SORBONNE UNIVERSITÉ

M1 SESI

Modélisation SystemC d'un processeur RISC-V pipeliné

Rapport final
dans le cadre du projet PSESI

Étudiants:
M. Timothée Le Berre
M. Louis Geoffroy Pitailler
M. Kevin Lastra

Encadrants :
Mm. Daniela Genius
M. Pirouz Bazargan Sabet
M. Franck Wajsbürt

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

Table de Matières

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).

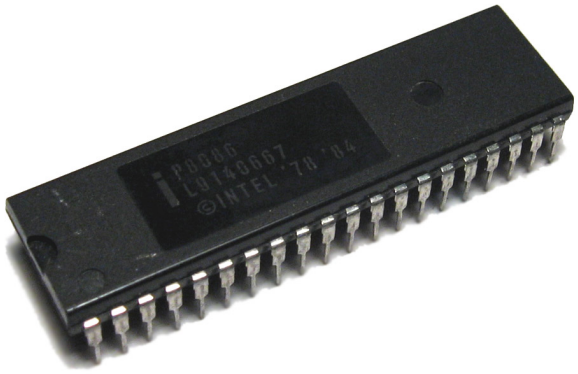


Figure 1: Intel 8086 [1]

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 taches 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 Standford.
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 Standford- 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 multitâches 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 articulé 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 Pourquoi choisir ce projet ?

Au cours de notre premier semestre de M1 SESI, nous avons eu l'occasion d'implémenter une architecture ARMv2a en VHDL. Cela nous a énormément plu et 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 à 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.

Néanmoins, après discussion avec notre encadrante et les deux autres groupes travaillant également sur RISC-V, il a été décidé 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.

2 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

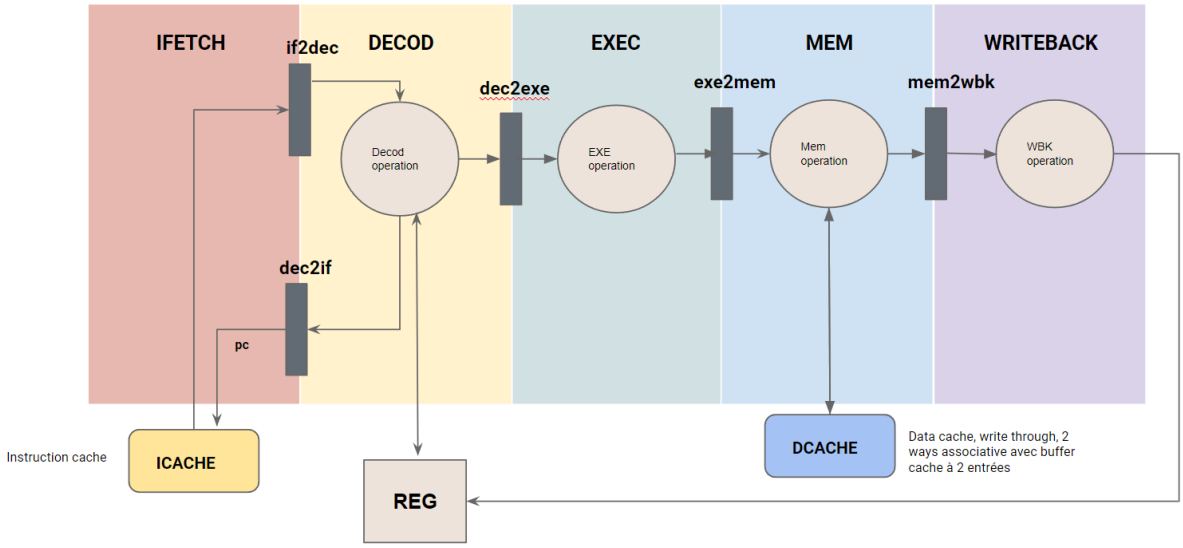


Figure 2: Schéma du pipeline, les blocs entre chaque étage désignent des fifos et le nom qui leur est attribué

Le langage imposé pour cette réalisation est SystemC [4], en effet il s'agit du langage utilisé pour l'UE MASSOC du S2 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.
Nous pensions éventuellement réaliser une plateforme de TME, et une implémentation superscalaire similaire à SS2, mais nous n'avons pas eu le temps de tout faire et avons préféré nous concentrer sur les objectifs principaux.

3 Situation actuelle du projet

Au mois de mai 2022, nous avons fini la partie scalaire et l'implémentation l'extension Ziscr. Nous avons de plus réussi à faire tourner une suite de test semi-officielle pour le jeu d'instruction RV32I, la suite **RISCOF** de *INCORE*, permettant de valider de manière assez fiable le bon fonctionnement du processeur.
Nous avons en effet choisi cette suite de tests car elle était extrêmement bien documenté et très complète. Elle comporte en effet une centaine de test par instruction ce qui permet de très bien testé le processeur.
Le cache de donnée est également terminé est branché sur l'étage MEM. Le cache d'instruction est quant à lui fini, mais présente un problème de stall lorsqu'il rencontre un branchement avec bypass au même cycle.
Enfin, l'implémentation Kernel est encore en chantier. Nous avons actuellement fini la gestion des exceptions et le chargement du gestionnaire d'exception. Il reste néanmoins à bien gérer le changement de mode User/-Machine et à faire passer tout ça sur le banc de test.

3.1 Résumé graphique

Tous les membres de notre projet ont participé à l'ensemble de la conception, mais certains membres se sont plus concentré sur certaines taches. Le tableau ci-dessous indique les taches accomplies et celles qui sont en cours, et qui a majoritairement contribué à la réalisation de la tache.

	Janvier	Février	Mars	Avril	Mai			
Documentation RiscV	X							
CORE RiscV								
Etage IFETCH		X						
Etage DecodE		X						
Etage EXEC		X						
Etage MEMORY		X						
Etage WRITEBACK		X						
Débogage CORE		X						
Bypass			X					
CSR				X				
Suite de test RiscV				X				
Gestion Exceptions/Interruptions				X				
Extension RiscV								
Kernel								
M (multiplication/division)								
Ziscr				X				
Chip								
ICache				X				
DCache				X				
Interfaçage avec le CORE								
Mips R3000								
Mis à jour Mips3000R		X						
Documentations Mips3000R		X						

X : Représente le mois où la tache a été finalisée, si une ligne ne contient rien c'est qu'elle est toujours en cours.

Timothée Le Berre	
Louis Geoffroy Pitailler	
Kevin Lastra	

4 MIPSR3000

Un des premiers objectifs que nos encadrants nous a chargé de réaliser au début du projet était la mise à jour et documentation de l'implémentation SystemC d'un processeur MIPSR3000 [9] développé par D. Hommais and P. Bazargan Sabet et traduit en SystemC par F. Pecheux.

À ce jour cette implémentation a été documentée et actualisé à la version "systemc2.3.3".

4.1 Qu'est-ce qu'est le MIPSR3000

R3000 est un microprocesseur RISC 32-bits développé par MIPS Technologies, qui implémente un set d'instruction MIPS I. Introduit en 1988, il était le deuxième processeur implémenté par cette compagnie, le R3000 est le successeur du R2000 qui a été développé en 1986.

4.2 Structure

L'architecture réalisé par F. Pecheux est pipelinée sur 5 étages à savoir : Ifetch, Decode, Execute, Memory et Writeback, et possède un coprocesseur 0 qui aide à la gestion des exceptions et des interruptions en stockant des informations sur ces dernières.

Dans la structure de fichiers, chaque étage est divisé en 3 fichiers :
1. Le fichier principal où est définie toute la logique de l'étage(ex. decode.h).
2. Le MUX, ce multiplexeur va contrôler le flux d'information dans la fifo utilisant 3 signaux BUBBLE, HOLD et KEEP.
3. La FIFO, qui est l'interface entre 2 étages.

4.3 Etages du MIPSR3000

- IFetch
  Cet étage est chargé de faire la demande de la prochaine instruction et de l'envoyer à l'étage Decod. Dans cette implémentation, différente de notre implémentation RISCV, IFetch détecte si l'instruction provenant du cache est un delayed slot et si c'est le cas, l'insère dans decode.
- Decod
  Decod peut décoder 3 types d'instruction I, J et R.
- Execute
  Ici, selon l'instruction, on choisit quelle valeur, entre le résultat de l'ALU et du SHIFTER, sera envoyée vers l'étage MEMORY. Par ailleurs, on fait le traitement de 3 exceptions différentes :

  1. IABUSER, erreur de bus
  2. IAMALGN, erreur d'alignement
  3. IASVIOL, erreur de segmentation

  On trouve aussi le traitement des instructions SYSCALL et BREAK.
- Memory
  Cet étage est principalement chargé de communiquer avec la mémoire pour récupérer les données si besoin, mais il est aussi chargé du traitement de plusieurs erreurs :

  1. DABUSER
  2. LAMALGN

3. SAMALGN
4. LASVIOL
5. SASVIOL

Avec ces erreurs et celle mentionnée dans le schéma ci dessous 3, on trouve le résultat pour EXCCODE qui fait office de numéro d'exception.

```
           | 0    if INTRQ                                    interrupt request
           | 4    if LAMALGN or LASVIOL or IAMALGN or IASVIOL
           | 5    if SAMALGN or SASVIOL
           | 6    if IABUSER                                  instruction adresse bus error
           | 7    if DABUSER                                  data adresse bus error
EXCCODE =  | 8    if SYSCALL                                  syscall exception
           | 9    if BREAK                                    break exception
           | 0xA  if ILLGINS                                  unknown instruction
           | 0xB  if C0UNUSE                                  coprocessor 0 unused
           | else 0xC
```

Figure 3: Extrait de commentaires du code MIPSR3000

- Write back
  L'unique objectif de cet étage est l'écriture sur la banc de registre, qui est un tableau de 32 cases de 32 bits permettant de stocker des données sans recourir à la mémoire externe (RAM).

5 Réalisation RISCV et choix architecturaux

5.1 Introduction

Comme indiqué précédemment, notre processeur est un CPU 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.

5.2 Structure du pipeline

5.2.1 IFETCH

IFETCH est un étage relativement simple qui contient uniquement une 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. 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 5.4.

5.2.2 Decod

Decod est l'un des étages les plus complexes de l'architecture puisque son rôle est de récupérer l'instruction en provenance de IFECTH 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 [12].
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 5.3.2
- 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	LUI
imm[31:12]			rd	0010111	AUIPC
imm[20|10:1|11|19:12]			rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	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	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
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	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Figure 4: 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 fifos (flushage) dans le cas d'un branchement qui réussi. 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**.

Extension M

Enfin, nous avons commencé fin avril l'ajout de l'extension M permettant d'ajouter un multiplieur 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. Avoir ces deux entités est donc plus que nécessaire si l'on veut avoir un processeur puissant.

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Figure 5: Opcode, extrait de la documentation RISCV, extension M : [7]

5.2.3 EXEC

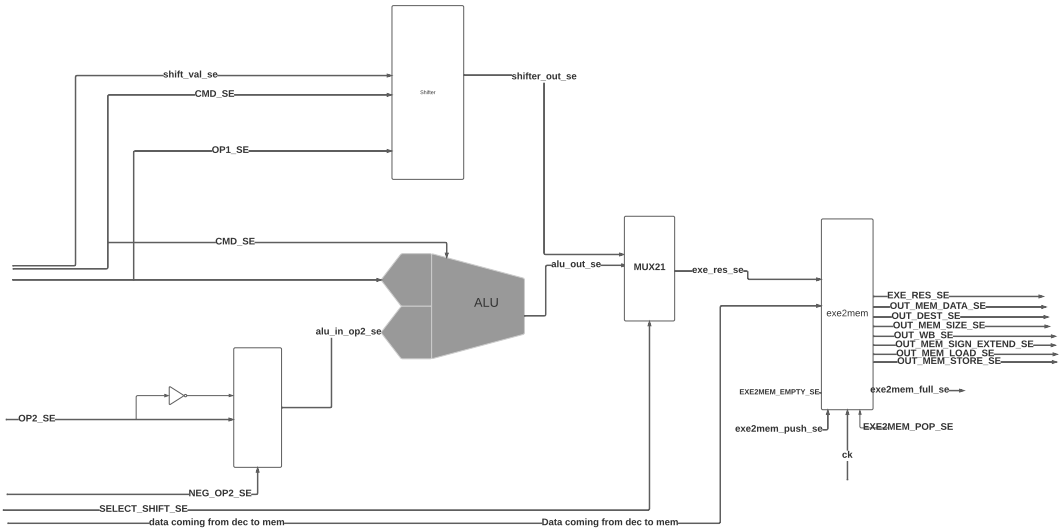


Figure 6: Schéma simplifié de l'architecture de EXEC

EXEC est constitués de 2 parties primordiales : l'ALU (*arithemic 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 permettent 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.

5.2.4 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.

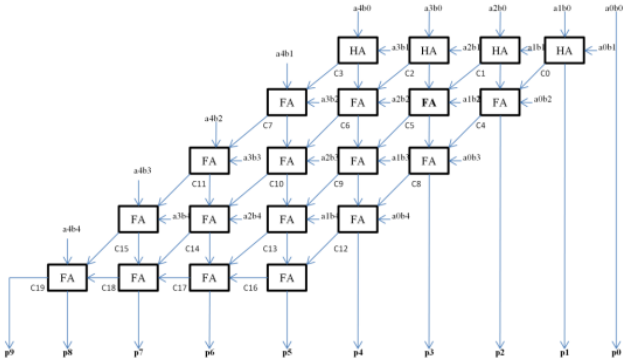


Figure 7: Array multiplier[13]

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 long. L'idée avancée par Mr. Bazargan Sabet est d'utiliser un multiplieur pipeliné sur 3 étages se basant sur l'algorithme "Wallace Tree multiplier" ou "multiplieur de Wallace"[6]. L'idée étant qu'en cas de multiplication, Decod informe EXEC, MEM et WBK qu'une opération de ce type va être lancé. L'instruction est ensuite envoyée vers le multiplieur qui va s'étendre du cycle EXEC au cycle WBK et qui va se charger de l'opération.

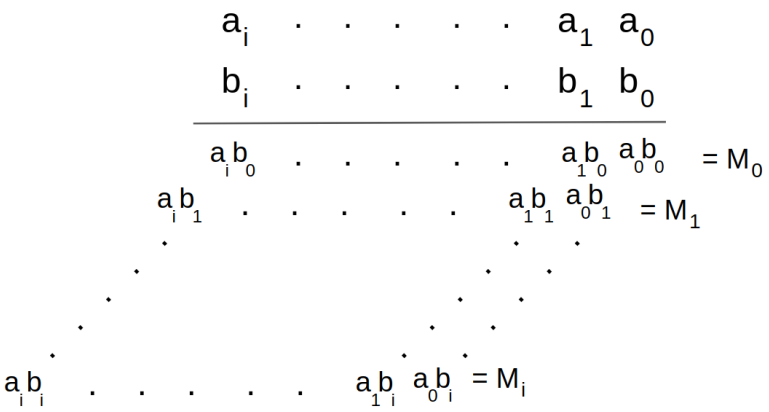


Figure 8: Multiplication(a_i représentation binaire de a, b_i représentation binaire de b)

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

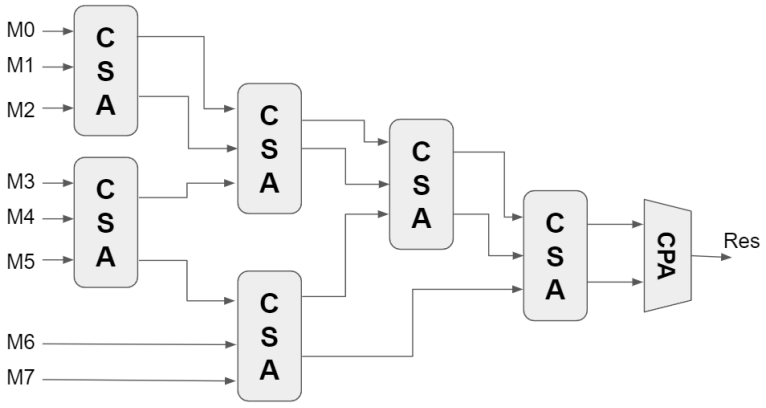


Figure 9: Wallace tree multiplier

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.

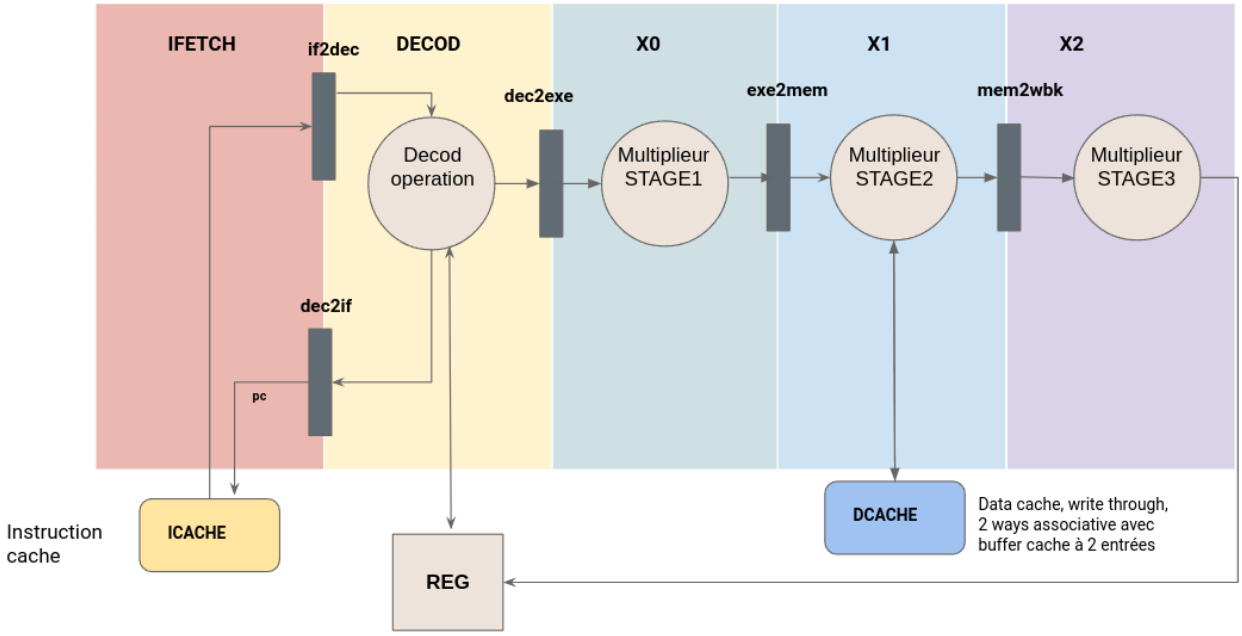


Figure 10: Schéma du Multiplieur intégré au pipeline

5.2.5 Diviseur

Le diviseur est actuellement en phase de développement. Il est prévu qu'il puisse gérer les divisions en entre 3 à 32 cycles directement depuis le cycle EXEC.

5.2.6 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. La réalisation de cet étage est relativement simple, elle ne comporte pas grand-chose hormis la définition des accès mémoire. Les données récupérées sont ensuite envoyées à WBK à l'aide de la fifo **MEM2WBK**.
On trouve néanmoins toute la gestion des exceptions dans cet étage, mais cela sera développé dans la section 5.3.

5.2.7 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.

5.2.8 REG

Enfin, le banc de registre 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 CORE, 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 même interfaces pour les instructions de branchement comme par exemple jal.
On notera que l'adresse de lecture est sur 6 bits dans REG malgré que les adresses des opcode soit uniquement sur 5, en effet dans le cas des branchements on lit la valeur de PC, d'où l'intérêt de passer ca sur 6 bits.

5.3 Implémentation de l'extension Zicsr et mode Kernel

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 2 points essentiels :

- Gestion des Interruptions et des exceptions,
- Extension CSR

5.3.1 Gestion des interruptions et des exceptions :

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 riscv, 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 non alignée
- Instruction access fault : Essaie d'accéder à une zone mémoire sans les privilèges nécessaire
- Illegal instruction : L'instruction n'existe pas
- Load address misaligned : Adresse de load non alignée
- Load access fault : Bus erreur suite à l'accès d'un load
- Environment call from U-mode : Appel système en mode User
- Environment call from M-mode : Appel système en mode Machine

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 soit 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 de write back dans MEM lorsqu'une Instruction est détectée. On précise que la désactivation dans MEM se fait avant l'accès mémoire.

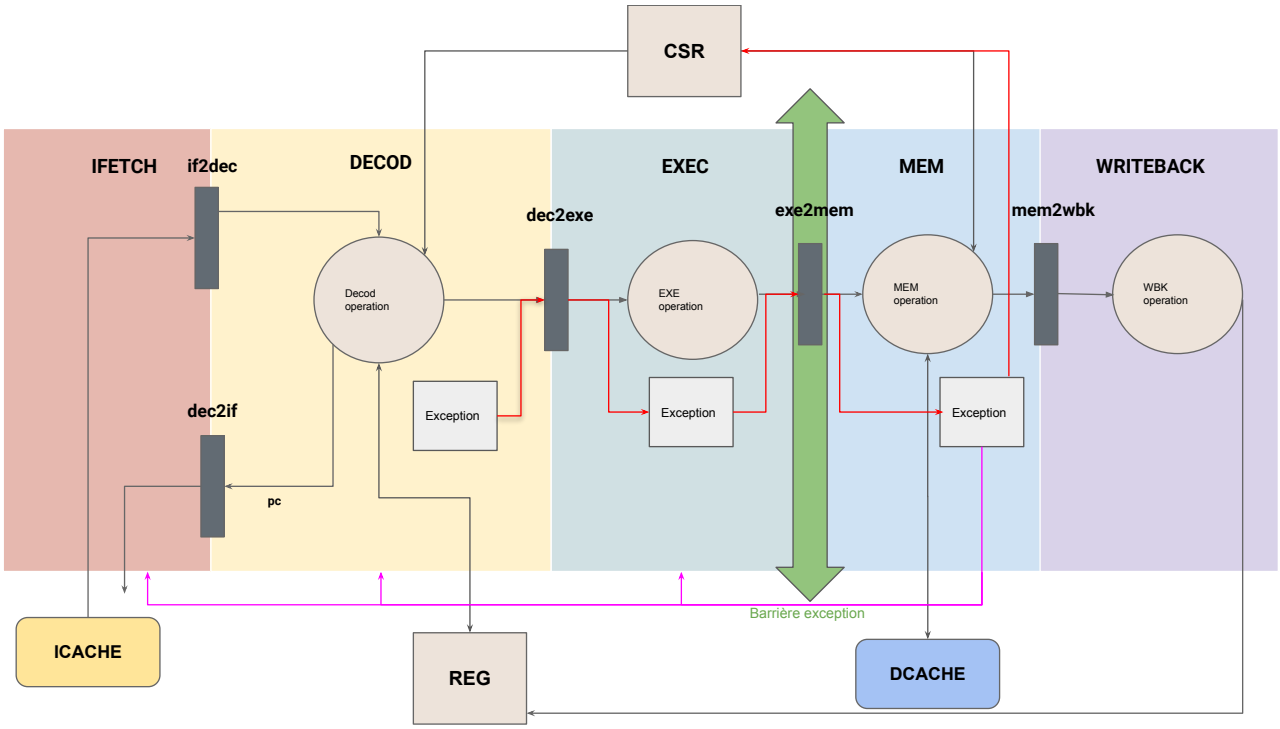


Figure 11: Schéma du pipeline avec la gestion des exceptions

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 :

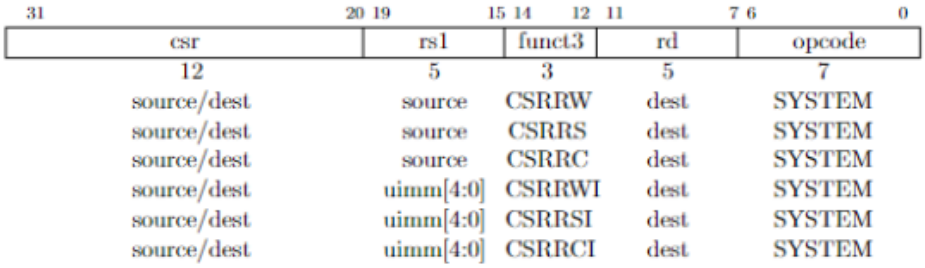
- Instruction address misaligned : Decod
- Instruction access fault : EXE
- Illegal instruction : DEC
- Load address misaligned : EXE
- Load access fault : MEM
- Environment call from U-mode : DEC
- Environment call from M-mode : DEC

Dans chaque étage on effectue un "ou" logique entre toutes les exceptions que l'on propage dans l'étage suivant. Par exemple dans Decod on va effectuer :
illegal_instruction **ou** adress_missaligned **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 CSR. On récupère également les signaux de toutes les exceptions dans MEM pour pouvoir écrire dans les CSR le code correspondant à l'exception détectée.
Enfin, lorsque MEM détecte une exception, il envoie un signal à destination de IFETCH, Decod et EXEC qui

vont vider les fifos en y mettant des NOP. Le pipeline va changer de mode pour passer en mode machine et va sauter à l'adresse mtvec pour exécuter le code de gestion de l'exception détectée, autrement appelé un trap.

5.3.2 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 12: 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 [7] 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,
- mstatus : c'est le registre de statut du processeur, il garde de nombreuses informations sur le processeur comme par exemple le mode courant du processeur, l'activation ou non des interruptions...
- misa : donne les extensions implémenté dans l'architecture
- mie : Contient des informations sur les interruptions machine activée
- mtvec : contient l'adresse de base des fonctions trap
- 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

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 ne peut pas écrire ou lire le CSR qui est en cours de traitement.
Pour palier 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 d'autre instruction CSR et qu'il doit donc geler si jamais il est en train de décoder une instruction CSR.
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.

5.4 Caches

Suite à de nombreuses discussions entre les membres de notre groupe, nous avons décidé d'implémenter des caches L1 séparé. 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.

5.4.1 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 état), 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 :

1. IDLE
   L'état par défaut qui répond aux requêtes du processeur et en cas de MISS fait une demande au niveau supérieur de mémoire (RAM, qui dans notre cas est simulé par une map C++).

2. WAIT MEM
   Cet état attend la réponse de la mémoire.

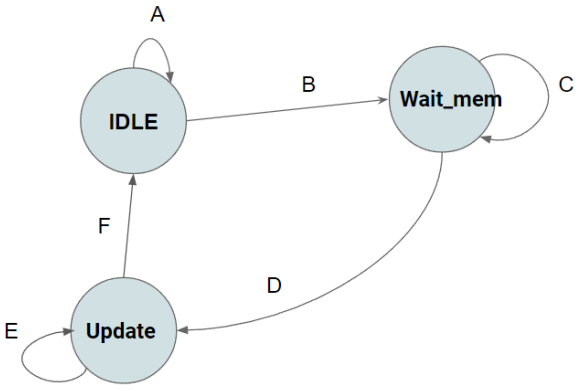


Figure 13: Machine a états du cache d'instruction

3. UPDATE
   Cet état est chargé de recevoir les données du bus et de les charger dans les lignes de cache correspondantes.

Les transitions sont gérées par les conditions suivantes :

1. A : si la requête fait HIT
2. B : si la requête fait MISS
3. C : boucle tant qu'il ne reçoit pas la réponse favorable du slave (ACKNOWLEDGE*).
4. D : réponse favorable du slave
5. E : boucle tant que la rafale de données envoyées par la RAM(*) n'est pas fini
6. F : la dernière rafale de données est arrivée

*ACKNOWLEDGE : pour notre implementation, le signal acknowledge prends deux valeurs possibles :
- "ready", acknowledge = '1',
- "not ready" acknowledge = '0'

5.4.2 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. Ce cache est a été conçu en accord avec la politique Write-through. Cette stratégie va nous permettre maintenir la cohérence de la mémoire si l'on passe par la suite sur un système multi-core. 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 de ce cache contient 5 états distincts. De même que pour Icache, nous avons créé plusieurs machines à état et implémenter celle qui nous a semblé la plus efficace. Elle est constitué des états suivants :

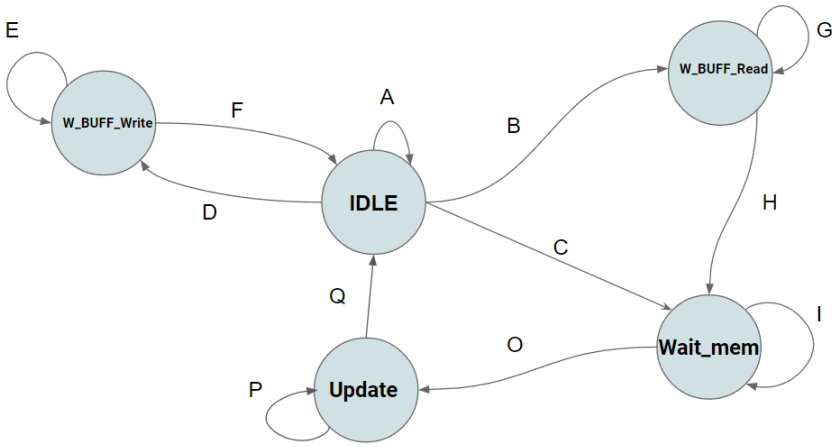


Figure 14: Machine à états du cache de données

1. IDLE
   État par défaut qui répond aux requêtes de lecture du processeur. En cas d'écriture ou de miss de lecture, écrit la requête dans le buffer cache.
2. WAIT MEM
   Cet état attend la réponse de la RAM.
3. UPDATE
   Cet état est chargé de recevoir les données du bus et de les mettre dans les lignes de cache correspondantes.
4. WAIT BUFFER READ
   Cet état va boucler tant que le buffer cache se vide pour ensuite écrire la requête dans le buffer cache et passer à l'état WAIT MEM.
5. WAIT BUFFER WRITE
   Cet état va boucler en attentant que le buffer cache soit vide pour ensuite écrire la donnée dans le buffer cache.

Les transitions sont gérées par les conditions suivants:

1. A : si (load et HIT) ou (store et buffer n'est pas full)

2. B : si le buffer est plein, que l'instruction est un load et qu'elle fait MISS.
3. C : si le buffer n'est pas plein, que l'instruction est un load et qu'elle fait MISS.
4. D : si le buffer est plein, et que l'instruction est un store
5. E : boucle tant que le buffer est plein
6. F : le buffer n'est plus plein
7. G : boucle tant que le buffer est plein
8. H : le buffer n'est plus plein
9. I : boucle tant que le slave ne répond pas favorablement
10. O : slave répond favorablement donc on enregistre la première donnée de la rafale
11. P : boucle tant que la rafale n'est par terminée
12. Q : la rafale est terminée

5.5 Protocole de validation

5.5.1 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 Gtkwave 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 Gtkwave.

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 prends comme argument un fichier .s ou .c, il appelle ensuite le compilateur RISCV 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 jump 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 :

```
extern void _bad();
extern void _good();

__asm__(".section .text") ;
__asm__(".global _start") ;

__asm__("_start:");
__asm__("addi x2,x0, 0x100");
__asm__("addi x1,x1, 4");
__asm__("sub x2, x2,x1 ");
__asm__("jal x5, main");


int fib(int n) {
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return 1;
    }
    else {
        return fib(n-1) + fib(n-2);
    }
}



int main() {
    if (fib(10) == 55) {
        _good();
    }
    else {
        _bad();
    }
}
__asm__("nop");
__asm__("_bad:");
__asm__("    add x0, x0, x0");
__asm__("_good :");
__asm__("    add x1, x1, x1");
```

5.5.2 Gestionnaire d'exception

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 un linker script *seg.ld* définissant une section text et une section Kernel. Nous avons ensuite écrit un gestionnaire d'exception que nous avons placé à la base de la section Kernel. Nous avons ensuite modifié le makefile pour que ce dernier compile notre gestionnaire d'exception et enfin dans le fichier core_tb nous avons forcé le linker à aller chercher ce fichier pour linker le fichier que l'on souhaite exécuter sur le core ainsi que le gestionnaire d'exception. Cela nous a ainsi permis de toujours avoir un code à l'adresse du gestionnaire d'exception défini dans le matériel. En effet si l'on ne fait pas ca et qu'un programme génère une exception, lorsque Decod va charger MTVEC il n'y aura aucun code à l'adresse correspondante. Il est donc nécessaire de toujours chargé le gestionnaire d'exception lorsque l'on exécute un programme sur le Core.

```
/* Grouping sections into segments for the link editor. */


SECTIONS
{
    . = 0x10054 ;
    seg_text :
    {
        *(.text)
    }
    . = 0x80000000 ;
    seg_kernel :
    {
        *(.kernel)
    }
}

INPUT(tests/exception)
```

5.6 Suite de test officielle

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.

6 Problèmes rencontrés lors de la conception

Plusieurs problématiques ont surgi depuis le début du projet, mais grâce au travail d'équipe et a la bonne répartition des taches, nous avons pu les surmonter.

La première difficulté majeure a été celle de rendre fonctionnel le code SystemC [4] du MIPS qui nous a été fourni, en effet les conventions utilisées étaient difficiles à comprendre et le code était peu documenté. Une fois le code rendu compilable, il a fallu le décortiquer complètement, ce qui a pris beaucoup de temps.
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 dans Decod.
Les Bypass auront également été difficile à implémenter dans la mesure ou c'était la première fois que nous implémentions une architecture avec des Bypass. En effet il y a beaucoup de cas particulier auxquels il faut faire attention car ils peuvent poser de nombreux problèmes.
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. D'où le fait que ce chantier là a pris du retard et que nous le finirons en juin et non pas en mai comme annoncé durant le rapport de pré-soutenance.

7 Objectifs restants

Au début du mois de mai, ayant bien avancé sur le projet, nous avons choisi de le mettre en stand-by jusqu'à début juin où notre stage va commencer. En effet, nous avons beaucoup d'autres projets à rendre et nous avons atteint la quasi-totalité des objectifs que nous nous étions posé. Le processeur que nous avons actuellement est un RV32I-Zcsr avec deux caches séparés et une gestion des exceptions/interruptions.
Nous prévoyons de continuer ce projet au cours de notre stage de 3 mois au Lip6. Les objectifs principaux que nous nous sommes fixés sont :

1. Réaliser une plateforme de TP RISC-V qui pourrait être ensuite utilisée dans le Master
2. Réaliser une implémentation RISC-V tournant sur FPGA
3. Étoffer le processeur, pour rajouter des extensions et des optimisations (Superscalaire, prédiction de branchement, extensions A et M).

Après discussion avec Mr. Wajsbürt nous avons convenu que l'implémentation FPGA aura pour but de viser une plateforme Altera De10 Lite. Au cours de ce stage, nous aurons la joie d'accueillir un 4ème membre, Mr. Samy Attal qui sera en charge de la partie FPGA/VHDL. Le tableau ci-dessous fait office de Roadmap, il s'agit d'un récapitulatif graphique des objectifs que nous nous sommes fixés et les deadlines correspondantes.

	Juin	Juillet	Aout				
Fin de l'implémentation Kernel et des tests	X						
Debeug Interface Icache/core	X						
Implémentation extension M	X						
SS2			X				
Prédiction de branchement		X					
Plateforme de TP			X				
1ere description VHDL (scalaire simple)	X						
2eme description VHDL (scalaire avec Kernel/CSR)		X					
Description finale			X				

X : Représente le mois où la tache devra être finalisée.

Timothée Le Berre	
Louis Geoffroy Pitailler	
Kevin Lastra	
Samy Attal	

L'implémentation du User Mode devrait être finalisé courant juin, car le chantier est déjà bien entamé. De même, les caches et l'extension M devrait également être fini au mois de juin.
En parallèle, Samy commencera à traduire tout le modèle SystemC en VHDL.
L'idée étant de paralléliser au maximum le travail, Samy sera en charge du modèle VHDL pendant que Timothée, Kevin et Louis se chargeront d'approfondir le modèle SystemC. Nous voulons en effet implémenter un SS2 et ajouter l'extension A ainsi qu'une prédiction de branchement.
La prédiction de branchement permet de gagner un cycle à chaque branchement et a aussi un intérêt pédagogique, car ce dispositif est aujourd'hui très courant dans les processeurs. Nous souhaitons donc savoir comment l'implémenter.
Enfin, s'il nous reste du temps, nous souhaitons passer sur une architecture multi-coeur. Mais de même que

c'était le cas au mois de mars pour la partie Kernel, il est difficile de se projeter si loin, car nous n'avons jamais implémenté de SS2, cela peut donc prendre plus de temps que prévu.
Enfin, Timothée qui sera également chargé de mettre en place une plateforme de TME pour les M1 ou les M2, cela est encore en discussion.

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] 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.

[9] 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.

[10] David A. Patterson John L. Hennessy (2021). Computer organization and design RISC-V edition, second edition.

[11] Jurij Šilc, Jurij Silc, Borut Robic, Theo Ungerer (1999). Processor architecture : From dataflow to superscalar and beyond.

[12] https://github.com/lovisXII/RISC-V-project/tree/main/Compte_rendu

[13] https://www.researchgate.net/figure/Conventional-Array-Multiplier_fig1_306034550