Car parking system

Lab 2

Matteo Golinelli, Nicolò Vantini, Marcus Gregory

19 November 2025

1 Introduction

In this project an advanced car parking system is designed from finite state machines to a VHDL implementation. The system is designed to control the traffic entering a car park composed of 7 available spots with gates (2 entering gates and 2 exiting), a 7-segment display showing available spots and green/red lights for availability status.

2 Assumptions

We decided to divide the system into three different entities, in order to better separate the different demands of the system specifications. Also, we used the required input and output signal, but we changed the internal signals used to communicate and manage the different entities.

2.1 Entering gate

During the development of the entering-gate finite state machine we decided to make some assumptions in order optimize it and make it simpler to code. In particular, we added:

- an INPUT signal *enable*: this signal comes from the architecture Control (see later for further explanation) and is set to 1 only when a car can be accepted in the parking lot (can be 0, i.e the parking lot is full)
- an OUTPUT signal *car_detected* that is set to 1 when the front sensor detects a new car, it is later used in the controller to compute the value of the signal *enable*
- an OUTPUT signal *car_entered* to update the car counter in the controller architecture

2.2 Exiting gate

For the exiting gate an extra output state was added to communicate the leaving of a car. The signal is sent on the transition from `CAR_EXITED` to `IDLE`. This is due to the `CAR_EXITING` and `CAR_EXITED` states, which exist in case a car does not completely leave or a car passes the sensor while another is exiting. Only sending the signal on the transition to `IDLE` ensures that the signal cannot be sent without going through the whole process again.

2.3 Control unit

The system is managed by a control unit that is responsible for tracking the free spots, sending and receiving payment requests, showing the free spots on the display and turning on and off the appropriate LED. The unit consists of a finite state machine which drives the entering gates, and multiple processes running in parallel which manage the other functionalities. This architecture was chosen to avoid input race conditions.

2.3.1 Gate control

While designing the system, we approached the problem of two cars entering the two gates simultaneously. By receiving the *car_detected* signal, the system verifies the amount of spots left free, and drives the *enable* signals appropriately. In the case of two cars approaching simultaneously, whilst only one parking spot is left, gate 1 is prioritized.

2.3.2 Payment control

Another interesting topic is how to manage the exiting cars. In fact, the system only has one *payment_request* output and one *payment_done* input. We decided to not face the case in which two cars exit at the same time, because it was not explicitly reported in the system specifications. In our design, no distinction is made between which gate sends the *payment_request*, and both are fed with the same *payment_done* signal. This creates a non-ideal situation in which, if two cars approach the exits, the payment of one would trigger both gates to open. Future improvements of the system addressing this issue would be to have a payment request input and output for each exit, or to process multiple requests inside the *Parking_Controller* entity and drive the output gates accordingly.

2.3.3 Other processes

The control unit features separate processes for different tasks.

- A process updates the counter that keeps track of the free parking spots.
- A process drives the green and red LEDs.
- A process sets the correct output for the 7-segment display.
- A process controls the payment options.

3 FSM diagrams

3.1 Entering gate

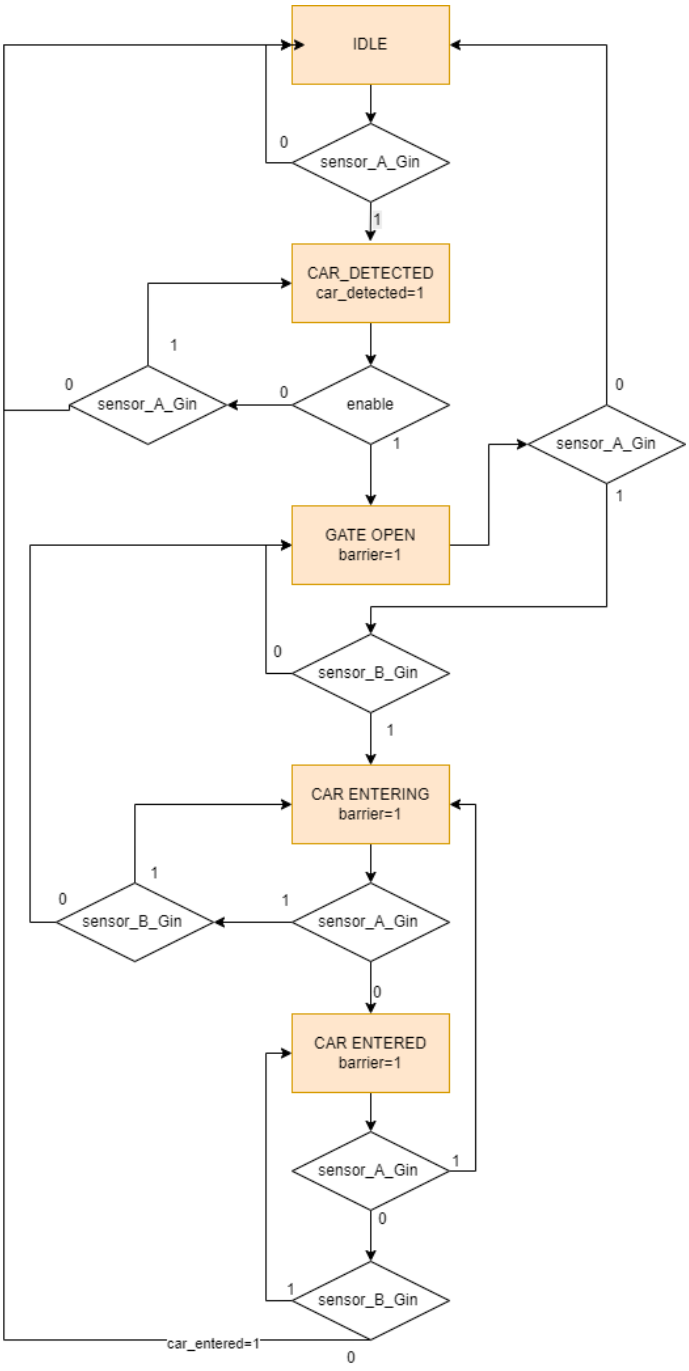

Figure 1: Diagram of the entering gate FSM

3.2 Exiting gate

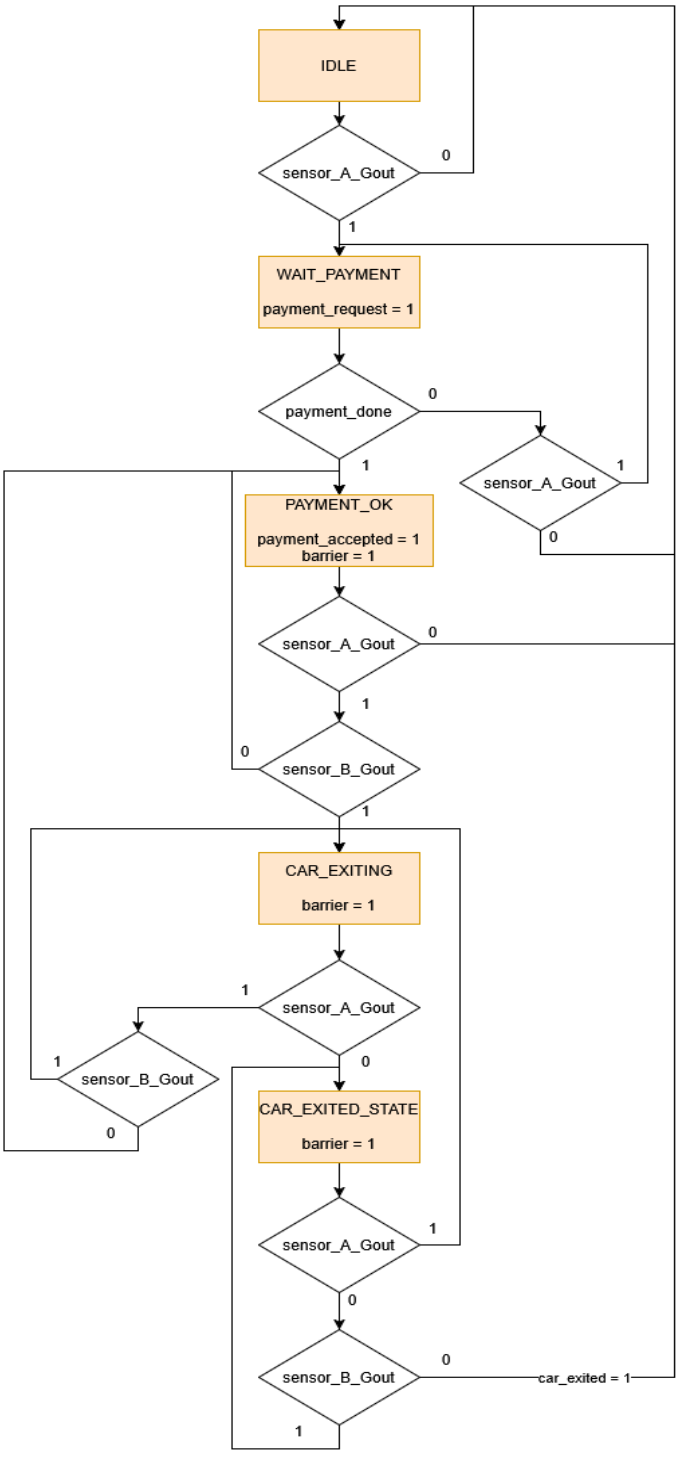

Figure 2: Diagram of the exiting gate FSM

3.3 Control unit

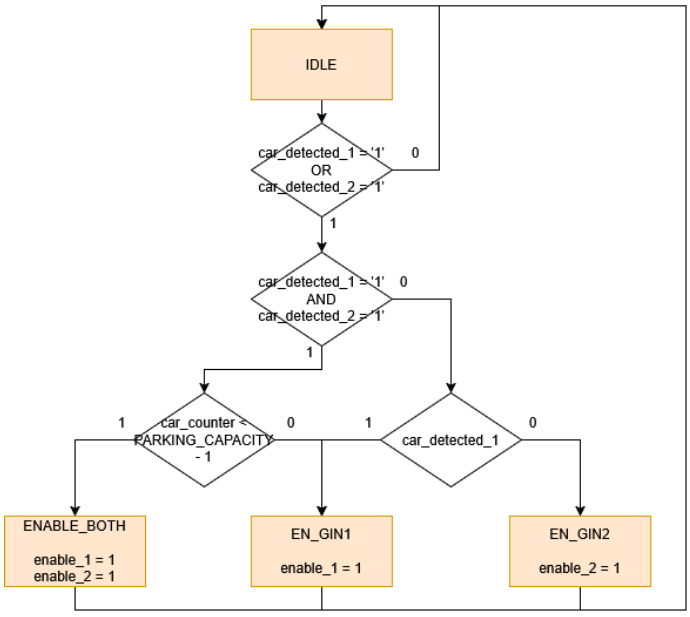

Figure 3: Diagram of the control unit FSM

4 Results

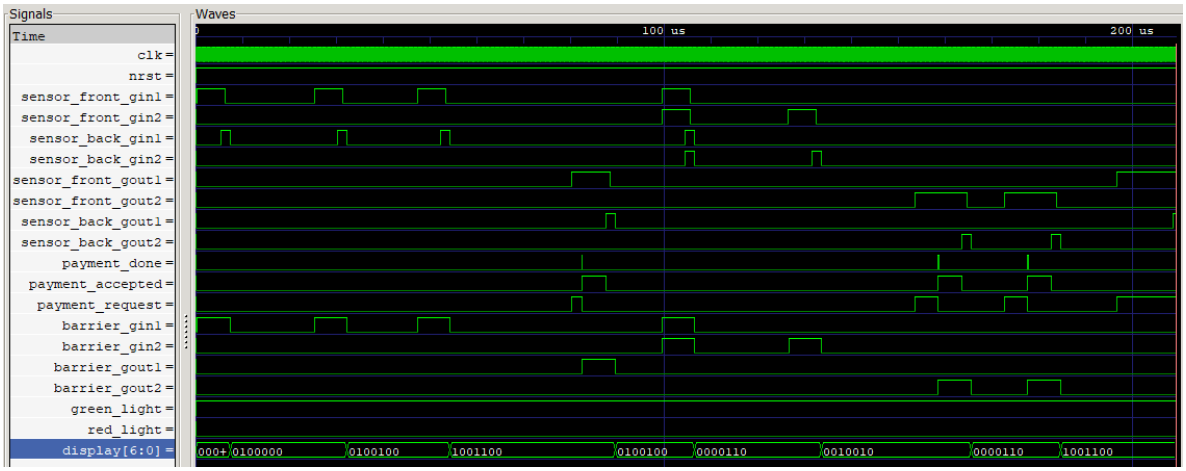

Figure 4: Case 1

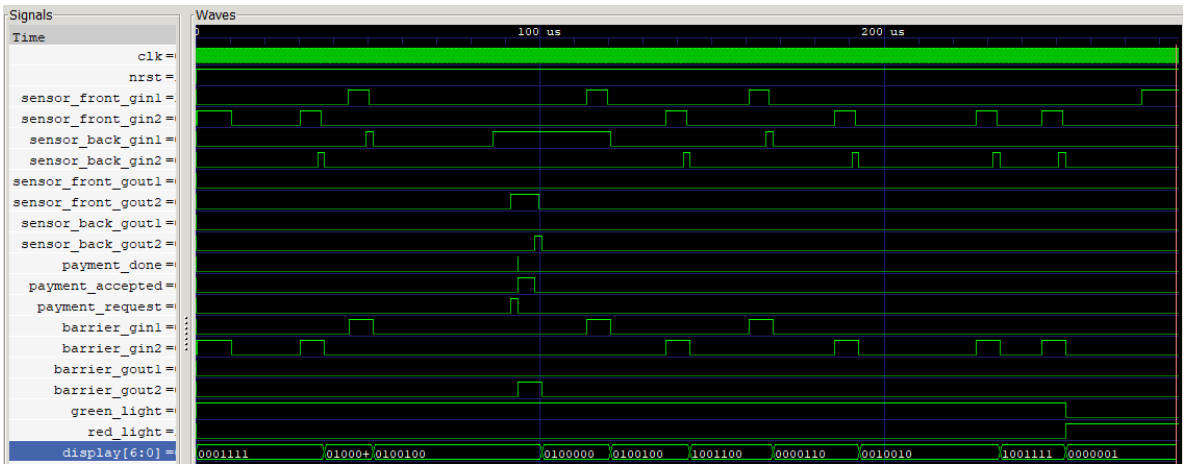

Figure 5: Case 2

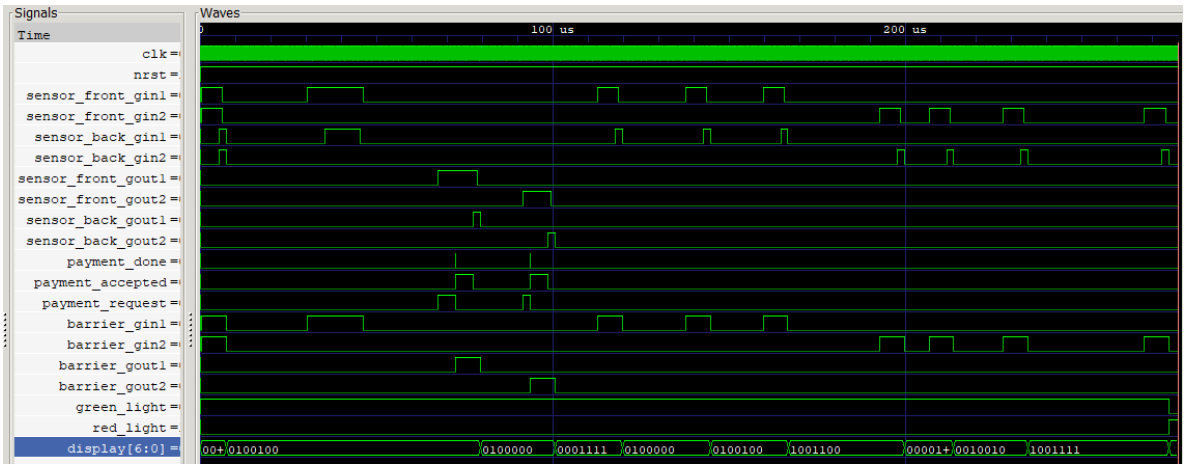

Figure 6: Case 3

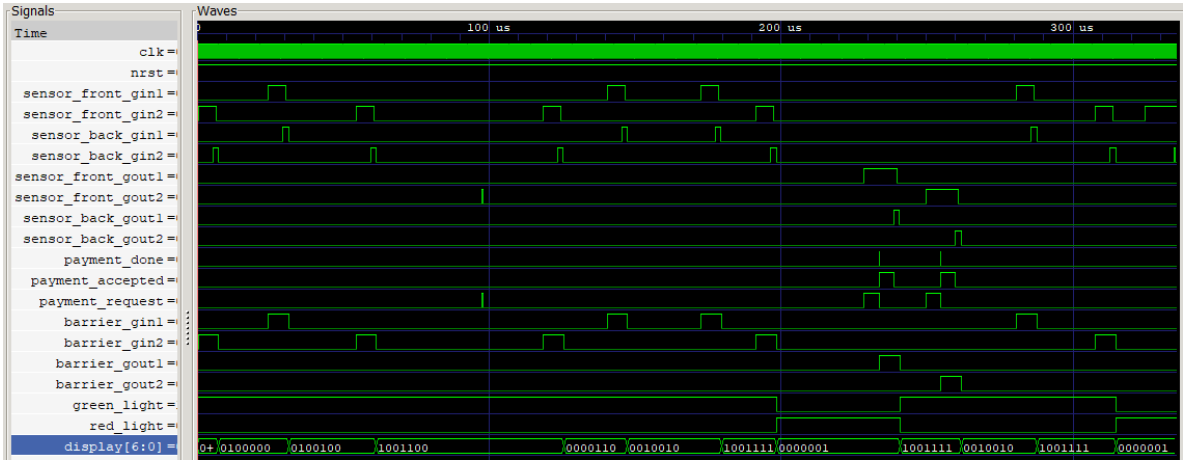

Figure 7: Case 4

5 VHDL code

5.1 Gate Entrance

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

entity Gin_fsm is
    port(
    clk          : in  std_logic;
    nrst         : in  std_logic;
    sensor_A_Gin   : in std_logic;
    sensor_B_Gin   : in std_logic;
    enable       : in std_logic;
    car_detected: out std_logic;
    car_entered : out std_logic;
    barrier      : out std_logic
    );
end entity Gin_fsm;

architecture behavioural of Gin_fsm is

    type State is (IDLE,CAR_DETECTED_STATE,GATE_OPEN,CAR_ENTERING,
    ↪  CAR_ENTERED_STATE);
    signal current_state : State;

begin
    compute_current_state : process(clk, nrst) is
    begin
        if nrst = '0' then
            current_state <= IDLE;
        elsif rising_edge(clk) then
            car_entered <= '0';
            case current_state is
                when IDLE =>
                    if sensor_A_Gin = '1' then
                        current_state <= CAR_DETECTED_STATE;
                    end if;

                when CAR_DETECTED_STATE =>
                    if enable = '1' then
                        current_state <= GATE_OPEN;
                    elsif sensor_A_Gin = '0' then
                        current_state <= IDLE;
                    end if;

                when GATE_OPEN =>
                    if sensor_A_Gin = '0' then
                        current_state <= IDLE;
                    elsif sensor_B_Gin = '1' then
                        current_state <= CAR_ENTERING;
                    end if;
```

```
                when CAR_ENTERING =>
                    if sensor_A_Gin = '0' then
                        current_state <= CAR_ENTERED_STATE;
                    elsif sensor_B_Gin = '0' then
                        current_state <= GATE_OPEN;
                    end if;

                when CAR_ENTERED_STATE =>
                    if sensor_A_Gin = '1' then
                        current_state <= CAR_ENTERING;
                    end if ;
                    if sensor_B_Gin = '0' then
                        car_entered <= '1';
                        current_state <= IDLE;
                    end if;

                when others =>
                    current_state <= IDLE; -- default IDLE, AND car_entered '0'
                    ↪
            end case;
        end if;
    end process compute_current_state;

    compute_output_logic : process(current_state) is
    begin
        -- defaulte : IDLE state
        car_detected <= '0';
        barrier      <= '0';
        case current_state is
            when CAR_DETECTED_STATE =>
                car_detected <= '1';
            when GATE_OPEN =>
                barrier <= '1';
            when CAR_ENTERING =>
                barrier <= '1';
            when CAR_ENTERED_STATE =>
                barrier <= '1';
            when others =>
                null; -- default IDLE state
        end case;
    end process compute_output_logic;
end architecture behavioural;
```

5.2 Gate Exit

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

entity Gout_fsm is
    port(
    clk          : in  std_logic;
    nrst         : in  std_logic;
    sensor_A_Gout   : in std_logic;
    sensor_B_Gout   : in std_logic;
    payment_done : in std_logic;
    payment_request : out std_logic;
    payment_accepted : out std_logic;
    barrier      : out std_logic;
    car_exited : out std_logic
    );
end entity Gout_fsm;

architecture behavioural of Gout_fsm is

    type State is (IDLE,WAIT_PAYMENT,PAYMENT_OK,CAR_EXITING, CAR_EXITED_STATE);
    signal current_state : State;

begin
    compute_current_state : process(clk, nrst) is
    begin
        if nrst = '0' then
            current_state <= IDLE;
        elsif rising_edge(clk) then
            car_exited <= '0';
            case current_state is
                when IDLE =>
                    if sensor_A_Gout = '1' then
                        current_state <= WAIT_PAYMENT;
                    end if;

                when WAIT_PAYMENT =>
                    if payment_done = '1' then
                        current_state <= PAYMENT_OK;
                    elsif sensor_A_Gout = '0' then
                        current_state <= IDLE;
                    end if;

                when PAYMENT_OK =>
                    if sensor_A_Gout = '0' then
                        current_state <= IDLE;
                    elsif sensor_B_Gout = '1' then
                        current_state <= CAR_EXITING;
                    end if;

                when CAR_EXITING =>
```

```
                    if sensor_A_Gout = '0' then
                        current_state <= CAR_EXITED_STATE;
                    elsif sensor_B_Gout = '0' then
                        current_state <= PAYMENT_OK;
                    end if;

                when CAR_EXITED_STATE =>
                    if sensor_A_Gout = '1' then
                        current_state <= CAR_EXITING;
                    end if ;
                    if sensor_B_Gout = '0' then
                        car_exited <= '1';
                        current_state <= IDLE;
                    end if;

                when others =>
                    current_state <= IDLE; -- default IDLE, AND car_exited '0'
                    ↪
            end case;
        end if;
    end process compute_current_state;

    compute_output_logic : process(current_state) is
    begin
        -- defaulte : IDLE state
        payment_request <= '0';
        payment_accepted <= '0';
        barrier      <= '0';
        case current_state is
            when WAIT_PAYMENT =>
                payment_request <= '1';
            when PAYMENT_OK =>
                payment_accepted <= '1';
                barrier <= '1';
            when CAR_EXITING =>
                barrier <= '1';
            when CAR_EXITED_STATE =>
                barrier <= '1';
            when others =>
                null; -- default IDLE state
        end case;
    end process compute_output_logic;
end architecture behavioural;
```