RO PUF code:

```
module ro_with_en (
    input clk,
    input en,
    output reg ro_out
);
    always @(posedge clk) begin
        if (en)
            ro_out <= ~ro_out;
        else
            ro_out <= 0;
    end
endmodule

//------------------------------
// Module 2: Gated Counter
//------------------------------
module gated_counter (
    input clk,
    input rst,
    input en,
    output reg [15:0] count
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            count <= 0;
        else if (en)
            count <= count + 1;
    end
endmodule

//------------------------------
```

```
// Module 3: Synchronizer
//------------------------------
module sync_ff (
    input clk,
    input async_in,
    output reg sync_out
);
    reg tmp;
    always @(posedge clk) begin
        tmp <= async_in;
        sync_out <= tmp;
    end
endmodule

//------------------------------
// Module 4: FSM Controller (Verilog-Compatible)
//------------------------------
module fsm_controller (
    input clk,
    input rst,
    output reg enable_ro,
    output reg sample_now
);
    reg [15:0] timer;
    reg [1:0] state, next;

    parameter IDLE   = 2'b00;
    parameter ENABLE = 2'b01;
    parameter SAMPLE = 2'b10;
    parameter DONE   = 2'b11;
    parameter MEASURE_CYCLES = 10000;
```

```
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            state <= IDLE;
            timer <= 0;
        end else begin
            state <= next;
            if (state == ENABLE)
                timer <= timer + 1;
            else
                timer <= 0;
        end
    end

    always @(*) begin
        enable_ro = 0;
        sample_now = 0;
        case (state)
            IDLE:   next = ENABLE;
            ENABLE: next = (timer >= MEASURE_CYCLES) ? SAMPLE : ENABLE;
            SAMPLE: next = DONE;
            DONE:   next = IDLE;
            default: next = IDLE;
        endcase

        if (state == ENABLE)
            enable_ro = 1;
        if (state == SAMPLE)
            sample_now = 1;
    end
endmodule

//-----------------------------
```

```
// Module 5: Hamming(7,4) Encoder
//------------------------------
module hamming74_encoder (
    input  [3:0] data_in,
    output [6:0] code_out
);
    assign code_out[2] = data_in[0];
    assign code_out[4] = data_in[1];
    assign code_out[5] = data_in[2];
    assign code_out[6] = data_in[3];

    assign code_out[0] = data_in[0] ^ data_in[1] ^ data_in[3];
    assign code_out[1] = data_in[0] ^ data_in[2] ^ data_in[3];
    assign code_out[3] = data_in[1] ^ data_in[2] ^ data_in[3];
endmodule

//------------------------------
// Module 6: Hamming(7,4) Decoder
//------------------------------
module hamming74_decoder (
    input  [6:0] code_in,
    output [3:0] data_out,
    output reg  single_bit_error
);
    wire p1, p2, p4;
    wire [2:0] syndrome;

    assign data_out[0] = code_in[2];
    assign data_out[1] = code_in[4];
    assign data_out[2] = code_in[5];
    assign data_out[3] = code_in[6];
```

```
    assign p1 = code_in[0] ^ code_in[2] ^ code_in[4] ^ code_in[6];
    assign p2 = code_in[1] ^ code_in[2] ^ code_in[5] ^ code_in[6];
    assign p4 = code_in[3] ^ code_in[4] ^ code_in[5] ^ code_in[6];
    assign syndrome = {p4, p2, p1};

    always @(*) begin
        single_bit_error = (syndrome != 3'b000);
    end
endmodule

//------------------------------
// Module 7: Top Module
//------------------------------
module top (
    input clk,
    input rst,
    output [3:0] led,
    output error_led
);
    wire enable_ro, sample_now;
    wire [3:0] ro1_out, ro2_out;
    wire [3:0] sync_ro1_out, sync_ro2_out;
    wire [15:0] count1 [3:0];
    wire [15:0] count2 [3:0];
    reg  [3:0] puf_bits;
    wire [6:0] encoded_puf;
    wire [3:0] decoded_puf;
    wire       error_flag;

    genvar i;
    generate
        for (i = 0; i < 4; i = i + 1) begin : ro_puf_gen
```

```
        ro_with_en ro1 (.clk(clk), .en(enable_ro), .ro_out(ro1_out[i]));
        ro_with_en ro2 (.clk(clk), .en(enable_ro), .ro_out(ro2_out[i]));

        sync_ff sync1 (.clk(clk), .async_in(ro1_out[i]), .sync_out(sync_ro1_out[i]));
        sync_ff sync2 (.clk(clk), .async_in(ro2_out[i]), .sync_out(sync_ro2_out[i]));

        gated_counter cnt1 (.clk(sync_ro1_out[i]), .rst(rst), .en(enable_ro), .count(count1[i]));
        gated_counter cnt2 (.clk(sync_ro2_out[i]), .rst(rst), .en(enable_ro), .count(count2[i]));
    end
endgenerate

fsm_controller fsm (
    .clk(clk),
    .rst(rst),
    .enable_ro(enable_ro),
    .sample_now(sample_now)
);

integer j;
always @(posedge clk or posedge rst) begin
    if (rst) begin
        puf_bits <= 0;
    end else if (sample_now) begin
        for (j = 0; j < 4; j = j + 1) begin
            puf_bits[j] <= (count1[j] < count2[j]) ? 1'b1 : 1'b0;
        end
    end
end

hamming74_encoder h_enc (
    .data_in(puf_bits),
    .code_out(encoded_puf)
```

```
    );

    hamming74_decoder h_dec (
        .code_in(encoded_puf),
        .data_out(decoded_puf),
        .single_bit_error(error_flag)
    );

    assign led = decoded_puf;
    assign error_led = error_flag;

    // ILA Debug Instance
    ila_0 ila_inst (
        .clk(clk),
        .probe0(puf_bits),
        .probe1(encoded_puf),
        .probe2(decoded_puf),
        .probe3(error_flag),
        .probe4(enable_ro),
        .probe5(sample_now)
    );
endmodule
```