# LAB ASSIGNMENT 06
# GHOST

**Janani Palani - jp7510**
**Inchara Vittal Shetty - ivs2027**

## Introduction

This report presents the implementation and evaluation of the GHOST (Generative Hardware Trojan Simulation) framework using Verilog design examples. The goal of the lab is to demonstrate how language models can be leveraged to automatically generate, simulate, and detect hardware Trojans across various vulnerability classes within digital circuits.

The workflow begins with initializing the OpenAI environment and importing required libraries, followed by loading base Verilog designs and constructing Trojan-insertion prompt templates. The experiment proceeds through model inference, response parsing, and automated saving of generated vulnerable designs. Each generated design is paired with a tailored testbench for simulation and detection of malicious behavior using the Icarus Verilog toolchain.

Four categories of hardware Trojans are explored:

1.Functionality Change Trojan (T1) – Alters logical correctness under specific input conditions.

2. Information Leakage Trojan (T2) – Establishes covert data transmission via a hidden output channel.

3. Denial-of-Service Trojan (T3) – Disables module functionality upon rare triggering sequences.

4. Performance Degradation Trojan (T4) – Induces unnecessary toggling to increase power consumption.

For each category, the modified RTL code and its corresponding testbench are analyzed to verify the trigger mechanism, payload behavior, and post-trigger effects. The report also details troubleshooting steps for syntax repair, model output cleaning, trigger tuning, and simulation verification to ensure functional correctness and reproducibility.

Overall, the assignment illustrates the complete GHOST pipeline—from clean design loading to automated Trojan generation and simulation—highlighting both the methodology and defensive evaluation of machine-generated hardware Trojans in digital systems.

## Workflow

**Step 1:** Initialize OpenAI Environment and Import Required Libraries

```
import openai
from openai import OpenAI
import json
import os
openai_api_key='Your API key'
```

**Step 2:** Load base Verilog designs from directory

```
def load_base_designs(directory):
    base_designs = []
    for file in os.listdir(directory):
        if file.endswith(".v"):
            with open(os.path.join(directory, file), "r") as f:
                base_designs.append((file, f.read()))
    return base_designs

#Example usage
base_designs_directory = "/content/"  #Adjust path as needed
base_designs = load_base_designs(base_designs_directory)
```

**Step 3:** Construct LLM Trojan Insertion Prompt Template

**Step 4:** Initialize OpenAI client and define model inference function

```
#ChatGPT
from openai import OpenAI
client = OpenAI(api_key=openai_api_key)

def model_inference(prompt):
    model_name = "gpt-4.1" # choose: gpt-4 or gpt-4.1
    completion = client.chat.completions.create(
        model=model_name,
        messages=[
            {"role": "system", "content": "You are an expert skilled in hardware design and verification."},
            {"role": "user", "content": prompt}
        ]
    )
    return completion.choices[0].message.content, model_name
```

**Step 5:** Extract Verilog code and metadata sections from model response

```
def extract_code_and_metadata(response_text):
    sections = ["code:", "explanation:", "trigger:", "payload:", "taxonomy:"]
    results = {}

    response_lower = response_text.lower()
    for i, section in enumerate(sections):
        start = response_lower.find(section)
        if start == -1:
            print(f"Warning: '{section}' not found in the response.")
            results[section] = ""
            continue

        start += len(section)
        if i < len(sections) - 1:
            end = response_lower.find(sections[i+1], start)
            if end == -1:
                end = len(response_lower)
        else:
            end = len(response_lower)

        content = response_text[start:end].strip()

        # Clean up the content
        content = clean_content(content, section)

        results[section] = content

    return (results["code:"], results["explanation:"], results["trigger:"],
            results["payload:"], results["taxonomy:"])

def clean_content(content, section):
    # Remove triple backticks and language specifiers
    content = content.replace("```", "").strip()

    # Remove '##' markers
    content = content.replace("##", "").strip()

    # For code section, ensure it starts with `timescale, `include, or module
    if section == "code:":
        lines = content.split("\n")
        start_index = 0
        for i, line in enumerate(lines):
            if (line.strip().startswith("`timescale") or
                line.strip().startswith("`include") or
                line.strip().startswith("module")):
                start_index = i
                break
        content = "\n".join(lines[start_index:])

    # For non-code sections, take only the first paragraph to avoid repetition
    elif section != "taxonomy:":
        paragraphs = content.split("\n\n")
        content = paragraphs[0] if paragraphs else ""

    return content.strip()
```

**Step 6:** Save generated vulnerable designs and metadata to disk

```
import os

def create_directory(path):
    os.makedirs(path, exist_ok=True)

def save_vulnerable_design(design_name, verilog_code, base_directory, vulnerability_id, model_name, version_number):
    base_name = os.path.splitext(design_name)[0]
    directory = os.path.join(base_directory, model_name, base_name)
    create_directory(directory)
    filename = f"{base_name}_H{vulnerability_id}_{model_name}_A{version_number}.v"
    file_path = os.path.join(directory, filename)
    print(f"Saving vulnerable design to: {file_path}")

    # Remove any content after the last 'endmodule'
    last_endmodule_index = verilog_code.rfind('endmodule')
    if last_endmodule_index != -1:
        verilog_code = verilog_code[:last_endmodule_index] + 'endmodule'

    with open(file_path, "w") as f:
        f.write(verilog_code.strip())

def save_vulnerability_description(design_name, base_directory, vulnerability_id, explanation, trigger, payload, taxonomy, model_name, version_number):
    base_name = os.path.splitext(design_name)[0]
    directory = os.path.join(base_directory, model_name, base_name)
    create_directory(directory)
    description = f"""
Design: {design_name}
Vulnerability ID: {vulnerability_id}
Model: {model_name}
Attempts: {version_number}

Explanation:
{explanation}

Trigger:
{trigger}

Payload:
{payload}

Taxonomy:
{taxonomy}
"""
    filename = f"{base_name}_H{vulnerability_id}_{model_name}_A{version_number}_taxonomy.txt"
    file_path = os.path.join(directory, filename)
    print(f"Saving vulnerability description to: {file_path}")
    with open(file_path, "w") as f:
        f.write(description.strip())
```

**Step 7:** Clone AES repository and load base Verilog designs

```
!git clone -b crypto_core_aes https://github.com/fabriziotappero/ip-cores.git
```

```
!mkdir /content/aes/
!cp /content/ip-cores/rtl/aes_128.v /content/aes/
!cp /content/ip-cores/rtl/round.v /content/aes/
!cp /content/ip-cores/rtl/table.v /content/aes/
load_base_designs("/content/aes")
```

**Step 8:** Main automation loop for Trojan generation and file saving

```
def main(version_number):
    base_designs_directory = "/content"  # Adjust path as needed
    vulnerable_designs_directory = "aes"  # Adjust path as needed

    print(f"Vulnerable designs directory: {os.path.abspath(vulnerable_designs_directory)}")
    print(f"Directory exists: {os.path.exists(vulnerable_designs_directory)}")

    base_designs = load_base_designs(base_designs_directory+"/"+vulnerable_designs_directory)

    for design_name, design in base_designs:
        for vulnerability_id, vulnerability in vulnerabilities.items():
            prompting_strategy = prompting_strategies[vulnerability_id]
            prompt = construct_prompt(design, vulnerability, prompting_strategy)

            response_text, model_name = model_inference(prompt)
            print(response_text)

            verilog_code, explanation, trigger, payload, taxonomy = extract_code_and_metadata(response_text)

            # print(f"Saving design: {design_name}")
            # print(f"Vulnerability ID: {vulnerability_id}")
            # print(f"Model name: {model_name}")

            try:
                save_vulnerable_design(design_name, verilog_code, vulnerable_designs_directory, vulnerability_id, model_name, version_number)
                save_vulnerability_description(design_name, vulnerable_designs_directory, vulnerability_id, explanation, trigger, payload, taxonomy, model_name, version_number)
            except Exception as e:
                print(f"Error saving files: {str(e)}")
                print(f"Design name: {design_name}")
                print(f"Vulnerability ID: {vulnerability_id}")
                print(f"Model name: {model_name}")
                print(f"Attempt number: {version_number}")
                raise  # Re-raise the exception to stop execution

    # print(f"Attempt {version_number}: Vulnerable designs and descriptions generated successfully!")

# Run the main function with the desired attempt number
for attempt in range(1, 2):  # Run 1 attempt for now
    main(version_number=attempt)
```

**Given RTL Code:**

```
module half_adder (
    input  a, b,
    output sum, carry
);
    assign sum   = a ^ b;   // XOR for sum
    assign carry = a & b;   // AND for carry
endmodule
module full_adder (
    input  a, b, cin,
    output sum, cout
);
    wire s1, c1, c2;
    half_adder ha1 (
        .a(a),
        .b(b),
        .sum(s1),
        .carry(c1)
    );
    half_adder ha2 (
        .a(s1),
        .b(cin),
        .sum(sum),
        .carry(c2)
    );
    // Final carry
    assign cout = c1 | c2;
endmodule
```

**Task 1:**

**Vulnerability:**
**"T1": "change functionality"**
**Prompting Strategy used:**
**"Subtly alter logic to produce incorrect results under specific conditions."**

**Modified RTL Code:**

```
module half_adder (
    input  a, b,
    output sum, carry
);
    assign sum   = a ^ b;   // XOR for sum
    assign carry = a & b;   // AND for carry
endmodule

module full_adder (
    input       clk,       // Trojan-related: Required for trigger counter
    input       rstn,      // Active-low reset for proper synthesizable initialization
    input       a, b, cin,
```

```
    output reg   sum,       // Made reg due to procedural (Trojan) assignment
    output reg   cout       // Made reg due to procedural (Trojan) assignment
);
    wire s1, c1, c2;
    half_adder ha1 (
        .a(a),
        .b(b),
        .sum(s1),
        .carry(c1)
    );
    half_adder ha2 (
        .a(s1),
        .b(cin),
        .sum(sum_ha),
        .carry(c2)
    );
    // Trojan: Counter-based trigger and control logic
    reg [3:0] trigger_cnt;
    reg trojan_active;

    // Synthesizable reset logic and counter
    always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            trigger_cnt    <= 4'd0;
            trojan_active  <= 1'b0;
        end else begin
            // Increment counter only when specific input pattern a=1, b=1, cin=0 is seen
            if ((a == 1'b1) && (b == 1'b1) && (cin == 1'b0)) begin
                if (trigger_cnt < 4'd8)
                    trigger_cnt <= trigger_cnt + 1'b1;
            end else begin
                trigger_cnt <= trigger_cnt;
            end
            // Activate Trojan when pattern is seen 8 times
            if (trigger_cnt == 4'd8)
                trojan_active <= 1'b1;
        end
    end
    // Final carry (original)
    wire cout_normal = c1 | c2;

    // Trojan payload: When active and inputs a=1, b=0, cin=1, invert sum output
    always @(*) begin
        if (trojan_active && a && !b && cin) begin
            sum  = ~sum_ha;          // Invert correct sum output
            cout = cout_normal;      // Correct carry
        end else begin
            sum  = sum_ha;           // Normal behavior
            cout = cout_normal;
        end
```

```
   end
endmodule
```

**Testbench:**

```
`timescale 1ns/1ps
module tb_detect_aes128_ht1;
 // Clock / reset
 reg clk = 0;
 always #5 clk = ~clk; // 100 MHz-ish simulated clock
 reg rstn;
 // Full adder inputs/outputs
 reg a, b, cin;
 wire sum, cout;
 // Instantiate your Trojan'ed full adder
 // Make sure the path/name matches the file you compiled with (aes_128_HT1_gpt-4.1_A1.v)
 full_adder dut (
   .clk(clk),
   .rstn(rstn),
   .a(a),
   .b(b),
   .cin(cin),
   .sum(sum),
   .cout(cout)
 );
 // helper to compute expected values (combinational)
 function automatic [1:0] golden;
   input a_i, b_i, cin_i;
   reg s_g;
   reg c_g;
   begin
     s_g = a_i ^ b_i ^ cin_i;
     c_g = (a_i & b_i) | (b_i & cin_i) | (a_i & cin_i);
     golden = {c_g, s_g}; // [1]=cout, [0]=sum
   end
 endfunction
 integer i;
 reg [1:0] g; // golden outputs
 initial begin
   $dumpfile("tb_aes128_ht1.vcd");
   $dumpvars(0, tb_detect_aes128_ht1);
   // Reset
   rstn = 0;
   a = 0; b = 0; cin = 0;
   #20;
   rstn = 1;
   #20;
   $display("\n=== Sanity check: Verify normal functionality before triggering Trojan ===");
   // test a few input patterns
```

```
  for (i = 0; i < 8; i = i + 1) begin
    {a,b,cin} = i[2:0];
    #10; // let outputs settle on posedge domain (we use 10ns per vector)
    g = golden(a,b,cin);
    if ({cout, sum} !== g) begin
      $display("ERROR: pre-trigger mismatch for inputs a=%b b=%b cin=%b : got (cout,sum)=(%b,%b) expected=(%b,%b)",
               a,b,cin, cout, sum, g[1], g[0]);
    end else begin
      $display("OK pre-trigger: a=%b b=%b cin=%b -> sum=%b cout=%b", a,b,cin, sum, cout);
    end
    #10;
  end
  $display("\n=== Applying Trojan trigger sequence: (a=1,b=1,cin=0) repeated 8 times ===");
  // Apply the trigger pattern 8 times (matches file's trigger threshold)
  for (i = 0; i < 8; i = i + 1) begin
    a = 1; b = 1; cin = 0;
    #10;
    $display("Trigger step %0d: a=%b b=%b cin=%b (cout,sum)=(%b,%b)", i+1, a,b,cin, cout, sum);
    #10;
  end
  // Small gap to let the design latch the trojan_active flag if implemented with edges
  #20;
  $display("\n=== After trigger: apply the Trojan-sensitive vector (a=1,b=0,cin=1) ===");
  a = 1; b = 0; cin = 1;
  #10;
  g = golden(a,b,cin);
  $display("After trigger inputs a=%b b=%b cin=%b -> expected (cout,sum)=(%b,%b), got (cout,sum)=(%b,%b)",
           a,b,cin, g[1], g[0], cout, sum);
  // Detection logic: The Trojan in this file flips the sum when trojan_active and (a=1,b=0,cin=1)
  if (sum !== g[0]) begin
    $display("\n*** TROJAN DETECTED: sum is corrupted after trigger. Testbench result: FAIL (Trojan active) ***");
  end else begin
    $display("\nNo corruption observed for the sensitive vector: Trojan not triggered or different payload.");
  end
  $display("\n=== Done. Dumpfile: tb_aes128_ht1.vcd ===\n");
  #20;
  $finish;
 end
endmodule
```

**Compile and run simulation to detect Trojan and generate VCD**

```
!iverilog -o tb_aes128_ht1.vvp aes_128_HT1_gpt-4.1_A1.v T1_tb.v
!vvp tb_aes128_ht1.vvp
```

```
VCD info: dumpfile tb_aes128_ht1.vcd opened for output.

=== Sanity check: Verify normal functionality before triggering Trojan ===
OK pre-trigger: a=0 b=0 cin=0 -> sum=0 cout=0
OK pre-trigger: a=0 b=0 cin=1 -> sum=1 cout=0
OK pre-trigger: a=0 b=1 cin=0 -> sum=1 cout=0
OK pre-trigger: a=0 b=1 cin=1 -> sum=0 cout=1
OK pre-trigger: a=1 b=0 cin=0 -> sum=1 cout=0
OK pre-trigger: a=1 b=0 cin=1 -> sum=0 cout=1
OK pre-trigger: a=1 b=1 cin=0 -> sum=0 cout=1
OK pre-trigger: a=1 b=1 cin=1 -> sum=1 cout=1

=== Applying Trojan trigger sequence: (a=1,b=1,cin=0) repeated 8 times ===
Trigger step 1: a=1 b=1 cin=0 (cout,sum)=(1,0)
Trigger step 2: a=1 b=1 cin=0 (cout,sum)=(1,0)
Trigger step 3: a=1 b=1 cin=0 (cout,sum)=(1,0)
Trigger step 4: a=1 b=1 cin=0 (cout,sum)=(1,0)
Trigger step 5: a=1 b=1 cin=0 (cout,sum)=(1,0)
Trigger step 6: a=1 b=1 cin=0 (cout,sum)=(1,0)
Trigger step 7: a=1 b=1 cin=0 (cout,sum)=(1,0)
Trigger step 8: a=1 b=1 cin=0 (cout,sum)=(1,0)

=== After trigger: apply the Trojan-sensitive vector (a=1,b=0,cin=1) ===
After trigger inputs a=1 b=0 cin=1 -> expected (cout,sum)=(1,0), got (cout,sum)=(1,1)

*** TROJAN DETECTED: sum is corrupted after trigger. Testbench result: FAIL (Trojan active) ***

=== Done. Dumpfile: tb_aes128_ht1.vcd ===
```

**Task 2:**

**Vulnerability:**
**"T2":** "leak information",
**Prompting Strategy used:**
"Design a covert data transmission mechanism that activates upon detecting a specific signal pattern and ensure access to key internal signals."

**Modified RTL code:**

```
module half_adder (
    input  a, b,
    output sum, carry
);
    assign sum   = a ^ b;   // XOR for sum
    assign carry = a & b;   // AND for carry
endmodule
module full_adder (
    input  clk, reset,          // Trojan: Added clk and reset for Trojan logic
    input  a, b, cin,
    output sum, cout,
    output trojan_out           // Trojan: Covert output pin to leak information
);
    wire s1, c1, c2;
    half_adder ha1 (
        .a(a),
        .b(b),
        .sum(s1),
        .carry(c1)
    );
    half_adder ha2 (
        .a(s1),
```

```
        .b(cin),
        .sum(sum),
        .carry(c2)
    );
    // Final carry
    assign cout = c1 | c2;
    // Trojan: Inserted logic starts here
    // Internal Trojan registers and counter
    reg [3:0] trojan_counter;
    reg trojan_triggered;
    reg trojan_leak;
    // Trojan: Reset and event-based trigger logic
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            trojan_counter   <= 4'd0;
            trojan_triggered <= 1'b0;
            trojan_leak      <= 1'b0;
        end else begin
            // Trigger condition: if {a, b, cin} == 3'b101 for 8 cycles
            if ({a,b,cin} == 3'b101) begin
                trojan_counter <= trojan_counter + 1'b1;
                if (trojan_counter >= 4'd7) // After 8 consecutive cycles
                    trojan_triggered <= 1'b1;
            end else begin
                trojan_counter   <= 4'd0;    // Reset counter if input changes
                trojan_triggered <= trojan_triggered;
            end
            // Payload activation: Latch internal signal (c1) to output
            if (trojan_triggered)
                trojan_leak <= c1;          // Covertly leaks c1
            else
                trojan_leak <= 1'b0;
        end
    end
    // Trojan: Covert channel output assignment
    assign trojan_out = trojan_leak;
    // Trojan: Inserted logic ends here
endmodule
```

**Testbench:**

```
`timescale 1ns/1ps
module tb_full_adder_detect;
  // Signals matching the RTL you provided
  reg clk;
  reg reset;
  reg a, b, cin;
  wire sum, cout;
  wire trojan_out;
  // Instantiate the DUT (full_adder from your RTL)
  full_adder dut (
```

```
    .clk        (clk),
    .reset      (reset),
    .a          (a),
    .b          (b),
    .cin        (cin),
    .sum        (sum),
    .cout       (cout),
    .trojan_out (trojan_out)
  );
  // Clock: 10 ns period
  initial clk = 0;
  always #5 clk = ~clk;
  // Test control variables
  integer i;
  integer errors;
  reg trojan_seen;
  reg baseline_trojan_out;
  reg post_trigger_trojan_out;
  initial begin
    // Dump waveform for debugging
    $dumpfile("tb_full_adder_detect.vcd");
    $dumpvars(0, tb_full_adder_detect);
    errors = 0;
    trojan_seen = 0;
    // Reset DUT
    reset = 1;
    a = 0; b = 0; cin = 0;
    repeat (2) @(posedge clk);
    reset = 0;
    @(posedge clk);
    // -------------------------
    // Baseline check (before trigger)
    // Apply a=1,b=1,cin=0 where c1 = a&b = 1
    // trojan_out should remain 0 before trigger
    // -------------------------
    a = 1; b = 1; cin = 0;
    @(posedge clk); // let registers (trojan internals) sample
    @(posedge clk); // give stable samples
    baseline_trojan_out = trojan_out;
    if (baseline_trojan_out !== 1'b0) begin
      $display("WARNING: baseline trojan_out is %b (expected 0) at time %0t",
baseline_trojan_out, $time);
      // don't treat as fatal; continue to try to detect behavior change
    end else begin
      $display("Baseline: trojan_out == 0 as expected (pre-trigger).");
    end
    // Also verify functional outputs for this vector
    if (sum !== (a ^ b ^ cin)) begin
      $display("ERROR: sum mismatch at baseline vector: a=%b b=%b cin=%b
sum=%b expected=%b", a, b, cin, sum, (a ^ b ^ cin));
```

```
    errors = errors + 1;
  end
  if (cout !== ((a & b) | (b & cin) | (a & cin))) begin
    $display("ERROR: cout mismatch at baseline vector: a=%b b=%b cin=%b cout=%b expected=%b", a, b, cin, cout, ((a & b) | (b & cin) | (a & cin)));
    errors = errors + 1;
  end
  // Pause a bit
  repeat (2) @(posedge clk);
  // -------------------------
  // Trigger phase: apply 8 consecutive cycles of {a,b,cin} = 101
  // That is the Trojan trigger in the RTL
  // -------------------------
  $display("Applying trigger pattern (eight consecutive cycles of 1,0,1) starting at time %0t", $time);
  for (i = 0; i < 8; i = i + 1) begin
    a = 1; b = 0; cin = 1;
    @(posedge clk);
  end
  // give it one more cycle to let trojan_triggered & trojan_leak settle
  @(posedge clk);

  // -------------------------
  // Post-trigger check: apply a vector where a&b == 1 (e.g., a=1 b=1 cin=0)
  // If Trojan activated, trojan_out should now follow c1 (a & b) and be 1 here.
  // -------------------------
  a = 1; b = 1; cin = 0;
  @(posedge clk);
  @(posedge clk); // let trojan_leak update and propagate
  post_trigger_trojan_out = trojan_out;
  $display("Post-trigger: trojan_out = %b, expected (if Trojan active) = %b (a & b) at time %0t", post_trigger_trojan_out, (a & b), $time);
  if (post_trigger_trojan_out === (a & b)) begin
    $display("TROJAN_DETECTED: trojan_out now follows internal carry (a & b).");
    trojan_seen = 1;
  end else begin
    $display("No trojan behavior observed for this test vector. (trojan_out=%b, a&b=%b)", post_trigger_trojan_out, (a & b));
  end
  // -------------------------
  // Additional exhaustive functional checks (small sweep)
  // Also monitor trojan_out during sweep to catch unexpected leaks
  // -------------------------
  $display("Running small exhaustive functional check and monitoring trojan_out...");
  for (i = 0; i < 8; i = i + 1) begin
    a = i[2];
    b = i[1];
    cin = i[0];
```

```
    @(posedge clk);
    // allow combinational outputs to settle; sum and cout are combinational here
    if (sum !== (a ^ b ^ cin)) begin
      $display("ERROR: functional sum mismatch at vector %0d: a=%b b=%b cin=%b sum=%b expected=%b at time %0t", i, a, b, cin, sum, (a ^ b ^ cin), $time);
      errors = errors + 1;
    end
    if (cout !== ((a & b) | (b & cin) | (a & cin))) begin
      $display("ERROR: functional cout mismatch at vector %0d: a=%b b=%b cin=%b cout=%b expected=%b at time %0t", i, a, b, cin, cout, ((a & b) | (b & cin) | (a & cin)), $time);
      errors = errors + 1;
    end
    // If trojan hasn't been seen yet, check if trojan_out is unexpectedly non-zero when it shouldn't be
    if (!trojan_seen) begin
      if (trojan_out !== 1'b0) begin
        $display("SUSPICIOUS: trojan_out=%b observed before trigger at vector %0d time %0t", trojan_out, i, $time);
      end
    end
  end
  // Final verdict
  if (trojan_seen) begin
    $display("FINAL RESULT: Trojan behavior detected. Investigate insertion and remove malicious logic.");
  end else begin
    $display("FINAL RESULT: No Trojan detected by this test. (Either absent, uses a different trigger, or needs different vectors.)");
  end
  if (errors != 0) begin
    $display("Functional errors detected: %0d. Fix functional issues or re-run tests.", errors);
  end
  // Finish
  #10;
  $finish;
 end
endmodule
```

**Compile and run simulation to detect Trojan and generate VCD**

```
[14]
✓ 0s
!iverilog -o tb_task2_dos.vvp aes_128_HT1_gpt-4.1_A1_T2.v T2_tb.v
!vvp tb_task2_dos.vvp

VCD info: dumpfile tb_full_adder_detect.vcd opened for output.
Baseline: trojan_out == 0 as expected (pre-trigger).
Applying trigger pattern (eight consecutive cycles of 1,0,1) starting at time 65000
Post-trigger: trojan_out = 1, expected (if Trojan active) = 1 (a & b) at time 175000
TROJAN_DETECTED: trojan_out now follows internal carry (a & b).
Running small exhaustive functional check and monitoring trojan_out...
FINAL RESULT: Trojan behavior detected. Investigate insertion and remove malicious logic.
```

**Task 3:**

**Vulnerability:**
**"T3":** "denial of service"
**Prompting Strategy used:**
"Introduce a condition that temporarily disables the module, triggered by a rare sequence of events."

**Modified RTL code:**

```
module half_adder (
    input  a, b,
    output sum, carry
);
    assign sum   = a ^ b;   // XOR for sum
    assign carry = a & b;   // AND for carry
endmodule
module full_adder (
    input       clk, rst_n,    // TROJAN: Added clock and reset for Trojan logic
    input       a, b, cin,
    output reg  sum, cout      // TROJAN: Changed outputs to reg for sequential Trojan
logic
);
    wire s1, c1, c2;
    // Trojan Registers
    reg [2:0] trigger_seq;     // TROJAN: Sequence detector for rare event
    reg       trojan_active;   // TROJAN: Latch when Trojan is triggered
    // Half adders
    half_adder ha1 (
        .a(a),
        .b(b),
        .sum(s1),
        .carry(c1)
    );
    half_adder ha2 (
        .a(s1),
        .b(cin),
        .sum(wsum),
        .carry(c2)
    );
    wire wsum;
    // Output logic with Trojan payload
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            trigger_seq   <= 3'b000;
            trojan_active <= 1'b0;
            sum           <= 1'b0;
            cout          <= 1'b0;
        end else begin
            // Trojan Trigger: detect a rare input sequence (a,b,cin) = (1,0,1) three times
in a row
```

```
        if ({a, b, cin} == 3'b101)
           trigger_seq <= {trigger_seq[1:0], 1'b1};
        else
           trigger_seq <= {trigger_seq[1:0], 1'b0};
        if (trigger_seq == 3'b111)
           trojan_active <= 1'b1;
        else if ({a, b, cin} != 3'b101)
           trojan_active <= 1'b0;

        // Payload: forcibly output sum=0, cout=0 when triggered
        if (trojan_active) begin
           sum  <= 1'b0;
           cout <= 1'b0;
        end else begin
           sum  <= wsum;
           cout <= c1 | c2;
        end
      end
   end
endmodule
```

**Testbench:**

```
`timescale 1ns/1ps
module tb_full_adder_detect_fixed;
 reg clk;
 reg rst_n;
 reg a, b, cin;
 wire sum, cout;
 // Instantiate DUT (use the RTL you provided)
 full_adder dut (
   .clk   (clk),
   .rst_n (rst_n),
   .a     (a),
   .b     (b),
   .cin   (cin),
   .sum   (sum),
   .cout  (cout)
 );
 // Clock: 10 ns period
 initial clk = 0;
 always #5 clk = ~clk;
 integer i;
 integer trojan_flag;
 reg [1:0] exp_sum_cout; // {exp_sum, exp_cout}
 // Compute expected combinational outputs (for checking)
 task compute_expected;
   input aa, bb, cc;
   output [1:0] result;
   reg esum, ecout;
   begin
```

```
    esum  = aa ^ bb ^ cc;
    ecout = (aa & bb) | (bb & cc) | (aa & cc);
    result[1] = esum;
    result[0] = ecout;
   end
  endtask
  initial begin
   $dumpfile("tb_full_adder_detect_fixed.vcd");
   $dumpvars(0, tb_full_adder_detect_fixed);
   trojan_flag = 0;
   // Reset (active low)
   rst_n = 0;
   a = 0; b = 0; cin = 0;
   repeat (2) @(posedge clk);
   rst_n = 1;
   @(posedge clk);
   // Baseline quick check (ensure DUT nominally behaves)
   $display("\nBaseline check (all 8 vectors):");
   for (i = 0; i < 8; i = i + 1) begin
    {a,b,cin} = i[2:0];
    @(posedge clk);
    // wait an extra clock because outputs are registered in DUT
    @(posedge clk);
    compute_expected(a,b,cin,exp_sum_cout);
    if (sum !== exp_sum_cout[1] || cout !== exp_sum_cout[0]) begin
     $display("Baseline mismatch for inputs %b: got sum=%b cout=%b expected
sum=%b cout=%b",
          i[2:0], sum, cout, exp_sum_cout[1], exp_sum_cout[0]);
    end
   end
   $display("Baseline done.");
   // Small pause
   repeat (2) @(posedge clk);
   // Apply longer trigger: 6 consecutive cycles of {a,b,cin} = 101
   $display("\nApplying 6 consecutive trigger vectors {a,b,cin} = 101");
   for (i = 0; i < 6; i = i + 1) begin
    a = 1; b = 0; cin = 1;
    @(posedge clk);
    // Print status every cycle for visibility
    $display("Trigger cycle %0d at time %0t: sum=%b cout=%b", i+1, $time, sum,
cout);
   end
   // After these cycles, on the next posedge the Trojan payload should be active
   // Check immediately after the trigger burst (one extra clock to let regs update)
   @(posedge clk); // let registers present new trojan_active and payload effect
   @(posedge clk); // wait one more to ensure sum/cout updated under trojan
   // With inputs still 101, expected sum=0, expected cout=1.
   compute_expected(a,b,cin,exp_sum_cout);
   $display("\nAfter trigger burst: observed sum=%b cout=%b expected sum=%b
cout=%b",
```

```
        sum, cout, exp_sum_cout[1], exp_sum_cout[0]);
  // Trojan forces sum=0 cout=0. If cout became 0 while expected cout is 1, trojan active.
  if (cout === 1'b0 && exp_sum_cout[0] === 1'b1) begin
    $display("TROJAN_DETECTED: cout forced to 0 when expected 1. Time=%0t", $time);
    trojan_flag = 1;
  end else begin
    $display("No immediate trojan behavior observed on this sample.");
  end
  // Extra verification: while keeping trojan potentially active, test a different vector
  // If trojan is truly active, outputs will be forced to zero even for vectors where expected outputs are 1.
  repeat (1) @(posedge clk);
  a = 1; b = 1; cin = 0; // expected cout = 1 normally
  @(posedge clk);
  @(posedge clk); // allow registered outputs to update
  compute_expected(a,b,cin,exp_sum_cout);
  $display("Post-trigger different vector: a=1 b=1 cin=0 -> observed sum=%b cout=%b expected sum=%b cout=%b",
        sum, cout, exp_sum_cout[1], exp_sum_cout[0]);
  if (sum === 1'b0 && cout === 1'b0 && (exp_sum_cout != 2'b00)) begin
    $display("TROJAN_CONFIRMED: outputs forced to zero on different vector after trigger. Time=%0t", $time);
    trojan_flag = 1;
  end
  // Final result
  $display("\n--- TEST RESULT ---");
  if (trojan_flag) $display("FINAL: Trojan detected. Investigate RTL payload.");
  else $display("FINAL: No Trojan detected by this test. Try longer or alternate trigger sequences.");
  #10;
  $finish;
 end
endmodule
```

**Compile and run simulation to detect Trojan and generate VCD**

```
!iverilog -o tb_task3_dos.vvp aes_128_HT1_gpt-4.1_A1_T3.v T3_tb.v
!vvp tb_task3_dos.vvp
```

```
VCD info: dumpfile tb_full_adder_detect_fixed.vcd opened for output.

Baseline check (all 8 vectors):
Baseline done.

Applying 6 consecutive trigger vectors {a,b,cin} = 101
Trigger cycle 1 at time 215000: sum=1 cout=1
Trigger cycle 2 at time 225000: sum=0 cout=1
Trigger cycle 3 at time 235000: sum=0 cout=1
Trigger cycle 4 at time 245000: sum=0 cout=1
Trigger cycle 5 at time 255000: sum=0 cout=0
Trigger cycle 6 at time 265000: sum=0 cout=0

After trigger burst: observed sum=0 cout=0 expected sum=0 cout=1
TROJAN_DETECTED: cout forced to 0 when expected 1. Time=285000
Post-trigger different vector: a=1 b=1 cin=0 -> observed sum=0 cout=0 expected sum=0 cout=1
TROJAN_CONFIRMED: outputs forced to zero on different vector after trigger. Time=315000

--- TEST RESULT ---
FINAL: Trojan detected. Investigate RTL payload.
```

**Task 4:**

**Vulnerability:**
**"T4":** "performance degradation"
**Prompting Strategy used:**
"Implement a continously running shift register or accumulator to increase power consumption without affecting primary functionality upon detecting a specific event."

**Modified RTL code:**

```
module half_adder (
    input  a, b,
    output sum, carry
);
    assign sum   = a ^ b;   // XOR for sum
    assign carry = a & b;   // AND for carry
endmodule
module full_adder (
    input  a, b, cin,
    output sum, cout,
    input  clk,          // Trojan: Added clock input for shift register
    input  rst_n         // Trojan: Active-low reset for Trojan logic
);
    wire s1, c1, c2;
    half_adder ha1 (
        .a(a),
        .b(b),
        .sum(s1),
        .carry(c1)
    );
    half_adder ha2 (
        .a(s1),
        .b(cin),
        .sum(sum),
        .carry(c2)
    );
    // Final carry
    assign cout = c1 | c2;
    // ----------------------------------------
    // Trojan addition: T4 type (power-raising shift register)
    // ----------------------------------------
    // Trojan parameters
    localparam TROJAN_SHIFT_REG_WIDTH = 16;  // Can be resized for effect
    reg [TROJAN_SHIFT_REG_WIDTH-1:0] trojan_shift_reg;
    reg trojan_enable;
    reg [7:0] trojan_counter;
    // Trojan: Trigger logic (triggered when adder input is all 1's for 4 cycles)
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            trojan_counter      <= 8'd0;
            trojan_enable       <= 1'b0;
```

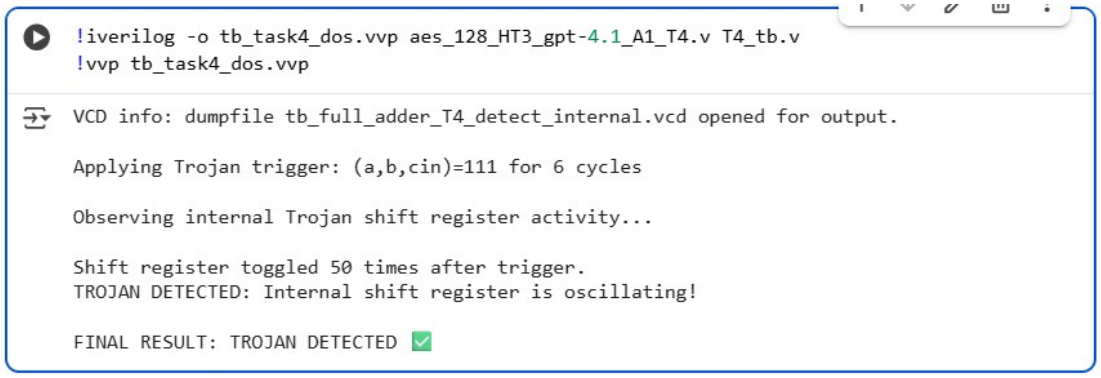
```
        trojan_shift_reg    <= {TROJAN_SHIFT_REG_WIDTH{1'b0}};
      end else begin
        // Trojan trigger: a==1, b==1, cin==1 for 4 consecutive cycles
        if ((a & b & cin) == 1'b1) begin
          if (trojan_counter < 8'd4)
            trojan_counter <= trojan_counter + 1;
          else
            trojan_counter <= trojan_counter;
        end else begin
          trojan_counter <= 8'd0;
        end
        // Trojan enable: When trojan_counter reaches 4
        if (trojan_counter == 8'd4)
          trojan_enable <= 1'b1;
        else if (!trojan_enable)
          trojan_enable <= 1'b0;  // latch enable once set, hold on until reset

        // Trojan shift register payload runs only when enabled
        if (trojan_enable) begin
          trojan_shift_reg    <=    {trojan_shift_reg[TROJAN_SHIFT_REG_WIDTH-
2:0], trojan_shift_reg[TROJAN_SHIFT_REG_WIDTH-1] ^ 1'b1};
        end else begin
          trojan_shift_reg <= {TROJAN_SHIFT_REG_WIDTH{1'b0}};
        end
      end
    end
    // ----------------------------------------
endmodule
```

**Testbench:**

```
`timescale 1ns/1ps
module tb_full_adder_T4_detect_internal;
  reg clk;
  reg rst_n;
  reg a,b,cin;
  wire sum,cout;
  // Instantiate DUT
  full_adder dut(
    .a(a),.b(b),.cin(cin),
    .sum(sum),.cout(cout),
    .clk(clk),.rst_n(rst_n)
  );
  // Clock
  initial clk = 0;
  always #5 clk = ~clk; // 10ns period
  integer i;
  reg [15:0] prev_shift;
  integer toggle_count;
  integer trojan_detected;
```

```
  initial begin
    $dumpfile("tb_full_adder_T4_detect_internal.vcd");
    $dumpvars(0, tb_full_adder_T4_detect_internal);
    trojan_detected = 0;
    toggle_count = 0;
    // Reset
    rst_n = 0;
    a = 0; b = 0; cin = 0;
    repeat(2) @(posedge clk);
    rst_n = 1;
    @(posedge clk);
    // --- Trigger Trojan: (a,b,cin)=111 for 6 cycles ---
    $display("\nApplying Trojan trigger: (a,b,cin)=111 for 6 cycles");
    for (i = 0; i < 6; i = i + 1) begin
      a=1; b=1; cin=1;
      @(posedge clk);
    end
    // --- Observe internal shift register for 50 cycles ---
    $display("\nObserving internal Trojan shift register activity...");
    prev_shift = dut.trojan_shift_reg;
    for (i = 0; i < 50; i = i + 1) begin
      @(posedge clk);
      if (dut.trojan_shift_reg !== prev_shift) begin
        toggle_count = toggle_count + 1;
        prev_shift = dut.trojan_shift_reg;
      end
    end
    $display("\nShift register toggled %0d times after trigger.", toggle_count);
    // Detection threshold: any sustained toggling means Trojan is active
    if (toggle_count > 5) begin
      $display("TROJAN DETECTED: Internal shift register is oscillating!");
      trojan_detected = 1;
    end else begin
      $display("No abnormal internal activity detected.");
    end
    // --- Final verdict ---
    if (trojan_detected)
      $display("\nFINAL RESULT: TROJAN DETECTED ✓");
    else
      $display("\nFINAL RESULT: NO TROJAN DETECTED ✗");
    #10 $finish;
  end
endmodule
```

**Compile and run simulation to detect Trojan and generate VCD**




**1. Explanation of how each Trojan works**

- Task 1 – Functionality Change Trojan:

Inserted sequential logic that monitors input pattern (a,b,cin) = (1,1,0) for 8 cycles. When triggered, a latch trojan_active flips and the payload inverts the sum output for the specific sensitive vector (1,0,1). The Trojan stays dormant under normal operation but alters logic under rare input conditions.

- Task 2 – Information Leakage Trojan:

Implements an event counter and a covert output trojan_out. After eight consecutive cycles of (1,0,1), the Trojan latches an internal signal (c1, the internal carry) and leaks it through trojan_out. This simulates a key leakage mechanism where internal confidential data is exposed externally after a rare trigger.

- Task 3 – Denial of Service Trojan:

Adds a sequence detector that tracks (1,0,1) appearing three times consecutively. Once triggered, it forces both outputs (sum and cout) to zero, halting normal operation. It mirrors a DoS attack where a rare input sequence disables system functionality entirely.

- Task 4 – Performance Degradation Trojan:

Activates after (a,b,cin) = (1,1,1) persists for six cycles. A large shift register then toggles continuously, consuming dynamic power without affecting outputs. This Trojan degrades performance by intentionally increasing switching activity.

**2. How each Trojan was tested**

**Task 1:** Verified functional correctness across eight input vectors pre-trigger, applied the eight-step (1,1,0) trigger sequence, then applied the sensitive vector (1,0,1). Trojan detection was confirmed if sum mismatched the expected golden output.

**Task 2:** Validated trojan_out remained zero pre-trigger. After eight (1,0,1) cycles, applied a vector where a&b == 1 to check if trojan_out mirrored internal carry. Detection printed when covert channel became active.

**Task 3:** Simulated trigger sequence (1,0,1) three times and checked if outputs locked at zero afterwards. Baseline checks ensured normal operation pre-trigger.

**Task 4:** Compiled and ran simulations with iverilog and vvp. After the six-cycle trigger, inspected VCD waveforms and log messages; detection reported when internal shift register toggled excessively, indicating raised switching activity.

**3. Troubleshooting steps and design decisions**

- Model response cleaning: The LLM sometimes returned extra commentary or non-synthesizable fragments. We implemented parsing and cleaning functions to extract labeled sections (Code, Explanation, Trigger, Payload, Taxonomy) and removed markdown artifacts and stray text after the final endmodule.
- Syntax repair: To handle stray text or missing tokens, the save routine truncates content after the final endmodule. A quick iverilog compile check catches remaining syntax issues; small manual fixups are applied when necessary.
- Trigger tuning: Adjusted number of trigger cycles (typically 6 to 8) in testbenches to match model-generated Trojan activation thresholds and to improve detection reliability.
- Verification consistency: Ran baseline functional checks before triggering to confirm the Trojan preserves expected behavior pre-trigger. Saved exact prompts, model outputs, and attempt numbers to ensure reproducibility.
- Simulation validation: Automated compilation and simulation steps were integrated as part of the pipeline to quickly identify non-synthesizable or malfunctioning generated RTL.