



Vision Stéréoscopique

Mémoire de diplôme

Projet de diplôme basé sur
les systèmes numériques embarqués



Projet nommé Vision Stéréoscopique **suivi par l'étudiant** David FISCHER **de la filière des** télécommunications **classe** TE3, **ceci dans le laboratoire de** systèmes numériques **encadré par** Monsieur Jacques TINEMBART **de** mi-septembre **à fin** novembre 2006.

Table des matières

| | |
|--|----|
| Table des matières..... | 2 |
| 1. Cahier des charges..... | 3 |
| 2. Principaux termes..... | 4 |
| 3. Introduction..... | 5 |
| 3.1 Brève description..... | 5 |
| 3.2 Objectif du projet..... | 5 |
| 3.3 Plan de travail..... | 6 |
| 3.4 Connaissances accumulées..... | 6 |
| 4. Etude préliminaire..... | 7 |
| 4.1 Vue globale..... | 7 |
| 4.2 Outils de travail..... | 8 |
| 4.2.1 Matériel à disposition..... | 8 |
| 4.2.2 Environnement logiciel..... | 9 |
| 4.2.3 La carte Cyclone USB2..... | 10 |
| 5. Principe de fonctionnement..... | 11 |
| 5.1 Etapes du traitement..... | 11 |
| 5.2 Contrôle global et positionnement de la pupille..... | 11 |
| 5.3 Introduction, principe de la FPGA..... | 12 |
| 5.4 Le langage de description matériel..... | 13 |
| 5.5 ZonesBayer & MoyenneurBayer..... | 14 |
| 5.5.1 Introduction, la caméra CMOS..... | 14 |
| 5.5.2 But du système..... | 15 |
| 5.5.3 Fonctionnement détaillé de ZonesBayer..... | 16 |
| 5.5.4 Fonctionnement détaillé de MoyenneurBayer..... | 17 |
| 5.6 Bayer_a_RVB..... | 19 |
| 5.6.1 But du système..... | 19 |
| 5.6.2 Fonctionnement détaillé..... | 19 |
| 5.7 RVB_a_HSL..... | 21 |
| 5.7.1 Introduction, l'espace HSL..... | 21 |
| 5.7.2 But du système..... | 21 |
| 5.7.3 L'arithmétique à virgule fixe..... | 23 |
| 5.7.4 Fonctionnement détaillé..... | 25 |
| 5.8 RVB_ou_HSL..... | 27 |
| 5.8.1 But du système..... | 27 |
| 5.8.2 Fonctionnement détaillé..... | 27 |
| 5.9 ProtocoleTITI..... | 28 |
| 5.9.1 Introduction, le protocole Titi..... | 28 |
| 5.9.2 But du système..... | 28 |
| 5.9.3 Fonctionnement détaillé..... | 29 |
| 5.10 InterfaceFX2..... | 30 |
| 5.10.1 Introduction, le microcontrôleur FX2..... | 30 |
| 5.10.2 But du système..... | 30 |
| 5.10.3 Fonctionnement détaillé..... | 30 |
| 5.11 Pupille..... | 31 |
| 5.11.1 Introduction..... | 31 |
| 5.11.2 But du système..... | 31 |
| 5.11.3 Fonctionnement détaillé..... | 31 |
| 5.12 Contrôle..... | 32 |
| 5.12.1 Introduction, la configuration des caméras..... | 32 |
| 5.12.2 But du système..... | 32 |
| 5.13 Divers Modules Utiles..... | 33 |
| 5.14 Etude des capacités limites de notre système..... | 34 |
| 6. Firmware du FX2..... | 37 |
| 6.1 Introduction, notre configuration..... | 37 |
| 7. Interface PC..... | 38 |
| 7.1 Introduction, CypressEzUSB..... | 38 |
| 7.2 Introduction, DirectDraw..... | 38 |
| 7.3 But du logiciel..... | 39 |
| 8. Développements futurs..... | 40 |
| 8.1 L'interface parallèle - série envisagé..... | 40 |
| 9. Conclusion..... | 41 |
| 9.1 Ce qui a été réalisé..... | 41 |
| 9.2 Ce qui sera prochainement réalisé..... | 41 |
| 9.3 Perspectives d'avenir..... | 41 |
| 9.4 Remerciements..... | 41 |

1. Cahier des charges

e i gEcole d'ingénieurs
de Genève**Automne 2006****Session de diplôme****SYSTEMES NUMERIQUES
VISION STEREOSCOPIQUE****Descriptif :**

Pour permettre à des personnes aveugles de s'orienter dans un environnement complexe, MM. Vinckenbosch et Bologna ont proposé un algorithme permettant d'associer des sons à des couleurs, substituant l'ouïe à la vision.

Pour tester d'autres algorithmes faisant intervenir la notion de profondeur, il est nécessaire de disposer d'un système de vision stéréoscopique. De plus, afin de limiter les informations à traiter, la détermination du centre de la pupille permet de définir un lieu privilégié de traitement. De cette manière, il est possible d'appliquer un algorithme de traitement différent entre la zone visée et la zone périphérique, à la manière d'un œil sain.

Travail demandé :

- Un système de capture d'images provenant d'une caméra CMOS (VHDL).
- Un module de conversion d'images du format RGB vers HSV (VHDL).
- Un algorithme de détection de la pupille en temps réel (VHDL).
- Un logiciel configurant le FX2 de Cypress de manière à permettre la transmission des coordonnées de la pupille ainsi que l'image du panorama visé. Un canal permettra la configuration du système par l'hôte.
- Une petite application de démonstration (par exemple : croix positionnant l'endroit visé par l'œil au milieu de l'image collectée.)
- Un rapport circonstancié ainsi qu'un CDROM contenant toutes les informations nécessaires à la reprise du projet par un tiers fournis en fin de projet. Les modules VHDL conçus devront être accompagnés de leurs testeurs respectifs.

Mots-clés :

Vision stéréoscopique, suivi du regard, USB 2.0, VHDL, HSV, RGB

Unité d'enseignement
Et de recherche UER4

Classe: TE3

Timbre de l'Ecole



Candidat :

M. FISCHER DAVID

Filière d'études :

TélécommunicationTravail de diplôme soumis à une
convention de stage en entreprise : nonTravail de diplôme soumis à un contrat
de confidentialité : non

Professeur(s) responsable(s) :

Tinembart Jacques

En collaboration avec : MM
Vinckenbosch & Bologna (Projet Hasler)

2. Principaux termes

Quelques termes théoriques

- Couleurs RVB** Façon de coder les couleurs, (de manière scientifique : la chrominance). Les lettres signifient simplement que nous codons, pour chaque pixel, la quantité de *rouge*, de *vert* et de *bleu*.
- Couleurs HSL** Autre manière de coder les couleurs. HSL veut dire, comme expliqué dans la partie théorique, que nous codons la quantité de teinte (*hue*), *saturation* et de *luminance*.
- Vision stéréoscopique** Dans la nature les prédateurs fixent une proie avec leurs deux yeux, ceux-ci étant légèrement décalés l'un par rapport à l'autre. Ceci leur permet d'estimer les distances par effet de parallaxe.

Quelques termes d'ingénieur

- Datasheet** Documentation permettant à un ingénieur de se faire une idée sur le fonctionnement d'un système (électronique en particulier).
- Caméra CMOS** Caméra usinée avec un capteur photographique de technologie CMOS, (*complementary metal oxide semi-conductor*)
- Mémoire FIFO** Entité mémoire disposant d'un port d'écriture et d'un port de lecture implémentant la politique du premier entré, premier sorti (du terme *first in → first out*). Principalement utilisé pour la gestion du contrôle de flux et de l'interfaçage entre logiques fonctionnant à différents rythmes d'horloge.
- Firmware FX2** En français « micro logiciel ». Nom que porte un programme s'exécutant sur un matériel embarqué disposant d'un microprocesseur prévu pour exécuter ce logiciel. Dans notre cas, le firmware du processeur USB2.
- FPGA** Désigne un circuit programmable configurable, composé d'opérateurs logiques combinatoires et de bascules synchrones, dans lequel la fonction à réaliser peut être implantée à partir d'un code (p.ex VHDL)
- I²C** Protocole de communication série développé par Philips. Le but d'origine était de relier l'ensemble des composants d'un appareil électronique (*Inter Integrated Circuit Bus*) sous la forme d'un bus.
- USB** Protocole de communication série développé par Intel, servant à relier des périphériques externes à un ordinateur (*universal serial bus*)
- VHDL** Langage de description matériel de haut niveau, très puissant !

3. Introduction

3.1 Brève description

Ce projet de diplôme s'est déroulé au sein du Laboratoire de Systèmes Numériques de l'Ecole d'Ingénieurs de Genève du 18 septembre au 5 décembre 2006 sous la direction de M. Jacques TINEMBART.

Comme indiqué dans le cahier des charges, le but est de permettre à des personnes aveugles de s'orienter dans un environnement complexe. Monsieur M. Vinckenbosch et M. Bologna ont proposé une manière de représenter l'information visuelle sous la forme de sons, substituant l'ouïe à la vision.

Dans ce cadre, je devrais concevoir la première ébauche d'un système embarqué permettant de concrétiser cette analyse théorique.

3.2 Objectif du projet

Etant donné que cette mise en œuvre prend beaucoup de temps, mon projet de diplôme se situera au niveau de l'implémentation matérielle des algorithmes de traitement de l'image et de la pupille, sous la forme d'un système embarqué raccordé en USB2.0 avec une interface logiciel PC.

Pour se faire mon travail débutera par une phase de mise en place du matériel et de l'environnement logiciel, suivie par une étude minimale des contraintes, une fois cette phase achevée je pourrai commencer par développer du code implantable dans le matériel mis à disposition, constitué d'une FPGA.

Dans un premier temps une démarche de test minimale (simulation) sera effectuée en continu sur les différents modules constituant la partie de l'algorithmique embarquée.

Pour me donner une idée des résultats attendus de la part des modules VHDL, je ferai sous forme logicielle (C++) l'implémentation de certains algorithmes de traitement. Cet « étalon » me permettra de, pour débiter, vérifier que les résultats sortant des simulations VHDL ne sont pas complètement invalides. Chacun des modules étant étudié pour tirer meilleur partie de l'architecture FPGA et d'être toujours dans les contraintes liées au matériel et au temps réel.

En parallèle je devrai prendre en main la configuration du matériel pour permettre à la carte de développement mise à disposition de fonctionner selon mon désir, de façon plus précise pour entre autre constituer un flux USB2.0 entre l'embarqué et le PC.

Une fois les premiers test d'interfaçage concluants, je continuerai par finaliser la mise en place des connections caméra, de la logique contrôlant le tout ainsi que la partie interface logiciel.

Ce projet sera réalisé d'une telle façon qu'il pourra être repris pour permettre à mon successeur de réaliser un système complet répondant aux besoins de l'application...



3.3 Plan de travail

Etudes & Développements

- Etude du matériel mis à disposition, dont caméras et carte Cyclone USB2
- Etude théorique des contraintes, ébauche des entités FPGA en VHDL
- Développement des étalons logiciel C++ pour certains algorithmes
- Développement des modules FPGA en VHDL, simulations
- Etude des algorithmes de traitement d'image pour positionner une pupille
- Etude de la façon de maîtriser les caméras CMOS (I²C)
- Etude du livre concernant le microcontrôleur USB2, le FX2 de Cypress

Buts poursuivis

- Développement d'un firmware pour le FX2 permettant la transmission des flux entre la carte et le PC, dans un sens comme dans l'autre (en mode Bulk)
- Développement d'une interface logicielle affichant les images et d'autres informations provenant de notre système et contrôlant celui-ci
- Développement de la partie de contrôle des caméras pas le bus I²C
- Interfaçage des vraies caméras avec la FPGA et acquisition des images provenant de celles-ci

Imprévus et démarche de réalisation finale

- [bug matériel] Modification du système afin de le décomposer en deux éléments distincts ; le premier s'occupe d'effectuer les différents traitements nécessaires afin de pouvoir, par la suite, facilement transformer une image en son et le deuxième s'occupe de la partie concernant le positionnement de la pupille.
- Maîtrise de l'I²C pour la caméra monochrome → C++ ARM-XILINX → VHDL FPGA
- [bug logiciel] Refonte complète et utilisation de DirectDraw pour la partie affichage temps réel des images provenant de la caméra → Logiciel
- [tests] Création d'une version hybride traitant les données de la caméra monochrome comme s'il s'agissait de données couleurs, dans le but de concevoir (tester) un logiciel d'affichage performant. Ceci m'a permis d'avancer sans avoir besoin de la caméra couleur → VHDL FPGA
- Maîtrise de l'I²C pour la caméra couleur → C++ ARM-XILINX → VHDL FPGA
- Modifications du firmware et du logiciel pour gérer l'envoi d'ordres au matériel

3.4 Connaissances accumulées

Afin de bien comprendre et approfondir les éléments mis en jeu dans ce projet, plein de liens intéressants sont à votre disposition, en plus des introductions descriptives se trouvant tout au long de ce mémoire et les documents fourni dans le CD du diplôme.

[..\\LIENS.html](#)

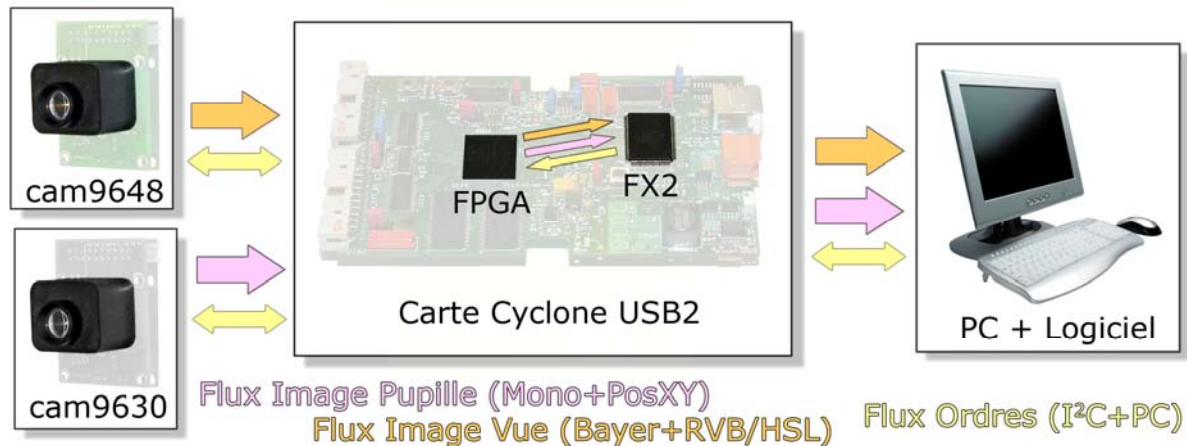
Finalement une liste non exhaustive des notions requises peut se résumer ainsi :

- Notions en rapport avec le hardware et software (programmation)
- Notions de VHDL / FPGA
- Notions de C++ / DirectDraw

4. Etude préliminaire

4.1 Vue globale

Le système se compose d'une caméra couleur et d'une caméra monochrome, toutes de technologie CMOS, reliées à une carte de développement Cyclone USB2. Celle-ci est constituée entre autres d'une FPGA qui accueillera nos modules et d'un microcontrôleur FX2 permettant de relier la carte au PC par l'intermédiaire d'un bus USB2.0.



Le système embarqué faisant office de calculateur, c'est lui qui gère les caméras et la liaison PC.

Finalement, le logiciel d'interface PC nous permet de contrôler notre système et de recevoir des informations comme l'image traitée et le positionnement de la pupille de la part de notre matériel.

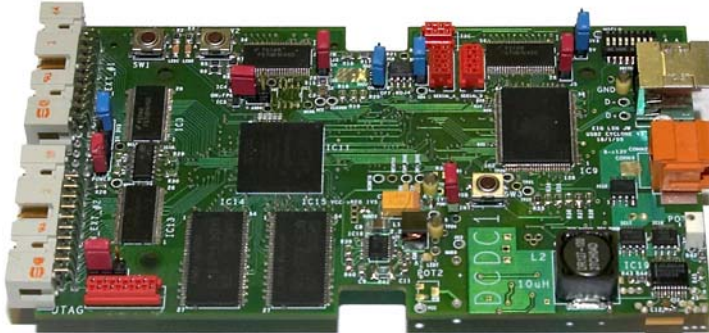
4.2 Outils de travail

4.2.1 Matériel à disposition



← Module Caméra **CAM9630** (aussi en CAM9648) [labo]

Petite carte permettant de brancher une caméra CMOS du type KAC-9630 (9648) de Kodak sur une carte de développement Cyclone USB2 ayant des connecteurs externes.



← Carte **Cyclone USB2** [labo]

Carte de laboratoire complète, permet de concevoir un système embarqué complexe intégrant une FPGA, de la RAM, un FX2, ainsi que divers connecteurs externes, I²C et USB2.



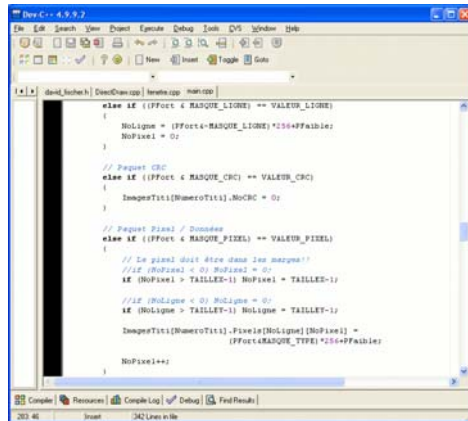
← Carte **ARM-XILINX** [labo]

Carte de laboratoire utilisée pendant l'année scolaire, elle permet de concevoir et tester de nombreuses applications embarquées, notamment dans mon cas, de tester la norme I²C.

Oscilloscope & source de tension (un classique)



4.2.2 Environnement logiciel



← DevC++ de Bloodshed

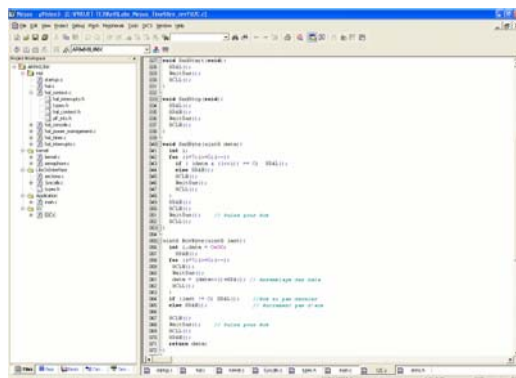
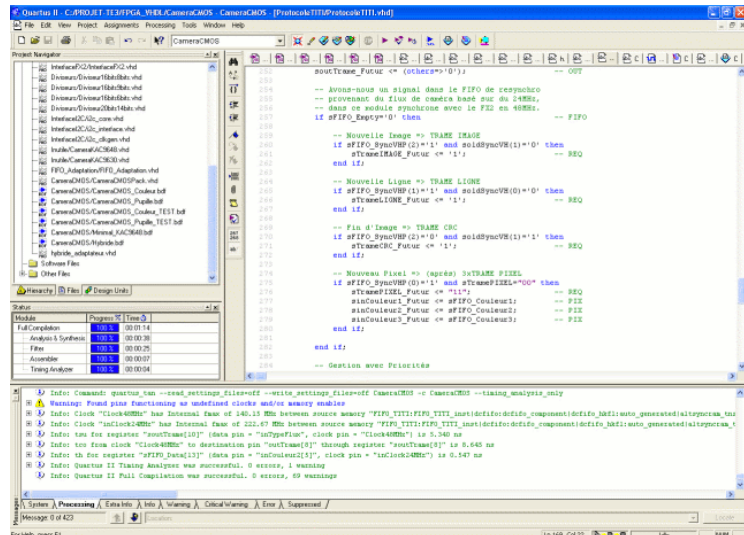
Environnement de développement C++ gratuit.

Me permet de compiler les implémentations logicielles (étalons) ainsi que d'éditer les codes C++ que je compilerais avec la librairie Qt.

Quartus II d'Altera →

Environnement complet de développement VHDL permettant de passer du code source à l'implémentation en FPGA.

Permet de nombreuses études comme par exemple la simulation temporelle ou encore la vision de l'utilisation de la FPGA au point de vue placement/routage.



← Keil hVision 2

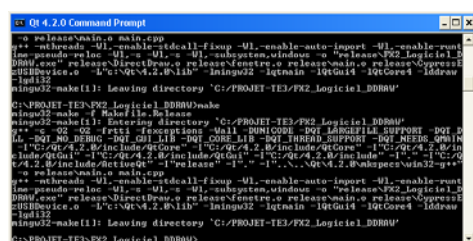
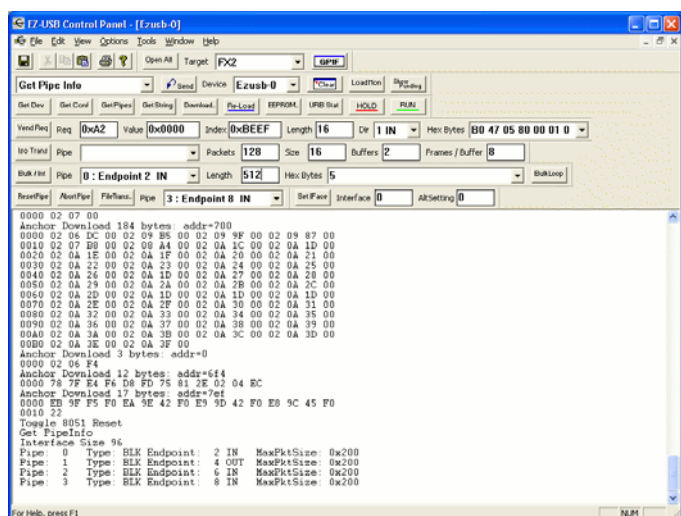
Environnement de développement spécifique à la programmation des microprocesseurs embarqués et permet notamment l'exécution pas à pas.

Utile pour l'implémentation du firmware FX2 et les premiers tests concernant l'I²C.

EZ-USB Control Panel →

Environnement de test concernant le FX2 de la compagnie Cypress.

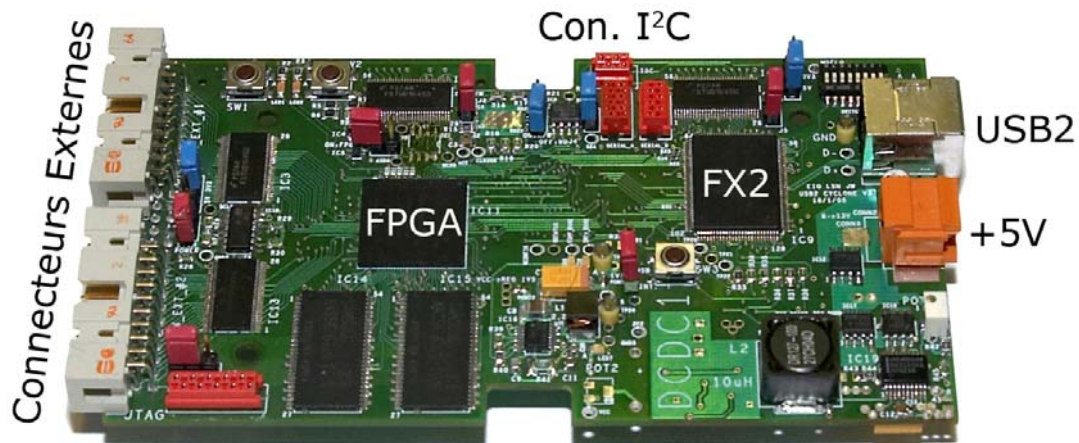
Permet notamment de télécharger le firmware dans le microcontrôleur ainsi que de vérifier / envoyer un flux dans les différents endpoint.



← Qt Command Prompt

Permet tout simplement (qmake -project / qmake / make) de compiler un projet C++ utilisant la librairie Qt.

4.2.3 La carte Cyclone USB2



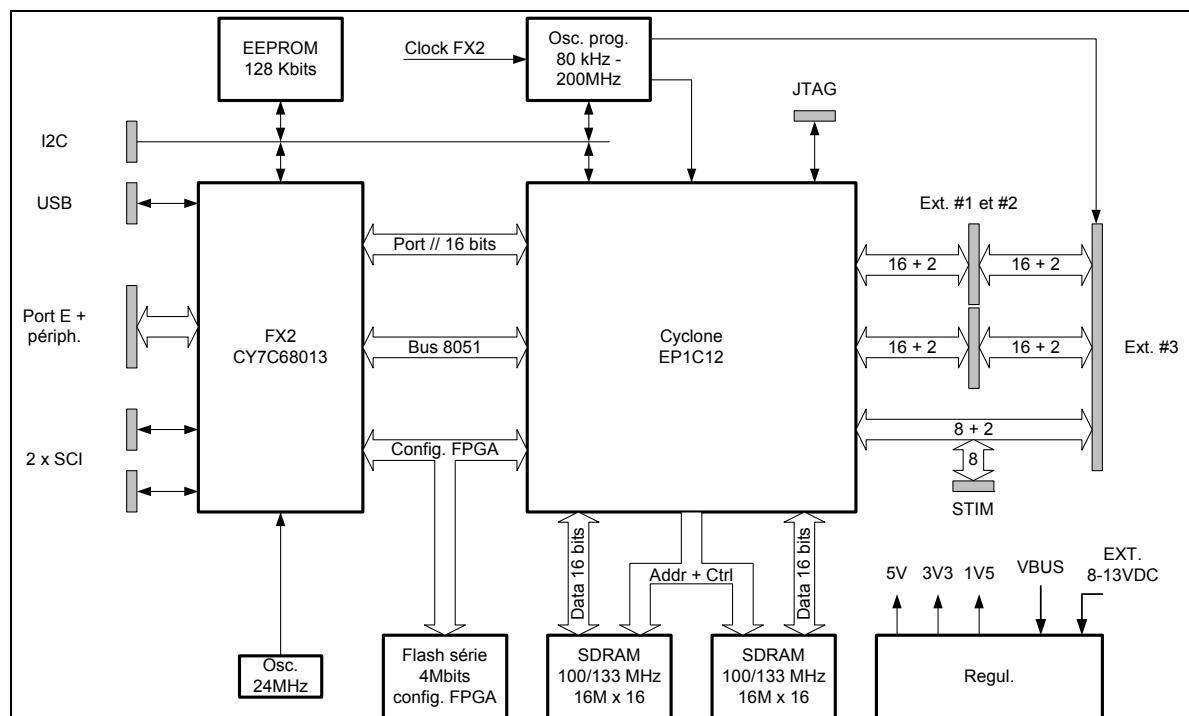
La carte USB2-Cyclone est un des nombreux développements du Laboratoire de Systèmes Numériques de l'Ecole d'Ingénieurs de Genève.

Cette plateforme de développement permet entre autres de concevoir un système embarqué qui peut être interfacé avec un ordinateur grâce à son connecteur USB2.

Quelques caractéristiques :

- Une FPGA d'Altera, modèle EP1C12F256C7 (voir chapitre dédié)
- Un microcontrôleur FX2 de Cypress (CY7C68013), permettant de relier notre système (à un ordinateur par exemple) par le biais de la norme USB2.
- 64 Mo de mémoire vive accessible par la FPGA
- ... Connecteur I²C, EEPROM ...
- Plusieurs broches de la FPGA sont directement reliées au FX2
- Les connecteurs externes sont accessibles depuis la FPGA

Ce qui s'est schématisé ainsi (source : usb2-cyclone-usrman....doc) :

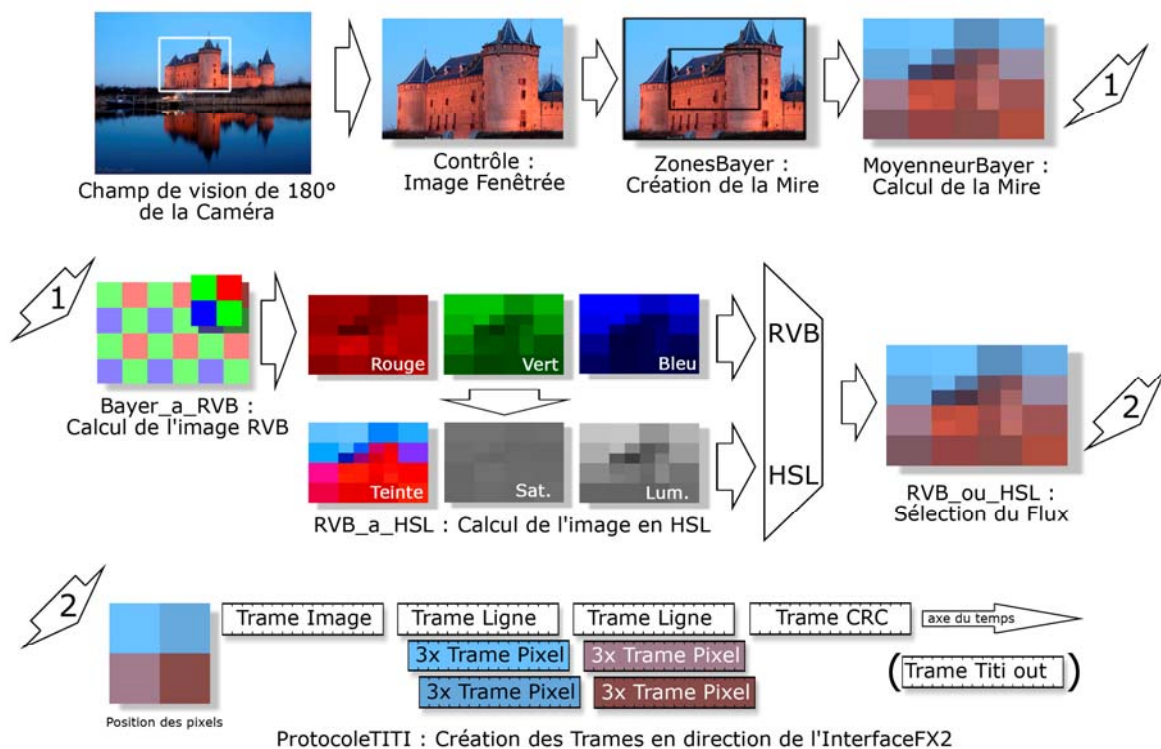


5. Principe de fonctionnement

5.1 Etapes du traitement

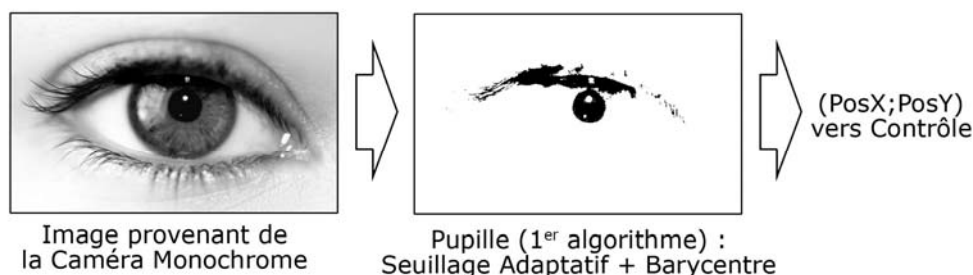
Partant de l'idée globale, j'ai commencé par étudier comment découper le traitement général en plusieurs étapes, chacune étant en quelque sorte un maillon de la chaîne de traitement de l'image. Tout ceci s'effectue à l'intérieur de la FPGA sous la forme de modules. Le prochain chapitre traitera uniquement de la partie FPGA de mon projet.

C'est ainsi que j'ai produit (après de nombreuses ébauches) la solution de traitement d'images convertissant une source en Bayer en un flux USB. Celle-ci sera représentée sous la forme RVB ou HSL et découpée en deux zones de résolutions différentes (explications du pourquoi et du comment de tout ceci dans la suite de ce document) :



5.2 Contrôle global et positionnement de la pupille

L'algorithme permettant de situer le regard de la personne non-voyante effectue un traitement sur l'image provenant de la caméra monochrome dans le but de positionner la pupille...



La logique de Contrôle s'occupe de configurer :

- Les modes de fonctionnement de toutes les caméras par le biais de l'I²C
- le fenêtrage (d'après la position de la pupille) de la caméra couleur par l'I²C
- les paramètres de ZonesBayer (facteurs de réduction, position de la zone...)
- le paramètre de RVB ou HSL (type de flux en sortie)
- et de récupérer les ordres de la part de notre logiciel pour les traiter...

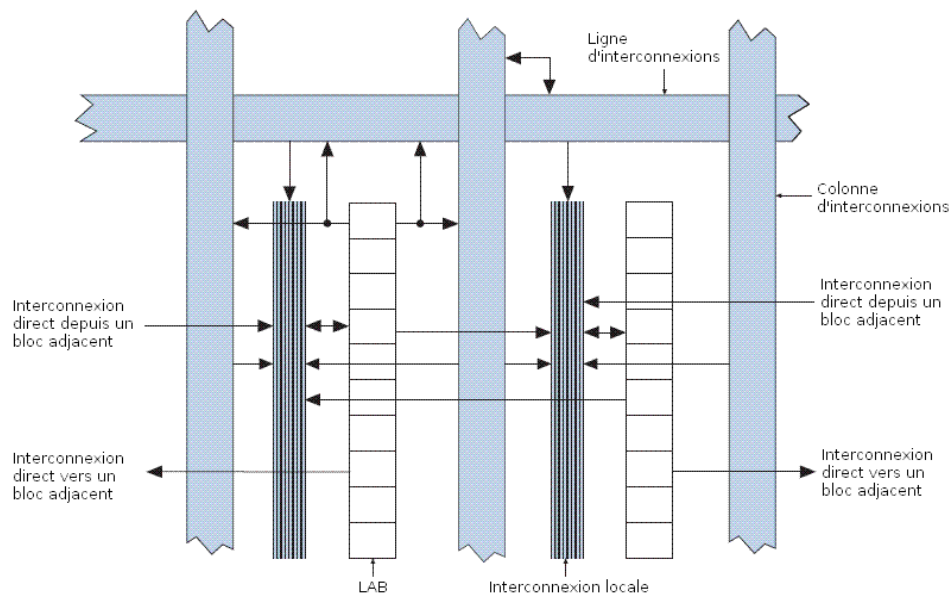
Modules FPGA

5.3 Introduction, principe de la FPGA



Pub représentant l'engineering board (Cyclone de la compagnie Altera)

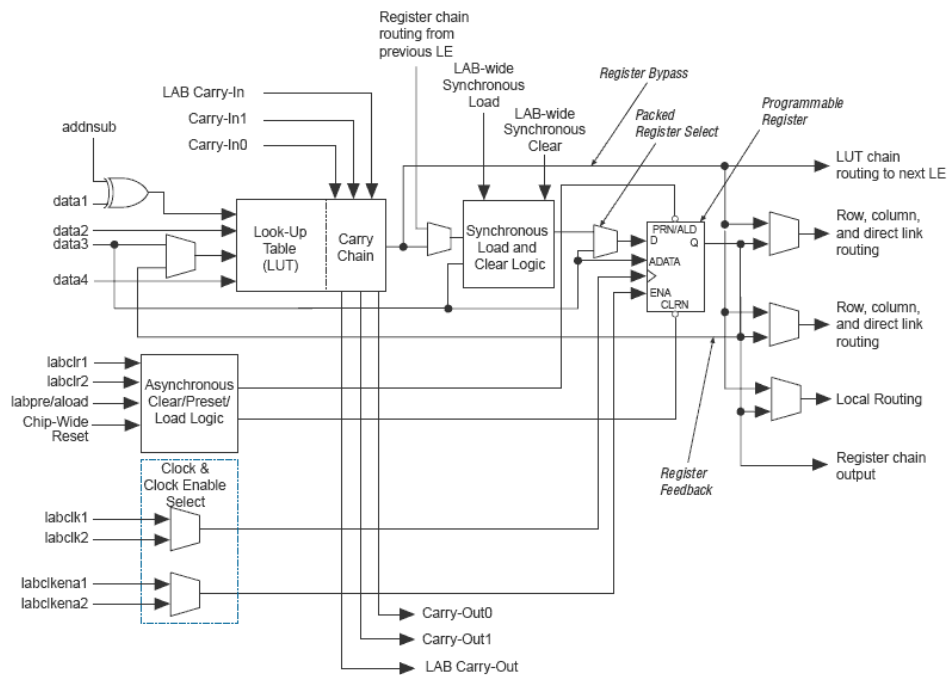
Le terme FPGA est l'abréviation anglaise de « **field-programmable gate array** » ce qui signifie « réseau de portes programmables in-situ ». C'est une des nombreuses technologies de circuit logique programmable existante.



Connexion des cellules LAB de la FPGA Cyclone
Chaque LAB contient 10 LE (logical element)

Une FPGA est constituée de nombreuses cellules logiques élémentaires pouvant être librement configurées et interconnectées, définitivement ou, comme dans notre cas, de façon réversible par programmation pour au final implémenter notre système !

Le langage utilisé est le VHDL. Il permet en quelque sorte de nous affranchir de la lourde tâche qu'est la programmation d'une FPGA et rend notre code portable sur d'autres technologies de circuit logique programmable. Ceci nous laisse l'occasion de nous centrer sur l'implémentation logique liée à notre algorithme, car nous n'avons pas vraiment envie de router et configurer les 1248 LAB (12'480 LE) de la FPGA Cyclone manuellement !



Un LE (logical element) programmable !

Voici les caractéristiques de notre FPGA modèle Cyclone EP1C12F256C7 d'Altera.

- Une FPGA formée de 1248 LAB's contenant 10 LE's par LAB, ce qui donne au final 12'480 (en fait 12'060) LE éléments logiques programmables et routables.
- Une RAM interne agencé sous le format bloc M4K (4 Kbits + Parité), les 52 blocs M4K du Cyclone nous donne une capacité interne de 239,616 bits que nous pouvons cadencer à une fréquence maximale de 250 MHz.
- Notre FPGA contient même 2 PLLs « phase locked loops » permettant de générer plusieurs horloges de fréquence multiple de l'horloge globale en interne.

Pour donner un ordre de grandeur de la tâche qui incombe à notre circuit logique programmable nous devons implémenter : la gestion des trois caméras / le procédé permettant de suivre avec la caméra noir&blanc la pupille du non-voyant / la logique de contrôle permettant entre autres le fenêtrage dans les images provenant des caméras couleur (par le biais du protocole I²C) / l'algorithme de la mire avec sa mémoire / l'algorithme de conversion de la matrice de bayer en RVB avec ses tampons de ligne / l'algorithme de transcodage RVB → HSL / la transformation de tout ceci en trames titi / l'interface de gestion du FX2 / et finalement la partie de sonorisation (développement futur) ! A bout de souffle !

5.4 Le langage de description matériel

VHDL ou « *very high speed integrated circuit hardware description language* » traduit « *Langage de description du matériel pour circuit intégré à très haute vitesse* ».

Un langage de description du matériel, dans notre cas le VHDL, permet, entre autre, d'implémenter des fonctionnalités, des algorithmes, dans une technologie de type circuit logique programmable (FPGA par exemple), tout ceci dans le but de construire une électronique du type embarquée. La notion de signal et la logique non séquentielle sont les différences majeures apportées par le VHDL vis-à-vis d'un langage logiciel.

Le logiciel Quartus de la compagnie Altera permet facilement de coder en VHDL, d'analyser et de synthétiser ce dernier (compiler dans un langage logiciel), de faire l'assemblage sur la FPGA, de router les éléments logiques (faire l'édition de liens en logiciel) et non seulement de voir le chemin critique mais en plus de simuler la réaction de la logique ainsi réalisée avec un ensemble de signaux d'excitation !

5.5 ZonesBayer & MoyenneurBayer

5.5.1 Introduction, la caméra CMOS

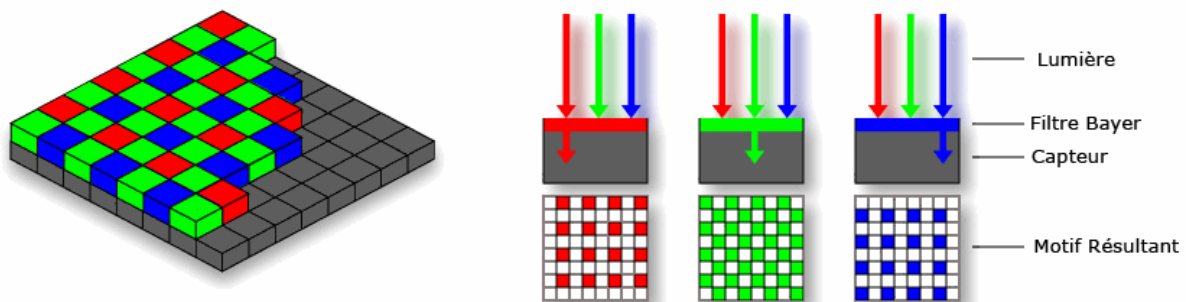
La première étape du traitement des images de l'environnement étant la capture d'une image par le biais d'une caméra couleur du type CMOS, nous devrons étudier le type de flux que celle-ci nous fournit pour pouvoir le traiter.

Voici un bref descriptif du module caméra utilisé au cours de ce développement, modèle KAC9648 de Kodak :

- | | |
|---------------------------|----------------------|
| ➤ Modèle de caméra | KAC9648 de Kodak |
| ➤ Résolution du capteur | 1032 x 1312 pixels |
| ➤ Résolution active | 1032 x 1288 pixels |
| ➤ Vitesse d'acquisition | 18 images / seconde |
| ➤ Format des pixels | matrice de bayer RVB |
| ➤ Quantification du pixel | sur 10 bits / pixel |



Une matrice de bayer RVB est un filtre constitué d'un motif de base répétitif, déposé sur un capteur de luminosité. Elle permet à ce dernier de capter une composante colorimétrique différente par pixel, ce qui, après un décodage approprié, nous permet d'avoir une image en couleurs.



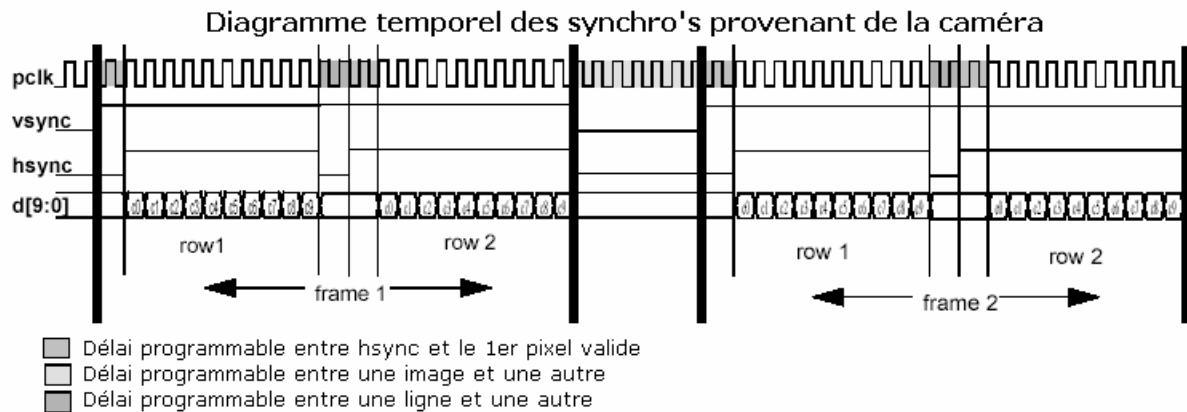
Au niveau matériel du terme, cette information visuelle nous provient sous la forme d'un pixel quantifié sur 10 bits passant par le port **d[9..0]** de la caméra comme noté dans le datasheet.

Sans informations de synchronisation, le flux provenant de la caméra ne pourrait tout simplement pas être décodé convenablement. C'est pourquoi cette caméra nous fournit plusieurs signaux de synchronisation, dont la synchro verticale, horizontale et pixel.

Je vais dorénavant vous parler du seul mode utilisé dans mon projet, le mode film continu ; les explications qui suivent se basent uniquement sur ce dernier :

- Le signal de synchronisation verticale **vsync**, passe par une phase d'inactivité et s'active pendant chaque image, nous permettant de connaître la situation par rapport au flot d'images contiguës.
- Le signal de synchronisation horizontale **hsync**, uniquement actif au cours de l'envoi d'une ligne, ce qui nous permet de nous situer dans une image.
- Le signal d'horloge pixel **pclk**, nous dicte le rythme du flux de pixels (celui-ci étant directement dérivé de l'horloge principale **mclk** que l'on envoie à la caméra pour son propre fonctionnement ce qui va simplifier notre logique).

Tout ceci peut-être résumé dans un chronogramme provenant du datasheet (traduit) :



5.5.2 But du système

Reprenons notre objectif ; comme exprimé dans le principe de fonctionnement nous voulons constituer une image répondant au principe de la zone centrale de la vision et de celle périphérique, moins détaillée.



Pour ce faire nous devons en premier lieu récupérer les différents signaux de synchronisation et d'images provenant de la caméra couleur pour ensuite effectuer un calcul de moyenne (des couleurs) au niveau des pixels et finalement fournir une image, dont chacun des pixels provient de l'image source réduite (moyennée).

Au niveau des contraintes le moyeneur est capable de traiter séparément les différentes composantes du motif de bayer pour en calculer une composante moyenne et ceci avec différents facteurs de réduction en fonction du positionnement dans l'image !

Pour résumer, il produira des images de bayer moyennées à partir d'images de bayer.

Simplifier la problématique suppose de découper le problème en deux parties :

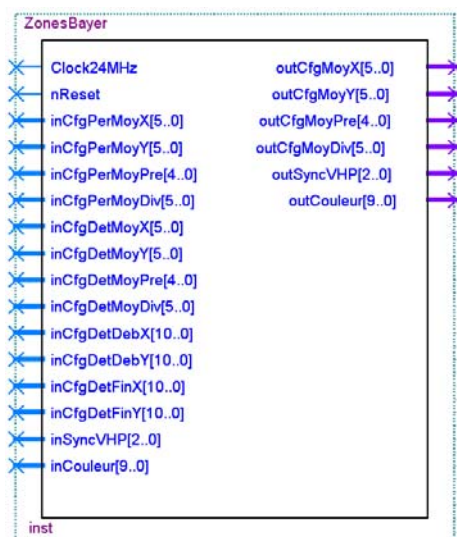
➤ ZonesBayer

Utilise vsync, hsync et pclk, pour en déduire les facteurs de réduction en fonction du pixel en cours dans l'image traitée ; retarde les signaux pour les synchroniser avec les facteurs calculés

➤ MoyenneurBayer

Utilise le flux d'images en prenant en compte les facteurs provenant de ZonesBayer pour effectuer un calcul de moyenne en temps réel et nous fournir en finalité un flux d'images moyennées

5.5.3 Fonctionnement détaillé de ZonesBayer



inCfgPerMoy ... : configuration de la périphérie
inCfgDetMoy... : configuration de la zone détaillée

...X/Y : facteur de réduction sur l'axe X/Y
...Pre : facteur de pré division /Pre (de inCouleur)
...Div : facteur de division (ex : si ...Pre=1 \rightarrow X.Y)

inCfgDet... : positionnement de la zone détaillée
...Deb X / Y : position du 1^{er} pixel pris \rightarrow détaillé
...Fin X / Y : position du dernier pixel pris

inSyncVHP : pack des synchros en entrée
inCouleur : pixel de l'image entrant

outSyncVHP : pack des synchros en sortie
outCouleur : pixel de l'image sortant

Le module est composé du processus synchrone et de trois processus combinatoires.

Processus en rapport avec les Zones

Comme nous travaillons sur un motif de Bayer, nous prenons un groupe de quatre pixels (2x2) comme étant un seul. La numérotation des pixels prend cela en compte et de ce fait, le positionnement de la zone détaillée s'effectue sur un multiple de ce motif. Pour se faire, j'ai créé une « machine d'état » dont l'état change en fonction des signaux de synchronisation inSyncVHP (V, H, P).

Chacun des trois signaux de synchronisation est noté suivant son initiale ; et une notation du genre 1 \rightarrow 0 correspond à un flanc descendant, 0 \rightarrow 1 à un flanc montant.

- si $\text{inSyncVHP}(V, H) = (0, 0)$ c'est une pause entre deux images
 \rightarrow Initialisation des registres du positionnement en X&Y
- si $\text{inSyncVHP}(V, H) = (1, 0)$ c'est une pause entre deux lignes
 \rightarrow Initialisation des registres de positionnement en X
 - si en plus $H=1 \rightarrow 0$ est dans un flanc descendant
 \rightarrow Incréméntation du registre de positionnement en Y
 si c'est un multiple de 2 \rightarrow incréméntation de la numérotation en Y
- si $\text{inSyncVHP}(V, H, P) = (1, 1, 1)$ alors c'est un nouveau pixel !
 \rightarrow Incréméntation du registre de positionnement en X
 si c'est un multiple de 2 \rightarrow incréméntation de la numérotation en X

Processus de Changement de Configuration

Notre module doit prendre en compte la configuration de la mire en cours, provenant de la logique de contrôle, qu'entre deux images ; sous peine de cafouiller.

- si $\text{inSyncVHP}(V, H) = (0, 0)$ c'est une pause entre deux images
 \rightarrow Mémorisation de la configuration (inCfg...)

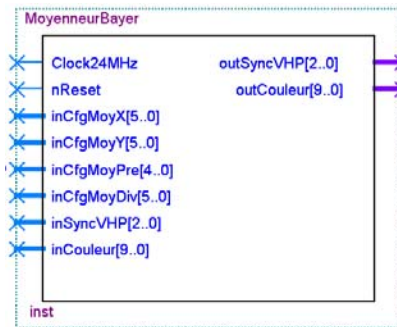
Remarque : Je ne l'ai pas encore dit et ne le ferais plus mais la mémorisation des registres et la mise à zéro des registres impulsionsnels (accusés de lecture etc...) s'effectuent par défaut dans le processus.

Processus de Sortie

Ce processus permet de donner en sortie, au MoyenneurBayer une configuration lui permettant de créer une zone périphérique et détaillée.

- si la numérotation entre dans la zone détaillée (inCfgDet Deb / Fin)
 - La sortie prendra la configuration concernant la zone détaillée
- sinon
 - La sortie prendra la configuration concernant la zone périphérique

5.5.4 Fonctionnement détaillé de MoyenneurBayer



inCfgMoy... : configuration du moyenneur

...**X/Y** : facteur de réduction sur l'axe X/Y

...**Pre** : facteur de pré division /Pre (de inCouleur)

...**Div** : facteur de division (ex : si ...Pre=1 → X·Y)

inSyncVHP : pack des synchros en entrée

inCouleur : pixel de l'image entrant

outSyncVHP : pack des synchros en sortie

outCouleur : pixel de l'image sortant

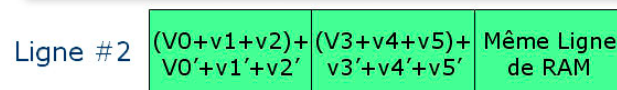
Remarque : L'image source étant constituée d'un motif de Bayer j'ai eu l'obligation d'effectuer le calcul de moyenne de façon spécifique, en prenant en compte le fait que d'être positionné sur un pixel pair / impair d'une ligne paire / impaire me fait changer de composante du motif.

Ce motif m'a forcé à traiter un groupe de 2x2 pixels comme étant un élément de base, quand je parle de pixels, de voisins et de lignes je parle en fait d'une seule des composantes du motif : je traite chaque composante séparément.

Ce calcul sur les pixels implique le fait d'additionner les voisins sous la forme d'un bloc de inCfgMoyX par inCfgMoyY pixels. Ceci nous oblige à stocker en mémoire le résultat intermédiaire, calculé en prenant inCfgMoyX pixels sur la même ligne.

Dès l'arrivée d'une prochaine ligne, nous continuons notre addition et prenons en compte chaque résultat intermédiaire, ce qui peut se schématiser ainsi :

Déroulement en 3x2 [Bayer]



Remarque : Avant tout traitement, d'après la valeur de `inCfgMoyPre` le pixel en entrée est pré divisé puis stocké dans un signal comportant 16 bits. Cette étape a pour but d'éviter que l'additionneur sature dans le cas où les facteurs de réduction sont trop importants et impliquent un trop grand nombre de pixels dans le même calcul (avant l'opération de division en pipeline). Il faut donc ne pas l'oublier et corriger le facteur de division en conséquence.

Pour pouvoir prendre en compte des facteur de réduction verticaux non constants dans l'image (typiquement notre mire) j'ai dû modifier l'algorithme de ce module pour l'obliger à stocker dans une mémoire le positionnement vertical relatif (en Y) de la même manière que je stocke les résultats intermédiaires de l'addition (calcul de la moyenne).

Le module est composé du processus synchrone et de deux processus combinatoires.

Processus en rapport avec Bayer et la Moyenne

J'ai conçu l'algorithme de ce processus de telle manière qu'il puisse être utilisé par chacun des éléments du motif de Bayer : par chacune des composante couleur.

Remarque : L'algorithme décrit une version simplifiée, épurée de quelques détails.

Chacun des trois signaux de synchronisation est noté suivant son initiale ; et une notation du genre $1 \rightarrow 0$ correspond à un flanc descendant, $0 \rightarrow 1$ à un flanc montant. Les signaux (suffixe s) correspondant aux entrées retardées d'un coup d'horloge.

Dans le groupe 2x2 de pixels du motif de Bayer la notation HG signifie pixel du coin en haut à gauche : donc le tout 1^{er} (H = Haut, B = Bas, G = Gauche et D = Droite)

- si $\text{sinSyncVHP}(V) = (0)$ c'est une pause entre deux images
→ Initialisation du registre de position en Y
- si $\text{sinSyncVHP}(H) = (0)$ c'est une pause entre deux lignes
→ Initialisation des registres de positionnement en X
→ Initialisation du déplacement en mémoire ($\text{AdLecture}=0$ $\text{AdEcriture}=-1$)
- si $\text{inSyncVHP}(H)=(0 \rightarrow 1)$ et $\text{sinCfgMoyX}=1$ → $\text{AdLecture} = 1$
- si $\text{sinSyncVHP}(V, H, P) = (1, 1, 1)$ alors c'est un nouveau pixel !
→ Incrémentation des registres de positionnement en X (modulo sinCfgMoyX)
 - si l'on est à G et que c'est le 1^{er} pixel (en X) → Incrémente AdEcriture
 - si l'on est à D et (avant dernier pixel en X ou $\text{sinCfgMoyX}=1$) → $\text{AdLecture}++$
 - si c'est le 1^{er} pixel (en X) de la ligne
→ La variable de Calcul vaut le signal couleur
sinon
→ La variable de Calcul vaut le signal couleur + « Data »
 - si c'est le dernier pixel de la ligne (déduit par sinCfgMoyX)
→ Signal d'écriture mémoire pixel
→ Signal d'écriture mémoire posY si nous sommes sur un pixel BD
 - si ce n'est pas la 1^{ère} ligne du groupe (contenu de la mémoire posy)
→ La variable de Calcul vaut Calcul + contenu de la mémoire pixel
→ Signaux en direction du diviseur si nous sommes sur la dernière ligne
→ Sauvegarde de la variable de Calcul dans « Data »
- si $\text{sinSyncVHP}(H)=(1 \rightarrow 0)$ → Mise à jour du registre de position en Y ($H \leftarrow B$)

Processus de Multiplexage

Le processus faisant le traitement de la moyenne des pixels peut être multiplexé, c'est pourquoi j'ai implémenté ce processus permettant de le « mettre en commun ».

- si nous sommes sur un pixel HG
→ Multiplexer les signaux communs avec ceux traitant le pixel HG
- ainsi de suite... (pour chacune des quatre composantes)

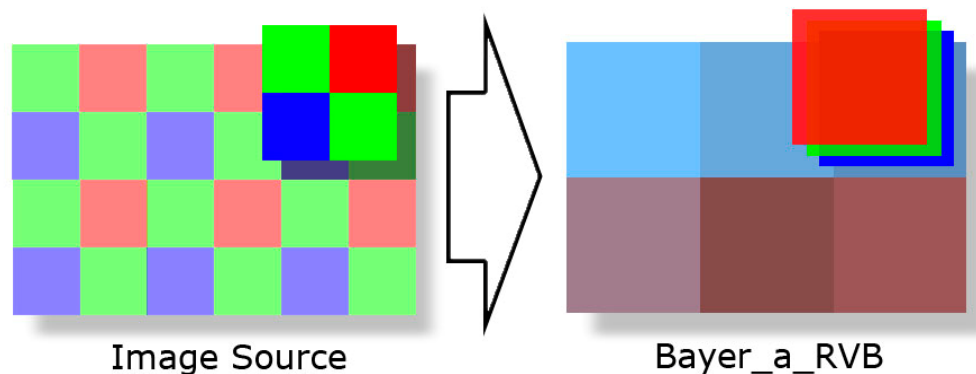
5.6 Bayer_a_RVB

5.6.1 But du système

Comme indiqué plus haut, MoyenneurBayer ne fait que produire une image ayant la caractéristique d'être le résultat d'un traitement sur une image source. Sinon je l'aurais appelé MoyenneurBayer_a_RVB !

Le présent module permet donc de créer une image RVB à partir d'une image provenant d'un capteur ayant un masque de Bayer.

Habituellement un tel traitement implique une interpolation des deux composantes manquante pour chaque pixel à partir des voisins. Cependant, dans un souci de simplicité, un autre type de traitement est appliqué.

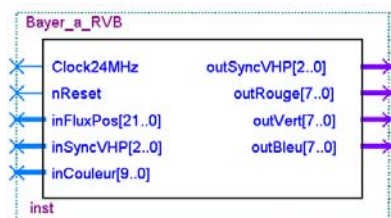


Le fait est qu'au cours de mon étude j'ai gardé à l'esprit que la caméra nous fournit bien trop de pixels par rapport au flux final. Après application de MoyenneurBayer, chacune des composantes étant le fruit de plusieurs, l'écart existant entre les différentes composantes du motif de Bayer, s'entrelaçant, devient non significatif.

C'est pourquoi ce module, pour produire une image RVB, regroupe exactement un motif 2x2 de pixels, contenant 1 pixel rouge, 2 verts, 1 bleu, en un pixel complet en sortie.

Vous comprendrez dès lors que la résolution est encore divisée par deux : il ne faut pas l'oublier et en tenir compte au moment de choisir les facteurs de division pour connaître le nombre exact de pixels en sortie ! Finalement, le format en sortie passe de 10 bits de la caméra à 8 bits, en oubliant les bits de poids faible, car la précision résultante est suffisante.

5.6.2 Fonctionnement détaillé



inFluxPos : pack des informations de fenêtrage

inSyncVHP : pack des signaux de synchros en entrée

inCouleur : pixel de l'image entrant (du motif de Bayer)

outSyncVHP : pack des signaux de synchros en sortie

outRouge / outVert / outBleu : pixel sortant en RVB

Comme nous devons regrouper 2x2 pixels, il nous est nécessaire de stocker les lignes impaires (1^{ère} ligne de chaque groupe) pour ensuite les renvoyer en sortie.

Pour ce faire ce module intègre deux mémoires FIFO, chacune s'occupant de stocker une des composantes du motif de Bayer (les pixels de la ligne impaire). L'envoi en sortie du pixel RVB se produit dès que nous avons reçu toutes les composantes faisant partie du même « groupe de 2x2 pixels ». L'envoi s'effectue qu'aux pixels pairs des lignes paires.

Le module est composé du processus synchrone et de trois processus combinatoires.

Processus en rapport avec Bayer

Comme cité plus haut, nous prenons un groupe de quatre pixels (2x2) correspondant au motif de Bayer pour produire un seul en RVB. J'ai donc créé une « machine d'état » dont l'état change en fonction des signaux de synchronisation inSyncVHP (V, H, P).

Chacun des trois signaux de synchronisation est noté suivant son initiale ; et une notation du genre $1 \rightarrow 0$ correspond à un flanc descendant, $0 \rightarrow 1$ à un flanc montant.

- si $\text{inSyncVHP}(V, H) = (0, 0)$ c'est une pause entre deux images
→ Initialisation des registres du positionnement en X&Y
- si $\text{inSyncVHP}(V, H) = (1, 0)$ alors c'est une pause entre deux lignes
→ Initialisation des registres de positionnement en X
 - si en plus $H=1 \rightarrow 0$ est dans un flanc descendant
→ Incréméntation des registres de positionnement en Y
- si $\text{inSyncVHP}(V, H, P) = (1, 1, 1)$ alors c'est un nouveau pixel !
→ Incréméntation des registres de positionnement en X
si ligne impaire → stockage
si ligne paire et pixel pair → Accusé de lecture des mémoires FIFO internes
→ La suite (sortie) implique un autre processus

Processus de Changement de Configuration

Notre module doit prendre en compte les informations du fenêtrage en cours, provenant de la logique de contrôle, qu'entre deux images ; sous peine de cafouiller.

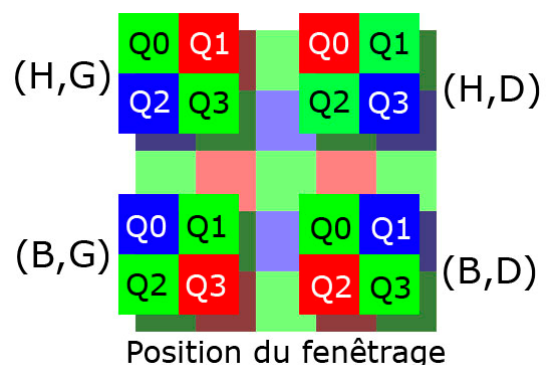
- Si $\text{inSyncVHP}(V, H) = (0, 0)$ alors c'est une pause entre deux images
→ Mémorisation de la configuration (inPosX , inPosY)

Processus de Sortie

A cause du fenêtrage de l'image provenant de la caméra, il se pourrait que l'image source ne commence pas toujours par la même composante du motif de bayer, c'est pourquoi j'ai pris en compte la position du fenêtrage pour multiplexer la sortie finale.

De façon générale les deux premières composantes du motif sont stockées en mémoire FIFO (nommé ici Q0, Q1), une autre à été retardée Q2 et la dernière viens de nous être donnée en entrée Q3.

- si $sPos(Y, X) = (H, G)$
→ La sortie rouge prendra Q1
→ La sortie verte prendra $(Q0+Q3)/2$
→ La sortie bleue prendra Q2
- si $sPos(Y, X) = (H, D)$
→ La sortie rouge prendra Q0
→ La sortie verte prendra $(Q1+Q2)/2$
→ La sortie bleue prendra Q3
- Ainsi de suite (d'après le dessin)

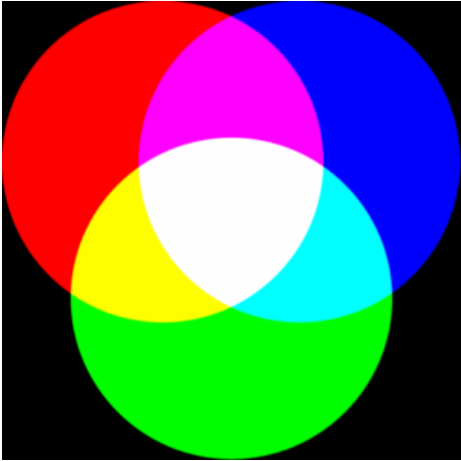
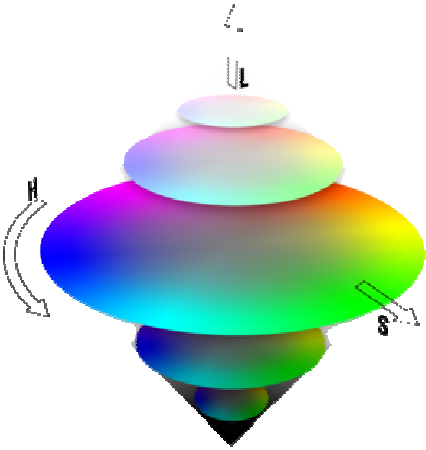


5.7 RVB_a_HSL

5.7.1 Introduction, l'espace HSL

Jusqu'à ce point notre système s'occupait de traiter une image codant la quantité de trois composantes primaires (rouge, verte, bleue) ; cependant ce n'est pas le meilleur choix pour notre application !

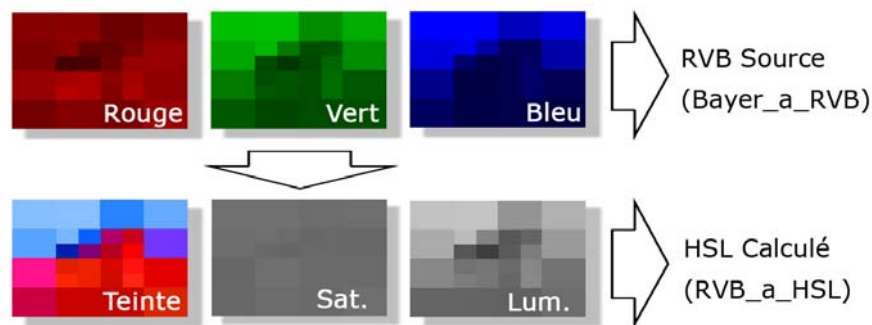
La représentation la plus intuitive de la lumière, celle qui serait la meilleure candidate à la transformation sous forme de sons est l'espace colorimétrique HSL. Car celui-ci exprime une couleur par sa teinte, sa saturation et sa luminance.

| Codage RVB | Codage HSL |
|--|--|
|  |  |
| <p>RGB in English, 'Rouge,Vert,Bleu'</p> <p>Format d'affichage de notre écran qui reflète le fonctionnement de notre œil composé de cellules sensibles.</p> <p>Toutes les couleurs perçues par nous sont en fait formées par l'assemblage des trois couleurs primaires captées par nos yeux. Donc ce format ne fait que coder l'intensité des trois couleurs primaires.</p> <p>Petit exemple : le violet est formé d'un maximum de rouge et de bleu mais le vert n'intervient pas.</p> | <p>Format intuitif adapté au futur du projet</p> <p>H comme Teinte Rouge ↔ ... ↔ Jaune</p> <p>S comme Saturation, terne ↔ vif</p> <p>L veut dire Luminance, clair ↔ foncé</p> <p>La figure est très parlante, nous voyons bien avec le double cône que plus une couleur est lumineuse/sombre plus elle tire vers le blanc/noir car le cône se rétrécit.</p> <p>Par exemple un orange pourrait être aussi bien lumineux que sombre et terne que vif.</p> |

5.7.2 But du système

L'objectif du système étant de calculer, avec l'algorithme proposé, la représentation HSL d'un pixel provenant en RVB et finalement de synchroniser les deux flux... Pour être exploitable ce système doit nécessairement avoir un temps de propagation au plus égal au temps de repos entre deux pixels.

Pour permettre à un multiplexeur qui serait situé après ce module, de sélectionner le type de flux, nous devons retarder le flux RVB ; autrement dit, nous devons créer un registre à décalage entre l'entrée et sa copie retardée.



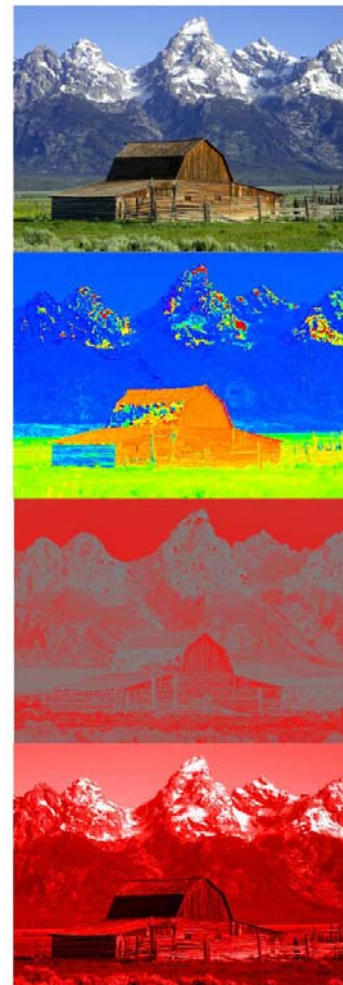
L'équation ci-dessous nous permet, partant d'un pixel représenté par les trois composantes primaires de calculer l'image de celui-ci dans l'espace colorimétrique HSL :

$$H = \begin{cases} \text{undefined} & \text{if } MAX = MIN \\ 60 \times \frac{G-B}{MAX-MIN} + 0, & \text{if } MAX = R \\ & \text{and } G \geq B \\ 60 \times \frac{G-B}{MAX-MIN} + 360, & \text{if } MAX = R \\ & \text{and } G < B \\ 60 \times \frac{B-R}{MAX-MIN} + 120, & \text{if } MAX = G \\ 60 \times \frac{R-G}{MAX-MIN} + 240, & \text{if } MAX = B \end{cases}$$

$$S = \begin{cases} 0 & \text{if } L = 0 \\ \frac{MAX-MIN}{MAX+MIN} = \frac{MAX-MIN}{2L}, & \text{if } 0 < L \leq \frac{1}{2} \\ \frac{MAX-MIN}{2-(MAX+MIN)} = \frac{MAX-MIN}{2-2L}, & \text{if } L > \frac{1}{2} \end{cases}$$

$$L = \frac{1}{2}(MAX + MIN)$$

composantes entre 0.0 et 1.0 (360.0 pour H)
MIN,MAX : minimum et maximum de R,G,B



Etant donné que l'algorithme présenté ci-dessus est constitué de nombreux chemins conditionnels et surtout de divisions, nous devons le construire de cette manière si nous voulons entrer dans les contraintes :

- Ramener H entre 0.0 et 1.0
- Des circuits de divisions en pipeline
- une méthodologie de multiplexage adapté
- Une astuce de division pour l'effectuer sur un nombre à virgule fixe 0,8 bits
- La copie exacte du flux RVB mais retardée pour synchroniser le flux RVB au HSL

5.7.3 L'arithmétique à virgule fixe

Comme vous venez de le voir les équations de passage du RVB au HSL implique des opérations sur les nombres décimaux : ce qu'il faudra implémenter !

Pour représenter les nombres dans le monde binaire qu'est celui des processeurs et circuits logiques programmables type FPGA et autres nous avons plusieurs méthodes, ayant chacune ses propres avantages et inconvénients.

Concernant les nombres réels (communément appelés à virgule) je vais vous en présenter deux, et développer celle que j'ai finalement retenue !

Les nombres à virgule flottante ou la norme IEEE 754

Les processeurs des ordinateurs ont dans leur jeu d'instructions les opérations arithmétiques basées sur les nombres au format IEEE 754. C'est pourquoi quand vous développez un programme vous aurez de la facilité à utiliser le format « float » basé sur cette norme !

Signe S = 1 si Nombre[31] = 0 sinon -1

Un nombre vaut selon ce format $S \cdot M \cdot 2^E$



Exposant E = Nombre[30..23]-127

Mantisse (en binaire) M = 1.Nombre[22..0]

Mantisse (en décimal) M = $1 + \text{Nombre}[22..0]/2^{24}$

*Représentation de la version 32 bits de IEEE 754 (64 et 80 existent)
Dynamique de -10^{38} à 10^{38} , précision effective de 6 décimales.*

Cependant, dans l'optique d'une implémentation dans un système embarqué, nous voulons économiser un maximum de composants, c'est pourquoi ce format, fort pratique, mais complexe à mettre en œuvre ne sera pas utilisé.

Signe S = 1

Ce Nombre (π) vaut **3.1415927**



E = 128 - 127 = 1

Mantisse (en binaire) M = 1.10010010000111111011011

Mantisse (en décimal) M = 1.570794

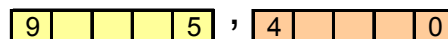
Exemple de la représentation de $\pi = 3.14\ 15\ 92\ 65\ 35\ 89\ 79\ 32\ 38\ 46...$

Les nombres à virgule fixe

Le format à virgule fixe est simple à comprendre, un nombre à virgule fixe n'est rien d'autre qu'un nombre entier divisé par une certaine puissance de 2.

Ce qui veut dire que nous coderons un nombre de cette manière :

Valeur = $\sum \text{poids} \cdot \text{bits}$



poids des bits en non signé

$2^4\ 2^3\ 2^2\ 2^1\ 2^0$

$2^{-1}\ 2^{-2}\ 2^{-3}\ 2^{-4}\ 2^{-5}$

poids des bits en signé

$-2^4\ 2^3\ 2^2\ 2^1\ 2^0$

$2^{-1}\ 2^{-2}\ 2^{-3}\ 2^{-4}\ 2^{-5}$

Format à virgule fixe codé sur 10 bits 5,5

Le format que nous utiliserons pour coder l'intensité des pixels sera le 8 bits 0,8 plus économique en éléments logiques que le format 10 bits sortant de la caméra et finalement assez précis pour être utilisable.

Comme la caméra nous fournit des données au format 10 bits entier convertit 8 bits (découpé depuis le poids fort pour être effectué correctement) nous ferons comme si celles-ci représentent un nombre dans ce format, donc entre la valeur 0.000 et 0.996.

Valeur = Σ poids • bits

7 0 0 0 0 0 0 0

poids des bits en non signé $2^{-1} 2^{-2} 2^{-3} 2^{-4} 2^{-5} 2^{-6} 2^{-7} 2^{-8}$

Format des couleurs codées sur 8 bits

Et comme d'habitude un petit exemple :

Valeur = $2^{-1} + 2^{-3} = 0.625$

1 0 1 0 0 0 0 0

poids des bits en non signé $2^{-1} 2^{-2} 2^{-3} 2^{-4} 2^{-5} 2^{-6} 2^{-7} 2^{-8}$

0.625 en virgule fixe 8 bits 0,8

Les opérations en virgule fixe

Je vais exprimer les 4 opérations de façon mathématique pour commencer...

Format utilisé (n+m) bits n,m avec n=0 et m=8 \rightarrow 8 bits 0,8

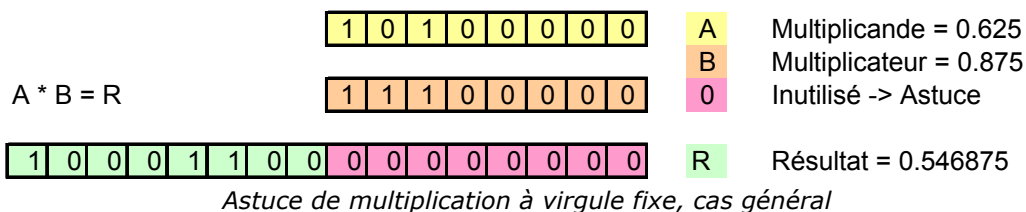
$k = 2^{-m}$ facteur de décalage dans notre cas 1/256

A, B les bits des deux nombres lus comme des entiers

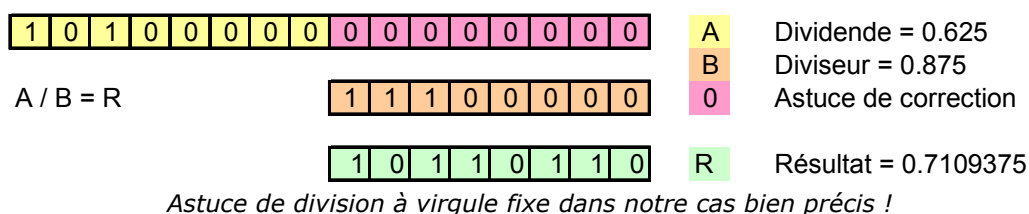
$k \cdot A$, $k \cdot B$ sont les nombres représentés avec le format n,m

- $k \cdot A + k \cdot B = k \cdot (A+B)$ \rightarrow aucune différence avec l'arithmétique entière
- $k \cdot A - k \cdot B = k \cdot (A-B)$ \rightarrow aucune différence avec l'arithmétique entière
- $k \cdot A * k \cdot B = k^2 \cdot (A*B)$ \rightarrow erreur, nous devons diviser le résultat par k
- $k \cdot A / k \cdot B = (A/B)$ \rightarrow erreur, nous devons multiplier le résultat par k

Cependant, pour éviter le dépassement de capacité du type overflow (dans le cas de la multiplication) et underflow (pour la division) nous devons passer par deux astuces matérielles dont voici la schématique :



Pour l'opération de multiplication nous prenons un registre intermédiaire ayant le double de bits, les bits de poids fort de celui-ci seront lus comme étant le résultat.

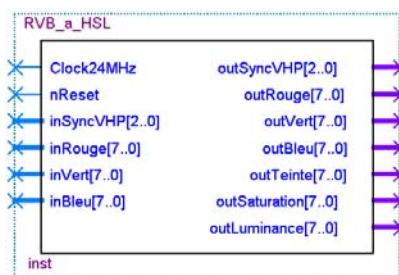


Concernant la division, nous stockons au préalable la valeur du dividende dans les bits de poids fort d'un registre temporaire que nous divisons, « par le diviseur ».

Cette astuce étant utilisée dans un cas bien précis, où le résultat ne dépasse pas le format 8 bits 0.8 (conversion RVB à HSL), donc nous pouvons seulement prendre, sans autres, les bits de poids faible du résultat.

L'autre astuce mise en jeu dans mon projet fut, après avoir quand même appliqué la division, de vérifier s'il y a un résultat saturant et si tel est le cas, le résultat est saturé à $11111111 = 255/256 = 0.996...$ au lieu de 1.0 !

5.7.4 Fonctionnement détaillé



inSyncVHP : pack des signaux de synchros en entrée

inRouge / inVert / inBleu : pixel entrant en RVB

outSyncVHP : pack des signaux de synchros en sortie

outRouge / outVert / outBleu : pixel sortant en RVB

outTeinte / outSat... / outLum... : pixel sortant en HSL

Pour synchroniser l'entrée en RVB et la sortie calculée en HSL j'ai implémenté un registre à décalage qui permet de retarder la sortie du flux RVB, avec les synchros en commun.

Le module est composé du processus synchrone et de trois processus combinatoires.

Processus en rapport avec le Min et le Max

Premièrement, comme vue dans l'équation plus haut, nous devons trouver la composante ayant la valeur minimale et maximale. C'est pourquoi ce processus fait partie du 1^{er} étage du pipeline de calcul... Je l'ai implémenté en utilisant les variables, permettant de penser comme dans la programmation logicielle : en séquentiel.

← Variable MAX vaut inRouge, QUI est le maximum vaut 'R'

➤ si inVert est plus grand que MAX → MAX vaut inVert → QUI vaut 'V'

➤ si inBleu est plus grand que MAX → MAX vaut inBleu → QUI vaut 'B'

← Variable MIN vaut inRouge

➤ si inVert est plus petit que MIN → MIN vaut inVert

➤ si inBleu est plus petit que MIN → MIN vaut inBleu

← Injection des variables (MAX, MIN & QUI) dans les registres à décalage du pipeline

← Injection de MAX - MIN → MAXmMIN(...)

← Injection de MAX + MIN → MAXpMIN (...) (info : sur 1 bit supplémentaire)

Le calcul de la composante luminosité ne faisant pas partie d'un processus, elle est simplement trouvée en oubliant le bit de poids faible d'un certain étage du registre MAXpMIN du pipeline.

Le but étant de calculer trois composantes, je ferais en sorte que celles-ci sortent en même temps du pipeline, c'est pourquoi il me sera nécessaire de trouver l'étage idéal duquel chacune des composantes tirera ses signaux (trouver, pour chaque registre source, le numéro d'étage duquel nous tirerons sa valeur, évidemment « valide » tout au long du pipeline).

Dans un souci de simplicité, cet élément sera mis de côté dans cette explication :

← Luminance vaut MAXpMIN sans le LSB

Processus de calcul de la Saturation

Maintenant que les maximaux sont présents dans le pipeline nous pouvons nous occuper de la composante saturation. Celle-ci (comme la teinte) se calcule en faisant une division. Donc nous allons commencer par calculer les éléments en jeu dans celle-ci.

- si MAXpMIN sature (>255)
 - Le dénominateur vaut 'not' MAXpMIN
 - sinon
 - Le dénominateur vaut MAXpMIN
- Le numérateur (sur 16 bits) vaut MAXmMIN avec encore 8 bits de poids faible à 0

Plus tard... le résultat de la division étant arrivé...

- si le numérateur est égal à zéro alors → La variable S vaut 0
 - sinon si le résultat sature → La variable S vaut la saturation (255)
 - sinon → la variable S vaut la partie basse du résultat
- ← Injection de S dans le registre de sortie de la Saturation

Processus de calcul de la Teinte

- le signal nommé SUB vaut

| | |
|----------------|----------|
| ← Vert - Bleu | si QUI=R |
| ← Bleu - Rouge | si QUI=V |
| ← Rouge - Vert | si QUI=B |
 - le signal nommé ADD vaut

| | |
|----------------|----------|
| ← zéro | si QUI=R |
| ← 1/3 (de 256) | si QUI=V |
| ← 2/3 (de 256) | si QUI=B |
- ← Le dénominateur vaut MAXmMIN
- si le bit de poids fort de SUB vaut 0 [\sim ceci correspond à une valeur absolue]
 - Le numérateur (15 bits) vaut SUB sans le MSB avec encore 8 LSB à 0
 - sinon
 - Le numérateur (15 bits) vaut not SUB sans le MSB avec ... 8 LSB à 0

Plus tard... le résultat de la division étant arrivé...

- si le dénominateur vaut 0 (MAX vaut MIN) → La variable H vaut 0
- sinon
 - si le résultat sature → La variable H vaut 1/6 (de 256)
 - sinon → La variable H vaut la partie basse du résultat divisé par 6
 - si le bit de poids fort de SUB vaut 0 [remet le signe dans 'SUB']
 - La variable H vaut ADD + H
 - sinon
 - La variable H vaut ADD - H

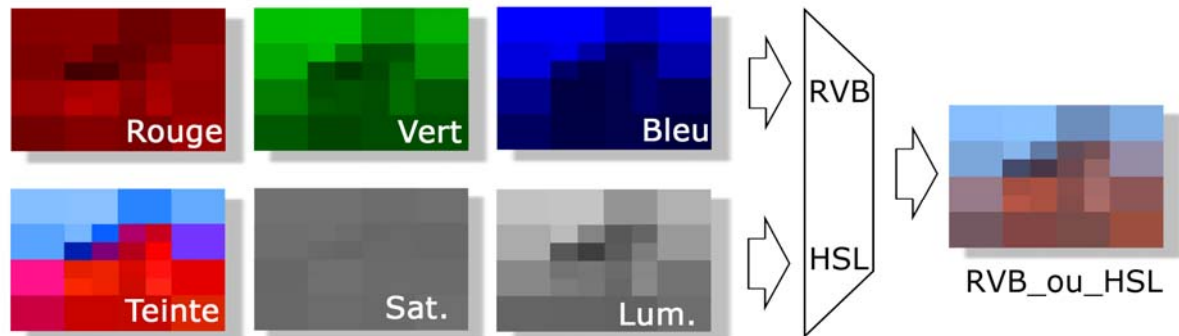
← Injection de H dans le registre de sortie de la Teinte

Voilà, les trois composantes HSL sortent en même temps...

5.8 RVB_ou_HSL

5.8.1 But du système

Maintenant que nous avons deux flux d'image, un en RVB, l'autre en HSL, nous voudrions choisir, suivant le désir de l'utilisateur, lequel sera fourni en sortie de notre système.

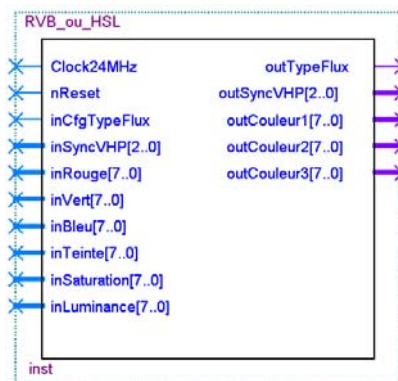


Flux RVB & HSL Source (RVB_a_HSL)

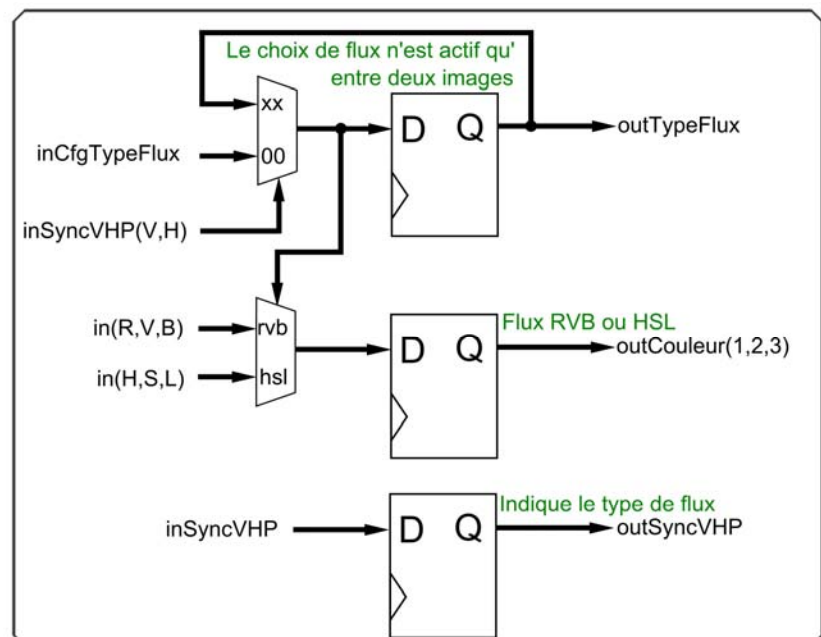
Cependant le changement de type de flux ne doit s'opérer qu'entre deux images au risque de temporairement fournir une image indécodable : formée de quelques pixels RVB et d'autres en HSL (vice-versa). C'est pourquoi ce multiplexeur à un minimum « d'intelligence » permettant de prendre une décision justifiée au cas où la logique de contrôle nous demanderait de changer de flux en plein milieu du transfert d'une image par exemple.

Remarque : Comme expliqué précédemment, RVB_a_HSL nous fournit deux flux image ayant des signaux de synchronisations communs : ce qui m'a permis d'effectuer des tests sur un seul flux de synchronisation (voir fonctionnement détaillé).

5.8.2 Fonctionnement détaillé



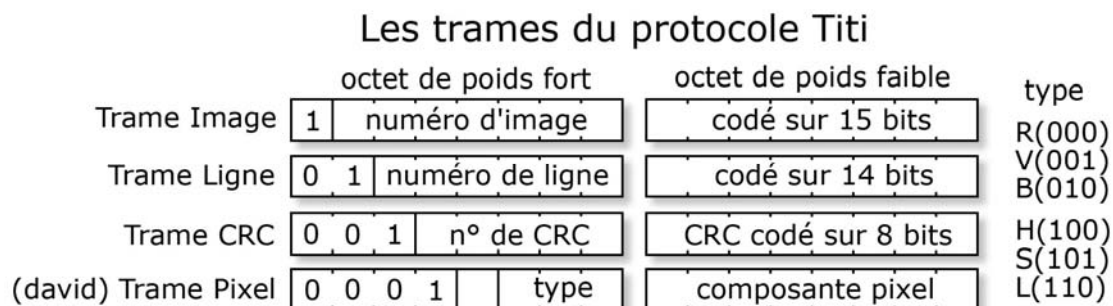
Un circuit logique exprime mieux le fonctionnement de ce multiplexeur de flux qu'un long discours :



5.9 Protocole TITI

5.9.1 Introduction, le protocole Titi

Ce protocole proposé par Monsieur Jacques TINEMBART, permet dans une certaine mesure de s'affranchir des erreurs de transmission de trame (voir ci-dessous), en numérotant les images ainsi que les lignes. Ainsi, notre logiciel pourra, même si certaines trames pixels sont perdues, se synchroniser au niveau image et ligne. La numérotation des trames se fait selon un code de Huffman.



En fin de compte le flux de trame représentant une image se compose de ceci :

- 1 Trame Image en début d'image
- 1 Trame Ligne en début de chaque ligne
- 3 Trames Pixel pour chacun des pixels
- 1 Trame Crc en fin d'image

Ce qui nous explique ceci :

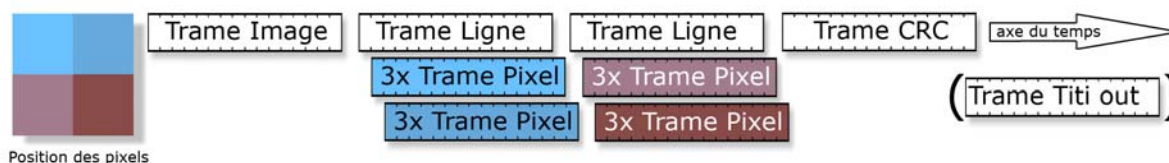
- Nous pouvons connaître le nombre d'images reçues / perdues
- Nous pouvons connaître le nombre de lignes reçues / perdues
- Nous pouvons savoir si les données reçues ont été modifiées
- Si nous perdons des trames pixels, seule la ligne en cours est affectée

Et d'une certaine mesure, il nous serait possible de remédier à ceci de façon logicielle :

- Si nous perdons une trame image, nous ferons un test sur le n° de trame ligne...
- Si nous perdons des trames ligne, plusieurs lignes pourraient se suivre en une plus grande... (à corriger de façon logicielle le cas où nous connaîtrions le nombre exact de pixel par ligne)

5.9.2 But du système

Le but poursuivi par ce module est de passer du monde synchros & pixels au monde trames titi qui passeront plus tard vers l'extérieur par le biais d'un flux USB...

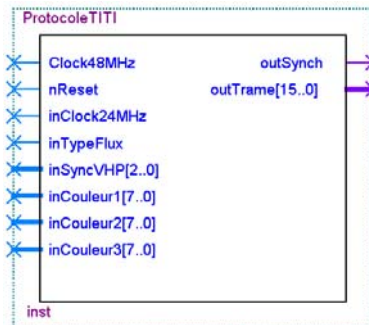


La plus grosse contrainte étant le fait que pour transmettre un pixel nous avons besoin de 3 trames pixel (titi), chacune représentant une des trois composantes de celui-ci en RVB ou HSL. Ce module implémente une mémoire FIFO qui a pour but de stocker les pixels en attente de tramage, ceci afin de prévenir le cas où une utilisation intensive (par

exemple un pixel arriverait pas coup d'horloge d'entrée) ne laisserait pas le temps à ce module d'envoyer les trames entre deux pixels.

Etant donné que le flux de pixels provenant de MoyenneurBayer→Bayer_a_RVB est amoindri, il ne nous sera pas nécessaire de calculer la quantité de pixels devant être stocké dans la mémoire FIFO de contrôle de flux, celle-ci pouvant être minimale (2 mots).

5.9.3 Fonctionnement détaillé



inTypeFlux : nous indique le type de flux RVB/HSL entrant

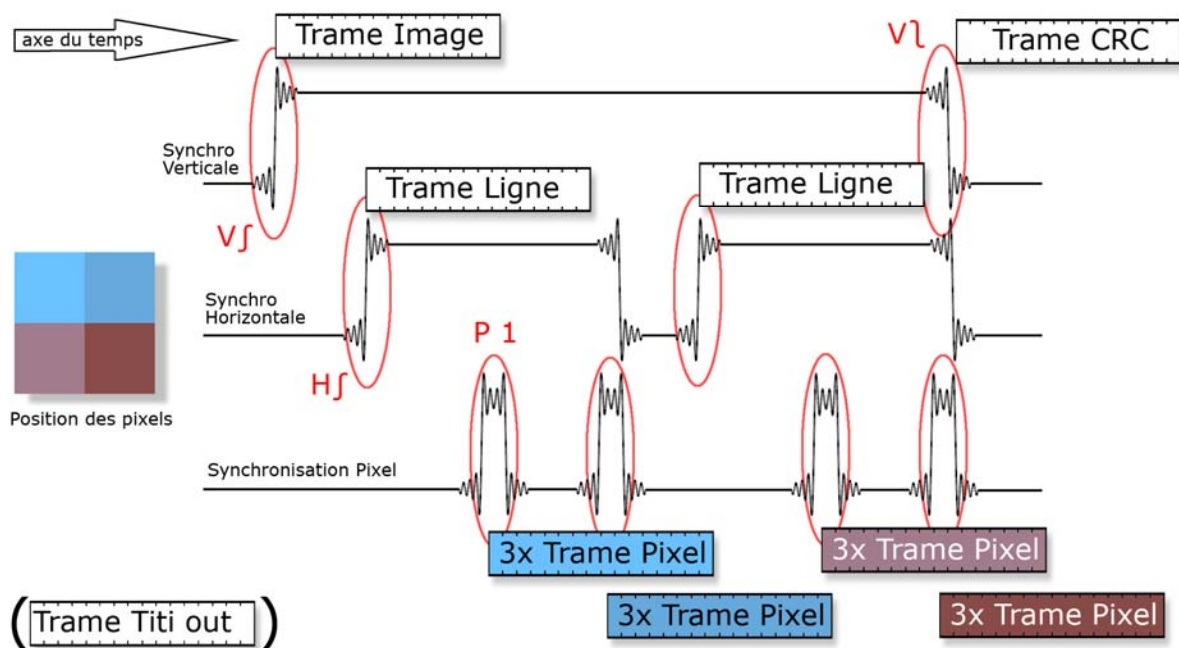
inSyncVHP : pack des signaux de synchros en entrée

inCouleur1 / in...2 / in...3 : pixel entrant en RVB/HSL

outSync : synchronisation de sortie (il y a une trame ?!)

outTrame : trame du protocole titi en sortie

Voici un petit schéma expliquant l'idée générale de l'algorithme :



Pour représenter un pixel avec le protocole titi il nous est nécessaire d'envoyer trois trames composantes et ceci d'affilée. En plus nous pourrions potentiellement recevoir des signaux de synchronisation faisant intervenir deux trames simultanées (flanc montant verticale et horizontale simultanée).

C'est pourquoi j'ai implémenté cet algorithme de telle manière qu'il prenne en compte chaque type de demande (trame image, ligne, pixel, crc) comme des requêtes.

Une hiérarchisation des requêtes permet de transmettre celle-ci dans l'ordre logique et dans n'importe quel cas (excepté ce exemple : l'image n'a pas eu le temps de se faire tramer qu'une autre arrive, dans ce cas les trames image et ligne(s) sont envoyées et les trames pixels des deux images se mélangent ...).

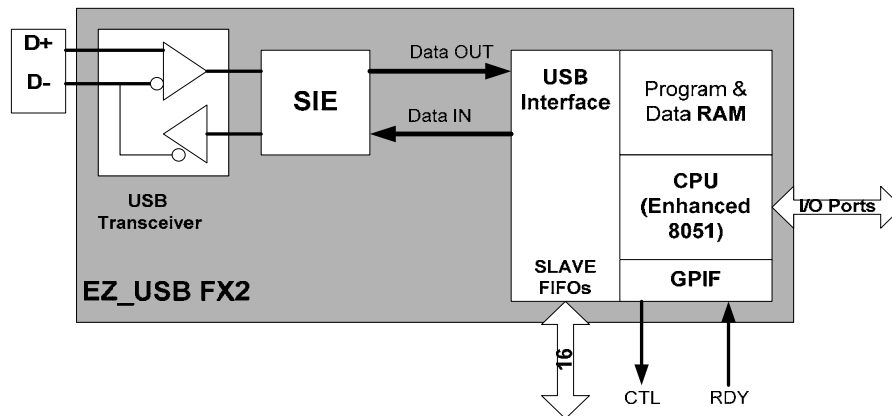
Une mémoire FIFO (servant de tampon) est également nécessaire pour traiter chaque pixel en trois temps, en trois trames... A vous d'imaginer l'algorithme qui va avec cette explication ; -)

5.10 InterfaceFX2

5.10.1 Introduction, le microcontrôleur FX2

Le FX2 de la compagnie Cypress est un microcontrôleur interfaçant un port d'accès parallèle en flux au protocole USB2.

Dessin provenant de mon collègue, Monsieur Nasreddine LAOUAR :



Ce microcontrôleur contient, du côté port parallèle, des mémoires FIFO représentant chacun des 4 endpoint configurables par le firmware. Un endpoint est un canal logique transporté par le flux USB.

Dès que nous connectons le système embarqué sur le port USB de l'ordinateur, une énumération est effectuée auprès de l'hôte avec la configuration par défaut. Ensuite nous pouvons télécharger le firmware du FX2 par le biais de l'USB. Dès la mise en route de celui-ci le FX2 se fait ré énumérer avec notre configuration.

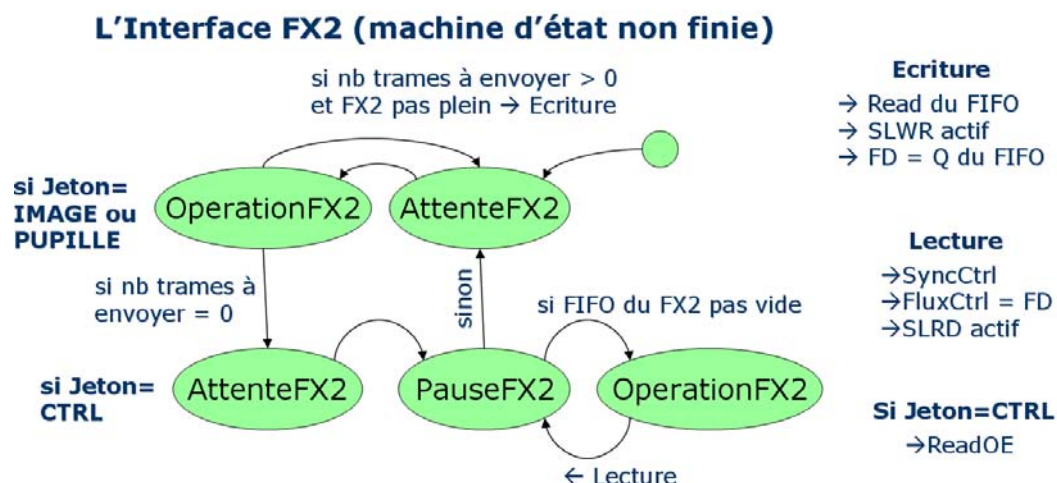
5.10.2 But du système

Il nous est nécessaire de communiquer avec le microcontrôleur FX2 de la façon qui lui convient afin d'être compris.

Dans ce but ce module permet de gérer le flux produit par notre logique de traitement pour l'envoyer en direction du FX2. Il est également capable de recevoir un flux d'ordre de la part du logiciel pour le retransmettre à la logique de contrôle qui l'exécutera.

5.10.3 Fonctionnement détaillé

Etant en pleine finalisation, je ne présente pas l'algorithme mais cette machine d'état :



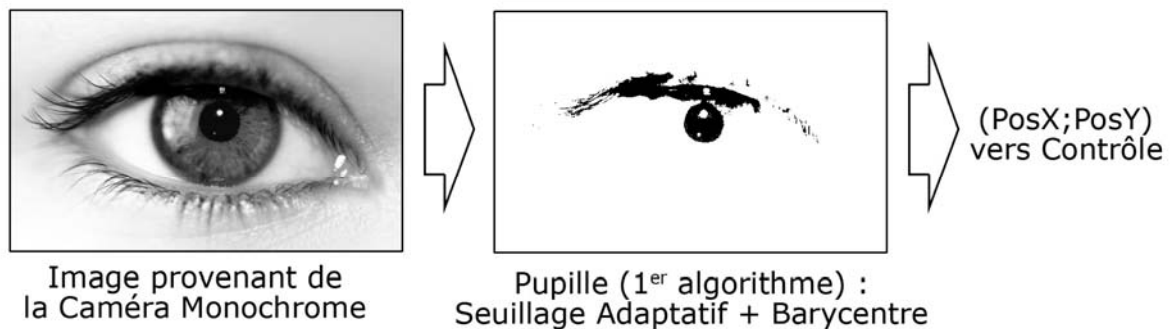
5.11 Pupille

5.11.1 Introduction

Un système équipé de caméras ne jouera pas de façon optimale le rôle des yeux si l'on ne prend pas en compte le fait que ceux-ci peuvent bouger dans leur orbite.

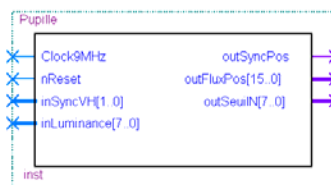
5.11.2 But du système

C'est pourquoi ce système a adopté une façon de simuler cette capacité qui permet à la personne non-voyante d'indiquer la partie de son champ visuel qu'il voudrait « voir ». Pour ce faire, nous filmerons l'œil de la personne pour en déduire, après avoir traité l'image de l'œil de façon appropriée, la position de la pupille ce qui, en quelque sorte, nous indiquera « quoi montrer ».



Pour ce faire, j'ai développé un premier algorithme (qui n'a pas la prétention d'être optimal) qui calcule le barycentre des pixels de l'image qui sont pris comme sombres (entrant dans le seuil). Le niveau du seuil est réadapté à chaque fin d'image en prenant en compte le fait que la pupille devrait remplir un certain pourcentage de l'image.

5.11.3 Fonctionnement détaillé



inSyncVH : pack des signaux de synchros en entrée

inLuminance : pixel (monochrome) de l'image en entrée

outSyncPos : indique l'état de la sortie (nouvelle position ?)

outFluxPos : pack des positions (sur l'axe X et Y) en sortie

outSeuil : seuil utilisé par l'algorithme (en sortie)

Le module est composé du processus synchrone et d'un seul processus combinatoire.

Processus de Barycentre

- si $inSyncVH(V, H) = (0, 0)$ c'est une pause entre deux images
 - Initialisation des registres du positionnement en X&Y
 - Initialisation des registres du barycentre (somme des X,Y,P)
 - si en plus V (1 → 0) est en flanc descendant (c'est une fin d'image)
 - Initialisation du pipeline de division → entrée = SommeX
 - Mémorisation des registres du barycentre
- si $inSyncVH(V, H) = (1, 1)$ alors c'est un nouveau pixel !
 - si la luminance du pixel entre dans le seuil (plus petite ou égale à)
 - Ajouter la position de ce pixel dans la SommeX, SommeY, SommeP++
 - Incrémentation du registre de positionnement en X (X++)
- si $inSyncVH(V, H) = (1, 0)$ c'est une pause entre deux lignes → Y++

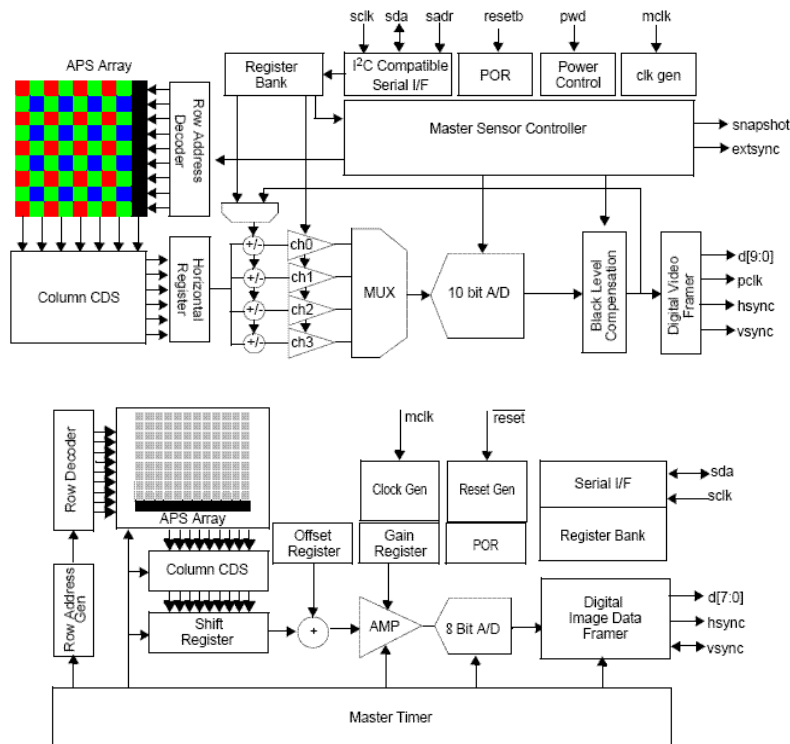
S'en suit une partie traitant du pipeline qui permet d'utiliser celui-ci pour faire les divisions du pipeline (X/Y) et le calcul du %age de pixels pris pour réajuster le seuil !

5.12 Contrôle

5.12.1 Introduction, la configuration des caméras

Le protocole I²C définit une manière sériele d'accès aux périphériques, et nos caméras utilisent celui-ci pour permettre d'accéder à leurs registres internes de configuration.

Voici les schémas 'interne' de nos caméras couleur KAC9648 et monochrome KAC-9630 :



5.12.2 But du système

Comme son nom l'indique ce module sert à

- coordonner les différents étages du traitement pour les configurer de telle manière qu'ils fonctionnent comme demandé par l'utilisateur (flux ordre du FX2)
 - ZonesBayer : sélection du type de mire
 - RVB_ou_HSL : sélection du type de flux
- configurer les caméras par le biais de GestionI²C
 - Paramètres d'initialisation
 - Paramètres du fenêtrage, etc...

5.12.3 Fonctionnement détaillé

Au moment où j'écris ce mémoire la logique de contrôle n'est pas encore dans sa version définitive, c'est pourquoi j'ai omis de parler de toute l'algorithmique du module.

Par défaut nos caméras fonctionnent en mode slave, c'est-à-dire qu'elles attendent des syncros de notre part, c'est pourquoi je configure un certain nombre de registres pour avoir ceci :

Reset de la KAC-9648

- RESET
- MASTER
- POLARITÉ (des synchronisations)

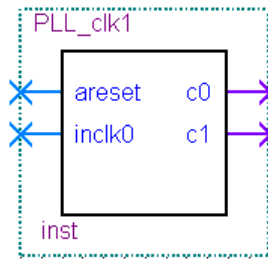
Reset de la KAC-9630

- MASTER

Gestion KAC-9648

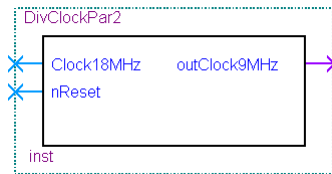
- FENETRAGE

5.13 Divers Modules Utiles

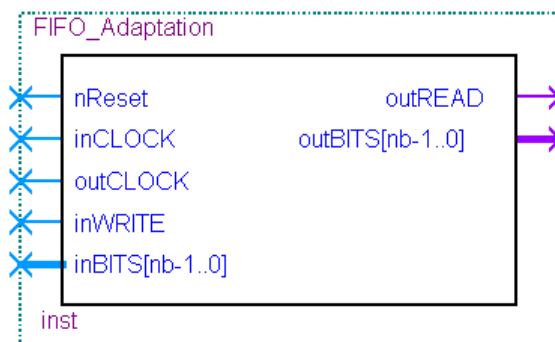


Dans une FPGA il est possible de créer un multiple d'une certaine horloge (provenant d'une des broches prévues à cet effet)

La fréquence de fonctionnement du FX2 étant de 48 MHz, je pourrai créer un sous multiple pour la caméra couleur (24 MHz). Je ferai pareil pour la caméra monochrome, cependant, il est impossible de générer les 9 MHz à partir de la source en 48 MHz ! C'est pourquoi la PLL me génère un 18 MHz que je diviserai par 2...



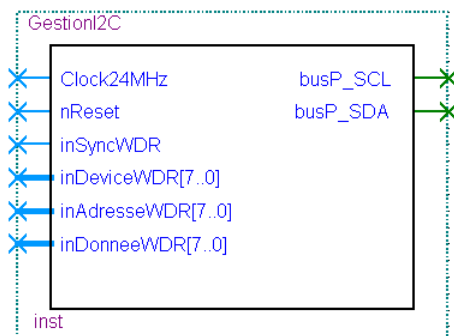
Ce module (un tout petit bout de code VHDL) me permet de diviser l'horloge primaire 18 MHz pour produire les 9 MHz nécessaires au bon fonctionnement de la caméra monochrome.



Comment faire communiquer deux systèmes fonctionnant à des horloges différentes ?

La réponse : ce module intègre une mémoire FIFO multi-clock permettant de le faire !

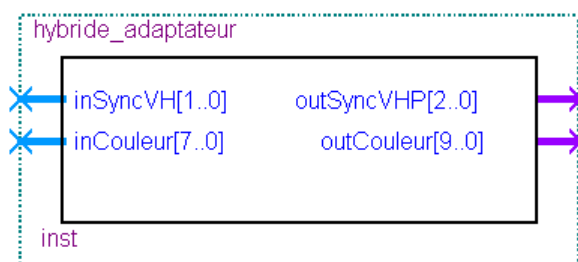
A chaque fois que ce module voit que sa mémoire FIFO n'est plus vide (écriture effectuée en entrée) il lui ordonne de sortir la donnée par le biais du port de sortie.



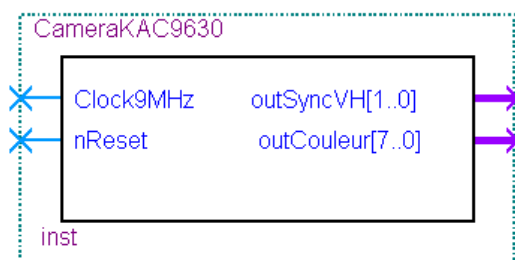
Le module permettant de communiquer par le biais du protocole I²C m'a été généreusement fourni !

Cependant, il m'a été utile de créer un module « de plus haut niveau » simplifiant l'utilisation de cette norme dans le cas de la programmation de registres (caméra). La version présente ici traite → l'écriture.

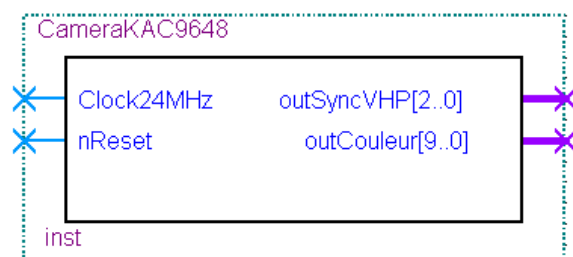
→ L'algorithme a été développé en traduisant du code C++ séquentiel en machine d'état VHDL



Ayant réussi, par le biais de l'I²C, à initialiser la caméra monochrome et essuyé un échec (maintenant réglé) du côté de la caméra couleur, j'ai décidé de créer un projet hybride traitant les images de la caméra monochrome comme si elles provenaient d'une caméra couleur, dans le but de finaliser mon interface logiciel...



... fournissent tout ce qu'une caméra pourrait produire (une image et des synchros) !



5.14 Etude des capacités limites de notre système

Dans le but de connaître les limites de notre système, je vais effectuer une petite étude des contraintes mémoire et de débit tout au long du traitement.

Premièrement nous devons prendre en compte certaines caractéristiques propres à l'implémentation :

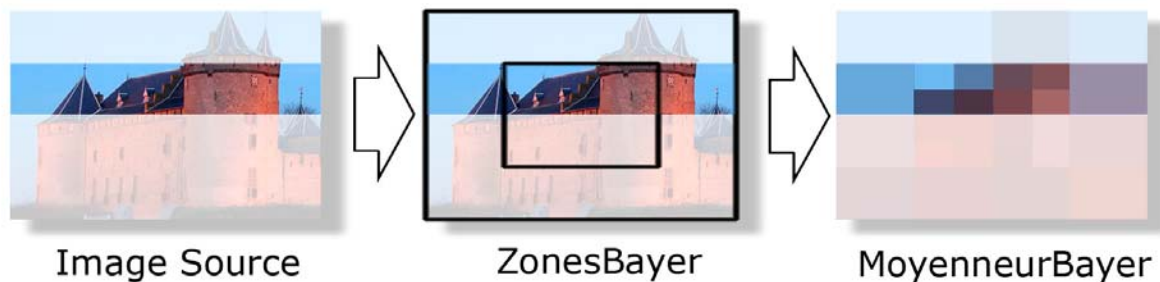
- Fréquence de fonctionnement de la FX2 : IFCLK → 48 MHz
- Fréquence de fonctionnement de la caméra couleur : mclk_c → 24 MHz
- Fréquence de fonctionnement de la caméra monochrome : mclk_p → 9 MHz

Dès lors voici un résumé des limites propres à chacun des modules :

ZonesBayer ne posera aucun problème !

MoyenneurBayer est constitué de **mémoire** ce qui nous oblige à en choisir la quantité.

Remarque : Il ne faut pas oublier que ce calcul ne fait que représenter le cas bien précis où ZonesBayer nous demande d'effectuer notre réduction d'une certaine façon.



Pour donner quelques chiffres je vais prendre un exemple réaliste d'après les caractéristiques de mon projet : une image provenant de ma caméra avec un champ de vision de 180°, avec un fenêtrage représentant 60° des 180° de l'image captée avec une zone détaillée représentant un angle 15°.

La ligne qui demande le plus de mémoire est celle qui contient des pixels faisant partie de la zone périphérique mais aussi de celle dite détaillée. Dans ce cas précis la quantité requise se calcule de la manière suivante :

$$\text{TailleEnX} = 420 / \text{MoyPerX} = 6 / \text{MoyDetX} = 3 / \text{MoyDetDebX} = 96 / \text{MoyDetFinX} = 323$$

- *Nombre de pixels dans la zone détaillée*
 $\text{NbDetX} = (\text{MoyDetFinX} - \text{MoyDetDebX} + 1)$ → 228 pixels
- *Nombre de pixels dans la zone périphérique*
 $\text{NbPerX} = (\text{TailleEnX} - \text{NbDetX})$ → 192 pixels
- *Nombre de pixels total à stocker en mémoire pour cette ligne*
 $\text{NbPixels} = \text{NbDetX} / \text{MoyDetX} + \text{NbPerX} / \text{MoyPerX}$ → 108 mots

De plus le calcul de la division donnant la moyenne m'a obligé à utiliser un module de **division en pipeline**. Dans un souci de « souplesse » mon VHDL contient un autre paramètre qui définit la profondeur du pipeline mis en jeu : ce qui veut dire que nous devons prendre une décision ! Et d'après mes résultats une profondeur de 1 est largement suffisante pour fonctionner à la fréquence de la caméra qui est de 24 MHz.

Bayer_a_RVB doit également stocker certaines lignes en **mémoire FIFO**, comme celles-ci proviennent de MoyenneurBayer la quantité de mémoire nécessaire est tout simplement la même.

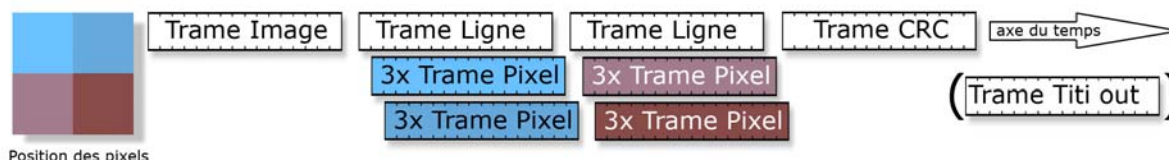
RVB_a_HSL effectue certains calculs, notamment des **divisions** et la profondeur du pipeline nécessaire pour celle-ci est de 4.

J'ai pris ce chiffre comme référence car dans l'avenir du projet la FPGA devra traiter le flux provenant de deux caméras, et, de ce fait, la quantité d'éléments logiques nécessaires pourrait être réduite si mon successeur utilise ce module en multiplexage (donc RVB_a_HSL devrait fonctionner deux fois plus rapidement), ce qui laissera de la place en FPGA pour d'autres algorithmes !

RVB_ou_HSL ne posera aucun problème !

ProtocoleTITI posera le plus de problèmes potentiels au niveau du flux !

Car chacun des pixels nous provient sous la forme RVB ou HSL, ce qui implique le fait d'envoyer trois trames contiguës par pixel !



Nous devons, pour éviter de saturer la **mémoire FIFO**, envoyer les trames correspondant à une ligne dans un délai plus court que la durée d'une ligne entière (avec les temps de repos).

Remarque : Etant donné que dans notre cas MoyenneurBayer augmente le nombre de coups d'horloge entre deux pixels sortants, (en passant de 1 pclk entre deux pixels à 3 ou plus...) et que Bayer_a_RVB augmente encore cet écart d'un facteur de 2, nous n'aurons pas vraiment de soucis.

Sachant qu'une ligne est formée d'une trame ligne et de trois trames pixel (pour chacun des pixels de la source), il nous est dès lors possible de quantifier la limite :

Pour l'exemple, $NbPixels/2 = 108$ (maximum), $TailleEnXd = (TailleEnX+d)$ $d=8 = 428$

d = Temps de repos suivant la caméra...

- | | | |
|--|--------------------------------|--------------|
| ➤ <i>Nombre de pixels dans une ligne</i> | $NbPixels' = NbPixels/2$ | → 54 pixels |
| ➤ <i>Nombre de trames par ligne</i> | $NbTrames = 1+3 \cdot TailleX$ | → 163 trames |
| ➤ <i>Temps total à disposition</i> | $TMax = TailleEnXd/mclk_c$ | → 17,83 us |
| ➤ <i>Temps total nécessaire</i> | $TMod = NbTrames/IFCLK$ | → 3,40 us |
| ➤ <i>Condition de non saturation</i> | $OK ? = TMod < TMax$ | → OK |

Remarque : Il est inutile de calculer la quantité de mots nécessaires en mémoire FIFO si le temps nécessaire par pixel est moins important que le temps à disposition entre deux pixels.

Cependant voici le calcul qui prouve que le nombre de mots pour ma mémoire FIFO n'est pas fonction du flot de pixels en entrée :

- *Temps de traitement du pixel à disposition* $(3 \times 2) / mclk_c$ → 250 ns
- *Temps de traitement du pixel en zone détaillée* $3 / IFCLK$ → 62 ns !

Dans tous les cas la mémoire FIFO nous est quand même utile car la vitesse de fonctionnement de ce module, égale à la vitesse de l'InterfaceFX2, diffère de celui qui nous fournit les pixels, fonctionnant sur la même horloge que la caméra.

InterfaceFX2 posera le problème suivant : si logiciel se trouvant de l'autre côté du flux USB ne lit pas le contenu des FIFO du FX2, notre module d'interface, en attendant que celui-ci se vide, saturera sa propre FIFO.

Dans ce module et celui nommé ProtocoleTITI j'ai implémenté la gestion du FIFO plein.

Ces deux mémoires FIFO sont gérées de cette façon :

- A partir du moment qu'elle est pleine, le mode alerte est activé
- A partir du moment qu'elle est vidée, le mode alerte est désactivé
- L'accès en écriture de la mémoire FIFO est masqué pendant toute l'alerte

6. Firmware du FX2

6.1 Introduction, notre configuration

Le microcontrôleur FX2 est capable d'exécuter un mini logiciel qui nous permet de configurer celui-ci et exécuter certains traitements (si nécessaire) sur le flux entre les entrées/sorties FIFO \leftrightarrow sorties/entrées USB.

C'est pourquoi je vais programmer un firmware spécifique à mon projet.

L'horloge utilisée par le FX2 est produite en interne et tourne à 48 MHz ; celle-ci est accessible par la broche nommée IFCLK que l'on va utiliser pour « driver » la FPGA.

Nous utilisons le FX2 de cette manière (tous en flux Bulk) :

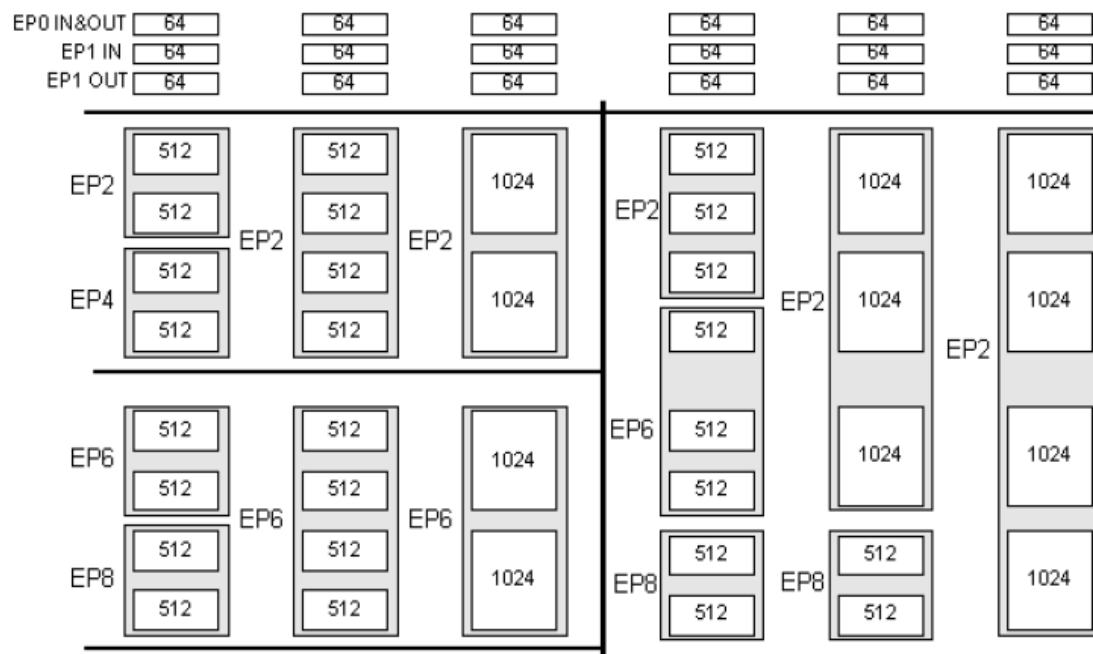
- Endpoint 2 en IN (FPGA→PC) transfert de trames images sur 16 bits
- Endpoint 4 en OUT (PC→FPGA) récupération des ordres logiciel
- Endpoint 6 en IN (FPGA→PC) transfert de la position de la pupille
- Endpoint 8 - libre d'accès

J'ai fait le nécessaire pour que le FX2 ne fasse aucun traitement sur les données et donc celui-ci fonctionne en Auto-IN et Auto-OUT et passent par le biais des FIFO d'interface.

Ce qui veut dire que le FX2 fera le nécessaire pour transmettre le contenu du FIFO dès qu'une certaine quantité (paramétrable) de mots est stockée.

Finalement, entre le FX2 et la FPGA la communication se fait de manière synchrone.

Voici un schéma de la part du datasheet représentant les modes des FIFO :



7. Interface PC

7.1 Introduction, CypressEzUSB

Une des deux bibliothèques utilisées au cours de la création de ce logiciel est CypressEzUSB, celle-ci contient quelques fonctions utiles dans l'utilisation d'un périphérique ayant un microcontrôleur FX2 et donc connecté en USB.

→ virtual bool **open** (const QString &firmwareFilename);

Permet de télécharger le firmware provenant d'un fichier bix dans la Cypress FX2

virtual bool close (void);

virtual bool setInterface (unsigned number, unsigned alternateSetting);

→ virtual unsigned **bulkRead** (unsigned pipeNum, char *buffer, size_t size);

Permet d'effectuer une lecture bulk du FIFO d'un certain endpoint du FX2 et d'y stocker dans un buffer de notre programme

→ virtual unsigned **bulkWrite** (unsigned pipeNum, const char *buffer, size_t size);

Permet d'effectuer une écriture bulk du contenu d'un buffer de notre programme en direction du FIFO d'un certain endpoint du FX2

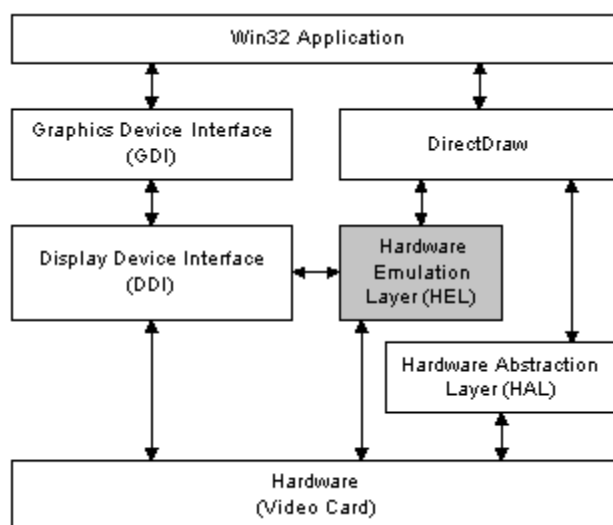
virtual bool GetHandle (void);

virtual void setOverlapped (bool Value);

7.2 Introduction, DirectDraw

Etant donné que l'ambition du logiciel est d'afficher une vidéo (une suite d'images en temps réel) il nous faudra un moyen simple et rapide d'accéder à la mémoire vidéo : ce que DirectDraw fait à merveille (sous Windows) !

Cette bibliothèque est un des éléments de DirectX, conçu par Microsoft, qui est dédié à la gestion de l'affichage graphique en 2D (accès mémoire vidéo & configuration & co).



Ce moteur graphique 2D permet aux développeurs de jeux (entre autres) de profiter des performances des cartes graphiques actuelles en tirant profit des fonctionnalités graphiques spécifiques aux cartes graphiques.

Une des fonctionnalités puissante de cette API est le double buffering (le fait de calculer / modifier l'image dans une mémoire vidéo hors écran pendant qu'une autre zone mémoire est affichée) et le page flipping (affichage en alternance d'une des deux zones).

← Voici une représentation en couche

Bien entendu ceci n'est qu'une introduction non exhaustive et je ne vais pas m'étaler d'avantage sur les fonctionnalités (ceci n'est pas une pub).

DirectDraw est orienté objet, voici la liste des objets faisant partie de cet API :

| <i>Objet</i> | <i>Résumé de sa fonction, son utilisation courante</i> |
|--------------------------|---|
| DirectDraw | Cet objet représente la base ; il permet de modifier le mode vidéo, de détecter les capacités de notre carte graphique, de gérer la mémoire vidéo ainsi que de gérer les objets enfants. |
| DirectDrawSurface | Cet objet est largement utilisé ; il représente une surface image, ce qui veut dire que cet objet est autant utilisé pour la mémoire vidéo que pour les images stockées en mémoire ! DirectDrawSurface nous permet d'effectuer des animations non saccadées (double buffering), de faire des overlays graphiques : et tout ceci peut être interfacé avec le standard GDI. |
| DirectDrawPalette | Certains modes graphiques sont représentés par une palette... |
| DirectDrawClipper | Cet objet nous permet de créer une application graphique sans se soucier du mode utilisé, plein écran ou fenêtré. Ce qui implique que le contenu produit devra être « redimensionné » ou rendu (c'est selon) dans une petite partie de l'écran : sur la fenêtre. |

7.3 But du logiciel

Notre logiciel a pour but d'afficher le résultat du traitement de l'image, d'indiquer la position calculée par l'algorithme de traitement de la pupille et de configurer le système embarqué développé au cours de ce projet de diplôme.

Une fois encore, la finalisation se fera après ce mémoire, et donc je n'ai pas de capture d'écran très concrète à vous montrer...

J'écirai un petit mode d'emploi s'il est nécessaire !

8. Développements futurs

8.1 L'interface parallèle - série envisagé

Une interface parallèle série (dans notre cas suivi d'un autre en série parallèle) permet de correctement transmettre sur un faible nombre de conducteurs (généralement deux) les données provenant d'un composant envoyant les données de façon parallèle.

Par exemple les données de couleur produites par nos une seule de nos caméras seraient, sans interface sériel, transmises par 10 conducteurs car chaque pixel est constitué de 10 bits, que nous voulons acheminer vers notre calculateur situé plusieurs dizaines de centimètres plus bas : un véritable sac de nœud de fils !

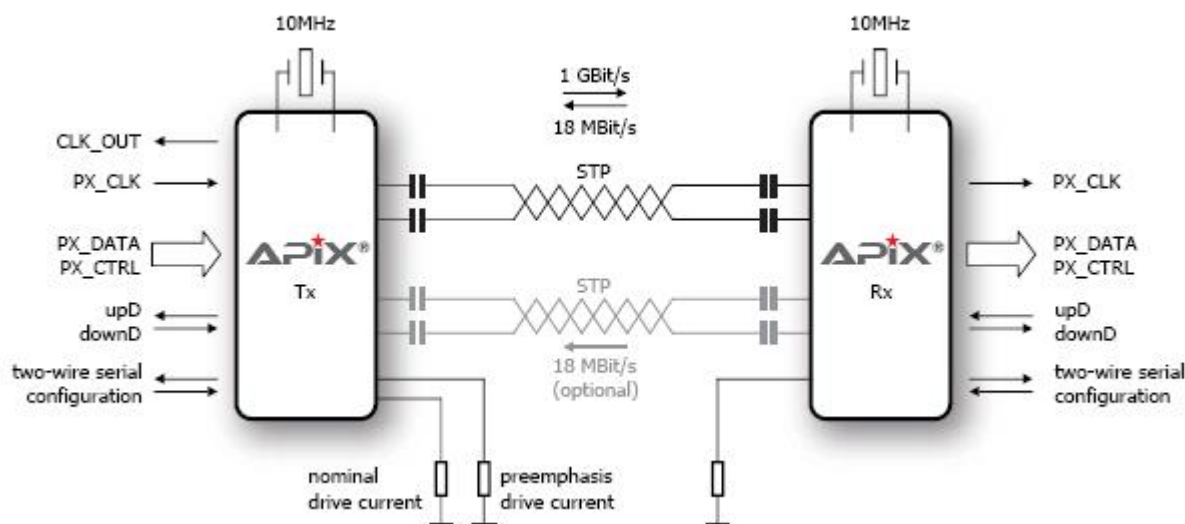


Figure 1 – Illustration du système de sérialisation (retenu) de la compagnie inova

L'APIX INAP125T10 / INAP125R10 (T veut dire transmission, utilisé de la caméra à la nappe sériel et la puce avec un R nous sert à la réception, de cette nappe à notre calculateur) d'Inova permet, chose très utile, de transmettre les signaux du bus I²C !

- PX_DATA gère l'envoi de pixels quantifiés à 10 bits (ou plus)
- PX_CTRL pour gérer la transmission des signaux comme vsync hsync de
- PX_CLK peut-être cadencé de 6 à 60 MHz
- Bande passante de 1000 Mbps en down → et de 18 Mbps en up ←→
- Le sens « upload » servant au bus I²C contrôlant la caméra connectée...
- Oscillateur intégré permettant d'utiliser un quart à bas prix

Cependant j'ai, après avoir pris contact avec l'entreprise, découvert que le prix du engineering board était trop important (1'960 €).

9. Conclusion

9.1 Ce qui a été réalisé

Tout premièrement je citerai l'étude du problème de façon globale, c'est-à-dire le fait de bien réfléchir pour être capable de décomposer la difficulté en un nombre assez important de sous solutions. Dans mon cas, de créer les modules à implémenter dans notre FPGA.

Toutes les implémentations réalisées au cours de ce projet de diplôme sont complètement opérationnelles ; elles répondent aux exigences du traitement temps réel et aux contraintes liées aux caractéristiques de la FPGA.

Les premiers tests effectués avant la remise de ce mémoire me font déjà entrevoir que le résultat final sera en accord avec le cahier des charges et les ajouts (zones+moyenneur) que M. Vinckenbosch et M. Bologna m'ont proposé de développer.

Etant un sujet de recherche je suis sûr que ce développement apportera son lot de connaissances.

9.2 Ce qui sera prochainement réalisé

Me restant encore une semaine et demi, je passerai mon temps à :

- Implémenter la gestion du flux d'ordre dans InterfaceFX2 → VHDL
- Concevoir le code réalisant l'ordre dans Contrôle → VHDL
- Implémenter l'I²C pour initialiser la caméra couleur et la fenêtrer → VHDL
- Finaliser le logiciel que l'utilisateur puisse effectivement envoyer l'ordre ! → C++

Ce qui se fera, je l'espère dans le délai imparti.

9.3 Perspectives d'avenir

De nombreuses façons d'utiliser ce système sont possibles.

Dans le futur, le traitement des données provenant de deux caméras couleurs serait bien mieux implémenté si l'on utilisait RVB_a_HSL de façon multiplexé pour éviter de doubler le nombre de composants utilisés par ce module très lourd.

Il serait aussi possible de continuer la chaîne de modules avec des algorithmes dédiés à la conversion image → son pour réaliser l'objectif principal qui est de simplifier la vie des personnes aveugles en leur substituant l'ouïe à la vue.

Une autre manière de modifier mon projet serait de remplacer ZonesBayer par un algorithme de création de zones plus complexe, MoyenneurBayer étant capable de réagir en conséquence.

9.4 Remerciements

Je tiens à remercier mon professeur, monsieur Jacques Tinembart pour ses conseils éclairés ainsi que toute l'équipe pour l'ambiance générale dégagant du laboratoire de systèmes numériques. Je garderais un très bon souvenir de mes camarades de la classe de télécommunication TE3 ainsi que de cette école à la fois sérieuse et conviviale. Merci à ma famille d'avoir été et d'être là pour moi...

Et merci à vous de m'avoir « lu » !

David FISCHER

