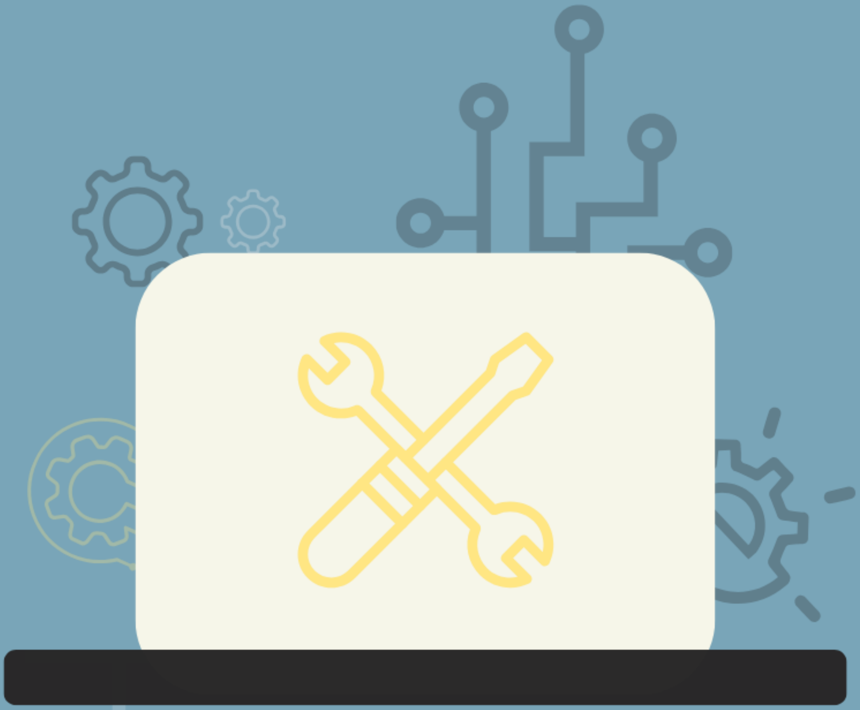
Ain Shams University - Faculty of Engineering



VERILOG LINT TOOL DESIGN

PROJECT 2 REPORT

CSE 312: Electronic Design Automation

Presented to:

Dr. Eman El Mandouh

Eng. Abdelrahman Sherif

Presented by Team 5:

Abdallah Belal	22P0036
Ahmed Khaled	22P0024
Malak Zaiter	22P0052
Nour Hossam	2201386
Seif Elden Ibrahim	22P0023
Yousef Sameh	22P0010

Table of Contents

Table of Figures

Abstract

The objective of this project is to design and implement Verilog Linter, a static analysis tool that checks Verilog HDL (Hardware Description Language) code for common design issues without requiring simulation or synthesis. The tool parses a "Design Under Test" (DUT) and identifies potential violations, including arithmetic overflow, unreachable FSM states, uninitialized registers, combinational loops, unreachable branches, missing default cases, and X propagation issues. By analyzing Verilog code through pattern matching and logical evaluation, the tool produces a detailed report highlighting detected violations, their corresponding line numbers, and impacted variables. Implemented using C++, the linter leverages regular expressions and efficient data structures to ensure fast and accurate analysis. This project aims to enhance hardware design verification by providing an accessible, modular, and extensible static analysis solution for developers working with Verilog HDL.

Introduction

This project involves the development of a Verilog Linter tool named **Lolinta**, designed to perform static analysis of Verilog code. **Lolinta** identifies common design issues without requiring simulation or testbenches, making it an efficient tool for improving Verilog design quality. By analyzing Verilog designs and providing actionable feedback, **Lolinta** ensures high reliability and adherence to industry best practices.

Project objectives

The main objectives of the **Lolinta** Verilog Linter project are as follows:

- **Automate Code Inspection**: automatically detect design issues in Verilog code through static analysis.
- **Ensure Code Quality**: identify common violations such as unreachable FSM states, arithmetic overflow, and dead code.
- **Improve Verification Efficiency**: provide clear, actionable reports without requiring simulations or testbenches.
- **Design an Extensible Tool**: create a modular and scalable tool that can be enhanced with additional features in the future.

System Design

The system design of **Lolinta** is structured into three main components: the **Parser**, the **Static Checker Engine**, and the **Report Generator**. These components work together to analyze Verilog code, detect violations, and produce actionable feedback for users.

Overall Architecture

The system is divided into the following modules, each with specific responsibilities:

1. **Parser**: Responsible for reading and tokenizing Verilog code by reads the source file and storing its contents line by line.
2. **Static Checker Engine**: Analyzes the parsed code to detect specific violations.
3. **Report Generator**: Formats and outputs a comprehensive report listing the identified issues.

Component Design

1. Verilog Parser

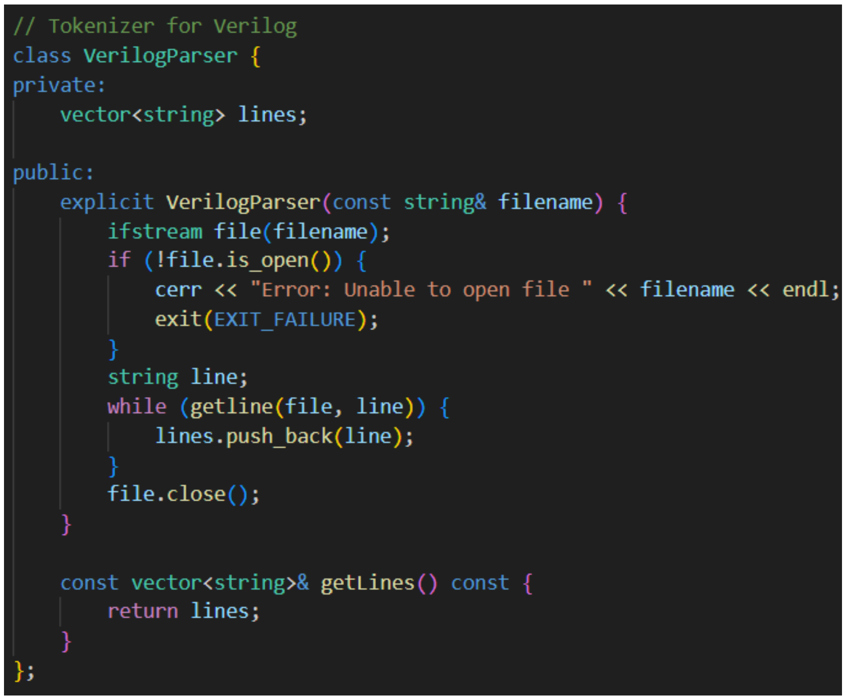
Purpose: The parser reads the input Verilog file line by line and prepares it for analysis by the Static Checker Engine.

Features:

- Handles single flattened modules (no hierarchy or instantiation).
- Ignores testbench code as per project requirements.

Implementation:

- Reads the file using standard file input techniques.
- Stores the code in memory as a vector of strings, allowing line-by-line analysis.



```
// Tokenizer for Verilog
class VerilogParser {
private:
    vector<string> lines;

public:
    explicit VerilogParser(const string& filename) {
        ifstream file(filename);
        if (!file.is_open()) {
            cerr << "Error: Unable to open file " << filename << endl;
            exit(EXIT_FAILURE);
        }
        string line;
        while (getline(file, line)) {
            lines.push_back(line);
        }
        file.close();
    }

    const vector<string>& getLines() const {
        return lines;
    }
};
```

Figure 1: Verilog Parser code snippet

Workflow:

- Open the file and read it line by line.
- Store each line in a vector<string> for further processing.

2. Static Checker Engine

The Static Checker Engine is the core of the Verilog Linter. It performs static analysis to detect design issues. The following checks are implemented:

[1] Arithmetic Overflow:

Purpose: Detects operations that may exceed the bit-width of registers.
Implementation:

- Uses regular expressions to identify arithmetic operations.
- Evaluates the bit-width of operands to check for potential overflow.

```
void checkArithmeticOverflow() {
    regex arithmeticPattern(R"(\s*(\w+)\s*=\s*(.+);)");
    regex operationPattern(R"((\w+)\s*([\+\-\*/])\s*(\w+))");

    for (size_t i = 0; i < lines.size(); ++i) {
        string line = lines[i];
        smatch match;

        if (regex_search(line, match, arithmeticPattern)) {
            string expression = match[2];
            if (regex_search(expression, match, operationPattern)) {
                violations.push_back({ "Potential arithmetic overflow in operation: " +
match[0], i + 1 });
            }
        }
    }
}
```

Figure 2: Check Arithmetic Overflow function

[2] **Unreachable Blocks:**

Purpose: Identifies code blocks that are unreachable due to constant conditions.

Implementation:

- Looks for conditions that are always *false* (e.g., *if (1'b0)*).

```
void checkDeadCode() {

    regex ifConditionRegex(R"(\s*(if|else if)\s*\(\s*(.*?)\s*\))");

    for (size_t i = 0; i < lines.size(); ++i) {

        string line = lines[i];

        smatch match;

        if (regex_search(line, match, ifConditionRegex)) {

            string condition = match[2];

            if (condition == "1'b0" || condition == "0") {

                violations.push_back({ "Unreachable if-else statement at line: " +
to_string(i + 1), i + 1 });

            }

        }

    }

}
```

Figure 3: Check Dead Code function

[3] **Unreachable FSM States:**

Purpose: Ensures all FSM states in *case* blocks are reachable.

Implementation:

- Tracks all defined states and transitions.
- Compares reachable states with defined states.

```
void checkUnreachableFSMStates() {
    regex casePattern(R"(case\s*\(?\s*(\w+)\s*\)?)");
    regex statePattern(R"(\s*(\w+)\s*:)");

    unordered_set<string> allStates;
    unordered_set<string> reachableStates;

    for (const string& line : lines) {
        smatch match;
        if (regex_search(line, match, casePattern)) {
            allStates.insert(match[1]);
        }
        if (regex_search(line, match, statePattern)) {
            reachableStates.insert(match[1]);
        }
    }

    for (const auto& state : allStates) {
        if (reachableStates.find(state) == reachableStates.end()) {
            violations.push_back({ "Unreachable FSM state: " + state, 0 });
        }
    }
}
```

Figure 4: Check Unreachable FSM State function

[4] Unreachable Registers:

Purpose: Ensures all declared registers (*reg*) are initialized.

Implementation:

- Tracks all *reg* declarations and assignments.

```
void checkUninitializedRegisters() {
    regex regPattern("\\breg\\b\\s+(\\w+)");
    regex assignmentPattern("(\\w+)\\s*=\\s*");
    unordered_set<string> declaredRegisters;
    unordered_set<string> initializedRegisters;
    for (int i = 0; i < lines.size(); ++i) {
        string line = lines[i];
        smatch match;
        if (regex_search(line, match, regPattern)) {
            declaredRegisters.insert(match[1]);
        }
        if (regex_search(line, match, assignmentPattern)) {
            initializedRegisters.insert(match[1]);
        }
    }
    for (const string& reg : declaredRegisters) {
        if (initializedRegisters.find(reg) == initializedRegisters.end()) {
            violations.push_back({ "Uninitialized register: " + reg, 0 });
        }
    }
}
```

Figure 5: Check Uninitialized Registers function

[5] Multi-Driven Bus/Register:

Purpose: Detects conflicts caused by multiple drivers attempting to control the same signal or register in combinational logic, leading to undefined behavior.

Implementation:

- The program builds a dependency graph using regex to identify signal assignments.
- It tracks assignments for each signal and checks if a signal is driven by multiple sources simultaneously.
- Reports violations for multi-driven signals.

Key Features:

- Supports complex dependency tracking.
- Handles multiple signal declarations in a single line.

```
    void checkMultiDrivenBus() {
        regex assignPattern(R"(assign\s+(\w+)\s*=\s*(.*?);)"); // Matches 'assign bus = 
...;'
        unordered_map<string, vector<string>> busAssignments;   // Tracks conditions 
per bus


        for (size_t i = 0; i < lines.size(); ++i) {
            string line = lines[i];
            smatch match;


            // Search for 'assign' statements
            if (regex_search(line, match, assignPattern)) {
                string busName = match[1];      // Extract the bus name
                string condition = match[2];    // Extract the assigned condition or 
value


                busAssignments[busName].push_back(condition);


                // Check for conflicting drivers
                if (busAssignments[busName].size() > 1) {
                    violations.push_back({ "Bus value conflict detected: " + busName, 
static_cast<int>(i + 1) });
                }
            }
        }
    }
```

Figure 6: Multi-Driven Bus code snippet

[6] Non-Full/Parallel Case Statements:

Purpose: Checks the completeness and correctness of *case* statements in Verilog code:

1. Identifies missing *default* cases.
2. Detects duplicate or overlapping conditions that could cause ambiguity.

Implementation:

- Scans for *case* blocks using regex.
- Validates:
    - Presence of a *default* case.
    - Uniqueness of each condition within the block.
- Reports violations for missing defaults and duplicate conditions.

Key Features:

- Ensures robust handling of edge cases.
- Identifies potential bugs in *case* logic.

```
regex casePattern(R"(case\s*\((\w+)\))");
regex endCasePattern(R"(endcase)");
regex defaultPattern(R"(default:)");
regex conditionPattern(R"(\s*(\d+'[bB]?\w+|[a-zA-Z_]\w*)\s*:)");


for (int i = 0; i < lines.size(); ++i) {
    string line = lines[i];
    smatch match;

    if (regex_search(line, match, casePattern)) {
        unordered_set<string> cases;
        bool hasDefault = false;
        int caseStartLine = i;

        while (++i < lines.size()) {
            line = lines[i];
            if (regex_search(line, defaultPattern)) {
                hasDefault = true;
            }
```

Figure 7: Non-Full/Parallel Case Statements code snippet (1)

```
            smatch conditionMatch;
            if (regex_search(line, conditionMatch, conditionPattern)) {
                string condition = conditionMatch[1];
                if (!cases.insert(condition).second) {
                    violations.push_back({ "Duplicate condition in case statement: " +
condition, i + 1 });
                }
            }

            if (regex_search(line, endCasePattern)) {
                break;
            }
        }

        if (!hasDefault) {
            violations.push_back({ "Missing default case in case statement starting at
line: " + to_string(caseStartLine + 1), caseStartLine + 1 });
        }
    }
}
```
Figure 8: Non-Full/Parallel Case Statements code snippet (2)

[7] **Infer Latch:**

Purpose: Detects unintended latches caused by incomplete assignments in always blocks. Latches occur when signals are conditionally assigned without handling all possible conditions.

Implementation:

- Extracts *always* blocks from the Verilog code.
- Tracks *if* and *else* conditions for coverage.
- Flags signals with incomplete conditions as latch candidates.

Key Features:

- Handles nested *if-else* constructs.
- Validates conditions comprehensively.

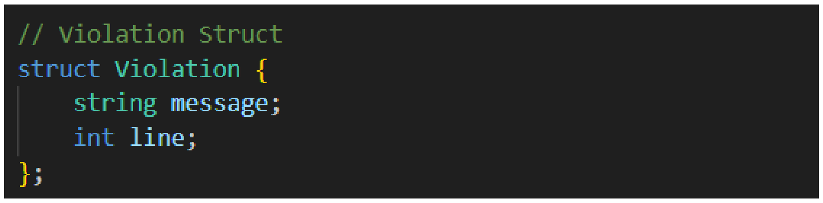
```
regex alwaysBlockRegex(R"(always\s*@\*\s*begin([\s\S]*?)end\s*)");

smatch alwaysBlockMatch;

vector<string> alwaysBlocks;


auto codeBegin = verilogCode.cbegin();

auto codeEnd = verilogCode.cend();

while (regex_search(codeBegin, codeEnd, alwaysBlockMatch, alwaysBlockRegex)) {

    alwaysBlocks.push_back(alwaysBlockMatch[1].str());

    codeBegin = alwaysBlockMatch.suffix().first;

}


for (const auto& block : alwaysBlocks) {

    stack<bool> ifHasElseStack;


    istringstream blockStream(block);

    string line;

    while (getline(blockStream, line)) {

        line = regex_replace(line, regex(R"(^\s+|\s+$)"), "");


        if (regex_search(line, regex(R"(\bif\s*\()"))) {

            ifHasElseStack.push(false);

        } else if (regex_search(line, regex(R"(\belse\b)"))) {

            if (!ifHasElseStack.empty()) {

                ifHasElseStack.top() = true;

            }

        } else if (regex_search(line, regex(R"(\bend\b)")) && !ifHasElseStack.empty()) {

            if (!ifHasElseStack.top()) {

                violations.push_back({ "Potential inferred latch in always block.", 0 });

            }

            ifHasElseStack.pop();

        }

    }

}
```

Figure 9: Infer Latch code snippet

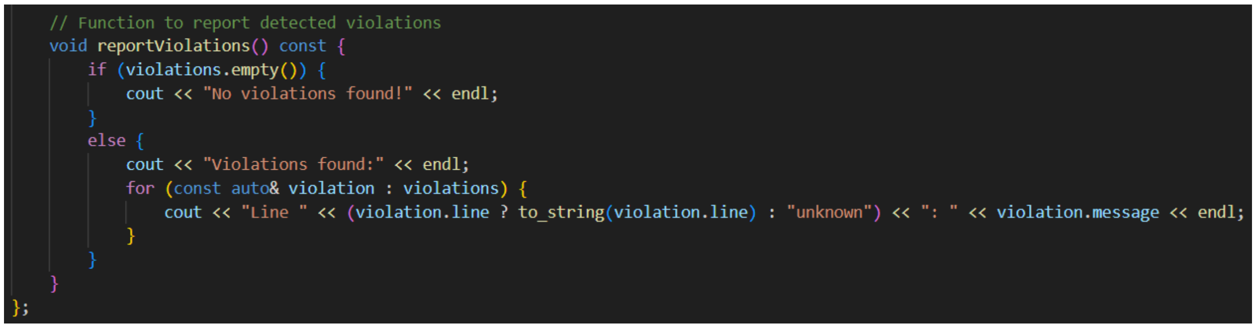
3. Report Generator

Purpose: Summarizes the results of the static analysis and provides a user-friendly report.
Features:

- Lists all detected violations, including:
  - Type of issue.
  - Line number where it occurs.
  - Affected signals or variables.
- Generates output in a text format for easy reference.



```
// Violation Struct
struct Violation {
    string message;
    int line;
};
```

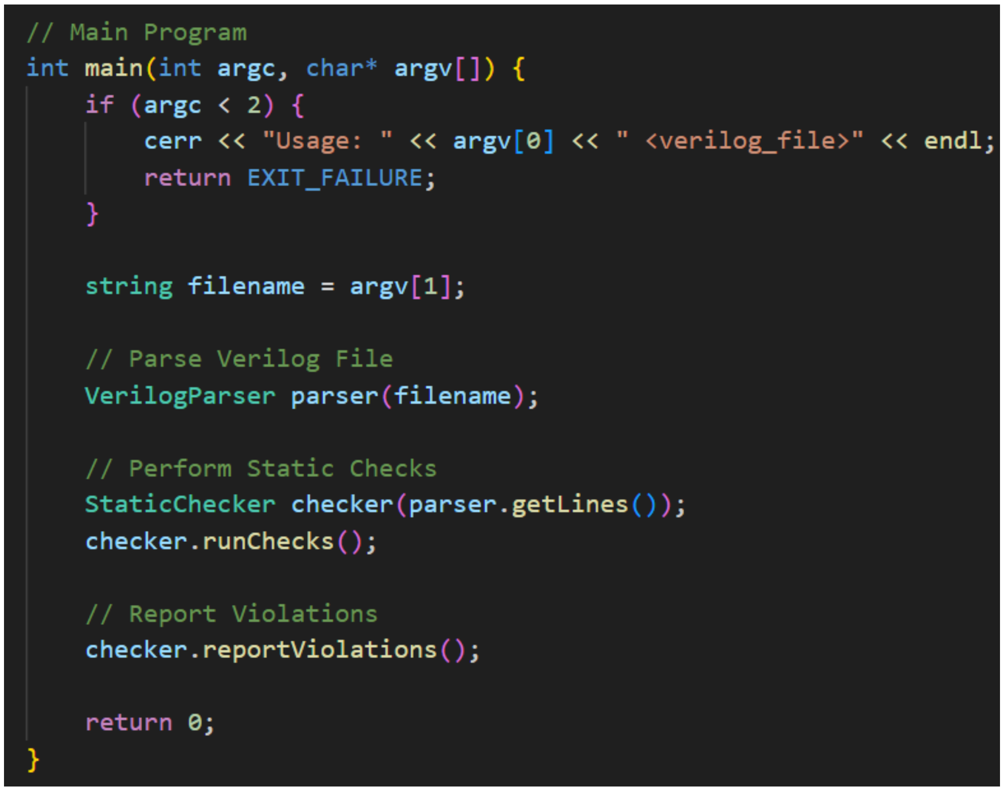


```
    // Function to report detected violations
    void reportViolations() const {
        if (violations.empty()) {
            cout << "No violations found!" << endl;
        }
        else {
            cout << "Violations found:" << endl;
            for (const auto& violation : violations) {
                cout << "Line " << (violation.line ? to_string(violation.line) : "unknown") << ": " << violation.message << endl;
            }
        }
    }
};
```

Figure 10: Report Violation function

Workflow

1. **Input**:
   - User provides the Verilog file to the system as input.
   - The file must contain a single flattened module representing the design under test (DUT).
2. **Processing**:
   - The VerilogParser reads and tokenizes the Verilog code.
   - The StaticChecker applies a series of checks to detect violations.
3. **Output**:
   - The reportViolations function produces a text-based report summarizing the detected issues.



```
// Main Program
int main(int argc, char* argv[]) {
    if (argc < 2) {
        cerr << "Usage: " << argv[0] << " <verilog_file>" << endl;
        return EXIT_FAILURE;
    }

    string filename = argv[1];

    // Parse Verilog File
    VerilogParser parser(filename);

    // Perform Static Checks
    StaticChecker checker(parser.getLines());
    checker.runChecks();

    // Report Violations
    checker.reportViolations();

    return 0;
}
```

Figure 11: Main Function code snippet

Design Principles

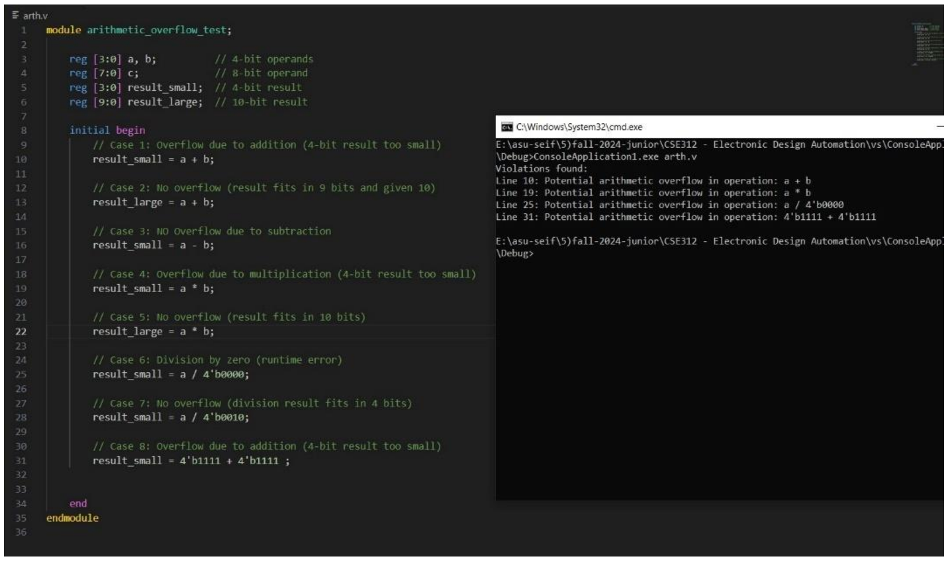
- **Modularity**: Each component is independent, enabling easy maintenance and future enhancements.
- **Scalability**: The tool is designed to accommodate additional checks and features in subsequent versions.
- **User-Focused Reporting**: Clear and actionable feedback ensures effective debugging and code improvement.
- **Performance**: Optimized algorithms ensure the tool processes large Verilog files efficiently.

Testing and Results

To ensure the correctness and reliability of the modules, we followed a systematic approach. Each module was tested individually first, focusing on its functionality in isolation. Once verified, we combined the modules into three test cases to simulate complex interactions and identify potential integration issues.

Individual Tests

1. **Arithmetic Flow Test:**
    - **Objective**: Verify arithmetic operations for overflow scenarios.
    - **Results**:

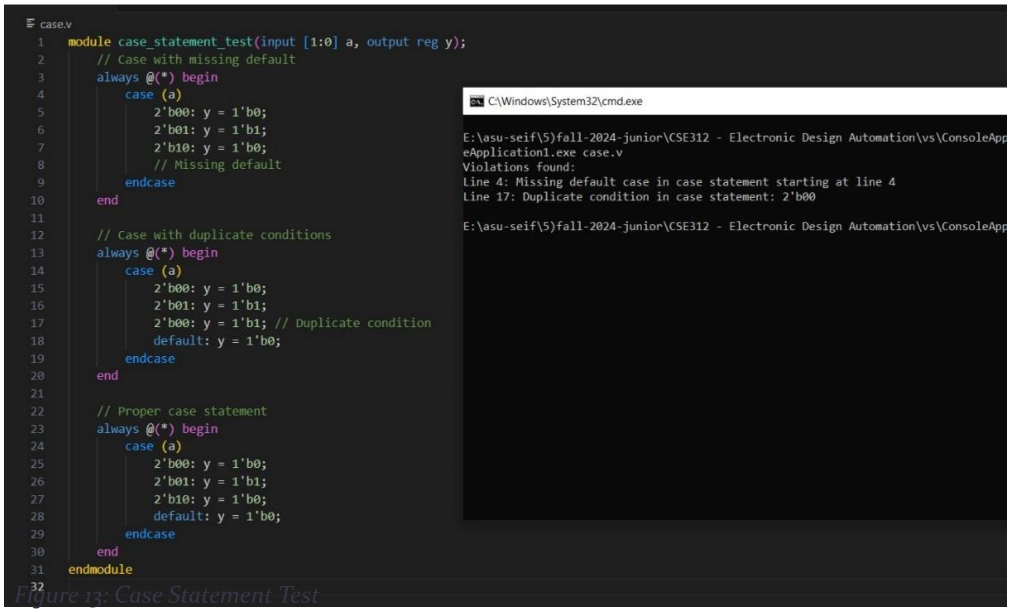


```
arth.v
1   module arithmetic_overflow_test;
2
3       reg [3:0] a, b;         // 4-bit operands
4       reg [7:0] c;            // 8-bit operand
5       reg [3:0] result_small; // 4-bit result
6       reg [9:0] result_large; // 10-bit result
7
8       initial begin
9           // Case 1: Overflow due to addition (4-bit result too small)
10          result_small = a + b;
11
12          // Case 2: No overflow (result fits in 9 bits and given 10)
13          result_large = a + b;
14
15          // Case 3: NO Overflow due to subtraction
16          result_small = a - b;
17
18          // Case 4: Overflow due to multiplication (4-bit result too small)
19          result_small = a * b;
20
21          // Case 5: No overflow (result fits in 10 bits)
22          result_large = a * b;
23
24          // Case 6: Division by zero (runtime error)
25          result_small = a / 4'b0000;
26
27          // Case 7: No overflow (division result fits in 4 bits)
28          result_small = a / 4'b0010;
29
30          // Case 8: Overflow due to addition (4-bit result too small)
31          result_small = 4'b1111 + 4'b1111 ;
32
33
34      end
35  endmodule
36
```

```
C:\Windows\System32\cmd.exe
E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleAppl
\Debug>ConsoleApplication1.exe arth.v
Violations found:
Line 10: Potential arithmetic overflow in operation: a + b
Line 19: Potential arithmetic overflow in operation: a * b
Line 25: Potential arithmetic overflow in operation: a / 4'b0000
Line 31: Potential arithmetic overflow in operation: 4'b1111 + 4'b1111

E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleAppl
\Debug>
```

Figure 12: Arithmetic Flow Test

2. **Case Statement Test:**
    - **Objective**: Test case statements for correctness.
    - **Results**:
        - Identified issues with missing default cases.
        - Duplicate conditions were flagged as problematic.
        - Properly structured case statements functioned correctly.

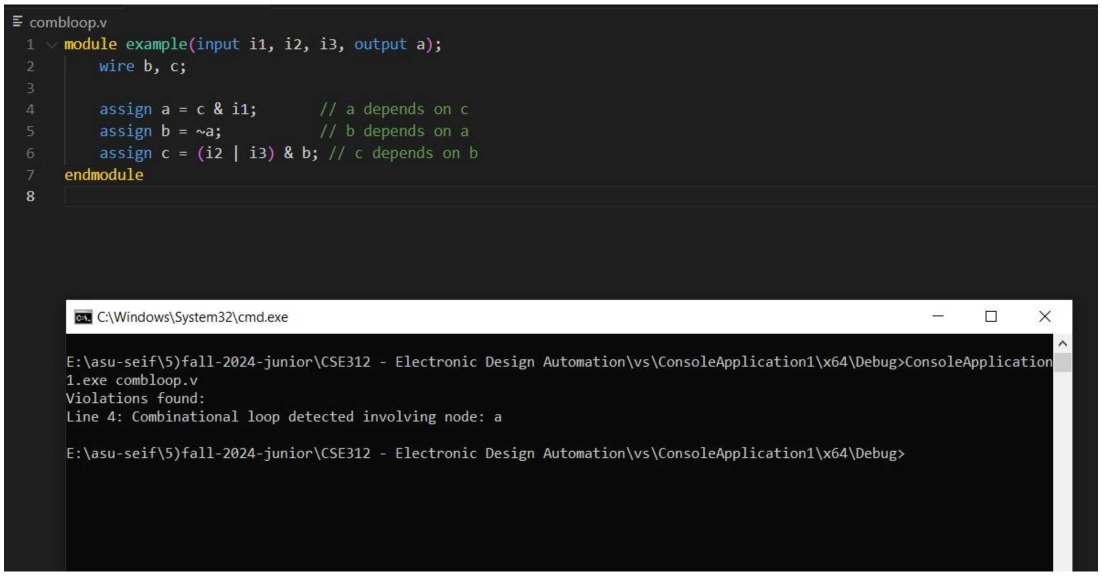


```
case.v
1   module case_statement_test(input [1:0] a, output reg y);
2       // Case with missing default
3       always @(*) begin
4           case (a)
5               2'b00: y = 1'b0;
6               2'b01: y = 1'b1;
7               2'b10: y = 1'b0;
8               // Missing default
9           endcase
10      end
11
12      // Case with duplicate conditions
13      always @(*) begin
14          case (a)
15              2'b00: y = 1'b0;
16              2'b01: y = 1'b1;
17              2'b00: y = 1'b1; // Duplicate condition
18              default: y = 1'b0;
19          endcase
20      end
21
22      // Proper case statement
23      always @(*) begin
24          case (a)
25              2'b00: y = 1'b0;
26              2'b01: y = 1'b1;
27              2'b10: y = 1'b0;
28              default: y = 1'b0;
29          endcase
30      end
31  endmodule
32
```

```
C:\Windows\System32\cmd.exe

E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApp
eApplication1.exe case.v
Violations found:
Line 4: Missing default case in case statement starting at line 4
Line 17: Duplicate condition in case statement: 2'b00

E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApp
```

Figure 13: Case Statement Test

3. **Combination Loop Test:**
   - **Objective**: Detect and resolve combinational loops in the module.
   - **Results**:
     - Demonstrated dependency resolution and propagation of values.
     - A combinational loop was detected involving *a*, *b*, and *c*:
       - a depends on c, b depends on a, and c depends on b, forming a feedback loop.

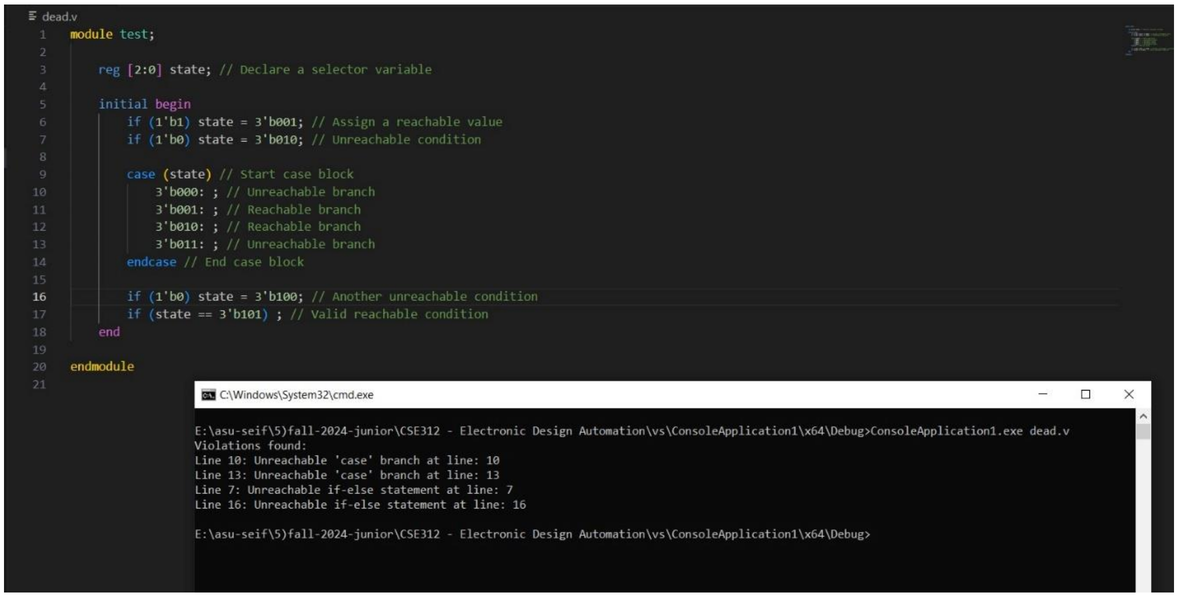


```
combloop.v
1  module example(input i1, i2, i3, output a);
2      wire b, c;
3
4      assign a = c & i1;       // a depends on c
5      assign b = ~a;           // b depends on a
6      assign c = (i2 | i3) & b; // c depends on b
7  endmodule
8
```

```
C:\Windows\System32\cmd.exe

E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>ConsoleApplication
1.exe combloop.v
Violations found:
Line 4: Combinational loop detected involving node: a

E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>
```

Figure 14: Combination Loop Test

4. **Dead Code Test:**
   - **Objective**: Verify state transitions and unreachable code.
   - **Results**:
     - Detected unreachable branches and conditions.
     - Verified reachable branches worked as expected.

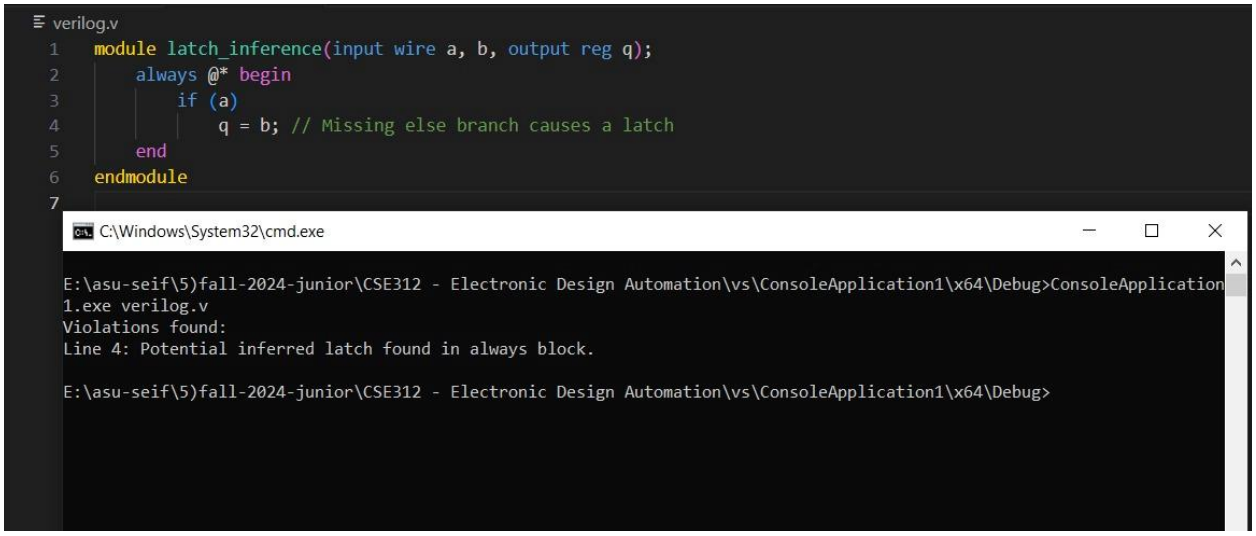


```
dead.v
1   module test;
2
3       reg [2:0] state; // Declare a selector variable
4
5       initial begin
6           if (1'b1) state = 3'b001; // Assign a reachable value
7           if (1'b0) state = 3'b010; // Unreachable condition
8
9           case (state) // Start case block
10              3'b000: ; // Unreachable branch
11              3'b001: ; // Reachable branch
12              3'b010: ; // Reachable branch
13              3'b011: ; // Unreachable branch
14          endcase // End case block
15
16          if (1'b0) state = 3'b100; // Another unreachable condition
17          if (state == 3'b101) ; // Valid reachable condition
18      end
19
20  endmodule
21
```

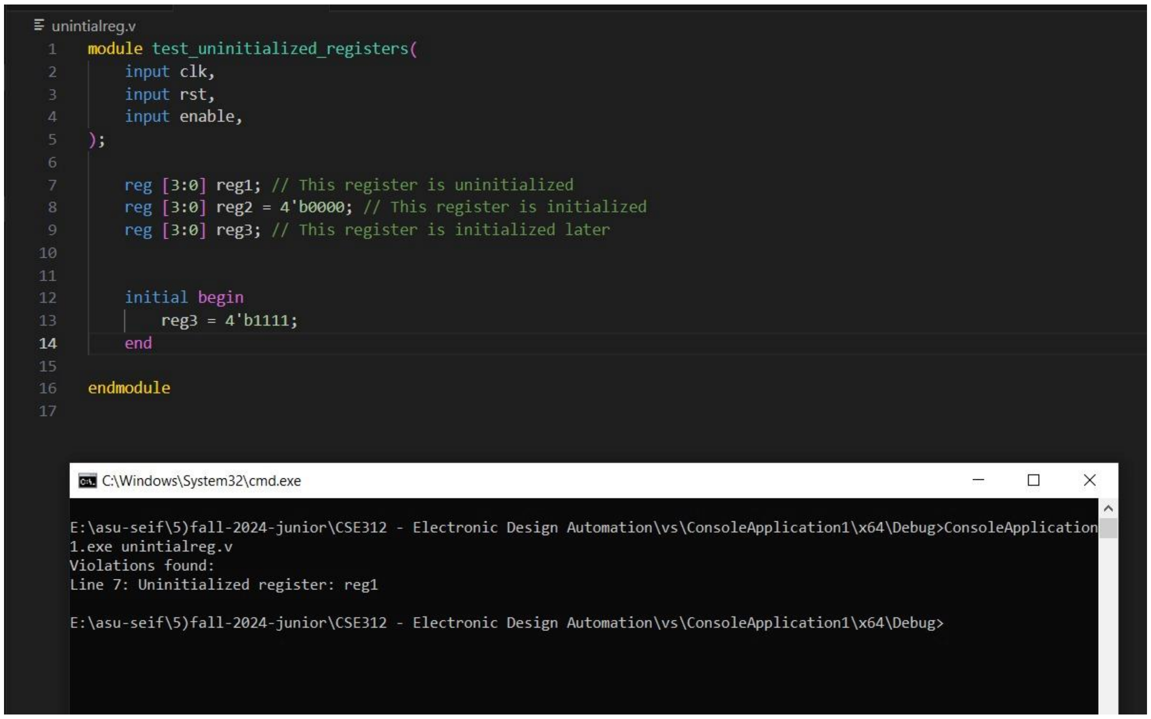
```
C:\Windows\System32\cmd.exe

E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>ConsoleApplication1.exe dead.v
Violations found:
Line 10: Unreachable 'case' branch at line: 10
Line 13: Unreachable 'case' branch at line: 13
Line 7: Unreachable if-else statement at line: 7
Line 16: Unreachable if-else statement at line: 16

E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>
```

Figure 15: Dead Code Test

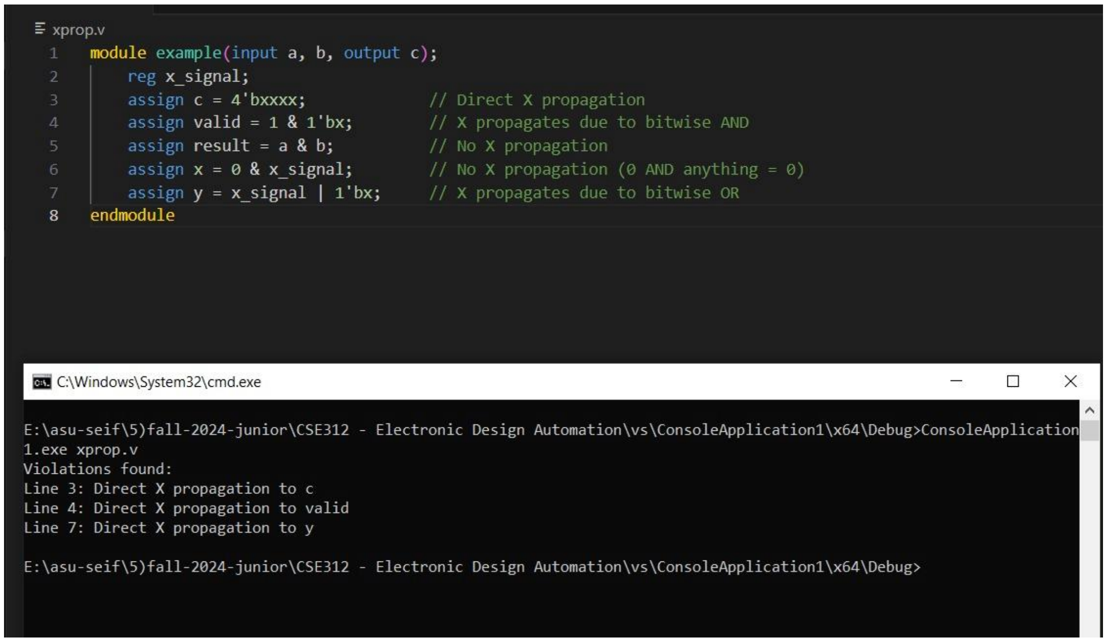
5. **Latch Inference Test:**
    - **Objective**: Identify latch inference due to incomplete logic.
    - **Results**:
        - Missing else branches caused unintended latch inference.


verilog.v
```
module latch_inference(input wire a, b, output reg q);
    always @* begin
        if (a)
            q = b; // Missing else branch causes a latch
    end
endmodule
```
C:\Windows\System32\cmd.exe
```
E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>ConsoleApplication
1.exe verilog.v
Violations found:
Line 4: Potential inferred latch found in always block.

E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>
```
Figure 16: Latch Inference Test

6. **Uninitialized Registers Test**
    - **Objective**: Test the behavior of initialized and uninitialized registers.
    - **Results**:
        - Uninitialized registers are flagged as potential hazards.
        - Registers explicitly initialized behaved as expected.


unintialreg.v
```
module test_uninitialized_registers(
    input clk,
    input rst,
    input enable,
);

    reg [3:0] reg1; // This register is uninitialized
    reg [3:0] reg2 = 4'b0000; // This register is initialized
    reg [3:0] reg3; // This register is initialized later


    initial begin
        reg3 = 4'b1111;
    end

endmodule
```
C:\Windows\System32\cmd.exe
```
E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>ConsoleApplication
1.exe unintialreg.v
Violations found:
Line 7: Uninitialized register: reg1

E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>
```
Figure 17: Uninitialized Registers Test

7. **FSM Test:**
   - **Objective**: Test FSM transitions and unreachable states.
   - **Results**:
     - S2 was unreachable due to missing transition logic.
     - State transitions between S0 and S1 verified.

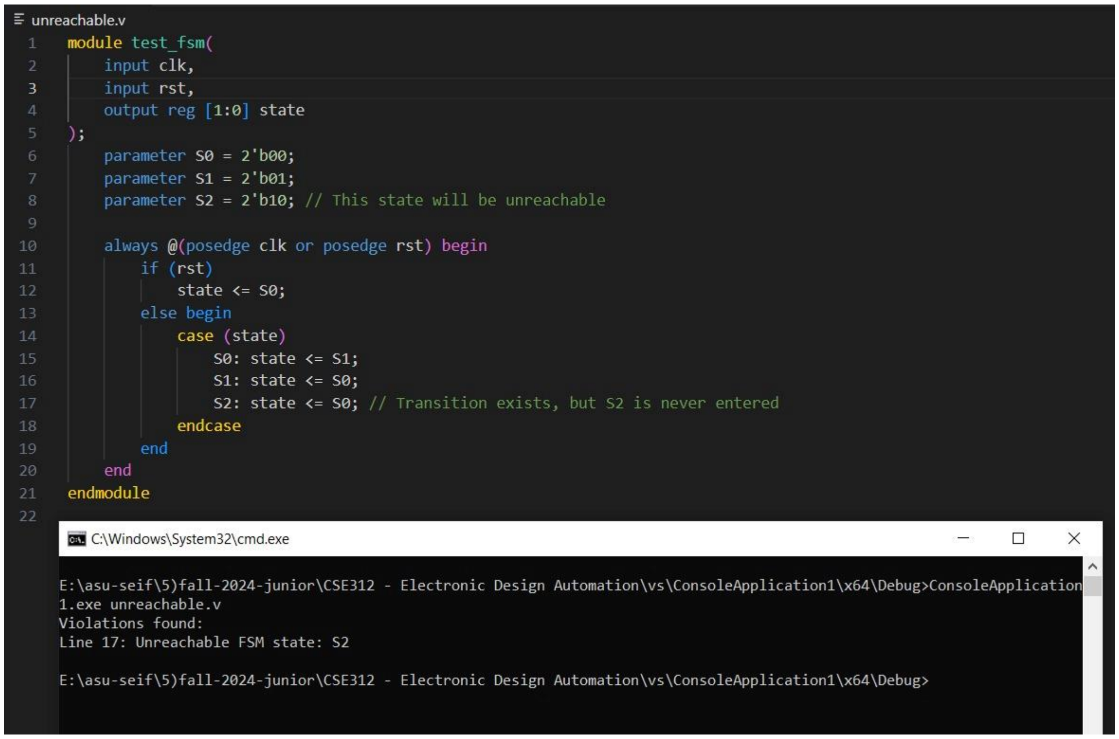


xprop.v
```verilog
module example(input a, b, output c);
    reg x_signal;
    assign c = 4'bxxxx;             // Direct X propagation
    assign valid = 1 & 1'bx;        // X propagates due to bitwise AND
    assign result = a & b;          // No X propagation
    assign x = 0 & x_signal;        // No X propagation (0 AND anything = 0)
    assign y = x_signal | 1'bx;     // X propagates due to bitwise OR
endmodule
```

C:\Windows\System32\cmd.exe
```
E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>ConsoleApplication
1.exe xprop.v
Violations found:
Line 3: Direct X propagation to c
Line 4: Direct X propagation to valid
Line 7: Direct X propagation to y

E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>
```

Figure 18: FSM Test

8. **X Propagation Test:**
   - **Objective**: Test X propagation behavior in logic operations.
   - **Results**:
     - Identified cases where X propagated through operations.
     - Proper masking handled X propagation effectively.

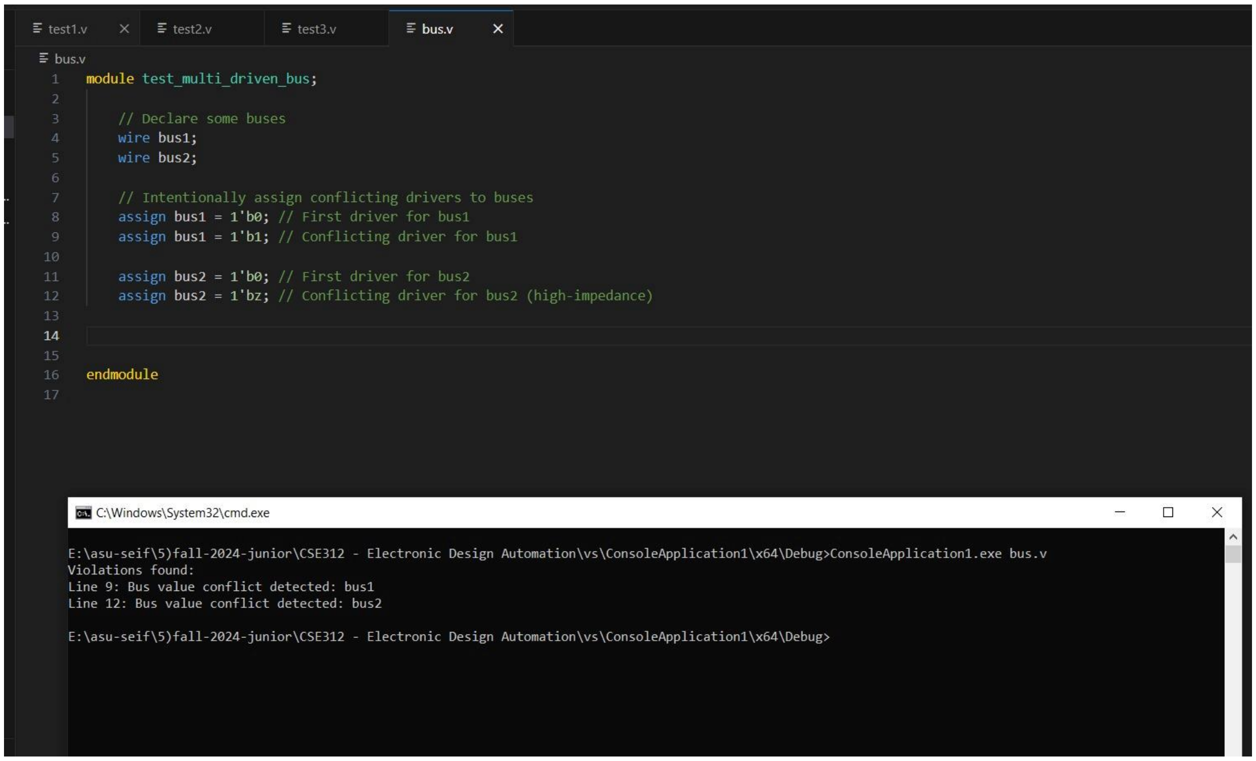


unreachable.v
```verilog
module test_fsm(
    input clk,
    input rst,
    output reg [1:0] state
);
    parameter S0 = 2'b00;
    parameter S1 = 2'b01;
    parameter S2 = 2'b10; // This state will be unreachable

    always @(posedge clk or posedge rst) begin
        if (rst)
            state <= S0;
        else begin
            case (state)
                S0: state <= S1;
                S1: state <= S0;
                S2: state <= S0; // Transition exists, but S2 is never entered
            endcase
        end
    end
endmodule
```

C:\Windows\System32\cmd.exe
```
E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>ConsoleApplication
1.exe unreachable.v
Violations found:
Line 17: Unreachable FSM state: S2

E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>
```

Figure 19: X Propagation Test

9. **Multi-Driven Bus Test:**
    - **Objective:** Test for conflicting drivers assigned to shared buses and identify potential bus contention issues.
    - **Results:**
        - Detected multiple conflicting drivers for bus1 and bus2.
        - Highlighted scenarios where bus contention occurs due to improper driver assignments.



```verilog
module test_multi_driven_bus;

    // Declare some buses
    wire bus1;
    wire bus2;

    // Intentionally assign conflicting drivers to buses
    assign bus1 = 1'b0; // First driver for bus1
    assign bus1 = 1'b1; // Conflicting driver for bus1

    assign bus2 = 1'b0; // First driver for bus2
    assign bus2 = 1'bz; // Conflicting driver for bus2 (high-impedance)


endmodule
```

```
E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>ConsoleApplication1.exe bus.v
Violations found:
Line 9: Bus value conflict detected: bus1
Line 12: Bus value conflict detected: bus2

E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>
```

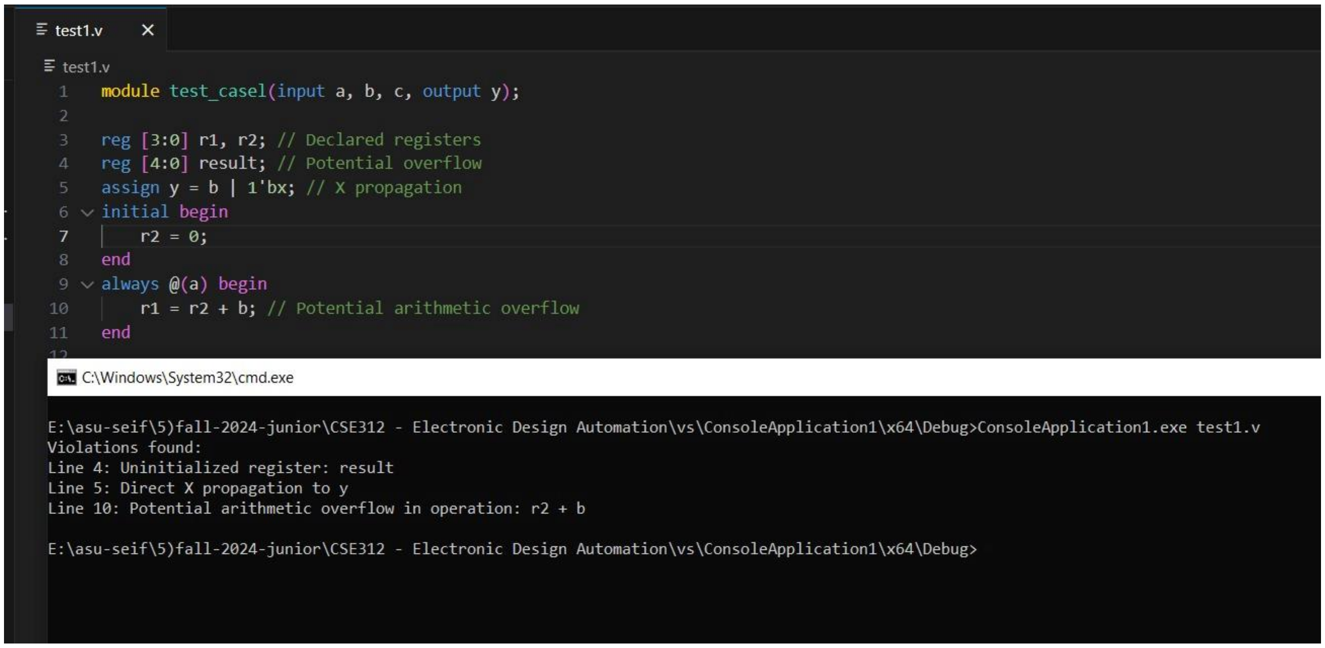
Figure 20: Multi-Driven Bus Test

Combined Test Cases

Test Case 1: Uninitialized Registers and Arithmetic Overflow

- **Purpose**: To detect uninitialized registers that can cause undefined behavior, direct X propagation in logic assignments and potential arithmetic overflow in operations.

- **Violations Detected**:
  - Uninitialized Register: The *result* register was declared but not initialized, flagged at line 4.
  - Direct X Propagation: The signal *y* propagates an undefined value (*1'bx*), flagged at line 5.
  - Arithmetic Overflow: The addition *r2* + *b* potentially exceeds the 4-bit width of *r1*, flagged at line 10.



```
1  module test_casel(input a, b, c, output y);
2
3  reg [3:0] r1, r2; // Declared registers
4  reg [4:0] result; // Potential overflow
5  assign y = b | 1'bx; // X propagation
6  initial begin
7      r2 = 0;
8  end
9  always @(a) begin
10     r1 = r2 + b; // Potential arithmetic overflow
11 end
```

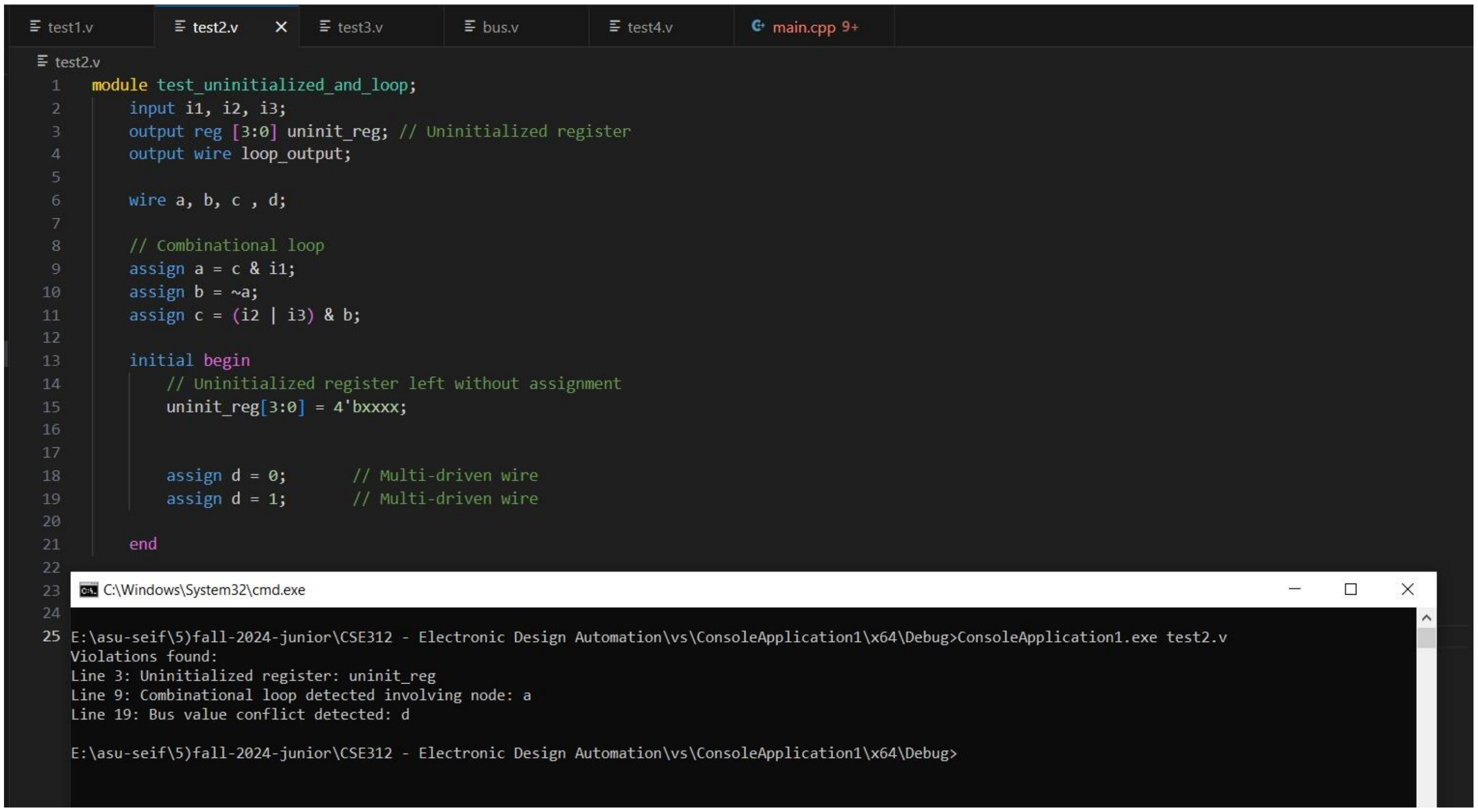
```
E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>ConsoleApplication1.exe test1.v
Violations found:
Line 4: Uninitialized register: result
Line 5: Direct X propagation to y
Line 10: Potential arithmetic overflow in operation: r2 + b

E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>
```

Figure 21: Test Case 1

Test Case 2: Multi-Driven Wire, Uninitialized Registers, and Combinational Loops

- **Purpose:** To verify the linter's ability to detect uninitialized registers that can lead to undefined behavior, identify combinational loops caused by circular dependencies in logic, and flag issues with multi-driven wires, where conflicting values are assigned.

- **Violations Detected:**
    - Uninitialized Register:
        - *uninit_reg* was declared but left uninitialized detected at line 3.
    - Combinational Loop:
        - Circular dependency involving node *a* was detected at line 9.
    - Multi-Driven Wire:
        - The wire *d* was driven by two conflicting values (*0* and *1*) at line 19.



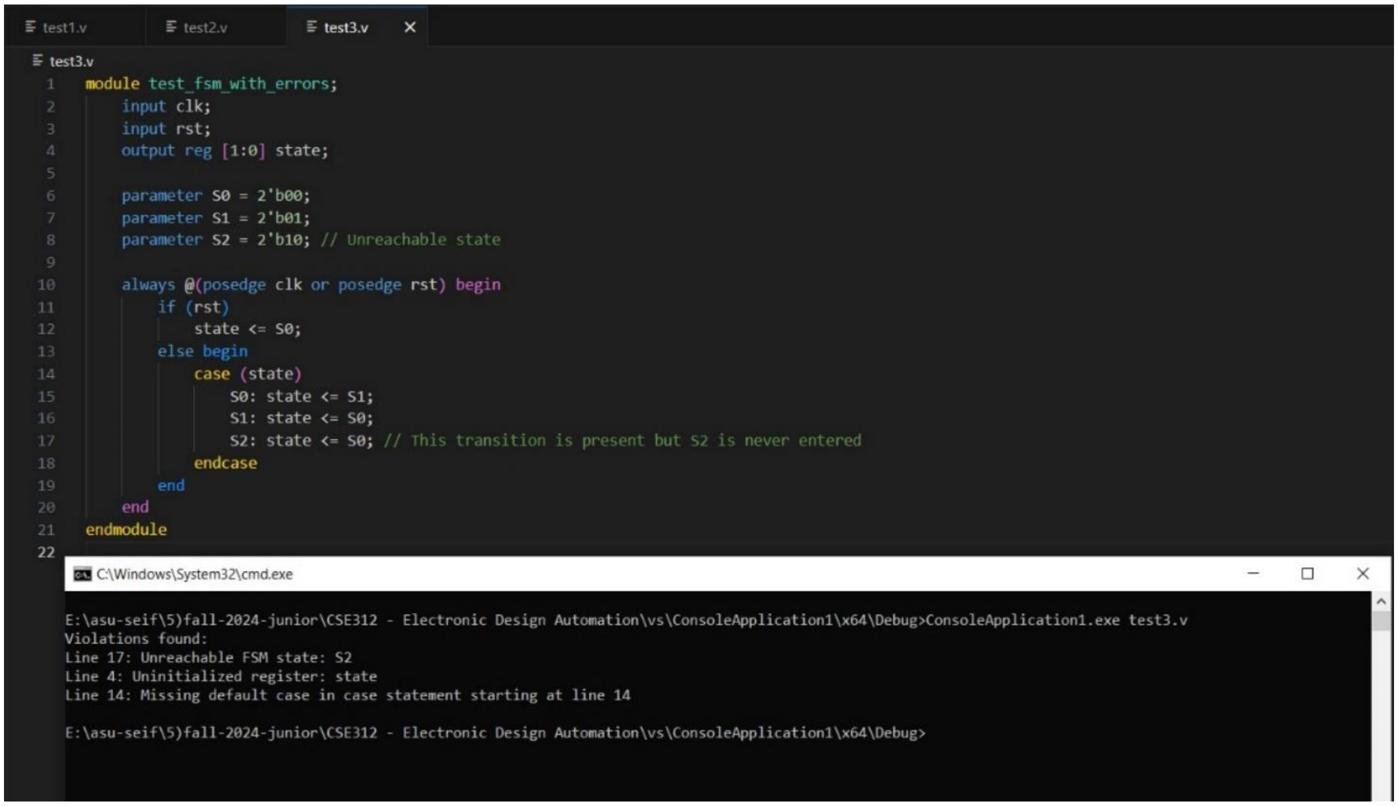
```
1   module test_uninitialized_and_loop;
2       input i1, i2, i3;
3       output reg [3:0] uninit_reg; // Uninitialized register
4       output wire loop_output;
5   
6       wire a, b, c , d;
7   
8       // Combinational loop
9       assign a = c & i1;
10      assign b = ~a;
11      assign c = (i2 | i3) & b;
12  
13      initial begin
14          // Uninitialized register left without assignment
15          uninit_reg[3:0] = 4'bxxxx;
16  
17  
18          assign d = 0;       // Multi-driven wire
19          assign d = 1;       // Multi-driven wire
20  
21      end
22  
23  
24  
25  
```

```
E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>ConsoleApplication1.exe test2.v
Violations found:
Line 3: Uninitialized register: uninit_reg
Line 9: Combinational loop detected involving node: a
Line 19: Bus value conflict detected: d

E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>
```

Figure 22: Test Case 2

Test Case 3: FSM Design with Missing Defaults, Unreachable States, and Initialization

- **Purpose:** To evaluate FSM design for completeness, including missing default cases and unreachable states, as well as proper initialization of state registers.
- **Violations Detected:**
  - Unreachable FSM State: State *S2* was defined but never entered due to missing transitions, flagged at line 17.
  - Uninitialized Register: The *state* register was declared but not initialized, flagged at line 4.
  - Missing Default Case: A *case* statement lacked a *default* branch, flagged at line 14.



```
 1  module test_fsm_with_errors;
 2      input clk;
 3      input rst;
 4      output reg [1:0] state;
 5
 6      parameter S0 = 2'b00;
 7      parameter S1 = 2'b01;
 8      parameter S2 = 2'b10; // Unreachable state
 9
10      always @(posedge clk or posedge rst) begin
11          if (rst)
12              state <= S0;
13          else begin
14              case (state)
15                  S0: state <= S1;
16                  S1: state <= S0;
17                  S2: state <= S0; // This transition is present but S2 is never entered
18              endcase
19          end
20      end
21  endmodule
22
```

```
E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>ConsoleApplication1.exe test3.v
Violations found:
Line 17: Unreachable FSM state: S2
Line 4: Uninitialized register: state
Line 14: Missing default case in case statement starting at line 14

E:\asu-seif\5)fall-2024-junior\CSE312 - Electronic Design Automation\vs\ConsoleApplication1\x64\Debug>
```

Figure 23: Test Case 3

Validation Against Commercial Linters

The tool's results for all three test cases were compared with a commercial Verilog linter. The results matched perfectly, confirming tool's accuracy.

Test Case	Violation detected by Lolinta	Violation detected by Commercial Linter
1	Uninitialized *result*, X propagation, Arithmetic overflow	Same
2	Uninitialized *uninit_reg,* Combinational loop	Same
3	Uninitialized *state*, Unreachable FSM state *S2*, Missing default case	Same

Summary

The updated Verilog Linter successfully detected all violations in the new test cases, including uninitialized registers, combinational loops, unreachable FSM states, and missing default cases. The results were consistent with those of a commercial Verilog linter, further validating the tool's accuracy and practicality for hardware design validation.

Conclusion

The Verilog Linter project successfully demonstrated how static analysis could be applied to Verilog HDL for detecting common design violations. The tool automates the process of code inspection, providing clear, actionable reports that reduce debugging efforts. With additional features and enhancements, the tool could be integrated into modern EDA toolchains to support larger hardware development projects.

References

[1] Lecture slides
[2] Tutorial slides
[3] Sigasi, "Linting Verilog in Visual Studio Code," Accessed: Dec. 19, 2024. [Online]. Available https://www.sigasi.com/manual/vscode/linting-verilog/