PROJET NESTOR

Systèmes numériques 2

Produit Bruno - Joachim Burket – T2-a

Fribourg, 21.05.2016

Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

TABLE DES MATIÈRES

1 INTRODUCTION

Le but de ce projet est de réaliser un jeu sur écran VGA. Le choix est libre, et nous avons décidé de faire un jeu à deux joueurs, dans lequel ils doivent se tirer dessus.
Le projet sera codé en VHDL à l'aide de la suite ISE de Xilinx. La plateforme de développement que nous utilisons est une Nexys3, basée sur une Spartan 6.
Aucun cahier des charges spécifique n'est imposé, nous l'avons donc défini par rapport aux fonctionnalités de jeu souhaitées.
La suite du projet s'est déroulée selon les étapes suivantes :

- Réfléchir aux fonctions précises du jeu
- Etablir un cahier des charges
- Analyser le hardware
- Etablir une Architecture du projet
- Coder les composants du projet
- Tester les composants

2 DESCRIPTION DU JEU

NEStor est un jeu ou deux joueurs doivent s'affronter sur une carte en 2D.
Le player 1 se prénomme « Nestor », et le player 2 « Grudu ». Les deux joueurs commencent de part et d'autre de la carte. Ils peuvent se déplacer de tous les côtés, et le personnage regarde dans la direction du déplacement. Des murs sont placés sur la carte (éventuellement possibilité de détruire certains murs). Chaque joueur est muni d'un EXTERMINATOR 2000 et peut tirer dans la direction dans laquelle le personnage regarde. Le joueur peut tirer *trois* (à définir) cartouches, puis doit attendre qu'elle ait touché un adversaire ou un mur pour en retirer.

Le but du jeu est de tuer l'adversaire en lui tirant dessus.
Nestor et Grudu ont 8 de points de vie avant de mourir. Ils peuvent également avoir certaines caractéristiques, *exemple :* Nestor à moins de points de vie mais se déplace plus rapidement, ou encore les tirs de Grudu sont plus puissants pour casser des murs, ou ses balles se déplacent plus vite, …).

2.1.1 Tâches à implémenter

Notre objectif est d'implémenter le jeu ci-dessus sur l'écran VGA 800x600, et de contrôler les personnages avec des manettes de NES.

Les fonctionnalités à implémenter seront :

- Gérer le début de partie (init)
- Création de la map de base
- Création d'un joueur
- Déplacements d'un joueur dans la carte
- Collisions avec les obstacles + fin de l'écran
- Gérer les tirs du joueur
- Gérer les balles tirées
- Gérer l'impact des balles
- Gérer et afficher les points de vie d'un joueur
- Gérer les manettes NES (5V ou 3.3V ?) (Ou Nintendo64 ?)
- Gérer la fin de la partie

2.1.2 Optionnel

- Son
- Murs cassants
- Grenades
- >2 joueurs
- Joueurs avec des caractéristiques différentes
- Couleurs
- Plusieurs maps
- Gestion d'un menu (son, couleurs…)
- Sauvegarder des scores

3 ANALYSE

Cette partie traite de l'étude du matériel à disposition pour ce projet, comprenant la FPGA et ses périphériques.

3.1 VGA

Le VGA (Video Graphic Array) a été inventé en 1987 par IBM. Le standard était défini pour une définition de 640x480 pixels, et pour un connecteur D-SUB 15 pins. Le standard SVGA quant à lui, définit les écrans de définition 800x600 et 256 couleurs, qui est celle qui est utilisée dans ce projet.

3.1.1 Hardware

Le connecteur VGA possède 15 pins, mais seules 5 sont nécessaires :

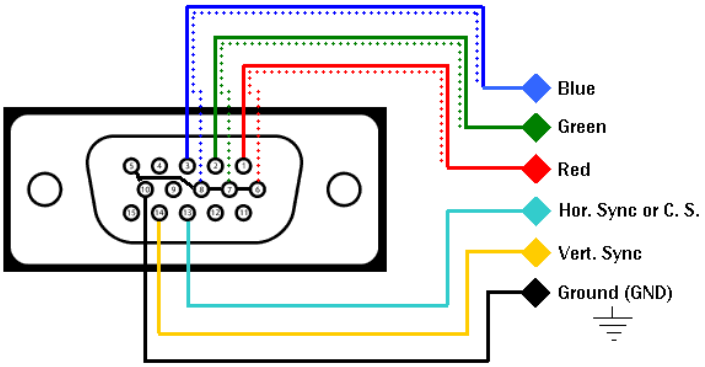


Figure 1 : Pinning du connecteur VGA

- Blue : sortie du bleu sur 2 bits
- Green : sortie du vert sur 3 bits
- Red : sortie du rouge sur 3 bits
- Hsync : synchronisation horizontale. Indique que le curseur doit revenir à 0 une ligne en dessous
- Vsync : synchronisation verticale. Indique que le curseur doit revenir en haut à gauche de l'écran.

Les pins R-G-B sont des pins analogiques. La conversion D/A se fait avec des résistances comme suit :

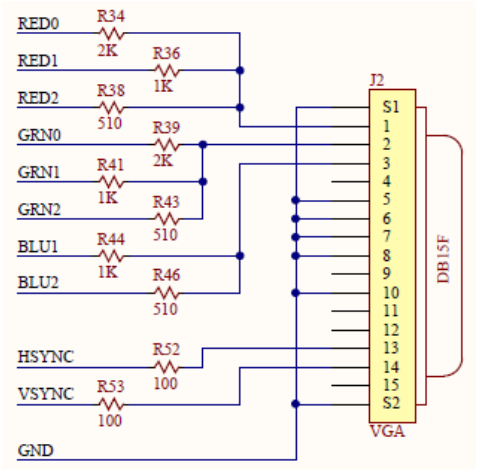


Figure 2 : Conversion D/A des signaux RGB

Le bit de poids faible à une résistance de 2KΩ, et celui de poids fort une de 510Ω. Ce qui veut dire que le bit RED0 donne une tension plus faible que le bit RED2 lorsqu'il est à 1. La somme des tensions des signaux est ensuite faite, et le signal résultant est envoyé sur la pin de la couleur (pins 1, 2 et 3).

3.1.2 Timing

Sur les anciens écrans CRT (écrans cathodiques) en VGA, certains délais de temps étaient nécessaires pour le rafraîchissement de l'écran. Ce dernier se fait ligne après l'autre, de gauche à droite, puis de haut en bas. Lorsque le rafraichissement d'une ligne est terminé, il faut faire revenir le « curseur » tout à gauche de l'écran, une ligne en dessous. Et lorsque le rafraichissement de la dernière ligne est terminé, le curseur doit revenir tout en haut à gauche. Pendant de temps, aucune information n'est affichée. Ce temps de « blanking » permet au faisceau d'électrons de se déplacer. La Figure 3 démontre ces temps, sur un écran vga en 640x480 :

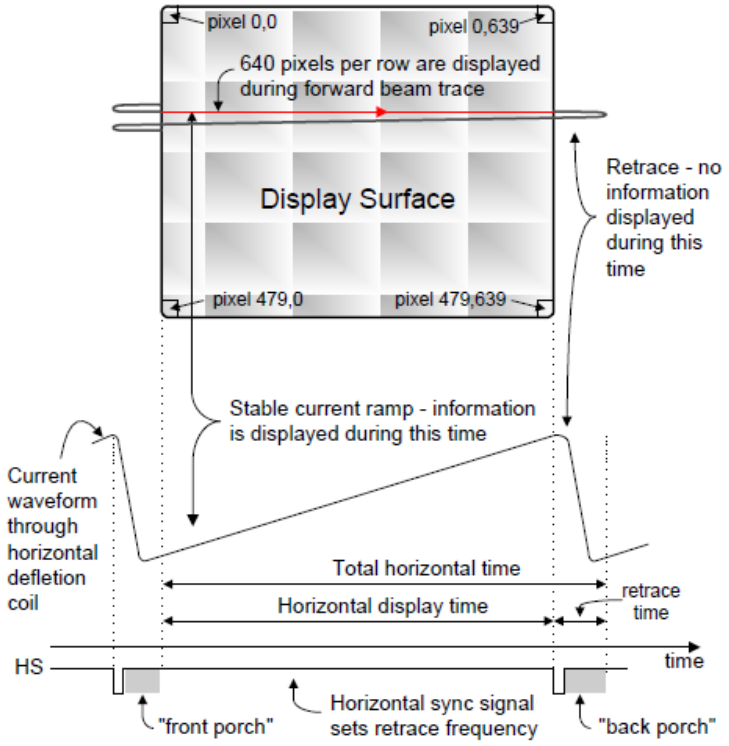


Figure 3 : Timing d'un écran VGA 640x480

La pulsation de Hsync indique que le curseur doit revenir à 0. Sur la figure, on observe que lorsque le faisceau d'électrons se déplace de gauche à droite, 640 pixels sont affichés, et que lorsqu'un Hsync est généré, le faisceau revient au début sans affichage (blank). Les petites boucles de retours à gauche et à droite de l'écran

sont appelées « front porch » et « back porch ». Ces boucles ainsi que le retour du faisceau impliquent des délais.

Ces temps étaient nécessaires pour les écrans cathodiques. Les écrans LCD n'auraient pas besoin de ces délais, mais ils les ont adoptés afin qu'ils puissent utiliser le VGA.

L'écran qui sera utilisé durant ce projet à une définition de 800x600 pixels. La clock pour les pixels doit alors être de 40 [MHz], pour un rafraichissement de l'écran à 60 [Hz]. Cette clock de pixels incrémente un compteur horizontal, qui lui incrément un compteur vertical. Afin de permettre les timings de « front porch » et « back porch » cités en dessus, ces compteurs vont plus loin que 800 et 600. Les valeurs sont les suivantes :

40 MHz pixel clock	Active Video	Front porch	Sync pulse	Back porch
Horizontal	800	40	128	88
Vertical	600	1	4	23

Le schéma suivant explique également comment ces timings sont gérés avec les compteurs :

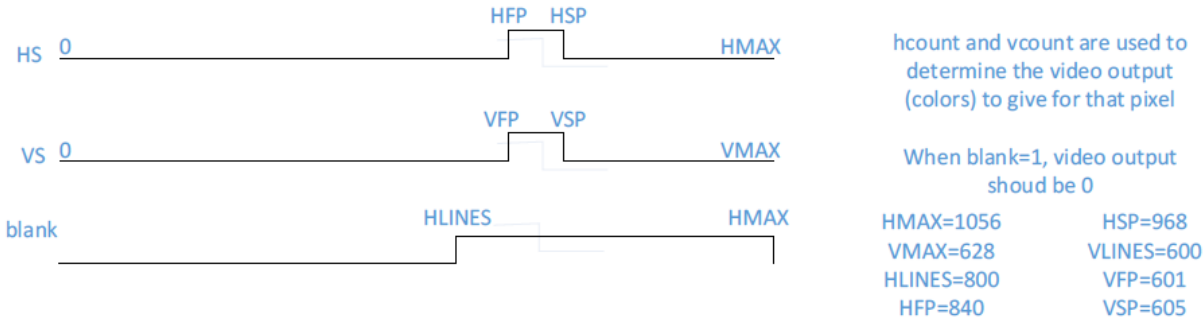


Figure 4 : Compteurs des pixels

- Horizontal Front Porch = HFP - HLINES
- Horizontal Back Porch = HMAX - HSP
- Hsync pulse = HSP - HFP
- Vertical Front Porch = VFP - VLINES
- Vertical Back Porch = VMAX - VSP
- Vsync pulse = VSP - VFP

3.2 Carte de développement

Nous utilisons la plateforme de développement Nexys3 développée par Digilent. Elle est basée sur une FPGA Xilinx Spartan-6.
La carte est composée des caractéristiques suivantes :

- FPGA Xilinx Spartan-6 LX16 324 pin
- RAM Cellular 16Mbyte (x16)
- Mémoire non-volatile PCM 16 Mbytes en SPI
- Mémoire non-volatile PCM 16 Mbytes en parallèle
- Ethernet 10/100
- Port USB2 pour la programmation et le transfert de données
- Port USB-UART et USB-HID (pour souris et clavier)
- Port VGA 8 bit
- Oscillateur CMOS 100MHz
- 72 Entrées/Sorties d'extension
- GPIO pour 8 LEDs, 5 boutons poussoirs, 8 switchs et 4 affichages 7 segments

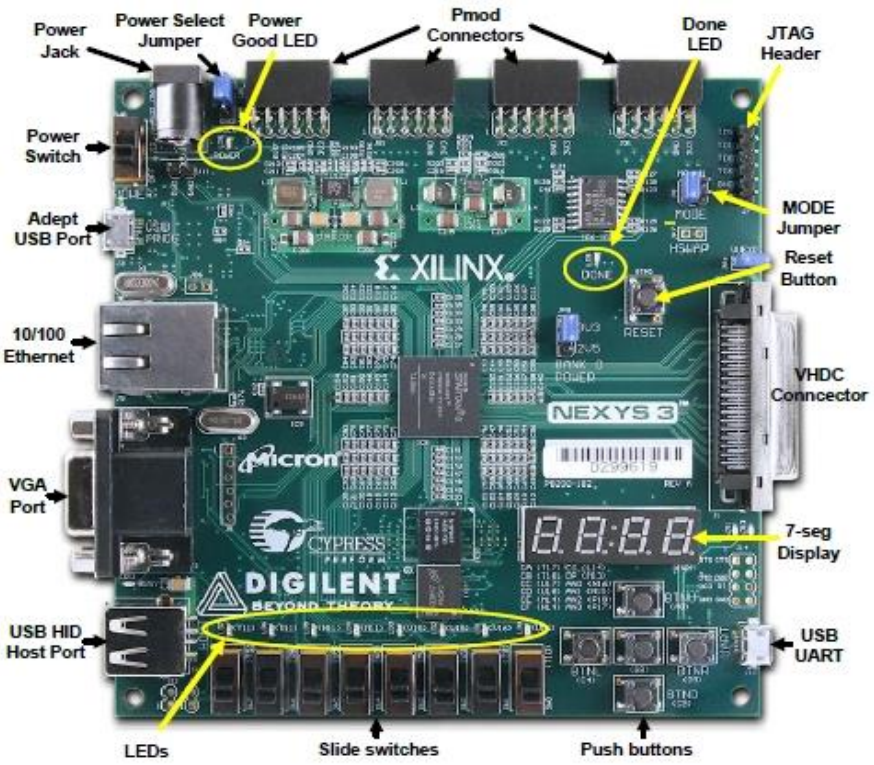


Figure 5 : Illustration de la carte Nexys3

La Nexys3 dispose de boutons, de switches, d'affichages 7 segments, et de LEDs, qui peuvent être utiles pour ce projet.

La Spartan-6 LX16 est composée de :

- 2278 espaces (slices) contenant chacun 4 LUTs à 6 entrées et 8 bascules
- 576kbits de RAM
- Deux domaines d'horloges (4 DCMs et 2 PLLs)
- 32 DSP
- Clock jusqu'à 500MHz et plus

Device	Logic Cells(1)	Configurable Logic Blocks (CLBs)			DSP48A1 Slices(3)	Block RAM Blocks		CMTs(5)	Memory Controller Blocks (Max)(6)	Endpoint Blocks for PCI Express	Maximum GTP Transceivers	Total I/O Banks	Max User I/O
		Slices(2)	Flip-Flops	Max Distributed RAM (Kb)		18 Kb(4)	Max (Kb)						
XC6SLX16	14,579	2,278	18,224	136	32	32	576	2	2	0	0	4	232

Figure 6 : Caractéristiques Spartan-6 LX16

3.3 Joysticks

Afin de contrôler les personnages, et de pouvoir jouer « agréablement », un joystick est une solution intéressante.
Nous avons reçu pour le projet deux joysticks PmodJSTK de Digilent. Ces joysticks se connectent par une interface SPI en slave.

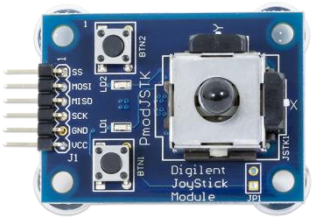


Figure 7 PmodJSTK de Digilent

Ces joysticks remplacent les manettes NES, initialement prévues pour ce projet. Ceci car il été difficile de trouver des manettes en bon état et à un prix acceptable.

4 CONCEPTION

La partie conception expose l'approche qui a été utilisée lors de ce projet. Elle décrit également pourquoi certaines techniques ont été employées plutôt que d'autres.

4.1 Architecture

Au départ nous sommes partis sur une architecture simple en séparant les diverses fonctions à implémenter en composants avec une entité VHDL. Notre architecture a beaucoup évolué tout au long du projet, car nous ne savions pas forcément toutes les précisions de certains composants. En plus nous avons changé l'algorithme d'affichage au milieu du projet, vu que nous nous sommes rendus compte qu'il n'était pas adapté.

Voici notre première architecture :

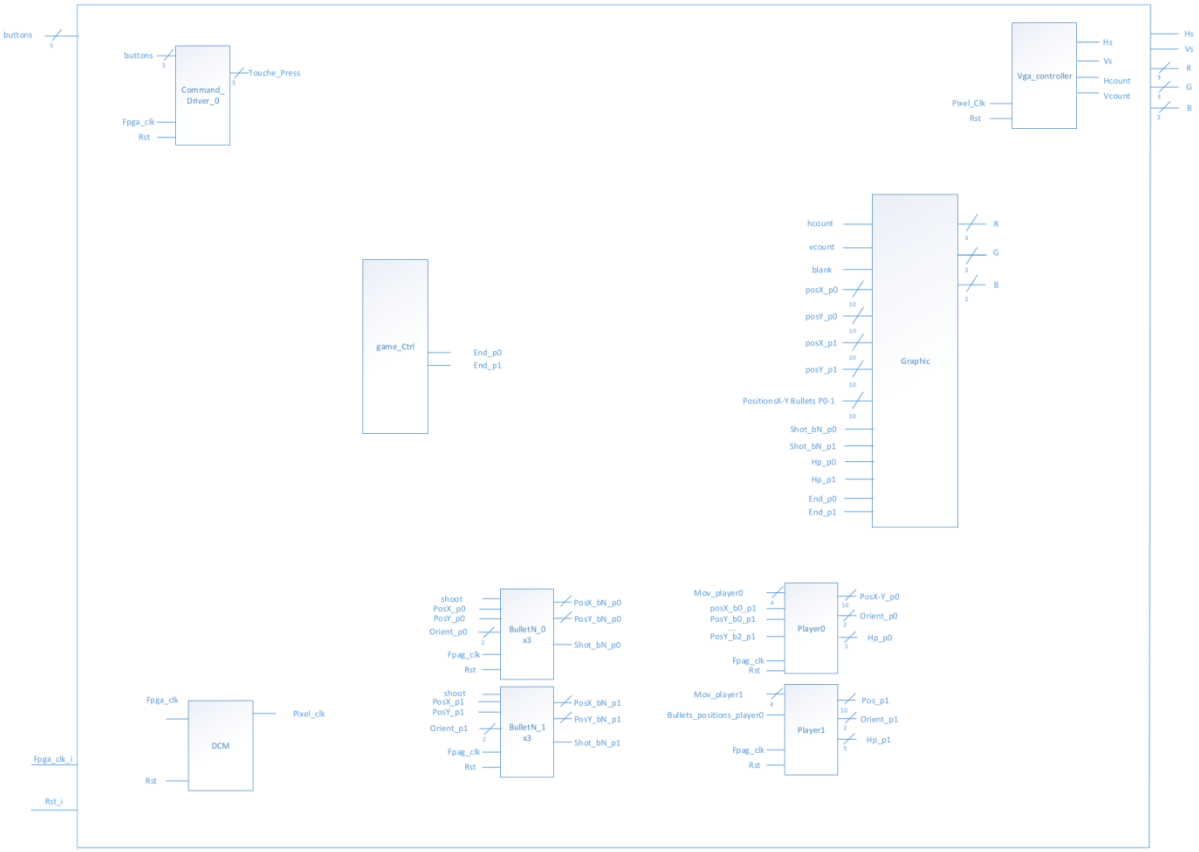


Figure 8 : Architecture initiale du projet

Certains composants ont changé tout au long du projet, ou n'existent même plus.

4.2 Top level

Voici le composant Top level du projet.

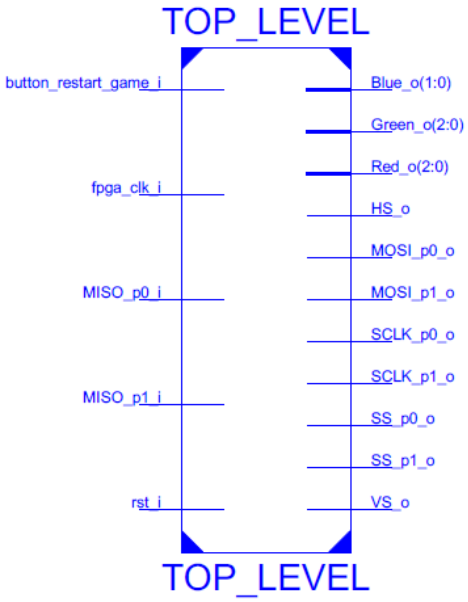


Figure 9 : Composant Top level

- Button_restart_game_i : bouton pour rejouer une partie. Est mappée sur le switch SW1 de la carte
- Fpga_clk_i : Clock de la carte à 100 MHz

- MISO_p0_i : entrée data pour le bus SPI du joystick du joueur2
- MISO_p1_i : entrée data pour le bus SPI du joystick du joueur 1
- Rst_i : Reset de la carte. Mappé sur le switch SW0
- Blue_o, Green_o, Red_o : Sorties de la couleur sur l'écran
- HS_o : Horizontal sync
- VS_o : Vertical sync
- MOSI_p0_o : sortie pour le SPI
- MOSI_p1_o : sortie pour le SPI
- SCLK_p0_o : sortie pour le SPI
- SCLK_p1_o : sortie pour le SPI
- SS_p0_o : sortie pour le SPI
- SS_p1_o : sortie pour le SPI

4.3 DCM

Pour le projet, plusieurs clocks étaient nécessaires. La clock en entrée de la FPGA est à 100 [MHz].

- 40 [MHz] : pixel_clk_s. C'est la clock utilisée pour les compteurs des pixels.
- 5 [MHz] : game_clk_s. Clock utilisée pour la gestion du jeu. Que ce soit pour les déplacement des balles, des joueurs, lecture des collisions, …
  Une fréquence de 5 [MHz] a été choisie car c'est la fréquence la plus basse que peut sortir le DCM.
- 100 [MHz] : spi_clk_s. Clock nécessaire pour la gestion des joysticks en SPI
- 30 [MHz] : clock de réserve

4.4 Contrôleur du jeu

Le contrôleur du jeu est le composant qui gère les états du jeu. Il était donc logique de faire une machine d'état pour ce composant. Celle implémentée est une machine de Moore. Les états sont :

- Game
  - Le jeu est en cours
- End_game_p0
  - Le joueur 0 a gagné
- End_game_p1
  - Le joueur 1 a gagné

Voici la machine d'états :

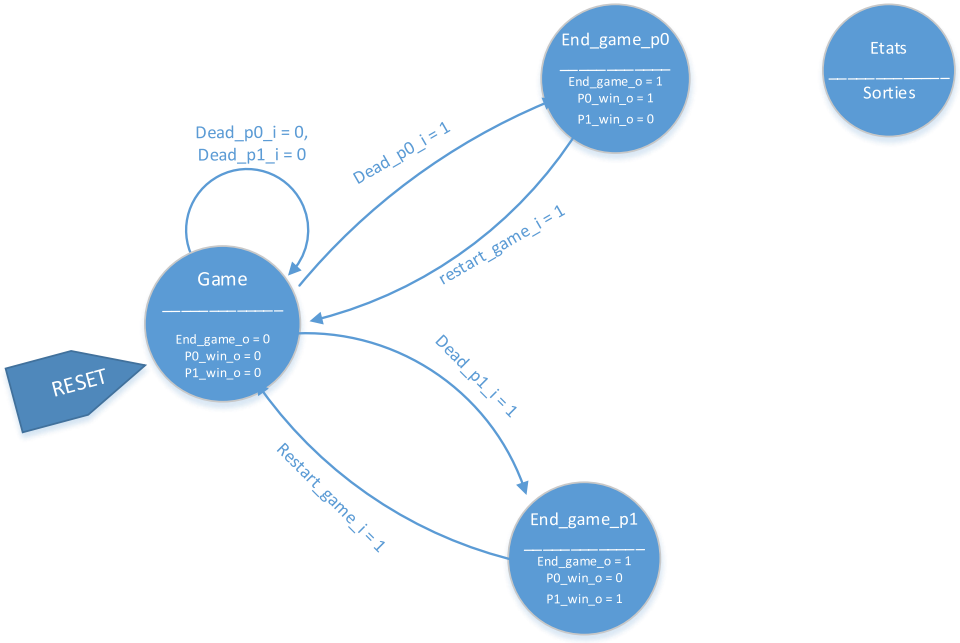


Figure 10 : Machine d'état du contrôleur de jeu

Au démarrage, le jeu est prêt à être joué. Ensuite, lorsqu'un joueur n'a plus de vie, un écran de fin est affiché, de la couleur du vainqueur. Pour que le jeu repasse dans l'état « game », l'entrée « restart_game_i » doit être activée (mappée sur un switch SW1 de la carte).

4.5 Gestion des Graphiques

Cette partie présente tout d'abord comment stocker dans la FPGA les images que l'on veut afficher, puis comment les joueurs, les balles ainsi que la carte ont été gérés.

4.5.1 Stockage d'images

Pour gérer les graphiques, il faut déjà pouvoir afficher une image sur l'écran VGA, et donc stocker cette image dans une ROM dans la FGPA.
Une ROM peut être stockée de deux manières :

1) Distributed RAM :

Générée lorsqu'on lit dans de la RAM de manière asynchrone, elle est générée avec des LUTs et des bascules. La taille maximale de distributed RAM dans la Spartan6 est 136 Kbits.
Afin de déclarer une ROM, il suffit de déclarer un RAM en constant. Le code suivant permet de déclarer une ROM :

```
type rom_type is array (0 to 127) of std_logic_vector (19 downto 0);
constant ROM : rom_type:= (
    X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300", X"08602",
    X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500", X"04001", X"02500",
    (…)
    X"04078", X"01110", X"02500", X"02500", X"0030D", X"02341", X"08201", X"0410D"
);
```

Figure 11 : Déclaration d'une ROM

La ROM déclarée est une 128x20. Elle a donc un bus d'adresses sur 7 bits, et une sortie de données sur 20 bits. Et pour lire dedans, il faut utiliser le code suivant :

```
do <= RAM( conv_integer(addr));
```

Figure 12 : Lecture dans une Distributed RAM

L'adresse de la RAM doit être un integer, c'est pour ça que si « addr » est un std_logic_vector, il faut le convertir.

Pour simplifier le stockage d'une image, nous avons écrit un script en python. Il prend en entrée une image « jpg », et ressort un fichier texte avec la déclaration de la ROM. Ce code se trouve à l'annexe 13.1.

2) Bloc RAM :

Générée lorsqu'on lit dans de la RAM de manière synchrone, elle est stockée dans les blocs de RAM internes de la FPGA. C'est de la RAM physique, la spartan6 en possède 576kbits, utiliser cette technique est donc un gain important de LUTs. Il n'existe pas de ROM dans la FPGA, celles crée sont donc stockées dans la RAM en constantes.
Pour que le synthétiseur puisse utiliser cette RAM physique, il faut faire attention à la déclaration et l'utilisation. Sans ça, XST implémente la RAM avec des LUTS.
La déclaration est la même que pour une « distributed RAM ».
Lecture synchrone :

```
process (clk)
begin
  do <= RAM( conv_integer(addr));
end process;
```

Figure 13 : Lecture dans un bloc RAM

Il est également possible d'utiliser l'IPCORE générateur, ceci à l'avantage d'être sûr que la RAM faite utilise bien des blocs physiques. Pour cela :

New source => IPCORE => Bloc memory generator.
Sélectionner « single port ROM » pour une mémoire à accès simple, ou « dual port ROM » pour une à accès double (donc deux composants peuvent la lire en même temps).
Ensuite, il faut indiquer la tailler de la ROM, en bit de Width et de Depth.
L'exemple ci-dessous est une ROM single port de 1024 lignes de 8 bits.

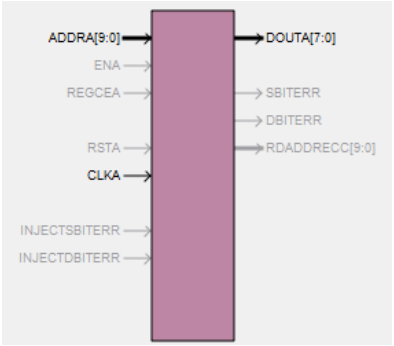


Figure 14 : ROM single port 1024x8

Il faut ensuite sélectionner le fichier « .coe » contenant ce que l'on veut stocker dans la ROM :

Figure 15 : chargement du fichier coe de la ROM

Enfin, cliquer sur « generate » pour créer la ROM.

Pour générer le .coe à donner à l'IPCORE à partir d'une image jpg, un script MATLAB provenant de http://www.lbebooks.com/downloads/exportal/vhdl_nexys_example24.pdf a été utilisé. Le code se trouve également en annexe.

4.5.2 Cartes

Le stockage de la carte a été le plus difficile. Nous avons dû faire plusieurs essais avant de trouver une solution qui ne remplissait pas toute la FPGA, et permettait une gestion des collisions la plus simple possible.

1er essai :
La carte était au format 800x600x8, et uniquement en deux couleurs (une pour les murs, et une pour le sol). Le fait d'avoir seulement deux couleurs permettait de gérer simplement les collisions des joueurs avec les murs, en lisant la couleur de la carte où se trouvait le joueur. Elle était stockée dans des LUTs.
Vu que pour tester les collisions, il fallait faire plusieurs lectures dans la carte à des endroits différents, chaque lecture instanciait une nouvelle carte. La FPGA a ainsi été remplie très vite, et il n'était plus possible de continuer. A ce stade, un joueur pouvait se déplacer dans la carte, et les collisions fonctionnaient.

2ème essai :
Pour afficher la carte 800x600x8 à l'écran, il est nécessaire de lire dedans qu'une seule fois par coup de clock. C'est en revanche lorsqu'il lire dedans pour gérer les collisions, nous lisions plusieurs fois par coup de clock. Ainsi, l'idée est d'avoir deux cartes distinctes. Une simplement pour l'affichage en 800x600x8, et une pour gérer les collisions en 200x150x1 (carte réduite 4x, et 1 seul bit pour gérer les collisions). Ceci permettant ainsi de faire les lectures des collisions, sans prendre trop de place dans la FPGA. De plus, au lieu d'avoir une carte à deux couleurs uniquement, qui facilitait grandement la gestion des collisions, une carte plus jolie a pu être affichée.
Nous avons donc tenté d'afficher la carte suivante sur l'écran, en gérant les collisions sur une autre carte :

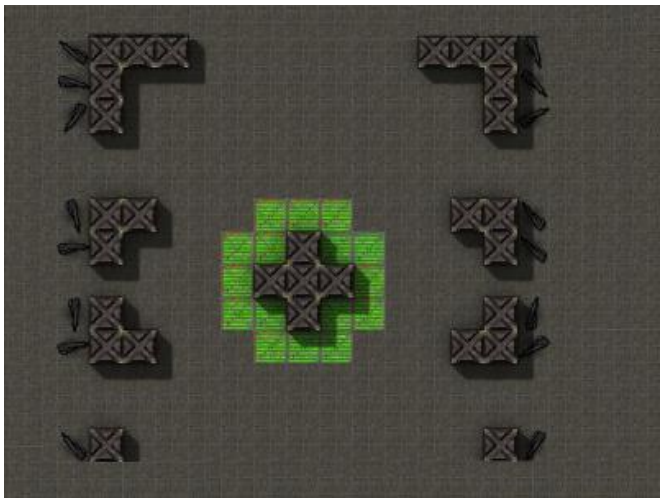


Figure 16 : Carte devant être affichée

Ce que nous ne savions pas, c'est que vu que l'ancienne carte qui était affichée n'était qu'en deux couleurs, elle ne prenait pas une place de 800x600x8, mais de 800x600. Et donc cette nouvelle carte était trop volumineuse pour la FPGA.

3ème essai :
Les LUTs de la FPGA permettent de stocker au maximum 136kbits, alors que dans les blocs de RAM il est possible d'aller jusqu'à 576 kbits. Pour cela, nous avons essayé de stocker la carte à afficher dans les blocs de RAM.
L'idée de séparer la une carte à afficher et une carte simplement pour la gestion des collisions reste.
Calculs de la taille de la carte :

- 800*600*8 = 3840 kbits
  - Pas possible de stocker une image parfaite de toute la carte, il faut donc réduire la taille
- 400*300*8 = 960 kbits
  - Toujours trop grand…
- 200*150*8 = 240 kbits
  - Passerai, mais la carte ne serait pas jolie.

Etant donné que les RAM ne sont pas assez grandes pour stocker toute la carte, il a fallu réfléchir à une autre méthode.
Le fait de stocker toute la carte permet d'afficher des cartes hétérogènes, et avec beaucoup de détails. Mais la RAM de la FPGA étant trop petite, ceci n'est pas possible.
Nous partons alors sur l'idée de stocker uniquement un bout de mur et un bout de sol. Chacun fait une taille de 40*40*8, ce qui fait une taille dans la RAM de 25,6 kbits.

Figure 17 : Carré de texture des murs

Figure 18 : Carré de texture du sol

Les carrés sont ensuite affichés les uns à côté des autres, en fonction de la disposition de la carte.
La taille prise dans les RAMs est très acceptable, et cette technique permet de modifier l'agencement des murs plutôt simplement.

La dernière technique présentée est celle qui est utilisée. La petite taille des sprites à stocker permet de stocker également les sprites des joueurs et les cartes des collisions dans la RAM.

4.5.3 Joueurs

Chaque joueur est un composant à part entière, et sort sa position X et Y, ainsi que son orientation. Sa position X et Y correspond au point en haut à gauche du sprite.
La gestion graphique du joueur réside donc en l'affichage de son image sur l'écran, en fonction de sa position et de son orientation.
Le stockage des sprites des personnages se faisait dans des LUTs au début, elles ont été transférées dans des blocs de RAMs. Chaque image fait une taille de 32x32x8.

1) Nestor

Figure 19: Nestor sprites

2) Grudu

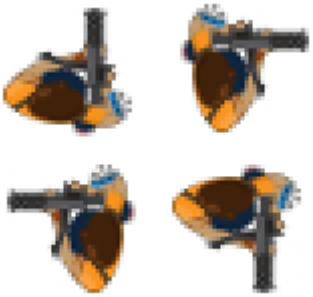


Figure 20 : Grudu sprites

Comme les sprites sont des carrés de 32x32, il faut « détourer » le joueur. Pour cela, lors de l'affichage, si la couleur est blanche, elle n'est pas affichée, et c'est donc le sol qui est affiché.

4.5.4 Balles

Chaque balle est également un composant, qui ressort sa position X et Y. La balle est un rond dans une sprite de 8x8. Etant donné que ce n'est pas trop imposant à stocker, elle a été stockée dans des LUTs.

4.5.5 Points de vie

Pour indiquer le niveau de vie de chaque joueur, le nom de chaque joueur avec une barre de vies se trouvent en haut de l'écran. Les positions sont définies par ce croquis.

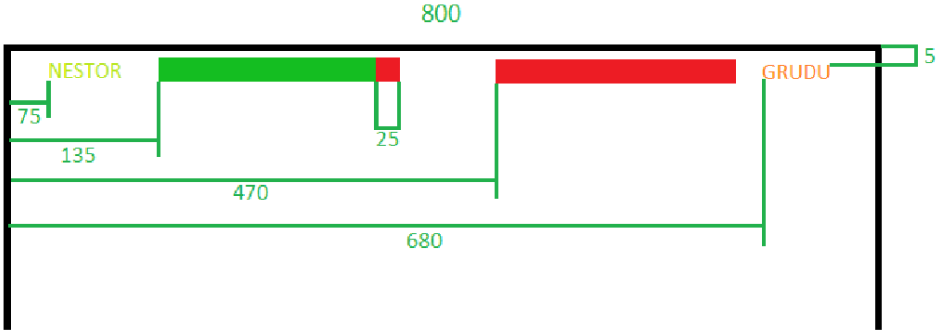


Figure 21 : Croquis de l'agencement de la bannière en haut de l'écran

Les distance sont en pixels. Les textes des joueurs sont à la couleur du joueur respectif.
Toutes les constantes sont stockées dans le fichier « local.vhd ». Il est donc possible de modifier la disposition de la bannière facilement.

4.6 Gestion des collisions

Comme expliqué plus haut, une carte permet de lire où sont les murs de la carte. Cette carte est en 200x150, afin de diminuer la place prise dans les blocs de RAM. Elle se présente sous cette forme :

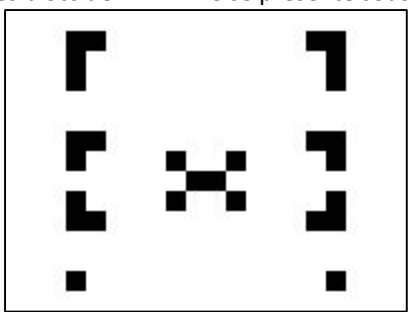


Figure 22 : Carte des collisions

Vu que la carte des collisions est compressée d'un facteur 4, les joueurs et les balles doivent aussi être plus petits.

	Affichage	Collisions
Carte	800x600	200x150
Joueur	32x32	8x8
Balle	8x8	2x2

Les joueurs et les balles ressortent leurs positions par rapport à la carte des collisions. Il faut donc multiplier par 4 la position en X et en Y avant d'afficher un objet dans le composant « graphic ».

La gestion des collisions se fait comme suit. Pour un joueur par exemple, pour savoir s'il est contre un mur, il va lire dans les quatre coins autour de lui si la carte des collisions est blanche ou noir. Si elle est blanche, il peut se déplacer, sinon il s'arrête.

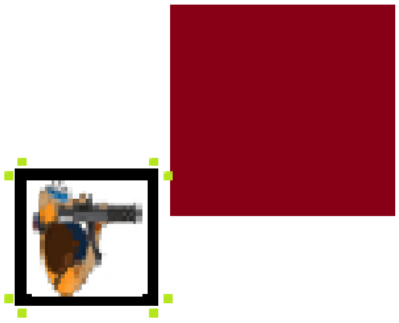


Figure 23 : Gestion collisions mur

Le carré brun est un mur, et les points verts sont les endroits où on lit dans la carte des collisions. Ces lectures se font donc dans les coins de la sprite carrée, même si le personnage ne remplit pas tout le carré.

Vu que deux joueurs et six balles doivent lire « en même temps » dans la carte des collisions, il faudrait un arbitrer pour les lectures. Au lieu de ça, nous avons instancié plusieurs fois des ROM « dual port ROM », c'est-à-dire pouvant être accédées par deux composants en même temps. Ainsi, il y en a une par joueur, et trois pour toutes les balles. La raison pour laquelle nous avons instancié une ROM par joueur, et non une ROM pour les deux, est qu'il faut lire en même temps depuis deux points du joueur, afin qu'il ne puisse pas traverser à moitié un mur.

4.7 Joueurs

Le composant d'un joueur gère le déplacement de ce dernier, ainsi que ses vies.

Pour le déplacement, la clock « game_clk_s » est utilisée. Etant donné qu'elle est encore trop rapide, la vitesse du déplacement est ralentie dans un compteur. Ceci a l'avantage de pouvoir gérer la vitesse des joueurs en changeant une simple constante. Le composant ressort sa position X et Y. Celle-ci correspond au point en haut à gauche de l'image du joueur.
Le composant doit également gérer les collisions avec les murs et les bords de la carte. Il se charge également d'indiquer en sortie son orientation, afin que l'affichage soit fait correctement.
Pour les vies, le composant reçoit en entrée un signal « hit_i », envoyé par les balles du joueur adverse. Lorsque le signal est à 1 (pendant un coup de clock), les vies sont décrémentées. La sortie « hp_p_o » indique au composant graphique les points de vie à afficher.
Lorsque le nombre de vie est égal à zéro, la sortie « dead_p0_o » est mise à 1.

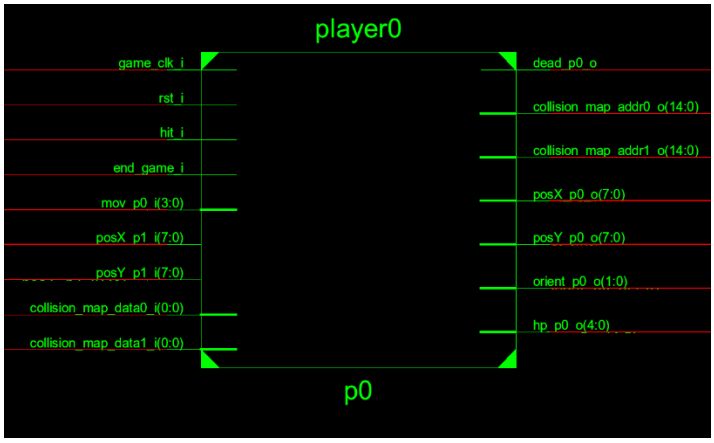


Figure 24 schéma bloc player

4.8 Balles

Chaque balle du jeu est un composant, ce qui ne permet pas un nombre infini de balles. Chacun des joueurs dispose donc de trois balles. Lorsque les trois sont tirées, il faut attendre qu'une d'elle ait touché l'adversaire, un mur, ou le bord de la carte pour qu'elle soit à nouveau disponible.
Afin de gérer les disponibilités des balles, un contrôleur (« bullet_ctrl_p ») est utilisé pour chaque joueur. Ce composant se charge de mettre en forme le signal reçu pour tirer. C'est lui qui gère quelle balle est tirée en fonction de celles disponibles. Afin qu'un long appui sur la touche du shoot ne tire qu'une balle, un monoflop est repris d'un ancien travail pratique. Ainsi, lorsque le bouton est pressé pendant une durée quelconque, le signal « shoot_b_o » est actif durant un seul coup de clock.

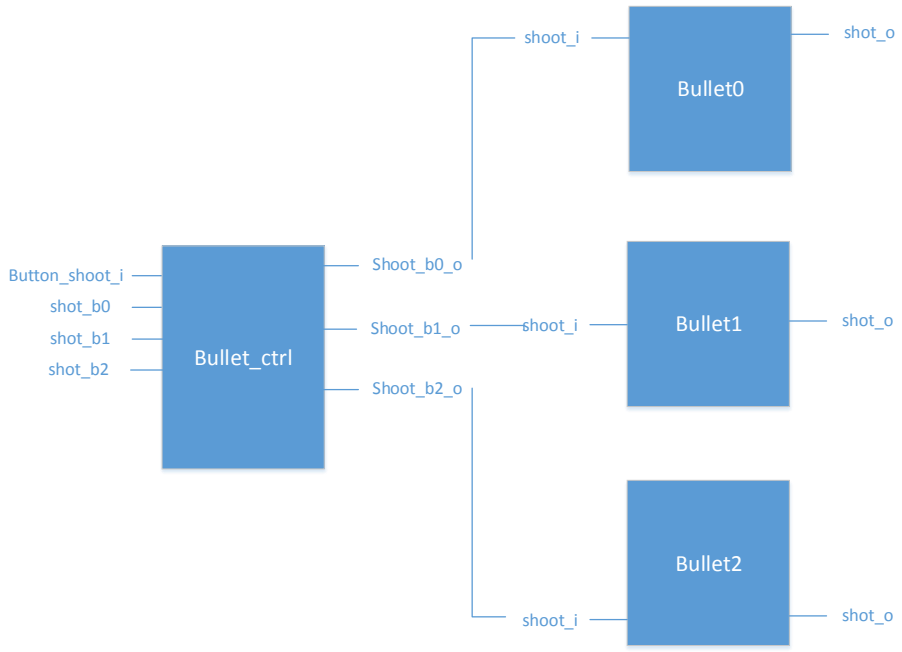


Figure 25 : Gestion des balles

Le composant « bullet » reçoit donc une impulsion d'un flanc de clock en entrée, et sait ainsi que la balle doit partir. Lorsque la balle est tirée, la sortie shot_o est à 1.

4.9 Joysticks

Pour les composants joystick Il existe une librairie écrite par Digilent même, sur laquelle nous nous sommes basés. Les joysticks se connectent par SPI sur les pins GPIO de la FPGA comme expliqué précédemment dans le document. Au lieu de directement connecter les deux joysticks sur le même bus, nous avons simplement implémenté 2 master SPI dans la FPGA. Ceci est simplement dû au fait que physiquement nous ne pouvons pas connecter les deux joysticks sur les mêmes pins (en séparant le pin SS) pour créer un bus.

Les deux masters SPI se présentent donc sous la forme d'un composant de type « top level » qui rassemble plusieurs autres composants. En plus de ces deux masters il y a un diviseur d'horloge à 5 Hz qui est utilisé par les deux masters.

De ce fait notre schéma global est un schéma à plusieurs étages :

- TOP levels
  - Composants de jeu
  - Master SPI1
    - Composants SPI
  - Master SPI2
    - Composants SPI

Les Masters SPI fournissent un service qui converti un flux de 5 bytes sur les deux bus SPI en un std_logic_vector (7 downto 0) parallèle pour indiquer l'état des boutons.
Ce service est fourni avec une fréquence de 5 Hz avec une horloge SPI interne de 66.67kHz.

Pour le tir des personnages nous avons décidé de ne pas l'implémenter sur les boutons des joysticks car 5Hz n'est pas suffisant.

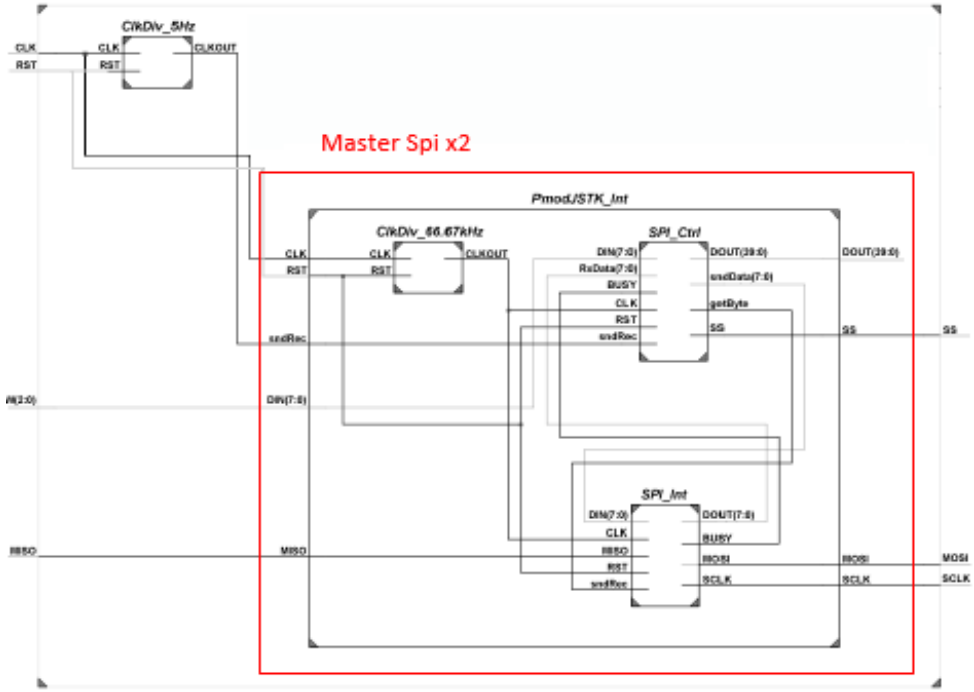


Figure 26 Schéma bloc d'un master SPI Joystick

Les deux Joysticks se connectent sur les pins « J » de type GPIO, sur le « A » supérieur et le « C » supérieur :

Figure 27 Connexion des Joysticks

Connexion du Joystick pour Grudu
Connexion du Joystick pour Nestor

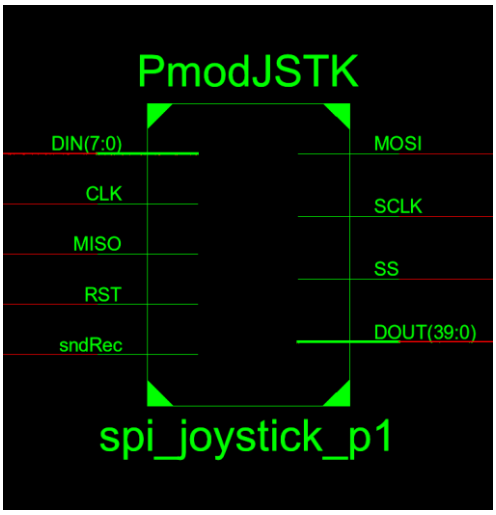


Figure 28 Schéma bloc Joystick

5 IMPLÉMENTATION

Cette partie décrit comment les fonctionnalités ont été implémentées.
Afin de pouvoir rester consistant dans notre projet, nous avons aussi décidé de respecter des directives de nomenclature usuelle en VHDL. Ceci est nécessaire dans un projet comme celui-ci pour ne pas se perdre dans les centaines de signaux déclarés. Nous avons aussi pour cela décidé de bien commenter notre code.

Voici la nomenclature qui est utilisée dans les codes VHDL :

- Toutes les **constantes** sont en MAJUSCULE et finissent par « **_c** »
- Toutes les **entrées** finissent par « **_i** » pour « input »
- Toutes les **sorties** finissent par « **_o** » pour « output »
- Tous les **signaux** finissent par « **_s** »
- Tous les signaux concernant le **joueur 0** (Nestor) contiennent « **p0** »
- Tous les signaux concernant le **joueur 1** (Grudu) contiennent « **p1** »
- Tous les signaux concernant une **balle** x contiennent « **bx** », x étant le numéro de la balle
- Les constantes sont placées dans le package « local »

5.1 Contrôleur du jeu

Le contrôleur de jeu étant une machine de Moore, il est implémenté en trois process :

- Le combinatoire d'entrée
- La bascule
- L'assignation des sorties

5.2 Lecture dans des ROM

Certains composants, comme « player », « bullet » ou encore « graphic » doivent lire dans des ROM.
L'utilisation de ces ROM est un peu différente des LUTs qui étaient implémentées au début.
Pour lire une donnée dans une ROM, il faut lui donner une adresse, et elle ressort une donnée. Alors qu'avec les LUTs utilisées au début du projet, on donnait deux adresses (en X et en Y), et elle sortait une donnée. Il a donc fallu modifier la façon de procéder :

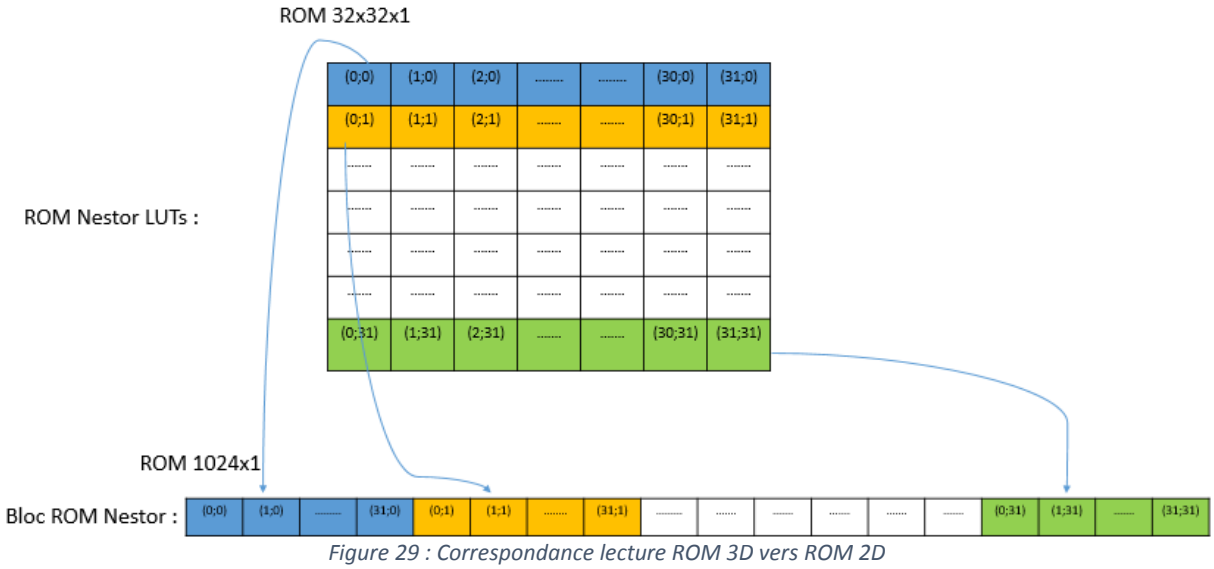


Figure 29 : Correspondance lecture ROM 3D vers ROM 2D

L'exemple de code suivant permet de lire dans ces ROM :

```
-- calcul de l'adresse pour lire dans la RAM du personnage
nestor_addr_temp_s <= (hcount_s - posX_p0_s) + ((vcount_s - posY_p0_s) * 32) ;
```

5.3 Joueurs

La gestion des joueurs est faite dans les composants « player0 » et « player1 ». Les codes sont les mêmes pour les deux joueurs, mais nous n'avons pas fait deux instances d'un seul composant, pour permettre d'assigner des caractéristiques propres à chaque joueur.

Le composant est séparé en 4 process :

- read_collision_map :

Calcul l'adresse pour lire dans la carte des collisions, en fonction de la direction du déplacement et de la position du joueur.

- compteur_deplacement :

Compteur qui, à chaque overflow, déplace le joueur dans la direction demandée. Si le résultat de la lecture dans la carte des collisions est « 0 », alors le personnage se déplace, sinon il s'arrête. Il faut également contrôler que le joueur ne soit pas au bord de la carte.

- update_orient_p :

Bascule qui, en fonction de l'entrée des boutons, met à jour le signal de l'orientation du joueur. Cette bascule a été faite, car lorsqu'aucun bouton n'est pressé, il faut mémoriser le dernier état.

- health_points :

Process qui, dans une bascule, décrémente les points de vie du joueur lorsqu'il est touché.

5.4 Balles

5.4.1 Bullet_ctrl

Le monoflop est une machine d'état de Moore. Il est donc composé d'un combinatoire d'entée, d'un registre, et de l'assignation des sorties.

5.4.2 Bullet

Le composant « bullet » est composé de trois process.

- read_collision_map :

Comme pour le composant du « player », ce process calcul l'adresse pour lire dans la carte des collisions, en fonction de la direction du déplacement et de la position de la balle

- compteur_deplacement :

Lorsqu'il y a une impulsion de tir, déplace la balle dans la bonne orientation à chaque overflow d'un compteur. Check si la balle touche un mur, le bord de la carte, ou le joueur adverse. Si c'est le cas, la balle n'est plus tirée.

- update_pos_depart_bullet :

Met à jour la position de départ d'une balle en fonction de la position et l'orientation du joueur.

5.5 Graphique

L'implémentation des graphiques se fait essentiellement dans le composant « graphic ».

5.5.1 Carte

Carte de l'agencement des murs et du sol en 20x15 pixels :

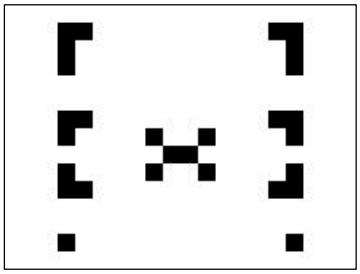


Figure 30 : Carte de l'agencement du sol et des murs

Chaque pixel de la carte représente un mur s'il est noir, ou le sol s'il est blanc.
La carte est parcourue 40x moins vite que les sprites du mur et du sol. Ainsi, lorsqu'un pixel est lu blanc, il faut afficher le sol, et lorsqu'il est noir le mur.
Cette façon de faire permet de modifier assez simplement l'organisation de la carte. Pour cela, il faut juste modifier cette carte et la carte des collisions (même carte mais en 200x150).

5.5.2 Ecran de fin

L'écran de fin est produit à l'aide de deux images. Un compteur permet de faire un clignotement de type « gif ». Voici le résultat :

6 RÉSUMÉ DES RESSOURCES UTILISÉES

TOP_LEVEL Project Status (06/22/2016 - 21:28:13)			
Project File:	ISE_NEStor.xise	**Parser Errors:**	No Errors
Module Name:	TOP_LEVEL	**Implementation State:**	Synthesized
Target Device:	xc6slx16-2csg324	**• Errors:**	No Errors
Product Version:	ISE 14.7	**• Warnings:**	267 Warnings (0 new)
Design Goal:	Balanced	**• Routing Results:**	
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	
Environment:	System Settings	**• Final Timing Score:**	

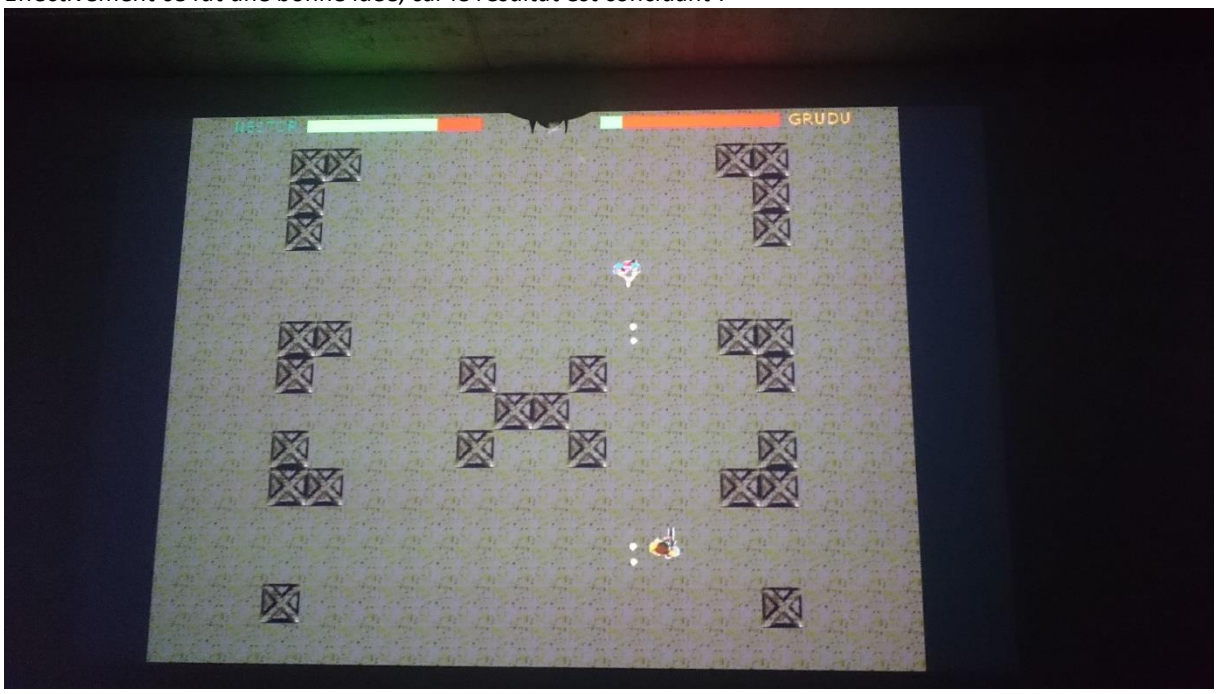
Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	646	18224	3%
Number of Slice LUTs	2884	9112	31%
Number of fully used LUT-FF pairs	532	2998	17%
Number of bonded IOBs	21	232	9%
Number of Block RAM/FIFO	27	32	84%
Number of BUFG/BUFGCTRLs	8	16	50%
Number of DSP48A1s	21	32	65%
Number of PLL_ADVs	1	2	50%

Figure 31 : Résumé des ressources utilisée

7 TESTS ET VALIDATION

Pour les tests nous avons principalement testé sur l'écran. Ceci est dû au fait que ni une simulation ou un testbench permet de vraiment se rendre compte de toutes les erreurs graphiques. Il est aussi à noter que les compteurs vont tellement à des grandes valeurs qu'une simulation est très longues. Nous avons donc décidé de simuler juste les composants séparément.

Effectivement ce fut une bonne idée, car le résultat est concluant :



8 PROBLÈMES RENCONTRÉS

1)

Un premier problème rencontré a été que lorsqu'un signal « integer » est déclaré, il est de base initialisée à 0. Si lors du reset, le signal en question est assigné avec une autre valeur que 0, le synthétiseur doit utiliser des

ressources supplémentaires pour implémenter le registre. Il peut même, comme c'était notre cas, donner une erreur et ne pas synthétiser.
Le screenshot suivant nous a permis de découvrir la cause de cette erreur :

```
signal reg: integer range 0 to 7;
...
process (clk, rst)
begin
   if rst='1' then
       reg <= 7;
   elsif (clk'event and clk='1') then
       reg <= reg + 1;
   end if;
end process;
```

In the above example the default value of the reg signal it is equal to 0. The fact that the reg register is reset to 7 and initialized to 0 requires the synthesis tool to use additional FPGA resources for its implementation.

If an initialized value of 0 is not necessary for functionality, the signal reg should be manually initialized to 7 to match the reset value specified in the associated process. If it is determined that an initialized value of 0 is needed, then it is suggested to change the asynchronous rst signal to be described synchronously. This allows to avoid using latches for its implementation.

Figure 32 : résolution 1er problème rencontré

Solution : Il faut initialiser le signal à sa déclaration à la même valeur que celle voulue lors du reset :

```
signal reg : integer range 0 to 7 := 7 ;
```

2)
Les balles ne partaient pas toujours du bon endroit du personnage. Nous avons donc demandé conseil à M. Cunha, qui nous a proposé de constamment mettre à jour la position de départ de la balle, ce qui fût concluant.

3)
Lorsque l'implémentation du bus SPI pour les joysticks a été faite, une clock de 100 [MHz] était nécessaire. La clock d'entrée de la FPGA a donc été utilisée, mais le synthétiseur donnait l'erreur suivante :
ERROR:Xst:2035 - Port has illegal connections. This port is connected to an input buffer and other components.

Après plusieurs recherches, nous avons vu qu'il n'est en fait pas possible de réutiliser la clock qui rentre dans une DCM. Il a donc fallu sortir une fréquence de 100 [MHz] de la DCM afin de régler le problème.

9 AMÉLIORATIONS FUTURES

Beaucoup d'améliorations futures sont possibles pour ce projet. Certaines seraient nécessaires pour une meilleure lecture du code, comme :

- Une meilleure séparation du composant graphique. En effet, le composant est beaucoup trop grand. Il serait donc judicieux de créer des composants qui se chargent du calcul de l'adresse de la ROM. Le composant « graphic » contiendrait alors uniquement le process « draw », qui se charge de dessiner sur l'écran.
- Optimisation du composant TOP_LEVEL, avec des « for generate ». Le composant TOP_LEVEL fait environ 1000 lignes. Il est donc très difficile de s'y retrouver dans ce gros composant. Le fait d'utiliser des « for generate » pour créer les six balles, ainsi que les multiples ROMs améliorerait grandement le code.

Nous avons également vu que la carte Nexys3 dispose de beaucoup de mémoire RAM (3 * 16 MBytes), externe à la FPGA. Cette RAM pourrait être utilisée pour stocker des cartes, ce qui permettrai d'avoir des graphiques plus jolis, et d'en stocker plusieurs. Ainsi il serait possible, par exemple, de sélectionner une carte, ou un joueur au lancement du jeux.

10 CONCLUSION

Finalement ce projet nous a beaucoup plus, car nous avions les bases pour pouvoir réaliser un tel projet, mais il a quand même fallu beaucoup travailler afin de trouver comment les choses sont faites.
Nous avons eu du plaisir d'avoir une marge de manœuvre complète, afin de réaliser un projet à notre image. Le fait de se lancer dans un projet tel que celui-ci nous a aussi permis de nous confronter aux problèmes réels du monde du VHDL.

Nous avons pu nous rendre compte que les FPGAs sont limités en taille et que, contrairement à la programmation de haut niveau, la description matérielle dépend beaucoup de la qualité du code.

Une chose qui nous a aussi frappés, est la complexité de programmes capricieux comme ISE, qui, malgré que ce soient des outils très puissants, donnent souvent des erreurs très peu claires. Nous avons souvent été confronté à des problèmes liés aux outils plus qu'au code.

Beaucoup de changement de stratégie à cause de la place dans la FPGA

- ⇨ Permis de se rendre compte que contrairement à de la programmation sur CPU, la programmation sur FPGA nécessite de savoir ce qui est fait derrière le code, ainsi qu'une bonne connaissance de la FPGA sur laquelle nous travaillons

11 FIGURES

12 SOURCES

Les sources qui ont été utilisées durant le projet sont les suivantes :

- LBE Books : http://www.lbebooks.com/
- Vidéos LBE Books : https://www.youtube.com/user/LBEbooks
- Exemple de projet : http://static.armandas.lt/res/fpga_based_vga_driver_and_arcade_game.pdf
- Exemple de projet : http://web.mit.edu/6.111/www/s2004/PROJECTS/2/index.htm
- Datasheet de la Nexys 3 : http://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPNexys3/documentation/Nexys3_rm.pdf
- UCF de la Nexys 3 : http://www.xilinx.com/support/documentation/university/ISE-Teaching/HDL-Design/14x/Nexys3/Supporting%20Materials/Nexys3_Master.ucf

13 ANNEXES

13.1 Code python jpg_to_binary

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-


"""

jpg to binary array.py: This script convert any image into a 8bit RBG format displayed in hex
(for exemple : 0xff)
The output is to be declared as an array in VHDL in order to send it to a VGA output

Standard usage:       python3 jpg_to_binary_array.py image.jpg
With image scaling:   python3 jpg to binary array.py image.jpg 32 32

The output file is named <yourimagename>.txt

"""

__author__  ="Bruno Produit"


import sys
import PIL
from PIL import Image

im = Image.open(sys.argv[1])#Can be in different formats

# rescale if asked
if len(sys.argv)>2:
    size =(int(sys.argv[2]), int(sys.argv[3]))
    im = im .resize(size, Image.ANTIALIAS)

# load image and create output file with 4 last characters stripped
output = open("".join([sys.argv[1][:-4],".txt"]),'w')
pix = im.load()

#print (im.size) #Get the width and hight of the image for iterating over
#print (pix[5,5]) #Get the RGBA Value of the a pixel of an image

output.write("type memory_m is array(0 to ")
output.write(str(im.size[0]-1))
output.write(", 0 to ")
output.write(str(im.size[1]-1))
output.write(") of std_logic_vector(7 downto 0);\nconstant mem : memory_m :=\n(")
output.write("(")

# iterate over image
for i in range(0, im.size[0]):
if(i !=0):
     output.write("), \n(")
for j in range(0, im.size[1]):

# take out and convert each pixel to it's binary value
      Blue =  bin(round(pix[i,j][0]>>5))[2:].zfill(3)# bit shift of 5, stripped "0b", do not
stip zeros
      Green = bin(round(pix[i,j][1]>>5))[2:].zfill(3)# bit shift of 5, stripped "0b", do not
stip zeros
      Red =   bin(round(pix[i,j][2]>>6))[2:].zfill(2)# bit shift of 6 (only 2 bits for blue),
stripped "0b", do not stip zeros
      data ='{:02x}'.format(int(''.join([Blue, Green, Red]),2))# concatenate and convert to
hex
      output.write("x\"")
      output.write(data)
      output.write("\"")
if(j != im.size[1]-1):
         output.write(", ")

output.write(")")
output.write(");")
```

13.2 Code matlab IMG2coe8

```
function img2 = IMG2coe8(imgfile, outfile)
% Create .coe file from .bmp image
% .coe file contains 8-bit words (bytes)
% each byte contains one 8-bit pixel
% color byte: [R2,R1,R0,G2,G1,G0,B1,B0]
% img2 = IMG2coe8(imgfile, outfile)
% img2 is 256-bit color image
% imgfile = input .bmp file
% outfile = output .coe file
% Example:
% img2 = IMG2coe8('REH45x226.bmp', 'REH8.coe');

img = imread(imgfile);
height = size(img,1);
width = size(img,2);

s = fopen(outfile,'wb');%opens the output file

fprintf(s,'%s\n','; VGA Memory Map ');
fprintf(s,'%s\n','; .COE file with hex coefficients ');
fprintf(s,'; Height: %d, Width: %d\n\n', height, width);

fprintf(s,'%s\n','memory_initialization_radix=16;');
fprintf(s,'%s\n','memory_initialization_vector=');

cnt =0;

img2 = img;

for r=1:height
for c=1:width
        cnt = cnt +1;
        R = img(r,c,1);
        G = img(r,c,2);
        B = img(r,c,3);
        Rb = dec2bin(double(R),8);
        Gb = dec2bin(double(G),8);
        Bb = dec2bin(double(B),8);
        img2(r,c,1) = bin2dec([Rb(1:3) '00000']);
        img2(r,c,2) = bin2dec([Gb(1:3) '00000']);
        img2(r,c,3) = bin2dec([Bb(1:2) '000000']);
        Outbyte = [ Rb(1:3) Gb(1:3) Bb(1:2) ];
        if (Outbyte(1:4) == '0000')
            fprintf(s,'0%X',bin2dec(Outbyte));
        else
            fprintf(s,'%X',bin2dec(Outbyte));
        end
        if ((c == width) && (r == height))
            fprintf(s,'%c',';');
        else
            if (mod(cnt,32) == 0)
                fprintf(s,'%c\n',',');
            else
                    fprintf(s,'%c',',');
            end
        end
    end
end

fclose(s);
```