ECOLE
NATIONALE
SUPERIEURE
D'**E**LECTRONIQUE ET DE
RADIOELECTRICITE DE
BORDEAUX

Présenté par : Dominique BOUVERON
Florent RENAHY
Responsables : Patrice KADIONIK
Patrice NOUEL

RAPPORT DE PROJET DE FIN D'ETUDES

—

CARTE D'ACQUISITION VIDEO POUR L'ETUDE D'ALGORITHMES DE TRAITEMENT D'IMAGES

Avril - Juin 1998

REMERCIEMENTS

Nous tenons à remercier les différentes personnes qui par leurs conseils ou leurs suggestions ont permis de rendre ce projet aussi instructif que plaisant :

Monsieur Patrice NOUEL, pour son aide sur les outils Mentor et ses conseils avisés,

Monsieur Patrice KADIONIK, pour son soutien et son aide tout au long du projet,

Monsieur Yannick BERTHOUMIEU, pour son aide concernant les algorithmes de traitement d'images,

Monsieur Laurent DULAU, pour son support et ses connaissances des outils Mentor,

Nous tenons également à exprimer notre gratitude envers tout le personnel technique de l'ENSERB, sans oublier notre sponsor officiel *The Coca Cola Company.*

TABLE DES MATIERES

1. PRESENTATION DU PROJET :

1.1. INTRODUCTION :

Les concepteurs d'applications de traitement du signal sont souvent forcés de choisir entre flexibilité et performance à cause des solutions limitées disponibles. Les processeurs DSP sont peu coûteux, flexibles et peuvent être utilisés dans de nombreuses applications. Toutefois, ces processeurs ne peuvent délivrer qu'une puissance de calcul limitée en temps réel à cause du séquencement à travers le goulot d'étranglement de l'accumulateur - multiplieur. Les concepteurs qui désirent de meilleurs résultats doivent alors utiliser plusieurs processeurs DSP en parallèle. D'autre part, les processeurs DSP dédiés et les ASICs offrent de meilleures performances, mais ne sont pas flexibles. Peu de processeurs dédiés existent sur le marché, et les ASICs requièrent beaucoup de temps en développement, test et production.

Les applications industrielles d'imagerie demandent du filtrage en temps réel ce qui requiert une performance de plusieurs milliers de MIPS. Les fonctions de traitement numérique du signal sont de plus en plus utilisées dans le traitement de signaux RF pour des applications telles que analyseur de spectre RF, radar... Ces applications requièrent des solutions de traitement numérique de signal puissantes et flexibles. Les circuits programmables fournissent performance et flexibilité requises dans de telles applications. Parce que les algorithmes sont optimisés pour l'architecture du composant, la performance de ces circuits peut dépasser celle des processeurs DSP et ASICs.

Les solutions traditionnelles de traitement numérique du signal sont résumées ci-dessous :

- ***Microprocesseurs*** – Ils sont toujours utilisés dans des applications moins exigeantes en puissance de calcul comme générer du son sur les PCs.
- ***Processeurs DSP*** – Comme les concepteurs demandaient de plus en plus de puissance de calcul pour des algorithmes complexes, le processeur DSP a été développé. Le processeur DSP a généralement une architecture de type RISC, et fournit une puissance de calcul de 3 à 50 MIPS. Ils peuvent réaliser en un cycle une opération qui requiert beaucoup de cycles dans un microprocesseur.
- ***Application-specific standard products*** (ASSPs) – La demande de performance croissante a conduit au développement de circuits dédiés, spécifiques tels que des filtres FIR et des processeurs de transformée de Fourier rapide (FFT). Ces circuits permettent d'optimiser une fonction au niveau matériel.
- ***Application Specific Integrated Circuits*** (ASIC) – Des modules pour des fonctions de traitement du signal tels que le décodage MPEG simplifient la conception d'un ASIC. Cependant les coûts et temps de développement de ces circuits les dédient aux grandes séries.

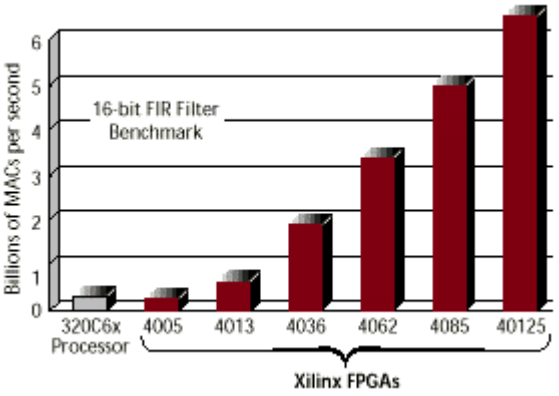
Les circuits programmables fournissent une puissance de calcul importante en maintenant la complète flexibilité d'un processeur DSP. Des algorithmes traditionnellement considérés comme une séquence d'instructions peuvent être analysés pour déterminer un parallélisme possible. La possibilité de reconfiguration sur la carte (**I**n-**C**ircuit **R**econfigurability) permet une minimisation et une adaptation spécifique à chaque algorithme.

Implanter des fonctions de traitement du signal dans un composant programmable fournit les avantages suivants :

- **Parallélisme** – Implémenter des algorithmes au niveau matériel permet un parallélisme au niveau calcul, ce qui conduit à des performances supérieures par rapport à des processeurs DSP. L'échelle d'une fonction peut être augmentée (par ex. en augmentant le nombre d'étages pour un filtre FIR) avec une dégradation minime des performances. Au contraire, le traitement séquentiel des processeurs DSP conduit à une dégradation des performances proportionnel à l'augmentation du nombre de traitements.

- ***In-circuit reconfigurability*** (ICR) – Ceci permet de charger un nouvel algorithme dans le circuit en fonctionnement sur la carte. Cette possibilité permet des mises à jour d'algorithmes rapides ce qui est important pour notre application.

- **Efficacité** – De nombreux algorithmes tels que les filtres FIR nécessitent la multiplication par une constante. L'architecture du composant programmable (par ex. Altera FLEX) contient des *look-up tables* (LUTs) qui peuvent simplifier la multiplication. La table contient les valeurs pré-calculées, éliminant ainsi la taille et le coût en performance d'une cellule multiplieur.



L'amélioration des performances des circuits programmables résulte de l'augmentation de la fréquence d'horloge et du nombre de cellules logiques du circuit.

Figure 1 : Comparaison entre un processeur DSP et les circuits FPGA Xilinx

Le traitement d'images en temps réel conduit à employer des circuits électroniques rapides, capables de manipuler les grandes quantités d'informations générées par la source vidéo. Pour cela, les circuits logiques programmables sont particulièrement adaptés. Des circuits intégrés spécifiques (ASIC) ont été développés pour des traitements particuliers, ils ne permettent pas de modifier la fonction de traitement une fois le circuit réalisé. Nous allons d'abord étudier le cahier des charges pour ensuite détailler l'architecture de la carte qui exploite la reconfigurabilité offerte par les circuits programmables pour permettre la mise au point rapide de filtrages vidéo.

1.2. CAHIER DES CHARGES :

Dans une application de traitement d'images, le signal vidéo traverse un ensemble de modules qui forment une chaîne de traitement.

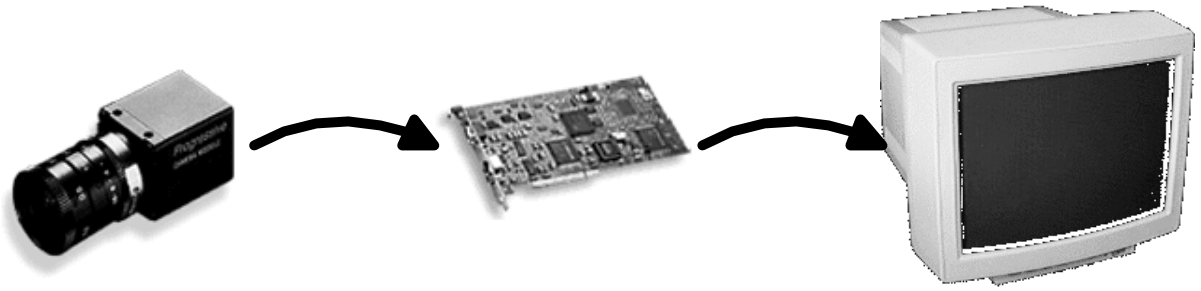


Figure 2 : Chaîne de traitement vidéo

Le système de traitement vidéo complet inclut la numérisation du signal vidéo et filtrage des pixels par des opérateurs de calcul (de bas niveau). Des algorithmes de plus haut niveau agissent sur des zones d'images mémorisées, afin d'extraire les informations utiles pour l'application. Dans notre projet, nous nous limitons à la partie acquisition - traitement - restitution d'un signal vidéo fourni par une caméra de télésurveillance.

Le signal vidéo est à la norme CCIR 625 lignes, 50Hz, 1Vcc, 75 ohms en charge. L'acquisition du signal vidéo sera effectuée par un convertisseur analogique - numérique flash 8 bits afin d'obtenir une bonne résolution de l'image.

Les données seront traitées par un composant programmable de type FPGA Xilinx afin d'autoriser le téléchargement de nouveaux algorithmes pour le test et la mise au point d'algorithmes de traitement d'images. L'algorithme de détection de contours de type gradient sera implanté dans ce composant programmable.

Le flot d'images traité sera alors visualisé sur un moniteur de contrôle. Le but étant d'obtenir un traitement pseudo temps réel de l'image, le temps de traitement d'une image devra être le plus court possible.

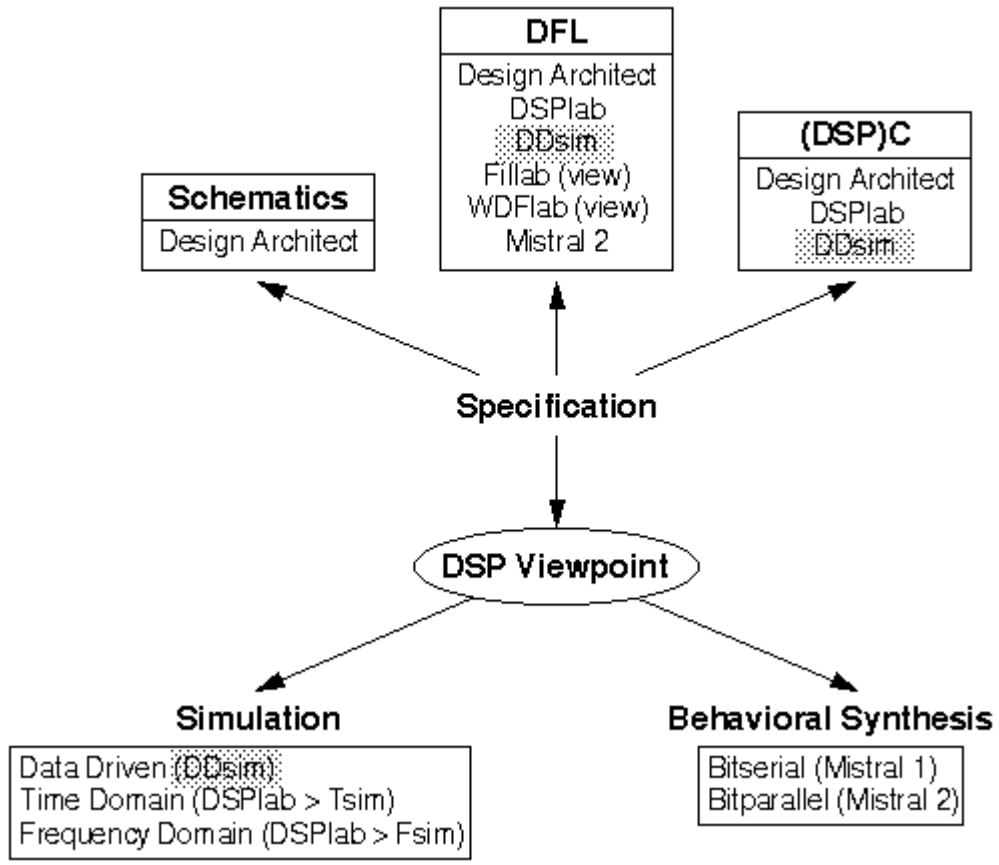
Pour ce projet, on utilisera les composants programmables FPGA Xilinx puisque tous les outils de développement sont à notre disposition à l'ENSERB. La carte doit être simple et de faible coût pour une utilisation pédagogique.

2. DSP STATION DE MENTOR GRAPHICS (FRONTIER DESIGN) :

DSP Station est un ensemble d'outils permettant la conception, la simulation et l'optimisation de systèmes de traitement numérique du signal. Cet environnement, bien que très orienté vers les processeurs DSP, ne leur est pas exclusivement réservé. En effet, ces outils travaillent à partir d'un langage très haut niveau, le Data Flow Language (DFL).

Le langage DFL a les caractéristiques typiques d'un langage de description comportemental pour les algorithmes DSP. C'est un langage applicatif pour l'écriture d'algorithmes DSP, avec des types de données et des opérations *bit-true*. Il dispose d'opérateurs de retard temporel, de bibliothèques de fonctions mathématiques usuelles, de différents modes d'affectation de variables, etc. Les opérations ne sont effectuées que lorsque tous les opérandes sont disponibles en entrée, ce qui est intéressant pour le traitement parallèle des données et donc, indirectement, pour la rapidité du système.

Figure 3 : Chaîne de développement DSP Station



2.1. LIBRAIRIE DSP_IMAGELIB :

De nombreuses librairies sont fournies avec DSP Station. Par exemple, la librairie image (dans le répertoire $EDC_DSP_IMAGELIB) met à notre disposition les fonctions *Source* et *Sink*.

Presque tous les éléments des librairies ont été implémentés avec des types génériques DFL. Pour les simulations haut niveau, cela ne posera aucun problème. Cependant pour les simulations bit-true, il est nécessaire de vérifier le code DFL pour voir quel type a été choisi, et si nécessaire modifier le code pour l'adapter à notre application.

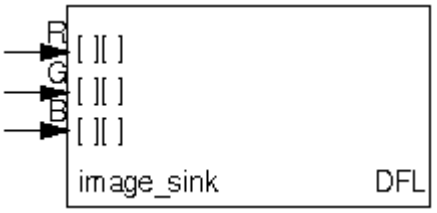
Le composant *Sink* permet de sauvegarder une image au format GIF ou JPEG, alors que le composant *Source* permet de lire une image aux mêmes formats. Des exemples d'utilisation de ces composants sont disponibles dans le fichier *image_demo.dfl* dans le répertoire $EDC_DSP_IMAGELIB/Examples où l'on peut trouver également l'implémentation en DFL des algorithmes de traitement d'images *blur*, *emboss*, *filtre médian*, *détection de contour* ...

La fonction *Sink* sauvegarde une image couleur dans un fichier `P_FILENAME`. Les entrées sont trois tableaux du type `unsigned_int<P_DEPTH>[P_HEIGHT][P_WIDTH]`. Ceux-ci contiennent les informations pour les 3 canaux Rouge, Vert et Bleu. Les paramètres suivants doivent être spécifiés :

P_WIDTH	largeur en pixels de l'image
P_HEIGHT	hauteur en pixels de l'image
P_DEPTH	nombre de bits utilisés pour chaque canal couleur
P_FILEFORMAT	format de l'image sauvegardée sur disque (gif ou jpeg)
P_FILENAME	nom du fichier sauvegardé sur le disque
P_DISPLAY_CMD	commande exécutée après sauvegarde de l'image sur disque pour la visualiser
P_JPEG_QUALITY	paramètre optionnel pour spécifier le taux de compression JPEG

Voici un exemple d'utilisation de la fonction *Sink* :

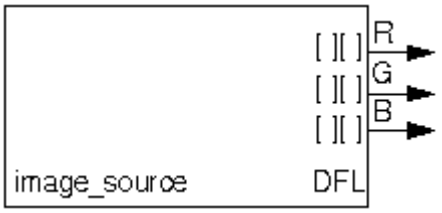
```
#define HEIGHT 125
#define WIDTH  125
#define DEPTH  8
#define PIXEL  unsigned_int<DEPTH>
#define IMAGE PIXEL[HEIGHT][WIDTH]
var R,G,B : IMAGE;
(R,G,B) := IMAGE_SINK :: image_sink
{
P_HEIGHT      : WEIGHT,
P_WIDTH       : WIDTH,
P_DEPTH       : DEPTH,
P_DISPLAY_CMD : "xv",
P_FILENAME    : "$HOME/lena.gif",
P_FILEFORMAT  : "gif"
} (R,G,B);
```



La fonction *Source* lit une image couleur d'un fichier `P_FILENAME` au format `P_FILEFORMAT` et exécute la commande `P_DISPLAY_CMD` sur ce fichier. Les données lues sont stockées dans 3 tableaux du type `unsigned_int<P_DEPTH>[P_HEIGHT][P_WIDTH]`. Ceux-

ci contiennent les informations pour les 3 canaux Rouge, Vert et Bleu. Des paramètres identiques à ceux de la fonction *Sink* doivent être spécifiés.

Voici un exemple d'utilisation de la fonction Source :




```
#define HEIGHT 125
#define WIDTH  125
#define DEPTH  8
#define PIXEL  unsigned_int<DEPTH>
#define IMAGE PIXEL[HEIGHT][WIDTH]
var R,G,B : IMAGE;
(R,G,B) := IMAGE_SOURCE :: image_source {
P_HEIGHT      : WEIGHT,
P_WIDTH       : WIDTH,
P_DEPTH       : DEPTH,
P_DISPLAY_CMD : "xv",
P_FILENAME     : "$EDC_DSP_IMAGELIB/Examples/lena.jpg",
P_FILEFORMAT  : "jpg"
}();
```

Dans notre application, nous n'utilisons que des images noir et blanc. Pour contourner ce problème, il existe deux moyens : l'un consiste à créer une matrice d'entrée pour y stocker le résultat de le conversion couleur → gris effectuée par la fonction MONO., l'autre consiste à n'utiliser que la matrice R par exemple si le fichier lu a été converti en échelle de gris auparavant.

Nous allons tout d'abord expliquer en quoi consiste la détection de contour avec le gradient de Prewitt, avant de voir son implémentation en DFL.

2.2. DETECTION DE CONTOUR AVEC LE GRADIENT DE PREWITT:

Soit un contour d'orientation θ au point (x, y). Ce contour est détecté par le maximum de la dérivée dans la direction ϕ du gradient $\vec{\nabla} f(x,y)$ soit le maximum de la fonction $g(\phi) = \vec{\nabla} f(x,y).\vec{n}$ où $\vec{n}$ est le vecteur unitaire dans la direction du gradient $\vec{n} = (\cos\phi, \sin\phi)$.

.

$$g(\phi) = \cos\phi.\frac{df}{dx} + \sin\phi.\frac{df}{dy}$$

Dans le cas discret (qui est notre cas puisque l'on travaille avec des pixels), on approxime les dérivées partielles par de simples différences :

$$\frac{df}{dx} = f(i+1, j) - f(i, j) = \Delta_x f(i, j)$$
$$\frac{df}{dy} = f(i, j+1) - f(i, j) = \Delta_y f(i, j)$$

L'opérateur norme du gradient est donné par

$$\left\|\vec{\nabla}\vec{f}\right\| = \max(\left|\Delta_x f\right|, \left|\Delta_y f\right|)$$

Raisonnons avec des matrices qui représentent les images d'entrée (A) et de sortie (B), où $A, B \in \mathrm{M}([0,511],[0,511])$.

Les dérivées directionnelles horizontale et verticale s'expriment sous la forme :
gradient en x : A_x (i, j) = H * A(i, j) (* = convolution)
gradient en y : A_y (i, j) = V * A(i, j).

On aura alors $B(i,j) = \max(|A_x(i,j)|, |A_y(i,j)|)$

avec $H = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$ et $V = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$

H et V sont appelées masques, avec $H, V \in \mathrm{M}([-1,1],[-1,1])$

Elles dépendent du type de gradient calculé, ici gradient de Prewitt. Développement :

$$A_y(i,j) = \sum_{k=-1}^{1}\sum_{l=-1}^{1} A(i-k, j-l).V(k,l)$$

$$A_x(i,j) = \sum_{k=-1}^{1}\sum_{l=-1}^{1} A(i-k, j-l).H(k,l)$$

On constate que l'on ne pourra pas calculer les pixels des bords de l'image de sortie, correspondant aux indices i={0, 511}et j={0,511}, puisque (i- k) et (j- l) ne doivent pas avoir les valeurs -1 ou 512. Les pixels de sortie effectivement calculés seront donc B(i, j) avec $(i,j) \in [1,510]^2$

On peut remarquer que **H = a + b + c et V = a - c - d** avec :

$$a = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, b = \begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, c = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}, d = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

D'où :

A_x = Ca + Cb + Cc
A_y = Ca - Cc - Cd
Ca = -A(i+1,j+1) + A(i-1,j-1) ; Cb = -A(i,j+1) + A(i,j-1)
Cc = -A(i-1,j+1) + A(i+1,j-1) ; Cd = -A(i-1,j) + A(i+1,j)

En résumé , avec **A(i, j)** matrice d'entrée et **B(i, j)** matrice de sortie :

$B(i,j) = (1/3).\max(|Ca + Cb + Cc|, |Ca - Cc - Cd|)$, $(i,j) \in [1,510]^2$
Ca = -A(i+1,j+1) + A(i-1,j-1) ; Cb = -A(i,j+1) + A(i,j-1) ;
Cc = -A(i-1,j+1) + A(i+1,j-1) ; Cd = -A(i-1,j) + A(i+1,j) ;

Le facteur (1/3) introduit servira à normaliser les coefficients de la matrice. Effectivement : $\max(A_x, A_y) \in$ [0,3*255] puisque les pixels sont codés sur un octet.

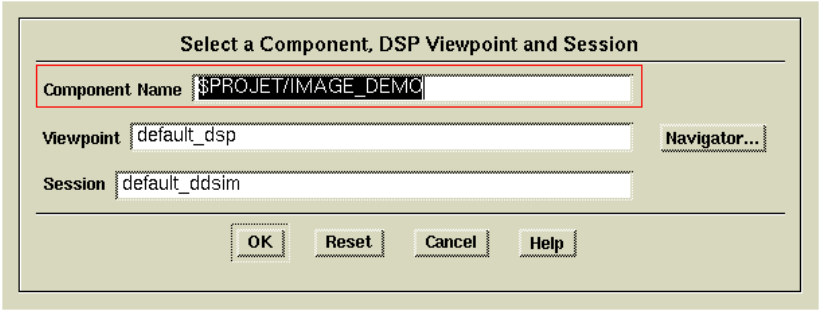
L'implémentation en DFL de cet algorithme est alors simple. On prendra comme dimensions de l'image 512x512. On choisira des types de variables adaptées pour viser une implantation matérielle minimale.

Remarque : *les sources DFL 'image_demo.dfl' (programme principal), et 'edetect.dfl' (algorithme de détection de contours) sont disponibles en annexe.*

2.3. DDSIM, DYNAMIC DATAFLOW SIMULATOR :

DDsim, ou le simulateur dynamique de flot de données, est un environnement de simulation interne à DSP Station qui effectue une simulation des flots de données dans le temps. Avant d'effectuer une simulation avec DDSIM, il faut d'abord spécifier l'algorithme DSP à simuler. Cela peut être soit un schéma, du code DFL ou encore (DSP)C. L'édition graphique peut être réalisée avec Design Architect. Quant au code DFL, cela s'effectue dans DDSIM, DSPlab ou Mistral2. Ensuite, il faut créer le graphe des flots de signaux (Signal Flow Graph) en compilant le code DFL pour la création d'un « point de vue » (*viewpoint*).

DDsim peut être lancé depuis l'environnement Design Manager (dmgr). Apparaît avec la fenêtre de session DDsim la boîte de dialogue suivante , où on spécifie le composant



IMAGE_DEMO compilé auparavant dans DSPlab par exemple.

Figure 4 : Sélection du composant IMAGE_DEMO

Dès que le design a été chargé, on peut lancer la simulation en cliquant sur le bouton *Run*. Apparaît alors l'image source sur laquelle sera appliqué le filtre de détection de contours. A la fin du calcul, l'image traitée est également affichée. Il faut appuyer sur le bouton *Halt* pour arrêter la simulation.

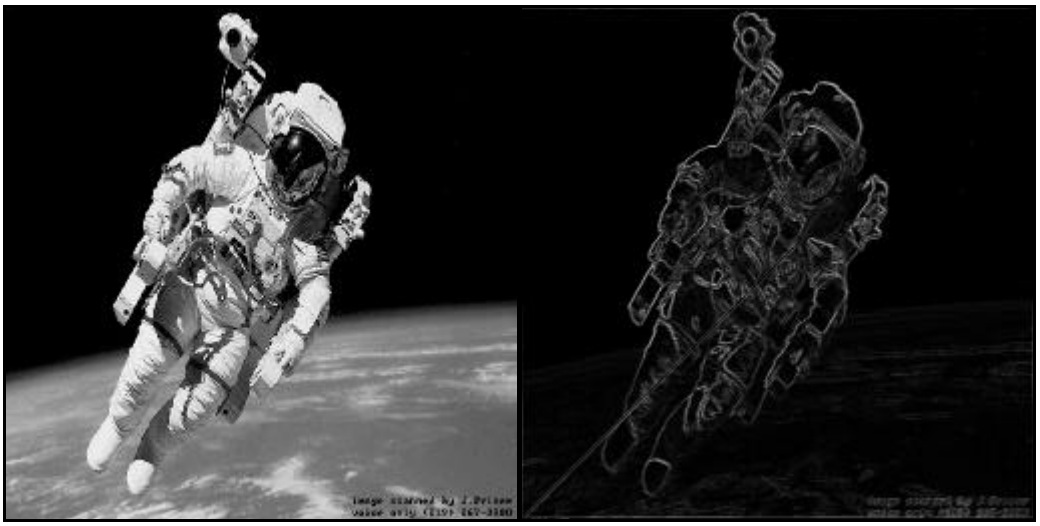
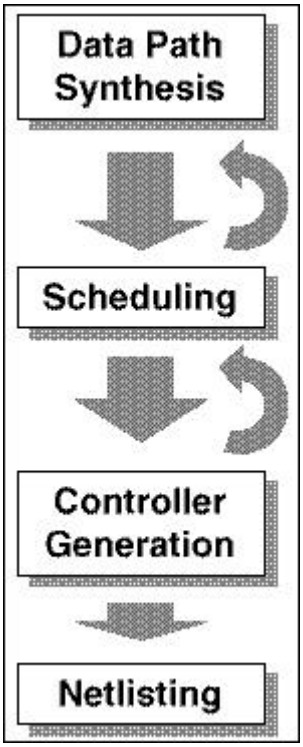


Figure 5 : Image avant et après traitement

Nous n'avons pu déterminer l'origine exacte de la ligne diagonale sur l'image traitée, toutefois elle n'est pas due à l'algorithme. En effet, codé en C, cette erreur n'apparaît plus. On suppose qu'elle est due à la conversion couleur → gris → couleur dans le programme principal.

2.4. SYNTHESE DU CODE DFL AVEC MISTRAL 2 :

Mistral2 est un programme de synthèse pour créer à partir d'un graphe de flot de signaux (SFG) d'un système numérique une architecture du type *bitparallel*. Cette synthèse est effectuée en plusieurs étapes :



- **Datapath synthesis** – description de l'architecture du processeur en terme d'unités d'exécution et interconnexions (bus) nécessaires pour exécuter l'algorithme spécifié par le SFG :
- **Scheduling** – conversion de la description des transferts entre registres sur un axe de temps divisé en cycles machines. L'optimisation consiste en la minimisation du nombre total de cycles machines nécessaires.
- **Busmerging** – le coût d'interconnexion (bus, multiplexeurs) peut être réduit en effectuant cette opération après le processus *Scheduling*. Parce que la probabilité de collisions de bus augmente si le nombre de bus est réduit, moins de transferts entre registres pourront se dérouler en un cycle machine. Donc cette opération pourrait augmenter le nombre de cycles machines nécessaires.
- **Controller synthesis** – un contrôleur capable d'ordonner le transfert entre registres est généré depuis le microcode symbolique.
- **Netlist generation** – cette étape construit une description du processeur généré. Le fichier généré est une liste des nets contenant les instances des unités d'exécution et du contrôleur.

Dans cette architecture de type *bitparallel*, tous les bits sont traités pendant la même période d'horloge. Cela résulte en une occupation matérielle plus importante par rapport à l'architecture *bitserial*. L'architecture du processeur est un ensemble d'éléments (**EX**ecution **U**nit) communiquant par un bus dédié et contrôlés par un programme stocké dans le contrôleur.

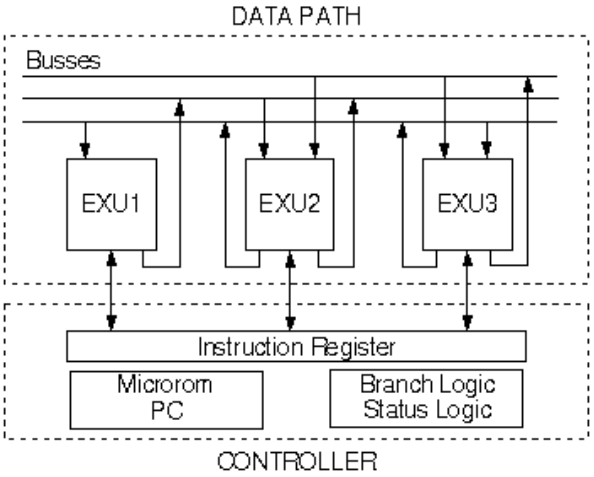


Figure 6 : Architecture d'un processeur Mistral 2

Les unités basiques utilisées dans les unités d'exécution sont : une mémoire lecture - seule (ROM / ROMCTRL), une mémoire RAM, une unité arithmétique logique (ALU), différents multiplieurs et des ports d'entrée/sortie.

On peut visualiser dans Mistral 2 l'architecture générée pour notre composant dans le menu View → Architecture.

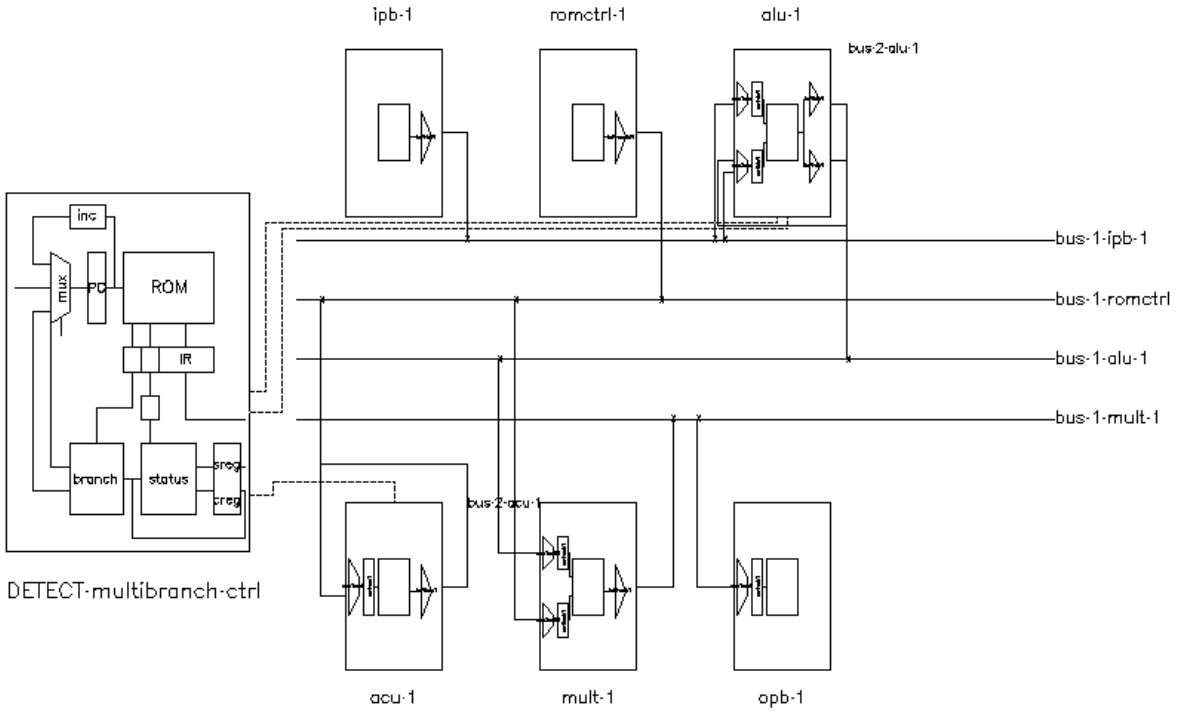


Figure 7 : Architecture du composant EDETECT.

La dernière étape consiste à générer le code VHDL qui sera synthétisé à l'aide de Galileo. Puisque les composants de la librairie ms2_std_lib ne sont pas compilés sur le système, il est nécessaire d'inclure dans le fichier VHDL le code source des composants de cette librairie avant de compiler le fichier. Voici le résultat de la synthèse par Galileo :

```
-- Start optimization for design .work.mistral2_edetect_topview.struct
Using default wire table: 4025e-3

      Pass     Area    Delay     DFFs  PIs   POs --CPU--
               (FGs)    (ns)                    min:sec
      1       1036       97       186    11    11   14:01
Info, Added global buffer BUFG for port phi
```

On lance ensuite Design Manager pour obtenir l'occupation du composant dans un circuit X4025e, ce qui nous donne les résultats suivants :

```
   Number of CLBs:            620 out of  1024   60%
      CLB Flip Flops:     184
      4 input LUTs:      1028
      3 input LUTs:       419 (32 used as route-throughs)

   Number of bonded IOBs:      22 out of   192   11%
      IOB Flops:            2
      IOB Latches:          0
   Number of global buffers:    1 out of     8   12%
   Number of primary CLKs:      1 out of     4   25%
```

Pour l'implantation d'algorithmes bas niveau de traitement d'images, il apparaît que la synthèse par Mistral 2 du code DFL soit sur-dimensionnée, mais nous ne maîtrisons pas tous les paramètres d'optimisation de Mistral 2. Néanmoins pour des traitements plus importants, cette méthode simplifierait la tâche du concepteur.

Dans notre cas, suite à cette étude, le choix a été de développer directement en VHDL afin d'optimiser le calcul et notamment les accès aux différentes mémoires.

3. ARCHITECTURE DE LA CARTE :

On peut décomposer le système en cinq modules principaux :

• **Conversion analogique/numérique** – elle est effectuée par un convertisseur ADC flash 8 bits TDA8708A. Le signal vidéo, après filtrage passe-bas, est échantillonné à une fréquence choisie par l'utilisateur. Cette partie est contrôlée par le composant programmable XC4006e **RAM_CONTROL.**

• **Synchronisation vidéo** – le circuit LM1881 est chargé de prendre, dans le signal vidéo, les différents signaux de synchronisation (*csync*, *vsync* et *burst*) et de les fournir au séquenceur.

• **Séquenceur** – intégré dans le composant programmable **RAM_CONTROL,** il devra gérer l'acquisition et la restitution du signal vidéo, ainsi que réguler l'accès aux mémoires entre les convertisseurs et le circuit XC4006e **CALCUL.**

• **Traitement de l'image** -- intégré dans le composant programmable **CALCUL,** il est effectué de manière asynchrone par rapport à l'acquisition des données. Contenant uniquement l'algorithme de traitement ainsi qu'une mémoire tampon de 9 pixels, il accède aux données dans la mémoire d'entrée, effectue le calcul pour chaque pixel de l'image et écrit le résultat dans la mémoire de sortie.

• **Conversion numérique/analogique** -- elle est effectuée par un convertisseur DAC 8 bits TDA 8702. Associé à un étage de mise en forme, il génère un signal vidéo qui sera envoyé sur un moniteur de contrôle afin de visualiser l'image traitée.

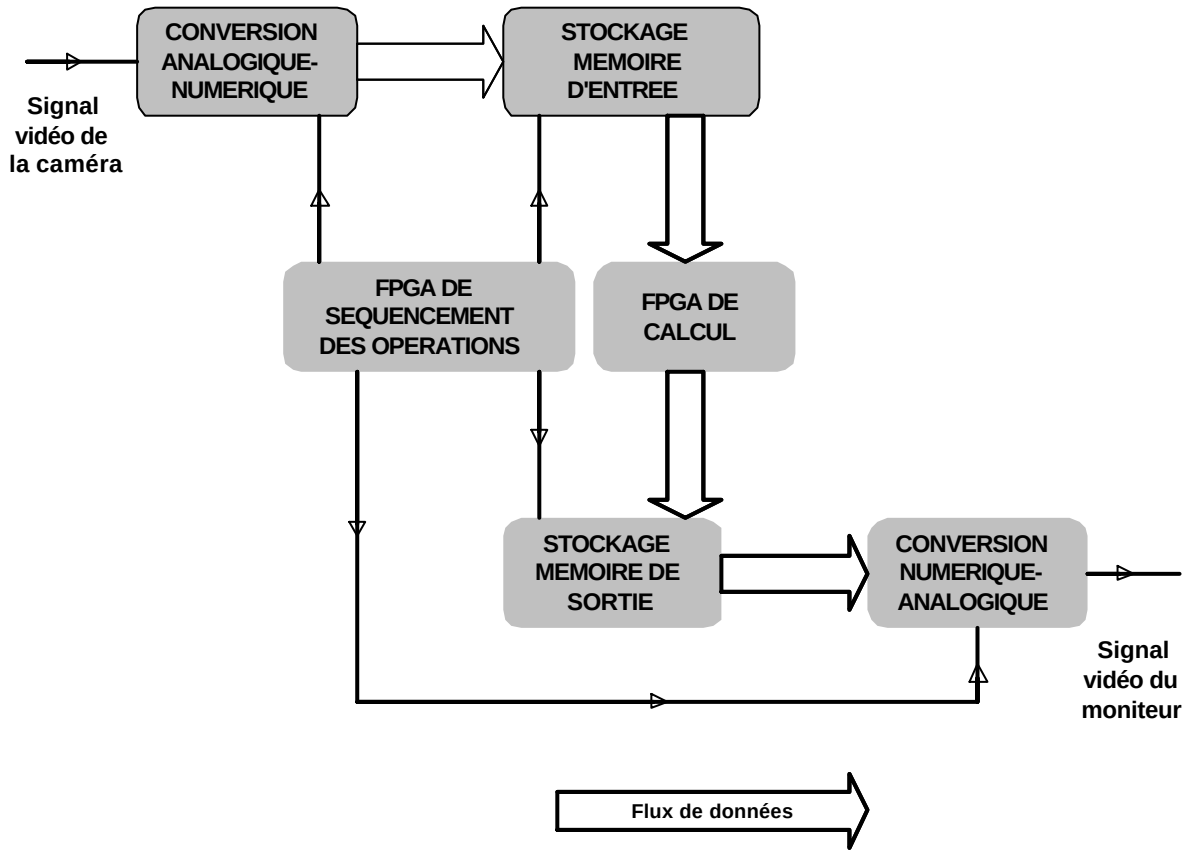


Figure 8 : Synoptique du montage

3.1. PARTIE ANALOGIQUE :

Le circuit LM 1881 extrait, du signal vidéo, les signaux de synchronisation nécessaires au déroulement des opérations. Ces signaux sont :

- signal Composite Synchro appelé Csync qui donne les tops lignes et trames.
- signal Vertical Synchro appelé Vsync qui indique le changement de trames.
- signal Burst qui fournit un top juste avant le début du « blanking » qui est la partie du signal utile sur une ligne.

Le circuit TDA 8708A est un convertisseur analogique/numérique 8 bits. Il intègre un amplificateur vidéo avec blocage et contrôle automatique de gain. En lui fournissant les signaux appropriés (GATE A et GATE B selon la documentation du composant) il peut calibrer automatiquement son échelle de conversion en fonction du signal vidéo qu'il reçoit.

L'inconvénient avec ce convertisseur est que la plage de conversion (avec le fonctionnement en mode 2 qui est notre cas) va de 64 pour le niveau de noir à 248 au maximum pour le niveau de blanc. Le signal GATE A doit être présent pendant le top de synchronisation ligne afin que le convertisseur sache à quel niveau se trouve celui-ci. Il en fixe le niveau à zéro. Le signal GATE B doit être présent au moment du top Burst qui donne alors le niveau du noir de l'image, qui est codé 64.

Enfin un « Peak level comparator » surveille le niveau maximum du signal vidéo (correspondant au niveau du blanc) et le code à 248. Si le signal vidéo tend à excéder 248 le gain en interne sera limiter pour éviter tout dépassement de capacité du convertisseur.

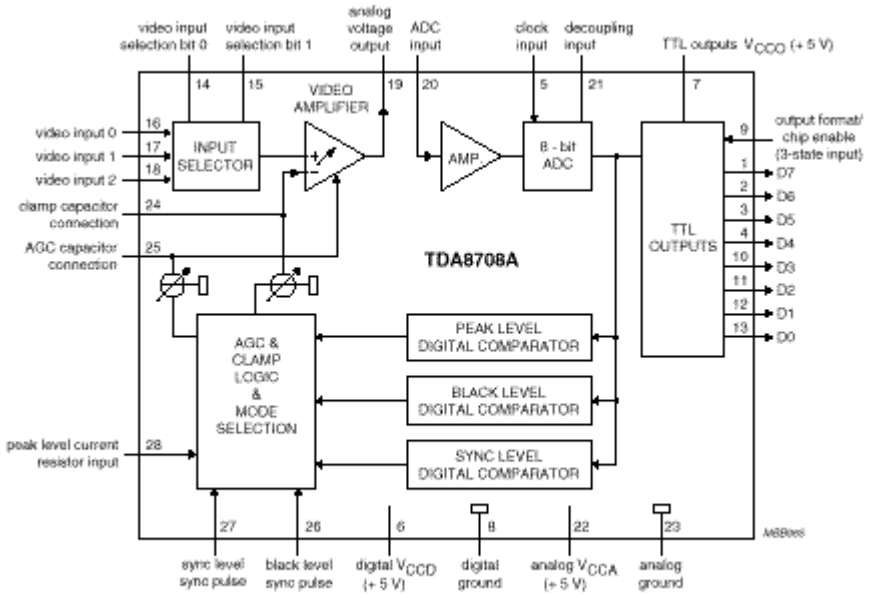


Figure 9 : Schéma interne du TDA8708A

Nous avons pris le filtre suivant donné dans la documentation du composant, placé entre les broches 19 et 20.

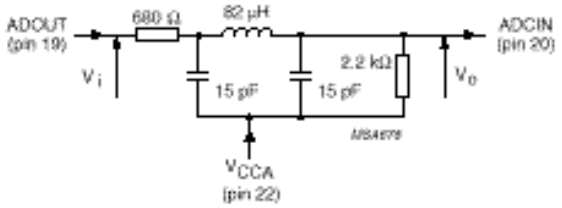


Figure 10 : Filtre passe-bas d'ordre 5 type Chebyshev

En ce qui concerne l'étage de sortie, le problème est la restitution de l'image sur un moniteur par le convertisseur. En effet, lors de l'échantillonnage, seule l'information utile est conservée ; on perd ainsi les signaux de synchronisation vidéo. Pour recomposer le signal vidéo en sortie, il est nécessaire d'ajouter le signal de synchronisation aux données traitées (qui sont stockées dans la mémoire de sortie). Pour récupérer le signal de synchronisation, deux méthodes sont possibles :

- la première consiste à régénérer artificiellement le signal de synchronisation dans un composant programmable ou par un circuit tel que le générateur de synchronisation Philips SAA1101,
- la seconde consiste à utiliser le signal de synchronisation extrait par le circuit LM1881 du signal vidéo d'entrée. Cette méthode, plus simple, nous permet alors d'utiliser les mêmes compteurs ligne et colonne pour la lecture des mémoires.

Le circuit TDA 8702 est un convertisseur numérique/analogique de type flash. La conversion s'effectue sur le niveau d'horloge à l'état bas. Si l'horloge est en permanence à l'état bas tous les changements de code en entrée sont répercutés en sortie. Après une conversion, si l'horloge repasse à l'état haut la valeur à convertir est mémorisée et les éventuels changements à l'entrée ne sont pas pris en compte.

Ce convertisseur a une sortie courant, d'où l'utilisation d'un transistor à la broche 15.
Soient X0 le signal à la sortie de l'amplificateur opérationnel, qui rentre sur la broche X0 du multiplexeur analogique, X1 le signal qui rentre sur la broche X1 du multiplexeur, *Vout* le signal à l'émetteur du transistor (*Vout* rentre dans le sommateur) et *Vref* le deuxième signal rentrant dans le sommateur.

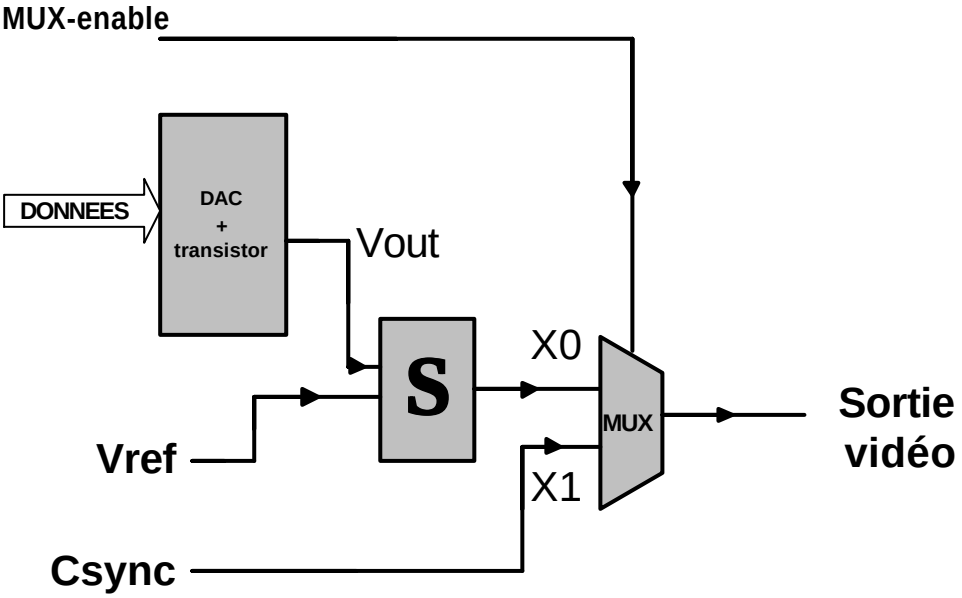


Figure 11 : Etage de sortie vers le moniteur de contrôle

On a alors, avec les notations du schéma :

$$Vref = \frac{-5}{1+\frac{RV5}{R9}}, \quad X1 = \frac{Csync}{1+\frac{R11}{RV4}}, \quad X0 = \left(1+\frac{RV2}{R10}\right)\left(\frac{Vref \times RV3 + Vout \times RV1}{RV3 + RV1}\right)$$

On voit donc bien avec ces formules l'influence des résistances variables sur le signal vidéo de sortie.

On va prendre RV3 = RV1 et RV2 = R10 de manière à avoir $X0 = Vref + Vout$.
Après une première phase de test, nous avons choisi comme valeurs optimales :

$$RV1 = RV3 = 10K\Omega$$
$$RV2 = R10 = 100K\Omega$$

RV1,2,3 sont des résistances variables ce qui laisse la possibilité de les ajuster au mieux lors de la phase de mise au point.

Le signal Vout est compris entre 3V et 4,4V (voir schéma) ; le but est de le ramener entre 0,6V et 2V avant de l'envoyer sur le multiplexeur. Donc Vref, le signal négatif que l'on ajoute à Vout, doit être de 2,4V d'où (comme R9 = 1,5K) la valeur de RV5 = 1,6K. Nous avons ici aussi prévu une résistance variable de 2,2K.

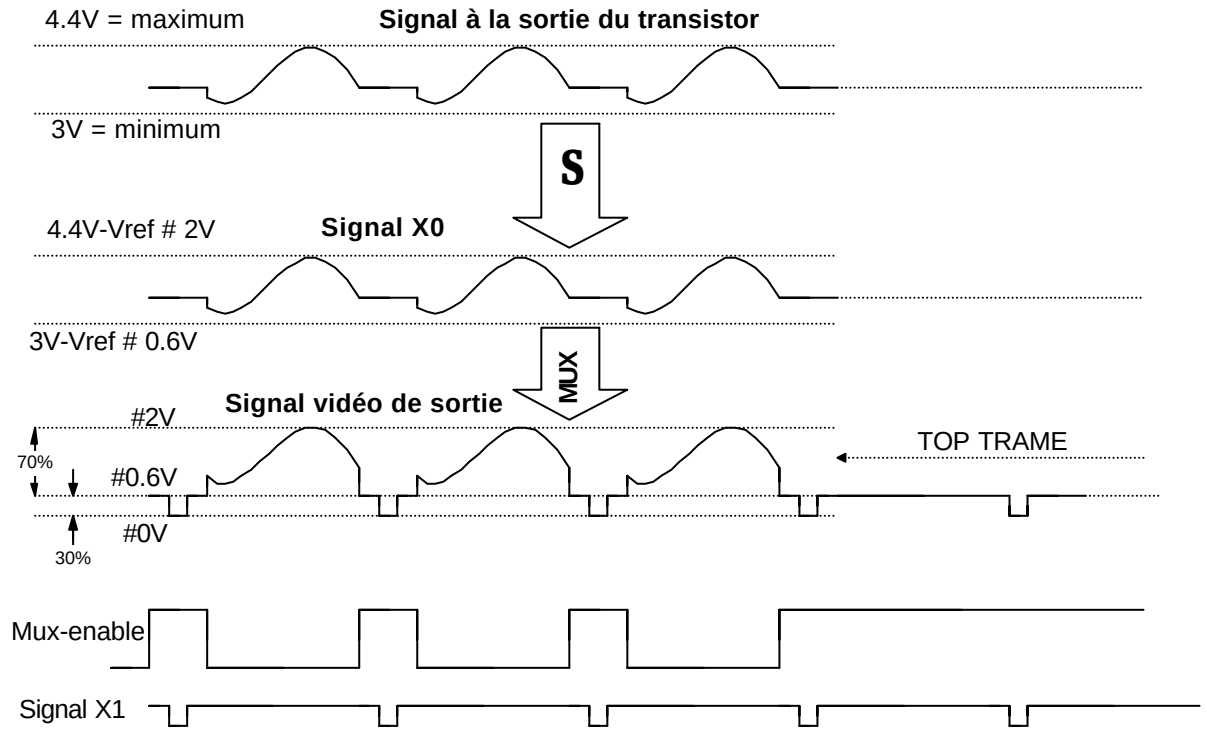


Figure 12 : Restitution du signal vidéo

Le signal X1 doit être compris entre 0 et 0,6V d'où le choix de R11=10K et RV4 = 1,3K. Ainsi le multiplexage des signaux X0 et X1 donnera bien un signal vidéo compris entre 0 et 2V avec des tops de synchronisations compris entre 0 et 0,6V.

Le signal de multiplexage est fourni par le circuit Xilinx RAM_CONTROL. Il est fait de telle manière qu'il laisse passer le signal X0 (niveau 0) pendant le blanking des lignes valides. Sinon, il est à 1 et laisse donc passer le signal de synchronisation X1.

Le signal résultant du multiplexage peut-être envoyé directement sur le moniteur.

3.2. PARTIE NUMERIQUE :

Plusieurs méthodes ont été étudiées pour permettre un traitement pseudo temps réel de l'image. Il a été choisi de concevoir la partie traitement asynchrone par rapport à l'acquisition/restitution du signal vidéo. Le seul problème consiste à réguler les accès aux mémoires pour éviter les conflits entre les parties traitement et acquisition, ce qui sera fait dans le composant RAM_CONTROL.

Nous allons d'abord étudier le circuit CALCUL chargé du traitement de l'image, puis le circuit ram_CONTROL chargé de réguler les accès aux mémoires, avant d'aborder l'aspect synthèse et simulation.

3.2.1. ETUDE DU CIRCUIT CALCUL:

Ce circuit est composé de 2 composants vhdl : *fifo* et *calcul*. Dans le cas de téléchargement de nouveaux algorithmes sur la carte, seul ce circuit sera modifié. Pour le calcul d'un pixel, 9 pixels sont nécessaires ; c'est pourquoi une mémoire tampon de 9 octets a été implantée dans ce circuit.

Il a été choisi de choisir une structure type fifo pour cette mémoire. On évite ainsi d'utiliser un compteur d'adresses supplémentaire. De plus, ce composant ayant été réalisé et testé lors d'un projet de 2ème année, cela a simplifié la conception du circuit. Seules quelques modifications ont été nécessaires pour l'adapter à notre cas.

L'échantillonnage des données *pixel_in[7 :0]* dans la mémoire fifo se fait sur front montant de l'horloge *clock*, alors que le signal *write_fifo* est à l'état haut. Dès que les 9 pixels ont été stockés dans la mémoire, le signal *rdy* passe alors à l'état haut. Les signaux *empty_fifo* et *empty_col* sont utilisés pour effacer soit toute la mémoire au début de chaque colonne de l'image d'entrée, soit les 3 premiers pixels échantillonnés.

Dès que le signal *rdy* passe à l'état haut, le calcul peut commencer. Dans notre application, il est possible de choisir entre 3 algorithmes de calcul différents : recopie de l'image, détection de contours et le filtre *emboss*. La machine d'état implantée, active sur front descendant de l'horloge *clock*, dispose des 3 états suivants :

- Etat ***attente*** : le but est d'attendre que les 9 pixels soient disponibles en entrée avant de commencer le calcul. C'est pendant cet état que le vidage de la mémoire fifo est effectué. Dès que le signal *debut_calcul* passe à l'état haut, la machine d'état passe alors à l'état ***calcul***.

- Etat ***calcul*** : l'algorithme de calcul est choisi avec les bits *sel[1 :0]*.

L'image résultat B(i, j) est une combinaison linéaire des neufs pixels de départ. Le filtre peut être exprimé par la matrice des coefficients de pondération :

$$F = \begin{bmatrix} a & b & c \\ d & e & f \\ j & g & h \end{bmatrix} = \begin{bmatrix} K(i-1, j-1) & K(i, j-1) & K(i+1, j-1) \\ K(i-1, j) & K(i, j) & K(i+1, j) \\ K(i-1, j+1) & K(i, j+1) & K(i+1, j+1) \end{bmatrix}$$

soit

$$B(i, j) = \sum_{-1}^{1} \sum_{-1}^{1} K(i+k, j+l).A(i+k, j+l)$$

Si le filtre F a tendance à ajouter les voisins au pixel B(i, j) il y a moyennage et F est du type passe-bas. Au contraire, si F a tendance à soustraire les pixels voisins, il y a dérivation et F est du type passe-haut.

Exemples de filtres :

MOYENNE

$$F = \left(\frac{1}{9}\right) \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

GAUSSIEN

$$F = \left(\frac{1}{16}\right) \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

GRADIENT VERTICAL

$$F = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Dans le cas de la recopie de l'image, le pixel de sortie est égal au pixel K(i, j). Pour les filtres de détection de contours et *emboss*, le calcul s'effectue avec des opérandes type entier avant d'être converti en std_logic_vector. Tous les calculs s'effectuent en un cycle d'horloge. Ensuite, la machine d'état passe à l'état ***fin_calcul***.

- Etat ***fin_calcul*** : si le calcul s'effectue pendant l'échantillonnage d'une ligne vidéo, il est nécessaire d'attendre que le composant CALCUL ait accès à la mémoire de sortie pour pouvoir écrire le résultat, soit lorsque *conv_enable* passe à l'état bas.

3.2.2. ETUDE DU CIRCUIT RAM_CONTROL:

Ce circuit est composé des 6 composants vhdl : *adc_control, compteur_colonne, compteur_ligne, compteur_entree, compteur_sortie* et *mux*, et réalise leur interconnexion (cf.

schéma généré par Renoir). Il gère l'acquisition et la restitution du signal vidéo, et régule l'accès aux mémoires entre les convertisseurs et le circuit XC4006e **CALCUL**.

a) Etude du composant *adc_control* :

Cette entité est utilisée pour générer les signaux de contrôle du convertisseur ADC TDA8708A. En effet, ce circuit a besoin des deux signaux Gate A (Sync level sync pulse) et Gate B (black level sync pulse) pour sélectionner le mode de fonctionnement.

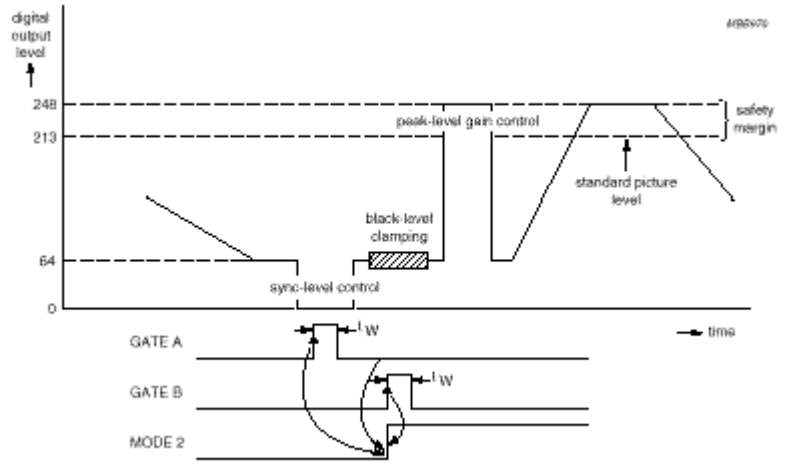


Figure 12 : Mode de contrôle 2

L'horloge *clk_pixel* cadencée à 10 MHz nous permet de générer les signaux pendant une durée déterminée, sachant que la durée minimum des impulsions est $t_w = 2\ \mu s$.

b) Etude du composant *compteur_colonne* :

Cette entité est utilisée pour n'autoriser l'échantillonnage du signal vidéo que pendant la phase de blanking. Pendant cette phase, le signal *pixel_valid* est alors mis à l'état haut. Elle intègre le compteur *add_low* qui génère les adresses poids faible des mémoires pendant la phase de conversion. Ce compteur est remis à zéro à chaque impulsion de synchronisation horizontale (csync).

c) Etude du composant *compteur_ligne* :

Cette entité intègre deux processus : l'un permet de générer le signal de parité *odd_even* qui change d'état à chaque front descendant de l'impulsion de synchronisation verticale (vsync), l'autre intègre le compteur *add_high* qui génère les adresses poids fort des mémoires pendant la phase de conversion. Seules les lignes 30 à 285 de chaque trame vidéo sont considérées valides pour l'échantillonnage du signal vidéo (le signal *ligne_valid* est alors mis à l'état haut).

d) Etude du composant *compteur_entree* :

Cette entité intègre le compteur d'entrée utilisé pour lire les données dans la mémoire d'entrée pour les stocker dans la mémoire fifo. Ce compteur est scindé en deux compteurs correspondant aux coordonnées du pixel dans la mémoire, la coordonnée (x) générée par le compteur 'entree_colonne', la coordonnée (y) par le compteur 'entree_ligne'.

D'après la méthode de lecture de la matrice correspondant à l'image d'entrée, 3 pixels sont d'abord lus sur la ligne courante, puis le compteur ligne est incrémenté. Au début de chaque colonne, il est nécessaire de lire 9 pixels pour commencer le calcul. Pour le calcul suivant, seuls 3 pixels sont lus puisque les 6 derniers pixels lus précédemment sont utilisés également dans ce calcul. Cette méthode permet de réduire considérablement le nombre d'accès à la mémoire et ainsi d'augmenter la rapidité du système puisque chaque cycle de lecture d'un pixel prend un cycle d'horloge. Le nombre d'accès mémoire en entrée est donc de : 510 * 9 + 509 * 510 * 3 accès mémoires. Tout est résumé sur la figure page suivante.

Le compteur colonne est composé de deux compteurs : *compt_pixel* et *compteur*. Le compteur *compt_pixel* est incrémenté à chaque front montant de l'horloge si le signal *conv_enable* est à l'état bas. Le signal d'écriture dans la mémoire fifo *wr_fifo* est à l'état actif dès que la machine d'état associée à ce compteur n'est plus dans l'état de repos *etat_fin_lecture*. Dès la fin de la lecture du 3ème pixel, le signal *fin_lecture* est mis à l'état haut, ce qui provoque l'incrémentation du compteur ligne.

A la fin de chaque colonne, le compteur *compteur* est incrémenté, jusqu'à atteindre la dernière colonne valide de la matrice d'entrée. A ce moment-là, l'image entière a été traitée, et on recommence un cycle complet de traitement.

e) Etude du composant *compteur_sortie* :

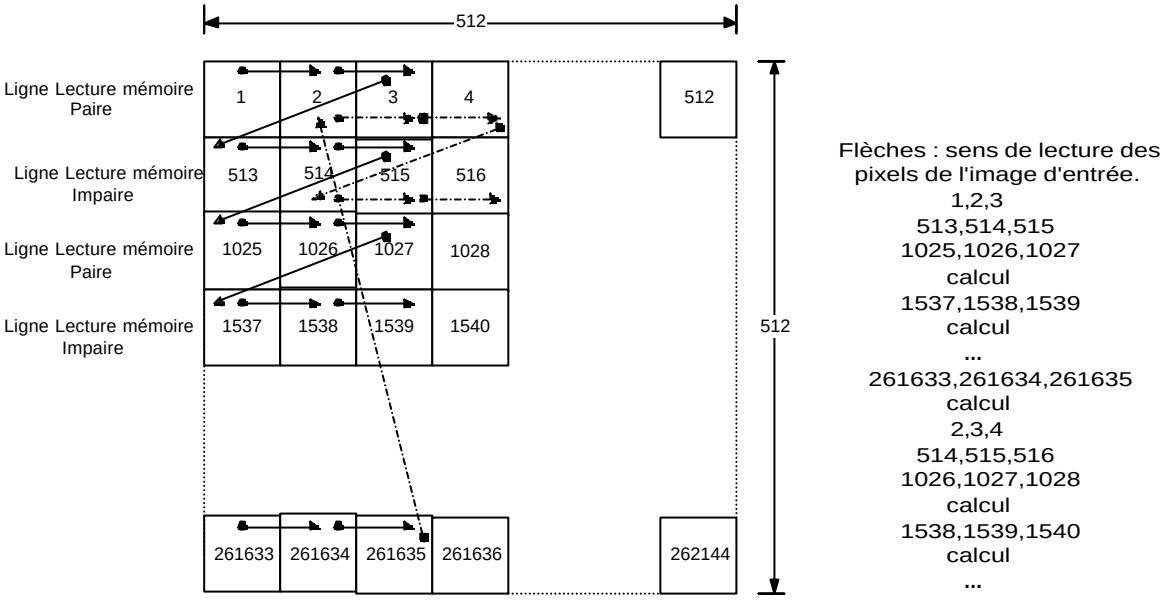
Cette entité intègre le compteur de sortie utilisé pour écrire les résultats des calculs dans la mémoire de sortie. Ce compteur est scindé en deux compteurs correspondant aux coordonnées du pixel dans la mémoire, la coordonnée (x) générée par le compteur 'sortie_colonne', la coordonnée (y) par le compteur 'sortie_ligne'.

Dès la fin d'un calcul, le signal *enable_write* (= *fin_calcul*) étant mis à l'état haut, la machine d'état passe dans l'état écriture puis le compteur est incrémenté pour stocker le prochain résultat de calcul. A chaque fin de colonne, le compteur colonne est alors incrémenté.

f) Etude du composant *mux* :

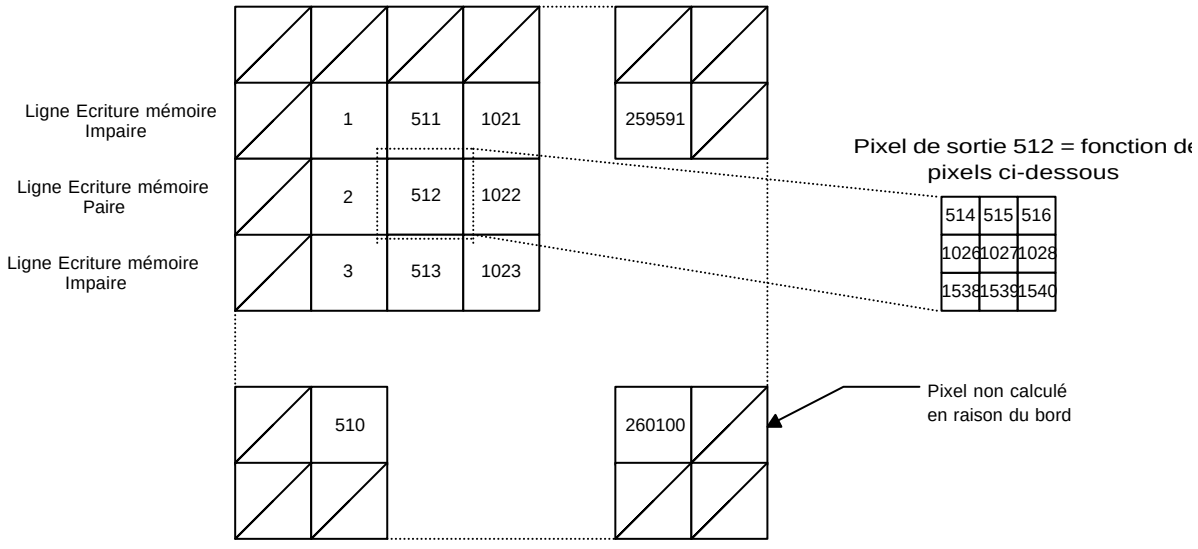
Cette entité effectue le multiplexage des bus d'adresses des mémoires. Ces bus sont partagés entre les compteurs associés à l'entité de calcul et les compteurs d'adresses utilisés lors de la phase de conversion. Le signal *conv_enable* est utilisé pour gérer l'accès aux bus de données des mémoires entre les convertisseurs (CNA et CAN) et les composants programmables.

LECTURE DE L'IMAGE D'ENTREE



Nombre d'accès mémoire = 510*9 + 509*510*3

ECRITURE DE L'IMAGE DE SORTIE



Nombre de calculs = nombre d'écriture mémoire = 260100 = 510*510

Numérotation 1,2,3... ordre de calcul et d'écriture des pixels dans la mémoire de sortie.

Figure 13 : Lecture et écriture dans les mémoires

3.2.3. SYNTHESE DES CIRCUITS PROGRAMMABLES :

Après la simulation comportementale, le fonctionnement des circuits couplés correspondait à notre attente. Nous allons d'abord voir la synthèse des deux circuits avant d'approfondir l'aspect simulation ensuite puisqu'alors interviendront les retards. Etant donné la fréquence d'horloge élevée du signal *clock* et le fait que notre design utilise les deux fronts de l'horloge, les retards dus aux composants programmables sont alors critiques.

a) Synthèse du circuit CALCUL :

La synthèse s'effectue avec Galileo. Au niveau du design d'entrée, on spécifie comme fichier **top.vhd**, au format VHDL. Dans la boîte de dialogue « VHDL Options », on spécifie comme *TOP ENTITY* : **top** et comme *ARCHITECTURE* : **struct**. Au niveau du design de sortie, on choisit le format XNF et comme technologie : XC4000E.

On spécifie également un fichier de contraintes pour forcer le routage au niveau des broches du Xilinx. Voici le résultat de la synthèse par Galileo :

```
Start optimization for design .work.top.struct
Using default wire table: 4025e-3
Pass     Area    Delay     DFFs  PIs   POs --CPU--
         (FGs)    (ns)                    min:sec
 1        224       84      92    15     9   00:29
Info, Added global buffer BUFG for port clock
```

On lance ensuite Design Manager pour obtenir l'occupation du composant dans un circuit X4006e, en choisissant comme format de sortie pour la simulation le VHDL. Le placement - routage nous donne les résultats suivants :

```
Number of External IOBs            23 out of 61     37%
      Flops:                           8
      Latches:                         0
Number of Global Buffer IOBs        1 out of 8      12%
      Flops:                           0
      Latches:                         0

Number of CLBs                    151 out of 256    58%
   Total CLB Flops:                84 out of 512    16%
   4 input LUTs:                  229 out of 512    44%
   3 input LUTs:                   40 out of 256    15%

Number of PRI-CLKs                   1 out of 4      25%
```

L'important est d'avoir une estimation de la taille en FGs occupée par chaque partie du composant avec Galileo.

	Nombre de FGs	Pourcentage utilisé
fifo	20	10%

automate de traitement	5	2%
algorithme de gradient	132	59%
algorithme *'emboss'*	65	29%

Quand on compare par rapport aux 1036 FGs du design généré par Mistral 2 pour le simple algorithme du gradient, il est clair que l'implémentation directement en VHDL de l'algorithme du gradient est plus efficace. Etant donné que le circuit n'est occupé qu'à 58% actuellement, il reste de la place pour ajouter de nouveaux algorithmes.

b) Synthèse du circuit RAM_CONTROL :

La synthèse s'effectue avec Galileo. Au niveau du design d'entrée, on spécifie comme fichier **ram_control.vhd**, au format VHDL. Dans la boîte de dialogue « VHDL Options », on spécifie comme *TOP ENTITY* : **ram_control** et comme *ARCHITECTURE* : **struct**. Au niveau du design de sortie, on choisit le format XNF et comme technologie : XC4000E.

On spécifie également un fichier de contraintes pour forcer le routage au niveau des broches du Xilinx.

Voici le résultat de la synthèse par Galileo :

```
-- Start optimization for design .work.ram_control.struct
Using default wire table: 4025e-3

      Pass     Area    Delay     DFFs  PIs   POs --CPU--
               (FGs)    (ns)                  min:sec
      1        184       33        98     7    51   00:52
Info, Added global buffer BUFG for port clk_pixel
Info, Added global buffer BUFG for port clock
Info, Added global buffer BUFG for port csync
```

On lance ensuite Design Manager pour obtenir l'occupation du composant dans un circuit X4006e, en choisissant comme format de sortie pour la simulation le VHDL. Le placement - routage nous donne les résultats suivants :

```
Number of External IOBs            55 out of 61     90%
      Flops:                          2
      Latches:                        0
Number of Global Buffer IOBs        3 out of 8      37%
      Flops:                          0
      Latches:                        0

Number of CLBs                    101 out of 256    39%
   Total CLB Flops:                96 out of 512    18%
   4 input LUTs:                  186 out of 512    36%
   3 input LUTs:                   17 out of 256     6%
Number of PRI-CLKs                  2 out of 4      50%
Number of SEC-CLKs                  1 out of 4      25%
Number of STARTUPs                  1 out of 1     100%
```

3.2.4. SIMULATION DES CIRCUITS PROGRAMMABLES :

Après le placement - routage avec les options choisies, les fichiers time_sim.vhd et time_sim.sdf ont été créés dans le répertoire associé au design.

Pour le composant CALCUL, puisque le signal *raz* n'est pas routé directement sur le net GSR, il est nécessaire de fixer la durée du GLOBAL RESET qui a été automatiquement inséré par un composant ROC, il faut éditer le fichier time_sim.vhd et mettre en commentaires les 4 lignes depuis...

```
configuration CFG_ROC_V of ROC is  ... end CFG_ROC_V ;
```

et les remplacer par la ligne suivante (Dans l'architecture STRUCTURE OF TOP, après la déclaration de COMPONENT ROC):

```
FOR ALL:ROC USE ENTITY work.roc(roc_v) GENERIC MAP(width=>100 ns);
```

Il faut rajouter la bibliothèque SIMPRIM par la commande :

```
qhmap simprim /net/xilinx/mentor/data/vhdl/simprim
```

Pour tester les composants CALCUL et RAM_CONTROL, on utilise une entité de test en VHDL. Cette entité (dans le fichier *test_mix.vhdl*) effectue le test des deux entités interconnectées comme le seront les deux composants programmables sur la carte. Le process 'test_mix ' génère les signaux de synchronisation en respectant les caractéristiques du signal vidéo.

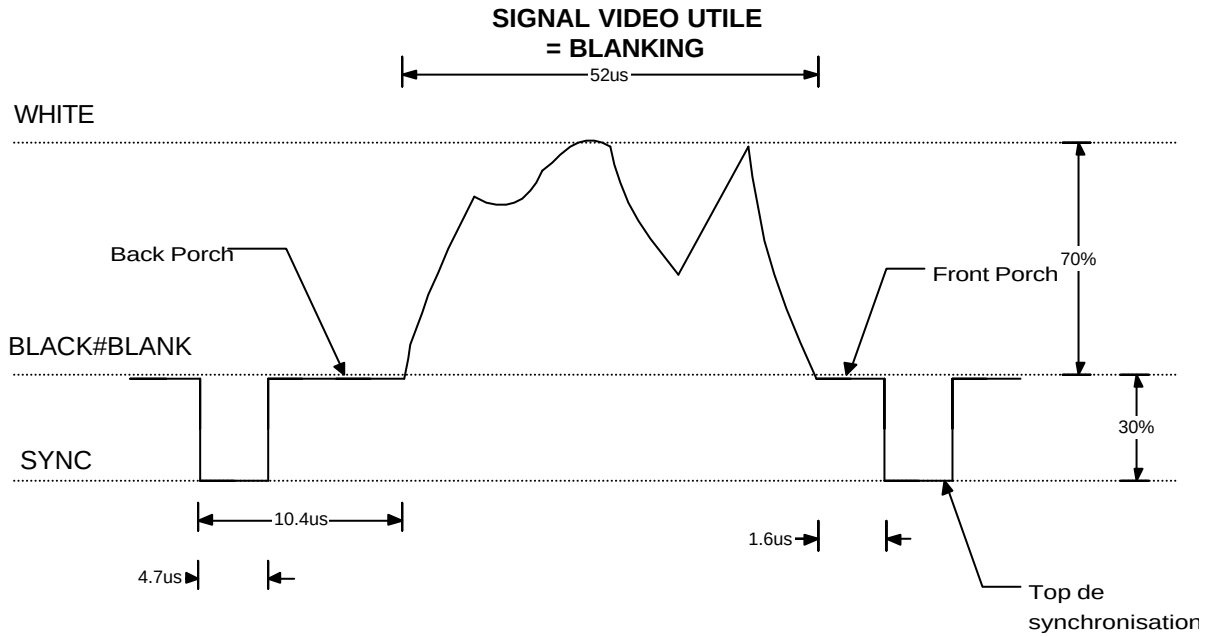


Figure 14 : Caractéristiques du signal vidéo

On a ainsi défini les temps Tp_csync = 4.7 µs, Tp_vsync = 2 µs, Tp_burst = 2.4 µs et Tp_ligne = 52 µs.

Afin d'obtenir des vecteurs de sortie facilement interprétables, un fichier texte contenant l'image d'entrée '*entree.pgm*' est lu par le process *lecture* pour appliquer les données sur le bus *pixel_in* du composant CALCUL. Les données sur le bus *pixel_out* sont sauvegardées ensuite dans un fichier de sortie '*sortie.pgm*' à chaque impulsion d'écriture dans la mémoire de sortie.

Pour obtenir le fichier '*entree.pgm*', il faut une image au format PGM de dimensions 512x512. Une méthode simple pour obtenir un tel fichier est d'utiliser XV. En appuyant sur la touche 'S' en majuscule, on peut alors imposer la taille de l'image (512x512). Ensuite, dans les options de sauvegarde de l'image, on choisit :

```
Format : PBM/PGM/PPM (Ascii)
```

```
Colors : Grayscale
```

Le programme de conversion écrit en C (*projet.c*) permet alors de convertir ce fichier image créé avec XV dans un format utile pour la simulation. Toutes les indications pour compiler ce programme en C sont écrites dans le code. La conversion du format PGM au format de simulation est effectuée en tapant la commande :

```
conv_to_simu image.pgm
```

On dispose alors du fichier entree.pgm d'une taille de 2304943 octets. Ce fichier devra être placé dans le répertoire d'où est lancée la simulation, avec l'option de prise en compte des retards :

```
qhsim -sdftyp /u1=time_sim_top.sdf -sdftyp /u0=time_sim_ram.sdf test_mix
```

Après chargement du design, il faut changer la durée de la simulation dans le menu Options → Properties → Default Run. On effectuera la simulation pendant une durée de 65331575 ns. La simulation avec retards complète dure environ 15h. On obtient ensuite le fichier 'sortie.pgm' que l'on convertira vers un format reconnu par XV :

```
simu_to_img image.pgm
```

Voici le fichier image généré après la simulation pour le filtre de détection de contours :

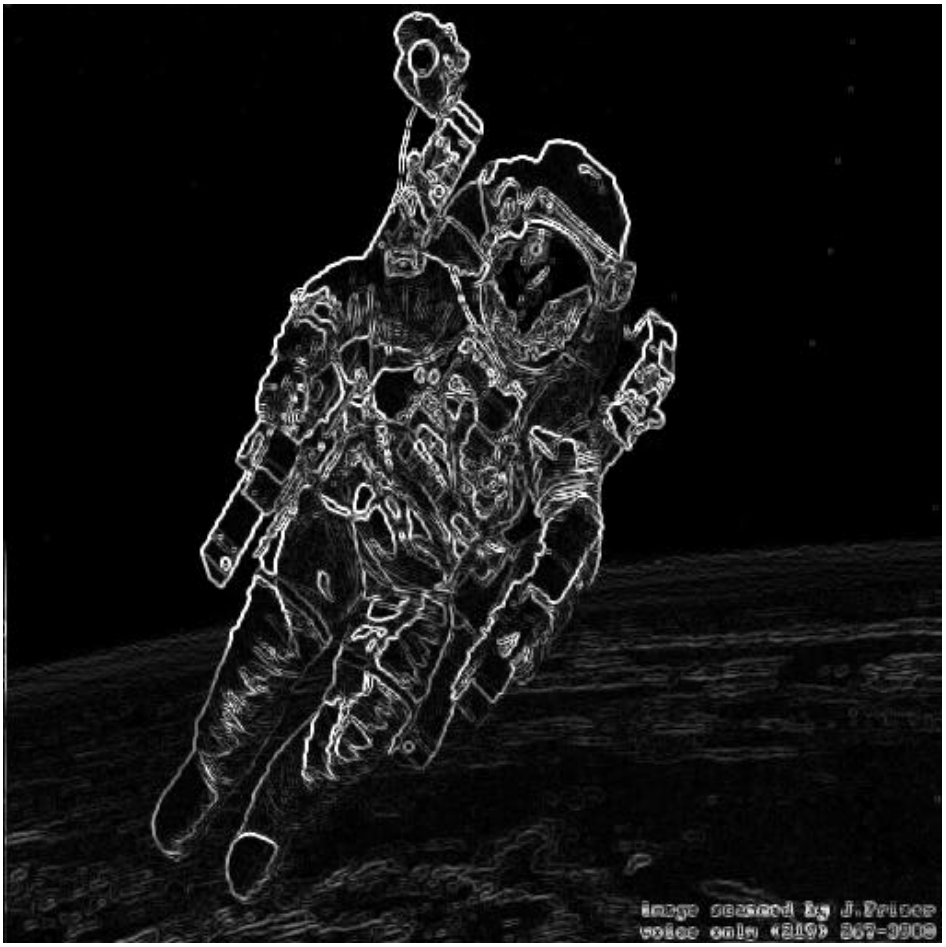


Figure 15 : Résultat de la simulation avec retards

4. CONCLUSION :

4.1. ETAT D'AVANCEMENT DU PROJET :

La partie la plus importante du projet a été consacrée à l'étude et au choix de l'architecture de la carte. Après la synthèse par Mistral 2 du code DFL, le choix a été pris de coder directement en VHDL afin de valider la carte. Pour les algorithmes bas niveau implantés, cela n'a guère posé de problème. Pour des algorithmes plus complexes, il serait intéressant de valider la chaîne DDsim → Mistral 2 afin de faciliter la tâche du concepteur.

Ensuite, la partie analogique d'acquisition/restitution du signal vidéo a été testée. Au niveau simulation, les deux circuits programmables CALCUL et RAM_CONTROL ont été validés. Après routage et réalisation de la plaque, il nous reste à l'équiper et à la tester, ce qui devrait être possible d'ici la fin du mois de juin.

4.2. AMELIORATIONS POSSIBLES :

Une interface avec un ordinateur hôte serait intéressante. Le gestion de l'interface est alors prise en charge par un microcontrôleur type 68HC11. On disposera alors de plusieurs modes de fonctionnement :

- sélection de la fréquence d'échantillonnage : 5 ou 10 MHz, en modifiant la valeur d'un registre interne au composant programmable,
- télécharger dans la mémoire d'entrée une image stockée sur l'ordinateur hôte (pour le test ou le traitement d'une image) ou de la carte vers l'ordinateur (pour récupérer une image acquise),
- télécharger dans la mémoire de sortie une image stockée sur l'ordinateur (pour visualiser sur le moniteur de contrôle) ou de la carte vers l'ordinateur (pour récupérer le résultat du traitement d'une image en mode test par exemple),
- téléchargement des algorithmes de traitement dans la mémoire de configuration du composant programmable...

Afin d'accélérer le temps de traitement, on pourrait prévoir plusieurs unités de traitement en parallèle, ce qui rendrait possible le traitement d'images couleur.

5. REFERENCES BIBLIOGRAPHIQUES :

LIVRES TECHNIQUES

- R. Besson, *Cours de Télévision Moderne*, Editions Radio
- Vincent FRISTOT, *Carte de traitement d'images en temps réel*, Laboratoire de Traitement d'Images et de Reconnaissances des Formes
- Mentor Graphics Corporation, *DSP Architect - Introduction to DFL Training Workbook*
- Mentor Graphics Corporation, *DSP Station - Mistral 2 User's and Référence Manual*
- Patrice NOUEL, *Implantation Xilinx d'un circuit décrit en VHDL*
- Electronique Radio Plans , *Carte universelle à MC 68HC11 A1*, Numéro 562 Septembre 1994

REFERENCES INTERNET

- CircuitWorld Online Services, *http://www.circuitworld.com/*
- Chip Directory, *http://www.chipdir.com*
- WEBLinx, *http://www.xilinx.com*

ANNEXES

- Sources DFL de l'algorithme de détection de contours

  Le programme principal image_demo.dfl, aprés la lecture de l'image, applique le filtre de détection de contours, et sauvegarde l'image sur disque.

- Programme en C (test.c) pour le test de l'algorithme de détection de contours

  Ce programme a été développé pour tester la lecture/écriture des fichiers au format PGM, ainsi que l'algorithme de détection de contours.

- Schémas générés par Renoir HDL2Graphics

  Ces schémas ont été générés avec Renoir pour obtenir une représenation graphique du design VHDL.

- Sources VHDL des circuits CALCUL et RAM_CONTROL

  Cet ensemble de fichiers VHDL représente le contenu des circuits programmbles CALCUL et RAM_CONTROL.

- Simulation avec retards des circuits programmables

  Cet ensemble de quatre simulations réalisées sous Qhsim représentent les points importants du fonctionnement du système: initialisation, fin de colonne, acquisition d'une ligne vidéo...

- Utilitaire programmé en C (projet.c) pour la conversion de fichiers simulation

  Ce programme permet d'obtenir les programmes conv_to_simu et simu_to_img nécessaires á l'exploitation du résultat des simulations.

- Schéma de la carte d'acquisition vidéo

  Le schéma complet est composé de trois feuilles: acquisition du signal vidéo, traitement et restitution du signal vidéo.

- Plan d'équipement de la carte (typons inclus)

  Cette partie inclut le plan d'équipement de la carte, ainsi que les typons.