Chapitre XI- Etude de processeurs industriels.

Objectifs du chapitre :

- Valider SEP pour son application à des processeurs industriels (PINE, OAK+ et ARM7).
- Application à une architecture bi-cœur (3771).

Ce chapitre présente une étude réalisée dans le but de valider SEP sur une application industrielle. Le cœur de processeur PINE – processeur de DSP Group – n'est plus utilisé par VLSI Technology dans ses applications industrielles depuis environ quatre années. Cependant le PINE est le prédécesseur du OAK et du OAK+ – processeurs de DSP Group –, et leurs architectures sont très proches.

C'est pourquoi nous avons dans un premier temps modélisé le processeur PINE et nous avons validé ce modèle sur trois parties critiques d'applications de traitement numérique du signal, une application de filtrage de type FIR (Finite Impulse Response), un papillon de FFT (Fast Fourier Transform) et un papillon de Viterbi. La section 1- présente le résultat de cette étude.

Dans un deuxième temps, nous avons voulu modéliser une architecture bi-cœur de VLSI Technology bâptisée 3771. Cette architecture est constituée d'un cœur de OAK+, d'un cœur de ARM7 – processeur de ARM Limited inc. – et d'une interface matérielle de communication. C'est ainsi que nous avons adapté le modèle de PINE pour construire le OAK+ (cf. section 3-), nous avons alors modélisé le ARM7 (cf. section 2-) et l'interface de communication (cf. section 3-). Le modèle de cette architecture a été validé avec une application caractéristique de communication par interruptions matérielles.

L'information technique utilisée a été reconstruite à partir de différentes sources d'information et en particulier à partir de la documentation technique **[VLSI96] [DSP97] [ARM95]**.

1- Le processeur PINE.

1-1. L'architecture générale.

Le processeur PINE est un processeur de traitement numérique du signal (*DSP*). L'architecture qui nous intéresse est constituée du cœur du PINE et de la mémoire d'instructions qui contient le programme à exécuter lors du *reset* (cf. Figure 1). Le bus PPAN porte les adresses vers la mémoire d'instructions, puis sur front montant sur le signal read, l'instruction est délivrée sur le bus IDP. Sur tous les modèles présentés nous indiquerons la taille des bus et des registres bien que SEP n'utilise pas, la plupart du temps, cette information.

C'est une architecture de type Harvard avec deux mémoires de données (X et Y). Le cœur du PINE est composé de trois modules. Le module de calcul (*CU*), le module de génération d'adresses (DAU) pour les mémoires de données et le module de contrôle (PCU).

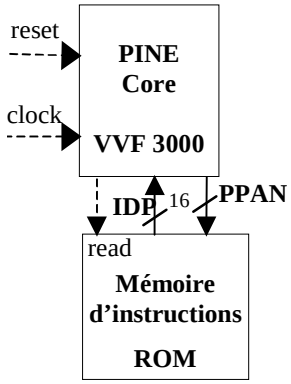


Figure 1 – Cœur étendu du PINE : « combo »

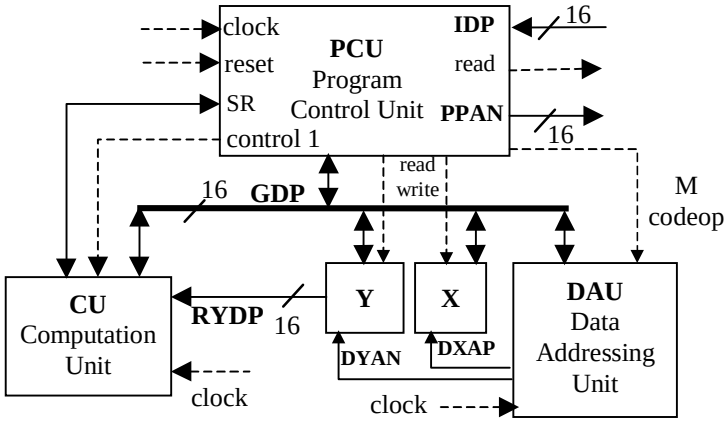


Figure 2 – PINE Core - VVF3000.

Le DAU transmet les adresses sur 10 bits vers les bus d'adresses DXAP et DYAN. Ces unités communiquent à travers un bus général GDP et des signaux de contrôle. La mémoire Y peut délivrer une donnée soit sur le bus global GDP soit sur un bus propre RYDP.

Enfin de l'information sur l'état du processeur est échangée par le registre d'état SR du PCU. Les registres d'états contiennent notamment les *flags* en provenance de l'ALU, et les bits d'extension des accumulateurs. De plus, ils contiennent de l'information de configuration comme le bit SP (Shift Pointer), les bits M (modulo) et les codes opération à destination de l'ALU ou du module d'adressage.

La Figure 2 illustre cette configuration.

La mémoire d'instructions est une mémoire de type ROM, les deux mémoires X et Y sont des mémoires de type RAM. Ces trois mémoires sont construites avec la procédure indiquée au chapitre III- section 3-. La mémoire Y est construite avec deux services le lecture (*read*) qui partagent la même entrée d'adresse et qui ont deux sorties de données distinctes, une sortie vers le bus GDP et une sortie vers le bus RYDP.

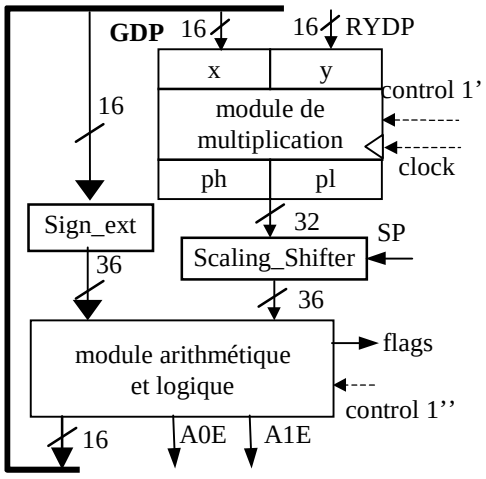


Figure 3 – Le module de calcul : CU.

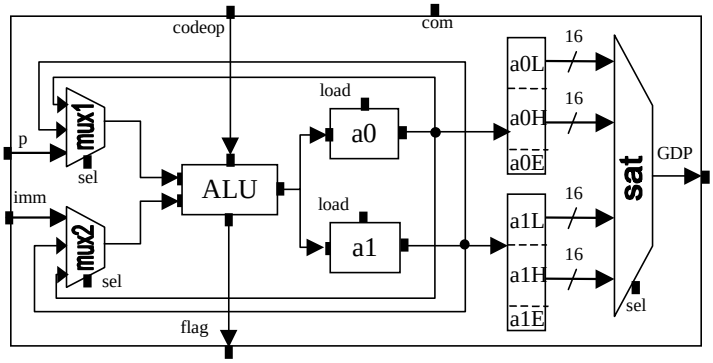


Figure 4 – Le module arithmétique et logique : ALB.

1-2. Le module de calcul : CU.

Le module de calcul (cf. Figure 3) a été présenté au chapitre VII. Pour des raisons de lisibilité, les schémas sont rappelés et détaillés dans cette section.

Ce module est constitué d'un module de multiplication *MU* (cf. Figure 5), d'un module arithmétique et logique *ALB* (cf. Figure 4) et deux composants élémentaires.

sign_ext

Le composant *sign_ext* permet une extension de signe de données codées sur 16 bits transformées en données de valeur et de signe identiques codées sur 36 bits. Etant donné que le codage binaire des données n'est pas utilisé à ce niveau de modélisation, l'intégration de ce composant dans le modèle n'est pas utile.

scaling_ shifter

Le composant *scaling_shifter* a le même rôle excepté qu'en plus, il décale éventuellement la donnée d'entrée de 0, 1 ou 4 bits sur la droite ou sur la gauche, c'est-à-dire effectue une multiplication ou une division par 1, 2 ou 16 suivant la valeur de l'identificateur SP.

Considérons maintenant le module ALB (cf. Figure 4**).**

Les composants multiplexeur et ALU ont déjà été définis dans le chapitre X. Cependant l'ALU a besoin d'être améliorée afin de fournir un plus grand nombre d'identificateurs d'opération (*flags*). Pour cela, il y a plusieurs possibilités. Dans un premier temps, il faut nécessairement construire les nouveaux services de calcul des nouveaux *flags*. Pour cela, étant donné que l'ajout n'entraîne aucune modification des services existants nous modifions directement le fournisseur de services UAL défini au chapitre X :

```
public class ALU implements sep.model.ServiceProvider {
  protected Value res = LevelValue.Undefined;
  public Value execute(Value[] values, String cop) {
        return res = sep.type.Type.perform(cop, values);
  }
  public boolean lt() { // Less than
        return ((LevelValue)sep.type.Type.perform("Lt", res, new IntValue(0))).isHigh();
  }
  public boolean gt() { // Greater than
        return ((LevelValue)sep.type.Type.perform("Gt", res, new IntValue(0))).isHigh();
  }
  public boolean equal() { return res.equals(new IntValue(0)); }
  public boolean le() {  return !gt(); }  // Less or equal
  public boolean ge() { return !lt(); }  // greater or equal
}
```

Notons qu'il était aussi possible de définir un nouveau fournisseur de services ALU héritant de la classe UAL.

A partir de ce fournisseur de services, il est possible de créer un composant ALU qui a toujours un service combinatoire *execute* sur n entrées *in* et émet sa sortie sur le port *out*.

Les services de *flags* (*lt, gt, equal, le, ge*) sont des services exécutés après chaque exécution du service *execute*. Ils émettent tous leur sortie sur le port de sortie *flags* avec une fonction de résolution de type *Multiple*. Cette fonction de résolution, prend la liste des sorties *flags* et les agrègent en une valeur multiple (liste ordonnée de plusieurs valeurs) :

```
public class Multiple implements Resolution {
        public Value solve(Value values []) { return new MultipleValue(values) ; }
}
```

SAT

Le composant SAT utilise le fournisseur de services *Multiplexeur*(cf. chapitre X), mais comme service séquentiel déclenché à chaque changement de valeur sur le port *sel*.

accumulateur

Les composants *accumulateurs* pourraient être modélisés par des modules par composition d'unités arithmétiques, mais pour des raisons d'efficacité, nous avons choisi de les modéliser en définissant une nouveau fournisseur de service appelé *Accu* qui hérite de *Registre*. La

solution idéale serait de modéliser par composition de composants et d'effectuer les optimisations automatiquement lors de la compilation et lors de la création de la classe Java qui représente le module créé.

Un composant accumulateur a toujours un service *load* identique au registre, il a de plus un service *loadL* qui charge l'entrée dans la partie basse (16 bits) de la valeur mémorisée et un service *loadH* qui charge l'entrée dans la partie haute (16 bits). Ce composant a de plus deux paramètres *LowSize* et *HighSize* qui permettent de modifier la taille des parties haute et basse. Les services utilisent l'opérateur *BitValue slice (BitValue v, int debut, int taille)* qui décale la valeur *v* de *debut* bit sur la droite et conserve *taille* bits parmi les bits restants.

Voici le code Java du fournisseur de service *Accu* :

```
public class Accu extends Registre { // on hérite de la méthode load et de l'attribut content.
    public BitValue load(BitValue in) { return (BitValue)super.load(in); }
    public BitValue loadL(BitValue in) { // content&(2^36-2^16) + in % (2^16-1).
        return load(sep.type.Type.perform("add", in, getSlice(content, getLowSize(), getHighSize()+4)));
    }
    public Value loadH(Value in) { // content&(2^16-1) + in%(2^16-1)*2^16 + content&(2^36-2^32).
       return load(sep.type.Type.perform("add", getSlice(content, 0, getLowSize()),
                                                 getSlice(content, getLowSize()+getHighSize(), 4),
                                            sep.type.Type.perform("shr", in, 16)));
    }
    public int getLowSize() { return lowSize; } public void setLowSize(int s) { lowSize = s; }
    public int getHighSize() { return highSize; } public void setHighSize(int s) { highSize = s; }
    private BitValue getSlice(BitValue v, int debut, int taille) {
        return sep.type.Type.perform("slice", v, debut, taille);
    }
}
```

Enfin, le composant qui décompose la valeur d'un accumulateur en ses parties basse, haute et étendue est appelé *slice*, il utilise l'opérateur arithmétique *slice* définit précédemment. Remarquons que pour être indépendant du codage binaire, il suffit de remplacer les opérations de décalage de bits par des fonctions de multiplication ou de division par des puissances de 2.

Considérons pour finir le module MU (cf. Figure 5**).**

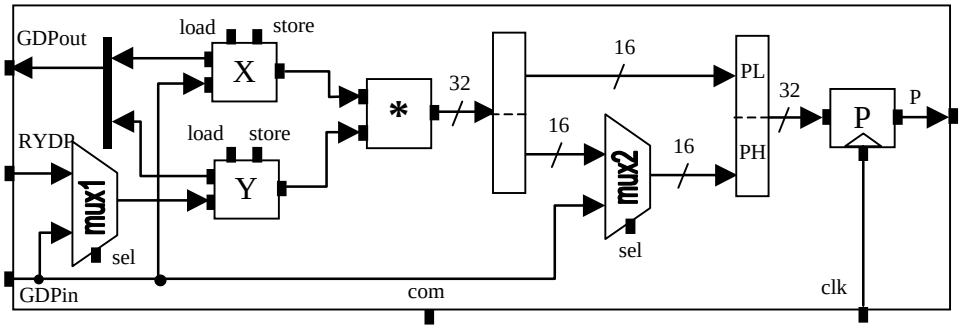


Figure 5 – Le module de multiplication : MU.

multiplieur

Le composant *multiplieur* * est aussi réalisé par un composant UAL. En effet, il effectue une opération sur n opérandes et cette opération est définie dans l'arbre de sous-typage.

Le seul composant qui reste à définir est le composant *concat* qui à partir de n entrées in_1, in_2, …, in_n de k_1, k_2, …, k_n bits construit une sortie de $k_1+k_2+\ldots+k_n$ bits par concaténation des entrées. Ce composant est un composé de composants arithmétiques pour réaliser la fonction :

$$in_1 + \sum_{i=2}^{n} (in_i \times 2^{\sum_{j=1}^{i-1} k_j}).$$

1-3. Le module de contrôle : PCU.

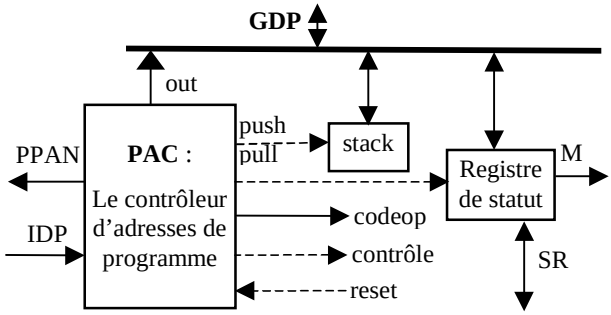


Figure 6 – Le module de contrôle.

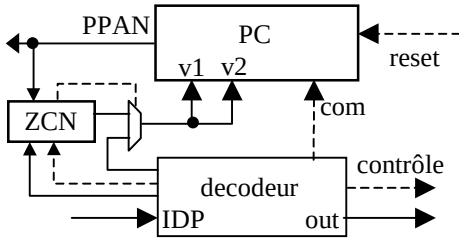


Figure 7 – Le PAC.

Le module de contrôle (cf. Figure 6) est constitué d'un registre de statut, d'une pile matérielle (*stack*) et d'un contrôleur (*PAC*).

registre de statut

Le registre de statut est un simple registre pour lequel il est possible de sélectionner une partie de l'information par des sélections de bits.

stack

La pile matérielle (*stack*) est un composant séquentiel avec deux services *push* et *pull* qui permettent respectivement d'empiler ou de dépiler une donnée.

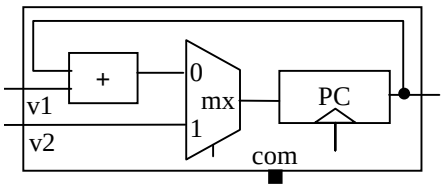
Le fournisseur de service *Stack* est décrit ci-dessous. En fait, une pile est une mémoire où l'accès ne se fait que par le sommet de la pile.

```
public class Stack extends Memory {
        protected int top = 1 ;
        public void push(Value v) { write(top++, v) ; }
        public Value pull() { return read(--top) ; }
}
```

Nous choisissons de rendre le service *push* plus prioritaire que le service *pull*.

contrôleur

Le contrôleur (cf. Figure 7) est un module constitué du module PC (cf. figure ci-contre), d'un décodeur générique dont le code SEP-ISDL est donné en annexe et d'une unité de gestion des boucles matérielles (ZCN).



PC

Le module PC possède les services de module ***pc:=pc+v1*** et ***pc:=v2***.

ZCN

Enfin, le bloc *ZCN* permet de gérer deux types de boucles matérielles. Une boucle *rep* qui répète n fois une instruction. Un registre *lcLow* permet de mémoriser le nombre n de répétitions. Une boucle bkrep qui répète m fois un bloc d'instructions. Un registre *lcHigh* permet de mémoriser le nombre m de répétitions. Deux registres nous permettent de mémoriser l'adresse de la première et de la dernière instruction du bloc.

Voici le code du fournisseur de service ZCN :

```
public class ZCN implements sep.model.ServiceProvider {
        public final int NORMAL=0, BOUCLE=1;
        protected int lcLow = -1, lcHigh = -1, last, first;
        protected Value out = LevelValue.Undefined;

        public void rep(int n) { lcLow = n; }
        public void bkrep(int m, int last) { lcHigh = m; this.last = last; first = -1; }
```

```
	public int next(int pc) {
		if (lcLow>=0) {
			out = new IntValue(0);
			lcLow--;
			return BOUCLE;
		} else if (lcHigh>=0) {
			if (first == -1)
				first = pc;
			else if (pc==last) {
				out = new IntValue(first-last);
				lcHigh--;
				return BOUCLE;
			}
			return NORMAL;
		}
	}
	public Value output() { return out; }
}
```

Le composant ZCN est un composant séquentiel. Il a quatre services : *rep*, *bkrep*, *next* et *output*.

Le service *next* est déclenché par un signal de type *posedge* sur le port *next*. Ce port doit être connecté à l'horloge du processeur. Son paramètre *pc* est fourni par l'unité PC, c'est l'adresse de l'instruction qui va être exécutée. Sa sortie sélectionne un multiplexeur qui permet de choisir la valeur de saut (cf. Figure 7). Quand il n'y a pas de boucle matérielle (valeur de retour NORMAL), c'est le décodeur qui fixe l'adresse de la prochaine instruction à exécuter. Quand il y a une boucle matérielle (valeur de retour BOUCLE), c'est le composant ZCN qui fixe l'adresse de la prochaine instruction à exécuter.

Le service *rep* permet d'initialiser une boucle *rep*. Pour cela, il suffit de fixer le nombre d'itérations. A partir de la prochaine instruction, c'est-à-dire à partir de l'instruction à répéter, et pendant *lcLow* appels, le ZCN choisit pc+0 comme prochaine instruction à exécuter.

Le service *bkrep* permet d'initialiser une boucle *bkrep*. Il est nécessaire de fixer le nombre d'itérations ainsi que l'adresse de la dernière instruction du bloc. A chaque appel du service next, si pc est différent de la dernière instruction du bloc, alors le décodeur choisit la prochaine instruction à exécuter (pc+1), sinon le ZCN choisit pc+(first-last), c'est-à-dire *first*, comme adresse pour la prochaine instruction à exécuter.

Le service *output* est un service contrôlé par le service *next*, il est déclenché après chaque appel du service *next*. En fait, c'est ce service qui sélectionne l'adresse de la prochaine instruction à exécuter quand le ZCN est en boucle.

1-4. Le module de génération d'adresses : DAU.

C'est un module assez compliqué de l'ordre d'une dizaine de milliers de portes logiques.

Ce composant contient huit registres, les six registres généraux (r0, r1, r2, r3, r4, r5) et deux registres de configuration (cfgi, cfgj). A partir du contenu de ces registres, il peut générer soit une adresse vers un des trois bus de sortie (DXAP, DYAN, GDP), soit deux adresses simultanément, la première est alors obligatoirement envoyée sur le bus DXAP et la deuxième sur le bus DYAN.

De plus, ce composant est capable d'effectuer lors de la génération d'une adresse une modification incrémentale de la valeur des registres adressés.

Enfin, l'opération d'incrémentation peut être effectuée modulo un nombre contenu dans les registres de configuration.

En définitive, ce composant est une mémoire évoluée de 8 registres, le fournisseur de services *DAU* hérite de la classe *Memory* définie au chapitre III. Voici le code Java de ce fournisseur de services :

```
public class DAU extends Memory {
        public final int CFGI = 6, CFGJ=7;
        protected int [] incr = new int [8];
        public void setSize(int size) { setSize(8); }
        protected int readInt(int adr) { return ((IntValue)read(adr)).intValue(); }
        protected void write(int adr, int value) { write(adr, new IntValue(value)); }

        public IntValue read(int adr, String mode, int modulo) {
// mode d'incrémentation : +0, +1, -1, +S
            int res = readInt(adr);
            if (mode.equals("+1")) {
                incr[adr]++;
                write(adr, res+1);
            } else if (mode.equals("-1")) {
                incr[adr]--;
                write(int adr, res-1);
            } else if (mode.equalsIgnoreCase("+S")) {
                int tmp = 1;
                if (adr<4) tmp = read(CFGI) % 128; // 7 bits de poids faible
                else if (adr<CFGI) tmp = read(CFGJ) % 128;
                incr[code] += tmp;
                write(adr, res+tmp);
            } else // No Modulo
                return new IntValue(res);

            if ((modulo & (1<<adr)) != 0) { // Auto Modulo
                int tmp = 0;
                if (code<4) tmp = read(CFGI) >> 7;
                else if (code<CFGI) tmp = read(CFGJ) >> 7;
                if (tmp!=0 && incr[adr]%(tmp+1) == 0) {
                        write(adr, res-incr[adr]);
                        incr[adr] = 0;
                }
            }
            return new IntValue(res);
 }
```

Le composant DAU est un composant séquentiel muni d'un service d'écriture *write*. Ce service est identique au service *write* d'une mémoire excepté qu'il ne manipule que des entiers. L'information sur le type est utilisée (cf. chapitre VI) pour valider la composition de composants.

De plus, ce composant a trois services *read*. Les valeurs de retour des services sont associées aux ports DXAP, DYAN et GDP. Pour chaque service les paramètres sont associés à des ports multiples.

Le paramètre *adr* entier est un nombre qui représente le registre à manipuler. Les entiers de 0 à 5 représentent les registres r0 … r5. Les entiers 6 et 7 représentent respectivement les registres de configuration CFGI et CFGJ. Le paramètre *mode* est une chaîne de caractères qui représente le mode d'incrémentation. Ce paramètre peut prendre quatre valeurs possibles ("", "+1", "-1", "+s") correspondant respectivement à : aucune modification, une incrémentation de 1, une décrémentation de 1 et une incrémentation par une valeur représentée par les sept bits de poids faibles du registre CFGI pour les registres r0, r1, r2, r3, du registre CFGJ pour les registres r4 et r5.

1-5. Les applications.

Cette section présente trois applications typiques en traitement numérique du signal utilisées pour valider le modèle du PINE.

- **Le filtre FIR :**

Un filtre FIR (réponse impulsionnelle finie) à L coefficients **a**, appliqué à un signal d'entrée discret **X**, produit le signal de sortie **Y** dont le n^{ième} échantillon vérifie l'équation suivante :

$$\forall n \geq 0 \; y(n) = \sum_{i=0}^{L-1} a(i) \times x(n-i)$$

On distingue alors deux types de filtres FIR, les filtres glissants (cf. Figure 8) et les filtres non-glissants (cf. Figure 9).

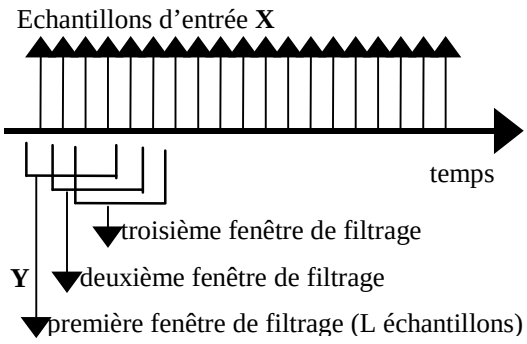


Figure 8 – Filtre FIR glissant.

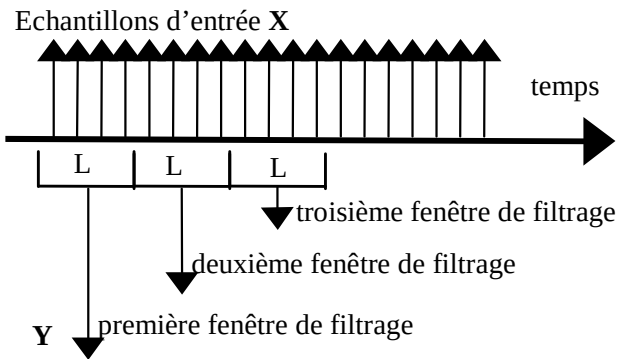


Figure 9 - Filtre FIR non glissant avec fenêtres non recouvrantes.

L'opération de base de cet algorithme est la multiplication d'une donnée d'entrée X avec un coefficient a et l'accumulation avec une valeur précédente pour obtenir après L étapes la valeur d'un échantillon de sortie. Les architectures comme le PINE ont été créées spécialement pour traiter efficacement ce type d'opérations.

```
    ; filtre glissant : on calcule dix valeurs de sorties avec 8 coefficients.
    ; r1 contient l'adresse où l'on écrit les données de sortie dans la XRAM.
    ; r4 contient l'adresse des coefficients (dans la YRAM), on utilise un adressage modulo 8
    ; L'adressage modulo 8 signifie en réalité que l'on part d'une adresse initiale quelconque, qu'on
    ; réalise des incrémentations d'une valeur au plus égale à 8 par rapport à l'adresse initiale. le
    ; registre d'état st2 permet de déclencher ou non l'auto-incrémentation matérielle automatique.
    ; r2 contient l'adresse du début de la fenêtre de filtrage.
    ; r0 contient l'adresse de la donnée d'entrée à traiter (dans la XRAM)
mov #0x11,r1
mov #0,r4
mov #0,r2

mov #0x10,st2          ; Autorise l'autoincrementation pour r4
mov ##0x0381,cfgj      ; Positionne le modulo à 8 et l'incrément à 1

bkrep #9,0015          ; 10 valeurs de sortie (boucle matérielle : répéter 9+1 fois jusqu'à 15)
mov r2,r0
moda clr,a0            ; Efface l'accumulateur de sortie
mpy (r0)+,(r4)+        ; Amorce le pipeline du mac : p = r0xr4 et r0++ et r4++.
rep #6                 ; boucle matérielle : répéter 7 fois l'instruction suivante
mac (r0)+,(r4)+,a0     ; a0 = a0 + (r0) x (r4) et r0=r0+1 et r4=r4+1
add p,a0               ; Vide le pipeline du mac
```

```
mov (a0h),(r1)+         ; On écrit le résultat dans la mémoire.
mov (a0l),(r1)+
modr (r2)+              ; fenêtre de filtrage suivante =+1 (@ 15 : fin de la boucle matérielle).
```

- **Le Papillon de FFT :**

La transformée de Fourier est une transformation de base parmi les plus importantes en traitement numérique du signal. Il existe de nombreux algorithmes pour la programmer, la plus répandue est appelée FFT (Fast Fourier Transform).

L'algorithme programmé est celui de la FFT radix 2 (papillon à deux entrées) à décimation temporelle sur des nombres complexes. Cet algorithme est basé sur une succession d'opérations élémentaires appelées papillons de FFT. La Figure 10 illustre le schéma de base d'un papillon de FFT.

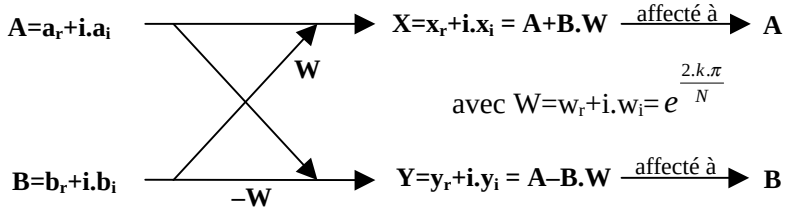


Figure 10 – Un papillon de FFT.

Soient deux nombres complexes A et B, appliquer un papillon de FFT avec le coefficient complexe W donne deux nombres complexes X et Y dont les coefficients complexes vérifient les équations suivantes :

$$x_r = a_r + (b_r.w_r - b_i.w_i)$$
$$x_i = a_i + (b_r.w_i + b_i.w_r)$$
$$y_r = a_r - (b_r.w_r - b_i.w_i) = 2.a_r - x_r$$
$$y_i = a_i - (b_r.w_i + b_i.w_r) = 2.a_i - x_i$$

L'algorithme que nous avons implémenté réalise une papillon de FFT en place, c'est-à-dire que les données calculées (X et Y) viennent remplacer en mémoire les données d'entrée (A et B).

Les données d'entrée a_r, a_i, b_r et b_i sont placées dans la mémoire X. Les coefficients w_r et w_i sont placés dans la mémoire Y. La transformée est effectuée en place, donc les sorties x_r, x_i, y_r et y_i viennent remplacer a_r, a_i, b_r et b_i dans la mémoire X.

```
; papillon de FFT.
; Initialisation des variables
mov #0,r0               ; ar
mov #2,r1               ; br
mov #0,r4               ; tr
mov #1,r5               ; ti
modA clr,a0             ; a0 = 0

mpy (r1)+,(r4)          ; p = br.tr

; Les parties réelles
mov (r0),a0l            ; a0 = ar
mov a0,a1               ; a1 = ar
moda shl,a1             ; a1 = 2*ar
mac (r1)-,(r5),a0       ; a0 = ar + br.tr,          p = bi*ti
msu (r1),(r5),a0        ; a0 = ar + br.tr - bi.ti,  p = br*ti
sub a0,a1               ; a1=a1-a0= ar - br.tr + bi.ti
mov a0l,(r0)+           ; ecrit xr en mémoire
```

```
mov a1l,(r1)+          ; écrit yr en mémoire

; Les parties imaginaires
mov (r0),a0l           ; a0 = ai
mov a0,a1              ; a1 = ai
moda shl,a1            ; a1 = 2*ai
mac (r1)-,(r4),a0      ; a0 = ai + br.ti,          p = bi*tr
mac (r1)+,(r4),a0      ; a0 = ai + br.ti + bi.tr,  p = br*tr
sub a0,a1              ; a1=a1-a0= ar - br.ti - bi.tr
mov a0l,(r0)+          ; écrit xi en mémoire
mov a1l,(r1)+          ; écrit yi en mémoire
```

- **Le papillon de Viterbi :**

L'algorithme de Viterbi a été introduit en 1967 comme une solution pour décoder les codes convolutionnels. Il consiste à choisir un chemin dans un treillis tel que la distance de Hamming entre ce chemin et la séquence des bits reçus soit minimum.

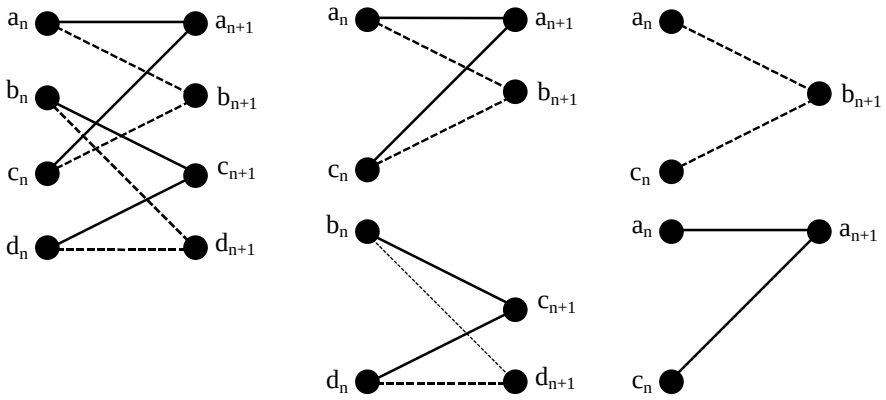


Figure 11 - Papillon de Viterbi

La partie de cet algorithme qui nous intéresse est la partie critique, à savoir le papillon. En effet, le papillon sera exécuté de nombreuses fois et le coût total d'exécution de l'application est très fortement lié au coût d'exécution du papillon. Nous avons choisi un treillis d'ordre 3 c'est-à-dire à $4 = 2^{3-1}$ états.

Le papillon peut se décomposer en 2 sous-papillons. Chaque noeud du treillis a deux valeurs de sortie et deux valeurs d'entrée. Parmi, les deux valeurs d'entrée possibles on ne retiendra que l'entrée sur le chemin de coût minimal (distance de Hamming).

Il s'agit donc de calculer à chaque étape et pour chaque noeud (4) les coûts des 2 chemins entrants, de les comparer et de conserver celui dont le coût est le plus faible.

La Figure 11 montre une représentation graphique d'un papillon de Viterbi. Cette figure montre les décompositions successives en sous-papillons qui mènent à l'opération élémentaire (segment), comparaison de deux distances de Hamming.

```
; Un segment du papillon de Viterbi.

mov    #0, r0          ; coût du noeud 'a' vaut 0
moda   clr, a0

;debut segment a0--->a1

mov    #0, r1          ; données d'entrées
moda   clr, a1         ; efface l'accumulateur = coût du segment
mov    (r1), a0l
cmp    0, a0
```

```
brr     10          ; saut zero1a si a0=0, cout = 0
cmp     3, a0
brr     9           ; saut onze1a si a0=3, cout = 2
add     #1,a1       ; dans tous les autres cas cout = 1
br      10          ; saut zero1a

; onze1a
add #2, a1          ; coût = 2

; zero1a
add r0,a1           ; ajoute le nouveau coût à l'ancien.
mov a1l, r0         ; sauvegarde le coût pour le premier segment
```

2- Le processeur ARM7.

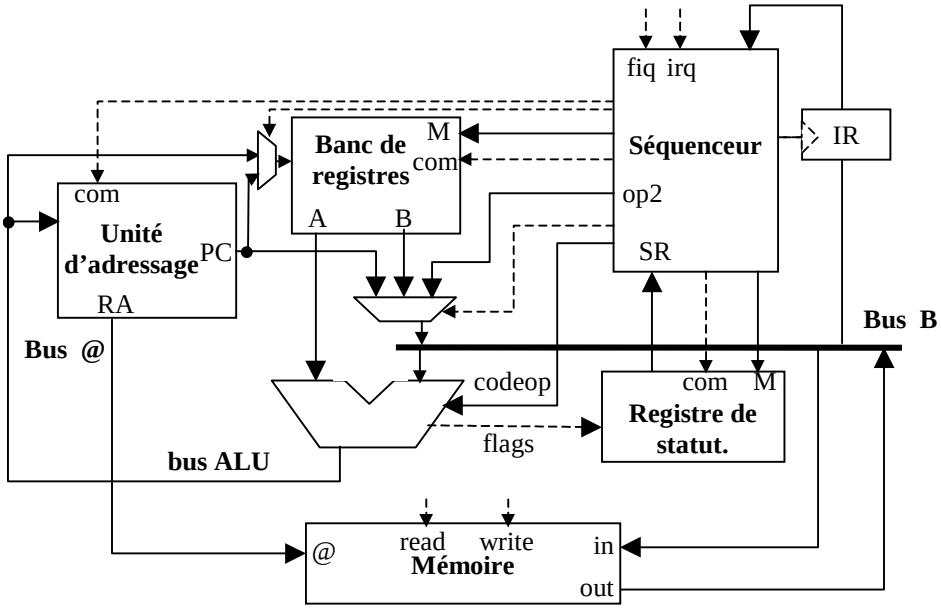


Figure 12 – Le modèle du ARM7.

Le cœur du ARM7 est composé d'une unité de calcul, d'un banc de registres, d'une unité d'adressage, d'un registre de statut et d'une unité de décodage (cf. Figure 12). La mémoire est une mémoire de données et d'instructions, c'est une mémoire de type RAM.

Dans le modèle que nous avons réalisé, l'unité de calcul est réduite à une ALU car les applications modélisées ne nécessitent ni multiplieur, ni unité de décalage. Leur modélisation n'aurait rien coûté puisque ces composants ont déjà été modélisés pour le OAK Plus.

2-1. Le séquenceur et la gestion des interruptions matérielles.

La différence essentielle entre ce processeur et le processeur maîtrise est qu'il prend en compte les interruptions matérielles (FIQ et IRQ). Lors d'une interruption matérielle, le contexte des registres généraux et du registre d'états est modifié de façon matérielle.

Le traitement d'une interruption est un mécanisme de préemption faible, c'est-à-dire qu'un signal d'interruption n'est pris en compte qu'au début du cycle suivant, les actions entreprises pendant le cycle sont terminées. De plus l'interruption IRQ est prioritaire par rapport à l'interruption FIQ.

Ce mécanisme d'interruptions est modélisé grâce à un composant appelé *InterruptManager*. L'instruction donnée au décodeur d'instructions dépend des signaux

d'interruption. Au début du cycle, lorsqu'une instruction est lue dans la mémoire, si aucun des signaux d'interruption n'est présent, l'instruction est donnée au décodeur d'instructions, si seul le signal FIQ est présent une instruction appelée FIQ est donnée au décodeur d'instructions, si le signal IRQ est présent une instruction appelée IRQ est donnée au décodeur d'instructions.

Les deux instructions IRQ et FIQ provoquent le changement de contexte des registres par l'émission d'un signal *M* vers le banc de registres et le registre de statut. De plus, une saut à une adresse configurable est effectué, la routine d'interruptions se trouvant à cette adresse sera alors exécutée par le mécanisme normal jusqu'à l'instruction RETI. En effet, l'instruction RETI qui signifie retour d'interruption provoque à nouveau le changement de contexte en rétablissant la valeur du signal *M*. Notons, que la routine d'interruption FIQ est interruptible par le même mécanisme lorsque le signal IRQ est présent. De plus, ce mécanisme peut être généralisé à un nombre quelconque de niveaux d'interruptions.

Voici le code Java du composant *InterruptManager* :

```
public class InterruptManager implements ServiceManager {
        public Value decode(Value normal, boolean [] interrupt, String []intInstructions, int M) {
                for (int i=0; i<M; i++)
                        if (interrupt[i])
                                return intInstructions[i];
                return normal;
        }
}
```

Interrupt Manager

Le composant *InterruptManager* est un composant séquentiel qui a un service *decode* déclenché par un changement de valeur sur le port *normal*.

Le paramètre *normal* est affecté au port *normal* qui reçoit l'instruction en provenance de la mémoire, c'est-à-dire l'instruction qu'il faut exécuter si aucun signal d'interruption n'est présent.

Le paramètre *interrupt* est associé à un port multiple qui reçoit les signaux d'interruptions. Le paramètre *intInterruptions* représente les signaux d'interruption. Chaque signal d'interruption est associé à une entrée qui reçoit l'instruction à exécuter lorsque le signal d'interruption associé est présent. Remarquons que le signal d'interruption est le signal le plus prioritaire. Dans notre modèle du ARM7, il sera associé au signal *IRQ*. L'entrée *intInterruptions[0]* est associée à l'instruction *"IRQ"*. L'entrée *interrupt[1]* est associée au signal *FIQ* et *intInterruptions[1]* est associée à l'instruction *"FIQ"*.

Le paramètre M est associé au port d'entrée M en provenance du décodeur d'instructions. Celui-ci porte la valeur du niveau d'interruption actuelle. Remarquons que si le niveau d'interruption est 1 (FIQ), seul le niveau 0 (IRQ) d'interruption peut être déclenché, si IRQ n'est pas présent, on exécutera les instructions en provenance de la mémoire (entrée *normal*), ces instructions correspondent à la routine d'interruption *FIQ*.

Le module *séquenceur* est alors principalement constitué du composant *InterruptManager* et d'un décodeur générique d'instructions. Le code SEP-ISDL utilisé pour modéliser le ARM7 est disponible en annexe.

2-2. Le banc de registres.

Le banc de registres contient quinze registres généraux numérotés de r0 à r14. Il a deux services de lecture vers les bus A et B, et un service d'écriture à partir du bus ALU.

Un tel banc de registre peut être modélisé de la façon présentée au chapitre III à partir du fournisseur de services *Memory*.

Cependant, le banc de registres du ARM7 a en plus neuf registres appelés fantômes (*banked registers*). Sept registres fantômes appelés (R8_FIQ … R14_FIQ) remplacent les registres (R8 … R14) lorsque le processeur est dans le mode d'interruption FIQ. Deux registres fantômes appelés R13_IRQ et R14_IRQ remplacent les registres R13 et R14 dans le mode d'interruption IRQ (cf. Figure 13).

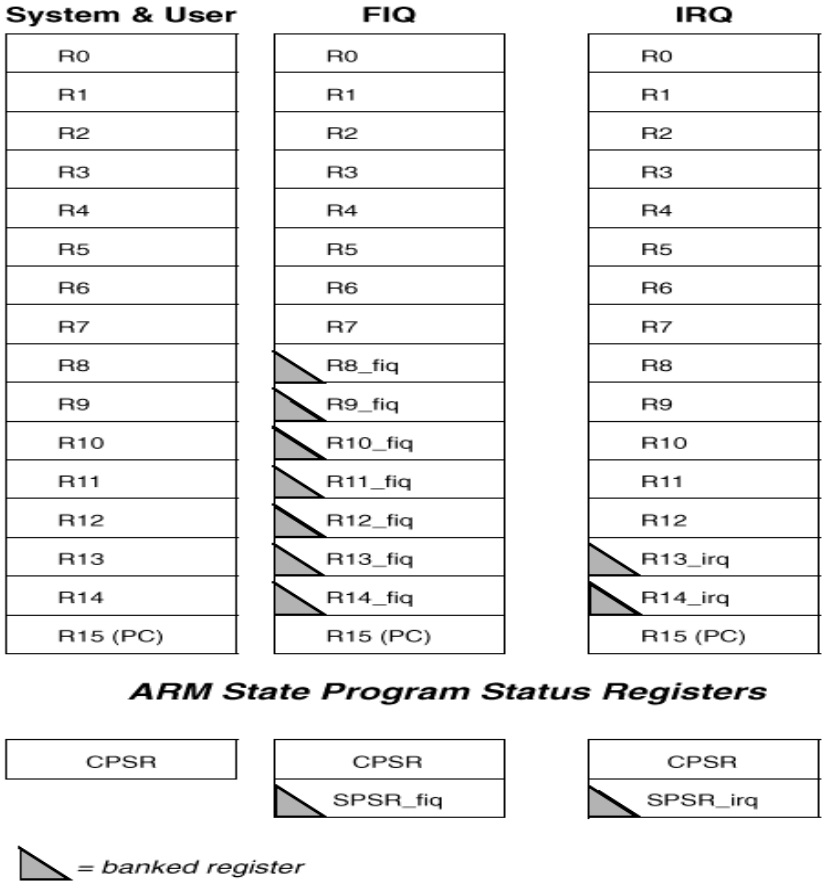


Figure 13 – Banc de registres et registres fantômes.

En fait, au niveau du modèle de programmation, le programmeur ne s'aperçoit pas de l'existence des registres fantômes, il ne manipule que des registres R0… R14. L'architecture fait la conversion à la volée suivant le mode d'interruption.

Le modèle que nous avons réalisé suit ce principe. Le banc de registres modélisé contient 24 (15+9) registres. Un composant appelé fantôme placé en entrée du banc de registres convertit en fonction du mode d'interruption les adresses 8…14 qui correspondent aux registres r8…r14 en 15…22 qui correspondent aux adresses des registres fantômes (r8_FIQ…r14_FIQ) lorsque le mode est FIQ. Une transformation analogue transforme les adresses 13 et 14 en 23 et 24 lorsque le mode d'interruption est IRQ.

Voici le code Java du fournisseur de service *Fantome* :

```
public class Fantome implements ServiceProvider {
        public int transforme (int adresse, int mode, int [] limite, int [] decalage) {
                if (adresse>=limite [mode])
                        return adresse+decalage[mode];
                return adresse;
        }
}
```

Fantome

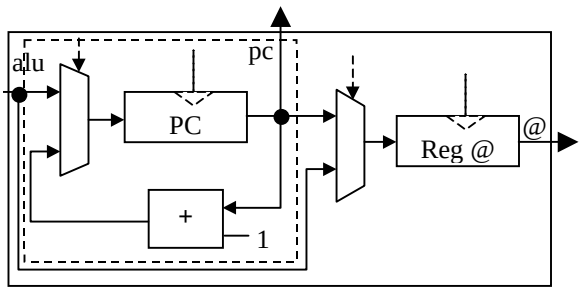
Ce composant *Fantome* est un composant combinatoire avec un service transforme.

Le paramètre *adresse* associé au port d'entrée *adresse* est un entier entre 0 et 14 qui fait référence au registre à adresser. Le paramètre *mode* associée au port *mode* est un entier qui représente le mode d'interruption (0 est le mode normal, 1 le mode FIQ, 2 le mode IRQ …). Le paramètre entier *limite* est associé au port multiple limite, ce port porte une valeur qui correspond à la valeur à partir de laquelle il faut décaler (FIQ : 8, IRQ : 13). Le paramètre entier *décalage* représente la valeur du décalage à effectuer en fonction du mode (FIQ : 7, IRQ : 9).

Le registre PC et les registres de configurations sont modélisés séparément (cf. Figure 12). Un dispositif analogue est mis en place pour les registres fantômes de statut.

2-3. L'unité d'adressage.

Cette unité est très simple, c'est le module décrit par la figure ci-contre. A partir de cette description les services de module *pc:=pc+1*, *pc:=alu*, *ra:=pc* et *ra:=alu* sont construits en sélectionnant les multiplexeurs adaptés et en activant les deux registres *PC* et *reg@*.



Cette unité n'est pas détaillée car tous les mécanismes nécessaires à sa construction ont déjà été expliqués et la construction des composants qui la constituent a été détaillée. De plus, la partie encadrée avec des pointillés correspond à l'unité PC définie pour modéliser le processeur PINE où l'entrée *v1* est fixée avec la constante 1 et l'entrée *v2* est connectée au bus *alu*.

La section 3- présente un exemple de code utilisé avec le modèle du processeur ARM7.

3- Le processeur OAK+.

Le processeur OAK+ ressemble beaucoup à son prédécesseur le PINE. Il y a principalement deux modifications qui nous concernent. L'unité de calcul a été sensiblement enrichie (cf. section 3-1) et le nombre de boucles matérielles imbriquées a été augmenté (cf. section 0). De plus, nous avons modélisé le mécanisme d'interruption (cf. section 3-3).

3-1. L'unité de calcul.

L'unité de calcul a été enrichie par l'ajout d'un bloc BMU (Bit Manipulation Unit) qui contient essentiellement un composant *BarrelShifter* et deux nouveaux accumulateurs b0 et b1. Ce bloc, positionné en parallèle du module de calcul du PINE permet d'effectuer des opérations de manipulations de bits. Cet ajout ne modifie par énormément la structure du processeur, mais de nouvelles instructions ont été ajoutées ainsi que des chemins de données. En effet, le module de calcul et le BMU communiquent notamment en échangeant les valeurs de leurs accumulateurs.

La Figure 14 présente la nouvelle structure de l'unité de calcul. Les modifications du code SEP-ISDL pour le décodeur sont présentées en annexe. Le composant CU PINE est le module unité de calcul du PINE que nous réutilisons pour le OAK+. Deux entrées (b0, b1) et deux sorties (a0, a1) lui ont été ajoutées et son module ALB a été modifié comme présenté par la

Figure 15. Les trois services selb0, selb1, selALU ont aussi été ajoutés, ils permettent de sélectionner correctement le multiplexeur.

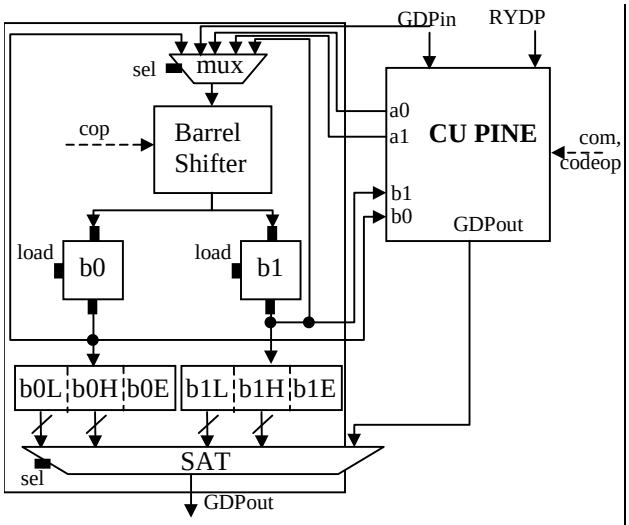


Figure 14 – Unité de calcul du OAK+.

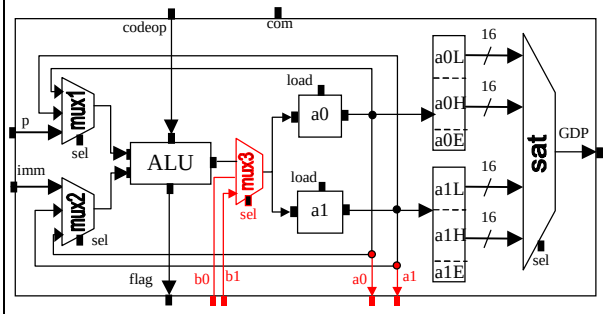


Figure 15 – Unité ALB du OAK+.

BarrelShifter

Le composant *BarrelShifter* définit huit services : *shl*, *shr*, *shl4*, *shr4*, *rol*, *ror*, *clr*, *id*. Ces services sont les services de manipulation de bits standards. Le code du fournisseur de services *BarrelShifter* est fourni en annexe. De plus les services *selA0*, *selA1*, *selB0*, *selB1*, *selImm*, *GDP:=b0H*, *GDP:=b0L*, *GDP:=b1H*, *GDP:=b1L* sont définis pour positionner l'unité *SAT*.

3-2. Les boucles matérielles.

Les boucles matérielles sont toujours gérées de la même façon, seulement il est désormais possible d'imbriquer quatre niveaux de boucles *bkrep*. Le composant ZCN a donc été ré-écrit en remplaçant les variables *lcHigh*, *last* et *first* par des tableaux. Le nombre d'imbrications est choisi par un paramètre *nestedLoop*.

Le code du nouveau composant ZCN est présenté en annexe. En fait, c'est ce composant qui est utilisé pour modéliser le PINE avec un paramètre *nestedLoop* réglé à 1.

3-3. Les interruptions.

Pour le OAK+, nous avons modélisé le mécanisme d'interruptions qui ne l'avait pas été pour le PINE. Le OAK+ possède deux interruptions matérielles INT2 et NMI, l'interruption NMI est prioritaire sur l'interruption INT2. Le mécanisme d'interruption étant analogue à celui du ARM7, il a été modélisé de la même manière avec un composant *InterruptManager* (cf. section 2-1).

4- L'interface de communication du 3771.

L'architecture bi-cœur que nous étudions (VVS3771) est composée d'un ARM7, d'un OAK+ et d'une interface de communication (cf. Figure 16). Cette interface de communication est essentiellement constituée d'une mémoire double port qui permet des accès simultanés par le OAK+ et par le ARM7, et une partie de contrôle qui contient plusieurs registres qui permettent de configurer la communication et l'envoi d'interruptions.

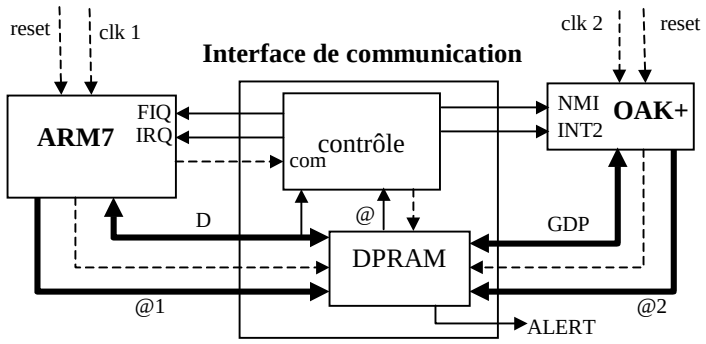


Figure 16 – Le BipCore.

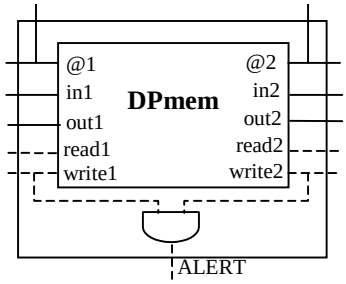


Figure 17 – Mémoire double-port (DPRAM).

4-1. La mémoire double port.

La mémoire double port (DPRAM) est en fait composée de deux mémoires de 16 bits. Ainsi, il est possible de lire ou d'écrire deux mots de 16 bits (32 bits) par cycle.

En fait, étant donné que le bus de données du OAK est un bus de 16bits, celui-ci ne peut transférer avec la DPRAM qu'un seul mot de 16 bits par cycle avec un des deux blocs. Le bit 0 de l'adresse est utilisé pour choisir entre les deux blocs. Vue du OAK, la DPRAM est simplement une mémoire de 16 bits.

Pour le ARM, la DPRAM peut être une mémoire de 8,16 ou 32 bits suivant la valeur indiquée par 2 bits appelés AABSIZE.

Pour implémenter cette mémoire dans SEP (cf. Figure 17), nous choisissons d'utiliser le fournisseur de services *Memory* présenté au chapitre X. Grâce au mécanisme présenté au chapitre III, nous créons à partir de ce fournisseur de services un composant séquentiel *DPmem* avec deux services de lecture (*read*) et deux services d'écriture (*write*). La lecture est prioritaire par rapport à l'écriture. Lors de deux écritures simultanées le contenu de la mémoire est indéterminée, c'est le comportement annoncé dans la documentation de la DPRAM. Cependant un mécanisme externe permet de détecter deux écritures simultanées afin de pouvoir analyser une application.

4-2. Le composant contrôle.

En fait, il est très rare que les deux cœurs fonctionnent indépendamment. Le rôle d'une telle architecture est d'utiliser le processeur RISC pour contrôler une application et alimenter en données le processeur DSP. Ce dernier implémente plus efficacement les algorithmes de traitement du signal. Une fois les calculs effectués, les données sont rendues au processeur RISC qui les interprète.

Le mécanisme de communication qui nous intéresse fonctionne par interruptions. Le ARM7 configure l'interface de communication afin qu'elle émette une interruption vers le OAK+ lorsque toutes les données à transmettre ont été transférés vers la DPRAM. De même, le ARM7 peut aussi configurer l'interface pour qu'elle lui envoie une interruption lorsque le OAK+ a terminé de charger ses résultats dans la DPRAM.

Cette configuration est faite par l'intermédiaire de registres. Le ARM7 peut mettre une adresse dans les registres (O2AIP1, O2AIP2, A2OIP1, A2OIP2). Une interruption est envoyée vers les ARM lorsque les données aux adresses O2AIP1 ou O2AIP2 sont accédées, ou vers le OAK lorsque les données aux adresses A2OIP1 ou A2OIP2 sont accédées.

De plus, le registre ISCR0 permet de choisir le type d'interruptions (FIQ ou IRQ) et la condition (lecture ou écriture) sous laquelle une interruption est envoyée vers le ARM. De même, le registre ISCR1 permet de choisir le type d'interruptions (INT2 ou NMI) et la condition sous laquelle une interruption est envoyée vers le OAK+ :

ISCR0	AIS1	AIE1	AEW1	AER1	AIE2	AIS2	AEW2	AER2
ISCR1	OIS1	OIE1	OEW1	OER1	OIE2	OIS2	OEW2	OER2

Le A ou le O précisent si l'interruption doit être envoyée vers le ARM7 ou vers le OAK+. Le 1 ou le 2 précisent si la configuration concerne le premier registre (O2AIP1, A2OIP1) ou le deuxième (O2AIP2, A2OIP2). Le bit xERy configure l'interruption lors d'une lecture, le bit xEWy configure l'interruption lors d'une écriture. Le bit xIEy active ou désactive le mécanisme d'interruption. Enfin le bit xISy choisi le type d'interruption (IRQ ou FIQ, INT2 ou NMI).

banc de registres

Le banc de registres (cf. Figure 18) est un module qui contient six registres, une entrée *in* et une entrée de commande *com* relative à des services de modules. Les services de modules (*storeO2AIP1*, …, *storeISCR2*) permettent de sélectionner le registre dont on veut modifier la valeur. Tous les registres sont accessibles directement en lecture.

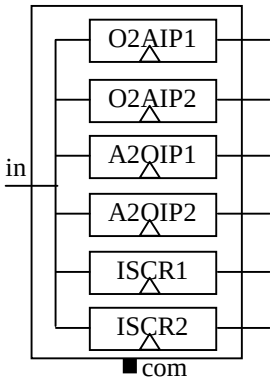


Figure 18 – Banc de registres.

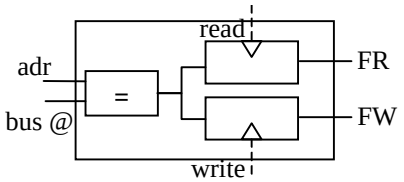


Figure 19 – Le composant compare.

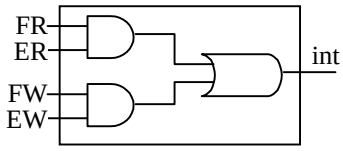


Figure 20 – Le composant Logique.

compare

Un module *compare* (cf. Figure 19) permet de vérifier qu'un accès en lecture (FR) ou en écriture (FW) est fait à une adresse donnée (*adr*). Pour cela, il utilise la donnée présente sur le bus d'adresses ainsi que les signaux de lecture (*read*) et d'écriture (*write*) sur la DPRAM.

Logique

Enfin un module *Logique* (cf. Figure 20) permet de déterminer si une interruption doit ou non être émise. Il utilise les signaux FR et FW fournis par le composant *compare* et les bits de configuration (ER, EW) des registres ISCR0 et ISCR1. En fait, pour optimiser les composants constitués d'expressions logiques, un composant logique a été créé, ce composant peut être configuré avec une expression logique, ici l'expression serait : *int = (FR et ER) ou (FW et EW)*.

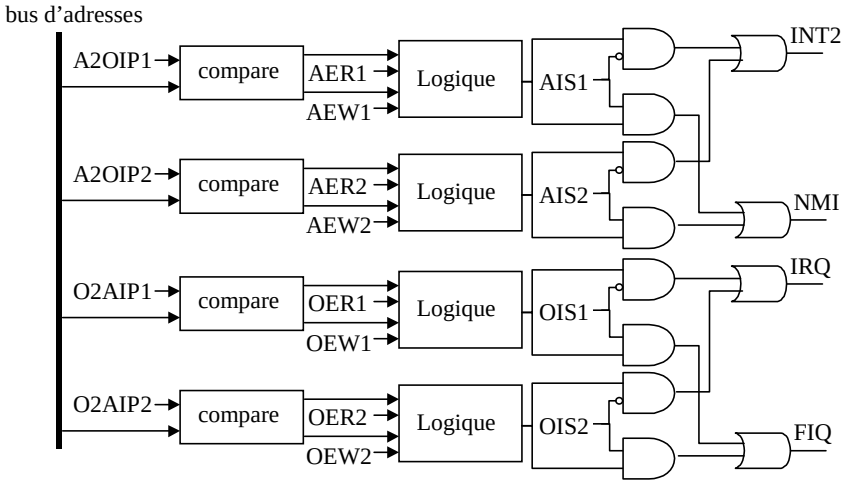


Figure 21 – Le composant contrôle.

Avec ces trois composants, nous pouvons maintenant constituer un module pour modéliser le composant de contrôle (cf. Figure 21).

4-3. L'application.

L'application que nous avons modélisée permet de valider le mécanisme de communication par interruptions (cf. Figure 22).

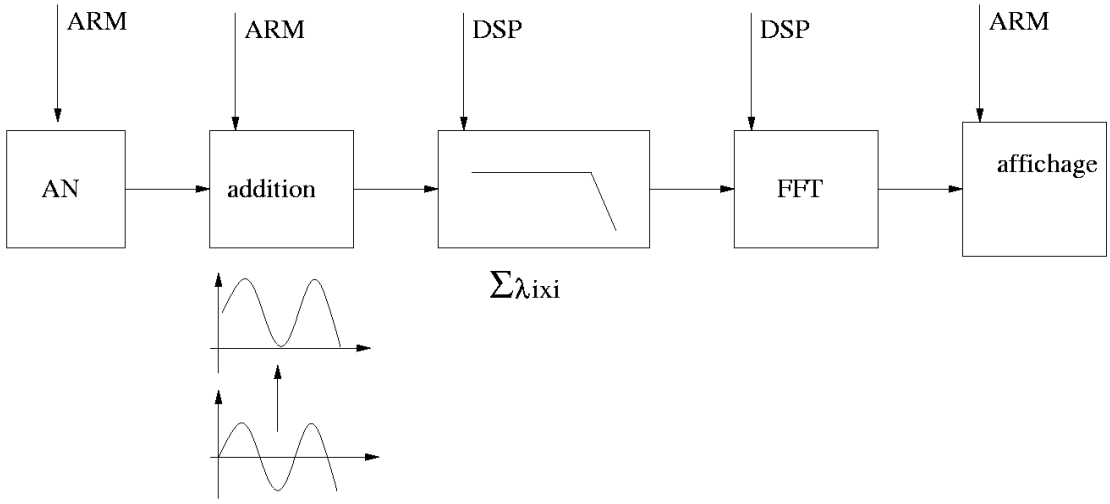


Figure 22 – L'application.

Ainsi à l'origine, la mémoire du ARM contient des données à traiter, un programme *shift* exécuté lors du reset du processeur, ainsi qu'un programme *download* positionné à l'adresse de l'interruption FIQ.

Le programme *shift*, lit des données réelles en mémoire, fait un décalage afin de les rendre toutes positives, les données décalées sont écrites dans la DPRAM. De plus, l'interface de communication est configurée pour qu'à la fin du téléchargement des données l'interruption INT2 du OAK+ soit déclenchée.

Le programme *download* télécharge des données de la DPRAM vers la mémoire du ARM7. Ce programme sera exécuté lors que l'interruption FIQ sera levée. L'interface de communication est configurée pour que cette interruption soit levée lorsque les données résultats seront écrites dans la DPRAM par le OAK+.

La mémoire de programme du OAK+ contient à l'adresse de l'interruption INT2, un programme qui permet de télécharger les données de la DPRAM vers la mémoire X du OAK+. Puis un filtre FIR est exécuté avec les coefficients contenus dans la mémoire Y. Puis, une FFT est exécutée avec les coefficients W aussi présents dans la mémoire Y. Enfin, le résultat est téléchargé dans la DPRAM à une adresse prédéfinie.

Le code assembleur des programmes modélisés est disponible en annexe.

En conclusion, les processeurs OAK+ et ARM7 ont été modélisés avec SEP. Les applications classiques utilisées pour valider leur modèle ont donné les résultats attendus relatifs aux nombres de cycles, c'est-à-dire le même résultat que celui qui est obtenu avec les processeurs réels.

De plus, le modèle de l'interface de communication et le mode de communication par interruptions ont été validés de deux façons différentes. Premièrement, une configuration constituée où deux ARM7 communiquent par l'interface de communication. Deuxièmement une application classique de contrôle d'un processeur DSP par un processeur RISC a aussi été simulée avec succès.