Université de RENNES I
IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires

Plateforme REMIX

Manuel d'utilisation
Testbench de l'opérateur REMIX
Version 2.2

par
Hélène DAROLLES

le 12 mai 2006

Table des matières

1 Projet ReMIX

Le projet ReMIX a pour objectif de proposer une architecture matérielle destinée à accélérer la recherche d'information dans les masses de données. Son coeur est une mémoire de très grande capacité étroitement couplée à des ressources reconfigurables.

On peut alors accéder rapidement aux données (comparativement à des supports de stockage de type disque magnétique) et envisager des stratégies de recherche innovante orientées, notamment, sur l'indexation des données.

La programmation du système ReMIX est prise en charge par un *framework* qui guide le concepteur depuis la spécification d'une nouvelle application jusqu'à sa mise en oeuvre sur le support reconfigurable.

1.1 Plateforme ReMIX

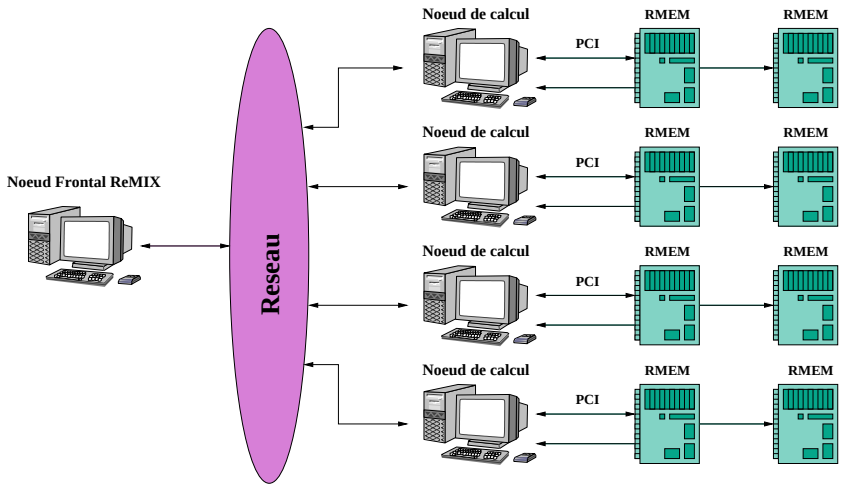

FIG. 1 – *Plateforme REMIX : architecture générale*

Le système ReMIX se présente sous la forme d'un cluster de 4 noeuds de calcul commandé par une machine hôte (noeud frontal).

Chaque noeud de calcul est un PC enrichi de 2 cartes *mémoires reconfigurables* : les cartes RMEM, que nous étudierons plus en détails par la suite. Ces cartes embraquent chacune 64 Go de mémoire flash et disposent de ressources reconfigurables conséquentes pour effectuer des traitements à la volée en sortie de la mémoire.

Le système complet dispose donc d'une mémoire de 512 Go associée à quelques 240 000 cellules logiques.

1.2 Carte RMEM

La carte RMEM est le coeur du système ReMIX. Sa conception a eu pour objectif d'associer sur un même support une mémoire de grande taille et des ressources reconfigurables utilisées pour réaliser un traitement à la volée des données au fur et à

mesure de leur lecture.

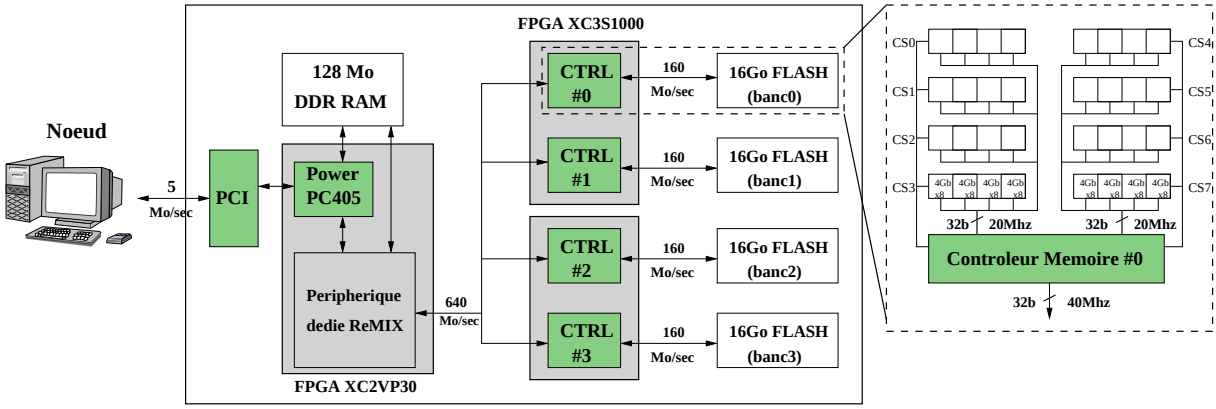

FIG. 2 – *Architecture d'une carte RMEM*

La figure ci-dessus décrit son architecture au niveau système. La carte RMEM se base sur une carte de développement pour FPGA, à laquelle nous avons associé une carte d'extension mémoire Flash-NAND.

La carte de développement est basée sur une interface PCI et dispose d'un FPGA Virtex-II Pro, associé à 128 Mo de SDRAM. Le Virtex-II Pro embarque l'équivalent de 30 000 cellules logiques, 136 blocs mémoire de 18 Kbits et de 2 coeurs de processeurs PowerPC405.

La carte mémoire est composée de 64 boîtiers mémoire de 1 Go, chaque boîtier intègre 2 composants Flash qui peuvent être utilisés de manière concurrente. Ceux-ci sont regroupés en 4 bancs indépendants de 16 Go, chaque banc étant organisé en une matrice de 8 x 4 composants dans laquelle on peut accéder à 8 composants en parallèle.

Chaque carte mémoire contient également deux circuits FPGA utilisés pour implémenter les contrôleurs des 128 composants Flash, et pour gérer l'interface entre chacun des 4 bancs mémoire et le circuit Virtex-II Pro de la carte PCI.

Le Virtex-II Pro intègre une architecture de type système sur puce construite autour du processeur PowerPC et d'un bus CoreConnect. Le système intègre plusieurs périphériques (contrôleur mémoire SDRAM, DMA, etc.) connectés au bus sytème, parmi lesquels un contrôleur dédié aux échanges de données (lecture et écriture) entre la carte mémoire Flash et la mémoire SDRAM de la carte PCI.

C'est ce périphérique d'interface que nous allons étudier, et que nous désignerons par la suite périphérique ReMIX.

1.3 Périphérique ReMIX

Le rôle de ce périphérique, dont l'architecture est représentée ci-dessous est double :

- Il gère les opérations de lecture et d'écriture des données de/sur la carte mémoire Flash
- Il intègre un bloc *opérateur* matériel spécialisé pour chaque application, qui permet de traiter à la volée les données lues à partir de la mémoire Flash

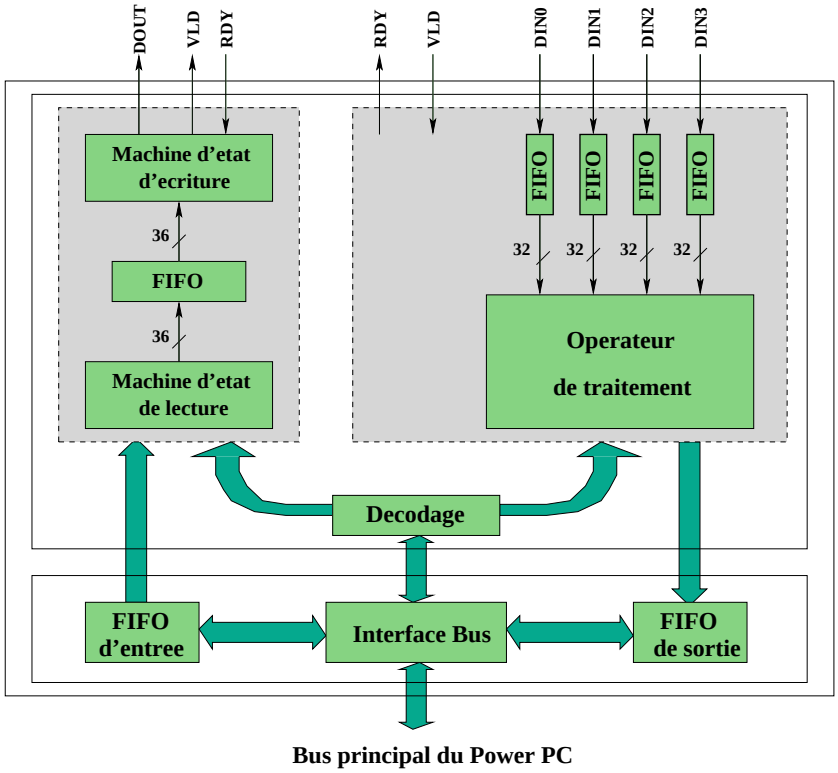

FIG. 3 – *Schéma de principe du périphérique dédié ReMIX*

L'opérateur reconfigurable peut recevoir simultanément 4 flux de données (4 x 32 bits) en provenance des 4 bancs de la mémoire Flash et produit en résultat un flux de données sur 64 bits. La synchronisation, en entrée comme en sortie, est assurée par des mémoires de type FIFO. 15 000 cellules logiques et 90 blocs mémoire RAM de 18 Kbits sont disponibles pour implémenter l'opérateur ReMIX, dont le fonctionnement est directement contrôlé par le PowerPC au travers du bus système.

Grâce à l'accès en parallèle de plusieurs composants Flash, la bande passante mémoire en entrée de l'opérateur est de 640 Mo/sec, soit deux ordres de grandeur de plus que celle disponible sur l'interface PCI de la carte ReMIX. Le bus PCI est donc un goulot d'étranglement, qui limite l'intérêt d'une utilisation directe de cette mémoire par l'hôte, puisque l'on dispose alors d'une bande passante trés réduite (inférieure par un ordre de grandeur à celle d'un disque dur), ce qui réduit sensiblement les gains liés au

temps d'accès aléatoire faible de la Flash (toujours par rapport au disque).

En pratique, ce type d'échange est exclu : le principe de fonctionnement de ReMIX consiste à utiliser l'opérateur matériel pour filtrer les données directement à la source et ainsi ne transmettre que l'information utile. Dans ce contexte, et en supposant que le taux de filtrage de l'opérateur est suffisant, la bande passante délivrée par l'interface PCI est suffisante. On peut résumer le déroulement d'un traitement sur cette architecture, c'est à dire la recherche et le traitement d'un objet dans la base d'index, comme suit :

1. Un ou plusieurs ordres de lecture sont envoyés depuis l'hôte (via le processeur PowerPC) au contrôleur Flash par le périphérique d'interface ReMIX.
2. L'opérateur ReMIX est paramétré par l'hôte (toujours via le PowerPC) en vue du traitement des données qui seront lues à partir de la mémoire Flash.
3. Les données lues, disponibles dans les FIFOs à l'entrée de l'opérateur, sont alors filtrées à la volée par ce dernier, qui génère un flux de résultats.
4. Ce flux résultat est transféré vers la mémoire SDRAM par l'utilisation d'un DMA, pour être finalement envoyé à l'hôte par l'interface PCI.
5. Une fois le traitement terminé, l'opérateur signale la fin des calculs au PowerPC par un signal d'interruption.

2 Opérateur ReMIX

2.1 Interface de l'opérateur

Voici l'interface générique de l'opérateur ReMIX.

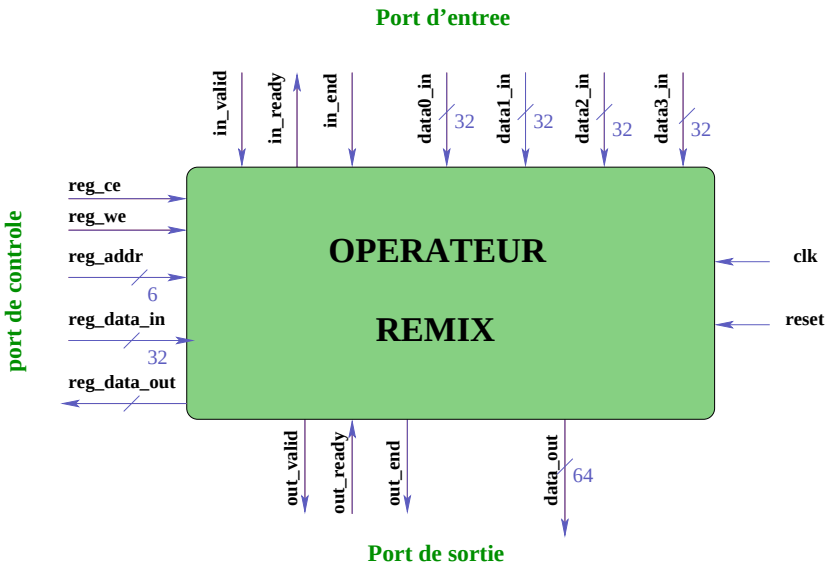

FIG. 4 – *Interface de l'opérateur ReMIX*

2.1.1 Alimentation du port d'entrée

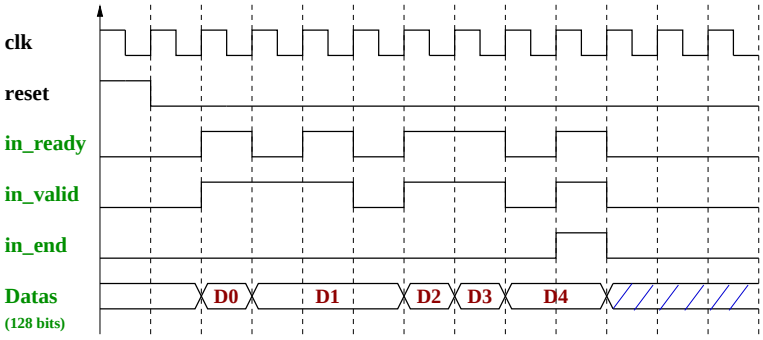

FIG. 5 – *Alimentation du port d'entrée de l'opérateur*

Signification des signaux :

- **in_valid** : activé lorsque toutes les données en entrées sont valides sur les ports in_data(0) à in_data(n-1) avec n=nombre de ports d'entrée sélectionnés. Reste à l'état haut tant que la donnée n'a pas été acquitée sur les ports d'entrées des données.
- **in_ready** : activé lorsque l'opérateur acquitte les données valides en entrées.
- **in_end** : activé lorsque la dernière donnée arrive sur le port d'entrée et le reste tant que in_valid est à l'état haut.

2.1.2 Récupération des résultats sur le port de sortie

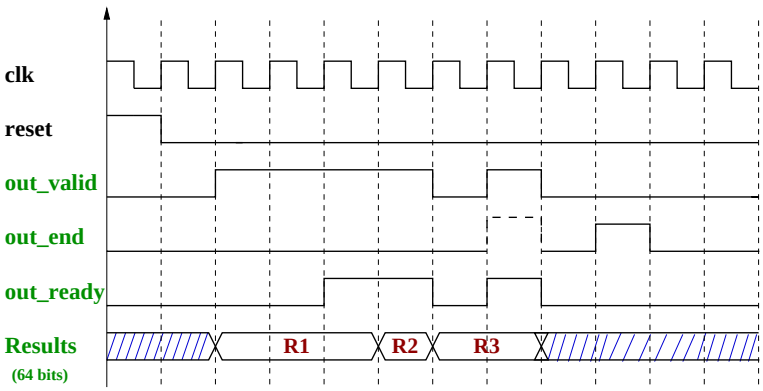

FIG. 6 – *Alimentation du port de sortie de l'opérateur*

Signification des signaux :

- **out_valid** : activé lorsque la donnée en sortie de l'opérateur est valide.
- **out_ready** : activé lorsque la donnée en sortie est acquittée.
- **out_end** : désigne la fin d'un cycle du filtre.
  Actif après que le dernier résultat sorte de l'opérateur.

2.1.3 Port de contrôle de l'opérateur

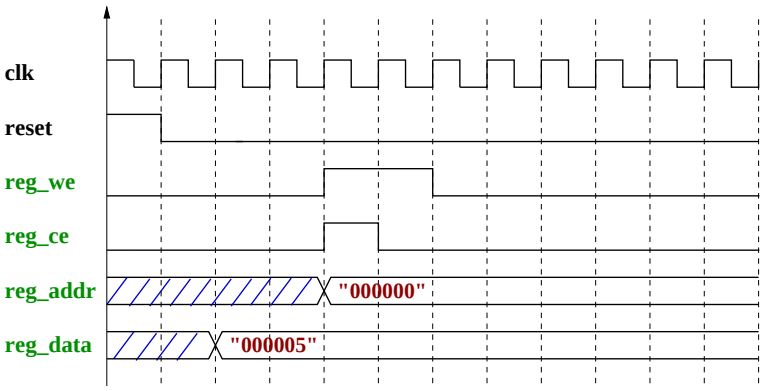

FIG. 7 – *Chronogramme représentant les registres*

Signification des signaux :
- **reg_addr** : adresse de la donnée.
- **reg_data_in** : 32 bits de données destinés au port du registre de l'opérateur.
- **reg_data_out** : 32 bits de données destinés à être écrit dans le registre.
- **reg_ce** : activé lorsqu'on veut permettre l'accès au registre (et seulement si reg_we, reg_addr et reg_data sont prêts).
- **reg_we** : à l'état haut : accès au registre en écriture.
  à l'état bas : accès au registre en lecture.

2.2 Fonctionnement général de l'opérateur

2.2.1 Cycle de fonctionnement élémentaire

Tout d'abord l'opérateur est paramétré en vue d'un traitement des données qui seront lues à partir de la mémoire flash. On intègre donc en entrée de l'opérateur une à quatre Fifos (selon le nombre de données de 32 bits à tester), qui vont gérer le flux de données en entrée de l'opérateur.

On démarre ensuite la lecture des fifos :
On lit les données disponibles dans les fifos N fois et on les envoie à l'opérateur.
Cette variable N est un paramètre récupéré dans un registre du module incluant l'opérateur (port de contrôle de l'opérateur).

Dernière étape :
Les données lues sont filtrées à la volée par l'opérateur, qui génère un flux de résultats.
Ce flux de résultats est géré en sortie de l'opérateur par une fifo : la fifo de sortie.

Le cycle de fonctionnement élémentaire prend fin lorsque l'opérateur active le signal out_end.

2.2.2 Alimentation du port d'entrée

On alimente les données en entrée de l'opérateur par l'intermédiaire de fifos, qui gèrent le flux de donnée en entrée. Comme on doit faire un certains nombre de lectures des fifos, on doit gérer un compteur qui permet de donner le nombre de données déjà lues dans les fifos. Puis on fait N lectures des données des fifos (N = read_size). Et tant qu'il y a des données disponibles dans toutes les fifos, on active le signal in_valid du port d'entrée de l'opérateur.

Lorsqu'on atteint la valeur (N-1), on lit la dernière donnée disponible des fifos, et en même temps on active le signal in_end du port d'entrée de l'opérateur. Le cycle d'entrée étant fini, on doit réinitialiser le compteur pour pouvoir démarrer un autre cycle.

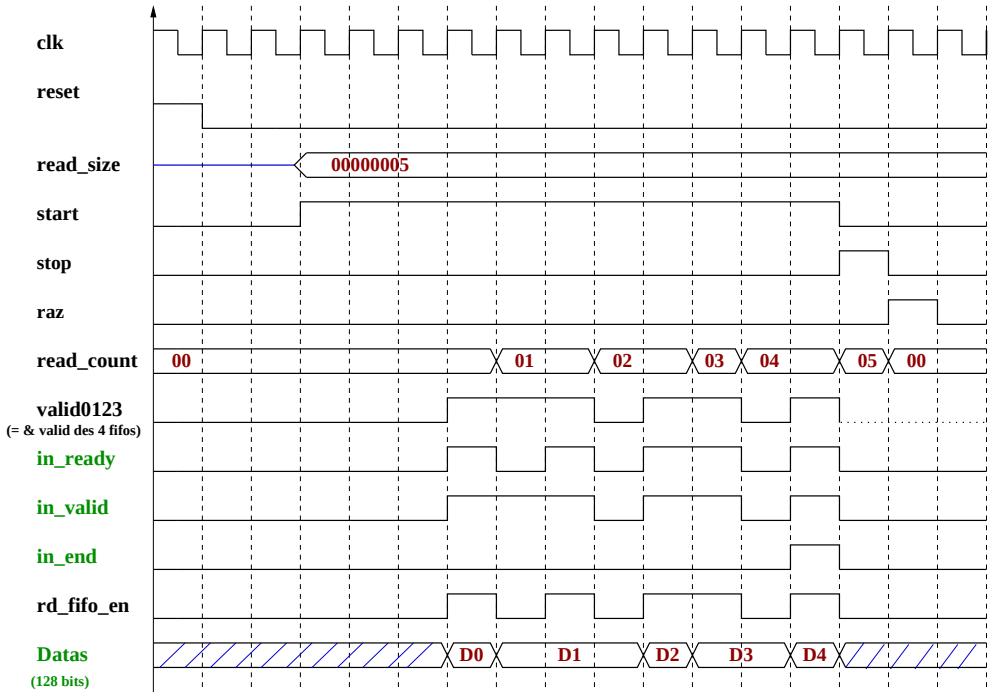

FIG. 8 – *Alimentation du port d'entrée de l'opérateur*

Signification des signaux :

- Start, Stop et Raz correspondent aux signaux du compteur:
  - **Start** : démarrage du compteur : activé lorsqu'on a écrit la donnée dans le registre.
  - **stop** : activé quand le nbre de données à envoyer à l'opérateur est atteint.
  - **raz** : remise à zéro du compteur
- **read_size** : nombre maximal de données à envoyer à l'opérateur. Valeur programmée par le processeur à chaque cycle de fonctionnement et stockée dans un registre.
- **read_count** : état du compteur : nombre de données déjà envoyées à l'opérateur. Initialisé à zéro (soit à l'activation du reset soit à l'activation du signal raz).

– **valid0123** <= and (valid0,valid1,valid2,valid3) (signaux valid des 4 fifos)
valid : il y a au moins une donnée valide dans la fifo.

2.2.3 Récupération des résultats en sortie

Le flux de résultats généré par l'opérateur est transmis à la fifo de sortie. Ces résultats sont écrit dans la fifo si les données en sortie de l'opérateur sont bien valides et que la fifo n'est pas pleine.

Une fois que l'opérateur a fini de traiter les dernières données, il n'y a plus de résultats : on peut arrêter le compteur.

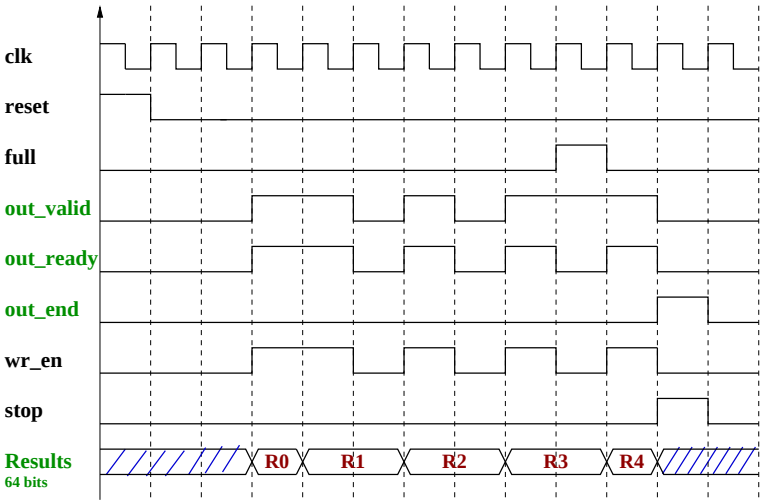

FIG. 9 – *Récupération des résultats en sortie de l'opérateur*

Signification des signaux :
– **full** de la fifo de sortie : activé lorsque la fifo est pleine.
– **wr_en** : commande l'écriture de la donnée dans la fifo de sortie.
Activé lorsque la fifo n'est pas pleine et que les données en sortie sont valides.
– **stop** : passe à l'état haut lorsque l'opérateur a terminé son traitement et que le signal out_end est activé

2.2.4 Contrôle de l'opérateur

Ci-dessous, la cartographie de la mémoire, divisée en deux parties :
– l'espace filterperiph : données utilisées pour effectuer les opérations de paramètrage de l'opérateur
(ex : nombre maximum de données maximum à envoyer à l'opérateur,.etc)
– l'espace opérateur: données utilisées par l'opérateur selon sa fonctionnalité.

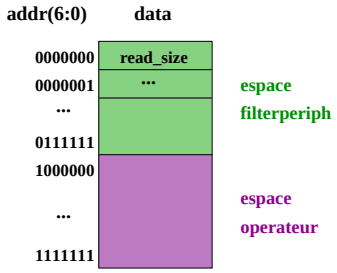

FIG. 10 – *Cartographie de la mémoire*

Ils partagent un espace d'adresse sur 7 bits : addr(6:0), dont le premier bit sert à définir auquel de ces deux espaces d'adressage est destinée la donnée.

- addr(6) = '0' =>destinée au filterperiph (interne)
- addr(6) = '1' =>destinée à l'opérateur (externe).

3 Utilisation du Testbench

Le principe général du testbench est d'automatiser les tests que l'on souhaite appliquer à un opérateur ReMIX. Nous envoyons à l'opérateur un certain nombre de données simultanément. Celui-ci les traite puis renvoie le résultat en sortie que nous vérifions.

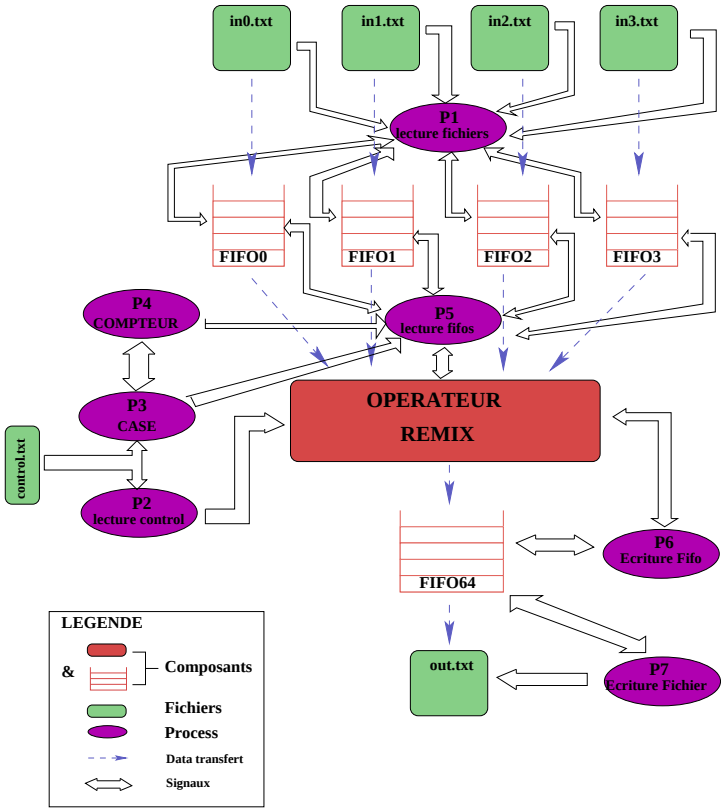

FIG. 11 – *Représentation globale du testbench*

Pour envoyer les données que l'on souhaite voir traiter par l'opérateur, il faut les écrire dans un fichier puis les récupérer dans les fifos en entrée. On associe donc à chaque flux d'entrée (32 bits) de l'opérateur, un fichier (in.txt).
Donc on créé autant de fichier qu'il existe de fifos en entrée.
Le paramétrage de l'opérateur se fait à l'aide du fichier de contrôle (control.txt).

Pour vérifier le bon fonctionnement de l'opérateur, il faut pouvoir récupérer et lire les résultats dans un fichier. Pour cette raison, on associe un fichier à la fifo de sortie (out.txt), qui contiendra les résultats de l'opérateur.

3.1 Répertoire de travail

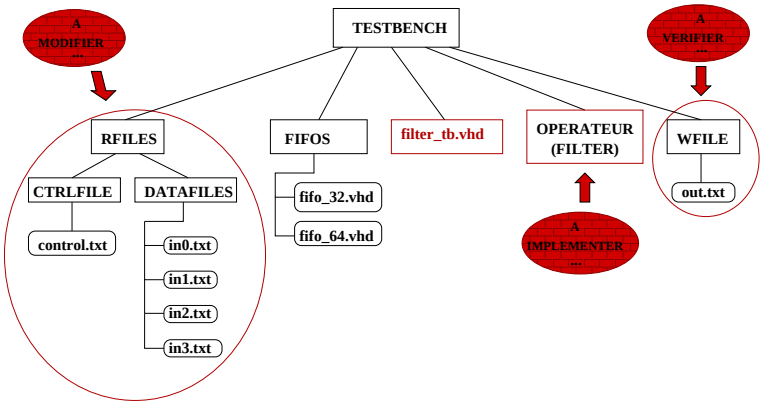

FIG. 12 – *Répertoire de travail*

L'organisation du répertoire de travail est la suivante :
- **RFILES** contient 2 répertoires :
  - **DATAFILE** : contient les 4 fichiers de données en entrées.
  - **CTRLFILE** : contient le fichier de récupération des données des registres.
- **WFILE** : contient le fichier de résultat (out.txt).
- **FIFOS** : contient les descriptions des 2 fifos dont nous avons besoin (une fifo de 32 bits pour les entrées et une fifo de 64 bits pour les sorties).
- **OPERATEUR** : contient tous les fichiers utilisés pour l'implémentation de l'opérateur ReMIX.
- **filter_tb.vhd** : fichier du testbench de l'opérateur.

3.2 Format des fichiers

Les 4 fichiers de données (in0.txt, in1.txt, in2.txt,in3.txt) contiennent des lignes de données sur 32 bits.

Le fichier de contrôle (control.txt), récupérant les données dans les registres, contient alternativement des lignes de 7 bits, correspondant à l'adresse de la donnée, puis des

lignes de 32 bits, correspondant à la donnée.

Pour chaque cycle, on doit donner un série de paramètres pour l'opérateur (adresses externes), ainsi qu'une adresse interne contenant le nombre de données à envoyer au filtre :

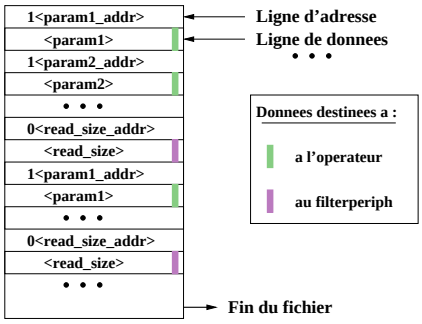

FIG. 13 – *Format du fichier de control de l'opérateur*

Le fichier en sortie (out.txt) contient le résultat : c'est à dire une ligne de données sur 64 bits.

3.3 Environnement de Simulation

3.3.1 Installation des outils et des fichiers d'environnement

Les outils sont installés sur le cluster ReMIX dans le répertoire **/local/share**, accessibles sur tous les noeuds ReMIX (Remix1 à Remix4) – et bientôt sur Remix0 : le noeud frontal– en exécutant la commande :

source envfile

avec envfile égal à :
- /local/share/modeltech/envmodelsim pour modelsim :
  pour la SIMULATION
- /local/share/xilinx/settings.sh ou csh pour les outils xilinx :
  pour l'IMPLEMENTATION

Pour afficher les fenêtres graphiques sur notre PC il faut :
- autoriser remix1 à afficher sur notre machine avec :

  xhost + remix1 tapé sur notre machine

- déporter l'affichage de remix1 vers notre machine (commande tapé sur remix1):

  export DISPLAY=<nom_de_machine>:0

  ou

  setenv DISPLAY <nom_de_machine>:0

3.3.2 Présentation de l'outil de simulation

* **Introduction et lancement des outils**

L'outil **Modelsim SE** de **Mentor Graphics** permet la simulation temporelle au niveau RT (transfert de registres) ou au niveau porte à partir du langage VHDL ou Verilog.

L'appel de cet outil de simulation se fait à l'aide de la commande **vsim**.
Bien que ce ne soit pas obligatoire, il est préférable de travailler non pas avec des fichiers mais avec un projet qui contiendra les fichiers mais aussi les configurations liées à ce projet.
Pour créer un projet dans ModelSim:

File->New->Project

Choisir un répertoire de travail sur votre disque :

File ->Change Directory

Choisir un nom de projet et laisser le nom de bibliothèque work.

* **Création et analyse des fichiers**

Après avoir créé tous les fichiers (description VHDL) nécessaires à notre projet, nous allons tester notre circuit. Avant de tester le circuit, il faut l'analyser. Pour cela il nous faut ajouter au projet les fichiers VHDL à compiler :

- le(s) fichier(s) de code
- le fichier de simulation
- et le fichier de configuration

File ->Add to Project ->Existing File

Les fichiers de codes sont les fichiers relatifs à l'opérateur, le fichier du testbench (filter_tb.vhd), la description VHDL des 2 fifos utiles (fifo_32.vhd et fifo_64.vhd).
Le fichier filter_tb.vhd fait office de simulateur et de fichier de configuration.
On compile ces fichiers avec :

Compile ->Compile All

En cas d'erreurs, il est préférable pour débugger de compiler les fichiers un à un :

Compile ->Compile Selected

Voici un aperçu de l'outil Modelsim après avoir mis en place l'environnement de simulation :

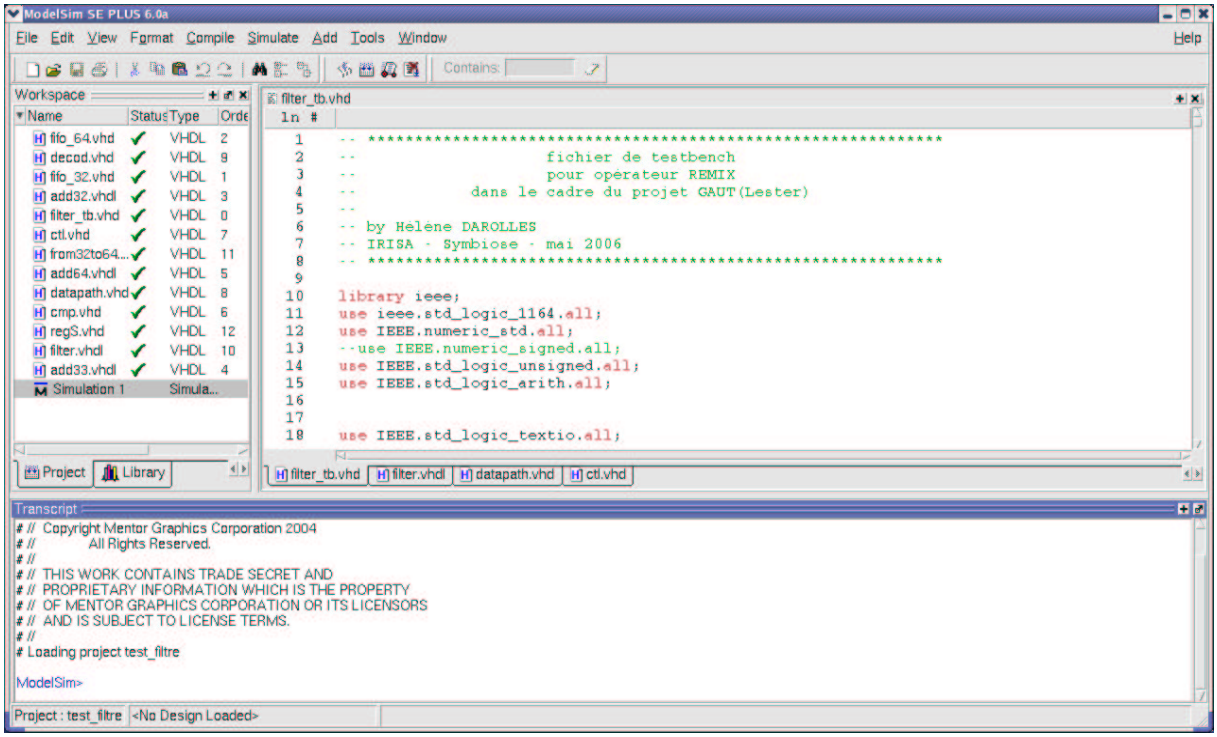

FIG. 14 – *Environnement de Simulation : ModelSim SE de ModelTech*

Pour plus de détails sur l'utilisation de Modelsim, reportez-vous à la rubrique d'aide du logiciel, ou bien aux documentations disponibles sur Internet.

3.4 Simulation

3.4.1 Lancement de la simulation

* **<u>Simulation "à la main"</u>**
Choisir l'onglet Library, puis, en double-cliquant sur la configuration <nom_du_fichierTB>.
Ceci est équivalent à :

Simulate ->Start simulation :

Onglet Design : *Design Unit = work.<nom_du_fichierTB>,*

Pour faire apparaître l'ensemble des fenêtre de simulation :

View ->Debug Window ->All,

Avant de continuer la simulation, une petite explication de cet outil s'impose. L'écran est composé de plusieurs parties :

- La fenêtre WORK SPACE (à gauche) avec ses onglets, permet d'accéder aux fichiers, aux entités compilées dans les bibliothèques,etc.
- Le code VHDL

- Les fenêtres "Active Prosess, "Locals", "Objects" et "Dataflow", permettent de sélectionner les signaux à visualiser.
- La fenêtre "Wave" dessine les chronogrammes.

Nous allons maintenant continuer la simulation.

On sélectionne dans la fenêtre **Objects**, l'ensemble des signaux que l'on veut visualiser, et on les copie dans la fenêtre "Wave":

Add to Wave ->Selected Signals.

Puis faire avancer la simulation de 100 ns :

Simulate ->Run ->Run 100 ns

Ou on utilise les icônes. Si on veut simuler plus longtemps : on augmente le pas (Run Length) et on clique sur l'icône run.

Certains signaux sont des données sur 32 ou même 64 bits. Il est possible, pour une meilleure lisibilité des chronogrammes, d'afficher les signaux en héxadécimal en procédant de la manière suivante après avoir sélectionné le ou les signaux:

Format ->Radix ->Héxadécimal

Pour ajouter des séparateur entre les différents signaux (par soucis de clarté) :

Add ->Divider,

ou bien clic droit sur un signal de la fenêtre "Wave" : *Insert Divider*
Puis rentrer un nom de diviseur.

Pour enregistrer la simulation (paramètrage et configuration) :

- Cliquer sur le fenêtre des signaux Wave
- Sauvegarder sous le nom de <nom_du_fichierTB>.do

En enregistrant notre simulation, cela nous évite de repasser par toutes les étapes précédentes : c'est un gain de temps.

*** Simulation "automatique"**

Dans la fenêtre de WorkSpace :

Add to Project ->Simulation Configuration,

Onglet Design : *Design Unit = work.<nom_du_fichierTB>,*

Onglet Others : *Others Options = -do <nom_du_fichierTB>.do*

Cette dernière commande permet de récupérer les signaux à visualiser.
Ainsi la simulation automatique apparaît dans le Work Space. On peut alors lancer la

simulation en double-cliquant dessus.

*** "Trucs et Astuces"**
Pour modifier/raccourcir le nom des signaux de la fenêtre Wave :

Tools ->Options ->Wave Preference : 1# element

Pour faire un zoom "adapté" :
Pour que le chronogramme montre uniquement l'intervalle de temps la plus intéressante de la simulation. Cette intervalle de temps peut être défini avec la commande :

View ->Wave ->Zoom ->Zoom range,

3.4.2 Récupération des résultats

Pour voir les résultats en sortie de l'opérateur, on doit visualiser le signal out_data qui correspond au signal de sortie de l'opérateur. Ce qui permet de vérifier si l'opérateur a correctement traité les données qu'on lui a envoyées. Enfin il faut visualiser le signal data_fifo_64_out correspondant au signal de sortie de la fifo de sortie. Ces dernières données visualisées sont également sauvegardées dans le fichier **out.txt**.

4 Fonctionnement du Testbench

Les entrées de l'opérateur sont connectées à des fifos alimentées par des fichiers.
La sortie de l'opérateur est connectée à une fifo dont le contenu est transféré vers le fichier **out.txt**.
Le fichier control.txt permet le paramétrage de l'opérateur et le démarrage d'un traitement.

4.1 Récupération des données des fichiers

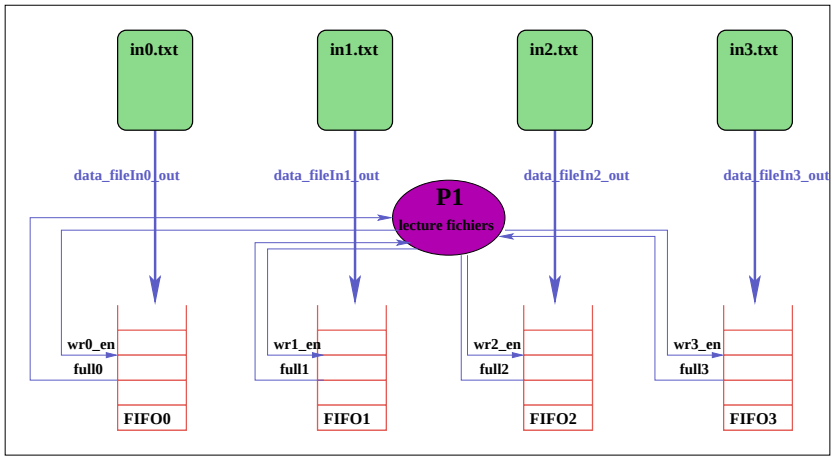

FIG. 15 – *Lecture des fichiers en entrées*

On ouvre en mode lecture les fichiers d'entrées (in0.txt, in1.txt, in2.txt, in3.txt).

Après avoir initialisé les signaux **data_fileIn_out** et **wr_en** à "0" pour chacunes des 4 fifos_32, tant que ce n'est pas la fin du fichier in0.txt et que la fifo0 associée n'est pas pleine (**full0**="0"), sur front montant on lit la ligne du fichier que l'on connecte sur l'entrée **data_fileIn0_out** de la fifo0, et on met le signal **wr0_en** de la fifo à '1'. On effectue la même opération pour chacune des 3 autres fifos.

A la fin de la boucle, on ferme les 4 fichiers, et on remet accessoirement les valeurs sur les entrées modifiées de la fifo à "0".

Remarques :
On ne peut pas lire une ligne dans un fichier et l'associer directement à un signal.
Pour lire une ligne dans un fichier, on doit passer par 2 étapes :

1. Lire la ligne dans le fichier et l'associer à une variable **Lin** de type "ligne" : procedure**READLINE**(file f : TEXT; L : inout LINE)
2. Lire la variable Lin et l'associer au signal **donnee0** de type std_logic_vector : procedure**READ**(L : inout LINE; VALUE : in STD_LOGIC_VECTOR)

4.2 Lecture des Registres

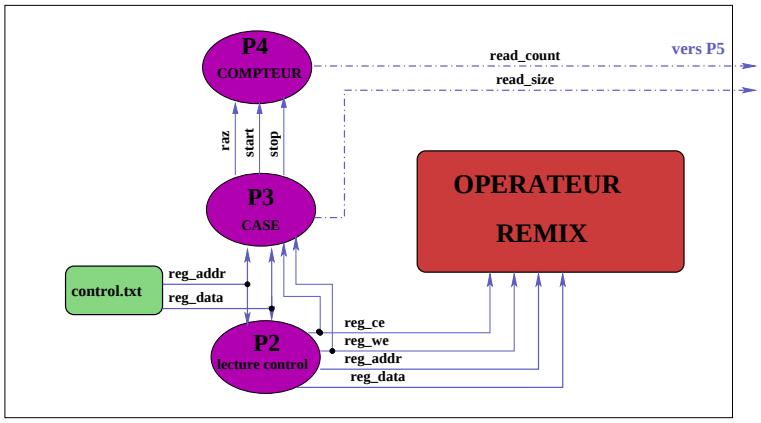

FIG. 16 – *Récupération des données des registres*

4.2.1 Lecture du fichier

On ouvre le fichier control.txt, contenant les adresses et les données correspondantes, en mode lecture.
Après avoir initialisé les valeurs des entrées (**reg_ce**, **reg_we**, **reg_addr** et **reg_data**) de l'opérateur, on lit une ligne du fichier **control.txt**.
On récupère les bits 37 à 32 de la ligne que l'on associe au signal **reg_addr** et les 32 dernier bits au signal **reg_data**.
le signal **reg_ce** correspond au 1er bit de la ligne. On met le signal **reg_we** à '1' (en écriture). Ces 4 signaux sont envoyés à l'opérateur.

On effectue une lecture du fichier control.txt tant que le bit de poids fort (reg_ce) est à '1' (data destinée à l'opérateur). Lorsqu'il passe à '0', cela correspond à une nouvelle valeur pour le signal read_size. L'écriture dans le registre read_size provoque le démarrage du cycle. On doit donc attendre la fin du traitement (out_end = '1'), et lancer un nouveau traitement.

On répète cette procédure tant que le fichier n'est pas vide. Et lorsque le fichier est vide, on le ferme.

4.2.2 Décodage de l'adresse

Dans ce processus, on initialise le compteur à l'aide du signal raz et on attribue la valeur au signal **read_size**. Le signal **reg_ce** précise si les données sont destinées au testbench (signal **read_size**) ou à l'opérateur (signaux quelconques utilisés par l'opérateur).
Par exemple pour l'adresse "000000" avec ce="0", on récupère la donnée pour le testbench dans **read_size**, qui correspond au nombre maximum de données de 128 bits (4 fois 32 bits) à envoyer à l'opérateur.

4.2.3 Compteur

Pour pouvoir savoir combien de données de 128 bits ont déjà été envoyées à l'opérateur, on a besoin d'implémenter un compteur. Ce compteur est remis à zéro si **reset**='1' ou si on reçoit le signal **raz**.
On incrémente la valeur du compteur (signal **read_count**) lorsqu'on lit les 4 données des 4 fifos en même temps. C'est à dire lorsque les signaux **rd0_en**, **rd1_en**, **rd2_en**, **rd3_en** sont à "1". Par soucis d'économie, on a relié ces 4 signaux au signal **rdfifo_en**.

4.3 Envoie des données à l'opérateur

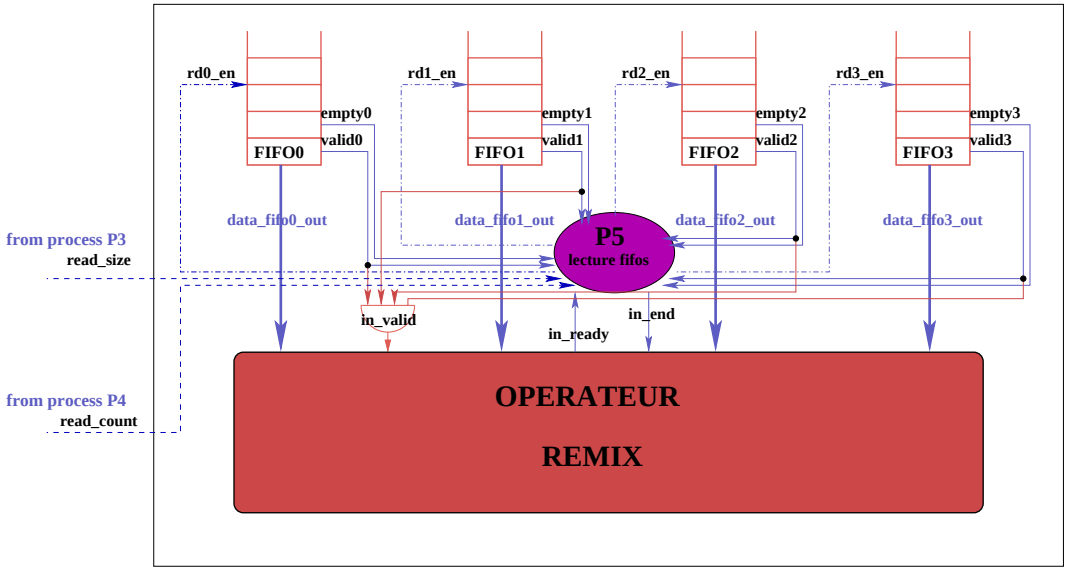

FIG. 17 – *Lecture des fifos 32 bits*

Le processus P5 permet de lire les données des 4 fifos (**data_fifo_out**) et de les envoyer à l'opérateur (**in_data**).
Il contrôle ainsi les entrées **in_valid** et **in_end** de l'opérateur.
Les données en entrées sont valides lorsque les signaux **valid0**, **valid1**, **valid2**, **valid3** sont à "1" et que le compteur a démarré (**start**="1").

Tant qu'on n'a pas atteint la valeur maximale du nombre de données à envoyer à l'opérateur (**read_size!=read_count**), que le compteur a démarré (**start="1"**) et que l'opérateur acquitte bien les données reçues (**in_ready="1"**), alors on lit les données des 4 fifos : **rdfifo_en = "1"**.
Lorsqu'on atteint la dernière donnée à lire (**read_count=read_size-1**) alors le signal **in_end** passe à "1".

Remarque :
Le signal de sortie **valid** des fifos prend déjà en compte le signal de sortie **empty**. C'est à dire qu'une donnée de la fifo est valide si celle-ci n'est pas vide.

4.4 Récupération des résultats

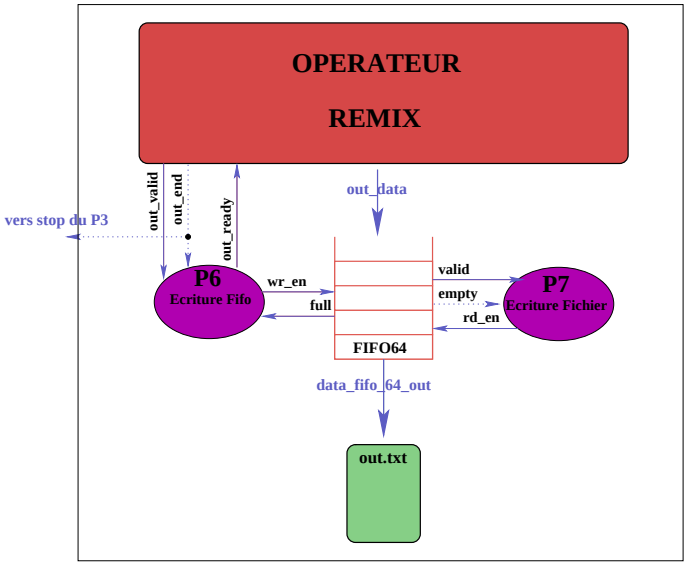

FIG. 18 – *Ecriture des résultats de l'opérateur dans le fichier de sortie*

4.4.1 Ecriture dans la fifo_64

Tout d'abord on acquitte (**out_ready="1"**) les données en sortie de l'opérateur (**out_data**) et on écrit les données dans la fifo (**wr_en="1"**), si celles-ci sont valides (**out_valid="1"**), et qu'on peut les écrire dans la fifo (elle n'est pas pleine : **full="0"**).

Le signal de sortie **in_end** de l'opérateur (si c'est la dernière donnée) permet d'arrêter le compteur en activant le signal **stop**.

4.4.2 Ecriture dans le fichier résultat

On ouvre ensuite le fichier de sortie **out.txt** en mode écriture.
Si la donnée en sortie de la fifo (**data_fifo_64_out**) est valide (**valid="1"**), alors on l'écrit dans le fichier out.txt, à l'aide des procédures :

- **WRITE**(L : inout LINE; VALUE : in STD_LOGIC_VECTOR)
- **WRITELINE**(file f : TEXT; L : inout LINE)

On ferme le fichier de résultats.

5 Exemple d'utilisation

5.1 Spécification de l'opérateur

Pour pouvoir valider le testbench, nous avons implémenté comme opérateur ReMIX un filtre simple. Celui-ci fait la somme de toutes les données qu'on recoit en entrées et sort cette somme quand on dépasse un certain seuil.

5.2 Description VHDL de l'opérateur

5.2.1 description VHDL du filtre

Nous avons donc décomposé le filtre en deux partie distinctes : le chemin de données et la partie contrôle

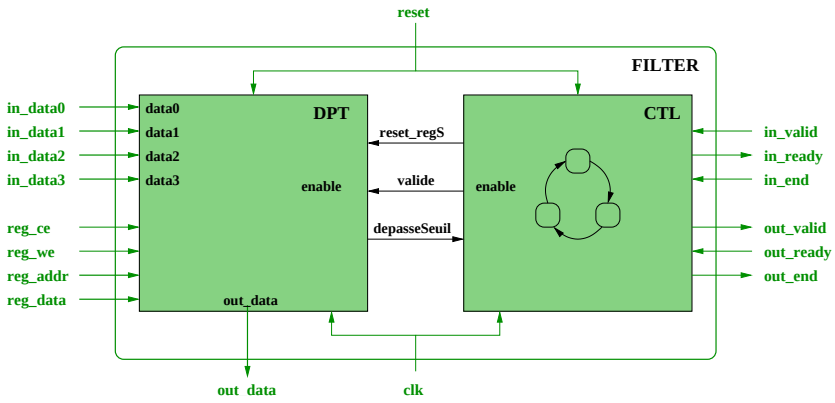

FIG. 19 – *Organisation du filtre*

Nous avons décrit le filtre dans le fichier **filter.vhd** que nous avons fourni en Annexes.

5.2.2 description VHDL de la partie chemin de données

Le chemin de données permet d'effectuer les calculs nécessaires au bon fonctionnement de l'opérateur. Il est représenté par le circuit ci-dessous :

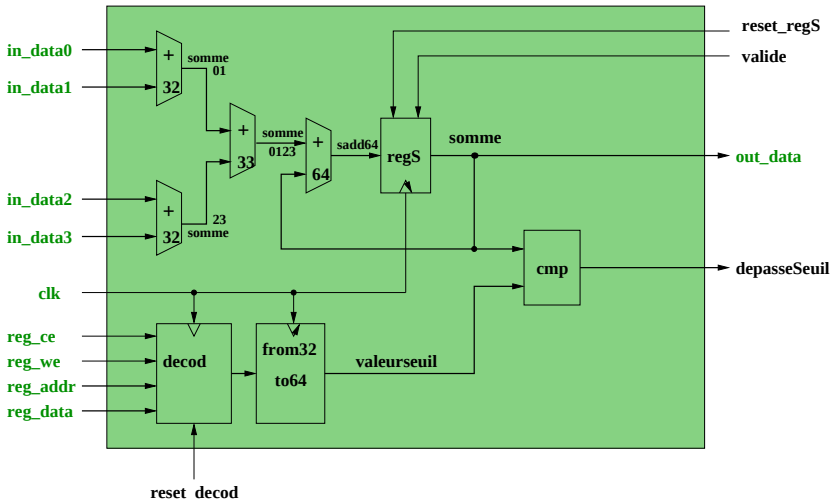

FIG. 20 – *Circuit de la partie datapath du filtre*

L'implémentation du chemin de données est décrite dans le fichier **datapath.vhd** fourni en Annexes.

5.2.3 description VHDL de la partie contrôle

La partie contrôle est un automate de mealy, et permet de contrôler la partie chemin de données :

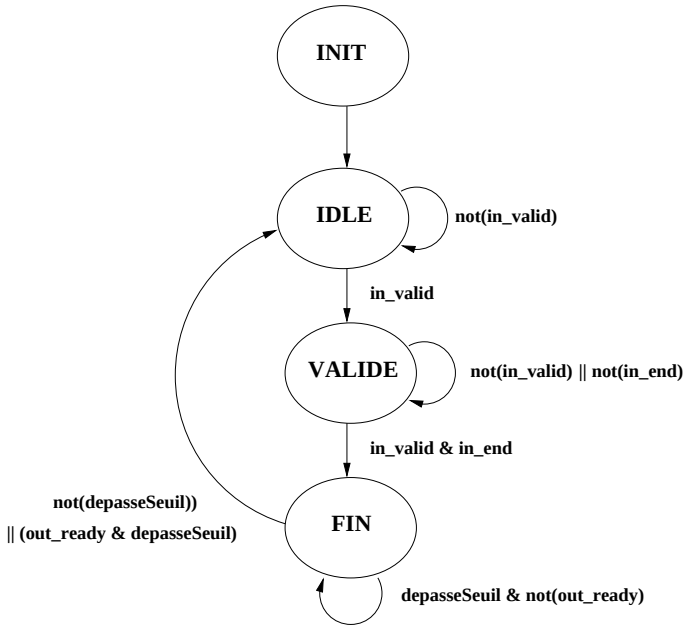

FIG. 21 – *Automate de la partie contrôle du filtre*

L'implémentation de cet automate est décrite dans le fichier **ctl.vhd** fourni en Annexes.

5.3 Exemples de fichiers en entrées

Exemple de fichier (**in0.txt**)contenant les données en entrée de l'opérateur :

```
00000000000000000001000000000000
00000000000000000010000000000000
00000000000000000011000000000000
00000000000000000100000000000000
00000000000000000101000000000000
00000000000000000110000000000000
00000000000000000111000000000000
00000000000000001000000000000000
00000000000000001001000000000000
00000000000000001010000000000000
00000000000000001011000000000000
00000000000000001100000000000000
00000000000000001101000000000000
00000000000000001110000000000000
00000000000000001111000000000000
```

Exemple de fichier (**control.txt**) contenant les données des registres :

```
10000000000000000000000111001100010000
00000000000000000000000000000000000100
```

5.4 Lancement de la simulation

On lance la simulation sur 300 ns.

Remarque :
Nous avons modifié le format des données (Héxadécimal ->Décimal) pour pouvoir bien voir les valeurs des données en entrée et en sortie.

Nous avons effectué un Zoom adapté, et ce pour chacunes des trois captures d'écran.

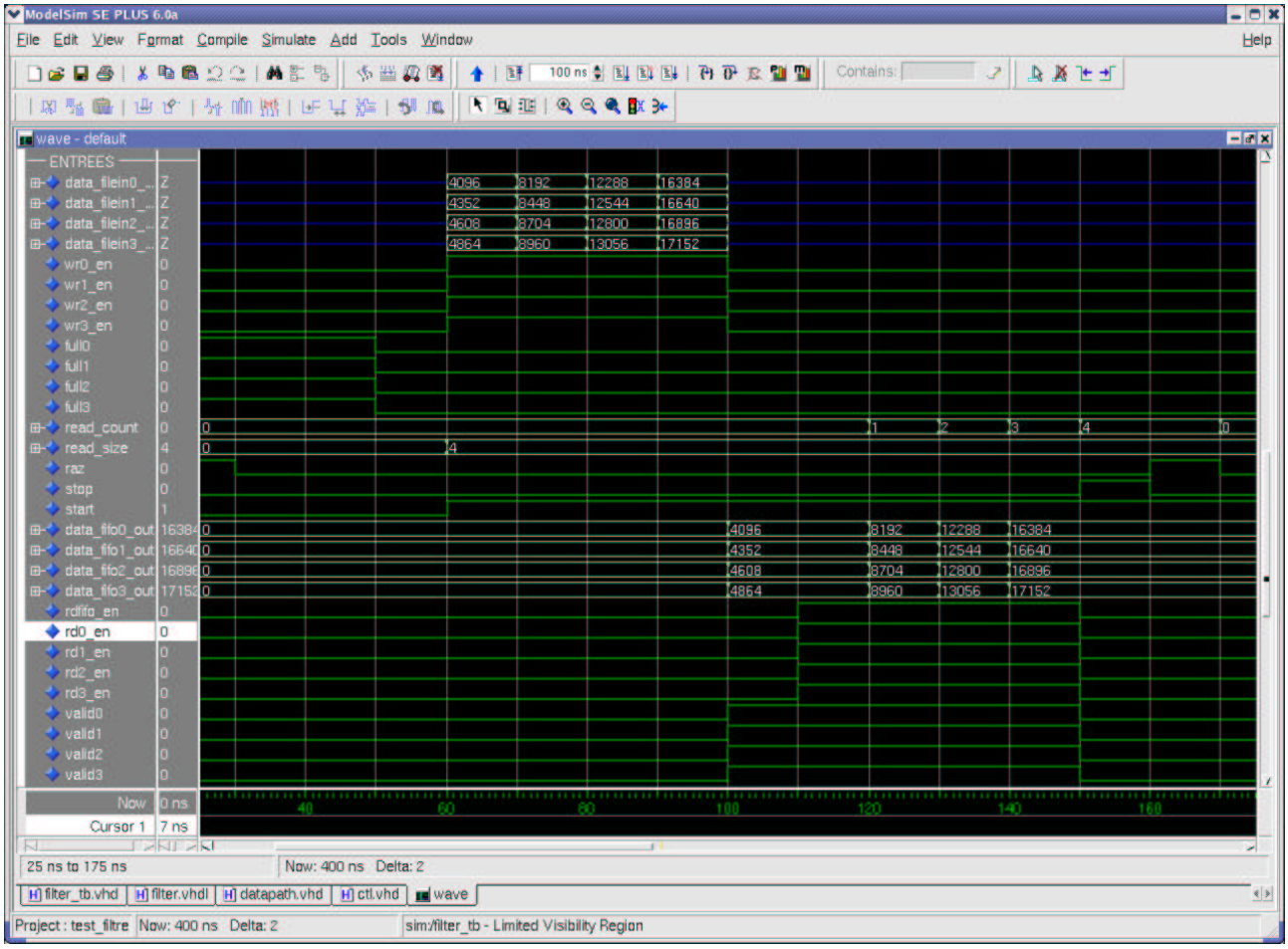

FIG. 22 – *Simulation montrant les signaux en entrée*

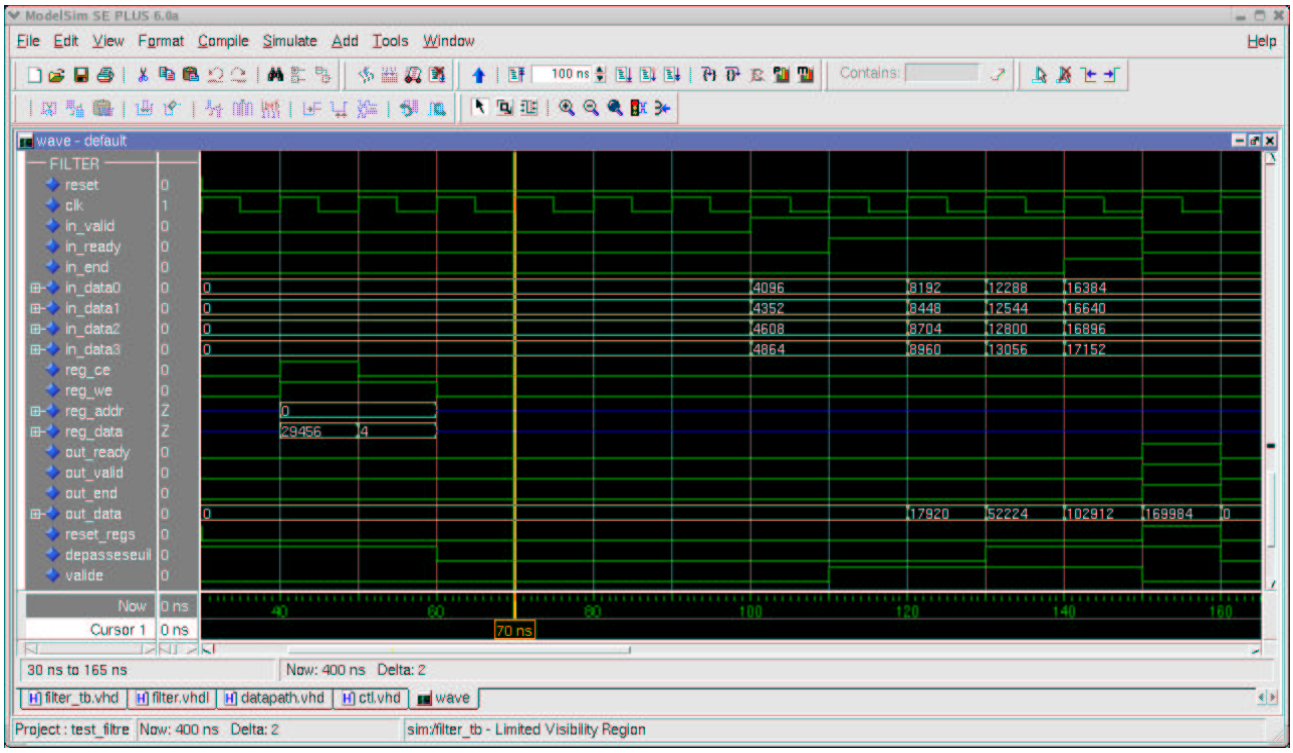

FIG. 23 – *Simulation montrant les signaux du filtre*

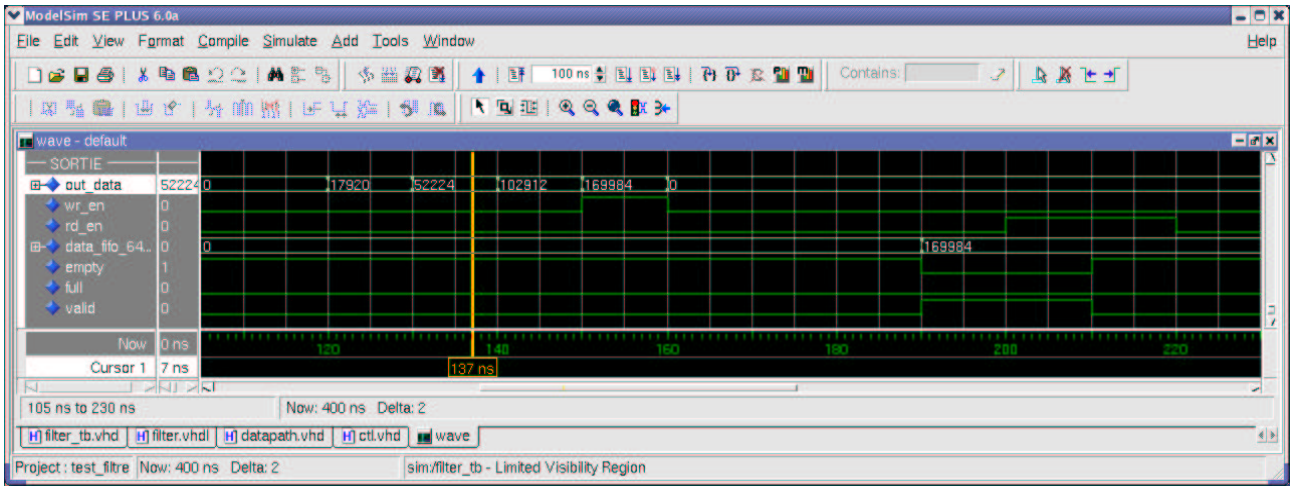

FIG. 24 – *Simulation montrant les signaux en sortie*

5.5 Fichiers Résultats

Le fichier **out.txt** contenant le résultat de la simulation est le suivant :

```
00000000000000000000000000000000000000000000000101001100000000000
```

Ce qui est équivalent en décimal au résultat suivant : **169 984**.

Le fichier control.txt incluant les données des registres, contient comme valeur de Seuil en décimal : **29 456**, et comme nombre maximum de données à envoyer au filtre :**4**.

Cette valeur seuil est effectivement dépassée lorsque la somme devient égale à : **52 224**.

Sur la simulation montrant les signaux de sortie on voit bien que le résultat correspond à la somme des 4 données de 128 bits en entrée, et c'est bien le résultat qui est écrit dans le fichier de sortie out.txt.

6 Annexes : Listing des fichiers

6.1 Testbench

6.1.1 "filter_tb.vhd"

```
-- ************************************************************
--                    fichier de testbench
--                    pour opérateur REMIX
--            dans le cadre du projet ReMIX/GAUT(Lester)
--              Responsable : Dominique LAVENIER
--              Encadrant : Gilles GEORGES
-- by Hélène DAROLLES
-- IRISA - Symbiose - mai 2006
-- ************************************************************

library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;


use IEEE.std_logic_textio.all;

library std;
use std.textio.all;

library work;
use work.all;

entity filter_tb is
end filter_tb;


architecture BEH of filter_tb is

-- ***********************************************************************
-- ******************** DECLARATION DES CONSTANTES **********************
-- ***********************************************************************

     constant FILE0_IN  :  string := "infile/in0.txt";       -- contient les data en entrées
     constant FILE1_IN  :  string := "infile/in1.txt";       -- contient les data en entrées
     constant FILE2_IN  :  string := "infile/in2.txt";       -- contient les data en entrées
     constant FILE3_IN  :  string := "infile/in3.txt";       -- contient les data en entrées
     constant FILE_CTL  :  string := "infile/control.txt";   -- contient les reg
     constant FILE_OUT  :  string := "outfile/out.txt";      -- contient le résultat

     constant PERIODE   : time := 10 ns;

-- ***********************************************************************
-- ******************* DECLARATION DES COMPOSANTS ***********************
```

```
-- ***********************************************************************

     component filter
        port(
             reset        : in std_logic;
             clk          : in std_logic;
             in_valid     : in std_logic;
             in_ready     : out std_logic;
             in_end       : in std_logic;
             in_data0     : in std_logic_vector(31 downto 0);
             in_data1     : in std_logic_vector(31 downto 0);
             in_data2     : in std_logic_vector(31 downto 0);
             in_data3     : in std_logic_vector(31 downto 0);
             reg_ce       : in std_logic;
             reg_we       : in std_logic;
             reg_addr     : in std_logic_vector(5 downto 0);
             reg_data     : in std_logic_vector(31 downto 0);
             out_valid    : out std_logic;
             out_ready    : in std_logic;
             out_end      : out std_logic;
             out_data     : out std_logic_vector(63 downto 0)
             );
     end component;

      -- %%%%%%%%%%%%%%%%%  LES FIFOS  %%%%%%%%%%%%%%%%%

     component fifo_32
        port(
             din    : in std_logic_vector(31 downto 0);
             rd_clk : in std_logic;
             rd_en  : in std_logic;
             rst    : in std_logic;
             wr_clk : in std_logic;
             wr_en  : in std_logic;
             dout   : out std_logic_vector(31 downto 0);
             empty  : out std_logic;
             full   : out std_logic;
             valid  : out std_logic
             );
     end component;

                  -- %%%%%%%%%%%%%%%%%%%%%%%%%%

     component fifo_64
        port(
             din    : in std_logic_vector(63 downto 0);
             rd_clk : in std_logic;
             rd_en  : in std_logic;
             rst    : in std_logic;
             wr_clk : in std_logic;
             wr_en  : in std_logic;
             dout   : out std_logic_vector(63 downto 0);
             empty  : out std_logic;
             full   : out std_logic;
             valid  : out std_logic
             );
     end component;

-- ***********************************************************************
-- **********************  DECLARATION DES SIGNAUX **********************
-- ***********************************************************************

     signal reset        : std_logic;
     signal clk          : std_logic;

                  -- %%%%% SIGNAUX FILTER %%%%%
```

```
signal in_valid     : std_logic;
signal in_ready     : std_logic;
signal in_end       : std_logic;
signal reg_ce       : std_logic;
signal reg_we       : std_logic;
signal reg_addr     : std_logic_vector(5 downto 0);
signal reg_data     : std_logic_vector(31 downto 0);
signal out_valid    : std_logic;
signal out_ready    : std_logic;
signal out_end      : std_logic;
signal out_data     : std_logic_vector(63 downto 0);

         -- %%%%% SIGNAUX FIFO_32 (0,1,2,3)%%%%

SIGNAL data_fileIn0_out : std_logic_vector(31 downto 0);
SIGNAL data_fileIn1_out : std_logic_vector(31 downto 0);
SIGNAL data_fileIn2_out : std_logic_vector(31 downto 0);
SIGNAL data_fileIn3_out : std_logic_vector(31 downto 0);

SIGNAL data_fifo0_out   : std_logic_vector(31 downto 0);
SIGNAL data_fifo1_out   : std_logic_vector(31 downto 0);
SIGNAL data_fifo2_out   : std_logic_vector(31 downto 0);
SIGNAL data_fifo3_out   : std_logic_vector(31 downto 0);

SIGNAL wr0_en   : std_logic;
SIGNAL wr1_en   : std_logic;
SIGNAL wr2_en   : std_logic;
SIGNAL wr3_en   : std_logic;
SIGNAL rd0_en   : std_logic;
SIGNAL rd1_en   : std_logic;
SIGNAL rd2_en   : std_logic;
SIGNAL rd3_en   : std_logic;
SIGNAL rdfifo_en : std_logic;

SIGNAL full0    : std_logic;
SIGNAL full1    : std_logic;
SIGNAL full2    : std_logic;
SIGNAL full3    : std_logic;
SIGNAL empty0   : std_logic;
SIGNAL empty1   : std_logic;
SIGNAL empty2   : std_logic;
SIGNAL empty3   : std_logic;
SIGNAL valid0  : std_logic;
SIGNAL valid1  : std_logic;
SIGNAL valid2  : std_logic;
SIGNAL valid3  : std_logic;

             -- %%%% SIGNAUX FIFO_64 %%%%%

SIGNAL data_fifo_64_out : std_logic_vector(63 downto 0);
SIGNAL empty  : std_logic;
SIGNAL full   : std_logic;
SIGNAL valid : std_logic;
SIGNAL rd_en  : std_logic;
SIGNAL wr_en  : std_logic;

             -- %%%%%%%  REGISTRE   %%%%%%%

-- nb de données maximum à envoyer au filtre :
SIGNAL read_size  : std_logic_vector(31 downto 0);
-- valeur à ne pas dépasser pour la fonction du filtre donnée :
SIGNAL read_seuil : std_logic_vector(31 downto 0);
-- compteur sur le nb de données déjà envoyées au filtre :
SIGNAL read_count : std_logic_vector(31 downto 0);
-- signaux de gestion du compteur :
SIGNAL start      : std_logic;
SIGNAL stop       : std_logic;
SIGNAL raz        : std_logic;
```

```
-- ************************  BEH BEGIN ****************************

begin

 -- ****************************************************************
 -- **************** PROCESS 0 : CLOCK ET RST *********************
 -- ****************************************************************

 GENERATE_CLOCK : process --génération de la clock
 begin
    clk <= '1';
    WAIT FOR PERIODE/2;
    clk <= '0';
    WAIT FOR PERIODE/2;
 end process GENERATE_CLOCK;

 GENERATE_RST : process
 begin
     reset <= '1';
     WAIT FOR (3*PERIODE);
     reset <= '0';
     WAIT;
 end process GENERATE_RST;

 -- ****************************************************************
 -- ************************  PROCESS 1 : **************************
 -- ******* lecture des donnees contenues dans les fichiers *******
 -- ****************************************************************

 DATAIN_FILES : process
    file stimuli0_file : text open read_mode is FILE0_IN;
    file stimuli1_file : text open read_mode is FILE1_IN;
    file stimuli2_file : text open read_mode is FILE2_IN;
    file stimuli3_file : text open read_mode is FILE3_IN;

    variable donnee0 : std_logic_vector(31 downto 0);
    variable donnee1 : std_logic_vector(31 downto 0);
    variable donnee2 : std_logic_vector(31 downto 0);
    variable donnee3 : std_logic_vector(31 downto 0);

    variable Lin : line;

 begin
       -- ************ INITIALISATIONS ************

    data_fileIn0_out  <= (others => 'Z');
    data_fileIn1_out  <= (others => 'Z');
    data_fileIn2_out  <= (others => 'Z');
    data_fileIn3_out  <= (others => 'Z');

    wr0_en <= '0';
    wr1_en <= '0';
    wr2_en <= '0';
    wr3_en <= '0';

               -- **********  BEGIN  ************

    wait until (reset = '0');

    while ((not endfile(stimuli0_file)) or (not endfile(stimuli1_file))
        or (not endfile(stimuli2_file)) or (not endfile(stimuli3_file)) ) loop
       wait until rising_edge(clk);

                 -------- Fifo0 --------

       if ((not endfile(stimuli0_file)) and full0='0') then
          readline(stimuli0_file, Lin);
```

```
                read(Lin,donnee0);
                data_fileIn0_out <= donnee0;
                wr0_en <= '1';
             else
                data_fileIn0_out <= (others => 'Z');
                wr0_en <= '0';
             end if;

                      -------- Fifo1 --------

             if ((not endfile(stimuli1_file)) and full1='0') then
                readline(stimuli1_file,Lin);
                read(Lin,donnee1);
                data_fileIn1_out <= donnee1;
                wr1_en <= '1';
             else
                data_fileIn1_out <= (others => 'Z');
                wr1_en <= '0';
             end if;

                      -------- Fifo2 --------

             if ((not endfile(stimuli2_file)) and full2='0') then
                readline(stimuli2_file, Lin);
                read(Lin,donnee2);
                data_fileIn2_out <= donnee2;
                wr2_en <= '1';
             else
                data_fileIn2_out <= (others => 'Z');
                wr2_en <= '0';
             end if;

                      -------- Fifo3 --------

             if ((not endfile(stimuli3_file)) and full3='0') then
                readline(stimuli3_file, Lin);
                read(Lin,donnee3);
                data_fileIn3_out <= donnee3;
                wr3_en <= '1';
             else
                data_fileIn3_out <= (others => 'Z');
                wr3_en <= '0';
             end if;
          end loop;

          wait until rising_edge(clk);

          file_close(stimuli0_file);
          file_close(stimuli1_file);
          file_close(stimuli2_file);
          file_close(stimuli3_file);
          data_fileIn0_out <= (others => 'Z');
          data_fileIn1_out <= (others => 'Z');
          data_fileIn2_out <= (others => 'Z');
          data_fileIn3_out <= (others => 'Z');
          wr0_en    <= '0';
          wr1_en    <= '0';
          wr2_en    <= '0';
          wr3_en    <= '0';

          wait;
       end process;

    -- ****************************************************************
    -- ***************************  PROCESS 2 : **********************
    -- ********************  control.txt : Registres *****************
    -- ****************************************************************
```

```
REG_FILE : process

   file control_reg : text open read_mode is FILE_CTL;

   variable reg_line : line;
   variable line_reg : std_logic_vector(38 downto 0);

begin

     -- ***************  INITIALISATION *******************

   reg_we <= '0';
   reg_ce <= '0';
   reg_addr <= (others => 'Z');
   reg_data <= (others => 'Z');

           -- ************* BEGIN **************

   wait until (reset='0');
   while(not endfile(control_reg)) loop
      wait until rising_edge(clk);
      readline(control_reg,reg_line);
      read(reg_line,line_reg);
      reg_addr <= line_reg(37 downto 32);
      reg_data <= line_reg(31 downto 0);
      reg_ce <= line_reg(38);
      reg_we <= '1';
   end loop;
   wait until rising_edge(clk);
   reg_we <= '0';
   reg_ce <= '0';
   reg_addr <= (others => 'Z');
   reg_data <= (others => 'Z');

   wait until rising_edge(clk);
   file_close(control_reg);

end process;


-- ************************************************************
-- ******************** PROCESS 3 : CASE ********************
-- ************************************************************

CASE_REG : process(clk,reset)
begin
   if reset='1' then
      read_size  <= (others=>'0');
      start      <= '0';
      raz        <= '1';
   elsif rising_edge(clk) then
      raz <= '0';
      if reg_we='1' and reg_ce='0' then
         case reg_addr(5 downto 0) is
            when "000000" =>
               read_size <= reg_data;
               start     <= '1';
            --when "000001" =>
            -- A COMPLETER
            when others =>
               NULL;
         end case;
      end if;
      if(out_end='1') then
          raz <= '1';
      end if;
   end if;
end process;
```

```
-- ************************************************************
-- ******************* PROCESS 4 : COMPTEUR  *****************
-- ************************************************************
COMPTEUR : process(clk,reset)
begin
   if(reset='1') then
      read_count <= (others=>'0');
   elsif rising_edge(clk) then
      if(raz='1') then
         read_count <= (others=>'0');
      elsif(rdfifo_en='1') then
         read_count <= read_count + 1;
      end if;
   end if;
end process;

-- ************************************************************
-- ************************  PROCESS 5 : *********************
-- ************ lecture des donnees de la fifo ***************
-- ***************    => envoie au filtre     ****************
-- ************************************************************

DATA_FIFO : process(reset,clk,valid0,valid1,valid2,valid3,
                    in_ready,read_count,read_size,start,rdfifo_en)

begin
   if not(read_count=read_size) then
       rdfifo_en <= in_ready and start;
   else
      rdfifo_en <= '0';
   end if;
   rd0_en <= rdfifo_en;
   rd1_en <= rdfifo_en;
   rd2_en <= rdfifo_en;
   rd3_en <= rdfifo_en;
   if (not(read_count=X"00000000") and read_count=read_size-1) then
      in_end <= '1';
   else
      in_end <= '0';
   end if;
   in_valid <= valid0 and valid1 and valid2 and valid3 and start;
end process;

-- ************************************************************
-- ************************  PROCESS 6 : *********************
-- *********  ecriture dans fifo64 à partir du filtre **********
-- ************************************************************

DATAOUT_FILTER : process(full,out_valid,out_end)

begin
   out_ready <= (not full) and out_valid;
   wr_en <= (not full) and out_valid;
   stop <= out_end;
end process;

-- ******************************************************************
-- ***************************  PROCESS 7 : **************************
-- *****  ecriture des donnees de la fifo dans fichier resultat ******
-- ******************************************************************

DATAOUT_FILE : process

   file res_file : text open write_mode is FILE_OUT;
   variable Lout : Line;

begin
```

```
      rd_en <= '0';
      wait until (reset = '0');

      loop
         if (valid='1') then
            rd_en <= '1';
            write(Lout, data_fifo_64_out);
            writeline(res_file,Lout);
         else
            rd_en <= '0';
         end if;
         wait until rising_edge(clk);
      end loop;

      wait until rising_edge(clk);

      file_close(res_file);
      wait;
   end process;

   -- ************************************************************************
   -- ********************** INSTANTIATION DES COMPOSANTS ******************
   -- ************************************************************************

   inst_filter : filter
      port map(
               reset      => reset,
               clk        => clk,
               in_valid   => in_valid,
               in_ready   => in_ready,
               in_end     => in_end,
               in_data0   => data_fifo0_out,
               in_data1   => data_fifo1_out,
               in_data2   => data_fifo2_out,
               in_data3   => data_fifo3_out,
               reg_ce     => reg_ce,
               reg_we     => reg_we,
               reg_addr   => reg_addr,
               reg_data   => reg_data,
               out_valid  => out_valid,
               out_ready  => out_ready,
               out_end    => out_end,
               out_data   => out_data
               );

   inst_fifo_32_0 : fifo_32
      port map(
               din    => data_fileIn0_out,
               rd_clk => clk,
               rd_en  => rd0_en,
               rst    => reset,
               wr_clk => clk,
               wr_en  => wr0_en,
               dout   => data_fifo0_out,
               empty  => empty0,
               full   => full0,
               valid  => valid0
               );

   inst_fifo_32_1 : fifo_32
      port map(
               din    => data_fileIn1_out,
               rd_clk => clk,
               rd_en  => rd1_en,
               rst    => reset,
               wr_clk => clk,
               wr_en  => wr1_en,
               dout   => data_fifo1_out,
```

```
                    empty  => empty1,
                    full   => full1,
                    valid  => valid1
                    );

         inst_fifo_32_2 : fifo_32
            port map(
                    din    => data_fileIn2_out,
                    rd_clk => clk,
                    rd_en  => rd2_en,
                    rst    => reset,
                    wr_clk => clk,
                    wr_en  => wr2_en,
                    dout   => data_fifo2_out,
                    empty  => empty2,
                    full   => full2,
                    valid  => valid2
                    );
550
         inst_fifo_32_3 : fifo_32
            port map(
                    din    => data_fileIn3_out,
                    rd_clk => clk,
                    rd_en  => rd3_en,
                    rst    => reset,
                    wr_clk => clk,
                    wr_en  => wr3_en,
                    dout   => data_fifo3_out,
                    empty  => empty3,
                    full   => full3,
                    valid  => valid3
                    );

         inst_fifo_64 : fifo_64
            port map(
                    din    => out_data,
                    rd_clk => clk,
                    rd_en  => rd_en,
                    rst    => reset,
                    wr_clk => clk,
                    wr_en  => wr_en,
                    dout   => data_fifo_64_out,
                    empty  => empty,
                    full   => full,
                    valid  => valid
                    );
      -- ************************************************************************
      -- ******************************   FIN   *********************************
      -- ************************************************************************


   end BEH;

   configuration cfg_TB_click_here of filter_tb is
   for BEH
   end for;
   end cfg_TB_click_here;
```

6.2 Exemple d'un Opérateur ReMIX : un filtre simple

6.2.1 filtre : "filter.vhd"

```
   -- ************************************************************
   --                    fichier d'un opÃl'rateur REMIX:
   --
   --      Filtre Simple retournant la somme des donnÃl'es en entrÃl'es
```

```
--      lorsqu'elle dÃl'passe un certain Seuil donnÃl'
--
--              dans le cadre du projet ReMIX/GAUT(Lester)
--              Responsable : Dominique LAVENIER
--              Encadrant : Gilles GEORGES
-- by HÃl'lÃÍne DAROLLES
-- IRISA - Symbiose - mai 2006
-- ************************************************************

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;

entity filter is
 port (
        reset     : IN std_logic;
        clk       : IN std_logic;
        in_valid  : IN std_logic;
        in_ready  : out std_logic;
        in_end    : IN std_logic;
        in_data0  : IN std_logic_VECTOR(0 to 31);
        in_data1  : IN std_logic_VECTOR(0 to 31);
        in_data2  : IN std_logic_VECTOR(0 to 31);
        in_data3  : IN std_logic_VECTOR(0 to 31);
        reg_ce    : IN std_logic;
        reg_we    : IN std_logic;
        reg_addr  : IN std_logic_VECTOR(0 to 5);
        reg_data  : IN std_logic_VECTOR(0 to 31);
        out_ready : IN std_logic;
        out_valid : OUT std_logic;
        out_end   : OUT std_logic;
        out_data  : OUT std_logic_VECTOR(0 to 63));
end filter;

architecture IMP of filter is
-------------------------------------------------------------------------------
-- Begin architecture
-------------------------------------------------------------------------------
component datapath
   port(
        reset_decod  : in std_logic;
        reset_regS   : in std_logic;
        clk          : in std_logic;
        data0        : in std_logic_vector(31 downto 0);
        data1        : in std_logic_vector(31 downto 0);
        data2        : in std_logic_vector(31 downto 0);
        data3        : in std_logic_vector(31 downto 0);
        depasseSeuil : out std_logic;
        enable       : in std_logic;
        out_data     : out std_logic_vector(0 to 63);
        reg_ce   : in std_logic;
        reg_we   : in std_logic;
        reg_addr : in std_logic_VECTOR(0 to 5);
        reg_data : in std_logic_VECTOR(0 to 31)
       );
end component;

component ctl
   port(
        reset        : IN std_logic;
        clk          : IN std_logic;
        in_valid     : IN std_logic;
        in_ready     : out std_logic;
        in_end       : IN std_logic;
        depasseSeuil : in std_logic;
        out_ready    : IN std_logic;
        out_valid    : OUT std_logic;
```

```
        out_end      : OUT std_logic;
        reset_out    : out std_logic;
        enable       : out std_logic
       );
   end component;
----------------  DECLARATION DES SIGNAUX INTERNES --------------------
SIGNAL reset_regS   : std_logic;
SIGNAL depasseSeuil : std_logic;
SIGNAL valide       : std_logic;
----------------  INSTANCIATIONS --------------------
begin -- architecture IM

inst_datapath : datapath
port map(
        reset_decod  => reset,
        reset_regS   => reset_regS,
        clk          => clk,
        data0        => in_data0,
        data1        => in_data1,
        data2        => in_data2,
        data3        => in_data3,
        depasseSeuil => depasseSeuil,
        enable       => valide,
        reg_ce       => reg_ce,
        reg_we       => reg_we,
        reg_addr     => reg_addr,
        reg_data     => reg_data,
        out_data     => out_data
        );

inst_ctl : ctl
port map(
        reset        => reset,
        clk          => clk,
        in_valid     => in_valid,
        in_ready     => in_ready,
        in_end       => in_end,
        depasseSeuil => depasseSeuil,
        enable       => valide,
        out_ready    => out_ready,
        out_valid    => out_valid,
        out_end      => out_end,
        reset_out    => reset_regS
        );

end architecture IMP;
```

6.2.2 Partie datapath du filtre : "datapath.vhd"

```
-- ************************************************************
--       fichier de la partie datapath de l'opÃl'rateur REMIX
--            dans le cadre du projet ReMIX/GAUT(Lester)
--              Responsable : Dominique LAVENIER
--              Encadrant : Gilles GEORGES
-- by HÃl'lÃÍne DAROLLES
-- IRISA - Symbiose - mai 2006
-- ************************************************************

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity datapath is
port (
      reset_decod  : in std_logic;
      reset_regS   : in std_logic;
```

```
        clk          : in std_logic;
        data0        : in std_logic_vector(31 downto 0);
        data1        : in std_logic_vector(31 downto 0);
        data2        : in std_logic_vector(31 downto 0);
        data3        : in std_logic_vector(31 downto 0);
        depasseSeuil : out std_logic;
        enable       : in std_logic;
        out_data     : out std_logic_vector(0 to 63);
        reg_ce       : in std_logic;
        reg_we       : in std_logic;
        reg_addr     : in std_logic_VECTOR(0 to 5);
        reg_data     : in std_logic_VECTOR(0 to 31)
        );
end datapath;

architecture imp of datapath is

component add32
   port(
         data0    : in std_logic_vector(31 downto 0);
         data1    : in std_logic_vector(31 downto 0);
         data_out : out std_logic_vector(32 downto 0)
        );
end component;

component add33
   port(
         data0    : in std_logic_vector(32 downto 0);
         data1    : in std_logic_vector(32 downto 0);
         data_out : out std_logic_vector(63 downto 0)
        );
end component;

component add64
   port(
         data0    : in std_logic_vector(63 downto 0);
         data1    : in std_logic_vector(63 downto 0);
         data_out : out std_logic_vector(63 downto 0)
        );
end component;

component regS
   port(
         datain  : in std_logic_vector(0 to 63);
         dataout : out std_logic_vector(0 to 63);
         clk     : in std_logic;
         enable  : in std_logic;
         reset   : in std_logic
        );
end component;

component decod
    port(
          reset     : in std_logic;
          clk       : in std_logic;
          we        : in std_logic;
          ce        : in std_logic;
          addr      : in std_logic_vector(0 to 5);
          data      : in std_logic_vector(0 to 31);
          seuil_out : out std_logic_vector(0 to 31)
          );
end component;

component from32to64
   port(
          datain  : in std_logic_vector(0 to 31);
          dataout : out std_logic_vector(0 to 63);
          clk     : in std_logic
```

```
            );
    end component;

    component cmp
100    port(
            data0 : in std_logic_vector(63 downto 0);
            data1 : in std_logic_vector(63 downto 0);
            Sup   : out std_logic
           );
    end component;

    ----------------  DECLARATION DES SIGNAUX INTERNES --------------------
    SIGNAL somme01      : std_logic_vector(32 downto 0);
    SIGNAL somme23      : std_logic_vector(32 downto 0);
    SIGNAL somme0123    : std_logic_vector(63 downto 0);
    SIGNAL sadd64       : std_logic_vector(63 downto 0);
    SIGNAL somme        : std_logic_vector(63 downto 0);
    SIGNAL valeurseuil  : std_logic_vector(63 downto 0);
    SIGNAL valseuil     : std_logic_vector(31 downto 0);

    begin -- architecture IMP

    ----------------  INSTANCIATIONS --------------------

    inst_add32_1 : add32
    port map(
             data0    => data0,
             data1    => data1,
             data_out => somme01
            );

    inst_add32_2 : add32
    port map(
             data0    => data2,
             data1    => data3,
             data_out => somme23
            );

    inst_add33 : add33
    port map(
             data0    => somme01,
             data1    => somme23,
             data_out => somme0123
            );

    inst_add64 : add64
    port map(
             data0    => somme,
             data1    => somme0123,
             data_out => sadd64
            );

    inst_regS : regS
    port map(
150          datain  => sadd64,
             dataout => somme,
             enable  => enable,
             clk     => clk,
             reset   => reset_regS
            );


    inst_decod : decod
    port map(
             reset     => reset_decod,
             clk       => clk,
             we        => reg_we,
             ce        => reg_ce,
```

```
          addr      => reg_addr,
          data      => reg_data,
          seuil_out => valseuil
          );

inst_32to64 : from32to64
port map(
          datain  => valseuil,
          dataout => valeurseuil,
          clk     => clk
          );

inst_cmp : cmp
port map(
          data0  => somme,
          data1  => valeurseuil,
          Sup    => depasseSeuil
         );

out_data <= somme;
end imp;
```

6.2.3 Partie contrôle du filtre : "ctl.vhd"

```
-- ************************************************************
--       fichier de la partie conrÃt'le de l'opÃl'rateur REMIX
--            dans le cadre du projet ReMIX/GAUT(Lester)
--              Responsable : Dominique LAVENIER
--              Encadrant : Gilles GEORGES
-- by HÃl'lÃÍne DAROLLES
-- IRISA - Symbiose - mai 2006
-- ************************************************************

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;

entity ctl is
port (
      reset        : in std_logic;
      clk          : in std_logic;
      in_valid     : in std_logic;
      in_ready     : out std_logic;
      in_end       : in std_logic;
      depasseSeuil : in std_logic;
      out_ready    : in std_logic;
      out_valid    : out std_logic;
      out_end      : out std_logic;
      enable       : out std_logic;
      reset_out    : out std_logic
     );
end entity ctl;

architecture mealy of ctl is

-------------------------------------------------------------------------------
-- Begin architecture
-------------------------------------------------------------------------------

type state_type is (INIT,IDLE,VALIDE,FIN);
signal EP, EF       : state_type;
signal data_valid   : std_logic;
signal sortie_valid : std_logic;
signal sortie_fin   : std_logic;
signal reset_somme  : std_logic;
```

```
begin -- architecture IMP

CLKRST : process (clk,reset)
begin
     if (reset='1') then EP <= INIT;
     elsif (clk'event and clk = '1') then EP <= EF;
     end if;
end process;

-- fonction de transition

INPUT : process (EP, in_valid, in_end, out_ready, depasseSeuil)
begin
     case EP is
               when INIT =>
                   EF <= IDLE;
               when IDLE =>
                  if(in_valid='1') then
                          EF <= VALIDE;
                  else
                          EF <= IDLE;
                  end if;
               when VALIDE =>
                  if (in_valid='1' and in_end='1') then
                          EF <= FIN;
                  else
                          EF <= VALIDE;
                  end if;
               when FIN =>
                   if(out_ready='1' and depasseSeuil='1') then
                       EF <= IDLE;
                   elsif(depasseSeuil='0') then
                       EF <= IDLE;
                   else
                       EF <= FIN;
                   end if;
     end case;
end process;

-- fonction de generation

OUTPUT : process (EP, in_valid, in_end, out_ready, depasseSeuil)
begin
   case EP is
        when INIT =>
           data_valid <= '0';
           sortie_valid <= '0';
           sortie_fin <= '0';
           reset_somme <= '1';
        when IDLE =>
           data_valid <= '0';
           sortie_valid <= '0';
           sortie_fin <= '0';
           reset_somme <= '0';
        when VALIDE =>
           data_valid <= in_valid;
           sortie_valid <= '0';
           sortie_fin <= '0';
           reset_somme <= '0';
        when FIN =>
           data_valid <= '0';
           sortie_valid <= depasseSeuil;
           sortie_fin <= '1';
           reset_somme <= out_ready;
   end case;
end process;

enable <= data_valid;
```

```
in_ready <= data_valid;
out_valid <= sortie_valid;
out_end <= sortie_fin;
reset_out <= reset_somme;

end architecture mealy;
```