LAB 2: Multi-Gate Parking Controller System

VHDL Design and Simulation Lab Exercise

AUTHOR
Philippe Velha - Logic Networks

1 Introduction

Welcome to the lab2 project! In this project, you'll design and implement a VHDL-based parking management system that controls multiple entry and exit gates, tracks available parking spots, and manages payment processes.

Group Work

This is a **group project** designed to be completed before the exam. You are encouraged to:

- Discuss design approaches with other groups
- Share ideas about state machine implementations
- Help each other debug issues
- Compare simulation results

However, each group should write their own code, report and understand the complete design.

2 System Overview

2.1 The Parking Lot Layout

Your parking system manages a parking facility with:

- **Capacity**: 7 parking spots (configurable via generic)
- **Entry Gates**: 2 gates (Gin1 and Gin2)
- **Exit Gates**: 2 gates (Gout1 and Gout2)
- **Display**: 7-segment display showing available spots
- **Indicators**: Green/Red lights for availability status

2.2 How the System Works

2.2.1 Entry Process

1. **Car Detection**: When a car approaches an entry gate, sensor A (before the barrier) detects it
2. **Capacity Check**: System checks if parking spots are available
3. **Barrier Opens**: If spots available, the barrier opens
4. **Car Entry**: Car passes through, sensor B (after barrier) confirms passage
5. **Count Update**: Available spots decrease by 1
6. **Barrier Closes**: After car clears sensor B, barrier closes

2.2.2 Exit Process

1. **Car Detection**: Sensor A at exit gate detects departing car
2. **Payment Request**: System requests payment output payment_request to '1'
3. **Payment Processing**: Driver completes payment (not to be designed)
4. **Payment Confirmation**: System receives payment_done signal
5. **Barrier Opens**: Exit barrier opens
6. **Car Exit**: Car leaves, sensor B confirms
7. **Count Update**: Available spots increase by 1
8. **Barrier Closes**: Barrier closes after car clears

2.2.3 Visual Indicators

- **Green Light**: ON when spots are available
- **Red Light**: ON when parking is full
- **Payment Request**: We are going to assume a simple led
- **7-Segment Display**: Shows number of available spots (0-7)

3 System Architecture

3.1 Entity Interface

```
entity Parking_Controller is
    generic (
        PARKING_CAPACITY : integer := 7
    );
    port (
        -- Clock and Reset
        clk              : in  std_logic;
        nrst             : in  std_logic;

        -- Entry Gate 1 Sensors
        sensor_A_Gin1    : in  std_logic;   -- Before barrier
        sensor_B_Gin1    : in  std_logic;   -- After barrier

        -- Entry Gate 2 Sensors
        sensor_A_Gin2    : in  std_logic;
        sensor_B_Gin2    : in  std_logic;

        -- Exit Gate 1 Sensors
        sensor_A_Gout1   : in  std_logic;
        sensor_B_Gout1   : in  std_logic;

        -- Exit Gate 2 Sensors
        sensor_A_Gout2   : in  std_logic;
        sensor_B_Gout2   : in  std_logic;

        -- Payment Interface
        payment_done     : in  std_logic;
        payment_accepted : out std_logic;
        payment_request  : out std_logic;

        -- Barrier Controls
        barrier_Gin1     : out std_logic;   -- '1' = open
        barrier_Gin2     : out std_logic;
        barrier_Gout1    : out std_logic;
        barrier_Gout2     : out std_logic;

        -- Visual Indicators
        Green_Light       : out std_logic;
        Red_Light         : out std_logic;
        display           : out std_logic_vector(6 downto 0)
    );
end entity;
```

3.2 State Machines

Your design requires **four separate state machines**, one for each gate:

3.2.1 Entry Gate State Machine (Gin1 and Gin2)

```
States:
- IDLE: Waiting for car
- CAR_DETECTED: Sensor A triggered, checking capacity
- CAR_ENTERING: Barrier open, car passing through
- CAR_ENTERED: Car fully inside, preparing to close barrier
```

State Transitions:

```
IDLE → CAR_DETECTED: sensor_A = '1' AND car_count < CAPACITY
CAR_DETECTED → CAR_ENTERING: sensor_B = '1'
CAR_DETECTED → IDLE: sensor_A = '0' (car left)
CAR_ENTERING → CAR_ENTERED: sensor_A = '0' AND sensor_B = '1'
CAR_ENTERING → CAR_DETECTED: sensor_A = '1' AND sensor_B = '0' (reverse)
CAR_ENTERED → IDLE: sensor_B = '0' (car cleared)
```

3.2.2 Exit Gate State Machine (Gout1 and Gout2)

```
States:
- IDLE: Waiting for car
- WAIT_PAYMENT: Car detected, waiting for payment
- PAYMENT_OK: Payment accepted, barrier opening
- CAR_EXITING: Car passing through barrier
- CAR_EXITED: Car fully outside
```

State Transitions:

```
IDLE → WAIT_PAYMENT: sensor_A = '1' AND car_count > 0
WAIT_PAYMENT → WAIT_PAYMENT: sensor_A = '0' (loop until stays)
WAIT_PAYMENT → PAYMENT_OK: payment_done = '1'
PAYMENT_OK → CAR_EXITING: sensor_B = '1'
CAR_EXITING → CAR_EXITED: sensor_A = '0' AND sensor_B = '1'
CAR_EXITED → IDLE: sensor_B = '0'
```

4 Design Tasks

All the tasks provide some hints and suggestions. Modifications are accepted if you justify them by a proper argument. Your are the designers, you choose. However, I am the client therefore you need to convince me if you change something.

4.1 Task 1: Define Types and Signals

Start by defining the necessary types and internal signals:

```
architecture Behavioral of Parking_Controller is
    -- State machine types
    type gate_state_type is (IDLE, CAR_DETECTED, CAR_ENTERING, CAR_ENTERED
    type exit_state_type is (IDLE, WAIT_PAYMENT, PAYMENT_OK,
                             CAR_EXITING, CAR_EXITED);

    -- State signals for each gate
    signal state_Gin1, next_state_Gin1 : gate_state_type;
    signal state_Gin2, next_state_Gin2 : gate_state_type;
    signal state_Gout1, next_state_Gout1 : exit_state_type;
    signal state_Gout2, next_state_Gout2 : exit_state_type;

    -- Car counter
    signal car_count : integer range 0 to PARKING_CAPACITY := 0;

    -- Edge detection for sensors
    signal sensor_A_Gin1_prev, sensor_B_Gin1_prev : std_logic;
    -- ... (add for other gates)

    -- Increment/decrement flags
    signal inc_Gin1, inc_Gin2 : std_logic;
    signal dec_Gout1, dec_Gout2 : std_logic;

    -- Payment signals
    signal payment_accepted_Gout1, payment_accepted_Gout2 : std_logic;

begin
```

Design Hint

Use separate signals for each gate's state machine to keep the logic clear and maintainable. Each gate operates independently!

4.2 Task 2: State Register Process

Implement the synchronous state register:

```
stateregister: process(clk, nrst)
begin
    if nrst = '0' then
        -- Reset all states to IDLE
        state_Gin1 <= IDLE;
        state_Gin2 <= IDLE;
        state_Gout1 <= IDLE;
        state_Gout2 <= IDLE;

        -- Reset sensor history
        sensor_A_Gin1_prev <= '0';
        sensor_B_Gin1_prev <= '0';
        -- ... (reset others)
    elsif rising_edge(clk) then
        -- Update states
        state_Gin1 <= next_state_Gin1;
        -- ... (update others)

        -- Store sensor values for edge detection
        sensor_A_Gin1_prev <= sensor_A_Gin1;
        -- ... (store others)
    end if;
end process;
```

4.3 Task 3: Entry Gate State Machines

Implement the combinational logic for entry gates. Here's a template for Gin1 (to be completed and modified accordingly):

```
gin1: process(state_Gin1, sensor_A_Gin1, sensor_B_Gin1,
              sensor_A_Gin1_prev, sensor_B_Gin1_prev, car_count)
begin
    -- Default values
    next_state_Gin1 <= state_Gin1;
    barrier_Gin1 <= '0';
    inc_Gin1 <= '0';

    case state_Gin1 is
        when IDLE =>
            -- Check for car arrival and capacity
            if sensor_A_Gin1 = '1' and car_count < PARKING_CAPACITY then
                next_state_Gin1 <= CAR_DETECTED;
            end if;

        when CAR_DETECTED =>
            -- TODO: Implement transitions
            -- - Back to IDLE if car leaves
            -- - To CAR_ENTERING if sensor B triggered

        when CAR_ENTERING =>
            barrier_Gin1 <= '1';  -- Keep barrier open
            -- TODO: Implement transitions
            -- - To CAR_ENTERED when car fully inside
            -- - Handle reverse scenario

        when CAR_ENTERED =>
            barrier_Gin1 <= '1';
            -- TODO: Implement transition back to IDLE
            -- Set inc_Gin1 to increment counter

    end case;
end process;
```

Critical Logic

Make sure to:

1. Only open barriers when there's capacity (entry gates)

2. Only open barriers after payment (exit gates)
3. Properly detect when car has fully passed (sensor A = 0, sensor B = 1)
4. Set increment/decrement flags at the right time

4.4 Task 4: Exit Gate State Machines

Implement exit gates similarly. Key differences:

- Check `car_count > 0` instead of capacity
- Handle payment request and acceptance
- Use 5 states instead of 4 but can be modified

```
gout1: process(state_Gout1, sensor_A_Gout1, sensor_B_Gout1,
               payment_done, car_count)
begin
    -- Default values
    next_state_Gout1 <= state_Gout1;
    barrier_Gout1 <= '0';
    payment_request <= '0';
    payment_accepted_Gout1 <= '0';
    dec_Gout1 <= '0';

    case state_Gout1 is
        when IDLE =>
            -- TODO: Detect car and check if parking has cars

        when WAIT_PAYMENT =>
            payment_request <= '1';  -- Turn on payment light
            -- TODO: Wait for payment or car to leave

        when PAYMENT_OK =>
            payment_accepted_Gout1 <= '1';
            -- TODO: Wait for car to start exiting

        when CAR_EXITING =>
            barrier_Gout1 <= '1';
            -- TODO: Detect when car fully exits

        when CAR_EXITED =>
            barrier_Gout1 <= '1';
            -- TODO: Return to IDLE and decrement counter

    end case;
end process;
```

4.5 Task 5: Car Counter Management

Implement the counter that tracks available spots:

```
countermgt: process(clk, nrst)
begin
    if nrst = '0' then
        car_count <= 0;
    elsif rising_edge(clk) then
```

```
        -- Handle increments from entry gates
        if inc_Gin1 = '1' then
            car_count <= car_count + 1;
        elsif inc_Gin2 = '1' then
            car_count <= car_count + 1;
        -- Handle decrements from exit gates
        elsif dec_Gout1 = '1' then
            car_count <= car_count - 1;
        elsif dec_Gout2 = '1' then
            car_count <= car_count - 1;
        end if;
    end if;
end process;
```

Race Condition

Notice the `elsif` structure. Why? What happens if two cars enter/exit simultaneously? Discuss with your group how to handle this in a real system!

4.6 Task 6: 7-Segment Display Decoder

Implement a function to convert the available spots count to 7-segment display:

```
function int_to_7seg(value : integer) return std_logic_vector is
    variable result : std_logic_vector(6 downto 0);
begin
    case value is
        when 0 => result := "0000001";  -- Display '0'
        when 1 => result := "1001111";  -- Display '1'
        when 2 => result := "0010010";  -- Display '2'
        when 3 => result := "0000110";  -- Display '3'
        when 4 => result := "1001100";  -- Display '4'
        when 5 => result := "0100100";  -- Display '5'
        when 6 => result := "0100000";  -- Display '6'
        when 7 => result := "0001111";  -- Display '7'
        when others => result := "1111111";  -- Error
    end case;
    return result;
end function;
```

Then use it:

```
-- Display available spots
display <= int_to_7seg(PARKING_CAPACITY - car_count);

-- Control indicator lights
Green_Light <= '1' when car_count < PARKING_CAPACITY else '0';
Red_Light <= '1' when car_count = PARKING_CAPACITY else '0';

-- Payment accepted from either exit gate
payment_accepted <= payment_accepted_Gout1 or payment_accepted_Gout2;
```

5 Testing Your Design

5.1 Testbench Overview

Two testbenches are provided:

5.1.1 Test Case 1 (parking_tb_case1.vhd)

Scenario: Basic functionality test

- Tests single car entry through Gin1
- Tests single car exit through Gout1 with payment
- Verifies counter increments and decrements
- Checks display updates

Expected Behavior:

1. Initially: car_count = 0, display shows “7”
2. After entry: car_count = 1, display shows “6”
3. After exit: car_count = 0, display shows “7”

5.1.2 Test Case 2 (parking_tb_case2.vhd)

Scenario: Advanced test with multiple gates

- Multiple cars entering through different gates
- Concurrent operations
- Full parking lot scenario
- Multiple exits with payment processing

Expected Behavior:

- System handles simultaneous gate operations
- Correctly rejects entry when full
- Payment system works for all exit gates

5.2 Running Simulations

1. **Compile** your design:

```
   ghdl -a carpark2gates.vhd
   ghdl -a parking_tb_case1.vhd
```

2. **Elaborate or compile**:

```
   ghdl -e parking_tb_case1
```

This might not work in some platforms:

```
   ghdl -c carpark2gates.vhd parking_tb_case1.vhd
```

3. **Run simulation**:

```
ghdl -r parking_tb_case1 --vcd=case1.vcd --stop-time=xxus
```

4. **View waveforms**:

```
gtkwave case1.vcd
```

or use a plugin in vscode

5.3 What to Observe in Waveforms

Pay attention to:

- **The test benches**: have already some warning and errors that are checked
- **State transitions**: Are they happening at the right time?
- **Barrier control**: Opens only when appropriate?
- **Counter updates**: Increments/decrements correctly?
- **Sensor sequences**: Proper detection of car passage?
- **Payment flow**: Request → Done → Acceptance → Barrier?
- **Timing**: No glitches or unexpected behavior?

Debugging Tips

If your simulation doesn't work:

1. Check state transitions one gate at a time
2. Add `report` statements to track state changes (look the other assessments)
3. Verify sensor edge detection logic
4. Ensure counter increments happen only once per car
5. Check that barriers close after cars pass

6 Deliverables

6.1 What to Submit

1. **VHDL Source Code**: `carpark2gates.vhd`

2. **Simulation Reports**: Screenshots of waveforms showing:

   - the different Test Cases (1,2,3,4) passing
   - Key signals annotated
   - you can add the vcd files

3. **Design Document** (brief):

   - State machine diagrams if modified or if you want to include your annotated algorithmic state machine
   - Explanation of design choices
   - Any assumptions made
   - Known limitations and what could be improved

7 Discussion Questions (Optional in the report)

As you work, consider these questions with your group:

1. **Concurrency**: What happens if two cars arrive at different entry gates at exactly the same time?
2. **Failure Modes**: What if a sensor gets stuck? How could you detect this?
3. **Real-World Concerns**:
   - How would you handle sensor noise?
   - What about cars backing up?
   - Network connectivity for payment systems?
4. **Scalability**: How would you modify this for:
   - 100 parking spots?
   - 10 entry/exit gates?
   - Multiple levels?
5. **State Machine Design**: How could you change it to improve the system?

8 Resources

8.1 VHDL Quick Reference

Common Syntax:

```
-- Conditional signal assignment
signal <= value1 when condition else value2;

-- If statement
if condition then
    -- statements
elsif other_condition then
    -- statements
else
    -- statements
end if;

-- Case statement
case signal is
    when value1 => -- statements
    when value2 => -- statements
    when others => -- statements
end case;
```

8.2 Useful Libraries

Already included in your template:

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

9 Getting Started Checklist

- [ ] Read through entire document
- [ ] Discuss overall architecture approach
- [ ] Sketch algorithmic state machine diagrams on paper
- [ ] Set up VHDL development environment
- [ ] Create entity and architecture skeleton (using templates)
- [ ] Implement and test one gate at a time by creating a simplified testbench
- [ ] Test with Case 1 testbench
- [ ] Debug and refine
- [ ] Test with Case 2 testbench
- [ ] Test with Case 3 testbench
- [ ] Test with Case 4 testbench
- [ ] Document your design

10 Good Luck!

Remember: This is a learning exercise. Don't be afraid to:

- Ask questions
- Try different approaches
- Make mistakes and learn from them
- Collaborate with other groups
- Explain your reasoning to others

The goal is to understand digital system design, state machines, and VHDL, not just to get working code. Have fun! 🚗 🅿️