VLSI

Rapport Partiel de projet :

Modélisation d'un cœur de processeur ARM

Louis Geoffroy Pitailler, Timothée Le Berre

December 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 ARMv2. Pour ce faire on utilise le langage de description matérielle VHDL.
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 decode,
- Decode : récupère l'instruction chargé par Fetch et procède à son décodage afin de sélectionner les opérandes, registres et calculs nécessaires à son exécution,
- 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 :

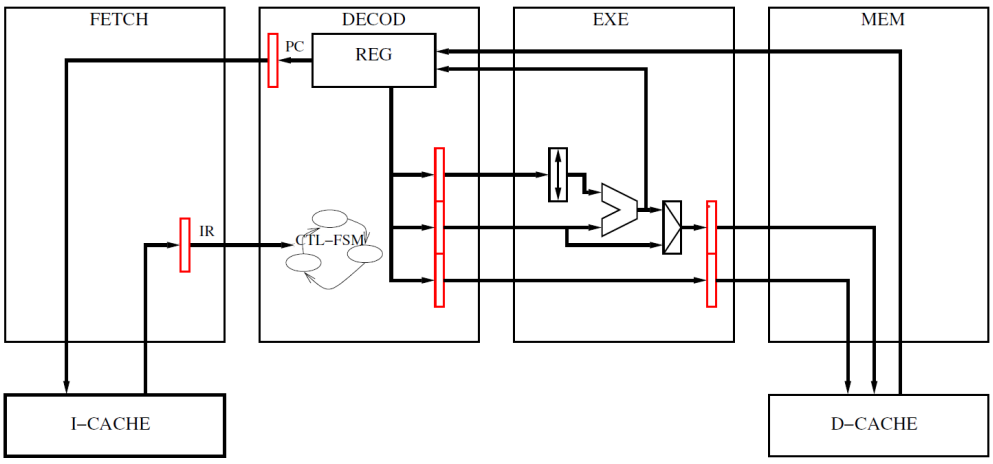


Figure 1: Schéma simplifié du pipeline

2 EXE

2.1 Introduction :

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

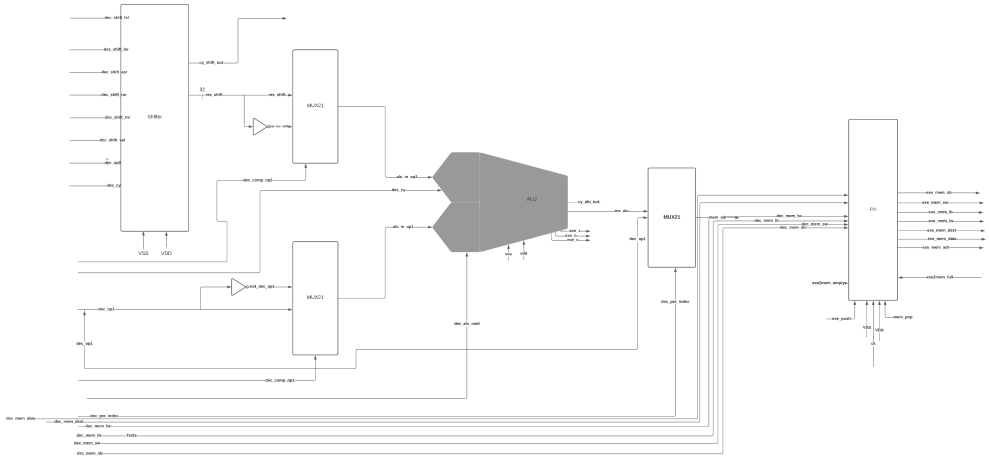


Figure 2: Etage EXE

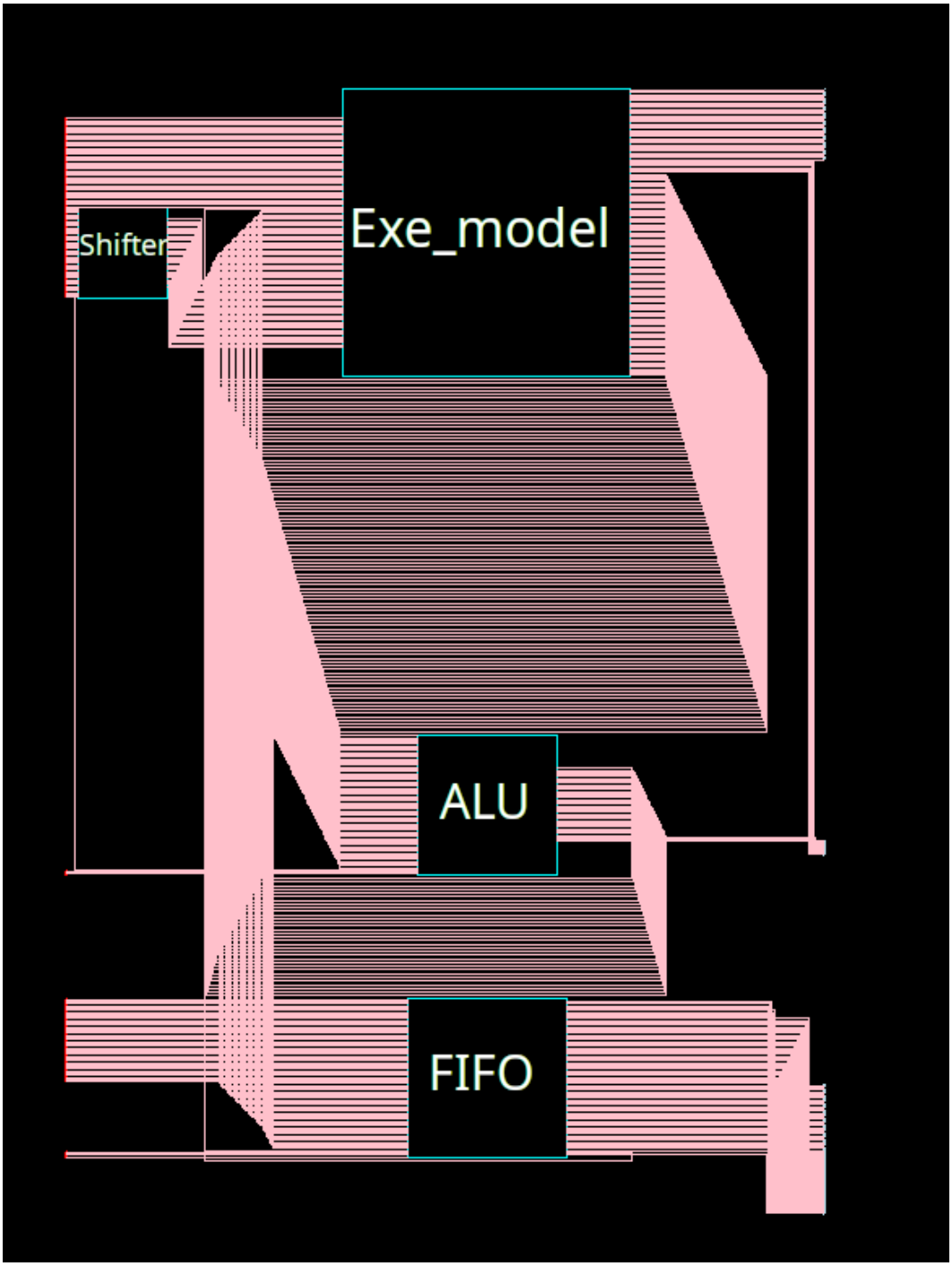


Figure 3: Etage EXE : circuit

2.2 ALU : Arithmetic logic Unit

La modélisation est assez simple, on envoie un signal de commande sur 2 bits à l'ALU et on sélectionne ainsi 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 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 arithmetic 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 map en fonction de la valeur de shift que l'on veut.
En combinant ces shifters, on peut shifter n'importe quelle valeur entre 0 et 31.

2.3.1 Shifter right :

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.

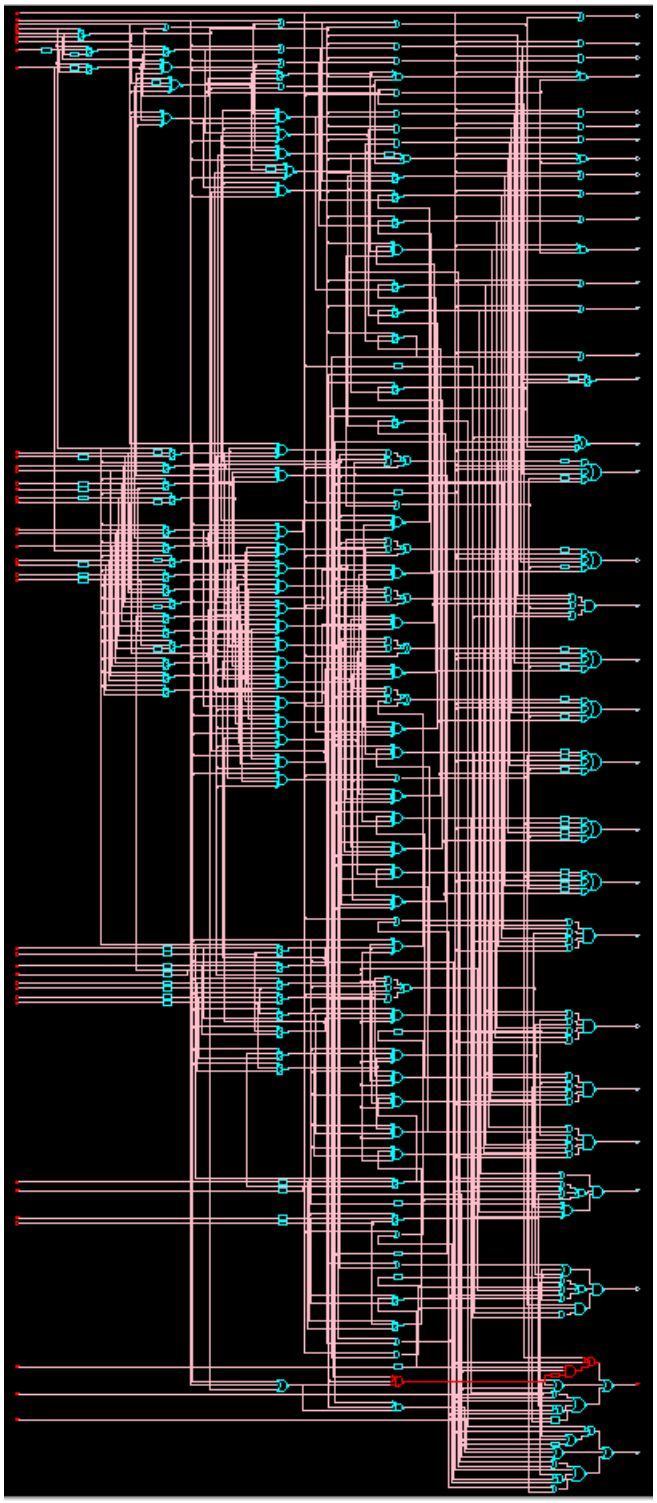


Figure 4: Circuit du shift right

2.3.2 Shift left

La mise en œuvre est exactement la même que pour le Shifter right à la différence près que la retenue n'est pas la même. En effet, dans ce cas, on ne prend pas le dernier bit de poids faible, mais le 1er bit de poids fort que l'on stocke dans la retenue.

2.3.3 Ror

Pour la rotation, on effectue une concaténation de n bits (n étant la valeur du shift value) vers la gauche.
Cette entitée permet également de gérer les shifts rrx, en effet un shift rrx n'est rien d'autre qu'un shift ror pour lequel la retenue est égale à 1. La gestion de cette retenue étant différente, on implémente directement la gestion de ce type de shift dans l'entité ror.

3 DECOD :

DECODE 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 IFECTH mais il doit également gérer l'ensemble du pipeline en indiquant à quel étage quoi faire 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.

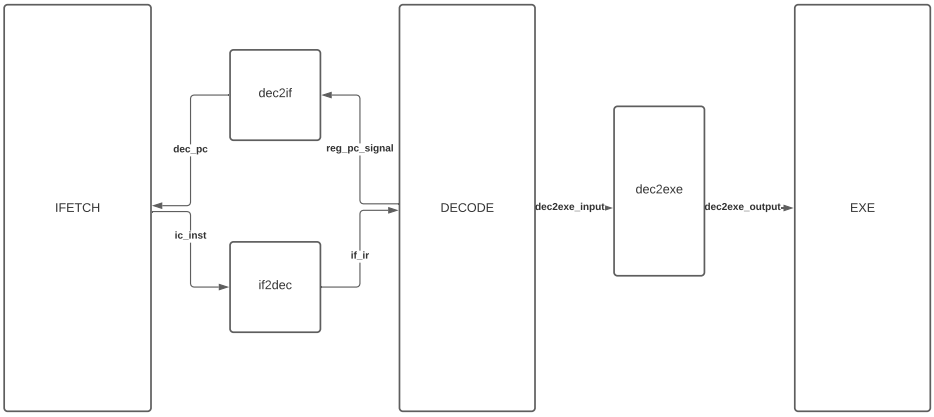


Figure 5: Schema global des étages IFECTH, DECODE et EXE

3.1 REG : Registre Bank

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

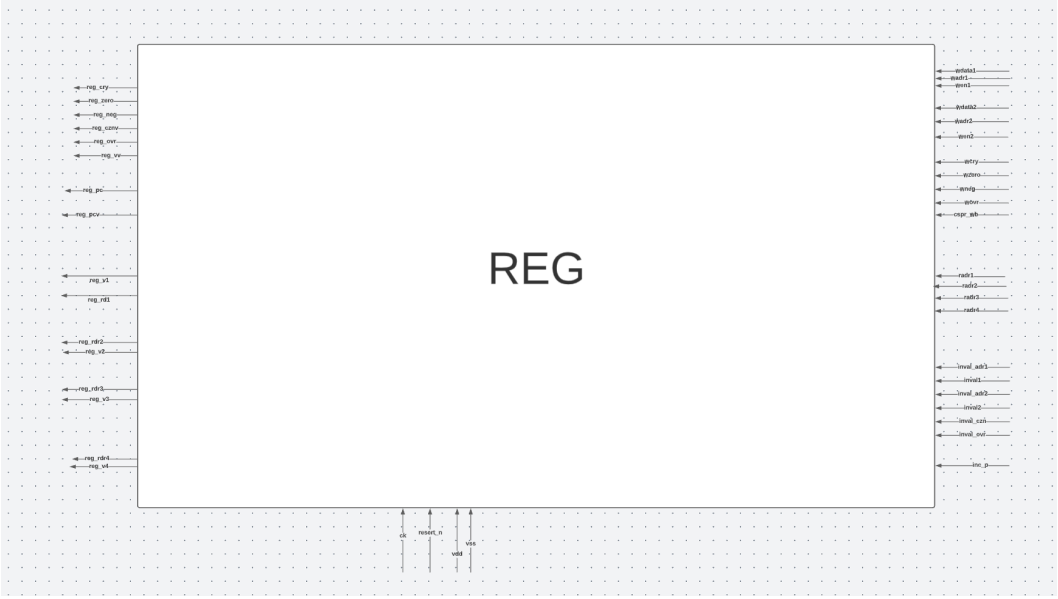


Figure 6: Banc de registre

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 :

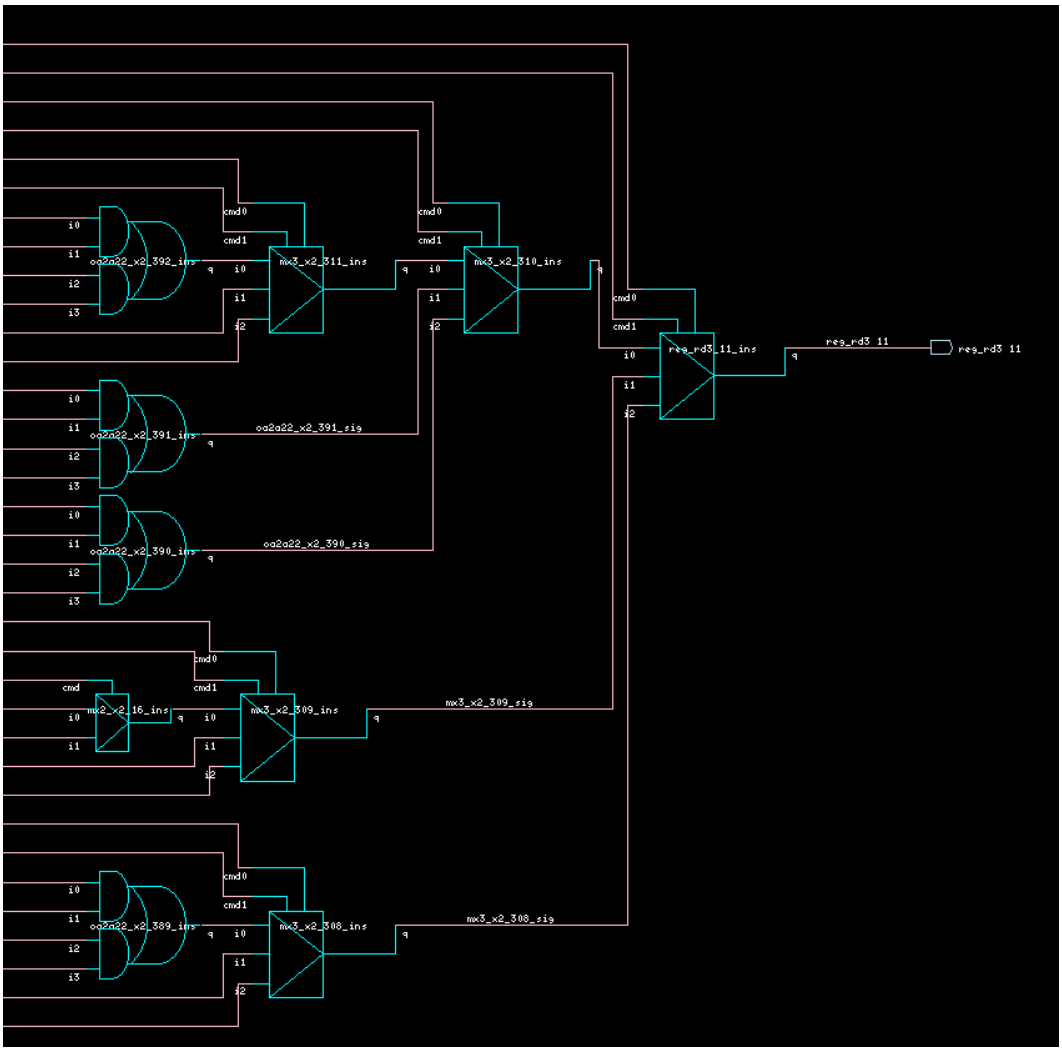


Figure 7: Lecture d'un bit dans un registre

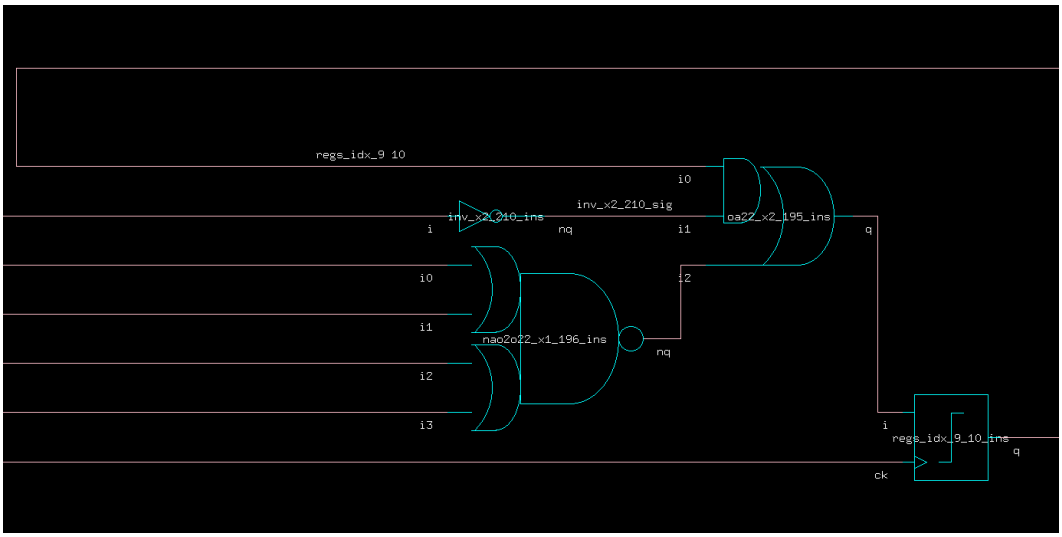


Figure 8: Écriture d'un bit dans un registre

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 on peut accéder au minimum à un byte à la fois, donc les adresses accessibles sont des multiples de 4. Ainsi lorsque l'on passe à l'instruction suivante il convient d'incrémenter PC de 4.

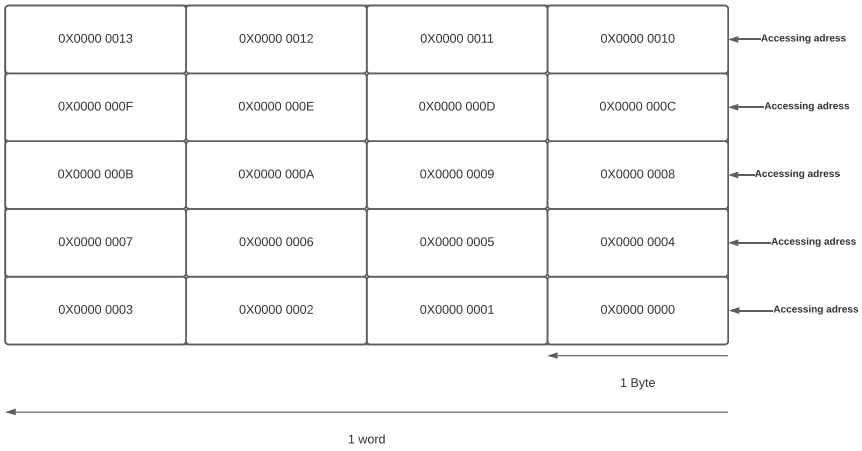


Figure 9: Strcture de la mémoire

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.

3.2 DECODE

3.2.1 Machine à état :

Pour que decode fonctionne correctement, nous avons besoin d'utiliser une machine à état. Nous avons choisit une machine de Mealy.

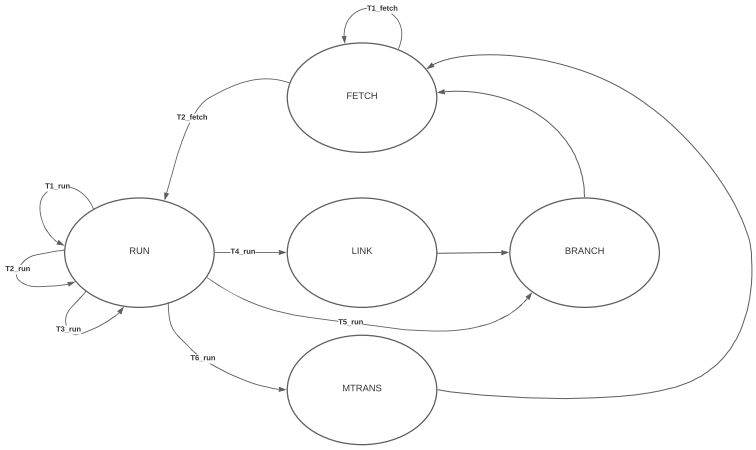


Figure 10: Machine à état

Les transitions permettant le passage d'un état à un autre sont recensés dans le tableau ci dessous. Nous avons dans un premier temps choisit de ne pas implémenter les transferts multiples. En effet ces derniers étant assez compliqué à implémenter nous avons préférer nous focaliser sur le reste des instructions existantes dans le but d'avoir un processeur parfaitement fonctionnel avant de commencer leur implémentation.
C'est pour cela que tout ce que l'on fait lorsque l'on est dans l'état MTRANS consiste à retourner dans l'état FETCH.

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	NA	Non implémenté,retour à FETCH

3.3 Décodage des instructions :

Les instructions reçues par décode 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 3 types d'opérations listées ci dessous.

Ces opérations sont les plus usuelles et les plus simples. Pour les décoder nous nous sommes aider de la documentation ARM fournie lors du CM 3.
Dans tous les cas, les 4 bits de poids fort de l'instruction désigne 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** - $Z = 1$	1000 **HI** - $C = 1$ et $Z = 0$
0001 **NE** - $Z = 0$	1001 **LS** - $C = 0$ ou $Z = 1$
0010 **HS**/**CS** - $C = 1$	1010 **GE** - supérieur ou égal
0011 **LO**/**CC** - $C = 0$	1011 **LT** - strictement inférieur
0100 **MI** - $N = 1$	1100 **GT** - strictement supérieur
0101 **PL** - $N = 0$	1101 **LE** - inférieur ou égal
0110 **VS** - $V = 1$	1110 **AL** - toujours
0111 **VC** - $V = 0$	1111 **NV** - réservé.

Figure 11: Prédicats

3.3.1 Regop_t : Regular operation :

La structure d'un instruction régulière est la suivante :

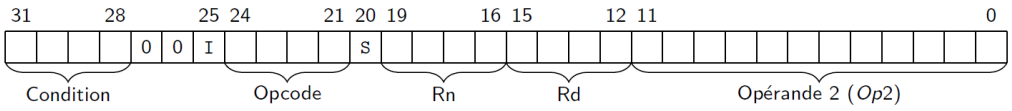


Figure 12: regop_t encodage

La condition d'exécution de l'instruction est celle définition par la liste fournie sur la figure 11. Les opcodes sont quant à eux définis sur la figure suivante 13 :

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 13: regop_t Opcode

L'opérande 2 va quant à elle se décoder de différente manière selon la valeur du bit 25 qui indique si on considère une 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 :

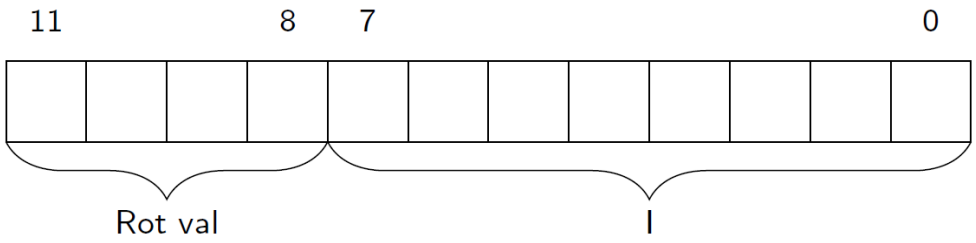


Figure 14: Décodage Op2 dans le cas où I = 1

Les 8 bits de poids faible désigne l'opérande que l'on souhaite utiliser. Cett opérande n'étant que sur 2 bits on utilise les 4 bits suivant pour faire un rotation de cette dernière afin de pouvoir coder des entiers supérieurs à $2^8 - 1$.

Dans le cas où l'opérande n'est pas de type immédiat les 4 bits de poids fort désigne un 2ème registre de lecture tandis que les 8 bits suivants vont avoir

deux codages différents en fonction de la valeur du bit 4 :

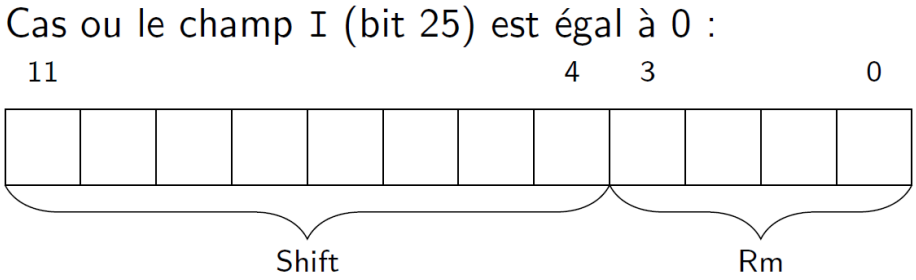


Figure 15: Codage des 12 bits de poids faible d'une regop dans le cas où I = 0

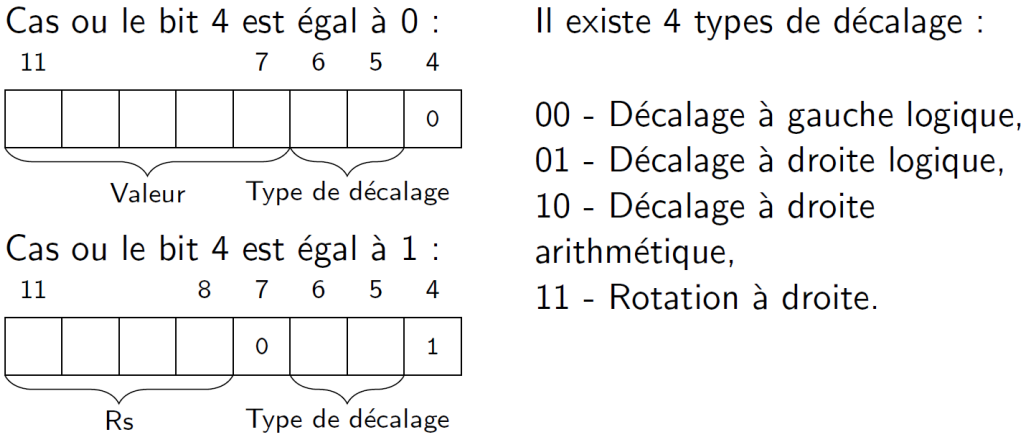


Figure 16: Deux codages différents selon la valeur du bit 4

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.
Tous les types de décalage possibles sont ceux que le Shifter est en mesure de faire.
Ainsi on envoie les opérandes décodées dans deux signaux à EXE, on décode également si l'on souhaite faire le complément à deux d'un signal, typiquement pour l'instruction sub il faut que l'on prenne le complément de l'opérande 2.

Un autre point important est l'autorisation de Write back ou non le résultatn typiquement pour les instructions tst, teq, cmp et cmn on ne souhaite pas que le résultat calculé par l'ALU soit écrit dans le banc de registre. Nous avons donc désactivé le Write Back dans le cas où l'on serait en train de décoder ces instructions .

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 ARM V2 l'exécution des branchements se fait en suffixant l'instruction d'un des prédicats explicitant dans quel cas le branchement est réalisé.

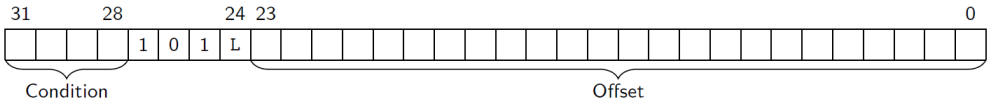


Figure 17: Codage d'une instruction de branchement

Deux cas se présente lorsque l'on effectue un branchement, si la condition d'exécution n'est pas satisfaite l'instruction est jettée et on continue l'exécution du programme en séquentiel. Si elle réussi 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 : Transfer operation :

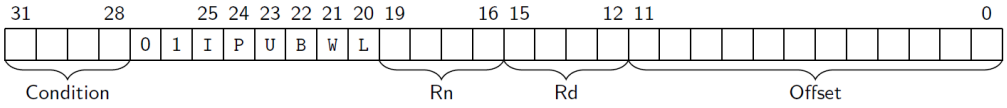


Figure 18: Codage d'une instruction de transfert simple

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 19: Détail de la signification des bits des instructions de trans$_t$

Les instructions de transfert simples permettent d'écire dans des registres ou de lire dedans. Ces accès sont faisabes soit par byte soit par word.
Dans le cas d'une écriture nous avons veiller à désactiver l'invalidation du registre rd, en effet ce dernier étant uniquement lu il n'est pas nécesssaire 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 lón 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 $2^{12} - 1$.

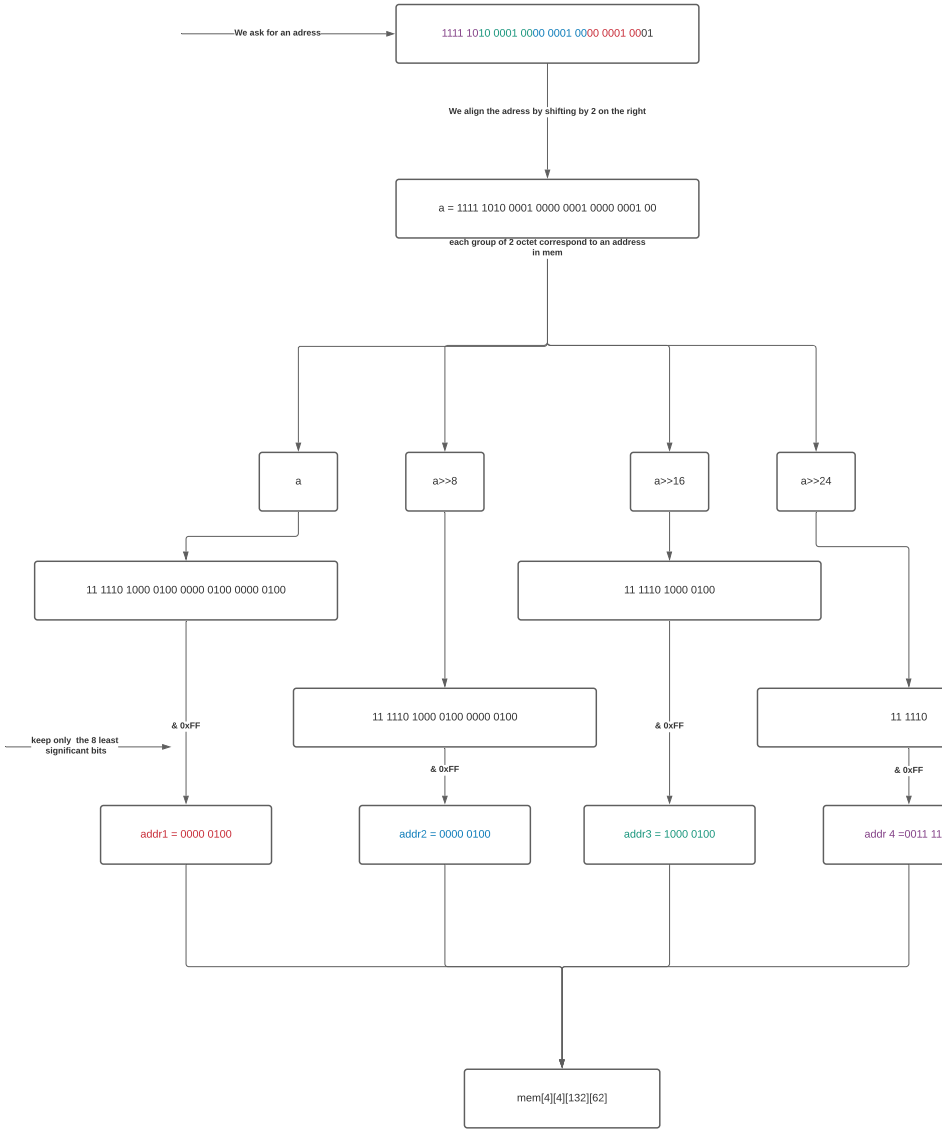
4 Protocole de test :

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 testbench envoyant des nombres aléatoires dans les entrées de nos entitées 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 decode effectuait son travail correctement.

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

Une fois decode finit 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 du la simuler à l'aide d'un fichier que nous avons appelé core.c .
Le principe étant que notre testbench ghdl puisse aller écrire et lire en mémoire et ce à l'aide de fonctions externe définient en c.
Nous avons donc créer un quadruple tableau de pointeur de taille 256 simulant la mémoire. On aurait pu choisir un tableau simple mais nous aurions alors alloué un espace mémoire beaucoup trop important, l'intéret de faire comme nous avons fait étant de limité la taille en mémoire occupé par ce tableau.
Les instructions assembleur étant écritent en hexadécimal dans un fichier texte

il va falloir que nous allions les récupérer.
Nous avons donc définie trois fonctions : get_inst, get_mem et write_mem, la premiere récupérant une instruction en mémoire, la deuxième lisant la mémoire et enfin la dernière permettant d'écrire en mémoire.
get_inst va récupérer une instruction dans notre fichier texte Les adresses accéder pouvant ne pas être aligné on va s'assurer que c'est le cas en shiftant de 2 vers la droite n'importe quel adresse reçue.


We ask for an adress
1111 1010 0001 0000 0001 0000 0001 0001
We align the adress by shifting by 2 on the right
a = 1111 1010 0001 0000 0001 0000 0001 00
each group of 2 octet correspond to an address in mem
a
a>>8
a>>16
a>>24
11 1110 1000 0100 0000 0100 0000 0100
11 1110 1000 0100
11 1110 1000 0100 0000 0100
11 1110
keep only the 8 least significant bits
& 0xFF
& 0xFF
& 0xFF
& 0xFF
addr1 = 0000 0100
addr2 = 0000 0100
addr3 = 1000 0100
addr 4 =0011 11
mem[4][4][132][62]

4.2 Simulation complète :

Pour tester nos prototypes, nous avons réalisé des test bench pour chacun des modèles.
Pour ce faire on a utilisé une fonction générant un nombre aléatoire sur n bits dans le but de tester toutes les configurations possibles.
Nous affichons ensuite tous les signaux dans le terminal en utilisant des REPORT, et nous vérifions à la main les résultats pour les différents cas.
Nous avons également utilisé GTKwave afin d'avoir une bonne visualisation temporelle de l'évolution de nos signaux.

5 Conclusion

Il nous reste à faire le branchement final des différents étages, et à réaliser un test bench pour vérifier que tout fonctionne correctement.
Nous avons réalisé une FIFO générique (donc d'une profondeur arbitraire), et même si les GENERIC ne sont pas synthétisables, il nous suffira de les remplacer par des entiers littéraux pour avoir une FIFO de longueur 4 ou 8 par exemple.