SORBONNE UNIVERSITÉ

Projet VLSI

Jorge Alberto MENDIETA OROZCO
Lou THIRION

Master Informatique
Parcours Systèmes Électroniques et Systèmes Informatiques
M1S1

Professeur : **Jean-Lou Desbarbieux**

Table des matières

1 Introduction

Ce projet a pour but l'implémentation d'un processeur ARM v3 en utilisant VHDL comme langage de description de matériel. L'architecture de ce processeur est basée sur un pipeline asynchrone à 4 étages : Instruction Fetch (IFETCH), Décodage (DECOD), Exécution (EXEC), Mémoire (MEM).

Les deux principaux étages sur lesquels nous avons travaillé étaient **l'étage d'exécution (EXEC)** et **l'étage de décodage d'instructions (DECOD)**.

2 Étage d'exécution (EXEC)

Cet étage comporte principalement l'implémentation d'une Unité Arithmétique Logique (*ALU* pour ces sigles en anglais), un Décaleur (*shifter*), quelques Multiplexeurs et un registre FIFO pour stocker le résultat.

2.1 Unité Arithmétique Logique - ALU

L'ALU comporte deux entrées de 32 bits, une sortie de 32 bits, et une commande permettant de choisir l'opération à réaliser. Elle est similaire â un multiplexeur.

Notre ALU comporte 4 opérations au total, nous devons donc utiliser un `std_logic_vector` de deux bits ($2^2 = 4$ valeurs possibles).

Nous avons utilisé une case pour sélectionner l'opération à réaliser.

2.1.1 Additionneur

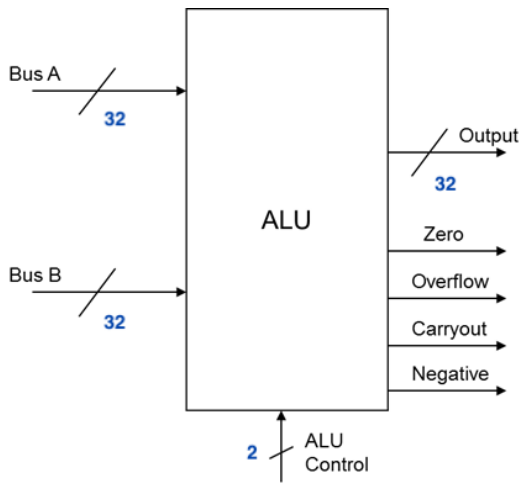

FIGURE 1 – Diagramme de l'ALU utilisé

Nous avons implémenté un additionneur complet (avec retenue) de 32 bits pour l'ALU. À l'aide de la librairie `IEEE.numeric_std` nous pouvons réaliser des additions. Pour cela il fallait d'abord convertir le `std_logic_vector` vers un type `unsigned` pour utiliser l'opération `+` et ainsi réaliser une addition de façon simple sans recourir à la mise en place des portes logiques (XOR, AND, OR) pour chaque bit. Notre implémentation est simple et utilise les outils de synthèse VHDL afin de traduire le code au matériel (FPGA).

Néanmoins pour notre implémentation nous avons concaténé un bit additionnel aux signaux pour faire l'addition et obtenir ainsi un carry out stocké dans le MSB (bit 33). Un exemple est montré ci dessous :

```
temp <= std_logic_vector(unsigned('0' & a) + unsigned('0' & b)
```

```
        + unsigned'('0' & cin));
```

2.1.2 Sélection d'opérations

L'ALU étant exclusivement combinatoire, elle fonctionne en partie comme un multiplexeur grâce à une commande qui fait une sélection de l'opération à réaliser. Notre signal de commande est un `std_logic_vector` de 2 bits, afin d'obtenir $2^2 = 4$ opérations différentes.

- **Addition** : Quand la commande `dec_alu_cmd` est égale à "00", l'ALU utilise l'additionneur complet (avec retenue d'entrée).
- **ET logique** : Quand la commande `dec_alu_cmd` est égale à "01", l'ALU réalise une opération booléenne AND.
- **OU logique** : Quand la commande `dec_alu_cmd` est égale à "10", l'ALU réalise une opération booléenne OR.
- **XOR logique** : Quand la commande `dec_alu_cmd` est égale à "11", l'ALU réalise une opération booléenne XOR (OU exclusif).

La partie logique de l'ALU a été facile à implémenter, puisque les données en entrée sont déjà des `std_logic_vector` on peut simplement utiliser des opérations booléennes (and, or, xor).

2.1.3 Génération des drapeaux

L'ALU génère 4 drapeaux de condition (ou simplement, drapeaux) qui sont activés lors des conditions suivantes [2] :

- **Drapeau Zéro (Z)** : S'active quand le résultat produit est égal à zéro.

```
if (res_s = x"0000_0000") then
  z_f <= '1';
else
  z_f <= '0';
end if;
```

- **Drapeau Négatif (N)** : S'active quand le bit le plus significatif (MSB) est égal à '1' (complément à 2).

```
if (res_s(31) = '1') then
  n_f <= '1';
else
  n_f <= '0';
end if;
```

- **Drapeau débordement (V)** : S'active lors d'un changement de signe par rapport aux entrées.

```
if (((op1(31) = '0') and (op2(31) = '0') and (res_s(31) = '1'))
```

```
  or ((op1(31) = '1') and (op2(31) = '1') and (res_s(31) = '0'))) then
  v_f <= '1';
else
  v_f <= '0';
end if;
```

- **Drapeau Retenue (C)** : S'active dans deux cas, quand une opération arithmétique génère une retenue (ex. addition, soustraction) ainsi qu'avec les opérations de décalage (dernier bit décalé).

Ces drapeaux seront mémorisés par le registre CPSR quand une instruction utilise un writeback.

2.1.4 Drapeau Débordement

Ce drapeau est plus complexe à calculer par rapport aux autres car il faut d'abord bien comprendre dans quel cas il y a débordement :

Il existe deux règles d'activation du drapeau de débordement en mathématiques binaires/entiers :

1. Si la somme de deux nombres, dont les bits de signe sont à 0, donne un résultat dont le bit de signe est à 1, le drapeau de débordement est activé.

   0100 + 0100 = 1000 (le drapeau de débordement est activé)

2. Si la somme de deux nombres, dont les bits de signe sont à 1, donne un résultat dont le bit de signe est à 0 alors le drapeau de débordement est activé. Avec le bit de signe à 0, le drapeau de débordement est activé.

   1000 + 1000 = 0000 (le drapeau de débordement est activé)

Dans le cas contraire, le drapeau de débordement est désactivé.

Il suffit de regarder les bits de signe (MSB) des trois nombres (opérande 1, opérande 2, résultat) pour décider si le drapeau de débordement est activé ou désactivé.

Si l'arithmétique en complément à deux (**signée**) est utilisée, alors si le drapeau de débordement est activé cela signifie que la réponse est **erronée** ; deux nombres positifs ont été ajoutés et un nombre négatif a été obtenu, ou vice-versa, deux nombres négatifs ont été ajoutés et un positif a été obtenu.

Si on fait des opérations arithmétiques **non signées**, le drapeau de débordement ne signifie rien et doit être ignoré.

Méthode du calcul du drapeau de débordement Le débordement ne peut se produire que si l'on additionne deux nombres de même signe et que l'on obtient un signe différent. Ainsi, pour détecter un débordement, il faut simplement regarder le bit de signe (MSB). [1]

Avec deux opérandes et un résultat, nous avons trois bits de signe (chacun 1 ou 0) à prendre en compte, de sorte que nous avons exactement $2^3 = 8$ combinaisons possibles des trois bits. Seuls deux de ces huit cas possibles sont considérés comme des dépassements de capacité.

Dans la Table 1 figurent uniquement les bits de signe des deux opérandes d'addition et du résultat :

Op1	**Op2**	**Résultat**	**Débordement ?**
0	0	0	
0	0	1	Oui
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	Oui
1	1	1	

Table 1 – Calcul de débordement pour une addition signée (seulement le MSB est visualisé)

De manière similaire, la Table 2 montre les bits lors d'une soustraction. Notez que la soustraction d'un nombre positif est la même chose que l'addition d'un nombre négatif, de sorte que les conditions qui s'appliquent à la soustraction sont les mêmes que pour l'addition. Donc les conditions qui déclenchent le drapeau de débordement sont :

Op1	**Op2**	**Résultat**	**Débordement ?**
0	0	0	
0	0	1	
0	1	0	
0	1	1	Oui
1	0	0	Oui
1	0	1	
1	1	0	
1	1	1	

Table 2 – Calcul de débordement pour une soustraction signé (seulement le MSB est visualisé)

2.2 Shifter

Le shifter est un composant de l'étage EXEC permettant d'effectuer un décalage de bit.

Le shifter a une fonction importante, il peut réaliser des multiplications ou divisions par un multiple de 2. Ce composant est principalement utilisé lors des instructions de branchements, accès mémoire, et opérations de traitement de données qui ont besoin d'un décalage.

Dans cette implémentation d'un processeur ARM v3, il faut utiliser un Barrel Shifter. La différence entre un shifter et un *barrel shifter* réside dans les instructions de rotation. Puisque l'espace pour coder les instructions est limité, il n'existe pas d'Opcode pour faire une rotation gauche, néanmoins on peut effectuer une rotation gauche avec des rotations droite consécutives.

2.2.1 Opérations du shifter

Logical Shift Left - LSL Les bits sont décalés à gauche par le montant spécifié, en remplissant les bits vides avec '0'. Équivalent à << en C, c'est-à-dire multiplication non signée par une puissance de 2.

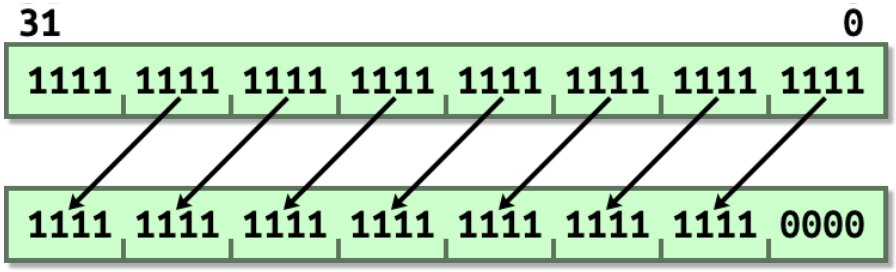

FIGURE 2 – Décalage logique à gauche

Logical Shift Right - LSR Les bits sont décalés à droite par le montant spécifié, en remplissant les bits vides avec '0'. Équivalent à >> en C, c'est-à-dire division non signée par une puissance de 2.

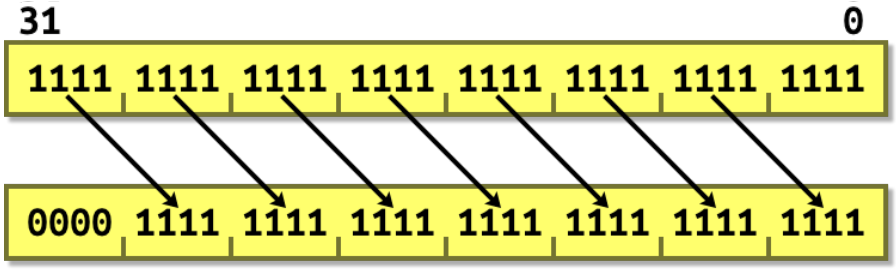

FIGURE 3 – Décalage logique à droite

Arithmetic Shift Right - ASR Les bits sont décalés à droite par le montant spécifié, en remplissant les bits vides avec le signe du bit le plus significatif ('1' ou '0'). Équivalent à >> en C, c'est-à-dire division signée par une puissance de 2.

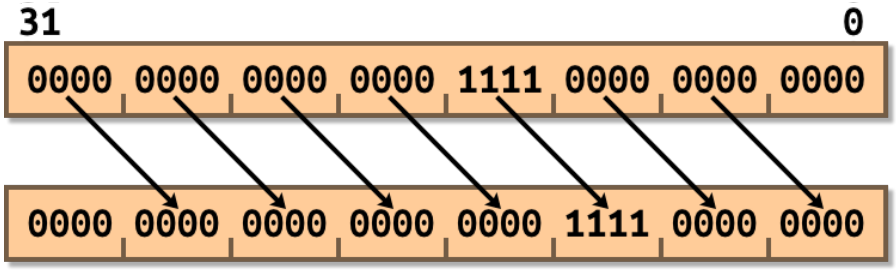

Figure 4 – Décalage arithmétique à droite

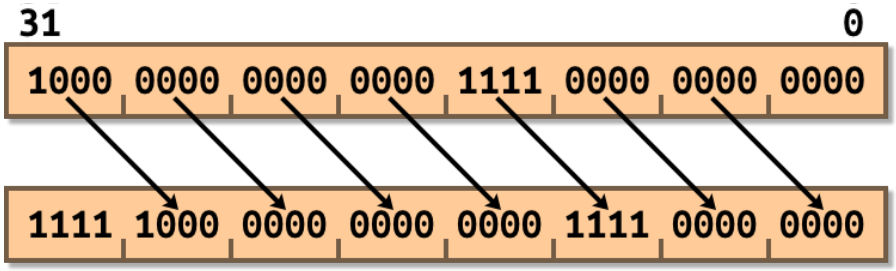

Figure 5 – Décalage arithmétique à droite

Rotate Right - ROR

Les bits sont décalés à droite par le montant spécifié, en remplissant les bits de poids fort par les bits de poids faible. Il n'existe pas d'équivalent en C.

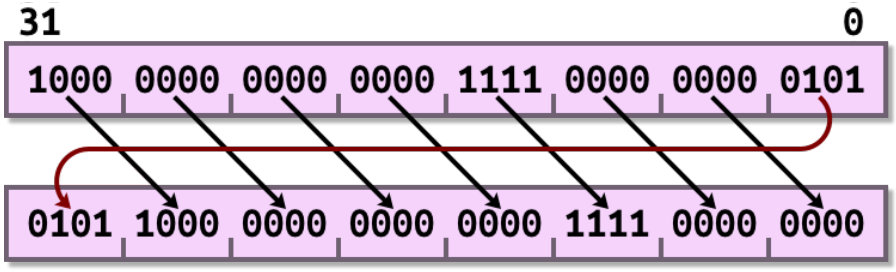

Figure 6 – Rotation à droite

Rotate Right Extended - RRX

Similaire au ROR, la rotation à droite étendue applique une rotation à droite des bits d'un montant d'un bit à la fois. On dit que c'est une rotation "étendue" car la carry est incluse dans la rotation en remplissant le bit de poids fort, la nouvelle carry correspond à l'ancien bit de poids faible. Il n'existe pas d'équivalent en C.

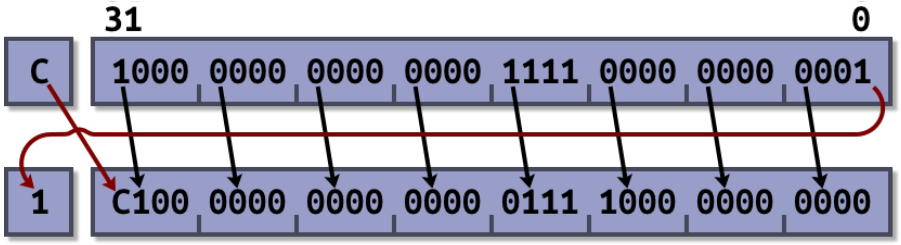

Figure 7 – Rotation à droite étendue

Nous remarquons qu'il y a 5 opérations pour le shifter, mais seulement deux bits pour la sélection de l'opération ($2^2 = 4$). La rotation étendue est exécutée uniquement lorsque le montant de rotation est égal à zéro.

De façon similaire à l'additionneur, nous avons concaténé un bit extra au signal d'entrée et sortie pour obtenir la retenue.

Grâce à la librairie `std_numeric`, nous pouvons nous servir de fonctions telles que `shift_left`, `shift_right`, `rotate_right` pour réaliser les opérations nécessaires de décalage et rotation.

Pour cela il faut faire un cast des données en entrée (`std_logic_vector`) vers `unsigned` sauf pour le décalage arithmétique où on fait un cast vers `signed` pour éteindre le signe.

Puis on capture soit le MSB (pour le décalage vers la gauche ou les rotations à droite) soit le LSB (pour le décalage à droite) pour la carry.

2.3 Multiplexeurs

Dans un premier temps, nous pensions créer nous-même l'architecture des multiplexeurs puis les instancier dans EXEC. Cela aurait permis d'abstraire le fonctionnement du multiplexeur et de rendre plus lisible le code de EXEC.

Finalement nous nous sommes rendu compte que cela rajoutait un composant presque vide à notre projet et que nous n'y gagnions pas forcement en lisibilité. Nous sommes donc resté sur une utilisation directe des structures de contrôle que propose VHDL(when...else, if..else).

2.4 FIFO

Étant donné que le processeur a besoin de plusieurs FIFO de tailles différentes, nous avons d'abord crée une architecture générale à l'aide du mot clé `generic`.

```
entity fifo is
    generic(
        N : integer
    ); -- FIFO Size
    port
    (
```

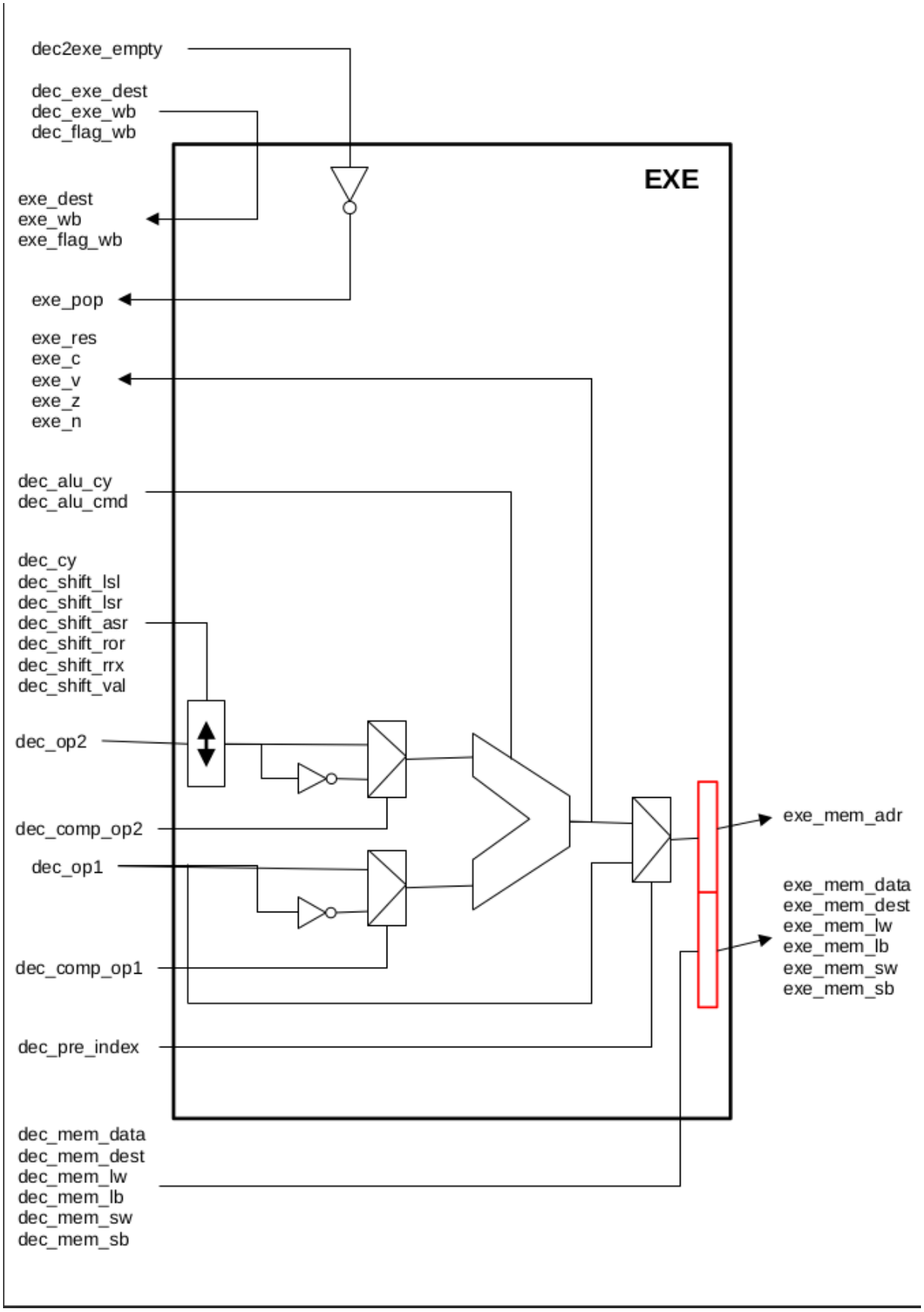

FIGURE 8 – Étage EXE détaillé

```
        din  : in std_logic_vector(N - 1 downto 0);
        dout : out std_logic_vector(N - 1 downto 0);
    ...
```

La taille de la FIFO est donc fonction de N.

Lors de l'instantiation du composant, on fournit la taille de la nouvelle FIFO. Cela nous permet de travailler sur un fichier seulement en simplifiant le développement.

Exemple pour la FIFO de DECOD de 127 bits :

```
exec2mem : fifo_generic
    generic map(N => 127)
    port map
    (
        din(126)           => pre_index,
        din(125 downto 94) => op1,
      ...
```

2.5 Mise en place de l'étage EXEC

Une fois tous les composants testés (Additionneur, ALU, Shifter, FIFO) nous avons procédé à l'implémentation de l'étage d'exécution du processeur.

Comme nous pouvons le voir sur la figure 8 l'opérande 1 est branchée directement à la première entrée de l'ALU. Par contre, l'opérande 2 est branchée au Shifter, puis à la deuxième entrée de l'ALU pour pouvoir effectuer des opérations de décalage.

3 Étage de décodage (DECOD)

Cet étage comporte trois composants principaux :

- **Le Banc de registres** contenant les 16 registres (dont le PC)
- **La Machine à états** qui gère l'état du processeur et donc ses actions à partir de l'instruction qui vient d'être décodée
- **Les FIFO** qui font le lien avec l'étage précédent FETCH et l'étage suivant EXEC. Elles sont implémentées grâce à l'architecture générique décrite un peu plus haut.

3.1 Banc de registres

Les registres permettent de stocker les résultats des instructions afin qu'ils puissent être utilisés par les instructions suivantes. Le banc de registres est constitué de 16 registres dont le Program Counter (PC) qui est mis à jour à chaque cycle, et des flags CPSR.

Pour cette partie, nous avons opté, dans un premier temps, pour la création de 16 signaux correspondant à chaque registre (R0 - R15). Néanmoins, nous avons trouvé que le code était trop long et donc nous avons cherché une façon plus simple de le faire.

Un extrait de la première version du code est montré dessous :

```
case (wadr1) is
       when x"0" =>
         -- Verify if the regiter is invalid
         test_validtyBit := regs_v and (b"0000_0000_0000_0001");
         if (not(test_validtyBit = b"0000_0000_0000_0000")) then
           reg0   <= wdata1;
           regs_v <= regs_v or b"0000_0000_0000_0001";
         end if;
```

Dans un second temps, nous avons simplifié notre implémentation à l'aide d'un tableau de 16 `std_logic_vector` de 32 bits. Comme ceci, les numéros des registres peuvent être utilisés comme index du tableau ce qui simplifie grandement le code.

Un extrait de la deuxième version est montré ci-dessous :

```
-- Assign invalidation bit to the register corresponding to the address
reg_bank_validity(invadr1_i) <= '0' when (inval1 = '1');
reg_bank_validity(invadr2_i) <= '0' when (inval2 = '1');

-- Always save EXEC result
if (wen1 = '1') then
  if (reg_bank_validity(wadr1_i) = '0') then
    reg_bank(wadr1_i) <= wdata1;
    -- Validate after writing
    reg_bank_validity(wadr1_i) <= '1';
  end if;
end if;
```

Un problème est survenu lors des tests car au "démarrage" les valeurs étaient indéfinies et donc le casting vers un nombre entier n'était pas possible. La solution a été de contraindre les valeurs possibles entre 0x0 et 0xF. Si la valeur était hors ce rang, elle basculait à 0x0. Avec cette solution, le code reste synthétisable.

```
signal wadr1_i   : integer range 0 to 15 := 0;
...
wadr1_i <= to_integer(unsigned(wadr1)) when (wadr1 >= x"0" and wadr1 <= x"F")
        else 0;
...
```

Cela a facilité grandement l'accès aux registres et rendu le code plus facile à comprendre, et par conséquent à déboguer.

3.1.1 Algorithme d'écriture des registres

Afin d'écrire dans un registre particulier du banc de registres, il faut vérifier que certaines conditions soient remplies afin de compléter l'écriture.

Le CPSR vérifie deux parties séparément, les drapeaux CZN et le drapeau V pour les instructions arithmétiques.

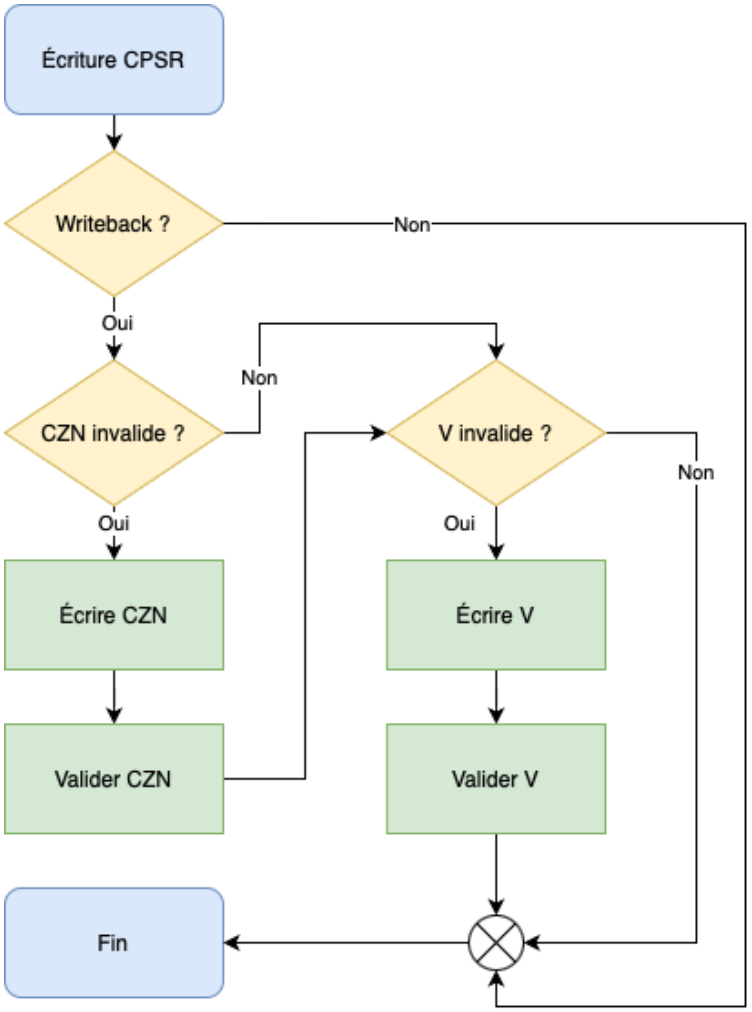

Figure 9 – Écriture de CPSR

L'écriture dans les 16 registres suit le même principe pour le CPSR, sauf qu'il existe deux ports d'écriture. Le premier port appartient à l'étage EXE et a donc la priorité par rapport au deuxième port appartenant à l'étage MEM.

En cas d'un conflit où les deux ports d'écriture ont la même adresse (registre de destination), il faut ignorer l'écriture provenant de MEM car c'est une valeur plus ancienne par rapport au résultat produit par l'étage EXEC.

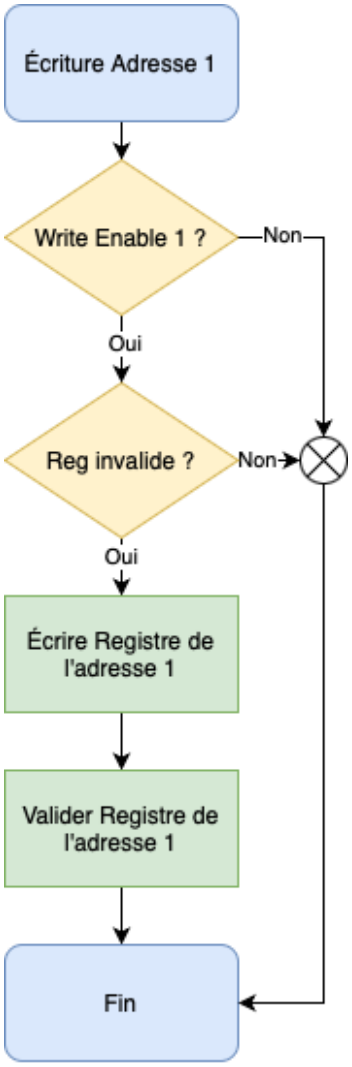

FIGURE 10 – Écriture de port 1

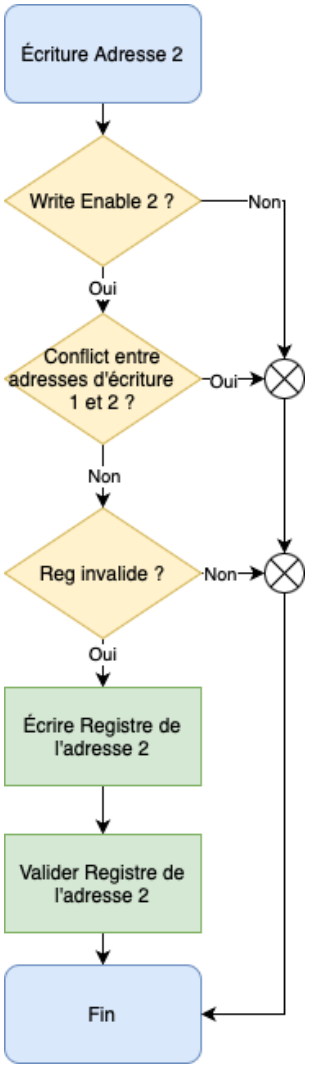

FIGURE 11 – Écriture de port 2

3.1.2 Program Counter

Le Program Counter (PC) est un registre particulier car il augmente de 4 octets à chaque cycle d'horloge quand le signal `inc_pc` est activé.

C'est comme cela que le processeur peut lire instruction par instruction. Le PC lit le mot suivant pour décoder la prochaine instruction codé sur 32 bits (4 octets).

Lors d'un branchement, l'écriture dans ce registre fonctionne normalement en suivant les algorithmes décrits par la figure 10 et la figure 11.

3.2 Décodage des instructions

Une fois le banc de registres complété, nous avons procédé à l'implémentation de l'étage DECOD. À l'aide du manuel de référence ARM et des documents donnés lors du cours, nous avons commencé par les 16 conditions d'exécution. Ces conditions doivent vérifier les drapeaux du CPSR ainsi que leur validité.

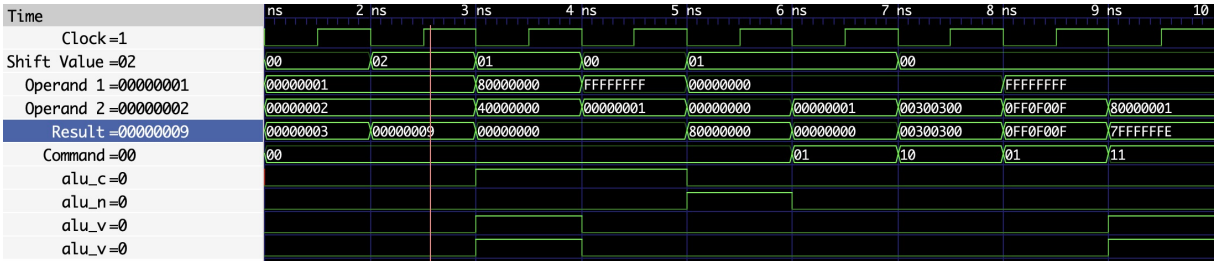

Figure 12 – Visualisation des signaux de l'étage EXEC du test bench

Ensuite, il fallait spécifier le type d'instruction parmi les 6 possibilités (traitement de données, branchements, swap, multiplication, accès mémoire simple et multiple), et leurs Opcode correspondants.

Les commandes du shifteur et sa valeur de décalage ont été assez directes à décoder : nous l'avons donc implémenté facilement.

Néanmoins, la sélection des opérandes pour l'ALU et leur validité a été plus compliqué à gérer ainsi que l'offset pour les instructions de branchement.

3.3 Machine à états

La machine à états finis (FSM) de l'étage DECOD est de type Mealy car les valeurs de sortie sont déterminées non seulement par l'état actuel, mais également par les entrées.

Ce FSM comporte principalement 5 états :

1. Fetch
2. Run
3. Branch
4. Link
5. Transfert Mémoire

4 Plateforme de simulation

Nous avons utilisé l'IDE Visual Studio Code pour coder les fichiers VHDL.

Nous avons optimisé le processus de compilation et simulation des tests bench avec un fichier makefile adapté à nos besoins.

Nous avons rendu les tests bench auto-testant en utilisant des assertions pour montrer des messages sur le terminal.

```
assert(exe_res_s = x"0030_0300") report "Incorrect res, exe_res = 0x" &
    to_hstring(exe_res_s) severity error;
```

En outre, le logiciel GTKWave a été essentiel afin de pouvoir visualiser les signaux internes (Figure 12 et Figure 13).

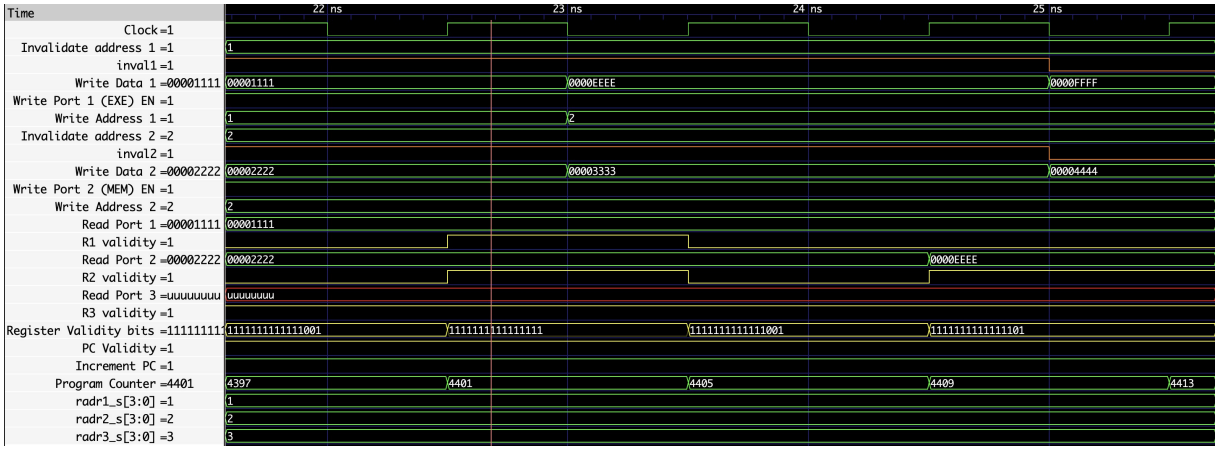

FIGURE 13 – Visualisation des signaux du banc de registres du test bench

5 Points à améliorer

Malheureusement nous n'avons pas pu nous concentrer davantage sur l'étage DECOD et la machine à états. Nous aurions pu tester le fonctionnement du processeur avec des programmes en assembleur. De plus, implémenter le branchement aurait été particulièrement intéressant car cela nous aurait permis d'observer comment le PC est modifié lors d'un saut d'adresse.

Par ailleurs, le banc de registres a été fait correctement dans un premier temps, néanmoins nous avons passé trop de temps sur cette partie car lors des tests nous n'avons pas compris que nous ne pouvions pas invalider et écrire dans un même cycle. Il fallait invalider, puis écrire (donc deux cycles d'horloge).

6 Conclusions

En premier lieu, le développement de ce projet nous a permis de mieux comprendre le fonctionnement des processeurs pipelinés, non seulement ceux basés sur l'architecture ARM mais également ceux basés sur l'architecture MIPS, étudiée dans d'autres UE.

En outre, nous avons compris le mécanisme des cycles de gel et comment ce mécanisme se déclenche quand les opérandes ne sont pas valides.

Enfin, le projet nous aura aussi fait découvrir ou redécouvrir les différents outils de description de matériels, de synthèse et de routage, nécessaires à la VLSI.

Acronymes

CPSR Current Program Status Register.

FIFO First In First Out.

FPGA Field Programmable Gate Array.

LSB Least Significant Bit.

MSB Most Significant Bit.

Opcode Operation Code.

PC Program Counter.

VHDL VHSIC Hardware Description Language.

Glossaire

Barrel Shifter Circuit électronique capable d'effectuer des opérations de décalage et de rotation binaires.

Références

[1] Ian D. Allen. *The CARRY flag and OVERFLOW flag in binary arithmetic*. `http://teaching.idallen.com/dat2343/11w/notes/040_overflow.txt`. Accessed : 2023-10-1.

[2] *ARM Architecture Reference Manual*. I. ARM Limited. Juill. 2005.