
Filière électronique

Système embarqué

# Projet SE : Le Cube

Anthonin Picard
Clémence Gillet
Juliette Objois
Sébastien Duvivier

*Encadrants :* Bertrand LE GAL
Jérémie CRENNE

# Sommaire

# Partie 1

# Présentation du projet

Au cours de ce projet nous allons prendre en main un lot de **matrices RGB 32x32** vendues par **Adafruit** dans le but de développer des applications mettant en jeu un ou plusieurs écrans chaînés. Le challenge majeur de ce projet réside dans l'absence de datasheet présent sur les écrans bien qu'une bibliothèque pour Arduino soit fournie par Adafruit. A terme, l'objectif est de faire communiquer la matrice et un FPGA pour afficher des jeux, afficher des informations en provenance de capteurs et faire défiler des messages.

Le matériel utilisé est :
- un lot de 6 matrices 32x32 avec leurs câbles de communication et d'alimentation
- une arduino Uno
- un carte développement Digilent Nexys A7
- des périphériques de contrôle tel que des joysticks.

Le projet s'est décomposé en 2 étapes :
- Compréhension du protocole de communication et d'affichage
- Développement d'applications

Dans un premier temps, nous allons donc d'abord utiliser la carte Arduino Uno pour prendre en main la matrice et trouver des idées d'applications. En effet, une bibliothèque arduino développée par Adafruit propose de nombreuses fonctions comme par exemple dessiner des formes, écrire des caractères ou encore adresser les pixels un par un. Cette étape sera réalisée très rapidement afin de ne pas perdre de temps pour la suite.

Puis, nous passerons sur FPGA afin de gagner en fréquence et d'être capable plus tard de chaîner les matrices entres elles. Nous devrons donc comprendre le fonctionnement de la matrice. Cette partie sera la plus complexe à réaliser car l'on trouve très peu de documentation à ce sujet.

# Partie 2

# Fonctionnement de la matrice

## 2.1 Analyse des composants

Afin de comprendre le comportement et le protocole attendu par la matrice, il est intéressant de regarder les composants présents au dos du panneau de LED. On trouve 3 composants principaux :

- 74HC245 ( x2 ) qui est un **octal** 3-state bus tranceiver pouvant être configuré dans les 2 directions.
- TC7258EN ( x2 ) qui est un **décodeur** 4 vers 16 lignes.
- DB5020B ( x12 ) qui est un **driver** de LEDS 16 sorties.

Le décodeur prend en entrée les 4 bits d'adresse et sélectionne la ligne à driver. Chaque décodeur contrôle donc 16 lignes ce qui permet d'adresser au total 2 lignes simultanément.

Les 12 drivers possédant chacun 16 sorties, il est possible de driver au total 192 LEDs soit le nombre de LEDs présents dans 2 lignes : 32 pixels/ligne x 3 LEDs/pixel x 2 lignes.

Le contrôle des composants se fait à travers le protocole présenté dans la partie suivante.

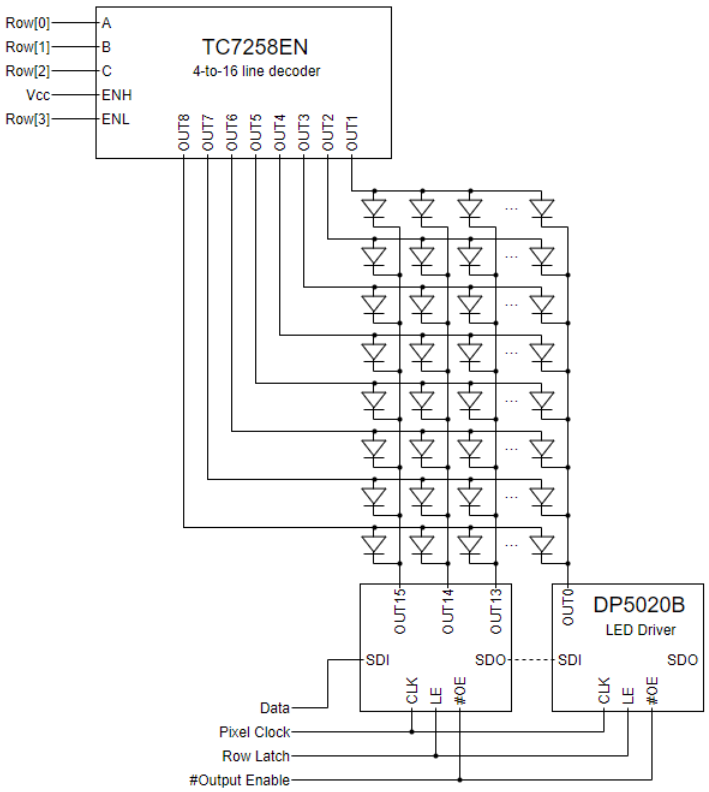



Figure 2.1 – Schéma de fonctionnement des composants

## 2.2 Protocole de communication

### 2.2.1 Fréquence d'utilisation

Une horloge est fournie à la matrice, pouvant atteindre **30 MHz** en théorie. Cette limite est fixée par un composant interne à la matrice (cf datasheet). Cependant, les nombreux tests effectués ont montrés qu'il est possible d'afficher une image statique à une fréquence de **45MHz**. Les différents tests ont donc été effectués à basse fréquence (10MHz), avant d'augmenter celle-ci.

### 2.2.2 Fonctionnement de la matrice

La matrice est composée de deux parties : la partie haute, et la partie basse. Cela permet d'afficher une image deux fois plus rapidement et d'obtenir un rafraîchissement plus fluide.

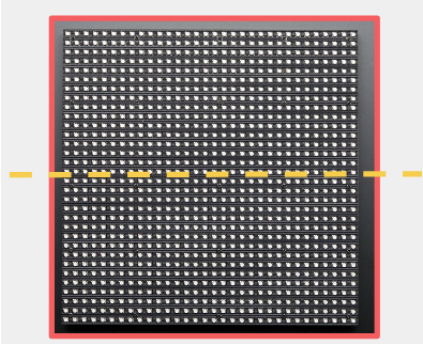


FIGURE 2.2 – La matrice est composée de deux parties

Afin de parvenir à cet affichage rapide, deux pixels sont affichés en même temps. Il faut donc fournir deux fois les composants RGB de la matrice.

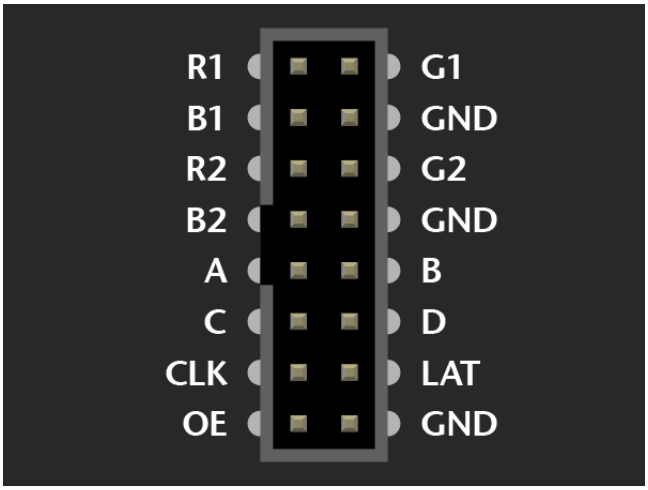



FIGURE 2.3 – Broches d'entrées de la matrice

Les broches R1/G1/B1 sont les composantes RGB de la partie haute de la matrice, et R2/G2/B2 servent à la partie basse.

### 2.2.3 Sélection du pixel

7 broches sont utilisées pour sélectionner le pixel :

- **D/C/B/A** servent à sélectionner le ligne sur laquelle écrire. Par exemple, si (DCBA)=(0001), alors la deuxième ligne sera sélectionnée. La couleur sera donc affichée sur la deuxième ligne (et la 17ième, puisque deux pixels sont écrit simultanément !). Ces 4 bits adressent donc 16 lignes pour la partie haute et basse en même temps.
- Le signal **CLK** sert à sélectionner précisément un pixel sur une ligne. Chaque front montant de celui-ci signifie que les entrées (RGB) sont à prendre en compte. Ainsi, lors du prochain front montant, ce sera le pixel suivant qui sera pris en compte (sur la même ligne). Quand une ligne entière a été envoyée, il faut envoyer un "latch" afin de revenir à la ligne.
- **LAT (Latch)** : Ce signal permet de revenir à la ligne. Si les broches (DCBA) ne changent pas, cela permet d'effectuer une PWM. Si la sortie de ces broches change, cela permet d'afficher sur une nouvelle ligne.
- **OE (Output Enable**) : Ce signal permet de valider l'affichage de la matrice entière. Ainsi, pour afficher une autre image, il faut préalablement mettre brièvement ce signal à 1.

## 2.3 Description du bloc VHDL correspondant

### 2.3.1 Interface description

Afin de manipuler facilement la matrice, le module "Papillote" a été décrit. L'idée est de créer une interface, afin que chacun puisse utiliser facilement la matrice. Un framebuffer a été implanté afin de stocker l'image. Il est possible d'écrire à n'importe quel moment, puisque le Bloc RAM est à double port (un port en écriture, prioritaire, et un port en lecture). L'écriture s'effectue sur front montant du signal CLK, lorsque WE vaut "1". Ainsi, il est possible d'écrire une donnée à chaque cycle d'horloge.

L'interface possède pour entrées :

- clk : l'horloge.
- pwm_mode : doit être mis à 1. (0 signifie qu'il n'y a pas de PWM, et donc que uniquement 8 couleurs sont disponibles).
- rst : reset.
- x : entier, la coordonnée X à laquelle écrire.
- y : entier, la coordonnée Y à laquelle écrire.
- we : Write-Enable : vaut 1 lorsque l'on souhaite écrire dans le framebuffer.
- r_in : la couleur rouge du pixel ( vecteur de 8 bits).
- g_in : la couleur verte du pixel ( vecteur de 8 bits).
- b_in : la couleur bleu du pixel ( vecteur de 8 bits).

### 2.3.2 FSM

Ce module fonctionne grâce à la FSM suivante :

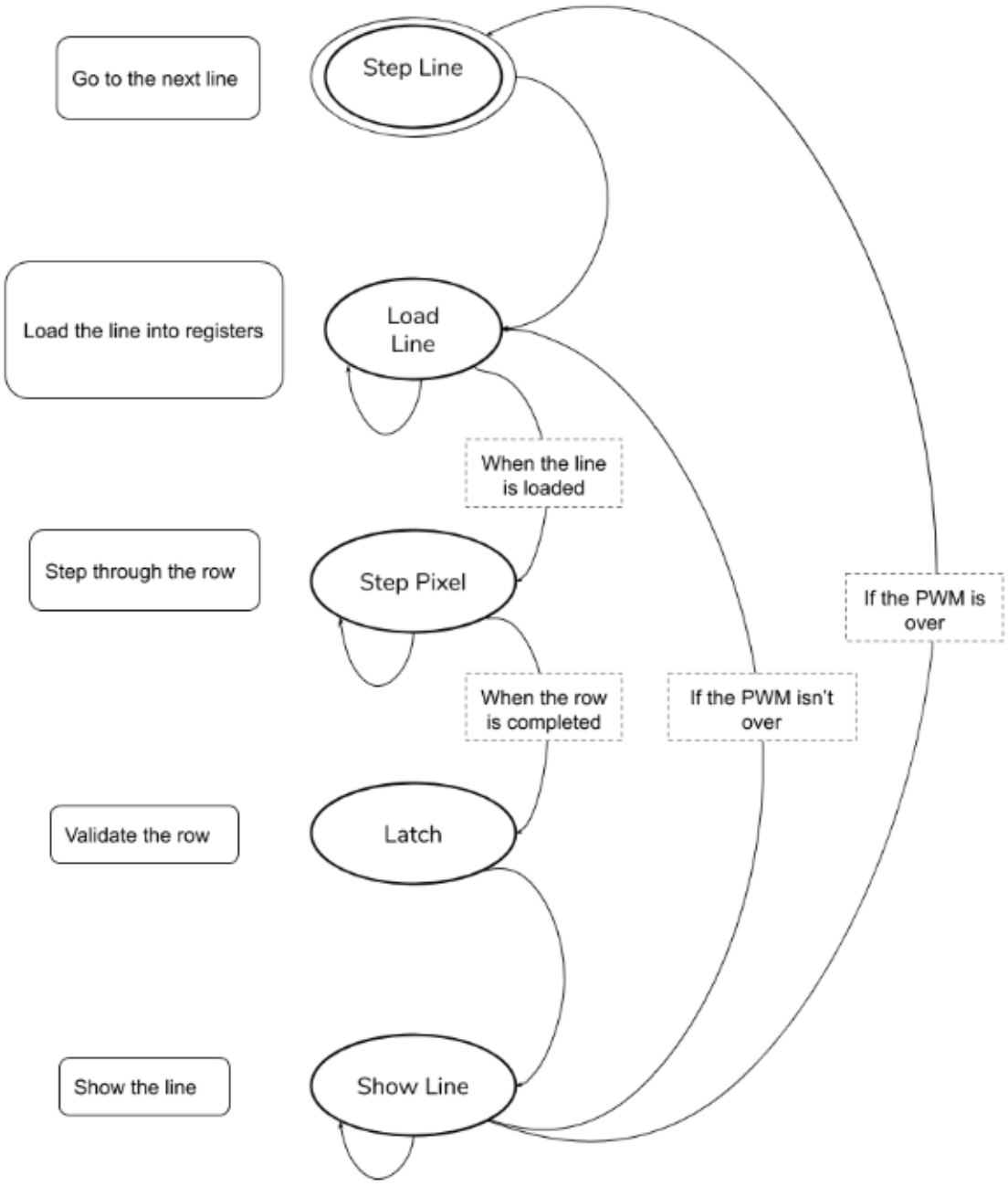



FIGURE 2.4 – Machine d'état

— L'état "**Load Line**" sert à charger l'ensemble de la ligne courante. Celle-ci se trouve initialement en mémoire RAM, il est donc nécessaire de la charger en registre afin d'y accéder rapidement.
— "**Step Pixel**" correspond à l'envoi des informations à la matrice. Deux pixels sont envoyés par coup d'horloge. C'est dans cet état que le signal CLK permet de valider les entrées.
— Le "**Latch**" permet de valider la ligne entière. Une fois ce signal envoyé, la matrice commence l'affichage d'une nouvelle ligne. Cela peut-être la même, ou bien la ligne suivante.
— L'état **"Show-line"** permet d'attendre en affichant un pixel. Cela est nécessaire afin que l'oeil humain perçoive la couleur. Chaque ligne est parcouru 256 fois, afin d'effectuer correctement la PWM.
— Enfin, "**Step Line**" valide entièrement l'image actuel, grâce à la broche "OE".

### 2.3.3 PWM

La mémoire de ce module est constituée de **deux blocs RAM** (un pour la partie haute, un pour la partie basse). Chacune stock 32*16 mots de 24 bits (le rouge est codé sur 8 bits, ainsi que le vert et le bleu). Il est donc nécessaire d'effectuer une PWM afin de transmettre cette information de 8 bits sur une unique broche. L'idée sous-jacente est la suivante : si la valeur en mémoire est 127, alors la broche doit être mise à 1 la moitié du temps.

Cela est réalisé avec un simple compteur, allant jusqu'à 256. Si la valeur en mémoire est supérieure à ce compteur, alors le sous-pixel est allumé. Autrement, il est éteint.

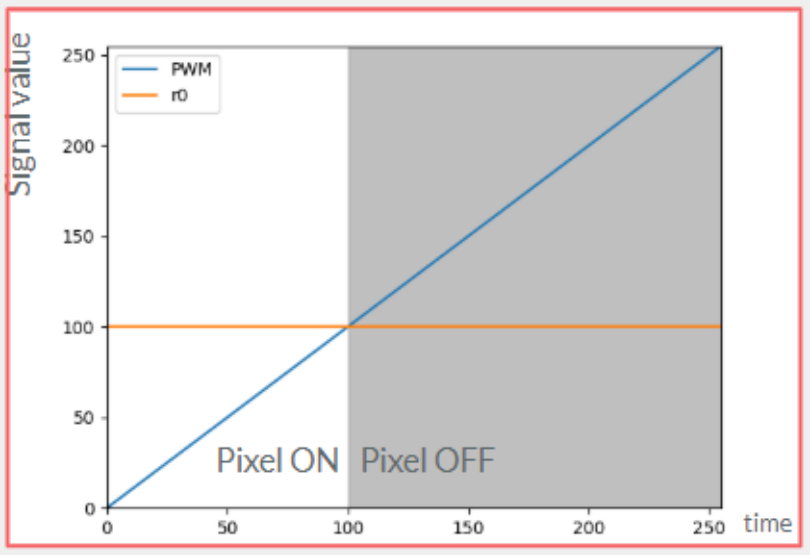



Figure 2.5 – Principe de la PWM

# Partie 3

# Applications

## 3.1 Le générateur de caractère

L'objectif final de ce bloc était d'être capable d'afficher un **message défilant** sur la matrice de LED, voir sur le cube entier. Mais dans un premier temps, nous allons afficher un message statique sur la matrice. Chaque caractère sera de taille **5x8 pixels** comme décrit sur l'image ci-dessous, et sa position sera déterminé par le coin **en bas à gauche** indiqué par un croix :

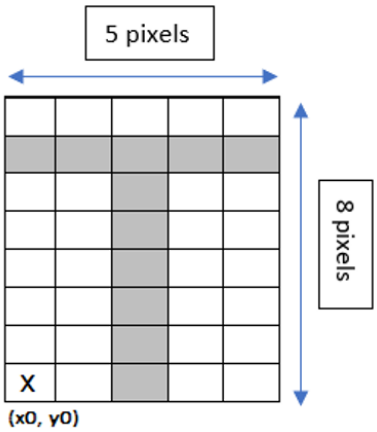



FIGURE 3.1 – Taille d'un caractère

Pour écrire dans la RAM de la matrice, nous devons écrire la valeur de chaque pixel un par un. Pour cela, il faut indiquer la couleur (R, G, B) et l'adresse (x, y).

### 3.1.1 Description de l'architecture

Voici l'architecture générale. Les blocs ***geneChar*** et ***TOP_drawChar*** sont les blocs relatifs à l'écriture de caractère dans la ram dont nous allons développer le fonctionnement. Le bloc ***LEDMatrixDriver*** correspond à l'écriture de la ram dans la matrice de LED. Ce bloc sera identique pour toutes les applications et correspond à la description réalisée dans la partie précédente du fonctionnement de la matrice. Voici le schéma de l'architecture réalisée :

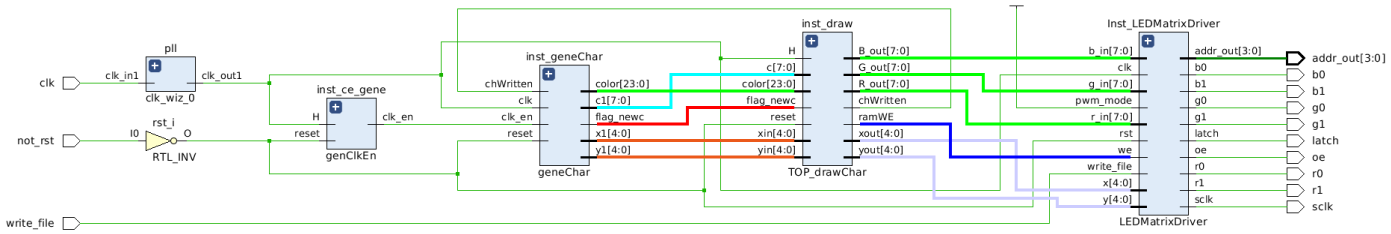



FIGURE 3.2 – Architecture générale

### Le générateur de caractère

Le premier bloc est le **générateur de caractère**. Voici un schéma présentant ses entrées et sorties :

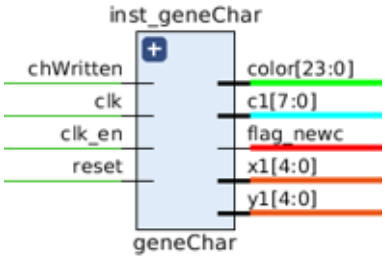



FIGURE 3.3 – Bloc geneChar

Celui-ci nous permet d'écrire le mot ou la phrase que l'on souhaite afficher de la façon suivante :

```
TYPE ram_type IS ARRAY ( 0 to 4) of integer;
signal chToWrite : ram_type := (H,E,L,L,O);
```

FIGURE 3.4 – Code VHDL issus du fichier "geneChar.vhd"

Pour un message statique, nous limiteront le nombre de caractère à 5. Pour un message défilant, nous pouvons imaginer écrire des phrase aussi longue qu'on le souhaite.

Entrées :
- clk (1 bit) : horloge à 30MHz.
- clk_en (1 bit) : signal indiquant "l'envoi" d'un nouveau caractère.
- chWritten (1 bit) : signal permettant de vérifier que la lettre précédente à bien été écrite dans la RAM avant d'envoyer la suivante afin d'être sur d'écrire correctement et entièrement chaque caractère.

Sorties :
- color (24 bits) : couleur du caractère. Dans l'ordre, le rouge , le vert puis le bleu chacun codés sur 8 bits.
- c1 (8 bits) : valeur ascii du caractère à écrire.
- flag_newc (1 bit) : flag positionné à 1 lorsqu'un nouveau caractère doit être écrit dans la RAM.
- x1 et y1 (5 bits) : Coordonnées du caractère. Ces coordonnées correspondent au coin an bas à gauche du caractère comme indiqué précédemment.

### L'affichage sur la matrice

Ce second bloc a pour objectif d'écrire chaque caractère dans la ram afin que ceux-ci soient affichés sur la matrice.

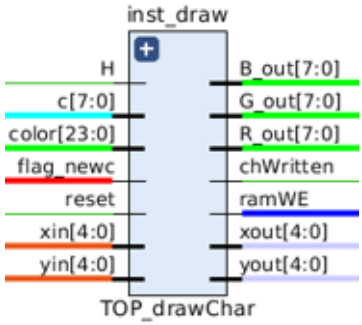



FIGURE 3.5 – Bloc drawChar

Entrées :
- H (1 bit) : hotloge à 30MHz.
- c (8 bits) : valeur ascii du caractère à écrire.
- color (24 bits) : couleur du caractère.
- xin et y in (5 bits) : coordonnées du caractère.

Sorties :
- R_out, G_out, B_out (8 bits) : couleurs du caractère donc de chaque pixels.
- chWritten (1 bit) : signal positionnée à 1 lorsque les 40 pixels d'un caractère sont écrit dans la RAM.
- ramWE (1 bit) : "write enable" à 1 lorsque l'on souhaite écrire dans la RAM.
- xout, yout (5 bits) : coordonnées de chaque pixels.

Ce bloc est composée de trois blocs :

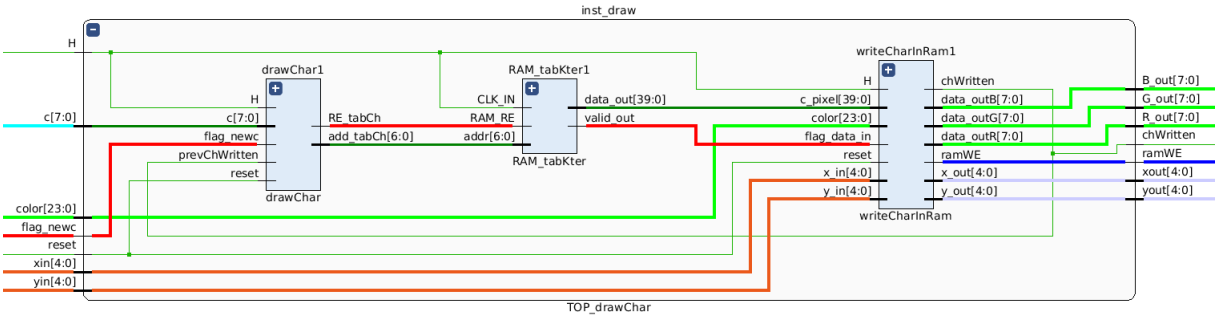



FIGURE 3.6 – Architecture générale

**drawChar :** Ce bloc vérifie que le caractère demandé est un caractère affichable et convertit les caractères accentués en caractère non accentué. Il envoie un signal (RE_ramCh) indiquant qu'il faut lire une valeur dans la ram contenant les caractère à l'adresse indiqué par le signal add_tabCh.

**RAM_tabKter :** Ce bloc contient une ram permettant de convertir chaque caractère ascii en un signal sur 40 bits indiquant les pixels devant être allumés pour afficher un caractère. Ces 40 bits correspondent aux 40 pixels de chaque caractère de la façon suivante (b0 étant le bit de poids faible et b39 le bit de poids fort) :

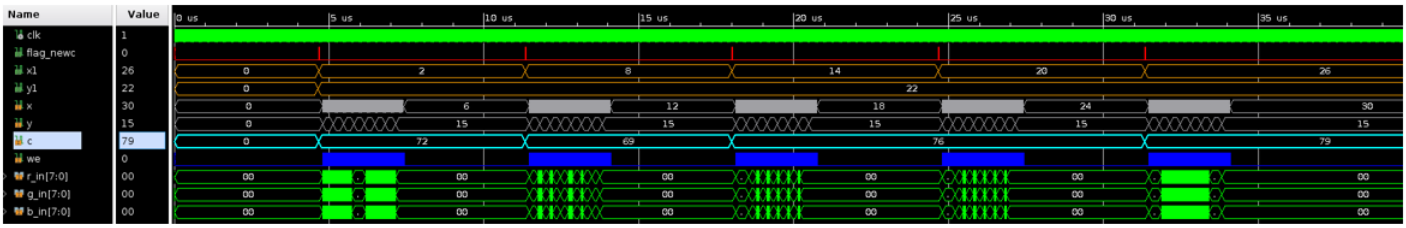
| | | | | |
|---|---|---|---|---|
| b35 | b36 | b37 | b38 | b39 |
| ... | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| b5 | ... | | | |
| b0 | b1 | b2 | b3 | b4 |

FIGURE 3.7 – Schéma d'un caractère

**writeCharInRam :** Ce bloc permet d'écrire le signal précèdent de 40 bits dans la ram de la matrice. En effet, comme expliqué précédemment, nous devons écrire ces 40 pixels un par un dans la ram. Le signal chWritten est à '1' lorsque les 40 pixels du caractère courant sont écrit. Cela indique que l'on peut passer au caractère suivant.

### 3.1.2 Simulations

Afin de vérifier le bon fonctionnement de notre application, nous avons réaliser une simulation.




Légende :

- flag_newc : indique lorsqu'un nouveau caractère doit être écrit dans la RAM.
- x1 et y1 : coordonnées du caractère.
- x et y : coordonnées de chaque pixels
- c : valeur ascii du caractère.
- we : write enable à 1 lorsque l'on souhaite écrire dans la RAM.
- r_in, g_in, b_in : couleur de chaque pixel.

FIGURE 3.8 – Simulation de l'application

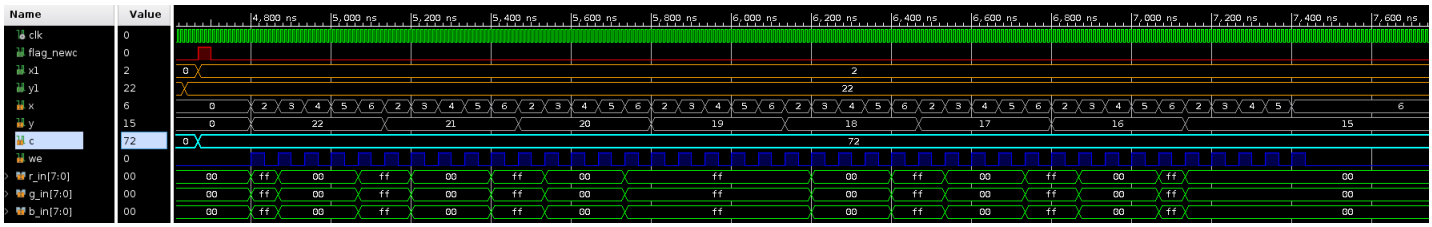



FIGURE 3.9 – Zoom sur la simulation

On observe le comportement désiré, notamment :

- L'écriture des 5 caractères suite au passage à 1 du flag flag_newc.
- Ce flag attend la validation du signal chWritten afin de s'assurer que la lettre précédente est complètement écrite dans la RAm avant de passer à 1.
- Pour chaque caractère, les coordonnées x et y balaye l'ensemble des 40 pixels et les valeurs de R, G et B correspondent bien à la couleur demandée c'est-à-dire 0xFFFFFF (blanc) pour le caractère et 0X000000 (noir) pour le fond.

### 3.1.3 Fichier PPM

Les simulations sous Vivado permettent de tester les blocs VHDL mais dans le cas de nos applications, ce n'est pas très visuel et donc plutôt compliqué de vérifier le bon fonctionnement de nos blocs. Nous avons donc utilisé les fichier ppm pour nous aider. Un **fichier PPM (Portable Pixmap)** est un format de fichier d'image très simple à générer. Deux étapes sont nécessaire afin de générer notre image PPM :

- Dans une **simulation VHDL**, on écrit dans un fichier .txt les valeurs de chaque pixel (de 0 à 255) pour les RAMs rouge, verte et bleu. Un signal write_file est positionné à 1 lorsque l'on souhaite écrire dans le fichier les valeurs. Cela permet d'écrire les valeurs une fois que les 5 lettres sont écrites dans la RAM et pas avant.
- Un **programme C** permet de convertir ce fichier .txt en fichier ppm.

Le fichier PPM peut alors facilement s'ouvrir avec n'importe quel gestionnaire d'image.

Dans le cas de l'écriture de caractère, nous avons pu vérifier le bon fonctionnement de notre application et observer le mot que nous souhaitons afficher :

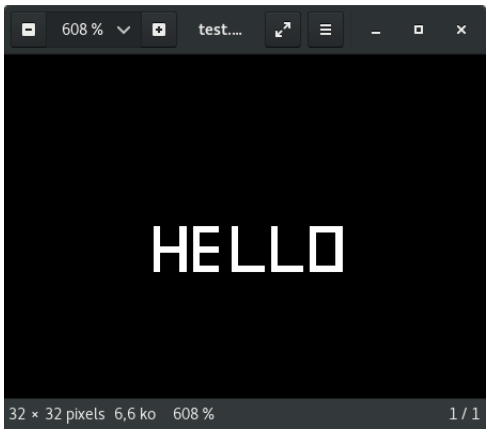



FIGURE 3.10 – Image PPM

### 3.1.4 Message défilant

La seconde étape est de réaliser un **message défilant**. L'architecture reste la même mais nous modifions le bloc geneChar afin de modifier les valeurs des coordonnées. En effet, nous rendons le calcul de celles-ci automatique. L'abscisse de la première lettre est appelée **x0** et se déplace vers la gauche à chaque cycle. Un cycle se termine lorsque toutes les lettres ont été écrites. Alors x0 prend la valeur de x0 - 1 et on écrit de nouveau chaque lettre. Les coordonnées des autres lettres seront dépendantes de x0 comme indiqué sur la figure ci-dessous.

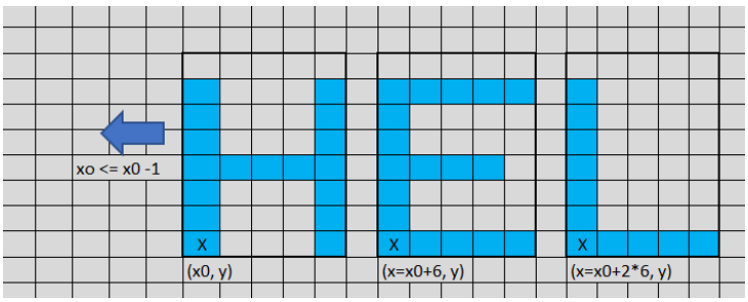



FIGURE 3.11 – Image PPM

Pour tester le bon fonctionnement de ce module, nous utilisons encore une fois des fichiers PPM que nous avons généré à trois moments différents :


Figure 3.12 – Image PPM

On observe bien l'évolution du message défilant.

### 3.1.5 Conclusion

Finalement, les messages statiques et défilants fonctionnent bien d'après les simulations et les fichiers PPM cependant, nous n'avons pas pu observer le même résultat sur la matrice. Cela est potentiellement dû au protocole de communication entre la RAM et la matrice.

## 3.2 L'effet Matrix

### 3.2.1 Présentation

L'effet Matrix consiste à faire défiler vers le bas sur chaque colonne un segment d'une certaine taille. De plus le début de l'affichage doit se faire à des temps différés. L'exemple suivant montre ce qui se passe entre 2 séquences sur notre matrice de LEDs.

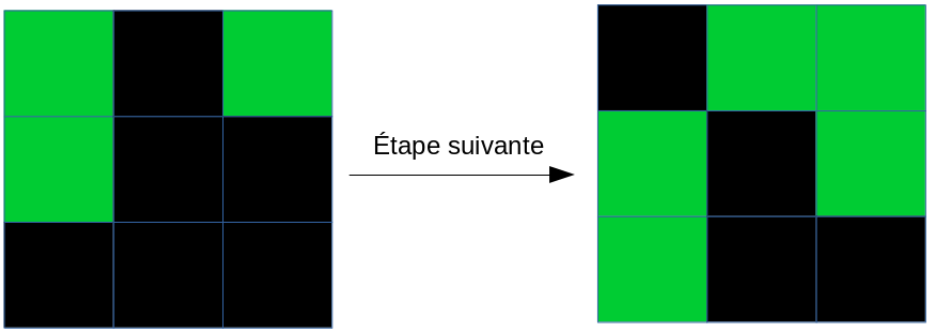



FIGURE 3.13 – Schéma explicatif du principe de l'effet Matrix

### 3.2.2 Module Matrix effect

Le Module "Matrix effect" possède 6 entrées et 6 sorties qui vont dans la RAM, comme on peut le voir sur le schéma suivant :


FIGURE 3.14 – Module Matrix effect

Entrées :
- clk_proc (1 bit) : Horloge permettent les calculs et l'écriture dans la RAM
- clk_step (1 bit) : horloge définissant la durée entre deux séquences
- Reset (1 bit) : remise à zéro du système
- R_in, G_in, B_in (8 bits) : signaux définissant la couleur des segments

Sorties :
- w_en (1 bit) : on peut écrire dans la RAM si ce bit est à '1'
- x_out, y_out (5 bits) : position du pixel à mettre à jour dans la RAM
- R_out, G_out, B_out (8 bits) : couleur du pixel à mettre à jour dans la RAM

### 3.2.3 Description des blocs

L'architecture globale de notre module est la suivante, mais on a en réalité 32 blocs Latency et col_matrix qui sont en parallèles :

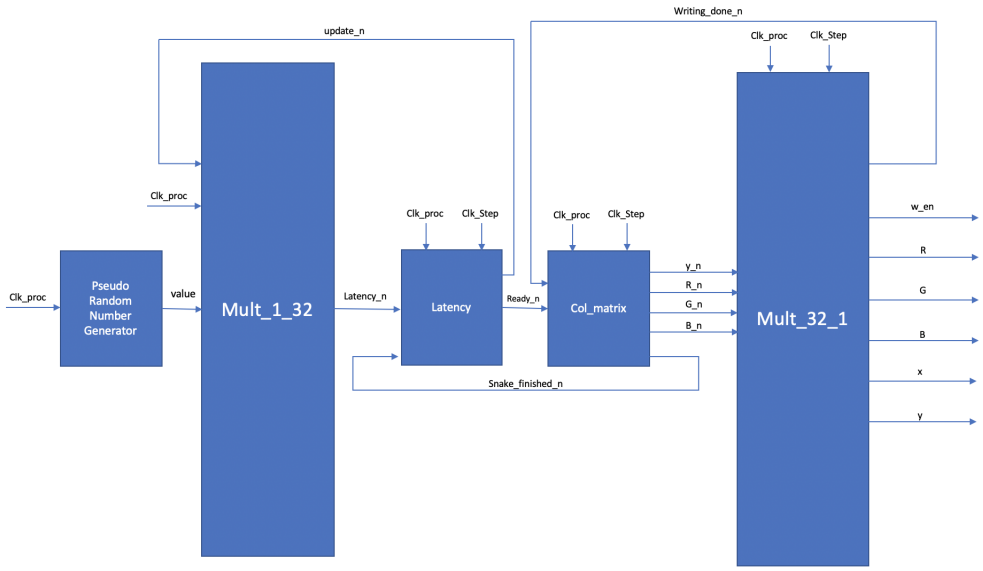



FIGURE 3.15 – Architecture globale

#### Générateur de nombre pseudo-aléatoire

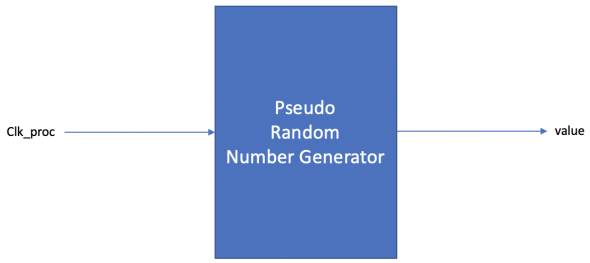



FIGURE 3.16 – Bloc du générateur de nombre pseudo-aléatoire

Le générateur de nombre pseudo-aléatoire permet comme son nom l'indique de génère un nombre aléatoire grâce à plusieurs portes logiques **xor** dans son programme, ce bloc m'a été fourni par Bertrand Le Gal. Ce bloc permet de générer une valeur dans une tranche de valeur choisi (entre 0 et 31 pour nous), et va générer un nombre différent à chaque front montant d'horloge.

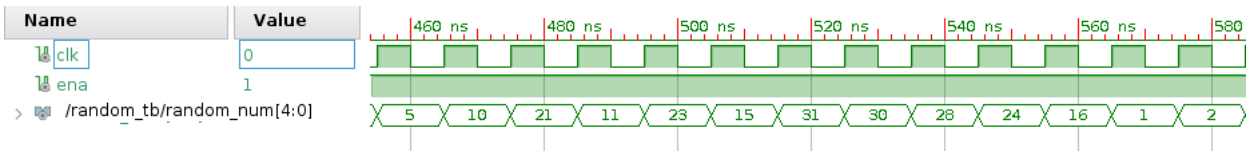



FIGURE 3.17 – Banc d'essai du générateur de nombre pseudo aléatoire

### Multiplexeur 1-32

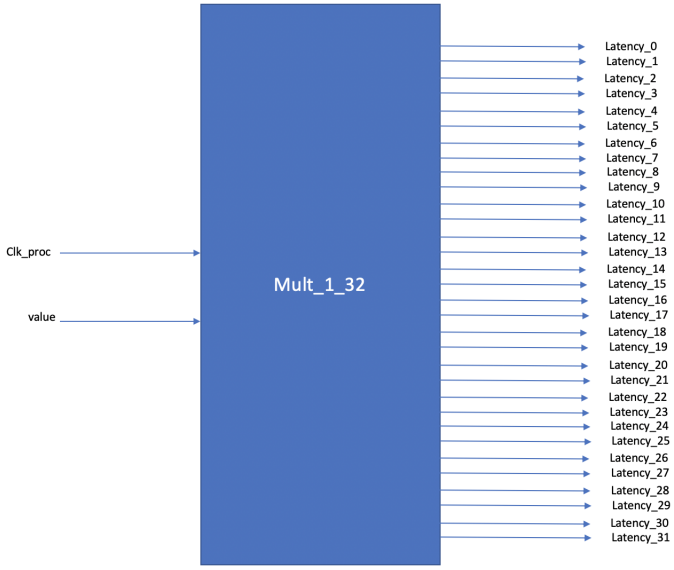



FIGURE 3.18 – Bloc du multiplexeur 1-32

On utilise un multiplexeur afin de réaliser l'initialisation de notre attente sur chaque colonne, car si l'on utilisait un bloc de générateur de nombre pseudo aléatoire pour chaque colonne, un même bloc à un même moment donnerait une même valeur. De ce fait, on utilise un multiplexeur qui a une valeur différente à chaque front montant d'horloge et qui charge une nouvelle valeur **value_n** seulement si **upgrade_n** est à '1' (le bloc latence a besoin d'une nouvelle valeur).

### Latence

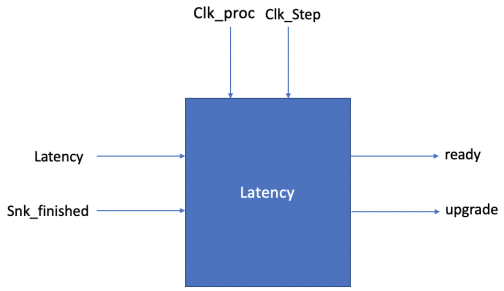



FIGURE 3.19 – Bloc pour la latence

Le bloc latence est propre à chaque colonne et permet d'attendre le temps du nombre d'étapes de la valeur récupérée à l'aide de son signal **upgrade**, en incrémentant un compteur à chaque fois que la valeur de **clk_step** est à '1', jusqu'à ce que le compteur atteigne la valeur récupérée. Puis, on signale au bloc suivant de commencer à faire défiler le segment. Cette étape recommencera à chaque fois que le serpent a fini de défiler.

### Calcul des mises à jour à faire dans la RAM

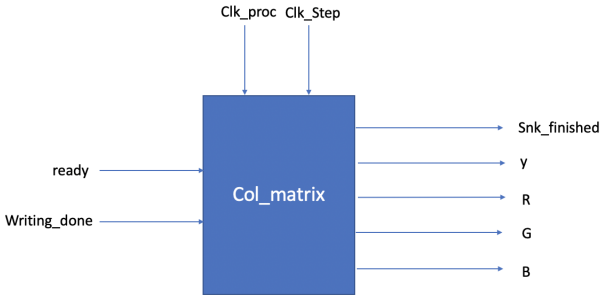



FIGURE 3.20 – Bloc réalisant les calculs de mise à jour de la RAM

Le calcul de ce que l'on va écrire dans la RAM pour chaque colonne à chaque étape se fait dans ce bloc. A chaque étape, on peut avoir au maximum deux changements dans la RAM :

- Allumer la prochaine LED de notre segment
- Eteindre la dernière LED de notre segment

On va avoir deux compteurs l'un suivant la LED que l'on allume et l'autre suivant la LED que l'on éteint. Pour chaque étape, on envoie dans un premier temps la position et la couleur de la LED à allumer. Puis, dès que l'écriture est faite dans la RAM, on calcule la position de la LED à éteindre et on la transmet. On a mis une taille de segment de 26 LEDs, ce qui donne pour une colonne les signaux suivants :

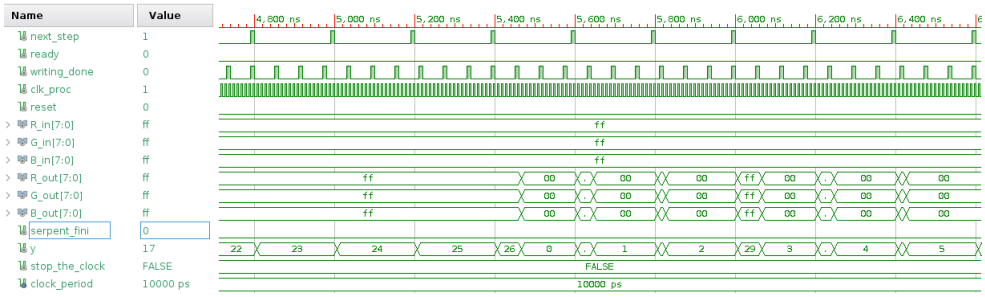



FIGURE 3.21 – Banc d'essai du calcul de position

**Multiplexeur 32-1**

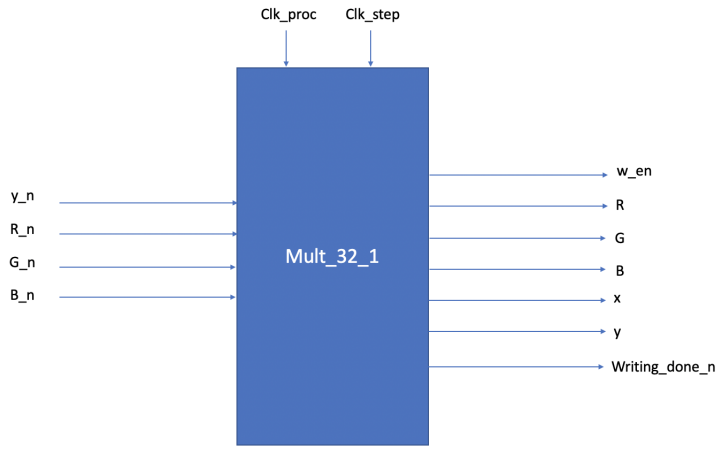



FIGURE 3.22 – Bloc du multiplexeur 1-32

L'écriture dans la mémoire ne peut se faire que de façon séquentielle, on ne peut écrire que dans une adresse à la fois. Cependant, en entrée on a 32 données à écrire à la fois dans la RAM. Afin de palier à ce problème, on utilise un multiplexeur 32-1 qui va envoyer une donnée différente à chaque front montant de l'horloge **clk_proc**.

Grâce à un compteur, on écrit une donnée d'une colonne après l'autre dans la RAM. Cette réalisation ne pouvant être au maximum que de 2 fois par colonne par étape, on limite l'écriture à seulement 64 écritures dans la RAM par étape.

### 3.2.4 Tests sur le module global

Les tests suivants montrent l'évolution des signaux entrants dans la RAM en sortie de notre module, en fonction de notre **clk_proc** et de notre **clk_step** pour une couleur verte de segment (R=00 ; G=FF ; B=00).

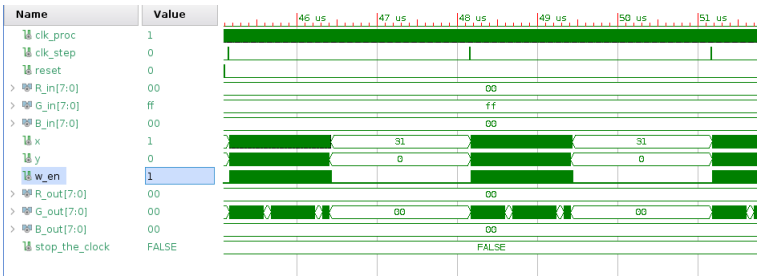



FIGURE 3.23 – Banc d'essai du multiplexeur 32-1 (1)

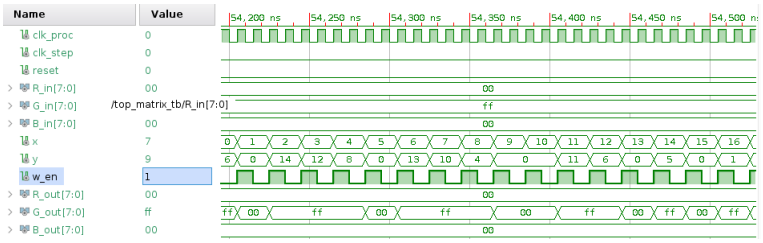



FIGURE 3.24 – Banc d'essai du multiplexeur 32-1 (2)

On peut voir que l'on écrit seulement $2 \times 32$ fois dans la RAM par étape. De plus, l'écriture dans la RAM se fait une colonne après l'autre. Puis les tests sur la matrice ont été concluant :

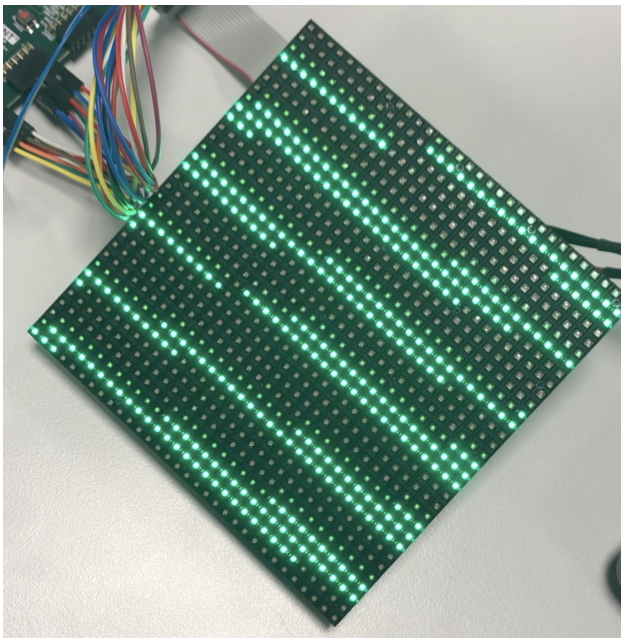


Figure 3.25 – Test de l'application

## 3.3 Tic Tac Toe

L'application Tic Tac Toe a pour objectif de jouer au morpion sur la matrice de LED. L'interaction se fait à l'aide des boutons poussoir présents sur la Nexys A7. Le bloc du jeu communique directement avec le bloc chargé du protocole et de l'affichage. Le schéma d'entrée/sortie du jeu est donc le suivant :

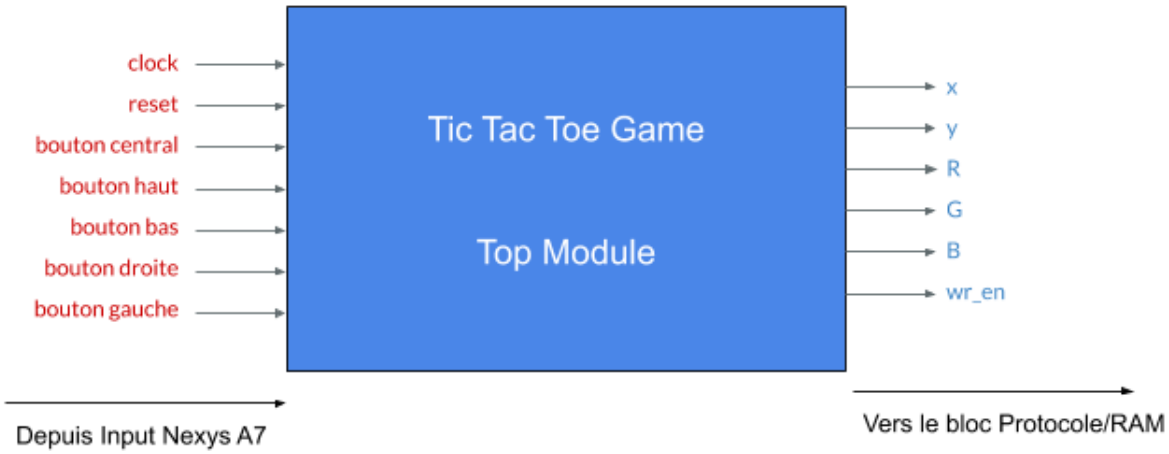



FIGURE 3.26 – Architecture générale

Le joueur sélectionne la case dans laquelle il souhaite se placer en appuyant sur les boutons, un carré blanc entourant la case signale l'emplacement du curseur. Lorsque le joueur presse le bouton central, une croix de couleur apparaît dans la case et c'est ensuite au tour de l'autre joueur.

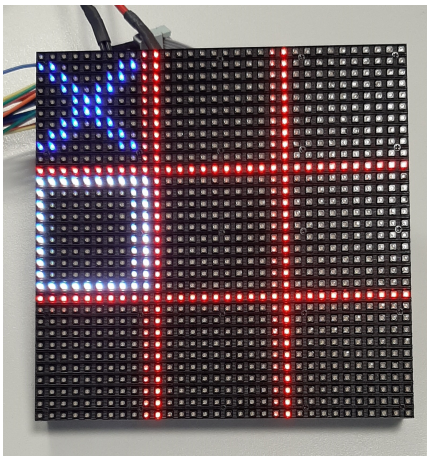


FIGURE 3.27 – Exemple d'affichage

### 3.3.1 Architecture

L'application est séparée en 3 blocs principaux :
- Un filtre anti-rebond pour chaque bouton
- Une machine d'état déterminant la position où le joueur souhaite se placer, s'il souhaite ou non valider son déplacement et la couleur du joueur
- Une machine d'état déterminant les pixels à modifier pour effectuer l'affichage

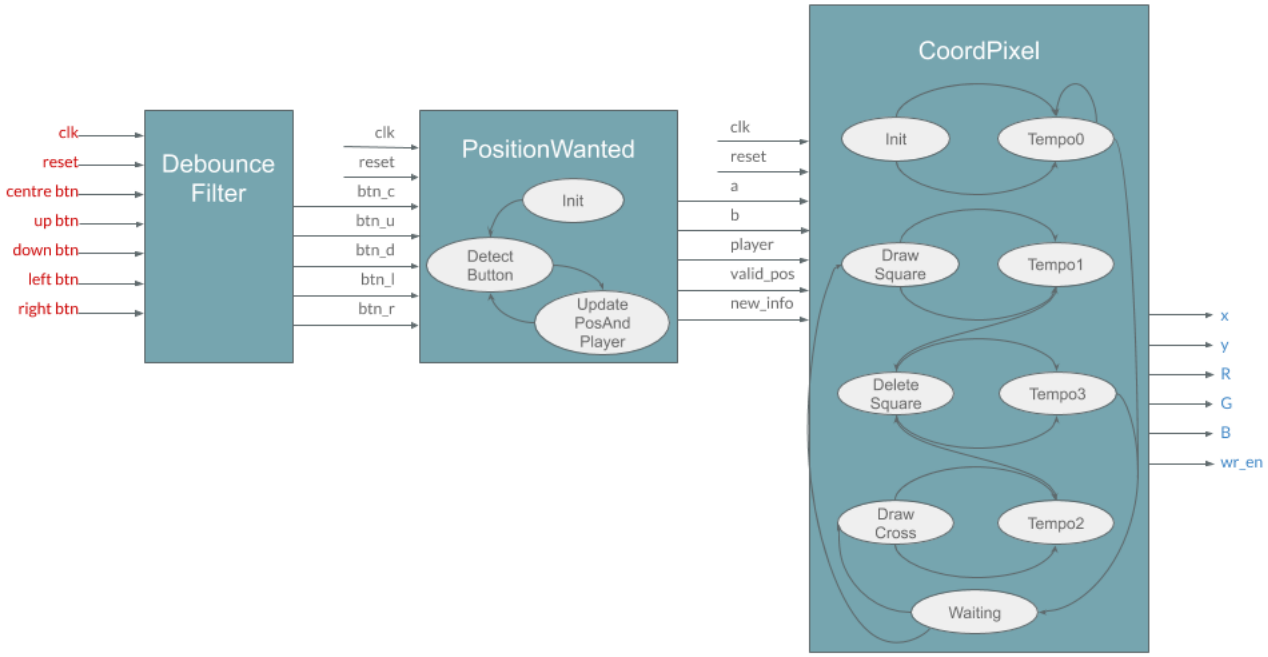



FIGURE 3.28 – Schéma bloc global de la solution retenue pour le TicTacToe

### 3.3.2 Initialisation

Une RAM contenant les valeurs des pixels de la matrice est préchargée dans le bloc du protocole de communication afin d'afficher une grille de 3x3 dont chaque case fait 10x10 pixels.

### 3.3.3 Filtre Anti-rebond

L'utilisateur interagit via les boutons poussoirs. Néanmoins, afin de ne pas traiter plusieurs fois l'information, il est nécessaire de ne détecter l'appuie que pendant un cycle d'horloge. C'est le rôle du bloc Debounce Filter qui est appliqué à chaque bouton en entrée.

### 3.3.4 Position, validation et joueur

Le bloc PosWanted récupère les information en provenance des boutons et interprète les actions. Si les boutons directionnels sont pressés, le curseur est déplacé, si le bouton central est activé, une croix de la couleur du joueur doit être dessinée.

A l'initialisation, le curseur est placé sur la case la plus en haut à gauche ( a = 0 et b = 0 ), le joueur n°1 joue et valid_pos est à 0. L'état suivant est DetectButton. Le bloc attend une nouvelle entrée. Lorsqu'un bouton est pressé, il est nécessaire de vérifier que le déplacement est possible ($0 \leq a < 3$ et $0 \leq b < 3$ ). Dans ce cas, new_info passe à 1 pendant un cycle d'horloge. Si le bouton central est pressé, a et b ne change pas mais valid_pos passe à 1 et le numéro du joueur est mis à jour dans l'état UpdataPosAndPlayer.

### 3.3.5 Gestion des pixels à afficher

Le bloc CoordPixel se charge d'écrire dans la mémoire les valeurs des pixels à modifier. On trouve dans ce bloc 2 listes stockant les coordonnées permettant de tracer un carré ou une croix : Depuis la position ( a,b ), les pixels à activer pour tracer un carré définis par le pseudo_code suivant : *ListSquareX = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]*

*ListSquareY = [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 8, 7, 6, 5, 4, 3, 2, 1 ]*
*Pour i allant de 0 à 35 faire*
*x = a + ListSquareX(i);*
*y = b + ListSquareY(i);*
*Attendre 100 cycles d'horloge;*
*Fin pour*

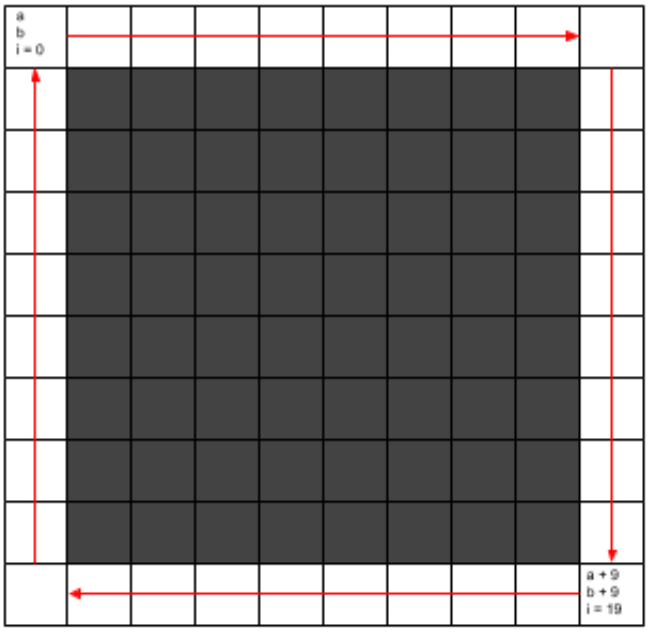



FIGURE 3.29 – Dessin d'un contour

Il en va de même pour dessiner une croix.

Afin de laisser le temps au bloc de lecture, seule une écriture tous les 100 cycle d'horloge est effectuée. C'est le rôle de l'état Tempo dans la machine d'état. Ce dernier se charge aussi de passer d'un état de dessin à l'autre puisque quand un carré curseur est dessiné, il est nécessaire de supprimer le précédent. Les coordonnées du curseur sont donc stockés entre 2 positions.

Un état spéciale est dédié à l'initialisation puisqu'un carré doit être dessiné sans qu'un précédent n'ait été tracé.

# Partie 4

# Conclusion

Pour conclure, ce projet nous a permis de comprendre et utiliser le protocole de communication avec la matrice. Nous avons créé un bloc VHDL permettant de réaliser cette communication très facilement. Pour autant, certains problèmes subsistent avec ce bloc.

Nous pouvons penser à quelque pistes d'amélioration. Notamment travailler sur le chaînage des matrices. En effet, pour le moment, lorsque l'on chaîne les matrices, elles sont toutes identiques. Nous pouvons penser à changer cela pour imaginer par exemple un message défilant sur un cube entier.

Une autre piste d'amélioration serait de rendre la RAM reliée à la matrice plus fiable afin de s'assurer que chaque valeurs écrite dans cette RAM s'affiche correctement sur la matrice.

Ce projet nous a aussi permis de comprendre à quel point il est important de documenter notre travail afin de le rendre accessible aux personnes voulant améliorer ou utiliser notre projet.

# Partie 5

# Sources

https ://github.com/attie/led_matrix_tinyfpga_a2/blob/master/doc/led_matrix_overview.md
http ://www.rayslogic.com/propeller/Programming/AdafruitRGB/AdafruitRGB.htm