ÉCOLE
POLYTECHNIQUE
UNIVERSITÉ PARIS-SACLAY

Conception expérimentale micro et nanoélectronique

Eduardo Guimarães

Mauricio García

Index

Introduction

Ce rapport traite du projet développé dans le cadre du cours Conception expérimentale micro et nanoélectronique. Nous devrions définir un projet à réaliser avec la carte FPGA et à développer en VHDL qui devrait être intéressant et dont la réalisation nous auraient aidé à apprendre d'importants concepts d'applications pratiques de projets électroniques.

Dans ce contexte, nous avons décidé de créer un jeu utilisant la carte FPGA. Pour ce faire, nous utilisons, en plus de la carte, un moniteur VGA pour la visualisation du jeu et un contrôleur de jeu vidéo NES pour contrôler un personnage à l'écran.

Dans ce rapport, nous expliquerons les différentes parties du projet impliquées et l'importance de leur connexion pour le fonctionnement de l'ensemble.

Pour créer le jeu, nous devons manipuler quelques composants différents, ainsi qu'une logique interne au module de jeu principal, qui comprend le composant qui contrôle la connexion au moniteur VGA. À partir de cet élément principal, nous établissons la connexion avec le moniteur et contrôlons les signaux envoyés. De plus, nous nous connectons à des instances d'autres composants, chargées de créer une fréquence de 25 MHz, de nous connecter à la manette NES, de lire d'après la mémoire les images des personnages et les blocs affichés à l'écran (blocs qui construisent l'environnement du jeu) et détermination de la position de chaque bloc.

À travers cette connexion, le composant principal, simplement appelé "VGA", obtient les signaux et les valeurs nécessaires pour contrôler ce qui doit être affiché à l'écran. Avec ces signaux, nous créons une logique qui détermine chaque pixel à afficher.

Jeu Vidéo

Inspiration

L’environnement dans lequel se déroule notre jeu s’inspire de jeux comme Super Mario Bros et Super Mario World, dans lesquels un personnage court et saute sur des blocs. Voir la comparaison faite ci-dessous entre une scène de Super Mario Bros et notre jeu.

Conception du jeu

Ainsi, chaque plate-forme est constituée de plusieurs blocs individuels. Les informations relatives à la position sur l'écran de chaque plate-forme sont fournies au module principal par un simple composant auxiliaire appelé Environnement. Ce composant transmet quatre arrays d’entiers (avec cinq entiers dans chaque array) au module VGA principal: pos_x,

pos_y, size_x et size_y. Les deux premiers sont des arrays contenant les positions sur l'écran du centre de chaque plate-forme. Chaque valeur de pos_x est la position du pixel dans lequel sera le centre, sur l'axe des x, de chaque plate-forme. De manière analogue à pos_y, chaque valeur correspond au numéro de la ligne dans laquelle le centre est, sur l'axe des ordonnées, de chaque plate-forme.

Les arrays size_x et size_y contiennent la taille, en x et y, de chaque plate-forme, c'est-à-dire la longueur et la hauteur de chaque plate-forme. Nous avons choisi de transmettre ces valeurs également sous forme de arrays pour le cas où nous aurions décidé d'utiliser des plates-formes de tailles différentes (en fait, la plate-forme centrale est plus grande que les quatre autres).

L'affichage de chaque bloc est effectué en utilisant les positions et les tailles de chaque plate-forme, en prenant les informations des pixels à travers la ROM de la carte FPGA. Ainsi, dans notre module principal, nous incluons une logique qui vérifie si le pixel actuel se trouve dans l'une des régions indiquées comme faisant partie de l'une des régions de la plate-forme, dont les informations proviennent du composant Environnement. Ainsi, si le pixel actuel se trouve dans une région de la plate-forme, nous prenons les informations de pixel correspondantes dans la mémoire ROM, qui contient les données de couleur des blocs.

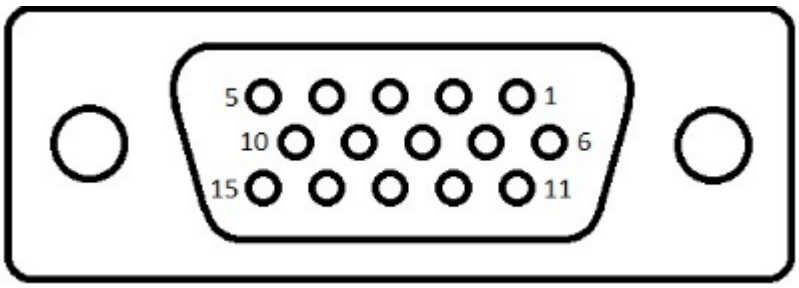
En plus de ces cinq plateformes, le jeu a une partie de terrain avec un comportement légèrement différent. Le personnage peut également courir et sauter dessus, mais peut également descendre et y entrer. L'affichage de cette région est très simple. Lorsque la valeur de v_count (qui compte le numéro de la ligne en cours) est inférieure à une certaine valeur, nous faisons en sorte que les signaux VGA_R, VGA_G et VGA_B prennent les valeurs correspondant à la couleur bleue ; lorsque v_count dépasse cette valeur, VGA_R, VGA_G et VGA_B se voient attribuer les valeurs relatives à la couleur brune vue dans la figure ci-dessus.

Périphériques

Pour pouvoir créer ce jeu, nous avons utilisé des différents composants liés à un "module" central appelé VGA. Maintenant nous allons expliquer en détail le fonctionnement interne de chacune de ces parties, en précisant même l'utilisation des différents signaux et variables qui sont à l'intérieur de chaque partie, aussi comme la logique nécessaire pour le fonctionnement du système comme un tout complet.

Connexion au moniteur VGA

La connexion à un moniteur VGA s'effectue via 15 broches différentes, comme indiqué dans le tableau ci-dessous:



Pin	Signal	Description	Connection
1	R	analog red, 0-0.7V	DAC output
2	G	analog green, 0-0.7V or 0.3-1V (if sync-on-green)	DAC output
3	B	analog blue, 0-0.7V	DAC output
4	EDID Interface	function varies depending on standard used	no connect
5	GND	general	GND
6	GND	for R	GND
7	GND	for G	GND
8	GND	for B	GND
9	no pin	or optional +5V	no connect
10	GND	for h_sync and v_sync	GND
11	EDID Interface	function varies depending on standard used	no connect
12	EDID Interface	function varies depending on standard used	no connect
13	h_sync	horizontal sync, 0V/5V waveform	FPGA output
14	v_sync	vertical sync, 0V/5V waveform	FPGA output
15	EDID Interface	function varies depending on standard used	no connect

Nous avons donc dû établir les connexions appropriées dans le cadre de notre projet, mais le plus important était de contrôler les valeurs des couleurs R, G et B, via des signaux de sortie std_logic_vector de 8 bits nommés VGA_R, VGA_G et VGA_B, car ce sont ces ports qui déterminent la couleur de chaque pixel individuellement.

Nous devons d'abord savoir comment les images sont envoyées au moniteur VGA. Qu'est-ce qui se passe ici est qu'il n'y a pas d'envoi continu de vidéo. Chaque frame est construit individuellement sur l'écran. De plus, chaque frame est construit pixel par pixel, la couleur de chaque pixel étant déterminée par les broches des signaux de sortie VGA_R, VGA_G et VGA_B.

Pour la construction de chaque image, nous parcourons chaque pixel d'une ligne de l'écran. À la fin de chaque ligne, nous avons un signal de polarisation indiquant la fin de la ligne, il correspond au h_sync du tableau ci-dessus, que nous avons appelé VGA_HS. Nous

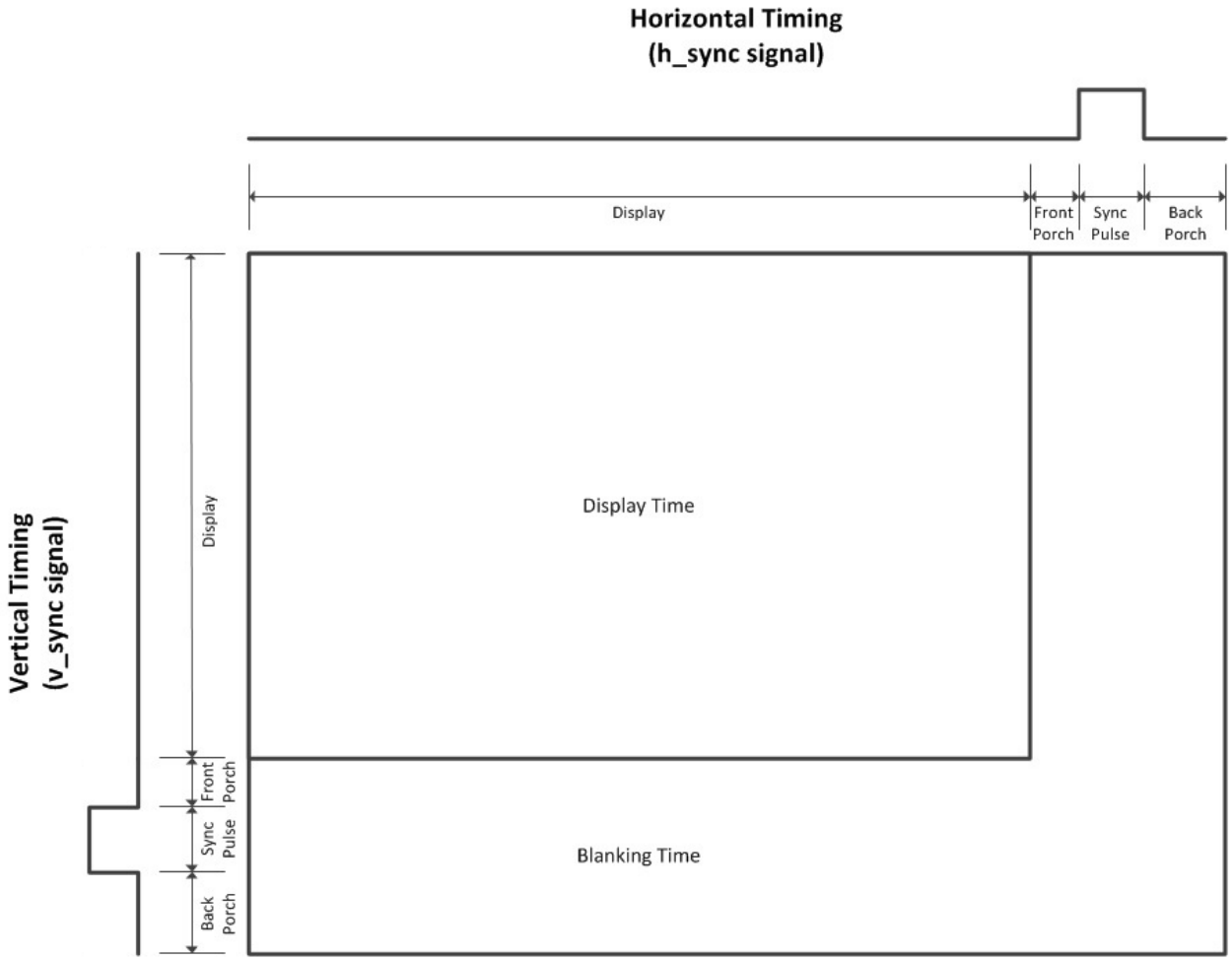
parcourons donc chaque pixel d'une ligne ; à la fin de chaque ligne, nous passons par une période de transition durant laquelle h_sync passe de 1 à 0 (et retournant à 1 encore pendant la transition) et pendant laquelle il n'y a pas d'écriture de valeur associée à un pixel de l'écran. Après cette période, nous passons à la ligne suivante et continuons le processus en "écrivant" ligne par ligne à l'écran. Une fois que toutes les lignes ont été écrites, de manière similaire à ce qui se passe pour chaque ligne, nous passons par une période de transition au cours de laquelle les signaux ne correspondent à aucun pixel, pendant cela un autre signal, v_sync (que nous appelons VGA_VS), passe de 1 à 0, revenant à 1 encore pendant cette période de transition qui marque la fin de l'écriture de tout un écran, c'est-à-dire d'une image vidéo.

Ainsi, nous écrivons ligne par ligne jusqu'à la fin de l'écran, c'est-à-dire qu'à la fin de chaque ligne et après l'écriture de toutes les lignes, nous traversons une courte période de transition. C'est comme si nous avions ce que nous pouvons comparer à des pixels qui ne sont pas réellement affichés, comme s'il s'agissait de pixels situés hors de la région réellement visible.

De cette manière, nous devons déterminer à chaque moment à quel pixel de l'écran nous nous référons. Pour cela, nous utilisons deux compteurs : h_count et v_count. h_count a une plage allant de zéro à h_period - 1, où h_period correspond à la valeur totale des "pixels" contenus dans chaque ligne, ce qui inclut tous les pixels visibles et la période de transition qui ne correspond à aucun pixel réel. De même, v_count va de zéro à v_period-1, v_period étant la valeur du nombre total de lignes, y compris toutes les lignes de pixels effectivement visibles et toutes les lignes fictives correspondant à la période de transition à la fin de chaque image.

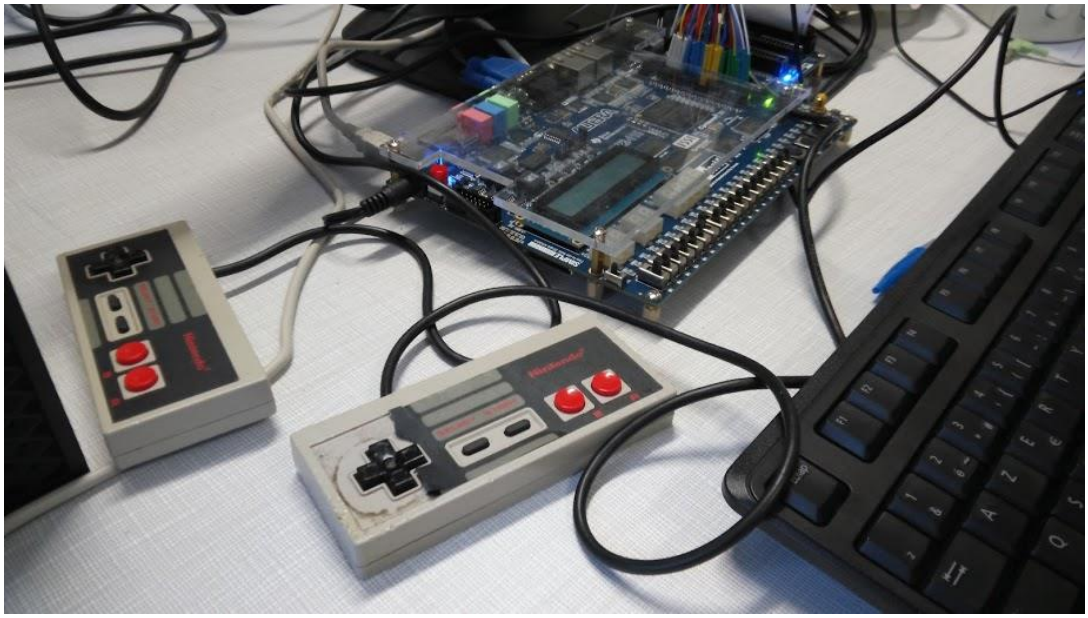
Ainsi, h_count est incrémenté chaque fois qu'un pixel est écrit sur l'écran (ou pendant les intervalles de temps correspondant aux "pixels" non visibles), revenant à zéro à la fin de l'écriture d'une ligne entière. De la même manière, v_count est incrémenté chaque fois qu'une ligne de pixels est écrite à l'écran (ou pendant les intervalles de temps respectifs que nous pouvons considérer comme des lignes non visibles à l'écran), revenant à zéro lorsque toutes les lignes terminent d'être écrites, c'est-à-dire qu'un frame finit par être représenté à l'écran et que nous passons au frame suivant.

Cela montre qu'en ce qui concerne le moniteur VGA, le plus important dans notre logique est le suivi de h_count et de v_count et, en fonction des valeurs de ces variables, nous contrôlons l'affichage correct de chaque pixel. Le contrôle de la valeur des couleurs de chaque pixel en fonction de h_count et v_count se fait avec une logique supplémentaire détaillée dans les sections correspondant au personnage, à l'environnement et aux blocs.



Manettes Nintendo

Le personnage choisi pour notre jeu est un petit fantôme qui court et saute à travers l'écran. Les commandes de contrôle des personnages sont effectuées via un contrôleur de jeu NES, comme dans la figure ci-dessous.



Pour pouvoir le contrôler de cette manière, nous utilisons une instance du composant NES_joystick. Ce composant transmet les informations sur les boutons appuyés à travers le registre Data_reg de type std_logic_vector de 8 bits, correspondant aux 8 touches de la manette NES. Chaque bit de ce registre est associé à l'un des boutons et, si un bit est égal à zéro, cela signifie que le bouton correspondant est enfoncé, sinon le bouton correspondant est relâché.

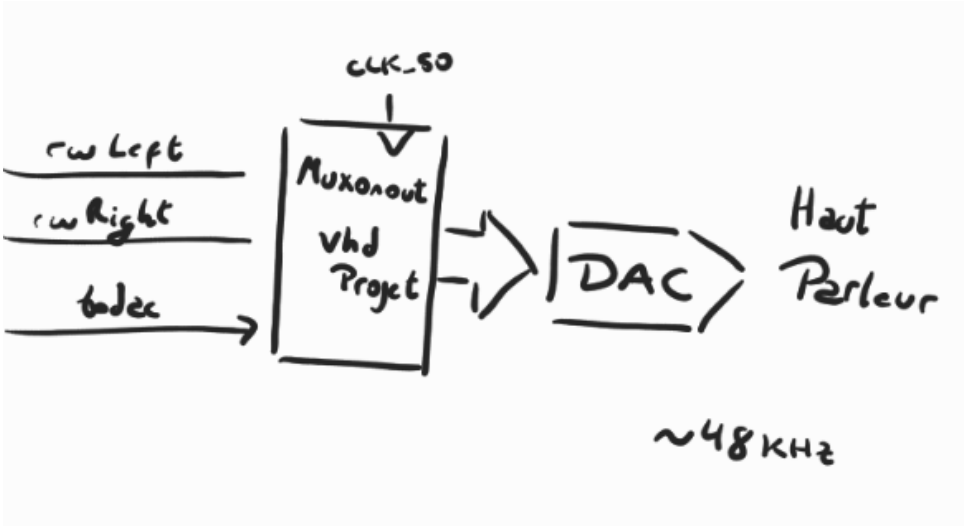
En plus de ce signal de sortie de ce composant, nous avons le signal ready, de type std_logic, qui indique si les données sont prêtes à être lues. Nous avons également trois autres signaux de sortie de ce composant, Data_from_NS, qui fournit une ligne de données à partir du joystick, PS_to_NS, qui détermine si la transmission de données sera parallèle ou série, et CLK_to_NS, qui est un clock donnée à une manette de jeu Nintendo. Nous n'avons pas besoin d'utiliser les trois derniers dans notre module VGA principal. Évidemment il existe toujours un signal d'entrée correspondant au clock de 50 MHz fournie à la manette.

Haut-Parleurs

Sur la carte reprogrammable on dispose d'un convertisseur analogique/digital et l'inverse, dans notre projet nous nous servirons du dernier. Pour utiliser cet élément, nous avons le projet déjà développé muxonout.vhl dans lequel se donnent les variables clock, todac, wrLeft, wrRigth dans le côté de l'utilisateur.

Tout d'abord, nous verrons le fonctionnement des haut-parleurs et ses relations avec les variables ci-dessus définies. L'horloge d'une enceinte traditionnelle a une fréquence d'échantillonnage approximée de 48 kHz en tant que celle du FPGA normalement correspond à 50 MHz, aussi les signaux qu'elle doit recevoir doivent être des ondes centrées en zéro et de plus elles ont comme récepteur soit le côté gauche de l'enceinte ou soit le côté droit.

La bibliothèque muxontout nous permets d'implémenter ces conditionnes dans notre code en faisant une coordination entre les deux horloges (avec un relation 1:1024) et il nous donne la sortie todac (de 16 bits) qui sera envoyé vers le côté droit quand wrRight atteint le valeur 1 et le côté gauche quand wrLeft l'atteint.



Mémoire ROM et Flash

Dans une FPGA nous trouvons différents lieux pour sauvegarder de l'information, parmi eux nous soulignons la mémoire ROM et la mémoire Flash. La première corresponde à une mémoire de lecture seulement (Read Only Memory) ce que permets de stocker des donnés en absence de courant électrique, en tant que la deuxième est une mémoire réinscriptible non volatile de 8 Mo.

Pour accéder à la mémoire ROM, il suffit de créer une bloque logique dont l'entrée est l'adresse associe aux éléments et la sortie est l'élément lié à cette adresse. Pour autant, après avoir défini cette liste, elle ne serait plus modifiable. En échange, la mémoire Flash est plus difficile à modifier et pour la lire nous nous servons des variables FL_ADDRESS, FL_DQ (Pour habiliter la lecture les variables FL_CE_N et FL_OE_N doivent être 1).

Mise en œuvre

Dans cette partie nous montrerons la mise en œuvre de l'algorithme et sa représentation par blocs logiques pour bien comprendre son fonctionnement. Pour cela, nous séparerons le jeu en trois parties, en premier la physique du jeu, en suite le contrôle du fantôme, et finalement l'ennemie et la musique.

Affichage du jeu

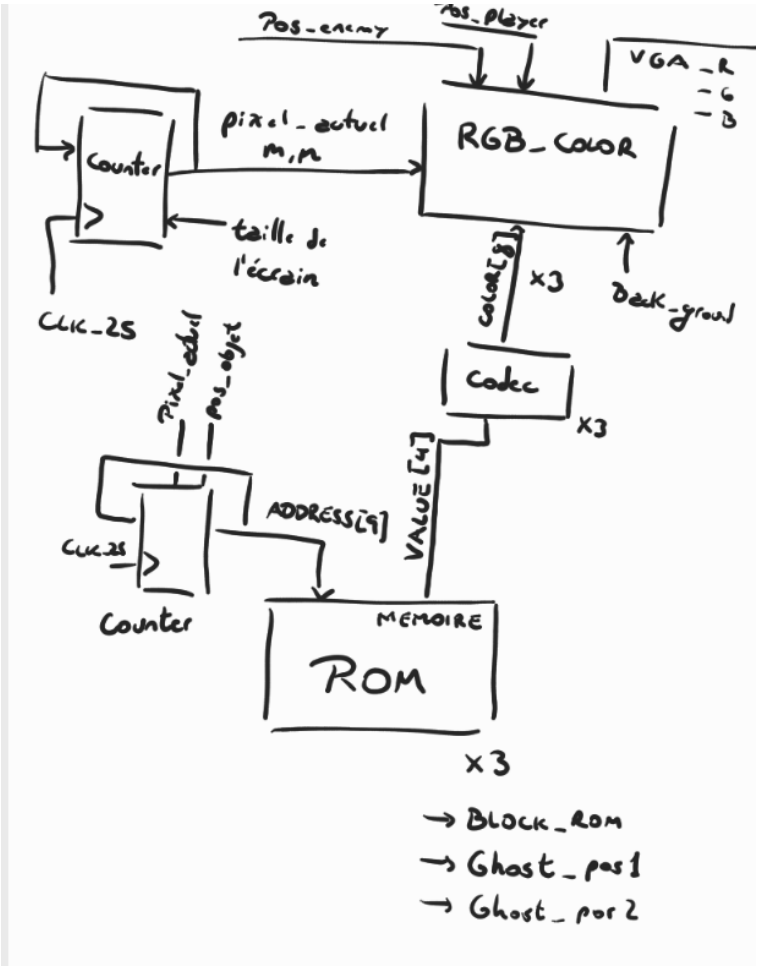
Les informations relatives à la position sur l'écran de chaque plate-forme sont fournies au module principal par un simple composant auxiliaire appelé Environnement. Ce composant transmet quatre arrays d'entiers (avec cinq entiers dans chaque array) au module VGA principal: pos_x, pos_y, size_x et size_y. Les deux premiers sont des arrays contenant les positions sur l'écran du centre de chaque plate-forme. Chaque valeur de pos_x est la position du pixel dans lequel sera le centre, sur l'axe des x, de chaque plate-forme. De manière analogue à pos_y, chaque valeur correspond au numéro de la ligne dans laquelle le centre est, sur l'axe des ordonnées, de chaque plate-forme.

Les arrays size_x et size_y contiennent la taille, en x et y, de chaque plate-forme, c'est-à-dire la longueur et la hauteur de chaque plate-forme. Nous avons choisi de transmettre ces valeurs également sous forme de arrays pour le cas où nous aurions décidé d'utiliser des plates-formes de tailles différentes (en fait, la plate-forme centrale est plus grande que les quatre autres).

L'affichage de chaque bloc est effectué en utilisant les positions et les tailles de chaque plate-forme, en prenant les informations des pixels à travers la ROM de la carte FPGA. Ainsi, dans notre module principal, nous incluons une logique qui vérifie si le pixel actuel se trouve dans l'une des régions indiquées comme faisant partie de l'une des régions de la plate-forme, dont les informations proviennent du composant Environnement. Ainsi, si le pixel actuel se trouve dans une région de la plate-forme, nous prenons les informations de pixel correspondantes dans la mémoire ROM, qui contient les données de couleur des blocs.

En plus de ces cinq plateformes, le jeu a une partie de terrain avec un comportement légèrement différent. Le personnage peut également courir et sauter dessus, mais peut également descendre et y entrer. L'affichage de cette région est très simple. Lorsque la valeur de v_count (qui compte le numéro de la ligne en cours) est inférieure à une certaine valeur, nous faisons en sorte que les signaux VGA_R, VGA_G et VGA_B prennent les valeurs correspondant à la couleur bleue ; lorsque v_count dépasse cette valeur, VGA_R, VGA_G et VGA_B se voient attribuer les valeurs relatives à la couleur brune.

Plus rigoureusement, nous avons développé un bloque logique qui prends comme entrées les positions du fantôme, l'ennemie et l'environnement, pour ensuite choisir selon la position actuelle celle qui corresponde entre eux pour être affiché. Dès qu'il le trouve, il cherche sa description dans la mémoire ROM et le pixel correspondante comme il se montre dans le diagramme de blocs ci-dessous.



La mémoire de lecture seulement a une capacité limitée, c'est pourquoi les images stockées sont en noir et blanc de 4-bit par pixel, cela donne une taille de 1,6 Kbits pour chaque image. Et pour donner le couleur aux images nous faisons une relation de proportion entre les variables VGA_R, VGA_G et VGA_B. Aussi, pour créer une animation nous avons utilisé deux images du fantôme dont la différence se fait aux niveaux des pies qu'ils sont légèrement décalés dans la deuxième image et on alterne les images pour émuler du mouvement.

Contrôle du fantôme et l'ennemie

Personnage Principal

Afin de contrôler le personnage principal, le fantôme, nous créons les variables px, py et jump. px et py gardent les valeurs relatives à la position du personnage, alors que jump est utilisé pour contrôler le saut du personnage afin qu'il ne puisse sauter que quand il est penché sur une plate-forme ou sur le sol, ne pouvant pas sauter à nouveau s'il est au milieu de l'air. Afin de transmettre les commandes de la manette NES au personnage, nous créons une boucle avec une variable appelée Cnt, qui est incrémentée à chaque signal du clock. Cette boucle est utilisée pour contrôler la vitesse à laquelle notre personnage peut se déplacer sur l'écran. Cette vitesse est limitée par le nombre donnée à une valeur appelée speed, définie dans le corps de

GENERIC. Chaque fois que Cnt atteint la valeur de speed, Cnt revient à zéro et, à ce moment, nous permettons au personnage de se déplacer sur l'écran. Si un tel contrôle n'existait pas, le personnage pourrait se déplacer avec chaque impulsion du clock, ce qui signifierait qu'il serait trop rapide pour un être humain le contrôler. Nous avons donc limité les moments où le personnage peut se déplacer à cet instant précis, où Cnt est égal à speed.

Lorsque Cnt atteint la valeur de speed, nous autorisons le personnage à se déplacer, c'est-à-dire que nous mettons à jour les valeurs de px, py et jump en fonction des valeurs des registres de la manette NES, qui proviennent du composant NES_joystick et sont stockées dans le std_logic_vector REG_JOYSTICK_1, contenant les informations sur les boutons sur lesquels on appuie (sachant qu'un bit égal à zéro signifie que le bouton correspondant est enfoncé).

Si la flèche droite est enfoncée et que la flèche gauche est relâchée, px est incrémenté d'une unité, l'inverse pour la flèche gauche. Si vous appuyez simultanément sur les deux flèches, px ne change pas de valeur.

Pour le contrôle de saut, nous utilisons la variable jump, qui varie entre zéro et maxJump, qui est une valeur définie dans GENERIC. maxJump définit la valeur de la hauteur, en nombre de pixels, que le personnage peut sauter. Si jump est égal à maxJump, cela signifie que le personnage peut sauter, à partir de là, nous commençons à diminuer jump et py. Les axes de référence que nous avons utilisé étaient l'axe des abscisses allant de gauche à droite et l'axe des ordonnées allant de haut en bas. Donc, quand py est décrémenté, cela signifie que le personnage monte pendant un saut. Dans le même temps, nous décrémentons la variable jump. Lorsque jump atteint zéro, le personnage cesse de monter et passe à descendre, ce qui signifie que nous incrémentons la valeur de py au lieu de la diminuer, jusqu'à ce que le fantôme touche une plate-forme. Lorsque le personnage s'appuie sur une plate-forme, jump revient à la valeur de jumpMax, ce qui signifie que le personnage est à nouveau autorisé à sauter. Nous vérifions si le personnage s'appuie contre l'une des plates-formes par des structures conditionnelles avec des IFs qui vérifient la distance du personnage à chacune des cinq plates-formes, en tenant compte de la position et de la taille de chaque plate-forme, dont les informations proviennent du composant Environnement.

En plus des cinq plates-formes, le personnage peut marcher sur le sol brun situé dans la partie basse de l'écran. Si le personnage touche cette partie, nous pouvons permettre à py d'augmenter si la flèche vers le bas est enfoncée. Cela signifie que le fantôme peut entrer dans cette région. À l'intérieur de cette région, le personnage peut se déplacer librement, son contrôle étant assuré par les flèches de la manette NES et en l'absence de l'effet de gravité, ce qui ferait le personnage tomber. Encore une fois, notons que nous voyons si le personnage touche cette région par sa position (définie par px et py) et par sa taille, définie par P_size, déclarée dans GENERIC.

L'ennemie

Pour donner un sens au jeu, nous avons créé un ennemi qui est une version de fantôme qui poursuit notre personnage principal. Si cet ennemi atteint et touche le personnage principal, cela signifie que vous avez perdu et que le jeu ira recommencer, le personnage principal et l'ennemi revenant à leurs positions initiales.

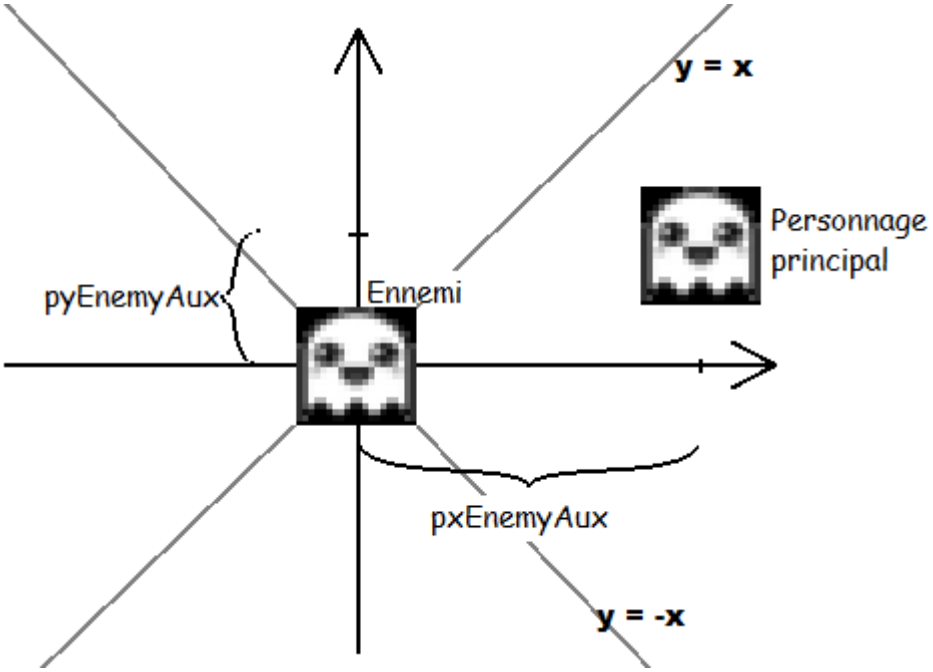
L'image utilisée pour cet ennemi est la même que celle utilisée pour le personnage principal. Cependant, nous avons légèrement modifié sa couleur en ajoutant des valeurs de offset aux valeurs transmises aux signaux de sortie VGA_R, VGA_G et VGA_B.

La logique de déplacement de l'ennemi est contrôlée par des variables ayant des fonctions similaires à celles des variables correspondantes du personnage principal. Ainsi, pxEnemy et pyEnemy sont utilisés pour stocker les valeurs de position de l'ennemi à l'écran, tout comme CntEnemy et speedEnemy sont utilisés pour contrôler la vitesse de l'ennemi et les moments où il est autorisé à se déplacer, en utilisant pratiquement les mêmes structures et boucles que celles utilisés pour contrôler le personnage principal.

La seule différence dans le contrôle de l'ennemi réside dans le contrôle de ses mouvements. Alors que les mouvements du personnage principal sont contrôlés par le joueur via la manette NES, le contrôle de l'ennemi est fait par les variables pxEnemy et pyEnemy, ayant une logique spécifique pour l'ennemi. Notons tout d'abord que l'ennemi n'est pas soumis à l'effet de gravité utilisé par le personnage principal, tout comme nous n'utilisons des informations indiquant s'il se trouve appuyé sur ou en touchant une plate-forme. L'ennemi ne fait que suivre le personnage principal.

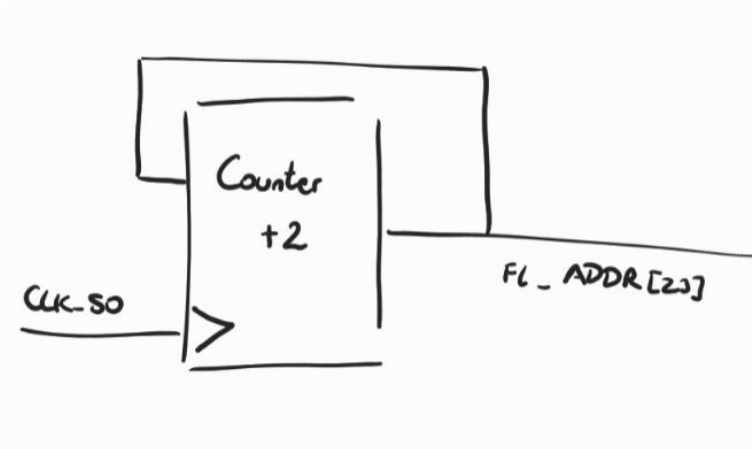
Ce contrôle pour suivre le personnage principal se fait à travers des structures conditionnelles avec des IFs, vérifiant dans quelle direction se situe le personnage principal par rapport à la position de l'ennemi. Ainsi, nous comparons pxEnemyAux et pyEnemyAux, qui sont des variables représentant respectivement la différence entre pxEnemy et px et la différence entre pyEnemy et py. En traçant un système d'axes partant de la position de l'ennemi et en comparant les valeurs de pxEnemyAux et de pyEnemyAux, nous pouvons savoir dans quel quadrant de ce système d'axes se trouve le personnage principal. En utilisant ces informations dans notre structure conditionnelle avec des IFs, nous déterminons s'il convient de diminuer ou d'augmenter les valeurs de pxEnemy et pyEnemy, qui contrôlent la position de l'ennemi à l'écran. Ainsi, l'ennemi peut monter, descendre, aller à gauche, à droit ou en diagonale, en suivant la direction du personnage principal.

Comme illustré dans la figure ci-dessous, si pxEnemyAux est égal à pyEnemyAux, cela signifie que l'ennemi et le personnage principal sont sur la même diagonale y = x. Si pxEnemyAux est égal à -pyEnemyAux, l'ennemi et le personnage principal sont sur la même diagonale y = -x. Pour savoir quelle direction l'ennemi doit suivre, il suffit de vérifier la valeur du signal de pxEnemyAux et de pyEnemyAux. Hors ces cas simples, il suffit de savoir qui est le plus grand parmi pxEnemyAux et pyEnemyAux pour déterminer dans quelle des quatre régions du graphe ci-dessous se trouve le personnage principal, ce qui déterminera si pxEnemy et pyEnemy doivent être incrémentés, décrémentés ou aucun des deux.

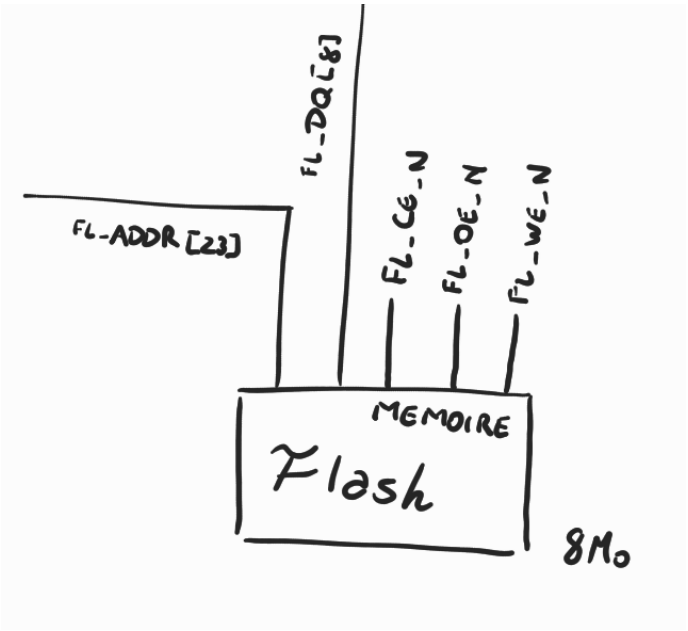


Musique du jeu

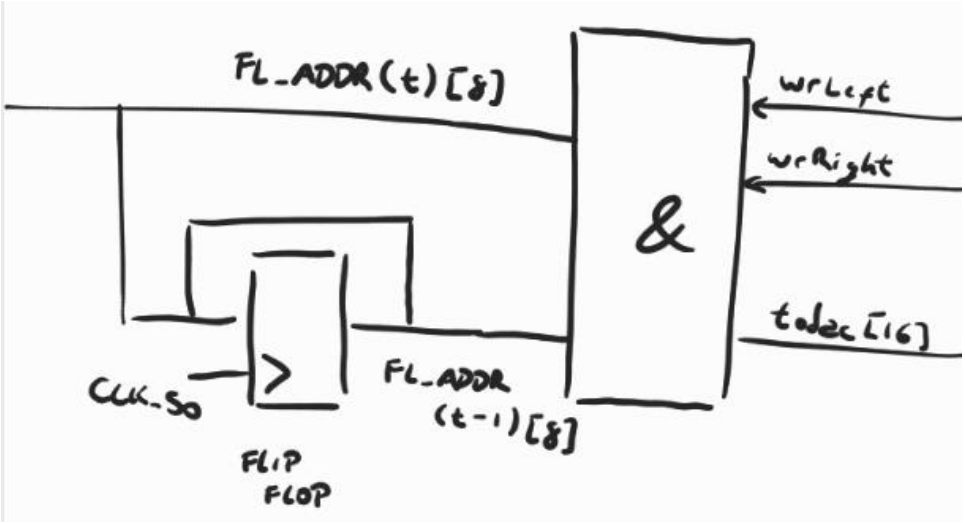
Pour en finir, nous avons créé un deuxième process dont la fonction était de lire la mémoire flash où nous avons mis la musique pour le jeu. Cette mémoire est conçue par octets (8-bits) et la sortie du son est de 16-bits, alors pour retrouver la chanson originale nous prenons la valeur de deux adresses consécutives et ainsi de suite. Pour cela, nous avons un compteur qui monte de deux en deux dés 0 jusqu'à 2^23-1 en binaire de 23-bits.



On donne ensuite l'adresse à la mémoire flash pour obtenir l'octet correspondante (avec les conditionnes pour la lecture) comme il se montre dans la figure ci-dessous.



Après, les données obtenues de la mémoire flash sont unies pour former la sortie du son de 16-bits (todac), quand wrLeft est 1 il est envoyé sur l'enceinte gauche et quand wrRight est 1 il est envoyé sur le droit.



Conclusion

Avec ce projet nous avons eu la possibilité d'expérimenter avec une carte FPGA et avec son langage de développement VHDL avec lequel nous avons appris la conception de circuit logiques et compris le fonctionnement interne de la carte, pour ensuite développer un projet d'électronique.

Nous avons travaillé avec plusieurs périphériques, ainsi que logique interne du jeu principal. À partir de cet élément, nous avons établi la connexion avec le moniteur, les manettes, les haut-parleurs et contrôlons les signaux envoyés tout en utilisant deux procès avec un horloge de 25MHz pour l'écran et de 50MHz pour les enceintes.

Ce projet nous a permis de jeter un œil sur la micro technologie et comment s'en servir pour développer un projet, aussi nous avons appris à chercher les descriptions des composantes sur les sites d'internet et ses data sheets. Ce que nous permets de voir ses avantages et désavantages dans notre modèle.