3- Schéma structurel du projet

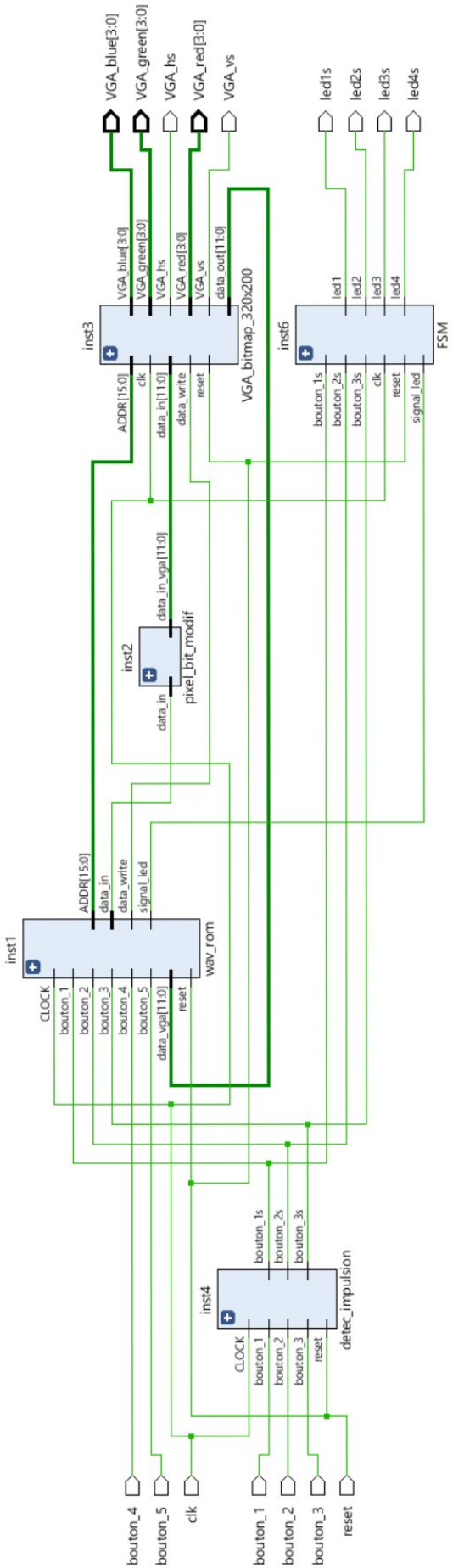

Figure-3 : schéma bloc du circuit

II. Modules du projet

1- pixel_bit_modif

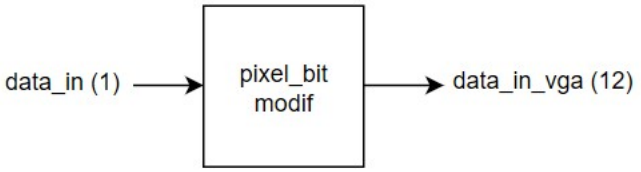

Figure-4 : vue externe du bloc

Ce bloc sert à générer le signal de l'information qui sera envoyé par la suite au VGA . En effet , le signal Data envoyé au VGA est codé sur 12 bits ce qui permet un affichage en couleur . Par contre , nous avons choisi d'utiliser un signal d'information codé sur 1 bit pour faciliter la tâche Nous ne traitons que des images en noir et blanc .

```
Process (data_in)
begin
    if (data_in = "0") then
        data_in_vga <= "000000000000";
    else
        data_in_vga <= "111111111111";
    end if;
end Process;
```

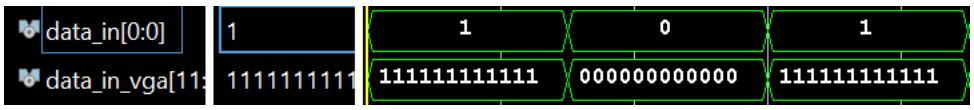

Figure-5 : simulation du bloc

2- VGA_bitmap320

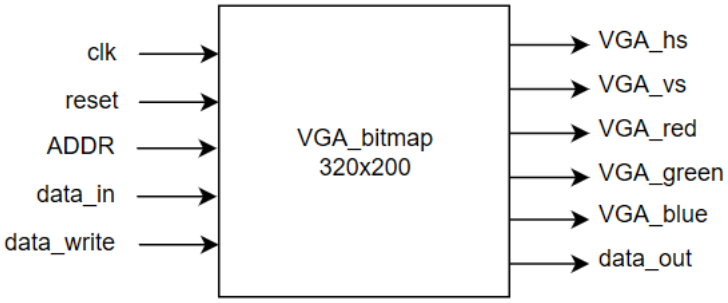

Figure-6 : vue externe du bloc

Il s'agit d'un module fourni qui gère l'affichage sur l'écran VGA, le signal ADDR représente l'adresse de chaque pixel et data_in l'information à écrire à l'adresse ADDR . Le signal data_write sert à indiquer le type d'accès à la mémoire du VGA : data_write <= 0 pour récupérer les données et data_write <= 1 pour une écriture dans la mémoire .

Explication du choix de la résolution 320 x 200

Le jeu 'Taquin' consiste à modifier l'image initiale de manière à la mettre en ordre ce qui impose dans un premier temps une mémorisation de l'image initiale , la recopier dans une autre mémoire pour pouvoir effectuer des modifications sans perdre l'image initiale . De plus , le jeu impose l'ajout d'une autre mémoire puisque la modification de la mémoire ne peut pas se faire directement . Le choix est donc fait pour ne pas dépasser les limites (RAM) de la carte FPGA à disposition (Nexys4) .

3- FSM

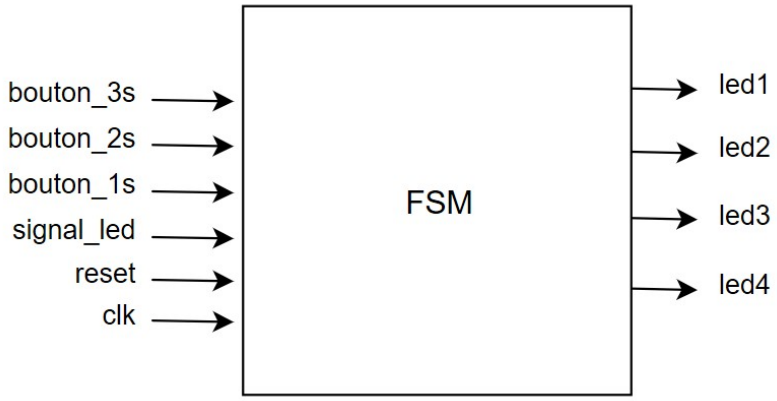

Figure-7 : Vue externe du bloc

Cette machine d'état permet de déterminer l'état du jeu , on précise donc quatre états de jeu :
- Game_off : L'écran n'affiche rien et c'est l'état initial , pour entrer dans le jeu il faut désactiver le 'reset' et entrer en mode d'initialisation de l'image (image initiale à mettre en ordre).
- Init : L'écran affiche l'image de départ du jeu.
- Game_on : C'est l'état qui indique que le joueur est en train de jouer.
- Game_over : Le joueur a réussi à mettre en ordre l'image initiale et considéré gagnant.

***Les états du jeu sont affichés avec 4 LED , l'affichage sur l'écran VGA de ces états nécessite plusieurs autres allocations de la mémoire , nous avons donc choisi de faire un affichage sur les LED.**

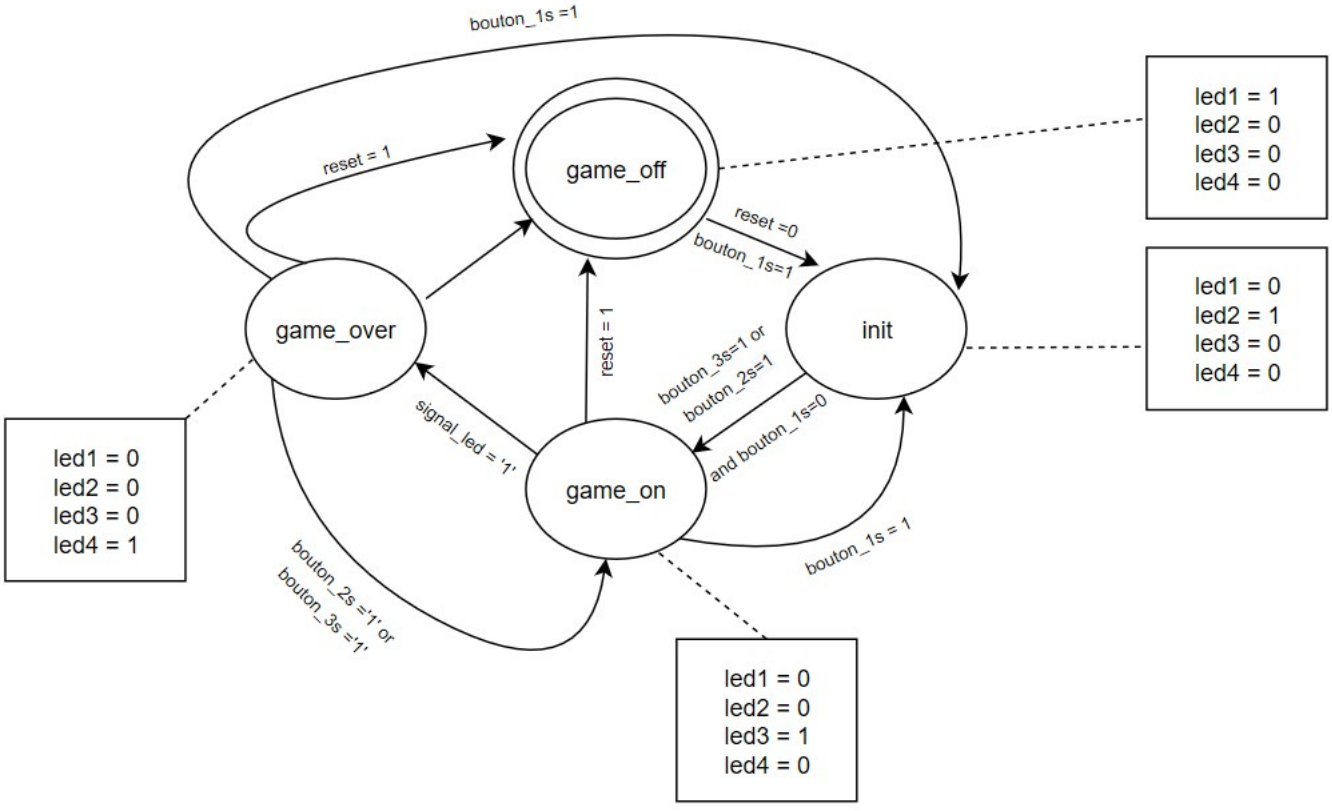

Figure-8 : Machine d'état

Une seule LED s'allume dans chaque état et l'état des LEDS est défini comme suit :

game_off → led1-ON

Init → led2-ON

game_on → led3-ON

game_over → led4-ON

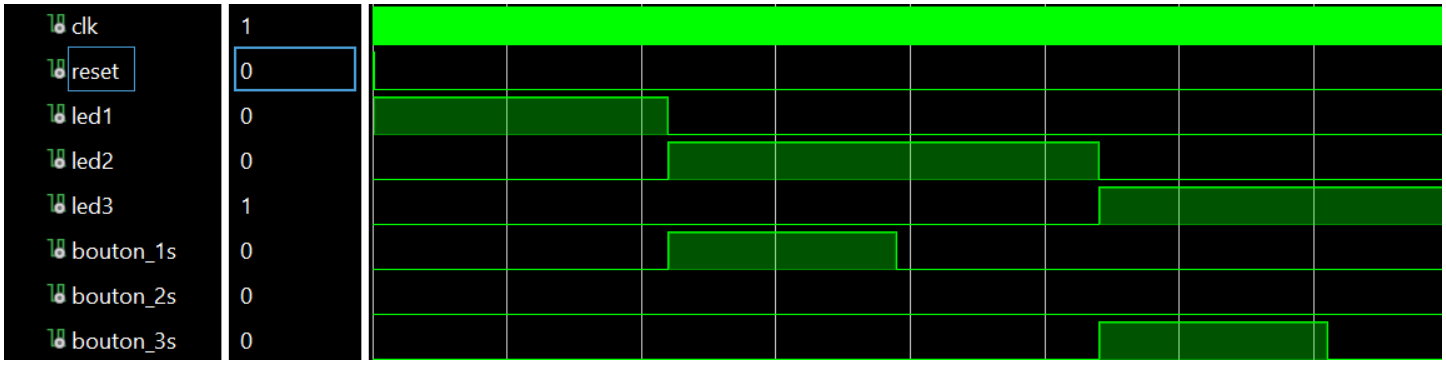

Figure-9 : simulation de la machine d'état

4- Detec_impulsion

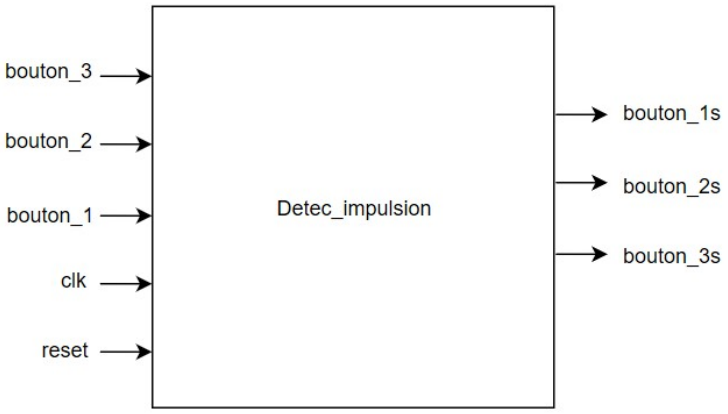

Figure-10 : Vue externe du bloc

Ce bloc sert à éliminer le phénomène de rebondissement des boutons poussoirs . En effet , nous avons réussi à éliminer ce phénomène en ajoutant une temporisation de 30 ms qui est largement supérieure au temps de rebondissement.

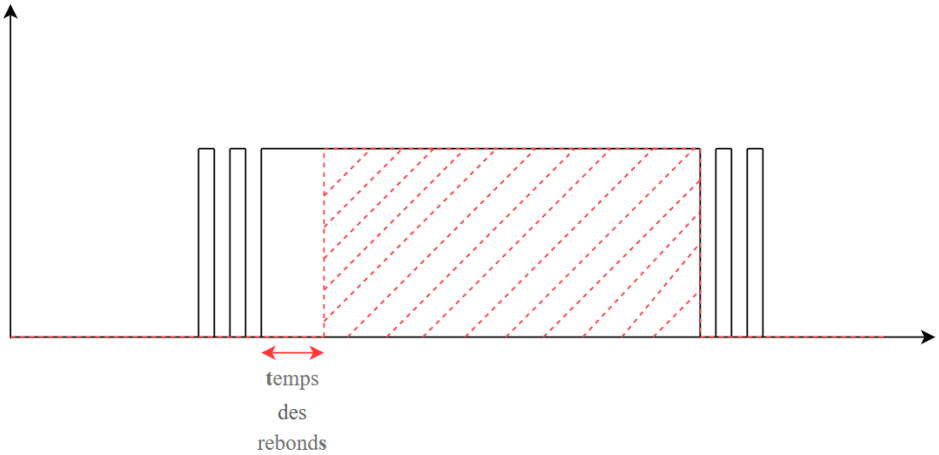

Figure-11 : Réponse des boutons poussoirs aprés l'anti-rebonds

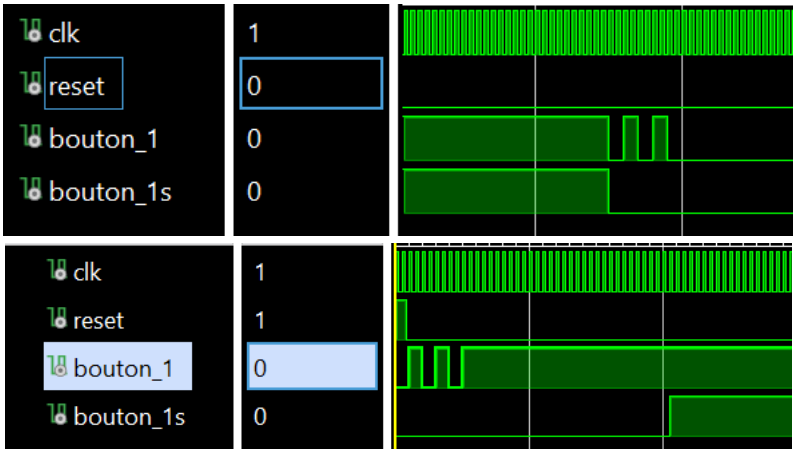

Figure-12 :Simulation de l'anti-rebonds

5- wav_ROM

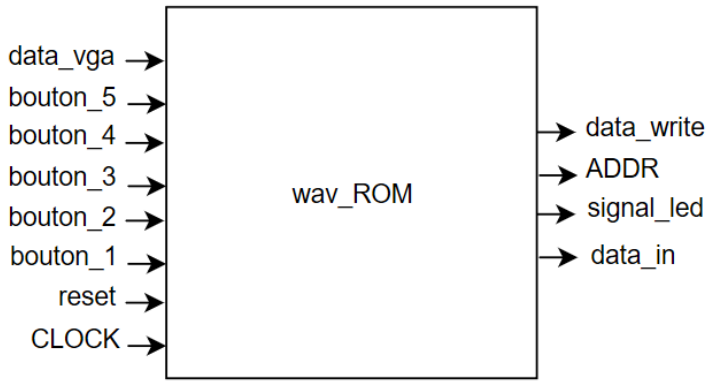

Figure-13 : Vue externe du bloc

Étape 1 : Stocker l'image initiale

L'image initiale est stockée dans une ROM : le signal 'memory' est un vecteur de taille 64000 de UNSIGNED . 64000 est le nombre de pixels à afficher (320 x 200).

```
  TYPE rom_type IS ARRAY (0 TO 63999) OF UNSIGNED (0 DOWNTO 0);
  SIGNAL memory : rom_type :=(TO_UNSIGNED(1, 1),
TO_UNSIGNED(1, 1),
TO_UNSIGNED(1, 1),
TO_UNSIGNED(1, 1),
TO_UNSIGNED(1, 1),
TO_UNSIGNED(1, 1),
TO_UNSIGNED(1, 1),
TO_UNSIGNED(1, 1),
TO_UNSIGNED(1, 1),
TO_UNSIGNED(1, 1),
TO_UNSIGNED(1, 1),
TO_UNSIGNED(1, 1),
```

la génération des pixels de l'image a été faite en Python (Voir Annexe pour le code)

Étape 2 : Afficher l'image initiale

Lors d'une première analyse du projet , nous avons rendu compte qu'il faut , à un certain moment , remplir une mémoire et afficher (dans le même état) ce qu'elle contient .Pour faire le décalage souhaité , nous avons utilisé le compteur d'adresses ADDR_W défini comme suit :

```
if (ADDR_2 = to_unsigned (63998,16) )then
    ADDR_W  <= to_unsigned (0,16);
elsif (ADDR_2 = to_unsigned (63999,16) )then
    ADDR_W  <= to_unsigned (1,16);
else
    ADDR_W  <= ADDR_2 + to_unsigned (2,16);
end if;
```

La définition de ce compteur se fait chaque front montant de l'horloge ce qui permet de le décaler d'une seule valeur.

```
if(( bouton_1 = '1')) then
        data_in <= STD_LOGIC_VECTOR( memory2(to_integer(ADDR_W)));
        data_write <= '1';
        memory2(to_integer (unsigned (ADDR_2 ))) <= memory(to_integer(ADDR_2));
        memory3(to_integer (unsigned (ADDR_2 ))) <= memory(to_integer(ADDR_2));
```

Les mémoires 'memory2' et 'memory3' sont donc bien remplis . L'affichage « data_in < = ce qui est dans 'memory2' » se fait en front montant de l'horloge ce qui implique un bon affichage de l'image désirée.

Étape 3 : Détection de la position de l'adresse en (x , y)

Cette localisation de la position de l'adresse était nécessaire puisque le concept du jeu nécessite à faire un changement de mémoire pour des positions spécifiques de l'adresse du pixel. En revanche, l'utilisation de l'opérateur « modulo » n'est pas une solution valable en VHDL pour des chiffres très grands (320 et 100 dans notre cas) . De ce fait , nous avons défini les signaux x et y comme suit :

-x est incrémenté de 1 quand l'adresse s'incrémente et atteint une valeur limite de 319 puis fait le retour à 0 .
-y s'incrémente lorsque la valeur 0 est attente par x .

*Les deux signaux sont initialisés à 0 (pareil pour l'adresse)

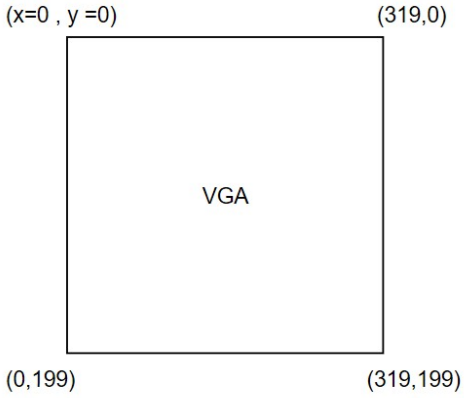

Figure-14 : Position de l'adresse

Figure-15 : Variations des compteurs de position x et y

Figure-16 : Variations des compteurs de position x et y à la limite d'une ligne

Étape 4 : Modification des mémoires

La modification de la mémoire se fait selon le changement désiré :
Bouton2 appuyé => inverser horizontalement
Bouton3 appuyé => inverser verticalement

*La démarche reste la même pour les deux types de changement. Nous limiterons l'explication de la démarche pour un seul type de changement .

Dans un premier temps , nous avons travaillé sur une image à 4 parties :

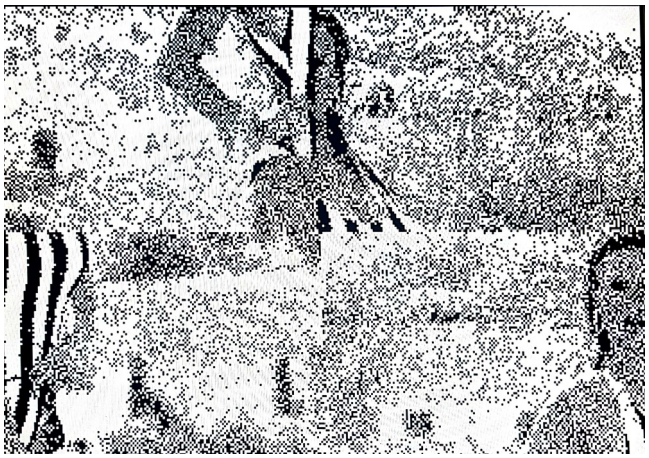

Figure-17 : Image après modification (switch)

Un mouvement est donc possible pour chaque type de « Switch » :
Pour le Switch horizontal , 2 possibilités sont à prendre en compte :
1- Le vide se trouve à la partie supérieure de l'image
2- Le vide se trouve à la partie inférieure de l'image

Nous utilisons la mémoire 3 pour faire le Switch :
- memory3(adresse + 160) <= memory2(adresse) pour 0 <= x <=159
- memory3(adresse - 160) <= memory2(adresse) pour 160 <= x <=319
L'écran est divisé en 2 ce qui explique le fait de décaler par 160 = 320 /2 puisque le Switch est horizontal .

```
if ( (x <=159 and y <= 99 ) ) then
      memory3(to_integer (unsigned (ADDR_2 + to_unsigned (160,16) ))) <= memory2(to_integer (unsigned (ADDR_2))) ;
elsif ( x > 159 and y <= 99 ) then
     memory3(to_integer (unsigned (ADDR_2 - to_unsigned (160,16)))) <= memory2(to_integer (unsigned (ADDR_2))) ;
else
     memory3(to_integer (unsigned (ADDR_2))) <= memory2(to_integer (unsigned (ADDR_2))) ;

end if;
```

Figure-18 :Partie du script Switch horizontal lorsque le vide est en haut

```
if ( (x <=159 and y > 99 and y <= 199 ) ) then
      memory3(to_integer (unsigned (ADDR_2 + to_unsigned (160,16) ))) <= memory2(to_integer (unsigned (ADDR_2))) ;

elsif ( x > 159 and y > 99 and y <= 199 ) then
     memory3(to_integer (unsigned (ADDR_2 - to_unsigned (160,16)))) <= memory2(to_integer (unsigned (ADDR_2))) ;
else
     memory3(to_integer (unsigned (ADDR_2))) <= memory2(to_integer (unsigned (ADDR_2))) ;

end if;
```

Figure-19 :Partie du script Switch horizontal lorsque le vide est en bas

Étape 5 : Mémorisation de la position du vide

Le jeu de taquin consiste à déplacer une case vide sur l'image ce qui implique une obligation de mémoriser la position de cette case . En effet , le joueur déplace le vide et ne fait pas de simples inversions de cases. La mémorisation se fait chaque appui sur un bouton . Dans notre cas , deux positions sont possibles pour le vide ce qui implique un changement des cordonnées pendant chaque appui sur le bouton ; l'appui est détecté par un changement d'état du bouton au front montant de l'horloge ;

```
    bouton_test_3 <= bouton_3;

  if ( bouton_3 = '1' and  bouton_test_3 = '0' ) then
       if (y0<2) then
        y0 <= 10 ;
        else
        y0 <= 0 ;
        end if;
  else
        y0 <= y0 ;
   end if;
```

```
                        bouton_test <= bouton_2;
if ( bouton_2 = '1' and  bouton_test = '0' ) then
     if (x0<2) then
         x0 <= 10 ;
     else
         x0 <= 0 ;
     end if;
else
     x0 <= x0;
end if;
```

Figure-20 :Script de modification de la position du vide

*La valeur 10 est aléatoire et indique la deuxième position dans chaque cas

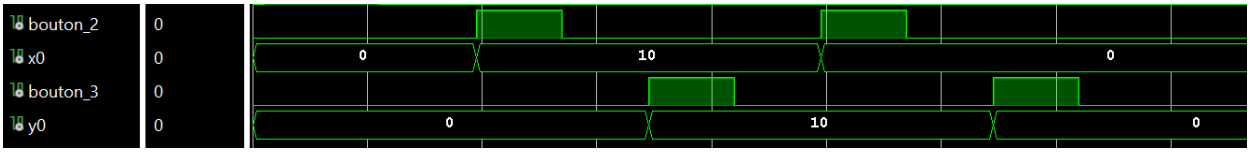

Figure-21 : Simulation de la position du vide

Étape 6 : Générer un signal de gagne

La mémoire 3 contient la version finale affichée à l'écran , pour tester si le joueur a gagné , il faut vérifier si la mémoire 3 contient l'image désirée (en ordre) . Le fait de bien parcourir la mémoire3 permet de comparer les deux mémoires.

Une variable 'var' est incrémentée de 1 si les mémoires sont identiques et on considère le jeu gagné si cette variable dépasse 64000 qui est le nombre de pixels .

Ceci permet de générer le signal 'signal_led ' qui change l'état du jeu à Game_over.

```
if ( memory3(to_integer(ADDR_2)) = memory(to_integer(ADDR_2))) then
    var <= var + 1 ;
else
    var <= 0 ;
end if;
if (var >= 63999 ) then
    signal_led <= '1' ;
else
    signal_led <= '0';
end if ;
```

Figure-22 : Comparaison avec l'image initiale et génération du signal de gagne

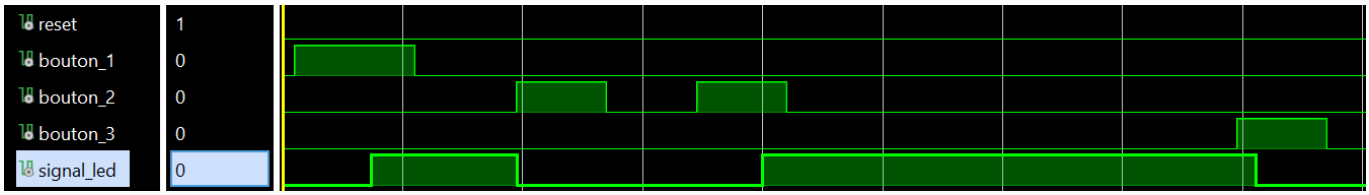

Figure-23 : Simulation du signal de gagne

III. Amélioration

Le jeu de Taquin n'est pas encore fini , il nous reste à faire le Switch à 16 parties de l'écran . Comme amélioration du code , nous avons réussi à faire le Switch horizontal à 4 parties de l'image . La complexité du code augmente et la définition des valeurs initiales et finales du Switch est beaucoup plus complexe: le Switch ne peut se faire que de gauche à droite , l'autre partie n'est pas encore développée (bouton 5) .

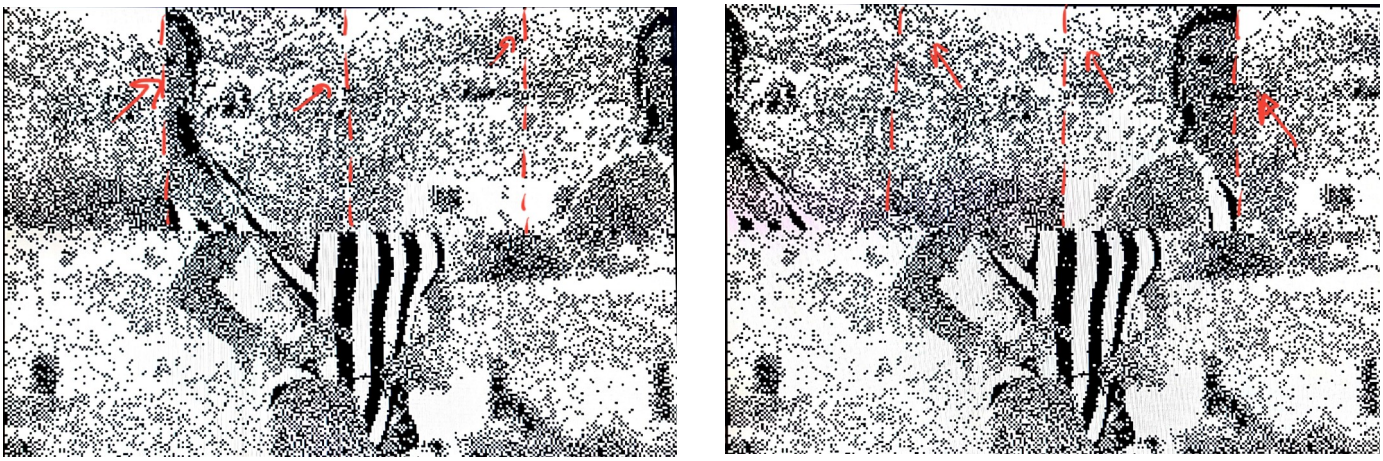

Figure-24 : Résultat du Switch horizontal à 4 parties

La mémorisation de la position du vide est différente puisqu'il y a 4 positions . Une fois le vide est à la position finale sa position est maintenue et le Switch n'est plus valable .

```
if ( bouton_2 = '1' and  bouton_test = '0' ) then
     if (x0<240) then
         x0 <= x0+80 ;
     else
         x0 <= 240 ;
     end if;
else
     x0 <= x0;
end if;
```

IV. Ressources consommées

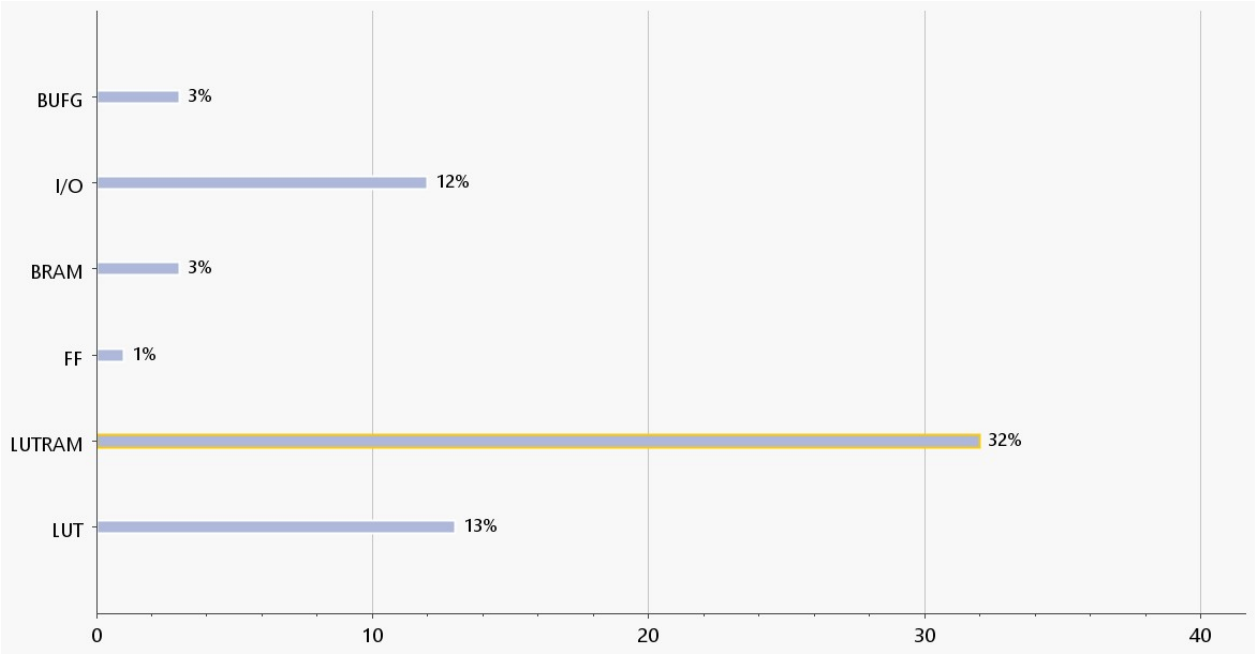

Figure-25 : Ressources consommées du circuit

Le type de ressources le plus consommé : LUTRAM avec un pourcentage de 32%. Le diagramme précédent issu du rapport synthèse fourni par VIVADO présente l'utilisation respective des Flip-Flops, tableaux internes, des ports In/Out de la carte et les tampons. La fréquence maximale à partir du Worst Pulse Width Slack (WPWS) qui est égale à 3,75 ns est : Fmax=266,66 MHz

Worst Pulse Width Slack (WPWS):	3.75 ns

V. Conclusion

Notre objectif était de réaliser un jeu 'Taquin' . Nous avons pu définir les blocs et les adapter pour faire des corrections . Nous avons essayé de comprendre les différentes manipulations d'une image sur un écran VGA ; nous avons réussi à construire une simple image et l'afficher , mettre une image prête dans une mémoire RAM , la stocker dans une ROM et lire à partir de l'écran VGA . La gestion des mémoires était la partie la plus importante du projet puisque le jeu de Taquin n'est en fait qu'une mémorisation des états de l'image et un changement de mémoire . À l'heure où nous terminons ce rapport nous ne n'avons pas pu atteindre cet objectif à 100 % . Néanmoins, le résultat est satisfaisant . En effet , même si tous les cas ne sont pas codés, l'architecture est entièrement présente et Il suffit d'implémenter chaque cas manquant . Avec plus de temps nous aurions pu finir tous les cas du Switch et compléter le jeu .

VI. Annexe

```python
import cv2
img = cv2.imread('/home/amou/Bureau/A1.jpg',2)
ret, bw_img = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
cv2.imshow("Binary Image",bw_img)
list = bw_img
f=0
file = open("    Girl.txt", "w")
for i in list:
    for j in i:
        #print(j)
        if(j==255):
                j=1

        file.write("TO_UNSIGNED("+str(j)+", 1),\n")
        print("TO_UNSIGNED("+str(j)+", 1),")
        f=f+1

    #print(j)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Figure-26 : Script génération des pixels en python