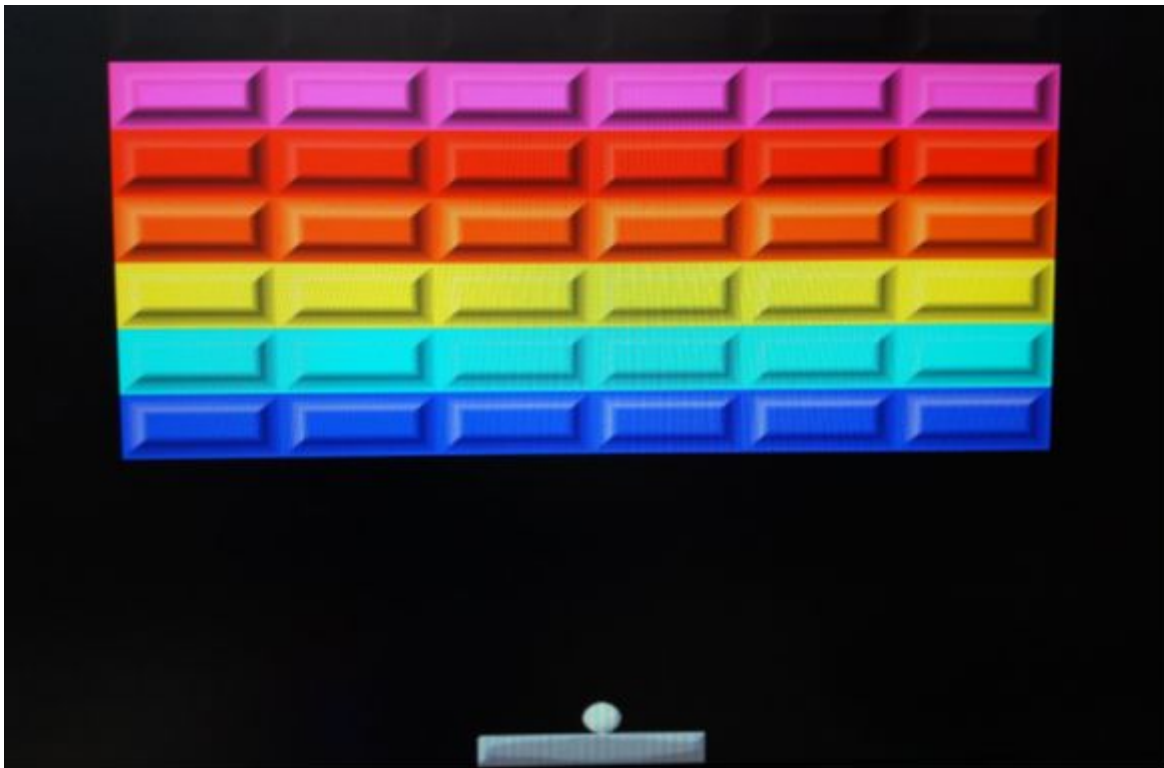


# Compte-rendu de TL FPGA

## Jeu du Casse-Briques



# Sommaire

## **Introduction**

## **I - Briques et plateforme**

- 1) Mémoire des briques
- 2) Gestion des niveaux
- 3) Déplacement de la plateforme

## **II - Dynamique de la balle (NIOS)**

## **III - Affichage**

- 1) Images
- 2) Implémentation
- 3) Fonctionnement

## **IV - Compléments**

- 1) Score
- 2) Vie
- 3) Gestion de la pause
- 4) Son

## **V - Ressources**

## **Conclusion**

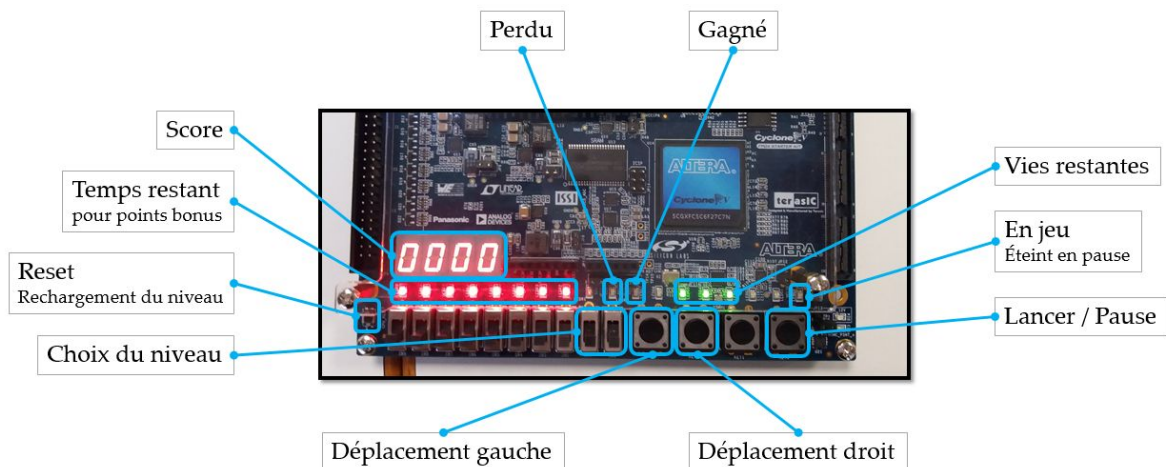
# Introduction

Dans le cadre du TL FPGA, nous avons implémenté le jeu du Casse-Briques sur la carte de développement du Cyclone V. L'objectif principal de ce projet est d'apprendre à manipuler l'environnement Quartus en développant les différentes fonctions nécessaires en VHDL et en C via l'implémentation d'un processeur de type NIOS sur le FPGA.

L'interface de jeu se fait sur un écran connecté via le port HDMI de la carte de développement. Un de nos objectifs était d'utiliser un maximum de ressources différentes à disposition afin d'interagir avec l'utilisateur.

Le jeu du Casse-Brique est composé d'une plateforme ou raquette que le joueur contrôle horizontalement via deux boutons (déplacement gauche et droit). Il doit la placer de façon à intercepter et diriger une balle vers les briques qu'il cherche à casser pour gagner la partie. Chaque brique nécessite d'être touchée entre 1 et 7 fois avant de disparaître. La résistance de la brique est indiquée par sa couleur.

Dans la version finale de notre projet, le joueur dispose de 3 vies. Il peut donc perdre deux fois lors du même niveau avant d'avoir définitivement perdu et de devoir recommencer le niveau. Le score du joueur est affiché durant la partie. Ce score dépend du nombre de briques cassées, ainsi qu'un bonus dépendant du temps que le joueur a eu besoin pour compléter un niveau, s'il le réussit.



*Descriptif de l'interface avec le joueur*

# I - Briques et plateforme

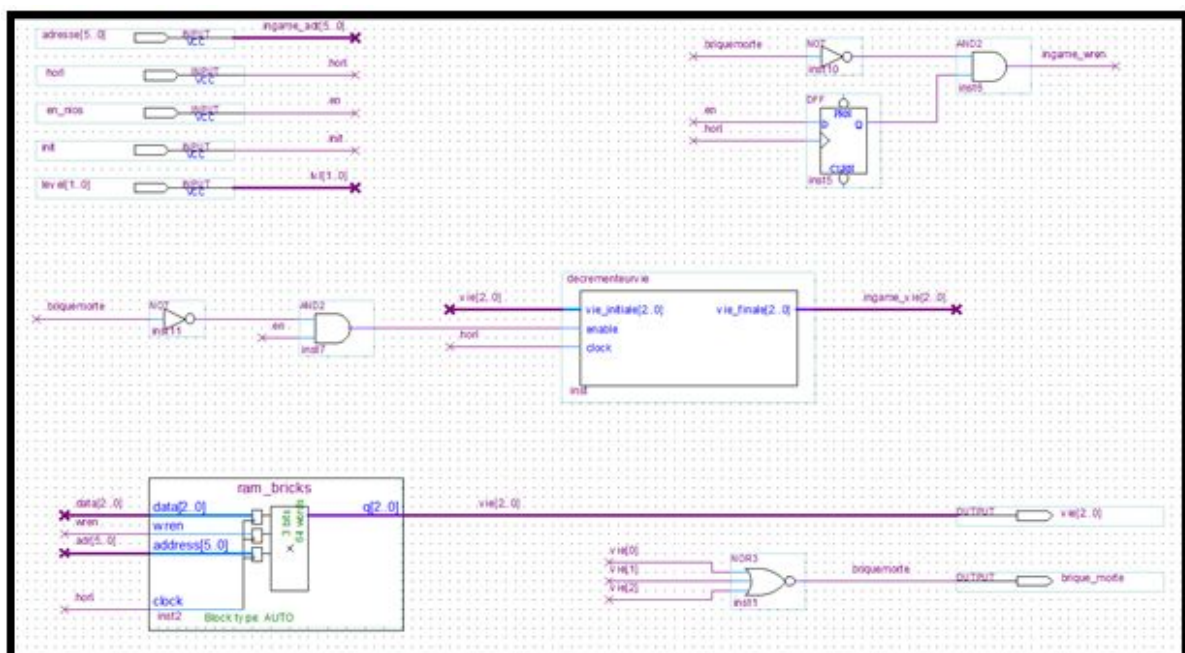
Nous allons détailler ici la façon dont nous avons choisi de mémoriser la position des briques et leur résistance (quantité de vie de la brique), l'implémentation de la gestion de plusieurs architectures de briques définissant chacun des quatres niveaux et le fonctionnement du bloc gérant le déplacement de la plateforme.

## 1) Mémoire des briques

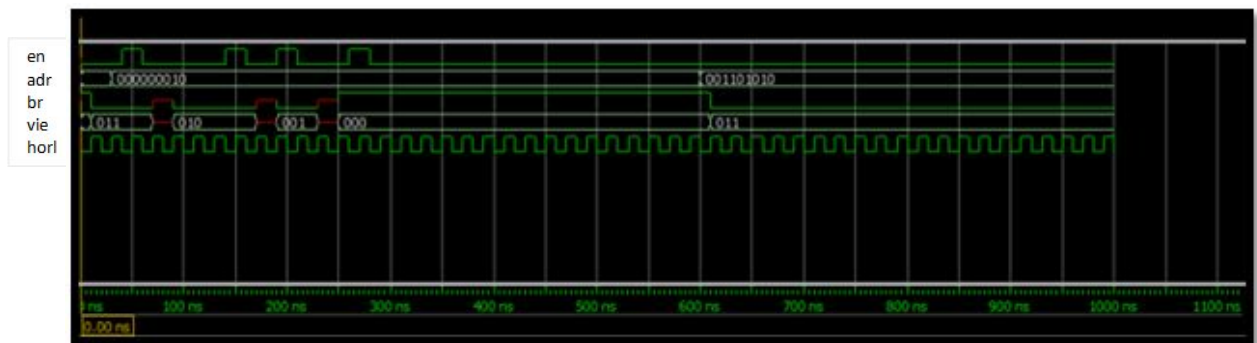


Ce bloc, synchrone, a pour fonction de stocker la vie des briques et décrémenter la vie d'une brique lors d'une collision sur ordre du Nios (**en\_nios=1**).

Il permet également de stocker les différents niveaux (**level[1..0]**). Ses sorties sont **vie[2..0]**, qui affiche la vie de la brique située à **adresse[5..0]**, et **brique\_morte** permettant de savoir si la brique est morte ou pas (utile pour la gestion des collisions).



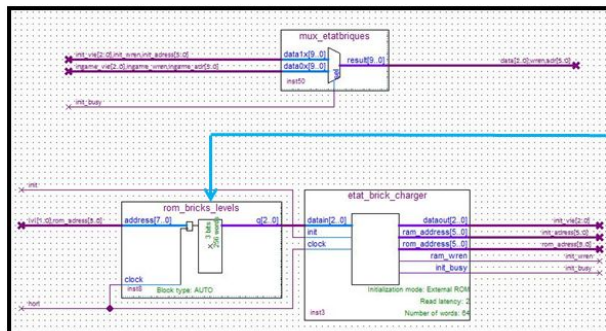
Son fonctionnement peut être résumé par la simulation suivante réalisée avec ModelSim :



## 2) Gestion des niveaux

La gestion de plusieurs niveaux se fait en complétant l'architecture de mémoire des briques présentée précédemment. En effet, il suffit de charger la RAM EtatBriques avec la disposition initiale des briques et leur résistance correspondant au niveau désiré.

Pour cela, nous utilisons une ROM de 256 mots de 3 bits, c'est-à-dire quatre niveaux de 64 briques. On initialise ensuite la RAM EtatBriques lors du chargement de niveau par la plage correspondante dans la ROM BricksLevels.



Utilisé pour  
détecter la  
victoire

1	DEPTH = 256;	-- The size of memory in words
2	WIDTH = 3;	-- The size of data in bits
3	ADDRESS_RADIX = UNS;	-- The radix for address values
4	DATA_RADIX = BIN;	-- The radix for data values
5		
6	CONTENT	-- start of (address : data pairs)
7	BEGIN	
8		
9	[0..255] : 000;	-- INIT to 0
10		
11		
12	-- LVL 1 [0..63]	
13	[49..54] : 001;	
14	[41..46] : 010;	
15	[33..38] : 011;	
16	[25..30] : 100;	
17	[17..22] : 101;	
18	[9..14] : 110;	
19	[1..6] : 111;	
20		
21		
22	-- LVL 2 [64..127]	
23	[66..69] : 001;	
24		
25		
26	-- LVL 3 [128..191]	
27	[128..135] : 111;	
28		
29		
30	-- LVL 4 [192..255]	
31	192 : 101;	
32	194 : 101;	
33	196 : 101;	
34	198 : 101;	
35		
36		
37		
38		
39		
40		
41		
42		
43		
44		
45		
46		
47		
48		
49		
50		
51		
52		
53		
54		
55		
56		
57		
58		
59		
60		
61		
62		
63		
64		
65		
66		
67		
68		
69		
70		
71		
72		
73		
74		
75		
76		
77		
78		
79		
80		
81		
82		
83		
84		
85		
86		
87		
88		
89		
90		
91		
92		
93		
94		
95		
96		
97		
98		
99		
100		
101		
102		
103		
104		
105		
106		
107		
108		
109		
110		
111		
112		
113		
114		
115		
116		
117		
118		
119		
120		
121		
122		
123		
124		
125		
126		
127		
128		
129		
130		
131		
132		
133		
134		
135		
136		
137		
138		
139		
140		
141		
142		
143		
144		
145		
146		
147		
148		
149		
150		
151		
152		
153		
154		
155		
156		
157		
158		
159		
160		
161		
162		
163		
164		
165		
166		
167		
168		
169		
170		
171		
172		
173		
174		
175		
176		
177		
178		
179		
180		
181		
182		
183		
184		
185		
186		
187		
188		
189		
190		
191		
192		
193		
194		
195		
196		
197		
198		
199		
200		
201		
202		
203		
204		
205		
206		
207		
208		
209		
210		
211		
212		
213		
214		
215		
216		
217		
218		
219		
220		
221		
222		
223		
224		
225		
226		
227		
228		
229		
230		
231		
232		
233		
234		
235		
236		
237		
238		
239		
240		
241		
242		
243		
244		
245		
246		
247		
248		
249		
250		
251		
252		
253		
254		
255		
256		

Disposition des briques / Nombre de briques par niveau

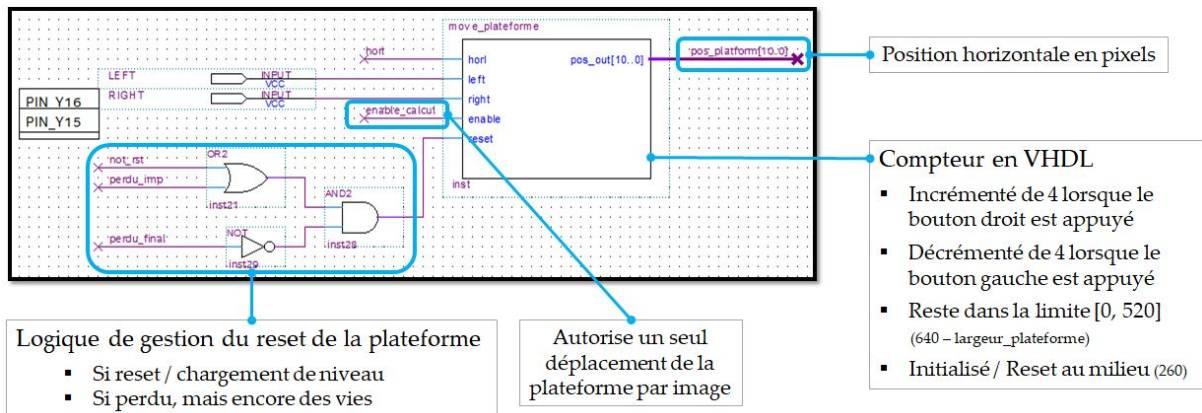
Un multiplexeur permet de choisir entre le mode d'initialisation et le mode d'utilisation de la RAM.

Les niveaux sont décrits dans un fichier .mif afin d'initialiser la ROM BricksLevels.

On utilise également un second fichier de configuration contenant le nombre de briques de chaque niveau. Il permet de détecter lorsque toutes les briques sont détruites, et donc la victoire du joueur.

### 3) Déplacement de la plateforme

Le joueur peut déplacer la plateforme vers la gauche ou vers la droite grâce aux deux boutons associés. Cette information est traitée par le bloc MovePlatform qui calcule la position horizontale de la plateforme. Ce bloc est un simple compteur, incrémenté ou décrétementé selon la direction de déplacement voulu.



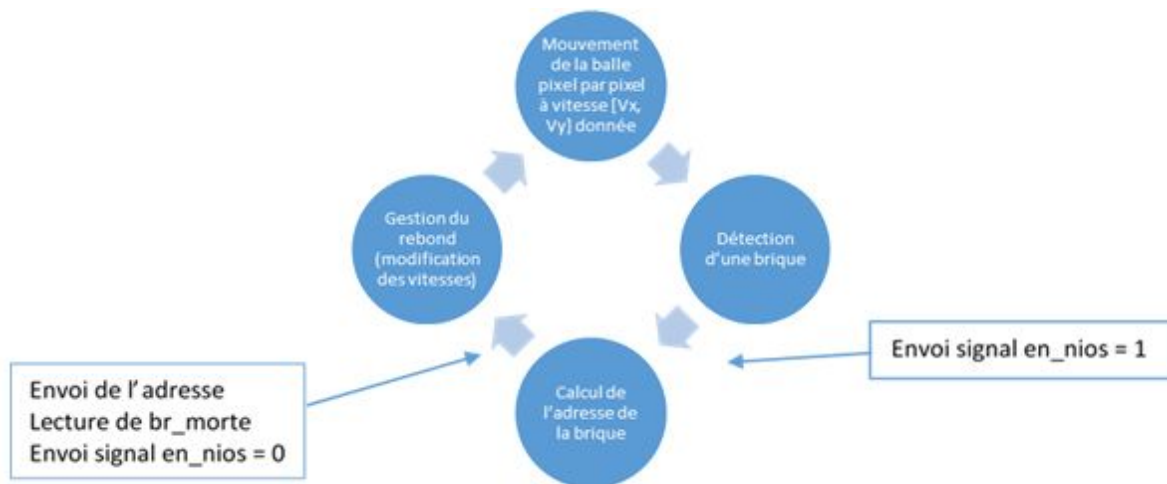
La position de la plateforme est réinitialisé au centre de l'écran lorsqu'un niveau est chargé ou lorsque le joueur perd mais continue à jouer car il possède encore des vies.



## II - Dynamique de la balle (Nios)

La gestion du mouvement de la balle sur l'écran est réalisée par un processeur Nios, qui a également pour rôle de gérer les différentes collisions et l'interruption du jeu en cas de fin de partie.

Les collisions peuvent être de trois sortes : avec les bords de l'écran, avec la plateforme, ou avec une brique. Le cycle suivant résume le fonctionnement du Nios dans le cas le plus fréquent, une collision avec une brique :

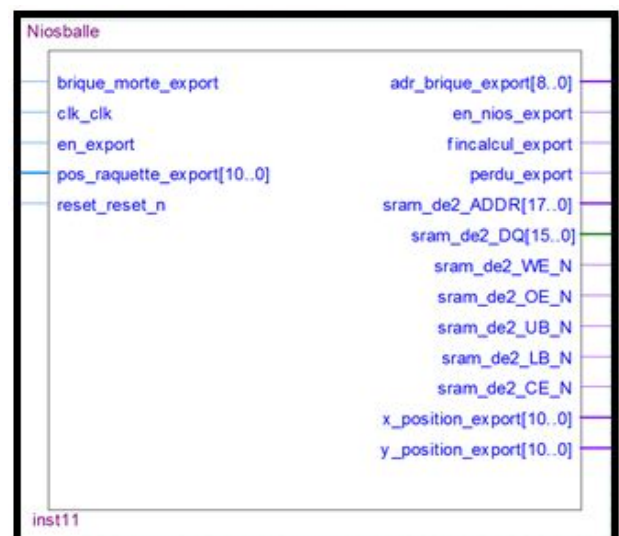


Les entrées du Nios sont :

- **brique\_morte** : indique si la dernière brique dont l'adresse a été envoyée est morte ou non, et donc si la balle doit rebondir ou pas
- **Clk** : Horloge
- **en** : permet la gestion des calculs du Nios (utile pour la synchronisation)
- **pos\_raquette** : position de la plateforme gérée par le joueur
- **reset** : permet de réinitialiser la position et la vitesse de balle en début de partie

Les sorties principales du Nios sont :

- **adr\_brique** : adresse de la brique avec laquelle la balle entre en collision, envoyée au bloc Etatbriques
- **en\_nios** : signal passant à l'état haut lorsque la balle entre dans l'emplacement potentiel d'une brique, envoyé au bloc Etatbriques
- **fincalcul** : gestion des calculs du Nios (utile à la synchronisation)
- **perdu** : indique que la balle a atteint le bord bas de l'écran
- **x\_position** et **y\_position** : position de la balle, envoyée à l'afficheur



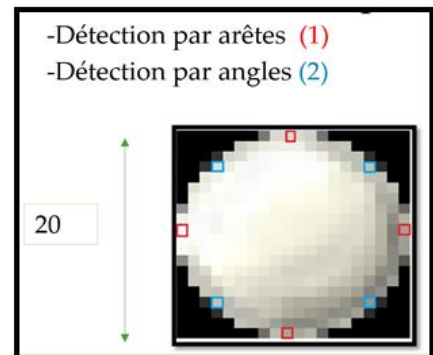
La gestion du mouvement de la balle, des différents types de collision et de la perte de la partie en cours s'est faite en C à l'aide d'Eclipse et de l'outil de construction associé sur Quartus.

Après instanciation des positions et des vitesses initiales, une boucle infinie réalise une évolution de la position de la balle par déplacements élémentaires, permettant ainsi de gérer les différentes collisions et la vitesse. La balle, qui est un carré de dimension 20x20 pixels, est caractérisée par son coin haut gauche.

Le rebond sur la plateforme est affecté d'un coefficient sur la vitesse latérale qui dépend de l'évolution de la position de la raquette entre le moment où la balle s'apprête à toucher la plateforme et le moment effectif de la collision, améliorant la jouabilité. Ainsi, si le joueur est en train de déplacer la plateforme vers la gauche lors de la collision, la balle sera dirigée vers la gauche.

Le rebond avec les bords de l'écran est géré simplement avec les positions des arêtes, les dimensions de l'écran étant connues.

Le rebond sur les briques est géré par un double système d'arêtes et d'angles. En effet, une brique étant caractérisée par les numéros de lignes et de colonnes qui la délimitent, une simple détection par arête pose problème lorsque la collision a lieu en diagonale d'une brique (les briques détectées sont alors les briques adjacentes à la brique, et la balle peut traverser la brique).



L'ajout de la détection par angles résout le problème des briques « traversées » par les balles, même si elle mène parfois à un autre cas limite qui est la perte de plusieurs points de vie d'une brique, liée à plusieurs détections de la balle, qui n'a pas le temps de sortir de la brique.

Lorsqu'une brique est détectée, le Nios envoie **en\_nios=1** à Etatbriques, calcule l'adresse de la brique puis l'envoie. Il lit ensuite l'état de la brique (morte ou non), de manière à pouvoir modifier ou non le déplacement de la balle. La procédure de détection d'une brique est donc la suivante :

```
// Retourne un booléen indiquant si la brique présente au pixel (px_x, px_y) est morte
char askBrickMemory(int px_x, int px_y)
{
    IOWR(EN_NIOS_BASE, 0, 1);
    IOWR(ADR_BRIQUE_BASE, 0, (px_x / 80) + 8 * (px_y / 40));
    char br_morte = IORD(BRIQUE_MORTE_BASE, 0);
    IOWR(EN_NIOS_BASE, 0, 0);
    return br_morte;
}
```

La modification de la vitesse se fait ensuite de la manière suivante (exemple d'une détection par arête gauche) :










```
// Haut Gauche
else if(!askBrickMemory((int) position_x + 3, (int) position_y + 3))
{
    if(vitesse_x < 0 && vitesse_y < 0)
    {
        vitesse_x = fabs(vitesse_x);
        vitesse_y = fabs(vitesse_y);
    }
    else
    {
        vitesse_x = fabs(vitesse_x);
    }
}
```



### III - Affichage

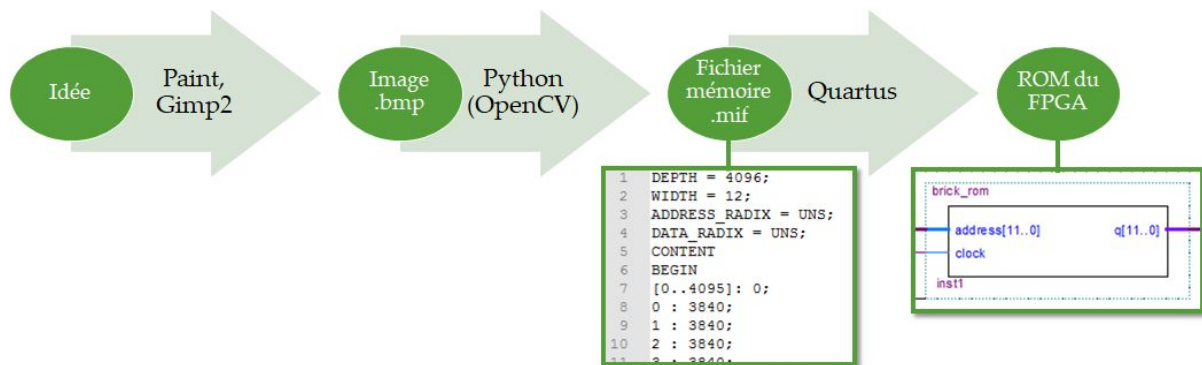
#### 1) Images

Nous utilisons ces différentes images lors de l'affichage :

Briques							Balle	Plateforme
								
1	2	3	4	5	6	7		

Nous les avons dimensionné afin qu'elle puisse s'adapter avec différentes résolutions d'affichages car cette caractéristique n'a été fixée que par la suite.

Nous les avons dessinés à l'aide de logiciels de dessin, puis nous avons implémenté un script Python, utilisant la librairie OpenCV, afin de convertir les fichiers images .bmp ou fichiers mémoires .mif qui seront chargés dans les ROMs du FPGA stockant les images.

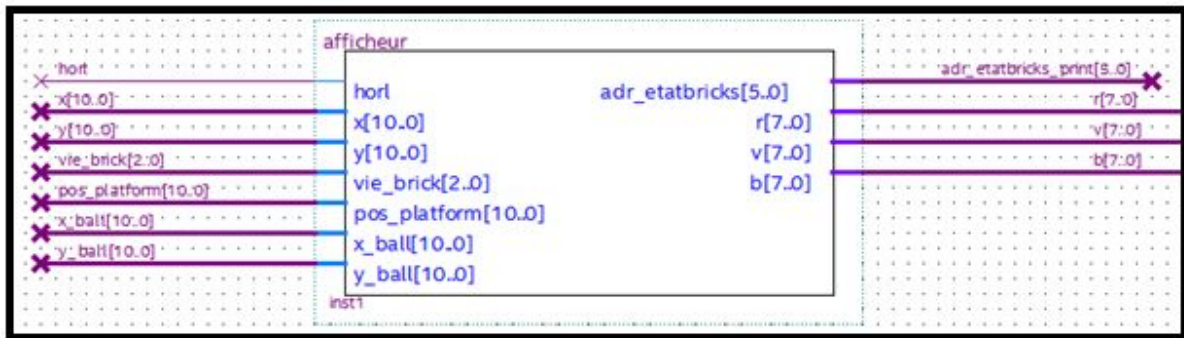


Dans un but de réduction des ressources utilisées sur le FPGA, chaque pixel des images est stocké sur 12 bits, 4 pour chaque composante RGB.

Ainsi, une brique de dimension 80x40 représente 3200 mots de 12 bits, soit 4,8ko. L'image de la plateforme de 120x20 pèse 2400 mots (3,6ko) et l'image de la balle de 20x20 pèse 400 mots (0,6ko). Malheureusement, la taille des mémoires implantables dans le FPGA doivent être des multiples de 2. Ainsi, les briques sont stockées dans des ROMs de 4096 mots par exemple.

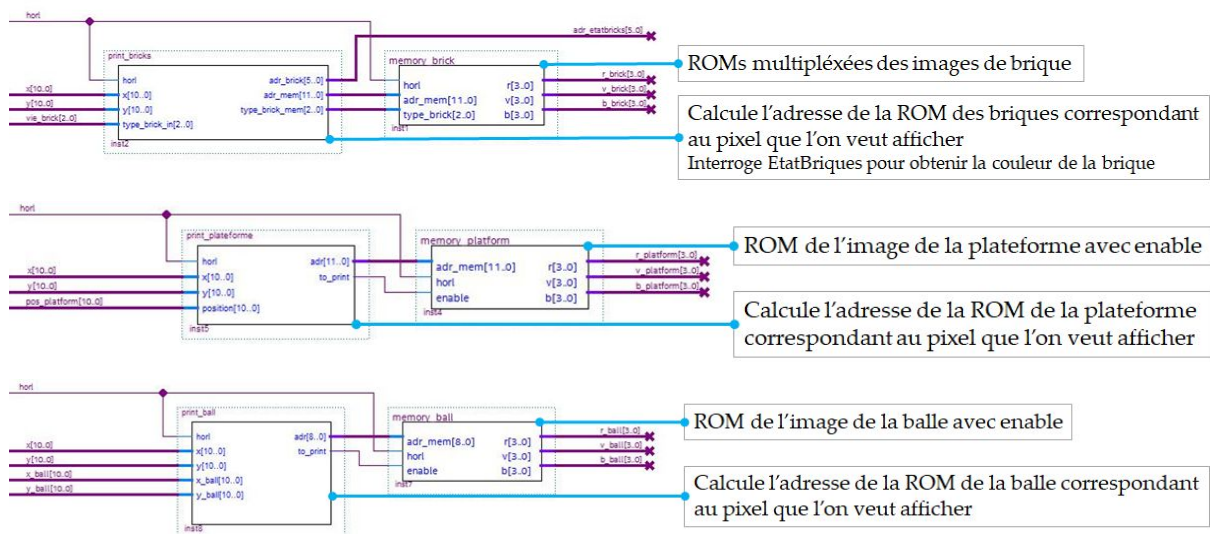
## 2) Implémentation

La gestion de l'affichage est gérée via le bloc suivant :



Ce bloc prend en entrée les coordonnées du pixel que l'on souhaite afficher et retourne les composantes RGB correspondantes. Pour cela, il demande également la position de la plateforme et de la balle et interroge la RAM EtatBriques afin de connaître l'emplacement et la couleur des briques.

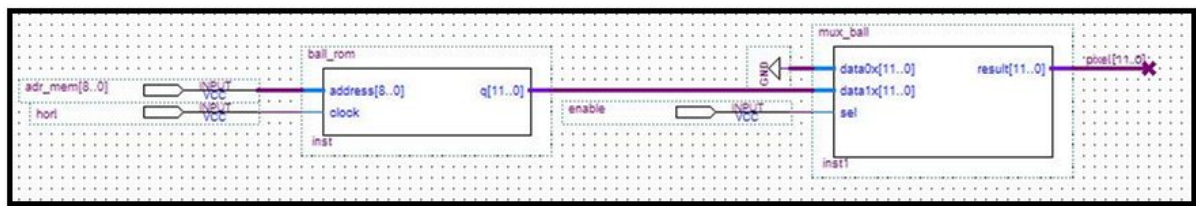
Ainsi, on a un découpage de ce bloc en trois sous-parties : une pour les briques, une pour la plateforme et une dernière pour la balle. Cependant, elles s'articulent selon une architecture similaire : un bloc contrôleur en amont et un bloc mémoire en aval. Le contrôleur détermine si l'élément doit être affiché à ces coordonnées et l'adresse en mémoire ROM du pixel de l'image de l'élément.



*Détail du bloc gérant l'affichage*

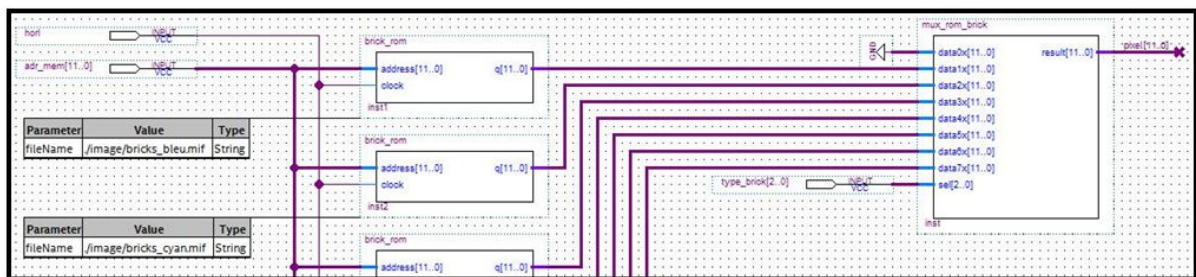
Les fonctions des contrôleurs nécessitent d'effectuer des multiplications et des divisions. C'est pour cela que la synthèse de ces blocs intègre des DSP.

Pour les mémoires de la plateforme et de la balle, si le signal *enable* est à l'état logique bas, les composantes de sortie sont forcées à 0 (noir).



Bloc mémoire de l'image de la balle

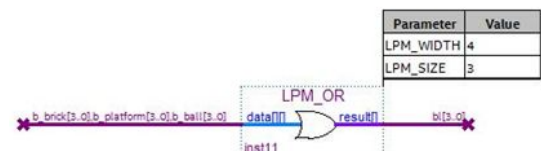
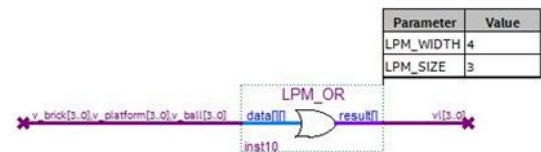
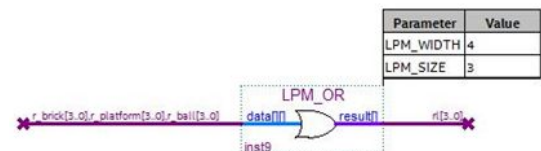
Pour la mémoire des briques, l'entrée `type_brick` commande un multiplexeur qui permet de choisir la brique adapté à la vie en mémoire. Si cette entrée est à l'état logique bas, les composantes de sortie sont alors également forcées à 0. Cela est équivalent au signal `enable` des deux autres mémoires images.



Bloc mémoire des bricks de la balle

Au final, les composantes en sortie de ces trois sous-parties sont combinés via une fonctions logique OU. En effet, étant donné que les différents éléments affichés ne se superposent pas, seul l'élément à afficher présente des composantes des non-nuls.

Pour rappel, les composantes des images sont stockées sur 4 bits. Ainsi, les 4 autres bits (poids faible) sont mis à zéro.



### 3) Fonctionnement

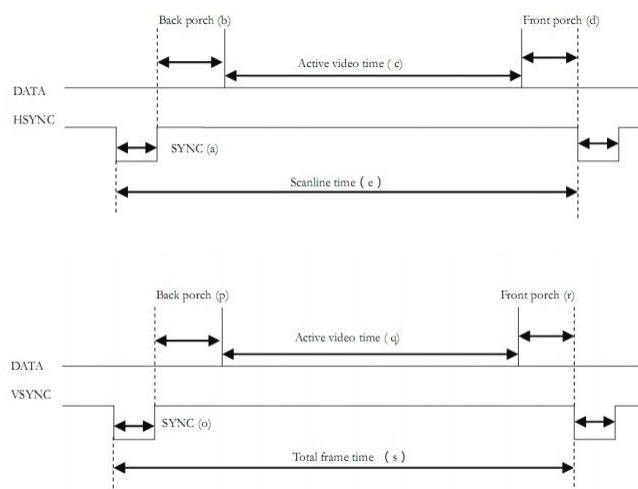
Le mode de balayage d'affichage peut être résumé en deux cas, l'un est le balayage progressif et l'autre est le balayage entrelacé. Le balayage progressif part du coin supérieur gauche de l'écran, de gauche à droite, pixel par pixel. À la fin d'un balayage d'une ligne, l'affichage retourne à la ligne grâce au signal de synchronisation horizontale. Lorsque toutes les lignes sont balayées, une trame est formée, le signal de synchronisation verticale est utilisé et l'affichage reprend en haut à gauche de l'écran. Ensuite, c'est le démarrage de la prochaine trame. Le balayage entrelacé signifie qu'une ligne sur deux lignes est allumée par le faisceau d'électrons. Après avoir une synchronisation verticale, les lignes restantes sont ensuite balayées. Nous utiliserons le premier dans notre projet.

Pixel Data [23:0]																	
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R[7:0]						G[7:0]						B[7:0]					

*Interface HDMI pour les composantes d'un pixel*

Le temps nécessaire pour effectuer un balayage d'une trame (plein écran) est appelé temps de balayage vertical. Et son inverse est la fréquence de champ, c'est-à-dire la fréquence de rafraîchissement d'un écran. L'affichage standard pour un écran VGA a une fréquence de trame de 60 Hz. La fréquence de clock de l'affichage est alors de  $799 \times 524 \times 60 = 25\text{MHz}$ . Car la taille de l'affichage est plus grande que la taille actuelle de l'écran.

$\{h\_total, h\_sync, h\_start, h\_end\} \leq \{12'd799, 12'd95, 12'd141, 12'd781\}$   
 $\{v\_total, v\_sync, v\_start, v\_end\} \leq \{12'd524, 12'd1, 12'd34, 12'd514\}$   
 Fréquence BCLK =  $799 \times 524 \times 60 \text{ Hz} = 25120560 \text{ Hz} \approx 25\text{MHz}$



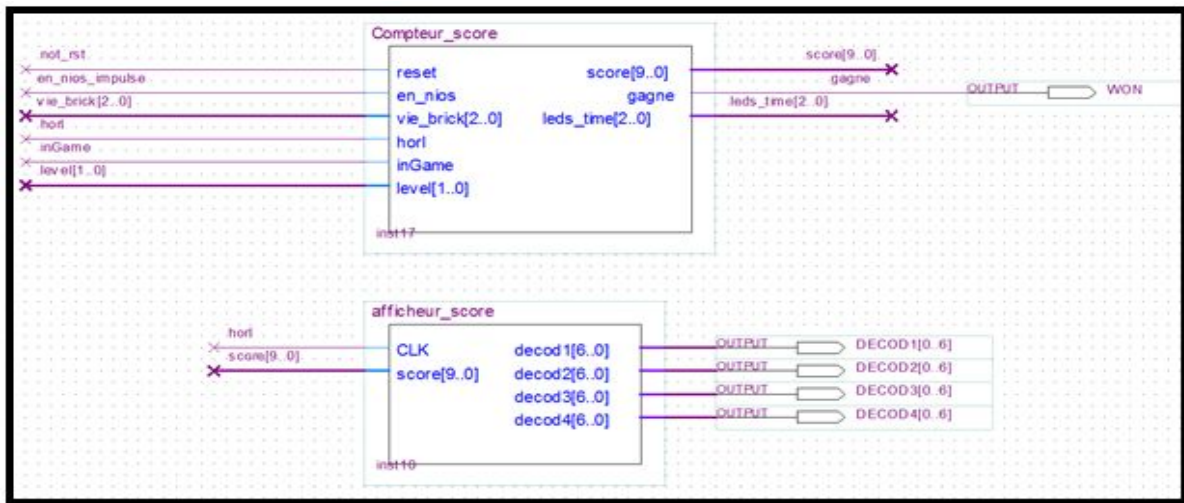
*Signaux de la synchronisation*

Afin de favoriser la détection et les calculs de la position de tous les composants affichés à l'écran, comme le bord de la balle ou d'une brique, nous convertissons les coordonnées de l'affichage à l'écran en coordonnées de l'affichage réel à l'écran. Par exemple,  $(h\_start, v\_start)$  est converti en coordonnées  $(0, 0)$  et on autorise les déplacement des objets en dehors de cette fenêtre d'affichage.



## IV - Compléments

### 1) Score

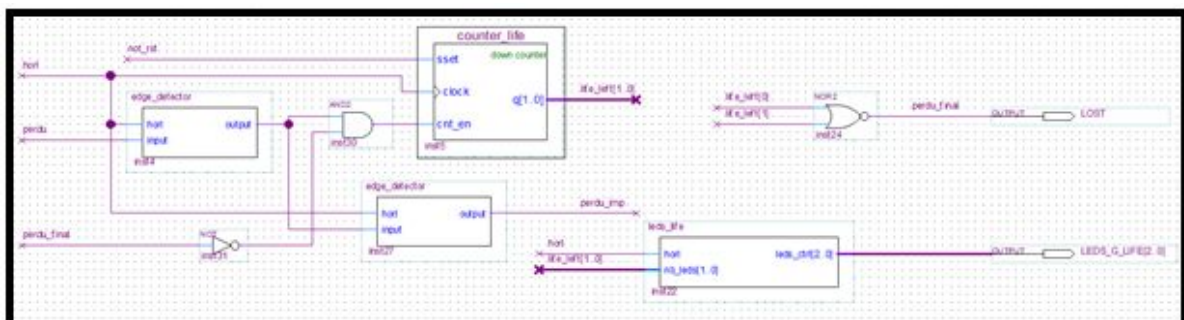


La première fonctionnalité ajoutée a été une méthode de calcul et d'affichage du score, constituée du bloc **Compteur\_score**, d'un bloc **afficheur\_score** qui affiche le score sur les quatre afficheurs 7-Segments, et d'un bloc **leds\_time** (non représenté ici) permettant de donner une indication de l'évolution du bonus de temps par des leds rouge qui s'éteignent au fur et à mesure de la partie.

Le bloc **Compteur\_score** permet le calcul du score, qui est la somme d'un score proportionnel au nombre de briques tuées (+5 points par brique cassée), ainsi que d'un bonus de temps, qui est décroissant avec la durée du jeu, et ne s'ajoute qu'en cas de victoire. Pendant le jeu, seul le score relatif aux briques tuées est envoyé, le bonus de temps étant représenté sur les leds rouge.

Le bloc **afficheur\_score** permet d'afficher le score actuel sur les afficheurs 7-Segments.

### 2) Vies



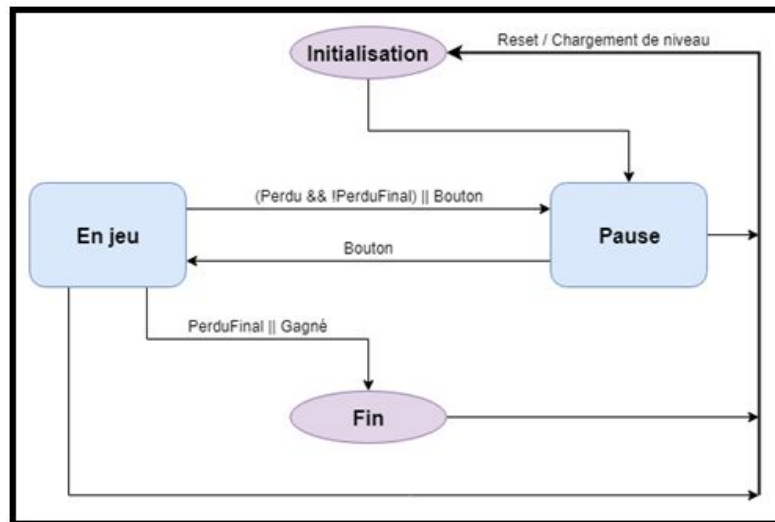
Le joueur dispose de 3 vies par session de jeu. Pour cela un compteur (**counter\_life**) initialisé à la valeur 3 se décrémente dès qu'une partie est perdue (réception du signal `perdu` du Nios). Le nombre de vies restantes est ensuite affiché sur 3 leds verte par l'intermédiaire du bloc **leds\_life**. La partie se termine lorsque le compteur atteint 0. Il faut alors recharger un niveau.

### 3) Gestion de la pause

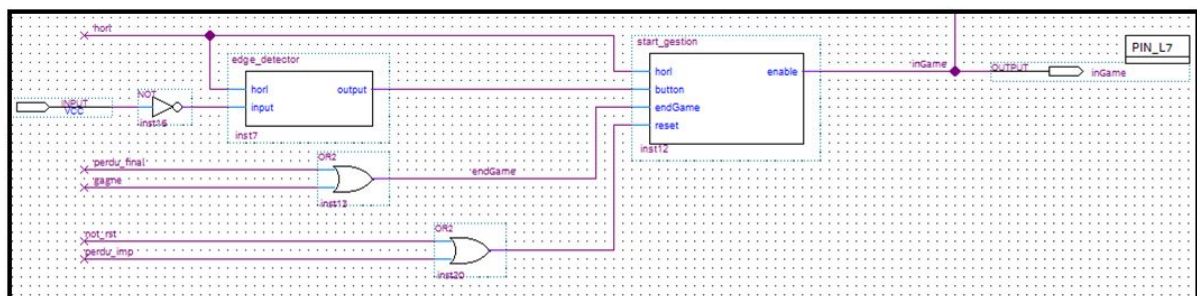
Le déroulement de la partie est mis en pause lors de différents évènements :

- ☐ Démarrage
- ☐ Perdu mais encore des vies
- ☐ Perdu et aucune vie (fin de la partie)
- ☐ Bouton

Ainsi, on a une machine d'état sous cette forme :



L'implémentation de cette machine se fait dans le bloc suivant :



### 4) Son

I2S est une norme de bus développée pour la transmission de données audio entre appareils audio numériques et largement utilisée dans les appareils multimédias.

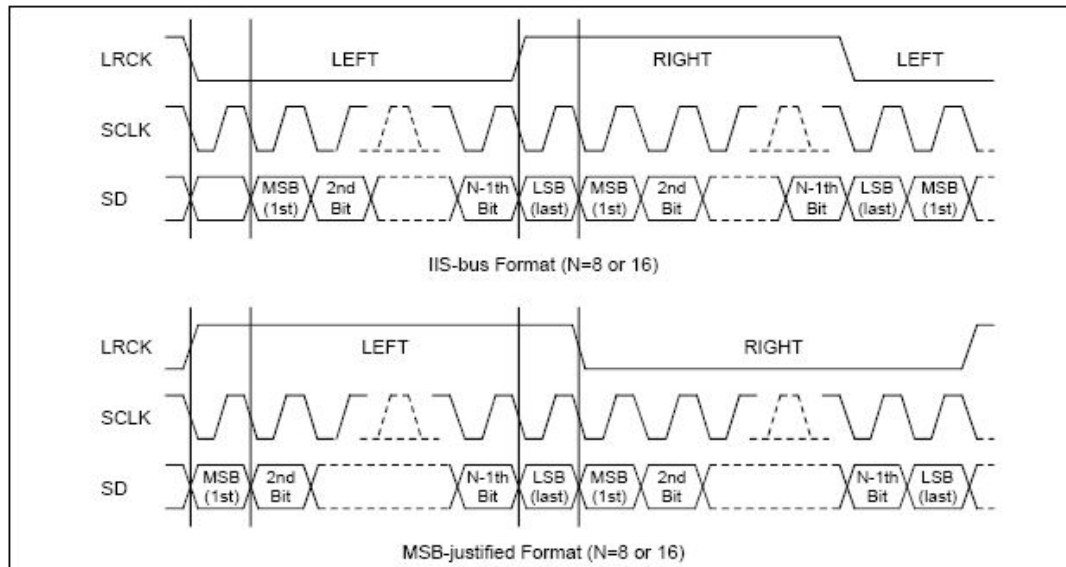
Cette norme a 3 signaux principaux :

1. L'horloge série SCLK, également appelée horloge binaire (BCLK), qui correspond à chaque bit audio numérique. Fréquence de SCLK =  $2 \times$  fréquence d'échantillonnage  $\times$  nombre d'échantillons.
2. L'horloge de trame LRCK, également appelée WS, permet de commuter les données des canaux gauche et droit. Lorsque LRCK est égal à "1", cela signifie que les données du canal



droit est en cours de transmission et inversement pour "0". La fréquence de LRCK est égale à la fréquence d'échantillonnage.

3. Les données série SDATA sont les données audio. De plus, afin de mieux synchroniser le système, un autre signal MCLK, appelé horloge principale ou horloge système (Sys Clock), correspond à 256 fois ou à 384 fois la fréquence d'échantillonnage.



*Signaux de l'interface audio*

Au final, nous avons essayé plusieurs fréquences d'échantillonnage pour les data. La simulation est bonne, mais l'implémentation n'a pas donné de résultats concluants dans le temps imparti..

## **V - Ressources**

Notre projet, dans sa dernière version utilise les ressources suivantes :

Top-level Entity Name	main
Family	Cyclone V
Device	5CGXFC5C6F27C7
Timing Models	Final
Logic utilization (in ALMs)	2,234 / 29,080 ( 8 % )
Total registers	2671
Total pins	116 / 364 ( 32 % )
Total virtual pins	0
Total block memory bits	464,476 / 4,567,040 ( 10 % )
Total DSP Blocks	5 / 150 ( 3 % )
Total HSSI RX PCSs	0 / 6 ( 0 % )
Total HSSI PMA RX Deserializers	0 / 6 ( 0 % )
Total HSSI TX PCSs	0 / 6 ( 0 % )
Total HSSI PMA TX Serializers	0 / 6 ( 0 % )
Total PLLs	1 / 12 ( 8 % )
Total DLLs	0 / 4 ( 0 % )

Nous utilisons relativement peu de ressources logiques. Ce qui peut être gage d'une implémentation optimisée.

Le nombre de pins d'entrée/sortie du FPGA utilisé est assez important. En effet, nous voulions utiliser au maximum les fonctionnalités proposées par la carte de développement telles que les LEDs, les afficheurs, etc...

La majorité de la mémoire utilisée l'est pour stocker les différentes images.

Finalement, les DSP sont principalement utilisées dans les contrôleurs d'affichage afin de calculer le l'adresse mémoire correspondant au pixel à afficher. Et la PLL est utilisé pour générer le signal d'horloge de l'HDMI.

## **Conclusion**

Durant ce projet, nous avons atteint les objectifs de la mission et nous avons également ajouté des fonctionnalités supplémentaires, telles que la pause ou les vies du joueur.

Nous avons rencontré des difficultés avec la détection des collisions durant toute la durée du projet. L'implémentation de l'affichage nous a également posé beaucoup de problèmes, tout comme le son qui n'a pas pu être finalisé dans le temps imparti. L'intégration des différents blocs conçus séparément a aussi posé problème.

Cependant, nous avons finalement réussi à surmonter la majorité de ces difficultés en nous organisant efficacement et en prenant en main Quartus et la programmation en VHDL de manière générale.

Ce TL nous a été fort bénéfique car nous avons appris concrètement à programmer un FPGA et à utiliser plusieurs des fonctionnalités disponibles. Ce projet a donc été très intéressant et enrichissant pour nous trois.