ARITHMETIQUE & LOGIQUE DES RETINES PROGRAMMABLES

Jeu d'instructions de Pvlsar 34

Antoine MANZANERA – Février 2006

Résumé : Nous décrivons dans ce rapport l'implantation des primitives de calcul, logiques et arithmétiques, dans les rétines programmables booléennes. A partir du jeu d'instruction de Pvlsar 34, on dérive les opérations booléennes élémentaires. Puis, les primitives numériques de base sont présentées en détail.

I Algorithmique des rétines programmables

La rétine numérique Pvlsar 34 est une *machine massivement parallèle* de 40 000 processeurs *interconnectés selon une grille 2d* 200x200 en *topologie 4-connexe*.

Le mode de parallélisme est purement *SIMD* (Single Instruction Multiple Data) : toute la grille est *commandée par un séquenceur externe* à la rétine qui envoie une séquence d'instructions à la rétine selon une fréquence fixe, et à chaque pas de calcul, tous les 40 000 processeurs exécutent exactement la même instruction. Un programme rétinien est donc entièrement défini par la séquence d'instructions envoyée par le séquenceur.

Chaque processeur est doté d'une *mémoire numérique* dont la vocation est de représenter les *données d'un pixel* : sous-ensemble minimal du contenu d'une image captée, ou du résultat d'un traitement sur cette image.

L'entrée des données se fait de manière *optique* : chaque processeur est pourvu d'un *photocapteur* et d'un mécanisme de *conversion analogique-numérique* qui lui permet de coder dans sa mémoire une grandeur numérique représentant une intensité lumineuse.

Chaque processeur est doté d'une *unité de calcul* permettant de *lire, combiner de manière logique et écrire* des données numériques à partir de et vers sa mémoire.

Chaque couple de processeurs adjacents au sens de la 4-connexité *partage* une partie de leur mémoire, ce qui permet de communiquer *des données entre pixels voisins*.

Du point de vue des accès à la mémoire numérique de la rétine, il est important de comprendre qu'*il n'existe pas de moyen de lire ou d'écrire depuis l'extérieur directement* un pixel ou une zone spécifiée de la mémoire de la rétine. Le seul moyen d'accéder à la mémoire numérique de la rétine, en lecture ou en écriture, est *par le bord de la grille*, où le contenu de la mémoire d'une partie des processeurs est accessible par le bus de communication avec le processeur hôte. Par suite, l'accès à un pixel ou une zone quelconque de la mémoire numérique se fait par *combinaison de décalages* effectués grâce aux partage de mémoire entre processeurs adjacents.

En plus de la sortie numérique séquentielle, Pvlsar 34 possède une *primitive d'extraction globale*, parallèle et analogique sous la forme d'une mesure de courant qui fournit un indicateur de la somme sur tous les processeurs de la grille, des valeurs correspondant au contenu d'un registre binaire spécifié. Cet indicateur ne fournit qu'une approximation de la somme, avec une incertitude augmentant avec le nombre de valeurs binaires à 1 sur la grille dans le registre spécifié. Cette primitive peut cependant être utilisée avec profit pour estimer des grandeurs statistiques spatiales sur la rétine sans avoir à extraire séquentiellement les données numériques concernées, donc en temps constant.

Au-delà du parallélisme massif et du mode de communication avec l'extérieur que nous venons d'évoquer, la principale incidence de l'architecture de la rétine numérique sur l'algorithmique réside dans le caractère extrêmement rudimentaire des processeurs élémentaires. Chaque processeur étant associé à un élément photosensible du capteur, la surface qu'il occupe sur le circuit est très réduite. Par conséquent, la *mémoire* dédiée à chaque processeur est *très faible* et son *jeu d'instructions très réduit*. Pvlsar 34 dispose de 48 bits (organisés en un tableau de 6 colonnes d'un octet chacune) de mémoire par pixel dont 4 bits sont en fait partagés par les 4 pixels voisins, et un jeu d'instruction limité au calcul du OU logique entre 2 bits et à la négation d'1 bit.

La (faible) mémoire disponible par pixel constitue l'intégralité de la mémoire utilisable par le programmeur pour les algorithmes exécutés sur la rétine. La conséquence majeure est que, contrairement à un système classique où tous les niveaux courants de mémoire sont utilisés de manière plus ou moins transparente par l'utilisateur, celui-ci doit sur la rétine *veiller à la répartition de sa mémoire entre les différents niveaux* : (1) le niveau « RAM » correspondant aux données, par ex. : images incidentes, descripteurs statistiques mis à jour régulièrement, (2) le niveau « Cache » correspondant aux opérandes d'une fonction complexe, par ex. : les bits homologues et la retenue courante dans le calcul d'une somme arithmétique bit à bit, et (3) le niveau « Registre », correspondant aux bits de calcul utilisés par une fonction complexe, par ex. le registre intermédiaire utilisé pour stocker NON(A ET B) dans le calcul de A XOR B.

Une conséquence secondaire est la nécessité d'une *gestion extrêmement rigoureuse de la dynamique* des données manipulées lors d'opérations arithmétiques, pour contrôler la précision des calculs.

D'autres contraintes plus particulièrement spécifiques à l'architecture de Pvlsar 34 ont une influence non négligeable sur la manière de programmer :

- l'existence des registres partagés, qui implique la spécialisation de certains registres binaires aux opérations de communications entre pixels voisins.
- La contrainte selon laquelle les 2 opérandes d'une opération booléenne doivent nécessairement se trouver sur la même ligne du tableau des registres, qui implique des déplacements de données au sein de la mémoire du pixel pour chaque opération.

Une façon raisonnable d'aborder ce problème pour faciliter le développement de programmes rétiniens, sans doute au détriment de l'optimalité des algorithmes, est pour le programmeur de segmenter mentalement la mémoire de chaque pixel en répartissant a priori les registres par fonction : RAM, Cache, Registres, Communication.

II Pvlsar 34 et son jeu d'instructions

Les règles du jeu d'instruction de Pvlsar 34 sont les suivantes :

- les opérandes sont les registres binaires formant la mémoire de chaque pixel. Ils sont organisés en grille de 8 lignes par 6 colonnes, et on les désigne par leurs cordonnées : (x_i,y_j) désigne le bit qui se trouve à la $i^{ème}$ colonne et à la $j^{ème}$ ligne (voir Figure 1(a)).
- L'unité booléenne (BU) peut lire, écrire, et effectuer des calculs sur ces registres binaires.
- Certains registres sont partagés par des pixels voisins, de manière à permettre la communication de données nécessaires aux traitements spatiaux (voir Figure 1(b)).

Les instructions rétine que nous utilisons dans les algorithmes sont les suivantes :

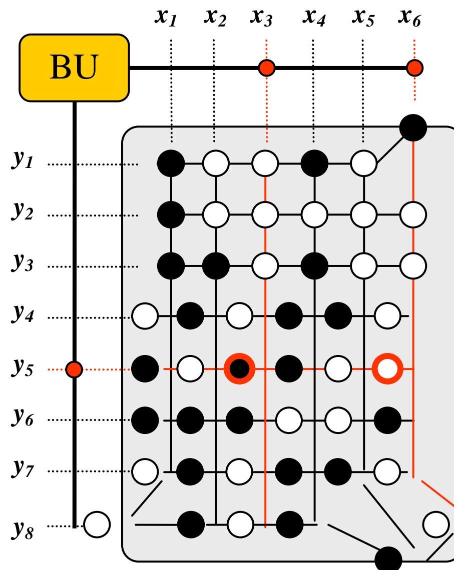
- **rd yj xi** : *lecture* par la BU de la valeur du registre binaire (x_i,y_j). Cette valeur est attribué au *palpeur*, registre binaire volatile de la BU, et supprimé du registre (x_i,y_j), dont la valeur se retrouve donc indéterminée.

- **wd yj xi** : *écriture* par la BU dans le registre (x_i,y_j), de la valeur binaire de son palpeur.

- **wc yj xi** : *écriture* par la BU dans le registre (x_i,y_j), du *complémentaire* de la valeur binaire de son palpeur.

- **ror yj xi etxk** : calcul par la BU du OU logique entre les valeurs des registres binaires (x_i,y_j) et (x_k,y_j). Cette valeur est attribué au *palpeur*, et les valeurs des registres (x_i,y_j) et (x_k,y_j) se retrouvent indéterminées.

- **one** : attribution de la valeur *1 logique* (tautologie) au *palpeur*.



(a)

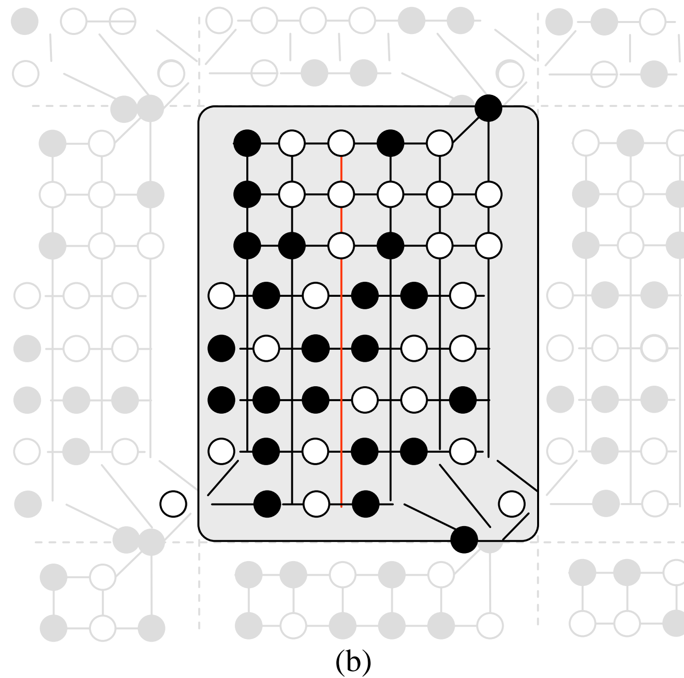


Figure 1 : Un pixel de la rétine Pvlsar 34. (a) Organisation de la mémoire et principe de calcul de l'Unité Booléenne (BU) (b) Topologie d'interconnexion par partage de registres

III Opérations booléennes

La Figure 2 montre des exemples de procédure de calcul d'instruction « élémentaire » à partir de ce jeu d'instructions. Dans ces exemples, on fait subir un certain nombre d'opérations à 2 bits A et B, arbitrairement placés, pour fixer les idées, dans les registres (x_3,y_2) et (x_5,y_6), respectivement, et on place le résultat C du calcul dans le registre (x_2,y_4). D'autre part, la plupart de ces procédures utilisent des registres de la ligne y7 comme variables intermédiaires ; on suppose donc que le contenu de ces registres peut être écrasé.

Calcul de C = OR(A,B)	**Calcul de C = AND(A,B)**

rd y2 x3 ; // lecture de A
wd y2 x3 ; // rafraîchissement du registre
wd y7 x1 ; // écriture de A
rd y6 x5 ; // lecture de B
wd y6 x5 ; // rafraîchissement du registre
wd y7 x2 ; // écriture de B
	rd y2 x3 ; // lecture de A
	wd y2 x3 ; // rafraîchissement du registre
	wc y7 x1 ; // écriture complémentée de A
	rd y6 x5 ; // lecture de B
	wd y6 x5 ; // rafraîchissement du registre
	wc y7 x2 ; // écriture complémentée de B

<table>
<tr><td>ror y7 x1 etx2 ; // calcul de A ou B
wd y4 x2 ; // écriture directe</td><td>ror y7 x1 etx2 ; // calcul de ¬A ou ¬B
wc y4 x2 ; // écriture complémentée</td></tr>
<tr><td>Calcul de C = AND(A,NOT(B))
rd y2 x3 ; // lecture de A
wd y2 x3 ; // rafraîchissement du registre
wc y7 x1 ; // écriture complémentée de A
rd y6 x5 ; // lecture de B
wd y6 x5 ; // rafraîchissement du registre
wd y7 x2 ; // écriture de B
ror y7 x1 etx2 ; // calcul de ¬A ou B
wc y4 x2 ; // écriture complémentée</td><td rowspan="2">Calcul de C = XOR(A,B)
rd y2 x3 ; // lecture de A
wd y2 x3 ; // rafraîchissement du registre
wd y7 x1 ; // écriture de A
wc y7 x3 ; // écriture complémentée de A
rd y6 x5 ; // lecture de B
wd y6 x5 ; // rafraîchissement du registre
wc y7 x2 ; // écriture complémentée de B
wd y7 x4 ; // écriture de B
ror y7 x1 etx2 ; // calcul de A ou ¬B
wc y7 x1 ; // écriture complémentée
ror y7 x3 etx4 ; // calcul de ¬A ou B
wc y7 x2 ; // écriture complémentée
ror y7 x1 etx2 ; // calcul du XOR
wd y4 x2 ; // écriture directe</td></tr>
<tr><td>Calcul de C = A.Nord
rd y2 x3 ; // lecture de A
wd y2 x3 ; // rafraîchissement du registre
wd y8 x5 ; // écriture au Sud
rd y1 x6 ; // lecture au Nord
wd y4 x2 ; // écriture directe</td></tr>
</table>

Figure 2 : Procédures élémentaires correspondant aux instructions du pseudo-code.

Bien entendu, chaque variable peut être indifféremment attribué à un autre registre. Seules les registres (x_8,y_1), (x_1,y_8), (x_5,y_8) et (x_6,y_8) conservent un usage particulier lié aux communications de données.

IV Acquisition et codage

V Addition, soustraction, valeur absolue

V.1 Procédures en pseudo-code booléen

Nous présentons dans un cette partie les premières primitives arithmétiques Ces primitives sont explicitées à l'aide des opérateurs booléens binaires ET (conjonction logique notée $\wedge$), OU (disjonction logique notée $\vee$), XOR (ou exclusif noté $\oplus$), et de l'opérateur booléen unaire NON (négation logique notée $\neg$).

La Figure 3 représente les opérations d'addition et de soustraction de deux nombres X et Y codés sur n bits $X = \{X_i\}_{i=0..n-1}$ et $Y = \{Y_i\}_{i=0..n-1}$. Le résultat est $Z = \{Z_i\}_{i=0..n-1}$. Ces opérateurs utilisent au plus 3 bits supplémentaires pour le calcul (variables A, B, et S). Telles quelles, ces procédures peuvent être utilisés pour deux nombres X et Y positifs, donc sans bit de signe. Dans ce cas, l'addition peut générer un débordement de capacité, qui correspond au cas où la dernière retenue sortante S est égale à 1. Dans le cas de la soustraction, si Y est plus grand que X, alors le résultat Z est négatif : S représente alors le bit de signe de Z, qui est codé en complément à 2 : les bits $\{Z_i\}_{i=0..n-1}$ correspondent au code naturel de Z lorsque S vaut 0 (Z est positif), et au code naturel de 2^n+Z lorsque S vaut 1 (Z est négatif).
Lorsque l'opérande de gauche Y est un nombre signé, il suffit de remplacer pour l'addition (resp. pour la soustraction) l'initialisation de S à 0 (resp. à 1) par l'initialisation de S au bit de signe (resp. au complémentaire du bit de signe) de Y. Dans ce cas, le débordement éventuel de capacité correspond au cas où X et Y ont le même signe, et la dernière retenue sortante indique un signe différent. Si l'on note S_X, S_Y et S, respectivement le bit de signe de X, celui de Y, et la dernière retenue sortante de l'opération, le débordement de capacité D correspond à l'expression logique:

$$D = (\neg S \wedge S_X \wedge S_y) \vee (S \wedge \neg S_X \wedge \neg S_y)$$

$S = 0$ For $i = 0$ to $n-1$ | $A = X_i \oplus Y_i$ | $B = X_i \wedge Y_i$ | $Z_i = A \oplus S$ | $A = A \wedge S$ | $S = A \vee B$	$S = 1$ For $i = 0$ to $n-1$ | $A = X_i \oplus \neg Y_i$ | $B = X_i \wedge \neg Y_i$ | $Z_i = A \oplus S$ | $A = A \wedge S$ | $S = A \vee B$	$B = S \wedge X_0$ For $i = 1$ to $n-1$ | $X_i = X_i \oplus B$ | $A = S \wedge X_i$ | $B = B \vee A$
Addition	***Soustraction***	***Valeur absolue***

Figure 3 : Addition et soustraction de deux nombres – Valeur absolue

Ces opérateurs étant principalement basés sur le full-adder logiciel (voir Figure 4), leur complexité asymptotique est de *5n*. Le coût en mémoire maximum est de *3n + 3*, dont *3n* bits pour les données et *3* bits pour le calcul. D'autre part, si on peut « écraser » l'un des opérandes X ou Y, le coût en mémoire devient *2n + 3*.

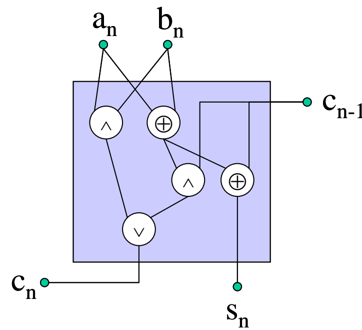


Figure 4 : Le full-adder logiciel

La Figure 4 représente également le calcul de la valeur absolue d'un nombre $X = \{X_i\}_{i=0..n-1}$. Ici le calcul se fait de manière destructive. Le coût en mémoire est donc de *n+3* et le coût en temps de calcul de *3n*.

La Figure 5 présente l'algorithme de multiplication de deux nombres *A* et *B* codés sur n bits dans la rétine. Le résultat *S* étant codé sur *2n* bits, et le coût en mémoire de calcul étant le même que pour l'addition, le coût en mémoire est donc de *4n+3* bits, dont *4n* pour les données, et *3n+3* bits si l'un des opérandes peut être écrasé. Le coût asymptotique en temps de calcul est $6n^2$.

$s = a \cdot b_0$ ***(n op.)***
for i = 1 to n-1
$\left\{ \begin{array}{l} s >> 1 \\ s = s + a \cdot b_i \end{array} \right.$ ***(n+5n op.)***

Figure 5 : Multiplication par addition et décalage

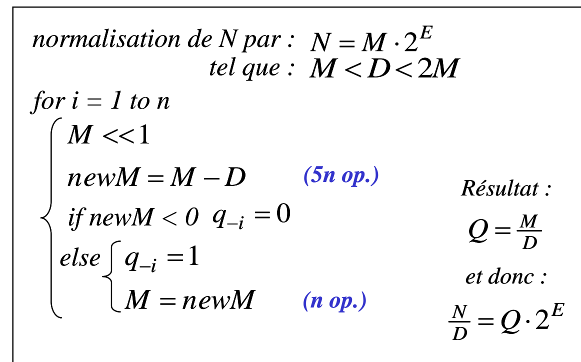


Figure 6 : Division restaurante

La division de deux nombres codés sur la rétine est l'opération arithmétique binaire la plus complexe. La Figure 6 présente l'algorithme de division restaurante, dont le coût asymptotique en nombre d'opérations est de $6n^2$, et le coût en mémoire est de $6n + Log(E_{max}) + 3$, où E_{max} est la valeur du plus grand exposant dans la normalisation en (mantisse, exposant) des numérateurs. Notons que la complexité de l'algorithme est encore augmentée par la phase de normalisation.

Pour les opérateurs arithmétiques, il convient de noter que les algorithmes peuvent se simplifier considérablement lorsqu'un des deux opérandes est une constante, le cas extrême étant celui de la multiplication ou division par une constante puissance de deux, qui se calcule alors par un simple décalage à gauche ou à droite.

V.2 Procédures PVLSAR 34

Nous détaillons dans cette section le code des procédures arithmétiques de base (addition, soustraction, valeur absolue), à partir du jeu d'instruction de PVLSAR 34.

(a) Affectation des registres

Nous commençons par attribuer les registres disponibles de la rétine aux données qui vont être manipulées. Par exemple, si l'on souhaite effectuer les opérations arithmétiques sur 2 images A et B codées sur 6 bits, en supposant qu'on place l'image A sur la ligne 3, et l'image B sur la ligne 5 :

```
// Image A
#define A_5 y3 x1
#define A_4 y3 x2
#define A_3 y3 x3
#define A_2 y3 x4
#define A_1 y3 x5
```

```
#define A_0 y3 x6
// Image B
#define B_5 y5 x1
#define B_4 y5 x2
#define B_3 y5 x3
#define B_2 y5 x4
#define B_1 y5 x5
#define B_0 y5 x6
```

Une ligne est réservée pour les calculs, par exemple la ligne 2. Notez la définition des symboles etCal_x utilisés pour effectuer les calculs. Ne pas oublier que lorsqu'on effectue un calcul, les 2 registres opérandes doivent se trouver sur la même ligne.

```
// Registres de calcul (dans les procédures)
// Attention tous les registres Cal_i
// doivent être sur la même ligne
#define Cal_1 y2 x1
#define Cal_2 y2 x2
#define etCal_2 etx2
#define Cal_3 y2 x3
#define etCal_3 etx3
#define Cal_4 y2 x4
#define etCal_4 etx4
#define Cal_5 y2 x5
#define etCal_5 etx5
#define Cal_6 y2 x6
#define etCal_6 etx6
```

Enfin, on réserve quelques registres pour des variables temporaires (bits de signe, retenues, bascules diverses,…) :.

```
// Registres Temporaires
#define Tmp_1 y6 x1
#define Tmp_2 y6 x2
#define Tmp_3 y6 x3
```

(b) Ecriture des procédures bit-wise

On écrit des procédures bit-wise (bit à bit) correspondant à la transcription en assembleur Pvlsar 34 du pseudo-code correspondant à une itération de la boucle qui décrit les bits selon leur poids.

Les 3 tableaux suivants présentent les codes correspondant respectivement à l'addition, la soustraction et la valeur absolue. Notons que ces procédures, par convention, n'utilisent – en dehors des variables passées en paramètres – que des registres de la ligne de calcul Cal_x.
L'objectif est de rendre l'allocation de registre la plus simple pour le programmeur.

```
//----------------------------------------------------------------
//
// Procédure bit-wise d'addition entre le bit A_i (X)
// et le bit B_i (Y) avec retenue entrante C_in (Carry)
// en sortie : X vaut le bit de somme S_i ; Y vaut B_i
// Carry vaut la retenue sortante C_out
//
//----------------------------------------------------------------
void Addition_Bit (unsigned short X,unsigned short Y,unsigned short Carry) {

//Copie (destructive) de A_i en Cal_1 et non(A_i) en Cal_2 et Cal_3
rd X;
wd Cal_1;
wc Cal_2;
wc Cal_3;
//Copie (non destructive) de B_i en Cal_4 et et non(B_i) en Cal_5 et Cal_6
rd Y;
wd Y;
wd Cal_4;
wc Cal_5;
wc Cal_6;
//Calcul de non(A_i ou B_i) en Cal_1
ror Cal_1 etCal_5;
wc Cal_1;
//Calcul de non(non(A_i) ou B_i) en Cal_2
ror Cal_2 etCal_4;
wc Cal_2;
//Calcul de A=A_i xor B_i en Cal_1 et non(A) en Cal_2 et Cal_4
ror Cal_1 etCal_2;
wd Cal_1;
wc Cal_2;
wc Cal_4;
//Calcul de B=A_i et B_i=non(non(A_i) ou B_i) en Cal_3
ror Cal_3 etCal_6;
wc Cal_3;
//Copie de C_in en Cal_5 et non(C_in) en Cal_6
rd Carry;
wd Carry;
wd Cal_5;
wc Cal_6;
//Calcul de non(C_in ou non(A)) en Cal_2
ror Cal_2 etCal_5;
wc Cal_2;
//Calcul de non(non(C_in) ou A)) en Cal_1
ror Cal_1 etCal_6;
wc Cal_1;
//Mise a jour de A_i en IO_2
ror Cal_1 etCal_2;
wd X;
```

```
//Copie de non(C_in) en Cal_1
rd Carry;
wc Cal_1;
//Calcul de A=A et C=non(non(A) ou non(C_in))
ror Cal_1 etCal_4;
wc Cal_1;
//Mise a jour de C_out=B ou C_in
ror Cal_1 etCal_3;
wd Carry;

}//------------------------------------------------------------
//
// Procédure bit-wise de soustraction entre le bit A_i (X)
// et le bit B_i (Y) avec retenue entrante C_in (Carry)
// en sortie : X vaut le bit de somme S_i ; Y vaut B_i
// Carry vaut la retenue sortante C_out
//
//------------------------------------------------------------
void Soustraction_Bit (unsigned short X,unsigned short Y,unsigned short Carry) {

//Copie (destructive) de A_i en Cal_1 et non(A_i) en Cal_2 et Cal_3
rd X;
wd Cal_1;
wc Cal_2;
wc Cal_3;
//Copie (non destructive) de B_i en Cal_4 et Cal_5 et non(B_i) en Cal_6
rd Y;
wd Y;
wd Cal_4;
wd Cal_5;
wc Cal_6;
//Calcul de non(A_i ou B_i) en Cal_1
ror Cal_1 etCal_4;
wc Cal_1;
//Calcul de non(non(A_i) ou non(B_i)) en Cal_2
ror Cal_2 etCal_6;
wc Cal_2;
//Calcul de A=A_i xnor B_i en Cal_1 et non(A) en Cal_2 et Cal_4
ror Cal_1 etCal_2;
wd Cal_1;
wc Cal_2;
wc Cal_4;
//Calcul de B=A_i et non(B_i)=non(non(A_i) ou B_i) en Cal_3
ror Cal_3 etCal_5;
wc Cal_3;
//Copie de C_in en Cal_5 et non(C_in) en Cal_6
rd Carry;
wd Carry;
wd Cal_5;
wc Cal_6;
//Calcul de non(C_in ou non(A)) en Cal_2
ror Cal_2 etCal_5;
wc Cal_2;
//Calcul de non(non(C_in) ou A)) en Cal_1
```

```
ror Cal_1 etCal_6;
wc Cal_1;
//Mise a jour de S_i=A xor C_in en IO_2
ror Cal_1 etCal_2;
wd X;
//Copie de non(C_in) en Cal_1
rd Carry;
wc Cal_1;
//Calcul de A=C_in et A=non(non(C_in) ou non(A))
ror Cal_1 etCal_4;
wc Cal_1;
//Mise a jour de C_out=A ou B
ror Cal_1 etCal_3;
wd Carry;

}
```

```
//------------------------------------------------------------
// Procedure bit-wise de calcul de la valeur absolue
// du bit X_i (X), avec la bascule B (Flag) et le bit de
// signe S (Sign), En sortie X vaut le bit du nombre en valeur
// absolue |X_i|, Sign est inchangé, et Flag est mis à jour
//------------------------------------------------------------

void ValeurAbsolue_Bit (unsigned short X,unsigned short Sign,unsigned short Flag) {

// Copie de B sur Cal_3 et Cal_4 et non(B) sur Cal_5
rd Flag;
wd Cal_3;
wd Cal_4;
wc Cal_5;
// Copie de X_i sur Cal_1 et non(X_i) sur Cal_2
rd X;
wd Cal_1;
wc Cal_2;
// Calcul de A = B et non(X_i) sur Cal_1
ror Cal_1 etCal_5;
wc Cal_1;
// Calcul de C = non(B) et X_i sur Cal_2
ror Cal_2 etCal_3;
wc Cal_2;
// Calcul de X_i = A ou C (B xor X_i)
// mise à jour et copie complémentée sur Cal_2
ror Cal_1 etCal_2;
wd X;
wc Cal_2;
// copie (et sauvegarde) de S sur Cal_3;
rd Sign;
wd Sign;
wd Cal_3;
// Calcul de A = non(S) et X_i sur Cal_2
ror Cal_2 etCal_3;
wc Cal_2;
```

```
// Calcul de B = A ou B
rmt1 Cal_2 etCal_4;;
wd Flag;
}
```

(c) Calcul numérique

Une fois les routines arithmétiques bit-wise écrites, le calcul des opérateurs numériques s'effectue au sein du programme par une séquence - selon les bits de différents poids - de cycles : (1) lecture du bit (2) Calcul de la procédure, et (3) mise à jour du bit. Voici à titre d'exemple le calcul de la différence entre les 2 nombres *A* et *B* définies plus haut (Notons que conformément à la convention adoptée plus haut, la valeur initiale de *A* est écrasée et remplacée par $A+B$; la valeur de *B* est conservée :

```
// Initialisation à 1 de la retenue entrante
one;
wd Tmp_1;
// Calcul du bit de poids faible au bit de poids fort
Soustraction_bit(A_0,B_0,Tmp_1);
Soustraction_bit(A_1,B_1,Tmp_1);
Soustraction_bit(A_2,B_2,Tmp_1);
Soustraction_bit(A_3,B_3,Tmp_1);
Soustraction_bit(A_4,B_4,Tmp_1);
Soustraction_bit(A_5,B_5,Tmp_1);
// A la fin, Tmp_1 représente le bit de signe
```

Supposons maintenant qu'on souhaite calculer la valeur absolue de *A*, toujours en écrasant la valeur initiale de *A*, soit $A = |A|$. Voici la séquence correspondante :

```
// Initialisation de la bascule par le ET entre le bit de
// signe et le bit de poids faible de A ; on place la
// bascule dans le registre Tmp_2
rd A_0;
wd A_0;
wc Cal_1;
rd Tmp_1;
wd Tmp_1;
wc Cal_2;
ror Cal_1 etCal_2;
wc Tmp_2;
// Calcul du bit de poids 1 au bit de poids fort
ValeurAbsolue_Bit (A_1,Tmp_1,Tmp_2);
```

```
ValeurAbsolue_Bit (A_2,Tmp_1,Tmp_2);
ValeurAbsolue_Bit (A_3,Tmp_1,Tmp_2);
ValeurAbsolue_Bit (A_4,Tmp_1,Tmp_2);
ValeurAbsolue_Bit (A_5,Tmp_1,Tmp_2);
```

VI Comparaison et seuillage

VI.1 Procédures en pseudo-code booléen

Nous présentons à présent les opérateurs non linéaires de base. La Figure 7 présente le calcul de la comparaison stricte entre 2 nombres X et Y. A la fin du calcul, E vaut 1 si et seulement si $X < Y$, et F vaut 1 si et seulement si $Y < X$. Le coût en mémoire est de $2n + 6$, le coût en temps de calcul de $8n$. Si l'on se contente d'une comparaison large, le coût est moindre : il suffit de faire une soustraction et d'utiliser le bit de signe S (voir plus haut).

Une fois l'indicateur de comparaison stricte ou large calculé, Le *min* (ou le *max*) entre *X* et Y se calcule sur $3n+3$ bits ($2n+3$ si l'un des opérandes peut être écrasé). Voir Figure 7. Ici E et F peuvent être remplacés par $\neg S$ et S, respectivement, après le calcul de la différence X-Y.

<table>
<tr>
<td rowspan="2">D = E = F = 0
For i = n-1 downto 0
| $A = X_i \wedge \neg Y_i$
| $B = Y_i \wedge \neg X_i$
| $G = A \wedge \neg D$
| $E = E \vee G$
| $G = B \wedge \neg D$
| $F = F \vee G$
| $A = A \vee B$
| $D = D \vee A$

Comparaison stricte</td>
<td>For i = 0 to n-1
| $B = E \wedge X_i$
| $C = \neg E \wedge Y_i$
| $Min_i = B \vee C$
| $B = F \wedge Y_i$
| $C = \neg F \wedge X_i$
| $Max_i = B \vee C$

Calcul du min et du max</td>
</tr>
<tr>
<td>T = 0
For i = 0 to n-1
| if (t_i == 1)
| | $T = T \wedge X_i$
| if (t_i == 0)
| | $T = T \vee X_i$

Seuillage</td>
</tr>
</table>

Figure 7 : Calcul du min et du max - Seuillage

Enfin la Figure 7 présente le calcul du seuil d'une valeur *X* codée dans la rétine par rapport à une constante $T=\sum_{i=0}^{n-1} t_i 2^i$. Le nombre total d'opérations est de *n.* Le calcul du seuil n'utilise pas de bit supplémentaire pour le calcul.

VI.2 Procédures PVLSAR 34

(a) Procédures bit-wise

```
//------------------------------------------------------------
// Procédure bit-wise de comparaison de deux nombres X et Y
// Reg_X et Reg_Y représentent les 2 bits homologues de X et Y
// M_X est l'indicateur que X est plus grand que Y (alias : F)
// M_Y est l'indicateur que Y est plus grand que X (alias : E)
// Flag est la bascule (alias : D)
//-----------------------------------------------------------
void Compare_Bit(unsigned short Reg_X,unsigned short Reg_Y, unsigned short M_X,unsigned
short M_Y, unsigned short Flag) {

// Lecture des opérandes principaux
rd Reg_X;wd Reg_X;
wd Cal_1; wc Cal_2;
rd Reg_Y; wd Reg_Y;
wd Cal_3; wc Cal_4;
// Calcul du OU dans A (A = X_i et non(Y_i)), sur Cal_1,
// et de non(A) sur Cal_4
ror Cal_1 etCal_4;
wc Cal_1;
wd Cal_4;
// Calcul du OU dans B (B = Y_i et non(X_i)), sur Cal_2,
// et de non(B) sur Cal_3
ror Cal_2 etCal_3;
wc Cal_2;
wd Cal_3;
// Ecriture de D sur Cal5 et sur Cal_6
rd Flag;wd Flag;
wd Cal_5;
wd Cal_6;
// Calcul de G = A et non(D) sur Cal_4
ror Cal_4 etCal_5;
wc Cal_4;
// Ecriture de E sur Cal_5
rd M_Y;wd M_Y;
wd Cal_5;
```

```
// Calcul de E = E ou G
ror Cal_4 etCal_5;
wd M_Y;
// Calcul de G = B et non(D) sur Cal_3
ror Cal_3 etCal_6;
wc Cal_3;
// Copie de F sur Cal_4
rmt1 M_X;wd M_X;
wd Cal_4;
// Calcul de F = F ou G
ror Cal_3 etCal_4;
wd M_X;
// Calcul de A = A ou B sur Cal_1
ror Cal_1 etCal_2;
wd Cal_1;
// Copie de D sur Cal_2
ror IO_1;wd IO_1;
wd Cal_2;
// Calcul de D = D ou A
ror Cal_1 etCal_2;
wd Flag;

}
```

```
//-------------------------------------------------------------------
//
//  Seuillage de X par thr ; procédure bit-wise
//  Reg représente le bit courant de X,
//  S est le registre résultat du seuil
//
//-------------------------------------------------------------------
void Seuillage_bit(unsigned short X,int i,int thr,unsigned short S) {

rd Reg;
wc Cal_1;
wd Cal_2;

rd S;

if ( (( thr >>i)%2) == 0 )
  {
    wd Cal_3;
    ror Cal_2 etCal_3;
    wd S;
  }else{
    wc Cal_3;
    ror Cal_1 etCal_3;
    wc S;
  }

}
```

(b) Calcul numérique

Ici encore, des exemples suffiront à faire comprendre le principe général des routines numériques. Ainsi la comparaison stricte entre A et B, où les valeurs de A et de B sont conservées :

```
// Initialisation des indicateurs et bascules à zéro
one;
wc Tmp_1;
wc Tmp_2;
wc Tmp_3;
// Calcul du bit de poids fort au bit de poids faible
Compare_bit(A_5,B_5,Tmp1,Tmp_2,Tmp_3);
Compare_bit(A_4,B_4,Tmp1,Tmp_2,Tmp_3);
Compare_bit(A_3,B_3,Tmp1,Tmp_2,Tmp_3);
Compare_bit(A_2,B_2,Tmp1,Tmp_2,Tmp_3);
Compare_bit(A_1,B_1,Tmp1,Tmp_2,Tmp_3);
Compare_bit(A_0,B_0,Tmp1,Tmp_2,Tmp_3);
```

Et pour le seuillage, le code suivant calcule par exemple dans le registre Tmp_1, l'indicateur booléen de la condition "A est supérieur ou égal à 5" :

```
// Initialisation
one;
wd Tmp_1;
Seuillage_bit(A_0,0,5,Tmp_1);
Seuillage_bit(A_1,1,5,Tmp_1);
Seuillage_bit(A_2,2,5,Tmp_1);
Seuillage_bit(A_3,3,5,Tmp_1);
Seuillage_bit(A_4,4,5,Tmp_1);
Seuillage_bit(A_5,5,5,Tmp_1);
```

VII Ecriture de fonctions génériques

VII.1 Motivation

La première motivation est bien entendu la réutilisation de routines déjà écrites. Une routine écrite de façon générique sera utilisable par un autre programme ou un autre utilisateur beaucoup plus simplement.

La seconde motivation principale est la mémoire disponible pour les programmes Nios, qui est de 18ko aujourd'hui, ce qui est relativement faible. Il est donc important de pouvoir factoriser du code dans des routines, plutôt que de le dupliquer. Un appel de fonction prend seulement 2 instructions assembleurs.

VII.2 Généricité par rapport au bit mémoire concerné

L'écriture d'une fonction prenant en paramètre les bits mémoire sur lesquels elle doit travailler se fait en passant en paramètre un unsigned short représentant les coordonées du bit en mémoire.

Exemple :

```
// Définition de la function
void write_one_on_bit(unsigned short X)
{
  one; wd X;
}

// Exemple d'appel
write_one_on_bit(y1 x2);
```

VII.3 Généricité par rapport à une ligne mémoire

L'indice de la ligne se passe également en paramètre par un unsigned short, mais cette fois ci pour accéder à un bit particulier de la ligne, il faut effectuer un ou logique avec l'indice, comme dans l'exemple suivant :

```
void write_one_on_line(unsigned short Line)
{
     one;
     wd Line|x0;
     wd Line|x1;
     wd Line|x2;
     wd Line|x3;
```

```
     wd Line|x4;
     wd Line|x5;
}

// Exemple d'appel
// Noter le 'm' suffixé à y1, désignant la ligne y1.
write_one(y1m);
```

VII.4 Généricité par rapport à un ensemble de bits quelquonque

Cette généricité se base sur le principe des itérateurs de la STL (Standard Template Library) du C++. L'idée est de fournir à la fonction un moyen de parcourir les différents bits sur lesquels elle doit travailler de façon optimale. Le principe s'appuie sur le fait qu'en C un pointeur peut s'incrémenter, et que la mémoire est contiguë.

L'exemple suivant est une illustration du principe des itérateurs dans le cas général avec une fonction fill qui peut remplir n'importe quelle zone mémoire avec une certaine valeur :

```
// Définition de la fonction
void fill(unsigned short *begin, unsigned short *end, int value)
{
  while (begin != end)
  {
    *begin = value;
    ++begin;
  }
}

// Exemple d'appel
unsigned short tab[50]; // déclaration d'un tableau de 50 valeurs
fill(tab, tab+50, 12); // remplissage du tableau avec des 12
```

On peut se servir du même concept dans un programme Nios pour écrire des fonctions complètement génériques sur les bits mémoire à utiliser.

Exemple :

```
// Définition de la fonction
void write_one_on_nbits(unsigned short *begin, unsigned short
*end)
{
  one;
  while (begin != end)
  {
    wd *begin;
    ++begin;
  }
}

// Définition des bits sur lesquels la fonction doit travailler.
unsigned short bits[] = { y1 x0, y2 x4, y3 x1 };

// Remplissage des bits avec des 1.
write_one_on_nbits(bits, bits+3);
```

VII.5 Généricité par rapport à une sous-procédure

Le besoin de généricité par rapport à une procédure s'exprime souvent quand une fonction doit manipuler des pixels voisins, par exemple une fonction qui copie la valeur d'un bit d'un voisin dans sa propre mémoire. On peut avoir envie de copier celui du voisin Nord, Sud, Est, Ouest, etc., et il serait intéressant de n'écrire le code qu'une seule fois.

Ceci est possible dans les programmes Nios, en passant des pointeurs sur fonction en paramètres à des procédures.

Exemple :

```
// Définit le type func_t : pointeur sur fonction qui ne prend et
ne renvoie rien
typedef void (*func_t)(void)

void get_N()
{
  wd Com_S;
  rd Com_N;
```

```
}

void get_E()
{
  wd Com_W;
  rd Com_E;
}

// Copie le bit Src d'un voisin dans le bit Dest local
void copie_voisin(func_t get_neighbor, unsigned short Src,
unsigned short Dest)
{
  rd Src; wd Src;
  get_neighbor();
  wd Dest;
}

// Exemple d'appel
copie_voisin(get_N, y1 x1, y2 x1);
copie_voisin(get_E, y1 x1, y2 x1);
```

VII.6 Impact de la généricité sur les performances

Le Nios travaillant à une fréquence plus élevée que la rétine, la perte liée à l'utilisation d'appels de fonctions et de boucles est généralement nulle ou négligeable. L'utilisation de fonctions peut même entraîner un gain si on profite du compartimentage des procédures pour gérer finement l'utilisation de la pile d'instruction rétines en empilant au maximum les instructions afin de toujours garder de l'avance sur l'exécution des instructions rétines.

Le gain en terme de taille de code est en revanche très significatif par rapport à des approches de type macros ou copier/coller de code. Le gain en terme de génie logiciel est également conséquent puisqu'il devient possible d'écrire une bibliothèque de routines rétiniennes optimisées, que chacun peut utiliser à loisir sans y retoucher.