Université Paul Sabatier

Chasse au lapin avec une carte Altéra

Projet de l'année 2007-2008

Cornet Guillaume – Matos José Mauro

Table des matières

Introduction:

Dans le cadre de notre préparation au diplôme de licence en ingénierie électrique, nous avons été amenés à effectuer notre projet de troisième année sous l'encadrement de nos professeurs.

En cours d'année le choix de nos projets nous a été présenté. Ces projets étaient divisés en deux groupes. Un groupe de projet plutôt a tendance électrotechnique et d'autres plutôt à tendances électroniques. Le point commun entre tous était le faite qu'il faisait tous appel à la programmation, sous différent langages, pour être réalisé. Nous avons le projet qui à l'époque nous paraissait le plus intéressant, par son contenu mais aussi il faut l'admettre par son nom ; « Chasse aux lapins avec une carte altéra ».

Il s'agissait au cours de ce projet d'étudier et de développer un système capable de détecter un passage, puis envoyer un signal de commande numérique à un caméscope afin que celui-ci enregistre l'objet déclencheur du passage. Avec l'avènement de l'informatique et l'électronique numérique, des systèmes embarqués sont en plein développement. Ainsi dans les prochaines années, les connaissances acquises durant la réalisation de ca projet pourrons se révéler très utiles. Dans la mesure où durant notre cursus antérieur nous avions développé très peu d'outils tel que celui que nous nous proposons de vous présenter existent, il paraissait d'autant plus intéressant d'en développer un.

Par ailleurs, avant le début de projet nous avons été informé que le caméscope était indisponible au moment du projet, donc il a fallut modifier celui-ci. Le cahier des charger fut modifié et il fallait maintenant réaliser le même projet mais avec une caméra ALTERA et il nous faillait maintenant gérer l'enregistrement et l'affichage.

Des le début nous avons décidé le travail en deux partie égale. Guillaume : Son, gestion caméra. José : Mémoire et affichage. Nous n'aurions pas pu prédire au début, que le projet allé se révéler aussi complexe donc tout le travail n'a pas pu être terminé.

Ce rapport s'articule ainsi en quatre parties. La première partie est constituée de la présentation du projet, des objectifs de départ et de la solution technique retenue. La seconde partie propose une étude de la mémoire. La troisième partie une étude de l'affichage. Finalement la quatrième partie est consacrée à l'étude de l'application sonore de la carte ALTERA DE1.

Cahier des charges et organisation du travail :

Schéma fonctionnel de 1er degré :

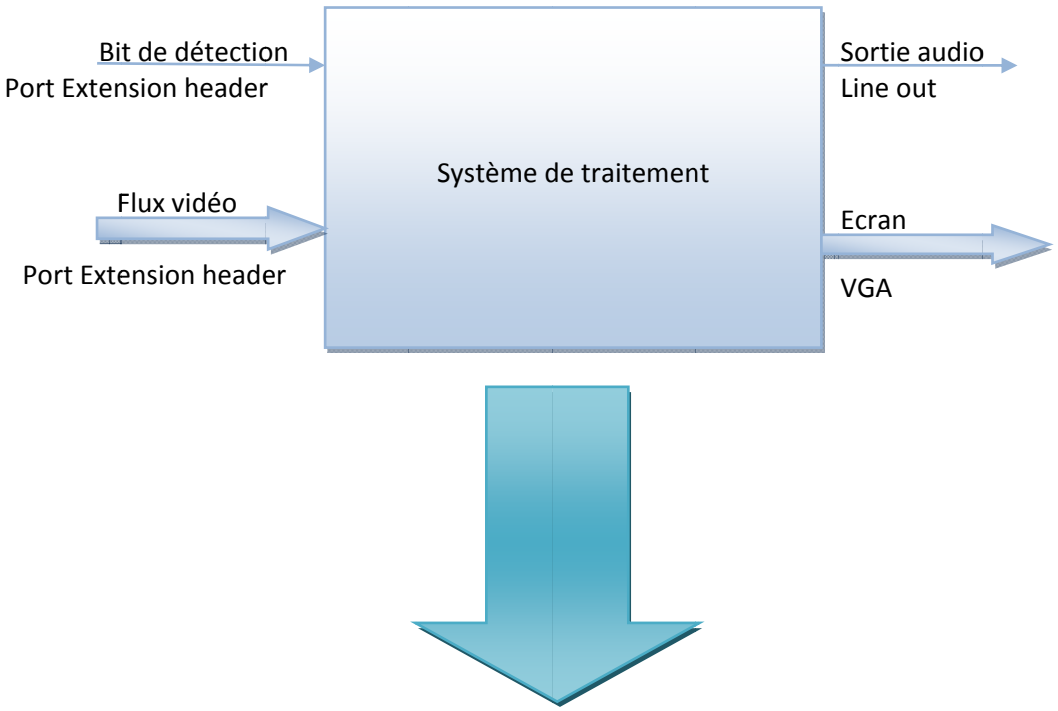
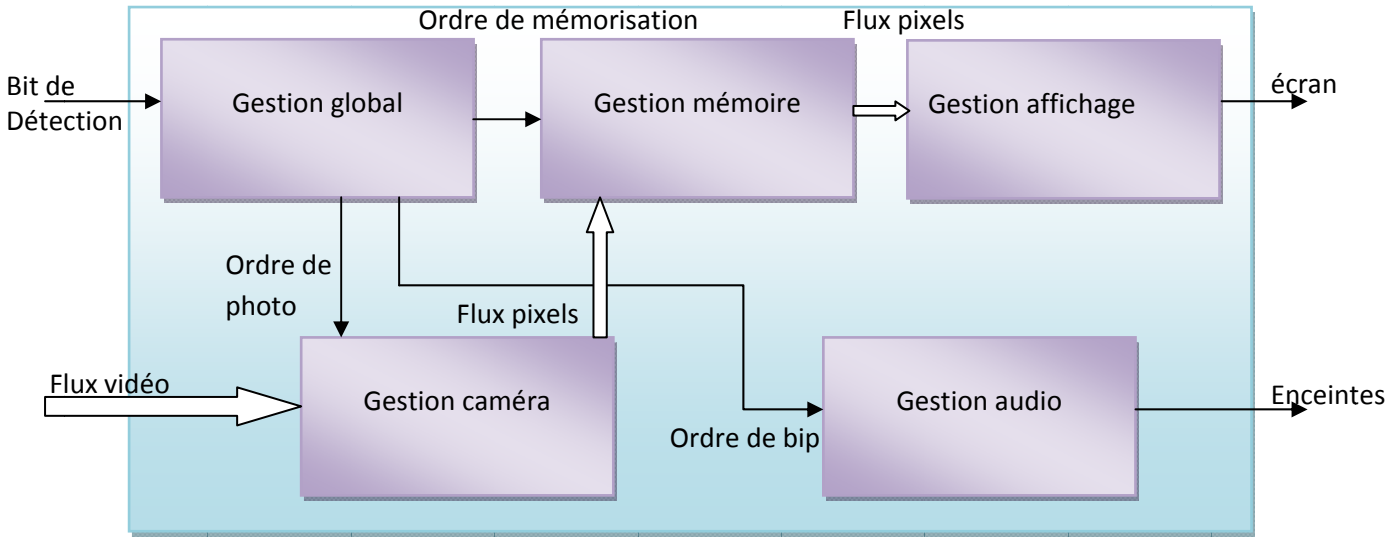


Schéma fonctionnel de 2nd degré :



Mémoire

Introduction

Pour la réalisation d'un projet informatique un minimum complexe, l'utilisation de la mémoire pour le stockage de données s'impose. L'utilisation de registre de type tampon présent dans notre FPGA* nous permet de sauvegarder très peu d'information du à la réalisation de ceci. Créer un registre tampon demande un certain nombre de portes logiques. Le nombre de celle-ci étant limité 262000 portes logiques, sur un le Cyclone II*. Prenons l'exemple de ce FPGA pour démontrer la quantité maximum de données pouvant être sauvegardées sur celui-ci en utilisant toute sa capacité pour la mémorisation (pas de fonctionnement autre que la sauvegarde de données). Une bascule D* ou « flip-flop» est constituée au minimum de 6 portes logiques du type NAND*, donc un calcul très approximatif nous permet de conclure que la quantité maximal de données mémorisées dans un FPGA est environ de 262000/6 donc 43 Kbits soit 5.3 Ko. Sachant que le but de notre projet est de mémoriser une photo au format VGA*, enregistré sous un format non compressé RGB*, à une taille de 640*480 pixels*. Donc en admettant que chaque couleur (rouge, vert et bleu) de ce pixel est codée sur 4 bits, donc il nous faudrait 12 bits pour représenter un pixel. Un simple calcul nous permet de connaitre la taille d'une photo en format VGA-RGB : 640*480*12 = 3685400 bits soit 460 Ko. On peut en conclure très facilement que l'utilisation de la mémoire s'impose pour notre projet.

Un autre choix s'impose à nous, le choix de la mémoire à utiliser pour réaliser notre projet. Sur notre carte ALTERA DE1* nous disposons de 3 types de mémoires différents : SDRAM*, SRAM* et FLASH*. Toutes ont leur propre capacité, mode fonctionnement, timings*, et fréquence de fonctionnement. Mais elles sont toutes accessibles à partir du FPGA.

Cette partie du rapport portera sur l'étude que j'ai porté sur chacune de ces mémoires : lecture de la documentation technique (pas toujours facile à lire), l'analyse du fonctionnement, détermination des vitesses maximales de fonctionnent, respect des temps d'accès et finalement leur possibilité d'intégration dans notre projet.

[1] ** : Pour toutes explications se référer à l'explicatif de la page suivante.*

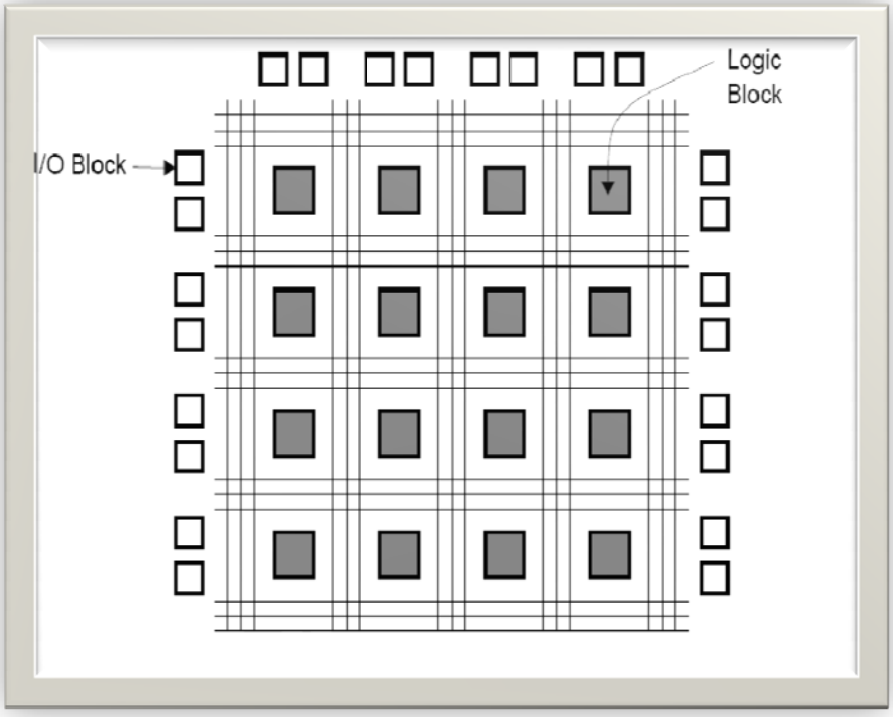
Explicatif

Avant de commencer la description du projet je voudrais commencer par un petit passage explicatif des termes utilisés, pour permettre au lecteur d'avoir toutes les explications regroupées.

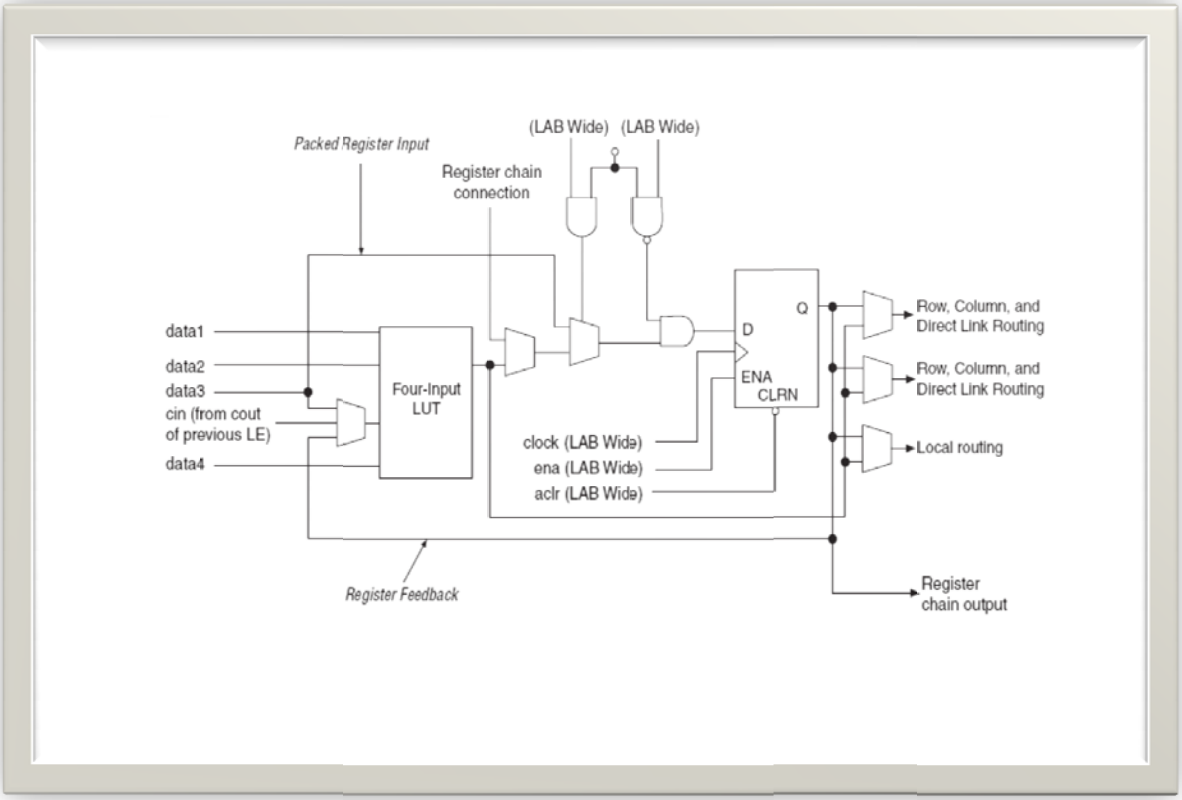
- FPGA

Un FPGA (Field-Programmable Gate Array) est un circuit intégré logique qui peut être reprogrammé après sa fabrication par l'utilisateur grâce à un compilateur optimisé pour le composant étudié. La programmation peut se faire dans n'importe quel langage informatique connu, tout dépend du compilateur. Un bloc logique est de manière générale constitué d'une table de correspondance et d'une bascule D. La table de correspondance sert à implémenter des équations logiques ayant généralement 4 à 6 entrées et une sortie. Elle peut toutefois être considérée comme une petite mémoire, un multiplexeur ou un registre à décalage. Le registre permet de mémoriser un état (machine séquentielle) ou de synchroniser un signal. Les blocs logiques sont organisés en une forme de filet avec des lignes d'interconnections entre bloc logiques qui se croise orthogonalement comme les avenues dans les grandes villes modernes. Un FPGA contient aussi d'autre éléments internes : des multiplieurs (en nombre très réduit car très dur a implanter), des PLL, des blocs mémoires et même pour certain un microprocesseur. L'avantage du FPGA par rapport au microprocesseur et au PIC, est que celui-ci réalise tout les calculs simultanément et non instruction par instruction comme le fait un microprocesseur. L'inconvénient majeur et que le nombre d'instructions maximum pouvant être réalisées par un FPGA est déterminé par la taille de celui-ci.

Voici une image représentant succinctement l'architecture interne d'un FPGA :



Voici maintenant la représentation d'un bloc logique présent dans le FPGA que nous utilisons sur notre carte.



- Cyclone II

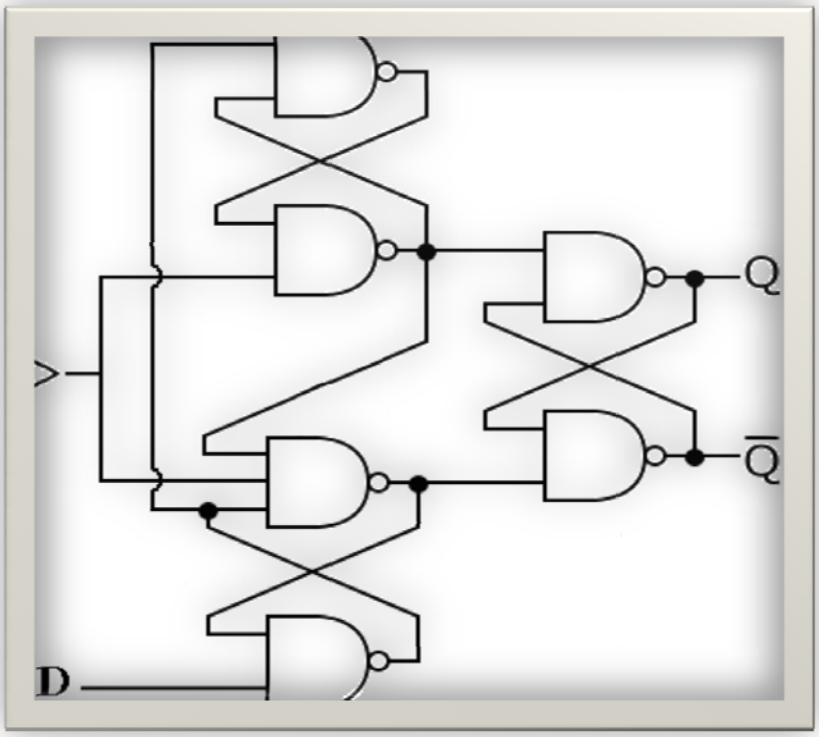
Le FPGA utilisé pour notre projet est un Altera Cyclone II. Un FPGA récent et moderne d'un niveau milieu de gamme. Il est contient : 18000 unités logiques, 52 blocs mémoires de 4Kbits, 4 PLL, 26 multiplieurs et un microprocesseur interne du type NIOS II. Le tout gravé à 90nm. Ce circuit est largement assez puissant pour nous permettre de réaliser l'intégralité de notre projet.

Voici une image du circuit.

- Bascule D ou « flip-flop »

C'est une fonction logique qui permet de sauvegarder pendant un certain temps les données présente à son entrée. Les données en entrée son recopié sur la sortie à chaque front montant de l'horloge. Tant qu'il n'y à pas de front d'horloge la bascule garde la même valeur sur sa sortie indépendamment des variations de l'entrée. Ceci est appelé l'effet mémoire de la bascule. La bascule D est à la base de l'architecture de la mémoire SRAM et de la mémoire tampon des microprocesseurs. Une bascule est réalisée avec des portes logiques (dans notre cas des portes NAND), et celle-ci sont réalisées avec des transistors. Nous pouvons constater alors que les bascules sont la base de l'informatique

Voici un schéma de réalisation d'une bascule D à base de portes logiques NAND.



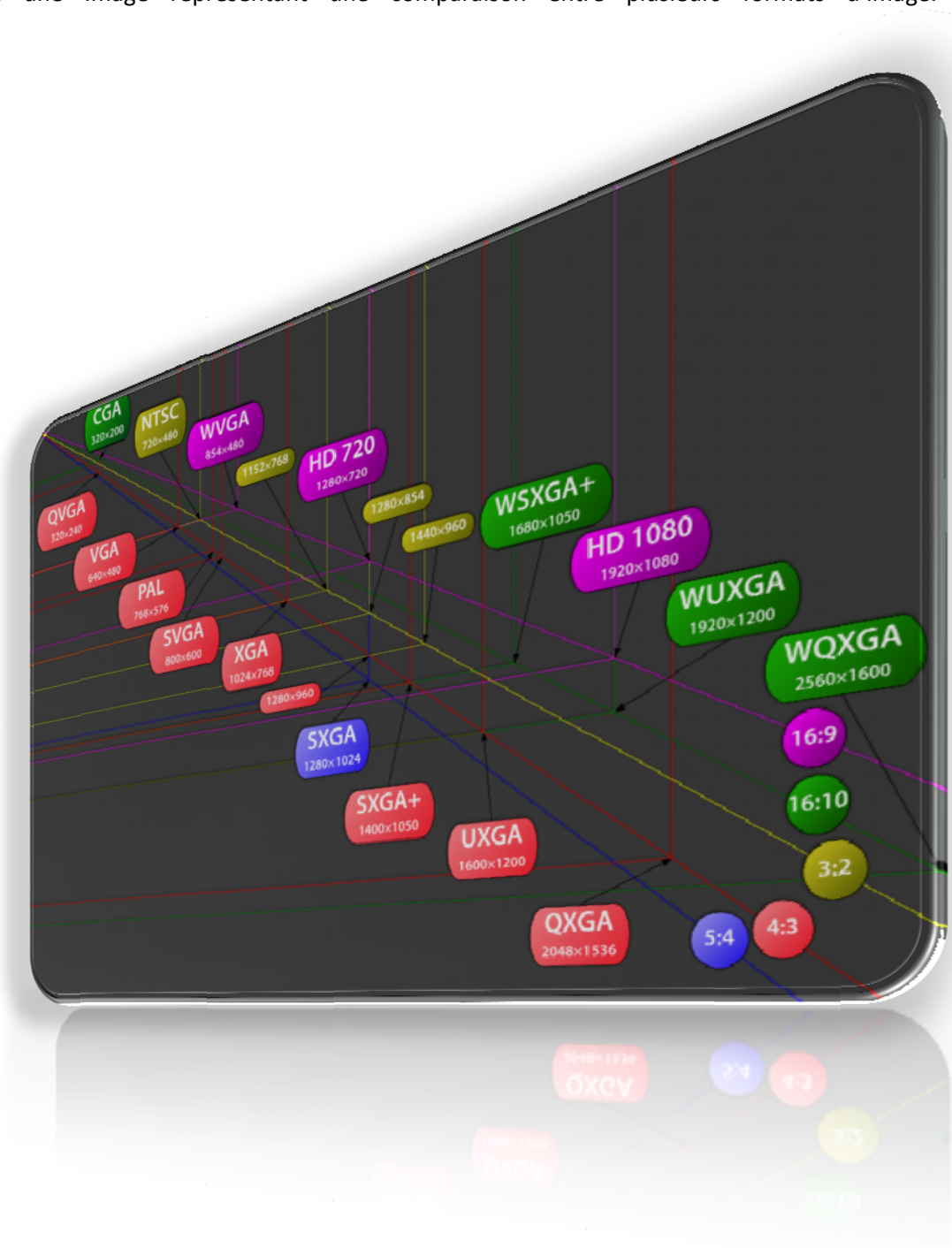
- Portes logiques

Ce sont les composants de base de l'électronique numérique, tout en informatique est réalisé grâce aux portes logiques. Pour connaitre les différents types de porte logique et leur fonctionnement s'adresser aux cours d'électronique numérique…

- VGA

Les images en informatique ont des tailles standardisées qui rendent plus facile la diffusion de celle-ci sur différent support. Chaque tailles ou format d'image à son propre nom, la base étant le format VGA (Video Graphics Array**)** qui a une taille de 640 pixels de largeur sur 480 pixels de hauteur. Tous les autres formats sont des multiples du format de base. Mais VGA est aussi un connecteur pour brancher un composant à un écran selon un transfert d'image expliqué postérieurement.

Voici une image représentant une comparaison entre plusieurs formats d'image.

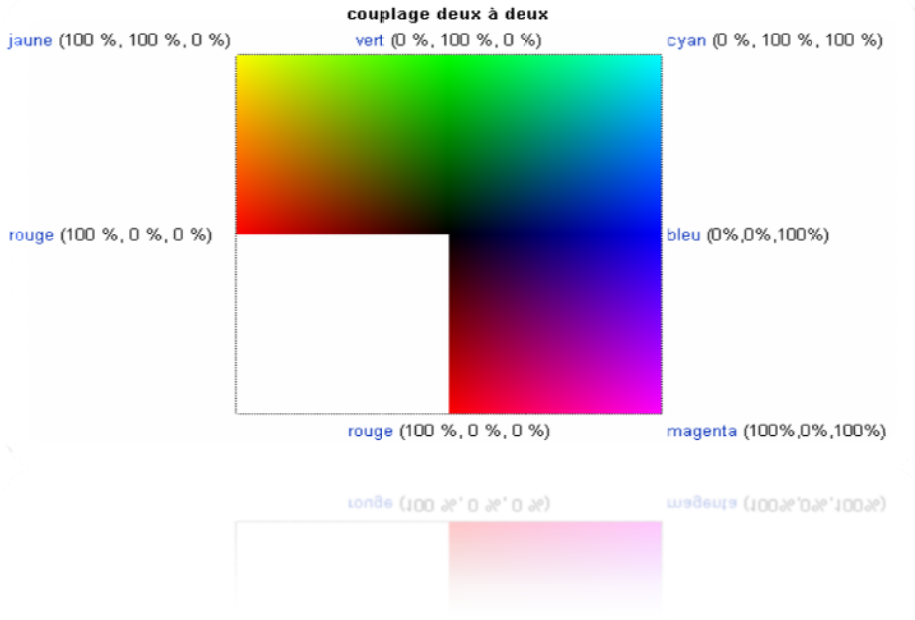


Voici maintenant une image représentant un câble/port VGA.

- RGB

Le RGB (Red Green Blue) est un format de codage de pixel, où chaque pixel est codé par trois composantes de couleur. Ces composantes de couleur sont les couleurs primaires : le rouge, le vert et le bleu. La combinaison de différentes teintes de ces trois couleurs peuvent donner n'importe quelle couleur du spectre lumineux. Chaque couleur est codée par un nombre fini de bit, donc le nombre de teintes est limité par le nombre de combinaisons de bits disponible 2^n, ou n est le nombre de bits. Par exemple si chaque couleur est codée sur 4 bits (comme sur notre carte) donc il existe 16 teintes. Pour un total de 3^{16} couleurs affichable sur l'écran.

Voici un schéma récapitulatif de combinaison des couleurs sous un format RGB.



- PIXEL

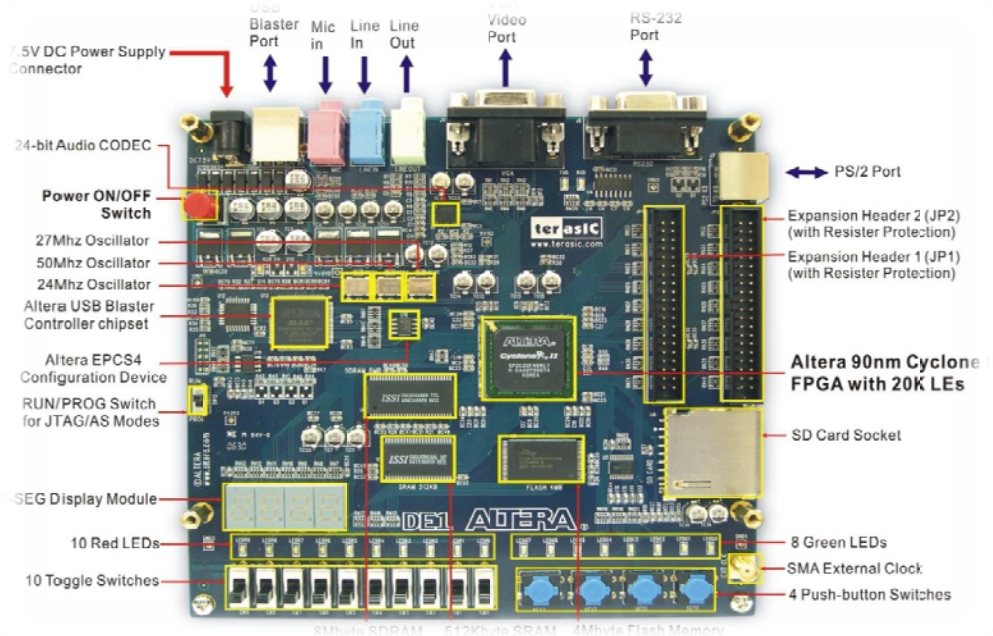
  Un pixel (Picture ELements) est un carré de couleur qui est la base d'une image numérique. En associant plusieurs pixels de couleurs différentes comme sur une mosaïque, on arrive à créer une image. Pour afficher un pixel à l'écran il faut afficher ses trois composantes RGB, et faire varier l'intensité lumineuse de chaque composante pour afficher différentes couleurs.

Voici une image qui montre bien la forme carrée des pixels

- ALTERA DE1

  La carte Altera DE1 est la carte que nous utilisons pour réalise notre projet. Le cœur de la carte Altera est le FPGA Cyclone II, celui-ci est entouré de plusieurs composants (mémoire, contrôleur VGA, contrôleur Audio, port de communication série/parallèle/USB, etc.) Qui nous permettent de réaliser un maximum de fonction.

Voici une photo de la carte avec la description de chaque circuit présent sur celle-ci.



SRAM

Introduction

L'utilisation de la mémoire étant prouvée nécessaire, mon choix c'est d'abord porté sur la mémoire SRAM (Static Random Acces Memory). Celle-ci semblait la plus simple d'utilisation des trois mémoires disponibles.

Description technique

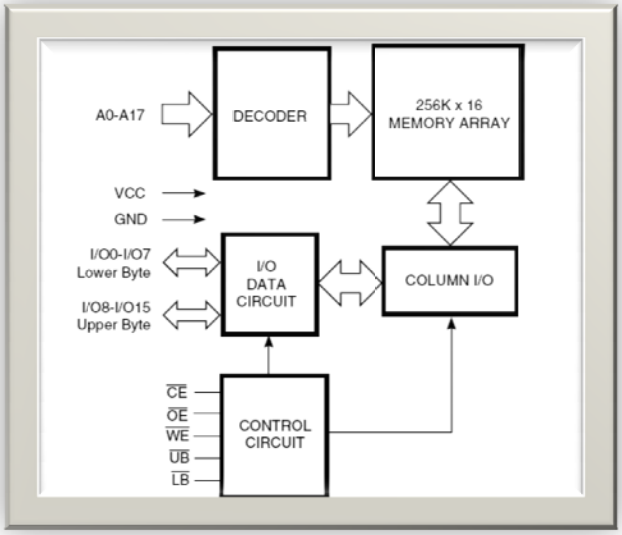
La mémoire SRAM est une mémoire volatile, ceci veut dire que les informations sauvegardées sur cette mémoire ne sont pas conservé après une mise hors tension de celle-ci. Une SRAM est composé de plusieurs centaines de milliers de bascule D. Ceci rend la mémoire très rapide et très facile d'utilisation. Mais rend son intégration très compliquée car une bascule D contient 6 portes logiques et chaque portes logiques contient plusieurs transistors et résistances. Donc au final il faut intégrer énormément de transistor pour réaliser une mémoire SRAM, ce qui la rend très couteuse à la production et à l'achat. Pour cette raison la taille de la mémoire présente est très réduite ce qui se révélera fatal pour la suite du projet.

Le circuit de mémoire SRAM intégré sur notre carte a une taille de 256 Ko. La mémoire n'est pas synchronisée sur une horloge, ceci veut dire que lorsqu'un signal de commande arrive sur la mémoire l'action commence à s'exécuter de suite et non au prochain front d'horloge comme sur un circuit synchronisé. Mais il faut comme même attendre un certain temps avant de pouvoir récupérer les informations en sortie, ceci semble instantané à l'échelle humaine, mais a l'échelle informatique ces temps de réponses compte, et son appelés timings. Ils sont dus aux temps de propagation dans le circuit électronique, au temps de variation d'un signal (passage de « 0 » à « 1 » et vice-versa), etc.

Finalement ils existent trois bus (association de plusieurs canaux de communication binaires) pour communiquer avec la mémoire : le bus de donnée, le bus d'adresse et finalement le bus de commande, sans compter les entrées de VCC et de GND.

I. Le bus de donnée est : bidirectionnel (envoi de donnée dans deux sens) et parallèle (envoi d'un certain nombre de bits simultanément). Il sert à recevoir (lire) ou à envoyer (écrire) des données sur la mémoire. Dans le cas de notre mémoire le bus de donnée a une largeur de 16 bits
II. Le bus de d'adresse est : unidirectionnel (envoi de donnée dans un seul sens) et parallèle (envoi d'un certain nombre de bits simultanément). Il sert à choisir la l'adresse de la case mémoire que l'on souhaite sélectionner, sachant que chaque case mémoire a une adresse unique. La taille de la case mémoire est déterminée par la largeur du bus de donnée.
III. Le bus de commande est : unidirectionnel (sauf pour la mémoire flash) et parallèle. Il sert à commander la mémoire (choisir le mode de fonctionnement, donner un ordre de lecture/écriture, initialiser la mémoire, etc.) La largeur du bus de commande dépend du nombre de signaux nécessaire pour commander la mémoire. Dans un cas général plus le bus de commande est large plus la mémoire est compliquer à contrôler. Le bus de commande inclus le signal d'horloge (pour la mémoire SDRAM). Dans le cas de la mémoire SRAM le bus de commande est composé de : WE, CS, OE, LB, UB.

Voici un schéma fonctionnel simplifié de la composition interne d'une mémoire SRAM.



Objectif à atteindre

L'objectif à atteindre était de pouvoir lire/écrire sur une case mémoire, puis postérieurement lire/écrire sur plusieurs case mémoire à la suite. Tout ceci en respectant les timings de la mémoire. Pour vérifier le bon fonctionnement de la mémoire, les données à écrire sur la mémoire seraient lues sur les interrupteurs présents sur la carte et les données lues seraient affichées sur les LED aussi présentes sur la carte DE1.

Travail réalisé

Tout d'abord il m'a fallut lire la doc technique de la carte, pour comprendre son fonctionnement, puis en me basant sur les timings et les chronogrammes, j'ai écris un programme en VHDL qui en simulation reproduisait les chronogrammes de la mémoire autant à l'écriture qu'à la lecture.

Il a d'abord fallut comprendre à quoi servait chaque pin de notre mémoire.

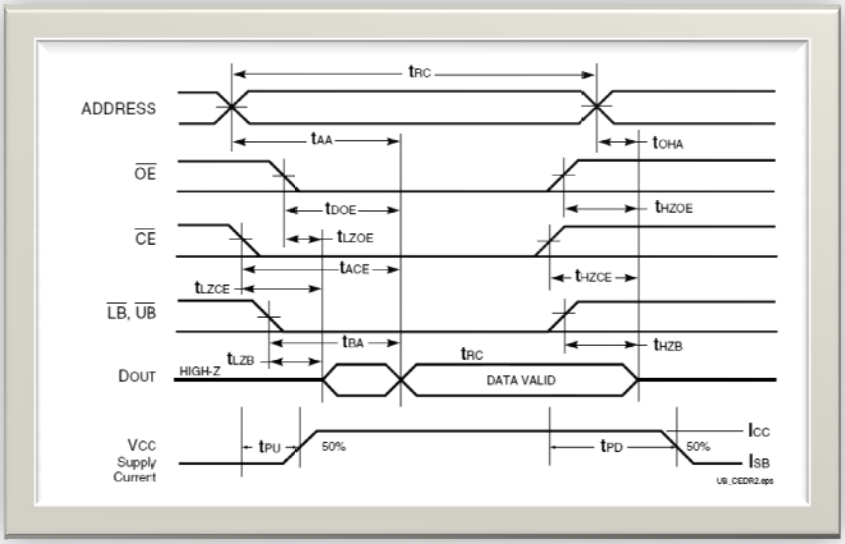
- A [0..17] : C'est le bus d'adresse.
- IO [0..15] : C'est la bus de donnée.
- CE : C'est le bit qui active/désactive la mémoire.
- OE : C'est le bit qui détermine si l'on veut lire la mémoire.
- WE : C'est le bit qui détermine si l'on veut écrire la mémoire.
- LB : C'est un bit de masquage des poids faible.
- UB : C'est un bit de masquage des poids fort.

A0-A17	Address Inputs
I/O0-I/O15	Data Inputs/Outputs
$\overline{CE}$	Chip Enable Input
$\overline{OE}$	Output Enable Input
$\overline{WE}$	Write Enable Input
$\overline{LB}$	Lower-byte Control (I/O0-I/O7)
$\overline{UB}$	Upper-byte Control (I/O8-I/O15)
NC	No Connection
Vcc	Power
GND	Ground

Tous ces bits sont actifs à « 0 ».

Les bits LB et UB n'ont à priori aucune importance, dans le fonctionnement classique de la mémoire. Ils servent juste à masquer une partie du bus de donnée.

Puis il a fallut comprendre grâce au chronogramme, comment il fallait agir sur ces différents bits pour commander la mémoire. Voici le chronogramme représentant un ordre de lecture, le chronogramme d'écriture étant exactement pareil, ne sera pas présenté.

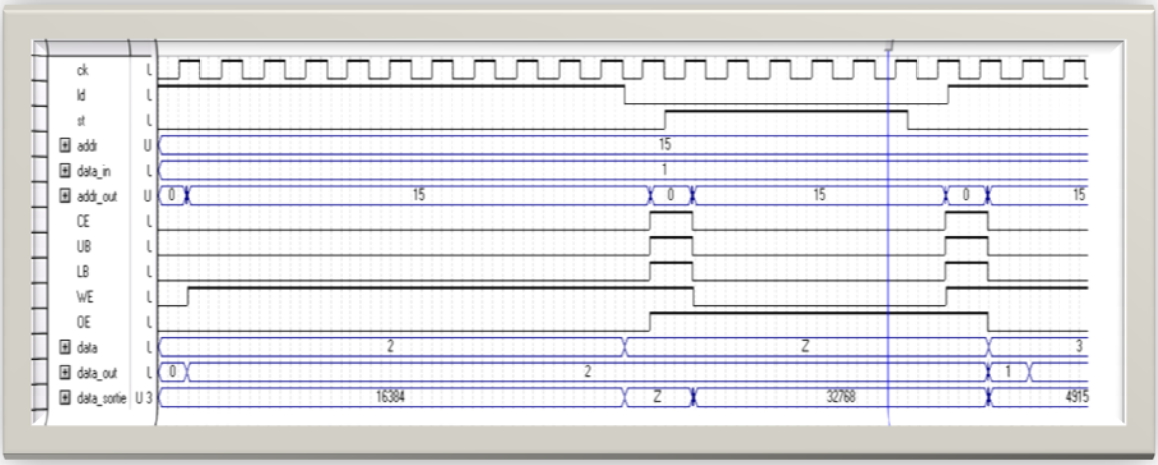


On peut voir sur ce chronogramme, qu'il existe des temps d'attentes entre chaque changement d'état. Ces temps d'attente sont appelés timings. Il faut absolument les respectés sinon la mémoire ne fonctionne pas correctement. Par exemple il faut attendre 15 ns entre la sélection de la case mémoire et l'apparition des données en sortie. Pour résoudre ce problème, j'ai tout simplement fait fonctionner le programme de contrôle de la mémoire à une fréquence de 24 MHz, ainsi une période d'horloge serait plus longue que le timing le plus long de la mémoire (15 ns).

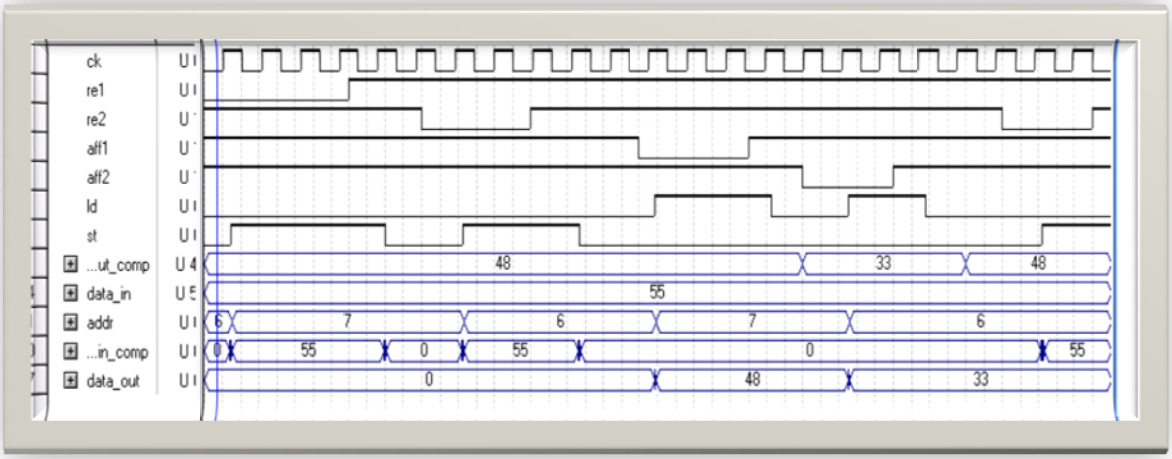
Le problème des timings maintenant résolu. Il fallait écrire un programme qui puisse lire/écrire sur la mémoire, tout en affichant les données sur les LED et les lires sur les interrupteurs. Pour cela j'ai divisé mon programme en deux parties, une partie qui gère directement la mémoire, « gestion-mémoire », en agissant sur les différents bus de la mémoire puis une deuxième partie qui contrôle la mémoire à partir du mode de fonctionnement choisi par l'utilisateur. Le programme de contrôle envoi des signaux de (ST, LD), pour donner l'ordre d'écrire ou de lire en mémoire tout en envoyant la bonne adresse et les bonnes données au programme de gestion.

- Gestion_mem : Ce programme est universel car il peut fonctionner sur n'importe quelle mémoire SRAM. Les bits ST et LD, commande le bloc, il faut juste envoyer un « 1 » sur le bit ST si l'on veut enregistrer les données présente sur les bus de donnée à l'entrée, à l'adresse présente sur le bus d'adresse. Idem pour LD, si l'on veut lire une case mémoire. Le programme en soit n'est pas très compliqué, on lit juste les entrées ST et LD, puis à partir de ces entrées, il met à « 0 » les signaux nécessaires (WE,CE dans le cas de ST et OE,CE dans le cas de LD) tout en gardant LD et UD à « 1 ». La grande difficulté de ce programme fut apprendre à géré le bus bidirectionnel. Il faut en effet déclarer le bus en tant que INOUT dans l'entité. Puis dans le programme il faut le brancher sur un signal et utiliser ce signal dans la description du programme. Il faut aussi avant de lire sur le bus de donnée, le mettre en état de haute

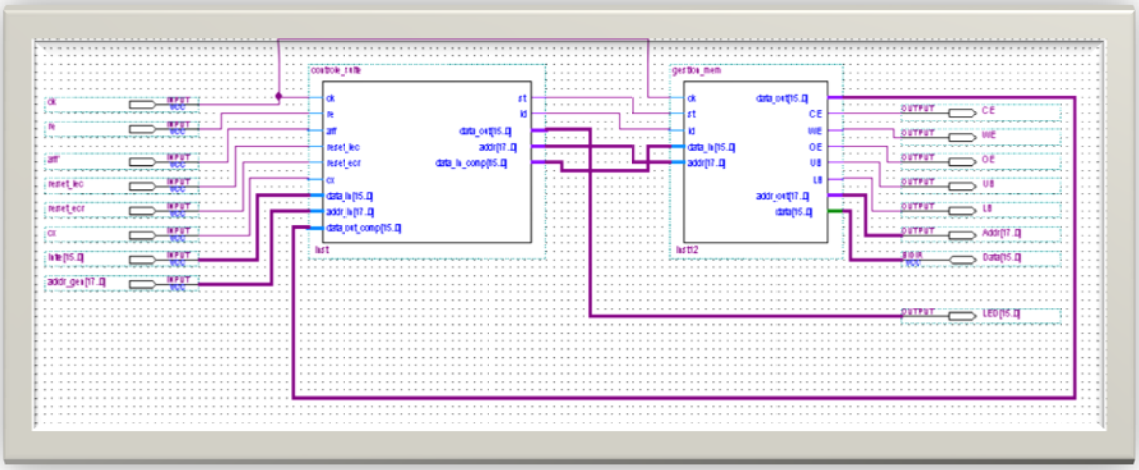
impédance : « data <= "ZZZZZZZZZZZZZZZZ"; ». J'ai eu un peu de mal à trouver qu'il manquait cette ligne de code dans mon programme. J'ai trouvé la solution en lisant un programme en VHDL déjà près sur le site d'Altéra. Voici une représentation de la simulation de ce programme.

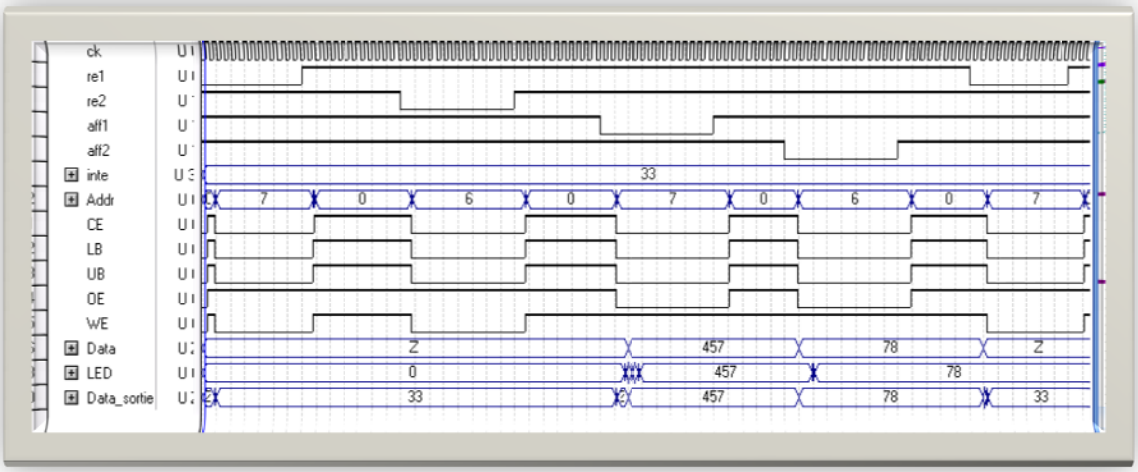


- Contrôle_mem : Ce programme gère les différents types de fonctionnement de la mémoire. Il a été développé en deux parties, une première partie où l'utilisateur ne peut écrire/lire que sur la première case mémoire. Cette première version fut un programme test, pour tester le bon fonctionnement du programme gestion_mem. Après avoir testé ce programme j'ai réalisé une version plus élaborée où à chaque demande d'enregistrement/lecture les données sont sauvegardées/lues dans la case mémoire suivante. Les compteurs de lecture et d'écriture étant indépendants, on peut écrire dans une case mémoire et lire une autre en même temps. Aucune difficulté particulière c'est présenté pendant la réalisation de ce programme. Voici la simulation de ce programme.



- Controlleur_mem : Ce programme n'est rien d'autre que l'union des deux programmes précedents, l'interconnexion de ceci et la bonne assignation des pins du composant. Voici le schéma bloc du programme final et sa simulation.





Conclusions

Finalement le programme a très bien fonctionné. Il était près à être intégrer dans le programme général du projet, mais il y avait un gros problème la taille de la mémoire SRAM ne permettait pas d'enregistrer une photo au format VGA. Donc il m'a fallut faire fonctionner une autre mémoire pour pouvoir réaliser mon projet. Je me suis décidé de poursuivre avec la mémoire FLASH, qui elle avait la taille suffisante pour enregistrer une image au format VGA. L'utilisation de cette mémoire allait ce révéler assez similaire à l'utilisation de la mémoire SRAM.

FLASH

Introduction

La mémoire FLASH était assez grande pour contenir une photo au format VGA et son fonctionnement semblait à priori assez simple.

Description technique

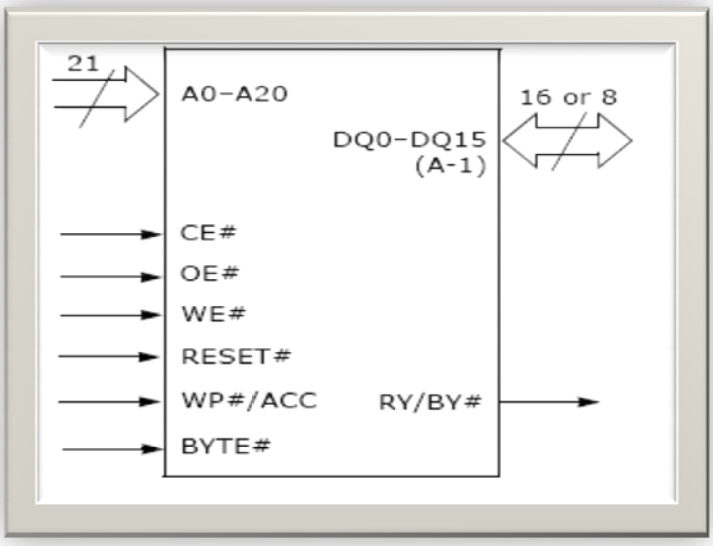
La mémoire FLASH est réalisée avec des portes NAND ou NOR, mais cette mémoire n'est pas volatile. La mémoire FLASH est moins chère à intégrer que la mémoire SRAM, d'où sa taille plus importante. Mais le faite que les données soit sauvegardé même si la mémoire est hors tension, rend plus long le processus d'écriture/lecture, car il faut modifier physiquement le circuit.

Le mémoire présente sur notre carte à une capacité de 4 Mo. Elle est asynchrone, et dispose des même trois bus de la mémoire SRAM, sans compter les VCC et les GND. Cette mémoire à aussi des timings spécifiques qui sont plus long que ceux de la mémoire SRAM, par conséquent celle-ci est beaucoup plus lente que la mémoire SRAM. La grande différence la FLASH et la SRAM se trouve surtout au niveau du bus de commande et des spécificités de fonctionnement.

I. Le bus de donnée est : bidirectionnel (envoi de donnée dans deux sens) et parallèle (envoi d'un certain nombre de bits simultanément). Dans le cas de notre mémoire le bus de donnée a une largeur de 16 bits mais il peut être transformé en bus de 8 bits grâce au bit BYTE du bus de contrôle.

II. Le bus d'adresse est : unidirectionnel (envoi de donnée dans un seul sens) et parallèle (envoi d'un certain nombre de bits simultanément). Notre mémoire a un bus d'adresse de 21 bits de longs mais si le mode 8 bits est choisit le 16ème bit de donnée se transforme en un 22ème bit d'adresse. Ainsi on réduit la taille des cases est réduite de moitié, mais on double leur nombre donc on ne perd pas d'espace.

III. Le bus de commande : il est a priori unidirectionnel mais sur notre mémoire on peut voire que l'on a un bit de sortie qui nous informe sur l'état de la mémoire. Les bits de commande de base sont présents mais on retrouve d'autres bits spécifiques à l'utilisation de la FLASH.



Objectif à atteindre

Les objectifs à atteindre sont exactement les mêmes que pour la mémoire SRAM.

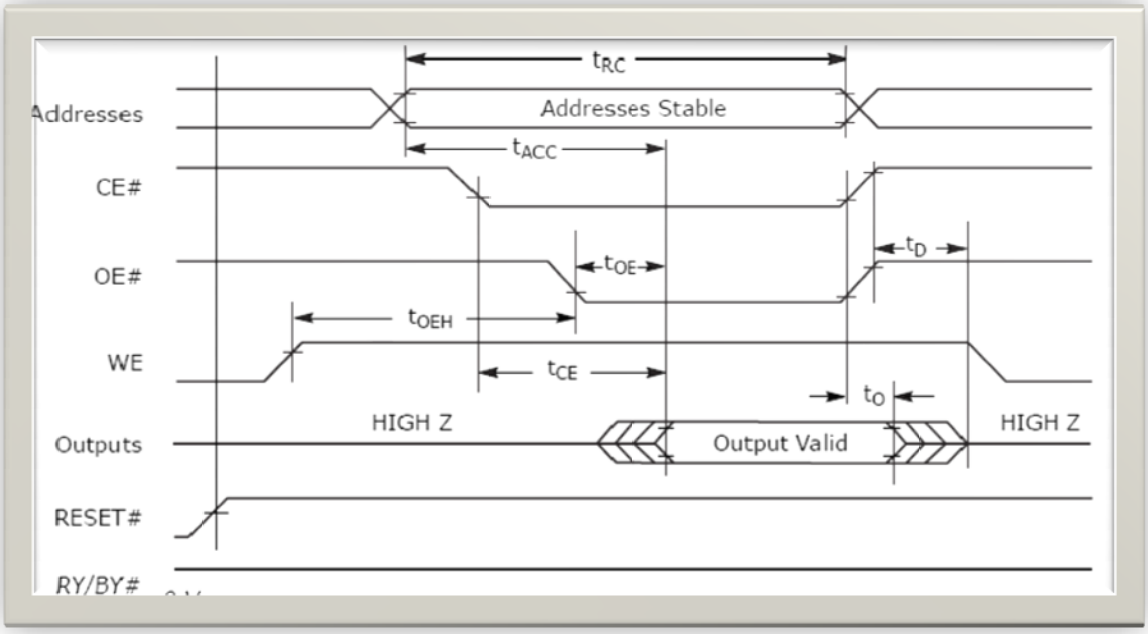
Travail réalisé

Tout d'abord il a fallut analyser le fonctionnement de la mémoire, pour cela il faut déjà savoir à quoi servaient les entrées additionnelles, RESET, WP/ACC, BYTE et la sortie RY/BY. Les autres entrées étant les même que pour la mémoire SRAM.

- RESET : comme son nom l'indique ce bit sert à interrompre l'utilisation de la mémoire sans avoir à la débrancher. Il sert aussi à remettre tout les paramètres de la mémoire à leur état initial.
- BYTE : ce bit sert à choisir entre le mode de fonctionnement 8 bits ou 16 bits.
- WP/ACC : Ce bit permet de choisir entre plusieurs modes de fonctionnement spécifiques à la mémoire FLASH.
- RY/BY : ce bit permet de savoir si la mémoire FLASH est disponible ou occupée.

Les modes de fonctionnement spécifiques sont assez compliqués, pour être honnête je n'ai pas très bien compris à quoi ils servaient. J'ai cru comprendre qu'il permettait l'exécution d'un programme présent dans la mémoire ou accéléré le fonctionnement de celle-ci. On pouvait également, protéger certain secteur pour éviter qu'il soit effacé lors d'un formatage. Il y avait aussi un mode formatage automatique de la mémoire. La mémoire est divisée en secteur, ainsi un programme qui veuille utiliser la mémoire, peut savoir combien d'espace disponible il reste dans celle-ci. La raison pour laquelle je ne n'ai pas étudié ces différents modes de fonctionnement est que les bits BYTE, WP/ACC et RY/BY ne sont pas branché sur le FPGA donc inaccessible pour l'utilisateur.

Finalement pour utiliser la mémoire FLASH il faut juste utiliser le même programme que précédemment mais il a fallut changer la taille du bus d'adresse. Voici un chronogramme de lecture de la mémoire FLASH qui prouve que celle-ci a le même comportement que la mémoire SRAM.



Il ne sert à rien de présenter les chronogrammes de la mémoire flash car ce sont exactement les mêmes que ce de la mémoire SRAM.

Conclusions

La mémoire FLASH présentait l'avantage d'être assez grande pour contenir la totalité d'une image au format VGA. Le faite quelle ne soit pas volatile nous permettait aussi de pouvoir conserver les photos, même si la carte aurait été mise hors tension. Mais en avançant un plus dans le projet je me suis aperçu que cette mémoire avait un temps d'accès (90 ns) beaucoup trop long pour afficher un pixel à l'écran. Car le mode d'affichage VGA à 50Hz, qui est un minimum, ne permet pas d'attendre les pixels autant de temps. Finalement je me suis tourné vers la dernière mémoire disponible sur la carte, la mémoire SDRAM.

SDRAM

Introduction

La capacité de la mémoire SDRAM (8 Mo), était assez grande pour contenir une photo au format VGA, et sa vitesse de fonctionnement semblait convenir parfaitement à l'application recherchée. Je me suis intéressé à cette mémoire en dernier car, c'est a priori, la mémoire qui a le fonctionnement le plus compliqué à mettre en œuvre.

Description technique

La mémoire SDRAM (Synchronous Dynamic Random Acces Memory) est une mémoire volatile dont le mode de fonctionnement diffère énormément des autres mémoires précédemment traitées. La mémoire SDRAM a été créée pour permettre de réduire les couts de fabrication des mémoires SRAM tout en ayant une capacité supérieure. Une cellule de mémoire SDRAM repose sur l'architecture, un transistor et un condensateur. Les bits sont sauvegardés dans un condensateur, si le condensateur est vide le bit est à « 1 » sinon le bit est à « 0 ». Les condensateurs sont disposés sous une forme de matrice ou l'on accède par colonne et non par bit (sinon la logique de commande interne serait trop compliquée). Puis les bits sélectionnés sont identifiés grâce à un multiplexeur branché sur chaque colonne. Donc la lecture de la mémoire SDRAM ne se fait pas exclusivement case mémoire par case mémoire, on peut aussi lire toutes une colonne à la fois, mais aussi plusieurs cases mémoires à la suite en accédant une seule fois dans à la mémoire. Mais le gros problème de la mémoire SDRAM est que les condensateurs qui la composent ont des fuites inévitables, donc au bout de 64 ms (standard JEDEC), la valeur du bit contenu dans ce condensateur peut changer. Ce qui met en danger l'intégrité des données sauvegardées en mémoires. La mémoire SDRAM à plusieurs modes de fonctionnement différents, d'où la nécessité d'initialiser la mémoire après sa mise sous tension.

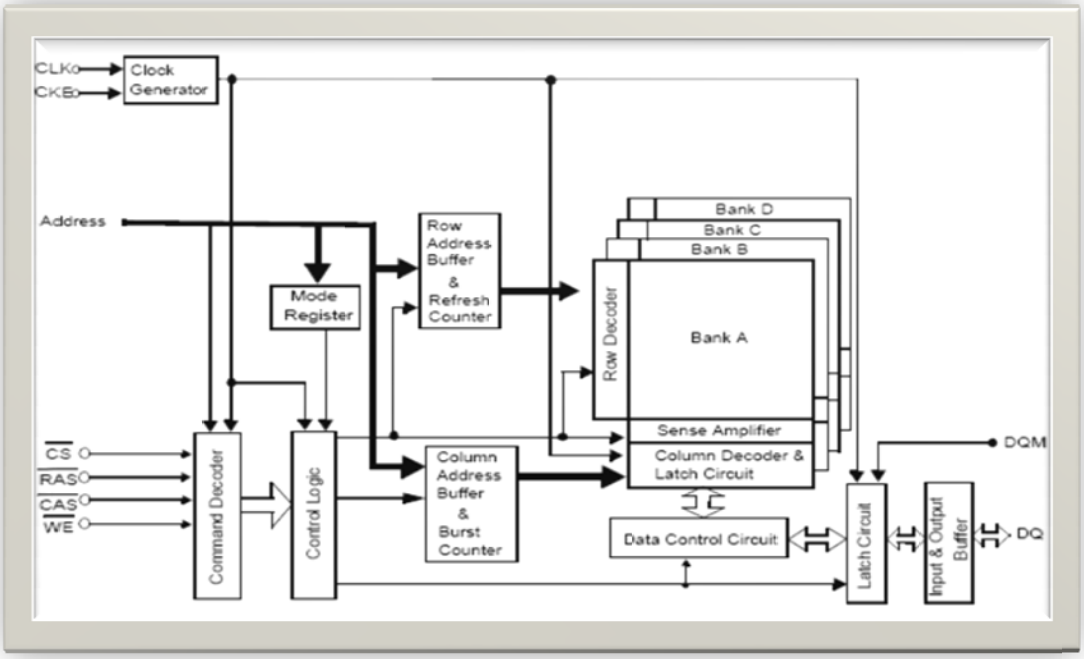
Le circuit de SDRAM disponible sur notre carte Altéra DE1 a une capacité de 8 Mo. Cette mémoire est synchrone cela veut dire que les données seront synchronisées avec l'horloge principal du système. Cette mémoire a aussi trois bus de communication différent.

I. Le bus de données : c'est exactement le même que pour la SRAM.
II. Le bus d'adresse : c'est aussi exactement le même que pour la mémoire SRAM, sauf que celui-ci a largeur de 12 bits. Le bit 10 du bus est spécial car il sert aussi comme bit de commande pour choisir le mode de fonctionnement de la mémoire. La mémoire SDRAM est

aussi divisée en 4 blocs de 2 Mo chacun, ainsi nous pouvons accéder à 4 cases mémoire simultanément, le seul inconvénient est que ces 4 blocs partage le même bus de données, il faut réaliser une logique de commande vraiment complexe pour gérer les 4 blocs simultanément. Pour accéder à ces 4 blocs nous disposons de 2 bits supplémentaires d'adresse.

III. Le bus de commande : il est composé de plusieurs signaux de commande qui permette de choisir entre toutes les fonctions de la mémoire SDRAM. La mémoire étant synchrone le bus de commande contient un signal d'horloge.

Voici un schéma représentant la structure fonctionnelle de la mémoire SDRAM.



Voici une description des pins de circuit de la mémoire SDRAM.

PIN DESCRIPTIONS

CLK	Master Clock	DQM	DQ Mask Enable
CKE	Clock Enable	A0-11	Address Input
$\overline{CS}$	Chip Select	BA0,1	Bank Address
$\overline{RAS}$	Row Address Strobe	Vdd	Power Supply
$\overline{CAS}$	Column Address Strobe	Vddq	Power Supply for DQ
$\overline{WE}$	Write Enable	Vss	Ground
DQ0 ~ DQ15	Data I/O	Vssq	Ground for DQ

Objectif à atteindre

Les objectifs à atteindre sont toujours les mêmes.

Travail réalisé

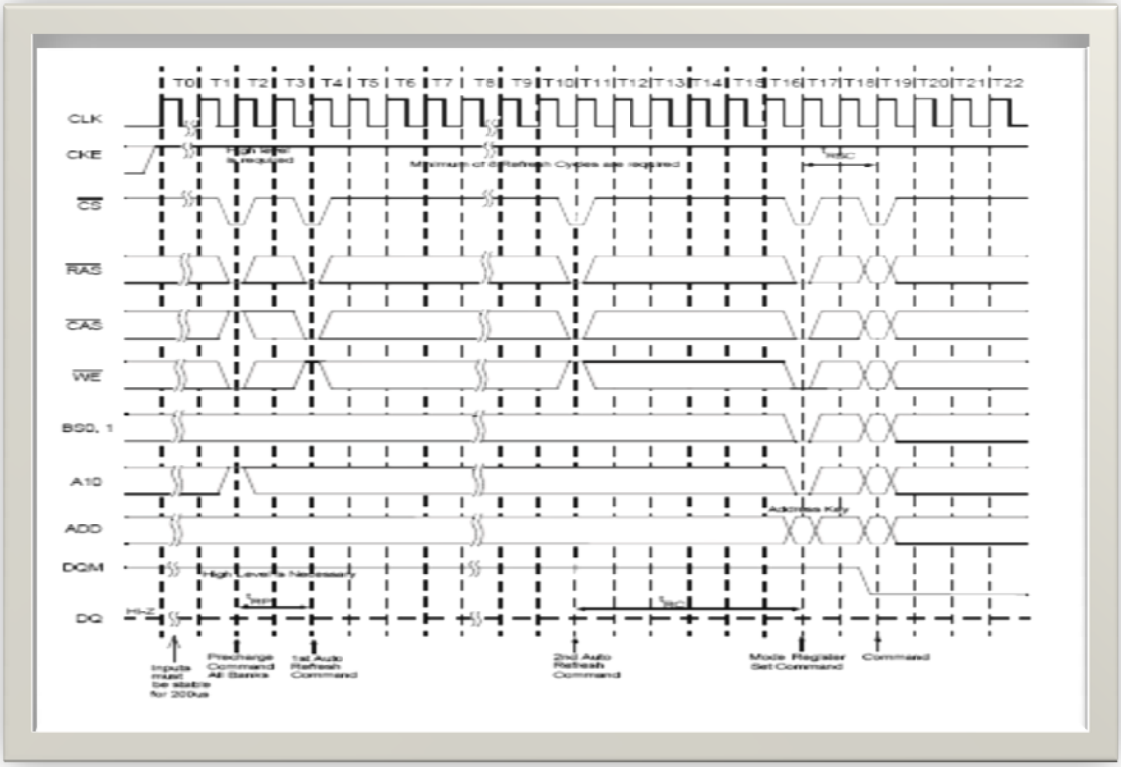
Tout d'abord comme pour les deux mémoires précédentes il fallait que je comprenne le fonctionnement de cette mémoire et pour cela il fallait que je comprenne le l'utilité de chaque bit de bus de commande. Mais je me suis vite rendu compte que individuellement chaque bit n'avait aucune importance mais ce sont les différentes combinaisons que l'on réalise avec ceci qui ont un impacte sur le système. Sauf peut être DQM qui ne sert qu'au masquage des bits de poids fort de la sortie et CKE qui sert lui au masquage de l'horloge. Puis il à fallut que je comprenne la signification du diagramme d'état expliquant le fonctionnement de la mémoire. Puis il a fallu que je m'organise pour rafraichir une colonne de la mémoire toutes les 7.8 µs. Finalement j'ai du créer un programme d'initialisation.

Initialisation de la mémoire

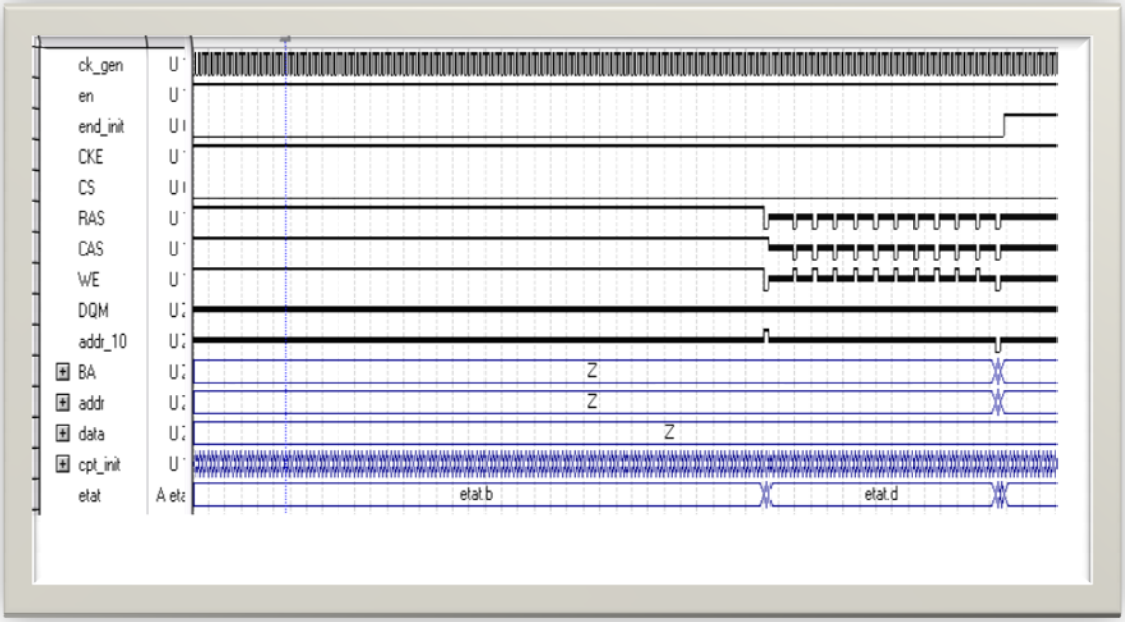
L'initialisation se fait en plusieurs étapes :

I. Il faut alimenter la mémoire, garder un état de NOP (No OPération) sur les entrées, garder les signaux DQM et CKE à 1.
II. Il faut rester dans un état de NOP pendant 200 µs.
III. Il faut lancer une commande de « precharge all banks ».
IV. Attendre que Trp ce soit écoulé puis lancer un minimum de 8 commande de CBR (auto-rafraîchissement de la mémoire.
V. Finalement il faut procéder à l'initialisation du registre de contrôle.

Voici un chronogramme qui montre les différentes étapes d'initialisation.

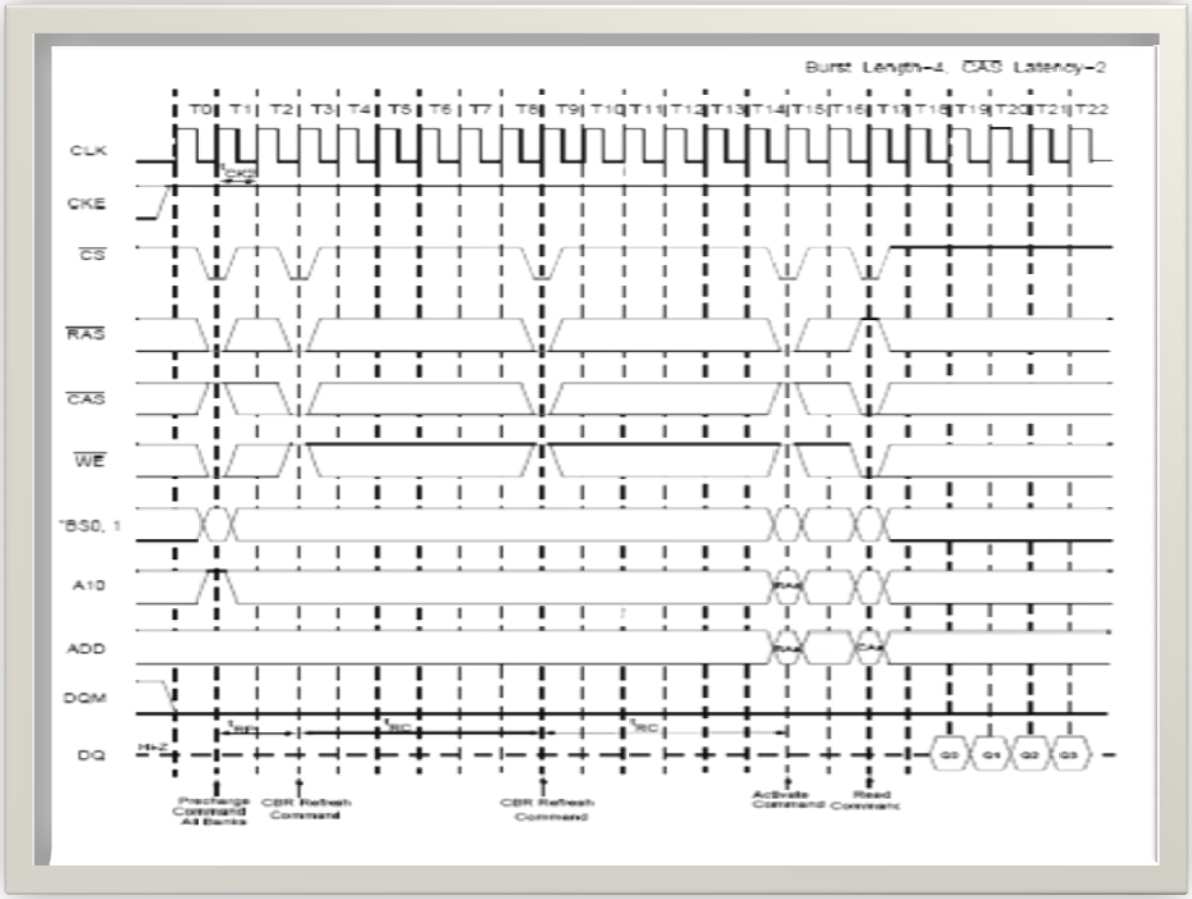


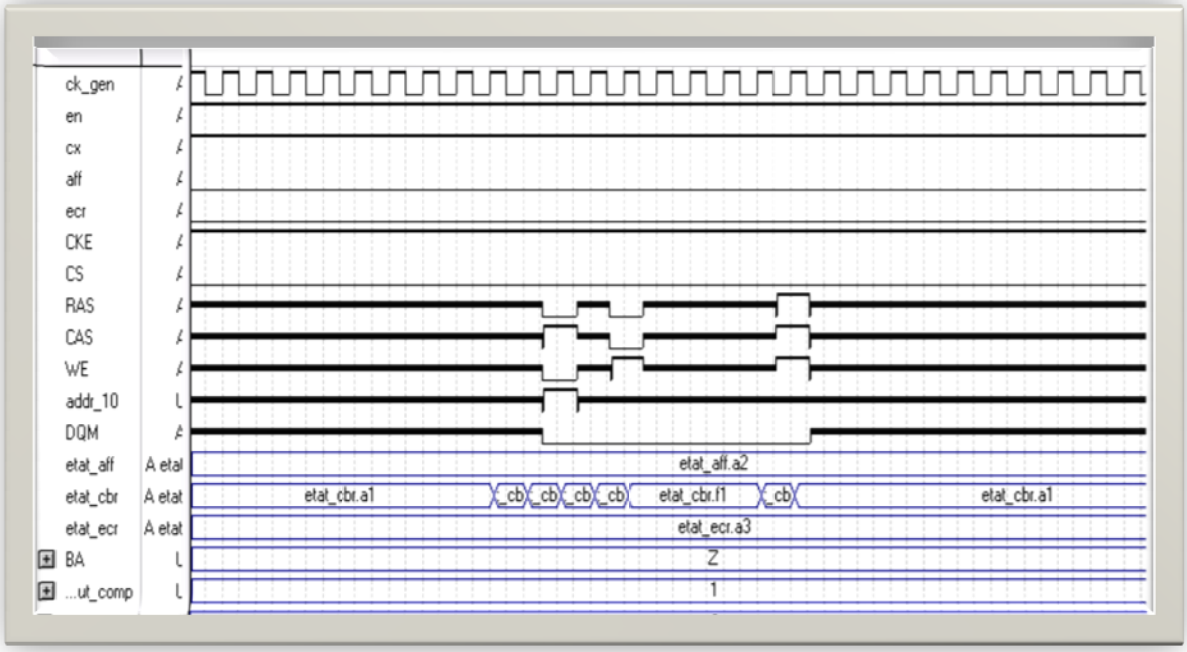
Le programme qui gère l'initialisation est basé sur un compteur qui fait varier les états d'une machine à état en fonction de l'état du compteur. Ce compteur tourne a une fréquence de 50 MHz donc le compteur c'incrémente toute les 20 ns, ceci m'a permis de respecter les timings de la mémoire. En restant dans un état le nombre de cycle d'horloge j'arrive à prévoir le temps pendant lequel je resterais dans cet état. Mon programme reproduit le chronogramme donné dans la documentation de la mémoire. Je laisse même quelque petite marge de temps supplémentaire pour être bien sur que la mémoire fonctionne bien. Voici la simulation de mon programme d'initialisation.



Les auto-refreshs (CBR)

La mémoire SDRAM à besoin que la donnée soit réécrite sur chaque condensateur toutes les 64 ms. Il y a deux façon de procéder soit on rafraichit toutes les colonnes chaque 64 ms soit on rafraichit une colonne toute les 7.8 µs. Il y a 8042 colonne, donc le temps de écoulé entre deux rafraichissement d'une même colonne sera 64 ms. J'ai retenu la deuxième méthode car si l'on applique la première on risque de perdre trop d'information pendant le temps de rafraichissement de toutes les colonnes successivement. Donc dans le programme il fallait que je rajoute une fonction qui interrompt l'action en cours pour aller rafraichir une colonne. Le problème est qu'après une interruption de CBR, la mémoire part en état d'attente et ne revient pas terminer l'action en cour. Donc il fallait aussi que j'enregistre l'état de l'action en cour pour pouvoir le reprendre après le CBR. Il y a aussi un temps d'attente (TRC = 70 ns) à respecter après une demande de CBR avant de pouvoir exécuter n'importe quelle action. Voici un chronogramme qui montre l'exécution d'une demande de CBR et image qui montre une simulation de CBR réalisé avec mon programme.


Burst Length=4, CAS Latency=2
CLK
CKE
CS
RAS
CAS
WE
BS0, 1
A10
ADD
DQM
DQ
Precharge Command All Banks
CBR Refresh Command
CBR Refresh Command
Activate Command
Read Command


ck_gen
en
cx
aff
ecr
CKE
CS
RAS
CAS
WE
addr_10
DQM
etat_aff
etat_cbr
etat_ecr
BA
...ut_comp
etat_aff.a2
etat_cbr.a1
_cb
_cb
_cb
_cb
etat_cbr.f1
_cb
etat_cbr.a1
etat_ecr.a3
Z
1

Programmation du registre de mode

Avant de pouvoir utiliser la mémoire il à fallut, programmer le registre de mode qui permet de choisir la longueur des bursts en sortie (lecture de plusieurs cases à la suite en ne donnant qu'un seul ordre de lecture). On peut choisir entre 5 longueurs de burst différentes (1, 2, 4, 8 et une page entière). On peut choisit aussi la latence du CAS (2 ou 3 cycles d'horloge), la latence du CAS dépend de la fréquence d'utilisation de la mémoire, si la fréquence d'horloge est élevée il est préférable d'appliquer une latence CAS de 3 cycles d'horloge. Pour mon programme j'ai choisi la lecture case par case et une latence de CAS de 2 cycles d'horloge car la fréquence d'horloge maximal disponible sur la carte DE1 est cadencé à 50MHz.

Diagramme d'état d'utilisation de la mémoire

La mémoire SDRAM étudié possède plusieurs états principaux de fonctionnement.

I. Un état « idle » ou de repos ou la mémoire n'exécute aucune action. Nous pouvons rester dans cet état autant de temps que nous le voulons. Il est juste interrompu par une demande éventuelle de CBR.
II. L'état de auto-refresh qui doit être accédé toutes les 7,8 µs.
III. Un état d'écriture/lecture : les données sont écrites/lues sur une colonne qui a été initialisait auparavant. Les données sont écrites/lues sous forme de burst. La colonne reste « ouverte » après une action d'écriture/lecture. Il faut préciser qu'une colonne ne peut restée ouverte que pendant une durée maximale de 100µs.
IV. Un état bis d'écriture/lecture : les données sont écrites/lues sur une colonne qui a été initialisait auparavant. Les données sont écrites/lues sous forme de burst. La colonne se « referme » (on exécute un precharge de la colonne) après une action d'écriture/lecture. Puis après ce precharge on revient à l'état de repos.

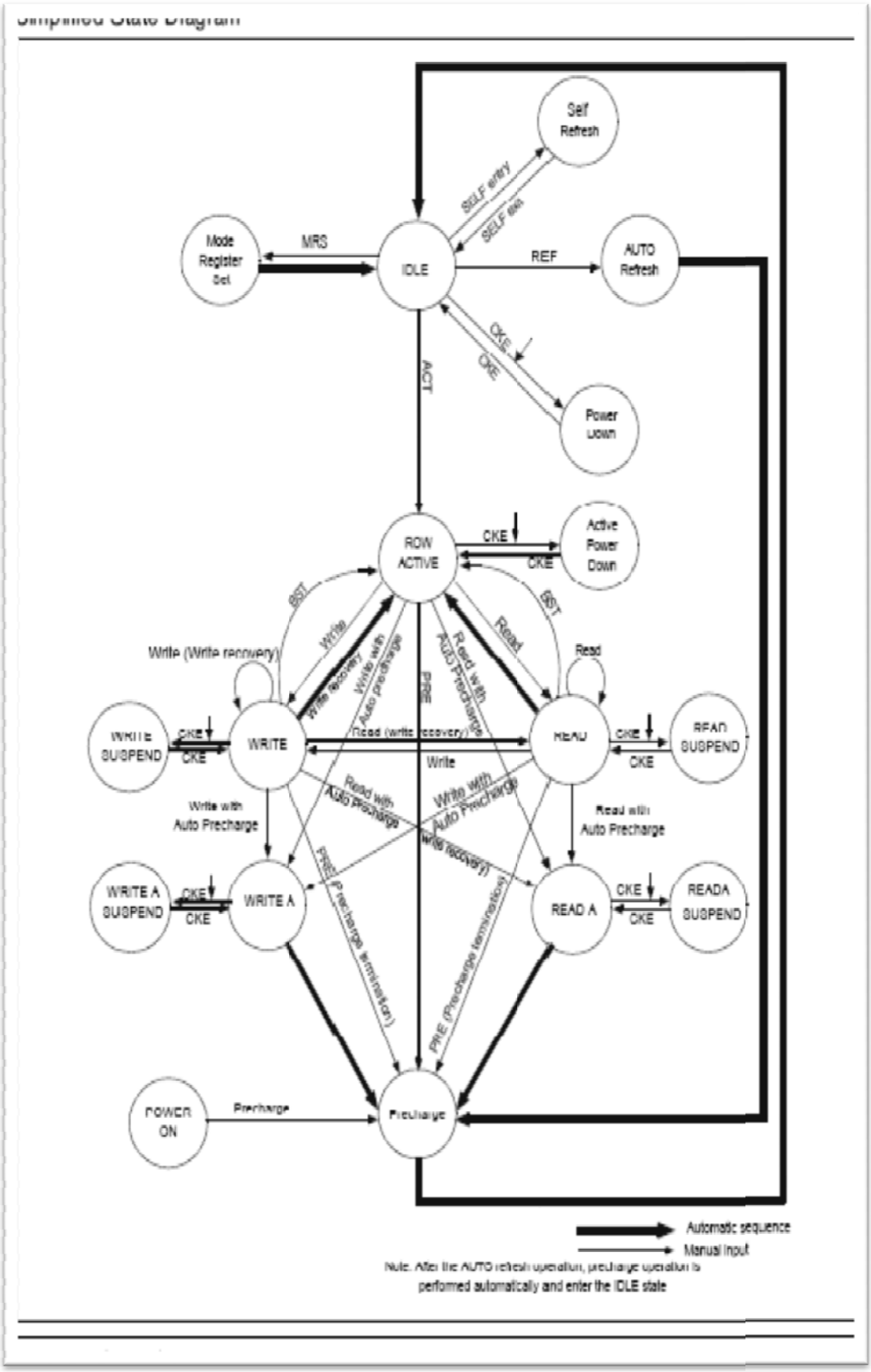
Sur al page suivant nous avons une représentation simplifiée du diagramme d'état de la mémoire.

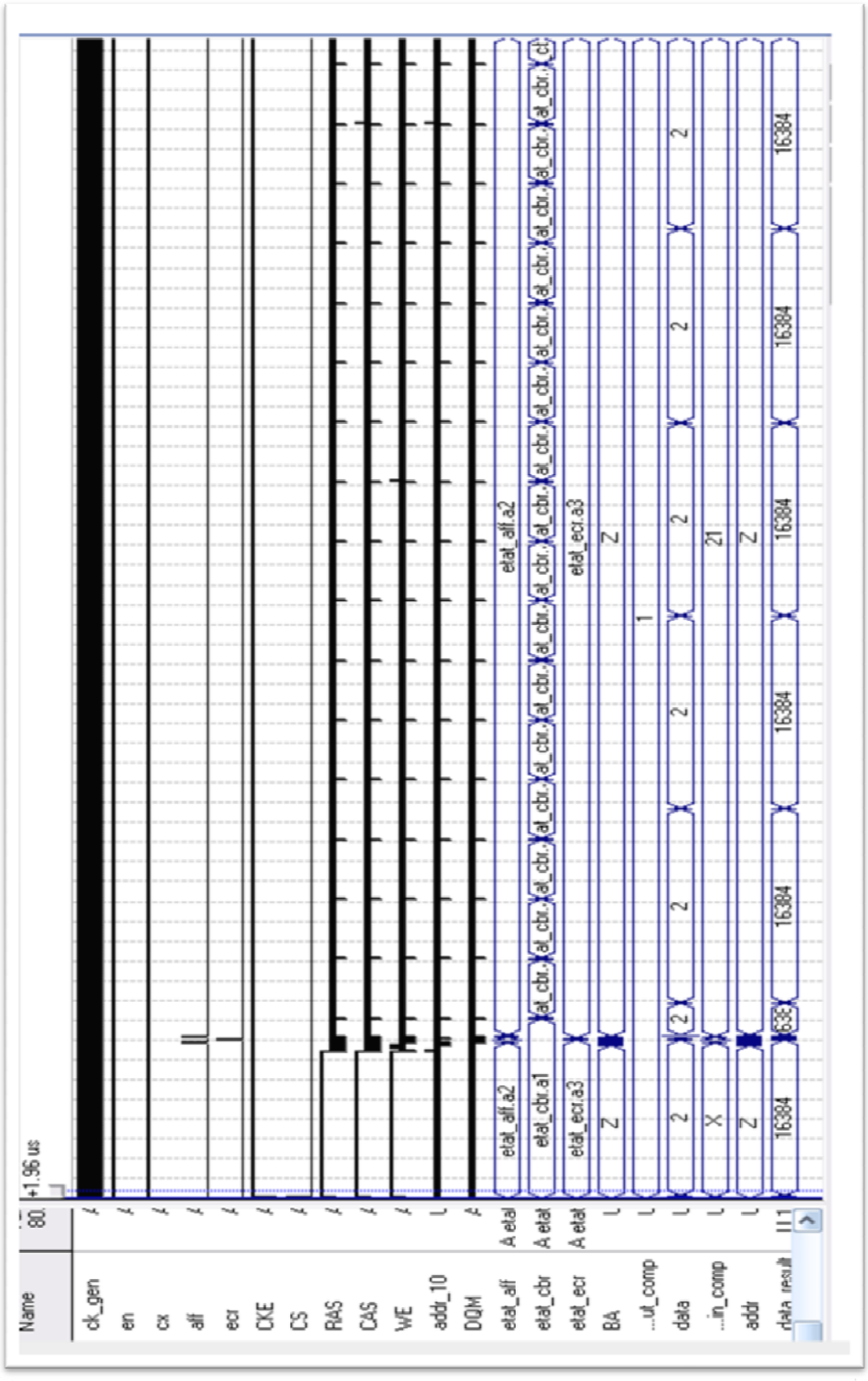
Dans mon programme de gestion de le mémoire SDRAM je me suis seulement intéressé aux états de lecture/écriture bis, car pour tester ma mémoire il fallait que j'écrive dans la mémoire case par case, donc la perte de temps engendré par la fermeture à chaque action ne serait pas contraignante. La lecture/écriture au contraire de la mémoire SRAM se fait selon une suite d'action déterminée et non juste en activant quelque signaux de commande, d'où la nécessité d'utiliser des machines à états pour les commandes de lecture/écriture.

Donc finalement dans mon programme principal j'ai du mettre en œuvre 4 machines à états différentes pour gérer tous les modes de fonctionnement de la mémoire. Le programme résultant est très long et compliqué, mais d'après la simulation tous les modes de fonctionnements et temporisations sont respecté. La simulation de celui-ci se trouve deux pages plus bas.

Conclusions

La mémoire SDRAM, remplissait entièrement le cahier des charges (taille et vitesse d'utilisation suffisante) mais à cause de la grande difficulté de mise en œuvre de cette mémoire, le programme de gestion de celle-ci n'a pas pu être finalisé entièrement.


Simplified State Diagram
Self Refresh
SELF entry
SELF exit
Mode Register Set
MRS
IDLE
REF
AUTO Refresh
CKE
CKE
ACT
Power Down
ROW ACTIVE
CKE
CKE
Active Power Down
BST
Write
Write recovery
Write with Auto precharge
PRE
Read with Auto Precharge
Read
BST
Read
Write (Write recovery)
WRITE SUSPEND
CKE
CKE
WRITE
Read (write recovery)
Write
READ
CKE
CKE
READ SUSPEND
Write with Auto Precharge
Read with Auto precharge
Write with Auto Precharge
Write recovery
Read with Auto Precharge
WRITE A SUSPEND
CKE
CKE
WRITE A
PRE (Precharge termination)
READ A
CKE
CKE
READA SUSPEND
PRE (Precharge termination)
POWER ON
Precharge
Precharge
Automatic sequence
Manual input
Note: After the AUTO refresh operation, precharge operation is performed automatically and enter the IDLE state


Name
ck_gen
en
cx
aff
ecr
CKE
CS
RAS
CAS
WE
addr_10
DQM
etat_aff
etat_cbr
etat_ecr
BA
...ut_comp
data
...in_comp
addr
data_result
etat_aff.a2
etat_cbr.a1
etat_ecr.a3
16384

Conclusions

Après avoir faire fonctionner les trois mémoires individuellement, même si j'admets j'aurais du commencer par la mémoire SDRAM. Mais la programmation des autres mémoires ma permis de prendre une certaine expérience avant d'attaquer la programmation de la mémoire SDRAM. Je peux dire maintenant que j'ai appris énormément dans le domaine des mémoires intégrées.

VGA

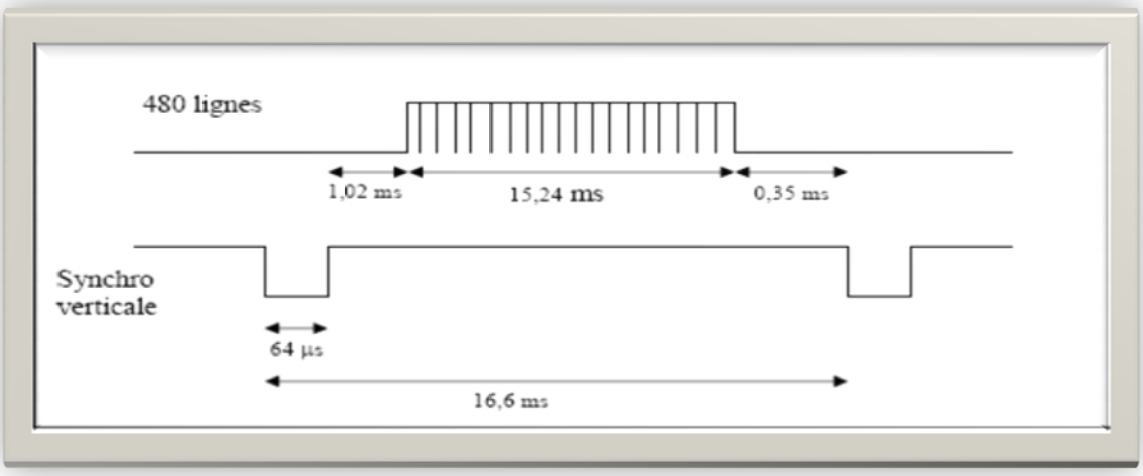
Introduction

Il existe aujourd'hui plusieurs standards de communication vidéo (VGA, DVI, HDMI, etc.) Dans notre cas la carte Altera intègre un port d'affichage VGA, donc si l'on voulait afficher la photo à l'écran il fallait crée un programme qui puissent envoyer des données en format RGB-VGA. Les données à afficher sont envoyer vers convertisseur analogique-numérique, qui converti ces données en signaux analogique, afin de les envoyer à travers le câble VGA vers un écran de PC standard.

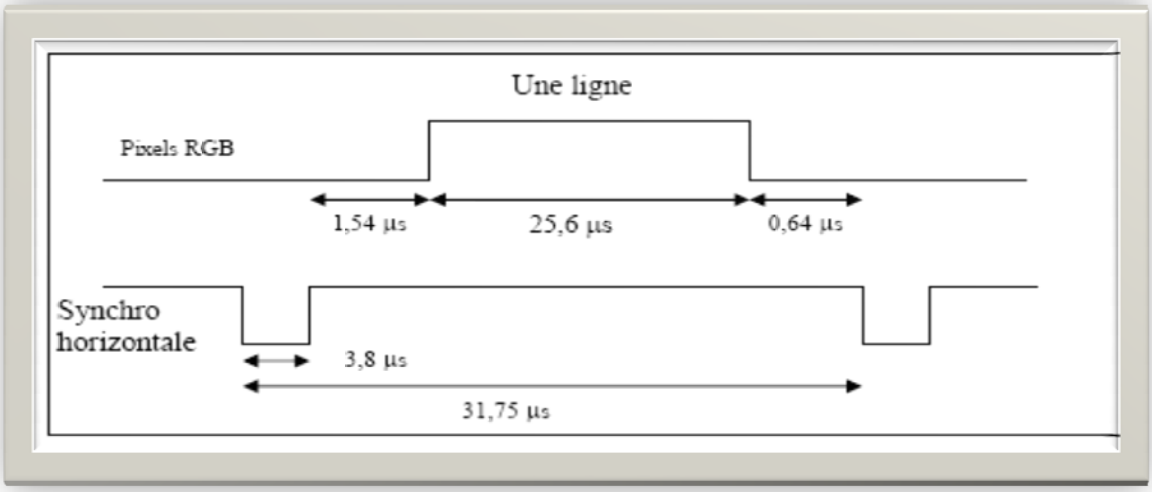
Description technique

Le format de communication VGA-RGB est déjà expliqué dans le paragraphe explicatif du début du rapport. Par contre il n'est pas mentionné que pour communiquer avec un écran il faut d'abord utiliser des signaux de synchronisation et envoyer les pixels à l'écran selon une temporisation bien spécifique. Cette temporisation dépend de la fréquence d'affichage à l'écran, dans le cadre de notre projet j'ai choisit une fréquence d'affichage de 60 Hz. Si l'on veut évaluer approximativement la fréquence pixel, il suffit de diviser 1/60 par le produit 680X480. Cela donne 50 ns environ entre 2 pixels. Voici pourquoi la mémoire flash était trop lente pour l'affichage à l'écran.

La trame VGA étant découpée en 525 lignes une fréquence image de 60 Hertz entraîne une fréquence ligne de 31,5 kHz. Une ligne correspond pour une fréquence pixel à 25 MHz à un comptage de 794 points. Le balayage vertical se décompose en une synchro (2 lignes) suivi de 32 lignes noires suivi des 480 lignes de pixel suivies de 11 lignes noires. Voici une représentation des timings de la synchro verticale à 60 Hz.



Le balayage horizontal se décompose en un signal de synchronisation (95 points) un pallier avant noir (43 points), les 640 pixels et un pallier arrière (16 points). Voici une représentation des timings de la synchro horizontale (31,5 kHz).



Objectif à atteindre

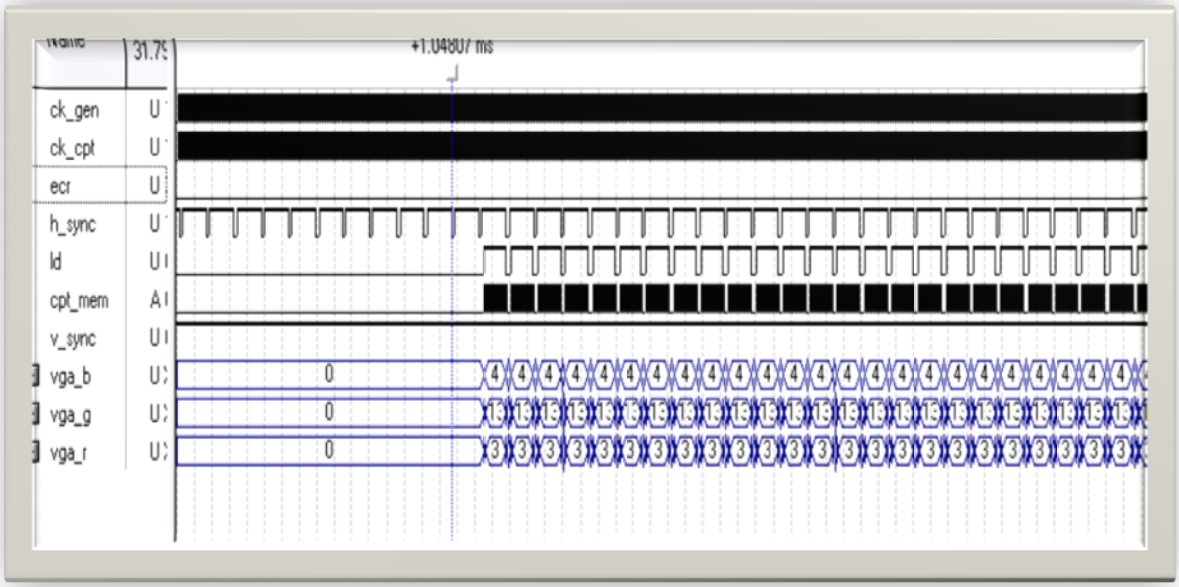
L'objectif principal était de pouvoir afficher une image contenu en mémoire sur l'écran VGA. Cette étape du projet a été divisée en deux parties. Une première partie ou l'on affichait une couleur unie (choisie avec les interrupteurs) à l'écran. Puis en deuxième afficher le contenu de la mémoire à l'écran.

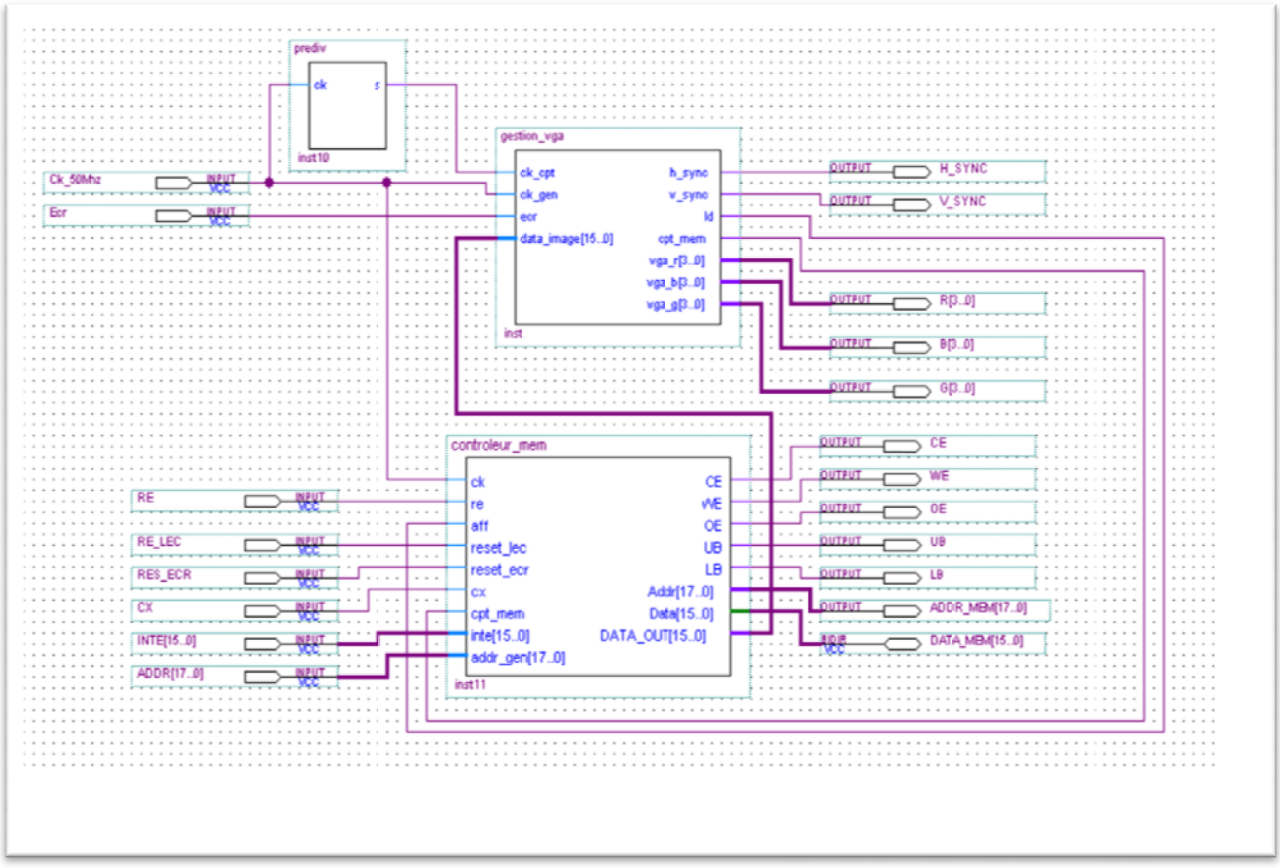
Travail réalisé

Il a fallu créer deux machines à état pour gérer l'affichage, une d'entre elle pour l'affichage de chaque colonne puis l'autre pour l'affiche des lignes. Ces machines sont synchronisées grâce à un compteur général qui permet de gérer la durée des signaux de synchronisation. En premier temps ce programme affichait à l'écran le même pixel définit par l'utilisateur. Puis en deuxième temps le programme générait un signal de demande de lecture à chaque affichage de pixel. Il fallait à ce point, brancher les deux blocs controlleur_mem et gestion_vga, en vérifiant que les timings des deux blocs était compatible. Voici la simulation du programme de gestion_vga et le schéma blocs du programme d'affichage final.

Conclusions

Pour conclure, le programme d'affichage associé au programme de gestion mémoire fonctionne très bien. Par contre même si une image est enregistrée en mémoire, on ne visualisera pas cette image à l'écran car une conversion RAW to RGB est nécessaire pour afficher l'image. Je n'ai pas eu le temps de traiter cette conversion en détail. Je pas non plus eu le temps d'étudier le fonctionnement de la caméra en détail. Les seules caractéristiques que j'ai mémorisées, étaient que l'initialisation de la caméra se fait en I2C, les pixels ont codés sur 10 et que le format de l'image est le RAW DATA. Voila donc je n'ai réussit à finaliser que le contrôle mémoire et le contrôle VGA. Les programmes en VHDL ne seront pas inclus en annexe du à la longueur de ceci.


ck_gen
ck_cpt
ecr
h_sync
ld
cpt_mem
v_sync
vga_b
vga_g
vga_r

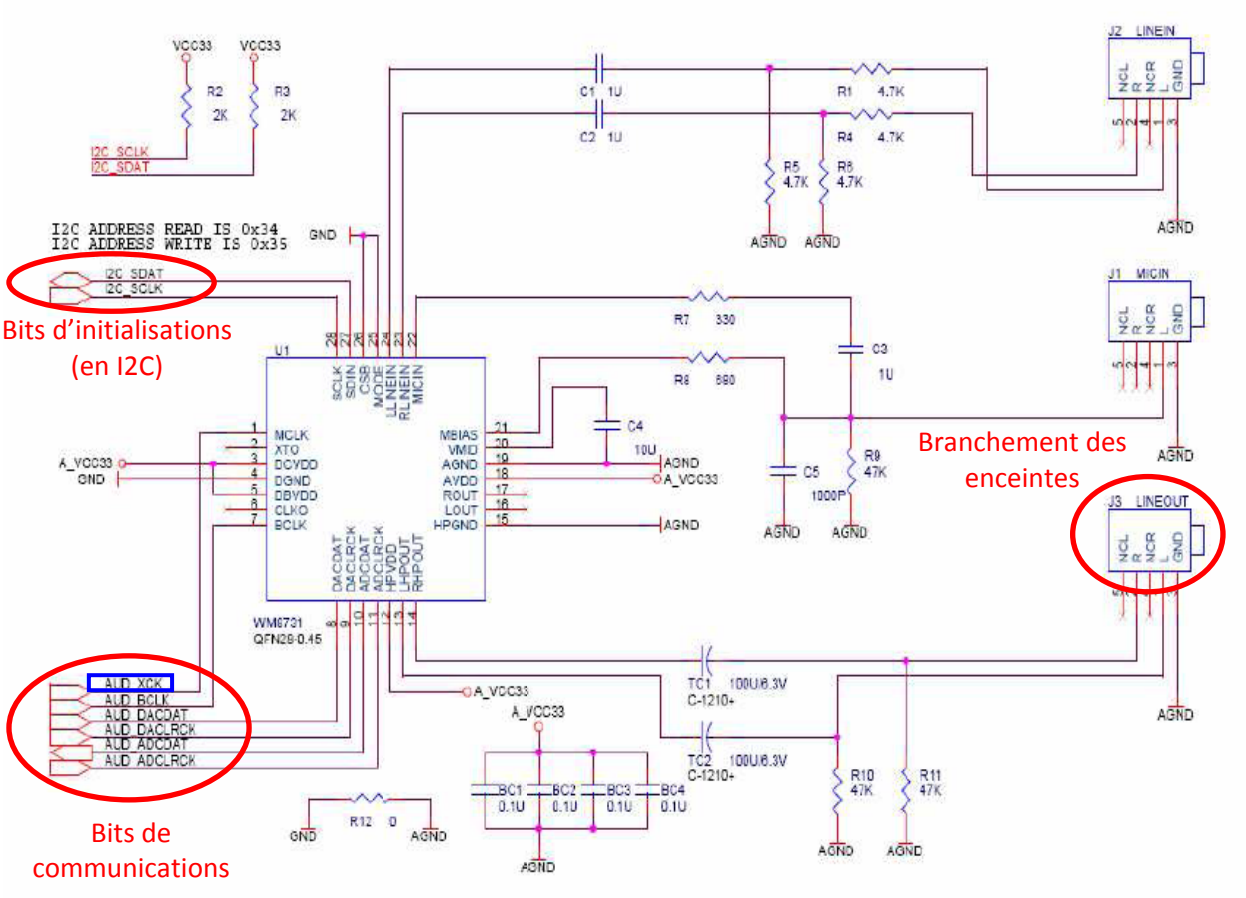

prediv
gestion_vga
controleur_mem
Ck_50Mhz
Ecr
RE
RE_LEC
RES_ECR
CX
INTE[15..0]
ADDR[17..0]
H_SYNC
V_SYNC
R[3..0]
B[3..0]
G[3..0]
CE
WE
OE
UB
LB
ADDR_MEM[17..0]
DATA_MEM[15..0]

Audio :

Etude

L'émission du bip se fera grâce au Codec WM8731 et sa sortie Line_out où 2 enceintes seront branchées.

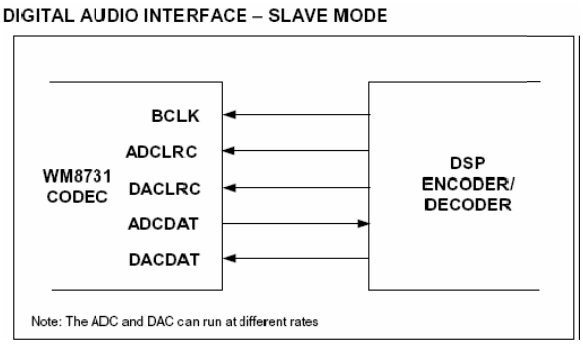
En étudiant le **schéma de câblage**, on repère les parties importantes :



Note :
D'après la datasheet, AUD_XCK est l'horloge principale du Codec, afin de le faire fonctionner nous devons envoyer un signal à 24Mhz.

Avant de commencer à programmer l'envoi des informations, il est important de déterminer l'interface entre le Codec et le FPGA : plusieurs modes de communication sont possibles.

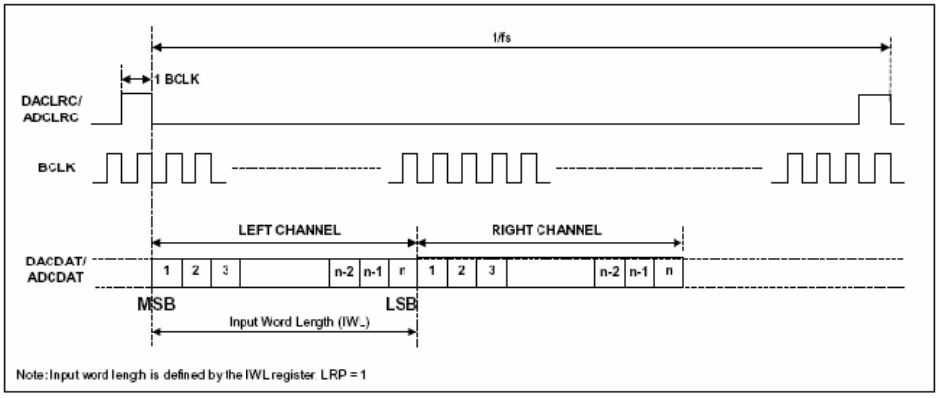
Nous avons choisit arbitrairement le **mode slave** du Codec



Les signaux de synchro BCLK et ADCLRC seront générés par le FPGA

L'envoi des données se fera en série par la broche DACDAT.

et le **mode DSP** de communication.



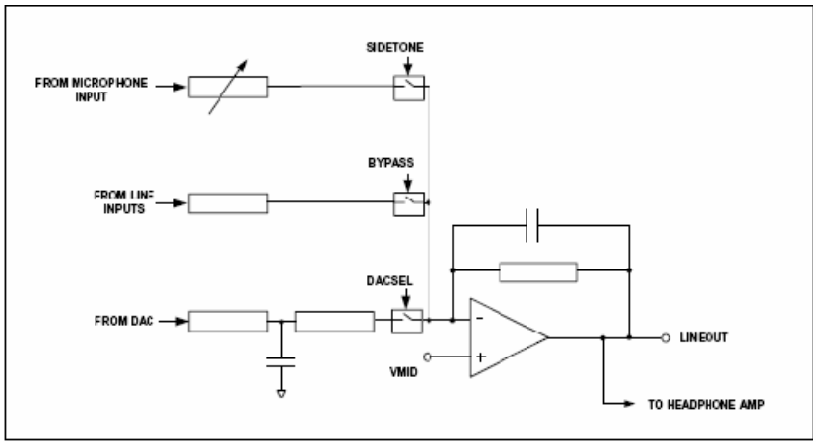
Note :

Le passage a zéro de DACLRC représente le début de l'échantillon à envoyer, il aura donc une fréquence de 48Khz afin d'avoir une qualité CD.

BCLK cadence l'envoi des bits du mot de l'échantillon. La taille de ce mot peut être réglée : 16, 20, 24 ou 32 bits (ici nous avons choisit 16 bits). 2 mots de 16 bits doivent être envoyé durant une période de DACLRC : sa fréquence sera donc FBCLK > 32+48Khz= 1,536Mhz

Nous avons choisit FBCLK = 2.4Mhz

De plus, trois sources peuvent être envoyées sur la sortie line out : depuis le microphone, depuis la Line_in ou depuis le convertisseur numérique/ analogique :



Par défaut, la Line_in est envoyée sur la Line_out.

Afin de sélectionner les modes de transmission et la source de Line_out, nous devons configurer 2 registres internes au Codec.

Registre R7 (0Eh) :

Adresse du registre	bit	nom	Valeur par défaut	description
0000111	1..0	Format[1..0]	10	Mode de communication 11 : mode DSP 10 : mode I2S 01 : MSB-first, left justified 00 : MSB first, right justified
	3..2	IWL[1..0]	10	Longueur du mot 11 : 32 bits 10 : 24 bits 01 : 20 bits 00 : 16 bits
	4	LRP	0	1 : MSB sur 2nd front montant de BCLK après front de DACLRC 0 : MSB sur1er front montant de BCLK après front de DACLRC
	5	LRSWAP	0	1 : voie droite en premier 0 : voie gauche en premier
	6	MS	0	1 : mode master 0 : mode esclave
	7	BCLKINV	0	1 : inverse BCLK 0 : n'inverse pas BCLK

Registre R4 (08h) :

Adresse du registre	bit	nom	Valeur par défaut	description
0000100	3	Bypass	1	1 : valide Bypass 0 : ne valide pas
	4	Dacsel	0	1 : valide Dacsel 0 : ne valide pas
	5	Sidetone	0	1 : valide Sidetone 0 : ne valide pas

Initialisation en I2C :

Principe

L'initialisation du codec se fait par un protocole standardisé, l'I2C. Ce protocole est utilisé pour qu'un micro-processeur puisse communiquer avec plusieurs composants différents avec seulement 2 fils (SDA et SCL).

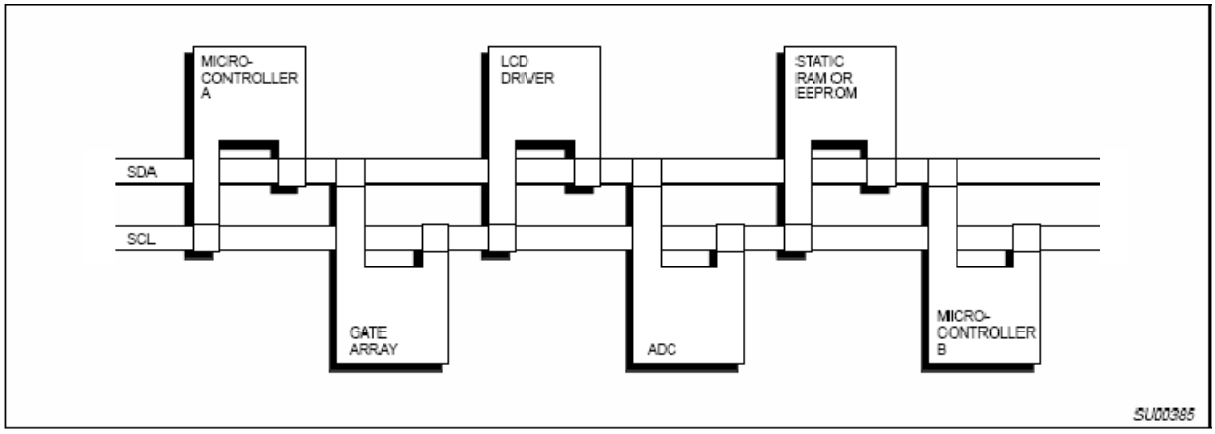


Figure 3. Example of an I²C-bus configuration using two microcontrollers

L'I2C est une communication série cadencée sur le bit SCLK. La discussion commence avec la mise a zéro de SDIN pendant un état haut de SCLK (**bit de START**), puis l'adresse du composant est envoyé, si celui-ci reconnaît son adresse il le signal en mettant SDIN à zéro (**acknoledge**). Ensuite deux mots de huit bits sont envoyés, chacun étant suivit d'une acknoledge pour confirmer leur réception. La discussion entre les 2 composants se termine par la mise à 1 de SDIN sur un état haut de SCLK (**bit de STOP**).

2-WIRE SERIAL CONTROL MODE

The WM8731/L supports a 2-wire MPU serial interface. The device operates as a slave device only. The WM8731/L has one of two slave addresses that are selected by setting the state of pin 15, (CSB).

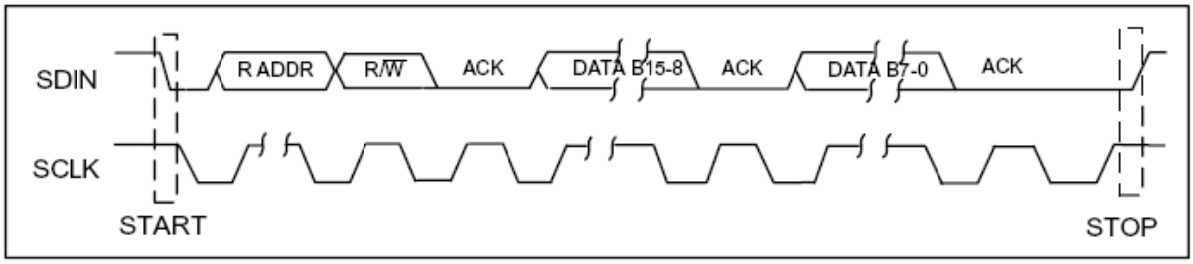


Figure 33 2-Wire Serial Interface

Note :

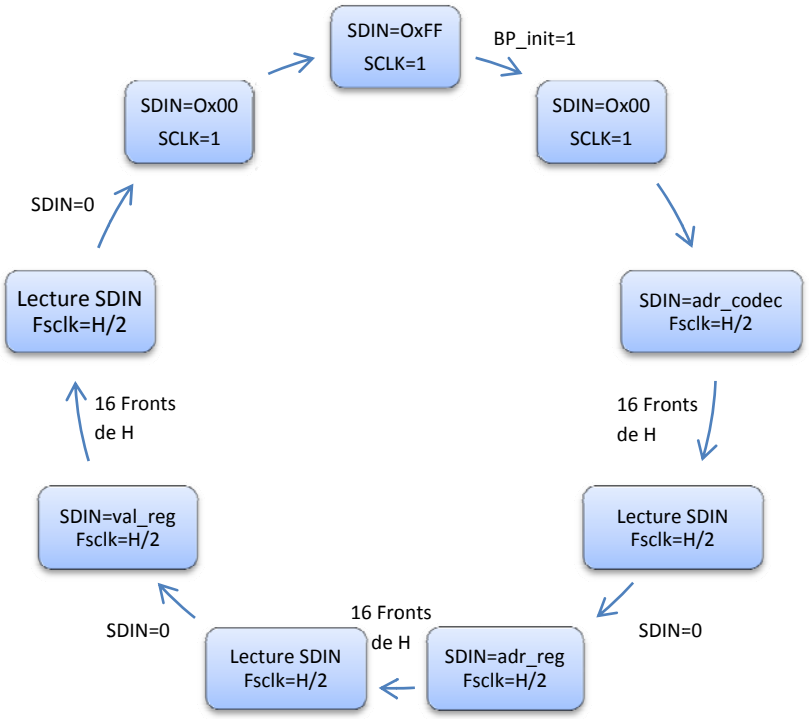
Ici, le premier mot de huit bits représente l'adresse du registre et le second, les valeurs de celui-ci.

L'adresse du Codec est donnée par sa documentation : **Ox34**

Programmation

Les deux signaux doivent pouvoir être asynchrones (pour le bit de START et e STOP), le diagramme d'état devra être cadencé sur une fréquence 2 fois plus grande que celle de SCLK.

Diagramme d'état



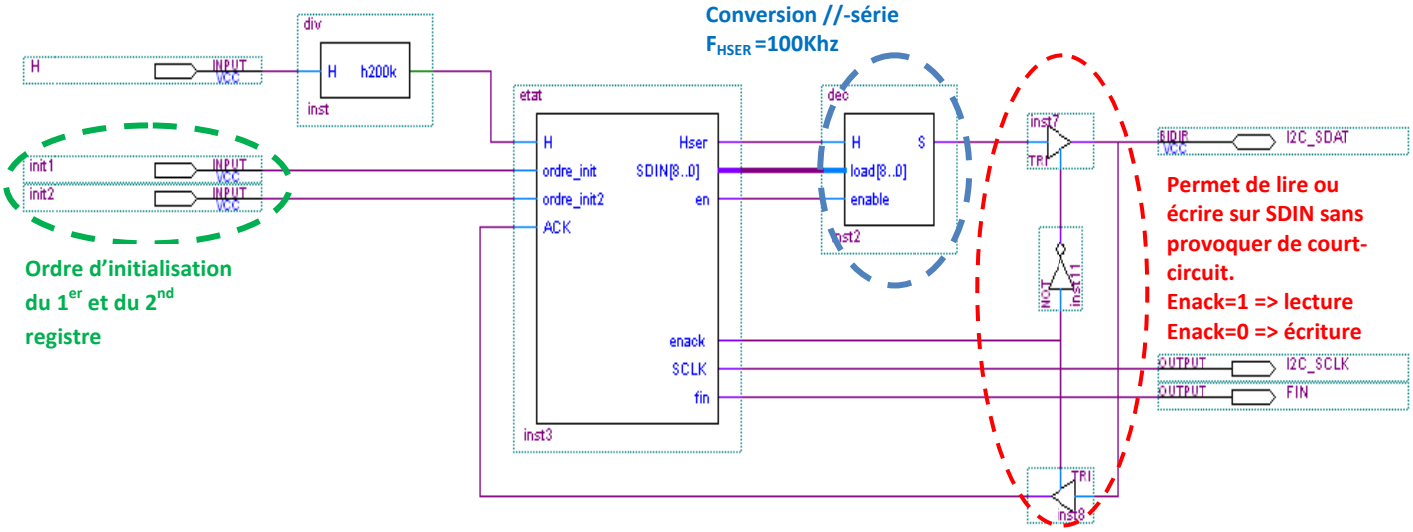
Le diagramme d'état est cadencé sur H de 200Khz.

Donc F_{SCLK} =100Khz.

SDIN est d'abord un mot de 8 bits en parallèle, il sera mit en série sur le bit SDIN par un autre registre cadencé sur SCLK.

On envoi le mot en parallèle et on attend 16 fronts sur H pour envoyer les 8 bits en série

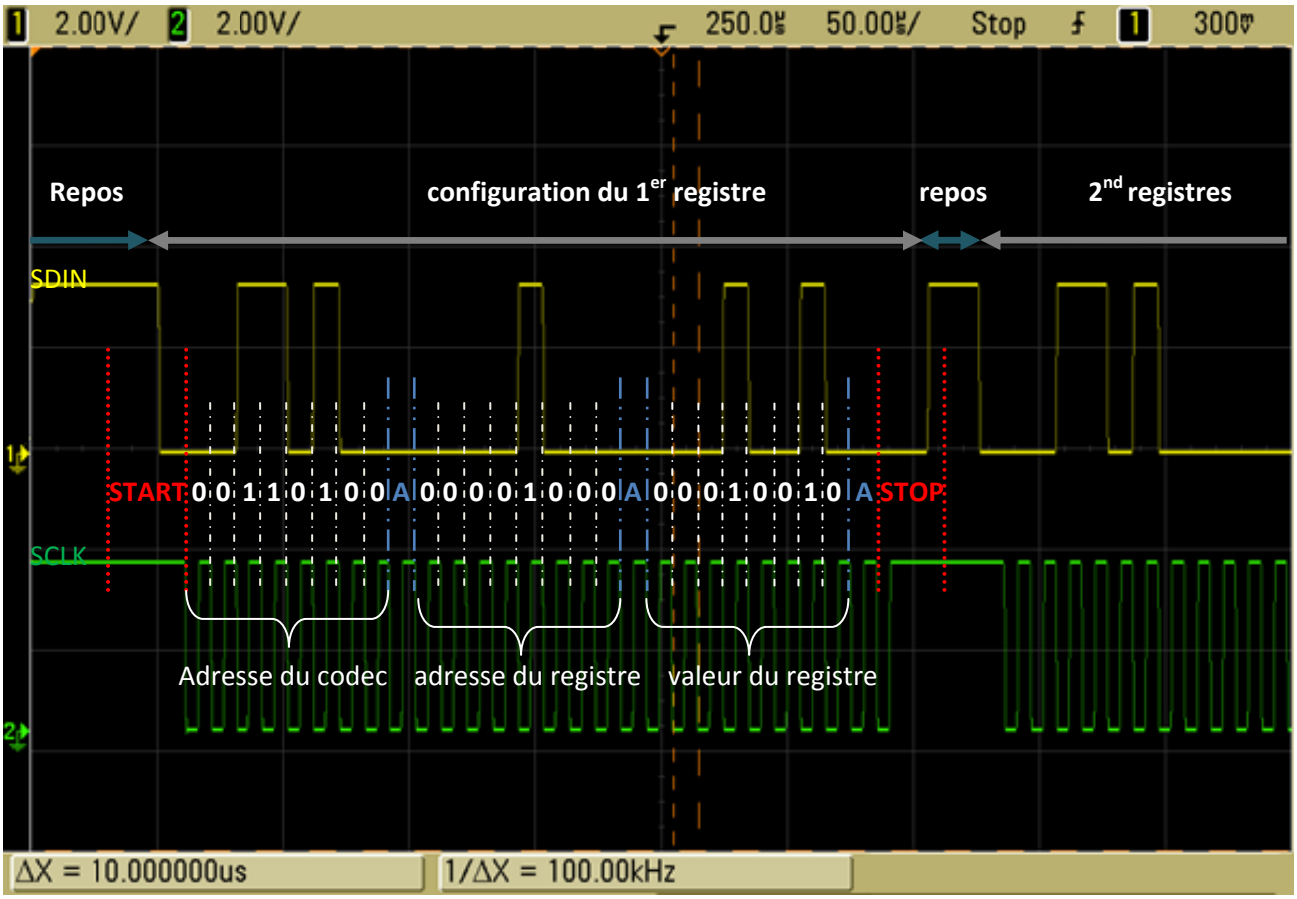
Schéma



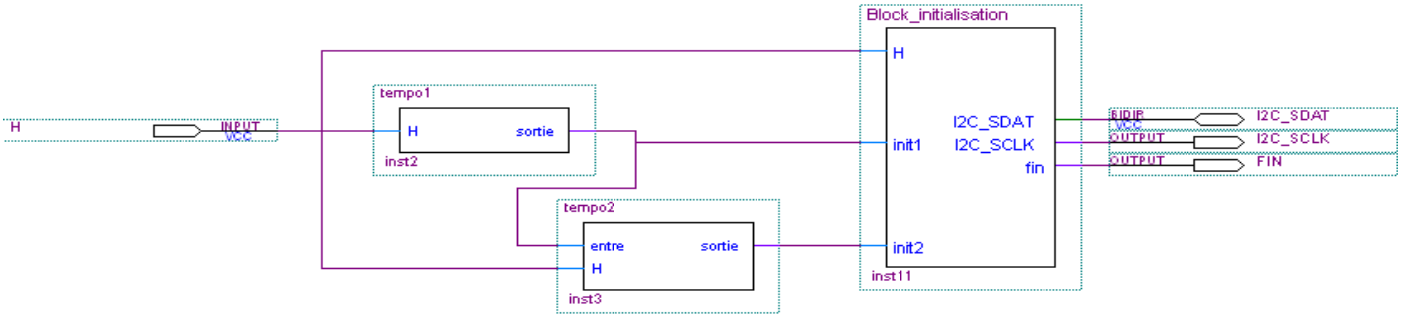
Note : 2 registres sont à initialiser, il suffit de refaire le même cycle en changeant l'adresse et la valeur du registre.

Programme en annexe

Résultat



La communication Codec/FPGA se fait bien, le Codec répond correctement à son adresse par une Acknoledge. Les mots de 8 bits sont bien compris et pris en compte, l'initialisation est effectuée.



Note :

Initialement les ordres d'initialisations étaient demandés par l'utilisateur grâce à 2 boutons poussoirs, afin de l'automatiser nous avons placé 2 tempos de 400µs.

Pour être sûr que le bit d'acknoledge soit mit à zéro par le Codec, nous écrivons un 1 puis nous lisons SDIN (si SDIN=0 alors l'acknoledge est validée).

Envoi d'un son

Une foi que l'initialisation faite, nous pouvons commencer la programmation en vue d'émettre un bip.

Voici la solution que nous avons choisie :

Création du sinus numérique

Programme

Nous créons un sinus numérisé sur 48 échantillons (nombre choisit arbitrairement) de 16 bits, le nombre de bits étant fixé par nous lors de l'initialisation.

La sortie Sin_out évoluera au rythme de l'horloge Sin_cont. En faisant varier cette horloge, nous faisons varier la fréquence du sinus et donc le son qui en résultera :

Si la F Sin_cont=21120Hz

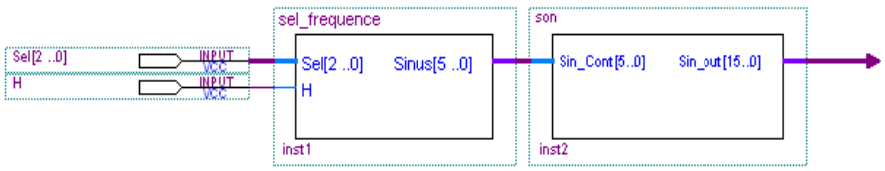
Alors F Sin=21120/48(echantillon)
= 440Hz (note la)

```
entity son is
port( Sin_Cont : in std_logic_vector(5 downto 0);
      Sin_out : out std_logic_vector(15 downto 0));
end son;

architecture sig of son is

begin
process(Sin_Cont)
begin
case Sin_Cont is
    when "000000"   =>  Sin_Out     <=      "1000000000000000"     ;
    when "000001"   =>  Sin_Out     <=      "1001000010110100"     ;
    when "000010"   =>  Sin_Out     <=      "1010000100100000"     ;
    when "000011"   =>  Sin_Out     <=      "1011000011111011"     ;
    when "000100"   =>  Sin_Out     <=      "1011111111111111"     ;
    when "000101"   =>  Sin_Out     <=      "1100110111101011"     ;
    when "000110"   =>  Sin_Out     <=      "1101101010000001"     ;
    when "000111"   =>  Sin_Out     <=      "1110010110001011"     ;
    when "001000"   =>  Sin_Out     <=      "1110111011011001"     ;
    when "001001"   =>  Sin_Out     <=      "1111011001000000"     ;
    when "001010"   =>  Sin_Out     <=      "1111101110100010"     ;
    when "001011"   =>  Sin_Out     <=      "1111111011100110"     ;
    when "001100"   =>  Sin_Out     <=      "1111111111111111"     ;
    when "001101"   =>  Sin_Out     <=      "1111111011100110"     ;
    when "001110"   =>  Sin_Out     <=      "1111101110100010"     ;
    when "001111"   =>  Sin_Out     <=      "1111011001000000"     ;
    when "010000"   =>  Sin_Out     <=      "1110111011011001"     ;
    when "010001"   =>  Sin_Out     <=      "1110010110001011"     ;
    when "010010"   =>  Sin_Out     <=      "1101101010000001"     ;
    when "010011"   =>  Sin_Out     <=      "1100110111101011"     ;
    when "010100"   =>  Sin_Out     <=      "1011111111111111"     ;
    when "010101"   =>  Sin_Out     <=      "1011000011111011"     ;
    when "010110"   =>  Sin_Out     <=      "1010000100100000"     ;
    when "010111"   =>  Sin_Out     <=      "1001000010110100"     ;
    when "011000"   =>  Sin_Out     <=      "1000000000000000"     ;
    when "011001"   =>  Sin_Out     <=      "0110111101001011"     ;
    when "011010"   =>  Sin_Out     <=      "0101111011100000"     ;
    when "011011"   =>  Sin_Out     <=      "0100111100000101"     ;
    when "011100"   =>  Sin_Out     <=      "0100000000000001"     ;
    when "011101"   =>  Sin_Out     <=      "0011001000010101"     ;
    when "011110"   =>  Sin_Out     <=      "0010010101111110"     ;
    when "011111"   =>  Sin_Out     <=      "0001110011101100"     ;
    when "100000"   =>  Sin_Out     <=      "0001000100100111"     ;
    when "100001"   =>  Sin_Out     <=      "0000100110111111"     ;
    when "100010"   =>  Sin_Out     <=      "0000010001011101"     ;
    when "100011"   =>  Sin_Out     <=      "0000001000011001"     ;
    when "100100"   =>  Sin_Out     <=      "0000000000000000"     ;
    when "100101"   =>  Sin_Out     <=      "0000001000011001"     ;
    when "100110"   =>  Sin_Out     <=      "0000010001011101"     ;
    when "100111"   =>  Sin_Out     <=      "0000100110111111"     ;
    when "101000"   =>  Sin_Out     <=      "0001000100100111"     ;
    when "101001"   =>  Sin_Out     <=      "0001110011101100"     ;
    when "101010"   =>  Sin_Out     <=      "0010010101111110"     ;
    when "101011"   =>  Sin_Out     <=      "0011001000010101"     ;
    when "101100"   =>  Sin_Out     <=      "0100000000000001"     ;
    when "101101"   =>  Sin_Out     <=      "0100111100000101"     ;
    when "101110"   =>  Sin_Out     <=      "0101111011100000"     ;
    when "101111"   =>  Sin_Out     <=      "0110111101001011"     ;
when others => Sin_Out <= "0000000000000000";
end case;
end process;
end sig;
```

Schéma

Sel[2 ..0] permet de sélectionner différentes fréquences pour le sinus :

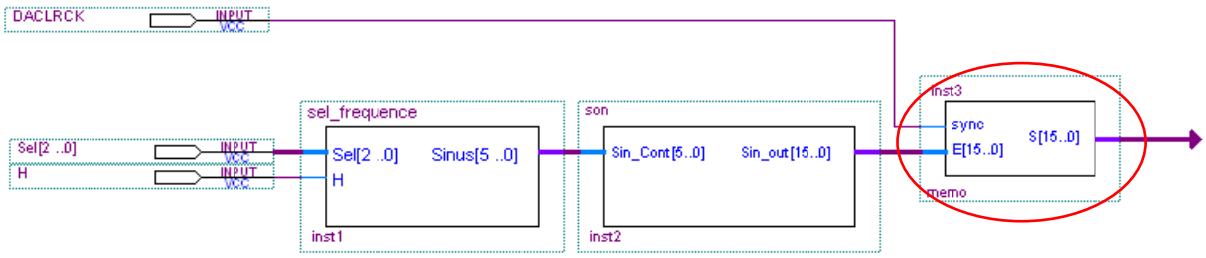
Do Ré Mi Fa Sol La Si
262 294 330 349 392 440 494 Hz



Échantillonnage et mise en série

L'envoi des données vers le Codec se fait à 48Khz (cadencement sur DACLRCK), ce sinus sera donc échantillonner à cette fréquence.

Schéma



Dans l'éventualité d'un changement du sinus pendant l'envoi en série, ce qui engendrerait un échantillon erroné, nous utiliserons une mémorisation.

À chaque front de DACLRCK (qui indique l'envoi d'un échantillon) le registre « memo » copiera le sinus afin de l'envoyer vers la conversion parallèle/série.

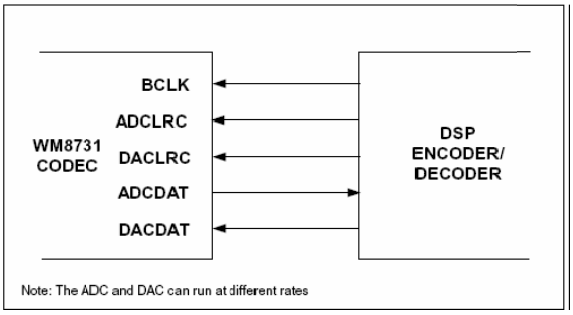
Programme

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

entity memo is
    port( E     : in  std_logic_vector(15 downto 0);
          sync  : in std_logic;
          S     : out std_logic_vector(15 downto 0));
end memo;

architecture mem of memo is


signal X : std_logic_vector (4 downto 0);
constant N : integer := 16;
signal go: std_logic;


begin

process(sync)
begin

    if (sync'event and sync='1') then

        S <= E;

    end if;

end process;

end mem;
```

Conversion parallèle/série

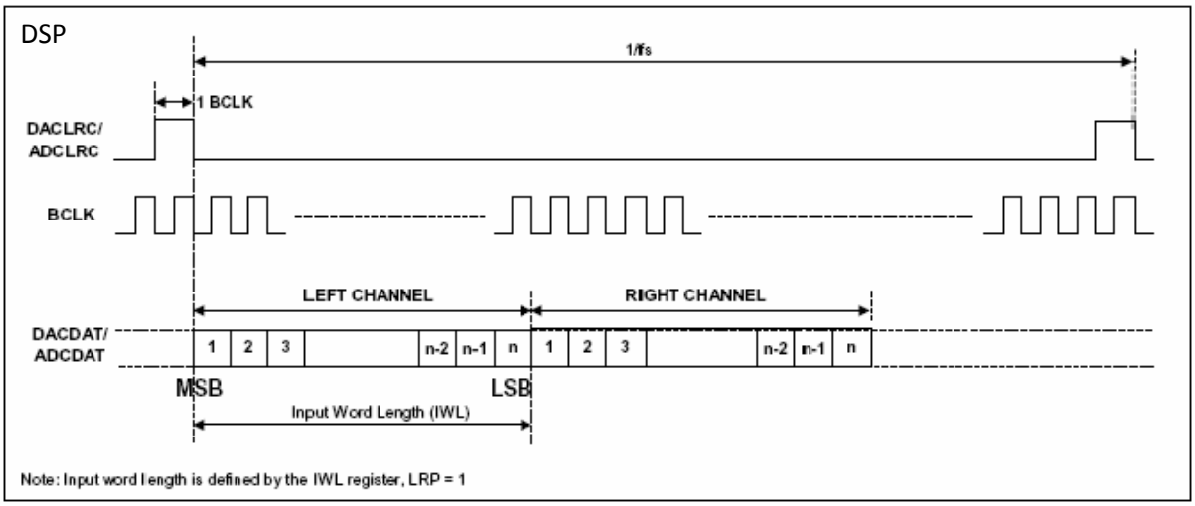
L'envoi des échantillons doit se faire suivant un protocole choisit lors de l'initialisation. Ici nous avons choisit le mode Esclave du Codec et le mode DSP de communication :

DIGITAL AUDIO INTERFACE – SLAVE MODE



La mise en esclave du Codec impose au FPGA de générer lui-même les signaux de synchronisme DACLRC et BCLK, afin d'envoyer les données sur DACDAT.

(ADCDAT et ADCLRC ne servent pas car ils sont utiles lors de l'enregistrement de données issues du microphone.)



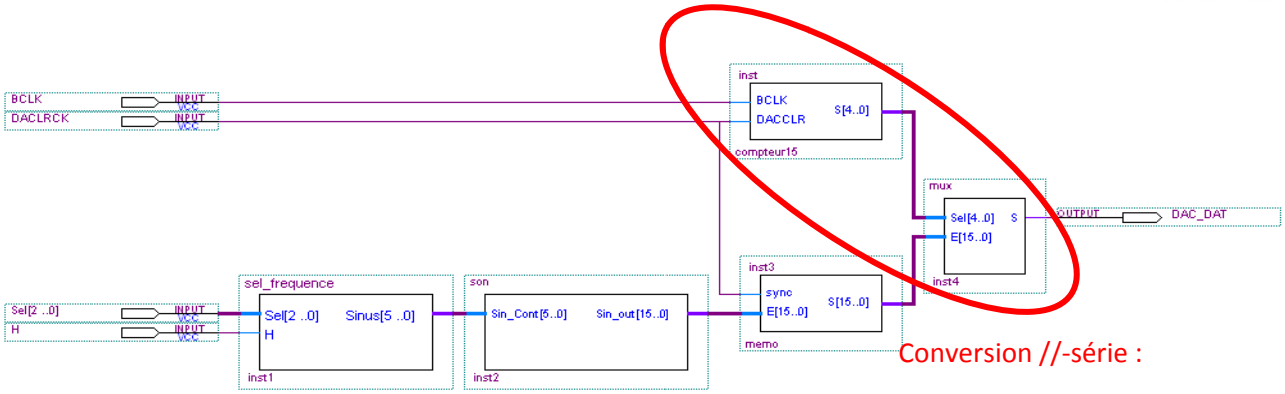
Le mode DSP fonctionne ainsi :

- Le début de l'envoi est indiqué sur un front descendant de DACLRC
- Les bits sont envoyés lors des fronts descendants de BCLK afin d'avoir un bit stable lors de la lecture sur front montant.
- Le bit de poids fort est envoyé en premier
- La voie gauche est envoyer en premier (peut être inversé si initialisation du registre R7), ici la voie de gauche est laissée a zéro.

Note :

Pour facilité la programmation, le début de l'envoi peut commencer au second front de BCLK après le passage a zéro de DACLRC. Ceci en initialisant le bit LRP du registre R7. Voir chapitre initialisation.

Schéma



Conversion //-série :

Multiplexeur commandé par un synchroniseur

Programmes

Synchroniseur

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

entity compteur15 is
    port( DACCLR, BCLK: in std_logic;
          S : out std_logic_vector(4 downto 0));
end compteur15;

architecture compt of compteur15 is


signal X : std_logic_vector (4 downto 0);
constant N : integer := 16;
signal go: std_logic;


begin

process(BCLK, DACCLR)
begin

    if (BCLK='0' and DACCLR='1')                then
                go<='1';
                X<="00000";

    else
        if (BCLK'event and BCLK='0' and go='1') then
            X<=X+1;
            if X=N then
                go <='0';
            end if;
        end if;
    end if;


S<=X-1;

end process;

end compt;
```

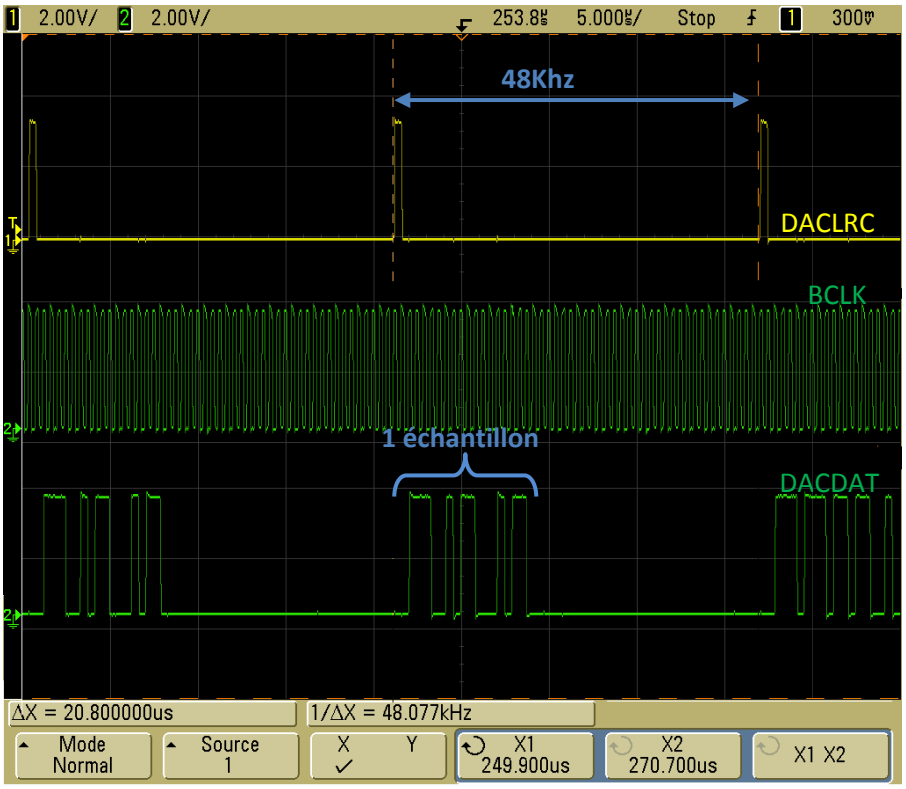
Multiplexeur

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

entity mux is
    port( Sel   : in std_logic_vector(4 downto 0);
          E     : in std_logic_vector(15 downto 0);
          S     : out std_logic);
end mux;

architecture arch of mux is
begin

with sel select
    S   <=  E(15)   when "00000",
            E(14)   when "00001",
            E(13)   when "00010",
            E(12)   when "00011",
            E(11)   when "00100",
            E(10)   when "00101",
            E(9)    when "00110",
            E(8)    when "00111",
            E(7)    when "01000",
            E(6)    when "01001",
            E(5)    when "01010",
            E(4)    when "01011",
            E(3)    when "01100",
            E(2)    when "01101",
            E(1)    when "01110",
            E(0)    when "01111",
        '0'     when others;
end arch;
```

Le synchroniseur commence à compter de 0 à 15 à partir du front descendant de DACCLR, cadencé sur BCLK. Le multiplexeur envoi donc du MSB au LSB de l'échantillon sur vers bit DACDAT en fonction du synchroniseur.

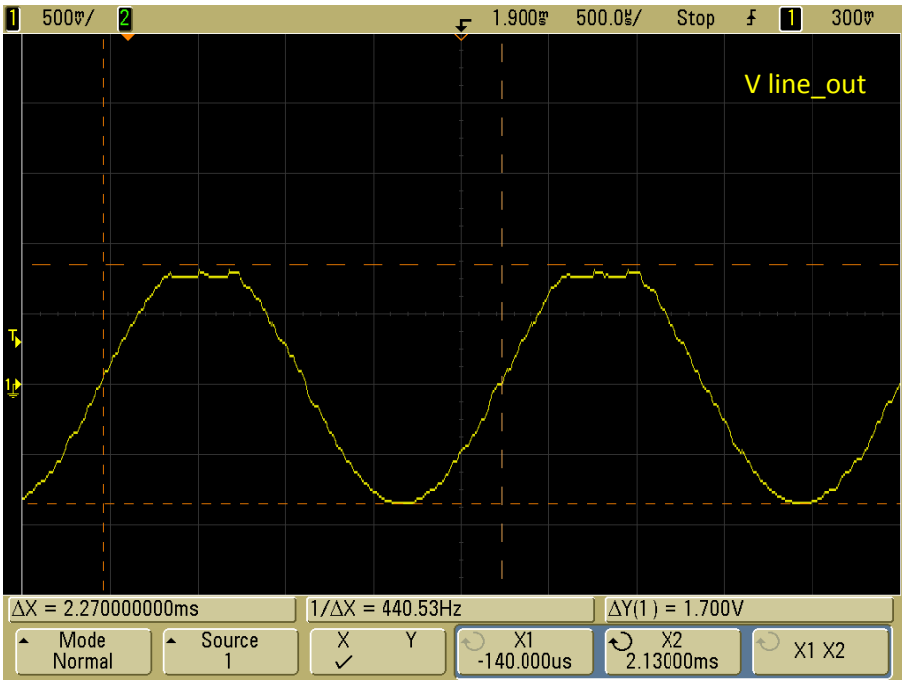
Résultat

Entrées numériques du codec



Fbclk> 48Khz * 2 * 16
Fdaclrc *2mots de 16 bits

Sortie analogique du codec



Note :
On peut observer des marches d'escalier sur le sinus, elles sont dues au faible nombre d'échantillons (48) sur une période. Ces marches sont légèrement arrondies grâce au divers filtre interne du codec.

Améliorations possibles

Une amélioration facile à réaliser est d'utiliser la partie analogique-numérique du Codec avec un microphone. Nous pourrions alors enregistrer un son en mémoire puis le renvoyer en tant que l'alarme sonore.

Fonctionnement :

Ecrire 48000 échantillons du microphone en mémoire, ce qui correspond à une seconde si la fréquence d'échantillonnage est à 48kHz, puis en les renvoyer à chaque prise de photo vers line_out.

Grâce au programme d'écriture et de lecture réalisé par José il serait facile de mettre en œuvre ce système.

Une autre amélioration possible serait d'enregistrer un son en MP3 en mémoire puis le renvoyer sur line_out. Cependant il serait nécessaire de créer un programme afin de décompresser le format MP3.

Conclusions

Pour conclure, le projet n'a pas pu être entièrement complété, ce qui est du surtout à un manque de temps et à la complexité de celui-ci. Même si certaine parti du projet final sont entièrement fonctionnel comme le générateur de son. Mais finalement ce qu'il faut retenir de ce projet est ce que nous avons appris pendant sa réalisation. Tout ce que nous avons appris nous servira surement un jour dans le monde du travail.

Summary

This year we were given the development of a new project. This project falls into consideration as a part to obtain our bachelors degree diploma. During the middle of the year we were proposed multiple projects subjects, all regarding programming in different languages (C++, VHDL, etc.) But there was one main difference between all this projects, some were energy conservation oriented and others more digital electronics oriented.

The project we chose was more hardware design oriented. Our first specification chart was to develop a program that based on movement detection could send a recording order to a digital camera. At first the project seemed very interesting, but after a couple of weeks we were informed that the camera was not available during our project. So we were given another specification chart but this time, we had to use a ALTERA camera, this meant we had to, not only send a recording message to a camera, but control everything's, camera-board communication, picture display, etc. It seemed interesting and challenging at the time. We had to work in a ALTERA DE1 board, which is a development board produced to study the functions of a FPGA Cyclone II. This board also features multiples components such as: Three different types of memory, a VGA controller, an audio controller and many components useful to take most profit of the FPGA.

At first what we did was cut the work in half, as we taught that working separately was more efficient that working together. I took the display and memory management part while my partner took the audio and camera management part.

First of all the memory management. On this DE1 board we are able to use three different types of memory: SRAM, FLASH and SDRAM memory. I first started working with the SDRAM memory which is the easiest to use. After reading the datasheets, I was able to start programming my FPGA with the VHDL language. I made a program that could reproduce the read and write cycles from this memory. I made a simulation and program my FPGA. It all worked very fine, I was now able to write data directly into the memory using the switches on the board to chose the data to be written. After a couple of upgrades, my final program was able to function in two modes, the test mode and the working mode. The working mode saves data in a continuous way, while the test mode only saves one data at the time. But the problem was that the size of this memory was not enough to hold a picture at a VGA format. Next of I started working with the flash memory. This memory chip could hold a lot more data that the other one, and also could data even when the board was turned off. This memory had a lot more functionalities that the SDRAM, but in the end managing this memory was exactly the same as the SRAM, so I just copied the previous program and I modify some minor stuff, and this memory was ready to be used. But I could not use it either for my project because it had a very long access time, which was fatal if I wanted to use it to display a picture. Finally I moved on to the third and final memory the SDRAM. This was much more complex to use because of it internal design. This memory offer a lot more capabilities that previous memory, it's cheaper and can be integrated in larger amounts that the previous two memories. This memory does not use « flip-flop logic » to hold the data, but in uses capacitors, that leak and eventually will

lose the data after a period of time. Because of this I had to integrated in my program an refresh memory command every 7,8 µs. Besides this, this new memory doesn't work at all like the previous ones, so I had to start from scratch to make this new memory controller. It took me a lot of time to understand how this memory actually worked, and it also took me a lot of time making it work. But in the end I managed to make a program to work even if it took a lot of time and 600 lines of code.

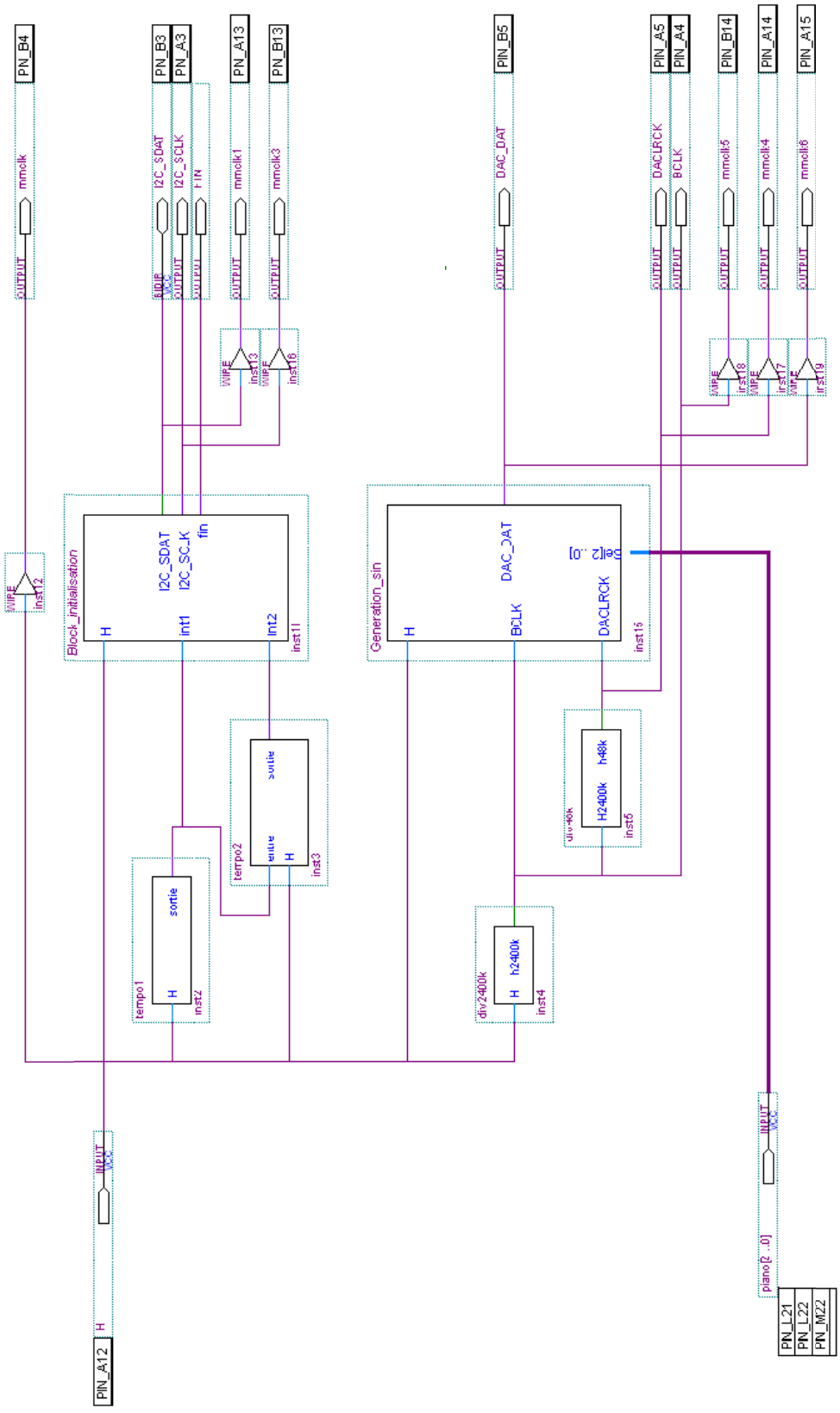
Next off I will present the VGA controller. To be able to display the picture we were supposed to communicate with the screen. For this we had to create a VGA controller. At first we tried to understand how this format worked. We did a search on the internet and we found a document explaining how to it worked. Based on this document I wrote a program able to reproduces the display cycles of the screen. The image is displayed line by line at a frequency of 50 Hz to display the whole screen. Finally I made a program that allowed me to display a single color on the screen. This color is chosen with the switch. After some modification made to my program, I was able to display the data held in the memory, at the time it were just random colors, because we weren't able to take a pictures just yet.

Finally the audio management part proved to be more difficult that we taught. First off we had to understand how the audio controller worked. It was the first time that I was using this kind of device, I had to read the whole datasheet to try to understand, I also had to do some internet searching. Finally after a long time figuring out how this device worked I started programming it. First off we have to initialize the device which is done by the I2C standard. At the time I didn't know how I2C worked, I had to do some research to finally get it. Then after the initialization was complete I need it to create a digital sinus in VHDL. To do this we must chose a sampling frequency and the number of samples needed to make the sin signal sound just right. After this I started working on the configuration of the device itself. I had to choose between several the operating mode, frequency, etc. After I put it all together in a single program and I started the testing process. I was able to see my control signals trough a scope, and a bit analyzer. After the testing was over I made some modification to my final program and it was ready to go. At first it only issued one sound, but afterwards I modified did so that the user could choose the frequency of the sound he wanted to hear. In the end my system worked has a sound synthesizer. As for the camera management we didn't got the time to work on it, has we received the camera quite late.

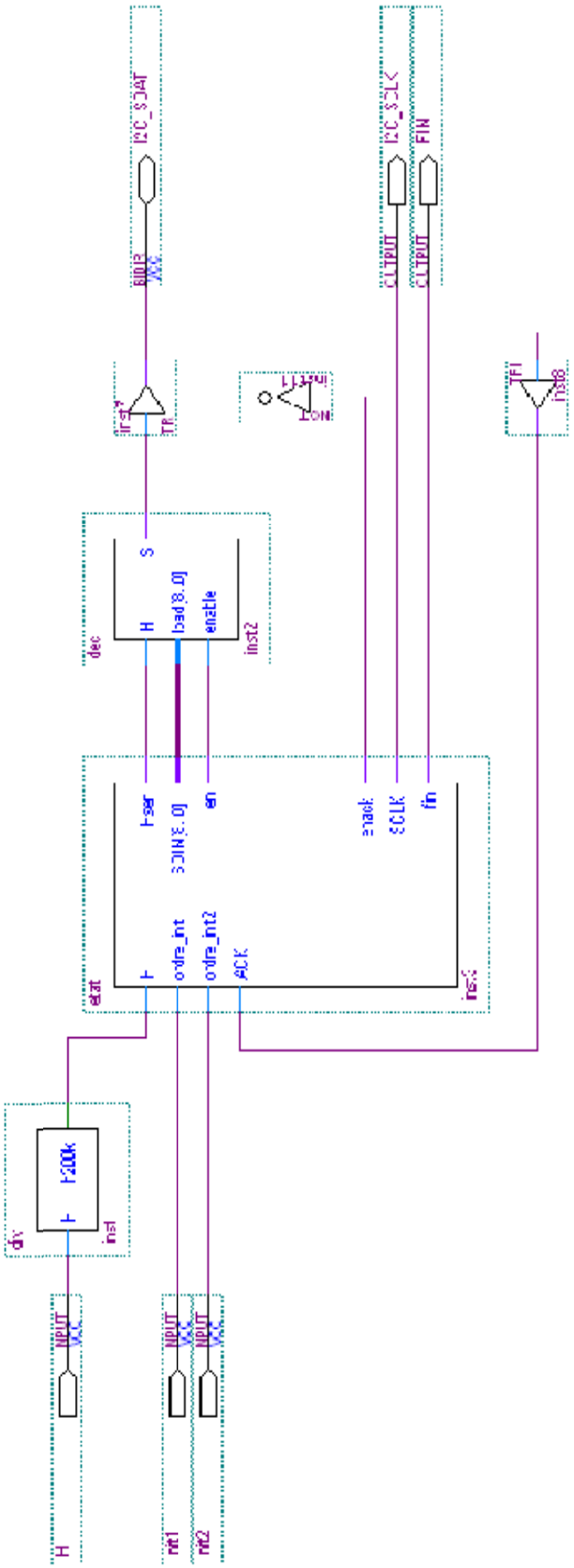
To conclude we would like to say that, we did not get the chance to finish our project, we had some parts that worked but not the whole thing. Thought it was really a hard project we enjoyed developing it, because we learned many great things about how memory, VGA, and sound controllers work. This will hopefully be one day very useful at work. Our final thought is that even we did not have a working prototype at the end; we gain some invaluable experience doing this project.

Annexes

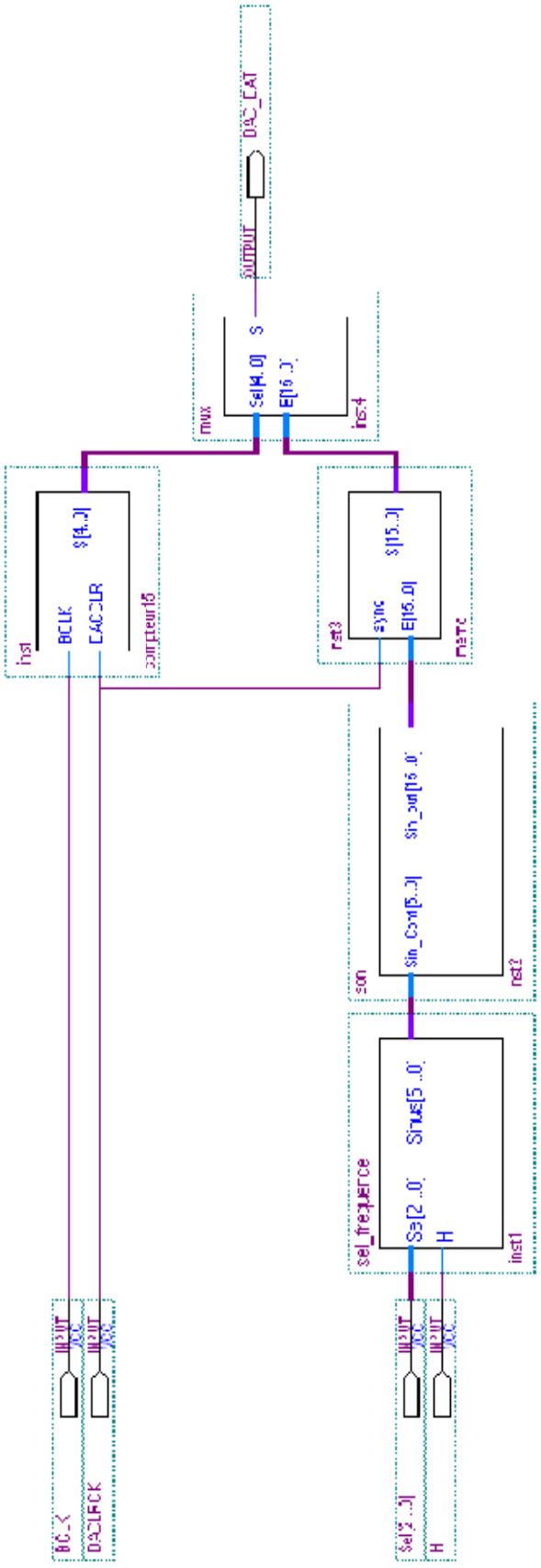
Block fonction sonore



Block initialisation



Block génération et envoi du sinus



Programme machine d'état d'initialisation

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

entity etat is
    port( H         : in    std_logic;                      -- horloge de la machine d'etat
          ordre_init: in    std_logic;
          ordre_init2: in   std_logic;
          ACK       : in    std_logic;                      --bit de lecture de SDIN
          enack     : out   std_logic;                      --validation de la lecture de SDIN
          en        : out   std_logic;                      --validation de la conversion //-serie
          Hser      : out   std_logic;                      --horloge de la conversion //-serie
          SCLK      : out   std_logic;                      --sortie SCLK
          fin       : out   std_logic;                      --fin d'initialisation
          SDIN      : out   std_logic_vector(8 downto 0));  --sortie parallele de SDIN
end etat;

architecture arch of etat is

constant        addr_codec  : std_logic_vector  := "001101001";     -- : adresse du codec       + bit d'ack
constant        addr_reg_R4 : std_logic_vector  := "000010001";     -- : adresse du registre R4 + bit d'ack
constant        val_reg_R4  : std_logic_vector  := "000100101";     -- : valeur  du registre R4 + bit d'ack
constant        addr_reg_R7 : std_logic_vector  := "000011101";     -- : adresse du registre R7 + bit d'ack
constant        val_reg_R7  : std_logic_vector  := "000100111";     -- : valeur  du registre R7 + bit d'ack
signal          A           : std_logic_vector(5 downto 0):= "000000";  -- : signal interne de SCLK
signal          B           : std_logic_vector(5 downto 0):= "000000";  -- : signal interne de Hser
signal          stop        : std_logic:= '0';                  -- : singal interne de fin
shared variable etat        : integer;                          -- : indicateur d'etat
shared variable X           : integer;                          -- : Compteur

begin

process(H)
begin

    if (h'event and h = '1') then                               -- synchronisation sur une horlo
                                                                -- 2 fois plus rapide que SCLK
                                                                -- pour bit de START et de STOP
      B<=B+1;
      case etat is

      When 0 =>   SDIN<="111111111";                            -- etat de repos
                  en<='1';
                  A<="000001";
                  if ordre_init='0' and stop ='0'     then    etat:=1;
                                                              X:=0;
                                    else                      etat:=0;
                  end if;

      When 1 =>   en<='0';
                  X:=X+1;
                  if X=3                  then    etat:=2;
                                                  X:=0;
                                          else    etat:=1;
                  end if;

      when 2 =>   SDIN<="000000000";                            -- passage a 0 de SDIN
                  en<='1';                                      -- durant etat 1 de SCLK
                  etat:=3;                                      -- => bit de START


      when 3 =>   A<=A+2;
                  en<='0';
                  X:=X+1;
                  if X=1                  then    etat:=4;
                                                  X:=0;
                                          else    etat:=3;
                  end if;


      When 4  =>  A<=A+1;
                  SDIN<=addr_codec;                             -- envoi de adresse du codec
                  if X>0 then en<='1';
                  else en<='0';
                  end if;
                  X:=X+1;
                  if X>=17                then    etat:=5;
                                                  enack<='1';
                                                  X:=0;
                                          else    etat:=4;
                                                  enack<='0';
                  end if;

      when 5  =>  A<=A+1;                                       -- lecture de l'acknoledge
                  enack<='1';
                  if ACK='0'              then    etat:=6;
                                          else    etat:=5;
                  end if;


      when 6  =>  A<=A+1;                                       -- envoi de l'adresse du 1° regi
                  SDIN<=addr_reg_R4;
                  if X>0                  then    en<='1';
                                                  enack<='0';
                  else en<='0';
                  end if;
                  X:=X+1;
                  if X>=17                then    etat:=7;
                                                  enack<='1';
                                                  X:=0;
                                           else   etat:=6;
                                                  enack<='0';
                  end if;

      when 7  =>  A<=A+1;                                       --lecture de l'acknoledge
                  enack<='1';
                  if ACK='0'              then    etat:=8;
                                          else    etat:=7;
                  end if;

      when 8  =>  A<=A+1;                                       -- envoi des valeur du 1° regist
                  SDIN<=val_reg_R4;
                  if X>0                  then    en<='1';
                                                  enack<='0';
                  else en<='0';
                  end if;
                  X:=X+1;
                  if X>=17                then    etat:=9;
                                                  enack<='1';
                                                  X:=0;
                                           else   etat:=8;
                                                  enack<='0';
                  end if;

      when 9  =>  A<=A+1;                                       --lecture de l'acknoledge
                  enack<='1';
                  if ACK='0'              then    etat:=10;

                                          else    etat:=9;
                  end if;

      when 10 =>  A<=A+1;
                  X:=X+1;
                  SDIN<="000000000";
                  en<='1';
                  enack<='0';
                  if X=2                  then    etat:=11;
                                                  X:=0;
                                          else    etat:=10;
                  end if;

      when 11 =>  A<="000001";                                  -- mise a 1 de SCLK
                  SDIN<="000000000";
                  en<='1';
                  enack<='0';
                  etat:=12;

      when 12 =>  en<='0';
                  A<="000001";
                  SDIN<="000000000";
                  etat:=13;

      When 13 =>  SDIN<="111111111";                            -- passage a 1 de SDIN
                  en<='1';                                      -- durant un etat 1 de SCLK
                  A<="000001";                                  -- => bit de STOP
                  if ordre_init2='0'  then    etat:=14;
                                              X:=0;
                                      else    etat:=13;
                  end if;

      When 14 =>  en<='0';
                  X:=X+1;
                  if X=3                  then    etat:=15;
                                                  X:=0;
                                          else    etat:=14;
                  end if;

      when 15 =>  SDIN<="000000000";
                  en<='1';
                  etat:=16;

      when 16 =>  A<=A+2;
                  en<='0';
                  X:=X+1;
                  if X=1                  then    etat:=17;
                                                  X:=0;
                                          else    etat:=16;
                  end if;

      When 17  => A<=A+1;
                  SDIN<=addr_codec;
                  if X>0 then en<='1';
                  else en<='0';
                  end if;
                  X:=X+1;
                  if X>=17                then    etat:=10;
                                                  enack<='1';
                                                  X:=0;
                                          else    etat:=17;
                                                  enack<='0';
                  end if;

      when 18  => A<=A+1;
                  enack<='1';
                  if ACK='0'              then    etat:=19;
                                          else    etat:=18;
                  end if;

      when 19  => A<=A+1;
                  SDIN<=addr_reg_R7;
                  if X>0                  then    en<='1';
                                                  enack<='0';
                  else en<='0';
                  end if;
                  X:=X+1;
                  if X>=17                then    etat:=20;
                                                  enack<='1';
                                                  X:=0;
                                           else   etat:=19;
                                                  enack<='0';
                  end if;

      when 20  => A<=A+1;
                  enack<='1';
                  if ACK='0'              then    etat:=21;
                                          else    etat:=20;
                  end if;

      when 21  => A<=A+1;
                  SDIN<=val_reg_R7;
                  if X>0                  then    en<='1';
                                                  enack<='0';
                  else en<='0';
                  end if;
                  X:=X+1;
                  if X>=17                then    etat:=22;
                                                  enack<='1';
                                                  X:=0;
                                           else   etat:=21;
                                                  enack<='0';
                  end if;

      when 22  => A<=A+1;
                  enack<='1';
                  if ACK='0'              then    etat:=23;

                                          else    etat:=22;
                  end if;

      when 23  => A<=A+1;
                  X:=X+1;
                  SDIN<="000000000";
                  en<='1';
                  enack<='0';
                  if X=2                  then    etat:=24;
                                                  X:=0;
                                          else    etat:=23;
                  end if;

      when 24  => A<="000001";
                  SDIN<="000000000";
                  en<='1';
                  enack<='0';
                  stop<='1';
                  etat:=25;

      when 25  => en<='0';
                  A<="000001";
                  SDIN<="000000000";
                  etat:=0;

      when others => etat :=0;
      end case;
      end if;
  end process;
  fin  <= stop;
  Hser <=B(0);
  SCLK <=A(0);
end arch;
```