



SORBONNE UNIVERSITÉ

Master 1 SESI

(Systèmes Électroniques et Systèmes Informatiques)



Projet RiVer :
Design et implémentation d'un processeur RISC-V pipeliné



Étudiants:

M. Timothée Le Berre
M. Louis Geoffroy Pitailler
M. Kevin Lastra
M. Samy Attal

Nous souhaitons remercier
Mme. Roselyne Chotin, Mme. Daniela Genius,
M. Pirouz Bazargan Sabet, M. Franck Wajsbürt et M. Damien Fruleux
qui nous ont énormément aidé lors de la réalisation de ce projet.

Table de Matières

1	Introduction	4
1.1	Historique de l'architecture RISC :	4
1.2	L'origine du projet	6
1.3	Objectifs	7
1.4	Structure du github	8
2	Travail effectué et roadmap	9
3	Implémentation SystemC : Coeur RiVer	11
3.1	Introduction	11
3.2	Structure du pipeline	11
3.2.1	IFETCH	11
3.2.2	Decod	11
3.2.3	EXEC	13
3.2.4	Multiplieur	15
3.2.5	Diviseur	17
3.2.6	MEM	17
3.2.7	WBK	18
3.2.8	REG	18
3.3	Extension CSR, exceptions et interruptions	19
3.3.1	Extension CSR :	19
3.3.2	Exceptions traitées dans notre architecture :	20
3.3.3	Réponse à une exception et reset :	22
3.3.4	Reset :	22
3.3.5	Réponse à une exception :	23
3.4	Traitements des Interruptions :	23
3.4.1	Utilisation des registres Csr pour implémenter un timer	23
3.5	Optimisation du cœur	23
3.5.1	Superscalaire 2 étages :	23
3.5.2	Prédiction de branchement	32
3.6	Protocole de validation	36
3.6.1	Tests basiques	36
3.6.2	Gestionnaire d'exceptions	37
3.7	Suite de tests officielle riscosf	38
4	SoC	39
4.1	Master RiVer	40
4.1.1	Caches	40
4.1.2	Wrapper RiVer-Wishbone B4	42
4.2	Cibles	44
5	Implémentation FPGA	46
5.1	Modèle VHDL	46
5.1.1	Interface C/VHDL	48
5.2	FPGA	49
5.2.1	Conception du circuit	49
5.2.2	Configuration de la RAM	54
5.2.3	Analyse post-implémentation	58
6	Miniriscv, un cœur allégé pour l'enseignement	60

7 Conclusion	61
7.1 Problèmes rencontrés lors de la conception	61
7.2 Perspectives d'amélioration	61

1 Introduction

1.1 Historique de l'architecture RISC :

En 1971, Intel sort son premier microprocesseur, l'Intel 4004 basé sur une architecture 4 bits CISC (Complex Instruction Set Computing).

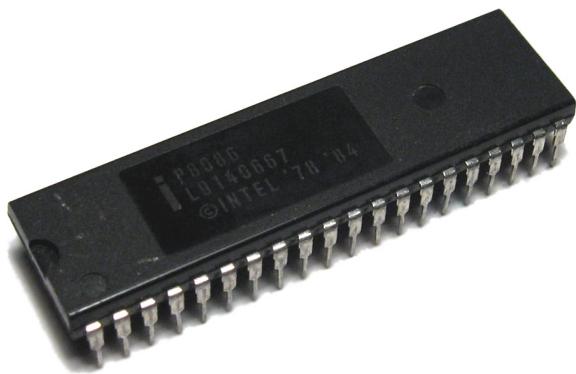


Figure 1: Intel 8086 [1]

Les Processeurs CISC ont largement dominé le marché jusque dans les années 80 où une nouvelle architecture fait son apparition : l'architecture RISC.

Les architectures CISC sont beaucoup plus complexes que les RISC, en effet ces derniers implémentent des fonctions très complexes en matériel. On peut par exemple citer l'Intel 8086 [1] qui implémente matériellement des instructions permettant de faire des comparaisons entre des chaînes de caractères.

Les architectures RISC au contraire implémentent uniquement des fonctions basiques et simples d'un point de vue matériel, la philosophie du RISC étant en effet de laisser les tâches complexes au compilateur.

RISC était à l'origine un projet mené par David Patterson à l'Université de Berkeley en Californie entre 1980 et 1984.

Cette architecture va vite montrer de gros avantages par rapport à l'architecture CISC et de nombreux projets vont se développer en se basant dessus.

En 1981, le MIPS (Microprocessor without Interlocked Pipeline Stages) -qui se base sur une architecture de type RISC- fait son apparition à l'Université de Stanford.

La technologie MIPS va être commercialisée à partir de 1984 et sa première implémentation, le R2000, deviendra l'un des processeurs les plus utilisés pour concevoir des circuits embarqués.

A la fin des années 80, à Sorbonne Université (anciennement Pierre et Marie Curie) tous les cours ayant nécessité d'une architecture processeur utilisent des architectures différentes. C'est pourquoi le professeur Alain Greiner décide d'homogénéiser tout ça en cherchant une architecture à la fois pédagogique et puissante comme base de l'enseignement. Il va donc d'abord se tourner vers le DLX -développé et utilisé par Stanford- mais va vite se rendre compte que l'architecture est peu utilisée et peu connue en dehors du cadre scolaire. C'est pourquoi il va plutôt se tourner vers MIPS.

Cette architecture étant relativement simple, elle permet de présenter les principes de base de l'architecture des processeurs, tout en étant suffisamment puissante pour être réellement implémentée. De plus, l'architecture peut supporter un système d'exploitation multitâche tel qu'UNIX, puisqu'elle supporte deux modes de fonc-

tionnement :

- Un mode utilisateur : certaines zones de la mémoire et certains registres du processeur réservés au système d'exploitation sont protégés et donc inaccessibles,
- Un mode superviseur, toutes les ressources sont accessibles.

Néanmoins, cette architecture est une architecture commerciale et son implémentation est contrôlée, c'est pourquoi le Lip6 et plus particulièrement le Master SESI souhaite changer d'architecture pour passer sur RISC-V, une architecture libre de droits. De plus, RISC-V est assez proche du MIPS ce qui facilite sa compréhension et son implémentation.

C'est pourquoi 3 projets, dont le nôtre, ont été articulés autour de ce jeu d'instruction. En plus de notre projet, il y a un projet visant à prendre un RISC-V synthétisable pour l'implémenter sur une carte FPGA dans le but de faire tourner un Linux dessus. Enfin, un autre projet vise à reprendre une description VHDL d'un MIPS 32 réalisé par le Lip6, pour l'adapter au jeu d'instruction RISC-V.

1.2 L'origine du projet

Ce projet a été dans un premier temps réalisé durant l'UE PSESI de la 1ère année du Master SESI à Sorbonne Université. Il a par la suite donné lieu à un stage qui a permis d'améliorer le modèle initial.

Au cours de notre premier semestre de M1 SESI, nous avons eu l'occasion d'implémenter un cœur scalaire 4 étages basé sur une architecture ARMv2a en VHDL.

Lorsque Mme. Genius a proposé d'implémenter une architecture RISC-V en SystemC [4] nous avons tout de suite postulé afin de pouvoir participer à ce projet.

Cela nous a permis de nous familiariser avec le jeu d'instruction RISC-V et avec le langage SystemC [4]. RISC-V étant l'une des implémentations de RISC les plus utilisées - aux côtés de l'architecture ARM -, il est très intéressant d'en étudier son fonctionnement.

Ne voulant pas simplement réaliser une architecture scalaire et voulant aller plus loin que ce que nous avions déjà eu l'occasion de faire en VLSI avec l'architecture ARM, nous avons décidé d'avancer rapidement sur le projet dans le but de finir début mars l'implémentation scalaire et d'ensuite pouvoir nous concentrer sur une implémentation superscalaire SS2 pipeliné à 5 étages.

Le SS2 désigne un super-scalaire de 2 étages où les étages EXE, MEM et WBK sont dupliqués. Il s'agit d'une architecture imaginée par Mr. Pirouz Bazargan. Nous l'avons étudié dans le cadre de notre UE ARCHI au 1er semestre et nous souhaitions donc l'implémenter dans notre design.

Au cours du semestre, dans le cadre de l'UE PSESI, 3 projets avaient été lancé sur l'architecture RISC-V, le nôtre, un projet visant à implémenter un OS sur FPGA dans lequel un cœur RISC-V tournerait et enfin une implémentation VHDL partant d'un cœur MIPS.

Comme le projet sur FPGA visait à implémenter un OS, il a été décidé, après discussion avec notre encadrante, de commencer d'abord par l'ajout d'une partie Kernel à notre design pour ensuite passer à une implémentation SS2 s'il nous restait du temps.

Nous avons donc choisi d'implémenter le jeu d'instruction RV32IZiscr avec un mode User et Machine. RV32I est le jeu d'instruction de base de RISC-V en version 32 bits, et Zicsr est l'extension ajoutant des registres de status et de contrôle de processeurs, indispensables pour les fonctions systèmes. Ces extensions ont été choisies dans le but de se rapprocher du MIPS étudié au Lip6.

Au mois de mai, après la soutenance de projet, notre encadrante a accepté de nous accorder un stage de 3 mois durant l'été 2022. Cela fut l'occasion pour nous d'accueillir un membre supplémentaire, Mr. Samy Attal. Les deux autres projets n'ayant que peu avancé au cours du semestre, l'implémentation VHDL et l'implémentation sur FPGA n'auront pas été reprises et Mr. Attal a eu la responsabilité de refaire ce portage en partant de notre modèle SystemC. Les objectifs supplémentaires que nous nous sommes fixés seront détaillés dans la section 1.3

1.3 Objectifs

L'objectif premier de notre projet était d'implémenter une architecture RISC-V 32 bits pipelinée sur 5 étages sans extension.

Le but étant de remplacer l'architecture MIPS du Lip6. Nous avons choisi d'utiliser les mêmes étages que ceux présents dans le MIPS pour réaliser notre implémentation, à savoir :

- IFETCH
- DECODE
- EXECUTE
- MEMORY
- WRITE-BACK

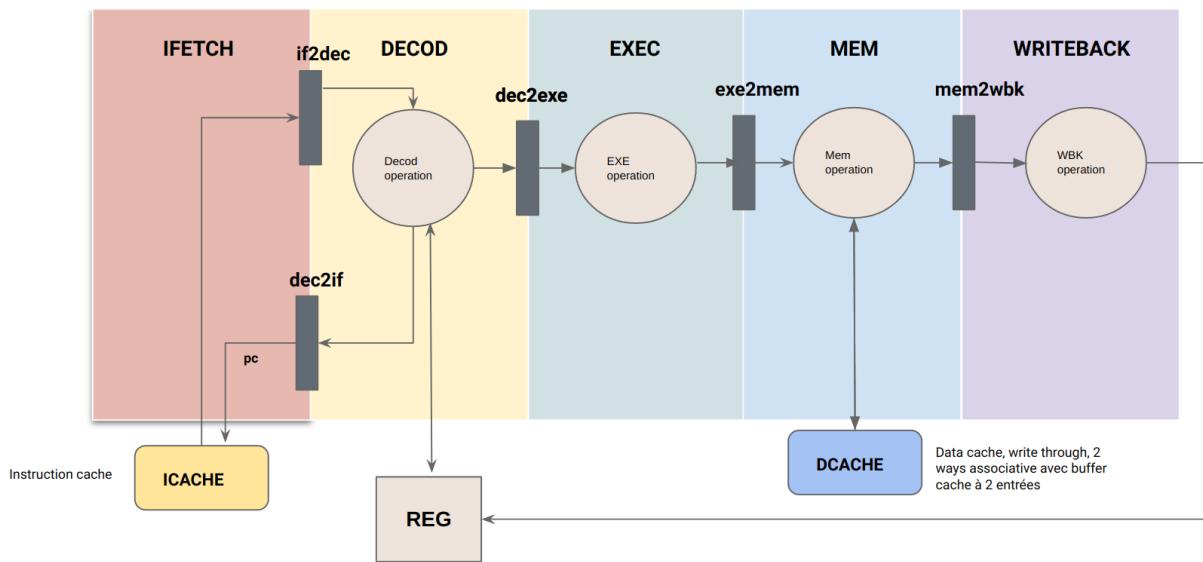


Figure 2: Schéma du pipeline, les blocs entre chaque étage désignent des fifos et le nom qui leur est attribué

Le langage imposé pour cette réalisation est SystemC [4], en effet, il s'agit du langage utilisé pour l'UE MASSOC du M2 d'où ce choix. Notre implémentation doit se rapprocher le plus possible du MIPS R3000 mentionné plus haut afin de ne pas rendre trop rude la transition pour l'enseignement.

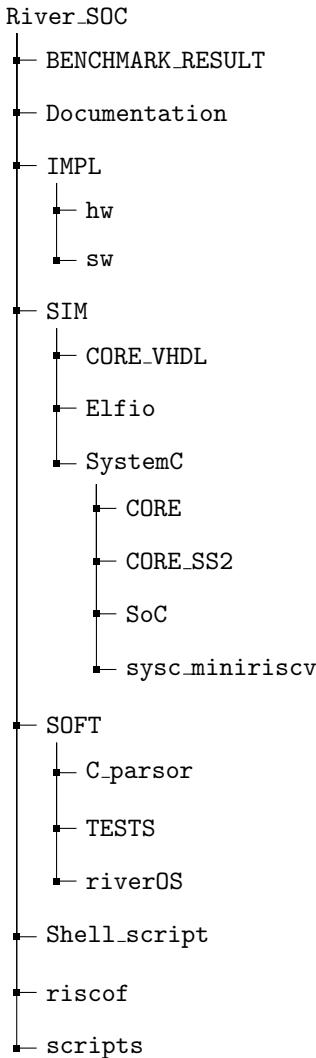
Nous avions ensuite implémenté l'extension Zicsr ainsi que les modes de privilège et les interruptions, ainsi que des caches d'instruction et de données, toujours pour se rapprocher du processeur MIPS des TME.

Enfin au cours de notre stage de 3 mois, nous avons développé un total de 4 coeurs différents :

- Un cœur SystemC RV32IMZiscr avec mode user, un mode machine et avec un mécanisme de prédition de branchement que nous avons appelé **RiVer**,
- Le même cœur, mais cette fois-ci traduit en VHDL,
- Un cœur superscalaire RV32IZiscr avec mode user et mode machine,
- Un mini-riscv pour la plateforme de TP qui implémente uniquement le jeu d'instruction RV32I

Enfin un bus système basé sur Whishbone, un mécanisme Snoopy pour le cache et des périphériques auront également été développé.

1.4 Structure du github



Le git contient l'ensemble du projet, que ce soit l'implémentation hardware ou encore la partie software. Le Readme présent sur Github contient plus d'information sur la hiérarchie des dossiers, si vous souhaitez de plus ample information à leur sujet, vous pouvez consulter notre [git](#).

Les 3 dossiers les plus importants sont les suivants :

- **IMPL** : contient le code source des IP FPGA et des drivers développés
- **SIM** : contient le code source de tous les cœur (VHDL et SystemC) ainsi que la librairie ELFIO utilisée pour parser les fichiers elf dans le cas des cœur SystemC
- **SOFT** : contient de nombreux tests assembleurs et C utilisés pour déboguer les coeurs. Contient également le code de reset, le code du gestionnaire d'exception et un prototype d'OS en Rust. Le dossier C_pasor contient un modification du code utilisé pour l'interface GHDL/C, ce code modifié permet d'extraire une zone de la mémoire dans un fichier texte.

Pour utiliser un des coeurs du projet, il suffit d'aller dans le dossier souhaité CORE, CORE_SS2 ou CORE_VHDL, compiler le projet en effectuant la commande **make**.

Enfin, il suffit de passer en argument de l'exécutable un fichier assembleur, un fichier c ou bien un fichier ELF

et le cœur exécutera le code.

2 Travail effectué et roadmap

L'ensemble du travail effectué a été réalisé entre janvier 2022 et août 2022. Au fur et à mesure du projet, nous avons fixé une roadmap nous permettant de déterminer des dates limites et ainsi de prévoir en amont l'avancement du projet. Le tableau 2 récapitule l'ensemble des tâches effectuées, recense la date à laquelle elles ont été finalisées, et quel membre du projet y ont contribué.

Timothée Le Berre	Blue
Louis Geoffroy Pitailler	Red
Kevin Lastra	Green
Samy Attal	Dark Blue

	Janvier	Février	Mars	Avril	Mai	Juin	Juillet	Aout		
Documentation RiscV	X									Red
Mips R3000			X							Green
Maj et documentation MIPS30000			X							
CORE RiscV										
Etage IFETCH			X						Blue	
Etage DECODE			X						Red	
Etage EXEC			X						Blue	Red
Etage MEMORY			X							
Etage WRITEBACK			X						Blue	Red
Débogage CORE			X						Red	
Bypass				X						
Suite de test RiscV basiques					X					
Test extension I "RiscOf"						X			Blue	
Infrastructure Git, intégration continue							X			
Prédiction de branchement								X		Green
RAS								X		
Extension RiscV										
Ziscr				X					Blue	Red
Gestion des exceptions						X			Red	
M (multiplication/division)								X		Green
SoC										
ICache				X						Green
DCache				X						
Interfaçage caches-coeur					X					
Bus Wishbone							X			
Dual core								X		Green
SS2										
Interface des étages							X			Red
Bypass							X			
Debeug								X		
Benchmark								X		Red
VHDL/FPGA										
Interface C/GHDL						X			Red	
RV32I						X				
M								X		
Zicsr (kernel)								X		
Prédiction de branchement								X		
Implémentation simple FPGA							X			
Implémentation finale FPGA								X		
Développement des IP								X		
Développement des drivers								X		
TPs, système d'exploitation										
Nettoyage, commentaires du code							X		Blue	
Adaptation des sujet de TP									X	
Adaptation pour OS (timer & TTY)								X		
Début d'adaptation de k06								X		

X : Représente le mois où la tache a été finalisée

3 Implémentation SystemC : Coeur RiVer

3.1 Introduction

Le coeur que nous avons réalisé est un coeur scalaire comportant 5 étages de pipeline avec un mode User et un mode machine. Nous avons implémenté les extensions I, M et Zicsr.

Cette section a pour objectif de détailler en profondeur les choix architecturaux qui ont été fait pour la réalisation du processeur.

3.2 Structure du pipeline

3.2.1 IFETCH

IFETCH est l'étage chargé de faire l'interface avec le cache d'instruction. Il permet de faire une requête au cache pour demander l'instruction en lien avec la valeur du PC qu'il reçoit de Decod ou avec la valeur envoyé par le prédicteur de branchement qui sera détaillé dans la section 3.5. Une fois l'instruction reçue, il va la transmettre à la fifo IF2DEC qui fait l'interface avec l'étage Decod.

Cet étage comporte une Interface avec le cache d'instructions qui sera détaillé dans la section 4.1.1.

3.2.2 Decod

Decod est l'un des étages les plus complexes de l'architecture puisque son rôle est de récupérer l'instruction en provenance de IFETCH et de la décoder pour informer tout le reste du pipeline des tâches à effectuer.

Pour décoder les instructions, nous nous sommes aidés de la spécification de RISCV [7] et nous avons écrit une synthèse des instructions que nous avons utilisées, accessible sur notre GitHub [13].

Nous avons classé les instructions en 8 catégories distinctes :

- R-TYPE : Les instructions de type R sont des instructions arithmétiques qui utilisent uniquement des registres,
- I-TYPE : Instructions où l'opérande 2 est de type immédiat,
- B-type : Instructions de branchement conditionnel,
- U-type : lui et auipc (add upper immediat to pc),
- J-type : Branchements inconditionnels
- S-TYPE : Instructions de type Store
- CSR-TYPE : Instructions CSR (control and status register), cf section 3.3.1
- System-type : Instructions système (ecall/ebreak)

Cette répartition nous a permis de faciliter le décodage, nous nous sommes ensuite servis de la documentation pour récupérer les opcode correspondant à chaque instruction.

RV32I Base Instruction Set					
imm[31:12]				rd	0110111
imm[31:12]				rd	0010111
imm[20 10:1 11 19:12]				rd	1101111
imm[11:0]	rs1	000	rd	1100111	JALR
imm[12 10:5]	rs2	000	imm[4:1 11]	1100011	BEQ
imm[12 10:5]	rs2	001	imm[4:1 11]	1100011	BNE
imm[12 10:5]	rs2	100	imm[4:1 11]	1100011	BLT
imm[12 10:5]	rs2	101	imm[4:1 11]	1100011	BGE
imm[12 10:5]	rs2	110	imm[4:1 11]	1100011	BLTU
imm[12 10:5]	rs2	111	imm[4:1 11]	1100011	BGEU
imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU
imm[11:5]	rs2	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	010	imm[4:0]	0100011	SW
imm[11:0]	rs1	000	rd	0010011	ADD
imm[11:0]	rs1	010	rd	0010011	SLTI
imm[11:0]	rs1	011	rd	0010011	SLTIU
imm[11:0]	rs1	100	rd	0010011	XORI
imm[11:0]	rs1	110	rd	0010011	ORI
imm[11:0]	rs1	111	rd	0010011	ANDI
0000000	shamt	001	rd	0010011	SLLI
0000000	shamt	101	rd	0010011	SRLI
0100000	shamt	101	rd	0010011	SRAI
0000000	rs2	000	rd	0110011	ADD
0100000	rs2	000	rd	0110011	SUB
0000000	rs2	001	rd	0110011	SLL
0000000	rs2	010	rd	0110011	SLT
0000000	rs2	011	rd	0110011	SLTU
0000000	rs2	100	rd	0110011	XOR
0000000	rs2	101	rd	0110011	SRL
0100000	rs2	101	rd	0110011	SRA
0000000	rs2	110	rd	0110011	OR
0000000	rs2	111	rd	0110011	AND

Figure 3: Opcode, extrait de la documentation RISCV [7]

Decod est composé de plusieurs parties, il comprend en effet deux fifos **DEC2IF** et **DEC2EXE** permettant de transmettre les informations aux étages IFETCH et EXE respectivement.

Il comprend ensuite un additionneur qui effectue l'incrémentation de PC. En effet, pour respecter ce qui était fait en MIPS, nous avons fait en sorte que le calcul d'adresse se fasse dans cet étage. Dans le cas d'un branchement, on effectue les comparaisons nécessaires (égalité, inférieur, supérieur...) directement dans Decod et l'on génère un signal inc_pc_sd qui indique si l'on stoppe l'incrémentation standard de l'adresse (+4) ou si l'on ajoute l'offset du branchement à PC.

Nous avons retiré les "delayed slots" après les branchements et nous avons ajouté un protocole de vidage des fifo (flushage) dans le cas d'un branchement qui réussit. Une fois tous les signaux décodés, Decod envoie toutes les informations nécessaires à EXE à l'aide de la fifo **DEC2EXE** et envoie la valeur de PC suivante à IFETCH grâce à **DEC2IF**.

3.2.2.1 M extension

Enfin, nous avons commencé fin avril l'ajout de l'extension M permettant d'ajouter un multiplicateur et un diviseur en matériel. En effet, n'ayant pas ces deux entités, il nous est impossible d'effectuer efficacement les opérations de multiplication, division ou modulo dans un programme C. Il est donc nécessaire de redéfinir ces fonctions à chaque programme.

Un processeur ne possédant pas ces composants matériels peut émuler les opérations normalement effectué par le composant à l'aide du gestionnaire d'exception. Mais l'exécution d'une multiplication, par exemple, sera moins optimisées et prendra plus de cycle qu'avez un multiplicateur. L'ajout de ces composants permet donc de réduire le nombre de cycle nécessaire à l'exécution de ces opérations et permet donc d'augmenter les performances du processeur.

RV32M Standard Extension					
	rs2	rs1	000	rd	0110011
0000001			000	rd	0110011
0000001			001	rd	0110011
0000001			010	rd	0110011
0000001			011	rd	0110011
0000001			100	rd	0110011
0000001			101	rd	0110011
0000001			110	rd	0110011
0000001			111	rd	0110011

Figure 4: Opcode, extrait de la documentation RISCV, extension M : [7]

3.2.3 EXEC

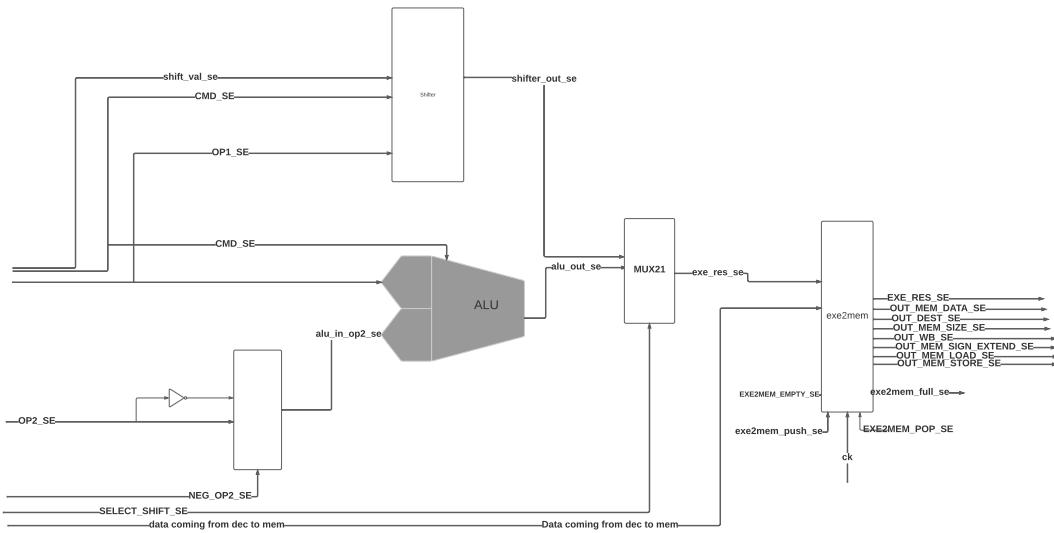


Figure 5: Schéma simplifié de l'architecture de EXEC

EXEC est constitué de 2 parties primordiales : l'ALU (*arithmetic logic unit*) et le shifter. Pour sélectionner l'entité que l'on veut utiliser, Decod envoie un signal SELECT_SHIFT_SE à EXEC qui indique si l'on sélectionne le shifter ou l'ALU.

Une commande sur 2 bits indique ensuite l'opération à réaliser, opérations logic pour l'ALU et décalage pour le shifter. Pour réaliser les soustractions, nous avons placé un inverseur commandé dans EXEC qui permet de faire le complément à deux d'un signal et ainsi de réaliser une soustraction.

Le résultat de EXEC est ensuite envoyé dans la fifo **EXE2MEM** qui permet de transmettre toutes les données nécessaires à l'étage MEM.

3.2.4 Multiplieur

Afin de pouvoir utiliser les multiplications il est nécessaire que nous implémentions un multiplieur. L'implémentation que nous avions choisi était initialement l'Array Multiplier.

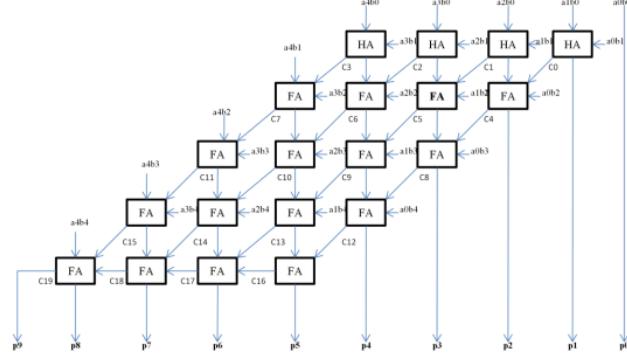


Figure 6: Array multiplier[14]

Néanmoins, après discussion avec Mr. Bazargan Sabet, nous avons décidé de changer d'implémentation, car cette dernière est assez peu optimisée et nécessite des temps de propagation très longs. L'idée avancée par Mr. Bazargan Sabet est d'utiliser un multiplieur pipeliné sur 3 étages (x_0 , x_1 et x_2) se basant sur l'algorithme "Wallace Tree multiplier" ou "multiplieur de Wallace" [6]. L'idée étant qu'en cas de multiplication, Decod va propager un signal pour informer les étages EXE, MEM et WKB qu'une opération de ce type va être lancée, ce qui va bloquer les bypass et l'écriture en mémoire. L'instruction est ensuite envoyée vers l'étage x_0 qui va commencer à calculer les rangées de produits partiels, suite à ce calcul les rangée vont être propagées dans le premier étage de l'arbre de wallace, les quelques vont s'étendre jusqu'à l'étage x_1 , et finalement la sortie de l'arbre va être additionnée dans l'étage x_2 .

$$\begin{array}{r}
 \begin{matrix} a_i & \dots & \dots & \dots & \dots & \dots & a_0 \\ b_i & \dots & \dots & \dots & \dots & \dots & b_0 \end{matrix} \\
 \hline
 \begin{matrix} 0/1 & \dots & 0/1 & a & b & \dots & \dots & \dots & a & b & \dots & \dots & a & b & 0 \\ & & & 31 & 0 & & & & 0 & 0 & & & & & & = M_0 \end{matrix} \\
 \begin{matrix} 0/1 & \dots & 0/1 & a & b & \dots & \dots & \dots & a & b & \dots & \dots & a & b & 0 \\ & & & 31 & 1 & & & & 0 & 1 & & & & & & = M_1 \end{matrix} \\
 \begin{matrix} 0/1 & \dots & 0/1 & a & b & \dots & \dots & \dots & a & b & \dots & \dots & a & b & 0 & 0 \\ & & & 31 & 2 & & & & 0 & 2 & & & & & & = M_2 \end{matrix} \\
 \vdots \\
 \begin{matrix} 0/1 & a & b & \dots & a & b & 0 & \dots & \dots & \dots & 0 & \dots & \dots & \dots & 0 & = M_{31} \\ & 31 & 31 & & 0 & 31 & & & & & & & & & & & \end{matrix}
 \end{array}$$

Figure 7: Multiplication(a_i représentation binaire de a , b_i représentation binaire de b)

M_i : correspond à une rangée de produit partiel de $\sum_{n=0}^i a_n * b_i$

Multiplication signée

3 cas à prendre en compte :

1. si A et B sont négatifs, on doit uniquement appliquer le complément A2.
2. si B est négatif, les bits de poids fort de notre produit partiel vont être égal à 1.
3. si A est négatif, on inverse A et B et donc on applique les cas numéro 2.

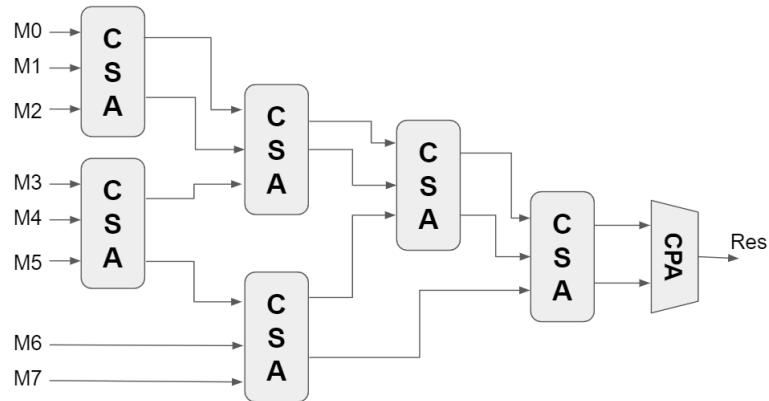


Figure 8: Wallace tree multiplier

CSA : Carry save adder

CPA : Carry propagation adder

Pour implémenter le multiplieur, il convient donc d'implémenter ce dernier dans les étages EXE, MEM et WBK que l'on renomme respectivement X0,X1 et X2. Ces derniers vont ainsi se charger d'effectuer le calcul nécessaire pour exécuter l'instruction multiplication.

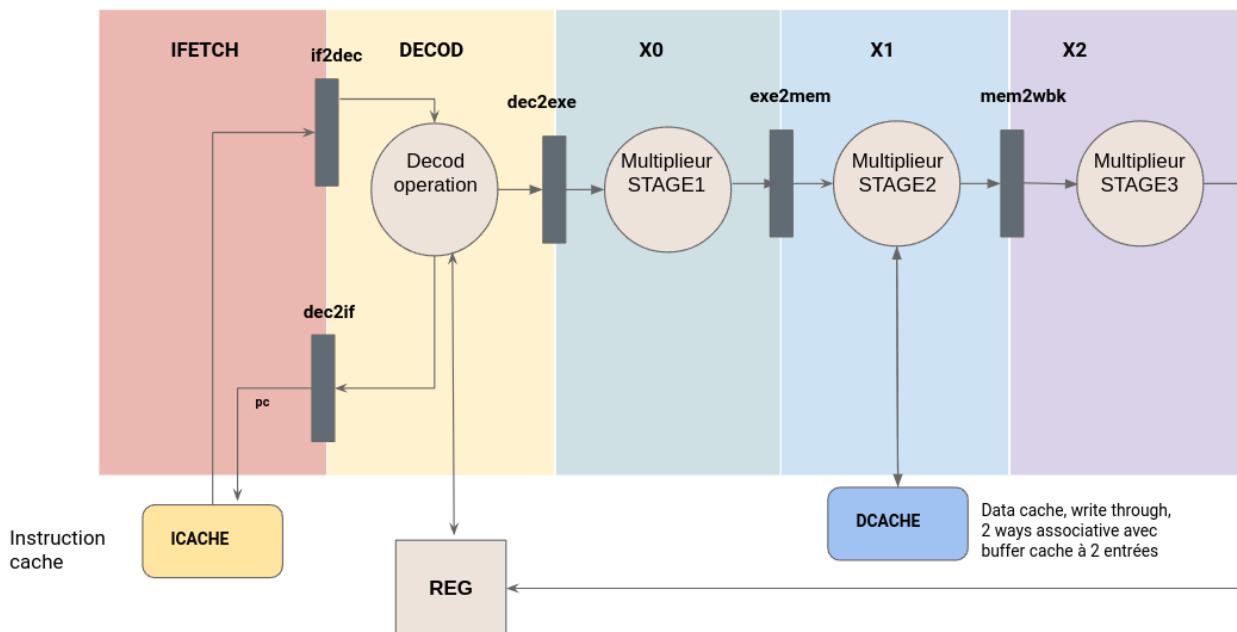


Figure 9: Schéma du Multiplieur intégré au pipeline

3.2.5 Diviseur

Notre implémentation du diviseur est composée de 3 registres à décalage (2 de 64bits et 1 de 32bits), un soustracteur et une machine à état.

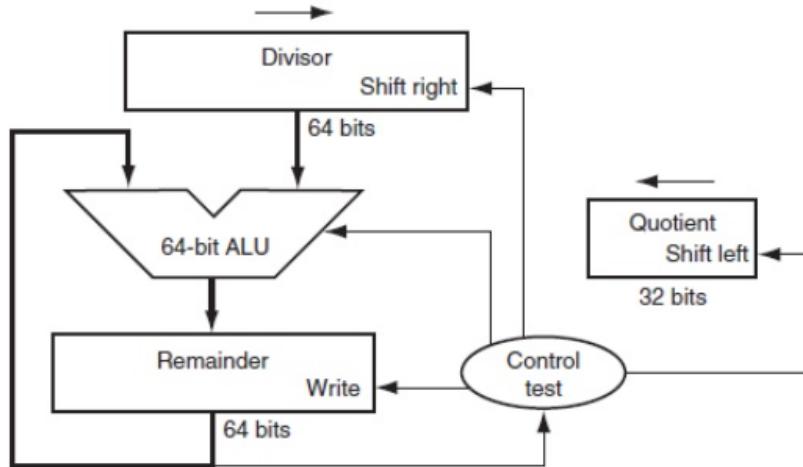


Figure 10: Schéma de l'algorithme de division[16]

Algorithm 1 Algorithme de division

```

cpt ← 0
while cpt <= 31 do
    if diviseur >= reste then
        reste ← reste - diviseur
    end if
    quotient ← quotient << 1                                ▷ logic shift left
    quotient ← quotient + 1
    diviseur ← diviseur >> 1                                ▷ logic shift right
    cpt ← cpt + 1
end while

```

Division par zéro

Le résultat d'une division peut varier selon l'instruction utilisée :

Pour REM et REMU on renvoie le dividende, pour DIV et DIVU on renvoie -1.

3.2.6 MEM

MEM effectue les accès mémoire load et store. Il reçoit un signal EXE_MEM_SIZE_SM qui indique si l'accès se fait en octets, en half-word ou en word. Ainsi, dans le cas d'un store, cela indique quels sont les bits stockés en mémoire et dans le cas d'un load quels sont les bits que l'on souhaite garder dans le registre destination.

Dans notre implémentation, nous avons permis de faire des accès alignés sur le type d'accès. C'est-à-dire que dans le cas d'un lb, il est possible de faire des accès octets-alignés, dans le cas d'un lh half-word-alignés.

Dans le cas où l'on effectue un load, un masquage sera fait dans l'étage MEM permettant ainsi de charger que les bits liés au type d'accès. Les bits sont toujours chargés dans les bits de poids faible du registre de destination.

Dans le cas d'un store, les accès peuvent aussi être type-alignés, mais cette fois-ci le masquage n'est pas fait dans l'étage MEM. En effet, nous n'avions initialement pas pensé à la problématique des sb et sh qui ne doivent modifier que la partie du registre qu'ils souhaitent stocker. C'est pourquoi l'adresse que nous envoyions à notre core_tb était toujours sur 32 bits.

Le problème étant que notre mémoire est simulé par une map c++ qui est indexé par l'adresse, or dans le cas d'un store byte à l'adresse 0xF000C121 par exemple, la map ne doit pas alloué l'adresse 0xF000C121, elle doit écrire le 2ème octet de l'adresse 0xF000C120.

Pour palier le problème, nous avons envoyé la taille de l'accès au cœur qui réalise un masquage sur l'adresse reçue et ne remplace que les bits nécessaires dans la map c++.

Enfin, les données traitées sont envoyées à WBK à l'aide de la fifo **MEM2WBK**.

On trouve également toute la gestion des exceptions dans cet étage, mais cela sera développé dans la section [3.3](#).

3.2.7 WBK

WBK reçoit les signaux en provenance de MEM et il va faire une écriture dans le banc de registre si nécessaire (en effet les instructions store n'écrivent rien). Un signal REG_WB_SW est envoyé à REG indiquant si oui ou non la donnée reçue doit être enregistrée.

3.2.8 REG

Enfin, le banc de registres contient 33 registres, le 33ème étant PC. Nous avons choisi de placer PC dans REG afin d'utiliser les mêmes signaux pour les instructions de données.

En effet, REG est placé directement dans le cœur, il n'appartient pas à un étage spécifique puisque Decod comme WBK peuvent y faire des accès, Decod faisant des accès en lecture et WBK faisant des accès en écriture.

Ainsi placer PC dans le banc de registre nous permet d'utiliser les mêmes interfaces pour les instructions de branchement, par exemple jal.

On notera que l'adresse de lecture est sur 6 bits dans REG malgré le fait que les adresses des opcode soient uniquement sur 5.

En effet dans le cas des branchements, on lit la valeur de PC, d'où l'intérêt de passer ça sur 6 bits.

3.3 Extension CSR, exceptions et interruptions

Une fois notre processeur scalaire avec l'extension de base I (Integer) implémenté, nous sommes passés à l'implémentation de la partie Kernel qui consistait en 3 points essentiels :

- Extension CSR,
- Gestion des Interruptions et des exceptions,
- Ajout d'un mode user et d'un mode kernel

3.3.1 Extension CSR :

31	20 19	15 14	12 11	7 6	0
csr	rs1	funct3	rd	opcode	
12	5	3	5	7	
source/dest	source	CSRRW	dest	SYSTEM	
source/dest	source	CSRRS	dest	SYSTEM	
source/dest	source	CSRRC	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM	

Figure 12: RISC-V CSR Instruction

Instruction	funct3	opcode
CSRRW	001	1110011
CSRRS	010	1110011
CSRRC	011	1110011
CSRRWI	101	1110011
CSRRSI	110	1110011
CSRRCI	111	1110011

Figure 11: Opcode Instructions CSR

Cette extension est nécessaire pour l'ajout d'un mode Machine/superviseur puisqu'elle permet d'ajouter des instructions gérant des registres CSR (control status registers). Les CSR sont l'équivalent des registres du coprocesseur-0 en MIPS et les instructions CSR sont l'analogue des instructions type mfc0/mtc0. Ainsi, il y a des instructions permettant de lire/écrire ces registres statut.

Pour les implémenter, nous avons donc ajouté le banc de registre CSR. Ce banc stocke des informations sur l'architecture implémentée ainsi que sur l'état actuel du pipeline. On peut lire dans la spécification livre 2 [8] que l'architecture RISC-V prévoit jusqu'à 4096 CSR, néanmoins ils ne sont pas tous définis ni tous nécessaires. C'est pourquoi nous n'avons implémenté que ceux nécessaires à notre architecture, à savoir :

- mvendorid : identifiant du vendeur du CPU,
- marchid : donne des informations sur la base utilisé pour l'architecture, 32 dans notre cas,
- mimpid : donne la version du CPU,
- mhartid : correspond a l'ID du CPU,

- mstatus : c'est le registre de statut du processeur, il garde de nombreuses informations sur le processeur par exemple le mode courant du processeur, l'activation ou non des interruptions...
- misa : donne les extensions implémentées dans l'architecture
- mie : Contient des informations sur les interruptions machine activée
- mtvec : contient l'adresse de base des fonctions trap, permettant de traiter une exception/interruption, supporte le **mode vectorisé** et le **mode direct**
- mstatush : idem que mstatus
- mepc : stocke l'adresse qui a causé l'interruption/l'exception
- mcause : contient un code identifiant la cause de l'exception/interruption qui s'est produite
- mtval : Quand un trap est pris en mode machine, mtval stocke une information sur l'exception
- mip : contient des informations sur les interruptions à venir
- mscratch : Contient l'adresse d'une pile permettant au programme de sauvegarder les valeurs des registres csr
- kernel : Registre CSR custom, placé à l'adresse **0x800**, que nous avons ajouté afin de gérer l'adresse superviseur en software.

Pour ajouter le décodage des instructions dans le pipeline, il a été nécessaire de modifier quasiment l'ensemble des étages. En effet, les instructions CSR sont des opérations atomiques, c'est-à-dire que quand il y a une instruction CSR dans le pipeline, une autre instruction manipulant le même registre ne peut pas écrire ou lire le CSR qui est en cours de traitement.

Pour pallier à ce problème, nous avons donc propagé un signal `csr_enable` dans chaque étage dans le but de dire si une instruction de type CSR est en cours dans cet étage. Ce signal est ensuite redirigé sur Decod pour l'informer qu'il ne peut pas effectuer une instruction manipulant le même CSR que celle déjà présente dans le pipeline et qu'il doit donc geler si tel était le cas.

De plus, comme expliqué dans la partie précédente, nous avons défini une frontière entre EXE et MEM pour la gestion des exceptions, les CSR sont donc écrits à la fin de MEM i.e. au moment où l'on regarde quel est la cause de l'exception. Une fois ce traitement terminé, la mise à jour des CSR se termine à la fin du cycle MEM.

A la fin du projet nous avons également décidé d'ajouter un registre custom pour gérer l'adresse du mode privilégié. Ce registre est initialisé à `0xFFFFFFFF` lors du reset, cela permet d'éviter des problèmes lors des tests avec riscof. En effet des programmes ne prenant pas en compte cette fonctionnalité pourrait être amené à ne pas tourner et à générer des exceptions puisqu'il tourne en mode user. Si l'on laisse la valeur de ce registre à 0 lors du reset du processeur et que le code d'un programme ne l'initialise pas cela va générer de nombreuses exceptions d'accès illégaux.

Pour valider notre modèle, nous avons utilisé le framework riscof [15] et nous avons donc adapté notre architecture quand c'était nécessaire afin de coller à la suite de test.

C'est pourquoi nous avons choisi d'écrire la valeur de PC responsable d'une exception de type misaligned ou access-fault ainsi que la valeur d'un store/load faisant le même type d'erreur dans mtval. Cela n'est pas obligatoire si l'on suit la spécification, mais en pratique, c'est utile pour le traitement d'une exception.

3.3.2 Exceptions traitées dans notre architecture :

Afin de pouvoir implémenter le mode machine, nous avons dû rendre possible la détection et la gestion des exceptions et des interruptions dans notre processeur.

M. Pirouz Bazargan Sabet nous a expliqué comment il avait implémenté et géré les exceptions dans le pipeline du MIPS. Grâce à ses explications, nous avons eu une base nous permettant d'implanter ces dernières.

Afin de mettre en place la gestion des exceptions/interruptions sur notre processeur RISC-V, nous avons dans un premier temps recensé toutes les exceptions de la spécification que nous allions implémenter. Voici une liste

exhaustive de ces exceptions :

- Instruction address misaligned : Adresse de l'instruction mal alignée
- Instruction access fault : Essaie d'accéder à une zone mémoire sans les privilèges nécessaires
- Illegal instruction : L'instruction n'existe pas
- Load address misaligned : Adresse de load mal alignée
- Load access fault : Accès à une zone avec un privilège trop faible
- Store address misaligned : Adresse d'un store mal alignée
- Store access fault : Accès à une zone avec un privilège trop faible
- Environment call from U-mode : Appel système en mode User
- Environment call from M-mode : Appel système en mode Machine
- Environment call from wrong mode : mret dans le mauvais mode
- Mret : L'instruction MRET est traitée comme une exception

Une fois les exceptions à implémenter listées, il a fallu déterminer dans quel étage nous allions les gérer et pour cela, nous avons dû placer une "barrière" dans le pipeline.

En effet, lorsqu'une instruction déclenche une exception, les instructions suivantes ne doivent pas être prises en compte. Or, d'un point de vue micro-architectural, cela signifie que ces instructions ne doivent ni modifier la mémoire ni le banc de registre REG/CSR.

Cela signifie que toutes instructions qui arrivent dans MEM ou dans WBK après qu'une exception est détectée ne doivent pas modifier la mémoire ou les bancs de registres. Pour faire cela, il convient de désactiver l'accès mémoire et le signal permettant d'écrire dans le banc de registre dans WBK lorsqu'une instruction est détectée. On précise que la désactivation dans MEM se fait avant l'accès mémoire.

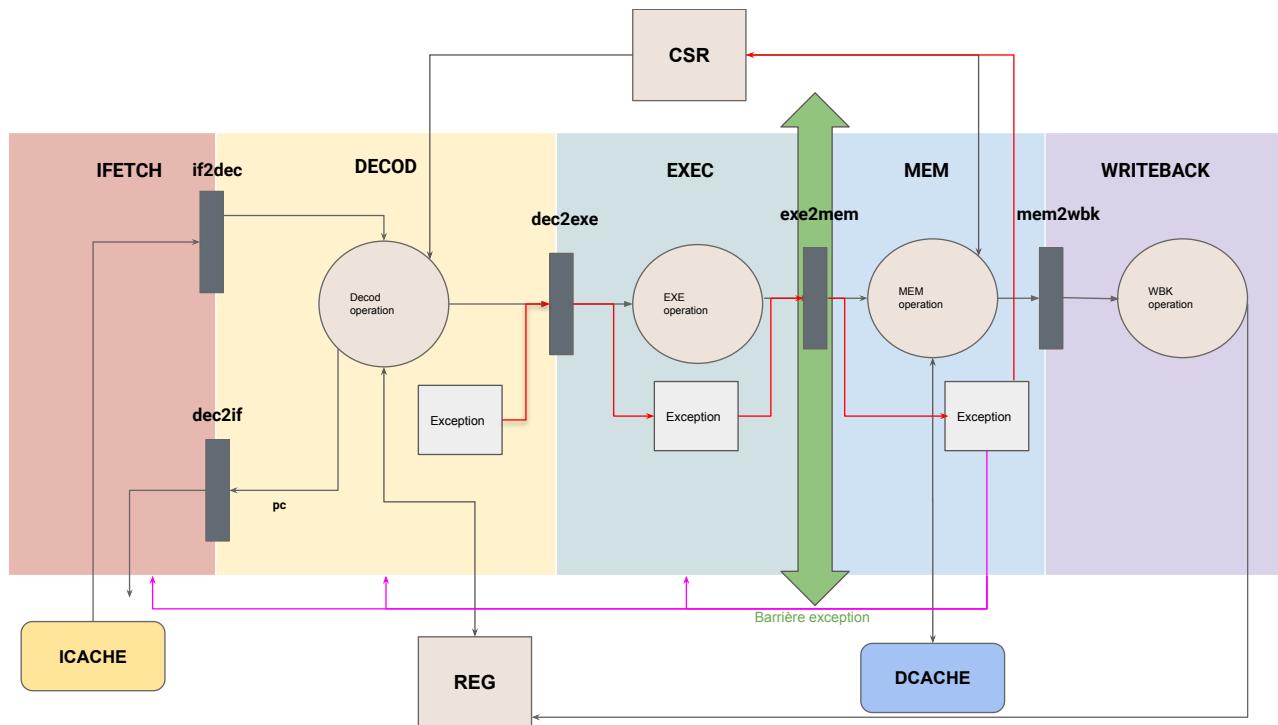


Figure 12: Schéma du pipeline avec la gestion des exceptions

Le pipeline est donc le même que précédemment à quelques différences près. Chaque étage détecte les exceptions qui lui sont propres, on peut les répartir comme suit :

- Instruction address misaligned : Decod
- Instruction access fault : EXE
- Illegal instruction : DEC
- Load address misaligned : EXE
- Load access fault : EXE
- Store address misaligned : EXE
- Store access fault : EXE
- Environment call from U-mode : DEC
- Environment call from M-mode : DEC
- Environment call wrong mode : DEC (custom)
- Mret : DEC

Dans chaque étage on effectue un "ou" logique entre toutes les exceptions que l'on propage dans l'étage suivant. Par exemple, dans Decode on va effectuer :

illegal_instruction **ou** adress_misaligned **ou** syscall_u_mode **ou** syscall_s_mode

Ensuite, on va envoyer le résultat de ce **ou** dans EXE où l'on va faire un nouveau **ou** avec toutes les exceptions de cet étage.

Cela permet de soulager l'étage MEM, en effet on pourrait simplement envoyer tous les signaux dans MEM et faire le **ou** dans cet étage, mais cela augmenterait la durée de propagation des signaux dans MEM qui est déjà longue en raison des accès mémoire.

De plus, nous avons modifié le pipeline pour que le PC de chaque instruction soit transmis d'un étage à un autre. Ainsi, lorsqu'une instruction arrive dans MEM, si une exception est détectée il suffit d'écrire la valeur de PC dans les mepc.

3.3.3 Réponse à une exception et reset :

3.3.4 Reset :

Notre processeur démarre en mode machine, toutes les instructions du reset se feront donc avec le mode de privilège le plus haut. Ce dernier se terminant par une instruction mret il va sortir du mode machine pour passer dans le mode user et ainsi passer au main.

Le code de reset initialise les registres status nécessaire au fonctionnement du coeur, on doit initialiser :

- l'adresse du main dans mepc
- l'adresse du gestionnaire d'exception dans mtvec
- une adresse, permettant à un OS de sauvegarder temporairement les valeurs des csr en mémoire, dans mscratch,
- l'adresse de départ de la zone privilégiée dans le registre csr kernel (**0x800**)

Une fois les registres csr initialisé il faut sauvegardé l'adresse des fonctions du gestionnaire d'exception dans l'isr vector.

Enfin il faut initialiser le stack pointeur sp.

3.3.5 Réponse à une exception :

Lorsqu'une ou plusieurs exceptions arrivent dans MEM il va donc falloir les traitées et interrompre le programme en cours dans le pipeline.

Si plusieurs exceptions arrivent au même cycle dans l'étage MEM, le processeur va les traiter avec un ordre de priorité défini dans le spec RISCV.

Une fois que MEM détecte une exception il va effectuer les étapes suivantes :

- Modification du registre mstatus pour changer le mode du pipeline avec un privilège plus haut (passage du mot user au mode machine dans la plupart des cas)
- Ecriture de l'adresse de l'instruction responsable de l'exception dans le registre csr mepc
- Ecriture de la valeur responsable de l'exception s'il y en a une dans mtval (exemple valeur de load mal-alignée)
- Ecriture du numéro d'exception dans le registre csr mcause
- Envoie d'un signal à Ifetch, Decod et Exec leur indiquant qu'une exception a eu lieu

En réponse au signal reçu, les étages Ifetch, Dec et Exec vont vider leur fifos en y insérant des instructions nop. Decod va ensuite chargé la valeur de mtvec dans PC.

Notre implémentation supporte les deux modes proposés par mtvec, à savoir un mode direct et un mode vectorisé, peu importe le mode présent, c'est Decod qui se charge d'envoyer la bonne adresse dans Ifetch.

Enfin, Ifetch va donc récupérer la valeur du gestionnaire d'Interruption et il va exécuter le code de notre gestionnaire d'Interruption

3.4 Traitement des Interruptions :

3.4.1 Utilisation des registres Csr pour implémenter un timer

En l'absence de réel standard sur les timer dans les processeurs RISC-V, nous avons décidé d'utiliser les Csr comme registre de contrôle pour un composant "Timer", permettant de remplir les registres time et timeh donnant le temps système, ainsi que de générer des interruptions à intervalle régulier, utiles en particulier pour les programmes types système d'exploitation.

La spécification prévoit des numéros de registre d'utilisation libre décidée par les concepteurs du processeur. Nous avons choisi deux de ces registres, le 5C0 et le 5C1, pour contrôler le timer.

Le premier est le registre de configuration, donc le premier bit décide d'activer ou non les interruptions, et le deuxième décide si l'interruption timer se "réarme" automatiques une fois traitée, ou s'il faut le faire depuis le programme.

Le second registre est le "diviseur" du timer, qui permet de diviser la fréquence des interruptions (en pratique, il s'agit du nombre de "ticks" du timer entre deux interruptions).

Ces registres sont "write only", c'est-à-dire qu'on peut écrire dedans, mais qu'une lecture renvoie toujours 0.

3.5 Optimisation du cœur

Après avoir terminé l'implémentation de notre cœur avec les extensions I, M et le mode User/Machine, nous avons souhaité améliorer certains mécanisme dans l'optique d'augmenter les performances du cœur. Pour ce faire, nous avons choisi d'implémenter un mécanisme de prédiction de branchement et de designer un autre cœur superscalaire.

3.5.1 Superscalaire 2 étages :

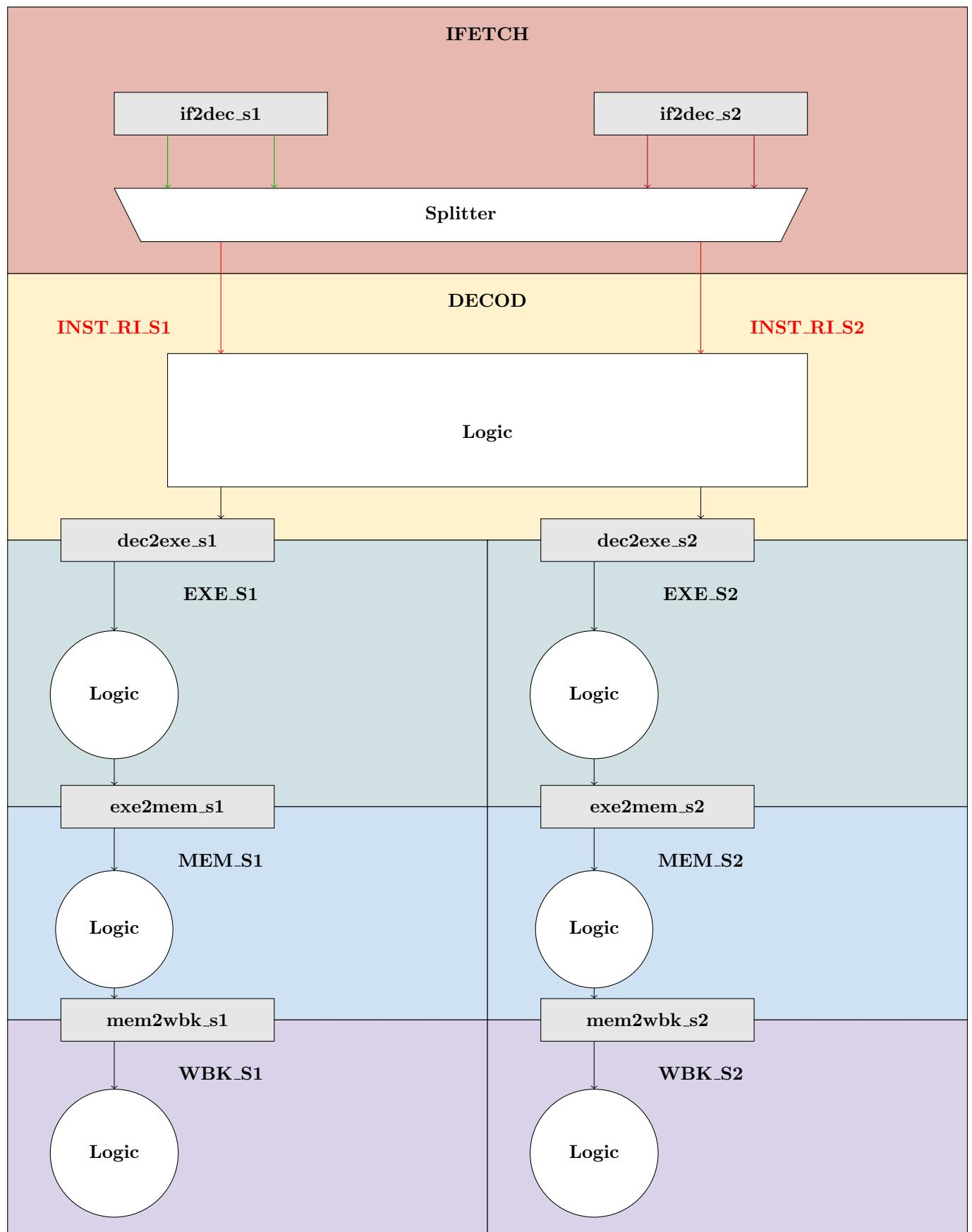
Une fois le scalaire avec partie Kernel terminé, nous sommes donc passés à la réalisation du double pipeline pour le Super-scalaire.

Nous avons choisi de dupliquer les étages EXEC, MEM et WBK tout en laissant le multiplicateur dans un pipeline séparé et nous avons modifié les étages DECOD et IFETCH afin qu'ils soient en mesure de traiter deux instructions/adresses en un seul cycle.

Pour réaliser le SS2 nous avons divisé sa création en deux étapes :

- un travail préparatoire visant à faire le gros du travail c'est-à-dire renommer les signaux et préparer les interfaces entre tous les étages,
- implémenter la logique permettant de gérer les dépendances de données et l'implémentation de tous les bypass.

Nous avons choisi de conserver la priorité de l'étage S1 sur l'étage S2, en effet en cas de dépendance de données sur DECOD l'inversion du chargement des instructions se fera directement dans IFETCH. Cela sera plus détaillé dans la section [3.5.1.1](#).



3.5.1.1 IFETCH

Nous avons modifié IFETCH afin que ce dernier soit en mesure de demander deux adresses au cache en un seul cycle. Nous avons choisi de conserver deux adresses distinctes plutôt que d'utiliser une seule adresse alignée sur 8. Ce choix avait été fait, car nous avons implémenté un mécanisme de prédiction de branchement sur notre cœur scalaire et il aurait été plus simple pour ce dernier de récupérer deux adresses de DECOD. Néanmoins, par manque de temps, la prédiction de branchement n'aura pas été implémenté sur SS2.

Pour qu'IFETCH soit en mesure de transmettre ces adresses à DECOD, nous avons remplacé la fifo **if2dec** par deux fifos circulaires de deux places, **if2dec_s1** et **if2dec_s2**.

Afin de gérer les dépendances de données dans DECOD, ce dernier génère un signal **PRIOR_PIPELINE_RD** qui indique comment les instructions doivent être chargées.

En effet, IFETCH a deux sorties **INSTR_RI_S1** et **INSTR_RI_S2** qui permettent d'envoyer l'instruction à décoder à l'étage DECOD.

Dans le cas où **PRIOR_PIPELINE_RD** vaut 0, on se trouve dans le cas dit "normal" où S1 est prioritaire sur S2. Si ce même signal vaut 1, on se trouve dans le cas où S2 est prioritaire sur S1. Dans ce cas, le signal **PRIOR_PIPELINE_RD** va indiquer à IFETCH d'inverser le chargement des instructions sur les ports de sorties **INSTR_RI_S1** et **INSTR_RI_S2** en utilisant le composant que nous avons appelé **splitter** et qui est réalité très proche d'un multiplexeur.

La table de vérité du Splitter est la suivante :

if2dec_s1	if2dec_s2	PRIOR_PIPELINE_RD	INST_R1_S1	INST_R1_S2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

On peut ainsi traduire la table de vérité par la relation suivante :

$$\begin{aligned} INST_R1_S1 &= PRIOR_PIPELINE_RD.if2dec_s2 + !(PRIOR_PIPELINE_RD).if2dec_s1 \\ INST_R1_S2 &= PRIOR_PIPELINE_RD.if2dec_s1 + !(PRIOR_PIPELINE_RD).if2dec_s2 \end{aligned}$$

3.5.1.2 DECOD

L'étage DECOD qui est maintenant en mesure de décoder deux instructions en un seul cycle doit être en mesure de traiter les dépendances de données du type RAW (read after write). En effet, si l'une des adresses source de l'instruction S2 est la même que le registre destination de l'instruction S1, il ne faut pas charger l'instruction S2.

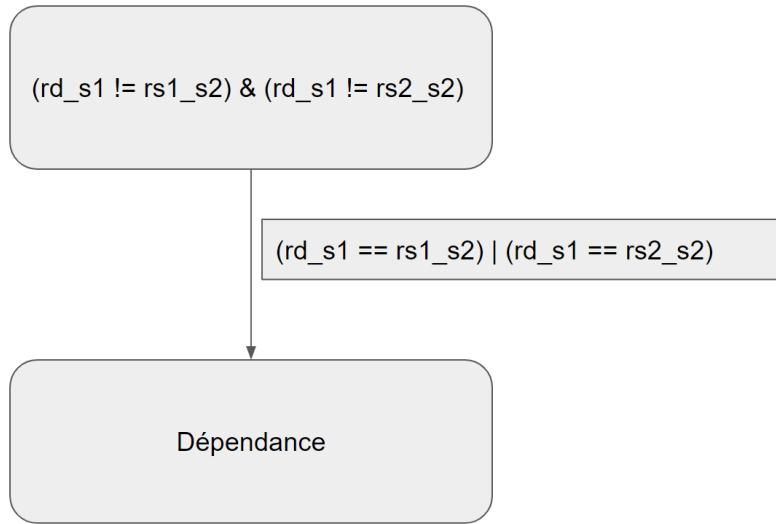


Figure 13: Traitement des dépendances de données

Si une dépendance est détectée, il va donc falloir :

- Charger un NOP dans la fifo dec2exe_s2,
- Pop uniquement la fifo if2dec_s1 si la priorité actuel est sur S1, sinon Pop uniquement if2dec_s2,
- Inverser la priorité de chargement des instructions dans IFETCH en changeant la valeur de PRIO_PIPELINE_RD

En pratique avec un programme comme celui-ci :

```

1 add x2,x0,x1
2 add x3,x2,x0
3 sll x8,x0,x1

```

On aura :

	cycle i	cycle i+1
S1	add x2,x0,x1	add x3,x2,x0
S2	NOP	sll x8,x0,x1

Nous avons élargi ce mécanisme d'inversion du chargement des instructions en cas de dépendance de données dans les cas des instructions csr, en effet dans le cas où deux instructions CSR seraient décodées au même cycle, la 2ème ne sera pas chargée et sera traitée exactement comme si une dépendance de données avait été détectée. Nous avons fait cela afin de ne pas gérer tout type de problème sur des CSR manipulant la même adresse de registre, permettant ainsi de garantir l'atomicité de ces instructions.

Enfin, le mécanisme d'incrémentation de PC aura aussi été modifié puisque DECOD ne génère plus une seule adresse, mais deux, il génère donc une adresse incrémentée de 4 par rapport au PC relatif de la 1ère instruction et un PC incrémenté de 8 par rapport à la 1ère instruction (autrement dit PC+4 et PC+8).

3.5.1.3 Accès mémoire et autres étages

Les étages EXE et WBK sont quasiment similaires à ceux du scalaire hormis pour la partie bypass qui sera détaillé dans la section 3.5.1.4. Afin d'optimiser les accès mémoires, nous avons choisi de **rendre possible les accès mémoires simultanés** sur S1 et S2.

Néanmoins, faire cela pose un problème majeur, en effet lorsque l'on fait :

- un store sur A1 dans S1
- un load sur A1 dans S2

Il faut s'assurer que le processeur traite d'abord la requête de store, puis la requête de load. Si l'on utilise des caches il faut que ces derniers soit en mesure de traiter la requête prioritaire (celle de S1) avant de traiter la moins prioritaire (celle de S2). En pratique, la résolution de ce problème n'a pas été traité puisque les caches n'ont pas été modifiés pour le SS2 étant donné que Kevin qui était en charge des caches travaillait sur la prédiction de branchement au moment où Louis s'est occupé du SS2.

Cependant ce problème n'apparaît pas en simulation lorsque les caches sont désactivés. En effet sans cache, la mémoire simulée par le core_tb via une map c++, qui est une mémoire parfaite puisqu'elle répond en un seul cycle. De plus le core_tb a été construit de sorte à ce qu'il réponde toujours d'abord aux requêtes de MEM_S1 avant de répondre à celle de MEM_S2 et comme S1 est toujours prioritaire sur S2 cela permet d'éviter le problème mentionné ci dessus.

Il faudrait cependant traiter ce problème différemment dans le cadre d'une implémentation sur ASIC/FPGA puisque la mémoire possède une latence.

Enfin, l'étage MEM est également en charge du traitement des exceptions :

Nous avons choisi de créer le squelette nécessaire à la désactivation des accès mémoires simultané et donc à l'inversion de priorité entre MEM_S1 et MEM_S2, néanmoins les signaux existent, mais ne sont jamais utilisés, S1 sera donc toujours prioritaire sur S2 dans notre implémentation.

Ainsi, si une exception est détectée dans S1 elle sera prioritaire sur celle dans S2. Afin d'éviter d'avoir une gestion indépendante des exceptions, nous avons choisi de transmettre tous les signaux d'une exception détectée dans S1 directement dans S2. C'est alors ce dernier qui va se charger d'accéder aux bancs de registre CSR et de mettre à jour les valeurs des registres avec les valeurs calculés dans S1 ou dans S2, selon où s'est produit l'exception. Enfin, c'est également M2 qui se chargera de transmettre un signal aux autres étages les informant qu'une exception a eu lieu, ces derniers n'étant néanmoins pas au courant de l'étage dans lequel cela s'est produit.

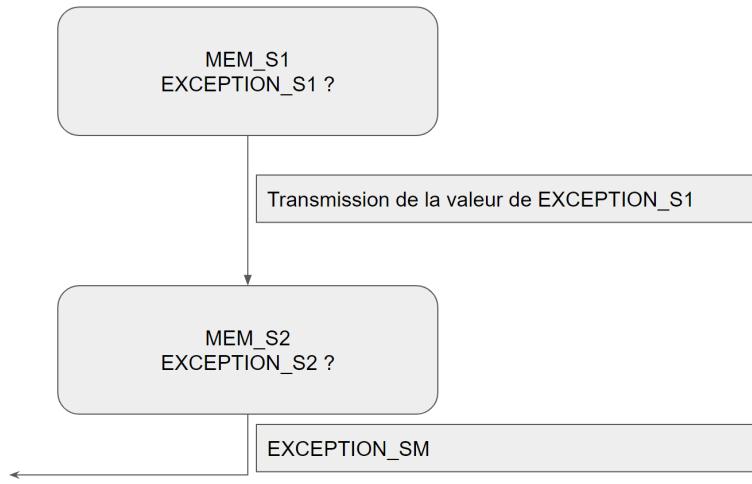


Figure 14: Transmission des signaux d’exception entre les étages MEM_S1 et MEM_S2

3.5.1.4 Bypass

Pour assurer un fonctionnement optimal du SS2, il faut augmenter le nombre total de bypass à 12.

I	D	E1	M1	W1						
		E2	M2	W2						
I	D	E1	M1	W1						
		E2	M2	W2						
I	D	E1	M1	W1						
		E2	M2	W2						
I	D	E1	M1	W1						
		E2	M2	W2						
I	D	E1	M1	W1						
		E2	M2	W2						
	I	D	E1	M1	W1					
		E2	M2	W2						
		I	D	E1	M1	W1				
			E2	M2	W2					
			I	D	E1	M1	W1			
				E2	M2	W2				
				I	D	E1	M1	W1		
					E2	M2	W2			

Figure 15:

- Bypass sur EXE :

- E1 → E1
- E1 → E2
- E1 → D
- E2 → E1
- E2 → E2
- E2 → D

- Bypass sur MEM :

- M1 → E1
- M1 → E2
- M1 → D
- M2 → E1
- M2 → E2
- M2 → D

Nous n'avons pas mis de bypass en entrée de MEM afin d'éviter d'allonger le chemin critique de cet étage qui est déjà long, en raison des accès mémoire.

Le fonctionnement des bypass est assez similaire à celui que l'on va trouver dans le scalaire simple. Un point que l'on a cependant dû ajouter est le cas suivant :

Si l'adresse de destination de EXE_S1/MEM_S1 est identique à l'adresse de destination de EXE_S2/MEM_S2 et que l'on doit bypass une donnée, alors on prend le bypass de l'étage S2 puisque c'est cette donnée-là qui sera la plus récente. Encore une fois, nous rappelons que la priorité de nos étages étant fixe, ce sera toujours S2 qui aura les données les plus récentes, dans le cas où cette priorité constante serait supprimée, il faudrait vérifier quel est le pipeline avec la donnée la plus récente pour gérer ce cas bien précis.

3.5.1.5 Benchmark et conclusion sur l'implémentation du SS2

L'implémentation du SS2 nous aura permis de détecter certains problèmes qui nous avaient échappé sur le cœur scalaire comme lorsque l'on a :

```

1 add x3 ,x2 ,x1
2 csrrc x2 , csr_register , x3

```

En effet, dans ce cas le registre x3 va être bypass, néanmoins l'instruction csrrc effectue un et logique entre la valeur du registre statut et l'inverse de la valeur dans x3 (inverse au sens inversion des bits).

C'est pourquoi il faut bypass $\bar{x}3$ et non pas simplement la valeur dans x3.

Enfin, dans l'optique de mesurer le gain en performance de l'implémentation SS2 et de l'implémentation scalaire, nous avons automatisé la mesure du nombre de cycles nécessaires à l'exécution d'un programme. Il aurait été intéressant de faire des mesures de CPI, mais nous n'avons pas eu le temps d'automatiser son calcul. En effet, le calcul du nombre d'instructions exécutées dans un programme est un peu plus compliqué qu'un simple mesure de cycles.

Nous avons tout de même effectué cette mesure du nombre de cycles à l'aide d'un script shell générant un fichier que nous avons ensuite placé dans excel afin d'obtenir les résultats suivants :

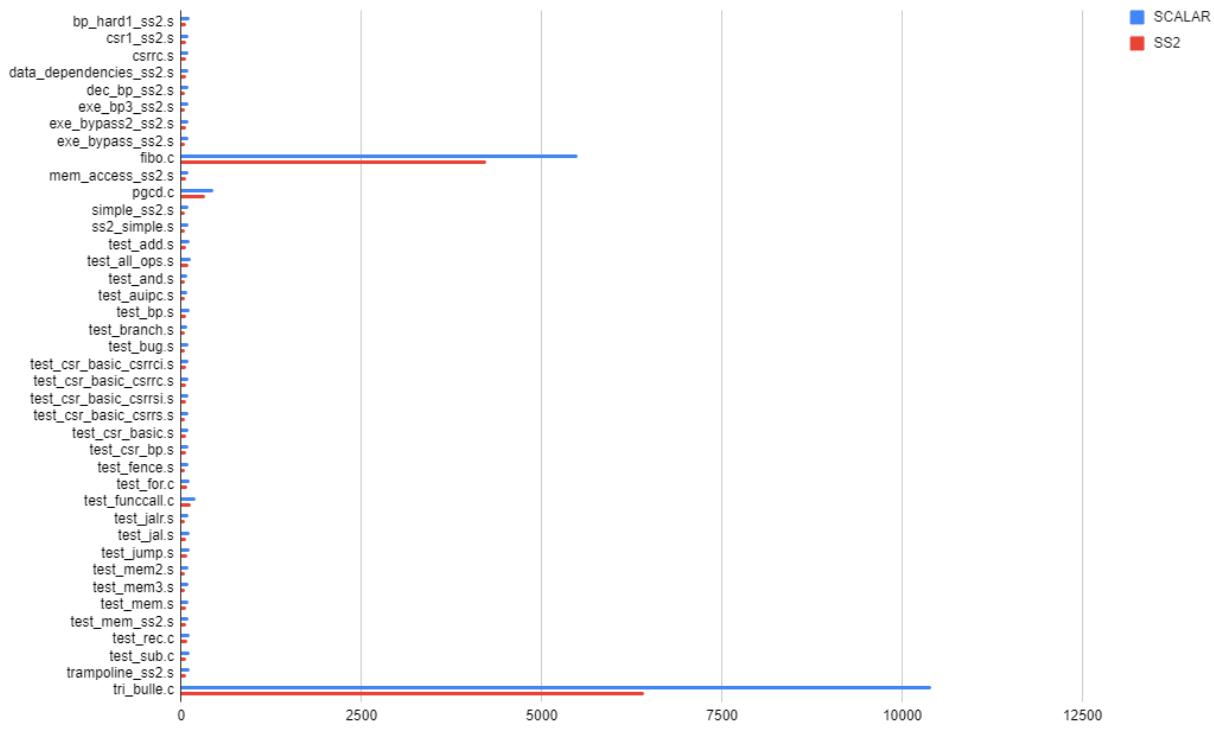


Figure 16: Comparaison du nombre de cycles par programme sur des tests custom

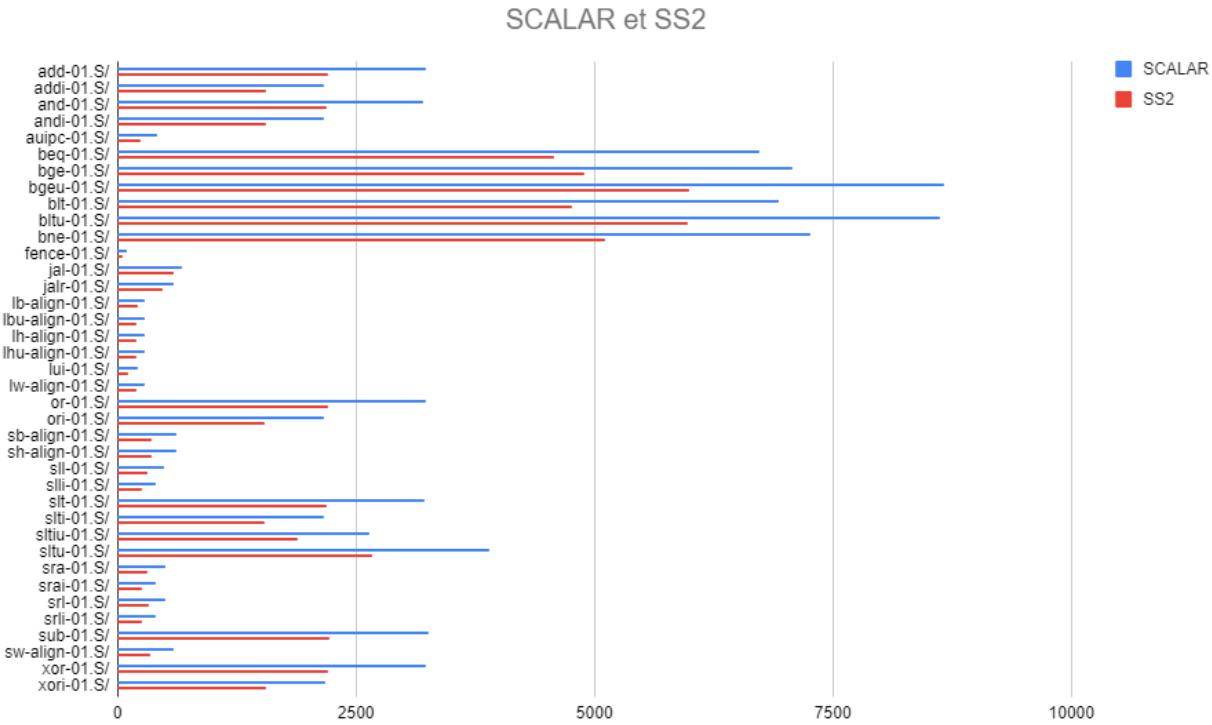


Figure 17: Comparaison du nombre de cycles par programme sur la suite riscof

Sur nos tests custom le SS2 est en moyenne 1,62 fois plus rapide tandis que sur les tests de la suite riscof le gain descend à 1,50. Cet écart est tout à fait logique dans la mesure où nos tests assemblateurs custom sont assez courts, les tests c étant un peu plus long. Les tests riscof sont quant à eux beaucoup plus long et contiennent donc beaucoup plus de dépendance de données. En effet, dans nos programmes il y a peu de dépendance de données, ce qui va donc impliquer que le SS2 tend à être deux fois plus rapide, car il n'y aura pas de gel sur la 2ème instruction décodée dans le cycle DECOD.

Si l'on prend par exemple le test fibo.c, on a un gain de 1.30, ce qui est bien plus faible que la moyenne obtenue. Le compilateur (gcc) ne prend pas en compte le fait que l'on utilise une architecture super-scalaire et le code n'est donc pas optimisé pour ce genre d'architecture, d'où le fait que ce gain soit beaucoup plus réduit. En effet, le code contient beaucoup de dépendances de données ce qui entraîne de nombreux gels sur le chargement de la 2ème instruction dans DECOD.

Nous n'avons pas effectué de mesure de chemin critique sur le modèle SystemC mais il pourrait être intéressant de le faire par la suite pour voir si l'implémentation super-scalaire présente vraiment un gain.

3.5.2 Prédiction de branchement

Lorsqu'un branchement a lieu dans DECOD et que ce dernier réussit, un cycle d'horloge va être perdu. En effet, il va falloir flusher l'instruction qui avait été chargé dans la fifo if2dec et aller chercher l'instruction se trouvant à l'adresse du branchement.

La prédiction de branchement est un mécanisme qui permet d'éviter de perdre ce cycle en anticipant si le branchement va réussir et ainsi en chargeant les instructions correspondantes.

Notre prédition de branchement utilise 2 mécanismes différents [25]:

- **2-bit Saturating Counter**
- **Return address stack**

3.5.2.1 2-bit Saturating Counter

Ce mécanisme utilise une table associative qui a pour clef l'adresse d'un branchement et pour valeur une adresse et un état. L'adresse est celle où le branchement saute s'il réussit et l'état est le résultat de la prédition, c'est-à-dire si elle a échoué ou réussi.

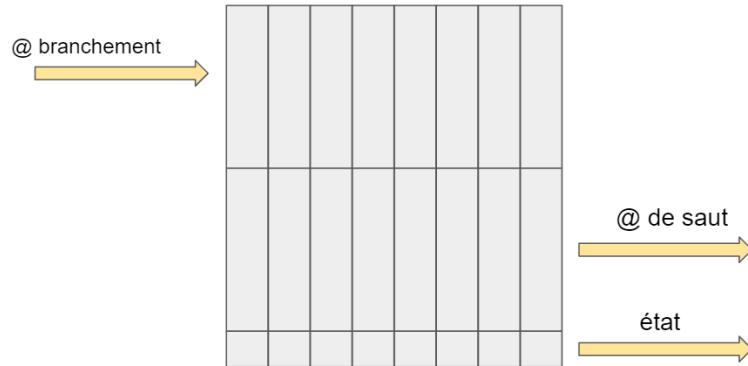


Figure 18: Table associative du prédicteur de branchement

L'état stocké dans le tableau associatif correspond à l'un des états de la machine à état [figure 19].

L'état de cette machine change lorsque le branchement réussit ou échoue. Un branchement va être considéré comme pris lorsque son état vaut **weakly taken** ou **strongly taken**, tandis qu'il ne sera pas pris lorsqu'il sera dans l'état **strongly not taken** ou **weakly not taken**.

Ifetch détecte que l'instruction reçue est un branchement, cette dernière va être envoyée à Decod comme d'habitude. Mais avant de l'envoyer dans Decod, Ifetch "regarde" si cette instruction est stockée dans son tableau associatif.

Si c'est le cas, alors une fois l'instruction envoyée à Decod, si la prédition indique que le branchement va réussir, Ifetch va ignorer la valeur de PC contenu dans dec2if et va aller chercher l'instruction à l'adresse du branchement.

Ainsi si le branchement réussit aussi dans Decod, il ne sera pas nécessaire de flusher l'instruction dans if2dec et le PC envoyé dans dec2if sera celui de branchement + 4.

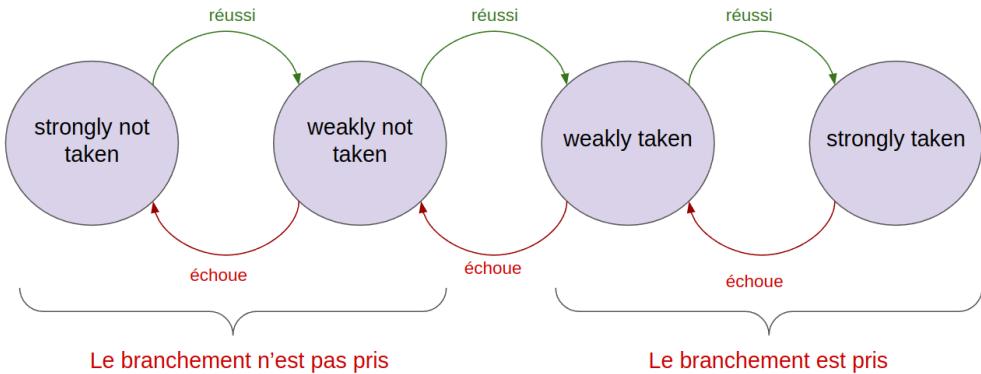


Figure 19: Graphe d'états de la prédiction de branchement

3.5.2.2 RAS(return address stack)

La prédiction de branchement présentée dans le paragraphe 3.5.2.1 ne permet pas de prédire les retours de fonctions. En effet, cette adresse de retour peut varier puisque par convention l'instruction RET lit l'adresse de saut dans le registre x1.

Pour palier à ce problème, nous avons ajouté dans Ifetch une pile appelée **RAS(return address stack)** qui va stocker les adresses de retour des fonctions.

On peut lire dans la spec [8] :

"Une instruction jal devrait empiler l'adresse de retour sur la return address stack seulement quand rd=x1/x5. Jarl instruction devrait empiler/dépiler la RAS comme montré dans la table [figure 20]"

<i>rd</i>	<i>rs1</i>	<i>rs1=rd</i>	RAS action
<i>!link</i>	<i>!link</i>	-	none
<i>!link</i>	<i>link</i>	-	pop
<i>link</i>	<i>!link</i>	-	push
<i>link</i>	<i>link</i>	0	push and pop
<i>link</i>	<i>link</i>	1	push

Figure 20: Conditions pour empiler et dépiler la RAS

Ce tableau recense les conditions d'empilement et de dépilement de la RAS, on peut par exemple voir que dans le cas où l'on a un registre destination et un registre source utilisé avec l'instruction jalr et que ce registre source est égal au registre destination et qu'il vaut 0, alors il faudra à la fois empiler et dépiler.

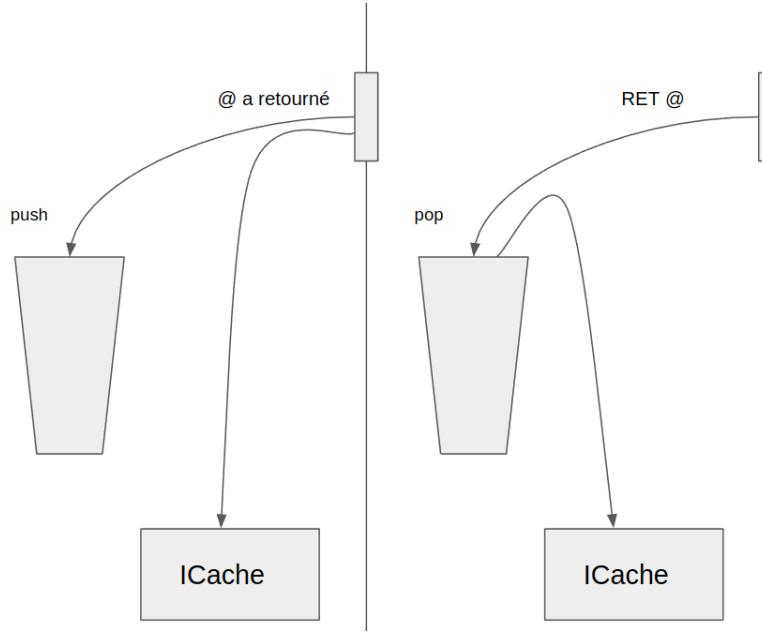


Figure 21: Implémentation du RAS dans ifetch

3.5.2.3 Résultats obtenus

Afin de déterminer le gain obtenu grâce à la prédition de branchement, nous avons effectué des comparaisons entre le nombre de cycles nécessaire pour exécuter un programme avec et sans prédition de branchement. Les résultats sont présentés dans le tableau 3.5.2.3.

Programme de test	Nb cycles sans opt.	Nb cycles avec pred. de branch. sans RAS (PB)	% gain	Nb avec PB et RAS	% gain	Nb branch réussits
<i>div_2.c</i>	403	378	6.2%	378	6.2%	31
<i>fibbo.c</i>	5499	5223	5.0%	5047	8.2%	677
<i>pgcd.c</i>	49283	48396	1.8%	48396	1.8%	1783
<i>factoriel.c</i>	326	323	0.9%	323	0.9%	11
<i>test_funcall.c</i>	197	194	1.5%	192	2.5%	13
<i>test_jump.s</i>	120	119	0.8%	119	0.8%	9
<i>tri_bulle.c</i>	10405	10238	1.6%	10238	1.6%	456

*Les tests ont été réalisés sans les caches.

On constate donc que l'on gagne environ 2,6% de cycles avec la prédition de branchement sans le mécanisme de pile. Avec le mécanisme de pile, on monte cette fois-ci à 3,2%. On remarque cependant que l'on gagne près de 3% de cycles sur la suite de fibonacci récursive avec le mécanisme de pile. Cela semble cohérent puisque le programme étant récursif, il fait un grand nombre d'appels de fonction et donc un grand nombre de retour de fonction.

3.6 Protocole de validation

3.6.1 Tests basiques

Pour valider notre implémentation, nous avons fonctionné comme suit :

- Dans un premier temps, nous avons réalisé des tests benchs (cf [GitHub](#)) pour chacun des étages afin de vérifier leur fonctionnement. Ces tests consistaient à envoyer des signaux avec des valeurs aléatoires dans l'étage testé et de regarder ce que nous obtenions en sortie.
- Dans un second temps, nous avons compilé quelques programmes assembleur assez simples avec une ou deux instructions, nous avons ensuite complexifié les programmes et nous avons conçu des tests comportant toutes les instructions d'un même type. Nous avons par exemple un test qui s'appelle `test_all_ops` qui test toutes les instructions de type I
- Enfin, nous avons écrit des programmes C comme la suite de Fibonacci ou un algorithme de PGCD que nous avons compilé et nous avons vérifié que tout fonctionnait correctement.

Afin de réaliser tous nos tests, nous avons utilisé Gt_kwave pour visualiser les signaux de nos entités, nous avons en effet créé des fonctions `trace()` dans chacun de nos fichiers qui tracent tous les signaux d'une même entité dans un fichier `.vcd` visualisable dans Gt_kwave.

Enfin, pour compiler directement du code, nous avons utilisé la librairie C++ ELFIO qui permet de parser un fichier ELF. Pour utiliser cette librairie, notre fichier `core_tb.cpp` prend comme argument un fichier `.s` ou `.c`, il appelle ensuite le compilateur RISC-V et produit un `objdump` ainsi qu'un exécutable.

Cet exécutable est ensuite parsé à l'aide de ELFIO et l'on récupère ainsi les instructions qui sont stockées dans notre RAM, qui est simulée par une map qui forme un couple (adresse, instruction).

Nous avons également les segments `_bad` et `_good` sur lesquels on saute en fin de programme pour voir si tout s'est correctement déroulé.

Voici un exemple d'un de nos programmes de test, il s'agit de la suite de Fibonacci récursive :

```

1 extern void _bad();
2 extern void _good();
3
4 __asm__(".section .text");
5 __asm__(".global _start");
6
7 __asm__("_start:");
8 __asm__("addi x2,x0, 0x100");
9 __asm__("addi x1,x1, 4");
10 __asm__("sub x2, x2, x1");
11 __asm__("jal x5, main");
12
13 int fib(int n) {
14     if (n == 0) {
15         return 0;
16     }
17     else if (n == 1) {
18         return 1;
19     }
20     else {
21         return fib(n-1) + fib(n-2);
22     }
23 }
24
25
26
27 int main() {
28     if (fib(10) == 55) {
29         _good();
30     }
31     else {
32         _bad();
33     }
34 }
35 __asm__("nop");
36 __asm__("_bad:");
37 __asm__("    add x0, x0, x0");
38 __asm__("_good :");
39 __asm__("    add x1, x1, x1");
40

```

3.6.2 Gestionnaire d'exceptions

Lorsque nous avons commencé à implémenter la partie machine, nous avons dû modifier nos codes de test afin d'avoir un code faisant office de gestionnaire d'exception. En effet, lorsqu'une exception est détectée dans MEM on va charger la valeur de MTVEC dans PC, registre qui contient l'adresse où se trouve le gestionnaire d'exception. Voulant nous rapprocher au maximum du MIPS, nous avons placé cette adresse à 0x8000 0000, le problème étant que rien ne se trouvait initialement à cette adresse.

Pour remédier à ce programme, nous avons ajouté deux linker script **app.ld** et **kernel.ld** définissant une section text et une section Kernel, c'est en effet Mr. Franck Wajsbürt qui nous a conseillé de faire comme ça afin de faire un distinction entre script user et script machine.

Ces deux linker script sont identiques à la différence près que app.ld se termine par une instruction INPUT(kernel).

L'intérêt d'avoir deux linker script est le suivant, le fichier kernel.ld permet de générer un fichier objet nommé **kernel** qui contient le code de reset et le gestionnaire d'exception que l'on va linker avec le fichier user lors de la compilation du fichier test. Lors de la compilation du test ou autrement dit du fichier user on va utiliser le fichier app.ld, l'instruction INPUT(kernel) permettant de forcer le compilateur à générer un fichier elf présentant à la fois le code user et le code machine.

```

1 SECTIONS
2 {
3     . = 0x10054 ;
4     seg_text :
5     {
6         *(.text)
7     }
8     . = 0x80000000 ;
9     seg_reset :
10    {
11        *(.reset)
12    }
13    . = 0x81000000 ;
14    seg_kernel :
15    {
16        *(.kernel)
17    }
18 }
19 INPUT(kernel)
20
```

3.7 Suite de tests officielle riscof

Pour mieux valider notre processeur, nous avons voulu utiliser une suite de test plus complète et externe (pour éviter d'être biaisé dans la réalisation de nos tests). Nous avons choisi d'utiliser la Riscv-V Compatibility Framework. Il s'agit d'un programme de test qui compare une "signature" (un segment de la mémoire) entre deux implémentations de RISC-V sur toute une batterie de tests.

Nous avons fait ce choix, car la suite est très complète (plusieurs centaines de tests unitaires pour la plupart des instructions), et qu'elle a une valeur "officielle". En effet, bien qu'elle ne soit pas développée par la fondation risc-v, mais par une entreprise privée (incore), valider cette suite de test est un pré-requis pour être approuvé par la fondation risc-v et pouvoir utiliser la marque déposée "risc-v".

Pour faire fonctionner cette suite de tests, nous avons dû réaliser un "plugin" permettant de brancher le framework de test avec notre programme, ainsi qu'un outil pour dumper la mémoire dans un fichier.

Il nous a aussi fallu modifier notre test-bench, pour lire les sections et les symboles définis dans les tests, tels que le début du test, la fin du test, le début de la zone mémoire à dumper et la fin de celle-ci.

Nous avons ensuite choisi l'implémentation "spike" comme implémentation de référence pour les tests.

Ces tests nous ont permis de corriger de nombreux bugs : beaucoup d'instruction avec des cas limites qui ne réagissaient pas correctement.

4 SoC

Après avoir fini l'optimisation des branchements au travers de la prédition de branchements, en simultanée du développement du cœur SS2, il a été décidé de modéliser un SoC complet en SystemC et en VHDL. En effet, l'ajout d'un bus système était nécessaire à l'ajout de périphérique pour l'implémentation sur carte FPGA. La première étape de cette implémentation a été le choix d'un protocole BUS.

Après avoir étudié plusieurs types de Bus (PiBus, AXI4, Avalon, etc..), nous avons décidé de nous tourner vers le protocole Wishbone B4 que nous avons découvert lors des conférences FSIC2022.

Ce protocole est open source, simple à implémenter et très bien documenté [17]. La spécification nous offre différents choix pour le type d'interconnexion, par exemple "Point-to-Point" ou "crossbar switch". Nous avons choisi "shared bus" car c'est ce qui se rapprochait le plus du PiBus que nous avons étudié en 2ème semestre de M1 dans le cadre de l'UE Multi.

Nous avons choisi un système **round-robin** pour l'arbitre, ce qui va nous aider à bien partager le bus entre les coeurs. L'arbitre va aussi, selon l'adresse envoyée au bus, choisir la cible qui va traiter les accès mémoires.

Nous avons placé un total de 2 initiateurs (2 coeurs RiVer) et une cible (la RAM) sur le bus.

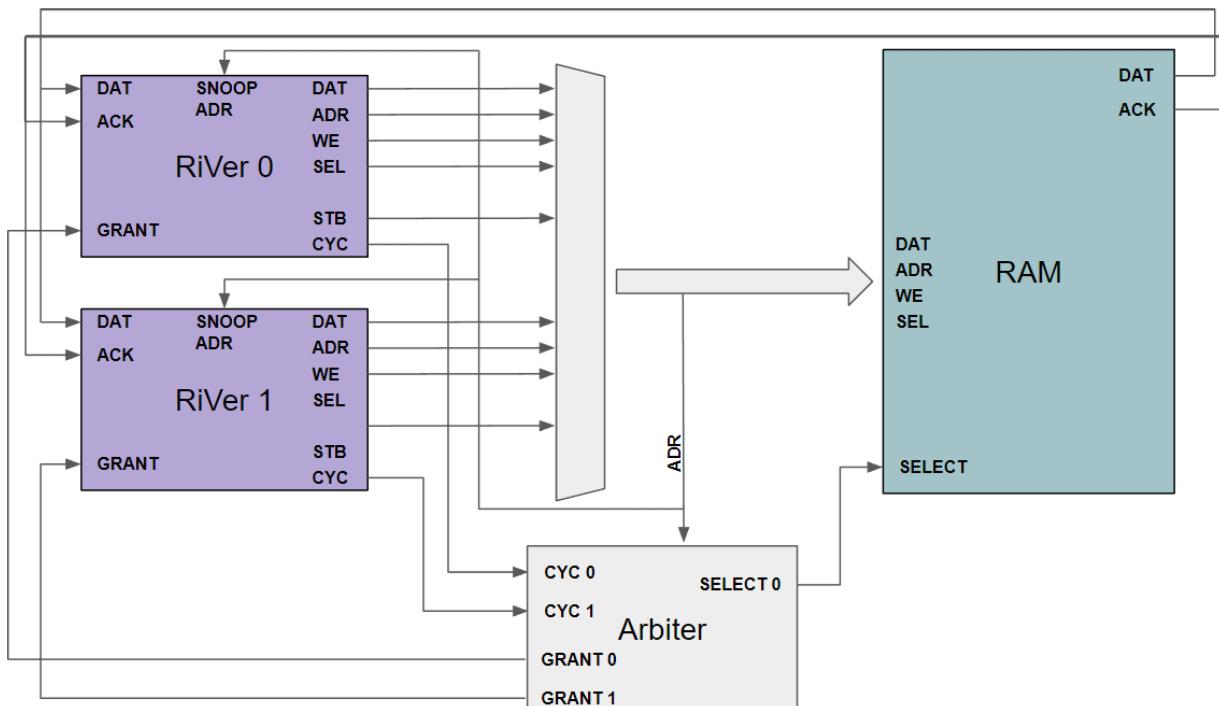


Figure 22: Schéma du SoC

4.1 Master RiVer

L'initiateur RiVer est composé des éléments suivants :

1. Un cœur RiVer RV32IM
2. 2 caches, à savoir un cache d'instruction et un cache de données
3. Le wrapper

Les caches contiennent un mécanisme "snoop", permettant d'assurer la cohérence des données, qui sera détaillé dans la section 4.1.1.

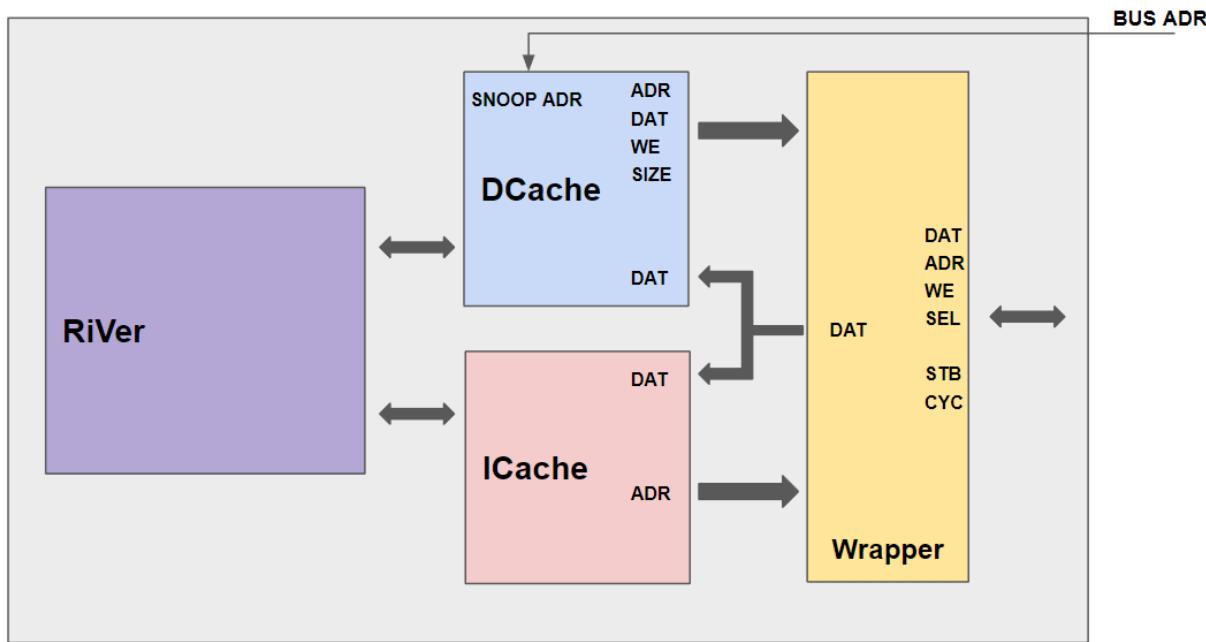


Figure 23: Schéma du Master RiVer

4.1.1 Caches

Suite à de nombreuses discussions entre les membres de notre groupe, nous avons décidé d'implémenter des caches L1 séparés. En effet, les caches permettent de réduire le CPI (cycle par instruction) en limitant les accès mémoire. De plus, nous avons passé près de 2 semestres à étudier le fonctionnement des caches, d'où notre intérêt pour leur implémentation.

4.1.1.1 Cache d'instruction

Le cache d'instruction est un cache composé 256 lignes de 4 colonnes. Chaque case pouvant contenir un mot (32 bits) la capacité totale de notre cache est de 32Kb.

Pour concevoir la MAE (machine à états), nous avons défini 3 états distincts. En effet, après plusieurs tests de différentes machines à état, la plus optimisée que nous avons réussi à mettre en place est composé des états suivants :

- **IDLE**
L'état par défaut qui répond aux requêtes du processeur et en cas de MISS fait une demande au niveau supérieur de mémoire.
- **WAIT MEM**
Cet état attend la réponse de la mémoire.
- **UPDATE**
Cet état est chargé de recevoir les données du bus et de les charger dans les lignes de cache correspondantes.

4.1.1.2 Cache de données

Le cache de données est un cache 2-ways associative, chaque way contenant 128 lignes et 4 colonnes, soit un total d'une capacité de 16Kb par colonne. Ce cache a été conçu en accord avec la politique Write-through. Cette stratégie va nous permettre résoudre quelques problèmes de cohérence de la mémoire. Afin d'améliorer la performance du processeur, nous avons ajouté un buffer-cache à 2 place, ce qui permet de réduire le nombre de stall créés par le cache en cas d'écriture.

La machine à état du cache de données et la même du cache d'instruction à différence que celui du cache de données traite les requêtes d'écriture et lecture :

1. **IDLE**
État par défaut qui répond aux requêtes de lecture du processeur. En cas d'écriture, il écrit la requête dans le buffer cache et en cas de miss de lecture, il écrit la requête dans le buffer cache et il va passer dans l'état WAIT_MEM pour attendre la réponse de la RAM.
2. **WAIT MEM**
Cet état attend la réponse de la RAM (ACKNOWLEDGE) et il met à jour la première donnée qu'il reçoit.
3. **UPDATE**
Cet état est chargé de recevoir les données restantes envoyées dans le bus et de les mettre dans les lignes de cache correspondantes.

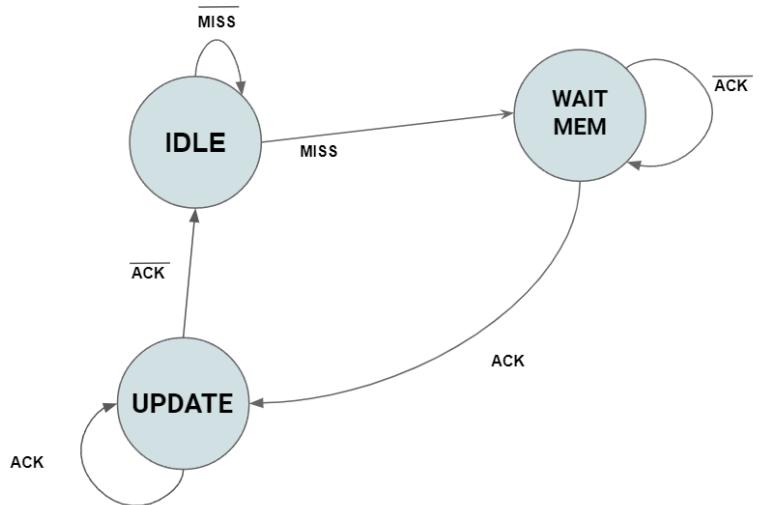


Figure 24: Machine à états du cache d'instruction

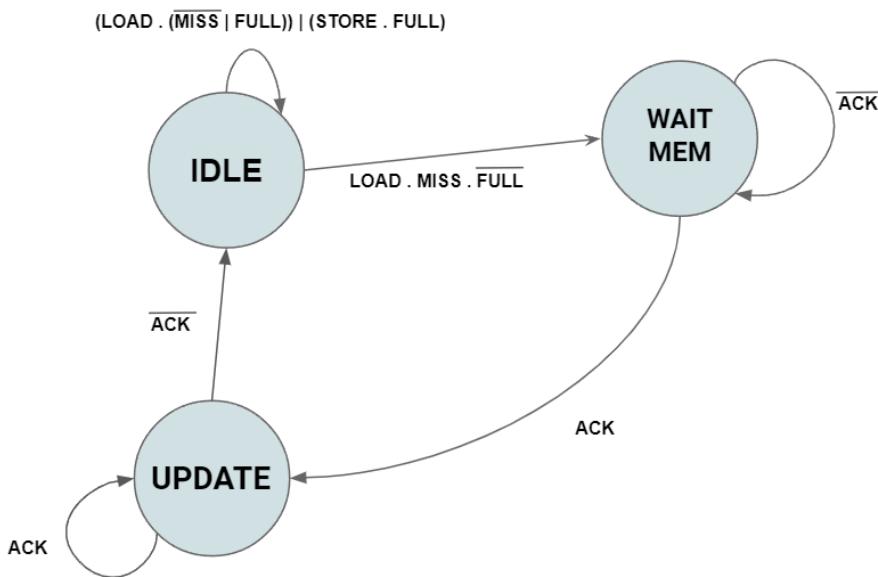


Figure 25: Machine à états du cache de données

4.1.1.3 Snoopy cache

Afin de concevoir un système multicœur, il faut assurer la cohérence de la mémoire.

Notre architecture utilise des caches Write-through avec un buffer d'écriture posté. La présence de ces buffers perturbe la cohérence mémoire. En effet, si un cœur effectue des écritures successives, les données seront présentes uniquement dans son cache L1 le temps que celle-ci soit écrite en mémoire.

Le problème étant que si un autre cœur tente de manipuler ces adresses, il faut qu'il soit au courant que l'autre cœur a effectué une écriture.

C'est pourquoi la présence d'un mécanisme de surveillance est nécessaire. Snoopy permet d'assurer cette cohérence en lisant les adresses présentes sur le bus. Si une écriture est effectuée et que la donnée est présente dans le cache ou dans le buffer-cache, Snoopy sera en charge d'invalider la ligne ou la donnée.

4.1.2 Wrapper RiVer-Wishbone B4

Le rôle du wrapper est double. En effet, il sert dans un premier temps à traduire les requêtes envoyées par les caches en requêtes compréhensibles par le protocole Wishbone.

Dans un second temps, il gère la transmission des données sur le bus, au travers de machine à état.

La machine à état prend en compte le fait que le cache d'instruction et le cache de données peuvent faire faire une requête au même cycle.

Ainsi, cet MAE priviliege les requêtes du cache de données afin de lever plus rapidement un stall du pipeline, libérant ainsi les étages MEM, EXE et DEC.

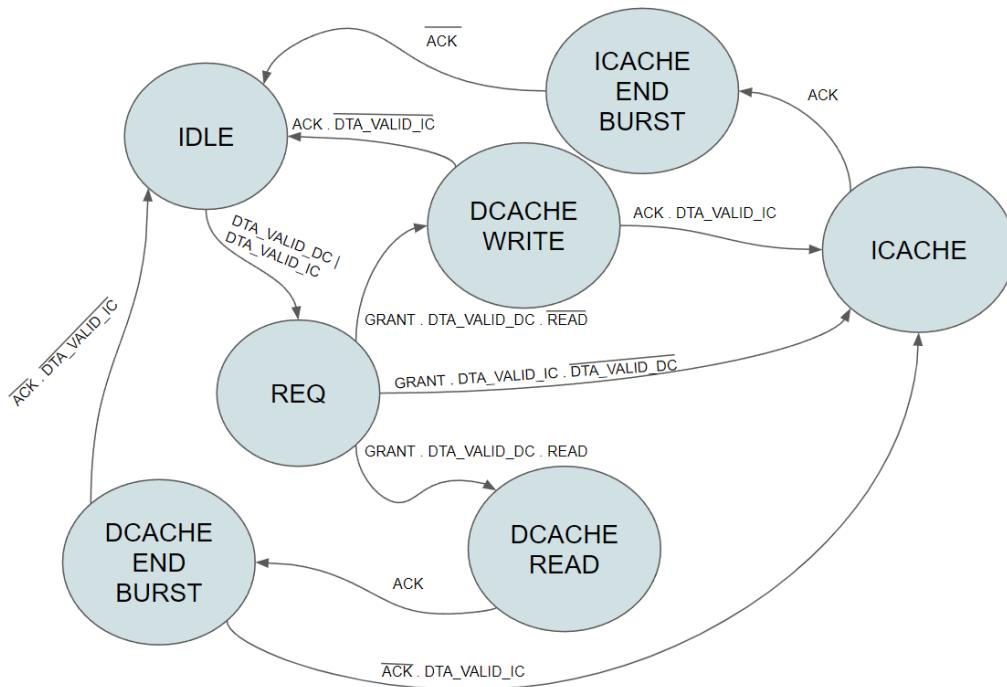


Figure 26: Machine à états du wrapper du RiVer

Les états de cet MAE sont les suivants :

- **IDLE** :
Etat par défaut, en attente que le cache d'instruction ou le cache de données envoie une donnée valide,
- **REQ** :
Demande d'accès au bus par un initiateur,
- **DCACHE WRITE** :
Requête d'écriture du cache de données. Lorsque la cible a répondu (**ACKNOWLEDGE**) et que la donnée est valide passage dans l'état **ICACHE**, sinon retourne dans l'état **IDLE**,
- **DCACHE READ** :
Demande de lecture du cache de données. Une fois la demande faite, il attend la réponse de la cible (**ACKNOWLEDGE**). Lorsque la réponse arrive, il passe dans l'état **DCACHE END BURST**
- **DCACHE END BURST** :
Le wrapper continue à lire les réponses de la cible tant que le signal **ACKNOWLEDGE** est à l'état haut,
- **ICACHE** :
Écriture de la requête du cache d'instruction sur le BUS et attente de la mise à l'état haut du signal **ACKNOWLEDGE** par la cible. Quand le signal est levé, il transmet la première réponse au CACHE d'instruction qui va passer dans l'état **ICACHE END BURST**
- **ICACHE END BURST** :
Le wrapper continue de lire les réponses du SLAVE tant que le signal **ACKNOWLEDGE** est à l'état haut. Quand il revient à l'état bas, il retourne dans l'état **IDLE**.

4.2 Cibles

Le bus SystemC ne comprend que peu de cible, en effet, on ne trouve que la RAM qui contient 2 éléments :

1. Une unordered map qui fait office de ram
2. Un wrapper qui permet de traduire les requêtes pour les rendre compréhensibles par le bus

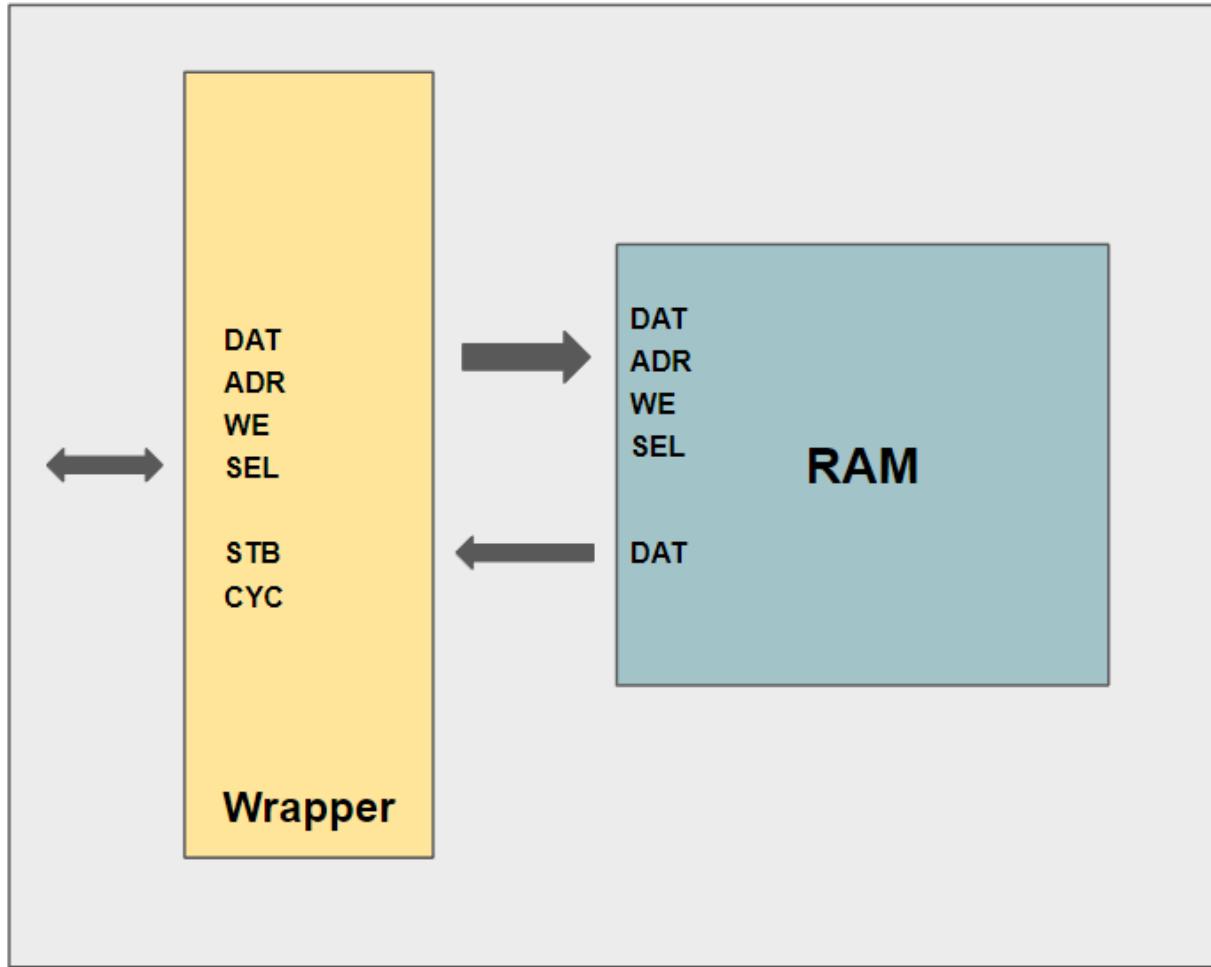


Figure 27: Schéma du Slave RAM

Le wrapper permet de lire les requêtes venant du bus et de les traduire en une ou plusieurs requêtes pour la RAM.

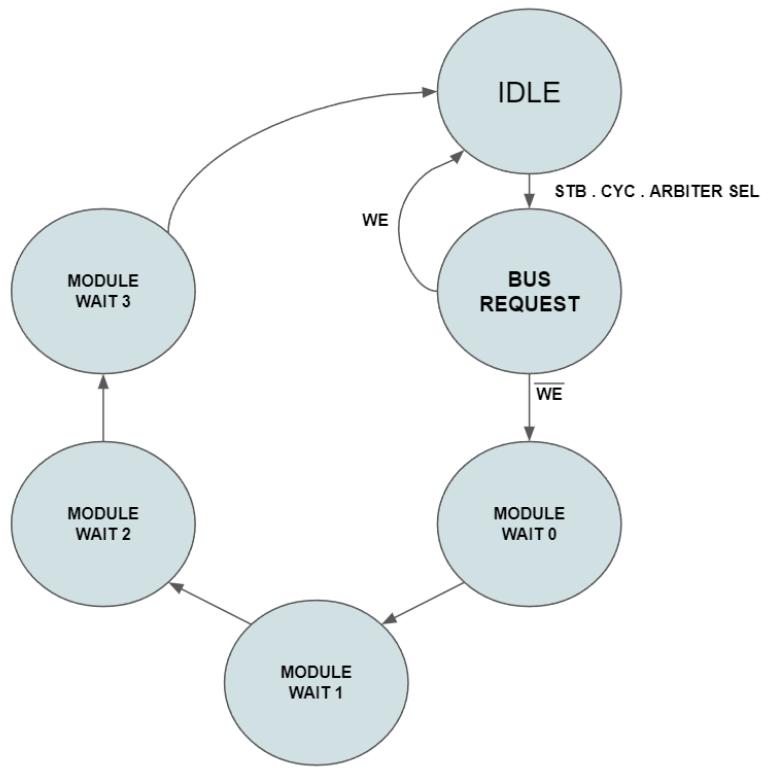


Figure 28: Machine à états du wrapper de la RAM

Le wrapper est contrôlé par une MAE dont voici un bref descriptif :

- **IDLE :**
État par défaut qui attend que le comparateur d'adresse (ou dans notre cas l'arbitre) le choisisse pour répondre à une requête bus,
- **BUS REQUEST**
Traite la requête du bus, deux cas sont possibles :
Lecture Envoie de la première adresse (ADR+0) par le wrapper et passage dans l'état **MODULE WAIT 0**,
Écriture Envoie de la donnée à écrire ainsi que son adresse par le wrapper, puis retour dans l'état **IDLE**,
- **MODULE WAIT 0**
Lecture de la RAM et écriture de la réponse sur le bus, puis envoie de l'adresse suivante ADR+4,
- **MODULE WAIT 1**
Lecture de la RAM et écriture de la réponse sur le bus, puis envoie de l'adresse suivante ADR+8,
- **MODULE WAIT 2**
Lecture de la RAM et écriture de la réponse sur le bus, puis envoie de l'adresse suivante ADR+12,
- **MODULE WAIT 3**
Lecture de la RAM et écriture de la réponse sur le bus, puis retour dans l'état Idle,

5 Implémentation FPGA

Nous avons décidé d'implémenter notre processeur sur FPGA afin de le tester physiquement.

Nous avons pour cela choisi la carte Nexys A7^[20] de Xilinx avec Vivado 2020.2 avec laquelle nous avons l'habitude de travailler en TP de FPGA.

Mais afin de pouvoir réaliser cette implémentation, nous devons dans un premier temps traduire le modèle SystemC en VHDL. En effet, SystemC étant initialement un langage prévu pour faire des simulations plutôt haut niveau et pas des descriptions RTL, nous n'avons pas trouvé d'outil permettant de traduire notre implémentation SystemC en VHDL. C'est pourquoi Samy Attal a eu la responsabilité de traduire manuellement l'ensemble du projet.

Nous allons aborder dans ce chapitre les principales différences entre le modèle VHDL et SystemC ainsi que l'implémentation FPGA.

5.1 Modèle VHDL

Le cœur décrit en VHDL est quasiment similaire, en termes de performance, à celui en SystemC. Nous avons donc un cœur VHDL RV32IM-Zicsr avec prédition de branchements qui a pu être validé par le framework riscosf.

Nous avons utilisé GHDL^[18] comme compilateur VHDL et simulateur ainsi que GTKwave^[19] afin de visualiser les signaux.

Le choix des ces outils a été fait afin de coller à ce qui a pu être fait en SystemC, les noms des signaux sont quasiment identiques, mais bien évidemment des différences dans leurs affectations sont présentes.

Le choix de GHDL est également justifié par le fait qu'il est possible de lier un programme C au testbench VHDL afin notamment de simuler et remplir une RAM en C à partir d'un exécutable (beaucoup plus simple à faire qu'en VHDL).

La différence majeure étant la machine à états du diviseur, en effet, il s'agit d'une machine de Moore dans l'implémentation VHDL, tandis qu'en SystemC il s'agit d'une machine de Mealy (plus rapide) qui est difficilement implantable dans le cœur VHDL (bien que fonctionnelle seule). Le problème dans l'implémentation étant notamment liés aux conditions de start/stop de la MAE et les conditions de push/pop des fifo en post-synthèse.

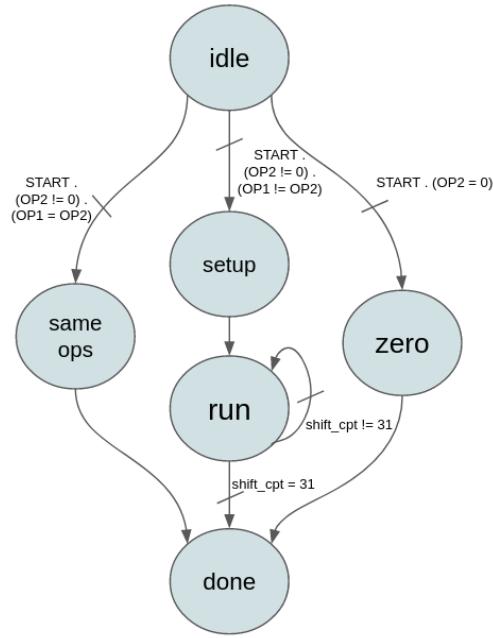


Figure 29: MAE de Moore du diviseur (VHDL)

Nous avons donc un premier état d'attente, dans lequel lorsqu'une instruction de division entre dans l'étage EXE va comparer et vérifier les valeurs des opérandes. Nous avons alors deux cas spéciaux que nous pouvons séparer du cas standard afin de gagner en nombre de cycles :

- l'opérande 2 (le dénominateur) est nul, on passe alors dans l'état "zero" qui va donner comme résultat 0xFFFFFFFF (-1) comme spécifié dans la documentation de RISC-V.
- les deux opérandes sont identiques (et le dénominateur est non nul) : on passe alors dans l'état "same ops" qui va donner comme résultat 1 s'il s'agit d'une division, ou 0 s'il s'agit d'un modulo).

Sinon, on passe dans l'état "setup" qui va charger dans les registres représentés en figure 10 par les bonnes valeurs, et l'état "run" va permettre d'effectuer le calcul (comme dans le modèle SystemC).

5.1.1 Interface C/VHDL

Pour créer une testbench complète, proche de celle que l'on trouve en SystemC nous avons réalisé une interface entre GHDL et un programme C. Cette interface définit notamment des fonctions qui pourront être appelées dans un programme VHDL et qui permettront d'interagir avec une RAM simulée par un tableau d'entier.

Afin de réaliser cette interface nous avons repris un code qui nous avait été fourni par Jean-Lou Desbardieux lors de notre 1er semestre de M1 dans le cadre de l'UE VLSI. Ce programme permet de parser un fichier elf et d'extraire les instructions et adresse de ce dernier.

Nous n'avons en effet pas été en mesure d'utiliser la librairie ELFIO que nous avons utilisé en SystemC puisque cette dernière est écrite en C++ et que nous n'avons pas trouvé de moyen permettant de faire une interface C++/GHDL.

D'autre part l'intérêt du parseur que nous avons utilisé est qu'il est écrit en C pur et ne nécessite rien hormis la bibliothèque **elf.h** qui est une librairie de base de linux.

Une fois le parseur simplifié et adapté pour notre utilisation nous avons écrit le programme **ram_sim.c** qui définit la RAM, place les instructions en mémoire et définit des fonctions, appelables en VHDL, permettant d'écrire ou de lire la RAM.

Nous avons également adapté ce programme pour qu'il soit en mesure de faire tourner le framework riscof nécessaire à la validation de notre implémentation.

5.2 FPGA

5.2.1 Conception du circuit

Notre implémentation FPGA s'est déroulée en plusieurs étapes.

La première étape consistait à tester le bon fonctionnement du processeur à l'implémentation. Nous lui avions associé un tableau mémoire simulant le cache d'instructions (écrit en VHDL) qui contenait en dur les instructions à exécuter ainsi qu'un tableau mémoire simulant le cache de données dont les bits de poids faible d'une des cases mémoire étaient connectés à des LEDS présentes sur la carte afin de pouvoir déboguer. Cette étape est représentée en figure 30.

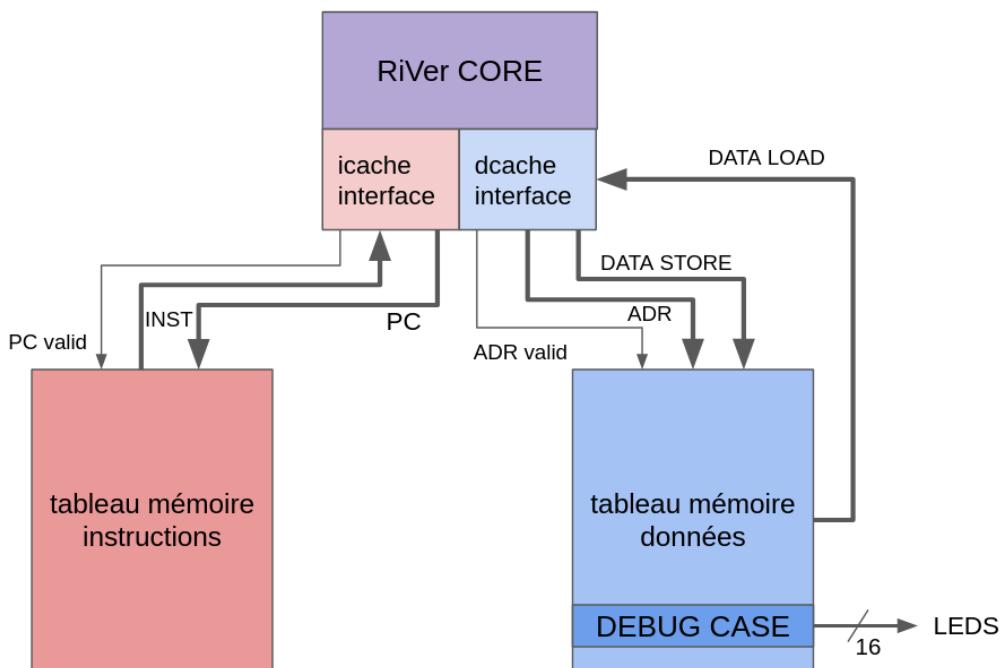


Figure 30: Implémentation basique sur FPGA

C'est ainsi que nous avons pu tester chaque version du processeur en post-synthèse jusqu'à la version finale (RV32I, RV32I-Zicsr, RV32IM-Zicsr, RV32IM-Zicsr avec prédition de branchement).

La deuxième étape, après avoir validé le bon fonctionnement du cœur, consistait à intégrer le processeur sur un bus avec une RAM. Nous avons décidé d'implémenter le bus Wishbone, étant open source et plus simple à mettre en œuvre. Tout en sachant qu'il serait assez simple de convertir les interfaces de nos composants compatibles Wishbone à un autre bus (comme AXI4). On peut ainsi plus facilement porter notre processeur sur différentes cibles et famille de FPGA sans craindre le manque de support du bus AXI4 si notre cible n'est pas de Xilinx, ou Avalon si notre cible n'est pas d'Altera.

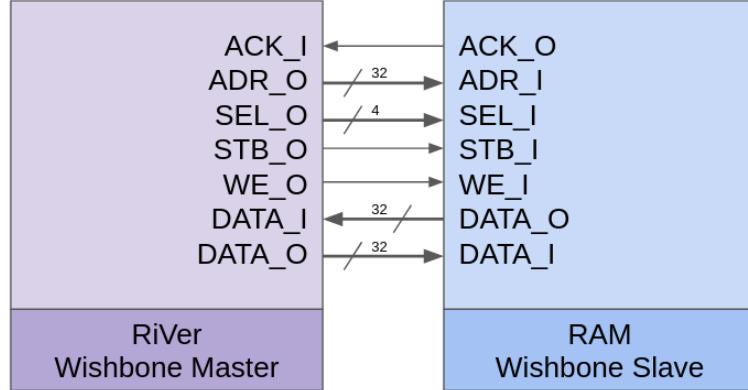


Figure 31: Wishbone CPU et RAM

Sont représentées en figure 31 les interfaces des wrappers Wishbone utilisées dans notre système. À noter que dans le cadre de notre implémentation FPGA, nous avons choisis par simplicité de n'implémenter que le mode single write / single read présent dans la spécification Wishbone B4. Une amélioration vers un mode de transaction pipeliné ou par bloc est tout à fait envisageable afin de tendre vers ce qui a pu être précédemment fait dans le modèle SystemC. La RAM étant à ce stade simplement un tableau décrit en VHDL (plus facile pour le debug).

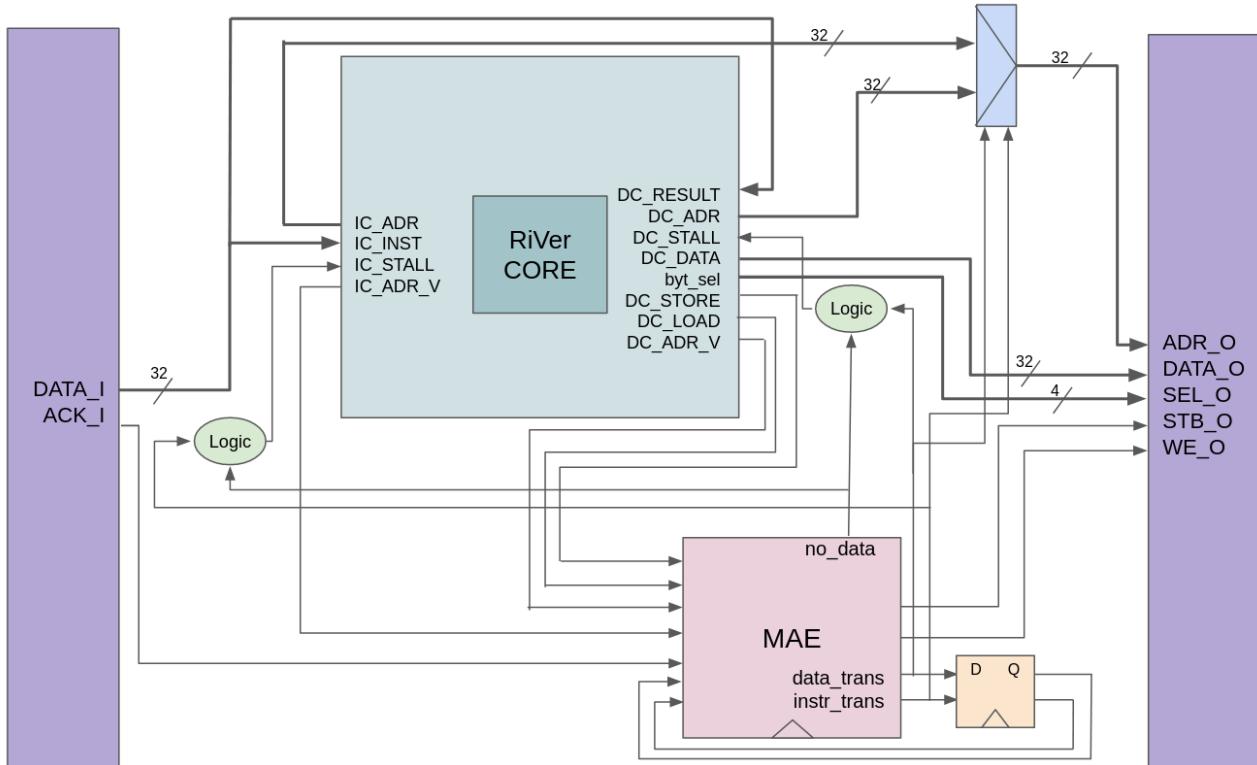


Figure 32: Interface du cœur avec Wishbone

2

La figure 32 représente la façon dont nous avons interfacé notre processeur au bus Wishbone. Il s'agit du wrapper.

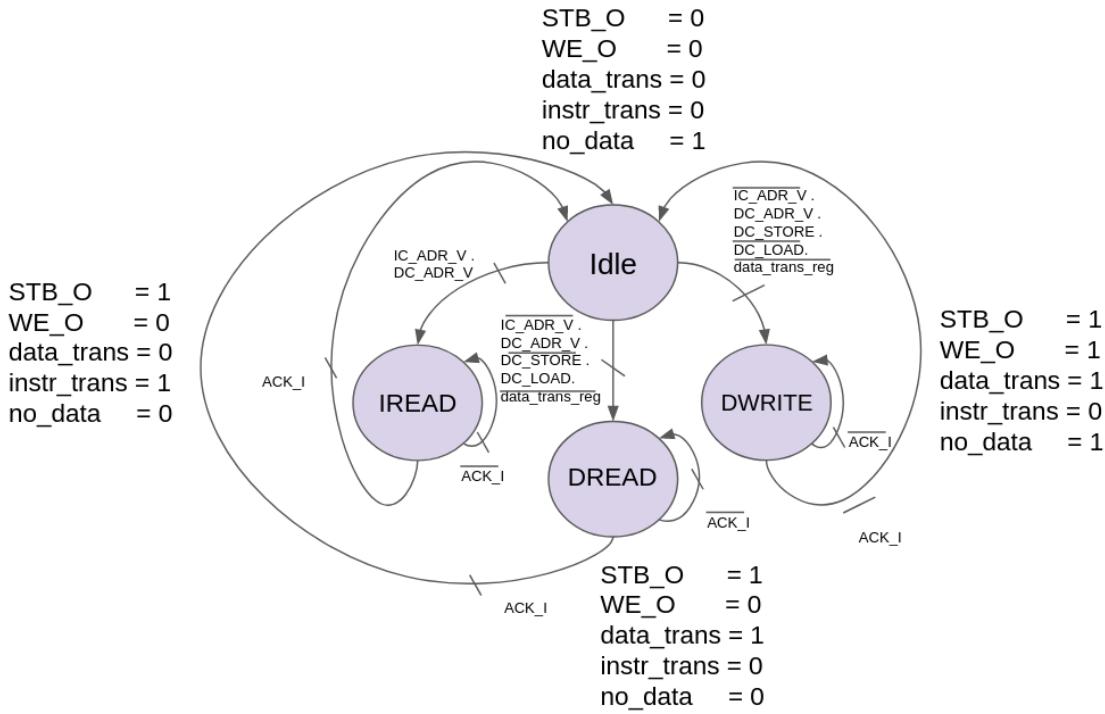


Figure 33: MAE de Moore du wrapper Wishbone du processeur

La MAE, représentée sur la figure 33, contrôle en plus des signaux STB_O (pour initier une transaction) et WE_O (pour commander une écriture) utilisés dans le cadre du protocole Wishbone. Les signaux no_data, data_trans et instr_trans qui servent à indiquer aux interfaces d'ifetch et de mem s'ils sont destinataires ou non de la donnée disponible provenant du bus.

Une fois le bon fonctionnement du bus vérifié avec un master (le processeur) et un slave (la RAM) nous avons décidé de créer et connecter au bus un périphérique qui permettra de contrôler les LEDs directement présentes sur la carte (La Nexys A7 contient seize LEDs ainsi que deux LEDs RGB). Cette IP pourrait être également customisable (par le nombre et le type de leds qu'elle contrôle) afin de pouvoir s'adapter plus facilement à n'importe quelle cible de carte de développement FPGA.

La figure 34 illustre l'implémentation de ce périphérique.

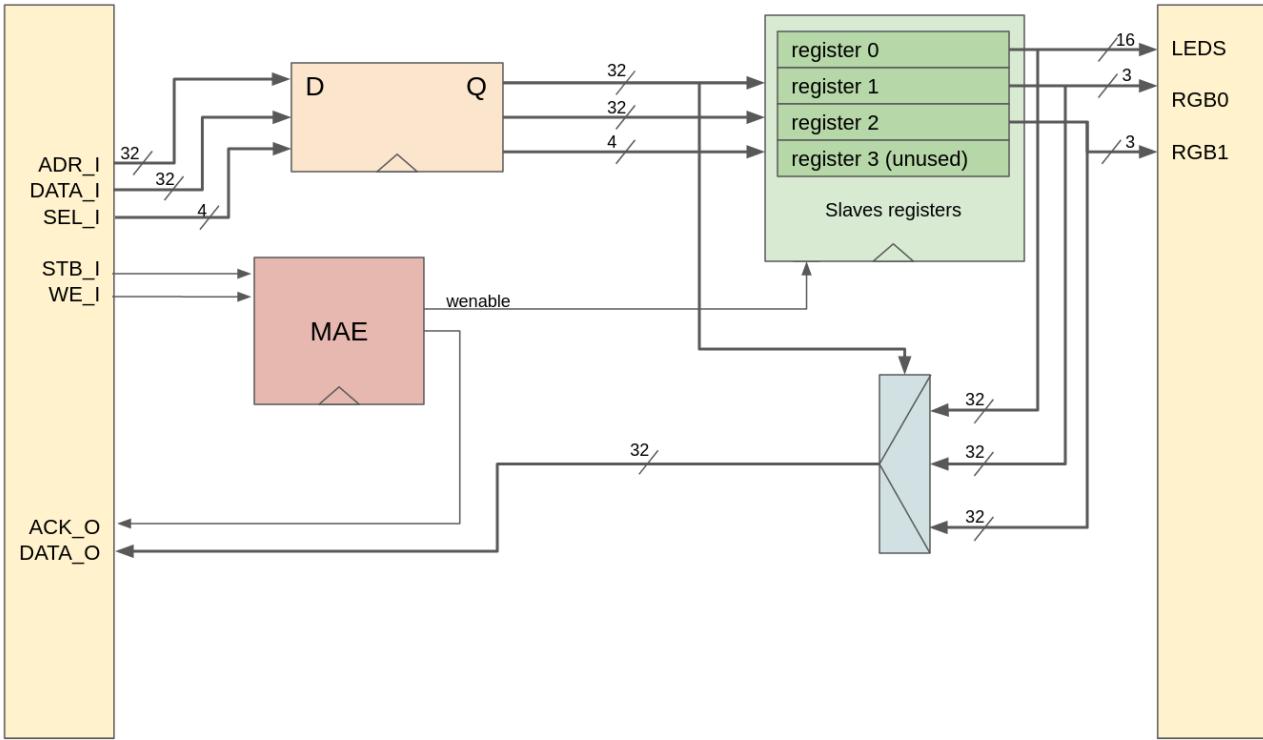


Figure 34: Pérophérique LEDs

Nous avons ici 4 registres adressables (nombre minimum de registres pour assurer l'alignement dans la table d'adressage) sur le bus accessibles en lecture et écriture (initialement l'adresse de base a été choisie pour être 0x40000000 mais cela est tout à fait configurable). Le dernier registre n'étant pas utilisé.

Nous y trouvons également une bascule D permettant de sauvegarder les valeurs de l'adresse, donnée et la sélection des octets. Dans le but de synchroniser la demande du maître avec les signaux de la MAE.

Cette MAE va alors servir à autoriser ou non l'écriture dans un registre (en cas d'un accès mémoire en écriture) et mettre à l'état haut le signal ACK_O afin d'informer le maître que sa demande a été traitée.

Dans la figure 35 nous retrouvons le graphe de transition de la MAE du wrapper LEDs ainsi que sa fonction de génération.

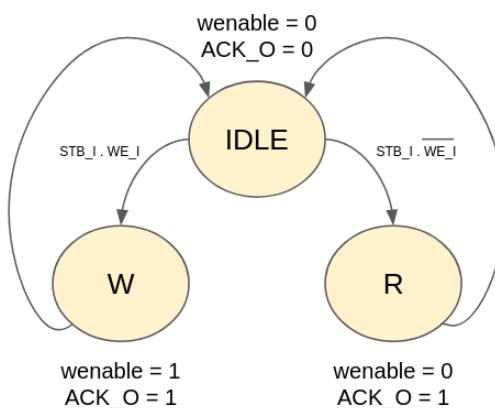


Figure 35: MAE de Moore du wrapper Wishbone du périphérique LEDs

Voici ci-dessous le schéma résumant l'implémentation complète sur FPGA. Bien évidemment, grâce au bus, il est possible de rajouter autant de périphériques compatibles Wishbone que l'on souhaite, le mode single read/write étant normalement nativement compatible avec tous les slaves Wishbone (On aurait alors un système contenant un master et plusieurs slaves).

Le système actuel ne permet pas d'avoir plusieurs masters sur le bus. Il faudrait rajouter les ports GNT_I et CYC_O, ainsi qu'un arbitre de bus à cette fin. Il est cependant toujours possible de rajouter plusieurs coeurs dans le même master avec un arbitrage qui sera effectué en interne du composant et non sur le bus.

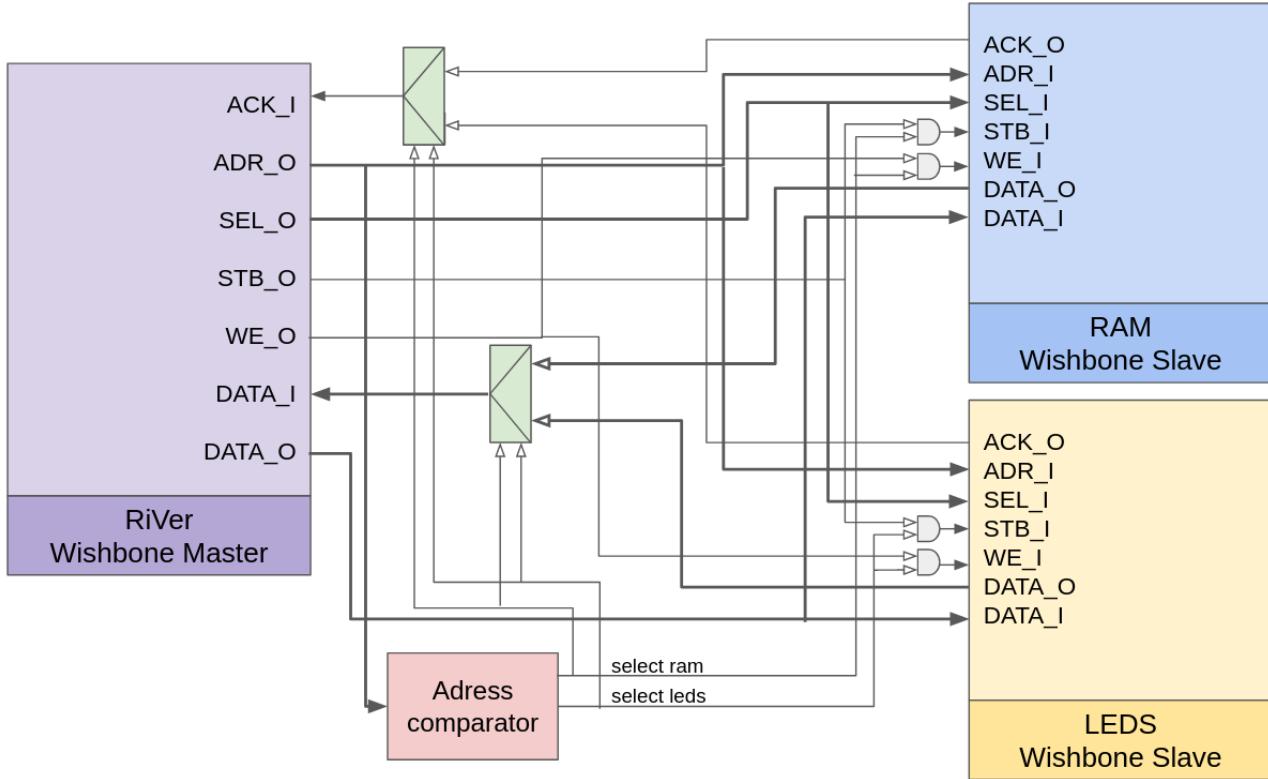


Figure 36: Wishbone CPU, RAM et LEDS

Le bloc "Adresse comparator" permet, comme indiqué sur le schéma, de sélectionner le slave en fonction de l'adresse émise par le master (chaque slave ayant une adresse de base).

La RAM étant toujours à ce stade un tableau de mémoire décrit en VHDL, il est assez compliqué de la préremplir par des instructions ou des données (à moins de le faire à la main, ce qui n'est pas nécessairement pratique). Nous allons dans la prochaine partie expliquer comment nous avons pu pallier à ces problèmes et optimiser l'utilisation des ressources mémoires sur FPGA.

5.2.2 Configuration de la RAM

Nous avons choisi d'utiliser une IP de Xilinx : Block Mem Generator[21] qui va nous permettre d'utiliser de façon optimale les ressources BRAM (Block RAM) présentes dans les cartes Xilinx, ainsi que de l'initialiser via un "Coefficient file" qui contient les valeurs en hexadécimal que l'on souhaite (à savoir les données et instructions).

Afin de réaliser ceci nous devons légèrement modifier l'interface Wishbone de la RAM, qui devient ainsi un contrôleur de RAM.

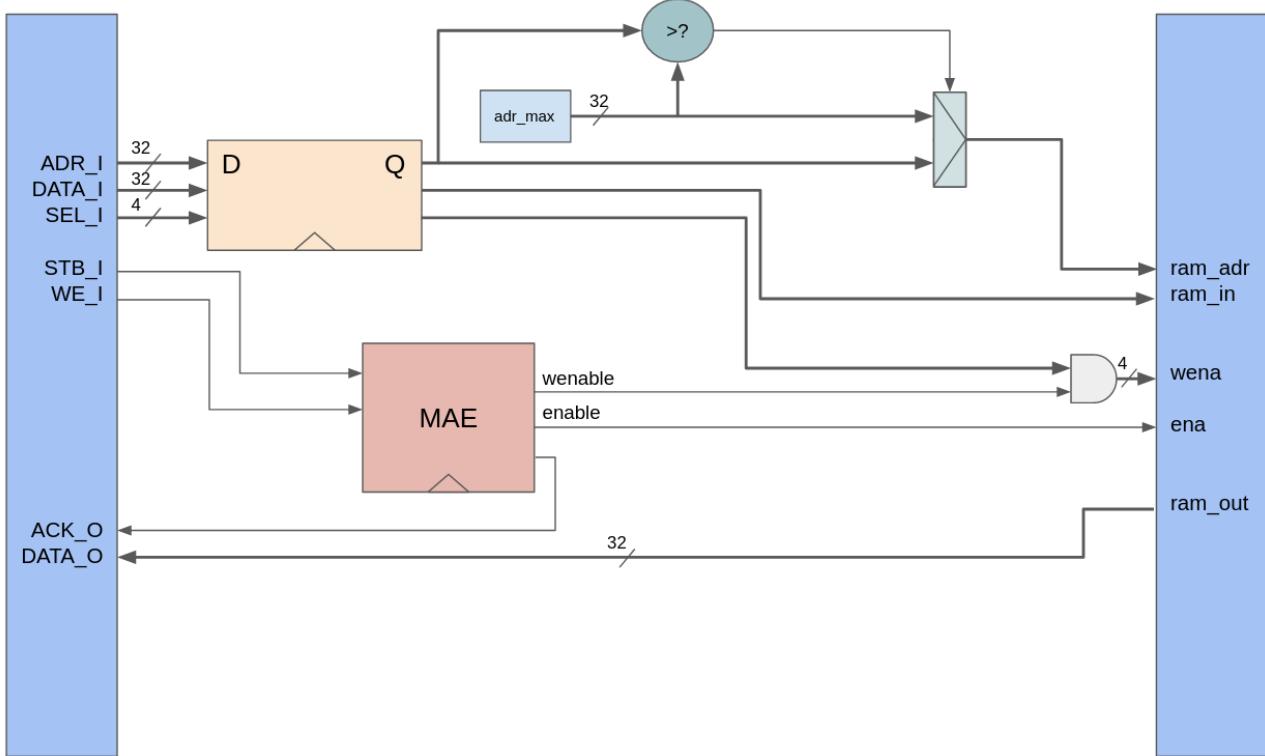


Figure 37: Wishbone contrôleur de RAM

Nous avons sur la figure 37 le schéma du contrôleur de RAM qui consiste en une bascule D qui va sauvegarder la demande du master et permettre la synchronisation entre la demande (adresse, data, sélection d'octets) et la MAE. Nous avons en plus de l'adresse, des entrée et sortie de la RAM (ram_adr, ram_in, ram_out) deux signaux :

- wena : write enable, sur 4 bits qui va permettre de sélectionner les octets à écrire.
- ena : enable, qui va activer la RAM que ce soit pour une écriture ou lecture.

adr_max étant une constante définie avec le choix de la taille de la RAM (étant donné qu'il n'est pas possible avec les ressources sur notre carte d'avoir les 4 Go adressable par 32 bits de la RAM) et ainsi éviter d'adresser une zone mémoire qui n'existe pas.

La figure 38 représente le graphe des transitions et la génération des sorties de la MAE du contrôleur de RAM. Nous y retrouvons 4 états :

- IDLE : état d'attente
- r : lecture de la RAM
- w : écriture dans la RAM
- done : fin de l'opération

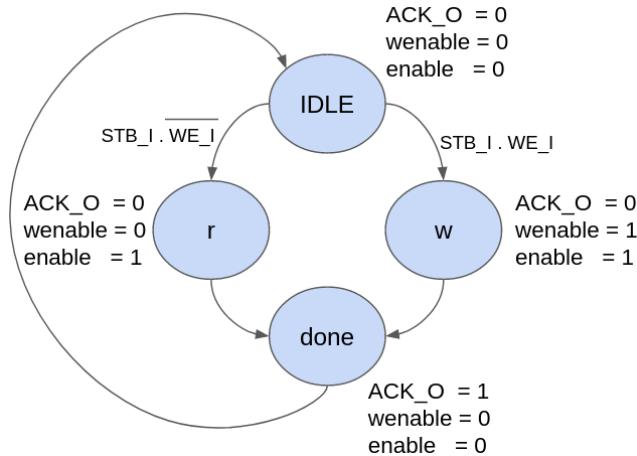


Figure 38: MAE de Moore du contrôleur de RAM

Notre implémentation sur FPGA est représentée sur la figure 39, nous avons donc connecté au bus Wishbone notre processeur avec deux périphériques : une RAM et LEDS.

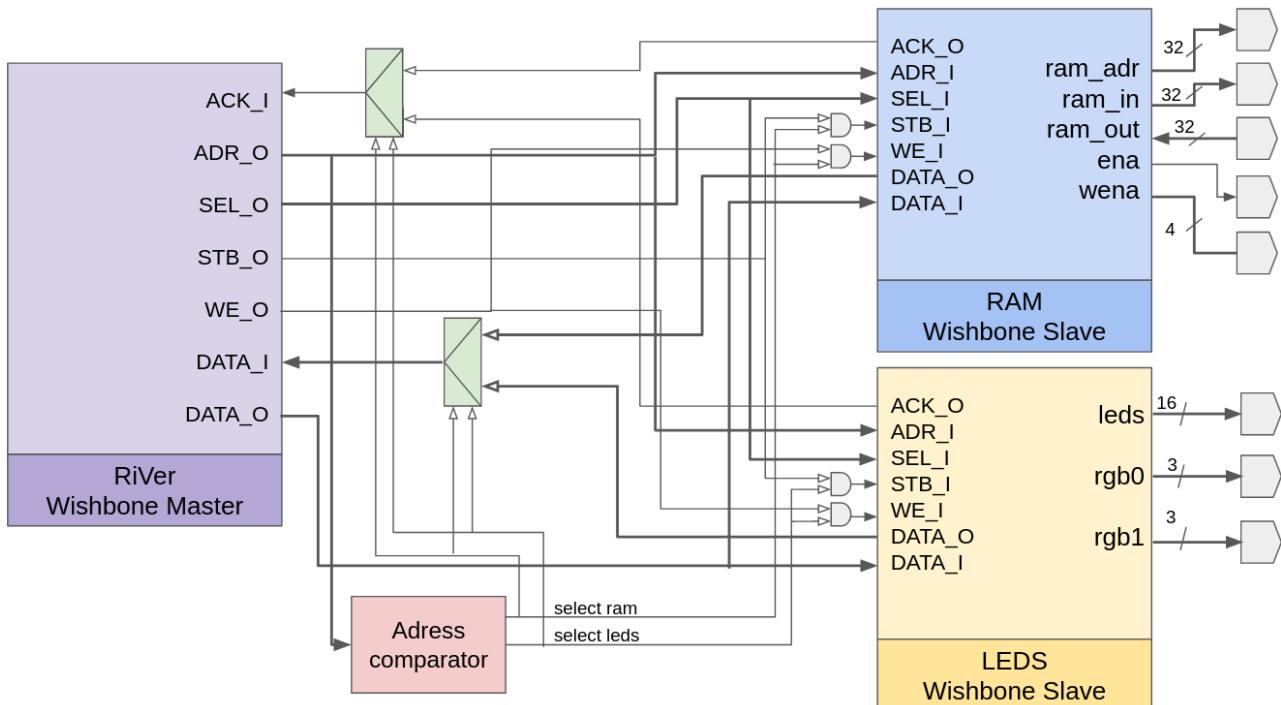


Figure 39: Implémentation finale sur FPGA

Nous observons sur la figure 40 le block design sur Vivado. Nous avons le top, qui est la description en RTL (VHDL) de toute la figure précédente et l'IP Block Memory Generator que nous pouvons configurer en taille ainsi que de nombreux paramètres qui ne nous intéressent pas dans le cadre de notre implémentation (comme l'ajout de registres dans le but de faire des accès pipelinés à la mémoire).

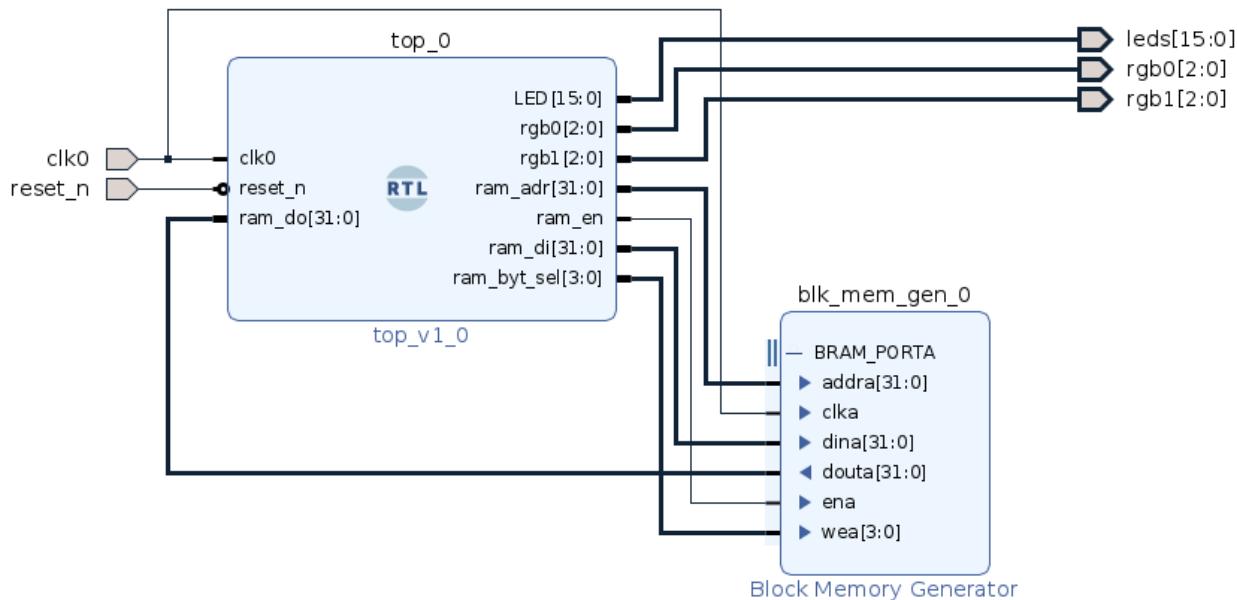


Figure 40: Block design Vivado

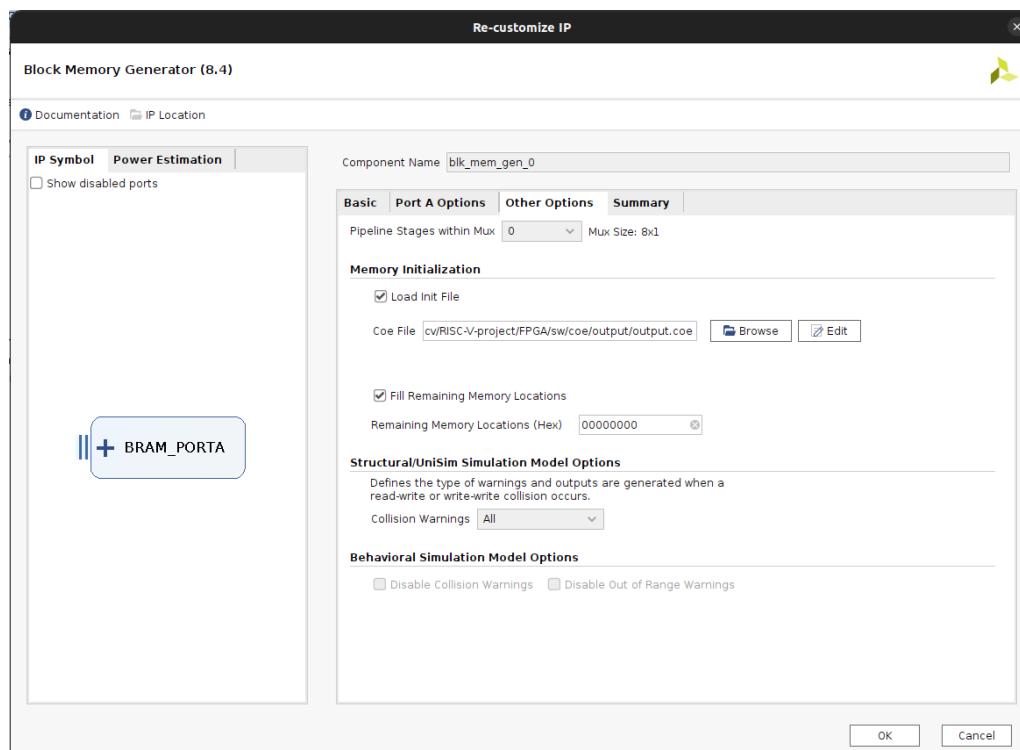


Figure 41: Initialisation de la RAM

La figure 41 présente les différentes options Le fichier d'initialisation de la RAM. Le fichier .coe est généré à partir de l'exécutable que nous souhaitons exécuter par notre processeur :

Grâce à un premier script[22] trouvé sur Github, nous allons pouvoir transformer l'exécutable en fichier .hex ou .txt qui va être en fait la suite des instructions/données dans la mémoire (à partir de l'adresse 0x00000000). Un deuxième script[23] va nous permettre de transformer le fichier précédent en fichier .coe demandé par l'IP de RAM. Tout cela est bien évidemment fait automatiquement via un Makefile dans le répertoire IMPL/sw présent sur le git de notre projet[24].

Nous nous sommes également amusés à écrire des drivers en C (présents dans le répertoire IMPL/sw) pour programmer notre système et faire clignoter des LEDs, ce qui est tout à fait appréciable à observer.

Nous avons pu écrire deux drivers :

- driver_wb : driver générique de notre système contenant des fonctions d'écriture et lecture des registres, ainsi qu'une fonction d'attente et toutes les informations du système et du processeur.
- driver_leds : driver pour contrôler les LEDs contenant des fonctions pour allumer et éteindre les LEDs.

5.2.3 Analyse post-implémentation

Nous allons dans cette section analyser notre implémentation physique du système vu précédemment sur FPGA. Nous rappelons que notre cible dans le cadre de ce projet est une Nexys A7 de Xilinx, mais il peut bien évidemment être porté sur d'autres cibles d'autres constructeurs. Nous avons gardé les paramètres par défaut de Vivado pour la synthèse et l'implémentation.



Figure 42: Utilisation des ressources FPGA

La figure 42 présente l'utilisation des ressources FPGA. Nous remarquons en particulier que les LUT et flipflop sont principalement pour le processeur. Leurs utilisations par d'autres composants restent négligeables (sauf la RAM pour les BRAM).

Notre RAM utilise 95% de la BRAM présente dans le FPGA soit environ près de 4860 Ko.

Nous remarquons alors que nous n'utilisons qu'environ 7000 LUT (11% de la capacité maximale) de la carte, mais cela devrait être mis en comparaison avec d'autres implémentations similaires de cœur RISC-V. Mais nous constatons qu'il est faisable d'implémenter plusieurs coeurs (huit au maximum dans la configuration actuel, cela ferait 88% de LUT utilisés) sans penser à toutes leurs interfaces via le bus Wishbone.

Il semble également important de noter que dans notre implémentation actuelle, nous n'avons pas de caches. Ce qui est bien évidemment pas optimale au niveau de la performance, mais en théorie assez facilement implémentable étant donné que tous les wrappers ont déjà été implémentés.

Une implémentation utilisant toutes les ressources du FPGA pourraient être composée de 2 à 3 coeurs de notre RiVer ainsi qu'un cache d'instructions et de données (dépendant de leur taille) qui utiliseront probablement des LUTRAM.

Summary	
Name	Path 1
Slack	0.029ns
Source	riverplatform_1top_0/U0/wb_master_cpu/river_core0/dec_i/dec2if/data_reg[69]_replica_1/C
Destination	(rising edge-triggered cell FDRE clocked by clk0 {rise@0.000ns fall@12.000ns period=24.000ns}) riverplatform_1top_0/U0/wb_master_cpu/river_core0/ifetch_i/branch_adr_reg[28][11]/CE
Path Group	clk0
Path Type	Setup (Max at Slow Process Corner)
Requirement	12.000ns (clk0 fall@12.000ns - clk0 rise@0.000ns)
Data Path Delay	11.224ns (logic 2.893ns (25.776%) route 8.331ns (74.224%))
Logic Levels	14 (CARRY4=3 LUT4=4 LUT5=3 LUT6=4)
Clock Path Skew	0.306ns
Clock Un...rtainty	0.035ns

Figure 43: Chemin critique

Sur la figure 43 le chemin critique (temps de propagation le plus long entre deux bascules D) se trouve dans l'étage ifetch et la prédiction de branchement et la fifo entre ifetch et decod. La période minimale que nous avons pu forcer est de 24 ns (demi-période de 12 ns) soit une fréquence maximale 41,77 MHz ou pour simplifier, nous pouvons imposer une fréquence de 40 MHz (sur Nexys A7). Ce qui nous semble être une assez bonne valeur de fréquence étant donné que nous n'avons pas cherché à optimiser le chemin critique de notre processeur. Cette fréquence reste alors tout à fait acceptable, notamment dans le monde des micro-contrôleurs.

Pour cela nous pouvons donc imaginer des solutions afin de maximiser la fréquence (minimiser le chemin critique) de découper en deux l'étage ifetch ou d'optimiser la prédiction de branchement.

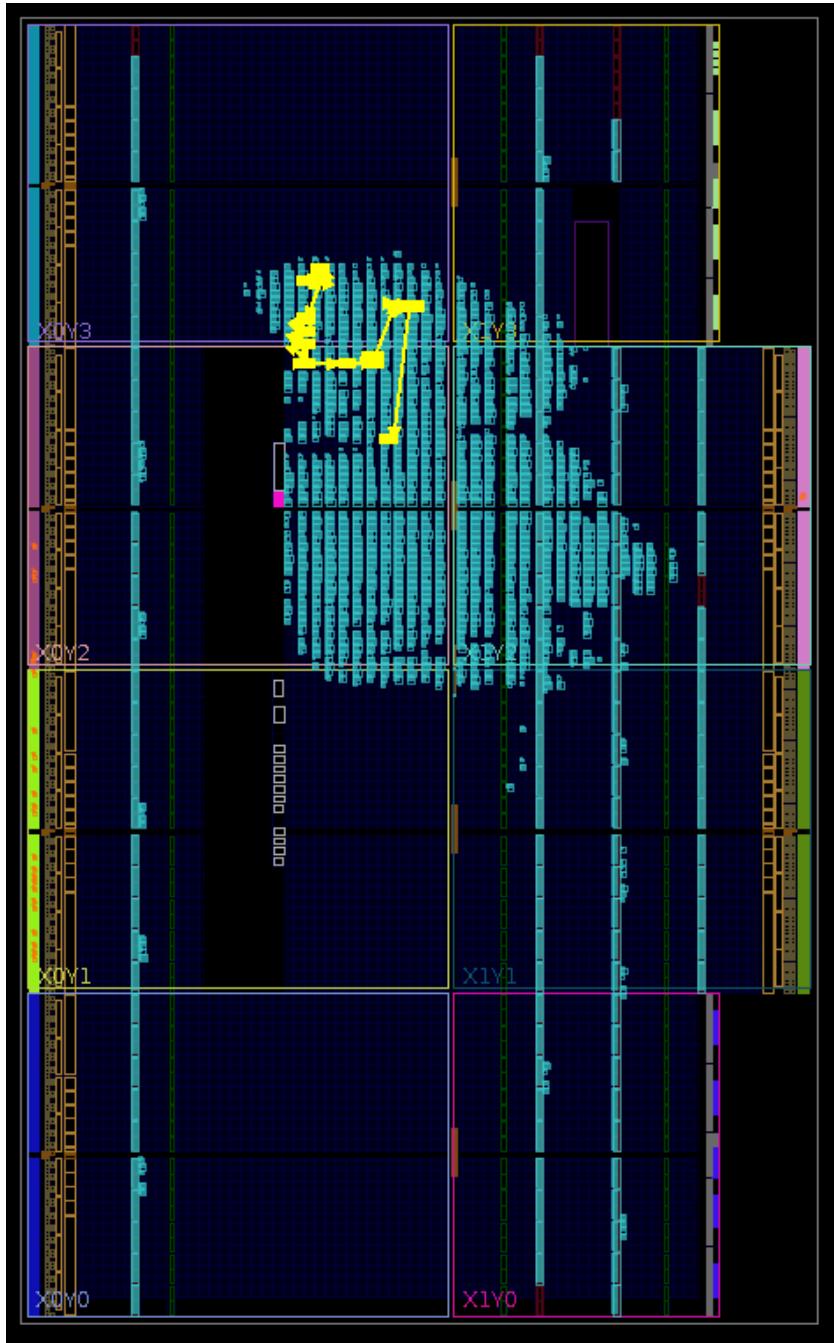


Figure 44: Chemin critique post-implémentation

Nous retrouvons sur la figure 44 le chemin critique en post-implémentation qui a pu être mis en évidence (en jaune) grâce à Vivado.

Ceci conclut cette partie FPGA. Nous avons pu effectuer plusieurs tests avec plusieurs programmes faits par nos soins (clignotement de leds, chenillard...) afin de confirmer le bon fonctionnement de notre système. Il est également possible d'imaginer de porter notre processeur sur un ASIC, notamment avec la suite de logiciels d'Alliance/Coriolis du LIP6.

6 Miniriscv, un cœur allégé pour l'enseignement

Nous avons récupéré une partie du processeur (processeur simple, sans prédition de branchement, multiplication, superscalaire, ou mode de privilège / fonction dites "kernel" dans le reste de ce rapport), que nous avons réorganisée pour qu'elle soit plus compréhensible, et nous y avons ajouté un grand nombre de commentaires. L'idée est que ce code soit compréhensible par des étudiants qui y mettraient le temps, ou au moins lisible et utilisable lors de TP en un temps assez limité.

Au moment de l'écriture de ce rapport, deux TP prévoient d'utiliser ce processeur : l'un pour expliquer le systemC et le fonctionnement du processeur en lui-même, où les étudiants doivent exécuter des instructions sur le processeur, puis y ajouter un composant simple (qui aura été retiré au préalable), le Shifter.

Le deuxième TP rentre moins dans les détails de l'utilisation du processeur, mais montre un exemple de communication simple entre le processeur et la RAM en utilisant l'interface VCI. Le fait d'utiliser le même processeur exactement aide à la compréhension du TP.

7 Conclusion

7.1 Problèmes rencontrés lors de la conception

La première difficulté que nous avons rencontré a été sur la répartition du travail. En effet, le projet étant assez conséquent il nous a fallu bien définir qui s'occuperaient de quoi afin d'éviter de se marcher sur les pieds lors de la réalisation du projet. De plus définir des étapes clef ainsi que des dates limites nous a beaucoup aidé dans la répartition du travail.

La deuxième difficulté rencontrée était la synthèse de la spécification RISCV, il a en effet fallu faire le tri entre ce que nous comptions implémenter ou pas et il nous a fallu bien comprendre le jeu d'instructions afin de l'implémenter correctement.

Les bypass auront également été difficiles à implémenter dans la mesure où c'était la première fois que nous implémentions une architecture avec des bypass. En effet, il y a beaucoup de cas particuliers auxquels il faut faire attention, car ils peuvent poser de nombreux problèmes. D'autre part l'implémentation SS2 double la quantité de bypass utilisés et complexifie les dépendances de données, le débogage de cet architecture aura donc été assez long.

Enfin la mise en place d'un mode machine/user, ce que nous appelons plus simplement mode Kernel nous aura pris énormément de temps. En effet, nous n'avons jamais étudié ce type d'architecture auparavant étant donné que c'est au programme de M2. Nous n'avions donc aucune idée initialement de ce qu'il fallait faire et de comment gérer ça.

Enfin pour la partie FPGA nous avons eu quelques difficultés à trouver une solution simple et universelle pour programmer notre processeur implémenté. Nous avons notamment abandonné l'idée de reprogrammer dynamiquement par JTAG. La documentation sur l'usage du JTAG étant assez succincte (sans passer par AXI4 dans le cadre des IP Xilinx) pour charger directement un code. Il est cependant possible de rajouter cette fonctionnalité avec le système actuel et quelques modifications.

7.2 Perspectives d'amélioration

Au cours de ce projet qui a duré près de 8 mois, nous avons eu l'occasion de développer énormément de chose et d'expérimenter beaucoup d'implémentations différentes.

Le coeur scalaire étant totalement opérationnel il pourrait être capable de supporter un système d'exploitation type linux à condition d'ajouter la gestion de la mémoire virtuelle et d'ajouter un mode superviseur pour la partie kernel. Enfin il faudrait également ajouter une exception dans le cas où un utilisateur tenterait d'utiliser une instruction CSR. Nous n'avons pas implémenté cette fonctionnalité mais si un OS est porté sur le coeur et que cette exception n'est pas générée, de grosse faille de sécurité pourrait apparaître.

Si ces fonctionnalités étaient assurées il serait possible de faire tourner un OS très basique sur le cœur contenant simplement la librairie standard. Mr. Franck Wajsburst a développé un OS type linux très simplifié que Timothée Le Berre avait commencé à adapter pour notre processeur, ses travaux pourraient donc être repris avec comme optique d'exécuter ce code directement sur notre cœur.

D'autre part, lors de l'implémentation sur FPGA Samy Attal a découvert un bug sur le mécanisme RAS de la prédiction de branchement que nous n'avons pas eu le temps de corriger.

Le SoC dual cœur n'a que peu été testé et beaucoup de problème peuvent encore apparaître.

Il pourrait être intéressant d'ajouter une latence sur les accès mémoire et ainsi de mesurer l'impact réel des caches, en effet la mémoire étant parfaite dans notre simulateur les caches n'ont aucun intérêt si une latence n'est pas introduite.

Enfin le SS2 pourrait être amélioré afin de tendre d'avantage vers un modèle synthétisable et la prédiction de branchement pourrait également être ajouté sur cet implémentation.

Enfin pour la partie VHDL, le code pourrait être adapté afin de le faire synthétiser par la suite logiciels d'Alliance/Coriolis du LIP6 et analyser une potentielle implémentation sur silicium.

References

- [1] <https://www.x86-guide.net/fr/cpu/Intel-8086-PDIP-cpu-no662.html>
- [2] <https://www.cpushack.com/MIPSCPU.html>
- [3] https://en.wikipedia.org/wiki/IBM_System/370
- [4] <https://www.accellera.org/>
- [5] https://en.wikipedia.org/wiki/IBM_document_processorsIBM_801
- [6] https://en.wikipedia.org/wiki/Wallace_tree
- [7] Waterman, A., Lee, Y., Patterson, D., Asanovic, K., level Isa, V. I. U. (2014). The RISC-V instruction set manual. Volume I: User-Level ISA'.
<https://riscv.org/technical/specifications/>
- [8] Waterman, A., Lee, Y., Patterson, D., Asanovic, K., level Isa, V. I. U. (2014). The RISC-V instruction set manual. Volume II: Privileged Architecture.
<https://riscv.org/technical/specifications/>
- [9] Asanović, K., Patterson, D. A. (2014). Instruction sets should be free: The case for risc-v. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146.
- [10] Utting, M., Kearney, P. (1992). Pipeline specification of a MIPS R3000 CPU. Technical Report 92-6, Software Verification Research Centre, Department of Computer Science, University of Queensland.
- [11] David A. Patterson John L. Hennessy (2021). Computer organization and design RISC-V edition, second edition.
- [12] Jurij Šilc, Jurij Silc, Borut Robic, Theo Ungerer (1999). Processor architecture : From dataflow to superscalar and beyond.
- [13] https://github.com/lovisXII/RISC-V-project/tree/main/Compte_rendu
- [14] https://www.researchgate.net/figure/Conventional-Array-Multiplier_fig1_306034550
- [15] <https://github.com/riscv-software-src/riscof>
- [16] https://www.brainkart.com/article/Arithmetic-Operations-Division_8616/
- [17] https://cdn.opencores.org/downloads/wbspec_b3.pdf
- [18] <http://ghdl.free.fr/>
- [19] <http://gtkwave.sourceforge.net/>
- [20] <https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual>
- [21] <https://docs.xilinx.com/v/u/en-US/pg058-blk-mem-gen>
- [22] <https://github.com/irmo-de/xilinx-risc-v/blob/main/firmware/makehex.py>
- [23] <https://github.com/kooltz/xilinx-coe-generator>
- [24] https://github.com/lovisXII/RiVer_SoC/tree/main/IMPL/sw
- [25] https://en.wikipedia.org/wiki/Branch_predictorStatic_branch_prediction