1. Streetlight Control During Fog

Description

In an intelligent city, streetlights need to automatically turn ON during foggy conditions to avoid accidents.
If the visibility drops below 50 meters, streetlights must turn ON.
Otherwise, they must stay OFF.

Input/Output Details

- **Inputs:**
  - `visibility` (integer: 0 to 500)
- **Outputs:**
  - `light_status` (1-bit: 0 or 1)

Inputs Explanation

- `visibility`: Current visibility range measured in meters.

Outputs Explanation

- `1` → Turn ON streetlight
- `0` → Keep streetlight OFF

Constraints

- 0 <= visibility <= 500

Hints

- Use simple conditional checking:
  - If visibility < 50 → light_status = 1
  - Else → light_status = 0

Tutorial Content

In combinational circuits, decision-making based on input values is handled by simple conditions.
Here, based on the visibility, you just need to generate a HIGH or LOW signal for the streetlight.

Important Concepts:

- Comparator
- Simple IF-ELSE logic
- Output assignment based on input condition

Solution Guide

You can use a simple `assign` statement (in Verilog/SystemVerilog) or a `when-else` (in VHDL).

Test Cases

Test Cases to Display

json
CopyEdit

```
[
  { "input": "30", "output": "1" },
  { "input": "70", "output": "0" }
]
```

Hidden Test Cases

json
CopyEdit

```
[
  { "input": "10", "output": "1" },
  { "input": "100", "output": "0" }
]
```

Solutions

✅ Verilog Solution

verilog
CopyEdit

```
module streetlight_control(
    input [8:0] visibility, // 9 bits to cover up to 500
    output light_status
);

assign light_status = (visibility < 50) ? 1'b1 : 1'b0;

endmodule
```

✅ VHDL Solution

vhdl
CopyEdit

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

```
entity streetlight_control is
    Port ( visibility : in  STD_LOGIC_VECTOR (8 downto 0);
           light_status : out  STD_LOGIC);
end streetlight_control;

architecture Behavioral of streetlight_control is
begin
    process(visibility)
    begin
        if to_integer(unsigned(visibility)) < 50 then
            light_status <= '1';
        else
            light_status <= '0';
        end if;
    end process;
end Behavioral;
```

✅ System Verilog Solution

systemverilog
CopyEdit

```
module streetlight_control(
    input logic [8:0] visibility,
    output logic light_status
);

always_comb begin
    if (visibility < 50)
        light_status = 1;
    else
        light_status = 0;
end

endmodule
```

2. Temperature Monitoring for Data Centers

Description

In a data center, server rooms must be monitored to avoid overheating.
If the room temperature crosses 30°C, an alarm must be triggered.
Otherwise, the alarm should stay OFF.

Input/Output Details

- **Inputs:**
  - `temperature` (integer: 0 to 100)
- **Outputs:**
  - `alarm` (1-bit: 0 or 1)

Inputs Explanation

- `temperature`: The current measured room temperature.

Outputs Explanation

- `1` → Alarm ON (Overheating)
- `0` → Alarm OFF (Normal)

Constraints

- 0 <= temperature <= 100

Hints

- Compare temperature against the threshold value (30).

Tutorial Content

Threshold-based monitoring is a basic logic design used in many industries like automotive, healthcare, and data centers.
Simple comparison operators can help design such decision systems.

Concepts Used:

- Comparators
- Threshold checking
- Output toggling based on condition

Solution Guide

Same approach as before:
If temperature > 30 → alarm = 1
Else → alarm = 0

Test Cases

Test Cases to Display

json
CopyEdit

```
[
  { "input": "28", "output": "0" },
  { "input": "35", "output": "1" }
]
```

Hidden Test Cases

json
CopyEdit

```
[
  { "input": "30", "output": "0" },
  { "input": "45", "output": "1" }
]
```

Solutions

✅ Verilog Solution

verilog
CopyEdit

```
module temperature_monitor(
    input [7:0] temperature,
    output alarm
);

assign alarm = (temperature > 30) ? 1'b1 : 1'b0;

endmodule
```

✅ VHDL Solution

vhdl
CopyEdit

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

```
entity temperature_monitor is
    Port ( temperature : in  STD_LOGIC_VECTOR (7 downto 0);
           alarm : out  STD_LOGIC);
end temperature_monitor;

architecture Behavioral of temperature_monitor is
begin
    process(temperature)
    begin
        if to_integer(unsigned(temperature)) > 30 then
            alarm <= '1';
        else
            alarm <= '0';
        end if;
    end process;
end Behavioral;
```

✅ System Verilog Solution

systemverilog
CopyEdit

```
module temperature_monitor(
    input logic [7:0] temperature,
    output logic alarm
);

always_comb begin
    if (temperature > 30)
        alarm = 1;
    else
        alarm = 0;
end

endmodule
```

3. Automatic Street Light Controller

Description

Street lights should automatically turn ON after sunset and OFF after sunrise.
You are tasked with designing a controller that takes **light intensity** as input.
If light intensity is low (below 40%), the lights should turn ON.
If light intensity is high (above or equal to 40%), the lights should stay OFF.

Input/Output Details

- **Inputs:**
  - `light_intensity` (integer: 0 to 100)
- **Outputs:**
  - `light_status` (1-bit: 0 or 1)

Inputs Explanation

- `light_intensity`: Percentage of ambient light (0% = complete darkness, 100% = full daylight).

Outputs Explanation

- `1` → Light ON (Dark environment)
- `0` → Light OFF (Bright environment)

Constraints

- 0 <= light_intensity <= 100

Hints

- Use simple comparison operations.
- Threshold is 40%.

Tutorial Content

Automatic lighting control is widely used in smart cities.
Threshold-based control reduces manual effort and saves energy.
This simple logic circuit can be later expanded into IoT-based systems.

Concepts Used:

- Comparator circuits
- Output control based on input conditions

Solution Guide

- Compare light intensity with 40.
- If it’s less than 40 → Light ON.
- Else → Light OFF.

Test Cases

Test Cases to Display

json
CopyEdit

```
[
  { "input": "35", "output": "1" },
  { "input": "50", "output": "0" }
]
```

Hidden Test Cases

json
CopyEdit

```
[
  { "input": "40", "output": "0" },
  { "input": "10", "output": "1" }
]
```

Solutions

✅ Verilog Solution

verilog
CopyEdit

```
module street_light_controller(
    input [7:0] light_intensity,
    output light_status
);

assign light_status = (light_intensity < 40) ? 1'b1 : 1'b0;

endmodule
```

✅ VHDL Solution

vhdl
CopyEdit

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

entity street_light_controller is
    Port ( light_intensity : in  STD_LOGIC_VECTOR (7 downto 0);
           light_status : out  STD_LOGIC);
end street_light_controller;

architecture Behavioral of street_light_controller is
begin
    process(light_intensity)
    begin
        if to_integer(unsigned(light_intensity)) < 40 then
            light_status <= '1';
        else
            light_status <= '0';
        end if;
    end process;
end Behavioral;
```

✅ System Verilog Solution

systemverilog
CopyEdit

```systemverilog
module street_light_controller(
    input logic [7:0] light_intensity,
    output logic light_status
);

always_comb begin
    if (light_intensity < 40)
        light_status = 1;
    else
        light_status = 0;
end
```

```
endmodule
```

4. Arjuna's Archery Target Detection System

Description

Arjuna, the greatest archer, is practicing blindfolded archery.
He must identify if the sound of the **moving target** is coming from the left side or right side.
He uses a **sound sensor**, and based on the signal strength, he must immediately decide:

- If signal from the **left** is stronger → Turn left.
- If signal from the **right** is stronger → Turn right.
- If both signals are equal → Stay still and prepare.

You must design a simple decision system that takes two inputs (left signal, right signal) and generates an output command.

Input/Output Details

- **Inputs:**
  - `left_signal` (integer: 0 to 255)
  - `right_signal` (integer: 0 to 255)
- **Outputs:**
  - `command` (2 bits):
    - `00` → Stay

- `01` → Turn Left
- `10` → Turn Right

Inputs Explanation

- `left_signal`: Strength of sound from the left side.
- `right_signal`: Strength of sound from the right side.

Outputs Explanation

- `00`: Stay still (both signals are equal)
- `01`: Turn left (left side is stronger)
- `10`: Turn right (right side is stronger)

Constraints

- 0 <= left_signal, right_signal <= 255

Hints

- Use simple comparison: greater-than, less-than, equality checking.

Tutorial Content

In smart systems like autonomous vehicles and robotic arms, decision-making based on environmental sensor data is critical.
Similarly, Arjuna uses sound sensors to make quick movement decisions.
Designing such systems teaches fundamental comparison and decision logic — the foundation of control systems.

Concepts Used:

- Comparison of two signals
- Output encoding based on comparison

Solution Guide

Approach:

- Compare left_signal and right_signal
- If left_signal > right_signal → 01 (turn left)
- If right_signal > left_signal → 10 (turn right)
- If equal → 00 (stay)

Test Cases

Test Cases to Display

json
CopyEdit

```
[
  { "input": "120 100", "output": "01" },
  { "input": "80 90", "output": "10" },
  { "input": "50 50", "output": "00" }
]
```

Hidden Test Cases

json
CopyEdit

```
[
  { "input": "200 100", "output": "01" },
  { "input": "150 200", "output": "10" },
  { "input": "0 0", "output": "00" }
]
```

Solutions

✅ Verilog Solution

verilog
CopyEdit

```
module arjuna_target_direction(
    input [7:0] left_signal,
    input [7:0] right_signal,
    output reg [1:0] command
);

always @(*) begin
    if (left_signal > right_signal)
        command = 2'b01; // Turn Left
    else if (right_signal > left_signal)
        command = 2'b10; // Turn Right
    else
        command = 2'b00; // Stay
end

endmodule
```

✅ VHDL Solution

vhdl
CopyEdit

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

entity arjuna_target_direction is
    Port ( left_signal : in  STD_LOGIC_VECTOR (7 downto 0);
           right_signal : in  STD_LOGIC_VECTOR (7 downto 0);
           command : out  STD_LOGIC_VECTOR (1 downto 0));
end arjuna_target_direction;

architecture Behavioral of arjuna_target_direction is
begin
    process(left_signal, right_signal)
    begin
        if to_integer(unsigned(left_signal)) >
to_integer(unsigned(right_signal)) then
            command <= "01";
        elsif to_integer(unsigned(right_signal)) >
to_integer(unsigned(left_signal)) then
            command <= "10";
        else
            command <= "00";
        end if;
    end process;
end Behavioral;
```

✅ System Verilog Solution

systemverilog
CopyEdit

```
module arjuna_target_direction(
    input logic [7:0] left_signal,
    input logic [7:0] right_signal,
    output logic [1:0] command
);

always_comb begin
    if (left_signal > right_signal)
```

```
        command = 2'b01; // Turn Left
    else if (right_signal > left_signal)
        command = 2'b10; // Turn Right
    else
        command = 2'b00; // Stay
end

endmodule
```

5. Krishna's Cattle Monitoring System

Description

Lord Krishna, known for his love of cows, is managing a large herd in Gokul.
He wants a **simple system** that **counts** how many cows have crossed a small bridge during the day.

Every time a cow crosses, a **sensor** sends a pulse (high signal '1').
When no cow is crossing, the signal is '0'.
You need to build a system that **counts** the number of '1's seen.

> Krishna wants to check how many cows safely crossed the bridge at sunset! 🌇

Input/Output Details

- **Inputs:**
  - `sensor_signal` (1-bit, 0 or 1) — keeps coming continuously
  - `clk` (clock pulse) — new data every clock edge

- **Outputs:**
  - `cow_count` (integer output)

Inputs Explanation

- `sensor_signal`: 1 means cow crossing, 0 means no cow crossing.

Outputs Explanation

- `cow_count`: Total number of cows crossed.

Constraints

- Only count rising '1's (no double-counting if signal stays high).
- Reset after 255 cows (mod 256).

Hints

- Detect rising edge if needed.
- Use a counter that increments when signal is 1.

Tutorial Content

In real-time systems like animal farms, highways, and industrial automation, counting moving objects is important.
This simple exercise builds intuition for **event detection**, **counting**, and **basic finite monitoring**.

Concepts Used:

- Monitoring Input Signals
- Counting Pulses
- Simple Register Update

Solution Guide

Approach:

- Monitor sensor_signal at each clock cycle.
- If signal == 1 and it was previously 0 → increment count.
- Reset counter if it reaches 256.

Test Cases

Test Cases to Display

json
CopyEdit

```
[
  { "input": "sensor_signal: [0,1,0,1,0,0,1]", "output": "3" },
  { "input": "sensor_signal: [1,1,1,0,0,1]", "output": "2" }
]
```

Hidden Test Cases

json
CopyEdit

```
[
  { "input": "sensor_signal: [0,0,0,1,1,0,1,0,1]", "output": "3" },
  { "input": "sensor_signal: [1,0,1,0,1,0]", "output": "3" }
]
```

Solutions

✅ Verilog Solution

verilog
CopyEdit

```
module krishna_cattle_counter(
    input clk,
    input sensor_signal,
    output reg [7:0] cow_count
);

reg prev_signal;

always @(posedge clk) begin
    prev_signal <= sensor_signal;

    if (sensor_signal && ~prev_signal) begin
        cow_count <= cow_count + 1;
    end
end

endmodule
```

✅ VHDL Solution

vhdl
CopyEdit

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

entity krishna_cattle_counter is
    Port ( clk : in  STD_LOGIC;
           sensor_signal : in  STD_LOGIC;
           cow_count : out  STD_LOGIC_VECTOR (7 downto 0));
```

```
end krishna_cattle_counter;

architecture Behavioral of krishna_cattle_counter is
signal count : unsigned(7 downto 0) := (others => '0');
signal prev_signal : STD_LOGIC := '0';
begin

process(clk)
begin
    if rising_edge(clk) then
        prev_signal <= sensor_signal;

        if (sensor_signal = '1' and prev_signal = '0') then
            count <= count + 1;
        end if;
    end if;
end process;

cow_count <= std_logic_vector(count);

end Behavioral;
```

✅ System Verilog Solution

systemverilog
CopyEdit

```
module krishna_cattle_counter(
    input logic clk,
    input logic sensor_signal,
    output logic [7:0] cow_count
);

logic prev_signal;

always_ff @(posedge clk) begin
    prev_signal <= sensor_signal;

    if (sensor_signal && !prev_signal) begin
```

```
        cow_count <= cow_count + 1;
    end
end

endmodule
```

Advanced Digital Pattern Matching with Constraints

Description

In a smart data transmission system, you are tasked with implementing a pattern detection module. This module must search for a **dynamic bit-pattern** within a continuous **data stream** in real-time.

Real-Life Scenario:

Imagine you're Arjuna, working on a high-security communication system where encrypted data packets are being transmitted in real-time. These packets consist of bits that are dynamically generated and transmitted. Arjuna needs to detect certain **patterns of bits** (representing specific commands or messages) within the stream. For example, if a particular pattern "110101" appears, the system needs to flag it as a critical event and trigger a response. However, the challenge is that the pattern is not fixed—it can change depending on certain parameters, such as the time of day or security protocols.

Your job is to design the pattern matching logic that efficiently finds these dynamically changing patterns in the bit stream while meeting the following constraints:

- The pattern changes based on a **timer value** that cycles every 5 minutes.
- The system has to operate at very high speeds, processing millions of bits per second.
- The system should have minimal latency and maximum throughput.

Task:

- Implement a dynamic **bit pattern detector** that matches different bit patterns based on an internal timer (which updates every 5 minutes).

- When the timer reaches certain intervals, a new pattern is selected.
- The pattern should be matched within a continuous incoming stream of bits.
- The system should flag an alert when the pattern is found, even if there is a delay of a few bits (tolerant of small bit misalignments).

Functional Requirements:

1. **Dynamic Pattern Update:** The bit pattern used for matching will change every 5 minutes based on a timer value (in real-time).
2. **Pattern Matching:** The system needs to efficiently match a specific pattern against a continuous stream of incoming bits (streaming data).
3. **Alert Mechanism:** Upon matching the pattern, an alert signal should be raised.
4. **Bit Misalignment Tolerance:** If the pattern is delayed or misaligned by a few bits, the system should still identify it correctly within a **5-bit window**.

Input/Output Details

Inputs:

1. **data_stream** (N bits) — Continuous stream of incoming bits that need to be scanned for a pattern.
2. **timer** (3 bits) — A timer value that is updated every 5 minutes. This determines which bit pattern is currently active.
3. **pattern_change_flag** (1 bit) — A flag that indicates when the pattern changes. The pattern updates based on the timer value.

Outputs:

1. **pattern_found** (1 bit) — A flag that goes HIGH when the matching pattern is found in the data stream.

2. **current_pattern** (N bits) — The current pattern used for matching.
3. **alert_signal** (1 bit) — An output signal that alerts when a pattern is detected.

Inputs Explanation:

- **data_stream**: A real-time stream of bits, for example, 64-bits long.
- **timer**: This will update every 5 minutes, influencing the pattern to be detected.
- **pattern_change_flag**: This flag indicates when a pattern needs to be updated based on the timer.

Outputs Explanation:

- **pattern_found**: If the data stream matches the current pattern, this output becomes HIGH.
- **current_pattern**: This is the bit pattern being searched for in the current time window.
- **alert_signal**: Once the **pattern_found** is HIGH, the alert signal is triggered to notify the system.

Constraints:

1. The data stream length (N) can range from 32 to 128 bits.
2. The timer is updated every 5 minutes, so there are multiple patterns to search for over time.
3. The system must operate at high throughput, processing **millions of bits per second**.
4. The solution must have **minimal latency** in pattern detection and flagging alerts.
5. The system should be **tolerant** of minor bit misalignments (within a 5-bit window).

Hints:

- **Finite State Machine (FSM)**: Implement a FSM to handle the real-time pattern changes and matching logic.
- **Shift Registers**: Use shift registers for pattern matching to handle bit streams effectively.
- **Bitwise Operations**: Use bitwise operations to efficiently compare portions of the stream to the current pattern.
- Consider **buffering** and **timing synchronization** to ensure minimal latency while searching the data stream.
- **Pattern Matching Algorithms**: Implement an efficient pattern matching algorithm such as the **Knuth-Morris-Pratt (KMP)** or **Rabin-Karp** algorithm to handle large bit streams with minimal delays.

Tutorial Content:

This problem combines **real-time pattern matching** with **dynamic pattern updates**. Understanding how to handle high-speed data streams and the synchronization of pattern changes is critical in this problem. The challenge lies in building an efficient system that can process millions of bits per second while ensuring accurate detection of dynamic patterns.

Concepts Used:

- **Finite State Machines (FSM)**: For state transitions and handling time-based changes in patterns.
- **Shift Registers**: For efficiently moving and comparing the data stream.
- **Bitwise Operations**: For direct manipulation of bits in the data stream.
- **Pattern Matching Algorithms**: Efficient algorithms to match patterns in real-time.

Solution Guide:

1. Use a **shift register** to store incoming bits from the data stream.
2. Implement a **finite state machine (FSM)** that updates the current pattern every 5 minutes based on the timer.
3. Use **bitwise comparison** to search for the current pattern in the data stream.
4. If the pattern is found within a 5-bit window, trigger the **pattern_found** flag and generate an alert.
5. Ensure the system can handle a **large stream** of bits without overwhelming the hardware.

Test Cases

Test Cases to Display

json
CopyEdit

```
[
  { "input": "data_stream: 1101010101010101, timer: 000,
pattern_change_flag: 1", "output": "pattern_found: 1, alert_signal: 1"
},
  { "input": "data_stream: 1010110011010101, timer: 001,
pattern_change_flag: 1", "output": "pattern_found: 0, alert_signal: 0"
},
  { "input": "data_stream: 1011010101011101, timer: 010,
pattern_change_flag: 1", "output": "pattern_found: 1, alert_signal: 1"
}
]
```

Hidden Test Cases

json
CopyEdit

```
[
```

```
  { "input": "data_stream: 0101010101010101, timer: 011,
pattern_change_flag: 1", "output": "pattern_found: 0, alert_signal: 0"
},
  { "input": "data_stream: 1101101101010101, timer: 100,
pattern_change_flag: 1", "output": "pattern_found: 1, alert_signal: 1"
}
]
```

Verilog Solution

verilog
CopyEdit

```
module pattern_detector(
    input [127:0] data_stream,
    input [2:0] timer,
    input pattern_change_flag,
    output reg pattern_found,
    output reg [127:0] current_pattern,
    output reg alert_signal
);

reg [127:0] pattern_memory;
reg [127:0] shift_register;

always @(posedge pattern_change_flag or posedge timer) begin
    case(timer)
        3'b000: pattern_memory <=
128'b11010101010101011010101010101010; // Example pattern 1
        3'b001: pattern_memory <=
128'b10101100110101011010011010101010; // Example pattern 2
        3'b010: pattern_memory <=
128'b10110101010111011010101010101010; // Example pattern 3
        // Add more patterns as necessary
        default: pattern_memory <= 128'b0;
    endcase
end

always @(data_stream or pattern_memory) begin
```

```
    shift_register <= data_stream;
    if (shift_register == pattern_memory) begin
        pattern_found <= 1;
        alert_signal <= 1;
    end else begin
        pattern_found <= 0;
        alert_signal <= 0;
    end
end

endmodule
```

VHDL Solution

vhdl
CopyEdit

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity pattern_detector is
    Port ( data_stream : in  STD_LOGIC_VECTOR (127 downto 0);
           timer : in  STD_LOGIC_VECTOR (2 downto 0);
           pattern_change_flag : in  STD_LOGIC;
           pattern_found : out  STD_LOGIC;
           current_pattern : out  STD_LOGIC_VECTOR (127 downto 0);
           alert_signal : out  STD_LOGIC);
end pattern_detector;

architecture Behavioral of pattern_detector is
    signal pattern_memory : STD_LOGIC_VECTOR(127 downto 0);
    signal shift_register : STD_LOGIC_VECTOR(127 downto 0);
begin
    process(pattern_change_flag, timer)
    begin
        if pattern_change_flag = '1' then
            case timer is
```

```
                when "000" => pattern_memory <=
"11010101010101011010101010101010"; -- Example pattern 1
                when "001" => pattern_memory <=
"10101100110101011010011010101010"; -- Example pattern 2
                when "010" => pattern_memory <=
"10110101010111011010101010101010"; -- Example pattern 3
                -- Add more patterns as necessary
                when others => pattern_memory <=
"00000000000000000000000000000000";
            end case;
        end if;
    end process;

    process(data_stream, pattern_memory)
    begin
        shift_register <= data_stream;
        if shift_register = pattern_memory then
            pattern_found <= '1';
            alert_signal <= '1';
        else
            pattern_found <= '0';
            alert_signal <= '0';
        end if;
    end process;
end Behavioral;
```

System Verilog Solution

systemverilog
CopyEdit

```
module pattern_detector(
    input logic [127:0] data_stream,
    input logic [2:0] timer,
    input logic pattern_change_flag,
    output logic pattern_found,
    output logic [127:0] current_pattern,
    output logic alert_signal
);
```

```
logic [127:0] pattern_memory;
logic [127:0] shift_register;

always_ff @(posedge pattern_change_flag or posedge timer) begin
    case(timer)
        3'b000: pattern_memory <=
128'b11010101010101011010101010101010; // Example pattern 1
        3'b001: pattern_memory <=
128'b10101100110101011010011010101010; // Example pattern 2
        3'b010: pattern_memory <=
128'b10110101010111011010101010101010; // Example pattern 3
        default: pattern_memory <= 128'b0;
    endcase
end

always_ff @(data_stream or pattern_memory) begin
    shift_register <= data_stream;
    if (shift_register == pattern_memory) begin
        pattern_found <= 1;
        alert_signal <= 1;
    end else begin
        pattern_found <= 0;
        alert_signal <= 0;
    end
end

endmodule
```

Dynamic Traffic Signal Controller with Adaptive Timing

Description

In a smart city, traffic signals need to adjust their timing dynamically based on real-time traffic conditions. A control system needs to manage multiple traffic signals across different intersections, adjusting their timing based on the traffic flow. The system should adapt the timing of green, yellow, and red signals based on live vehicle detection.

Real-Life Scenario:

Imagine Arjuna is designing a **smart city traffic management system** to optimize the flow of traffic in a busy metropolitan area. The traffic signal system needs to adjust in real-time to the traffic situation. For example, if the system detects high traffic flow on one road, it should extend the green signal duration to clear more cars. On the other hand, if a road is empty, the signal should shorten the green light duration and move quickly to the next phase.

Your task is to design a **dynamic traffic signal controller** that adapts to traffic flow by adjusting the timing of each signal's phases. You need to develop a system where the controller dynamically changes the green, yellow, and red light durations based on the number of vehicles detected at each intersection.

The challenge includes:

1. Multiple traffic lights at different intersections.
2. Real-time detection of vehicle count on each road.
3. Adaptive green light duration based on vehicle detection.

Task:

- Create a digital design that controls the timing of traffic lights at an intersection.
- The system will adapt the green light duration based on the vehicle count detected on the incoming road.
- It must handle multiple intersections and allow different patterns of green, yellow, and red light durations.

- The system needs to prioritize certain roads over others when traffic congestion is detected.

Input/Output Details

Inputs:

1. **vehicle_count** (4 bits) — The number of vehicles detected on the road (0 to 15 vehicles).

2. **intersection_request** (1 bit) — A flag to indicate whether an intersection needs to change the light due to high traffic.

3. **timer** (16-bit counter) — A counter for keeping track of time intervals for each light phase.

4. **emergency_signal** (1 bit) — A flag that signals if there is an emergency vehicle requiring priority (e.g., fire truck, ambulance).

Outputs:

1. **green_light** (1 bit) — Indicates whether the green light is on.

2. **yellow_light** (1 bit) — Indicates whether the yellow light is on.

3. **red_light** (1 bit) — Indicates whether the red light is on.

4. **extended_green_signal** (1 bit) — Indicates whether the green light should be extended due to heavy traffic.

5. **phase_timer** (16-bit) — A timer that controls how long each light phase lasts.

Inputs Explanation:

- **vehicle_count**: This represents the number of vehicles detected at a particular intersection. If the count is high, the green signal should last longer.

- **intersection_request**: This flag is set if the intersection's traffic signal needs to change to accommodate more cars.

- **timer**: Keeps track of time and controls the light phase transitions.

- **emergency_signal**: This flag is used to override the normal traffic flow and immediately prioritize the green light for emergency vehicles.

Outputs Explanation:

- **green_light**: The green light is activated if the vehicle count is high, or if the system detects a request for priority on a road.

- **yellow_light**: The yellow light is activated just before the red light to indicate a phase transition.

- **red_light**: The red light is activated to stop traffic.

- **extended_green_signal**: If there is heavy traffic, the green light will be extended.

- **phase_timer**: This timer will dictate how long the green, yellow, or red light will remain active.

Constraints:

1. **Vehicle count** ranges from 0 to 15, meaning up to 15 cars can be detected.

2. **Timer** increments every clock cycle and must be able to handle long durations for traffic phases.

3. **Emergency signal** must override all other signals and immediately give priority to the green light.

4. The system must be efficient and able to handle real-time updates, with minimal latency for vehicle detection and signal switching.

Hints:

- Use a **finite state machine (FSM)** to control the transitions between the green, yellow, and red light phases.
- The **timer** can be used to implement a basic time-based switch between signals, while the **vehicle count** determines the duration of the green light.
- The system must handle **emergency signals** by overriding the normal sequence and giving priority to the green signal for emergency vehicles.
- Consider **multiplexing** between multiple intersections and handling multiple vehicle counts in parallel.

Tutorial Content:

This problem involves designing a **dynamic traffic signal controller** with an adaptive algorithm to adjust the signal timings based on the real-time vehicle count at the intersection. The solution requires an understanding of FSMs, multiplexing, and managing multiple real-time signals and sensors.

Concepts Used:

- **Finite State Machines (FSM)**: Used for managing the transitions between the light phases.
- **Timers**: To manage how long each light phase remains active.
- **Vehicle Counting Logic**: Used to determine how long the green light should stay active.
- **Emergency Handling**: Ensures that emergency vehicles are always given priority.

Solution Guide:

1. **FSM Design**: Design a finite state machine that handles the transitions between the green, yellow, and red lights.

2. **Timer Mechanism**: Implement a timer that controls how long each phase lasts. The green light will be longer if the vehicle count is high.

3. **Emergency Signal Override**: If the **emergency_signal** is active, the system should immediately switch to a green light for that road.

4. **Adaptive Timing**: If a road has a high vehicle count, the green light should stay longer; otherwise, it should transition faster to red.

Test Cases

Test Cases to Display

json
CopyEdit

```
[
  { "input": "vehicle_count: 12, timer: 0001, emergency_signal: 0, intersection_request: 1", "output": "green_light: 1, yellow_light: 0, red_light: 0, extended_green_signal: 1, phase_timer: 300" },
  { "input": "vehicle_count: 3, timer: 0010, emergency_signal: 0, intersection_request: 0", "output": "green_light: 1, yellow_light: 0, red_light: 0, extended_green_signal: 0, phase_timer: 60" },
  { "input": "vehicle_count: 0, timer: 0100, emergency_signal: 1, intersection_request: 0", "output": "green_light: 1, yellow_light: 0, red_light: 0, extended_green_signal: 0, phase_timer: 120" }
]
```

Hidden Test Cases

json
CopyEdit

```
[
  { "input": "vehicle_count: 10, timer: 0101, emergency_signal: 0, intersection_request: 1", "output": "green_light: 1, yellow_light: 0, red_light: 0, extended_green_signal: 1, phase_timer: 240" },
  { "input": "vehicle_count: 0, timer: 0110, emergency_signal: 0, intersection_request: 1", "output": "green_light: 1, yellow_light: 0, red_light: 0, extended_green_signal: 0, phase_timer: 60" }
```

]

Verilog Solution

verilog
CopyEdit

```
module traffic_signal_controller(
    input [3:0] vehicle_count,
    input [15:0] timer,
    input emergency_signal,
    input intersection_request,
    output reg green_light,
    output reg yellow_light,
    output reg red_light,
    output reg extended_green_signal,
    output reg [15:0] phase_timer
);

reg [1:0] current_state, next_state;
reg [3:0] count;

always @(posedge timer) begin
    if (emergency_signal) begin
        green_light <= 1;
        yellow_light <= 0;
        red_light <= 0;
        extended_green_signal <= 1;
        phase_timer <= 60;  // Emergency override duration
    end else if (intersection_request) begin
        if (vehicle_count > 8) begin
            green_light <= 1;
            extended_green_signal <= 1;
            phase_timer <= 240;  // Longer green for heavy traffic
        end else begin
            green_light <= 1;
            extended_green_signal <= 0;
            phase_timer <= 120;
        end
```

```
    end else begin
        green_light <= 0;
        yellow_light <= 1;
        red_light <= 0;
        extended_green_signal <= 0;
        phase_timer <= 30;  // Transition to yellow
    end
end

endmodule
```

VHDL Solution

vhdl
CopyEdit

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity traffic_signal_controller is
    Port ( vehicle_count : in  STD_LOGIC_VECTOR (3 downto 0);
           timer : in  STD_LOGIC_VECTOR (15 downto 0);
           emergency_signal : in  STD_LOGIC;
           intersection_request : in  STD_LOGIC;
           green_light : out  STD_LOGIC;
           yellow_light : out  STD_LOGIC;
           red_light : out  STD_LOGIC;
           extended_green_signal : out  STD_LOGIC;
           phase_timer : out  STD_LOGIC_VECTOR (15 downto 0));
end traffic_signal_controller;

architecture Behavioral of traffic_signal_controller is
begin
process(timer, emergency_signal, intersection_request, vehicle_count)
begin
    if emergency_signal = '1' then
        green_light <= '1';
```