Problem 1 : Priority Encoder Module

The following Verilog module implements a 4-to-2 **priority encoder** with an additional `valid` output signal. It takes a 4-bit input vector `in[3:0]` and produces a 2-bit output `out[1:0]` representing the highest-priority asserted input (with bit 3 having the highest priority and bit 0 the lowest).

- The `always @(*)` block ensures combinational logic.
- The `casez` statement allows for "don't care" conditions (using 'z') to simplify pattern matching.
- If one or more input bits are high, the encoder outputs the binary index of the highest-priority high bit and sets `valid = 1`.
- If no input bits are high, `valid` is set to 0 and `out` is set to 0.

Here's the Verilog code:

```
module priorityEncoder(input [3:0] in, output reg [1:0] out, output
     reg valid);
  always @ (*) begin
    casez (in)
      4'b1zzz : begin valid = 1'b1; out = 2'd3; end
      4'b01zz : begin valid = 1'b1; out = 2'd2; end
      4'b001z : begin valid = 1'b1; out = 2'd1; end
      4'b0001 : begin valid = 1'b1; out = 2'd0; end
      default : begin
        valid = 1'b0;
        out = 2'd0;
      end
    endcase
  end
endmodule
```

Problem 2 : 4 bit Mod 16 synchronous Up Counter

The `synchronousUpCounter` module implements a 4-bit synchronous up counter with asynchronous reset and enable control. Two versions of the counter are shown:

1. Implementation using T Flip-Flop K-Map Logic

- This version mimics a T flip-flop–based design derived from Karnaugh map simplifications.
- On every rising edge of the clock:
  - If `reset` is high, the counter resets to 0.

– If `enable` is high, the next state of each bit is determined by toggling behavior:

$$\texttt{count[0]} \leftarrow \sim \texttt{count[0]}$$
$$\texttt{count[1]} \leftarrow \texttt{count[1]} \oplus \texttt{count[0]}$$
$$\texttt{count[2]} \leftarrow \texttt{count[2]} \oplus (\texttt{count[0]} \& \texttt{count[1]})$$
$$\texttt{count[3]} \leftarrow \texttt{count[3]} \oplus (\texttt{count[0]} \& \texttt{count[1]} \& \texttt{count[2]})$$

- This design mirrors the behavior of T flip-flops where toggling occurs when input T = 1.

Present State				Transition				Next State			
Q_3	Q_2	Q_1	Q_0	T_3	T_2	T_1	T_0	Q_3'	Q_2'	Q_1'	Q_0'
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	1	0	0	1	1	0	0	1	0
0	0	1	0	0	0	0	1	0	0	1	1
0	0	1	1	0	1	1	1	0	1	0	0
0	1	0	0	0	0	0	1	0	1	0	1
0	1	0	1	0	0	1	1	0	1	1	0
0	1	1	0	0	0	0	1	0	1	1	1
0	1	1	1	1	1	1	1	1	0	0	0
1	0	0	0	0	0	0	1	1	0	0	1
1	0	0	1	0	0	1	1	1	0	1	0
1	0	1	0	0	0	0	1	1	0	1	1
1	0	1	1	0	1	1	1	1	1	0	0
1	1	0	0	0	0	0	1	1	1	0	1
1	1	0	1	0	0	1	1	1	1	1	0
1	1	1	0	0	0	0	1	1	1	1	1
1	1	1	1	1	1	1	1	0	0	0	0

Karnaugh Map for T_3

Q_3Q_2 \ Q_1Q_0	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	0	0	1	0
10	0	0	0	0

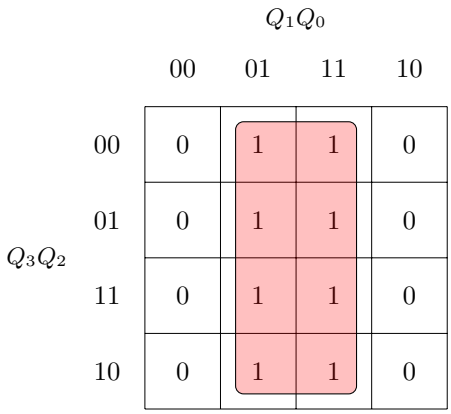
$$T_3 = Q_2Q_1Q_0$$

Karnaugh Map for T_2

Q_3Q_2 \ Q_1Q_0	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	0	0	1	0
10	0	0	1	0

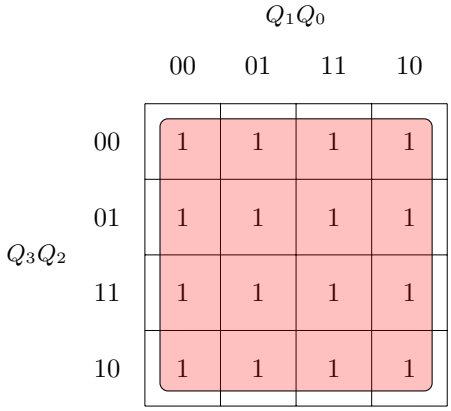
$$T_2 = Q_1Q_0$$

Karnaugh Map for T_1



$$T_1 = Q_0$$

Karnaugh Map for T_0



$$T_0 = 1$$

Code snippet:

```
module synchronousUpCounter(
  input clk,
  input reset,
  input enable,
  output reg [3:0] count
);
  always @ (posedge clk or posedge reset) begin
    if (reset == 1)
      out <= 4'd0;
    else if (enable == 1) begin
      count[0] <= ~count[0];
      count[1] <= count[1] ^ count[0];
```

```
      count[2] <= count[2] ^ (out[0] & count[1]);
      count[3] <= count[3] ^ (count[0] & count[1] & count[2]);
    end
  end
endmodule
```

2. Implementation using Adder Logic

- This alternative version uses a standard binary incrementer (adder logic).
- On the rising edge of the clock:
  - If `reset` is high, the counter resets to 0.
  - If `enable` is high, the counter is incremented by 1.
- This approach is more compact and commonly used in synthesizable designs.

Code snippet:

```
/* implemented using adder logic */
module synchronousUpCounter (
  input clk,
  input reset,
  input enable,
  output reg [3:0] out
);
always @ (posedge clk or posedge reset) begin
  if (reset == 1) out <= 4'd0;
  else if (enable == 1) begin
    out <= out + 1'b1;
  end
end
endmodule
```

Problem 3: Even Parity Generator

The following Verilog module computes the **parity bit** of an 8-bit input vector.

- The module takes an 8-bit input vector `data[7:0]`.
- It produces a single-bit output `parity`, which is the result of a bitwise XOR (exclusive OR) operation on all bits of `data`.
- The reduction operator `^` (XOR reduction) is used to compute the parity efficiently:
$$\texttt{parity} = data[0] \oplus data[1] \oplus \cdots \oplus data[7]$$
- If the number of 1s in `data` is odd, the output is 1 (odd parity). If even, the output is 0.

Here is the Verilog code:

```
module parity(input [7:0] data, output parity);
  assign parity = ^ data;
endmodule
```