VLSI

Rapport de Projet

Description Matérielle du Processeur ARM

2023 - 2024

Youba FERHOUNE
Drifa AMIRI

Contents

1 Introduction

Dans le cadre de ce projet, l'objectif est de détailler le design d'un processeur basé sur l'architecture ARM. La finalité est que ce processeur puisse exécuter de manière complète un programme écrit en langage assembleur ARM. Cette modélisation est réalisée en utilisant le langage de description matérielle VHDL.

Le CPU que l'on modélise est un processeur pipeline sur 4 étages :

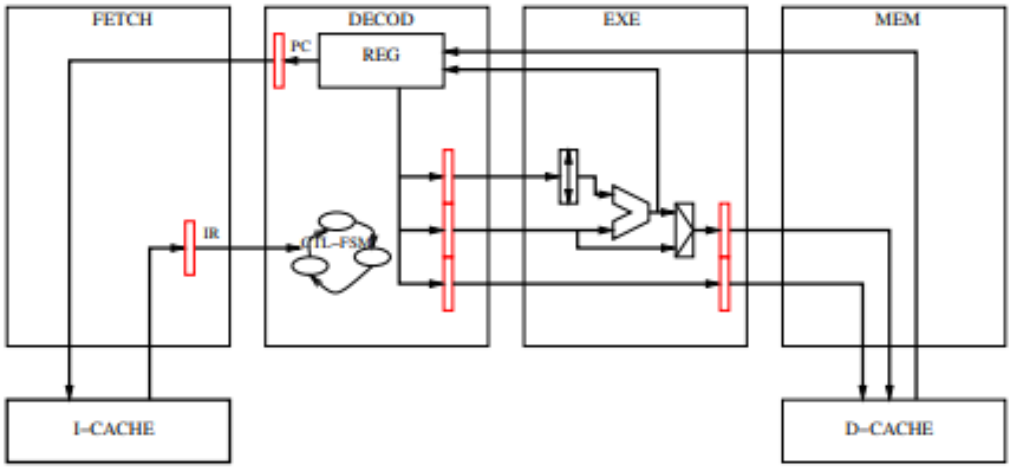

Figure 1: Schéma simplifie du pipeline

- FETCH : Cette étape récupère l'instruction en mémoire et la transmet à l'étape de décodage.
- DECOD : À ce stade, l'instruction chargée par l'étape Fetch est décodée. Cela implique la sélection des opérandes, des registres et des calculs nécessaires à son exécution. L'étape de décodage inclut également la gestion du banc de registres.
- EXE : Cette phase se consacre à l'exécution des opérations arithmétiques de base.
- MEM: Si l'instruction exécutée nécessite des accès mémoire, cette étape s'occupe de les effectuer.

2 EXE

2.1 Introduction

La première étape de notre modélisation concerne l'étage EXE, qui est assez simple à concevoir. Il se compose principalement de deux éléments essentiels : l'ALU (Unité Arithmétique et Logique) et le shifter. Voici une description de son organisation :

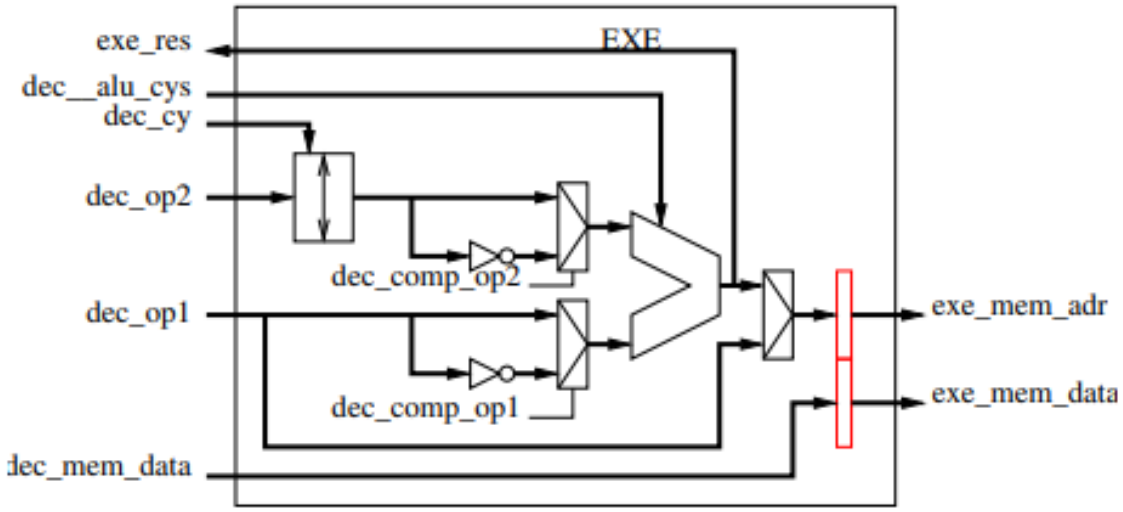

Figure 2: Schéma simplifie du l'étage EXE

2.2 ALU

Pour comprendre comment l'ALU fonctionne, imaginez-le comme le cerveau mathématique du processeur. Il peut effectuer différentes opérations comme "et", "ou", "xor" (ou exclusif) et "addition". Cela dépend d'un petit signal de contrôle de 2 bits qui lui dit quelle opération effectuer.

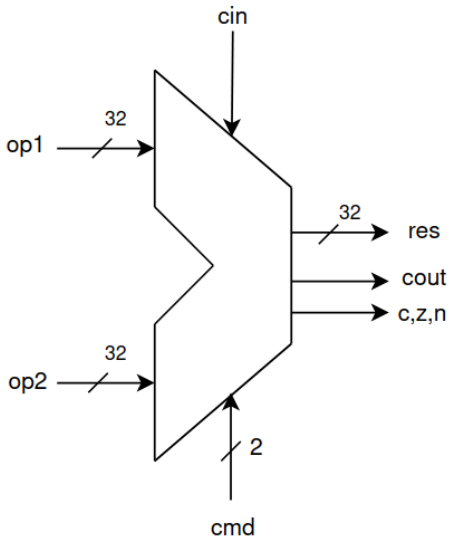

Figure 3: Schéma de l'ALU

CMD	Opération
00	Addition
01	And
10	Or
11	Xor

Table 1: les operations realises par l'ALU

Pour réaliser ces opérations, nous avons utilisé des fonctions logiques fournies par le VHDL et des conversions sur les opérandes pour pouvoir faire des additions car les additions se font que sur des Unsigned . Dans le cadre global de l'étage EXE, nous avons ajouté des inverseurs pour pouvoir réaliser des "complément à 2". Cela nous permet de faire des soustractions de manière plus efficace

2.3 Shifter

Le shifter est contrôlé par 5 entrées chaqu'une a un 1 bit spécifiant le type de décalage à effectuer, accompagnés de 5 bits indiquant la valeur du décalage. Ce composant permet d'effectuer des multiplications et des divisions par des puissances de 2. Par exemple, un décalage de 1 vers la droite correspond à une division par 2, tandis qu'un décalage de 1 vers la gauche équivaut à une multiplication par 2.

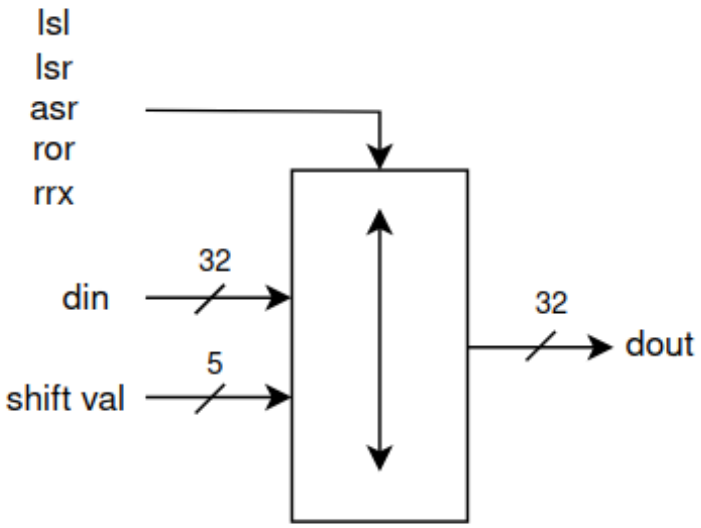

Figure 4: Schéma simplifie du shifter

Pour la rotation, le processus implique la concaténation des n bits (où n représente la valeur du shift) de poids faible avec les bits restants. Les bits restants sont concaténés à droite des bits de poids faible. Notre shifter est également conçue pour gérer les shifts RRX qui sont représentés comme un ROR avec une valeur de décalage de 0.

3 DECOD

DECOD représente la phase la plus cruciale et complexe du processeur. Sa responsabilité principale est de décoder les instructions provenant de l'unité IFETCH. De plus, il joue un rôle essentiel dans la gestion globale du pipeline en commandant les différents étages après avoir interprété l'instruction reçue. Cet étage se divise essentiellement en deux parties : le banc de registres, et le décodage des instructions et le contrôle général du processeur (machine à états).

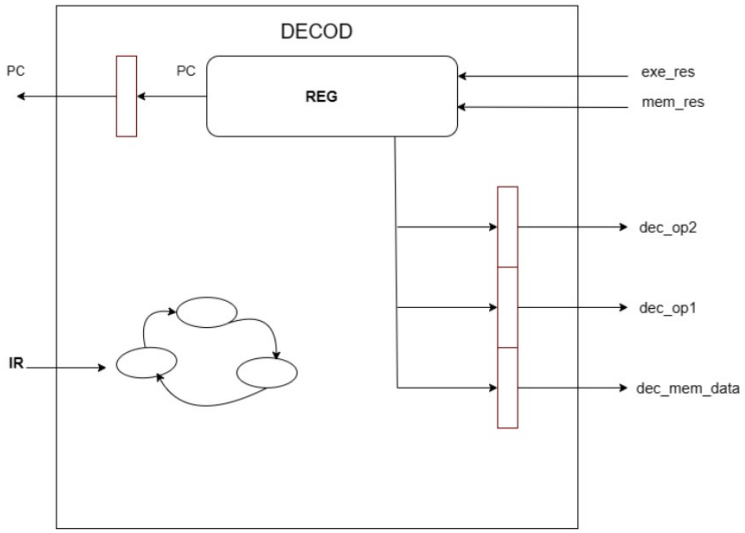

Figure 5: Schéma simplifie du DECOD

3.1 Banc de registres REG

Le banc de registres que nous avons réalisé est composé de 16 registres de 32 bits, accompagnés des flags C, Z, N et V. Ce banc de registres peut gérer deux écritures, avec l'une ayant la priorité sur l'autre en cas d'adresse identique. De plus, il permet trois lectures de 32 bits et une lecture de 5 bits spécifiquement dédiée à la valeur d'un shift.

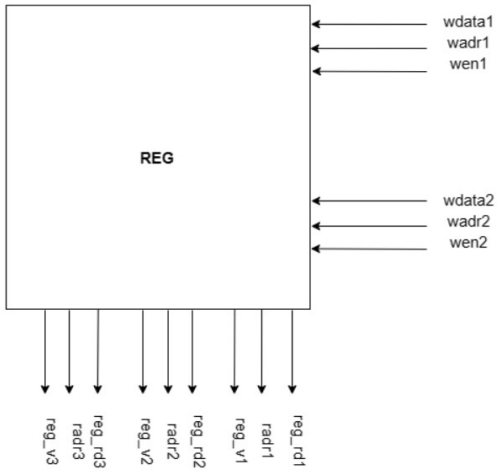

Figure 6: Schéma de REG

Chaque registre est associé à un bit de validité. Un registre est invalidé par l'unité de décodage (DECOD) lorsqu'une instruction qui y écrit est lancée. Il redevient valide lorsque le résultat de l'instruction est effectivement écrit dans le registre. Les trois flags C, Z et N partagent un seul bit de validité et un autre bit de validité est associé au flag V.

L'étage REG prend en entrée les registres à invalider, et la validité de chaque registre est rétablie dès qu'une opération d'écriture est effectuée sur celui-ci. Les opérations de lecture sont accompagnées du bit de validité correspondant au registre.

Le banc de registres est représenté par un tableau de vecteurs de type **'std_logic'**, et les bits de validité sont stockés dans un vecteur de type **'std_logic'**. Pendant les opérations de lecture ou d'écriture, les adresses subissent une conversion en entiers afin de servir d'index dans le tableau. Lors de l'écriture dans les registres et en cas de conflit d'adresse la priorité est donnée au premier registre écrit.

REG est également responsable de la gestion du PC (Program Counter) : si un flag de contrôle a la valeur 1, le PC est incrémenté de 4 ; sinon, il conserve sa valeur actuelle.

Pour gérer l'incrémentation du PC, un signal nommé **'inc_pc'** est utilisé pour indiquer si l'incrément PC + 4 doit se produire.

3.2 Décodage des instructions

DECOD reçoit des instructions sous forme de mots de 32 bits, nécessitant une interprétation pour en déchiffrer le sens.

Pour décoder l'instruction nous avons utilisé les informations fournies dans la documentation ARM présentée lors du cours.

Comme les 4 bits du poids fort de l'instruction représentent la condition d'exécution, nous avons commencé par coder ces conditions. Ces dernières sont récapitulées ci-dessous :

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.

Figure 7: predicats

Par la suite, nous avons entrepris de déterminer la nature de l'opération à effectuer. Nous avons défini quatre catégories d'opérations, énumérées ci-dessous.

3.2.1 Traitement de données

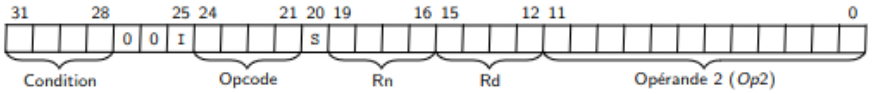

Figure 8: configuration d'une instruction de type traitement de données

Pour ce type, nous avons encodé les opcodes conformément à la figure ci-dessous :

0000 - **AND** : $Rd <= Rn\ AND\ Op2$
0001 - **EOR** : $Rd <= Rn\ XOR\ Op2$
0010 - **SUB** : $Rd <= Rn - Op2$
0011 - **RSB** : $Rd <= Op2 - Rn$
0100 - **ADD** : $Rd <= Rn + Op2$
0101 - **ADC** : $Rd <= Rn + Op2 + C$
0110 - **SBC** : $Rd <= Rn - Op2 + C - 1$
0111 - **RSC** : $Rd <= Op2 - Rn + C - 1$
1000 - **TST** : Positionne les *flags* pour $Rn\ AND\ Op2$
1001 - **TEQ** : Positionne les *flags* pour $Rn\ XOR\ Op2$
1010 - **CMP** : Positionne les *flags* pour $Rn - Op2$
1011 - **CMN** : Positionne les *flags* pour $Rn + Op2$
1100 - **ORR** : $Rd <= Rn\ OR\ Op2$
1101 - **MOV** : $Rd <= Op2$
1110 - **BIC** : $Rd <= Rn\ AND\ NOT\ Op2$
1111 - **MVN** : $Rd <= NOT\ Op2$

Figure 9: Opcodes

Les instructions TST, TEQ, CMP et CMN ne font que positionner les flags pour différentes opérations. Par exemple CMP on l'utilise avant un branchement.

Les bits de 11 à 0 indiquent l'opérande 2 celui-ci peut être un immédiat ou un registre et c'est le bit 25 qui indique le type de cet opérande.

Dans le cas où l'opérande 2 est un registre :

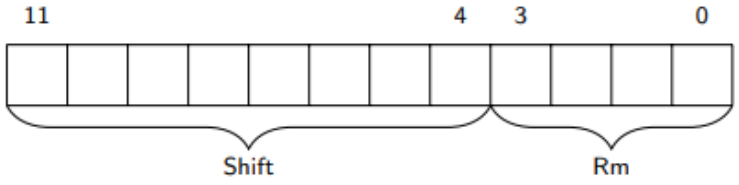

Figure 10: Op2 dans le cas ou c'est un registre

Les 4 bits du poids faible représentent le registre utilisé comme second opérande, et dans les bits restants on a le shift appliqué à ce registre.

Cas où l'opérande 2 est un immédiat :

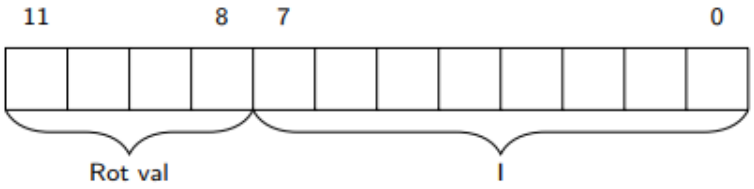

Figure 11: Op2 dans le cas ou c'est un immediat

Les 8 bits du poids faible stockent la valeur de l'immédiat utilisé comme second opérande et les 4 bits du poids fort indiquent le décalage appliqué à cette valeur.

3.2.2 Branchement

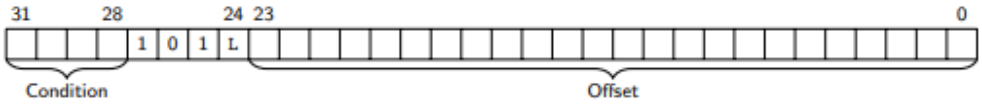

Figure 12: configuration d'une instruction de branchement

Une instruction de branchement est une commande dans le code d'un programme qui détermine la séquence d'instructions à exécuter en fonction de certaines conditions ou de résultats antérieurs dans le programme. Elle permet de modifier le flux d'exécution, par exemple en sautant vers une autre partie du code, en fonction des résultats d'une évaluation conditionnelle.

Si la condition d'exécution n'est pas satisfaite on continue l'exécution du programme principale. Dans le cas inverse on doit arrêter l'incrémentation du PC en mettant **inc_pc** à 0 et ainsi calculer la nouvelle valeur de $\mathrm{PC} = \mathrm{PC} + 8 + \mathrm{offset} \times 4$.

3.2.3 Accès mémoires simples

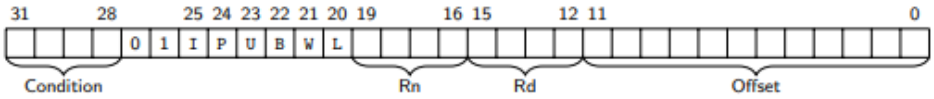

Figure 13: acces memoires simples

Les instructions de transfert simples autorisent l'écriture ou la lecture dans des registres. Lorsqu'il s'agit d'une opération d'écriture, nous nous assurons de désactiver l'invalidation du registre Rd. Étant donné que ce registre est uniquement lu il n'est pas nécessaire de le rendre invalide.

Condition : L'instruction n'est exécutée qui si la condition sur les *flags* est satisfaite ;
I : L'*Offset* correspond à un immédiat si égal 0 ;
P : Pré/Post indexation (Pré si 1) ;
U : *Up/Down* ajout de l'*Offset* si égal 1 ;
B : *Byte/Word* octet si égal 1 ;
W : *Write-back* modification adresse de base si égal 1 ;
L : *Load/Store* lecture mémoire si égal 1 ;
Rn : Registre de base (adresse) ;
Rd : Registre source (écriture) ou destination ;
Offset : Immédiat ou registre combiné au registre de base pour constituer l'adresse.

Figure 14: signification des bits en cas d'accès mémoires simples

3.2.4 Accès mémoires multiples

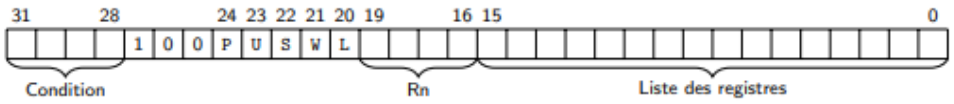

Figure 15: acces memoires multiples

Condition : L'instruction n'est exécutée qui si la condition sur les *flags* est satisfaite ;
P : Pré/Post indexation (Pré si 1) ;
U : *Up/Down* ajout de l'*Offset* si égal 1 ;
S : Voir spécification détaillée ;
W : *Write-back* modification adresse de base si égal 1 ;
L : *Load/Store* lecture mémoire si égal 1 ;
Rn : Registre de base (adresse) ;
Liste : Liste des registres source / destination.

Figure 16: significations des bits des instructions accès mémoires multiples.

Un accès mémoire multiple fait référence à la possibilité d'effectuer simultanément des opérations de lecture ou d'écriture sur plusieurs emplacements mémoire.

3.3 Machine à états

Afin d'assurer le bon fonctionnement de l'étage decod, l'utilisation d'une machine à états est nécessaire. Dans ce contexte, il s'agit d'une machine de Mealy.

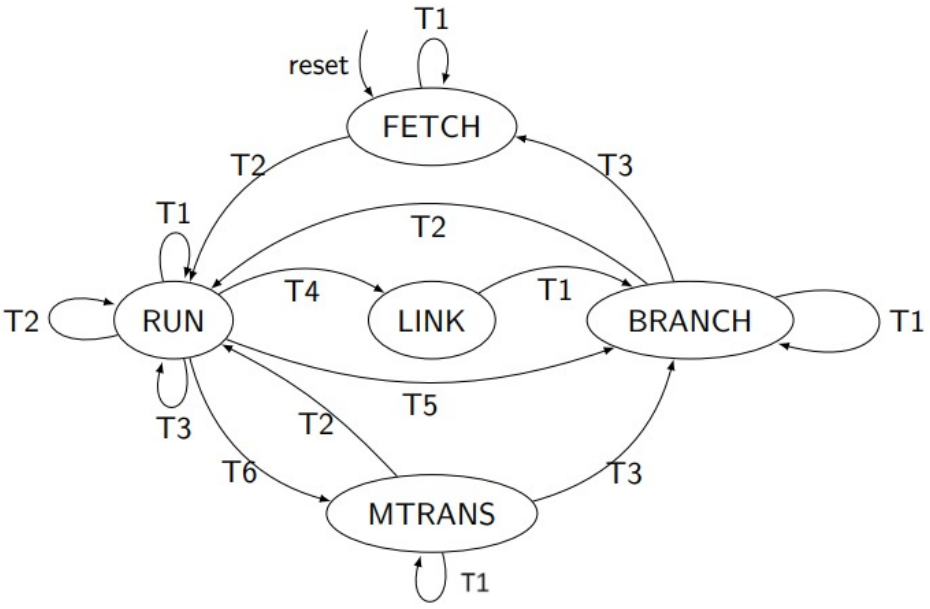

Figure 17: machine a états.

Les changements d'état, qui marquent la transition d'un état à un autre, sont répertoriés ci-dessous.

Étape	**Condition**
T1_FETCH	Si la fifo (decode to ifetch) est vide.
T2_FETCH	Si la fifo (decode to ifetch) est pleine.
T1_RUN	Si la condition d'exécution n'est pas vérifiée.
T2_RUN	Si la fifo (ifetch to decode) est vide.
T3_RUN	Si la fifo (decode to exec) est pleine.
T4_RUN	Détection d'un branch and link.
T5_RUN	Détection d'un branchement.
T6_RUN	Détection d'un transfert multiple.

4 Tests

Nous avons effectué des vérifications à pour chaque étage du pipeline du processeur une fois cet étage réalisé. Pour ce faire, nous avons écrit à des tests bench individuels à chaque étage après sa conception afin de garantir son bon fonctionnement. Ces tests comprenaient des asserts, où des nombres aléatoires étaient introduits dans les entrées de nos entités, et nous vérifiions ensuite la cohérence des sorties.

Par exemple, dans le cadre du test de l'unité arithmétique et logique (ALU), nous avons injecté deux opérandes aléatoires (op1 et op2) accompagnés d'une commande aléatoire (choisie parmi les valeurs 00 à 11). Par la suite, nous avons procédé à une vérification approfondie pour garantir le bon fonctionnement global de l'ALU.

Un autre cas de test concerne l'évaluation du banc de registres (REG). Nous avons injecté deux signaux de 32 bits générés de manière aléatoire en entrée (wdata) pour les inscrire dans des registres sélectionnés de manière aléatoire (wadr). Ensuite, nous avons minutieusement vérifié le bon fonctionnement de cette entité en comparant les valeurs lues dans ces registres avec celles qui avaient été inscrites lors du cycle précédent. Nous avons également assuré la conformité aux spécifications en veillant à ce que, lorsqu'à un instant donné, les canaux wdata1 et wdata2 tentaient simultanément d'écrire dans le même registre wadr, le canal 1 bénéficiait de la priorité, en tant que provenant de l'étage exe. Une autre aspect examiné était la détection des instructions de branchement. En inscrivant une valeur dans le registre 15 (correspondant au registre pc), dès qu'une condition de branchement était détectée, le programme effectuait un saut à l'adresse spécifiée tout en incrémentant de 4 à chaque cycle.

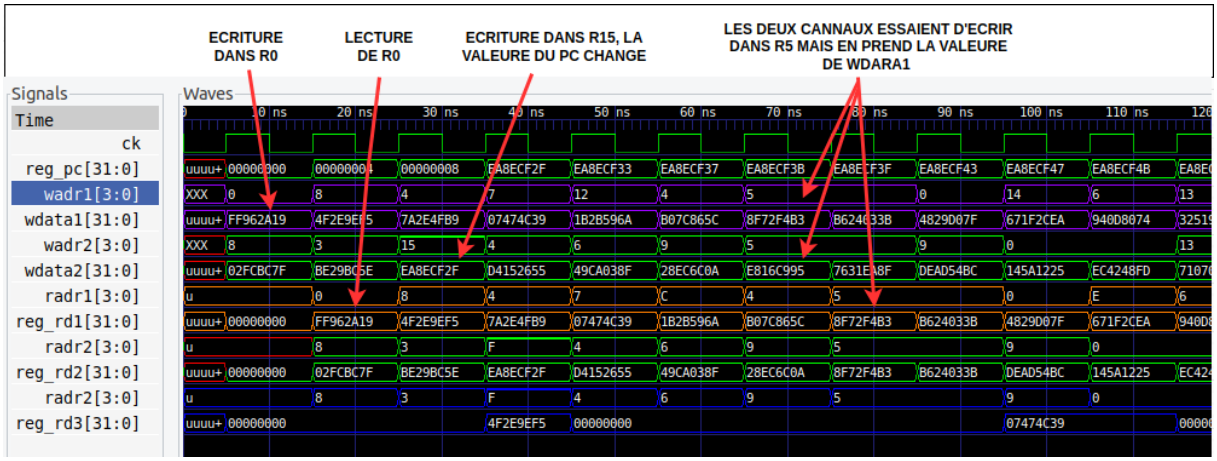

Figure 18: Test de reg sur gtkwave

Cependant, une fois que tous les étages ont été réalisés, il a été nécessaire de les tester conjointement pour vérifier qu'ils communiquaient correctement entre eux, et surtout que l'unité de décodage (DECOD) accomplissait correctement sa tâche.

4.1 Simulation complète de CPU

Après avoir achevé la conception de l'unité DECOD, nous avons intégré l'ensemble de nos fichiers dans le cœur du processeur. Toutefois, étant donné que notre processeur n'a pas un accès physique à la mémoire, nous avons dû recourir à une simulation en utilisant une interface vers du langage C. L'idée principale était de permettre à notre suite de tests VHDL d'écrire et de lire en mémoire en utilisant des fonctions externes définies en langage C.

Ensuite, nous avons simulé l'intégralité de ce processeur en utilisant des tests écrits en langage assembleur ARM. Ces tests ont été compilés à l'aide d'un petit simulateur d'instructions ARM qui a été présenté lors des premiers TMEs.

Afin de valider le bon fonctionnement intégral du processeur, nous avons réalisé des tests avec un code en assembleur ARM, incluant des instructions de calcul telles que l'ADD. En analysant la sortie sur GTKWave, lors du premier cycle, l'étage IFETCH récupère l'instruction encodée en HEXA. La valeur du compteur de programme (PC) est initialisée à 0 durant ce cycle. Au cycle suivant, l'étage DECODE prépare les opérandes nécessaires (opérandes 1 et 2, ainsi que le registre de destination) et les commandes pour la transition vers l'étage EXE (la commande pour l'ALU indiquant une opération d'addition, sans besoin du complément à 2 ni du shift pour nos opérandes). Dans le cycle suivant, la nouvelle valeur calculée par l'étage EXE est correctement écrite dans le registre de destination du banc de registres.

Ci-dessous, vous trouverez un extrait de code écrit en langage assembleur ARM, accompagné des résultats visibles sur GTKWave.

```
00000000 <_start>:
   0:    e3a04018    mov    r4, #24
   4:    e3a0100f    mov    r1, #15
   8:    e2044002    and    r4, r4, #2
   c:    e3a0200a    mov    r2, #10
  10:    e3811005    orr    r1, r1, #5
  14:    e2422003    sub    r2, r2, #3
  18:    e1a00000    nop
  1c:    e1a00000    nop
```

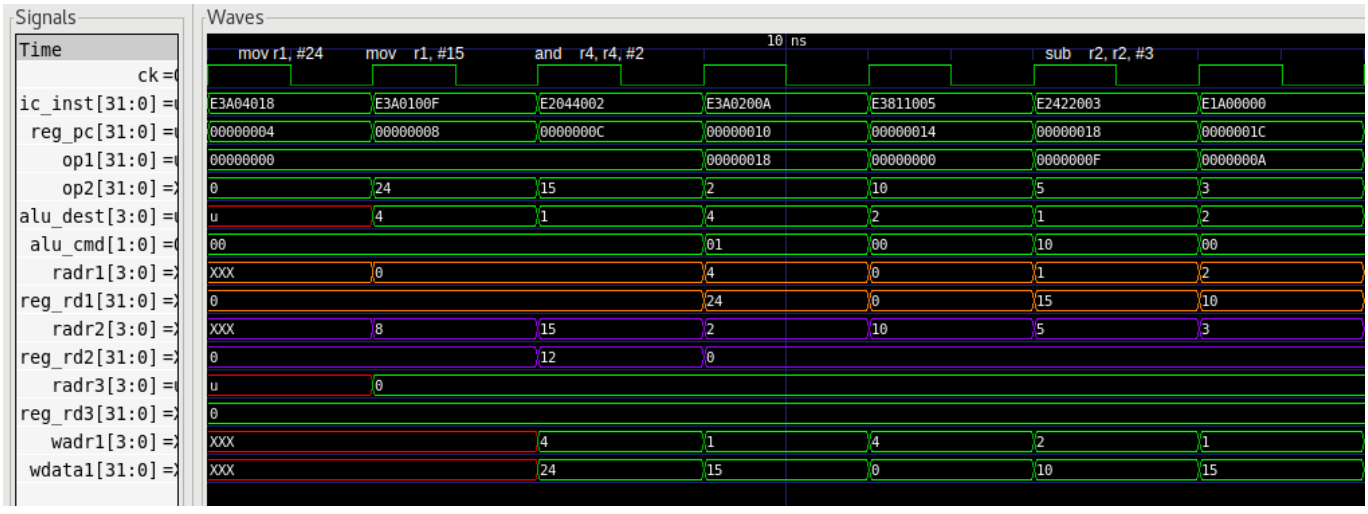

Figure 19: résultat de la simulation du code assembleur sur gtkwave

5 Conclusion

Ce projet nous a permis de comprendre le fonctionnement d'un processeur en pipeline, d'appréhender le cheminement des instructions à travers les différents stades du pipeline, et d'analyser de manière approfondie le travail accompli par chaque étage de ce pipeline.

Au final, nous sommes très fiers des résultats obtenus. Notre processeur est capable d'exécuter des instructions de traitement de données. Cependant, il n'a pas encore réussi à effectuer des branchements, ce qui empêche actuellement notre processeur d'être autotestant avec une approche où, en cas de succès ou d'échec du résultat calculé, le programme sera automatiquement dirigé vers une adresse de branchement représentant une étiquette « good » ou « bad ». Nous sommes conscients que la source du problème réside dans l'invalidation du registre de PC lors d'un branchement, et nous prévoyons de le résoudre prochainement. Nous envisageons également d'ajouter des bypass pour éliminer certains cycles de gel en cas de dépendance de données dans le programme