VLSI

Rapport de projet :

Modélisation d'un cœur de processeur ARM

Louis Geoffroy Pitailler, Timothée Le Berre

Décembre 2021

Sommaire

1 Introduction

Au cours de ce projet, on cherche à décrire le cœur d'un processeur basé sur une architecture ARM. L'objectif étant que ce processeur soit en mesure d'exécuter totalement un programme écrit en assembleur ARMv2a. Pour ce faire on utilise le langage de description matérielle VHDL.
Voici le git utilisé pour notre projet : https://github.com/lovisXII/ARM_CPU.
Le CPU que l'on modélise est un processeur pipeliné sur 4 étages :

- Fetch : récupère l'instruction en mémoire et l'envoie à l'étage decod
- Decod : récupère l'instruction chargé par Fetch et procède à son décodage afin de sélectionner les opérandes, registres et calculs nécessaires à son exécution.
  Decod contient le banc de registre.
- Exe : effectue les opérations arithmétiques de bases
- Mem : effectue des accès mémoires si l'instruction exécutée en nécessite.

Voici un schéma simplifié du pipeline que l'on va modéliser :

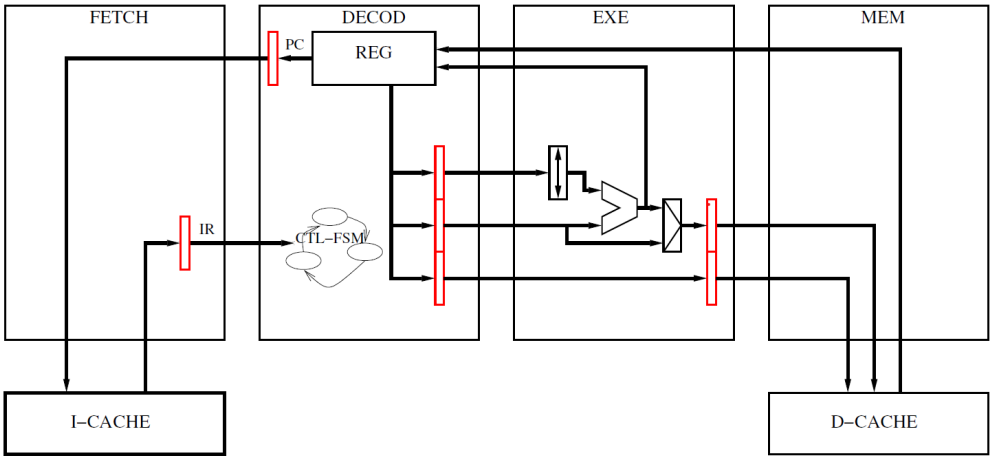

Figure 1: Schéma simplifié du pipeline

Pour la modélisation, nous avons utilisé le langage de description matérielle VHDL et pour la synthèse les outils de la suite Alliance.

2 EXE

2.1 Introduction :

La première étape de notre modélisation a été l'étage EXE, c'est en effet le plus simple à concevoir. Il est constitué de 2 parties primordiales : l'ALU (*arithemic logic unit*) et le shifter. Voici un schéma de son architecture :

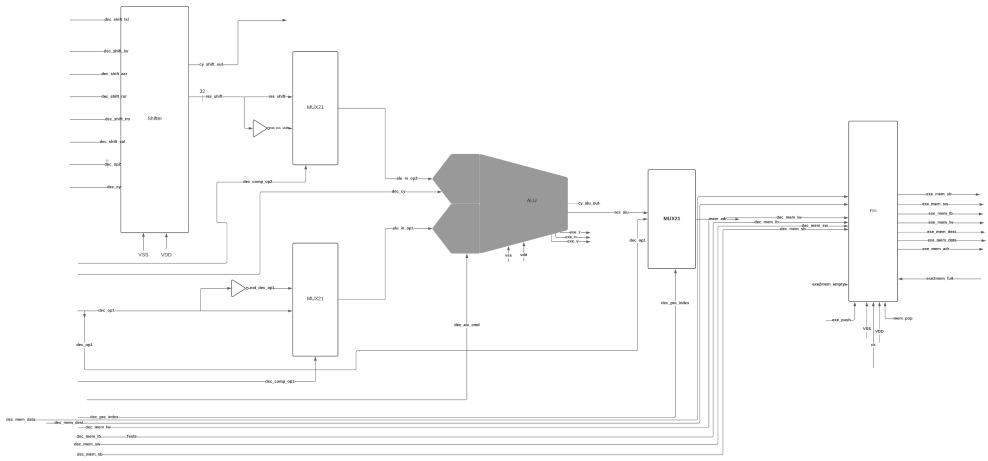

Figure 2: Étage EXE

2.2 ALU : Arithmetic logic Unit

La modélisation de cette unité repose sur l'envoie d'un signal de commande sur 2 bits sélectionnant l'opération à exécuter : and, or, xor, add.
Pour effectuer ces opérations, nous avons utilisé les fonctions logiques fournis par le VHDL et une conversion d'entier signé afin de pouvoir utiliser l'opérateur "+". Dans l'architecture globale de EXE nous avons ajouté des inverseurs commandés afin d'être en mesure de faire le complément à 2 du signal et ainsi d'effectuer des soustractions.

2.3 Shifter

Le shifter est quant à lui commandé par 4 bits indiquant le type de shift que l'on fait et 5 bits indiquant la valeur de shift.
Un shifter permet de faire des multiplications et des divisions par des puissances de 2. Un shift de 1 vers la droite est par exemple une division par 2 tandis qu'un shift de 1 vers la droite est une multiplication par 2.
Étant donné que l'on peut manipuler des entiers signés et non signés, on a besoin de shift conservant le signe du nombre calculé, c'est pourquoi un shift

arithmétique est également nécessaire.
Pour réaliser notre shifter nous avons donc créé 3 entités : un shifter right, un shifter left et un shifter ror.
Pour chacun des types de shifter que nous avons créé le raisonnement est le suivant :
Si le bit n de la valeur de shift est à 1 cela signifie que l'on fait un décalage de 2^n.
Nous avons donc créé des entités effectuant des shits de 1,2,4,8 et 16 que l'on sélectionne en fonction de la valeur de shift que l'on veut.
En combinant ces shifters, on peut shift n'importe quelle valeur entre 0 et 31.

2.3.1 Shifter logique et arithmétique :

Cette entité va permettre de gérer les shifts right logique et arithmétique à l'aide d'une commande que l'on envoie en entrée de l'entité.
Les shifts sont simplement gérés via des concaténations. Par exemple pour un shift right logique de 1 on va récupérer le bit 0 que l'on stocke dans la carry et on décale tous les bits vers la droite en concaténant à gauche avec un zéro.
Pour les shifts right arithmétiques la logique est la même, mais on garde le bit 31 à la même position afin de conserver le signe.

2.3.2 Ror

Pour la rotation, on effectue une concaténation entre les n bits (n étant la valeur du shift value) de poids faibles et les bits restant, les bits restant étant concaténé à droite des bits de poids faible.
Notre entité ror permet également de gérer les shifts rrx, qui est encodée comme un ror de valeur 0. La gestion de cette opération est différente, et donc on l'implémente directement dans l'entité ror.

3 DECOD :

DECOD est l'étage le plus important et le plus complexe du processeur, en effet ce dernier doit gérer le décodage des instructions en provenance de IFETCH mais il doit également gérer l'ensemble du pipeline en commandant les différents étages après avoir décodé l'instruction reçue.
L'étage DECOD est composé de deux parties principales : d'un côté, le banc de registre et de l'autre, le décodage des instructions et le contrôle général du processeur.

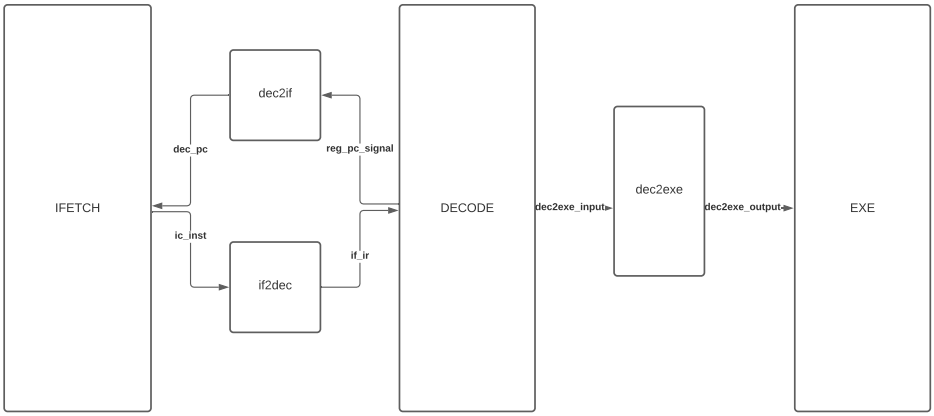

Figure 3: Schéma global des étages IFETCH, DECOD et EXE

3.1 REG : Registre Bank

Sur cette architecture ARM, le banc de registre fait partie de l'étage DECOD.

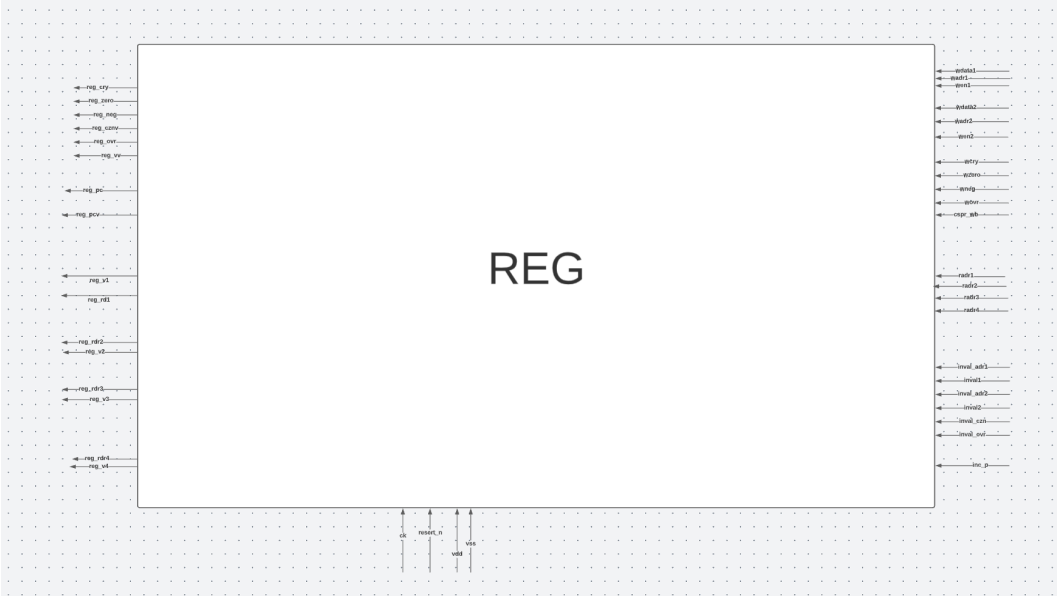

Figure 4: Banc de registre

Il contient 16 registres, ainsi que les flags C, Z, N et V. Le banc de registres peut gérer deux écritures (dont une prioritaire sur l'autre en cas d'adresse identique), 3 lectures 32-bits et une lecture 5-bits servant uniquement pour la lecture du registre codant la valeur d'un shift.
À chaque registre est associé un bit de validité. Un registre est invalidé par DECOD quand une instruction qui écrit dedans est lancée et redevient valide quand le résultat de l'instruction est écrit dans le registre.
Les 4 flags partagent un bit de validité.
En pratique, l'étage REG reçoit les registres à invalider en entrée et chaque registre redevient valide dès qu'une écriture est effectuée dessus. Les lectures sont accompagnées du bit de validité du registre correspondant.
En VHDL, notre banc de registre est représenté par un tableau de STD_LOGIC_VECTOR, et les bits de validité par un tableau de bits. Lors d'une lecture ou d'une écriture, les adresses sont converties en entiers pour être utilisées comme index pour les tableaux.
L'écriture dans les registres se fait à chaque front montant de l'horloge, en donnant bien priorité au premier registre écrit s'il y a un conflit d'adresse.

REG s'occupe aussi de gérer PC : si un flag de contrôle vaut 1, PC est incrémenté de 4, et sinon il garde sa valeur. On réalise la synthèse à l'aide des outils de la suite alliance et on affiche le résultat à l'aide de xsch. Voici un exemple de ce que l'on obtient lors de la synthèse de REG :

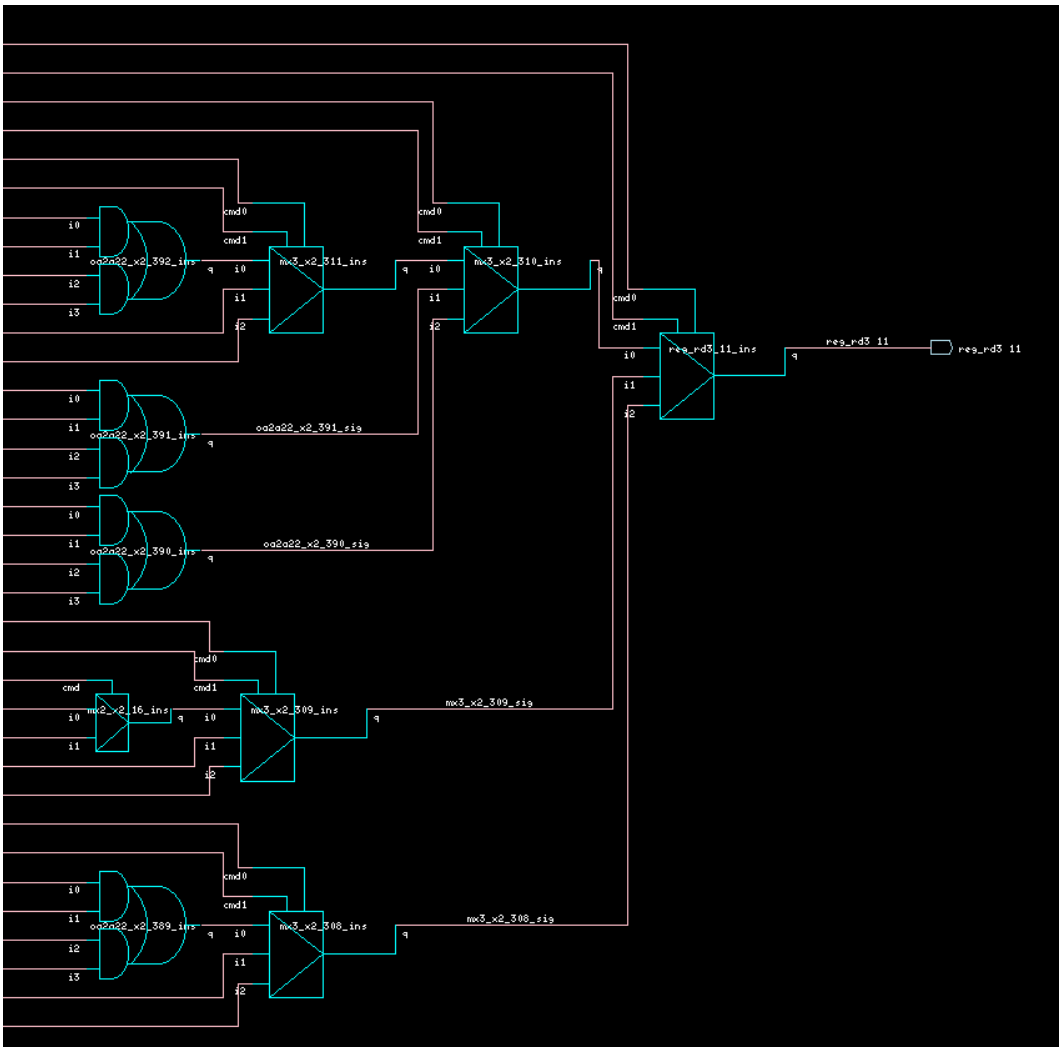

Figure 5: Lecture d'un bit dans un registre

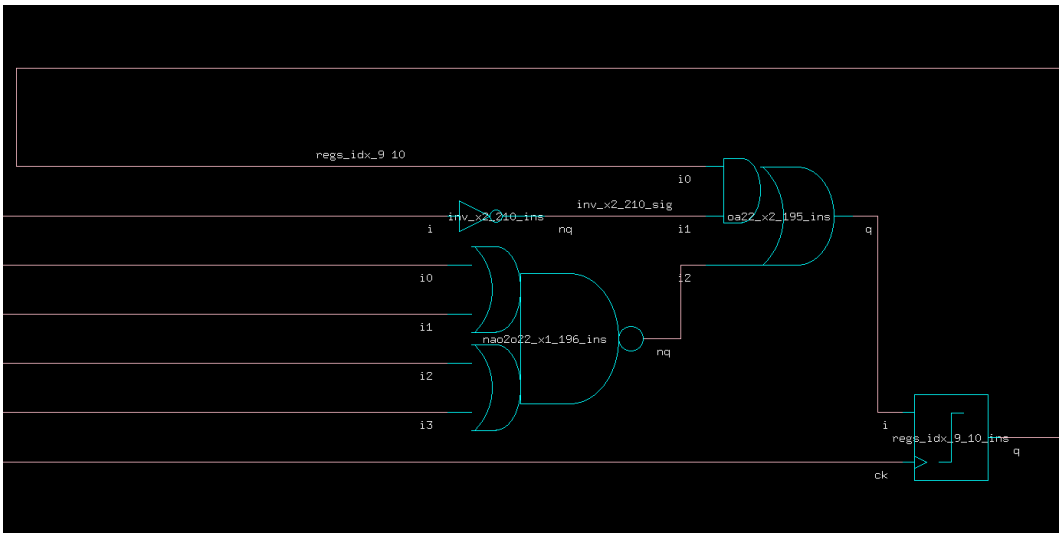

Figure 6: Écriture d'un bit dans un registre

3.1.1 Gestion de PC :

Le PC (program counter) est le registre qui contient l'adresse mémoire de l'instruction prochainement exécutée. Les adresses sont sur 32 bits et sont alignées. Ainsi lorsque l'on passe à l'instruction suivante, il convient d'incrémenter PC de 4.

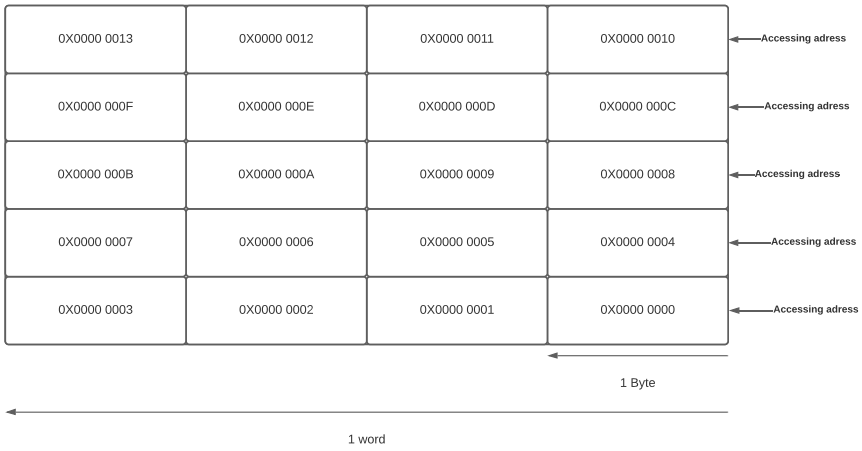

Figure 7: Structure de la mémoire

Nous avons choisi d'implémenter dans notre design l'incrémentation de PC directement dans le banc de registre.
Pour gérer l'incrémentation de PC nous utilisons un signal *inc_pc* indiquant si l'incrémentation $PC + 4$ doit avoir lieu ou non. Lorsque l'on détecte un branchement ou un link il faut stopper l'incrémentation de 4. Dans ce cas, on va charger la valeur de PC dans l'opérande 1 et la valeur de l'offset du branchement dans l'opérande 2. On enverra ensuite ces deux valeurs à EXE pour que l'alu puisse les additionner.
On stoppe aussi l'incrémentation de PC quand le pipeline est congestionné, ou quand on a besoins d'un cycle de gel.

3.2 DECOD

3.2.1 Machine à état :

Pour que decod fonctionne correctement, nous avons besoin d'utiliser une machine à état. Il s'agit ici d'une machine de Mealy.

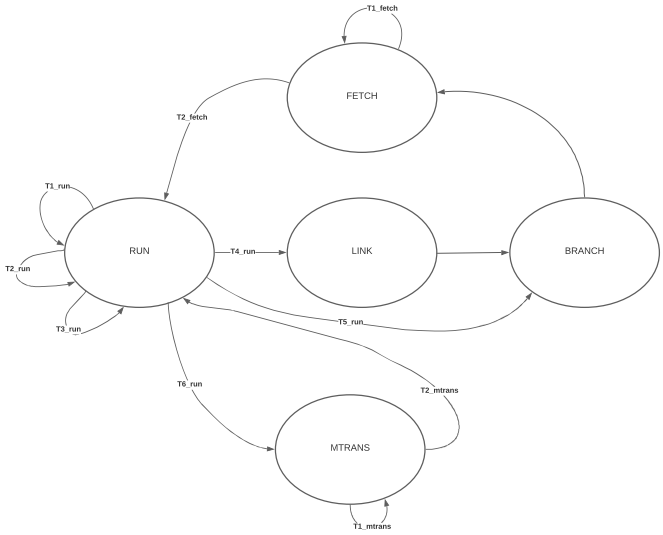

Figure 8: Machine à état

Les transitions permettant le passage d'un état à un autre sont recensés dans le tableau ci-dessous.
Initialement, nous n'avions pas implémenté les transferts multiple et notre machine à état les ignorait (comme une instruction dont le prédicat serait faux). Mais nous avons fini par ajouter les transferts multiples, et donc ils sont présents dans ce tableau.

Etat	Transition	Action
FETCH	T1_fetch	Chargement d'une nouvelle instruction
	T2_fetch	Fifo if2dec est pleine et contient donc une instruction, passage à run
RUN	T1_run	Envoie d'une nouvelle valeur de PC à IFETCH
	T2_run	Prédicat faux, l'instruction est jetée
	T3_run	La condition est valide, exécution de l'instruction
	T4_run	L'instruction est un appelle de fonction, passage à LINK
	T5_run	L'instruction est un branchement, passage à branch
	T6_run	L'instruction est un transfert multiple, passage à MTRANS
LINK	NA	Sauvegarde de R15 dans R14
BRANCH	NA	Purge de l'instruction suivant un branchement pris
MTRANS	T1_mtrans	On reste dans l'état MTRANS pour continuer les accès mémoires tant que l'on a pas lu/écrit tous les registres
	T2_mtrans	Tous les registres ont été lus, passage à RUN

3.3 Décodage des instructions :

Les instructions reçues par DECOD sont des binaires 32 bits dont il faut décoder le sens. Pour ce faire on cherche à identifier le type d'opération à effectuer. Nous avons défini 4 types d'opérations listées ci-dessous.

Pour décoder l'instruction en provenance de IFETCH nous nous sommes aidés de la documentation ARM fournie lors du CM 3.
Dans tous les cas, les 4 bits de poids fort de l'instruction désignent le prédicat, c'est-à-dire la condition d'exécution de l'instruction. L'ensemble des prédicats possible est recensé dans le tableau suivant :

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 9: Prédicats

3.3.1 Regop_t : Regular operation :

La structure d'une instruction régulière est la suivante :

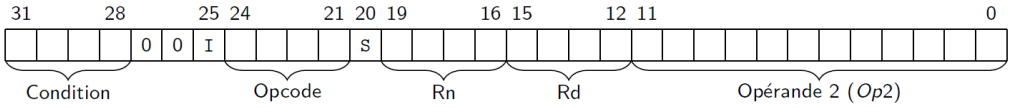

Figure 10: regop_t encodage

La condition d'exécution de l'instruction est celle définition par la liste fournie sur la figure 9. Les opcodes sont quant à eux définis sur la figure suivante 11 :

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 11: regop_t Opcode

L'opération que l'on effectue va nous donner énormément d'information comme par exemple si l'on souhaite faire le complément à deux de l'une de nos deux opérandes dans le but de faire une soustraction par exemple.
Le bit 20, nous indique si les flags qui vont être calculé par EXE doivent être Write Back ou non.
Les instructions tst, teq, cmp et cmn doivent invalider le registre destination, en effet ces opérations n'écrivent pas dans les registres, elles ne font que positionner les flags pour différentes opérations.
Les bits 15 à 12 indique le registre d'écriture destination, les bits 19 à 16 indique quant à eux le registre source et enfin les bits 11 à 0 codent l'opérande

2.
Cette dernière se décode de différentes manières selon la valeur du bit 25 qui indique si on considère un opérande de type immédiat ou non.
Deux cas vont alors se présenter, celui où I vaut 1 et celui où il vaut 0. Dans le cas où l'opérande 2 est de type immédiat l'encodage sera le suivant :

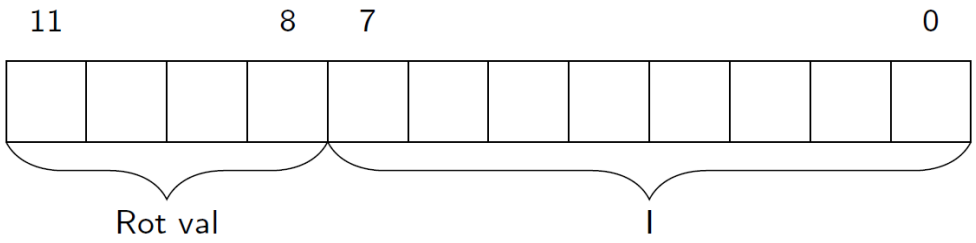

Figure 12: Décodage Op2 dans le cas où I = 1

Les 8 bits de poids faible désignent l'opérande que l'on souhaite utiliser. Cet opérande n'étant que sur 2 bits on utilise les 4 bits suivant pour faire une rotation de cette dernière afin de pouvoir coder des entiers supérieurs à 2$\hat{8}$-1.

Dans le cas où l'opérande n'est pas de type immédiat les 4 bits de poids fort désignent un 2 registre de lecture tandis que les 8 bits suivants vont avoir deux codages différents en fonction de la valeur du bit 4 :

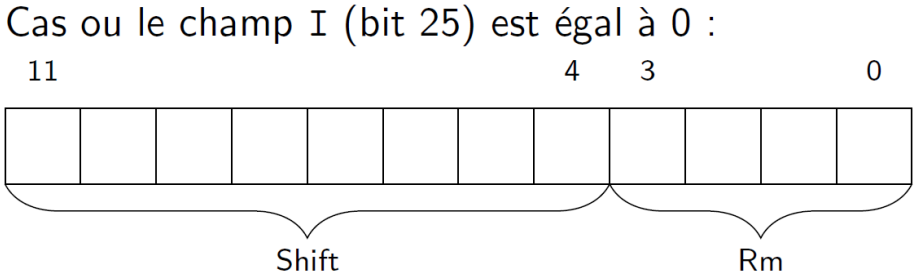

Figure 13: Codage des 12 bits de poids faible d'une regop dans le cas où I = 0

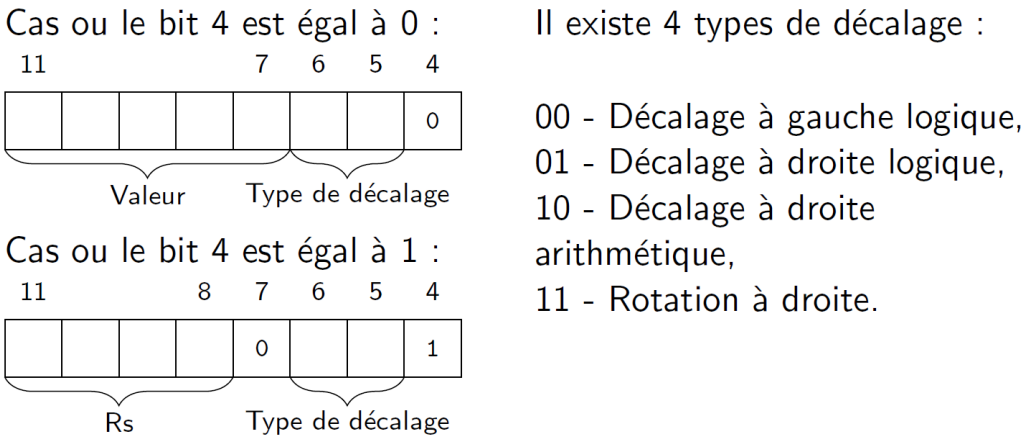

Figure 14: Deux codages différents selon la valeur du bit 4

Dans le premier cas la valeur du décalage est un entier tandis que dans le deuxième cas, on va chercher la valeur de décalage directement dans un registre en ne conservant que ses 5 bits de poids faible.

3.3.2 Branch_t : branchement operation :

Les branchements sont des opérations élémentaires permettant entre autre l'exécution de boucle, dans le cas de l'architecture que nous avons réalisé il existe deux types de branchement : les links et les branchements "classiques". En ARMv2a l'exécution des branchements se fait en suffixant l'instruction d'un des prédicats explicitant dans quel cas le branchement est réalisé.

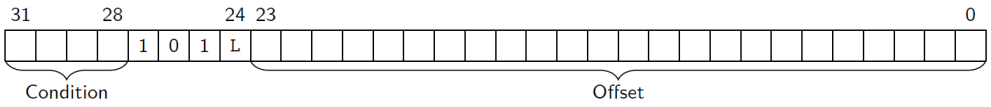

Figure 15: Codage d'une instruction de branchement

Deux cas se présente lorsque l'on effectue un branchement : si la condition

d'exécution n'est pas satisfaite l'instruction est jetée et on continue l'exécution du programme en séquentiel. Si elle réussit, on doit passer inc_pc à 0 afin d'arrêter d'incrémenter PC par 4 et ainsi de calculer la valeur de PC + offset. En ARM V2 la valeur ajoutée à PC est en réalité PC = PC +8 + (OFFSET *4).
Lorsque l'on fait un link on doit d'abord passer dans l'état LINK afin de sauvegarder la valeur de PC dans le registre 14.
Si l'on ne fait pas de link on passe directement dans l'état BRANCH.
Dans les deux cas nous avons ajouté un signal if_flush qui nous permet de vider la fifo if2dec et ainsi d'ignorer l'instruction qui avait séquentiellement été chargée pendant le décodage du branchement.
Enfin on charge la valeur de PC dans l'opérande 1 et la valeur de l'offset concaténé de 2 à droite et de 8 à gauche dans l'opérande 2, la concaténation à gauche permettant de l'étendre sur 32 bits et celle à droite de le multiplier par 4, et on envoie comme commande à l'alu une addition.

3.3.3 Trans_t : opérations de transfert

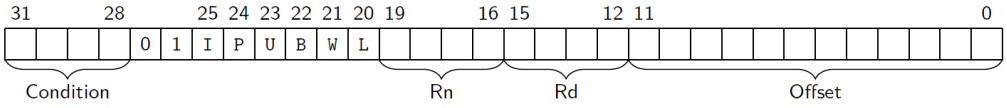

Figure 16: Codage d'une instruction de transfert simple

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 17: Détail de la signification des bits des instructions de trans_t

Les instructions de transfert simples permettent d'écrire dans des registres ou de lire dedans. Ces accès sont faisables soit par octet, soit par mot.
Dans le cas d'une écriture nous avons veillé à désactiver l'invalidation du registre Rd, en effet ce dernier étant uniquement lu il n'est pas nécessaire de l'invalider.
Le codage de l'offset est identique à celui de l'opérande 2 dans les instructions de type regop_t à la différence près que dans le cas où on l'on est dans une instruction de type immédiat il n'y a pas de rotation de la valeur comme c'était le cas pour les instructions regop. La plus grande valeur possible pour l'offset est donc $2^{12} - 1$.

3.3.4 Transferts multiples :

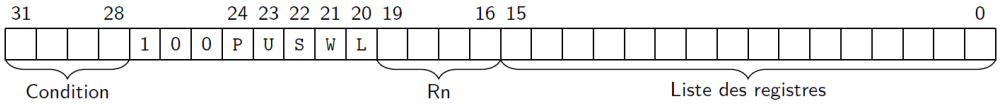

Figure 18: Détail de la signification des bits des instructions de transferts multiples

Le jeu d'instruction ARM permet de faire des transferts multiples c.-à-d. qu'en une seule instruction, nous allons être en mesure d'écrire ou de lire dans au plus 16 registres.
Pour réaliser ces instructions nous avons extrait les bits 15 à 0 dans un signal interne register_list nous permettant d'accéder à un registre par cycle.
Une fois le transfert multiple détecté par DECOD on passe dans l'état MTRANS et on y reste tant que tous les registres n'ont pas été accédés.
Pour savoir si tous les registres ont été accéder nous avons procédé comme suit :
À chaque cycle, on accède à un registre, on regarde donc notre signal register_list stockant les bits 15 à 0 et on cherche le premier 1 dans cette suite de bits.

Une fois que l'on rencontre un 1 on va lire ou écrire dans le registre correspondant et l'on passe à 0 le bit de register_list que l'on vient de lire, et on continue comme ça jusqu'à ce que register_list soit totalement nul ce qui signifie que l'on a accédé à tous les registres souhaités.
Une qu'on a accédé à tous les registres, on retourne dans RUN pour exécuter les instructions suivantes.
Dans la machine à état proposé dans les TME les MTRANS possédait une transition vers BRANCH supposément dans le cas où l'instruction de transfert multiple écrivait dans r15, nous n'avons pas implémenté cette possibilité dans notre architecture.

4 Protocole de test :

Chacun des étages du pipeline du processeur ayant été réalisé nous avons cherché à vérifier que tout fonctionnait correctement. Pour ce faire nous avions testé individuellement chaque étage après sa conception afin de vérifier que ces étages était fonctionnel.
Pour ce faire nous avions réalisé une suite de test envoyant des nombres aléatoires dans les entrées de nos entités et nous vérifions si les sorties étaient cohérentes.
Typiquement pour l'alu, nous envoyions deux opérandes aléatoires avec une commande aléatoire et l'on vérifiait que tout fonctionnait.
Mais une fois tous les étages réalisé il nous a fallu tout tester ensemble pour vérifier qu'il communiquait bien tous entre et surtout que DECOD effectuait son travail correctement.

4.1 Simulation complète à l'aide de core.c :

Une fois DECOD fini nous avons mappé l'ensemble de nos fichiers dans le core. Le problème étant que notre processeur n'a pas physiquement accès à

une mémoire, nous avons donc dû la simuler avec une interface vers du C .
Le principe étant que notre suite de test VHDL puisse aller écrire et lire en mémoire, et ce, à l'aide de fonctions externe définies en c.
La documentation de GHDL décrit assez précisément comment appeler une fonction C depuis VHDL.
Pour pouvoir simuler une mémoire de 4go (à savoir l'ensemble de l'espace adressable), sans réellement allouer une telle quantité de mémoire, on utilise un tableau à quatre niveaux (de type int ****), donc les blocs de mémoire sont alloués de manière fainéante lors des écritures (si on essaye d'accéder en lecture à un bloc non alloué, on peut simplement retourner 0 en supposant la mémoire initialisée).
Le programme C prend en argument un fichier texte contenant les instructions en hexadécimal (en toute lettre, par exemple "e0000000"). Le fichier est lu à l'initialisation et les entiers correspondant aux instructions sont placés dans la mémoire. Nous avons ensuite défini trois fonctions : get_inst, get_mem et write_mem, la première récupérant une instruction en mémoire, la deuxième lisant la mémoire et enfin la dernière permettant d'écrire en mémoire.
On ignore les bits de poids faible de toutes les adresses (en les shiftant simplement vers la droite).

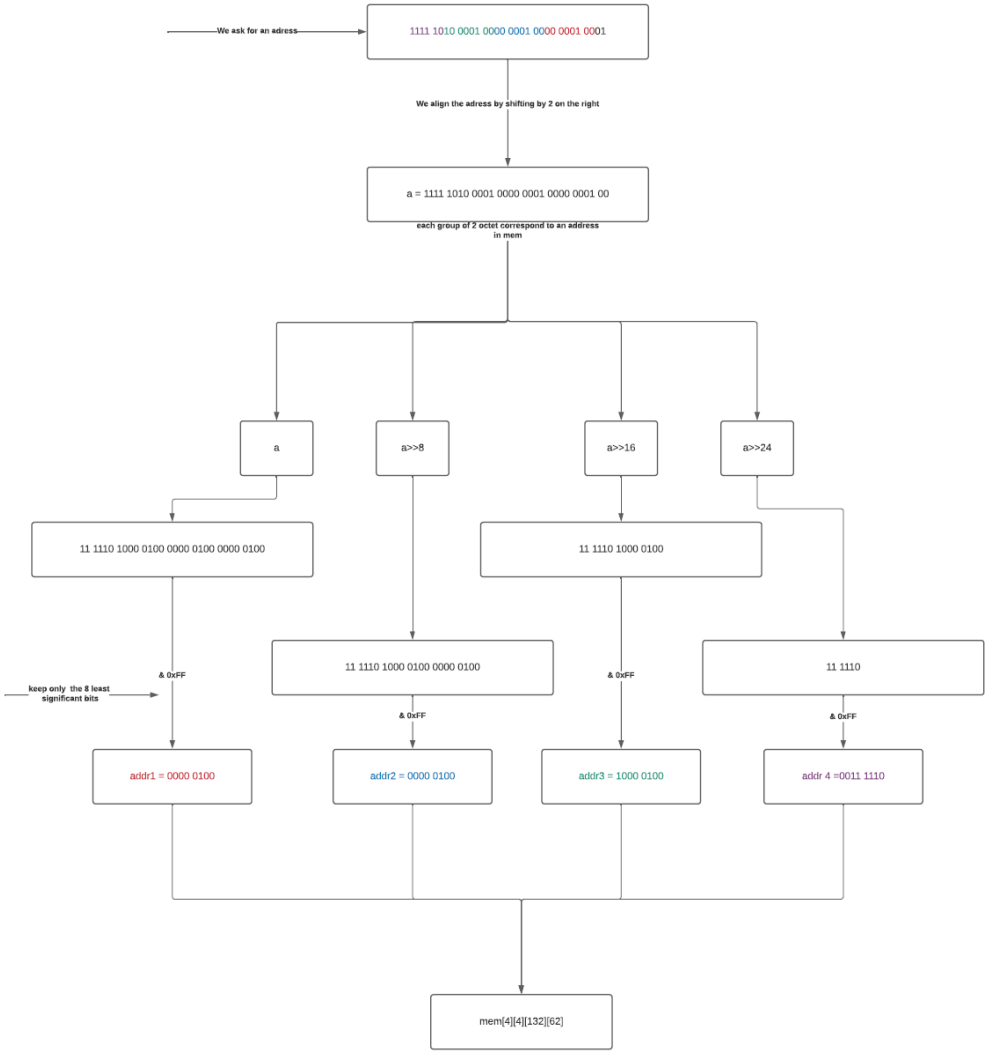

Figure 19: Détail de la procédure d'alignement mémoire avec accès mémoire

À l'aide de la figure 26 on constate bien que chaque bit de l'adresse permet de coder l'élément de la mémoire auquel on veut accéder.
Ainsi on va dans un premier temps charger toutes nos instructions en mémoires. Nous allons ensuite les envoyer dans IFETCH à l'aide de get_inst.

Cette plateforme a ensuite été remplacé par la plateforme fournie sur moodle.

Dans les deux cas, cela nous permettait d'exécuter des instructions assembleur et de suivre les signaux avec Gtkwave, et ainsi de corriger de nombreux bugs.

4.2 Test de programme C :

Par la suite, nous avons écrit quelques petits tests en langage C, que nous avons compilé en assembleur ARMv2a puis en un exécutable lisible par le processeur.
Le C devait contenir quelques instructions assembleurs (insérées avec la fonction __ASM__), pour l'initialisation de la pile ainsi que la détection du résultat.
De plus, l'assembleur compilé nécessite quelques modifications mineures : pour une raison qui nous est inconnue, les indications générées par GCC sur les fonctions (les directives commençant par @) empêchent AS de compiler les sauts vers ces fonctions correctement, nous les avons donc supprimés.
De plus, malgré la compilation avec -NOSTDLIB, un appel à memset a été généré, nous l'avons donc supprimé également.
Moyennant ces petites modifications, nous avons pu exécuter deux programmes calculant la suite de Fibonacci, l'un utilisant un tableau et l'autre récursif. En particulier, la récursivité teste très bien les instructions mémoire push et pop, qui sont sans cesse nécessaires pour gérer les différents appels de fonction et retours.

5 Résultat de synthèse :

Afin de réaliser la synthèse de notre processeur, nous avons utilisé les outils de la suite Alliance.

5.1 Vasy, Boom, Boog et loon :

Pour réaliser la synthèse de nos modèles, nous avons utilisé Vasy, Boom, Boog et Loon, 4 programmes permettant de convertir le VHDL vers une représentation réduite en .vst, et d'optimiser le nombre de portes logique

nécessaire à la réalisation de notre circuit, tout en assurant un chemin critique minimal.

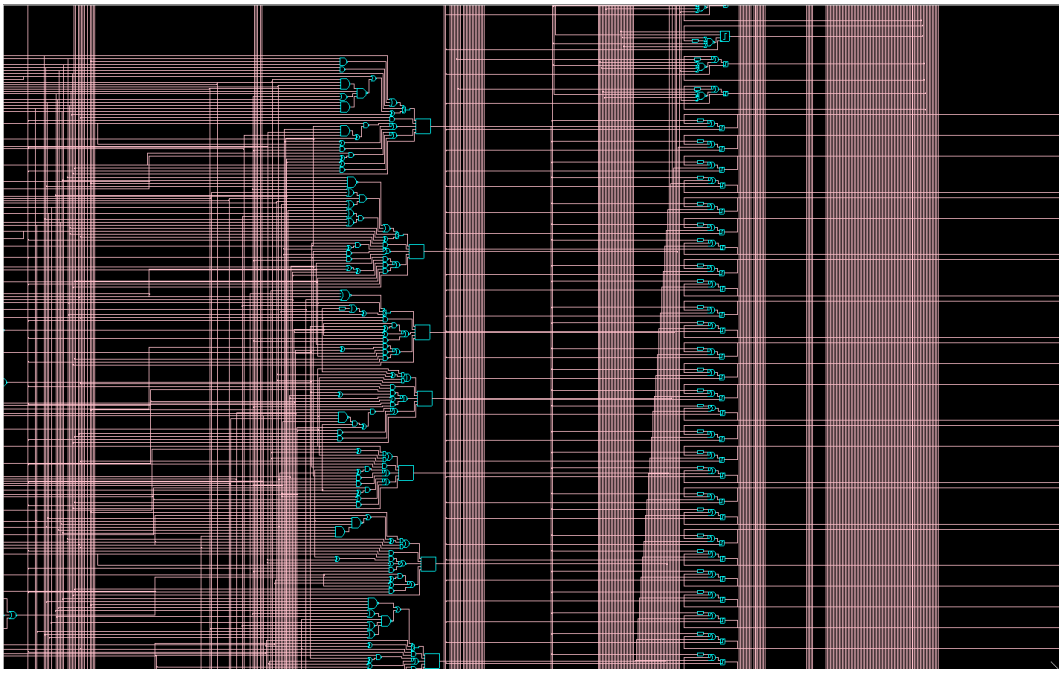

Figure 20: Résultat de synthèse de REG

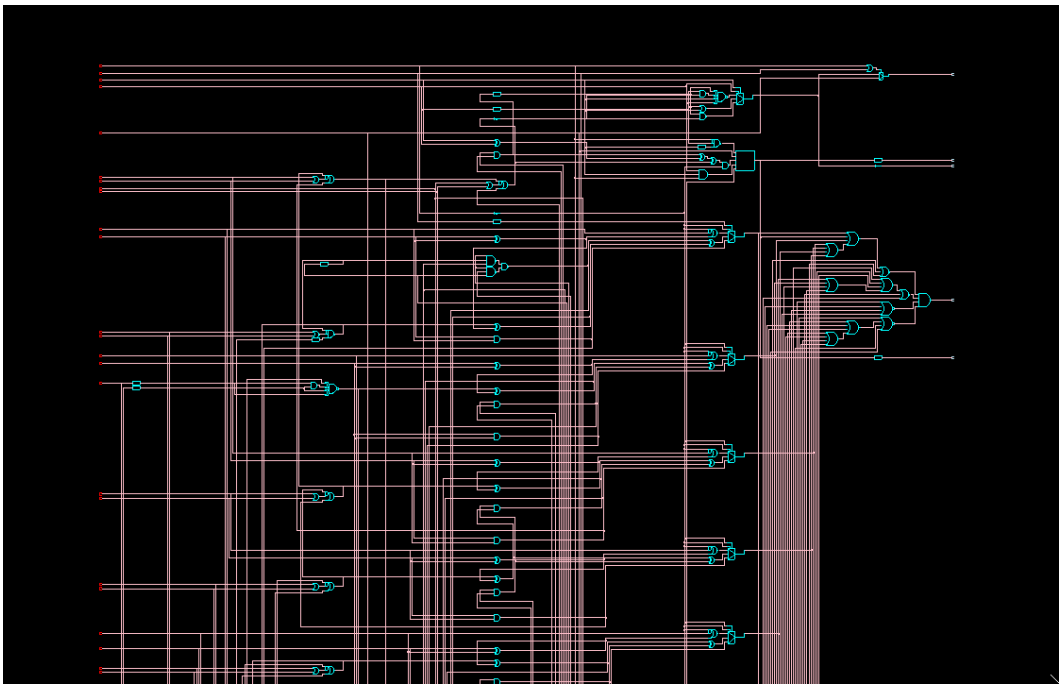

Figure 21: Résultat de synthèse de l'ALU

Les résultats de synthèse avec Loon, la dernière étape de la synthèse, sont les suivants :

Étage	Nombre de portes logiques	chemin critique
Ifetch	176	3288 ps
Decod	11043	15730 ps
Exe	2081	24839 ps
Mem	200	1388 ps
Total du core	14930	27253 ps

Table 1: Résultat des différents étages

5.2 Placement routage avec cgt :

Une fois les fichiers .vst généré par loon récupérer nous avons réalisé le placement routage à l'aide de Cgt de la suite Coriolis.
Les différents résultats obtenus sont présentés ci-dessous :

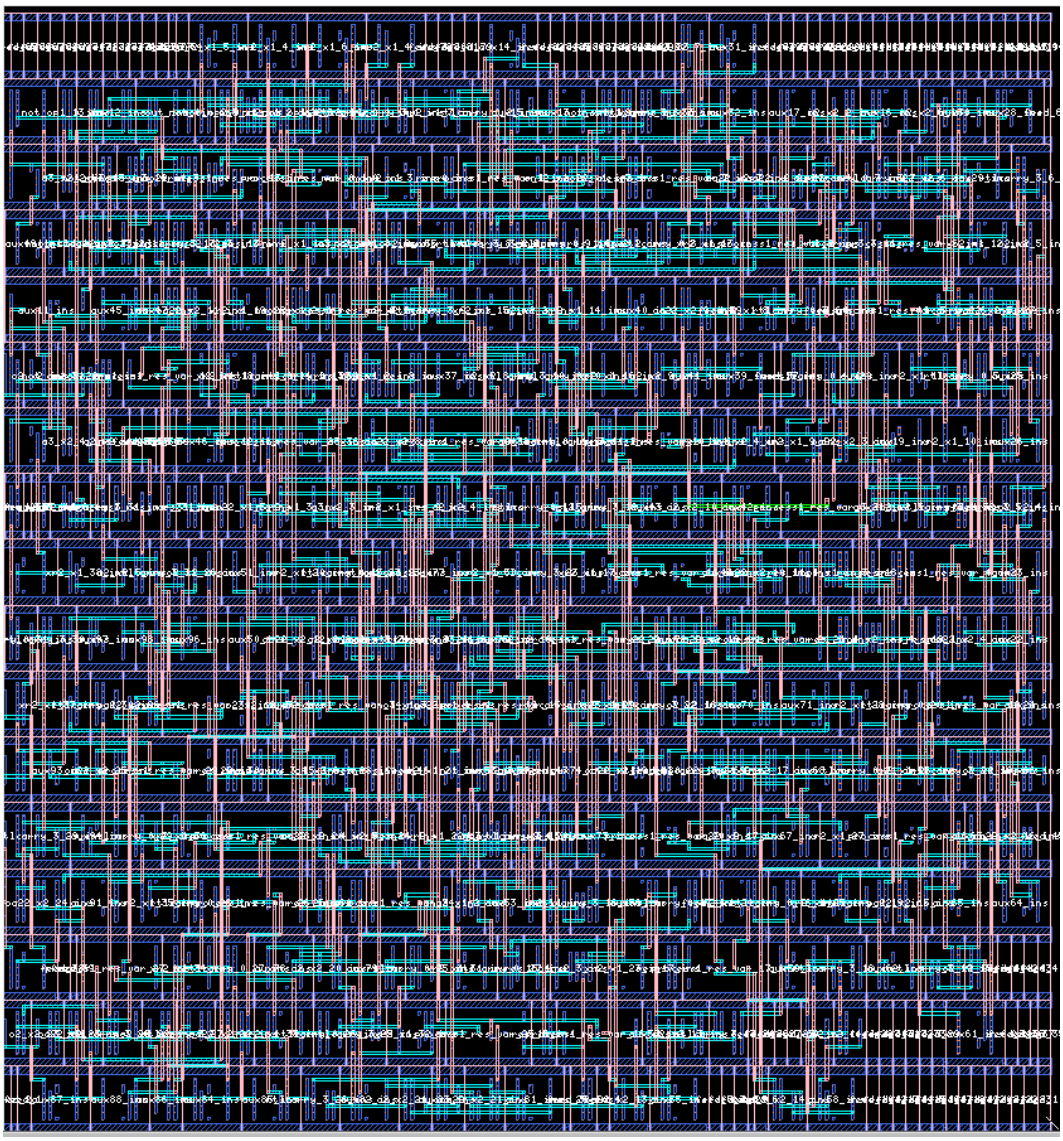

Figure 22: Placement routage de l'ALU

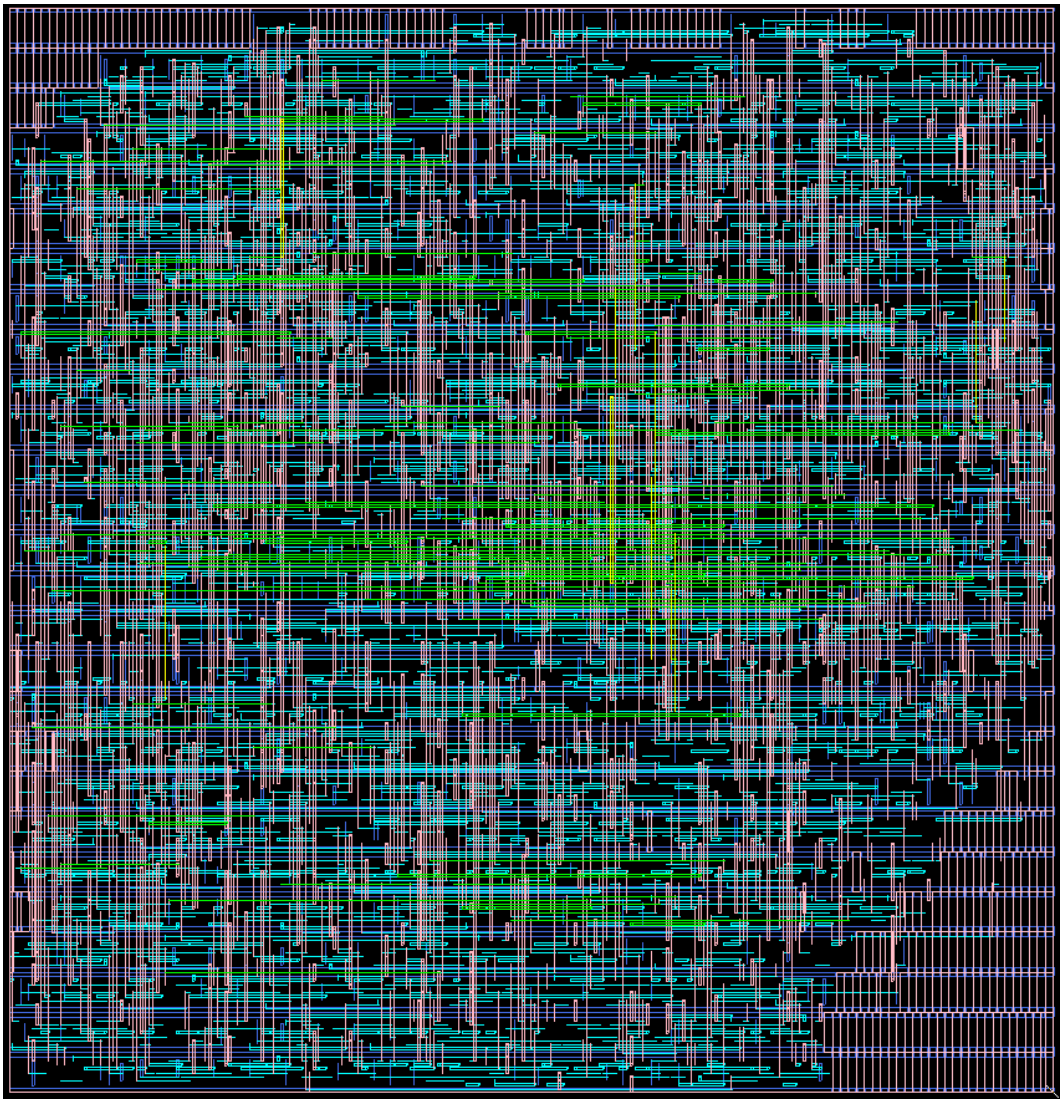

Figure 23: Placement routage du shifter

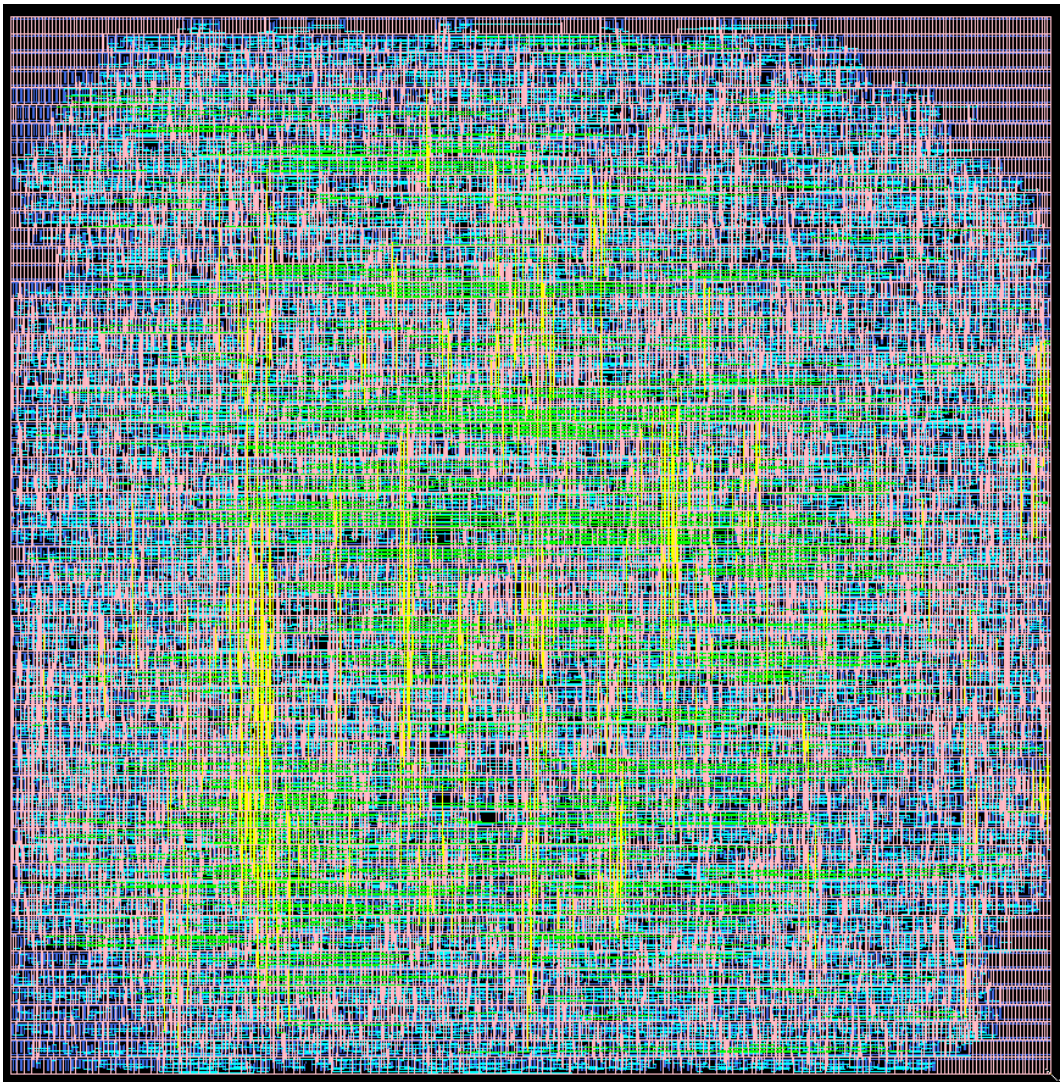

Figure 24: Placement routage de decod

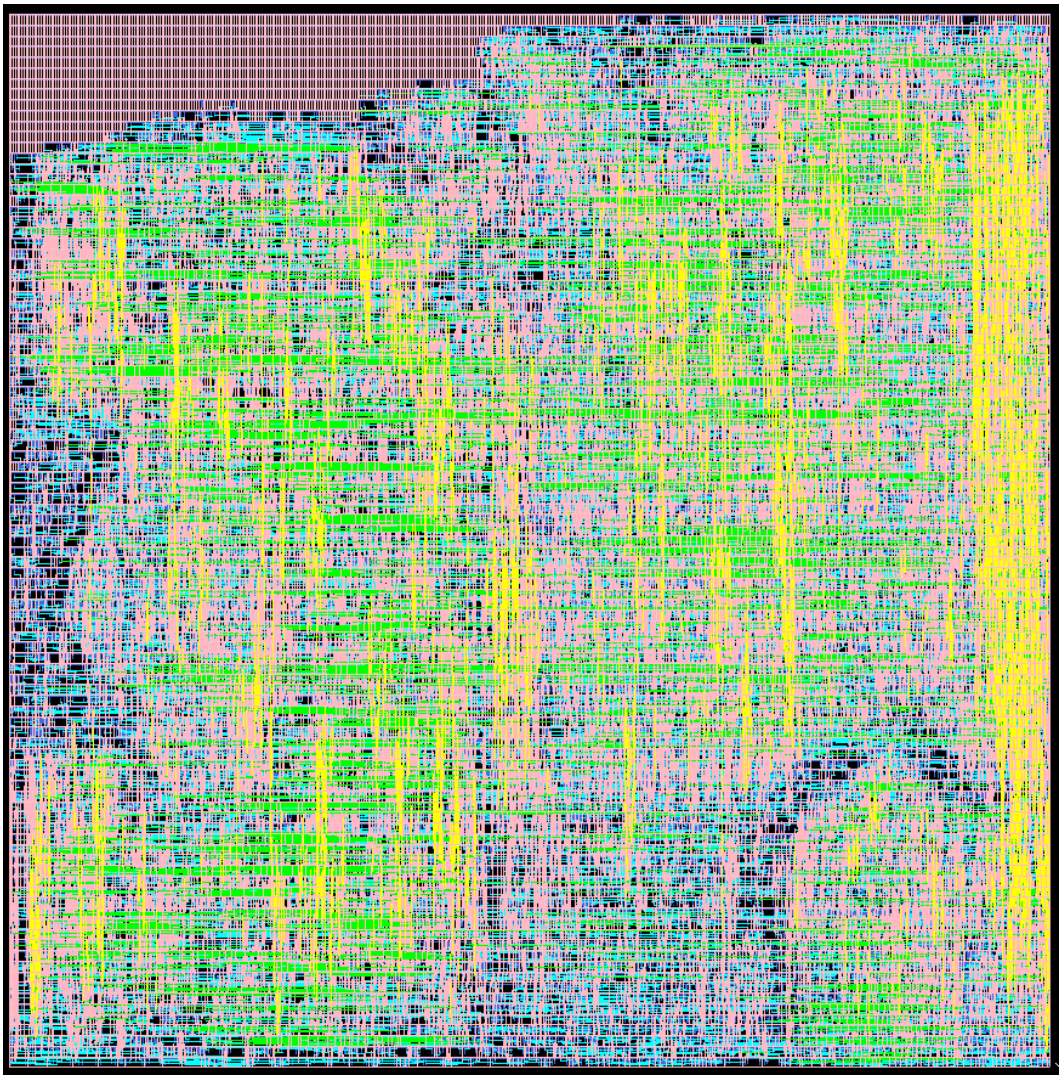

Figure 25: Placement routage du core

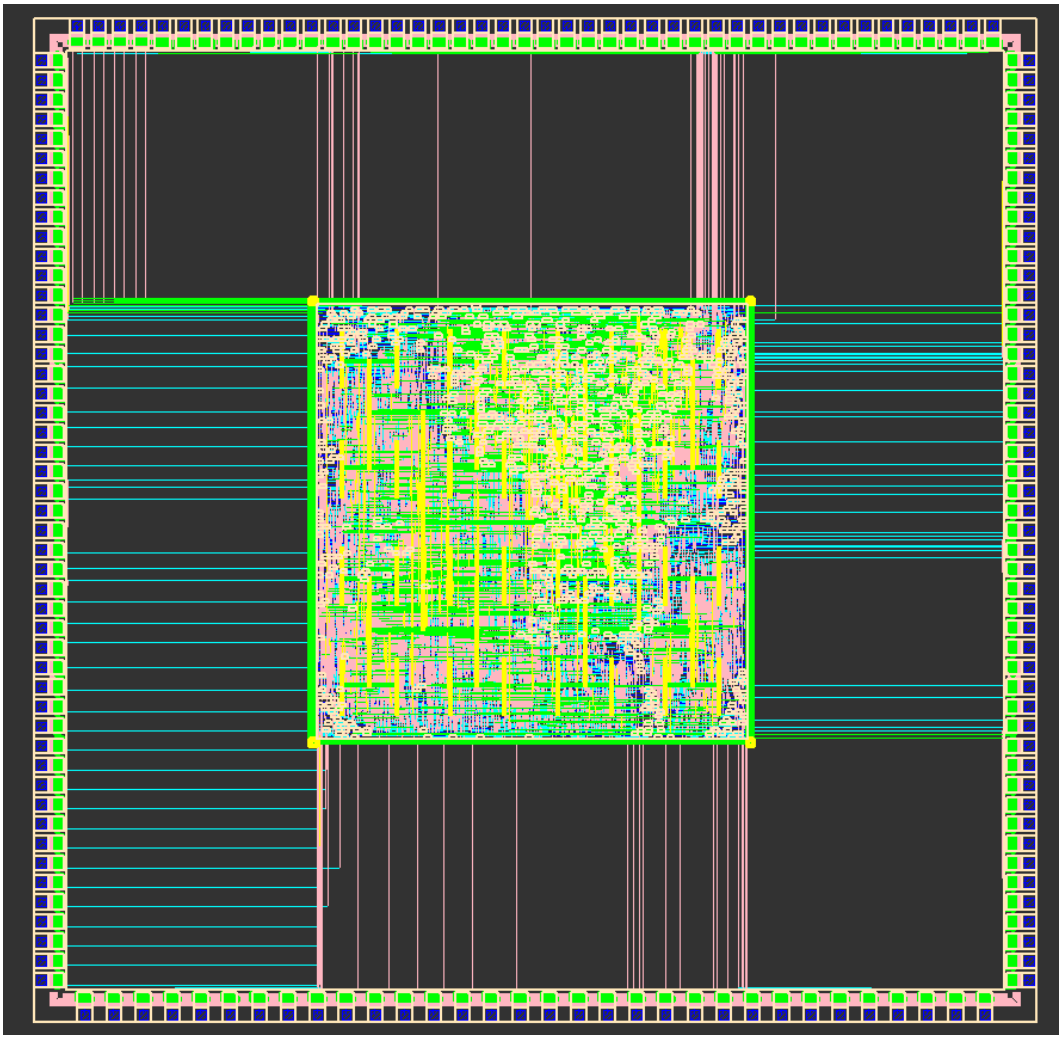

Figure 26: Placement routage du chip

6 Conclusion

Les résultats obtenus nous permettent de constater que notre processeur est plot-limited, de plus on remarque que nos plots sont très espacés. Pour résoudre ce problème, il faudrait que l'on modifie le script python utilisé pour le placement routage.
De plus afin d'améliorer notre architecture il nous reste à implémenter des by-

pass dans le but d'éviter les cycles de gels entre deux instructions présentant des dépendances de données.
Enfin une amélioration possible de l'architecture serait de passer le processeur en superscalaire en s'inspirant du SS2 présenté par Mr. Pirouz Bazargan dans le cadre de l'UE architecture des processeurs RISC.