Rapport projet VLSI : Conception logique d'un processeur ARM

Thaïs MILLERET 21101028 - Guillaume REGNAULT 21107756

Responsable de l'UE : Jean-Lou DESBARBIEUX
Sorbonne Université Master Informatique Semestre 1 09/2024 - 01/2025

Introduction

Dans le cadre de l'UE Conception de circuits intégrés numériques, nous avons réalisé en partie un processeur capable de décoder et exécuter le jeu d'instructions ARM. Le processeur est décrit à l'aide du langage de description matérielle VHDL.

1. Architecture du processeur

L'objectif principal est que le processeur puisse exécuter les instructions du jeu d'instructions ARM v2.3 conformément à leur spécification.

Comme pour n'importe quel processeur, nous essayons d'obtenir un CPI (nombre de Cycles Par Instruction) le plus bas possible, proche de 1. Le CPI est le nombre moyen d'instructions exécutées par le processeur par cycle d'horloge. Ainsi, nous cherchons à implémenter les instructions de manière à ce qu'elles s'exécutent en un nombre minimal de cycles.

Un autre objectif est d'avoir une période d'horloge courte. Pour cela, il faut que le temps de propagation dans l'étage le plus long (chemin critique) soit court. C'est possible en optimisant la réalisation de manière à limiter le nombre de portes logiques à traverser par étage.

Le jeu d'instructions ARM comprend des instructions de complexité très variable. Le cours propose donc de concevoir un pipeline asynchrone. On évite ainsi le gel simultané de tous les étages.

Un pipeline doit suivre trois règles :

1. les étages sont équilibrés (le temps de propagation de l'instruction dans un étage est similaire pour tous les étages)
2. les étages sont séparés par un matériel permettant la transmission des données à travers le pipeline (par exemple un banc de registres)
3. chaque matériel appartient à un seul étage

Une architecture asynchrone

La progression des instructions dans les étages du pipeline peut être découplée. Cela implique des contraintes dans la gestion de la propagation d'une instruction dans le pipeline.

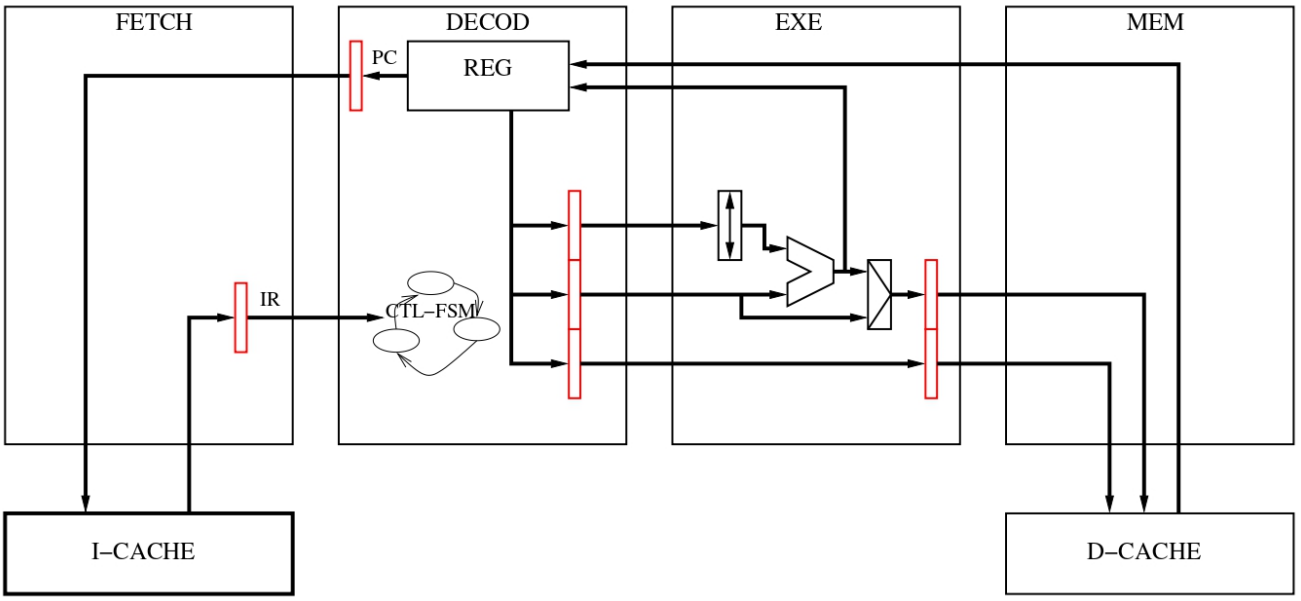
Par rapport à la règle 2 du pipeline, on ne peut pas se contenter d'un banc de registres simples. À la place, les étages du pipeline sont séparés par des FIFO (structure qui respecte l'ordonnancement First In First Out). La progression d'une instruction d'un étage au suivant dépend donc de l'état de la FIFO qui sépare les deux étages.
Dès que la FIFO qui alimente un étage n'est pas vide, c'est-à-dire que la FIFO contient les informations liées à une instruction, l'exécution de cette instruction dans cet étage peut commencer.

4 étages

L'architecture comporte 4 étages :

1. IFETCH : récupération de l'instruction
2. DECOD : décodage de l'instruction et write-back (écriture du résultat de l'instruction dans le banc de registres visible du logiciel)
3. EXE : calcul arithmétique ou logique
4. MEM : accès mémoire (lecture ou écriture)

Schéma du processeur :



La réalisation de IFETCH et MEM nous est fournie. Nous devons concevoir les étages EXE et DEC à l'aide des cours et des sujets de TP.

Registres

Il y a 16 registres User, en particulier :

- 13 : SP = Stack Pointer, pointeur vers le sommet de la pile
- 14 : LR = Link Register, adresse de retour de fonction
- 15 : PC = Program Counter, pointeur d'instruction dans le code
- 16 : CPSR = Current Program Status Register

Le registre CPSR contient des flags générés par l'ALU et mémorisés si l'instruction est suffixée S :

- N : le résultat de l'ALU est négatif
- Z : le résultat de l'ALU est égal à 0
- C : retenue générée par l'ALU dans le cas des instructions arithmétiques et par le shifter dans le cas des instructions logiques
- V : dépassement de capacité dans le cas d'une opération arithmétique signée

2. Jeu d'instructions ARM

Une instruction ARM peut être exécutée ou non selon un prédicat.

Schéma des prédicats (Cours 3 : Présentation architecture et jeu d'instructions ARM) :

0000 **EQ** - $Z = 1$
0001 **NE** - $Z = 0$
0010 **HS**/**CS** - $C = 1$
0011 **LO**/**CC** - $C = 0$
0100 **MI** - $N = 1$
0101 **PL** - $N = 0$
0110 **VS** - $V = 1$
0111 **VC** - $V = 0$
1000 **HI** - $C = 1$ et $Z = 0$
1001 **LS** - $C = 0$ ou $Z = 1$
1010 **GE** - supérieur ou égal
1011 **LT** - strictement inférieur
1100 **GT** - strictement supérieur
1101 **LE** - inférieur ou égal
1110 **AL** - toujours
1111 **NV** - réservé.

Cette partie présente rapidement les types d'instruction que le processeur doit pouvoir exécuter.

Traitement de données

Ces instructions utilisent l'ALU, et parfois le shifter, pour faire des calculs arithmétiques et logiques.

Multiplications

La multiplication est une opération arithmétique complexe qui implique plusieurs additions. Nous n'avons pas eu le temps de l'implémenter.

Branchements

Un branchement est une rupture de séquence : au lieu d'exécuter l'instruction suivante du code assembleur, on saute vers une autre instruction qui peut être située avant comme après l'instruction séquentielle.

Accès mémoire simples

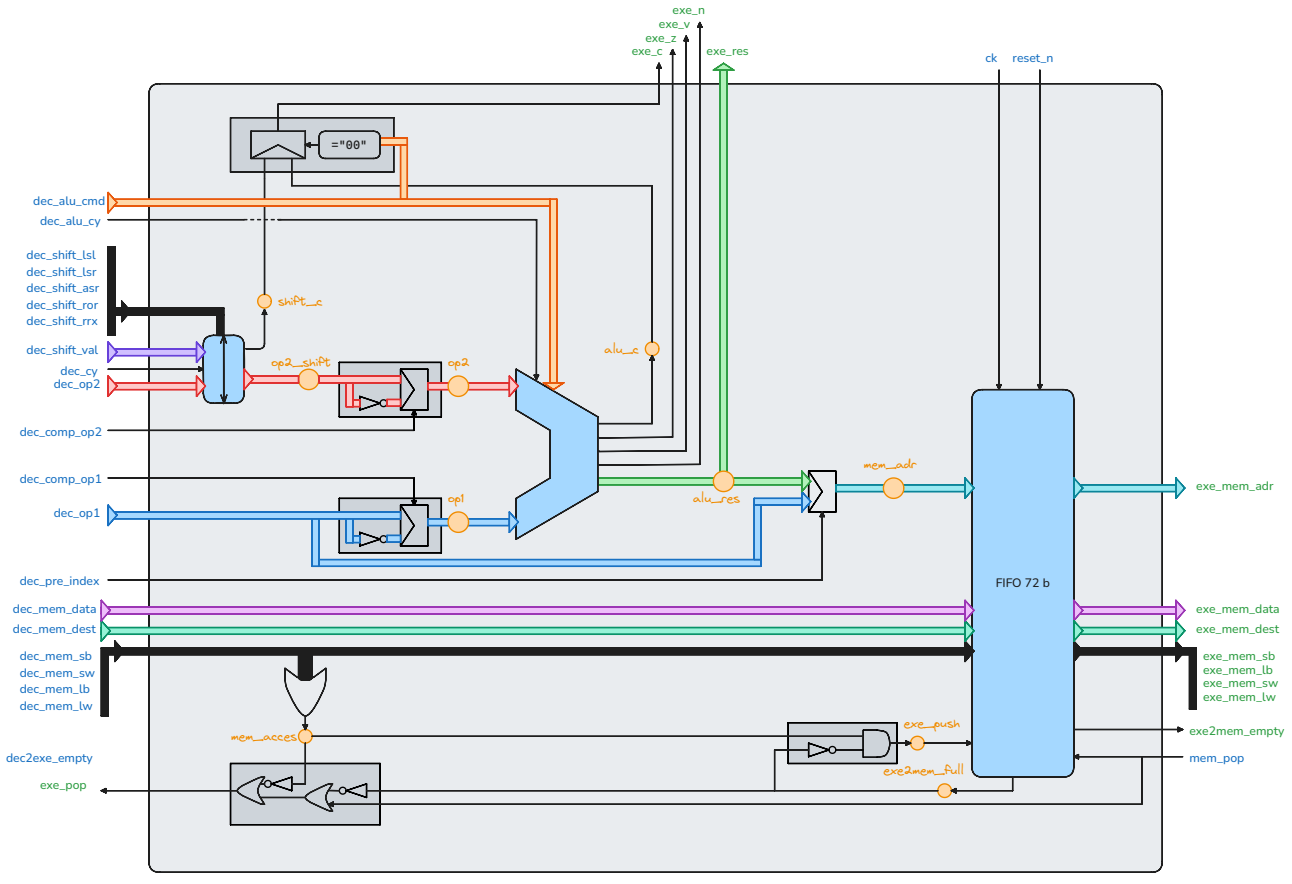
Ce type d'instruction correspond à un accès mémoire unique : on lit ou écrit un mot (ou un octet).

Accès mémoire multiples

Ce type d'instruction correspond à la lecture ou l'écriture de plusieurs registres du banc de registres User à la fois. On les utilise pour gérer la pile lors des appels de fonction. Nous n'avons pas eu le temps de l'implémenter.

3. Étage EXE

Schéma global de l'étage EXE (cf. fichier SVG pour zoomer) :



EXE est principalement constitué d'un shifter (décaleur) et d'une ALU. Il s'occupe des calculs arithmétiques et logiques. Cela permet le traitement des instructions arithmétiques et logiques, mais aussi du calcul d'adresse pour les transferts mémoire et les branchements.

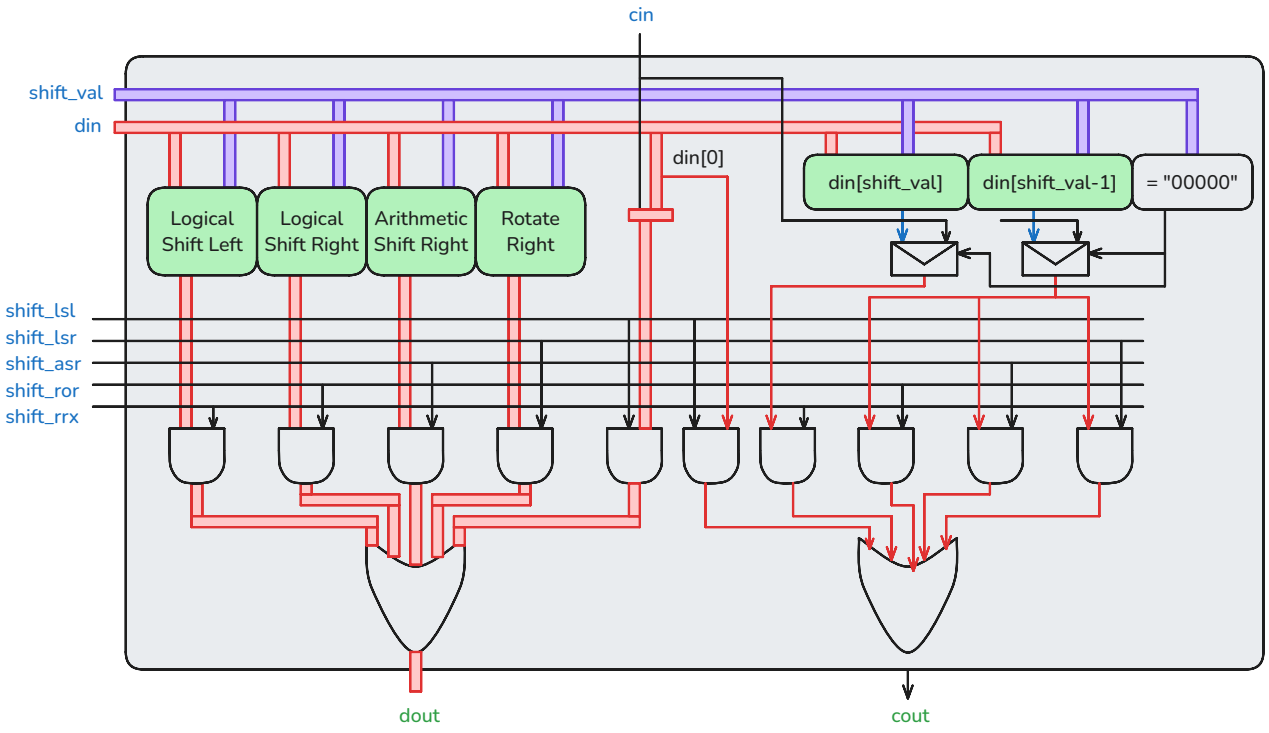
Il reçoit ses instructions de l'étage DECOD via une FIFO (profondeur d'une instruction codée sur 32 bits) et renvoie ses résultats au banc de registres REG situé dans DECOD. Les instructions mémoire sont directement transmises vers l'étage MEM via une FIFO.

Les instructions arithmétiques et logiques peuvent avoir jusqu'à 3 opérandes sources. Les deux premiers opérandes (dec_op1 et dec_op2) peuvent être inversés, d'où la présence d'un inverseur et d'un multiplexeur pour traiter chaque opérande avant son entrée dans l'ALU. Le 2e opérande passe par le shifter de l'ALU car elle peut être décalée, parfois grâce au 3e opérande source.

Shifter

Le shifter s'occupe des décalages et des rotations (sur Op2 ou pour les instructions de traitement de données). Il effectue un décalage logique à gauche ou à droite, un décalage arithmétique à droite, une rotation à droite ou une rotation à droite de 1 bit avec retenue (si RRX) en fonction des indications transmises par DECOD (via les entrées shift_lsl, shift_lsr, shift_asr, shift_ror et shift_rrx).

Schéma du shifter :

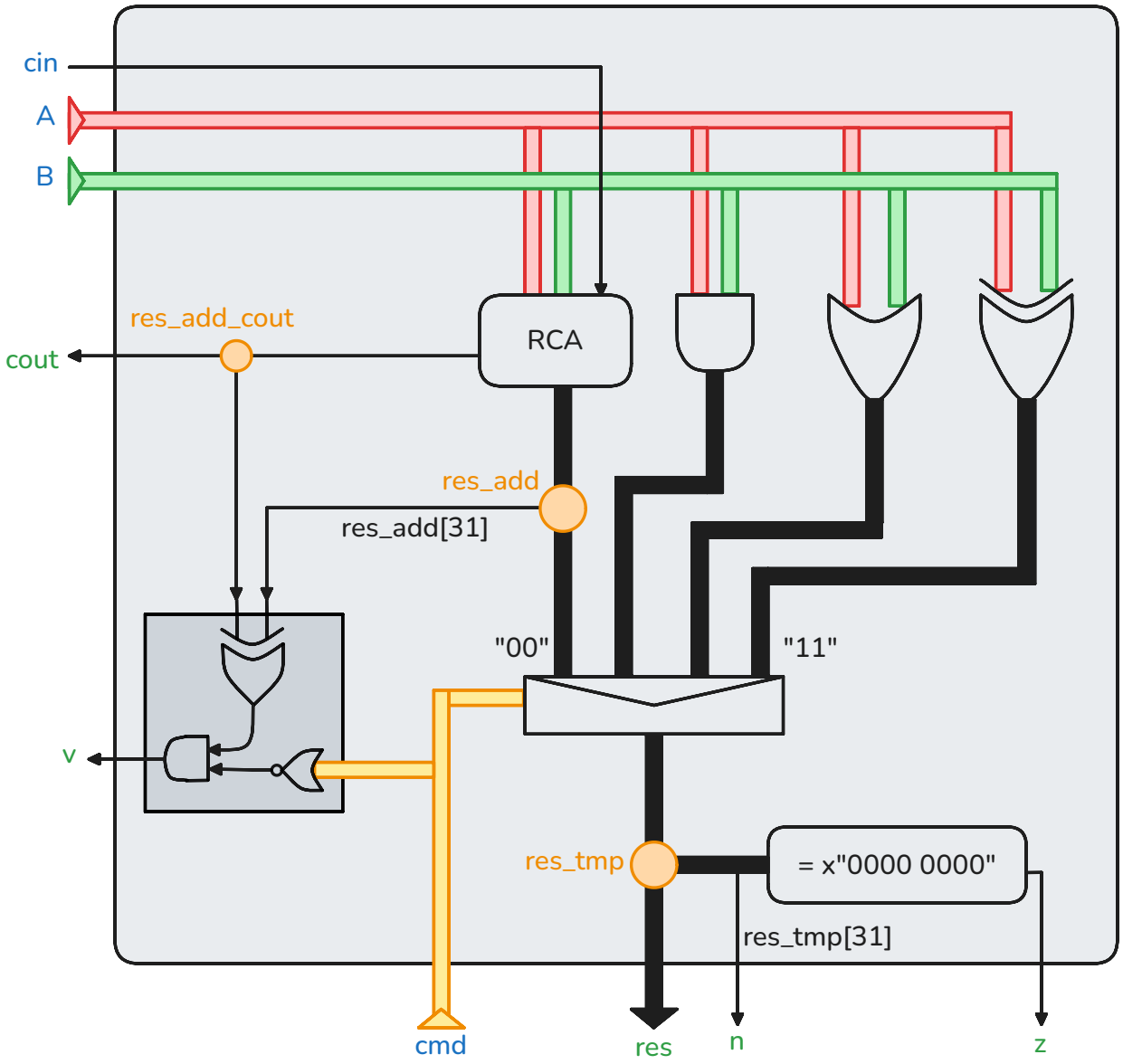


cin est la valeur à shifter sur 32 bits, dout est le résultat de l'opération, cin et cout sont les retenues respectivement entrante et sortante, shift_val est un entier sur 4

bits qui correspond à la valeur du shift (0 à 15).

ALU

Schéma de l'ALU :



L'Unité Arithmétique et Logique peut réaliser 4 opérations de base : addition, ET logique, OU logique et OU exclusif. Elle reçoit 2 opérandes (sur 32 bits) A et B et une retenue entrante cin. Elle fournit le résultat res (sur 32 bits) et 4 flags Z, N, V et Cout (ce sont les flags enregistrés dans le registre CPSR).

L'entrée cmd spécifie l'opération à effectuer :

- 00 : addition ADD avec la retenue cin,
- 01 : ET logique AND,

- 10 : OU logique OR,
- 11 : OU exclusif XOR.

Nous avons utilisé les opérateurs AND, OR et XOR fournis par VHDL.
Pour l'addition, on aurait pu utiliser l'opérateur + de VHDL sur les vecteurs A et B. Nous avons choisi d'implémenter un additionneur à propagation de retenue (Ripple Carry Adder) en branchant en série des Full Adder.

L'ALU réalise les 4 opérations mais une seule est envoyée en sortie via un multiplexeur dont le sélecteur est cmd.

Test bench de EXE

Le shifter et l'ALU ont été testés séparément pour limiter les tests à faire sur EXE dans sa globalité.

Le test bench du shifter contient des tests pour les 5 opérations (LSL, LSR, ASR, ROR et RRX) pour des valeurs aléatoires.

Le test bench de l'ALU contient des tests pour les 4 opérations (ADD, AND, OR et XOR) pour des valeurs aléatoires et des cas extrêmes. Nous avons également testé le bon fonctionnement des flags.

Pour tester EXE, nous avons envoyé artificiellement des instructions décodées à EXE (comme si elles venaient de DECOD). Le test est cadencé par une horloge. On vérifie la sortie de EXE vers REG (valeurs des signaux, par exemple par rapport à write-back) ou MEM (adresse, data, indications pour MEM). On vérifie également l'état de la FIFO qui relie EXE à MEM : on teste si elle est vide/pleine avant et après avoir pop/push dessus. L'état de la FIFO peut être également vérifié avec GTKWave.

4. Étage DECOD

DECOD contient un banc de registres REG et une machine à états. Il doit assurer deux fonctionnalités :

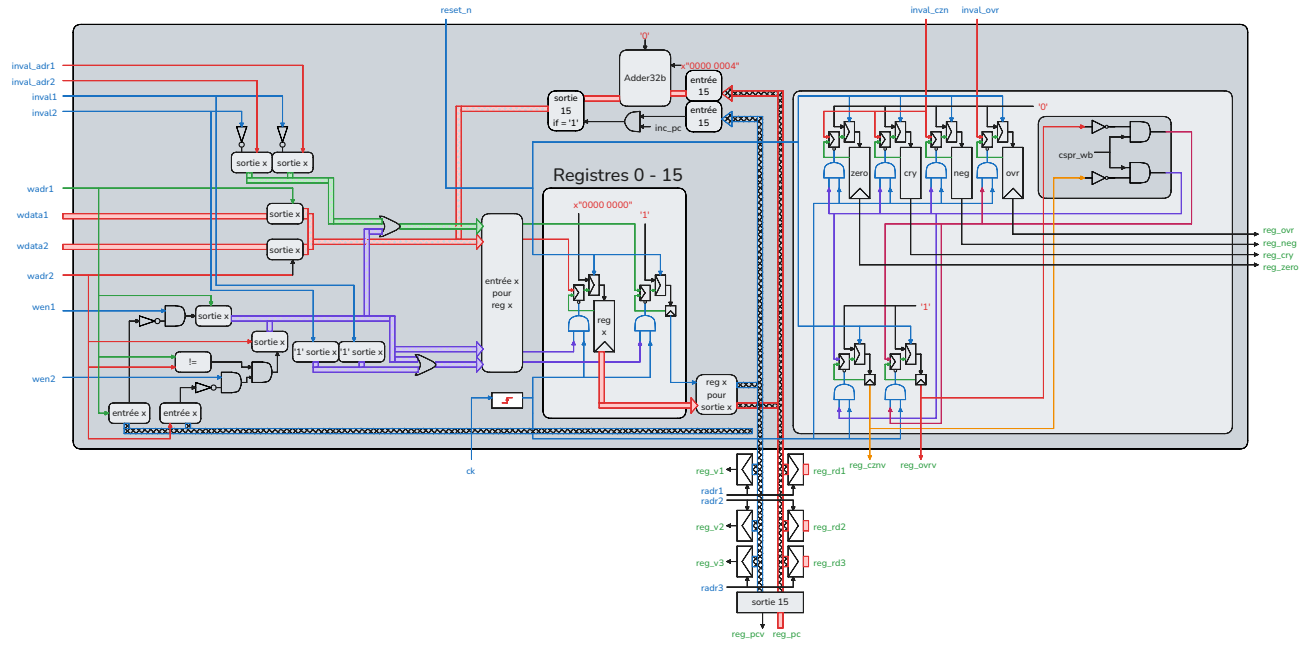
- décoder les instructions pour permettre leur exécution par les étages EXE et MEM,
- assurer le séquencement du pipeline, gérer les aléas et traiter les instructions multicycles (comme les transferts multiples).

L'instruction est reçue sous la forme d'un mot de 32 bits. DECOD la convertit en un mot de 127 bits interprétable par EXE et/ou MEM. C'est pourquoi DECOD contient un très grand nombre de signaux qui sont mis à jour selon la machine à états. Les signaux sont regroupés en 4 catégories : décodage des instructions, commande de EXE, commande MEM et séquencement/contrôle du pipeline (interaction avec les FIFO).

Comme l'architecture du processeur est asynchrone, il faut gérer la synchronisation des étages pour une instruction. Une instruction n'est lancée que si tous ses opérandes sources et les flags (stockés dans CSPR) sont valides. Le registre de destination de l'instruction est marqué comme non valide lors du lancement de l'instruction. Le registre de destination repasse à valide lorsque le résultat de l'instruction est disponible.

Banc de registres REG

Schéma de REG (cf. fichier SVG pour zoomer) :



Le banc de registres contient les 16 registres décrits dans la partie "Architecture du processeur" et les 4 flags fournis par l'ALU (C, Z, N et V).

Un registre contient une entrée din, une sortie dout, une horloge et un signal de reset (le reset peut être asynchrone par rapport à l'horloge). Si reset_n est à 0 on remet le registre à 0. Sur le front montant de l'horloge, on copie la valeur de din dans dout. La synchronisation des étages du pipeline nécessite que les registres

soient associés à des bits de validité : 1 bit pour la valeur du registre, 1 pour les flags C, N et Z et 1 pour le flag V.

Au reset, tous les registres sont considérés comme valide pour qu'on puisse lancer une instruction (les registres sources doivent être valides). Quand un registre est identifié par DECOD comme registre de destination d'une instruction, il est invalidé. Quand un résultat produit par EXE ou MEM est écrit dans REG, le registre de destination est validé. On ne peut écrire dans un registre que s'il est marqué comme invalide.
De même, on ne peut mettre à jour les flags que si leur bit de validité respectif est à 0. Les instructions logiques écrivent C, N et Z. Seules les instructions arithmétiques affectent V.

MEM et EXE peuvent produire simultanément un résultat. En cas de conflit (les résultats de MEM et EXE vont dans le même registre de destination), on ignore l'écriture de MEM car elle est nécessairement plus ancienne. En effet, une instruction exécutée par EXE ne peut être lancée que si ses opérandes sources sont à jour (registres sources valides) donc le résultat produit par EXE est effectivement plus récent que n'importe quelle donnée chargée par MEM.

Le registre 15 (PC) est un registre particulier. On lui associe un opérateur réalisant l'opération +4 et son contenu et sa validité sont accessibles directement via l'interface de REG. On remarque ici que le calcul de l'instruction suivante se fait dans DECOD.

Machine à états

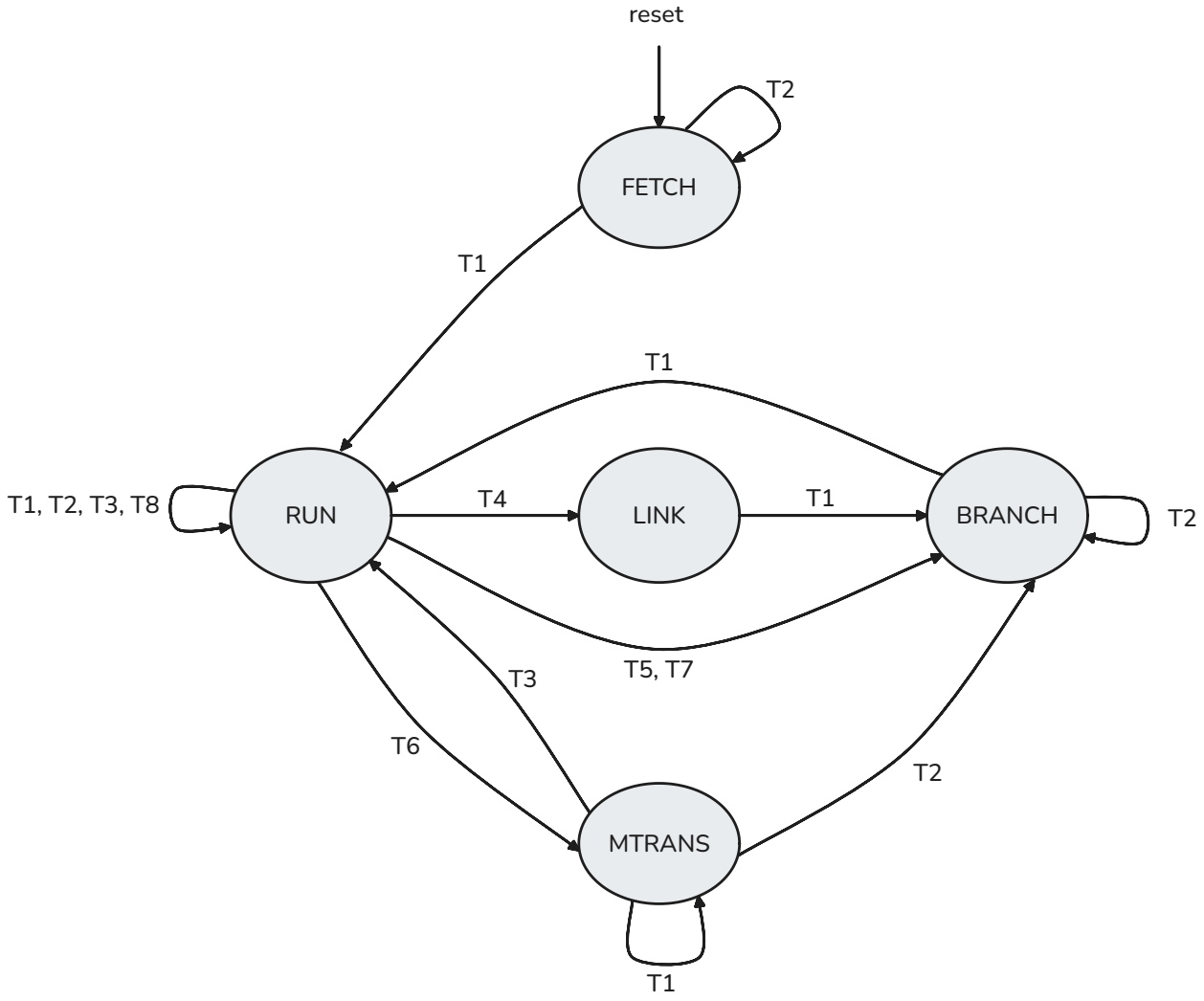
Le contrôle du flot d'instructions par DECOD repose sur une machine à états et suit une unique séquence, parfois incomplète selon l'instruction à exécuter :

1. envoyer l'adresse d'une instruction dans la FIFO dec2if ;
2. lire l'instruction chargée par IFETCH et stockée dans la FIFO if2dec ;
3. décoder l'instruction chargée ;
4. écrire le résultat du décodage dans la FIFO dec2exec à destination de EXE.

La machine à états contrôle ainsi la lecture et l'écriture dans les FIFO dec2if, if2dec et dec2exec grâce à 3 signaux : dec2if_push, if2dec_pop (port dec_pop) et dec2exe_push.

Les multiplications, les branchements et les transferts multiples doivent être traités en plusieurs cycles.

Schéma de l'automate de DECOD :



État FETCH

C'est l'état de démarrage (après reset). On est dans l'état FETCH lorsqu'on attend une instruction à traiter. On attend que le buffer d'instructions ne soit plus vide et que le PC soit valide.

Transition	États	Condition en VHDL	Explications
T1	(FETCH -> RUN)	dec2if_full = '1' and reg_pcv = '1'	On a des instructions et le PC est valide.
T2	(FETCH -> FETCH)	else	

État RUN

Le rôle de cet état est de décoder l'instruction entrante et de l'exécuter si rien de spécial n'est nécessaire.

Transition	États	Condition en VHDL	Explications
T1	(RUN -> RUN)	if2dec_empty = '1' and if2dec_pop = '0'	Cas où IFETCH n'a pas reçu d'adresse d'instruction (juste pour la sécurité car ce cas ne devrait pas arriver)
T2	(RUN -> RUN)	condv = '0' or operv = '0' or (dec2exe_full = '1' and exe_pop = '0')	Les opérandes ou les flags ne sont pas valides ou on ne peut pas push l'instruction que l'on va décoder.
T3	(RUN -> RUN)	cond = '0'	L'instruction n'est pas valide (condition d'exécution).
T4	(RUN -> LINK)	branch_t = '1' and blink = '1'	L'instruction est un branch and link.
T5	(RUN -> BRANCH)	branch_t = '0' and blink = '0'	L'instruction est un branchement.
T6	(RUN -> MTRANS)	mtrans_t = '1'	L'instruction est un transfert multiple.
T7	(RUN -> BRANCH)	(alu_dest = x"F" and alu_wb = '1') or (ld_dest = x"F" and (mem_lw = '1' or mem_lb = '1'))	Cas où une autre instruction qu'un branchement agit sur PC. (Ne devrait arriver en aucun cas sauf lors du return d'une fonction où on fait `MOV PC, LR`.)
T8	(RUN -> RUN)	else	C'est une instruction standard (mult, load, store, regular operation, swap).

État LINK

Cet état vient du fait que l'on sauvegarde le PC (dans le Link Register) avant de le modifier. Il ne sert qu'à enlever le flag de link avant de laisser le décodage envoyer l'instruction de modification de PC.

Transition	États	Condition en VHDL	Explications
T1	(LINK -> BRANCH)	aucune	toujours

État BRANCH

On attend dans cet état que le PC redevienne valide et que les buffers d'instruction et d'adresse soient vides. Il purge l'instruction suivant un branchement pris.

Transition	États	Condition en VHDL	Explications
T1	(BRANCH -> RUN)	reg_pcv = '1' and if2dec_empty = '1' and dec2if_empty = '1'	Tous les buffers ont été purgés et le PC est valide.
T2	(BRANCH -> BRANCH)	else	

État MTRANS

Cet état sert pour les transferts multiples. Il n'est actuellement pas fonctionnel dans notre projet et renvoie juste à l'état RUN.

Voici les conditions théoriques de changement d'état :

Transition	États	Condition en VHDL	Explications
T1	(MTRANS -> MTRANS)	mtrans_list != x"0"	Il reste des registres à transférer.
T2	(MTRANS -> BRANCH)	mtrans_list = x"F" and ldm_i = '1'	On a modifié le registre PC.
T3	(MTRANS -> RUN)	else	On a traité tous les registres et le PC n'a pas été modifié.

Idées pour les multiplications

Une idée est d'implémenter l'algorithme de Booth pour avoir des multiplications qui s'exécutent en peu de cycles.

Idées pour les transferts multiples

Nous n'avons pas réussi à implémenter les transferts multiples. Ce paragraphe explique comment on pourrait les gérer.

Si l'instruction est un transfert multiple, la machine à états nous amène à l'état MTRANS. On boucle dans l'état MTRANS. Tant que la mtrans_list n'est pas égale à x"0" ou x"F", on lance un load/store word du plus petit registre de la liste. Puis on le supprime une fois le tick d'horloge passé.
Dans le cas ou mtrans_list = x"F" et que ldm_i = '1', il faut passer à l'état BRANCH au moment du tick d'horloge car le registre de PC est invalide et va être modifié. Dans le cas contraire, on retourne à RUN à la fin.

Une autre problématique s'élève si on n'a pas de write-back. Dans ce cas l'idée est d'enregistrer au préalable la valeur du registre base dans un registre interne. Si on le modifie par un load dans l'instruction, alors il n'est pas gardé. Dans le cas où on n'a pas de write-back, on doit rétablir la valeur d'origine du registre sur lequel on a agi avant de lire la prochaine instruction. On utilise la valeur stockée dans le registre interne (instruction `OR rx, 0`).

Test bench de DECOD

Comme pour EXE, nous avons testé séparément REG pour simplifier le test bench de DECOD.

Le test bench de REG vérifie que les registres sont mis à jour au bon moment et renvoient la bonne valeur lorsqu'on les lit.

Nous testons DECOD directement dans le test bench du processeur. On vérifie que les entrées et sorties de DECOD sont celles attendues.
Le test bench du processeur s'occupe de simuler la RAM sauf que l'on envoie des mots sans se soucier des adresses liées.

Nous avons testé chaque instruction d'opération standard, chaque condition d'exécution (valide et non valide) ainsi que les instructions de branchement et d'accès mémoire simple.

5. Plateforme globale

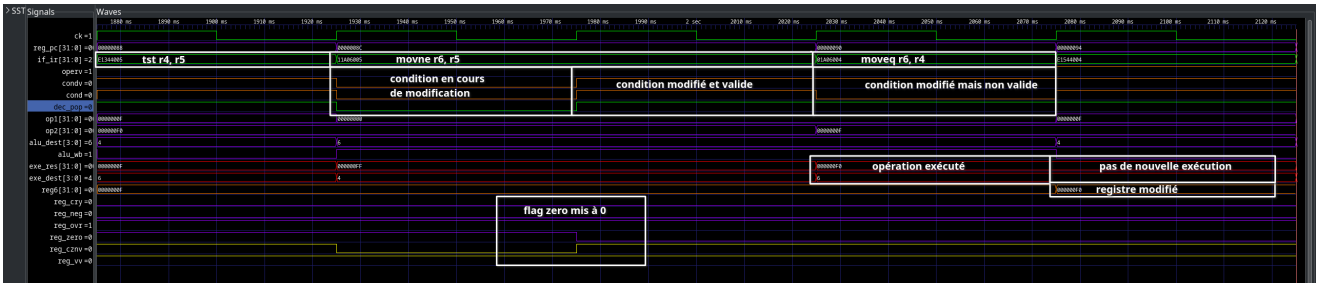
Nous avons écrit un test bench qui simule le processeur : on instancie et connecte les 4 étages IFETCH, DECOD, EXE et MEM. Le test est cadencé sur une horloge globale. Il y a un processus qui envoie les instructions à tester à IFETCH et un processus qui vérifie les valeurs en sortie de DECOD.

Nous vérifions les valeurs contenues dans les registres de REG grâce à GTKWave.

Test d'instructions arithmétiques et logiques

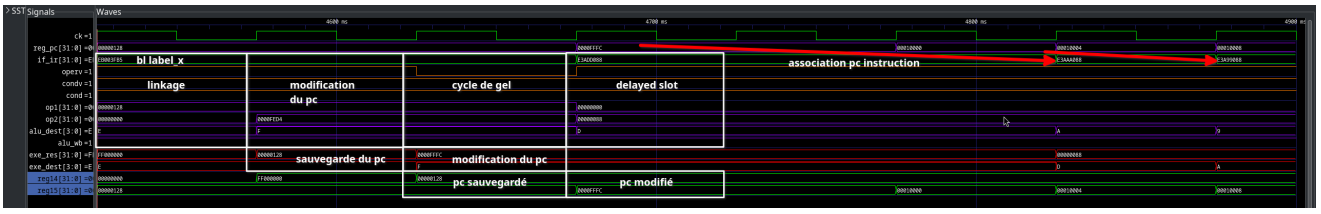
Pour une instruction arithmétique ou logique, nous vérifions la valeur du registre destination et les flags. Pour vérifier les flags set par une instruction, on la fait suivre par un MOV qui met une valeur arbitraire dans un registre. Si les flags sont corrects, le registre contient cette valeur après l'exécution de l'instruction et du MOV. Ce test avec MOV permet aussi de vérifier que les prédicats fonctionnent.

Tous les tests réussissent donc on n'a pas de message d'erreur sur le terminal lorsqu'on exécute le test bench. Voici une capture d'écran de GTKWave :



Test de branchement

Pour un branchement, nous vérifions la valeur du registre PC. Nous pouvons également vérifier que le branchement a fonctionné en regardant la valeur de PC dans GTKWave.



Test d'accès mémoire simple

Pour un store, nous vérifions que MEM reçoit la bonne valeur à stocker dans la RAM.
Pour un load, nous vérifions avec GTKWave que le registre contient la bonne valeur.

Pour un exemple d'accès mémoire simple sur GTKWave, voir le test de programme qui suit.

Test de programme : somme d'un vecteur

Nous avons testé la plateforme globale avec un programme simple : la somme des éléments d'un tableau.

Programme en C :

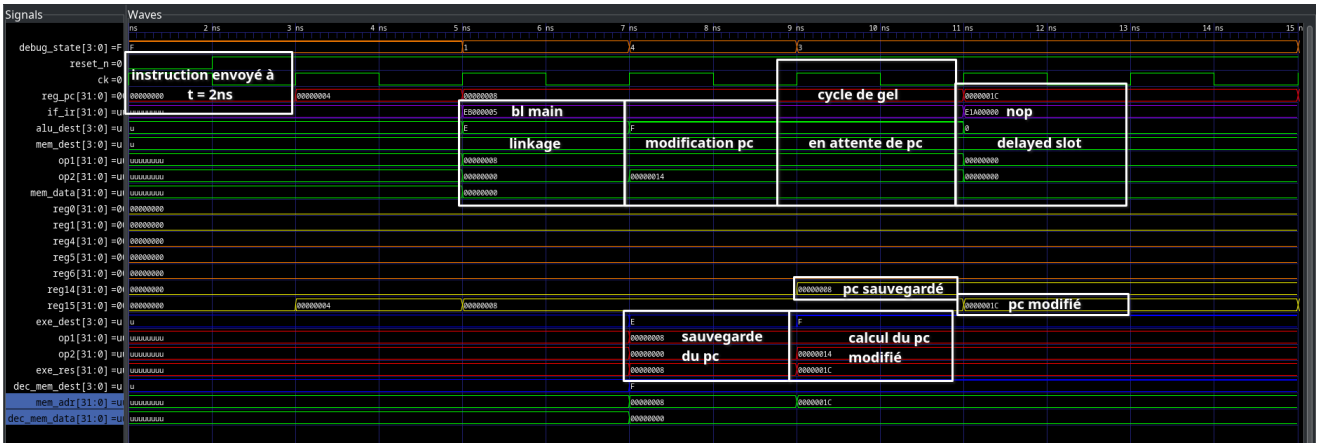
```c
int tab[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int main() {
        int sum = 0;
        int * ptr;
        for (ptr = tab; ptr < &tab[10]; ptr++) {
                sum = sum + (*ptr);
        }
        return sum;
}
```

Traduction manuelle en assembleur :

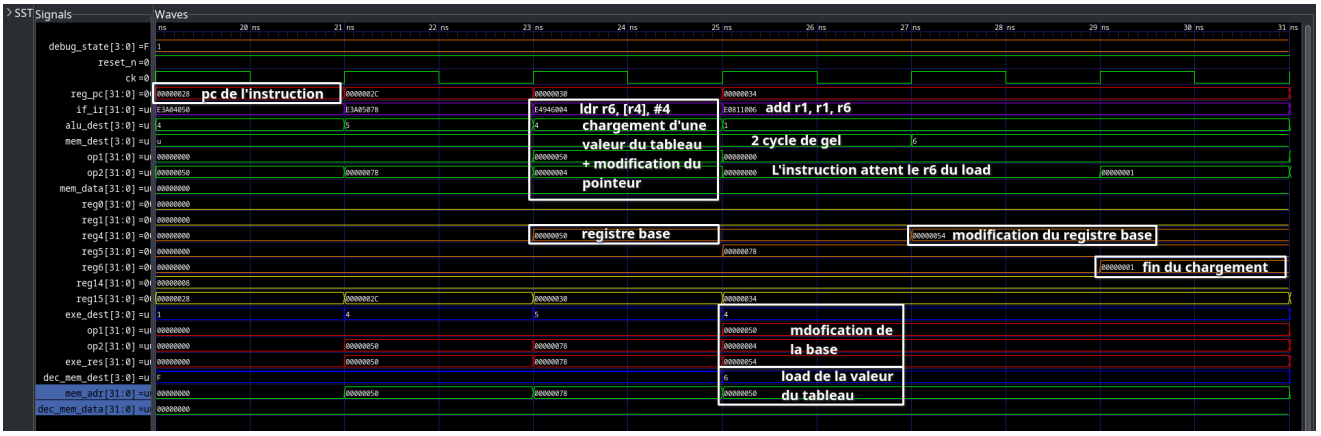
```
        .text
        .globl _start
_start:
        bl main
        nop
        mov r10, #TTY_out
        str r0, [r10]
        b _good
        nop
        b _bad
```

```
main:
        mov r1, #0
        mov r4, #AdrTab
        mov r5, #AdrTabFin
main_loop:
        ldr r6, [r4], #4
        add r1, r1, r6
        cmp r4, r5
        bne main_loop
        nop

        mov r0, r1
        mov pc, lr
        mov r0, r0
        b _bad
        mov r0, r0

AdrTab:
        .word 0x01
        .word 0x02
        .word 0x03
        .word 0x04
        .word 0x05
        .word 0x06
        .word 0x07
        .word 0x08
        .word 0x09
        .word 0x0a
AdrTabFin:
        .word 0x10
TTY_out:
        .word 0x00

_bad:
        add r0, r0, r0
_good:
        add r1, r1, r1
```

Branch and link sur GTKWave (saut de `_start` vers `main`) :

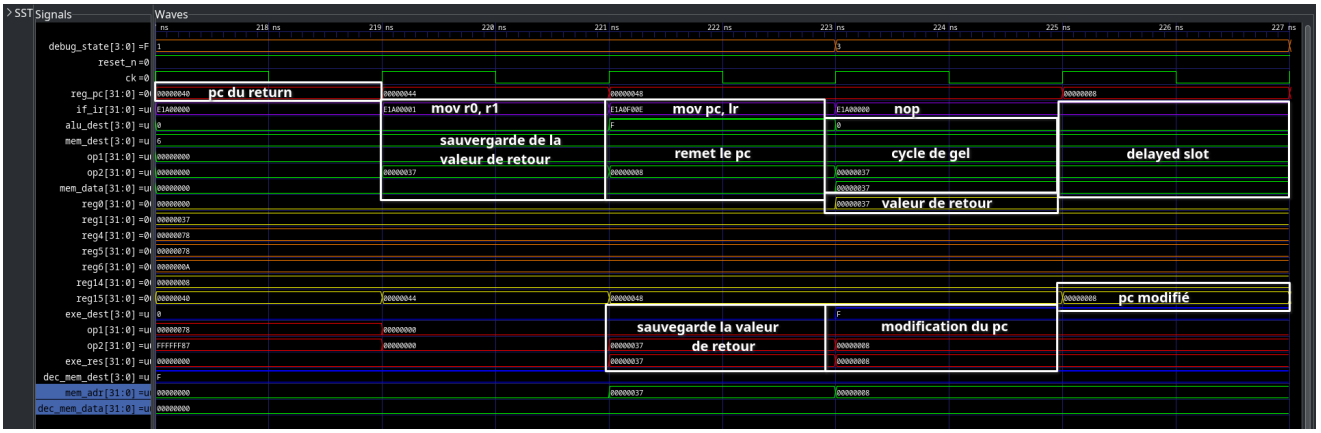


Load avec postindex sur GTKWave :



Le load est dans la boucle for. La capture d'écran correspond à l'itération 0.

Return sur GTKWave (return du main) :



Trace dans le terminal :

```
Chargement du segment .text adr = 0x0
Chargement du segment pile adr = 0x7ffff000 Taille = 0x1000
Symbol _good found at adr=84
Symbol _bad found at adr=80
main_tb.vhdl:231:17:@241ns:(report note): TTY out : 0x00000037
main_tb.vhdl:218:9:@246ns:(assertion note): GOOD!!!
main_tb.vhdl:220:9:@246ns:(assertion note): end of test
```

Conclusion

Nous avons pu simuler la majeure partie du jeu d'instructions ARM v2.3 sur notre processeur. Il manque juste la multiplication, les transferts multiples et les swaps. Nous avons aussi pu tester un petit programme simple écrit en assembleur.

Le code VHDL constitue la description logique de notre processeur (même s'il manque la gestion des transferts multiples). Avec plus de temps, nous aurions pu faire la synthèse logique puis le placement routage, c'est-à-dire récupérer toutes les cellules logiques qui correspondent à notre description (synthèse) et les placer pour obtenir un schéma des masques (routage).

Ce projet nous a permis d'approfondir nos connaissances sur le fonctionnement et la conception d'un processeur. Nous avons réalisé que la problématique liée à la synchronisation des étages du pipeline était plus complexe à résoudre que ce que nous avions appris auparavant. Il ne suffit pas de découper le processeur en étages à peu près équivalents, il faut aussi gérer les instructions qui prennent plusieurs cycles comme les transferts multiples et les cycles de gel.