

Documentation

Microprocesseur *mini-mips*

Professeur:
Yvon Savaria
yvon.savaria@polymtl.ca

Chargé de laboratoire:
Mickaël Fiorentino
mickael.fiorentino@polymtl.ca

Automne 2019

TABLE DES MATIÈRES

1	Introduction	2
2	Architecture	3
2.1	Jeu d'instruction	3
2.1.1	Branchements inconditionnels	4
2.1.2	Branchements conditionnels	5
2.1.3	Accès à la mémoire de donnée	5
2.1.4	Opérations arithmétiques & logiques	6
2.1.5	Opérations de décalages	7
2.2	Programmation	8
2.2.1	Interfaces Mémoires	8
2.2.2	Assembleur	8
3	Microarchitecture	10
3.1	Modules	11
3.1.1	Shifter	11
3.1.2	ALU	12
3.1.3	Compteur Programme	13
3.1.4	Banc de Registres	14
3.1.5	Compteur de performance	15
3.2	Pipeline	16
3.2.1	Fetch (F)	16
3.2.2	Decode (D)	16
3.2.3	Execute (E)	17
3.2.4	Update (U)	17
3.3	Gestion des conflits	17

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 semiconducteurs : En réduisant les tailles des transistors on obtient une densité d'intégration plus importante, une augmentation des fréquences d'opérations, et une diminution de la consommation énergétique.
2. Les techniques de conceptions des microarchitectures de processeurs : les pipeline, les caches, les prédicteurs de branche etc. permettent de tirer le meilleur parti des technologies de semiconducteurs en augmentant les performances par cycles.

À titre d'exemple, la FIGURE.1 montre le dessin des masques du microprocesseur *MIPS R5000* (1996), utilisé entre autres dans les bornes de jeux d'arcades. Il a été fabriqué par NEC en technologie CMOS $0.35\ \mu\text{m}$. Avec les caches, il contient 3.6 millions de transistors et occupe $84\ \text{mm}^2$ de surface.

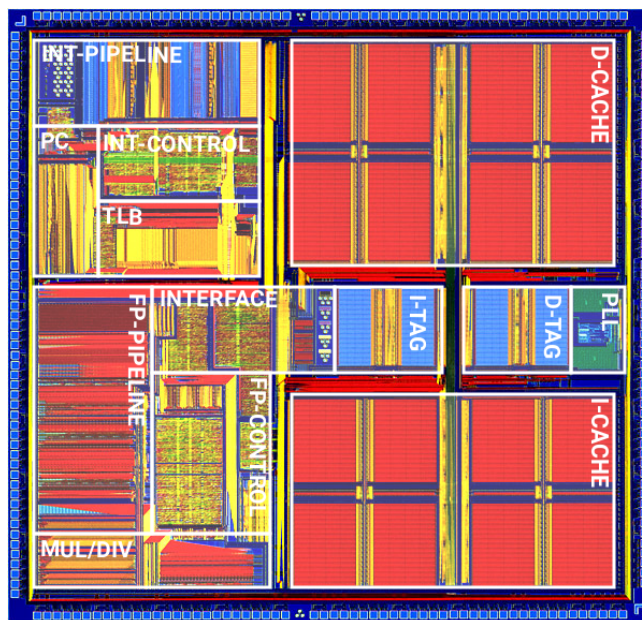


FIGURE. 1 – Exemple : MIPS R5000

Le projet du cours ELE8304 consiste à réaliser l'implémentation complète d'un microprocesseur MIPS simple que nous appellerons *mini-mips*. Son jeu d'instruction est dérivé de l'architecture MIPS32, et sa microarchitecture utilise un pipeline à 4 étages. L'objectif de ce projet 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-mips*, 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-mips*, 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 associé à un compilateur. Le jeu d'instruction du *mini-mips* est un sous ensemble du jeux d'instruction [MIPS32](#). 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-mips* possède un banc de registres de 32×32 bits ($r0 \dots r31$), avec $r0=0x0$.
- 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-mips* utilise un format d'instruction fixe de 32 bits. On distingue 3 formats d'instructions illustrés à la [FIGURE.2](#) : *R-type* pour les opérations entre les registres, *I-type* pour les opérations entre un registre et une valeur immédiate, et *J-type* pour les opération de sauts inconditionnels. Les portions *funct* et *op* encodent les types d'instructions (*add*, *sub*, *beq*, etc.). Les portions *rs* et *rt* encodent l'adresse des registres opérandes, et *rd* encodent l'adresse du registre de destination. La portion *shamt* encode une valeur immédiate sur 5-bit pour les opérations de décalages. Les portions *I-imm* et *J-imm* encodent des valeurs immédiates, respectivement de 16-bits et 26-bits.

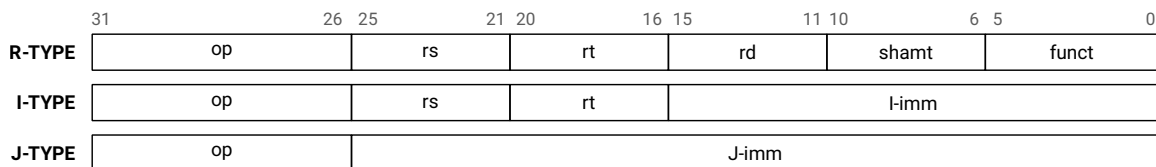


FIGURE. 2 – Formats d'instructions

La FIGURE.3 liste les 27 instructions supportées par le *mini-mips*. Notez que les instructions au format *R-type* ont un opcode de 000000.

000010	J-imm					J
000011	J-imm					JAL
000000	rs	N/A	rd	N/A	001001	JALR
000100	rs	rt	l-imm			BEQ
100011	rs	rt	l-imm			LW
101011	rs	rt	l-imm			SW
001111	N/A	rt	l-imm			LUI
001000	rs	rt	l-imm			ADDI
001001	rs	rt	l-imm			ADDIU
000000	rs	rt	rd	N/A	100000	ADD
000000	rs	rt	rd	N/A	100010	SUB
000000	N/A	rt	rd	shamt	000000	SLL
000000	N/A	rt	rd	shamt	000010	SRL
000000	N/A	rt	rd	shamt	000011	SRA
000000	rs	rt	rd	N/A	000100	SLLV
000000	rs	rt	rd	N/A	000110	SRLV
000000	rs	rt	rd	N/A	000111	SRAV
001010	rs	rt	l-imm			SLTI
001011	rs	rt	l-imm			SLTIU
000000	rs	rt	rd	N/A	101010	SLT
000000	rs	rt	rd	N/A	101011	SLTU
001110	rs	rt	l-imm			XORI
001101	rs	rt	l-imm			ORI
001100	rs	rt	l-imm			ANDI
000000	rs	rt	rd	N/A	100110	XOR
000000	rs	rt	rd	N/A	100101	OR
000000	rs	rt	rd	N/A	100100	AND

FIGURE. 3 – Liste des instructions supportées par le *mini-mips*

2.1.1 BRANCHEMENTS INCONDITIONNELS

Les opérations de branchements inconditionnels : **J** (*Jump*), **JAL** (*Jump And Link*) et **JALR** (*Jump And Link Register*), sont présentées à la FIGURE.4. Elles modifient la valeur du compteur programme.

J, JAL	J-imm				
000000	rs	N/A	rd	N/A	JALR

FIGURE. 4 – Instructions de branchement inconditionnels

L'instruction **J** utilise le format *J-type*, où l'immédiat de 26-bits encode la portée du saut. J-imm est étendu sur 32-bits non-signés pour former l'adresse de destination du saut (pc) à partir de la valeur courante du compteur programme :

$$pc \leftarrow pc[31:28] \& J-imm \& 0.$$

L'instruction **JAL** utilise le format *J-type*, où l'immédiate de 26-bits encode la portée du saut. L'adresse de l'instruction consécutive à l'instruction de branchement ($pc+4$) est sauvegardée dans le registre $r31$. $J-imm$ est étendu sur 32-bits non-signés pour former l'adresse de destination du saut (pc) à partir de la valeur courante du compteur programme :

$$RF[31] \leftarrow pc + 4$$

$$pc \leftarrow pc[31:28] \& J-imm \& 0$$

L'instruction **JALR** utilise le format *R-type*. L'adresse de l'instruction consécutive à l'instruction de branchement ($pc+4$) est sauvegardée dans le registre de destination (rd). L'adresse de destination du saut (pc) est contenue dans le registre pointé par rs :

$$RF[rd] \leftarrow pc + 4$$

$$pc \leftarrow RF[rs]$$

2.1.2 BRANCHEMENTS CONDITIONNELS

L'instruction de branchement conditionnel **BEQ** (*Branch On Equal*) est présentée à la FIGURE.5. Elle modifie la valeur du compteur programme.

BEQ	rs	rt	$I-imm$
------------	------	------	---------

FIGURE. 5 – Instruction de branchement conditionnel

L'instruction **BEQ** utilise le format *I-type*, où l'immédiate de 16-bits encode la portée du saut. Le branchement est réalisé relativement à la valeur courante du compteur programme, si la condition de branchement est remplie (les registres pointés par rs et rt sont égaux). $I-imm$ est étendu sur 32-bits signés pour former l'adresse de destination du saut (pc) à partir de la valeur courante du compteur programme :

$$pc \leftarrow RF[rs] = RF[rt] ? pc + sign(I-imm) \& 0 : pc+4$$

2.1.3 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.6. Elles opèrent un transfert entre le banc de registre et la mémoire de donnée.

LW	rs	rt	$I-imm$
SW	rs	rt	$I-imm$

FIGURE. 6 – Instructions d'accès à la mémoire de donnée

L'instruction **LW** utilise le format *I-type*, où l'immédiate de 16-bits (étendu sur 32-bits signés) encode l'adresse de lecture relativement à l'adresse contenue dans le registre *rs*. La valeur lue depuis la mémoire de donnée à cette adresse est sauvegardée dans le registre *rt* :

$$RF[rt] \leftarrow DMEM[RF[rs] + sign(I-imm)]$$

L'instruction **SW** utilise le format *I-type*, où l'immédiate de 16-bits (étendu sur 32-bits signés) encode l'adresse d'écriture relativement à l'adresse contenue dans le registre *rs*. La valeur du registre *rt* est écrite en mémoire à cette adresse :

$$DMEM[RF[rs] + sign(I-imm)] \leftarrow RF[rt]$$

2.1.4 OPÉRATIONS ARITHMÉTIQUES & LOGIQUES

Les opérations arithmétiques et logiques se déclinent en deux variantes : une variante utilisant le format *R-type* et une variante utilisant le format *I-type*. Elles sont présentées à la FIGURE.7.

LUI, ADDI[U], SLTI[U], XORI, ORI, ANDI	rs	rt	I-imm		
000000	rs	rt	rd	shamt	ADD, SUB, SLT[U], XOR, OR, AND

FIGURE. 7 – Opérations arithmétiques & logiques

L'instruction **LUI** (*Load Upper Immediate*) utilise le format *I-type*. Elle place les 16 bits de son immédiate dans les 16 bits de poids fort du registre de destination *rt*, et remplit le reste avec des zéros.

$$RF[rt] \leftarrow I-imm \& 0$$

Les opérations **ADDI** et **ADDIU** utilisent le format *I-type*. Elles opèrent une addition entre le contenu du registre *rs* et la valeur immédiate *I-imm* étendue sur 32-bits signés et non-signés respectivement :

$$RF[rt] \leftarrow RF[rs] + (un)sign(I-imm)$$

Les opérations **ADD** et **SUB** utilisent le format *R-type*. Elles opèrent respectivement une addition et une soustraction entre le contenu du registre *rs* et le contenu du registre *rt* :

$$RF[rd] \leftarrow RF[rs] + / - RF[rt]$$

Les opérations **ANDI**, **ORI** et **XORI** utilisent le format *I-type*. Elles opèrent respectivement un ET, un OU, et un XOR logique entre le contenu du registre *rs* et la valeur immédiate *I-imm* étendue sur 32-bits non-signés.

$$RF[rt] \leftarrow RF[rs] \text{ AND/OR/XOR } unsign(I-imm)$$

Les opérations **AND**, **OR** et **XOR** utilisent le format *R-type*. Elles opèrent respectivement un ET, un OU, et un XOR logique entre le contenu du registre *rs* et le contenu du registre *rt*.

$$RF[rd] \leftarrow RF[rs] \text{ AND/OR/XOR } RF[rt]$$

Les opérations **SLTI** (*Set Less Than Immediate*) et **SLTIU** (*Set Less Than Immediate Unsigned*) utilisent le format *I-type*. Elles comparent la valeur du registre *rs* avec la valeur immédiate *I-imm* étendue sur 32-bits signés et non-signés respectivement :

$$RF[rt] \leftarrow RF[rs] < (un)sign(I-imm) ? 1 : 0$$

Les opérations **SLT** (*Set Less Than*) et **SLTU** (*Set Less Than Unsigned*) utilisent le format *R-type*. Elles comparent la valeur du registre *rs* avec le contenu du registre *rt* :

$$RF[rd] \leftarrow RF[rs] < (un)sign(RF[rt]) ? 1 : 0$$

2.1.5 OPÉRATIONS DE DÉCALAGES

se déclinent en deux variantes : une variante utilisant le format *R-type* et une variante utilisant le format *I-type*. Elles sont présentées à la FIGURE.8.

000000	N/A	rt	rd	shamt	SLL, SRL, SRA
000000	rs	rt	rd	N/A	SLLV, SRLV, SRAV

FIGURE. 8 – Opérations de décalage

L'opération **SLL** (*Shift Left Logical*) utilise le format *R-type*. Elle réalise un décalage à gauche du registre *rt* par la valeur immédiate *shamt* :

$$RF[rd] \leftarrow RF[rt] << shamt$$

L'opération **SLLV** (*Shift Left Logical Variable*) utilise le format *R-type*. Elle réalise un décalage à gauche du registre *rt* par la valeur des 5-bits de poids faibles du registre *rs* :

$$RF[rd] \leftarrow RF[rt] << RF[rs]$$

Les opérations **SRL** (*Shift Right Logical*) et **SRA** (*Shift Right Arithmetic*) utilisent le format *R-type*. Elles réalisent respectivement un décalage à droite logique (on insère des zéro dans les champs vides) et arithmétique (on insère le bit poids fort dans les champs vides) du registre *rt* par la valeur immédiate *shamt* :

$$RF[rd] \leftarrow RF[rt] >> shamt$$

Les opérations **SRLV** (*Shift Right Logical Variable*) **SRAV** (*Shift Right Arithmetic Variable*) utilisent le format *R-type*. Elles réalisent respectivement un décalage à droite logique (on insère des zéro dans les champs vides) et arithmétique (on insère le bit poids fort dans les champs vides) du registre *rt* par la valeur des 5-bits de poids faibles du registre *rs* :

$$RF[rd] \leftarrow RF[rt] >> RF[rs]$$

2.2 PROGRAMMATION

2.2.1 INTERFACES MÉMOIRES

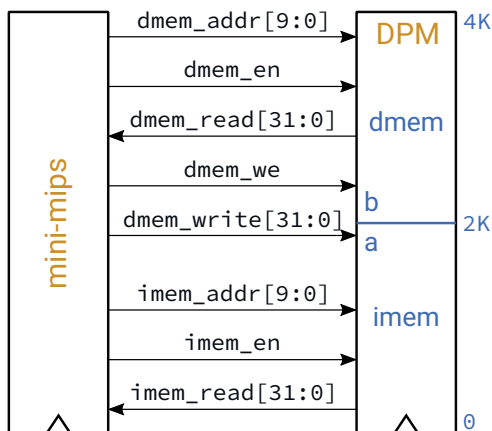


FIGURE. 9 – Mémoires

Le système de mémoire du *mini-mips* (FIGURE.9) 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 2k : L'espace d'adresses de la mémoire d'instruction (*imem*) est entre 0 et 2k sur le *port a*, et l'espace d'adresse de la mémoire de donnée (*dmem*) est entre 2k et 4k sur le *port b*. La mémoire est une instance du fichier *dpm.vhd*, 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 ASSEMBLEUR

Programmer le *mini-mips* 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 MIPS, 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-mips*.

```
% make help # Affiche l'aide
% make mips BENCHMARK=<f> # Compile le programme <f>
```

Le code assembleur présenté ici est extrait du code *mips_fibo.S*, qui calcule les 20 premières itérations de la suite de Fibonacci. Inspirez-vous de ce code pour créer vos bancs d'essais afin de valider le bon fonctionnement de votre processeur. Remarquez l'utilisation de *pseudo-instructions* : *nop*, *li*, et *beqz*, qui sont prises en charge par le compilateur par les transformations suivantes :

```
li := lui $t0, rs, imm[31:16]; ori rt, $t0, imm[15:0]

beqz := beq rt, $0, addr

nop := sll $0, $0, 0
```

```

1  #define MAX_FIBO_LOOP 20
2  #define MAX_FIBO_VAL  6765
3  main:
4      jal    fibo
5      li     $t0, MAX_FIBO_VAL
6      beq    $v0, $t0, pass
7      j      fail
8      nop
9  fibo:
10     li     $a0, 1
11     li     $s0, MAX_FIBO_LOOP
12     li     $t0, 0
13     li     $t1, 1
14     sw     $t0, 0($sp)
15     sw     $t1, -4($sp)
16  fiboloop:
17     sltu    $t3, $a0, $s0
18     beqz    $t3, endmain
19     addi    $a0, $a0, 1
20     lw      $t0, 0($sp)
21     lw      $t1, -4($sp)
22     add     $t2, $t0, $t1
23     sw      $t2, -8($sp)
24     sra     $sp, $sp, 2
25     addi    $sp, $sp, -1
26     sll     $sp, $sp, 2
27     j       fiboloop
28  endmain:
29     addi    $v0, $t2, 0
30     jalr    $0, $ra

```

3 MICROARCHITECTURE

Cette partie présente la microarchitecture que vous devrez concevoir dans le cadre du projet. Elle met en œuvre l'architecture du *mini-mips*, décrite au chapitre précédent, en utilisant 4 étages de pipeline. Dans un premier temps, la présentation des modules de base vous permettra de réaliser les descriptions matérielles des composants du *mini-mips*. 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. Les constantes utilisées de façon non génériques dans les modules *doivent* être définies dans un *package* (*mips_pkg.vhd*). Pour les inclure dans un module, utilisez les clauses suivantes :

```
library work;  
use work.mips_pkg.all;
```

L'interface du *core* est présentée à la FIGURE.10, et son interface VHDL à la SOURCE.1. Les signaux *_imem_* constituent l'interface avec la mémoire d'instruction, tandis que les signaux *_dmem_* constituent l'interface avec la mémoire de donnée.

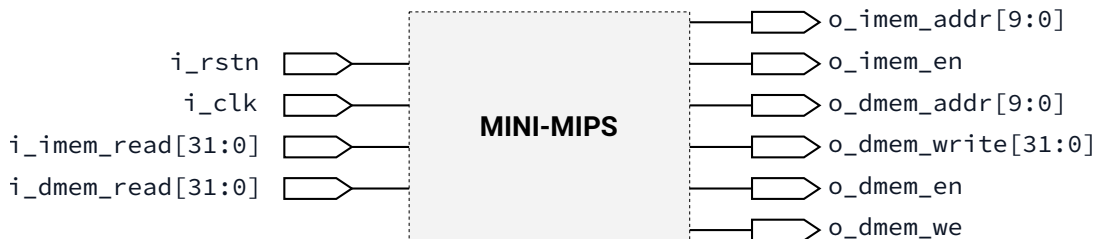


FIGURE. 10 – *mini-mips* (*mips_core.vhd*)

SOURCE. 1 – Interface VHDL du *mini-mips* (*core*)

```
entity mips_core is  
  port (  
    i_rstn      : in  std_logic;  
    i_clk       : in  std_logic;  
    o_imem_en   : out std_logic;  
    o_imem_addr : out std_logic_vector(MEM_ADDR_W-1 downto 0);  
    i_imem_read : in  std_logic_vector(RLEN-1 downto 0);  
    o_dmem_en   : out std_logic;  
    o_dmem_we   : out std_logic;  
    o_dmem_addr : out std_logic_vector(MEM_ADDR_W-1 downto 0);  
    i_dmem_read : in  std_logic_vector(RLEN-1 downto 0);  
    o_dmem_write : out std_logic_vector(RLEN-1 downto 0));  
end entity mips_core;
```

3.1 MODULES

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

3.1.1 SHIFTER

Le module *shifter* permet de décaler les bits d'un signal vers la droite ou vers la gauche. Il comporte un paramètre générique N qui définit la taille des signaux d'entrées et de sorties. Par défaut, $N = 5$. La FIGURE.11 montre le schéma de principe du *shifter*, et la SOURCE.2 montre son interface VHDL.

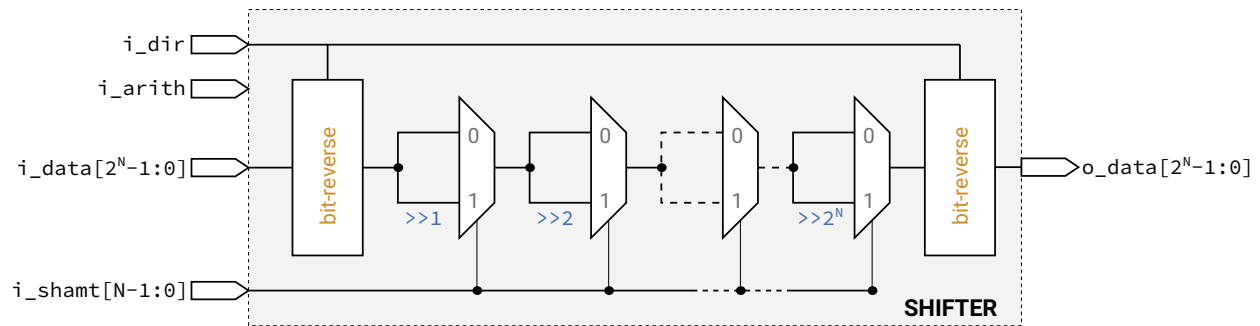


FIGURE. 11 – Module *shifter*

SOURCE. 2 – Interface VHDL du module *shifter*

```
entity mips_shift is
  generic (
    N : positive := 5);
  port (
    i_data  : in  std_logic_vector(2*N-1 downto 0);
    i_shamt : in  std_logic_vector(N-1 downto 0);
    i_arith : in  std_logic;
    i_dir   : in  std_logic;
    o_data  : out std_logic_vector(2*N-1 downto 0));
end entity mips_shift;
```

Le module *shifter* est combinatoire. Il doit se comporter de la façon suivante : La sortie `o_data` est affectée de l'entrée `i_data` décalée vers la droite lorsque le signal `i_dir` vaut 1, vers la gauche lorsqu'il vaut 0. Le nombre de bits à décaler est spécifié par le signal `i_shamt`. Les deux opérations de décalages (gauche et droite) sont réalisées à partir du décalage à droite, afin de minimiser la logique nécessaire. Pour réaliser un décalage à gauche, les bits du signal à décaler sont renversés en amont et en aval de l'opération de décalage à droite. Le décalage à droite doit être *logique* (des zéros remplacent les bits vacant) lorsque le signal `i_arith` vaut 0, et *arithmétique* (le bit de poids fort remplace les bits vacant) lorsque le signal `i_arith` vaut 1. Le décalage à gauche est toujours logique.

3.1.2 ALU

Le module *alu* réalise les opérations arithmétiques et logiques du *mini-mips*. 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*, *i_arith* et *i_sign*. Il ne contient pas de paramètres génériques. La FIGURE.12 montre le schéma de principe de l'*alu*, et la SOURCE.3 montre son interface VHDL.

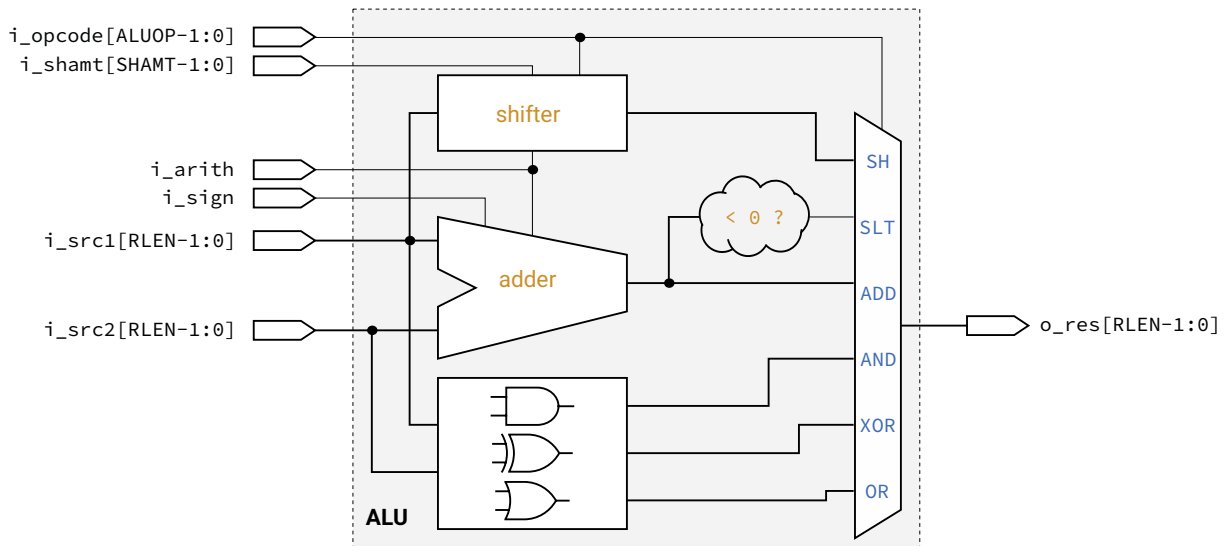


FIGURE. 12 – Module ALU

SOURCE. 3 – Interface VHDL du module *alu*

```

entity mips_alu is
  port (
    i_arith  : in  std_logic;
    i_sign   : in  std_logic;
    i_opcode : in  std_logic_vector(ALUOP-1 downto 0);
    i_shamt  : in  std_logic_vector(SHAMT-1 downto 0);
    i_src1   : in  std_logic_vector(RLEN-1 downto 0);
    i_src2   : in  std_logic_vector(RLEN-1 downto 0);
    o_res    : out std_logic_vector(RLEN-1 downto 0));
end entity mips_alu;

```

Le module *alu* est combinatoire. Il est composé de trois blocs principaux, dont les opérations sont résumées dans le TABLEAU.1.

- **Le bloc shifter** est une instance du module *shifter* réalisé précédemment. Il 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 bloc adder** réalise les opérations d'addition et de soustraction (ADD) sur les signaux `i_src1` et `i_src2` à partir des opérateurs définis dans la librairie `ieee.numeric_std`. L'opération *set-less-than* (SLT) utilise la sortie du bloc adder : Lorsque la sortie est négative, le résultat vaut 1, autrement le résultat vaut 0.
- **Le bloc logique** réalise les opérations logiques AND, OR, et XOR sur les signaux `i_src1` et `i_src2`.

TABLEAU. 1 – Opérations réalisées par l'ALU

Opcode	Bloc	Condition	Opération
ALUOP_ADD	Adder	$i_arith = 0$	$i_src1 + i_src2$
ALUOP_ADD	Adder	$i_arith = 1$	$i_src1 - i_src2$
ALUOP_SLT	Adder		$i_src1 - i_src2 < 0 ? 1 : 0$
ALUOP_SL	Shifter		$i_src1 \ll i_src2$
ALUOP_SR	Shifter	$i_arith = 0$	$i_src1 \gg i_src2$
ALUOP_SR	Shifter	$i_arith = 1$	$i_src1 \ggg i_src2$
ALUOP_XOR	Logique		$i_src1 \oplus i_src2$
ALUOP_OR	Logique		$i_src1 \vee i_src2$
ALUOP_AND	Logique		$i_src1 \wedge i_src2$

3.1.3 COMPTEUR PROGRAMME

Le rôle du module *Compteur Programme* (PC) dans le *mini-mips* 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. La FIGURE.13 montre le schéma de principe du *compteur programme*, et la SOURCE.4 montre son interface VHDL.

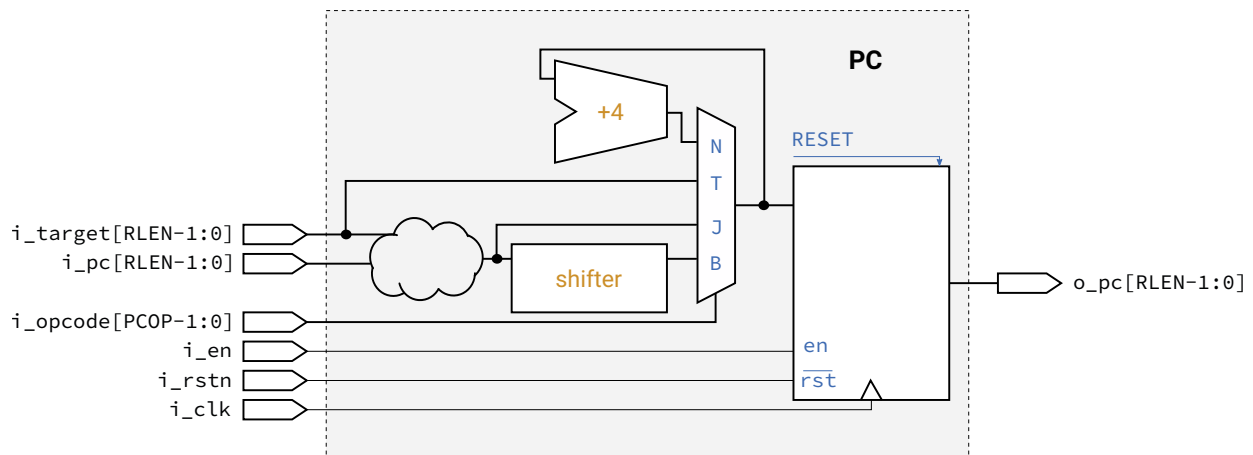


FIGURE. 13 – Module *Compteur Programme* (PC)

```

entity mips_pc is
  port (
    i_rstn    : in  std_logic;
    i_clk     : in  std_logic;
    i_en      : in  std_logic;
    i_opcode  : in  std_logic_vector(PCOP-1 downto 0);
    i_target  : in  std_logic_vector(RLEN-1 downto 0);
    i_pc      : in  std_logic_vector(RLEN-1 downto 0);
    o_pc      : out std_logic_vector(RLEN-1 downto 0));
end entity mips_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* lorsque *i_rstn* vaut 0. Lorsque *i_en* vaut 0, *pc* conserve sa valeur précédente. Dans les autres cas, la sortie du module dépend de *i_opcode* :

- *N (next)* : $pc \leftarrow pc + 4$
- *T (target)* : $pc \leftarrow i_target$
- *J (jump)* : $pc \leftarrow i_pc[31:28] \& i_target[25:0] \& 00$
- *B (branch)* : $pc \leftarrow i_pc + (i_target \ll 2)$

3.1.4 BANC DE REGISTRES

Le rôle du module *Banc de Registres* (RF) dans le *mini-mips* consiste à contrôler l'accès en lecture et en écriture aux 32 registres de 32 bits du système. La FIGURE.14 montre le schéma de principe du *banc de registre*, et la SOURCE.5 montre son interface VHDL.

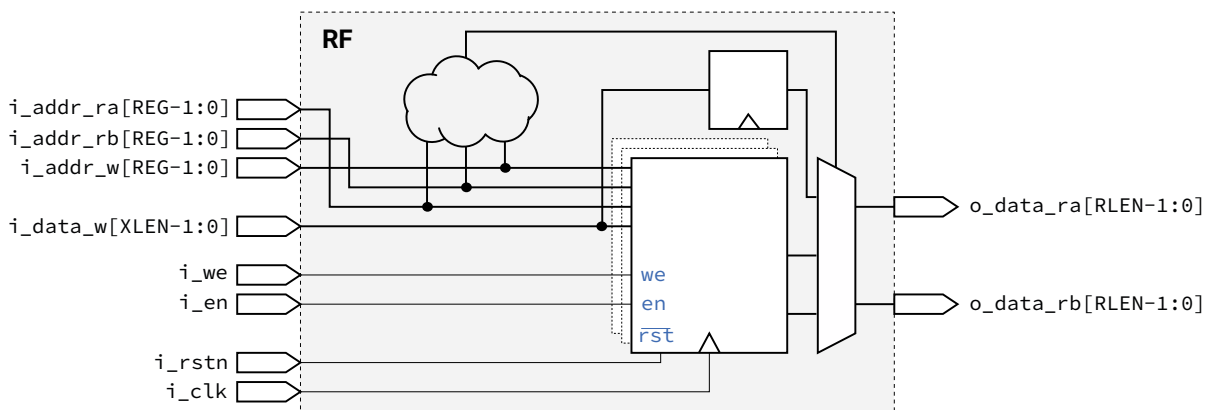


FIGURE. 14 – Module *Banc de Registres* (RF)

```

entity mips_rf is
  port (
    i_rstn    : in  std_logic;
    i_clk     : in  std_logic;
    i_en      : 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(RLEN-1 downto 0);
    i_addr_rb : in  std_logic_vector(REG-1 downto 0);
    o_data_rb : out std_logic_vector(RLEN-1 downto 0);
    i_addr_w  : in  std_logic_vector(REG-1 downto 0);
    i_data_w  : in  std_logic_vector(RLEN-1 downto 0));
end entity mips_rf;

```

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 et si *i_addr_w* est différent de zéro; 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*.

3.1.5 COMPTEUR DE PERFORMANCE

Le rôle du module *Compteur de performance* (PERF) dans le *mini-mips* consiste à compter le nombre de cycles et le nombre d'instructions exécutées par le processeurs. La métrique $CPI = \frac{\#cycles}{\#instructions}$ donne une mesure de la performance de votre processeur.

```

entity mips_perf is
  port (
    i_rstn    : in  std_logic;
    i_clk     : in  std_logic;
    i_en      : in  std_logic;
    o_cycles  : out std_logic_vector(RLEN-1 downto 0);
    o_insts   : out std_logic_vector(RLEN-1 downto 0));
end entity mips_perf;

```

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 *pipeline* 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-mips* est basée sur un pipeline à 4 étages, c'est-à-dire, qui décompose le traitement des instructions en 4 étapes distinctes, tel que l'illustre la FIGURE.15.

- **Fetch (F)** : Lire la prochaine instruction à traiter dans la mémoire d'instruction.
- **Decode (D)** : Déterminer le type d'instruction et lire les opérandes dans le Banc de Registres.
- **Execute (E)** : Appliquer les opérations nécessaires sur les opérandes.
- **Update (U)** : Écrire le résultat dans le Banc de Registres.

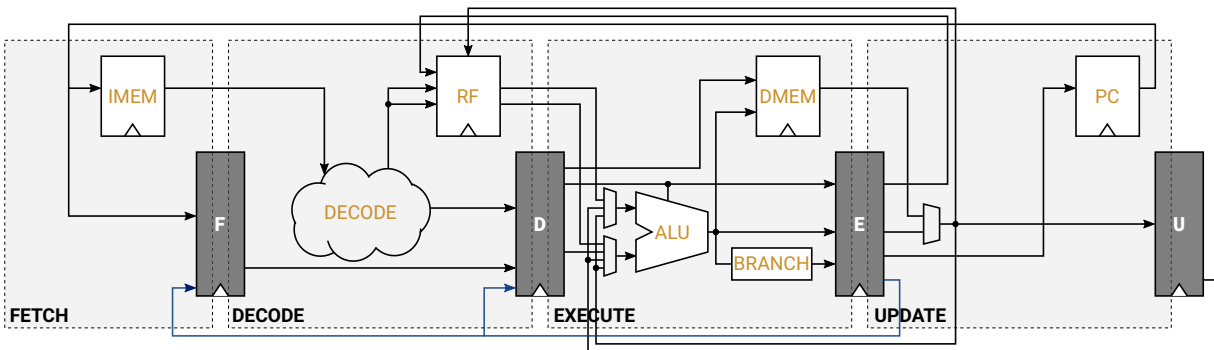


FIGURE. 15 – Pipeline à 4 étages du *mini-mips*

3.2.1 FETCH (F)

À l'étape **F**, le *mini-mips* va chercher une nouvelle instruction dans la mémoire d'instruction. 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 est connectée au 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.

3.2.2 DECODE (D)

Dans l'étape **D**, le *mini-mips* génère un ensemble de signaux de données et de contrôles à partir des informations extraites du mot d'instruction, et produit les opérandes pour l'opération à réaliser. En particulier, cette étape consiste à a :

- Générer les signaux permettant d'identifier l'instruction (*opcode, funct*).
- Générer les signaux d'adresses permettant de lire les opérandes dans le banc de registres (*rs, rt*), ainsi que le signal d'adresse permettant d'écrire dans le banc de registres (*rd*).

- 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, et générer l'ensemble des signaux de contrôles en fonction de l'instruction.

3.2.3 EXECUTE (E)

Dans l'étape **E**, le *mini-mips* 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. En particulier, cet étage est responsable de :

- Générer le résultat des opérations arithmétiques et logiques.
- Générer l'adresse de lecture ou d'écriture dans la mémoire de donnée.
- Lire ou écrire la mémoire de donnée.
- Résoudre la prédiction de branchement.

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 **D**.

3.2.4 UPDATE (U)

Dans l'étape **U**, le *mini-mips* met à jour ses registres d'état (le PC et le banc de registre). En particulier, cet étage est responsable de :

- Écrire le résultat de l'instruction dans le banc de registres à l'adresse pointée par *rd*. Dans le cas d'une opération *load*, la valeur issue de la mémoire de donnée doit être utilisée, dans les autres cas, la valeur issue de l'alu doit être utilisée.
- Mettre à jour le PC à partir des informations sur les sauts et branchements calculés à l'étape **E**.

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*) ou éliminé (*flush*), et certains résultats doivent être transmis entre les différents étages de pipeline (*forward*),

Dans le *mini-mips*, trois catégories de conflits peuvent se présenter : *structurel*, *de donnée*, ou *de contrôle*. La microarchitecture doit mettre en œuvre des mécanismes matériels permettant de prendre en charge ces conflits.

- **Un conflit structurel** apparaît lorsque deux instructions cherchent à accéder à une même ressource au même moment. Dans le *mini-mips*, 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-mips*, ce problème apparaît lorsqu'une instruction cherche à lire un opérande dans le banc de registre (étape **D**) qui n'a pas encore été mis à jour par une instruction précédente (étape **U**). La résolution de la majorité de ces conflits s'effectue par la méthode du *forwarding* : transmettre le résultat d'une instruction des étages **E** et **U** vers l'étage **E**.
- **Un conflit de contrôle** se produit à chaque fois que le compteur programme réalise un saut ou un branchement. Il faut 3 cycles (étage **E**) avant qu'une instruction de ce type 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*).

Le *mini-mips* résout les branchements 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 **E**, 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.