Sorbonne Université - M1 SESI - VLSI

RÉALISATION D'UN PROCESSEUR ARM

Rapport de fin de projet - janvier 2024

Riad SAHKI
Hakim IZM

Table des matières

1 Introduction

Le but de ce projet est de modéliser un processeur ARM à l'aide du langage de description matérielle VHDL. Notre processeur est basé sur un pipeline composé de quatre étages : **IFETCH**, **DECOD**, **EXE** et **MEM**.

Il nous est demandé de faire les étages **DECOD** et **EXE**, les autres étages nous sont fournis.

Tout au long de notre projet, nous avons utilisé trois outils : l'éditeur de code Visual Studio Code avec l'extension VHDL pour le développement, le compilateur open-source GHDL pour l'analyse de notre code VHDL ainsi que la simulation de nos testbenches.

Ce projet réalisé en binôme par Riad SAHKI et Hakim IZM a été encadré par Jean-Lou DESBARBIEUX dans le cadre de l'UE de VLSI sur toute la durée du premier semestre.

2 Architecture du processeur ARM

2.1 Registres

Notre processeur ARM dispose de 16 registres principaux (nommés de R0 à R15). Les registres R0 à R14 sont des registres normaux que nous utilisons dans toutes les instructions, le registre R15 est celui qui comporte le PC (program counter). C'est grâce au program counter que le processeur sait où il en est dans le fil d'exécution et quelle instruction exécuter.

En plus de ces 16 registres, il en existe un supplémentaire nommé CPSR (Current Program Status Register) dont le rôle est d'enregistrer les drapeaux générés suite à l'exécution de certaines instructions. Ce registre est consulté par le processeur dans le cas où des instructions conditionnelles seraient exécutées.

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15 (PC)
+
CPSR (flags)

TABLE 1 – Les registres du processeur ARM

2.2 Drapeaux

Nous avons précédemment évoqué le fait que des drapeaux étaient enregistrés dans le registre CPSR. Ces drapeaux sont levés par l'ALU suite au résultat de l'opération. Ils sont au nombre de quatre : **N**, **Z**, **C** et **V**.

DRAPEAU	**DESCRIPTION**
N	Le résultat de l'ALU est négatif (bit 31 = 0)
Z	Le résultat de l'ALU est égal à zéro (tous les 32 bits = 0)
C	Retenue générée par l'ALU dans le cas des instructions arithmétiques et le shifter dans le cas des instructions logiques
V	Dépassement de capacité dans le cas d'une opération arithmétique signée (overflow)

TABLE 2 – Drapeaux du registre CPSR

Ces drapeaux sont utilisés dans les instructions conditionnelles que nous évoquerons par la suite.

2.3 Instructions

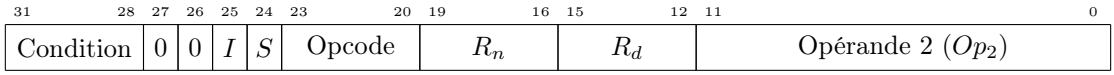
Les instructions en assembleur ARM sont divisées en plusieurs catégories :

— traitements de données ;
— multiplications ;
— swap ;
— branchements ;
— transferts mémoire ;
— coprocesseur ;
— interruptions logicielles.

Toutes ces instructions sont codées sur 32 bits, de manière différente en fonction de leur catégorie.

2.3.1 Instructions de traitement de données

Codage des instructions de traitement de données :



Condition (bits 31 à 28) : Voir la partie 2.3.2 sur les instructions conditionnelles

I (bit 25) : Mis à 1 si l'opérande 2 est un immédiat, à 0 s'il s'agit d'un autre registre.

S (bit 24) : Mis à 1 si les drapeaux doivent être mis à jour.

R_n **(bits 19 à 16) :** Registre correspondant généralement au premier opérande.

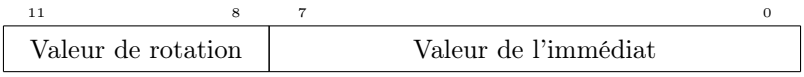
R_d **(bits 15 à 12) :** Registre correspondant généralement à la destination.

Opcode (bits 23 à 20) : Chaque opération est définie sur 4 bits tels que décrits sur le tableau suivant.

OPCODE	**INSTRUCTION ASSEMBLEUR**	**OPÉRATION**
0000	AND	$R_d \Leftarrow R_n$ AND Op_2
0001	EOR	$R_d \Leftarrow R_n$ XOR Op_2
0010	SUB	$R_d \Leftarrow R_n - Op_2$
0011	RSB	$R_d \Leftarrow Op_2 - R_n$
0100	ADD	$R_d \Leftarrow R_n + Op_2$
0101	ADC	$R_d \Leftarrow R_n + Op_2 + C$
0110	SBC	$R_d \Leftarrow R_n$ - $Op_2 + C - 1$
0111	RSC	$R_d \Leftarrow Op_2 - R_n + C - 1$
1000	TST	Positionne les flags pour R_n AND Op_2
1001	TEQ	Positionne les flags pour R_n XOR Op_2
1010	CMP	Positionne les flags pour R_n - Op_2
1011	CMN	Positionne les flags pour R_n + Op_2
1100	ORR	$R_d \Leftarrow R_n$ OR Op_2
1101	MOV	$R_d \Leftarrow Op_2$
1110	BIC	$R_d \Leftarrow R_n$ AND NOT Op_2
1111	MVN	$R_d \Leftarrow$ NOT Op_2

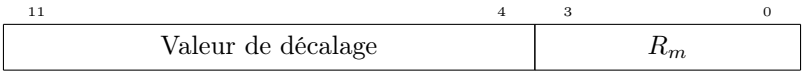
TABLE 3 – Liste des opcodes pour les instructions de traitement de données

Opérande 2 (bits 11 à 0) : Opérande issue d'un immédiat dans le cas où I = 1



La rotation s'effectue sur la valeur de l'immédiat (codé sur 32 bits) vers la droite.

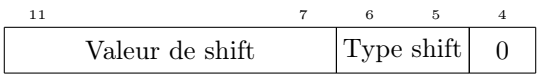
Opérande 2 (bits 11 à 0) : Opérande issue d'un registre dans le cas où I = 0



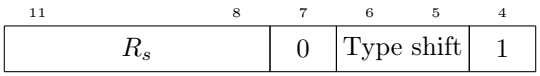
Ici, R_m correspond au registre qui sera utilisé comme seconde opérande et la valeur de décalage sera appliquée à la donnée contenue dans R_m.

Il existe deux manières de faire le décalage : soit en utilisant une valeur en dur sur 5 bits, soit en utilisant un registre R_s. Le cas rencontré est determiné par la valeur du bit n°4.

Cas où le bit 4 est égal à 0 :



Cas où le bit 4 est égal à 1 :



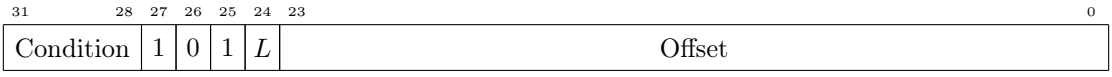
La valeur maximale pour un décalage est de 31 bits ; par conséquent, seuls les 5 bits de poids faible de R_s seront considérés.

Pour les deux cas :

Il existe quatre types de décalage que l'on peut choisir en définissant les bits 6 et 5 :
— 00 : décalage à gauche logique ;
— 01 : décalage à droite logique ;
— 10 : décalage à droite arithmétique ;
— 11 : rotation à droite.

2.3.2 Instructions de branchement

L'instruction assembleur pour les branchement est **B**. Les branchements sont codés de la manière qui suit :



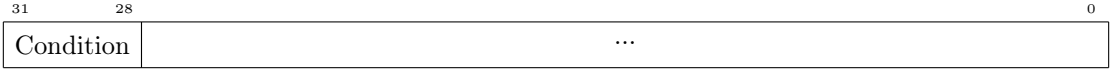
Condition (bits 31 à 28) : Voir la partie 2.3.2 sur les instructions conditionnelles

L (bit 24) : Link (retour à la suite du code après le branchement) si L = 1 : $R_{14} \Leftarrow PC + 4$

Offset (bits 23 à 0) : $PC \Leftarrow PC + 8 + (Offset \times 4)$

2.3.3 Instructions conditionnelles

Pour toutes les instructions ARM, les 4 bits de poids fort définissent le prédicat. Le prédicat correspondont à la condition que les drapeaux doivent réunir afin d'exécuter l'instruction en question. Dans le cas contraire, l'instruction est tout simplement ignorée.



PRÉDICAT	SUFFIXE ASSEMBLEUR	CONDITION VÉRIFIÉE
0000	EQ	$Z = 1$
0001	NE	$Z = 1$
0010	HS / CS	$C = 1$
0011	LO / CC	$C = 0$
0100	MI	$N = 1$
0101	PL	$N =$
0110	VS	$V = 1$
0111	VC	$V = 0$
1000	HI	$C = 1$ et $Z = 0$
1001	LS	$C = 0$ ou $Z = 1$
1010	GE	$\geqslant$ (supérieur ou égal)
1011	LT	$<$ (strictement inférieur)
1100	GT	$>$ (strictement supérieur)
1101	LE	$\leqslant$ (inférieur ou égal)
1110	AL	toujours
1111	NV	réservé

TABLE 4 – Liste des prédicats de condition

2.4 Étages

Le pipeline de ce processeur ARM est divisé en quatre étages : **IFETCH**, **DECOD**, **EXE** et **MEM**. Les étages du pipeline sont séparés par des fifos qui régulent la progression des instructions.

Ce processeur est doté d'une architecture asynchrone ; par conséquent, les instructions progressent dans les étages de telle sorte qu'elles soient le plus indépendantes possible.

Il est également important de gérer les dépendances entre les instructions. Chaque registre et flag dispose d'un bit de validité associé. Une instruction n'est exécutée que si toutes ses opérandes sources sont valides.

Le registre de destination est marqué comme invalide lors du lancement de l'instruction et jusqu'à la production effective du résultat.

L'étage EXE, principalement composé d'un shifter et d'une ALU, reçoit ses instructions de DECOD. De plus, EXE transmet les instructions mémoire vers MEM. EXE envoie également ses résultats au banc de registres si nécessaire. Le rôle principal de EXE est de calculer le résultat des instructions arithmétiques et logiques mais également des adresses des transferts mémoire ou bien des adresses de branchement.

Le schéma suivant, issu du cours, montre comment les différents étages s'articulent entre eux.

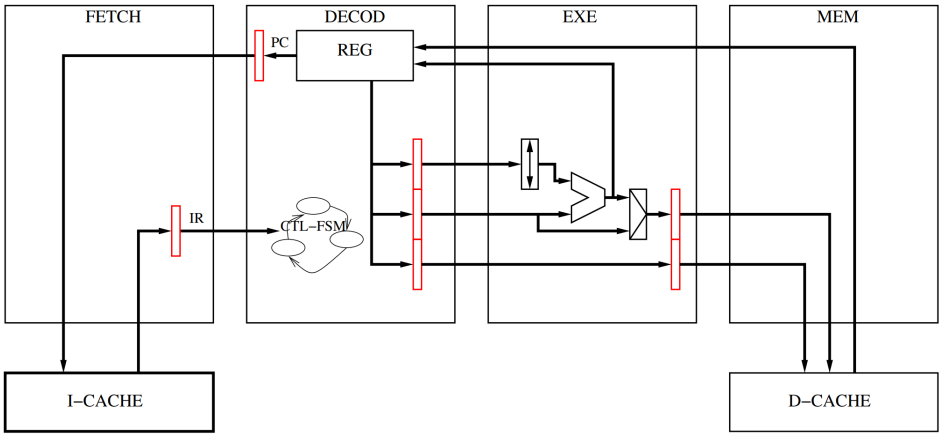


FIGURE 1 – Les étages du pipeline de notre processeur ARM

3 Étage EXEC

Dans cette partie nous allons détailler la conception du bloc EXE de notre processeur.

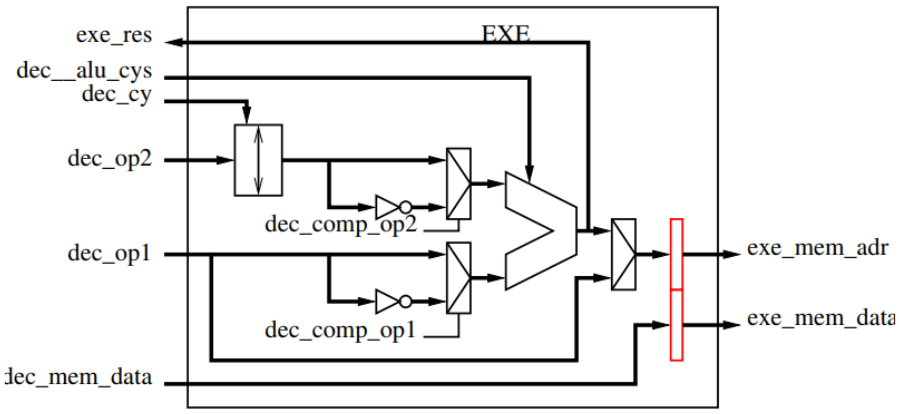


FIGURE 2 – Schema détaillé du bloc EXE

Comme dit précédemment, il est composé d'une ALU, d'un shifter et d'une fifo, il récupère de l'étage DECOD les opérandes ainsi que les informations nécessaires à l'exécution de l'instruction courante. Les opérations sont réalisées par l'ALU et le résultat est ensuite envoyé dans DECOD pour une écriture dans les registres ou dans MEM (calcul d'adresse) pour un accès mémoire. Chaque opérande de l'ALU est sélectionné entre sa valeur et son inverse à l'aide de multiplexeurs afin de réaliser des soustractions. L'opérande 2 passe aussi par le shifter, permettant par exemple de manipuler des immédiats de 32 bits codés sur 5 bits.

3.1 ALU

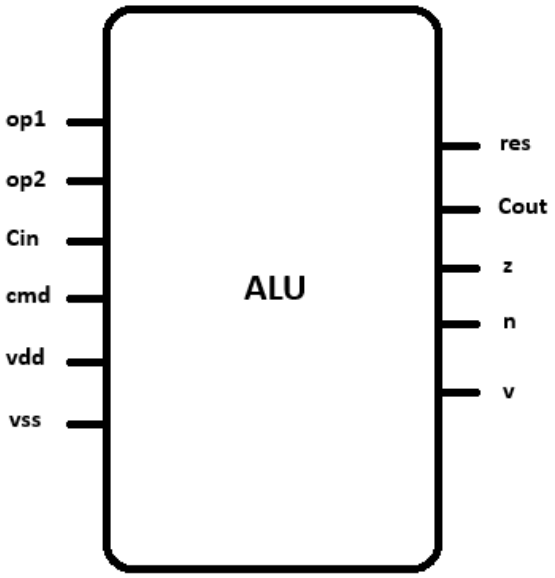


FIGURE 3 – Entrées et sorties de l'ALU

3.1.1 Implémentation de l'ALU

Notre implémentation de l'unité arithmétique logique est capable de réaliser 4 types d'opérations : Addition (32 bits), AND logique, OR logique et XOR logique. La commande de l'opération est codée sur 2 bits : 00 pour l'addition, 01 pour AND, 10 pour OR et 11 pour XOR. Afin de réaliser les additions sur 32 bits, nous avons mis en cascade 32 full adder 1 bit. En sortie, l'ALU délivre le résultat de l'opération ainsi que 4 drapeaux : Négatif (N), Zéro (Z), Carry out C) et Overflow (V) donnant plus d'informations sur le calcul effectué.

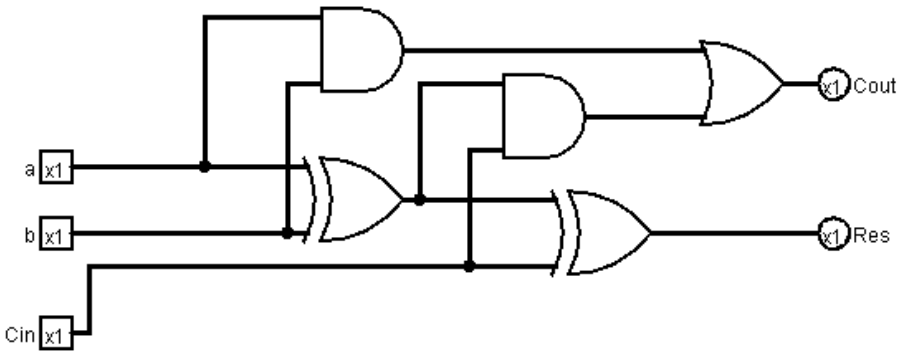


FIGURE 4 – Schema logique de l'additionneur 1 bit

3.1.2 Testbench de l'ALU

Voici la simulation de notre ALU, avec des tests sur toutes les différentes opérations qu'il est capable de réaliser. Nous avons d'abord testé les additions avec overflow et retenue sortante, puis les comparaisons

logiques OR, AND et enfin XOR. On peut voir que les flags (Négatif (N), Zéro (Z), Carry out C) et Overflow (V)) se mettent bien à jour en fonction du résultat.

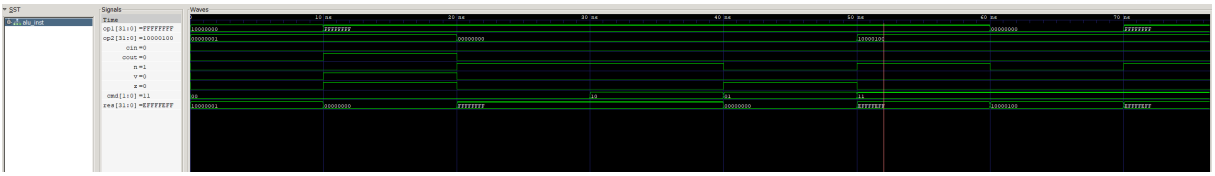


FIGURE 5 – Testbench ALU

3.2 Shifter

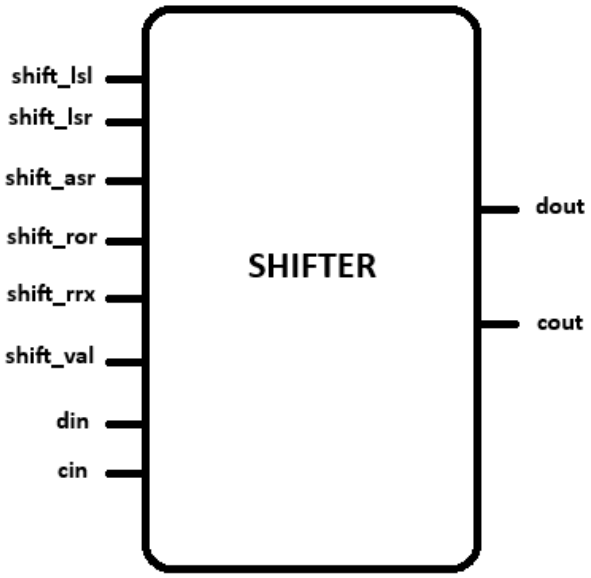


FIGURE 6 – Entrées et sorties du Shifter

3.2.1 Implémentation du shifter

L'implémentation de notre shifter repose sur une approche itérative, où des boucles sont utilisées pour effectuer des décalages successifs sur l'opérande din de 32 bits. Le processus démarre en ajoutant un bit à la fois à l'opérande, ce qui équivaut à un décalage unitaire. Ces itérations se répètent jusqu'à ce que le nombre spécifié de décalages, déterminé par le signal shift-val, soit atteint. Nous avons utilisé des variables temporaires pour stocker le résultat du décalage et le bit de report à chaque itération qui sont ensuite affectés aux sorties lorsque le décalage est terminé.

```
if shift_lsl = '1' then
    for i in 0 to shift_val'length loop
        temp := din & '0'; -- concaténation 0 à la fin du vect
        sig_cout <= temp(32); -- récupération de la carry (CF)
        sig_dout <= temp(31 downto 0); -- on met les 32 autres bits dans sig_dout
    end loop;
```

FIGURE 7 – Boucle décalage logique à gauche

3.2.2 Testbench du shifter

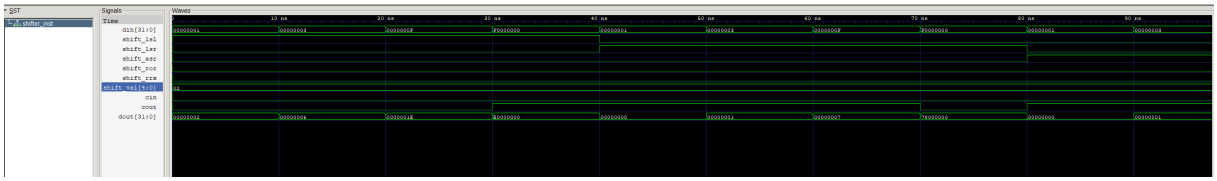


FIGURE 8 – Testbench Shifter

Dans ce testbench, nous avons testé les 5 modes de shift possibles ainsi que le bon fonctionnement de la retenue sortante.

3.3 FIFO 72 bits

Une FIFO (First In First Out) est utilisée pour connecter les étages "exec" et "mem" du processeur, facilitant la communication lors des accès mémoire. Elle assure la synchronisation entre ces étages en stockant et transmettant des données de 72 bits. Contrôlée par les signaux "push" et "pop", elle utilise les indicateurs "full" et "empty" pour signaler l'état dans lequel elle se trouve. Le transfert de signal de EXEC à MEM via la FIFO facilite l'envoi des informations lors des accès mémoire, telles que le type d'accès (load ou store), la donnée à écrire (dans le cas d'un store), l'adresse, et la destination d'écriture dans le registre (dans le cas d'un load).

3.4 Testbench de l'étage EXEC

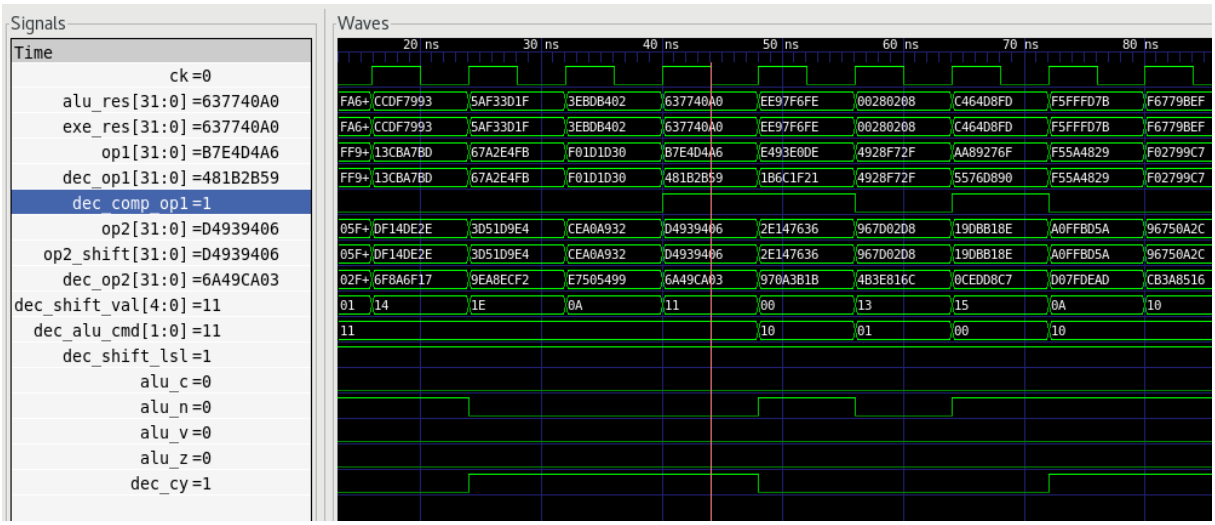


FIGURE 9 – Testbench de l'étage EXE

Voici le testbench de notre étage EXE. Dans cette simulation, nous avons généré des signaux aléatoires pour les opérandes, la séléction de la commande de l'ALU, la valeur du shift ainsi que l'inversion de OP1. On peut voir que le premier opérande s'inverse lorsque dec-comp-op1 passe à 1, l'opérande 2 qui shift à gauche du montant de dec-shift-val et le résultat de l'ALU qui est bien transmis à exe-res.

4 Étage DECOD

L'étage de décodage, a pour mission de décoder les instructions issues du bloc Fetch. Il se compose d'un banc de registres ainsi que d'une machine à états dédiée à la gestion des FIFOs. Ces FIFOs sont cruciales pour transmettre les instructions à exécuter vers l'unité d'exécution, gérant ainsi les différentes opérandes, les calculs et les adresses associés. Simultanément, elles facilitent l'envoi du PC vers le bloc Fetch, tout en recevant les instructions à exécuter, qui sont préalablement acheminées depuis Fetch. Cette architecture élaborée permet une coordination efficace des différentes étapes du processeur, contribuant ainsi à l'exécution précise et fluide des instructions.

4.1 Banc de registres

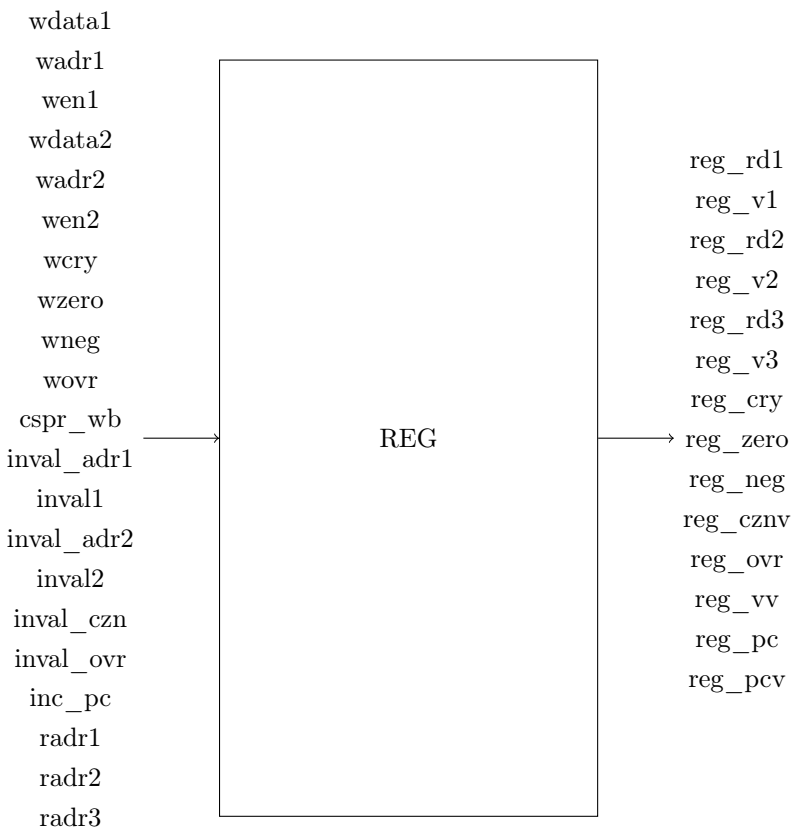


FIGURE 10 – Entrées et sorties de REG

Le bloc REG abrite les 16 registres du processeur, dont le PC, et les quatre indicateurs de drapeaux essentiels : C, Z, N, et V. Chaque registre de 32 bits est accompagné d'un bit de validité. Au démarrage et après un reset, tous les registres sont considérés comme valides, même s'ils n'ont pas de valeurs pertinentes à ce stade.

Lorsqu'un registre est identifié par le bloc DECOD comme destination d'une instruction, il devient invalide. Lorsqu'un résultat est généré par les blocs EXEC ou MEM et écrit dans REG, le registre de destination est marqué comme valide. Cependant, une écriture dans un registre n'est autorisée que si ce dernier est préalablement invalidé, signalant ainsi son inaccessibilité pour d'autres instructions potentielles. Le registre numéro 15 (PC) est traité de manière particulière, associé à un opérateur réalisant l'opération +4. Son contenu et sa validité sont directement accessibles depuis l'interface de REG.

Concernant les indicateurs de drapeaux, les instructions logiques agissent sur les trois drapeaux C, N et Z, tandis que seules les instructions arithmétiques ont une incidence sur le drapeau V.

Le banc de registres offre trois ports de lecture numérotés de 1 à 3 et deux ports d'écriture, le numéro 1 correspondant à EXEC et étant donc prioritaire. De plus, deux ports d'invalidation sont disponibles, permettant d'invalider un registre en vue d'une réécriture. Lors de la lecture d'un registre, sa valeur et son état de validité sont récupérés, la machine à états de DECOD déterminant ultérieurement si l'instruction associée peut être exécutée en fonction de la validité du registre. Ce principe s'applique également aux

indicateurs de drapeaux.

4.1.1 Implémentation du banc de registres

Pour réaliser REG, nous avons utilisé un tableau de 16 vecteurs de 32 bits pour le banc de registres ainsi qu'un tableau de 16 std-logic pour les bits de validités.

```
architecture Behavior OF Reg is

    type banc is array (0 to 15) of std_logic_vector(31 downto 0);
    type bancV is array (0 to 15) of std_logic;
    signal RegisterFile : banc;
    signal register_valid : bancV;
    signal FlagC_int, ValidFlagC_int, FlagZ_int, ValidFlagZ_int,
           FlagN_int, ValidFlagN_int, FlagV_int, ValidFlagV_int : STD_LOGIC;

begin

    process(ck, reset_n)
    begin
        if (reset_n = '0') then

            RegisterFile <= (others => (others => '0'));
            register_valid <= (others => '1');  -- Tous les registres valides au reset
            FlagC_int <= '0';
                        FlagN_int <= '0';
                        FlagZ_int <= '0';
                        FlagV_int <= '0';
            ValidFlagC_int <= '1';
                        ValidFlagN_int <= '1';
                        ValidFlagZ_int <= '1';
                        ValidFlagV_int <= '1';

        elsif (rising_edge(ck)) then
            -- Gestion de l'écriture dans le registre 1 (prioritaire)
            if (wen1 = '1') then
                                if (register_valid(to_integer(unsigned(wadr1))) = '1') then
                                        RegisterFile(to_integer(unsigned(wadr1))) <= wdata1;
                                        register_valid(to_integer(unsigned(wadr1))) <= '0';
                                end if;
            end if;
```

FIGURE 11 – Extrait du code de REG

4.1.2 Testbench du banc de registres

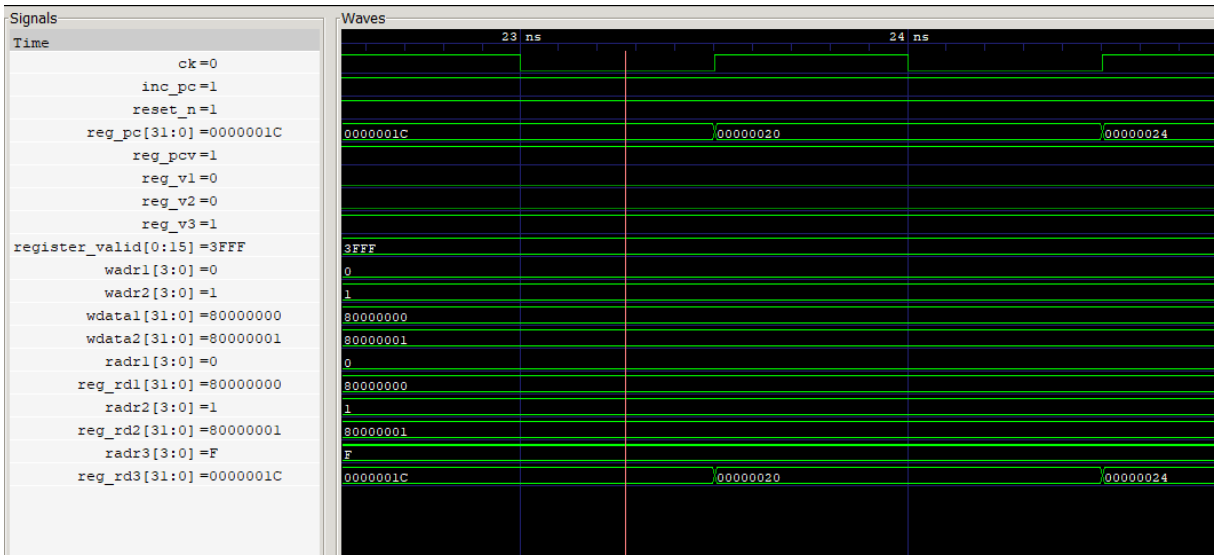


FIGURE 12 – Testbench de REG

L'inconvénient d'avoir utilisé un tableau de vecteurs pour le banc de registre est qu'on ne peut pas directement les visualiser sur gtkwave mais on peut quand même vérifier leur contenu à l'aide des ports de lecture. Dans ce testbench on donc a pu tester l'incrémentation du registre PC à chaque front d'horloge, les 3 ports de lecture et les 2 ports d'écriture ainsi que les bits de validité des registress qui se mettent à jour.

4.2 Décodage des instructions

Le processus de décodage des instructions dans l'étage "decod" de notre processeur s'effectue en plusieurs étapes. Tout d'abord, les prédicats sont décodés par des comparaisons entre les flags et le prédicat, déterminant ainsi si l'instruction peut être exécutée en fonction de la validité des flags provenant du banc de registres. Ensuite, le type de l'instruction est décodé, couvrant diverses opérations telles que le traitement des données, les branchements, les transferts mémoire, les transferts multiples, etc.

Pour les instructions de traitement de données, l'opcode de l'instruction contrôle l'ALU, déterminant également les inversions nécessaires et la gestion de la retenue (dans le cas d'une soustraction). Le bit d'immédiat indique si l'opérande 2 provient d'un registre, pouvant être décalé par une valeur de registre ou une valeur directe présente dans l'instruction. L'opérande 1 est un registre directement désigné par son adresse dans l'instruction.

Dans le cas des branchements, un bit indique s'il s'agit d'un appel de fonction, nécessitant la sauvegarde de PC+4. Le branchement s'effectue en ajoutant l'offset présent dans l'instruction à PC.

Pour les transferts mémoire, le processus est similaire à celui du traitement des données avec le calcul de l'adresse cible. Divers bits spécifient des détails tels que la direction (chargement ou stockage), la taille (octet ou mot), et les opérations sur l'adresse de base.

4.3 Machine à états

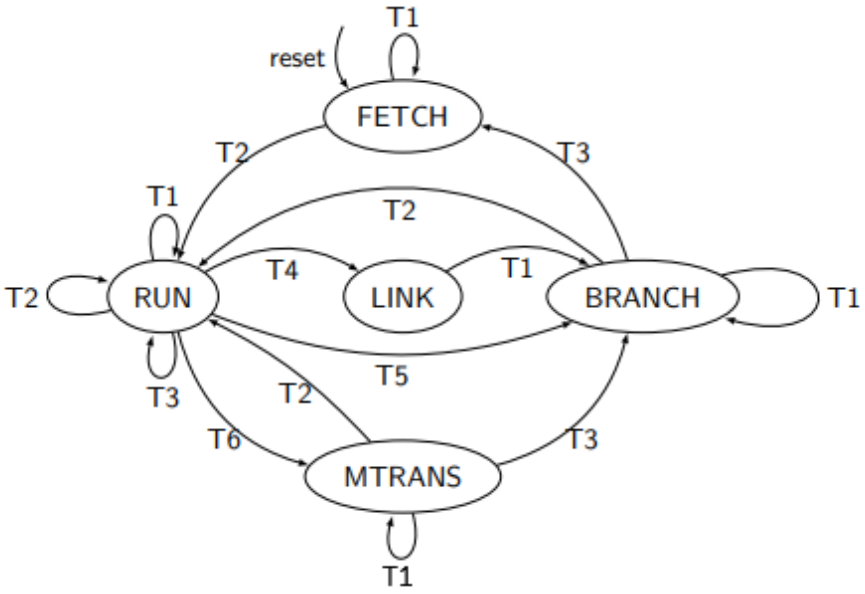


FIGURE 13 – Machine à Etats de Mealy de l'étage DECOD

L'état FETCH est dédié à l'initialisation du processeur, demandant la première instruction au cache d'instructions. Si la valeur de PC est valide et que la FIFO "dec to if" est vide, le processeur passe à l'état RUN ; sinon, il reste en FETCH.

L'état RUN est l'état standard, où l'exécution des instructions est déterminée en fonction des prédicats, de la validité des registres et des drapeaux. Les instructions de traitement de données et de transfert mémoire sont exécutées ici. En fonction des conditions, l'état peut basculer vers d'autres états tels que BRANCH ou MTRANS.

On reste à l'état RUN dans 3 cas :

— Si l'instruction est exécutable, on push dans dec-to-if si elle n'est pas pleine.

— Si le prédicat est faux, l'instruction doit être jetée, pop de if-to-dec et push dans dec-to-if si elle n'est pas pleine.

— Si l'instruction n'est pas exécutable, on push dans dec-to-if si il y a de la place.

L'état LINK sauvegarde PC+4 dans le registre de liaison, puis passe directement à l'état BRANCH. Il envoie également une information à EXEC pour sauvegarder PC.

L'état BRANCH calcule l'adresse cible de PC en ajoutant un offset au PC actuel. Si if-to-dec est vide on reste en BRANCH sinon on retourne à RUN.

L'état MTRANS permet de boucler pour les transferts mémoire multiples.

4.4 Simulation du processeur

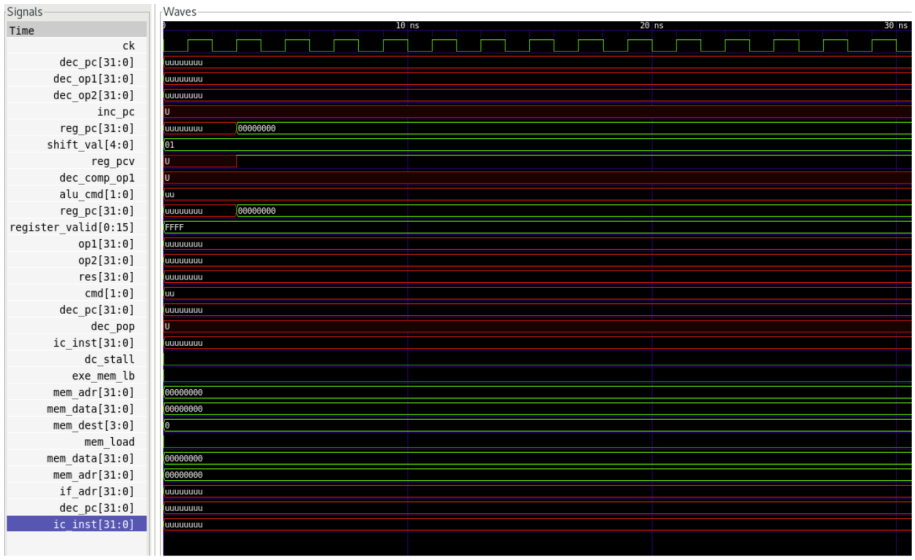


FIGURE 14 – Simulation du processeur avec un programme test

Voici le dernier test (peu glorieux) que nous avons effectué. On a essayé l'execution d'un programme assembleur composé d'une instruction MOV, d'un ADD et de deux nop. Malheureusement, on peut clairement voir que cela ne fonctionne pas, même dans l'IR de l'étage FETCH, aucune instruction n'est récupérée, notre registre PC reste bloqué à 0....

5 Conclusion

Ce projet a été un véritable défi, mais aussi une aventure passionnante dans l'univers complexe du VHDL et de l'architecture des processeurs. L'acquisition de nouvelles compétences a été au rendez-vous, que ce soit dans la conception des circuits ou dans la compréhension approfondie du fonctionnement interne d'un CPU.

Bien sûr, le regret persiste de ne pas avoir pu voir le processeur en action, mais parfois, les obstacles font partie intégrante de l'apprentissage. La synthèse aurait été la cerise sur le gâteau, mais le temps et quelques pépins techniques en ont décidé autrement.

Ce projet inachevé n'est pas une fin en soi, mais plutôt le point de départ d'une curiosité renouvelée. L'envie d'explorer davantage et de poursuivre cette aventure reste bien présente. Même si le résultat final n'est pas au rendez-vous, l'expérience acquise ouvre la voie à de nouvelles opportunités et découvertes.