Développement d'un microprocesseur MIPS de type RISC

Encadré par Caroline Lelandais-Perrault

Leonardo BORGES MAFRA MACHADO

Caio Henrique COSTA SOUZA

Gif-sur-Yvette

2018

Sommaire

Glossaire

ALU: Arithmetic logic unit

ISA: Instruction set architecture

MIPS: Microprocessor without interlocked pipeline stages.

RISC: Reduced instruction set computer

VHDL: VHSIC Hardware Description Language

VHSIC: Very High Speed Integrated Circuit

Introduction

Le but de ce rapport est décrire la synthèse et l'implémentation d'un microprocesseur de l'architecture MIPS de type RISC. L'architecture MIPS a été choisi car elle est simple et plusieurs nouvelles architectures s'en inspirent.

L'architecture MIPS a été développé par MIPS Computer Systems Inc. au début des années 1980. Elle est encore utilisée aujourd'hui en systèmes embarqués, mais sa période plus connue était dans les années 1990 où elle pourrait être trouvée dans les appareils photo numériques, les routeurs Cisco, les consoles de jeux Sony Playstation 2 et bien d'autres produits utilisés dans la vie quotidienne.

Le travail consistait à étudier l'architecture du processeur choisi, déterminer les instructions à implémenter, formuler un jeu d'instructions (ou ISA) adapté, développer les codes nécessaires en utilisant le langage de description VHDL et le tester dans plusieurs cas afin de garantir son bon fonctionnement.

Dans la suite il sera spécifié toutes les étapes détaillées pour arriver au processeur désiré.

Développement

Le but de cette partie est de donner tous les détails nécessaires pour la compréhension du développement du microprocesseur.

Le développement sera séparé en différentes parties étant le premier le plus important, l'architecture du microprocesseur. Au sein de l'architecture seront définies les opérations que le microprocesseur pourra effectuer et son jeu d'instructions. Ensuite, le processeur sera séparé en plusieurs composants qui seront détaillés individuellement puis joints dans le datapath (ou chemin de données).

Une fois la conception du microprocesseur est terminée, l'étape de développement sera lancée. Le langage de description VHDL a été utilisé pour implémenter chaque composant et les codes respectifs peuvent être trouvés dans l'annexe. Ensuite, les codes ont été testés et les résultats décrits dans section des résultats.

L'architecture

Après l'étude de plusieurs architectures possibles, l'architecture MIPS a été choisie comme base du projet et comme déjà décrit, elle est simple. Cette architecture possède un format d'instruction de taille fixée, l'accès à la mémoire est limité aux instructions de stockage (instruction load) et chargement (instruction store) et toutes les opérations sont effectuées dans les registres du microprocesseur. Cette architecture définit 32 registres de 32 bits chacun (MIPS32). Le tableau 1 montre la convention de registres adoptés.

Pour le jeu d'instructions c'était considéré la possibilité de créer un nouveau jeu, mais à la fin c'était choisi pour rester avec le jeu utilisé dans l'architecture MIPS pour d'éventuelles modifications futures dans le projet, comme l'ajout de pipeline. Garder le même jeu serait plus facile pour les tiers de modifier ce qui a déjà été fait en raison de la familiarité avec l'architecture MIPS.

Donc le jeu d'instructions choisi a une taille de 32 bits et il a 3 formats possibles, R-type utilisé principalement pour l'arithmétique, I-type utilisé pour la branche et immédiat, et J-type pour les sauts. Le tableau 2 montre en détail ces formats.

Nom	Nombre	Utilisation
$zero	0	La valeur constante 0
$at	1	Assembleur Temporaire
$v0-$v1	2-3	Valeurs pour les résultats de fonction et l'évaluation d'expression
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaires
$s0-$s7	16-23	Temporaires enregistrés
$t8-$t9	24-25	Temporaires
$k0-$k1	26-27	Réservé à OS Kernel
$gp	28	Pointeur global
$sp	29	Pointeur de pile
$fp	30	Pointeur de cadre
$ra	31	Adresse de retour

Tableau 1 - Convention de registres MIPS

Format	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Commentaires
R	op	rs	rt	rd	shamt	funct	Arithmétique
I	op	rs	rt	adresse / immédiate			Branche, immédiate
J	op	adresse					Saut

Tableau 2 - Formats d'instructions

Le rs est le registre de source, le rt le registre cible (register target), le rd le registre de destination, le shamt (cela veut dire shift amount) le montant qui sera décalé dans une opération de décalage et pour différencier les 3 formats possibles il suffit savoir l'opcode.

À cause de la complexité, il a été décidé d'implémenter un sous-ensemble du jeu d'instructions et aussi de ne pas utiliser du pipelining. Le tableau 3 montre les instructions choisies et l'annexe A montre en détails la syntaxe, la description, l'opération et le code pour toutes les instruction.

Le microprocesseur est divisé en deux grandes parties, d'un côté le chemin de données qui est composé pour le banc de registres, les mémoires et l'unité arithmétique, et de l'autre le contrôleur qui est une machine d'état qui génère les signaux de contrôle nécessaires dans le chemin de données.

	Mnem	OPcode	Funct	
R-type	sll $d,$t,a	000000	000000	$d = $t << a
	srl $d,$t,a	000000	000010	$d = $t >> a
	jr $s	000000	001000	PC = $s
	add $d,$s,$t	000000	100000	$d = $s + $t
	sub $d,$s,$t	000000	100010	$d = $s - $t
	and $d,$s,$t	000000	100100	$d = $s and $t
	or $d,$s,$t	000000	100101	$d = $s or $t
	xor $d,$s,$t	000000	100110	$d = $s nor $t
	nor $d,$s,$t	000000	100111	$d = $s xor $t
	slt $d,$s,$t	000000	101010	$d = 1 if $s < $t else $d = 0
	mult $d,$s,$t	000000	011000	$d = $s * $t
	div $d,$s,$t	000000	011010	$d = $s / $t
I-type	beq $s,$t,label	000100		PC = PC+4+label if $t==$d else PC = PC+4
	bne $s,$t,label	000101		PC = PC+4+label if $t!=$d else PC = PC+4
	addi $t,$s,i	001000		$t = $s + i
	slti $t,$s,i	001010		$t = 1 if $s < i else $t = 0
	andi $t,$s,i	001100		$t = $s and i
	ori $t,$s,i	001101		$t = $s xor i
	xori $t,$s,i	001110		$t = $s + i
	lb $t,i($s)	100000		$t = signExtend(mem[$s+i])
	lw $t,i($s)	100011		$t = mem[$s+i]
	lbu $t,i($s)	100100		$t= unsignedExtend(mem[$s+i])
	sw $t,i($s)	101011		mem[$s+i] = $t
	sb $t,i($s)	101000		mem[$s+i] = $t(7..0)
J-type	j label	000010		PC = PC(31..28) & label << 2
	jal label	000011		PC = PC(31..28) & label << 2 $ra = PC+4

Tableau 3 - Jeu d'Instructions

L'Implémentation

L'approche utilisée pour concevoir le datapath a consisté à détailler le jeu d'instructions pour savoir quelles étaient les composants nécessaires dans le chemin de données pour exécuter toutes les instructions. Ensuite le design du datapath a été fait en reliant ses composants et cela a permis savoir quelles étaient les signaux du contrôleur nécessaires pour contrôler le flux des données.

Et puis la machine d'état du contrôleur était faite en prenant compte quels signaux doivent être activés pour exécuter chaque instruction.

L'image 1 montre le désigne de niveau supérieur avec les connexions entre l'unité de contrôle, le chemin de données et le dehors du microprocesseur.

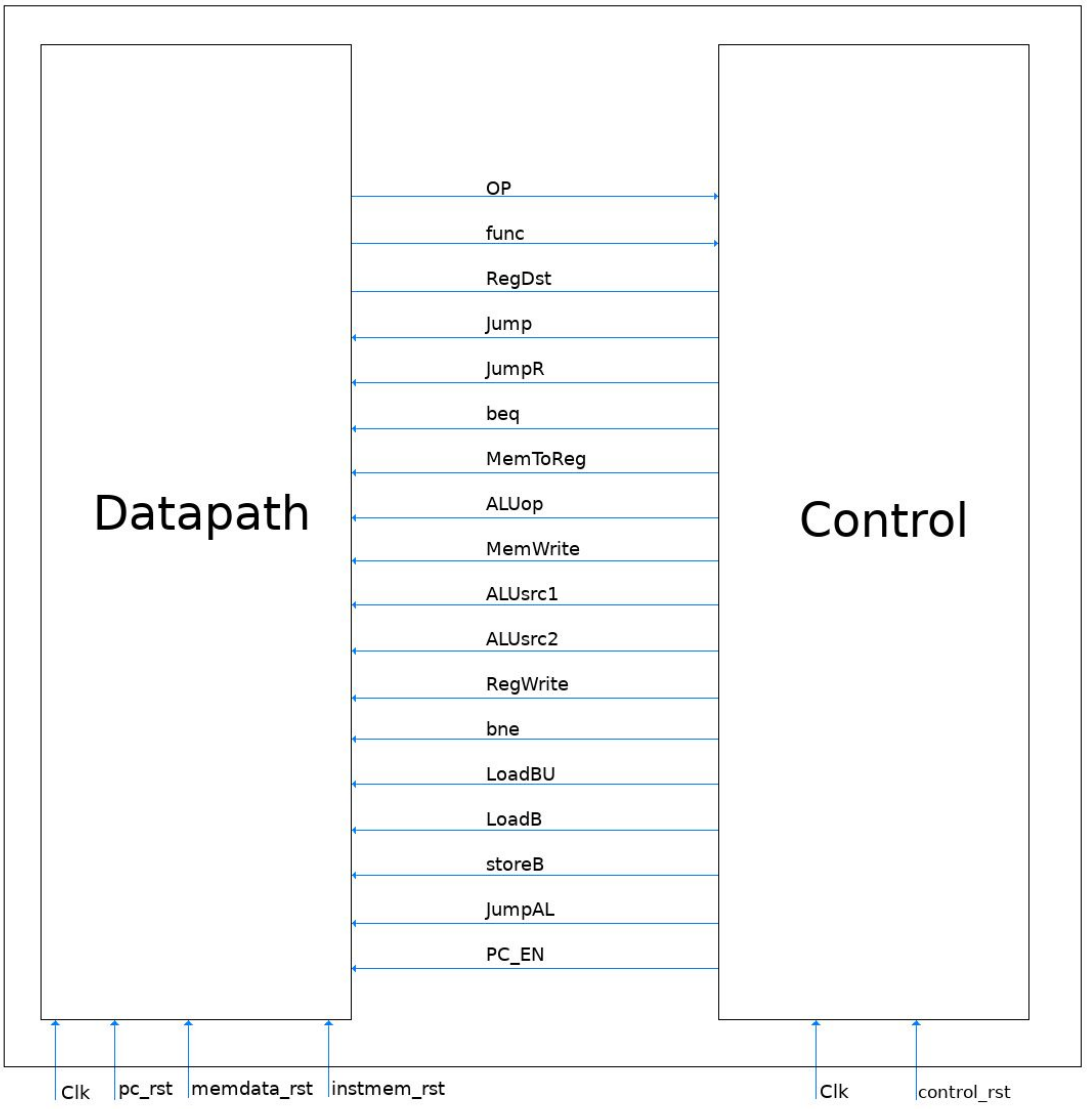

Image 1 - Désigne microprocesseur

Le chemin de données

Le chemin de données est dans l'image 2, ses entrées sont toutes les signaux de contrôle et les deux signaux de sortie sont les bits du OP code et les bits de la fonction de l'instruction actuel, c'est à dire les 6 premières bits et les 6 dernières respectivement de l'instruction. Le datapath relie quatre entités: l'ALU, la mémoire (de données), la mémoire d'instruction et le banc de registres. Son code est dans l'annexe C.

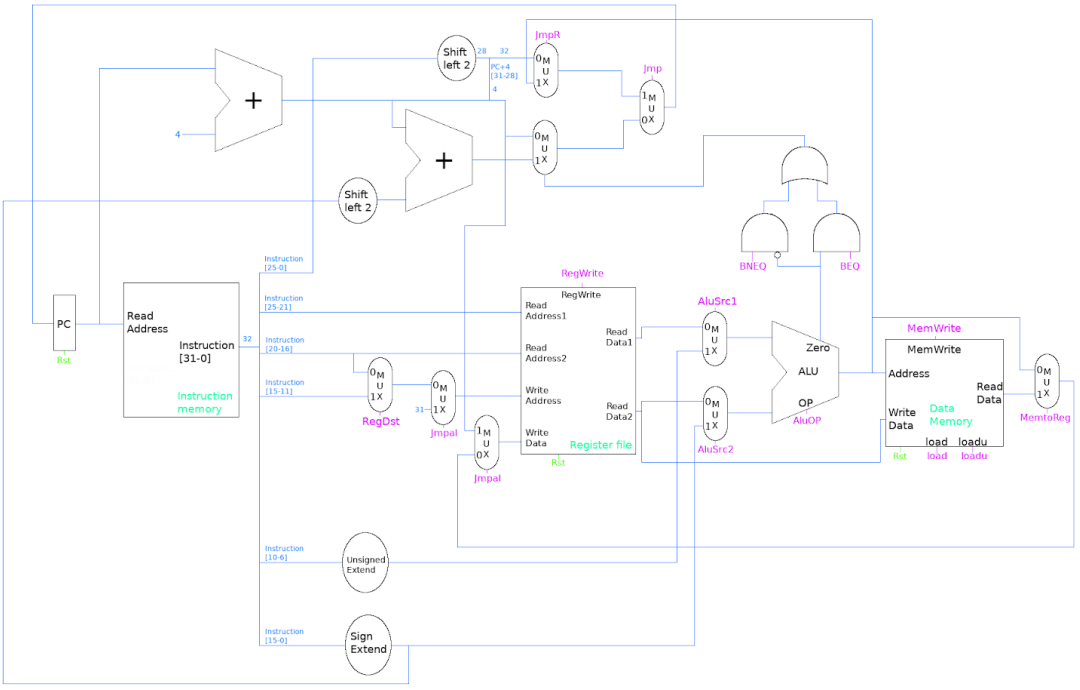

Image 2 - Désigne chemin de donnée. en violet sont les signaux qui viennent de l'unité de contrôle et en vert les signaux qui viennent dehors.

La mémoire

La mémoire de données dans l'image 3 permet d'avoir dans la sortie soit un mot aligné à quatre bytes, soit un byte avec une extension de signe, soit un byte avec extension de zéro. Elle permet aussi de stocker soit un byte, soit un mot aligné à quatre byte. L'adressage est en byte et l'adresse est divisé par quatre pour trouver les mots, donc si l'adresse n'est pas aligné, la mémoire prend le mot adressé par la partie entière de la division. Elle a été implémenté avec 128 bytes (32 mots).

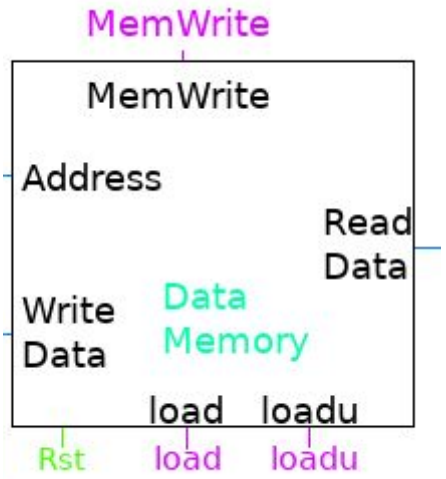

Image 3 - La mémoire de données

Le banc de registres

Le banc de registres dans l'image 4 a 31 registres, de 1 à 31, parce que dans le MIPS32 le contenu du registre 0 est toujours nul. Ce banc montre à la sortie le contenu de deux registre dont l'adresse sont dans le deux entrés d'adresse de lecture. Il a aussi comme entrées l'adresse d'écriture de données, les données qui seront stockés et un signal d'activation de stockage. Le banc de registre est adressé par le nombre du registre, de 0 à 31. Les essais d'écriture sur le registre 0 sont ignorés.

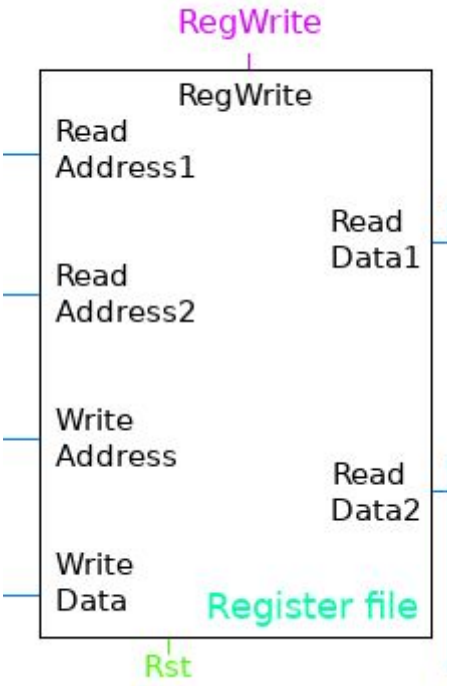

Image 4 - Le register file

La mémoire d'instruction

La mémoire d’instruction qui est montré dans l’image 5 est adressé en byte, la taille est de 128 bytes ou 32 mots, l’entrée est l'adresse de l’instruction et la sortie est le contenu qui est dans cette adresse. Comme la mémoire de données, elle est aligné à quatre bytes donc le reste de la division par quatre est ignoré.

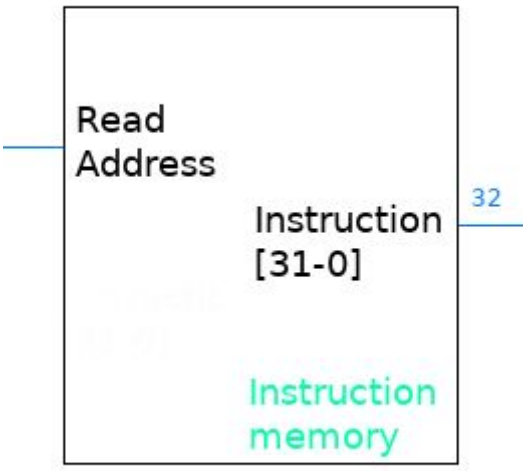

Image 5 - La mémoire d’instruction

L’ALU

L'ALU qui est montré dans l’image 6 a été synthétisé avec les 11 opérations suivantes: somme, soustraction, multiplication, division, décalage gauche logique, décalage droite logique, ensemble inférieur à, et, ou, nor et xor. Toutes ces opérations sont utilisées par le microprocesseur.

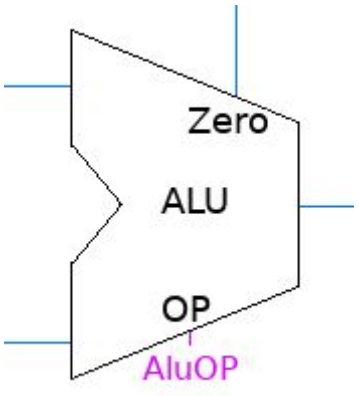

Image 6 - L'ALU

Le contrôleur

Le diagramme d'états se trouve dans l'image 7, une fois que le désigne du chemin de données était prêt, il a été utilisé pour faire les tableaux 4, 5 et 6 qui a permis de concevoir l'unité de contrôle. Ces tableaux contiennent les 5 états de la machine d'état de l'unité de contrôle: fetch, decode, execute, memory et write PC. Par exemple, d'après le tableau les instructions lw, lb et lbu doivent passer pour tous les états, donc elles sont les instructions qui prennent plus de cycles d'horloge, c'est-à-dire 5 cycles d'horloges. D'autre côté, les sautes jr, j et jal sont les instructions qu'utilisent moins cycles d'horloges, 3 cycles qui correspondent aux états fetch, decode et write PC.

C'est recommandé de consulter l'annexe B pour plus de détails concernant le contrôleur.

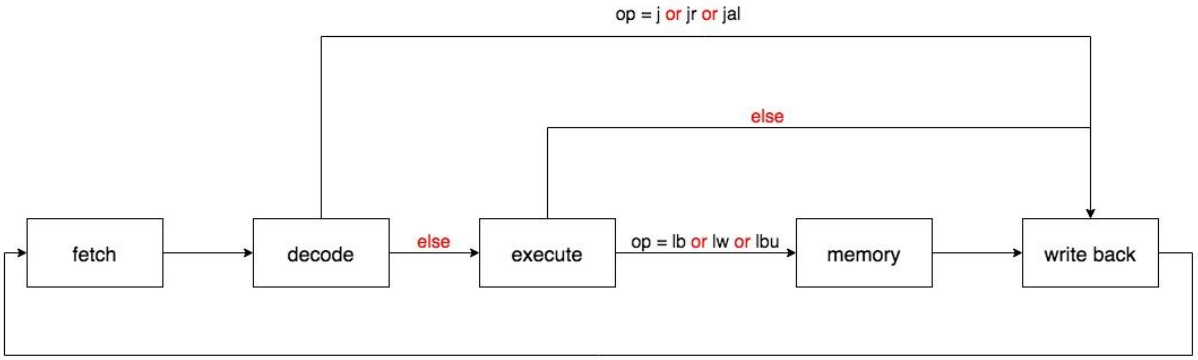

Image 7 - Diagramme états-transitions

		fetch	decode	execute	memory	write PC
J	j			X	X	pc_en jump
	jal			X	X	pc_en RegWrite jump jumpal

Tableau 4 - Diagramme de temps J type

		fetch	decode	execute	memory	write PC
R	add, sub, mult, div, and, or, xor, nor, slt			RegWrite RegDst	X	pc_en
	sll, srl			RegWrite ALUsrc1 RegDst	X	pc_en
	jr			X	X	pc_en jump jumpR

Tableau 5 - Diagramme de temps R type

		fetch	decode	execute	memory	write PC
I	beq				X	pc_en beq
	bne				X	pc_en bne
	addi,andi,ori,xori,slti			RegWrite ALUsrc2	X	pc_en
	lw			ALUsrc2	RegWrite MemToReg	pc_en
	lb			ALUsrc2 loadB	RegWrite MemToReg	pc_en
	lbu			ALUsrc2 loadB loadBU	RegWrite MemToReg	pc_en
	sw			MemWrite ALUsrc2	X	pc_en
	sb			MemWrite ALUsrc2 storeB	X	pc_en

Tableau 6 - Diagramme de temps I type

Assembleur

Un assembleur simple a été écrit en python pour pouvoir écrire des codes en langage assembly et générer le code équivalent en langage machine qui peut être mis dans la mémoire des instructions. Le code du assembleur est dans l'annexe C.

Les tests

L'unité de contrôle a été testée avec toutes les entrées possibles en utilisant un test automatisé et les codes en assembly des trois tests qui étaient faits peuvent être trouvé sur l'annexe C.

Le but du premier test était valider les instructions addi, beq, bne et sw. D'abord addi est utilisé pour mettre les valeurs 4, 5 et 9 dans les registres $t0, $t1 et $t2 respectivement. Puis, 1 est ajouté au registre $t0 ($t0=5), comme les valeurs de $t0 et $t1 sont égaux, beq doit faire un saut vers label, alors 1 est ajouté encore une fois à $t0 ($t0=6), cette fois $t0 est différent de $t1 alors le beq juste incrémente 4 au Program Counter (PC). Ensuite, 1 est ajouté à $t0 ($t0=7), comme $t0 et $t1 sont différent, bne fait un saut vers label. Ce saut fait une fois de plus deux incréments de 1 à $t0 ($t0=9) e le beq juste incrément le PC. Finalement, bne compare $t0 et $t2, comme ils sont égaux, bne juste incrément le PC, la valeur final de $t0 est stocké dans la position 0 de la mémoire et le programme est fini avec un boucle infinie.

Le propos du deuxième test était valider les instructions addi, jal, jr, j, mult, sub, et div. Il consiste à calculer (12/3-2)*7. Alors premièrement addi est utilisé pour mettre les valeurs 12, 3, 2 et 7 dans les registres $t0, $t1, $t2 et $t3 respectivement, donc le but est de calculer ($t0/$t1-$t2)*$t3. Pour valider les instructions de saut, les opérations arithmétiques sont écrits dans un mauvais ordre: produit, soustraction et division, et pour faire le calcul correctement les instructions de saut sont utilisés pour faire les opérations dans le bon ordre. D'abord jal est utilisé pour faire la division, puis jr est utilisé pour faire la soustraction, ensuite j est utilisé pour faire le produit. Le programme termine avec un boucle infinie.

Le troisième test vise valider les instructions addi, sb, lb, sw, or, xor, andi, and, ori, srl et sll. Il utilise les opérations logiques pour manipuler des bits, si les

instructions fonctionnent bien, à la fin les trois premiers mots de la mémoire doivent être "11111111111111111010101111001101", ou "FFFFABCD" en hexadécimal," "01001101010010010000000000000000" ou "MI" en ASCII et “01001101010010010101000001010011" ou "MIPS" en ASCII. Le programme termine avec un boucle infinie.

Résultats

Les trois tests étaient compilé avec l'assembleur en python et étaient mis dans la mémoire d'instruction, une inspection des formes d'ondes sur ModelSim montre que les trois programme de test étaient bien exécuté. Dans l'image 8 la valeur final de $t0 est 9, dans l'image 9 le résultat du calcul (12/3-2)*7=14 est stocké sur $t6 et dans l'image 10 les trois premiers mots de la mémoire de données est montré, le premier représenté en hexadécimal (FFFFABCD) et le deux suivant en ASCII (MI, MIPS).

Image 8 - Test 1

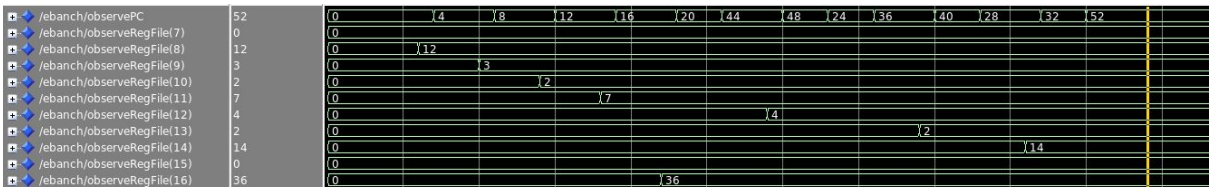

Image 9 - Test 2

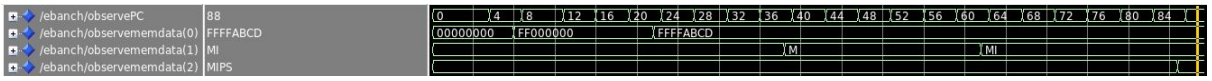

Image 10 - Test 3

Le processus de compilation en quartus donne une fréquence d'horloge maximale et cette valeur serait de 350 MHz en considérant les mémoires comme des composants externes.

Conclusion

Le microprocesseur a été développé en suivant les spécifications de l'architecture de MIPS32 après une période d'étude intensive et de discussion sur lequel choisir. La planification a été faite et suivie avec réunions hebdomadaires avec le professeur responsable qui a toujours donné quelques conseils et répondu aux doutes possibles.

Ensuite la conception de l'architecture a été développée, c'est-à-dire, le diagramme d'état de la machine, la vue d'ensemble des données et le chronogramme de tous les signaux de contrôle.

Dans la suite tout le code a été fait et testé en utilisant les méthodes spécifiées par le professeur et à la fin le projet était terminé. De nombreux scénarios différents ont été proposés et le microprocesseur a pu donner la bonne sortie dans chacun d'entre eux.

Le projet a également permis l'amélioration de diverses compétences des élèves telles que la gestion de projet, l'analyse des spécifications, les bonnes pratiques de codage, le test de la boîte blanche, le test de la boîte noire et la rédaction de rapports.

Bibliographie

[1] - Patterson, David A., and John L. Hennessy. Computer Organization and Design the Hardware-Software Interface. 5th ed., Morgan Kaufmann, 2014.

[2] - MIPS Reference Data Card (“Green Card”)

[3]- Rubio, Victor P. "A FPGA Implementation of a MIPS RISC Processor for Computer Architecture Education". New Mexico State University.

[4] - MIPS Technologies Inc. “MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set”. Revision 2.62, January 2, 2009.

[5] - http://people.cs.pitt.edu/~xujie/cs447/MIPS_Instruction.htm

Annexe A - Instructions détaillées

Instruction: sll
Syntaxe: sll $d, $t, amm
Description: Met en $d le contenu de $t décalé par amm à gauche, ajoute 4 au PC
Opération: $d = $t << amm; PC = PC+4
Code: 000000xxxxxddddddttttttaaaaa000000

Instruction: srl
Syntaxe: srl $d, $t, amm
Description: Met en $d le contenu de $t décalé par amm à droite, ajoute 4 au PC
Opération: $d = $t >> amm; PC = PC+4
Code: 000000xxxxxddddddttttttaaaaa000010

Instruction: jr
Syntaxe: jr $s
Description: Saute vers l'adresse qui est dans $s
Opération: PC = $s
Code: 000000ssssssxxxxxxxxxxxxxxxx000010

Instruction: add
Syntaxe: add $d, $s, $t
Description: Met en $d le contenu de $s plus le contenu de $t, ajoute 4 au PC
Opération: $d = $s + $t; PC = PC+4
Code: 000000ssssssddddddttttttxxxxx100000

Instruction: sub
Syntaxe: sub $d, $s, $t
Description: Met en $d le contenu de $s moins le contenu de $t, ajoute 4 au PC
Opération: $d = $s - $t; PC = PC+4
Code: 000000ssssssddddddttttttxxxxx100010

Instruction: and

Syntaxe: and $d, $s, $t

Description: Met en $d le contenu de $s et logique avec le contenu de $t, ajoute 4 au PC

Opération: $d = $s AND $t; PC = PC+4

Code: 000000ssssssddddddttttttxxxxx100100

Instruction: or

Syntaxe: or $d, $s, $t

Description: Met en $d le contenu de $s ou logique avec le contenu de $t, ajoute 4 au PC

Opération: $d = $s OR $t; PC = PC+4

Code: 000000ssssssddddddttttttxxxxx100101

Instruction: xor

Syntaxe: xor $d, $s, $t

Description: Met en $d le contenu de $s ou exclusif avec le contenu de $t, ajoute 4 au PC

Opération: $d = $s XOR $t; PC = PC+4

Code: 000000ssssssddddddttttttxxxxx100110

Instruction: nor

Syntaxe: nor $d, $s, $t

Description: Met en $d le contenu de $s non-ou logique avec le contenu de $t, ajoute 4 au PC

Opération: $d = $s NOR $t; PC = PC+4

Code: 000000ssssssddddddttttttxxxxx100111

Instruction: slt

Syntaxe: slt $d, $s, $t

Description: Met 1 en $d si le contenu de $s est plus petite que le contenu de $t, sinon met 0 en $d, ajoute 4 au PC

Opération: $d = 1 if $s < $t else 0; PC = PC+4

Code: 000000sssssddddddttttttxxxxx101010

Instruction: mult

Syntaxe: mult $d, $s, $t

Description: Met en $d les 32 bits de poids faible du produit du contenu de $s et le contenu de $t, ajoute 4 au PC

Opération: $d = $s * $d; PC = PC+4

Code: 000000sssssddddddttttttxxxxx011000

Instruction: div

Syntaxe: div $d, $s, $t

Description: Met en $d les 32 bits de poids faible de la partie entière de la division du contenu de $s par le contenu de $t, ajoute 4 au PC

Opération: $d = $s / $d; PC = PC+4

Code: 000000sssssddddddttttttxxxxx011010

Instruction: beq

Syntaxe: beq $s, $t, label

Description: Saute vers l'immédiat décalé à droite par 2 plus l'adresse de l'instruction qui suive le beq si le contenu de $s est égale au contenu de $t, sinon ajoute 4 au PC

Opération: PC = PC+4+(imm>>2) if $s==$t else PC+4

Code: 000100ssssstttttiiiiiiiiiiiiiiii

Instruction: bne

Syntaxe: bne $s, $t, label

Description: Saute vers l'immédiat décalé à droite par 2 plus l'adresse de l'instruction qui suive le bne si le contenu de $s est différent du contenu de $t, sinon ajoute 4 au PC

Opération: PC = PC+4+(imm>>2) if $s!=$t else PC+4

Code: 000101ssssstttttiiiiiiiiiiiiiiii

Instruction: addi

Syntaxe: addi $t, $s, imm

Description: Met en $t le contenu de $s plus l'immédiat, ajoute 4 au PC

Opération: $t=$s+imm; PC+4

Code: 001000ssssstttttiiiiiiiiiiiiiiii

Instruction: slti

Syntaxe: slti $t, $s, imm

Description: Met 1 en $t si le contenu de $s est plus petite que l'immédiat, sinon met 0 en $t, ajoute 4 au PC

Opération: $t = 1 if $s < imm else 0; PC = PC+4

Code: 001010ssssstttttiiiiiiiiiiiiiiii

Instruction: andi

Syntaxe: andi $t, $s, imm

Description: Met en $t le contenu de $s et logique avec l'immédiat, ajoute 4 au PC

Opération: $t = $s AND imm; PC = PC+4

Code: 001100ssssstttttiiiiiiiiiiiiiiii

Instruction: ori

Syntaxe: ori $t, $s, imm

Description: Met en $t le contenu de $s ou logique avec l'immédiat, ajoute 4 au PC

Opération: $t = $s OR imm; PC = PC+4

Code: 001101ssssstttttiiiiiiiiiiiiiiii

Instruction: xori

Syntaxe: xori $t, $s, imm

Description: Met en $t le contenu de $s ou exclusif avec l'immédiat, ajoute 4 au PC

Opération: $t = $s XOR imm; PC = PC+4

Code: 001110ssssstttttiiiiiiiiiiiiiiii

Instruction: lb

Syntaxe: lb $t, imm($s)

Description: Met en $t le contenu qui est dans le byte de l'adresse $s plus l'immédiat de la mémoire en faisant une extension de signe, ajoute 4 au PC

Opération: $t = signed_extend(MEM[imm+$s]); PC = PC+4

Code: 100000ssssstttttiiiiiiiiiiiiiiii

Instruction: lw

Syntaxe: lw $t, imm($s)

Description: Met en $t le contenu qui est dans le mot qui commence à l'adresse $s plus l'immédiat de la mémoire, ajoute 4 au PC

Opération: $t = MEM[imm+$s]; PC = PC+4

Code: 100011ssssstttttiiiiiiiiiiiiiiii

Instruction: lbu

Syntaxe: lbu $t, imm($s)

Description: Met en $t le contenu qui est dans le byte de l'adresse $s plus l'immédiat de la mémoire en faisant une extension avec zéro, ajoute 4 au PC

Opération: $t = unsigned_extend(MEM[imm+$s]); PC = PC+4

Code: 100100ssssstttttiiiiiiiiiiiiiiii

Instruction: sw

Syntaxe: sw $t, imm($s)

Description: Met dans le mot de la mémoire qui commence à l'adresse $s plus l'immédiat le contenu de $t, ajoute 4 au PC

Opération: MEM[imm+$s] = $t; PC = PC+4

Code: 101011ssssstttttiiiiiiiiiiiiiiii

Instruction: sb

Syntaxe: sb $t, imm($s)

Description: Met dans le byte d'adresse $s plus l'immédiat le contenu qui est dans le byte plus faible de $t, ajoute 4 au PC

Opération: MEM[imm+$s] = $t; PC = PC+4

Code: 101000ssssstttttiiiiiiiiiiiiiiii

Instruction: j

Syntaxe: j label

Description: Met en PC les quatre bits plus fort de PC concaténé à l'immédiat décalé deux byte à gauche

Opération: PC = PC(31..28) & label << 2

Code: 000010iiiiiiiiiiiiiiiiiiiiiiiiii

Instruction: jal

Syntaxe: jal label

Description: Met en PC les quatre bits plus fort de PC concaténé à l'immédiat décalé deux byte à gauche; met en $ra la valeur de PC plus quatre

Opération: PC = PC(31..28) & label << 2; $ra = PC+4

Code: 000011iiiiiiiiiiiiiiiiiiiiiiiiii

Annexe B - Diagrammes de temps du contrôleur

Dans l'image B1 il y a le diagramme de temps de l'instruction add et dans l'image B2 le diagramme de temps de l'instruction addi, il est intéressant à noter que dans l'état d'exécution, le signal RegWrite est égal à 1, ce qui signifie que la valeur calculée par l'ALU sera enregistrée dans le banc de registres. Un autre point intéressant à noter est que dans les instructions avec immédiat le signal ALUsrc2 sera égal à 1 car la valeur (entré B de l'ALU) va venir directement de l'instruction décodifiée et pas du banc de registres.

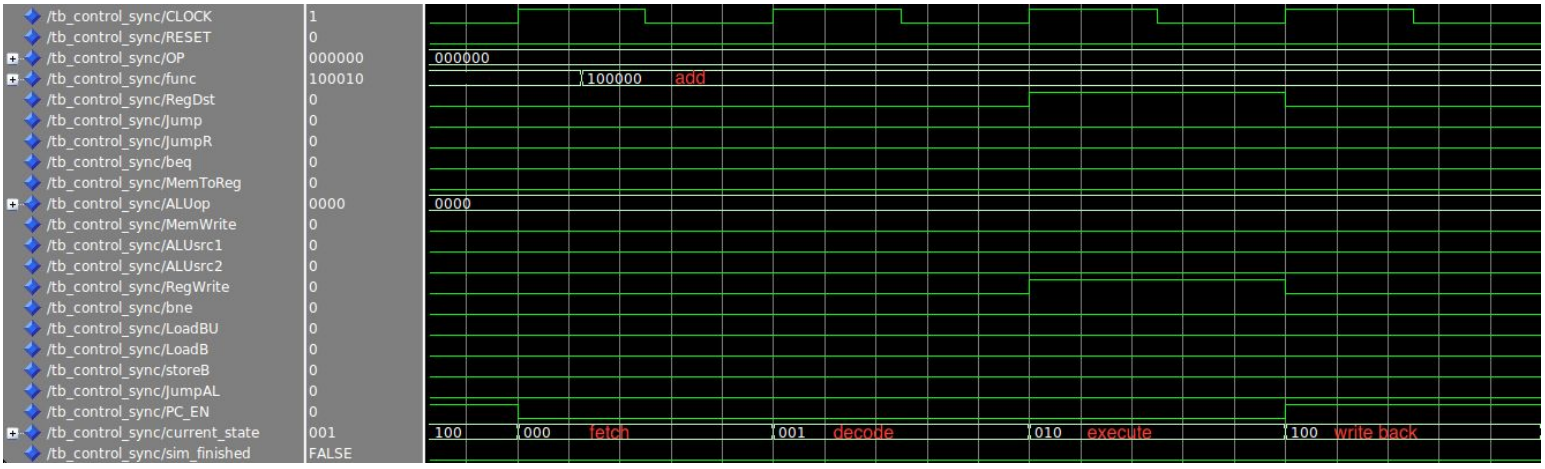

Image B1: Diagramme de temps de l'instruction add

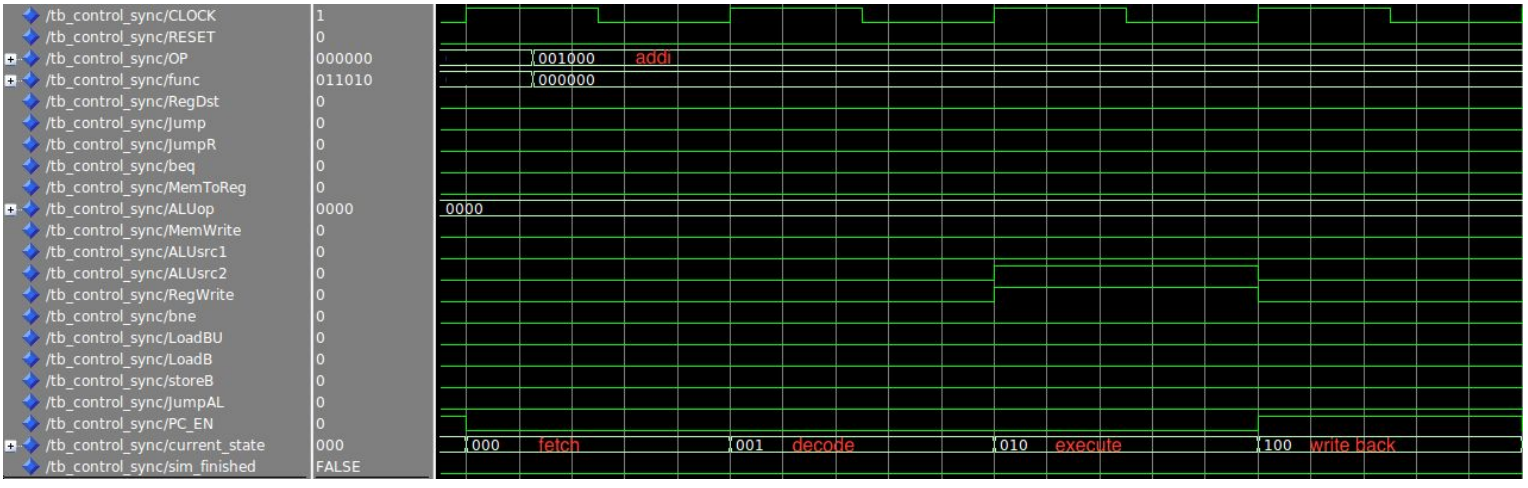

Image B2: Diagramme de temps de l'instruction addi

Dans l'image B3 c'est possible de voir le diagramme de temps de l'instruction décalage gauche, il est intéressant de noter que les instructions de décalage logique sont les seules instructions qui mettent le signal ALUsrc1 égal à 1 car le shamt (voir tableau 2) sera utilisé. L'image B4 montre le diagramme de temps de l'instruction jump, les instructions de saut prennent seulement trois cycles d'horloge.

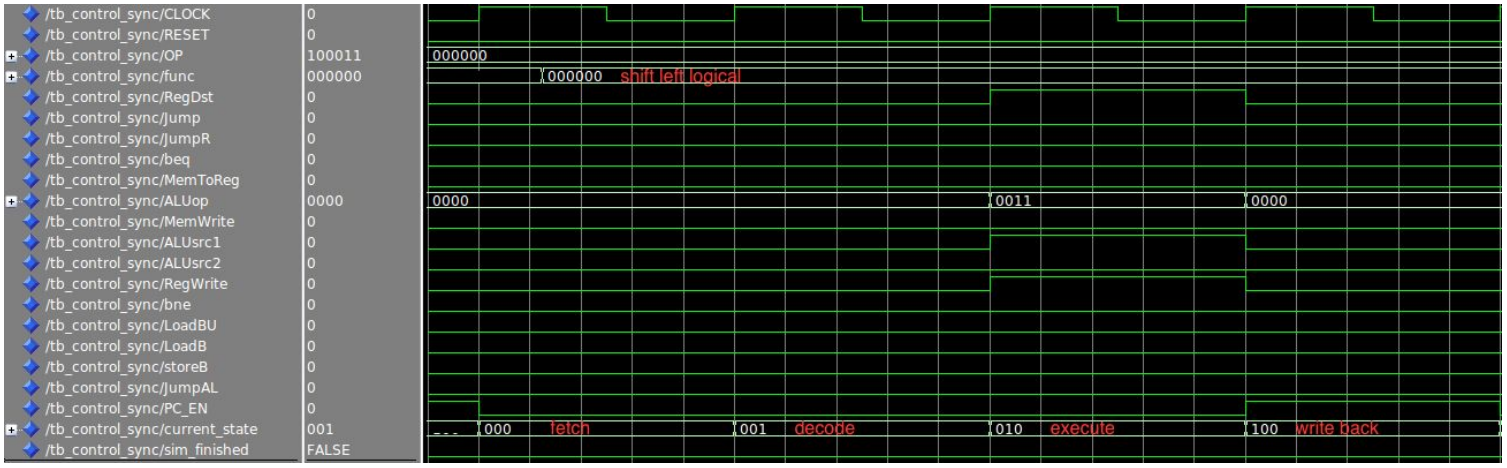

Image B3: Diagramme de temps de l'instruction shift left logical (sll)

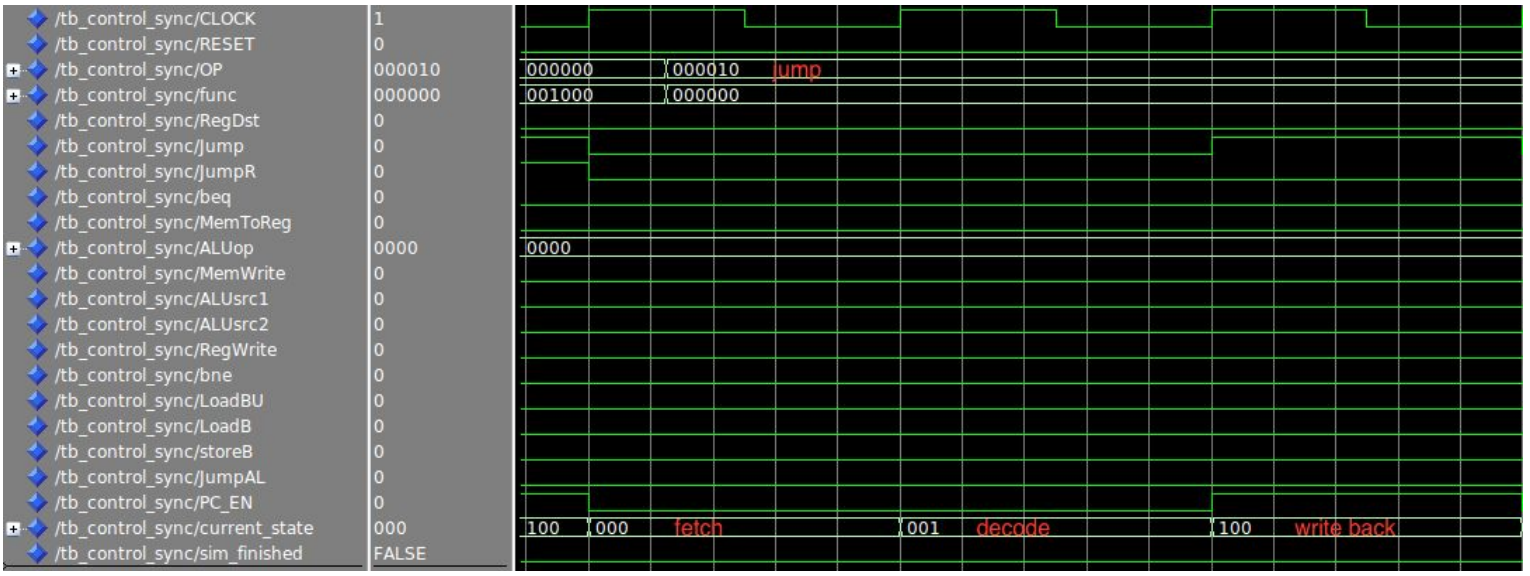

Image B4: Diagramme de temps de l'instruction jump (j)

C’est possible de voir dans l’image B5 le diagramme de temps de l'instruction de stockage, l’état d'exécution fait le calcul de l’adresse où sera enregistré le donné et le signal MemWrite est mis à 1. Dans l’image B6 il y a le diagramme de temps de l'instruction de chargement qui prend 5 cycles d’horloge et donc ces instructions (load word, load byte et load byte unsigned) sont ceux qui prennent plus de temps.

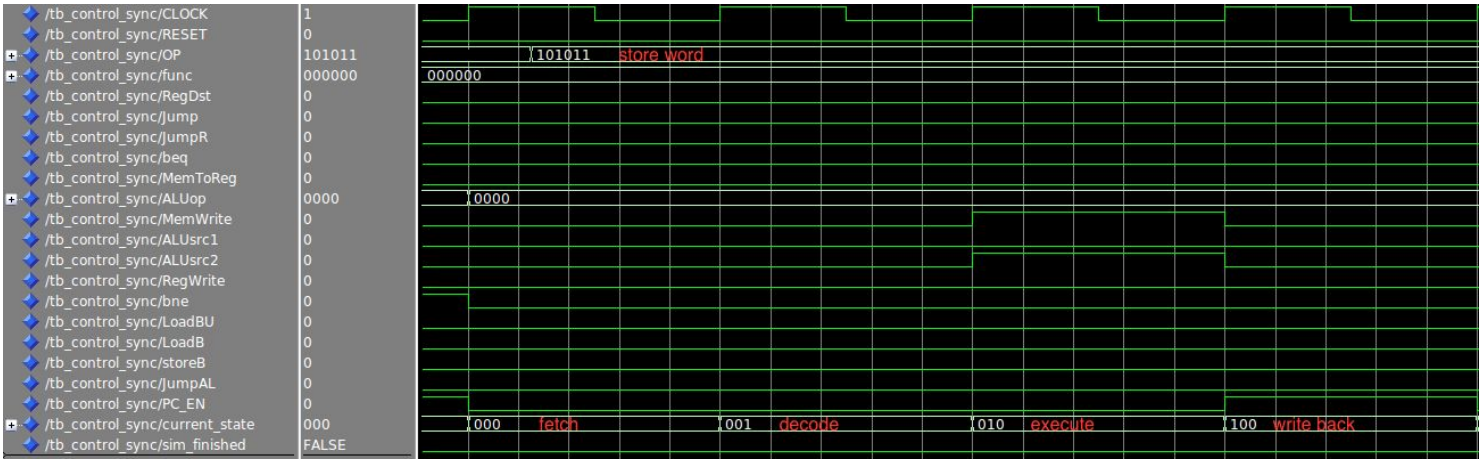

Image B5: Diagramme de temps de l'instruction store word (sw)

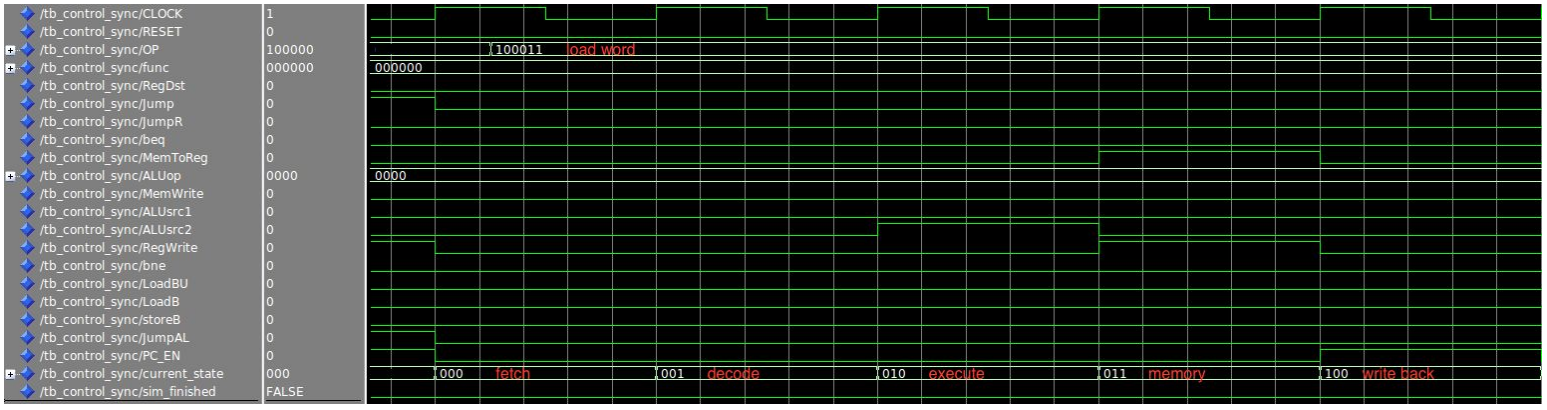

Image B6: Diagramme de temps de l'instruction load word (lw)

Annexe C - Codes

Il est possible de trouver tous les codes ci-dessous.

Microprocesseur

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity eMicroprocessor is
    port (
        signal micro_clk, control_rst, regfile_rst,
                pc_rst, memdata_rst : in std_logic
    );
end entity eMicroprocessor;

architecture aMicroprocessor of eMicroprocessor is
    component eDatapath is
    port (
            signal Jmp, JmpR, BEQ, BNEQ, RegDst,
                    Jmpal, RegWrite, MemWrite : in std_logic;
            signal clk, AluSrc1, AluSrc2, loadb, loadbu,
                    MemtoReg, storeb, pc_en : in std_logic;
            signal regfile_rst, pc_rst, memdata_rst : in std_logic;
            signal AluOP : in std_logic_vector (3 downto 0);
            signal OPcode, funct : out std_logic_vector (5 downto 0)
    );
    end component eDatapath;

    component control_sync is
    port (
        CLOCK    : in  std_logic;
        RESET    : in  std_logic;
        OP       : in  std_logic_vector(5 downto 0);  -- opcode
        func     : in  std_logic_vector(5 downto 0);  -- func code
        RegDst   : inout std_logic;  -- Register destination
        Jump     : inout std_logic;
        JumpR    : inout std_logic;  -- Jump register
        beq      : inout std_logic;  -- Branch if equals
        MemToReg : inout std_logic;  -- Memory to register
        ALUop    : inout std_logic_vector(3 downto 0);  -- ALU op code
        MemWrite : inout std_logic;  -- Memory write
        ALUsrc1  : inout std_logic;
        ALUsrc2  : inout std_logic;
        RegWrite : inout std_logic;  -- Register write
        bne      : inout std_logic;  -- Branch if not equals
        LoadBU   : inout std_logic;  -- Load byte unsigned
        LoadB    : inout std_logic;  -- Load byte
```

```
        storeB   : inout std_logic;
        JumpAL   : inout std_logic; -- Jump and link
        PC_EN    : inout std_logic;
        current_state : out std_logic_vector(2 downto 0)
    );
    end component control_sync;

    signal micro_Jmp, micro_JmpR, micro_BEQ, micro_BNEQ,
             micro_RegDst, micro_Jmpal : std_logic;
    signal micro_AluSrc1, micro_AluSrc2, micro_loadb,
             micro_loadbu : std_logic;
    signal micro_MemtoReg, micro_storeb, micro_pc_en,
             micro_RegWrite, micro_MemWrite : std_logic;
    signal micro_AluOP :  std_logic_vector (3 downto 0);
    signal micro_OPcode, micro_funct :  std_logic_vector (5 downto 0);

    begin
    instDapath : eDatapath
    port map(Jmp => micro_Jmp, JmpR => micro_JmpR, BEQ => micro_BEQ,
                BNEQ => micro_BNEQ, RegDst => micro_RegDst, Jmpal => micro_Jmpal,
                RegWrite => micro_RegWrite, MemWrite => micro_MemWrite,
                clk => micro_clk, AluSrc1 => micro_AluSrc1,
                AluSrc2 => micro_AluSrc2, OPcode => micro_OPcode,
                loadb => micro_loadb, loadbu => micro_loadbu,
                MemtoReg => micro_MemtoReg, funct => micro_funct,
                AluOP => micro_AluOP, storeb => micro_storeb,
                pc_en => micro_pc_en, regfile_rst => regfile_rst,
                pc_rst => pc_rst, memdata_rst => memdata_rst);

    instControl : control_sync
    port map(
    CLOCK => micro_clk,RESET => control_rst,OP => micro_OPcode,
    func => micro_funct,RegDst => micro_RegDst,
    beq => micro_BEQ, MemToReg => micro_MemtoReg,
    ALUop => micro_AluOP, MemWrite => micro_MemWrite,
    ALUsrc2 => micro_AluSrc2,RegWrite => micro_RegWrite,
    bne => micro_BNEQ,LoadBU => micro_loadbu,
    LoadB => micro_loadb,storeB => micro_storeb,
    JumpAL => micro_Jmpal,PC_EN => micro_pc_en,
    ALUsrc1 => micro_AluSrc1, Jump => micro_Jmp,
    JumpR => micro_JmpR);

end architecture aMicroprocessor;
```

Datapath

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity eDatapath is
    port (
            signal Jmp, JmpR, BEQ, BNEQ, RegDst, Jmpal,
                     RegWrite, MemWrite : in std_logic;
            signal clk, AluSrc1, AluSrc2, loadb, loadbu,
                     MemtoReg, storeb, pc_en : in std_logic;
            signal regfile_rst, pc_rst, memdata_rst : in std_logic;
            signal AluOP : in std_logic_vector (3 downto 0);
            signal OPcode, funct : out std_logic_vector (5 downto 0)
    );
end entity eDatapath;

use work.SpyOnMySigPkg.all;

architecture aDatapath of eDatapath is
signal RegFile_WriteData, RegFile_ReadData1 : std_logic_vector (31 downto 0);
signal RegFile_ReadAddr1, RegFile_ReadAddr2 : std_logic_vector (4 downto 0);
signal RegFile_WriteAddr : std_logic_vector (4 downto 0);
signal Mem_Address, InstMem_ReadAddr : std_logic_vector (6 downto 0);
signal Mem_WriteData, Mem_ReadData,
         RegFile_ReadData2 : std_logic_vector (31 downto 0);
signal ALU_A, ALU_B, ALU_Y : std_logic_vector (31 downto 0);
signal ALU_OP : std_logic_vector (3 downto 0);
signal ALU_Z : std_logic;
signal PC : std_logic_vector (31 downto 0) := (others => '0');
signal PCplus4 : std_logic_vector (31 downto 0) := x"00000004";
signal InstMem_Instruction31_0 : std_logic_vector (31 downto 0);
signal RegFiletmp_ReadAddr1, RegFiletmp_ReadAddr2,
         RegFiletmp_WriteAddr : INTEGER range 0 TO 31;

component eInstMem
    port (
        signal ReadAddr : in std_logic_vector (6 downto 0);
        signal clk : in std_logic;
        signal Instruction31_0 : out std_logic_vector (31 downto 0)
    );
end component;

component eRegFile
```

```
    port (
        signal ReadAddr1, ReadAddr2, WriteAddr : in INTEGER range 0 TO 31;
        signal WriteData : in std_logic_vector (31 downto 0);
        signal clk, RegWrite, regfile_rst : in std_logic;
        signal ReadData1, ReadData2 : out std_logic_vector (31 downto 0)
    );
end component;

component eMem
    port (
        signal Address : in std_logic_vector (6 downto 0);
        signal WriteData : in std_logic_vector (31 downto 0);
        signal clk, MemWrite, loadb,
                loadbu, storeb, memdata_rst : in std_logic;
        signal ReadData : out std_logic_vector (31 downto 0)
    );
end component;

component ALU
    port (
        A  : in  signed(31 downto 0);   -- operand A
        B  : in  signed(31 downto 0);   -- operand B
        OP : in  std_logic_vector(3 downto 0); -- opcode
        Y  : out signed(31 downto 0);  -- operation result
        Z  : out std_logic -- zero
    );
end component;

begin

process (PC) begin
    GlobalPC <= PC;
end process;

RegFiletmp_ReadAddr1 <= to_integer(unsigned(RegFile_ReadAddr1));
RegFiletmp_ReadAddr2 <= to_integer(unsigned(RegFile_ReadAddr2));
RegFiletmp_WriteAddr <= to_integer(unsigned(RegFile_WriteAddr));
InstMem : eInstMem
port map(clk => clk, ReadAddr => InstMem_ReadAddr,
            Instruction31_0 => InstMem_Instruction31_0);
RegFile : eRegFile
port map(ReadAddr1 => RegFiletmp_ReadAddr1, regfile_rst => regfile_rst,
    ReadAddr2 => RegFiletmp_ReadAddr2, clk => clk,
    WriteAddr => RegFiletmp_WriteAddr, WriteData => RegFile_WriteData,
    ReadData1 => RegFile_ReadData1, ReadData2 => RegFile_ReadData2,
    RegWrite => RegWrite);
Mem : eMem
port map(Address => Mem_Address, WriteData => Mem_WriteData,
```

```
            ReadData => Mem_ReadData, memdata_rst => memdata_rst,
            clk => clk, MemWrite => MemWrite, loadb => loadb,
            loadbu => loadbu, storeb => storeb);
cALU : ALU
port map(A => signed(ALU_A), B => signed(ALU_B), std_logic_vector(Y) => ALU_Y,
OP => ALU_OP, Z => ALU_Z);
ALU_OP <= AluOP;

RegFile_ReadAddr1 <= InstMem_Instruction31_0(25 downto 21);
RegFile_ReadAddr2 <= InstMem_Instruction31_0(20 downto 16);
RegFile_WriteAddr <= "11111" when Jmpal='1' else
                           InstMem_Instruction31_0(15 downto 11) when RegDst='1'
                           else InstMem_Instruction31_0(20 downto 16);
RegFile_WriteData <= PCplus4 when Jmpal='1' else
    Mem_ReadData when MemtoReg='1' else
    ALU_Y;

ALU_A <= (31 downto 5 => '0')
            & InstMem_Instruction31_0(10 downto 6) when AluSrc1='1' else
            RegFile_ReadData1;
ALU_B <= (31 downto 16 => InstMem_Instruction31_0(15))
            & InstMem_Instruction31_0(15 downto 0) when AluSrc2='1' else
            RegFile_ReadData2;

Mem_Address <= ALU_Y(6 downto 0);
Mem_WriteData <= RegFile_ReadData2;

process (clk, pc_rst)
    variable tmp : std_logic_vector (31 downto 0);
    begin
    if pc_rst = '1' then
        PC <= x"00000000";
    elsif rising_edge(clk) then
        if pc_en='1' then
            if Jmp='1' and JmpR='1' then
                PC <= ALU_Y;
            elsif Jmp='1' then
                PC <= PCplus4(31 downto 28)
                        & InstMem_Instruction31_0(25 downto 0) & "00";
            elsif ((BNEQ and not ALU_Z) or (BEQ and ALU_Z))='1' then
                tmp := (31 downto 18 => InstMem_Instruction31_0(15))
                        & InstMem_Instruction31_0(15 downto 0) & "00";
                PC <= std_logic_vector( signed(tmp) + signed(PCplus4) );
            else
                PC <= PCplus4;
            end if;
        elsif pc_en='0' then
        PC <= PC;
```

```
        end if;
        PCplus4 <= std_logic_vector(unsigned(PC)+4);
    end if;
    end process;
    OPcode <= InstMem_Instruction31_0(31 downto 26);
    funct <=  InstMem_Instruction31_0(5 downto 0);
    InstMem_ReadAddr <= PC(6 downto 0);

end architecture aDatapath;
```

Mémoire

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity eMem is
    port (
        signal Address : in std_logic_vector (6 downto 0);
        signal WriteData : in std_logic_vector (31 downto 0);
        signal clk, MemWrite, loadb,
                 loadbu, storeb, memdata_rst : in std_logic;
        signal ReadData : out std_logic_vector (31 downto 0)
    );
end entity eMem;

use work.SpyOnMySigPkg.all;

architecture aMem of eMem is
TYPE mem32w is ARRAY (0 to 31) of std_logic_vector(31 downto 0);
signal mem : mem32w := ((others => (others => '0')));

begin

    process (mem)
    begin
    for i in 0 to 31 loop
            Globalmemdata(i) <= mem(i);
    end loop;
    end process;

    process (clk, memdata_rst)
    variable WordNumb : integer range 0 to 31;
    variable ByteNumb : integer range 0 to 3;
    variable tmp : integer range 0 to 31;
    variable tmp2 : std_logic;
    begin
        if memdata_rst = '1' then
            mem <= ((others => (others => '0')));
        elsif rising_edge(clk) then
            WordNumb := to_integer(unsigned(Address))/4;
            ByteNumb := to_integer(unsigned(Address)) rem 4;
            -- Position of the first bit of the adressed byte
            tmp := 31-ByteNumb*8;
            -- First bit of the adressed byte
```

```
            tmp2 := mem(WordNumb)(tmp);
            if loadb='0' then
                ReadData <= mem(WordNumb);
            elsif loadb='1' and loadbu='1' then -- Zero extends
                ReadData(31 downto 8) <= x"000000";
                ReadData(7 downto 0) <= mem(WordNumb)(tmp downto tmp-7);
            elsif loadb='1' and loadbu='0' then -- Sign extends
                ReadData(31 downto 8) <= (others => tmp2);
                ReadData(7 downto 0) <= mem(WordNumb)(tmp downto tmp-7);
            end if;

            if MemWrite='1' and storeb='1' then
                mem(WordNumb)(tmp downto tmp-7) <= WriteData(7 downto 0);
            elsif MemWrite='1' then
                mem(WordNumb) <= WriteData;
            end if;
        end if;
    end process;
end architecture aMem;
```

Register File

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

ENTITY eRegFile is
    port (
        signal ReadAddr1, ReadAddr2, WriteAddr : in INTEGER range 0 TO 31;
        signal WriteData : in std_logic_vector (31 downto 0);
        signal clk, RegWrite, regfile_rst : in std_logic;
        signal ReadData1, ReadData2 : out std_logic_vector (31 downto 0)
    );
end ENTITY eRegFile;


use work.SpyOnMySigPkg.all;

architecture aRegfile of eRegFile is
type regmem is array (1 TO 31) of std_logic_vector(31 downto 0);
signal RegFile : regmem := ((others => (others => '0')));

begin

    process (RegFile)
    begin
    for i in 1 TO 31 LOOP
        GlobalRegFile(i) <= RegFile(i);
    end LOOP;
    end process;

    process (clk, regfile_rst)
    begin
    IF regfile_rst = '1' then
        RegFile <= ((others => (others => '0')));
    elsif rising_edge(clk) then
        IF ReadAddr1=0 then
            ReadData1 <= x"00000000";
        else
            ReadData1 <= RegFile(ReadAddr1);
        end IF;

        IF ReadAddr2=0 then
            ReadData2 <= x"00000000";
        else
```

```
            ReadData2 <= RegFile(ReadAddr2);
        end IF;

        IF RegWrite='1' and WriteAddr/=0 then
            RegFile(WriteAddr) <= WriteData;
        end IF;
    end IF;
    end process;
end architecture aRegFile;
```

Mémoire d'instruction

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity eInstMem is
    port (
        signal ReadAddr : in std_logic_vector (6 downto 0);
        signal clk : in std_logic;
        signal Instruction31_0 : out std_logic_vector (31 downto 0)
    );
end entity eInstMem;

use work.SpyOnMySigPkg.all;

architecture aInstMem of eInstMem is
type instmem is array (0 TO 31) of std_logic_vector(31 downto 0);
signal mem : instmem := ((("00100000000010000000000011111111"),
                          ("10100000000010000000000000000000"),
                          ("10000000000010000000000000000000"),
                          ("00100000000010010101010000110010"),
                          ("00000001000010010100100000100110"),
                          ("10101100000010010000000000000000"),
                          ("00100000000010110000000000001101"),
                          ("00100000000011000000000001000000"),
                          ("00000001011011000101100000100101"),
                          ("10100000000010110000000000000100"),
                          ("00110001011010110000000000000000"),
                          ("00100000000010110000000011111001"),
                          ("00100000000011000000000000001111"),
                          ("00000001011011000101100000100100"),
                          ("00110101011010110000000001000000"),
                          ("10100000000010110000000000000101"),
                          ("00100000000011010100100111111111"),
                          ("00000000000011010110101000000010"),
                          ("00110101101011010100110100000000"),
                          ("00000000000011010110110000000000"),
                          ("00110101101011010101000001010011"),
                          ("10101100000011010000000000001000"),
                            others=> (others=>'0')));
                             --here we put the code that we want to execute
```

```
begin
    process (mem)
    begin
    for i in 0 TO 31 LOOP
        Globalmem(i) <= mem(i);
    end LOOP;
    end process;

    process (clk)
    begin
    if rising_edge(clk) then
        Instruction31_0 <= mem(to_integer(unsigned(ReadAddr))/4);
    end if;
    end process;
end architecture aInstMem;
```

ALU

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
use work.constants.all;

entity ALU is
    port (
        A  : in  signed(31 downto 0);   -- operand A
        B  : in  signed(31 downto 0);   -- operand B
        OP : in  std_logic_vector(3 downto 0); -- opcode
        Y  : out signed(31 downto 0);  -- operation result
        Z  : out std_logic -- zero
    );
end ALU;

architecture behavioral of ALU is
begin
    process(A, B, OP)
    variable aux_y : signed(31 downto 0);
    begin
        case OP is  -- decode the opcode and perform the operation:
            when alu_add =>  aux_y := A + B;
            when alu_sub =>  aux_y := A - B;
            when alu_srl =>  aux_y := SHIFT_RIGHT(B, TO_INTEGER(A));
            when alu_sll =>  aux_y := SHIFT_LEFT(B, TO_INTEGER(A));
            when alu_slt =>
                if A < B then
                    aux_y := x"00000001";
                else
                    aux_y := (others => '0');
                end if;
            when alu_and  =>  aux_y := A and B;
            when alu_or   =>  aux_y := A or B;
            when alu_xor  =>  aux_y := A xor B;
            when alu_nor  =>  aux_y := A nor B;
            when alu_mult =>  aux_y := TO_SIGNED(TO_INTEGER(A)*TO_INTEGER(B),32);
            when alu_div  =>  aux_y := TO_SIGNED(TO_INTEGER(A)/TO_INTEGER(B),32);
            when others   =>  aux_y := A + B;
        end case;
        if aux_y = 0 then
            Z <= '1';
        else
            Z <= '0';
```

```
        end if;
        Y <= aux_y;
  end process;
end behavioral;
```

Contrôleur

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
USE work.constants.all;
USE work.functions_pack.all;

entity control_sync is
    port (
        CLOCK    : in  std_logic;
        RESET    : in  std_logic;
        OP       : in  std_logic_vector(5 downto 0);  -- opcode
        func     : in  std_logic_vector(5 downto 0);  -- func code
        RegDst   : out std_logic;  -- Register destination
        Jump     : out std_logic;
        JumpR    : out std_logic;  -- Jump register
        beq      : out std_logic;  -- Branch if equals
        MemToReg : out std_logic;  -- Memory to register
        ALUop    : out std_logic_vector(3 downto 0);  -- ALU op code
        MemWrite : out std_logic;  -- Memory write
        ALUsrc1  : out std_logic;
        ALUsrc2  : out std_logic;
        RegWrite : out std_logic;  -- Register write
        bne      : out std_logic;  -- Branch if not equals
        LoadBU   : out std_logic;  -- Load byte unsigned
        LoadB    : out std_logic;  -- Load byte
        storeB   : out std_logic;
        JumpAL   : out std_logic;   -- Jump and link
        PC_EN    : out std_logic;
        current_state : out std_logic_vector(2 downto 0)
    );
end control_sync;

architecture arch of control_sync is
    type state_type is (fetch, decode, execute, memory, write_back);
    signal state : state_type;

begin
    process(CLOCK, RESET)
    begin
        if RESET = '1' then
            state <= fetch;
        elsif CLOCK'event and CLOCK = '1' then
            case state is
```

```
                when fetch =>
                    state <= decode;

                when decode =>
                    if OP = R_type and func = c_jr then
                        state <= write_back;
                    elsif OP = c_j or OP = c_jal then
                        state <= write_back;
                    else
                        state <= execute;
                    end if;

                when execute =>
                    if OP = c_lb or OP = c_lw or OP = c_lbu then
                        state <= memory;
                    else
                        state <= write_back;
                    end if;

                when memory =>
                    state <= write_back;

                when write_back =>
                    state <= fetch;

                when others =>
                    state <= fetch;
            end case;
        end if;
    end process;

    process(state, OP, func)
    begin
        RegDst   <= '0';
        Jump     <= '0';
        JumpR    <= '0';
        beq      <= '0';
        MemToReg <= '0';
        ALUop    <= "0000";
        MemWrite <= '0';
        ALUsrc1  <= '0';
        ALUsrc2  <= '0';
        RegWrite <= '0';
        bne      <= '0';
        LoadBU   <= '0';
        LoadB    <= '0';
        storeB   <= '0';
        JumpAL   <= '0';
```

```
        PC_EN    <= '0';
        case state is
            when fetch =>
                current_state <= "000";

            when decode =>
                current_state <= "001";

            when execute =>
                current_state <= "010";
                if OP = c_sb or OP = c_sw then
                    MemWrite <= '1';
                end if;

                if OP(5 downto 3) = op_imm then
                    RegWrite <= '1';
                end if;

                if OP = R_type then
                    RegWrite <= '1';
                    RegDst   <= '1';
                    if func = c_sll or func = c_srl then
                        ALUsrc1 <= '1';
                    end if;
                elsif not (OP = c_beq or OP = c_bne) then
                    ALUsrc2 <= '1';
                end if;

                if OP = c_lb then
                    LoadB <= '1';
                end if;

                if OP = c_lbu then
                    LoadB  <= '1';
                    LoadBU <= '1';
                end if;

                if OP = c_sb then
                    storeB <= '1';
                end if;

                case OP is
                    when R_type =>
                        case func is
                            when c_sll  => ALUop <= alu_sll;
                            when c_srl  => ALUop <= alu_srl;
                            when c_mult => ALUop <= alu_mult;
                            when c_div  => ALUop <= alu_div;
```

```
                        when c_sub  => ALUop <= alu_sub;
                        when c_and  => ALUop <= alu_and;
                        when c_or   => ALUop <= alu_or;
                        when c_xor  => ALUop <= alu_xor;
                        when c_nor  => ALUop <= alu_nor;
                        when c_slt  => ALUop <= alu_slt;
                        when others => ALUop <= alu_add;
                    end case;
                when c_andi => ALUop <= alu_and;
                when c_ori  => ALUop <= alu_or;
                when c_xori => ALUop <= alu_xor;
                when c_slti => ALUop <= alu_slt;
                when others => ALUop <= alu_add;
            end case;

        when memory =>
            current_state <= "011";
            RegWrite <= '1';
            MemToReg <= '1';

        when write_back =>
            current_state <= "100";
            PC_EN    <= '1';

            if (OP = R_type and func = c_jr) or OP = c_j or OP = c_jal then
                Jump <= '1';
            end if;

            if OP = R_type and func = c_jr then
                JumpR <= '1';
            end if;

            if OP = c_beq or OP = c_bne then
                ALUop   <= alu_sub;
            end if;

            if OP = c_beq then
                beq <= '1';
            elsif OP = c_bne then
                bne <= '1';
            end if;

            if OP = c_jal then
                RegWrite <= '1';
                JumpAL <= '1';
            end if;

        when others =>
```

```
                current_state <= "111";

        end case;
    end process;

end arch;
```

Banch

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity eBanch is
end entity eBanch;

use work.SpyOnMySigPkg.all;

architecture aBanch of eBanch is

signal observeRegFile : type_regmem;
signal observemem : type_instmem;
signal observememdata : type_mem32w;
signal observePC : std_logic_vector (31 downto 0);
signal tb_clk, tb_rst : std_logic := '1';

component eMicroprocessor is
    port (
        signal micro_clk, control_rst, regfile_rst, pc_rst,
                memdata_rst : in std_logic
    );
end component;

begin

    tb_clk <=  '1' after 0.5 ns when tb_clk = '0' else
        '0' after 0.5 ns when tb_clk = '1';

    instMicroprocessor : eMicroprocessor
        port map(micro_clk => tb_clk, control_rst => tb_rst,
        regfile_rst => tb_rst, pc_rst => tb_rst, memdata_rst => tb_rst);

    process (GlobalRegFile)
    begin
        for i in 1 TO 31 loop
            observeRegFile(i) <= GlobalRegFile(i);
        end loop;
    end process;

    process (GlobalPC)
    begin
        observePC <= GlobalPC;
```

```
    end process;

    process (Globalmem)
    begin
   for i in 0 TO 31 loop
        observemem(i) <= Globalmem(i);
   end loop;
    end process;

    process (Globalmemdata)
    begin
   for i in 0 TO 31 loop
        observememdata(i) <= Globalmemdata(i);
   end loop;
    end process;

    process begin
        wait for 3 ns;
        tb_rst <= '0';
    end process;


end architecture aBanch;
```

Constantes

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

package constants is
    -- ALU constants
    constant alu_add  : std_logic_vector(3 downto 0) := "0000";
    constant alu_sub  : std_logic_vector(3 downto 0) := "0001";
    constant alu_srl  : std_logic_vector(3 downto 0) := "0010";
    constant alu_sll  : std_logic_vector(3 downto 0) := "0011";
    constant alu_slt  : std_logic_vector(3 downto 0) := "0100";
    constant alu_and  : std_logic_vector(3 downto 0) := "0101";
    constant alu_or   : std_logic_vector(3 downto 0) := "0110";
    constant alu_xor  : std_logic_vector(3 downto 0) := "0111";
    constant alu_nor  : std_logic_vector(3 downto 0) := "1000";
    constant alu_mult : std_logic_vector(3 downto 0) := "1001";
    constant alu_div  : std_logic_vector(3 downto 0) := "1010";


    -- Control constants
    constant R_type  : std_logic_vector(5 downto 0) := "000000";
    constant c_sll   : std_logic_vector(5 downto 0) := "000000";
    constant c_srl   : std_logic_vector(5 downto 0) := "000010";
    constant c_jr    : std_logic_vector(5 downto 0) := "001000";
    constant c_mult  : std_logic_vector(5 downto 0) := "011000";
    constant c_div   : std_logic_vector(5 downto 0) := "011010";
    constant c_add   : std_logic_vector(5 downto 0) := "100000";
    constant c_sub   : std_logic_vector(5 downto 0) := "100010";
    constant c_and   : std_logic_vector(5 downto 0) := "100100";
    constant c_or    : std_logic_vector(5 downto 0) := "100101";
    constant c_xor   : std_logic_vector(5 downto 0) := "100110";
    constant c_nor   : std_logic_vector(5 downto 0) := "100111";
    constant c_slt   : std_logic_vector(5 downto 0) := "101010";

    constant c_j     : std_logic_vector(5 downto 0) := "000010";
    constant c_jal   : std_logic_vector(5 downto 0) := "000011";
    constant c_beq   : std_logic_vector(5 downto 0) := "000100";
    constant c_bne   : std_logic_vector(5 downto 0) := "000101";

    constant op_imm  : std_logic_vector(2 downto 0) := "001";
    constant c_addi  : std_logic_vector(5 downto 0) := "001000";
    constant c_slti  : std_logic_vector(5 downto 0) := "001010";
    constant c_andi  : std_logic_vector(5 downto 0) := "001100";
```

```
    constant c_ori   : std_logic_vector(5 downto 0) := "001101";
    constant c_xori  : std_logic_vector(5 downto 0) := "001110";

    constant c_sb    : std_logic_vector(5 downto 0) := "101000";
    constant c_sw    : std_logic_vector(5 downto 0) := "101011";
    constant c_lbu   : std_logic_vector(5 downto 0) := "100100";
    constant c_lb    : std_logic_vector(5 downto 0) := "100000";
    constant c_lw    : std_logic_vector(5 downto 0) := "100011";

end constants;
```

Testbench controleur

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
USE work.constants.all;
USE work.functions_pack.all;

entity tb_control_sync is
end tb_control_sync;

architecture test of tb_control_sync is

    component control_sync is
        port (
            CLOCK    : in  std_logic;
            RESET    : in  std_logic;
            OP       : in  std_logic_vector(5 downto 0);  -- opcode
            func     : in  std_logic_vector(5 downto 0);  -- func code
            RegDst   : out std_logic;  -- Register destination
            Jump     : out std_logic;
            JumpR    : out std_logic;  -- Jump register
            beq      : out std_logic;  -- Branch if equals
            MemToReg : out std_logic;  -- Memory to register
            ALUop    : out std_logic_vector(3 downto 0);  -- ALU op code
            MemWrite : out std_logic;  -- Memory write
            ALUsrc1  : out std_logic;
            ALUsrc2  : out std_logic;
            RegWrite : out std_logic;  -- Register write
            bne      : out std_logic;  -- Branch if not equals
            LoadBU   : out std_logic;  -- Load byte unsigned
            LoadB    : out std_logic;  -- Load byte
            storeB   : out std_logic;
            JumpAL   : out std_logic;   -- Jump and link
            PC_EN    : out std_logic;
            current_state : out std_logic_vector(2 downto 0)
        );
    end component;

    -- control PORTS
    signal CLOCK    : std_logic;
    signal RESET    : std_logic;
    signal OP       : std_logic_vector(5 downto 0);  -- opcode
    signal func     : std_logic_vector(5 downto 0);  -- func code
    signal RegDst   : std_logic;  -- Register destination
```

```
    signal Jump     : std_logic;
    signal JumpR    : std_logic;  -- Jump register
    signal beq      : std_logic;  -- Branch if equals
    signal MemToReg : std_logic;  -- Memory to register
    signal ALUop    : std_logic_vector(3 downto 0);  -- ALU op code
    signal MemWrite : std_logic;  -- Memory write
    signal ALUsrc1  : std_logic;
    signal ALUsrc2  : std_logic;
    signal RegWrite : std_logic;  -- Register write
    signal bne      : std_logic;  -- Branch if not equals
    signal LoadBU   : std_logic;  -- Load byte unsigned
    signal LoadB    : std_logic;  -- Load byte
    signal storeB   : std_logic;
    signal JumpAL   : std_logic;    -- Jump and link
    signal PC_EN    : std_logic;
    signal current_state : std_logic_vector(2 downto 0);

    constant CLK_PERIOD : time := 100 ns;

    signal sim_finished : boolean := false;

begin

    dut: control_sync port map(CLOCK, RESET, OP, func, RegDst, Jump, JumpR,
                               beq, MemToReg, ALUop,
                                MemWrite, ALUsrc1, ALUsrc2, RegWrite, bne, LoadBU,
                               LoadB, storeB,
                               JumpAL, PC_EN, current_state);

    clk_generation: process
    begin
        if not sim_finished then
            CLOCK <= '1';
            wait for CLK_PERIOD/2;
            CLOCK <= '0';
            wait for CLK_PERIOD/2;
        else
            wait;
        end if;
    end process clk_generation;


    -- Test control
    simulation: process

        procedure async_reset is
        begin
            wait until rising_edge(CLOCK);
```

```
        wait for CLK_PERIOD/4;
        RESET <= '1';

        wait for CLK_PERIOD/2;
        RESET <= '0';
    end procedure async_reset;

    procedure check_code(constant test_identifier : in String;
                         constant op_in :
                                 in std_logic_vector(5 downto 0);
                         constant func_in :
                                 in std_logic_vector(5 downto 0);
                         constant expected_execute :
                                 in std_logic_vector(14 downto 0);
                         constant expected_memory :
                                 in std_logic_vector(5 downto 0);
                         constant expected_write_back :
                                 in std_logic_vector(13 downto 0);
                         constant thereIsExecuteState :
                                 in boolean;
                         constant thereIsMemoryState  :
                                 in boolean) is
        variable observerd_execute : std_logic_vector(14 downto 0);
        variable observed_memory  : std_logic_vector(5 downto 0);
        variable observed_write_back : std_logic_vector(13 downto 0);
    begin

        -- Assign values to circuit inputs

        OP <= op_in;
        func <= func_in;

        assert fetch_decode_assert(PC_EN, MemWrite, RegWrite) = '1'
        report "Test " & test_identifier & "- error fetch "
                & integer'image(to_integer(unsigned(current_state)))
        severity error;

        wait until rising_edge(CLOCK);
        wait for CLK_PERIOD/4;

        assert fetch_decode_assert(PC_EN, MemWrite, RegWrite) = '1'
        report "Test " & test_identifier & " - error decode "
                & integer'image(to_integer(unsigned(current_state)))
        severity error;

        wait until rising_edge(CLOCK);
        wait for CLK_PERIOD/4;
```

```
        if thereIsExecuteState then
            observerd_execute := PC_EN & MemWrite & RegWrite & ALUsrc1
                                 & ALUsrc2 & MemToReg
                                 & RegDst & JumpAL & LoadB & LoadBU
                                 & storeB & ALUop;
            assert match_assert(expected_execute, observerd_execute)
            report "Test " & test_identifier & " - error execute "
                    & integer'image(to_integer(unsigned(current_state)))
                    & lf &
                    "   " & to_string(expected_execute) & " = "
                    & to_string(observerd_execute)
            severity error;

            wait until rising_edge(CLOCK);
            wait for CLK_PERIOD/4;
        end if;

        if thereIsMemoryState then
            observed_memory := PC_EN & MemWrite & RegWrite & MemToReg
                               & RegDst & JumpAL;
            assert match_assert(expected_memory, observed_memory)
            report "Test " & test_identifier & " - error memory "
                   & integer'image(to_integer(unsigned(current_state)))
                   & lf &
                   "   " & to_string(expected_memory)  & " = "
                    & to_string(observed_memory)
            severity error;

            wait until rising_edge(CLOCK);
            wait for CLK_PERIOD/4;
        end if;

        observed_write_back := PC_EN & MemWrite & RegWrite & Jump & beq & bne
                               & ALUsrc1 & ALUsrc2
                               & JumpAL & JumpR & ALUop;
        assert match_assert(expected_write_back, observed_write_back)
        report "Test " & test_identifier & " - error write_back "
                & integer'image(to_integer(unsigned(current_state))) & lf &
                "   " & to_string(expected_write_back)  & " = "
                & to_string(observed_write_back)
        severity error;

        wait until rising_edge(CLOCK);
        wait for CLK_PERIOD/4;

    end procedure check_code;
```

```
    begin

        -- execute order    : PC_EN & MemWrite & RegWrite & ALUsrc1
        --& ALUsrc2 & MemToReg & RegDst & JumpAL & LoadB & LoadBU & storeB;
        -- memory order     : PC_EN & MemWrite & RegWrite & MemToReg
        --& RegDst & JumpAL;
        -- write back order : PC_EN & MemWrite & RegWrite & Jump & beq & bne
        --& ALUsrc1 & ALUsrc2 & JumpAL & JumpR;

        async_reset;
      --check_code(Identifier, OP, func, expected_execute, expected_memory,
      --expected_write_back, thereIsExecuteState, thereIsMemoryState)
        check_code("Multiplication",      R_type, c_mult,
        "00100010000" & alu_mult,
        "000000", "1000000000" & "0000",  true,  false);
        check_code("Division",            R_type, c_div,
        "00100010000" & alu_div,
        "000000", "1000000000" & "0000",  true,  false);
        check_code("Sum",                 R_type, c_add,
        "00100010000" & alu_add,
          "000000", "1000000000" & "0000",  true,  false);
        check_code("Subtraction",         R_type, c_sub,
        "00100010000" & alu_sub,
          "000000", "1000000000" & "0000",  true,  false);
        check_code("And bitwise",         R_type, c_and,
        "00100010000" & alu_and,
          "000000", "1000000000" & "0000",  true,  false);
        check_code("Or bitwise",          R_type, c_or,
        "00100010000" & alu_or,
           "000000", "1000000000" & "0000",  true,  false);
        check_code("Xor bitwise",         R_type, c_xor,
        "00100010000" & alu_xor,
          "000000", "1000000000" & "0000",  true,  false);
        check_code("Nor bitwise",         R_type, c_nor,
        "00100010000" & alu_nor,
          "000000", "1000000000" & "0000",  true,  false);
        check_code("Set less than",       R_type, c_slt,
        "00100010000" & alu_slt,
          "000000", "1000000000" & "0000",  true,  false);

        check_code("Shift left logical",  R_type, c_sll,
        "00110010000" & alu_sll,
          "000000", "1000000000" & "0000",  true,  false);
        check_code("Shift right logical", R_type, c_srl,
        "00110010000" & alu_srl,
          "000000", "1000000000" & "0000",  true,  false);

        check_code("Sum imm",             c_addi, "000000",
```

```
        "00101000000" & alu_add,
          "000000", "1000000000" & "0000",  true,  false);
        check_code("Set less than imm",    c_slti, "000000",
        "00101000000" & alu_slt,
          "000000", "1000000000" & "0000",  true,  false);
        check_code("And bitwise imm",      c_andi, "000000",
        "00101000000" & alu_and,
          "000000", "1000000000" & "0000",  true,  false);
        check_code("Or bitwise imm",       c_ori,  "000000",
        "00101000000" & alu_or,
           "000000", "1000000000" & "0000",  true,  false);
        check_code("Xor bitwise imm",      c_xori, "000000",
        "00101000000" & alu_xor,
          "000000", "1000000000" & "0000",  true,  false);


        check_code("Branch equals",        c_beq,  "000000",
        "00000000000" & "0000",
          "000000", "1000100000" & alu_sub,  true,  false);
        check_code("Branch not equals",    c_bne,  "000000",
        "00000000000" & "0000",
          "000000", "1000010000" & alu_sub,  true,  false);

        check_code("Store word",           c_sw,   "000000",
        "01001000000" & alu_add,
          "000000", "1000000000" & "0000",  true,  false);
        check_code("Store byte",           c_sb,   "000000",
        "01001000001" & alu_add,
          "000000", "1000000000" & "0000",  true,  false);

        check_code("Jump register",        R_type, c_jr,
        "00000000000" & "0000",
           "000000", "1001000001" & alu_add, false, false);
        check_code("Jump",                 c_j,    "000000",
        "00000000000" & "0000",
           "000000", "1001000000" & "0000",  false, false);
        check_code("Jump and link",        c_jal,  "000000",
        "00000000000" & "0000",
           "000000", "1011000010" & "0000",  false, false);

        check_code("Load word",            c_lw,   "000000",
        "00001000000" & alu_add,
          "001100", "1000000000" & "0000",  true,   true);
        check_code("Load byte",            c_lb,   "000000",
        "00001000100" & alu_add,
          "001100", "1000000000" & "0000",  true,   true);
        check_code("Load byte unsigned",   c_lbu,  "000000",
        "00001000110" & alu_add,
```

```
          "001100", "1000000000" & "0000",  true,   true);

        sim_finished <= true;
        wait;
    end process simulation;
end test;
```

Assembleur

```
from bitstring import Bits

OPcode = {'sll':'000000','srl':'000000','jr':'000000',
        'add':'000000','sub':'000000','and':'000000',
        'or':'000000','nor':'000000','slt':'000000',
        'mult':'000000','div':'000000','xor':'000000',
        'j':'000010','jal':'000011','beq':'000100',
        'bne':'000101','addi':'001000','slti':'001010',
        'andi':'001100','ori':'001101','xori':'001110',
        'lb':'100000','lw':'100011','lbu':'100100',
        'sw':'101011','sb':'101000'
        }

Funct = {'sll':'000000','srl':'000010','jr':'001000',
        'add':'100000','sub':'100010','and':'100100',
        'or':'100101','nor':'100111','slt':'101010',
        'mult':'011000','div':'011010','xor':'100110'
        }

Registers = {'$zero':0,'$at':1,'$v0':2,'$v1':3,'$a0':4,
        '$a1':5,'$a2':6,'$a3':7,'$t0':8,'$t1':9,'$t2':10,
        '$t3':11,'$t4':12,'$t5':13,'$t6':14,'$t7':15,
        '$s0':16,'$s1':17,'$s2':18,'$s3':19,'$s4':20,
        '$s5':21,'$s6':22,'$s7':23,'$t8':24,'$t9':25,
        '$k0':26,'$k1':27,'$gp':28,'$sp':29,'$fp':30,
        '$ra':31
        }

for r in Registers:
    Registers[r] = '{0:05b}'.format(Registers[r])

TypeR=['sll','srl','jr','add','sub','and','or','nor','slt','mult','div','xor']
TypeI=['beq','bne','addi','slti','andi','ori',
        'xori','lb','lw','lbu','sw','sb']
TypeJ=['j','jal']

with open('code.asm') as f:
    code = f.readlines()
code = [x.strip() for x in code]

addr = 0

Labels = {}
```

```
for line in code:
    binary = ''
    tmp = line.split()
    if line and tmp[0] not in TypeR and tmp[0] not in TypeI and tmp[0] not in TypeJ:
        Labels[tmp[0][:-1]] = addr
    elif line:
        addr += 1
print(Labels)


addr = 0
machinecode = ''

for line in code:
    binary = ''
    tmp = line.split()
    if line and tmp[0] not in TypeR and tmp[0] not in TypeI and tmp[0] not in TypeJ:
        binary = binary
    elif line:
        addr += 1
        binary += OPcode[tmp[0]]
        tmp1 = tmp[1].split(',')
        if tmp[0] in TypeR:
            if tmp[0] in ['sll','srl']:
                binary += '00000'
                binary += Registers[tmp1[1]]
                binary += Registers[tmp1[0]]
                binary += '{0:05b}'.format(int(tmp1[2]))
            elif tmp[0]=='jr':
                binary += Registers[tmp1[0]]
                binary += '0000000000'
                binary += '00000'
            else:
                binary += Registers[tmp1[1]]
                binary += Registers[tmp1[2]]
                binary += Registers[tmp1[0]]
                binary += '00000'
            binary += Funct[tmp[0]]
        elif tmp[0] in TypeI:
            if tmp[0] in ['beq','bne']:
                binary += Registers[tmp1[0]]
                binary += Registers[tmp1[1]]
                if tmp1[2] in Labels:
                    binary += Bits(int=Labels[tmp1[2]]-addr,length=16).bin
                else:
                    binary += '{0:016b}'.format(int(tmp1[2]))
            elif tmp[0] in ['addi','slti','andi','ori','xori']:
```

```
                binary += Registers[tmp1[1]]
                binary += Registers[tmp1[0]]
                binary += '{0:016b}'.format(int(tmp1[2]))
            else:
                tmp2 = tmp1[1].split('(')
                binary += Registers[tmp2[1][:-1]]
                binary += Registers[tmp1[0]]
                binary += '{0:016b}'.format(int(tmp2[0]))
        elif tmp[0] in TypeJ:
            if tmp1[0] in Labels:
                binary += '{0:026b}'.format(Labels[tmp1[0]])
            else:
                binary += '{0:026b}'.format(int(tmp1[0]))
        print(binary)
        machinecode += '(\"' + binary + '\"),'

print(machinecode)
```

Assembleur - code test

Code 1

```
addi    $t0,$zero,4
    addi    $t1,$zero,5
    addi    $t2,$zero,9
label:
    addi    $t0,$t0,1
    beq $t0,$t1,label
    addi    $t0,$t0,1
    bne $t0,$t2,label
    sw  $t2,0($zero)
```

Code 2

```
    addi    $t0,$zero,12
    addi    $t1,$zero,3
    addi    $t2,$zero,2
    addi    $t3,$zero,7

    addi    $s0,$zero,32

    jal Division
    jr  $s0

Produit:
    mult    $t6,$t5,$t3

Soustraction:
    sub $t5,$t4,$t2
    j   Produit

Division:
    div $t4,$t0,$t1
    jr  $ra
```

Code 3

```
    addi    $t0,$zero,255
    sb  $t0,0($zero)
    lb  $t0,0($zero)
    addi    $t1,$zero,21554
    xor $t1,$t0,$t1
    sw  $t1,0($zero)

    addi    $t3,$zero,13
    addi    $t4,$zero,64
    or  $t3,$t3,$t4
    sb  $t3,4($zero)

    andi    $t3,$t3,0
    addi    $t3,$zero,249
    addi    $t4,$zero,15
    and $t3,$t3,$t4
    ori $t3,$t3,64
    sb  $t3,5($zero)

    addi    $t5,$zero,18943
    srl $t5,$t5,8
    ori $t5,$t5,19712
    sll $t5,$t5,16
    ori $t5,$t5,20563
    sw  $t5,8($zero)
```