



UNIVERSITÉ DE LIÈGE
FACULTÉ DE SCIENCES APPLIQUÉES

Projet Digital electronics

Dino LED

Auteurs

Florian PAGANO - s183627
Adrien VINDERS - s194594
Laurent PIETTE - s190703
François STRAET - s181703

Professeur, assistant
Jean-Michel REDOUTÉ
Thibaud PEERS

Cours de Bachelier en sciences de l'ingénieur
16 mai 2021

Table des matières

1 Objectif	2
1.1 Représentations	2
2 Composants	2
3 Mise en place pratique	3
4 Schéma électrique	4
5 Pin planner	4
6 Code	4
6.1 Entité	4
6.2 Architecture	5
6.2.1 Signaux	5
6.2.2 Processes	5
6.3 Nombres aléatoires	6
7 Communication	6
8 Annexes	7
8.1 Code VHDL	7
8.2 Implémentation des composant logiques dans le CPLD	10

1 Objectif

L'idée du projet est de faire, avec la matrice led, un jeu semblable au jeu du dinosaure qui apparaît sur Google Chrome lorsque qu'il n'y a pas de connexion.

Un dino se déplace, en devant éviter les obstacles qui apparaissent. Pour cela, il peut s'accroupir, et devenir ainsi plus petit, et sauter, ce qui le déplace vers le haut pour une durée fixe. Le score est le nombre d'obstacle que le joueur a réussi à éviter.

Lorsque le dino percute un obstacle, la partie se termine.

N.B. : Le score est représenté en binaire sur les LEDs du CPLD.

1.1 Représentations

- Le dino sera vu comme trois LEDs vertes. Les différentes positions du dino sont reprises à la figure 1.
- Les obstacles sont des ensembles de LEDs rouges. Les différentes configurations des différents obstacles disponibles sont reprises à la figure 2.

NB : les obstacles défilent de la droite vers la gauche et la position du dino reste statique sur la matrice LED

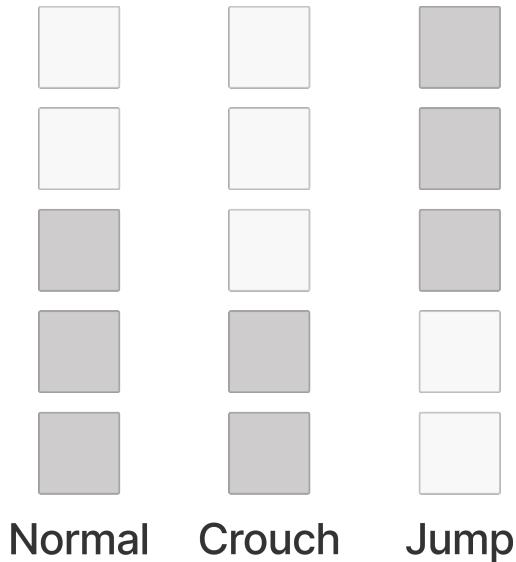


FIGURE 1: Positions du dino

2 Composants

- CPLD
- Pile 9V
- Matrice LED 5×7
- Boutons :
 - Reset
 - Saut
 - S'accroupir
- 8 LEDs pour afficher le score (sur la carte électronique à laquelle est fixée le CPLD)
- LED RGB

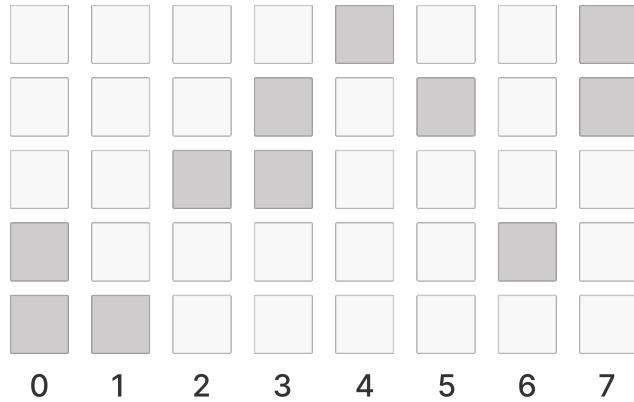


FIGURE 2: Différentes configurations des différents obstacles disponibles

- Rouge : partie perdue
- Bleue : partie en cours
- Verte : partie réinitialisée

3 Mise en place pratique

Une photo de la mise en place pratique des différents éléments est disponible à la figure 3, pour plus de précision quant aux connections entre les divers éléments, il est possible de se référer à la figure 4.

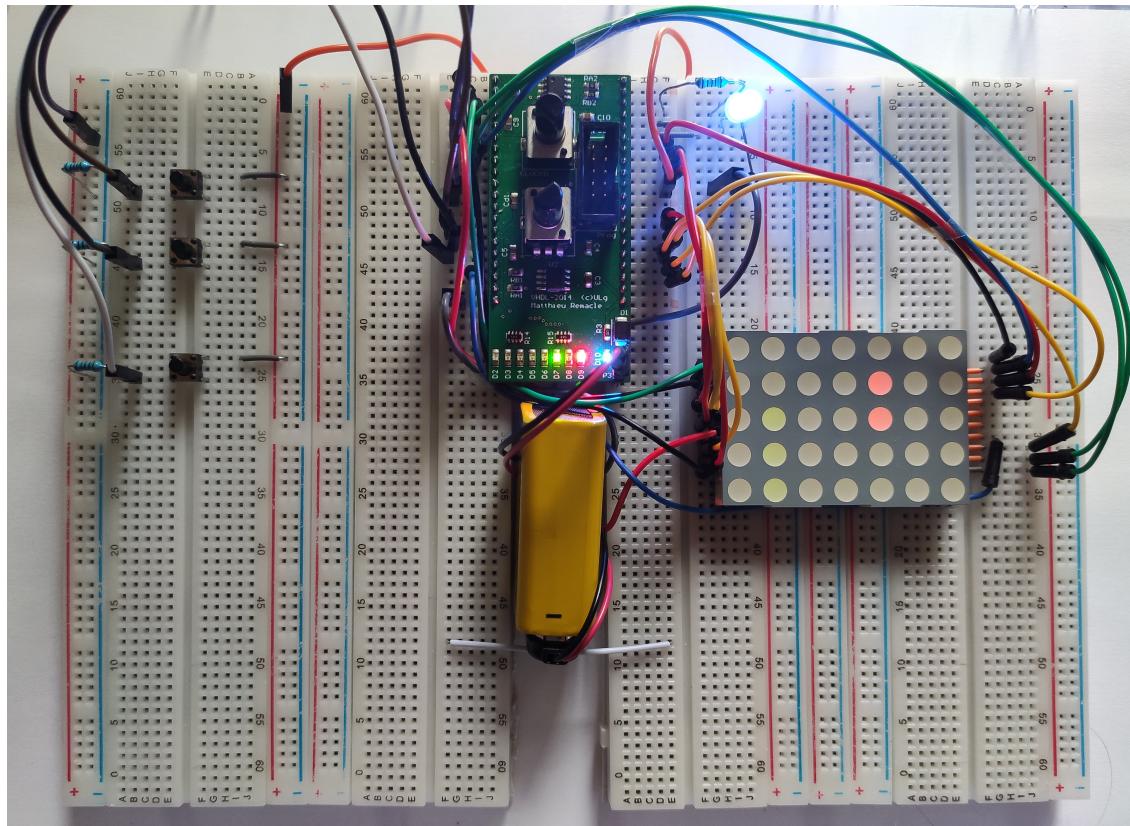


FIGURE 3: Mise en place pratique

4 Schéma électrique

Le schéma électrique est représenté à la figure 4.

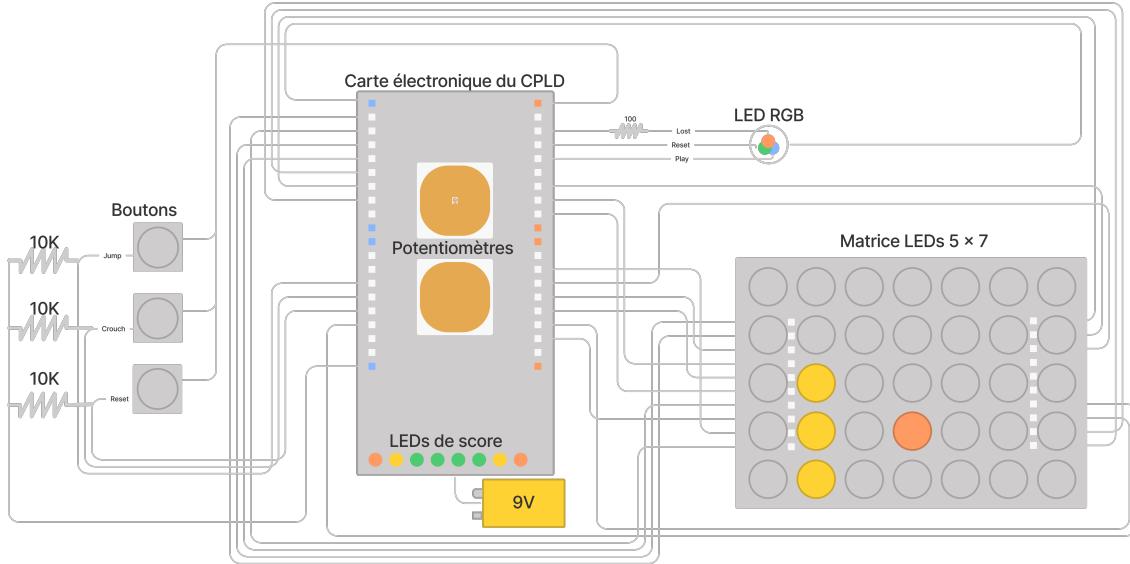


FIGURE 4: Schéma électrique

5 Pin planner

L'état du pin planner est montré à la figure 5.

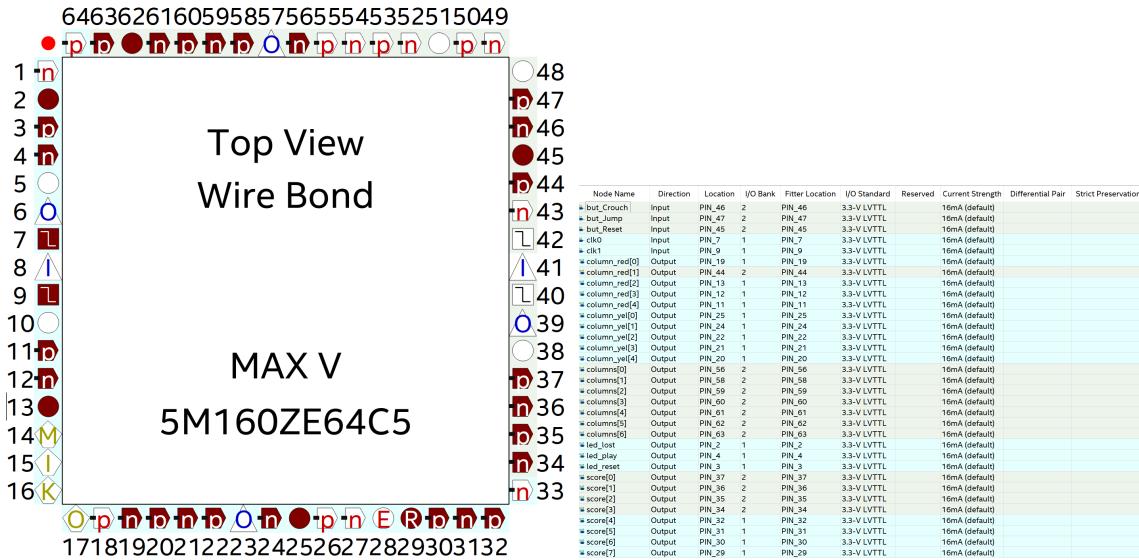


FIGURE 5: Pin planner

6 Code

6.1 Entité

Entrées

- `clk0` clock rapide
- `clk1` clock lente
- `but_Jump` bouton de saut
- `but_Crouch` bouton pour s'accroupir
- `but_Reset` bouton pour remettre la partie à zéro

Sorties

- `led_play`
- `led_lost`
- `let_reset`
- `lines` activation de chaque ligne de la matrice LED
- `column_red` activation de chaque colonne de la matrice LED, pour les LEDs rouges
- `column_yel` activation de chaque colonne de la matrice LED, pour les LEDs jaunes
- `score` le score, en tant que vecteur de booléens

6.2 Architecture

L'approche la plus naturelle dans notre cas est une *state machine* (machine à états finis), car nous pouvons retenir l'information relative aux positions des différents composants sur la matrice de LED, et les mettre à jour à chaque "pas de temps", c'est-à-dire lors de chaque *rising edge* des clocks.

6.2.1 Signaux

Les signaux sont repris à la table 1.

Nom du signal	Description
<code>cur_col</code>	L'index de la colonne de la matrice LED qui est actuellement rafraîchie
<code>vector_y</code>	Le vecteur d'activation de la colonne de LEDs jaunes où se trouve le dino
<code>vector_r</code>	Le vecteur d'activation de la colonne de LEDs rouges où se trouve l'obstacle
<code>lost</code>	Un booléen indiquant que la partie est perdue, et qu'il faut attendre le reset
<code>random</code>	Un entier aléatoire entre 0 et 7 généré grâce à la <code>cur_col</code>
<code>obst_pos</code>	La position de l'obstacle (l'index de la colonne où il se trouve)
<code>obst_cnt</code>	Un compteur, pour compter le nombre de <i>rising edges</i> de la clock du jeu pour le déplacement de l'obstacle
<code>nb_obst</code>	Un entier, le nombre d'obstacle que le dino a évité au cours de cette partie (directement lié à la sortie <code>score</code>)
<code>jmp_cnt</code>	Un compteur, pour compter le nombre de <i>rising edges</i> de la clock du jeu pour le temps d'un saut

TABLE 1: Description des signaux utilisés

6.2.2 Processes

Nous avons choisi de créer deux processus distincts, `game` et `display`, car la séparation est bien nette et sera utile pour générer des nombres aléatoires.

— Process game

Ce premier process est en charge des signaux, entrées et sorties liées au jeu lui-même et à l'application des règles, modifications des positions du dino et des obstacles.

À chaque *rising edge* de la clock lente (`clk1`), ce process réalisera les mises à jour suivantes :

- Si `lost` vaut '0', c'est-à-dire que la partie est toujours en cours :
 - Si le compteur de saut, `jmp_cnt`, est nul et que le bouton de saut est appuyé, alors on commence un saut : on assigne 40 à `jmp_cnt` et on affiche le dino en position de saut.
 - Si le compteur de saut est égal à un : remettre le dino en position normale
 - Si le bouton pour s'accroupir est enfoncé, on force `jmp_cnt` à 1, de sorte qu'au *rising edge* suivant, le dino sera affiché en position normale. En plus, il faut afficher le dino en position basse.
 - Incrémenter les compteurs :
 - `jmp_cnt` si il est non nul, vers le bas
 - `obst_cnt`, vers le bas, s'il est nul, le remettre à 10.
 - Si le compteur d'obstacle est nul : déplacer l'obstacle. Si ce dernier arrive à la dernière case, mettre un nouvel obstacle sur la première case.
- Si `but_Reset` est enfoncé, remettre tout à zéro
- Si aucun nombre aléatoire n'est dans `random` (= 7), alors on assigne à `random` la valeur de `cur_col`.
- **Process display**
 Ce process est responsable de l'affichage du dino et des obstacles.
 À chaque *rising edge* de la clock rapide (`clk0`), ce process réalisera les actions suivantes :
 - Augmenter `cur_col` de 1, si égal à 6, le remettre à 0.
 - Assigner à `column_red` et `column_yel` les valeurs d'activations liées à la `cur_col`-ième ligne.
 - Assigner à `columns` les valeurs d'activation de chacune des colonnes.

6.3 Nombres aléatoires

Pour générer des nombres aléatoires, nous avons utilisé la méthode qui nous a été proposée, mais, comme l'utilisateur n'appuie pas systématiquement sur un bouton lors du passage d'un obstacle, nous sommes "à court" d'opportunités de nombres aléatoires.

La logique derrière la génération des valeurs de `random` est la suivante :

- La valeur 7 signifie que la valeur actuelle a été "consommée". N.B. : par précaution, nous avons aussi défini un obstacle correspondant à cette valeur.
- Si un des boutons `but_Jump` ou `but_Crouch` est enfoncé, alors on assigne la valeur de `cur_col` à `random`, de sorte que la valeur qui restera est soit :
 - Celle qu'avait `cur_col` lorsque le bouton de saut a été enfoncé (à cause du mécanisme de saut)
 - Celle qu'avait `cur_col` lorsque le bouton pour s'accroupir a été relâché
- Si on observe que la valeur de `random` est 7 (car aucun bouton n'a été enfoncé), alors on lui assigne la valeur actuelle de `cur_col`.

Comme nous avons utilisé deux clocks différentes, il devrait être possible d'observer des patterns d'apparition d'obstacles si aucun bouton n'est enfoncé. Mais cela ne posera pas de problèmes, vu qu'il est très probable qu'il faille utiliser de bouton pour éviter les obstacles ;-D

7 Communication

Nous avons utilisé divers moyens de communication tout au long de ce projet. Nous avons utilisé GitHub pour la transmission et la sauvegarde du code. Discord, pour la communication aussi bien par message, que par vidéo. Pour la rédaction du rapport, nous nous sommes servis d'Overleaf pour le traitement de texte, ainsi que de Figma pour réaliser différentes figures.

8 Annexes

8.1 Code VHDL

```

library ieee ;
use ieee.std_logic_1164.all ;
use ieee.numeric_std.all;

entity display is
  port
  ( — input ports
    clk0          : in std_logic; — quicker clock
    clk1          : in std_logic; — slower clock
    but_Jump      : in std_logic; — Jump button
    but_Crouch    : in std_logic; — Crouch button
    but_Reset     : in std_logic; — Reset button

    — output ports
    led_play       : out std_logic; — led saying that we play
    led_lost       : out std_logic; — led saying that we lost the game
    led_reset      : out std_logic; — led saying that we reset the game
    columns        : out std_logic_vector(0 to 6); — Led matrix
    column_red    : out std_logic_vector(0 to 4); — Led matrix
    column_yel    : out std_logic_vector(0 to 4); — Led matrix
    score          : out std_logic_vector(0 to 7)); — nbr of obstacle avoided
end entity display ;

architecture display_arch of display is

  signal cur_col      : integer range 0 to 6 := 0;
  — current column displayed by display
  signal vector_r     : std_logic_vector(0 to 4) := "11111";
  — vector of red leds (0 = lit)
  signal vector_y     : std_logic_vector(0 to 4) := "00011";
  — vector of yellow leds
  signal jmp_cnt      : integer range 0 to 40 := 0;
  — counter of jumps
  signal lost          : std_logic := '0';
  — saying that we have lost
  signal random        : integer range 0 to 7 := 7;
  — random generated integer
  signal obst_pos      : integer range 0 to 6 := 0;
  — position of the obstacle on the column
  signal obst_cnt      : integer range 0 to 10 := 0;
  — count of obstacle
  signal nb_obst        : integer range 0 to 255 := 0;
  — number of obstacle generated

begin
  game : process( clk1 )
  begin
    if( rising_edge( clk1 ) ) then
      if (lost = '0') then — Game is on
        led_play <= '1';

```

```

led_lost <= '0';
led_reset <= '0';

— JUMP BUTTON
if (jmp_cnt = 0) then
  if (but_Jump = '1') then
    random <= cur_col; — compute the random number

    — display the player in the 3 upper leds (we jump)
    jmp_cnt <= 40;
    vector_y <= "11000";
  end if;
else
  if (jmp_cnt = 1) then
    vector_y <= "00011";
  end if;
  jmp_cnt <= jmp_cnt - 1;
end if;

if (but_Crouch = '1') then
  random <= cur_col; — compute the random number

  if (jmp_cnt /= 0) then
    jmp_cnt <= 1;
  end if;

  vector_y(2) <= '1'; — turn off the 3rd upper led
else
  vector_y(2) <= '0';
end if;

— OBSTACLE MOVE AND GESTION
if (obst_cnt = 0) then
  obst_cnt <= 10;

  — we generate a new obstacle when the previous one has cross the matrix
  if (obst_pos = 6) then
    obst_pos <= 0;

  — different obstacles
  case random is
    when 0 => vector_r <= "00111";
    when 1 => vector_r <= "01111";
    when 2 => vector_r <= "11011";
    when 3 => vector_r <= "11001";
    when 4 => vector_r <= "11110";
    when 5 => vector_r <= "11101";
    when 6 => vector_r <= "10111";
    when 7 => vector_r <= "11100"; — should not happen
  end case ;

  random <= 7;
  nb_obst <= nb_obst + 1 ;
  score <= std_logic_vector(to_unsigned(nb_obst, score'length));

```

```

        else
            obst_pos <= obst_pos + 1;
        end if ;
    else
        obst_cnt <= obst_cnt - 1;
    end if;

    — CHECK IF WE HIT AN OBSTACLE
    if (obst_pos = 5) then
        for r in 0 to 4 loop
            if (vector_y(r) = '0' and vector_r(r) = '0') then
                lost <= '1';
                vector_y <= "11111";
                vector_r <= "00000";
            end if ;
        end loop ;
    end if ;
    else
        led_play <= '0';
        led_lost <= '1';
    end if ; — end if (lost = '0')

    if (but_Reset = '1') then
        lost <= '0';
        score <= "00000000";
        nb_obst <= 0;
        obst_cnt <= 0;
        jmp_cnt <= 0;

        vector_y <= "00011";
        vector_r <= "11111";
        led_reset <= '1';
        led_play <= '0';
    end if ;

    — RANDOM NUMBER GENERATION IF WE DO NOT TOUCH BUTTONS
    if (random = 7) then
        random <= cur_col;
    end if ;
    end if ;
end process game ;

— DISPLAY THE LED MATRIX
display : process ( clk0 ) — clk0 : quicker clock
begin
    if ( rising_edge( clk0 ) ) then

        if (cur_col = 6) then
            cur_col <= 0 ;
        else
            cur_col <= cur_col + 1 ;
        end if ;

```

```

    if (cur_col = 5) then
        column_yel <= vector_y;
    else
        column_yel <= "11111";
    end if ;

    if (cur_col = obst_pos) then
        column_red <= vector_r;
    else
        column_red <= "11111";
    end if ;

    case cur_col is
        when 0 => columns <= "1000000" ;
        when 1 => columns <= "0100000" ;
        when 2 => columns <= "0010000" ;
        when 3 => columns <= "0001000" ;
        when 4 => columns <= "0000100" ;
        when 5 => columns <= "0000010" ;
        when 6 => columns <= "0000001" ;
    end case ;

end if ;

end process display ;

end architecture display_arch ;

```

8.2 Implémentation des composant logiques dans le CPLD

Le schéma des portes logiques implémentées dans le cpld est représenté à la figure 6, il est également disponible dans le dossier du projet, afin de bénéficier d'une meilleure visibilité.

NB : 102 éléments logique des 160 disponibles sont utilisés.

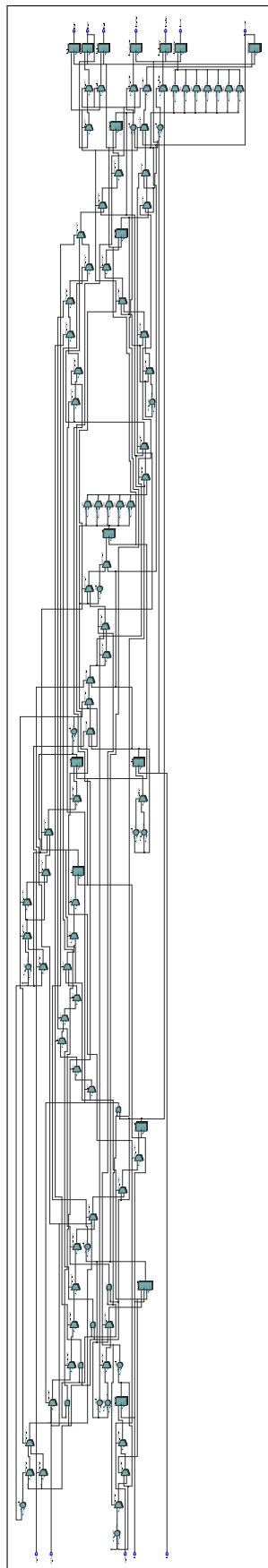


FIGURE 6: RTL viewer