Rapport d'avancement de Projet VLSI

Auteurs:

BISLIEV Magomed-Salakh
YANG Zixiao

Décembre 2022

Sommaire

I Introduction

Dans le cadre de ce projet nous réalisons la description en VHDL d'un processeur ARM pipepilé c'est-à-dire que l'exécution des instructions est découpée en plusieurs étapes. Notre processeur comporte 4 étages IFETCH, DECOD, EXE et MEM et chaque étage possède un rôle spécifique :

- IFETCH : récupération de l'instruction en mémoire
- DECOD : décodage de l'instruction
- EXE : réalisation des opérations élémentaires
- MEM : lecture ou écriture d'une donnée en mémoire

Actuellement nous sommes parvenus à clore l'étage exe et le banc de registre de Decod.

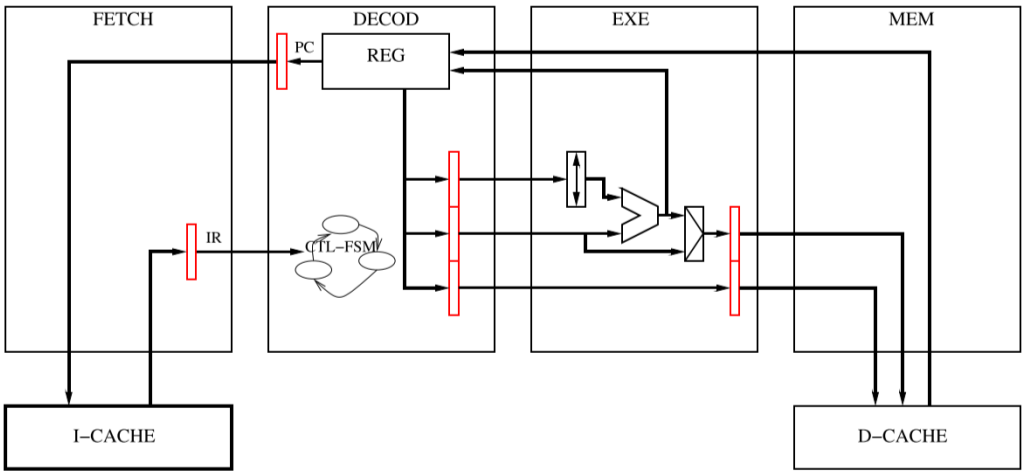

Fig. 1. Schéma du pipeline

II Étage Exe

Tout d'abord nous avons réalisé l'étage exe qui est chargé de réaliser des opérations élémentaires et pour cela il comporte deux blocs : l'arithmetic logic unit (ALU) et le shifter. Il comporte également une pile de type first in first out (FIFO) qui sépare cet étage de l'étage MEM. La pile nous permet de transmettre les adresses et données de l'étage exe vers l'étage mem en fonction de l'état (pleine ou vide) des piles (exe2mem et dec2exe) et de si on effectue un accès mémoire (lw, lb, sw ou sb). Le schéma ci-dessous décrit le comportement de l'étage et l'interconnexion des différents composants.

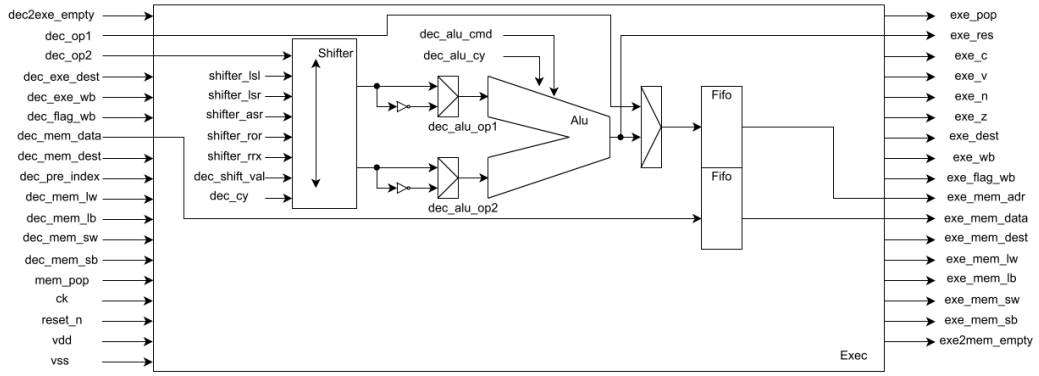

Fig. 2. Schéma global d'Exe

II.1 Arithmetic logic unit

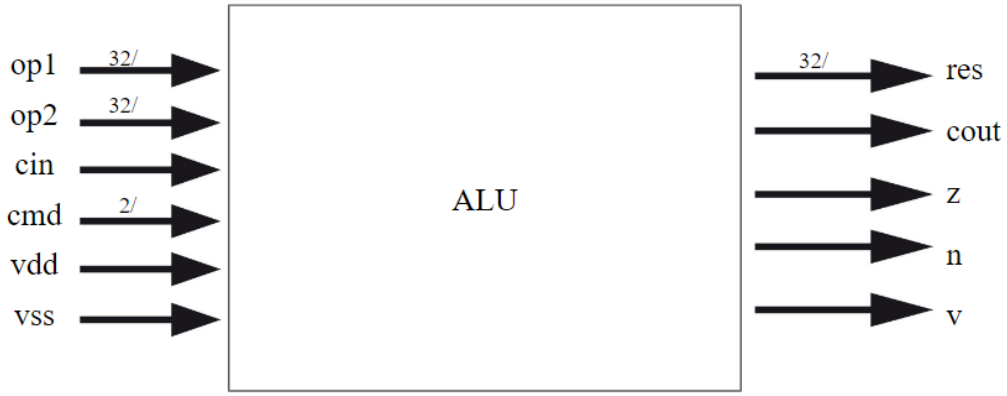

Fig. 3. Schéma bloc de l'Alu

L'Alu est le composant qui nous permet de réaliser les 4 opérations de base : ADD avec retenue, AND, OR et XOR. Mais afin de sélectionner l'opération voulue entre les deux opérandes de 32 bits, nous utilisons une commande sur deux bits. Les opérations AND, OR et XOR sont très simples à implémenter à l'aide des opérateurs qui sont déjà mis à disposition. Cependant, pour l'opération d'addition, nous avons utilisé un Full Adder en l'instanciant 32 fois. Lors de la synthèse nous avons constaté que l'instanciation avec un seul "generate (de 0 à 31)" génère une erreur sur les retenues car les retenues 1 à 31 ne sont pas assignées. En effet, nous avons dû réaliser une première instanciation pour produire le résultat de rang 0, ensuite l'instanciation avec un generate de 1 à 30 et une dernière instanciation pour produire le résultat de rang 31.

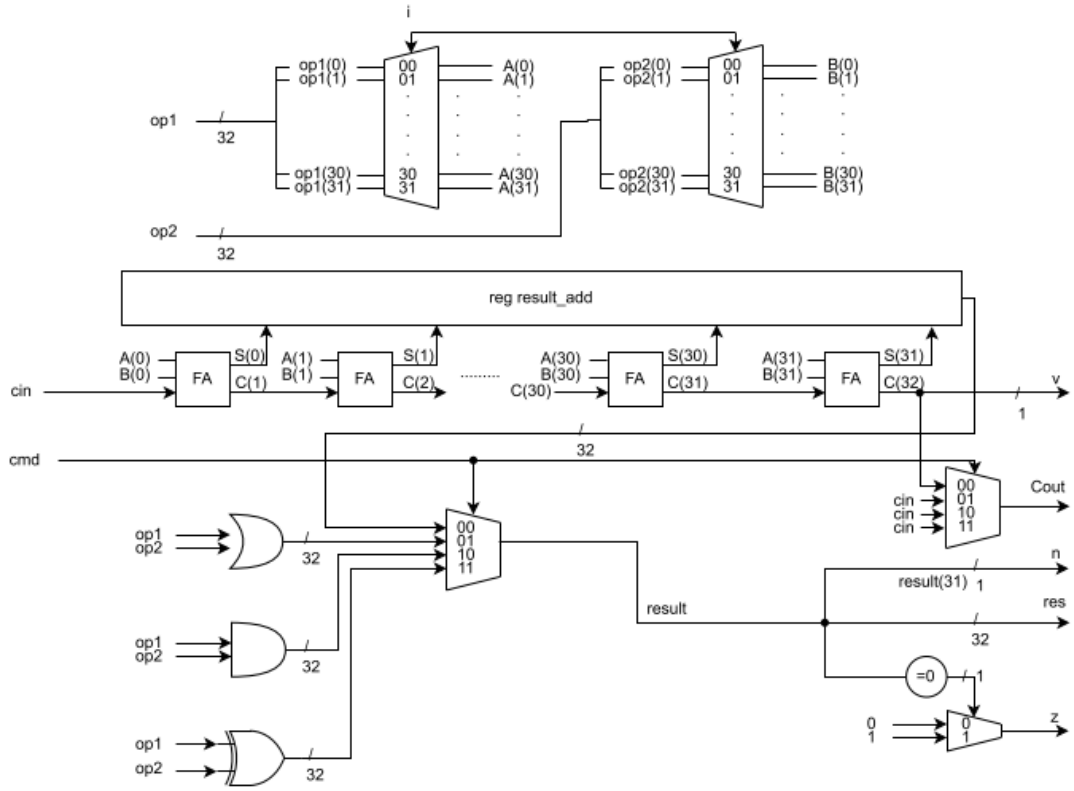

Fig. 4. Schéma des opérations réalisés par l'Alu

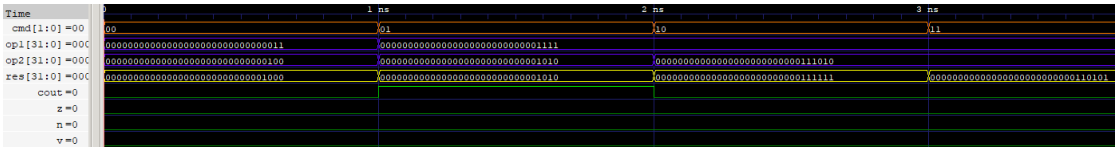

Fig. 5. Résultat de simulation pour l'Alu

La figure ci-dessus illustre le bon fonctionnement de l'alu. En effet, les opérations suivantes sont réalisées dans l'ordre add, and, or et xor en fonction de la commande sur 2 bits .

Dans ce projet nous réalisons la synthèse en 3 étapes en utilisant 3 logiciels de la suite alliance. Tout d'abord avec vasy nous réalisons une conversion .vhdl en .vbe afin d'obtenir des structures simples. Puis avec boom nous réalisons une optimisation booléenne donc nous avons toujours un .vbe. Enfin, avec boog nous réalisons une projection structurelle afin de convertir les équations en netlist de portes logiques. Afin d'illustrer ces 3 étapes nous pouvons nous appuyer sur un exemple très simple qui est le premier composant du projet, le Full Adder.

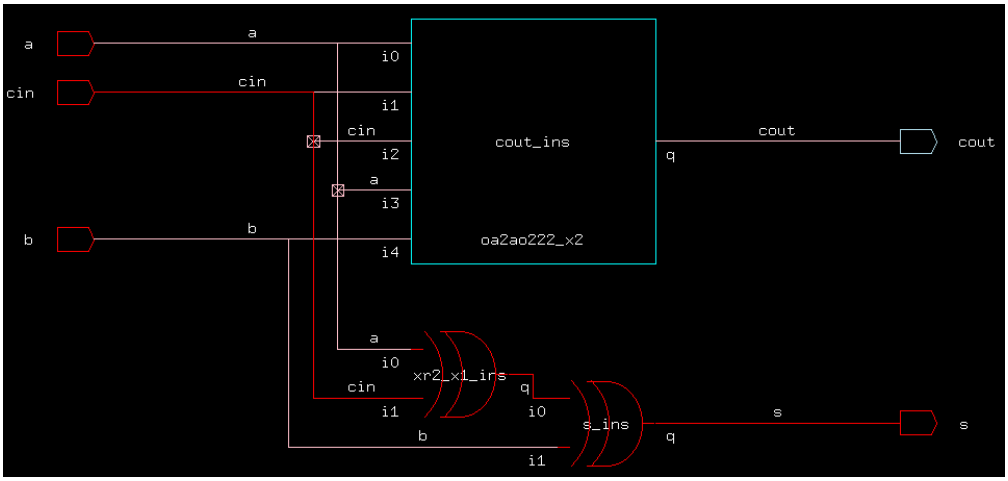

Fig. 6. Résultat de synthèse pour le Full Adder

cout = (A and B) or (Cin and B) or (A and Cin) = A and (B or Cin) or (B and Cin) Pour la sortie s on s'attend à obtenir le schéma en portes logiques suivant A xor B xor Cin, cela est cohérent avec le résultat de synthèse. Cependant pour la sortie cout on s'attendait à obtenir (A and B) or (Cin and B) or (A and Cin) (1) soit un oa23_x2 dans la terminologie sxlib, mais le bloc de synthèse correspond à un oa2ao222_x2, on remarque que cela correspond à l'équation logique suivante A and (B or Cin) or (B and Cin). On en déduit qu'il s'agit juste d'une optimisation de l'équation (1) réalisée par boom car le A a été mis en facteur. On peut également remarquer que la synthèse présente un chemin rouge, ce-dernier représente le chemin critique c'est-à-dire le chemin le plus long du composant.

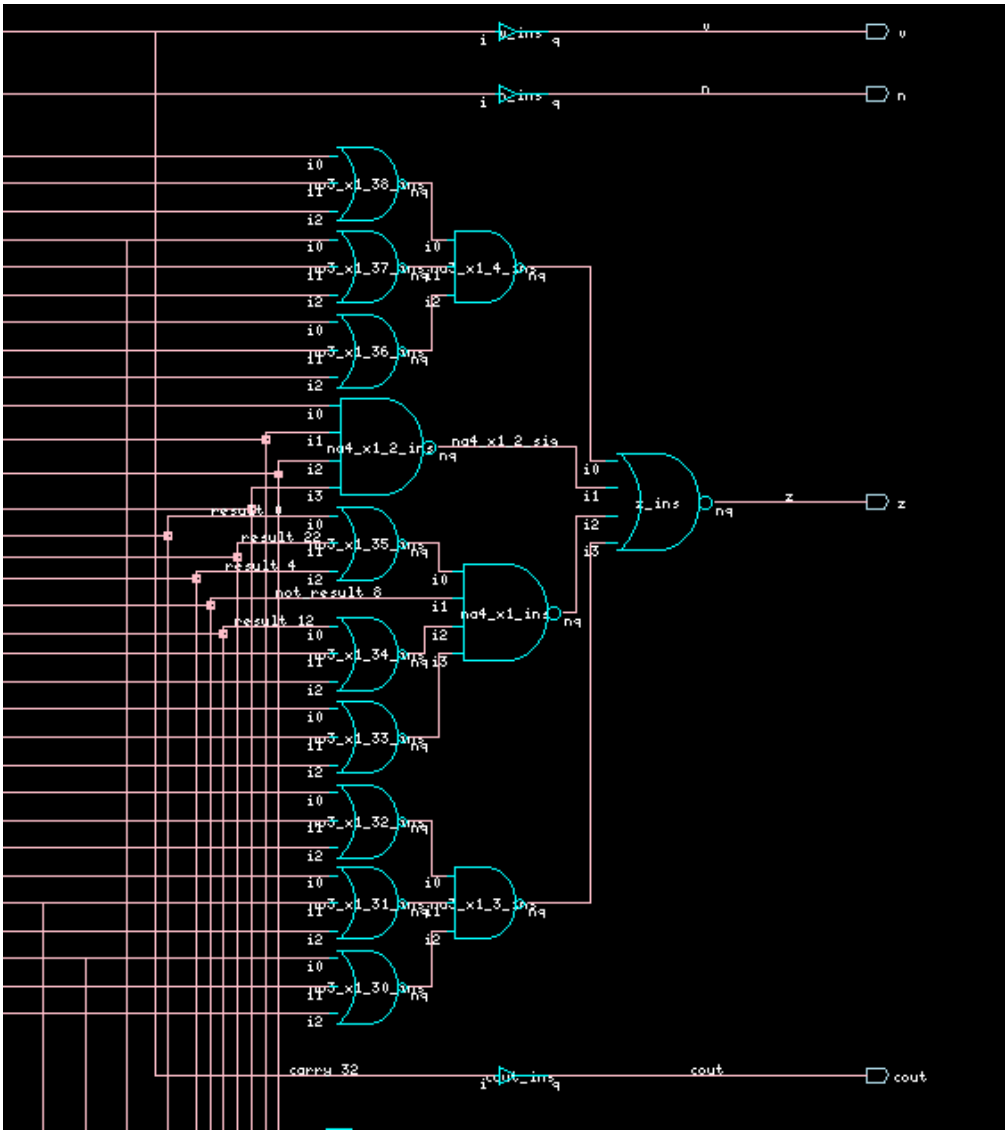

Fig. 7. Résultat de synthèse pour l'Alu (zoom sur la génération des flags czn et v)

- c : retenue de sortie
- z : résultat nul
- n : résultat négatif
- v : dépassement de capacité

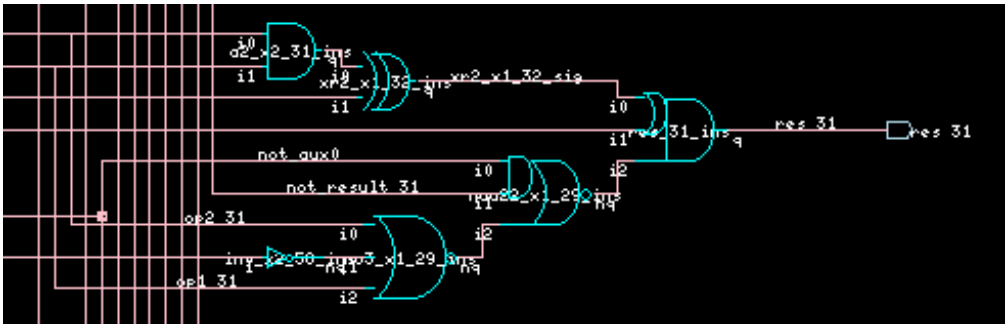

Fig. 8. Résultat de synthèse pour l'Alu (zoom sur le bit 31 du vecteur res)

Le vecteur res correspond à la sortie de l'alu sur 32 bits et le schéma en portes logiques des bits de rang inférieur reste identique à celui représenté sur la figure.

II.2 Shifter

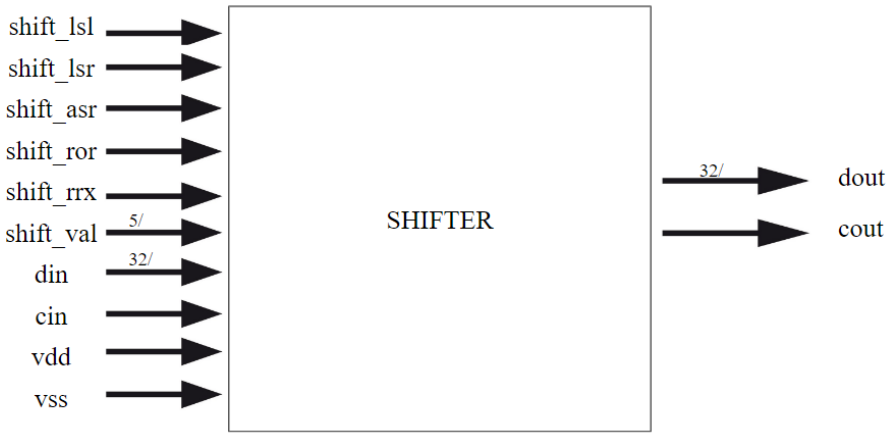

Fig. 9. Schéma bloc du Shifter

Un autre composant essentiel pour notre processeur ARM est le shifter. En effet, ce-dernier nous permet de réaliser des décalages et rotations. La technique employée consiste à réaliser des décalages ou des rotations progressifs. La valeur de décalage étant définie par une valeur binaire sur cinq bits, nous pouvons décomposer l'opération en des décalages de 2^N avec N=0, 1, 2, 3 et 4. Ainsi, chaque valeur de décalage comprise entre 0 et 31 peut être réalisée. On comprend donc que nous réalisons des concaténations de 0 ou 1 en fonction de s'il s'agit d'un décalage logique ou arithmétique.

II.2.1 Décalages logiques

Pour les décalages logiques : Logic shift left(LSL) et Logic shift right(LSR) nous réalisons simplement des concaténations de 0.

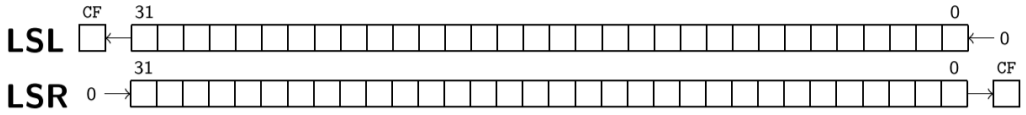

Fig. 10. Décalages logiques shift left(LSL) et shift right(LSR)

II.2.2 Décalage arithmétique vers la droite

Cependant, dans le cas d'un décalage arithmétique vers la droite : arithmetic shift right(ASR) nous devons tenir compte du bit de signe pour la concaténation.

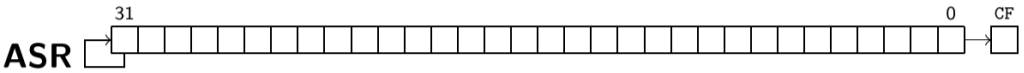

Fig. 11. Décalage arithmétique vers la droite (ASR)

II.2.3 Rotation vers la droite

Dans le cas d' une rotation nous utilisons directement les bits de notre opérande pour réaliser la concaténation.

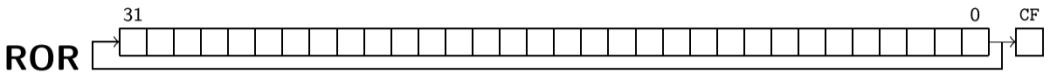

Fig. 12. Rotation vers la droite (ROR)

II.2.4 Rotation vers la droite avec étendue

Pour la rotation vers la droite avec étendue nous réalisons une seule concaténation avec la retenue d' entrée.

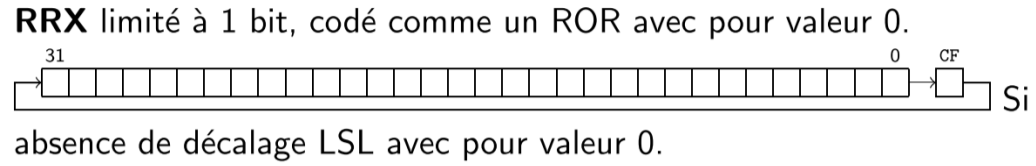

Fig. 13. Rotation vers la droite avec étendue (RRX)

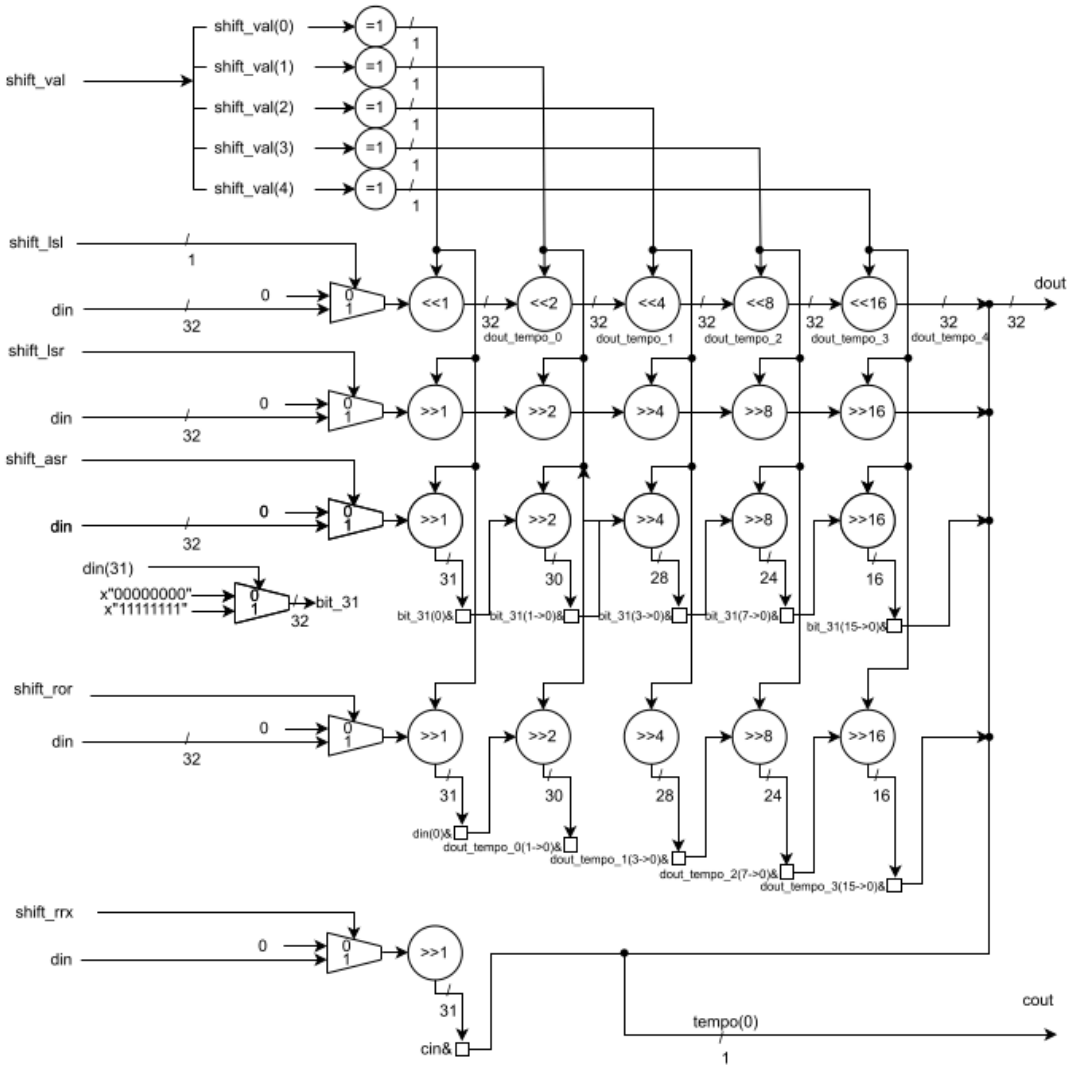

Fig. 14. Le principe des décalages

La figure ci-dessus illustre les différents décalages en fonction de l'état binaire de shift_lsl, shift_lsr, shift_asr, shift_ror et shift_rrx. Nous réalisons un décalage de 2^n bits lorsque le bit de rang n du vecteur shift_val est à 1. Le bit din(31) est utilisé pour réaliser une extension de signe dans le cas où l'on souhaite réaliser un ASR.

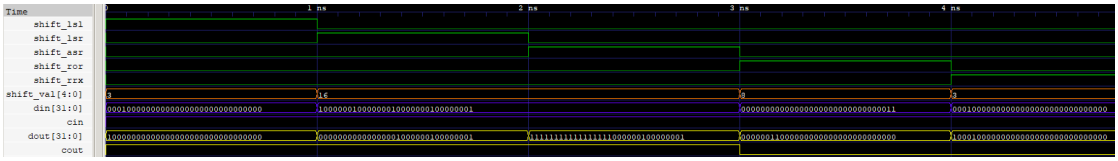

Fig. 15. Résultat de simulation pour le shifter

Afin de vérifier le bon fonctionnement du shifter, nous réalisons les 5 décalages possibles. Dans la réalisation de ce composant il a fallu particulièrement faire attention à réaliser correctement l'extension de signe dans le cas d' un décalage arithmétique et de correctement identifier la retenue de sortie en fonction du type de décalage car elle vaut dout(31) dans le cas d'un LSL et dout(0) sinon (LSR, ASR, ROR et RRX).

III Étage Decod

III.1 Le banc de registre

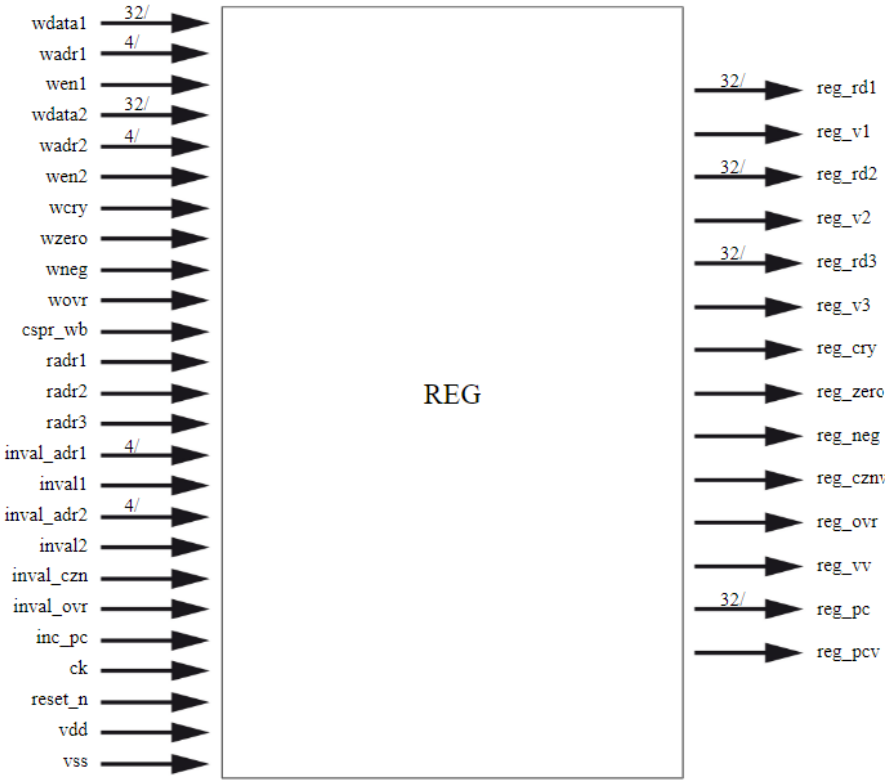

Fig. 16. Schéma bloc de reg

L'étage decod est l'étage le plus complexe du processeur car ce dernier est chargé de décoder les instructions de l'architecture ARM... . En effet, il existe différents types d'instructions et ces dernières utilisent des opérandes sur 32 bits, il convient d'utiliser des registres pour ces opérandes et pour cela nous avons utilisé un bloc appelé reg. Ce bloc contient :

- 3 ports de lecture numérotés de 1 à 3,
- 2 ports d' écriture, le numéro 1 correspond à exec et est donc prioritaire,
- 2 ports d' invalidation car une instruction peut produire 2 résultats,
- 4 flags : c,z,n,v et leurs 2 bits de validité (1 bit de validité pour czn et 1 pour ovr),
- PC, son bit de validité et sa commande de +4

Pour le banc de registre nous avons décidé de réaliser un process synchrone qui est sensible sur l'horloge(ck) et reset_n pour l'écriture dans les registres et la gestion des bits de validité associés, la mise à jour des flags et la gestion des bits de validité associés, enfin la gestion de pc (incrémentation de 4). Au reset tous les bits de validité sont initialisés à '1'. Cependant, pour la lecture nous avons décidé de la réaliser en concurrent. Pour l'écriture lorsqu'un registre est identifié comme une destination, ce dernier est invalidé, pour cela une solution est d'utiliser un vecteur invalid dont le bit de rang n indique s'il est égal à 1 que le registre n est invalidé. Lorsque l'écriture a été effectuée, le registre est à nouveau validé. Afin de gérer la priorité entre le port 1 et 2, nous avons créé 2 autres vecteurs wadr1_v et wadr2_v, lorsque l'on souhaite écrire dans le registre n via le port 1 il faut que wadr1_v(n) soit à 1, cependant lorsque l'on veut écrire dans le registre n via le port 2 il faut que wadr2_v(n) soit à 1 et wadr1_v(n) à 0.

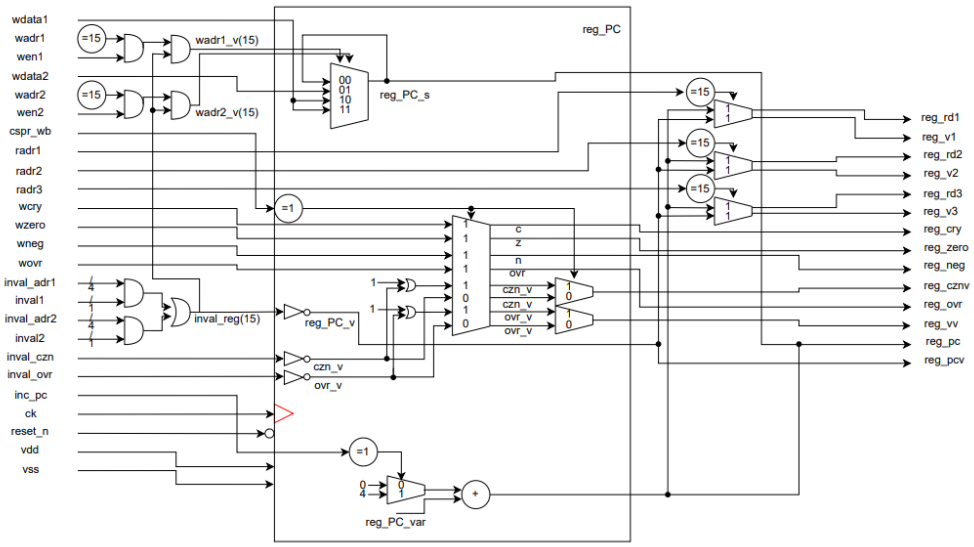

Fig. 17. Schéma global de reg

La figure ci-dessus illustre le principe de fonctionnement du banc de registre pour un registre précis mais le principe reste le même pour les 15 autres registres. En effet, ici nous utilisons un exemple où le registre 15 (PC) est la destination d' une écriture via le port 1 et 2, cela nous permet de montrer la prise en compte de la priorisation du port 1 lorsque l'adresse du registre de destination est la même pour les 2 écritures.

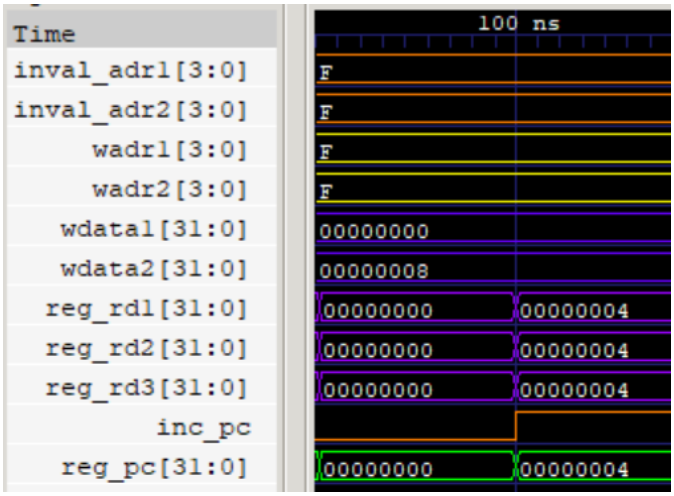

Fig. 18. Résultat de simulation pour le banc de registre

Le résultat de simulation ci-dessus, nous montre que pour une même adresse de destination qui est le registre PC ici, la donnée est écrite via le port 1. En effet, ici les 3 ports de lecture lisent le même registre (PC) et nous voyons bien que la valeur lue correspond à celle de wdata1 et non wdata2. Enfin, nous avons testé l'incrémentation du registre pc, nous remarquons qu'elle se fait bien lorsque inc_pc vaut 1.

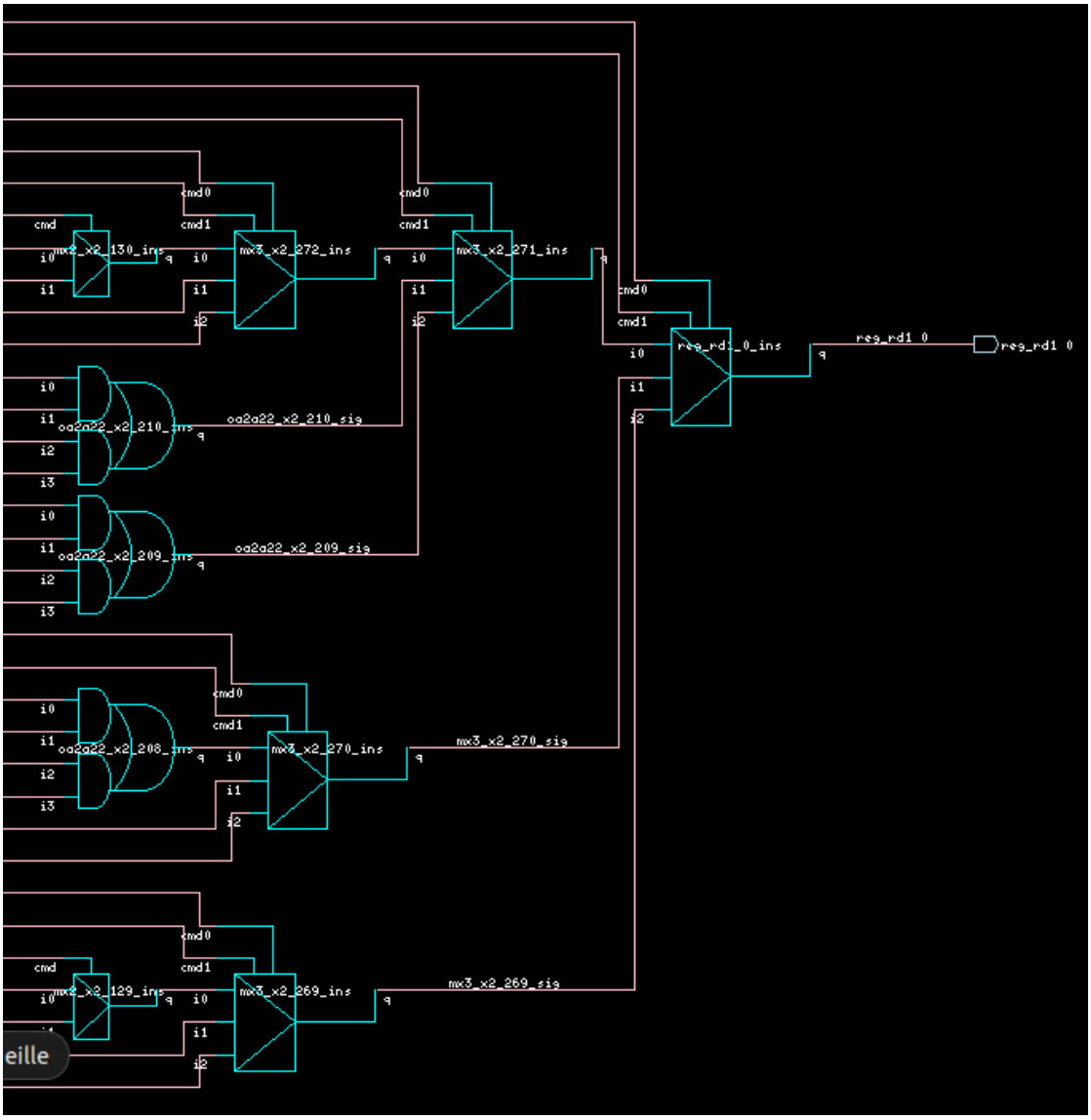

Fig. 19. Résultat de synthèse pour le banc de registre (zoom sur le port de lecture 1)

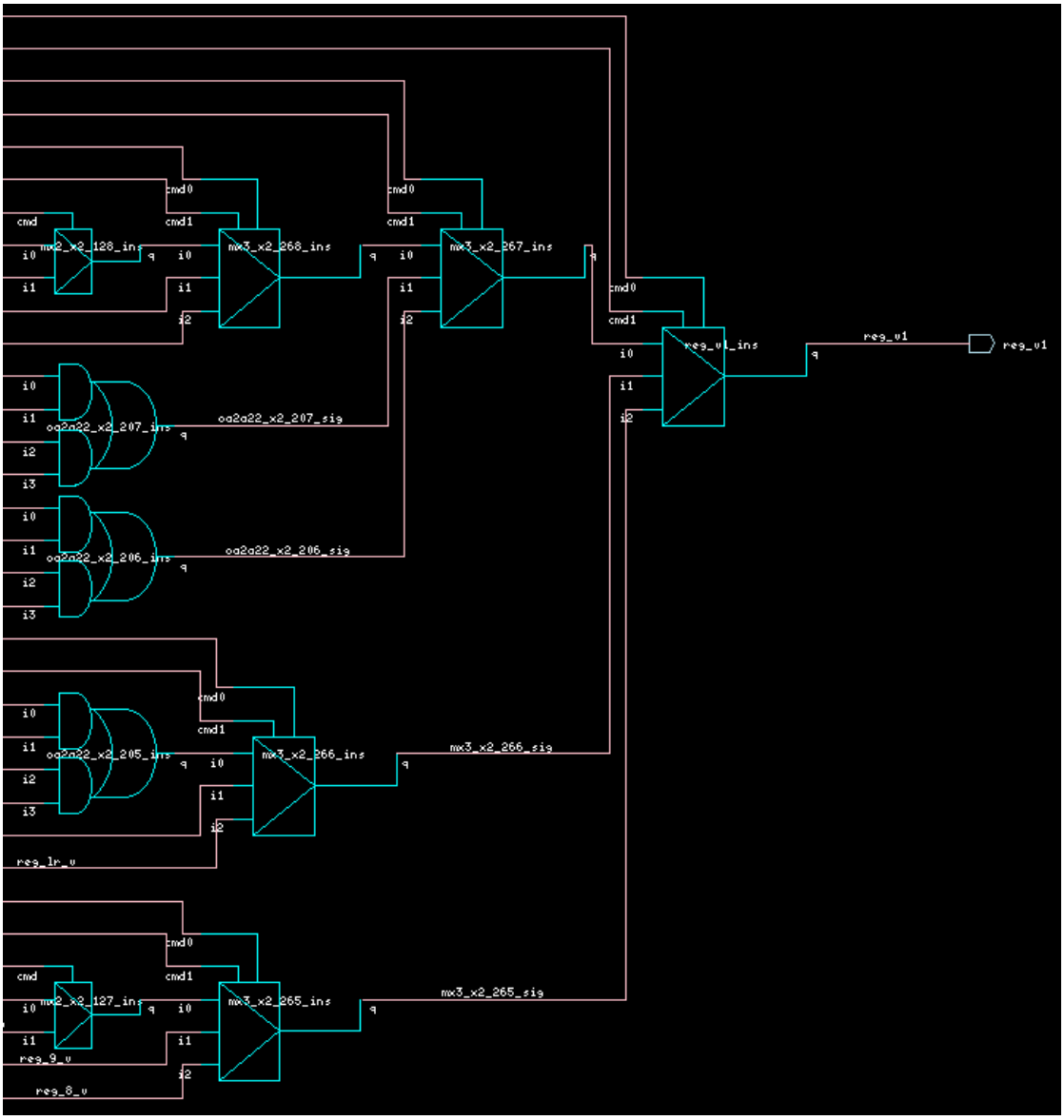

Fig. 20. Résultat de synthèse pour le banc de registre (zoom sur la gestion du bit de validité du port de lecture 1)

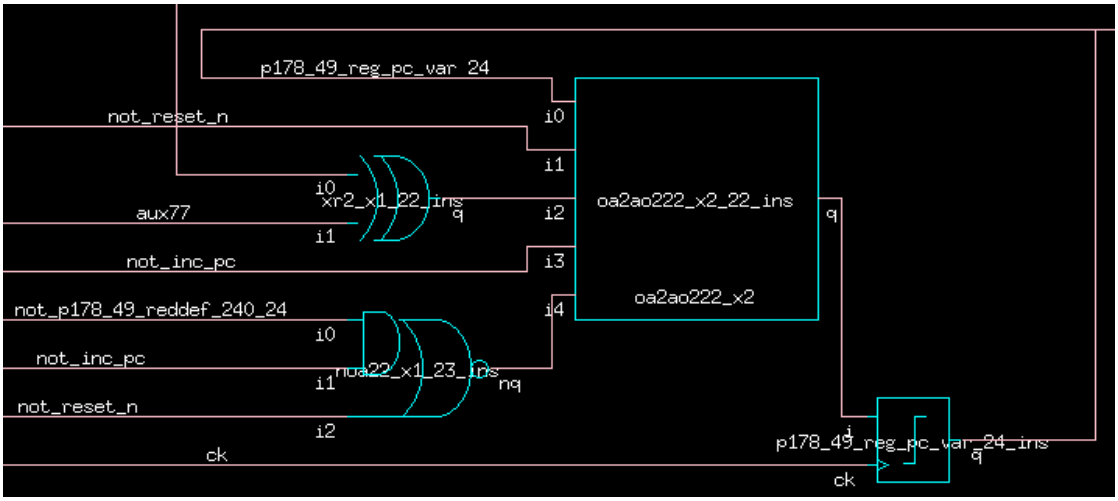

Fig. 21. Résultat de synthèse pour le banc de registre (zoom sur la gestion de PC)