Rapport VLSI

Ayrault Maxime

Introduction

Présentation du projet

Cette UE de VLSI avait pour but de créer un dessin des masques pour un clone de processeur ARM V3. Il a été écrit entièrement en VHDL.

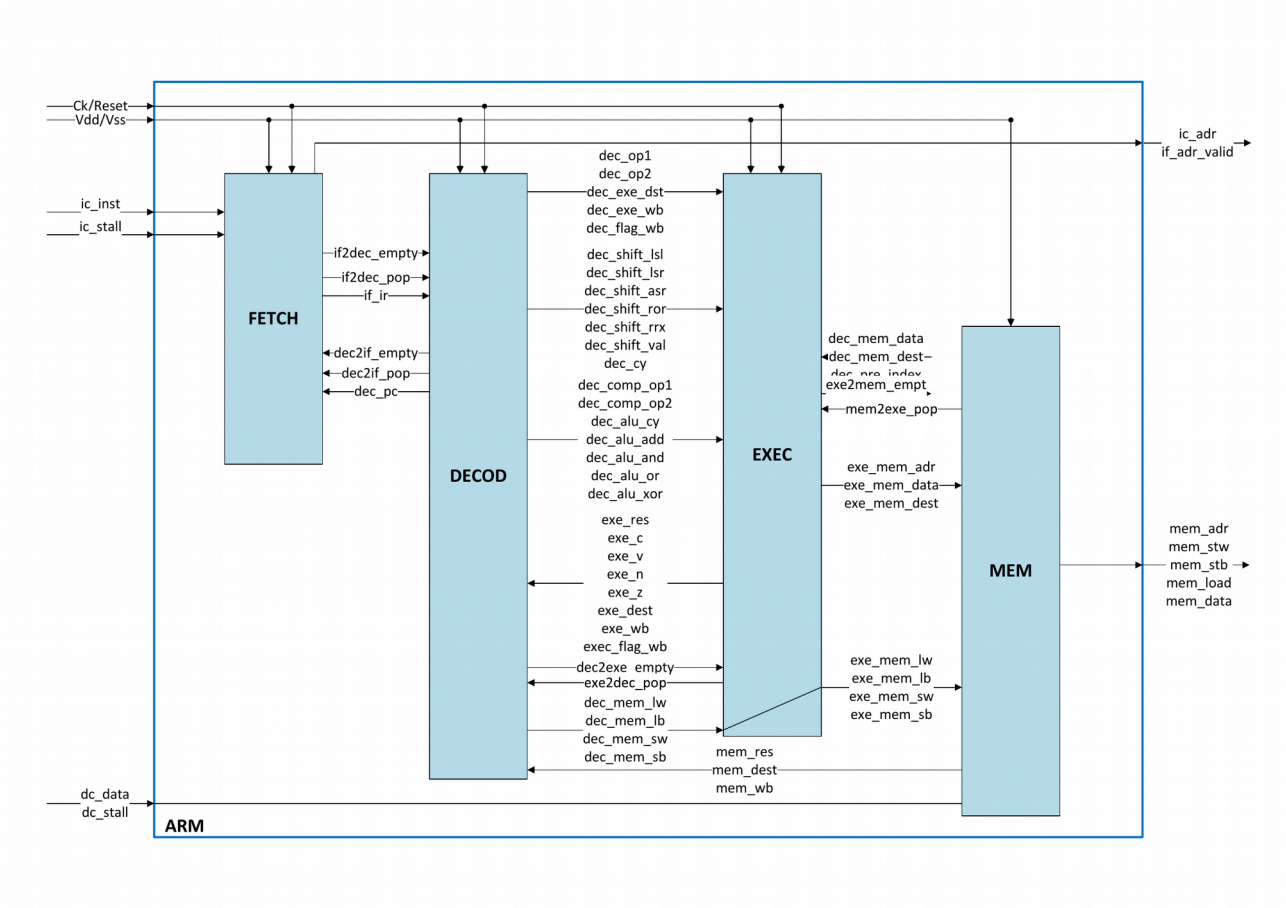
Notre processeur se base donc sur une architecture ARM pipeline à 5 étages

IFETCH : Récupérer l'instruction, à l'adresse donnée par le registre PC, dans le cache.
DECOD : Décoder l'instruction que IFETCH lui a envoyée, et transmettre les bonnes informations à l'étage suivant pour permettre l'exécution de l'instruction voulue.
EXEC : Exécuter les calculs demandés selon les informations reçues en entrée, calcul numérique, calcul logique, calcul d'adresse...
MEM : Ecrire la valeur contenue dans un registre à l'adresse calculée par EXE, ou lire une valeur en mémoire à l'adresse calculée.
Write Back : Mettre à jour le contenu des registres et leur validité.

Pour compiler, un Makefile est disponible dans chaque étage, il faut le lancer avec la commande :
```
$< make clean all
```
Un autre est disponible à la racine du dossier pour compiler le projet en entier. Il se lance de la même manière

Le code du projet est disponible ici :
https://github.com/maximouth/VHDL

I- Architecture complète



Dans notre réalisation il a été demandé de rassembler *DECOD* et *Write Back* dans le même étage *DECOD* comme présenté sur le schéma ci-dessus.

Cette partie se trouve dans le fichier *arm_core.vhdl*

Elle correspond au processeur fini contenant tous les étages, il est capable d'exécuter les instructions qu'il va chercher dans le cache instruction et de gérer la mémoire.

Il prend en entrée :

- `ic_inst` et `ic_shall` qui viennent du cache instruction et donnent l'instruction à l'adresse demandée et si celle-ci est valide.
- `dc_data` et `dc_shall` qui viennent du cache mémoire

- `CLK`, `RESET`, `VDD` et `VSS` qui servent au fonctionnement du processeur et à l'alimenter.

Il rend en sortie :

-`ic_adr` et `ic_adr_valid` qui correspondent à l'adresse d'une instruction et à la validité de l'adresse.
- `mem_adr`, `mem_stw`, `mem_stb`, `mem_load`, `mem_data` qui servent à aller écrire dans la mémoire.

II- IFETCH

IFETCH est le premier étage de notre processeur pipeline. Il sert à aller chercher dans le cache instruction, l'instruction à l'adresse contenue dans le registre PC (r15).

Composant(s) de l'étage :

L'étage *IFETCH* est composé d'une fifo 1 place de 32 bits.

Il prend en entrée :

-- *Icache interface*
- `ic_inst` et `ic_stall` qui représentent l'instruction que renvoie le cache et la validité de cette instruction.

-- *Decode interface*
- `dec2if_empty`, qui indique si la fifo de *DECOD* contenant une adresse à charger est vide ou non.,
- `dec_pc`, correspond à l'adresse à laquelle aller chercher la prochaine instruction.
- `dec_pop`, pour savoir s'il faut prendre ou non la valeur dans la fifo.

--*Global interface*
- `CK`, `RESET` , `VDD`, `VSS` pour alimenter l'étage et le gérer

Il rend en sortie :

-- *Icache interface*
- `if_adr`, `if_adr_valid` qui contiennent respectivement l'adresse et la validité de cette adresse.

-- Decode *interface*
- `if_pop`, indique si l'on demande à vider la valeur contenue dans la fifo.
- `if_ir`, correspond à l'instruction récupérée dans le cache.
- `if2dec_empty`, indique si la fifo de cet étage est vide ou non.

Comportement de l'étage

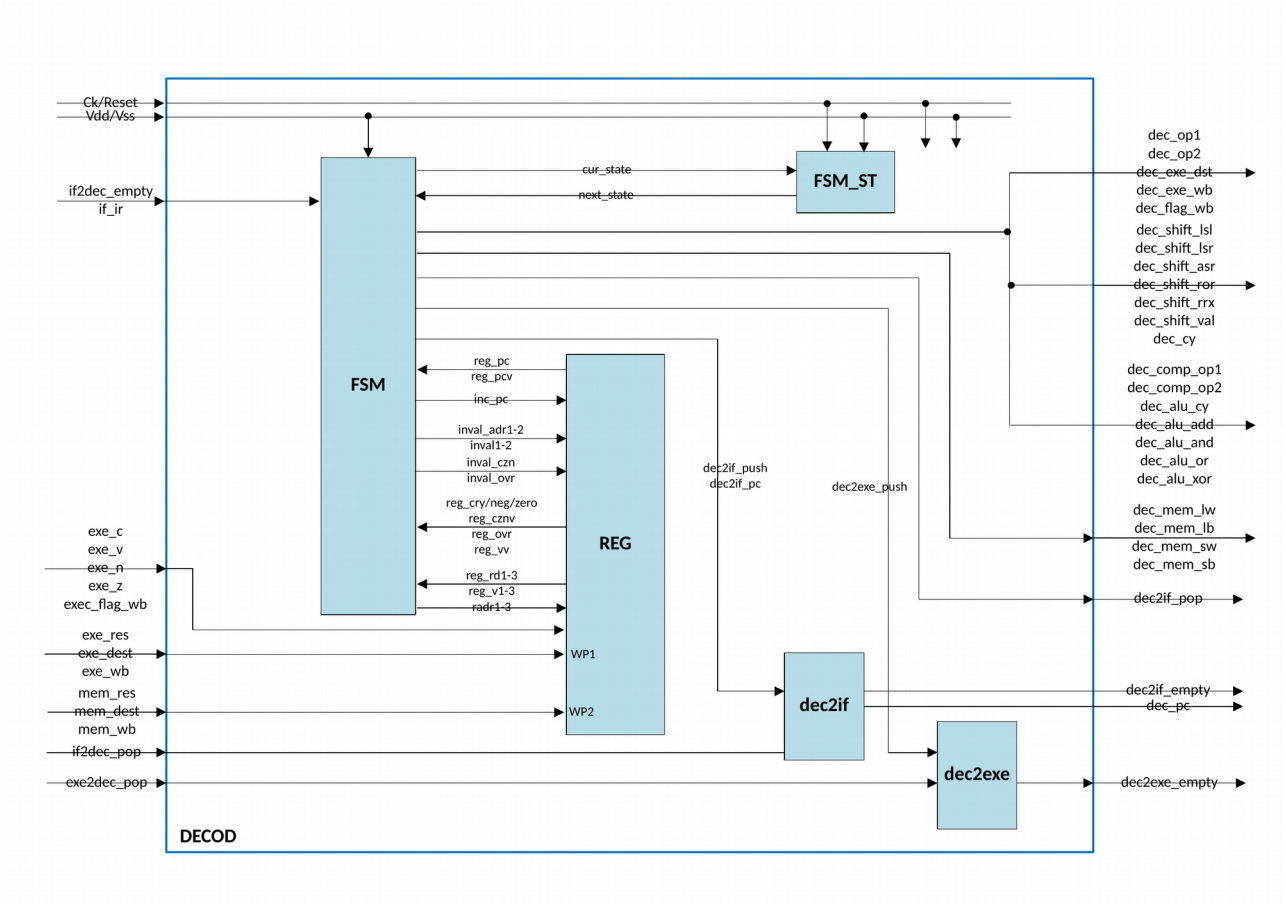
L'entrée de la fifo correspondant à *ic_inst* contient une instruction récupérée dans le cache instruction.
La sortie reliée à *if_ir* permet de communiquer à l'étage suivant l'instruction contenue dans la fifo.
Le push de la fifo reliée à *if2dec_push* permet de changer la valeur contenue dans la fifo,
et pop relié à *dec_pop* indique si la valeur dans la fifo a été lue ou non.

La valeur du registre PC reçue dans *if_adr* est considérée comme valide, si *if_adr_valid* est égale à 1, et doit être envoyée au cache pour récupérer l'instruction qui se trouve à cet endroit-là, puis est envoyée à l'entrée de la fifo.

Si *dec2if_empty* est égal à 0, *ic_stall* est égal à 0 et *if2dec_full* est égal à 0, la valeur contenue dans la fifo changera et sera égale à la valeur en entrée *ic_inst* et sera considérée comme correct.
La valeur contenue dans la Fifo de DECOD sera considérée comme lu et non valide.

III- DECOD / Write Back

DECOD est le deuxième étage, et c'est le plus complexe, de notre processeur. Il sert à décoder l'instruction qu'il reçoit, à envoyer à l'étage suivant toutes les informations dont il aura besoin pour ses calculs, et à gérer le banc de registre.
Le décodage d'une instruction peut prendre plus d'un cycle selon le type de l'instruction.
Un *branch and link* prendra 2 cycles, un cycle servira à mettre à jour l'adresse de retour dans r14, et le second à calculer la nouvelle valeur de PC et à mettre le registre à jour.



Composant(s) de l'étage :

L'étage *DECOD* est composé :

-d'une fifo 1 place de 32 bits, *dec2if* servant à contenir l'adresse de la prochaine instruction à charger.
- d'un fifo 1 place de 129 bit, *dec2exe* contenant le contenue de RD, RS op1 op2 et un bit de validité.
- du banc de registre (voir page VI)

Il prend en entrée :

-- *Write Back interface*
- `exe_dest`, `exe_wb` et `exe_res` qui contiennent le registre de destination sur 4 bits, le bit de validité de la commande et ce qu'il faut écrire.

-- *Mem Write Back interface*
- `mem_dest`, `mem_wb` et `mem_res` qui contiennent le registre de destination sur 4 bits, le bit de validité de la commande et ce qu'il faut écrire sur 32 bits.

-- *CSPR (flags)*
- `exe_c`, `exe_v`, `exe_n`, `exe_z` et `exe_wb` qui contiennent les flags servant au prédicat de chaque instruction, et le bit de mise à jour ('1' -> mettre à jour)

--IFETCH interface
- `if_ir` qui contient l'instruction récupérée dans *IFTECH*

--EXE interface
- `exe_pop` qui indique si l'étage EXE a utilisé et ce qu'on lui a donné ou non.

Il a en sortie :

-- ALU operand
- `dec_op1, dec_op2, dec_exe_dest, dec_exe_wb, dec_flag_wb`

-- ALU operand selection
- `dec_comp_op1, dec_comp_op1`

-- ALU command
- `dec_alu_cy, dec_alu_and, dec_alu_add, dec_alu_or, dec_alu_xor`

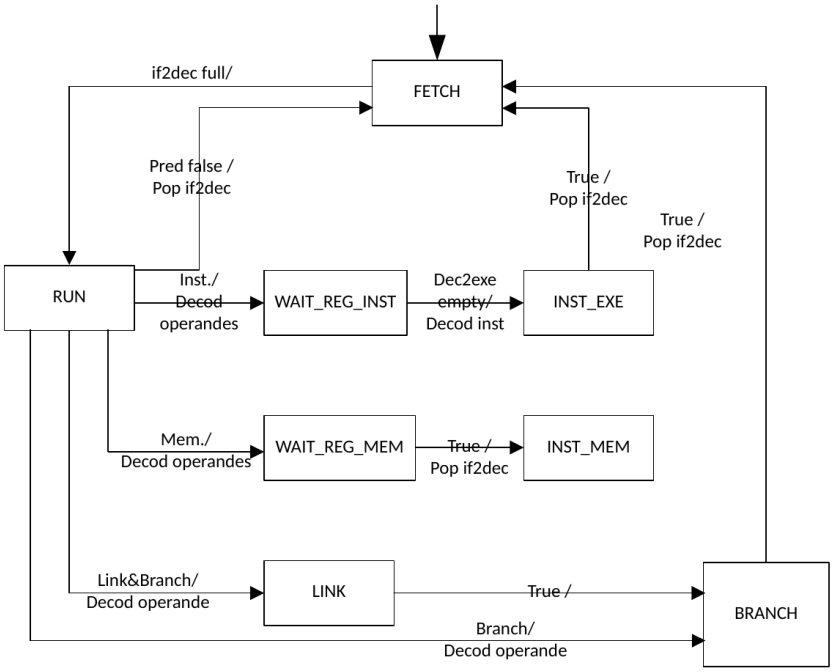
-- data to mem
- `dec_mem_data, dec_mem_dest, dec_pre_index, dec_mem_lw, dec_mem_lb, dec_mem_sw, dec_mem_sb`

-- shifter command
- `dec_shift_lsl, dec_shift_lsr, dec_shift_asr, dec_shift_ror, dec_shift_rrx, dec_shift_val, dec_shift_cy`

-- IFETCH and EXE interface
- *dec2exe_empty, dec2if_empty, dec_pop_out*

Comportement de l'étage

Cet étage est basé sur une machine à état dont le fonctionnement est décrit ci-dessous :



Etat FETCH :
C'est l'état de départ par lequel on commence.
Il vérifie grâce au signal *if2dec_full* que l'instruction qu'il reçoit en entrée est valide ou non, puis passe à l'état *RUN* sinon il reste dans l'état *FETCH* jusqu'à ce qu'il y ait une instruction à lire.

Etat RUN :
Il vérifie que le prédicat de l'instruction soit bon.

Si celui-ci n'est pas correct, il envoie un signal pop à *if2dec* pour obtenir une nouvelle instruction.

Si le prédicat est correct il décode le type d'instruction et demande au registre le contenu des registres dont il a besoin, puis passe dans l'état *WAIT_REG_INST, WAIT_REG_MEM, LINK* ou *BRANCH*

Etat WAIT_REG_INST :
Cet état sert d'attente pour avoir une réponse du registre, il passe à l'état *INST_EXE*.
La réponse sera donnée à la fin du cycle automatiquement.

Etat WAIT_REG_MEM :
Cet état sert d'attente pour avoir une réponse du registre, il passe à l'état *INST_MEM*. La réponse sera donnée à la fin du cycle.

Etat LINK :
Cet état envoie à EXE ce qu'il faut pour stocker dans r14 c'est-à-dire la valeur actuelle *PC + 4* puis passe à l'état *BRANCH*.

Etat BRANCH :

Cet état sert à modifier la valeur du registre *PC* selon ce la valeur d'opérande 2 que l'on a décodé dans l'instruction reçue.
Puis il envoie un pop à *if2dec* pour lui indiquer de charger l'instruction suivante et repasse dans l'état *FETCH*.

Etat INST_EXE :

Dans cet état on a décodé toute l'instruction. Il ne reste plus qu'à envoyer à *EXE* les bonnes informations pour l'exécution du calcul.
Une fois le passage d'information à *EXE* fait, il envoie un pop à *if2dec* pour lui indiquer de charger l'instruction suivante et repasse dans l'état FETCH.

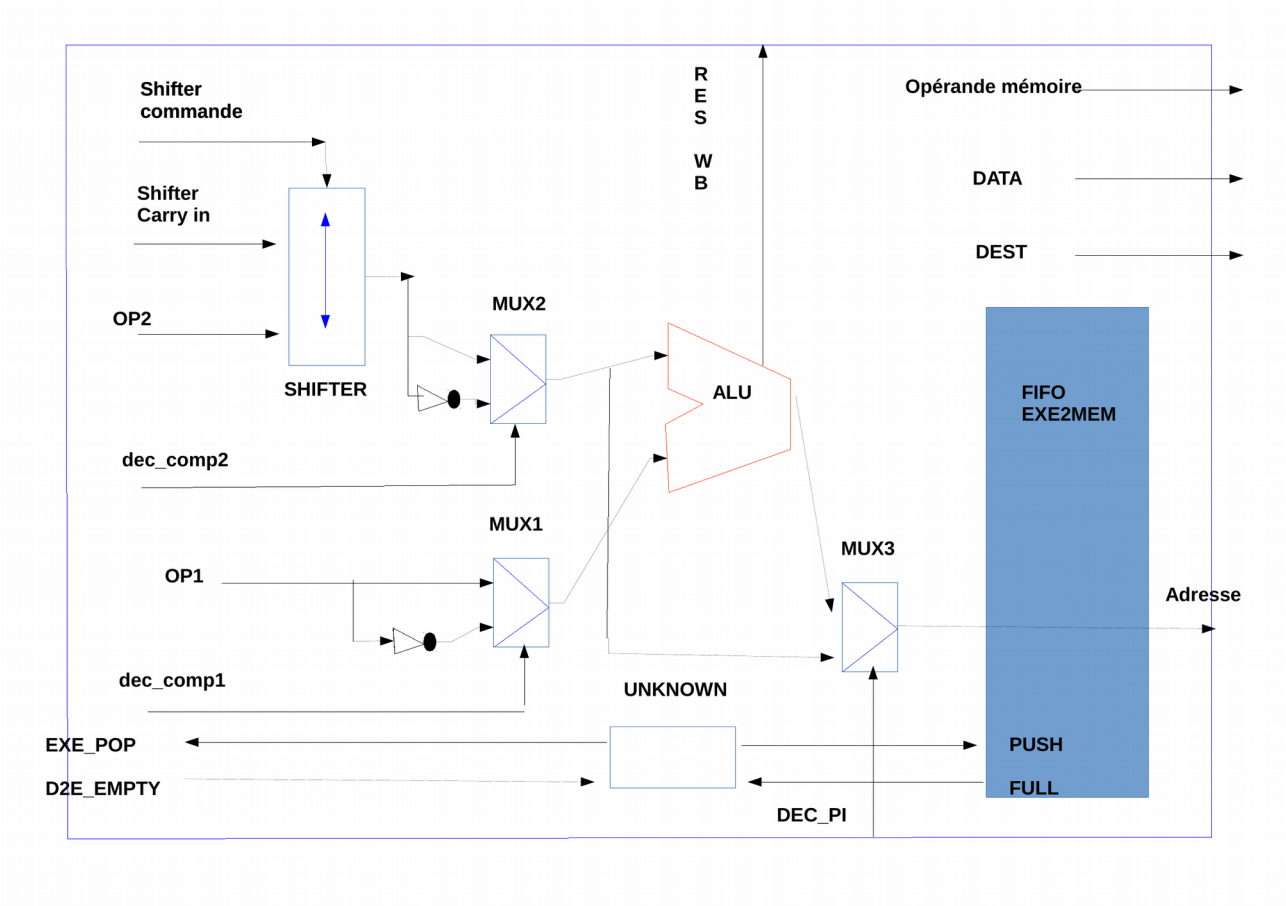
Etat INST_MEM :

Dans cet état on a décodé toute l'instruction, il ne reste plus qu'à envoyer à *EXE* les bonnes informations pour l'exécution du calcul d'adresse et à *MEM* les bonnes informations de *load* ou de *store* que l'on veut faire.

Une fois les informations transmises, il envoie un pop à *if2dec* pour lui indiquer de charger l'instruction suivante et repasse dans l'état *FETCH*.

IV EXEC

EXEC est le troisième étage de notre processeur. Il sert à faire tous les calculs et opérations logiques dont on a besoin, il possède aussi un bypass de *EXEC* vers *EXEC*. Il renvoie de nouvelles valeurs pour les flags qui seront potentiellement utilisés, et donne à l'étage **MEM** l'adresse de ce que l'on veut charger ou écrire en mémoire.



Composant(s) de l'étage :

L'étage *EXEC* est composé :

- d'une fifo 1 place de 72 bits, *dec2if* qui contient les informations suivantes : commande, data, dest et la sortie du 3eme multiplexeur.
- de 3 multiplexeurs.
- d'un composant « unknown » servant à gérer la fifo.
- d'un shifter. (voir VIII)
- de l'ALU. (voir VII)

Il prend en entrée :

-- ALU operand (entrée de l'alu)
- `dec_op1, dec_op2, dec_exe_dest, dec_exe_wb, dec_flag_wb`

-- ALU operand selection (commande servant à inverser les valeurs en entrée)
- `dec_comp_op1, dec_comp_op1`

-- ALU command (le type d'operation à réaliser dans l'alu)
- `dec_alu_cy, dec_alu_and, dec_alu_add, dec_alu_or, dec_alu_xor,` *dec_PI*

-- data to mem (les informations à envoyer à l'étage MEM)
- `dec_mem_data, dec_mem_dest, dec_pre_index, dec_mem_lw, dec_mem_lb, dec_mem_sw, dec_mem_sb`

-- shifter command (le type de decalage du shifter et la valeur qu'il prend en entré)
- `dec_shift_lsl, dec_shift_lsr, dec_shift_asr, dec_shift_ror, dec_shift_rrx, dec_shift_val, dec_shift_cy`

-- MEM and EXE interface (la gestion des fifos interne)
- dec2exe_empty, mem_pop

Il a en sortie :

-- Write Back du resultat de EXE
- exe_res, exe_dest, exe_wb

-- FLAGS (les flags générés par la dernière opération réalisée)
- exe_c, exe_z, exe_n, exe_v, exe_flag_wb

-- MEM interface (les informations à envoyer à l'étage MEM)
- `exe_mem_adr, exe_mem_data, exe_mem_dest, exe_mem_lw, exe_mem_lb, exe_mem_sw, exe_mem_sb, exe2mem_empty`

-- EXE interface (gestion des fifos internes)
- *dec2exe_empty*

Comportement de l'étage

EXEC est l'étage servant à réaliser toutes les opérations arithmétiques et logiques dont a besoin une instruction.

Il commence par préparer les deux opérandes qui iront dans l'*ALU*.
La première Op1 ne subit qu'un traitement, elle entre dans l'*ALU* de la même façon que dans l'étage ou bien inversé selon la commande *dec_compop1* qui gère le premier multiplexeur.

La deuxième passe d'abord par le *shifter* puis le résultat est inversé ou non selon la valeur de *dec_compop2* et rentre dans l'*ALU*.

L'*ALU* reçoit directement le type d'opération et les flags de l'étage précédent, puis sort le calcul.
Ce résultat est envoyé à deux endroits :
-D'un côté il sort de l'étage pour le *Write Back*
-De l'autre il rentre dans un multiplexeur à deux entrées : la sortie de l'ALU et l'opérande 1 et il en ressort la valeur de *dec_PI* qui désigne s' il y a une post indexation ou non.

Le résultat est ensuite envoyé à l'étage suivant comme adresse.

La gestion de la *fifo* est réalisée par le petit composant « *unknown* » qui remplit et vide les deux *fifos*.

V- MEM

MEM est le quatrième et dernier étage de notre processeur.

Il sert à préparer les opérandes avant de les écrire dans la mémoire, et envoie toute les informations au cache de données, ainsi qu'à *DECOD* en cas de *WB*.

Composant(s) de l'étage :

- Il n'a aucune composant.

Il prend en entrée :

-- EXE interface (les informations envoyées par EXE sur le type d'instructions à réaliser dans cet étage)

- `exe_mem_adr, exe_mem_data, exe_mem_dest, exe_mem_lw, exe_mem_lb, exe_mem_sw, exe_mem_sb, exe2mem_empty`

-- Dcache interface (gestion du cache de donnée)

- `dc_data, dc_stall`

Il a en sortie :

-- MEM WB (le write back de MEM vers DECOD)

- mem_res, mem_wb, mem_dest

-- DCache interface (les informations pour l'utilisation du cache de données)

- mem_adr, mem_stb, mem_stw, mem_load

-- EXE interface (gestion de la fifo)

- *exe2mem_empty*

Comportement de l'étage

MEM commence par regarder le type d'instruction à réaliser.

Il faut savoir quel octet du mot (4 octets) on veut récupérer dans le cas d'un *load byte*. On regarde pour cela l'adresse, selon ces deux derniers bits on est capable de savoir quelle partie nous intéresse et on stocke cet octet dans un signal interne *lb_data* qui complétera les 3 octets restant par des 0.
Cette opération est la première effectuée et est réalisée à chaque fois, peu importe le type de l'instruction à réaliser.

On prépare ensuite les informations du *write back*, si c'est nécessaire.
mem_res contient *lb_data* si la commande en entrée demande un *lb* et contient un octet sur les quatre les autres sont mis à 0, sinon dans *dc_data* on obtient la réponse directe du cache qui contient le mot entier soit les 4 octets du mot récupéré en mémoire.
Le registre de destination est celui qui est passé en entrée avec *exe_mem_dest*.

Le *Write Back* sera fait dans *DECOD* seulement si une instruction de chargement mémoire est demandée en entrée, qu'il y a de la place dans la fifo de *EXE* vers *MEM* et que la valeur du mot renvoyée par le cache de donnée est valide.

La gestion du cache de données

Avant d'envoyer une adresse à charger ou écrire dans le cache, il faut l'aligner sur un multiple de 4 pour charger le bon mot complet, dans le cas d'une instruction *load*, on remplace les 2 derniers bits par « 00 » pour aligner notre adresse.
Dans le cas d'une instruction *store*, il n'aura pas besoin de récupérer le résultat de mem_res, il ira écrire directement dans la mémoire.

Si notre instruction est une écriture mémoire, la sortie correspondant au store *word* ou *byte* sera mise à 1.
La valeur à écrire est placée dans *mem_data* et vient du signal en entrée *exe_mem_data*.

Le cache nous renverra la valeur lue en mémoire dans le cas d'un *load* ou sinon ira écrire dans la mémoire la valeur voulue dans l'adresse donnée.

VI- Banc de Registre

Le banc de registre contient les 16 registres accessibles par l'utilisateur. Il est là pour gérer la cohésion des registres entre les différentes écritures et lectures.

Il comporte 2 ports d'écritures et 4 de lectures, un port d'écriture des flags, et un accès instantané au registre PC.

Pour écrire :

Avant d'écrire une valeur dans un registre, il faut commencer par invalider la valeur contenue dans celui-ci pour ne pas qu'une autre personne vienne lire la valeur qu'il contient alors que celle-ci n'est pas encore modifiée. Il faut mettre dans inval_adr*numport* le numéro de registre dans lequel on veut écrire et dans inval*numport* la valeur 1.

Ce travail est fait dans *DECOD* quand il détecte qu'une instruction va écrire dans un registre.

Une fois la valeur du registre invalidée, on peut commencer à écrire. Il faut mettre le numéro de registre dans *wadrnumport*, la valeur de ce que l'on veut écrire dans wdata*numport*, et '1' dans wen*numport*.
La valeur du registre sera mise à jour au prochain cycle.

Ce travail est fait dans *DECOD*, les ports d'écriture sont remplis par les entrées servant au *Write Back* et au *Write Back* des instructions mémoires.

--> Modifier les flags :

Pour modifier les flags il faut procéder de la même façon, invalider les flags que l'on veut modifier, dans *inval_czn* si l'on veut modifier un de ces 3 flags ou *inval_ovr* pour l'overwrite.

Ce travail est fait dans *DECOD* quand il détecte qu'une instruction demande à modifier la valeur des flags à la fin de son exécution.

Puis écrire la nouvelle valeur des flags dans *wcry*, *wovr*, *wzero* et *wneg* avec *cspr_wb* comme bit de validité de la commande.

Ils rentrent dans le banc de registre par les entrées de DECOD correspondant aux flags.

Pour lire :

Pour lire le contenu d'un registre c'est plus simple, il suffit de mettre le numéro de registre dans radr*numport* puis on récupérera dans rdata*numport* le contenu du registre et dans dvalid*numport* la validité du registre.

VII- ALU

L'*ALU* est l'unité de calcul du processeur, il sert à faire toutes les opérations. Il est capable de faire des additions et les opérations logiques de bases and, or et xor.

On lui donne en entrée les deux opérandes qui contiennent les valeurs, la carry s' il y en a besoin, le type d'opération à réaliser et il nous rend un résultat et les nouveaux flags que cette opération a généré.

On a pris comme condition que les opérations ne sont effectuées que sur des entiers codés en complément à 2.

Pour ce type d'opération il ne peut y avoir qu'un fil de commande à '1' sinon l'*ALU* ne fait rien.

Aucun flag n'est généré pour les opérations logiques.

VIII – Shifter

Le *shifter* est le composant qui permet d'effectuer un décalage sur une valeur de 32 bits passée en entrée. Il permet de faire des décalages logique gauches et droits, arithmétique droit et des rotations.
Il retourne la nouvelle valeur du type de décalage voulu et décalé de la valeur passée, et génère une carry.

Il prend en entrée une valeur sur 32 bits qui sera décalée, une carry, le type de décalage voulu et la valeur du décalage voulu.

IX – Synthétisation - Placement/Routage

Une fois notre processeur crée et complet, nous allons pouvoir le synthétiser pour pouvoir récupérer le dessin des masques qui pourra partir en fonderie.

Voici la chaine à exécuter pour synthétiser un composant, elle devra être effectuée sur CHAQUE composant :

```
$< vasy -p -I vhdl -V -o -a composant.vhdl composant
$< boom -V composant composant_o
$< boog composant  composant
```

Si le résultat rendu par boom est moins performant que ce que l'on avait de façon originale, il faut relancer cette chaine en sautant la partie boom.

Il va falloir ensuite effectuer un Placement/Routage qui va permettre d'avoir le dessin des masques finaux, ils s'obtiennent de cette façon et on devra le faire sur CHAQUE composant :

```
$< cgt nom_fichier.vst
```

X-Réussi / Pas Réussi

Nous avons réussi à créer tous les composants et à les connecter entre eux.

En donnant les bons paramètres en entrée de *arm_core*, on obtient le bon résultat d'exécution à la fin de la chaine et la bonne gestion des registres, mais tous les composants ne synthétisent pas.

J'ai rencontré des difficultés sur la compréhension et la gestion des différentes FIFO qui servent à communiquer entre les étages. J'ai passé beaucoup de temps à analyser le sujet pour pouvoir réaliser le processeur.

La synthétisation des différents composants a aussi posé de nombreux problèmes, Il m'a fallu revoir la façon dont les composants sont créés et gérés pour les simplifier et permettre leur synthétisation.
Certains composants ne synthétisent pas complétement.