est-à-dire l'ensemble des spécifications de haut niveau qui définissent l'interface entre le logiciel et le matériel. Nous verrons dans un premier temps le jeu d'instruction du *mini-riscv*, puis son interface avec les mémoires d'instructions et de données, et nous finirons par quelques notions sur son assembleur, et les directives de compilation.

### 2.1 JEU D'INSTRUCTION

Le jeu d'instruction (*ISA : Instruction Set Architecture*) est la spécification qui définit la liste des instructions supportées par un processeur, les formats d'instructions, ainsi que les modes d'adressages des opérandes. À partir de ces informations découlent d'une part une microarchitecture qui implémente ces spécifications, et d'autre part une façon de programmer le processeur, c'est-à-dire un langage assembleur, et un compilateur. Le jeu d'instruction du *mini-riscv* est un sous ensemble du jeux d'instruction [RV32I](#). Il s'agit d'une architecture RISC (*Reduced Instruction Set Computer*), dont les principes de base sont les suivant :

- Les instructions sont encodées avec un format fixe
- Les opérations arithmétiques et logiques sont effectuées uniquement sur des registres : Le *mini-riscv* possède un banc de registres de  $32 \times 32$  bits ( $x0 \dots x31$ ), avec  $x0=0x0$ .
- La mémoire de donnée n'est accessible qu'à partir des instructions *load* ( $lw$ ) et *store* ( $sw$ ) : Pour effectuer des opérations sur des éléments en mémoire il faut d'abord les charger (*load*) dans un registre, puis effectuer l'opération avant d'enregistrer (*store*) le résultat.

Le jeu d'instruction du *mini-riscv* utilise un format d'instruction fixe de 32 bits. On distingue 6 formats d'instructions comme le montre la [FIGURE 2](#).

	31	30	25	24	21	20	19	15	14	12	11	8	7	6	0	
<b>R-TYPE</b>	funct7					rs2		rs1		funct3		rd			opcode	
<b>I-TYPE</b>	l-imm[11:0]							rs1		funct3		rd			opcode	
<b>S-TYPE</b>	S-imm[11:5]					rs2		rs1		funct3		S-imm[4:0]			opcode	
<b>B-TYPE</b>	B-imm[12]		B-imm[10:5]			rs2		rs1		funct3		B-imm[4:1]		B-imm[11]		opcode
<b>U-TYPE</b>	U-imm[31:12]										rd			opcode		
<b>J-TYPE</b>	J-imm[20]		J-imm[10:1]			J-imm[11]		J-imm[19:12]				rd			opcode	

FIGURE 2: Formats d'instructions

Les portions `funct7`, `funct3` et `opcode` encodent la nature des instructions (`add`, `sub`, `beq`, etc.). Les portions `rs1`, `rs2` et `rd` encodent l'adresse des registres (respectivement : opérandes et destination), et les portions `*-imm[]` encodent des valeurs immédiates. On distingue 5 formats de valeurs immédiates, dont les valeurs dans les différents formats d'instructions sont étendues sur 32-bits selon l'encodage présenté à la [FIGURE 3](#) (ici `inst[]` réfère aux portions des formats d'instructions).

	31	30	20	19	12	11	10	5	4	1	0
I-IMM	inst[31]						inst[30:25]		inst[24:21]		inst[20]
S-IMM	inst[31]						inst[30:25]		inst[11:8]		inst[7]
B-IMM	inst[31]				inst[7]	inst[30:25]		inst[11:8]		0	
U-IMM	inst[31]	inst[30:20]		inst[19:12]		0					
J-IMM	inst[31]		inst[19:12]		inst[20]	inst[30:25]		inst[24:21]		0	

FIGURE 3: Formats des valeurs immédiates

La FIGURE 4 détaille les 25 instructions supportées par le *mini-riscv*, ainsi que le format d'instruction associé à chacune d'entre-elles.

U-imm[31:12]				rd	0110111	<b>LUI</b>
J-imm[20   10:1   11   19:12]				rd	1101111	<b>JAL</b>
I-imm[11:0]		rs1	000	rd	1100111	<b>JALR</b>
B-imm[12 10:5]	rs2	rs1	000	B-imm[4:1 11]	1100011	<b>BEQ</b>
I-imm[11:0]		rs1	010	rd	0000011	<b>LW</b>
S-imm[11:5]	rs2	rs1	010	S-imm[4:0]	0100011	<b>SW</b>
I-imm[11:0]		rs1	000	rd	0010011	<b>ADDI</b>
I-imm[11:0]		rs1	010	rd	0010011	<b>SLTI</b>
I-imm[11:0]		rs1	011	rd	0010011	<b>SLTIU</b>
I-imm[11:0]		rs1	100	rd	0010011	<b>XORI</b>
I-imm[11:0]		rs1	110	rd	0010011	<b>ORI</b>
I-imm[11:0]		rs1	111	rd	0010011	<b>ANDI</b>
0000000	shamt	rs1	001	rd	0010011	<b>SLLI</b>
0000000	shamt	rs1	101	rd	0010011	<b>SRLI</b>
0100000	shamt	rs1	101	rd	0010011	<b>SRAI</b>
0000000	rs2	rs1	000	rd	0110011	<b>ADD</b>
0100000	rs2	rs1	000	rd	0110011	<b>SUB</b>
0000000	rs2	rs1	001	rd	0110011	<b>SLL</b>
0000000	rs2	rs1	010	rd	0110011	<b>SLT</b>
0000000	rs2	rs1	011	rd	0110011	<b>SLTU</b>
0000000	rs2	rs1	100	rd	0110011	<b>XOR</b>
0000000	rs2	rs1	101	rd	0110011	<b>SRL</b>
0100000	rs2	rs1	101	rd	0110011	<b>SRA</b>
0000000	rs2	rs1	110	rd	0110011	<b>OR</b>
0000000	rs2	rs1	111	rd	0110011	<b>AND</b>

FIGURE 4: Liste des instructions supportées par le *mini-riscv*

### 2.1.1 BRANCHEMENTS

Les instructions de sauts inconditionnels, **JAL** (*Jump And Link*) et **JALR** (*Jump And Link Register*), sont présentées à la FIGURE 5. Elles modifient la valeur du compteur programme.

- L'instruction **JAL** utilise le format J-TYPE, où l'immédiate de 20-bits (*offset*) encode la portée du saut relativement à la valeur courante du compteur programme : L'*offset* est étendu sur 32-bits signés selon le format d'immédiate J-IMM, et est ajouté au compteur programme pour former l'adresse de destination du saut. L'adresse de l'instruction consécutive au saut ( $pc+4$ ) est sauvegardée dans le registre de destination (*dest*).

j-imm[20   10:1   11   19:12]			rd	opcode
<b>OFFSET[20:1]</b>			<b>DEST</b>	<b>JAL</b>
l-imm[11:0]	rs1	funct3	rd	opcode
<b>OFFSET[11:0]</b>	<b>BASE</b>	<b>0</b>	<b>DEST</b>	<b>JALR</b>

FIGURE 5: Instructions de sauts inconditionnels

B-imm[12 10:5]	rs2	rs1	funct3	B-imm[4:1 11]	opcode
<b>OFFSET[12,10:5]</b>	<b>SRC2</b>	<b>SRC1</b>	<b>BEQ</b>	<b>OFFSET[11,4:1]</b>	<b>BRANCH</b>

FIGURE 6: Instruction de branchement conditionnels

- L’instruction **JALR** utilise le format I-TYPE. Elle effectue un saut indépendant de la valeur du compteur programme : L’immédiat de 12-bits (*offset*) est étendu sur 32-bits signés selon le format I-IMM, et est ajouté au registre *base* pour former l’adresse de destination du saut. L’adresse de l’instruction consécutive au saut (*pc+4*) est sauvegardée dans le registre de destination (*dest*).
- L’instruction de branchement conditionnel **BEQ** (*Branch On Equal*), est présentée à la FIGURE 6. Elle utilise le format B-TYPE, où l’immédiat de 12-bits (*offset*) encode la portée du saut. Le saut est réalisé relativement à la valeur courante du compteur programme, si la condition de branchement est remplie : Le saut n’est réalisé que si les registres *src1* et *src2* sont égaux. L’*offset* est étendu sur 32-bits signés selon le format B-IMM, et est ajouté au compteur programme pour former l’adresse de destination du saut.

### 2.1.2 ACCÈS À LA MÉMOIRE DE DONNÉE

Les instructions d’accès à la mémoire de donnée, *LW* (*Load Word*) et *SW* (*Store Word*), sont présentées à la FIGURE 7. Elles opèrent un transfert entre le banc de registre et la mémoire de donnée.

l-imm[11:0]		rs1	funct3	rd	opcode
OFFSET[11:0]		BASE	LW	DEST	LOAD
S-imm[11:5]	rs2	rs1	funct3	S-imm[4:0]	opcode
OFFSET[11:5]		SRC	BASE	SW	OFFSET[4:0]
					STORE

FIGURE 7: Instructions d’accès à la mémoire de donnée

- L’instruction **LW** utilise le format I-TYPE, où l’immédiat de 12-bits (*offset*) encode l’adresse de lecture relativement à l’adresse contenue dans le registre *base* : l’*offset* est étendu sur 32-bits signés et ajouté au registre *base* pour former l’adresse de lecture. La valeur lue depuis la mémoire de donnée à cette adresse est sauvegardée dans le registre *dest*.
- L’instruction **SW** utilise le format S-TYPE, où l’immédiat de 12-bits (*offset*) encode l’adresse d’écriture relativement à l’adresse contenue dans le registre *base* : l’*offset* est étendu sur 32-bits signés et ajouté au registre *base* pour former l’adresse d’écriture. La valeur du registre *src* est écrite en mémoire à cette adresse.

### 2.1.3 OPÉRATIONS ARITHMÉTIQUES & LOGIQUES

Les opérations arithmétiques et logiques se déclinent en trois variantes : une variante utilisant le format R-TYPE, une variante utilisant le format I-TYPE, et une variante utilisant le format U-TYPE.

U-imm[31:12]	rd	opcode
<b>U-imm[31:12]</b>	<b>DEST</b>	<b>LUI</b>

FIGURE 8: Instruction LUI (U-TYPE)

I-imm[11:0]	rs1	funct3	rd	opcode
<b>I-imm[11:0]</b>	<b>SRC</b>	<b>ADDI/SLTI[U]/ ANDI/ORI/XORI</b>	<b>DEST</b>	<b>OP-IMM</b>

FIGURE 9: Instructions arithmétiques et logiques opérant sur une valeur immédiate (I-TYPE)

funct7	rs2	rs1	funct3	rd	opcode
<b>0000000</b>	<b>SRC2</b>	<b>SRC1</b>	<b>ADD/SLT[U] AND/OR/XOR</b>	<b>DEST</b>	<b>OP</b>
<b>0100000</b>	<b>SRC2</b>	<b>SRC1</b>	<b>SUB</b>	<b>DEST</b>	<b>OP</b>

FIGURE 10: Instructions arithmétiques et logiques opérant sur des registres(R-TYPE)

funct7	rs2	rs1	funct3	rd	opcode
<b>0000000</b>	<b>SHAMT[4:0]</b>	<b>SRC</b>	<b>SLL/SRL SRA</b>	<b>DEST</b>	<b>OP</b>
<b>0100000</b>	<b>SHAMT[4:0]</b>	<b>SRC</b>	<b>SLLI/SRLI SRAI</b>	<b>DEST</b>	<b>OP-IMM</b>

FIGURE 11: Instructions de décalage

- L’instruction **LUI** (*Load Upper Immediate*) est présentée à la FIGURE 8, et utilise le format d’instruction U-TYPE. Elle place les 20 premiers bits (U-imm[31:12]) de son immédiate dans les 20 bits de poids fort du registre de destination rd, et remplit le reste avec des zéros.
- Les instructions arithmétiques et logiques, opérant sur une valeur immédiate sont présentées à la FIGURE 9. Elles utilisent le format I-TYPE, et partagent le même *opcode* (OP-IMM). Les opérations **ADDI**, **ANDI**, **ORI** et **XORI** opèrent respectivement une addition, un ET, un OU, et un XOR logique entre le contenu du registre src et la valeur immédiate I-imm étendue sur 32-bits signés. Les opérations **SLTI** et **SLTIU** (*Set Less Than Immediate* et *Set Less Than Immediate Unsigned*) comparent la valeur du registre src avec la valeur immédiate I-imm étendue sur 32-bits signés ou non-signés. Si src < I-imm alors dest vaut 1, sinon dest vaut 0.
- Les instructions arithmétiques et logiques opérant sur des registres sont présentées à la FIGURE 10. Elles utilisent le format R-TYPE, et partagent le même *opcode* (OP). Les opérations **ADD**, **SUB**, **AND**, **OR** et **XOR** opèrent respectivement une addition, une soustraction, un ET, un OU, et un XOR logique entre le contenu du registre src1 et du registre src2. À noter que l’instruction SUB ne se

distingue de l'instruction ADD que par la valeur du champ `funct7`. Les opérations **SLT** et **SLTU** (*Set Less Than* et *Set Less Than Unsigned*) comparent la valeur du registre `src1` avec la valeur du registre `src2` (signés et non-signés respectivement). Si `src1 < src2` alors `dest` vaut 1, sinon `dest` vaut 0.

- Les instructions de décalage sont présentées à la FIGURE 11. On distingue trois types de décalages : décalage à gauche (**SLL** : *Shift Left Logical*), décalage à droite logique (**SRL** : *Shift Right Logical*) et décalage à droite arithmétique (**SRA** : *Shift Right Arithmetic*). Chacun d'entre eux possèdent également une variante immédiate (**SLLI**, **SRLI**, **SRAI**). L'opération consiste à décaler le contenu du registre `src` de la valeur contenue dans `shamt` (5 premiers bits du contenu du registre `rs2`, ou de la valeur immédiate `I-imm`), et de sauvegarder le résultat dans le registre `dest`. À noter que l'instruction `SRA[I]` ne se distingue de l'instruction `SRL[I]` que par la valeur du champ `funct7`.

## 2.2 PROGRAMMATION

### 2.2.1 INTERFACES MÉMOIRES

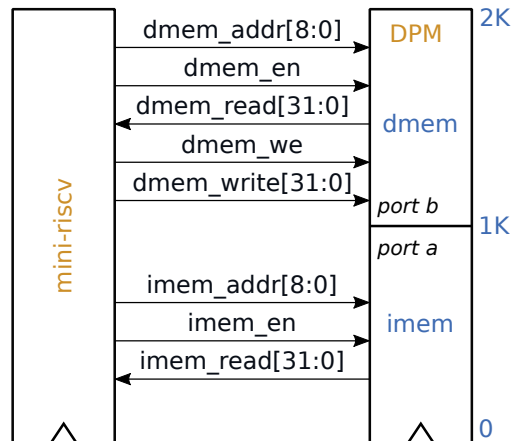


FIGURE 12: Mémoires

Le système de mémoire du `mini-riscv` (FIGURE 12) est composé d'une mémoire double-port, adressable par octet (chaque octet possède une adresse unique). Cette mémoire est séparée en deux espaces d'adresses de largeur 1kB : l'espace d'adresses de la mémoire d'instruction (`imem`) est entre 0 et 1kB sur le `port a`, et celui de la mémoire de données (`dmem`) est entre 1kB et 2kB sur le `port b`. La mémoire est une instance de l'entité `dpm` décrite dans le fichier VHDL `dpm.vhd` (ce fichier est fourni dans le dossier de travail). Elle s'initialise en début de simulation à partir d'un fichier `.hex` contenant la liste des instructions et des données du programme en format hexadécimal. En lecture et en écriture la mémoire a une latence de 1 cycle.

### 2.2.2 COMPILATION

Programmer le *mini-riscv* consiste à remplir sa mémoire d'instruction avec une succession de mots de 32-bits. Ces mots représentent les instructions du programme que l'on souhaite exécuter, et doivent être conforme aux formats d'instructions présentés précédemment. Les programmes peuvent être écrit en langage assembleur RISC-V, et compilés avec **gcc** en utilisant le Makefile fourni dans le dossier `asm/`. Son utilisation est cependant limitée par le jeu d'instruction du *mini-riscv*.

---

```
% make help # Affiche l'aide
% make riscv BENCHMARK=<f> # Compile le programme <f> pour le mini-riscv
```

---

### 2.2.3 ASSEMBLEUR

Vous trouverez des exemples de code assembleur dans les fichiers `riscv_basic.S`, qui teste les fonctionnalités de base du processeur, et `riscv_fibo.S`, qui calcule les 20 premières itérations de la suite de Fibonacci. Utilisez ces codes dans votre banc d'essai afin de valider le bon fonctionnement de votre processeur. Remarquez l'utilisation de *pseudo-instructions* : `nop`, `li`, et `beqz`, qui sont converties par le compilateur par les transformations suivantes :

---

```
li rd, imm[31:0]

lui tmp, imm[31:12]
ori rd, tmp, imm[11:0]

beqz rs, offset

beq rs, x0, offset

nop

addi x0, x0, 0
```

---

### 2.2.4 REGISTRES

TABLEAU 1: Convention de nom des registres

Nom	Numéro	Utilisation
zero	x0	Valeur zéro
ra	x1	Adresse de retour des fonctions
sp	x2	Stack Pointer
gp	x3	Global Pointer
tp	x4	Thread Pointer
t0-t2	x5-x7	Temporaires
s0-s1	x8-x9	Sauvegardes
a0-a7	x10-x17	Arguments de fonctions
s2-s11	x18-x27	Sauvegardes
t3-t6	x28-x31	Temporaires



### 3 MICROARCHITECTURE

Cette partie présente la microarchitecture que vous devrez mettre au point dans le cadre de ce laboratoire. Elle met en œuvre l'architecture du *mini-riscv*, décrite au chapitre précédent, en utilisant 5 étages de pipeline. Dans un premier temps, la présentation des modules de base vous permettra de mettre au point les descriptions matérielles des composants du *mini-riscv*. Dans un second temps, la présentation de chaque étage de pipeline vous permettra de réaliser la description matérielle du *core*. Enfin, les explications sur la gestion des conflits dans un pipeline vous aidera à faire fonctionner votre système. Notez que les constantes utilisées de façon non génériques dans les modules sont définies dans un *package* (`riscv_pkg.vhd`) fourni dans le dossier de travail. Pour les inclure dans un module, utilisez les clauses présentées à la SOURCE 1.

SOURCE 1: *Package* contenant les constantes

```
library work;  
use work.riscv_pkg.all;
```

L'interface du *core* est présentée à la FIGURE 13, et son interface VHDL à la SOURCE 2. Les signaux `*_imem_*` constituent l'interface avec la mémoire d'instruction, tandis que les signaux `*_dmem_*` constituent l'interface avec la mémoire de donnée.



FIGURE 13: *mini-riscv* (core)

SOURCE 2: Interface VHDL du *mini-riscv* (core)

```
entity riscv_core is  
  port (  
    i_rstn      : in  std_logic;  
    i_clk       : in  std_logic;  
    i_imem_read : in  std_logic_vector(31 downto 0);  
    o_imem_addr : out std_logic_vector(8  downto 0);  
    i_dmem_read : in  std_logic_vector(31 downto 0);  
    o_dmem_we   : in  std_logic;  
    o_dmem_addr : out std_logic_vector(8  downto 0);  
    o_dmem_write : out std_logic_vector(31 downto 0));  
end entity riscv_core;
```

### 3.1 MODULES

Cette partie discute des modules qui composent le *mini-riscv*. Il s'agit d'un ALU, composé d'un *adder* et d'un *shifter* génériques, d'un compteur programme, et d'un banc de registres.

#### 3.1.1 ADDER

Le module *adder* utilise la technique de conception *ripple-carry*, basée sur des *half-adder* en série, qui propage la retenue d'un étage à l'autre pour réaliser une addition sur plusieurs bits. La FIGURE 14 montre le schéma de principe d'un *half-adder*, et la FIGURE 15 montre le schéma de principe de l'*adder*. La SOURCE 3 montre l'interface VHDL du module *adder*.

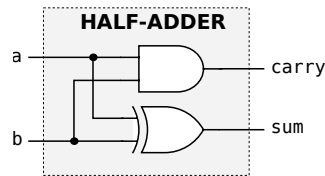


FIGURE 14: *half-adder*

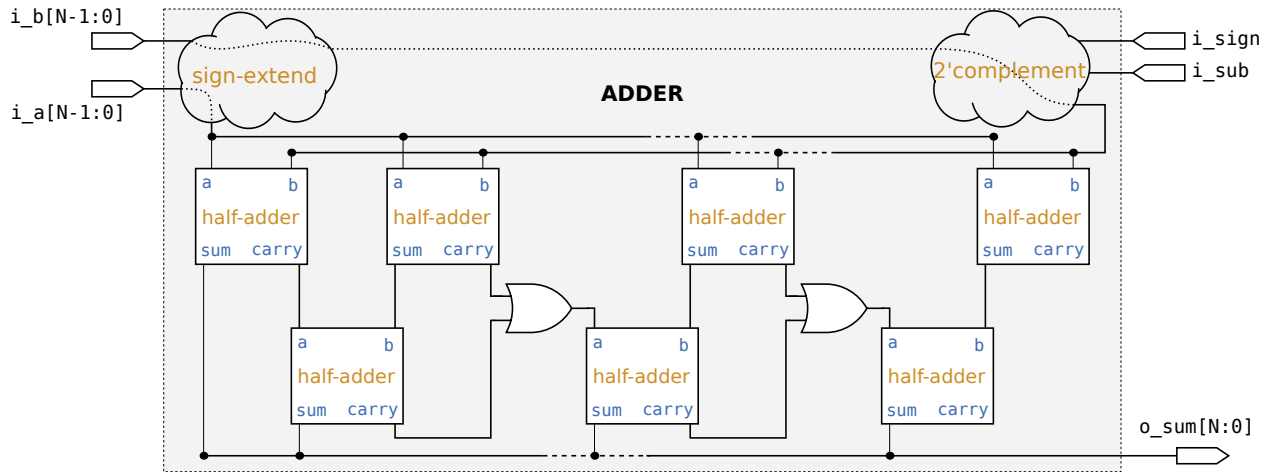


FIGURE 15: Module *adder*

Le module *adder* est combinatoire. Il comporte un paramètre générique  $N$ , qui définit la taille des signaux de données ainsi que le nombre de *half-adder* nécessaires. Par défaut,  $N = 32$ . L'*adder* réalise la somme des signaux  $i_a$  ( $N$  bits) et  $i_b$  ( $N$  bits) à partir d'une série de *half-adder*, et place le résultat sur le signal  $o\_sum$  ( $N+1$  bits).

Lorsque l'entrée  $i\_sign$  vaut 0, l'opération est réalisée sur des valeurs non-signées. À l'inverse, lorsque  $i\_sign$  vaut 1 l'opération est réalisée sur des valeurs signées. Lorsque l'entrée  $i\_sub$  vaut 0, l'opération à réaliser est une addition ( $i_a + i_b$ ). À l'inverse, lorsque  $i\_sub$  vaut 1, l'opération à réaliser est une soustraction ( $i_a - i_b$ ). Pour ce faire, le complément à deux du signal  $i_b$  devra être utilisé en entrée des *half-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;
    i_sub  : in  std_logic;
    o_sum  : out std_logic_vector(N downto 0));
end entity riscv_adder;

```

---

### 3.1.2 ALU

Le module *alu* réalise les opérations arithmétiques et logiques du *mini-riscv*. Il traite les opérandes *i\_src1* et *i\_src2*, et place un résultat sur la sortie *o\_res* en fonction des valeurs des signaux de contrôle *i\_opcode* et *i\_arith*. Il ne contient pas de paramètres génériques. La FIGURE 16 montre le schéma de principe de l'*alu*, et la SOURCE 4 montre son interface VHDL.

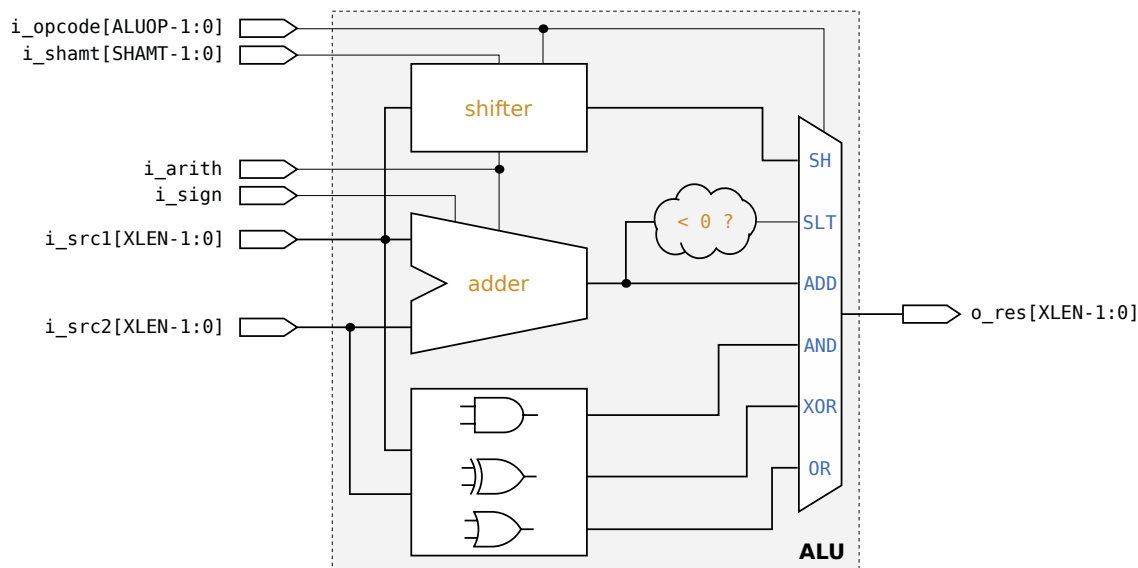


FIGURE 16: Module ALU

Le module *alu* est combinatoire. Il est composé de trois blocs principaux, dont les opérations sont résumées dans le TABLEAU 2.

- **Le bloc adder** est une instance du module *adder* réalisé précédemment. Il réalise les opérations d'addition et de soustraction (ADD) sur les signaux *i\_src1* et *i\_src2*. Notez que le signal *i\_arith* de l'ALU est connecté au signal *i\_sub* de l'*adder*. L'opération *set-less-than* (SLT) fait également partie du bloc adder : Lorsque le résultat en sortie de l'adder est négatif, le résultat vaut 1, autrement le résultat vaut 0.

- **Le bloc shifter** réalise les opérations de décalage à gauche (SL) et de décalage à droite (SR) sur le signal `i_src1`. Le nombre de bits à décaler est spécifié par la valeur du signal `i_shamt`. La nature du décalage est déduite des signaux de contrôle `i_opcode` et `i_arith` : Le décalage à droite doit être *logique* (des zéros remplacent les bits vacants) lorsque le signal `i_arith` vaut 0, et *arithmétique* (le bit de poids fort remplace les bits vacants) lorsque le signal `i_arith` vaut 1. Le décalage à gauche est toujours logique.
- **Le bloc logique** réalise les opérations logiques AND, OR, et XOR sur les signaux `i_src1` et `i_src2`.

SOURCE 4: Interface VHDL du module *alu*

---

```

entity riscv_alu is
  port (
    i_arith  : in  std_logic;
    i_sign   : in  std_logic;
    i_opcode : in  std_logic_vector(ALUOP_WIDTH-1 downto 0);
    i_shamt  : in  std_logic_vector(SHAMT_WIDTH-1 downto 0);
    i_src1   : in  std_logic_vector(XLEN-1 downto 0);
    i_src2   : in  std_logic_vector(XLEN-1 downto 0);
    o_res    : out std_logic_vector(XLEN-1 downto 0));
end entity riscv_alu;

```

---

TABLEAU 2: Opérations réalisées par l'ALU

Opcode	Bloc	Condition	Opération
ALUOP_ADD	Adder	$i\_arith = 0$	Addition
ALUOP_ADD	Adder	$i\_arith = 1$	Soustraction
ALUOP_SLT	Adder		1 si <code>adder_res</code> négatif, 0 sinon
ALUOP_SL	Shifter		Décalage à gauche
ALUOP_SR	Shifter	$i\_arith = 0$	Décalage à droite logique
ALUOP_SR	Shifter	$i\_arith = 1$	Décalage à droite arithmétique
ALUOP_XOR	Logique		XOR
ALUOP_OR	Logique		OR
ALUOP_AND	Logique		AND

### 3.1.3 COMPTEUR PROGRAMME

Le rôle du module *Compteur Programme* (PC) dans le *mini-riscv* consiste à gérer le bus d'adresse de la mémoire d'instruction. La sortie du compteur programme pointe toujours vers l'adresse en mémoire de la prochaine instruction à exécuter. Il comporte deux paramètres génériques : `RESET_VECTOR` et `XLEN`. Par défaut, `XLEN = 32`, et `RESET_VECTOR = 16#00000000#`. La FIGURE 17 montre le schéma de principe du *compteur programme*, et la SOURCE 5 montre son interface VHDL.

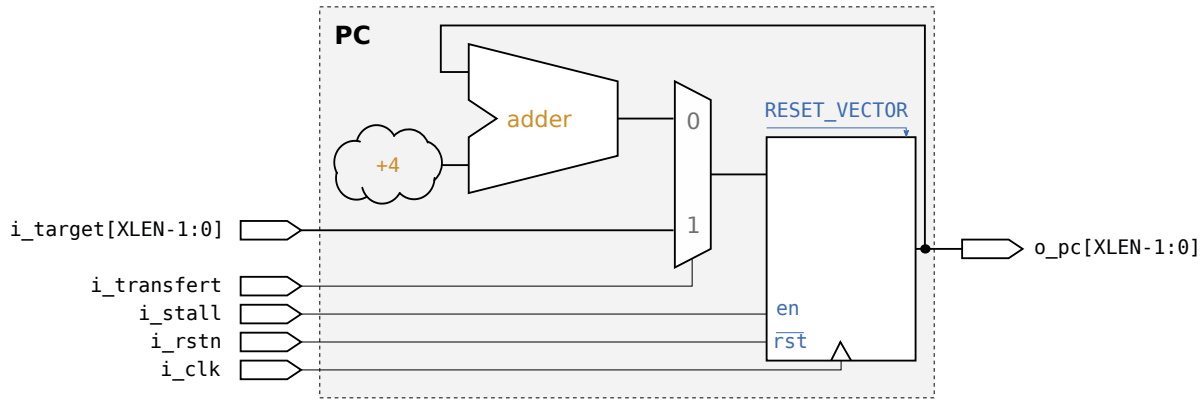


FIGURE 17: Module *Compteur Programme* (PC)

Le module est séquentiel, et doit se comporter de la façon suivante : À chaque front montant du signal d'horloge `i_clk`, la sortie `o_pc` est mise à jour (on note `pc` le signal en sortie du registre). La remise à zéro est asynchrone : `pc` est initialisé par `RESET_VECTOR` lorsque `i_rstn` vaut 0. Si `i_transfert` vaut 1, `pc` est affecté à `i_target`, sinon, `pc` est affecté à la sortie de l'*adder* (`pc + 4`). Lorsque `i_stall` vaut 1, `pc` conserve sa valeur précédente.

SOURCE 5: Interface VHDL du module *compteur programme*

```
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;
```

### 3.1.4 BANC DE REGISTRES

Le rôle du module *Banc de Registres* (RF) dans le *mini-riscv* consiste à contrôler l'accès en lecture et en écriture aux 32 registres de 32 bits. Il comporte deux paramètres génériques : `REG` et `XLEN`, qui définissent la taille des signaux d'adresse et de donnée. Par défaut, `REG = 5` et `XLEN = 32`. La FIGURE 18 montre le schéma de principe du *banc de registre*, et la SOURCE 6 montre son interface VHDL. Le module est séquentiel. Il doit se comporter de la façon suivante : Chaque adresse pointée par les signaux d'adresse (`i_addr_*`) correspond à un emplacement mémoire accessible par les signaux de données (`i_data_w` et `o_data_*`). L'adresse `0x0` contient toujours la valeur 0. La remise à zéro est asynchrone : La totalité des emplacements mémoires sont affectés à zéro lorsque `i_rstn` vaut 0. L'écriture des données s'effectue sur front montant du signal d'horloge `i_clk`. L'emplacement pointé par le signal d'adresse `i_addr_w` est

affecté : au signal `i_data_w` si l'entrée `i_we` vaut 1 ; à sa valeur précédente sinon. La lecture des données s'effectue sur front montant de l'horloge `i_clk`. Les signaux de sorties `o_data_ra/rb` reflètent les valeurs pointées par `i_addr_ra/rb` dans la mémoire. Lorsqu'une adresse de lecture est égale à l'adresse d'écriture, les signaux de sorties associés doivent refléter la valeur du signal `i_data_w`.

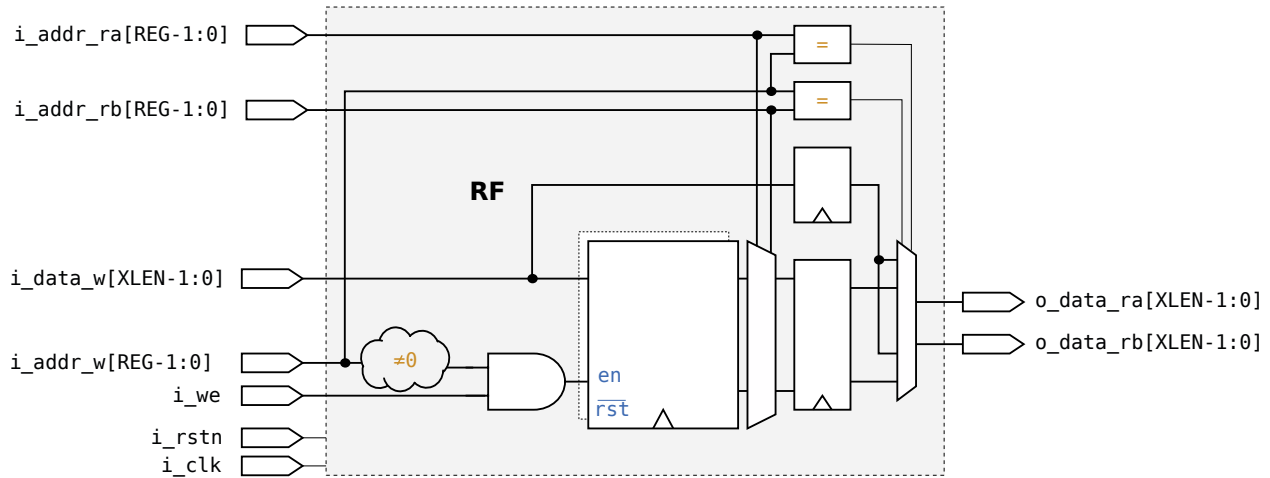


FIGURE 18: Module *Banc de Registres* (RF)

SOURCE 6: Interface VHDL du module *banc de 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-1 downto 0);
    o_data_ra  : out std_logic_vector(XLEN-1 downto 0);
    i_addr_rb  : in  std_logic_vector(REG-1 downto 0);
    o_data_rb  : out std_logic_vector(XLEN-1 downto 0);
    i_addr_w   : in  std_logic_vector(REG-1 downto 0);
    i_data_w   : in  std_logic_vector(XLEN-1 downto 0));
end entity riscv_rf;

```

---

## 3.2 PIPELINE

Le parallélisme d'instruction traduit la possibilité de traiter chacune des étapes d'une instruction en même temps que les autres étapes des instructions suivantes. Le *pipelining* est une technique de conception qui met en œuvre ce concept au niveau matériel : Chaque étape est associée à un étage du pipeline, qui peut être vu comme un couple logique combinatoire + registres.

La microarchitecture du *mini-riscv* est basée sur un pipeline à 5 étages, c'est-à-dire, qui décompose le traitement des instructions en 5 étapes distinctes, tel que l'illustre la FIGURE 19.

- **Fetch (IF)** : Lire la prochaine instruction à traiter dans la mémoire d'instruction.
- **Decode (ID)** : Déterminer le type d'instruction et lire les opérandes dans le Banc de Registres.
- **Execute (EX)** : Appliquer les opérations nécessaires sur les opérandes.
- **Memory (ME)** : Le cas échéant, accéder à la mémoire de donnée pour lire ou écrire une valeur.
- **Write-Back (WB)** : Écrire le résultat dans le Banc de Registres.



FIGURE 19: Pipeline à 5 étages

### 3.2.1 INSTRUCTION FETCH (IF)

Dans l'étape IF, le *mini-riscv* va chercher une nouvelle instruction dans la mémoire d'instruction, tel qu'illustré à la FIGURE 20. Le PC est le module chargé de fournir, à chaque cycle, l'adresse de la prochaine instruction à traiter. Par conséquent, la sortie du PC doit être interfacée avec le bus d'adresse de la mémoire d'instruction. La valeur retournée par la mémoire contient la prochaine instruction à traiter, dans un des formats décrit à la section 2. Ce signal est ensuite sauvegardé dans le registre d'état IF/ID. Notez que plusieurs signaux de données et de contrôle viennent de l'étage EX.

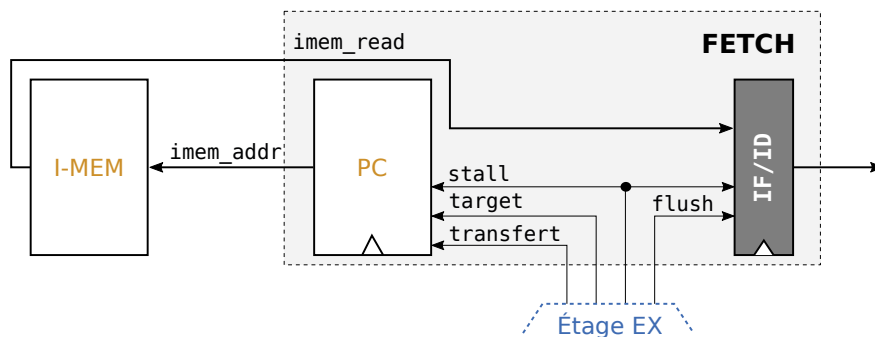


FIGURE 20: Étage de pipeline IF

### 3.2.2 INSTRUCTION DECODE (ID)

Dans l'étape ID, le *mini-riscv* génère un ensemble de signaux de données et de contrôles à partir des informations extraites du mot d'instruction fourni par le registre IF/ID, et produit les opérandes pour l'opération à réaliser. En particulier, cette étape consiste à :

- Générer les signaux permettant d'identifier l'instruction (`opcode`, `funct3`, `funct7`).
- Générer les signaux d'adresses permettant de lire les opérandes dans le Banc de registres (`rs1_addr`, `rs2_addr`), ainsi que le signal d'adresse permettant d'écrire dans le Banc de registres (`rd_addr`). Ce signal devra être transmis à travers tous les étages du pipeline jusqu'à l'étape WB.
- Accéder au Banc de registres à partir de ces adresses pour produire les opérandes. Notez que le Banc de registres possède une latence de 1 cycle.
- Générer les valeurs immédiates selon les formats présentés à la FIGURE 2.
- Générer l'ensemble des signaux de contrôles en fonction de l'instruction : est-ce un branchement? Un saut? Doit-on accéder (lecture ou écriture) à la mémoire de donnée? Doit-on écrire le résultat dans le Banc de registres? etc.

Tous les signaux de contrôles et les valeurs immédiates doivent ensuite être sauvegardées dans le registre d'état ID/EX.

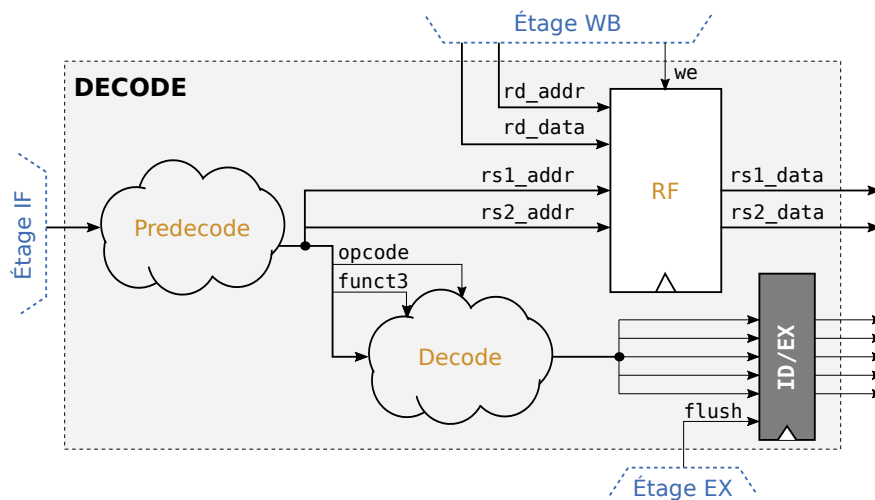


FIGURE 21: Étage de pipeline ID



### 3.2.3 EXECUTE (EX)

Dans l'étape EX, le *mini-riscv* exécute l'opération prévue par l'instruction. Les opérations arithmétiques et logiques sont traitées par le module ALU, et c'est à cette étape que les prédictions de branches et de sauts sont prises en charge, comme le montre la FIGURE 22. En particulier, cet étage est responsable de :

- Générer le résultat des opérations arithmétiques et logiques (add, sub, addi, and, lui, etc.)
- Générer l'adresse de lecture ou d'écriture dans la mémoire de donnée (lw, sw)
- Résoudre la prédiction de branchement (jal, jalr, beq).
- Générer l'adresse de destination des sauts (jal, jalr, beq)

Notez que les opérandes sur lesquels seront effectués les opérations doivent être calculés en amont, dépendamment du format de l'instruction et des signaux de contrôle générés à l'étape ID.

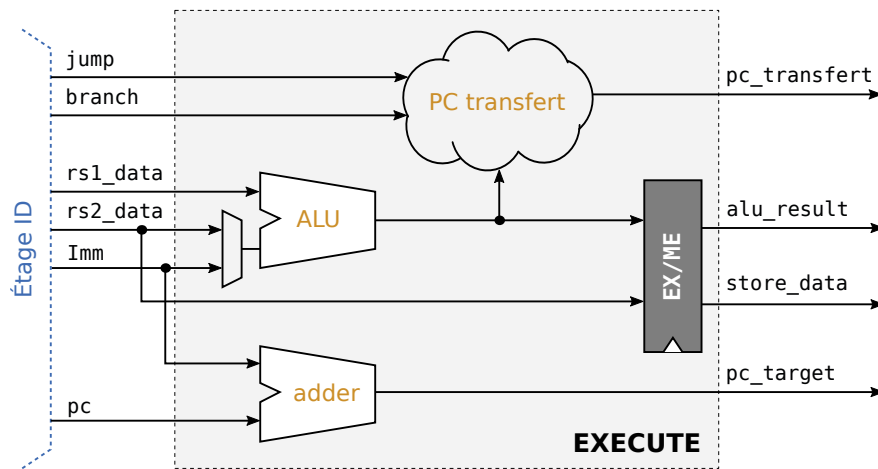


FIGURE 22: Étage de pipeline EX

### 3.2.4 MEMORY ACCESS (ME)

Dans l'étape ME, le *mini-riscv* n'accède à la mémoire de donnée que dans le cas d'une instruction *load* (lw) ou *store* (sw). Les résultats sont transmis au registre ME/WB, comme le montre la FIGURE 23.

### 3.2.5 WRITE-BACK (WB)

Dans l'étape WB, le *mini-riscv* écrit, le cas échéant, le résultat de l'instruction dans le Banc de registres à l'adresse pointée par *rd\_addr*. Dans le cas d'une opération *load*, la valeur issue de la mémoire de donnée (*dmem\_read*) doit être utilisée, dans les autres cas, la valeur issue de l'alu (*alu\_result*) doit être utilisée, comme le montre la FIGURE 24.

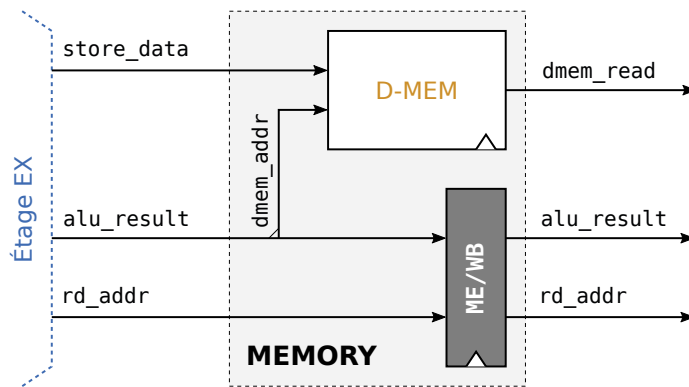


FIGURE 23: Étage de pipeline ME

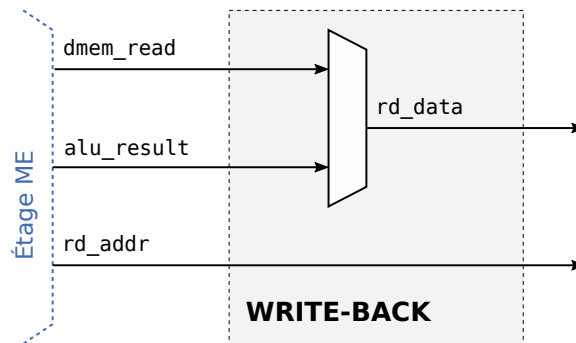


FIGURE 24: Étage de pipeline WB

### 3.3 GESTION DES CONFLITS

La mise en parallèle des instructions dans un pipeline génère des problèmes liés à la séquentialité des opérations. Un conflit se produit lorsque la séquentialité des opérations dans le programme est modifiée par le pipeline. Pour permettre le rétablissement de la séquentialité originale du programme, le flot d'instruction doit parfois être interrompu (*stall*), et certains résultats doivent être transmis entre les différents étages de pipeline (*forwarding*).

Dans le *mini-riscv*, trois catégories de conflits peuvent se présenter : structurel, de donnée, ou de contrôle. Votre microarchitecture devra mettre en œuvre les mécanismes matériels permettant de prendre en charge ces conflits. Notez que le programme de test (`riscv_fibo.asm`) a été conçu pour mettre en évidence le bon fonctionnement de votre microarchitecture à cet égard.

- **Un conflit structurel** apparaît lorsque deux instructions cherchent à accéder à une même ressource au même moment. Dans le *mini-riscv*, ce problème se produit lorsque deux instructions veulent simultanément lire et écrire une valeur à la même adresse dans le Banc de registre. Pour pallier ce problème, le Banc de registre a été conçu de sorte que la valeur à écrire soit dupliquée dans un registre indépendant.
- **Un conflit de donnée** peut se produire lorsqu'une instruction dépend du résultat d'une instruction précédente. Dans le *mini-riscv*, ce problème apparaît lorsqu'une instruction cherche à lire un

opérande dans le Banc de registre (étape ID) qui n'a pas encore été mis à jour par une instruction précédente (étape WB). La résolution de la majorité de ces conflits s'effectue par la méthode dites de *forwarding* : transmettre le résultat d'une instruction des étages ME et WB vers l'étage EX. Dans le cas de l'instruction *load*, le forwarding seul ne suffit plus étant donné que le résultat de l'instruction n'est connu qu'à partir de l'étape ME. Dans le *mini-riscv*, la résolution de ce type de conflit, en plus du *forwarding*, s'effectue en appliquant un *stall* de 1-cycle sur le pipeline. La FIGURE 25 montre la mise en œuvre du forwarding dans le pipeline.

- **Un conflit de contrôle** se produit à chaque fois que le compteur programme est altéré par un saut ou par un branchement. Il faut 3 cycles (étape EX) avant qu'une instruction *jal*, *jalr* ou *beq* n'affecte le compteur programme. Pendant les deux cycles qui précèdent la prise de décision, deux instructions se chargent dans le pipeline. Si le programme prévoit de faire un saut vers une nouvelle adresse, ces deux instructions doivent être retirées du pipeline (*flush*), sinon elles risquent de modifier l'état du processeur. Le *mini-riscv* résout les branchements conditionnels (*beq*) avec la stratégie de prédiction *branchement non pris* (*predict non-taken*). Cette stratégie suppose que, par défaut, le branchement n'est pas pris (par défaut la condition est fausse). Les instructions qui suivent le branchement sont donc chargées et débutent leur exécution. À l'étape EX, si le branchement est pris les deux instructions sont éliminées (*flush*), s'il n'est pas pris (tel que supposé) les deux instructions continuent leur exécution normalement.

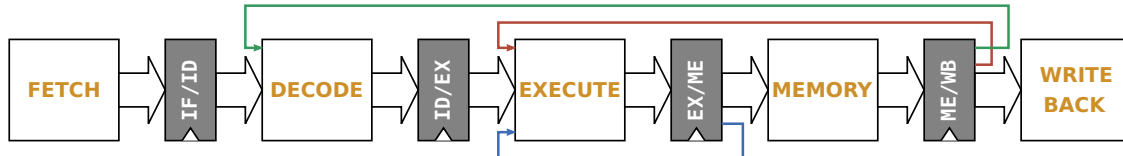


FIGURE 25: Mise en œuvre du *forwarding* dans le pipeline