Space Invaders

Projet intégré - Système numérique

Mai
11
2017

Samuel Riedo & Pascal Roulin

1

Introduction

Space Invaders est un jeu vidéo d'arcade créé par Tomohiro Nishikado, paru pour la première fois en 1978 au Japon. Il est l'un des tout premier *Shoot 'em up*, c'est-à-dire un type de jeux consistant à abatre un grand nombre d'ennemies en leur tirant dessus. Le principe du jeu consiste en un vasseau spacial attaqué par des vagues d'aliens qu'il doit détruire en leur tirant dessus sans se faire toucher par les tirs des aliens.

Space Invaders connu rapidement un succès mondial et est aujourd'hui considéré comme un grand classique de l'univers vidéoludique. Il a de ce fait connu de nombreux ports et suites sur un grand nombre de plate-forme, vieille comme récente.

1.1 Super Mario Bros

Dans un premier temps, nous avions souhaité reproduire *Super Mario Bros*. La première tentative pour recréer le monde 1-1 du jeu original fut de créer toute la map en une image, puis de la stocker dans une RAM ou ROM. Un scalling de 4 permet de drastiquement réduire le nombre de pixels à stocker, en passant de 6400x800 pour l'image d'origine à 1600x150 pour celle que nous utiliserons. Ceci représentait 240 000 pixels à stocker. En prenant en compte qu'un pixel fait exactement un Byte (deux bits pour la composante bleu, trois pour la rouge et trois pour la verte), les RAM et ROM à disposition de la Spartan 6 XC6LX16-CS324 ne pouvaient pas stocker toutes ses données.

Afin de contourner ce problème, l'image de base a été diviser en 8 images plus petites, faisant chacune 200x150 pixels, soit 30kB. Il était alors possible de stocker une image dans une ROM/RAM, puis une deuxième dans une seconde ROM/RAM, mais il était à nouveau impossible d'enregistrer les six suivantes sans dépasser les capacités de la carte.

Devant ses limitations hardware, la décision fu prise de changer de jeu. Il nous est apparu qu'avoir un font statique, ou alors une répétition permannente d'un même arrière plan était indispensable pour que le projet soit synthétisable sur notre carte. Un jeu tel que *Super Mario Bros*, avec des mondes très différents et non répétifs, n'est pas adapté à la programmation VHDL sur un hardware limité. Notre choix s'est alors porté sur *Space Invaders*.

1.2 Gameplay

Space Invaders est un jeu en deux dimension, aussi appelé jeu en 2D ou tout simplement jeu 2D. Le joueur contrôle un vaisseau spatial pouvant se déplacer uniquement sur l'axe X, et tirer des laser vers le haut de l'écran. Il est confronté à plusieurs aliens, se déplaçant aléatoirement dans la partie supérieur de l'écran. Ces derniers tire aléatoirement des lasers vers le bat pour détruire le vaisseau spacial contrôlé par le joueur.

Si le vaisseau du joueur se fait toucher par un laser alien, la partie est perdue. Si, au contraire, le joueur réussit à détruire tous les aliens sans se faire lui-même toucher, il gagne la partie. La figure ci-dessus représente une partie typique de *Space Invaders* sur borne arcade tel que le jeu était lors de son lancement initial en 1978.

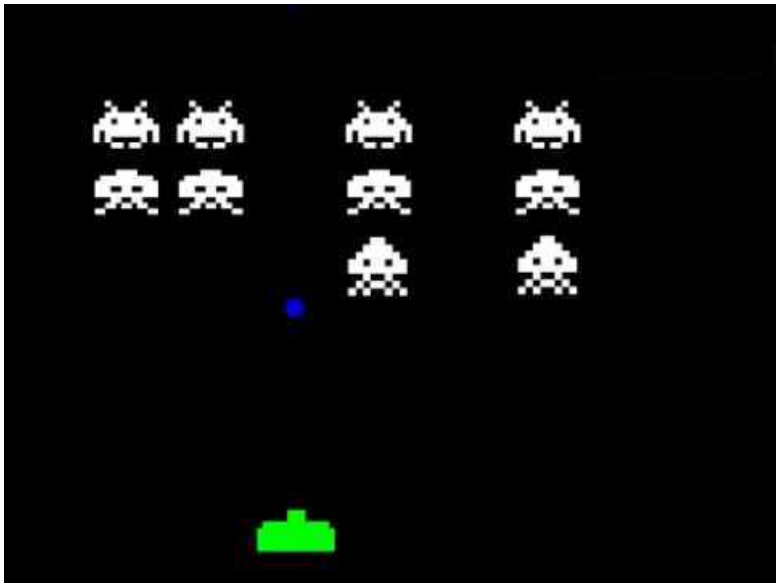


Fig. 1.1 : Space Invaders sur borne arcade

Le vaisseau du joueur est représenté par la forme verte. Ce dernier à tiré un laser, symbolisé par un poinds bleu. Les aliens, au nombre de 10, ont déjà été partiellement déssimer. Au début d'une partie, leur nombre et leur disposition forme une grille rectangulaire complète.

Types d'aliens

Bien que les aliens peuvent avoir plusieurs formes différents (trois sur la figure 1.1), celà n'influence en rien leur comportement. Il ne s'agit ni plus ni moins que d'un skin.

Les version suivantes de space invaders implémenteront de nouvelles fonctionnalités, tel que :

- Score, déterminer par les aliens détruits et le temps pour y arriver.
- Vaisseau alien traversant l'écran horizontalement de façon aléatoire. Le détruire rapport des points bonus.
- Bouclier pour protéger le vaisseaux.

2

Architecture

La réalisation du projet fut divisé en 6 principaux blocs, plus un top module et un package. Trois de ces blocs furent repris du travail pratique concernant l'affichage par VGA, alors que les autres ont été spécialement implémentés pour ce projet.

Le fait d'avoir déjà une base de départ nous a poussé à implémentés le jeu fonction par fonction, puis de tester et debugguer chaque nouvel ajout dès qu'il fut coder. Cette approche présente l'avantage, contrairement à un developpement de chaque composants indépendamment les uns des autres, de réduire le risque d'incompatibilité entre deux composants à fin ainsi que le lourd travaille de debuggage final. En revanche, cette technique ne permet pas de répartir efficassement le travail dans une équipe constituée de nombreuse personnes.

Bien que nous puissons tester le bon fonctionnement de chaque blocs et fonctions directement en programmant la FPGA pour voir le résultat sur l'écran, chaque bloc sera testé par une macro pour valider tous les cas pouvant intervenir dans le déroulement d'une partie. De plus, deux composants, *Input* et *rocketManager*, seront testé via un testbench.

2.1 alienRocket

Le bloc *alienRocket* gère les roquettes, aussi appelé laser ou missile, tirés par les aliens en direction du spationef. Il est chargé de générer de nouveaux missiles ainsi que de transmettre les informatins nécessaire au bloc *Display* pour afficher correctement une roquette à l'écran. Dans notre implémentation du jeu, le joueur fait face à 50 aliens, réparties en une grille de 5 lignes et 10 colonnes (tableau 2.1). Par colonne, chaque aliens le plus proche du bat de l'écran peut tirer une roquette vers le bat pour tenter de détruire le vaisseau du joueur. Une seule roquette peut être affiché à l'écran en même temps (sans compter les tirs du joueur). Cela signifie que tant que le missile n'a pas atteint le bat de l'écran, les aliens ne peuvent pas en tirer un nouveau.

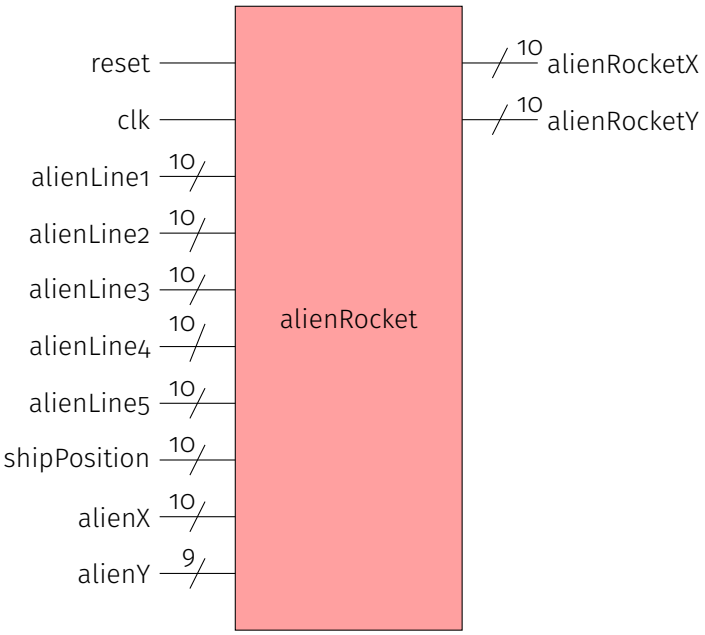
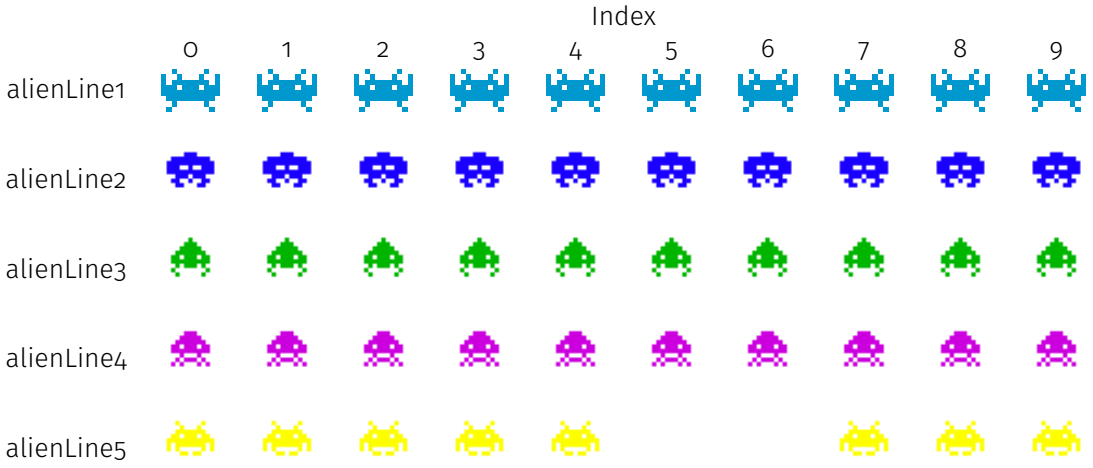


Fig. 2.1 : Schéma bloc

Lorsqu'une roquette peut être tiré, le choix de la colonne d'alien pouvant tirer se fait de manière aléatoire entre toutes les colonnes qui contiennes au moi un alien. Par exemple, soit les aliens encore en vie selon le tableau 2.1. Une roquette peut être tiré depuis les aliens 0-4 et 7-9 de la ligne 5 (alienLine5), ainsi que depuis les alien 5 et 6 de la ligne 4 (alienLine4).



Tab. 2.1 : Gestion des aliens dans le jeux

2.1.1 Entrées & Sorties

reset Reset du circuit, actif à l'état haut.

clk Horloge 40MHz, active sur front montant.

alienLine1 Indique, par un 0 un alien vivant, et par 1 un alien mort dans la ligne d'alien de la partie supérieur de l'écran.

alienLine2 Identique à alienLine1, pour la ligne d'alien en dessous.

alienLine3 Identique à alienLine2, pour la ligne d'alien en dessous.

alienLine4 Identique à alienLine3, pour la ligne d'alien en dessous.

alienLine5 Identique à alienLine4, pour la ligne d'alien en dessous.

shipPosition Nombre de pixels entre le bord gauche de l'écran et le bord droit du vaisseau de joueur. Cette valeur est utilisé pour générer de l'aléatoire.

alienX Nombre de pixels entre le bord gauche de l'écran et le board droit des aliens à l'index 0.

alienY Nombre de pixels entre le haut de l'écran et le bord supérieur des aliens contenues dans *alienLine1*.

alienRocketX Nombre de pixels entre le bord gauche de l'écran et la rocket tirée par les aliens.

alienRocketY Nombre de pixels entre le le haut de l'écran et le haut de la rocket tirée par les aliens.

2.2 Digital Clock Management

La norme VGA utilise une fréquence de 40MHz pour le balayage de l'écran. Or, l'horloge intégrée à notre carte dispose d'une fréquence de 100MHz. Le bloc DCM crée une horloge de 40MHz grâce à une horloge d'entrée de 100MHz. Ce type de montage étant très courant, il existe des outils, appelé IP Core, pour générer un composant selon nos besoins.

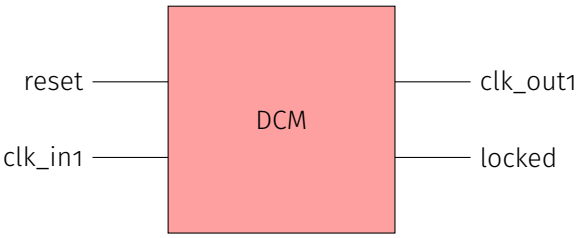


Fig. 2.2 : Schéma bloc

2.2.1 Entrées & Sorties

reset Reset du circuit, actif à l'état haut.

clk_in1 Horloge 100MHz, active sur front montant.

clk_out1 Horloge 40MHz.

locked Sortie non utilisée.

2.3 Display

Grâce aux signaux généraux par les composants *VGA_Internal* et *DCM*, *Display* est en mesure d'afficher des données à l'écran au moyen des trois sorties *red*, *green* et *blue*. Ses dernières sont codés sur trois bits, à l'exception de la composante bleu qui n'est uniquement constituée de deux bits.

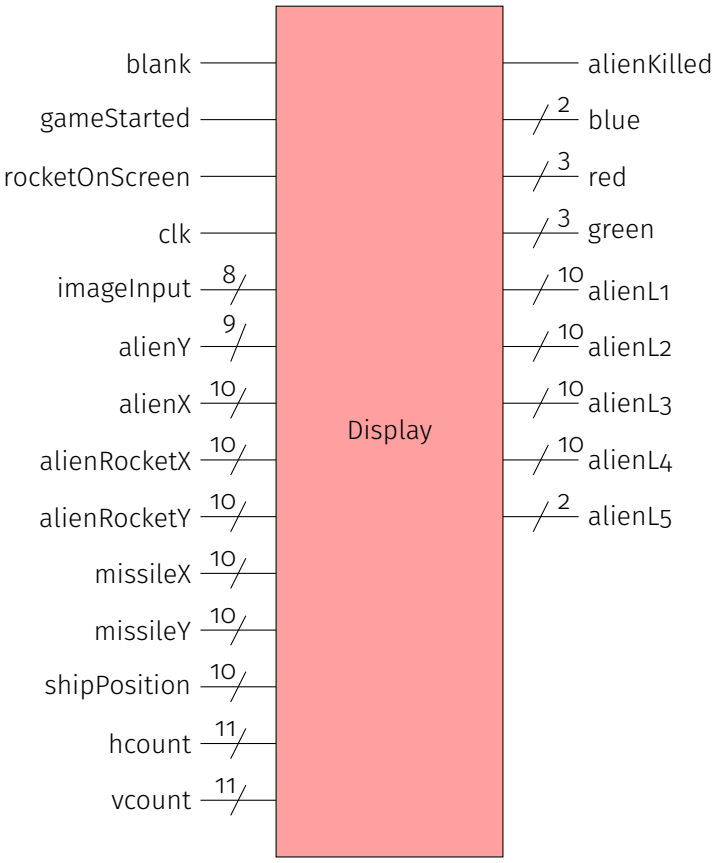


Fig. 2.3 : Schéma bloc

2.3.1 Entrées & Sorties

blank Si 1, le balayage est en dehors de l'écran et les composantes RGB, c'est-à-dire les sorties *red*, *blue* et *green* doivent être null.

gameStarted Indique par une valeur à 1 que le jeu à débuté.

rocketOnScreen Indique par une valeur à 1 qu'une rocket doit être affiché à l'écran.

clk Horloge 40MHz, active sur front montant.

imageInput Bus de données en provenance de la ROM contenant l'image d'accueil du jeu.

alienY Nombre de pixels entre le haut de l'écran et le bord supérieur des aliens contenues dans *alienLine1*.

alienX Nombre de pixels entre le bord gauche de l'écran et le board droit des aliens à l'index 0.

alienRocketX Nombre de pixels entre le bord gauche de l'écran et la rocket tirée par les aliens.

alienRocketY Nombre de pixels entre le le haut de l'écran et le haut de la rocket tirée par les aliens.

missileX Nombre de pixels entre le bord gauche de l'écran et la rocket lancée par les aliens.

missileY Nombre de pixels entre le haut de l'écran et le haut de la rocket lancée par les aliens.

shipPosition Nombre de pixels entre le bord gauche de l'écran et le bord gauche du vaisseau contrôlé par le joueur.

hcount Coordonnée X du balayage.

vcount Coordonnée Y du balayage.

alienKilled Indique par une valeur à 1 qu'un alien à été touché par une rocket lancée par le joueur.

blue Composante bleu de la sortie VGA.

red Composante rouge de la sortie VGA.

green Composante verte de la sortie VGA.

alienL1 Indique par une valeur à 1 la présence d'un alien au même index dans la rangée d'aliens la plus proche du haut de l'écran (voir tableau 2.1).

alienL2 Identique à *alienL1* pour la rangée d'aliens inférieur.

alienL3 Identique à *alienL2* pour la rangée d'aliens inférieur.

alienL4 Identique à *alienL3* pour la rangée d'aliens inférieur.

alienL5 Identique à *alienL4* pour la rangée d'aliens inférieur.

2.4 Input

Input est chargé de recevoir et traiter les actions faites par le joueur. Via quatre des cinq boutons de la croix directionnelle, le joueur peut démarrer une partie, déplacer son vaisseau et tirer. Toutes ses actions sont traiter dans *Input*, tout comme le déplacement aléatoire des aliens sur l'écran. Ses derniers peuvent se mouvoir en haut, bas, gauche, droite ainsi qu'en diagonale. La direction qu'ils prennent est aléatoire, en fonction d'un compteur *alienDirection* s'incrémentant à chaque coût d'horloge. De plus, la distance, ou saut, qu'ils parcourent en un déplacement est également aléatoire via un autre compteur *alienJump*. La taille maximal d'un saut peut être définie en modifiant la valeur *maxAlienJump* dans le package du jeu.

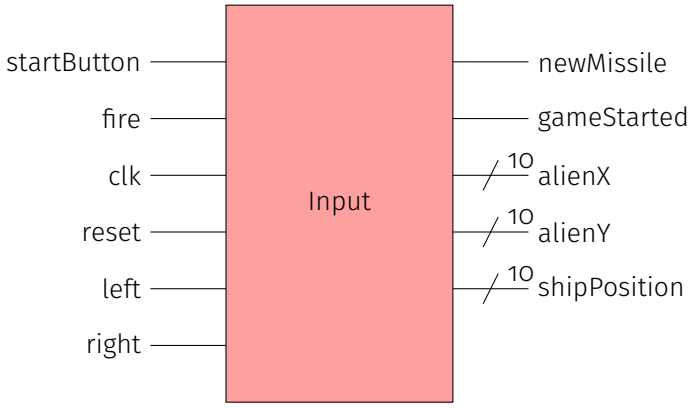


Fig. 2.4 : Schéma bloc

```
signal alienDirection : integer range 0 to 7:= 0; -- If 0, aliens move left, 1=up left, 2 = up, ...
signal alienJump      : integer range 1 to maxAlienJump := 1; -- Pixels number alien use as unit to move
process(reset, clk)
begin
  if reset = '1' then
    alienDirection <= 0;
    alienJump      <= 1;
  elsif rising_edge(clk) then
    -- alien direction
    if alienDirection >= 7 then
      alienDirection <= 0;
    else
      alienDirection <= alienDirection + 1;
    end if;
    -- alien jump
    if alienJump >= maxAlienJump then
      alienJump <= 1;
    else
      alienJump <= alienJump + 1;
    end if;
  end if;
end process;
```

2.4.1 Entrées & Sorties

startButton Bouton situé au centre de la croix directionnelle (B8). Utilisé pour démarrer une partie depuis l'écran d'accueil.

fire Bouton supérieur de la croix directionnelle (BTNU, A8). Utilisé pour tirer des lasers depuis le vaisseau vers les aliens pendant une partie.

clk Horloge 40MHz, active sur front montant.

reset Reset du circuit, actif à l'état haut.

left Bouton gauche de la croix directionnelle (BTNL, C4). Utilisé pour déplacer le vaisseau du joueur vers la gauche.

right Bouton right de la croix directionnelle (BTNR, D9). Utilisé pour déplacer le vaisseau du joueur vers la droite.

newMissile Indique par une valeur à 1 qu'un nouveau missile a été tiré par le vaisseau du joueur.

gameStarted Indique par une valeur à 1 que le jeu à débuté.

alienX Nombre de pixels entre le bord gauche de l'écran et le board droit des aliens à l'index 0.

alienY Nombre de pixels entre le haut de l'écran et le bord supérieur des aliens contenues dans *alienLine1*.

shipPosition Nombre de pixels entre le bord gauche de l'écran et le bord droit du vaisseau de joueur. Cette valeur est utilisé pour générer de l'aléatoire.

2.5 rocketManager

Le bloc *rocketManager* a pour rôle de générer les missiles du vaisseau spatial que le joueur pilote. Le bloc *Input* fourni à *rocketManager* une impulsion, via le signal *newMissile*, lorsque celui-ci doit générer un nouveau missile. Cette impulsion apparaît lorsque le jour clique sur le bouton pour tirer un nouveau missile. Dès qu'un missile est tiré (*fire*), celui-ci doit avoir une coordonnée en X, une coordonnée en Y ainsi qu'être signaler au bloc *Display* afin que celui-ci sache qu'un missile doit être affiché.

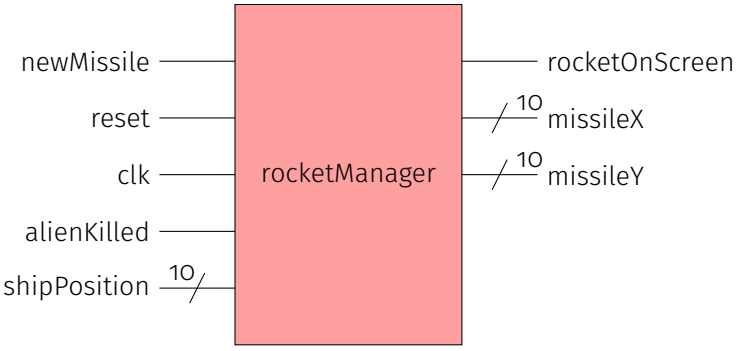


Fig. 2.5 : Schéma bloc

L'entrée *alienKilled* permet au bloc de savoir si un alien a été touché, et dans ce cas le signal de sortie *rocketOnScreen* passera à 0 et le missile n'apparaîtra plus à l'écran. Ce signal passe également à 0 lorsque le missile atteint le haut de l'écran.

RocketManager utilise le signal *shipPosition*, qui correspond à la coordonnée sur l'axe horizontale du vaisseau au moment du tire, afin de fournir en sortie (*missileX*) la position X du missile. Cette position reste la même tant que le missile n'a pas atteint un alien ou le haut de l'écran.

Concernant la position Y du missile, celle-ci se comporte comme un compteur. En effet, cette position est remise à la valeur de 570 (hauteur de l'écran moins la hauteur du vaisseau), afin que le missile parte du vaisseau, et est ensuite décrémenté jusqu'à la valeur minimum de 0 (le missile atteint donc le haut de l'écran).

Les positions X et Y sont fouris au bloc *Display* via leur signaux respectifs et ce dernier se charge d'afficher le missile.

Le bloc *rocketManager* utilise la clock afin de décrémenter le compteur de la position Y et le signal reset afin de remettre tous les signaux dans leur état d'origine.

2.5.1 Entrées & Sorties

newMissile Indique par une valeur à 1 qu'un nouveau missile a été tiré par le vaisseau du joueur.

reset Reset du circuit, actif à l'état haut.

clk Horloge 40MHz, active sur front montant.

alienKilled Indique par une valeur à 1 qu'un alien à été touché par une rocket lancée par le joueur.

shipPosition Nombre de pixels entre le bord gauche de l'écran et le bord droit du vaisseau de joueur. Cette valeur est utilisé pour générer de l'aléatoire.

rocketOnScreen Indique par une valeur à 1 qu'une rocket doit être affiché à l'écran.

missileX Nombre de pixels entre le bord gauche de l'écran et la rocket lancée par les aliens.

missileY Nombre de pixels entre le haut de l'écran et le haut de la rocket lancée par les aliens.

2.6 VGA Internal

Une interface VGA fonctionne selon les trois composantes RGB ainsi qu'une synchronisation horizontale et verticale. Les signaux RGB décrivent la couleur des pixels composant l'image selon un balayage effectué de gauche à droite, en ligne de haut en bas. L'écran recevant ce flux RGB est capable de savoir à quel pixel il correspond selon l'instant t auquel il lit ses données dans le balayage. Néanmoins, ce n'est pas l'écran qui est chargé de sauter automatiquement à la ligne suivante lorsque chaque pixel de la ligne actuel a été traité. C'est ce à quoi sert le signal *HS*, alors que *VS* indique un retour à la première ligne.

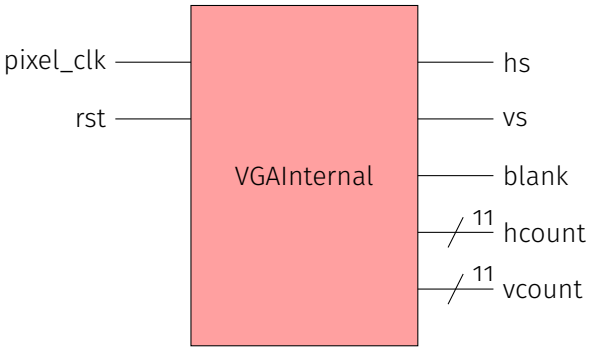


Fig. 2.6 : Schéma bloc

Retour à la ligne

Afin d'informer le monitor que le balayage est arrivé à la fin d'une ligne et que le prochain pixel sera le premier de la ligne suivante, le signal HS produit une impulsion de synchronisation. De même, lorsque le balayage est arrivé à la fin de la dernière ligne (et donc que toute une image a été transmise), le signal VS produit une impulsion pour indiquer un retour à la première ligne (et donc la transmission d'une nouvelle image).

Lorsque le balayage se trouve en dehors de l'écran, le signal *blank* prend comme valeur 0 afin d'indiquer au bloc *Display* de mettre les composantes de sorties RGB à 0. Ce comportement est définit dans la norme VGA et résulte dans une erreur d'affichage "Index out of bound" s'il n'est pas respecté.

2.6.1 Entrées & Sorties

pixel_clk Horloge 40MHz, active sur front montant.

rst Reset du circuit, actif à l'état haut.

hs Impulsion de synchronisation horizontale. Indique par une pulse à l'état haut un retour à la ligne du balayage de l'écran.

vs Impulsion de synchronisation verticale. Indique par une pulse à l'état haut un retour du balayage à la première ligne de l'écran.

blank Si 1, le balayage est en dehors de l'écran et les composantes RGB, c'est-à-dire les sorties *red*, *blue* et *green* doivent être null.

hcount Coordonnée X du balayage.

vcount Coordonnée Y du balayage.

2.7 Top Module

Le bloc *topModule* est comme son nom l'indique l'élément tout en haut de notre architecture. C'est lui qui instancie les composants et relie ceux-ci via des signaux intermédiaires.

Concernant ses entrées et sorties, celles-ci sont des éléments physiques de la carte FPGA et sont assignées via le fichier UCF.

Le signal d'entrée *fpga_clk* correspond à la clock de la FPGA. Cette clock fonctionne à une fréquence de 100 MHz. Le *reset* est assigné sur l'un des switchs (voir figure XXXXXXX) et permet le reset software du jeu.

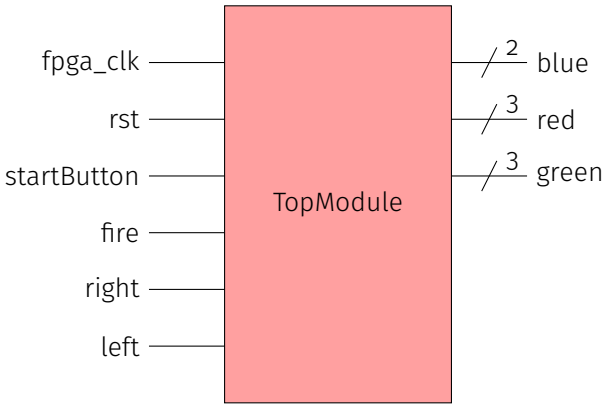


Fig. 2.7 : Schéma bloc

Plusieurs boutons sont utilisés afin de jouer et ont donc été assignés grâce au fichier UCF. Au niveau des signaux, il s'agit de *startButton*, de *fire*, de *right* et de *left*. Le bouton *fire* permet de tirer un missile depuis le vaisseau. Puisqu'il n'est possible que de tirer un seul missile à la fois, celui-ci n'est pas d'effet tant qu'il y a un missile en déplacement. Les boutons *right* et *left* sont explicites. Ils permettent le déplacement latéral du vaisseau que pilote le joueur. Le *startButton* permet de passer de l'écran de démarrage au jeu. Il n'a d'effet que durant cette étape afin qu'en cas d'appui involontaire durant la partie celle-ci ne s'en retrouve pas impactée.

Nous avions commencer par utiliser un seul bouton afin de passer de l'écran de démarrage au jeu et pour tirer des missiles. Cependant, lorsque le jeu commençait, un missile était automatiquement tiré et cela ne correspondait pas à ce que nous voulions. Nous avons chercher un moyen de corriger ce problème mais cela s'est avéré compliqué pour un problème mineur alors qu'une solution simple, et finalement plus pratique, était de séparer ces deux fonctions en deux boutons.

Concernant les sorties *blue*, *red* et *green*, celles-ci sont assignées sur la sortie VGA de la carte. Ces signaux correspondent aux trois couleurs (RGB) utilisé par le VGA afin d'afficher sur un écran. Puisque le VGA utilise 8 bits, les signaux *red* et *green* disposent de 3 bits chacun. Le signal *blue* ne dispose que de 2 bits. Il est tout de même possible de d'afficher 256 couleurs différentes.

2.7.1 Entrées & Sorties

fpga_clk Horloge 100MHz, active sur front montant.

rst Reset du circuit, actif à l'état haut.

startButton Bouton situé au centre de la croix directionnelle (B8). Utilisé pour démarrer une partie depuis l'écran d'accueil.

fire Bouton supérieur de la croix directionnelle (BTNU, A8). Utilisé pour tirer des lasers depuis le vaisseau vers les aliens pendant une partie.

right Bouton right de la croix directionnelle (BTNR, D9). Utilisé pour déplacer le vaisseau du joueur vers la droite.

left Bouton gauche de la croix directionnelle (BTNL, C4). Utilisé pour déplacer le vaisseau du joueur vers la gauche.

blue Composante bleu de la sortie VGA.

red Composante rouge de la sortie VGA.

green Composante verte de la sortie VGA.

2.8 Package

Annexes

3

3.1 Convertion d'image

Le bloc *Display* affiche des données à l'écran en affectant une valeur au trois signaux *red*, *green* et *blue*. Les deux premiers sont contenues sur trois bits, alors que le dernier est uniquement sur deux. Celà signifie que le jeux de couleurs à disposition vaut :

$$\begin{aligned} n_{colors} &= 2^8 \\ &= 256 \end{aligned}$$

Une couleur est ainsi affectée à chaque pixel, en commançant par celui en haut à gauche de l'écran, puis celui à sa droite et ainsi de suite jusqu'à arriver à la droite de l'écran et commencer la ligne suivante. Il apparait alors que puis afficher une image, il faut extraire le code couleur de chaqu'un de ses pixels, puis les stocker d'une des deux façon suivantes :

- Dans une RAM ou une ROM, celà implique de convertir l'image en un fichier *COE* de la forme suivante :

```
memory_initialization_radix=16;
memory_initialization_vector=
43,
26,
2,
6e,
6e,
4a,
d9,
b6,
6e,
dd,
b5,
6a,
dd,
b6,
6a;
```

  memory_initialization_radix=16; indique les valeurs sont stockées en hexadecimale. Il est possible de le faire en binaire ou en decimal.
- Dans un tableau 2D :

```
type memoryPicture is array(0 to 4, 0 to 2) of integer;
constant picture : memoryPicture :=(
(16#43#,16#26#,16#2#),
(16#6e#,16#6e#,16#4a#),
(16#d9#,16#b6#,16#6e#),
(16#dd#,16#b5#,16#6a#),
(16#dd#,16#b6#,16#6a#));
```

Avec un COE, chaque valeur est stocké à la suite. En utilisant un tableau VHDL, il est possible de stocker en deux dimensions les valeurs pour avoir une représentation identique à celle de l'écran. La forme *16#<value>#* indique en VHDL que le nombre est sous forme hexadecimale et peut être directement assigné à un signal de type sdl_logic_vector.

Afin de rapidement convertir des images en fichier COE ou en tableau 2D VHDL, nous avons écris un script Matlab. Le détail de son fonctionnement est inclus dans les commentaires du code.

Format des images

Seul les images au format "JPG" sont supportés par le script.

3.1.1 Script Matlab - Convertion en fichier COE

```
%read the image
I = imread('yourPicture.jpg');
[x,y,z] = size(I); % x = width, y = heigh
width = x-1;

%Extract RED, GREEN and BLUE components from the image
R = I(:,:,1);
G = I(:,:,2);
B = I(:,:,3);

%make the numbers to be of double format for
R = double(R);
G = double(G);
B = double(B);

%Raise each member of the component by appropriate value.
R = R.^(3/8); % 8 bits -> 3 bits
G = G.^(3/8); % 8 bits -> 3 bits
B = B.^(1/4); % 8 bits -> 2 bits

%tranlate to integer
R = uint8(R); % float -> uint8
G = uint8(G);
B = uint8(B);

%minus one cause sometimes conversion to integers rounds up the numbers wrongly
R = R-1;
G = G-1;
B = B-1;

%shift bits and construct one Byte from 3 + 3 + 2 bits
G = bitshift(G, 2);
R = bitshift(R, 5);
COLOR = R+G+B;

%save variable COLOR to a file in HEX format for the chip to read
fileID = fopen ('output.coe', 'w');
fprintf(fileID, 'memory_initialization_radix=16;\n');
fprintf(fileID, 'memory_initialization_vector=\n');

for i = 1:size(COLOR(:), 1)-1
    fprintf (fileID, '%x', COLOR(i)); % COLOR (dec) -> print to file (hex)
    fprintf (fileID, ',\n');
end
fprintf (fileID, '%x;', COLOR(size(COLOR(:), 1))); % last pixel
fclose (fileID);

%translate to hex to see how many lines
COLOR_HEX = dec2hex(COLOR);
```

3.1.2 Script Matlab - Convertion en tableau VHDL

```
%read the image
I = imread('yourPicture.jpg');
[x,y,z] = size(I); % x = width, y = heigh
width = x-1;

%Extract RED, GREEN and BLUE components from the image
R = I(:,:,1);
G = I(:,:,2);
B = I(:,:,3);

%make the numbers to be of double format for
R = double(R);
G = double(G);
B = double(B);

%Raise each member of the component by appropriate value.
R = R.^(3/8); % 8 bits -> 3 bits
G = G.^(3/8); % 8 bits -> 3 bits
B = B.^(1/4); % 8 bits -> 2 bits

%tranlate to integer
R = uint8(R); % float -> uint8
G = uint8(G);
B = uint8(B);

%minus one cause sometimes conversion to integers rounds up the numbers wrongly
R = R-1;
G = G-1;
B = B-1;

%shift bits and construct one Byte from 3 + 3 + 2 bits
G = bitshift(G, 2);
R = bitshift(R, 5);
COLOR = R+G+B;

%save variable COLOR to a file in HEX format for the chip to read
fileID = fopen ('output.vhd', 'w');
fprintf (fileID, 'type memoryPicture is array(0 to ');
fprintf (fileID, '%d', y-1);
fprintf (fileID, ', 0 to ');
fprintf (fileID, '%d', x-1);
fprintf (fileID, ') of integer;\n');
fprintf (fileID, 'constant picture : memoryPicture :=(\n(');
for i = 1:size(COLOR(:), 1)-1
    fprintf (fileID, '16#');
    fprintf (fileID, '%x', COLOR(i)); % COLOR (dec) -> print to file (hex)
    fprintf (fileID, '#');
    if width == 0 % line end
        fprintf (fileID, '),\n(');
        width=x-1;
    else % not end of line
        fprintf (fileID, ',');
        width=width-1;
    end
end
fprintf (fileID, '16#');
fprintf (fileID, '%x', COLOR(size(COLOR(:), 1))); % last pixel
fprintf (fileID, '#');
fprintf (fileID, '));');
fclose (fileID);

%translate to hex to see how many lines
COLOR_HEX = dec2hex(COLOR);
```