Sorbonne université - sciences et ingénierie


Master Systèmes électroniques
Systèmes informatiques (SESI)

UE : VLSI

**Rapport de projet**

---

# Création et simulation d'un processeur Arm

---

Camelia **BOURAS**

Owen **LIU**

Année universitaire 2023/2024

# Table des matières

# Introduction

Lors de l'UE VLSI nous avons étudié l'architecture et le jeu d'instruction du processeur ARM afin de nous préparer au projet de l'UE. Ce projet consiste à réaliser un processeur 32 bits à pipeline 4 étages.

Ce dernier est réalisé en langage VHDL, compilé et simulé grâce à GHDL et finalement visualisé sur GTKWave. La synthétisation se fait à l'aide Yosys.

Les consignes du projets se résume à décrire que les étages DECODE et EXEC du pipeline, tandis que les étages IFETCH et MEM sont déjà fournis ainsi que l'instanciation de tout les étages réalisés dans le fichier ARM-CORE.

Dans ce rapport nous allons retracer les semaines travails et l'avancement sur nôtre processeur ainsi que les problèmes rencontrés le long du projet.

En premier lieu nous allons parler de l'architecture générale du processeur en soulignant les objectifs primordiaux. Juste après nous enchaînerons sur l'étage EXEC puis l'étage DECODE. Finalement nous conclurons avec les résultats obtenus et les problèmes rencontrés.

# Chapitre I : Le processeur

## 1) Présentation du processeur

Nous avons fait le choix d'une architecture à 4 étages :

- FETCH : Recherche l'instruction assembleur à l'adresse indiqué par PC dans I-CACHE et nous la renvoie.
- DECOD : Décode l'instruction assembleur envoyé par FETCH, et gère le fonctionnement du processeur en fonction de cette dernière.
- EXE : Exécute les opérations demandé par DECOD, peut renvoyer des données vers DECOD et peut envoyer des informations à MEM
- MEM : Si l'instruction nécessite un accès mémoire, soit enregistre une valeur dans D-CACHE ou lit dasn D-CACHE et renvoie la valeur vers DECOD

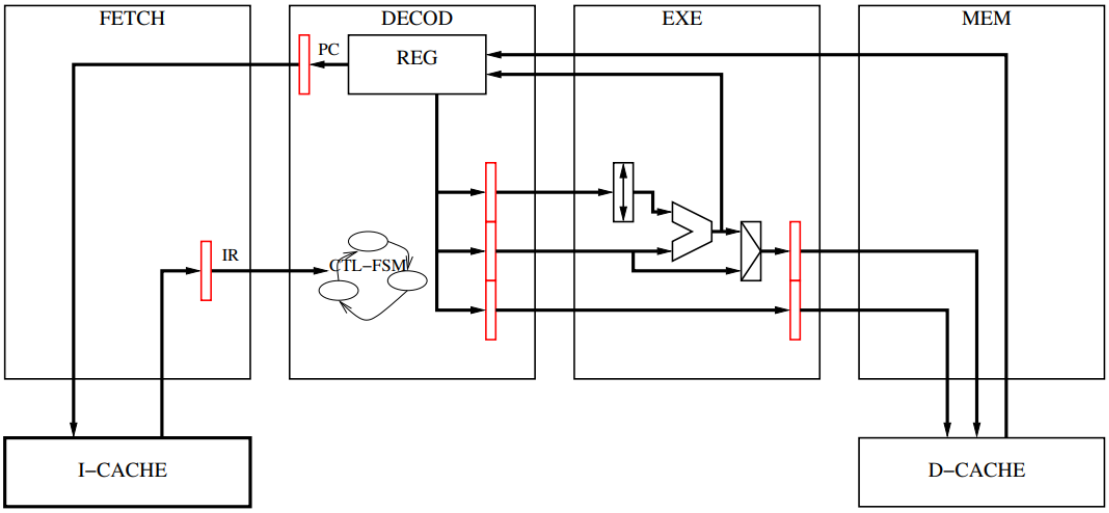


FIGURE 1 – Schéma général du processeur

## 2) Objectifs du projet

Nous avons à réaliser les étages DECOD et EXEC mais aussi de gérer la synchronisation entre les 4 étages du pipeline grace au fifos (exe2mem, dec2exe, dec2fetch, fetch2dec) et des bypasses.

Nous avons eu pour objectifs primordiaux :

- Exécuter les instructions (Prioritaire)
- Garantir (quand cele est possible) le traitement d'une instruction par cycle d'horloge
- Limiter le matériel nécessaire
- Supporter des interfaces mémoire (I-Cache et D-cache) qui ne répondent pas nécessairement dans le cycle.
- Minimiser le temps de cycle.

# Chapitre II : Étage EXEC

Une fois qu'une instruction est décodée, l'étage EXEC calcul le résultat demandé par DECOD, ces calculs peuvent être des opérations arthimétiques ou logique simples, des calculs d'adresse de branchement ou d'emplacement mémoire.

Nous avons alors dans cet étage du pipeline trois blocs essentiels :

- L'ALU, effectue différents types de calculs en fonction de la commande.
- Le Shifter, responsables du décalage de l'opérande 2.
- La Fifo, responsable de la sychronisation entre l'étage EXEC et MEM.

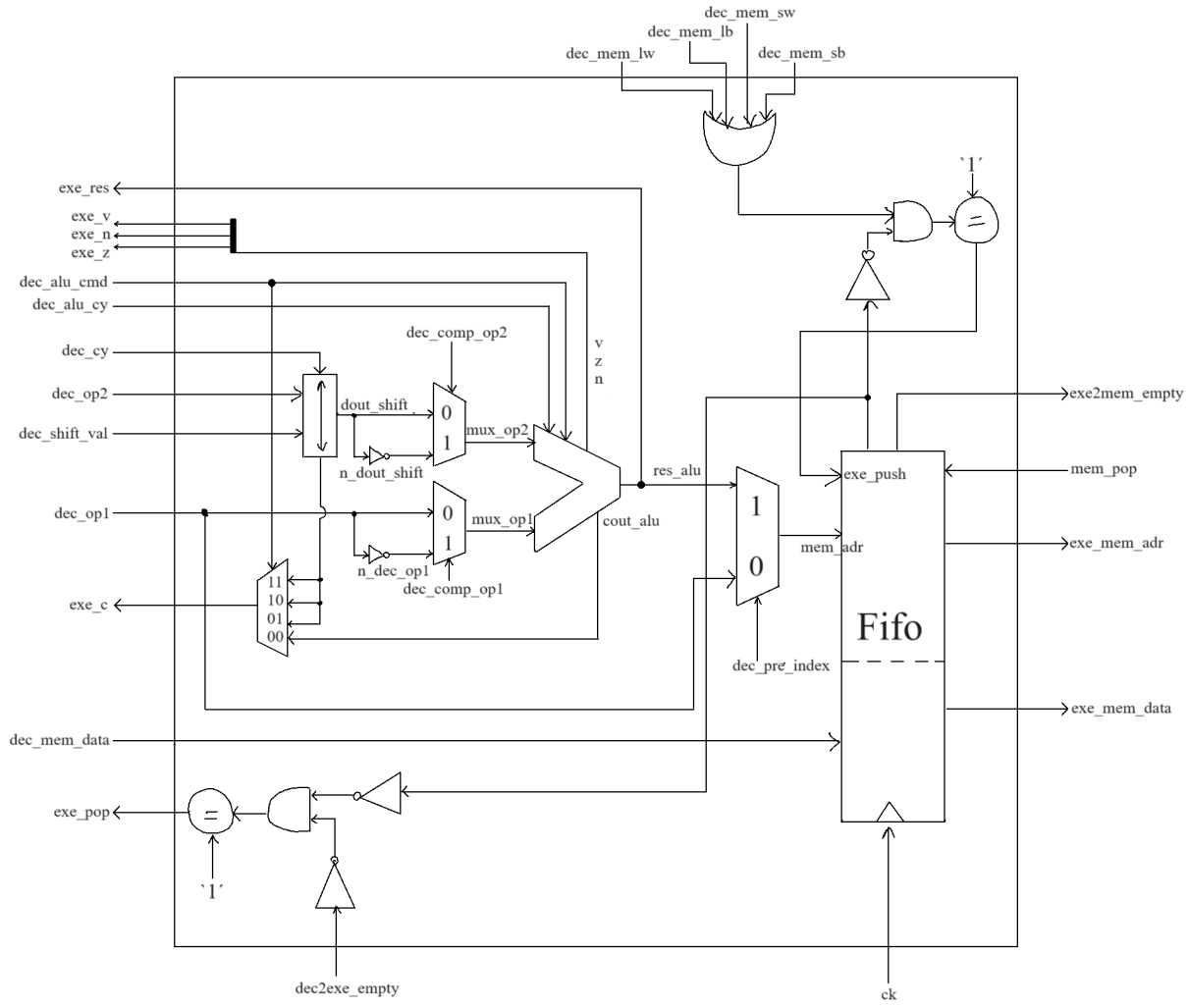


FIGURE 2 – Schéma de l'étage EXEC

# 1) Unité Arithmétique et Logique

## a) Description

L'ALU (Unité Arithmétique et Logique) est un composant essentiel de tout processeur. Dans le notre elle réalise les opérations arithmétiques et logiques comme l'addition, la soustraction, AND, OR et XOR. Elle nous donne aussi des informations sur les résultat grâce au flags Cout, Z, N et V. Grâce à toutes ces options nous répondons à la plupart des demandes de DECOD, il ne manque plus qu'un dernier module, le SHIFTER.

## b) Implémentation

Nous avons réaliser ce module entière de manière combinatoire et est synthétisable.

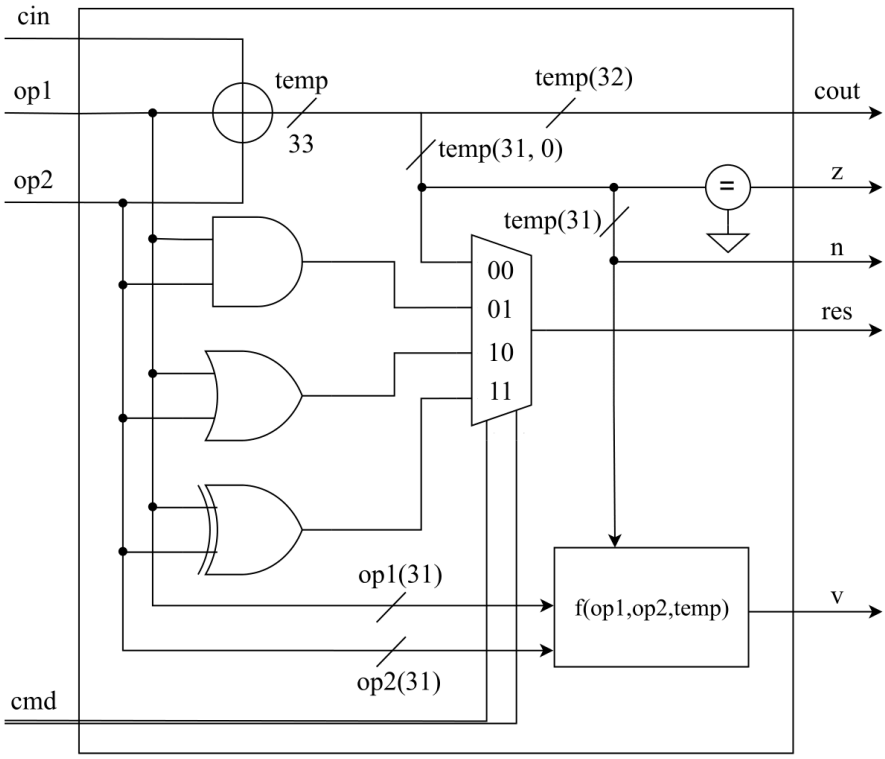


FIGURE 3 – Schéma de l'ALU

# 2) Le Shifter

## a) Description

Cette unité est spécifiquement dédiée aux opérations de décalage et de rotation sur un vecteur de bit. Le shifter est utilisé ici pour effectuer des manipulations sur l'opérande 2 avant qu'elle passe à travers l'ALU. Grâce à l'association de l'ALU et du SHIFTER nous résolvons tout les demandes de DECOD.

## b) Implémentation

Ce module est aussi entièrement combinatoire. Le Shifter est synthétisable grâce au fonction shift du package `numeric_std`.

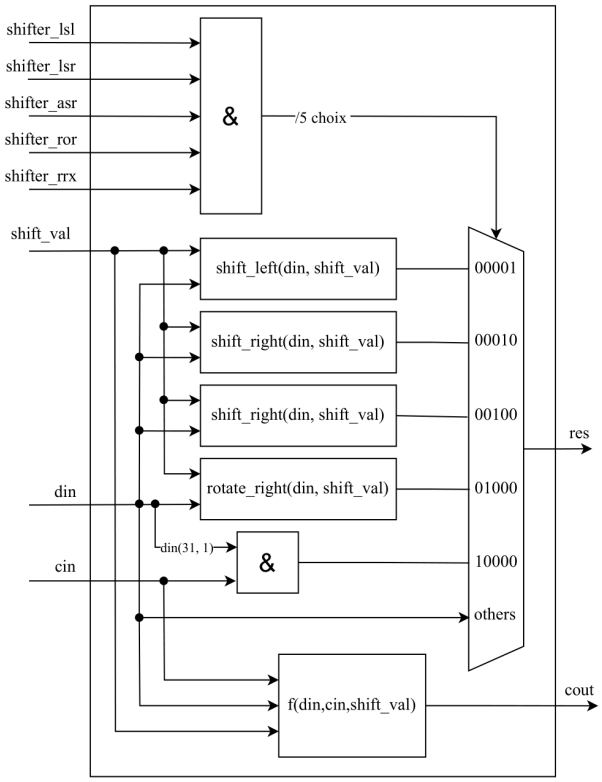


FIGURE 4 – Schéma du Shifter

# Chapitre III : Étage DECOD

DECOD communique avec ses deux voisins FETCH et EXEC, suivant cette unique séquence qui pourra être interrompue en fonction de l'instruction à exécuter :

- Envoie de l'adresse d'une instruction (PC) dans la fifo `dec2if`
- Lire et décoder l'instruction envoyé par FECTH
- Écrire le résultat du décodage dans la fifo `dec2exec`

Cette étage comporte trois blocs importantes :

- Le décodages de l'instruction
- Le banc de registre (REG)
- La machine à état (MAE)

Voici un shéma de comment intéragisse ses différents blocs :

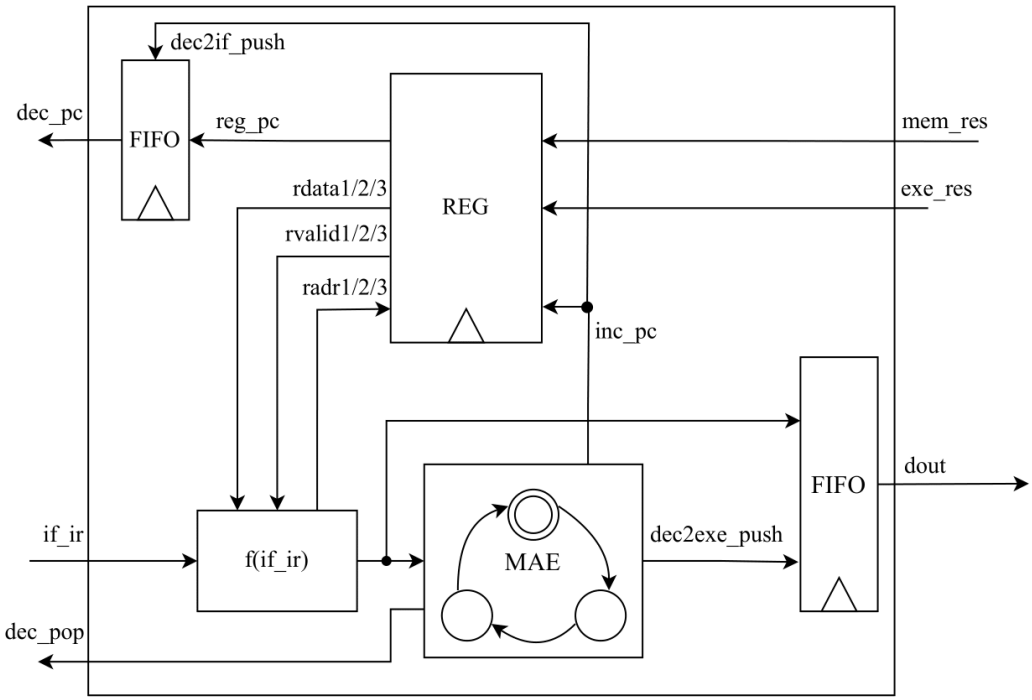


FIGURE 5 – Schéma de DECOD

## 1) Le décodage

Le décodage des instructions est faite de manière combinatoire :

- Lire et décoder l'instruction envoyé par FECTH
- Écrire le résultat du décodage dans la fifo `dec2exec`

DECOD doit interpréter l'OPCODE et déterminer l'opération à réaliser. Si une instruction demande à lire des registres, DECOD doit extraire les informations nécessaires par le biais du registre.

La dernière étape de DECOD (dec2exe_push) peut ne parfois pas être executée. Cela arrive quand l'instruction ne vérifie pas ses conditions d'éxecution.

## 2) Le banc de Registre (REG)

Notre processeur contiens au total 16 registres plus le registre CSPR pour les flags, dont R14 reservé à l'adresse de retour en cas de branch and link et R15 pour PC.

Ce banc de registre est responsable de l'identification des registres voulant être lu par une instruction, gérer les écriture et l'autorisations d'écriture.

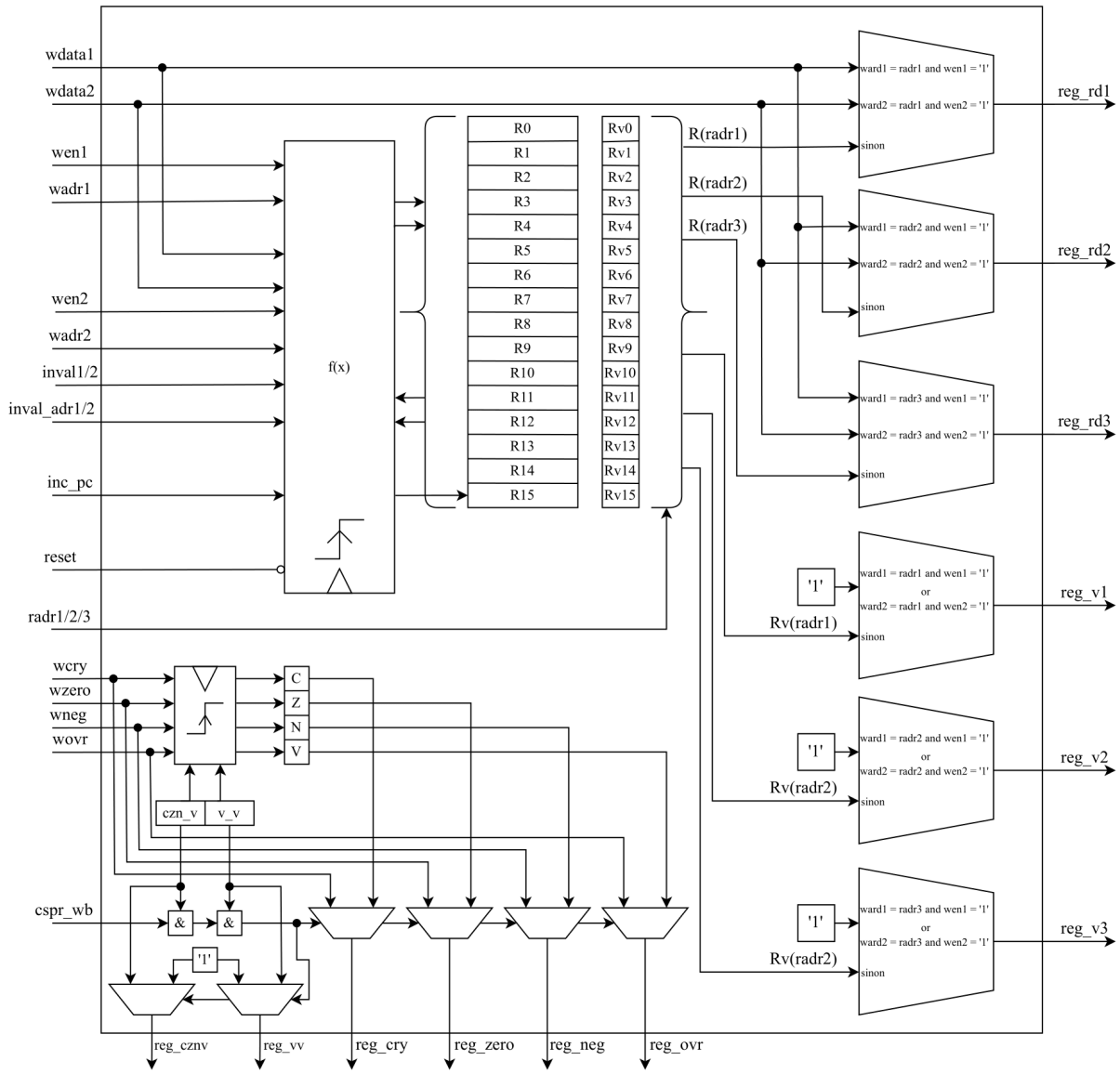


FIGURE 6 – Schéma de REG

## 3) La machine à état (MAE)

Dans DECOD on retrouve la machine à état. Sa fonction principale consiste à contrôler la lecture et l'écriture dans les trois fifos. Trois signaux vont être dédiés à ces actions :

- `dec2if_push` : contrôle l'envoie d'un adresse de PC à FETCH
- `if2dec_pop` : contrôle si l'on vide l'instruction envoyé par FETCH
- `dec2exe_push` : contrôle l'envoie d'un cacul à EXEC

Aussi en fonction de l'instruction décodé, la MAE devra se comporter différemment :

- FETCH :
  - T1 : Nous n'avons pas reçus d'adresse par DECOD
  - T2 : Nous avons une adresse et nous avons une instruction valide, passage à RUN
- RUN :
  - T1 : Transition de gel, nous ne faisons rien durant 1 cycle
  - T2 : L'instruction n'est pas valide on n'envoie pas d'envoie vers EXEC
  - T3 : L'instruction est valide on l'envoie vers EXEC
  - T4 : Nous avons un branchement simple
  - T5 : Nous avons un branch and link
- LINK : Nous allons vers BRANCH
- BRANCH :
  - T1 : Vidage de la fifo `if2dec`
  - T2 : `if2dec` vide, nous revenons vers RUN

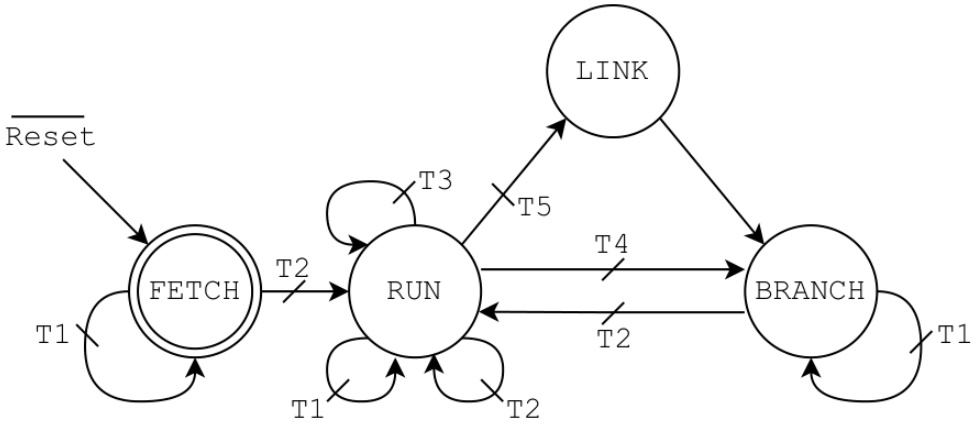


FIGURE 7 – Machine à états de DECOD

# Chapitre IV : Simulations et résultats

## 1) Simulation

### 1) Fonctionnement du pipeline

```
1  // pipeline //
2      .text
3      .globl   _start
4  _start: ldr  r0, data0 // r0 <= 0x12345678
5          mov  r0, r0
6          b    _good
7          b    _bad
8  _bad :   nop
9          nop
10 _good :  nop
11         nop
12 data0 :   .word 0x12345678
```

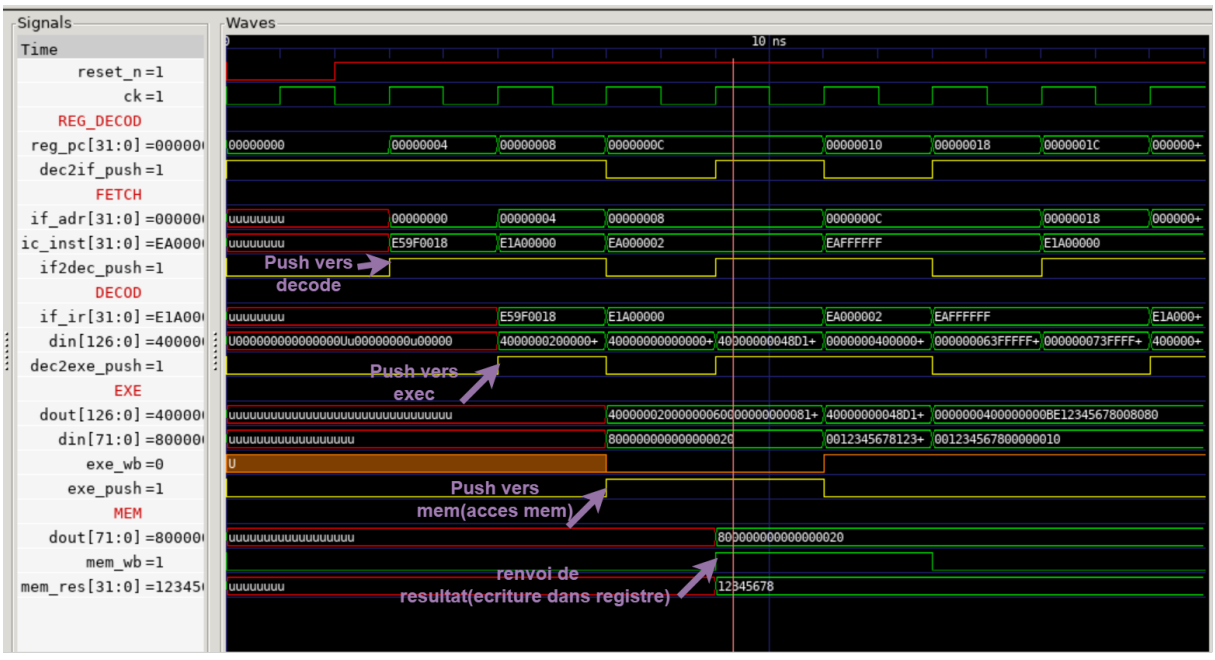


FIGURE 8 – Visualisation sur gtkwave du code `pipeline`

Dans la figure ci-dessus on retrouve la simulation de ce code assembleur, on remarque que :

- Après le reset asynchrone, on attend le premier front montant pour push l'adresse de la première instruction à FECTH, il renvoie l'instruction et est push vers DECOD.
- Elle est décodée et envoyée vers EXEC et ce dernier la push vers MEM vu qu'il s'agit d'un accés mémoire (ldr) .
- Enfin on récupère bien la bonne data qu'on renvoi dans le registre R0.

### 2) Opérations arithmétiques et flags

```
1  // Add_test //
2          .text
3          .globl   _start
4  _start: mov r0, #1000
5          mov r1, #2000
6          add r2, r0, r1 // 3000
7          sub r3, r1, r0 // 1000
8          sub r4, r0, r1 //-1000
9          mov r2, r4     //-1000
10         add r5, r3, r4 // 0
11         b _good
12         b _bad
13 _bad :  nop
14         nop
15 _good : nop
16         nop
```

On remarque bien que l'étage DECOD a bien décodé les instructions arithmétiques comme illustré dans la FIGURE 9. De plus on observe bien un cycle par instruction malgré les dépendances de données.

Dans la FIGURE 10 on peut bien visualiser les flags qui passent à 1 selon les résultat obtenu.

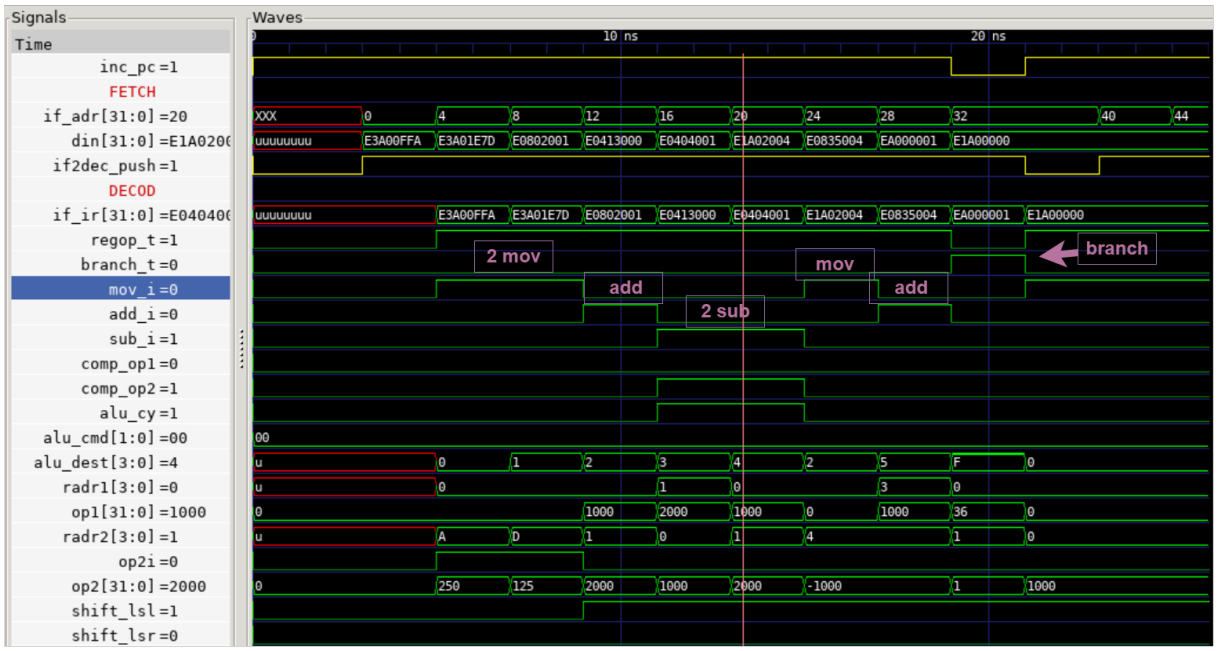


FIGURE 9 – Simulation gtkwave du code `add_test` p1

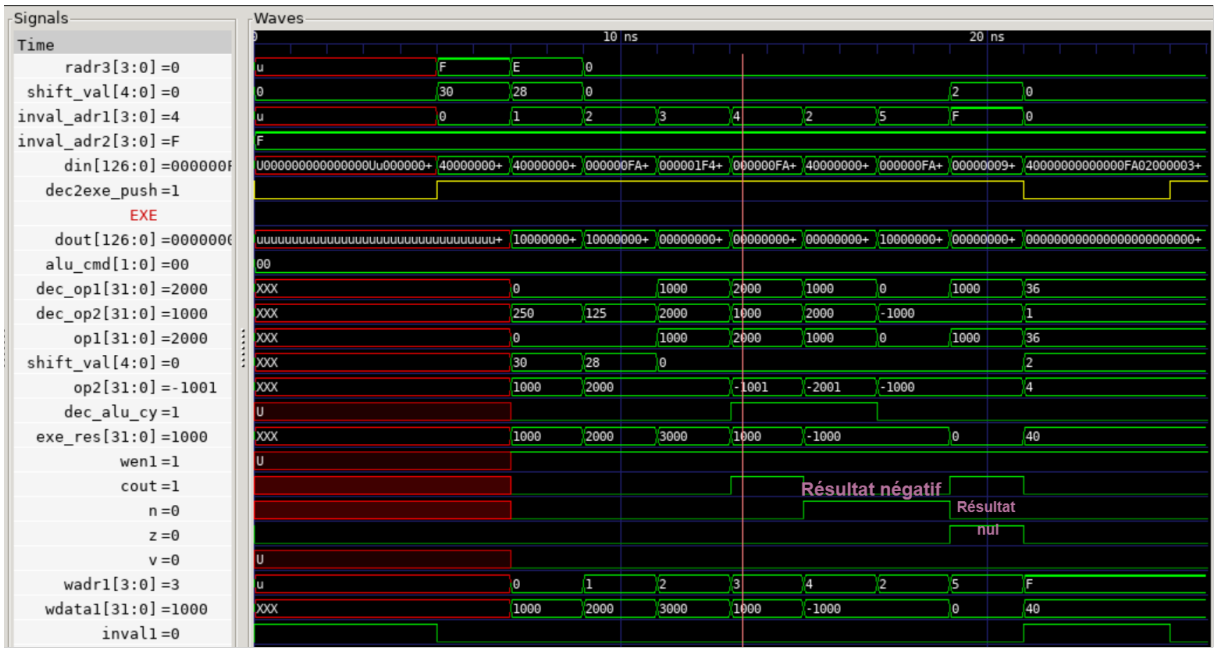


FIGURE 10 – Simulation gtkwave du code `add_test` p2

### 3) Instructions avec conditions

```
1  // cond_test //
2      .text
3      .globl  _start
4  _start: movs r0, #100 //Mise à jour flag Z = 0 N = 0 C = 0
5          mov  r1, #205
6          addeq r2, r0, r1 // 305 si Z = 1 NON
7          eorne r3, r1, r1 // 0 si Z = 0 OUI
8          addcs r1, r0, r0 //200 si C = 1 NON
9          rsbcc r4, r0, r1 //105 si C = 0 OUI
10         sbcpl r5, r1, r0 //205-100+C-1 = 104 si N = 0 OUI
11         subs  r6, r0, r1 //100-205 = -105 maj flag Z = 0 N = 1 C = 0 V = 0
12         rscmi r7, r1, r0 //100-205+C-1 = -106 si N = 1 OUI
13         mov r8, #0
14         mov r9, #1
15         b _good
16         nop
17         b   _bad
18 _bad :  nop
19         nop
20 _good : nop
21         nop
```

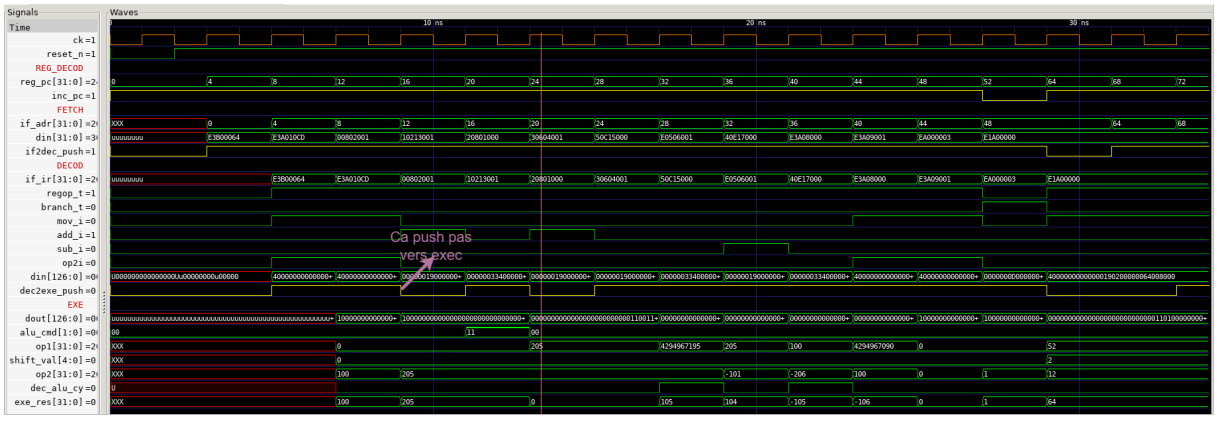


FIGURE 11 – Simulation gtkwave du code `cond_test`

### 4) Branchement

```
1  // branch_test2 //
2      .text
3      .globl  _start
4  _start:    b startup          //PC = 0
5             b _bad             //PC = 4
6  startup :  mov  r0, #1        //PC = 8
7             mov  r1, #1        //PC = 12
8             adds r2, r1, r0    //PC = 16
9             bne  _good         //PC = 20
10 _bad :     nop                //PC = 24
11            nop                //PC = 28
12 _good :    nop                //PC = 32
13            nop                //PC = 36
14 AdrStack:   .word 0x80000000
```

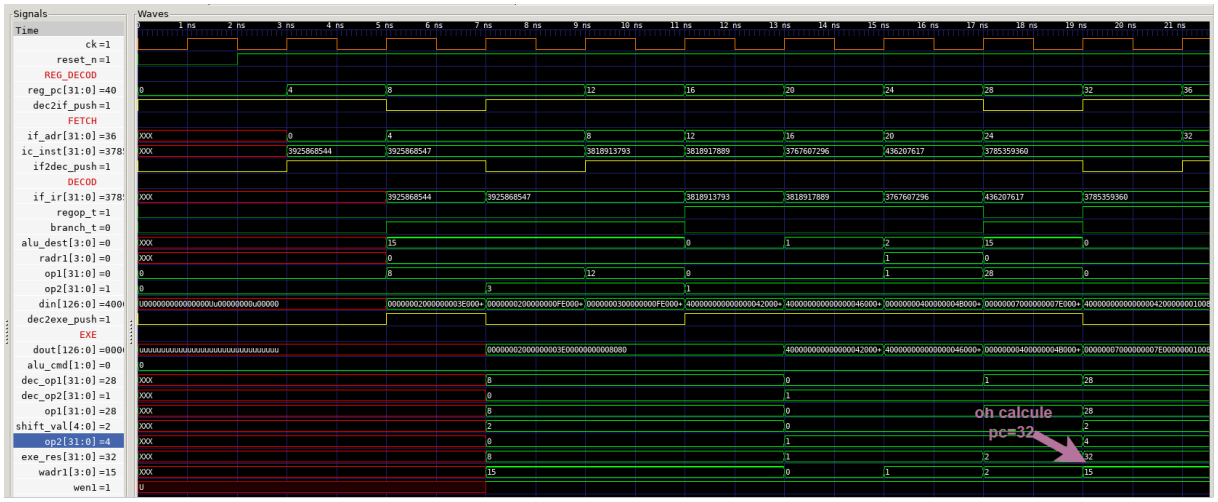


FIGURE 12 – Simulation gtkwave du code `branch_test2`

### 5) Acces mémoire

```
1  // load_store_test //
2      .text
3      .globl  _start
4  _start:  b    startup
5           b  _bad
6  startup: ldr  r0, data0
7           ldr  r1, data1
8           add  r2, r0, r1
9           sub  r3, r1, r1
10          str  r2, data2
11          str  r3, data3
12          ldr  r4, data2
13          ldr  r5, data3
14          mov  r4, r4
15          movs r5, r5
16          beq  _good
17 _bad :   nop
18          nop
19 _good :  nop
20          nop
21 data0    :   .word 0x00000010
22 data1    :   .word 0x12345678
23 data2    :   .word 0x00000000
24 data3    :   .word 0x14385012
25 AdrStack:   .word 0x80000000
```

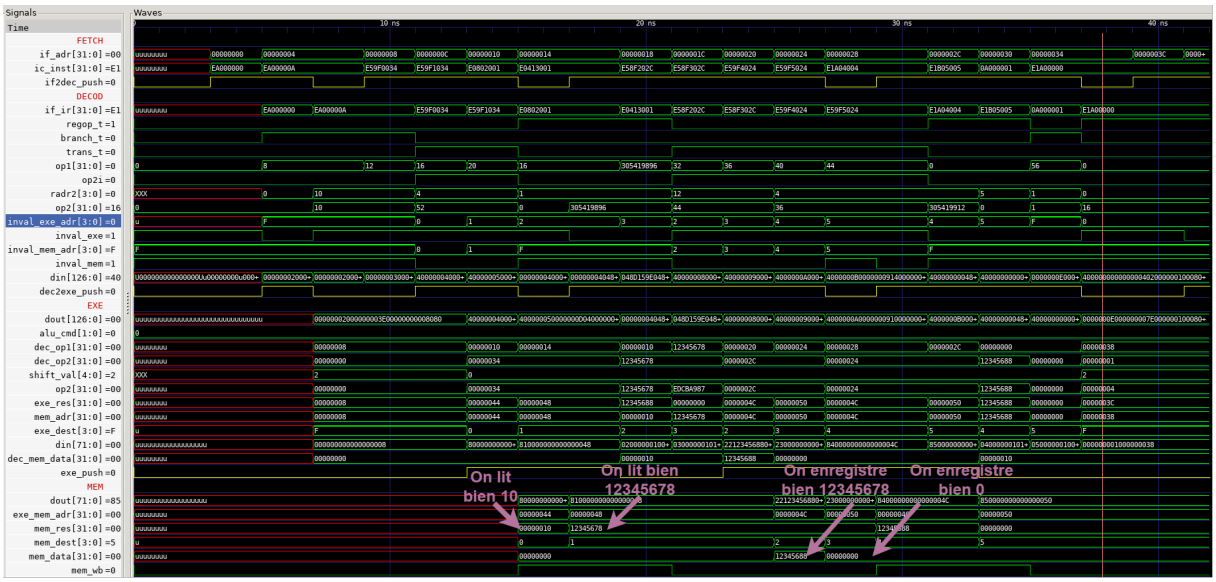


FIGURE 13 – Simulation gtkwave du code load test

### 6) Fonction complexe

Ce code englobe la majorité des fonctions que notre processeur peut exécuter. Ce dernier réalise une multiplication de data1 = 153206 et de data2 = 15 :

```
1  // test final //
2      .text
3      .globl  _start
4
5  _start: b    startup
6          b    _bad
7  startup: ldr r0, data1
8           ldr r1, data2
9           bl  mutl
10          str r2, res
11          ldr r3, res
12          b _good
13 mutl :   subs  r3, r0, r1
14          movpl r2, r0
15          movpl r0, r1
16          movpl r1, r2
```

```
17              mov     r2, #0
18  _loop :     add r2, r2, r1
19              subs r0, r0, #1
20              bne _loop
21              bx  r14
22              add r2, r2, #1000
23  _bad :
24      nop
25      nop
26  _good :
27      nop
28      nop
29  data1:  .word 153206
30  data2:  .word 15
31  res   : .word 0
```

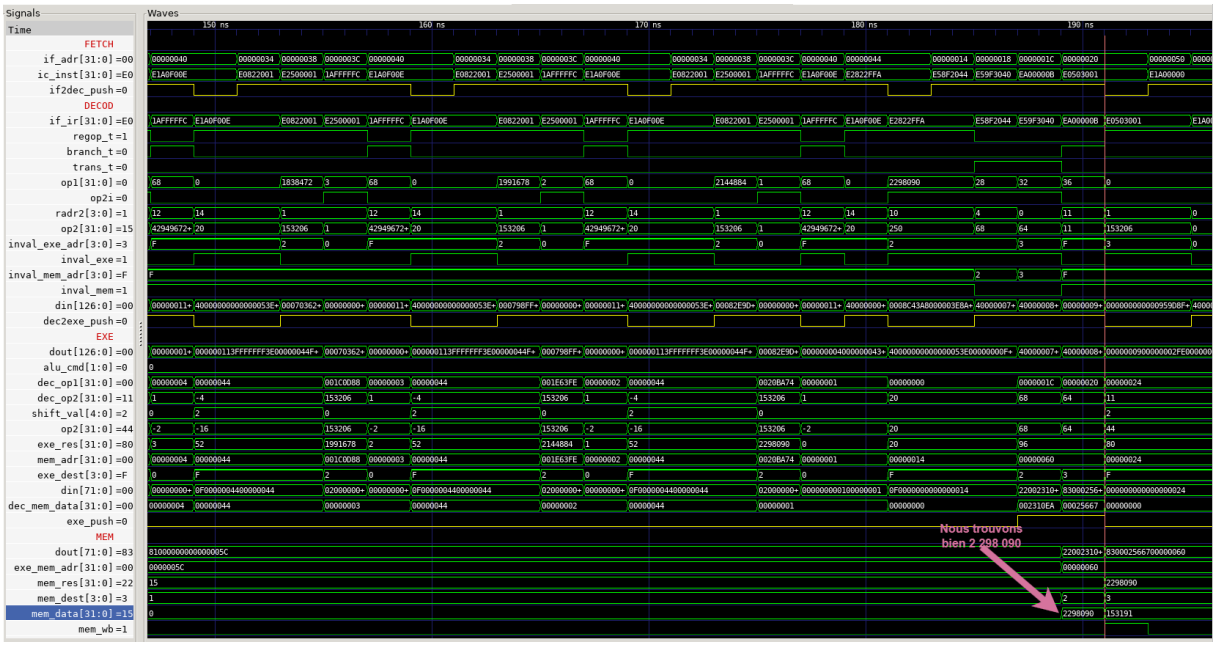


FIGURE 14 – Simulation gtkwave du code de multiplication

## 2) Problèmes rencontrés

Durant le long de ce projet, nous avons rencontré des problèmes divers et variés que nous avons pu surmonter finalement grâce aux instructions de notre professeur et à nos recherches.

Parmis les obstacles que nous avons croisé :

- Les bypass et leurs gestions afin de minimiser les cycles par instruction surtout quand il s'agit de dépandance de données entre branchement et mémoire.
- La gestion des branchements, pour éviter les delayed slot
- Le casting ou la conversion explicite d'un type de données en VHDL.

# Conclusion

Pour conclure, la conception et réalisation d'un processeur ARM en utilisant le langage VHDL ont été une tâche intéressante et enrichissante. Au cours de ce projet, nous avons réussi à atteindre plusieurs objectifs cruciaux comme le bon décodage et l'execution d'instructions en assembleur ainsi que la réduction du nombre de cycles par instruction.

Notre processeur, à la fin de cette conception, parvient avec succès à décoder presque toutes les instructions du jeu d'instructions d'un processeur ARM, à l'exception de quelques instructions complexes telles que la multiplication et les transferts multiples. Ces limites sont dues au fait que nous avions peu de temps. En conclusion, ce projet a été une expérience éducative et valorisante, offrant des aperçus approfondis dans la conception de processeurs et le langage VHDL. Les compétences acquises tout au long de ce processus seront sans aucun doute les plus précieuses dans notre parcours scolaire et professionnel. Ainsi notre engagement continu dans le vaste domaine des systèmes embarqués.