ÉCOLE NATİONALE DES SCİENCES APPLİQUÉES DE TETOUAN

GCSE2
2025-2026

Gestion Parking Intelligent

VHDL

Présenté par :

KUNAKA DANIEL
MAKRI YOUSSRA
SIMPORE TAOBATA
SABOR LAILA
ABOUL-MOUMOUNI DIALLO
BOUARRAF DOHA
EL BAROUDI MALAK
MOUHAFID HAFSA

Professeur : JAMAL ZBITOU

Module : Programmation des circuits FPGA


P
PARKING
Places disponibles : 65

SOMMAIRE

Introduction

Dans le cadre du module FPGA, notre projet consiste à concevoir un système intelligent et automatisé de gestion de parking, basé sur une carte FPGA et programmé en langage VHDL.

Ce système répond à un besoin réel : automatiser le fonctionnement des parkings modernes afin d'assurer une meilleure gestion du flux de véhicules, réduire les interventions humaines et optimiser l'utilisation des places disponibles.

Le système que nous développons doit être capable de :

- détecter automatiquement l'entrée et la sortie des véhicules,
- compter en temps réel le nombre de places disponibles,
- afficher ce nombre sur un afficheur 7-segments,
- contrôler automatiquement une barrière d'accès selon l'état du parking (places disponibles ou parking plein).

Ce projet combine plusieurs notions essentielles telles que la conception numérique, la modularité en VHDL, la simulation via testbench. Il constitue une application concrète des systèmes embarqués et des circuits logiques programmables.

PARTIE 1:

Présentation du Cahier des charges

Spécifications fonctionnelles

Le système doit répondre à quatre grandes exigences fonctionnelles :

1. **Comptage automatique des véhicules**
   → Entrée : -1 place | Sortie : +1 place
2. **Affichage en temps réel du nombre de places disponibles**
3. **Contrôle automatique de la barrière d'accès**
4. **Interdiction d'entrée si le parking est plein**

Contraintes matérielles

Le système doit fonctionner avec des composants bien précis :

Capteurs d'entrée/sortie véhicule

- Deux capteurs pour détecter les véhicules : un à l'entrée (voiture_entree), un à la sortie (voiture_sortie).

Capteurs de position et sécurité barrière

- sensor_open_limit → barrière complètement levée
- sensor_closed_limit → barrière complètement baissée
- sensor_passage → capteur IR ou ultrason sous la barrière
  → Interdit toute commande de fermeture tant qu'un véhicule est dessous

Actionneur de barrière

- 2 sorties pour commander le moteur de la barrière
  → motor_open et motor_close (jamais actifs en même temps)
  → pilotées via un pont en H L298N ou un servomoteur SG90

Afficheur 7-segments 4 digits multiplexé

- Afficheur 7-segments 4 digits multiplexé
  → 7 bits pour les segments (a à g)
  → 4 bits pour sélectionner le digit actif

Carte FPGA

- Horloge système à 50 MHz
- Bouton de reset

Diagramme fonctionnel

Le système est organisé de façon hiérarchique et modulaire autour de quatre blocs principaux :

1.Bloc Compteur de places → gère le nombre de places libres (incrémentation/décrémentation sécurisée)

2.Bloc Contrôle de barrière → machine à états qui pilote l'ouverture/fermeture en fonction des capteurs et du signal d'autorisation

3.Bloc Affichage 7 segments → convertit le nombre de places en signaux segments pour l'affichage multiplexé

4.Bloc Principal (top_parking) → module de plus haut niveau qui interconnecte les trois blocs précédents, contient le générique MAX_PLACES et génère la logique globale d'autorisation d'ouverture

Les flux de données circulent comme suit :

- Les signaux bruts des capteurs arrivent dans le top
- Le top décide si oui ou non il faut ouvrir la barrière
- Le compteur est mis à jour uniquement sur front valide d'entrée ou de sortie
- Le nombre de places est envoyé en continu à l'afficheur

Règles de gestion

1.Ouverture de la barrière

- Pour une entrée → voiture_entree = ‘1’ AND places_disponibles > 0
- Pour une sortie → voiture_sortie = ‘1’ (toujours autorisée, même si plein) → Cela évite le deadlock classique : si le parking est plein, on peut toujours sortir !

2.Fermeture automatique

- Dès que le véhicule a complètement franchi la barrière (sensor_passage retombe à ‘0’ après avoir été activé) et que la barrière est en position ouverte, on commande la fermeture.

3.Sécurité absolue

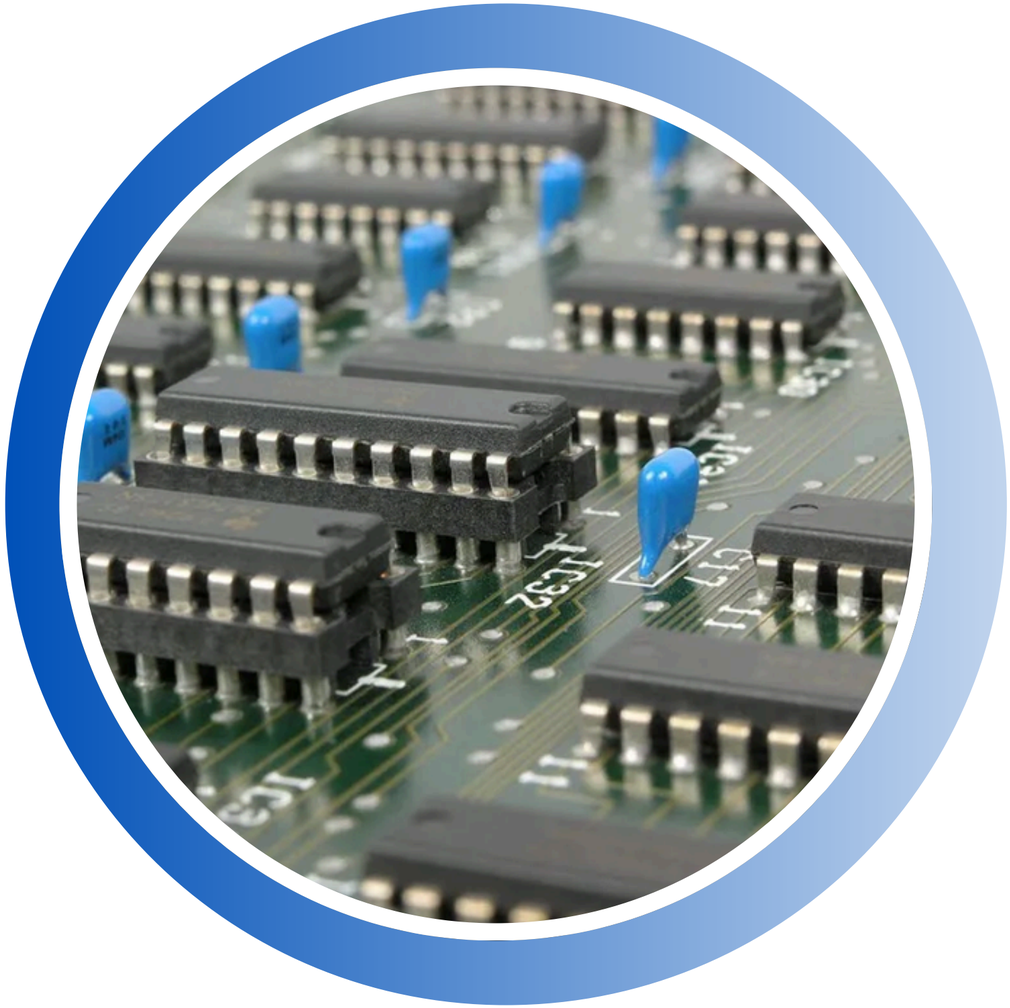
- Il est strictement interdit de fermer la barrière tant qu’un véhicule est détecté dessous (sensor_passage = ‘1’)

Cela garantit qu’on ne peut jamais écraser ou coincer un véhicule.

↳ Ces règles sont implémentées à la fois dans le module top_parking (logique d’autorisation) et dans le module de contrôle de barrière (machine à états sécurisée).

PARTIE 2:

Module Compteur de places



Module compteur de places – Gestion du nombre de places disponibles

Le module compteur de places joue un rôle central dans le système de gestion intelligent de parking. Il garde en mémoire le nombre de places disponibles et le met à jour en temps réel :

- Il décrémente lorsqu'une voiture entre .
- Incrémente lorsqu'une voiture sort.
- Il garantit que le compteur ne dépasse jamais la capacité maximale ni ne descend en dessous de zéro.

↳ Sa sortie fournit l'information essentielle aux autres modules, notamment pour l'affichage du nombre de places libres et pour la décision d'ouverture ou de fermeture de la barrière.

Logique du fonctionnement

- **Initialisation (reset synchrone)**

Au front d'horloge (Montant), si rst = '1', le compteur est remis à la capacité maximale (99 places).

- **Entrée d'une voiture**

Si voiture_entree = '1' et compteur > 0 → le compteur diminue de 1.

- **Sortie d'une voiture**

Si voiture_sortie = '1' et compteur < MAX_PLACES → le compteur augmente de 1.

- **Cas simultané**

Si voiture_entree = '1' et voiture_sortie = '1' en même temps → le compteur reste inchangé (événements annulés).

- **Sortie du module**

La valeur du compteur est envoyée en std_logic_vector pour être utilisée par :

- l'afficheur (7 segments ou LCD),
- le module barrière (autoriser/refuser l'entrée).

Interface du module

Nom du signal	Type	Direction	Rôle dans le système
clk	std_logic	Entrée	Horloge du système. Elle synchronise toutes les opérations du module.
rst	std_logic	Entrée	remet le compteur à sa valeur maximale
voiture_entree	std_logic	Entrée	Impulsion envoyée quand une voiture entre dans le parking.
voiture_sortie	std_logic	Entrée	Impulsion envoyée quand une voiture sort du parking.
nb_places_dispo	std_logic_vector(6 downto 0)	Sortie	Nombre de places disponibles, convertit en vecteur logique pour affichage ou décision

Explication du code VHDL

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;   -- mieux pour les conversions
```

Les bibliothèques

L'en-tête du code importe les bibliothèques IEEE nécessaires. STD_LOGIC_1164 définit les types logiques et leurs opérations, tandis que NUMERIC_STD fournit les conversions et calculs numériques

Explication du code VHDL

```vhdl
entity compteur_places is
    generic(
        MAX_PLACES : integer := 99  -- Capacité maximale du parking
    );
    port(
        clk       : in  std_logic;      -- Horloge
        rst       : in  std_logic;      -- Reset synchrone
        voiture_entree : in std_logic;  -- Signal d'entrée d'une voiture
        voiture_sortie : in std_logic;  -- Signal de sortie d'une voiture
        nb_places_dispo : out std_logic_vector(6 downto 0)  -- Nombre de places disponibles
    );
end compteur_places;
```

Entity

- **Generic MAX_PLACES:** Paramètre configurable de capacité (par défaut 99). On peut le modifier sans toucher au code interne.
- **Port clk (in)**: Horloge du système; toutes les actions sont synchronisées dessus.
- **Port rst (in):** Signal de réinitialion
- **Port voiture_entree (in)**: Impulsion quand une voiture entre; une seule période d'horloge idéalement.
- **Port voiture_sortie (in):** Impulsion quand une voiture sort; une seule période d'horloge idéalement.
- **Port nb_places_dispo (out)**: Valeur exportée en vecteur pour affichage/décision (7 bits suffisent pour 0–100).

Explication du code VHDL

```
architecture Behavioral of compteur_places is
    signal compteur : integer range 0 to MAX_PLACES := MAX_PLACES;
begin
```

Déclaration interne et initialisation

- Architecture Behavioral: Bloc qui décrit le comportement logique du module , sans détailler la structure physique des composants internes
- Le signal compteur est comme une petite mémoire qui garde le nombre de places libres. Il est borné entre 0 et la capacité maximale pour éviter les erreurs, et au démarrage il est initialisé à MAX_PLACES, ce qui correspond à un parking vide avec toutes les places disponibles.

Explication du code VHDL

```vhdl
 process(clk)
begin
    if rising_edge(clk) then
        -- Reset synchrone
        if rst = '1' then
            compteur <= MAX_PLACES;

        -- Cas simultané : entrée et sortie en même temps → pas de changement
        elsif voiture_entree = '1' and voiture_sortie = '1' then
            compteur <= compteur;

        -- Sortie seule : incrémentation si pas déjà au max
        elsif voiture_sortie = '1' and compteur < MAX_PLACES then
            compteur <= compteur + 1;

        -- Entrée seule : décrémentation si pas déjà à 0
        elsif voiture_entree = '1' and compteur > 0 then
            compteur <= compteur - 1;
        end if;
    end if;
end process;
```

Processus de comptage et reset

Mon process est cadencé par l'horloge. Au reset, le compteur revient à la capacité maximale. Si une voiture entre, on décrémente; si une voiture sort, on incrémente; et si les deux arrivent en même temps, on ne change rien.
L'ordre des conditions garantit qu'une seule action est exécutée par cycle.

Explication du code VHDL

```
    nb_places_dispo <= std_logic_vector(to_unsigned(compteur, 7));  -- 7 bits pour 0-100

end Behavioral;
```

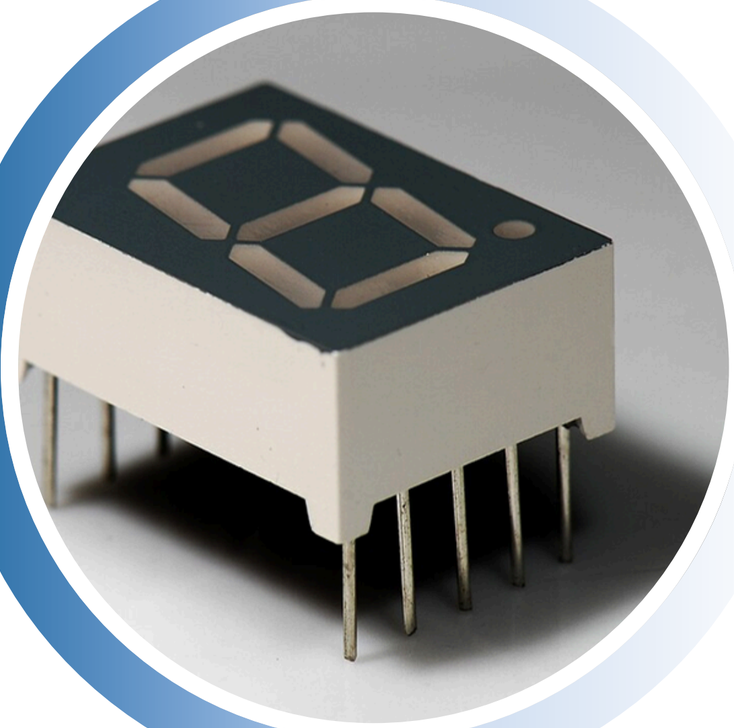
Conversion de la valeur vers la sortie binaire

- J'ai convertis l'integer en unsigned sur 7 bits , 7 bits car 2^7=128 couvre 0–99.
- J'ai convertis unsigned en std_logic_vector pour compatibilité avec les ports et les afficheurs
- J'ai gardé la logique interne en integer (simple, sûre) et ne convertis qu'en sortie pour l'affichage (7 segments)

PARTIE 3:

Module affichage 7 segments

Module Affichage 7 Segments



Son rôle est de **montrer en temps réel le nombre de places disponibles à l'utilisateur**.

Objectifs

- Afficher une valeur comprise entre 0 et 99
- Utiliser un afficheur à cathodes communes avec segments actifs haut.
- Masquer le zéro en tête
- Assurer un affichage stable et lisible grâce au multiplexage.

Axes de fonctionnement

1 Conversion binaire → BCD

Le compteur fournit une valeur binaire **(count_in).**
Le module transforme cette valeur en deux chiffres BCD : **unités et dizaines.**

2 Multiplexage

- Pendant **1 ms → affiche unités.**
- Pendant **1 ms → affiche dizaines.**
- **L'œil humain croit** que les deux sont **allumés en même temps.**

3 Masquage du zéro en tête

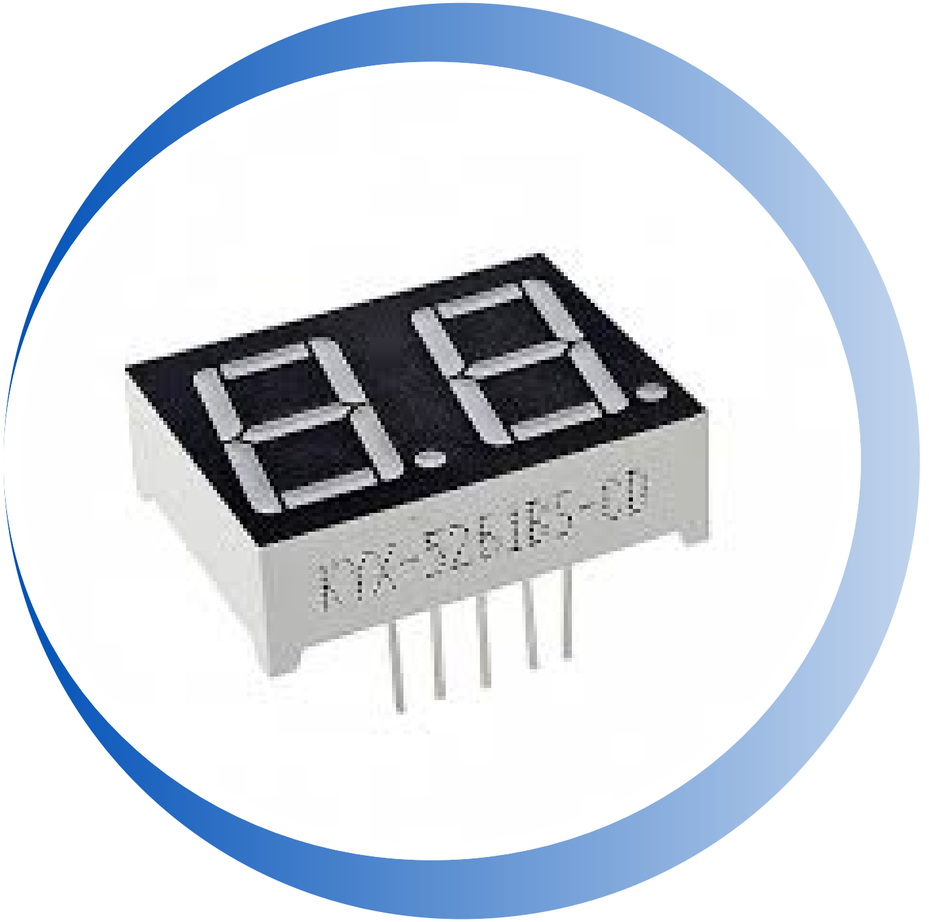
Si la valeur est **< 10**, on **n'affiche pas la dizaine**

4 Décodage BCD → segments

Chaque chiffre BCD est traduit en segments allumés.

- Exemple : **BCD "0010" → affiche "2"** en allumant les segments **a, d, e, g, c.**

Explication du code VHDL



le contrôleur principal de l'afficheur 7 segments.

ENTITY

```
entity sevenseg_ctrl is
    Port (
        clk      : in  STD_LOGIC;              -- horloge 50 MHz
        rst      : in  STD_LOGIC;              -- reset actif à 1
        count_in : in  UNSIGNED(6 downto 0);   -- valeur à afficher (0..99)
        cathode  : out STD_LOGIC_VECTOR(3 downto 0); -- cathodes (actif à 1)
        seg      : out STD_LOGIC_VECTOR(6 downto 0)  -- segments (actif à 1)
    );
end sevenseg_ctrl;
```

Entrées :

clk : horloge du FPGA (50 MHz).

rst : reset actif haut.

count_in : valeur binaire (0 à 99) à afficher.

Sorties :

cathode : sélectionne quel digit est actif (unités ou dizaines).

seg : contrôle les 7 segments (a–g).

Signaux internes

```
signal refresh_cnt : UNSIGNED(15 downto 0) := (others => '0');
signal digit_sel   : STD_LOGIC;
signal d_units     : STD_LOGIC_VECTOR(3 downto 0);
signal d_tens      : STD_LOGIC_VECTOR(3 downto 0);
signal current_bcd : STD_LOGIC_VECTOR(3 downto 0);
```

- **refresh_cnt :** compteur pour gérer la vitesse du multiplexage.
- **digit_sel :** bit qui alterne entre unités et dizaines.
- **d_units :** chiffre des unités en BCD.
- **d_tens :** chiffre des dizaines en BCD.
- **current_bcd :** chiffre actuellement affiché (selon digit_sel).

le contrôleur principal de l'afficheur 7 segments.

Bloc 1 – Compteur de rafraîchissement

```
process(clk, rst)
begin
    if rst = '1' then
        refresh_cnt <= (others => '0');
    elsif rising_edge(clk) then
        refresh_cnt <= refresh_cnt + 1;
    end if;
end process;
digit_sel <= refresh_cnt(15);
```

process(clk, rst)
On crée un processus qui réagit quand l'horloge (clk) change ou quand le reset (rst) est activé.
if rst = '1' then
Si le reset est activé, on remet le compteur refresh_cnt à zéro.
Sinon, à chaque front montant de l'horloge (50 MHz), on exécute le code suivant
refresh_cnt <= refresh_cnt + 1;
Le compteur refresh_cnt s'incrémente de 1 à chaque cycle d'horloge.
digit_sel <= refresh_cnt(15)
On prend le bit 15 du compteur pour décider quel digit afficher :
Si digit_sel = 0 → unités.
Si digit_sel = 1 → dizaines.

Bloc 2 – Conversion binaire → BCD

```
process(count_in)
    variable v : integer range 0 to 99;
    variable u, t : integer range 0 to 9;
begin
    v := to_integer(count_in);
    if v > 99 then v := 99; end if;
    u := v mod 10;
    t := (v / 10) mod 10;
    d_units <= STD_LOGIC_VECTOR(to_unsigned(u, 4));
    d_tens  <= STD_LOGIC_VECTOR(to_unsigned(t, 4));
end process;
```

- Ce bloc s'exécute dès que **count_in** change.
- On crée une variable entière pour stocker la valeur.
- On déclare deux variables locales dans le process : **u et t**
- On convertit la valeur binaire en entier.
- On limite à 99 pour éviter les erreurs.
- On calcule les unités (reste de la division par 10).
- On stocke ces chiffres en BCD (4 bits chacun).

le contrôleur principal de l'afficheur 7 segments.

Bloc 3 – Sélection du digit + masquage du zéro

```
process(digit_sel, d_units, d_tens, count_in)
    variable v_int : integer range 0 to 99;
begin
    v_int := to_integer(count_in);
    if digit_sel = '0' then
        cathode <= "0001"; -- unités
        current_bcd <= d_units;
    else
        cathode <= "0010"; -- dizaines
        if v_int < 10 then
            current_bcd <= "1111"; -- éteint
        else
            current_bcd <= d_tens;
        end if;
    end if;
end process;
```

Quand **digit_sel = 0,** le bloc active le digit des unités et envoie le chiffre des unités **(d_units)** vers le décodeur.
Quand **digit_sel = 1,** il active le digit des dizaines et envoie le chiffre des dizaines **(d_tens).**

- Si la valeur à afficher est **inférieure à 10**, il n'y a pas de dizaine à montrer.
- Dans ce cas, le bloc éteint le digit des dizaines pour éviter d'afficher un zéro à gauche.

Le chiffre choisi (unités ou dizaines) est stocké dans current_bcd

Bloc 4 – Décodage

```
decoder_inst : entity work.bcd_to_7seg
    port map (
        bcd => current_bcd,
        seg => seg
    );
```

On envoie **current_bcd** au décodeur **bcd_to_7seg.**
Le décodeur traduit le chiffre en segments allumés.
Exemple : **current_bcd = "0010" → affiche le chiffre "2"**.

Décodeur BCD vers 7 segments

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity bcd_to_7seg is
    Port (
        bcd : in  STD_LOGIC_VECTOR (3 downto 0); -- chiffre BCD (0 à 9)
        seg : out STD_LOGIC_VECTOR (6 downto 0)  -- segments a à g (actif haut)
    );
end bcd_to_7seg;

architecture rtl of bcd_to_7seg is
begin
    process(bcd)
    begin
        case bcd is
            when "0000" => seg <= "1111110"; -- 0 : a b c d e f
            when "0001" => seg <= "0110000"; -- 1 : b c
            when "0010" => seg <= "1101101"; -- 2 : a b d e g
            when "0011" => seg <= "1111001"; -- 3 : a b c d g
            when "0100" => seg <= "0110011"; -- 4 : b c f g
            when "0101" => seg <= "1011011"; -- 5 : a c d f g
            when "0110" => seg <= "1011111"; -- 6 : a c d e f g
            when "0111" => seg <= "1110000"; -- 7 : a b c
            when "1000" => seg <= "1111111"; -- 8 : tous les segments
            when "1001" => seg <= "1111011"; -- 9 : a b c d f g
            when others => seg <= "0000000"; -- éteint : aucun segment
        end case;
    end process;
end rtl;
```

- **Entrée bcd** : chiffre codé sur 4 bits (exemple : “0010” = 2).
- **Sortie seg** : vecteur de 7 bits pour les segments a à g.

-Dès que **bcd change**, le contenu du **process est recalculé**.

-le **case** agit comme une **table de correspondance** :
Si **bcd = "0000" → on allume les segments pour afficher 0...etc**

Module de Contrôle de Barrière : Automatisation d'Ouverture/Fermeture

Le module de contrôle de la barrière, l'élément physique qui gère l'accès des véhicules au parking.

- **Capteurs et actionneurs**
- **Automatisation du système**

PARTIE 4

Objectifs et Spécifications Fonctionnelles

Assurer l'ouverture et la fermeture séquencée et sécurisée de la barrière en réponse aux commandes du module principal et aux capteurs de position.

Entrées : Capteurs & Commandes

1. trigger_open :
Commande d'ouverture venant du module principal.

2. sensor_open_limit :
Capteur de fin de mouvement (barrière ouverte).

3. sensor_closed_limit :
Capteur de fin de mouvement (barrière fermée).

4. sensor_passage :
Capteur de passage du véhicule (déclenche la fermeture).

Sorties : Commandes Moteur

1. motor_open :
Active le moteur pour l'ouverture.

2. motor_close :
Active le moteur pour la fermeture.

Conception : Une Machine à États Finis (FSM)

1. **IDLE_CLOSED :** Barrière fermée, attente d'un trigger_open
2. **OPENING :** Moteur d'ouverture actif, attente de sensor_open_limit
3. **IDLE_OPEN :** Barrière ouverte, attente de sensor_passage
4. **CLOSING :** Moteur de fermeture actif, attente de sensor_closed_limit

ouvrir-attendre-fermer modélisé par une FSM, comportement déterministe et sécurisé.

Implémentation VHDL

Utilisation du modèle à trois processus : registre d'état, logique de prochain état et logique de sortie pour une conception synchrone et claire.

Détails de l'Implémentation VHDL

Entité VHDL

L'entité définit l'interface : horloge, reset, 4 entrées de capteurs/commandes et 2 sorties de moteur.

Processus 1 : Registre d'État

Ce processus synchrone gère la transition d'état sur le front montant de l'horloge. Nous avons choisi un Reset Synchrone pour garantir la stabilité de l'état initial.

Processus 3 : Logique de Sortie

Ce processus combinatoire est simple : il active motor_open uniquement dans l'état OPENING et motor_close uniquement dans l'état CLOSING. Dans les états IDLE, les moteurs sont désactivés.

Détails de l'Implémentation VHDL

Entité VHDL

L'entité définit l'interface : horloge, reset, 4 entrées de capteurs/commandes et 2 sorties de moteur.

```
ENTITY controle_barriere IS
  PORT (
    -- Signaux de contrôle
    clk   : IN  STD_LOGIC; -- Horloge (pour la logique séquentielle)
    rst   : IN  STD_LOGIC; -- Reset synchrone (pour initialiser)

    -- Entrées (capteurs et commandes)
    trigger_open        : IN  STD_LOGIC; -- Ordre d'ouverture (vient du module principal)
    sensor_passage      : IN  STD_LOGIC; -- Capteur: la voiture est passée
    sensor_open_limit   : IN  STD_LOGIC; -- Capteur: la barrière est en position haute
    sensor_closed_limit : IN  STD_LOGIC; -- Capteur: la barrière est en position basse

    -- Sorties (commandes moteur)
    motor_open  : OUT STD_LOGIC; -- Commande au moteur: Ouvrir
    motor_close : OUT STD_LOGIC  -- Commande au moteur: Fermer
  );
END ENTITY controle_barriere;
```

Détails de l'Implémentation VHDL

```vhdl
-- 1. Définition des états de notre FSM
TYPE state_type IS (
  IDLE_CLOSED, -- La barrière est fermée et attend
  OPENING,     -- La barrière est en train de s'ouvrir
  IDLE_OPEN,   -- La barrière est ouverte et attend le passage
  CLOSING      -- La barrière est en train de se fermer
);

-- 2. Signal interne pour mémoriser l'état actuel et le prochain état
SIGNAL state, next_state : state_type;
```

Processus 1 : Registre d'État

Ce processus synchrone gère la transition d'état sur le front montant de l'horloge. Nous avons choisi un Reset Synchrone pour garantir la stabilité de l'état initial.

```vhdl
-- PROCESS 1: Logique Séquentielle (Registre d'état)
-- Ce process mémorise l'état actuel.
-- Reset SYNCHRONE: le reset est évalué sur le front d'horloge
state_register_proc : PROCESS (clk)
BEGIN
  IF rising_edge(clk) THEN -- Détection du front montant
    IF (rst = '1') THEN
      state <= IDLE_CLOSED; -- État initial de reset
    ELSE
      state <= next_state; -- Mémorisation du prochain état
    END IF;
  END IF;
END PROCESS state_register_proc;
```

Détails de l'Implémentation VHDL

Processus 3 : Logique de Sortie

Ce processus combinatoire est simple : il active motor_open uniquement dans l'état OPENING et motor_close uniquement dans l'état CLOSING. Dans les états IDLE, les moteurs sont désactivés.

```
-- PROCESS 3: Logique Combinatoire (Logique de sortie)
-- Ce process détermine les sorties (commandes moteur)
-- en fonction de l'état actuel.
output_logic_proc : PROCESS (state)
BEGIN
  -- Valeurs par défaut pour éviter les "latches"
  motor_open  <= '0';
  motor_close <= '0';

  CASE state IS
    WHEN IDLE_CLOSED =>
      motor_open  <= '0';
      motor_close <= '0';

    WHEN OPENING =>
      motor_open  <= '1'; -- On active le moteur pour ouvrir
      motor_close <= '0';

    WHEN IDLE_OPEN =>
      motor_open  <= '0';
      motor_close <= '0';

    WHEN CLOSING =>
      motor_open  <= '0';
      motor_close <= '1'; -- On active le moteur pour fermer

    WHEN OTHERS =>
      motor_open  <= '0';
      motor_close <= '0';

  END CASE;
END PROCESS output_logic_proc;
```

Logique de Transition (Prochain État)

Processus 2 : Prochain État

C'est le cœur de la logique de contrôle. Il détermine le prochain état en fonction de l'état actuel et des entrées.

Transition 1 : Ouverture

De IDLE_CLOSED à OPENING : La transition se fait uniquement si trigger_open = '1'. Cela simule la réception de l'ordre d'ouverture.

Transition 2 : Attente

De OPENING à IDLE_OPEN : La transition se fait uniquement si sensor_open_limit = '1'. Le système attend la confirmation physique que la barrière est bien ouverte avant de s'arrêter.

Transition 3 : Fermeture

De IDLE_OPEN à CLOSING : La transition se fait uniquement si sensor_passage = '1'. Cela garantit que la barrière ne se ferme qu'après le passage effectif du véhicule.

Logique de Transition (Prochain État)

```
-- PROCESS 2: Logique Combinatoire (Calcul du prochain état)
-- Ce process calcule l'état suivant en fonction de l'état
-- actuel et des entrées (capteurs).
next_state_logic_proc : PROCESS (state, trigger_open, sensor_passage, sensor_open_limit, sensor_closed_limit)
BEGIN
  -- Par défaut, on reste dans le même état
  next_state <= state;

  CASE state IS
    -- État 1: Barrière fermée
    WHEN IDLE_CLOSED =>
      IF (trigger_open = '1') THEN
        next_state <= OPENING; -- On passe à l'ouverture
      END IF;

    -- État 2: Barrière en ouverture
    WHEN OPENING =>
      IF (sensor_open_limit = '1') THEN
        next_state <= IDLE_OPEN; -- On est arrivé en haut
      END IF;

    -- État 3: Barrière ouverte
    WHEN IDLE_OPEN =>
      IF (sensor_passage = '1') THEN
        next_state <= CLOSING; -- La voiture est passée, on ferme
      END IF;

    -- État 4: Barrière en fermeture
    WHEN CLOSING =>
      IF (sensor_closed_limit = '1') THEN
        next_state <= IDLE_CLOSED; -- On est arrivé en bas, retour
      END IF;

    -- Cas par défaut (sécurité)
    WHEN OTHERS =>
      next_state <= IDLE_CLOSED;

  END CASE;
END PROCESS next_state_logic_proc;
```

Vérification du Comportement

Le testbench (tb_controle_barriere) est essentiel pour valider que le module respecte les spécifications fonctionnelles avant l'intégration.

Il simule les signaux d'entrée et observe les sorties.

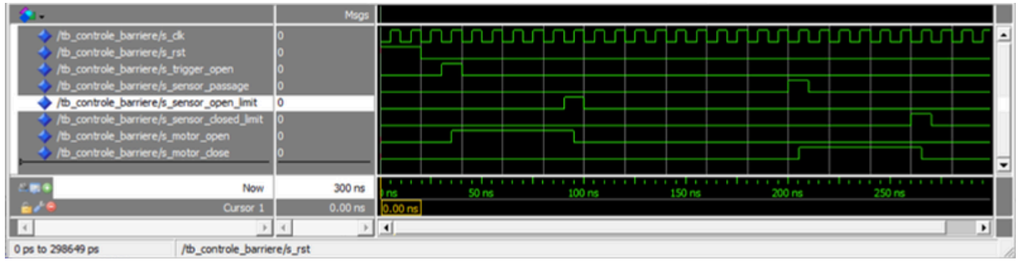
Scénario de Test

1. Reset : Vérification de l'état initial (IDLE_CLOSED).

2. Séquence d'Ouverture : Envoi de trigger_open, attente de motor_open = '1', puis envoi de sensor_open_limit = '1' pour vérifier l'arrêt du moteur.

3. Séquence de Fermeture : Envoi de sensor_passage = '1', attente de motor_close = '1', puis envoi de sensor_closed_limit = '1' pour vérifier le retour à l'état initial.

Résultats de Simulation

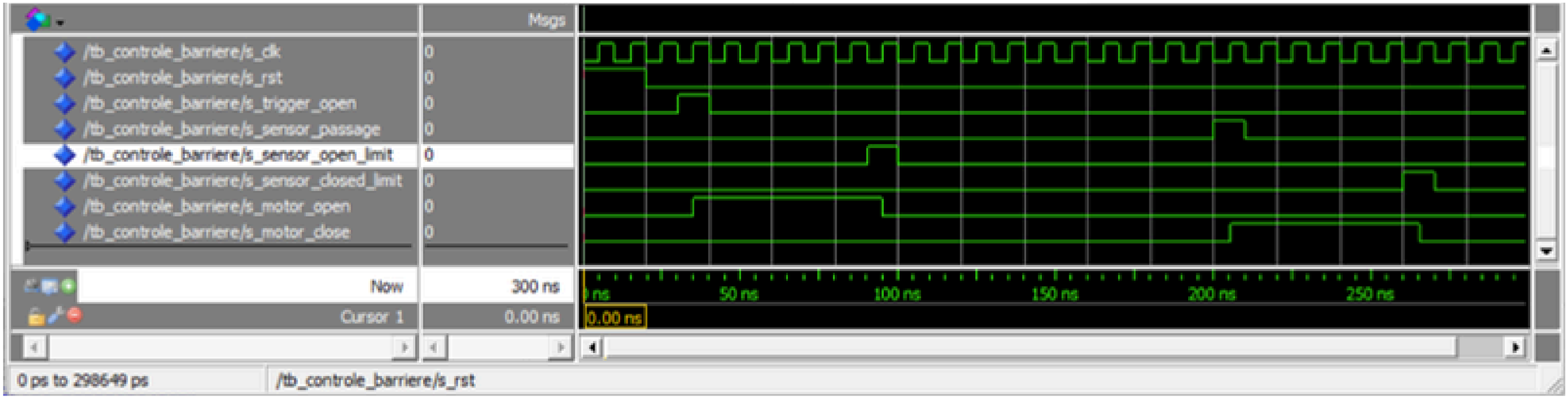


Vérification du Comportement

Résultats de Simulation

trigger_open -> motor_open -> sensor_open_limit → motor_open s'arrête ->

sensor_passage -> motor_close -> sensor_closed_limit -> motor_close s'arrête



PARTIE 5:
Module principale

- compteur_places
- sevenseg_counter
- controle_barriere

Rôle du module principal : top_parking

Ce module ne fait pas lui-même les calculs ni l'ouverture physique,
mais il organise, contrôle et connecte tous les autres modules.

Sans top_parking, les modules seraient isolés et ne pourraient pas communiquer.

Le top_parking reçoit toutes les informations venant : des capteurs (voiture entrée, voiture sortie), des capteurs de barrière, du reset
et de l'horloge. Puis il décide quel sous-module doit réagir. Il ne calcule pas lui-même, mais il dirige l'ensemble du système.

Il contient la logique de décision la plus importante
Il synchronise tous les signaux
Il convertit les données pour l'affichage
Il gère la détection "parking plein"

L’entité top_parking

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity top_parking is
    generic(
        MAX_PLACES : integer := 99
    );
    port(
        clk    : in  std_logic;
        rst    : in  std_logic;
        -- Capteurs entrÃ©e / sortie voitures
        voiture_entree : in std_logic;
        voiture_sortie : in std_logic;
        -- Capteurs barriÃ¨re
        sensor_passage      : in std_logic;
        sensor_open_limit   : in std_logic;
        sensor_closed_limit : in std_logic;
        -- Afficheur 7 segments
        cathode : out std_logic_vector(3 downto 0);
        seg     : out std_logic_vector(6 downto 0);
        -- Commandes moteur barriÃ¨re
        motor_open  : out std_logic;
        motor_close : out std_logic
    );
end top_parking;
```

Entrées capteurs / commandes

Signal	Type	Rôle
voiture_entree	in std_logic	Demande d’entrée d’une voiture
voiture_sortie	in std_logic	Demande de sortie d’une voiture
sensor_passage	in std_logic	Capteur qui détecte qu’une voiture est passée sous la barr
sensor_open_limit	in std_logic	Indique que la barrière est complètement ouverte
sensor_closed_limit	in std_logic	Indique que la barrière est complètement fermée

Sorties vers actionneurs / afficheur

Signal	Type	Rôle
cathode	out std_logic_vector(3 downto 0)	active le digit à afficher (unités/dizaines)
seg	out std_logic_vector(6 downto 0)	Segments allumés pour afficher le nombre de places
motor_open	out std_logic	Active le moteur pour ouvrir la barrière
motor_close	out std_logic	Active le moteur pour fermer la barrière

Déclaration des signaux internes

```
architecture structural of top_parking is
    ------------------------------------------------------------------------
    -- Signaux internes
    ------------------------------------------------------------------------
    signal nb_places_dispo : std_logic_vector(6 downto 0);
    signal count_unsigned  : unsigned(6 downto 0);
    signal parking_has_place  : std_logic;
    signal trigger_open : std_logic;

    -- Synchronisation des entrÃ©es (recommandÃ©)
    signal voiture_entree_sync : std_logic;
    signal voiture_sortie_sync : std_logic;
```

c'est ici que le module principal crée des signaux intermédiaires pour faire communiquer les autres modules entre eux.

1. nb_places_dispo :
C'est le nombre de places restantes, envoyé par le module compteur_places.

2. count_unsigned :
C'est nb_places_dispo, mais converti en format unsigned au lieu de std_logic_vector.

3. parking_has_place : C'est lui qui décide si on autorise l'entrée des voitures.

4. trigger_open : C'est le signal final qui dit : « Ouvre la barrière maintenant !

5. synchronisation des entrées : Les signaux venant de l'extérieur (capteurs, boutons…)
sont asynchrones donc on doit les synchronisés

```
begin
    ------------------------------------------------------------------------
    -- Synchronisation des signaux d'entrÃ©e (anti-mÃ©tastabilitÃ©)
    ------------------------------------------------------------------------
    sync_proc : process(clk)
    begin
        if rising_edge(clk) then
            if rst = '1' then
                voiture_entree_sync <= '0';
                voiture_sortie_sync <= '0';
            else
                voiture_entree_sync <= voiture_entree;
                voiture_sortie_sync <= voiture_sortie;
            end if;
        end if;
    end process;
```

Comme les signaux venant des capteurs arrivent de manière asynchrone, on utilise un petit process synchronisé avec l'horloge pour les stabiliser.

- Ce process est exécuté à chaque front d'horloge donc synchronisation avec clk
- On capture les signaux uniquement lorsqu'il y a un front montant.
- Si reset actif, on remet les signaux synchronisés à 0. C'est important pour démarrer dans un état stable.
- On transfère les signaux externes vers des signaux internes stabilisés.

Vue d’ensemble : les 5 blocs du parking intelligent

```
-------------------------------------------------------------
-- 1) Compteur places
-------------------------------------------------------------
compteur_inst : entity work.compteur_places
    generic map(MAX_PLACES => MAX_PLACES)
    port map(
        clk => clk,
        rst => rst,
        voiture_entree => voiture_entree_sync,
        voiture_sortie => voiture_sortie_sync,
        nb_places_dispo => nb_places_dispo
    );

-------------------------------------------------------------
-- 2) Conversion et dÃ©tection
-------------------------------------------------------------
count_unsigned <= unsigned(nb_places_dispo);
parking_has_place <= '1' when count_unsigned > 0 else '0';

-------------------------------------------------------------
-- 3) Logique d'ouverture CORRIGÃE
-------------------------------------------------------------
-- EntrÃ©e : seulement si places disponibles
-- Sortie : toujours autorisÃ©e (sinon deadlock si parking plei
trigger_open <= (voiture_entree_sync and parking_has_place)
                or voiture_sortie_sync;
```

1) Module compteur_places

Son rôle : gérer le nombre de places du parking

- Il compte combien de places sont encore disponibles.
- Quand une voiture entre → le compteur décrémente
- Quand une voiture sort → le compteur incrémente
- Il utilise clk pour avancer et rst pour se réinitialiser
- nb_places_dispo = résultat en binaire (0–99)

2) Conversion et détection

Son rôle : transformer la valeur binaire en un nombre utilisable pour comparer

- nb_places_dispo est en std_logic_vector (simple ensemble de bits)
- On le convertit en unsigned (nombre entier binaire)
- Ensuite, on peut faire :

count_unsigned > 0

→ Si oui → le parking a encore de la place

→ Si non → parking plein

3) Logique d’ouverture corrigée

Son rôle : décider si la barrière doit s’ouvrir ou non

La barrière doit s’ouvrir si :

✓ Une voiture veut entrer ET il reste des places

✓ Ou une voiture veut sortir (toujours autorisé)

Vue d’ensemble : les 5 blocs du parking

```
-- 4) Module barriÃ¨re
-------------------------------------------------------------
barriere_inst : entity work.controle_barriere
    port map(
        clk   => clk,
        rst   => rst,
        trigger_open        => trigger_open,
        sensor_passage      => sensor_passage,
        sensor_open_limit   => sensor_open_limit,
        sensor_closed_limit => sensor_closed_limit,
        motor_open  => motor_open,
        motor_close => motor_close
    );

-------------------------------------------------------------
-- 5) Afficheur 7 segments
-------------------------------------------------------------
sevenseg_inst : entity work.sevenseg_counter
    port map(
        clk      => clk,
        rst    => rst,
        count_in => count_unsigned,
        cathode  => cathode,
        seg      => seg
    );

end structural;
```

4) Module contrôle_barrière

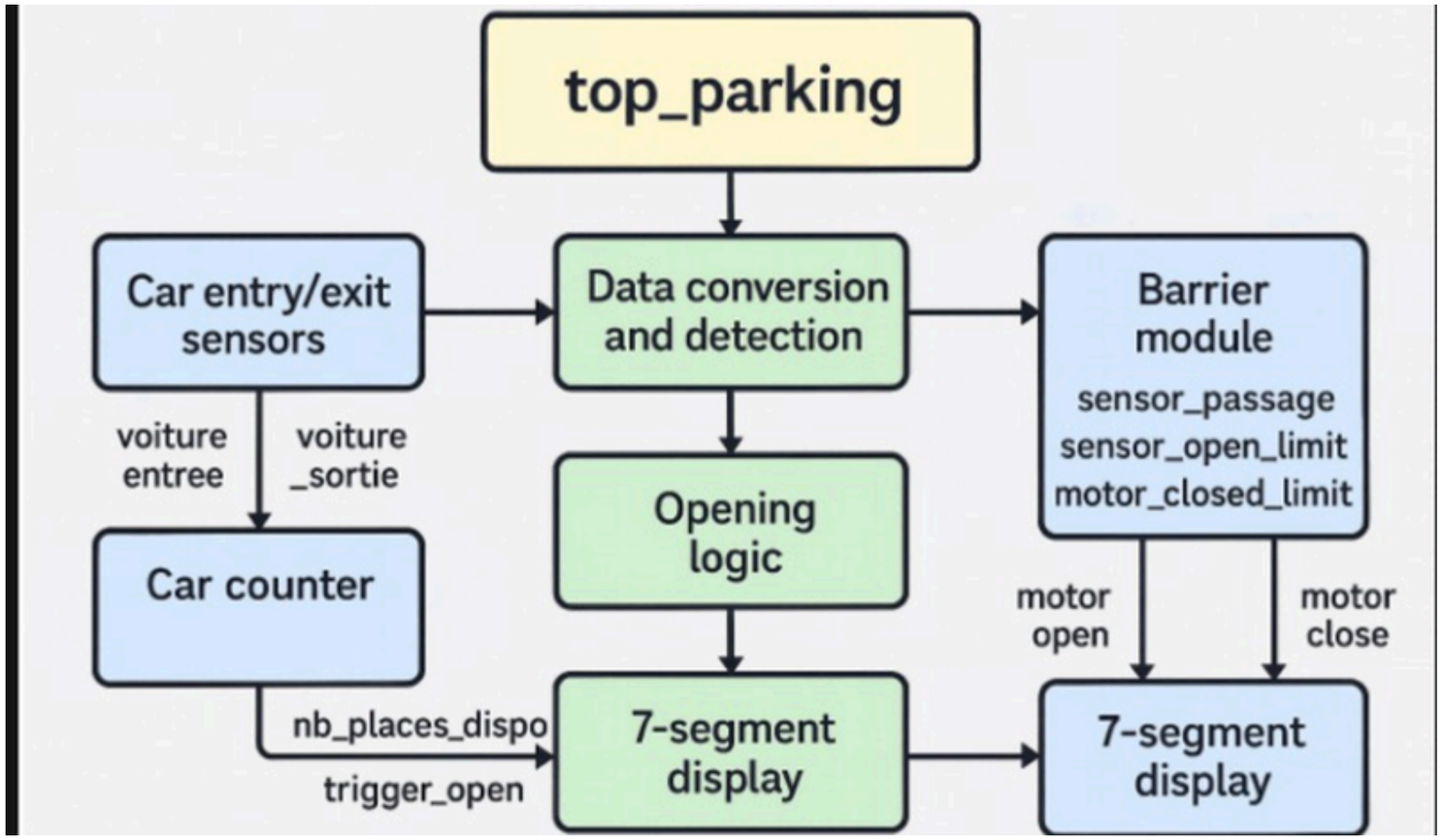
- Son rôle : faire bouger la barrière
- Ouvre la barrière si trigger_open = 1
- Attend le capteur sensor_open_limit (barrière complètement ouverte)
- Attend que la voiture passe (sensor_passage)
- Ferme la barrière
- Attend sensor_closed_limit (barrière complètement fermée)
- active le moteur dans le sens d’ouverture
- active le moteur dans le sens de fermeture

5) Module d'affichage 7 segments

Son rôle : afficher les places disponibles sur l’afficheur

- count_in = valeur des places disponibles en binaire
- Le module convertit ce nombre en chiffres (dizaines + unités)
- Il utilise :
- seg(6 downto 0) : quels segments allumer
- cathode(3 downto 0) : quel digit afficher (multiplexage)

Schéma du système top_parking



PARTIE 6:

Testbench & Simulation

Déclaration des signaux et constantes de Test Bench

```
ARCHITECTURE test OF tb_top_parking IS

  CONSTANT CLK_PERIOD    : TIME := 20 ns;
  CONSTANT MAX_PLACES    : INTEGER := 4;

  SIGNAL clk      : STD_LOGIC := '0';
  SIGNAL rst      : STD_LOGIC := '0';
  SIGNAL done     : BOOLEAN := FALSE;

  -- Capteurs (INPUT)
  SIGNAL voiture_entree          : STD_LOGIC := '0';
  SIGNAL voiture_sortie          : STD_LOGIC := '0';
  SIGNAL sensor_passage          : STD_LOGIC := '0';
  SIGNAL sensor_open_limit       : STD_LOGIC := '0';
  SIGNAL sensor_closed_limit     : STD_LOGIC := '1';

  -- Sorties (OUTPUT)
  SIGNAL motor_open              : STD_LOGIC;
  SIGNAL motor_close             : STD_LOGIC;
  SIGNAL cathode                 : STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL seg                     : STD_LOGIC_VECTOR(6 DOWNTO 0);

  -- Signaux de monitoring
  SIGNAL places_disponibles      :  INTEGER := MAX_PLACES;
  SIGNAL nb_voitures_entrees     :  INTEGER := 0;
  SIGNAL nb_voitures_sorties     :  INTEGER := 0;
  SIGNAL nb_tentatives_refusees  :  INTEGER := 0;
  SIGNAL parking_full            : STD_LOGIC := '0';
  SIGNAL parking_has_place       : STD_LOGIC := '1';
```

- Initialiser les paramètres essentiels de la simulation.
- Structurer le testbench pour une lecture claire et efficace.
- Déclarer tous les signaux nécessaires de Test Bench.

Process: Générateur d'horloge

```
clk_process : PROCESS
BEGIN
  WHILE NOT done LOOP
    clk <= '0';
    WAIT FOR CLK_PERIOD / 2;
    clk <= '1';
    WAIT FOR CLK_PERIOD / 2;
  END LOOP;
  WAIT;
END PROCESS;
```

- Génèrer le signal d’horloge utilisé dans tout le testbench.
- Continuer tant que la simulation n’est pas terminée (done = FALSE).

Instanciation du DUT

```
DUT : ENTITY work.top_parking
  GENERIC MAP (
    MAX_PLACES => MAX_PLACES
  )
  PORT MAP (
    clk                 => clk,
    rst                 => rst,
    voiture_entree      => voiture_entree,
    voiture_sortie      => voiture_sortie,
    sensor_passage      => sensor_passage,
    sensor_open_limit   => sensor_open_limit,
    sensor_closed_limit => sensor_closed_limit,
    motor_open          => motor_open,
    motor_close         => motor_close,
    cathode             => cathode,
    seg                 => seg
  );
```

- Instancier l'entité principale du parking (top_parking) dans le banc de test.
- Transmetre le paramètre générique MAX_PLACES au DUT.

Process: Monitoring

```
monitoring : PROCESS(clk)
  VARIABLE prev_parking_full : STD_LOGIC := '0';
BEGIN
  IF rising_edge(clk) THEN
    -- Calculer les places disponibles
    places_disponibles <= MAX_PLACES - nb_voitures_entrees + nb_voitures_sorties;


    -- Mise à jour des indicateurs
    IF places_disponibles = 0 THEN
      parking_full <= '1';
      parking_has_place <= '0';
    ELSE
      parking_full <= '0';
      parking_has_place <= '1';
    END IF;

    -- Afficher quand le parking devient plein
    IF parking_full = '1' AND prev_parking_full = '0' THEN
      REPORT "*** PARKING FULL ***" SEVERITY NOTE;
    END IF;

    prev_parking_full := parking_full;
  END IF;
END PROCESS;
```

- Ce processus surveille le parking pendant la simulation et met à jour les indicateurs (parking_full, parking_has_place).
- Il signale quand le parking est plein et assure un suivi précis de l'état du système.

Processus principal de test

OBJECTIVE: Génération de pulses d'un cycle et synchronisation avec les capteurs/moteurs.

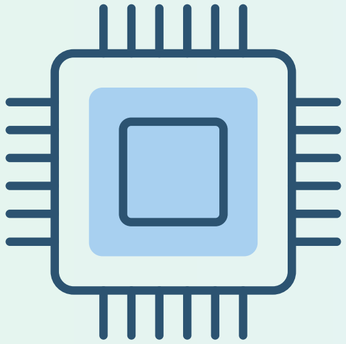
Procédure: entree_voiture

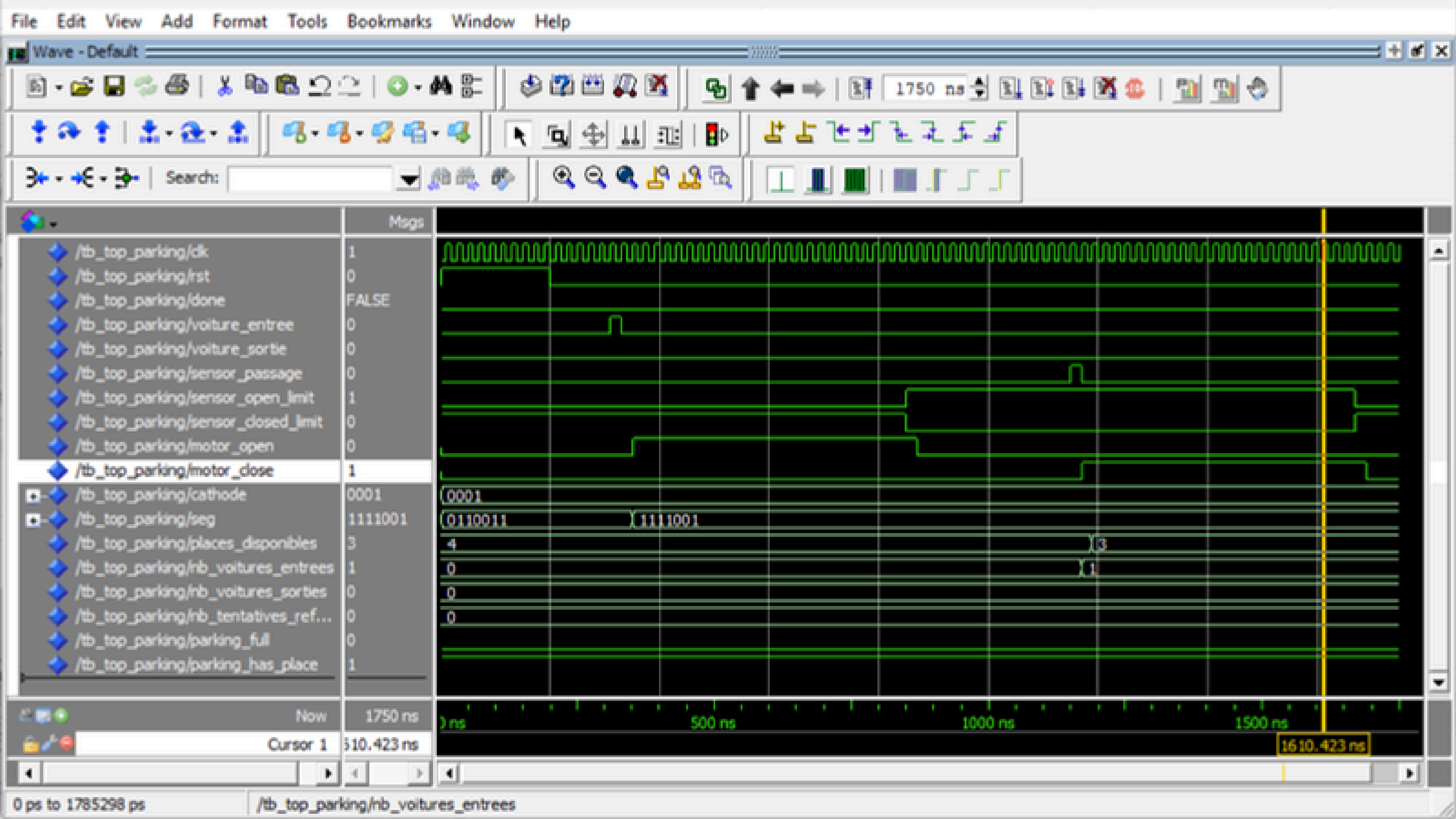
- Simule la présence (Capteur)
- Attend l'action du moteur (Barrière)
- Valide le franchissement

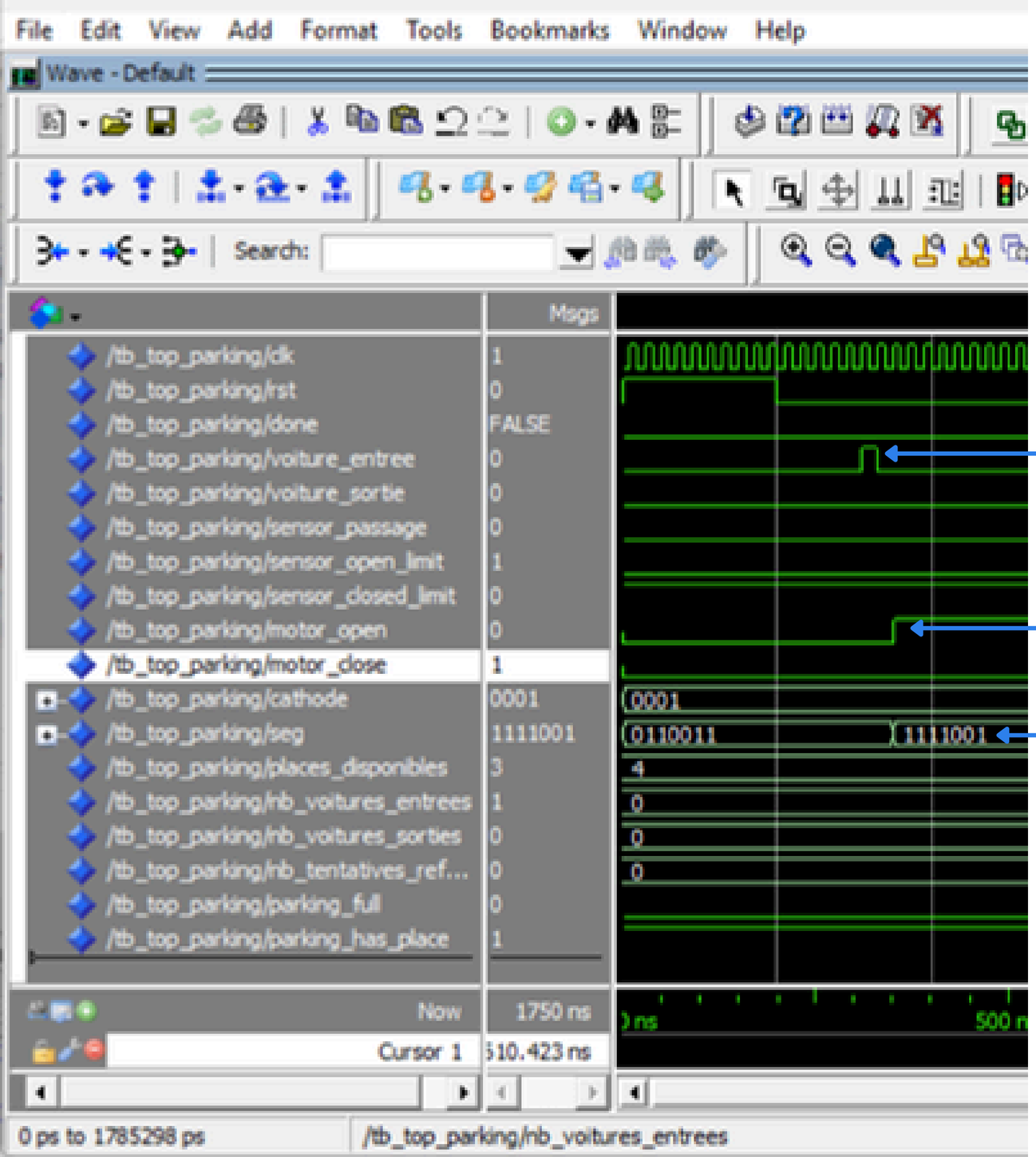
Procédure: sortie_voiture

- Simule la requête de paiement/ticket
- Attend l'ouverture
- Valide la sortie physique

SIMULATION



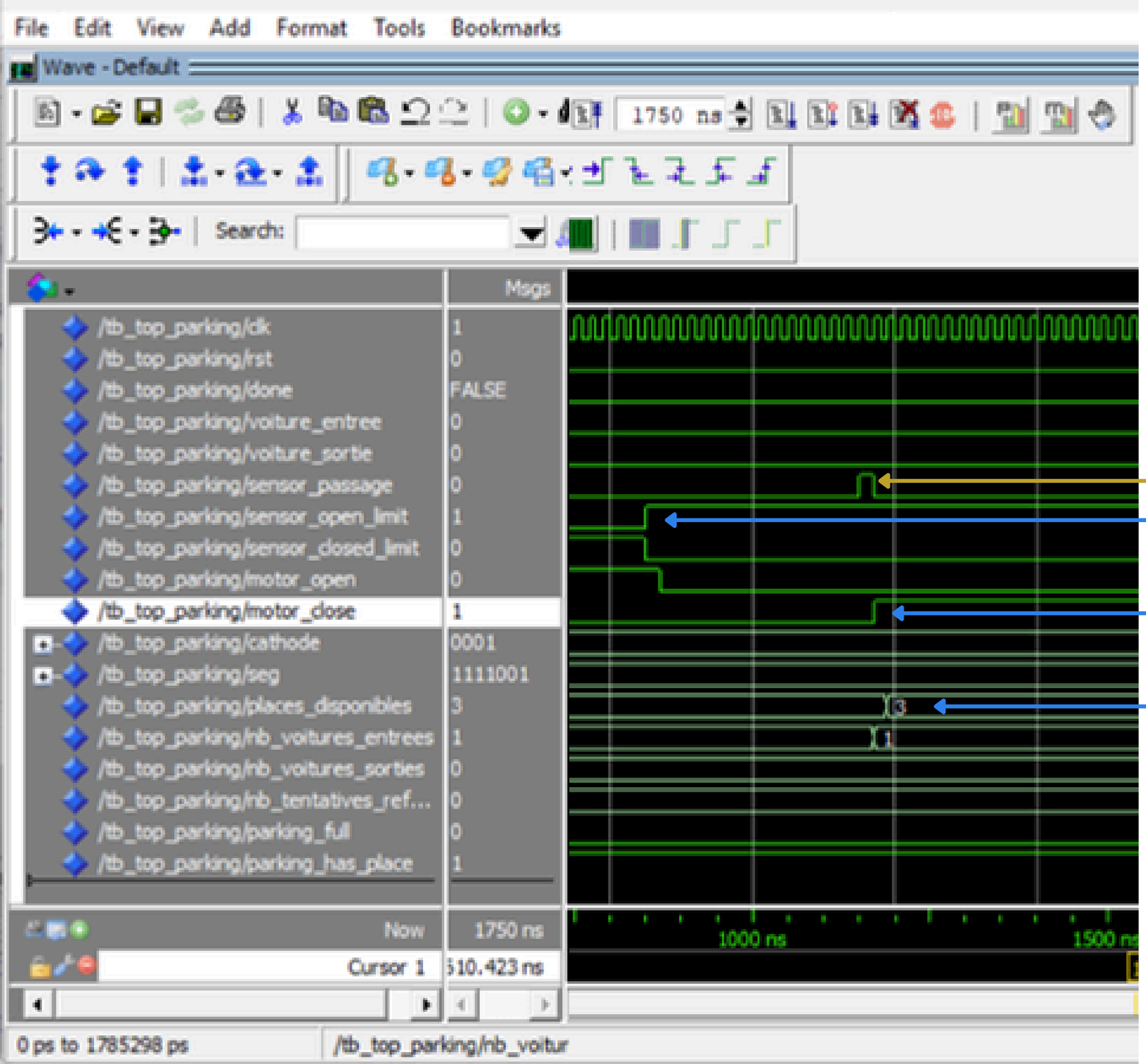

File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
1750 ns
Search:
Msgs
/tb_top_parking/clk 1
/tb_top_parking/rst 0
/tb_top_parking/done FALSE
/tb_top_parking/voiture_entree 0
/tb_top_parking/voiture_sortie 0
/tb_top_parking/sensor_passage 0
/tb_top_parking/sensor_open_limit 1
/tb_top_parking/sensor_closed_limit 0
/tb_top_parking/motor_open 0
/tb_top_parking/motor_close 1
/tb_top_parking/cathode 0001
/tb_top_parking/seg 1111001
/tb_top_parking/places_disponibles 3
/tb_top_parking/nb_voitures_entrees 1
/tb_top_parking/nb_voitures_sorties 0
/tb_top_parking/nb_tentatives_ref... 0
/tb_top_parking/parking_full 0
/tb_top_parking/parking_has_place 1
0001
0110011
1111001
4
3
0
1
0
0
Now
1750 ns
Cursor 1
1610.423 ns
500 ns
1000 ns
1500 ns
0 ps to 1785298 ps
/tb_top_parking/nb_voitures_entrees



Une voiture se présente pour entrer.

Le moteur se déclenche pour ouvrir la barrière.

Décrémentation du nombre de places libres sur l'afficheur.



Barrière ouverte totalement.

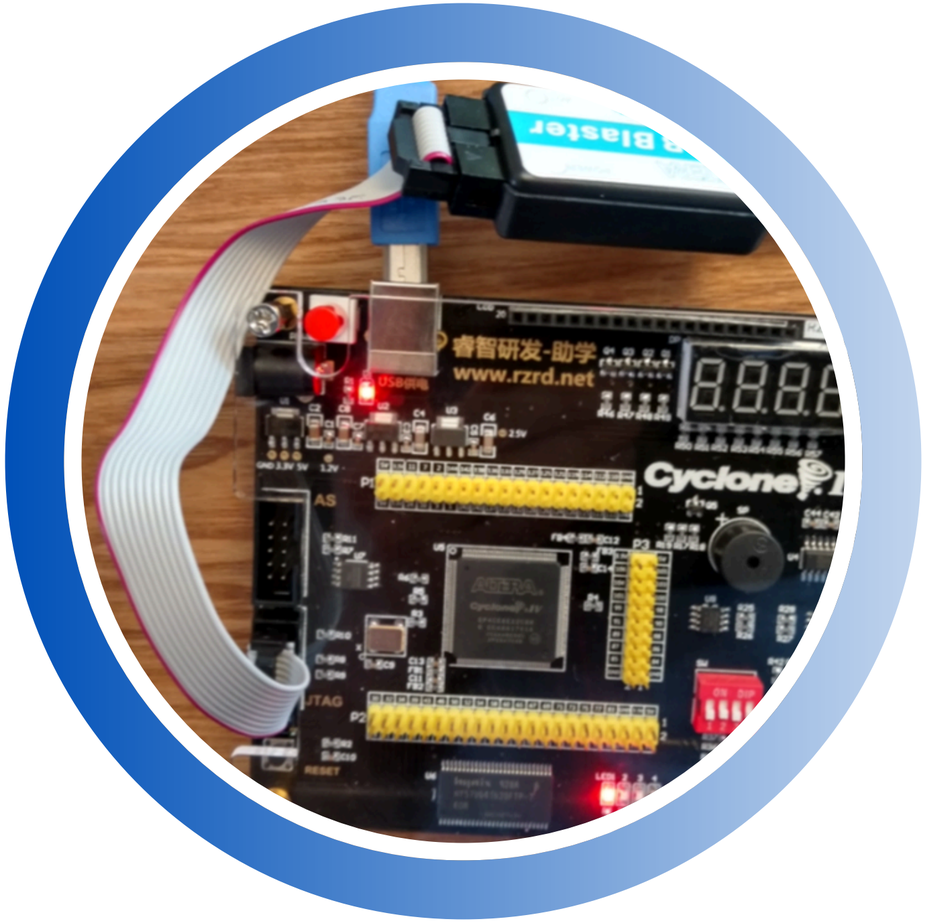
Voiture entrée complètement.

Fermeture de la barrière.

Décrémentation interne (réelle) du nombre de places libres.

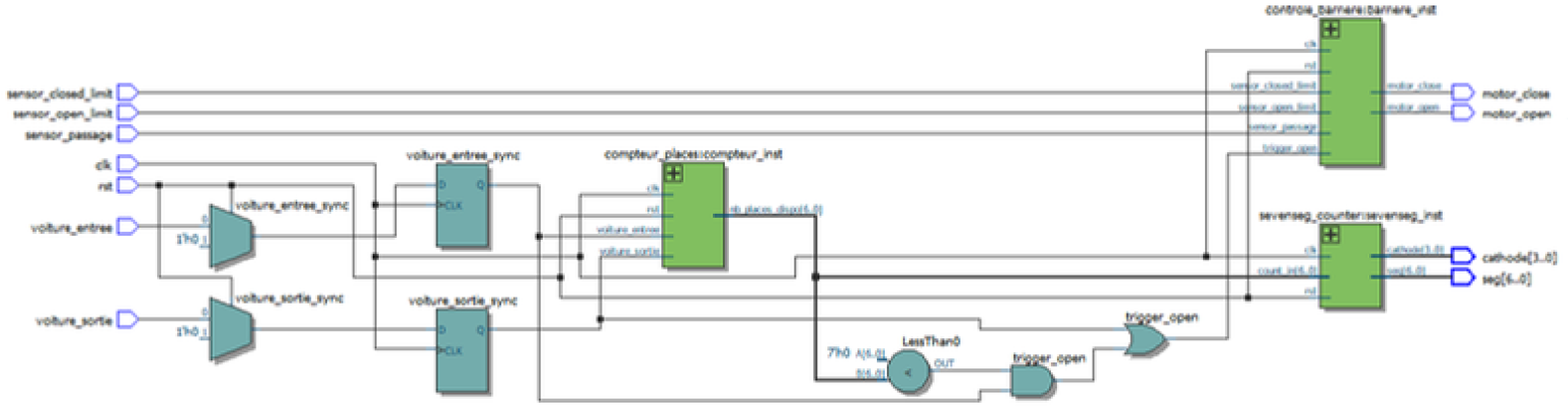
PARTIE 7:

Implémentation matérielle : Du code VHDL au système matériel fonctionnel



Étape 1 : Analyse & Synthèse

Après la synthèse par Quartus, dans le but de valider et optimiser notre code, nous analysons le circuit à deux niveaux
Ces deux vues permettent d'évaluer la cohérence entre l'intention de codage et l'implantation matérielle finale.

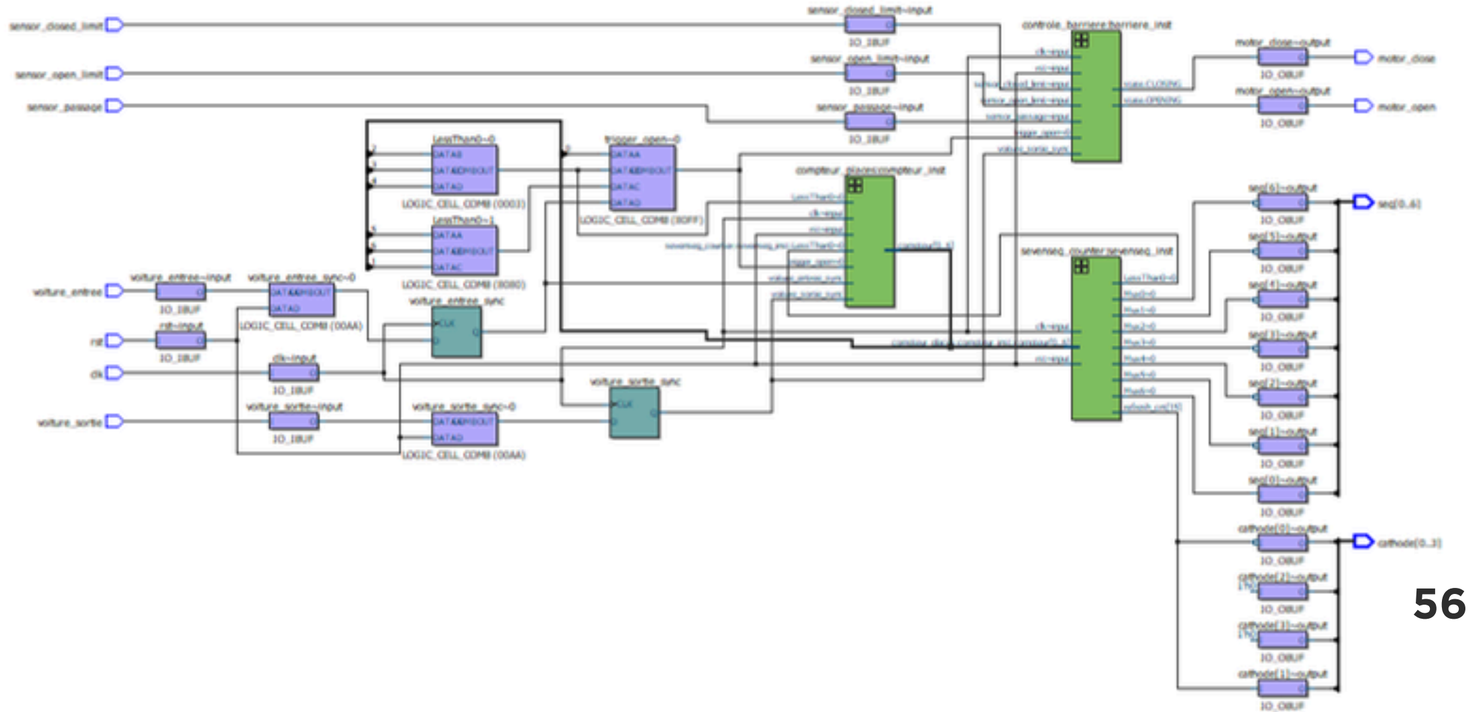


RTL Viewer

Fournit une vue logique abstraite issue de l'analyse RTL

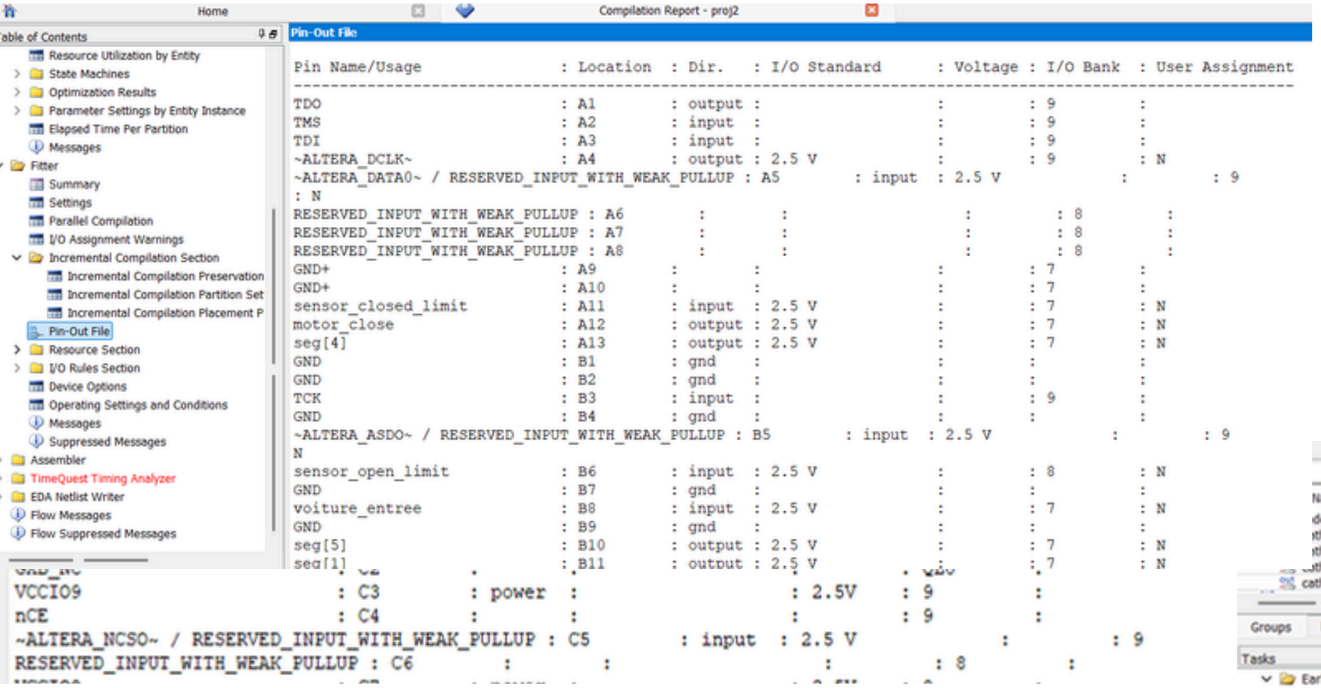
Technology Map Viewer

Représente la netlist synthétisée et mappée sur les ressources réelles du FPGA (LUT, registres, RAM).



Étape 2 : Assignation des pins

L’assignation des pins permet de relier les signaux logiques du design aux broches physiques du FPGA via le Pin Planner
Les principales sorties de cette étape sont le fichier .qsf, qui enregistre toutes les contraintes de broches, le fichier .pin, qui liste le pin-out final après compilation, ainsi que les rapports .rpt d’analyse des I/O.

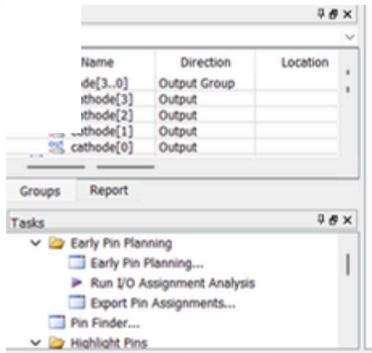


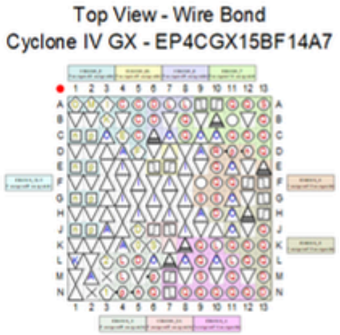
Fichier .pin

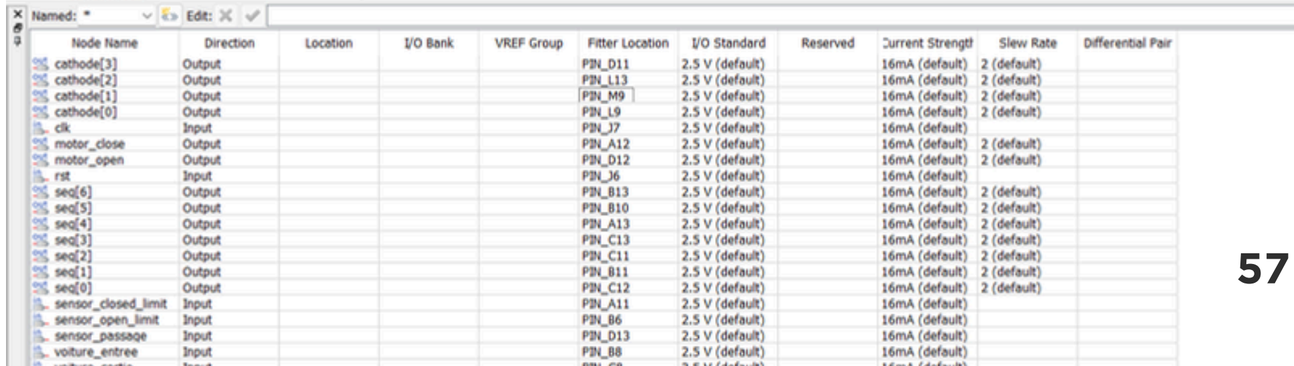
Récapitule l’ensemble des broches réellement utilisées par le design, avec leur nom logique, leur position physique et leur direction (input/output).

Pin Planner

Fournit une vue globale, graphique et interactive de l’assignation des pins. Il permet de visualiser la disposition physique du FPGA, et de modifier facilement les affectations





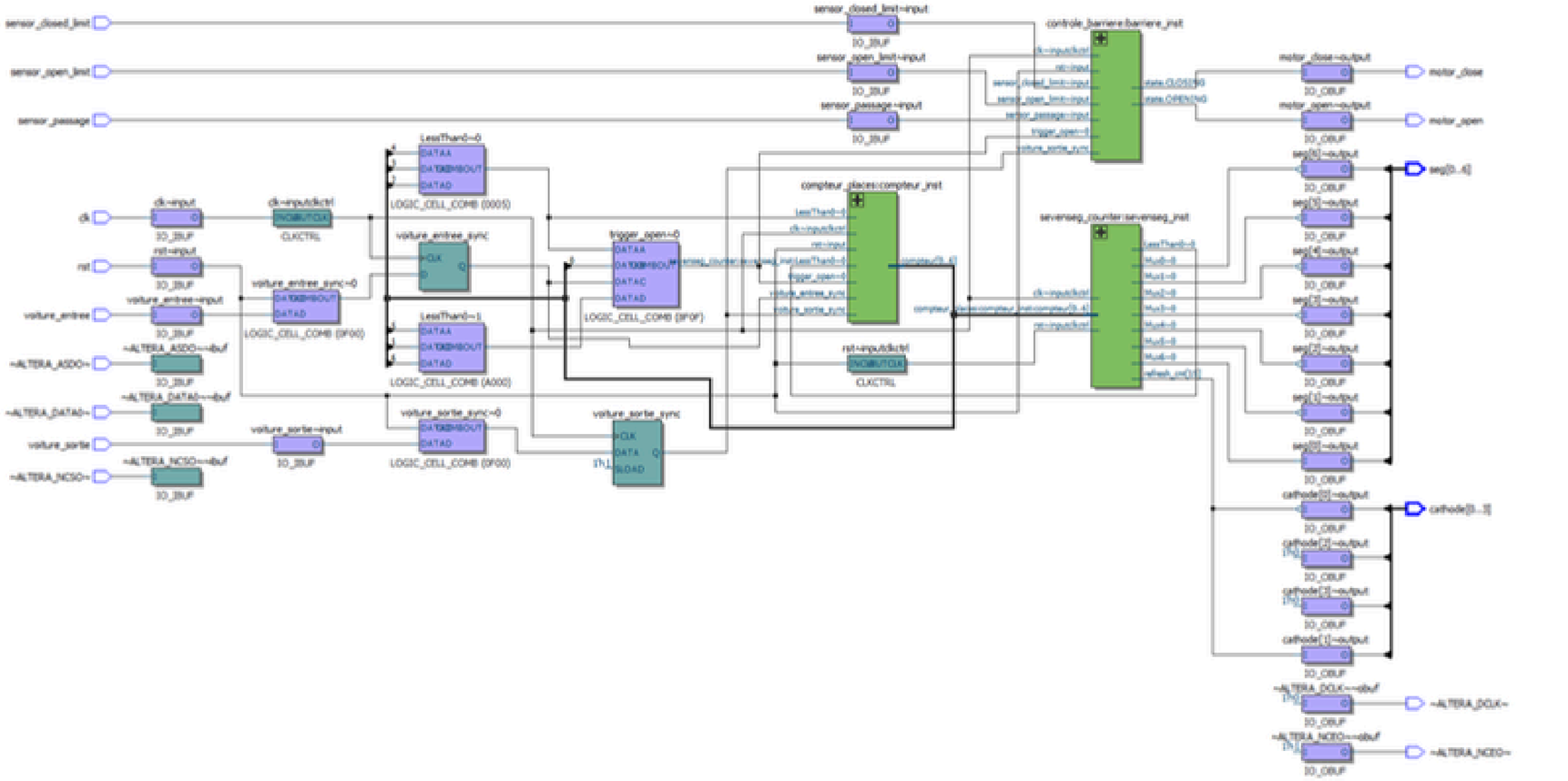


Étape 3 : Placement et Routage (Fitter)

L’étape de Placement et Routage, assurée par le module Fitter de Quartus, consiste à implanter physiquement le design synthétisé dans l’architecture réelle du FPGA. Durant cette phase, le Fitter détermine l’emplacement précis de chaque ressource logique (LUT, registres, blocs mémoire…) puis établit les connexions internes nécessaires en utilisant le réseau d’interconnexion du FPGA.

Technology Map Viewer after Fitting

NetList final avec placement et routage réèl dans le FPGA.

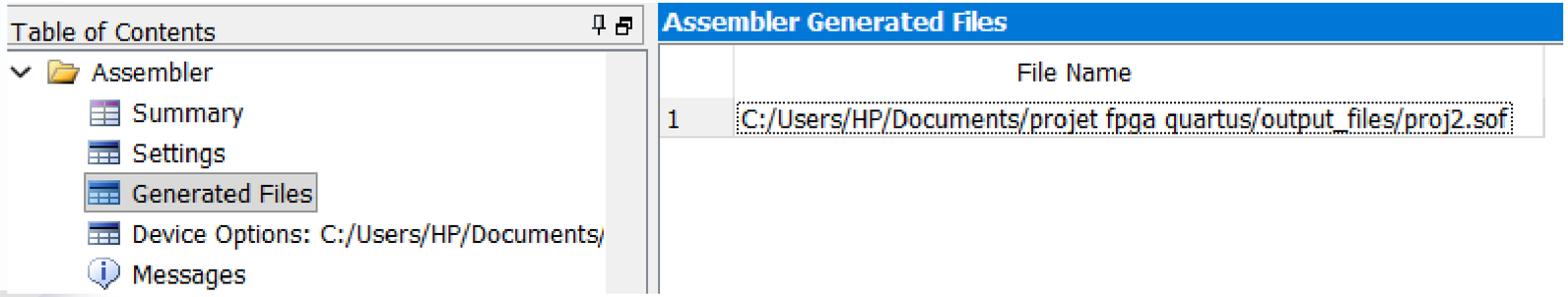


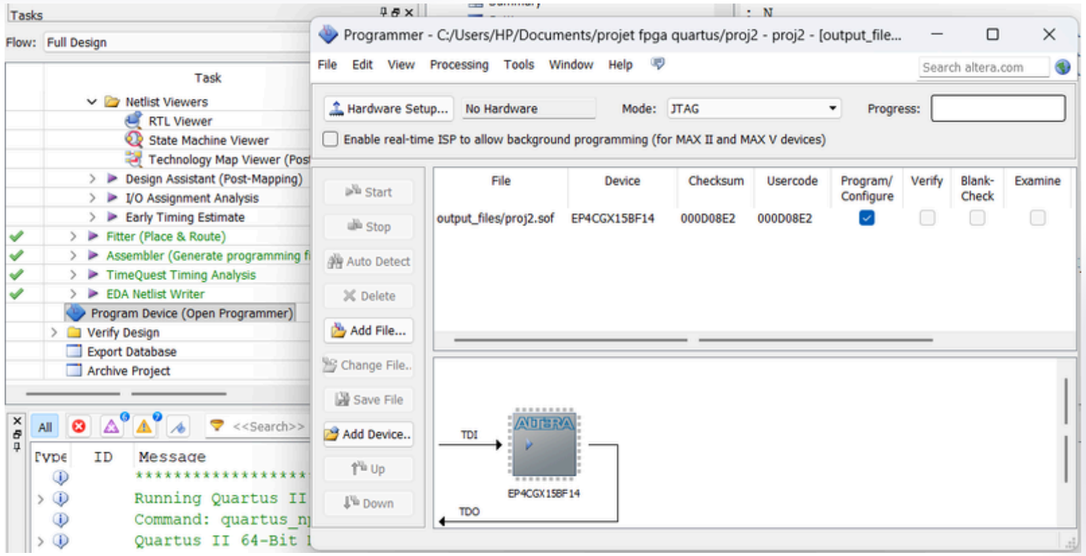
Étape 4 : Génération du bitstream et programmation du FPGA

Une fois le placement et le routage terminés, Quartus génère le fichier bitstream .sof, qui contient la configuration finale du FPGA. Ce fichier est ensuite chargé dans le FPGA via l'outil Programmer et l'interface USB-Blaster. Dès la programmation terminée, le FPGA adopte instantanément la logique décrite dans le bitstream et le système est prêt à fonctionner.

Fichier bitstream généré

C'est le fichier qui sera chargé dans le fPGA lors de la programmation

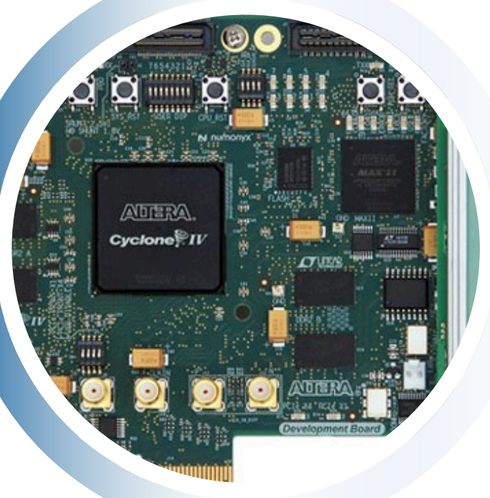




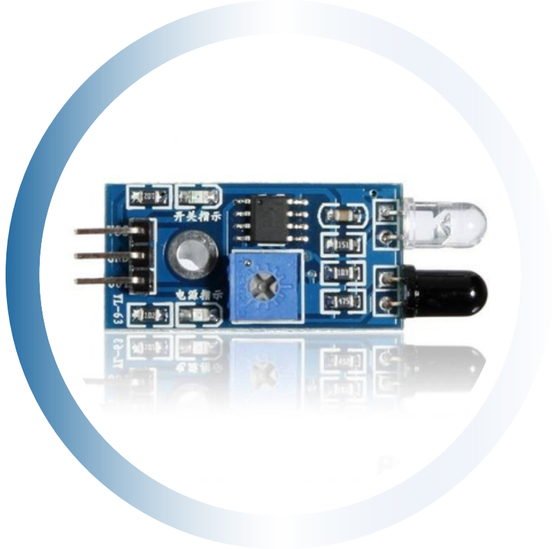
Programmation

Enfin, nous connectons la carte au PC via un câble JTAG (USB-Blaster), puis nous ouvrons l'outil Quartus Programmer afin de lancer la programmation du FPGA.

Matériel



FPGA basse consommation de la famille Cyclone IV GX

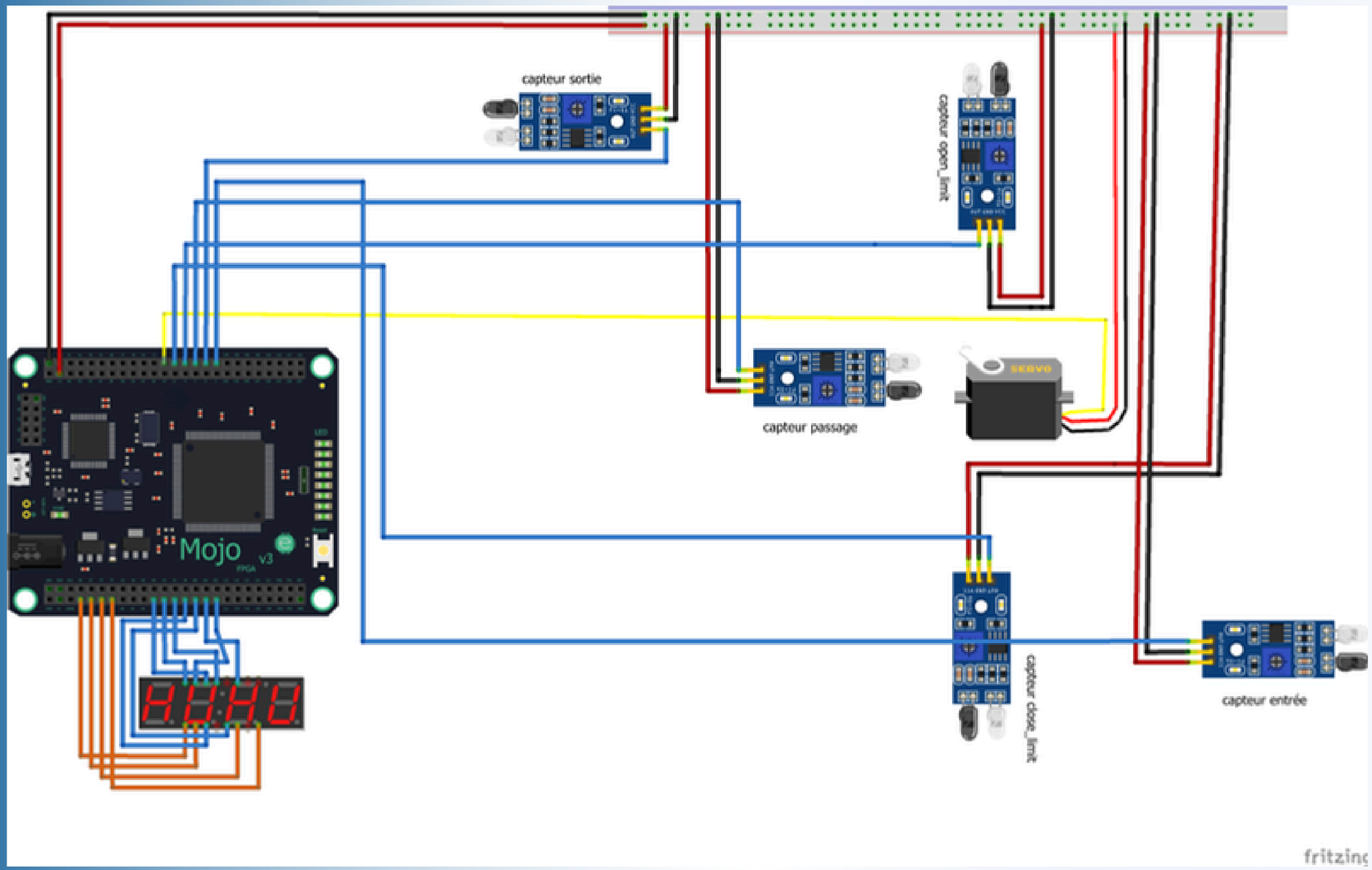


Module capteur infrarouge comprenant un émetteur et un récepteur IR

Micro-servo moteur offrant une rotation d'environ 0–180° contrôlée par PWM

Module d'affichage composé de quatre chiffres 7-segments multiplexés.

Montage final



Note : Fritzing ne disposant pas de cartes Altera/Intel, nous utilisons une Mojo v3 (FPGA Spartan) comme équivalent visuel pour le schéma.

Conclusion

Au terme de ce projet, nous avons réussi à concevoir et mettre en place un système intelligent de gestion de parking . L'ensemble des objectifs techniques fixés ont été atteints : le système assure correctement la détection des entrées et sorties, le comptage en temps réel du nombre de places disponibles, l'affichage dynamique sur 7-segments, ainsi que le contrôle automatique de la barrière selon l'état du parking. Les simulations réalisées via le testbench ont permis de valider le comportement global du système avant son implémentation matérielle.

Ce travail nous a offert une réelle mise en pratique des concepts essentiels liés à la conception numérique, la programmation VHDL et l'organisation hiérarchique d'un système embarqué. Il a également permis de renforcer notre compréhension du fonctionnement des FPGA et de la modularité dans les architectures matérielles.

ÉCOLE NATIONALE DES SCIENCES APPLIQUÉES DE TETOUAN

GCSE2
2025-2026

MERCI

POUR VOTRE ATTENTION

Présenté par :

KUNAKA DANIEL
MAKRI YOUSSRA
SIMPORE TAOBATA
SABOR LAILA
ABOUL-MOUMOUNI DIALLO
BOUARRAF DOHA
EL BAROUDI MALAK
MOUHAFID HAFSA

Professeur : JAMAL ZBITOU