SORBONNE UNIVERSITÉ
M1 SESI

Processeur ARMv2a

RAPPORT DE D'AVANCEMENT DU PROJET DE VLSI

Etudiants:
M. Samy Attal - 3802981
M. Kevin Lastra - 21103305

Table de Matières

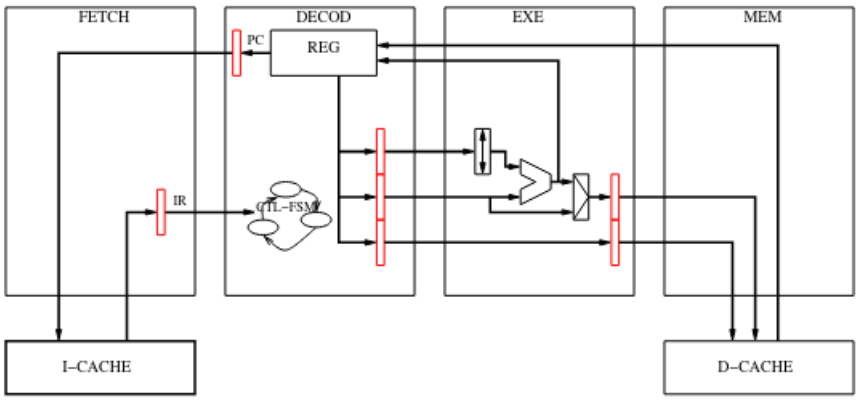
1 Introduction

Sous la direction de M. DESBARBIEUX Jean-Lou, notre binôme, composé de Samy Attal et Kevin Lastra, a travaillé sur l'implémentation du jeu d'instructions ARMv2a en VHDL comme projet de l'UE VLSI.

On a choisi ici de concevoir une architecture pipeline original de quatre étages:

- FETCH
- DECOD
- EXEC
- MEM

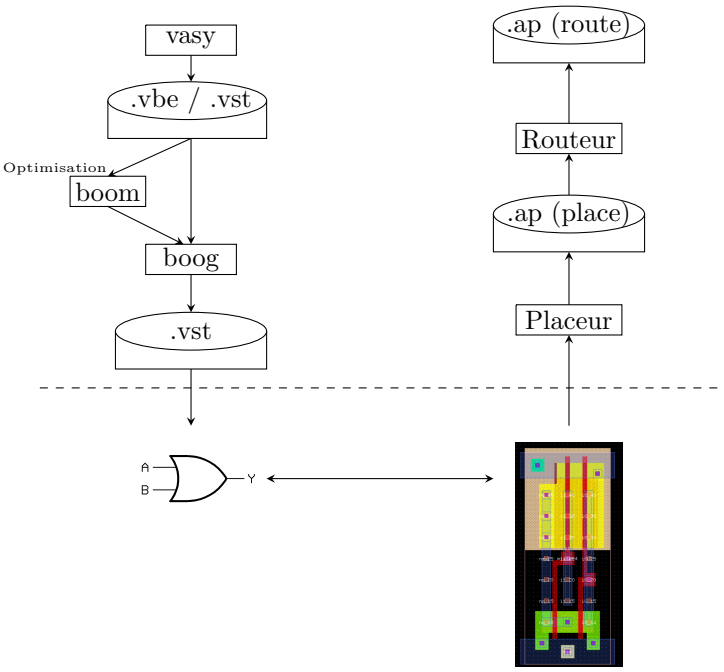
Nous relevons ici que le mode d'adressage initial est sur 26 bits alors que nous le faisons sur 32 bits, ce qui nous rapproche un peu plus du jeu ARMv3. Notre processeur aura aussi un cache d'instructions et un cache de donnée respectivement relié à fetch et mem.



1.1 Outils de travail

Nombreux outils diffèrent en été utiliser pour la réalisation du projet:

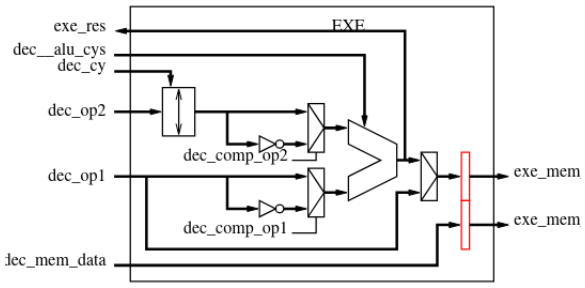
- Emacs et Visual studio code, ces éditeurs de texts sont des outils fiables et versatiles.
- Outils Alliance
  - vasy (VHDL **A**nalyser for **SY**nthesis)
    Un magnifique translateur du VHDL ieee vers un VHDL sous format vbe ou vst acceptable par les outils de synthèses suivant.
  - boom (**BOO**lean **M**inimizer)
    Optimisateur booléen qui fonctionne toujours très bien.
  - boog (**B**inding and **O**ptimizing **O**n **G**ates)
    Synthétiseur qui prend en sources des fichiers VHDL sous format vbe ou vst et grâce à une bibliothèque SXLIB les synthétise en différentes cellules logiques.
  - Xsch (**G**raphical **S**chematic **V**iewer)
    Visualiseur de schéma de cellules logiques (SXLIB).



2 Travail Réalisé

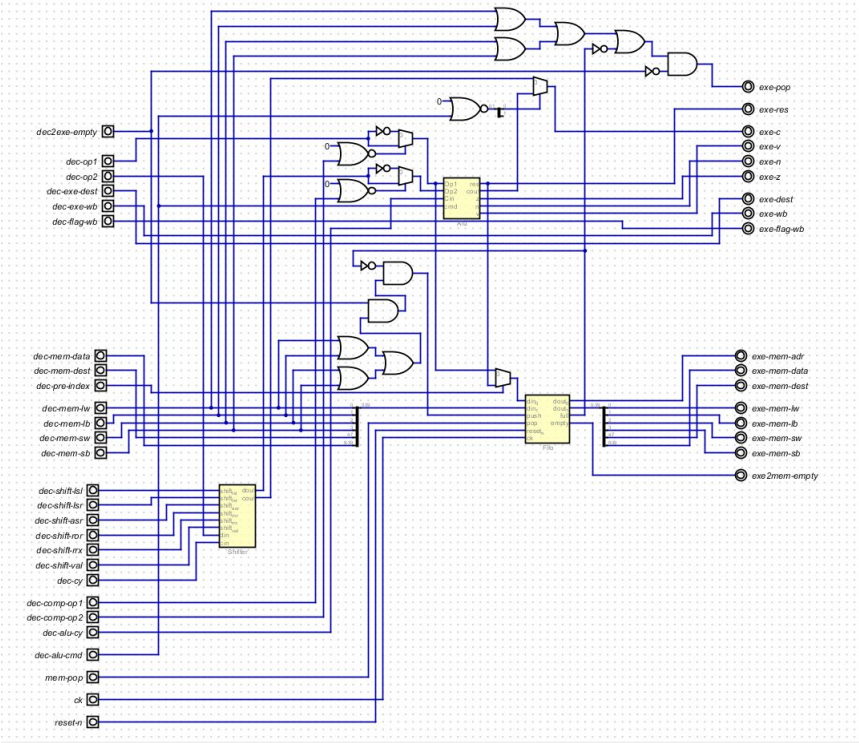
2.1 Étage EXEC

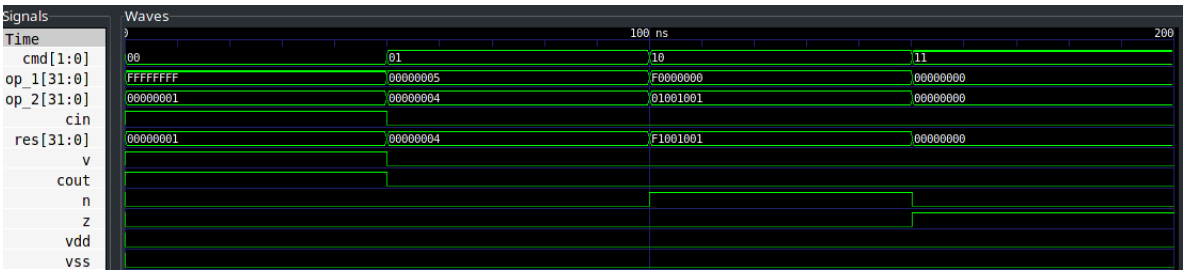
L'étage exec est la partie où les instructions vont pouvoir être exécutées. Il se compose principalement d'une ALU, d'un shifter et d'une fifo que nous développerons ci-dessous.



2.1.1 Fonctionnement

Exec va donc permettre d'effectuer les différents calculs du processeur. Cet étage prend de decod (l'étage precedent) les opérandes et toutes les informations necessaires au traitement de ces opérandes.
En effet, l'ALU et le shifter doivent être controlés. Le résultat de l'ALU doit pouvoir être retransmis à decod s'il s'agit d'une écriture dans un registre, ou vers la FIFO vers l'étage mem si l'instruction est un accès mémoire (l'adresse étant calculée par l'ALU).
Il est aussi important de noter qu'on peut effectuer l'opération de soustraction grâce à des inverseurs sur chaque opérande dont le résultat peut être choisi avec un multiplexeur.
Les commandes des FIFO (à l'interface de decod et de mem) sont réalisées par des fonctions combinatoires (voir schéma ci-dessous).



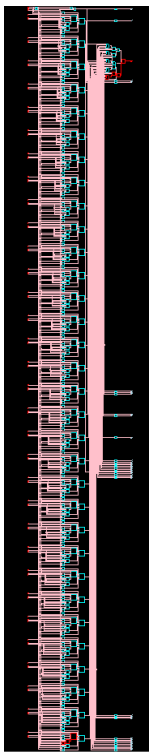


Test Bench de l'ALU

2.1.2 Composants

- **ALU**

L'unité arithmétique logique (ALU), est un circuit qui réalise des opérations arithmétiques et des opérations logiques. Elle sert notamment à faire des calculs de données ainsi que des calculs d'adresse dans le cas de transfert mémoire et branchement.
Notre implémentation de l'ALU permet de faire quatre types d'opérations élémentaires tels que addition sur 32 bits, AND logique, OR logique et XOR logique. On pourra faire une soustraction grâce au complement à 2 effectué par un inverseur situé en amont de l'ALU (voir schéma d'exec cours 5 p10) . Nous avons conçu un additionneur 32 bits qui est basiquement une instanciation de 32 fulladder de 1 bit. Ce qui nous fait un additionneur à décalage de retenue, qui n'est pas forcément le plus optimale niveau temps de propagation (étant donné qu'il faut attendre toute la propagation de la retenue afin d'avoir un résultat correct) mais on étudiera éventuellement une optimisation dans le futur.
schema



Synthèse de l'ALU

Nous remarquons sans surprise que le chemin critique de notre circuit est le flag Z, élégamment representé en rouge avec xsch, ce qui semble logique étant donnée la propagation de la retenue afin d'obtenir la valeur complète sur 32 bits.

Nous avons décidé de valider notre ALU en testant en simulation ses quatres opérations élementaires. Nous verifions également la bonne valeur des flags (Négatif (N), Zéro (Z), Carry out C) et Overflow (V)), voici le résultat de notre simulation :

```
no2_x1: 96
buf_x2: 35
a3_x2: 33
a2_x2: 33
na2_x1: 33
oa2ao222_x2: 32
noa2ao222_x1: 32
nxr2_x1: 32
nao22_x1: 32
o4_x2: 5
no3_x1: 3
o3_x2: 2
inv_x2: 2
no4_x1: 1
an12_x1: 1
Total: 372
```

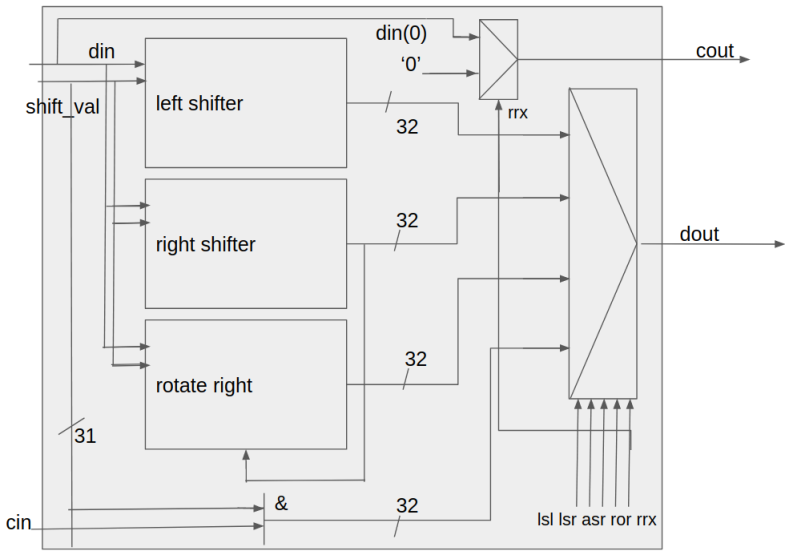
Nombre de portes logiques de l'ALU
(sans compter l'instanciation du adder 32 bits)

- ## Shifter

Un shifter précède l'ALU et sert à effectuer différents types de décalages (logiques à droite ou gauche, arithmétique à droite, rotation à droite et rotation avec retenue).

Ce qui permet d'avoir plus de flexibilité sur le deuxième opérande fournit à l'ALU ainsi que d'avoir des instructions de type immédiat contenant des valeurs sur 32 bits.

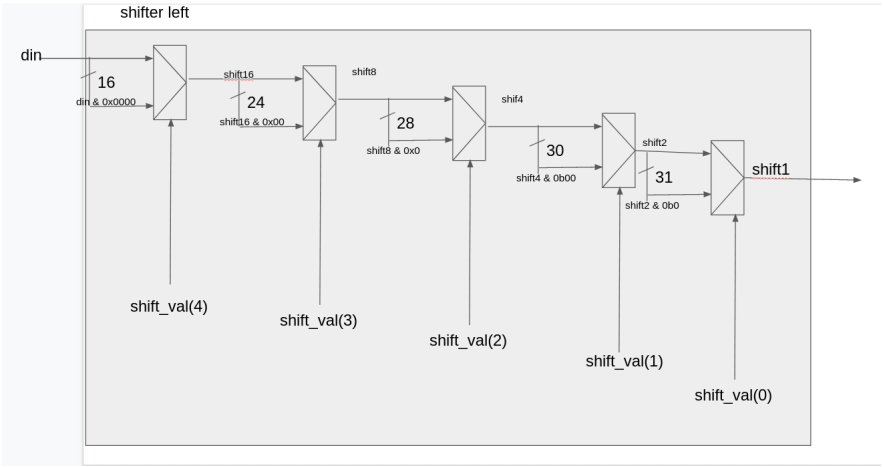
Notre implémentation du shifter consiste à une instanciation de quatre sous shifter telles que right shifter, left shifter, arithmetic shifter et rotate right qui effectue en parallèle les différentes opérations de décalage en fonction d'un signal sur cinq bits qui détermine la valeur du décalage ou de rotation. La sortie du shifter est sélectionnée par un multiplexeur commandé par le type de décalage (logical shift left, logical shift right, arithmetic shift right, rotate right, rotate extended).



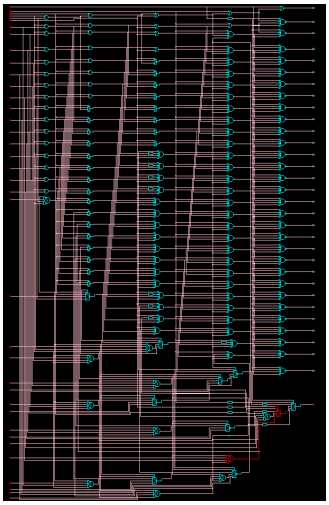
Graph shifter

– **Shift right / left (logic or arithmetic):**
Nous avons decidé de designer les opérations de shift logique ou arithmetique ainsi. Il s'agit de cascader les différents signaux (décalés ou pas) en fonction de chaque bit du signal de valeur de décalage.

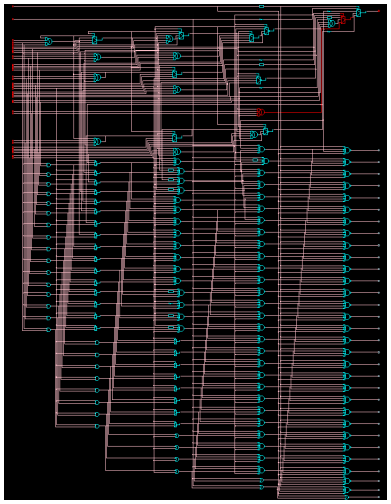
* Le bit de poid fort determinera si on effectue un décalage de 16 ou non du signal d'entrée.
* Le quatrième bit determinera si on effectue un décalage de 8 ou non du signal decalé ou non de 16 bits.
* Le troisème bit determinera si on effectue un décalage de 4 ou non du signal decalé ou non de 8 bits.
* Le deuxième bit determinera si on effectue un décalage de 2 ou non du signal decalé ou non de 4 bits.
* Le bit de poid faible determinera si on effectue un décalage de 1 ou non du signal decalé ou non de 2 bits.
* Et ce dernier signal est le signal affecté en sortie d'un des composants du shifter.



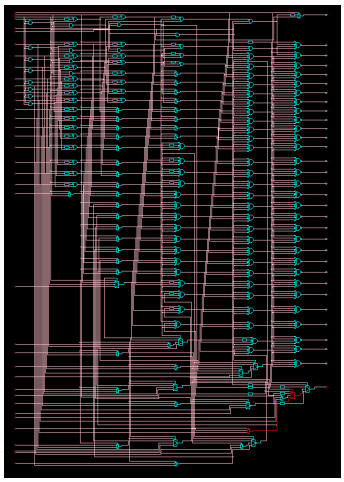
Left shifter



Synthese Right shifter



Synthese Left shifter



Synthese Arithmetic shifter

```
ao2o22_x2: 44
nao2o22_x1: 30
an12_x1: 24
nmx2_x1: 20
inv_x2: 16
oa2a22_x2: 11
mx3_x2: 7
mx2_x2: 4
on12_x1: 4
nmx3_x1: 3
o2_x2: 2
no2_x1: 1
Total: 166
```

Nombre de portes logiques du Right shifter

```
ao2o22_x2: 44
nao2o22_x1: 30
an12_x1: 24
nmx2_x1: 20
inv_x2: 16
oa2a22_x2: 11
mx3_x2: 7
mx2_x2: 4
on12_x1: 4
nmx3_x1: 3
o2_x2: 2
no2_x1: 1
Total: 166
```

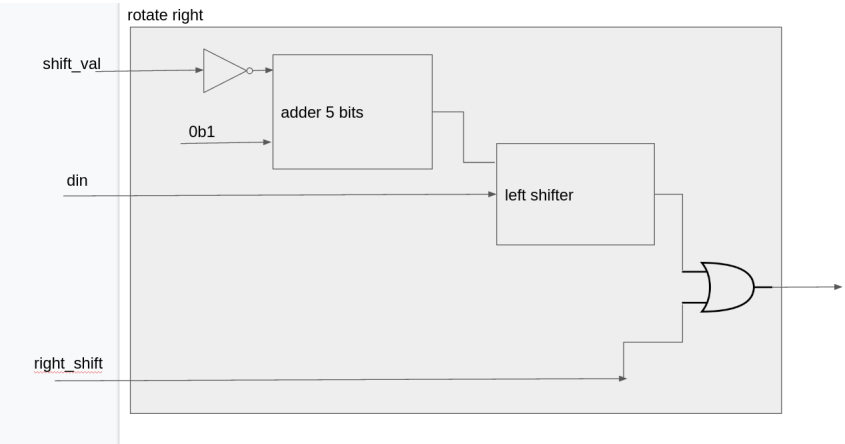
Nombre de portes logiques du Left shifter

```
inv_x2: 44
ao2o22_x2: 44
nao2o22_x1: 30
nao22_x1: 24
nmx2_x1: 20
mx2_x2: 16
na2_x1: 15
mx3_x2: 7
ao22_x2: 6
nmx3_x1: 3
Total: 209
```

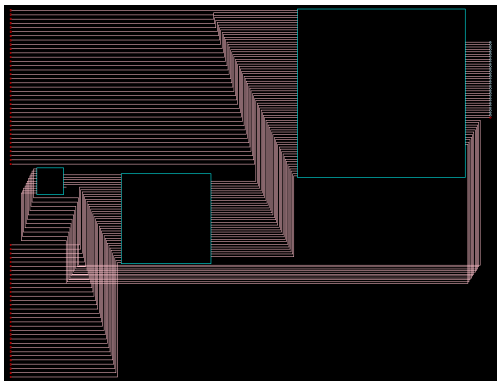
Nombre de portes logiques du Arithmetic shifter

Ici nous representons le fonctionnement pour le left shifter, le même principe s'applique donc pour les autres sous shifter. Le right shifter aura en entrée du premier multiplexeur le signal din et la concatenation des 16 bits de poid forts et 16 bits à 0, et ainsi de suite.
Le shifter arithmetique reprend reprend le right shifter mais avec une concactenation des bits de poid fort de din au lieu des bits à 0.

- **Rotate right:**
  Nous avons été très inspirés pour ce composant. En effet afin d'illustrer notre implémentation nous considerons un signal d'entrée (valeur arbitraire) 0xABCD EFGH, auquel nous faisons subir une rotation à droite de 4.
  Nous sommes donc censé obtenir en sortie 0xHABC DEFG.
  Nous reprenons alors le décalage fournit par le right shifter qui donne 0x0ABC DEFG.
  Il nous faut ensuite décaler le signal d'entrée à gauche de l'inverse de la valeur de décalage (donc 28), ce calcul est effectué par une additionneur 5 bits instancié.
  D'ici est fait un décalage à gauche par une instanciation d'un left shifter qui prend la sortie de l'additionneur pour décalage. Nous avons donc en sortie de ce shifter la valeur 0xH000 0000
  Ensuite il ne reste qu'à effecter un OR logique entre ces deux signaux pour recupérer la valeur de la rotation à droite.
  Le rotate right extended consiste en une concatenation de la retenue d'entrée et les 31 bits de poid faible du signal d'entrée. La retenue de sortie est dans ce cas le bit de poid faible du signal d'entreé.



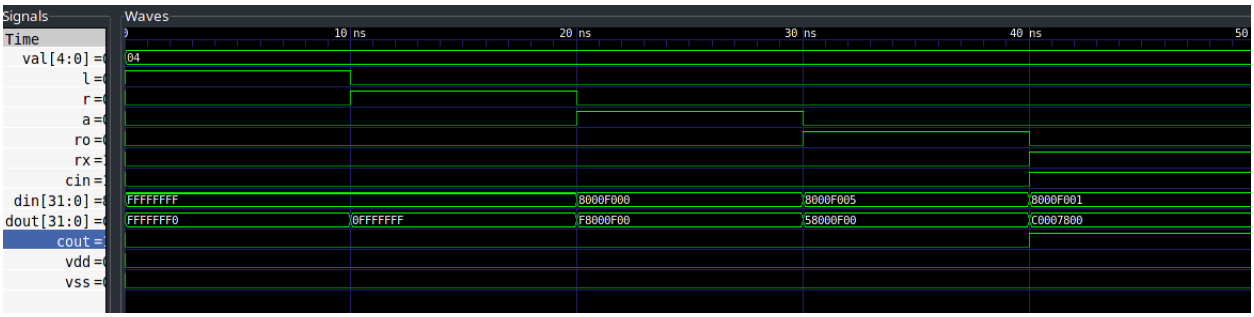
Rotate right (shifter)



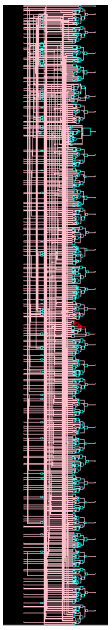
Synthese rotate shifter

```
o2_x2: 32
zero_x0: 5
inv_x2: 5
one_x0: 1
Total: 43
```

Nombre de portes logiques du shifter rotate
(sans compter les instanciations du left shifter et du adder 5 bits)



Test bench du shifter



Synthèse du shifter

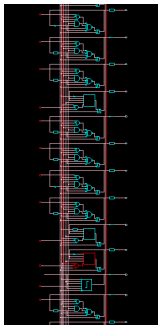
```
mx3_x2: 64
oa2a22_x2: 32
inv_x2: 5
a2_x2: 1
Total: 102
```

Nombre de portes logiques du shifter
(sans compter l'instanciation des sous shifter)

Nous avons simuler notre shifter sur les différents décalages qu'il est capable d'effectuer.
Voici notre test bench :

- FIFO (exec to mem)
  Nous avons besoin d'une FIFO (First In First Out) afin de connecter les étages exec et mem. Cette fifo va permettre la communication direct entre exec et decod vers le cache de donnée dans le cas d'accès mémoire (load ou store).
  Ce composant, cadencé par une horloge, prend en entrée (din) une donnée sur un certain nombre de bits, dans notre cas 72 bits, et une sortie (dout) de même taille. Il est commandé par deux signaux sur un bit : push et pop et possède deux flags full et empty.
  Son utilité est de synchroniser les différents étages du processeur (ici l'interface exec / mem).
  Le signal push sert à memoriser l'entée din dans la FIFO, le signal pop sert à écrire le signal mémorisé sur le port dout.
  Le flag empty indique s'il est possible d'écrire dans la FIFO et comme dans notre cas on a une profondeur

de 1, si la donnée mémorisée est valide ou non.
Le flag full indique que la donnée mémorisée dans la FIFO est valide mais qu'en cas d'écriture (push) on perdra la donnée (ce qui n'est pas souhaitable).
Le signal transmis par la FIFO exec vers mem sert alors dans le cas d'accès mémoire, exec envoie à mem toutes les informations utiles à son deroulement. Notament si l'accès mémoire est un load ou store, la donnée à écrire si c'est un store, ainsi que l'adresse (dans les deux cas) et la destination d'écriture dans registre dans le cas d'un load.

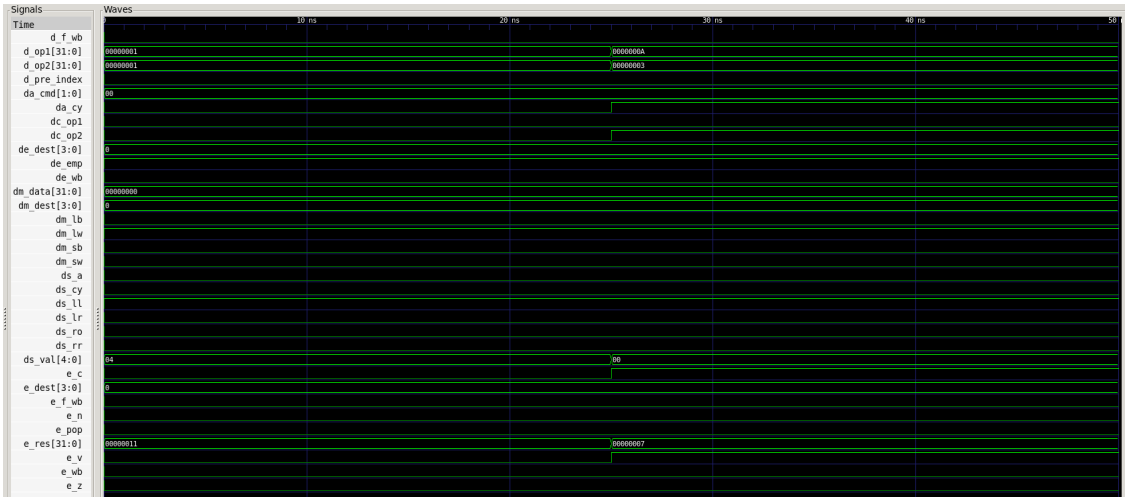


Synthese Fifo

2.1.3 Test Bench

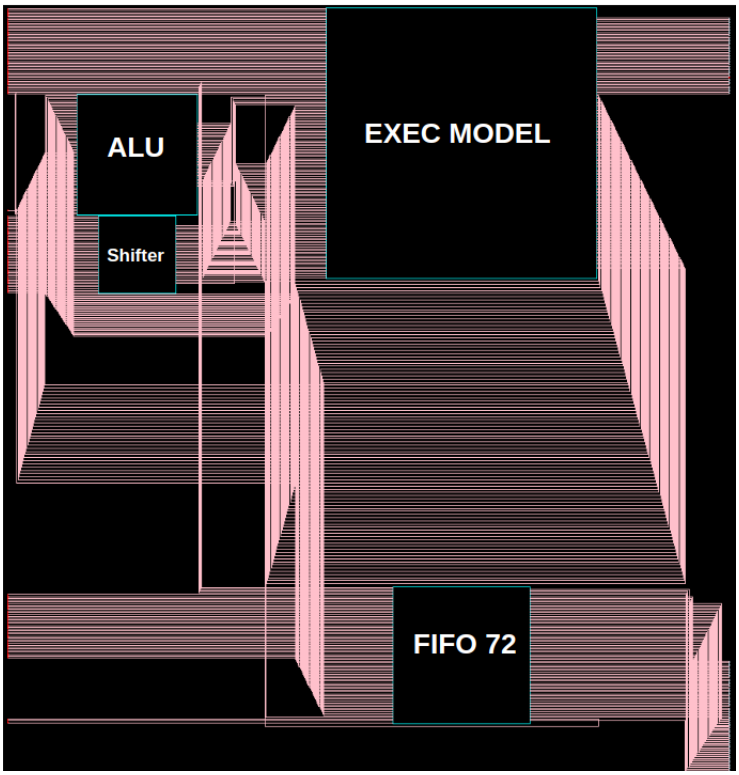
Nous testons notre étage exec en lui faisant faire de simples calculs :

- Dans un premier temps nous lui avons fourni les deux opérandes égale à 1, en commandant le shifter de sorte à ce qu'il fasse un décalage logique de 4 vers la gauche (de l'opérande 2) et en commandant l'ALU afin qu'il réalise une addition.
  Nous avons donc bien 0x10 + 0x1 = 0x11 (e res).
- Dans un second temps nous essyons d'effectuer 0xA - 0x3. Dans ce cas nous devons faire le complément à 2 de l'opérande 2 (donc dec comp op2 = 1 qui réalise l'inversion et introduire une retenue en entrée de l'ALU).
  Nous avons bien 0xA - 0x3 = 0x7.
  Les flags sont également corrects.



Test Bench d'Exec

2.1.4 Synthese



Synthese Exec

```
inv_x2: 67
mx2_x2: 64
buf_x2: 41
oa2a22_x2: 32
no2_x1: 2
no4_x1: 2
o2_x2: 1
oa2ao222_x2: 1
noa22_x1: 1
Total: 211
```

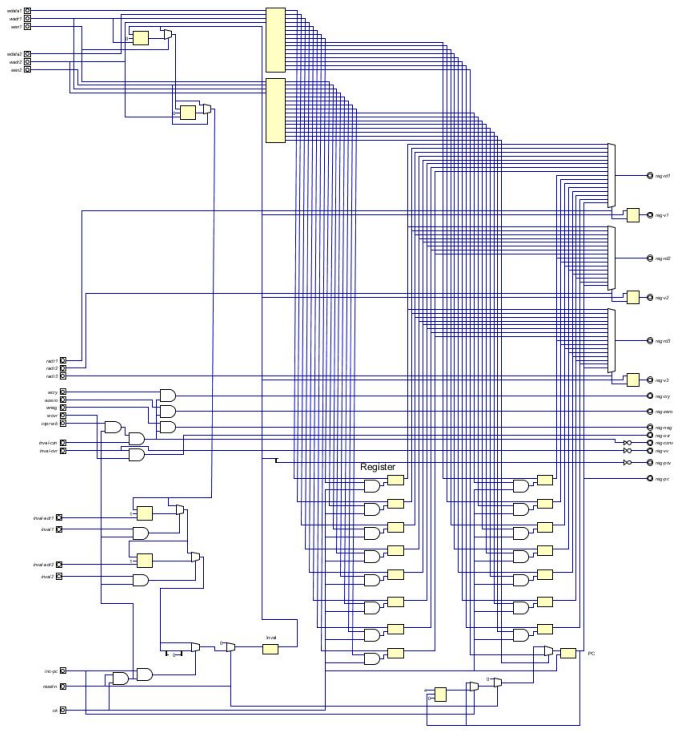
Nombre de portes logiques d'exec

2.2 Étage DECOD

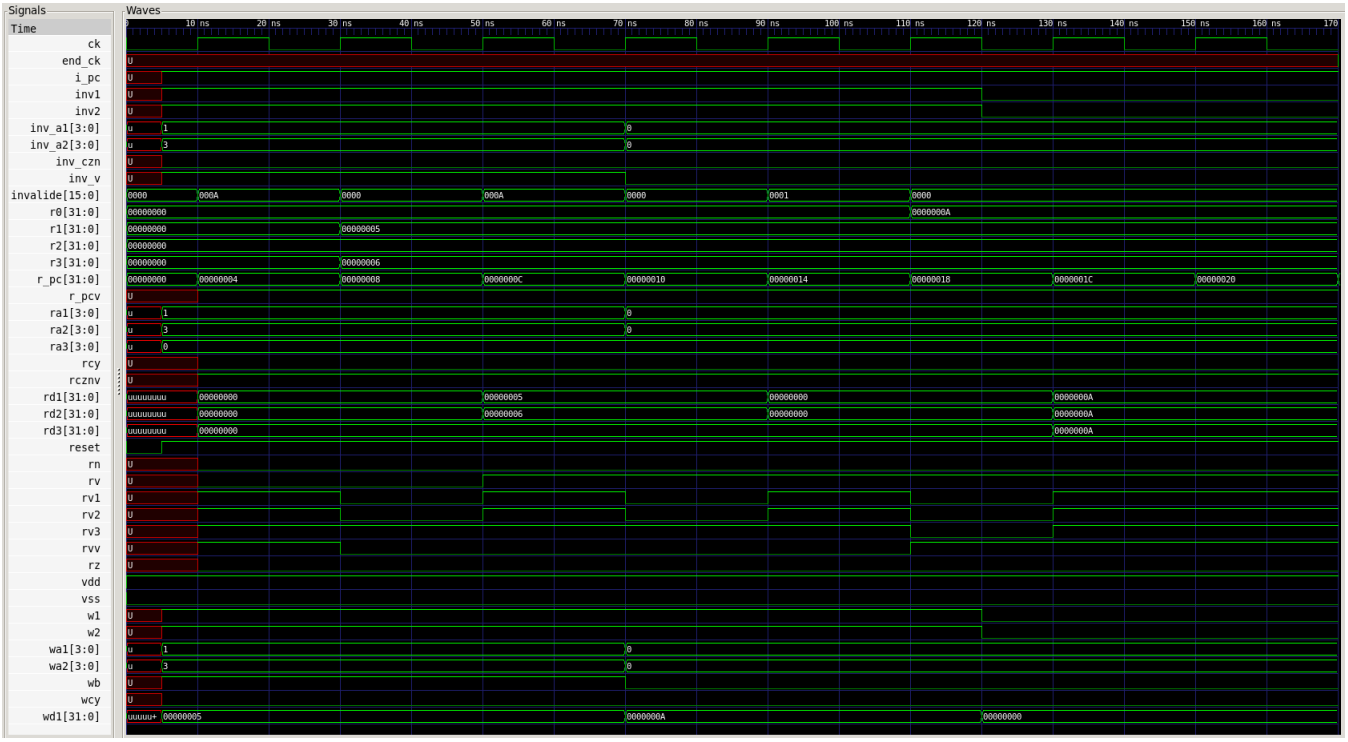
L'étage decod sert comme son nom l'indique à décoder les instructions fournit par le fetch. Il est composé du banc de registres et d'une machine à états qui gère ainsi ses fifo et donc les instructions à executer dans exec. Différentes fifo servent à l'envoi vers exec des différentes opérandes, calculs et adresses et envoi vers fetch du PC (Program Counter), et réception de l'instruction à être exécutée par le biais de fetch.

2.2.1 Composants

- Banc de registres
  Notre implémentation du banc de registres comporte les seize registres ainsi que le registre des flags CNZ (Carry, Negative, Zero) et un registre pour le flag V (Overflow). Un vecteur de seize bits permet d'informer sur l'état de validité de chaque registre ainsi qu'un bit de validité pour les flag CNZ et un pour le flag V.
  Il possède trois ports en lecture et deux ports en écriture.
  Les écritures se font de manière synchrone alors que la lecture est asynchrone.
  Si on veut écrire dans un registre il faut préalablement l'invalider afin d'informer sur l'impossibilité d'utiliser ce registre pour une autre instruction car ce dernier sera reécrit.
  Quand on lit un registre on récupère toujours sa valeur et son état de validité, c'est la machine à états de decod qui choisira de lancer ou non l'insutrction si le registre n'est pas valide. (Le même principe est appliqué aux flags).



Afin de valider notre banc de registre nous décidons de tester les cas critiques. En effet en cas d'écriture simultanée dans le même registre, le port d'écriture 0 (résultat venant d'exec) est prioritaire au port d'écriture 1 (résultat venant de mem), car il est forcement plus récent donc contient la vraie valeur du registre.



Test Bench du Band de registre

Dans ce test bench nous testons donc plusieurs choses :

- l'incrémentation de PC
- l'écriture simple dans deux registres différents
- l'écriture simultanée à la même adresse pour les deux ports d'écriture
- la lecture des registres ainsi que leurs validités

- Décodage des instructions
  On ne peut effectivement pas parler de l'étage decod sans expliquer comment les différentes instructions de notre processeur sont décodées.
  Nous utilisons ici plein de logique combinatoire afin d'affecter proprement nos signaux vers exec.
  Le décodage se passe en plusieurs étapes :

  - Dans un premier temps on décode les prédicats, nous effectuons des comparaisons entre les flags et le prédicat afin de determiner si l'instruction peut être executée ou non. (ainsi que si les prédicats sont valides en fonction de la validités des flags provenant du banc de registres).

  - Ensuite nous décodons le type de l'instruction reçue (traitement de données, branchement, transfert mémoire, transferts multiples, multiplication...

  - Puis en fonction de ces types d'instructions nous determinons comment controler exec et quelles opérandes nous devons lui envoyer : S'il s'agit :
    * d'un traitement de données c'est grâce à l'opcode de l'instruction que nous determinons le contrôle de l'ALU ainsi que les inversions préalable et la carry entrante (s'il s'agit par exemple d'une soustraction). Ensuite c'est le bit d'immédiat qui va determiner si on lit pour affecter un registre l'opérande 2 (qui pourra potentiellement être décalée par une valeur de registre ou par une valeur directement presente dans l'instruction) ou si l'opérande 2 est directement une valeur écrite dans l'instruction avec une rotation (x2) également directement écrite. (l'opérande 1 est un registre directement designé par son adresse dans l'instruction).

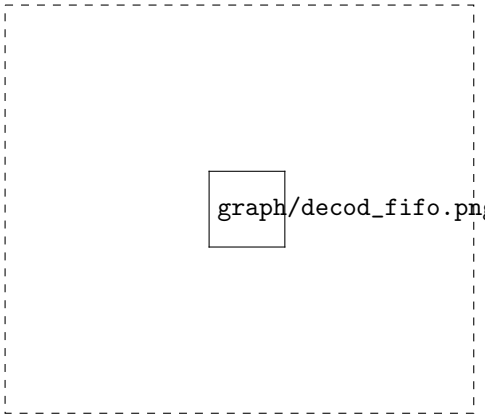
    * d'un branchement, un bit indique si c'est un appel de fonction ou non (on devra alors sauvgarder PC+4 dans le registre 14 (link register). Et le branchement s'effectue en additionnant l'offset directement présent (une extension sur 32 bits est necessaire) dans l'instruction à PC (registre 15).

    * d'un transfert mémoire, cela se passe quasiment comme un traitement de données, mais la valeur calculée est l'adresse et non un résultat d'opération. ?Nous avons plusieurs bits qui vont designer si on effectue un load ou un store, d'un octet ou d'un mot, si l'opérande 2 doit être additionnée ou soustraite à l'adresse de base (opérande 1 lu directement par un registre), si le calcul d'adresse doit être sauvgardé ou non dans le registre de base.

  En effet nous avons différentes façon de faire un transfert mémoire : Pré-index : (ex ldr r1, [r0, 4]) => r1 = mémorise le mot à l'adresse r0 + 4.
  Post-index : (ex ldr r1, [r0], 4) => r1 = mémorise le mot à l'adresse r0 et on enregistre la valeur r0 + 4 dans r0.
  Pré-index avec mise à jour : (ex ldr r1, [r0, 4]!) => r1 = mémorise le mot à l'adresse r0 + 4 et et on enregistre également la valeur r0 + 4 dans r0.
  Pré-index avec mise à jour : (ex Nous n'avons pas encore réalisé les instructions de multiplication et de transferts multiples.

- Machine à États
  La machine à états comporte en théorie cinq états (fetch, run, branch, link et mtrans). Nous n'avons à présent pas pu implémenté les instructions de type transfert multiples (l'état mtrans n'est donc pas encore présent dans la machine à états de notre processeur).
  Cette machine à états commande principalement les trois FIFO de l'étage (if to dec, dec to if et dec to exec).

  - if to dec : contient la valeur de l'instruction à être exectuée (instruction register).
  - dec to if : contient la valeur de PC (adresse de la prochaine instruction à être exectutée).
  - dec to exec : contient toutes les valeurs, opérandes et commandes des composants (ALU, shifter) de exec (ou également vers mem dans le cas d'accès mémoires), ainsi que les informations necessaires à decod à la reception des résultats (notamement la destination du resultat, write back ou non des registres, des flags etc.) Cette machine à états contrôle les signaux "pop" de if to dec, "push" de dec to if et "push" de dec to exec.



Ici le choix de faire une machine de Mealy a été choisie. Ce type de machine represente l'avantage par rapport à une machine de Moore de posséder moins d'états, donc d'être moins complexe. Car les états futurs dépendent directement des valeurs d'entrées (au lieu de dependre seulement de l'état actuel). Nous notons cependant que les machines de Mealy sont alors suceptibles d'être moins precises en terme de timing étant donneé le traitement des valeurs d'entrées.

Explication des états :

- FETCH sert à initialiser / demarrer le processeur, c'est à dire demander une première instruction au cache d'instructions.
  Dans cet état on push dec to if si seulement notre valeur de PC est valide et que dec 2 if est vide et on va à l'état RUN sinon on reste à FETCH.

- RUN est l'état "standard" du processeur, c'est ici qu'on va determiner si on execute ou non l'instruction en fonction des prédicats, de la validtité des registres et flags etc. C'est dans cet état qu'on fait les instruction de type traitement de données et transfert mémoire. Pour effectuer un branchement ou des transferts multiples on bascule vers les états correspondants.
  Dans cet état :
  * si l'instruction n'est pas executable (registres ou flags invalides, instruction précedente non executée), on push dans fetch que si la FIFO n'est pas pleine.

* si le prédicat est invalide on doit jetter l'instruction, donc on pop if to dec et on push si elle n'est pas pleine dans dec to if.

* si l'instruction est executable on push dans fetch (sauf si on a une intrcution de type branchement), dans exec et on pop de fetch.

– LINK est l'état où nous sauvgardons PC+4 dans le link register, puis on bascule directement dans l'état BRANCH.
Dans cet état on push dans exec (afin de sauvgarder PC) et et pop de fetch.

– BRANCH est l'état où on calcule l'adresse cible de PC, il s'agit de l'ajout d'un offset au PC actuel. Dans cet état, si la FIFO if to dec est vide on push dans fetch si seulement la FIFO n'est pas pleine et on reste à l'état BRANCH. Sinon si PC est valide on pop de fetch et on va à l'état RUN.

– MTRANS sera l'êtat où on va pouvoir reboucler en faisant les transfert mémoire et si on affecte le registre 15 (PC), on effectue un branchement.