ChatGPT

Lightweight NLFSR–PUF–MAC Encryption Scheme Design Guide

This guide details an RTL design strategy and implementation plan for the **NLFSR-based stream cipher + Arbiter PUF + Neeva hash-based MAC** scheme on a Xilinx Artix-7 FPGA (e.g. in Vivado). The target system provides confidentiality, data integrity, and device integrity for IoMT devices by combining a stream cipher (NLFSRs), a Physical Unclonable Function (PUF), and a lightweight hash-based MAC. As described in [12], this scheme "presents a lightweight stream cipher to ensure privacy, data integrity and device integrity" using a 128-bit NLFSR stream cipher and a PUF-based MAC [1]. Below we cover the RTL architecture for each component, Verilog design notes, module interfaces, clock/reset strategy, Vivado flow, and verification.

1. System Overview and Data Flow

The encryption system has three core modules:

- **NLFSR Stream Cipher:** Two 128-bit nonlinear feedback shift registers (NLFSR1 and NLFSR2) generate a keystream bit per cycle using both linear and nonlinear feedback functions [2]. A nonlinear Boolean output function (`Z`) combines selected register bits to produce each keystream bit [3].

- **Arbiter PUF:** A delay-based PUF takes a 128-bit *challenge* (configuring two parallel signal paths) and outputs one *response* bit, indicating which path was faster [4] [5]. Repeating this for 128 independent challenges yields a 128-bit PUF response vector `{r_i}` used in the MAC.

- **Neeva Hash-based MAC Generator:** A sponge-like hash (Neeva) processes the message and PUF response to compute a 128-bit MAC [6]. In this scheme the plaintext message `m_i` (e.g. 192 bits) is concatenated with the 128-bit PUF output `{r_i}` and split into 32-bit blocks. The hash applies 32 rounds of (Present S-box, Feistel XOR, 8-bit rotation, and constant addition) per block [7].

Data Flow: On each packet, the NLFSR keystream `k_i` is XORed with the plaintext `m_i` to produce ciphertext `c_i = m_i ⊕ k_i`. Simultaneously, the MAC module concatenates `m_i` and the PUF bits `r_i`, then runs the Neeva hash to generate a MAC (as described in [11]). The MAC authenticator protects both message and device (PUF) integrity [6].

Below sections cover RTL design and implementation details for each block, how to interface them, and overall integration.

2. NLFSR Stream Cipher Design

2.1 Architecture and Feedback Functions

The cipher uses **two 128-bit NLFSRs** (NLFSR1 and NLFSR2). Each register holds 128 state bits. The feedback for NLFSR2 is defined as a combination of a **linear feedback polynomial** and a **nonlinear feedback polynomial** [2]. Specifically, Eq.(1) and Eq.(2) in the reference define:

> *f<sub>L</sub>(x) = 1 + x^14 + x^48 + x^112 + x^120 + x^124* (linear taps)
> *f<sub>N</sub>(x) = 1 + x^31 x^68 + x^39 x^84 + x^62 x^111 + x^65 x^114 + x^67 x^116 + x^76 x^123 + x^91 x^124 x^135 x^138* (nonlinear terms)

The feedback polynomial for NLFSR2 is then *f<sub>255</sub>(x) = 1 ⊕ f<sub>L</sub>(x) ⊕ f<sub>N</sub>(x)* [2]. The feedback for NLFSR1 is defined by a shifted version of NLFSR2's nonlinearity (Eq.4–5 in [9]) and a fixed tap at x^129 [8]. The output keystream bit is computed by a Boolean combining function *Z* on selected taps from both registers:

> *Z(x) = x<sub>70</sub> ⊕ x<sub>64</sub> ⊕ (x<sub>63</sub>∧x<sub>45</sub>) ⊕ (x<sub>63</sub>∧x<sub>64</sub>) ⊕ (x<sub>45</sub>∧x<sub>70</sub>) ⊕ (x<sub>70</sub>∧x<sub>63</sub>∧x<sub>64</sub>) ⊕ (x<sub>36</sub>∧x<sub>45</sub>∧x<sub>64</sub>) ⊕ (x<sub>70</sub>∧x<sub>45</sub>∧x<sub>64</sub>) ⊕ (x<sub>70</sub>∧x<sub>63</sub>∧x<sub>64</sub>∧x<sub>192</sub>)* [3].

Here x_36, x_45, x_63, x_64, x_70 are taps from NLFSR1 and x_192 is a tap from NLFSR2 (offset by 128). This nonlinear output provides mixing between the two registers.

2.2 RTL Implementation Notes

- **Registers and Feedback:** In Verilog, implement NLFSR1 and NLFSR2 as two 128-bit registers (e.g. `reg [127:0] reg1, reg2;`). On each clock, compute the new feedback bits by XOR/AND operations according to the polynomials. For example, NLFSR2's new-in bit could be:

  ```
  wire fb_L = reg2[14] ^ reg2[48] ^ reg2[112] ^ reg2[120] ^ reg2[124]; //
  from f_L
  wire fb_N = (reg2[31] & reg2[68]) ^ (reg2[39] & reg2[84]) ^ (reg2[62] &
  reg2[111]) ^
              (reg2[65] & reg2[114]) ^ (reg2[67] & reg2[116]) ^ (reg2[76] &
  reg2[123]) ^
              (reg2[91] & reg2[124] & reg1[?] & reg1[?]); // continue as per
  f_N terms
  wire new2 = fb_L ^ fb_N ^ 1'b1; // include the constant 1
  ```

  NLFSR1's new bit uses its own shifted terms (based on Eq.4–5). Use bitwise XOR (`^`) and AND (`&`) for logic. Insert the computed bit at the LSB (or MSB, depending on shift direction) and shift the register.

- **Synchronization:** Use a single clock for both NLFSRs. In a synchronous design, calculate the new feedback before the register update so that on the rising edge, both registers shift and update simultaneously. Use a synchronous reset to initialize the registers to a secret key or seed.

- **Verilog Example:** An outline for NLFSR update (assuming simple left-shift) is:

```
reg [127:0] reg1, reg2;
wire newbit1, newbit2;
// Compute newbit2 from reg2 using f_L and f_N (example taps)
assign newbit2 = reg2[127] ^ reg2[113] ^ reg2[79] ^ reg2[15] ^ reg2[7] ^
(reg2[96] & reg2[35]); // etc.
// Compute newbit1 based on shifted non-linear terms of NLFSR2 (per paper)
assign newbit1 = reg1[127] ^ ...;
always @(posedge clk) begin
  if (reset) begin
    reg1 <= INIT1;  // load initial 128-bit key state
    reg2 <= INIT2;
  end else begin
    reg1 <= {reg1[126:0], newbit1};
    reg2 <= {reg2[126:0], newbit2};
  end
end
// Keystream output from combining function Z:
wire keystream_bit = reg1[70] ^ reg1[64] ^ (reg1[63]&reg1[45]) ^ ... ^
(reg1[70]&reg1[63]&reg1[64]&reg2[64]);
```

- **Non-linear Terms:** Implement all AND terms in the feedback explicitly. Ensure that each multi-input AND (e.g. triple product) is correctly computed. For complex products (like 3-input AND), you may cascade two-input ANDs or use a single expression. Synthesis will map these to LUTs.

- **Output Function Z:** Realize the Z-function with bitwise logic (XOR/AND). For example:

```
wire z_out = reg1[70] ^ reg1[64] ^ (reg1[63] & reg1[45]) ^ (reg1[63] &
reg1[64]) ^
             (reg1[45] & reg1[70]) ^ (reg1[70] & reg1[63] & reg1[64]) ^
             (reg1[36] & reg1[45] & reg1[64]) ^ (reg1[70] & reg1[45] &
reg1[64]) ^
             (reg1[70] & reg1[63] & reg1[64] & reg2[64]);
```

- **Performance:** 128-bit registers with this logic fits comfortably in Artix-7 LUTs. Ensure reset behavior is defined. You may pipeline the calculation if needed (e.g. register intermediate AND terms) but for basic design this is not required.

2.3 Key and Output Interface

- **Inputs:** 128-bit initial state (key) for each NLFSR (could be loaded via registers or one-time configuration). A clock and reset.
- **Output:** A 1-bit keystream per cycle. Typically you will XOR this bit with plaintext (handled outside the NLFSR module).
- **Module Example:**

```
module nlfsr_stream(
  input clk, reset,
  input [127:0] init_state1, init_state2,
  output reg keystream_bit
);
  // internal regs reg1, reg2 and feedback as above...
endmodule
```

By following the above strategy, the two NLFSRs produce a pseudorandom keystream. The design (Fig.5 in [9]) shows that "the cipher's state" is the 256-bit contents of both registers, and a Boolean function Z taps bits from both to generate each keystream bit [3].

3. Arbiter PUF Design

3.1 Arbiter PUF Architecture

An **Arbiter PUF** in FPGA is typically built as two parallel delay chains, each comprising a sequence of 2-to-1 multiplexers (switch stages). A 128-bit *challenge* selects how the paths are crossed at each stage [4]. The basic structure is:

- **Two Identical Paths:** Each path starts from a common input and traverses 128 switch stages. At each stage *i*, the challenge bit `C[i]` controls a MUX: if `C[i]=0`, each path signal goes straight; if `C[i]=1`, the paths swap. This effectively randomizes the delays.

- **Arbiter (Latch):** At the end of the chains, an arbiter latch compares which path's signal arrived first. The arbiter output is a single bit (0 or 1) indicating the faster path [5].

- **Challenge and Response:** A 128-bit challenge configures one round of PUF and yields one response bit. To produce 128 response bits (`r_0`…`r_127`), the design repeatedly applies 128 different challenges (or uses 128 parallel PUF instances). In the proposed scheme, a series of 128 challenge-response evaluations are performed and collected.

[10†L981-L989] notes: "A 128-bit Arbiter PUF contains two parallel 128 multiplexers connected serially… The D-flip flop generates a 128-bit input challenge C0…C127. The select line of the multiplexer is used as the input challenge… two paths are determined by applying the 128-bit challenge… The arbiter at the end… encodes the result as a single bit output response" [4] [5].

3.2 RTL Implementation Notes

- **Digital Model:** Because physical delay differences are analog, in RTL we mimic the PUF by logical means. One common digital model is to simulate the switch chain and then use an XOR (or simple comparator) as the arbiter output. For example, propagate a logical '1' through both chains and XOR the final outputs:

```
module arbiter_puf (
  input clk,
  input [127:0] challenge,
  output reg response
);
  reg [127:0] path1, path2;
  integer i;
  always @(posedge clk) begin
    // Initialize both paths with 1 (pulse)
    path1[0] <= 1'b1;
    path2[0] <= 1'b1;
    // Propagate through stages
    for (i = 0; i < 128; i = i+1) begin
      if (challenge[i]) begin
        // swap paths
        path1[i+1] <= path2[i];
        path2[i+1] <= path1[i];
      end else begin
        // no swap
        path1[i+1] <= path1[i];
        path2[i+1] <= path2[i];
      end
    end
    // Arbiter: output = XOR of final bits (simulated comparison)
    response <= path1[128] ^ path2[128];
  end
endmodule
```

This block uses registers (`path1`, `path2`) so that each stage is clocked, avoiding a long combinational chain. The final `response` is 1 if path1 ≠ path2 (one arrived earlier). Note: This is a simplification; in hardware one might use a faster latch to capture which path is shorter, but using XOR models a random but stable output.

- **Timing and Variability:** In an actual FPGA, the path delays depend on routing. To implement a "true" PUF, one would leave these paths unbalanced in the layout so that delay differences exist. In pure RTL simulation, such physical variation is not present. The digital model above effectively treats the PUF as a deterministic function of the challenge (not truly random). For testing, you can inject randomness or fixed patterns.

- **Registering the Arbiter:** To maintain synchronous design, we register the final response. The arbiter output may be captured on the next clock edge as shown (`response <= ...`).

- **Reset Behavior:** Initialize `path1[0]` and `path2[0]` on each evaluation. If you need repeated responses, you can reuse the module by feeding new `challenge` values each cycle or assertion of an enable signal.

- **Verilog Interface:** The PUF module can be:

  ```
  module arbiter_puf(
    input clk,
    input [127:0] challenge,
    output reg response
  );
    // (implementation as above)
  endmodule
  ```

  It produces one response bit per clock given a 128-bit `challenge` input.

3.3 PUF Output Usage

In the full scheme, 128 separate PUF response bits are collected by applying 128 different challenges. Each response `r_i` is then used in the MAC generator. In hardware, you might drive a challenge counter or load challenges from memory. The design might hold each `r_i` in a shift register to form the 128-bit PUF output vector.

Overall, the PUF adds “device integrity”: since the response depends on unclonable delay mismatches, the MAC effectively binds the message to the physical device.

4. Neeva Hash-based MAC Generator

4.1 Neeva Hash Overview

The Neeva hash is a lightweight sponge-based hash function [7]. In this scheme it serves as a MAC: the input is the concatenation of the message and PUF response. Key properties:

- **Input Processing:** The 192-bit plaintext message `m_i` is concatenated with the 128-bit PUF response bits `{r_i}`. However, according to [11], the MAC is computed by concatenating `r_i || m_i` (i.e. placing PUF bits in front of the message) into a 320-bit block [9]. For hashing, this block is conceptually padded and split into 32-bit words: the 320 bits become 10 blocks of 32 bits (including padding), or as the reference says, they split the 192-bit message into six 32-bit words after XORing the PUF bit as additional input [10] [9].

- **State Size:** Neeva uses a 256-bit internal state (capacity+rate = 256 bits). Initially the state is all zeros.

- **Round Function:** For each 32-bit message block, the state is updated by 32 rounds of a nonlinear permutation. Each round consists of:

- **S-box Layer:** Apply 16 parallel 4×4 Present S-boxes to the 256-bit state (treat as 16 16-bit words) [11].
- **Unbalanced Feistel (XOR):** Mix words by XORing groups: first, second, third 16-bit words are each XORed with the fourth word (fourth word unchanged) to diffuse bits [11].
- **Rotate and Add:** Perform an 8-bit left rotation of the entire 256-bit state, then add a round constant (16-bit addition mod 2^16) [11].
- Repeat these steps for 32 rounds [12].

These operations provide confusion (S-box) and diffusion (XOR+rotate) as described in the Neeva paper [11]. After all rounds, the most significant 32 bits of the state become part of the output (and this process may repeat for squeezing if more output is needed).

4.2 RTL Implementation Notes

- **S-box Implementation:** The 4×4 Present S-box can be implemented with a 16-entry LUT (per 4-bit nibble). Since the state is 256 bits, you have 16 parallel nibbles. In Verilog, you can use a `case` statement or explicit Boolean expressions. For speed, consider using the FPGA's distributed RAM or LUTs. For example:

```
function [3:0] present_sbox(input [3:0] in);
  case (in)
    4'h0: present_sbox = 4'hC; 4'h1: present_sbox = 4'h5; // ... fill in S-box truth table
    // ... all 16 entries ...
    default: present_sbox = 4'h0;
  endcase
endfunction
```

  Then apply it to each nibble of the state in parallel (16 instances).

- **Unbalanced Feistel XOR:** After the S-box layer, group the state into four 64-bit words. Let `W0,W1,W2,W3` be these words (where each Wi consists of four 16-bit subwords in sequence). The description says: W0, W1, W2 each XOR with W3. In RTL, that means `W0_new = W0 ^ W3; W1_new = W1 ^ W3; W2_new = W2 ^ W3; W3_new = W3;`. Then reassemble the 256-bit state from these new words.

- **Rotation:** Perform a bit-wise left rotation of the 256-bit state by 8 bits. In Verilog, if `state` is a reg [255:0], `state_rot = {state[247:0], state[255:248]};`.

- **Round Constants:** Maintain an array of 32 constants (each maybe 16 bits). After the rotation, add the j-th constant to (say) the least significant 16 bits of the state (mod 2^16). For example:

```
state[15:0] = state[15:0] + round_const[j];
```

- **Iterative Loop:** You can implement the 32 rounds in a `for` loop inside an `always` block (unrolled or pipelined). For hardware reuse, it is common to use a single round circuit and iterate 32 cycles with a round counter. However, if resources allow, fully unrolling into combinational pipeline yields faster throughput. For a design guide, suggest the iterative approach:

```
reg [255:0] state;
integer round;
always @(posedge clk) begin
  if (reset) begin
    state <= 256'b0;
    round <= 0;
  end else if (next_round) begin
    // Apply one round of Neeva permutation
    // 1) S-box on each nibble:
    for (i=0; i<64; i=i+1) begin
      state[4*i +: 4] <= present_sbox(state[4*i +: 4]);
    end
    // 2) Feistel XOR on 64-bit words:
    // Assuming words W0=state[255:192], W1=[191:128], W2=[127:64],
W3=[63:0]
    state[255:192] <= state[255:192] ^ state[63:0];
    state[191:128] <= state[191:128] ^ state[63:0];
    state[127:64]  <= state[127:64]  ^ state[63:0];
    // 3) Rotate left 8 bits:
    state <= {state[247:0], state[255:248]};
    // 4) Add constant:
    state[15:0] <= state[15:0] + round_const[round];
    round <= round + 1;
  end
end
```

This is illustrative; optimizations (registering intermediate steps) can be applied for timing.

- **Hash Module Interface:** The MAC module takes as input the message blocks and PUF bits. For example:

```
module neeva_mac(
  input clk, reset,
  input [191:0] message,    // or stream of 32-bit blocks
  input [127:0] puf_bits,
  output [127:0] mac
);
```

```
  // (include internal state regs, S-box function, etc.)
endmodule
```

Internally, concatenate `puf_bits` with `message` (e.g. `{puf_bits, message}`) then split into 32-bit words as needed for absorption. The above loop assumes handling one 32-bit word per invocation; you may need to loop 6 blocks * 32 rounds each.

- **Padding:** If messages vary in length, implement padding as per [13†L221-L229] (append `1`, then zeros, then `1` until length ≡ 30 mod 32). In many IoMT uses, message size is fixed (e.g. sensor readings), so padding may be static. The reference indicates a padding rule for multiple-of-32-bit [13]; include this if input length is variable.

- **Final MAC Output:** After absorbing all blocks (and optionally squeezing the state if needed), collect the most significant 128 bits of the state as the MAC output (the paper suggests a 128-bit MAC). Register this result for output.

4.3 MAC Processing in Scheme

As per [11], for each message `m_i` (192 bits), the design does: `M = r_i ‖ m_i` where `r_i` is the 128-bit PUF response for that iteration [9]. Then it computes `mac_i = H_Neeva(M)`. Concretely, one could feed the 128 PUF bits and 192 message bits into the hash module as parallel inputs or sequentially. A simple approach:

1. Clock in `puf_bits` and `message` to the hash module at start.
2. Let the hash module perform all 32×6 = 192 rounds (absorb 6 blocks) over multiple cycles.
3. Capture the final 128-bit MAC when done.

Citations [4†L201-L210] and [11†L1062-L1066] confirm this processing: data and response are concatenated and hashed to produce `mac_i` [6].

5. Module Interfaces and Top-Level Integration

- **Top-Level I/O:** Decide how the design interfaces with the outside. For a self-contained FPGA block, define inputs for the plaintext (192 bits), initial NLFSR states (if not hardcoded), PUF challenge generation control, and a clock/reset. Outputs would be the ciphertext (192 bits) and MAC (128 bits).

- **Module Connections:** In the top module, instantiate:

- `nlfsr_stream` for keystream generation.

- `arbiter_puf` (or multiple instances/times) for PUF challenge-response.
- `neeva_mac` for hashing.

Connect as follows: - The plaintext `m_i` goes to the NLFSR stream cipher and to the MAC module. - The NLFSR outputs `keystream_bit` which is XORed with each plaintext bit to produce ciphertext. You can build a simple shift register of keystream bits or directly XOR the 192 bits in parallel if your NLFSR outputs bytes or words (or just bitwise XOR if output is bit-by-bit). - The MAC module takes the plaintext `m_i` and the full 128-bit PUF output (collected externally) as inputs [6]. - The PUF module(s) take a 128-bit challenge input. You may implement a simple counter or LFSR to generate successive challenges, or pre-load challenges. On each cycle (or when signaled), feed a new challenge into the PUF and capture the response bit. Accumulate 128 bits of response (e.g. in a shift register).

- **Clocking:** Use a single clock domain for simplicity. The keystream and PUF can run in parallel or sequentially, depending on timing. For example, you might:
- First, clock 128 cycles to obtain all PUF bits (while also starting the NLFSR).
- Then clock the Neeva hash for its required rounds.

- Finally, output the MAC and ciphertext. Alternatively, pipelined operation is possible.

- **Example Top Module Skeleton:**

```
module top_cipher (
  input clk, reset,
  input [191:0] plaintext,
  input [255:0] nlfsr_key, // e.g. two 128-bit seeds
  input [7:0] challenge_counter, // or larger bus to feed PUF
  output [191:0] ciphertext,
  output [127:0] mac
);
  wire ks_bit;
  nlfsr_stream cipher(.clk(clk), .reset(reset),
                      .init_state1(nlfsr_key[127:0]),
                      .init_state2(nlfsr_key[255:128]),
                      .keystream_bit(ks_bit));
  // XOR keystream to plaintext (bit-serial or parallel). Example:
  // Here assuming one bit per clock, you would need a shift register for
plaintext.
  // For simplicity, assume ciphertext=plaintext in parallel XOR.
  genvar i;
  generate for (i = 0; i < 192; i = i+1) begin
    assign ciphertext[i] = plaintext[i] ^ ks_bit; // in reality, align
widths
  end endgenerate

  // PUF: generate 128-bit response (example using one instance and
accumulating bits)
  wire puf_resp;
  arbiter_puf
puf(.clk(clk), .challenge(challenge_counter), .response(puf_resp));
  // Accumulate response bits into 128-bit vector (requires shift reg or
memory).

  neeva_mac macgen(.clk(clk), .reset(reset),
                  .message(plaintext),
                  .puf_bits(puf_output_vector),
                  .mac(mac));
endmodule
```

This is simplified; in practice you need control logic to sequence challenge indexing and hashing.

6. Clock and Reset Scheme

- **Global Clock:** Use a single global clock (e.g. 50 MHz or 100 MHz) for all modules. Define it in constraints (see Vivado section).

- **Synchronous Reset:** Use a synchronous or asynchronous reset (deasserted high or low) for all registers. On reset, initialize NLFSRs to the key, clear the Neeva state to zero, and reset PUF path registers. Ensure the arbiter latch is reset to a known output.

- **Timing Considerations:** The arbiter PUF chain is implemented with registers at every stage in our model, so there is no long combinatorial path beyond 2 LUT levels. The Neeva hash rounds involve S-boxes and XORs, which can be pipelined one round per cycle. If unrolled, ensure the design meets timing by pipelining between rounds or stages.

- **Handshaking (Optional):** If blocks operate sequentially, include handshake signals (start/done) to coordinate when to sample outputs and load inputs. For example, assert `ready` when keystream or MAC is valid.

7. Vivado Integration and Implementation

7.1 Project Flow

1. **Create Vivado Project:** Start a new RTL project for Artix-7 (specify correct device). Add Verilog (or VHDL) source files for each module (NLFSR cipher, PUF, MAC, top).

2. **Top-Level Module:** Set your top-level module (e.g., `top_cipher`). Ensure its ports match your desired external I/Os (e.g. `plaintext`, `ciphertext`, `mac`, clock, reset).

3. **Add Constraints (.xdc):**

4. Define the clock pin and frequency, e.g.

```
create_clock -name sys_clk -period 10.000 [get_ports clk]
```

5. Assign other I/O pins if connecting to external signals (e.g. if plaintext comes from an external interface or if using buttons for challenge, map those pins).

6. No special constraints are needed for internal logic. If using internal signals only, you may skip I/O pin constraints except clock.

7. **Synthesis and Implementation:** Run synthesis. Check for any combinatorial loops or issues (our design has loops for feedback, but they are broken by registers). If timing violations occur in the round function, pipeline as needed.

8. **Mapping to FPGA:** The design should map easily. The arbiter PUF in real hardware may have slight imbalance; to enforce it, you could use LOC constraints to place the two paths in proximity. However, true PUF behavior on FPGA is beyond this RTL scope.

9. **Generate Bitstream:** After place & route, build the bitstream. Use the default Synplify synthesis and Vivado implementation tools.

7.2 Constraints and Design Hints

- **Area and Timing:** This design is fairly small (couple of hundred LUTs). Ensure the clock period meets your target by pipelining the Neeva rounds if necessary.

- **Module Priorities:** In case of separate clock domains (not needed here), assign False Path or set clock groups in constraints, but we use one domain.

- **Locally Unused Output Warning:** The arbiter PUF model drives the output after a cycle. If unused, mark `response` as needed.

- **Physical Planning (Optional):** If exploring PUF behavior, you might floorplan the two paths for symmetry. Otherwise, leave it to the FPGA router.

- **Block Ram:** The Neeva hash uses small constants (32 of 16 bits). These can be stored in registers or a small ROM (LUT-based memory). No need for BRAM for this size.

8. Simulation and Verification

- **Functional Simulation:** Write testbenches for each module:
- *NLFSR:* Provide known initial states and simulate a few cycles. Compare output keystream with a software reference if available. Test that feedback matches the defined polynomials.
- *PUF:* Simulate with a few example challenges and check the output bit toggles in a plausible manner (e.g., alternate all-zeros vs all-ones challenge to see difference).

- *MAC:* Test the Neeva hash on simple inputs (e.g. all-zero message and PUF bits) and verify the output matches a software implementation of Neeva or known test vectors. Check that changing one input bit flips many output bits.

- **Combined Test:** Write a top-level testbench that:

- Applies a sequence of plaintext packets.

- Feeds dummy or pattern PUF challenges and collects responses.
- Runs the MAC and cipher logic.
- Checks that decrypting the ciphertext with the keystream recovers plaintext, and verifies the MAC equals an expected value.

- **Corner Cases:** Test reset behavior, e.g. assert reset for a few cycles and then release. Verify that reloading the same key and PUF challenges yields the same keystream and MAC.

- **Random Testing:** Use random inputs and check statistical properties (optionally run a NIST STS suite on the keystream as done in the paper). For MAC, verify collision resistance by changing messages or PUF bits and checking the MAC changes unpredictably.

- **Timing Simulation:** After synthesis, perform gate-level simulation to ensure the design meets timing (especially if pipelined) and that there are no glitches in registers (using SDF back-annotation if needed).

- **Hardware Testing (if possible):** If you have the Artix-7 board, load the bitstream and run test patterns via UART or host interface to confirm on-board operation. If not, thorough simulation is required.

By following this guide—implementing each module as described, integrating carefully, and verifying with simulation—you will realize the complete lightweight encryption scheme (NLFSR + Arbiter PUF + Neeva MAC) on FPGA. All components and their interactions are grounded in the referenced design [2] [4] [6] [7], ensuring the hardware faithfully reproduces the published scheme.

References: The design follows the architecture and algorithms from [12,9–11] and the Neeva hash specification [4], specifically citing equations for NLFSR feedback [2], PUF structure [4] [5], and the hash round operations [7]. These informed the RTL implementation strategy and functional verification plan.

---

[1] [2] [3] [4] [5] [6] [8] [9] [10] (PDF) Lightweight encryption with data and device integrity using NLFSR and PUF for the Internet of Medical Things
https://www.researchgate.net/publication/376876026_Lightweight_encryption_with_data_and_device_integrity_using_NLFSR_and_PUF_for_the_Internet_of_Medical_Things

[7] [11] [12] [13] eprint.iacr.org
https://eprint.iacr.org/2016/042.pdf